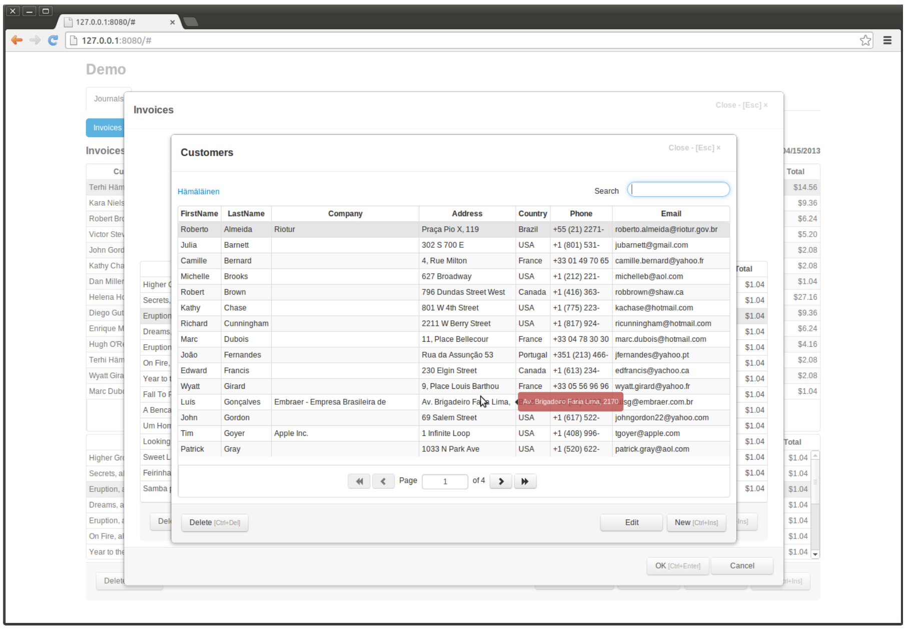
Jam.py

User Guide.

Table of contents

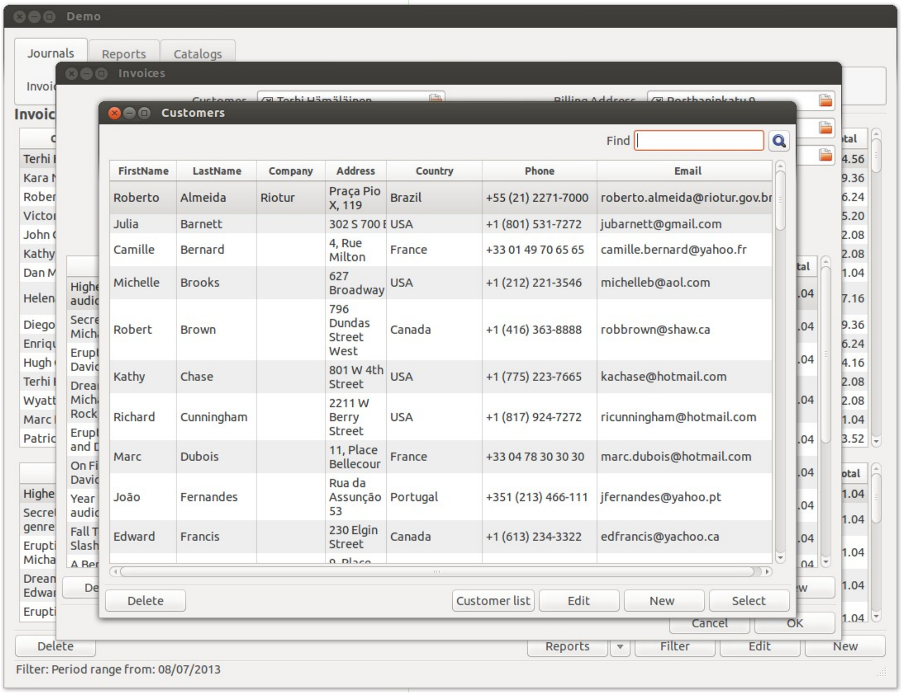
1 Overview.

Jam.py is an event-driven framework for developing client-server database applications. You can use jam.py to create web based applications. Server side is implemented in Python and uses Web.py library, the client side in JavaScript and uses JQuery and Bootstrap:



FirstName	LastName	Company	Address	Country	Phone	Email
Roberto	Almeida	Riotur	Praça Pio X, 119	Brazil	+55 (21) 2271-	roberto.almeida@riotur.gov.br
Julia	Barnett		302 S 700 E	USA	+1 (801) 531-	jubarnett@gmail.com
Camille	Bernard		4, Rue Milton	France	+33 01 49 70 65	camille.bernard@yahoo.fr
Michelle	Brooks		627 Broadway	USA	+1 (212) 221-	michelleb@aol.com
Robert	Brown		796 Dundas Street West	Canada	+1 (416) 363-	robbrown@shaw.ca
Kathy	Chase		801 W 4th Street	USA	+1 (775) 223-	kachase@hotmail.com
Richard	Cunningham		2211 W Berry Street	USA	+1 (817) 924-	ricunningham@hotmail.com
Marc	Dubois		11, Place Bellecour	France	+33 04 78 30 30	marc.dubois@hotmail.com
João	Fernandes		Rua da Assunção 53	Portugal	+351 (213) 466-	jfernandes@yahoo.pt
Edward	Francis		230 Elgin Street	Canada	+1 (613) 234-	edfrancis@yachoo.ca
Wyatt	Girard		9, Place Louis Barthou	France	+33 05 56 96 96	wyatt.girard@yahoo.fr
Luís	Gonçalves	Embraer - Empresa Brasileira de	Av. Brigadeiro Faria Lima,			sg@embraer.com.br
John	Gordon		69 Salem Street	USA	+1 (617) 522-	johngordon22@yahoo.com
Tim	Goyer	Apple Inc.	1 Infinite Loop	USA	+1 (408) 996-	tgoyer@apple.com
Patrick	Gray		1033 N Park Ave	USA	+1 (520) 622-	patrick.gray@aol.com

as well as local desktop applications in python and pygtk and desktop client - server applications: client in python and pygtk, server in python and web.py:



FirstName	LastName	Company	Address	Country	Phone	Email
Roberto	Almeida	Riotur	Praça Pio X, 119	Brazil	+55 (21) 2271-7000	roberto.almeida@riotur.gov.br
Julia	Barnett		302 S 700 E	USA	+1 (801) 531-7272	jubarnett@gmail.com
Camille	Bernard		4, Rue Milton	France	+33 01 49 70 65 65	camille.bernard@yahoo.fr
Michelle	Brooks		627 Broadway	USA	+1 (212) 221-3546	michelleb@aol.com
Robert	Brown		796 Dundas Street West	Canada	+1 (416) 363-8888	robbrown@shaw.ca
Kathy	Chase		801 W 4th Street	USA	+1 (775) 223-7665	kachase@hotmail.com
Richard	Cunningham		2211 W Berry Street	USA	+1 (817) 924-7272	ricunningham@hotmail.com
Marc	Dubois		11, Place Bellecour	France	+33 04 78 30 30 30	marc.dubois@hotmail.com
João	Fernandes		Rua da Assunção 53	Portugal	+351 (213) 466-111	jfernandes@yahoo.pt
Edward	Francis		230 Elgin Street	Canada	+1 (613) 234-3322	edfrancis@yachoo.ca

Filter: Period range from: 08/07/2013

The don't repeat yourself (DRY) principle underlying the framework allows a developer to focus on programming business logic and not waste time on the routine work of programming interface and server details. That allows to create feature rich, complex and robust applications practically on the fly.

2 Getting started

2.1 Installation.

1. Download the zip package.

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
```

note: on some unix like systems you may need to switch to root or run:

```
sudo python setup.py install
```

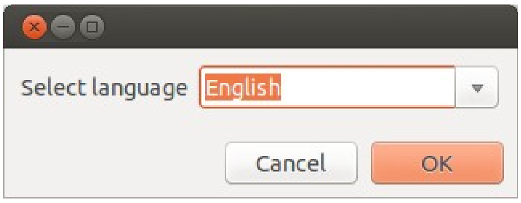
2.2 Creating a new project.

1. Create a new directory.

2. Go into the directory and run from command line:

