
Jam.py documentation Documentation

Andrew Yushev

Aug 02, 2023

Contents

1	Jam.py documentation	1
1.1	How the documentation is organized	1
1.2	Video Tutorials	1
2	Getting started	3
2.1	Installation	3
2.2	Creating a project	6
2.3	Demo project	9
2.4	Tutorial. Part 1. First project	11
2.5	Tutorial. Part 2. File and image fields	45
2.6	Tutorial. Part 3. Details	58
2.7	Jam.py deployment with Apache and mod_wsgi	65
3	Jam.py programming	67
3.1	Task tree	67
3.2	Workflow	69
3.3	Working with modules	70
3.4	Client side programming	70
3.5	Data programming	80
3.6	Server side programming	96
3.7	Programming reports	98
4	Jam.py FAQ	107
4.1	What is the difference between catalogs and journals	107
4.2	Howto upgrade an already created project to a new version of jampy?	107
4.3	What are foreign keys used for?	108
4.4	Can I use other libraries in my application	108
4.5	When printing a report I get an ods file instead of pdf	108
5	How to	109
5.1	How to install Jam.py on Windows	109
5.2	How to migrate development to production	111
5.3	How to migrate to another database	112
5.4	How to deploy	113
5.5	How do I write functions which have a global scope	121
5.6	How to validate field value	121
5.7	How to add a button to a form	123

5.8	How to execute script from client	123
5.9	How to change style and attributes of form elements	124
5.10	How to create a custom menu	126
5.11	How to append a record using an edit form without opening a view form?	126
5.12	How to prohibit changing record	127
5.13	How to link two tables	127
5.14	How change field value of selected records	131
5.15	How to save edit form without closing it	134
5.16	How to save changes to two tables in same transaction on the server	134
5.17	How to prevent duplicate values in a table field	135
5.18	How to implement some sort of basic multi-tenancy? For example, to have users with separate data.	136
5.19	Can I use Jam.py with existing database	137
5.20	How can I use data from other database tables	137
5.21	How I can process a request or get some data from other application or service	139
5.22	How can I perform calculations in the background	139
5.23	Is it supported to have details inside details?	140
5.24	Export to / import from csv files	142
5.25	Authentication	144
6	Business application builder	153
6.1	Sanitizing	153
6.2	Accept string	155
6.3	Project management	156
6.4	Roles	169
6.5	Users	170
6.6	Code editor	172
6.7	Task	174
6.8	Groups	175
6.9	Items	180
6.10	Details	201
6.11	Lookup List Dialog	202
6.12	Intergation with existing database	205
6.13	Saving audit trail/change history made by users	205
6.14	Record locking	209
6.15	Language support	212
6.16	Language translation	215
7	Jam.py class reference	217
7.1	Client side (javascript) class reference	217
7.2	Server side (python) class reference	362
8	Release notes	427
8.1	Version 1	427
8.2	Version 2	427
8.3	Version 3	427
8.4	Version 4	427
8.5	Version 5	430
8.6	Jam.py roadmap	442
	Index	443

1.1 How the documentation is organized

Here is an overview of how the documentation is organized, that will help you know where to look for certain things:

Getting started topics describe how to install the framework, create a new project, take you through a series of steps to develop a Web application and explains how to deploy it.

Programming guides discuss key topics and concepts at a fairly high level and provide useful background information and explanation.

Business application builder is a detailed description of the Application Builder used for application development and database administration.

Class reference guides contain technical reference for Jam.py classes APIs

FAQ topics covers most frequently asked questions.

How to contains code examples that can be useful to quickly accomplish common tasks

Or you can go to the *table of contents*

1.2 Video Tutorials

If you are new to Jam.py, we highly recommend that you watch these video tutorials

It is recommended to watch these videos with a resolution of 1080p

Tutorial 1 - [Working with files and images](#)

Tutorial 2 - [Working with details](#)

Tutorial 3 - [Users, roles, audit trail/change history](#)

Tutorial 4 - [Task tree](#)

Tutorial 5 - [Forms](#)

Tutorial 6 - [Form events](#)

Tutorial 7 - [Data aware controls](#)

Tutorial 8 - [Datasets](#)

Tutorial 9 - [Datasets Part 2](#)

Tutorial 10 - [Fields and filters](#)

Tutorial 11 - [Client-server interactions](#)

Tutorial 12 - [Working with data on the server](#)

Here you can learn how to install the framework, create a new project, develop a web application and deploy it.

2.1 Installation

2.1.1 Install python

Jam.py requires python. If it is not installed you can get the latest version of Python at <https://www.python.org/download/>

You can use the following versions of Python with Jam.py:

Python 2

- Python 2.7 and newer

Python 3

- Python 3.4 and newer

You can verify that Python is installed by typing python from your shell; you should see something like:

```
Python 2.7.12 (default, Nov 19 2016, 06:48:10)
[GCC 5.4.0 20160609] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

If Python 2 and Python 3 are installed try to type python3:

```
Python 3.5.2 (default, Nov 17 2016, 17:05:23)
[GCC 5.4.0 20160609] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

2.1.2 Install Jam.py

Installing an official release with *pip*

This is the recommended way to install Jam.py.

1. Install [pip](#). The easiest is to use the [standalone pip installer](#). If your distribution already has `pip` installed, you might need to update it if it's outdated. (If it's outdated, you'll know because installation won't work.)
2. If you're using Linux, Mac OS X or some other flavor of Unix, enter the command

```
sudo pip install jam.py
```

at the shell prompt.

If you're using Windows, start a command shell with administrator privileges and run the command

```
pip install jam.py
```

This will install Jam.py in your Python installation's `site-packages` directory.

Installing an official release manually

1. Download the package archive.
2. Create a new directory and unpack the archive there.
3. Go into the directory and run the setup command from command line

```
$ python setup.py install
```

This will install Jam.py in your Python installation's `site-packages` directory.

Note: on some unix like systems you may need to switch to root or run: `sudo python setup.py install`

2.1.3 Setting up a virtual environment

It is best practice to provide a dedicated environment for each Jam.py project you create. There are many options to manage environments and packages within the Python ecosystem, some of which are recommended in the [Python documentation](#).

To create a virtual environment for your project, open a new command prompt, navigate to the folder where you want to create your project and then enter the following:

```
...\> py -m venv project-name
```

This will create a folder called 'project-name' if it does not already exist and set up the virtual environment. To activate the environment, run:

```
...\> project-name\Scripts\activate.bat
```

The virtual environment will be activated and you'll see "(project-name)" next to the command prompt to designate that. Each time you start a new command prompt, you'll need to activate the environment again.

2.1.4 Install Jam.py

Jam.py can be installed easily using `pip` within your virtual environment.

In the command prompt, ensure your virtual environment is active, and execute the following command:

```
...\> py -m pip install jam.py
```

This will download and install the latest Jam.py release.

After the installation has completed, you can verify your Jam.py installation by executing `pip list` in the command prompt.

2.1.5 Common pitfalls

- If you are connecting to the internet behind a proxy, there might be problems in running the command `py -m pip install Jam.py`. Set the environment variables for proxy configuration in the command prompt as follows:

```
...\> set http_proxy=http://username:password@proxyserver:proxyport
...\> set https_proxy=https://username:password@proxyserver:proxyport
```

- If your Administrator prohibited setting up a virtual environment, it is still possible to install Jam.py as follows:

```
...\> python -m pip install jam.py
```

This will download and install the latest Jam.py release.

After the installation has completed, you can verify your Jam.py installation by executing `pip list` in the command prompt.

However, running `jam-project.py` will fail since it is not in the path. Check the installation folder:

```
...\> python -m site --user-site
```

The output might be similar to below:

```
C:\Users\youruser\AppData\Local\Packages\PythonSoftwareFoundation.Python.3.9_
↪qbz5n2kfra8p0\LocalCache\local-packages\Python39\site-packages
```

Replace `site-packages` at the end of above line with `Scripts`:

```
...\> dir C:\Users\youruser\AppData\Local\Packages\PythonSoftwareFoundation.
↪Python.3.9_qbz5n2kfra8p0\LocalCache\local-packages\Python39\Scripts
```

The output might be similar to below:

```
...\> Directory of
↪C:\Users\youruser\AppData\Local\Packages\PythonSoftwareFoundation.Python.3.9_
↪qbz5n2kfra8p0\LocalCache\local-packages\Python39\Scripts

13/04/2023  02:59 PM    <DIR>          .
13/04/2023  02:59 PM    <DIR>          ..
13/04/2023  02:59 PM                1,087 jam-project.py
                1 File(s)                1,087 bytes
                2 Dir(s)  177,027,321,856 bytes free
```

Create the new folder somewhere and run `jam-project` from from it:

```
...> python C:\Users\youruser\AppData\Local\Packages\PythonSoftwareFoundation.  
Python.3.9_qbz5n2kfra8p0\LocalCache\local-packages\Python39\Scripts\jam-project.  
py
```

Run the new project:

```
...> python server.py
```

2.2 Creating a project

Create a new directory.

Go into the directory and run from command line:

```
$ jam-project.py
```

The following files and folders will be created in the directory:

```
/
css/
js/
img/
reports/
static/
admin.sqlite
server.py
index.html
wsgi.py
```

To start Jam.py web server run **server.py** script.

```
$ ./server.py
```

Note: You can specify a port as parameter, for example

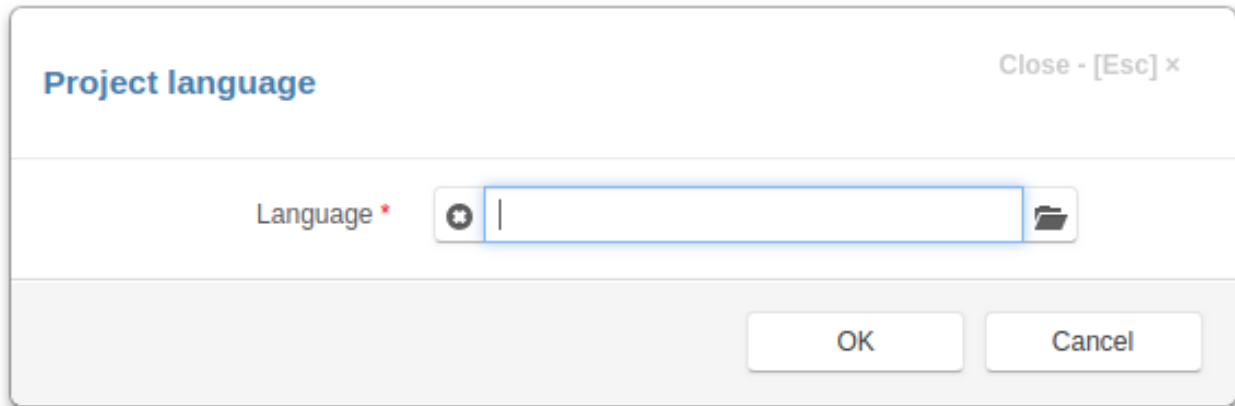
```
$ ./server.py 8081.
```

By default, the port is 8080. If you will specify another port, you need to use it in a browser in the next steps.

Open a Web browser and go to “/builder.html” on your local domain – e.g., <http://127.0.0.1:8080/builder.html>. You should see the select language dialog.

```
127.0.0.1:8080/builder.html
```

In the dialog that will appear, select the language and press OK button.



The screenshot shows a dialog box titled "Project language". In the top right corner of the dialog is the text "Close - [Esc] x". Below the title bar, there is a label "Language *" followed by a dropdown menu. The dropdown menu has a gear icon on the left and a folder icon on the right. At the bottom of the dialog, there are two buttons: "OK" and "Cancel".

In the New project dialog box fill in:


- **Caption** - the project name that appears to users
- **Name** - name of project (task) that will be used in programming code to get access to the task object. Should be a short valid python identifier. This name is used as a prefix when creating a table in the project database.
- **DB type** — select database type. If database is not Sqlite, it must be created in advance and its attributes should be entered in the corresponding form fields. To see examples of Database setup, follow the [link](#).

ParametersClose - [Esc] x


Caption *

Name *

DB type *



Sqlite



Server

Database *

Login

Password

Host

Port

Charset

OK

Cancel

When you press OK, the connection to the database will be checked, and in case of failure an error message will be displayed.

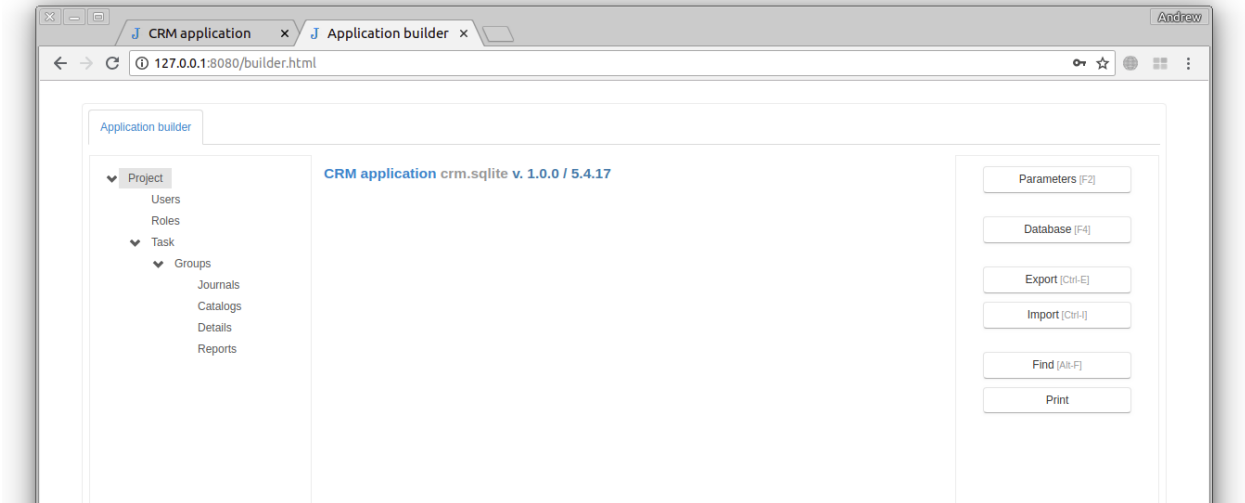
Note: Please note the following requirements:

- to use **FireBird** database, the python `fdb` library must be installed
 - **PostgreSQL** requires `psycopg2` library
 - **MySQL** requires `MySQLdb` library
 - **Oracle** requires `cx_Oracle` library
 - **MSSQL** requires `pymssql` library
 - to generate reports **LibreOffice** must be installed
-

Note: For **SQLite** database, when an item field is deleted or renamed, or foreign key is created, *Application builder*, creates a new table and copies records from the old one into it.

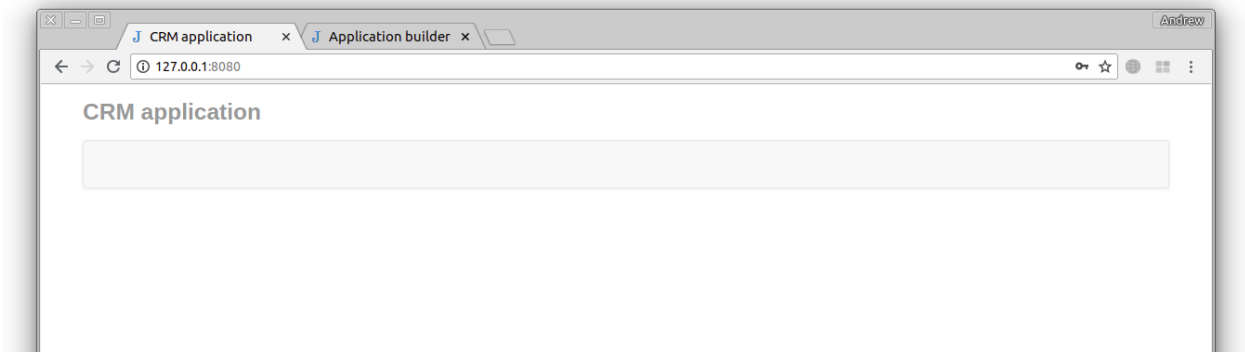
For **SQLite** database, Jam.py doesn't support importing of metadata into an existing project (project with tables in the database). You can only import metadata into a new project.

If all goes well, a new project will be created and the project tree will appear in the Application builder.



Now, to see the project itself, create a new page in the browser and type in the address bar:

```
127.0.0.1:8080
```



2.3 Demo project

The framework has a full fledged demo application that demonstrates programming techniques used in the framework.

The demo is located in the demo folder of the Jam.py package or you can download it by clicking on the [link](#).

To start the demo application go to the demo folder and run *server.py* script.

```
$ ./server.py
```

Open a Web browser and enter 127.0.0.1:8080 in the address bar.

To see *Application builder* open a new page in a browser and enter 127.0.0.1:8080/builder.html

The screenshot displays the Jam.py demo application in a web browser. The interface includes a navigation bar with links for Invoices, Catalogs, Reports, and Dashboard. The main content area is titled 'Invoices' and features a table with columns for Invoice Date, Customer, Billing Address, Billing City, Billing State, Billing Country, SubTotal, Tax, and Total. Below this table, there is a detailed view of a specific invoice item, showing columns for Album, Artist, Track, Quantity, UnitPrice, Amount, Tax, and Total. The application is running on a local server at 127.0.0.1:8080.

Invoices Table:

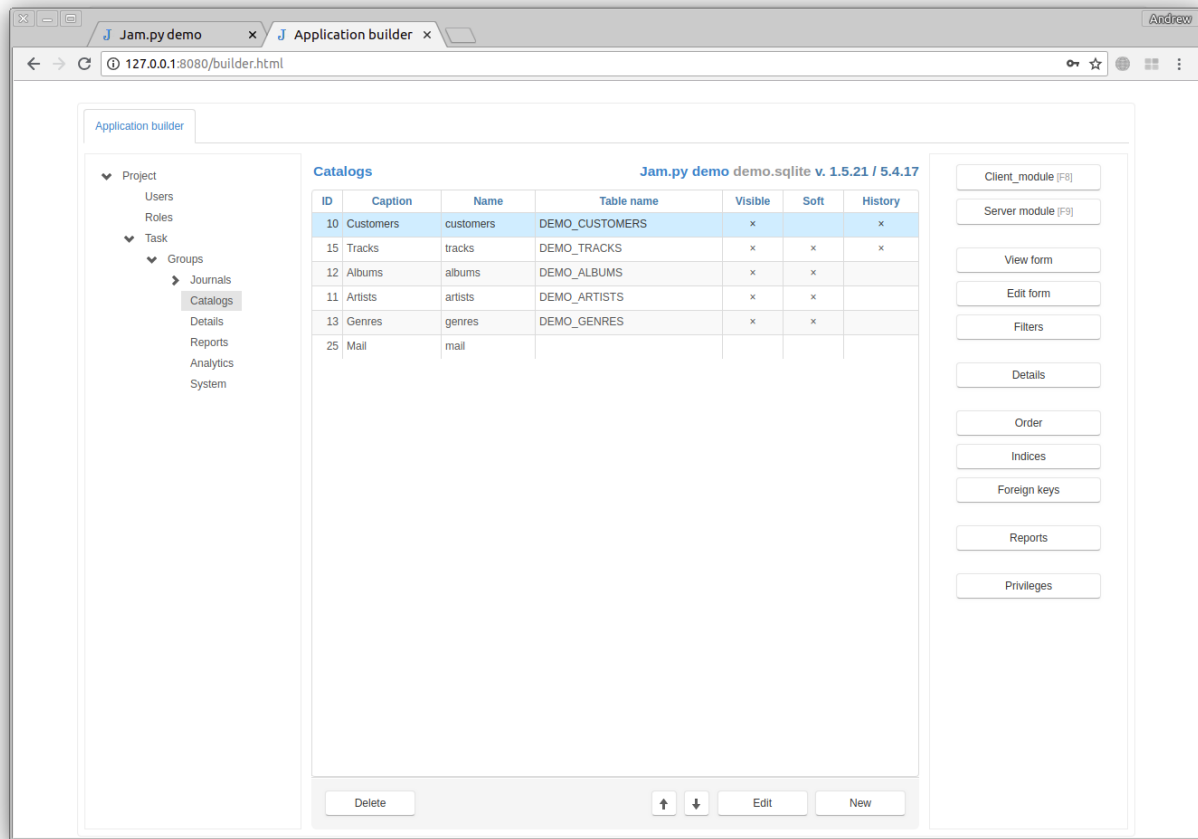
Invoice Date↓	Customer	Billing Address	Billing City	Billing State	Billing Country	SubTotal	Tax	Total
02/12/2019	Kara Nielsen	Sønder Boulevard 51	Copenhagen		Denmark	\$6.93	\$0.35	\$7.28
01/31/2019	Terhi Hämäläinen	Porthankatu 9	Helsinki		Finland	\$15.84	\$0.80	\$16.64
12/20/2018	Robert Brown	796 Dundas Street West	Toronto	ON	Canada	\$5.94	\$0.30	\$6.24
12/09/2018	Kara Nielsen	Sønder Boulevard 51	Copenhagen		Denmark	\$8.91	\$0.45	\$9.36
12/05/2018	Victor Stevens	319 N. Frances Street	Madison	WI	USA	\$5.94	\$0.30	\$6.24
12/04/2018	Kathy Chase	801 W 4th Street	Reno	NV	USA	\$1.98	\$0.10	\$2.08
12/04/2018	John Gordon	69 Salem Street	Boston	MA	USA	\$1.98	\$0.10	\$2.08
424						\$2357.30	\$119.00	\$2476.30

Page 1 of 61

Invoice Item Details:

Album↑	Artist	Track↑	Quantity	UnitPrice	Amount	Tax	Total
Bach: Goldberg Variations	Wilhelm Kempff	Aria Mit 30 Veränderungen, BWV	1	\$0.99	\$0.99	\$0.05	\$1.04
Bach: Orchestral Suites Nos. 1 - 4	Academy of St. Martin in the	Suite No. 3 in D, BWV 1068: III.	1	\$0.99	\$0.99	\$0.05	\$1.04
Bach: The Brandenburg	Orchestra of The Age of	Concerto No.2 in F Major,	1	\$0.99	\$0.99	\$0.05	\$1.04
Bach: The Cello Suites	Yo-Yo Ma	Suite for Solo Cello No. 1 in G	1	\$0.99	\$0.99	\$0.05	\$1.04
Bach: Toccata & Fugue in D Minor	Ton Koopman	Toccata and Fugue in D Minor,	1	\$0.99	\$0.99	\$0.05	\$1.04
		7			\$6.93	\$0.35	\$7.28

Buttons: Delete [Ctrl+Del], Reports, Edit, New [Ctrl+Ins]



2.4 Tutorial. Part 1. First project

Now, we'll walk you through the creation of a basic CRM application. Please follow the steps below:

2.4.1 New project

We'll assume that jam.py is already installed. If not, see [Installation](#) guide how to do it.

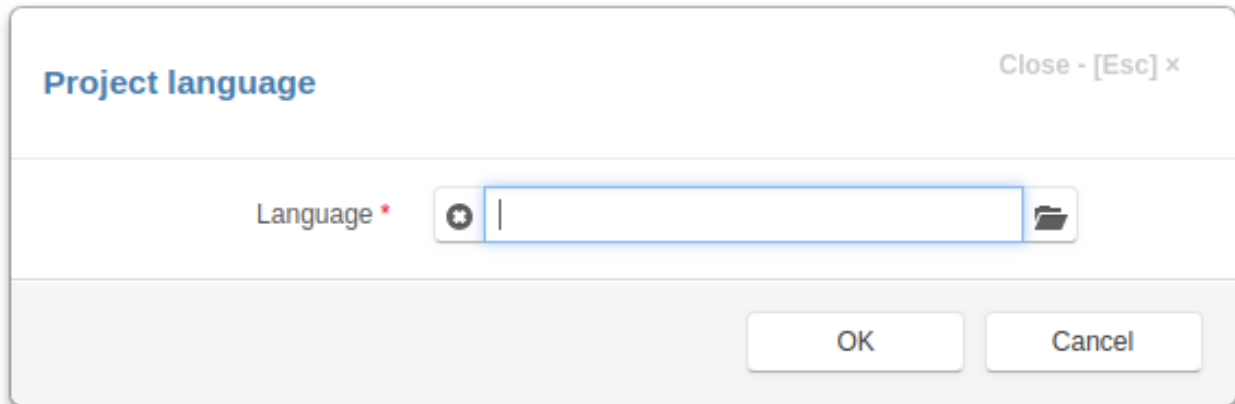
First we create a folder for the new project and in this folder we execute the *jam-project.py* script to create the project structures.

```
$ jam-project.py
```

After that we run *server.py* script that *jam-project.py* have created:

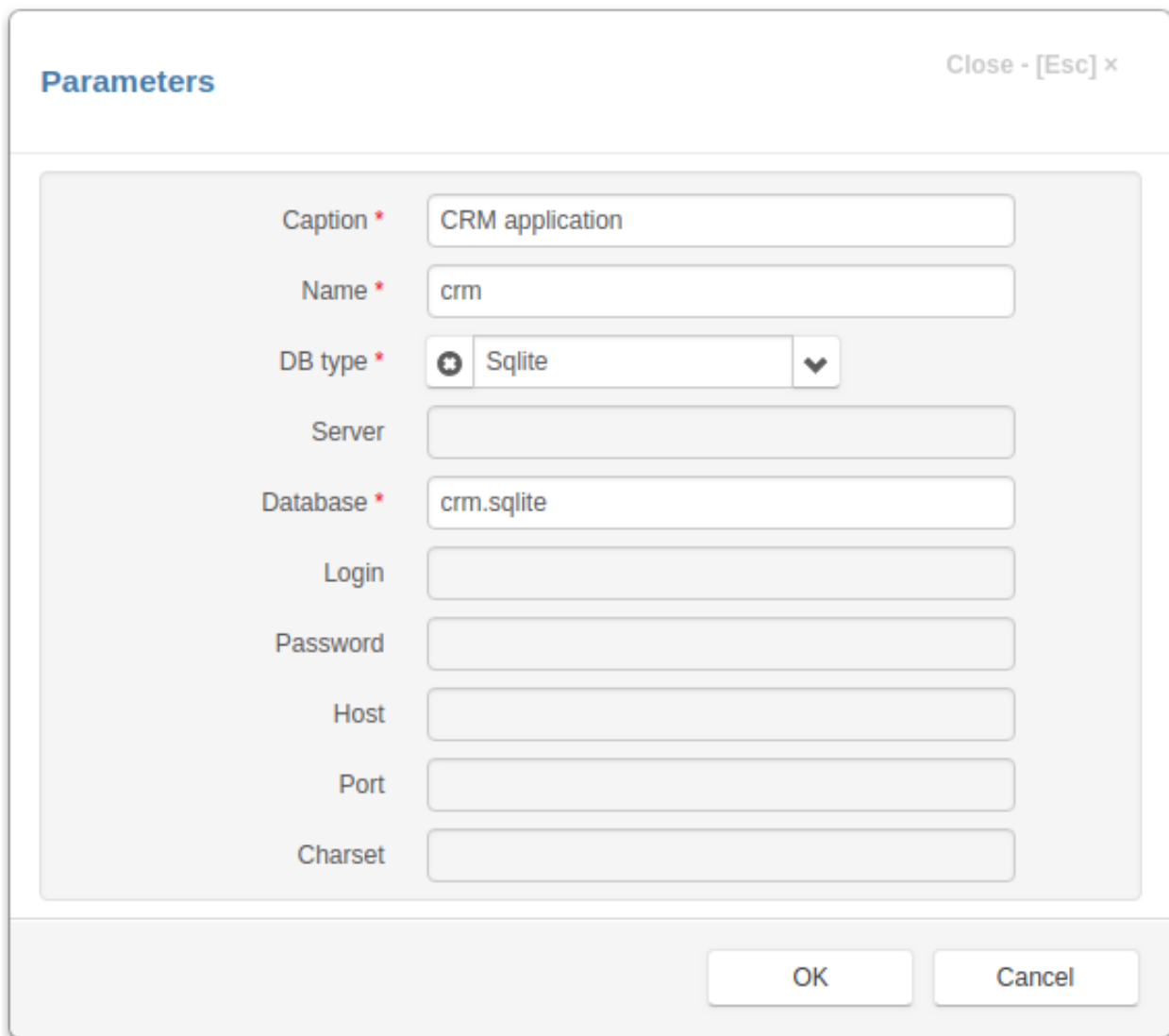
```
$ ./server.py
```

Now, to complete the creation of the project, open the web browser and go to 127.0.0.1:8080/builder.html to open the Application Builder. You should see the language selection dialog.



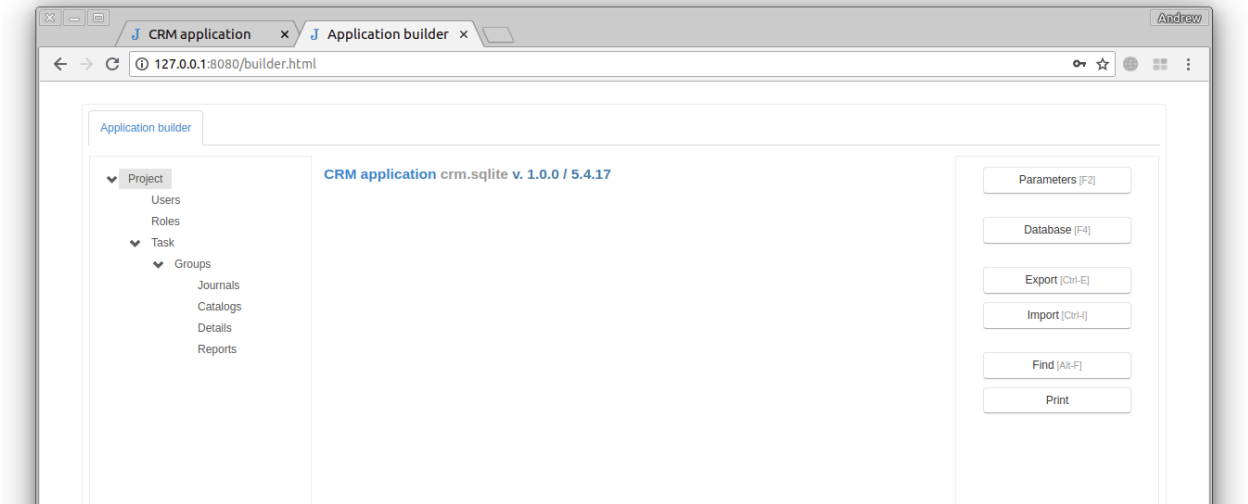
The 'Project language' dialog box has a title bar with 'Project language' on the left and 'Close - [Esc] x' on the right. The main area contains a label 'Language *' followed by a text input field with a gear icon on the left and a folder icon on the right. The input field is empty. At the bottom right are 'OK' and 'Cancel' buttons.

Select **English** and click **OK** button. The project parameters dialog box appears.

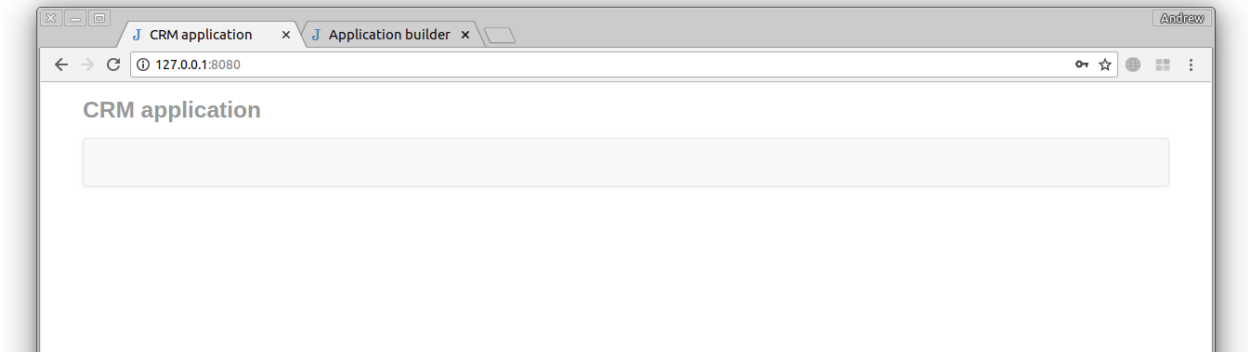


The 'Parameters' dialog box has a title bar with 'Parameters' on the left and 'Close - [Esc] x' on the right. The main area contains a form with the following fields: 'Caption *' with value 'CRM application', 'Name *' with value 'crm', 'DB type *' with a gear icon, 'Sqlite' text, and a dropdown arrow, 'Server' (empty), 'Database *' with value 'crm.sqlite', 'Login' (empty), 'Password' (empty), 'Host' (empty), 'Port' (empty), and 'Charset' (empty). At the bottom right are 'OK' and 'Cancel' buttons.

Fill out the form and click "OK". Now you should see the project tree in the left panel.



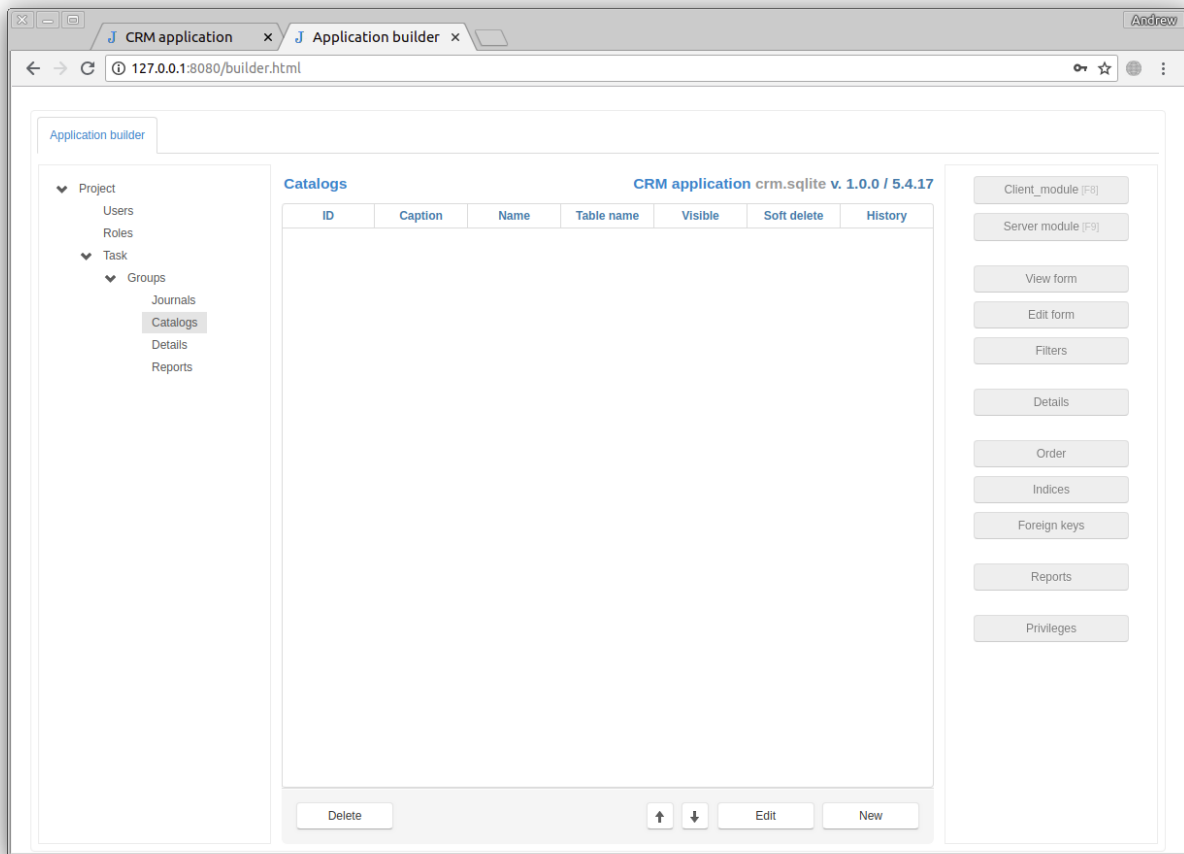
Open a new page, type 127.0.0.1:8080 in the address bar and press Enter. A new project appears with an empty menu.



2.4.2 New catalog

Let's go back to the Application builder page and create a "Customers" catalog.

Now we select the "Catalogs" group in the project tree and click the **New** button at the bottom right corner of the page



In the *Item Editor* dialog, fill in the caption and name of the new catalog

Item Editor

Close - [Esc] x

Caption *
Customers

Name *
customers

Table name
CRM_CUSTOMERS

Primary key field
+ id

Deleted flag
+ deleted

Visible ☒

Soft delete ☒

Virtual table ☐

History ☐

External js module ☐

Caption	Name ↑	Type	Size	Required	Read only	Lookup item	Lookup field	Master field	Typeahead	Lookup values
---------	--------	------	------	----------	-----------	-------------	--------------	--------------	-----------	---------------

Delete [Ctrl+Del]

Edit

New [Ctrl+Ins]

OK [Ctrl+Enter]

Cancel [Esc]

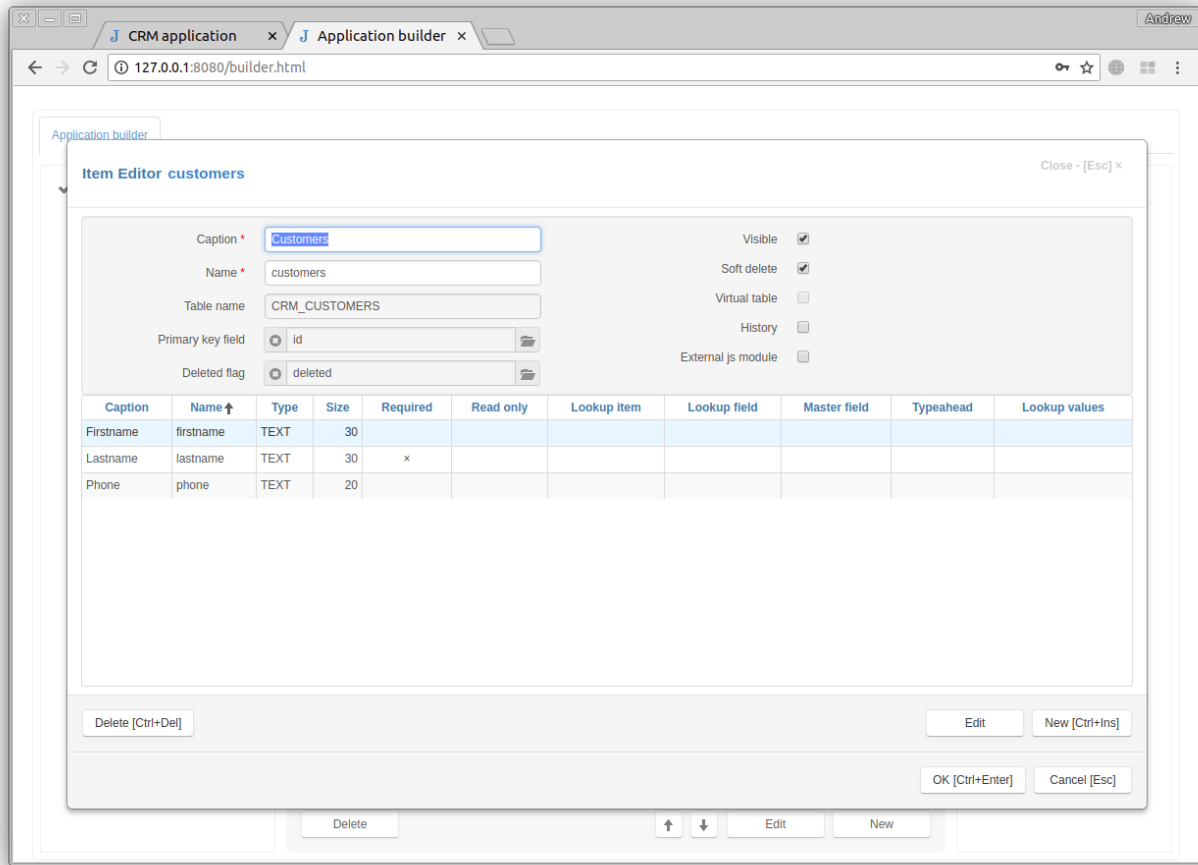
and click the **New** button in the bottom right corner of the dialog to add a new field. The *Field Editor* dialog appears. Type the caption and name of the “Firstname” field, select its type and click **OK** button.

The screenshot shows the 'Field Editor' dialog box. It has a title bar with 'Field Editor' and a close button 'Close - [Esc] x'. Below the title bar are three tabs: 'Field', 'Lookup', and 'Interface'. The 'Field' tab is active. Inside the 'Field' tab, there are several input fields and checkboxes:

- Caption ***: A text box containing 'Firstname'.
- Name ***: A text box containing 'firstname'.
- Type ***: A dropdown menu with a gear icon, 'TEXT', and a downward arrow.
- Size**: A text box containing '30'.
- Default value**: An empty text box.
- Required**: An unchecked checkbox.
- Read only**: An unchecked checkbox.

At the bottom right of the dialog are two buttons: 'OK [Ctrl+Enter]' and 'Cancel'.

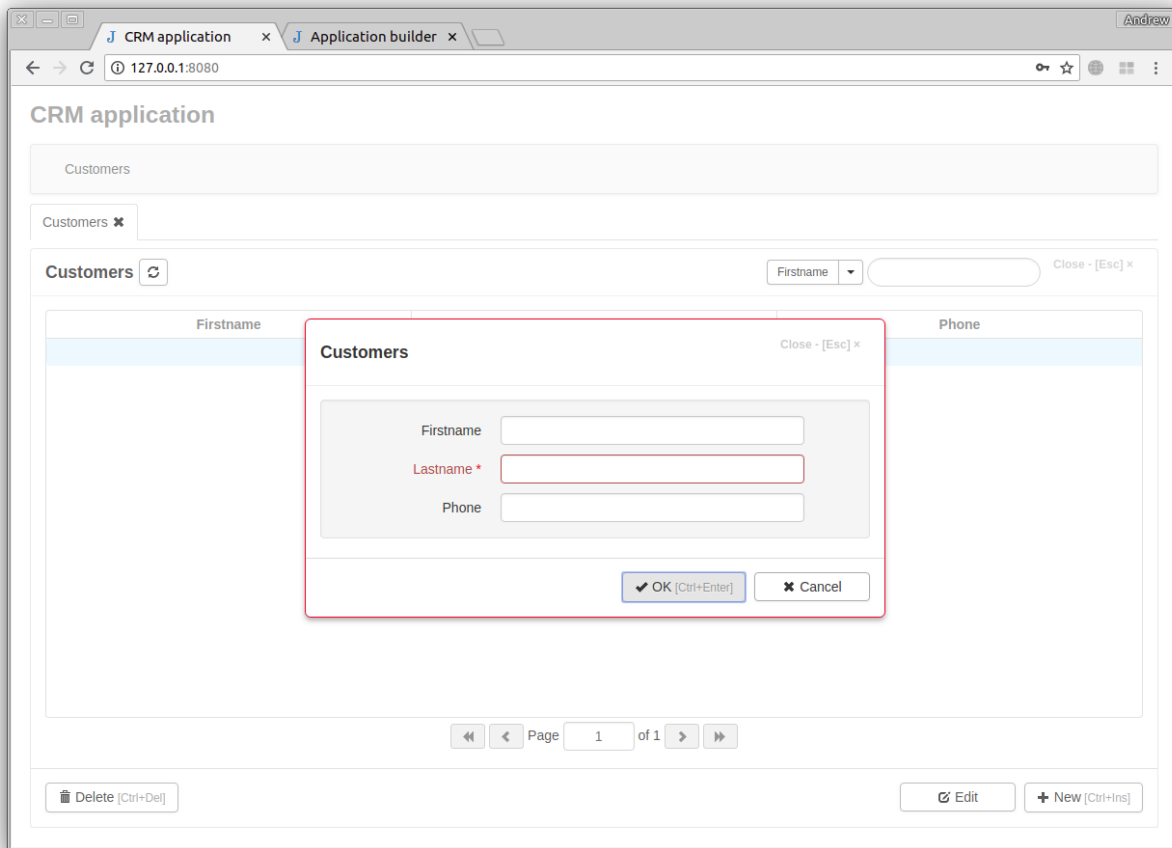
Similarly, we added the “Lastname” and “Phone” fields. When adding the “Lastname” field, we checked the **Required** attribute.



Now, to save the changes, click the **OK** button. When saving, the Application builder created the CRM_CUSTOMERS table in the ctm.sqlite database:

```
127.0.0.1 - - [07/Aug/2018 11:32:30] "POST /api HTTP/1.1" 200 -
CREATE TABLE "CRM_CUSTOMERS"
(
  "ID" INTEGER PRIMARY KEY,
  "DELETED" INTEGER,
  "FIRSTNAME" TEXT,
  "LASTNAME" TEXT,
  "PHONE" TEXT)
127.0.0.1 - - [07/Aug/2018 11:32:30] "POST /api HTTP/1.1" 200 -
```

Go to the Project page, refresh it and click the **New** button and then **OK** button:



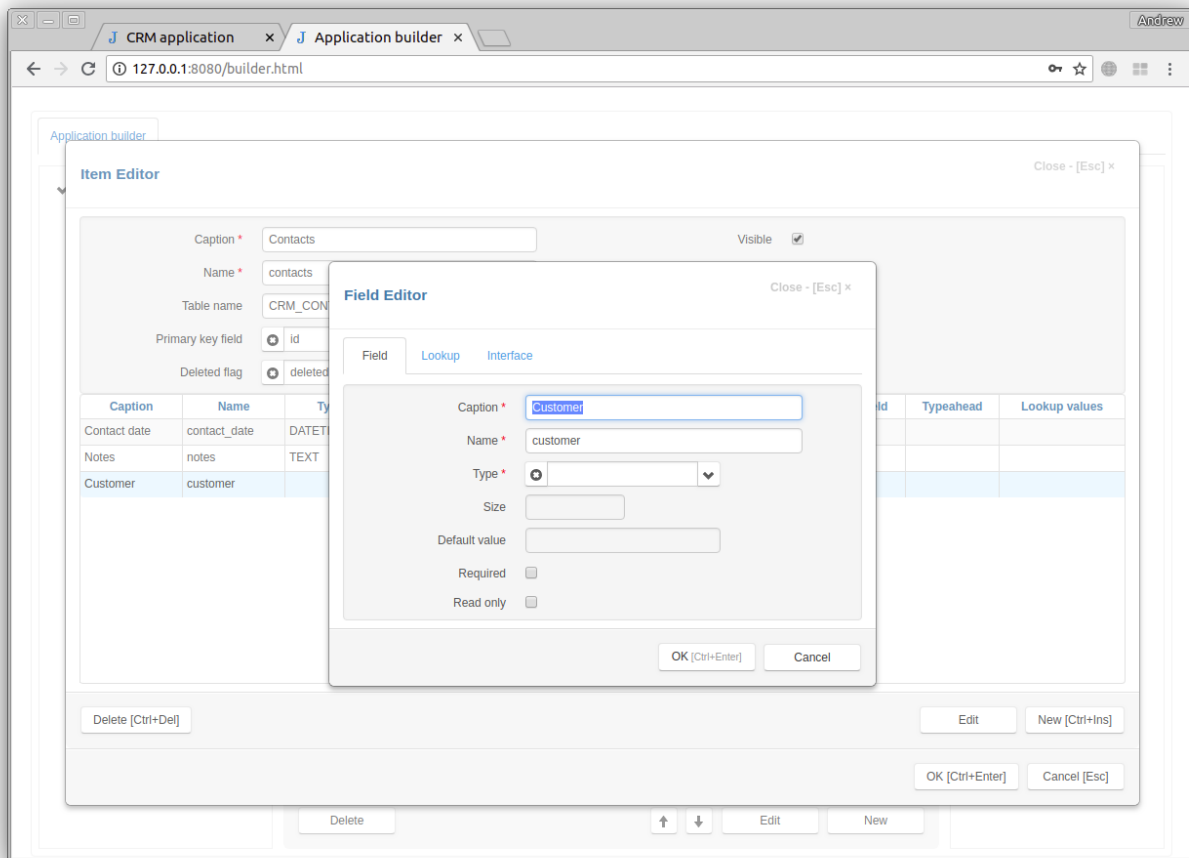
2.4.3 Lookup fields

Now we will create the “Contacts” item.

Select the “Journals” group in the project task tree and add a new journal in the same way that we created the “Customers” catalog.

First we add the “Contact date” field of the “datetime” type and then “Notes” fields of the “text” type.

Let’s add the *lookup field* “Customer” field that will store a reference to a record in the “Customers” catalog.



To create a lookup field, after specifying its caption and name, we need to select a lookup item. Select **Lookup** tab and click the button to the right of the **Lookup item** input

Field Editor Close - [Esc] x

Field **Lookup** Interface

Lookup item

Lookup field

Lookup field 2

Lookup field 3

Master field

Typeahead ☐

Lookup values

OK [Ctrl+Enter] Cancel

and double click the record to select it.

Items			Close - [Esc] x
ID	Name	Caption	
6	customers	Customers	

The same way specify a lookup field.

The screenshot shows the 'Field Editor' dialog box with the 'Lookup' tab selected. The dialog has a title bar with 'Close - [Esc] x'. Below the title bar are three tabs: 'Field', 'Lookup', and 'Interface'. The 'Lookup' tab contains the following fields:

Field Name	Value	Icon
Lookup item	customers	Folder icon
Lookup field	lastname	Folder icon
Lookup field 2		Folder icon
Lookup field 3		Folder icon
Master field		Folder icon
Typeahead	<input type="checkbox"/>	
Lookup values		Folder icon

At the bottom right of the dialog are two buttons: 'OK [Ctrl+Enter]' and 'Cancel'.

In the same way we add the “Firstname” and “Phone” lookup fields. For this fields we specify the “Customer” field as their **Master field** attribute.

Field Editor **firstname** Close - [Esc] x

Field

Lookup

Interface

Lookup item

*

customers

Lookup field

*

firstname

Lookup field 2

*

Lookup field 3

*

Master field

*

customer

Typeahead

☐

Lookup values


*


OK [Ctrl+Enter]


Cancel


Field Editor Close - [Esc] x


Field **Lookup** Interface

Lookup item * customers 


Lookup field * phone 

Lookup field 2 * 

Lookup field 3 * 

Master field * customer 

Typeahead ☐

Lookup values * 

OK [Ctrl+Enter] Cancel

Click the “OK” button to save the “Contacts” item.

```
127.0.0.1 - - [07/Aug/2018 14:24:11] "POST /api HTTP/1.1" 200 -
CREATE TABLE "CRM_CONTACTS"
(
  "ID" INTEGER PRIMARY KEY,
  "DELETED" INTEGER,
  "CONTACT_DATE" TEXT,
  "NOTES" TEXT,
  "CUSTOMER" INTEGER)
127.0.0.1 - - [07/Aug/2018 14:24:11] "POST /api HTTP/1.1" 200 -
```


As you can see, there are no “FIRSTNAME” and “PHONE” fields in the “CRM_CONTACTS” table. This is due to the fact that we have set **Master field** attribute of these fields to “Customer”. The “Customer” field will store a reference to a record in the “Customers” catalog and that record have the “Firstname” and “Phone” fields.

When we refresh the project page and click the **New** button we’ll see that there is a small button to the right of the “Customer” input.

ContactsClose - [Esc] ×

Contact date


+



Notes

Customer *

+



Firstname

Phone

✓ OK [Ctrl+Enter]

✗ Cancel

When we click on it and select a record in the “Customers” catalog the fields “Customer”, “Firstname” and “Phone” will be filled.

Contacts

Close - [Esc] x

Contact date

*

Notes

Customer *

*

Gordon

Firstname

John

Phone

999-111-2222

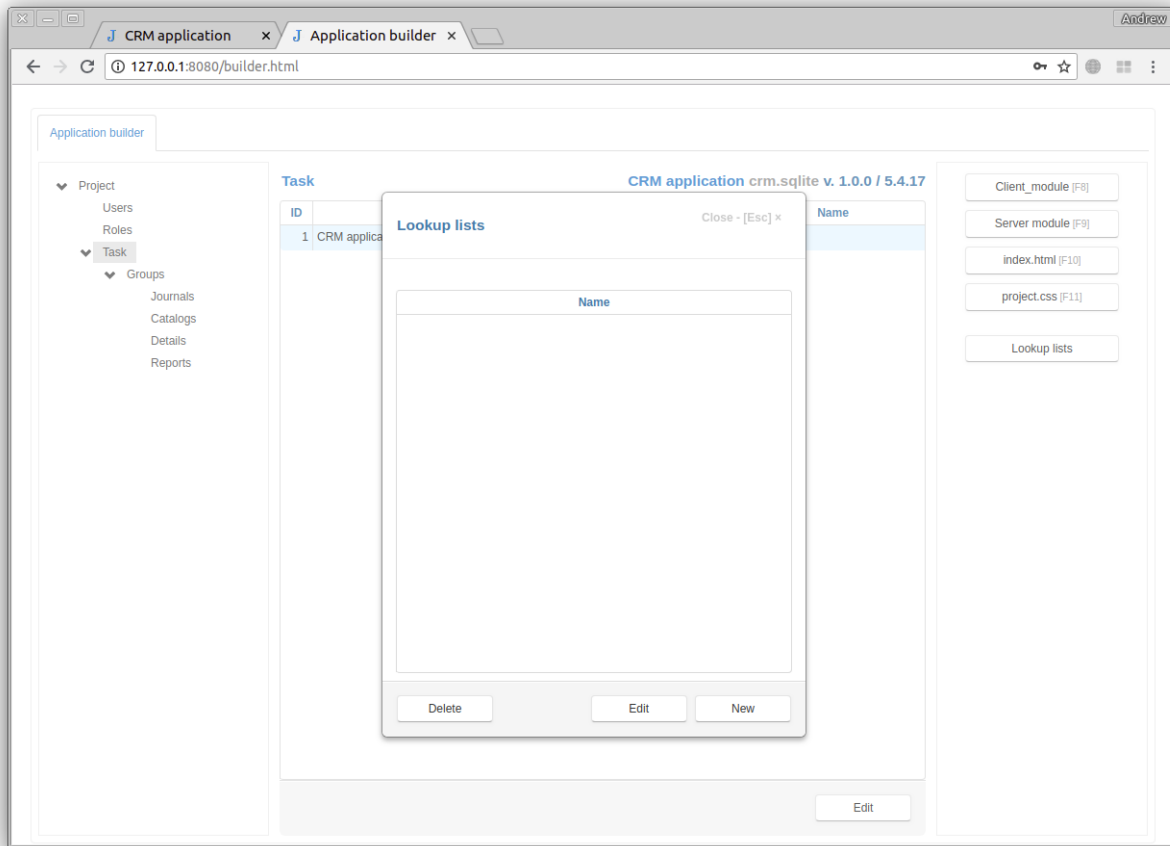
✓ OK [Ctrl+Enter]

✗ Cancel

2.4.4 Lookup lists

Now we create a *lookup List* “Status”.

Select the “Task” node in the project tree and click the **Lookup lists** button.



Click the **New** button and specify the new lookup list name and add a list of integer-text pairs:

Lookup lists

Close - [Esc] x

Name *

Status

Value	Lookup value
1	Contact

Lookup item

Close - [Esc] x

Value

4

Lookup value

Close

OK

Cancel

Delete [Ctrl+Del]

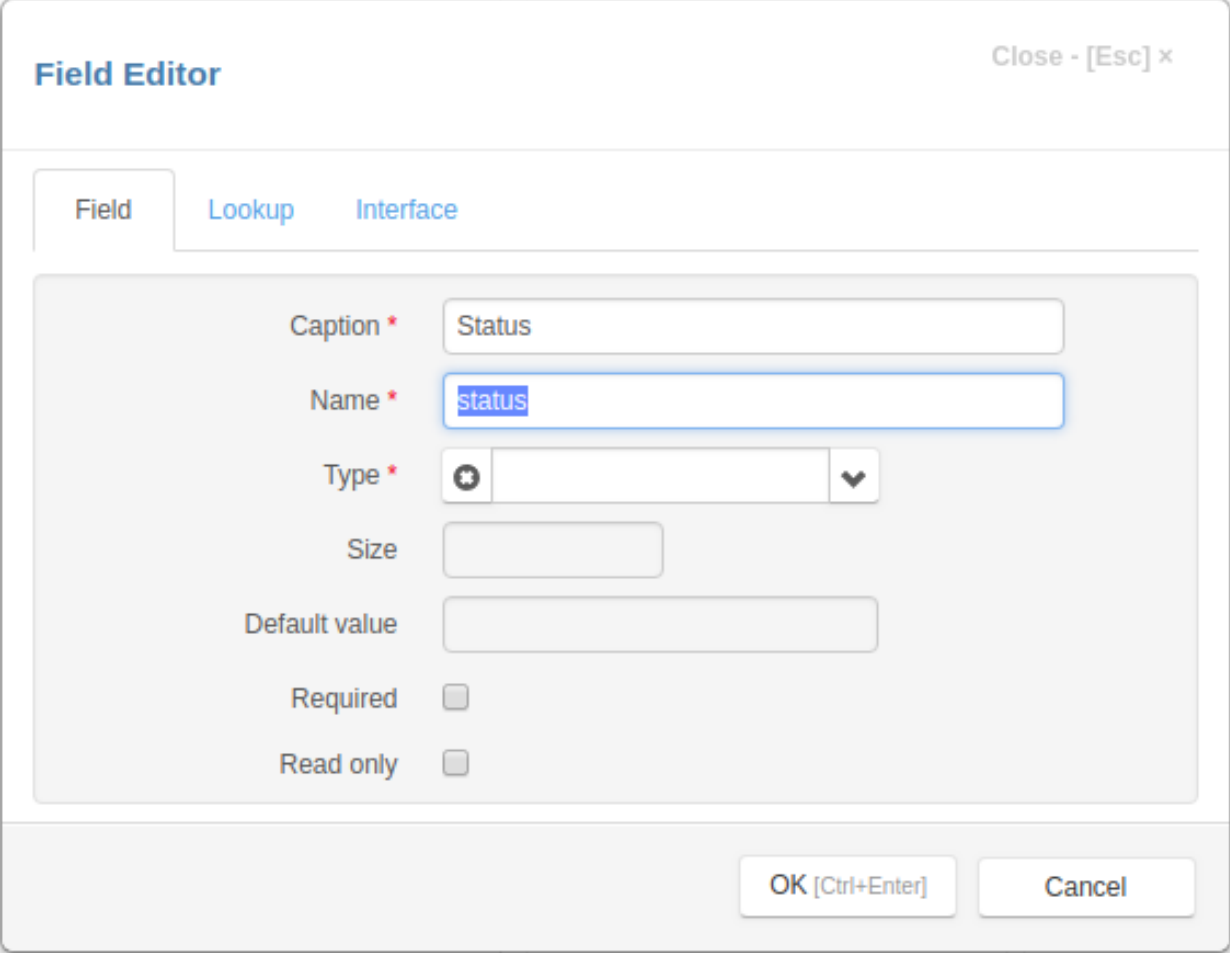
Edit

New [Ctrl+Ins]

OK

Cancel

Save the Lookup Lists dialog and open the “Contacts” journal to add the “Status” field

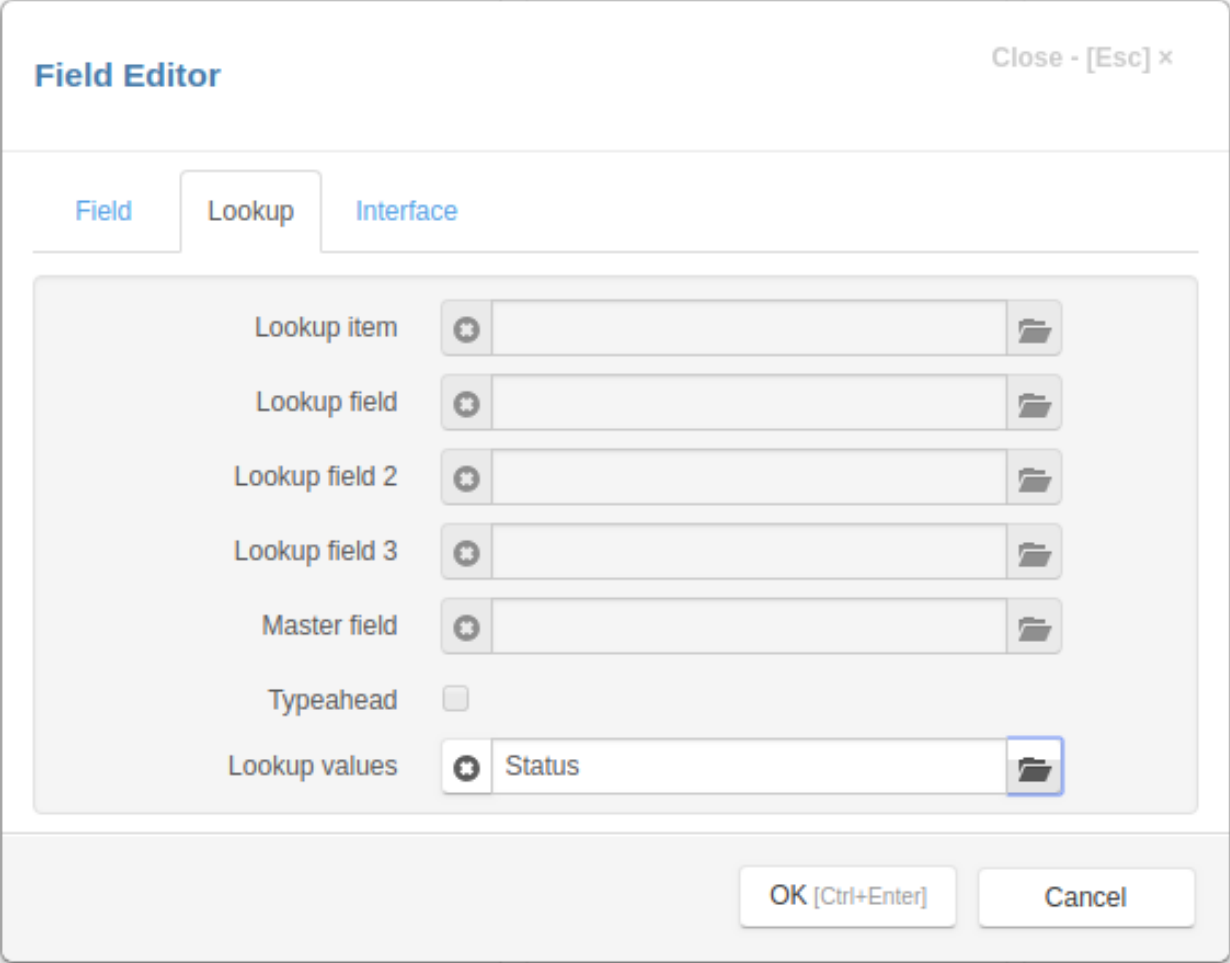


The image shows a 'Field Editor' dialog box with a title bar that includes a 'Close - [Esc] x' button. Inside the dialog, there are three tabs: 'Field', 'Lookup', and 'Interface'. The 'Field' tab is currently selected. The dialog contains several input fields and checkboxes:

- Caption ***: A text field containing the word 'Status'.
- Name ***: A text field containing the word 'status', which is highlighted with a blue selection box.
- Type ***: A dropdown menu with a gear icon on the left and a downward arrow on the right.
- Size**: A text field.
- Default value**: A text field.
- Required**: A checkbox that is currently unchecked.
- Read only**: A checkbox that is currently unchecked.

At the bottom right of the dialog, there are two buttons: 'OK [Ctrl+Enter]' and 'Cancel'.

and set the **Lookup values** attribute to the “Status” lookup list:



The image shows a 'Field Editor' dialog box with a title bar that includes a 'Close - [Esc] x' button. The dialog has three tabs: 'Field', 'Lookup', and 'Interface'. The 'Lookup' tab is currently selected. Inside the 'Lookup' tab, there are several configuration options, each with a required field icon (a circle with an asterisk) and a folder icon for selection:

- Lookup item: [required field] [folder icon]
- Lookup field: [required field] [folder icon]
- Lookup field 2: [required field] [folder icon]
- Lookup field 3: [required field] [folder icon]
- Master field: [required field] [folder icon]
- Typeahead: ☐
- Lookup values: [required field] Status [folder icon]

At the bottom of the dialog, there are two buttons: 'OK [Ctrl+Enter]' and 'Cancel'.

And finally, before saving, open the “Customer” field and set the **Required** and **Typeahead** attributes. When the **Typeahead** is checked, typeahead is enabled for the lookup field,

Field Editor customer Close - [Esc] x

Field

Lookup

Interface

Caption *

Customer

Name *

customer

Type *

⌵

INTEGER

⌵

Size

Default value

Required

☒

Read only

☐

OK [Ctrl+Enter]

Cancel

Field Editor customer Close - [Esc] x

Field

Lookup

Interface

Lookup item

Lookup field

Lookup field 2

Lookup field 3

Master field

Typeahead

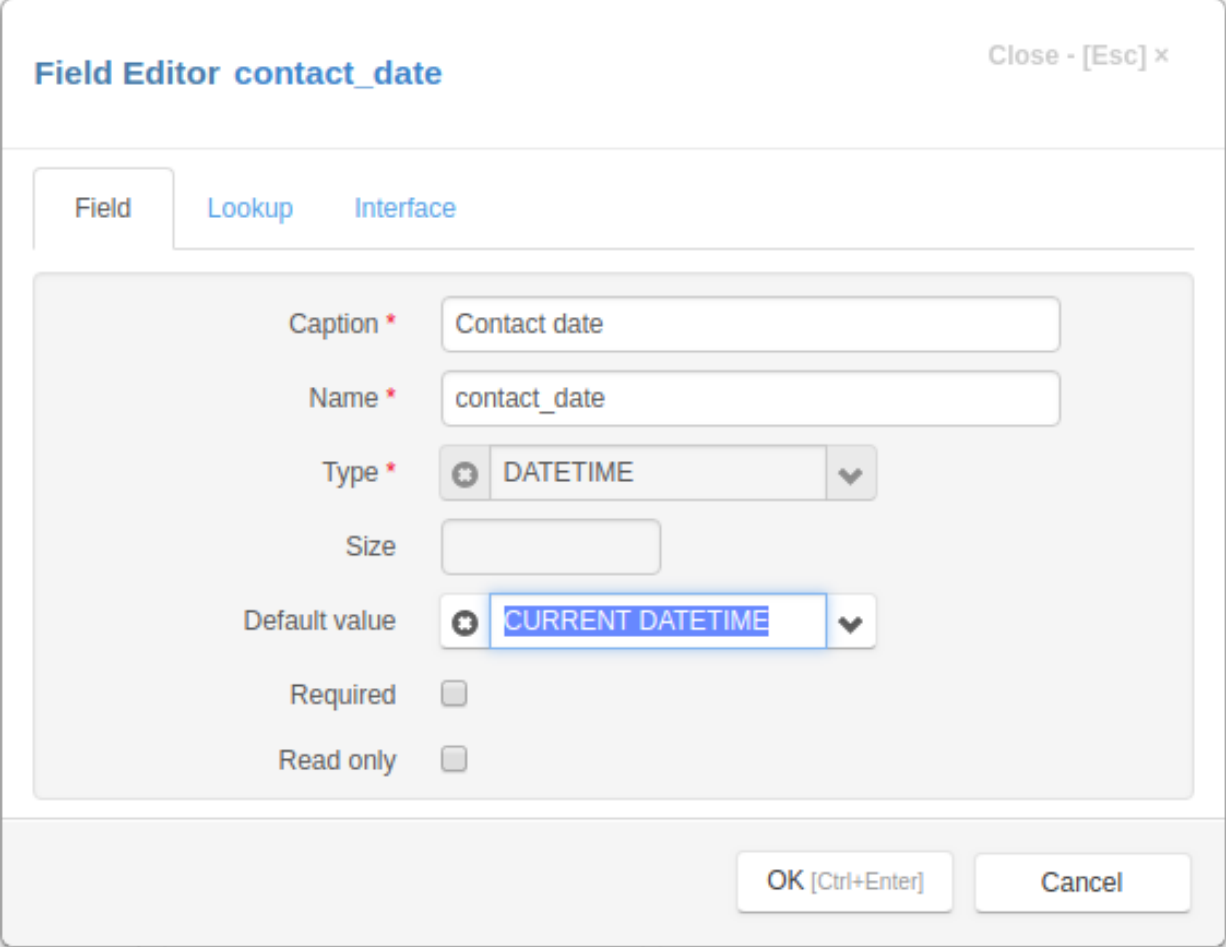
☒

Lookup values

OK [Ctrl+Enter]

Cancel

set **Default value** of the “Contact date” field to “CURRENT DATETIME”



The image shows a 'Field Editor' dialog box for a field named 'contact_date'. The dialog has a title bar with 'Field Editor contact_date' and a 'Close - [Esc] x' button. Below the title bar are three tabs: 'Field', 'Lookup', and 'Interface'. The 'Field' tab is selected. The dialog contains several input fields and checkboxes. The 'Caption' field is labeled 'Caption *' and contains the text 'Contact date'. The 'Name' field is labeled 'Name *' and contains the text 'contact_date'. The 'Type' field is labeled 'Type *' and is a dropdown menu with 'DATETIME' selected. The 'Size' field is an empty text box. The 'Default value' field is labeled 'Default value' and is a dropdown menu with 'CURRENT DATETIME' selected. There are two checkboxes: 'Required' and 'Read only', both of which are unchecked. At the bottom right of the dialog are two buttons: 'OK [Ctrl+Enter]' and 'Cancel'.

Field Editor contact_date Close - [Esc] x

Field Lookup Interface

Caption * Contact date

Name * contact_date

Type * DATETIME

Size

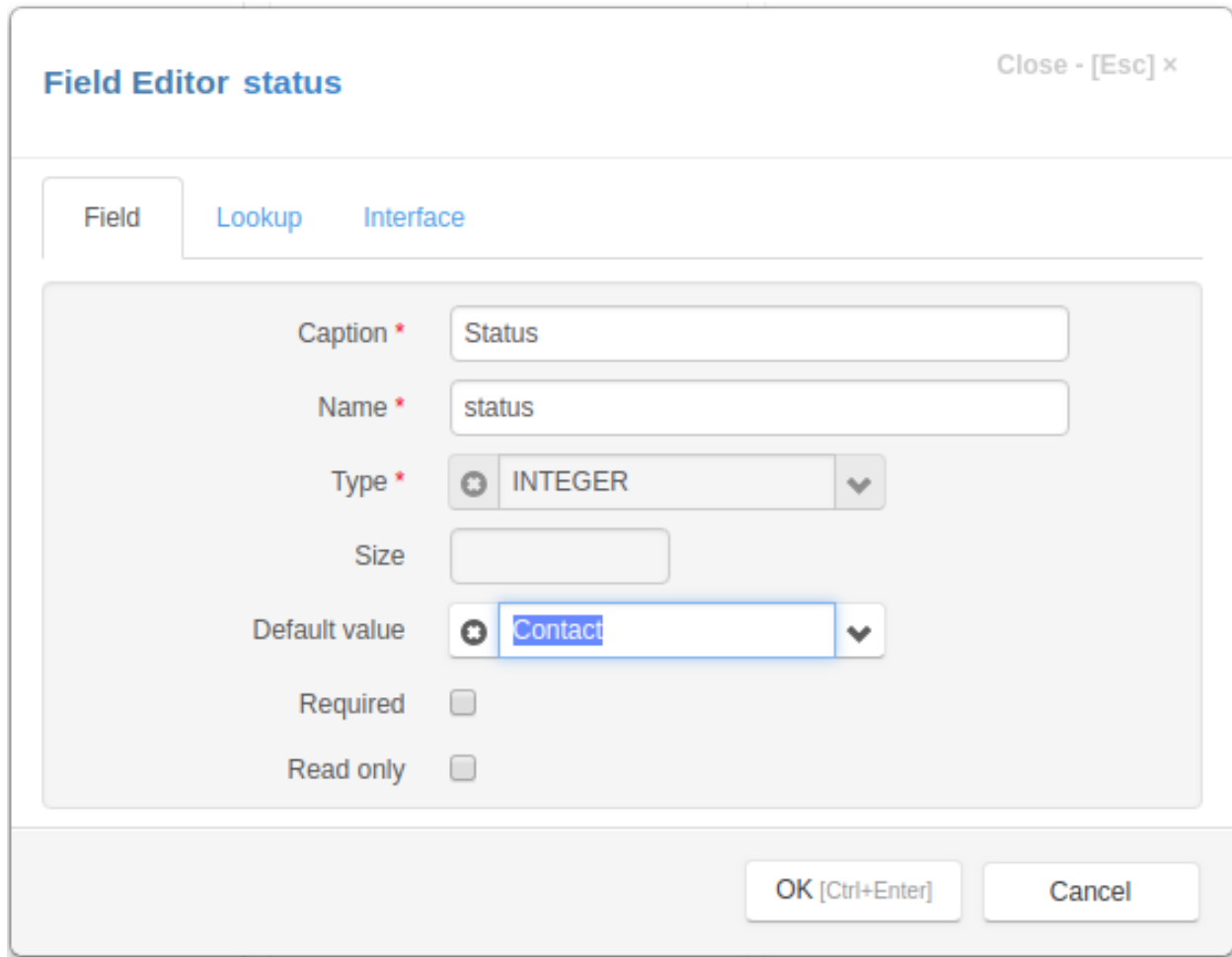
Default value CURRENT DATETIME

Required ☐

Read only ☐

OK [Ctrl+Enter] Cancel

and **Default value** of the “Status” field to “Contact” selecting them in the drop-down lists.



The image shows a 'Field Editor status' dialog box. At the top right is a 'Close - [Esc] x' button. Below the title bar are three tabs: 'Field' (selected), 'Lookup', and 'Interface'. The main area contains several fields: 'Caption *' with the value 'Status', 'Name *' with the value 'status', 'Type *' with a dropdown menu showing 'INTEGER', 'Size' with an empty text box, 'Default value' with a dropdown menu showing 'Contact', 'Required' with an unchecked checkbox, and 'Read only' with an unchecked checkbox. At the bottom right are 'OK [Ctrl+Enter]' and 'Cancel' buttons.

Field Editor status Close - [Esc] x

Field Lookup Interface

Caption * Status

Name * status

Type * INTEGER

Size

Default value Contact

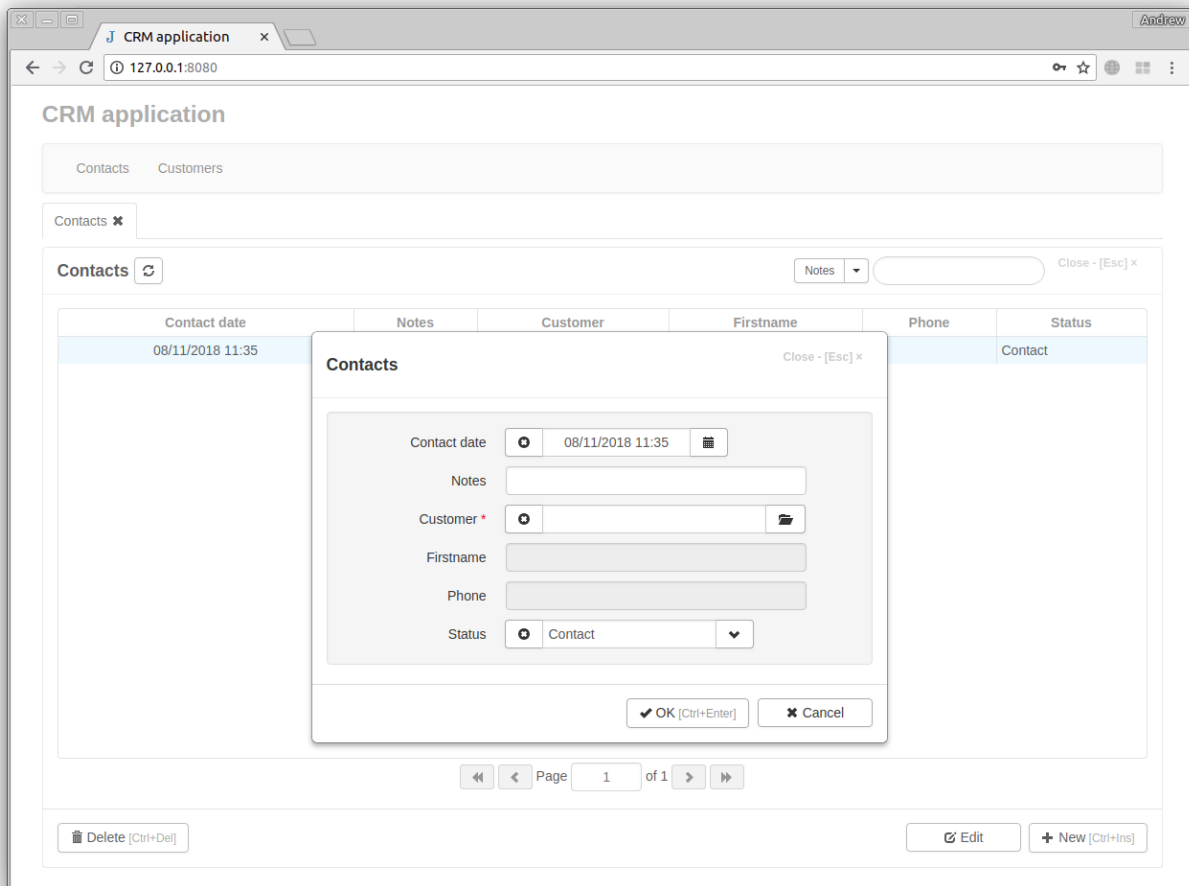
Required ☐

Read only ☐

OK [Ctrl+Enter] Cancel

2.4.5 Customizing Forms

When we refresh the project page we'll see that fields in the table and in the edit form of the "Contacts" journal are displayed in the order in which they were created.



To change how fields are displayed in the table, click the **View Form** button to open the *View Form Dialog*. Let's change the displayed fields using **left**, **right**, **up** and **down** buttons.

View form contacts Close - [Esc] x

Layout Form

Field	Width
contact_date	
customer	
phone	
status	
notes	

←

→

Field
deleted
firstname
id

↑

↓

Multiple selection ☐

Dbclick edit ☒

Number of rows

Height

Row lines

Selected row lines

Freeze columns

Sort fields

Summary fields

OK Cancel

Let's click on the button right to the **Sort fields** input and select the fields by which user can sort the contents of the table by clicking in the corresponding column header of the table.

