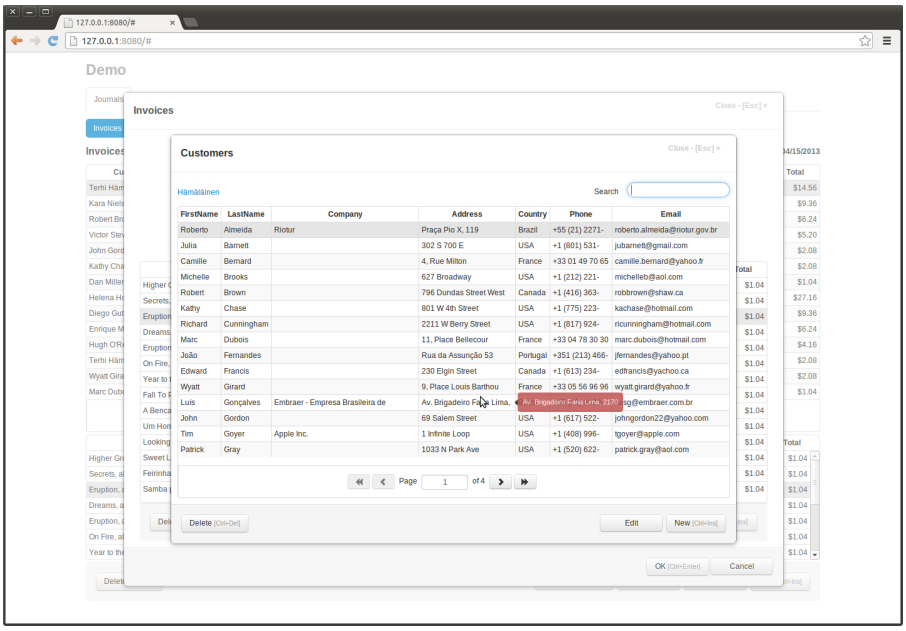
Jam.py

Руководство разработчика.

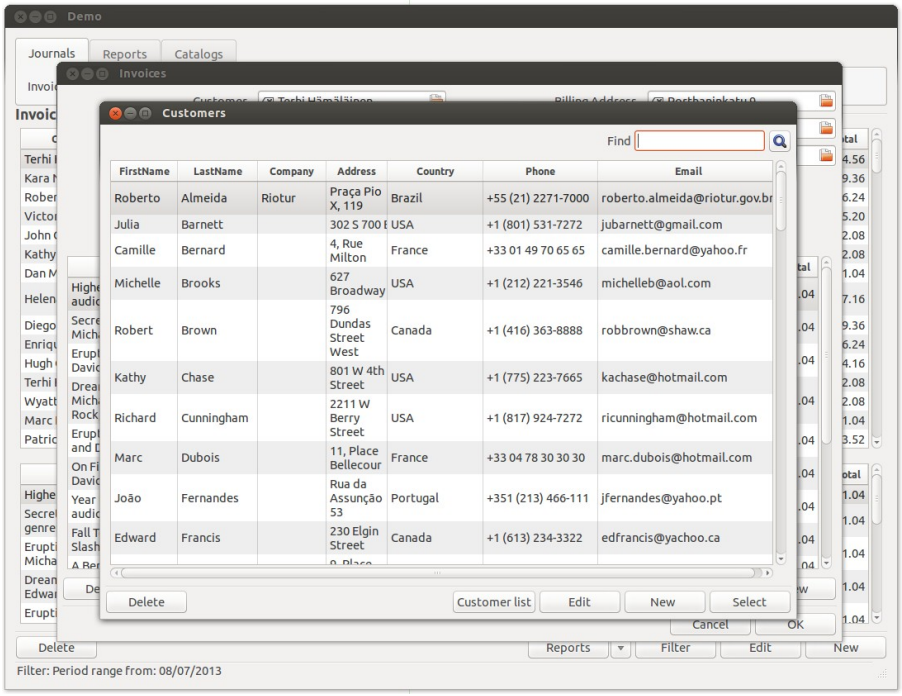
Оглавление

1 Обзор.

Jam.py – это, управляемый событиями, фреймвок для создания клиент-серверных приложений для работы с базами данных. Он позволяет создавать приложения веб-клиент -сервер серверная часть реализована на python, клиентская библиотека - на javascript и использует jquery и bootstrap:



А также, локальные десктоп приложения (python, pygtk) и приложения десктоп клиент (python, pygtk) - сервер (python):



Принцип повторного использования кода, лежащий в основании фреймвока, позволяет разработчику сосредоточится на программировании бизнес логики задачи.

2 Начало работы

2.1 Установка.

1. Загрузите архив пакета.

2. Создайте новую директорию и распакуйте туда архив.

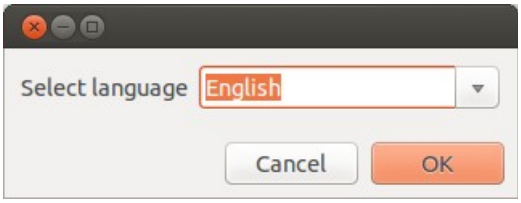
3. Перейдите в эту директорию и выполните в командной строке: python setup.py install.

2.2 Создание нового проекта.

1. Создайте новую директорию.

2. Перейдите в эту директорию и выполните в командной строке: jam-project.py.

3. В открывшемся окне выберите язык проекта на нажмите клавишу OK.

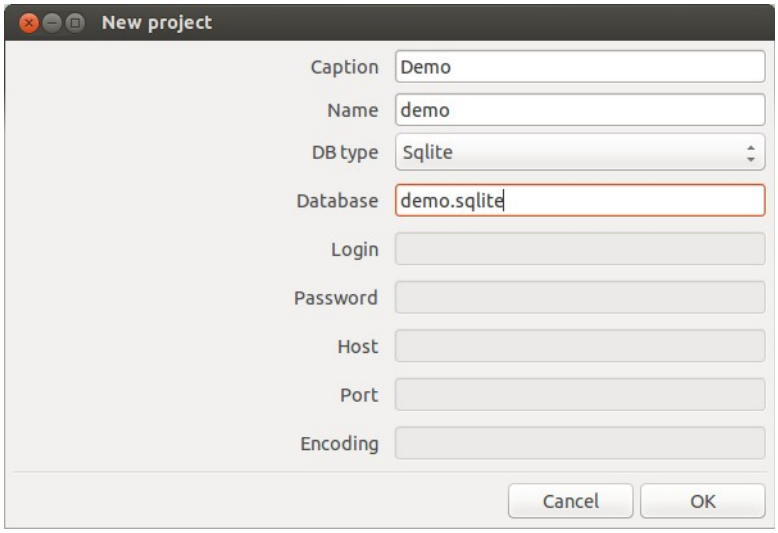


4. В появившемся диалоговом окне необходимо ввести параметры нового проекта:

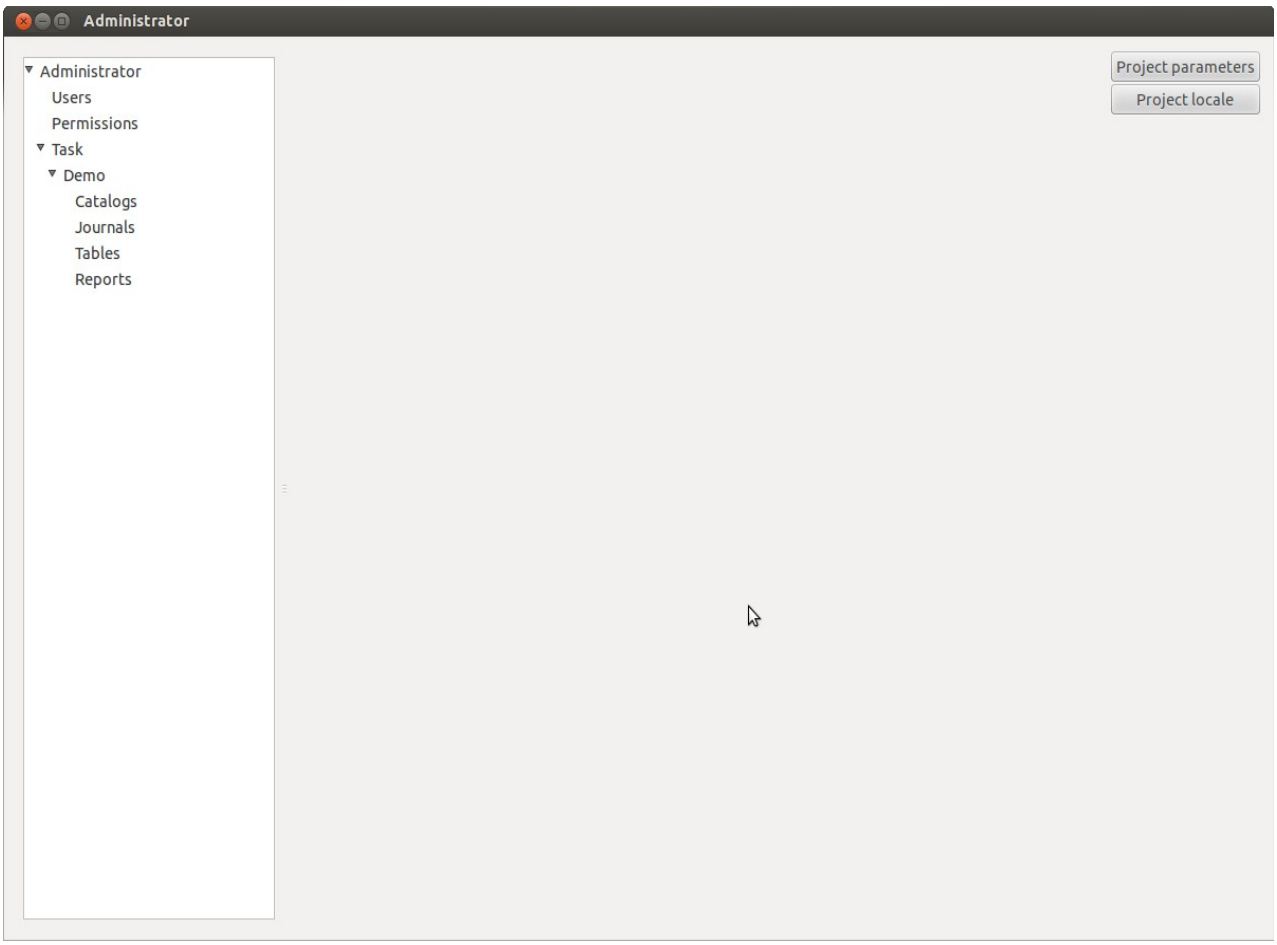
Наименование (caption) – название проекта - строка символов (до 200 знаков) .

Имя (name) – должно являться валидным питоновским именем.

Тип базы данных (DB type) .Если база данных - не Sqlite, то ее нужно создать заранее и ввести в форму необходимые атрибуты. При нажатии на клавишу OK, Администратор проверит подключение к базе, и в случае неудачи выдаст сообщение.



Если все будет нормально то будет создан новый проект и в администраторе появится дерево проекта.



В директории проекта будут созданные следующие файлы:

- **server.py** – файл запуска сервера. В качестве параметра может быть указан порт, например: server.py 8081. По умолчанию значение порта: 8080
- **admin.py** – файл запуска администратора. В качестве параметра может быть указан url и порт сервера, например: admin.py http://127.0.0.1:8080. В этом случае администратор будет запущен как удаленный desktop клиент. В отсутствии параметров – как локальное desktop приложение
- **main.py** - файл запуска локального desktop приложения
- **client.py** - файл запуска desktop клиента. В качестве параметра может быть указан url и порт сервера. Если параметр не указан по умолчанию берутся http://127.0.0.1:8080
- **index.html** - главный файл web клиента

и папки:

- **js** - javascrip файлы
- **css** - css файлы
- **img** — файлы изображений
- **ui** – папка, в которой хранятся файлы glade шаблонов, которые могут быть использованы при создании desktop приложений
- **static** - папка static сервера

Пожалуйста, обратите внимание на следующие требования:

- для запуска desktop приложений необходимо, чтобы был установлен GTK+2 и PyGTK
- для того, чтобы использовать базу данных firebird, надо установить библиотеку fdb
- для работы с PostgreSQL требуется библиотека psycopg2
- для формирования отчетов должен быть установлен OpenOffice

3 Разработка первого приложения.

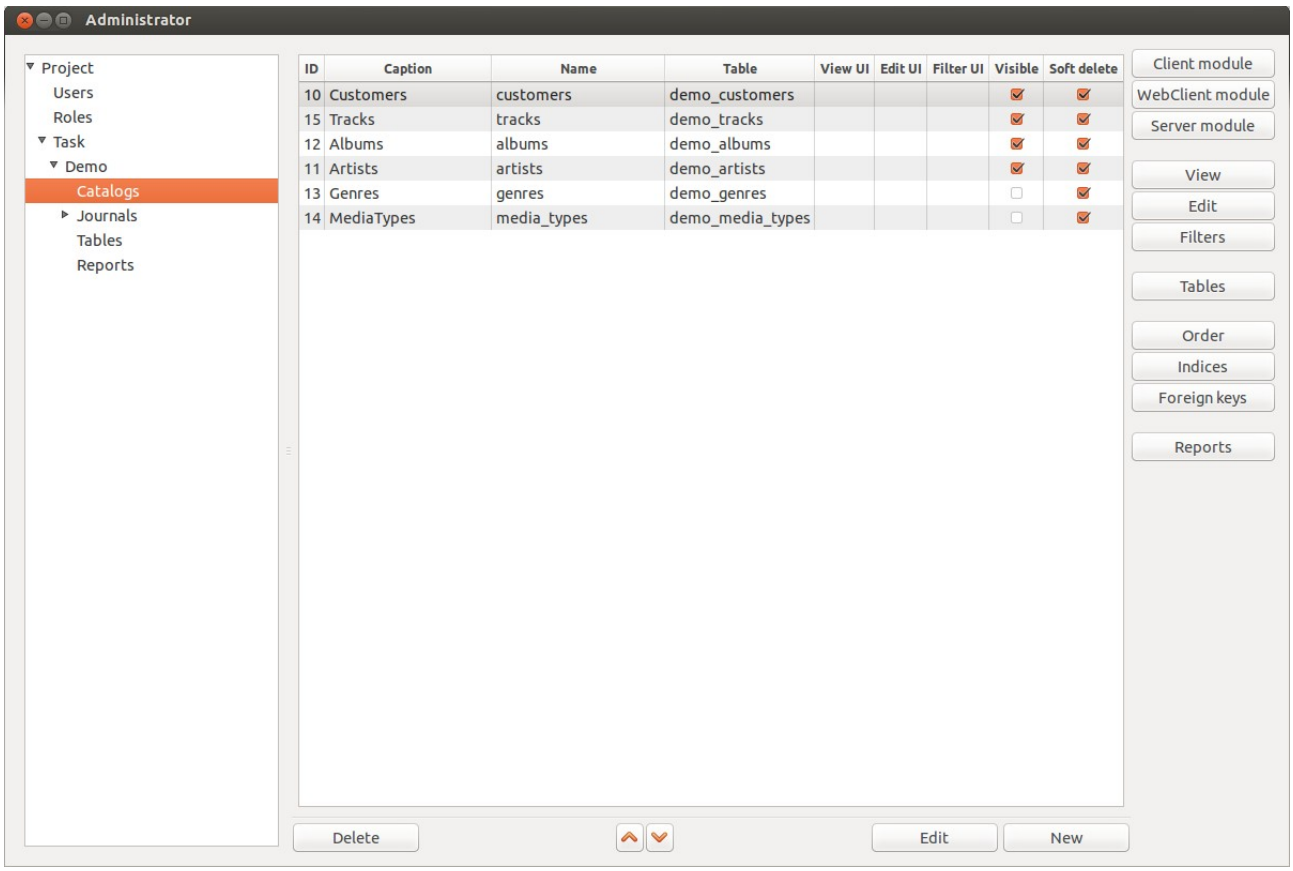
3.1 Проект Demo.

В папке, куда был распакован архив проекта, находится папка demo с демонстрационным проектом. Далее мы покажем, как разработать такой проект. Для того, чтобы посмотреть работу demo надо перейти в эту папку. Для запуска локального desktop приложения надо запустить скрипт main.py. Для того, чтобы посмотреть работу клиент-серверного приложения, необходимо сначала запустить сервер server.py в командной строке. После этого в браузере в адресной строке набрать 127.0.0.1:8080. Для запуска desktop-клиента надо в другом терминале перейти в папку demo и запустить client.py.

3.2 Администратор.

Теперь, с помощью скрипта admin.py, запустим Администратор. Администратор - это приложение, созданное на фреймвоке jam.py, в котором происходит разработка проекта. Фактически он содержит метаданные проекта — структуры таблиц, программный код, управляющий приложением и т.д.. При запуске на экране откроется окно Администратора. С левой стороны окна располагается дерево проекта. Если выбрать мышью какую либо из ветвей дерева, то в центральной части окна откроется ее содержание. Например, если мы выберем ветвь Справочники, то в центральной части будет выведен список справочников.

