Jam.py V7 文档

Andrew Yushev
Dean D. Babic
杨文增

2026 年 06 月 11 日

Contents

1 Jam.py V7 文档 1
1.1 简介 1
1.2 提供给 LLM(大预言模型) 的资料 1
1.3 文档结构概览 2
1.4 视频教程 2

2 入门指南 3
2.1 安装 3
2.2 创建项目 4
2.3 演示项目 9
2.4 教程第一部分：第一个项目 11
2.5 教程第二部分：文件和图像字段 46
2.6 教程第三部分：明细表 58
2.7 部署 70
2.8 理解 admin.sqlite 71

3 Jam.py 编程 73
3.1 任务树 73
3.2 工作流程 75
3.3 使用模块 75
3.4 客户端编程 76
3.5 数据编程 99
3.6 服务器端编程 115
3.7 报表编程 116
3.8 保留字 124

4 Jam.py 常见问题 125
4.1 外键的作用是什么？ 125
4.2 主表目录和业务台账的区别 125
4.3 如何将已创建的项目升级到新版本的 Jam.py？ 125
4.4 我可以在应用程序中使用其他库吗？ 126
4.5 打印报表时我得到的是 ods 文件而不是 pdf 126

5 如何... 127
5.1 如何在 Windows 上安装 Jam.py 127
5.2 如何实现多对多关系？ 129
5.3 如何从开发环境迁移到生产环境 130

5.4 如何迁移到另一个数据库 . . . 131
5.5 如何部署 . . . 132
5.6 如何编写具有全局作用域的函数？ . . . 140
5.7 如何验证字段的值 . . . 140
5.8 如何给表单添加按钮 . . . 142
5.9 如何从客户端执行 Python 代码 . . . 142
5.10 如何更改表单中元素的样式和属性 . . . 144
5.11 如何创建自定义菜单 . . . 146
5.12 如何在不打开视图表单的情况下使用编辑表单追加记录？ . . . 146
5.13 如何禁止更改记录 . . . 147
5.14 如何关联两个数据表 . . . 148
5.15 如何批量修改选中记录的字段值 . . . 151
5.16 如何在不关闭编辑表单的情况下保存数据 . . . 154
5.17 如何在服务器端的同一个事务中保存两个表的变更 . . . 154
5.18 如何防止表的字段中出现重复值 . . . 155
5.19 如何实现基础的多用户？例如，让不同用户拥有各自独立的数据。 . . . 156
5.20 我可以让 Jam.py 使用现有的数据库吗 . . . 157
5.21 如何使用其他数据库中的数据 . . . 157
5.22 如何处理来自其他应用或服务的请求，或获取其数据 . . . 160
5.23 如何在后台执行计算 . . . 162
5.24 支持在明细表中嵌套明细表吗？ . . . 163
5.25 CSV 文件的导入与导出 . . . 165
5.26 认证 . . . 167
5.27 如何将 v5 项目迁移到 v7 . . . 175
5.28 如何编写测试 . . . 176
5.29 在表单之间导航并保留数据的方法 . . . 178

6 应用程序构建器 181
6.1 项目管理 . . . 181
6.2 角色 . . . 195
6.3 用户 . . . 196
6.4 代码编辑器 . . . 198
6.5 任务 . . . 200
6.6 组 . . . 201
6.7 实体项 . . . 207
6.8 明细表 . . . 229
6.9 查找列表对话框 . . . 230
6.10 导入现有数据库表 . . . 233
6.11 保存用户所做的审计追踪/变更历史记录 . . . 235
6.12 记录锁定 . . . 239
6.13 语言支持 . . . 240
6.14 语言翻译 . . . 243
6.15 数据转义清理 . . . 244
6.16 可接受的字符串 . . . 246
6.17 路由 . . . 247
6.18 自定义构建器 . . . 249

7 Jam.py 类的参考 251
7.1 客户端的 (javascript) 类参考 . . . 251
7.2 服务端 (python) 的类参考 . . . 386
7.3 Jam.py 异常 . . . 450

8 发行说明 451
8.1 版本 1 . . . 451

8.2 版本 2 451
8.3 版本 3 451
8.4 版本 4 451
8.5 版本 5 452
8.6 版本 7 463
8.7 Jam.py 路线图 465

CHAPTER 1

Jam.py V7 文档

1.1 简介

欢迎使用 Jam.py！如果您是 Jam.py 新手，要进行无代码、低代码或多代码的 Web 应用程序开发，那么在这里可以找到使用 Jam.py V7 的相关文档。

与 Jam.py V5 最大的区别在于增加了路由 支持和采用了 Bootstrap 5，从而实现对移动设备的现代化支持（详见版本 7）。

此外，用于桌面或移动设备的完整界面都采用无代码方式驱动。只需为任意地设备选择所需的功能即可。

目标

安装 Python 和 Jam.py 后，再选择数据库和 Web 服务器，就实现应用程序设计的决策。

读者对象

Web 开发爱好者或开发人员，对 Web 开发或部署经验有限或无经验。

前提条件

建议具备一些 Python 和 JavaScript 知识。需要掌握命令行提示符的基本知识，并能够输入命令执行操作。

1.2 提供给 LLM(大预言模型) 的资料

已发布的 LLMS-full.txt 文件：

https://jampy-docs-v7.readthedocs.io/zh-cn/latest/llms-full.txt

以及：

https://jampy-docs-v7.readthedocs.io/zh-cn/latest/llms.txt

GitHub 上的 Jam.py 代码仓库：

https://github.com/jam-py-v5/jam-py-v7

1.3 文档结构概览

以下是文档结构的概览，以帮助您找到特定内容的所在位置：

入门指南 主题描述了如何安装框架、创建新项目、逐步开发 Web 应用程序并部署到生产环境。

编程指南 在较高层次上讨论关键主题和概念，并提供有用的背景信息和解释。

业务应用程序构建器 是对用于应用程序开发和数据库管理的应用程序构建器的详细描述。

类参考指南 包含关于 Jam.py 类 API 的技术参考。

常见问题 该主题涵盖了最常见的问题。

如何实现操作指南 包含一些代码示例，对于快速完成常见任务可能很有用。

请访问目录，或者查看 Jam.py 应用程序设计技巧，以获取关于如何构建应用程序或从 MS Access 迁移的详细步骤。

要获取本文档的单个 PDF 文件，请访问：https://jampy-docs-v7.readthedocs.io/_/downloads/zh-cn/latest/pdf/

1.4 视频教程

如果您是 Jam.py 的新手，我们强烈建议您观看这些视频教程。这些视频是针对 Jam.py V5 录制的，但与 V7 相比，用户界面的变化极小。

建议以 1080p 分辨率观看这些视频。

教程 1 - 处理文件和图像

教程 2 - 处理明细数据

教程 3 - 用户、角色、审计追踪/变更历史

教程 4 - 任务树

教程 5 - 表单

教程 6 - 表单事件

教程 7 - 数据感知控件

教程 8 - 数据集

教程 9 - 数据集第 2 部分

教程 10 - 字段和过滤器

教程 11 - 客户端-服务器交互

教程 12 - 在服务器上处理数据

CHAPTER 2

入门指南

在这里，您可以学习如何安装框架、创建新项目、开发 Web 应用程序并部署它。

2.1 安装

2.1.1 安装 Python

Jam.py 需要 Python。如果尚未安装，您可以在 https://www.python.org/download/ 获取最新版本的 Python。

您可以将以下 Python 版本与 Jam.py 一起使用：

Python 3：

- Python 3.4 及更高版本

您可以通过在 shell 中输入 `python` 来验证 Python 是否已安装；您应该会看到类似以下内容

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

如果安装了 Python 3，请尝试输入 `python3`

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

2.1.2 安装 Jam.py

使用 *pip* 安装官方发行版

这是安装 Jam.py 的推荐方式。

1. 安装 pip。最简单的方法是使用 独立的 pip 安装程序。如果您的系统已经安装了 pip，但版本过旧，则需要更新它。（如果版本过旧，你会发现无法安装。）
2. 如果您使用的是 Linux、Mac OS X 或其他类 Unix 系统，请在 shell 提示符下输入命令：

```
sudo pip install jam.py-v7
```

   如果您使用的是 Windows，请以管理员权限启动命令提示符，并运行命令：

   这会将 Jam.py 安装到 Python 安装目录的 site-packages 目录中。

手动安装官方发行版

1. 下载归档文件软件包。
2. 创建一个新目录，并将归档文件解压到该目录中。
3. 进入该目录，从命令行运行安装命令：

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

这会将 Jam.py 安装到 Python 安装目录的 site-packages 目录中。

> **备注**
>
> 在某些类 Unix 系统上，您可能需要切换到 root 用户或运行：sudo python setup.py install

> **Windows 上的 Python**
>
> 如果您是 Jam.py 的初学者并且使用 Windows，您可能会发现*如何在 Windows 上安装 Jam.py* 很有用。

2.2 创建项目

创建一个新目录。

进入该目录并从命令行运行：

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

对于 Windows 用户，jam-project.py 命令位于 Scripts 文件夹中，即该目录的上层文件夹是：

```
...\> ..\your_python_folder\Scripts\jam-project.py
```

在新目录中将创建以下文件和文件夹

```
/
    css/
    js/
    reports/
    static/
    locks/
    admin.sqlite
    langs.sqlite
    server.py
```

(续下页)

(接上页)

```
    index.html
    templates.html
    wsgi.py
```

要启动 Jam.py Web 服务器，请运行 `server.py` 脚本。

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

对于 Windows 用户：

```
...\>server.py
```

备注

你可以指定端口作为参数，例如：

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

默认端口是 8080。如果你指定了其他端口，则在后续步骤中需要在浏览器中指明该端口。

你将在命令行上看到类似输出

```
User Guide: https://jampy-docs-v7.readthedocs.io/
WARNING: This is a development server. Do not use it in a production deployment. Use a␣
↪production WSGI server instead.
 * Running on all addresses (0.0.0.0)
 * Running on http://127.0.0.1:8080
 * Running on http://127.0.0.1:8080
Press CTRL+C to quit
```

如果我们打开 Web 浏览器访问该应用 http://127.0.0.1:8080 ，将显示以下信息：

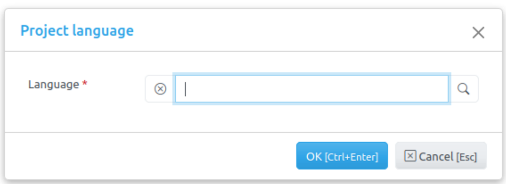
Information ×

The creation of the project has not been completed yet. Please run the Application Builder.

打开 Web 浏览器，访问本地域名的"/builder.html" ——例如：

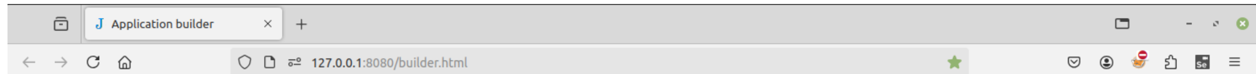
```
127.0.0.1:8080/builder.html
```

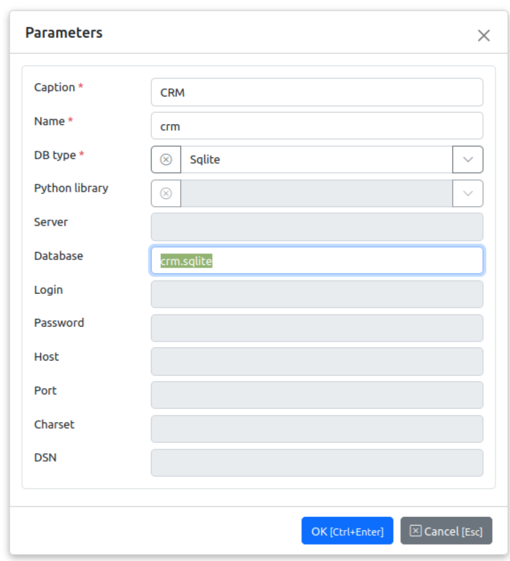
你应该会看到语言选择对话框。这定义了用户界面可以使用的语言。你可以点击输入框右侧的“查找”图标打开默认的语言列表，你可以从中选择已有的语言，或者使用右侧的“导入 (Import)”按钮导入自己的语言文件。更多信息请参阅语言支持 页面。选择适合你的语言，然后点击确定按钮。



接下来是新项目对话框。请填写:

- **标题 (Caption)** - 将向用户显示的项目名称。
- **名称 (Name)** - 项目（任务）的名称，将在代码（Python 或 JS）中用于访问任务对象。这应是一个简短且 **有效的 Python 标识符**。在项目数据库中创建表时，此名称也用作前缀。
- **数据库类型 (DB type)** - 选择数据库类型。如果数据库不是 Sqlite，必须提前创建，并在相应的属性输入框中输入值。查看数据库设置示例，请访问此链接 。





点击确定后，将检查数据库连接，如果失败则会显示错误消息。

2.2.1 数据库设置示例

改编自 Jam.py 设计技巧

Jam.py 支持多种不同的数据库服务器。例如 PostgreSQL、MariaDB、MySQL、MSSQL、Oracle、Firebird、IBM、SQLite、Databricks 以及使用 SQLCipher 的 SQLite。

如果你正在开发一个小型项目或不打算在生产环境中部署，SQLite 通常是最佳选择，因为它不需要运行单独的服务器。然而，SQLite 与其他数据库有许多不同之处，因此如果你正在开发一个重要的项目，建议使用与生产环境相同的数据库进行开发。

除了配置数据库后端，我们还需要确保安装了对应的 Python 数据库绑定（操作数据库的驱动包）。

- 如果使用 PostgreSQL，需要 `psycopg2` 或 `psycopg2-binary` 包。
- 如果使用 MySQL 或 MariaDB，Python 2.x 需要 `MySQLdb`。对于 Python 3.x，需要 `mysql-connector-python` 和 `mysqlclient` 包，以及数据库客户端开发文件。
- 如果使用 MSSQL，需要 `pymssql`。
- 如果使用 Oracle，需要 cx_Oracle 以及 Python 头文件（开发文件）。
- 如果使用 SQLCipher_（待补充），Linux 需要 `sqlcipher3-binary` 包。Windows 有独立的 DLL 可用。
- 如果使用 IBM_（待补充），需要 `ibm_db` 和 `ibm_db_dbi` 包。

- 如果使用 Firebird，需要 fdb 包。
- 如果使用 Databricks，需要 databricks-sql-connector。
- 要生成报告，必须安装 **LibreOffice**。

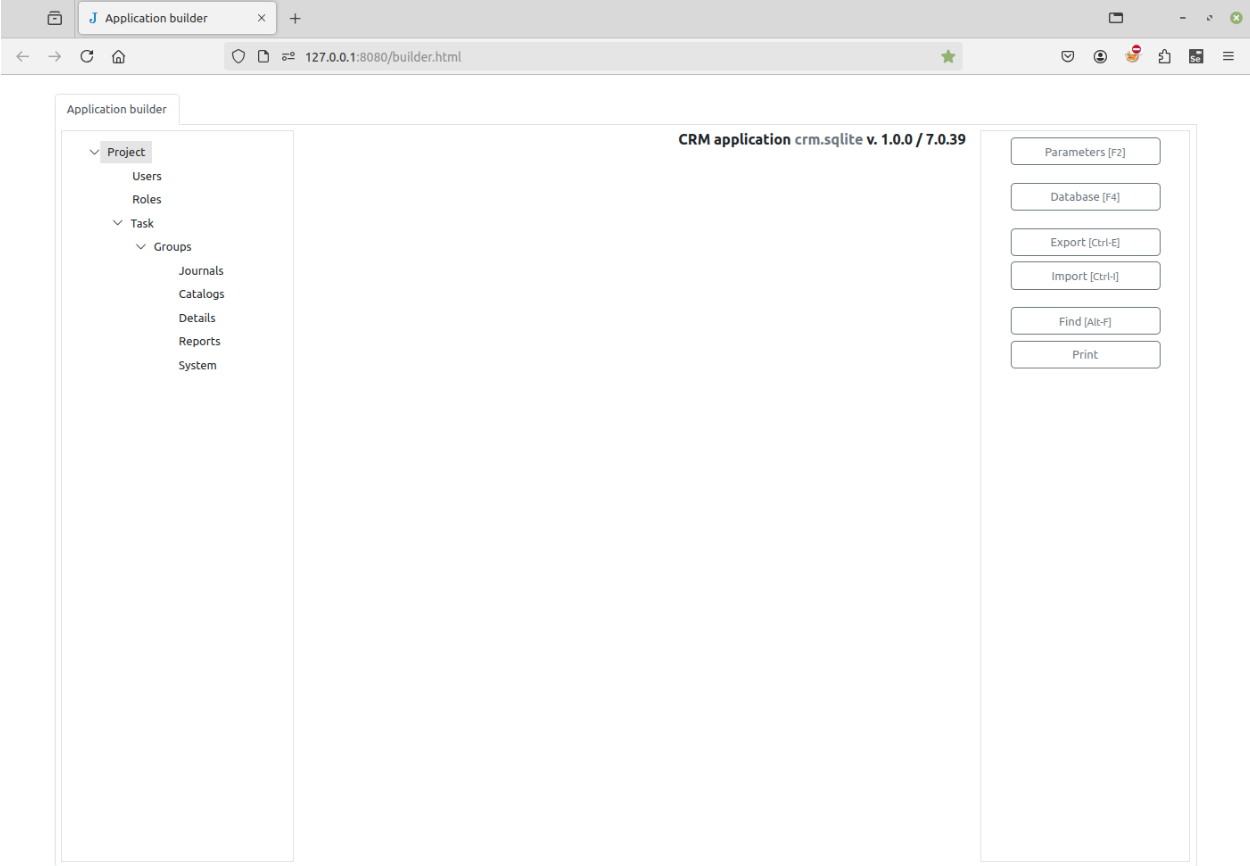
备注

对于 **SQLite** 数据库，当删除或重命名字段，或创建外键时，应用程序构建器 会创建一个新表并将记录从旧表复制到其中。

对于 **SQLite** 数据库，Jam.py 不支持将元数据导入到现有项目（数据库中已有表的项目）中。你只能将元数据导入到新项目中。

关于数据库设置的更多信息，请参阅项目管理 。

如果一切顺利，将创建一个新项目，项目树将显示在应用程序构建器中。

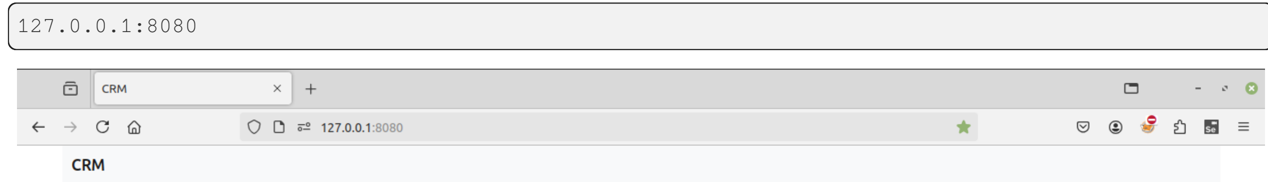


重要

如图中右上角所示，显示应用程序的名称、使用的数据库名称、应用程序版本号以及 Jam.py 框架的 版本号。

现在，要查看项目本身，请在浏览器中新建一个页面，并在地址栏中输入：

```
127.0.0.1:8080
```



2.3 演示项目

该框架包含一个功能完整的演示应用程序，展示了框架中使用的编程技术。

演示程序位于从 Github 下载的 Jam.py v7 软件包的 demo 文件夹中。此外，还提供了一个适用于 Jam.py v5 Windows x64 的独立版或便携式应用程序，可从 此处 获取。

该便携式应用程序仅依赖 LibreOffice 生成报告，无需其他配置。只需运行它，并按照以下步骤在浏览器中访问即可。

要启动演示应用程序，请进入 demo 文件夹并运行 *server.py* 脚本。

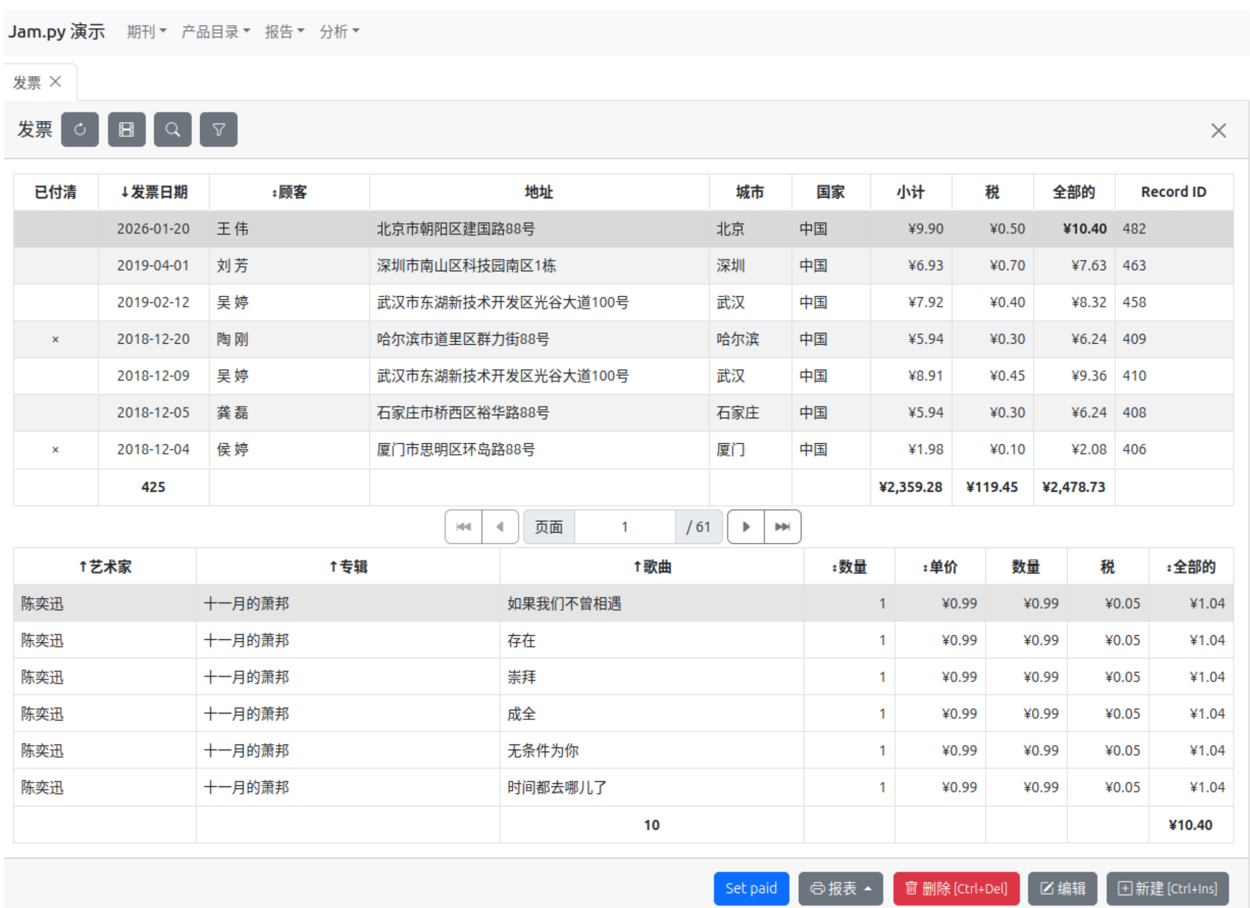
```
$ git clone https://github.com/jam-py-v7/jampy-docs-v7/
$ cd demo_zh_CN
$ ./server.py
```

打开 Web 浏览器，在地址栏中输入：

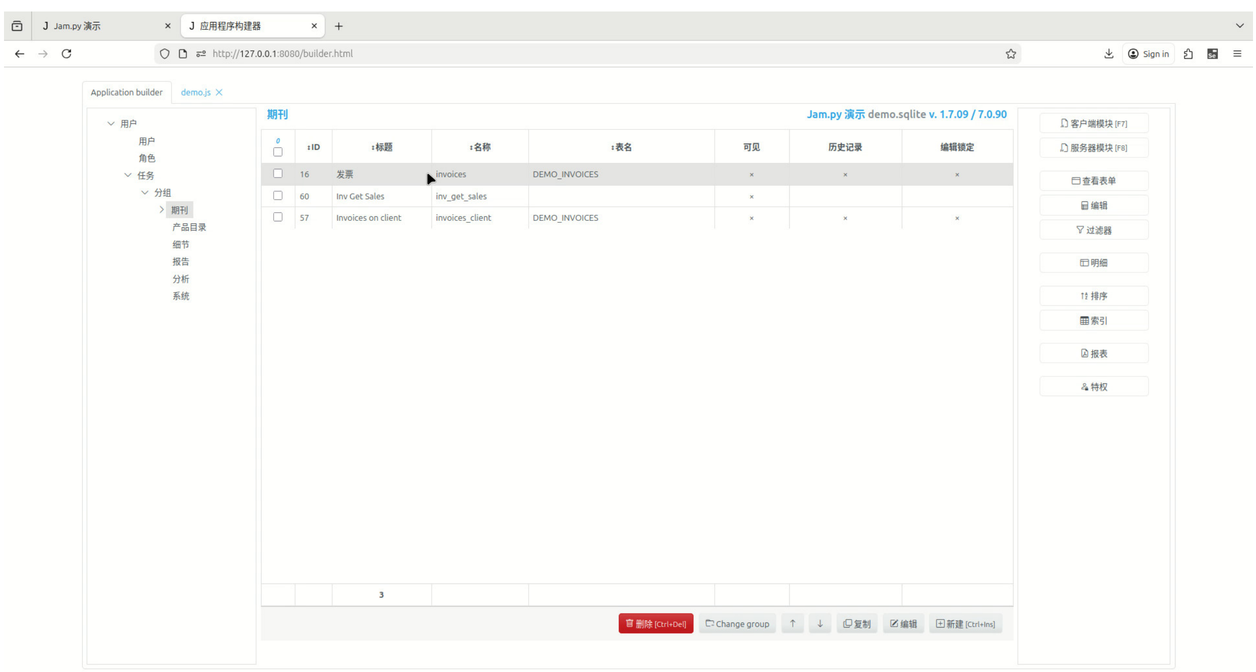
```
127.0.0.1:8080
```

要查看应用程序构建器，请在浏览器中打开一个新页面并输入：

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



业务应用程序构建器：



2.4 教程第一部分：第一个项目

现在，我们将引导您创建一个基本的客户关系管理（CRM）应用程序。请按照以下步骤操作：

2.4.1 新建项目 (project)

我们假设已经安装好 jam.py 。如果尚未安装，请参阅安装指南 了解如何安装。

首先为新项目创建一个文件夹。在此文件夹中，执行 `jam-project.py` 脚本来创建项目结构。

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

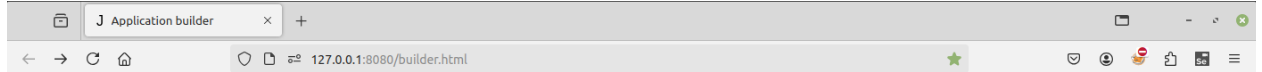
之后，运行 `jam-project.py` 创建的 `server.py` 脚本：

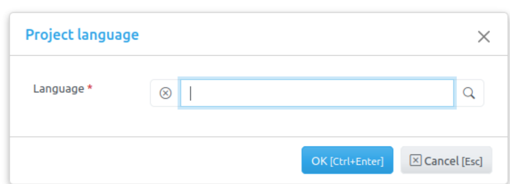
```
$ ./server.py
```

现在，要完成项目的创建，请打开 Web 浏览器并访问以下地址以打开应用程序构建器。

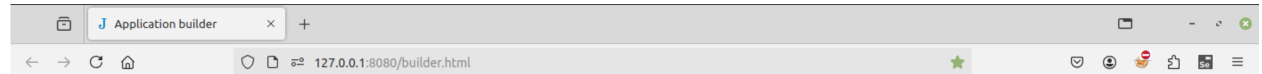
```
127.0.0.1:8080/builder.html
```

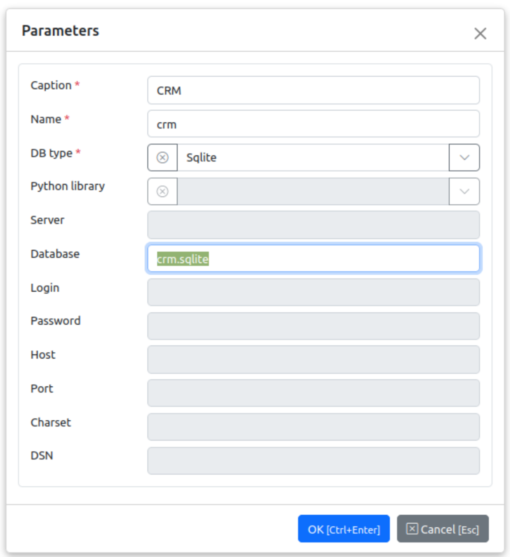
您应该会看到语言选择对话框。



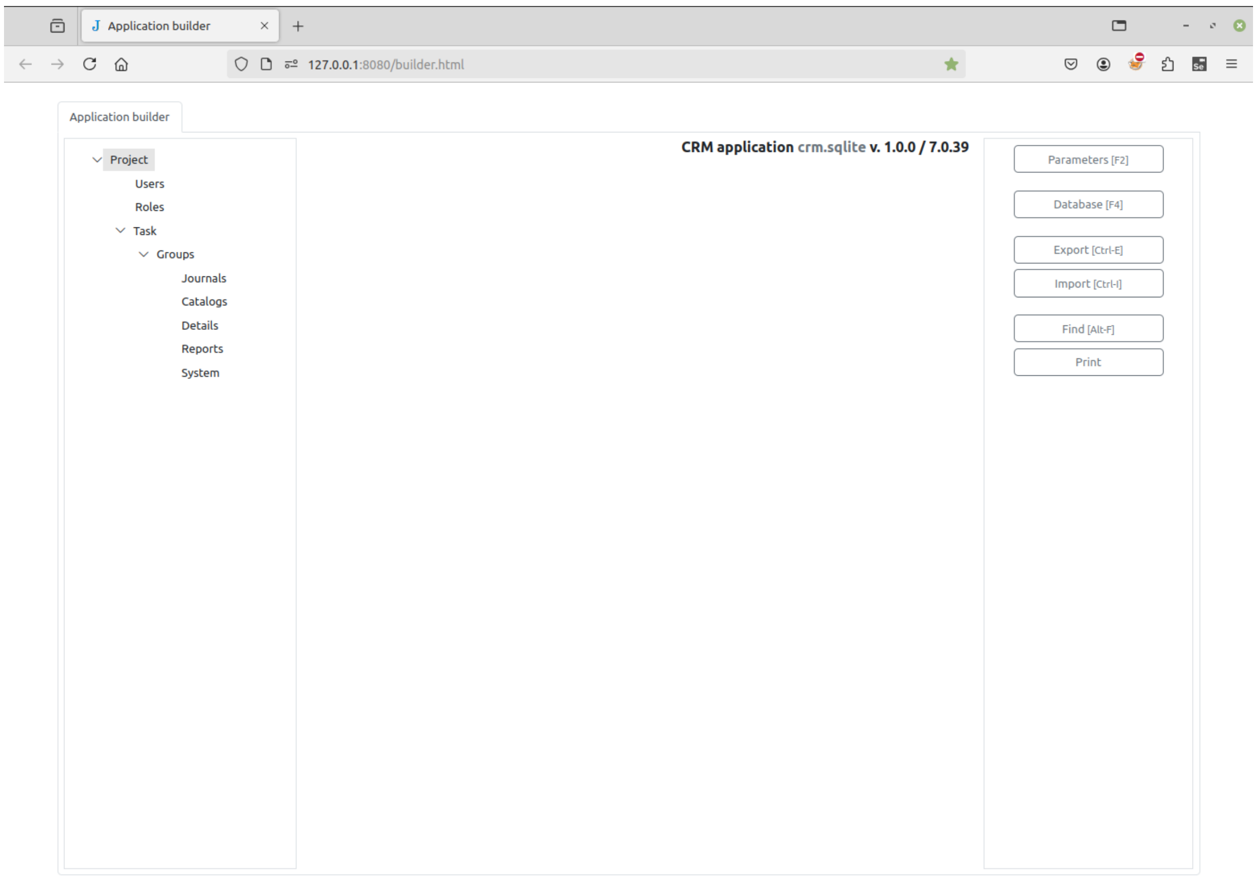


使用浏览按钮选择 (或查找) **Chinese zh-CN** ， 然后点击 **确定 (OK)** 按钮。将出现项目参数对话框。





如上图所示填写表单，然后点击 **确定 (OK)**。现在您应该在左侧面板中看到项目树。

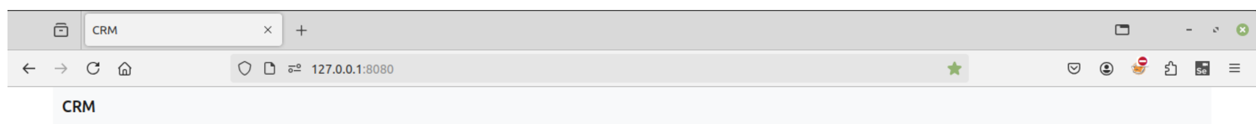


> **重要**
>
> 如图片右上角所示，显示应用程序的名称、使用的数据库名称、应用程序版本号以及 `Jam.py` 框架的 版本号。

打开一个新页面，在地址栏中输入

```
127.0.0.1:8080
```

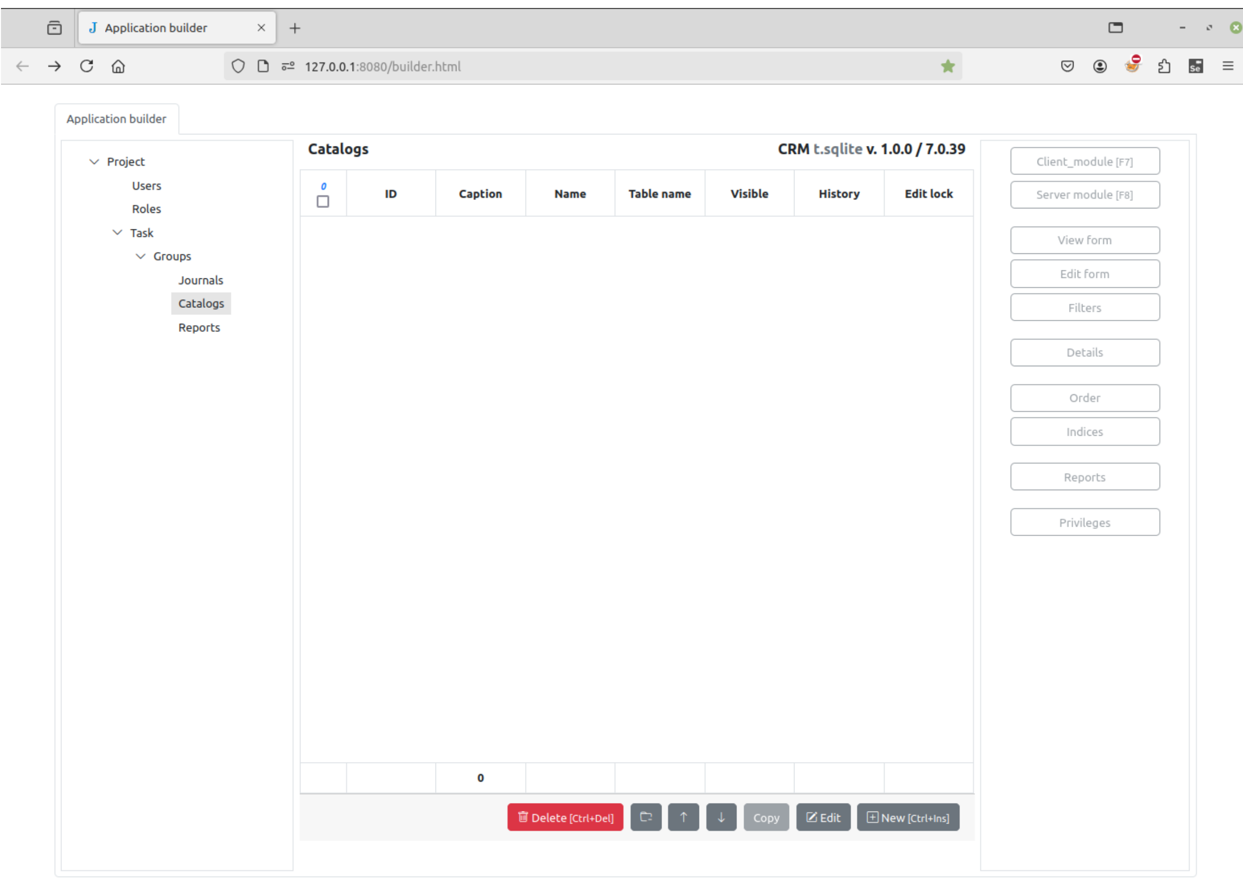
并按 Enter 键。将出现一个具有空菜单的新项目。



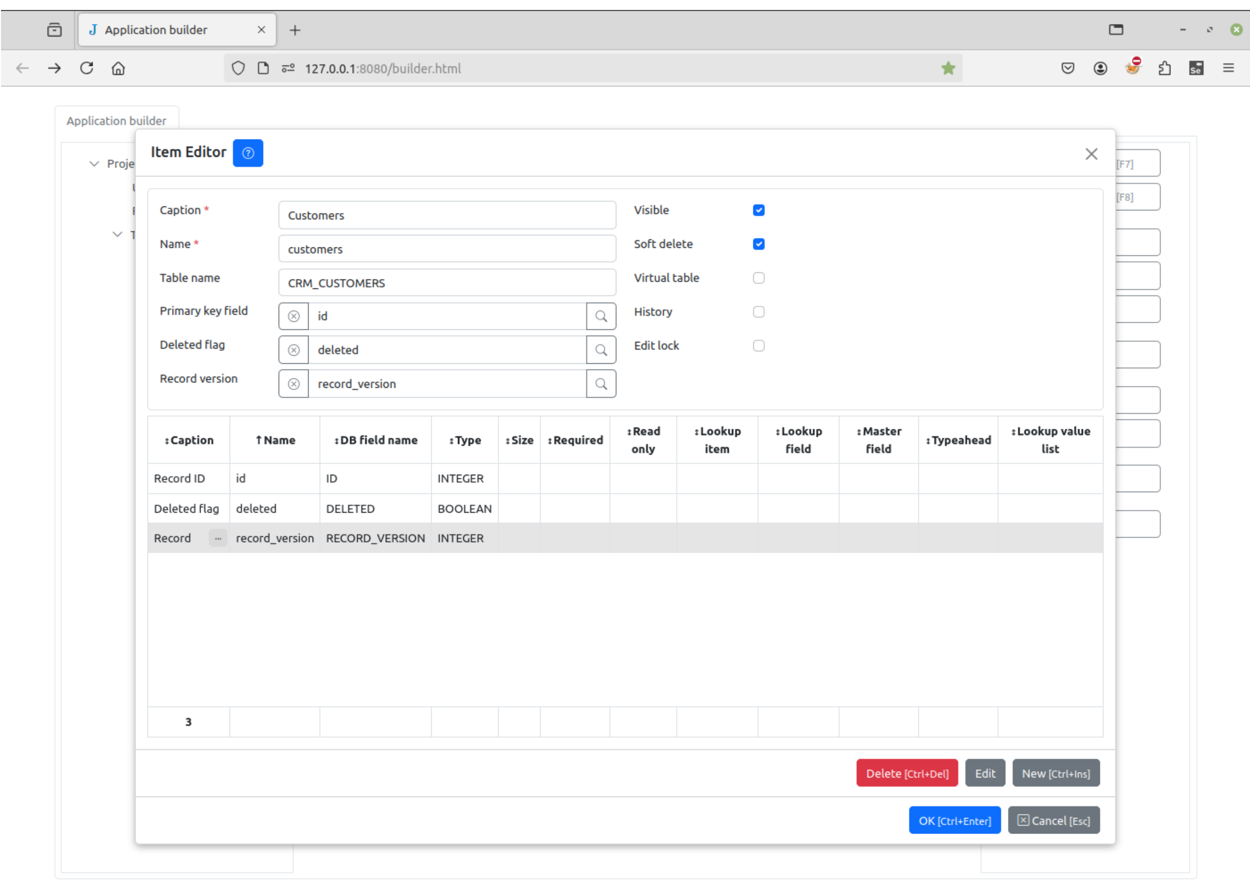
2.4.2 新建主表目录 (catalog)

让我们回到应用程序构建器页面，创建一项名为" 客户 (Customers)" 的主表数据项。每项新的主表对应数据库中的一个新数据表。

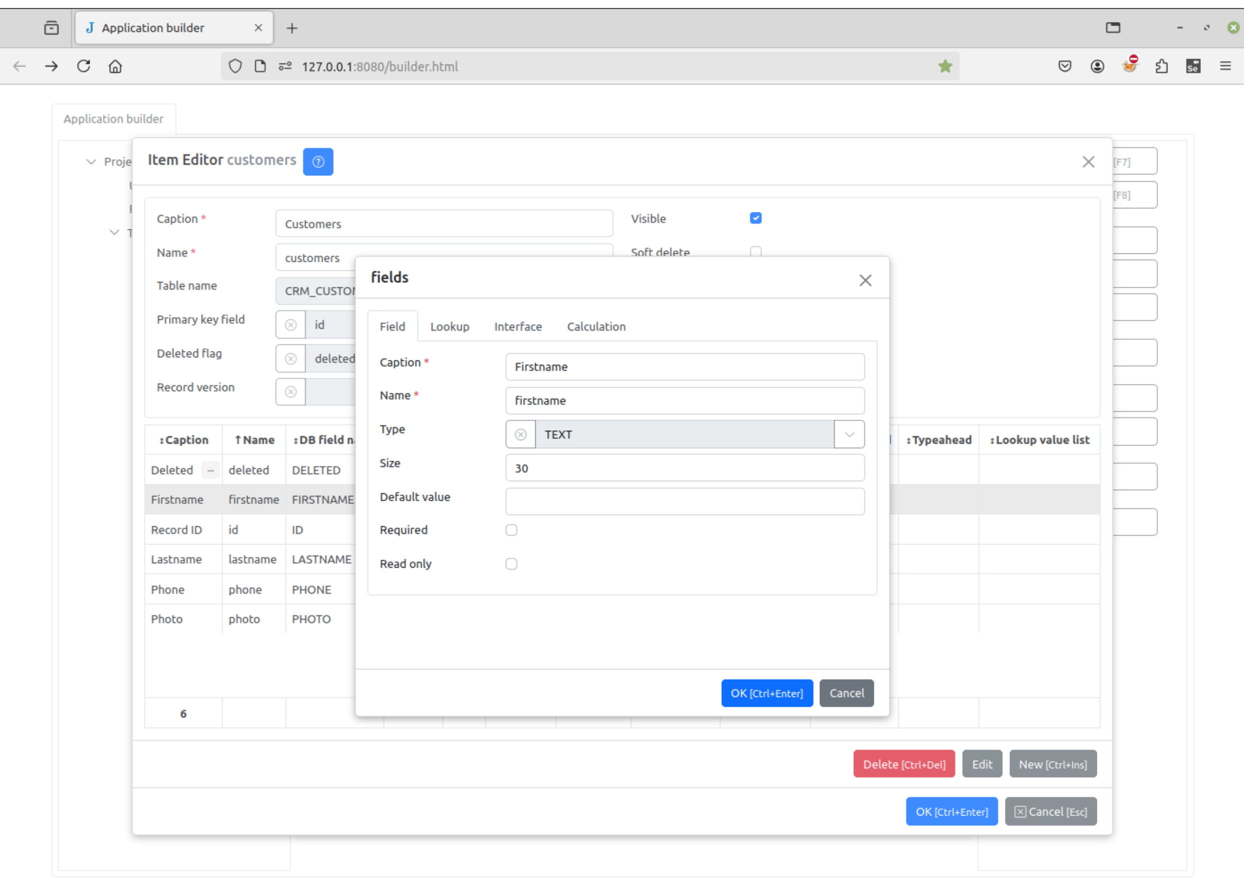
为此，在项目树中选择 "主表目录 (Catalogs)" 组，然后点击页面右下角的 **新建 (New)** 按钮



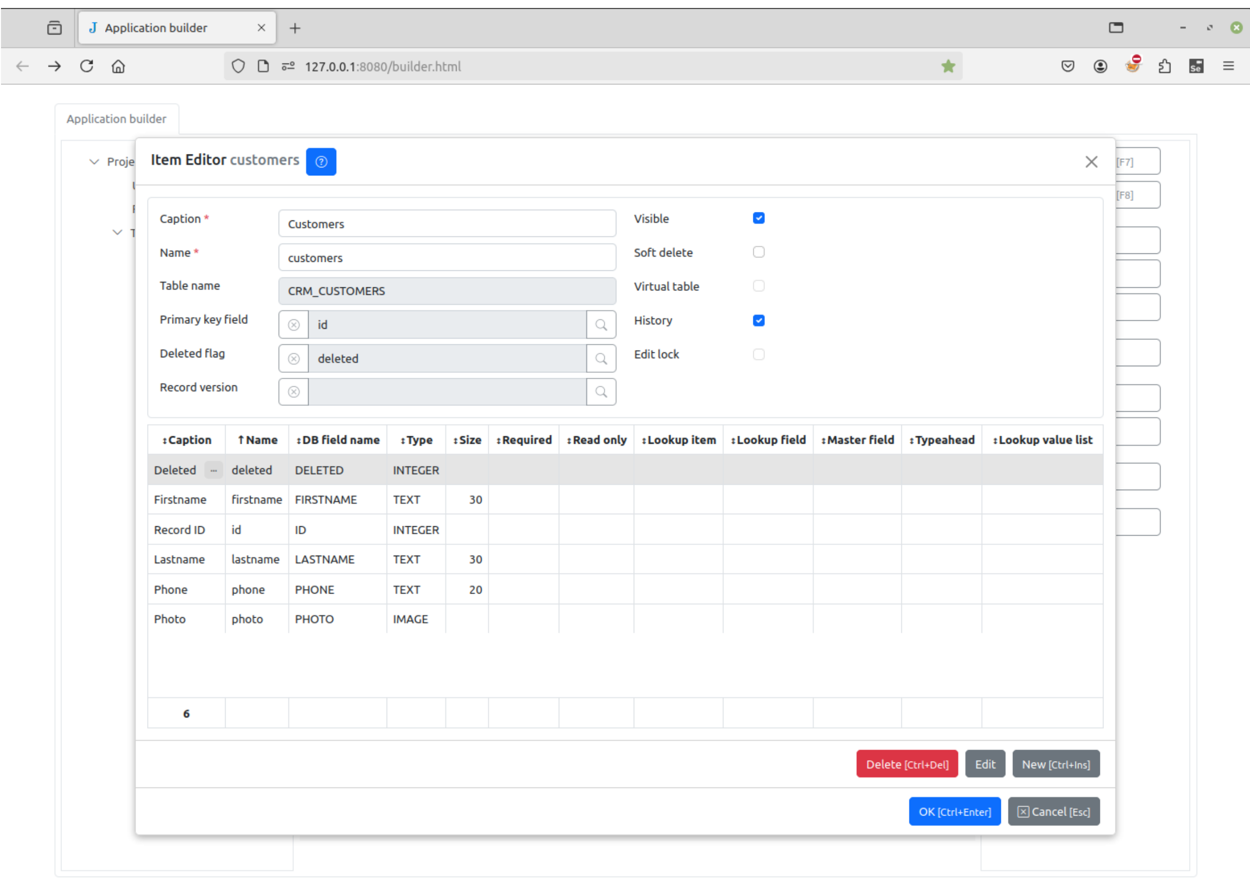
在出现的实体项编辑器对话框中，填写新数据表对应的目录的标题和名称。标题是向用户显示的名称，而名称是在代码（Python 或 JS）中引用此主表项对应的数据表时使用的变量名。名称必须是有效的 Python 标识符。



然后，点击对话框右下角的 **新建 (New)** 按钮添加新字段。将出现字段编辑器 对话框。输入 "名字 (Firstname)" 字段的标题和名称，并选择其类型（此处为 30 个字符的 `TEXT` 类型），然后点击 **确定 (OK)** 按钮。



类似地，添加 “姓氏 (Lastname)”（30 个字符的 `TEXT` 类型）和 “电话 (Phone)”（20 个字符的 `TEXT` 类型）字段。添加 “姓氏 (Lastname)” 字段时，请勾选 **必填 (Required)** 属性。这要求在添加新数据项时必须填写该字段。