Select fields to sort by

Close - [Esc] x

2	Name↑
<input checked="" type="checkbox"/>	contact_date
<input checked="" type="checkbox"/>	customer
<input type="checkbox"/>	notes
<input type="checkbox"/>	phone
<input type="checkbox"/>	status

To change the way the fields are displayed in the edit form click the **Edit Form** button to open the *Edit Form Dialog*

Edit form contacts Close - [Esc] x

Layout Form

+ -

Field

customer
status
notes
contact_date

← →

↑ ↓

Name

deleted
firstname
id
phone

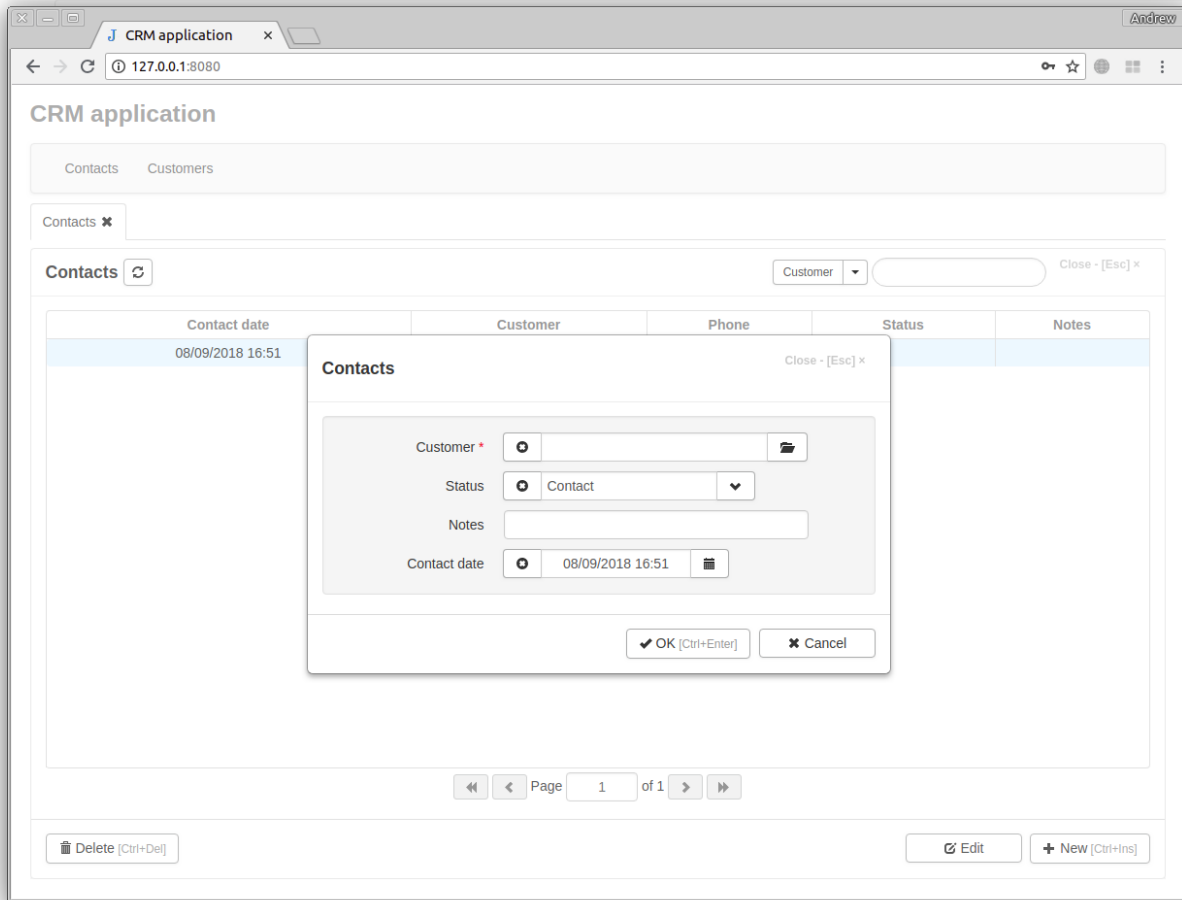
Columns 1

Label size Medium

In panel ☒

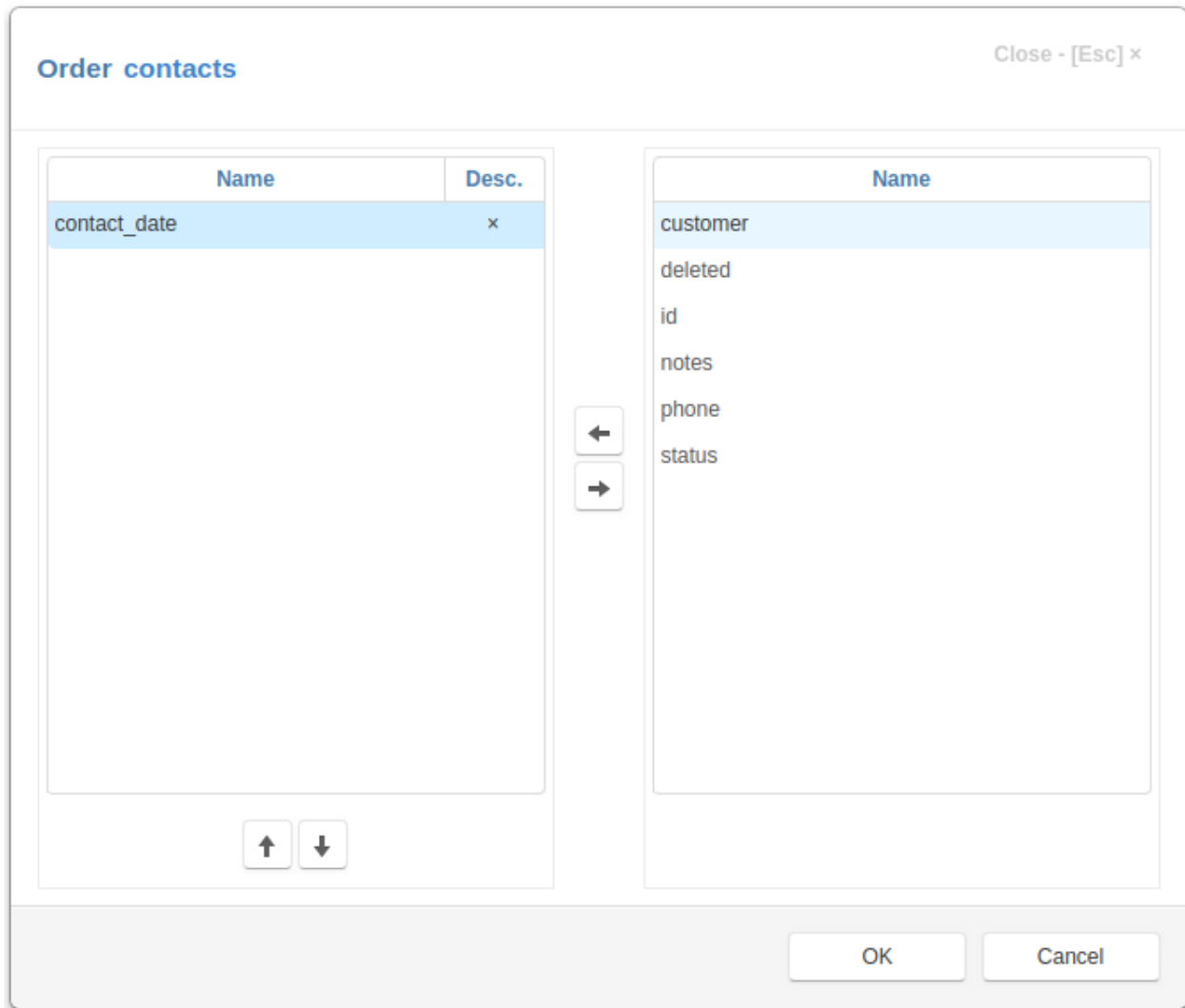
OK Cancel

To see the result of our work, go to the project page, refresh it and click the **New** button.



2.4.6 Indexes

Let's set the default sorting of records of the "Contacts" journal. To do so click the *Order* button:



And now we create a corresponding index for the “Contacts” journal database table. Click the **Indices** button to open *Indices Dialog* and then click the **New** button and specify the index:

Close - [Esc] x

Index * CRM_CONTACTS_IDX

Unique ☐

Name	Desc.
contact_date	x

↑

↓

Name
customer
deleted
id
notes
status

←

→

OKCancel

2.4.7 Filters

Filters are used to select records from the database table according to the specified criteria.

Click the **Filters** button to open *Filters Dialog*

Filters contacts

Close - [Esc] x

Caption	Name	Filter type	Field	Visible
---------	------	-------------	-------	---------

Delete

↑

↓

Edit

New


Now click the **New** button and fill out the following form:

Filters Close - [Esc] ×

Field

⚙

customer



Caption *

Customer

Name *

customer

Filter type

⚙

EQ

▼

Select all enabled

☐

Placeholder

Help

Visible

☒

OK

Cancel

Similarly, we created other filters:

Filters contacts

Close - [Esc] x

Caption	Name	Filter type	Field	Visible
Customer	customer	EQ	customer	x
Status	status	EQ	status	x
Contact date	contact_date	GE	contact_date	x
Notes	notes	CONTAINS	notes	x

Delete

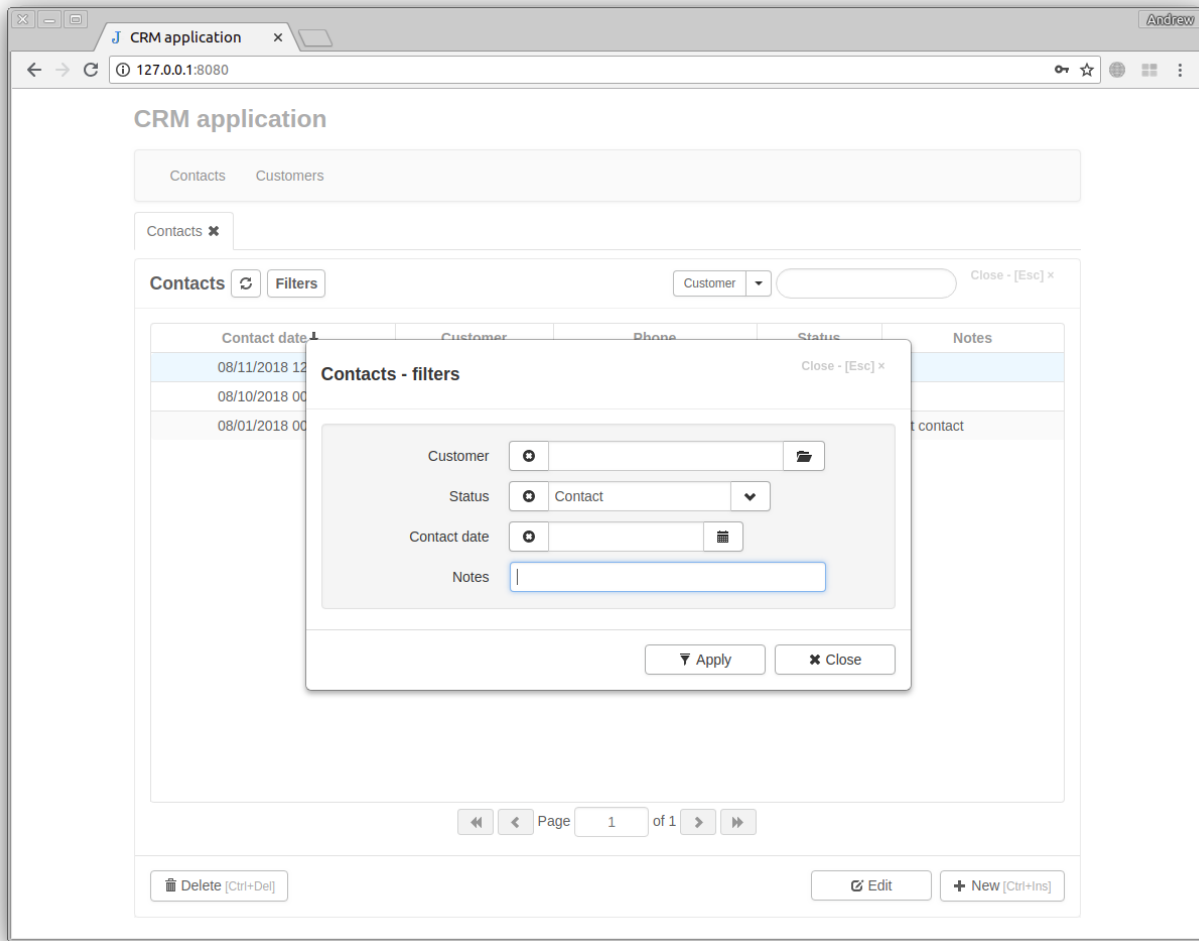
↑

↓

Edit

New

When we refresh the project page, the **Filters** button will appear in the header of the “Contacts” form. Clicking this button opens the “Filters” dialog box:



2.5 Tutorial. Part 2. File and image fields

In this part we will demonstrate how to work with files and images in Jam.py.

2.5.1 Adding Image field

Let's select the "Customers" catalog, Double-click it to open the *Item Editor Dialog* and add an image field "Photo":

Field Editor photo Close - [Esc] x

Field

Lookup

Interface

Caption *

Photo

Name *

photo

Type *

+

IMAGE

▼

Size

Default value

Required

☐

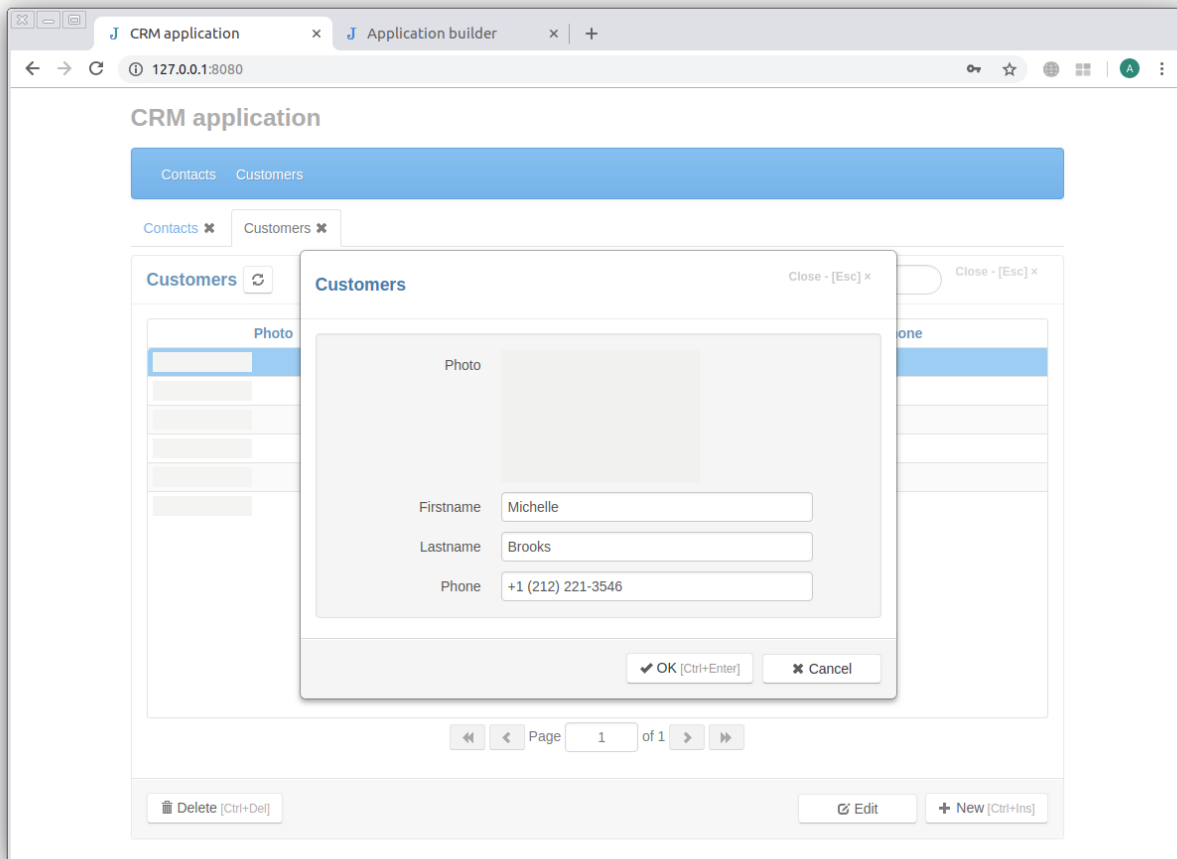
Read only

☐

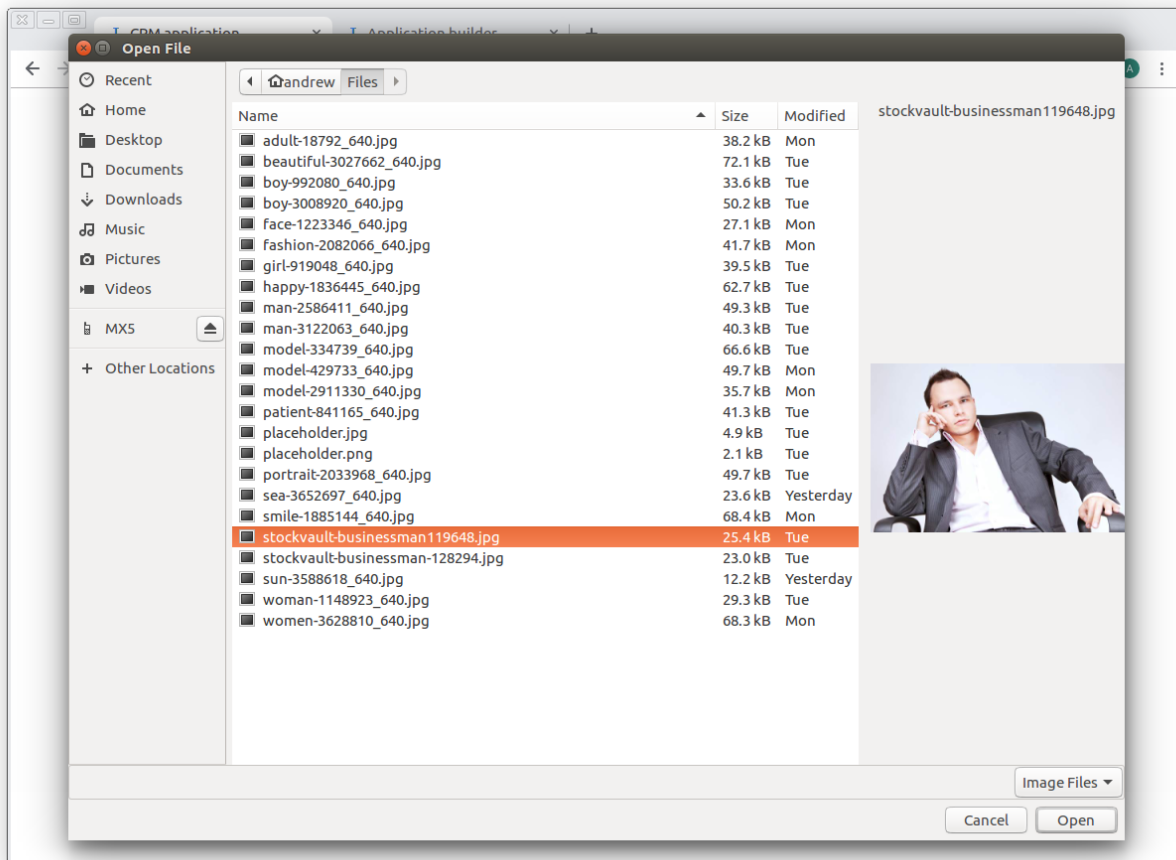
OK [Ctrl+Enter]

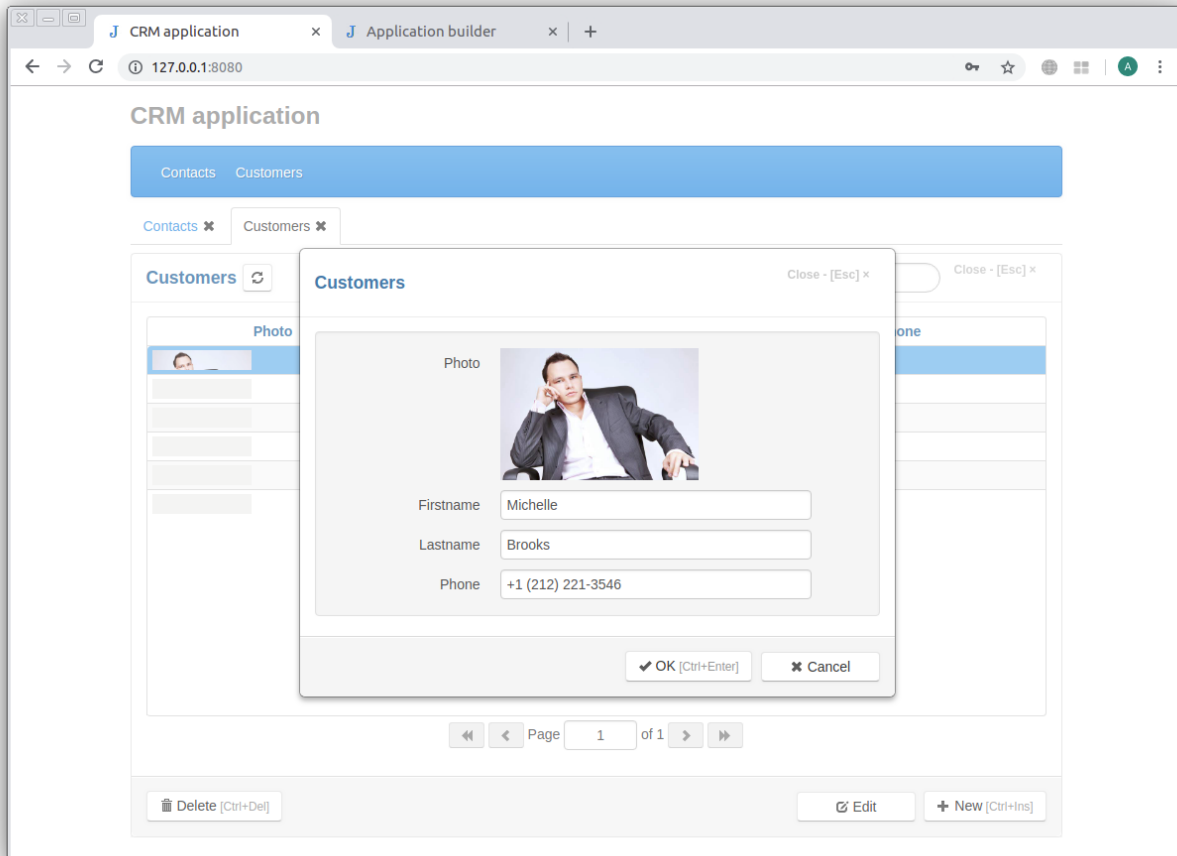
Cancel

Now refresh the project page, click the Customers menu item and open the edit form.



Double-click the image in the editing form to select an image from the Open File dialog box.





Note: To clear an image, hold down the Ctrl key and double-click the image.

Let's open the *Field Editor Dialog* in Application Builder and set **View width** to 120 and **Edit width** to 314 on the **Interface** tab.

Field Editor photo Close - [Esc] x

Field Lookup Interface

View width 100 Edit width 314

View height Edit height

Capture from camera ☐

Placeholder image

OK [Ctrl+Enter] Cancel

Note: You can set the image placehodler by double-clicking on it.

In the *View Form Dialog* we set **Row lines** to 4 and the width of the “Photo” field to 120.

View form customers Close - [Esc] x

Layout Form

Field	Width
photo	120
firstname	
lastname	
phone	

←

→

Field
deleted
id

↑

↓

Multiple selection ☐

Dblclick edit ☒

Number of rows

Height

Row lines

Selected row lines

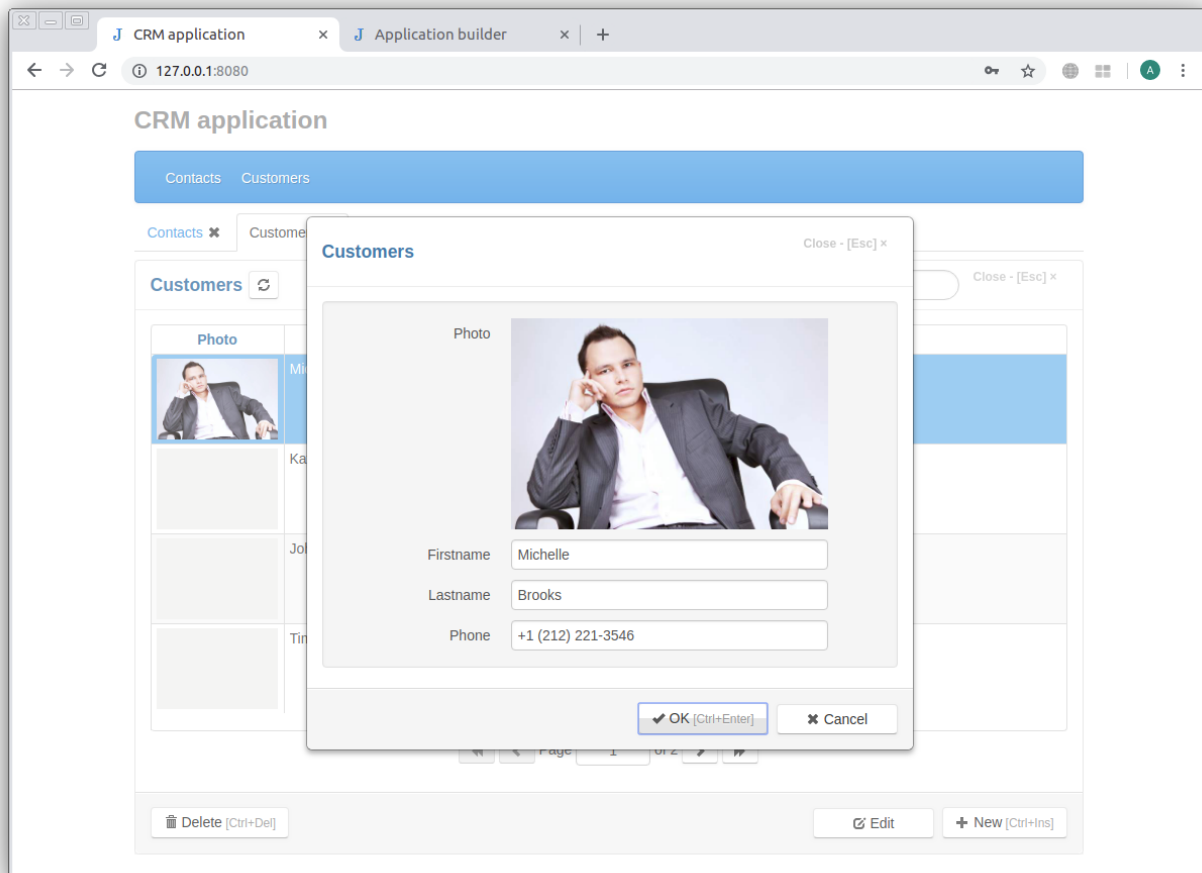
Freeze columns

Sort fields

Summary fields

OK Cancel

Now on the project page we will have:



2.5.2 Capturing image from camera

You can capture the image from the camera. To do so check the **Capture from camera** check box. In this case when the image is not set the video from camera will be displayed instead of the image placeholder.

Double-click the video to capture the image. To clear an image, hold down the Ctrl key and double-click the image, after that the video will be displayed.

2.5.3 Adding file field

Now we add a field that will store an appendix file to the “Contacts” journal.

Field Editor appendix Close - [Esc] x

Field

Lookup

Interface

Caption *

Appendix

Name *

appendix

Type *

⚙

FILE

▼

Size

Default value

Required

☐

Read only

☐

OK [Ctrl+Enter]

Cancel

This field will be displayed in the editing form as follows:

Contacts Close - [Esc] x

Customer *

Status

Notes

Contact date

Appendix

✓ OK [Ctrl+Enter] ✕ Cancel

The field input have three buttons on the right - to upload, to download and to open a file.

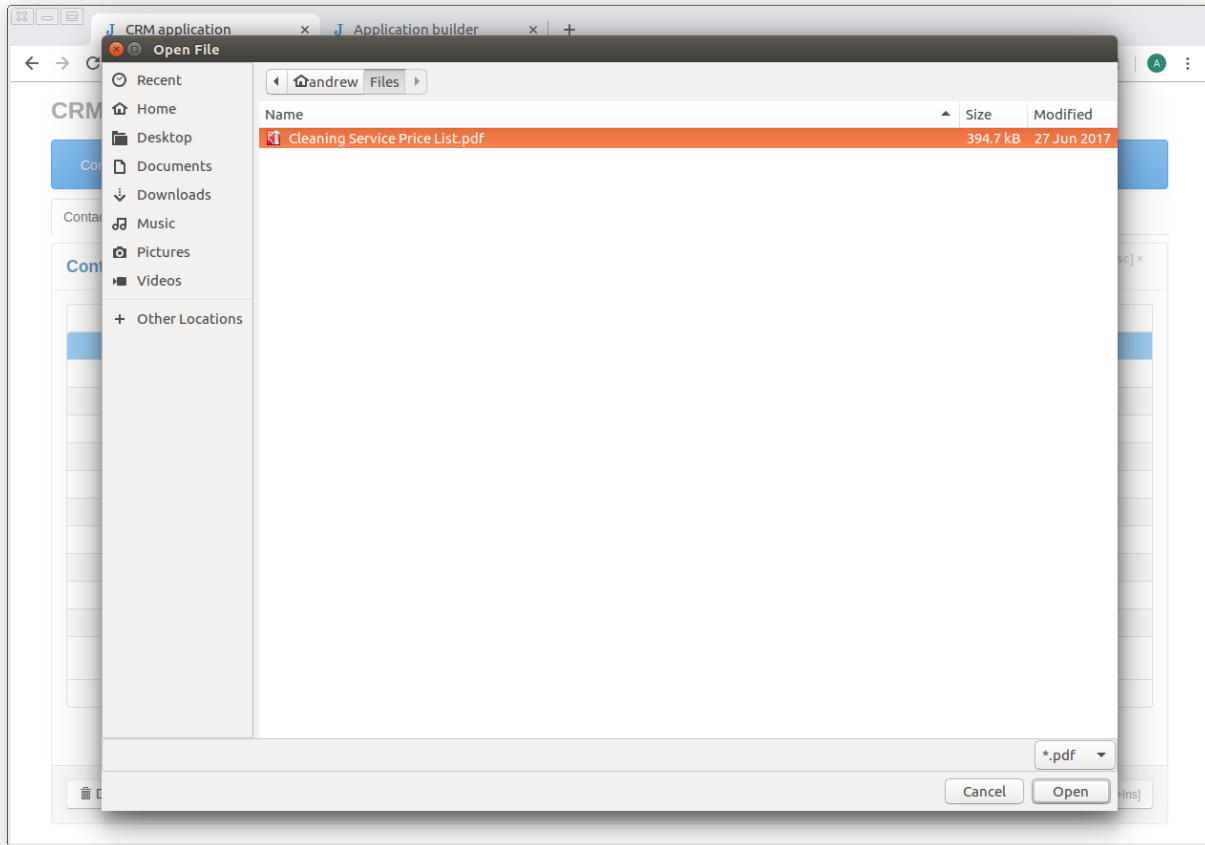
Let's open the *Field Editor Dialog* in Application Builder and uncheck the **Download btn** check box and set **Accept** attribute to '.pdf'.

The screenshot shows a window titled "Field Editor appendix" with a close button labeled "Close - [Esc] x". Below the title bar are three tabs: "Field", "Lookup", and "Interface". The "Interface" tab is selected. Inside the tab, there are four options, each with a checkbox and a label:

- Download btn: ☐
- Open btn: ☒
- Accept:
- Help:

At the bottom right of the window are two buttons: "OK [Ctrl+Enter]" and "Cancel".

Let's refresh the project page, open the "Contacts" edit form and upload a file by clicking the upload button:



Now we can open a file in the browser by clicking on the open button.

Contacts

Close - [Esc] x

Customer *

✱

Goyer

📁

Status

✱

Proposal

▼

Notes

Contact date

✱

09/17/2018 00:00

📅

Appendix

✱

Cleaning Service Price List.pdf

📄

▶

✓ OK [Ctrl+Enter]

✕ Cancel

CRM application x Cleaning_Service_Price_List x Application builder x +

127.0.0.1:8080/static/files/Cleaning_Service_Price_List2018-09-21_171809.287222.pdf?Cleaning%20Service%20Price%20List.pdf

Cleaning Services Price List

	Price	Quantity	Total
Cleaning Operative – 4 Hours	£60.00		
Cleaning Operative – 8 Hours	£120.00		
Cleaning Operative – 12 Hours	£180.00		
Production Waste – Standard Bin Bag	£5.00		
Bin Hire – 120 Litre Food Waste Bin	£12.50		
Bin Hire – 120 Litre Glass Bin	£17.00		
Bin Hire – 240 Litre Wheelie Bin	£17.00		
Bin Hire – 240 Litre Raw Meat Bin	£20.00		
Bin Hire – 1200 Litre Euro Bin	£30.00		
Bin Hire – 5 Litre Clinical Waste Bin including hazardous waste consignment note	£110.00		
Pallet Disposal	£5.00		
(All Prices are exclusive of VAT)		Sub Total	
		Vat	
		Grand Total	

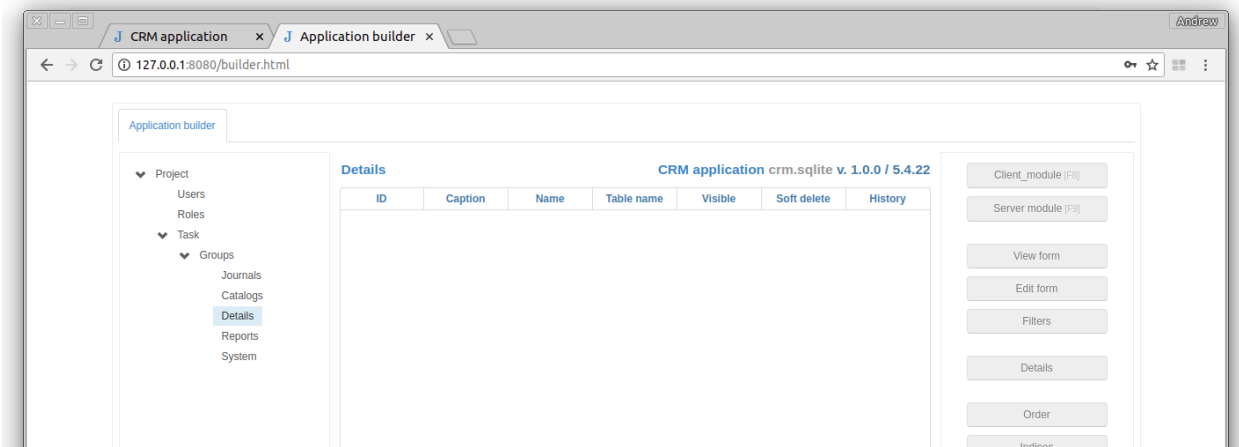
Note: Files and images are stored in the *static/files* folder on the server.

You can limit the size of files that can be uploaded to the server by setting **Max content length** attribute in the *project parameters*.

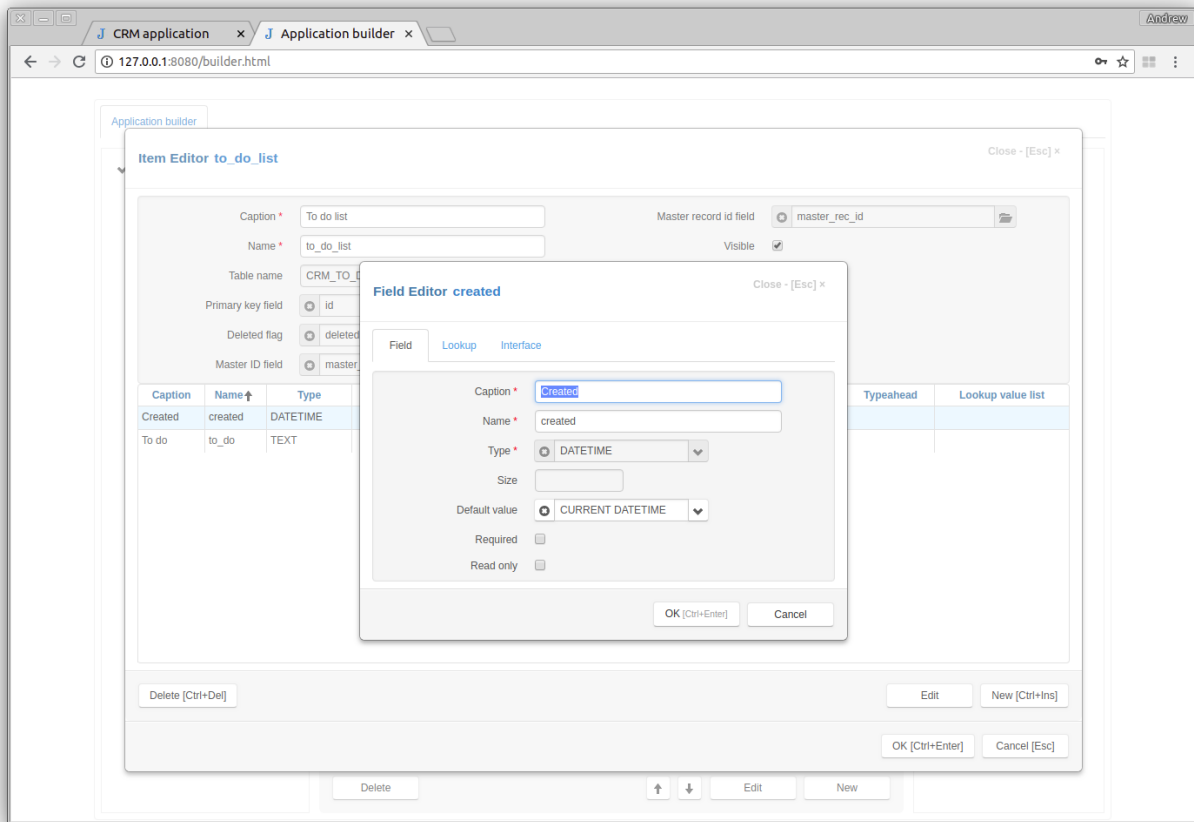
2.6 Tutorial. Part 3. Details

In this part of the tutorial we will explain how to work with details.

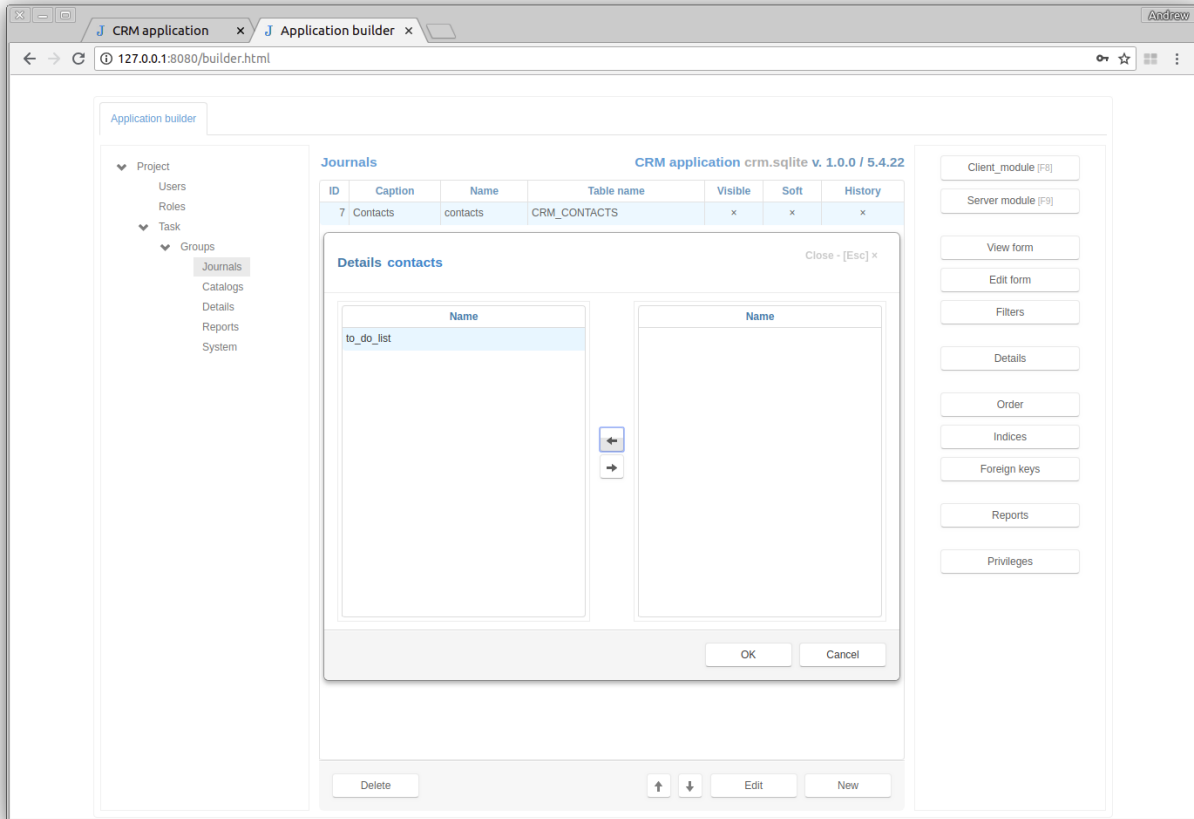
Let's select the “Details” group in the project tree and click the **New** button at the bottom right corner of the page



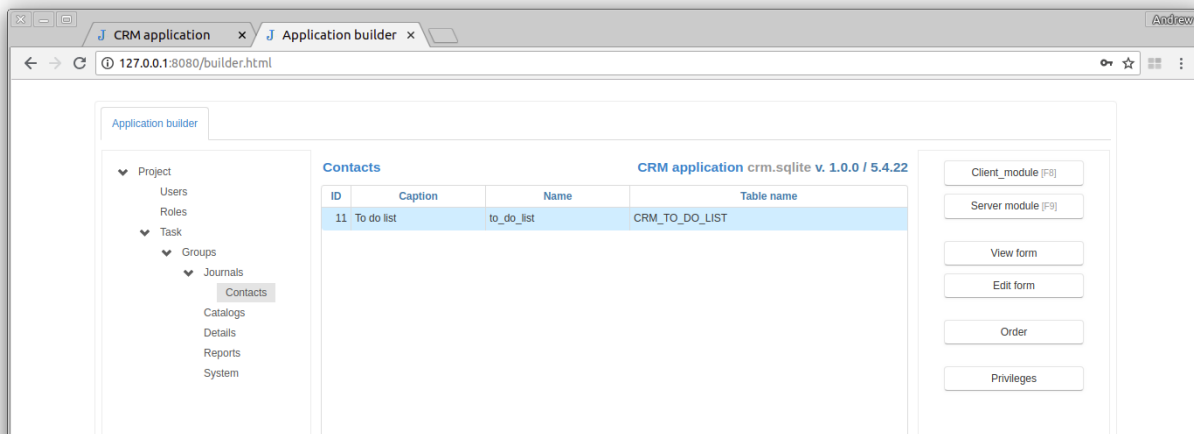
In the *Item Editor* dialog box, we will name the new item “To do list” and add the two fields “Created” and “To do” in the same way as in the previous tutorial:



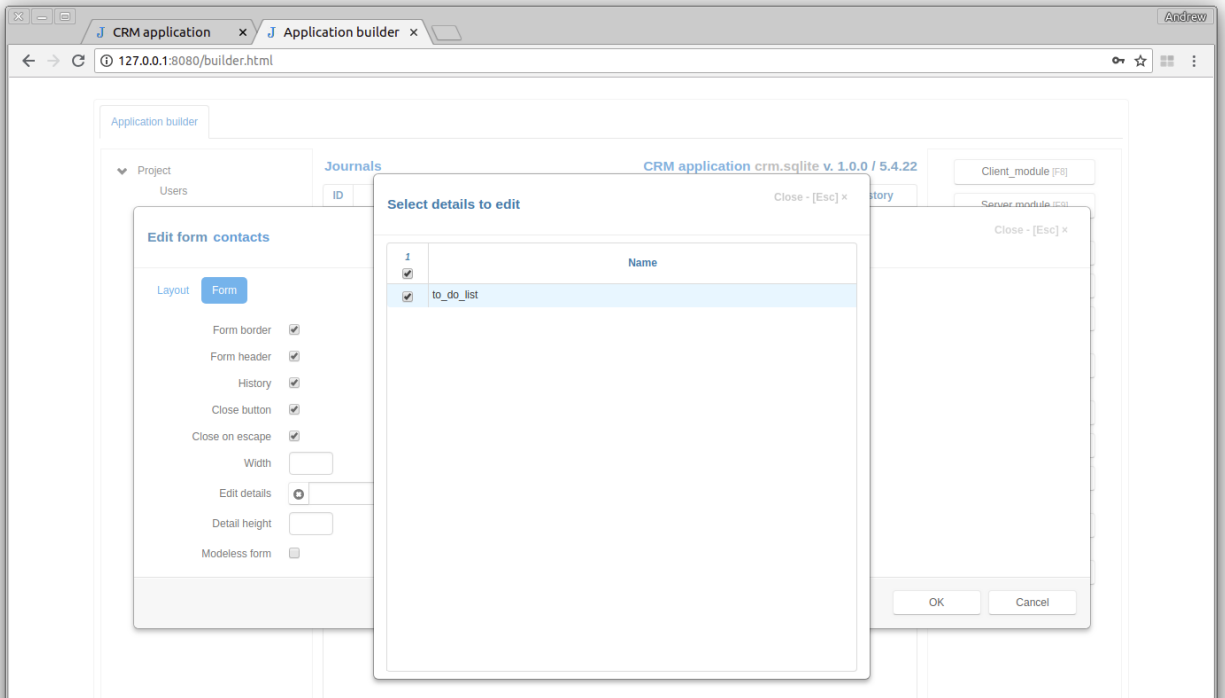
After saving the “To do list”, select the “Contacts” journal and click the **Details** button in the right pane to open the *Details Dialog*. Click the right arrow button to add the “To do list” to the “Contacts” details and the **OK** button to save changes.



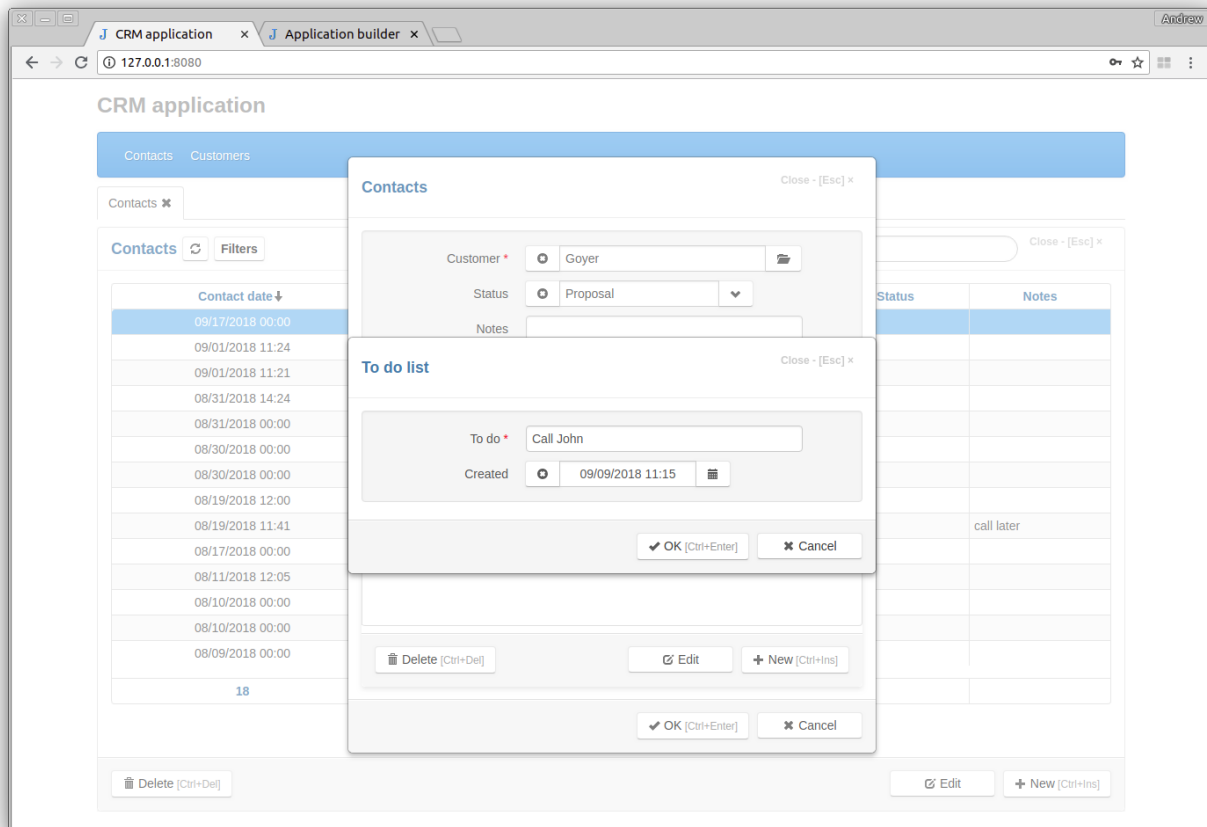
A new “To do list” item will be created as a child of the “Contacts” journal.



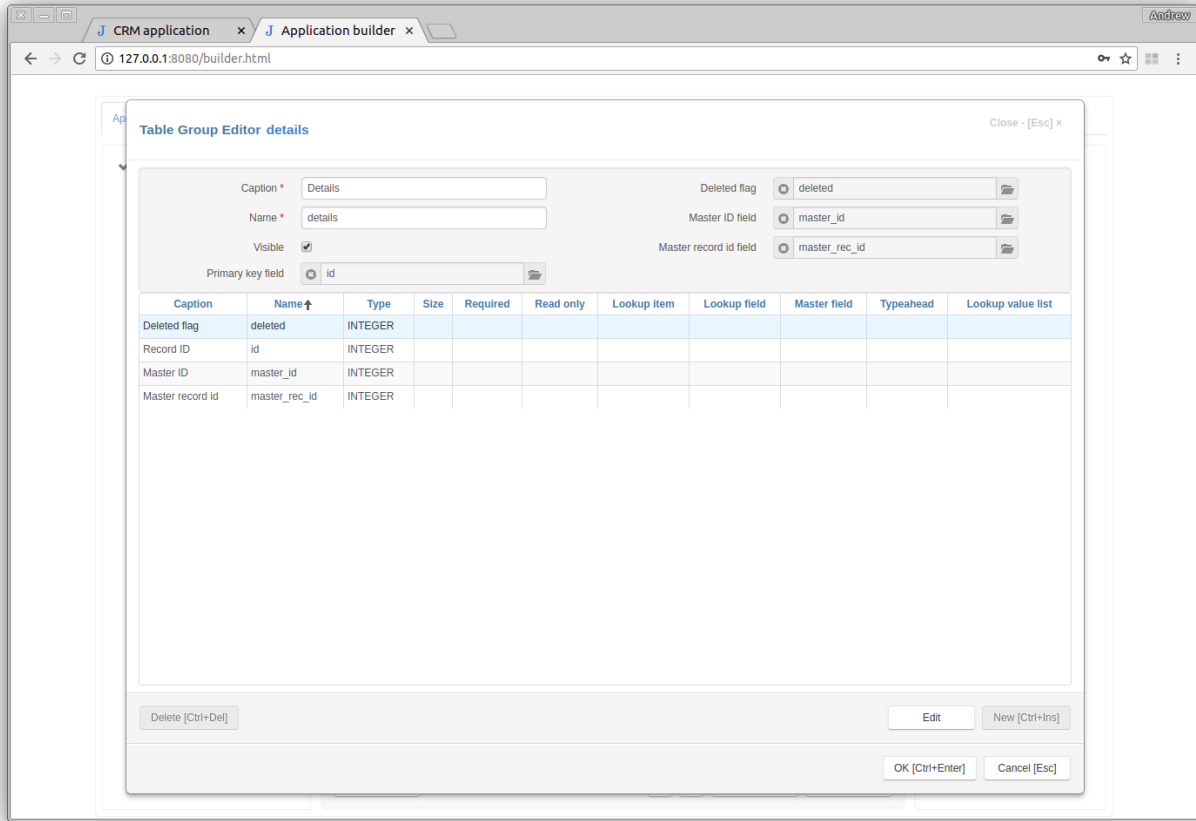
Select the “Contacts” journal again and click the **Edit form** button to open the *Edit Form Dialog*. Select **Form** tab, click the button to the right of the **Edit details** input and select the “To do list” check box.



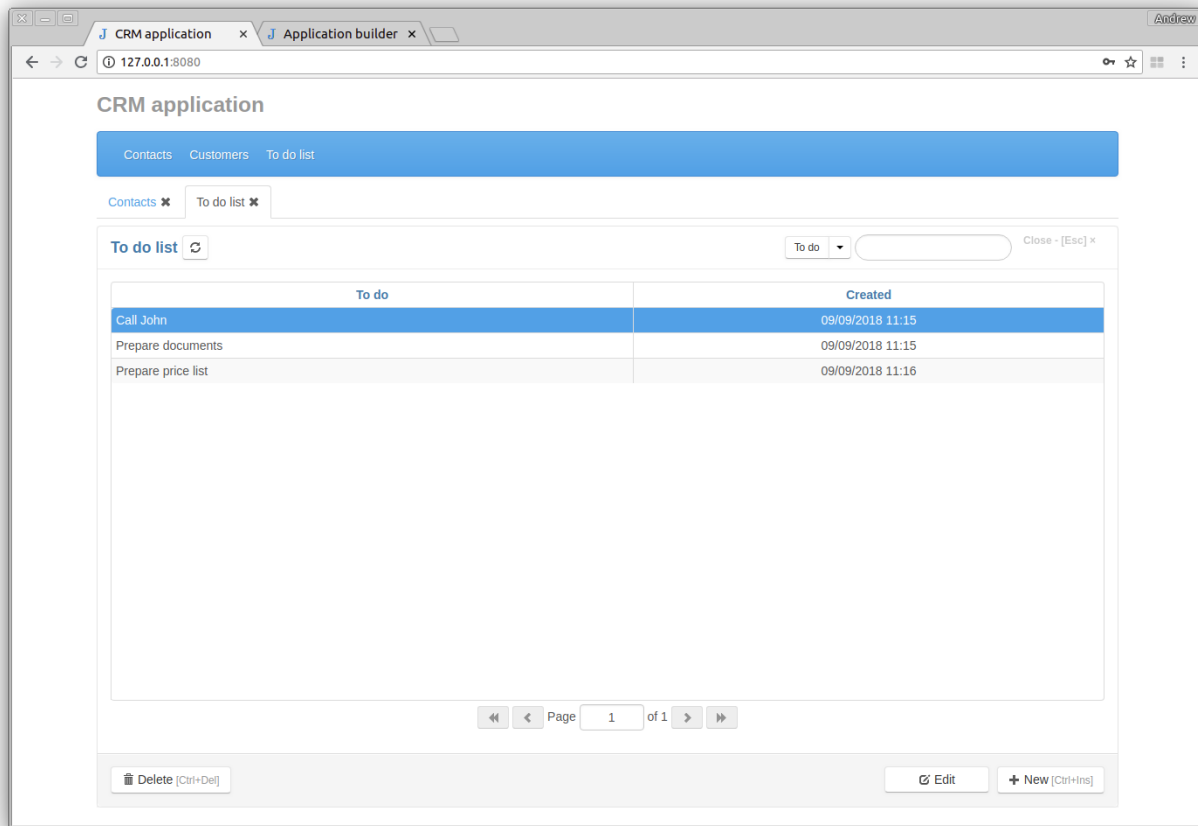
Let's update the project page and dblclick on the contact. Now we can add items to the to-do list of the contact.



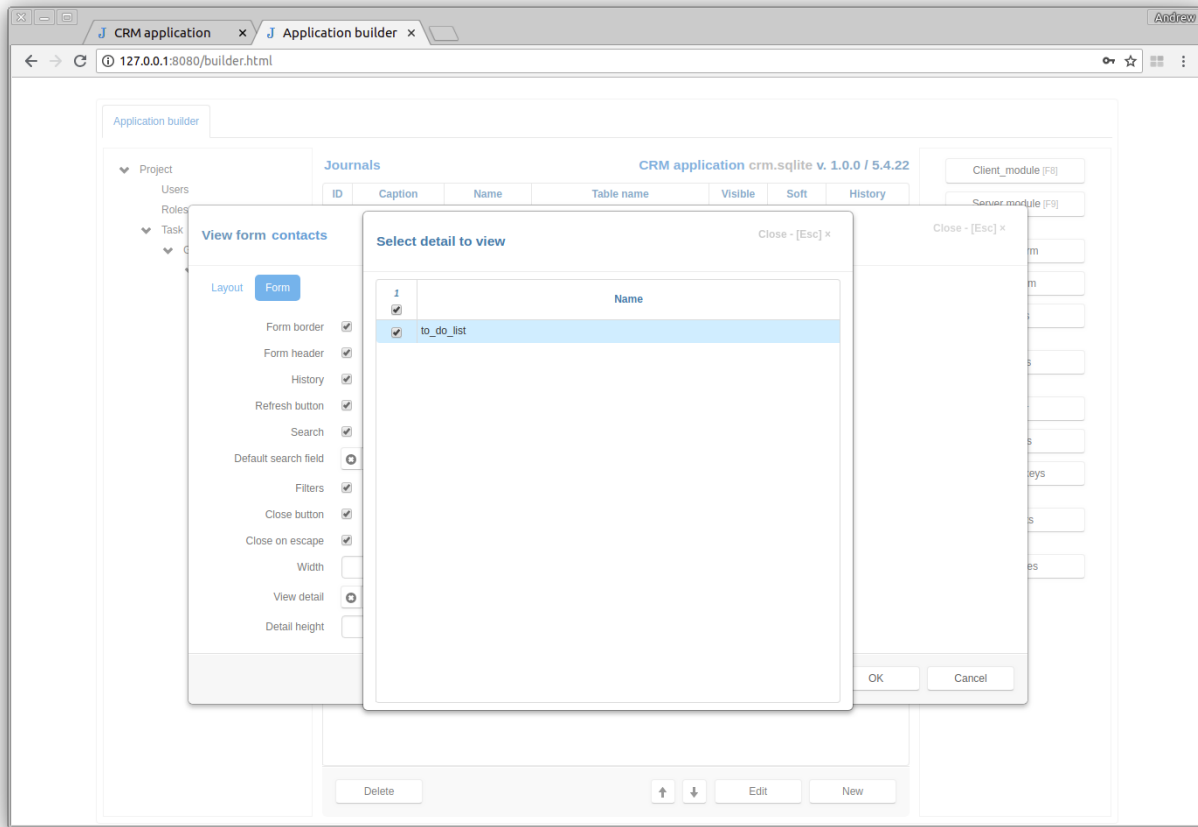
Click the **Groups** node in the project tree, dblclick the **Details** row and set Visible attribute to true.



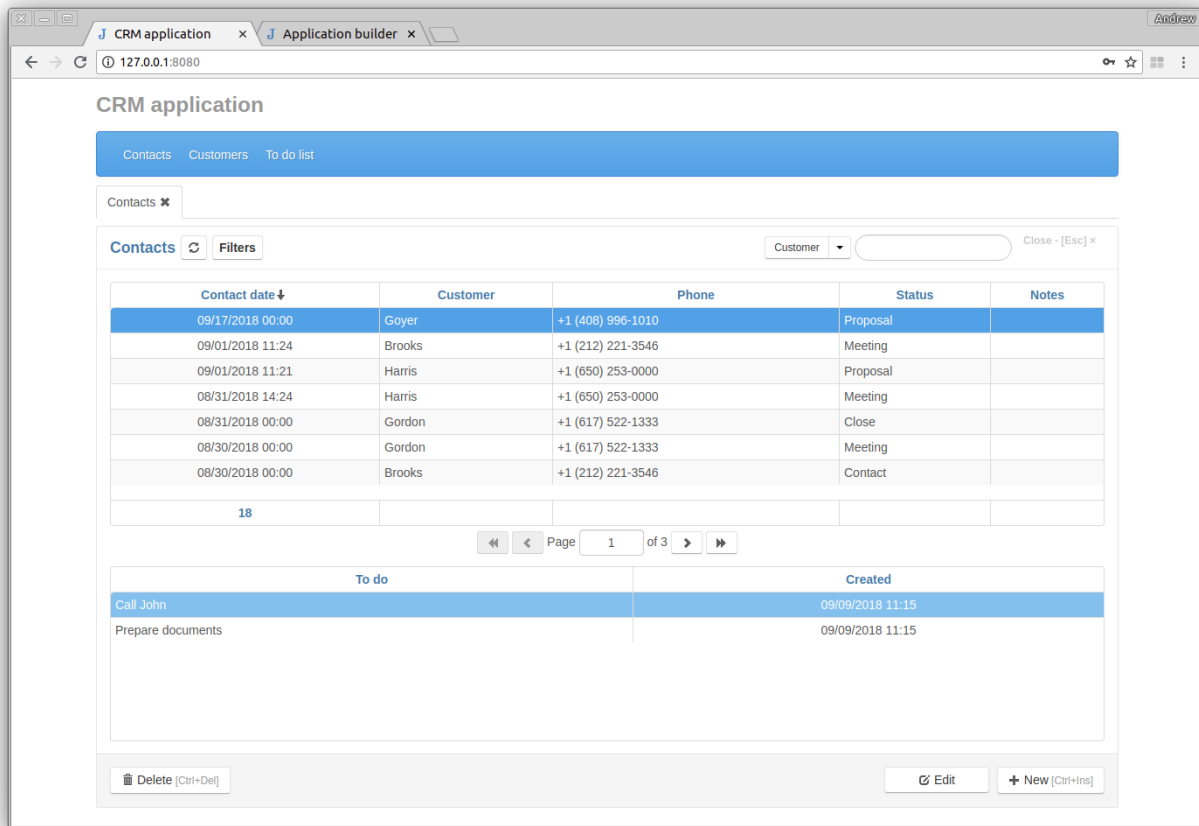
When we refresh the project page, we will see the “To do list” item in the main menu. Click on it to see the to do list of all contacts.



Select the “Contacts” journal again and click the **View form** button to open the *View Form Dialog*. Select **Form** tab, click the button to the right of the **View detail** input and select the “To do list” check box.



In the project page will see that the to-do list changes when the contact changes.



2.7 Jam.py deployment with Apache and mod_wsgi

Once you've got `mod_wsgi` installed and activated, edit your Apache server's `httpd.conf` file and add the following. If you are using a version of Apache older than 2.4, replace **Require all granted** with **Allow from all** and also add the line **Order deny,allow** above it.

```
WSGIScriptAlias / /path/to/mysite.com/mysite/wsgi.py
WSGIPythonPath /path/to/mysite.com

<Directory /path/to/mysite.com/mysite>
<Files wsgi.py>
Require all granted
</Files>
</Directory>

Alias /static/ /path/to/mysite.com/static/

<Directory /path/to/mysite.com/static>
Require all granted
</Directory>
```

The first bit in the `WSGIScriptAlias` line is the base URL path you want to serve your application at (`/` indicates the root url), and the second is the location of a “WSGI file” – see below – on your system, usually inside of your project package (`mysite` in this example). This tells Apache to serve any request below the given URL using the WSGI application defined in that file.

The `WSGI PythonPath` line ensures that your project package is available for import on the Python path; in other words, that `import mysite` works.

The `<Directory>` piece just ensures that Apache can access your `wsgi.py` file.

The next lines ensure that anything in the `/static/` URL space is explicitly served as a static files.

2.7.1 See also

See the additional information on the deployment in the [How to deploy](#)

Here, the basic concepts of Jam.py programming will be explained.

3.1 Task tree

All objects of the framework represent a tree of objects. These object are called items.

All items of the tree have common ancestor class `AbstractItem` (*client reference / server reference*) and common attributes:.

- `ID` - unique in the framework ID of the item
- `owner` - immediate parent and owner of the item
- `task` - root of the task tree
- `items` - list of child items
- `item_type` - type of the item
- `item_name` - the name of the item that will be used in programming code to get access to the item
- `item_caption` - the item name that appears to users

At the root of the tree is the task item.

The task contains group items. There are three types of groups that have the following values of the `item_type` attribute:

- “items” - these groups contain items with “item” `item_type`, that can have associated database table.
- “details” - such groups also contain items that can have associated database tables, but they can be used to create details for other items (see *Details*).
- “reports” - these groups contain reports - items with “report” `item_type`, that are used to create reports.

You can create your own groups.

Items that can have associated database table can own details, that are used to store records that belong to a record of the master.

For example the task tree of the *Demo project* is:

```
/demo/
  catalogs/
    customers
    tracks
    albums
    artists
    genres
    media_types
  journals/
    invoices/
      invoice_table
  details/
    invoice_table
  reports/
    invoice
    purchases_report
    customers_report
```

At the root of the task tree is a task with the `item_name` **demo**. It has four groups: **catalogs**, **journals**, **details** and **reports**. The **catalogs**, **journals** groups have `item_type` “items”. The items they own are wrappers over the corresponding database tables. There is one detail item with `item_name` **invoice_table**, that also has its own database table, and three reports in the **reports** group.

The **invoices** journal has the **invoice_table** detail, which keeps a list of tracks in an customer’s invoice. So there are two items with the same name “invoice_table” (detail_item and detail). They share the same database table.

Every item is an attribute of its owner and all items, tables and reports are attributes the task as well (they all have a unique `item_name`).

A task is a global object on the client. To access it, just type `task` anywhere in the code.

On the server, the task is not global. Jam.py is an event-driven environment. Each event has as a parameter the item (or field) that triggered the event. Functions defined in the server module of an item that can be executed from the client module using the *server* method have the corresponding item as the first parameter as well.

Knowing an item, we can access any other item of the task tree. For example to get access to the **customers** catalog we can write

```
def on_apply(item, delta, params):
    customers = item.task.catalogs.customers.copy()
```

or just

```
def on_apply(item, delta, params):
    customers = item.task.customers.copy()
```

The hierarchical structure of the project is one of the bases of the DRY (don’t repeat yourself) principle of the framework.

For example, some methods of the items, when executed, successively generate events for the task, group and the item.

This way we can define a basic behavior for all items in the event handler of the task, that can be expanded in the event handler of the group, and finally, if necessary, can be specified in the event handler of the item itself. For more details see *Form events*

3.1.1 Video

The [Task tree](#) video tutorial demonstrates the task tree using *Demo project*

3.2 Workflow

In the Jam.py framework, two tasks work at the same time: the Application builder and the Project. Each of them represents a tree of objects - there is the Application builder task tree and the Project task tree. Therefore, before considering the Jam.py workflow, you need to familiarize yourself with the concept of the *task tree*.

The the Jam.py workflow is the following:

- When server.py is run it creates WSGI application that, in its own turn, creates the Application builder task tree.
- The Project task tree is created on the server by Application builder after the server receives first request from the Project client. To do so, the Application builder uses metadata stored in admin.sqlite database in the root folder of the project. After creating a task tree the server application triggers the *on_created* event, that can be used to initialize the server task tree.
- When an application on the client (Application builder or Project) is first run in the browser (after `builder.html` or `index.html` have been loaded) the empty task object is built that sends to the server a request to initialize itself.
- If the project *safe mode* parameter is set, the framework checks if a user is logged, before executing any request. If not, then the application on the client creates a login form, and after the user inputs its login and password, the client task sends the server a request to login.
- After successful login or if the project *safe mode* parameter is not set, the server sends the client information about the requested task. The task on the client builds its tree, based on this information, assigns event handlers to its objects and executes *on_page_loaded* event handler.
- In this event handler a developer should attach JQuery event handler functions to HTML elements of the DOM, defined in the `index.html` file. In these functions a developer can use methods of items of the *task tree* to perform some specific tasks. These methods, when executed, trigger different events in which other methods could be called and so on. See *Client side programming*.
- Items of the task tree, that have corresponding database tables, have methods to read and write data in the server database. See *Data programming*.
- The report items generate the reports on the server, based on the LibreOffice templates. See *Programming reports*.
- All the items, whose methods generate a request to the server, do it the following way: they call the method of the task that sends to the server the *ID* of the task, the *ID* the item, the type of the request and its parameters. The server on receiving the request, based on passed IDs, finds the task (it can be Project task or Application builder task) and the item on the server, executes the corresponding method with passed parameters and returns the result of the execution to the client. These server methods could trigger their own events that can override the default behavior. See *Server side programming*

3.2.1 Video

[Form events](#) and [Client-server interactions](#) video tutorials illustrate the workflow of Jam.py project.

3.3 Working with modules

For every item of the project *task tree* a developer can write code that will be executed on the client or server. In Application builder for every item there is two upper-right buttons **Client module** and **Server module**. Clicking on these will open the *code editor*.

Every item has a predefined set of events that could be triggered by application. An event is a function defined in the module of an item that starts with the **on_** prefix. All published events are listed in the Events tab of the information pane of the *code editor*

In the *code editor* the developer can write code for these events as well as define some functions.

For example the following code means that immediately after adding a new record to the Invoices journal of the Demo project, the value of the invoicedate field will be equal to the current date.

```
function on_after_append(item) {  
    item.invoicedate.value = new Date();  
}
```

Note: These events and functions became attributes of the item and could be accessed anywhere in the project code.

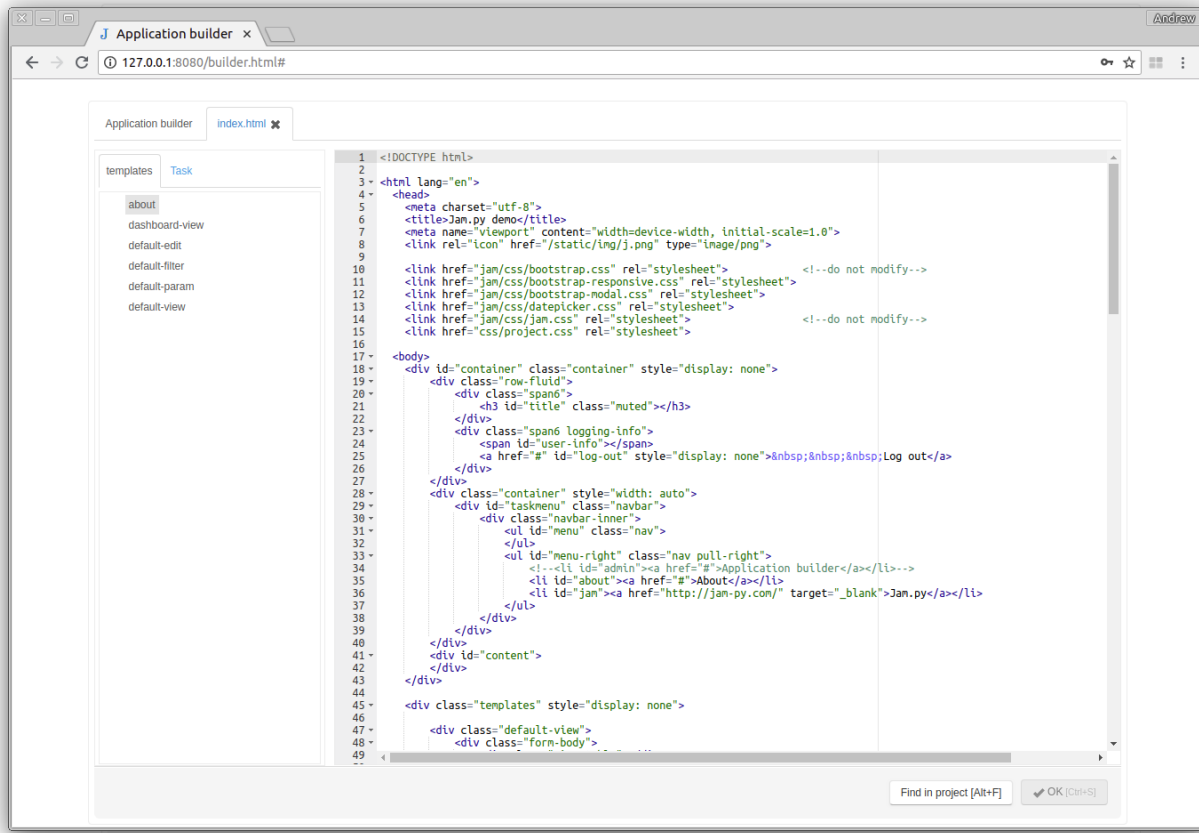
For example, the following code defined in the item client module will execute on_edit_form_created event handler defined in the **Customers** item for this item.

```
function on_edit_form_created(item) {  
    task.customers.on_edit_form_created(item);  
}
```

3.4 Client side programming

3.4.1 Index.html

When user opens a Jam.py application in a Web browser, the browser first loads the *index.html* file. This file is located in the root directory of a project.



It is the usual html file containing links to **css** and **js** files, that client application is using. The files that start with **jam** are located in the *jam* folder of the Jam.py package directory on the server.

For example

```
<link href="jam/css/jam.css" rel="stylesheet">
```

If needed, other files can be added here. For example some charting library. It is better to place them in the *js* and *css* folders of the *static* directory of the project.

For example

```
<script src="static/js/Chart.min.js"></script>
```

The *index.html* body tag have a *div* with class *templates*, that contains html templates of the project forms. See *Forms* and *Form templates*. for details.

At the end of the file there is a following script:

```
<script>
$(document).ready(function() {
    task.load()
});
</script>
```

In this script the *load* method of the task, that has been created when *jam.js* file was loaded, is called that loads information about the *task tree* from the server and, based on this information, builds its tree, loads modules, assigns event handlers to its items and triggers *on_page_loaded* event. See *Initializing application*

3.4.2 Initializing application

The *on_page_loaded* event is the first event triggered by an application on the client.

The new project uses *on_page_loaded* event handler to dynamically build the application's main menu and attach the on click event handler to menu items using JQuery.

```
function on_page_loaded(task) {

    $("#title").text(task.item_caption);
    $("#title").text(task.item_caption);

    if (task.safe_mode) {
        $("#user-info").text(task.user_info.role_name + ' ' + task.user_info.user_
↪name);
        $('#log-out')
        .show()
        .click(function(e) {
            e.preventDefault();
            task.logout();
        });
    }

    if (task.full_width) {
        $('#container').removeClass('container').addClass('container-fluid');
    }
    $('#container').show();

    task.create_menu($("#menu"), $("#content"), {view_first: true});

}
```

This event handler uses JQuery to select elements from the *index.html* to set their attributes and assign events.

```
<div id="container" class="container" style="display: none">
  <div class="row-fluid">
    <div class="span6">
      <h3 id="title" class="muted"></h3>
    </div>
    <div class="span6 logging-info">
      <span id="user-info"></span>
      <a id="log-out" href="#" style="display: none">Log out</a>
    </div>
  </div>
  <div class="container">
    <div id="taskmenu" class="navbar">
      <div class="navbar-inner">
        <ul id="menu" class="nav">
          </ul>
        </div>
      </div>
    </div>
    <div id="content">
      </div>
  </div>
</div>
```

Finally, the *create_menu* method of the task is called to dynamically create the main project menu.

3.4.3 Forms

One of the key concepts of the framework is the concept of form.

When the user clicks the menu item of the main menu, the *view* method of the corresponding item is executed, which creates the view form.

This view form can have the **New** and **Edit buttons**, clicking on which the *insert_record* and *edit_record* methods will be executed. These methods create an item edit form.

Forms are based on HTML *form templates* that determine their layout. Form templates are defined in the *Index.html* file, located in the root folder of the project.

The application already has default templates for viewing and editing data, for specifying filters and report parameters.

For example, all edit forms of the Demo project use the following html template:

```
<div class="default-edit">
  <div class="form-body">
    <div class="edit-body"></div>
    <div class="edit-detail"></div>
  </div>
  <div class="form-footer">
    <button type="button" id="ok-btn" class="btn expanded-btn">
      <i class="icon-ok"></i> OK<small class="muted">&nbsp;[Ctrl+Enter]</small>
    </button>
    <button type="button" id="cancel-btn" class="btn expanded-btn">
      <i class="icon-remove"></i> Cancel
    </button>
  </div>
</div>
```

You can define your own form templates to create your own custom forms. See *Form templates*

When some method creates a form the application finds corresponding html template.

If container (a JQuery object) parameter is specified, the method empties it and appends the html template to it, otherwise, it creates an empty modal form and appends the template to the form.

After this it assigns item's *prefix_form* attribute to the template, triggers an *on_prefix_form_created* events, shows the form and triggers *on_prefix_form_shown* events, where prefix is a type of the form (view, edit, filter, param). See *Form events* for details.

Below is an example of the *on_edit_form_created* event handler of the task:

```
function on_edit_form_created(item) {
  item.edit_form.find("#ok-btn").on('click.task', function() { item.apply_record() }
  ↪);
  item.edit_form.find("#cancel-btn").on('click.task', function(e) { item.cancel_
  ↪edit(e) });

  if (!item.master && item.owner.on_edit_form_created) {
    item.owner.on_edit_form_created(item);
  }
  if (item.on_edit_form_created) {
    item.on_edit_form_created(item);
  }

  item.create_inputs(item.edit_form.find(".edit-body"));
  item.create_detail_views(item.edit_form.find(".edit-detail"));
}
```

(continues on next page)

(continued from previous page)

```
    return true;
}
```

In this example, the `find` method of JQuery is used to find elements on the form.

First, we assign a JQuery `click` event to **OK** and **Cancel** buttons, so `cancel_edit` and `apply_record` methods will be executed when user clicks on the buttons. This methods cancel or apply changes made to the record respectively and call the `close_edit_form` method to close the form.

Then, if item is not a detail and has an event handler `on_edit_form_created`, defined in the owner's client module, this event handler is executed.

After that, if item has an event handler `on_edit_form_created`, defined in the item's client module, this event handler is executed.

In these event handlers some additional actions could be executed. For example you can assign click events to buttons or some other elements contained in your edit form template, change `edit_options`, create tables using the `create_table` method and so on.

Then the `create_inputs` method is called to create inputs in the element with class "edit-body"

Finally, `create_detail_views` method is called to create details in the element with class "edit-detail"

Note: If some elements are missing in the form template, an exception will not be raised.

The `close_prefix_form`, where `prefix` is the type of the form, closes the form of this type. But before form is closed the `on_prefix_form_close_query` and `on_prefix_form_closed` events are triggered. After form is closed it is removed from the DOM.

3.4.4 Form templates

Form templates of the project are located in the div with the `templates` class inside the `body` tag in the `Index.html` file.

When `load` method is executed, it cuts out the div with `templates` class from the `body` and stores it in the `templates` attribute as a JQuery object.

To add a form template for an item you should add a div with the `name-suffix` class in the `templates` div, where `name` is the `item_name` of the item and `suffix` is the form type: view, edit, filter, param.

For example:

```
<div class="invoices-edit">
    ...
</div>
```

is an edit form template of the **invoices** item.

For a detail before its name there should be the name of its master, separated by a hyphen:

```
<div class="invoices-invoice_table-edit">
    ...
</div>
```

If an item doesn't have a form template then the form template of its owner, if defined, will be used.

So the template

```
<div class="journals-edit">
    ...
</div>
```

will be used to create edit forms of items that **Journals** group owns and that do not have its own edit form template.

If, after searching this way, no template was found for an item, the template with the `default-suffix` class will be used to create a form.

So the template

```
<div class="default-edit">
    ...
</div>
```

will be used to create edit forms for items that have no templates defined for them and their owners.

When a new project is created the `index.html` already contains such templates.