Как правило, в нижней и правой части окна Администратора располагаются кнопки позволяющие производить некоторые действия над содержимым центральной части.



3.3 Создание первого справочника.

Все таблицы проекта, условно делятся на 3 категории. Справочники, журналы и таблицы. Деление это довольно условное.

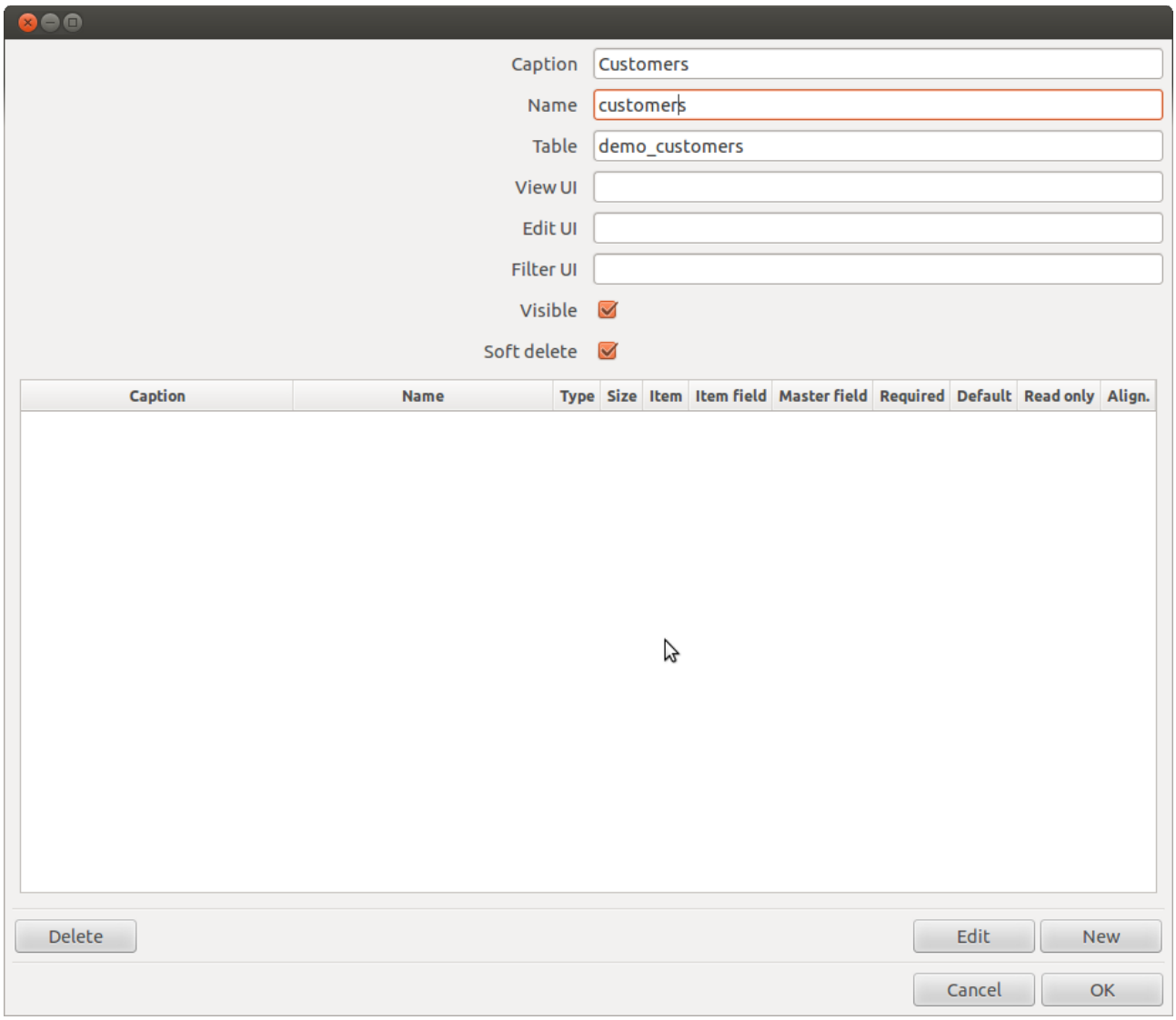
Справочники — это таблицы содержащие такую информацию как единицы измерения, список клиентов, организаций и т.д. При создании других таблиц, в том числе и справочников, мы можем создавать поля, которые являются ссылкой на запись в каком либо справочнике.

Журналы — это структуры, в которых хранятся информация о каких либо событиях, зафиксированных в каких либо документах, таких как накладные, заказы и т.д.

Таблицы по сути аналогичны журналам. Но помимо этого он могут быть прилинкованы к другим таблицам и хранить их табличную часть. Например список товаров накладной.

После того как мы создадим демо проект, смысл этого деления будет более понятен.

Выше мы уже создали проект Demo с Sqlite базой demo.sqlite. Запустим администратор нашего нового проекта и займемся созданием справочников. Начнем со справочника Клиенты. В дереве проекта выберем ветку справочники. Список справочников проекта пока пуст. Создадим новый справочник Customers (Клиенты). Для этого щелкнем мышью по кнопке Добавить в правой нижней области окна Администратора. На экране откроется окно редактирования справочника.



В верхней части этого окна зададим:

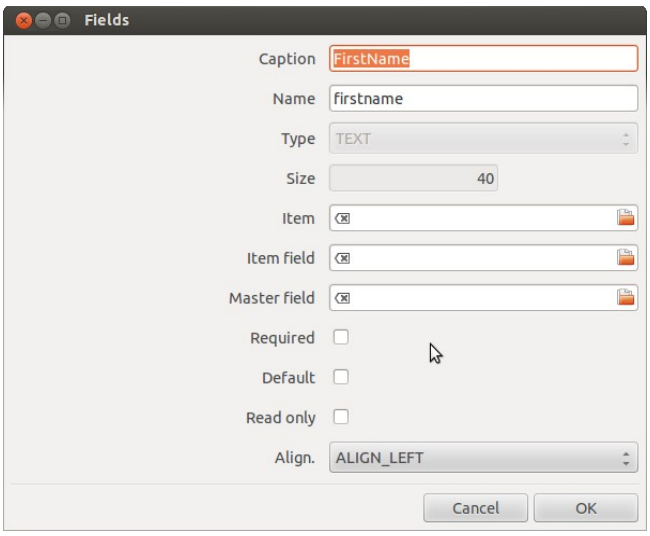
- имя справочника (name), оно должно быть уникальным в проекте и являться валидным питоновским именем, и может быть в дальнейшем использовано в программном коде.

- наименование (caption) – название справочника.

Администратор сформирует название таблицы базы demo.sqlite - demo_customers.

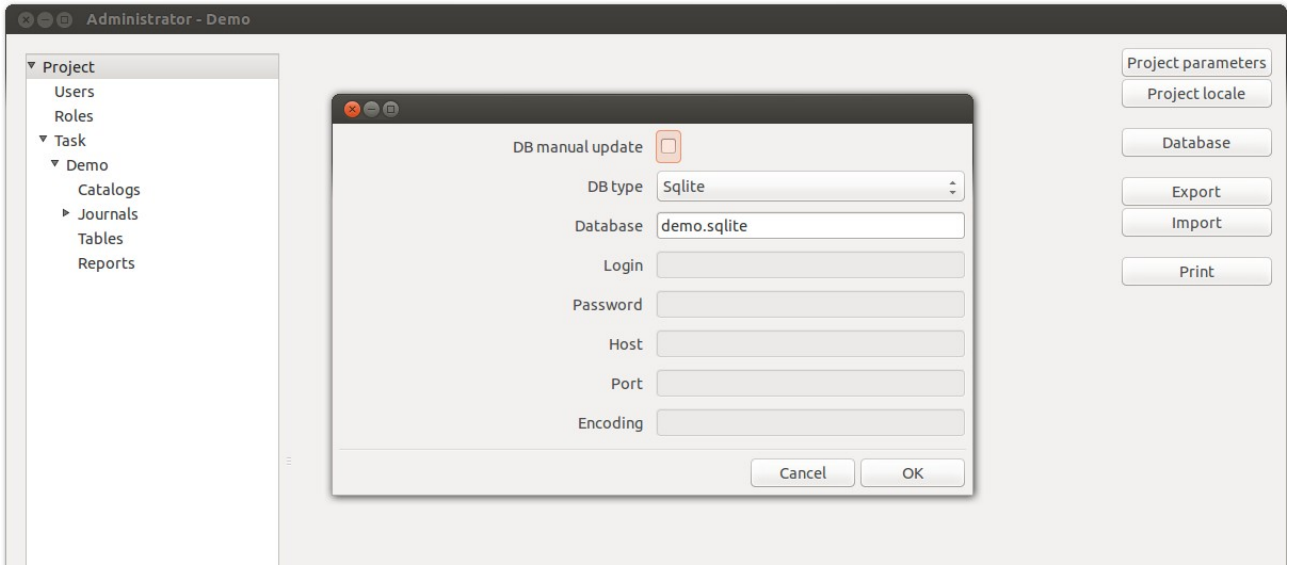
Остальные атрибуты мы пока пропустим, мы вернемся к ним при обсуждении программирования интерфейса, и перейдем к созданию полей. Для этого щелкнем по кнопке «Добавить» в правом нижнем углу окна.

В открывшемся окне Редактирования введем наименование поля, его имя (уникальное в справочнике, валидное питоновское имя), выберем из списка тип поля, зададим его длину и нажмем кнопку «Сохранить».

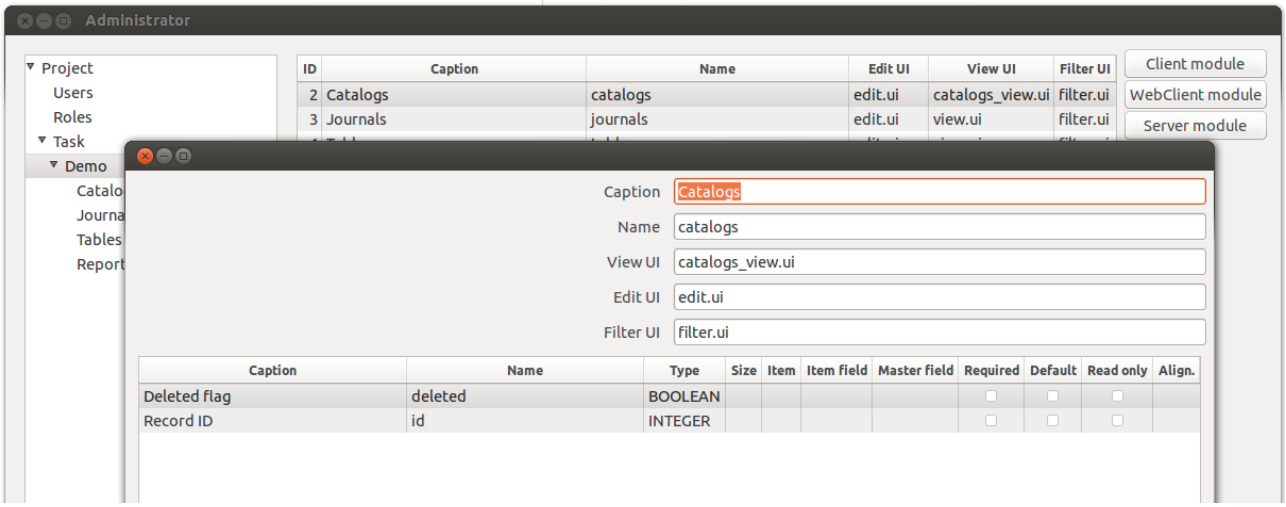


Мы добавили поле firstname. Теперь аналогично добавим поле lastname. Но перед сохранением поставим галочки напротив атрибутов «Обязательное» и «По умолчанию». Эти галочки означают, что при добавлении в справочник нового клиента, программа не позволит сохранить данные, если не будет задано значение этого поля и это поле будет полем по умолчанию справочника (в интерфейсе по умолчанию реализован поиск по этому полю).

Аналогичным образом добавим остальные поля и щелкнем по кнопке «Сохранить». Администратор создаст в базе данных проекта demo.sqlite таблицу demo_customers. Вообще когда мы создаем, изменяем или удаляем поля в Администраторе таблица базы данных соответствующим образом изменяется. Такое поведение можно изменить, если установить свойство DB manual update в значение True в свойстве Database ветки Project. Тогда поля в таблице придется изменять вручную.

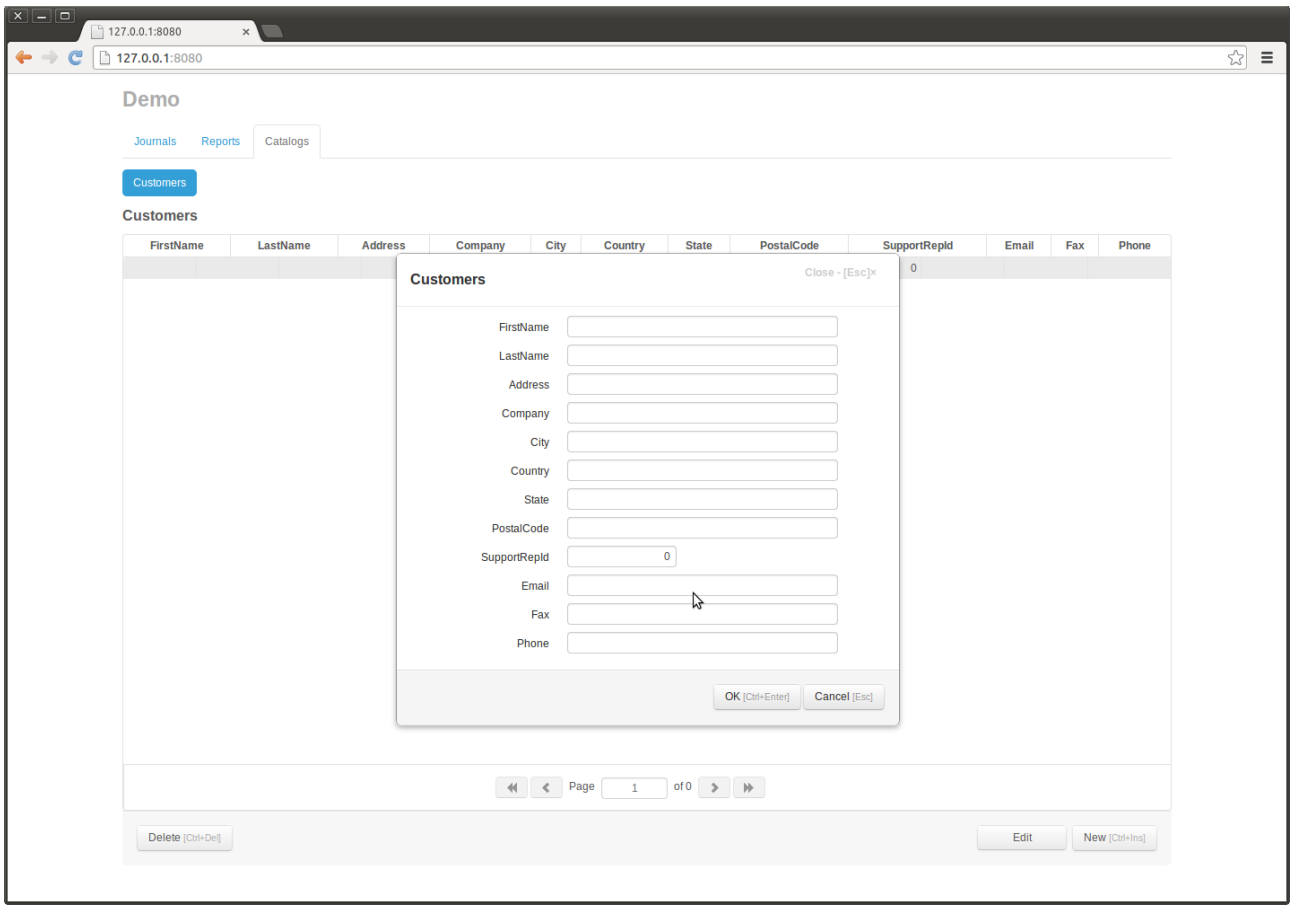


Если теперь открыть базу данных в SQLite Manager, например, и посмотреть структуру таблицы demo_customers, то мы увидим, что помимо полей заданных нами, в таблице присутствуют еще 2 поля — id и deleted. Выберем ветку Demo в дереве проекта и сделаем двойной щелчок по записи Catalogs. В открывшемся окне мы увидим определение этих двух полей.