```
jam-project.py
```

3. In the window that opens, select the language and press **OK** button.

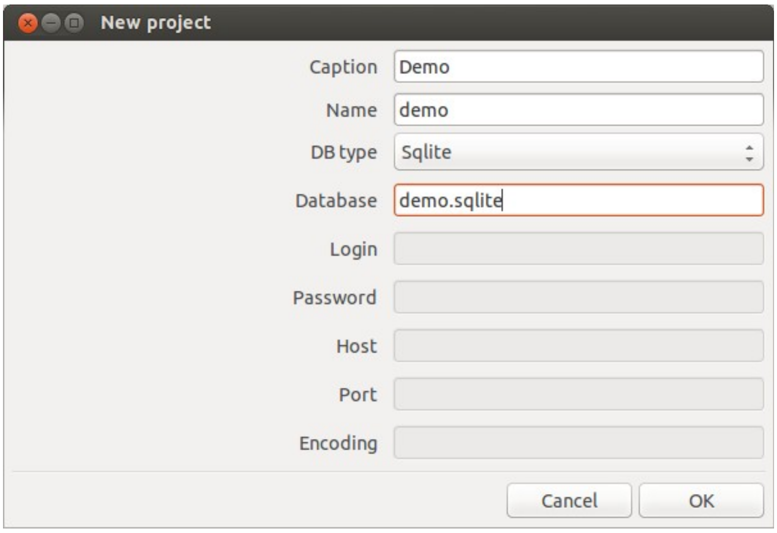


Select language: English

4. In the New project dialog box fill in:

- **Caption** is 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 valid python identifier.
- **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.

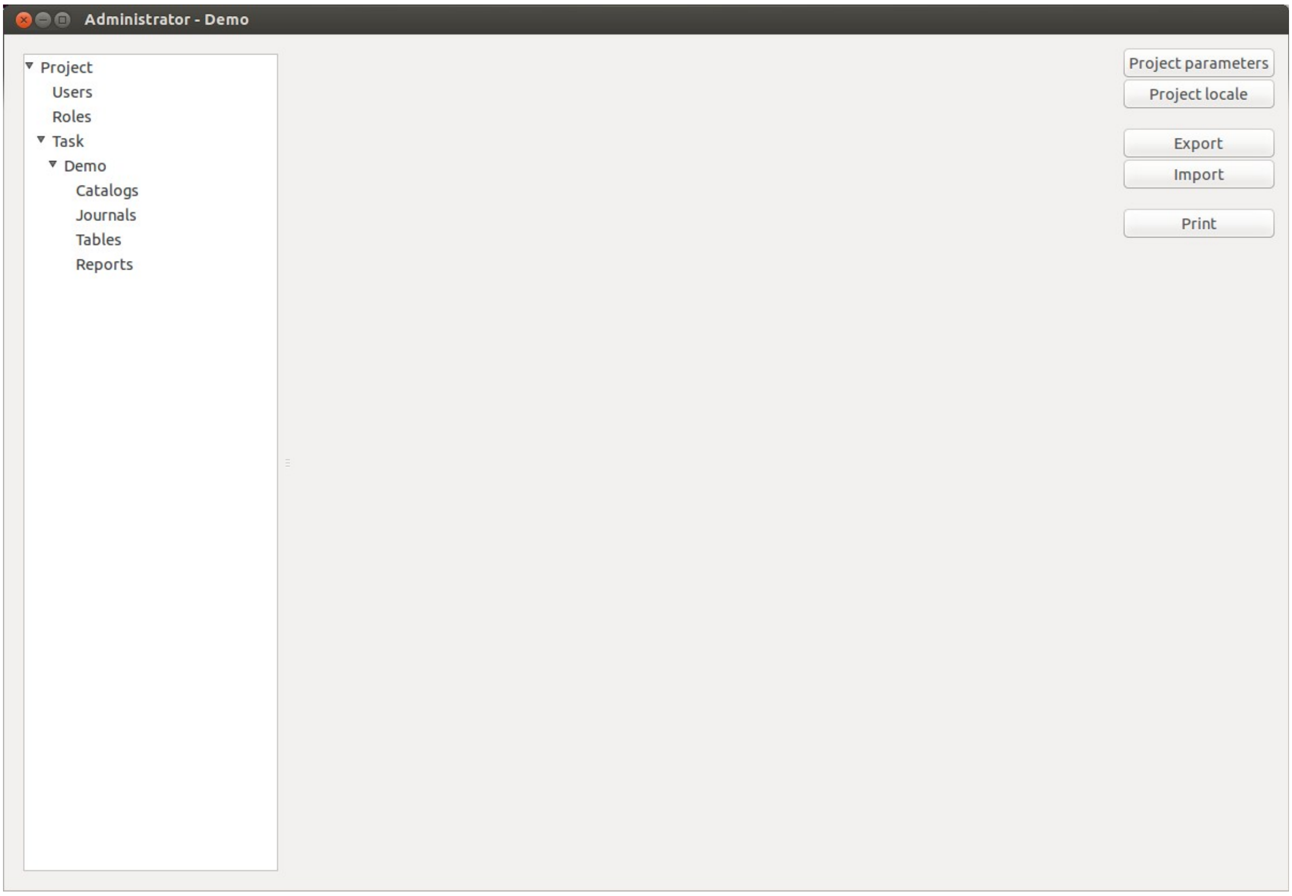
When you press **OK**, the connection to the database will be checked, and in case of failure an error message will be displayed.



New project

Caption: Demo
Name: demo
DB type: Sqlite
Database: demo.sqlite
Login:
Password:
Host:
Port:
Encoding:

If all goes well a new project will be created and a project tree will appear in Administrator.



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

files:

- **server.py** – run this file to start the server. You can specify a port as parameter, for example ./server.py 8081. By default, the port is 8080
- **admin.py** – run this file to start the project Administrator. You can specify an URL and a port as parameters, for example ./admin.py http://127.0.0.1:8080. In this case, Administrator will be launched as a remote desktop client. In the absence of parameters - as a local desktop application
- **main.py** - run this file to start the local desktop application
- **client.py** - run this file to start the remote desktop client. You can specify an URL and a port as parameters. By default, they are http://127.0.0.1:8080
- **index.html** - the main file of web client

folders:

js - javascrip files

- **css** - css files
- **img** - image files
- **ui** – the folder where the glade templates are stored, that are used to create desktop applications
- **static** - static directory of the server

Please note the following requirements:

- to run desctop applications you need to install GTK+2 и PyGTK
- to use FireBird database, the python fdb library must be installed
- PostgreSQL requires psycopg2 library
- to generate reports you should have OpenOffice to be installed

3 Building first jam.py application.

3.1 Demo project.

In the folder where the jam.py package was unzipped there is a demo folder that contains a demo project. In order to see how the demo works it is necessary to go into this folder. To start a local desktop applications run main.py script. To view the work of a client-server application, you must first start the server - server.py. After this in the browser address bar, type 127.0.0.1:8080. To run the remote desctop client go to the demo folder in an another terminal and run from the command line ./client.py.

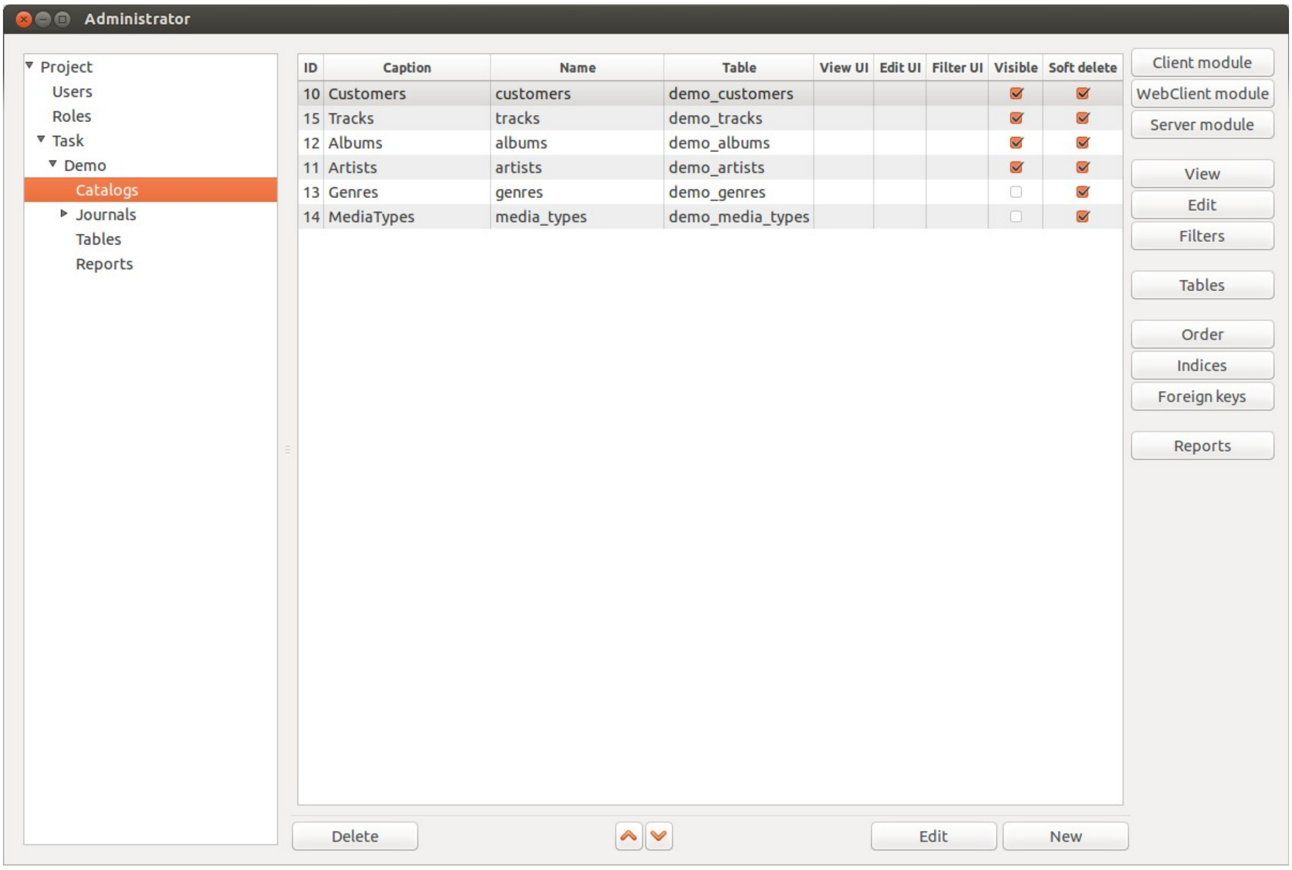
Next we'll try to show how to build such a project.

3.2 Administrator.

Now with the admin.py script run Administrator. Administrator - is a jam.py application intended for application development and database administration. In fact, it contains project metadata - database table structure definitions, programming code, etc.

On the left side of the Administrator window there is a tree view that contains a project tree. Let's click on the Catalogs node - in the center part of Administrator catalogs list will appear.

By selecting any node of the project tree, we open it's content in the center part of Administrator window, and as a rule, in the bottom and right side of the Administrator there are buttons that allow us to modify it's content.



ID	Caption	Name	Table	View UI	Edit UI	Filter UI	Visible	Soft delete
10	Customers	customers	demo_customers				☑	☑
15	Tracks	tracks	demo_tracks				☑	☑
12	Albums	albums	demo_albums				☑	☑
11	Artists	artists	demo_artists				☑	☑
13	Genres	genres	demo_genres				□	☑
14	MediaTypes	media_types	demo_media_types				□	☑

3.3 Building first catalog.

Earlier we created the new project named Demo. As you can see in the project tree there is a node named Demo. The tree where this node is a root node we will call a task tree. Each node of the task tree we will call a tree Item. In fact, they all have the same ancestor class - AbstractItem. The task tree root now have four child nodes (group items): Catalogs, Journals, Tables and Reports. Three of them Catalogs, Journals, Tables can have its own children each of which is associated with a database table. We will call them data items.

Accordingly, all of data items of the project are rather interchangeably divided into 3 categories: Catalogs, Journals, Tables.

Catalogs are data items that contain information of catalog type such as customers, organizations, tracks, etc. When creating other data items, we can create a field that is a reference to the record in a catalog.

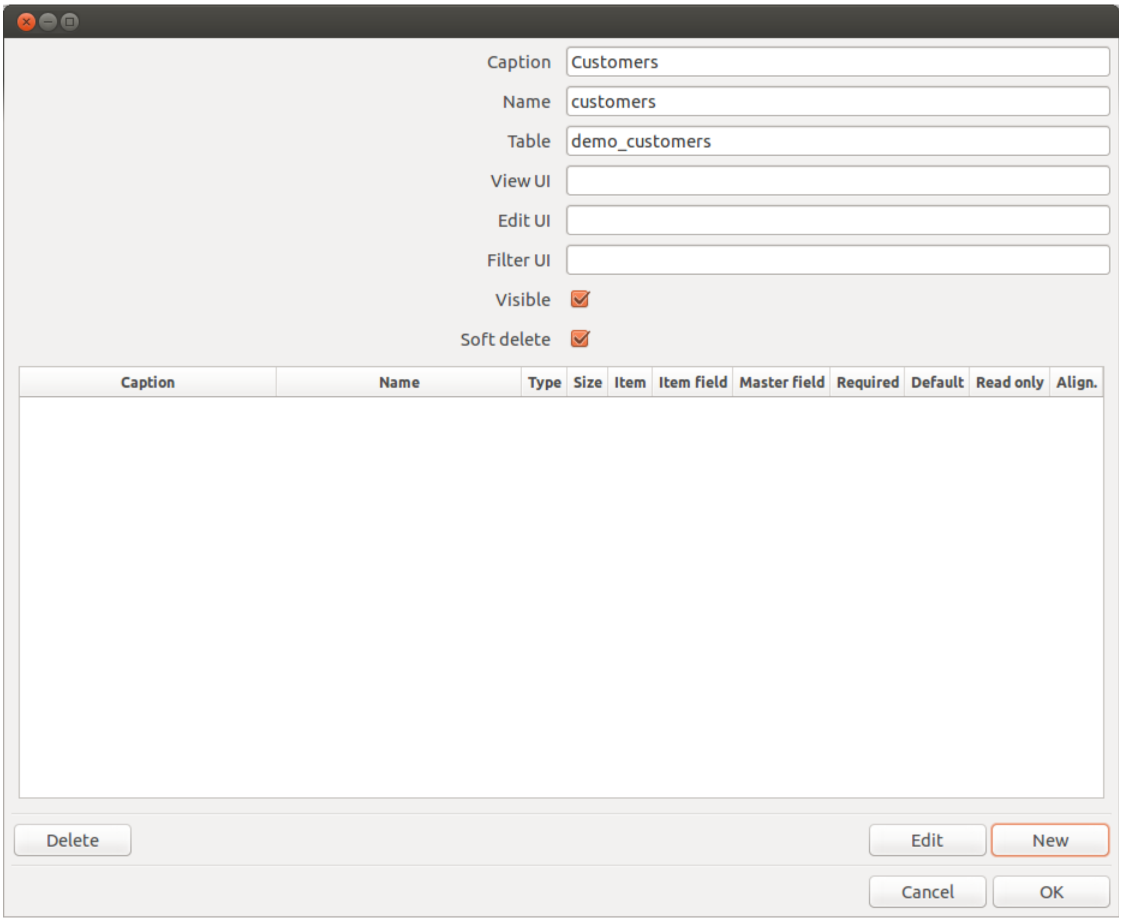
Journals are the structures that store information about events recorded in some documents, such as invoices, purchase orders, etc.

Tables are essentially similar to journals. But besides that they could be embedded into data items. Such as a list of tracks in an invoice.

Let's create catalogs. Click on the Catalogs node in the project tree. The catalog list is empty yet. Let's start with catalog Customers. Click on the New button in the lower-right pane of Administrator.

In the new catalog dialog fill in

- **Caption** is the catalog name that appears to users
- **Name –** the name of the catalog that will be used in programming code to get access to the catalog object. It should be unique in the project and should be a valid python identifier.

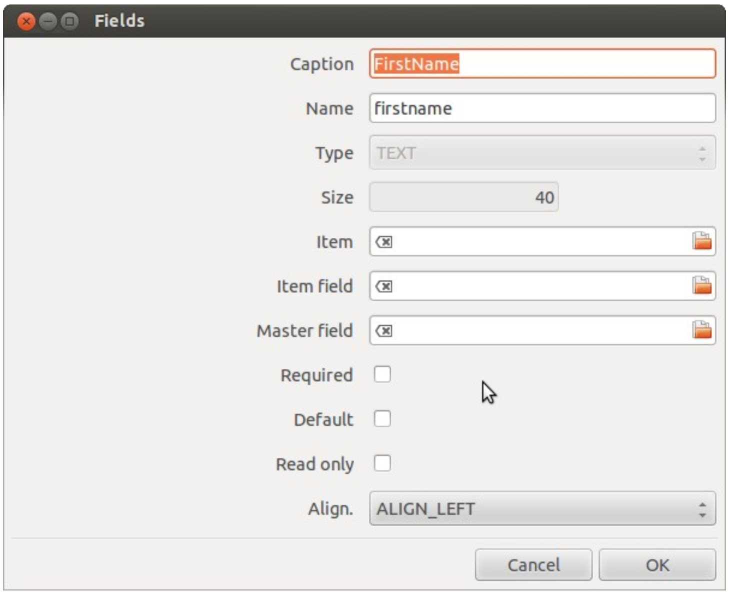


Caption: Customers
Name: customers
Table: demo_customers
View UI:
Edit UI:
Filter UI:
Visible ☑
Soft delete ☑

Caption	Name	Type	Size	Item	Item field	Master field	Required	Default	Read only	Align.

Administrator will generate the name of the table associated with the Customers catalog - DEMO_CUSTOMERS. Let's skip other attributes for a while, we will return to them later when we start discussing interface programming, and move on to creating fields. To do this, click on the **New** button in the lower right corner of the window.

In the window that appears enter the caption of the field, its name (unique in the catalog, valid python identifier), select the type of field, set its length and press the **OK** button.



Caption FirstName

Name firstname

Type TEXT

Size 40

Item

Item field

Master field

Required □

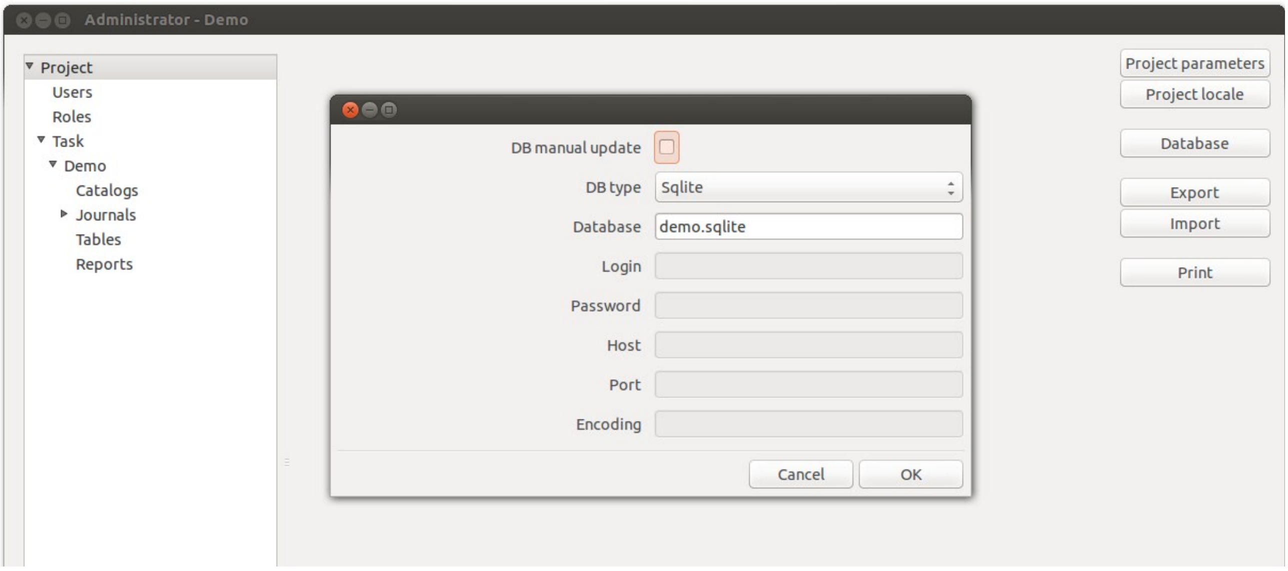
Default □

Read only □

Align. ALIGN_LEFT

We have added the field 'firstname'. Now, let's similarly add the field 'lastname'. But before saving it, click on the check boxes **Required** and **Default**. If check-box **Required** is checked, the client application will not allow a user to save a new record if this field is empty. As far as **Default** check-box is concerned, the default interface implements a search procedure on default field.

Likewise, add the remaining fields and click the **OK** button. Administrator will save new 'Customers' catalog and create a new table DEMO_CUSTOMERS in the project database demo.sqlite. Generally, when we create, modify or delete fields of some data item, framework accordingly updates associated database table. This behavior can be changed by setting the property 'DB manual update' of the project to True. To do so select 'Project' node of project tree then click on **Database** button and check **DB manual update** check-box. From now on fields in the database table should be updated manually. Please be careful with this option.



DB manual update □

DB type Sqlite

Database demo.sqlite

Login

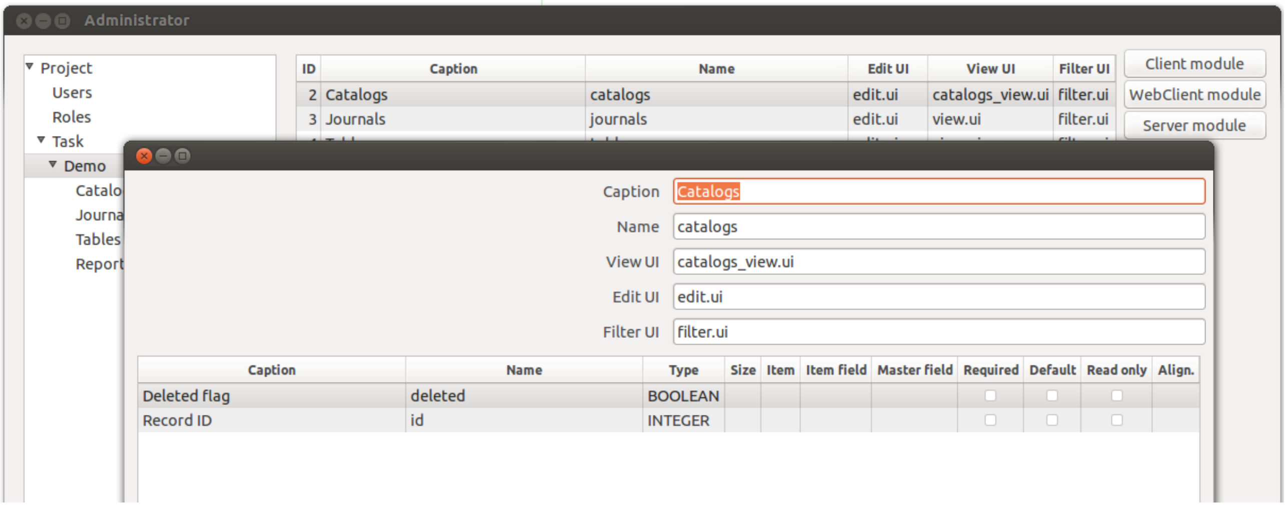
Password

Host

Port

Encoding

If we open the project database in SQLite Manager and examine the structure of demo_customers table, we'll see that in addition to the fields added by us, there are two more fields - 'id' and 'deleted'. They are fields common for all catalogs in the project. If we select the node Demo in the project tree and double-click the record Catalogs we'll see the definition of these two fields.



ID	Caption	Name	Edit UI	View UI	Filter UI
2	Catalogs	catalogs	edit.ui	catalogs_view.ui	filter.ui
3	Journals	journals	edit.ui	view.ui	filter.ui

Caption: Catalogs

Name: catalogs

View UI: catalogs_view.ui

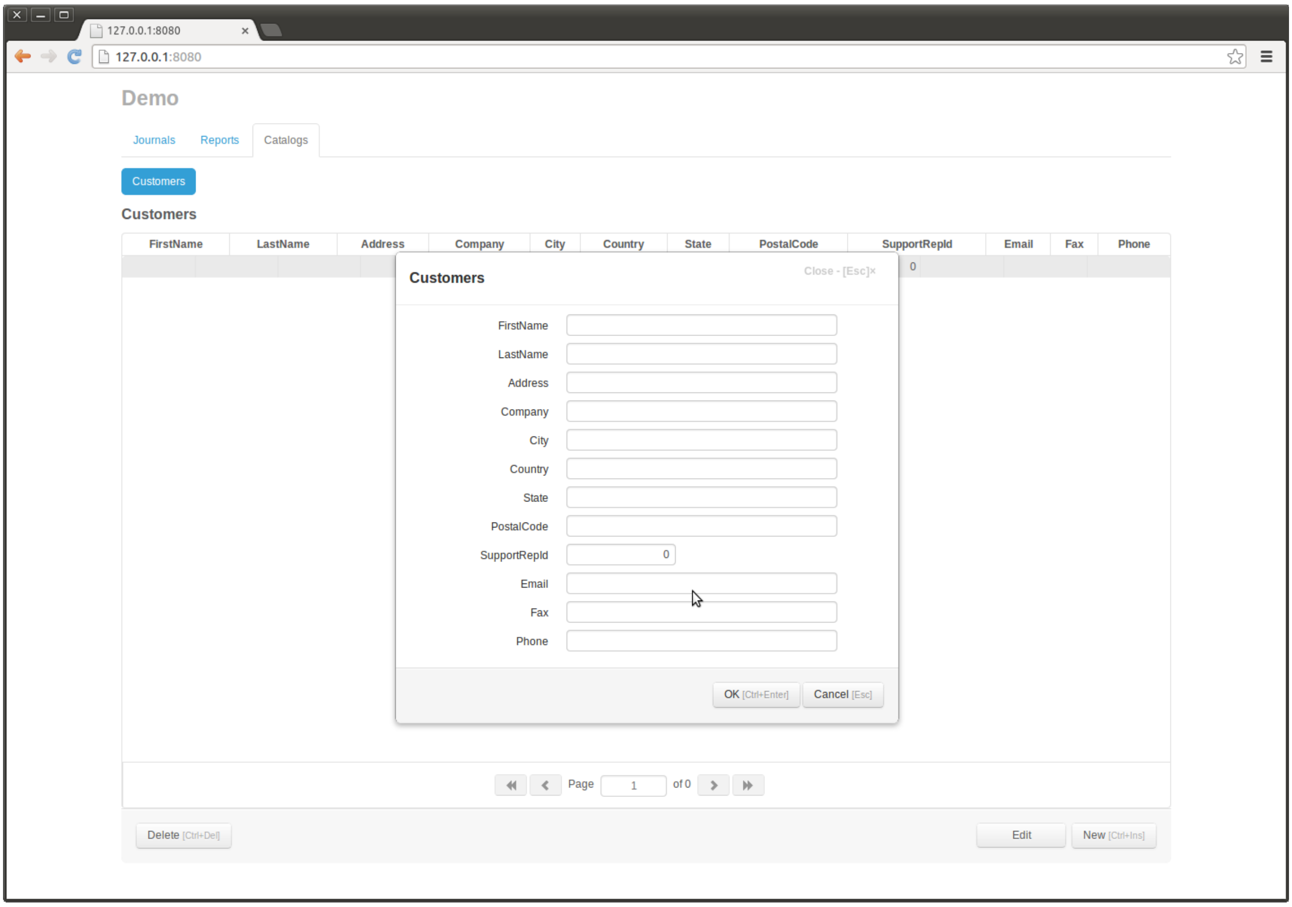
Edit UI: edit.ui

Filter UI: filter.ui

Caption	Name	Type	Size	Item	Item field	Master field	Required	Default	Read only	Align.
Deleted flag	deleted	BOOLEAN					□	□	□	
Record ID	id	INTEGER					□	□	□	

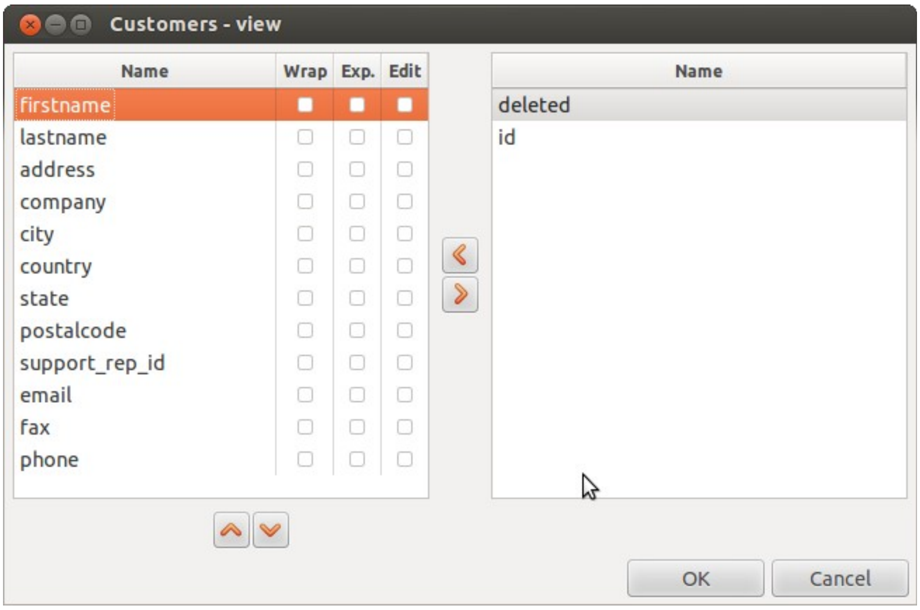
The first 'id' field will contain a unique identifier for each record in the demo_customers table. The second field is a deletion flag. When we were creating the 'Customers' catalog the check-box beside 'Soft delete' attribute was checked. The meaning of this attribute is that if it is set, then when we delete a record from this data item by means of the framework, it will not be erased physically from the associated table, but just marked as deleted.