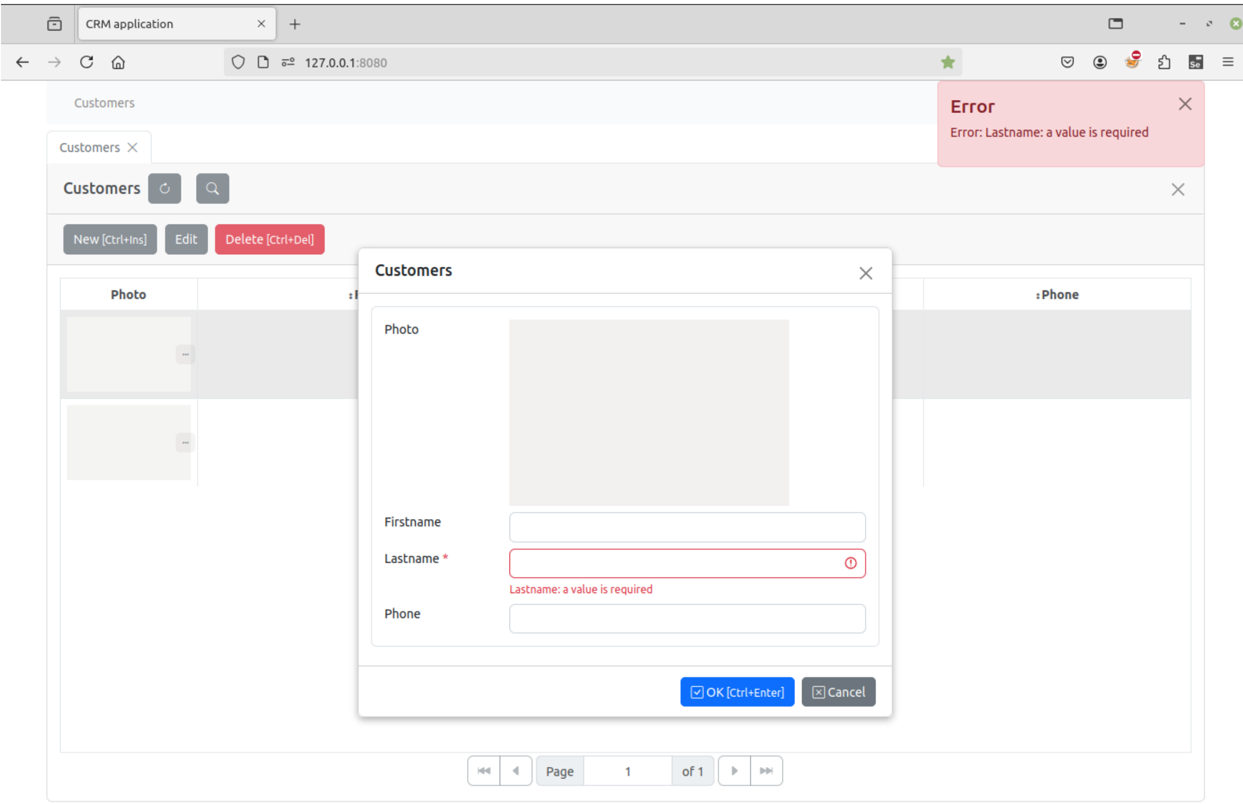
现在，请点击 **确定 (OK)** 按钮保存更改。保存时，应用程序构建器将在 `crm.sqlite` 数据库中创建 `CRM_CUSTOMERS` 表：

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

127.0.0.1 - - [07/Aug/2018 11:32:30] "POST /api HTTP/1.1" 200 -
```

转到项目的客户端页面并确保刷新了页面

```
127.0.0.1:8080
```

然后，点击页面右下角的 **新建 (New)** 按钮创建一个新客户。填写对话框，然后点击 **确定 (OK)** 按钮：



另请参阅

数据的转义清理

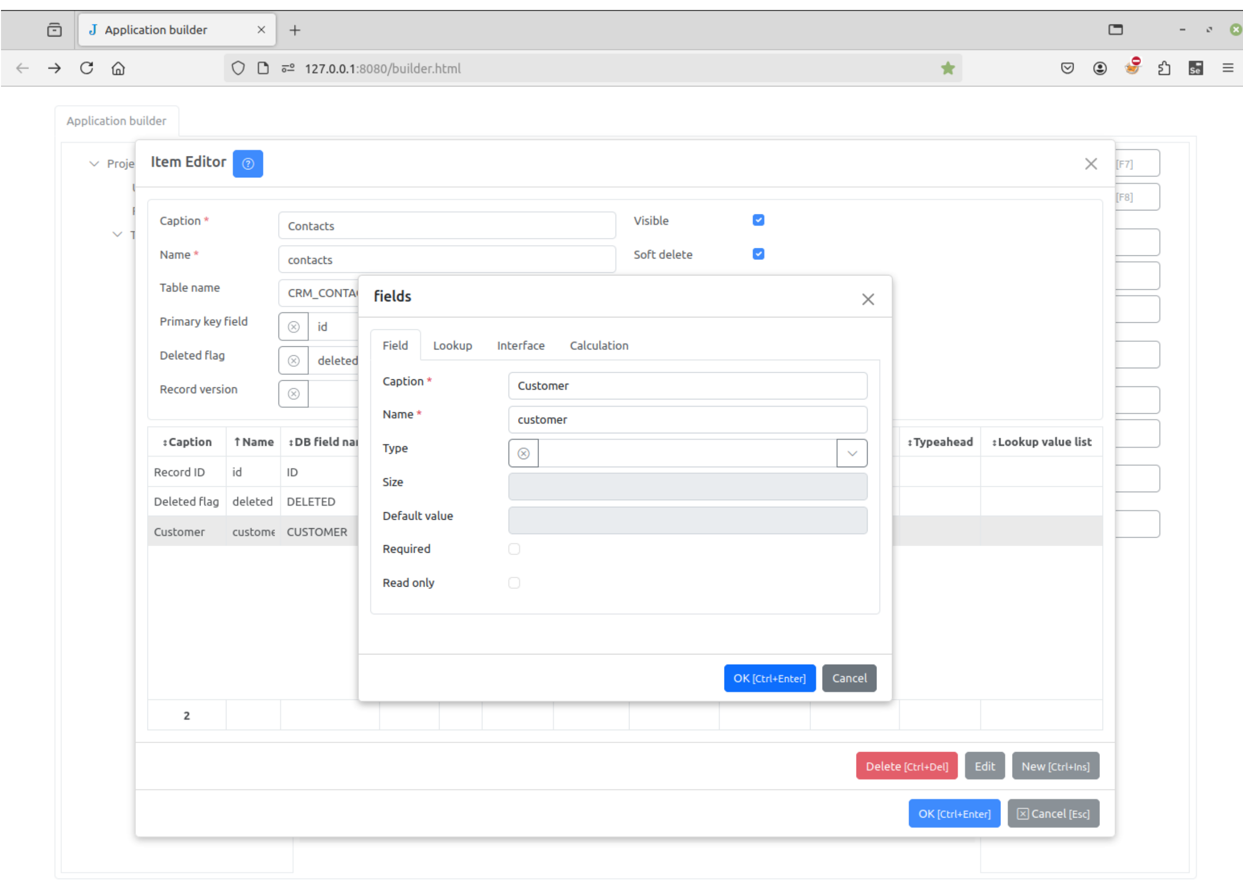
2.4.3 查找字段

现在，我们将创建名为"联系人 (Contacts)"的业务台账，并使用查找字段将联系人与客户关联起来。查找字段允许一个数据实体（即主表或业务台账）的元素引用另一个实体中的元素。

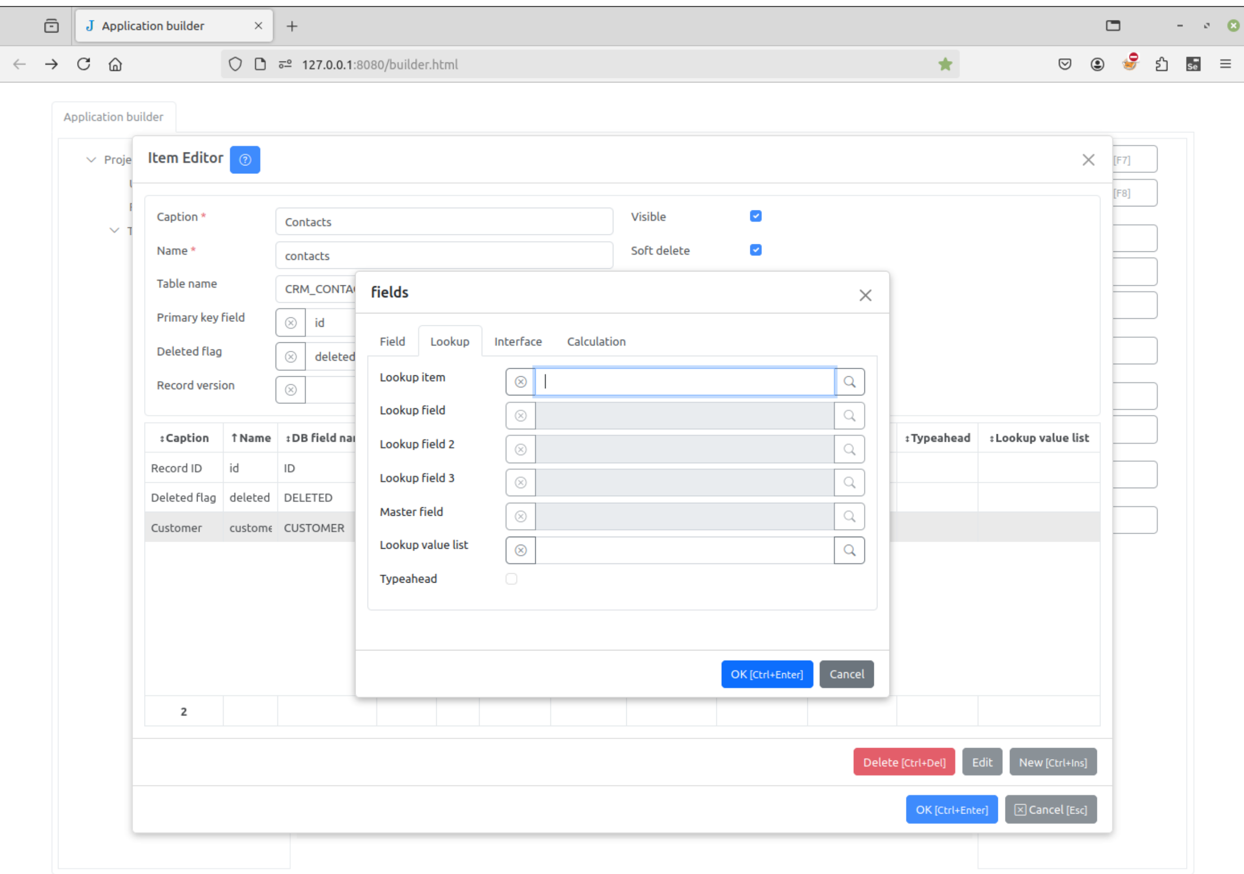
在项目任务树中选择"业务台账 (Journals)"，并以与创建"客户 (Customers)"主表相同的方式添加新的业务台账。业务台账和主表一样，对应数据库中的不同表。有关更多信息，请参阅此链接。

首先，添加一个 `DATETIME` 类型的"联系日期 (Contact date)"字段，以及一个 `TEXT` 类型的"备注 (Notes)"字段。

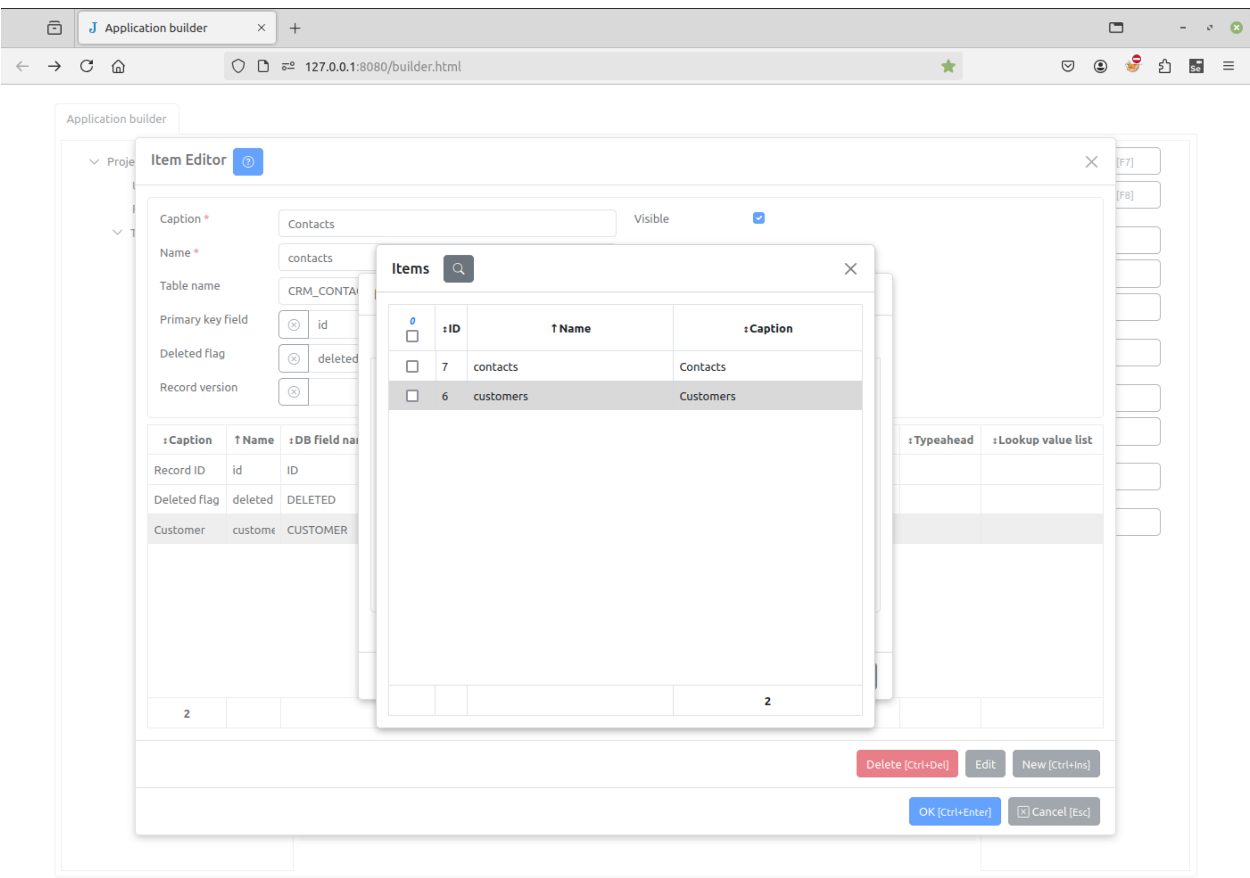
然后，添加查找字段 "业务客户 (Customer)"，该字段将存储对"客户 (Customers)"表中记录的引用。



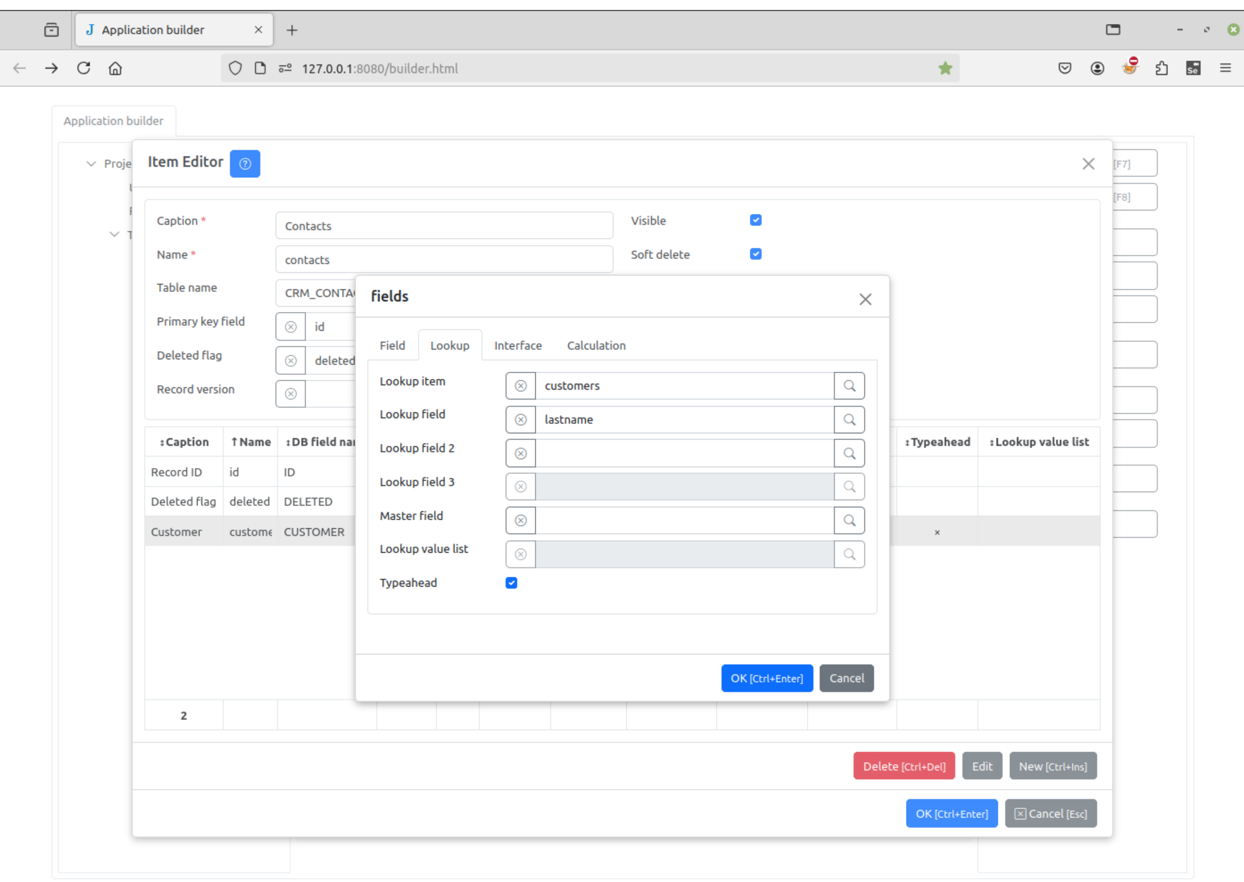
要创建查找字段，首先指定其标题和名称，并将类型留空。然后转到 **查找 (Lookup)** 选项卡，点击 **查找项 (Lookup item)** 输入框右侧的按钮。



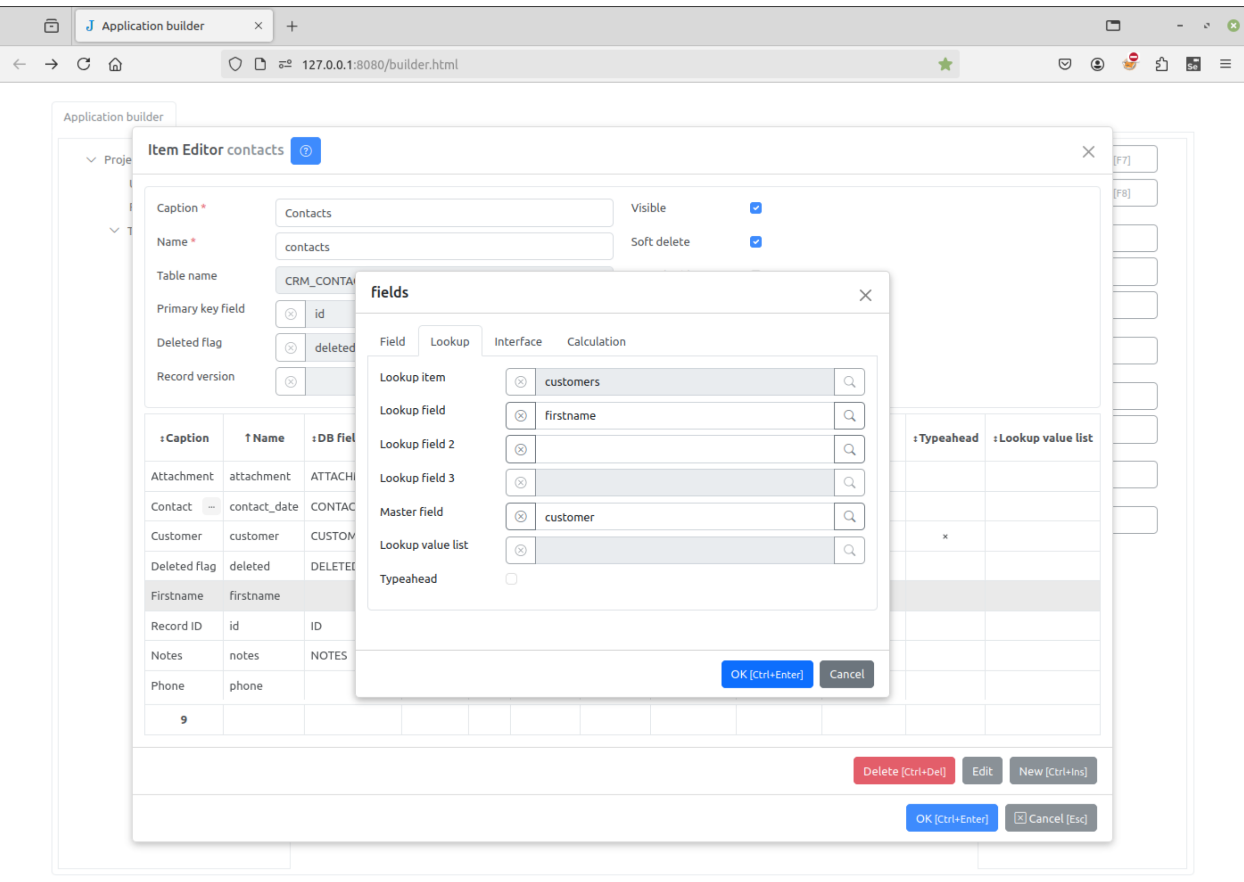
这将弹出一个实体项列表。双击“客户 (Customers)”实体项以选择它。

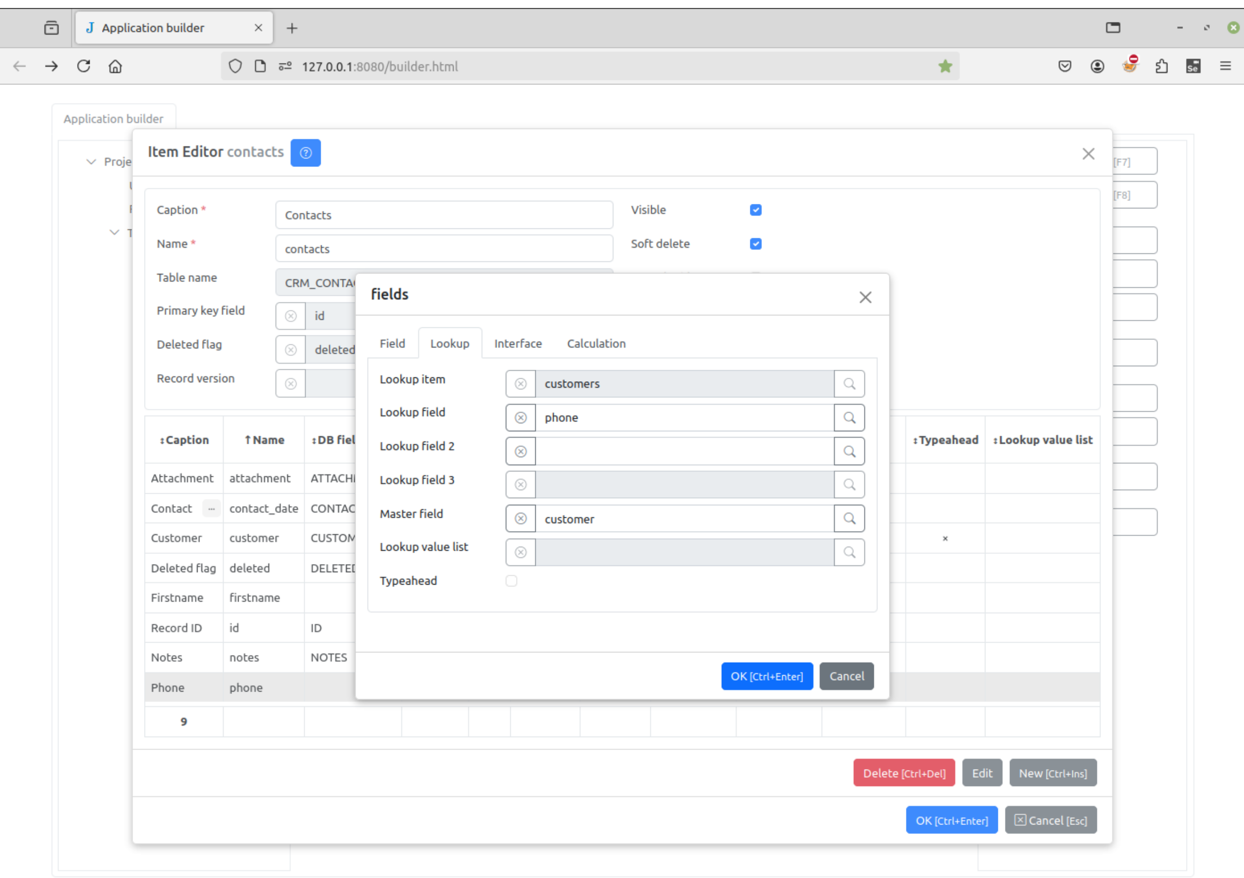


接下来，我们需要指定一个查找字段。这是用于定位客户的方式。这里我们选择“姓氏 (Lastname)”。将其他留空，然后点击 **确定 (OK)** 。



重复此过程，添加“名字 (Firstname)”和“电话 (Phone)”查找字段。对于这些字段，我们将“业务客户 (Customer)”字段指定为它们的 **主字段 (Master field)** 属性。这使它们与我们创建的第一个“姓氏 (Lastname)”查找字段关联起来，从而使所有三个查找字段都引用同一个客户。


Application builder
127.0.0.1:8080/builder.html
Item Editor contacts
Caption *
Contacts
Name *
contacts
Table name
Primary key field
Deleted flag
Record version
Visible
Soft delete
fields
Field
Lookup
Interface
Calculation
Lookup item
customers
Lookup field
firstname
Lookup field 2
Lookup field 3
Master field
customer
Lookup value list
Typeahead
OK [Ctrl+Enter]
Cancel
Attachment
Contact
Customer
Deleted flag
Firstname
Record ID
Notes
Phone
Delete [Ctrl+Del]
Edit
New [Ctrl+Ins]
Cancel [Esc]

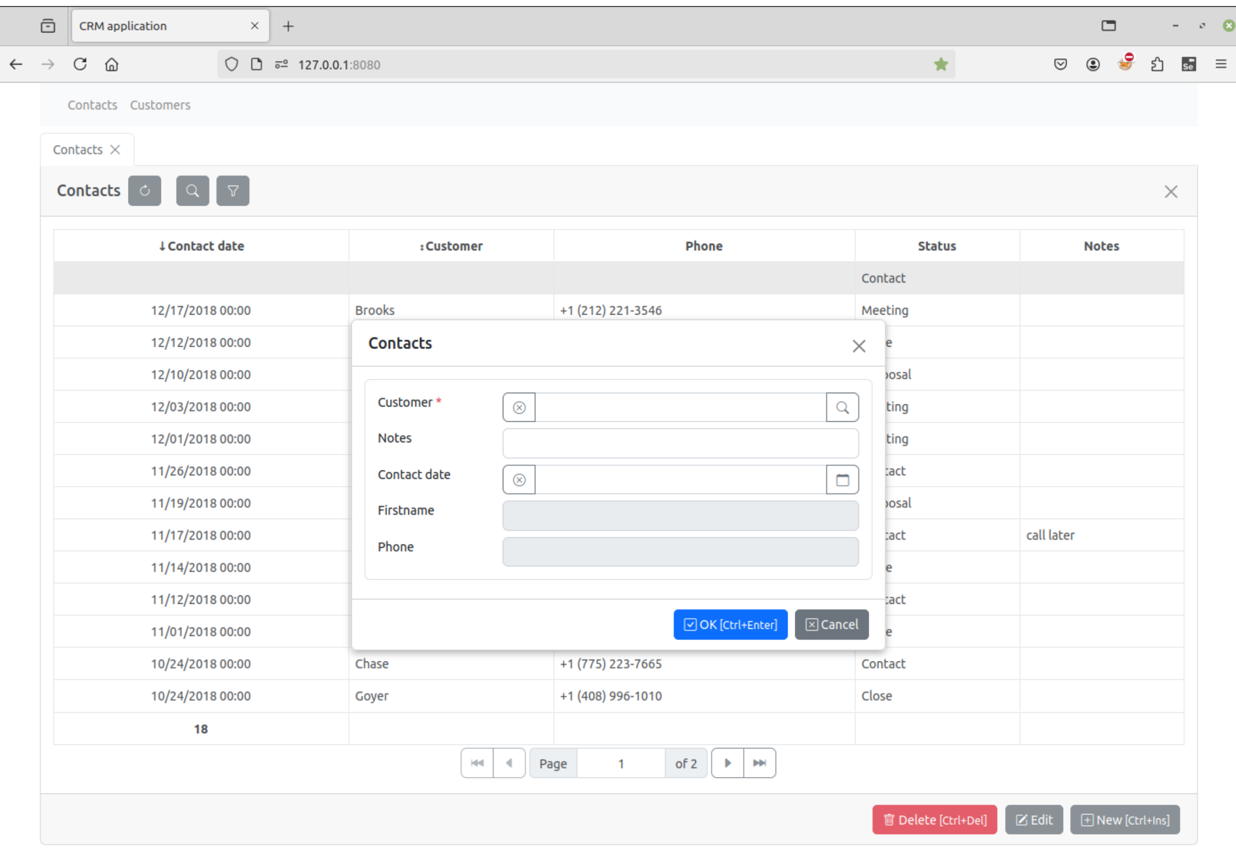


点击 **确定 (OK)** 按钮保存“联系人 (Contacts)”实体。

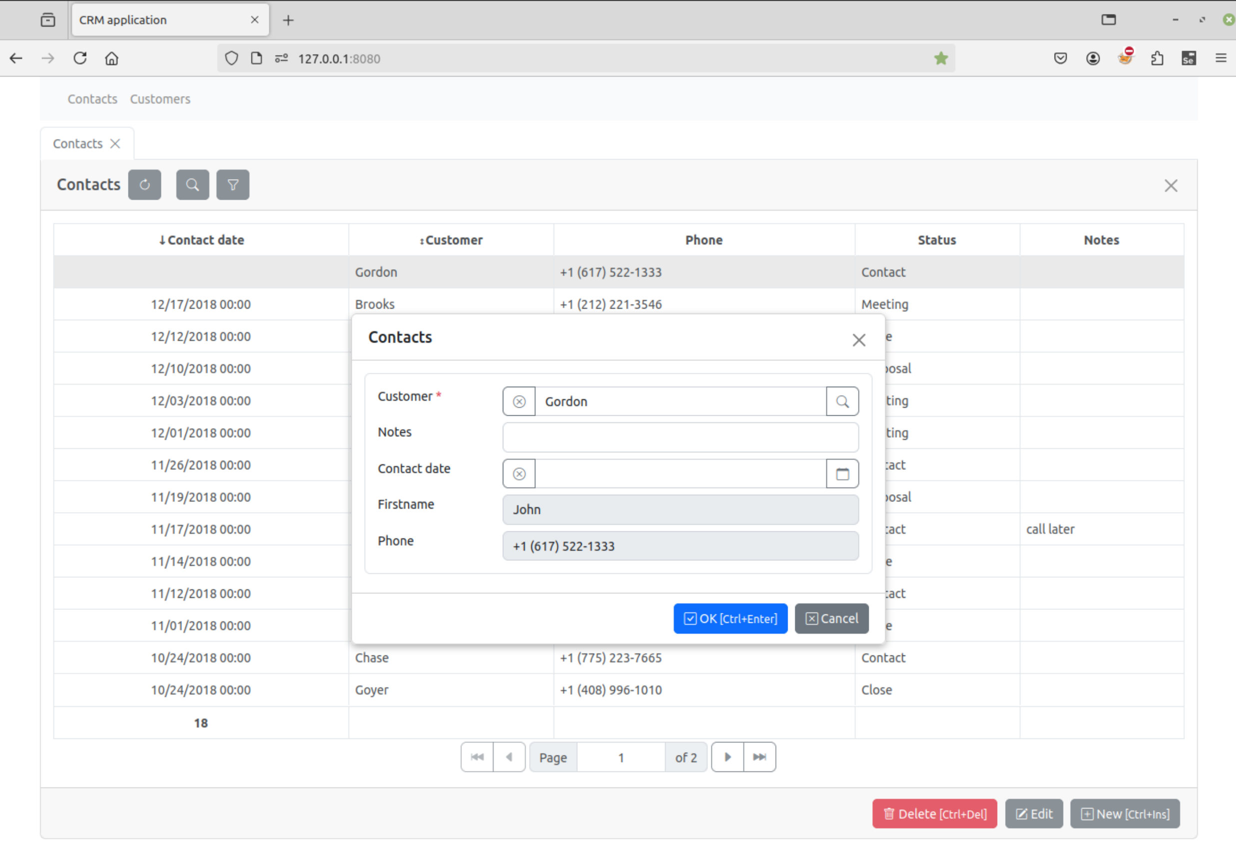
```
127.0.0.1 - - [07/Aug/2018 14:24:11] "POST /api HTTP/1.1" 200 -
CREATE TABLE "CRM_CONTACTS"
(
"ID" INTEGER PRIMARY KEY,
"DELETED" INTEGER,
"CONTACT_DATE" TEXT,
"NOTES" TEXT,
"CUSTOMER" INTEGER)