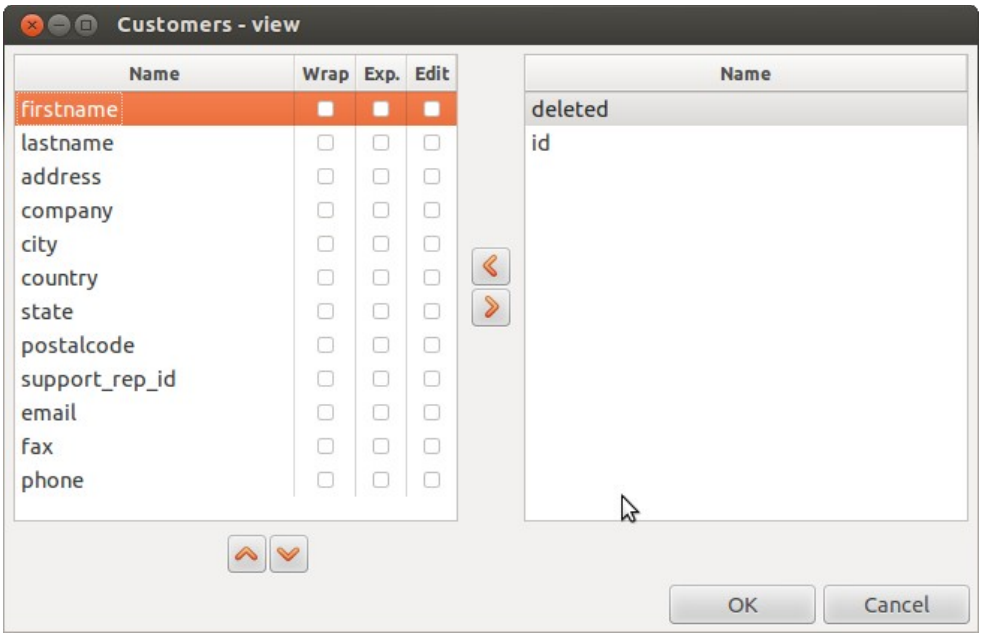


Первое поле id будет содержать уникальный идентификатор записи справочника. Второе поле является признаком удаления. При создании справочника напротив атрибута Soft delete стояла галочка. Смысл этого атрибута в том, что если он установлен,то при удалении записи она не будет удаляться физически, а просто помечаться как удаленная. Так что если мы удалим какую-нибудь запись из справочника то она перестанет отображаться при его просмотре, но если ссылка на эту запись хранится в каком-нибудь поле другого справочника, журнала или таблицы, то данные этой записи будут отображаться при просмотре.

Запустим сервер и в браузере введем адрес 127.0.0.1:8080. Щелкнем по меню «Справочники» (Catalogs). Мы увидим, что в этом меню появилось подменю «Клиенты» (Customers). Щелкнем под этому подменю, и затем по кнопке «Добавить» (New). В браузере появится таблица и модальная форма с созданными нами полями. Отметим, что при создании нового проекта, в нем реализован интерфейс по умолчанию, который можно программно изменять. Мы рассмотрим как это сделать в главе «Программирование интерфейса». Кстати, галочка напротив атрибута «Visible» (Видимый) при создании справочника определяет будет ли он отображаться в подменю Справочники. И это запрограммировано в интерфейсе по умолчанию.



Для того, чтобы изменить список по умолчанию отображаемых полей и их порядок при просмотре и редактировании, перейдем в Администратор, выберем справочник «Customers» и щелкнем по кнопке справа «Просмотр» (View). На экране появится окно в котором можно задать список и порядок полей при просмотре.

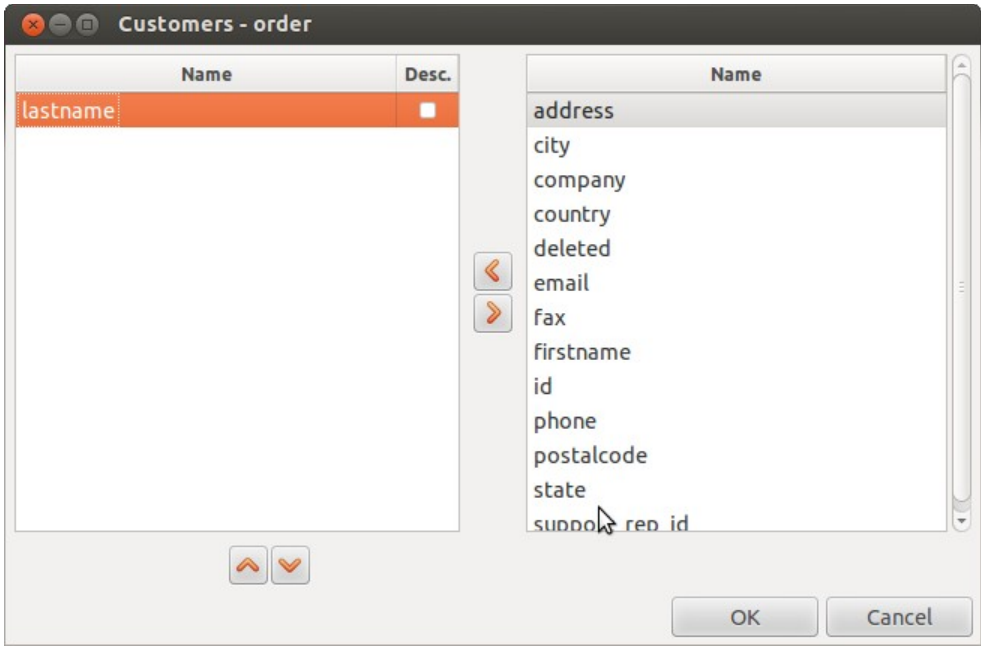


В левой части находятся отображаемые поля, в левой - все остальные. Перемещая кнопками в центре поля, мы изменяем список отображаемых полей, а кнопками внизу - их

порядок.

Аналогичным образом с помощью кнопки Редактирование (Edit), меняется список и порядок полей по умолчанию при редактировании.

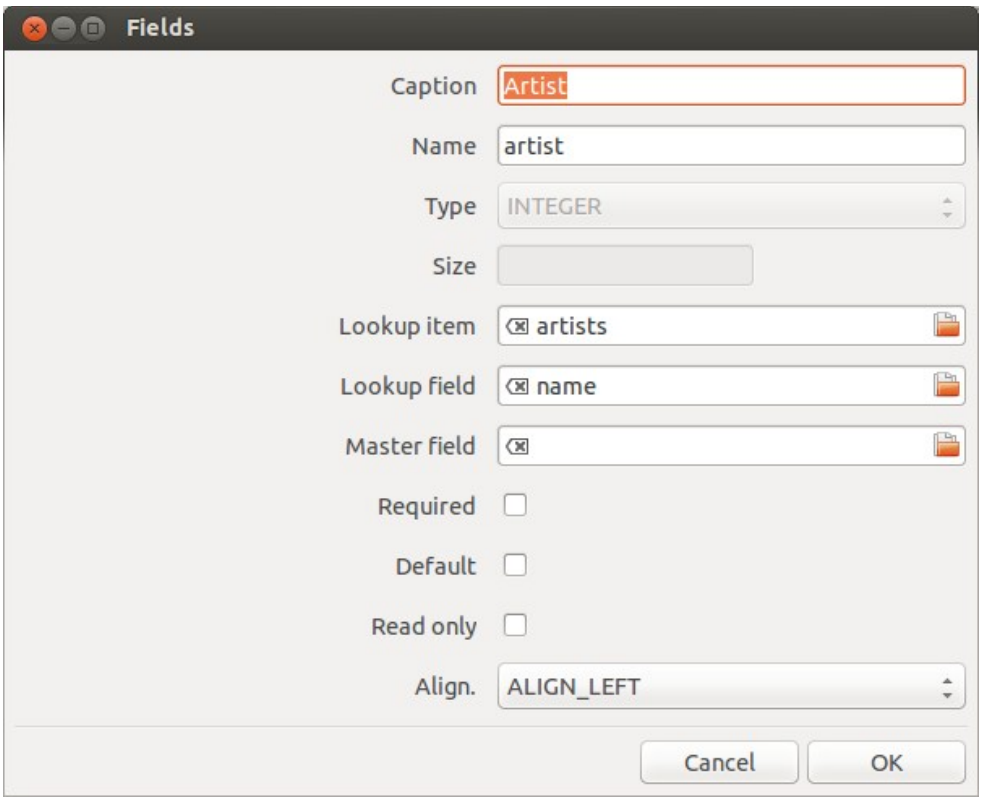
Если мы откроем проект Demo, то увидим, что строки справочника «Клиенты» отсортированы по полю lastname. Для задания сортировки по умолчанию, щелкнем по клавише «Порядок» (Order) и зададим сортировку по полю lastname.



Установив галочку в колонке Desc, можно задать сортировку по убыванию.

3.4 Завершаем создание справочников.

Аналогичным образом создаем справочники Artists, Genres и Media types. А вот в справочнике Albums (альбомы) одно из двух полей artist (исполнитель) является ссылкой на запись в справочнике Artists. При его создании необходимо выбрать в поле Item справочник Artists, а в поле Item field – поле name справочника Artists (класс AbsrtactItem — это общий предок всех ветвей из дерева задачи, дерево задачи — это дерево вершиной которого является ветка Demo, далее мы будем часто использовать item для обозначения этих ветвей). Такие поля в дальнейшем будем называть lookup полями.

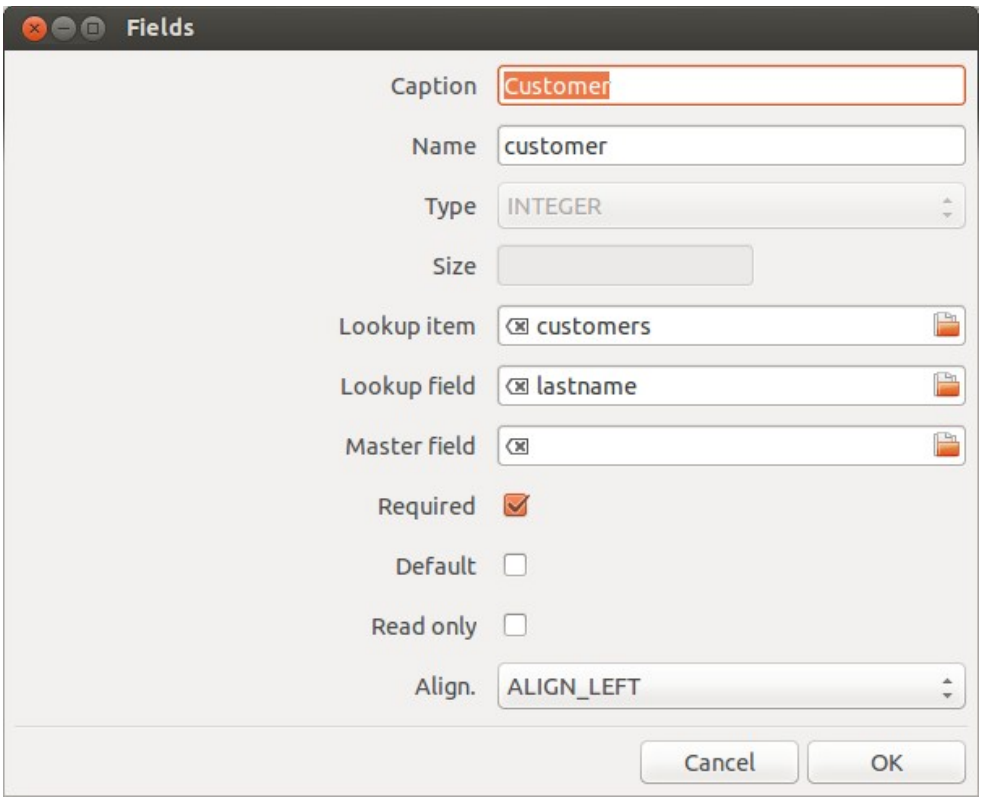


В последнем справочнике Tracks 3 таких поля: album, genre и media_type. Созданием справочника Tracks мы пока завершаем работу со справочниками.

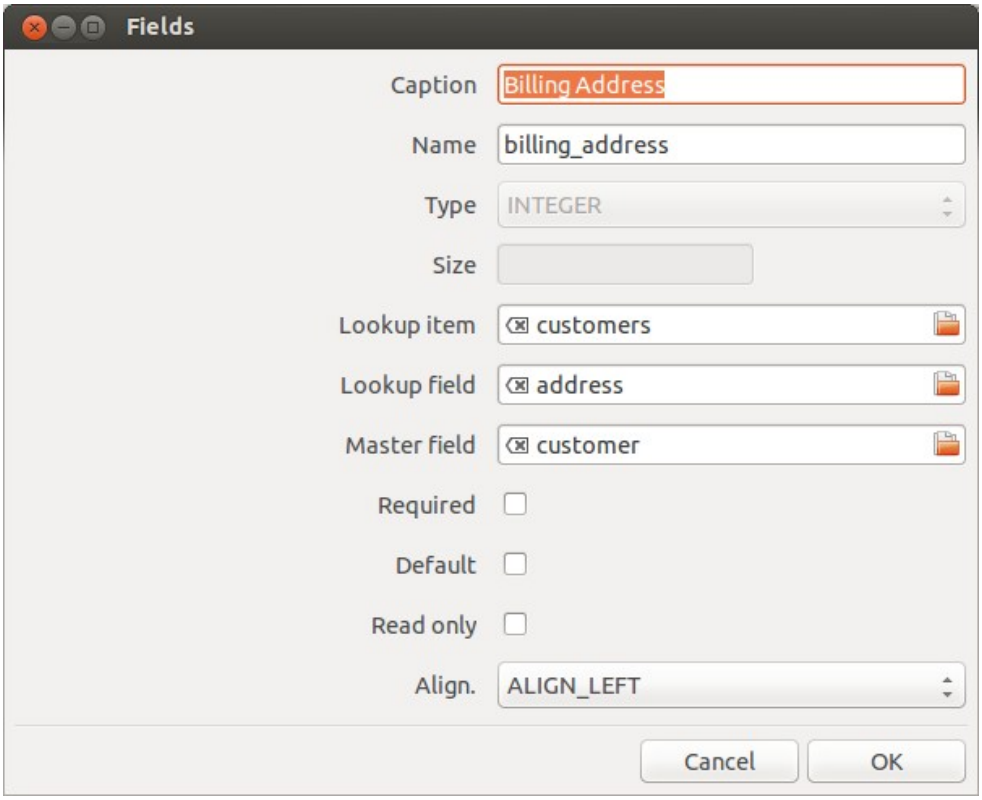
3.5 Создание журналов и таблиц.

В проекте Demo есть единственный журнал Invoices (Накладные) и таблица InvoiceTable, причем поля журнала Invoices — это шапка накладной, а в прилинкованной таблице InvoiceTtable будут храниться записи треков. В принципе создание структур данных журналов и таблиц ничем не отличается от создания структур справочников.

Поэтому здесь мы только покажем как создавать связанные поля. Если посмотреть журнал Invoices в проекте Demo, то поля Customer FirstName, BillingAdress, BillingCity являются полями FirstName, Adress, City записи справочника Customers на которую ссылается поле Customer. Поэтому достаточно создать в таблице Invoices одно INTEGER поле Customer, которое будет хранить id этой записи, а получение значений остальных полей реализовать в запросе.



Такое поле по отношению к другим полям мы будем называть master field и эти поля называть связанными. При этом, при создании остальных lookup полей с item Customers, будем выбирать в атрибут master field поле Customer. (Перед созданием таких полей надо сохранить item с master полем, чтобы ему присвоился ID).

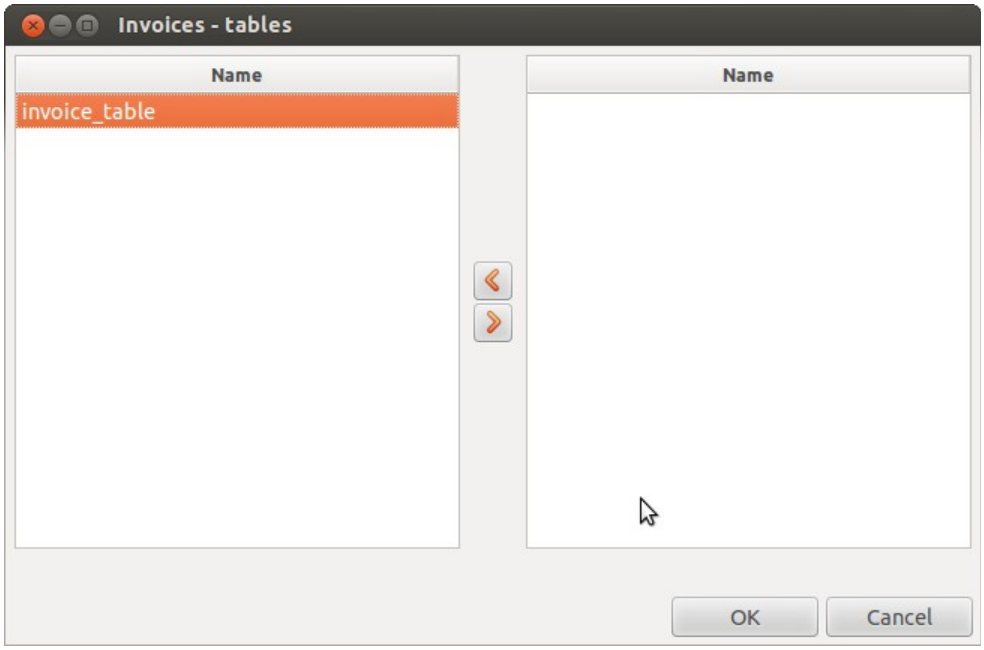


Теперь мы покажем как прилинковать (подвязать) таблицу в журналу.