Start the server and in the browser address bar enter 127.0.0.1:8080. Then click on the menu 'Catalogs'. You will see that there is a sub-menu 'Customers'. Click on it. The customers grid will appear on the page. Click again on the 'New' button in the right-bottom corner. The modal form will be created to add a new customer. All this is possible due to a default interface that is implemented in a new project. This interface can be programmatically changed. We will discuss how to do this in the chapter "Interface programming". By the way, the check-box besides the attribute 'Visible' in the new catalog dialog determines whether it will be shown in the sub-menu Catalogs. And it is programmed in the default interface.



FirstName	
LastName	
Address	
Company	
City	
Country	
State	
PostalCode	
SupportRepId	0
Email	
Fax	
Phone	

To change the default list of fields to be displayed when viewing, go to Administrator, click on the catalog 'Customers' and press the **View** button in the right panel. A window will appear where you can specify the list and the order of the fields when viewing.



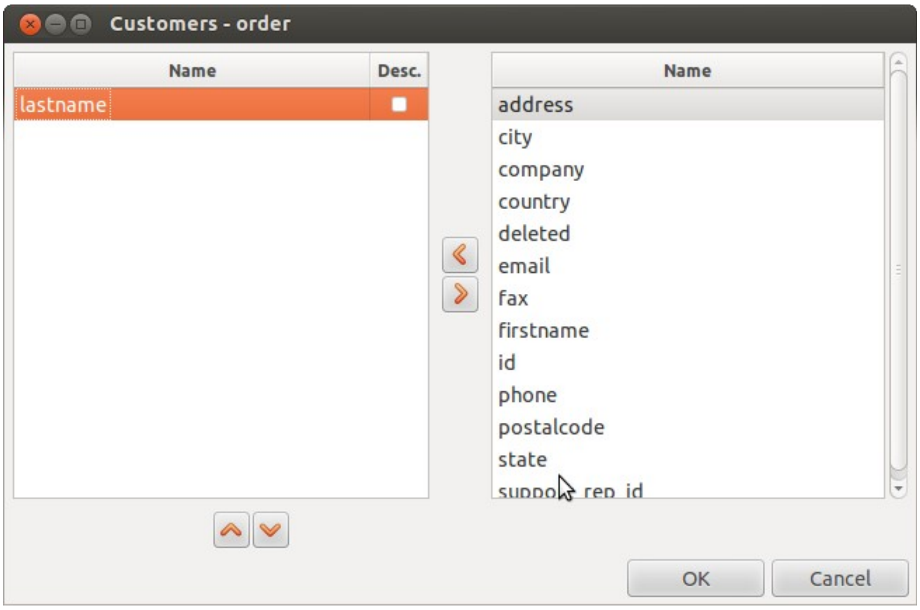
Name	Wrap	Exp.	Edit
firstname	□	□	□
lastname	□	□	□
address	□	□	□
company	□	□	□
city	□	□	□
country	□	□	□
state	□	□	□
postalcode	□	□	□
support_rep_id	□	□	□
email	□	□	□
fax	□	□	□
phone	□	□	□

Name
deleted
id

On the left side are selected fields on the left – all the rest. By clicking on the buttons in the center you can change the list of selected fields. Clicking on the bottom buttons – their order.

The same way you can change the default list of fields to be displayed when editing. Just click on the 'Edit' button.

If we open the Demo project, we'll see that records in the catalog 'Customers' are sorted by the lastname field. To set the default sorting order, click the button 'Order' and specify the fields to sort by and their order.



Customers - order

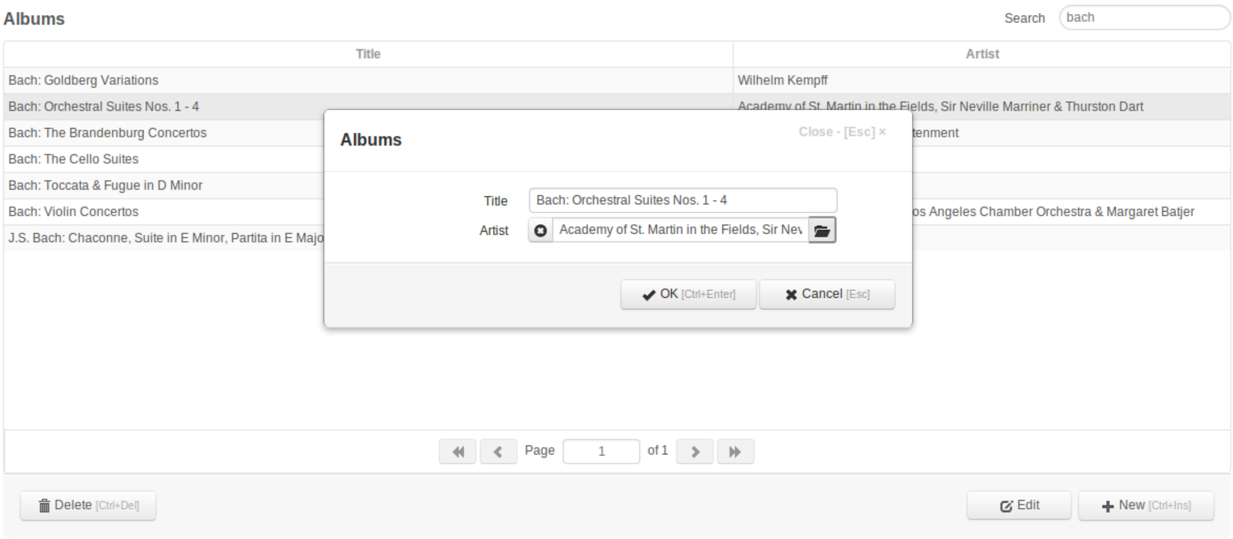
Name	Desc.
lastname	☐

Name
address
city
company
country
deleted
email
fax
firstname
id
phone
postalcode
state
support_rep_id

Check the check-box in Desc column to specify a descending ordering.

3.4 Complete catalogs building.

The same way we create the Artists, Genres and Media types catalogs. But in the Albums catalog the field Artist is a reference to a record in the Artists catalog.



Albums

Search: bach

Title	Artist
Bach: Goldberg Variations	Wilhelm Kempff
Bach: Orchestral Suites Nos. 1 - 4	Academy of St. Martin in the Fields, Sir Neville Marriner & Thurston Dart
Bach: The Brandenburg Concertos	tenment
Bach: The Cello Suites	
Bach: Toccata & Fugue in D Minor	
Bach: Violin Concertos	os Angeles Chamber Orchestra & Margaret Batjer
J.S. Bach: Chaconne, Suite in E Minor, Partita in E Majo	

Albums

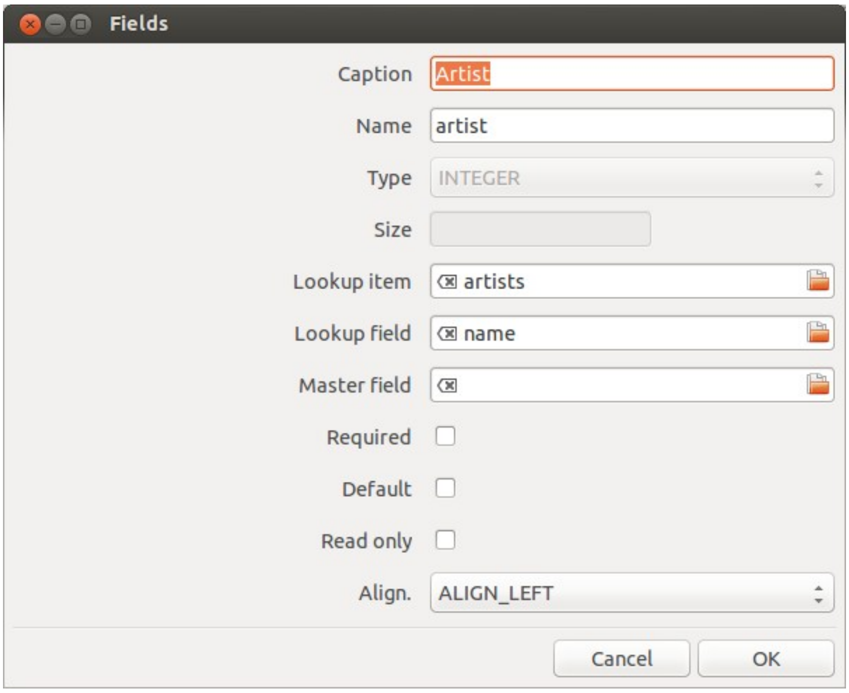
Title: Bach: Orchestral Suites Nos. 1 - 4

Artist: Academy of St. Martin in the Fields, Sir Nev

Page 1 of 1

To change its value we should click on the button to the right of the input and select a new artist name from the artists list.

So when creating this field we must to select 'artists' catalog in the Lookup item attribute and the 'name' field as the Lookup field.



Fields

Caption: Artist

Name: artist

Type: INTEGER

Size:

Lookup item: artists

Lookup field: name

Master field:

Required □

Default □

Read only □

Align.: ALIGN_LEFT

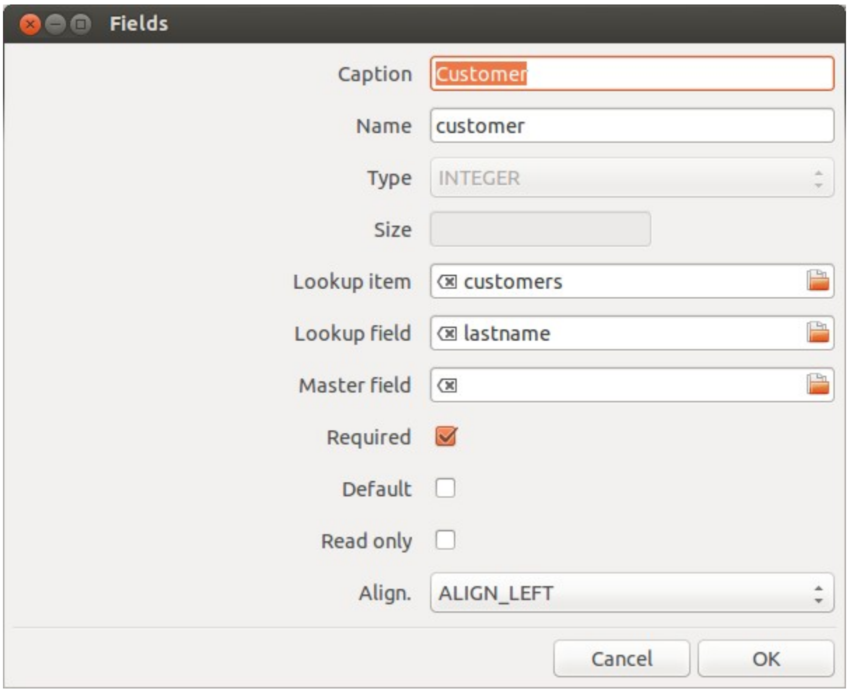
After we save changes to the item, in an underlying table 'DEMO_ALBUMS' of the database demo.sqlite an 'ARTIST' field of type INTEGER will be created. This field will store the id value of the record in 'Artists' catalog.

The last catalog - Tracks has three such fields: album, genre and media_type. With the creation of the Tracks catalog we complete the catalog building.

3.5 Creating journals and tables.

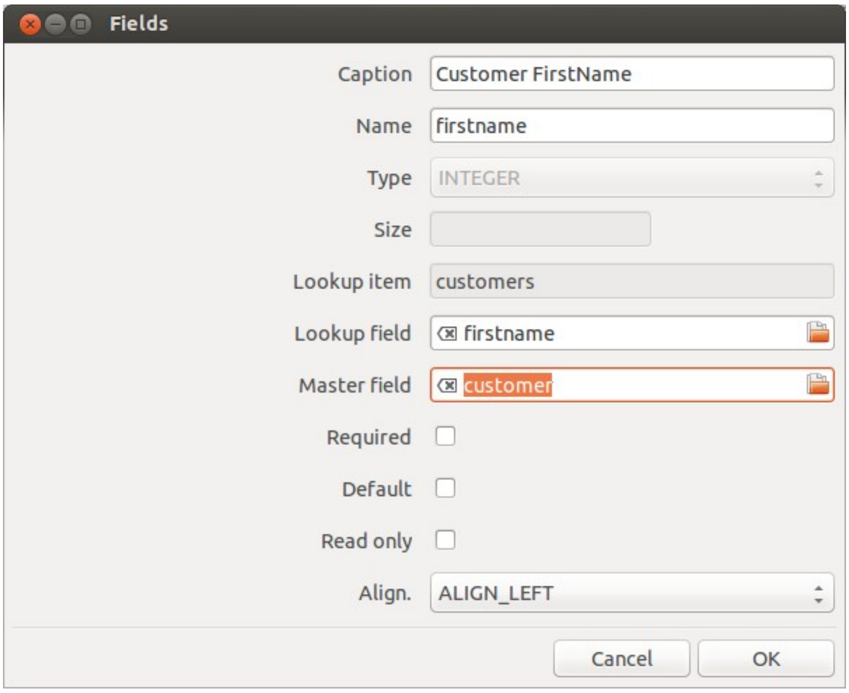
The project Demo has a journal - 'Invoices' and a table - 'InvoiceTable'. In principle, the creation of data structures for journals and tables is no different from creating data structures for catalogs. So here we'll just show how to create linked fields.

The journal 'Invoices' have a field named 'customer' that is a lookup field which lookup item is the 'Customers' catalog and lookup field is the 'lastname' field.



Caption	Customer
Name	customer
Type	INTEGER
Size	
Lookup item	customers
Lookup field	lastname
Master field	
Required	☑
Default	□
Read only	□
Align.	ALIGN_LEFT

However if want to add to the journal a field that will contain the first name of the customer there is no need to change underlying table DEMO_INVOICES (it already have a field CUSTOMER).



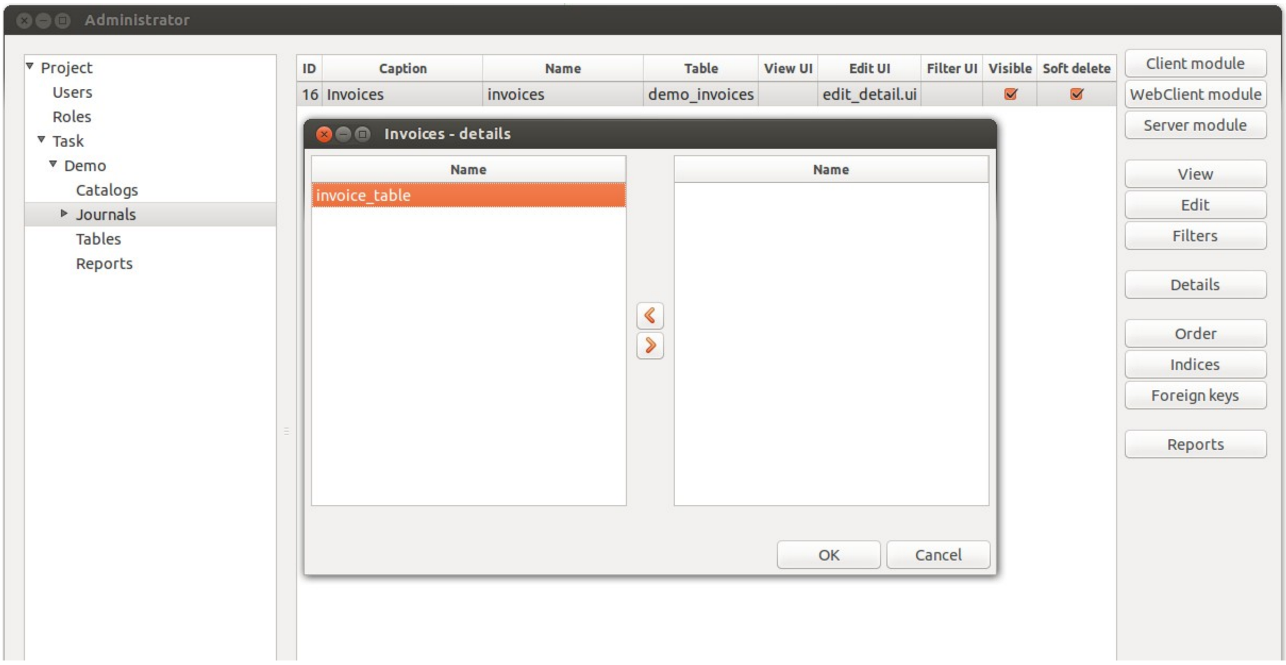
Caption	Customer FirstName
Name	firstname
Type	INTEGER
Size	
Lookup item	customers
Lookup field	firstname
Master field	customer
Required	□
Default	□
Read only	□
Align.	ALIGN_LEFT

To specify this we set 'Master field' attribute to field 'customer'. But before adding this field we must save item changes, so field customer will be saved and it's ID value will be assigned.

So the field firstname is linked to the field customer and customer is a master field of the firstname field and when we change customer field value by selecting record in Customers catalog the firstname field value will change simultaneously.

After creating journal 'Invoices' and a table 'InvoiceTable' we will now add a detail item 'InvoiceTable' to the journal 'Invoices'. To do so we select the journal 'Invoices' and then click on 'Details' button.

In the 'invoices-details' dialog move invoice_table to the left by clicking on the button in the center and save changes by clicking on **OK** button.



ID	Caption	Name	Table	View UI	Edit UI	Filter UI	Visible	Soft delete
16	Invoices	invoices	demo_invoices		edit_detail.ui		☑	☑

Invoices - details

Name
invoice_table

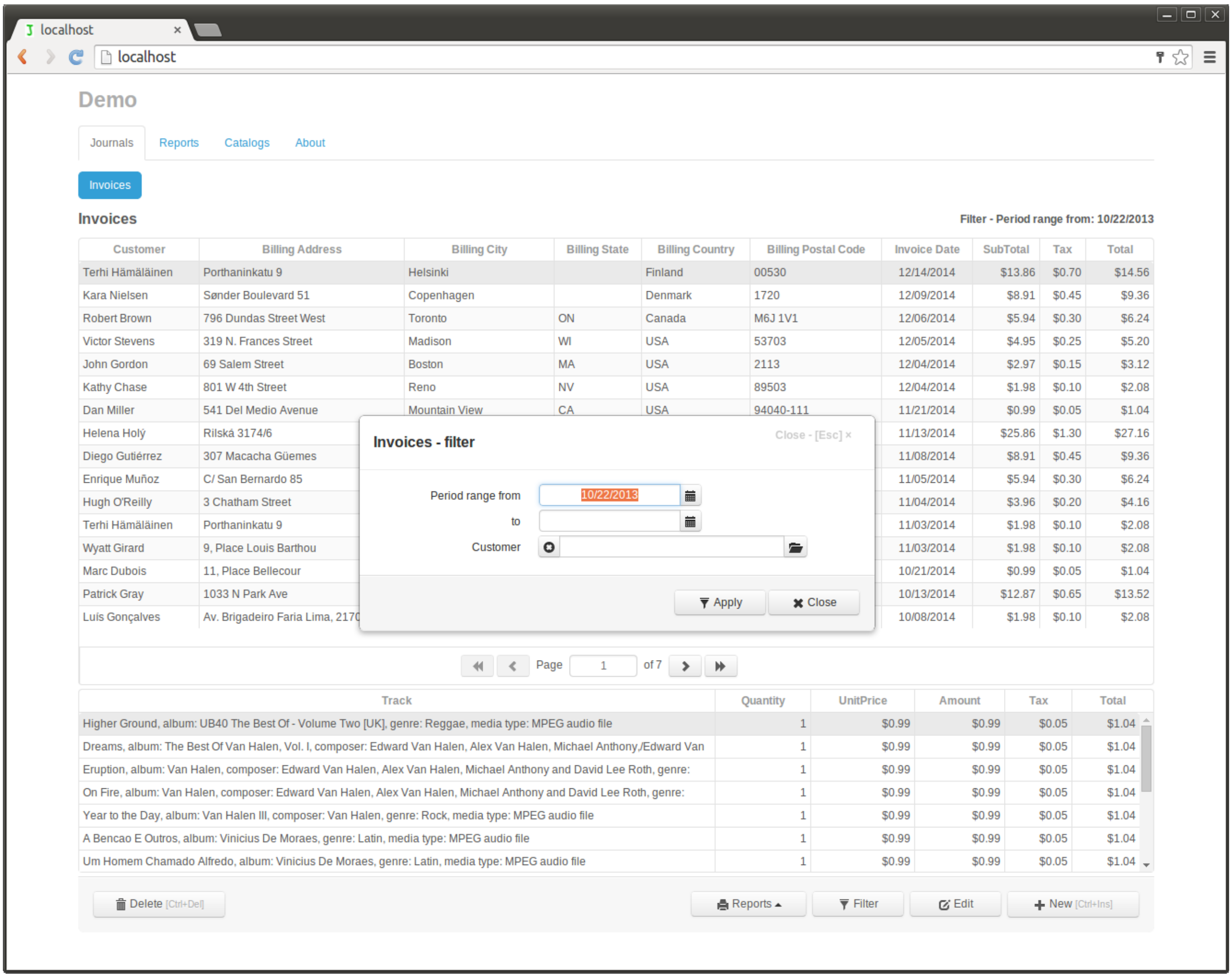
Name

As a result the node 'Journal' in the task tree will have a child node 'Invoices'. If we select it the detail items of 'Invoices' journal will be displayed in the center of the Administrator and we will be able to program their events and change their display options.

So we have the table item named invoice_table which owner is the Tables node of the project tree and detail item named invoice_table which owner (muster) is journal Invoices. Both of them get their data from underlying table DEMO_INVOICE_TABLE. But detail item deals only with track records that belong to the current invoice. If we select Demo node and then double-click Tables record we'll see that in addition to the fields id and deleted, there are two more fields — owner_id and owner_rec_id. So when we save invoice data, each track of this invoice will keep ID of the journal invoice in the owner_id field (each item in the project has its own ID) and id of the current record in the journal in the owner_rec_id field. This way we can link the same table to different journals, catalogs or tables.

3.6 *Creating filters.*

If you open journal Invoices in the Demo application and then click on the Filter button a modal dialog will appear that lets you specify journal filtering options.



Invoices

Filter - Period range from: 10/22/2013

Customer	Billing Address	Billing City	Billing State	Billing Country	Billing Postal Code	Invoice Date	SubTotal	Tax	Total
Terhi Hämäläinen	Porthaninkatu 9	Helsinki		Finland	00530	12/14/2014	$13.86	$0.70	$14.56
Kara Nielsen	Sønder Boulevard 51	Copenhagen		Denmark	1720	12/09/2014	$8.91	$0.45	$9.36
Robert Brown	796 Dundas Street West	Toronto	ON	Canada	M6J 1V1	12/06/2014	$5.94	$0.30	$6.24
Victor Stevens	319 N. Frances Street	Madison	WI	USA	53703	12/05/2014	$4.95	$0.25	$5.20
John Gordon	69 Salem Street	Boston	MA	USA	2113	12/04/2014	$2.97	$0.15	$3.12
Kathy Chase	801 W 4th Street	Reno	NV	USA	89503	12/04/2014	$1.98	$0.10	$2.08
Dan Miller	541 Del Medio Avenue	Mountain View	CA	USA	94040-111	11/21/2014	$0.99	$0.05	$1.04
Helena Holý	Rilská 3174/6					11/13/2014	$25.86	$1.30	$27.16
Diego Gutiérrez	307 Macacha Güemes					11/08/2014	$8.91	$0.45	$9.36
Enrique Muñoz	C/ San Bernardo 85					11/05/2014	$5.94	$0.30	$6.24
Hugh O'Reilly	3 Chatham Street					11/04/2014	$3.96	$0.20	$4.16
Terhi Hämäläinen	Porthaninkatu 9					11/03/2014	$1.98	$0.10	$2.08
Wyatt Girard	9, Place Louis Barthou					11/03/2014	$1.98	$0.10	$2.08
Marc Dubois	11, Place Bellecour					10/21/2014	$0.99	$0.05	$1.04
Patrick Gray	1033 N Park Ave					10/13/2014	$12.87	$0.65	$13.52
Luís Gonçalves	Av. Brigadeiro Faria Lima, 2170					10/08/2014	$1.98	$0.10	$2.08

Invoices - filter

Close - [Esc] ×

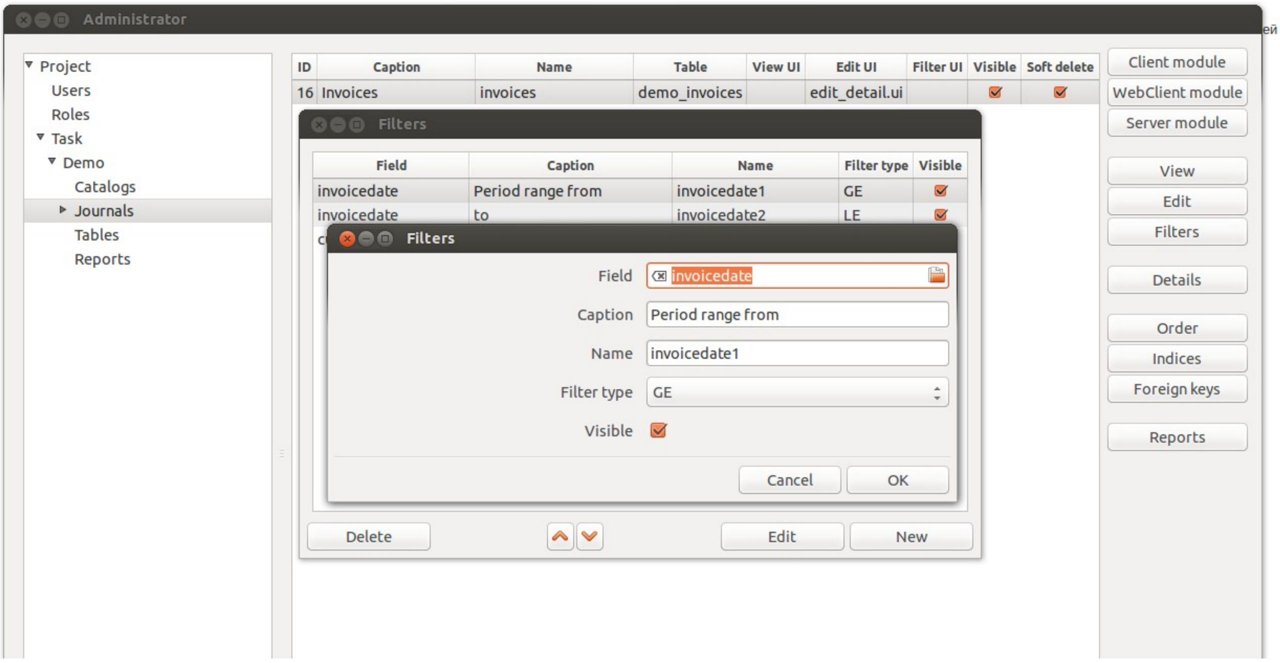
Period range from: 10/22/2013

to:

Customer:

Track	Quantity	UnitPrice	Amount	Tax	Total
Higher Ground, album: UB40 The Best Of - Volume Two [UK], genre: Reggae, media type: MPEG audio file	1	$0.99	$0.99	$0.05	$1.04
Dreams, album: The Best Of Van Halen, Vol. I, composer: Edward Van Halen, Alex Van Halen, Michael Anthony,/Edward Van	1	$0.99	$0.99	$0.05	$1.04