127.0.0.1 - - [07/Aug/2018 14:24:11] "POST /api HTTP/1.1" 200 -
```

如您所见，`CRM_CONTACTS` 表中没有“名字 (Firstname)”和“电话 (Phone)”字段。这是因为我们已将这些字段的 **主字段 (Master field)** 属性设置为“业务客户 (Customer)”。“业务客户 (Customer)”字段将存储对“客户 (Customers)”表中记录的引用，而该记录将包含“名字 (Firstname)”和“电话 (Phone)”字段。

刷新项目页面，在“联系人 (Contacts)”页面中，点击 **新建 (NEW)** 按钮。您会看到“业务客户 (Customer)”输入框右侧有一个小按钮。



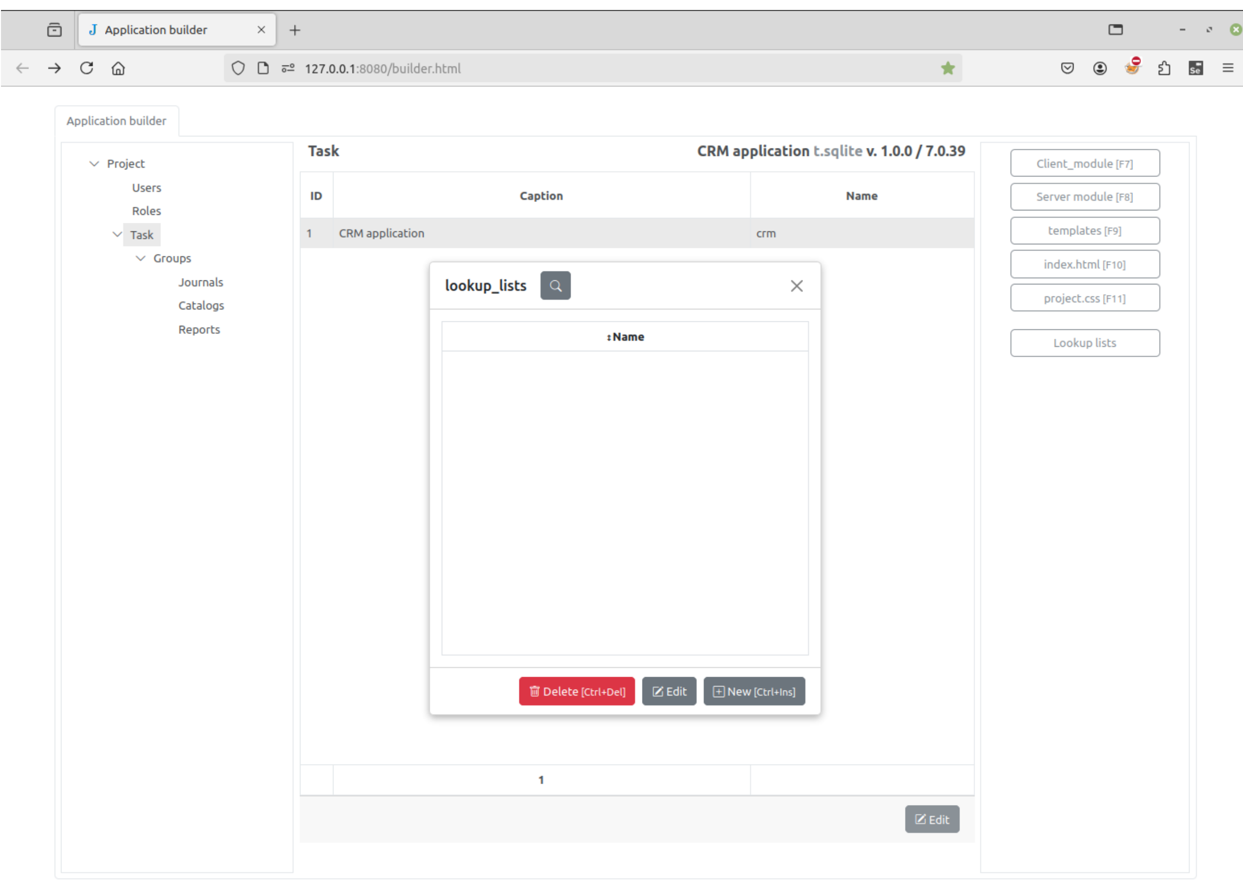
点击它，然后在“客户 (Customers)”表中选择一条记录，“业务客户 (Customer)”、“名字 (Firstname)”和“电话 (Phone)”字段将自动填充。



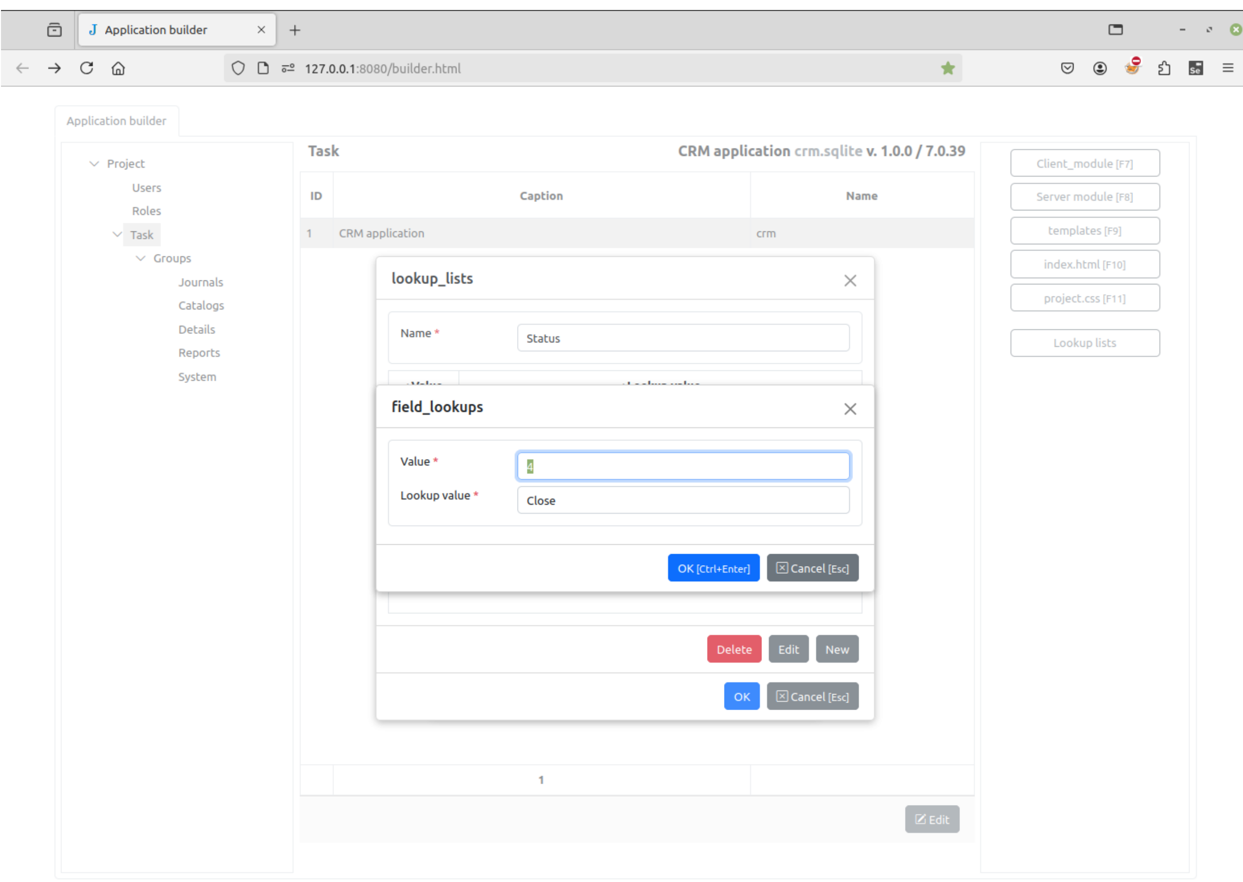
2.4.4 查找列表 (Lookup lists)

现在我们来创建一个查找列表 “状态 (Status)”。查找列表用于创建” 下拉框” 中的字段，其值域为有限的可选值集合。

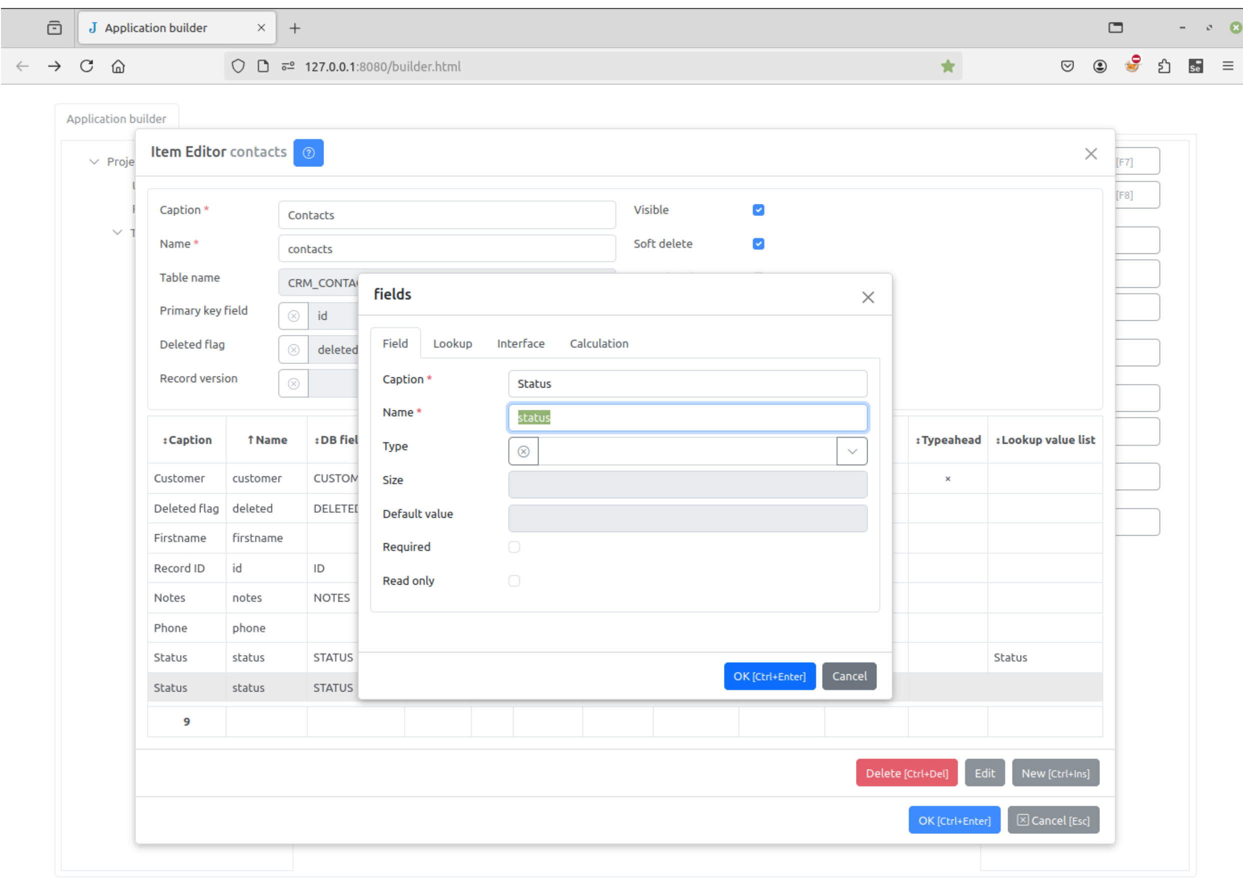
在项目树中选择 “任务 (Task)” 节点，然后在右侧点击 **查找列表 (Lookup lists)** 按钮。



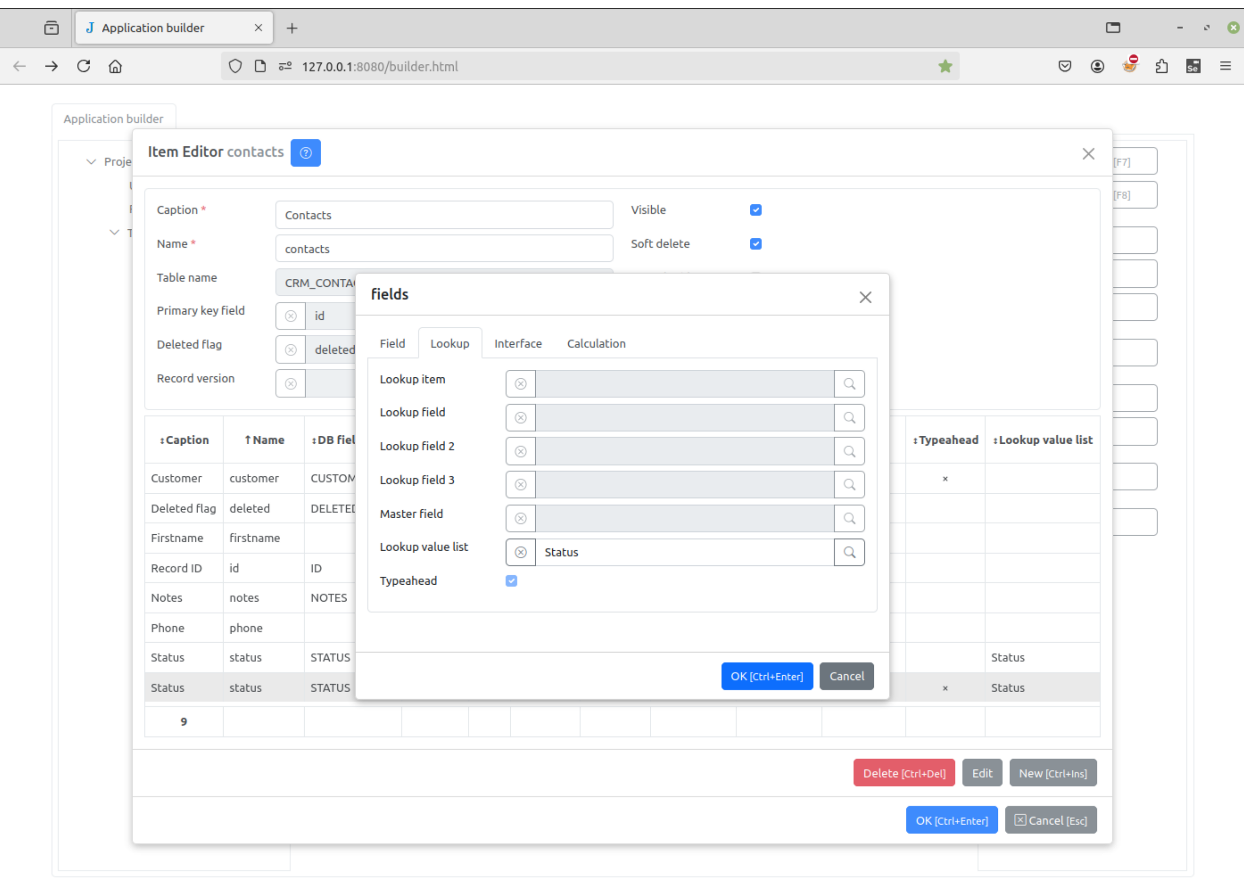
初始状态下它是空的。点击 **新建 (New)** 按钮创建一个新列表。指定新“查找列表”的名称为“状态 (Status)”，并点击 **新建 (New)** 添加几个“整数-文本”对（例如 1-已联系，2-未联系）：



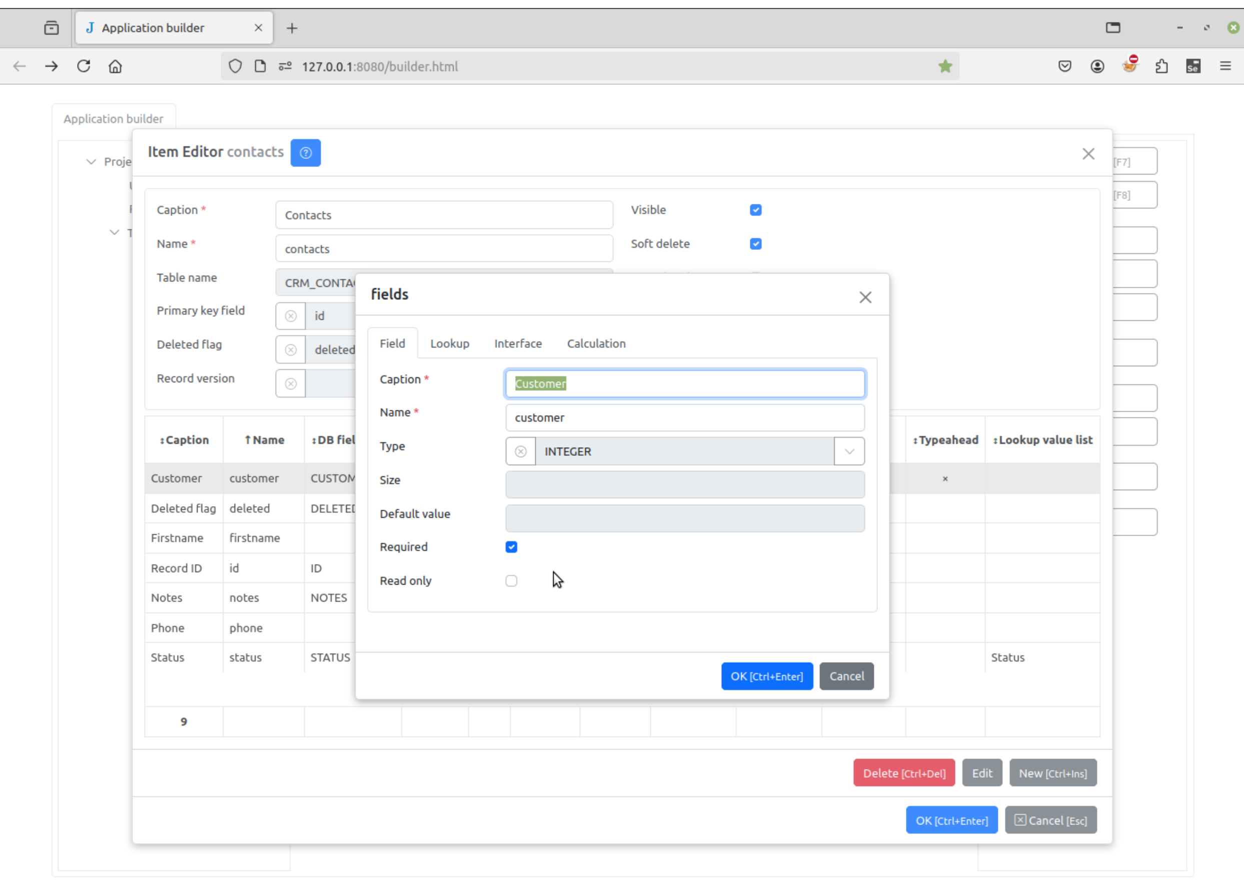
使用 **确定 (OK)** 按钮保存查找列表，然后编辑“联系人 (Contacts)”台账，添加新的“状态 (Status)”字段。

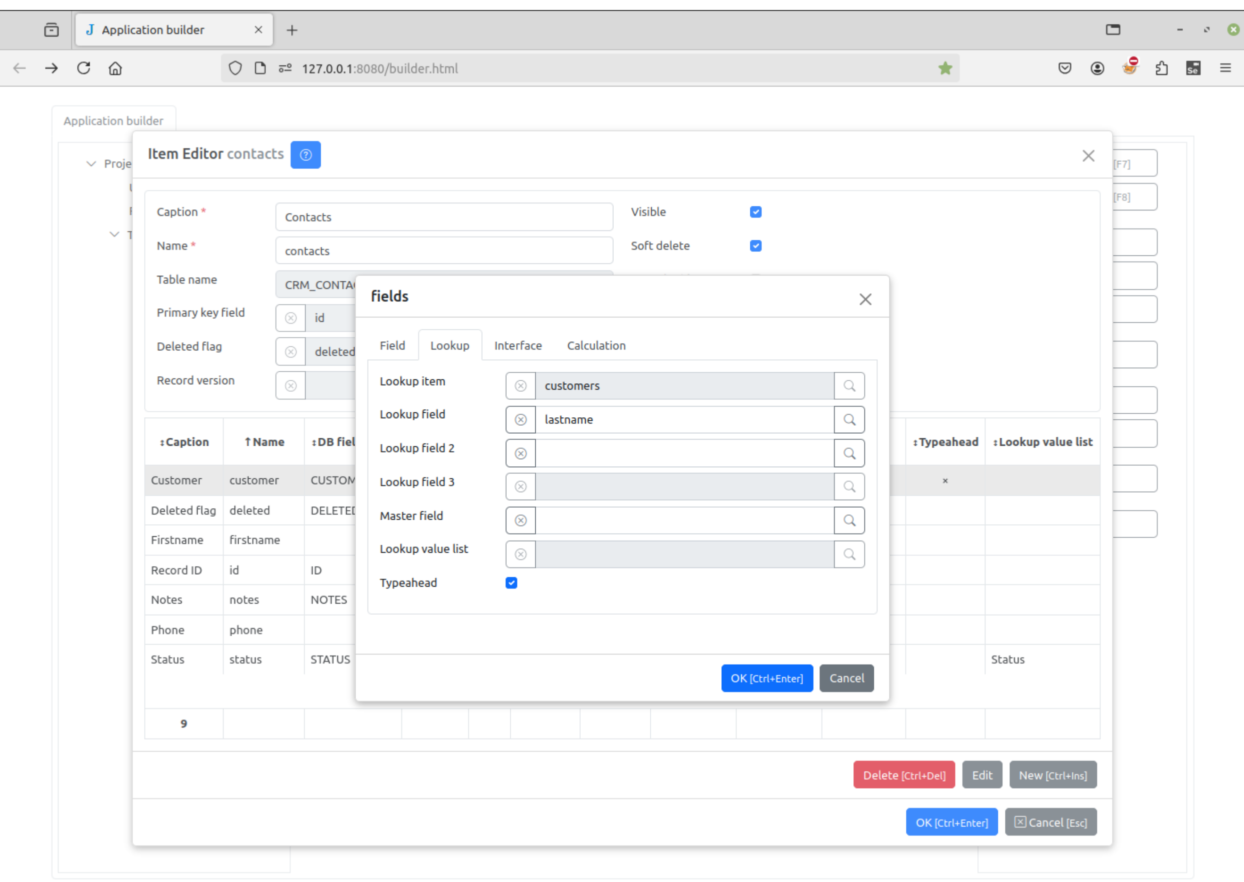


对于查找字段，设置标题和名称，并将类型留空。然后转到 **查找 (Lookup)** 选项卡，将 **查找值的列表 (Lookup value list)** 属性设置为"状态 (Status)"查找列表：

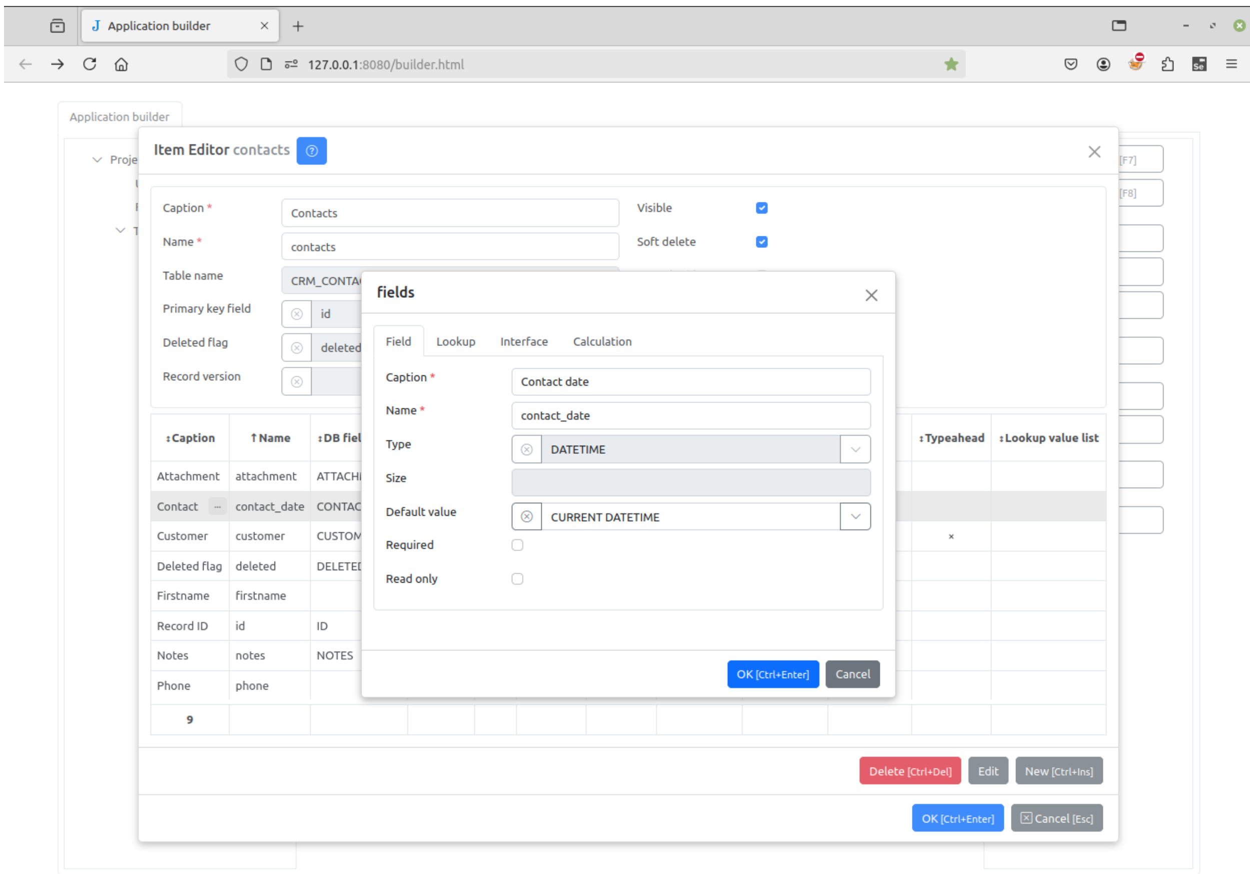


最后，在保存之前，打开我们之前创建的“业务客户 (Customer)”字段对话框，在“字段 (Field)”选项卡中选中 **必填 (Required)** 属性，在“查找 (Lookup)”选项卡中选中 **预输入 (Typeahead)** 属性。当勾选 **预输入 (Typeahead)** 时，将为查找字段启用自动补全/预输入功能。

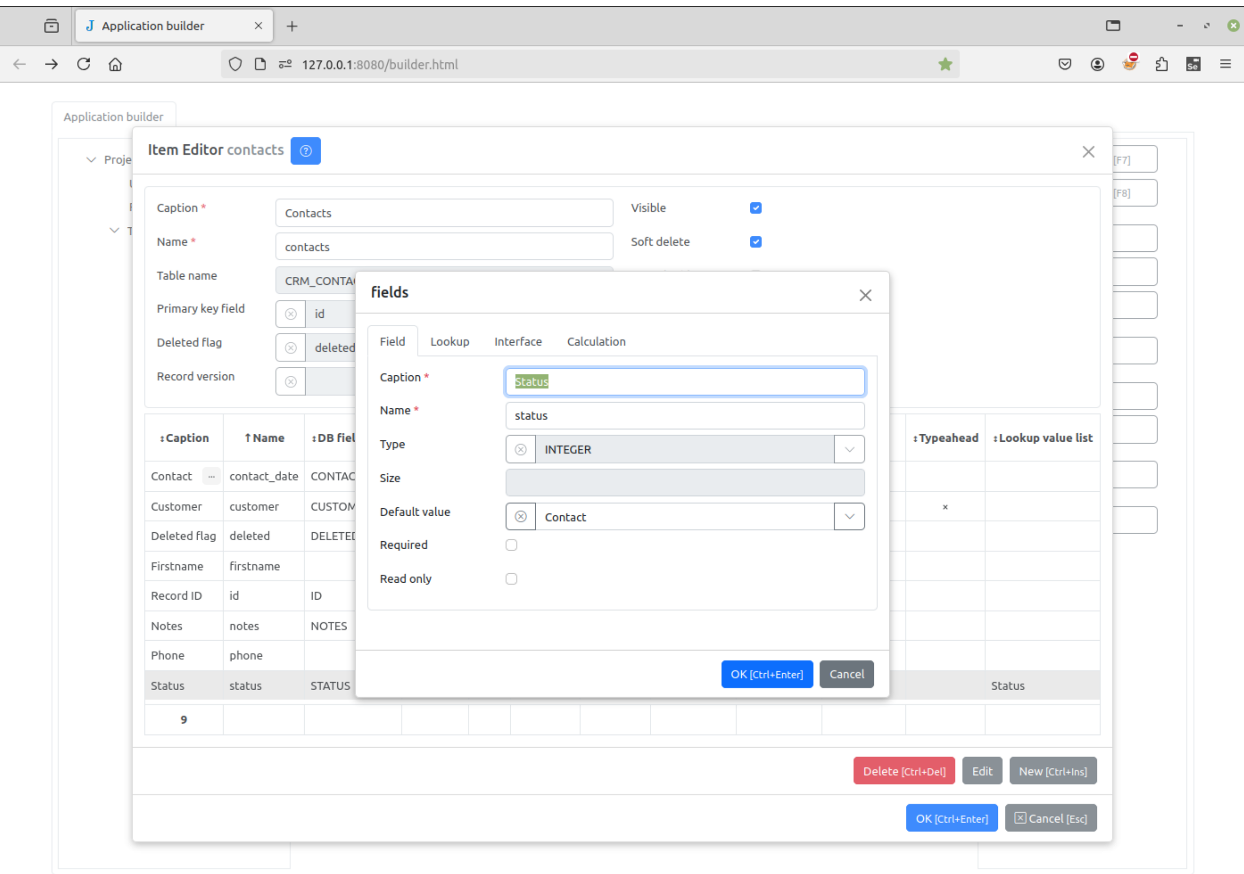

Application builder
127.0.0.1:8080/builder.html
Item Editor contacts
Caption *
Contacts
Name *
contacts
Table name
CRM_CONTA
Primary key field
id
Deleted flag
deleted
Record version
Visible
Soft delete
fields
Field
Lookup
Interface
Calculation
Caption *
Customer
Name *
customer
Type
INTEGER
Size
Default value
Required
Read only
OK [Ctrl+Enter]
Cancel
Caption
Name
Typeahead
Lookup value list
Customer
customer
Deleted flag
deleted
Firstname
firstname
Record ID
id
ID
Notes
notes
NOTES
Phone
phone
Status
status
STATUS
9
Delete [Ctrl+Del]
Edit
New [Ctrl+Ins]
Cancel [Esc]



在这里，我们还可以将“联系日期 (Contact date)”字段的**默认值 (Default value)** 设置为“当前日期时间 (CURRENT DATETIME)”，这样日期将自动初始化为当前日期和时间。

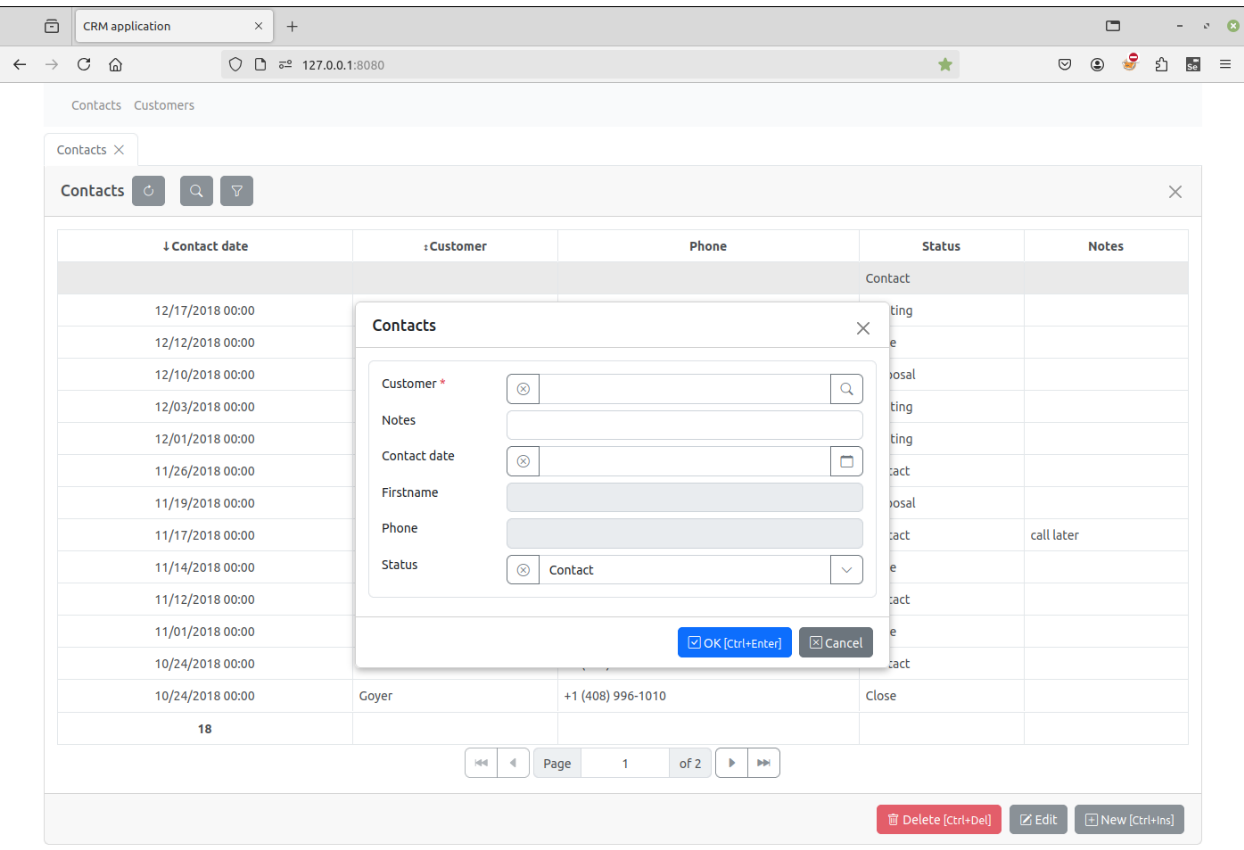


同样，我们可以为“状态 (Status)”字段选择 **默认值 (Default value)**，方法是在下拉列表中选择一个值。

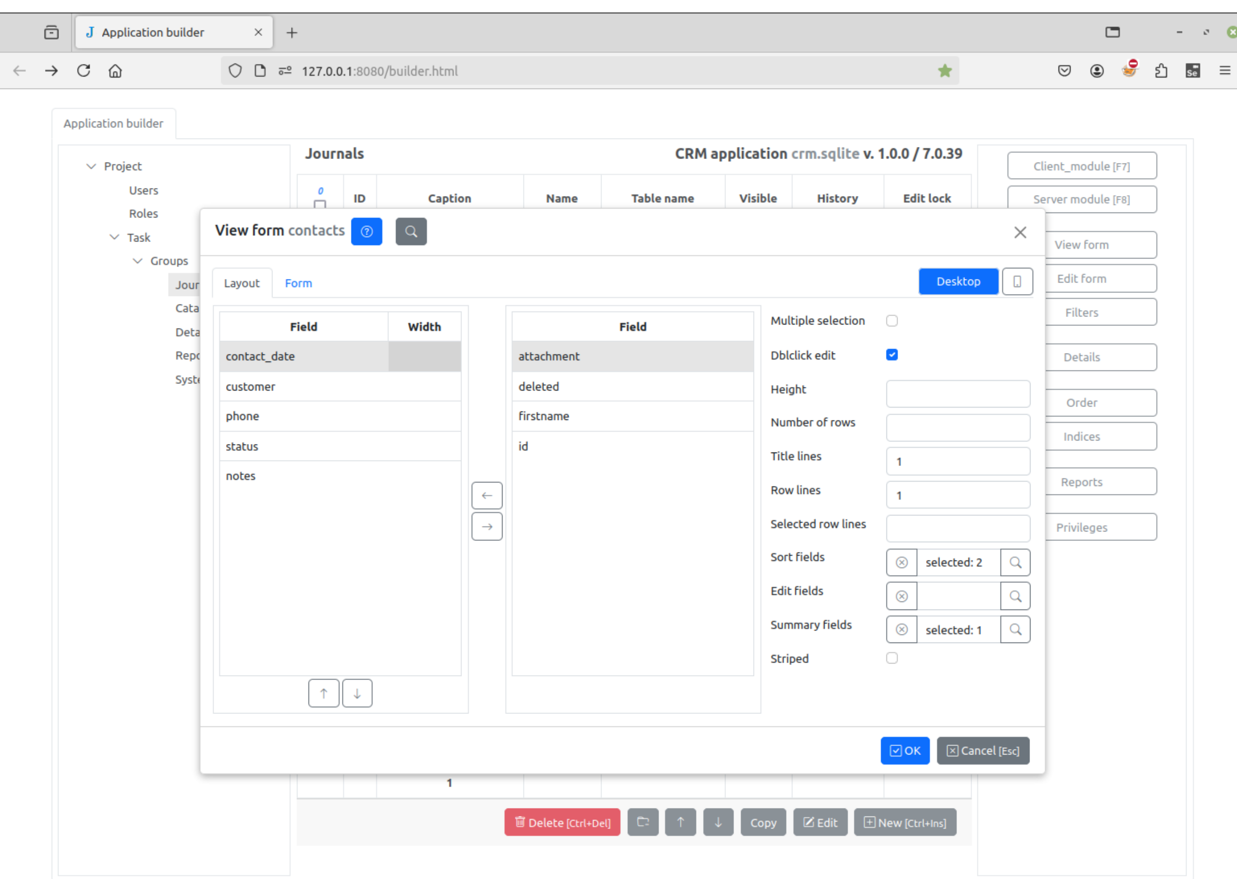


2.4.5 自定义表单

当我们刷新客户端的项目页面时，会看到“联系人 (Contacts)”业务台账的表格和编辑表单中的字段是按照它们被创建的顺序显示的。

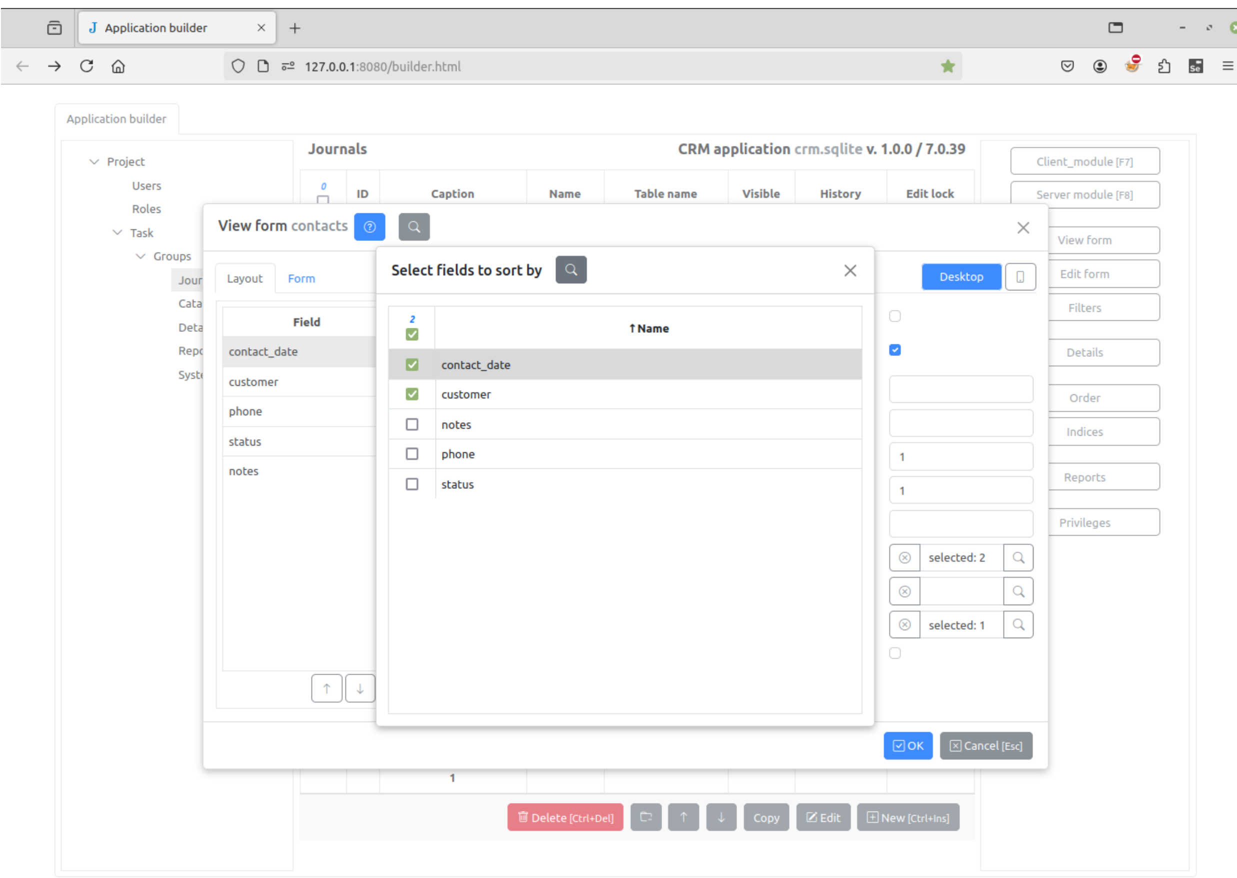


要更改字段在客户端的表格中的显示方式，请选择“联系人 (Contacts)”业务台账项，再点击 **查看表单 (View Form)** 按钮打开查看表单对话框。让我们使用“方向箭头”（← 、→ 、↑ 和 ↓ ）按钮来更改显示的字段。

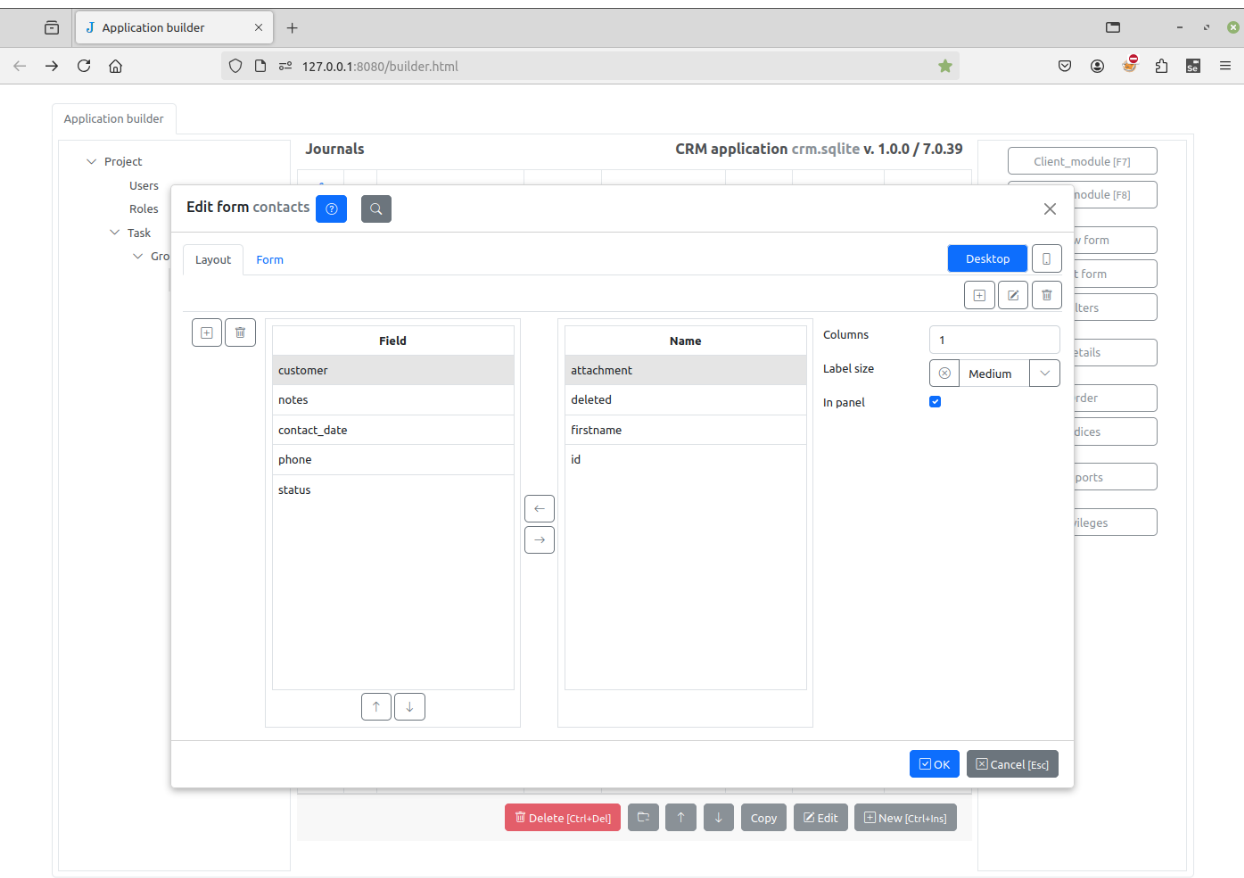


在上面的示例中，我们通过选择“名字 (firstname)”字段并按下 → 按钮将其隐藏（移到右边的字段列表中），并通过选择“备注 (notes)”字段并多次按下 ↓ 按钮将其移到最后。

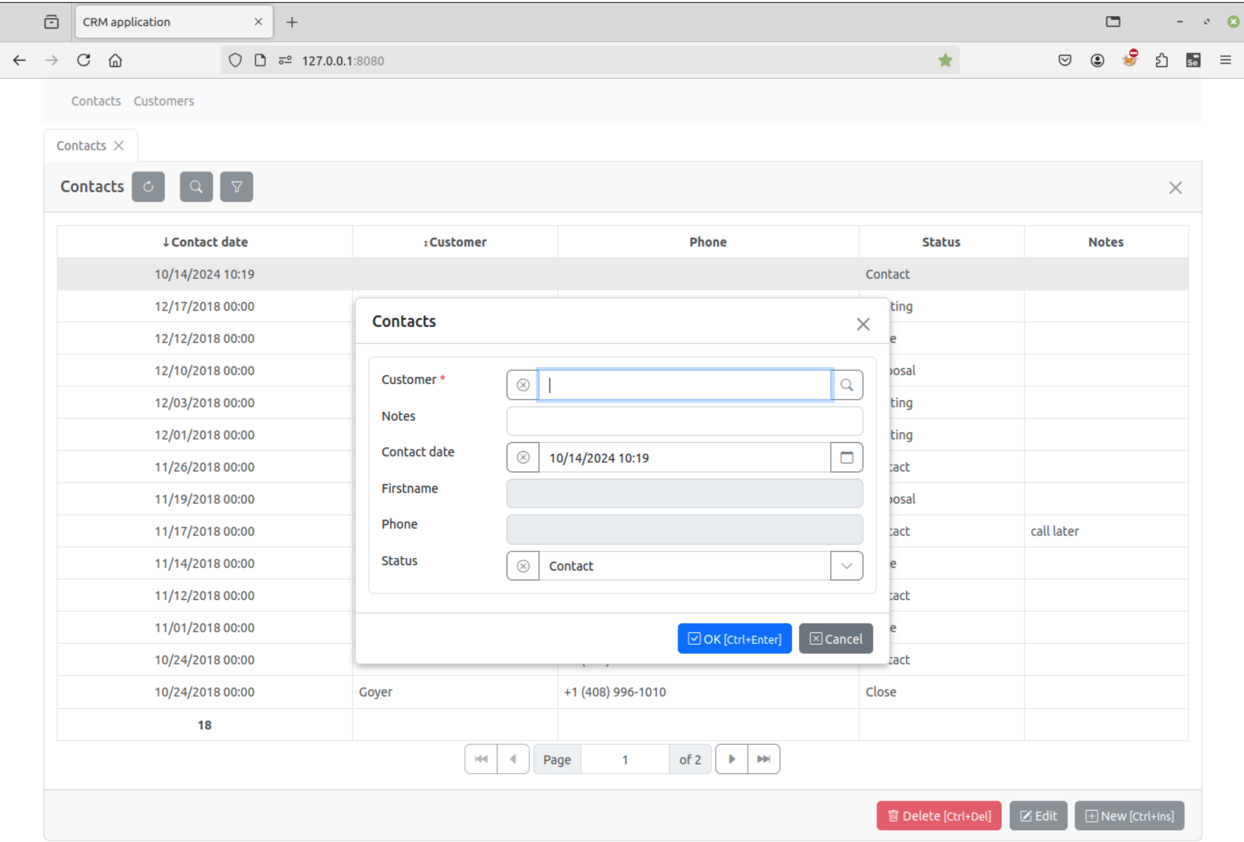
我们还可以更改哪些字段可用于对表格的数据进行排序。为此，点击 **排序字段 (Sort fields)** 输入框右侧的按钮，并选择表格中对应的列标题。然后点击 **确定 (OK)** 保存所有更改。



要更改字段在客户端的编辑表单中的显示方式，请点击 **编辑表单 (Edit Form)** 按钮打开编辑表单对话框。这与上面的 **查看表单 (View Form)** 操作方式非常相似。



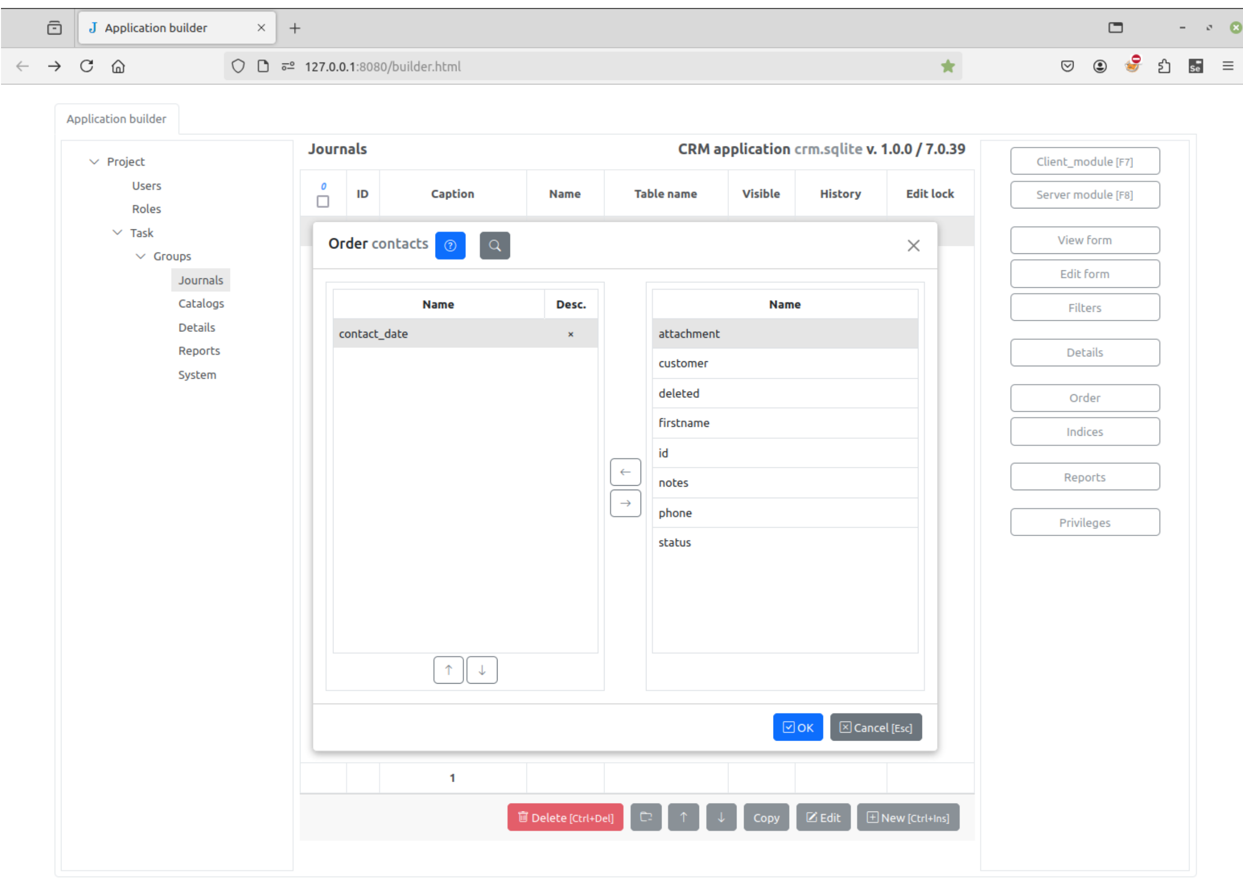
要查看我们的工作成果，请转到客户端的项目页面并刷新，然后点击 **新建 (New)** 按钮。



2.4.6 索引

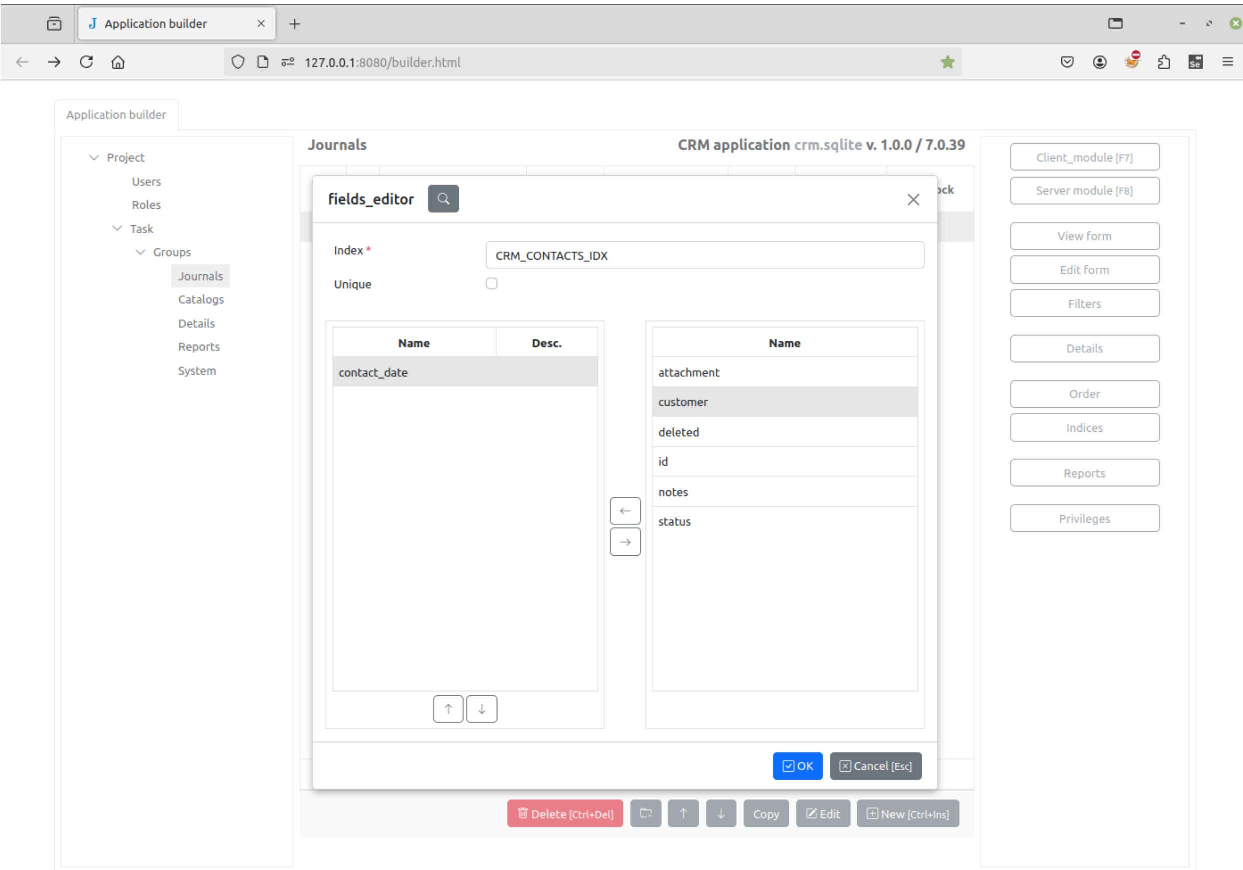
让我们设置“联系人 (Contacts)”业务台账中的记录在客户端页面中的默认排序。为此，点击排序 按钮。

默认情况下，记录按创建顺序显示。要更改此设置，使用**←**（左箭头）按钮将右侧列表中的某些列标题移动到左侧列表中，并使用**↑**（上箭头）和**↓**（下箭头）按钮更改其优先级（越靠上，优先级越高）。



在上面的示例中，我们设置默认排序为“按联系日期降序”排列。

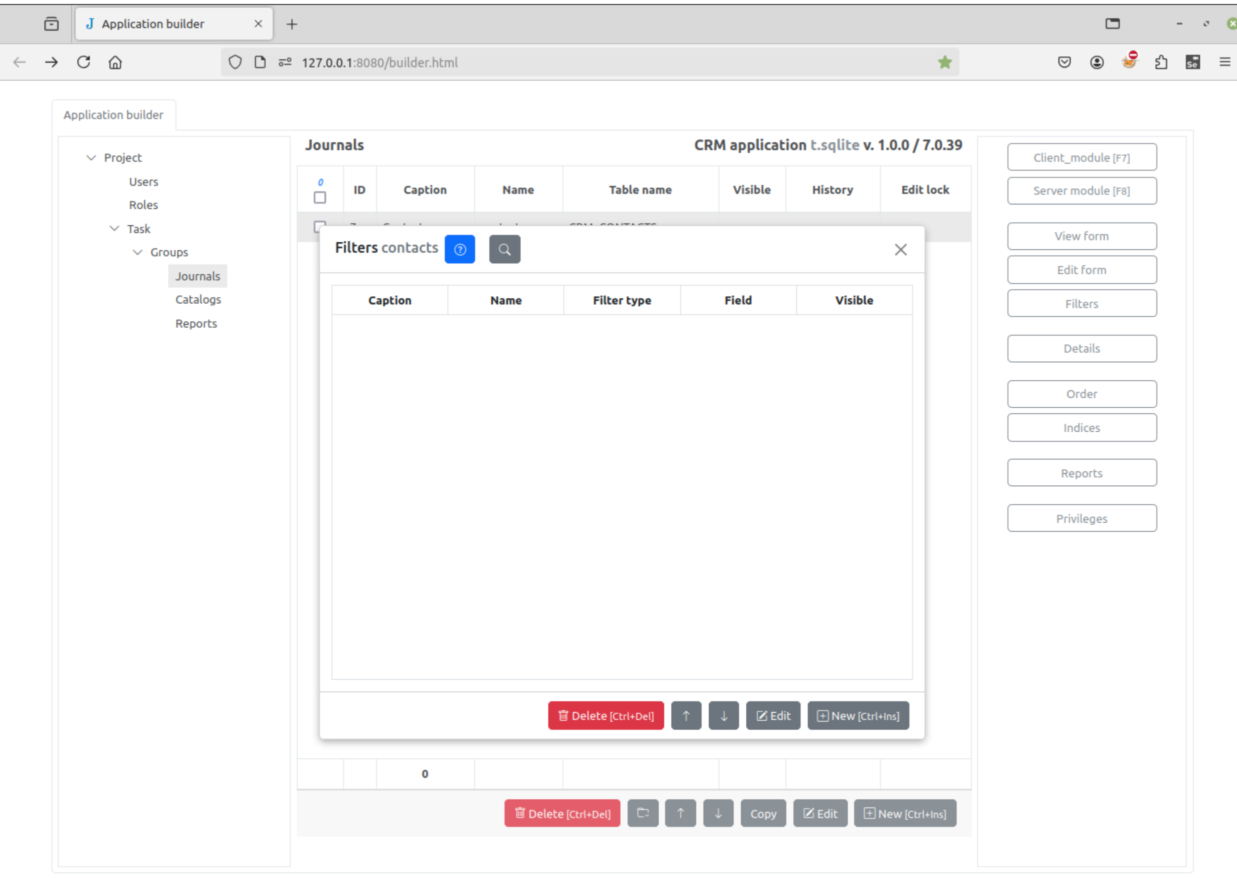
我们可以为“联系人 (Contacts)”业务台账数据库表创建相应的索引。点击 **索引 (Indices)** 按钮打开索引对话框，然后点击 **新建 (New)** 按钮并以类似方式指定索引：



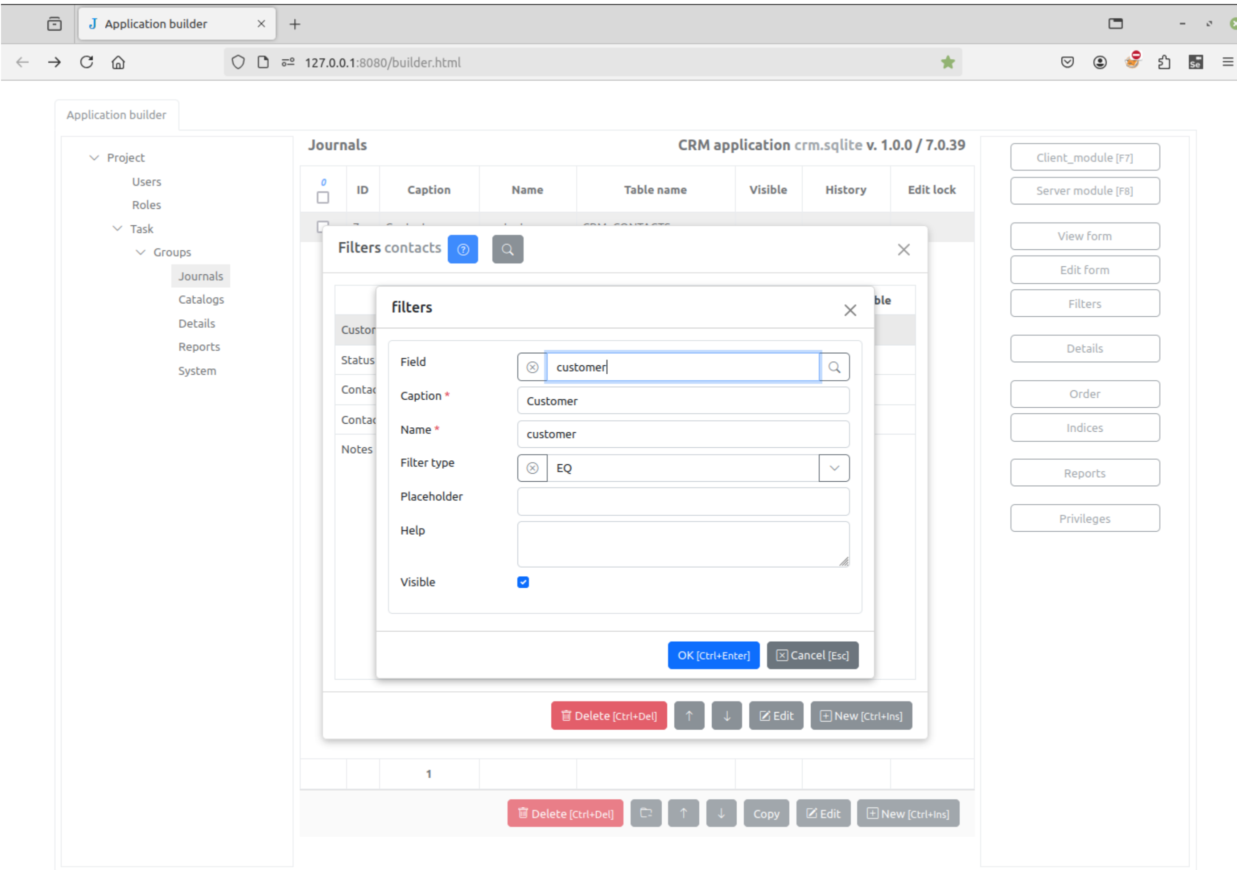
2.4.7 过滤器

过滤器 用于根据其指定的条件从数据库表中选择记录。

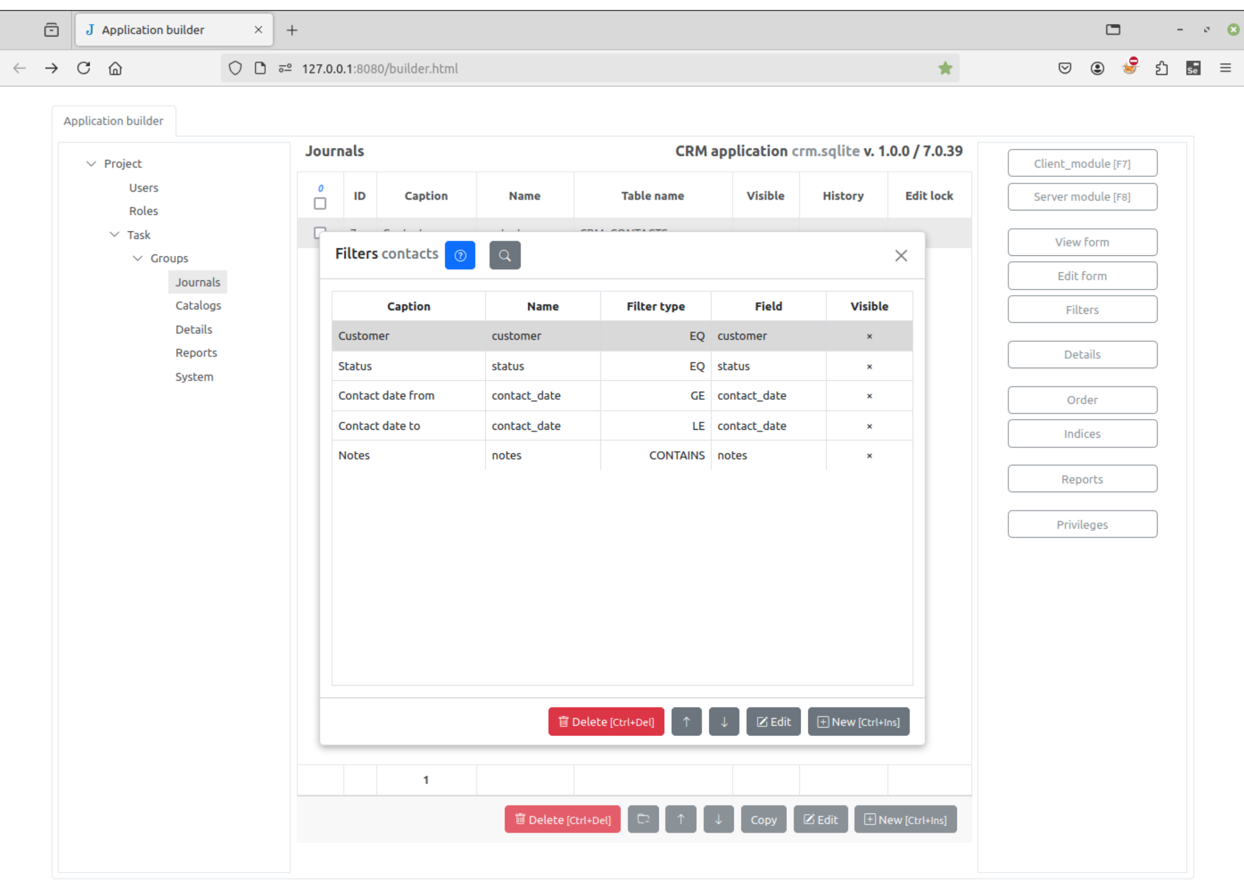
点击 **过滤器 (Filters)** 按钮打开过滤器对话框。



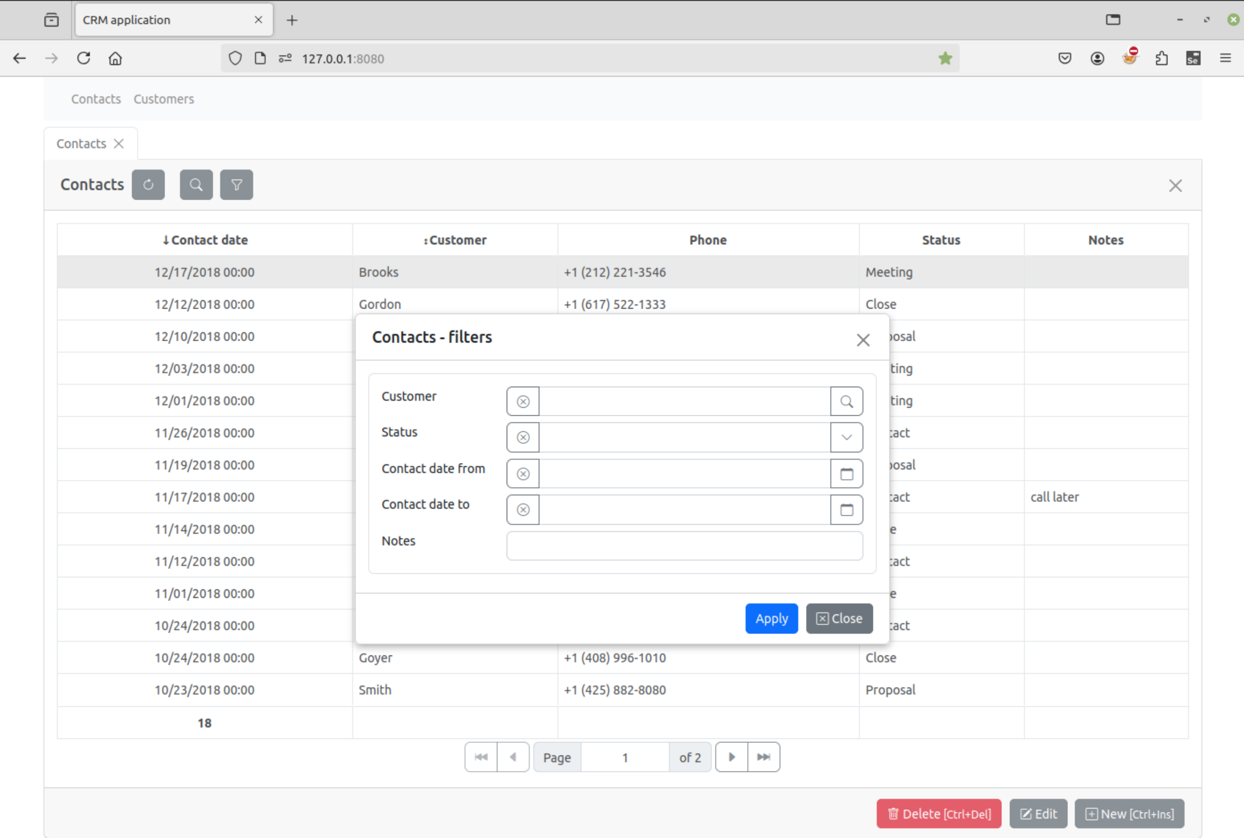
现在点击 **新建 (New)** 按钮并填写以下表单：



类似地，创建其他几个过滤器：



当我们刷新客户端的项目页面时，带 **漏斗图标**的过滤器按钮将出现在“联系人 (Contacts)”表单页面的标题栏中。点击此按钮将打开“过滤器”对话框：

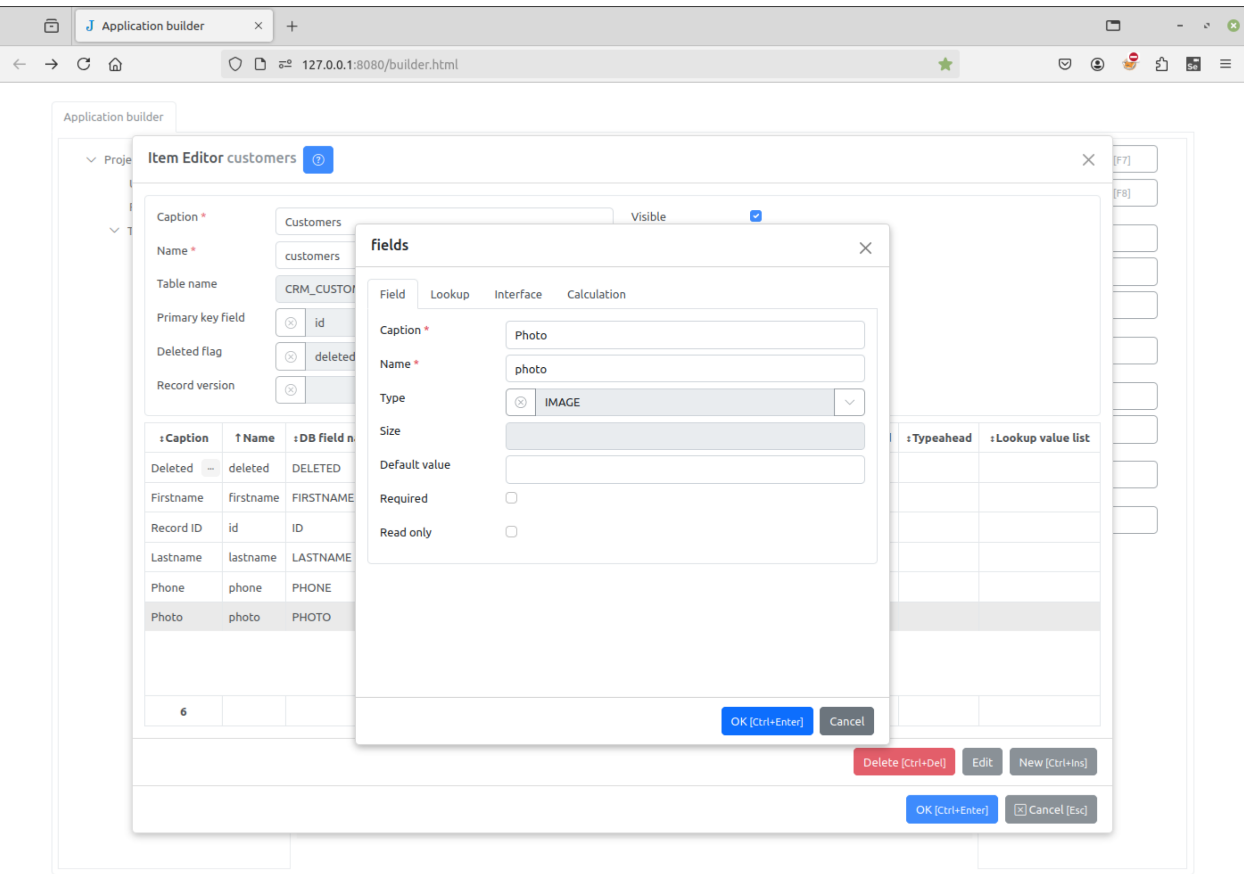


2.5 教程第二部分：文件和图像字段

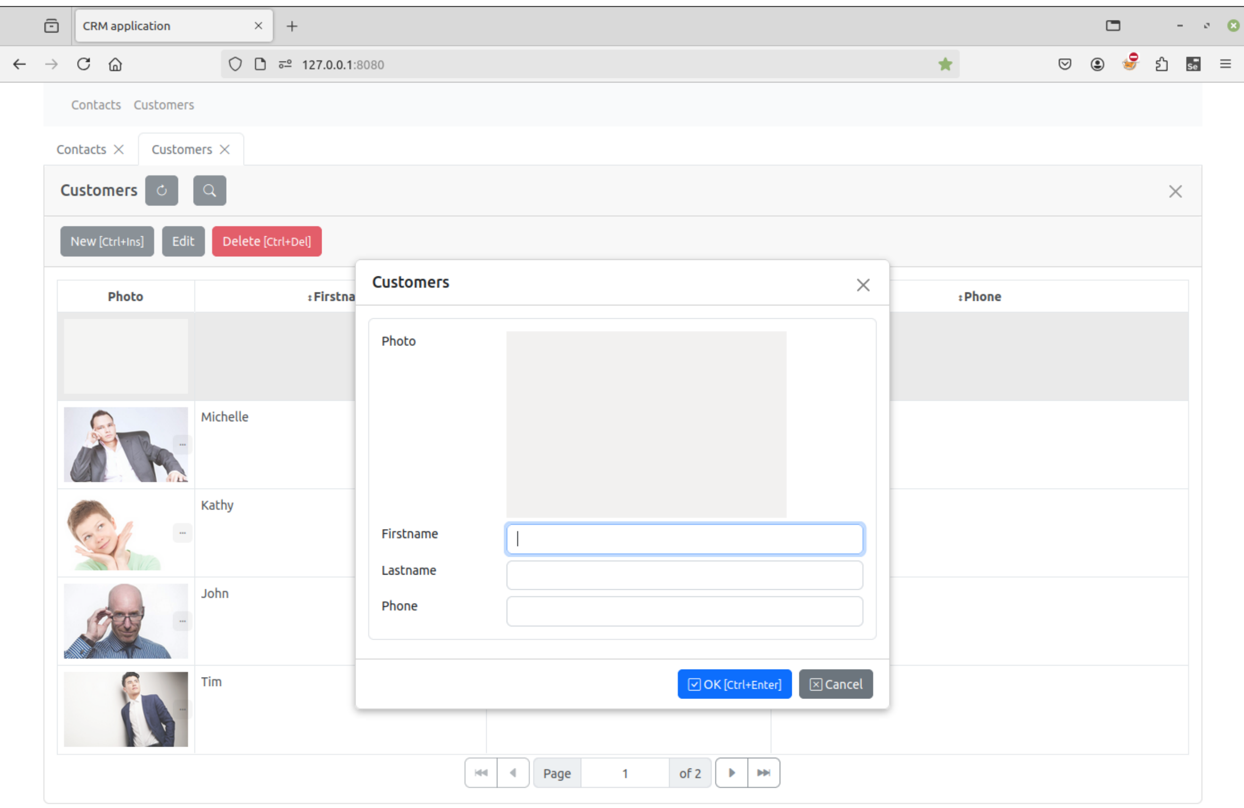
在本部分中，我们将演示如何在 Jam.py 中处理文件和图像。

2.5.1 添加图像字段

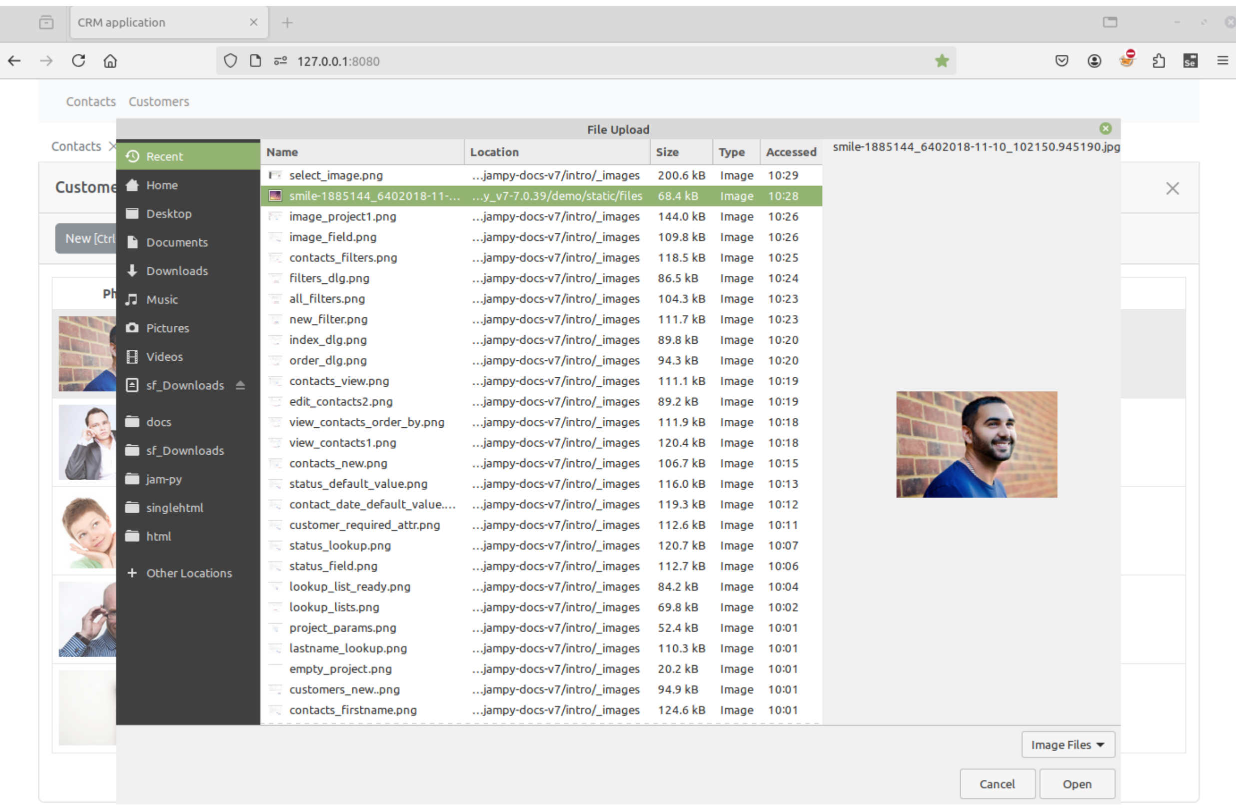
让我们在主表目录中选择“客户 (Customers)”，双击它以打开实体项编辑器对话框 并添加一个图像字段“照片 (Photo)”：

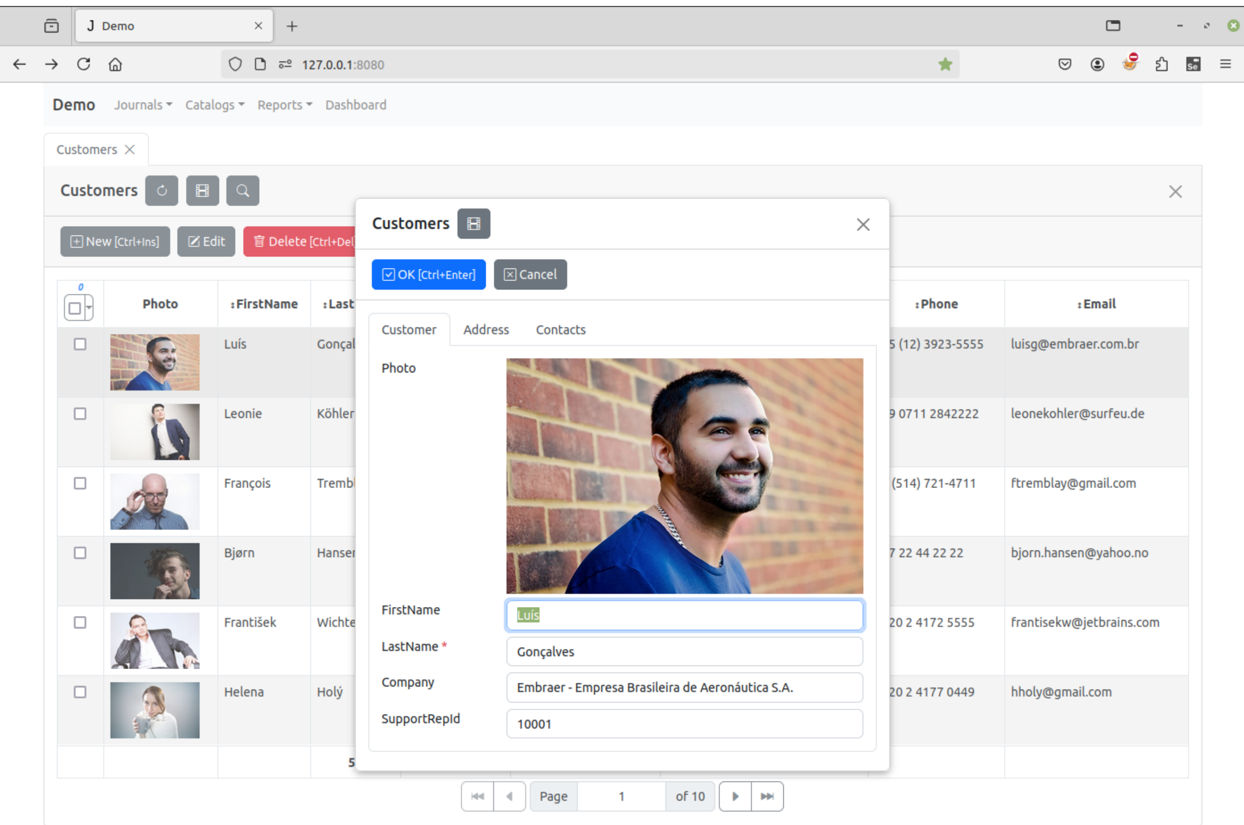


现在，刷新客户端的项目页面，点击“客户 (Customers)”菜单项并选中一条数据，然后打开编辑表单。



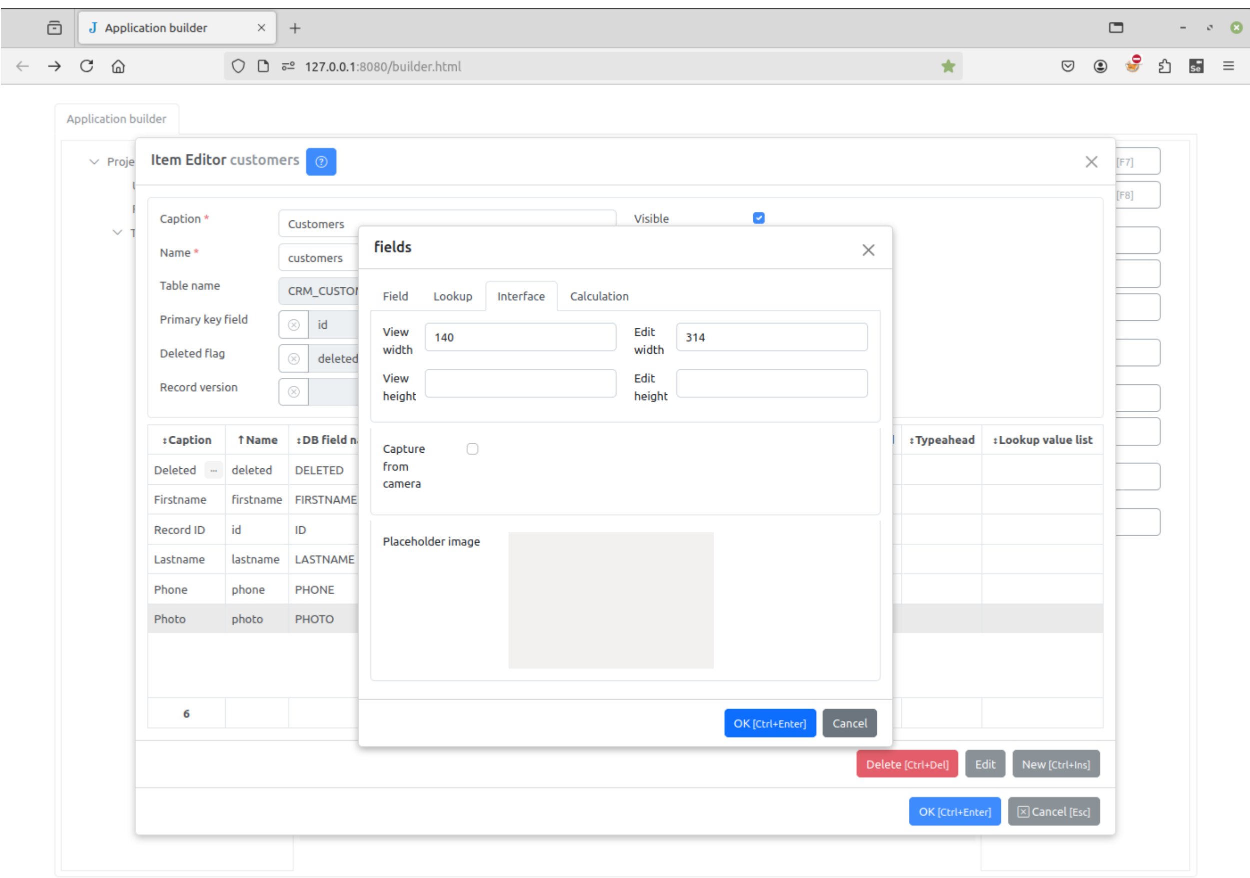
双击编辑表单中的图像，从“打开”对话框中选择一张图像文件。


CRM application
127.0.0.1:8080
Contacts
Customers
File Upload
Recent
Home
Desktop
Documents
Downloads
Music
Pictures
Videos
sf_Downloads
docs
sf_Downloads
jam-py
singlehtml
html
Other Locations
Name
Location
Size
Type
Accessed
select_image.png ...jampy-docs-v7/intro/_images 200.6 kB Image 10:29
smile-1885144_6402018-11-... ...y_v7-7.0.39/demo/static/files 68.4 kB Image 10:28
image_project1.png ...jampy-docs-v7/intro/_images 144.0 kB Image 10:26
image_field.png ...jampy-docs-v7/intro/_images 109.8 kB Image 10:26
contacts_filters.png ...jampy-docs-v7/intro/_images 118.5 kB Image 10:25
filters_dlg.png ...jampy-docs-v7/intro/_images 86.5 kB Image 10:24
all_filters.png ...jampy-docs-v7/intro/_images 104.3 kB Image 10:23
new_filter.png ...jampy-docs-v7/intro/_images 111.7 kB Image 10:23
index_dlg.png ...jampy-docs-v7/intro/_images 89.8 kB Image 10:20
order_dlg.png ...jampy-docs-v7/intro/_images 94.3 kB Image 10:20
contacts_view.png ...jampy-docs-v7/intro/_images 111.1 kB Image 10:19
edit_contacts2.png ...jampy-docs-v7/intro/_images 89.2 kB Image 10:19
view_contacts_order_by.png ...jampy-docs-v7/intro/_images 111.9 kB Image 10:18
view_contacts1.png ...jampy-docs-v7/intro/_images 120.4 kB Image 10:18
contacts_new.png ...jampy-docs-v7/intro/_images 106.7 kB Image 10:15
status_default_value.png ...jampy-docs-v7/intro/_images 116.0 kB Image 10:13
contact_date_default_value.... ...jampy-docs-v7/intro/_images 119.3 kB Image 10:12
customer_required_attr.png ...jampy-docs-v7/intro/_images 112.6 kB Image 10:11
status_lookup.png ...jampy-docs-v7/intro/_images 120.7 kB Image 10:07
status_field.png ...jampy-docs-v7/intro/_images 112.7 kB Image 10:06
lookup_list_ready.png ...jampy-docs-v7/intro/_images 84.2 kB Image 10:04
lookup_lists.png ...jampy-docs-v7/intro/_images 69.8 kB Image 10:02
project_params.png ...jampy-docs-v7/intro/_images 52.4 kB Image 10:01
lastname_lookup.png ...jampy-docs-v7/intro/_images 110.3 kB Image 10:01