Если мы выделим в дереве задачи ветку Demo, а затем двойным щелчком откроем запись Tables, то увидим, что помимо полей id и deleted, здесь присутствуют еще два поля owner_id

и owner_rec_id. Дело в том, что каждый Item и каждое поле каждого item'а имеют в системе уникальный ID. Так что, когда мы будем сохранять накладную, то для каждого трека в поле owner_id будет писаться ID журнала Накладные, а в поле owner_rec_id значение поля id данного трека. Таким образом, в принципе, одну и туже таблицу можно подвязать к нескольким журналам, справочникам или таблицам.

Теперь мы покажем как это сделать. Для этого выберем ветку Journals, выделим журнал и щелкнем по находящейся справа клавише Tables, выделим справа таблицу, переместим ее влево и щелкнем на кнопку OK.



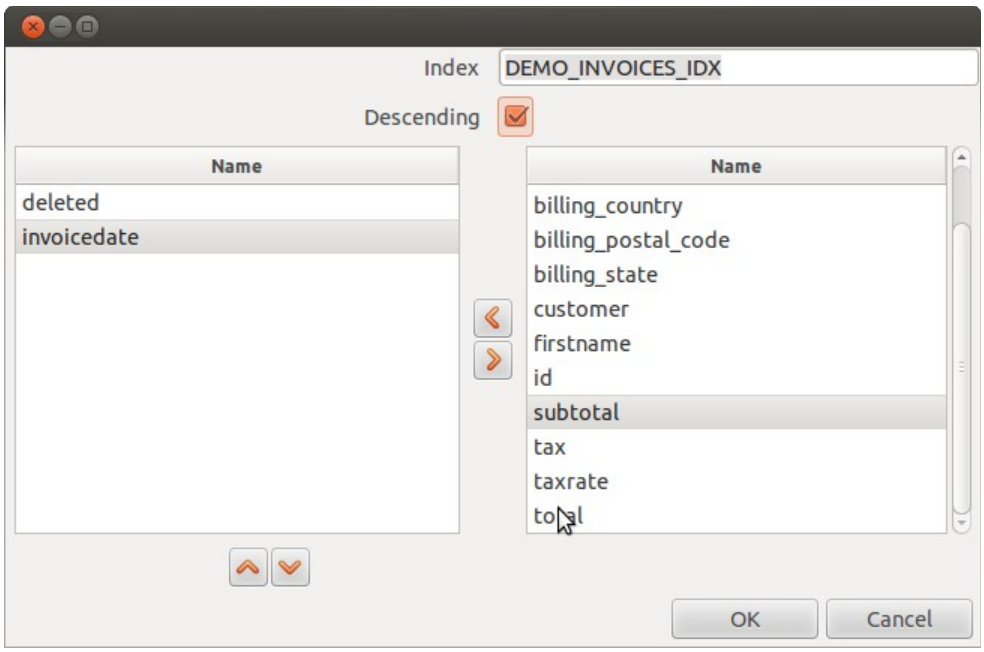
В результате у ветки Journals появится дочерняя ветка Invoices и если мы выделим ее, то в центре отобразится список прилинкованных таблиц этого журнала (в данном случае только одна), у которых мы сможем программировать события и изменять параметры их отображения.

3.6 Создание фильтров.

Если в Demo программе открыть журнал Invoices и щелкнуть по кнопке Filter, то откроется модальное окно задания параметров фильтрации журнала. Для создания фильтров в Администраторе надо выбрать журнал Invoices и щелкнуть по находящейся справа кнопке Filters. Откроется форма содержащее список имеющихся фильтров. Для изменения или добавления нового фильтра надо щелкнуть по соответствующей кнопке на форме. При этом откроется редактор фильтра. После чего надо задать поле, по которому будет производится фильтрация, наименование, имя и тип фильтра.

3.7 Создание индексов.

Мы создали все нужные нам структуры данных. Приступим теперь к созданию индексов. Выберем журнал Invoices (щелкнуть на ветке Journals и затем на записи Invoices) и щелкнем по кнопке Indices. В открывшемся окне щелкнем на кнопку «Добавить», и зададим индекс по убыванию по полю invoicedate. При необходимости надо изменить имя индекса.



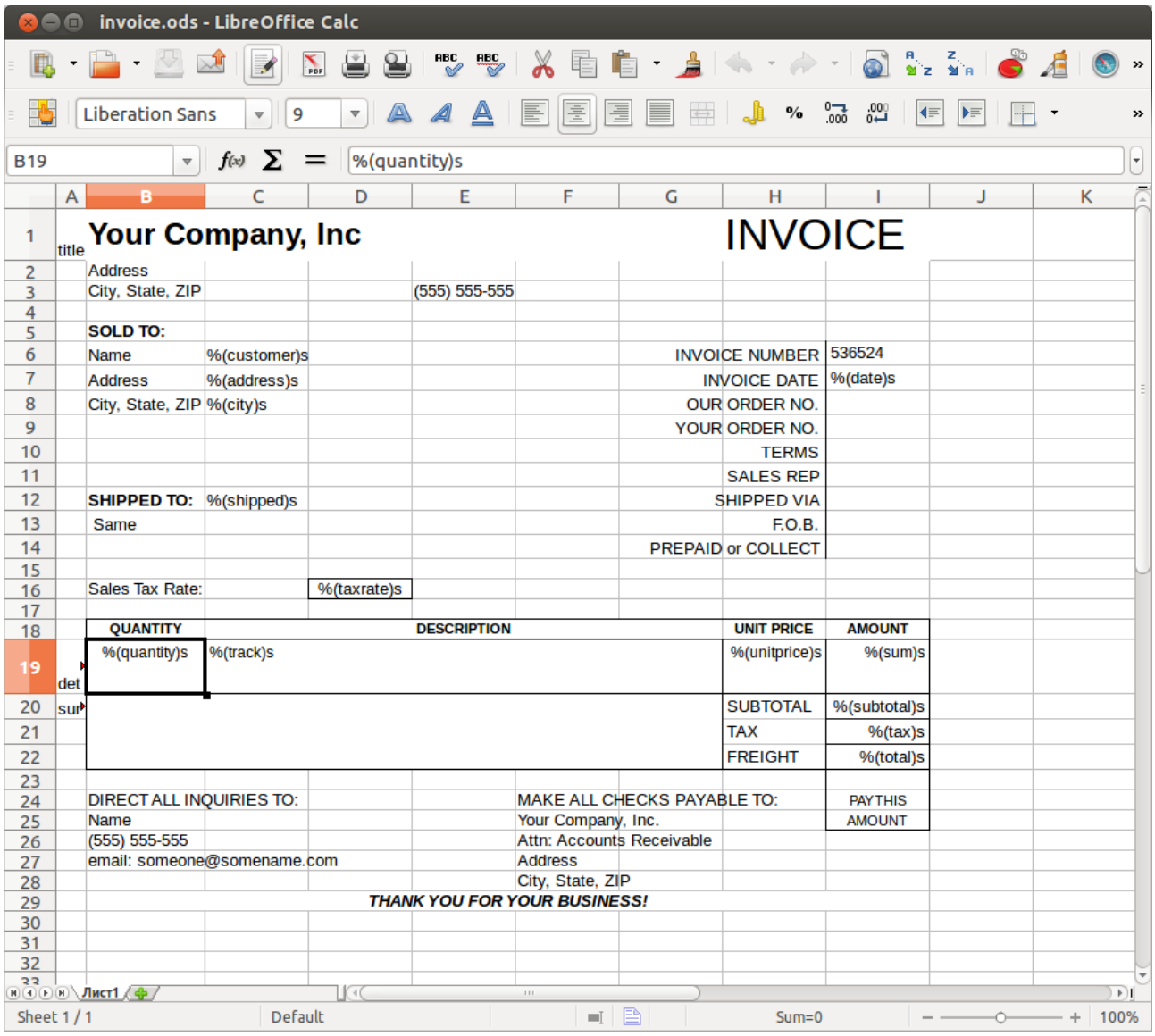
Теперь щелчком по кнопке OK создадим индекс.

Таким же образом создадим индекс для таблицы InvoiceTable по полям owner_id и owner_rec_id.

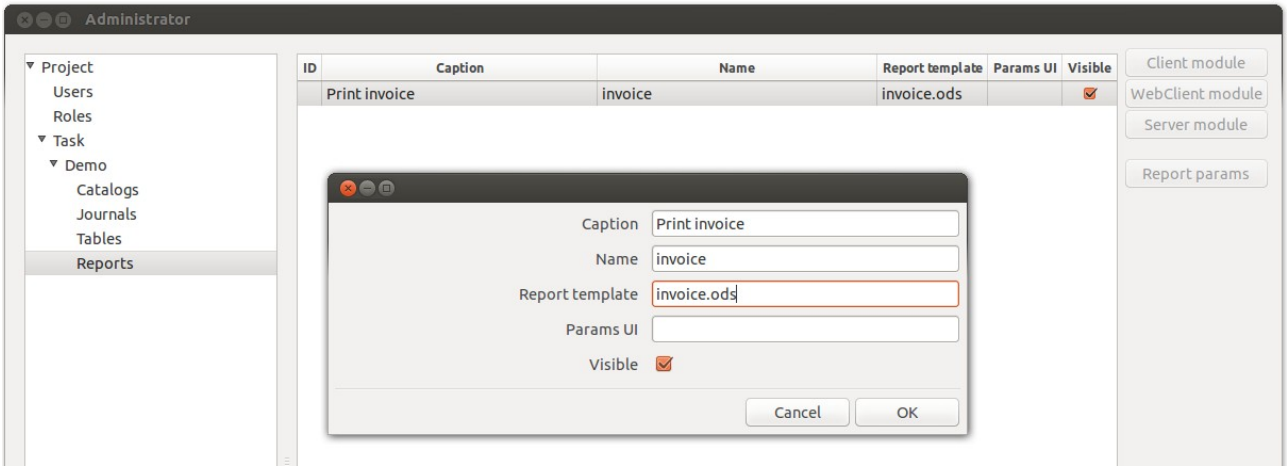
В случае, если определении item в lookup поле атрибут soft delete не установлен, то для сохранения целостности данных можно создать foreign key. Для этого надо щелкнуть на кнопку Foreign keys, выбрать поле и нажать клавишу OK.

3.8 Построение отчетов.

Для создания отчета необходимо сначала подготовить шаблон отчета в OpenOffice (LibreOffice) Calc. Файлы шаблонов находятся в папке report в директории проекта. Ниже на рисунке представлен шаблон отчета «Invoice» приложения Demo. Отчеты в jam.py являются бэнд-ориентированными. Каждый шаблон отчета разделен на бэнды (полосы). Для задания бэндов используется крайняя левая колонка. В шаблоне отчета «Invoice» заданы три бэнда: title, detail и summary. Помимо того, в шаблонах можно создавать программируемые ячейки. Например, в шаблоне «Invoice» ячейка I7 содержит текст %(date)s. Программируемая ячейка начинается символами %(, затем задается имя ячейки, после чего следуют символы)s.



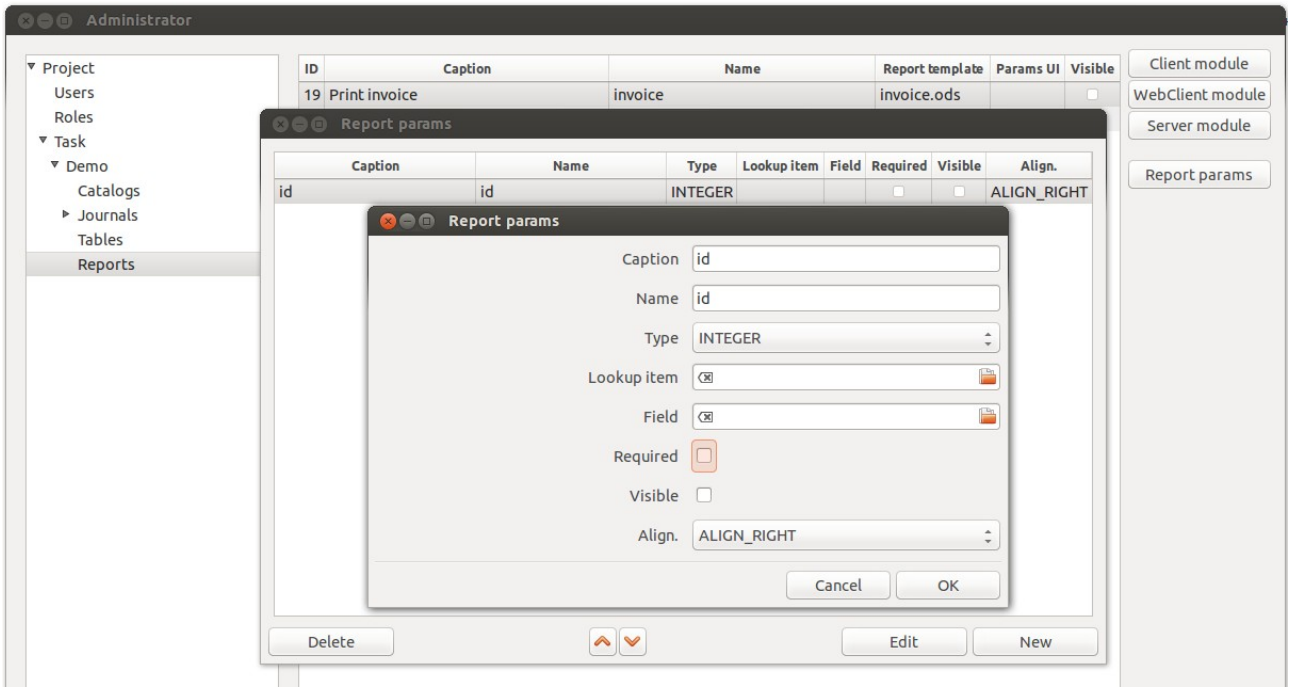
Добавим отчет invoice в наш проект. Для этого в дереве проекта выберем ветку Reports и щелкнем по кнопке «Add» (Добавить).



Теперь введем в поля Caption, Name и Report template наименование, имя отчета и имя файла с шаблоном отчета из папки reports и сохраним отчет.

Так как все отчеты генерируются на сервере, то необходимо передать на сервер id

накладной. Для этого мы создадим параметр отчета. Щелкнем по кнопке Report params (Параметры отчета) на панели справа и затем в диалоговом окне по кнопке «New» (Добавить).



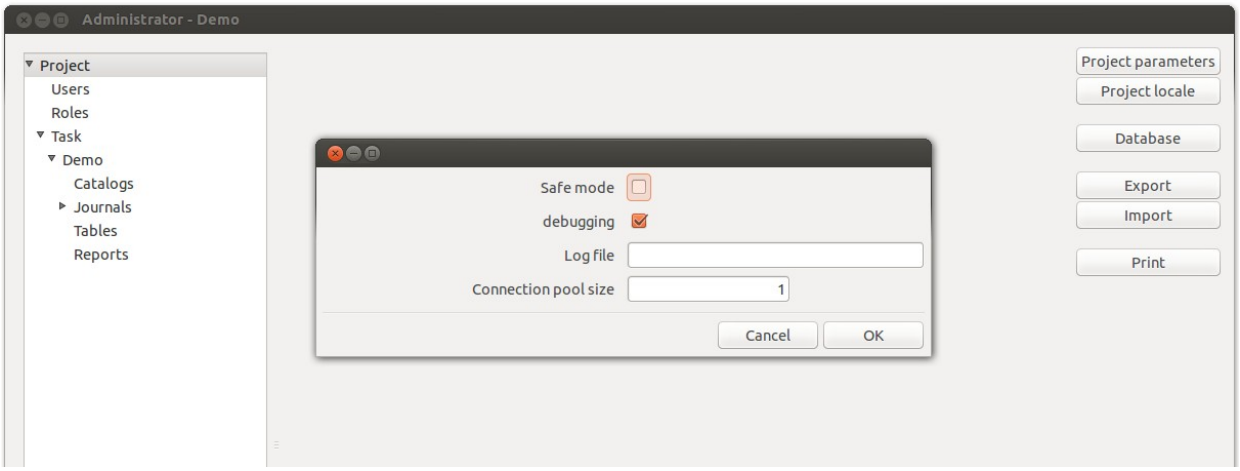
Теперь введем в поля Caption, Name и наименование, имя поля, укажем тип — integer, снимем галочку с поля Visible и сохраним параметр.