Below is an example of default edit form template from `index.html` file:

```
<div class="default-edit">
  <div class="form-body">
    <div class="edit-body"></div>
    <div class="edit-detail"></div>
  </div>
  <div class="form-footer">
    <button type="button" id="ok-btn" class="btn expanded-btn">
      <i class="icon-ok"></i> OK<small class="muted">&nbsp; [Ctrl+Enter]</small>
    </button>
    <button type="button" id="cancel-btn" class="btn expanded-btn">
      <i class="icon-remove"></i> Cancel
    </button>
  </div>
</div>
```

There are more template examples in the Form examples section.

3.4.5 Form events

After the form is created and the HTML form template is added to the DOM, the application triggers the following form events during the life cycle of the form:

- `on_view_form_created` - the event is triggered when the form has been created but not shown yet
- `on_view_form_shown` - the event is triggered when the the form has been shown
- `on_view_form_close_query` - the event is triggered when an attempt is made to close the form
- `on_view_form_closed` - the event is triggered when the form has been closed
- `on_view_form_keydown` - the event is triggered when the keydown event occurs for the form
- `on_view_form_keyup` - the event is triggered when the keyup event occurs for the form

For other form types - edit, filter and param, replace 'view' with the form type, for example `on_edit_form_created` for edit form.

We will first explain how to use the `on_view_form_created` event.

When the user clicks on menu item the application executes the *view* method of corresponding task tree item, this method creates a form using its HTML form template and triggers first the *on_view_form_created* event of the task.

When you create a new project, the task client module already contains the code, including the *on_view_form_created* event handler. This event handler is executed each time the view form is created and defines the default behavior of view forms.

You can open the task client module to see this event handler. If you need to change the default behavior for all view forms of the project, you should do it here.

Below we describe the major steps it performs:

- Initializes the *view_form* and *table_options* that are used by some methods when view form and table are created.
- Assigns JQuery event handlers for default buttons to methods of the item, depending on the user rights. In the example below the delete button is. Initialized:

```
if (item.can_delete()) {
    item.view_form.find("#delete-btn").on('click.task', function(e) {
        e.preventDefault();
        item.delete_record();
    });
}
else {
    item.view_form.find("#delete-btn").prop("disabled", true);
}
```

- Executes the *on_view_form_created* event handler of the item group and. *on_view_form_created* of the item if they are defined:

```
if (!item.master && item.owner.on_view_form_created) {
    item.owner.on_view_form_created(item);
}

if (item.on_view_form_created) {
    item.on_view_form_created(item);
}
```

- Creates a table to display the item data and tables for details if they have been specified by calling *create_view_tables* method
- Executes *open* method, that gets the item dataset from the server.
- Finally returns true to prevent calling of the *on_view_form_created* of the owner group and the item because the were already called see the *_process_event* method below.

After we initialized buttons and before creating tables we call the *on_view_form_created* event handler of the item itself.

For example, in the client module of the tracks item of the demo app the following *on_view_form_created* event handler is defined. In it we change the height attribute of the *table_options*, create the copy of the invoice_table set its attributes and call its *create_table* method that creates a table to display its data.

```
function on_view_form_created(item) {
    item.table_options.height -= 200;
    item.invoice_table = task.invoice_table.copy();
    item.invoice_table.paginate = false;
    item.invoice_table.create_table(item.view_form.find('.view-detail'), {
        height: 200,
        summary_fields: ['date', 'total'],
```

(continues on next page)

(continued from previous page)

```
});
item.alert('Double-click the record in the bottom table to see track sales.');
```

The module also has the *on_after_scroll* event handler that will be executed when the user moves to the other track and will get the sales of this track.

This example explains the principle of form events usage.

The order of triggering of events depends on the type of event.

There are three type of

The order in which events are generated depends on the type of event.

Close query events

When user tries to close the form the *on_close_query* event is first triggered (if defined) for the item.

If the event handler returns true the application closes the form, else if the event handler returns false the application leaves the form open, otherwise the *on_close_query* event is triggered (if defined) the same way for the item group and then for the task.

For example, by default there is the *on_edit_form_close_query* event handler in the task client module:

```
function on_edit_form_close_query(item) {
    var result = true;
    if (!item.virtual_table && item.is_changing()) {
        if (item.is_modified()) {
            item.yes_no_cancel(task.language.save_changes,
                function() {
                    item.apply_record();
                },
                function() {
                    item.cancel_edit();
                }
            );
            result = false;
        }
        else {
            item.cancel_edit();
        }
    }
    return result;
}
```

This code checks whether the record has been modified and then opens “Yes No Cancel” dialog.

If we want to close the form without this dialog we can defined the following event handler in the client module of the item:

```
function on_edit_form_close_query(item) {
    item.cancel()
    return true;
}
```

Keydown, keyup events

These events are triggered the same way as Close query events, starting from the item, but if the event handler returns true, the event handlers of the group and task are not executed.

For example, by default there is the *on_edit_form_keyup* event handler in the task client module:

```
function on_edit_form_keyup(item, event) {
  if (event.keyCode === 13 && event.ctrlKey === true) {
    item.edit_form.find("#ok-btn").focus();
    item.apply_record();
  }
}
```

This code saves the changes of the record to the database table when user presses Ctrl+Enter.

Suppose we want to save the changes when user presses Enter. Then we write the following event handler in the item client module:

```
function on_edit_form_keyup(item, event) {
  if (event.keyCode === 13) {
    item.edit_form.find("#ok-btn").focus();
    item.apply_record();
    return true;
  }
}
```

In this case the event handler of the task won't be called when the user press Enter.

All other events

For other events, the event handler of the task is called first, if it doesn't return true, the event handler of the group is executed if it doesn't return true the event handler of the item is called.

This mechanism is implemented the *_process_event* method of the Item class in the *jam.js* module.

```
_process_event: function(form_type, event_type, e) {
  var event = 'on_' + form_type + '_form_' + event_type,
      can_close;
  if (event_type === 'close_query') {
    if (this[event]) {
      can_close = this[event].call(this, this);
    }
    if (!this.master && can_close === undefined && this.owner[event]) {
      can_close = this.owner[event].call(this, this);
    }
    if (can_close === undefined && this.task[event]) {
      can_close = this.task[event].call(this, this);
    }
    return can_close;
  }
  else if (event_type === 'keyup' || event_type === 'keydown') {
    if (this[event]) {
      if (this[event].call(this, this, e)) return;
    }
    if (!this.master && this.owner[event]) {
      if (this.owner[event].call(this, this, e)) return;
    }
  }
}
```

(continues on next page)

(continued from previous page)

```

    }
    if (this.task[event]) {
        if (this.task[event].call(this, this, e)) return;
    }
}
else {
    if (this.task[event]) {
        if (this.task[event].call(this, this)) return;
    }
    if (!this.master && this.owner[event]) {
        if (this.owner[event].call(this, this)) return;
    }
    if (this[event]) {
        if (this[event].call(this, this)) return;
    }
}
}
}

```

3.4.6 Form options

For each type of form an item has an attribute that controls the modal form behavior:

- *view_options*
- *edit_options*
- *filter_options*
- *param_options*

This is an object that has the following attributes, specifying parameters of the modal form:

- *width* - the width of the modal form, the default value is 560 px,
- *title* - the title of the modal form, the default value is the value of a *item_caption* attribute,
- *close_button* - if true, the close button will be created in the upper-right corner of the form, the default value is true,
- *close_caption* - if true and *close_button* is true, will display 'Close - [Esc]' near the button
- *close_on_escape* - if true, pressing on the Escape key will trigger the corresponding *close_form* method.
- *close_focusout* - if true, the corresponding *close_form* method will be called when a form loses focus
- *template_class* - if specified, the div with this class will be searched in the task *templates* attribute and used as a form html template when creating a form

The *edit_options* has a *fields* attribute, that specify a list of field names that the *create_inputs* method will use, if *fields* attribute of its *options* parameter is not specified, the default value is a list of field names set in the *Edit Form Dialog* in the Application builder.

The *view_options* has a *fields* attribute, that specify a list of field names that the *create_table* method will use, if *fields* attribute of its *options* parameter is not specified, the default value is a list of field names set in the *View Form Dialog* in the Application builder.

The width of the modal form, created in the following example, will be 700 px.

```
function on_edit_form_created(item) {  
    item.edit_options.width = 700;  
}
```

3.4.7 Data-aware controls

To create a table to display an item's dataset use *create_table* method:

```
item.create_table(item.view_form.find(".view-table"), table_options);
```

To create data controls to edit fields of the of the dataset use *create_inputs* method:

```
item.create_inputs(item.edit_form.find(".edit-body"), input_options);
```

These methods have two parameters - **container** and **options**. The first parameter is a JQuery container in which the controls will be placed. The second - options, satisfying the way the data will be displayed. For detailed information see their API reference.

The methods are usually used in the **on_view_form_created** and **on_edit_form_created** event handlers.

All visual controls (tables, inputs, checkboxes), created by this methods are data-aware. This means that they immediately reflect any changes of the item dataset.

Sometimes it is necessary to disable this interaction. To do so use the *disable_controls* and *enable_controls* methods respectively.

Videos

Data aware controls

3.5 Data programming

3.5.1 Dataset

Jam.py framework uses a dataset concept that is very close to datasets of *Embarcadero Delphi*.

Note: There are other ways to read and modify the database data. You can use the *connect* method of the task to get a connection from the connection pool and use the connection to get access to the database using Python Database API.

All items with `item_type` "item" or "table" as well as their details (see *Task tree*) can access data from associated tables from the project database and write changes to it. They all are objects of the Item class

- *Item class* (on the client)
- *Item class* (on the server)

Both of these classes have the same attributes, methods, and events associated with the data handling.

To get a dataset (a set of records) from the project dataset table, use the open method. This method, based on parameters, generates an SQL query to get a dataset.

After dataset is opened, the application can navigate it, change its records or insert new ones and write changes to the item's database table.

For example, the following functions will set *support_rep_id* field values to the values of the *id* field on the client and server respectively:

```
function set_support_id(customers) {
  customers.open();
  while (!customers.eof()) {
    customers.edit();
    customers.support_rep_id.value = customers.id.value;
    customers.post();
    customers.next();
  }
  customers.apply();
}
```

```
def set_support_id(customers):
    customers.open()
    while not customers.eof():
        customers.edit()
        customers.support_rep_id.value = customers.id.value
        customers.post()
        customers.next()
    customers.apply();
```

These functions get the **customers** item as a parameter. Then the *open* method is used to get a list of records from the customers table and each record is modified. In the end the changes are saved in the database table, using the *apply* method (see [Modifying datasets](#)).

Note: There is a shorter way to navigate a dataset (see [Navigating datasets](#)). For example, in python, the following loops are equivalent:

```
while not customers.eof():
    print customers.firstname.value
    customers.next()

for c in customers:
    print c.firstname.value
```

Videos

[Datasets](#) and [Datasets Part 2](#) demonstrate almost all methods of working with datasets on specific examples

3.5.2 Navigating datasets

Each active dataset has a cursor, or pointer, to the current row in the dataset. The current row in a dataset is the one whose values can be manipulated by *edit*, *insert*, and *delete* methods, and the one, whose field values, data-aware controls on a form currently show.

You can change the current row by moving the cursor to point at a different row. The following table lists methods you can use in application code to move to different records:

Client method	Server method	Description
<i>first</i>	<i>first</i>	Moves the cursor to the first row in an item dataset.
<i>last</i>	<i>last</i>	Moves the cursor to the last row in an item dataset.
<i>next</i>	<i>next</i>	Moves the cursor to the next row in an item dataset.
<i>prior</i>	<i>prior</i>	Moves the cursor to the previous row in an item dataset.

In addition to these methods, the following table describes two methods that provide useful information when iterating through the records in a dataset:

Client method	Server method	Description
<i>bof</i>	<i>bof</i>	If the method returns true, the cursor is at the first row in the dataset, otherwise, the cursor is not known to be at the first row in the dataset.
<i>eof</i>	<i>eof</i>	If the method returns true, the cursor is at the last row in the dataset, otherwise, the cursor is not known to be at the last row in the dataset.

Each time the cursor move to another record in the dataset the following events are triggered:

Client event	Server event	Description
<i>on_before_scroll</i>	<i>before_scroll</i>	Occurs before an application scrolls from one record to another.
<i>on_after_scroll</i>	<i>scroll_after</i>	Occurs after an application scrolls from one record to another.

Using this methods we can navigate a dataset. For example,
on the client:

```
function get_customers(customers) {
    customers.open();
    while (!customers.eof()) {
        console.log(customers.firstname.value, customers.lastname.value);
        customers.next();
    }
}
```

on the server:

```
def get_customers(customers):
    customers.open()
    while not customers.eof():
        print customers.firstname.value, customers.lastname.value
        customers.next()
```

Shorter ways to navigate dataset

There is the *each* method on the client that can be used to navigate a dataset:

For example:

```
function get_customers(customers) {
    customers.open();
```

(continues on next page)

(continued from previous page)

```

customers.each(function(c) {
    if (c.rec_no === 10) {
        return false;
    }
    console.log(c.rec_no, c.firstname.value, c.lastname.value);
});
}

```

On the server we can iterate dataset rows the following way:

```

def get_customers(customers):
    customers.open()
    for c in customers:
        if c.rec_no == 10:
            break
        print c.firstname.value, c.lastname.value

```

Both functions will output customer names for the first 10 records in the dataset.

In both cases the **c** and **customers** are pointers to the same object.

3.5.3 Modifying datasets

When an application opens an item dataset, the dataset automatically enters *browse* state. Browsing enables you to view records in a dataset, but you cannot edit records or insert new records. You mainly use *browse* state to scroll from record to record in a dataset.

For more information about scrolling from record to record, see [Navigating datasets](#).

From *browse* state all other dataset states can be set. For example, calling the *insert* or *append* methods changes its state from *browse* to *insert*.

Two methods can return a dataset to *browse* state. *Cancel* ends the current edit, insert, and returns a dataset to *browse* state. *Post* writes changes to the dataset, and if successful, also returns a dataset to *browse* state. If this operations fail, the current state remains unchanged.

To check an item dataset state use `item_state` attribute or `is_new` `is_edited` or `is_changing` methods:

Client	Server	Description
<i>item_state</i>	<i>item_state</i>	Indicates the current operating state of the item dataset.
<i>is_new</i>	<i>is_new</i>	Returns true if the item dataset is in <i>insert</i> state.
<i>is_edited</i>	<i>is_edited</i>	Returns true if the item dataset is in <i>edit</i> state.
<i>is_changing</i>	<i>is_changing</i>	Returns true if the item dataset is in <i>insert</i> or <i>edit</i> state.

You can use the following item methods to insert, update, and delete data in dataset:

Client	Server	Description
<i>edit</i>	<i>edit</i>	Puts the item dataset into edit state.
<i>append</i>	<i>append</i>	Appends a record to the end of the dataset, and puts the dataset in <i>insert</i> state.
<i>insert</i>	<i>insert</i>	Inserts a record at the beginning of the dataset, and puts the dataset in <i>insert</i> state.
<i>post</i>	<i>post</i>	Saves the new or altered record, and puts the dataset in <i>browse</i> state.
<i>cancel</i>	<i>cancel</i>	Cancels the current operation and puts the dataset in <i>browse</i> state.
<i>delete</i>	<i>delete</i>	Deletes the current record and puts the dataset in <i>browse</i> state.

All changes made to the dataset are stored in memory, the item records changes to change log. Thus, after all the changes have been made, they can be stored in the associated database table by calling the `apply` method. The `apply` method generates and executes SQL query to save changes to the database.

Client	Server	Description
<i>log_changes</i>	<i>log_changes</i>	Indicates whether to log data changes.
<i>apply</i>	<i>apply</i>	Sends all updated, inserted, and deleted records from the item dataset to the server for writing to the database.

3.5.4 Fields

All items, working with database table data have a *fields* attribute - a list of field objects, which are used to represent fields in item's table records.

Every field have the following attributes:

Client	Server	Description
<i>owner</i>	<i>owner</i>	The item that owns this field.
<i>field_name</i>	<i>field_name</i>	The name of the field that will be used in programming code to get access to the field object.
<i>field_caption</i>	<i>field_caption</i>	The name of the field that appears to users.
<i>field_type</i>	<i>field_type</i>	Type of the field, one of the following values: text , integer , float , currency , date , date-time , boolean , blob .
<i>field_size</i>	<i>field_size</i>	A size of the field with type text
<i>required</i>	<i>required</i>	Specifies whether a nonblank value for a field is required.

To get access to the item dataset data, the Field class have the following properties:

Client	Server	Description
<i>value</i>	<i>value</i>	Use this property to get or set the field's value of the current record. When reading the value is converted to the type of the field. So for fields of type integer, float and currency, if value for this field in database table record is NULL, value of this property is 0. To get unconverted value use the <i>raw_value</i> property.
<i>text</i>	<i>text</i>	Use this property to get or set the value of the field as text.
<i>lookup_value</i>	<i>lookup_value</i>	Use this property to get or set lookup value, see Lookup fields .
<i>lookup_text</i>	<i>lookup_text</i>	Use this property to get or set the lookup value of the field as text, see Lookup fields .
<i>display_text</i>	<i>display_text</i>	Represents the field's value as it is displayed in data-aware controls. When the field is a lookup field it's value is the <i>lookup_text</i> value, otherwise it is the <i>text</i> value, with regard of project locale parameters. This behavior can be overridden by the <i>on_field_get_text</i> event handler of the item that owns the field.
<i>raw_value</i>	<i>raw_value</i>	Use this property to get field value of the current record as it is stored in database. No conversion is used.

In addition every field is an attribute of the item that owns it. So, to get access to a field of an item use the following syntax: `item.field_name`

```
invoices.total.value
```

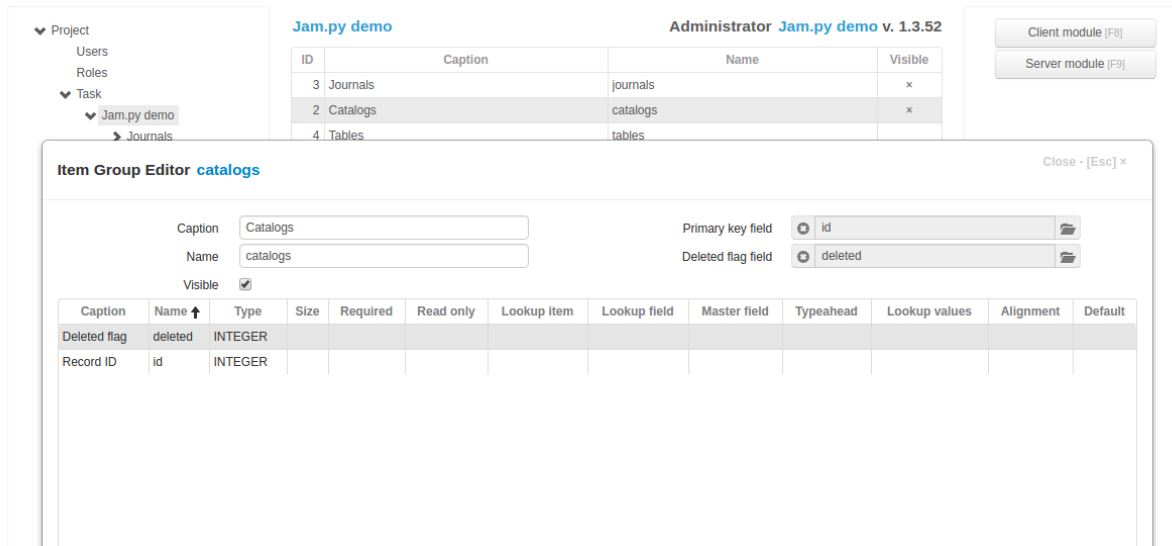
`invoices.total` is the reference to the **Total** field of the **Invoices** item and the `invoices.total.value` is the value of this field

Below are the values of the attributes of the fields of the **invoices** item in the [Demo project](#)


```
customer integer
  value: 2
  text: 2
  lookup_value: Köhler
  lookup_text: Köhler
  display_text: Leonie Köhler
firstname integer
  value: 2
  text: 2
  lookup_value: Leonie
  lookup_text: Leonie
  display_text: Leonie
billing_address integer
  value: 2
  text: 2
  lookup_value: Theodor-Heuss-Straße 34
  lookup_text: Theodor-Heuss-Straße 34
  display_text: Theodor-Heuss-Straße 34
id integer
  value: 1
  text: 1
  lookup_value: None
  lookup_text:
  display_text: 1
date date
  value: 2014-01-01
  text: 01/01/2014
  lookup_value: None
  lookup_text:
  display_text: 01/01/2014
total currency
  value: 2.08
  text: $2.08
  lookup_value: None
  lookup_text:
  display_text: $2.08
```

3.5.5 Common fields

Items that have access to the database data can have common fields. They are defined in the group they belong to:



Here two fields are defined: **id** and **deleted**.

The **id** field is set as a primary key and will store a unique identifier for each record in the database table. This value is automatically generated by the framework when inserting a new record into the table.

The **deleted** field is set as a deletion flag. When the 'Soft delete' check-box is checked in the *Item Editor Dialog*, the delete method does not erase a record physically from the table, but uses this field to mark the record as deleted. The open method takes this into account when an SQL query is generated to get records from the database table.

For detail groups two more fields could be defined — **master_id** and **master_rec_id**. They are used to link detail records to the a record in master table, see *Details*.

3.5.6 Lookup fields

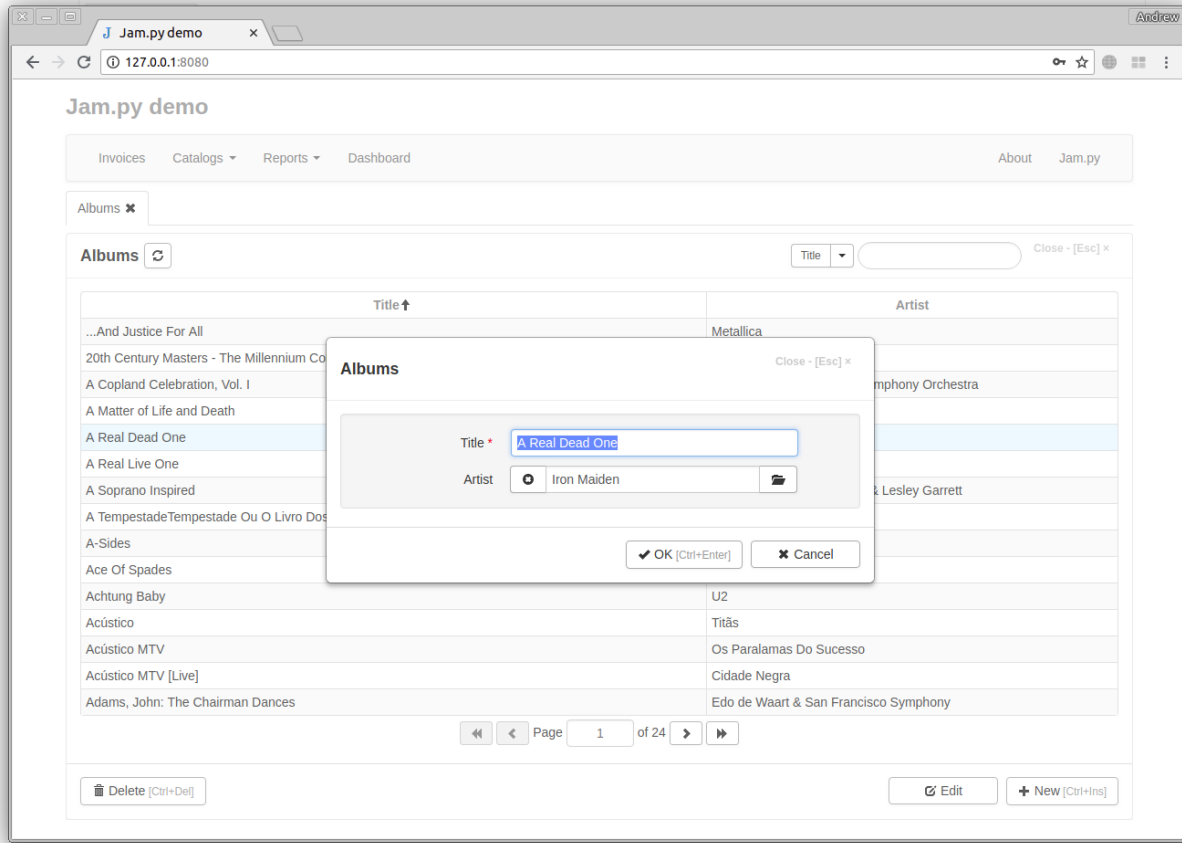
A lookup field can display a user friendly value that is bound to another value in the another table or value list. For example, the lookup field can display a customer name that is bound to a respective customer ID number in another item's table or list.

When entering a value in the lookup field the user chooses from a list of values. This can make data entry quicker and more accurate.

The two types of lookup fields that you can create are a lookup field, based on lookup item, and a value list.

Lookup item based lookup field

In the framework you can add a field to an item to look up information in another item's table. For example in the Demo application **Albums** catalog there is the **Artist** lookup field.



To set the value of the field the user must click on the button to the right of the field input and select a record from the “Artists” catalog that will appear. Then the value of this field will be the id of the record. The other way to set value of the field is to use typeahead, if **Typeahead** flag is set in the *Field Editor Dialog*:

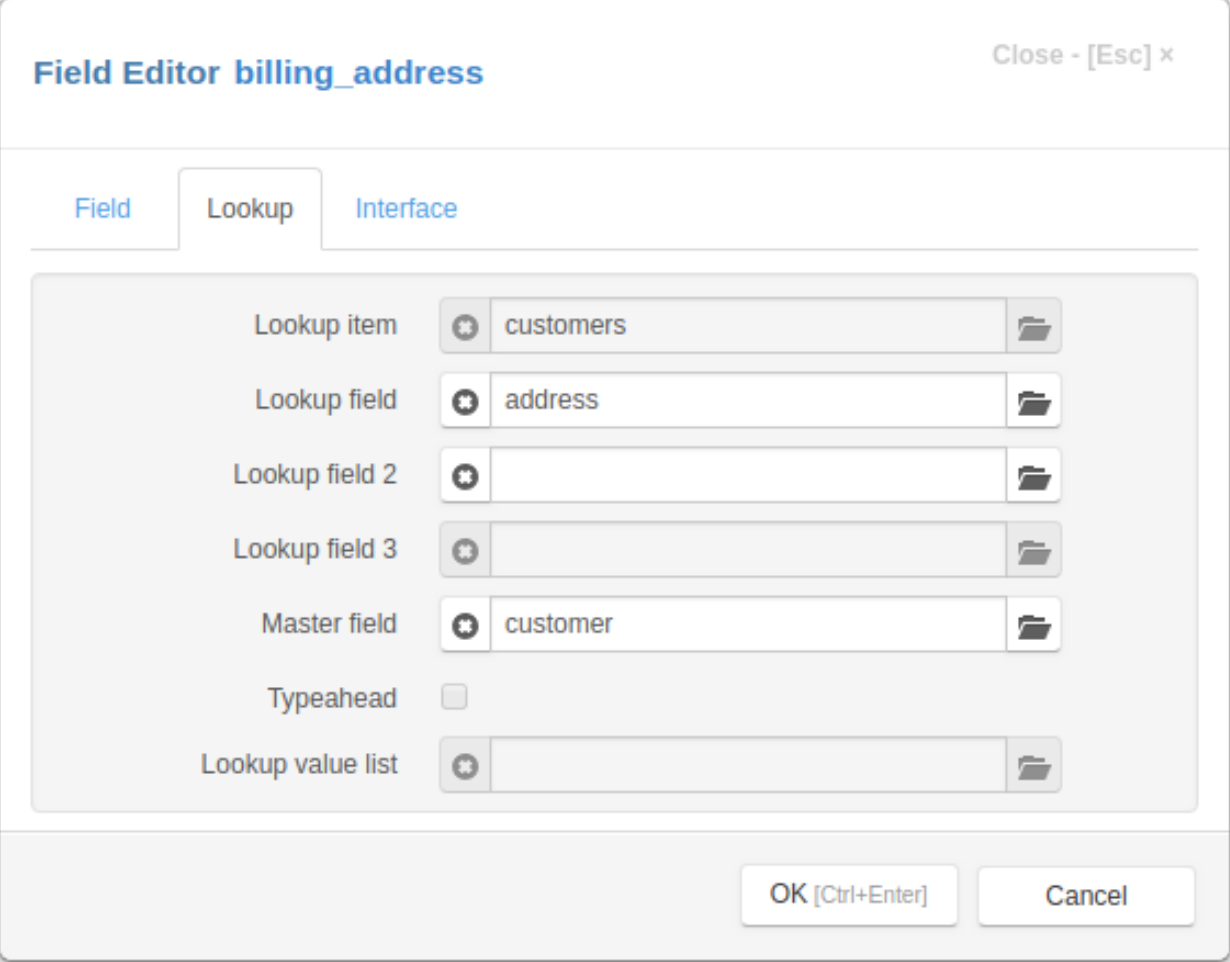
For such fields **Lookup item** and **Lookup field** must be specified in the *Field Editor Dialog*:

The SQL query that is generated on the server, when the `open` method is called and `expanded` parameter is set to `true` (default), uses `JOIN` clause to get lookup values for such fields. Thus each such field has a pair of values: the first value stores a reference to a record in the lookup item table (the value of its primary key field), and the second value have the value of the lookup field in this record.

To get access to this values use the following properties of lookup fields:

Client	Server	Description
<i>value</i>	<i>value</i>	A value, that is stored in the item table, that is a reference to a record in the lookup item table.
<i>lookup_value</i>	<i>lookup_value</i>	A value of the lookup field in the lookup item table.

Sometimes there is a need to have two or more values from the same record in the lookup item table. For example, the “Invoices” journal in Demo has several lookup fields (“Customer”, “Billing Address”, “Billing City”, and so on) that have information about a customer, all stored in one record in the “Customers” item table, describing that customer. In order to avoid creating unnecessary fields in the “Invoices” item table, storing the same reference to a record, and creating `JOIN`s for each such field, all lookup fields except “Customers” have **Master field** value pointing to the “Customers” field. These fields don’t have corresponding fields in the items’ underlying database table. Their value property is always equal to the value property of the master field and the SQL query that is generated on the server, when the `open` method is called, uses one `JOIN` clause for all this fields.



The image shows a 'Field Editor' dialog box for the 'billing_address' field. It has three tabs: 'Field', 'Lookup', and 'Interface'. The 'Lookup' tab is selected. Inside the 'Lookup' tab, there are several input fields with a folder icon to their right:

- Lookup item: customers
- Lookup field: address
- Lookup field 2: (empty)
- Lookup field 3: (empty)
- Master field: customer
- Typeahead: ☐
- Lookup value list: (empty)

At the bottom right, there are two buttons: 'OK [Ctrl+Enter]' and 'Cancel'.

When user clicks on the button to the right of the field input or uses typeahead, the application creates a copy of the lookup item of the field, sets its *lookup_field* attribute to the field, and triggers *on_field_select_value* event. Write this event handler to specify fields that will be displayed, set up filters for the lookup item, before it will be opened and displayed for a user to select a value for the field.

The lookup field in the lookup item can also be a lookup field, for example:

The screenshot shows a dialog box titled "Field Editor artist" with a "Close - [Esc] x" button in the top right corner. Below the title bar are three tabs: "Field", "Lookup", and "Interface". The "Lookup" tab is selected. The main area contains several fields for configuring a lookup:

- Lookup item:** A text box containing "tracks" with a folder icon on the right.
- Lookup field:** A text box containing "album" with a folder icon on the right.
- Lookup field 2:** A text box containing "artist" with a folder icon on the right.
- Lookup field 3:** A text box containing "name" with a folder icon on the right.
- Master field:** A text box containing "track" with a folder icon on the right.
- Typeahead:** A checkbox that is currently unchecked.
- Lookup value list:** A text box with a folder icon on the right.

At the bottom right of the dialog are two buttons: "OK [Ctrl+Enter]" and "Cancel".

To set up such a field use **Lookup field 2** and **Lookup field 3** attributes.

Value list

Sometimes a source of a lookup field can be defined as a value list. For example, a **MediaType** field in the **Tracks** catalog of the *Demo project* has a **Lookup value list** attribute set to the MediaTypes lookup list:

Field Editor media_type Close - [Esc] x

Field **Lookup** Interface

Lookup item

Lookup field

Lookup field 2

Lookup field 3

Master field

Typeahead ☒

Lookup value list

OK [Ctrl+Enter] Cancel

Use the *Lookup List Dialog* of the task to define such lookup lists.

See also

Lookup fields

Lookup lists

3.5.7 Filtering records

There are three ways to define what records an item *dataset* will get from the database table when the `open` method is called:

- to specify `where` parameter (option) of the `open` method,
- call the `set_where` method, before calling the `open` method,
- or use *filters*.

When `where` parameter is specified, it is always used even if the `set_where` method was called or item has filters whose values have been set.

When `where` parameter is omitted the parameter passed to the `set_where` method are used.

For example on the client in the following code in the first call of the `open` method the `where` option will be used to filter records, in the second call the parameters passed to `set_where` and only the third time the value of `invoicedate1` filter will be used

```
function test(invoices) {
    var date = new Date(new Date().setYear(new Date().getFullYear() - 1));

    invoices.clear_filters();
    invoices.filters.invoicedate1.value = date;

    invoices.open({where: {invoicedate__ge: date}});

    invoices.set_where({invoicedate__ge: date});
    invoices.open();

    invoices.open();
}

date = datetime.datetime.now() - datetime.timedelta(days=3*365)
```

The same code on the server looks the following way:

```
from datetime import datetime

def test(invoices):
    date = datetime.now()
    date = date.replace(year=date.year-1)

    invoices.clear_filters()
    invoices.filters.invoicedate1.value = date

    invoices.open(where={'invoicedate__ge': date})

    invoices.set_where(invoicedate__ge=date)
    invoices.open()

    invoices.open()
```

In the framework, the following symbols and corresponding constants are defined to filter records:

Filter type	Filter symbol	Constant	SQL Operator
EQ	'eq'	FILTER__EQ	EQ
NE	'ne'	FILTER__NE	NE
LT	'lt'	FILTER__LT	LT
LE	'le'	FILTER__LE	LE
GT	'gt'	FILTER__GT	GT
GE	'ge'	FILTER__GE	GE
IN	'in'	FILTER__IN	IN
NOT IN	'not_in'	FILTER__NOT_IN	NOT IN
RANGE	'range'	FILTER__RANGE	RANGE
ISNULL	'isnull'	FILTER__ISNULL	ISNULL
EXACT	'exact'	FILTER__EXACT	EXACT
CONTAINS	'contains'	FILTER__CONTAINS	CONTAINS with the “%” sign to find records where field value contains a search string
STARTWITH	'startwith'	FILTER__STARTWITH	STARTWITH with the “%” sign to find records where field value starts with a search string
ENDWITH	'endwith'	FILTER__ENDWITH	ENDWITH with the “%” sign to find records where field value ends with a search string
CONTAINS ALL	'contains_all'	FILTER__CONTAINS_ALL	CONTAINS ALL with the “%” sign to find records where field value contains all words of a search string

The where parameter of the open method is a dictionary, whose keys are the names of the fields that are followed, after double underscore, by a filter symbol. For EQ filter the filtering symbol ‘__eq’ can be omitted. For example {'id': 100} is equivalent to {'id__eq': 100}.

See also

Dataset

Filters

Client

open

set_where

Server

open

set_where

3.5.8 Filters

For each item that have access to a database table a list of filter objects can be created.

To create filters use an *Filters Dialog* of the Application builder.

Filters provide a convenient way for users to visually specify parameters of the request made by the application to the project database

Each filter has the following attributes:

- `owner` — an item that owns this filter,
- `filter_name` — the name of the filter that can be used in programming code
- `filter_caption` - the name of the filter used in the visual representation in the client application,
- `filter_type` — type of the filter, see *Filtering records*,
- `visible` — if the value of this attribute is `true`, a visual representation of this filter will be created by the *create_filter_inputs* method, when a `filters` option is not specified,
- `value` — a value of the filter,

All filters of the item are attributes of the `filters` of its object. By using `filter_name` we can get access to the filter object:

```
invoices.filters.invoicedate1.value = new Date()
```

Another way to get access to the filter is to use *filter_by_name* method:

```
invoices.filter_by_name('invoicedate').value = new Date()
```

See also

Dataset

Filtering records

Client

filters

Filter class

assign_filters

clear_filters

each_filter

filter_by_name

Server

filters

Filter class

clear_filters

filter_by_name

3.5.9 Details

Details are used in the framework to work with tabular data, pertaining to a record in an item's table.

For example, the **Invoices** journal in the Demo application has the **InvoiceTable** detail, which keeps a list of tracks in an customer's invoice.

Details and detail items share the same underlying database table.

To create a detail, you must first create a detail item (select Details group of the project tree and click on New button) and then use the *Details Dialog* (select item in the project tree and click on Details button) to add a detail to an item.

For example the following code

```
def on_created(task):
    task.invoice_table.open()
    print task.invoice_table.record_count()

    task.invoices.open(limit=1)
    task.invoices.invoice_table.open()
    print task.invoices.invoice_table.record_count()
```

will print:

```
2259
6
```

Details have two *common fields* - `master_id` and `master_rec_id`, that are used to store information about the ID of the master (each item have its own unique ID) and the value of the primary field of the record of its master. This way each table can be linked to several items. As well as each item can have several details. To get access to details of an item use its `details` attribute. To get access to the master of the detail use its `master` attribute.

Detail class, used to create details, is an ancestor of the Item class and inherits all its attributes, methods and events.

Note: The `apply` method of the Detail class does nothing. To write changes made to a detail use `apply` method of its master.

To work with a detail its master must be active

To make any changes to a detail its master must be in an edit or insert mode

Examples

In this example from the client module of the **Invoices** item of *Demo project*, the **Invoice_table** detail is reopened every time the cursor of its master moves to another record.

```
var ScrollTimeout;

function on_after_scroll(item) {
    clearTimeout(ScrollTimeout);
    ScrollTimeout = setTimeout(
        function() {
            item.invoice_table.open(function() {});
        },
        100
    );
}
```

And just as an example:

```
from datetime import datetime, timedelta

def on_created(task):
    invoices = task.invoices.copy()
    invoices.set_where(invoicedate__gt=datetime.now()-timedelta(days=1))
    invoices.open()
    for i in invoices:
        i.invoice_table.open()
        i.edit()
        for t in i.invoice_table:
            t.edit()
            t.sales_id.value = '101010'
            t.post()
        i.post()
    invoices.apply()
```

The same code on the client will be as follows:

```
function on_page_loaded(task) {
    var date = new Date(),
        invoices = task.invoices.copy();

    invoices.set_where({invoicedate__gt: date.setDate(date.getDate() - 1)});
    invoices.open();
    invoices.each(function(i) {
        i.invoice_table.open();
        i.edit();
        i.invoice_table.each(function(t) {
            t.edit();
            t.sales_id.value = '101010';
            t.post();
        });
        i.post();
    });
    invoices.apply();
}
```

3.6 Server side programming

In most cases, the client sends a request to the server when following methods of an item are executed:

- *open*
- *apply*
- *print*
- *server*

In these cases the client sends to the server the *ID* of the item's task, the *ID* of the item, the type of the request and its parameters.

The server on receiving the request, based on passed IDs, finds the task (it can be Project task or Application builder task) and the item on the server, executes the corresponding method with passed parameters and returns the result of the execution to the client. The server method can trigger events that can modify its default behavior.

Every item of the task tree have the *environ* and *session* attributes that store context of the current request.

The most common server events are:

- *on_created* - The event is triggered by the task when it has just been created by the server application. It can be used to initialize the project.
- *on_apply events* - These events are triggered when the *apply* method of the item is called on the *client* or the *server*
- *on_open_events* - These events are triggered when the *open* method of the item is called on the *client* or the *server*
- *on_generate* - “The event is triggered when the *print* method of a report is called on the client.

Note: Note that the task tree on the server is immutable, you can not change the attributes of the items in the task tree.

You must use the *copy* method to create a copy of an item. This copy is an exact copy of an item at the time of creating of the task tree. It is not added to the *task tree* and will be destroyed by Python garbage collector when no longer needed.

3.6.1 on_apply events

When the *apply* method of the item is called on the *client* or the *server*, the server application, by default, generates SQL query, based on changes made to the dataset and executes it.

This behavior can be changed by writing an *on_apply* event handler in the item server module.

Sometimes it becomes necessary to execute some code, when changes are saved, for all items. In this case the *on_apply* event handler of the task (declared in the task server module) can be used.

The following code describes how these events are handled:

```
#...
result = None
if self.task.on_apply:
    result = self.task.on_apply(self, delta, params, connection)
if result is None and self.on_apply:
    result = self.on_apply(self, delta, params, connection)
if result is None:
    result = self.apply_delta(delta, params, connection)
#...
return result
```

It checks if the task has an *on_apply* event handler. If the *on_apply* event handler is declared in the task server module, it is executed.

If the *on_apply* event handler of the task is not declared or the result of the event handler returns *None*, the method checks whether the item has an *on_apply* event handler. If it is declared in the item server module, it is executed.

If the result returned by the item event handler is *None*, the *apply_delta* method of the item is called that generates SQL query, execute it and returns the result

Example

Here is an example how on_apply can be used

3.6.2 on_open_events

When the open method of the item is called on the *client* or the *server*, the server application executes the following code:

```
result = None
if self.task.on_open:
    result = self.task.on_open(self, params)
if result is None and self.on_open:
    result = self.on_open(self, params)
if result is None:
    result = self.execute_open(params)
```

It checks if the task has an `on_open` event handler. If the `on_open` event handler is declared in the task server module, it is executed.

If the `on_open` event handler of the task is not declared or the result of the event handler returns `None`, the method checks whether the item has an *on_open* event handler. If it is declared in the item server module, it is executed.

If the result returned by the item event handler is `None`, the `execute_open` method of the item is called that generates SQL query, execute it and returns the result

Example

Here is an example how on_open can be used

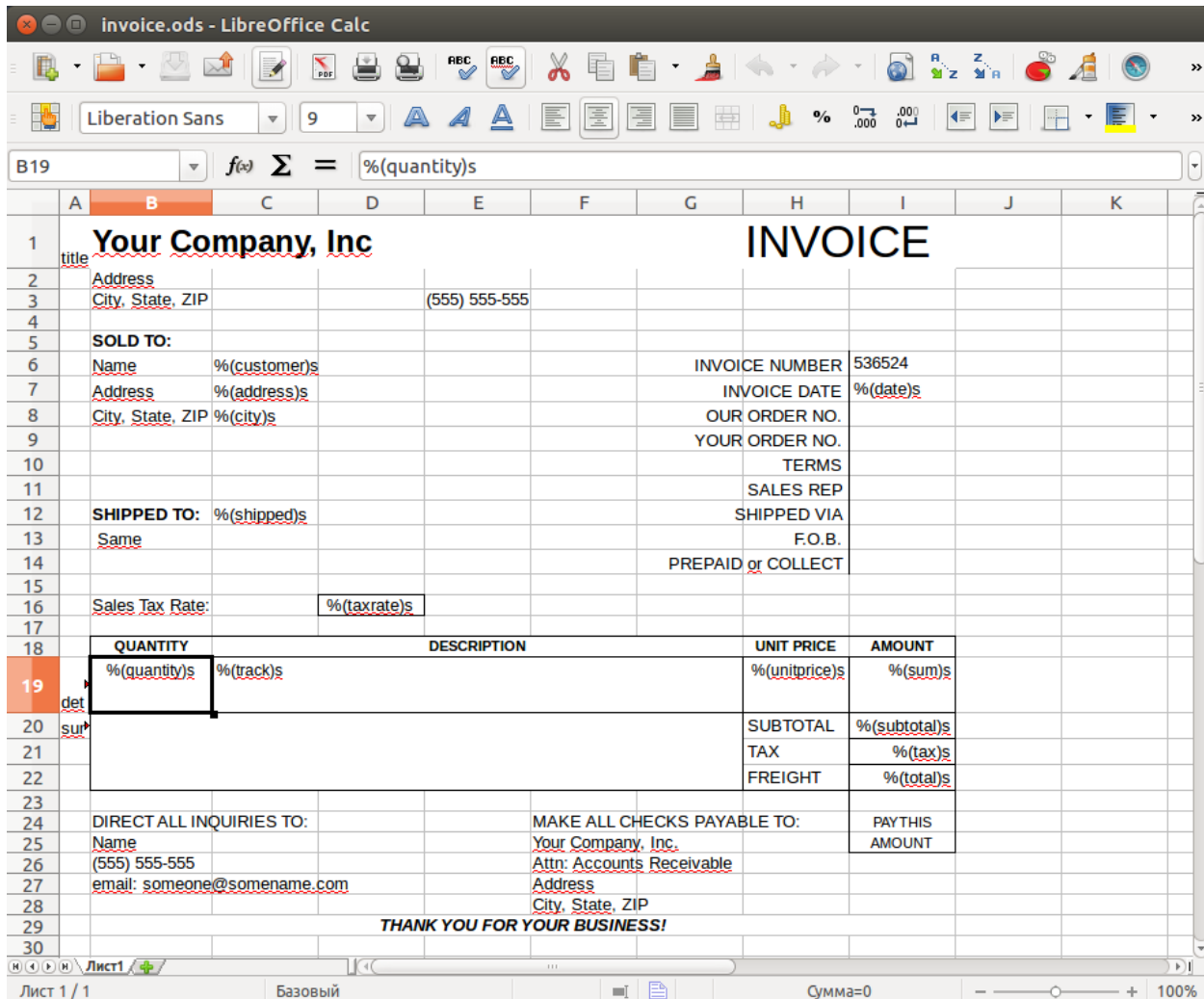
3.7 Programming reports

3.7.1 Report templates

To create a report, you must first prepare a report template in LibreOffice Calc.

The template files are located in the report folder of the project directory.

The following figure shows a template of the Invoice report.



Reports in Jam.py are band-oriented.

Each report template is divided into bands. To set bands use the leftmost column of a template spreadsheet.

In the Invoice report template there are three bands: **title**, **detail** and **summary**.

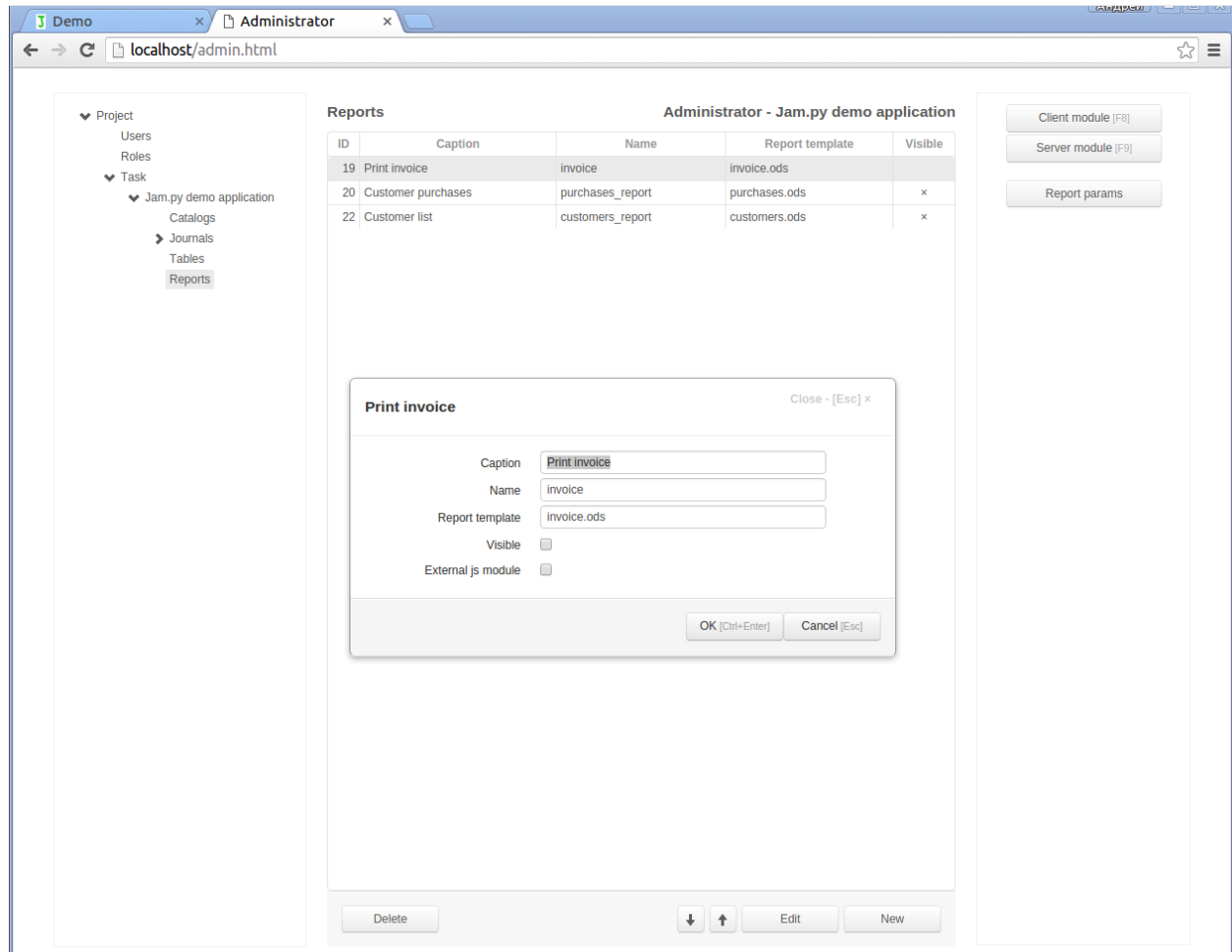
In addition, templates can have programmable cells.

For example, in the template of Invoice report the I7 cell contains the text `%(date)s`.

Programmable cell begins with `%`, then follows the name of the cell in the parenthesis which is followed by character `s`.

3.7.2 Creating a report

To add a new report to Jam.py project, choose the Reports node in the project tree, then click the New button and fill in the caption, name and the template file name of the report.



If a **visible** checkbox is set, the default code adds the report to the **Reports** menu of the project.

3.7.3 Report parameters

You can specify the parameters of the report. For example, the **Customer purchases** report of the Demo project have three parameters.

Report params Close - [Esc] x

Caption	Name	Type	Lookup item	Lookup field	Required	Visible	Align.
Customer	customer	INTEGER	Customers	lastname		x	ALIGN_LEFT
Period range from	invoicedate1	DATE			x	x	ALIGN_CENTER
to	invoicedate2	DATE			x	x	ALIGN_CENTER

Delete

↓↑

Edit

New

To add or change a report parameter click **Report params** button in the left panel of the Application builder. A form will appear displaying the list of existing parameters. Then click New or Edit button of the form to add or change the parameter.

Report params Close - [Esc] x

Caption

Name

Type

Lookup item

Lookup field

Required ☒

Visible ☒

Align.

In the dialog box fill in:

- **Caption** - the name of the parameter that appears to users
- **Name** - the name of the parameter will be used in programming code to get access to the parameter object.
- **Type** - the data type of the parameter
- **Visible** - the client application creates a form to specify the parameters before printing the report. If this checkbox is checked, the input element for this parameter will appear in the form
- **Required** - if this checkbox is checked and **Visible** attribute is set, the client application will require a users to specify the parameter value before printing the report
- **Align** - specifies how a value of the parameter will be aligned in the input element

You can create a lookup parameter, For example, the **Customer purchases** report has a **Customer** parameter that can be selected from **Customers** catalog:

Report params Close - [Esc] x

Caption:

Name:

Type:

Lookup item:

Lookup field:

Required: ☐

Visible: ☒

Align.:

In this case you should specify

- **Lookup item** - the item to select the parameter value from
- **Lookup field** - the field in the lookup item

Form for setting the parameters of **Customer purchases** report is as follows:

Jam.py demo application

Journals Reports Catalogs About

Customer purchases Customer list

Customer purchases Close - [Esc] x

Customer:

Period range from:

to:

pdf

3.7.4 Client-side report programming

To print a report on the client use the *print* method.

As a result of calling this function, the client calls *create_param_form* method to create a form for editing the report parameters, based on the html template defined in the index.html file (see *Forms*).

This method, after creating the form, triggers the following events:

- *on_param_form_created* of the task.
- *on_param_form_created* of the report group that owns the report, if one is defined
- *on_param_form_created* of the report, if one is defined.

The default code has the *on_param_form_created* event handler, defined for the task. In this event, the click on the **Print** button is connected to the report's *process_report* method.

```
function on_param_form_created(item) {
    item.create_param_inputs(item.param_form.find(".edit-body"));
    item.param_form.find("#ok-btn").on('click.task', function() {
        item.process_report()
    });
    item.param_form.find("#cancel-btn").on('click.task', function() {
        item.close_param_form()
    });
}
```

In its turn the *process_report* method triggers

- *on_before_print_report* event handler of the report group
- *on_before_print_report* event handler of the report

In this event handlers developer can define some common (report group event handler) or specific (report event handler) attributes of the report.

For example, in the default code, there is the *on_before_print_report* event handler of the report group, in which report's *extension* attribute is defined:

```
function on_before_print_report(report) {
    var select;
    report.extension = 'pdf';
    if (report.param_form) {
        select = report.param_form.find('select');
        if (select && select.val()) {
            report.extension = select.val();
        }
    }
}
```

In the following event handler, defined in the client module of the **invoice** report of the Demo application, the value of the report *id* parameter is set:

```
function on_before_print_report(report) {
    report.id.value = report.task.invoices.id.value;
}
```

After that the *process_report* method sends asynchronous request to the server to generate the report (see *Server-side programming*).

The server returns to the method an url to a file with generated report.

The method then checks if the `on_open_report` event handler of the report group is defined. If this events handler if defined calls it, otherwise checks the `on_open_report` of the report. If it is defined then calls it.

If none of this events are defined, it (depending on the report `extension` attribute) opens the report in the browser or saves it to disc.

3.7.5 Server-side report programming

When a server gets a request from a client to generate report, it first of all creates a copy of the report and then this copy calls the `generate` method.

This method triggers the `on_before_generate` event. In this event handler developer should write a code that generates the content of the report.

For example for the **invoice** report of the Demo application this event is as follows:

```
def on_generate(report):
    invoices = report.task.invoices.copy()
    invoices.set_where(id=report.id.value)
    invoices.open()

    customer = invoices.firstname.display_text + ' ' + invoices.customer.display_text
    address = invoices.billing_address.display_text
    city = invoices.billing_city.display_text + ' ' + invoices.billing_state.display_
↪text + ' ' + \
        invoices.billing_country.display_text
    date = invoices.invoice_date.display_text
    shipped = invoices.billing_address.display_text + ' ' + invoices.billing_city.
↪display_text + ' ' + \
        invoices.billing_state.display_text + ' ' + invoices.billing_country.display_
↪text
    taxrate = invoices.taxrate.display_text
    report.print_band('title', locals())

    tracks = invoices.invoice_table
    tracks.open()
    for t in tracks:
        quantity = t.quantity.display_text
        track = t.track.display_text
        unitprice = t.unitprice.display_text
        sum = t.amount.display_text
        report.print_band('detail', locals())

    subtotal = invoices.subtotal.display_text
    tax = invoices.tax.display_text
    total = invoices.total.display_text
    report.print_band('summary', locals())
```

First, we use the `copy` method to create a copy of the invoices journal.

```
invoices = report.task.invoices.copy()
```

We create the copy because multiple users can simultaneously generate the same report in parallel threads.

Then we call the `set_where` method of the copy:

```
invoices.set_where(id=report.id.value)
```

where `report.id.value` is report id parameter, the value of which we set in the *on_before_print_report* event handler on the client and which is equal to the current **id** field value of the **invoice** journal.

Then, using the *open* method, we obtain the records on the server. After that the *print_band* method is used to print title band:

```
report.print_band('title', locals())
```

But before that we assign values to four local variables: customer, address, city and date that correspond to programmable cells in the title band in the report template.

Then the same way we generate detail and summary bands.

When the report is generated and the value of report *extension* attribute, set on the client, is not equals 'pdf' the server converts the ods file using **LibreOffice**.

Once the report is generated it is stored in a report folder of the static directory and the server sends the client the report file url.

4.1 What is the difference between catalogs and journals

When a new project is created, its *task tree* has the following groups: **Catalogs**, **Journals**, **Details** and **Reports**.

Catalogs and **Journals** belong to the Item Group type and have the same functional purpose. See *Groups*.

We created them to distinguish between two types of data items:

- data items that contain information of catalog type such as customers, organizations, tracks, etc. - **Catalogs**
- data items that store information about events recorded in some documents, such as invoices, purchase orders, etc. - **Journals**

4.2 Howto upgrade an already created project to a new version of jam.py?

To upgrade an existing project to a new package you must update the package.

You can do it using pip.

If you're using Linux, Mac OS X or some other flavor of Unix, enter the command:

```
sudo pip install --upgrade jam.py
```

If you're using Windows, start a command shell with administrator privileges and run the command

```
pip install --upgrade jam.py
```

4.3 What are foreign keys used for?

Foreign keys that you can create in the Application Builder prevent deletion of a record in the lookup table if a reference to it is stored in the lookup field.

For example, when a foreign key is created on the “Customer” field for “Invoices” item, user won’t be able to delete a customer in “Customers” catalog if a reference to it is stored in “Invoices”.

The soft delete attribute of the lookup item must be set to false (see *Item Editor Dialog*) for the lookup field to appear in the *Foreign Keys Dialog*

4.4 Can I use other libraries in my application

You can add javascript libraries to use them for programming on the client side.

It is better to place them in the *js* folders of the *static* directory of the project. And refer to them using the *src* attribute in the `<script>` tag of the *Index.html* file.

For example, *Demo project* uses Chart.js library to create a dashboard:

```
<script src="/static/js/Chart.min.js"></script>
```

On the server side you can import python libraries to your modules.

For exapmple the mail item server module import smtplib library to send emails:

```
import smtplib
```

4.5 When printing a report I get an ods file instead of pdf

When a report is generated the server application first creates an ods file.

If *extension* attribute of the report is set to ‘pdf’ or any other format except ‘ods’, the application first creates an ods file and then uses LibreOffice in “headless” mode to convert the ods file to that format.

If LibreOffice is currently running on the server this conversion may not happen. You must close LibreOffice on the server for the conversion to take place.

Here is a useful code that you can use in your applications:

5.1 How to install Jam.py on Windows

Adapted from Django Docs

The below document is adopted from [Django Docs](#).

This document will guide you through installing Python 3.x and Jam.py on Windows. It also provides instructions for setting up a virtual environment, which makes it easier to work on Python projects. This is meant as a beginner's guide for users working on Jam.py projects and does not reflect how Jam.py should be installed when developing patches for Jam.py itself.

The steps in this guide have been tested with Windows 10. In other versions, the steps would be similar. You will need to be familiar with using the Windows command prompt.

5.1.1 Install Python

Jam.py is a Python web framework, thus requiring Python to be installed on your machine. At the time of writing, Python 3.8 is the latest version.

To install Python on your machine go to <https://www.python.org/downloads/>. The website should offer you a download button for the latest Python version. Download the executable installer and run it. Check the boxes next to “Install launcher for all users (recommended)” then click “Install Now”.

After installation, open the command prompt and check that the Python version matches the version you installed by executing:

```
...\> py --version
```

5.1.2 About pip

`pip` is a package manager for Python and is included by default with the Python installer. It helps to install and uninstall Python packages (such as Jam.py!). For the rest of the installation, we'll use `pip` to install Python packages from the command line.

5.1.3 Setting up a virtual environment

It is best practice to provide a dedicated environment for each Jam.py project you create. There are many options to manage environments and packages within the Python ecosystem, some of which are recommended in the [Python documentation](#).

To create a virtual environment for your project, open a new command prompt, navigate to the folder where you want to create your project and then enter the following:

```
...\> py -m venv project-name
```

This will create a folder called 'project-name' if it does not already exist and set up the virtual environment. To activate the environment, run:

```
...\> project-name\Scripts\activate.bat
```

The virtual environment will be activated and you'll see "(project-name)" next to the command prompt to designate that. Each time you start a new command prompt, you'll need to activate the environment again.

5.1.4 Install Jam.py

Jam.py can be installed easily using `pip` within your virtual environment.

In the command prompt, ensure your virtual environment is active, and execute the following command:

```
...\> py -m pip install jam.py
```

This will download and install the latest Jam.py release.

After the installation has completed, you can verify your Jam.py installation by executing `pip list` in the command prompt.

5.1.5 Common pitfalls

- If you are connecting to the internet behind a proxy, there might be problems in running the command `py -m pip install Jam.py`. Set the environment variables for proxy configuration in the command prompt as follows:

```
...\> set http_proxy=http://username:password@proxyserver:proxyport  
...\> set https_proxy=https://username:password@proxyserver:proxyport
```

- If your Administrator prohibited setting up a virtual environment, it is still possible to install Jam.py as follows:

```
...\> python -m pip install jam.py
```

This will download and install the latest Jam.py release.

After the installation has completed, you can verify your Jam.py installation by executing `pip list` in the command prompt.

However, running `jam-project.py` will fail since it is not in the path. Check the installation folder:

```
...> python -m site --user-site
```

The output might be similar to below:

```
C:\Users\youruser\AppData\Local\Packages\PythonSoftwareFoundation.Python.3.9_
  ↳qbz5n2kfra8p0\LocalCache\local-packages\Python39\site-packages
```

Replace `site-packages` at the end of above line with `Scripts`:

```
...> dir C:\Users\youruser\AppData\Local\Packages\PythonSoftwareFoundation.
  ↳Python.3.9_qbz5n2kfra8p0\LocalCache\local-packages\Python39\Scripts
```

The output might be similar to below:

```
...> Directory of
  ↳C:\Users\youruser\AppData\Local\Packages\PythonSoftwareFoundation.Python.3.9_
  ↳qbz5n2kfra8p0\LocalCache\local-packages\Python39\Scripts

13/04/2023 02:59 PM <DIR> .
13/04/2023 02:59 PM <DIR> ..
13/04/2023 02:59 PM      1,087 jam-project.py
                1 File(s)      1,087 bytes
                2 Dir(s) 177,027,321,856 bytes free
```

Create the new folder somewhere and run `jam-project` from from it:

```
...> python C:\Users\youruser\AppData\Local\Packages\PythonSoftwareFoundation.
  ↳Python.3.9_qbz5n2kfra8p0\LocalCache\local-packages\Python39\Scripts\jam-project.
  ↳py
```

Run the new project:

```
...> python server.py
```

5.2 How to migrate development to production

Migrating development to production is very simple in Jam.py due to the ability to export and import its metadata.

To understand the concept of metadata and the process of exporting and importing metadata, please read the topic [Export/import metadata](#). The process of importing metadata depends on the type of project database.

5.2.1 New project migration

- Create an empty database in the production environment
- Run `jam-project.py` script to create a new project
- Set up the server. See
 - *Jam.py deployment with Apache and mod_wsgi*,
 - *How to deploy*.
- In the browser start the Application Builder and finish the creation of the project with an empty database.

- open [Parameters](#) dialog to set up the project. Setup the following parameters:
 - **Production** to true
 - **Safe mode**
 - **Debugging** to false
- Export the metadata of the development project to a zip file in the Application Builder by clicking the [Export](#) button.
- Import the metadata to the new project.

Note: For projects with **SQLite** database you can simply copy the development project folder to the production environment.

5.2.2 Existing project migration

- Export the metadata of the development project to a zip file.
- Import the metadata to the production project.

Note: For **SQLite** database, Jam.py doesn't support importing of metadata into an existing project (project with tables in the database). You can only import metadata into a new project.

5.2.3 Importing metadata with server shutdown

Stop the server and copy the metadata zip file to *migration* folder in the project directory. If the folder doesn't exist, create it.

Start the server. The web application, while initializing itself, will import the metadata file. You can see the information on how the file was imported in the log file in the *logs* folder of the project directory. If the import is successful, the zip file will be deleted.

5.2.4 Importing metadata without server shutdown

Click the [Import](#) button in the Application Builder.

Note: By default the web application in the process that imports the metadata waits for 5 minutes or until all previous request to the application in **this process** will be processed before it starts to change the database. For projects that run on multiple processes you can set the **Import delay** parameter in the [Parameters](#) to delay the change the database or use Importing metadata with server shutdown.

5.3 How to migrate to another database

You can migrate your data to another database.

For example, you developed your project with SQLite database and want to move to Postgress.

To do this, follow these steps:

1. Create an empty Postgress database
2. Create a new project with this database
3. Export the metadata of the SQLite project to a zip file in the Application Builder by clicking the *Export* button.
4. Import the metadata to the new project. The web application with create database structures in the Postgress database.
5. copy data from SQLite to Postgress database using the *copy_database* method of the task:

- create in the sever module of the task the following function:

```
from jam.db.db_modules import SQLITE

def copy_db(task):
    task.copy_database(SQLITE, '/home/work/demo/demo.sqlite')
```

- then you can execute it one of the following ways:

- call this function in the *on_created* event handler:

```
from jam.db.db_modules import SQLITE

def copy_db(task):
    task.copy_database(SQLITE, '/home/work/demo/demo.sqlite')

def on_created(task):
    copy_db(task)
```

- create a button in some form and use the task *server* method to execute it

```
function on_view_form_created(item) {
    item.add_view_button('Copy DB').click(function() {
        task.server('copy_db')
    });
}
```

- or run from from debbuging console of the browser:

```
task.server('copy_db')
```

6. Remove the code that was used.

Note: You can not migrate to SQLite database of the current database has foreign keys

5.4 How to deploy

5.4.1 How to deploy project on PythonAnywhere

- Use pip to install Jam.py. To do this, open the bash console and run the following command (for Python 3.7):

```
pip3.7 install --user jam.py
```

- Create a zip archive of your project folder, upload the archive in the **Files** tab and unzip it.

We assume that you are registered as *username* and your project is now located in the */home/username/project_folder* directory.

- Open the **Web** Tab. Add a new web app. In the Code section specify
 - Source code: */home/username/project_folder*
 - Working directory: */home/username/project_folder*

In the WSGI configuration file: */var/www/username_pythonanywhere_com_wsgi.py* file add the following code

```
import os
import sys

path = '/home/username/project_folder'
if path not in sys.path:
    sys.path.append(path)

from jam.wsgi import create_application
application = create_application(path)
```

- Reload the server.

5.4.2 A step-by-step guide to deploy a Jam.py on the AWS

This is adapted from https://devops.profitbricks.com/tutorials/install-and-configure-mod_wsgi-on-ubuntu-1604-1/

I hope someone finds it useful.

- Create an AWS account and login
- Go to EC2, create an instance (in this case an Ubuntu 16.04 t2.micro)
- Download the private key when prompted
- Convert pem to ppk using Puttygen (see: <https://stackoverflow.com/questions/3190667/convert-pem-to-ppk-file-format>)
- Get EC2 instance public DNS from AWS dashboard
- SSH into EC2 instance using Putty (pointed to the Public DNS and your ppk)
- Username is ubuntu
- Refresh package library:

```
sudo apt-get update
```

- Install pip:

```
sudo apt-get install python3-pip
```

- Install jam.py:

```
sudo pip3 install jam.py
```

- Install Apache:

```
sudo apt-get install apache2 apache2-utils libexpat1 ssl-cert
```

- Install mod-wsgi:

```
sudo apt-get install libapache2-mod-wsgi-py3
```

- Restart Apache:

```
sudo /etc/init.d/apache2 restart
```

- Move here:

```
cd /var/www/html/
```

- Create directory:

```
sudo mkdir [appname]
```

- Move here:

```
cd [appname]
```

- Create app:

```
sudo jam-project.py
```

- Check it's there:

```
ls
```

- Create the config:

```
sudo nano /etc/apache2/conf-available/wsgi.conf
```

- Paste the following

```
WSGIScriptAlias / /var/www/html/[appname]/wsgi.py
WSGIProxyPath /var/www/html/[appname]

<Directory /var/www/html/[appname]>
    <Files wsgi.py>
        Require all granted
    </Files>
</Directory>

Alias /static/ /var/www/html/[appname]/static/

<Directory /var/www/html/[appname]/static>
    Require all granted
</Directory>
```

- Exit and save
- Give file permissions to apache:

```
sudo chmod 777 /var/www/html/[appname]
```

- Give ownership to apache:

```
sudo chown -R www-data:www-data /var/www
```

- Enable wsgi:

```
sudo a2enconf wsgi
```

- Restart apache:

```
sudo /etc/init.d/apache2 restart
```

- Create security group on AWS to allow you to connect HTTP on port 80
- Assign instance to security group
- Test
- If it's not working, check the error logs to see what's going on:

```
nano /var/log/apache2/error.log
```

This was initially published by Simon Cox on <https://groups.google.com/forum/#!msg/jam-py/Zv5JfkLRFy4/22tolZ-hAQAJ>

5.4.3 How to deploy jam-py app at Linux Apache http server?

So basically deploying straight into the ie an cloud server with open 22, 80 and 443 port. Prerequisite is a signed certificate for the DNS server name (YOUR_SERVER DNS entry from below). One can use a self signed, etc, not covering those. Also, Python installed and sudo access (or root for Linux). I have no idea at all about the MS Servers, sorry.

The App is in read only mode. You can access admin.html page, but can't change anything. Took me some fiddling with Google Cloud server, this is a micro Ubuntu instance, plain apache2 install with apt-get.

- Install wsgi module for Apache :

```
apt-get install libapache2-mod-wsgi
```

- Enable ssl, wsgi module for apache:

```
a2enmod ssl wsgi
```

- Create a custom file for jam-py app, ie /etc/apache2/sites-available/test.conf, for example (still wip):

```
<IfModule mod_ssl.c>
<VirtualHost YOUR_IP:443>
    ServerName YOUR_SERVER
    ServerAlias
    ServerAdmin YOUR_EMAIL
    ErrorLog ${APACHE_LOG_DIR}/test-error-sec.log
    CustomLog ${APACHE_LOG_DIR}/test-access-sec.log combined

    #below is for cx_Oracle
    SetEnv LD_LIBRARY_PATH /u01/app/oracle/product/11.2.0/xe/lib
    SetEnv ORACLE_SID XE
    SetEnv ORACLE_HOME /u01/app/oracle/product/11.2.0/xe
    #finish cx_Oracle

    DocumentRoot /var/www/html/simpleassets

    SSLEngine on
    SSLCertificateFile "/etc/ssl/private/your.crt"
```

(continues on next page)

(continued from previous page)

```

SSLCertificateKeyFile  "/etc/ssl/private/your.key"
SSLCertificateChainFile "/etc/ssl/private/your_chain.crt"
SSLCACertificateFile   "/etc/ssl/private/your_CA.crt"

WSGIDaemonProcess  web user=www-data group=www-data processes=1 threads=5
WSGIScriptAlias / /var/www/html/simpleassets/wsgi.py

<Directory /var/www/html/simpleassets>
    Options +ExecCGI
    SetHandler wsgi-script
    AddHandler wsgi-script .py

    Order deny,allow
    Allow from all
    Require all granted

    <Files wsgi.py>
        Order deny,allow
        Allow from all

        # comment the following for ubuntu <13
        Require all granted
    </Files>
</Directory>

<Directory /var/www/html/simpleassets/static>
    # comment the following for ubuntu < 13
    Require all granted
</Directory>
</VirtualHost>
</IfModule>

```

The above file is using signed certificate your.crt with your.key, and CA, chain file obtained from CA. Please review resources on the net about certificates and the dns. You'll need to obtain and copy those files in /etc/ssl/private folder. Change YOUR_xyz with your preference.

The /var/www/html is the default Ubuntu folder for serving web pages.

- Install jam-py as usual.

I created the /var/www/html/simpleassets folder where unzipped jam-py SimpleAssets project. Follow procedure explained there how to deploy these:

Basically, Export your project, save the zip file and copy it to your web hosting server desired folder. Copy admin.sqlite and your database as well (providing you're using sqlite3 database). If using some other database ie mysql, you'll need to export/import the database.

- Enable test.conf (the above file name with no extension):

```
a2ensite test; systemctl restart apache2
```

That is it. At the moment, I've left port 80 as is, and jam-py is running only on https port. To debug problems, I would start with SeLinux or apparmor. With Ubuntu this might help:

```
sudo /etc/init.d/apparmor stop
```

Now, here is the question of how to run TWO jam-py instances on one https server?

One possible answer to this problem is the DNS. You might decide to set your DNS to ie second_instance.YOUR_SERVER name (the above live example would be jam2.research...).

So the above test.conf file would be almost the same except YOUR_SERVER is now called second_instance.YOUR_SERVER

The /etc/apache2/sites-available/test3.conf file:

```
<IfModule mod_ssl.c>
<VirtualHost YOUR_IP:443>
    ServerName second_instance.YOUR_SERVER
    ServerAlias
    ServerAdmin YOUR_EMAIL
    ErrorLog ${APACHE_LOG_DIR}/test3-error-sec.log
    CustomLog ${APACHE_LOG_DIR}/test3-access-sec.log combined
    #below is for cx_Oracle
    SetEnv LD_LIBRARY_PATH /u01/app/oracle/product/11.2.0/xe/lib
    SetEnv ORACLE_SID XE
    SetEnv ORACLE_HOME /u01/app/oracle/product/11.2.0/xe
    #finish cx_Oracle
    DocumentRoot /var/www/html/simpleassets3
    SSLEngine on
    SSLCertificateFile "/etc/ssl/private/your.crt"
    SSLCertificateKeyFile "/etc/ssl/private/your.key"
    SSLCertificateChainFile "/etc/ssl/private/your_chain.crt"
    SSLCACertificateFile "/etc/ssl/private/your_CA.crt"

    WSGIDaemonProcess assets3 user=www-data group=www-data processes=1 threads=5
    WSGIScriptAlias / /var/www/html/simpleassets3/wsgi.py

    <Directory /var/www/html/simpleassets3>
        Options +ExecCGI
        SetHandler wsgi-script
        AddHandler wsgi-script .py

        Order deny,allow
        Allow from all
        Require all granted

    <Files wsgi.py>
        Order deny,allow
        Allow from all

        # comment the following for ubuntu <13
        Require all granted
    </Files>
</Directory>

<Directory /var/www/html/simpleassets3/static>
    # comment the following for ubuntu < 13
    Require all granted
</Directory>
</VirtualHost>
</IfModule>
```

The jam-py application second_instance lives now in ie /var/www/html/simpleassets3, and WSGIDaemonProcess is adjusted to new daemon, called assets3. Everything else is almost the same.

This is possible because the SSL certificate is a * (star, or wildcard) certificate, enabling you to run multiple services

on one DNS domain.

This was initially published by Dražen Babić on <https://github.com/jam-py/jam-py/issues/35>

5.4.4 How to do with Nginx with Gunicorn?

Green Unicorn (gunicorn) is an HTTP/WSGI server designed to serve fast clients or sleepy applications. That is to say; behind a buffering front-end server such as nginx or lighttpd.

By default, gunicorn will listen on 127.0.0.1. Navigate to jam App folder, or use (ie in scripts, cron job, etc)

```
python /usr/bin/gunicorn --chdir /path/to/jam/App wsgi
```

or from /path/to/jam/App:

```
gunicorn wsgi
[2018-04-13 15:01:44 +0000] [8650] [INFO] Starting gunicorn 19.4.5
[2018-04-13 15:01:44 +0000] [8650] [INFO] Listening at: http://127.0.0.1:8000 (8650)
[2018-04-13 15:01:44 +0000] [8650] [INFO] Using worker: sync
[2018-04-13 15:01:44 +0000] [8654] [INFO] Booting worker with pid: 8654
.
.
```

To start jam.py on all interfaces and port 8081:

```
gunicorn -b 0.0.0.0:8081 wsgi
[2018-04-13 15:03:34 +0000] [8680] [INFO] Starting gunicorn 19.4.5
[2018-04-13 15:03:34 +0000] [8680] [INFO] Listening at: http://0.0.0.0:8081 (8680)
[2018-04-13 15:03:34 +0000] [8680] [INFO] Using worker: sync
[2018-04-13 15:03:34 +0000] [8684] [INFO] Booting worker with pid: 8684
.
.
```

Spin up 5 workers if u like with `--workers=5`

Nginx:

comment out default location in /etc/nginx/sites-enabled/default (Linux Mint):

```
#location / {
#    # First attempt to serve request as file, then
#    # as directory, then fall back to displaying a 404.
#    try_files $uri $uri/ =404;
#}
```

and add:

```
# Proxy connections to the application servers
# app_servers
location / {

    proxy_pass          http://app_servers;
    proxy_redirect      off;
    proxy_set_header    Host $host;
    proxy_set_header    X-Real-IP $remote_addr;
    proxy_set_header    X-Forwarded-For $proxy_add_x_forwarded_for;
    proxy_set_header    X-Forwarded-Host $server_name;
```

(continues on next page)

(continued from previous page)

}

add in /etc/nginx/nginx.conf 127.0.0.1:8081 if this is your Gunicorn server address and port:

```
# Configuration containing list of application servers
upstream app_servers {
server 127.0.0.1:8081;
}
```

This also enables to have different App servers on different ports

```
Client Request ----> Nginx (Reverse-Proxy)
                        |
                        /|\
| | `--> App. Server I. 127.0.0.1:8081
| | `--> App. Server II. 127.0.0.1:8082
| | `--> App. Server III. 127.0.0.1:8083
```

Restart nginx and viola!

Congratulations! We can now test Nginx with Jam.py.

Now, certs:

in /etc/nginx/sites-enabled/jam we can have something like this to pass everything from http to https to 8001 port (or any other as per above):

```
server {
    listen 80;
    server_name YOUR_SERVER;

    access_log off;

    location /static/ {
        alias /path/to/jam/App/static/;
    }

    location / {
        proxy_pass http://127.0.0.1:8001;
        proxy_set_header X-Forwarded-Host $server_name;
        proxy_set_header X-Real-IP $remote_addr;
        add_header P3P 'CP="ALL DSP COR PSAa PSDa OUR NOR ONL UNI COM NAV"';
    }

    return 301 https://$server_name$request_uri;
}

server {
    listen 443;
    server_name YOUR_SERVER_FQDN;

    access_log off;

    location /static/ {
        alias /path/to/jam/App/static/;
    }
}
```

(continues on next page)

(continued from previous page)

```

location = /favicon.ico {
    alias /path/to/jam/App/favicon.ico;
}

ssl on;
ssl_certificate /etc/nginx/ssl/YOUR_SERVER.crt;
ssl_certificate_key /etc/nginx/ssl/YOUR_SERVER.key;
add_header Strict-Transport-Security "max-age=31536000";

location / {
    client_max_body_size 10M;
    proxy_pass http://127.0.0.1:8001;
    proxy_set_header X-Forwarded-Host $server_name;
    proxy_set_header X-Real-IP $remote_addr;
    add_header P3P 'CP="ALL DSP COR PSAa PSDa OUR NOR ONL UNI COM NAV"';
}

```

That's it!

Congratulations! We can now test Nginx with Jam.py on https port!

This was initially published by Dražen Babić on <https://github.com/jam-py/jam-py/issues/67>

5.5 How do I write functions which have a global scope

Each function defined in the server or client module of an item becomes an attribute of the item.

Thus, using the *task tree*, you can access any function declared in the client or server module in any project module.