empty_project.png ...jampy-docs-v7/intro/_images 20.2 kB Image 10:01
customers_new..png ...jampy-docs-v7/intro/_images 94.9 kB Image 10:01
contacts_firstname.png ...jampy-docs-v7/intro/_images 124.6 kB Image 10:01
smile-1885144_6402018-11-10_102150.945190.jpg
Image Files
Cancel
Open



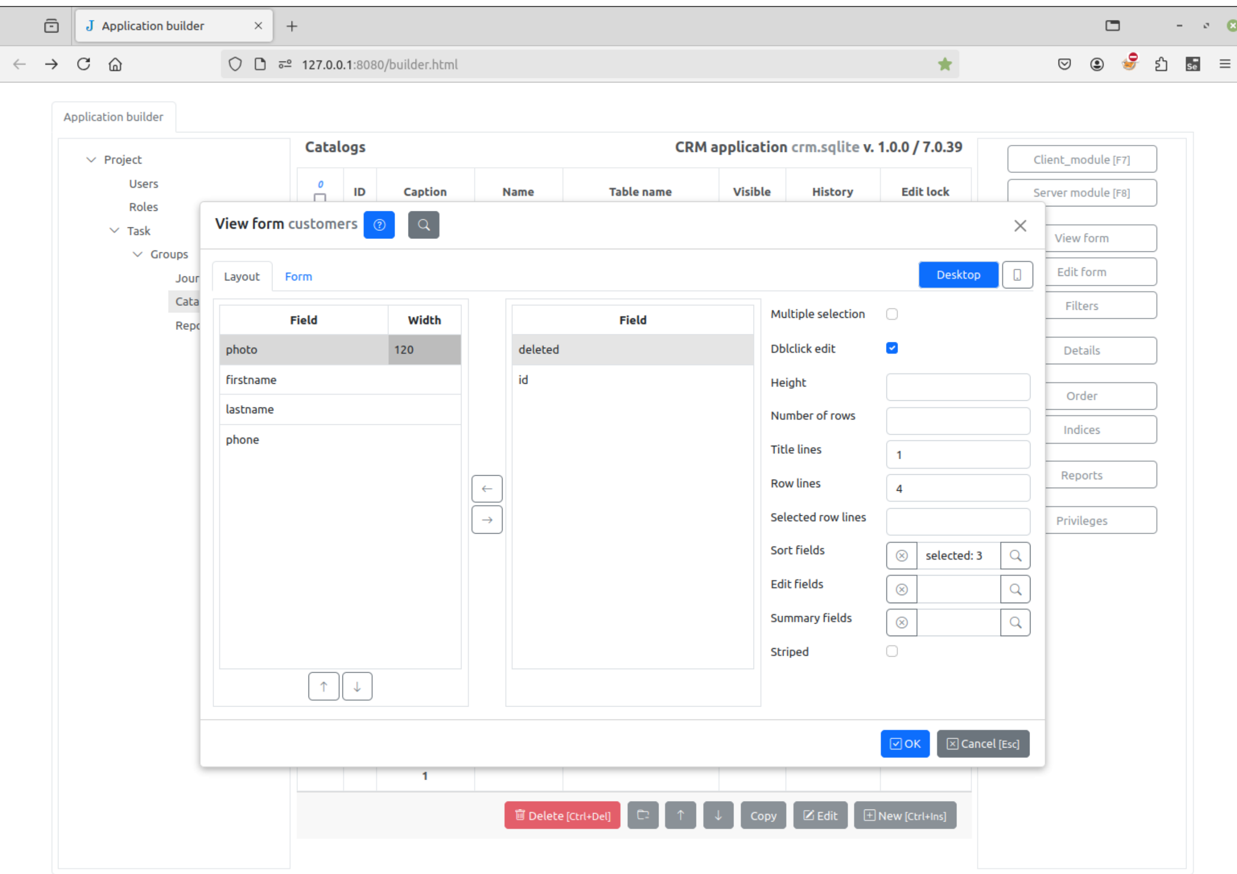
> **备注**
>
> 要清除图像，请按住 Ctrl 键并双击图像。

像本节开始一样，让我们在应用程序构建器中打开字段编辑器对话框 并在 **界面 (Interface)** 选项卡上将 **查看宽度 (View width)** 设置为 120，将 **编辑宽度 (Edit width)** 设置为 314。

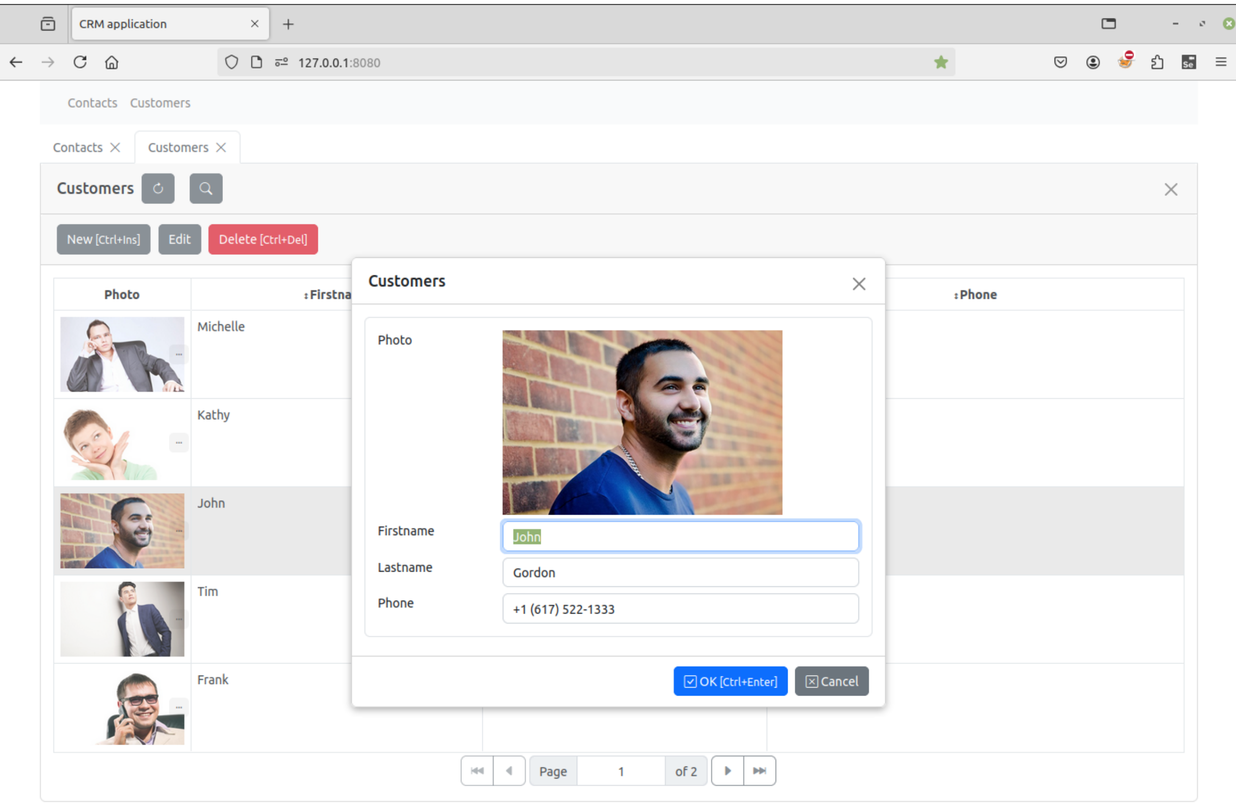


> **备注**
>
> 您可以通过双击来设置图像占位符。

在查看表单对话框中，我们将 **行数 (Row lines)** 设置为 4（每条记录的高度占据 4 行文本的高度），并将“照片”字段的宽度设置为 120。



现在在客户端的项目页面上我们将看到:



另请参阅

可接受的字符串

2.5.2 从摄像头捕获图像

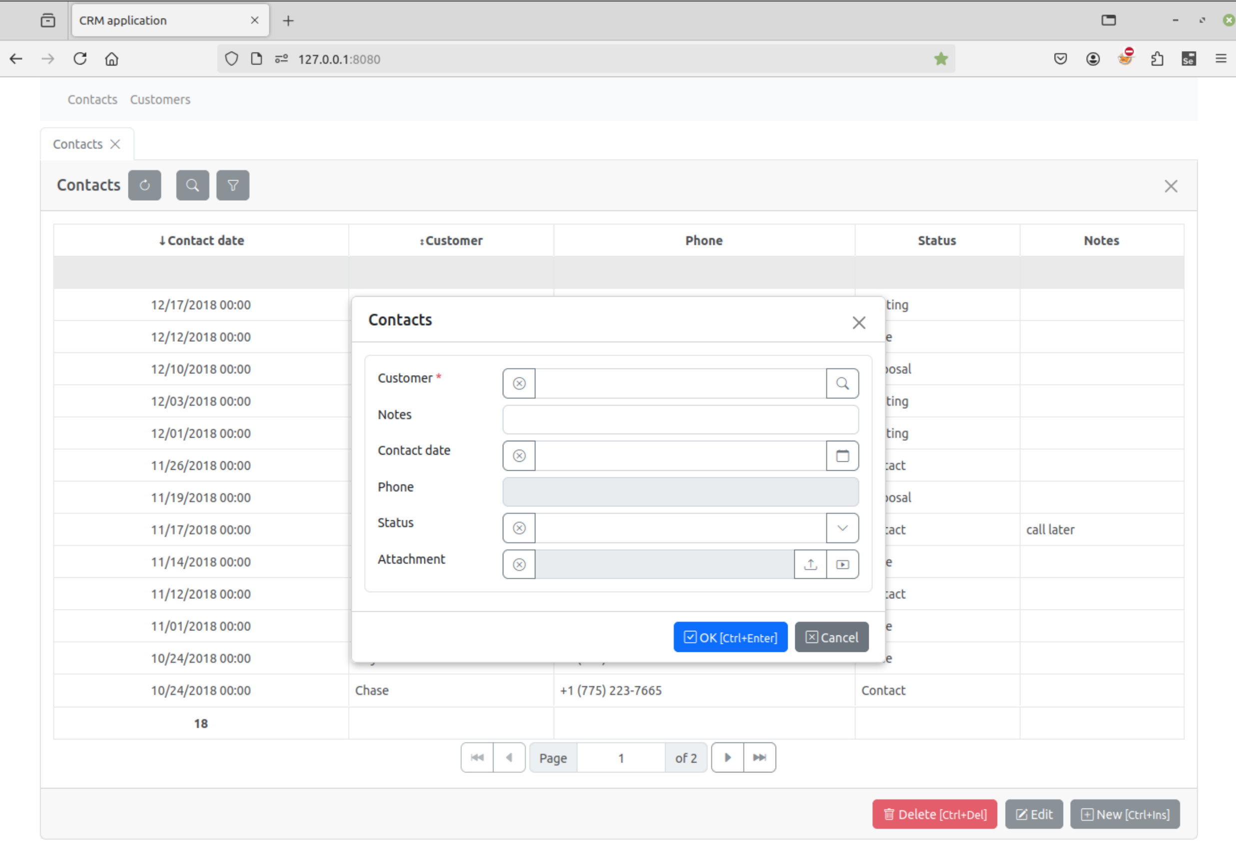
您可以从摄像头捕获图像。为此，在应用程序构建器中打开字段编辑器对话框，在 **界面 (Interface)** 选项卡勾选 **从摄像头捕获 (Capture from camera)** 复选框。在这种情况下，当图像未设置时，将显示摄像头视频流而不是图像占位符。

双击视频以捕获图像。要清除图像，请按住 Ctrl 键并双击图像，之后将显示视频流。

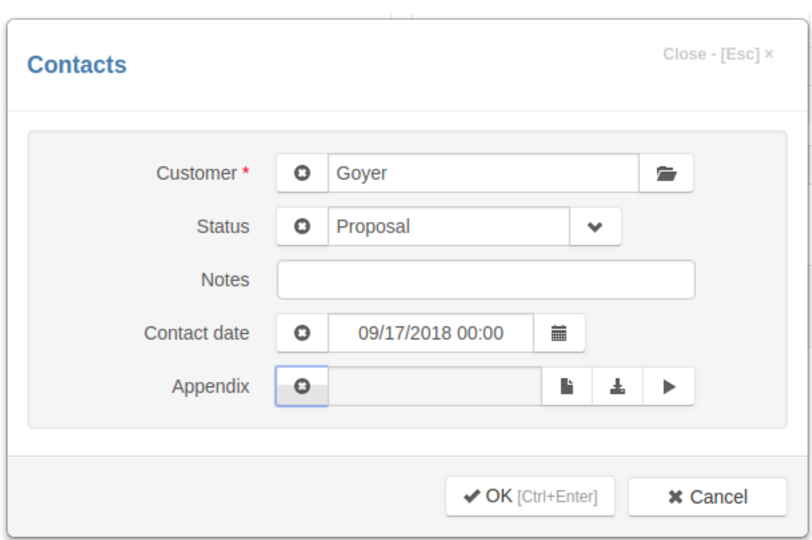
图像会自动上传到服务器，前提是应用程序的项目参数 中的可接受的值列表 中添加了 “.png”、“.jpg” 等图像文件的扩展名。

2.5.3 添加文件字段

现在，我们向业务台账中的“联系人 (Contacts)”台账中添加一个用于存储附件的字段“附件 (attachment)”，字段类型为“FILE”。



该字段在编辑表单中将如下显示：

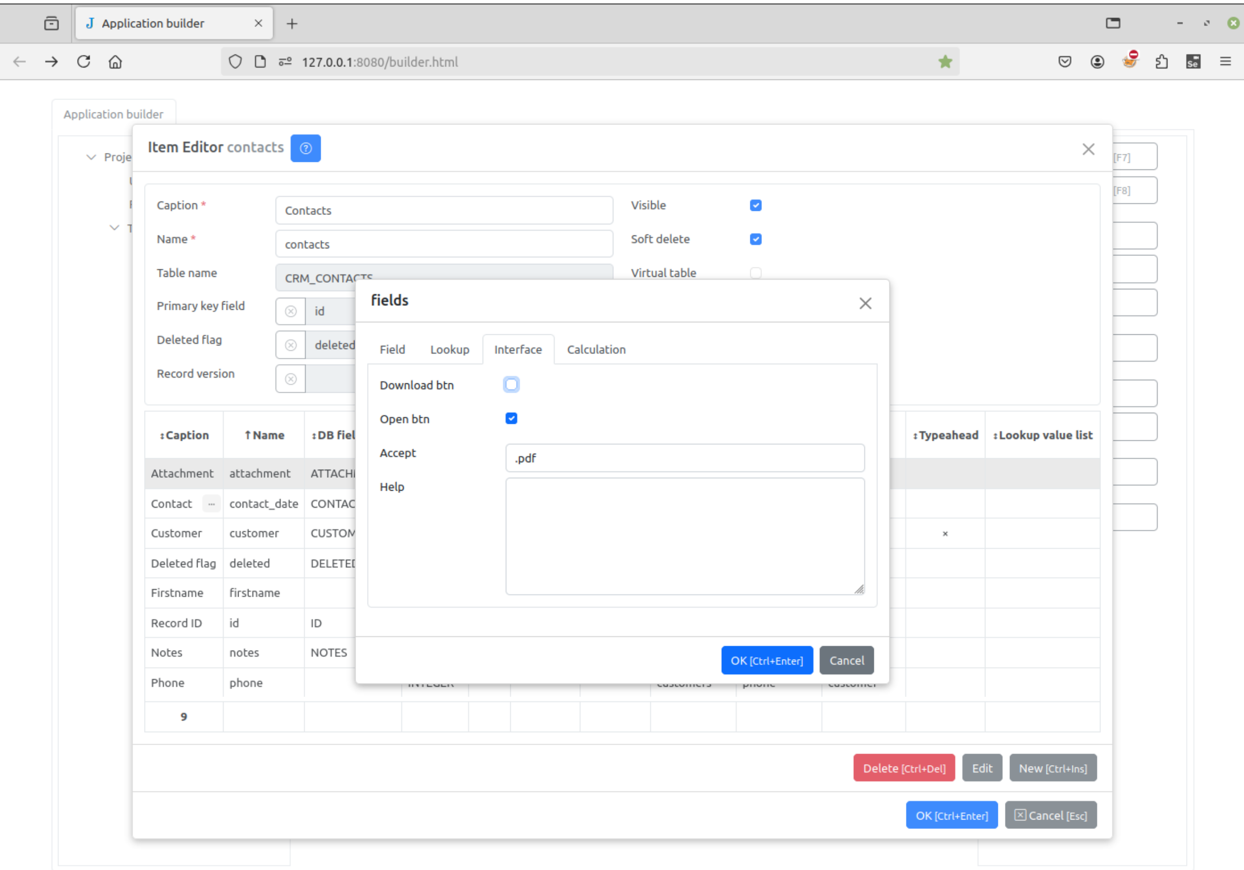


字段输入框右侧有三个按钮——用于上传、下载和打开文件。

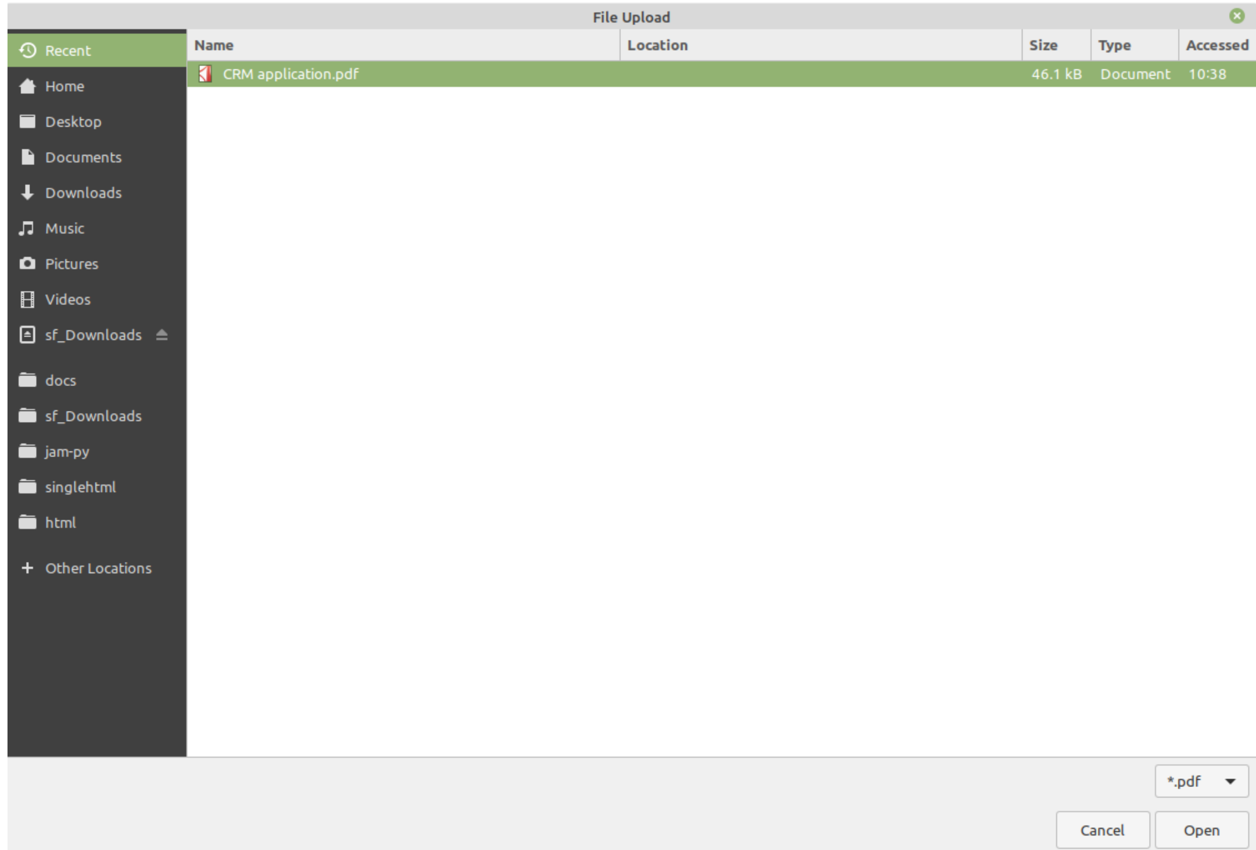
在应用程序构建器中打开字段编辑器对话框 在 **界面 (Interface)** 选项卡中取消勾选 **下载按钮 (Download btn)**

复选框，并将 **接受 (Accept)** 属性设置为'.pdf'。

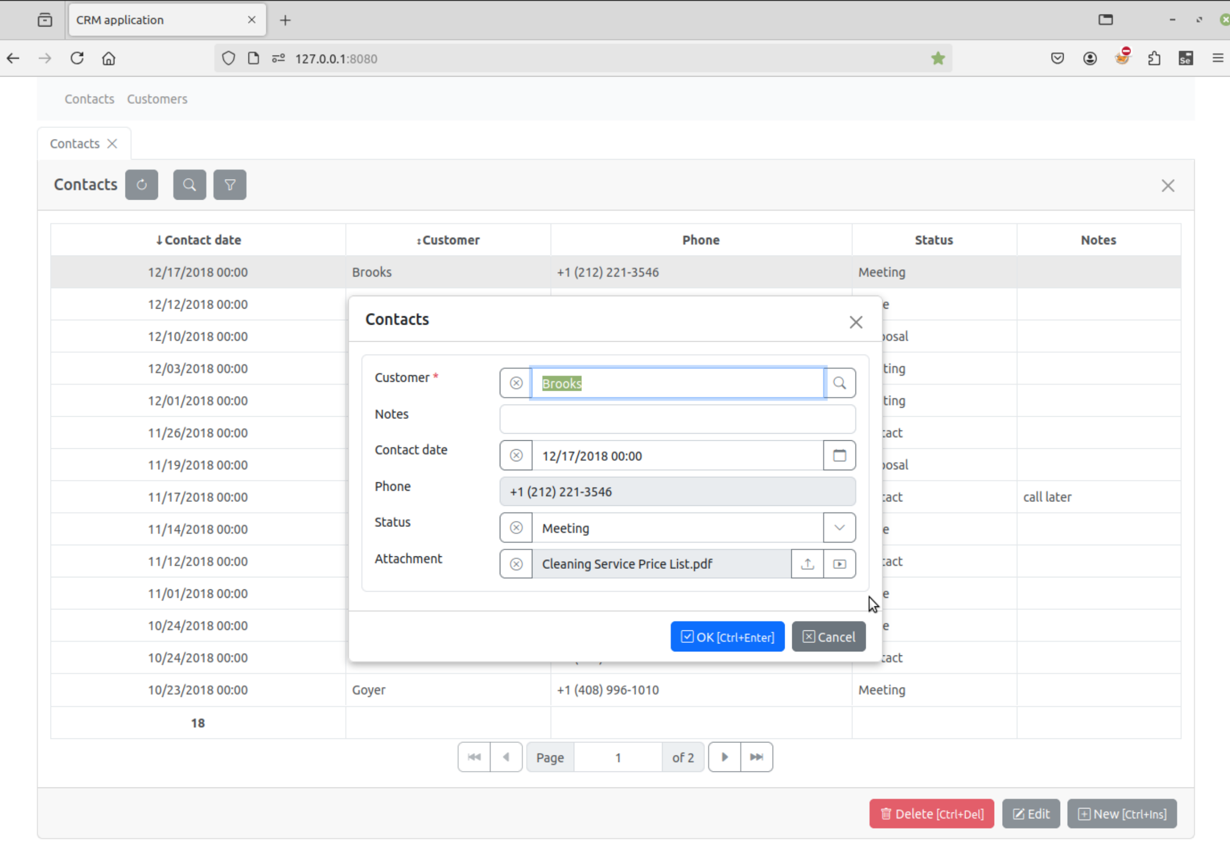
在添加值之前，请查看可接受的值列表。



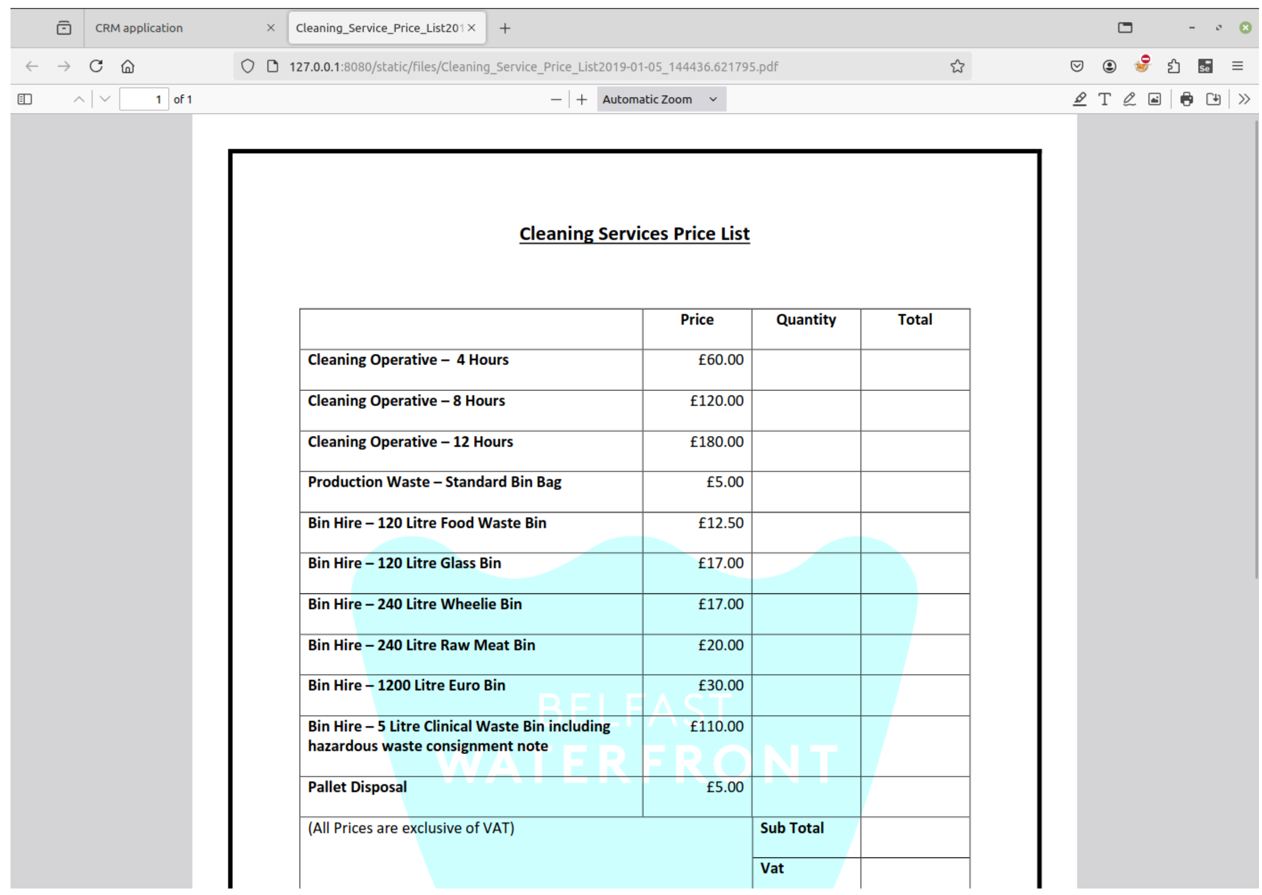
让我们刷新客户端的项目页面，打开"联系人 (Contacts)"编辑表单，通过点击上传按钮上传文件：



现在，我们可以通过点击打开按钮在浏览器中打开文件。



CRM application
127.0.0.1:8080
Contacts
Customers
Contacts
Contacts
Contact date
Customer
Phone
Status
Notes
12/17/2018 00:00
Brooks
+1 (212) 221-3546
Meeting
12/12/2018 00:00
12/10/2018 00:00
12/03/2018 00:00
12/01/2018 00:00
11/26/2018 00:00
11/19/2018 00:00
11/17/2018 00:00
call later
11/14/2018 00:00
11/12/2018 00:00
11/01/2018 00:00
10/24/2018 00:00
10/24/2018 00:00
10/23/2018 00:00
Goyer
+1 (408) 996-1010
Meeting
18
Page
1
of 2
Contacts
Customer *
Brooks
Notes
Contact date
12/17/2018 00:00
Phone
+1 (212) 221-3546
Status
Meeting
Attachment
Cleaning Service Price List.pdf
OK [Ctrl+Enter]
Cancel
Delete [Ctrl+Del]
Edit
New [Ctrl+Ins]



Cleaning Services Price List

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

备注

文件和图像存储在服务器上的 *static/files* 文件夹中。

在服务端应用程序构建器页面中，您可以通过选择“项目 (Project)”后，再点击页面右侧的参数 来设置 **最大内容长度 (Max content length)** 属性来限制可以上传到服务器的文件大小。

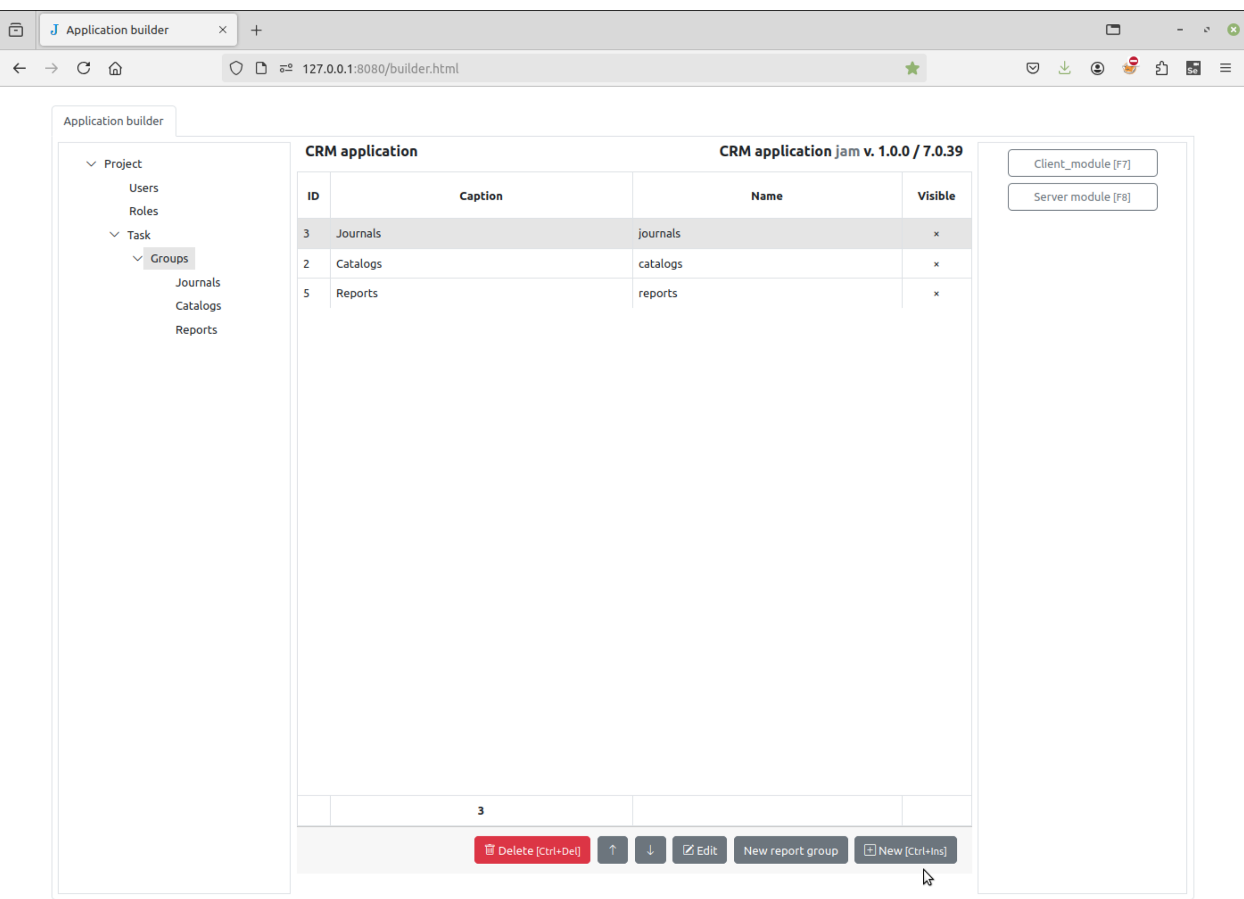
另请参阅

可接受的字符串

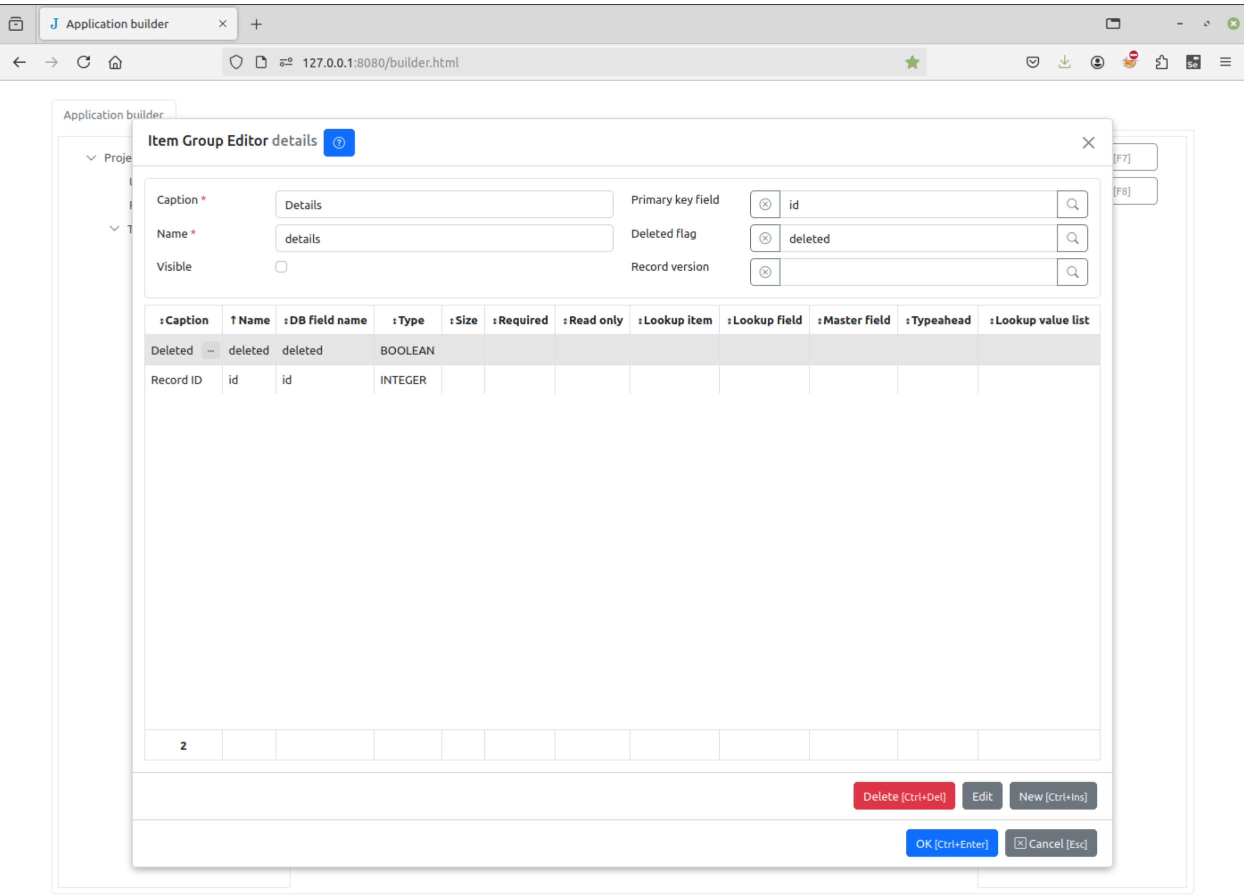
2.6 教程第三部分：明细表

在本部分教程中，我们将解释如何使用 **明细表 (Details)** 组。

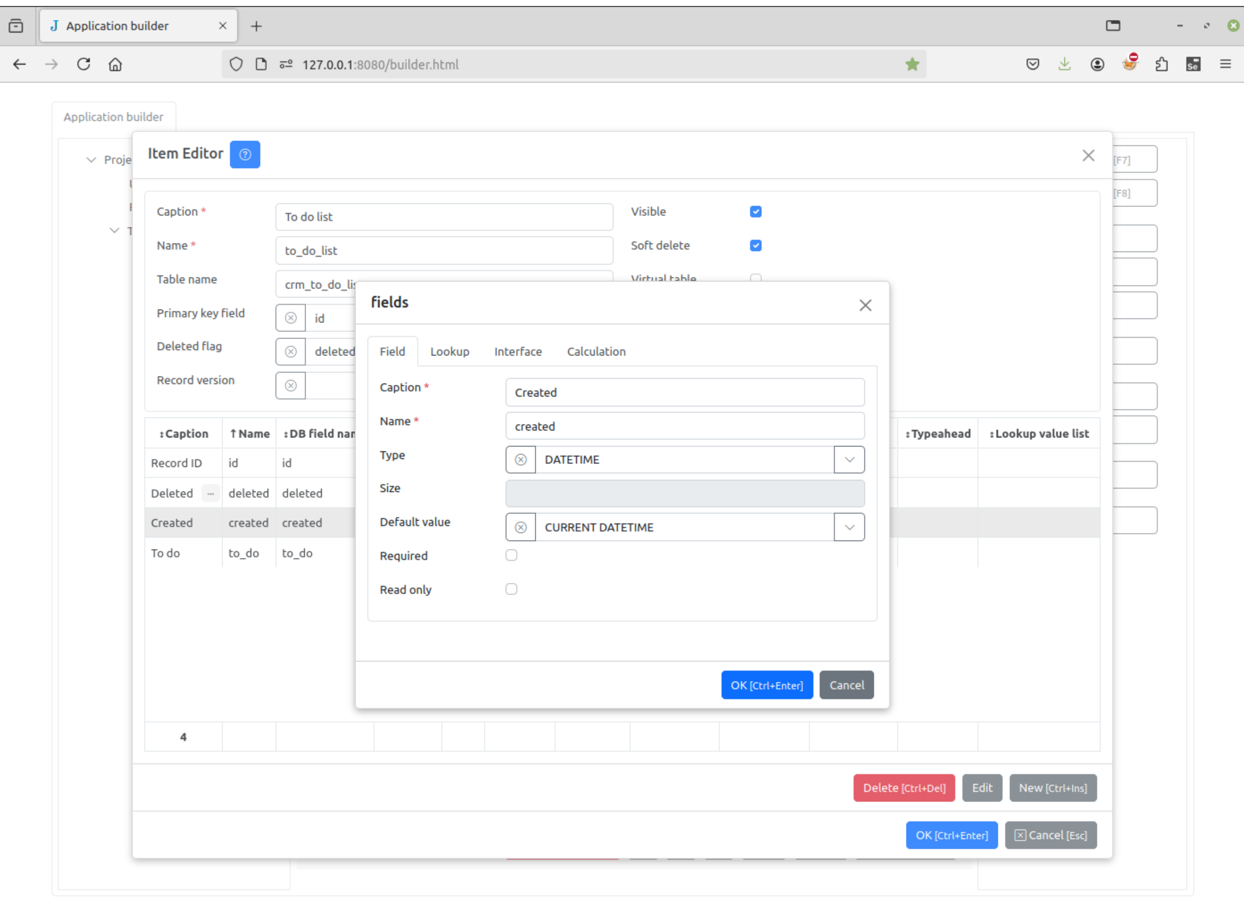
让我们在项目树中选择“任务 (Task)/分组 (Groups)”，并点击页面右下角的 **新建 (New)** 按钮，新建一个组：



将其命名为"明细表 (Details)",并添加两个字段 ID(id, 类型为 INTEGER)、已删除 (deleted, 类型为 BOOLEAN):

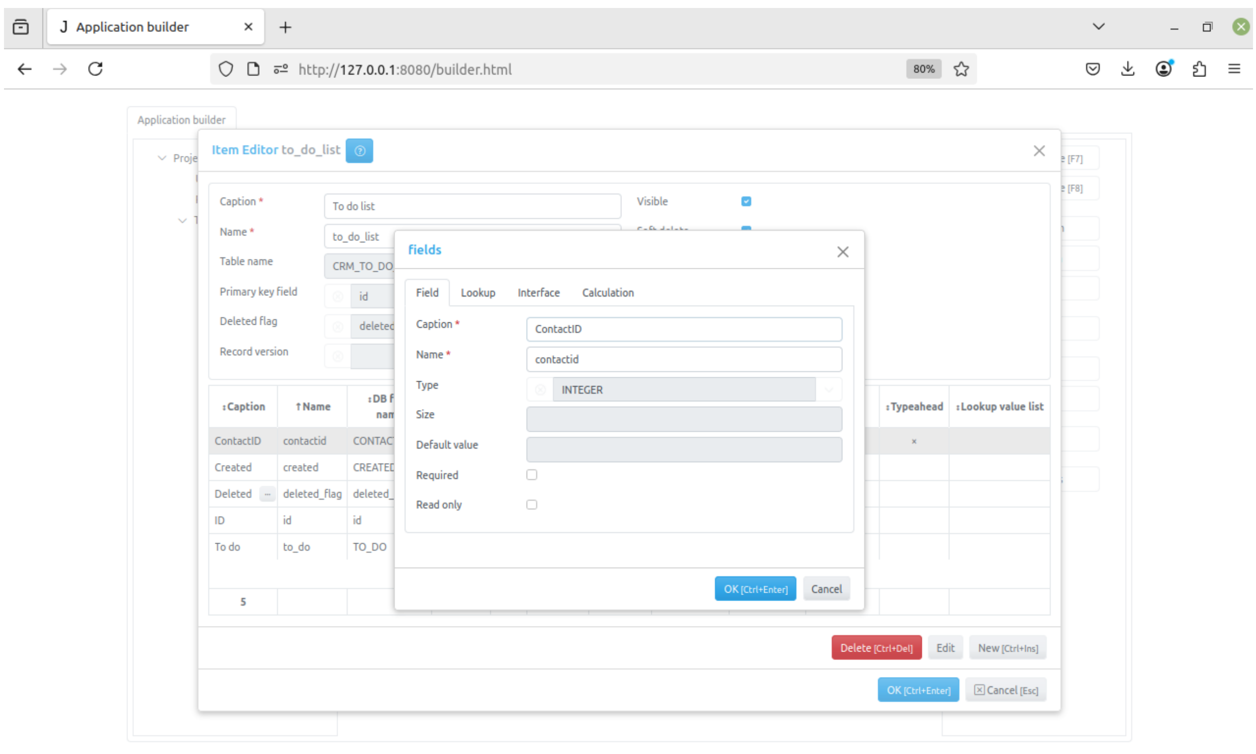


现在创建一个明细项。先选中刚刚新建的“明细表 (Details)”，再点击页面右下角的“新建 (New)”，然后在打开的实体项编辑器 对话框中，将明细项命名为“待办事项列表 (To do list)”，并按照前面的教程介绍的方式添加类型为 DATETIME 的“创建时间 (Created datetime)”和类型为 TEXT 的“待办事项 (To do)”两个字段：

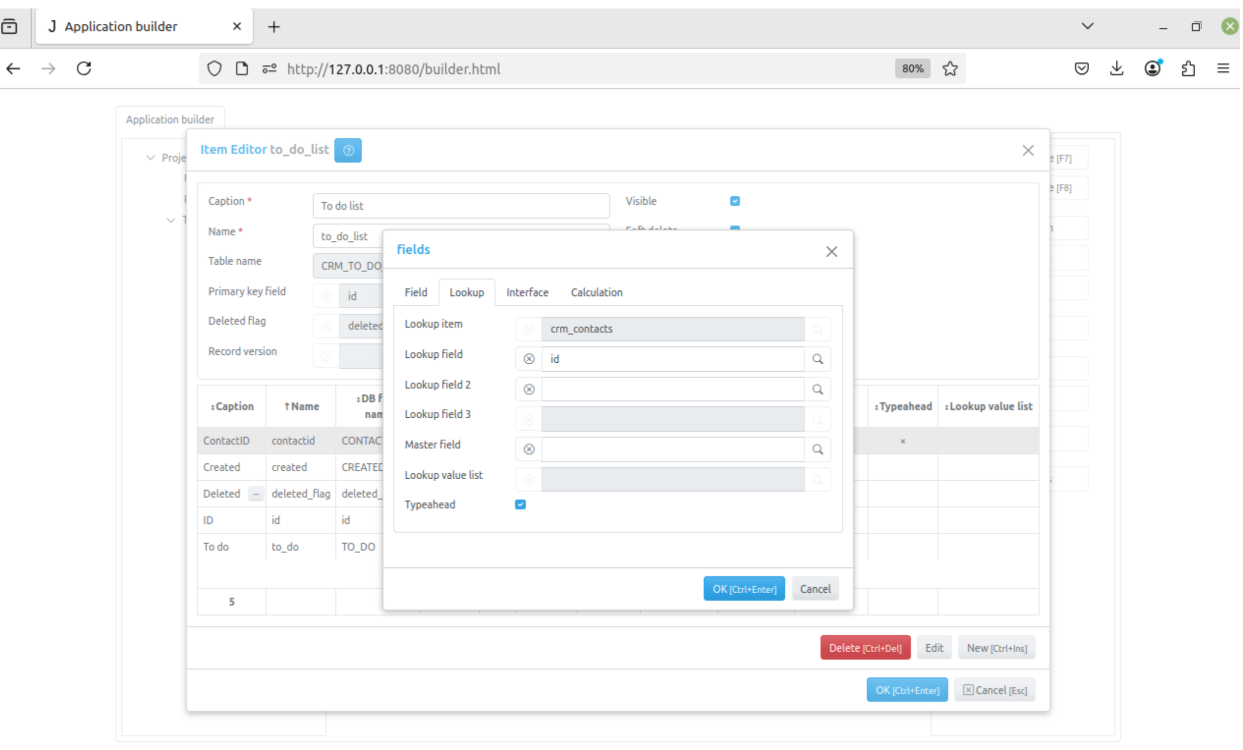


> **备注**
>
> 现在最重要的一步是创建将明细表与其主表“链接”起来的字段。这是与 Jam.py v5 的一个关键区别，在 v5 中明细表记录被视为作为未启用的功能。

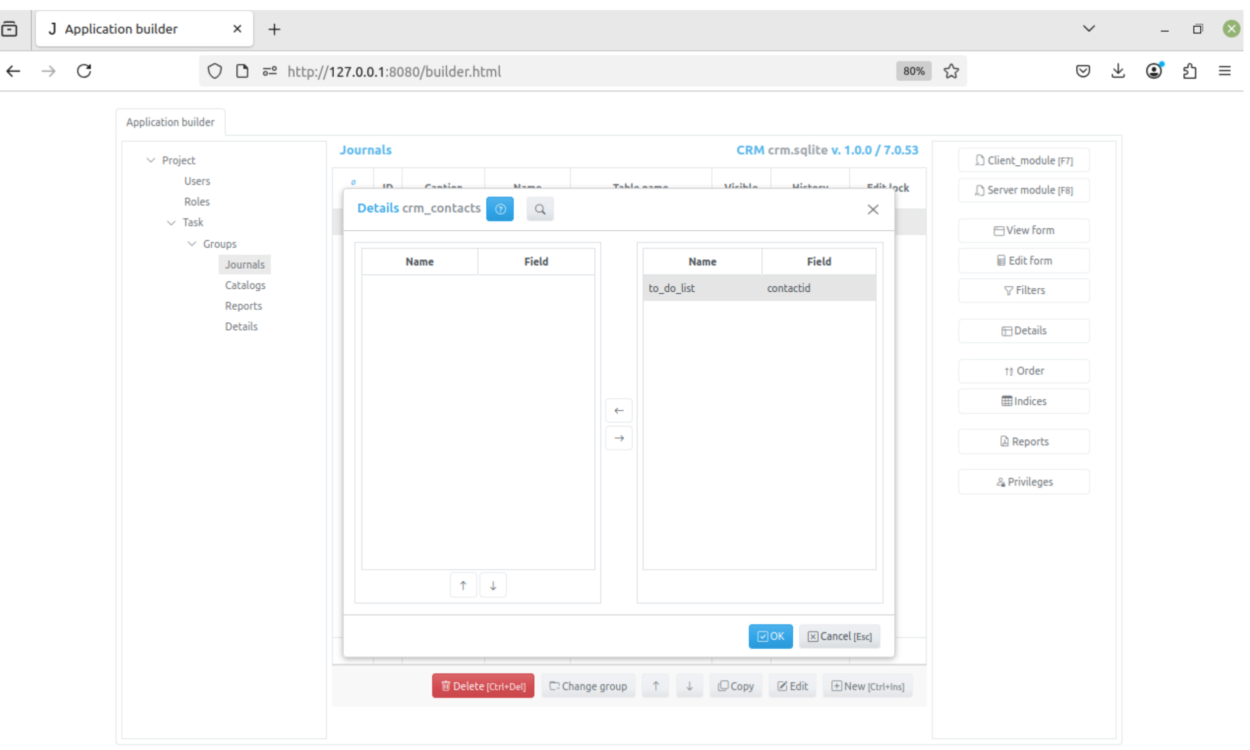
例如，创建字段“联系人 ID(ContactID)”：



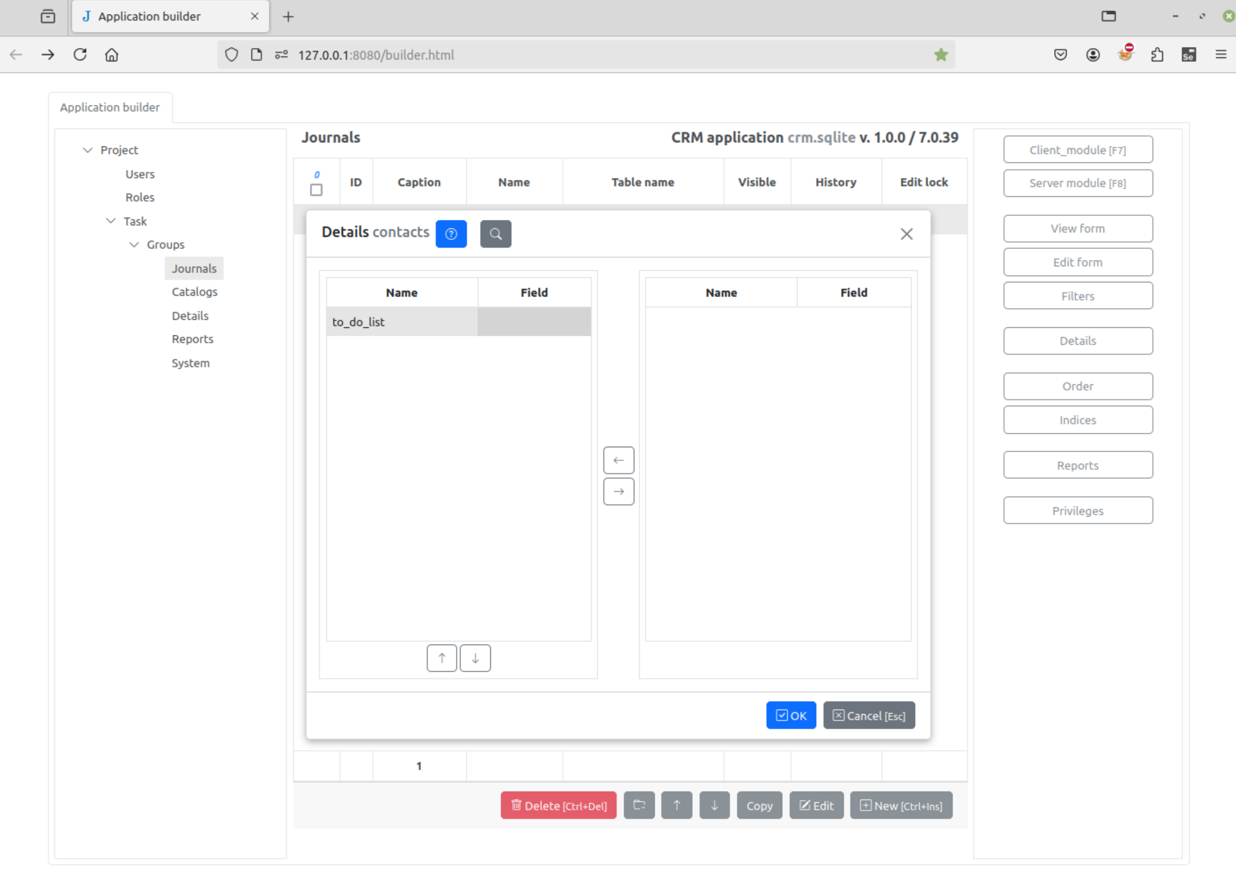
"ID" 字段是连到 "联系人 (Contacts)" 表的查找字段。



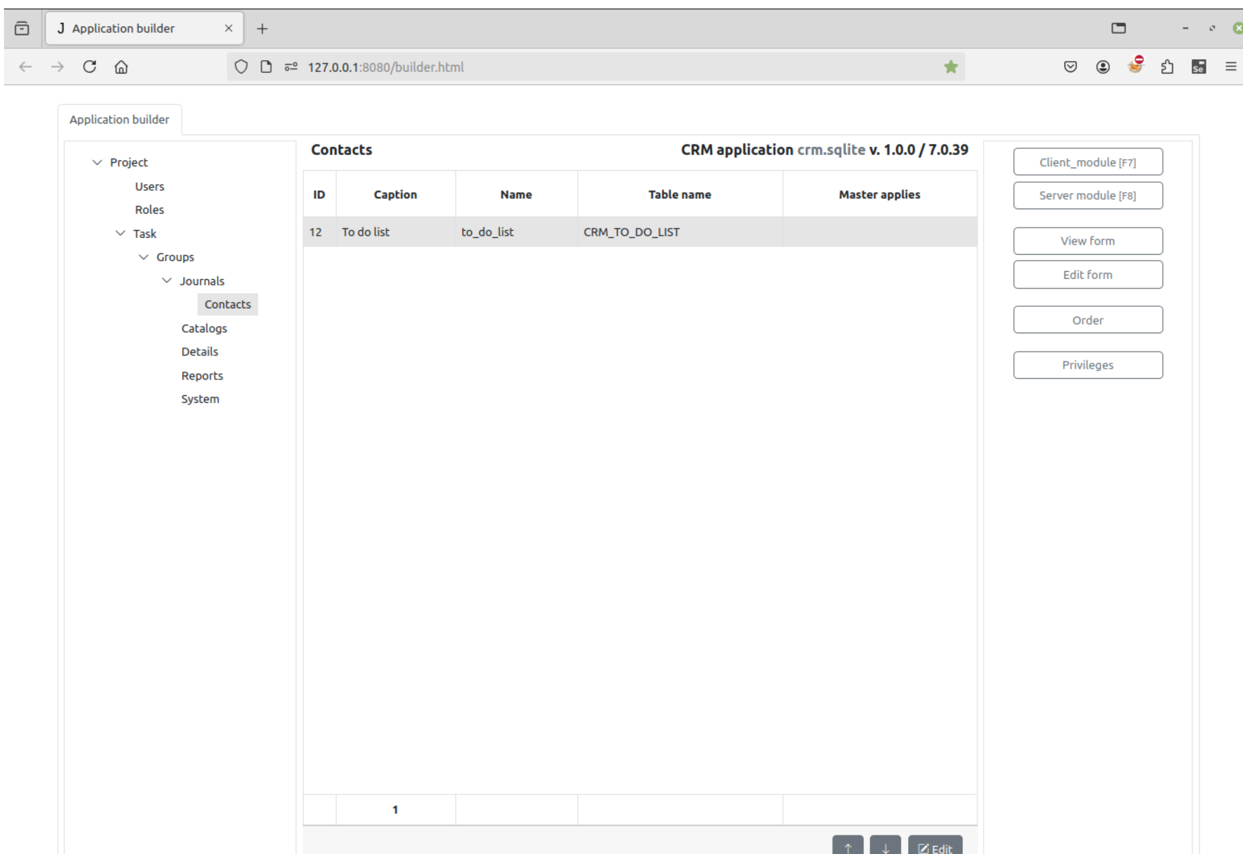
保存 "待办事项列表 (To do list)" 后，选择 "联系人 (Contacts)" 台账，然后点击右侧窗格中的 **明细表 (Details)** 按钮，打开明细表对话框 。



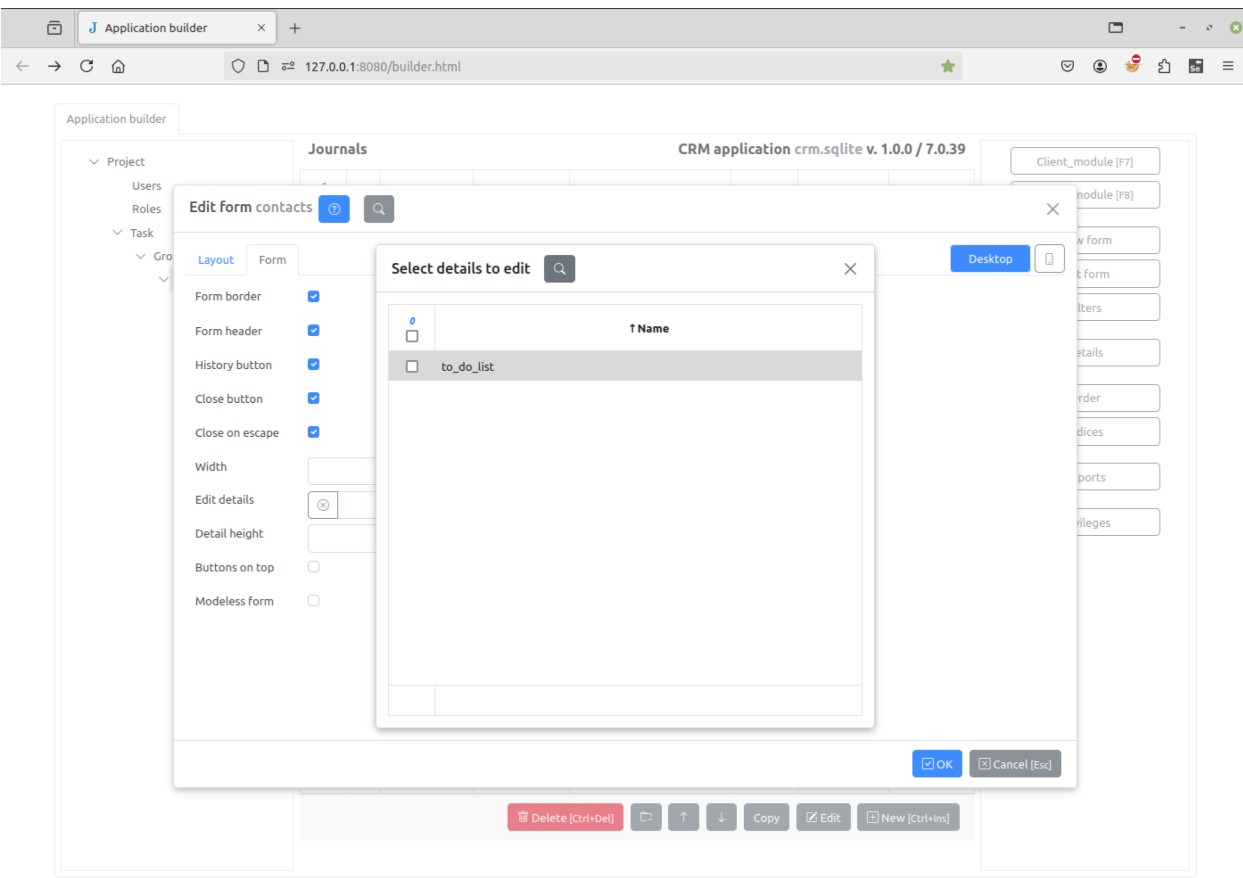
点击右箭头按钮将“待办事项列表 (To do list)”添加到“联系人 (Contacts)”的明细表中，然后点击 **确定**按钮保存更改。



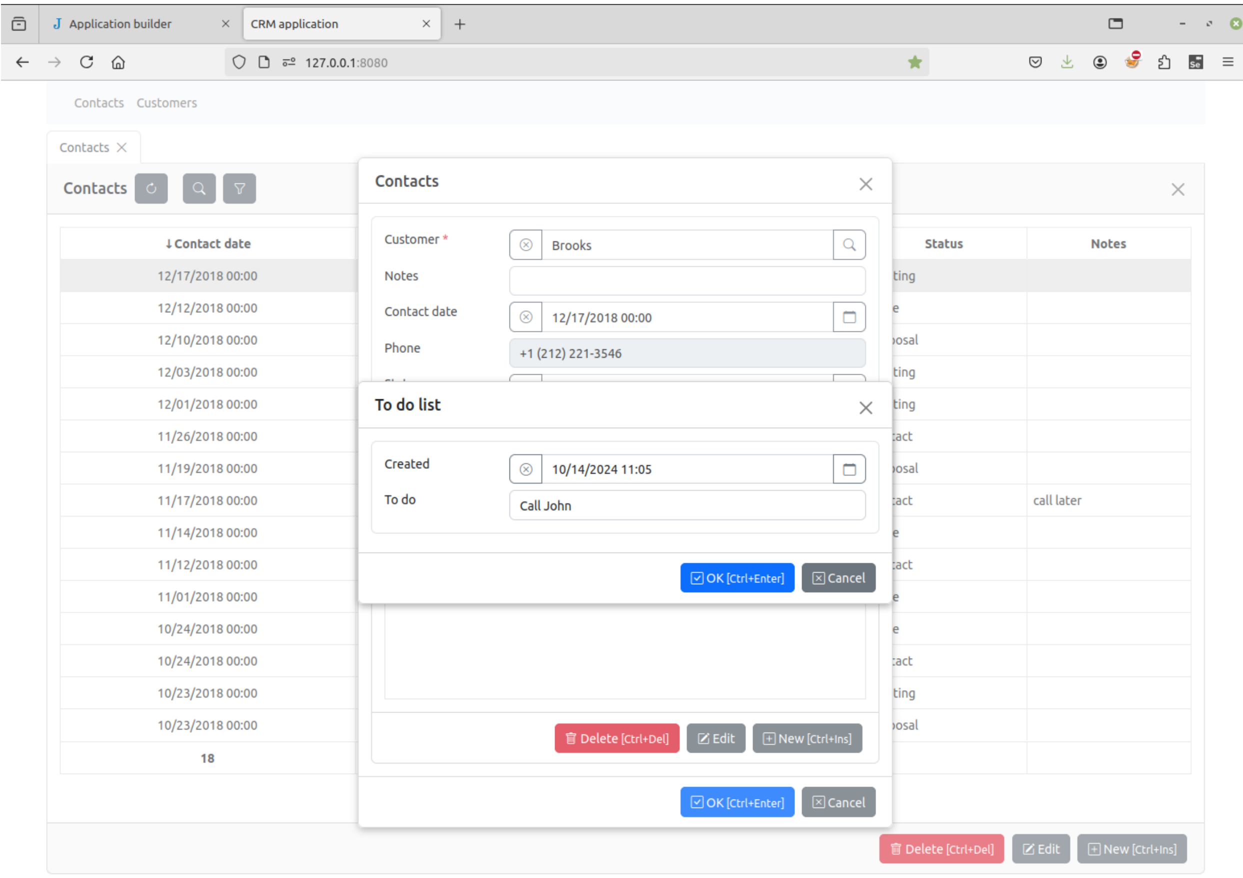
一个新的“待办事项列表 (To do list)”实体将被创建为“联系人 (Contacts)”台账的子项。



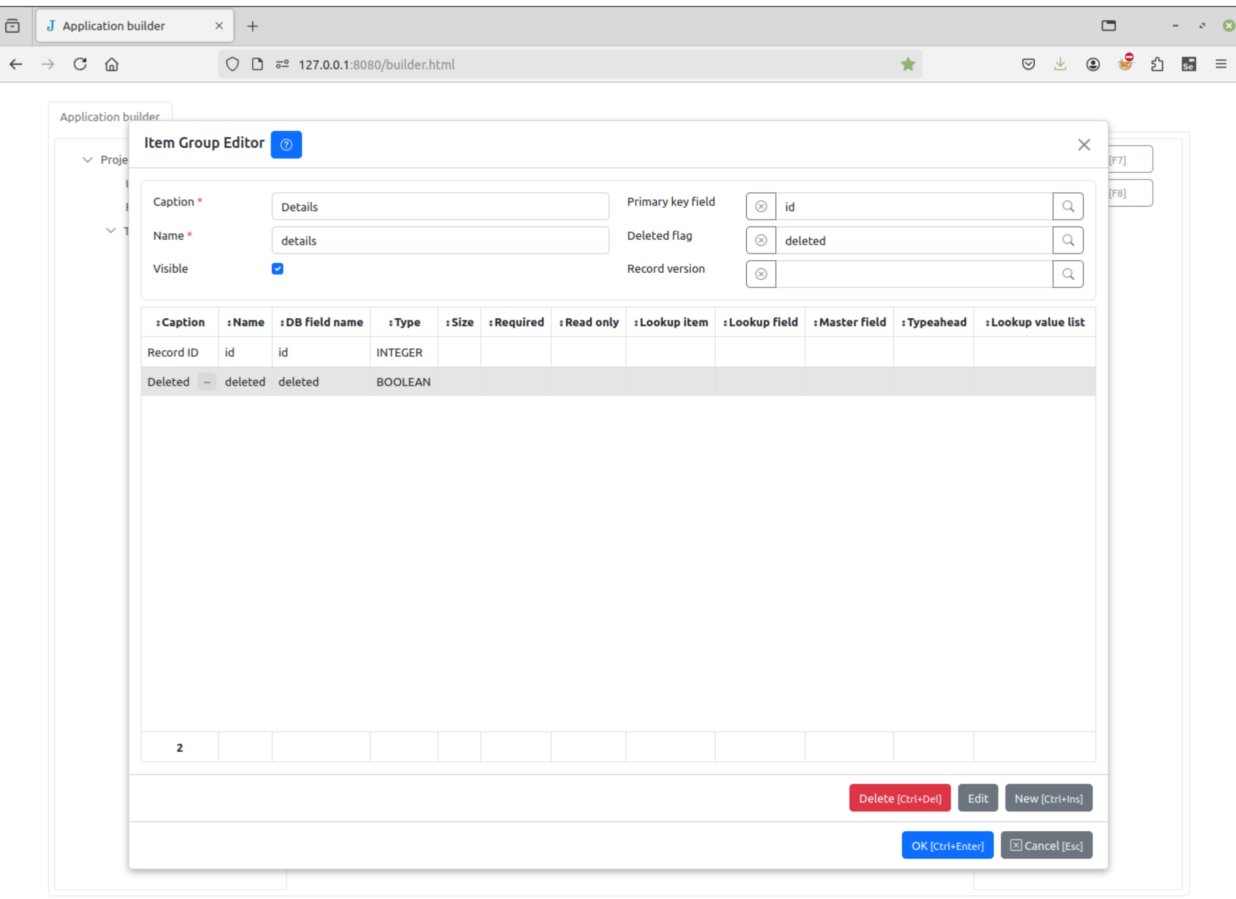
再次选择“联系人 (Contacts)”台账，点击页面右侧 **编辑表单 (Edit form)** 按钮打开编辑表单对话框。选择 **表单 (Form)** 选项卡，点击 **编辑明细 (Edit details)** 输入框右侧的按钮，然后勾选“待办事项列表 (To do list)”复选框。



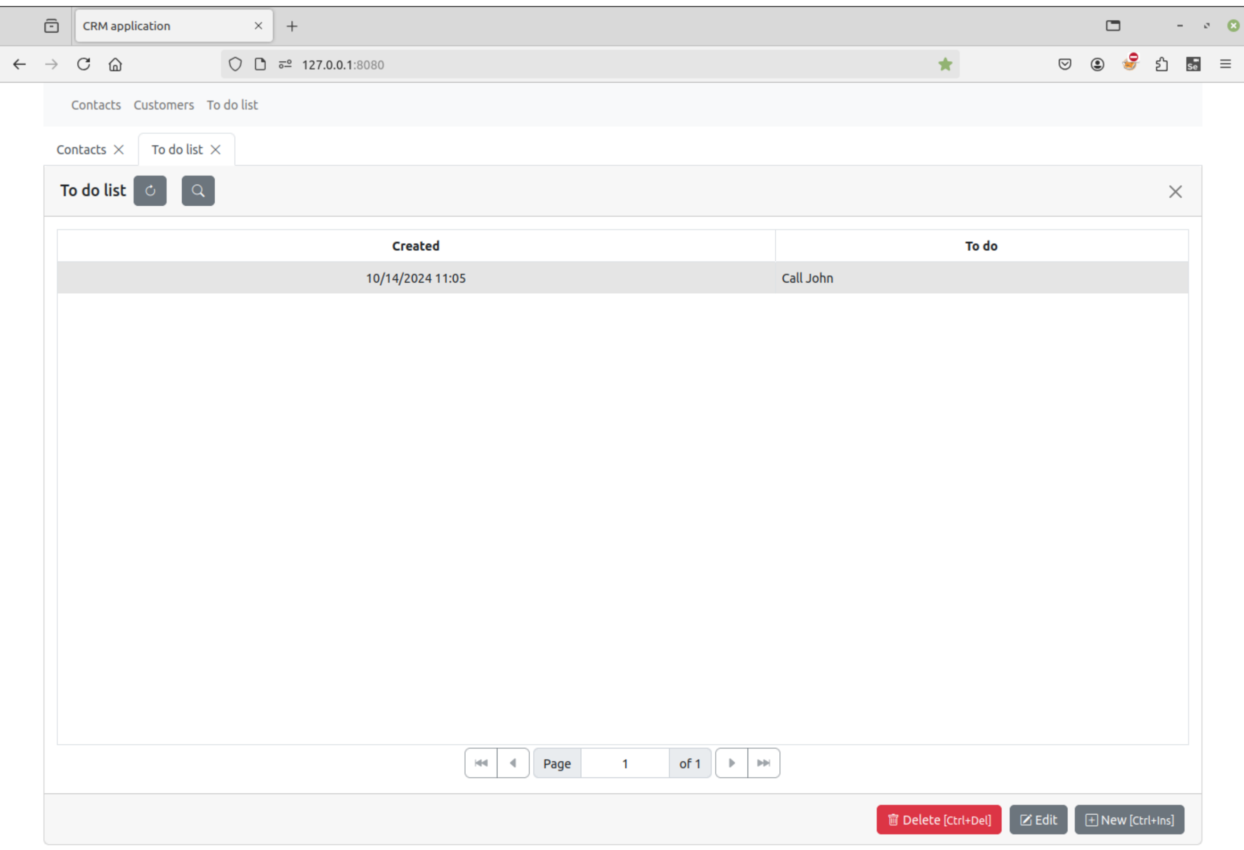
让我们更新客户端的项目页面并在“联系人 (Contacts)”台账中双击某个联系人。现在我们可以为该联系人的待办事项列表添加内容。



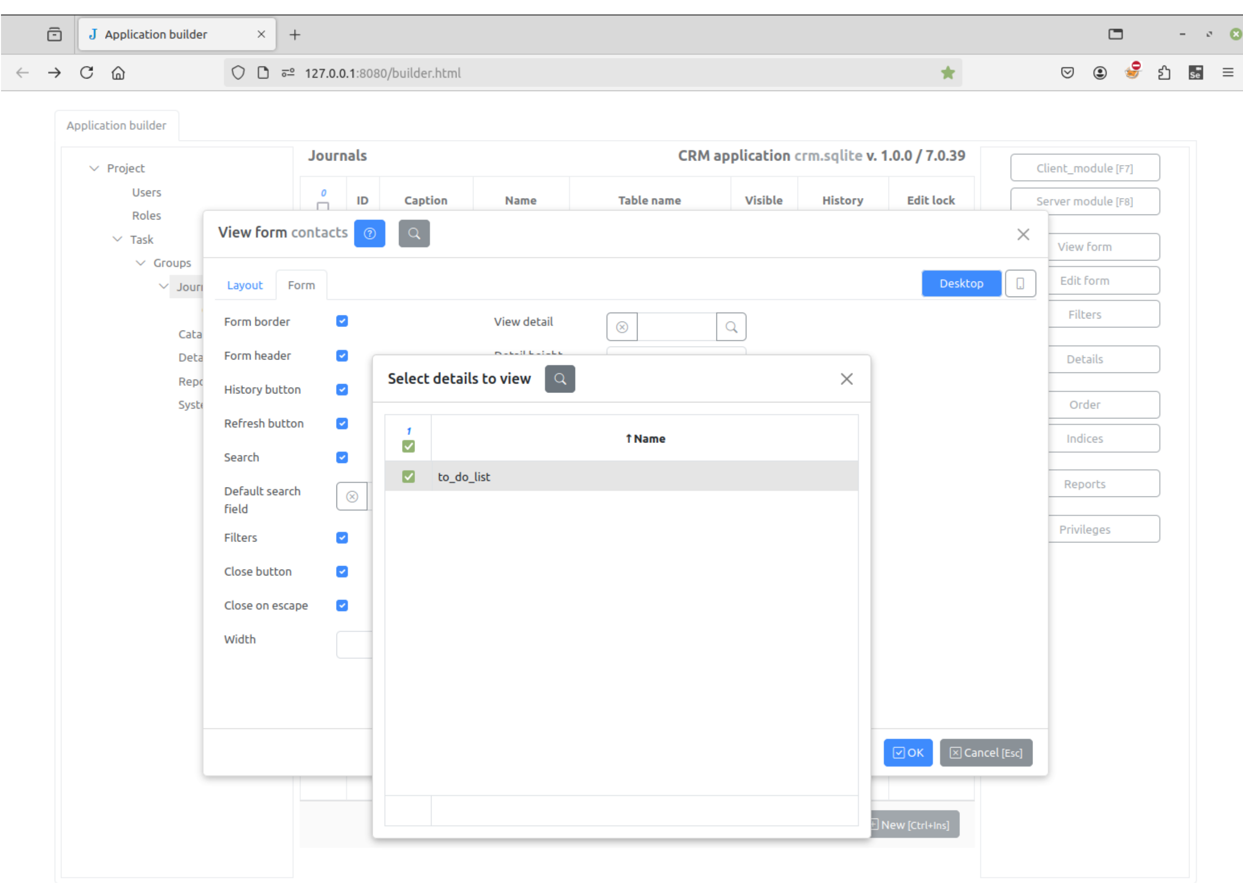
点击项目树中的 **分组 (Groups)** 节点，双击 **明细表 (Details)** 行，并将 "可见 (Visible)" 属性设置为 "是 (true)"。



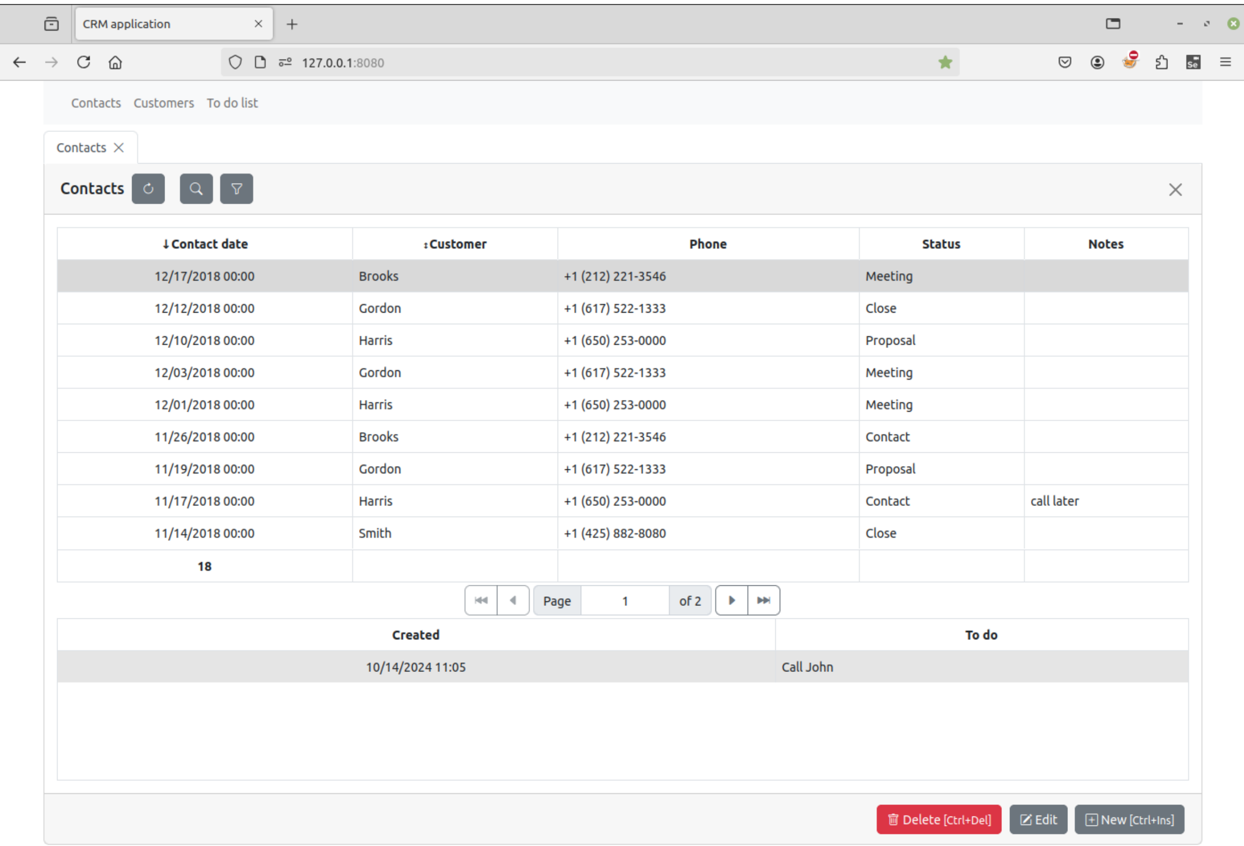
当我们刷新客户端项目页面时，我们将在主菜单中看到"待办事项列表 (To do list)"菜单项。点击它以查看所有联系人的待办事项列表。



再次选择“联系人 (Contacts)”台账，点击 **查看表单 (View form)** 按钮打开查看表单对话框。选择 **表单 (Form)** 选项卡，点击 **查看明细 (View detail)** 输入框右侧的按钮，然后勾选“待办事项列表 (To do list)”复选框。



在客户端项目页面中，当更改选中的联系人时，您将看到页面底部的待办事项列表也会相应更改。



2.7 部署

2.7.1 使用 Apache 和 mod_wsgi 部署 Jam.py

一旦你安装并激活了 `mod_wsgi`，请编辑你的 Apache 服务器的 httpd.conf 文件并添加以下内容。如果你使用的是 Apache 2.4 之前的版本，请将 **Require all granted** 替换为 **Allow from all**，并在其上方添加一行 **Order deny,allow**。

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

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

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

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

`WSGIScriptAlias` 行中的第一部分是你想用来服务应用程序的基础 URL 路径（`/` 表示根 URL），第二部分是系统中 "WSGI 文件" 的位置——见下文——通常位于你的项目包内（本例中为 `mysite`）。这告诉 Apache 使用该文件中定义的 WSGI 应用程序来处理给定 URL 下的任何请求。

`WSGIPythonPath` 行确保你的项目包可以在 Python 路径上被导入；换句话说，确保 `import mysite` 能够正常工作。

`<Directory>` 部分只是确保 Apache 能够访问你的 `wsgi.py` 文件。

接下来的几行确保以 `/static/` 开头的所有请求，都会被明确地当作静态文件来处理。

另请参阅

关于部署的更多信息，请参见如何部署

2.8 理解 admin.sqlite

Jam.py V7 中 admin.sqlite 数据库的思维导图描述了 Jam.py 数据库引擎的模式。其目的是快速查找所需信息和相关代码。

CHAPTER 3

Jam.py 编程

在这里，我们将解释 Jam.py 编程的基本概念。

3.1 任务树

框架的所有对象被组织为一个对象树。这些对象被称为实体项。

树中的所有实体项都有一个共同的祖先类 `AbstractItem`（客户端参考 / 服务端参考），并具有以下共同属性：

- `ID` - 标识实体项的 ID ，在框架内唯一
- `所有者 (owner)` - 实体项的直接父级和所有者
- `任务 (task)` - 任务树的根节点
- `子项 (items)` - 子实体项列表
- `类型 (item_type)` - 该实体项的类型
- `名称 (item_name)` - 该实体项的名称，在编程代码中用于访问该实体项
- `标题 (item_caption)` - 向用户显示的描述该实体项的标题

树的根节点是“任务 (task)”实体项。

“任务 (task)”实体项包含子实体项组（group items）。实体项组共有三种类型，其 `item_type` 属性对应以下取值（取值由组决定，无法手动修改）：

- “业务台账 (journals)” - 这个组里包含 `item_type` 为“item”的实体项，这些实体项可以有与之关联的数据库表。
- “主表目录 (catalogs)” - 这个组里也包含可以有与之关联的数据库表的实体项，但它们可用于为其他实体项创建明细表（参见明细表）。
- “报表 (reports)” - 这个组里包含报表 - `item_type` 为“report”的实体项，用于创建报表。

组的数量没有限制。建议为组命名时使用具备业务逻辑含义的名称，因为该名称会用于下拉菜单。

有与之关联的数据库表的实体项可以拥有明细表，这类明细表用于存储属于主表记录的子记录。

例如，演示项目 的任务树为