Сам процесс генерации отчета на сервере мы рассмотрим ниже в главе «Программирование отчетов».

Далее мы перейдем к рассмотрению программирования в фреймворке jam.py, но перед этим рассмотрим параметры проекта и систему безопасности.

3.9 Параметры проекта.

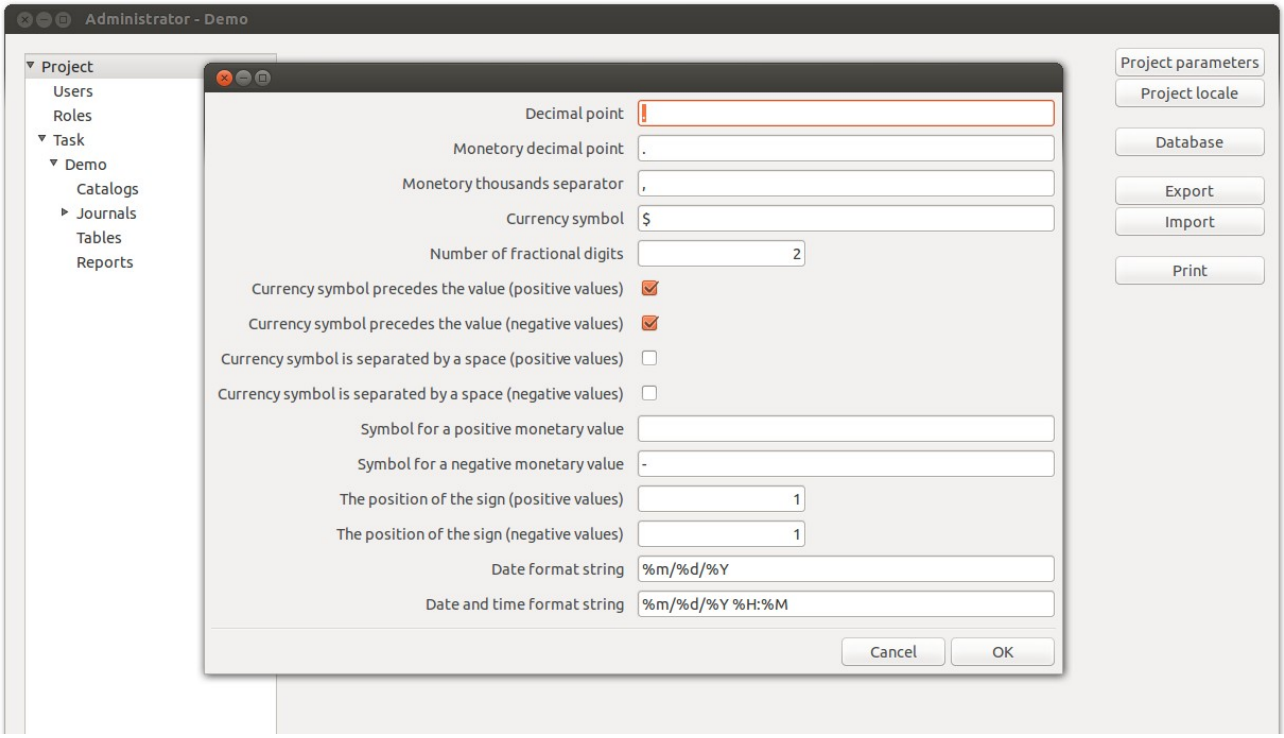
Для изменения параметров проекта надо выбрать на правой панели ветку «Проект» (Project) и щелкнуть по кнопке «Параметры проекта» (Project parameters).



- И внести изменения в параметры проекта:
- Safe mode — безопасный режим. Если установлен безопасный режим, то для работы в

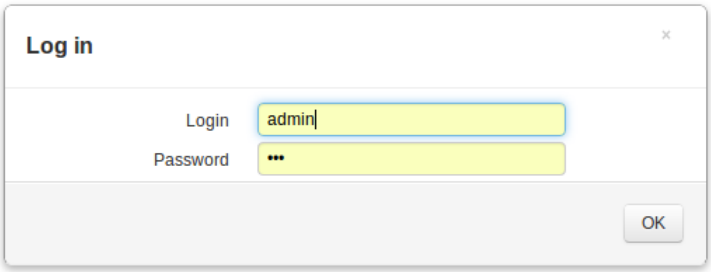
системе необходима аутентификация пользователя (см. ниже «Пользователи и роли»).

- Log file — имя log файла. Если задан log файл, то вывод в stdout/stderr перенаправляется в этот файл
- Connection pool size — размер пула подключений к базе данных.



3.10 Пользователи и роли.

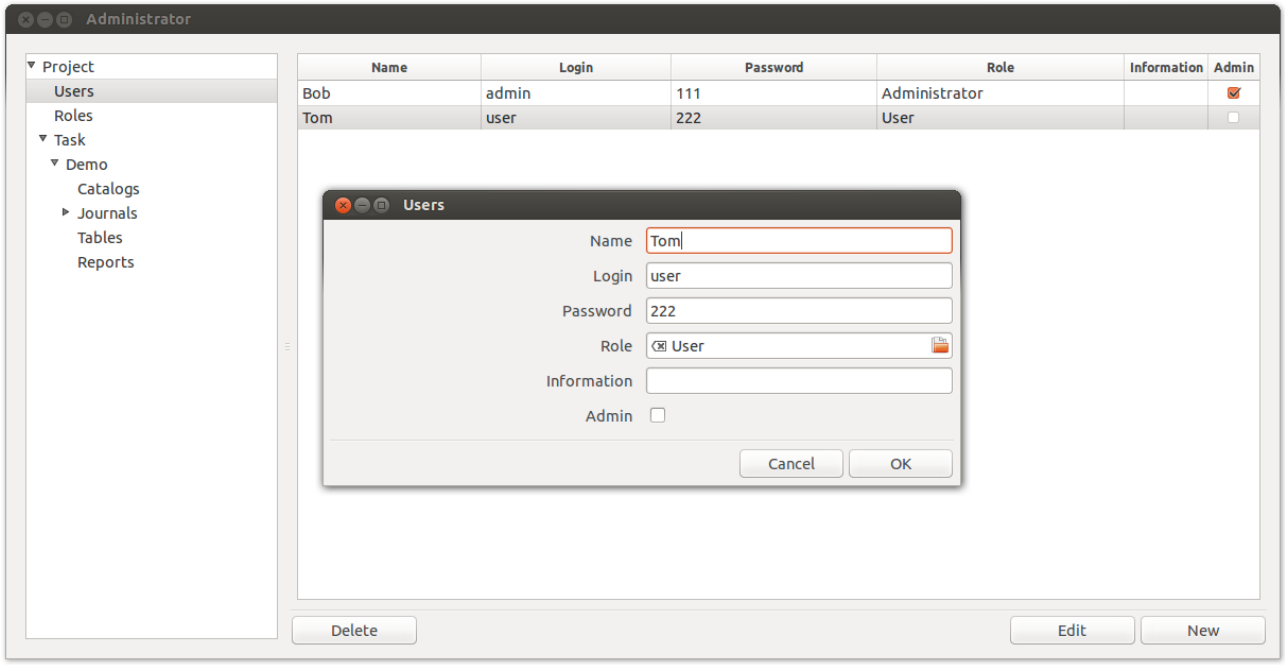
Если в параметрах проекта был устанвлен атрибут безопасный режим (safe mode), то, после перезапуска сервера, для работы на клиенте пользователь должен пройти аутентификацию: ввести свой логин и пароль.



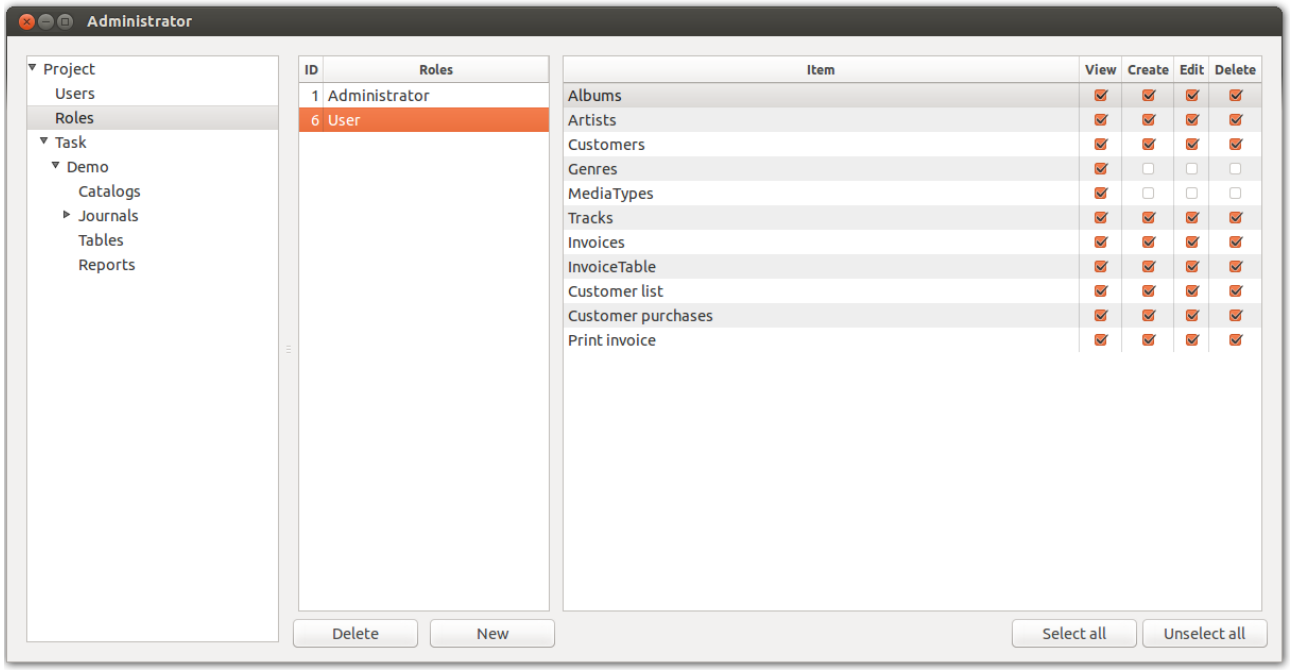
Но перед этим этот пользователь должен быть зарегистрирован в системе. Для регистрации пользователя надо выбрать на правой панели ветку «Пользователи» (Users) и щелкнуть по кнопке Добавить (New). После чего ввести:

- Name - имя пользователя
- Login - логин
- Password - пароль
- Role - роль пользователя (см. ниже)
- Information - некоторая дополнительная информация

- Admin — если этот флаг установлен, то пользователь имеет право работать в Администраторе проекта.



Каждому пользователю должна быть присвоена одна из множества ролей заданных в системе. Роль определяет права пользователь по просмотру, созданию, изменению и удалению данных задачи. Для работы с ролями надо выбрать в дереве проекта ветку «Роли» (Roles). Для добавления и удаления роли воспользуйтесь кнопками «Добавить» (New) и «Удалить» (Delete). Для задания разрешений пользователю в крайней левой панели поставьте напротив соответствующего item'а галочку в колонках View, Create, Edit, Delete (разрешено просматривать, создавать, изменять, удалять соответственно) .



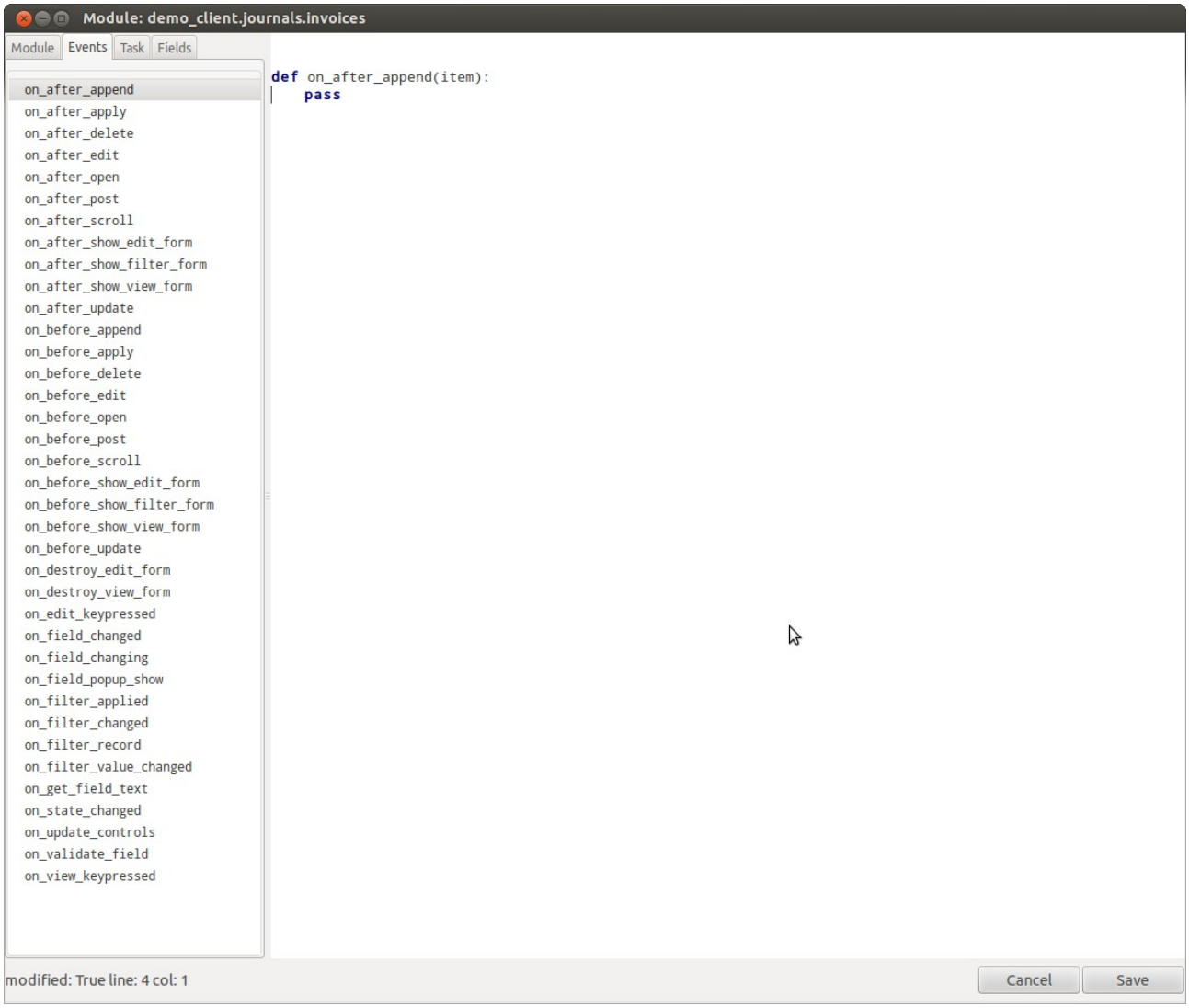
4 Программирование jam.py.

В предыдущей главе мы создали все необходимые структуры данных. Теперь для того, чтобы закончить проект нам надо разобраться как программируется jam.py. Сейчас для того, чтобы проиллюстрировать основные принципы программирования в jam.py мы, для журнала Invoices, определим обработчик события on_after_append, которое срабатывает сразу после добавления новой записи в журнал. Выделим ветку «Журналы» и щелкнем по записи Invoices. На правой панели вверху находятся 3 кнопки «Модуль клиента» (Client module), «Модуль web-клиента» (WebClient module), «Модуль сервера» (Server module). Щелкнем по кнопке «Модуль клиента». На экране откроется окно редактора событий журнала Invoices.

4.1 Редактор событий.

В редакторе событий слева располагается информационная панель с 4 вкладками.

- Module - на вкладке отображаются все определенные в редакторе события и функции, по двойному щелчку курсор переходит на соответствующую функцию,
- Events - отображаются все опубликованные для журнала Invoices события, по двойному щелчку генерируется оболочка для события (см. рисунок),
- Task - дерево задачи, по двойному щелчку под курсором будет напечатано имя ветки,
- Fields - список полей текущего item'a, по двойному щелчку под курсором будет напечатано имя поля.