For example, if we have a function `some_func` declared in the Customers client module, we can execute it in any module of the project. Note that the task is a global variable on the client.

```
task.customers.some_func()
```

On the server, the task is not global, but an item that triggered / called it is passed to each event handler and function called by the *server* method. Therefore, if the `some_func` function is declared in the Customers server module, it can be executed in a function or event handler as follows:

```
def on_apply(item, delta, params):
    item.task.customers.some_func()
```

Note that event handlers are just functions and can also be called from other modules.

5.6 How to validate field value

Write the *on_field_validate* event handler to validate field value.

For example, The event will triggered when the *post* method is called, that saves the record in memory or when the user leaves the input used to edit the unitprice field value.

```

function on_field_validate(field) {
    if (field.field_name === 'unitprice' && field.value <= 0) {
        return 'Unit price must be greater than 0';
    }
}

```

(continues on next page)

(continued from previous page)

```

    }
}

```

As an example, below is the code that doesn't use the *on_field_validate* method and checks the value of the unitprice field and prevents the user from leaving the input when the value is less than or equal to zero:

```

function on_edit_form_shown(item) {
    item.each_field( function(field) {
        var input = item.edit_form.find('input.' + field.field_name);
        input.blur( function(e) {
            var err;
            if ($(e.relatedTarget).attr('id') !== "cancel-btn") {
                err = check_field_value(field);
                if (err) {
                    item.alert_error(err);
                    input.focus();
                }
            }
        });
    });
}

function check_field_value(field) {
    if (field.field_name === 'album' && !field.value) {
        return 'Album must be specified';
    }
    if (field.field_name === 'unitprice' && field.value <= 0) {
        return 'Unit price must be greater than 0';
    }
}

```

In the *on_edit_form_shown* event handler, we iterate through all the fields using the *each_field* method and find the input data for each field, if it exists.

In the *on_edit_form_shown* event handler we iterate through all the fields using the *each_field* method and find the input for each field, if it exists. Each input has a class with the name of the field (*field_name*).

Then we assign a jQuery blur event to it, in which we call the *check_field_value* function, and, if it returns text string, we warn the user and focus the input. Before calling the function, we check whether the “Cancel” button was pressed.

We declared the *on_edit_form_shown* event handler in the item's module, so it will work in this module only.

We can declare the following event handler in the task client module so we can write *check_field_value* function in any module we need to enable this field validation. The *on_edit_form_shown* of the task is called first for every item when edit form is shown. See *Form events*.

```

function on_edit_form_shown(item) {
    if (item.check_field_value) {
        item.each_field( function(field) {
            var input = item.edit_form.find('input.' + field.field_name);
            input.blur( function(e) {
                var err;
                if ($(e.relatedTarget).attr('id') !== "cancel-btn") {
                    err = item.check_field_value(field);
                    if (err) {
                        item.alert_error(err);
                    }
                }
            });
        });
    }
}

```

(continues on next page)

(continued from previous page)

```

        input.focus();
    }
    });
});
}
}

```

In this event handler we check if the item has the `check_field_value` attribute. Each function declared in a module becomes an attribute of the item.

5.7 How to add a button to a form

The simplest way to add a button to an edit / view form is to use `add_edit_button` / `add_view_button` correspondingly. You can call this functions in the `on_edit_form_created` / `on_view_form_created` event handlers.

For example the Customers item uses this code in its client module to add buttons to a view form:

```

function on_view_form_created(item) {
    item.table_options.multiselect = false;
    if (!item.lookup_field) {
        var print_btn = item.add_view_button('Print', {image: 'icon-print'}),
            email_btn = item.add_view_button('Send email', {image: 'icon-pencil'});
        email_btn.click(function() { send_email() });
        print_btn.click(function() { print(item) });
        item.table_options.multiselect = true;
    }
}

```

In this code the item's `lookup_field` attribute is checked and if it is defined (the view form is not created to select a value for a lookup field) the two buttons are created and for them JQuery click events are assigned to `send_email` and `print` functions declared in that module.

5.8 How to execute script from client

You can use `server` method to send a request to the server to execute a function defined in the server module of an item.

En the example below we create the `btn` button that is a JQuery object. Then we use its `click` method to attach a function that calls the `server` method of the item to run the `calculate` function defined in the server module of the item.

The code in the client module:

```

function on_view_form_created(item) {
    var btn = item.add_view_button('Calculate', {type: 'primary'});
    btn.click(function() {
        item.server('calculate', [1, 2, 3], function(result, error) {
            if (error) {
                item.alert_error(error);
            }
            else {
                console.log(result);
            }
        });
    });
}

```

(continues on next page)

(continued from previous page)

```

    }
  })
});
}

```

The code in the server module:

```

def calculate(item, a, b, c):
    return a + b + c

```

5.9 How to change style and attributes of form elements

You can access any DOM element on forms using jQuery.

In the following example, in the `on_edit_form_created` event handler defined the item client module we find the **OK** button, hide it, and change the text of the **Cancel** button to “Close” in the edit form:

```

function on_edit_form_created(item) {
    item.edit_form.find("#ok-btn").hide();
    item.edit_form.find("#cancel-btn").text('Close');
}

```

When an application creates input controls, it adds a class with a name that is the *field_name* attribute of the corresponding field to each input.

Thus, using the jQuery *selectors*, we can find the input of the customer field as follows (we select the input with the “customer” class in the edit form):

```

item.edit_form.find("input.customer")

```

Having found the element of the form you can use JQuery methods to change it.

As the field inputs are created by *create_inputs* after the *on_edit_form_created* event have been triggered (see the *on_edit_form_created* event handler in the task client module) you must write *on_edit_form_shown* event handler to change inputs.

For example this code

```

function on_edit_form_shown(item) {
    item.edit_form.find('input.name').css('color', 'red');
    item.edit_form.find('input.name').css('font-size', '24px');
    item.edit_form.find('input.tracks_sold').width(20);
    item.edit_form.find('input.genre').parent().width('40%');
    item.edit_form.find('input.composer').prop('type', 'password');
}

```

will change form inputs this way:

Please, note that if you need to change the width of input with prepend or append buttons (inputs of date, datetime and lookup fields) set the width of the input parent:

```
item.edit_form.find('input.album').parent().width('50%');
```

Another way to change the style of DOM elements is to use CSS. When the task node is selected in the Application Builder, the “project css” button is located on the right pane. Click on it to open the *project.css* file, which is located in the project folder. You can use it to input CSS that defines the style of the DOM elements of the project.

Each item form created in the project has css classes that enable developer to identify the form.

Each form has a class identifying it's type: 'view-form', 'edit-form', 'filter-form' or 'param-form'.

For example, the following code will remove the images in the buttons at the bottom of the form:

```
.view-form .form-footer .btn i {
    display: none;
}
```

More edit form examples:

```
.edit-form #ok-btn {
    font-weight: bold;
    background-color: lightblue;
}

.edit-form.invoices input.total {
    color: red;
}
```

Also each form has a class with a name that is the *item_name* attribute of the item.

The following code will remove images in the buttons only in the **Invoices** view form:

```
.view-form.invoices .form-footer .btn i {
    display: none;
}
```

You can change the way tables are displayed. The tables that are created by the `create_table` method have a css class “dbtable” and a class with a name that is the `item_name` attribute of the item. each column of the table also have a class with a name that is the `field_name` attribute of the corresponding field.

The example, the following code will display cells of the **Invoices** table **Customer** column bold:

```
.dbtable.invoices td.customer {  
    font-weight: bold;  
}
```

One more way to change the way the field column is displayed is to write the `on_field_get_html` event handler.

For example:

```
function on_field_get_html(field) {  
    if (field.field_name === 'total') {  
        if (field.value > 10) {  
            return '<strong>' + field.display_text + '</strong>';  
        }  
    }  
}
```

5.10 How to create a custom menu

To create a custom menu you must specify a `custom_menu` option for the task’s `create_menu` method in the task’s client module.

5.11 How to append a record using an edit form without opening a view form?

You must first call the `open` method of the item to initiate its dataset. For example, if you want to add a new record to invoices in the Demo application, you can do so as follows:

```
var invoices = task.invoices.copy();  
invoices.open({ open_empty: true });  
invoices.append_record();
```

In this code, we create a copy of the item using the `copy` method so that this operation does not affect the Invoices view form if it is open in a tab.

You can also change the record, but before you do this, you must get it from the server. Below is the code that modifies the record with id 411. We check that the record exists using the `rec_count` property, otherwise we display a warning.

```
var invoices = task.invoices.copy();  
invoices.open({ where: {id: 411} });  
if (invoices.rec_count) {  
    invoices.edit_record();  
}  
else {  
    invoices.alert_error('Invoices: record not found.');}
```

In the example above the `open` method is not executed synchronously.

The code below does it asynchronously:

```
var invoices = task.invoices.copy();
invoices.open({ where: {id: 411} }, function() {
  if (invoices.rec_count) {
    invoices.edit_record();
  }
  else {
    invoices.alert_error('Invoices: record not found.');
```

Invoices has the `Modeless` attribute set in the Edit form dialog, so the the edit form with be opened in a tab. You can change it by setting modeless attribute of `edit_options` to make the edit form modal:

```
var invoices = task.invoices.copy();
invoices.edit_options.modeless = false;
```

5.12 How to prohibit changing record

Let's assume that we have an item with a boolean field “posted”, and if the value of the field is true, we must prohibit changing or deleting the record.

We can do this by writing the `on_after_scroll` event handler and using `permissions` property:

```
function on_after_scroll(item) {
  if (item.rec_count) {
    item.permissions.can_edit = !item.posted.value;
    item.permissions.can_delete = !item.posted.value;
    if (item.view_form) {
      item.view_form.find("#delete-btn").prop("disabled", item.posted.value);
    }
  }
}
```

In this event handler we check the value of the “posted” field and set the `permissions` property attributes to true.

We can also write the `on_apply` event handler in the server module of the item:

```
def on_apply(item, delta, params, connection):
  for d in delta:
    if d.posted.old_value:
      raise Exception('Document posted. No change allowed')
```

5.13 How to link two tables

We'll explain how to link two items on example of the tracks and invoicetable items from the demo application. We'll link the record of tracks with the corresponding list of sold tracks from invoicetable that contains all sold tracks from invoices.

The default behavior if `view_form` is defined in the `on_view_form_created` event handler declared in the task client module.

We will change it in the `on_view_form_created` event handler in the tracks client module

```
function on_view_form_created(item) {
    item.table_options.height -= 200;
    item.invoice_table = task.invoice_table.copy();
    item.invoice_table.paginate = false;
    item.invoice_table.create_table(item.view_form.find('.view-detail'), {
        height: 200,
        summary_fields: ['date', 'total'],
    });
}
```

Then we reduce height of the table that displays tracks data by 200 pixels

```
item.table_options.height -= 200;
```

create a *copy* of `invoice_table`, set its *paginate* attribute to false and call the *create_table* method to create a table that will display the sold tracks

```
item.invoice_table = task.invoice_table.copy();
item.invoice_table.paginate = false;
item.invoice_table.create_table(item.view_form.find('.view-detail'), {
    height: 200,
    summary_fields: ['date', 'total'],
});
```

For this table we set the height to 200 pixels and define to summary fields.

This table will always be empty if we won't define the following *on_after_scroll* event handler:

```
function on_after_scroll(item) {
    if (item.view_form.length) {
        if (item.rec_count) {
            item.invoice_table.set_where({track: item.id.value});
            item.invoice_table.set_order_by(['-invoice_date']);
            item.invoice_table.open(true);
        }
        else {
            item.invoice_table.close();
        }
    }
}
```

The *on_after_scroll* event is triggered whenever the current record is changed. So when the track is changed we call *open* method, pre-setting the filter and order

```
item.invoice_table.set_where({track: item.id.value});
item.invoice_table.set_order_by(['-invoice_date']);
item.invoice_table.open(true);
```

This method sends a request to the server, that generates sql query, executes it and returns a dataset that contains sold records of this track ordered in descending order of `invoice_date` field.

If the tracks dataset is empty we clear the sold records dataset by calling the *close* method.

Because controls in Jam.py are data-aware every change of sold records dataset will be displayed in the table that we created in the *on_view_form_created* event handler.

Now every time the track has changed the application send request to the server to renew the sold tracks. This is not effective and sometimes can lead to delays. To avoid this we use the JavaScript `setTimeout` function:

```

var scroll_timeout;

function on_after_scroll(item) {
    if (!item.lookup_field && item.view_form.length) {
        clearTimeout(scroll_timeout);
        scroll_timeout = setTimeout(
            function() {
                if (item.rec_count) {
                    item.invoice_table.set_where({track: item.id.value});
                    item.invoice_table.set_order_by(['-invoice_date']);
                    item.invoice_table.open(true);
                }
                else {
                    item.invoice_table.close();
                }
            },
            100
        );
    }
}

```

This function guarantees that the data will be updated no more than once every 100 milliseconds.

Since the `invoicetable` is a *detail* it has the `master_rec_id` field that stores a reference to invoice that has this record, we can show the user an invoice that contains the current sold record. To do so we pass to the `create_table` method the function that will be executed when user double click the record:

```

item.invoice_table.create_table(item.view_form.find('.view-detail'), {
    height: 200,
    summary_fields: ['date', 'total'],
    on_dbclick: function() {
        show_invoice(item.invoice_table);
    }
});

```

and define the function as follows:

```

function show_invoice(invoice_table) {
    var invoices = task.invoices.copy();
    invoices.set_where({id: invoice_table.master_rec_id.value});
    invoices.open(function(i) {
        i.edit_options.modeless = false;
        i.can_modify = false;
        i.invoice_table.on_after_open = function(t) {
            t.locate('id', invoice_table.id.value);
        };
        i.edit_record();
    });
}

```

In this function we create a copy of the invoices journal and find the invoice. When the open method is executed we will show the invoice by calling its `edit_record` method. But before calling it we set its attributes so that it will be modal and the user won't be able to modify it.

Besides we dynamically assign `on_after_open` event handler to the `invoice_table` detail of the invoice we get. In this event handler we will find the current record in the sold records by calling the `locate` method.

Finally we will check the `lookup_field` attribute of tracks. This attribute is true if the item was created to select a value for the lookup field when a user clicks on the button to the right of lookup field input. We will make so that the sold

tracks are not shown when the user selects the value for the lookup field.

In addition, we add an alert informing the user about the possibility of seeing the invoice.

Finally the code of the *on_view_form_created* will be as follows:

```
function on_view_form_created(item) {
    if (!item.lookup_field) {
        item.table_options.height -= 200;
        item.invoice_table = task.invoice_table.copy();
        item.invoice_table.paginate = false;
        item.invoice_table.create_table(item.view_form.find('.view-detail'), {
            height: 200,
            summary_fields: ['date', 'total'],
            on_dblclick: function() {
                show_invoice(item.invoice_table);
            }
        });
        item.alert('Double-click the record in the bottom table ' +
            'to see the invoice in which the track was sold.');
```

```
    }
}

var scroll_timeout;
```

```
function on_after_scroll(item) {
    if (!item.lookup_field && item.view_form.length) {
        clearTimeout(scroll_timeout);
        scroll_timeout = setTimeout(
            function() {
                if (item.rec_count) {
                    item.invoice_table.set_where({track: item.id.value});
                    item.invoice_table.set_order_by(['-invoice_date']);
                    item.invoice_table.open(true);
                }
                else {
                    item.invoice_table.close();
                }
            },
            100
        );
    }
}
```

```
function show_invoice(invoice_table) {
    var invoices = task.invoices.copy();
    invoices.set_where({id: invoice_table.master_rec_id.value});
    invoices.open(function(i) {
        i.edit_options.modeless = false;
        i.can_modify = false;
        i.invoice_table.on_after_open = function(t) {
            t.locate('id', invoice_table.id.value);
        };
        i.edit_record();
    });
}
```

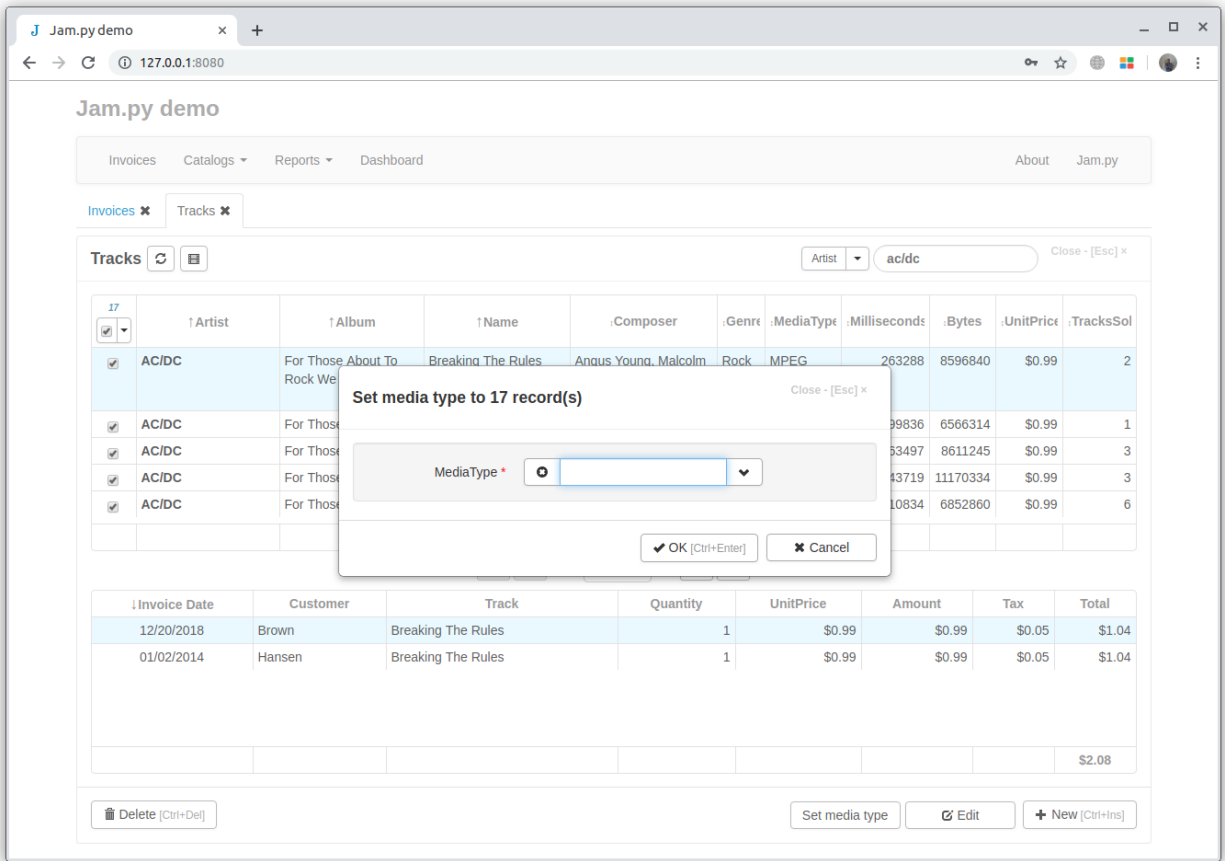
The screenshot shows the Jam.py demo application running in a web browser. The interface includes a top navigation bar with links for Invoices, Catalogs, Reports, and Dashboard. Below this, there are tabs for Invoices and Tracks. The Tracks tab is active, displaying a table of tracks. The table has columns for Artist, Album, Name, Composer, Genre, Media Type, Milliseconds, Bytes, Unit Price, and Tracks Sold. The track 'Inject The Venom' by Angus Young, Malcolm Young, and Brian Johnson is selected. Below the main table, there is a summary row showing 3504 tracks. At the bottom, there is a table of invoices with columns for Invoice Date, Customer, Track, Quantity, Unit Price, Amount, Tax, and Total. The total amount for the selected track is \$6.24.

!Artist	!Album	!Name	.Composer	.Genre	.Media Type	.Milliseconds	.Bytes	.Unit Price	.Tracks Sold
AC/DC	For Those About To	Breaking The Rules	Angus Young, Malcolm	Rock	MPEG	263288	8596840	\$0.99	2
AC/DC	For Those About To	C.O.D.	Angus Young, Malcolm	Rock	MPEG	199836	6566314	\$0.99	1
AC/DC	For Those About To	Evil Walks	Angus Young, Malcolm	Rock	MPEG	263497	8611245	\$0.99	3
AC/DC	For Those About To	For Those About To	Angus Young, Malcolm	Rock	MPEG	1343719	11170334	\$0.99	3
AC/DC	For Those About To	Inject The Venom	Angus Young, Malcolm Young, Brian Johnson	Rock	MPEG audio file	210834	6852860	\$0.99	6
AC/DC	For Those About To	Night Of The Long	Angus Young, Malcolm	Rock	MPEG	205688	6706347	\$0.99	2
AC/DC	For Those About To	Put The Finger On You	Angus Young, Malcolm	Rock	MPEG	205662	6713451	\$0.99	3
AC/DC	For Those About To	Snowballed	Angus Young, Malcolm	Rock	MPEG	203102	6599424	\$0.99	2
AC/DC	For Those About To	Spellbound	Angus Young, Malcolm	Rock	MPEG	270863	8817038	\$0.99	2
AC/DC	Let There Be Rock	Bad Boy Boogie	AC/DC	Rock	MPEG	267728	8776140	\$0.99	0
3504									

!Invoice Date	Customer	Track	Quantity	Unit Price	Amount	Tax	Total
12/20/2018	Brown	Inject The Venom	1	\$0.99	\$0.99	\$0.05	\$1.04
10/05/2018	Almeida	Inject The Venom	1	\$0.99	\$0.99	\$0.05	\$1.04
06/29/2018	Bernard	Inject The Venom	1	\$0.99	\$0.99	\$0.05	\$1.04
12/29/2017	Dubois	Inject The Venom	1	\$0.99	\$0.99	\$0.05	\$1.04
							\$6.24

5.14 How change field value of selected records

In this example, we will show how to change the “Media Type” field of the “Tracks” catalog to the same value for the selected records.



First we set the multiselect attribute of the `table_options` to true to display the check box in the leftmost column of the “Tracks” table for the user to select the records and create the **Set media type** button in the `on_view_form_created` event handler in the client module of “Tracks”.

```
function on_view_form_created(item) {
    item.table_options.multiselect = true;
    item.add_view_button('Set media type').click(function() {
        set_media_type(item);
    });
}
```

When this button is pressed, the `set_media_type` function defined in the module is executed.

In this function we create a copy of the “Tracks” item. We pass to the `copy` method the handlers option equal to false. It means that all the settings to the item made in the Form Dialogs in the Application Builder and all the functions and events defined in the client module of the item will be unavailable to the copy.

Then we analyze the `selections` attribute that is the array of the values of primary key field of the records, selected by the user.

After it we initialize the dataset of the copy by calling the `open` method with `open_empty` option. We also set the fields options so that the dataset will have only one field `media_type`. We set the required attribute of that field to true.

And finally, before calling the `append_record` method, we dynamically assign the `on_edit_form_created` event handler to change the on click event of the **OK** button, that was defined in the client module of the task.

In the new on click event handler we, first, call the `post` method to check that the media type value is set, if exception is raised we call `edit` method to allow the user to set it.


```

function set_media_type(item) {
    var copy = item.copy({handlers: false}),
        selections = item.selections;
    if (selections.length > 1000) {
        item.alert('Too many records selected.');
```

When the user clicks the **OK** button, the item's `server` method executes the `set_media_type` function on the server, which changes the field value of the selected records.

After changing the records on the server we, on the client, unselect the records, refresh the data of the page, cancel editing by calling the `cancel_edit` method and inform the user of the results.

```

def set_media_type(item, media_type, selections):
    copy = item.copy()
    copy.set_where(id__in=selections)
    copy.open(fields=['id', 'media_type'])
    for c in copy:
```

(continues on next page)

(continued from previous page)

```

        c.edit()
        c.media_type.value = media_type
        c.post()
    c.apply()
    return True

```

5.15 How to save edit form without closing it

You can do it by adding a button that will save the record without closing the edit form.

Below is examples for synchronous and asynchronous cases.

```

function on_edit_form_created(item) {
    var save_btn = item.add_edit_button('Save and continue');
    save_btn.click(function() {
        if (item.is_changing()) {
            item.post();
            try {
                item.apply();
            }
            catch (e) {
                item.alert_error(error);
            }
            item.edit();
        }
    });
}

```

```

function on_edit_form_created(item) {
    var save_btn = item.add_edit_button('Save and continue');
    save_btn.click(function() {
        if (item.is_changing()) {
            item.disable_edit_form();
            item.post();
            item.apply(function(error) {
                if (error) {
                    item.alert_error(error);
                }
                item.edit();
                item.enable_edit_form();
            });
        }
    });
}

```

5.16 How to save changes to two tables in same transaction on the server

Below is two examples.

In the first example each *apply* method gets its own connection from connection pool and commits it after saving changes to the database.

In the second example the connection is received from connection pool and passed to each *apply* method so changes are committed at the end.

```
import datetime

def change_invoice_date(item, invoice_id):
    now = datetime.datetime.now()

    invoices = item.task.invoices.copy(handlers=False)
    invoices.set_where(id=invoice_id)
    invoices.open()
    invoices.edit()
    invoices.invoice_date.value = now
    invoices.post()
    invoices.apply()

    customer_id = invoices.customer.value
    customers = item.task.customers.copy(handlers=False)
    customers.set_where(id=customer_id)
    customers.open()
    customers.edit()
    customers.last_modified.value = now
    customers.post()
    customers.apply()
```

```
import datetime

def change_invoice_date(item, invoice_id):
    now = datetime.datetime.now()

    con = item.task.connect()
    try:
        invoices = item.task.invoices.copy(handlers=False)
        invoices.set_where(id=invoice_id)
        invoices.open()
        invoices.edit()
        invoices.invoice_date.value = now
        invoices.post()
        invoices.apply(con)

        customer_id = invoices.customer.value
        customers = item.task.customers.copy(handlers=False)
        customers.set_where(id=customer_id)
        customers.open()
        customers.edit()
        customers.last_modified.value = now
        customers.post()
        customers.apply(con)

    con.commit()
    finally:
        con.close()
```

5.17 How to prevent duplicate values in a table field

One of the ways to do it is to write the *on_apply* event handler.

In the example below, the delta parameter is a dataset that contains the changes that will be stored in the users table.

We go through the records of changes and if the record was not deleted or the login field didn't change we look for a record in the table with the same login and if it exists raise the exception. If the user is editing the record on the client using an edit form he won't be able to save it and will see the corresponding alert message.

```
def on_apply(item, delta, params, connection):
    for d in delta:
        if not (d.rec_deleted() or d.rec_modified() and d.login.value == d.login.old_
↪value):
            users = d.task.users.copy(handlers=False)
            users.set_where(login=d.login.value)
            users.open(fields=['login'])
            if users.rec_count:
                raise Exception('There is a user with this login - %s' % d.login.
↪value)
```

5.18 How to implement some sort of basic multi-tenancy? For example, to have users with separate data.

You can implement a multi-tenancy using Jam.py.

For example, if some item have a user_id field, the following code in the server module of the item will do the job:

```
def on_open(item, params):
    if item.session:
        user_id = item.session['user_info']['user_id']
        if user_id:
            params['__filters'].append(['user_id', item.task.consts.FILTER_EQ, user_
↪id])

def on_apply(item, delta, params, connection):
    if item.session:
        user_id = item.session['user_info']['user_id']
        if user_id:
            for d in delta:
                if d.rec_inserted():
                    d.edit()
                    d.user_id.value = user_id
                    d.post()
                elif d.rec_modified():
                    if d.user_id.old_value != user_id:
                        raise Exception('You are not allowed to change record.')
                elif d.rec_deleted():
                    if d.user_id.old_value != user_id:
                        raise Exception('You are not allowed to delete record.')
```

It uses a *session* attribute of the item to get a unique user id and *on_open* and *on_apply* event handlers.

The *on_open* event handler ensures that the sql select statement that applications generates will return only records where the user_id field will be the same as the ID of the user that sends the request.

And the *on_apply* event handler sets the user_id to the ID of the user that appended or modified the records.

You can use a more general approach and add the following code to the server module of the task. Then a multi-tenancy will be applied to every item that have a user_id field:

```

def on_open(item, params):
    if item.field_by_name('user_id'):
        if item.session:
            user_id = item.session['user_info']['user_id']
            if user_id:
                params['__filters'].append(['user_id', item.task.consts.FILTER_EQ,
↵user_id])

def on_apply(item, delta, params, connection):
    if item.field_by_name('user_id'):
        if item.session:
            user_id = item.session['user_info']['user_id']
            if user_id:
                for d in delta:
                    if d.rec_inserted():
                        d.edit()
                        d.user_id.value = user_id
                        d.post()
                    elif d.rec_modified():
                        if d.user_id.old_value != user_id:
                            raise Exception('You are not allowed to change record.')
                    elif d.rec_deleted():
                        if d.user_id.old_value != user_id:
                            raise Exception('You are not allowed to delete record.')

```

5.19 Can I use Jam.py with existing database

Please read this: *Intergation with existing database*

5.20 How can I use data from other database tables

You can use data from other database tables.

First you must specify table name and fields information. You can do it the following way:

- Select project node in the task tree and click **Database** button.
- Set DB manual mode and specify the database connection attributes.
- Import tables information as described in the *Integration with existing database*
- Select project node in the task tree, click **Database** button restore previous values.

Then in the server module of the new items you must add code to read and write the data to the database

Below is the code for MySQL database (auto incremented primary field):

```

import MySQLdb
from jam.db import mysql

def on_open(item, params):
    connection = item.task.create_connection_ex(mysql, database='demo', \
        user='root', password='111', host='localhost', encoding='UTF8')
    try:
        sql = item.get_select_query(params, mysql)

```

(continues on next page)

(continued from previous page)

```

        rows = item.task.select(sql, connection, mysql)
    finally:
        connection.close()
    return rows, ''

def on_apply(item, delta, params):
    connection = item.task.create_connection_ex(mysql, database='demo', \
        user='root', password='111', host='localhost', encoding='UTF8')
    try:
        sql = delta.apply_sql(params, mysql)
        result = item.task.execute(sql, None, connection, mysql)
    finally:
        connection.close()
    return result

```

If database use generators to get primary field values you must specify them for new records (Firebird):

```

import fdb
from jam.db import firebird

def on_open(item, params):
    connection = item.task.create_connection_ex(firebird, database='demo.fdb', \
        user='SYSDBA', password='masterkey', encoding='UTF8')
    try:
        sql = item.get_select_query(params, firebird)
        rows = item.task.select(sql, connection, firebird)
    finally:
        connection.close()
    return rows, ''

def get_id(table_name, connection):
    cursor = connection.cursor()
    cursor.execute('SELECT NEXT VALUE FOR "%s" FROM RDB$DATABASE' % (table_name + '_
↳SEQ'))
    r = cursor.fetchall()
    return r[0][0]

def on_apply(item, delta, params):
    connection = item.task.create_connection_ex(firebird, database='demo.fdb', \
        user='SYSDBA', password='masterkey', encoding='UTF8')
    for d in delta:
        if not d.id.value:
            d.edit()
            d.id.value = get_id(item.table_name, connection)
            for detail in d.details:
                for r in detail:
                    if not r.id.value:
                        r.edit()
                        r.id.value = get_id(r.table_name, connection)
                    r.post()
            d.post()
    try:
        sql = delta.apply_sql(params, firebird)
        result = item.task.execute(sql, None, connection, firebird)
    finally:
        connection.close()
    return result

```

You can use the task `on_open` and `on_apply` events. Below is the code from task client module:

```
import MySQLdb
from jam.db import mysql

def on_open(item, params):
    if item.item_name in ['table1', 'table2']: # or
    #if item.table_name in ['table1', 'table2']:
        connection = item.task.create_connection_ex(mysql, database='demo', \
            user='root', password='111', host='localhost', encoding='UTF8')
        try:
            sql = item.get_select_query(params, mysql)
            rows = item.task.select(sql, connection, mysql)
        finally:
            connection.close()
        return rows, ''

def on_apply(item, delta, params):
    if item.item_name in ['table1', 'table2']:
        connection = item.task.create_connection_ex(mysql, database='demo', \
            user='root', password='111', host='localhost', encoding='UTF8')
        try:
            sql = delta.apply_sql(params, mysql)
            result = item.task.execute(sql, None, connection, mysql)
        finally:
            connection.close()
        return result
```

Note: Do not set History attribute to True for this tables. If you do so you'll get the exception. History table must be one for all databases that you use in the project. You can try to create the history table in the other database and write the `on_open` and `on_apply` event handlers for it.

5.21 How I can process a request or get some data from other application or service

You can access the data of your application for reading and writing by sending a post request that has 'ext' added to url. For example:

```
http://example.com/ext/edit
```

When an web app on the server receives such request it generates the *on_ext_request* event

5.22 How can I perform calculations in the background

You can use this code in the task server module to run a background thread in the web application once a 3 minutes (can be changed by setting interval) to perform some calculations:

```
import threading
import time
import traceback
```

(continues on next page)

(continued from previous page)

```

def background(task):
    interval = 3 * 60
    time.sleep(interval)
    while True:
        if not time:
            return
        with task.lock('background'):
            try:
                print('background')
                # some code to execute in background for example:
                # tracks = task.tracks.copy()
                # tracks.open()
                # for t in tracks:
                #     t.edit()
                #     t.sold.value = #some value
                #     t.post()
                # tracks.apply()
            except Exception as e:
                traceback.print_exc()
            time.sleep(interval)

def on_created(task):
    bg = threading.Thread(target=background, args=(task,))
    bg.daemon = True
    bg.start()

```

Note: When multiple web applications are running in parallel processes, the background function will be executed in each process. To prevent simultaneous execution of this function, we use the lock method of the task.

5.23 Is it supported to have details inside details?

Yes, you can have details inside details.

Suppose we have three objects - “Polls”, “Questions” and “Answers.” “Answers” is a detail of “Questions”. We will make “Questions” a detail of “Polls”.

One way to do this is to add an integer field “poll” to the “Questions” and the following code to the “Poll” client module:

```

function on_edit_form_created(item) {
    var q = task.questions.copy();

    item.edit_form.find('.form-footer').hide();

    q.view_options.form_header = false;

    q.on_view_form_created = function(quest) {
        quest.paginate = false;
    };

    q.on_before_append = function(quest) {
        if (!item.id.value) {

```

(continues on next page)

(continued from previous page)

```
        quest.alert_error('Poll is not specified.');
```

```
        quest.abort();
```

```
    }
```

```
};
```

```
q.on_before_post = function(quest) {
```

```
    quest.poll.value = item.id.value;
```

```
};
```

```
q.set_where({poll: item.id.value});
```

```
q.view(item.edit_form.find('.edit-detail'));
```

```
}
```

```
function on_field_changed(field, lookup_item) {
```

```
    var item = field.owner;
```

```
    item.apply();
```

```
    item.edit();
```

```
}
```

```
function on_before_delete(item) {
```

```
    var q = task.questions.copy();
```

```
    q.set_where({poll: item.id.value});
```

```
    q.open();
```

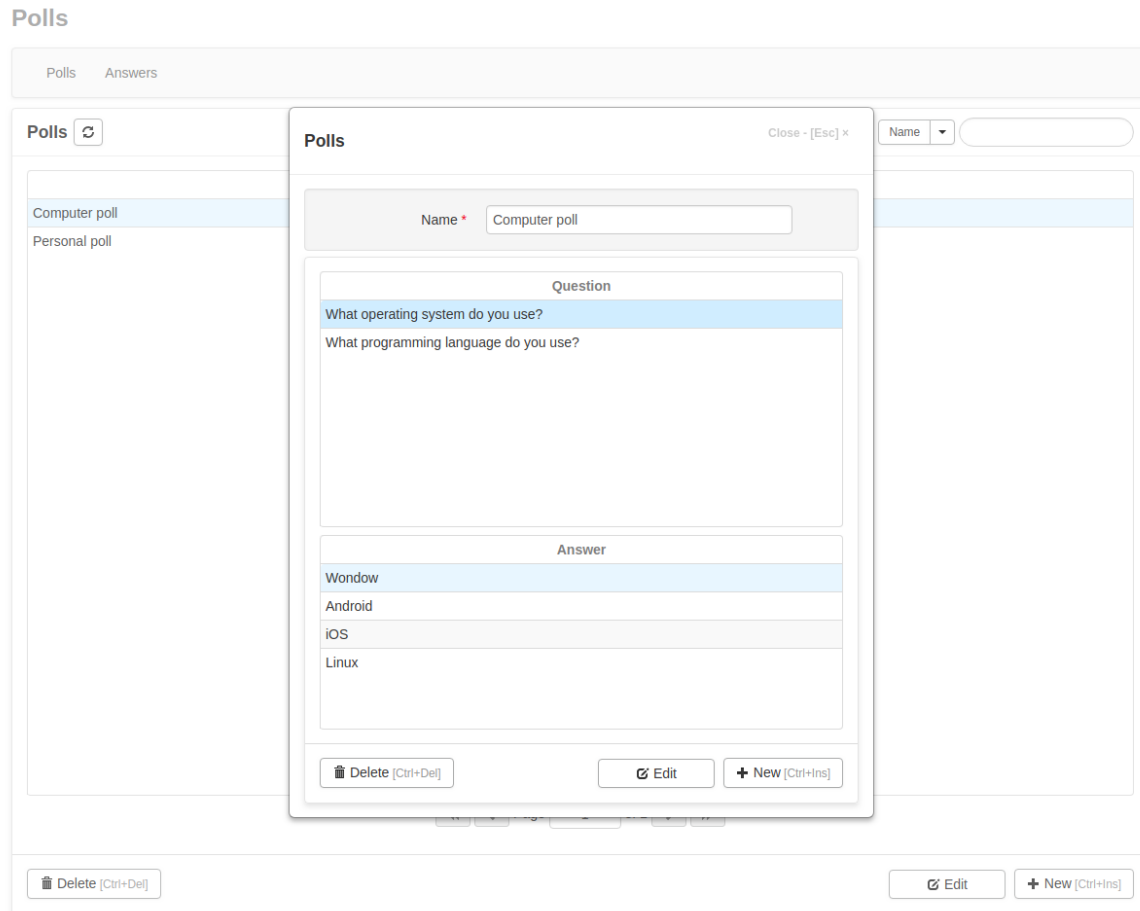
```
    while (!q.eof()) {
```

```
        q.delete();
```

```
    }
```

```
    q.apply();
```

```
}
```



5.24 Export to / import from csv files

First, in the client module of the item we create two buttons that execute the corresponding functions when you click on them:

```
function on_view_form_created(item) {
    var csv_import_btn = item.add_view_button('Import csv file'),
        csv_export_btn = item.add_view_button('Export csv file');
    csv_import_btn.click(function() { csv_import(item) });
    csv_export_btn.click(function() { csv_export(item) });
}

function csv_export(item) {
    item.server('export_csv', function(file_name, error) {
        if (error) {
            item.alert_error(error);
        }
        else {
            var url = [location.protocol, '///', location.host, location.pathname].
↪join('');
            url += 'static/files/' + file_name;
            window.open(encodeURIComponent(url));
        }
    })
}
```

(continues on next page)

(continued from previous page)

```

    });
}

function csv_import(item) {
    task.upload('static/files', {accept: '.csv', callback: function(file_name) {
        item.server('import_csv', [file_name], function(error) {
            if (error) {
                item.warning(error);
            }
            item.refresh_page(true);
        });
    });
}});
}

```

These functions execute the following functions defined in the server module. In this module we use the Python csv module. We do not export system fields - primary key field and deletion flag field.

Below is the code for Python 3:

```

import os
import csv

def export_csv(item):
    copy = item.copy()
    copy.open()
    file_name = item.item_name + '.csv'
    path = os.path.join(item.task.work_dir, 'static', 'files', file_name)
    with open(path, 'w', encoding='utf-8') as csvfile:
        fieldnames = []
        for field in copy.fields:
            if not field.system_field():
                fieldnames.append(field.field_name)
        writer = csv.DictWriter(csvfile, fieldnames=fieldnames)
        writer.writeheader()
        for c in copy:
            dic = {}
            for field in copy.fields:
                if not field.system_field():
                    dic[field.field_name] = field.text
            writer.writerow(dic)
    return file_name

def import_csv(item, file_name):
    copy = item.copy()
    path = os.path.join(item.task.work_dir, 'static', 'files', file_name)
    with open(path, 'r', encoding='utf-8') as csvfile:
        copy.open(open_empty=True)
        reader = csv.DictReader(csvfile)
        for row in reader:
            print(row)
            copy.append()
            for field in copy.fields:
                if not field.system_field():
                    field.text = row[field.field_name]
            copy.post()
        copy.apply()

```

For Python 2, this code looks like this:

```
import os
import csv

def export_csv2(item):
    copy = item.copy()
    copy.open()
    file_name = item.item_name + '.csv'
    path = os.path.join(item.task.work_dir, 'static', 'files', file_name)
    with open(path, 'wb') as csvfile:
        fieldnames = []
        for field in copy.fields:
            if not field.system_field():
                fieldnames.append(field.field_name.encode('utf8'))
        writer = csv.DictWriter(csvfile, fieldnames=fieldnames)
        writer.writeheader()
        for c in copy:
            dic = {}
            for field in copy.fields:
                if not field.system_field():
                    dic[field.field_name.encode('utf8')] = field.text.encode('utf8')
            writer.writerow(dic)
    return file_name

def import_csv2(item, file_name):
    copy = item.copy()
    path = os.path.join(item.task.work_dir, 'static', 'files', file_name)
    with open(path, 'rb') as csvfile:
        item.task.execute('delete from %s' % item.table_name)
        copy.open(open_empty=True)
        reader = csv.DictReader(csvfile)
        for row in reader:
            print(row)
            copy.append()
            for field in copy.fields:
                if not field.system_field():
                    field.text = row[field.field_name.encode('utf8')].decode('utf8')
            copy.post()
        copy.apply()
```

5.25 Authentication

In the Jam.py repository there is the “Authentication” project *export* file. This project demonstrates the first three topics of this section.

<http://jam-py.com/repository/auth.zip>

You can download it, create a new project and *import* this file.

5.25.1 How to authenticate from custom users table

By default, all user information is stored in a table in the admin.sqlite database. This table has a fixed structure that cannot be changed.

In this section, we describe how to authenticate a user using data from the custom users table.

First, we create an item group **Authentication** select it and add an item **Users** that has the following fields:

Item Editor **users**
Close - [Esc] x

Caption * Users
Name * users
Table name USERS_USERS
Primary key field id
Deleted flag deleted

Visible ☒
Soft delete ☒
Virtual table ☐
History ☐
External js module ☐

:Caption	↑Name	:DB field name	:Type	:Size	:Required	:Read only	:Lookup item	:Lookup field	:Master field	:Typeahead	:Lookup value list
Login	login	LOGIN	TEXT	100	x						
Name	name	NAME	TEXT	100	x						
Password	password	PASSWORD	TEXT	100							
Password hash	password_hash	PASSWORD_HASH	TEXT	200							
Role	role	ROLE	INTEGER		x						Roles

We won't store in the table the user password and use this field in the interface. We will store the password salted hash in the password_hash field.

We also created the *lookup list* "Roles" that we used in the "Roles" field definition.

We added to it the same roles (ids and names) as in the table *Roles* We 'll have to synchronize this roles in the future.

Lookup lists

Close - [Esc] ×

Name *

Roles

:Value	:Lookup value
1	Administrator
2	User

Delete [Ctrl+Del]

Edit

New [Ctrl+Ins]

OK

Cancel

In the *Roles* it is necessary to allow view the **Users** item only people that will be responsible for it

We removed password_hash field from field lists in the *View Form Dialog* and *Edit Form Dialog*

In the User server module we define the following *on_apply* event handler:

```
def on_apply(item, delta, params, connection):
    for d in delta:
        if not (d.rec_deleted() or d.rec_modified() and d.login.value == d.login.old_
↪value):
            users = d.task.users.copy(handlers=False)
            users.set_where(login=d.login.value)
            users.open(fields=['login'])
            if users.rec_count:
                raise Exception('There is a user with this login - %s' % d.login.
↪value)
```

(continues on next page)

(continued from previous page)

```

    if d.password.value:
        d.edit();
        d.password_hash.value = delta.task.generate_password_hash(d.password.
↪value)

        d.password.value = None
        d.post();

```

In this event handler we check if there is a users with the same login and raise the exception if such user exists, otherwise we generate hash using the *generate_password_hash* method of the task and set the password value to None.

In the client module we defined the following *on_field_get_text* event handler. It displays '*****' string insted of the password.

```

function on_field_get_text(field) {
    var item = field.owner;
    if (field.field_name === 'password') {
        if (item.id.value || field.value) {
            return '*****';
        }
    }
}

```

Finally, we define the *on_login* event handler in the task server module:

```

def on_login(task, form_data, info):
    users = task.users.copy(handlers=False)
    users.set_where(login=form_data['login'])
    users.open()
    if users.rec_count == 1:
        if task.check_password_hash(users.password_hash.value, form_data['password']):
            return {
                'user_id': users.id.value,
                'user_name': users.name.value,
                'role_id': users.role.value,
                'role_name': users.role.display_text
            }

```

Now we must add an admin to **Users** that has rights to work with users. After that we can set Safe mode in the project *Parameters*

5.25.2 How to create registration form

In this topic we'll assume that you have created a **Users** item from the previous topic.

Now we create a *register.html* file.

It contains a registration form:

```

<form id="login-form" target="dummy" class="form-horizontal" style="margin: 0;">
    <div class="control-group">
        <label class="control-label" for="name">Name</label>
        <div class="controls">
            <input type="text" id="name" placeholder="Login">
        </div>
    </div>
</form>

```

(continues on next page)

(continued from previous page)

```

<div class="control-group">
  <label class="control-label" for="login">Login</label>
  <div class="controls">
    <input type="text" id="login" placeholder="Login">
  </div>
</div>
<div class="control-group">
  <label class="control-label" for="password1">Password</label>
  <div class="controls">
    <input type="password" id="password1"
      placeholder="Password" autocomplete="on">
  </div>
</div>
<div class="control-group">
  <label class="control-label" for="password2">Repeat password</label>
  <div class="controls">
    <input type="password" id="password2"
      placeholder="Repeat password" autocomplete="on">
  </div>
</div>
<div class="alert alert-success" style="margin: 0; display: none">
  You have been successfully registered.
</div>
<div class="alert alert-error" style="margin: 0; display: none">
</div>
<div class="form-footer">
  <input type="button" class="btn expanded-btn pull-right"
    id="register-btn" value="OK" tabindex="3">
</div>
</form>

```

and a javascript code:

```

$(document).ready(function() {

  function register(name, login, password) {
    $.ajax({
      url: "ext/register",
      type: "POST",
      contentType: "application/json;charset=utf-8",
      data: JSON.stringify([name, login, password]),
      success: function(response, textStatus, jqXHR) {
        if (response.result.data) {
          show_alert(response.result.data);
        }
        else {
          $(".div.alert-success").show();
          setTimeout(
            function() {
              window.location.href = "index.html";
            },
            1000
          );
        }
      },
      error: function(jqXHR, textStatus, errorThrown) {
        console.log(errorThrown);
      }
    });
  }
});

```

(continues on next page)

(continued from previous page)

```

    }
    });
}
function show_alert(message) {
    $("div.alert-error")
        .text(message)
        .show();
}

$('input').focus(function() {
    $("div.alert").hide();
});

$("#register-btn").click(function() {
    var name = $("#name").val(),
        login = $("#login").val(),
        password1 = $("#password1").val(),
        password2 = $("#password2").val();
    if (!name) {
        show_alert('Name is not specified');
    }
    else if (!login) {
        show_alert('Login is not specified');
    }
    else if (!password1) {
        show_alert('Password is not specified');
    }
    else if (password1 !== password2) {
        show_alert('Passwords do not match');
    }
    else {
        register(name, login, password1)
    }
})
})

```

When the user clicks on the **OK** button, the javascript will send to the server the ajax post request with url “ext/register” and parameters “name, login, password”.

When server receives the request starting with ‘ext/’ it triggers the *on_ext_request* event.

The task server module has the following *on_ext_request* event handler:

```

def on_ext_request(task, request, params):
    reqs = request.split('/')
    if reqs[2] == 'register':
        name, login, password = params
        users = task.users.copy(handlers=False)
        users.set_where(login=login)
        users.open()
        if users.rec_count:
            return 'Existing login, please use different login'
        users.append()
        users.name.value = name
        users.login.value = login
        users.password_hash.value = task.generate_password_hash(password)
        users.role.value = 2

```

(continues on next page)

(continued from previous page)

```
users.post()
users.apply()
```

It checks if there is 'register' in url and then looks if there is no user with the login and then register the user.

5.25.3 How to give user ability to change the password

First we create a “Change password” item. While creating it we set the “Virtual table” and “Visible” attributes to false in the *Item Editor Dialog*. And we add to it two fields: “Old password”, “New password”

We’ll use this item for displaying “Change password” dialog.

To open this dialog we add a “Change password” menu item with id “pass” in the index.html:

```
<div class="container">
  <div id="taskmenu" class="navbar">
    <div class="navbar-inner">
      <ul id="menu" class="nav">
      </ul>
      <ul id="menu-right" class="nav pull-right">
        <li id="pass"><a href="#">Change password</a></li>
      </ul>
    </div>
  </div>
</div>
```

and in the task client module *on_page_loaded* event handler add the following code:

```
if (task.change_password.can_view()) {
  $("#menu-right #pass a").click(function(e) {
    e.preventDefault();
    task.change_password.open({open_empty: true});
    task.change_password.append_record();
  });
}
else {
  $("#menu-right #pass a").hide();
}
```

It will check if the user has the right to view item and then opens an empty dataset and creates an edit form, otherwise it hides this menu item.

In the “Change password” client module we add the following code:

```
function on_edit_form_created(item) {
  item.edit_form.find("#ok-btn")
    .off('click.task')
    .on('click', function() {
      change_password(item);
    });
  item.edit_form.find("#cancel-btn")
    .off('click.task')
    .on('click', function() {
      item.close_edit_form();
    });
}
```

(continues on next page)

(continued from previous page)

```

function change_password(item) {
    item.post();
    item.server('change_password', [item.old_password.value, item.new_password.value],
    ↪ function(res) {
        if (res) {
            item.warning('Password has been changed. <br> The application will be_
    ↪reloaded.',
                function() {
                    task.logout();
                    location.reload();
                });
        }
        else {
            item.alert_error("Can't change the password.");
            item.edit();
        }
    });
}

function on_field_changed(field, lookup_item) {
    var item = field.owner;
    if (field.field_name === 'old_password') {
        item.server('check_old_password', [field.value], function(error) {
            if (error) {
                item.alert_error(error);
            }
        });
    }
}

function on_edit_form_close_query(item) {
    return true;
}

```

In it we reassign **OK** and **Cancel** button click events. By default they are defined in the task client module to save record changes to the database and cancel editing. In the `on_edit_form_close_query` even handler we return true so the `on_edit_form_close_query` declared in the task client module, that shows “Yes No Cancel” dialog won’t be executed.

The `on_field_changed` event handler will check if old password is correct. It and the `change_password` function send requests to the server to execute functions defined in the item server module:

```

def change_password(item, old_password, new_password):
    user_id = item.session['user_info']['user_id']
    users = item.task.users.copy(handlers=False)
    users.set_where(id=user_id)
    users.open()
    same_password = item.task.check_password_hash(users.password_hash.value, old_
    ↪password)
    if users.rec_count== 1 and same_password:
        users.edit()
        users.password_hash.value = item.task.generate_password_hash(new_password)
        users.post()
        users.apply()
        return True
    else:

```

(continues on next page)

(continued from previous page)

```
        return False

def check_old_password(item, old_password):
    user_id = item.session['user_info']['user_id']
    users = item.task.users.copy(handlers=False)
    users.set_where(id=user_id)
    users.open()
    same_password = item.task.check_password_hash(users.password_hash.value, old_
    ↪password)
    if users.rec_count == 1 and same_password:
        return
    else:
        return 'Invalid password'
```

They use session to get id of the current user.

After changing the password the client reloads.

Business application builder

Application builder - is a Jam.py web application intended for application development and database administration.

To run the Application builder go to a Web browser and type in the browser address bar

```
127.0.0.1:8080/builder.html
```

Note: Please note that *server.py* must be running

On the left side of the Application builder page there is a panel that contains the project tree. When you select any node of the project tree, as a rule, its content will be opened in the central part of the page, and the bottom and right side of the page may have buttons that allow you to modify the content.

To see the changes made in Application builder go to the Project page and reload it.

6.1 Sanitizing

To prevent Cross Site Scripting (XSS) attacks, Jam.py sanitizes field values displayed in the table columns.

For example, if field contains the following text:

```
"<span style='color: red'>USA</span>"
```

when unsanitized it will be displayed in the table column as follows:

↑ Country
USA

When the field text sanitized, it is transformed to the following:

```
"&lt;t;span style='color: red'&gt;USA&lt;t;/span&gt;"
```

as you can see symbols ‘<’ and ‘>’ are replaced with ‘<’ and ‘>,’ and the table column will be displayed this way:

↑ Country
USA

There are two ways to prevent sanitizing.

First is to set **Do not sanitize** attribute in the Interface tab in the *Field Editor Dialog*

Second is to write the `on_field_get_html` event handler. If the this event handler returns a value it is not sanitized.

6.2 Accept string

An accept string can be a combination of the following values, separated by comma.

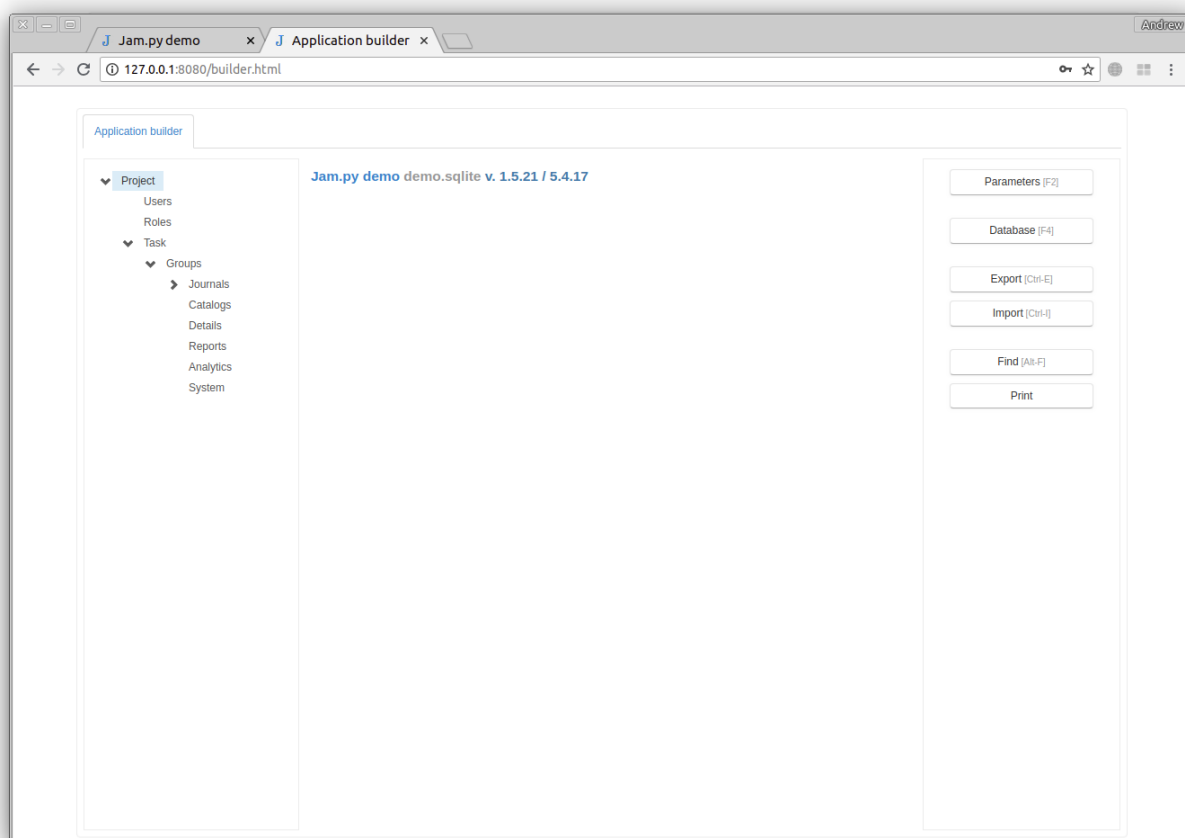
Value	Description
file_extension	Specify the file extension(s) (e.g: .gif, .jpg, .png, .doc)
audio/*	All sound files
video/*	All video files
image/*	All image files

For example:

```
.pdf, .xls  
image/*, .pdf, .xls  
audio/*  
audio/*, video/*
```

6.3 Project management

After the Application builder is first run or when the **Project** node is selected in the project tree, the Application builder page will look as follows:



Click on the links below to see the purpose of the buttons in the right panel of the page.

6.3.1 Parameters

After clicking on the **Parameters** button the Parameters Dialog will appear. It has two tabs **General** and **Interface**.

General tab

Parameters ?

Close - [Esc] x

General

Interface

Production

☐

Safe mode

☐

Debugging

☐

Language *

English en-US

Persistent connection

☒

Connection pool size

3

Compressed JS, CSS files

☐

All JS modules in a single file

☒

Dynamic JS modules loading

☐

History

History

Lock item

Locks

Session timeout (seconds)

86400

Session ignore change ip

☒

Max content length (MB)

0

Import delay (seconds)

0

Delete reports after (hours)

24

Upload file extension

.txt, .csv

Version

1.5.29

OK

Cancel

On the General tab, you can specify general parameters of the project:

- **Production** - if this checkbox is checked, you will not be able to change the project database in Application Builder.
- **Safe mode** - if safe mode is enabled, authentication is needed for user to work in the system (See [Users](#) and [Roles](#)).
- **Debugging** - if this checkbox is checked, the Werkzeug library debugger will be invoked when an error on the server occurs.
- **Language** - use it to open Language Dialog. See [Language support](#)
- **Persistent connection** - if this checkbox is checked the application creates a connection pool otherwise a connection is created before executing the sql query.
- **Connection pool size** — the size of the server database connection pool.
- **Compressed JS, CSS files** - If this button is checked the server returns compressed *js* and *css* files when *index.html* page is loaded.
- **All JS modules in a single file** - If this checkbox is unchecked, the application will generate a javascript file in the project *js* folder for every item in the *task tree*, that has code in its Client module, with the name *item_name.js*, where *item_name* is the name of an item. Otherwise, the application will generate a javascript file with the name *task_name.js*, where *task_name* is the name of the project *task* (for example *demo.js*), that will contain javascript code of all items, except items, whose **External js module** checkbox in the *Item Editor Dialog* is checked (separate files will be created for them).
- **Dynamic JS modules loading** - If this checkbox is unchecked and the application generates more than one javascript file, only file named *task_name.js* will be loaded when application is run. All other files must be loaded dynamically. See [Working with modules](#).
- **History item** - to specify item, that will store change history, see [Saving audit trail/change history made by users](#)
- **Session timeout (seconds)** - number of second of inactivity that is allowed before the session expires.
- **Session ignore change ip** - if false, the session is only valid when it is accessed from the same ip address that created the session.
- **Max content length (MB)** - use it to limit the total content length of the request to the server, in megabytes.
- **Import delay (seconds)** - if set the application will wait the number of seconds set in the parameter before changing the project dataset while [importing project metadata](#) , otherwise it waits for 5 minutes or until all previous request to the server in the current process will be processed.
- **Delete reports after (hours)** - if a value is specified the generated reports that are located in the static/reports folder will be deleted after specified number of hours have passed.
- **Upload file extensions** - is an [Accept string](#) that defines the types of files that could be uploaded to the server by the task [upload](#) method. Uploading files that do not match these types is prohibited.
- **Version** — specify the version of the project here.

Note: When **Connection pool size** or **Persistent connection** parameters are changed, the server application must be restarted for changes to take effect.

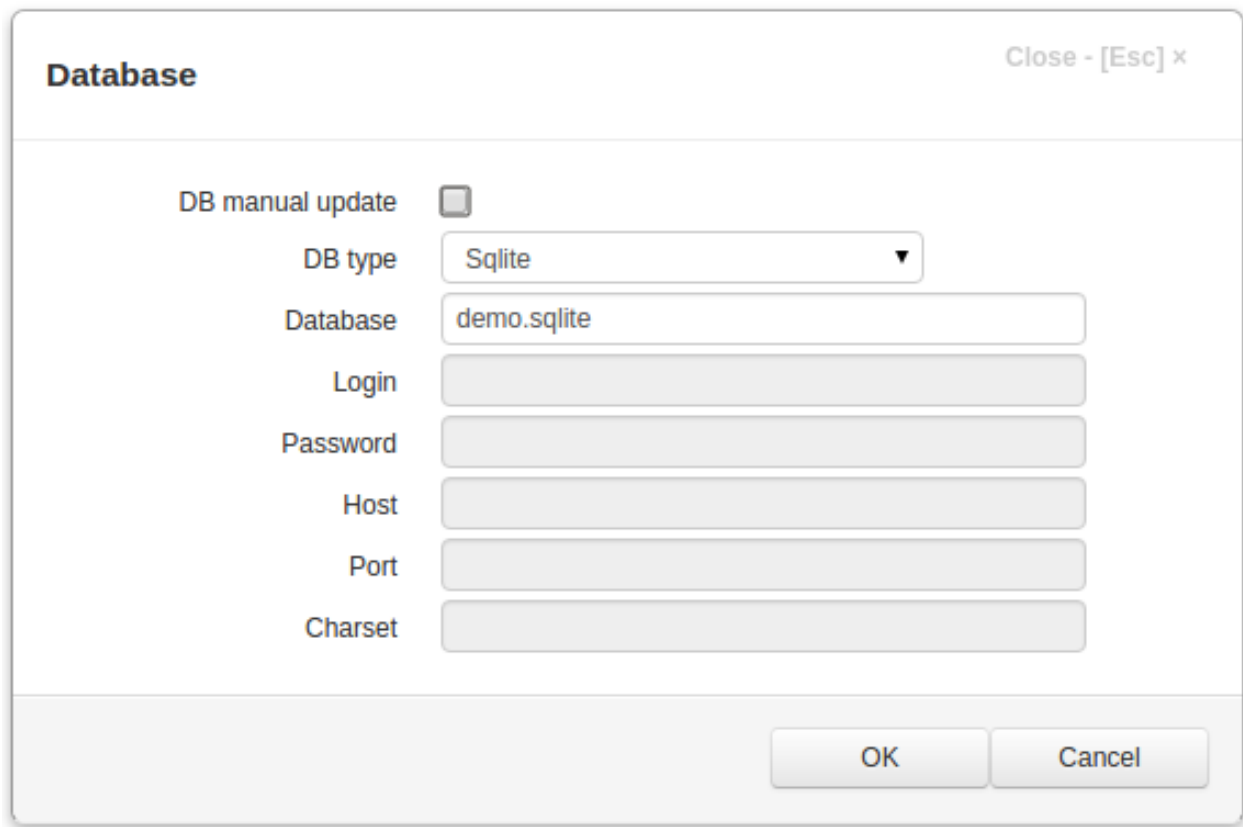
Interface tab

The screenshot shows a 'Parameters' dialog box with a title bar that includes a 'Close - [Esc] x' button. Inside the dialog, there are two tabs: 'General' and 'Interface'. The 'Interface' tab is currently selected. Under this tab, there are four settings: 'Theme' with a dropdown menu set to 'Bootstrap', 'Small font' with an unchecked checkbox, 'Full width' with an unchecked checkbox, and 'Display forms in tabs' with a checked checkbox. At the bottom right of the dialog are 'OK' and 'Cancel' buttons.

On the Interface tab, you can specify interface parameters of the project:

- **Theme** - use this parameter to select the theme of the project from one of predefined themes
- **Small font** - if this button is checked, the default font size will be 12px, otherwise it is 14px
- **Full width** - if this button is checked the project will fill the page width, without left and right margins
- **Display forms in tabs** - if this button is checked, the forms will be opened tabs

6.3.2 Database



The screenshot shows a 'Database' dialog box. It has a title bar with the text 'Database' and 'Close - [Esc] x'. The main area contains a checkbox labeled 'DB manual update' which is currently unchecked. Below this is a dropdown menu for 'DB type' with 'Sqlite' selected. There are then several text input fields: 'Database' (containing 'demo.sqlite'), 'Login', 'Password', 'Host', 'Port', and 'Charset'. At the bottom right of the dialog are two buttons: 'OK' and 'Cancel'.

In this dialog project database parameters are displayed. When they have been changed and OK button is clicked, the Application builder will check connection to the database and if it failed to connect an error will be displayed.

Note: When any **Database** parameter is changed, except **DB manual update**, the server application must be restarted for changes to take effect.

If **DB manual update** checkbox is unchecked (default), then when changes to an item, that have an associated database table, are saved, this database table is automatically modified. For example, if we add a new field to some item in the *Item Editor Dialog* , the new field will be added to the associated database table. If this checkbox is checked, no modifications to the database tables are made.

Note: Please be very careful when using this option.

Examples of database setups

SQLite

Database Close - [Esc] x

DB manual mode ☐

DB type *

⚙

Sqlite

▼

Server

Database *

demo.sqlite

Login

Password

Host

Port

Charset

OK

Cancel

PostgreSQL

DatabaseClose - [Esc] x

DB manual mode ☐

DB type *

⚙

PostgreSQL

▼

Server

Database *

demo

Login

postgres

Password

111

Host

localhost

Port

5432

Charset

UTF8

OK

Cancel

MySQL

Database Close - [Esc] x

DB manual mode ☐

DB type *

MySQL

▼

Server

Database *

demo

Login

root

Password

111

Host

localhost

Port

Charset

UTF8

OK

Cancel

FireBird

DatabaseClose - [Esc] x

DB manual mode ☐

DB type * FireBird

Server

Database *

Login

Password

Host

Port

Charset

MSSQL

DatabaseClose - [Esc] ×

DB manual mode ☐

DB type *

✱

MSSQL

▼

Server

.\\SQLEXPRESS

Database *

demo

Login

sa

Password

1111

Host

127.0.0.1

Port

1433

Charset

UTF8

OK

Cancel

Oracle

Database

Close - [Esc] ×

DB manual mode ☐

DB type *

✱

Oracle

▼

Server

Database *

xe

Login

system

Password

1111

Host

Port

Charset

OK

Cancel

6.3.3 Export

Press this button to export project *metadata* to zip file.

See also

Import

Metadata file

How to migrate development to production

6.3.4 Import

Use this button to import project *metadata* from zip file.

See also

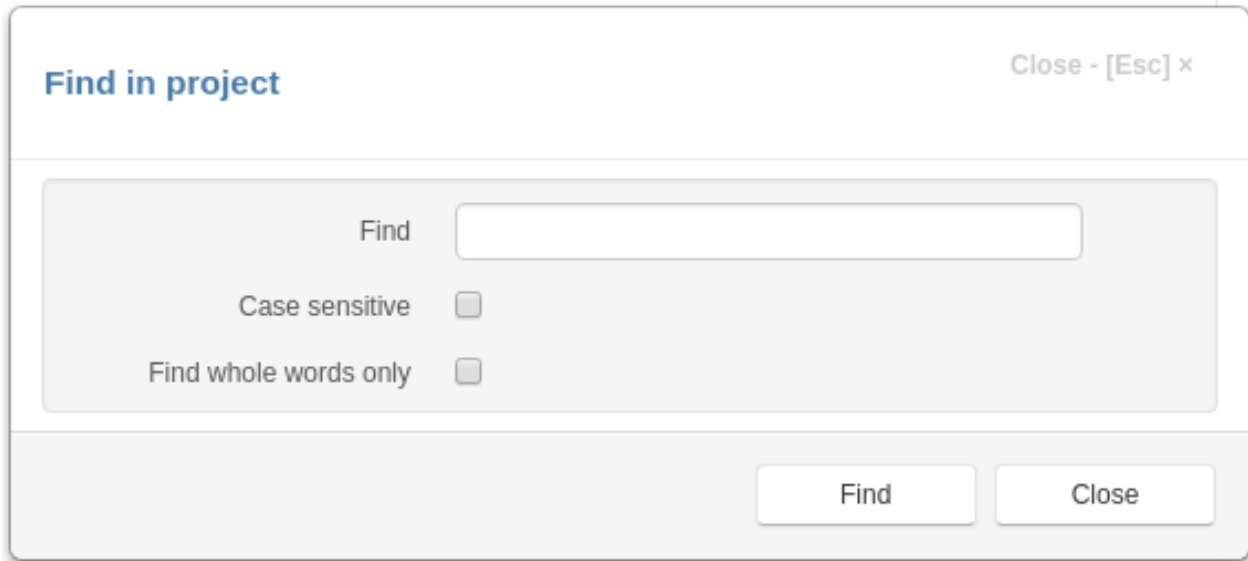
[Export](#)

[Metadata file](#)

[How to migrate development to production](#)

6.3.5 Find

Press this button to search for the character string in all modules of the project.

A dialog box titled "Find in project" with a close button labeled "Close - [Esc] x". Inside the dialog, there is a "Find" label next to a text input field. Below the input field, there are two checkboxes: "Case sensitive" and "Find whole words only", both of which are currently unchecked. At the bottom right of the dialog, there are two buttons: "Find" and "Close".

6.3.6 Print

Press this button to print all modules of the project.

6.3.7 Export/import metadata

All the code, parameters and data structure information of the project is stored in the *admin.sqlite* SQLite database located in the project folder. This information we call the metadata.

Export metadata

The project metadata can be exported to a zip file in the Application Builder by clicking the [Export](#) button.

This file contains the following information:

- Information about project data structure (information about DB tables, fields, indexes defined in the project items), code and settings that is stored in the *admin.sqlite* database.
- Files from the following folders:
 - css
 - js

- static/css
- static/js
- static/img
- reports (*ods* files of project report templates)
- utils (this folder may contain python libraries or files used in the project)

Import metadata

The metadata file can be imported to another project.

The web application while importing the metadata performs the following operations:

1. sets the `under_maintenance` flag so that incoming requests are not processed by the application. When receiving the response to these requests the client application shows the message “Web site currently under maintenance”. Web applications that run in parallel processes upon receiving requests check whether the `under_maintenance` flag is set and, if so, also do not process requests.
2. unzips the metadata file to the temporary folder in the project directory
3. checks the data integrity - items with the same *ID* in the current project and imported metadata must have the same type and the same *table_name* attribute if one is specified for an item
4. analyzes the metadata to generate sql queries to update the project database structure and records of the *admin.sqlite* database.
5. if the **Import delay** attribute in the project *Parameters* is set, waits the number of seconds set in the attribute, otherwise waits for 5 minutes or until all previous request to the server will be processed.
6. updates the project database and modifies the records of the *admin.sqlite*, see *below*
7. copies files from the temporary folder to the project folder
8. reloads the task tree
9. writes the import log file to the logs folder and sends the import log to the Application builder to be displayed
10. deletes the temporary folder
11. removes the `under_maintenance` flag
12. increases the build flag, so web applications that run in parallel processes and monitor this flag reload their task tree

Updating project databases

The way the the project database is updated depends on the type of the project database.

- Updating databases that support DDL statement rollback (**Postgress, Firebird, MSSQL**)
 - creates the connection to the project database
 - starts to execute sql statements to update the project database.
 - if an error occurs while updating the project database, rollbacks the changes, and goes to the step 9.
 - after the project database has been updated, makes a copy of the *admin.sqlite* file and starts modifying the *admin.sqlite* database
 - if an error occurs while modifying the records of *admin.sqlite*, restores *admin.sqlite* from the copy, roll-backs the changes to the project database and goes to the step 9.

- commits the changes, deletes the copy of the *admin.sqlite*, closes connection and goes to the step 7.
- Updating databases that do not support DDL statement rollback (MySQL, Oracle)
 - creates the connection to the project database
 - starts to execute sql statements to update the project database.
 - if an error occurs while updating the project database writes error to the import log and continues until all statements will be processed
 - commits the changes
 - starts modifying the *admin.sqlite* database
 - closes connection and goes to the step 7.

Causes of errors

Due to the fact that all items and fields of Jam.py projects have a unique ID attribute, Jam.py very accurately generates sql queries to modify the project database.

While generating sql queries the application currently compares only metadata in the current and imported project. The errors can occur when the application, for example, tries to add to a table a field that doesn't exist in the current project metadata but exists in the database table, you created this field outside of Application Builder. This situation can be corrected using Manual mode in Application Builder, see [Database](#), and changing the database.

If you won't change tables, field and indexes of production database, there will be no problems. Carry out development on the development project and then import its metadata into production.

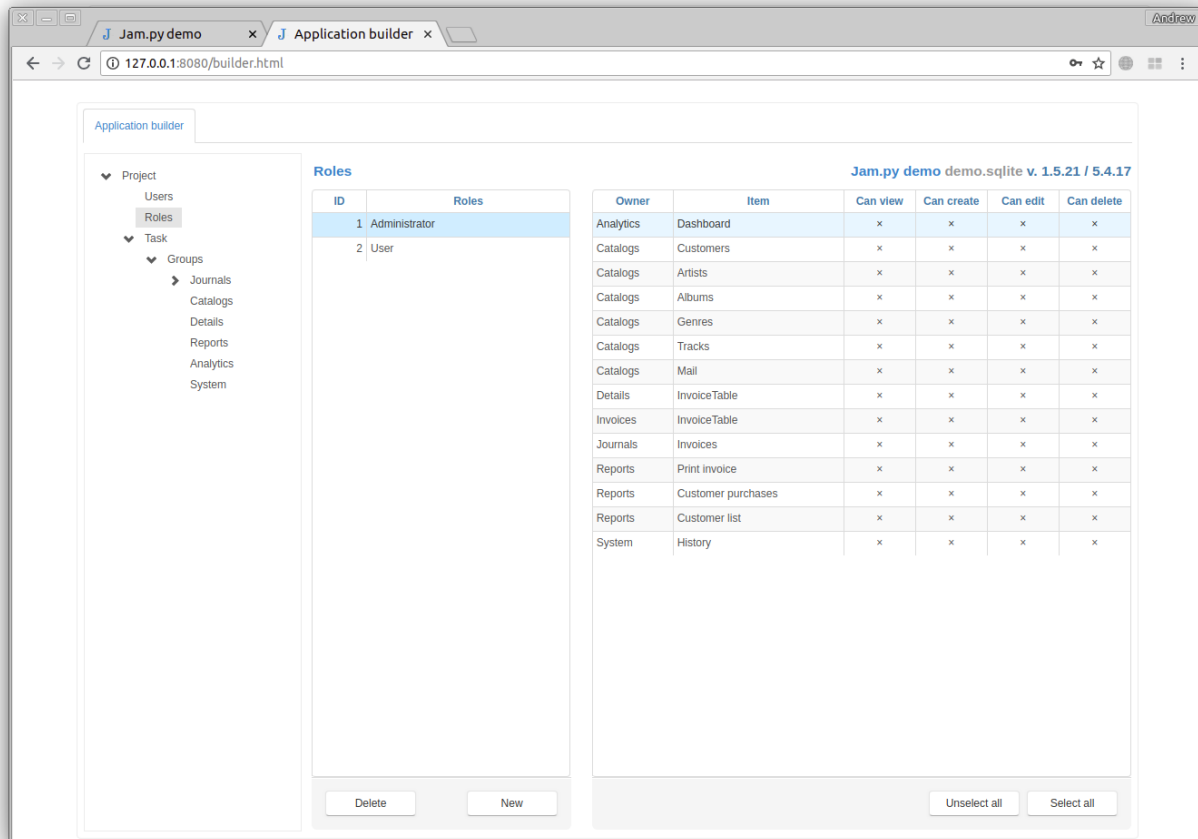
Note: For the databases that do not support DDL statement rollback (MySQL, Oracle) we recommend that you make a backup of the project database and *admin.sqlite* before performing the import.

Note: For **SQLite** database, Jam.py doesn't support importing of metadata into an existing project (project with tables in the database). You can only import metadata into a new project.

6.4 Roles

Select Roles node in the project tree to create and modify roles that defined users privileges. Each user must be assigned to one of roles defined in the project. A role defines the user's rights to view, create, modify, and delete data.

To add or delete a role, use New and Delete buttons. To set permissions for a role, select the role in a role list and put or remove a check mark next to the appropriate column by clicking on it with the mouse: View, Create, Edit, Delete (allowed to view, create, modify and delete, respectively).



6.5 Users

If the **Safe mode** checkbox in the *project parameters* is checked, authentication is needed for a user to work in the system.



But before that, the user must be registered in the framework. To register a user select Users node, click New and fill in the form that appears:

Users Close - [Esc] x

Name *

Login *

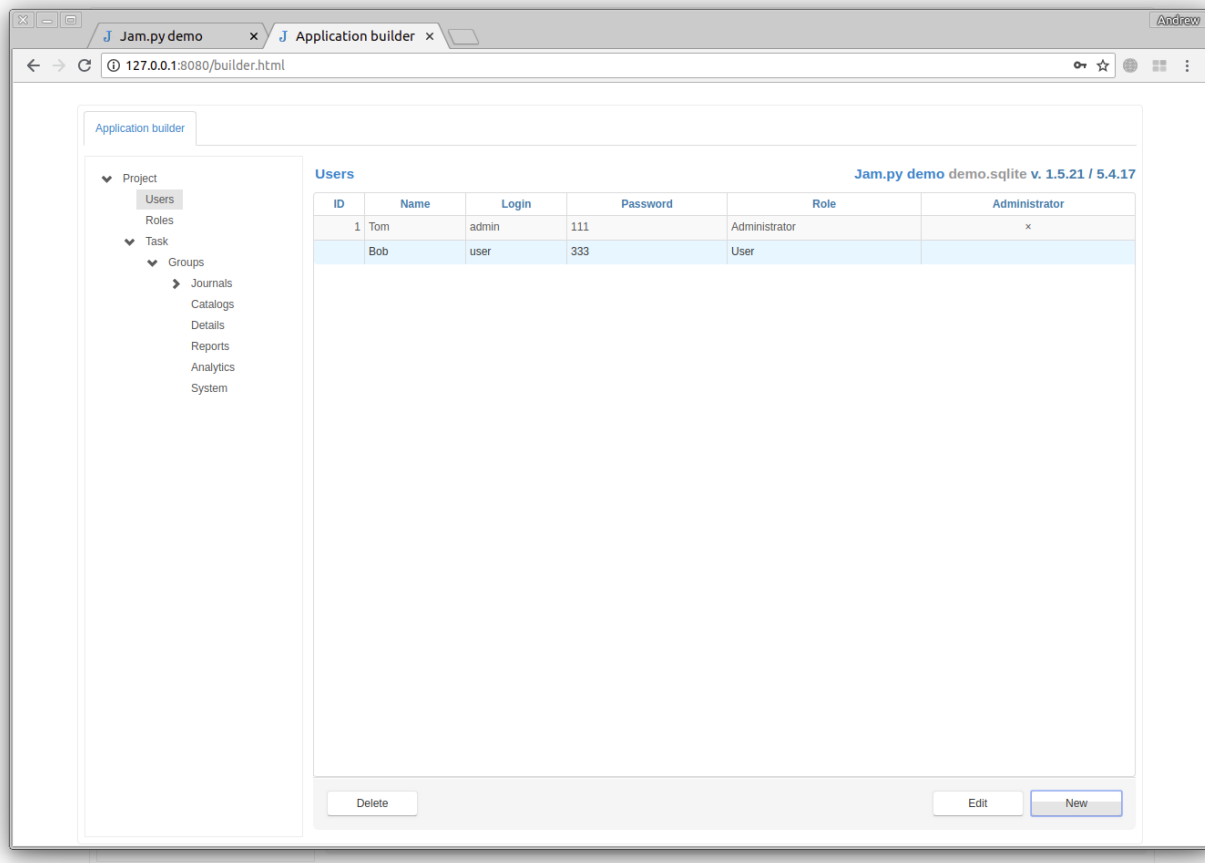
Password

Role *  

Administrator ☐

OK Cancel

- **Name** – user name
- **Login** - login
- **Password** - password
- **Role** – user role
- **Information** - some additional information
- **Admin** - if this flag is set, the user has the right to work in the Application builder.