```
/demo/
    journals/
        invoices/
            invoice_table
        invoices_on_client/
            invoice_table
    catalogs/
       customers
       tracks/
            invoice_table
       albums
       artists
       genres
       mail
    details/
        invoice_table
    reports/
        invoice
        purchases_report
        customer_list
    analytics
        dashboard
    system/
        history
```

任务树的根节点是一个 实体项名称 (item_name) 为 **demo** 的任务。它包含五个组：**catalogs** 、**journals** 、**details** 、**reports** 、**analytics** 和 **system** 。**catalogs** 和 **journals** 组的 item_type 为 "items"。它们所对应的实体项是对应数据库表的包装器。有一个明细表实体项，其 实体项名称 (item_name) 为 **invoice_table** ，它也有自己的数据库表，而 **reports** 组中有三个报表。

发票 (invoices) 业务台账拥有 **invoice_table** 明细表，用于保存销售给客户的曲目列表。因此，有三个同名的 "invoice_table" 实体项："明细" 组中的明细表、Journal/Invoices 下的明细表以及 Catalogs/Tracks 下的明细表。

每个实体项都是其所有者的属性，所有实体项、数据表和报表也都是 "任务" 的属性（它们都有唯一的 实体项名称 (item_name))。

在客户端，"任务" 是一个全局对象。要在代码中的任何位置访问它，只需输入 task 。

在服务器端，任务不是全局的。Jam.py 是一个事件驱动的环境。每个事件都有一个触发该事件的实体项（或字段）参数。在实体项的服务器模块中定义的函数可以通过*server* 方法从客户端模块中执行，这些函数的第一个参数也是相应的实体项。

了解了一个实体项后，我们就可以访问任务树中的任何其他实体项。例如，要访问 **客户 (customers)** 主表项，我们可以这样写：

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

或者直接写：

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

项目的层次结构是该框架 DRY （don't repeat yourself，不要重复自己）原则的核心基础之一。

例如，实体项（item）的一些方法在执行时，会按顺序为任务（task）、实体组（group）和实体项（item）生成事件。

借助这种机制，我们可在任务的事件处理器中为所有实体项定义基础行为；该基础行为可在实体组的事件处理器中进行扩展；最终，若有需要，还可在实体项自身的事件处理器中对其进行定制化调整。更多详细信息，请参阅表单窗体事件

3.1.1 视频

任务树 (Task tree) 视频教程使用演示项目 演示了如何使用任务树。

3.2 工作流程

在 Jam.py 框架中，同时运行着两个任务：应用程序构建器和项目。每个任务都代表一个对象树——有应用程序构建器任务树和项目任务树。因此，在了解 Jam.py 工作流程之前，您需要先熟悉任务树 的概念。

Jam.py 的工作流程如下：

- 当运行 server.py 时，它会创建 WSGI 应用程序，而 WSGI 应用程序又会创建应用程序构建器任务树。
- 当服务器收到来自项目客户端的第一个请求时，项目任务树由应用程序构建器在服务器端创建。为此，应用程序构建器需要使用存储在项目根文件夹中的 admin.sqlite 数据库中的元数据。创建任务树后，服务器应用程序会触发*on_created* 事件，该事件可用于初始化服务器任务树。
- 当应用程序（应用程序构建器或项目）首次在客户端浏览器中运行时（加载 `builder.html` 或 `index.html` 后），会构建一个空的任务对象，该对象向服务器发送请求以对自身初始化。
- 如果项目的安全模式 参数已设置，框架在执行任何请求之前会检查用户是否已登录。如果未登录，则客户端应用程序会创建登录表单，用户输入其用户名和密码后，客户端任务会向服务器发送登录请求。
- 登录成功后，或者如果项目的安全模式 参数未设置，服务器会向客户端发送有关所请求任务的信息。客户端的任务根据此信息构建其树，为其对象分配事件处理程序，并执行*on_page_loaded* 事件处理程序。
- 在此事件处理程序中，开发人员应将 JQuery 事件处理程序函数附加到 `index.html` 文件中定义的 DOM 的 HTML 元素。在这些函数中，开发人员可以使用任务树 项目的方法来执行某些特定任务。这些方法在执行时，会触发不同的事件，在这些事件中可以调用其他方法，依此类推。请参阅客户端编程。
- 任务树中具有对应数据库表的项目，有可以在服务器数据库中读取和写入数据的方法。请参阅数据编程。
- 报告项目基于 LibreOffice 模板在服务器上生成报告。请参阅报告编程。
- 所有其方法生成对服务器的请求的项目，都以以下方式执行：它们调用任务的方法，向服务器发送任务的*ID* 、项目的*ID* 、请求类型及其参数。服务器收到请求后，根据传递的 ID，找到服务器上的任务（可以是项目任务或应用程序构建器任务）和项目，使用传递的参数执行相应方法，并将执行结果返回给客户端。这些服务器方法可能会触发它们自己的事件以覆盖默认的行为。请参阅服务器端编程。

3.2.1 视频

表单事件 (Form events) 和 客户端和服务端的交互 (Client-server interactions) 视频教程说明了 Jam.py 项目的工作流程。

3.3 使用模块

对于项目的任务树 中的每个实体项，开发人员都可以编写将在客户端或服务器上执行的代码。在应用程序构建器中，每个实体项的右上角都有两个按钮：**客户端模块 (Client module)** 和 **服务器模块 (Server module)** 。点击这些按钮将打开代码编辑器 。

每个实体项都有一组由应用程序触发的预定义事件。在实体项模块中定义的每个事件，都是是以 **on_** 前缀开头的函数。所有已发布的事件都列在代码编辑器 信息窗格的 Events（事件）选项卡中。

在代码编辑器 中，开发人员可以为这些事件编写代码，也可以定义一些其它函数。

例如，以下代码意味着在向演示项目的 Invoices（发票）业务台账中添加新记录后，invoicedate（发票日期）字段的值将等于当前日期。

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

备注

这些事件和函数将成为实体项的属性，并且可以在实体项代码中的任何地方访问。

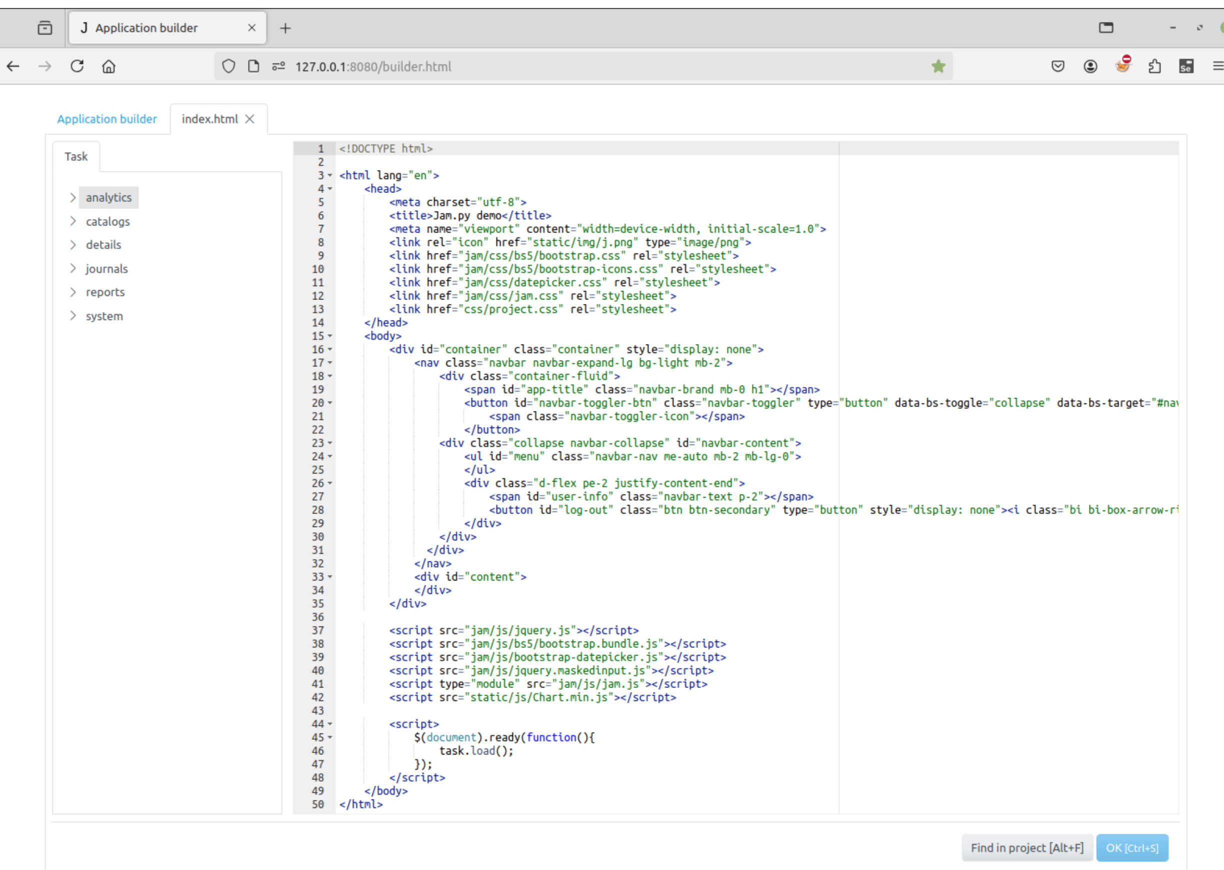
例如，在实体项客户端模块中定义的以下代码将为该实体项执行定义在 **客户 (Customers)** 实体项中的 on_edit_form_created 事件处理程序。

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

3.4 客户端编程

3.4.1 Index.html

当用户在 Web 浏览器中打开 Jam.py 应用程序时，浏览器首先加载 *index.html* 文件。该文件位于项目的根目录中，可通过点击任务 *(Task)* 树上的 `index.html [F10]` 来访问/修改。



该 html 文件包含客户端应用程序使用的 **css** 和 **js** 文件的链接。以 **jam** 开头的文件位于服务器上 Jam.py 软件包目录的 *jam* 文件夹中。

例如：

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

如果需要，可以在此处添加其他文件。例如，某些图表库。最好将它们放在项目 *static* 目录的 *js* 和 *css* 文件夹中。

例如：

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

关于表单模板，请参阅表单窗体 和表单窗体模板 以获取详细信息。

在文件末尾，有以下脚本：

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

在此脚本中，调用了在加载 *jam.js* 文件时创建的任务的*load* 方法，该方法从服务器加载有关任务树 的信息，并根据此信息构建项目的对象树、加载模块、为其实体分配事件处理程序，并触发*on_page_loaded* 事件。请参阅初始化应用程序

3.4.2 templates.html

当某些方法创建表单时，应用程序会在此文件中查找相应的 html 模板。

html 模板文件位于项目的根目录中，可以通过点击任务 *(Task)* 树上的 `templates.html [F9]` 来访问或修改。

您可以定义自己的表单模板来创建自定义表单。请参阅表单窗体模板 。

默认的 `template.html` 内容如下：