Выберем вкладку Events и дважды щелкнем по событию on_after_append. В редакторе будет создана функция `on_after_append`. Отметим, что все события в jam.py начинаются на on_, а в качестве параметра, как правило, передается сгенерировавший это событие объект. Напишем тело этой функции:

```
import datetime


def on_after_append(item):
    item.invoicedate.value = datetime.datetime.now()
```

Этот код означает, что сразу после добавления новой записи, значение поля invoicedate будет равно текущей дате. Сохраним этот текст, нажав на клавишу «Сохранить» (Save). Нажмем на клавишу «WebClient Module» и создадим аналогичный обработчик на javascript:

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

В заключение, приведем сочетания клавиш редактора:

- Ctrl+S — сохранить,
- Ctrl+F — поиск,

- Ctrl+H — замена,
- Ctrl+L — перейти на строку,
- Ctrl+I — увеличить отступ выделенных строк,
- Ctrl+U — уменьшить отступ выделенных строк,
- Ctrl+E — переключить комментирование выделенных строк,

4.2 Дерево задачи.

После запуска сервера на основе метаданных из базы admin.sqlite сервер строит объект task, который является деревом. Причем каждый объект этого дерева, в том числе и task происходят от класса AbstractItem. Когда пользователь запускает клиента, клиент грузит с сервера данные, на основе которых стоит аналогичное дерево. Все эти объекты (items) имеют общие атрибуты:

- ID - уникальный ID item'a в фреймворке ,
- owner - родитель,
- task - задача,
- items - список непосредственных детей,
- item_type - тип - одно из значений "task", "catalogs", "journals", "tables", "reports",
- "catalog", "journal", "table", "report", "detail",
- item_name - имя,
- item_caption - наименование,

и методы:

- find(name) — ищет среди непосредственных детей item с item.item_name равным name и возвращает его, если не находит возвращает None (undefined),
- item_by_ID(ID) - ищет среди всех детей item с item.ID равным ID и возвращает его, если не находит возвращает None (undefined).

Так следующий код:

```
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)
```

Выдаст:

```
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
```

Кроме того каждый item является атрибутом своего owner и все справочники, журналы и таблицы атрибутом task. И поэтому:

```
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)
```

Напечатает:

```
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 Программирование данных.

Все справочники, журналы, таблицы и прилинкованные таблицы (item'ы с item_type "journal", "table", "report", "detail") имеют доступ к соответствующим таблицам из базы данных проекта, могут получать и изменять эти данные.

В качестве примера приведем, следующую функцию, распечатывающую фамилии и имена клиентов:

```
def print_customers(customers):
```

```
    customers.open()
    for c in customers:
        print c.fisrtname.value, c.lastname.value
```

В функцию передается item customers. В результате работы метода open на сервере генерируется и выполняется sql запрос, который возвращает список записей. Затем организуется цикл по всем записям и для каждой записи на терминал выводится имя и фамилия клиента. Эта функция будет работать как на клиенте так и на сервере. Для web клиента этот функционал реализуется следующим образом:

```
function print_customers(customers) {
    customers.open();
    customers.each(function(c) {
        console.log(c.firstname.value, c.lastname.value);
    })
}
```

4.3.1 Поля.

Все item'ы работающие с данными имеют атрибут fields. Fields это список полей. Каждое поле имеет следующие атрибуты:

- ID - уникальное ID поля в фреймворке,
- owner - item владелец поля,
- field_name - имя поля,
- field_caption - наименование,
- field_type - тип одно из значений "text", "integer", "float", 'currency', "date", "datetime", "boolean", "blob",
- field_size - длина поля для полей типа "text",
- required - поле должно быть заполнено при вводе (контролируется на программном уровне),
- read_only - поле для чтения (не доступно для изменения в интерфейсе),
- is_default - поле по умолчанию,
- lookup_item - для lookup полей, полей хранящих id записи другого item, ссылка на этот item.
- lookup_field - имя поля lookup item
- master_field - ссылка на master field поле.

Для доступа к данным поля обладают следующими свойствами (property):

- value - позволяет получать и изменять значение поля для текущей записи, тип значения соответствует типу поля. Так для полей типа integer, float и currency, если заничение поля для записи в таблице базы данных равно NULL, то значение value равно 0, для того чтобы получить не приведенное к типу значение надо использовать свойство raw_value,
- text - текстовое значение поля (с учетом параметров локализации),

- lookup_value - для lookup полей, значение lookup поля для текущей записи,
- lookup_text - текстовое значение lookup поля,
- display_text – для item'a определено событие on_get_field_text и возвращаемое значение не равно None (undefined), то берется это значение, в противном случае text для простых полей и lookup_text для lookup полей.

```
    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"
```

В примере приведенном выше значение lookup_text поля customer равно Hämäläinen, а значение display_text - Terhi Hämäläinen, так как для журнала Invoices определен обработчик события on_get_field_text:

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


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

4.3.2 Фильтры.

Ранее для журнала Invoices мы создали фильтры. Теперь покажем как можно их программировать.

Каждый item имеет аттрибут filters — список фильтров созданных в администраторе. Каждый фильтр имеет следующие атрибуты:

- owner – item, которому принадлежит фильтр,
- filter_name - имя фильтра,
- filter_caption - наименование фильтра, используется при визульном представлении на клиенте
- filter_type - тип фильтра, задается при создании фильтра в Администраторе
- visible - если аттрибут visible имеет значние true, то при создании фильтров на клиенте с помощью функции item'a create_filter_entries, создается визульное представление

данного фильтра.

- value - значение фильтра, для того, чтобы обнулить фильтр надо присвоить ему значение None (python) или null (javascript)

По имени фильтра можно получить доступ к самому объекту фильтр:

- Python на клиенте и сервере:

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

- Javascript:

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

В приведенном выше примере изменяется значение фильтра invoicedate1.

Также получить доступ к фильтру можно с помощью функции filter_by_name item'a которому этот фильтр принадлежит.

4.3.3 Получение данных.

Для получения данных , помимо выполнения непосредственного SQL запроса к базе данных, который будет описан в главе программирование сервера, используется метод open.

- Python на клиенте и сервере: `open(self, expanded=None, fields=None, where=None, order_by=None, open_empty=False, params=None, offset=None):`
- Javascript: `.open(options, callback)`

Для javascript порядок параметров не имеет значения. Параметр options – это объект (словарь), атрибуты которого соответствуют параметрам функции на python, с такими же значениями по умолчанию.

Если метод вызван на клиенте, то на сервер отправляется запрос с параметрами вызова.

На сервере, на основании параметров, генерируется и выполняется SQL запрос, результат которого - список записей передается item'y, вызвашему метод open. Если для item'a на сервере определено событие on_select, то параметры можно перехватить и самостоятельно сформировать список записей (см. программирование сервера).

Все запросы для клиента на python выполняются синхронно. Для клиента на javascript все зависит от параметра callback. Если в вызове функции open отсутствует парамер-функция, то запрос выполнятся синхронно, в противном случае запрос выполняется ассинхронно и после, того как записи будут получены, будет выполнена эта функция.

Параметр fields представляет собой список имен полей и задает поля для которых будут получены данные. Если параметр не задан, то данные будут получены для всех полей.

Если параметр expanded имеет значение true (по умолчанию), то в записях для lookup полей будут возвращены не только их значения (value) , но и lookup значения (lookup_value). В противном случае, lookup значения не возвращаются.

Параметр where определяет фильтрацию записей в sql запросе на сервере. Если этот параметр не задан, то по умолчанию записи фильтруются в соответствии с значениями заданными для фильтров (при их наличии) описанных выше. Параметр where представляет собой словарь, ключами которого являются имена полей за которыми после двойного подчеркивания следует символ фильтрации. В фреймворке определены следующие символы для фильтрации значения поля:

- 'eq' — равно,

- 'ne' - не равно,
- 'lt' - меньше чем,
- 'le' - меньше или равно,
- 'gt' - больше чем,
- 'ge' - больше или равно,
- 'in' — к значению поля применяется SQL оператор IN,
- 'not_in' - SQL оператор NOT IN,
- 'range' - SQL оператор BETWEEN,
- 'isnull' - SQL оператор IS NULL,
- 'exact' - точное равенство,
- 'contains' - значение поля содержит,
- 'startwith' - значение поля начинается с,
- 'endwith' - значение поля заканчивается на,

Для символа фильтрации '__eq' может быть опущен. Например {'id': 100} эквивалентно {'id__eq': 100}

Пример использования параметра where:

- Python на клиенте и сервере:

```
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});
```

Вызов метода set_where перед выполнением метода open аналогично заданию параметра where:

- Python на клиенте и сервере: `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();
```

После выполнения метода open фильтрация записей задаваемая методом set_where сбрасывается.

Если параметр order не задан, то возвращаемые записи отсортированы согласно порядку заданному в Администраторе - кнопка Order (Порядок). Параметр order представляет собой список имен полей. Если перед именем поля стоит знак '-', то по этому полю записи будут отсортированы по убыванию:

- Python на клиенте и сервере:

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

- Javascript:

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

Переопределить упорядочивание записей можно, выполнив перед методом open метод set_order_by.

Например:

- Python на клиенте и сервере: `set_order_by(self, *fields)`

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

- Javascript: `function .set_order_by(fieldList)`

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

После выполнения метода open упорядочивание записей задаваемое методом set_order_by сбрасывается: если перед следующим вызовом open не вызывается set_order_by, то записи будут отсортированы по умолчанию.

На результат возвращаемый методом open влияет значение аттрибута auto_loading. Если его значение равно True, то в результате работы метода open будут возвращены не все записи соответствующие параметрам, а только первые, количество которых соответствует значению атрибута limit начиная с номера равного значению параметра offset. Этот механизм используется для автоматической подгрузки значений во время просмотра в гриде.

После выполнения этого метода свойство active будет установлено в значение True.

Узнать количество записей можно с помощью метода record_count.

Метод open инициализирует структуры для работы с данными и должен быть выполнен перед вызовом любых других методов для работы с данными.

4.3.4 Копирование и клонирование item'a.

Item'ы работающие с данными имеют метод copy:

- Python на клиенте и сервере:

```
def copy(self, filters=True, details=True, handlers=True):
```

- Javascript:

```
.copy: function(options)
```

Параметр options у метода на javascript представляет собой объект (словарь), атрибуты

которого соответствуют параметрам функции на python, с такими же значениями по умолчанию.

Этот метод создает копию объекта, обладающую всеми атрибутами и методами объекта. Копируются поля. Кроме того, можно задать следующие параметры:

- filters - если равен True (по умолжанию), то копируются фильтры,
- details - если равен True (по умолжанию), то копируются подчиненные таблицы,
- handlers - если равен True (по умолжанию), то копируются определенные для объекта события,

Полученный в результате копирования объект не добавляется в дерево задачи, оно остается неизменным.

Пример:

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

Помимо копирования можно создать клон объекта:

- Python на клиенте и сервере:

  ```
  def clone(self, keep_filtered=True):
  ```

- Javascript:

  ```
  clone: function(keep_filtered)
  ```

В этом случае создается копия (не копируются фильтры, подчиненные таблицы и обработчики событий), которая имеет общий набор записей с объектом. Если значение параметра keep_filtered равно True и для объекта определена фильтрация (определено событие on_filter_record и атрибут filtered равен True) то эта фильтрация поддерживается и для клона.

4.3.5 Навигация по записям.

После получения данных курсор (указатель на текущую запись) установлен на первой записи). Изменить позицию курсора можно с помощью следующих методов:

- first - в результате выполнения курсор переходит на первую запись
- last - курсор переходит на последнюю запись
- next - курсор переходит на следующюю запись
- prior - курсор переходит на предыдущюю запись
-

Кроме того определены методы eof и bof:

- eof - возвращает true, если список записей пуст или был вызыван метод last или в результате вызова next делается попытка выйти за пределы списка записей
- bof - возвращает true, если список записей пуст или был вызыван метод fist или в результате вызова prior делается попытка выйти за пределы списка записей

С помощью свойства rec_no получить или установить значение номера текущей записи.

Например следующий код сохраняет текущее положение курсора печатает список

клиентов, после чего курсор устанавливается в первоначальную позицию.

- Python на клиенте и сервере:

```
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;
```

Определена более короткая форма перебора записей:

- Python на клиенте и сервере:

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

эквивалентно

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

- Javascript:

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

эквивалентно

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

Выше c и customers эквивалентны – являются указателями на один и тот же объект. Для выхода из цикла в javascript надо в функции возвратить false.

При изменении положения курсора, перед изменением вызывается (если он определен) обработчик события on_before_scroll, после изменения – обработчик события on_after_scroll.

4.3.6 Изменение данных.

Для изменения текущей записи надо войти в режим редактирования вызвав метод edit, изменить значения полей и сохранить изменения выполнив метод post.

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

Добавление осуществляется аналогично, только вместо edit надо вызвать метод append

(добавление записи вконец списка) или insert (вставка первой записи). При этом item переводится в режим добавления.

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

Если изменяемые данные сохранять не надо, то для отмены изменений вместо post надо вызвать метод cancel.

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

Для удаления надо вызвать метод delete.

```
    invoices.delete()
```

Такие изменения производятся над текущим набором записей и никак не влияют на значения, хранимые в базе данных. Если значение аттрибута log_changes item'a равно True (значение по умолчанию), то поддерживается лог накапливающий изменения, которые можно сохранить в базе данных вызвав метод apply.

В следующем примере удаляются все записи из списка записей, после чего они удаляются из базы.

```
item.first();
    while not item.eof():
        item.delete()
item.apply()
```

Перед вызовом каждого из этих методов вызывается обработчик события (если определен) on_before + имя метода (on_before_apply для apply), после вызова - обработчик события on_after + имя метода.

4.3.7 Работа с подчиненными таблицами.

Если item имеет подчиненные таблицы (details) и значение атрибута details_active равно True, то, при переходе на другую запись, записи подчиненных таблиц автоматически обновляются (происходит их переоткрытие). В противном случае их надо открывать самим.

```
invoices.details_active = False
for inv in invoices:
    inv.invoice_table.open()
```

Значение details_active по умолчанию равно False.

Подчиненные таблицы имеют атрибут disabled, имеющий значение по умолчанию False. Если значение disabled равно True, то обновление таблицы не происходит.

Для изменения записей подчиненной таблицы, item владелец должен быть переведен в состояние редактирования.

```
invoices.edit()
for t in invoices.invoice_table:
    t.edit()
    t.date.value = invoices.date.value
    t.post()
invoices.post()
invoices.apply()
```

За сохранение изменений подчиненных таблиц на сервере отвечает item владелец.

4.4 Программирование клиента.

4.4.1 Главная форма.

При запуске клиента на python после получения данных с сервера и инициализации дерева задачи создается главная форма приложения. И перед тем как эта форма будет отображена на экране клиент генерирует событие on_before_show_main_form.

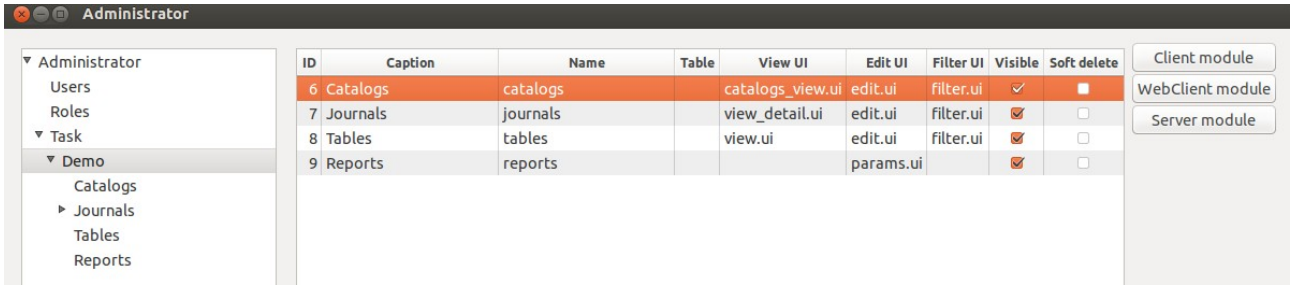
Аналогично в браузере после загрузки DOM выполняется инициализация дерева задачи и также генерируется событие on_before_show_main_form.

4.4.2 Формы.

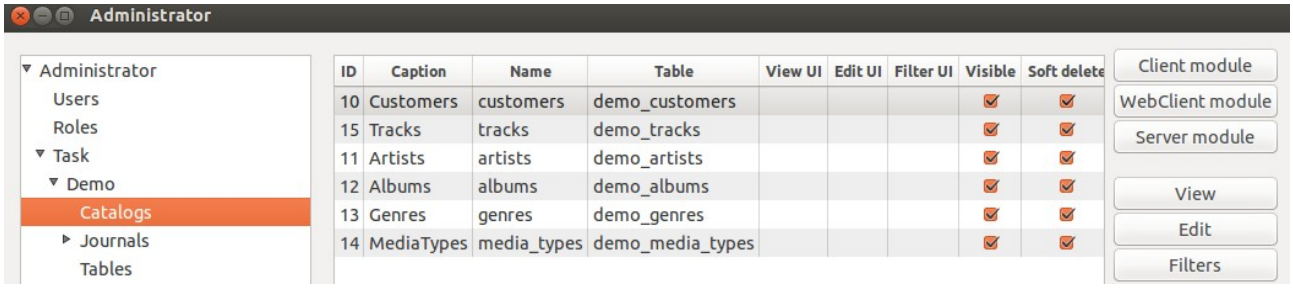
Одним из основных понятий фреймворка для клиента является понятие формы. Формы строятся на шаблонах. Для каждого item' а, работающего с данными, могут быть заданы шаблоны форм просмотра, редактирования и шаблон формы фильтра. Для отчетов - шаблон формы задания параметров. Для задачи главная форма приложения.