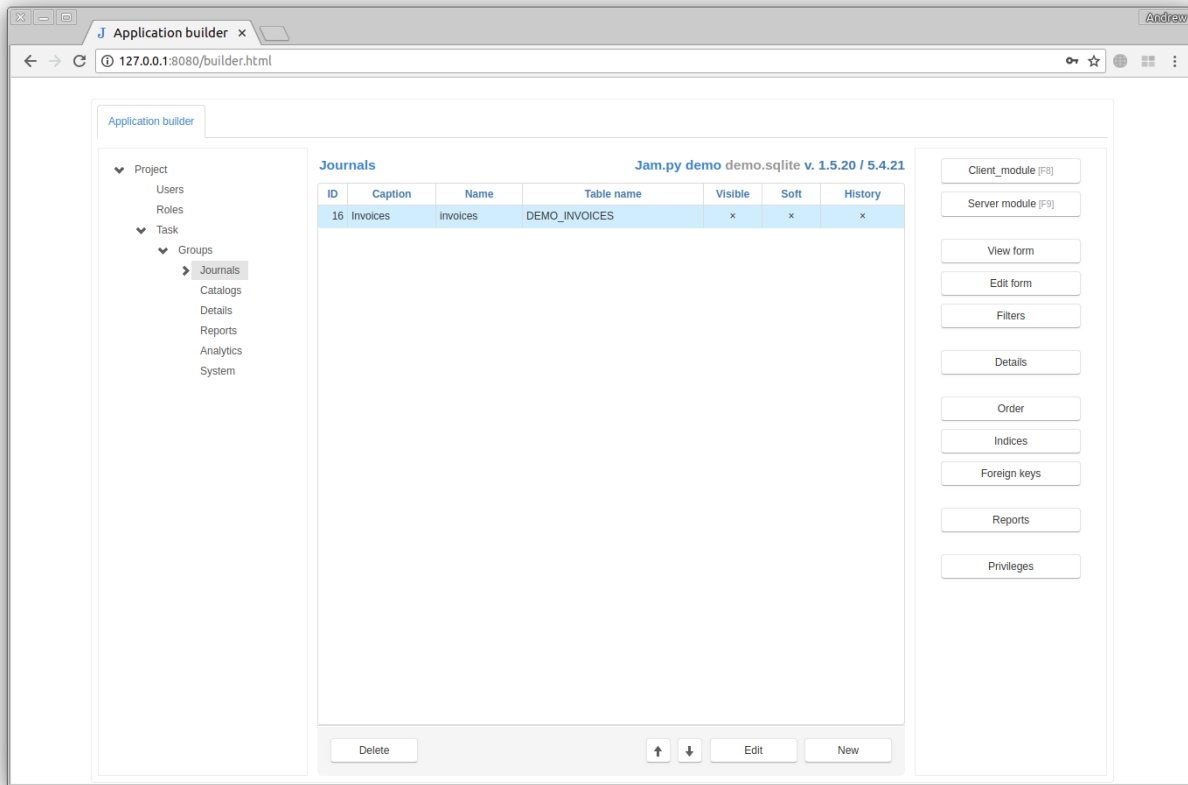


6.5.1 See also

on_login event

6.6 Code editor

For every item of the project *task tree* there are two buttons in the upper-right corner of the *Application builder* : **Client module** and **Server module**.



By clicking on these buttons the Code Editor for the client or server module of the item will be opened. (See [Working with modules](#))

To the left of the **Editor** there is an information pane with four tabs:

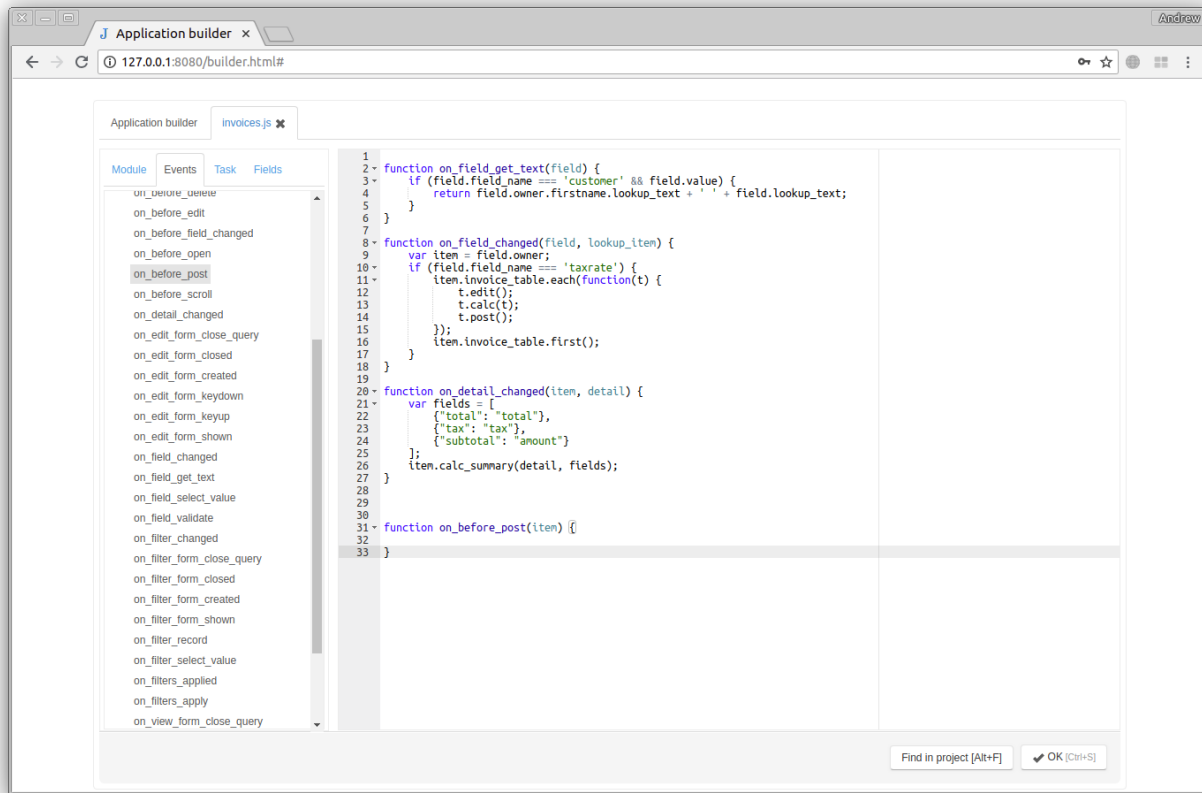
- **Module** - this tab displays all events and functions defined in the editor, double-click on one of them to move the cursor to the proper function.
- **Events** - displays all the published event of the item, double-click to add a wrapper for the event at the current cursor position (see the `on_before_post` event on the figure above).
- **Task** - the *task tree*, double-click on the node to enter the `item_name` at the current cursor position.
- **Fields** - the field list of the current item, double-click on one of the fields to enter the `field_name` at the current cursor position.

To save changes click the **OK** button or press Ctrl-S.

To search the project modules, click the **Find in project** button or press Alt-F to display the *Find inproject Dialog*

Jam.py uses the *ace editor* editor to implement its code editor.

Here are keyboard shortcuts for the *ace editor*.



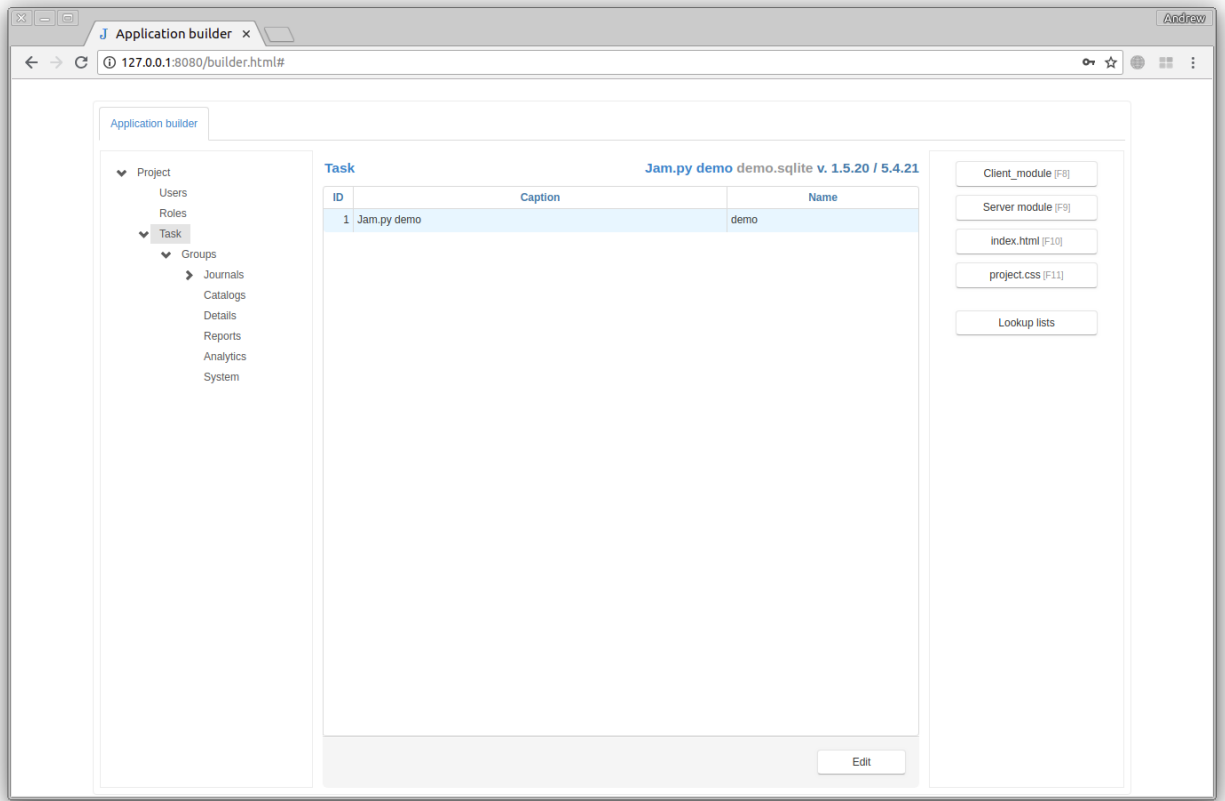
6.7 Task

Select **Task** node to get to the root of the project *task tree*.

Press the **Edit** button in the bottom of the page to change the name and caption of the task.

Use buttons in the right panel of the page to edit

- **Client** and **Server** modules of the task, see *Working with modules*, *Code editor*
- *index.html* file from the project root folder that contains project page and templates of the forms, see *Forms*, *Code editor*
- *project.css* file from *css* directory the project root folder, see *Code editor*
- **Lookup lists** - click on the button to open *Lookup lists* Dialog

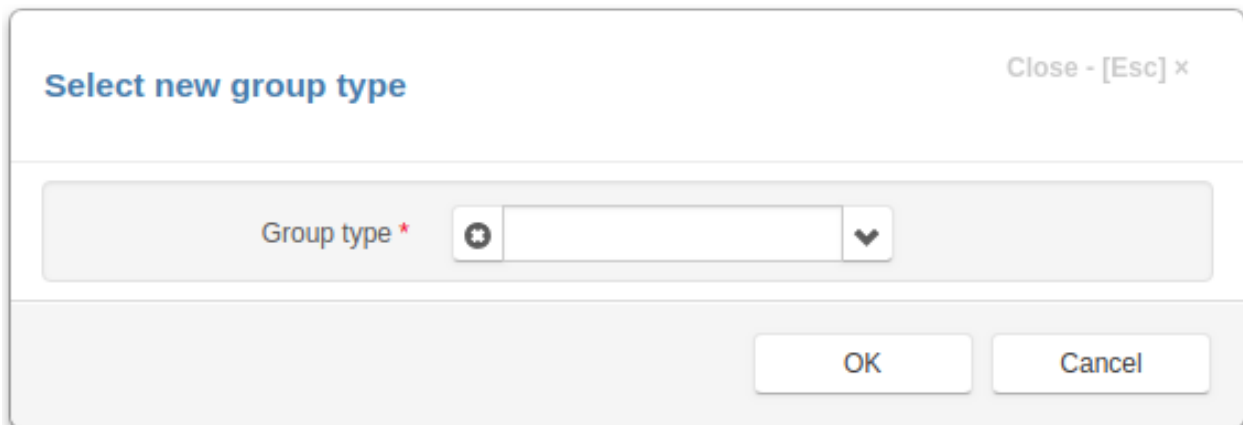


6.8 Groups

Select the node with the name of the task to get to the groups of the project *task tree*.

At the bottom of the page there are 3 buttons:

- **Delete** - click the button to delete an empty group.
- **Edit** - click this button to modify the selected group, the corresponding Group Editor will appear.
- **New** - use this button to create a new group. After that the you have to select a new group type in the Dialog box that will appear.



There are groups of three types: Item group, Report group, Table group, see [task tree](#). For each of this group, its own editor will be shown:

6.8.1 Item Group Editor

Item Group Editor opens when a developer wants to create a new item group or modify an existing one. See [Task tree](#)

The upper part of the **Item Group Editor** have the following fields:

- **Caption** - the group name that appears to users.
- **Name** - the name of the group that will be used in programming code to get access to the group object. It should be unique in the project and should be a valid python identifier.
- **Visible** - use this checkbox to set group's visible attribute. The value of this attribute can be used in code on the client to create menu items and so on.
- **Primary key field** - by clicking on the button to the right of this attribute you can specify the common primary key field for items that the group will own. You have to create this field first, see below
- **Deleted flag field** - by clicking on the button to the right of this attribute you can specify the common field that will serve as a deleted flag for items that the group will own. You have to create this field first, see below

In the center part of the **Item Group Editor** dialog there is a table containing a list of fields, defined for the item. These fields are *common* to all items the group will own.

To add, modify or delete a field use the following buttons:

- **New** - click the button to invoke the [Field Editor Dialog](#) to create a new field.
- **Edit** - click the button to invoke the [Field Editor Dialog](#) to modify a selected field.
- **Delete** - click the button to delete a field selected in in the field list.

In the bottom-right corner of the Dialog form there are two buttons:

- **OK** - click the button to save change you made.
- **Cancel** - click the buttons to cancel the operation.

Note: You can create new or modify existing fields and set **Primary key field** and **Deleted flag field** attributes only when creating a new group or editing an empty one.

For existing item groups, that already own items you can only change **Caption**, **Name** and **Visible** attributes.

Item Group Editor **journals** Close - [Esc] x

Caption *
 Primary key field

Name *
 Deleted flag

Visible ☒

Caption	Name ↑	Type	Size	Required	Read only	Lookup item	Lookup field	Master field	Typeahead	Lookup value list
Deleted flag	deleted	INTEGER								
Record ID	id	INTEGER								

6.8.2 Report Group Editor

Report Group Editor opens when the developer wants to create a new report group or change an existing report group.

The upper part of the **Report Group Editor** have the following fields:

- **Caption** - the group name that appears to users.
- **Name** - the name of the group that will be used in programming code to get access to the group object. It should be unique in the project and should be a valid python identifier.
- **Visible** - use this checkbox to set group's visible attribute. The value of this attribute can be used in code on the client to create menu items and so on.

In the bottom-right corner of the Dialog form there are two buttons:

- **OK** - click the button to save change you made.
- **Cancel** - click the buttons to cancel the operation.

Report Group Editor reports Close - [Esc] x

Caption *

Name *

Visible ☒

OK [Ctrl+Enter] Cancel [Esc]

6.8.3 Detail Group Editor

Detail Group Editor opens when a developer wants to create a new detail group or modify an existing one. See [Task tree](#)

The upper part of the **Detail Group Editor** have the following fields:

- **Caption** - the group name that appears to users.
- **Name** - the name of the group that will be used in programming code to get access to the group object. It should be unique in the project and should be a valid python identifier.
- **Visible** - use this checkbox to set group's visible attribute. The value of this attribute can be used in code on the client to create menu items and so on.
- **Primary key field** - by clicking on the button to the right of this attribute you can specify the common primary field for items that the group will own. You have to create this field first, see below
- **Deleted flag field** - by clicking on the button to the right of this attribute you can specify the common field that will serve as a deleted flag for items that the group will own. You have to create this field first, see below
- **Master ID field** - by clicking on the button to the right of this attribute you can specify the common field that will store ID of a master item for all detail items that the group will own, see [Details](#). You have to create this field first, see below
- **Master record id field** - by clicking on the button to the right of this attribute you can specify the common field that will store primary key value of a master item record for all detail items that the group will own, see [Details](#). You have to create this field first, see below

In the center part of the **Detail Group Editor** dialog there is a table containing a list of fields, defined for the item. These fields are *common* to all items the group will own.

To add, modify or delete a field use the following buttons:

- **New** - click this button to invoke the [Field Editor Dialog](#) to create a new field.
- **Edit** - click this button to invoke the [Field Editor Dialog](#) to modify a selected field.
- **Delete** - click the button to delete a field selected in in the field list.

In the bottom-right corner of the Dialog form there are two buttons:

- **OK** - click the button to save change you made.
- **Cancel** - click the buttons to cancel the operation.

Note: You can create new or modify existing fields and set **Primary key field**, **Deleted flag field** and **Master ID field**, **Master record id field** attributes only when creating a new group or editing an empty one.

For existing detail groups, that already own items you can only change **Caption**, **Name** and **Visible** attributes.

Table Group Editor details

Close - [Esc] x

Caption * Details

Deleted flag + deleted

Name * details

Master ID field + master_id

Visible ☐

Master record id field + master_rec_id

Primary key field + id

Caption	Name ↑	Type	Size	Required	Read only	Lookup item	Lookup field	Master field	Typeahead	Lookup value list
Deleted flag	deleted	INTEGER								
Record ID	id	INTEGER								
Master ID	master_id	INTEGER								
Master record id	master_rec_id	INTEGER								

Delete [Ctrl+Del]

Edit

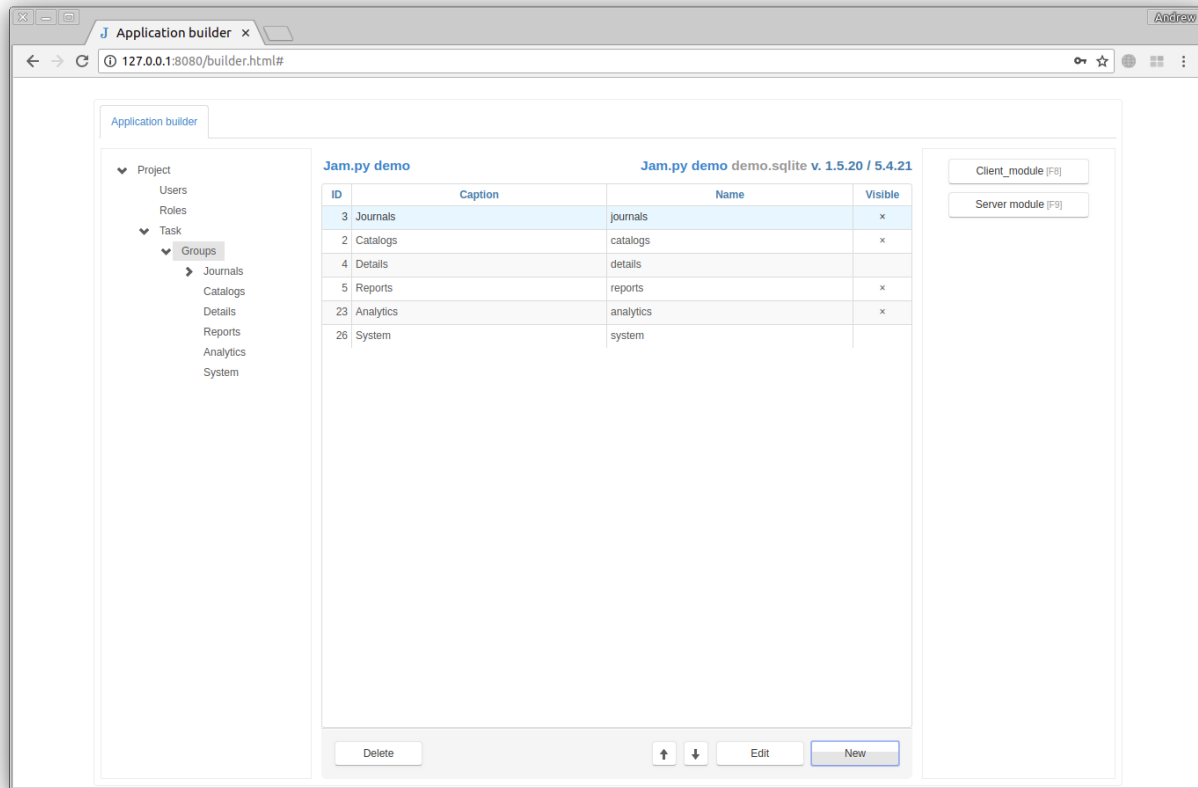
New [Ctrl+Ins]

OK [Ctrl+Enter]

Cancel [Esc]

Use buttons in the right panel of the page to edit Client and Server modules of a selected group, see

- *Working with modules*,
- *Code editor*



6.9 Items

Select a group node in the project tree to get access to items that this group owns, see [Task tree](#).

At the bottom of the page there are 3 buttons:

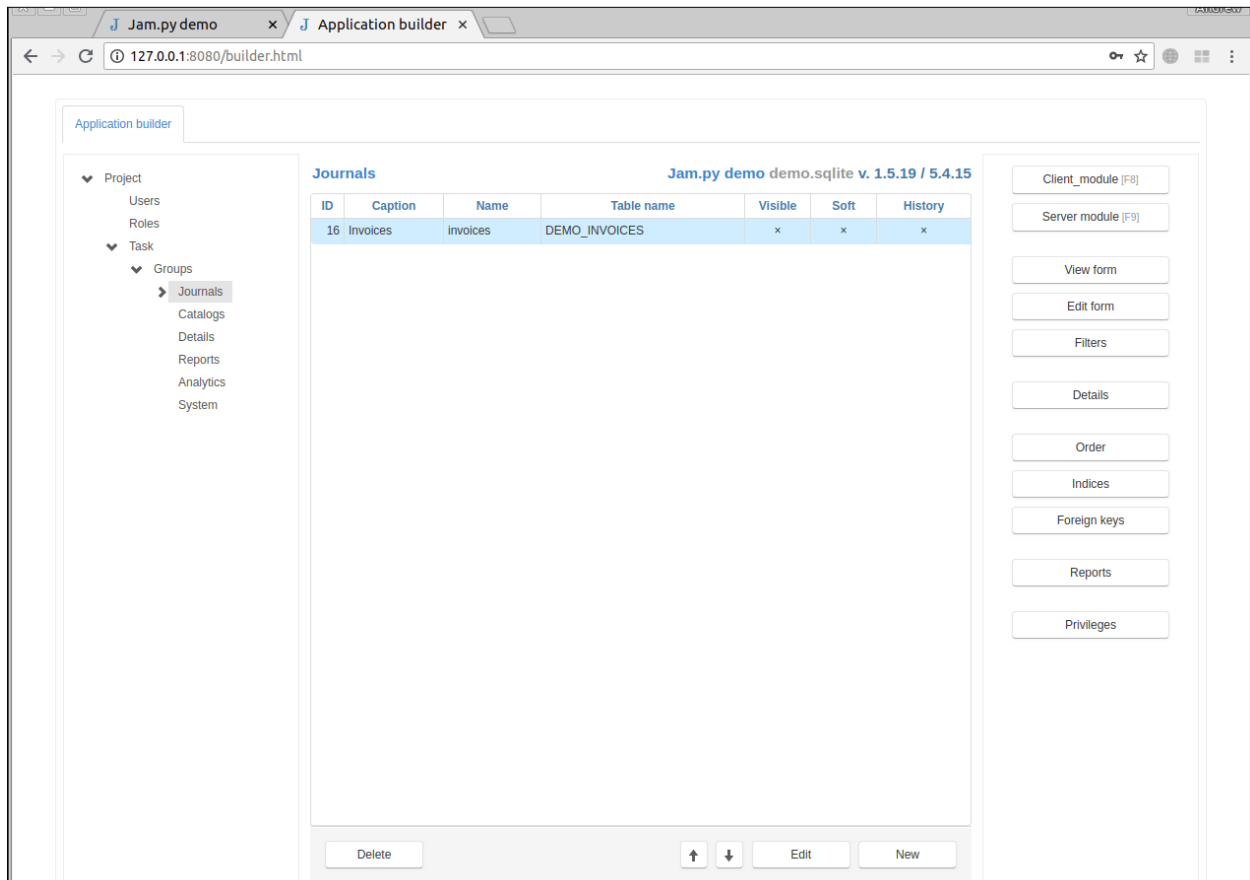
- **New** - click on New to create a new item in the [Item Editor Dialog](#)
- **Edit** - use this button to modify item's attributes as well to add, change or delete fields in the [Item Editor Dialog](#)
- **Delete** - click on the button to delete an item and its underlying database table.

You can use the up and down arrows to arrange the items in the list. This may be useful for creating a menu or display it in some way on the web page.

The right panel of the page have following buttons:

- **Client module** - click on this button to open the [Code editor](#) to edit client module of an item, see [Working with modules](#).
- **Server module** - click on this button to open the [Code editor](#) to edit server module of an item, see [Working with modules](#).
- **View Form** - use this button to invoke the [View Form Dialog](#) to set how the view form will be displayed.
- **Edit Form** - use this button to invoke the [Edit Form Dialog](#) to set how the edit form will be displayed.
- **Filters** - use this button to invoke the [Filters Dialog](#) to create, modify and delete item filters. See [Filters](#).
- **Details** - use this button to invoke the [Details Dialog](#) to add or remove details linked to the item.

- **Order** - use this button to invoke the *Order Dialog* to specify how records will be ordered by default. See *open* method
- **Indices** - lick this button to open the *Indices Dialog* to create and delete indices for the item database table.
- **Foreign keys** - lick this button to open the *Foreign Keys Dialog* to create foreign keys for the database table.
- **Reports** - lick this button to open the *Reports Dialog* to specify reports that could printed for the item. A new project has a function that can be used to create a dropdown button to print the reports.
- **Privileges** - click this button to open a dialog to configure the privileges assigned to user roles for this item.



6.9.1 Item Editor Dialog

Item Editor dialog opens when a developer selects a Group node in the project tree of the Application builder and click on the **New** or **Edit** button to create a new item or modify a selected one. See *Items*.

Item Editor invoices
Close - [Esc] x

Caption * Invoices
Name * invoices
Table name DEMO_INVOICES
Primary key field id
Deleted flag deleted

Visible ☒
Soft delete ☒
Virtual table ☐
History ☒
Edit lock ☒

:Caption	↑Name	:DB field name	:Type	:Size	:Required	:Read	:Lookup	:Lookup	:Master	:Typeahead	:Lookup value list
Billing Address	billing_address	BILLING_ADDRESS	INTEGER				customers	address	customer		
Billing City	billing_city	BILLING_CITY	INTEGER				customers	city	customer		
Billing Country	billing_country	BILLING_COUNTRY	INTEGER				customers	country	customer		
Billing Postal Code	billing_postal_code	BILLING_POSTAL_CODE	INTEGER				customers	postalcode	customer		
Billing State	billing_state	BILLING_STATE	INTEGER				customers	state	customer		
Customer	customer	CUSTOMER	INTEGER		x		customers	lastname		x	
Invoice Date	date	DATE	DATE		x						
Customer FirstName	firstname	FIRSTNAME	INTEGER				customers	firstname	customer		
SubTotal	subtotal	SUBTOTAL	CURRENCY			x					
Tax	tax	TAX	CURRENCY			x					
Tax Rate	taxrate	TAXRATE	FLOAT								
Total	total	TOTAL	CURRENCY		x	x					

Delete [Ctrl+Del]
Edit
New [Ctrl+Ins]
OK [Ctrl+Enter]
Cancel [Esc]

The upper part of the **Item Editor dialog** have the following fields:

- **Caption** - the item name that appears to users.
- **Name** - the name of the item that will be used in programming code to get access to the item object. It should be unique in the project and should be a valid python identifier.
- **Table** - the name of the table that will be created in the project database. This name is specified when creating an item, and can not be changed later.
- **Primary key field** - by clicking on the button to the right of this attribute you can specify the primary key field for the item. If the primary key field was defined for the group that owns the item it will be displayed there by default, otherwise you have to create this field first.
- **Deleted flag field** - by clicking on the button to the right of this attribute you can specify the field that will serve as a deleted flag for the item. If the deleted flag field was defined for the group that owns the item it will be displayed there by default, otherwise you have to create this field first.
- **Visible** - use this checkbox to set item's visible attribute. The value of this attribute can be used in code on the client to create menu items and so on.
- **Soft delete** - when this check-box is checked, the delete method does not erase a record physically from the table, but uses this field to mark the record as deleted. See [Common fields](#), [delete](#) method (server), [delete](#) method (client).
- **Virtual table** - if this checkbox is checked, no database table will be created. Use this options to create an item with in-memory dataset or to use its modules to write code. This checkbox must be set when creating an item

and can not be changed later.

- **History** - if this checkbox is checked, the application will saving for this item audit trail/change history made by users, see *Saving audit trail/change history made by users*
- **Edit lock** - if this checkbox is checked, the application will use record locking while users concurrently edit a record, see *Record locking*

In the center part of the **Item Editor dialog** there is a table containing a list of fields, defined for the item. To add, modify or delete a field use the following buttons:

- **New** - click this button to invoke the *Field Editor Dialog* to create a new field.
- **Edit** - click this button to invoke the *Field Editor Dialog* to modify a selected field.
- **Delete** - click this button to delete a field selected in in the field list.

In the bottom-right corner of the Dialog form there are two buttons:

- **OK** - click this button to save change you made. If the **Virtual table** checkbox is not checked and **DB manual update** parameter in the project *Database Dialog* is not set, the application will generate and execute SQL query to update the item table in the project Database (changes made to the fields will be applied to the table).
- **Cancel** - click this buttons to cancel the operation.

6.9.2 Field Editor Dialog

Use the **Field Editor Dialog** to create a new or modify an existing field.

It has two tabs **Field**, **Lookup** and **Interface**.

Field tab

The screenshot shows a dialog box titled "Field Editor customer" with a "Close - [Esc] x" button in the top right corner. Below the title bar are three tabs: "Field", "Lookup", and "Interface". The "Field" tab is selected. The main area contains the following fields and controls:

- Caption ***: A text box containing "Customer".
- Name ***: A text box containing "customer".
- Type ***: A dropdown menu with "INTEGER" selected and a small asterisk icon to its left.
- Size**: An empty text box.
- Default value**: An empty text box.
- Required**: A checkbox that is checked.
- Read only**: An unchecked checkbox.

At the bottom right of the dialog are two buttons: "OK [Ctrl+Enter]" and "Cancel".

The **Field** tab have the following fields:

- **Caption** - the field name that appears to users.
- **Name** - the name of the field that will be used in programming code to get access to the field object. It should be a valid python identifier.
- **Type** - type of the field — one of the following values:
 - **TEXT**
 - **INTEGER**
 - **FLOAT**
 - **CURRENCY**
 - **DATE**
 - **DATETIME**
 - **BOOLEAN**
 - **LONGTEXT**
 - **FILE**

– IMAGE

- **Size** - the size of the field for text fields.
- **Default value** - the default value of the field, for boolean fields use 0 or 1
- **Required** - if this checkbox is checked, the post method will raise an exception if this field is empty. See *Modifying datasets*.
- **Read only** - this checkbox is checked, the field value can not be changed in the interface controls created by the *create_inputs* method on the client.

Lookup tab

The screenshot shows the 'Field Editor customer' dialog box with the 'Lookup' tab selected. The dialog contains the following fields and controls:

- Lookup item:** A text field containing 'customers'.
- Lookup field:** A text field containing 'lastname'.
- Lookup field 2:** An empty text field.
- Lookup field 3:** An empty text field.
- Master field:** An empty text field.
- Typeahead:** A checked checkbox.
- Lookup value list:** An empty text field.

At the bottom right of the dialog are two buttons: 'OK [Ctrl+Enter]' and 'Cancel'. The title bar of the dialog reads 'Field Editor customer' and 'Close - [Esc] x'.

- **Lookup item** - the lookup item for *Lookup fields*
- **Lookup field** - the lookup field for *Lookup fields*
- **Lookup field 2** - the lookup field 2 for *Lookup fields*
- **Lookup field 3** - the lookup field 3 for *Lookup fields*
- **Master field** - the master field for *Lookup fields*
- **Typeahead** - if this checkbox is checked, typeahead is enabled for the lookup field
- **Lookup value list** - use it to specify a *lookup list* for an integer field

Interface tab

The screenshot shows a dialog box titled "Field Editor country" with a "Close - [Esc] x" button in the top right corner. Below the title bar are three tabs: "Field", "Lookup", and "Interface", with "Interface" being the active tab. The main content area of the dialog contains several settings:

- Alignment:** A dropdown menu with a gear icon on the left, showing "ALIGN_LEFT" and a downward arrow.
- Mask:** A text input field.
- TextArea:** A checkbox, currently unchecked.
- Do not sanitize:** A checkbox, currently unchecked.
- Placeholder:** A text input field.
- Help:** A large text area for entering help text.

At the bottom right of the dialog are two buttons: "OK [Ctrl+Enter]" and "Cancel".

- **Mask** - use this attribute to specify the *field_mask*
- **TextArea** - for text fields if this attribute is set the textarea element will be created for these fields in the *Edit Form Dialog*
- **Do not sanitize** - set this attribute to prevent default sanitizing of the field value, see *Sanitizing*
- **Alignment** - determines the alignment of text in the controls that display this field.
- **Placeholder** - use this attribute to specify the placeholder that will be displayed by the field input.
- **Help** - if any text / html-message is specified, a question mark will be displayed to the right of the input, so when the user moves the mouse pointer over this mark, a pop-up window appears displaying this message.

Interface tab for FILE field

Field Editor file Close - [Esc] x

Field Lookup **Interface**

Download btn ☒

Open btn ☒

Accept

Help

OK [Ctrl+Enter] Cancel

Name *

File 📁 ⬇️ ▶️

- **Download btn** - uncheck the box to hide the download button (middle)
- **Open btn** - uncheck the box to hide the open button (right)
- **Accept** - the attribute specifies the types of files that can be loaded. This is an *Accept string*.

Note: Please note that **Accept** attribute is required. Uploaded files are checked on the server against this attribute.

Interface tab for Image field

Field Editor photo Close - [Esc] x

Field Lookup **Interface**

View width Edit width

View height Edit height

Capture from camera ☐

Placeholder image

- **View width** - specifies the width of an image in pixels when it is displayed in the table of the view form. If it not specified the width is auto
- **View height** - specifies the height of an image in pixels when it is displayed in the table of the view form. If it not specified the height is auto
- **Edit width** - specifies the width of an image in pixels when it is displayed in the edit form. If it not specified the width is auto
- **Edit height** - specifies the height of an image in pixels when it is displayed in the edit form. If it not specified the height is auto
- **Capture from camera** - if this checkbox is set, the user will be able to capture image from camera.
- **Placeholder image** - double-click the image to set the placeholder image, that will be displayed when field image is not set. Hold Ctrl key and double-click the image to clear the placeholder image.

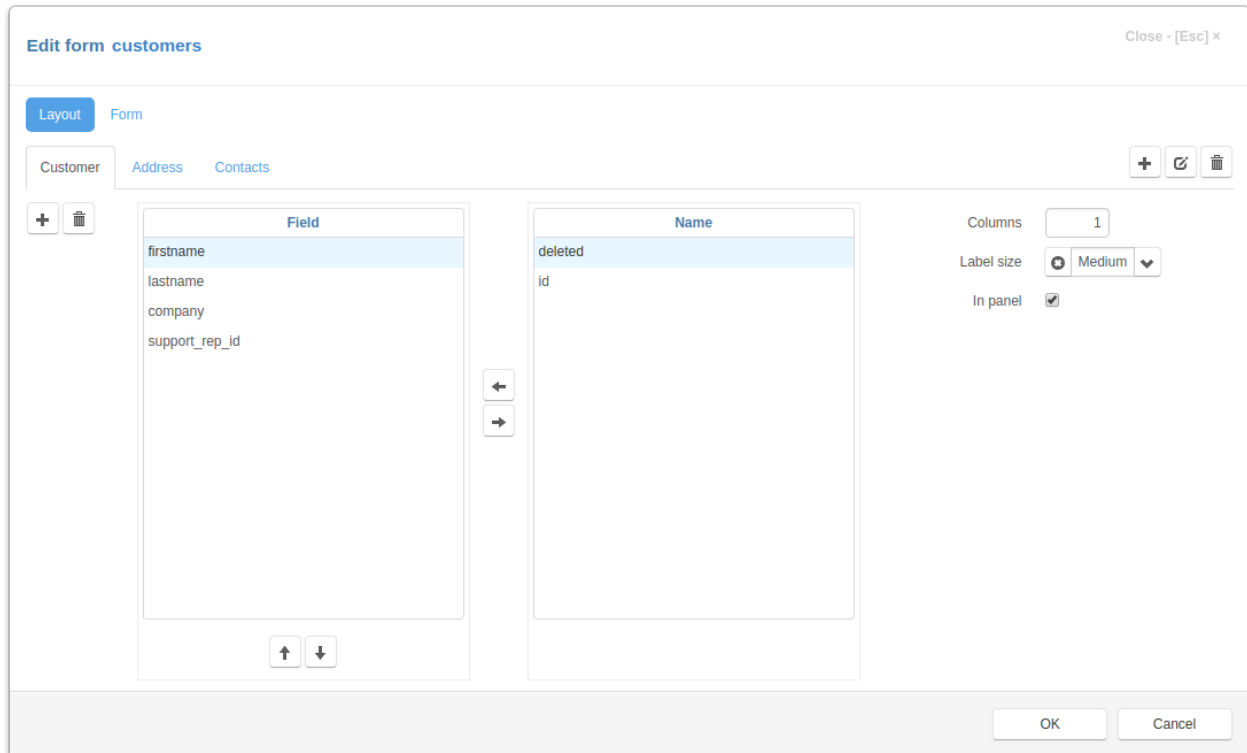
6.9.3 Edit Form Dialog

The **Edit Fields Dialog** opens when a developer selects the item in the Application builder and clicks the **Edit Form** button.

It has two tabs **Layout** and **Form**.

Layout tab

On the **Layout** tab, you can specify the fields that the user can edit, their order, create tabs and bands for grouping field inputs.



The **Layout** tab has two lists of fields. The left list contains the fields that were selected for editing. In the right list there are available fields that you can select.

To select a field, select it in the right list and use the **Left arrow** button in the center or press **Space** key on a keyboard.

To unselect a field, select it in the left list and use the **Right arrow** button in the center or press **Space** key on a keyboard.

To order the selected fields use the buttons that located below left list.

On the right side of the “Layout” tab are the controls that you can use to specify the display options for the fields selected for editing on the form.

- **Columns** - the number of columns that will be created for field inputs
- **Label size** - select a value that determines the size of the labels displayed to the right of the field input:
 - xSmall
 - Small
 - Medium

- Large
- xLarge

- **In panel** - if set, the div containing the inputs will have an inset effect

You can create tabs and bands and customize fields that you can edit on each tab or band.

On the right side of the tab there are three buttons for adding, editing or deleting tabs of the edit form.

On the left side of the tab there are two buttons for adding and deleting of bands.

Each tab can have several bands.

After creating tabs and bands, you can use field lists and controls on the right to customize the fields that will be edited on each tab and band.

Form tab

The screenshot shows a dialog box titled "Edit form customers" with a close button in the top right corner labeled "Close - [Esc] x". Inside the dialog, there are two tabs: "Layout" and "Form", with "Form" being the active tab. The "Form" tab contains several settings:

- Form border**: ☒
- Form header**: ☒
- History**: ☒
- Close button**: ☒
- Close on escape**: ☒
- Width**:
- Edit details**: with a button icon to the right and a folder icon to the right of the input field.
- Detail height**:
- Buttons on top**: ☐
- Modeless form**: ☐

At the bottom right of the dialog are "OK" and "Cancel" buttons.

On this tab are the controls that you can use to specify the options of the edit form

- **Form border** - if set, the border will be displayed around the form
- **Form header** - if set, the form header will be created and displayed containing form title and various buttons
- **History** - if set and *saving change history is enabled*, the history button will be displayed in the form header
- **Close button** - if set, the close button will be created in the upper-right corner of the form
- **Close on escape** - if set, pressing on the Escape key will close the form
- **Width** - an integer, the width of the modal form, if not set the value is 600 px
- **Edit details** - click the button to the right of the input field to select details, that will be available for editing in the edit form
- **Detail height** - an integer, the height of the details displayed in the edit form, if not set, the height of the detail table is 262px
- **Buttons on top** - if this check box is checked the buttons are displayed on the top of the view form, when form has a default form template

- **Modeless form** - if this check box is checked the form will be modeless, otherwise - modal.

Click the **OK** button to save to result or **Cancel** to cancel the operation.

After saving, you can see the changes by refreshing the project page.

6.9.4 View Form Dialog

The **View Form Dialog** opens when a developer selects the item in the Application builder and clicks the **View Form** button.

It has two tabs **Layout** and **Form**.

Layout tab

On the Layout tab, you can specify how the table is displayed in the view form of the item.

View form invoices Close - [Esc] x

Layout **Form**

Field	Width
date	
customer	
billing_address	
billing_city	
billing_state	
billing_country	
subtotal	80
tax	80
total	80

← →

Field
billing_postal_code
deleted
firstname
id
taxrate

Multiple selection ☐

Dbclick edit ☒

Number of rows

Height

Row lines

Selected row lines

Freeze columns

Sort fields selected: 2

Summary fields selected: 4

↑ ↓

OK Cancel

Setting table fields

The **Layout** tab has two lists of fields. The left list contains the fields that were selected be displayed in the table. In the right list there are available fields that you can select.

To select a field, select it in the right list and use the **Left arrow** button in the center or press **Space** key on a keyboard.

To unselect a field, select it in the left list and use the **Right arrow** button in the center or press **Space** key on a keyboard.

To order the selected fields use the buttons that located below left list.

You can specify the width of the selected columns. To do this, select the field and enter its width in the Width column. The value can be specified in any supported CSS unit, for example, in pixels - `px`, in percentage, relative to the parent element - `%`. The width specified as an integer value is interpreted as the width specified in pixels.

Examples of column width values:

- `100px`
- `100`
- `50%`
- `2cm`

Setting table options

On the right side of the “Layout” tab are the controls that you can use to specify the options of the table displayed in the view form:

- **Multiple selection** - if set, a leftmost column with check-boxes will be created to select records. So, that when a user clicks on the check-box, the value of the primary key field of the record will be added to or deleted from the *selections* attribute.
- **Dbclick edit** - if set, the edit form will be displayed when the user double-clicks on the table row.
- **Number of rows** - an integer number, if set, specifies the number of rows displayed by the table, otherwise, if **Height** is not specified, the application calculates the height of the table, based on the page height
- **Height** - an integer number, if set, specifies the height of the table in pixels, otherwise, if **Number of rows** is not specified, the application calculates the height of the table, based on the page height
- **Row lines** - an integer, specifying the number of lines of text displayed in a table row, if it is 0, the height of the row is determined by the contents of the row cells
- **Selected row lines** - an integer value, if **Row lines** is set and this value is greater than 0, it specifies the minimal number of lines of text displayed in the selected row of the table
- **Freeze columns** - an integer, if it is greater than 0, it specifies number of first columns that become frozen - they will not scroll when the table is scrolled horizontally.
- **Sort fields** - click the button to the right of the input field to open the list of fields and select the fields by which you can sort the contents of the table by clicking in the corresponding column header of the table.
- **Summary fields** - click the button to the right of the input field to open the list of fields and the fields for which the summary will be calculated and displayed in the corresponding column footer. For numeric fields sums will be calculated, for not numeric fields - the number of records.

You can get or change these values programmatically on the client by using the *table_options* attribute of the item

Form tab

On this tab are the controls that you can use to specify the options of the view form

- **Form border** - if set, the border will be displayed around the form
- **Form header** - if set, the form header will be created and displayed containing form title and various buttons
- **History** - if set and *saving change history is enabled*, the history button will be displayed in the form header
- **Refresh button** - if set, the refresh button will be created in the form header, that will allow users to refresh the page
- **Search** - if set, the search input will be created in the form header
- **Default search field** - click the button to the right of the input field to select a default search field
- **Filters** - if set and there are visible filters, the filter button will be created in the form header
- **Close button** - if set, the close button will be created in the upper-right corner of the form
- **Close on escape** - if set, pressing on the Escape key will close the form
- **Width** - an integer, the width of the modal form, if not set the value is 600 px
- **View details** - click the button to the right of the input field to select details, that will be displayed in the view form
- **Detail height** - an integer, the height of the details displayed in the view form, if not set, the height of the detail table is 232px
- **Buttons on top** - if this check box is checked the buttons are displayed on the top of the view form, when form has a default form template

You can get or change these values programmatically on the client by using the *view_options* attribute of the item

Click the **OK** button to save to result or **Cancel** to cancel the operation. After saving, you can see the changes by refreshing the project page.

6.9.5 Filters Dialog

Use **Filters Dialog** to create and modify item filters. See [Filters](#)

Caption	Name	Filter type	Field	Visible
From	invoicedate1	GE	date	x
to	invoicedate2	LE	date	x
Customer	customer	IN	customer	x

Buttons: Delete, Up arrow, Down arrow, Edit, New

To add or edit a filter click on the appropriate button on the form. The following form will appear:

The screenshot shows a 'Filters' dialog box with the following fields and values:

- Field:** A dropdown menu showing 'date' with a folder icon to its right.
- Caption *:** A text input field containing 'From'.
- Name *:** A text input field containing 'invoicedate1'.
- Filter type:** A dropdown menu showing 'GE' with a downward arrow to its right.
- Placeholder:** An empty text input field.
- Help:** An empty text input field.
- Visible:** A checked checkbox.

At the bottom of the dialog are 'OK' and 'Cancel' buttons.

Fill in the following fields:

- **Field** - the field which will be used to filter records.
- **Caption** - the filter name that appears to users.
- **Name** - the name of the filter that will be used in programming code to get access to the filter object. It should be a valid python identifier.
- **Filter type** - select filter type.
- **Placeholder** - use this attribute to specify the placeholder that will be displayed by the field input.
- **Help** - if any text / html-message is specified, a question mark will be displayed to the right of the input, so when the user moves the mouse pointer over this mark, a pop-up window appears displaying this message.

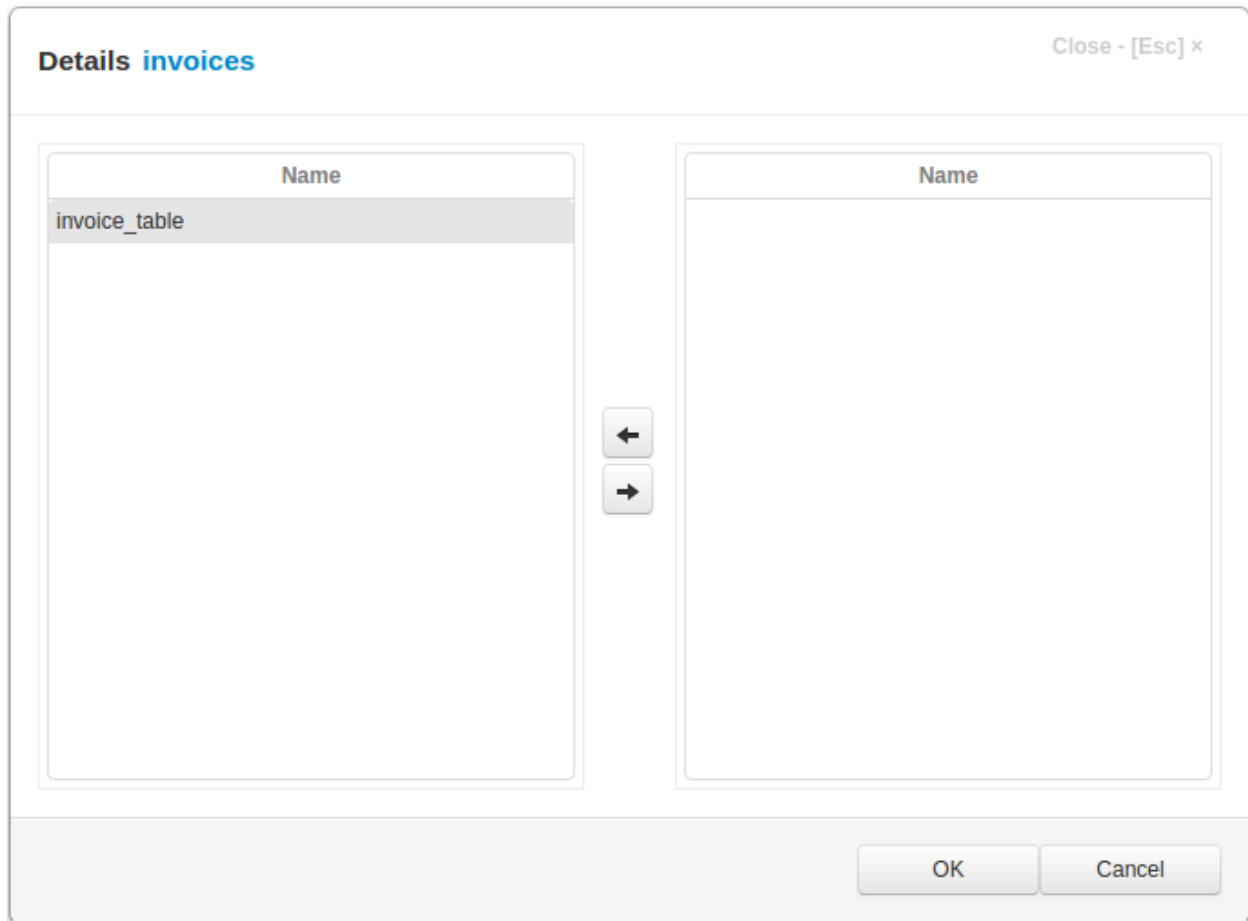
Help - if any text / html-message is specified, a question mark will be displayed to the right of the input, so when the user moves the mouse pointer over this label, a pop-up window appears displaying this message.

- **Visible** - if this checkbox is not checked, this filter will not be displayed in the item Filters dialog.

Use the up and down arrows to place the filters in the order in which they will be displayed. See [create_filter_inputs](#)

6.9.6 Details Dialog

Use this dialog to setup details of an item. See [Details](#).



The **Details Dialog** has two panels. The left panel lists details that have been added. The right panel have available detail items that could be added as details.

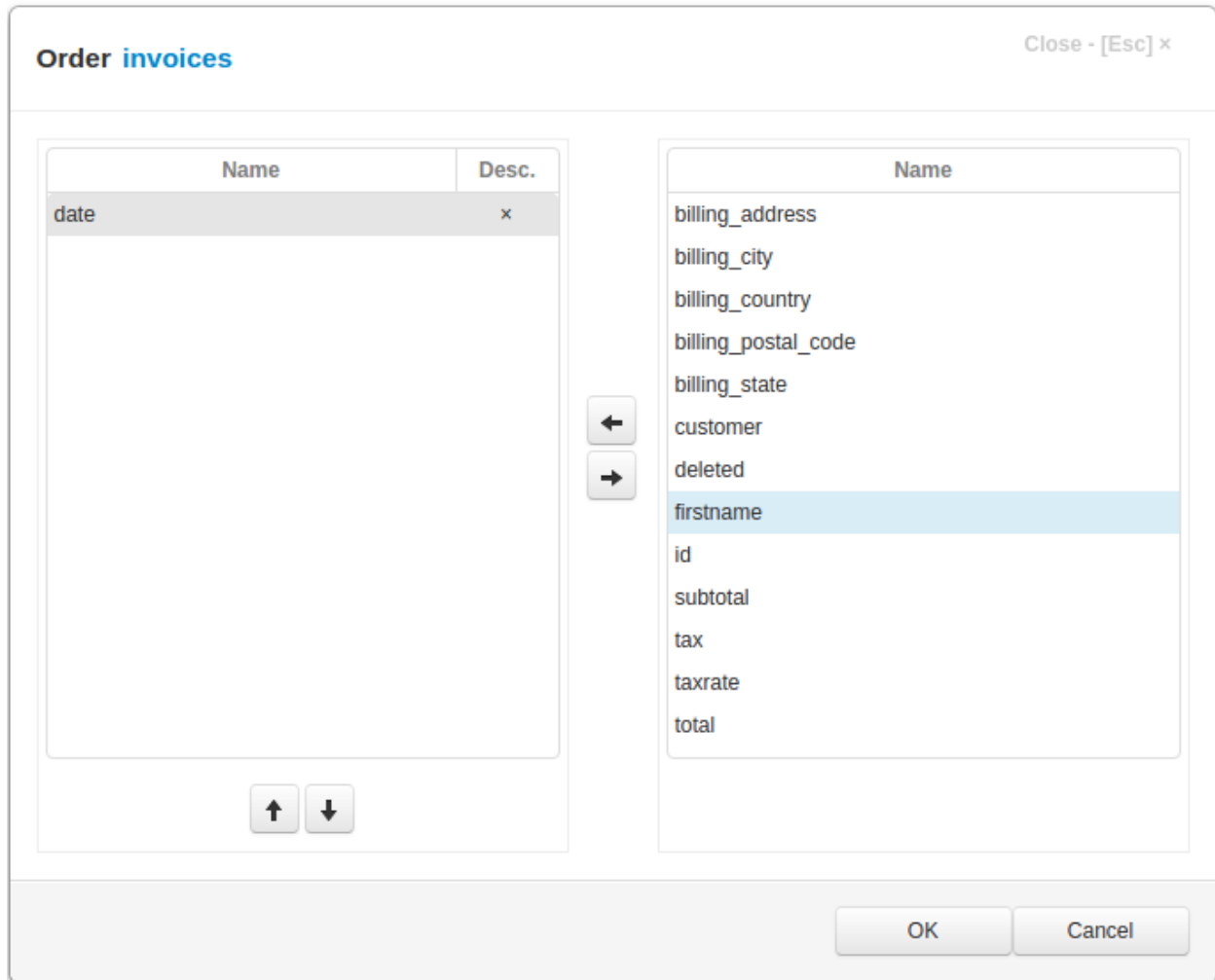
To add a detail item as detail, select it in the right panel and use the **Left arrow** button in the center or press **Space** key on a keyboard.

To remove a detail, select it in the left panel and use the **Right arrow** button in the center or press **Space** key on a keyboard.

Click the **OK** button to save to result or **Cancel** to cancel the operation.

6.9.7 Order Dialog

The **Order Dialog** opens when a developer selects the item in the Application builder (see [Items](#)) and clicks on the **Order** button to specify how records will be ordered by default. See [open](#) method



The **Order Dialog** has two panels. The left panel lists the fields that have been selected. The right panel have available fields that could be selected.

To select a field, select it in the right panel and use the **Left arrow** button in the center or press **Space** key on a keyboard.

To unselect a field, select it in the left panel and use the **Right arrow** button in the center or press **Space** key on a keyboard.

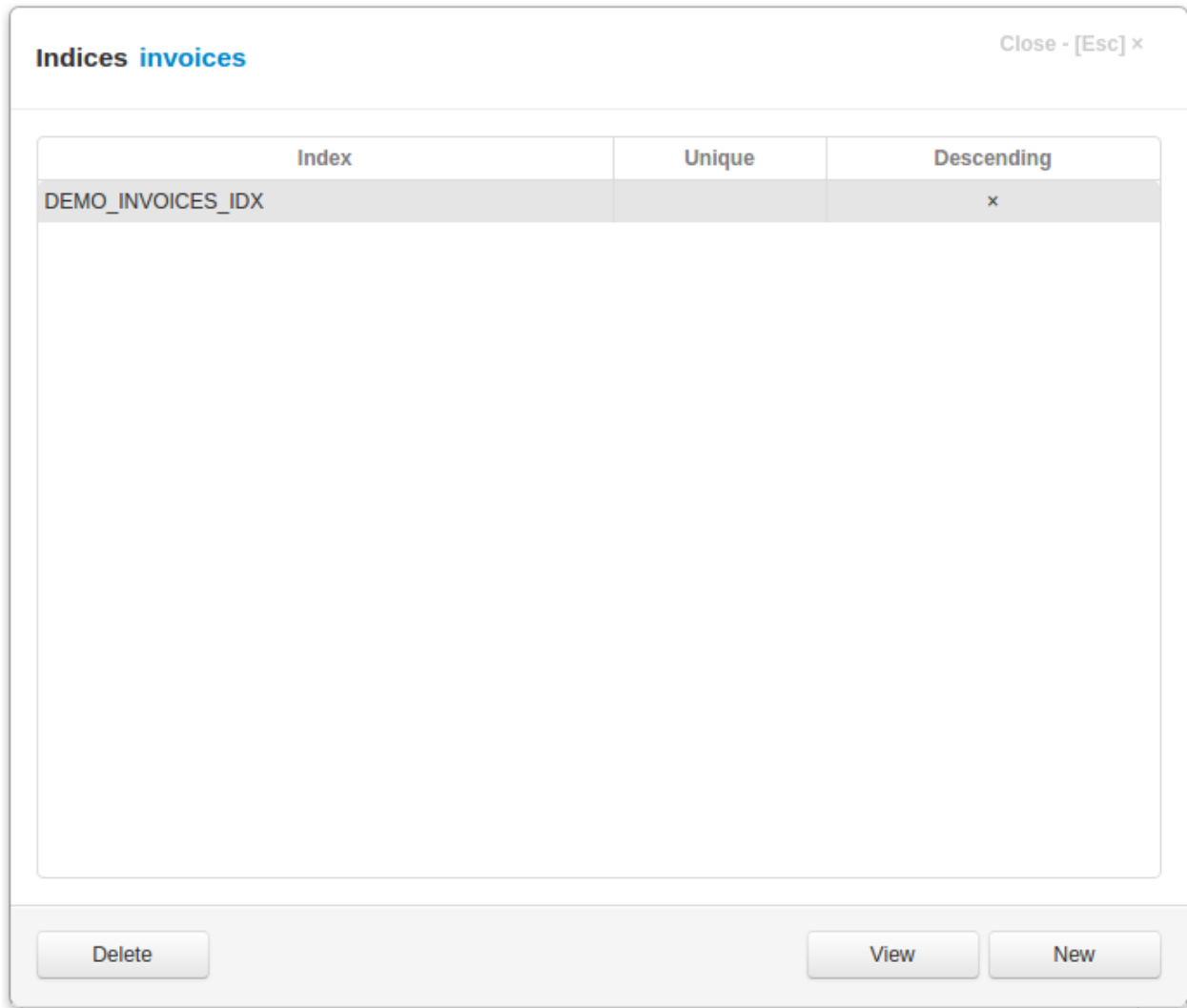
To order the selected fields use the buttons that located below left panel.

Click the **Desc** column to set descending/ascending sorting order for the field.

Click the **OK** button to save to result or **Cancel** to cancel the operation.

6.9.8 Indices Dialog

The **Indices Dialog** lists the indices that were created for the item table in the project database.



To delete an index click the **Delete** button. The application will generate the SQL query to drop the index and execute it on the server.

To create a new index click the **New** button. The following dialog will appear:

Editor Close - [Esc] x

Index

Descending ☒

Name

- deleted
- invoicedate

Name

- billing_address
- billing_city
- billing_country
- billing_postal_code
- billing_state
- customer
- firstname
- id
- subtotal
- tax
- taxrate
- total

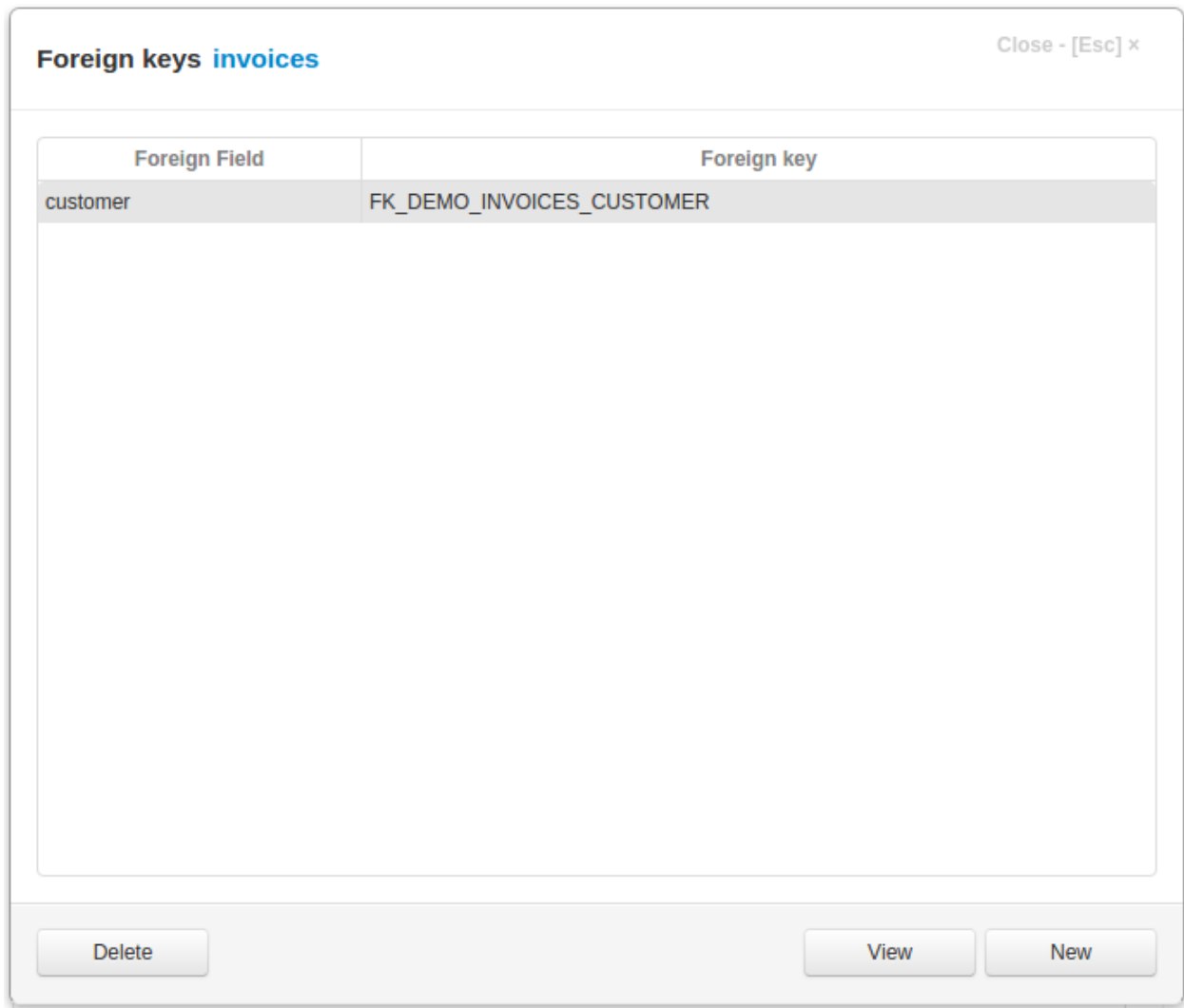
Specify the fields to create an index on, by using left and right arrow buttons. Check the **Descending** checkbox if you want to create a descending index. If necessary, change the name of the index.

Click the **OK** button to create the index. The application will generate the SQL query to create the index and execute it on the server.

Click **Cancel** button to cancel the operation.

6.9.9 Foreign Keys Dialog

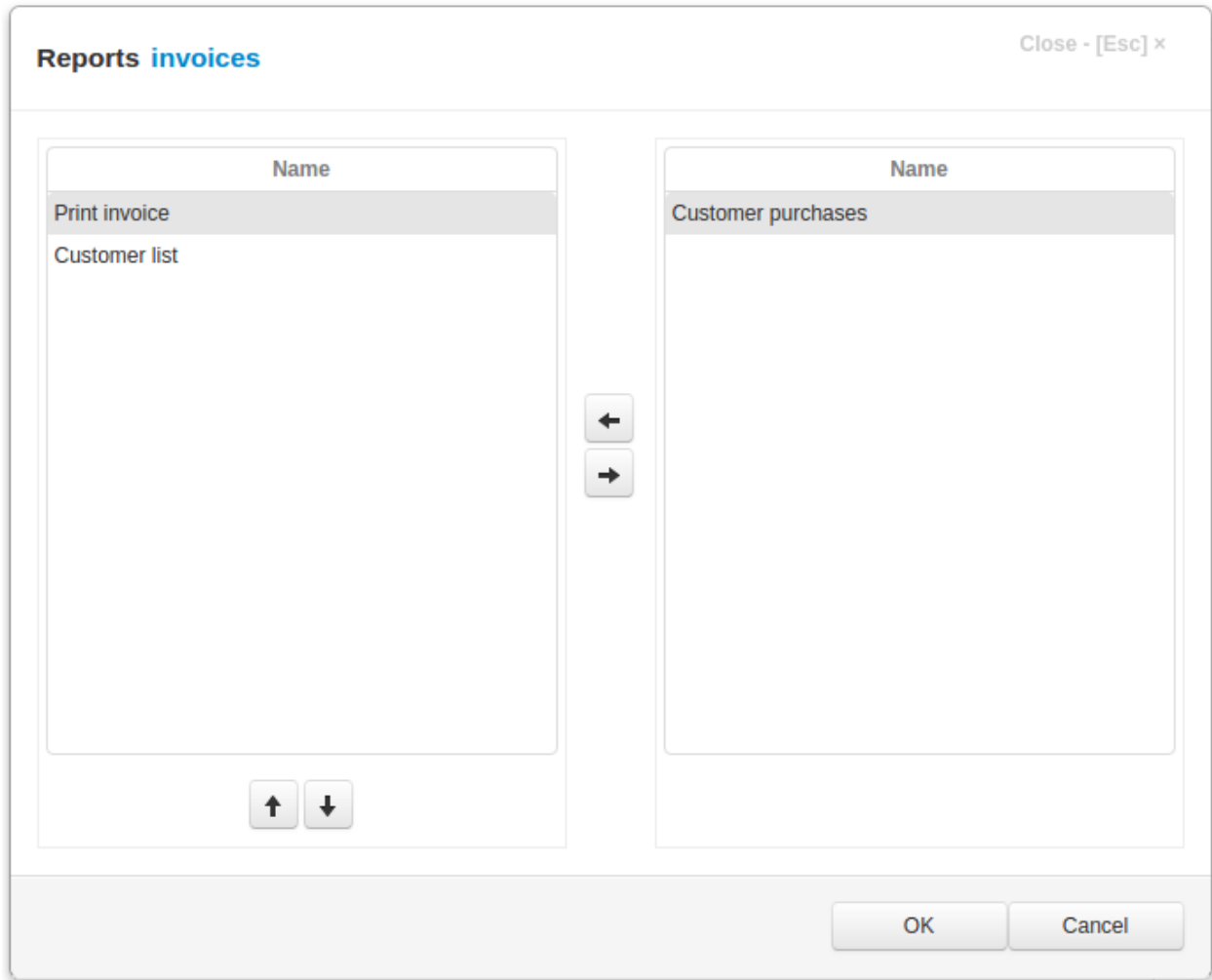
If an item has a lookup field, and in the definition of lookup item the soft delete attribute is not set, in order to maintain the integrity of the data, we can create a foreign key. See [Foreign keys topic](#) in FAQ



To do so click the **New** button, select the field and click **OK**.

6.9.10 Reports Dialog

The **Reports Dialog** opens when a developer selects the item in the Application builder (see *Items*) and clicks on the **Order** button to specify reports that could printed for the item. A new project code has a function that can be used to print the reports.



The **Reports Dialog** has two panels. The left panel lists the reports that have been selected. The right panel have available reports that could be selected.

To select a report, select it in the right panel and use the **Left arrow** button in the center or press **Space** key on a keyboard.

To unselect a report, select it in the left panel and use the **Right arrow** button in the center or press **Space** key on a keyboard.

To order the selected reports use the buttons that located below left panel.

Click the **OK** button to save to result or **Cancel** to cancel the operation.

6.10 Details

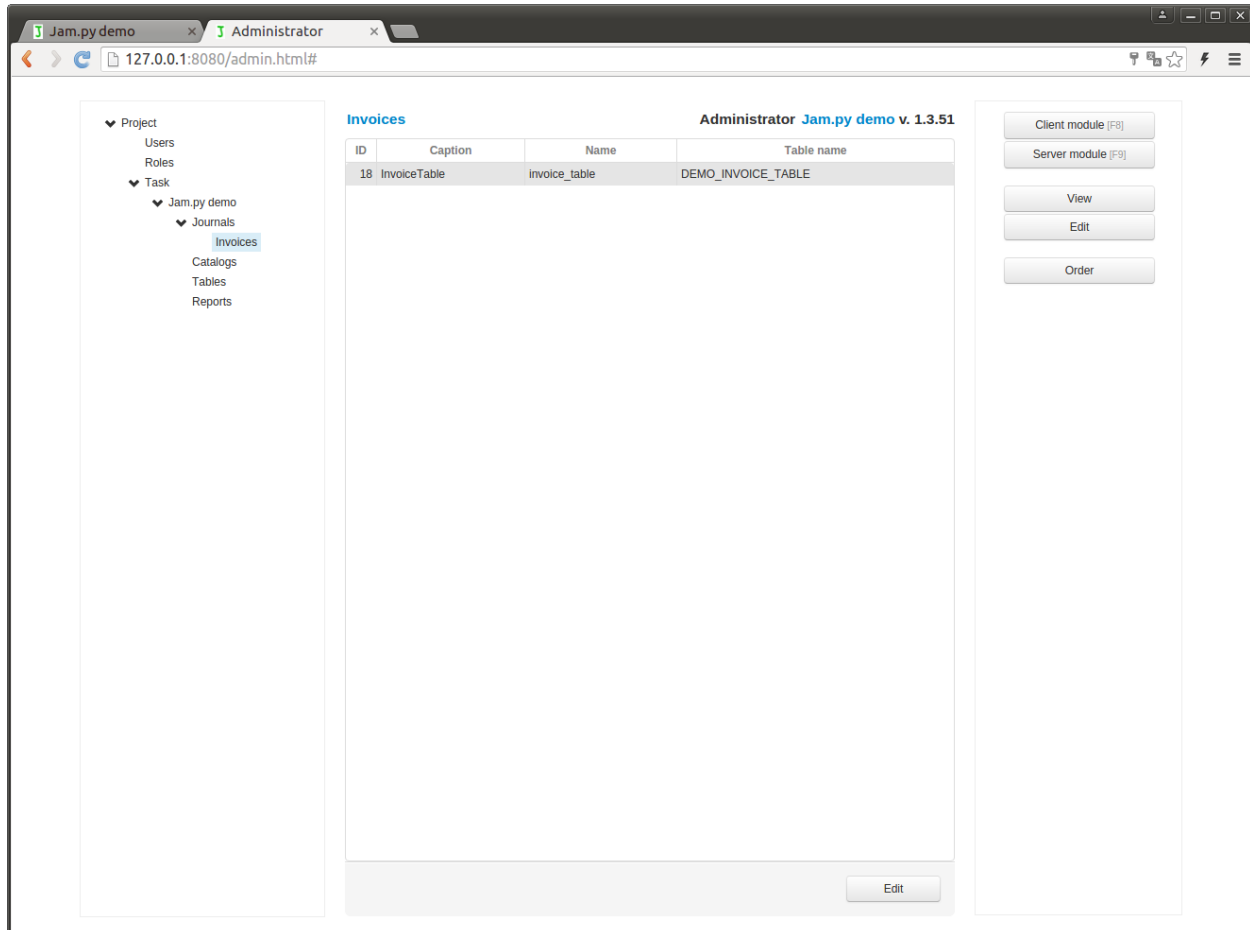
To work with a detail of an item, expand a group node that owns the item and select that item in the tree. In the center of the Application builder all details of this item will be displayed.

The right panel of the page have following buttons:

- **Client module** - click on this button to open the *Code editor* to edit client module of a detail, see *Working with modules*.

- **Server module** - click on this button to open the *Code editor* to edit server module of a detail, see *Working with modules*.
- **View Form** - use this button to invoke the *View Form Dialog* to set the fields to be displayed in tables on the client and their order, by default. See *create_table* method
- **Edit Form** - use this button to invoke the *Edit Form Dialog* to set the fields to be displayed in edit forms on the client and their order, by default. See *create_inputs* method.
- **Order** - use this button to invoke the *Order Dialog* to specify how records will be ordered by default. See *open* method

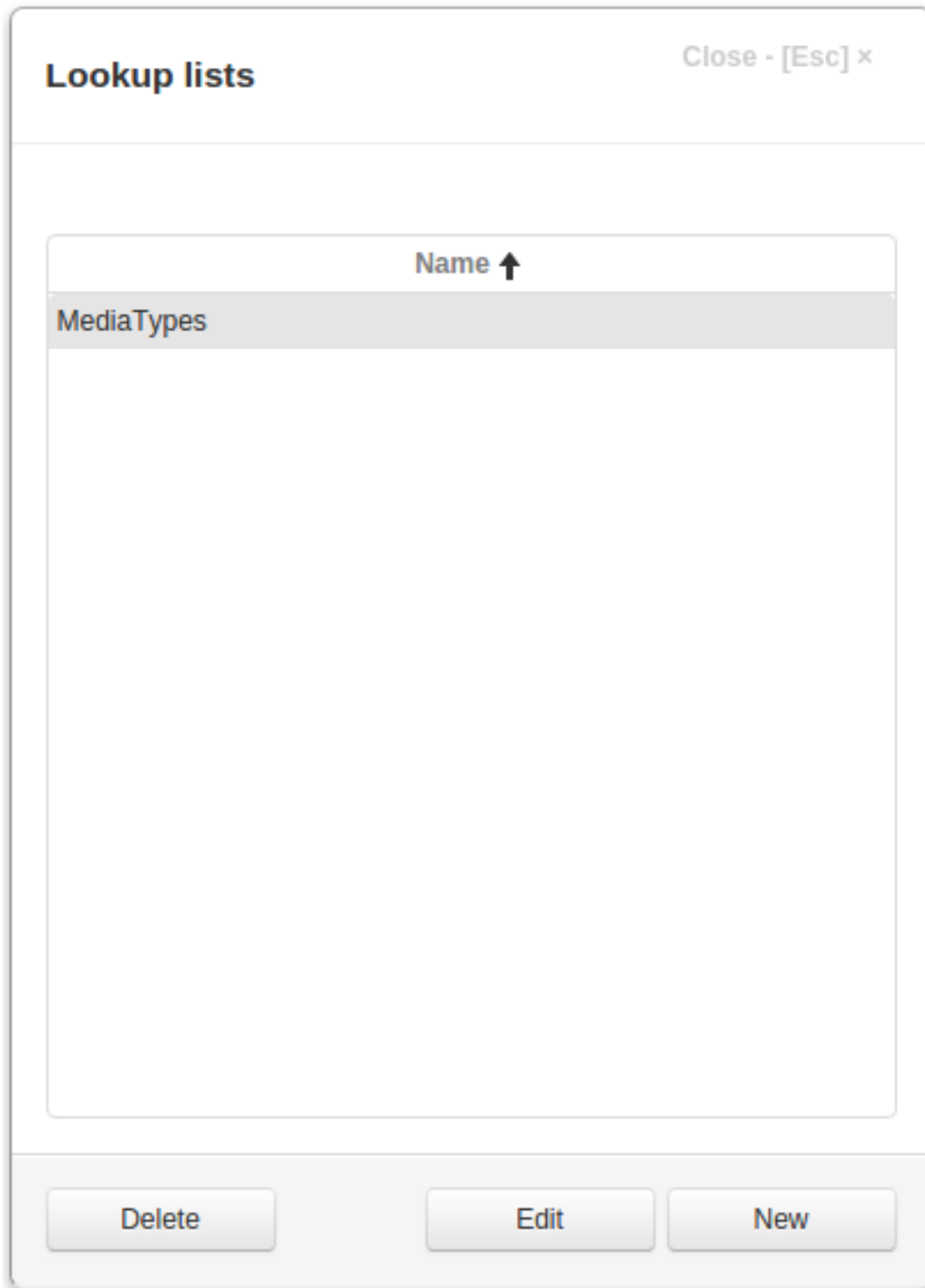
Use **Edit** button at the bottom of the page to change `item_name` or `caption` of a *detail*.



6.11 Lookup List Dialog

Lookup list is a list of integer-text pairs that can be used as a datasource for *lookup fields*.

Note: The length of the lookup list should not exceed 10



Click on the **Edit/New** buttons to edit/create a lookup list.

Lookup lists Close - [Esc] x

Name

Value	Lookup value
1	MPEG audio file
2	Protected AAC audio file
3	Protected MPEG-4 video file
4	Purchased AAC audio file
5	AAC audio file

Delete [Ctrl+Del]

Edit

New [Ctrl+Ins]

OK

Cancel

Then use the **Edit/New** buttons to edit/add a lookup pairs to the list.

Lookup item Close - [Esc] x

Value

Lookup value

OK

Cancel

6.12 Intergation with existing database

You can use Jam.py with existing database, that is supported by the framework.

- Create a new project with existing database.
- If you want to import tables in catalogs or journals groups, delete *Common fields*:
Select Groups node in the project tree, dbl click corresponding group and delete common fields.
Or create new empty groups.
- Select Project node and click Database button. Set *DB manual mode* to true.
- Select group you want to import a table to and click Import button.
- In the form that will appear dbl click on the table to import it.
- In the *Item Editor Dialog* check that all fields have valid types. If field type is displayed in the red, try to select appropriate type.

You can import a subset of fields in the table.

Before saving, specify the primary key field for the item and generator name, if necessary.

- After saving the imported item, go to the project page and check how it is displayed.
- After importing several tables, you can specify lookup fields (in DB manual mode).

Note: Please, do be very careful when performing this operations.

When DB manual mode is removed any changes to the item will be reflected in the corresponding DB table. If you delete the item, the table will be dropped from the database.

Note: The database table to be imported must have a primary key with one field.

Note: Binary fields must not be imported.

Note: This is a new feature, so if you have some comments, suggestions or found some bugs please send a message.