```
<div class="default-top-view">
    <div class="form-header">
        <button id="new-btn" class="btn btn-secondary" type="button">
            <i class="bi-plus-square"></i> New<small class="muted"> [Ctrl+Ins]</small>
        </button>
        <button id="edit-btn" class="btn btn-secondary" type="button">
            <i class="bi-pencil-square"></i> Edit
        </button>
        <div id="report-btn" class="btn-group dropdown">
            <a class="btn btn-secondary dropdown-toggle" data-toggle="dropdown" href="#">
                <i class="bi bi-printer"></i> Reports
                <span class="caret"></span>
            </a>
            <ul class="dropdown-menu bottom-up">
            </ul>
        </div>
        <button id="delete-btn" class="btn btn-danger" type="button">
            <i class="bi-trash"></i> Delete<small class="muted"> [Ctrl+Del]</small>
        </button>
    </div>
    <div class="card-body form-body">
        <div class="view-table"></div>
        <div class="view-detail"></div>
    </div>
</div>

<div class="default-view">
    <div class="form-body">
        <div class="view-table"></div>
        <div class="view-detail"></div>
    </div>
    <div class="form-footer">
        <div id="report-btn" class="btn dropdown dropup">
          <a class="btn btn-secondary dropdown-toggle" href="#" role="button" data-bs-toggle=
↪"dropdown" aria-expanded="false">
            Reports
          </a>
          <ul class="dropdown-menu">
          </ul>
        </div>
        <button id="delete-btn" class="btn btn-danger" type="button">
            <i class="bi-trash"></i> Delete<small class="muted"> [Ctrl+Del]</small>
        </button>
        <button id="edit-btn" class="btn btn-secondary" type="button">
            <i class="bi-pencil-square"></i> Edit
        </button>
        <button id="new-btn" class="btn btn-secondary" type="button">
            <i class="bi-plus-square"></i> New<small class="muted"> [Ctrl+Ins]</small>
        </button>
```

(续下页)

(接上页)

```
    </div>
</div>

<div class="default-top-edit">
    <div class="form-header">
        <button type="button" id="ok-btn" class="btn btn-primary">
            <i class="bi bi-check-square"></i> OK<small class="muted"> [Ctrl+Enter]</small>
        </button>
        <button type="button" id="cancel-btn" class="btn btn-secondary">
            <i class="bi bi-x-square"></i> Cancel
        </button>
    </div>
    <div class="form-body">
        <div class="edit-body"></div>
        <div class="edit-detail"></div>
    </div>
</div>

<div class="default-edit">
    <div class="form-body">
        <div class="edit-body"></div>
        <div class="edit-detail"></div>
    </div>
    <div class="form-footer">
        <button type="button" id="ok-btn" class="btn btn-primary">
            <i class="bi bi-check-square"></i> OK<small class="muted"> [Ctrl+Enter]</small>
        </button>
        <button type="button" id="cancel-btn" class="btn btn-secondary">
            <i class="bi bi-x-square"></i> Cancel
        </button>
    </div>
</div>

<div class="default-param">
    <div class="form-body">
        <div class="edit-body">
        </div>
    </div>
    <div class="form-footer">
        <select id='extension' class="form-select" style="width: 5rem">
            <option>pdf</option>
            <option>ods</option>
            <option>xls</option>
            <option>html</option>
        </select>
        <button type="button" id="ok-btn" class="btn btn-primary">
            <i class="bi bi-printer"></i> Print
        </button>
        <button type="button" id="cancel-btn" class="btn btn-secondary">
            <i class="bi bi-x-square"></i> Close
        </button>
    </div>
</div>

<div class="default-filter">
    <div class="form-body">
        <div class="edit-body">
```

(续下页)

(接上页)

```
        </div>
    </div>
    <div class="form-footer">
        <button type="button" id="ok-btn" class="btn btn-primary">
            <i class="bi bi-filter"></i> Apply
        </button>
        <button type="button" id="cancel-btn" class="btn btn-secondary">
            <i class="bi bi-x-square"></i> Close
        </button>
    </div>
</div>
```

3.4.3 初始化应用程序

on_page_loaded 事件是应用程序在客户端触发的第一个事件。

新项目使用*on_page_loaded* 事件处理程序来动态构建应用程序的主菜单，并使用 JQuery 为菜单项添加点击事件处理程序。

```
function on_page_loaded(task) {

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

    if (task.small_font) {
        $('html').css('font-size', '14px');
    }
    if (task.full_width) {
        $('#container').removeClass('container').addClass('container-fluid');
    }

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

    $('#container').show();

    task.create_menu($("#menu"), $("#content"), {
        splash_screen: '<h1 class="text-center">Jam.py Demo Application</h1>',
        view_first: true
    });
}
```

此事件处理程序使用 JQuery 从*index.html* 中选择元素，以设置它们的属性并分配事件。

```
<div id="container" class="container" style="display: none">
    <nav class="navbar navbar-expand-lg bg-light mb-2">
        <div class="container-fluid">
            <span id="app-title" class="navbar-brand mb-0 h1"></span>
            <button id="navbar-toggler-btn" class="navbar-toggler" type="button" data-bs-toggle=
```

(续下页)

(接上页)

```
↪"collapse" data-bs-target="#navbar-content" aria-controls="navbar-content" aria-expanded=
↪"false" aria-label="Toggle navigation">
                <span class="navbar-toggler-icon"></span>
            </button>
        <div class="collapse navbar-collapse" id="navbar-content">
            <ul id="menu" class="navbar-nav me-auto mb-2 mb-lg-0">
            </ul>
            <div class="d-flex pe-2 justify-content-end">
                <span id="user-info" class="navbar-text p-2"></span>
                <button id="log-out" class="btn btn-secondary" type="button" style="display:␣
↪none"><i class="bi bi-box-arrow-right"></i></button>
            </div>
        </div>
      </div>
    </nav>
    <div id="content">
    </div>
</div>
```

最后，调用任务的*create_menu* 方法来动态创建项目主菜单。

3.4.4 表单

框架的关键概念之一是 **表单 (form)** 。

当用户点击主菜单的菜单项时，会执行相应实体的*view* 方法，该方法会创建查看表单。

此查看表单可以有 新建 (New) 和 编辑 (Edit) 按钮，点击这些按钮将执行
insert_record

和*edit_record* 方法。
这些方法会创建实体编辑表单。

表单基于 HTML 表单窗体模板 ，这些模板决定了表单的布局。表单模板定义在项目的根目录中的*Index.html* 文件中。

应用程序已具有用于查看和编辑数据、指定过滤器和报告参数的默认模板。

例如，演示项目的所有编辑表单都使用以下 html 模板：

```
<div class="default-edit">
    <div class="form-body">
        <div class="edit-body"></div>
        <div class="edit-detail"></div>
    </div>
    <div class="form-footer">
        <button type="button" id="ok-btn" class="btn btn-primary">
            <i class="bi bi-check-square"></i> OK<small class="muted"> [Ctrl+Enter]</small>
        </button>
        <button type="button" id="cancel-btn" class="btn btn-secondary">
            <i class="bi bi-x-square"></i> Cancel
        </button>
    </div>
</div>
```

您可以定义自己的表单模板来创建自定义表单。请参阅表单窗体模板 。

当某些方法创建表单时，应用程序会查找相应的 html 模板。

如果指定了 container（一个 Jquery 对象）参数，该方法会先清空它，再将 html 模板添加到其中；否则，它会创建一个空的模态表单，并将模板添加到表单中。

此后，它将实体的 prefix_form 属性分配给模板，触发 on_prefix_form_created 事件，显示表单并触发 on_prefix_form_shown 事件，其中 prefix 是表单的类型（view（查看）、edit（编辑）、filter（过滤）、param（参数））。详情请参阅表单窗体事件。

以下是任务的 on_edit_form_created 事件处理程序的示例：

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

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

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

    return true;
}
```

在此示例中，使用了 JQuery 的 find 方法来查找表单上的元素。

首先，我们为 **确定 (OK)** 和 **取消 (Cancel)** 按钮分配 JQuery 的 click 事件，以便在用户点击按钮时执行

cancel_edit 和*apply_record* 方法。

这些方法分别取消或应用对记录所做的更改，并调用*close_edit_form* 方法来关闭表单。

然后，如果实体不是明细表，并且在其所有者的客户端模块中定义了事件处理程序 on_edit_form_created ，则执行此事件处理程序。

之后，如果实体在其客户端模块中定义了事件处理程序 on_edit_form_created ，则执行此事件处理程序。

在这些事件处理程序中，可以执行一些额外的操作。例如，您可以为编辑表单模板中包含的按钮或其他元素分配点击事件，更改*edit_options* ，使用*create_table* 方法创建表格，等等。

然后调用*create_inputs* 方法，在 class 为”edit-body” 的元素中创建输入控件。

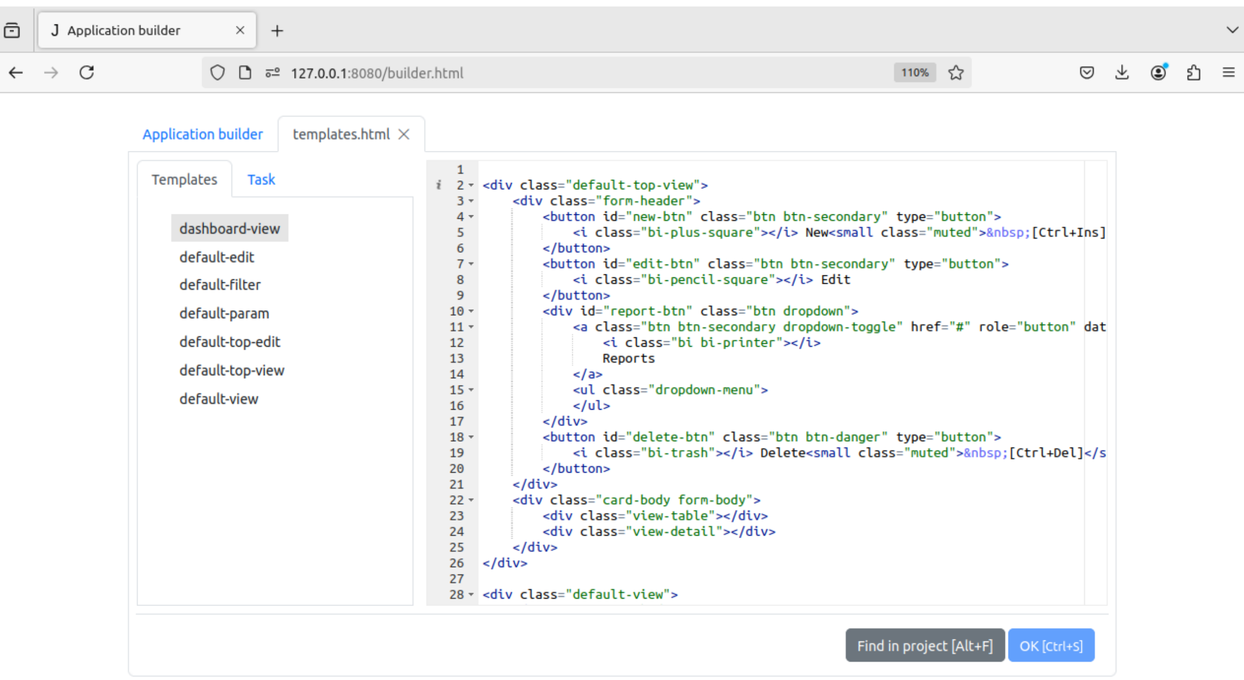
最后，调用*create_detail_views* 方法，在 class 为”edit-detail” 的元素中创建明细表。

> **备注**
>
> 即便表单模板中缺失部分元素，该方法也不会抛出异常。

close_prefix_form，其中 prefix 是表单的类型，用于关闭该类型的表单。但在表单关闭之前，会触发 on_prefix_form_close_query 和 on_prefix_form_closed 事件。表单关闭后，会从 DOM 中移除。

3.4.5 表单模板

项目的表单模板位于*templates.html* 文件中。该文件位于项目的根目录中，可通过点击任务 *(Task)* 树上的 templates [F9] 来访问/修改。



当执行*load* 方法时，文件被处理并作为 JQuery 对象存储在*templates* 的属性中。

要为实体项添加表单模板，您应该在 templates.html 中添加一个 class 为 `name-suffix` 的 div，其中 `name` 是实体项的*item_name* ，而 `suffix` 是表单类型：view（查看）、edit（编辑）、filter（过滤）、param（参数）。

例如：

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

是 **发票 (invoices)** 的编辑表单模板。

对于明细表，其名称前应加上其主表的名称，并用连字符分隔：

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

如果实体项没有表单模板，则将使用其所有者的表单模板（如果已定义）。

因此，模板

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

将用于创建 **业务台账 (Journals)** 组所拥有且没有自己的编辑表单模板的实体项的编辑表单。

如果通过这种方式搜索后，没有为实体项找到模板，则将使用 class 为 `default-suffix` 的模板来创建表单。

因此，模板

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

将用于为那些没有为其及其所有者定义模板的实体项创建编辑表单。

创建新项目时，index.html 已经包含此类模板。

下面是 index.html 文件中默认编辑表单模板的示例：

```
<div class="default-edit">
    <div class="form-body">
        <div class="edit-body"></div>
        <div class="edit-detail"></div>
    </div>
    <div class="form-footer">
        <button type="button" id="ok-btn" class="btn btn-primary">
            <i class="bi bi-check-square"></i> OK<small class="muted"> [Ctrl+Enter]</small>
        </button>
        <button type="button" id="cancel-btn" class="btn btn-secondary">
            <i class="bi bi-x-square"></i> Cancel
        </button>
    </div>
</div>
```

更多模板示例，请参阅表单窗体示例 部分。

3.4.6 表单事件

表单创建完成且 HTML 表单模板添加到 DOM 后，应用程序会在表单的生命周期内触发以下表单事件：

- `on_view_form_created` - 当表单已创建但尚未显示时触发此事件
- `on_view_form_shown` - 当表单已显示时触发此事件
- `on_view_form_close_query` - 当尝试关闭表单时触发此事件
- `on_view_form_closed` - 当表单已关闭时触发此事件
- `on_view_form_keydown` - 当表单发生 keydown 事件时触发此事件
- `on_view_form_keyup` - 当表单发生 keyup 事件时触发此事件

对于其他表单类型——edit（编辑）、filter（过滤）和 param（参数），将 "view" 替换为表单类型，例如，编辑表单的 `on_edit_form_created`。

我们将首先解释如何使用 `on_view_form_created` 事件。

当用户点击菜单项时，应用程序执行相应任务树实体项的*view* 方法，该方法使用其 HTML 表单模板创建表单，并首先触发任务的*on_view_form_created* 事件。

当您创建一个新项目时，任务客户端模块已包含一些代码，其中包括*on_view_form_created* 事件处理程序。每次创建查看表单时都会执行此事件处理程序，并定义查看表单的默认行为。

您可以打开任务客户端模块查看此事件处理程序。如果您需要更改项目中所有查看表单的默认行为，应在此处进行更改。

下面我们描述其主要执行步骤：

- 初始化*view_form* 和*table_options* ，这些属性在创建查看表单和表格时被某些方法使用。
- 根据用户权限，将默认按钮的 JQuery 事件处理程序分配给实体项的方法。在下面的示例中，初始化了删除按钮：

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

（续下页）

(接上页)

```
        item.delete_record();
    });
}
else {
    item.view_form.find("#delete-btn").prop("disabled", true);
}
```

- 执行实体组的*on_view_form_created* 事件处理程序，以及实体项的*on_view_form_created* 事件处理程序（如果已定义）：

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

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

- 创建一个表格来显示实体项数据，并为明细表创建表格（如果已通过调用 `create_view_tables` 方法指定）
- 执行*open* 方法，从服务器获取实体项数据集。
- 最后返回 true，以防止调用所有者组和实体项的 `on_view_form_created` 事件处理程序，因为它们已被调用（参见下面的 `_process_event` 方法）。

在我们初始化按钮之后、创建表格之前，我们调用了实体项本身的 `on_view_form_created` 事件处理程序。

例如，在演示应用程序的 tracks（曲目）实体项的客户端模块中，定义了以下*on_view_form_created* 事件处理程序。在其中，我们更改了*table_options* 的 "高度 (height)" 属性，创建了 "发票表格 (invoice_table)" 的副本，设置其属性，并调用其*create_table* 方法来创建一个表格以显示其数据。

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

该模块还有一个*on_after_scroll* 事件处理程序，当用户移动到另一首曲目时将执行此处理程序，并获取该曲目的销售数据。

这个示例解释了表单事件的使用原理。

事件的触发顺序取决于事件的类型。事件生成的顺序也取决于事件的类型。

关闭查询事件

当用户尝试关闭表单时，首先会为实体项触发 on_close_query 事件（如果已定义）。

如果事件处理程序返回 true，应用程序将关闭窗体；如果事件处理程序返回 false，应用程序将保持窗体打开；否则，将以相同方式触发实体组的 on_close_query 事件（若已定义），再触发任务的 on_close_query 事件（若已定义）。

例如，默认情况下，任务客户端模块中有一个*on_edit_form_close_query* 事件处理程序：

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

此代码检查记录是否已被修改，然后打开“是/否/取消”对话框。

如果我们想在不使用此对话框的情况下关闭表单，可以在实体项的客户端模块中定义以下事件处理程序：

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

Keydown、Keyup 事件

这些事件的触发方式与关闭查询事件相同，均从实体项开始；但如果事件处理程序返回 true，则不会执行组和任务的事件处理程序。

例如，默认情况下，任务客户端模块中有一个*on_edit_form_keyup* 事件处理程序：

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

此代码在用户按下 Ctrl+Enter 时将记录的更改保存到数据库表。

假设我们希望在用户按下 Enter 时保存更改。那么我们在实体项客户端模块中编写以下事件处理程序：

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

在这种情况下，当用户按下 Enter 时，任务的事件处理程序将不会被调用。

所有其他事件

对于其他事件，首先调用任务的事件处理程序；如果任务的事件处理程序没有返回 true，则执行组的事件处理程序；如果组的事件处理程序没有返回 true，再调用实体项的事件处理程序。

此机制由 *jam.js* 模块中 Item 类的 `_process_event` 方法实现。

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

3.4.7 表单选项

对于每种类型的表单，实体项都有一个控制模态表单行为的属性：

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

这是一个对象，具有以下属性，用于指定模态表单的参数：

- `width` - 模态表单的宽度，默认值为 560 px 。

- `title` - 模态表单的标题，默认值为*item_caption* 属性的值。
- `close_button` - 如果为 true，将在表单的右上角创建关闭按钮，默认值为 true。
- `close_caption` - 如果为 true 且 close_button 为 true，将在按钮附近显示 “Close - [Esc]”（关闭 - [Esc]）。
- `close_on_escape` - 如果为 true，按下 Escape 键将触发相应的 close_form 方法。
- `close_focusout` - 如果为 true，当表单失去焦点时，将调用相应的 close_form 方法。
- `template_class` - 如果指定，将在任务的*templates* 属性中搜索具有此 class 的 div，并在创建表单时用作表单 html 模板。
- `label_on_top` - 在顶部显示的标签。

edit_options 有一个 `fields` 属性，用于指定字段名称列表。如果其 `options` 参数的 `fields` 属性未指定，则*create_inputs* 方法将使用此列表，默认值是在应用程序构建器的编辑表单对话框 中设置的字段名称列表。

view_options 有一个 `fields` 属性，用于指定字段名称列表。如果其 `options` 参数的 `fields` 属性未指定，则*create_table* 方法将使用此列表，默认值是在应用程序构建器的视图表单对话框 中设置的字段名称列表。

在以下示例中创建的模态表单的宽度将为 700 px。

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

3.4.8 表单示例

目前，框架使用 Bootstrap 5 ，它具有一个简单易用的网格系统，这个网格系统最多使用 12 列，允许您创建任何类型的布局。它是响应式的，并包含许多组件，如下拉菜单、下拉按钮、按钮组、导航、导航栏、选项卡、面包屑、徽章、进度条等。

默认编辑表单模板

如果实体项及其所有者没有自己的编辑表单模板，则使用此模板创建编辑表单。

```
<div class="default-edit default-details-edit">
    <div class="form-body">
        <div class="edit-body"></div>
    </div>
    <div class="form-footer">
        <button type="button" id="ok-btn" class="btn btn-primary">
            <i class="bi bi-check-square"></i> OK<small class="muted"> [Ctrl+Enter]</small>
        </button>
        <button type="button" id="cancel-btn" class="btn btn-secondary">
            <i class="bi bi-x-square"></i> Cancel
        </button>
    </div>
</div>
```

下面的事件位于任务客户端模块中，当任何实体项的编辑表单刚刚创建时会触发该事件。

它使用*create_inputs* 方法在 class 为”edit-body” 的 div 中创建输入控件。但在此之前，它检查实体项的客户端模块中是否定义了 `init_inputs` 函数，该函数可用于指定方法的 options 参数。

然后，它为 OK 和 Cancel 按钮分配 jQuery 事件。

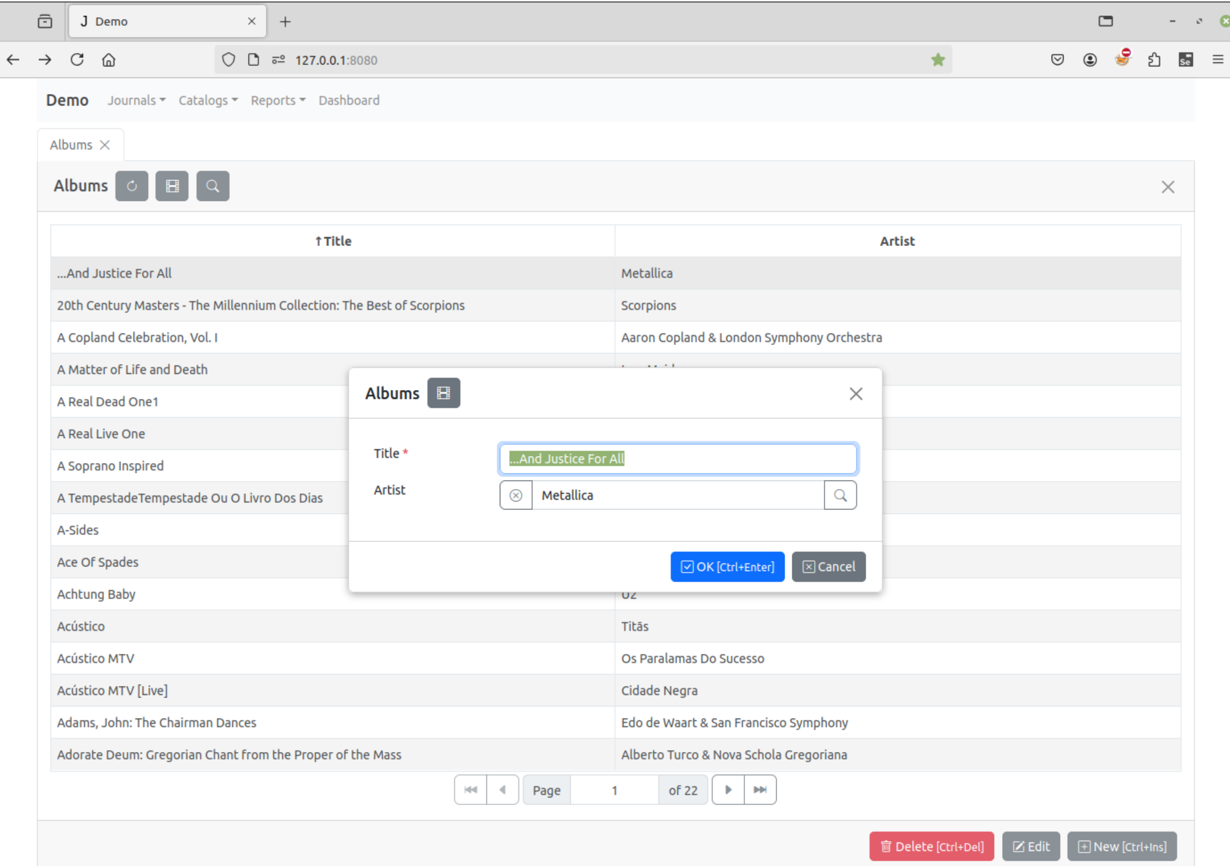
```
function on_edit_form_created(item) {
    var options = {
```

(续下页)

(接上页)

```
            col_count: 1
        };

    if (item.init_inputs) {
        item.init_inputs(item, options);
    }

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

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

主表目录中的 **专辑 (Albums)** 主表项的编辑表单如下所示：



> **备注**
>
> 如果没有具有相应 id 的按钮，上述代码不会生成异常。
>
> 如果您想在实体项的客户端模块的相应事件中，覆盖任务客户端模块中声明的按钮的 JQuery 事件，可以使用 jQuery 的 `off` 方法：

```
item.edit_form.find("#ok-btn")
    .off('click.task')
    .on('click', function() { some_other_function(item) });
```

如果表单中没有相应的容器，`create_inputs` 方法不会执行任何操作。

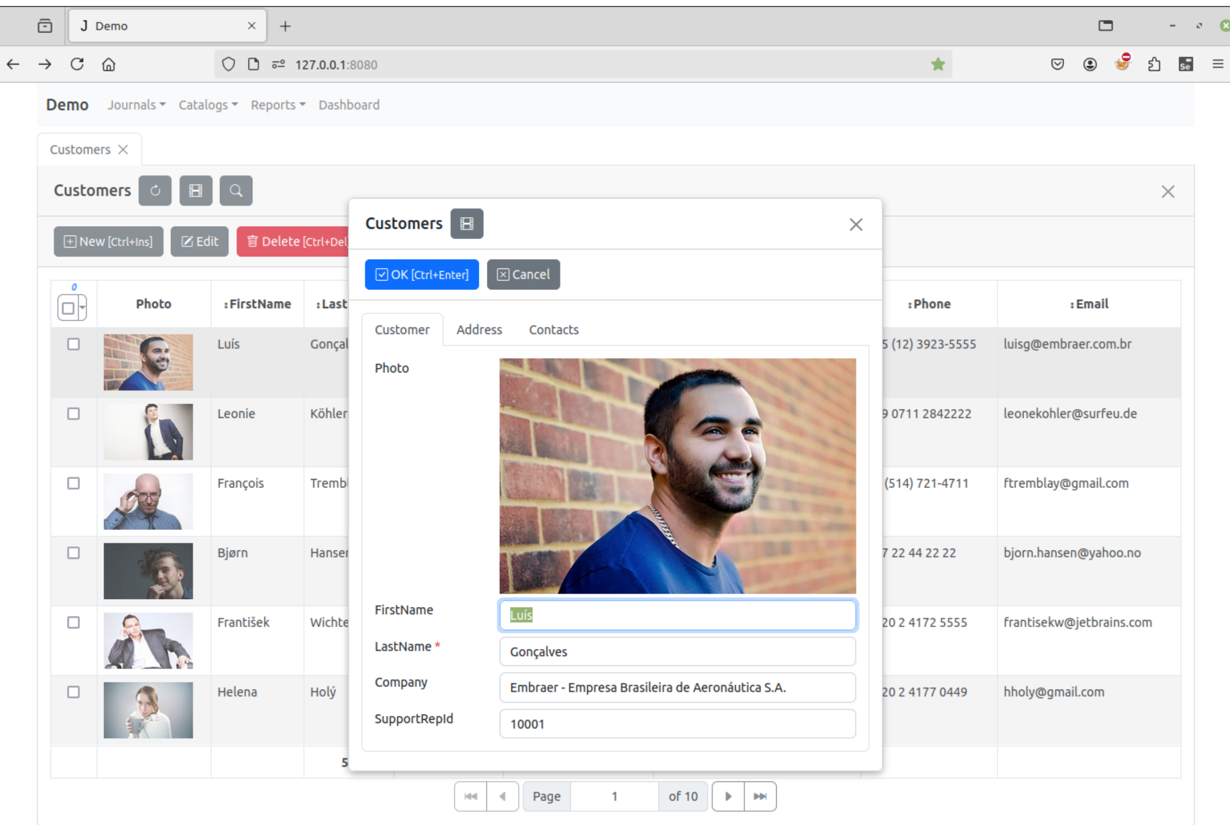
带选项卡的编辑表单模板

此示例使用 Bootstrap 3 的选项卡。由于 Bootstrap 5 不使用 href，需要对示例代码进行更新：