4.4.2.1 Формы клиента на python и pygtk.

В директории проекта в папке ui находятся шаблоны форм клиента. Шаблоны форм это файлы интерфейсов, созданных в Glade - редакторе пользовательского интерфейса для GTK+ и GNOME. Шаблоны форм задаются в Администраторе. На рисунке ниже колонки View UI, Edit UI и Filter Ui - шаблоны форм просмотра, редактирования и фильтра.

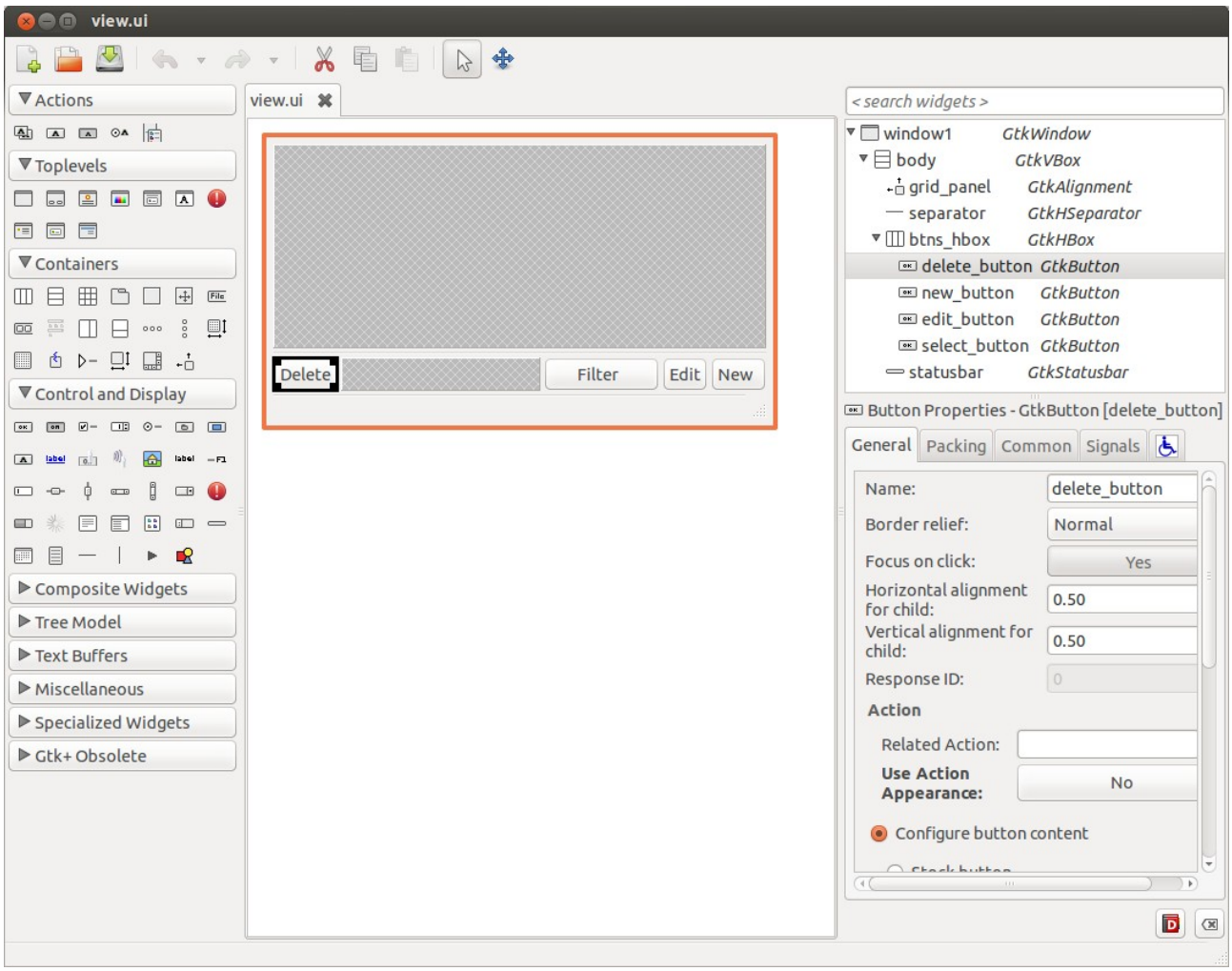


Если для item' а не задан шаблон, то он ищется у его владельца. Так если в дереве задачи выбрать ветку Catalogs, то мы увидим, что для справочников шаблоны не заданы. И таким образом все справочники имеют общие шаблоны форм, заданные для группы Catalogs.



Если для какого-то справочника требуется отличный от других шаблон, то его надо создать в Glade, сохранить в папке ui и в окне редактирования атрибутов этого справочника указать для данного шаблона имя файла интерфейса.

На рисунке ниже изображен шаблон формы view.ui в редакторе Glade. Обратите внимание, что имя главного окна шаблона должно быть window1.



После создания форм просмотра, редактирования и фильтра получить доступ к виджетам на этих формах можно с помощью аттрибутов view_form, edit_form и filter_form соответственно. Например

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

сделает кнопку Delete на форме просмотра невидимой.

Аналогичным атрибутом для главной формы является main_form. Для формы задания параметров отчета - params_form.

Кроме того эти атрибуты являются обертками (wrapper) над gtk.Window. Получить доступ к самому gtk.Window можно через атрибут window формы. Код ниже разворачивает форму просмотра на весь экран:

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

Кроме того атрибут форм builder является объектом gtk.Builder. Так что можно подключить обработчики сигналов, описанные в ui-шаблоне:

```
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 Формы клиента в браузуре.

В файле index.html в директории проекта внутри тега body div с классом templates содержит в себе шаблоны проекта.

```
<!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>
```

После инициализации задачи на клиенте этот div вырезается из страницы, а у задачи появляется свойство templates, которое является jquery объектом хранящим этот div. Так в demo в событии on_before_show_main_form строка

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

добавляет меню в content.

Понятие формы для javascript клиента довольно условное — это div описывающий форму представления данных item'a в браузере. Для того, чтобы добавить шаблон формы для item'a надо в div с классом templates в файле index.html создать div с классом name-suffix, где name это имя item'a, a suffix - это тип формы: view, edit, filter, params. Например:

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

Для подчиненной таблицы перед именем через тире должно быть указано имя владельца:

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

Если шаблон формы item'a не задан, то будет взят шаблон владельца. Так шаблон

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

будет использован для создания формы редактирования item'a справочника, у которого не задан свой собственный шаблон.

При создании форм просмотра, редактирования, фильтра и парамеров отчета в item' е создаются свойства view_form, edit_form, filter_form и params_form соответственно. Эти свойства являются jquery объектами. Например

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

сделает кнопку Delete на форме просмотра невидимой.

4.4.3 Методы клиента.

4.4.3.1 Метод view.

В обработчике события on_before_show_main_form в Demo создается главное меню приложения. При этом при щелчке мышью на элементы меню вызывается метод view соответствующего item:

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

Если для клиента на javascript параметр container не задан, то форма просмотра будет создана в модальном окне, в противном случае будет добавлена в container (container – это объект jquery).

В ходе выполнения метода view

- сначала ищется шаблон просмотра item'a, на основании которого создается объект view_form
- если для задачи опреден обрабочик события on_before_show_view_form, то вызывется этот обработчик, которому этот item передается как параметр
- если определен, вызывается обрабочик события on_before_show_view_form владельца-группы , то вызывется этот обработчик, которому этот item передается как параметр
- если определен, вызывается обрабочик события on_before_show_view_form самого item' a.
- окно визульно отображается на экране
- если для задачи опреден обрабочик события on_after_show_view_form, то вызывется этот обработчик, которому этот item передается как параметр

- если определен, вызывается обрабочик события on_after_show_view_form владельца-группы , то вызывется этот обработчик, которому этот item передается как параметр
- если определен, вызывается обрабочик события on_after_show_view_form самого item' a.

Ниже приведен код метода view для клиента на 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 Методы append_record, insert_record, edit_record.

В demo, в событии on_before_show_view_form задачи, кнопкам Добавить, Удалить, Изменить подвязываются методы обработчики: insert_record, delete_record, edit_record.

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

В методах append_record, insert_record, edit_record сначала выполняется методы append, insert, edit соответственно, переводящие item в режим добавления или редактирования, после чего вызывается метод create_edit_form, в котором создается форма редактирования.

В ходе выполнения метода create_edit_form

- сначала ищется шаблон редактирования item'a, на основании которого создается объект edit_form
- если для задачи опреден обрабочик события on_before_show_edit_form, то вызывется этот обработчик, которому этот item передается как параметр
- если определен, вызывается обрабочик события on_before_show_edit_form владельца-группы , то вызывется этот обработчик, которому этот item передается как параметр
- если определен, вызывается обрабочик события on_before_show_edit_form самого

item' a.

- окно визульно отображается на экране
- если item имеет подчиненные таблицы и атрибут detail_active равен False, то выполняется метод open подчиненных таблиц
- если для задачи опреден обрабочик события on_after_show_edit_form, то вызывется этот обработчик, которому этот item передается как параметр
- если определен, вызывается обрабочик события on_after_show_edit_form владельца-группы , то вызывется этот обработчик, которому этот item передается как параметр
- если определен, вызывается обрабочик события on_after_show_edit_form самого item' a.

4.4.3.3 Методы post_record и apply_record.

Для сохранения результатов редактирования записи в окне редактирования в приложении demo используется метод apply_record. В время выполнения этого метода, в случае если запись была изменена, то вызывается метод post и затем запись сохраняется на сервере в результате выполнения метода apply, в противном случае вызывается метод cancel. И в конце закрывается окно редактирования.

В методе post_record выполняются аналогичные действия за исключением сохранения данных на сервере.

4.4.3.4 Метод delete_record.

При выполнении этого метода, если свойство item'a read_only не равно false, и при подтверждении удаления, выполняется метод delete, после чего запись удалается на сервере методом apply.

4.4.3.5 Метод create_grid.

Метод create_grid позволяет создать табличное представление записей item'a:

- 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
    };
```

Ниже опишем основные параметры для javascript грида:

- container - элемент контейнер, который будет содержать в себе таблицу, для web клиента - это объект Jquery, для pygtk — gtk объект,
- fields - список имен полей, если параметр задан, то в таблице будут созданы колонки для редактирования полей с именами из этого списка, если не задан (по умолчанию) то берутся поля просмотра заданные в Администраторе (кнопка «Просмотр»),
- striped - если значение параметра true, то строки выводятся черезстрочно полосатыми,
- dblclick_edit - если значение параметра равно true (по умолчанию), то двойной щелчок по строке таблицы активизирует редактирование записи,
- on_dblclick - позволяет задать процедуру, выполняемую по двойному щелчку по сроке таблицы,
- multi_select - если значение этого параметра равно true, то в гриде появляется крайняя левая колонка с чекбоксом для выбора записи. При этом, если значение передаваемой в параметрах функции multi_select_get_selected для этой записи будет возвращать true, то чекбоксе будет помечен галочкой. При щелчке по чекбоксу будет вызываться заданная в параметрах функция multi_select_set_selected с параметром, передающим состояние чекбокса и дающим возможность сохранить состояние записи. Если задан параметр-функция multi_select_select_all то чекбокс будет создан в заголовке крайнего левого столбца и эта функция будет вызвана по щелчку по этому чекбоксу. В примере ниже для item'a tracks параметр multi_select устанавливается в значение true, создается словарь selected_records, который будет хранить информацию о выделенных записях, и который будут использовать параметры функции multi_select_get_selected и multi_select_set_selected.

```
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 - если значение этого параметра равно true, то возможно редактирование полей item'a в гриде. При этом если параметр editable_fields не задан (по умолчанию), то возможно редактирование всех полей, в противном случае, только полей имена, которых перечислены в этом параметре. Если парметр always_show_editor равен true, то редактор поля присутсвует всегда, в противном случае (по умолчанию) для перевода грида в режим редактирования надо нажать на клавишу Enter или если keypress_edit равен true (по умолчанию), нажать на любую клавишу. В параметре selected_field модно задать имя поля, которое будет выделено при создании грида. Сохранение нового значения просходит при нажатии клавиши Enter или при переходе на другую запись и новое значение сохранается локально, для сохранения на сервере необходимо выполнить метод apply.
- sortable - если этот параметр задан, то возможно выполнить сортировку записей щелчком по заголовку столбца, при этом если параметр sort_fields не задан (по умолчанию), то возможно сортировка по любому полю, в противном случае, только полям, имена которых перечислены в этом параметре. Сортировка выполняется на сервере.
- auto_fit_width - если этот параметр true, то грид стремиться отображать все поля без использования горизонтальной прокрутки, в том числе и при изменении размера колонки.
- expand_selected_row - если параметр word_wrap равен true и значение expand_selected_row больше 0, то в случае если текст поля выделенной записи не вмещается в ячейку грида, высота ячейки увеличивается так, чтобы текст полность вместился в ячейку. Значение expand_selected_row задает минимальную высоту (количество строк) выделенной строки грида

Обратите внимание, что поведение грида определяется атрибутом auto_loading item'a. Если значение auto_loading равно true питоновский грид по мере необходимости подгружает записи с сервера в соответствии с значением атрибута limit. Если значение auto_loading равно true, то грид клиента на javascript создает паджинатор и по заданным параметрам расчитывает значение атрибута limit item'a. Если значение auto_loading равно false, то грид просто отображает все записи item'a.
Этот метод возвращает объект DBGrid.

4.4.3.6 Метод create_entries.

Метод create_entries позволяет создавать визуальные элементы интерфейса для редактирования полей item'a:

- 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
    };
```

В качестве параметров в метод передаются:

- container - элемент контейнер, который будет parent' ом, для web клиента — это объект Jquery, для pygtk — gtk объект,
- fields - список имен полей, если параметр задан, то будут созданы элементы для редактирования полей с именами из этого списка, если не задан (по умолчанию) то берутся поля редактирования заданные в Администраторе (кнопка «Редактирование»)
- col_count - количество колонок в которых будут созданы элеметы редактирования, по умлочанию 1. (В Демо при редактировании накладной col_count = 2)
- tabindex — если tabindex задан, то он будет tabindex' ом первого input элемента, у всех последующих элементом tabindex будет увеличиваться на 1.

4.4.3.7 Взаимодействие данных и визуальных элементов клиента.

По умолчанию любые изменения данных item'a сразу же отображаются в визульных элементах клиента — гридах, entries и input'ах. Но иногда необходимо отключить эту связь. Отключить и подключить эту связь можно с помощью методов disable_controls и enable_controls соответственно. Обновить данные визуальных элементов можно с помощью метода update_controls (при этом грида будет пересоздан). Узнать о состоянии визуальных элементов можно с помощью методов controls_enabled и controls_disabled.