6.13 Saving audit trail/change history made by users

To save change history made by users to must specify the item that will store them.

To do so, open project parameters and click the button to the right of the History item input:

Parameters Close - [Esc] x

General

Interface


Safe mode


☒

Debugging

☐

Language *

 English en-US



Persistent connection

☒

Connection pool size

Compressed JS, CSS files

☐


All JS modules in a single file


☒

Dynamic JS modules loading


☐


History





Lock item





Session timeout (seconds)

Session ignore change ip

☒

Max content length (MB)

Delete reports after (hours)

Version

OK

Cancel

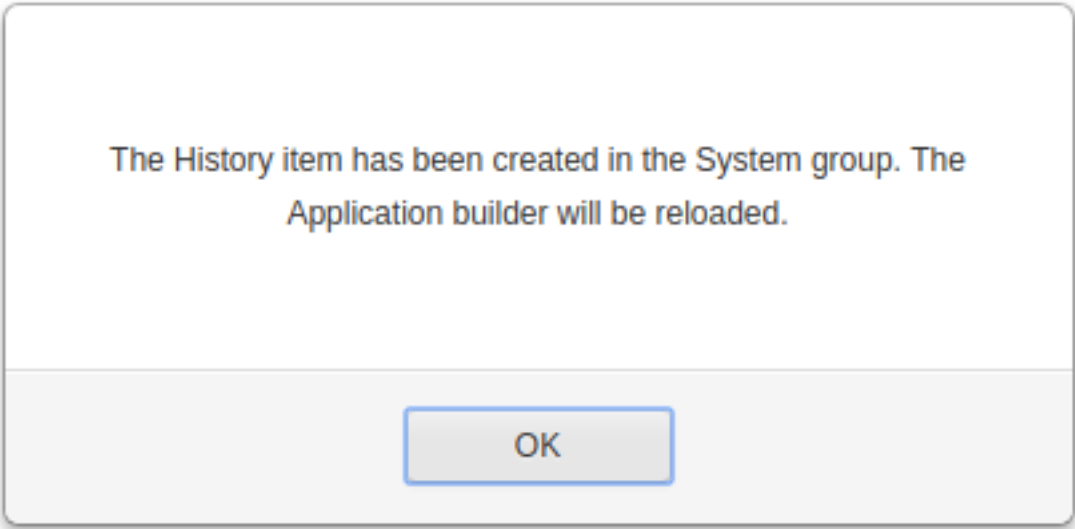
In the dialog that will appear click on the Create history item button

The screenshot shows a modal dialog box titled "Items". The title bar includes the text "Close - [Esc] x". The main area of the dialog contains a table with two columns: "Caption" and "Name". The table is currently empty. At the bottom of the dialog, there is a button labeled "Create history item".

Caption	Name
---------	------

Create history item

The following message will appear when the item will be created:



After that you have to set **Keep history** attribute of an item to save the history its changes:

Item Editor invoices

Close - [Esc] x

Caption *

Invoices

Name *

invoices

Table name

DEMO_INVOICES

Primary key field

id

Deleted flag

deleted

Visible☒

Soft delete☒

Virtual table☐

History☒

Edit lock☒

:Caption	↑Name	:DB field name	:Type	:Size	:Required	:Read	:Lookup	:Lookup	:Master	:Typeahead	:Lookup value list
Billing Address	billing_address	BILLING_ADDRESS	INTEGER				customers	address	customer		
Billing City	billing_city	BILLING_CITY	INTEGER				customers	city	customer		
Billing Country	billing_country	BILLING_COUNTRY	INTEGER				customers	country	customer		
Billing Postal Code	billing_postal_code	BILLING_POSTAL_CODE	INTEGER				customers	postalcode	customer		
Billing State	billing_state	BILLING_STATE	INTEGER				customers	state	customer		
Customer	customer	CUSTOMER	INTEGER		x		customers	lastname		x	
Invoice Date	date	DATE	DATE		x						
Customer FirstName	firstname	FIRSTNAME	INTEGER				customers	firstname	customer		
SubTotal	subtotal	SUBTOTAL	CURRENCY			x					
Tax	tax	TAX	CURRENCY			x					
Tax Rate	taxrate	TAXRATE	FLOAT								
Total	total	TOTAL	CURRENCY		x	x					

Delete [Ctrl+Del]

Edit

New [Ctrl+Ins]

OK [Ctrl+Enter]

Cancel [Esc]

To see the history of changes of a record click the icon to the left of the close button on the right part of the header of

the edit form.

The screenshot shows the 'Invoices' application window. A modal window titled 'History: Invoices' is open, displaying a list of history entries. The first entry is '04.12.2017 15:20:46: Modified by Admin' and the second is '03.12.2017 11:06:12: Created by Admin'. The modal also shows field updates: 'Field SubTotal: new value: 7,92', 'Field Tax: new value: 0,40', and 'Field Total: new value: 8,32'. The background application shows a form for editing an invoice, with fields for Customer, Billing Address, City, State, Country, and Postal Code. A table of items is visible on the right, with columns for UnitPrice, Amount, Tax, and Total.

UnitPrice	Amount	Tax	Total
0,99	1,98	0,10	2,08
0,99	0,99	0,05	1,04
0,99	0,99	0,05	1,04
0,99	0,99	0,05	1,04
0,99	0,99	0,05	1,04
0,99	0,99	0,05	1,04
0,99	0,99	0,05	1,04

Or you can do it using the [show_history](#) method

Note: Changes are saved when dataset changes are applied to the database using apply method ([client/server](#)). Changes to database made with custom SQL requests are not saved in the history.

Note: These changes can significantly increase the size of the database. Please be careful.

6.14 Record locking

In Jam.py, application you can implement a record locking while users concurrently edit a record.

Jam.py uses optimistic locking model, also referred to as optimistic concurrency control.

When an application executes the [edit_record](#) method, it receives the current version of the record from the server and saves it. When the user starts saving the record, the server application checks the current version of the record. If it differs from the stored value (another user changed it while the record were being edited), the application warns the user and prohibits saving.

This record locking mechanism is very easy to implement.

To do so you need to create an item that will store record version.

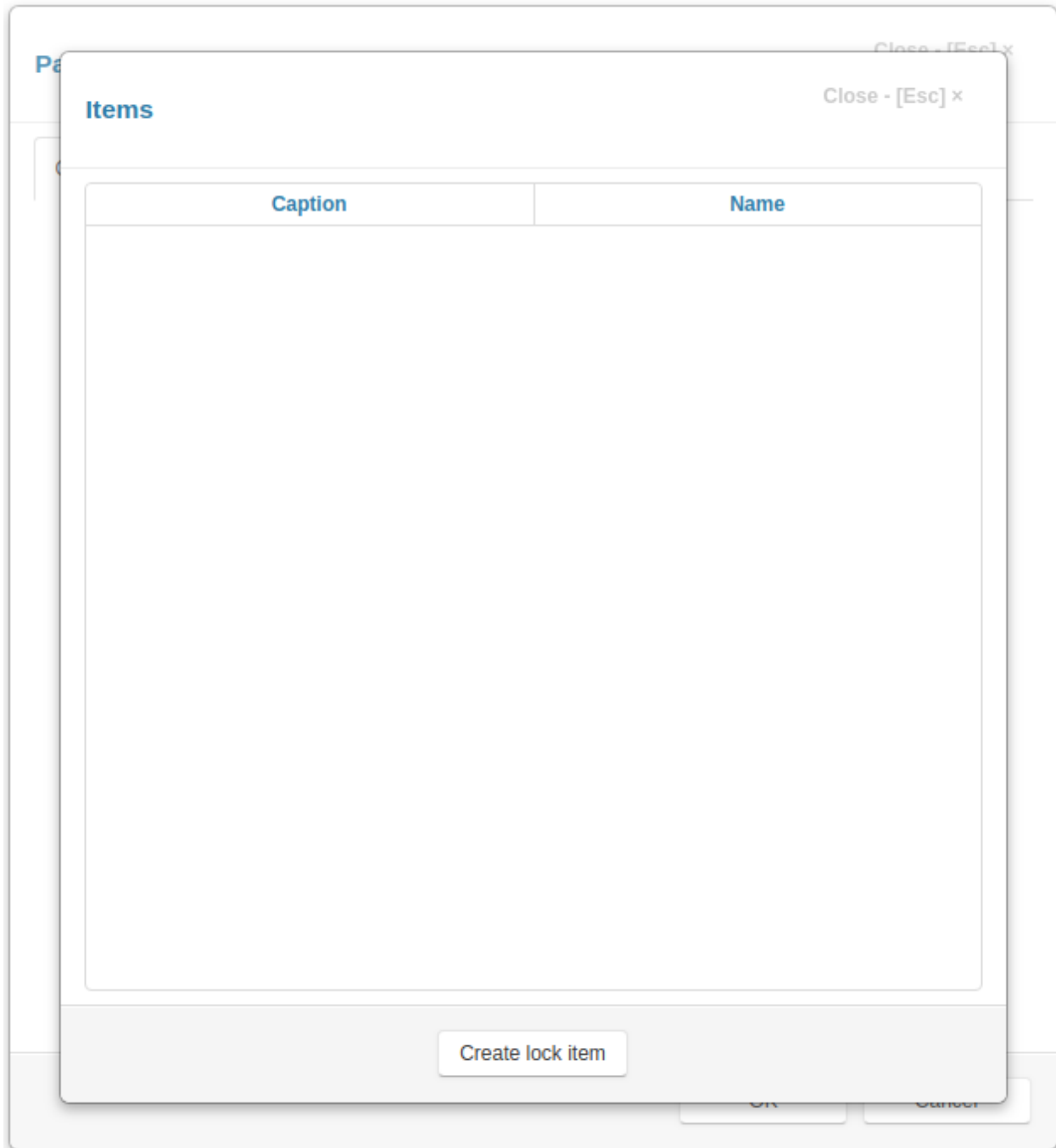
Open *project parameters* and click the button to the right of the **Lock item** input:

The screenshot shows the 'Parameters' dialog box with the 'Interface' tab selected. The dialog has a title bar with 'Parameters' and a 'Close - [Esc] x' button. Below the title bar are two tabs: 'General' and 'Interface'. The 'Interface' tab contains the following settings:

- Safe mode: ☒
- Debugging: ☐
- Language *: English en-US (with a folder icon button to the right)
- Persistent connection: ☒
- Connection pool size: 3
- Compressed JS, CSS files: ☐
- All JS modules in a single file: ☒
- Dynamic JS modules loading: ☐
- History: (empty text box with a folder icon button to the right)
- Lock item: (empty text box with a folder icon button to the right)
- Session timeout (seconds): 86400
- Session ignore change ip: ☒
- Max content length (MB): (empty text box)
- Delete reports after (hours): 24
- Version: 1.5.28

At the bottom right of the dialog are 'OK' and 'Cancel' buttons.

In the dialog that will appear click on the **Create lock item** button:



After that you must set **Edit lock** attribute in the *Item Editor Dialog*:

Item Editor invoices
Close - [Esc] x

Caption * Invoices
Name * invoices
Table name DEMO_INVOICES
Primary key field id
Deleted flag deleted

Visible ☒
Soft delete ☒
Virtual table ☐
History ☒
Edit lock ☒

:Caption	↑Name	:DB field name	:Type	:Size	:Required	:Read	:Lookup	:Lookup	:Master	:Typeahead	:Lookup value list
Billing Address	billing_address	BILLING_ADDRESS	INTEGER				customers	address	customer		
Billing City	billing_city	BILLING_CITY	INTEGER				customers	city	customer		
Billing Country	billing_country	BILLING_COUNTRY	INTEGER				customers	country	customer		
Billing Postal Code	billing_postal_code	BILLING_POSTAL_CODE	INTEGER				customers	postalcode	customer		
Billing State	billing_state	BILLING_STATE	INTEGER				customers	state	customer		
Customer	customer	CUSTOMER	INTEGER		x		customers	lastname		x	
Invoice Date	date	DATE	DATE		x						
Customer FirstName	firstname	FIRSTNAME	INTEGER				customers	firstname	customer		
SubTotal	subtotal	SUBTOTAL	CURRENCY			x					
Tax	tax	TAX	CURRENCY			x					
Tax Rate	taxrate	TAXRATE	FLOAT								
Total	total	TOTAL	CURRENCY		x	x					

Delete [Ctrl+Del]
Edit
New [Ctrl+Ins]
OK [Ctrl+Enter]
Cancel [Esc]

6.15 Language support

Use Language Dialog to add, select and change your language.

Languages

Close - [Esc] x

Language↑

English en-US

French fr-FR

Hungarian hu-HU

Portuguese pt-PT

Russian ru-RU

Serbian sr-RS

Select

Edit

New

Import

Export

6.15.1 Language locale

Use language locale to set up how the field value will be displayed. See *display_text*

Languages

Close - [Esc] x

Before adding or editing a language, please read this [information](#).

Language * Serbian

Country * SERBIA

Locale

Translation

Decimal point

,

Monetary decimal point

,

Monetary thousands separator

.

Currency symbol

RSD

Number of fractional digits

2

Currency symbol precedes the value (positive values)

☐

Currency symbol precedes the value (negative values)

☐

Currency symbol is separated by a space (positive values)

☒

Currency symbol is separated by a space (negative values)

☒

Symbol for a positive monetary value

Symbol for a negative monetary value

-

The position of the sign (positive values)

1

The position of the sign (negative values)

1

Date format string

%d.%m.%Y

Date and time format string

%d.%m.%Y %H:%M

6.15.2 Language translation

See *Language translation*

Languages
Close - [Esc] x

Before adding or editing a language, please read this [information](#).

Language * Serbian
Country * SERBIA

Locale
Translation

Key	English	Translation
invalid_bool	%s invalid value - must be a boolean	%s pogrešna vrednost - mora biti logička vrednost
report_params	Report params	Parametri izveštaja
foreign_index	Foreign Index	Strani indeks
help	Help	Pomoć
can_delete	Can delete	Brisanje
import_checking_integrity	Import: checking data integrity	Uvoz: Provera integriteta podataka
size_is_required	The field size is required for text fields.	Veličina polja je zahtevana za tekst polja.
import_analyzing	Import: analyzing changes	Uvoz: Analiziranje promena
invalid_field_name	The field name is invalid.	Ime polja je pogrešno.
log_out	Log out	Odjavi se
filters	Filters	Filteri
error_modifying_table	Error while modifying table: %s	Greška pri promeni tabele: %s
yes	Yes	Da
invalid_int	%s invalid value - must be an integer	%s pogrešna vrednost - mora biti ceo broj
find	Find	Nadji
alignment	Alignment	Poravnanje
can_create	Can create	Kreiranje
tables	Details	Detalji

6.16 Language translation

All language translations are stored in the langs.sqlite database in the “jam” folder in the package.

Note: Therefore, if you made some changes to the translation database and installed a new version of the package, you will use the translation database of this package where **there will be no changes made by you**.

Please, export your translation to a file!!!

If you want your language translation to be included to Jam.py package, export it to a file and send it me, Andrew Yushev.

Please note that Jam.py is constantly evolving and by submitting your translation you agree to make the necessary changes in the future. If you don't mind you will be included to the contributors list.

Note: Do not change the following symbols `%`, `%(item)s`, `%(field)s`, `%(filters)s`

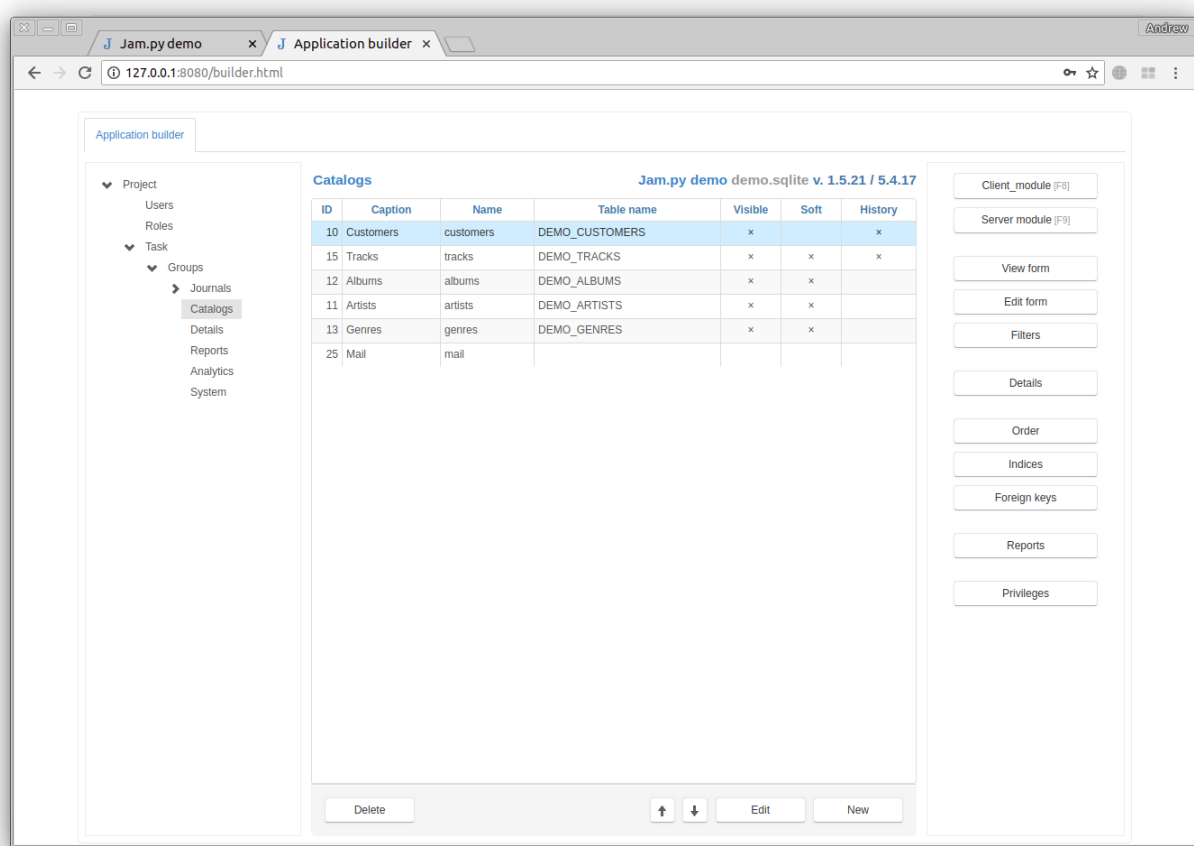
For example

english:

Can't delete the field `%(field)s`. It's used in field definitions:`%(fields)s`

ru^ssian translation:

%(field)s. :%(fields)s



Server side is implemented in Python and uses Werkzeug library, the client side in JavaScript and uses JQuery and Bootstrap

7.1 Client side (javascript) class reference

All objects of the framework represent a *task tree*. Bellow is classes for each kind of task tree objects:

7.1.1 AbstractItem class

```
class AbstractItem()
```

domain: client

language: javascript

AbstractItem class is the ancestor for all item objects of the *task tree*

Below the attributes and methods of the class are listed.

Attrubutes

ID

ID

domain: client

language: javascript

class *AbstractItem*

Description

The ID attribute is the unique in the framework id of the item

The ID attribute is most useful when referring to the item by number rather than name. It is also used internally.

item_caption

item_caption

domain: client

language: javascript

class *AbstractItem*

Description

Item_caption attribute specifies the name of the item that appears to users

item_name

item_name

domain: client

language: javascript

class *AbstractItem*

Description

Specifies the name of the item as referenced in code. Use item_name to refer to the item in code.

item_type

item_type

domain: client

language: javascript

class *AbstractItem*

Description

Specifies the type of the item.

Use the type attribute to get the type of the item. It can have one of the following values:

- “task”,
- “items”,
- “details”,

- “reports”,
- “item”,
- “detail_item”,
- “report”,
- “detail”

items

items

domain: client

language: javascript

class *AbstractItem*

Description

Lists all items owned by the item.

Use items to access any of the item owned by this object.

owner

Indicates the item that owns this item.

owner

domain: client

language: javascript

class *AbstractItem*

Description

Use owner to find the owner of an item.

task

Indicates the root of the *task tree* that owns this item.

task

domain: client

language: javascript

class *AbstractItem*

Description

Use task attribute to find the root of the *task tree* of which the item is a member.

Mehods

abort

abort (*message*)

domain: client

language: javascript

class *AbstractItem*

Description

Use **abort** method to throw exception.

It can be usefull when you need to abort execution of some 'on_before' events.

Example

The following code will throw exception with the text:

execution aborted: invoice_table - a quantity value is required

```
function on_before_post(item) {  
    if (item.quantity.value === 0) {  
        item.abort('a quantity value is required');  
    }  
}
```

alert

alert (*mess, options*)

domain: client

language: javascript

class *AbstractItem*

Description

Use the **alert** method to create a pop-up message in the upper-right corner application that disappears after the first click on the page.

The *mess* parameter specifies the text that will be displayed.

The *options* parameter is an object with the following attributes:

- `type` - indicates the type of the message - its font, background color and header text, if it is not specified in the header parameters. This must be one of the following:

- 'info',
- 'error',
- 'success'

default value is 'info'

- `header` - specifies the header of the alert
- `pulsate` - if true, the header will pulsate, the default value is true
- `show_header` - if false, the header will not be displayed.

The methods `alert_error` and `alert_success` are the same as `alert` with the corresponding `type` options.

Example

```
item.alert_error('Failed to send the mail: ' + err);
item.alert('Successfully sent the mail');
```

can_view

can_view()

domain: client

language: javascript

class *AbstractItem*

Description

Use **can_view** method to determine if a user have a right to get access to an *item* dataset or to see report generated by *report* when the project *Safe mode parameter* is set. If the project *Safe mode parameter* is not set the method always returns true.

The user privileges are set in the *roles node* of the project tree.

Example

```
if (item.visible && item.can_view()) {
    $('#submenu')
        .append($('- </li>'))
        .append(
            $('

```

each_item

each_item (*function(item)*)

domain: client

language: javascript

class *AbstractItem*

Description

Use each_item method to iterate over *items* owned by this object.

The each_item() method specifies a function to run for each child item (child item is passed as a parameter).

You can break the each_item loop at a particular iteration by making the callback function return false.

Example

The following code will output all catalogs of the project in a browser console:

```
function on_page_loaded(task) {  
    task.catalogs.each_item(function(item) {  
        console.log(item.item_name);  
    })  
}
```

hide_message

hide_message (*form*)

domain: client

language: javascript

class *AbstractItem*

Description

Use **hide_message** method to close a modal form created by *message* method

The **form** parameter is a JQuery object returned by *message* method.

item_by_ID

item_by_ID (*ID*)

domain: client

language: javascript

class *AbstractItem*

Description

`item_by_ID` searches among all items of the project *task tree*, starting with the current item, for an item whose *ID* attribute is equal to the ID parameter.

load_module

load_module (*callback*)

domain: client

language: javascript

class *AbstractItem*

Description

Use **load_module** method to dynamically load javascript file of an item module, before executing callback.

The method checks whether the module has been loaded, if not, loads the module from the server, initializes the item and then executes the **callback** function, otherwise just the **callback** function is executed. The item is passed to the callback function as a parameter.

The request to the sever is executed asynchronously.

Example

Bellow, the `do_some_work` function is executed only when an item module has been loaded:

```
function some_work(item) {
    item.load_module(do_some_work);
}

function do_some_work(item) {
    // some code
}
```

See also

Working with modules

load_modules

load_script

load_modules

load_modules (*module_array*, *callback*)

domain: client

language: javascript

class *AbstractItem*

Description

Use **load_modules** method to dynamically load specified modules before executing the **callback**.

The method works the same way as [load_module](#), only loads and initializes all modules of items specified in the **module_array**.

Example

Bellow, the `do_some_work` function is executed only when modules of the item and its owner has been loaded:

```
function some_work(item) {
    item.load_modules([item, item.owner], do_some_work);
}

function do_some_work(item) {
    // some code
}
```

See also

[Working with modules](#)

[load_module](#)

[load_script](#)

load_script

load_script (*js_filename, callback, onload*)

domain: client

language: javascript

class [AbstractItem](#)

Description

Use **load_script** method to load javascript file from the server, before executing callback.

The method checks whether the file has been loaded, if not, loads it from the server, executes (if specified) **onload** function and then executes the **callback**, otherwise just the **callback** function is executed. The item is passed to the callback function as a parameter.

The **js_filename** should specify the path to javascript file relative to the server directory.

The request to the sever is executed asynchronously.

Example

Bellow, the `do_some_work` function is executed only when `lib.js` file from server `js` directory has been loaded.

loaded:

```
function some_work(item) {
    item.load_script('js/lib.js', do_some_work);
}

function do_some_work(item) {
    // some code
}
```

See also

Working with modules

load_module

load_modules

message

message (*mess, options*)

domain: client

language: javascript

class *AbstractItem*

Description

Use **message** method to create a modal form.

The **mess** parameter specifies the text or html content that will appear in the body of the form.

The **options** parameter is an object with the following attributes:

- **title** - the title of the form,
- **width** - the width of the form, the default width is 400px
- **height** - the height of the form,
- **margin** - use the margin attribute to define margins of the form body
- **text_center** - if true, the body tags will be centered, the default value is false,
- **buttons** - an object that define buttons that will be created in the footer of the form, keys of the object are button names, values - functions, that will be executed when button clicked,
- **button_min_width** - the min width of the buttons, the default value is 100px,
- **center_buttons** - if true, the buttons will be centered, the default value is false,
- **close_button** - if this value is true, an application will create a close button in the upper-right corner of the form, the default value is true,
- **close_on_escape** - if true, the form will be closed, when user press Escape, the default value is true,
- **print** - if this value is true, an application will create a print button in the upper-right corner of the form to print the body of the form, the default value is false

The method returns a jquery object of the form. To programmatically close the form pass this object to *hide_message* method.

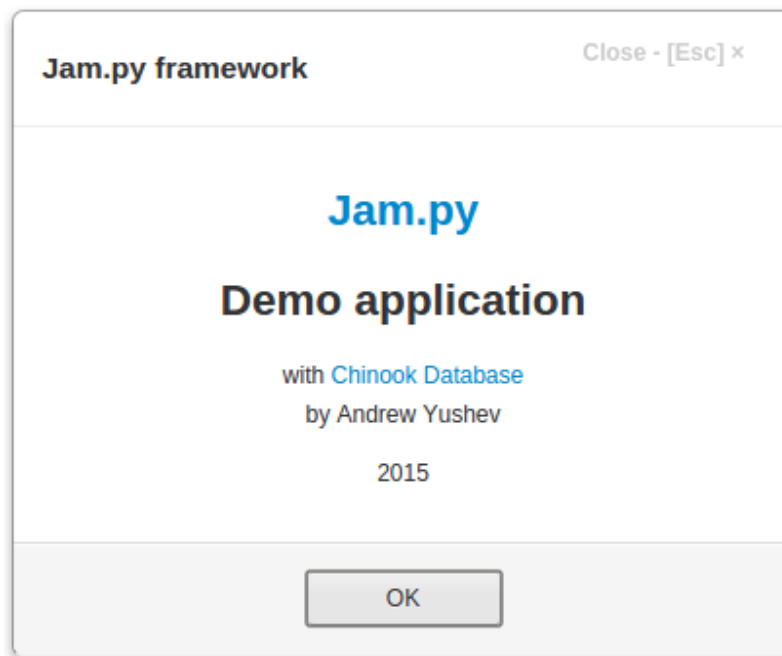
Examples

The following code will create a yes-no-cancel dialog:

```
function yes_no_cancel(item, mess, yesCallback, noCallback, cancelCallback) {  
    var buttons = {  
        Yes: yesCallback,  
        No: noCallback,  
        Cancel: cancelCallback  
    };  
    item.message(mess, {buttons: buttons, margin: "20px",  
        text_center: true, width: 500, center_buttons: true});  
}
```

```
task.message(  
    '<a href="http://jam-py.com/" target="_blank"><h3>Jam.py</h3></a>' +  
    '<h3>Demo application</h3>' +  
    ' with <a href="http://chinookdatabase.codeplex.com/" target="_blank">Chinook_  
↪Database</a>' +  
    '<p>by Andrew Yushev</p>' +  
    '<p>2015</p>'  
    {title: 'Jam.py framework', margin: 0, text_center: true, buttons: {"OK":_  
↪undefined},  
        center_buttons: true}  
);
```

the result of the code above will be:



question

Creates a modal form with **OK**, **Cancel** buttons

question (*mess*, *yes_callback*, *no_callback*, *options*)

domain: client

language: javascript

class *AbstractItem*

Description

Use **question** to create a modal form with **Ok** and **Cancel** buttons.

The **mess** parameter specifies the text or html content that will appear in the body of the form.

If **yes_callback**, **no_callback** functions are specified they will be executed when user clicks on the **Ok** or **Cancel** button, respectively, and then the form will be closed.

Example

The following code creates a modal form, and delete selected record record when the user clicks the OK button:

```

item.question('Delete record?',
    function() {
        item.delete();
    }
);

```

server

server (*func_name*, *params*, *callback*)

domain: client

language: javascript

class *AbstractItem*

Description

Use `server` method to execute a function defined in the server module of an item.

`Sever` method executes a function with a name `func_name` defined in the server module of an item with parameters specified in `params`.

If `callback` is specified, the function on the server is executed asynchronously, after which the `callback` is executed with parameter that is the result of the server function execution, otherwise the function is executed synchronously and returns the result of the server function.

If exception was raised during the operation on the server and the `callback` parameter is not passed (synchronous execution), the client throws an exception. If the `callback` parameter is present, it is passed to the `callback` as parameter.

When exception is raised during the server function execution, the application on the client throws exception with the server exception text.

The first parameter of the function on the server must be `item`, it must be followed by the parameters specified in the function on the client.

`params` is a list of parameters. If there are not parameters, the `params` can be omitted.

Example

The function defined in the **Invoices** journal server module:

```
def get_total(item, id_value):
    result = 0;
    copy = item.copy()
    copy.set_where(id=id_value)
    copy.open()
    if copy.record_count():
        result = copy.total.value
    else:
        raise Exception, 'Journal "invoices" does not have a record with id %s' % id_
↪value
    return result;
```

the following code in the **Invoices** journal client module will execute this server function:

```
task.invoices.server('get_total', [17], function(total, err) {
    if (err) {
        throw err;
    }
    else {
        console.log(total);
    }
});
```

warning

warning (*mess, callback*)

domain: client

language: javascript

class *AbstractItem*

Description

Use **warning** to create a modal form with the **Ok** button.

The **mess** parameter specifies the text or html content that will appear in the body of the form.

If **callback** function are specified it will be executed when user clicks the button and then the form will be closed.

Example

```
item.warning('No record selected.');
```

yes_no_cancel

yes_no_cancel (*mess, yes_callback, no_callback, cancel_callback*)

domain: client

language: javascript

class *AbstractItem*

Description

Use **yes_no_cancel** to create a modal form with **Yes No, Cancel** buttons.

The **mess** parameter specifies the text or html content that will appear in the body of the form.

If **yes_callback**, **no_callback**, **cancel_callback** functions are specified they will be executed when user clicks on the **Yes**, **No** or **Cancel** button, respectively, and then the form will be closed.

Example

The following code is executed when user clicks on the close button in the upper right corner of an item edit form.

```
function on_edit_form_close_query(item) {
    var result = true;
    if (item.is_changing()) {
        if (item.is_modified()) {
            item.yes_no_cancel('Data has been modified. Save changes?',
                function() {
                    item.apply_record();
                },
                function() {
                    item.cancel_edit();
                }
            );
            result = false;
        }
        else {
            item.cancel();
        }
    }
    return result;
}
```

7.1.2 Task class

class **Task**()

domain: client

language: javascript

Task class is used to create the root of the *Task tree* of the project.

Below the attributes, methods and events of the class are listed.

It, as well, inherits attributes and methods of its ancestor class *AbstractItem class*

Attrubutes

forms_container

forms_container

domain: client

language: javascript

class *Task*

Description

The `forms_container` is a JQuery object in which the application will create forms.

To initialize `forms_container` use the *set_forms_container* method or the *create_menu* method.

The default code uses the *create_menu* method.

See also

forms_in_tabs

create_menu

set_forms_container

forms_in_tabs

forms_in_tabs

domain: client

language: javascript

class *Task*

Description

If the `forms_in_tabs` attribute is set and *forms_container* is specified the application will create forms in tabs.

This attribute can be set in the **Interface** tab of *Parameters*.

safe_mode

safe_mode

domain: client

language: javascript

class *Task*

Description

Check the `safe_mode` attribute to determine if the *safe mode* parameter of the project is set.

Example

```
function on_page_loaded(task) {

    $("#title").html(task.item_caption);
    if (task.safe_mode) {
        $("#user-info").text(task.user_info.role_name + ' ' + task.user_info.user_
↪name);
        $('#log-out')
            .show()
            .click(function(e) {
                e.preventDefault();
                task.logout();
            });
    }

    task.tasks.view($("#content"));
}
```

See also

Parameters

user_info

on_page_loaded

templates

templates

domain: client

language: javascript

class *Task*

Description

The `templates` attribute stores the form templates of the project.

See also

Form templates

Forms

user_info

user_info

domain: client

language: javascript

class *Task*

Description

Use `user_info` attribute to get user information when project *Safe mode parameter* is set.

`user_info` is an object that has the following attributes:

- `user_id` - the user id
- `user_name` - the user name
- `role_id` - user role id
- `role_name` - the role assigned to the user
- `admin` - if true the user can work in the Application builder

If safe mode is false the `user_info` attribute is an empty object.

Example

```
function on_page_loaded(task) {
    $("#title").html('Jam.py demo application');
    if (task.safe_mode) {
        $("#user-info").text(task.user_info.role_name + ' ' + task.user_info.user_
↪name);
        $('#log-out')
            .show()
            .click(function(e) {
                e.preventDefault();
                task.logout();
            });
    }
    // some initialization code
}
```

See also

load

login

logout

Users

Roles

Methods

add_tab

add_tab (*container*, *tab_name*, *options*)

domain: client

language: javascript

class *Task*

Description

The `add_tab` method creates a tab for a container.

The `container` is JQuery object for a container element.

The `tab_name` is the name of the tab.

Use can use the `options` to specify optional parameters. It is the object that can have the following attributes:

- `tab_id` - a unique string identifying the tab
- `show_close_btn` - if it is set to `true` the close tab button will appear that can be used to close the tab
- `set_active` - if it is set to `true` the new tab will became active
- `on_close` - a callback function that will be called when the close tab button is clicked

The function returns the JQuery object of the div with `tab-pane` class that will be displayed when tab became active.

Example

The following code will create tabs for editing Customers catalog. It uses *create_inputs* method:

```
function on_edit_form_created(item) {
    var container = item.edit_form.find('.tabs');
    task.init_tabs(container);
    item.create_inputs(task.add_tab(container, 'Customer'),
        {fields: ['firstname', 'lastname', 'company', 'support_rep_id']}
    );
    item.create_inputs(task.add_tab(container, 'Address'),
        {fields: ['country', 'state', 'address', 'postalcode']}
    );
    item.create_inputs(task.add_tab(container, 'Contact'),
        {fields: ['phone', 'fax', 'email']}
    );
}
```

Below is the edit html template for Customers catalog:

```
<div class="customers-edit">
  <div class="form-body">
    <div class="tabs">
    </div>
  </div>
  <div class="form-footer">
    <button type="button" id="ok-btn" class="btn btn-ary expanded-btn">
      <i class="icon-ok"></i> OK<small class="muted">&nbsp; [Ctrl+Enter]</small>
    </button>
    <button type="button" id="cancel-btn" class="btn expanded-btn">
      <i class="icon-remove"></i> Cancel
    </button>
  </div>
</div>
```

See also

init_tabs

close_tab

close_tab

close_tab (*container*, *tab_id*)

domain: client

language: javascript

class *Task*

Description

Use the `close_tab` method to close tab in the container identified by `tab_id`.

See also

init_tabs

add_tab

create_menu

create_menu: **function** (*menu*, *forms_container*, *options*)

domain: client

language: javascript

class *Task*

Description

The `create_menu` method created a menu based on the project *task tree*.

If display forms in tabs attribute of the *project parameters* is set, initializes tabs that will be created to display forms.

It iterates through the items of the *task tree* and adds items to the menu for which the visible attribute is set to true, and the user has the right to view them.

The method uses to assign on click event to the menu items so that for reports the *print* method will be executed when a user clicks it and the *view* method will be executed for other items.

The following parameters could be passed to the method:

- `menu` - a JQuery object of the menu element from index.html file
- `forms_container` a JQuery object of the element that will contain the forms created by the *view* method
- `options` - an object that can have the following attributes:
 - `custom_menu` - use this option to create a custom menu, see below for details
 - `view_first` - if it is true the view form of the first item in the menu will be displayed after menu is created, the default value is false
 - `create_single_group` - if it is true and only one group in the task tree has items the menu item for the group will be created that have a drop down menu for group items, otherwise the menuitems for each item will be created, the default value is false
 - `splash_screen` - an html that will be displayed in the forms_container when all tabs are closed

Custom menu option

To create your own custom menu you must set a `custom_menu` option.

This option is a list of menu objects, each object can be:

- Jam.py item or item group
- array: the first element of the array is the name of the menu item, and the second is the list of menu objects
- object with one attribute: the key of the attribute is the name of menu item and the value - a list of menu objects
- object with one attribute: the key of the attribute is the name of menu item and the value - function to be executed when the menu item is clicked

To add a separator, an empty string (``) can be added to the list of menu objects

Example

```
task.create_menu($("#menu"), $("#content"), {
  splash_screen: '<h1 class="text-center">Jam.py Demo Application</h1>',
  view_first: true
});
```

An example with custom menu:

```
let menu = [
  ['First', [task.invoices, task.customers]],
  ['Second': [task.catalogs, '', task.reports]],
  {Third: [task.tracks, {Params: function() {alert('params clicked')}}]},
  {Fourth: [task.task.analytics, {'Artists list': [task.artists]}]},
  task.reports,
  {Params: function() {alert('params clicked')}}},
];
task.create_menu($("#menu"), $("#content"), {
  custom_menu: menu,
  splash_screen: '<h1 class="text-center">Jam.py Demo Application</h1>',
  view_first: true
});
```

init_tabs

init_tabs (*container*, *tabs_position*)

domain: client

language: javascript

class *Task*

Description

The `init_tabs` method initializes tabs for a container.

The `container` is JQuery object for a container element.

The `tabs_position` parameter specifies where tabs, created by the [add_tab](#) method will be positioned. It is string that can be one of the following values:

- tabs-below
- tabs-left
- tabs-right

If this parameter is omitted tabs will be positioned at the top of the container.

After this method is called you can use the [add_tab](#) method to create tabs.

See also

[add_tab](#)

[close_tab](#)

load

load (*callback*)

domain: client

language: javascript

class *Task*

Description

Load method loads the project *task tree* from the server and initializes it.

When a Web browser loads the jam.js library in index.html file, jam.js creates an empty task object. The `load` method loads the project *task tree* from the server and initializes it (see *workflow*). After that the application triggers *on_page_loaded* event.

Example

The following code is from the project index.html file.

```
<script src="/jam/js/jam.js"></script>
<script src="/js/events.js"></script>

<script>
$(document).ready(function() {
    task.load();
});
</script>
```

See also

login

logout

user_info

Users

Roles

login

login (*callback*)

domain: client

language: javascript

class *Task*

Description

The `login` method creates a login form using the login form div defined in the templates of the index.html file. It is called by the *load* method when the project *Safe mode parameter*

See also

load

logout

user_info

Users

Roles

logout

logout ()

domain: client

language: javascript

class *Task*

Description

Call `logout` to logout a user.

Example

```
function on_page_loaded(task) {
    $("#title").html('Jam.py demo application');
    if (task.safe_mode) {
        $("#user-info").text(task.user_info.role_name + ' ' + task.user_info.user_
↪name);
        $('#log-out')
            .show()
            .click(function(e) {
                e.preventDefault();
                task.logout();
            });
    }
    // some initialization code
}
```

See also

load

login

user_info

Users

Roles

set_forms_container

set_forms_container (*container*, *options*)

domain: client

language: javascript

class *Task*

Description

The `set_forms_container` can be used to initialize the *forms_container* attribute that will contain forms of the application.

If the *forms_in_tabs* attribute is set the applications also initializes the tabs that will be used to display forms.

The container is JQuery object that will be used as a container for the application forms.

The options parameter can have the following attribute:

- `splash_screen` - an html that will be displayed in the forms_container when all tabs are closed

Example

```
task.set_forms_container($("#content"), {
    splash_screen: '<h1 class="text-center">Jam.py Demo Application</h1>'
});
```

See also

forms_container

forms_in_tabs

create_menu

upload

upload (*options*)

domain: client

language: javascript

class *Task*

Description

Use the `upload` method to select a file in the File open dialog box and upload it to the *static/files* directory in the server folder.

When saving the file on the server, the file name is changed by the Werkzeug `secure_filename` function and then the current date is added to it. See <http://werkzeug.pocoo.org/docs/0.14/utils/>

The `options` parameter is an object that may have the following attributes:

- `callback` - is a callback function that is executed when the file is downloaded. It is passed, as parameters, the name of the file stored on the server, the name of the downloaded file and the path to the folder where the file was saved.
- `show_progress` - if true and the uploaded file is large, the progress bar will be displayed. the default value is true
- `accept` - the attribute specifies the types of files that can be submitted through a file upload, see [Accept string](#)

Note: Please note that the `accept` attribute specifies only types of files that can be picked up by the user in the browser.

The server checks all uploaded files for compliance with the **Upload file extensions** attribute of the [Project parameters](#).

Events

`on_edit_form_close_query`

`on_edit_form_created(item)`

domain: client

language: javascript

class *Task class*

Description

The `on_edit_form_close_query` event is triggered by the *close_edit_form* method of the item.

The `item` parameter is the item that triggered the event.

See also

Forms

close_edit_form

`on_edit_form_created`

`on_edit_form_created(item)`

domain: client

language: javascript

class *Task class*

Description

The `on_edit_form_created` event is triggered by the `create_edit_form` method of the item when the form has been created but not shown yet.

The `item` parameter is the item that triggered the event.

This event, if defined, is triggered for every item of the task, whose `create_edit_form` method has been called.

See also

Forms

`create_edit_form`

on_edit_form_keydown

`on_edit_form_keydown(item, event)`

domain: client

language: javascript

class *Task class*

Description

The `on_edit_form_keydown` event is triggered when the keydown event occurs for the `edit_form` of the item.

The `item` parameter is the item that triggered the event.

The `event` is JQuery event object.

See also

Forms

`create_edit_form`

on_edit_form_keyup

`on_edit_form_keyup(item, event)`

domain: client

language: javascript

class *Task class*

Description

The `on_edit_form_keyup` event is triggered when the `keyup` event occurs for the *edit_form* of the item.

The `item` parameter is the item that triggered the event.

The `event` is JQuery event object.

See also

Forms

create_edit_form

on_edit_form_shown

`on_edit_form_shown(item)`

domain: client

language: javascript

class *Task class*

Description

The `on_edit_form_shown` event is triggered by the *create_edit_form* method of the item when the form has been shown.

The `item` parameter is the item that triggered the event.

This event, if defined, is triggered for every item of the task, whose *create_edit_form* method has been called.

See also

Forms

create_edit_form

on_filter_form_close_query

`on_filter_form_close_query(item)`

domain: client

language: javascript

class *Task class*

Description

The `on_filter_form_close_query` event is triggered by the *close_filter_form* method of the item.

The `item` parameter is the item that triggered the event.

See also

Forms

create_filter_form

close_filter_form

on_filter_form_created

on_filter_form_created(item)

domain: client

language: javascript

class *Task class*

Description

The `on_filter_form_created` event is triggered by the *create_filter_form* method of the item when the form has been created but not shown yet.

The `item` parameter is the item that triggered the event.

This event, if defined, is triggered for every item of the task, whose *create_filter_form* method has been called.

See also

Forms

create_filter_form

on_filter_form_shown

on_filter_form_shown(item)

domain: client

language: javascript

class *Task class*

Description

The `on_filter_form_shown` event is triggered by the *create_filter_form* method of the item when the form has been shown.

The `item` parameter is the item that triggered the event.

This event, if defined, is triggered for every item of the task, whose *create_filter_form* method has been called.

See also

Forms

create_filter_form

on_page_loaded

on_page_loaded(task)

domain: client

language: javascript

class *Task class*

Description

The `on_page_loaded` event is the first event triggered on the client. See *Workflow*.

Use it to initialize the client.

The `task` parameter is the root of the client *task tree*.

See also

Workflow

Task tree

on_param_form_close_query

on_param_form_close_query(item)

domain: client

language: javascript

class *Task class*

Description

The `on_param_form_close_query` event is triggered by the *close_param_form* method.

The `report` parameter is the report that triggered the event.

See also

Forms

Client-side report programming

close_param_form

on_param_form_created

on_param_form_created(item)

domain: client

language: javascript

class *Task class*

Description

The on_param_form_created event is triggered by the *create_param_form* method, that, usually, is called by then *print* method.

The report parameter is the report that triggered the event.

See also

Forms

Client-side report programming

print

create_param_form

on_param_form_shown

on_param_form_shown(item)

domain: client

language: javascript

class *Task class*

Description

The on_param_form_shown event is triggered by the *create_param_form* method, that, usually, is called by then *print* method.

The report parameter is the report that triggered the event.

See also

Forms

Client-side report programming

print

create_param_form

on_view_form_close_query

on_view_form_close_query(item)

domain: client

language: javascript

class *Task class*

Description

The on_view_form_close_query event is triggered by the *close_view_form* method of the item.

The `item` parameter is the item that triggered the event.

See also

Forms

close_view_form

on_view_form_created

on_view_form_created(item)

domain: client

language: javascript

class *Task class*

Description

The on_view_form_created event is triggered by the *view* method of the item when the form has been created but not shown yet.

The `item` parameter is the item that triggered the event.

This event, if defined, is triggered for every item of the task, whose *view* method has been called.

See also

Forms

view

on_view_form_keydown

on_view_form_keydown(item, event)

domain: client

language: javascript

class *Task class*

Description

The `on_view_form_keydown` event is triggered when the keydown event occurs for the *view_form* of the item.

The `item` parameter is the item that triggered the event.

The `event` is JQuery event object.

See also

Forms

view

`on_view_form_keyup`

`on_view_form_keyup(item, event)`

domain: client

language: javascript

class *Task class*

Description

The `on_view_form_keyup` event is triggered when the keyup event occurs for the *view_form* of the item.

The `item` parameter is the item that triggered the event.

The `event` is JQuery event object.

See also

Forms

view

`on_view_form_shown`

`on_view_form_shown(item)`

domain: client

language: javascript

class *Task class*

Description

The `on_view_form_shown` event is triggered by the *view* method of the item when the form has been shown.

The `item` parameter is the item that triggered the event.

This event, if defined, is triggered for every item of the task, whose *view* method has been called.

See also

Forms

view

7.1.3 Group class

class `Group()`

domain: client

language: javascript

Group class is used to create group objects of the *task tree*

Below the events of the class are listed.

It, as well, inherits attributes and methods of its ancestor class *AbstractItem class*

Events

`on_edit_form_close_query`

`on_edit_form_close_query(item)`

domain: client

language: javascript

class *Group class*

Description

The `on_edit_form_close_query` event is triggered by the *close_edit_form* method of the item.

The `item` parameter is the item that triggered the event.

See also

Forms

close_edit_form

on_edit_form_created

on_edit_form_created(item)

domain: client

language: javascript

class *Group class*

Description

The `on_edit_form_created` event is triggered by the *create_edit_form* method of the item when the form has been created but not shown yet.

The `item` parameter is the item that triggered the event.

This event, if defined, is triggered for every item of the group, whose *create_edit_form* method has been called.

See also

Forms

create_edit_form

on_edit_form_keydown

on_edit_form_keydown(item, event)

domain: client

language: javascript

class *Group class*

Description

The `on_edit_form_keydown` event is triggered when the keydown event occurs for the *edit_form* of the item.

The `item` parameter is the item that triggered the event.

The `event` is JQuery event object.

See also

Forms

create_edit_form

on_edit_form_keyup

on_edit_form_keyup(item, event)

domain: client

language: javascript

class *Group class*

Description

The on_edit_form_keyup event is triggered when the keyup event occurs for the *edit_form* of the item.

The `item` parameter is the item that triggered the event.

The `event` is JQuery event object.

See also

Forms

create_edit_form

on_edit_form_shown

on_edit_form_shown(item)

domain: client

language: javascript

class *Group class*

Description

The on_edit_form_shown event is triggered by the *create_edit_form* method of the item when the form has been shown.

The `item` parameter is the item that triggered the event.

This event, if defined, is triggered for every item of the group, whose *create_edit_form* method has been called.

See also

Forms

create_edit_form

on_filter_form_close_query

on_filter_form_close_query(item)

domain: client

language: javascript

class *Group class*

Description

The on_filter_form_close_query event is triggered by the *close_filter_form* method of the item.

The *item* parameter is the item that triggered the event.

See also

Forms

close_filter_form

on_filter_form_created

on_filter_form_created(item)

domain: client

language: javascript

class *Group class*

Description

The on_filter_form_created event is triggered by the *create_filter_form* method of the item when the form has been created but not shown yet.

The *item* parameter is the item that triggered the event.

This event, if defined, is triggered for every item of the group, whose *create_filter_form* method has been called.

See also

Forms

create_filter_form

on_filter_form_shown

on_filter_form_shown(item)

domain: client

language: javascript

class *Group class*

Description

The `on_filter_form_shown` event is triggered by the *create_filter_form* method of the item when the form has been shown.

The `item` parameter is the item that triggered the event.

This event, if defined, is triggered for every item of the group, whose *create_filter_form* method has been called.

See also

Forms

create_filter_form

`on_view_form_close_query`

`on_view_form_close_query(item)`

domain: client

language: javascript

class *Group class*

Description

The `on_view_form_close_query` event is triggered by the *close_view_form* method of the item.

The `item` parameter is the item that triggered the event.

See also

Forms

close_view_form

`on_view_form_created`

`on_view_form_created(item)`

domain: client

language: javascript

class *Group class*

Description

The `on_view_form_created` event is triggered by the `view` method of the item when the form has been created but not shown yet.

The `item` parameter is the item that triggered the event.

This event, if defined, is triggered for every item of the group, whose `view` method has been called.

See also

Forms

view

`on_view_form_keydown`

`on_view_form_keydown(item, event)`

domain: client

language: javascript

class *Group class*

Description

The `on_view_form_keydown` event is triggered when the keydown event occurs for the `view_form` of the item.

The `item` parameter is the item that triggered the event.

The `event` is JQuery event object.

See also

Forms

view

`on_view_form_keyup`

`on_view_form_keyup(item, event)`

domain: client

language: javascript

class *Group class*

Description

The `on_view_form_keyup` event is triggered when the `keyup` event occurs for the *view_form* of the item.

The `item` parameter is the item that triggered the event.

The event is JQuery event object.

See also

Forms

view

on_view_form_shown

`on_view_form_shown(item)`

domain: client

language: javascript

class *Group class*

Description

The `on_view_form_shown` event is triggered by the *view* method of the item when the form has been shown.

The `item` parameter is the item that triggered the event.

This event, if defined, is triggered for every item of the group, whose *view* method has been called.

See also

Forms

view

7.1.4 Item class

class `Item()`

domain: client

language: javascript

Item class is used to create item objects of the *task tree* that may have an associated database table.

Below the attributes, methods and events of the class are listed.

It, as well, inherits attributes and methods of its ancestor class *AbstractItem class*

Attributes and properties

active

active

domain: client

language: javascript

class *Item class*

Description

Specifies whether or not an item dataset is open.

Use `active` read only property to determine whether an item dataset is open.

The *open* method changes the value of `active` to `true`. The *close* method sets it to `false`.

When the dataset is open its records can be navigated and its data can be modified and the changes saved in the item database table.

See also

Dataset

Navigating datasets

Modifying datasets

can_modify

active

domain: client

language: javascript

class *Item class*

Description

Set the `can_modify` property to `false` if you need to prohibit changing of the item in the visual controls.

When `can_modify` is `true` the *can_create*, *can_edit*, *can_delete* methods return `false`.

By default the `can_modify` property is `true`.

details

details

domain: client

language: javascript

class *Item class*

Description

Lists all *detail* objects of the item.

See also

Details

each_detail

edit_form

edit_form

domain: client

language: javascript

class *Item class*

Description

Use `edit_form` attribute to get access to a JQuery object representing the edit form of the item.

It is created by the *create_edit_form* method.

The *close_edit_form* method sets the `edit_form` value to undefined.

Example

In the following example the button defined in the item edit html template is assigned a click event:

```
item.edit_form.find("#ok-btn").on('click.task',
    function() {
        item.apply_record();
    }
);
```

See also

Forms

create_edit_form

close_edit_form

edit_options

edit_options

domain: client

language: javascript

class *Item class*

Description

The `edit_options` attribute is a set of options that determine how the edit form will be displayed on the browser page.

These options are set in the *Edit Form Dialog* in Application Builder.

You can change edit options in the *on_edit_form_created* event handler of the item. See example.

`edit_options` is an object that has the following attributes:

Option	Description
<code>width</code>	the width of the modal form, the default value is 600 px,
<code>title</code>	the title of the form, the default value is the value of a <i>item_caption</i> attribute,
<code>form_border</code>	if true, the border will be displayed around the form
<code>form_header</code>	if true, the form header will be created and displayed containing form title and various buttons
<code>history_button</code>	if true and <i>saving change history is enabled</i> , the history button will be displayed in the form header
<code>close_button</code>	if true, the close button will be created in the upper-right corner of the form
<code>close_on_escape</code>	if true, pressing on the Escape key will execute the <i>close_edit_form</i> method to close the form
<code>edit_details</code>	the list of the detail names, that will be available for editing in the edit form, if edit form template contains the div with class 'edit-detail' (the default edit form template have this div)
<code>detail_height</code>	the height of the detail displayed in the view form, if not specified the height of the detail table is 200px
<code>fields</code>	specify the list of field names that the <i>create_inputs</i> method will use, if fields attribute of its options parameter is not specified
<code>template_class</code>	if specified, the div with this class will be searched in the task <i>templates</i> attribute and used as a form html template when creating a form. This attribute must be set before creating the form
<code>modeless</code>	if set the edit forms will be created modeless, otherwise - modal

Example

```
function on_edit_form_created(item) {
    item.edit_options.width = 800;
    item.edit_options.close_on_escape = false;
}
```

See also

Forms

create_edit_form

close_edit_form

fields

fields

domain: client

language: javascript

class *Item class*

Description

Lists all *field* objects of the item.

Example

```
function customer_fields(customers) {
  customers.open({limit: 1});
  for (var i = 0; i < customers.fields.length; i++) {
    console.log(customers.fields[i].field_caption, customers.fields[i].display_
↵text);
  }
}
```

See also

Fields

Field class

each_field

filter_form

filter_form

domain: client

language: javascript

class *Item class*

Description

Use `filter_form` attribute to get access to a JQuery object representing the filter form of the item.

It is created by the *create_filter_form* method.

The *close_filter_form* method sets the `filter_form` value to undefined.

Example

In the following example the button defined in the item filter html template is assigned a click event:

```
item.filter_form.find("#cancel-btn").on('click',
    function() {
        item.close_filter()
    }
);
```

See also

Forms

create_filter_form

close_filter_form

filter_options

filter_options

domain: client

language: javascript

class *Item class*

Description

Use the `filter_options` attribute to specify parameters of the modal filter form.

`filter_options` is an object that has the following attributes:

- `width` - the width of the modal form, the default value is 560 px,
- `title` - use it to get or set the title of the filter form,
- `close_button` - if true, the close button will be created in the upper-right corner of the form, the default value is true,
- `close_caption` - if true and `close_button` is true, will display 'Close - [Esc]' near the button
- `close_on_escape` - if true, pressing on the Escape key will trigger the *close_filter_form* method.
- `close_focusout` - if true, the *close_filter_form* method will be called when a form loses focus
- `template_class` - if specified, the div with this class will be searched in the task *templates* attribute and used as a form html template when creating a form

Example

```
function on_filter_form_created(item) {
    item.filter_options.width = 700;
}
```

See also

Forms

create_filter_form

close_filter_form

Filtered

filtered

domain: client

language: javascript

class *Item class*

Description

Specifies whether or not filtering is active for a dataset.

Check `filtered` to determine whether or not local dataset filtering is in effect. If `filtered` is `true`, then filtering is active. To apply filter conditions specified in the *on_filter_record* event handler, set `filtered` to `true`.

See also

on_filter_record

filters

filters

domain: client

language: javascript

class *Item class*

Description

Lists all *filter* objects of the item.

Example

```
function invoices_filters(invoices) {  
    for (var i = 0; i < invoices.filters.length; i++) {  
        console.log(invoices.filters[i].filter_caption, invoices.filters[i].value);  
    }  
}
```


See also

Filters

Filter class

each_filter

item_state

item_state

domain: client

language: javascript

class *Item*

Description

Examine `item_state` to determine the current operating mode of the item. `Item_state` determines what can be done with data in an item dataset, such as editing existing records or inserting new ones. The `item_state` constantly changes as an application processes data.

Opening a item changes state from inactive to browse. An application can call *edit* to put an item into edit state, or call *insert* or *append* to put an item into insert state.

Posting or canceling edits, insertions, or deletions, changes `item_state` from its current state to browse. Closing a dataset changes its state to inactive.

To check `item_state` value use the following methods:

- *is_new* - indicates whether the item is in insert state
- *is_edited* - indicates whether the item is in edit state
- *is_changing* - indicates whether the item is in edit or insert state

`item_state` value can be:

- 0 - inactive state,
- 1 - browse state,
- 2 - insert state,
- 3 - edit state,
- 4 - delete state

item *task* attribute have consts object that defines following attributes:

- "STATE_INACTIVE": 0,
- "STATE_BROWSE": 1,
- "STATE_INSERT": 2,
- "STATE_EDIT": 3,
- "STATE_DELETE": 4

so if the item is in edit state can be checked the following way:

```
item.item_state == 2
```

or:

```
item.item_state == item.task.consts.STATE_INSERT
```

or:

```
item.is_new()
```

See also

Modifying datasets

log_changes

log_changes

domain: client

language: javascript

class *Item class*

Description

Indicates whether to log data changes.

Use `log_changes` to control whether or not changes made to the data in an item dataset are recorded. When `log_changes` is `true` (the default), all changes are recorded. They can later be applied to an application server by calling the [apply](#) method. When `log_changes` is `false`, data changes are not recorded and cannot be applied to an application server.

See also

Modifying datasets

apply

lookup_field

lookup_field

domain: client

language: javascript

class *Item class*

Description

Use `lookup_field` to check if the item was created to select a value for the lookup field. See [Lookup fields](#)

Example

```
function on_view_form_created(item) {
  item.table_options.multiselect = false;
  if (!item.lookup_field) {
    var print_btn = item.add_view_button('Print', {image: 'icon-print'}),
        email_btn = item.add_view_button('Send email', {image: 'icon-pencil'});
    email_btn.click(function() { send_email() });
    print_btn.click(function() { print(item) });
    item.table_options.multiselect = true;
  }
}
```

paginate

paginate

domain: client

language: javascript

class *Item class*

Description

The `paginate` attribute determines the behavior of a table created by the *create_table* method

When `paginate` is set to `true`, a paginator is created, and the table calculates the number of the rows displayed, based on its height. The table will internally manipulate the `limit` and `offset` parameters of the *open* method, depending on its height and current page, reopening the dataset when page changes.

If `paginate` value is `false`, the table will displays all available records of the dataset.

See also

create_table

open

permissions

permissions

domain: client

language: javascript

class *Item class*

Description

Set the `permissions` property attributes to prohibit changing of the item in the visual controls.

The `permissions` property is an object that has the following attributes:

- `can_create`
- `can_edit`
- `can_delete`

By default these attributes are set to `true`.

When these attributes are set to `false` the corresponding

- *`can_create`*,
- *`can_edit`*,
- *`can_delete`*

methods return `false`.

See also

How to prohibit changing record

`read_only`

`read_only`

domain: `client`

language: `javascript`

class *Item class*

Description

Read the `read_only` property to determine whether the data can be modified in data-aware controls.

Set `read_only` property to `true` to prevent data from being modified in data-aware controls.

When you assign a value to the `read_only` property, the application sets the `read_only` property of all the details and the *`read_only`* property of each field to that value.

If the user role prohibits editing of the record, `read_only` always returns `true`.

See also

`read_only`

Example

In this example we first set `read_only` attribute of the invoices item to `true`. It makes all fields and `invoice_table` detail read only. After that we allow a user to edit customer field and `invoice_table` detail.

```
function on_edit_form_created(item) {
    item.read_only = true;
    item.customer.read_only = false;
    item.invoice_table.read_only = false;
}
```

rec_count

rec_count

domain: client

language: javascript

class *Item class*

Description

Read the `rec_count` property to get the number of records owned by the item's dataset.

If the module declares an *on_filter_record* event handler and the *Filtered* attribute is set, this property calculates the number of records that satisfy this filter, otherwise the *record_count* method is used to calculate the number of records.

See also

record_count

Example

```
function edit_invoice(invoice_id) {
    var invoices = task.invoices.copy();
    invoices.open({ where: {id: invoice_id} }, function() {
        if (invoices.rec_count) {
            invoices.edit_record();
        }
        else {
            invoices.alert_error('Invoices: record not found.');
        }
    });
}
```

rec_no

rec_no

domain: client

language: javascript

class *Item class*

Description

Examine the `rec_no` property to determine the record number of the current record in the item dataset. `rec_no` can be set to a specific record number to position the cursor on that record.

Example

```
function calculate(item) {
    var subtotal,
        tax,
        total,
        rec;
    if (!item.calculating) {
        item.calculating = true;
        try {
            subtotal = 0;
            tax = 0;
            total = 0;
            item.invoice_table.disable_controls();
            rec = item.invoice_table.rec_no;
            try {
                item.invoice_table.each(function(d) {
                    subtotal += d.amount.value;
                    tax += d.tax.value;
                    total += d.total.value;
                });
            }
            finally {
                item.invoice_table.rec_no = rec;
                item.invoice_table.enable_controls();
            }
            item.subtotal.value = subtotal;
            item.tax.value = tax;
            item.total.value = total;
        }
        finally {
            item.calculating = false;
        }
    }
}
```

See also

Dataset

Navigating datasets

selections

selections

domain: client

language: javascript

class *Item* class

Description

The `selections` attribute stores a list of a primary key field values.

When a **Multiple selection** check box is checked on the **Layout** tab in the *View Form Dialog* or `multiselect` attribute of the *table_options* is set programmatically, the check box in the leftmost column of the table appears and each time a user clicks on the check box, the `selections` attribute changes.

It can also be changed programmatically by using `add` or `remove` methods or assigning an array.

Example

In this example, the `send_email` function, on the client, uses **Customers** selection attribute to get array of primary key field values selected by users and send them to the `send_email` function defined in the server module of the item using the *server* method

```
function send_email(subject, message) {
    var selected = task.customers.selections;
    if (!selected.length) {
        selected.add(task.customers.id.value);
    }

    item.server('send_email', [selected, subject, message],
        function(result, err) {
            if (err) {
                item.alert('Failed to send the mail: ' + err);
            }
            else {
                item.alert('Successfully sent the mail');
            }
        }
    );
}
```

On the server, this array is used to retrieve information about selected customers using *open* method

```
import smtplib

def send_email(item, selected, subject, mess):
    cust = item.task.customers.copy()
    cust.set_where(id__in=selected)
    cust.open()
    to = []
    for c in cust:
        to.append(c.email.value)

    # code that sends email
```

table_options

table_options

domain: client

language: javascript

class *Item class*

Description

The `table_options` attribute is a set of options that determine how the table of the view form of will be displayed. Options defined in it are used by the *create_table* method if its `options` parameter don't override corresponding option.

These options are set in the **Layout** tab of the *View Form Dialog* in Application Builder.

You can change `table_options` in the *on_view_form_created* event handler of the item. See example.

The `table_options` parameter is an object that may have the following attributes:

Option	Description
row_count	specifies the number of rows displayed by the table
height	if row_count is not specified, it determines the height of the table, the default value is 480. The table at creation calculates the number of rows displayed (row_count), based on the value of this parameter.
fields	a list of field names. If specified, a column will be created for each field whose name is in this list, if not specified (the default) then the fields attribute of an view_options will be used
title_line_count	specifies the number of lines of text displayed in a title row, if it is 0, the height of the row is determined by the contents of the title cells
row_line_count	specifies the number of lines of text displayed in a table row, if it is 0, the height of the row is determined by the contents of the cells
expand_selected_row	if row_line_count is set and expand_selected_row is greater than 0, it specifies the minimal number of lines of text displayed in the selected row of the table
title_word_wrap	specifies if the column title text can be wrapped.
column_width	the width of the columns are calculated by a Web Browser. You can use this option to force the width of columns. The option is an object, key values of which are field names, the values are column widths as CSS units
editable_fields	the list of field names could be edited in the table.
selected_field	if editable_fields are set, specifies the name of the field whose column will be selected, when the selected row is changed.
sortable	if this option is specified, it is possible to sort the records by clicking on the table column header. When a sort_fields option is not specified (default), a user can sort records on any field, otherwise, only on the fields whose names are listed in this option.
sort_fields	the list of field names on which the table can be sorted, by clicking on the corresponding table column header. If an item is a detail the operation is performed on the client, otherwise sorting is performed on the server (the open method is used internally).
summary_fields	a list of field names. When it is specified, the table calculates sums for numeric fields and displays them in the table footer, for not numeric fields it displays the number of records.
freeze_count	integer value. If it is greater than 0, it specifies number of first columns that become frozen - they will not scroll when the table is scrolled horizontally.
show_hints	if true, the tooltip will be displayed when the user hovers the mouse over a table cell, and the cell text does not fit in the cell size. The default value is true.
hint_fields	a list of field names. If it is specified, the tooltip will be displayed only for fields from this list, regardless of the value of show_hints option value.
on_click	specifies the function, that will be executed when a user click on a table row. The item will be passed as a parameter to the function.
on_dblclick	specifies the function, that will be executed when a user double click on a table row. The item will be passed as a parameter to the function.
dblclick_edit	if the value of the option is set to true and the on_dblclick option is not set, the edit form will be shown when a user double click on a table row.
multi-select	if this option is set, a leftmost column with check-boxes will be created to select records. So, that when a user clicks on the check-box, the value of the primary key field of the record will be added to or deleted from the selections attribute.
select_all	if true, the menu will appear in the leftmost column of the table header, which will allow the user selects all records that match the current filters and the search value.
row_callback	back callback functions called each time fields of the record are changed. Two parameters are passed to the function - item, whose record has changed and JQuery object of the corresponding row of the table. Please be careful - the item passed to the function can be not item itself, but its clone that share the same dataset.

Example

```
function on_view_form_created(item) {
    item.table_options.row_line_count = 2;
    item.table_options.expand_selected_row = 3;
}
```

The code in the following two examples does the same:

```
item.invoice_table.create_table(item.view_form.find('.view-detail'), {
    height: 200,
    summary_fields: ['date', 'total'],
});
```

```
item.invoice_table.table_options.height = 200;
item.invoice_table.table_options.summary_fields = ['date', 'total'];
item.invoice_table.create_table(item.view_form.find('.view-detail'));
```

See also

View Form Dialog

on_view_form_created

create_table

view_form

view_form

domain: client

language: javascript

class *Item class*

Description

Use `view_form` attribute to get access to a JQuery object representing the view form of the item.

It is created by the *view* method.

The *close_view_form* method sets the `view_form` value to undefined.

Example

In the following example the button defined in the item html template is assigned a click event:

```
item.view_form.find("#new-btn").on('click',
    function() {
        item.insert_record();
    }
);
```

See also

Forms

view

create_view_form

close_view_form

view_options

view_options

domain: client

language: javascript

class *Item class*

Description

The `view_options` attribute is a set of options that determine how the view form of will be displayed on the browser page.

These options are set in the *View Form Dialog* in Application Builder.

You can change view options in the *on_view_form_created* event handler of the item. See example.

`view_options` is an object that has the following attributes:

Option	Description
<code>width</code>	the width of the modal form, the default value is 600 px
<code>title</code>	the title of the form, the default value is the value of a <i>item_caption</i> attribute,
<code>form_border</code>	if true, the border will be displayed around the form
<code>form_header</code>	if true, the form header will be created and displayed containing form title and various buttons
<code>history_button</code>	if true and <i>saving change history is enabled</i> , the history button will be displayed in the form header
<code>refresh_buttons</code>	if true, the refresh button will be created in the form header, that will allow users to refresh the page by sending request to the server
<code>enable_search</code>	if true, the search input will be created in the form header
<code>search_field</code>	the name of the field that will be the default search field
<code>enable_filters</code>	if true and there are visible filters, the filter button will be created in the form header
<code>close_button</code>	if true, the close button will be created in the upper-right corner of the form
<code>close_on_escape</code>	if true, pressing on the Escape key will execute the <i>close_view_form</i> method to close the form
<code>view_details</code>	the list of detail names, that will be displayed in the view form, if view form template contains the div with class 'view-detail' (the default view form template have this div)
<code>detail_height</code>	the height of the details displayed in the view form, if not specified the height of the detail table is 200px
<code>modeless</code>	if true, the form will be displayed as modeless
<code>template_class</code>	if specified, the div with this class will be searched in the task <i>templates</i> attribute and used as a form html template when creating a form. This attribute must be set before the form is created

Example

```
function on_view_form_created(item) {  
  item.view_options.width = 800;  
  item.view_options.close_button = false;  
  item.view_options.close_on_escape = false;  
}
```

See also

Forms

view

virtual_table

virtual_table

domain: client

language: javascript

class *Item class*

Description

Use the read-only `virtual_table` property to find out if the item has a corresponding table in the project database.

If `virtual_table` is `True` there is no corresponding table in the project database. You can use these items to work with in-memory dataset or use its modules to write code. Calling the *open* method creates an empty data set, and calling the *apply* method does nothing.

Methods

add_edit_button

add_edit_button (*text, options*)

domain: client

language: javascript

Description

Use `add_edit_button` to dynamically add a button in the edit form.

This method have the same parameters as the *add_view_button* method

add_view_button

add_view_button (*text*, *options*)

domain: client

language: javascript

Description

Use `add_view_button` to dynamically add a button in the view form.

This method is usually used in the `on_view_form_created` events.

The following parameters are passed to the method:

- `text` - the text that will be displayed on the button
- `options` - options that specify additional properties of the button

The `options` parameter is an object that may have following attributes:

- `parent_class_name` is a class name of the parent element, the default value is 'form-footer'
- `btn_id` - the id attribute of the button
- `btn_class` - the class of the button
- `type` - specifies the type (color) of the button, it can be one of the following text values:
 - primary
 - success
 - info
 - warning
 - danger
- `image` - an icon class, one of the icons by Glyphicons from <http://getbootstrap.com/2.3.2/base-css.html>
- `secondary`: if this attribute is set to true, the button will be right aligned if **Buttons on top** attribute of the *View Form Dialog* is set, otherwise left aligned.
- `expanded` - if set to true the button will have class 'expanded-btn' and that defines its min-width to 120px, default true

The method returns a JQuery object of the button.

Examples

```
function on_view_form_created(item) {
    var btn = item.add_view_button('Select', {type: 'primary'});
    btn.click(function() {
        item.select_records('track');
    });
}
```

```
function on_view_form_created(item) {
    if (!item.view_form.hasClass('modal')) {
```

(continues on next page)

(continued from previous page)

```
var print_btn = item.add_view_button('Print', {image: 'icon-print'}),
    email_btn = item.add_view_button('Send email', {image: 'icon-pencil'});
email_btn.click(function() { send_email() });
print_btn.click(function() { print(item) });
}
```

append

append()

domain: client

language: javascript

class *Item class*

Description

Open a new, empty record at the end of the dataset.

After a call to `append`, an application can enable users to enter data in the fields of the record, and can then post those changes to the item dataset using `post` method, and then apply them to the item database table, using `apply` method.

The `append` method

- checks if item dataset is *active* , otherwise raises exception
- if the item is a *detail* , checks if the master item is in edit or insert *state* , otherwise raises exception
- if the item is not a *detail* checks if it is in browse *state* , otherwise raises exception
- triggers the *on_before_append* event handler if one is defined for the item
- open a new, empty record at the end of the dataset
- puts the item into insert *state*
- triggers the *on_after_append* event handler if one is defined for the item.
- updates *data-aware controls*

See also

Modifying datasets

append_record

append_record(*container*)

domain: client

language: javascript

class *Item class*

Description

Open a new, empty record at the end of the dataset and creates an *edit_form* for visuall editing of the record.

If *container* parameter (Jquery object of the DOM element) is specified the edit form html template is inserted in the container.

If *container* parameter is not specified but **Modeless form** attribute is set in the *Edit Form Dialog* or modeless attribute of the *edit_options* is set programmatically and task has the *forms_in_tabs* attribute set and the application doesn't have modal forms, the modeless edit form will be created in the new tab of the *forms_container* object of the task.

In all other cases the modal form will be created.

If adding of a record is allowed in modeless mode, the application calls the *copy* method to create a copy of the item. This copy will be used to append the record.

The *append_record* method

- calls the *can_create* method to check whether a user have a right to append a record, and if not, returns
- checks whether the item is in edit or insert *state* , and if not, calls the *append* method to append a record
- calls the *create_edit_form* method to create a form for visuall editing of the record

See also

Modifying datasets

append

can_create

apply

apply (*callback*, *params*, *async*)

domain: client

language: javascript

class *Item class*

Description

Sends all updated, inserted, and deleted records from the item dataset to the application server for writing to the database.

The *apply* method can have the following parameters:

- *callback*: if the parameter is not present and *async* parameter is *false* or *undefined*, the request to the server is sent synchronously, otherwise, the request is executed asynchronously and after the response is received, the callback is executed
- *params* - an object specifying user defined params, that can be used on the server in the *on_apply* event handler for some additional processing
- *async*: if its value is *true*, and *callback* parameter is missing, the request is executed asynchronously

The order of parameters doesn't matter.

The `apply` method

- checks whether the item is a detail, and if it is, returns (the master saves the details changes)
- checks whether the item is in edit or insert *state* , and if so, posts the record
- checks if the change log has changes, and if not, executes callback if it is passed and then returns
- triggers the *on_before_apply* event handler if one is defined for the item
- sends changes to the server
- server on receiving the request checks whether *on_apply* event handler is defined for the item, and if it is, executes it, otherwise generates and executes SQL query to write changes to the database, see also *on_apply events* topic
- when generating an SQL query, checks whether a user, that send the request, has rights to make these changes, if not raises an exception
- writes changes to the database
- after writing changes to the database, server sends to the client results of the execution
- if exception was raised during the operation on the server the client throws an exception, before throwing exception, if the callback parameter is passed, it is called and the error is passed as the callback function parameter
- the client, based on the results, updates the change log
- triggers the *on_after_apply* event handler if one is defined for the item
- if the callback parameter is passed, it is called.

Note: The server, before writing new records to the database table, generates values for the primary fields. The client updates these fields, based on information received from the server. If you change values of some other fields in the *on_apply* event handler, these changes will not be reflected on the client. You can update them yourself using, for example, *refresh_record* method

Example

```
var self = this;
this.apply(function(err) {
    if (err) {
        self.alert_error(err);
    }
    else {
        //some code to execute after applying changes
    }
});
```

See also

Modifying datasets

apply_record

apply_record()

domain: client

language: javascript

class *Item class*

Description

Writes changes to the application dataset.

The `apply_record` method

- calls the *apply* to writes changes to the dataset.
- calls the *close_edit_form* method to destroy the `edit_form`

See also

Modifying datasets

close_edit_form

apply

assign_filters

assign_filters(*item*)

domain: client

language: javascript

class *Item class*

Description

Use `assign_filters` to set filter values of the `item` to values of filters of the `item` parameter.

Example

```
function calc_footer(item) {
  var copy = item.copy({handlers: false, details: false});
  copy.assign_filters(item);
  copy.open(
    {fields: ['subtotal', 'tax', 'total'],
     funcs: {subtotal: 'sum', tax: 'sum', total: 'sum'}},
    function() {
      var footer = item.view_form.find('.dbtable.' + item.item_name + 'tfoot');
      copy.each_field(function(f) {
        footer.find('div.' + f.field_name)
```

(continues on next page)

(continued from previous page)

```
        .css('text-align', 'right')
        .css('color', 'black')
        .text(f.display_text);
    });
}
);
}
```

See also

Filtering records

Filters

bof

bof ()

domain: client

language: javascript

class *Item class*

Description

Test `bof` (beginning of file) method to determine if the cursor is positioned at the first record in an item dataset.

If `bof` returns true, the cursor is unequivocally on the first row in the dataset. `bof` returns true when an application

- Opens an item dataset.
- Calls an item's *first* method.
- Call an item's *prior* method, and the method fails (because the cursor is already on the first row in the dataset).

`bof` returns false in all other cases.

Note: If both *eof* and `bof` return true, the item dataset is empty.

See also

Dataset

Navigating datasets

calc_summary

calc_summary (*detail, fields*)

domain: client

language: javascript

class *Item* *class*

Description

Use the `calc_summary` method to calculate sums for fields of a detail and save these values in fields of its master in the `on_detail_changed` event handler.

The `detail` parameter is the detail for the fields of which the sums are calculated.

The `fields` parameter is an object that defines the correspondence between the master and detail fields. The keys of this object are the master fields, the values are the corresponding details fields. If the detail field is a numeric field, its sum is calculated, otherwise the resulting value will be the number of records. The value of this object can be a function that returns the result of the calculation for a record of the detail.

Example

```
function on_detail_changed(item, detail) {
    var fields = [
        {"total": "total"},
        {"tax": "tax"},
        {"subtotal": function(d) {return d.quantity.value * d.unitprice.value}}
    ];
    item.calc_summary(detail, fields);
}
```

See also

on_detail_changed *Details*

can_create

can_create ()

domain: client

language: javascript

class *Item*

Description

Use `can_create` method to determine if a user have a right to create a new record.

This method takes into account the user permissions set in the *roles node* in the Application Builder when the project *safe mode parameter* is set as well as the values of the *permissions* attribute and the value of *can_modify* attribute.

Example

```
if (item.can_create()) {
    item.view_form.find("#new-btn").on('click',
        function() {
            item.append_record();
        }
    );
}
else {
    item.view_form.find("#new-btn").prop("disabled", true);
}
```

See also

Parameters

can_delete

can_delete()

domain: client

language: javascript

class *Item*

Description

Use `can_delete` method to determine if a user have a right to delete a record of an item dataset.

This method takes into account the user permissions set in the *roles node* in the Application Builder when the project *safe mode parameter* is set as well as the values of the *permissions* attribute and the value of *can_modify* attribute.

Example

```
if (item.can_delete()) {
    item.view_form.find("#delete-btn").on('click',
        function() {
            item.delete_record();
        }
    );
}
else {
    item.view_form.find("#delete-btn").prop("disabled", true);
}
```

can_edit

can_edit()

domain: client

language: javascript

class *Item*

Description

Use `can_edit` method to determine if a user have a right to edit a record of an item dataset.

This method takes into account the user permissions set in the *roles node* in the Application Builder when the project *safe mode parameter* is set as well as the values of the *permissions* attribute and the value of *can_modify* attribute.