```
<div class="customers-edit">
    <div class="form-body">
        <ul class="nav nav-tabs" id="customer-tabs">
            <li class="active"><a href="#cust-name">Customer</a></li>
            <li><a href="#cust-address">Address</a></li>
            <li><a href="#cust-contact">Contact</a></li>
        </ul>
        <div class="tab-content">
            <div class="tab-pane active" id="cust-name">
            </div>
            <div class="tab-pane" id="cust-address">
            </div>
            <div class="tab-pane" id="cust-contact">
            </div>
        </div>
    </div>
    <div class="form-footer">
        <button type="button" id="ok-btn" class="btn btn-ary expanded-btn">
            <i class="icon-ok"></i> OK<small class="muted"> [Ctrl+Enter]</small>
        </button>
        <button type="button" id="cancel-btn" class="btn expanded-btn">
            <i class="icon-remove"></i> Cancel
        </button>
    </div>
</div>
```

以下事件处理程序声明在 **客户 (Customers)** 实体项的客户端模块中。它为与选项卡对应的窗格创建输入控件：它会为面板创建对应的输入控件，这些面板分别对应各个选项卡：

```
function on_edit_form_created(item) {
    item.edit_form.find('#customer-tabs a').click(function (e) {
      e.preventDefault();
      $(this).tab('show');
    });
    item.create_inputs(item.edit_form.find("#cust-name"),
        {fields: ['firstname', 'lastname', 'company', 'support_rep_id']}
    );
    item.create_inputs(item.edit_form.find("#cust-address"),
        {fields: ['country', 'state', 'address', 'postalcode']}
    );
    item.create_inputs(item.edit_form.find("#cust-contact"),
        {fields: ['phone', 'fax', 'email']}
    );
}
```



使用网格布局的编辑表单模板

此示例使用 Bootstrap 的网格系统：

```
<div class="tracks-edit">
    <div class="container-fluid form-body">
        <div class="row mb-3">
            <div id="edit-top" class="col-12 edit-border"></div>
        </div>
        <div class="row">
            <div id="edit-left" class="col-md-6 edit-border"></div>
            <div id="edit-right" class="col-md-6 edit-border"></div>
        </div>
    </div>
</div>
```

或：

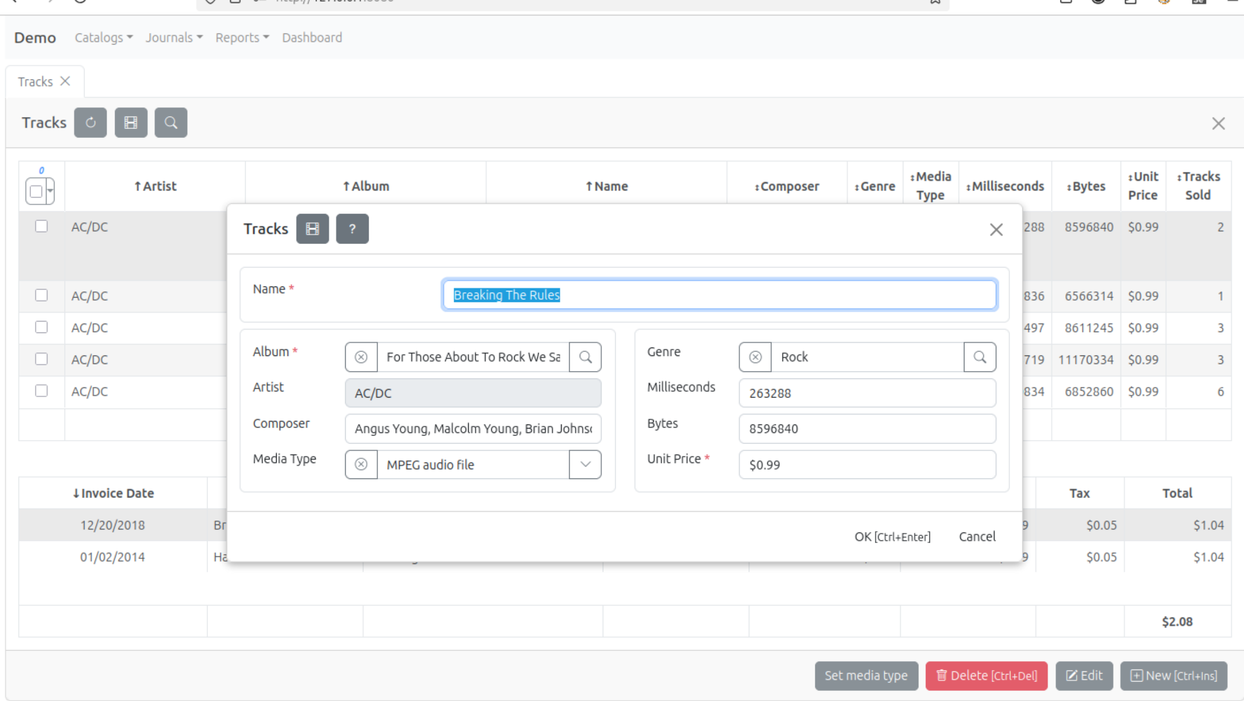
```
<div class="tracks-edit">
    <div class="form-body">
        <div class="row">
            <div id="edit-top edit-border"></div>
            <div class="col-lg-6">
                    <div id="edit-left" class="edit-border"></div>
            </div>
            <div class="col-lg-6">
                    <div id="edit-right" class="edit-border"></div>
```

(续下页)

(接上页)

```
            </div>
        </div>
    </div>
</div>
```

```
function on_edit_form_created(item) {
    item.edit_options.width = 900;

    item.create_inputs(item.edit_form.find("#edit-top"), {
        fields: ['name']
    });

    item.create_inputs(item.edit_form.find("#edit-left"), {
        fields: ['album', 'artist', 'composer', 'media_type']
    });

    item.create_inputs(item.edit_form.find("#edit-right"), {
        fields: ['genre', 'milliseconds', 'bytes', 'unitprice']
    });
}
```



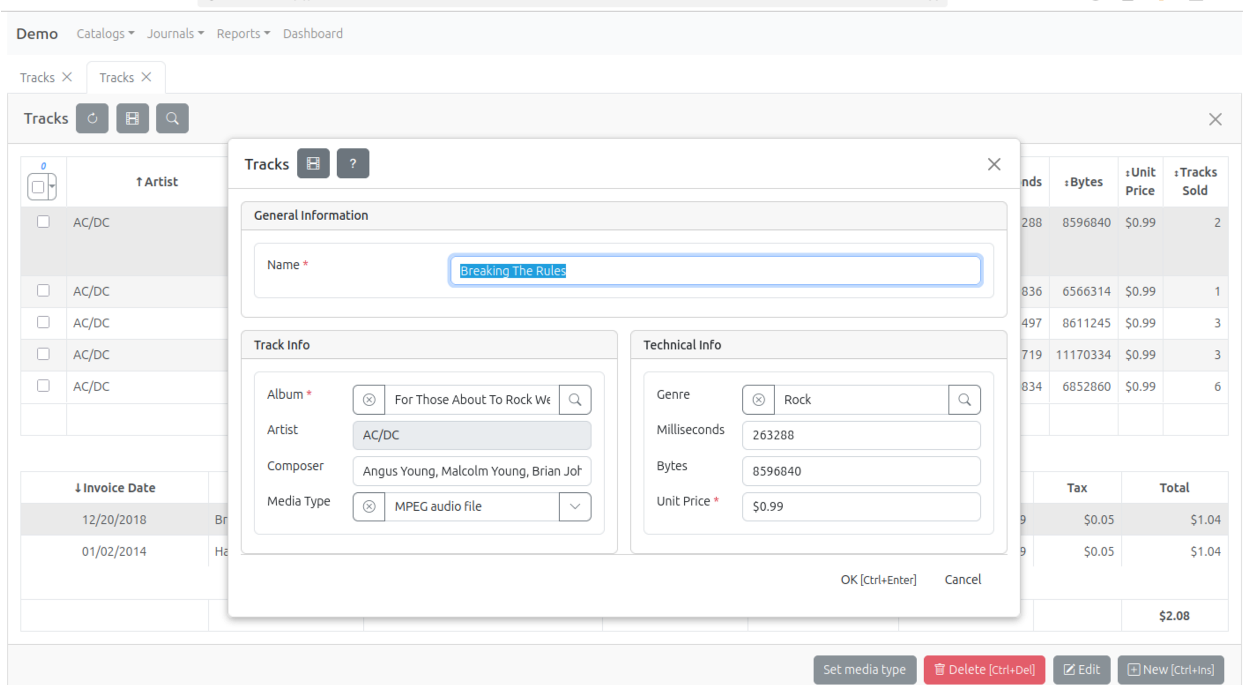
每个部分周围都有标题，例如 “曲目信息 (Track Info)” / “技术信息 (Technical Info)”：

```
<div class="tracks-edit">
    <div class="card mb-3">
        <div class="card-header">
            <h6 class="mb-0">General Information</h6>
        </div>
        <div class="card-body">
            <div id="edit-top"></div>
        </div>
```

(续下页)

(接上页)

```
    </div>
    <div class="row g-3">
        <div class="col-md-6">
            <div class="card h-100">
                <div class="card-header">
                    <h6 class="mb-0">Track Info</h6>
                </div>
                <div class="card-body" id="edit-left"></div>
            </div>
        </div>
        <div class="col-md-6">
            <div class="card h-100">
                <div class="card-header">
                    <h6 class="mb-0">Technical Info</h6>
                </div>
                <div class="card-body" id="edit-right"></div>
            </div>
        </div>
    </div>
</div>
```



主表目录查看表单模板

在这个例子中，有一个 class 为"form-header" 的 div。

id 为"form-title" 的元素在任务的*on_view_form_created* 方法中用于显示实体项的标题，并为其分配一个 JQuery onclick 事件来执行 view 方法以重新创建查看表单。

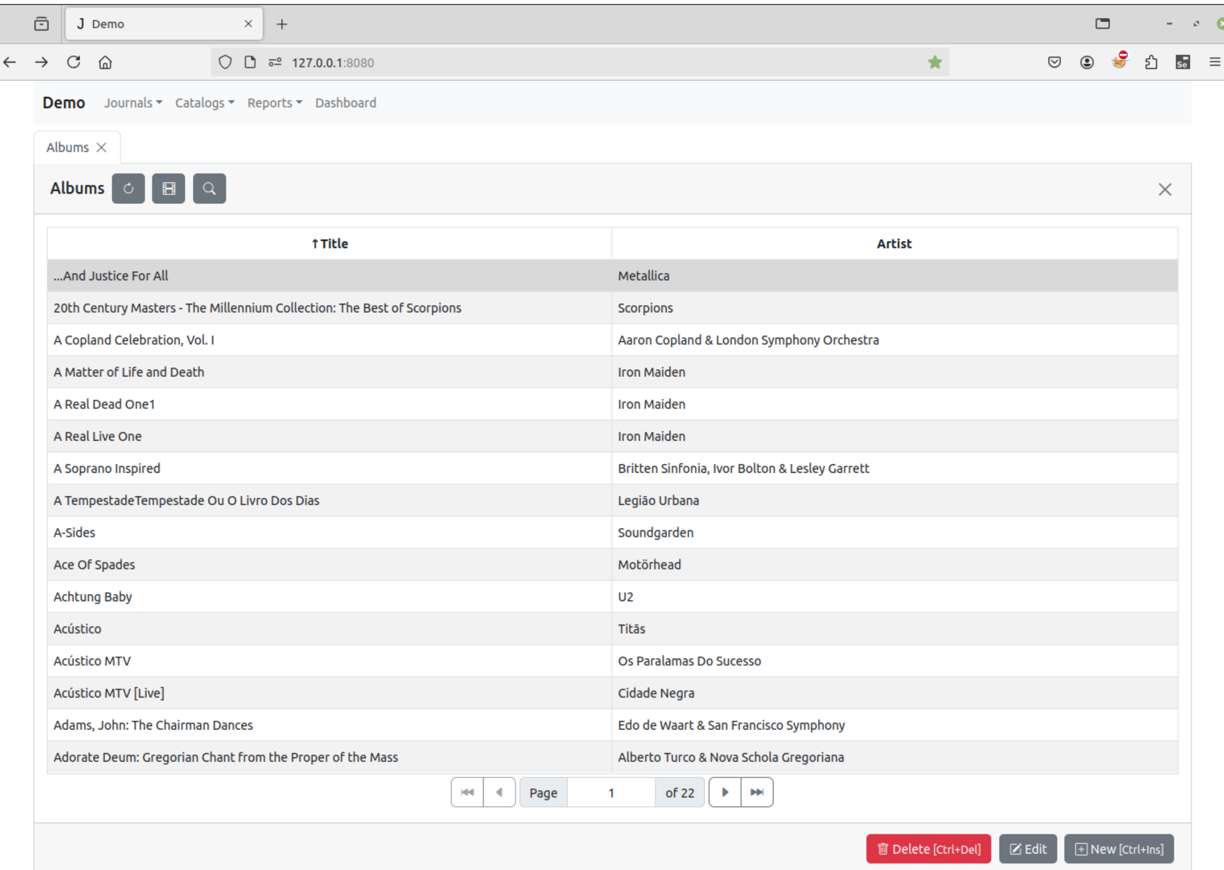
id 为"selected-div" 和"search-form" 的元素在主表目录组的*on_view_form_created* 中用于在点击右侧按钮选择值时显示查找字段的当前值，并相应地实现目录的搜索功能。

class 为"view-table" 的 div 在任务的*on_view_form_created* 事件处理程序中用于通过*create_table* 方法创建一个表格来显示实体项数据:

```
if (item.view_form.find(".view-table").length) {
    if (item.init_table) {
        item.init_table(item, table_options);
    }
    item.create_table(item.view_form.find(".view-table"), table_options);
    item.open(true);
}
```

id 为"report-btn" 的 div 在任务的*on_view_form_created* 事件处理程序中用于用实体项表白对话框 中定义的报告填充下拉按钮菜单项（如果它们存在)。

```
<div class="catalogs-view">
    <div class="form-body">
        <div class="form-header">
            <h4 id="form-title" class="header-text"><a href="#"></a></h4>
            <h5 id="selected-div" class="header-text" style="display: none">
                <a id="selected-value" href="#"></a>
            </h5>
            <form id="search-form" class="form-inline pull-right">
                <label  class="control-label" for="search-input">Search by
                    <span class="label" id="search-fieldname"></span>
                </label>
                <input id="search-input" type="text" class="input-medium search-query"␣
↪autocomplete="off">
                <a id="search-field-info" href="#" tabindex="-1">
                    <span class="badge">?</span>
                </a>
            </form>
        </div>
        <div class="view-table">
        </div>
    </div>
    <div class="form-footer">
        <button id="delete-btn" class="btn expanded-btn pull-left" type="button">
            <i class="icon-trash"></i> Delete<small class="muted"> [Ctrl+Del]</small>
        </button>
        <div id="report-btn" class="btn-group dropup">
            <a class="btn expanded-btn dropdown-toggle" data-toggle="dropdown" href="#">
                <i class="icon-print"></i> Reports
                <span class="caret"></span>
            </a>
            <ul class="dropdown-menu bottom-up">
            </ul>
        </div>
        <button id="edit-btn" class="btn expanded-btn" type="button">
            <i class="icon-edit"></i> Edit
        </button>
        <button id="new-btn" class="btn expanded-btn" type="button">
            <i class="icon-plus"></i> New<small class="muted"> [Ctrl+Ins]</small>
        </button>
    </div>
</div>
```



按钮在顶部的查看表单模板

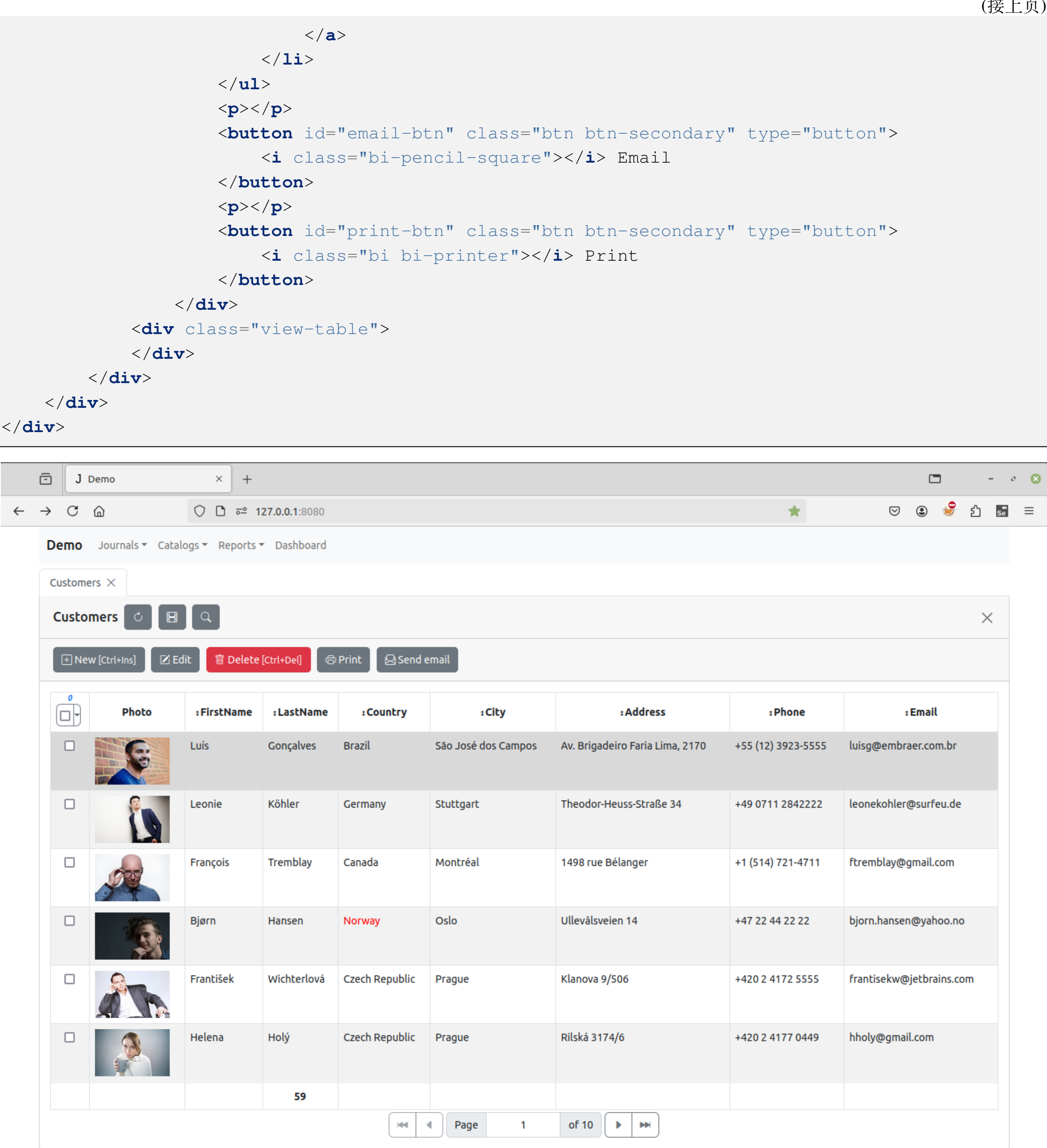
在此示例中，移除了表单页脚 div，并将按钮放置在表单标题 div 中。创建了 **操作 (Actions)** 下拉按钮。代码与上一个示例相同。

```
<div class="customers-view">
    <div class="form-body">
        <div class="form-header">
                <div id="action-btn" class="btn dropdown">
                    <a class="btn btn-secondary dropdown-toggle" href="#" role="button" data-bs-
↪toggle="dropdown">
                        <i class="bi bi-display"></i>
                        Action
                    </a>
                    <ul class="dropdown-menu">
                        <li id="new-btn">
                            <a class="dropdown-item" href="#">
                                <i class="bi-plus-square"></i>
                                New<small class="muted"> [Ctrl+Ins]</small>
                            </a>
                        </li>
                        <li id="edit-btn"><a class="dropdown-item" href="#"><i class="bi-pencil-
↪square"></i> Edit</a></li>
                        <li id="delete-btn">
                            <a class="dropdown-item" href="#">
                            <i class="bi-trash"></i>
                            Delete<small class="muted"> [Ctrl+Del]</small>
```

(续下页)

(接上页)

```
                            </a>
                        </li>
                    </ul>
                    <p></p>
                    <button id="email-btn" class="btn btn-secondary" type="button">
                        <i class="bi-pencil-square"></i> Email
                    </button>
                    <p></p>
                    <button id="print-btn" class="btn btn-secondary" type="button">
                        <i class="bi bi-printer"></i> Print
                    </button>
                </div>
            <div class="view-table">
            </div>
        </div>
    </div>
</div>
```



带明细表的查看表单模板

在此示例中，移除了 class 为"view-table" 的 div，并在**发票 (Invoices)** 台账的客户端模块中声明的*on_view_form_created* 事件处理程序中为主表和明细表实体项创建了两个 div："view-master" 和"view-detail" 表格：

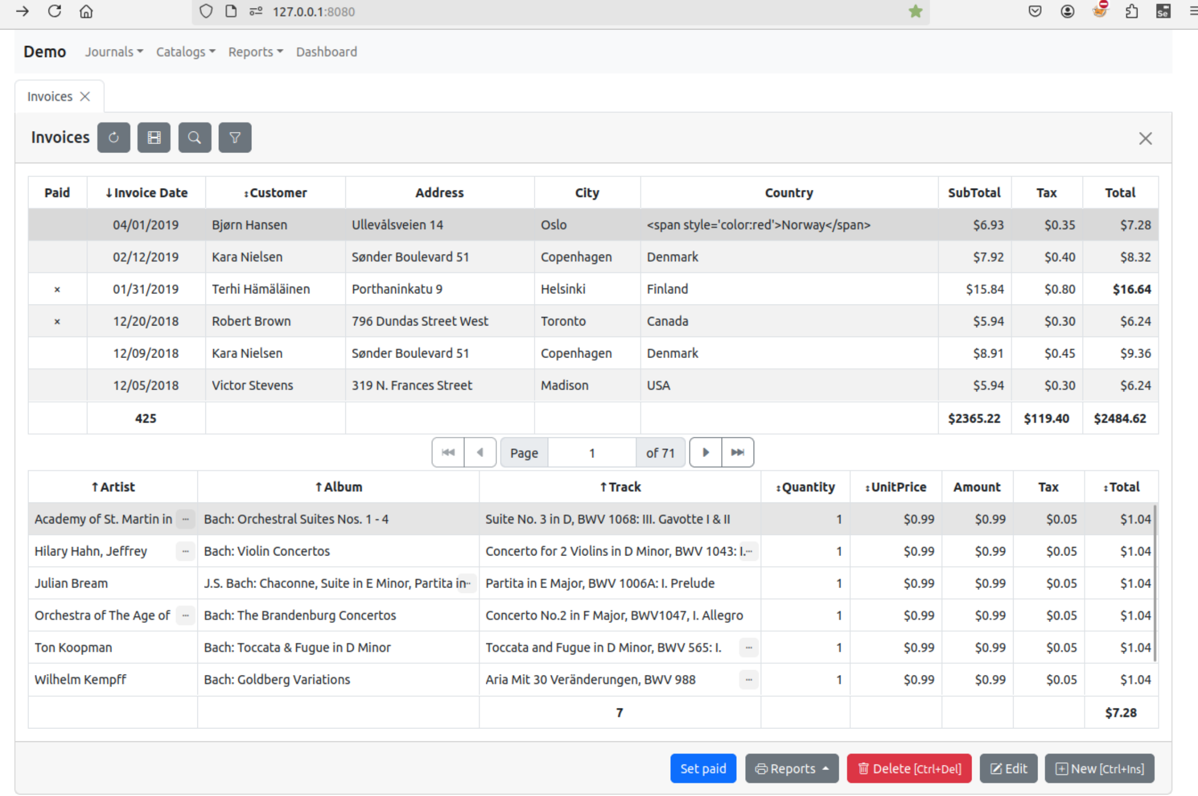
```
function on_view_form_created(item) {
    var height = $(window).height() - $('body').height() - 200 - 10;

    if (height < 200) {
        height = 200;
    }

    item.filters.invoicedate1.value = new Date(new Date().setYear(new Date().getFullYear() -␣
↪1));

    item.create_table(item.view_form.find(".view-master"), {
        height: height,
        sortable: true,
        show_footer: true,
        row_callback: function(row, it) {
            var font_weight = 'normal';
            if (it.total.value > 10) {
                font_weight = 'bold';
            }
            row.find('td.total').css('font-weight', font_weight);
        }
    });

    item.invoice_table.create_table(item.view_form.find(".view-detail"), {
            height: 200 - 4,
            dblclick_edit: false,
            column_width: {'track': '25%', 'album': '25%', 'artists': '10%'}
    });

    item.open(true);
}
```

```
<div class="invoices-view">
    <div class="form-body">
        <div class="form-header">
            <h4 id="form-title" class="header-text"><a href="#"></a></h4>
            <h5 id="filter-text" class="header-text pull-right"></h5>
        </div>
        <div class="view-master">
        </div>
        <div class="view-detail" style="margin-top: 4px; margin-bottom: 4px">
        </div>
    </div>
    <div class="form-footer">
        <button id="delete-btn" class="btn expanded-btn pull-left" type="button">
            <i class="icon-trash"></i> Delete<small class="muted"> [Ctrl+Del]</small>
        </button>
        <div id="report-btn" class="btn-group dropup">
            <a class="btn expanded-btn dropdown-toggle" data-toggle="dropdown" href="#">
                <i class="icon-print"></i> Reports
                <span class="caret"></span>
            </a>
            <ul class="dropdown-menu bottom-up">
            </ul>
        </div>
        <button id="filter-btn" class="btn expanded-btn" type="button">
            <i class="icon-filter"></i> Filter
```

(续下页)

(接上页)

```
        </button>
        <button id="edit-btn" class="btn expanded-btn" type="button">
            <i class="icon-edit"></i> Edit
        </button>
        <button id="new-btn" class="btn expanded-btn" type="button">
            <i class="icon-plus"></i> New<small class="muted"> [Ctrl+Ins]</small>
        </button>
    </div>
</div>
```



3.4.9 数据感知控件

要创建一个表格来显示实体项的数据集，请使用*create_table* 方法：

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

要创建数据控件来编辑数据集的字段，请使用*create_inputs* 方法：

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

这些方法有两个参数——**container(容器)** 和 **options(选项)** 。第一个参数是一个 JQuery 容器，控件将放置在其中；第二个参数指定数据显示的方式。详细信息请参阅它们的 API 参考。

这些方法通常在 **on_view_form_created** 和 **on_edit_form_created** 事件处理程序中使用。

所有由这些方法创建的可视控件（表格、输入框、复选框）都是数据感知的。这意味着它们会立即反映实体

项数据集的任何更改。

有时需要禁用这种交互。为此，请分别使用*disable_controls* 和*enable_controls* 方法。

视频

Data aware controls （数据感知控件）

3.5 数据编程

3.5.1 数据集

Jam.py 框架使用的数据集概念非常接近 Embarcadero Delphi 的数据集。

备注

还有其他读取和修改数据库数据的方法。您可以使用 **任务 (task)** 的*connect* 方法从连接池获取连接，并使用该连接通过 Python Database API 访问数据库。

所有 `item_type` 为 "item" 或 "table" 的实体项以及它们的明细表（参见任务树）都可以从项目数据库的关联表中访问数据并写入更改。它们都是 Item 类的对象

- *Item 类* （在客户端）
- *Item 类* （在服务器端）

这两个类具有与数据处理相关的相同属性、方法和事件。

要从项目数据集表中获取数据集（一组记录），请使用 open 方法。此方法根据参数生成 SQL 查询以获取数据集。

数据集打开后，应用程序可以浏览它、更改其记录或插入新记录，并将更改写入实体的数据库表。

例如，以下函数将把 *support_rep_id* 字段值设置为客户端和服务器上 *id* 字段的值：

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

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

这些函数将 **客户 (customers)** 实体作为参数。然后使用 *open* 方法从客户表中获取记录列表，并修改每条记录。最后，使用 apply 方法将更改保存在数据库表中（参见修改数据集 ）。

> **备注**
>
> 有一种更简短的浏览数据集的方法（参见导航数据集）。例如，在 Python 中，以下循环是等效的：

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

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

视频

Datasets 和 Datasets Part 2 通过具体示例演示了几乎所有的数据集操作方法。

3.5.2 浏览数据集

每个活动数据集都有一个游标或指针，指向数据集中的当前行。数据集中的当前行是可以通过 `edit`、`insert` 和 `delete` 方法操作的行，也是其字段值当前显示在表单上数据感知控件中的行。

您可以通过将游标指向移动到不同的行来更改当前行。下表列出了您可以在应用程序代码中用于移动到不同记录行的方法：

客户端方法	服务器端方法	描述
first	*first*	将游标移动到实体数据集的第一行。
last	*last*	将游标移动到实体数据集的最后一行。
next	*next*	将游标移动到实体数据集中当前行的下一行。
prior	*prior*	将游标移动到实体数据集中当前行的上一行。

除了这些方法之外，下表描述了两种在遍历数据集记录时能提供有用信息的方法：

客户端方法	服务器端方法	描述
bof	*bof*	如果该方法返回 true，则游标位于数据集的第一行；否则，不确定游标是否位于数据集的第一行。
eof	*eof*	如果该方法返回 true，则游标位于数据集的最后一行；否则，不确定游标是否位于数据集的最后一行。

每次游标在数据集中移动到另一条记录时，都会触发以下事件：

客户端事件	服务器端事件	描述
*on_before_*	`on_befor`	在应用程序显示的记录从一条切换到另一条之前发生。
on_after_s	`on_after`	在应用程序显示的记录从一条切换到另一条之后发生。

使用这些方法，我们可以浏览数据集。例如，

在客户端：

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

在服务器端：

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

更简洁的浏览数据集方式

客户端上有*each* 方法，可用于浏览数据集：

例如：

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

在服务器端，我们可以通过以下方式遍历数据集行：

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

两个函数都将输出数据集中前 10 条记录的客户姓名。

在这两种情况下，**c** 和 **customers** 都是指向同一对象的指针。

3.5.3 修改数据集

当应用程序打开一个实体数据集时，数据集自动进入 **浏览 (browse)** 状态。“浏览”状态允许您查看数据集中的记录，但您不能编辑记录或插入新记录。在数据集中，您主要使用“浏览”状态来从一条记录切换到另一条记录。

有关在记录之间切换的更多信息，请参阅导航数据集 。

可以将数据集从“浏览”状态设置为其他状态。例如，调用 *insert* 或 *append* 方法会将其状态从“浏览”更改为“插入”。

有两种方法可以将数据集返回到“浏览”状态。取消 `(Cancel)` 结束对当前记录的编辑、插入操作，并将数据集返回到“浏览”状态。提交 `(Post)` 将对当前记录的更改写入数据集。如果成功，也会将数据集返回到“浏览”状态；如果提交操作失败，则当前状态保持不变。

要检查数据集状态，请使用 `item_state` 属性或 `is_new` 、`is_edited` 或 `is_changing` 方法：

客户端	服务器端	描述
item_state	*item_state*	表示实体数据集的当前操作状态。
is_new	*is_new*	如果实体数据集处于 插入状态，则返回 true。
is_edited	*is_edited*	如果实体数据集处于 编辑状态，则返回 true。
is_changin	*is_changin*	如果实体数据集处于 插入或 编辑状态，则返回 true。

您可以使用以下实体对象的方法在数据集中插入、更新和删除数据：

客户端	服务器端	描述
edit	*edit*	将实体数据集置于编辑状态。
append	*append*	将一条记录追加到数据集的末尾，并将数据集置于 插入状态。
insert	*insert*	在数据集的开始处插入一条记录，并将数据集置于 插入状态。
post	*post*	保存新的或更改过的记录，并将数据集置于“浏览”状态。
cancel	*cancel*	取消当前操作，并将数据集置于“浏览”状态。
delete	*delete*	删除当前记录，并将数据集置于“浏览”状态。

对数据集所做的所有更改都存储在内存中，对实体记录的更改会写入日志。因此，在完成所有更改后，可以通过调用 `apply` 方法生成并执行 SQL 查询，将所有更改存储到关联的数据库表中。

客户端	服务器端	描述
log_changes	*log_changes*	表示是否记录数据更改。
apply	*apply*	将实体数据集中所有更新、插入和删除的记录发送到服务器，以写入数据库。

3.5.4 字段

所有与数据库中数据表关联的实体项都有一个*fields* 属性——一个字段对象列表，用于表示实体项对应的表中的字段。

每个字段都有以下属性：

客户端	服务器端	描述
owner	*owner*	拥有此字段的实体项。
owner	*owner*	拥有此字段的实体。
field_name	*field_name*	字段的名称，将在编程代码中用于访问字段对象。
field_name	*field_name*	字段的名称，将在编程代码中用于访问字段对象。
field_capti	*field_capti*	向用户显示的字段标题。
field_capti	*field_capti*	向用户显示的字段名称。
field_type	*field_type*	字段类型，为以下值之一：**text** 、**integer** 、**float** 、**currency** 、**date** 、**datetime** 、**boolean** 、**blob** 。
field_type	*field_type*	字段类型，为以下值之一：**text** 、**integer** 、**float** 、**currency** 、**date** 、**datetime** 、**boolean** 、**blob** 。
field_size	*field_size*	类型为 **text** 的字段的大小。
field_size	*field_size*	类型为 **text** 的字段的大小。
required	*required*	指定字段是否需要非空值。
required	*required*	指定字段是否需要非空值。

为了访问实体项对应的数据集数据，Field 类具有以下属性：

Client	Server	Description
value	*value*	使用此属性获取或设置当前记录的字段值。读取时，值会转换为字段的类型。因此，对于 integer、float 和 currency 类型的字段，如果数据库表记录中此字段的值为 NULL，则此属性的值为 0。要获取未转换的值，请使用 raw_value 属性。
text	*text*	使用此属性以文本形式获取或设置字段的值。
lookup_va	*lookup_va*	使用此属性获取或设置查找值，请参阅查找字段 。
lookup_tex	*lookup_tex*	使用此属性以文本形式获取或设置字段的查找值，请参阅查找字段 。
display_text	*display_text*	表示字段值在数据感知控件中的显示形式。如果字段是查找字段，则其值为 lookup_text 值；否则，根据项目区域设置参数，取其 text 值。此行为可由拥有该字段的实体的*on_field_get_text* 事件处理程序覆盖。
raw_value	*raw_value*	使用此属性获取当前记录的字段值，这与存储在数据库中的形式一致。不进行任何转换。

此外，每个字段都是其所属实体的一个属性。因此，要访问实体的字段，请使用以下语法：`item.field_name`

```
invoices.total.value
```

`invoices.total` 是对 **发票 (Invoices)** 实体的 **总计 (Total)** 字段的引用，而 `invoices.total.value` 是该字段的值。

以下是演示项目 中 **发票 (Invoices)** 实体项的字段的属性值：