Eruption, album: Van Halen, composer: Edward Van Halen, Alex Van Halen, Michael Anthony and David Lee Roth, genre:	1	$0.99	$0.99	$0.05	$1.04
On Fire, album: Van Halen, composer: Edward Van Halen, Alex Van Halen, Michael Anthony and David Lee Roth, genre:	1	$0.99	$0.99	$0.05	$1.04
Year to the Day, album: Van Halen III, composer: Van Halen, genre: Rock, media type: MPEG audio file	1	$0.99	$0.99	$0.05	$1.04
A Bencao E Outros, album: Vinicius De Moraes, genre: Latin, media type: MPEG audio file	1	$0.99	$0.99	$0.05	$1.04
Um Homem Chamado Alfredo, album: Vinicius De Moraes, genre: Latin, media type: MPEG audio file	1	$0.99	$0.99	$0.05	$1.04

To create or modify a filter in the Administrator choose journal Invoices and click on the Filter button located on the right panel. This opens a form containing the list of available filters. To add or edit a filter click on the appropriate button on the form. Filter editor will appear. After that, you should specify the field which will be used to filter records and fill in the caption, name and type of the filter.



ID	Caption	Name	Table	View UI	Edit UI	Filter UI	Visible	Soft delete
16	Invoices	invoices	demo_invoices		edit_detail.ui		☑	☑

Filters

Field	Caption	Name	Filter type	Visible
invoicedate	Period range from	invoicedate1	GE	☑
invoicedate	to	invoicedate2	LE	☑

Filters

Field: invoicedate

Caption: Period range from

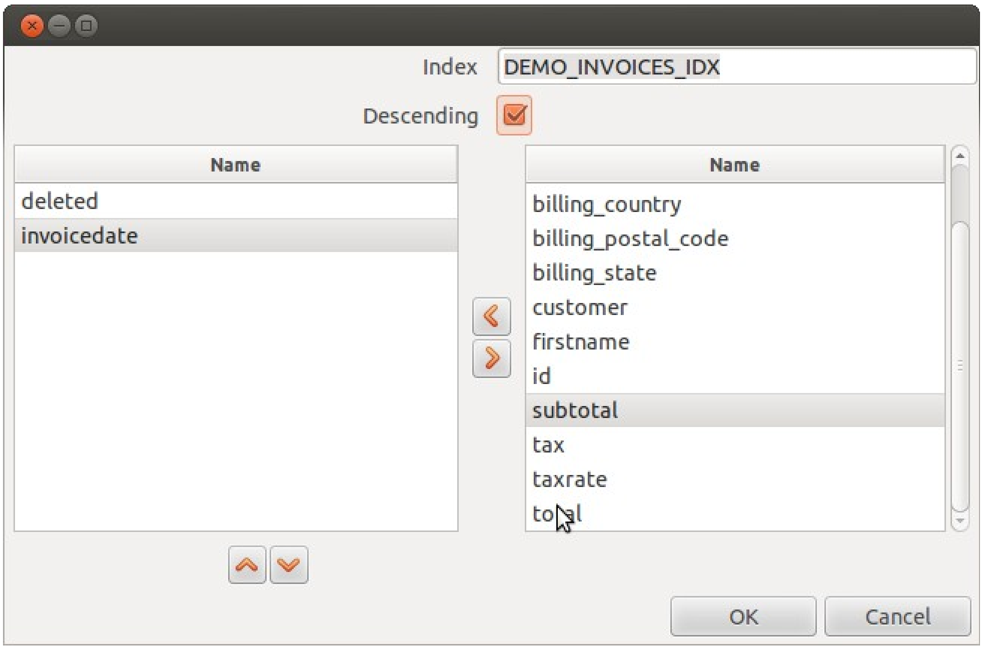
Name: invoicedate1

Filter type: GE

Visible ☑

3.7 *Creating indices.*

We have created all the needed data structures. We now proceed to the creation of indexes. Select Invoices and click on the Indices button. In a window that appear, lets click on the "New" button, and specify descending index on the field invoicedate. If necessary, change the name of the index.



Index: DEMO_INVOICES_IDX

Descending ☑

Name
deleted
invoicedate

Name
billing_country
billing_postal_code
billing_state
customer
firstname
id
subtotal
tax
taxrate
total

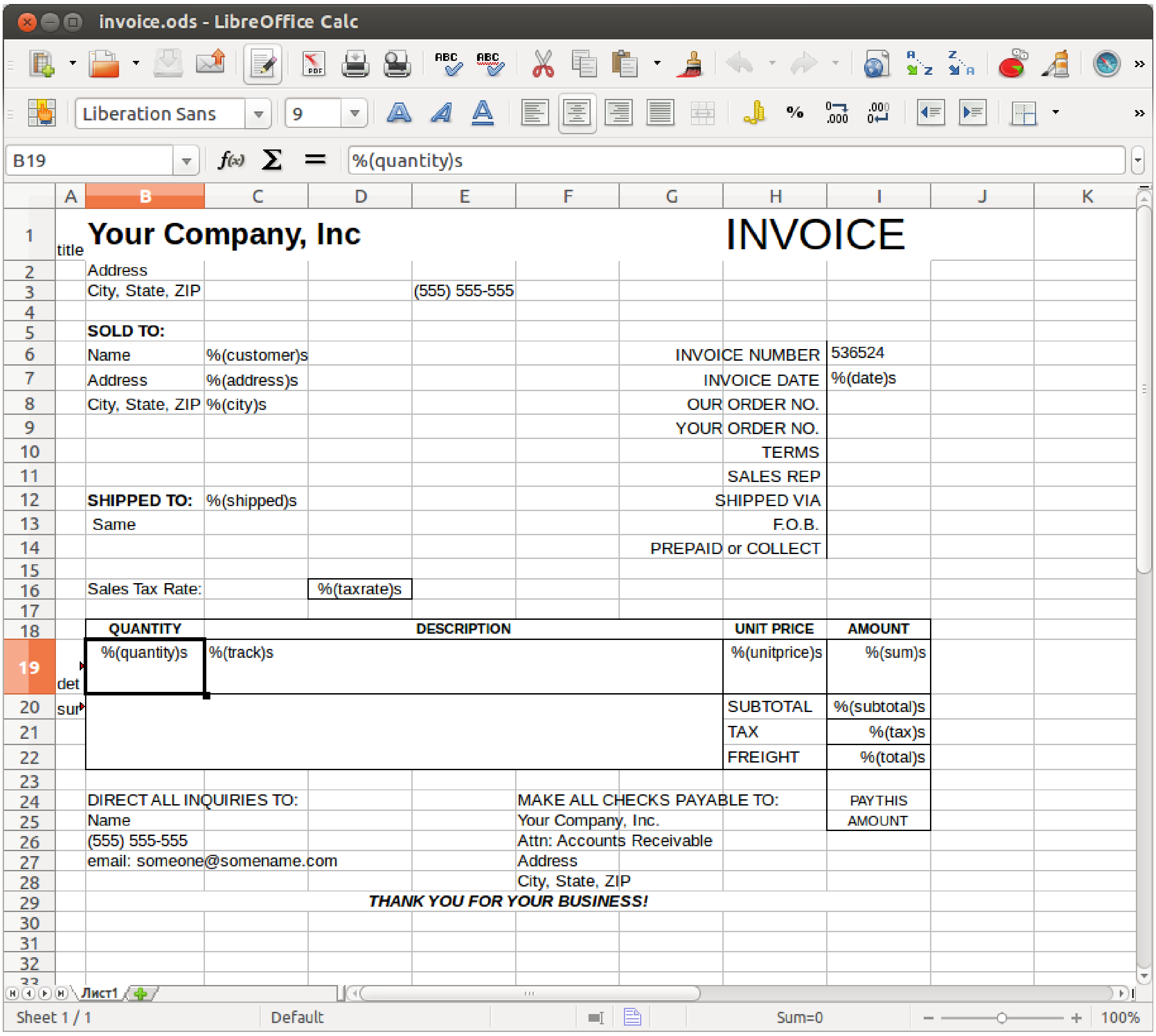
Now click on the OK button and create the index.

In the same way we'll create index for the table InvoiceTable on fields owner_id and owner_rec_id.

If an item has a lookup field and in the definition of lookup item of this field soft delete attribute is not set, in order to preserve the integrity of the data, we can create a foreign key. To do this, click on the Foreign keys button, select the field and press OK.

3.8 Building reports.

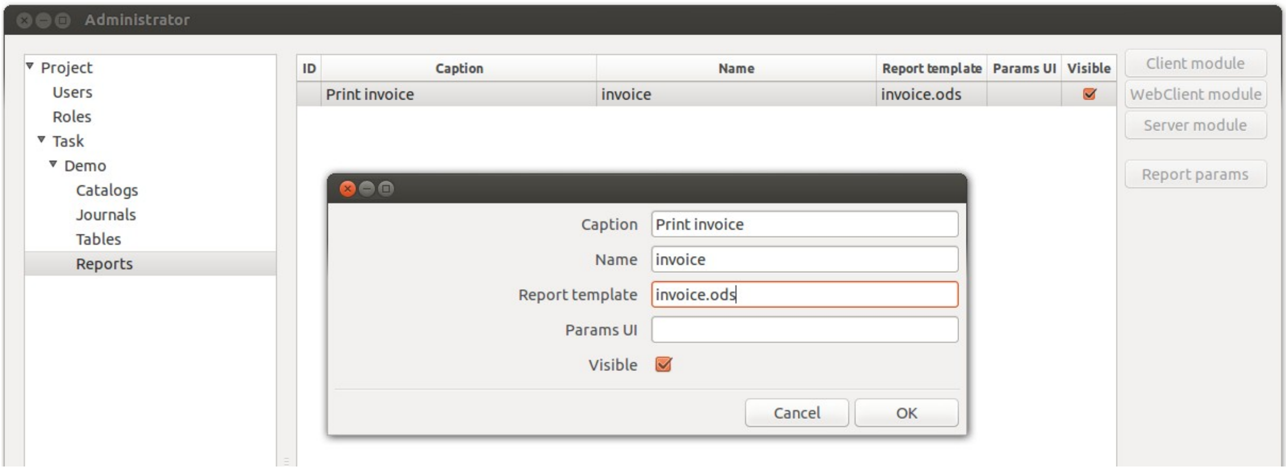
To create a report, you must first prepare a report template in OpenOffice (LibreOffice) Calc. The template files are located in the report folder of the project directory. The following figure shows a template of the Invoice report. Reports in jam.py are band-oriented. Each report template is divided 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 characters s.



%(quantity)s

	A	B	C	D	E	F	G	H	I	J	K
1	title	Your Company, Inc						INVOICE			
2		Address									
3		City, State, ZIP			(555) 555-555						
4											
5		SOLD TO:									
6		Name	%(customer)s					INVOICE NUMBER	536524		
7		Address	%(address)s					INVOICE DATE	%(date)s		
8		City, State, ZIP	%(city)s					OUR ORDER NO.			
9								YOUR ORDER NO.			
10								TERMS			
11								SALES REP			
12		SHIPPED TO:	%(shipped)s					SHIPPED VIA			
13		Same						F.O.B.			
14								PREPAID or COLLECT			
15											
16		Sales Tax Rate:		%(taxrate)s							
17											
18		QUANTITY	DESCRIPTION					UNIT PRICE	AMOUNT		
19	det	%(quantity)s	%(track)s					%(unitprice)s	%(sum)s		
20	sum							SUBTOTAL	%(subtotal)s		
21								TAX	%(tax)s		
22								FREIGHT	%(total)s		
23											
24		DIRECT ALL INQUIRIES TO:				MAKE ALL CHECKS PAYABLE TO:			PAY THIS		
25		Name				Your Company, Inc.			AMOUNT		
26		(555) 555-555				Attn: Accounts Receivable					
27		email: someone@somename.com				Address					
28						City, State, ZIP					
29		*THANK YOU FOR YOUR BUSINESS!*									
30											
31											
32											

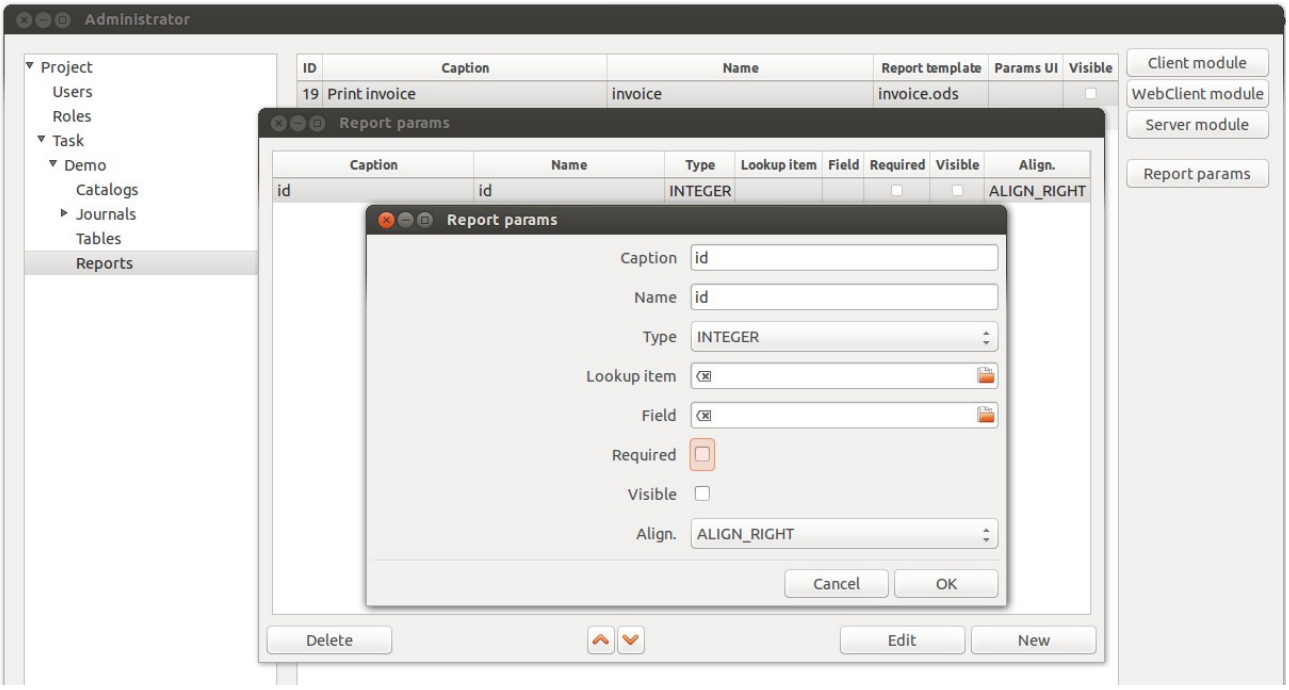
Let's add Invoice report to our project. To do this, choose the **Reports** node in the project tree, click the **New** button and fill in the caption, name and template file name of Invoice report in the **Caption**, **Name** and **Report template** fields accordingly.



ID	Caption	Name	Report template	Params UI	Visible
	Print invoice	invoice	invoice.ods		☑

Caption	Print invoice
Name	invoice
Report template	invoice.ods
Params UI	
Visible	☑

Since all reports are generated on the server, it is necessary to pass to the server the id value of the current invoice. To do this, we will create a report parameter. Let's click on the **Report params** button in the right pane, and then in the dialog box that appears click on the **New** button.



ID	Caption	Name	Report template	Params UI	Visible
19	Print invoice	invoice	invoice.ods		□

Caption	Name	Type	Lookup item	Field	Required	Visible	Align.
id	id	INTEGER			□	□	ALIGN_RIGHT

Caption	id
Name	id
Type	INTEGER
Lookup item	
Field	
Required	□
Visible	□
Align.	ALIGN_RIGHT

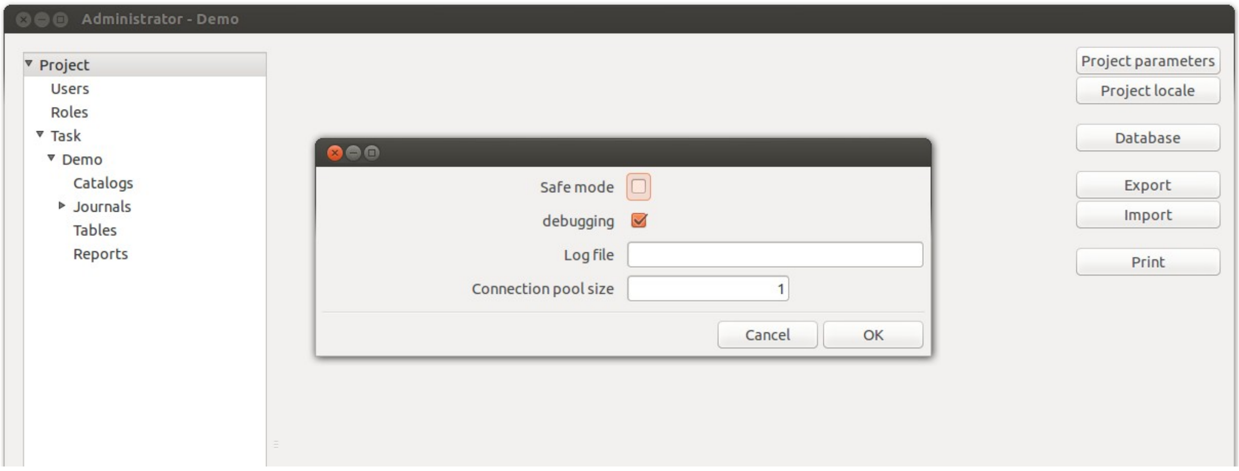
Lets fill in the form that appears caption, name and type of the parameter, uncheck check-box **Visible** and click on **OK** to save information.

The very process of generating a report on the server will be discussed later in the chapter "Report Programming".

Now before proceeding to consider the programming of jam.py framework we'll discuss project parameters and its security system.

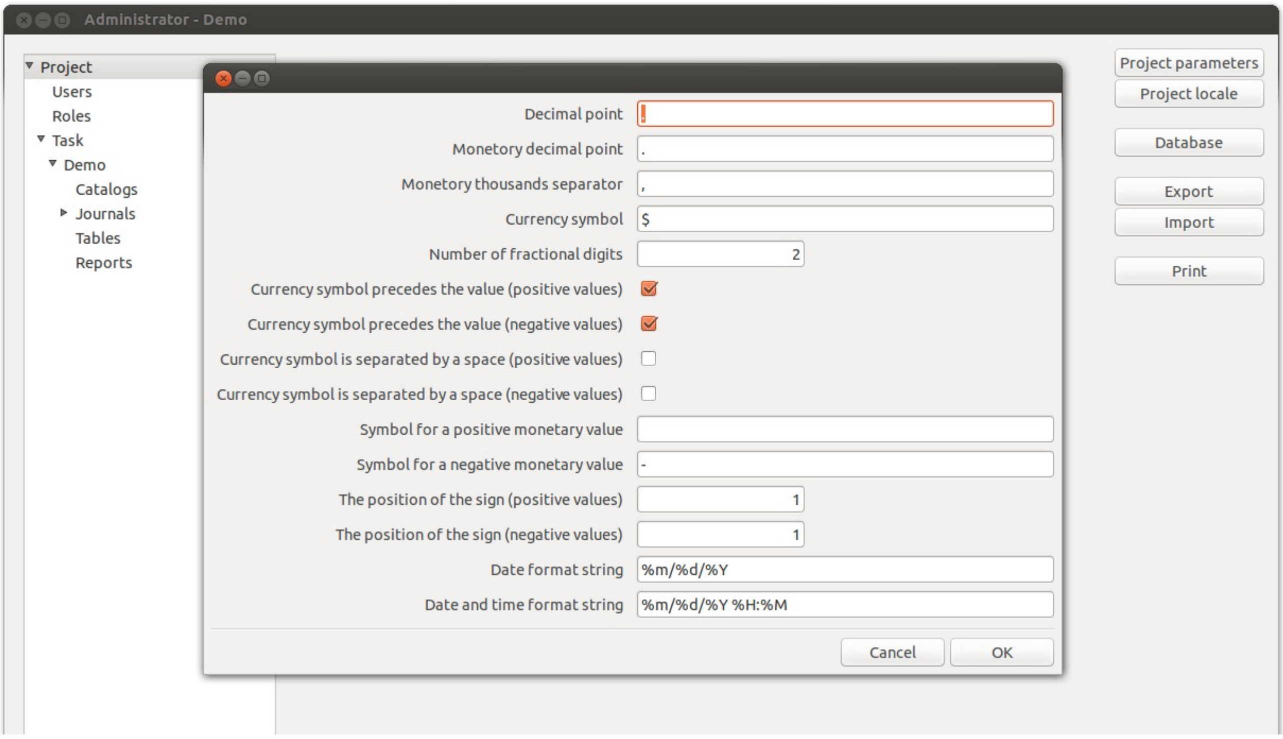
3.9 Project parameters.

To setup the parameters of the project select node **Project** and then click on **Project parameters** button.



Safe mode ☐
debugging ☑
Log file
Connection pool size 1

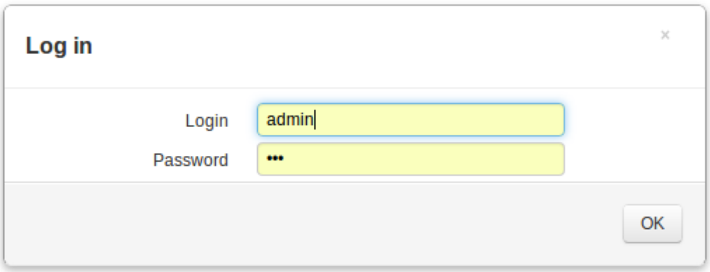
- **Safe mode -** If safe mode is enabled, authentication is needed for user to work in the system (See "Users and Roles").
- **Log file -** If you specify a log file, output to stdout / stderr is redirected to that file.
- **Connection pool size** — the size of the server database connection pool.



Decimal point
Monetory decimal point .
Monetory thousands separator ,
Currency symbol $
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 %m/%d/%Y
Date and time format string %m/%d/%Y %H:%M

3.10 Users and roles.

If parameter **Save mode** is set to work on client user must enter his login and password



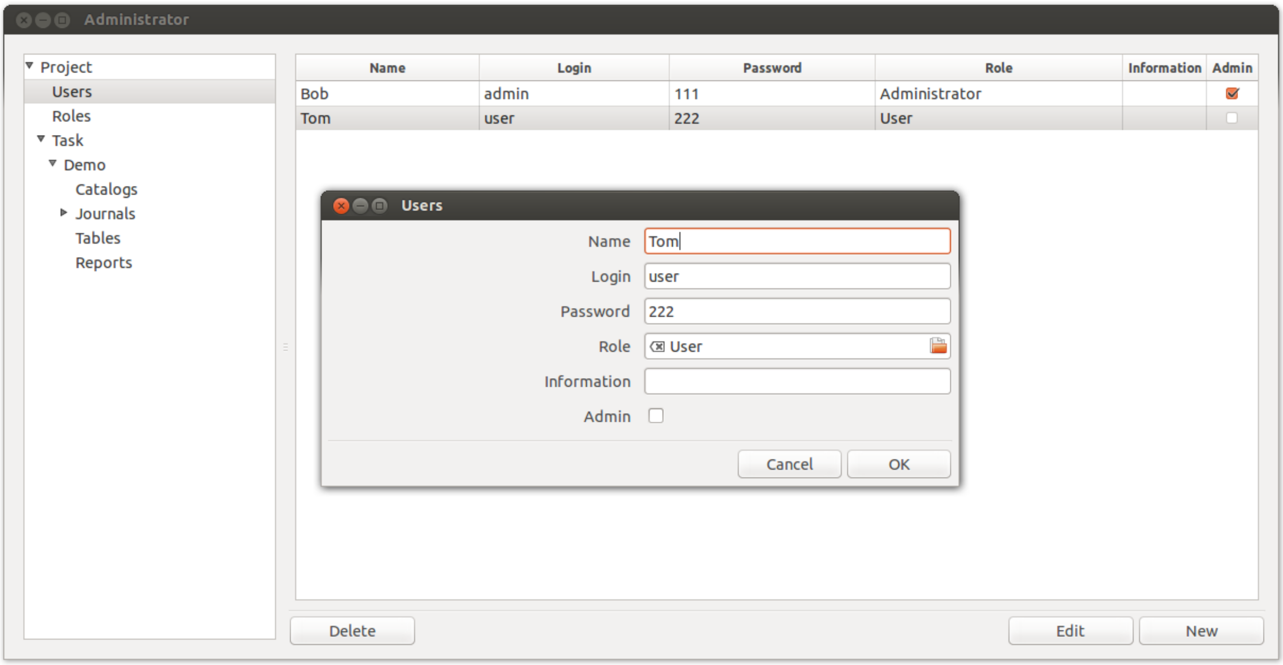
Log in

Login: admin

Password: •••

But before that, the user must be registered in the system. To register user select Users node, click New and fill in the form that appears:

- **Name –** user name
- **Login** - login
- **Password** - password
- **Role** – user roles (see below)
- **Information** - some additional information
- **Admin** - if this flag is set, the user has the right to work in Administrator.

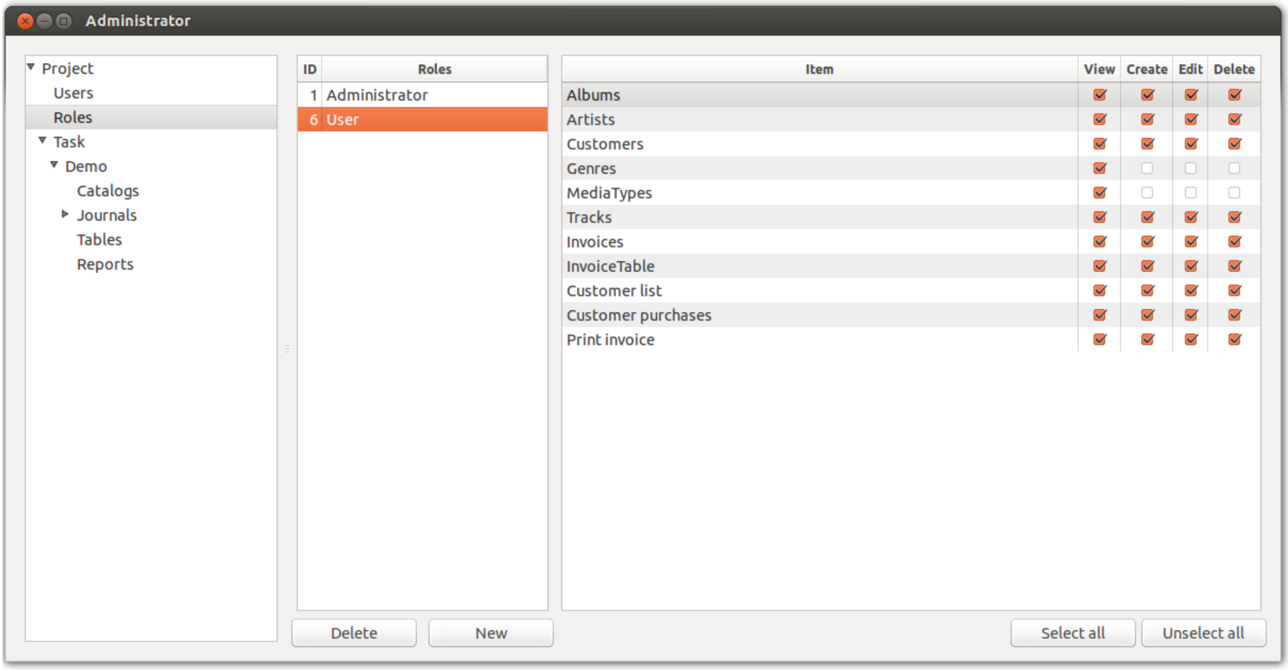


Name	Login	Password	Role	Information	Admin
Bob	admin	111	Administrator		☑
Tom	user	222	User		□

Users

Name	Tom
Login	user
Password	222
Role	User
Information	
Admin	□

Each user must be assigned to one of roles defined in the system. A role defines the user's rights to view, create, modify, and delete data. To work with roles select node **Roles** in the project tree. To add or delete a role, use the buttons **New** and **Delete**. To set permissions for a role, put a check mark next to the appropriate column of the item: View, Create, Edit, Delete (allowed to view, create, modify and delete, respectively).



- Project
  - Users
  - Roles
  - Task
    - Demo
      - Catalogs
      - Journals
      - Tables
      - Reports

ID	Roles
1	Administrator
6	User

Item	View	Create	Edit	Delete
Albums	☑	☑	☑	☑
Artists	☑	☑	☑	☑
Customers	☑	☑	☑	☑
Genres	☑	□	□	□
MediaTypes	☑	□	□	□
Tracks	☑	☑	☑	☑
Invoices	☑	☑	☑	☑
InvoiceTable	☑	☑	☑	☑
Customer list	☑	☑	☑	☑
Customer purchases	☑	☑	☑	☑
Print invoice	☑	☑	☑	☑

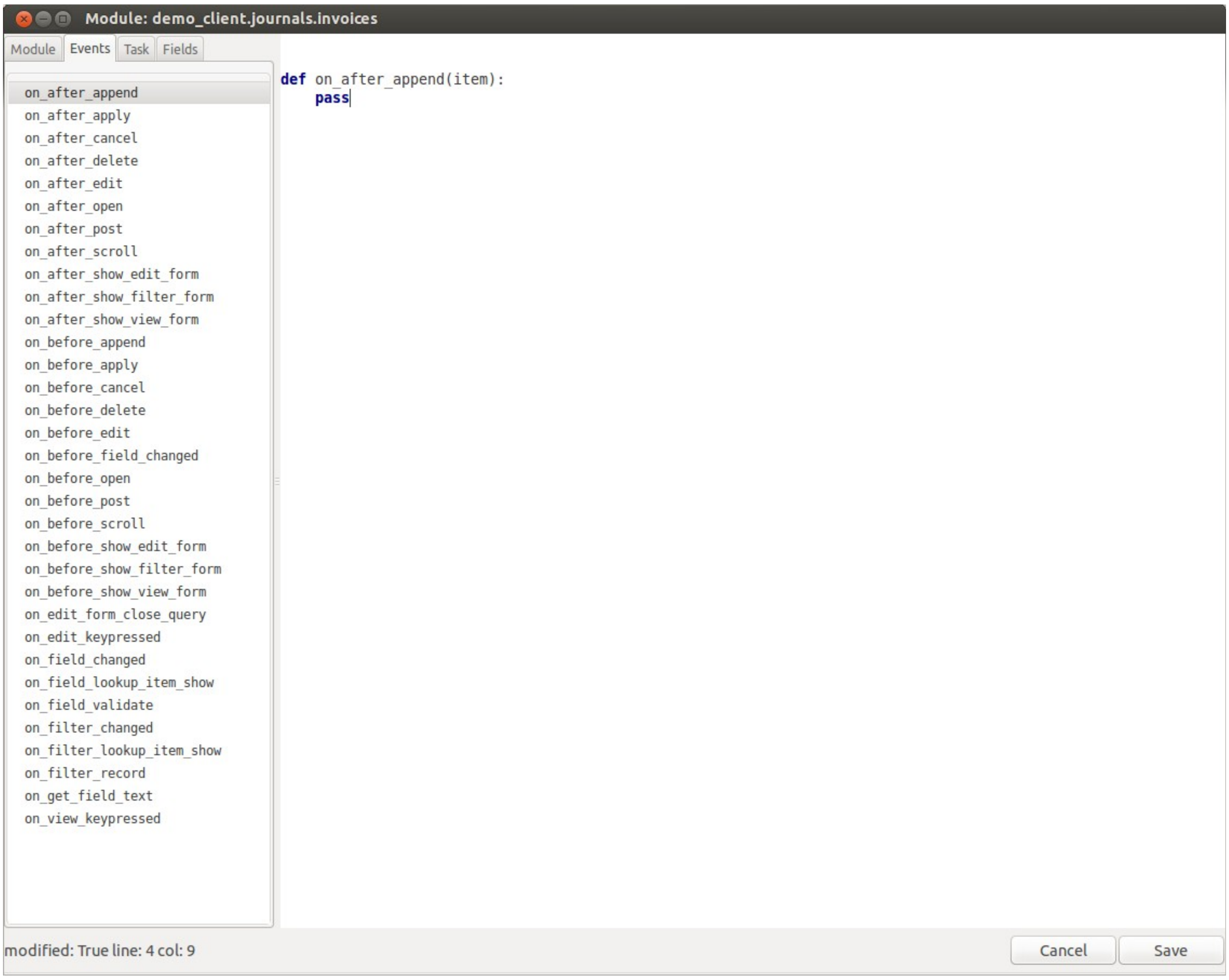
4 Jam.py programming.

In the previous chapter we have created all the necessary data structures. Now, in order to finish a project, we need to understand how to program in jam.py. Now, in order to illustrate the basic principles of programming in jam.py, we define an event handler on_after_append Invoices journal. This event fires immediately after adding a new record to the journal. To do so click on Journals node and select Invoices record. In the right pane of Administrator window on the top there are three buttons Client module, WebClient module, Server module. Click on the Client module button. Event Editor of the journal Invoices will appear.

4.1 *Event Editor .*

In the event editor to the right 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 generated wrapper for the event (see. Figure) .
- **Task** - the task tree, double-click on the node to print it's name under the cursor.
- **Fields** - the field list of the current item, double-click on one of the fields to print it's field_name under the cursor.