Example

```

if (item.can_edit()) {
    item.view_form.find("#edit-btn").on('click',
        function() {
            item.edit_record();
        }
    );
}
else {
    item.view_form.find("#edit-btn").prop("disabled", true);
}

```

cancel

cancel ()

domain: client

language: javascript

class *Item class*

Description

Call `cancel` to undo modifications made to one or more fields belonging to the current record, as long as those changes are not already posted to the item dataset.

Cancel

- triggers the *on_before_cancel* event handler if one is defined for the item.
- to undo modifications made to the current record and its details if the record has been edited or removes the new record if one was appended or inserted.
- puts the item into browse *state*
- triggers the *on_after_cancel* event handler if one is defined for the item.
- updates *data-aware controls*

See also

Modifying datasets

cancel_edit

cancel_edit ()

domain: client

language: javascript

class *Item class*

Description

Cancel visual editing on the record

The `cancel_edit` method

- calls the *close_edit_form* method to destroy the `edit_form`
- calls the *cancel* method to undo modifications made to the record

See also

Modifying datasets

close_edit_form

cancel

clear_filters

clear_filters ()

domain: client

language: javascript

class *Item class*

Description

Use `clear_filters` to set filter values of the item to null.

See also

Filtering records

Filters

clone

clone (*keep_filtered*)

domain: client

language: javascript

class *Item class*

Description

Use the clone method to create a copy of an item that shares with it its dataset. The clone item has its own cursor, so you can navigate it and the cursor position of the item doesn't change.

Set the `keep_filtered` parameter to true if you want the clone to have the same local filter as the item.

Example

```
function calc_sum(item) {
  var clone = item.clone(),
      result = 0;
  clone.each(function(c) {
    result += c.sum.value;
  })
  return result;
}
```

See also

on_filter_record

close

close ()

domain: client

language: javascript

class *Item class*

Description

Call `close` to close an item dataset. After dataset is closed the *active* property is `false`.

See also

Dataset

open

close_edit_form

close_edit_form()

domain: client

language: javascript

class *Item class*

Description

Use `close_edit_form` method to close the edit form of the item.

The `close_edit_form` method triggers the *on_edit_form_close_query* event handler of the item, if one is defined. If the event handler is defined and

- returns `true` - the form is destroyed, the item's *edit_form* attribute is set to undefined and the methods exits
- return `false` - the operation is aborted and the methods exits,

If it don't return a value (undefined) the method triggers the *on_edit_form_close_query* of the group that owners the item, if one is defined for the group. If this event handler is defined and

- returns `true` - the form is destroyed, the item's *edit_form* attribute is set to undefined and the methods exits
- return `false` - the operation is aborted and the methods exits,

If it don't return a value (undefined) the method triggers the *on_edit_form_close_query* of the task. If this event handler is defined and

- returns `true` - the form is destroyed, the item's *edit_form* attribute is set to undefined and the methods exits
- return `false` - the operation is aborted and the methods exits,

If no event handler is defined or none of these event handlers return `false`, the form is destroyed and the item's *edit_form* attribute is set to undefined.

See also

Forms

create_edit_form

edit_form

close_filter_form

close_filter_form()

domain: client

language: javascript

class *Item class*

Description

Use `close_filter_form` method to close the filter form of the item.

The `close_filter_form` method triggers the `on_filter_form_close_query` event handler of the item, if one is defined. If the event handler is defined and

- returns ```false``` - the form is destroyed, the item's `filter_form` attribute is set to undefined and the methods exits
- return `false` - the operation is aborted and the methods exits,

If it don't return a value (undefined) the method triggers the `on_filter_form_close_query` of the group that owners the item, if one is defined for the group. If this event handler is defined and

- returns `true` - the form is destroyed, the item's `filter_form` attribute is set to undefined and the methods exits
- return `false` - the operation is aborted and the methods exits,

If it don't return a value (undefined) the method triggers the `on_filter_form_close_query` of the task. If this event handler is defined and

- returns `true` - the form is destroyed, the item's `filter_form` attribute is set to undefined and the methods exits
- return `false` - the operation is aborted and the methods exits,

If no event handler is defined or none of these event handlers return `false`, the form is destroyed and the item's `filter_form` attribute is set to undefined.

See also

Forms

create_filter_form

filter_form

close_view_form

close_view_form()

domain: client

language: javascript

class *Item class*

Description

Use `close_view_form` method to close the view form of the item.

The `close_view_form` method triggers the `on_view_form_close_query` event handler of the item, if one is defined. If the event handler is defined and

- returns `true` - the form is destroyed, the item's `view_form` attribute is set to undefined and the methods exits
- return `false` - the operation is aborted and the methods exits,

If it don't return a value (undefined) the method triggers the `on_view_form_close_query` of the group that owners the item, if one is defined for the group. If this event handler is defined and

- returns `true` - the form is destroyed, the item's *view_form* attribute is set to undefined and the methods exits
- return `false` - the operation is aborted and the methods exits,

If it don't return a value (undefined) the method triggers the *on_view_form_close_query* of the task. If this event handler is defined and

- returns `true` - the form is destroyed, the item's *view_form* attribute is set to undefined and the methods exits
- return `false` - the operation is aborted and the methods exits,

If no event handler is defined or none of these event handlers return `false`, the form is destroyed and the item's *view_form* attribute is set to undefined.

See also

Forms

view

view_form

copy

copy (*options*)

domain: client

language: javascript

class *Item class*

Description

Use copy to create a copy of an item. The created copy is not added to the *task tree* and will be destroyed by JavaScript garbage collection process when no longer needed.

All attributes of the copy object are defined as they were at the time of loading of the task tree when application started. See *Workflow*

Options parameter further specifies the created copy. It can have the following attributes:

- *handlers* - if the value of this attribute is `true`, all the settings to the item made in the Form Dialogs in the Application Builder and all the functions and events defined in the client module of the item will also be available in the copy. The default value is `true`.
- *filters* - if the value of this attribute is `true`, the filters will be created for the copy, otherwise there will be no filters. The default value is `true`.
- *details* - if the value of this attribute is `true`, the details will be created for the copy, otherwise there will be no details. The default value is `true`.

Example

The following code is used in the *Demo project* to asynchronously calculate total values of the fields, displayed at the foot of the **Invoice** journal table:

```
function on_filter_applied(item) {
    var copy = item.copy({handlers: false, details: false});
    copy.assign_filters(item);
    copy.open(
        {fields: ['subtotal', 'tax', 'total'],
        funcs: {subtotal: 'sum', tax: 'sum', total: 'sum'}},
        function() {
            var footer = item.view_form.find('.dbtable.' + item.item_name + ' tfoot');
            copy.each_field(function(f) {
                footer.find('div.' + f.field_name)
                    .css('text-align', 'right')
                    .css('color', 'black')
                    .text(f.display_text);
            });
        }
    );
}
```

See also

Task tree

Workflow

create_detail_views

create_detail_views (*container*)

domain: client

language: javascript

Description

Use `create_detail_views` to create view forms of the details of the item. These details can be specified in the **Edit details** attribute of the *Edit Form Dialog* or set in the `edit_details` attribute of the *edit_options*.

This method is usually used in the `on_edit_form_created` event handler.

The following parameters are passed to the method:

- `container` - a JQuery object that will contain view form of the details, if there is no container, the method returns.

If there is more than one detail, the method creates view forms in tabs.

If details are not *active*, the method calls their *open* method.

Example

```
function on_edit_form_created(item) {
    item.edit_form.find("#cancel-btn").on('click.task', function(e) { item.cancel_
↵edit(e) });
    item.edit_form.find("#ok-btn").on('click.task', function() { item.apply_record() }
↵);
}
```

(continues on next page)

(continued from previous page)

```
if (!item.master && item.owner.on_edit_form_created) {
    item.owner.on_edit_form_created(item);
}

if (item.on_edit_form_created) {
    item.on_edit_form_created(item);
}

item.create_inputs(item.edit_form.find(".edit-body"));
item.create_detail_views(item.edit_form.find(".edit-detail"));

return true;
}
```

create_edit_form

create_edit_form(*container*)

domain: client

language: javascript

class *Item class*

Description

Use `create_edit_form` method to create an edit form of the item for visual editing of a record.

The method searches for an item html template in the task *templates* attribute (See *Forms*), creates a clone of the template and assigns it to the item *edit_form* attribute.

If `container` parameter is specified the method empties it and appends the html template to it. Otherwise it creates a modal form and appends the html to it.

Triggers the *on_edit_form_created* of the task.

Triggers the *on_edit_form_created* of the group that owners the item, if one is defined for the group.

Triggers the *on_edit_form_created* of the item, if one is defined.

Assigns the JQuery keyup and keydown events to the `edit_form` so that when an JQuery event of the window occurs, the `on_edit_form_keyup` and `on_edit_form_keydown` events are triggered. They are triggered (if defined) in the same way: first the task event handler, the group event handler and then the event handler of the item itself. After that the JQuery `stopPropagation` method of the event is called.

If the form is modal, shows it. Before showing the form the method applies options specified in the *edit_options* attribute.

Triggers the *on_edit_form_shown* of the task.

Triggers the *on_edit_form_shown* of the group that owners the item, if one is defined for the group.

Triggers the *on_edit_form_shown* of the item, if one is defined.

See also

Forms

edit_form

edit_options

create_edit_form

close_edit_form

create_filter_form

create_filter_form(*container*)

domain: client

language: javascript

class *Item class*

Description

Use `create_filter_form` method to create an filter form of the item for visual editing item filters.

The method searches for an item html template in the task *templates* attribute (See *Forms*), creates a clone of the template and assigns it to the item *filter_form* attribute.

If `container` parameter is specified the method empties it and appends the html template to it. Otherwise it creates a modal form and appends the html to it.

Triggers the *on_filter_form_created* of the task.

Triggers the *on_filter_form_created* of the group that owners the item, if one is defined for the group.

Triggers the *on_filter_form_created* of the item, if one is defined.

Assigns the JQuery keyup and keydown events to the `filter_form` so that when an JQuery event of the window occurs, the `on_filter_form_keyup` and `on_filter_form_keydown` events are triggered. They are triggered (if defined) in the same way: first the task event handler, the group event handler and then the event handler of the item itself. After that the JQuery `stopPropagation` method of the event is called.

If the form is modal, shows it. Before showing the form the method applies options specidied in the *filter_options* attribute.

Triggers the *on_filter_form_shown* of the task.

Triggers the *on_filter_form_shown* of the group that owners the item, if one is defined for the group.

Triggers the *on_filter_form_shown* of the item, if one is defined.

See also

Forms

filter_form

filter_options

close_filter_form

create_filter_inputs

create_filter_inputs (*container, options*)

domain: client

language: javascript

Description

Use `create_filter_inputs` to create data-aware visual controls (inputs, checkboxes) for editing *filters* of an item. This method is usually used in `on_filter_form_created` events triggered by *create_filter_form* method.

The following parameters are passed to the method:

- `container` - a JQuery object that will contain visual controls, if container length is 0 (no container), the method returns.
- `options` - options that specify how controls are displayed

The `options` parameter is an object that may have following attributes:

- `filters` - a list of filter names. If specified, a visual control will be created for each filter whose name is in this list, if not specified (the default) then the fields attribute of an *filter_options* will be used (by default it lists all visible filters specified in the Application builder),
- `col_count` - the number of columns that will be created for visual controls, the default value is 1.
- `label_on_top`: the default value is false. If this value is false, the labels are placed to the left of controls, otherwise the are created above the controls
- `tabindex` - if `tabindex` is specified, it will the `tabindex` of the first visual control, `tabindex` of all subsequent controls will be increased by 1.
- `autocomplete` - the default value is false. If this attribute is set to true, the `autocomplete` attribute of controls is set to “on”

Before creating controls the application empties the container.

Example

```
function on_filter_form_created(item) {
    item.filter_options.title = item.item_caption + ' - filter';
    item.create_filter_inputs(item.filter_form.find(".edit-body"));
    item.filter_form.find("#cancel-btn").on('click.task', function() {
        item.close_filter()
    });
    item.filter_form.find("#ok-btn").on('click.task', function() {
        item.apply_filter()
    });
}
```

See also

filters

create_filter_form

*filter_form**filter_options***create_inputs****create_inputs** (*container, options*)**domain:** client**language:** javascript**Description**

Use `create_inputs` to create data-aware visual controls (inputs, checkboxes) for editing *fields* of the item.

This method is usually used in the `on_edit_form_created` events.

The following parameters are passed to the method:

- `container` - a JQuery object that will contain visual controls, if container length is 0 (no container), the method returns.
- `options` - options that specify how controls are displayed

The `options` parameter is an object that may have following attributes:

- `fields` - a list of field names. If specified, a visual control will be create for each field whose name is in this list, if not specified then the `fields` attribute of *edit_options* will be used (if defined), otherwise the layout, created in the *Edit Form Dialog* of Application builder, will be created
- `col_count` - the number of columns that will be created for visual controls, the default value is 1,

Before creating controls the application empties the container.

Example

```
function on_edit_form_created(item) {
    item.create_inputs(item.edit_form.find(".left-div"),
        {fields: ['firstname', 'lastname', 'company', 'support_rep_id']}
    );
}
```

See also*fields**Data-aware controls**create_edit_form*

create_table

create_table (*container*, *options*)

domain: client

language: javascript

Description

Use `create_table` method to create a table that displays records of the item dataset.

The behavior of the table is determined by the *paginate* attribute of the item.

When *paginate* is true, a paginator will be created, that will internally update the item dataset when the page is changed.

If the value of *paginate* is false, all available records of the item dataset will be displayed in the table.

The table, created by this method is data aware, when you change the dataset, these changes are immediately reflected in the table. So you can create a table and then call the *open* method.

The following parameters could be passed to the method:

- *container* - a JQuery object that will contain the table, if container length is 0 (no container), the method returns. Before creating the table the application empties the container.
- *options* - options that specify the way the table will be displayed. By default, the method uses the *table_options* that are set in the *View Form Dialog* in Application Builder when creating the table. The *options* attributes take precedence over the *table_options* attributes.

The *options* parameter is an object that may have the same attributes as *table_options*.

Examples

```
function on_edit_form_created(item) {
    item.edit_options.width = 1050;
    item.invoice_table.create_table(item.edit_form.find(".edit-detail"),
    {
        height: 400,
        editable_fields: ['quantity'],
        column_width: {"track": "60%"}
    });
}
```

See also

Forms

Data-aware controls

create_view_form

create_view_form (*container*)

domain: client

language: javascript

class *Item class*

Description

Use `create_view_form` method to create a view form of the item.

Then it searches for an item html template in the task *templates* attribute (See *Forms*) and creates a clone of the template and assigns it to the item *view_form* attribute.

If `container` parameter is specified the method empties it and appends the html template to it. Otherwise it creates a modal form and appends the html to it.

Triggers the *on_view_form_created* of the task.

Triggers the *on_view_form_created* of the group that owners the item, if one is defined for the group.

Triggers the *on_view_form_created* of the item, if one is defined.

Assigns the JQuery keyup and keydown events to the *view_form* so that when an JQuery event of the window occurs, the *on_view_form_keyup* and *on_view_form_keydown* events are triggered. They are triggered (if defined) in the same way: first the task event handler, the group event handler and then the event handler of the item itself. After that the JQuery `stopPropagation` method of the event is called.

If the form is modal, shows it. Before showing the form the method applies options specified in the *view_options* attribute.

Triggers the *on_view_form_shown* of the task.

Triggers the *on_view_form_shown* of the group that owners the item, if one is defined for the group.

Triggers the *on_view_form_shown* of the item, if one is defined.

Forms

view_form

view_options

close_view_form

delete

delete ()

domain: client

language: javascript

class *Item class*

Description

Deletes the active record and positions the cursor on the next record.

The `delete` method

- checks if item dataset is *active*, otherwise raises exception

- checks if item dataset is not empty, otherwise raises exception
- if item is a *detail* , checks if the master item is in edit or insert *state*, otherwise raises exception.
- if item is not a *detail* , checks if it is in browse *state*, otherwise raises exception.
- triggers the *on_before_delete* event handler if one is defined for the item.
- puts the item into delete *state*
- deletes the active record and positions the cursor on the next record
- puts the item into browse *state*
- triggers the *on_after_delete* event handler if one is defined for the item
- updates *data-aware controls*

See also

Modifying datasets

delete_record

delete_record ()

domain: client

language: javascript

class *Item class*

Description

- calls the *can_delete* method to check whether a user have a right to delete a record, and if not, returns
- asks a user to confirm the operation
- calls the *delete* method to delete the record
- calls the *apply* method to write changes to the application database

See also

Modifying datasets

delete

disable_controls

disable_controls ()

domain: client

language: javascript

class *Item class*

Description

Call `disable_controls` to “turn off” data-aware controls, so they will not reflect changes to the item dataset data.

Call [*enable_controls*](#) to re-enable data display in data-aware controls associated with the dataset and update values they display.

Example

```
function calculate(item) {
    var subtotal,
        tax,
        total,
        rec;
    if (!item.calculating) {
        item.calculating = true;
        try {
            subtotal = 0;
            tax = 0;
            total = 0;
            item.invoice_table.disable_controls();
            rec = item.invoice_table.rec_no;
            try {
                item.invoice_table.each(function(d) {
                    subtotal += d.amount.value;
                    tax += d.tax.value;
                    total += d.total.value;
                });
            }
            finally {
                item.invoice_table.rec_no = rec;
                item.invoice_table.enable_controls();
            }
            item.subtotal.value = subtotal;
            item.tax.value = tax;
            item.total.value = total;
        }
        finally {
            item.calculating = false;
        }
    }
}
```

See also

Data-aware controls

enable_controls

disable_edit_form

`disable_edit_form()`

domain: client

language: javascript

class *Item class*

Description

Call `disable_edit_form` to prevent any user actions when *edit_form* is visible.

Call *enable_edit_form* to re-enable edit form.

Example

```
function on_edit_form_created(item) {
    var save_btn = item.add_edit_button('Save and continue');
    save_btn.click(function() {
        if (item.is_changing()) {
            item.disable_edit_form();
            item.post();
            item.apply(function(error) {
                if (error) {
                    item.alert_error(error);
                }
                item.edit();
                item.enable_edit_form();
            });
        }
    });
}
```

See also

enable_edit_form

each

each (*function(item)*)

domain: client

language: javascript

class *Item class*

Description

Use each method to iterate over records of an item dataset.

The `each()` method specifies a function to run for each record. You can break the each loop at a particular iteration by making the callback function return `false`.

Example

In the example below the **t** and **item.invoice_table** are pointers to the same object:

```

var subtotal = 0,
    tax = 0,
    total = 0;
item.invoice_table.each(function(t) {
    subtotal += t.amount.value;
    tax += t.tax.value;
    total += t.total.value;
});

```

See also

Navigating datasets

each_detail

each_detail (*function(detail)*)

domain: client

language: javascript

class *Item class*

Description

Use each method to iterate over details of an item.

The each_detail() method specifies a function to run for each detail of the item (the current detail is passed as a parameter).

You can break the each_detail loop at a particular iteration by making the callback function return `false`.

See also

Details

each_field

each_field (*function(field)*)

domain: client

language: javascript

class *Item class*

Description

Use `each_field` method to iterate over *fields* owned by an item.

The `each_field()` method specifies a function to run for each field (the current field is passed as a parameter).

You can break the `each_field` loop at a particular iteration by making the callback function return `false`.

Example

```
function customer_fields(customers) {
  customers.open({limit: 1});
  customers.each_field(function(f) {
    console.log(f.field_caption, f.display_text);
  });
}
```

Fields

Field class

`each_filter`

`each_filter` (*function(filter)*)

domain: client

language: javascript

class *Item class*

Description

Use `each_filter` method to iterate over *filters* owned by an item.

The `each_filter()` method specifies a function to run for each filter (the current filter is passed as a parameter).

You can break the `each_filter` loop at a particular iteration by making the callback function return `false`.

Example

```
function customer_filters(customers) {
  customers.each_filter(function(f) {
    console.log(f.filter_caption, f.value);
  });
}
```

Filters

Filter class

edit

edit ()

domain: client

language: javascript

class *Item class*

Description

Enables editing of data in the dataset.

After a call to edit, an application can enable users to change data in the fields of the record, and can then post those changes to the item dataset using *post* method, and then apply them to database using *apply* method.

The `edit` method

- checks if the item dataset is active, otherwise raises exception
- checks if the item dataset is not empty, otherwise raises exception
- checks whether the item dataset is already in edit state, and if so, returns
- if item is a *detail*, checks if the master item is in edit or insert *state*, otherwise raises exception
- if item is not a *detail*, checks if it is in browse *state*, otherwise raises exception
- triggers the *on_before_edit* event handler if one is defined for the item
- puts the item into edit *state*, enabling the application or user to modify fields in the record
- triggers the *on_after_edit* event handler if one is defined for the item

See also

Modifying datasets

edit_record

edit_record (*container*)

domain: client

language: javascript

class *Item class*

Description

Puts the current record in edit *state* and creates an *edit_form* for visual editing of the record.

If `container` parameter (Jquery object of the DOM element) is specified the edit form html template is inserted in the container.

If `container` parameter is not specified but **Modeless form** attribute is set in the *Edit Form Dialog* or modeless attribute of the *edit_options* is set programmatically and task has the *forms_in_tabs* attribute set and the application

doesn't have modal forms, the modeless edit form will be created in the new tab of the *forms_container* object of the task.

In all other cases the modal form will be created.

If editing is allowed in modeless mode, the user can edit several records at the same time. In this case the application calls the *copy* method to create a copy of the item. This copy will be used to edit the record. The application will call its *open* method to get the record from the server by using the value of the primary key field as a filter.

In case of modal editing the application executes *refresh_record* methods to get from the server the latest data of the record.

If a *record locking* is enabled for the item, along with receiving the record data from the server the application receives the version of the record.

Then the `edit_record` method

- calls the *can_edit* method to check whether a user have a right to edit the record,
- if the user have a right to edit the record, checks whether the item is in edit or insert *state* , and if not, calls the *edit* method to edit the record
- calls the *create_edit_form* method to create a form for visual editing of the record

See also

Forms

Modifying datasets

edit

can_create

Record locking

enable_controls

enable_controls ()

domain: client

language: javascript

class *Item class*

Description

Call `enable_controls` to permit data display in data-aware controls and and redraw them after a prior call to *disable_controls*.

See also

Data-aware controls

disable_controls.

enable_edit_form

enable_edit_form()

domain: client

language: javascript

class *Item class*

Description

Call `enable_edit_form` to re-enable edit form after prior call to *disable_edit_form*

Example

```
function on_edit_form_created(item) {
    var save_btn = item.add_edit_button('Save and continue');
    save_btn.click(function() {
        if (item.is_changing()) {
            item.disable_edit_form();
            item.post();
            item.apply(function(error) {
                if (error) {
                    item.alert_error(error);
                }
                item.edit();
                item.enable_edit_form();
            });
        }
    });
}
```

See also

disable_edit_form

eof

eof()

domain: client

language: javascript

class *Item class*

Description

Test `eof` (end-of-file) to determine if the cursor is positioned at the last record in an item dataset. If `eof` returns `true`, the cursor is unequivocally on the last row in the dataset. `eof` returns `true` when an application:

- Opens an empty dataset.

- Calls an item's *last* method.
- Call an item's *next* method, and the method fails (because the cursor is already on the last row in the dataset).

eof returns false in all other cases.

Note: If both eof and *bof* return `true`, the item dataset is empty.

See also

Dataset

Navigating datasets

field_by_name

field_by_name (*field_name*)

domain: client

language: javascript

class *Item class*

Description

Call `field_by_name` to retrieve field information for a field when only its name is known.

The `field_name` parameter is the name of an existing field.

`field_by_name` returns the field object for the specified field. If the specified field does not exist, `field_by_name` returns null.

filter_by_name

filter_by_name (*filter_name*)

domain: client

language: javascript

class *Item class*

Description

Call `filter_by_name` to retrieve filter information for a filter when only its name is known.

The `filter_name` parameter is the name of an existing filter.

`filter_by_name` returns the filter object for the specified filter. If the specified filter does not exist, `filter_by_name` returns null.

first

first ()

domain: client

language: javascript

class *Item class*

Description

Call `first` to position the cursor on the first record in the item dataset and make it the active record. `First` posts any changes to the active record.

See also

Dataset

Navigating datasets

insert

insert ()

domain: client

language: javascript

class *Item class*

Description

Inserts a new, empty record in the item dataset.

After a call to `insert`, an application can enable users to enter data in the fields of the record, and can then post those changes to the item dataset using `post` method, and then apply them to the item database table, using `apply` method.

The `insert` method

- checks if item dataset is *active* , otherwise raises exception
- if the item is a *detail* , checks if the master item is in edit or insert *state* , otherwise raises exception
- if the item is not a *detail* checks if it is in browse *state* , otherwise raises exception
- triggers the *on_before_append* event handler if one is defined for the item
- inserts a new, empty record in the item dataset.
- puts the item into insert *state*
- triggers the *on_after_append* event handler if one is defined for the item.
- updates *data-aware controls*

See also

Modifying datasets

insert_record

insert_record (*container*)

domain: client

language: javascript

class *Item class*

Description

Open a new, empty record at the beginning of the dataset and creates an *edit_form* for visuall editing of the record.

If *container* parameter (Jquery object of the DOM element) is specified the edit form html template is inserted in the container.

If *container* parameter is not specified but **Modeless form** attribute is set in the *Edit Form Dialog* or modeless attribute of the *edit_options* is set programmatically and task has the *forms_in_tabs* attribute set and the application doesn't have modal forms, the modeless edit form will be created in the new tab of the *forms_container* object of the task.

In all other cases the modal form will be created.

If insertion of a record is allowed in modeless mode, the application calls the *copy* method to create a copy of the item. This copy will be used to insert the record.

The *insert_record* method

- calls the *can_create* method to check whether a user have a right to insert a record, and if not, returns
- checks whether the item is in edit or insert *state* , and if not, calls the *insert* method to insert a record
- calls the *create_edit_form* method to create a form for visuall editing of the record

See also

Forms

Modifying datasets

insert

can_create

is_changing

is_changing ()

domain: client

language: javascript

class *Item class*

Description

Checks if an item is in edit or insert state and returns `true` if it is.

An application calls *edit* to put an item into edit state and *append* or *insert* to put an item into insert state.

See also

Modifying datasets

is_edited

is_edited()

domain: client

language: javascript

class *Item class*

Description

Checks if an item is in edit state and returns `true` if it is.

An application calls *edit* to put an item into edit state.

See also

Modifying datasets

is_modified

is_modified()

domain: client

language: javascript

class *Item class*

Description

Checks if the current record of an item dataset has been modified during edit or insert opertaions. The method returns `false` after the *post* method is executed.

See also

Modifying datasets

is_new

is_new ()

domain: client

language: javascript

class *Item class*

Description

Checks if an item is in insert state and returns `true` if it is.

An application calls *append* or *insert* methods to put an item into insert state.

See also

Modifying datasets

last

last ()

domain: client

language: javascript

class *Item class*

Description

Call `last` to position the cursor on the last record in the item dataset and make it the active record.

See also

Dataset

Navigating datasets

locate

locate (*fields, values*)

domain: client

language: javascript

class *Item class*

Description

Implements a method for searching an item dataset for a specified record and makes that record the active record.

Arguments:

- `fields`: a field name, or list of field names
- `values`: a field value or list of field values

This method locates the record where the fields specified by `fields` parameter have the values specified by `values` parameter.

`Locate` returns true if a record is found that matches the specified criteria and the cursor repositioned to that record.

If a matching record was not found and the cursor is not repositioned, this method returns false.

See also

Dataset

Navigating datasets

next

next ()

domain: client

language: javascript

class *Item class*

Description

Call `next` to position the cursor on the next record in the item dataset and make it the active record. Next posts any changes to the active record.

See also

Dataset

Navigating datasets

open

open (*options, callback, async*)

domain: client

language: javascript

class *Item class*

Description

Call `open` to send a request to the server for obtaining an item dataset.

The `open` method can have the following parameters:

- `options` - an object that specifies the parameters of the request sent to the server
- `callback`: if the parameter is not present, the request is sent to the server synchronously, otherwise, the request is executed asynchronously and after the dataset is received, the callback is executed
- `async`: if its value is true, and `callback` parameter is missing, the request is executed asynchronously

The order of parameters doesn't matter.

The method initializes the item *fields*, formulates parameters of a request, based on the `options` and triggers the *on_before_open* event handler if one is defined for the item.

After that it sends the request to the server. If `callback` parameter-function is specified, the request is executed asynchronously, otherwise - synchronously.

The server, after receiving the request, checks if the corresponding item on the server (item of the *task tree* with the same ID attribute) has the *on_open* event handler. If so it executes this event handler and returns the result of the execution to the client, otherwise generates a SELECT SQL query, based on parameters of the request, executes this query and returns the result to the client.

The client, after receiving the result of the request, changes its dataset and sets *active* to true, the *item_state* to browse mode, goes to the first record of the dataset, triggers *on_after_open* and *on_filters_applied* event handlers (if they are defined for the item), and updates controls.

Then it calls `callback` function if it was specified.

Options

The `options` object parameter can have the following attributes:

- `expanded` - if the value of this attribute is true, the SELECT query, generated on the server, will have JOIN clauses to get lookup values of the *lookup fields*, otherwise no lookup values will be returned. The default value is `true`.
- `fields` - use this parameter to specify the WHERE clause of the SELECT query. This parameter is a list of field names. If it is omitted, the fields defined by the *set_fields* method will be used. If the *set_fields* method was not called before the `open` method execution, all the fields created by a developer will be used.
- `where` - use this parameter to specify how records will be filtered in the SQL query. This parameter is an object of key-value pairs, where keys are field names, that are followed, after double underscore, by a filtering symbols (see *Filtering records*). If this parameter is omitted, values defined by the *set_where* method will be used. If the *set_where* method was not called before the `open` method execution, and `where` parameter is omitted, then the values of *filters* defined for the item will be used to filter records.
- `order_by` - use `order_by` to specify sort order of the records. This parameter is a list of field names. If there is a sign '-' before the field name, then on this field records will be sorted in decreasing order. If this parameter is omitted, a list defined by the *set_order_by* method will be used.
- `offset` - use `offset` to specify the offset of the first row to return.
- `limit` - use `limit` to limit the output of a SQL query to the first so-many rows.
- `funcs` - this parameter can be an object of key-value pairs, where key is a field name and value is function name that will be applied to the field in the SELECT Query

- `group_by` - use `group_by` to specify fields to group the result of the query by. This parameter must be a list of field names.
- `open_empty` - if this parameter is set to `true`, the application does not send a request to the server but just initializes an empty dataset. The default value is `false`.
- `params` - use the parameter to pass some user defined options to be used in the `on_open` event handler on the server. This parameter must be an object of key-value pairs

Note: When the `paginate` attribute of the item is set to `true` and a table is created by the `create_table` method, the `limit` and `offset` parameters are set internally by the table depending on its row number and current page.

Examples

```
function get_customer_sales(task, customer_id) {
    var date1 = new Date(new Date().setYear(new Date().getFullYear() - 5)),
        date2 = new Date(),
        invoices = task.invoices.copy();

    invoices.open({
        fields: ['customer', 'invoicedate', 'total'],
        where: {customer: customer_id, invoicedate__ge: date1, invoicedate__le: date2}
        ↪,
        order_by: ['invoicedate']
    });
}
```

```
function get_customer_sales(task, customer_id) {
    var date1 = new Date(new Date().setYear(new Date().getFullYear() - 5)),
        date2 = new Date(),
        invoices = task.invoices.copy();

    invoices.set_fields(['customer', 'invoicedate', 'total']);
    invoices.set_where({customer: customer_id, invoicedate__ge: date1, invoicedate__
    ↪le: date2});
    invoices.set_order_by(['invoicedate']);
    invoices.open();
}
```

```
function get_sales(task) {
    var sales = task.invoices.copy();

    sales.open({
        fields: ['customer', 'id', 'total'],
        funcs: {'id': 'count', 'total': 'sum'},
        group_by: ['customer'],
        order_by: ['customer']
    });
}
```

post

`post()`

domain: client

language: javascript

class *Item class*

Description

Writes a modified record to the item dataset. Call `post` to save changes made to a record after *append*, *insert* or *edit* method was called.

The `post` method

- checks if an item is in edit or insert *state* , otherwise raises exception
- triggers the *on_before_post* event handler if one is defined for the item
- checks if a record is valid, if not raises exception
- If an item has *details* , post current record in details
- add changes to an item change log
- puts the item into browse *state*
- triggers the *on_after_post* event handler if one is defined for the item.

See also

Modifying datasets

prior

`prior()`

domain: client

language: javascript

class *Item class*

Description

Call `prior` to position the cursor on the previous record in the item dataset and make it the active record. last posts any changes to the active record.

See also

Dataset

Navigating datasets

record_count

record_count ()

domain: client

language: javascript

class *Item class*

Description

Call `record_count` to get the total number of records owned by the item's dataset.

Example

```
item.open()
if (item.record_count()) {
    // some code
}
```

See also

Dataset

open

refresh_page

refresh_page (*callback, async*)

domain: client

language: javascript

class *Item class*

Description

Call `refresh_page` to send to the server a request to get current data of the current page and refresh existing visual controls.

The `refresh_page` method can have the following parameters:

- `callback`: if the parameter is not present, the request is sent to the server synchronously, otherwise, the request is executed asynchronously and after that the callback is executed
- `async`: if its value is true, and `callback` parameter is missing, the request is executed asynchronously

refresh_record

refresh_record (*options, callback, async*)

domain: client

language: javascript

class *Item class*

Description

Call `refresh_record` to send to the server a request to get current data of the current record and refresh existing visual controls.

The `refresh_record` method can have the following parameters:

- **callback:** if the parameter is not present, the request is sent to the server synchronously, otherwise, the request is executed asynchronously and after that the callback is executed
- **async:** if its value is true, and callback parameter is missing, the request is executed asynchronously
- **options** - an object that can have an attribute `details` - a list of `item_names` of details the item. These details are refreshed too.

The order of the parameters does not matter

search

search (*field_name, value, search_type, callback*)

domain: client

language: javascript

class *Item class*

Description

Call `search` to send to the server a request to generate and execute an sql query to get all records which satisfy the search condition for the field. The query will also satisfy currently set filteres or where condition for an item. The existing visual controls will be update with the returned dataset.

Parameters:

- **field_name** - name of the field
- **value** - value of the condition
- **search_type** - type of search as a string, see Filter symbol in *Filtering records*
- **callback** - a callback function that will be executed after search is executed

See also

Dataset

Filtering records

select_records

select_records (*field_name*, *all_records*)

domain: client

language: javascript

class *Item class*

Description

Use the `select_records` method to add records to an item by selecting them from the lookup item of a field.

For example, this method is used in the Demo application to add tracks to an invoice by selecting them from Tracks catalog.

Parameters:

- The `field_name` parameter is a field name of a lookup field of the item
- If the `all_records` parameter is set to true, all selected records are added, otherwise the method omits existing records (they were selected earlier).

Example

```
function on_view_form_created(item) {  
    var btn = item.add_view_button('Select', {type: 'primary'});  
    btn.click(function() {  
        item.select_records('track');  
    });  
}
```

set_fields

set_fields (*field_list*)

domain: client

language: javascript

class *Item class*

Description

Use the `set_fields` method to define and store internally the `fields` option that will be used by the *open* method, when its own `fields` option is not specified.

After the *open* method executes it clears this internally stored value.

The `field_list` parameter is a list of field names.

Example

The result of the execution of following code snippets will be the same:

```
item.open({fields: ['id', 'invoicedate']});
```

```
item.set_fields(['id', 'invoicedate']);  
item.open();
```

See also

Dataset

open

set_order_by

set_order_by (*field_list*)

domain: client

language: javascript

class *Item class*

Description

Use the `set_order_by` method to define and store internally the `order_by` option that will be used by the *open* method, when its own `order_by` option is not specified. The *open* method clears internally stored parameter value.

The `field_list` parameter is a list of field names. If there is a sign '-' before a field name, then on this field records will be sorted in decreasing order.

Example

The result of the execution of following code snippets will be the same:

```
item.open({order_by: ['-invoicedate']});
```

```
item.set_order_by(['-invoicedate']);  
item.open();
```

See also

Dataset

open

set_where

set_where (*where*)

domain: client

language: javascript

class *Item class*

Description

Use the `set_where` method to define and store internally the `where` option that will be used by the *open* method, when its own `where` option is not specified. The *open* method clears internally stored parameter value.

The `where` parameter is an object of key-value pairs, where keys are field names, that are followed, after double underscore, by a filtering symbols (see *Filtering records*).

Example

The result of the execution of following code snippets will be the same:

```
item.open({where: {id: 100}});
```

```
item.set_where({id: 100});  
item.open();
```

See also

Dataset

open

show_history

show_history ()

domain: client

language: javascript

class *Item class*

Description

Class `show_history` method of an item to open a dialog displaying history of changes of the selected record

See also

Saving the history of changes made by users

update_controls

update_controls ()

domain: client

language: javascript

class *Item class*

Description

Call *update_controls* to tell associated controls to redraw to reflect current data.

See also

Data-aware controls

disable_controls

enable_controls

view

view (*container*)

domain: client

language: javascript

class *Item class*

Description

Use view method to create a view form of the item.

The method check if the javascript modules of the item and its owner are loaded, and if not (the **Dynamic JS modules loading** *parameter of the project* is set) then loads them.

If *container* parameter (Jquery object of the DOM element) is specified the view form html template is inserted in the container.

If the *init_tabs* method of the task is called for this container the tab is created for this form.

After that it calls the *create_view_form* method

Example

In the following code the view for of the **Tasks** journal is created in the *on_page_loaded* event handler:

```
function on_page_loaded(task) {  
  
    $("#title").html(task.item_caption);  
    if (task.safe_mode) {
```

(continues on next page)

(continued from previous page)

```

        $("#user-info").text(task.user_info.role_name + ' ' + task.user_info.user_
↪name);
        $('#log-out')
        .show()
        .click(function(e) {
            e.preventDefault();
            task.logout();
        });
    }

    task.init_tabs($("#content"));
    task.tasks.view($("#content"));

    $(window).on('resize', function() {
        resize(task);
    });
}

```

See also

Forms

view_form

view_options

create_view_form

close_view_form

Events

on_after_append

on_after_append(item)

domain: client

language: javascript

class *Item class*

Description

Occurs after an application inserts or appends a record.

The `item` parameter is an item that triggered the event.

Write an `on_after_append` event handler to take specific action immediately after an application inserts or appends a record in an item. `on_after_append` is called by *insert* or *append* method.

See also

Modifying datasets

on_after_apply

on_after_apply(item)

domain: client

language: javascript

class *Item class*

Description

Occurs after an application saves the change log to the project database.

The `item` parameter is an item that triggered the event.

Write an `on_after_apply` event handler to take specific action immediately after an application saves data changes to the project database.

On_after_apply is triggered by *apply* method.

See also

Modifying datasets

on_after_cancel

on_after_cancel(item)

domain: client

language: javascript

class *Item class*

Description

Occurs after an application cancels modifications made to the item dataset.

The `item` parameter is an item that triggered the event.

Write an `on_after_cancel` event handler to take specific action immediately after an application cancels modifications made to the item dataset.

See also

Modifying datasets

on_after_delete

on_after_delete(item)

domain: client

language: javascript

class *Item class*

Description

Occurs after an application deletes a record.

The `item` parameter is an item that triggered the event.

Write an `on_after_delete` event handler to take specific action immediately after an application deletes the active record in an item. `on_after_delete` is called by *delete* after it deletes the record, and repositions the cursor on the record prior to the one just deleted.

See also

Modifying datasets

on_after_edit

on_after_edit(item)

domain: client

language: javascript

class *Item class*

Description

Occurs after an application starts editing a record.

The `item` parameter is an item that triggered the event.

Write an `on_after_delete` event handler to take specific action immediately after an application starts editing a record. `on_after_edit` is called by *edit*.

See also

Modifying datasets

on_after_open

on_after_open(item)

domain: client

language: javascript

class *Item class*

Description

Occurs after an application receives a response from the server for obtaining a dataset.

The `item` parameter is an item that triggered the event.

Write an `on_after_open` event handler to take specific action immediately after an application obtains an dataset from the server. `on_after_open` is called by *open* method.

See also

Dataset

`on_after_post`

`on_after_post(item)`

domain: client

language: javascript

class *Item class*

Description

Occurs after an application posts a record to the item dataset.

The `item` parameter is an item that triggered the event.

Write an `on_after_post` event handler to take specific action immediately after an application posts a record in the item dataset. `on_after_post` is called by *post* method.

See also

Modifying datasets

`on_after_scroll`

`on_after_scroll(item)`

domain: client

language: javascript

class *Item class*

Description

Occurs after an application scrolls from one record to another.

The `item` parameter is an item that triggered the event.

Write an `on_after_scroll` event handler to take specific action immediately after an application scrolls to another record as a result of a call to the *first*, *last*, *next*, *prior*, and *locate* methods. `on_after_scroll` is called after all other events triggered by these methods and any other methods that switch from record to record in the item dataset.

Example

The following code is used in the *Demo project* to asynchronously open **invoice_table** detail dataset after the **Invoice** journal record has changed:

```
var ScrollTimeout;

function on_after_scroll(item) {
    clearTimeout(ScrollTimeout);
    ScrollTimeout = setTimeout(
        function() {
            item.invoice_table.open(function() {});
        },
        100
    );
}
```

See also

Navigating datasets

on_before_scroll

on_before_append

`on_before_append(item)`

domain: client

language: javascript

class *Item class*

Description

Occurs before an application inserts or appends a record.

The `item` parameter is an item that triggered the event.

Write an `on_before_append` event handler to take specific action immediately before an application inserts or appends a record in an item. `on_before_append` is called by *insert* or *append* method.

See also

Modifying datasets

on_before_apply

`on_before_apply(item, params)`

domain: client

language: javascript

class *Item class*

Description

Occurs before an application saves dataset changes to the project database.

The `item` parameter is an item that triggered the event.

The `params` parameter is an object that has been passed to the *apply* method or an empty object if this object is undefined. This object is passed to the server and can be used in the *on_apply* event handler to perform some actions when saving changes to the database.

Write an `on_before_apply` event handler to take specific action immediately before an application saves the change log to the project database.

`on_before_apply` is triggered by *apply* method.

See also

Modifying datasets

on_before_cancel

`on_before_cancel(item)`

domain: client

language: javascript

class *Item class*

Description

Occurs before an application cancels modifications made to the item dataset.

The `item` parameter is an item that triggered the event.

Write an `on_before_cancel` event handler to take specific action immediately before an application cancels modifications made to the item dataset.

See also

Modifying datasets

on_before_delete

on_before_delete(item)

domain: client

language: javascript

class *Item class*

Description

Occurs before an application deletes a record.

The `item` parameter is an item that triggered the event.

Write an `on_before_delete` event handler to take specific action immediately before an application deletes the active record in an item. `on_before_delete` is called by *delete* method before it deletes the record.

See also

Modifying datasets

on_before_edit

on_before_edit(item)

domain: client

language: javascript

class *Item class*

Description

Occurs before an application enables editing of the active record.

The `item` parameter an the item that triggered the event.

Write an `on_before_edit` event handler to take specific action immediately before an application enables editing of the active record in an item dataset. `on_before_edit` is called by *edit* method.

See also

Modifying datasets

on_before_field_changed

on_before_field_changed(field)

domain: client

language: javascript

class *Item class*

Description

Write an `on_before_field_changed` event handler to implement any special processing before field's data has been changed.

The `field` parameter is the field whose data is about to be changed. To get the item that owns the field, use the *owner* attribute of the field.

Before triggering this event handler the application assigns the new value that is about to be set to the `new_value` attribute to of the field. You can change the value of this attribute. This value will be used to change field's data.

Example

```
function on_before_field_changed(field) {  
    if (field.field_name === 'quantity' && field.new_value < 0) {  
        field.new_value = 0;  
    }  
}
```

See also

Fields

value

on_before_field_changed

on_before_open

on_before_open(item, params)

domain: client

language: javascript

class *Item class*

Description

Occurs before an application sends a request to the server for obtaining a dataset.

The `item` parameter is an item that triggered the event.

The `params` parameter is an object that has been passed to the *open* method or an empty object if this object is undefined. This object is passed to the server and can be used in the *on_open* event handler to perform some actions when obtaining a dataset

Write an `on_before_open` event handler to take specific action immediately before an application obtains an dataset from the server.

`on_before_open` is called by *open* method.

See also

Dataset

on_before_post

`on_before_post(item)`

domain: client

language: javascript

class *Item class*

Description

Occurs before an application posts a record to the item dataset.

The `item` parameter is an item that triggered the event.

Write an `on_before_post` event handler to take specific action immediately before an application posts a record in the item dataset. `on_before_post` is called by *post* method.

See also

Modifying datasets

on_before_scroll

`on_before_scroll(item)`

domain: client

language: javascript

class *Item class*

Description

Occurs before an application scrolls from one record to another.

The `item` parameter is an item that triggered the event.

Write an `on_before_scroll` event handler to take specific action immediately before an application scrolls to another record as a result of a call to the *first*, *last*, *next*, *prior*, and *locate* methods. `on_before_scroll` is called before all other events triggered by these methods and any other methods that switch from record to record in the item dataset.

See also

Navigating datasets

on_after_scroll

on_detail_changed

`on_detail_changed(item, detail)`

domain: client

language: javascript

class *Item class*

Description

Occurs after changes to detail record has been posted. It uses the `clearTimeout` and `setTimeout` Javascript functions so if records have been changed in a cycle it is triggered only when last record change occurs.

The `item` parameter is an item that triggered the event. The `detail` parameter is a detail that has been changed.

Write an `on_detail_changed` event handler to calculate, by using *calc_summary* method, sums for fields of a detail and save these values in fields of its master.

Example

```
function on_detail_changed(item, detail) {
    var fields;
    if (detail.item_name === 'invoice_table') {
        fields = [
            {"total": "total"},
            {"tax": "tax"},
            {"subtotal": "amount"}
        ];
        item.calc_summary(detail, fields);
    }
}
```

See also

Details calc_summary

on_edit_form_close_query

on_edit_form_close_query(item)

domain: client

language: javascript

class *Item class*

Description

The `on_edit_form_close_query` event is triggered by the *close_edit_form* method of the item.

The `item` parameter is the item that triggered the event.

See also

Forms

create_edit_form

edit_form

close_edit_form

on_edit_form_created

on_edit_form_created(item)

domain: client

language: javascript

class *Item class*

Description

The **on_edit_form_created** event is triggered by the *create_edit_form* method when the form has been created but not shown yet.

The `item` parameter is the item that triggered the event.

See also

Forms

create_edit_form

edit_form

on_edit_form_keydown

on_edit_form_keydown(item, event)

domain: client

language: javascript

class *Item class*

Description

The on_edit_form_keydown event is triggered when the keydown event occurs for the *edit form* of the item.

The `item` parameter is the item that triggered the event.

The `event` is JQuery event object.

See also

Forms

create_edit_form

edit_form

on_edit_form_keyup

on_edit_form_keyup(item, event)

domain: client

language: javascript

class *Item class*

Description

The on_edit_form_keyup event is triggered when the keyup event occurs for the *edit form* of the item.

The `item` parameter is the item that triggered the event.

The `event` is JQuery event object.

See also

Forms

create_edit_form

edit_form

on_edit_form_shown

on_edit_form_shown(item)

domain: client

language: javascript

class *Item class*

Description

The **on_edit_form_shown** event is triggered by the *create_edit_form* method when the form has been shown.

The `item` parameter is the item that triggered the event.

See also

Forms

create_edit_form

edit_form

on_field_changed

on_field_changed(field, lookup_item)

domain: client

language: javascript

class *Item class*

Description

Write an `on_field_changed` event handler to respond to any changes in the field's data.

The `field` parameter is the field whose data has been changed. To get the item that owns the field, use the *owner* attribute of the field.

The `lookup_item` parameter is not undefined when the field is a *lookup field* and a change has occurred when a user selected a record from a lookup item dataset.

Example

```
function on_field_changed(field, lookup_item) {
    var item = field.owner;
    if (field.field_name === 'quantity' || field.field_name === 'unitprice') {
        item.owner.calc_total(item);
    }
    else if (field.field_name === 'track' && lookup_item) {
        item.quantity.value = 1;
        item.unitprice.value = lookup_item.unitprice.value;
    }
}
```

(continues on next page)

(continued from previous page)

```
}  
}
```

See also

Fields

value

on_before_field_changed

on_field_get_html

on_field_get_html(field)

domain: client

language: javascript

class *Item class*

Description

Write an `on_field_get_html` event handler to specify the html that will be inserted in the table cell for the field.

If the event handler does not return a value, the application checks if the *on_field_get_text* event handler is defined and it returns a value, otherwise the *display_text* property value will be used to display the field value in the cell.

The `field` parameter is the field whose *display_text* is processed. To get the item that owns the field, use the *owner* attribute of the field.

Example

```
function on_field_get_html(field) {  
    if (field.field_name === 'total') {  
        if (field.value > 10) {  
            return '<strong>' + field.display_text + '</strong>';  
        }  
    }  
}
```

See also

Fields

on_field_get_text

on_field_select_value

on_field_select_value(field, lookup_item)

domain: client

language: javascript

class *Item class*

Description

When user clicks on the button to the right of the field input or uses typeahead, the application creates a copy of the lookup item of the field and triggers on_field_select_value event. Use on_field_select_value to specify fields that will be displayed, set up filters for the lookup item, before it will be opened.

The `field` parameter is the field whose data will be selected.

The `lookup_item` parameter is a copy of the lookup item of the field

Example

```
function on_field_select_value(field, lookup_item) {
  if (field.field_name === 'customer') {
    lookup_item.set_where({lastname__startswith: 'B'});
    lookup_item.view_options.fields = ['firstname', 'lastname', 'address', 'phone
↵'];
  }
}
```

See also

Fields

Lookup fields

on_field_validate

on_field_validate(field)

domain: client

language: javascript

class *Item class*

Description

Write an on_field_validate event handler to validate changes made to the field data.

The `field` parameter is the field whose data has been changed. To get the item that owns the field, use the *owner* attribute of the field.

The event handler must return a string if the field value is invalid. When an event handler returns a string, the application throws an exception.

The event is triggered when the *post* method is called or when the user leaves the input used to edit the field value.

Example

```
function on_field_validate(field) {  
    if (field.field_name === 'sum' && field.value > 10000000) {  
        return 'The sum is too big.';  
    }  
}
```

See also

Fields

value

How to validate field value

on_filter_changed

on_filter_changed(filter, lookup_item)

domain: client

language: javascript

class *Item class*

Description

Write an on_filter_changed event handler to respond to any changes in the filter's data.

The *filter* parameter is the filter whose data has been changed. To get the item that owns the filter, use the *owner* attribute of the filter.

See also

Filters

value

on_filter_form_close_query

on_filter_form_close_query(item)

domain: client

language: javascript

Description

The `on_filter_form_close_query` event is triggered by the `close_filter_form` method of the item.

The `item` parameter is the item that triggered the event.

See also

Forms

create_filter_form

filter_form

close_filter_form

on_filter_form_created

`on_filter_form_created(item)`

The `item` parameter is the item that triggered the event.

domain: client

language: javascript

Description

The **on_filter_form_created** event is triggered by the `create_filter_form` method when the form has been created but not shown yet.

The `item` parameter is the item that triggered the event.

See also

Forms

create_filter_form

filter_form

on_filter_form_shown

`on_filter_form_shown(item)`

The `item` parameter is the item that triggered the event.

domain: client

language: javascript

Description

The `on_filter_form_shown` event is triggered by the `create_filter_form` method when the form has been shown.

The `item` parameter is the item that triggered the event.

See also

Forms

create_filter_form

filter_form

on_filter_record

`on_filter_record(item)`

The `item` parameter is the item that triggered the event.

domain: client

language: javascript

Description

Use an `on_filter_record` event to filter dataset records locally. It is triggered when the cursor moves to another record and *Filtered* property is set to `true`

Write an `on_filter_record` event handler to specify for each record in a dataset whether it should be visible to the application. To indicate that a record passes the filter condition, the `on_filter_record` event handler must return `true`.

The `item` parameter is the item that triggered the event.

Example

```
function on_filter_record(item) {
    if (item.type.value === 2) {
        return true;
    }
}

function enable_filtering(item) {
    item.filtered = true;
}

function disable_filtering(item) {
    item.filtered = false;
}
```

See also

Filtered

on_filters_applied

on_filters_applied(item)

domain: client

language: javascript

class *Item class*

Description

Write an `on_filters_applied` event handler to make special processing when filters have been applied to the item dataset.

See also

Filters

on_field_get_text

on_field_get_text(field)

domain: client

language: javascript

class *Item class*

Description

Write an `on_field_get_text` event handler to perform custom processing for the *display_text* property. If the event handler does not return a value, the application uses the *display_text* property value to display the field value in the data-aware controls, otherwise the returned value will be used.

The `field` parameter is the field whose *display_text* is processed. To get the item that owns the field, use the *owner* attribute of the field.

Example

```
function on_field_get_text(field) {
    if (field.field_name === 'customer') {
        return field.owner.firstname.lookup_text + ' ' + field.lookup_text;
    }
}
```

See also

Fields

on_view_form_close_query

on_view_form_close_query(item)

domain: client

language: javascript

class *Item class*

Description

The on_view_form_close_query event is triggered by the *close_view_form* method of the item.

The `item` parameter is the item that triggered the event.

See also

Forms

view

view_form

close_view_form

on_view_form_created

on_view_form_created(item)

domain: client

language: javascript

class *Item class*

Description

The on_view_form_created event is triggered by the *view* method when form has been created but not shown yet.

The `item` parameter is the item that triggered the event.

See also

Forms

view

view_form

on_view_form_keydown

on_view_form_keydown(item, event)

domain: client

language: javascript

class *Item class*

Description

The on_view_form_keydown event is triggered when the keydown event occurs for the *view form* of the item.

The `item` parameter is the item that triggered the event.

The `event` is JQuery event object.

See also

Forms

view

view_form

on_view_form_keyup

on_view_form_keyup(item, event)

domain: client

language: javascript

class *Item class*

Description

The on_view_form_keyup event is triggered when the keyup event occurs for the *view form* of the item.

The `item` parameter is the item that triggered the event.

The `event` is JQuery event object.

See also

Forms

view

view_form

on_view_form_shown

on_view_form_shown(item)

domain: client

language: javascript

class *Item class*

Description

The **on_view_form_shown** event is triggered by the *view* method of the item when the form has been shown.

The **item** parameter is the item that triggered the event.

See also

Forms

view

view_form

7.1.5 Detail class

class **Detail**()

domain: client

language: javascript

Detail class inherits attributes, methods and events of *Item class*

Attributes

master

master

domain: client

language: javascript

class *Detail class*

Description

Use `master` attribute to get reference to the master of the detail.

See also

Details

7.1.6 Reports class

class `Reports()`

domain: client

language: javascript

Reports class is used to create the group object of the *task tree* that owns the reports of a project.

Below the events of the class are listed.

It, as well, inherits attributes and methods of its ancestor class *AbstractItem class*

Events

`on_before_print_report`

`on_before_print_report(item)`

domain: client

language: javascript

class *Reports class*

Description

The `on_before_print_report` event is triggered by the *process_report* method.

The `report` parameter is the report that triggered the event.

See also

Forms

Client-side report programming

process_report

`on_open_report`

`on_open_report(report)`

domain: client

language: javascript

class *Reports class*

Description

The `on_open_report` event is triggered by the *process_report* method.

The `report` parameter is the report that triggered the event.

See also

Client-side report programming

process_report

on_param_form_close_query

on_param_form_close_query(item)

domain: client

language: javascript

class *Reports class*

Description

The on_param_form_close_query event is triggered by the *close_param_form* method.

The report parameter is the report that triggered the event.

See also

Forms

Client-side report programming

print

create_param_form

on_param_form_created

on_param_form_created(item)

domain: client

language: javascript

class *Reports class*

Description

The on_param_form_created event is triggered by the *create_param_form* method, that, usually, is called by then *print* method.

The report parameter is the report that triggered the event.

See also

Forms

Client-side report programming

print

create_param_form

on_param_form_shown

on_param_form_shown(item)

domain: client

language: javascript

class *Reports class*

Description

The on_param_form_shown event is triggered by the *create_param_form* method, that, usually, is called by then *print* method.

See also

Forms

Client-side report programming

print

create_param_form

7.1.7 Report class

class *Report* ()

domain: client

language: javascript

Report class inherits

Below the attributes, methods and events of the class are listed.

It, as well, inherits attributes and methods of its ancestor class *AbstractItem class*

Attrubutes

extension

extension

domain: client

language: javascript

class *Report class*

Description

Use `extension` attribute to specify a report type. The server, based on the report template, first generates **ods** file. And if report extension is other than **ods** performs conversion using the LibreOffice.

The attribute value can be any extension that LibreOffice supports conversion to.

Example

```
function on_before_print_report(report) {  
    report.extension = 'html';  
}
```

See also

Client-side report programming

Server-side report programming

print

create_param_form

on_before_print_report

param_form

param_form

domain: client

language: javascript

class *Report class*

Description

Use `param_form` attribute to get access to a JQuery object representing the param form of the report.

It is created by the *create_param_form* method, that, usually, is called by then *print* method.

The *close_param_form* method sets the `param_form` value to undefined.

Example

```
function on_param_form_created(report) {
    report.create_param_inputs(report.param_form.find(".edit-body"));
    report.param_form.find("#cancel-btn").on('click.task', function() {
        report.close_param_form();
    });
    report.param_form.find("#ok-btn").on('click.task', function() {
        report.process_report();
    });
}
```

See also

Forms

print

create_param_form

close_param_form

param_options

param_options

domain: client

language: javascript

class *Report class*

Description

Use the `param_options` attribute to specify parameters of the modal param form.

`param_options` is an object that has the following attributes:

- `width` - the width of the modal form, the default value is 560 px,
- `title` - the title of the modal form, the default value is the value of a *report_caption* attribute,
- `close_button` - if true, the close button will be created in the upper-right corner of the form, the default value is true,
- `close_caption` - if true and `close_button` is true, will display 'Close - [Esc]' near the button
- `close_on_escape` - if true, pressing on the Escape key will trigger the *close_param_form* method.
- `close_focusout` - if true, the *close_param_form* method will be called when a form loses focus
- `template_class` - if specified, the div with this class will be searched in the task *templates* attribute and used as a form html template when creating a form

Example

```
function on_param_form_created(report) {  
    report.param_options.width = 800;  
    report.param_options.close_button = false;  
    report.param_options.close_on_escape = false;  
}
```

See also

Forms

print

create_param_form

close_param_form

Methods

close_param_form

close_param_form()

domain: client

language: javascript

class *Report class*

Description

Use `close_param_form` method to close the param form of the report.

The `close_param_form` method triggers the *on_param_form_close_query* event handler of the report, if one is defined. If the event handler is defined and

- returns true - the form is destroyed, the report's *param_form* attribute is set to undefined and the methods exits
- return false - the operation is aborted and the methods exits,

If it don't return a value (undefined) the method triggers the *on_param_form_close_query* of the group that owners the report, if one is defined for the group. If this event handler is defined and

- returns true - the form is destroyed, the report's *param_form* attribute is set to undefined and the methods exits
- return false - the operation is aborted and the methods exits,

If it don't return a value (undefined) the method triggers the *on_param_form_close_query* of the task. If this event handler is defined and

- returns true - the form is destroyed, the report's *param_form* attribute is set to undefined and the methods exits
- return false - the operation is aborted and the methods exits,

If no event handler is defined or none of these event handlers return false, the form is destroyed and the report's *param_form* attribute is set to undefined.

See also

Forms

Client-side report programming

print

create_param_form

create_param_form()

domain: client

language: javascript

class *Report class*

Description

The `create_param_form` method is called by the *print* method to create a form to set report parameters before sending a request to the server by the *process_report* method.

The method checks if javascript modules of the report and its owner are loaded, and if not (the *Dynamic JS modules loading parameter* is set) then loads them.

Then it searches for the report html template in the task *templates* attribute (See *Forms*) and creates a clone of the template and assigns it to the report *param_form* attribute.

Creates a form and appends the html to it.

Triggers the *on_param_form_created* of the task.

Triggers the *on_param_form_created* of the report group, if one is defined.

Triggers the *on_param_form_created* of the report, if one is defined.

Shows the form. Before showing the form the method applies options specified in the *param_options* attribute.

Triggers the *on_param_form_shown* of the task.

Triggers the *on_param_form_created* of the report group, if one is defined.

Triggers the *on_param_form_shown* of the report, if one is defined.

See also

Forms

Client-side report programming

print

create_param_inputs

create_param_inputs (*container, options*)

domain: client

language: javascript

Description

Use `create_param_inputs` to create data-aware visual controls (inputs, checkboxes) for editing of report parameters.

This method is usually used in `on_param_form_created` events triggered by `create_param_form` method, that, usually, is called by then `print` method.

The following parameters are passed to the method:

- `container` - a JQuery object that will contain visual controls, if container length is 0 (no container), the method returns.
- `options` - options that specify how controls are displayed

The `options` parameter is an object that may have following attributes:

- `params` - a list of param names. If specified, a visual control will be created for each param whose name is in this list, if not specified (the default) then control will be created for all visible params specified in the Application builder
- `col_count` - the number of columns that will be created for visual controls, the default value is 1
- `label_on_top`: the default value is false. If this value is false, the labels are placed to the left of controls, otherwise the are created above the controls
- `tabindex` - if `tabindex` is specified, it will the `tabindex` of the first visual control, `tabindex` of all subsequent controls will be increased by 1.
- `autocomplete` - the default value is false. If this attribute is set to true, the `autocomplete` attribute of controls is set to "on"

Before creating controls the application empties the container.

Example

```
function on_param_form_created(item) {
    item.create_param_inputs(item.param_form.find(".edit-body"));
    item.param_form.find("#cancel-btn").on('click.task', function() {
        item.close_param_form()
    });
    item.param_form.find("#ok-btn").on('click.task', function() {
        item.process_report()
    });
}
```

See also

[*create_param_form*](#)

[*param_form*](#)

[*param_options*](#)

print

print (*create_form*)

domain: client

language: javascript

class *Report class*

Description

Use `print` to print the report.

If `create_form` parameter is omitted or equals `true`, the method calls the *create_param_form* method to create a form based on the html template defined in the `index.html` file.

If `create_form` parameter is set to `false` and the report has no visible parameters, it calls *process_report* to send request to server to generate the report, otherwise it calls *create_param_form* method.

See also

Forms

Report parameters

Client-side report programming

create_param_form

process_report

process_report

process_report ()

domain: client

language: javascript

class *Report class*

Description

The *process_report* method sends the report to the server to generate its content and accepts the report file that the server returns to the client and opens or saves it.

It is called by the *print* method directly, if its `create_form` parameter equals `false` and there are no visible parameters. If there are visible parameters, the *print* method creates a form to specify parameter values and the form should call it (for example, by some button onclick event).

The checks if parameter values are valid and the triggers the following events:

- *on_before_print_report* event handler of the report group
- *on_before_print_report* event handler of the report

In this event handlers developer can define some common (report group event handler) or specific (report event handler) attributes of the report.

After that the *process_report* method sends asynchronous request to the server to generate a report content. (see *Server-side report programming*).

The server returns to the method an url to a file with the generated report content.

The method then checks if the *on_open_report* event handler of the report group is defined. If this events handler if defined calls it, otherwise checks the *on_open_report* of the report. If it is defined then calls it.

If none of this events are defined, it (depending on the report *extension* attribute) opens the report in the browser or saves it to disc.

Example

In the following event handler, defined in the client module of the **invoice** report of the Demo application, the value of the report **id** parameter is set:

```
function on_before_print_report(report) {  
    report.id.value = report.task.invoices.id.value;  
}
```

Events

on_before_print_report

on_before_print_report(report)

domain: client

language: javascript

class *Report class*

Description

The *on_before_print_report* event is triggered by the *process_report* method. Use *on_before_print_report* to take specific actions before sending request to the server to generate the report.

The *report* parameter is the report that triggered the event.

See also

Client-side report programming

process_report

on_open_report

on_open_report(report)

domain: client

language: javascript

class *Report class*

Description

The `on_open_report` event is triggered by the *process_report* method.

The `report` parameter is the report that triggered the event.

See also

Client-side report programming

process_report

`on_param_form_close_query`

`on_param_form_close_query(report)`

domain: client

language: javascript

class *Report class*

Description

The `on_param_form_close_query` event is triggered by the *close_param_form* method.

The `report` parameter is the report that triggered the event.

See also

Forms

Client-side report programming

close_param_form

`on_param_form_created`

`on_param_form_created(report)`

domain: client

language: javascript

class *Report class*

Description

The `on_param_form_created` event is triggered by the `create_param_form` method, that, usually, is called by then `print` method.

The `report` parameter is the report that triggered the event.

See also

Forms

Client-side report programming

print

create_param_form

on_param_form_shown

`on_param_form_shown(report)`

domain: client

language: javascript

class *Report class*

Description

The `on_param_form_shown` event is triggered by the `create_param_form` method, that, usually, is called by then `print` method.

The `report` parameter is the report that triggered the event.

See also

Forms

Client-side report programming

print

create_param_form

7.1.8 Field class

class `Field()`

domain: client

language: javascript

Attributes and properties

display_text

display_text

domain: client

language: javascript

class *Field class*

Description

Represents the field's value as a string.

Display_text property is a read-only string representation of a field's value to display in a data-aware control. If an *on_get_field_text* event handler is assigned, **display_text** is the value returned by this event handler. Otherwise, **display_text** is the value of the *lookup_text* property for *lookup fields*, and *text* property, converted according to the *language locale* settings, for other fields.

Display_text is the string representation of the field's value property when it is not being edited. When the field is being edited, the *text* property is used.

Example

```
function on_get_field_text(field) {  
    if (field.field_name === 'customer') {  
        return field.owner.firstname.lookup_text + ' ' + field.lookup_text;  
    }  
}
```

See also

Fields

Lookup fields

on_get_field_text

text

lookup_text

field_caption

field_caption

domain: client

language: javascript

class *Field class*

Description

Field_caption attribute specifies the name of the field that appears to users.

See also

Dataset

Fields

field_name

field_mask

field_mask

domain: client

language: javascript

class *Field class*

Description

You can use **field_mask** attribute to specify the name of the field that appears to

The mask allows a user to more easily enter fixed width input where you would like them to enter the data in a certain format (dates, phone numbers, etc).

A mask is defined by a format made up of mask literals and mask definitions. Any character not in the definitions list below is considered a mask literal. Mask literals will be automatically entered for the user as they type and will not be able to be removed by the user. The following mask definitions are predefined:

- a - Represents an alpha character (A-Z,a-z)
- 9 - Represents a numeric character (0-9)
- *- Represents an alphanumeric character (A-Z,a-z,0-9)

Example

```
function on_edit_form_created(item) {  
    item.phone.field_mask = '999-99-99';  
}
```

field_name

field_name

domain: client

language: javascript

class *Field class*

Description

Specifies the name of the field as referenced in code. Use **field_name** to refer to the field in code.

See also

Dataset

Fields

field_caption

field_size

field_size

domain: client

language: javascript

class *Field class*

Description

Identifies the size of the text field object.

See also

Dataset

Fields

field_type

field_type

domain: client

language: javascript

class *Field class*

Description

Identifies the data type of the field object.

Use the **field_type** attribute to learn the type of the data the field contains. It is one of the following values:

- “text”,
- “integer”,
- “float”,
- “currency”,

- “date”,
- “datetime”,
- “boolean”,
- “blob”

See also

Dataset

Fields

lookup_text

lookup_text

domain: client

language: javascript

class *Field class*

Description

Use **lookup_text** property to get the lookup value of the *lookup field* converted to string.

If the field is *lookup field* gives its lookup text, otherwise gives the value of the *text* property

See also

Fields

Lookup fields

lookup_value

text

lookup_type

lookup_type

domain: client

language: javascript

class *Field class*

Description

For *lookup fields* identifies the type of the *lookup_value*, otherwise returns the value of *field_type* attribute.

See also

Dataset

Fields

lookup_value

lookup_value

domain: client

language: javascript

class *Field class*

Description

Use **lookup_value** property to get the lookup value of the *lookup field*

If the field is *lookup field* gives its lookup value, otherwise gives the value of the *value* property

See also

Fields

Lookup fields

lookup_value

lookup_text

owner

owner

domain: client

language: javascript

class *Field class*

Description

Identifies the item to which a field object belongs.

Check the value of the owner attribute to determine the item that uses the field object to represent one of its fields.

Example

```
function calculate(item) {  
  
}  
  
function on_field_changed(field, lookup_item) {  
    if (field.field_name === 'taxrate') {  
        calculate(field.owner);  
    }  
}
```

See also

Dataset

Fields

raw_value

raw_value

domain: client

language: javascript

class *Field class*

Description

Represents the data in a field object.

Use **raw_value** read only property to read data directly from the item dataset. Other properties such as *value* and *text* use conversion. So the *value* property converts the **null** value to **0** for the numeric fields.

See also

Fields

value

text

Field read_only

read_only

domain: client

language: javascript

class *Field class*

Description

Determines whether the field can be modified in data-aware controls.

Set **read_only** to **true** to prevent a field from being modified in data-aware controls.

See also

Fields

required

required

required

domain: client

language: javascript

class *Field class*

Description

Specifies whether a not empty value for a field is required.

Use **required** to find out if a field requires a value or if the field can be blank. When **required** property is set to true, trying to post a null value will cause an exception to be raised.

See also

Fields

read_only

text

text

domain: client

language: javascript

class *Field class*

Description

Use **text** property to get or set the text value of the field.

Getting text property value

Gets the value of the *value* property and converts it to text.

Setting text property value

Converts the text to the type of the field and assigns its *value* property to this value

See also

Fields

Lookup fields

lookup_value

text

lookup_text

value

value

domain: client

language: javascript

class *Field class*

Description

Use **value** property to get or set the value of the field.

Getting value

When field data is **null**, the field converts it to **0**, if the field_type is “integer”, “float” or “currency”, or to empty string if field_type is “text”.

For *lookup fields* the value of this property is an integer that is the value of the id field of the corresponding record in the lookup item. To get lookup value of the field use the *lookup_value* property.

Setting value

When a new value is assigned, the field checks if the current value is not equal to the new one. If so it

- sets its **new_value** attribute to this value,
- triggers the *on_before_field_changed* event if one is defined for the field,
- changes the field data to the **new_value** attribute and sets it to **null**,
- mark item as modified, so the *is_modified* method will return **true**
- triggers the *on_field_changed* event if one is defined for the field
- updates data-aware controls

Example

```
function calc_total(item) {
    item.amount.value = item.round(item.quantity.value * item.unitprice.value, 2);
    item.tax.value = item.round(item.amount.value * item.owner.taxrate.value / 100,
↪2);
    item.total.value = item.amount.value + item.tax.value;
}
```

See also

Fields

Lookup fields

lookup_value

text

lookup_text

Methods

download

download()

domain: client

language: javascript

class *Field class*

Description

Call download for fields of type FILE to download the file.

Example

```
function on_view_form_created(item) {
    item.add_view_button('Download').click(function() {
        item.attachment.download();
    });
}
```

open

open()

domain: client

language: javascript

class *Field class*

Description

Call `open` for fields of type `FILE` to open the url to the file by using `window.open`.

Example

```
function on_view_form_created(item) {
  item.add_view_button('Open').click(function() {
    item.attachment.open();
  });
}
```

7.1.9 Filter class

class `Filter()`

domain: client

language: javascript

Attributes and properties

`filter_caption`

filter_caption

domain: client

language: javascript

class *Filter class*

Description

Filter_caption attribute specifies the name of the filter that appears to users.

See also

Filters

filter_name

Dataset

filter_name**filter_name****domain:** client**language:** javascript**class** *Filter class***Description**

Specifies the name of the filter as referenced in code. Use **filter_name** to refer to the field in code.

See also*Filters**filter_caption**Dataset***owner****owner****domain:** client**language:** javascript*Filter class***Description**

Identifies the item to which a filter object belongs.

Check the value of the owner attribute to determine the item that uses the filter object to represent one of its filters.

value**value****domain:** client**language:** javascript**class** *Filter class***Description**

Use **value** property to get or set the value of the filter.

Example

```
function on_view_form_created(item) {  
    item.filters.invoicedate1.value = new Date(new Date().setYear(new Date().  
    ↪getFullYear() - 1));  
}
```

See also

Filters

visible

Dataset

visible

visible

domain: client

language: javascript

class *Filter class*

Description

If the value of this property is **true** the input control for this filter will be created by the *create_filter_inputs* method, if the **filters** option is not specified.

See also

Filters

value

Dataset

7.2 Server side (python) class reference

All objects of the framework represent a *task tree*. Below are classes for each kind of task tree objects:

7.2.1 App class

class **App**

domain: server

language: python

App class is used to create a WSGI application

Below the attributes of the class are listed.

admin

admin

domain: server

language: python

class: *App class*

Description

Returns a reference to the Application builder task tree

See also

Workflow

Task tree

task

task

domain: server

language: python

class: *App class*

Description

Returns a reference to the Project task tree

See also

Workflow

Task tree

7.2.2 AbstractItem class

class AbstractItem

domain: server

language: python

AbstractItem class is the ancestor for all item objects of the *task tree*

Below the attributes and methods of the class are listed.

Attributes

environ

environ

domain: server

language: python

class *AbstractItem class*

Description

Specifies the WSGI environment dictionary of the current request from the client.

See also

Server side programming
session

ID

ID

domain: server

language: python

class *AbstractItem class*

Description

The `ID` attribute is the unique in the framework id of the item

The `ID` attribute is most useful when referring to the item by number rather than name. It is also used internally.

See also

Task tree

item_caption

item_caption

domain: server

language: python

class *AbstractItem class*

Description

Specifies the name of the item that appears to users

See also

Task tree

item_name

item_name

domain: server

language: python

class *AbstractItem class*

Description

Specifies the name of the item as referenced in code.

Use `item_name` to refer to the item in code.

See also

Task tree

item_type

item_type

domain: server

language: python

class: *AbstractItem class*

Description

Specifies the type of the item.

Use the `item_type` attribute to get the type of the item. It can have one of the following values

- “task”,
- “items”,
- “details”,
- “reports”,
- “item”,
- “detail_item”,

- “report”,
- “detail”

See also

Task tree

items

items

domain: server

language: python

class *AbstractItem class*

Description

Lists all items owned by the item.

Use `items` to access any of the item owned by this object.

See also

Task tree

owner

Indicates the item that owns this item.

owner

domain: server

language: python

class *AbstractItem class*

Description

Use `owner` to find the owner of an item.

See also

Task tree

session

session

domain: server

language: python

class *AbstractItem class*

Description

Use the `session` property to get access to session object of the current request from the client.

The session is a dictionary that has the following items:

- `ip` - ip address of the user
- `user_info` - dictionary containing information about the user
 - `user_id` - id identifying the user
 - `user_name` - name of the user
 - `role_id` - id of user role
 - `role_name` - name of user role

Example

```
def on_open(item, params):
    user_id = item.session['user_info']['user_id']
    if user_id:
        params['__filters'].append(['user_id', item.task.consts.FILTER_EQ, user_id])

def on_apply(item, delta, params):
    user_id = item.session['user_info']['user_id']
    if user_id:
        for d in delta:
            d.edit()
            d.user_id.value = user_id
            d.post()
```

See also

Server side programming

environ

task

task

domain: server

language: python

class *AbstractItem class*

Description

Indicates the root of the *task tree* that owns this item.

Use `task` attribute to find the root of the *task tree* of which the item is a member.

See also

Task tree

Methods

can_view

can_view (*self*)

domain: server

language: python

class *AbstractItem class*

Description

Use the `can_view` method to determine whether a user of the current session can view records if a data item or print a report.

See also

Roles

session

can_create

can_edit

can_delete

item_by_ID

item_by_ID (*self, ID*)

domain: server

language: python

class *AbstractItem class*

Description

`item_by_ID` searches among all items of the project *task tree*, starting with the current item, for an item whose *ID* attribute is equal to the `ID` parameter.

See also

Task tree

7.2.3 Task class

class Task

domain: server

language: python

Task class is used to create the root of the *Task tree* of the project.

Below the attributes, methods and events of the class are listed.

It, as well, inherits attributes and methods of its ancestor class *AbstractItem class*

Attrubutes

app

app

domain: server

language: python

class: *Task class*

Description

Returns a reference to WSGI *application object*.

The Framework uses *Werkzeug* WSGI Utility Library.

See also

Workflow

work_dir

work_dir

domain: server

language: python

class: *Task class*

Description

Returns the real absolute path to the project directory.

See also

Workflow

Mehods

check_password_hash

check_password_hash (*self*, *pwhash*, *password*)

domain: server

language: python

class *Task class*

Description

Use `check_password_hash` to check a password against a given salted and hashed password value.

The method is wrapper over Werkzeug **check_password_hash** function: <https://werkzeug.palletsprojects.com/en/0.15.x/utils/>

Example

```
def on_login(task, login, password, ip, session_uuid):
    users = task.users.copy(handlers=False)
    users.set_where(login=login)
    users.open()
    for u in users:
        if task.check_password_hash(u.password_hash.value, password):
            return {
                'user_id': users.id.value,
                'user_name': users.name.value,
                'role_id': users.role.value,
                'role_name': users.role.display_text
            }
```

See also

generate_password_hash

connect

connect (*self*)

domain: server

language: python

class *Task class*

Description

Use `connect` to procure a connection from the SQLAlchemy connection pool.

The return value of this method is a DBAPI connection.

A developer must return a connection to the connection pool when it is no longer needed by calling `close` method of the connection.

Example

```
def delete_rec(item, item_id):
    connection = item.task.connect()
    try:
        cursor = connection.cursor()
        cursor.execute('delete from %s where id=%s' % (item.table_name, item_id))
        connection.commit()
    finally:
        connection.close()
```

copy_database

copy_database (*self, dbtype, database=None, user=None, password=None, host=None, port=None, encoding=None, server=None*)

domain: server

language: python

class *Task class*

Description

Use `copy_database` to copy database data when migrating to another database.

See *How to migrate to another database*

Example

in the following code when the project task tree is created the application copies the data from the demo.sqlite database to the project database:

```
from jam.db.db_modules import SQLITE

def on_created(task):
    task.copy_database(SQLITE, '/home/work/demo/demo.sqlite')
```

create_connection

create_connection (*self*)

domain: server

language: python

class *Task class*

Description

Use `create_connection` to create a connection to the project database.

The method returns a new connection.

A developer must close a connection after it is no longer needed.

See also

execute

select

create_connection_ex

create_connection_ex (*self*, *db_module*, *database*, *user=None*, *password=None*, *host=None*,
port=None, *encoding=None*, *server=None*)

domain: server

language: python

class *Task class*

Description

Use `create_connection_ex` to create a connection to other databases.

The method returns a new connection.

A developer must close a connection after it is no longer needed.

See also

How can I use data from other database tables

execute

execute (*self*, *sql*)

domain: server

language: python

class *Task class*

Description

Use `execute` to execute an SQL query (except SELECT queries) using multiprocessing connection pool. For SELECT queries use the *select* method.