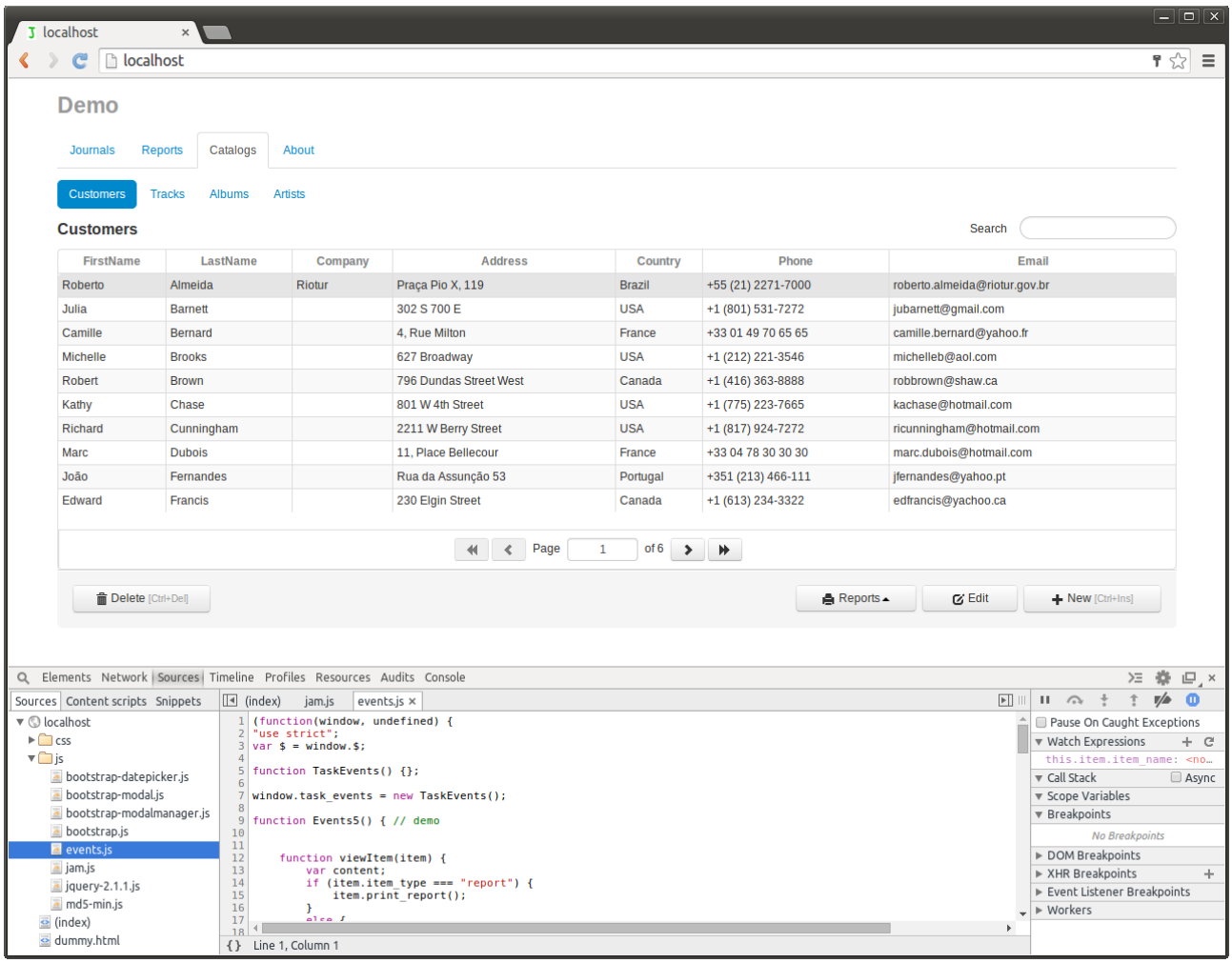
Например:

```
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
```

Выше сначала для подчиненной таблицы invoice_table сохраняется номер текущей записи и отключаются ее визуальные элементы, затем в цикле по всем записям перерассчитываются значения полей и рассчитываются суммарные значения, после чего курсор возвращается на первоначальную запись, подключаются ее визуальные элементы и происходит их обновление.

4.4.3.8 Отладка web клиента.

После сохранения изменений в модуле web клиента, фреймворк на основе всех web модулей задачи формирует файл events.js и сохраняет его в папке js директории проекта. Этот файл содержит, соответствующим образом структурированные, все события проекта.



Выше представлен проект Demo в браузере Chrome.

4.5 Программирование сервера.

4.5.1 SQL запросы.

При создании задачи на сервере, создается пул соединений к базе данных, использующий модуль multiprocessing, доступ к которому осуществляется через очередь запросов. Для выполнения sql-запросов через пул соединений задача имеет следующие методы:

- execute
- execute_select

Для выполнения select запросов можно использовать метод задачи execute_select:

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

где sql — SQL запрос. Метод возвращает список записей. Например:

```
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')))
```

Здесь запрос выполняется в отчете.

Для остальных запросов исрользуется метод execute:

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

этот метод возвращает typle - (success, result, error). В случае успешного выполнения success равно True, в противном случае False и error содержит текст ошибки, result — результат выполнения запроса. Параметром sql может быть как сам запрос, так и список запросов. Если sql — это запрос, то в params могут передаваться параметры запроса. Запрос выполняется в одной транзакции, после чего выполняется коммит.

4.5.2 События на сервере.

Для инициализации данных задачи предназначено событие on_created. После запуска сервера, при первом обращении к нему клиента, на основании хранящихся в файле admin.sqlite структуры данных и кода, строится дерево задачи сервера. Сразу после этого вызывается обработчик события on_created. Например:

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

Для всех item'ов на сервере, работающих с данными можно определить обработчики событий:

- on_select
- on_record_count
- on_apply

Событие on_apply можно использовать, если необходимо переопределить процедуру сохранения данных на сервере во время выполнения метода apply. Это событие имеет следующий вид:

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

В него передаются следующие параметры:

- item - ссылка на item,
- delta - дельта, содержащая изменения (рассмотрим подробднее ниже),
- params - параметры, передаваемые методом apply на сервер,
- privileges - словарь, содержащий информацию о разрешениях ('can_create', 'can_edit', 'can_delete', 'can_view') пользователя,
- user_info - словарь, содержащий информацию о пользователе,
- enviroment - словарь, содержащий стандартные WSGI переменные среды

В параметре delta содержатся изменения, которые необходимо сохранить на сервере. Сам по себе этот параметр представляет собой копию item'a, а набором записей является его лог изменений. При это характер изменений записи можно получить с помощью методов rec_inserted, rec_modified, rec_deleted, каждый из которых возвращает значение True, если запись добавлена, изменена или удалена соответственно. Если item имеет подчиненную таблицу, delta также будет имееть соответствующую таблицу, хранящую изменения. Атрибут details_active у delta равен True. Только обратите внимание, если будет удалена запись у item'a имеющего подчиненные таблицы, то в delta просто будет храниться эта удаленная запись, информация об удаляемых подчиненных записях в delta не сохраняется.

В результате работы функции apply на сервере сначала на основании данных об изменениях генерируется delta, за тем по этой delta генерируется запрос, который передается в метод execute задачи.

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

В результате чего изменения сохраняются в базе данных в одной транзакции. В случае успешного завершения этой транзакции метод apply обновляет лог изменений и, при добавлении новых записей, значения id полей этих записей.

Пример ниже взят из модуля сервера журнала Invoices. В нем при сохранении накладной в той же транзакции для каждого трека накладной перерасчитываются количество проданных штук:

```
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)
```

Выше в событии on_apply процедура process_delta возвращает список sql-запросов, изменяющих количество проданных треков. Затем эти запросы выполняются вместе с запросом, изменяющим данные.

Для того, чтобы переопределить как выполняется функция open на сервере можно использовать событие on_select:

```
    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
```

В событие передаются следующие параметры:

- item - ссылка на item
- params - параметры, передаваемые методом open на сервер
- user_info - словарь, содержащий информацию о пользователе,
- enviroment - словарь, содержащий стандартные WSGI переменные среды

В примере выше выполняются стандартные процедуры, выполняемые при открытии. В методе get_select_statement генерируется sql запрос, который выполняется задачей.

Событие должно возвратить tuple из списока записей и сообщения об ошибке. В списке записей данные для полей в каждой записи должны следовать в таком же порядке, в каком они были указаны в функции open. Сначала в этом порядке должны идти сами значения полей, а следом за ними, если параметр expanded был равен True, за ними должные следовать lookup значения для lookup полей.

Аналогично можно использовать событие on_record_count для переопределения вычисления общего количества записей item' a, которое используется в компоненте DBGrid для создания 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 Функции сервера.

Если для item'a в модуле сервера определена функция, начинающаяся на server_:

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

то на Python - клиенте эта функция может быть вызвана следующим образом:

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

на Javascript – клиенте синхронный вызов функции:

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

асинхронный вызов функции:

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

Если например, в модуле сервера можно определить следующую функцию

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

то на Python - клиенте эта функция может быть вызвана в модуле item'a следующим образом:

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

на Javascript – клиенте синхронный вызов функции:

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

Если функция сервера заканчивается на _env, то в нее передается дополнительный аргумент — словарь, содержащий стандартные WSGI переменные среды. Например на сервере:

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

на клиенте:

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

4.6 Программирование отчетов.

Для вызова печати отчета на клиенте используется метод print_report. В результате вызова этой функции создается форма редактирования параметров отчета. При создании формы редактирования генерируются события on_before_show_params_form последовательно для задачи, отчетов и самого отчета. В Demo в событии on_before_show_params_form задачи щелчок по кнопке Печатать активирует метод отчета process_report. Этот метод, перед отправкой отчета на сервер генерирует событие on_before_print_report сначала для владельца отчета item'a Reports и затем для самого отчета.

На сервере сначала создается копия отчета и затем для этой копии генерируется событие on_generate_report. Наример для отчета invoice это событие имеет следующий вид:

```
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())
```

Здесь сначала создается копия журнала invoices.

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

Копии создаются потому, что несколько пользователей могут одновременно генерировать один и тотже отчет в паралельных потоках.

Затем для этой копии выполняется метод set_where:

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

где report.id.value - это параметр id, значение которого задается на клиенте в событии on_before_print_report и который равен id текущей накладной.

Затем с помощью метода open мы получаем эту запись на сервере. После чего печатается бэнд title:

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

Но перед этим присваиваются значения четырем локальным переменным custome, address, city и date, которые соответствуют программируемым ячейкам на бэнде title в шаблоне отчета.

Затем аналогичным образом генерируются detail бэнды и бэнд summary.

После того как отчет сгенерирован от сохнаняется на сервере в папке report в директории static и на клиент отправляется url файла отчета.

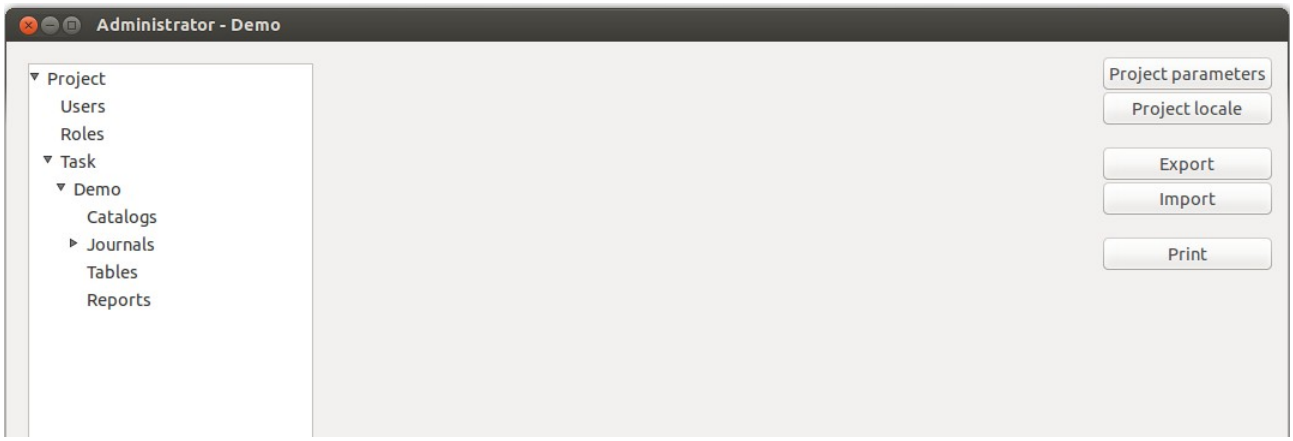
Отчет может быть конвертирован в другой формат, отличный ods. Формат можно задать на клиенте с помощью атрибута отчета extension. Конвертация осуществляется пакетом open

office. Open office может быть запущен на сервере в режиме сервера:

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

5 Утилиты jam.py.

Если в Администраторе на дереве проекта выбрать ветку Project станут доступны кнопки, позволяющие осуществить экспорт (Export), импорт (Import) метаданных проекта и распечатать программный код (Print):



5.1 Экспорт, импорт проекта.

Утилиты экспорта и импорта позволяют сохранить в файле метаданные проекта. При выгрузке в файле сохраняются:

- параметры проекта
- параметры локализации
- роли и их привелегии
- дерево задачи: итемы, их поля, фильтры, в том числе отчеты и их параметры, весь программный код

При импорте файла производится сравнение метаданных текущего проекта с метаданными, сохраненными с файле. При этом, на основе анализа деревьев задачи, находятся различия в структурах баз данных проектов, хранящих данные, и генерируется sql код. После этого делается попытка в одной транзакции осуществить эти изменения. В случае успеха, также в одной транзакции происходит изменение данных в sqlite базах admin.sqlite, хранящих метаданные. Если импорт осуществляется удаленно, то при успешном завершении импорта сервер останавливается. Его необходимо запустить заново.

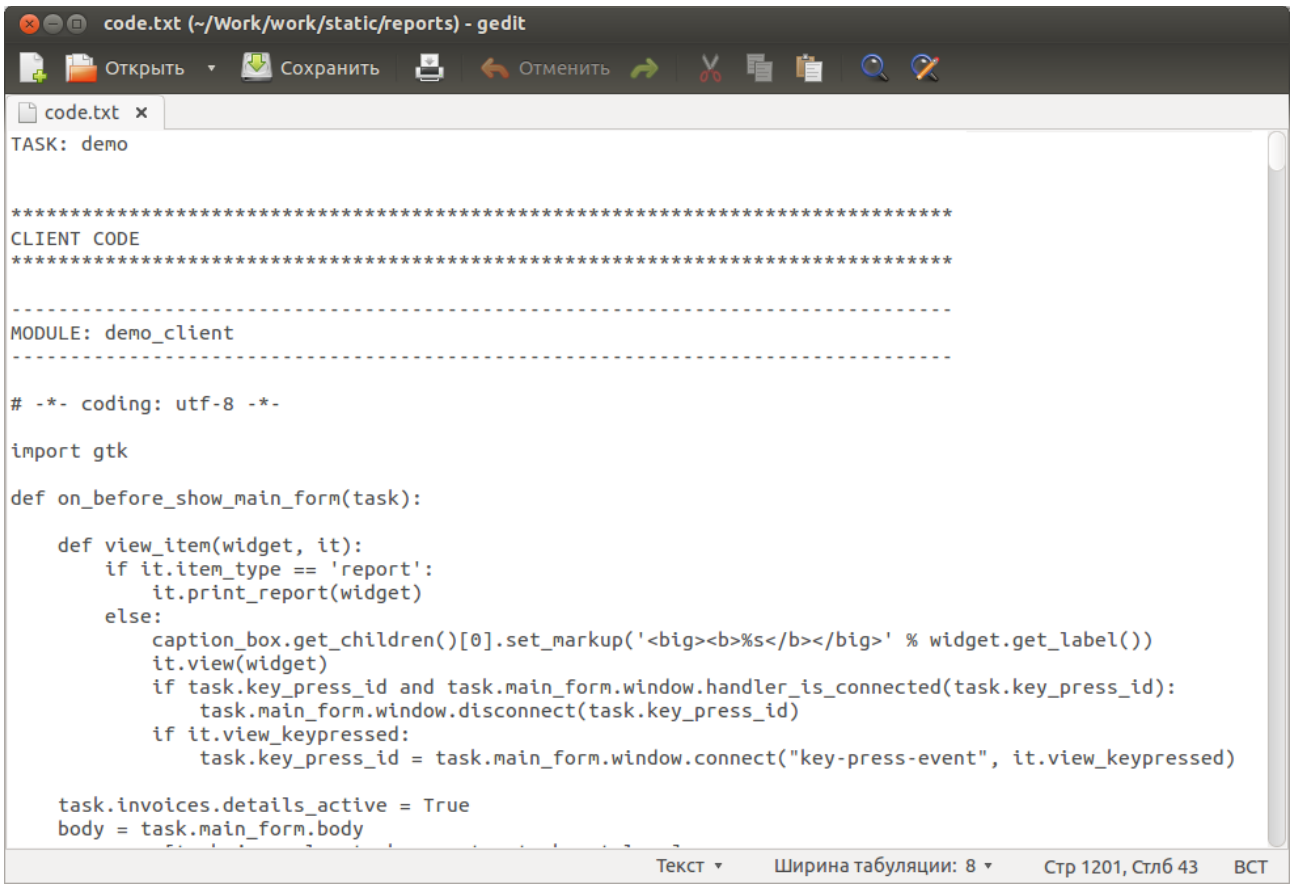
Если, в процессе разработки, изменения базы данных были сделаны вне Администратора, непосредственно в базе данных, то они не отражаются в файле экспорта.

Если база данных проекта — это база sqlite, то импорт проекта не возможен.

При импорте данных будьте очень осторожны. Делайте резервные копии базы данных проекта и файла admin.sqlite.

5.2 Печать кода.

Программный код проекта хранится в множестве модулей. Это может быть неудобно, если надо оснакомиться с кодом всего проекта. Это можно сделать, нажав на клавишу «Печать», при этом весь код будет выведен в одном файле:


code.txt (~/Work/work/static/reports) - gedit
Открыть
Сохранить
Отменить
code.txt
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>%s</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