```
def on_after_append(item):
    pass
```

modified: True line: 4 col: 9

Let's select the Events tab and double-click on_after_append event. In the editor on_after_append function will be created. Note that all events in jam.py start with on_, and a parameter of the event is an object that generated this event. Now we'll write the body of the function:

```
import datetime

def on_after_append(item):

    item.invoicedate.value = datetime.datetime.now()
```

This code means that immediately after adding a new record in the desktop or client desktop application, the value of the invoicedate field will be equal to the current date. Let's save this code by pressing **Save**.

Press the WebClient Module and create the same handler in JavaScript for web interface application (in browser):

```
function on_after_append(item) {

    item.invoicedate.value = new Date();

}
```

Finally we'll list Event Editor shortcuts:

- Ctrl+S — save,
- Ctrl+F — find,

- Ctrl+H — find and replace,
- Ctrl+L — go to the line,
- Ctrl+I — indent selected lines,
- Ctrl+U — unindented selected lines,
- Ctrl+E — comment/uncomment selected lines,

4.2 Task tree.

When the server is started and receives first request from the client it builds from the metadata stored in admin.sqlite a task tree. After that it sends data to the client which in turn builds it's own task tree. All items of this trees have common ancestor AbstractItem 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 — one of the following values "task", "catalogs", "journals", "tables", "reports", "catalog", "journal", "table", "report", "detail",
- **item_name** — the name of the item that will be used in programming code to get access to the item object,
- **item_caption** — is the item name that appears to users,

and methods:

- **find(name)** — looks among immediate children for an item with item_name that equals name parameter and returns it if it's found, otherwise return None for python or undefined for JavaScript
- **item_by_ID(ID)** - looks among all its children for an item with ID that equals ID parameter and returns it if it's found, otherwise return None for python or undefined for JavaScript

So the following code where task is the root of the project tree:

```
def print_item(item, ident):
    owner_name = None
    if item.owner:
        owner_name = item.owner.item_name
    print '%s %s - item_type: "%s", ID: %s, item_caption: "%s", owner: %s' % \
    (3 * ident * ' ', item.item_name, item.item_type, item.ID, item.item_caption, owner_name)
```

```
print_item(task, 0)
for group in task.items:
    print_item(group, 1)
    for item in group.items:
        print_item(item, 2)
        for detail in item.items:
            print_item(detail, 3)
```

will print:

```
demo - item_type: "task", ID: 5, item_caption: "Demo", owner: None
    catalogs - item_type: "catalogs", ID: 6, item_caption: "Catalogs", owner: demo
       customers - item_type: "catalog", ID: 10, item_caption: "Customers", owner: catalogs
       artists - item_type: "catalog", ID: 11, item_caption: "Artists", owner: catalogs
       albums - item_type: "catalog", ID: 12, item_caption: "Albums", owner: catalogs
       genres - item_type: "catalog", ID: 13, item_caption: "Genres", owner: catalogs
       media_types - item_type: "catalog", ID: 14, item_caption: "MediaTypes", owner: catalogs
       tracks - item_type: "catalog", ID: 15, item_caption: "Tracks", owner: catalogs
    journals - item_type: "journals", ID: 7, item_caption: "Journals", owner: demo
       invoices - item_type: "journal", ID: 16, item_caption: "Invoices", owner: journals
          invoice_table - item_type: "detail", ID: 18, item_caption: "InvoiceTable", owner: invoices
    tables - item_type: "tables", ID: 8, item_caption: "Tables", owner: demo
       invoice_table - item_type: "table", ID: 17, item_caption: "InvoiceTable", owner: tables
    reports - item_type: "reports", ID: 9, item_caption: "Reports", owner: demo
       invoice - item_type: "report", ID: 19, item_caption: "Print invoice", owner: reports
       purchases_report - item_type: "report", ID: 20, item_caption: "Customer purchases ", owner: reports
       customers_report - item_type: "report", ID: 22, item_caption: "Customer list", owner: reports
```

In addition every item is an attribute of its owner and all catalogs, journals and tables are attributes of the task. So:

```
albums = task.catalogs.albums
print_item(task.journals.invoices.invoice_table)
print_item(task.invoices.invoice_table)
print_item(task.invoice_table)
print_item(albums.task.invoices)
```

will print:

```
invoice_table - item_type: "detail", ID: 18, item_caption: "InvoiceTable", owner: invoices
invoice_table - item_type: "detail", ID: 18, item_caption: "InvoiceTable", owner: invoices
invoice_table - item_type: "table", ID: 17, item_caption: "InvoiceTable", owner: tables
invoices - item_type: "journal", ID: 16, item_caption: "Invoices", owner: journals
```

4.3 Data programming.

All catalogs, journals and tables as well as their detail items (items with item_type of "catalog", "journal", "table", "detail") have access to the underlying tables from the project database.

As an example, the following function will print the names of clients:

```
def print_customers(customers):

    customers.open()

    for c in customers:

        print c.fisrtname.value, c.lastname.value
```

The function print_customers gets a customers item as a parameter. Then as a result of the open method execution a SQL query is generated and executed on the server and resulting record list is returned to the item customers. After that a loop through all the records is performed and for each record the name and surname of the client is printed. This function will work both on the client and the server.

For the web client, this functionality is implemented as follows:

```
function print_customers(customers) {

    customers.open();

    customers.each(function(c) {

        console.log(c.firstname.value, c.lastname.value);

    })

}
```

4.3.1 Fields.

All items, working with database data have a fields attribute - list of field objects, which are used to reprisent fields in item records. Every field have the following attributes :

- **ID** — unique field ID in the framework,
- **owner** — an item that owns this field,
- **field_name** - the name of the field that will be used in programming code to get access to the field object,
- **field_caption** - is the field name that appears to users,
- **field_type** - type of the item — one of the following values: "text", "integer", "float", 'currency', "date", "datetime", "boolean", "blob",
- **field_size** — a size of the field with type "text",
- **required** — should have a value,
- **read_only —** can't be changed in the interface controls,
- **lookup_item** — for lookup fields, that store record id of another item, reference to this item.
- **lookup_field** — field name in lookup item.

- **master_field** - reference to master field.

To get access the data fields have the following properties:

- value — this property allows to get or set field value of the current record, the value is converted to the type of the field. So for fields of type integer, float и currency, if value for this field in database table record is NULL, value of this property is 0. To get unconverted value use property raw_value,
- text - the text value of the field,
- lookup_value - for lookup fields, the field value property is id value of the record in the lookup item, lookup_value is the value of lookup_field in this record,
- lookup_text - the text value of the lookup_value,
- display_text – if field owner have an on_get_field_text event handler and its result value is not None (undefined), then this property value is the result value. Otherwise for lookup fields it's value is the lookup_text property value and for other fields it is text property value with regard of project locale parameters.