The `sql` parameter can be a query string, a list of query strings, a list of lists and so on.

All queries are executed in one transaction and if execution succeeds the `COMMIT` command is called, otherwise `ROLLBACK` command is executed.

Example

```
sql = []
for i in ids:
    sql.append('UPDATE DEMO_CUSTOMERS SET QUANTITY=2 WHERE ID=%s' % i)
item.task.execute(sql)
```

See also

select

generate_password_hash

generate_password_hash (*self*, *password*, *method*='pbkdf2:sha256', *salt_length*=8)

domain: server

language: python

class *Task class*

Description

This method hash a password with the given method and salt with a string of the given length. The format of the string returned includes the method that was used so that *check_password_hash* can check the hash.

The method is wrapper over Werkzeug **generate_password_hash** function: <https://werkzeug.palletsprojects.com/en/0.15.x/utis/>

Example

```
def on_apply(item, delta, params, connection):
    for d in delta:
        if d.password.value:
            d.edit();
            d.password_hash.value = delta.task.generate_password_hash(d.password.
↪value)
            d.password.value = None
            d.post();
```

See also

[check_password_hash](#)

lock

lock (*self*, *lock_name*, *timeout=-1*)

domain: server

language: python

class *Task class*

Description

Use `lock` to implement a platform independent file lock in Python, which provides a simple way of inter-process communication.

This method is a wrapper around Python filelock library: <https://github.com/benediktschmitt/py-filelock>

Once lock has been acquired, subsequent attempts to acquire it block execution, until it is released.

`lock_name` parameter is the name of the lock. It must be unic in the application. The filelock library creates a file in the `locks` folder with this name and `.lock` extension that it uses to implement the lock.

`timeout` parameter - if the lock cannot be acquired within timeout seconds, a `Timeout` exception is raised.

Example

The code

```
def calculate(item):
    lock = item.task.lock('calculation'):
    lock.acquire()
    try:
        #some code
    finally:
        lock.release()
```

is equivalent to

```
def calculate(item):
    with item.task.lock('calculation'):
        #some code
```

The example with timeout:

```
from jam.third_party.filelock import Timeout

def calculate(item):
    try
        with item.task.lock('calculation', timeout=10):
            #some code
    except Timeout:
        print("Another instance of this application currently holds the lock.")
```

In the following example when saving invoice the app calculates sold tracks. Before doing this it acquires a lock:

```
def on_apply(item, delta, params):
    with item.task.lock('invoice_saved'):
        tracks_sql = []
        delta.update_deleted()
        for d in delta:
            for t in d.invoice_table:
                if t.rec_inserted():
                    sql = "UPDATE DEMO_TRACKS SET TRACKS_SOLD = COALESCE(TRACKS_SOLD, ↵
↵0) + ↵
                    %s WHERE ID = %s" % ↵
                    (t.quantity.value, t.track.value)
                elif t.rec_deleted():
                    sql = "UPDATE DEMO_TRACKS SET TRACKS_SOLD = COALESCE(TRACKS_SOLD, ↵
↵0) - ↵
                    (SELECT QUANTITY FROM DEMO_INVOICE_TABLE WHERE ID=%s) WHERE ID = ↵
↵%s" % ↵
                    (t.id.value, t.track.value)
                elif t.rec_modified():
                    sql = "UPDATE DEMO_TRACKS SET TRACKS_SOLD = COALESCE(TRACKS_SOLD, ↵
↵0) - ↵
                    (SELECT QUANTITY FROM DEMO_INVOICE_TABLE WHERE ID=%s) + %s WHERE ↵
↵ID = %s" % ↵
                    (t.id.value, t.quantity.value, t.track.value)
                tracks_sql.append(sql)
        sql = delta.apply_sql()
        return item.task.execute(tracks_sql + [sql])
```

select

select (*self*, *sql*)

domain: server

language: python

class *Task class*

Description

Use `select` to execute `select SELECT` SQL query. To execute the query the connection pool is used.

The `sql` parameter is a query to execute.

The method returns a list of records.

Example

```
recs = item.task.execute_select("SELECT * FROM DEMO_CUSTOMERS WHERE ID=41")
for r in rec:
    print(r)
```

See also

execute

Events

on_created

on_created(task)

domain: server

language: python

class *Task class*

Description

Use on_created to initialize the application on the server side.

The event is triggered when the project *task tree* has just been created. See *Workflow*

The task parameter is a reference to the *task tree*

Note: The execution time of the code in this handler must be very short because of detrimental effects to the end user's experience.

Example

```
def on_created(task):
    # some code
```

See also

Workflow

Task tree

on_ext_request

on_ext_request(task, request, params)

domain: server

language: python

class *Task class*

Description

Use on_ext_request to send a request to the server for processing.

The task parameter is a reference to the *task tree* The request is a string that must starts with '/ext' There could be a list of parameters.

Example

The following application will send every 60 seconds a request to the server of Demo application

```
#!/usr/bin/env python

try:
    # For Python 3.0 and later
    from urllib.request import urlopen
except ImportError:
    # Fall back to Python 2's urllib2
    from urllib2 import urlopen
import json
import time

def send(url, request, params):
    a = urlopen(url + '/' + request, data=str.encode(json.dumps(params)))
    r = json.loads(a.read().decode())
    return r['result']['data']

if __name__ == '__main__':
    url = 'http://127.0.0.1:8080/ext'
    while True:
        result = send(url, 'get_sum', [1, 2, 3])
        print(result)
        time.sleep(60)
```

The server will process this request and return the sum of parameters. The on_ext_request must be declared in task server module:

```
def on_ext_request(task, request, params):
    #print request, params
    reqs = request.split('/')
    if reqs[2] == 'get_sum':
        return params[0] + params[1] + params[2]
```

on_login

on_login(task, form_data, info)

domain: server

language: python

class *Task class*

Description

Use on_login to override default login procedure using Application Builder Users table.

task parameter is a reference to the *task tree*.

form_data is a dictionary containing the values that the user entered in the inputs in the login form. The keys of the dictionary are name attributes of the inputs.

info parameter is a dictionary with the following attributes:

- ip is the ip address of the request
- session_uuid is uuid of the session that will be created.

The event handler must return the dictionary with the following attributes:

- user_id - the unique id of the user
- user_name - user name
- role_id - ID of the role defined in the *Roles*
- role_name - role name

The login form is located in the *index.html* file. You can add your own custom inputs and get their values using form_data parameter

```
<form id="login-form" target="dummy" class="form-horizontal" data-caption="Log in">
  <div class="control-group">
    <label class="control-label" for="input-login">Login</label>
    <div class="controls">
      <input type="text" id="input-login" name="login" tabindex="1" placeholder=
↪ "login">
    </div>
  </div>
  <div class="control-group">
    <label class="control-label" for="input-password">Password</label>
    <div class="controls">
      <input type="password" id="input-password" name="password" tabindex="2"
      placeholder="password" autocomplete="on">
    </div>
  </div>
  <div class="form-footer">
    <input type="submit" class="btn expanded-btn pull-right" id="login-btn" value=
↪ "OK" tabindex="3">
  </div>
</form>
```

Example

In this example user information is stored in the table of the **Users** item in the project database:

```
def on_login(task, form_data, info):
    users = task.users.copy(handlers=False)
    users.set_where(login=form_data['login'])
    users.open()
    if users.rec_count == 1:
        if task.check_password_hash(users.password_hash.value, form_data['password']):
            return {
                'user_id': users.id.value,
                'user_name': users.name.value,
                'role_id': users.role.value,
                'role_name': users.role.display_text
            }
```

See also

session

environ

generate_password_hash

check_password_hash

7.2.4 Group class

class Group

domain: server

language: python

Group class is used to create group objects of the *task tree*

It, as well, inherits attributes and methods of its ancestor class *AbstractItem class*

7.2.5 Item class

class Item

domain: server

language: python

Item class is used to create item objects of the *task tree* that may have an associated database table.

Below the attributes, methods and events of the class are listed.

It, as well, inherits attributes and methods of its ancestor class *AbstractItem class*

Attributes and properties

active

active

domain: server

language: python

class *Item class*

Description

Specifies whether or not an item dataset is open.

Use `active` read only property to determine whether an item dataset is open.

The *open* method changes the value of `active` to `true`. The *close* method sets it to `false`.

When the dataset is open its records can be navigated and its data can be modified and the changes saved in the item database table.

See also

Dataset

Navigating datasets

Modifying datasets

details

details

domain: server

language: python

class *Item class*

Description

Lists all *detail* objects of the item.

See also

Details

fields

fields

domain: server

language: python

class *Item class*

Description

Lists all *field* objects of the item dataset.

Example

```
def customer_fields(customers):
    customers.open(limit=1)
    for f in customers.fields:
        print f.field_caption, f.display_text
```

See also

Fields

Field class

filters

filters

domain: server

language: python

class *Item class*

Description

Lists all *filter* objects of the item dataset.

Example

```
def invoices_filters(invoices):
    for f in invoices.filters:
        print f.filter_name, f.value
```

See also

Filters

Filter class

item_state

item_state

domain: server

language: python

class *Item class*

Description

Examine `item_state` to determine the current operating mode of the item. `Item_state` determines what can be done with data in an item dataset, such as editing existing records or inserting new ones. The `item_state` constantly changes as an application processes data.

Opening a item changes state from inactive to browse. An application can call *edit* to put an item into edit state, or call *insert* or *append* to put an item into insert state.

Posting or canceling edits, insertions, or deletions, changes `item_state` from its current state to browse. Closing a dataset changes its state to inactive.

To check `item_state` value use the following methods:

- *is_new* - indicates whether the item is in insert state
- *is_edited* - indicates whether the item is in edit state
- *is_changing* - indicates whether the item is in edit or insert state

`item_state` value can be:

- 0 - inactive state,
- 1 - browse state,
- 2 - insert state,
- 3 - edit state,
- 4 - delete state

item *task* attribute have consts object that defines following attributes:

- "STATE_INACTIVE": 0,
- "STATE_BROWSE": 1,
- "STATE_INSERT": 2,
- "STATE_EDIT": 3,
- "STATE_DELETE": 4

so if the item is in edit state can be checked the following way:

```
item.item_state == 2
```

or:

```
item.item_state == item.task.consts.STATE_INSERT
```

or:

```
item.is_new()
```

See also

Modifying datasets

log_changes

log_changes

domain: server

language: python

class *Item class*

Description

Indicates whether to log data changes.

Use `log_changes` to control whether or not changes made to the data in an item dataset are recorded. When `log_changes` is `true` (the default), all changes are recorded. They can later be applied to an application server by calling the [apply](#) method. When `log_changes` is `false`, data changes are not recorded and cannot be applied to an application server.

See also

Modifying datasets

apply

rec_no

rec_no

domain: server

language: python

class *Item class*

Description

Examine the `rec_no` property to determine the record number of the current record in the item dataset.

`rec_no` can be set to a specific record number to position the cursor on that record.

See also

Dataset

Navigating datasets

`table_name`

`table_name`

domain: server

language: python

class *Item class*

Description

Read this property to get the name of the corresponding table in the project database.

`virtual_table`

`virtual_table`

domain: server

language: python

class *Item class*

Description

Use the read-only `virtual_table` property to find out if the item has a corresponding table in the project database.

If `virtual_table` is `True` there is no corresponding table in the project database. You can use these items to work with in-memory dataset or use its modules to write code. Calling the *open* method creates an empty data set, and calling the *apply* method does nothing.

Methods

`append`

`append` (*self*)

domain: server

language: python

class *Item class*

Description

Open a new, empty record at the end of the dataset.

After a call to `append`, an application can enable users to enter data in the fields of the record, and can then post those changes to the item dataset using `post` method, and then apply them to the item database table, using `apply` method.

The `append` method

- checks if item dataset is *active* , otherwise raises exception
- if the item is a *detail* , checks if the master item is in edit or insert *state* , otherwise raises exception
- if the item is not a *detail* checks if it is in browse *state* , otherwise raises exception
- open a new, empty record at the end of the dataset
- puts the item into insert *state*

See also

Modifying datasets

apply

apply(self, connection=None, params=None, safe=False):

domain: server

language: python

class *Item class*

Description

Writes all updated, inserted, and deleted records from the item dataset to the database.

The `apply` method

- checks whether the item is a detail, and if it is, returns (the master saves the details changes)
- checks whether the item is in edit or insert *state* , and if so, posts the record
- checks if the change log has changes, and if not, returns
- triggers the `on_before_apply` event handler if one is defined for the item
- if `connection` parameter is `None` the task *connect* method is called to get a connection from task connection pool
- if `on_apply` event handler of the task is defined, executes it
- if *on_apply* event handler is defined for the item, executes it

- generates and executes SQL query to write changes to the database using the connection
- if `connection` parameter was not specified, commits changes to the database and returns connection to the connection pool
- after writing changes to the database, updates the change log and the item dataset - updates primary key values of new records
- triggers the `on_after_apply` event handler if one is defined for the item

Parameters

- `connection` - if this parameter is specified the application uses it to execute sql query that it generates (it doesn't commit changes and doesn't close the connection), otherwise it procures a connection from the task connection pool that will be returned to the pool after changes are committed.
- `params` - use the parameter to pass some user defined options to be used in the `on_apply` event handler. This parameter must be an object of key-value pairs
- `safe` - if set to `True`, the method checks if the user that called the method has a right to create, edit or delete records in the item's database table (if such operation is going to be performed) and, if not, raises an exception. The default value is `False`. See [Roles](#)

Examples

In the second example below, the changes are saved in one transaction.

```
def change_invoice_date(item, item_id):
    inv = item.copy()
    cust = item.task.customers.copy()
    inv.set_where(id=item_id)
    inv.open()
    if inv.record_count():
        now = datetime.datetime.now()
        cust.set_where(id=inv.customer.value)
        cust.open()

        inv.edit()
        inv.invoice_datetime.value = now
        inv.post()
        inv.apply()

        cust.edit()
        cust.last_action_date.value = now
        cust.post()
        cust.apply()
```

```
def change_invoice_date(item, item_id):
    con = item.task.connect()
    try:
        inv = item.copy()
        cust = item.task.customers.copy()
        inv.set_where(id=item_id)
        inv.open()
        if inv.record_count():
            now = datetime.datetime.now()
```

(continues on next page)

(continued from previous page)

```

        cust.set_where(id=inv.customer.value)
        cust.open()

        inv.edit()
        inv.invoice_datetime.value = now
        inv.post()
        inv.apply(con)

        cust.edit()
        cust.last_action_date.value = now
        cust.post()
        cust.apply(con)
    finally:
        con.commit()
        con.close()

```

See also*Modifying datasets***bof****bof** (*self*)**domain:** server**language:** python**class** *Item class***Description**

Test `bof` (beginning of file) method to determine if the cursor is positioned at the first record in an item dataset.

If `bof` returns true, the cursor is unequivocally on the first row in the dataset. `bof` returns true when an application

- Opens an item dataset.
- Calls an item's *first* method.
- Call an item's *prior* method, and the method fails (because the cursor is already on the first row in the dataset).

`bof` returns false in all other cases.

Note: If both *eof* and `bof` return true, the item dataset is empty.

See also*Dataset**Navigating datasets*

can_create

can_create (*self*)

domain: server

language: python

class *AbstractItem class*

Description

Use the `can_create` method to determine whether a user of the current session have a right to create a new record.

Example

```
def send_email(item, selected, subject, mess):  
    if not item.can_create():  
        raise Exception('You are not allowed to send emails.')  
    #code sending email
```

See also

Roles

session

can_view

can_create

can_edit

can_delete

can_delete

can_delete (*self*)

domain: server

language: python

class *AbstractItem class*

Description

Use the `can_delete` method to determine whether a user of the current session have a right to delete a record.

See also

Roles

session

can_view

can_create

can_edit

can_edit

can_edit (*self*)

domain: server

language: python

class *AbstractItem class*

Description

Use the `can_edit` method to determine whether a user of the current session have a right to edit a record.

See also

Roles

session

can_view

can_create

can_delete

cancel

cancel (*self*)

domain: server

language: python

class *Item class*

Description

Call `cancel` to undo modifications made to one or more fields belonging to the current record, as long as those changes are not already posted to the item dataset.

Cancel

- triggers the `on_before_cancel` event handler if one is defined for the item.

- to undo modifications made to the current record and its details if the record has been edited or removes the new record if one was appended or inserted.
- puts the item into browse *state*
- triggers the `on_after_cancel` event handler if one is defined for the item.

See also

Modifying datasets

`clear_filters`

`clear_filters` (*self*)

domain: server

language: python

class *Item class*

Description

Use `clear_filters` to set filter values of the item to `None`.

See also

Filtering records

Filters

`close`

`close` (*self*)

domain: server

language: python

class *Item class*

Description

Call `close` to close an item dataset. After dataset is closed the *active* property is `false`.

See also

Dataset

open

copy

copy (*self*, *filters=True*, *details=True*, *handlers=True*)

domain: server

language: python

class *Item class*

Description

Use copy to create a copy of an item. The created copy is not added to the *task tree* and will be destroyed by Python garbage collector when no longer needed.

All attributes of the copy object are defined as they were at the time of creating of the task tree. See *Workflow*

The method can have the following parameters:

- **handlers** - if the value of this parameter is `true`, all the functions and events defined in the server module of the item will also be available in the copy. The default value is `true`.
- **filters** - if the value of this parameter is `true`, the filters will be created for the copy, otherwise there will be no filters. The default value is `true`.
- **details** - if the value of this parameter is `true`, the details will be created for the copy, otherwise there will be no details. The default value is `true`.

Example

```
def on_generate(report):
    cust = report.task.customers.copy()
    cust.open()

    report.print_band('title')

    for c in cust:
        firstname = c.firstname.display_text
        lastname = c.lastname.display_text
        company = c.company.display_text
        country = c.country.display_text
        address = c.address.display_text
        phone = c.phone.display_text
        email = c.email.display_text
        report.print_band('detail', locals())
```

See also

Task tree

Workflow

delete

delete (*self*)

domain: server

language: python

class *Item class*

Description

Deletes the active record and positions the cursor on the next record.

The `delete` method

- checks if item dataset is *active*, otherwise raises exception
- checks if item dataset is not empty, otherwise raises exception
- if item is a *detail*, checks if the master item is in edit or insert *state*, otherwise raises exception.
- if item is not a *detail*, checks if it is in browse *state*, otherwise raises exception.
- puts the item into delete *state*
- deletes the active record and positions the cursor on the next record
- puts the item into browse *state*

See also

Modifying datasets

edit

edit (*self*)

domain: server

language: python

class *Item class*

Description

Enables editing of data in the dataset.

After a call to `edit`, an application can enable users to change data in the fields of the record, and can then post those changes to the item dataset using *post* method, and then apply them to database using *apply* method.

The `edit` method

- checks if item dataset is active, otherwise raises exception
- checks if item dataset is not empty, otherwise raises exception
- checks whether the item dataset is already in edit state, and if so, returns
- if item is a *detail*, checks if the master item is in edit or insert *state*, otherwise raises exception
- if item is not a *detail*, checks if it is in browse *state*, otherwise raises exception
- puts the item into edit *state*, enabling the application or user to modify fields in the record

See also

Modifying datasets

eof

eof (*self*)

domain: server

language: python

class *Item class*

Description

Test `eof` (end-of-file) to determine if the cursor is positioned at the last record in an item dataset. If `eof` returns true, the cursor is unequivocally on the last row in the dataset. `eof` returns true when an application:

- Opens an empty dataset.
- Calls an item's *last* method.
- Call an item's *next* method, and the method fails (because the cursor is already on the last row in the dataset).

`eof` returns false in all other cases.

Note: If both `eof` and *bof* return true, the item dataset is empty.

See also

Dataset

Navigating datasets

field_by_name

field_by_name (*self, field_name*)

domain: server

language: python

class *Item class*

Description

Call `field_by_name` to retrieve field information for a field when only its name is known.

The `field_name` parameter is the name of an existing field.

`field_by_name` returns the field object for the specified field. If the specified field does not exist, `field_by_name` returns `None`.

filter_by_name

filter_by_name (*self*, *filter_name*)

domain: server

language: python

class *Item class*

Description

Call `filter_by_name` to retrieve filter information for a filter when only its name is known.

The `filter_name` parameter is the name of an existing filter.

`filter_by_name` returns the filter object for the specified filter. If the specified filter does not exist, `filter_by_name` returns `None`.

first

first (*self*)

domain: server

language: python

class *Item class*

Description

Call `first` to position the cursor on the first record in the item dataset and make it the active record. `First` posts any changes to the active record.

See also

Dataset

Navigating datasets

insert

insert (*self*)

domain: server

language: python

class *Item class*

Description

Inserts a new, empty record in the item dataset.

After a call to `insert`, an application can enable users to enter data in the fields of the record, and can then post those changes to the item dataset using `post` method, and then apply them to the item database table, using `apply` method.

The `insert` method

- checks if item dataset is *active* , otherwise raises exception
- if the item is a *detail* , checks if the master item is in edit or insert *state* , otherwise raises exception
- if the item is not a *detail* checks if it is in browse *state* , otherwise raises exception
- inserts a new, empty record in the item dataset.
- puts the item into insert *state*

See also

Modifying datasets

is_changing

is_changing (*self*)

domain: server

language: python

class *Item class*

Description

Checks if an item is in edit or insert state and returns true if it is.

An application calls *edit* to put an item into edit state and *append* or *insert* to put an item into insert state.

See also

Modifying datasets

is_edited

is_edited (*self*)

domain: server

language: python

class *Item class*

Description

Checks if an item is in edit state and returns true if it is.

An application calls *edit* to put an item into edit state.

See also

Modifying datasets

is_modified

is_modified (*self*)

domain: server

language: python

class *Item class*

Description

Checks if the current record of an item dataset has been modified during edit or insert opertaions. The method returns `false` after the *post* method is executed.

See also

Modifying datasets

is_new

is_new (*self*)

domain: server

language: python

class *Item class*

Description

Checks if an item is in insert state and returns true if it is.

An application calls *append* or *insert* methods to put an item into insert state.

See also

Modifying datasets

last

last (*self*)

domain: server

language: python

class *Item class*

Description

Call `last` to position the cursor on the last record in the item dataset and make it the active record.

See also

Dataset

Navigating datasets

locate

locate (*self, fields, values*)

domain: server

language: python

class *Item class*

Description

Implements a method for searching an item dataset for a specified record and makes that record the active record.

Arguments:

- `fields`: a field name, or list of field names
- `values`: a field value or list of field values

This method locates the record where the fields specified by `fields` parameter have the values specified by `values` parameter.

`locate` returns true if a record is found that matches the specified criteria and the cursor repositioned to that record.

If a matching record was not found and the cursor is not repositioned, this method returns false.

See also

Dataset

Navigating datasets

next

next (*self*)

domain: server

language: python

class *Item class*

Description

Call `next` to position the cursor on the next record in the item dataset and make it the active record. Next posts any changes to the active record.

open

open(*self*, *options=None*, *expanded=None*, *fields=None*, *where=None*, *order_by=None*, *open_empty=False*, *params=None*, *offset=None*, *limit=None*, *funcs=None*, *group_by=None*, *safe=False*)

domain: server

language: python

class *Item class*

Description

Call `open` to generate and execute a SELECT SQL query to the item database table for obtaining a dataset.

The method initializes the item *fields*, formulates parameters of a request, and triggers the `on_before_open` event handler if one is defined for the item.

If there is a *on_open* event handler defined for the item, `open` executes this event handler and assigns a dataset to the result, returned by it, otherwise generates a SELECT SQL query, based on parameters of the request, executes this query and assigns the result of the execution to the dataset

After that it sets *active* to true, the *item_state* to browse mode, goes to the first record of the dataset, triggers `on_after_open`, if it is defined for the item.

Parameters

You can pass *options* dictionary to specify parameters of the request in the same form as for the *open* method on the client:

```
invoices.open({
    'fields': ['customer', 'invoicedate', 'total'],
    'where': {customer: customer_id, invoicedate__ge: date1, invoicedate__le: date2},
    'order_by': ['invoicedate']
})
```

or pass the keyworded arguments:

```

invoices.open(
    fields=['customer', 'invoicedate', 'total'],
    where={customer: customer_id, invoicedate__ge: date1, invoicedate__le: date2},
    order_by=['invoicedate']
)

```

- **expanded** - if the value of this parameter is true, the SELECT query will have JOIN clauses to get lookup values of the *lookup fields*, otherwise there will be no lookup values. The default value is `true`.
- **fields** - use this parameter to specify the WHERE clause of the SELECT query. This parameter is a list of field names. If it is omitted, the fields defined by the *set_fields* method will be used. If the *set_fields* method was not called before the *open* method execution, all available fields will be used.
- **where** - use this parameter to specify how records will be filtered in the SQL query. This parameter is a dictionary, whose keys are field names, that are followed, after double underscore, by a filtering symbols (see *Filtering records*). If this parameter is omitted, values defined by the *set_where* method will be used. If the *set_where* method was not called before the *open* method execution, and *where* parameter is omitted, then the values of *filters* defined for the item will be used to filter records.
- **order_by** - use *order_by* to specify sort order of the records. This parameter is a list of field names. If there is a sign '-' before the field name, then on this field records will be sorted in decreasing order. If this parameter is omitted, a list defined by the *set_order_by* method will be used.
- **offset** - use *offset* to specify the offset of the first row to get.
- **limit** - use *limit* to limit the output of a SQL query to the first so-many rows.
- **funcs** - this parameter can be a dictionary, whose keys are a field names and values are function names that will be applied to the fields in the SELECT Query
- **group_by** - use *group_by* to specify fields to group the result of the query by. This parameter must be a list of field names.
- **open_empty** - if this parameter is set to `true`, the application does not send a request to the server but just initializes an empty dataset. The default value is `false`.
- **params** - use the parameter to pass some user defined options to be used in the *on_open* event handler. This parameter must be an object of key-value pairs
- **safe** - if set to `True` the method checks if the user that called the method has a right to view the item's data and, if not, raises an exception. The default value is `False`. See *Roles*

Examples

In this example the parameters of the request are a dictionary:

```

import datetime

def get_sales(item):
    date1 = datetime.datetime.now() - datetime.timedelta(days=3*365)
    date2 = datetime.datetime.now()
    invoices = item.task.invoices.copy()

    invoices.open({
        'fields': ['customer', 'date', 'total'],
        'where': {'date__ge': date1, 'date__le': date2},
        'order_by': ['customer', 'date']
    })

```

Below the parameters are passed as a keyworded list:

```
import datetime

def get_sales(item):
    date1 = datetime.datetime.now() - datetime.timedelta(days=3*365)
    date2 = datetime.datetime.now()
    invoices = item.task.invoices.copy()

    invoices.open(
        fields=['customer', 'date', 'total'],
        where={'date__ge': date1, 'date__le': date2},
        order_by=['customer', 'date']
    )
```

The same result can be achieved by using `set_fields`, `set_where`, `set_order_by` methods:

```
import datetime

def get_sales(item):
    date1 = datetime.datetime.now() - datetime.timedelta(days=3*365)
    date2 = datetime.datetime.now()
    invoices = item.task.invoices.copy()

    invoices.set_fields('customer', 'date', 'total')
    invoices.set_where(date__ge=date1, date__le=date2);
    invoices.set_order_by('customer', 'date');
    invoices.open();
```

```
import datetime
```

```
def get_sales(item): date1 = datetime.datetime.now() - datetime.timedelta(days=3*365) date2 = date-
time.datetime.now() invoices = item.task.invoices.copy()

    invoices.set_fields(['customer', 'date', 'total']) invoices.set_where({'date__ge': date1, 'date__le': date2}); in-
voices.set_order_by(['customer', 'date']); invoices.open();
```

```
def get_sales(task) {
    sales = task.invoices.copy()

    sales.open(fields=['customer', 'id', 'total'],
        funcs={'id': 'count', 'total': 'sum'},
        group_by=['customer'],
        order_by=['customer'])
```

See also

Dataset

Filtering records

set_fields

set_order_by

set_where

post

post (*self*)

domain: server

language: python

class *Item class*

Description

Writes a modified record to the item dataset. Call `post` to save changes made to a record after *append*, *insert* or *edit* method was called.

The `post` method

- checks if an item is in edit or insert *state* , otherwise raises exception
- triggers the `on_before_post` event handler if one is defined for the item
- checks if a record is valid, if not raises exception
- If an item has *details* , post current record in details
- add changes to an item change log
- puts the item into browse *state*
- triggers the `on_after_post` event handler if one is defined for the item.

See also

Modifying datasets

prior

prior (*self*)

domain: server

language: python

class *Item class*

Description

Call `prior` to position the cursor on the previous record in the item dataset and make it the active record. last posts any changes to the active record.

See also

Dataset

Navigating datasets

record_count

record_count ()

domain: server

language: python

class *Item class*

Description

Call `record_count` to get the total number of records owned by the item's dataset.

Example

```
item.open()
if item.record_count():
    # some code
```

See also

Dataset

open

set_fields

set_fields (*self*, *lst=None*, **fields*)

domain: server

language: python

class *Item class*

Description

Use the `set_fields` method to define and store internally the `fields` parameter that will be used by the *open* method, when its own `fields` parameter is not specified. The *open* method clears internally stored parameter value.

The `fields` is arbitrary argument list of field names.

Parameters

You can specify the `fields` as a list, the way the *set_fields* method on the client does or as non-keyworded arguments.

Example

The result of the execution of following code snippets will be the same:

```
item.open(fields=['id', 'invoicedate'])
```

```
item.set_fields('id', 'invoicedate')
item.open()
```

```
item.set_fields(['id', 'invoicedate'])
item.open()
```

See also

Dataset

open

set_order_by

set_order_by (*self*, *lst=None*, **fields*)

domain: server

language: python

class *Item class*

Description

Use the `set_order_by` method to define and store internally the `order_by` parameter that will be used by the *open* method, when its own `order_by` parameter is not specified. The *open* method clears internally stored parameter value.

Parameters

You can specify the fields as a list, the way the *set_order_by* method on the client does or as non-keyworded arguments. If there is a sign '-' before a field name, then on this field records will be sorted in decreasing order.

Example

The result of the execution of following code snippets will be the same:

```
item.open(order_by=['customer', '-invoicedate'])
```

```
item.set_order_by('customer', '-invoicedate')
item.open();
```

```
item.set_order_by(['customer', '-invoicedate'])
item.open();
```

See also

Dataset

open

set_where

set_where (*self*, *dic=None*, ***fields*)

domain: server

language: python

class *Item class*

Description

Use the `set_where` method to define and store internally the where filters that will be used by the *open* method, when its own where parameter is not specified. The *open* method clears internally stored parameter value.

Parameters

You can specify the filters as a dictionary, the way the *set_where* method on the client does or as keyworded arguments

Example

The result of the execution of following code snippets will be the same:

```
import datetime

date = datetime.datetime.now() - datetime.timedelta(days=3*365)
item.open(where={'customer': 44, 'invoicedate__gt': date})
```

```
import datetime

date = datetime.datetime.now() - datetime.timedelta(days=3*365)
item.set_where({'customer': 44, 'invoicedate__gt': date})
item.open()
```

```
import datetime

date = datetime.datetime.now() - datetime.timedelta(days=3*365)
item.set_where(customer=44, invoicedate__gt=date)
item.open()
```


See also

Dataset

open

Events

on_apply

`on_apply(self, delta, params, connection)`

domain: server

language: python

class *Item class*

Description

Write `on_apply` event handler when you need to override the standard data saving procedure during the execution of the `apply` method on the *client* or *server*.

See *on_apply events* to understand how `on_apply` events are triggered.

The `on_apply` event handler has the following parameters:

- `item` - a reference to the item,
- `delta` - a delta containing item change log (discussed in more detail below),
- `params` - the parameters passed to the server by `apply` method,
- `connection` - the connection that will be used to save changes to the database.

The `delta` parameter contains changes that must be saved in the database. By itself, this option is an item's copy, and its dataset is the item's change log. The nature of the record change can be obtained by using following methods:

- `rec_inserted`
- `rec_modified`
- `rec_deleted`

each of which returns a value of `True`, if the record is added, modified or deleted, respectively.

If the item has a detail items, `delta` also has a corresponding detail items, storing detail changes.

Note: Please note that when a record is deleted from an item and this record has detail records, the change log will just keep this deleted record, information about the deleted records of the details is not stored. To add this deleted detail records, call `delta's update_deleted` method.

You do not need to open `delta` detail after the cursor has been moved to another record.

Delta dataset fields have an `old_value` attribute that can be used to get the value of a field before changes have been made.

Fields of the delta dataset have an `old_value` attribute that can be used to get the value of a field before changes have been made.

when the `on_apply` event handler is not defined the `apply_delta` method is executed, that generates SQL queries and executes them. After that it returns the information about the result of processing, that stores the id's of the new records as well. The client based on this information updates the item's change log and values of the primary fields of new records.

When `on_apply` event handler returns `None` the `apply_delta` is executed.

You can make some additional processing of the delta. In the following code, the a value of the date field is set to the current date before changes are applied to the database table.

```
import datetime

def on_apply(item, delta, params, connection):
    for d in delta:
        d.edit()
        d.date.value = datetime.datetime.now()
        d.post()
```

Note: Please note that changes made this way are not reflected in the item dataset on the client. You can use the item client methods *refresh_record* or *refresh_page* to display these changes.

In the following code, while saving the changes made to the invoices, the application as well updates the value of the `tracks_sold` field for tracks in this invoices. All this is done in one transaction.

```
def on_apply(item, delta, params, connection):
    tracks = item.task.tracks.copy()
    changes = {}
    delta.update_deleted()
    for d in delta:
        for t in d.invoice_table:
            if not changes.get(t.track.value):
                changes[t.track.value] = 0
            if t.rec_inserted():
                changes[t.track.value] += t.quantity.value
            elif t.rec_deleted():
                changes[t.track.value] -= t.quantity.value
            elif t.rec_modified():
                changes[t.track.value] += t.quantity.value - t.quantity.old_value
    ids = list(changes.keys())
    tracks.set_where(id__in=ids)
    tracks.open()
    for t in tracks:
        q = changes.get(t.id.value)
        if q:
            t.edit()
            t.tracks_sold.value += q
            t.post()
    tracks.apply(connection)
```

In the previous examples the `on_apply` event handler returns `None` so after that the `apply_delta` method is executed by the application.

The more general case is:

```
def on_apply(item, delta, params, connection):

    # execute some code before changes are written to the database
```

(continues on next page)

(continued from previous page)

```

result = item.apply_delta(delta, params, connection)

# execute some code after changes are written to the database

return result

```

See also*Server side programming**on_apply events**Modifying datasets***on_open**`on_open(item, params)`**domain:** server**language:** python**class** *Item class***Description**

Write `on_open` event handler when you need to override the standard procedure of fetching the records from the dataset during the execution of the open method on the *client* or *server*.

See *on_open_events* to understand how `on_open` events are triggered.

The `on_open` event handler has the following parameters:

- `item` - reference to the item,
- `params` - dictionary containing parameters passed to the server by the open method:
 - `__expanded` - corresponds to the `expanded` parameter of the server *open* method / `expanded` attribute of options parameter of the client *open* method
 - `__fields` - list of field names
 - `__filters` - list of items, each of which is a list with the following members:
 - * field name
 - * filter constant from *Filtering records*
 - * filter value
 - `__funcs` - functions dictionary
 - `__order` - list of items, each of which is a list with the following members:
 - * field name
 - * boolean value, if it is true the order is descending
 - `__offset` - corresponds to the `offset` parameter of the open method

- `__limit` - corresponds to the limit parameter of the open method
- `__client_request` - is true when request came from the client

params can also include user defined parameters passed to the open method.

Below is an example of params that the client open methods of invoices items sends to the server:

```
{
  '__fields': [u'id', u'deleted', u'customer', u'firstname', u'date',
    u'subtotal', u'taxrate', u'tax', u'total',
    u'billing_address', u'billing_city', u'billing_country',
    u'billing_postal_code', u'billing_state'],
  '__filters': [[u'customer', 7, [6]]],
  '__expanded': True,
  '__limit': 11,
  '__offset': 0,
  '__order': [[u'date', True]]
}

{
  '__fields': [u'id'],
  '__funcs': {u'id': u'count'},
  '__filters': [],
  '__expanded': False,
  '__offset': 0,
  '__order': [],
  '__summary': True
}
```

The server application generates an SQL query, based on params and executes them.

The server returns to the client the resulting records and the error message if it occurs during the execution.

Here is an example how server events can be used

See also

on_open_events

Server side programming

Dataset

7.2.6 Detail class

class Detail

domain: server

language: python

Detail class inherits attributes, methods and events of *Item class*

Attrubutes

master

master

domain: server

language: python

class *Detail class*

Description

Use `master` attribute to get reference to the master of the detail.

See also

Details

7.2.7 Reports class

class Reports

domain: server

language: python

Reports class is used to create the group object of the *task tree* that owns the reports of a project.

Below the events of the class are listed.

It, as well, inherits attributes and methods of its ancestor class *AbstractItem class*

Events

on_convert_report

on_convert_report (*report*)

domain: client

language: python

class: *Reports class*

Description

The framework converts reports internally, using LibreOffice. Use the **on_convert_report** event if you want to use some other service or change some parameters of report conversion.

The **report** parameter is the report that triggered the event.

Example

```
import os
from subprocess import Popen, STDOUT, PIPE

def on_convert_report(report):
    try:
        if os.name == "nt":
            import _winreg
            regpath = "SOFTWARE\\Microsoft\\Windows\\CurrentVersion\\App_
↳ Paths\\soffice.exe"
            root = _winreg.OpenKey(_winreg.HKEY_LOCAL_MACHINE, regpath)
            s_office = _winreg.QueryValue(root, "")
        else:
            s_office = "soffice"
            conversion = Popen([s_office, '--headless', '--convert-to', report.ext,
report.report_filename, '--outdir', os.path.join(report.task.work_dir,
↳ 'static', 'reports') ],
                stderr=STDOUT, stdout=PIPE)
            out, err = conversion.communicate()
            converted = True
    except Exception as e:
        print(e)
```

7.2.8 Report class

class Report

domain: server

language: python

Report class inherits

Below the attributes, methods and events of the class are listed.

It, as well, inherits attributes and methods of its ancestor class *AbstractItem class*

Attributes

report_filename

report_filename

domain: client

language: python

class *Report class*

Description

When the *template* attribute of the report is specified, the *generate* method saves the content of the generated report to a file in a report folder of the static directory and set the value of the **report_filename** attribute to the name of the saved file.

Its value can be used in the *on_after_generate* event handler.

See also

Server-side report programming

generate

report_url

report_url

domain: client

language: python

class *Report class*

Description

The *generate* method sends to the client the value that is stored in this attribute as url of the generated file.

When the *template* attribute of the report is specified, this value is set by the *generate* method after it save generated content. Otherwise a developer must set it himself.

See also

Server-side report programming

generate

template

template

domain: client

language: python

class *Report class*

Description

The **report_filename** attribute stores a file name of the report template. Usually it is set in the Application builder when the report is created. But it can be changed dynamically on the server in the *on_before_generate* event handler or be empty, if it's necessary to create, for exapmle, some **txt** file.

See also

Report templates

Creating a report

Server-side report programming

Methods

generate

generate (*self*)

domain: client

language: python

class *Report class*

Description

The method is used internally to generate the content of the report.

When a server gets a request from a client to generate report, it first of all creates a copy of the report and then this copy calls the method.

This method triggers the *on_before_generate* event.

If the report *template* is defined, parses it and triggers *on_parsed* and *on_generate* events.

In the *on_generate* event handler developer should write a code that generates the content of the report and saves it in the **ods** file, by using the *print_band* method to print bands .

When the report is generated and the value of report *extension* attribute, set on the client, is not equal **ods**, the server tries to convert the **ods** file.

To convert the file, it first checks if the report group (owner of the report) has *on_convert_report* event handler. If this handler is defined it uses it to convert the report. Otherwise it uses for conversion the **LibreOffice** installed on the server in **headless** mode.

After that the application saves generated report to a file in a report folder of the static directory, set the value of the *report_filename* attribute to the name of the saved file, generates the value of the *report_url* attribute and triggers *on_after_generate* event.

Once the report is generated it is saved in a **report** folder of the **static** directory and the server sends the client the report file url.

If the report *template* attribute is not set, the server triggers the *on_generate* and after that *on_after_generate* events. In this case you should save the generated content to a file yourself and set a value of the *report_url* attribute.

See also

Programming reports

Server-side programming

hide_columns

hide_columns (*self*, *col_list*)

domain: client

language: python

class *Report class*

Description

Use **hide_columns** method to hide some columns defined in the report *template*.

The **col_list** parameter specifies which columns should be hidden. This is a list of integers or letters, defining the position of the report columns.

Use this method in the *on_parsed* event handler.

Example

```
def on_parsed(report):  
    report.hide_columns(['A', 'C'])  
#     report.hide_columns([1, 3])
```

See also

Programming reports

Report templates

Server-side report programming

on_parsed

on_generate

print_band

print_band (*self*, *band*, *dic=None*)

domain: client

language: python

class *Report class*

Description

Use **print_band** method to set values of programmable cells of the band defined in the report *template* and add the band to the content of the report.

It has the following parameters:

- **band** - specifies the name of the band to be printed.

- **dic** - dictionary, containing values than will be assigned to programmable cells of the band.

Example

The following code generates content of the **Customer list** report of the Demo application:

```
def on_generate(report):
    cust = report.task.customers.copy()
    cust.open()

    report.print_band('title')

    for c in cust:
        firstname = c.firstname.display_text
        lastname = c.lastname.display_text
        company = c.company.display_text
        country = c.country.display_text
        address = c.address.display_text
        phone = c.phone.display_text
        email = c.email.display_text
        report.print_band('detail', locals())
```

See also

Programming reports

Report templates

Server-side report programming

generate

on_generate

Events

on_after_generate

on_after_generate (*report*)

domain: client

language: python

class *Report class*

Description

The **on_after_generate** event is triggered by the *generate* method, when the report has been generated and saved to a file with the name that is stored in the *report_filename* attribute.

The **report** parameter is the report that triggered the event.

See also

Programming reports

generate

on_before_generate

on_before_generate (*report*)

domain: client

language: python

class *Report class*

Description

The **on_before_generate** event is triggered by the *generate* method, before generating the report.

The **report** parameter is the report that triggered the event.

See also

Programming reports

generate

on_generate

on_generate (*report*)

domain: client

language: python

class *Report class*

Description

The **on_generate** event is triggered by the *generate* method.

Write the **on_generate** event handler to generate the content of the report. Use the *print_band* method to print bands, defined in the report *template*.

The **report** parameter is the report that triggered the event.

See also

Programming reports

Server-side report programming

Report templates

generate

on_parsed

on_parsed (*report*)

domain: client

language: python

class *Report class*

Description

The **on_parsed** event is triggered by the *generate* method, after the report *template* have been parsed.

Use this event handler you hide some columns in the report template by calling *hide_columns*

The **report** parameter is the report that triggered the event.

See also

Programming reports

Server-side report programming

Report templates

hide_columns

7.2.9 Field class

class **Field**

domain: server

language: python

Attrubutes and properties

display_text

display_text

domain: server

language: python

Field class

Description

Represents the field's value as a string.

`Display_text` property is a read-only string representation of a field's value to display it to users. If an `on_get_field_text` event handler is assigned, **display_text** is the value returned by this event handler. Otherwise, `display_text` is the value of the `lookup_text` property for *lookup fields* and `text` property converted according to the *language locale* settings for other fields.

`Display_text` is the string representation of the field's value property when it is not being edited. When the field is being edited, the `text` property is used.

Example

```
def on_generate(report):
    cust = report.task.customers.copy()
    cust.open()

    report.print_band('title')

    for c in cust:
        firstname = c.firstname.display_text
        lastname = c.lastname.display_text
        company = c.company.display_text
        country = c.country.display_text
        address = c.address.display_text
        phone = c.phone.display_text
        email = c.email.display_text
        report.print_band('detail', locals())
```

See also

Fields

Lookup fields

on_get_field_text

text

lookup_text

field_caption

field_caption

domain: server

language: python

Field class

Description

`Field_caption` attribute specifies the name of the field that appears to users.

See also

Dataset

Fields

field_name

field_name

field_name

domain: server

language: python

Field class

Description

Specifies the name of the field as referenced in code. Use `field_name` to refer to the field in code.

See also

Dataset

Fields

field_caption

field_size

field_size

domain: server

language: python

Field class

Description

Identifies the size of the text field object.

See also

Dataset

Fields

field_type

field_type

domain: server

language: python

Field class

Description

Identifies the data type of the field object.

Use the `field_type` attribute to learn the type of the data the field contains. It is one of the following values:

- “text”,
- “integer”,
- “float”,
- “currency”,
- “date”,
- “datetime”,
- “boolean”,
- “blob”

See also

Dataset

Fields

lookup_text

lookup_text

domain: server

language: python

Field class

Description

Use `lookup_text` property to get the lookup value of the *lookup field* converted to string.

If the field is *lookup field* gives its lookup text, otherwise gives the value of the *text* property

See also

Fields

Lookup fields

lookup_value

text

lookup_value

lookup_value

domain: server

language: python

Field class

Description

Use `lookup_value` property to get the lookup value of the *lookup field*

If the field is *lookup field* gives its lookup value, otherwise gives the value of the *value* property

See also

Fields

Lookup fields

lookup_value

lookup_text

owner

owner

domain: server

language: python

Field class

Description

Identifies the item to which a field object belongs.

Check the value of the owner attribute to determine the item that uses the field object to represent one of its fields.

See also

Dataset

Fields

raw_value

raw_value

domain: server

language: python

Field class

Description

Represents the data in a field object.

Use `raw_value` read only property to read data directly from the item dataset. Other properties such as *value* and *text* use conversion. So the *value* property converts the `null` value to 0 for the numeric fields.

See also

Fields

value

text

read_only

read_only

domain: server

language: python

Field class

Description

Determines whether the field can be modified in data-aware controls.

Set `read_only` to `true` to prevent a field from being modified in data-aware controls.

See also

Fields

required

required

required

domain: server

language: python

Field class

Description

Specifies whether a not empty value for a field is required.

Use `required` to find out if a field requires a value or if the field can be blank. When `required` property is set to `true`, trying to post a null value will cause an exception to be raised.

See also

Fields

read_only

text

text

domain: server

language: python

Field class

Description

Use `text` property to get or set the text value of the field.

Getting text property value

Gets the value of the *value* property and converts it to text.

Setting text property value

Converts the text to the type of the field and assigns its *value* property to this value

See also

Fields

Lookup fields

lookup_value

text

lookup_text

value

value

domain: server

language: python

Field class

Description

Use `value` property to get or set the value of the field.

Getting value

When field data is `null`, the field converts it to `0`, if the `field_type` is “integer”, “float” or “currency”, or to empty string if `field_type` is “text”.

For *lookup fields* the value of this property is an integer that is the value of the `id` field of the corresponding record in the lookup item. To get lookup value of the field use the *lookup_value* property.

Setting value

When a new value is assigned, the field checks if the current value is not equal to the new one. If so it

- sets its `new_value` attribute to this value,
- triggers the *on_before_field_changed* event if one is defined for the field,
- changes the field data to the `new_value` attribute and sets it to `null`,
- mark item as modified, so the *is_modified* method will return `true`
- triggers the *on_field_changed* event if one is defined for the field
- updates data-aware controls

See also

Fields

Lookup fields

lookup_value

text

lookup_text

7.2.10 Filter class

class **Filter**

domain: server

language: python

Attributes and properties

filter_name

filter_name

domain: server

language: python

class *Filter class*

Description

Specifies the name of the filter as referenced in code. Use `filter_name` to refer to the field in code.

See also

Filters

Dataset

owner

owner

domain: server

language: python

class *Filter class*

Description

Identifies the item to which a filter object belongs.

Check the value of the owner attribute to determine the item that uses the filter object to represent one of its filters.

value**value****domain:** server**language:** python**class** *Filter class***Description**

Use `value` property to get or set the value of the filter.

Example

```
function on_view_form_created(item) {  
    item.filters.invoicedate1.value = new Date(new Date().setYear(new Date().  
↪getFullYear() - 1));  
}
```

See also*Filters**Dataset*

8.1 Version 1

Version 1 was designed to develop database desktop applications based on the GTK Toolkit.

8.2 Version 2

In version 2, support for developing database applications with a web interface was added.

8.3 Version 3

In version 3, support for development of database desktop applications based on the GTK Toolkit was removed.

8.4 Version 4

In Version 4 the server side was reworked. Web.py library was replaced with werkzeug. Session support was added.

8.4.1 Version 4.0.70

Jam.py library:

- Bug, related to last column disappearance when table content is larger than its container, fixed.
- Bug, related to table footer, fixed.
- *lookup_type* property of *Field class* added

- Exception is now raised when developer forgets to add a value attribute to a field, when specifying a value for programmable cell.
- Bug, related to date and datetime fields in when clause for SQLite database, fixed.
- Administrator now shows project version / jam.py version information.

Demo application:

- Filter text bug in demo application fixed.
- Selection of search field for catalogs in demo application is added.
- Multiple record selection in invoices journal in demo application is added (used when delete button is clicked)
- Menu in demo application is changed.
- blue-theme.css file added

Twitter account created: https://twitter.com/jampy_framework

Jam.py Users Mailing List created: <https://groups.google.com/forum/#!forum/jam-py>

8.4.2 Version 4.0.71

Jam library:

- Tables are now responsive
- Fixed header columns and table columns mismatch
- Several themes added

Demo application:

- Dashboard added
- Theme selections added
- Resize function from Task client module removed

8.4.3 Version 4.0.74

Jam.py:

- Bug with *open* method, when *order_by* parameter is an empty list fixed
- Bug with sql generated when default order is set and fields parameter do not contain any of field of default order in the *open* method fixed
- The exception handling of errors occurring in the code, when inplace editor is used reworked so developer can find the reason of an error

Administrator:

- Tabs are now created for opened modules.

Demo application:

- Search for catalogs reworked.

8.4.4 Version 4.0.78

Jam.py:

- Import functionality for SQLITE databases is available now
- Creation of foreign field indexes for SQLITE databases are removed
- Popovers for fields with help attribute reworked

Admin:

- Bugs related to tabs are fixed

Demo:

- For Customers item email sending functionality is added. It demonstrates the use of *server* method to execute script defined in the server module from client module, how to use fields of item with *virtual_table* to create a form for input of data.

8.4.5 Version 4.0.79

Jam.py:

- Tables columns resizing reworked
- Mysql - bug with datetime fields in where clause fixed
- *on_ext_request* event published

Demo application:

- For Customers lookup modal view form Send and Print buttons are hidden now

Documentation:

- Faq - new topics added

8.4.6 Version 4.0.81

Jam.py:

- Displaying of wells in modeless forms is corrected
- Async parameter for the client apply method added
- Bug of clone method when expanded attribute is false fixed
- Bug related to lookup_value, lookup_text and display_text properties of fields and params with lookup_lists fixed
- Bug, when users were able to print reports when 'Can view' was disabled for their role, fixed
- Open, set_where, set_fields, set_order_by methods on the server can have the same parameters as corresponding methods on the client
- Edit_record, apply_changes, cancel_edit methods on the client modified so that user can open documents for viewing when can_edit method returns false
- When converting reports the soffice is passed norestore parameters
- Starting '/' signs are removed from css and js links in index.html file
-

Administrator:

- Validation of field names is corrected

Demo application:

- Select button added to the Invoices edit form to add selected tracks to the invoice.
- Visible items whose `set_view` method returns false are not added to the dynamic menu

8.4.7 Version 4.0.84

Library:

- Python 3 is now supported
- The work is started to support multiple languages
- Reports on server now have ext attribute

Demo:

- Example of using *on_convert_report* event of reports group on the server is created

8.4.8 Version 4.0.88

Library:

- Html templates reworked
- You can change 12px default font to 14px default font by replace `jam.css` to `jam14.css` in `index.html`

Demo:

- New examples of using html templates

8.5 Version 5

8.5.1 Version 5.0.1

Library:

- Default font is 14px now you can change it to 12px font by replacing

```
<link href="jam/css/jam.css" rel="stylesheet">
```

with

```
<link href="jam/css/jam12.css" rel="stylesheet">
```

in `index.html`

- Administrator is renamed to Application builder you can run it by typing `127.0.0.1:8080/builder.html`, `127.0.0.1:8080/admin.html` is also supported
- Asterisk is added to required fields now

To cancel it add

```
.control-label.required:after {
    content: "";
}
```

to project.css file

- Selection of lookup list value in report parameters for fixed.

Documentation:

- First version of Documentation completed
- New topics added:

refresh_record

refresh_page

search

- Jam.py roadmap added

Demo:

- Small font menu item is added to Themes menu

8.5.2 Version 5.1.1

Library:

- History of changes made by users can now be stored. See *Saving the history of changes made by users*
- Local filtering of dataset records is reworked and published. See *Filtered, on_filter_record*
- *clone* method is published
- Application is now throws an exception when an attempt is made to get or set a value to a field when the dataset is empty.

Application Builder:

- **Delete reports after** attribute is added to *Project parameters*
- Some changes to interface are made.

8.5.3 Version 5.2.1

Library:

- DBtable class declared jam.js reworked. Paginator div is removed from table and doesn't scroll when table is scrolled. For tables with pagination y scrolling is removed. You can pass *create_table* method two new options:
 - *summary_fields* - a list of field names. When it is specified and item paginate attribute is true, the table calculates sums for numeric fields and displays them in the table footer, for not numeric fields it displays the number of records.
 - *freeze_count* - an integer value. If it is greater than 0, it specifies number of first columns that become frozen - they will not scroll when the table is scrolled.
- Bug when inserting a new record and pressing Escape key doesn't cancel operation fixed
- Bug when history doesn't save user name fixed

Demo project:

- Code that calculated summary for table in invoices client module removed
- Code of `on_view_form_created` event handler in the task client module of demo application and new project is changed so after deleting a record the *refresh_page* method is called

8.5.4 Version 5.3.1

Library:

- A set of client methods of the task for working with tabs developed
 - *init_tabs*
 - *add_tab*
 - *close_tab*
- Forms are reworked. Each form now have a div with modal-header class declared in the index.html file. The elements for search input and filter text are removed from the form templates and placed in the form header.
- The *view*, *append_record*, *insert_record* and *edit_record* methods are reworked. If a container parameter is passed to these methods and the *init_tabs* method is called for the container, the tabs are created that contains the forms.

For existing projects add the line

```
task.init_tabs($("#content"));
```

at the beginning on the `on_page_loaded` event handler of the task client module to forms be displayed in tabs and add a `$("#content")` container parameter to `append_record`, `insert_record` and `edit_record` methods.

You can add a line

```
task.add_form_borders = false;
```

if you don't want to change html templates of the forms. Otherwise remove elements for search input and filter text (in the div with form-header class, remove it) from the form templates and add the div with modal-header class to templates.

Demo:

- Demo was rewritten to display forms in tabs and modeless edit forms

Documentation:

- *on_ext_request* example corrected for Python 3

8.5.5 Version 5.3.3

Library:

- Safe mode bug (after version 5.3.1) fixed
- Postgres import bug fixed
- Task attribute `edit_form_container` is defined in the `on_page_loaded` event handler of the task client module of a new project and demo application

```
task.edit_form_container = $("#content"); // comment this line to have modal edit_
↪ forms
```

8.5.6 Version 5.4.1

Application builder

- The language attribute is added to the Project parameters to select the language used in the Project and allows to add or edit the language.
- Interface tab added to Project parameters dialog
- Buttons “View” and “Edit” renamed to “View form” and “Edit form”
- The “View form” dialog lets now setup, besides fields used to create tables, table options such as columns to set sorting order and summary fields. Use “Form” tab to setup from options including detail that will be displayed in the view form
- The “Edit form” dialog allows to create tabs and bands to display field inputs in the edit forms. You can setup details that will be displayed and edited in the edit form in the “Form” tab

Library

- First stage of internalization completed. Developers can add their languages
- Processing of form events worked over. See `_process_event` method in `jam.js`
- To avoid concurrency problems and memory leaks the task tree on the server side is immutable now, except when `on_created` event is executed. You must use copy method when you need to call open method or want to change attributes of items in the event handlers or functions on the server
- The `create_detail_views` method added that allows to edit details in the edit forms
- Item class: `table_options` attribute added (contains table options setup in the `AppBuilder`)
- `DBAbstractInput` class: coping, pasting, Escape key processing worked over
- `DBTable` class: hints worked over
- Themes were corrected
- A lot of minor changes

Demo application

- Themes removed. You can set theme in the Project parameters Interface tab.
- Dynamic menu worker over

Note: If you created your project with a version of the library less than 4.3.1 add the following line in the `on_page_loaded` event handler in the task’s client module:

```
task.old_forms = true;
```

For libraries with versions 4.3, clear the code of client modules of catalogs and journals and replace client module of the task with the corresponding code of the Demo application or the new project. Make an archive of the project before doing it.

8.5.7 Version 5.4.11

Library:

- Metadata import/export and copy_database method of the server task reworked for compatibility with different databases, when a project moved to a database of different type
- *on_detail_changed* event and *calc_summary* method added
- *alert*, alert_info and alert_success methods added
- python 3 bugs of MYSQL, Postgres, Oracle database support fixed
- some bugs fixed related to SQL queries generation
- on_login event bugs fixed
- *field_mask* attribute for fields on client added
- date inputs use masks now
- create_menu method of the client task added.
- As much code as possible are moved from default code (and demo project) to the library
- Bugs related to non-ascii characters in the project path fixed

Application builder:

- keyboard shortcuts bugs fixed
- roles bugs fixed
- rights can be set for details
- mask attribute added to Fields Dialog
- Summary fields attribute added in the View Form Dialog for details
- Default search field, Detail height attributes added in the View Form Dialog
- Detail height attribute added in the Edit Form Dialog
- some minor bugs fixed

Note: To use masks in existing projects the following line must be added to index.html after package update:

```
<script src="jam/js/jquery.maskedinput.js"></script>
```

before

```
<script src="jam/js/jam.js"></script>
```

8.5.8 Version 5.4.14

Library:

- add_button method added

- *select_records* method added
- *alert* method bugs fixed
- bootstrap theme buttons changed
- Metadata import bug fixed - didn't display error that was raised when changes to DB were committed

8.5.9 Version 5.4.15

Library:

- Support for MS SQL SERVER added
- Jam.py supports deletion and changing of fields, and foreign indexes for SQLITE database now. As SQLITE doesn't support column changing and deletion and addition of foreign indexes to existing tables, Jam.py creates a new table and copies records into it from old one.
- for SQLITE database Jam.py doesn't support import of metadata to an existing project (project items of which have corresponding tables in database) now. You can import of metadata to a new project
- BLOB field type renamed to LONGTEXT and corresponding DB field changed from Blob (if it was) to Long text type wherever possible

Application Builder:

- History item creation bug fixed
- Foreign indexes creation bug fixed

8.5.10 Version 5.4.21

Library:

- SQLITE - case insensitive search implemented
- MSSQL bugs fixed
- Search reworked
- Field Dialogs - you can specify default values for DATE, DATETIME, BOOLEAN fields and for lookup fields that are based on lookup lists. These default values are assigned to fields when append or insert methods of element are called on the client or server. These default values are not applied when you are changing table record using direct SQL query.
- *select_records* method reworked
- *add_view_button*, *add_edit_button* methods added
- When user tries to close or reload page and there is an item that is being edited and its data has been modified the application warns user about it.
- A lot of miscellaneous bug fixed
- FAQ, Application Builder, Into chapter in the documentation reworked

8.5.11 Version 5.4.22

Library:

- *upload* method reworked

- Image and file field types added - *Tutorial. Part 2. File and image fields*
- **Buttons on top** attribute added to the Form tab of the *View Form Dialog*
- *refresh_record* method reworked, it can refresh details of the item
- *on_field_get_html* event added

Demo application:

- Invoices: *on_field_get_html* handler added
- Customer: image field “Photo” added
- Tracks: file field “File” added

8.5.12 Version 5.4.23

Library:

- *refresh_record* bugs fixed
- Image and file fields can be lookup fields now

Application builder:

- creating new group bug fixed

8.5.13 Version 5.4.24

Library:

- Language support reworked
- Images of image fields of Application builder items are stored in static/builder folder now to be able to export/import them to/from metadata file
- MSSQL ALTER TABLE bug fixed

8.5.14 Version 5.4.27

Library:

- Capturing image from camera options is now available. See **Capturing image from camera** in the *Field Editor Dialog*
- Bug in Chrome 7 with report parameters order is fixed.
- **Buttons on top** attribute added to the Form tab of the *Edit Form Dialog* Works for new projects, for existing project copy the div with class ‘default-top-edit’ from a new project index.html to your index.html
- *read_only* reworked
- *on_login* event params changed, previous params supported with warning in the logs
- There can be multiple details in view form
- Details order can be changed now
- Esprima-python library is used now for parsing javascript on the server
- German translation added

- Various minor bugs fixed
- Readme file changed

Demo application:

- Tracks catalog view form displays sold tracks.

8.5.15 Version 5.4.29

Library:

- Jam.py uses JQuery 3 now
- *lock* method added
- *create_connection_ex* method added
- *edit_record* method reworked, the edit form events are triggered after all data are get from the server
- connection of connection pull is recreated after one heure of inactivity
- minor bugs fixed

Documetation:

- *How can I perform calculations in the background*
- *How can I use data from other database tables*

8.5.16 Version 5.4.30

- Bug when creating a new project on some systems, related to encoding, fixed
- Greek language added
- For fields of longtext type when value is null value property returns empty string now.
- *select_records* method reworked. *all_record* parameter added. If the *all_records* parameter is set to true, all selected records are added, otherwise the method omits existing records (they were selected earlier).
- *view_form_created* and *edit_form_created* methods added to the Task class (reserved for future use)
- Code that used to create tables and detail tables in *on_view_form_created* event handler of task moved to *create_view_tables* method of Item class in jam.js module
- *table_container_class* and *detail_container_class* attributes added to items *view_options* to enable developer to change in *on_view_form_created* event handler of item
- *inputs_container_class* and *detail_container_class* attributes added to items *edit_options* to enable developer to change in *on__form_created* event handler of item
- In jam.css and jam12.css fixed the btn groups left magrin in form-header and form-footer class
- Some minor bugs fixed

8.5.17 Version 5.4.31

- Bug when reading index.html file fixed. Index.html must have a unicode encoding.
- German translation updated.
- Bug when Dashbord are opened fixed in Demo.

- Users item added to demo.
- on_login event handler in task server module that uses Users item to login is written (commented) and changing of password implemented. Uncomment on_login to see how it works. Description is here <https://groups.google.com/forum/#!topic/jam-py/Obkv5d3yT8A>

8.5.18 Version 5.4.36

Library:

- tables reworked, they now support virtual scrolling.
- some bugs fixed

Application Builder:

- Search added for items.

Demo application:

- User registration implemented

8.5.19 Version 5.4.37

Library:

- Jam.py can now be deployed on [PythonAnywhere](#). See *How to deploy project on PythonAnywhere*
- Directory of the project can be passed to the create_application function now (jam/wsgi.py module).
- Multiprocessing connection pool parameter removed from project *Parameters*
- Bugs related to processing of keyboard events by forms fixed
- Some bugs fixed

Documentation:

- *How to* section created. That section will contain code examples that can be useful to quickly accomplish common tasks.

8.5.20 Version 5.4.40

Library:

- Jam.py now uses SQLAlchemy connection pool
- when image field read_only attribute is set user can not change the image by double-clicking on it
- Some bugs fixed

Documentation:

deployment section added to *How to*

How to lock a record so that users cannot edit it at the same time topic added

8.5.21 Version 5.4.53

Library:

- *on_login* event changed
- *generate_password_hash* and *check_password_hash* methods added
- bugs related to moving to SQLAlchemy and tables with virtual scroll are fixed.
- tables resizing bug for numeric fields fixed
- tables with frozen cols bugs fixed
- details bug when renaming copy fixed
- minor bugs fixed

Documentation:

- latest docs changes
- how to section bug fixed
- ‘How to lock a record so that users cannot edit it at the same time’ topic removed - other algorithm will be used

8.5.22 Version 5.4.54

Library:

- MSSQL bug when selecting tables for import fixed
- delta old_value property code modified (not documented yet)

Documentation:

- *Authentication* section added to *How to*

8.5.23 Version 5.4.56

Library:

- *Record locking* is available
- task creation in wsgi.py modified to avoid ‘project have not been created yet’ message
- report parameters display_text bug fixed
- show_hints and hint_fields attributes can be added to the *table_options* or options parameter of the *create_table* method.
- *refresh_record* method restore positions of detail records

Documentation:

- *Form events* rewritten
- Some topics from *Jam.py FAQ* are moved to *How to*

Demo application

- on_apply event handler in Invoices server module modified

8.5.24 Version 5.4.57

Library:

- *Record locking* bug, when PostgreSQL, MSSQL or Firebird database is used, fixed

To use record locking for items for which you defined *on_apply* event handler you must change. Add the connection parameter, create a cursor and use the cursor to execute sql queries. Otherwise the record locking won't work.

For example, the code

```
def on_apply(item, delta, params):
    tracks_sql = []
    delta.update_deleted()
    for d in delta:
        for t in d.invoice_table:
            if t.rec_inserted():
                sql = "UPDATE DEMO_TRACKS SET TRACKS_SOLD = COALESCE(TRACKS_SOLD, 0) \
                    %s WHERE ID = %s" % \
                    (t.quantity.value, t.track.value)
            elif t.rec_deleted():
                sql = "UPDATE DEMO_TRACKS SET TRACKS_SOLD = COALESCE(TRACKS_SOLD, 0) - \
                    (SELECT QUANTITY FROM DEMO_INVOICE_TABLE WHERE ID=%s) WHERE ID = %s" \
                    (t.id.value, t.track.value)
            elif t.rec_modified():
                sql = "UPDATE DEMO_TRACKS SET TRACKS_SOLD = COALESCE(TRACKS_SOLD, 0) - \
                    (SELECT QUANTITY FROM DEMO_INVOICE_TABLE WHERE ID=%s) + %s WHERE ID = \
                    %s" % \
                    (t.id.value, t.quantity.value, t.track.value)
            tracks_sql.append(sql)
    sql = delta.apply_sql()
    return item.task.execute(tracks_sql + [sql])
```

must be changed to

```
def on_apply(item, delta, params, connection):
    with item.task.lock('invoice_saved'):
        cursor = connection.cursor()
        delta.update_deleted()
        for d in delta:
            for t in d.invoice_table:
                if t.rec_inserted():
                    sql = "UPDATE DEMO_TRACKS SET TRACKS_SOLD = COALESCE(TRACKS_SOLD, \
                        0) + \
                        %s WHERE ID = %s" % \
                        (t.quantity.value, t.track.value)
                elif t.rec_deleted():
                    sql = "UPDATE DEMO_TRACKS SET TRACKS_SOLD = COALESCE(TRACKS_SOLD, \
                        0) - \
                        (SELECT QUANTITY FROM DEMO_INVOICE_TABLE WHERE ID=%s) WHERE ID = \
                        %s" % \
                        (t.id.value, t.track.value)
                elif t.rec_modified():
                    sql = "UPDATE DEMO_TRACKS SET TRACKS_SOLD = COALESCE(TRACKS_SOLD, \
                        0) - \
```

(continues on next page)

(continued from previous page)

```

↪ID = %s" % \
        (SELECT QUANTITY FROM DEMO_INVOICE_TABLE WHERE ID=%s) + %s WHERE _
        (t.id.value, t.quantity.value, t.track.value)
        cursor.execute(sql)

```

8.5.25 Version 5.4.60

Library:

- Synchronization of parameters and reloading of the task tree when metadata changes for web applications running on parallel processes reworked.
- Import of metadata reworked. See [Export/import metadata](#)
- Created the ability to import metadata from the migration folder when the server is restarted. See [How to migrate development to production](#)
- Migration to another database is available now. See [How to migrate to another database](#)
- virtual_table is now a read-only property on the client [virtual_table](#) and server [virtual_table](#). For an item which virtual_table property is true, calling the open method creates an empty data set, and calling the apply method does nothing.
- When importing a table the virtual_table attribute id read only now.
- title_line_count option is added to the [table_options](#) specifies the number of lines of text displayed in a title row, if it is 0, the height of the row is determined by the contents of the title cells It can be set in Application Builder.

8.5.26 Version 5.4.69

Library:

- Werkzeug library upgraded to the version 0.15.4
- common.py module rewritten, consts object created
- adm_server.py module removed
- admin folder is created with modules
 - admin.py - application builder server side module
 - task.py - loading of task from admin.sqlite database
 - export_metadata.py
 - import_metadata.py
- builder folder added to package that contains Application Builder project that is used to create Jam.py Application Builder, see read.me file in the folder.
- task loading accelerated
- import of metadata rewritten
- import of metadata accelerated
- [permissions](#) property added
- logging created (currently under development and not documented yet)
- edit method on the client and server checks now if item state is in edit mode and if it is does nothing

- round methods are corrected on the client and server, value of currency fields are rounded before they are assigned
- inline editing is now available for any items (not details only)
- inline editing of lookup fields, list fields, date and datetime inputs reworked, bugs fixed
- fixed columns of tables bugs fixed
- tables striped option added
- search input is focused now by Ctrl-F, Escape returns focus to the table
- *enable_controls* redraws controls now, no need to call *update_controls* method
- lot of bugs fixed

Application builder:

- a link to the form-related documentation page has been added to the application Builder form headers

Documetation:

- search bug fixed
- topics related to the server *on_apply* and *on_open* events rewritten
- new topic added *How to prohibit changing record*

8.5.27 Version 5.4.109

- The work on sanitizing of field values is completed. See *Sanitizing*
- The **TextArea** attribute is added the **Interface Tab** of the *Edit Form Dialog* for TEXT fields
- The **Do not sanitize** attribute is added the **Interface Tab** of the *Edit Form Dialog*. See *Sanitizing*
- The **Accept** attribute of the **Interface Tab** of the *Edit Form Dialog* for FILE fields is required now. Uploaded files are checked on the server against this attribute.
- The **Upload file extensions** attribute is added to the *Project parameters* that defines file types that are allowed to be uploaded by the task *upload* method.
- The expanded options is added to the *add_edit_button* and *add_view_button* methods.

8.6 Jam.py roadmap

We plan to add the following features to Jam.py:

- Support for actions that can be represented as buttons panels, navbars, pop-up menus and simplifying support for keyboard events, internalization and mobile devices.
- Language support stage 2: internalization, support for multiple languages in the project.
- Support for Bootstrap 4.
- Support for mobile devices.
- Development of report wizards, simplifying report creation
- Rework of import/export utility: visual interface, control over merging of changes.
- DBTree component revision and creation of documentation

A

`abort()` (built-in function), 220
`AbstractItem` (built-in class), 363
`AbstractItem()` (class), 217
`active`, 380
`active` (None attribute), 255
`add_edit_button()` (built-in function), 272
`add_tab()` (built-in function), 233
`add_view_button()` (built-in function), 273
`admin`, 363
`alert()` (built-in function), 220
`app`, 369
`App` (built-in class), 362
`append()`, 384
`append()` (built-in function), 274
`append_record()` (built-in function), 274
`apply()` (built-in function), 275
`apply_record()` (built-in function), 277
`assign_filters()` (built-in function), 277

B

`bof()`, 387
`bof()` (built-in function), 278

C

`calc_summary()` (built-in function), 278
`can_create()`, 388
`can_create()` (built-in function), 279
`can_delete()`, 388
`can_delete()` (built-in function), 280
`can_edit()`, 389
`can_edit()` (built-in function), 280
`can_view()`, 368
`can_view()` (built-in function), 221
`cancel()`, 389
`cancel()` (built-in function), 281
`cancel_edit()` (built-in function), 282
`check_password_hash()`, 370
`clear_filters()`, 390

`clear_filters()` (built-in function), 282
`clone()` (built-in function), 283
`close()` (built-in function), 283, 390
`close_edit_form()` (built-in function), 284
`close_filter_form()` (built-in function), 284
`close_param_form()` (built-in function), 344
`close_tab()` (built-in function), 234
`close_view_form()` (built-in function), 285
`connect()`, 370
`copy()`, 391
`copy()` (built-in function), 286
`copy_database()`, 371
`create_connection()`, 372
`create_connection_ex()`, 372
`create_detail_views()` (built-in function), 287
`create_edit_form()` (built-in function), 288
`create_filter_form()` (built-in function), 289
`create_filter_inputs()` (built-in function), 290
`create_inputs()` (built-in function), 291
`create_menu:` `function()` (built-in function), 234
`create_param_form()` (built-in function), 345
`create_param_inputs()` (built-in function), 345
`create_table()` (built-in function), 292
`create_view_form()` (built-in function), 292

D

`delete()`, 391
`delete()` (built-in function), 293
`delete_record()` (built-in function), 294
`Detail` (built-in class), 408
`Detail()` (class), 338
`details`, 380
`details` (None attribute), 255
`disable_controls()` (built-in function), 294
`disable_edit_form()` (built-in function), 295
`display_text`, 416
`display_text` (None attribute), 351
`download()` (built-in function), 359

E

`each()` (*built-in function*), 296
`each_detail()` (*built-in function*), 297
`each_field()` (*built-in function*), 297
`each_filter()` (*built-in function*), 298
`each_item()` (*built-in function*), 222
`edit()`, 392
`edit()` (*built-in function*), 299
`edit_form` (*None attribute*), 256
`edit_options` (*None attribute*), 257
`edit_record()` (*built-in function*), 299
`enable_controls()` (*built-in function*), 300
`enable_edit_form()` (*built-in function*), 301
`environ`, 364
`eof()`, 393
`eof()` (*built-in function*), 301
`execute()`, 372
`extension` (*None attribute*), 341

F

`Field` (*built-in class*), 416
`Field()` (*class*), 350
`field_by_name()`, 393
`field_by_name()` (*built-in function*), 302
`field_caption`, 417
`field_caption` (*None attribute*), 351
`field_mask` (*None attribute*), 352
`field_name`, 418
`field_name` (*None attribute*), 352
`field_size`, 418
`field_size` (*None attribute*), 353
`field_type`, 419
`field_type` (*None attribute*), 353
`fields`, 381
`fields` (*None attribute*), 258
`Filter` (*built-in class*), 424
`Filter()` (*class*), 360
`filter_by_name()`, 394
`filter_by_name()` (*built-in function*), 302
`filter_caption` (*None attribute*), 360
`filter_form` (*None attribute*), 258
`filter_name`, 424
`filter_name` (*None attribute*), 361
`filter_options` (*None attribute*), 259
`filtered` (*None attribute*), 260
`filters`, 381
`filters` (*None attribute*), 260
`first()`, 394
`first()` (*built-in function*), 303
`forms_container` (*None attribute*), 230
`forms_in_tabs` (*None attribute*), 230

G

`generate()`, 412

`generate_password_hash()`, 373
`Group` (*built-in class*), 379
`Group()` (*class*), 248

H

`hide_columns()`, 413
`hide_message()` (*built-in function*), 222

I

`ID`, 364
`ID` (*None attribute*), 217
`init_tabs()` (*built-in function*), 236
`insert()`, 394
`insert()` (*built-in function*), 303
`insert_record()` (*built-in function*), 304
`is_changing()` (*built-in function*), 304, 395
`is_edited()` (*built-in function*), 305, 395
`is_modified()` (*built-in function*), 305, 396
`is_new()` (*built-in function*), 306, 396
`Item` (*built-in class*), 379
`Item()` (*class*), 254
`item_by_ID()`, 368
`item_by_ID()` (*built-in function*), 222
`item_caption`, 364
`item_caption` (*None attribute*), 218
`item_name`, 365
`item_name` (*None attribute*), 218
`item_state`, 382
`item_state` (*None attribute*), 261
`item_type`, 365
`item_type` (*None attribute*), 218
`items`, 366
`items` (*None attribute*), 219

L

`last()`, 397
`last()` (*built-in function*), 306
`load()` (*built-in function*), 236
`load_module()` (*built-in function*), 223
`load_modules()` (*built-in function*), 223
`load_script()` (*built-in function*), 224
`locate()`, 397
`locate()` (*built-in function*), 306
`lock()`, 374
`log_changes`, 383
`log_changes` (*None attribute*), 262
`login()` (*built-in function*), 237
`logout()` (*built-in function*), 238
`lookup_field` (*None attribute*), 262
`lookup_text`, 419
`lookup_text` (*None attribute*), 354
`lookup_type` (*None attribute*), 354
`lookup_value`, 420
`lookup_value` (*None attribute*), 355

M

master, 409
 master (*None attribute*), 338
 message () (*built-in function*), 225

N

next (), 398
 next () (*built-in function*), 307

O

on_after_generate () (*built-in function*), 414
 on_before_generate () (*built-in function*), 415
 on_convert_report () (*built-in function*), 409
 on_generate () (*built-in function*), 415
 on_parsed () (*built-in function*), 416
 open () (*built-in function*), 307, 359
 owner, 366, 420
 owner (*None attribute*), 219, 355, 361, 424

P

paginate (*None attribute*), 263
 param_form (*None attribute*), 342
 param_options (*None attribute*), 343
 permissions (*None attribute*), 263
 post (), 401
 post () (*built-in function*), 309
 print () (*built-in function*), 347
 print_band (), 413
 prior (), 401
 prior () (*built-in function*), 310
 process_report () (*built-in function*), 347

Q

question () (*built-in function*), 227

R

raw_value, 421
 raw_value (*None attribute*), 356
 read_only, 421
 read_only (*None attribute*), 264, 356
 rec_count (*None attribute*), 265
 rec_no, 383
 rec_no (*None attribute*), 265
 record_count (), 402
 record_count () (*built-in function*), 311
 refresh_page () (*built-in function*), 311
 refresh_record () (*built-in function*), 312
 Report (*built-in class*), 410
 Report () (*class*), 341
 report_filename, 410
 report_url, 411
 Reports (*built-in class*), 409
 Reports () (*class*), 339

required, 422
 required (*None attribute*), 357

S

safe_mode (*None attribute*), 231
 search () (*built-in function*), 312
 select (), 375
 select_records () (*built-in function*), 313
 selections (*None attribute*), 266
 server () (*built-in function*), 227
 session, 367
 set_fields (), 402
 set_fields () (*built-in function*), 313
 set_forms_container () (*built-in function*), 239
 set_order_by (), 403
 set_order_by () (*built-in function*), 314
 set_where (), 404
 set_where () (*built-in function*), 315
 show_history () (*built-in function*), 315

T

table_name, 384
 table_options (*None attribute*), 267
 task, 363, 367
 Task (*built-in class*), 369
 task (*None attribute*), 219
 Task () (*class*), 229
 template, 411
 templates (*None attribute*), 231
 text, 422
 text (*None attribute*), 357

U

update_controls () (*built-in function*), 316
 upload () (*built-in function*), 239
 user_info (*None attribute*), 232

V

value, 423, 425
 value (*None attribute*), 358, 361
 view () (*built-in function*), 316
 view_form (*None attribute*), 270
 view_options (*None attribute*), 271
 virtual_table, 384
 virtual_table (*None attribute*), 272
 visible (*None attribute*), 362

W

warning () (*built-in function*), 228
 work_dir, 369

Y

yes_no_cancel () (*built-in function*), 229