```
def print_field_data(field):
    print '%s: field_type: "%s"' % (field.field_name, field.field_type)
    print '            value: %s, value type: %s' % (field.value, type(field.value))
    print '             text: "%s"' % field.text
    print '     lookup_value: %s' % field.lookup_value
    print '      lookup_text: "%s"' % field.lookup_text
    print '     display_text: "%s"' % field.display_text


print_field_data(invoices.id)
print_field_data(invoices.deleted)
print_field_data(invoices.invoicedate)
print_field_data(invoices.customer)
print_field_data(invoices.firstname)
print_field_data(invoices.taxrate)
print_field_data(invoices.total)
```

```
id: field_type: "integer"
                value: 411, value type: <type 'int'>
                 text: "411"
         lookup_value: None
          lookup_text: ""
         display_text: "411"
deleted: field_type: "boolean"
                value: False, value type: <type 'bool'>
```

```
                 text: "No"
         lookup_value: None
          lookup_text: ""
         display_text: "No"
invoicedate: field_type: "date"
                value: 2014-12-14, value type: <type 'datetime.date'>
                 text: "12/14/2014"
         lookup_value: None
          lookup_text: ""
         display_text: "12/14/2014"
customer: field_type: "integer"
                value: 44, value type: <type 'int'>
                 text: "44"
         lookup_value: Hämäläinen
          lookup_text: "Hämäläinen"
         display_text: "Terhi Hämäläinen"
firstname: field_type: "integer"
                value: 44, value type: <type 'int'>
                 text: "44"
         lookup_value: Terhi
          lookup_text: "Terhi"
         display_text: "Terhi"
taxrate: field_type: "float"
                value: 5, value type: <type 'int'>
                 text: "5"
         lookup_value: None
          lookup_text: ""
         display_text: "5"
total: field_type: "currency"
                value: 14.56, value type: <type 'float'>
                 text: "14.56"
         lookup_value: None
          lookup_text: ""
         display_text: "$14.56"
```

In the example above the lookup_text value of the field customer is 'Hämäläinen', while the display_text value - 'Terhi Hämäläinenand'. This is because the journal Invoices has an event handler on_get_field_text:

Python:

```
def on_get_field_text(field):
    if field.field_name == 'customer':
        return field.owner.firstname.lookup_text + ' ' + field.lookup_text
```

JavaScript:

```
function on_get_field_text(field) {
    if (field.field_name === 'customer') {
        return field.owner.firstname.lookup_text + ' ' + field.lookup_text;
    }
}
```

4.3.2 Filters.

Previously, we have created filters for journal Invoices. We now show how use them in programming code.

Each item has an attribute filters - list of filter objects that were created in the Administrator. Each filter has the following attributes:

- **owner** – an item that owners this filter,
- **filter_name** — the name of the filter,
- **filter_caption** - the name of the filter used in the visual representation in the client application,
- **filter_type —** type of the filter,
- **visible** — if this attribute value is true, a visual representation of this filter will be created when creating filters in the client application by a create_filter_entries method of the owner item.
- **value** — filter value,

By filter_name we can get access to the filter object as well as by filter_by_name method. For example:

- Python (client and server):

  ```
  now = datetime.datetime.now() - datetime.timedelta(days=7)
  item.filters.invoicedate1.value = now
  ```

  or

  ```
  item.filter_by_name('invoicedate1').value = now
  ```

- JavaScript:

  ```
  var now = new Date();
  now.setDate(now.getDate() - 7);
  item.filters.invoicedate1.value = now;
  ```

or

```
item.filter_by_name('invoicedate1').value = now;
```

In the above example invoicedate1 filter value of the item has been changed.

4.3.3 Getting data records.

To obtain the data, in addition to direct SQL query to the database, which will be described in chapter 'Programming the server', use an open method:

- Python (client and server):

  ```
  def open(self, expanded=None, fields=None, where=None, order_by=None, open_empty=False,
  params=None, offset=None):
  ```

- JavaScript: `.open(options, callback)`

For JavaScript order of parameters does not matter. Options parameter is an object (a dictionary) whose attributes match the parameters of the python open function, with the same default values.

If the method is called on the client, it send a request to the server with the parameters of the call.

On the server, based on the parameters, SQL query is generated and executed, and the result of this query - the list of records — is returned to the open method. If an on_select event if defined on the server for the item, then the parameters of the request can be intercepted and independently generated list of records can be returned (see. Programming the server).

All requests of the python client run synchronously. For the JavaScript client all depends on the callback parameter. If this is not a parameter-function in the open function call, the request is executed synchronously, otherwise the request is executed asynchronously and after that, as records are received, this function will be executed.

The fields parameter is a list of field names and sets the fields for which the data will be obtained. If not specified, the data will be obtained for all fields.

When the expanded parameter is set to true (the default), there are lookup_value as well as values for lookup fields in the resulting records . Otherwise, lookup values are not returned.

This where parameter determines the filtering of records in sql query on the server. If this parameter is not specified, by default, the records are filtered according to the values stored in the filters (if any) described above. Where the parameter is a dictionary whose keys are the names of the fields that are followed, after double underscore, by a filtering symbol. In the framework, the following symbols are defined to filter field values:

- 'eq' — equal,
- 'ne' — not equal,
- 'lt' — less than,
- 'le' - less than or equal,

1. 'gt' — greater that,

- 'ge' - greater that or equal,
- 'in' — SQL operator IN is applied to the field value,
- 'not_in' - SQL operator NOT IN,
- 'range' - SQL operator BETWEEN,
- 'isnull' - SQL operator IS NULL,
- 'exact' - exact equality,
- 'contains' - field value contains,
- 'startwith' - field value starts with,
- 'endwith' - field value ends with,

For 'eq' filtering symbol '__eq' can be omitted. For example {'id': 100} is equivalent to {'id__eq': 100}.

For example:

- Python (client and server):

```
where = {
    'customer': report.customer.value,
    'invoicedate__ge': report.invoicedate1.value,
    'invoicedate__le': report.invoicedate2.value
    }
invoices.open(where=where)
```

- JavaScript:

```
where = {
    customer: report.customer.value,
    invoicedate__ge: report.invoicedate1.value,
    invoicedate__le: report.invoicedate2.value
    };
invoices.open({where: where});
```

Calling method set_where before performing the open method is similar to specifying the parameter where:

- Python (client and server): `def set_where(self, **fields):`

```
invoices.set_where(customer=report.customer.value,
    invoicedate__ge=report.invoicedate1.value,
    invoicedate__le=report.invoicedate2.value)
invoices.open();
```

- JavaScript: `.set_where(fieldsDict)`

```
invoices.set_where({
    customer: report.customer.value,
    invoicedate__ge: report.invoicedate1.value,
    invoicedate__le: report.invoicedate2.value
});
invoices.open();
```

After execution of an open method a filtering defined by a set_where method is reset.

If the order parameter is not specified, then the returned records are sorted according to the order specified in the Administrator (button Order). Order 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:

- Python (client and server):

```
customers.open(order_by=['-country', 'lastname'])
```

- JavaScript:

```
customers.open({order_by=['-country', 'lastname']});
```

Calling method set_order_by before performing the open method is similar to specifying the parameter order.

For example:

- Python (client and server): `def set_order_by(self, *fields)`

```
customers.set_order_by('-country', 'lastname')
customers.open()
```

- JavaScript: `.set_order_by(fieldList)`

```
customers.set_order_by(['-country', 'lastname']);
customers.open();
```

After execution of an open method a sorting order defined by a set_order_by method is reset.

After successful execution of this method the active property is set to True.

The result returned by the open method depends on the value of the auto_loading attribute . If this attribute value is set to True, the open method returns not all records but just the first, the number of which corresponds to the value of the limit attribute, starting with the record specified by the offset parameter.

This mechanism is used for automatic record loading when viewing data in a grid component.

Use a record_count method to get the total number of records that have currently been fetched.

This method initializes all the structure necessary for the data processing and must be performed before calling any other methods that are dealing with the data.

4.3.4 Navigating through records.

After receiving the data, a cursor of the item (pointer to the current record) is set to the first record. To change the position of the cursor use the following methods:

- first - the cursor jumps to the first record
- last - the cursor moves to the last record
- next - the cursor moves to the next record
- prior - the cursor moves to the previous record

In addition there are bof and eof methods:

- eof - returns true in the following cases:
  - record list is empty,
  - was called an item last method,
  - last calling of a next method failed because the current record is already the last record.
- bof - returns true in the following cases:
  - record list is empty,
  - was called an item first method,
  - last calling of a prior method failed because the current record is already the first record.

Use rec_no property to get or set the value of the current record number.
For example the following code saves the current cursor position prints a list of customers, then the cursor is placed in the original position.

- Python (client and server):

```
rec = customers.rec_no
customers.first()
while not customers.eof():
    print customers.lastname.display_text
    customers.next()
 customers.rec_no = rec
```

- JavaScript:

```
var rec = customers.rec_no;
customers.first();
while (!customers.eof()) {
    console.log(customers.lastname.display_text);
    customers.next();
}
customers.rec_no = rec;
```

There is a short-form of the record loop:

- Python (client and server):

```
for c in customers:
    pass
```

is equivalent to

```
customers.first()
while not customers.eof():
    customers.next()
```

- JavaScript:

```
customers.each(function(c) {
})
```

is equivalent to

```
customers.first();
while (!customers.eof()) {
    customers.next()
}
```

C and customers are the same in the above example, they are pointers to the same object. To exit the JavaScript short-form loop return false from callback function.

When the position of the cursor changes, then before change an on_before_scroll event handler is fired (if defined), after the change - an on_after_scroll event handler.

4.3.5 Changing the data.

After open method is executed the item is in a browse mode.

To change field values of the current record, item state should be changed to edit state by calling edit method. After changing field values save changes by calling post method().

```
invoices.edit()
invoices.invoicedate.value = datetime.datetime.now()
invoices.post()
```

Post method returns item into browse state again.

Addition is performed similarly, except that instead of the edit method you must call an append (to add a record to the end of the list) or insert (to insert as a first record of the list) method to transfer item in insert state:

```
invoices.append()
invoices.invoicedate.value = datetime.datetime.now()
invoices.post()
```

When there is no need to save changes then instead of the post call cancel method:

```
invoices.append()
invoices.invoicedate.value = datetime.datetime.now()
invoices.cancel()
```

Cancel method cancel changes and returns item to the browse state again.

To delete record call delete method:

```
invoices.delete()
```

All such modifications are made on the current set of records and do not affect the values stored in the database. If a log_changes attribute of the item is True (the default), then a log accumulating all changes is supported and they can be stored in the database by calling an apply method.

In the following example all records are deleted, after which they are removed from the database.

```
item.first();

while not item.eof():

    item.delete()

item.apply()
```

Before each of these methods is executed an event handler on_before + method name (on_before_apply for example) is fired (if defined), after - on_after + method name event handler.

To cancel execution of a method return False from on_before event handler or raise exception in it.

4.3.6 Working with details.

If an item has detail items and its details_active attribute value is True, then when skipping to another record, record lists of its details are automatically updated (they are reopened). Otherwise you should reopen them yourself.

```
invoices.details_active = False

for inv in invoices:

       inv.invoice_table.open()
```

Default value of details_active is False.

Detail items have an attribute disabled with a default value False. When value of disabled is True, then their record list is not updated.

To modify detail item, the state of its master item should be changed to edit state.

```
invoices.edit()

for t in invoices.invoice_table:

       t.edit()

       t.date.value = invoices.date.value

       t.post()

invoices.post()

invoices.apply()
```

Master item is responsible for storing changes of its details on the server by calling its apply method.

4.4 Client-side programming.

4.4.1 Main form.

When we run a python client, after receiving data from a server and a task tree initialization, the application main form is created. And before this form will be displayed the client generates an on_before_show_main_form event.

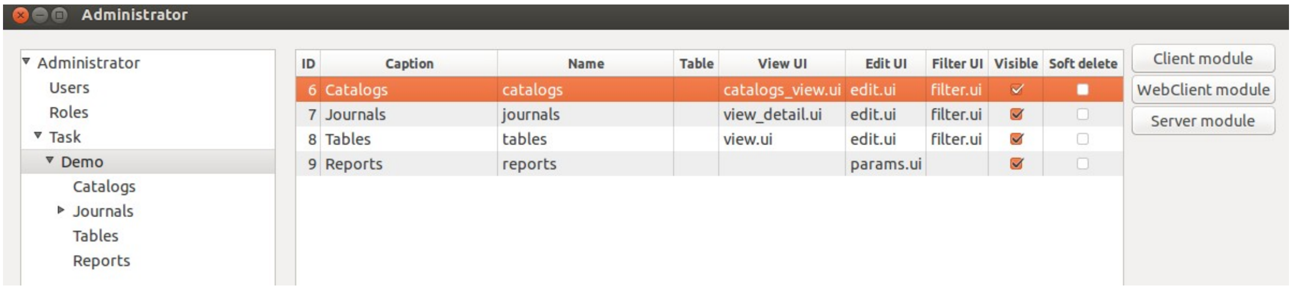
Similarly, a browser, after DOM is loaded and task tree is initialized, also generates an on_before_show_main_form event.

4.4.2 Forms.

One of the main concepts of the framework client-side programming is the concept of form. The forms are based on templates. For each item working with data form templates for viewing and editing of the data and template for filter form can be set. For reports — report parameters form template.

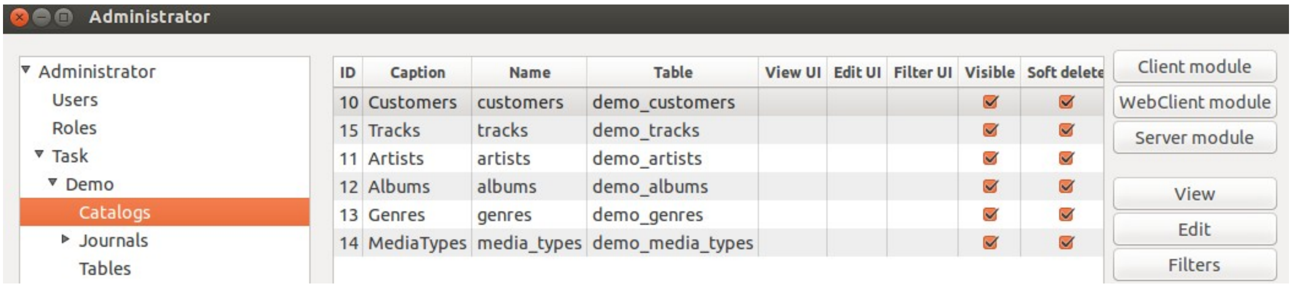
4.4.2.1 Client forms in python and pygtk.

Form templates are located in the **ui** folder of the project directory. Form templates are interface files created in Glade — a user interface designer for GTK + and GNOME. In the figure below the columns View UI, Edit UI and Filter Ui contain file names of view, edit, and filter form templates .



ID	Caption	Name	Table	View UI	Edit UI	Filter UI	Visible	Soft delete
6	Catalogs	catalogs		catalogs_view.ui	edit.ui	filter.ui	☑	□
7	Journals	journals		view_detail.ui	edit.ui	filter.ui	☑	□
8	Tables	tables		view.ui	edit.ui	filter.ui	☑	□
9	Reports	reports			params.ui		☑	□

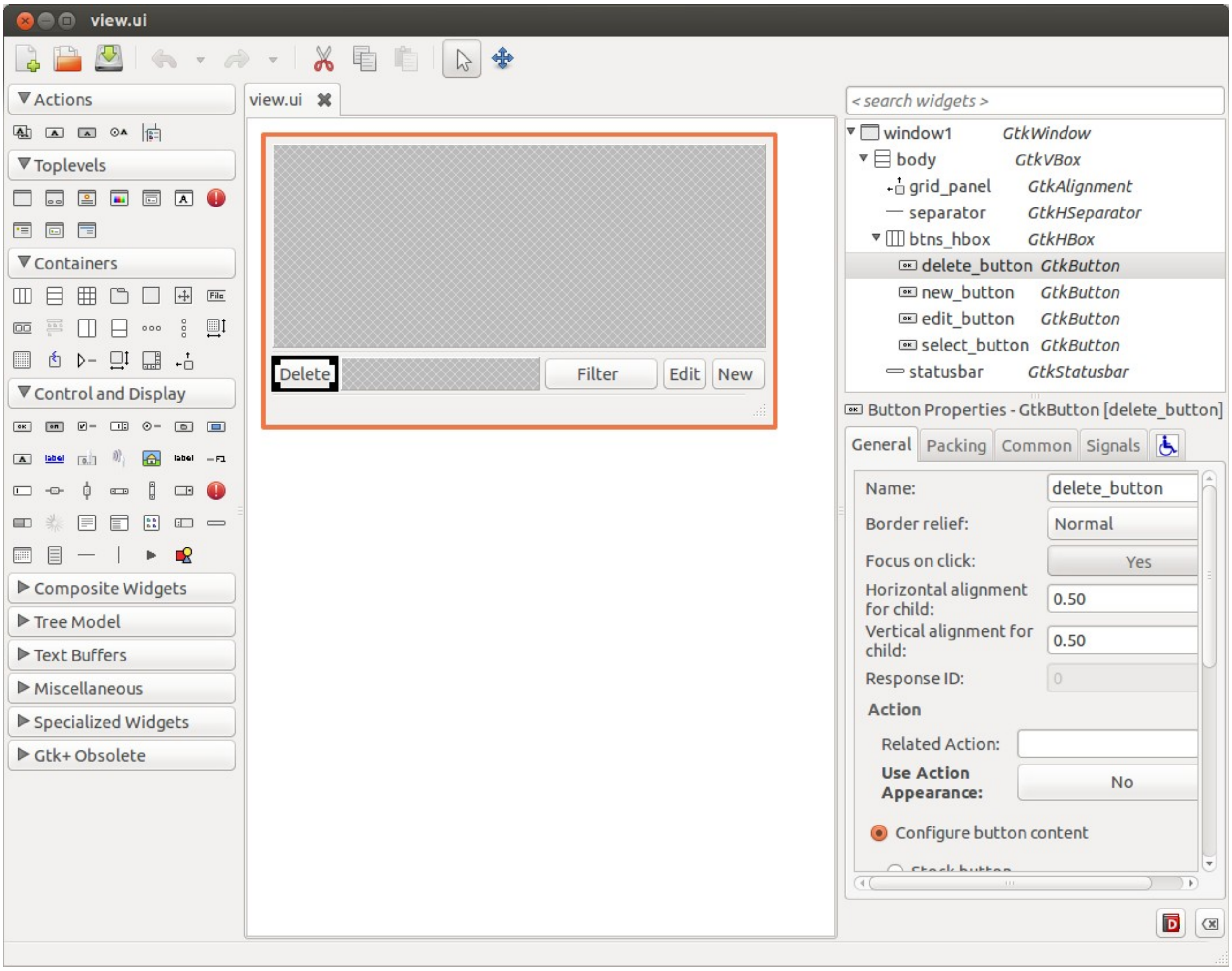
When a form template for an item is not set, the framework looks for a form template of its owner. If you select Catalogs node in the task tree, you'll see that form templates for catalogs are not set. So all catalogs have common form templates that are defined in their owner — Catalogs group item.



ID	Caption	Name	Table	View UI	Edit UI	Filter UI	Visible	Soft delete
10	Customers	customers	demo_customers				☑	☑
15	Tracks	tracks	demo_tracks				☑	☑
11	Artists	artists	demo_artists				☑	☑
12	Albums	albums	demo_albums				☑	☑
13	Genres	genres	demo_genres				☑	☑
14	MediaTypes	media_types	demo_media_types				☑	☑

If for some catalog there is a need for a template that is different from others, it is necessary to create a interface template in the Glade editor, save it to the file in the ui folder, and specify this file name as this catalog form template.

The figure below shows view.ui form template in the Glade editor. Please note that the name of the main window of the template should be 'window1'.



window1 GtkWindow
body GtkVBox
grid_panel GtkAlignment
separator GtkHSeparator
btns_hbox GtkHBox
delete_button GtkButton
new_button GtkButton
edit_button GtkButton
select_button GtkButton
statusbar GtkStatusbar

Name: delete_button
Border relief: Normal
Focus on click: Yes
Horizontal alignment for child: 0.50
Vertical alignment for child: 0.50
Response ID: 0
Action
Related Action:
Use Action Appearance: No
● Configure button content
○ Stock button

After view, edit or filter form is created to get access to widgets on these forms, you can use attributes of the view_form, edit_form and filter_form objects respectively.

For example:

```
item.view_form.delete_button.set_visible(False)
```

makes the button with the name 'delete_button' on the view form of the item invisible.

The same way the task.main_form is the object for the main form of the task item and report.params_form is an object for the form to specify parameter values of the report item.

Besides, these objects are wrappers over gtk.Window. Use window attribute of these objects to get access to the gtk.Window. The code below maximizes the view form:

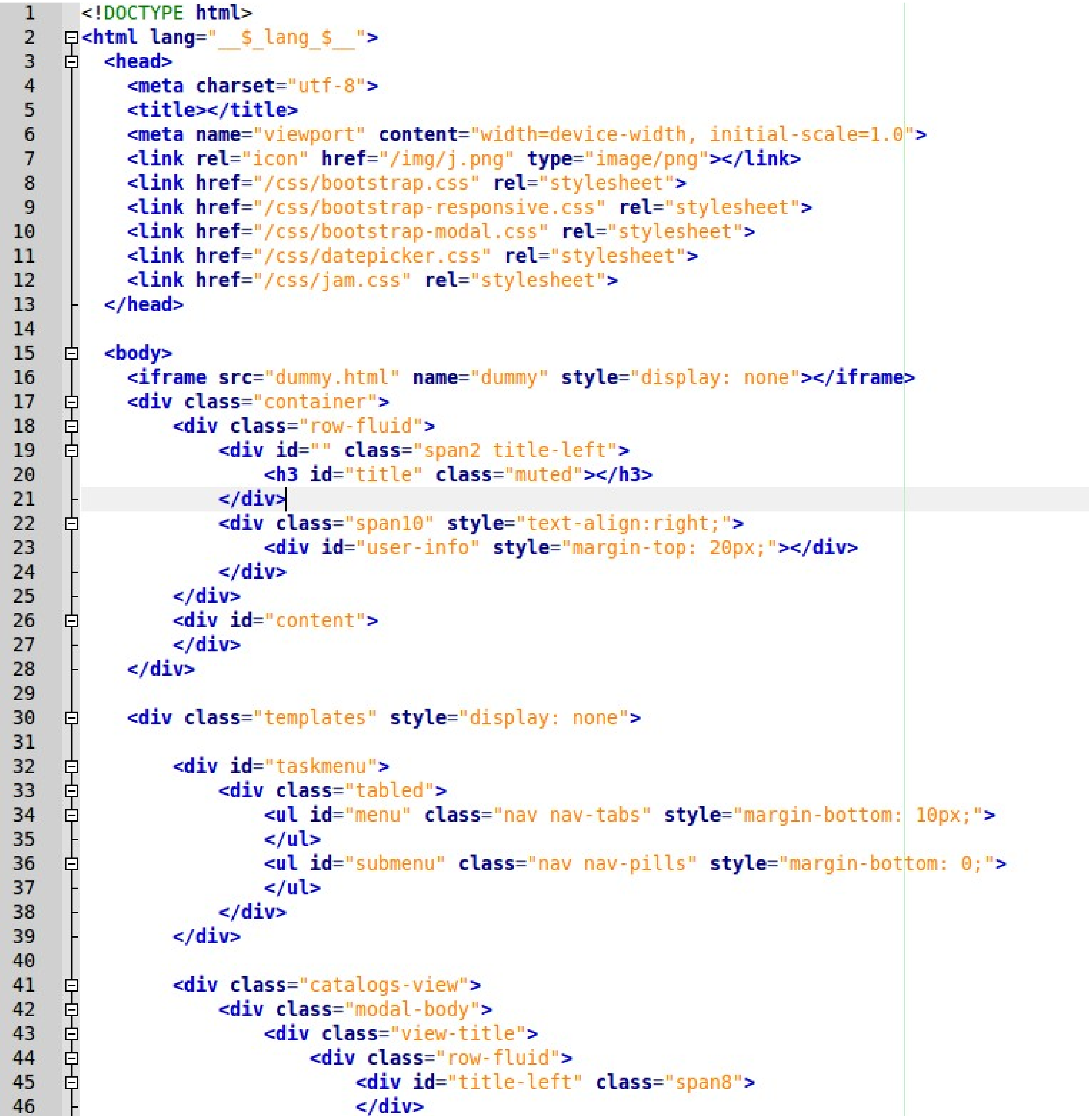
```
item.view_form.window.maximize()
```

Furthermore builder attribute of form objects is gtk.Builder object. So you can connect signal handlers defined in ui-template:

```
dic = {
        "on_ok_button_clicked" : item.apply_record,
        "on_cancel_button_clicked" : item.cancel_edit,
 }
item.edit_form.builder.connect_signals(dic)
```

4.4.2.2 Client forms in web interface.

Project templates are located in div with the **template** class inside the **body** tag in the file index.html of the project directory.



```
<!DOCTYPE html>
<html lang="__$_lang_$__">
  <head>
    <meta charset="utf-8">
    <title></title>
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <link rel="icon" href="/img/j.png" type="image/png"></link>
    <link href="/css/bootstrap.css" rel="stylesheet">
    <link href="/css/bootstrap-responsive.css" rel="stylesheet">
    <link href="/css/bootstrap-modal.css" rel="stylesheet">
    <link href="/css/datepicker.css" rel="stylesheet">
    <link href="/css/jam.css" rel="stylesheet">
  </head>

  <body>
    <iframe src="dummy.html" name="dummy" style="display: none"></iframe>
    <div class="container">
        <div class="row-fluid">
            <div id="" class="span2 title-left">
                <h3 id="title" class="muted"></h3>
            </div>
            <div class="span10" style="text-align:right;">
                <div id="user-info" style="margin-top: 20px;"></div>
            </div>
        </div>
        <div id="content">
        </div>
    </div>

    <div class="templates" style="display: none">

        <div id="taskmenu">
            <div class="tabled">
                <ul id="menu" class="nav nav-tabs" style="margin-bottom: 10px;">
                </ul>
                <ul id="submenu" class="nav nav-pills" style="margin-bottom: 0;">
                </ul>
            </div>
        </div>

        <div class="catalogs-view">
            <div class="modal-body">
                <div class="view-title">
                    <div class="row-fluid">
                        <div id="title-left" class="span8">
                        </div>
```

After task initialization on the client, this div is cut out of the page, but you can get access to its content through a templates attribute of the task item , which is a JQuery object storing this div content. For example

```
$("#content").append(task.templates.find("#mainmenu"));
```

will append tag with id mainmenu to from templates div to the tag with id content.

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 name of the item and suffix — the form type: view, edit, filter, params. For example:

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

	...

</div>
```

is an edit form template to the invoices item. This div have to contain html representation of the item data.

For a detail item before its name should be the name of its owner separated by a hyphen:

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

	...

</div>
```

If an item doesn't have a form template then its owner form template will be used. So the template

```
<div class="catalogs-edit">

	...

</div>
```

will be used as edit form template to create catalogs that do not have its own edit form templates.

After view, edit or filter form is created to get access to objects on these forms, you can use the view_form, edit_form and filter_form attributes of the item object respectively. They are JQuery objects. For example:

```
item.view_form.find(“#delete-btn”).hide();
```

makes the button with the id 'delete_button' on the view form of the item invisible.

4.4.3 Client Methods.

4.4.3.1 View method.

In the on_before_show_main_form events handler of the Demo application main menu is created.

And when we click on menu items of the main menu the view method of a corresponding item is executed:

- Python: `def view(self, widget):`
- JavaScript: `.view(container)`

The view form will be created in a modal window, except when container parameter (for JavaScript client) is specified. In this case it will be added to the container (container is a JQuery object).

During a view method execution

- client looks for a view template, on which a view_form object is created
- if for a task an on_before_show_view_form event handler is defined, this handler is executed to which this item is passed as a parameter
- if for an items's owner an on_before_show_view_form event handler is defined, this handler is executed to which this item is passed as a parameter
- if defined, an on_before_show_view_form event handler of an item is executed to which this item is passed as a parameter
- view form visually displayed on the screen
- if for a task an on_after_show_view_form event handler is defined, this handler is executed to which this item is passed as a parameter
- if for an items's owner an on_after_show_view_form event handler is defined, this handler is executed to which this item is passed as a parameter
- if defined, an on_after_show_view_form event handler of an item is executed to which this item is passed as a parameter

Below is the code of the view method for the client in python:

```
def view(self, widget):
    self.view_form = self.create_view_form(widget)
    if self.task.on_before_show_view_form:
        self.task.on_before_show_view_form(self)
    if self.owner.on_before_show_view_form:
        self.owner.on_before_show_view_form(self)
    if self.on_before_show_view_form:
        self.on_before_show_view_form(self)
    if self.view_form and self.view_form.window:
        self.view_form.window.connect("key-press-event", self.view_keypressed)
        self.view_form.window.connect('delete-event', self.check_view)
    self.view_form.show()
    if self.task.on_after_show_view_form:
        self.task.on_after_show_view_form(self)
    if self.owner.on_after_show_view_form:
        self.owner.on_after_show_view_form(self)
    if self.on_after_show_view_form:
        self.on_after_show_view_form(self)
    if self.view_form.window:
        self.view_form.window.connect("destroy", self.do_on_destroy_view_form)
    return self.view_form
```

4.4.3.2 Append_record, insert_record, edit_record methods.

In the on_before_show_view_form event handler of the Demo application insert_record and edit_record methods are connected to the New and Edit buttons:

- Python: `item.view_form.new_button.connect('clicked', item.insert_record)`
- JavaScript: `item.view_form.find("#new-btn").click(function() {item.insert_record();});`

When append record, insert record or edit record method is executed, it first fires append, insert or edit method, respectively, that puts item into insert or edit mode, after which the create_edit_form method is executed, which creates an item edit form.

During a create_edit_form method execution

- client looks for an edit template, on which a edit_form object is created
- if for a task an on_before_show_edit_form event handler is defined, this handler is executed to which this item is passed as a parameter
- if for an items's owner an on_before_show_edit_form event handler is defined, this handler is executed to which this item is passed as a parameter
- if defined, an on_before_show_edit_form event handler of an item is executed to which this item is passed as a parameter
- view form visually displayed on the screen
- if for a task an on_after_show_edit_form event handler is defined, this handler is executed to which this item is passed as a parameter
- if for an items's owner an on_after_show_edit_form event handler is defined, this handler is executed to which this item is passed as a parameter
- if defined, an on_after_show_edit_form event handler of an item is executed to which this item is passed as a parameter

4.4.3.3 Post_record u apply_record methods.

To save the results of the record editing use the apply_record method. If the record has been modified the post method is executed after which it is stored on the server as a result of the apply method, otherwise the cancel method is called. And at the end the edit form window closes.

The post_record method performs the same actions except for storing data on the server.

4.4.3.4 Delete_record method.

When executing this method, if an item read_only property is not false, and after user confirms the deletion, the delete method is executed, and then record is erased on the server by executing apply method.

4.4.3.5 Create_grid method.

Create grid method allows you to create a table view of item records:

- Python:

```
def create_grid(self, container, fields=None, dblclick_edit=True, headers=True, lines=False,
       border_width=6, striped=True, multi_select=False,multi_select_get_selected=None,
       multi_select_set_selected=None):
```

- JavaScript:

```
create_grid: function(container, options) {

   var default_options = {

         height: 480,

      fields: [],

      column_width: {},

      row_count: 0,

      word_wrap: false,

      title_word_wrap: false,

      expand_selected_row: 0,

      multi_select: false,

      multi_select_title: '',

      multi_select_colum_width: undefined,

      multi_select_get_selected: undefined,

      multi_select_set_selected: undefined,

               multi_select_select_all: undefined,

      tabindex: 0,

      striped: true,

      dblclick_edit: true,

      on_dblclick: undefined,

      on_pagecount_update: undefined,

      editable: false,

      always_show_editor: false,

      editable_fields: undefined,

      selected_field: undefined,

      append_on_lastrow_keydown: false,

      sortable: false,

      sort_fields: undefined,

      row_callback: undefined,

      title_callback: undefined,

      show_footer: undefined

    };
```

Below we describe the basic parameters for JavaScript grid:

- container - a JQuery object that will contain (be parent of) a table,
- fields - a list of field names, if specified, the grid will create a column for each field whose name is in this list, if not specified (the default) then view fields, specified in the Administrator (the «View» button), will be used,
- striped — the grid is striped if this value is true,
- dblclick_edit - if the value is true (the default), then double-click on a grid row activates editing of the correspondent record,
- on_dblclick - allows to specify the procedure that will be executed when user double-clicks the grid row,
- multi_select - if this parameter is set to true, a new leftmost column with check-boxes will be created to select records. So, if the function-parameter 'multi_select_get_selected' returns true for the record this record's check-box will be checked. When you click on the check-box the multi_select_set_selected function will be called with the state of the check-box as a parameter. If the function-parameter multi_select_select_all is specified than check-box will be created in the leftmost column of the grid title and this function will be called when user clicks on this check-box. In the example below for the tracks item the multi_select parameter is set to true, the dictionary selected_records is created, that will store information about the selected records, and that the function-parameters multi_select_get_selected and multi_select_set_selected will use:

```
function on_before_show_view_form(item) {
    var multi_select,
        multi_select_get_selected,
        multi_select_set_selected;

    item.auto_loading = true;
    if (item.item_name === "tracks") {
        item.selected_records = {};
        multi_select = true;
        multi_select_get_selected = function() {
            return item.selected_records[item.id.value]
        }
        multi_select_set_selected = function(value) {
            if (value) {
                item.selected_records[item.id.value] = 1;
            }
            else {
                delete item.selected_records[item.id.value];
            }
        }
    }
    item.view_grid = item.create_grid(item.view_form.find(".view-table"),
        {
            multi_select: multi_select,
            multi_select_get_selected: multi_select_get_selected,
            multi_select_set_selected: multi_select_set_selected,
        });
}
```

- editable - if this parameter is set to true, user can edit fields in the grid. If an editable_fields parameter is not specified (default), then it is possible to edit any field in the grid, otherwise only fields which names are listed in this parameter. When always_show_editor is true, then the editor is always present, otherwise (the default) to get the grid into the edit mode it is necessary to press the Enter key or if keypress_edit is true (the default), press any key. Use selected_field parameter to specify a field that will be selected when create is created. To save new values press Enter or move to another record. The new values is stored locally to save then in the server database, you should call the apply method.
- sortable - if this parameter is specified, it is possible to sort the item records by clicking on

the grid column header, when a sort_fields parameter is not specified (default), user can sort records on any field, otherwise, only on the fields whose names are listed in this parameter. Sorting is performed on the server.

- auto_fit_width - if this parameter is true, the grid tries to display all the columns without the use of a horizontal scroll bar, including when resizing columns.
- expand_selected_row - when the word_wrap parameter is set to true and expand_selected_row value greater than 0, then if the text of selected record field values does not fit in a grid columns, the selected row height is increased. Expand_selected_row value specifies the minimum height (number of lines) of the selected row.

Note that the behavior of the grid is determined by the 'auto_loading' attribute of the item. For the grid in python if this attribute value is set to true, the grid, when necessary, automatically loads records from the server in accordance with the value of the 'limit' attribute. The JavaScript grid, when auto_loading is true, creates a paginator and based on the specified parameters calculates the value of the limit attribute of the item. If auto_loading value is false, the grid displays all available records of the item.

This method returns a DBGrid object.

4.4.3.6 Create_entries method.

Create_entries method allows you to create visual controls for editing item fields:

- Python:

```
def create_entries(self, container, fields=None, col_count=1):
```

- JavaScript:

```
 create_entries: function(container, options) {
     var default_options = {
         fields: [],
         col_count: 1,
                  tabindex: undefined
     };
```

The following parameters are passed to the method:

- container - an object that will contain (be parent of) a visual controls, for web client it's JQuery object, for pygtk – a GTK widget,
- 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 (the default) then edit fields specified in the Administrator will be used (Edit button),
- col_count - the number of columns that will be created for visual controls, the default is 1. (In Demo application, in the invoice edit form, the col_count equals 2),
- tabindex - if tabindex is specified, it will the tabindex of the first visual control, tabindex of all subsequent controls will be increased by 1.

4.4.3.7 Interaction between data and visual controls.

By default, any data changes of an item are immediately displayed in visual controls of the client - tables, input, grids, entries and so on. But sometimes it is necessary to disable this connection. You can disable and enable these interactions by using the disable_controls and enable_controls methods respectively. To update visual controls use the update_controls method (in this case grids will be recreated). To learn about the state of visual controls use the controls_enabled and controls_disabled methods.

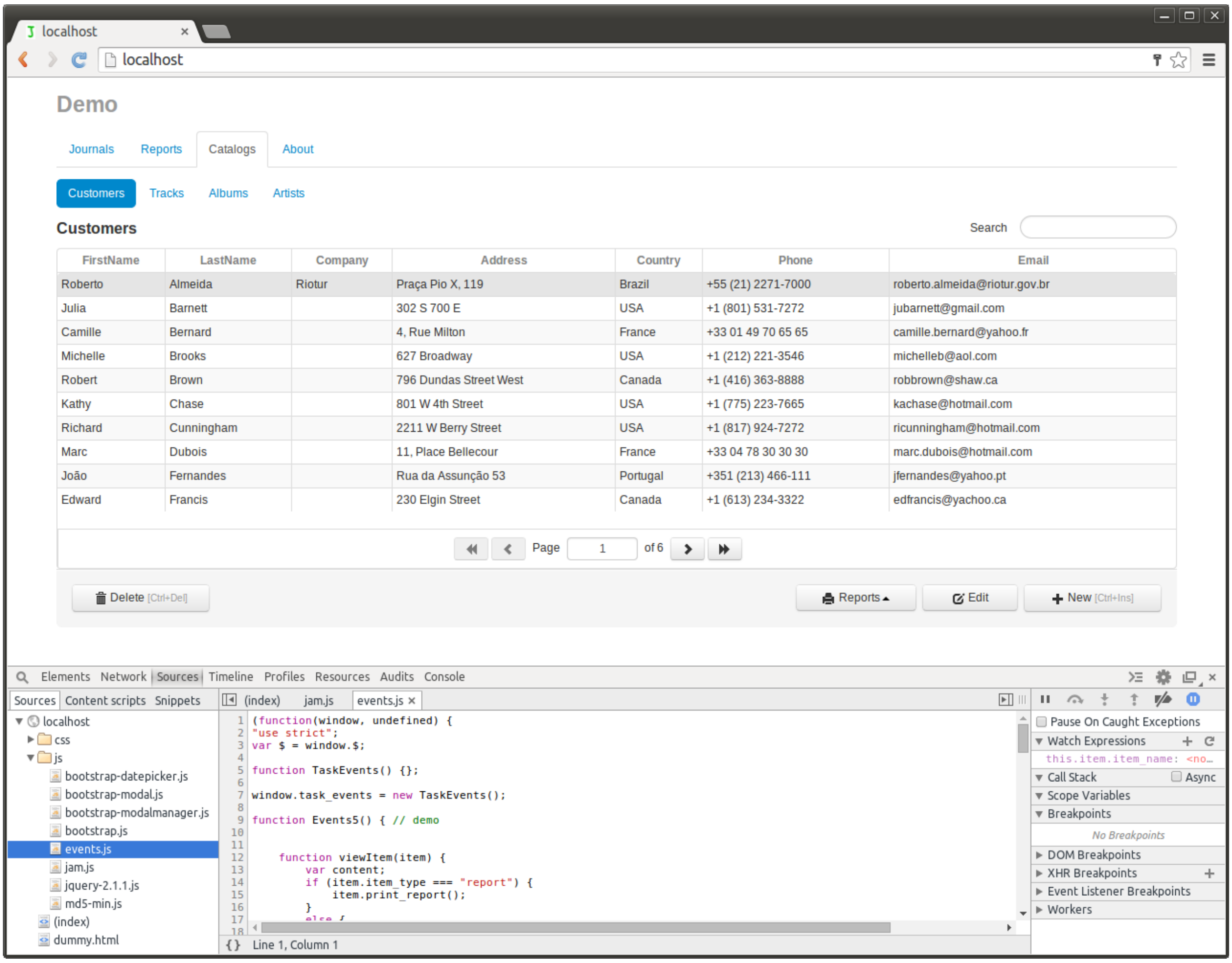
For Example:

```
subtotal = 0
tax = 0
total = 0
item.invoice_table.disable_controls()
rec = item.invoice_table.rec_no
try:
    for detail in item.invoice_table:
        detail.edit()
        calc_total(detail)
        detail.post()
        subtotal += detail.amount.value
        tax += detail.tax.value
        total += detail.total.value
finally:
    item.invoice_table.rec_no = rec
    item.invoice_table.enable_controls()
item.invoice_table.update_controls()
item.subtotal.value = subtotal
item.tax.value = tax
item.total.value = total
```

In the above code we save the current record number of the invoice_table detail item and disable its visual controls. Then we loop through all the records, recalculate their fields values and calculate total values of the invoice. After the loop we return the cursor to its original position, connect and update its visual controls.

4.4.3.8 Web client debugging.

After saving changes to the web client module, a framework, based on all web modules of the task, generates events.js file and saves it to the js folder of the project directory. This file contains, appropriately structured, all the events of the project.



Customers

FirstName	LastName	Company	Address	Country	Phone	Email
Roberto	Almeida	Riotur	Praça Pio X, 119	Brazil	+55 (21) 2271-7000	roberto.almeida@riotur.gov.br
Julia	Barnett		302 S 700 E	USA	+1 (801) 531-7272	jubarnett@gmail.com
Camille	Bernard		4, Rue Milton	France	+33 01 49 70 65 65	camille.bernard@yahoo.fr
Michelle	Brooks		627 Broadway	USA	+1 (212) 221-3546	michelleb@aol.com
Robert	Brown		796 Dundas Street West	Canada	+1 (416) 363-8888	robbrown@shaw.ca
Kathy	Chase		801 W 4th Street	USA	+1 (775) 223-7665	kachase@hotmail.com
Richard	Cunningham		2211 W Berry Street	USA	+1 (817) 924-7272	ricunningham@hotmail.com
Marc	Dubois		11, Place Bellecour	France	+33 04 78 30 30 30	marc.dubois@hotmail.com
João	Fernandes		Rua da Assunção 53	Portugal	+351 (213) 466-111	jfernandes@yahoo.pt
Edward	Francis		230 Elgin Street	Canada	+1 (613) 234-3322	edfrancis@yachoo.ca

Page 1 of 6

```
(function(window, undefined) {
"use strict";
var $ = window.$;

function TaskEvents() {};

window.task_events = new TaskEvents();

function Events5() { // demo


    function viewItem(item) {
        var content;
        if (item.item_type === "report") {
            item.print_report();
        }
        else {
```

`this.item.item_name: <no…`

Above is Demo project in the browser Chrome.

4.5 Sever side programming.

4.5.1 SQL queries.

When task is created on the server, it creates a pool of connections to the database using the multiprocessing module. This pool is accessed via a request queue. To run the sql query via a connection pool the task has the following methods:

- execute
- execute_select

To execute select queries use the execute_select method of the task:

```
def execute_select(self, sql):
```

where sql is SQL query. The method returns a list of records. For example:

```
sql = """
SELECT C.firstname || " " || C.lastname as name, count(*), SUM(I.total)
FROM %s AS I JOIN %s AS C ON I.customer = C.id
```

```
    WHERE I.invoicedate >= "%s" AND I.invoicedate <= "%s"
    GROUP BY I.customer
    ORDER BY name
    """
    rows = report.task.execute_select(sql % (report.task.invoices.table_name,
        report.task.customers.table_name, report.invoicedate1.value.strftime('%Y-%m-%d'),
        report.invoicedate2.value.strftime('%Y-%m-%d')))
```

Here, the query is executed in the report event.

For other queries use execute method:

```
def execute(self, sql, params=None):
```

his method returns a tuple - (result, error). If successful, the error is None, otherwise it contains error message, the result is result of the query execution. Sql parameter can be either the query and a list of requests. If a sql parameter is a query than a params can the contain the parameters of the query. The query is executed in a single transaction, and then the transaction is committed.

4.5.2 Server events.

To initialize the task use on_created event handler. It is fired when the task is just created.

For example:

```
def on_created(task):
        task.version = '1.0'
```

For all item on the server working with the data, you can define the following event handlers:

- on_select
- on_record_count
- on_apply

On_apply event can be used if you want to override the data saving procedure on the server during the execution of the method apply. This event has the following form:

```
def on_apply(item, delta, params, privileges, user_info, enviroment):
        pass
```

and has the following parameters

- item - a reference to the item,
- delta - a delta containing item changes (discussed in more detail below),
- params - the parameters passed to the server by apply method,
- privileges - a dictionary containing information about the user's permissions ('can_create', 'can_edit', 'can_delete', 'can_view'),
- user_info - a dictionary containing information about the user,

- enviroment - a dictionary containing standard WSGI environment variables.

The delta parameter contains changes that must be saved on the server. By itself, this option is an item's copy, and its set of records is the item's change log. The nature of the record change can be obtained by using methods rec_inserted, rec_modified or 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 item, delta also has a corresponding detail item storing detail changes. Details_active attribute of delta is True. Please note that if a record is deleted from an item and this record has detail records, the delta will just keep this deleted record, information about the deleted records of the detail is not saved. In this case if you need this detail records you must get them yourself (see example below).

When the data of the 'apply' method are send to the server, server creates an item's delta. Then based on changes stored in the delta the sql request is generated and is passed as a parameter to the 'execute' method of the task:

```
delta = self.delta(changes)
    sql = delta.apply_sql(privileges)
    self.task.execute(sql)
```

As a result, changes are stored in the database in a single transaction. Upon successful completion of this transaction, the 'apply' method on the client updates the change log, and when new records were added, id values of these records are updated.

The example below is taken from the server module of the Invoices journal. There, in the same transaction in which the invoice data are saved, the number of sold tracks is recalculated;

```
def process_delta(delta):

    def get_sold(invoice_table):
        result = {}
        track_ids = []
        for i in invoice_table:
            track_ids.append(i.track.value)
        if track_ids:
            tracks = delta.task.tracks.copy()
            tracks.set_where(id__in=track_ids)
            tracks.open(expanded=False, fields=['id', 'quantity'])
            for t in tracks:
                result[t.id.value] = t.quantity.value
        return result

    result = []
    invoice_table = delta.task.invoice_table.copy()
    for d in delta:
        if d.rec_deleted():
            invoice_table.set_where(owner_id=d.ID, owner_rec_id=d.id.value)
            invoice_table.open(expanded=False, fields=['track', 'quantity'])
            sold = get_sold(invoice_table)
            for i in invoice_table:
                sold[i.track.value] -= i.quantity.value
        else:
            sold = get_sold(d.invoice_table)
            for t in d.invoice_table:
                if t.rec_modified():
                    invoice_table.set_where(id=t.id.value)
                    invoice_table.open(expanded=False, fields=['track', 'quantity'])
                    sold[t.track.value] -= invoice_table.quantity.value
                if t.rec_inserted() or t.rec_modified():
                    sold[t.track.value] += t.quantity.value
                elif t.rec_deleted():
                    sold[t.track.value] -= t.quantity.value
    for track, quantity in sold.iteritems():
        result.append("UPDATE %s SET QUANTITY=%s WHERE ID=%s" % (d.task.tracks.table_name, quantity,
                      track))
    return result


def on_apply(item, delta, params, privileges, user_info, enviroment):
    tracks_sql = process_delta(delta)
    sql = delta.apply_sql()
    return item.task.execute([sql] + tracks_sql)
```

Above in the on_apply event handler the process_delta procedure returns a list of sql queries, that change the number of tracks sold. Then, these queries are executed together with the queries that change the invoice data.

To override the way the open method is executed on the server, you can use the on_select event:

```
def on_select(item, params, user_info, enviroment):
    error_mes = ''
    rows = []
    sql = item.get_select_statement(params)
    try:
        rows = item.task.execute_select(sql)
    except Exception, e:
        error_mes = str(e)
    return rows, error_mes
```

The following parameters are passed to the event handler:

- item - a reference to the item,
- params - the parameters passed by item's open method to the server,
- user_info - a dictionary containing information about the user,
- enviroment - a dictionary containing standard WSGI environment variables.

In the example above, the standard procedures are performed of the 'open' method when it is executed on the server. The 'get_select_statement' method generates an sql query that is executed by the task.

The event should return tuple of a list of records and an error message. In each record, fields values must follow in the same order in which fields were listed in the open function. If the expanded parameter is True, these values must be followed by lookup values of the lookup fields.

Similarly, you can use then on_record_count event handler to override the calculation of the total number of records of the item which is used by the grid component to create pagination:

```
def on_record_count(item, params, user_info, enviroment):
    error_mes = ''
    result = 0
    sql = item.get_record_count_query(params)
    try:
        rows = item.task.execute_select(sql)
        result = rows[0][0]
    except Exception, e:
        error_mes = str(e)
    return result, error_mes
```

4.5.3 Server functions.

If a function is defined in item's server module that starts with a server_ string:

```
def server_function_name(item, param1, param2, …):
        pass
```

then on client in Python, this function can be called as follows:

```
result = item.server_function_name(param1, param2, …)
```

on client in JavaScript the synchronous function call:

```
result = item.server_function('server_function_name', [param1, param2, …])
```

asynchronous function call (callback is some function):

```
item.server_function('server_function_name', [param1, param2, …], callback(result) {
})
```

For example, if we define in the item server module the following function:

```
def server_get_sum(item, value1, value2):
        return value1 + value2
```

we can call this function on client in Python the following way:

```
result = item.server_get_sum(1, 2)
```

on client in JavaScript synchronous call will be:

```
result = item.server_function('server_get_sum', [1, 2])
```

If the server function ends with _env, then it is passed an additional argument - a dictionary containing the WSGI standard environment variables values. For instance on the server:

```
        def server_get_sum_env(item, value1, value2, env):
```

on client:

```
        result = item.server_get_sum(1, 2)
```

4.6 Report programming.

To print a report on a client use the print_report method. As a result of calling this function, a client creates a form for editing the report parameters. When creating this form the on_before_show_params_form events are generated sequentially for task, reports item, and the report itself. In Demo application in the on_before_show_params_form of the task the click on the Print button is connected to the process_report method, which sends request to the server to generate the report. But before doing it an on_before_print_report event is, fired first for the report owner and then for report itself.

The server first of all creates a copy of the report and then this copy fires an on_generate_report event. For example for the Invoice report of the Demo application this event is as follows:

```
def on_generate_report(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.invoicedate.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 create a copy of the invoices journal.

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

We create copies 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 invoice id field value.

Then, using the open method, we obtain the record on the server. After that the title band is printed:

```
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.

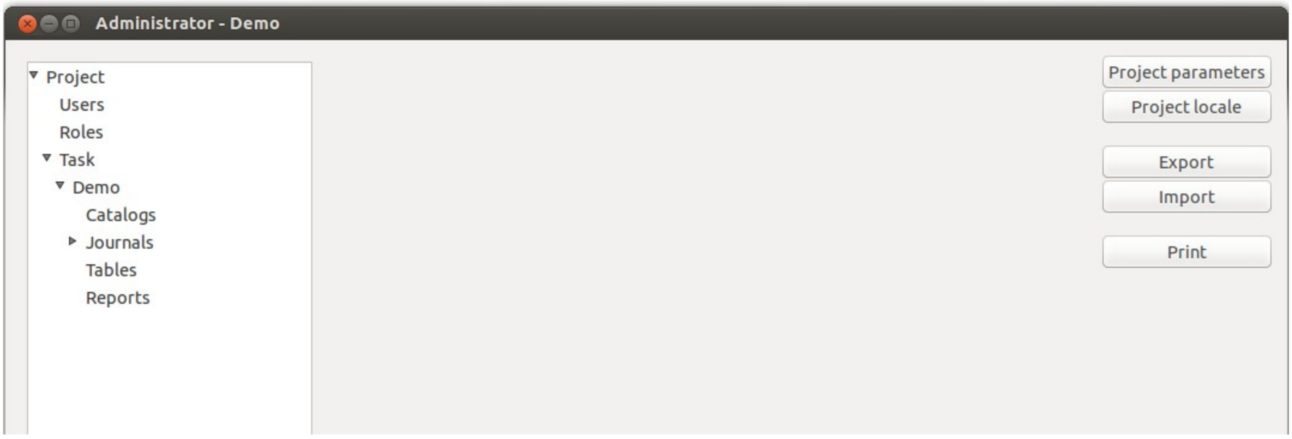
Once the report is generated it is stored in a report folder of the static directory and the server sends the client the report file url.

The report can be converted to another format other than ods. The format can be set on the client using the extension attribute of the report. The conversion is carried out by open office package. Open office can be run in the server mode:

```
soffice --headless --accept="socket,host=127.0.0.1,port=2002;urp;"
```

5 Jam.py utils.

If you select the Project node in the project tree of Administrator the buttons will be available to export and import the metadata of the project and to print its code:



5.1 Exporting and importing project metadata.

Export and import utilities allow developer to save the project metadata in a file. When exporting in the file are saved:

- project parameters
- project locale
- roles and their privileges
- task tree: items, their fields and filters, including reports and their parameters, the program code of all items

When importing project the framework compares the current project metadata with metadata stored in the file. Based on this analysis it finds the differences in the structures of database and generate sql queries. Thereafter, an attempt is made to execute these queries in one transaction. In case of success, the metadata that are stored in admin.sqlite database are updated in one transaction. If the import is done remotely, then upon successful completion of the import the server is stopped. It must be started again.

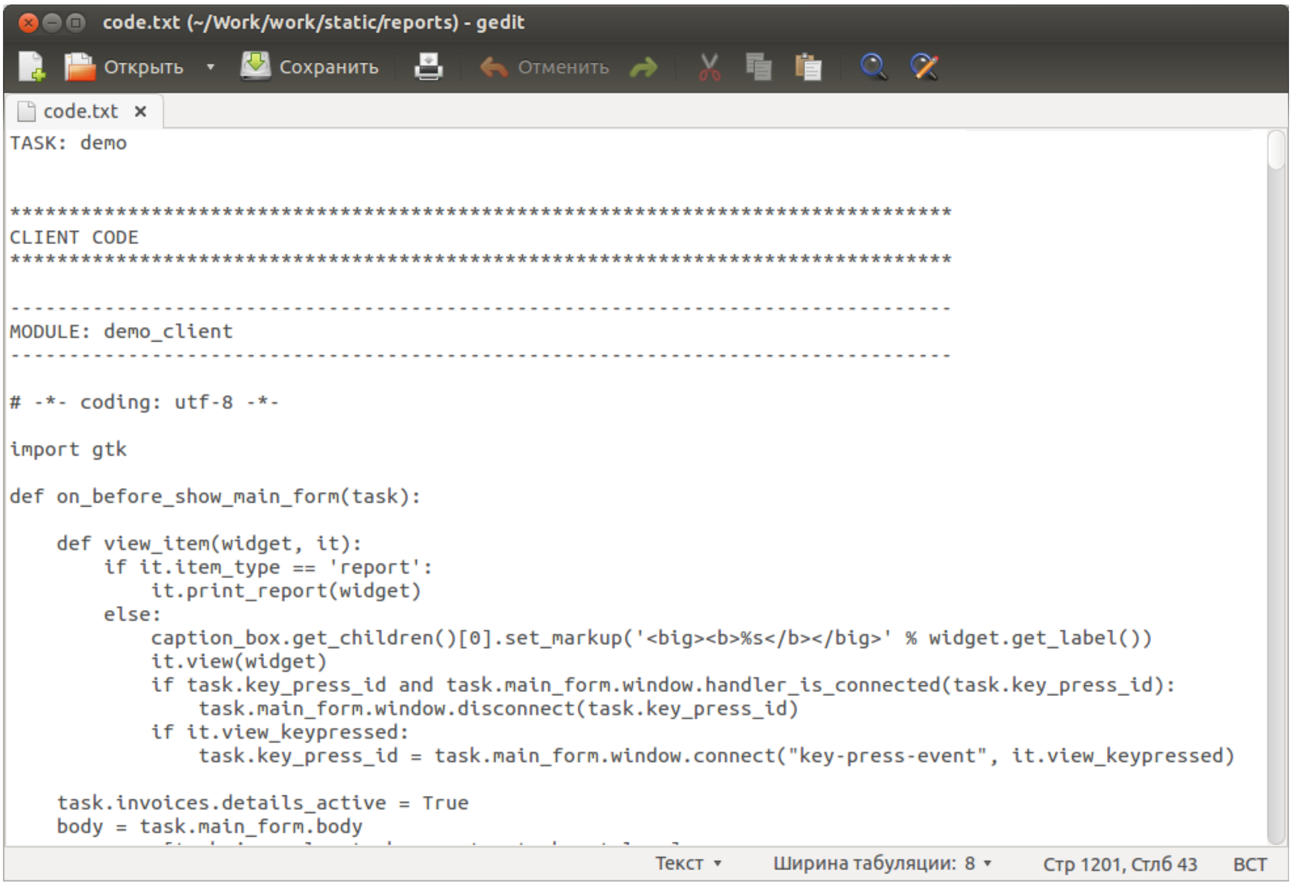
If database changes have been made outside Administrator (manually), directly in the database, they are not included in the export file.

An import is not possible for projects with a SQLITE database.

Be very careful when importing a project. Make backup copies of the project database and the file admin.sqlite.

5.2 Printing of programming code.

The programming code of the project is stored in a set of different modules. This can be inconvenient if you need to get acquainted with all the code of the project. Press the Print button for all the code of the project to be displayed in a single file:



```
TASK: demo


********************************************************************************
CLIENT CODE
********************************************************************************

--------------------------------------------------------------------------------
MODULE: demo_client
--------------------------------------------------------------------------------

# -*- coding: utf-8 -*-

import gtk

def on_before_show_main_form(task):

    def view_item(widget, it):
        if it.item_type == 'report':
            it.print_report(widget)
        else:
            caption_box.get_children()[0].set_markup('<big><b>%s</b></big>' % widget.get_label())
            it.view(widget)
            if task.key_press_id and task.main_form.window.handler_is_connected(task.key_press_id):
                task.main_form.window.disconnect(task.key_press_id)
            if it.view_keypressed:
                task.key_press_id = task.main_form.window.connect("key-press-event", it.view_keypressed)

    task.invoices.details_active = True
    body = task.main_form.body
```

Текст ▾   Ширина табуляции: 8 ▾   Стр 1201, Стлб 43   ВСТ