```
customer integer
   value:  2
   text:  2
   lookup_value:  Köhler
   lookup_text:  Köhler
   display_text:  Leonie Köhler
firstname integer
   value:  2
   text:  2
   lookup_value:  Leonie
   lookup_text:  Leonie
```

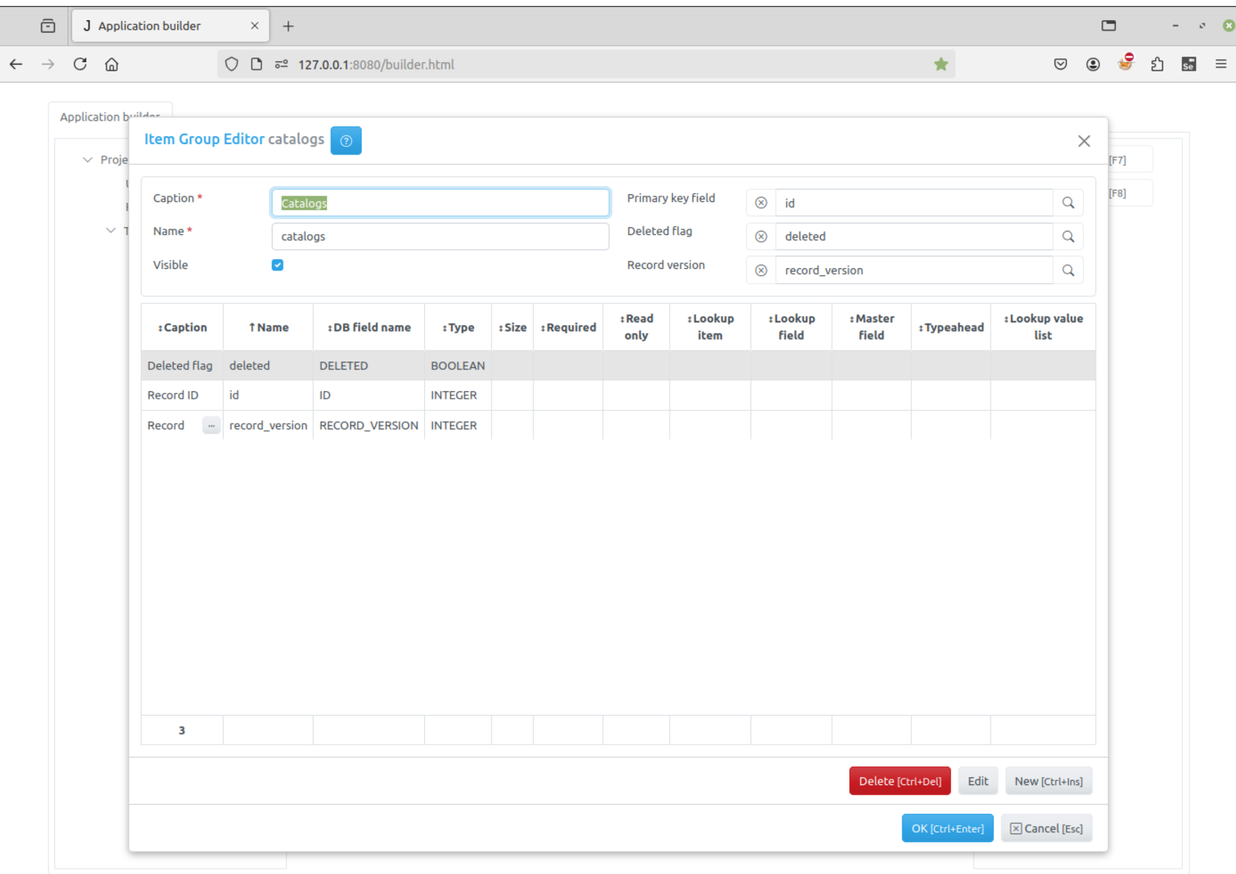
(续下页)

(接上页)

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

3.5.5 公共字段

可以访问数据库数据的实体项可以拥有公共字段。这在实体项所属的组中定义:



这里定义了两个字段：**id** 和 **deleted** 。

id 字段被设置为主键，将存储数据库表中每条记录的唯一标识符。当向表中插入新记录时，此值由框架自动生成。

deleted 字段被设置为删除标志。当在实体编辑器对话框 中勾选“软删除 (Soft delete)”复选框时，删除方法不会从表中真正地删除记录，而是使用此字段将记录标记为已删除。当生成从数据库表中获取记录的 SQL 查询时，open 方法会考虑到这一点。

记录版本 (Record version) 字段用于记录锁定 。

3.5.6 查找字段

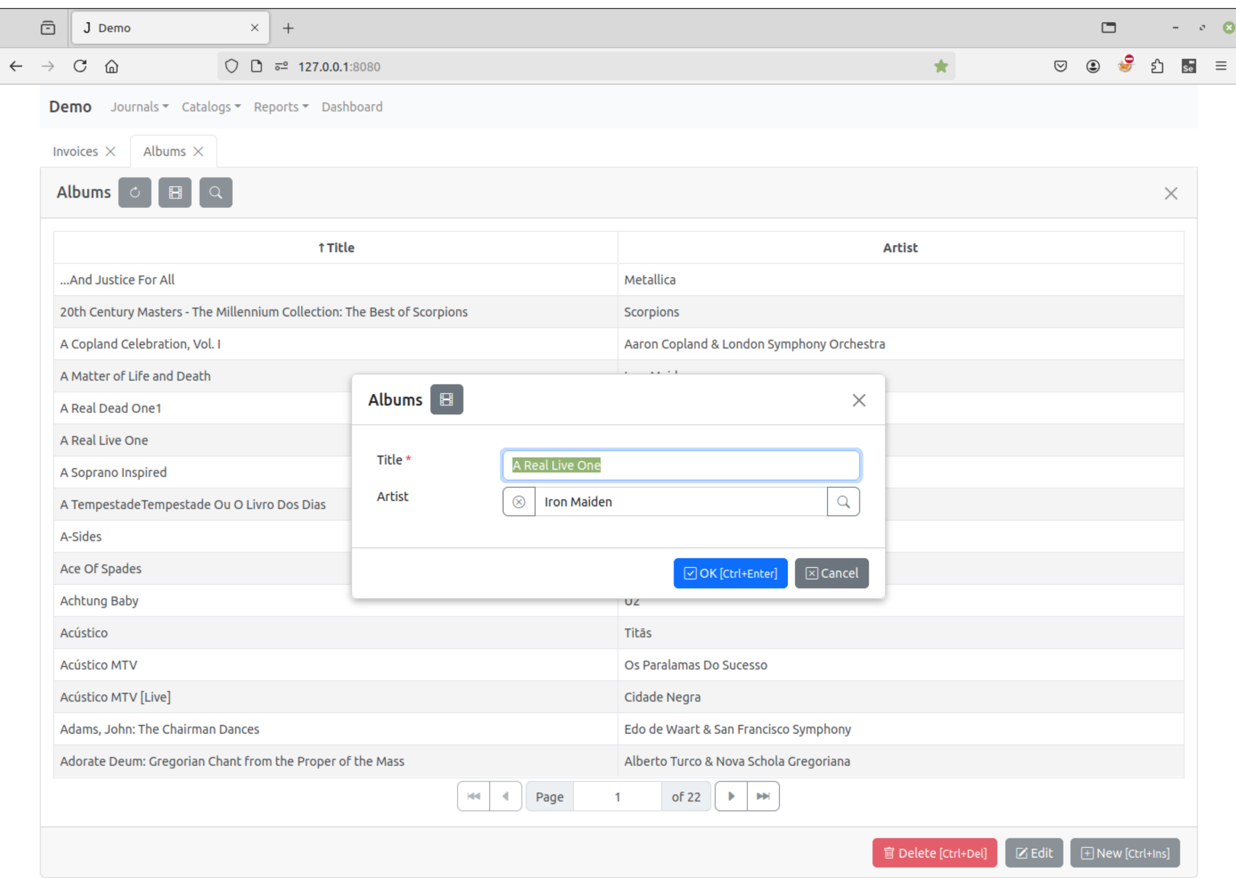
查找字段可以显示一个用户友好的值，该值绑定到另一个表或值列表中的另一个值。例如，查找字段可以显示一个客户名称，该名称绑定到另一个实体表或列表中的相应客户 ID 号。

要在查找字段中输入值时，用户可以从值列表中选择。这可以使数据输入更快、更准确。

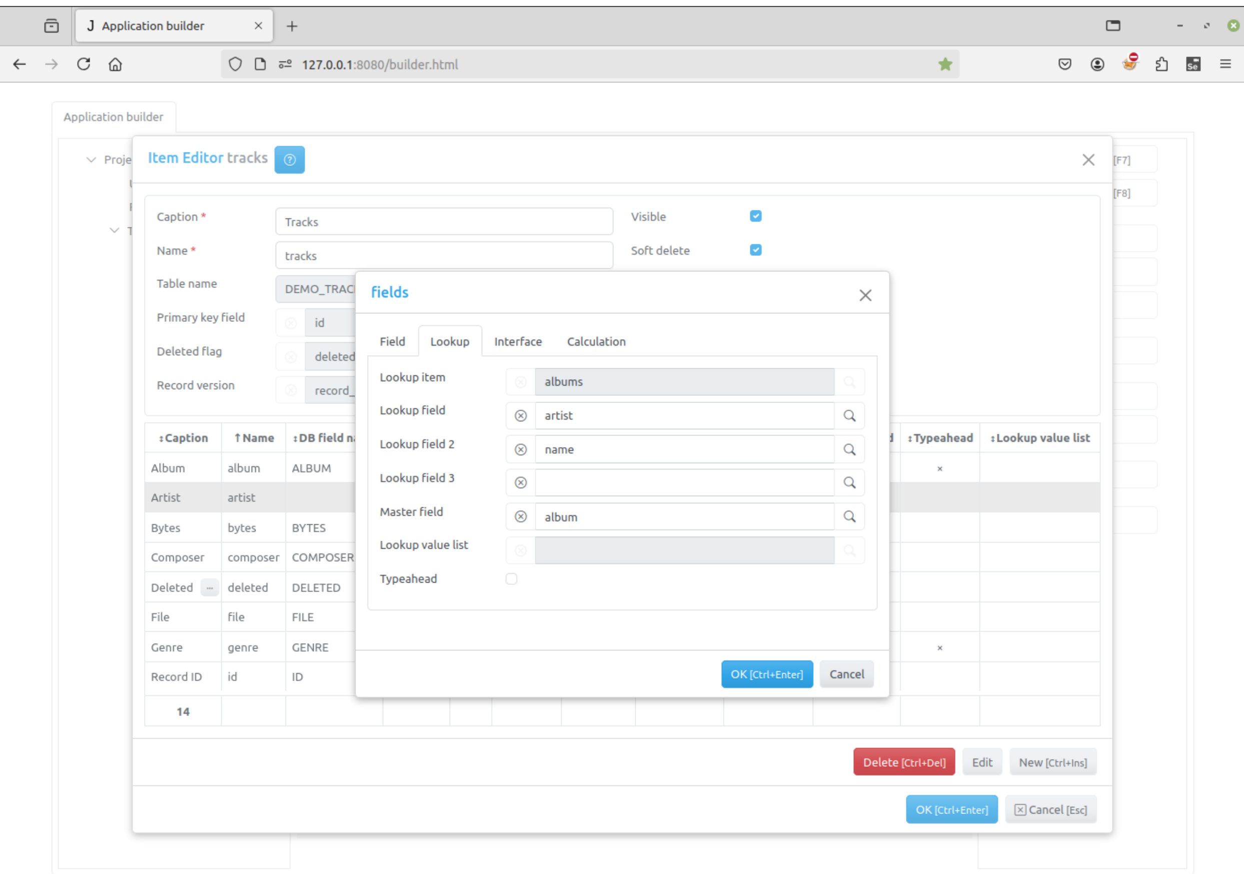
您可以创建的查找字段有两种类型：基于查找实体的查找字段，和基于值列表的查找字段。

基于查找实体的查找字段

在框架中，您可以向实体添加一个字段，以查找另一个实体表中信息。例如，在演示应用程序的 **专辑 (Albums)** 目录中，有一个 **艺术家 (Artist)** 查找字段。



要设置字段的值，用户必须点击字段输入框右侧的按钮，并从出现的 **艺术家 (Artist)** 列表中选择一条记录。然后该字段的值将是选中记录的 id 。如果在字段编辑器对话框 中设置了 **预输入 (Typeahead)** 标志，设置字段值的另一种方法可以使用预输入：



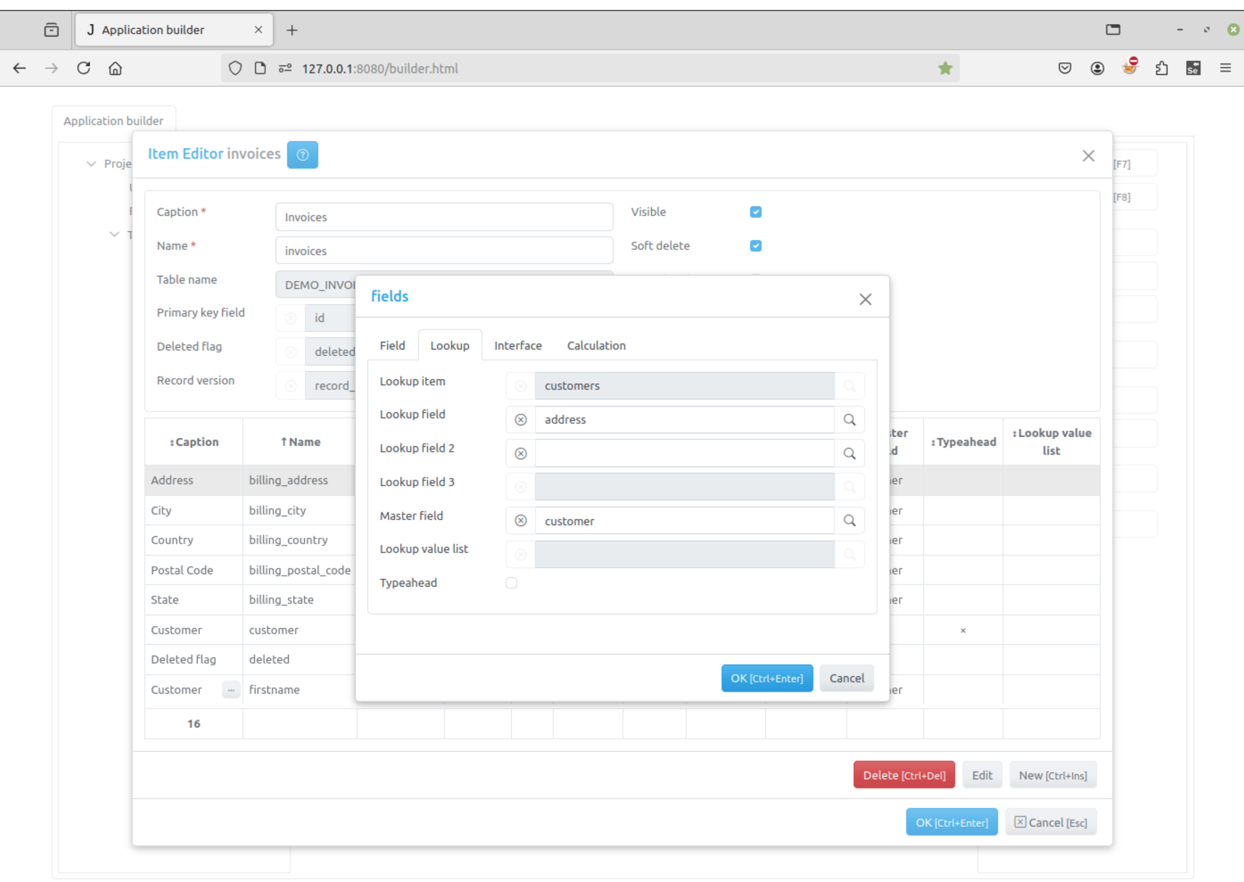
对于此类字段，必须在*字段编辑器对话框*中指定 **查找实体 (Lookup item)** 和 **查找字段 (Lookup field)**：

当调用 open 方法且 expanded 参数设置为 true（默认）时，服务器上生成的 SQL 查询使用 JOIN 子句来获取此类字段的查找值。因此，每个这样的字段都有一对值：第一个值存储对查找实体表中记录的引用（其主键字段的值），第二个值具有该记录中查找字段的值。

要访问这些值，请使用查找字段的以下属性：

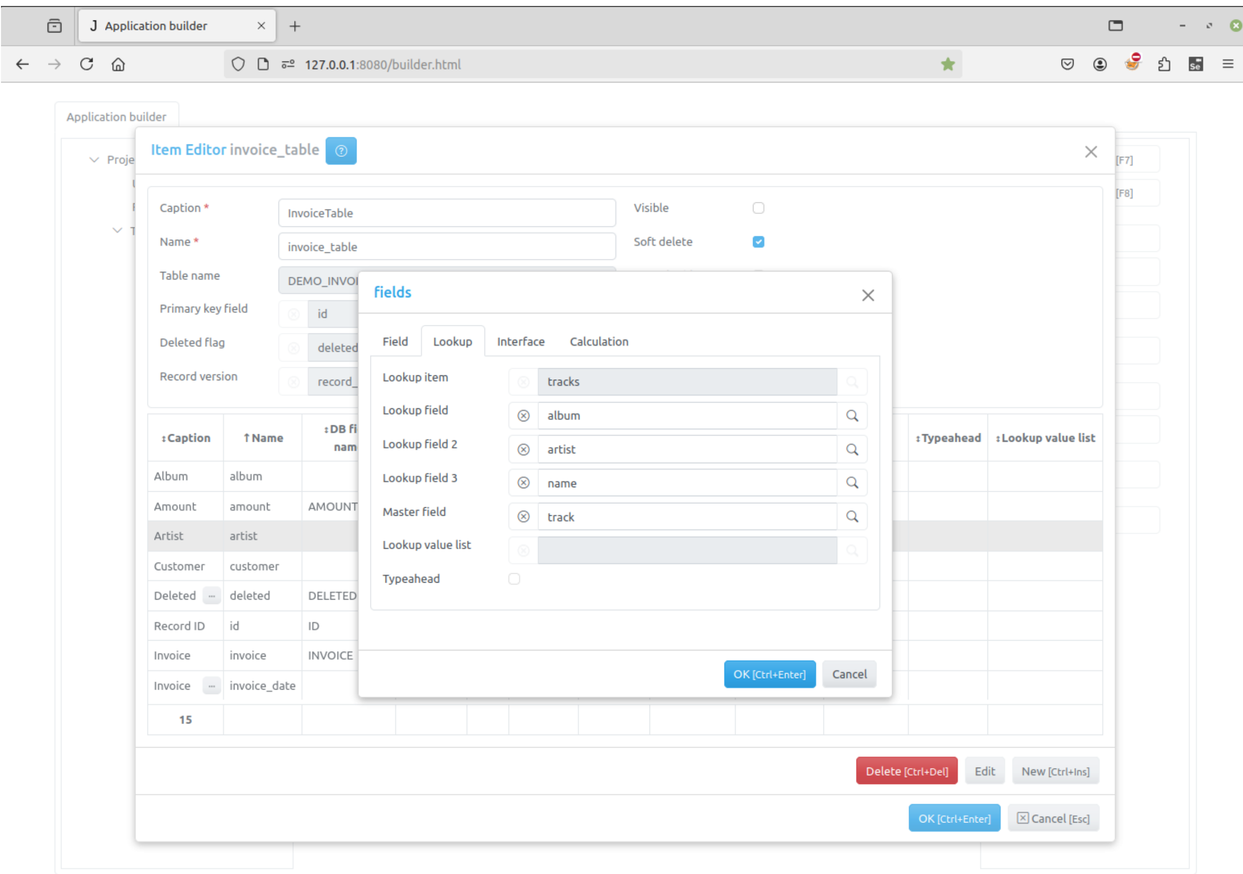
客户端	服务器端	描述
value	*value*	存储在实体表中的值，它是对查找实体表中记录的引用。
lookup_va	*lookup_va*	查找实体表中查找字段的值。

有时需要从查找实体表中的同一记录获取两个或多个值。例如，演示应用程序中，业务台账下的”发票 (Invoices)”有多个查找字段（“联系人 (Customer)”、”账单地址 (Billing Address)”、“账单城市 (Billing City)”等），这些字段包含有关客户的信息，所有这些信息都存储在“客户 (Customers)”实体项对应的表的一条记录中，用于描述该客户。为了避免在“发票 (Invoices)”实体对应的表中创建不必要的字段，存储相同的记录引用，并为每个此类字段创建 JOIN ，除“联系人 (Customer)”之外的所有查找字段都将其 **主字段 (Master field)** 值指向“联系人客户 (Customer)”字段。这些字段在实体底层的数据库表中没有对应的字段。它们的 value 属性始终等于主字段的 value 属性，并且当调用 open 方法时，在服务器上生成的 SQL 查询对所有此类字段使用一个 JOIN 子句。



当用户点击字段输入框右侧的按钮或使用预输入时，应用程序会创建字段对应的查找实体项的副本，设置其*lookup_field* 属性为该字段，并触发*on_field_select_value* 事件。编写此事件处理程序以指定将显示的字段，在打开查找实体并显示给用户以选择字段值之前，为其设置过滤器。

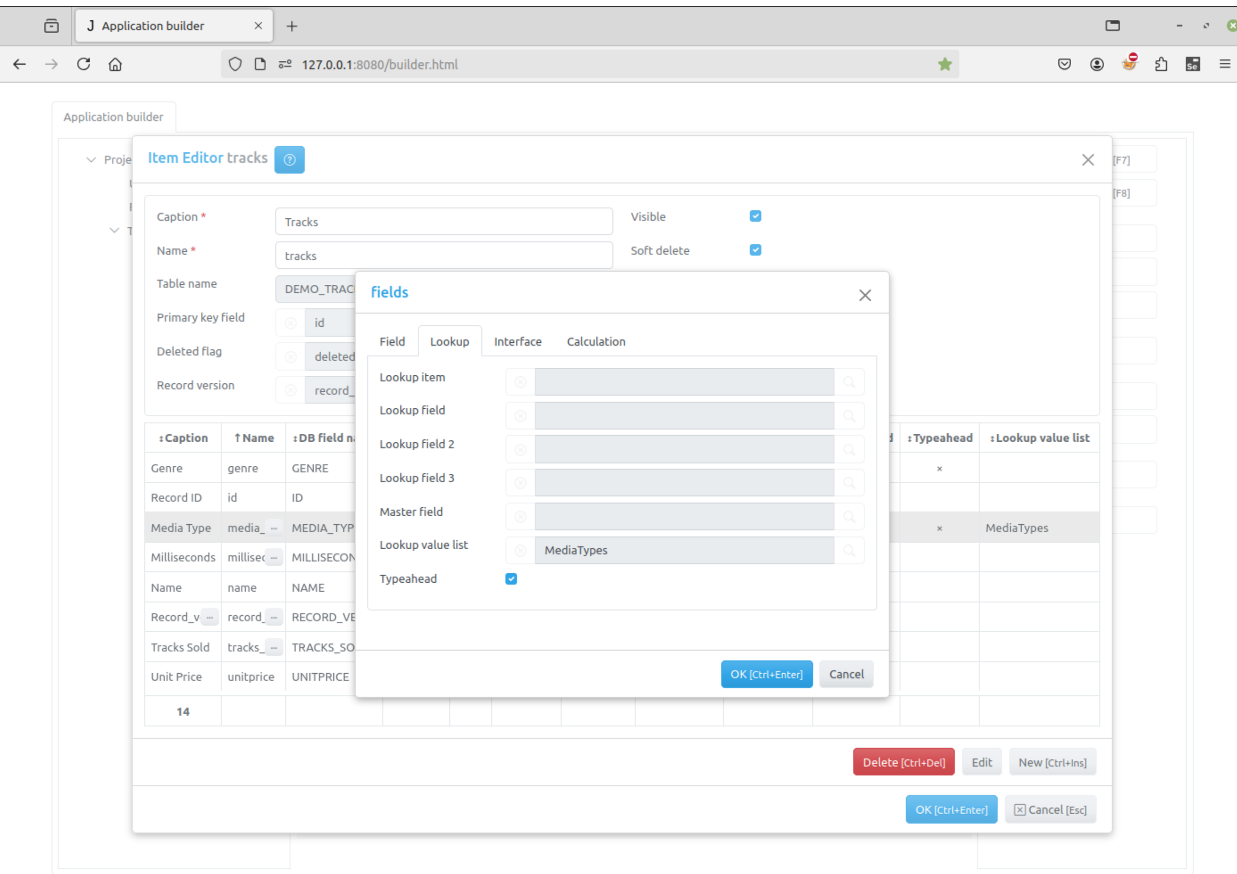
查找实体中的查找字段也可以是查找字段，例如：



要设置此类字段，请使用 **查找字段 2(Lookup field 2)** 和 **查找字段 3(Lookup field 3)** 属性。

值列表

有时，查找字段的来源可以定义为值列表。例如，演示项目 的 **曲目 (Tracks)** 目录中的 **媒体类型 (MediaType)** 字段具有 **查找值列表 (Lookup value list)** 属性，设置为 MediaTypes(媒体类型) 查找列表：



使用任务的查找列表对话框 来定义此类查找列表。

另请参阅

查找字段

查找列表

3.5.7 筛选记录

在调用 open 方法时，有三种方法可以定义从数据库的表中将获取实体数据集 的哪些记录：

- 指定 open 方法的 where 参数
- 在调用 open 方法之前调用 set_where 方法
- 或者使用过滤器

当指定了 where 参数时，即使调用了 set_where 方法或实体具有已设置值的过滤器，也始终使用 where 参数。

当省略 where 参数时，将使用传递给 set_where 方法的参数。

例如，在以下代码的 客户端模块 (Client Module) 中，在第一次调用 open 方法时，将使用 where 选项来筛选记录；在第二次调用时，将使用传递给 set_where 的参数，而只有在第三次才会使用 invoicedate1 过滤器的值：

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

(续下页)

(接上页)

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

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

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

    invoices.open();
}


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

在 服务器模块 (Server Module) 中的相同功能代码如下所示:

```
from datetime import datetime

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

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

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

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

    invoices.open()
```

在框架中，定义了以下符号和相应的常量来筛选记录:

Filter type	Filter symbol	Constant	SQL Operator
EQ	'eq'	FILTER_	=
NE	'ne'	FILTER_	<>
LT	'lt'	FILTER_	<
LE	'le'	FILTER_	<=
GT	'gt'	FILTER_	>
GE	'ge'	FILTER_	>=
IN	'in'	FILTER_	IN
NOT IN	'not_in'	FILTER_	NOT IN
RANGE	'range'	FILTER_	BETWEEN
ISNULL	'isnull'	FILTER_	IS NULL
EXACT	'exact'	FILTER_	=
CONTAINS	'contains'	FILTER_	使用 LIKE 和"%" 符号来查找字段值包含搜索字符串的记录
STARTWITH	'startwith'	FILTER_	使用 LIKE 和"%" 符号来查找字段值以搜索字符串开头的记录
ENDWITH	'endwith'	FILTER_	使用 LIKE 和"%" 符号来查找字段值以搜索字符串结尾的记录
CONTAINS ALL	'contains_all'	FILTER_	使用 LIKE 和"%" 符号来查找字段值包含搜索字符串所有单词的记录

open 方法的 where 参数是一个字典，其键是字段名称，后面跟着双下划线和过滤器符号。对于 EQ 过滤器，过滤符号'__eq' 可以省略。例如，{'id': 100} 等价于 {'id__eq': 100}。

'__isnull' 过滤符号与布尔值一起使用。

例如，在上面的 服务器模块 (Server Module) 代码中：

```
invoices.set_where(invoicedate__isnull = True)
```

另请参阅

数据集 ，过滤器

客户端

open ，*set_where*

服务器端

open ，*set_where*

3.5.8 过滤器

对于每个可以访问数据库中数据表的实体项，都可以创建一个包含过滤器对象的列表。

要创建过滤器，请使用应用程序构建器的过滤器对话框 。

过滤器为用户提供了一种可视化的便捷方式来指定请求的参数，该请求由应用程序发送给项目使用的数据库。

每个过滤器都有以下属性：

- owner —拥有此过滤器的实体，
- filter_name —过滤器的名称，可在编程代码中使用
- filter_caption —过滤器的标题，用于客户端应用程序中的可视化表示
- filter_type —过滤器的类型，请参阅过滤记录
- visible —如果此属性的值为 true ，当未指定 filters 选项时，*create_filter_inputs* 方法将为此过滤器创建可视化表示
- value —过滤器的值

实体项的所有过滤器都是其 filters 对象的属性。通过使用 filter_name ，我们可以访问过滤器对象：

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

另一种访问过滤器的方法是使用*filter_by_name* 方法：

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

另请参阅

数据集 ，过滤记录

客户端

filters

Filter 类

assign_filters

clear_filters

each_filter

filter_by_name

服务器端

filters

Filter 类

clear_filters

filter_by_name

3.5.9 明细表

在框架中，明细表用于处理与实体项对应的表中记录相关的数据。

例如，演示应用程序中，业务台账下的 **发票 (Invoices)** 拥有 **发票表格 (InvoiceTable)** 明细表，用于保存客户发票中的项目列表。

明细表与明细实体项共用同一个底层数据库表。

要创建明细表，您必须首先创建一个明细实体项（选择项目树中的明细表组 (Details)，然后点击新建按钮 (New)），然后使用明细表对话框（在项目树中选择实体项，然后点击细表按钮 (Details)）为实体项添加明细表。

例如，以下代码

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

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

将打印

```
2259
6
```

在 Jam.py v5 中，明细表有两个公共字段 - `master_id` 和 `master_rec_id`，用于存储有关主表 `ID` 的信息（每个实体项都有自己唯一的 ID）以及其主表记录的主字段值。这样，每个表都可以被链接到多个实体项。同样，每个实体项也可以有多个明细表。要访问实体项的明细表，请使用其 `details` 属性。要访问明细表的主表，请使用其 `master` 属性。

在 Jam.py v7 中，上述两个字段不再存在，但对于迁移的 v5 应用程序，它们仍受支持。Jam.py v7 应用程序将使用查找字段作为主表的明细表，通常作为外键字段。

用于创建明细表的 Detail 类是 Item 类的祖先，并继承了其所有的属性、方法和事件。

> **备注**
>
> Detail 类的 `apply` 方法不执行任何操作。要保存对明细表所做的更改，请使用其主表的 `apply` 方法。
>
> 要使用明细表，其主表必须处于活动状态。
>
> 要对明细表进行任何更改，其主表必须处于编辑或插入模式。

示例

在这个来自演示项目 的 **发票 (Invoices)** 实体项客户端模块的示例中，每次其主表的游标移动到另一条记录时，**发票表格 (Invoice_table)** 明细表都会重新打开。

```
var ScrollTimeOut;

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

以下是一个示例:

```
from datetime import datetime, timedelta

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

客户端上功能相同的代码如下:

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

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

(续下页)

(接上页)

```
            t.post();
        });
        i.post();
    });
    invoices.apply();
}
```

3.6 服务器端编程

在大多数情况下，当执行实体项的以下方法时，客户端会向服务器发送请求：

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

在这些情况下，客户端向服务器发送实体项任务的*ID*、实体项的*ID*、请求类型及其参数。

服务器收到请求后，根据传递的 ID，找到服务器上的任务（可以是项目任务或应用程序构建器任务）和实体项，使用传递的参数执行相应方法，并将执行结果返回给客户端。服务器方法可以触发事件，这些事件可以修改其默认行为。

任务树中的每个实体项都有*environ* 和*session* 属性，用于存储当前请求的上下文。

最常见的服务器事件有：

- *on_created* - 当任务刚刚被服务器应用程序创建时，由任务触发此事件。可用于初始化项目。
- *on_apply_events* - 当在客户端 或服务端 调用实体项的 `apply` 方法时触发这些事件。
- *on_open_events* - 当在客户端 或服务端 调用实体项的 `open` 方法时触发这些事件。
- *on_generate* - 当在客户端调用 **报表 (report)** 的*print* 方法时触发此事件。

> **备注**
>
> 请注意，服务器上的任务树是不可变的，您不能更改任务树中实体项的属性。
>
> 您必须使用*copy* 方法来创建实体项的副本。此副本是在创建任务树时生成的实体项的精确复制品。它不会添加到任务树 中，并且在不再需要时会被 Python 垃圾回收器销毁。

3.6.1 on_apply 事件

当在客户端 或服务端 调用项目的 `apply` 方法时，服务器应用程序默认会根据对数据集所做的更改生成 SQL 查询语句并执行。

可以通过在实体项的服务端模块中编写*on_apply* 事件处理程序来更改此行为。

有时需要在保存更改时为所有实体项执行一些代码。在这种情况下，可以使用任务的 `on_apply` 事件处理程序（在任务服务端模块中声明）。

以下代码描述了如何处理这些事件：

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

它会检查任务是否有 `on_apply` 事件处理程序。如果在任务服务器模块中声明了 `on_apply` 事件处理程序，那么它会被执行。

如果任务的 `on_apply` 事件处理程序未声明或事件处理程序的结果返回 `None`，该方法会检查实体项是否有*on_apply* 事件处理程序。如果在实体项服务端模块中声明了，则它会被执行。

如果实体项事件处理程序返回的结果是 `None` ，则调用实体项的 `apply_delta` 方法，该方法会生成 SQL 查询，并在执行后返回查询结果。

示例

这是一个如何使用 on_apply 的示例

3.6.2 on_open_events

当在*客户端* 或*服务端* 调用实体项的 `open` 方法时，服务器应用程序执行以下代码：

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

它会检查任务是否有 `on_open` 事件处理程序。如果在任务服务器端模块中声明了 `on_open` 事件处理程序，则它会被执行。

如果任务的 `on_open` 事件处理程序未声明或事件处理程序的结果返回 `None` ，该方法会检查实体项是否有*on_open* 事件处理程序。如果它在实体项服务器模块中被声明，则会被执行。

如果实体项事件处理程序返回的结果是 `None` ，则调用实体项的 `execute_open` 方法，该方法会生成 SQL 查询，并在执行后返回查询结果。

示例

这是一个如何使用 on_open 的示例

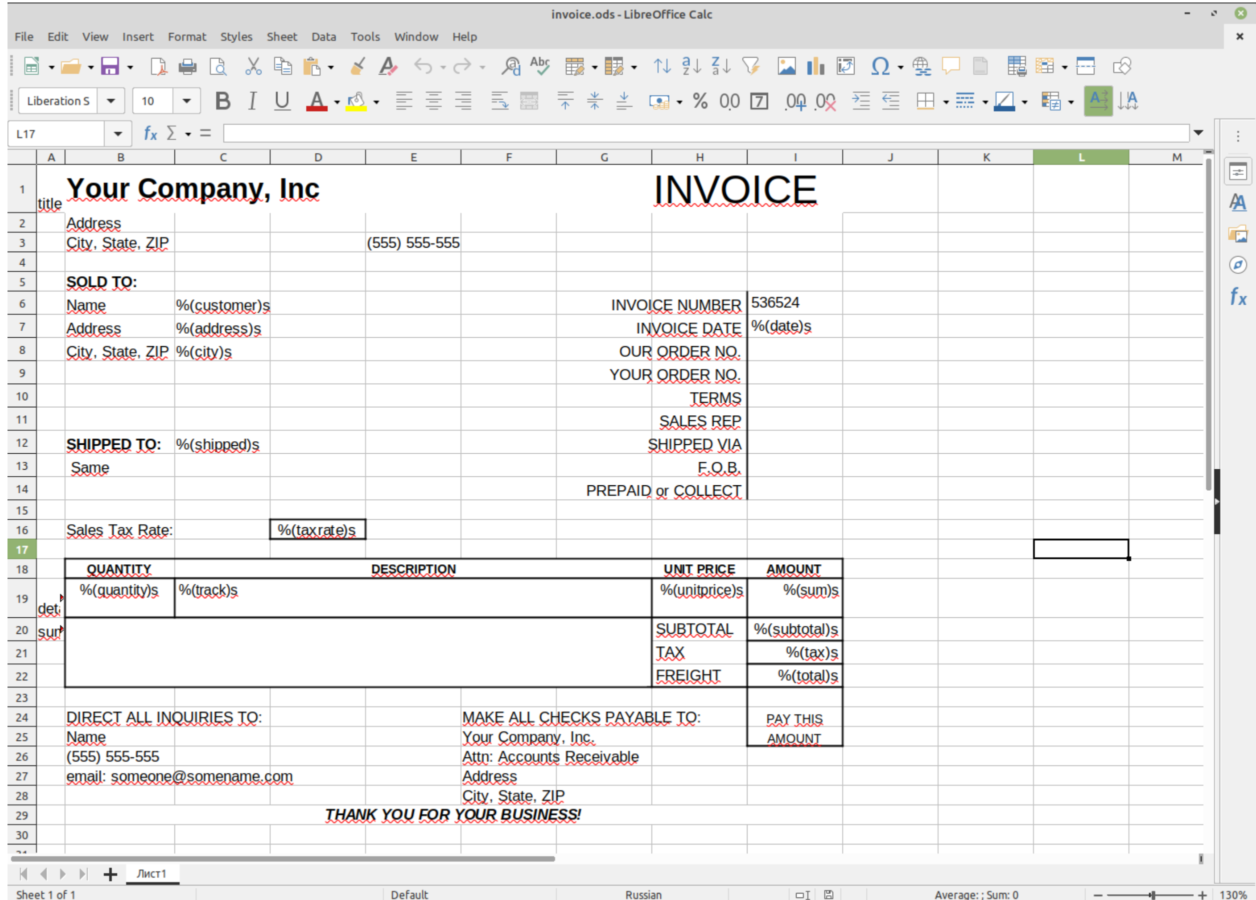
3.7 报表编程

3.7.1 报表模板

要创建报表，首先必须在 LibreOffice Calc 中准备报表模板。

模板文件位于项目目录的 report 文件夹中。

下图显示了“发票 (Invoice) 报表”的模板。



Jam.py 中的报表布局是面向区段的 (band-oriented) 。

每个报表模板都分为若干区段。要设置区段，请使用模板电子表格中最左侧列。

在 “发票 (Invoice) 报表” 模板中，有三个区段 (band)： **标题 (title)** 、**明细 (detail)** 和 **汇总 (summary)** 。

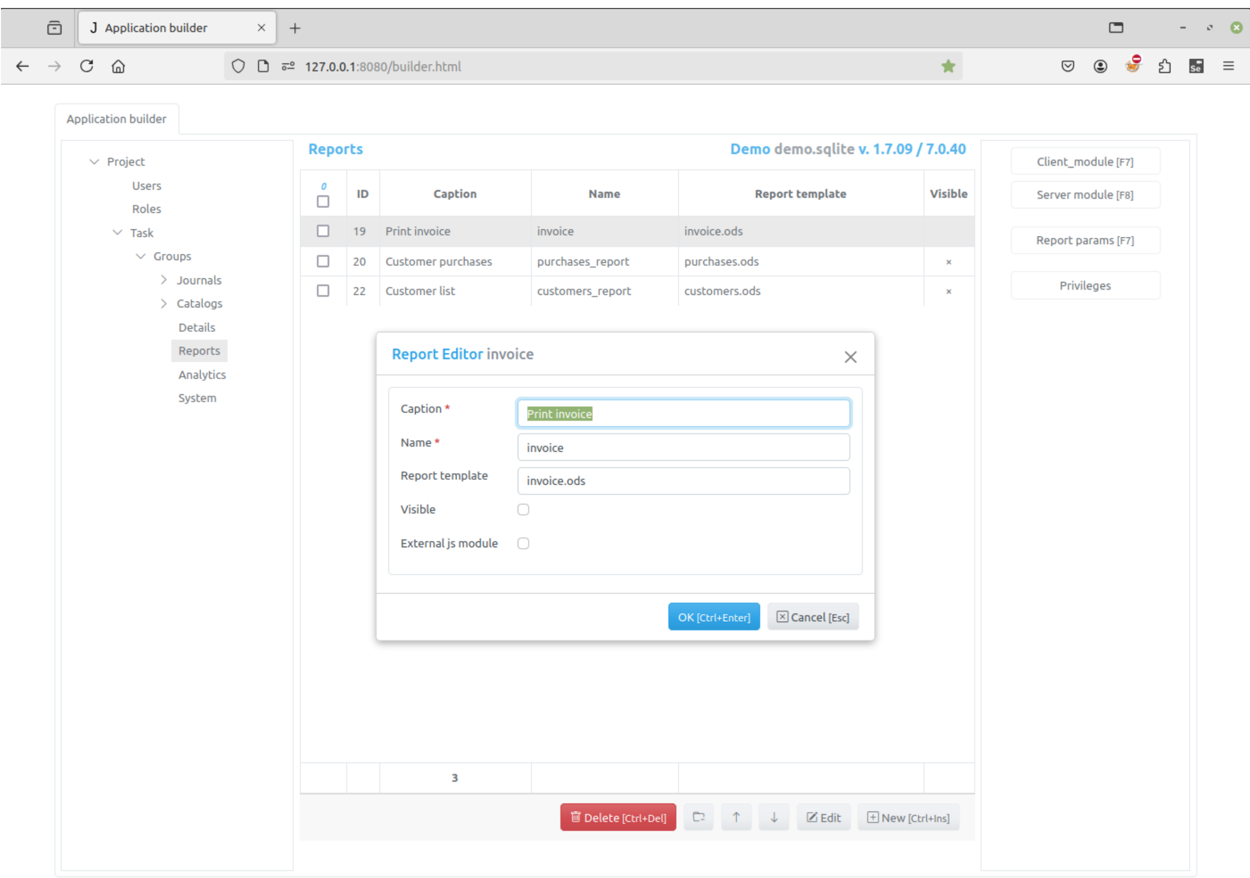
此外，模板可以包含可编程单元格。

例如，在 Invoice 报表模板中，I7 单元格包含文本%(date)s 。

可编程单元格以 **%** 开头，然后是一对括号，括号里面是填充模板的函数中传递的变量的名称，最后是字符 **s**。例如，填充模板的函数会使 date 变量的值，替换%(date)s 占用的单元格。

3.7.2 创建报表

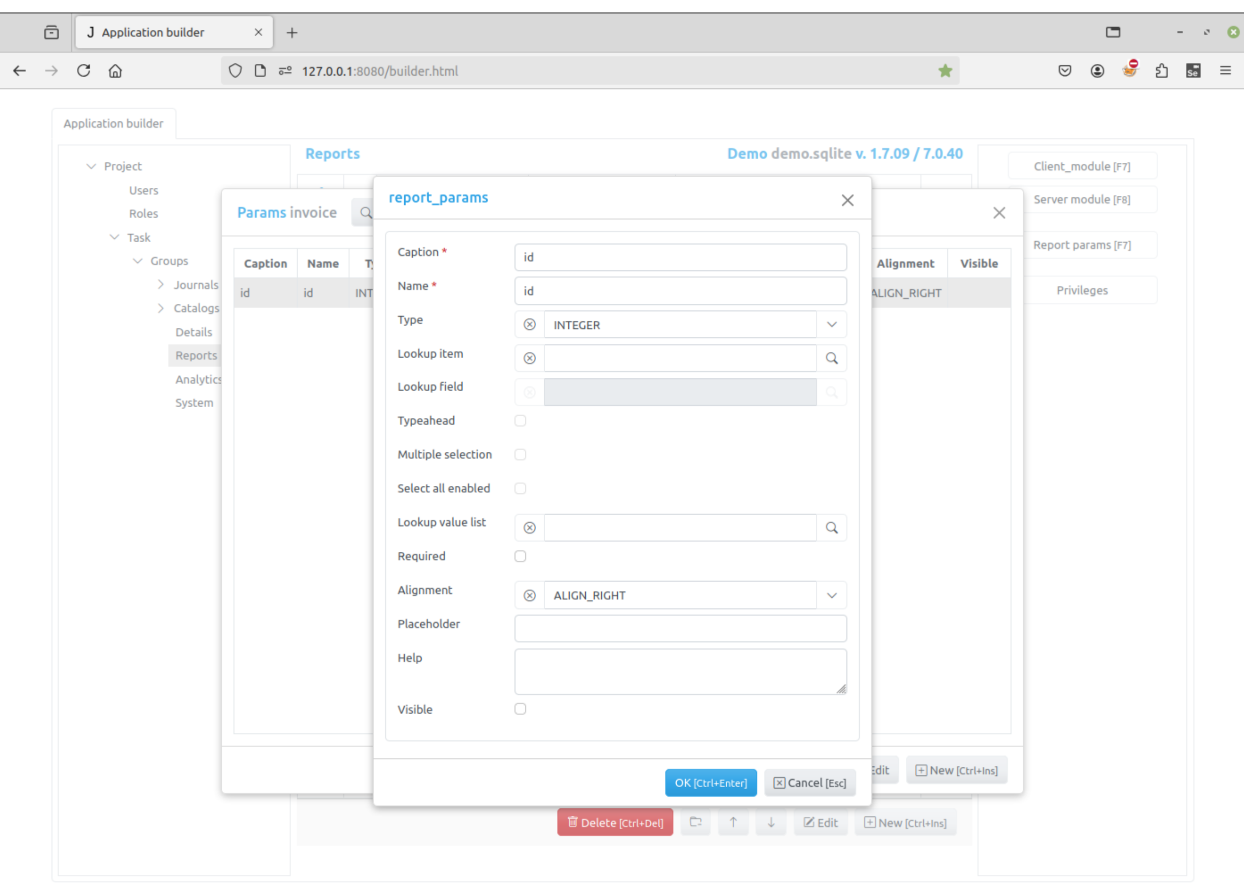
要向 Jam.py 项目添加新报表，请在项目树中选择 **报表 (Reports)** 节点，然后点击 **新建 (New)** 按钮，并填写报表的标题、名称和模板文件名。



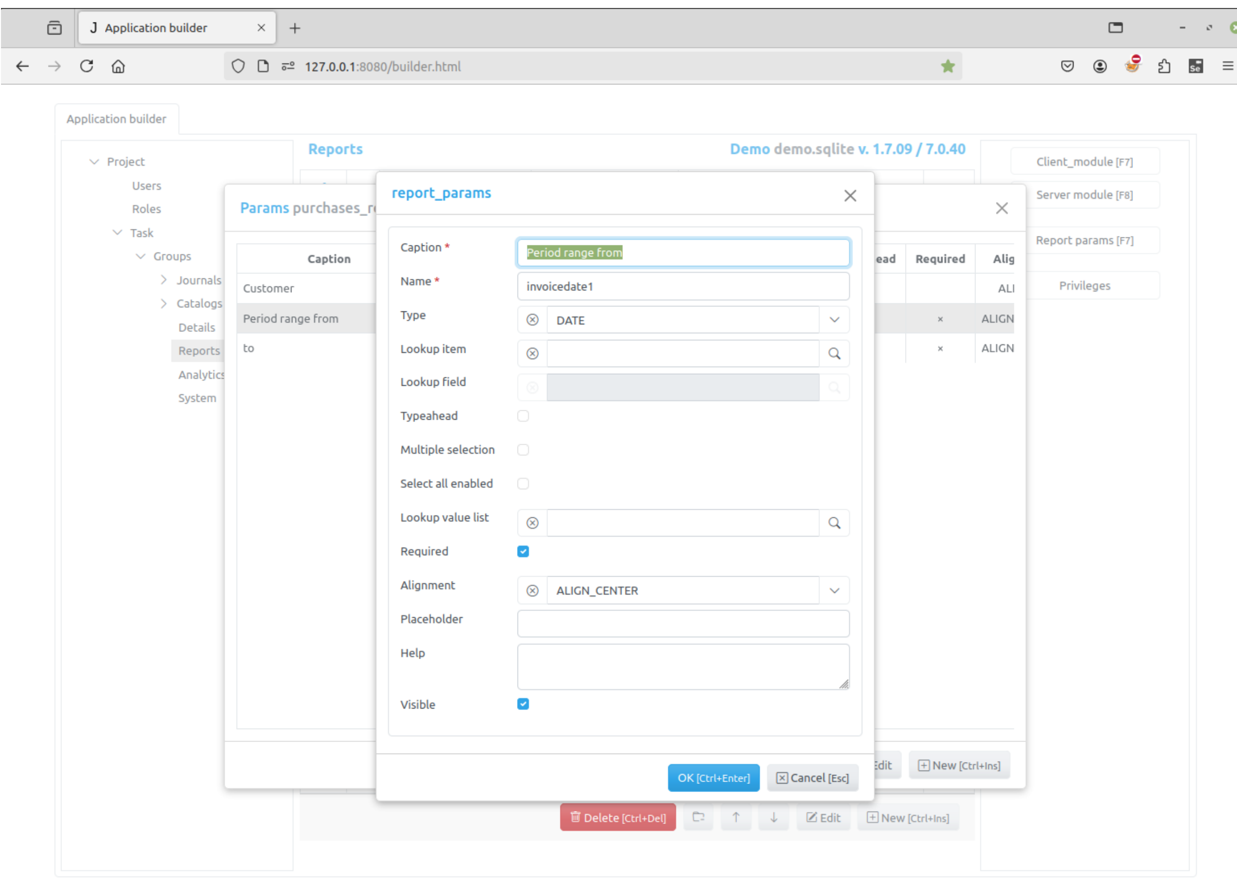
如果勾选了 **可见 (visible)** 复选框，代码会默认地在项目客户端显示 **报表 (Reports)** 菜单，其子菜单可打开报表。

3.7.3 报表参数

您可以指定报表的参数。例如，演示项目的 **客户购买情况 (Customer purchases)** 报表有三个参数。



要添加或更改报表参数，请在应用程序构建器左侧面板中点击 **报表参数 (Report params)** 按钮。将显示一个表单，列出已有的参数。然后点击表单中的 **新建 (New)** 或 **编辑 (Edit)** 按钮以添加或更改参数。



可以指定以下内容：

- **Caption** （标题）- 向用户显示的参数名称
- **Name** （名称）- 参数名称，将在编程代码中用于访问参数对象
- **Type** （类型）- 参数的数据类型
- **Lookup item** （查找项）- 用于选择参数值的实体项
- **Lookup field** （查找字段）- 查找项中的字段
- **Typeahead** （预输入）- 为查找字段启用自动补全/预输入功能
- **Multiple selection** （多选）- 启用多选功能
- **Select all enabled** （启用全选）- 启用全选。
- **Lookup value list** （查找值列表）- 查找值列表
- **Required** （必填）- 如果勾选此复选框且设置了 **可见 (Visible)** 属性，客户端应用程序将在打印报表前要求用户指定参数值。
- **Alignment** （对齐方式）- 指定参数值在输入元素中的对齐方式。
- **Placeholder** （占位符）- 使用此属性指定字段输入控件中将显示的占位符。
- **Help** （帮助）- 如果指定了任何 **text/HTML 消息**，将在输入框右侧显示一个问号，

当用户将鼠标指针移到该标记上时，将弹出一个窗口显示此消息。

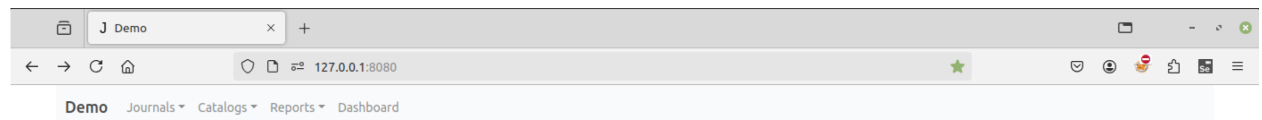
- **Visible** （可见）- 客户端应用程序在打印报表前创建一个用于指定参数的表单。如果勾选此复选框，此参数的输入元素将出现在表单中。

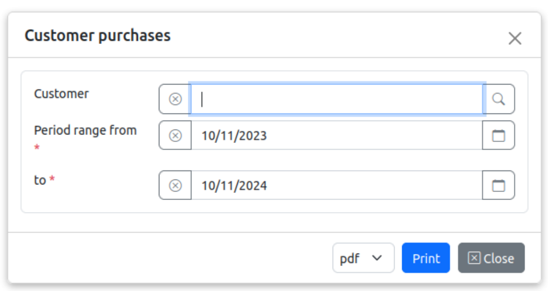
可以创建查找参数。例如，**客户购买情况 (Customer purchases)** 报表有一个 **客户 (Customer)** 参数，可以从 **客户 (Customers)** 主表中选择：

在这种情况下，我们应该指定：

- **Lookup item** （查找项）- 用于选择参数值的实体
- **Lookup field** （查找字段）- 查找项中的字段

设置 **客户购买情况 (Customer purchases)** 报表参数的表单如下所示：





3.7.4 客户端报表编程

要在客户端打印报表，请使用*print* 方法。

调用此函数的结果是，客户端会调用*create_param_form* 方法，基于 index.html 文件中定义的 html 模板，创建一个用于编辑报表参数的表单（参见表单窗体 ）。

此方法在创建表单后，会触发以下事件：

- 任务的*on_param_form_created*
- 拥有该报表的报表组的*on_param_form_created* （如果已定义）
- 报表的*on_param_form_created* （如果已定义）

默认代码中有一个为任务定义的*on_param_form_created* 事件处理程序。在此事件中，点击 **打印 (Print)** 按钮会调用报表的*process_report* 方法。

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

接着，会触发*process_report* 方法：

- 报表组的*on_before_print_report* 事件处理程序
- 报表的*on_before_print_report* 事件处理程序

在这些事件处理程序中，开发人员可以定义报表的一些通用（报表组事件处理程序）或特定（报表事件处理程序）属性。

例如，在默认代码中，有一个报表组的 on_before_print_report 事件处理程序，其中定义了报表的*extension* 属性：

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

在以下事件处理程序中，在演示应用程序的 **发票 (invoice)** 报表的客户端模块中定义的处理程序，设置了报表 **id** 参数的值：

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

之后，*process_report* 方法会向服务器发送异步请求以生成报表（参见服务器端编程 ）。

服务器会向该方法返回一个指向已生成的报表文件的 url 。

然后，该方法检查是否定义了报表组的*on_open_report* 事件处理程序。如果定义了此事件处理程序，则调用它；否则检查报表的*on_open_report*。如果已定义，则调用它。

如果这些事件都未定义，则根据报表的*extension* 属性，在浏览器中打开报表或将其保存到磁盘。

3.7.5 服务器端报表编程

当服务器收到客户端生成报表的请求时，它首先创建报表的副本，然后该副本调用*generate* 方法。

此方法会触发*on_before_generate* 事件。在此事件处理程序中，开发人员应编写生成报表内容的代码。

例如，演示应用程序的 **发票 (invoice)** 报表的事件处理程序如下：

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

    customer = invoices.firstname.display_text + ' ' + invoices.customer.display_text
    address = invoices.billing_address.display_text
    city = invoices.billing_city.display_text + ' ' + invoices.billing_state.display_text + ' '␣
↪+ \
        invoices.billing_country.display_text
    date = invoices.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
```

(续下页)

(接上页)

```
    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())
```

首先，我们使用*copy* 方法创建 **发票 (invoices)** 业务项的副本。

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

我们创建副本是因为多个用户可以在并行线程中同时生成相同的报表。

然后，我们调用副本的 set_where 方法：

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

其中 report.id.value 是报表的 id 参数，我们在客户端的*on_before_print_report* 事件处理程序中设置了其值，该值等于当前 **发票 (invoice)** 的 **id** 字段的值。

然后，我们使用*open* 方法在服务器上获取记录。之后，使用*print_band* 方法打印标题栏：

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

但在此之前，我们为四个局部变量赋值：customer（客户）、address（地址）、city（城市）和 date（日期），这些变量对应报表模板中标题区段的可编程单元格。

然后，以相同的方式生成明细区段和汇总区段。

报表生成后，如果客户端设置的报表*extension* 属性值不等于'pdf'，服务器会使用 **LibreOffice** 将其转换 ods 文件。

报表生成后，它将存储在 static 目录的 report 文件夹中，服务器会向客户端发送报表文件的 url。

3.8 保留字

由于 Jam.py 是 Python 和 JavaScript 语言框架，因此应使用有效的 Python 和 JavaScript 标识符。此外，对于所有支持的数据库，应使用它们各自的有效标识符。

包含示例列表的文件是 `keywords.py`，其内容如下：

```
keywords = [
        "BADGE",
        "LABEL",
        "HIDDEN",
        "PROGRESS",
        "TASK"
]
```

CHAPTER 4

Jam.py 常见问题

4.1 外键的作用是什么？

当查找字段中保存了对查找表内某条记录的引用时，在 Jam.py V5 应用构建器中创建的外键会阻止删除该记录。

举个例子：若为“发票 (Invoices)”项中的“联系人 (Customer)”字段创建了外键，只要“发票 (Invoices)”中存在对某客户的引用，用户就无法在“客户 (Customers)”主数据表中删除该客户。

若要让查找字段显示在外键对话框 中，需将查找实体项的软删除属性设为 false （详见实体项编辑器对话框 ）。

Jam.py V7 已取消对外键的支持。

4.2 主表目录和业务台账的区别

新建项目时，其任务树 包含以下分组：**主表目录 (Catalogs)** 、**业务台账 (Journals)** 、**明细表 (Details)** 和 **报表 (Reports)** 。

主表目录 (Catalogs) 与 **业务台账 (Journals)** 均属于实体项分组类型，功能用途相同。详见分组 。

我们设置这两类分组，是为了区分两种不同类型的数据实体：

- **主表目录 (Catalogs)** —包含各类相对静态不变的基础数据信息。例如，客户 (customers)、组织 (organizations)、曲目 (tracks) 等
- **业务台账 (Journals)** —包含各类业务动态事件生成的数据信息。例如，发票 (invoices)、采购订单 (purchase orders) 等。

4.3 如何将已创建的项目升级到新版本的 Jam.py？

要将现有的 V7 项目升级到新包，您必须更新该包。

您可以使用 pip 来完成此操作。

如果您使用的是 Linux、Mac OS X 或其他 Unix 系统，请输入以下命令：

```
sudo pip install --upgrade jam.py-v7
```

如果您使用的是 Windows，请以管理员权限启动 shell 命令行窗口并运行以下命令：

```
pip install --upgrade jam.py-v7
```

要将 v5 项目迁移到 v7 项目，请参阅*如何将 v5 项目迁移到 v7*

4.4 我可以在应用程序中使用其他库吗？

您可以添加 JavaScript 库，以在客户端编程中使用它们。

最好将它们放在项目中 *static* 目录的 *js* 文件夹下。并在*Index.html* 文件的 <script> 标签中使用 src 属性引用它们。

例如，*演示项目* 使用 Chart.js 库来创建数据仪表盘：

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

在服务器端，您可以将 Python 库导入到您已有的模块中。

例如，在邮件项目的服务器模块中导入 smtplib 库来发送电子邮件：

```
import smtplib
```

4.5 打印报表时我得到的是 ods 文件而不是 pdf

当生成报表时，服务端应用程序首先创建一个 ods 文件。

如果报表的*extension* 属性被设置为“pdf”或除“ods”之外的任何其他格式，应用程序会首先创建一个 ods 文件，然后使用“无头 (headless)”模式的 LibreOffice 将 ods 文件转换为该格式。

如果 LibreOffice 正在服务器上运行，这种转换就可能不会发生。您必须关闭服务器上的 LibreOffice 才能进行转换。

CHAPTER 5

如何...

这里有一些实用代码，你可以在自己的应用程序中直接使用：

5.1 如何在 Windows 上安装 Jam.py

改编自 Django Docs

以下文档参考了 Django Docs.

本文档将指导您在 Windows 上安装 Python 3.x 和 Jam.py。它还提供了设置虚拟环境的说明，这使得处理 Python 项目更加容易。这只是为从事 Jam.py 项目开发的用户准备的初学者指南，而不是在开发 Jam.py 本身的补丁时应如何安装 Jam.py 的说明。

本指南中的步骤已在 Windows 10 上测试。在其他版本中，步骤会类似。您需要熟悉使用 Windows 命令提示符。

5.1.1 安装 Python

Jam.py 是一个 Python Web 框架，因此需要在您的机器上先安装 Python 。在撰写本文时，Python 3.8 是最新版本。

要在您的机器上安装 Python，请访问 https://www.python.org/downloads/。该网站应该为您提供最新 Python 版本的下载按钮。下载可执行安装程序并运行它。选中 "Install launcher for all users (recommended)" 旁边的复选框，然后点击 "Install Now"。

安装后，打开命令提示符并通过执行以下命令检查 Python 版本是否与您安装的版本匹配

```
...\> python --version
```

5.1.2 关于 pip

pip 是 Python 的包管理器，默认已包含在 Python 安装程序中。它有助于安装和卸载 Python 包（如 Jam.py!）。在后续的安装过程中，我们将使用 pip 从命令行安装 Python 包。

5.1.3 设置虚拟环境

为您创建的每个 Jam.py 项目提供专用环境是最佳实践。Python 生态系统中有许多管理环境和包的选项，其中一些在 Python documentation 中被推荐。

要为您的项目创建虚拟环境，请打开一个新的命令提示符，将当前目录切换到您要创建项目的文件夹，然后输入以下命令

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

这将创建一个名为"project-name"的文件夹（如果它尚不存在）并设置虚拟环境内容。要激活环境，请运行

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

虚拟环境被激活后，您会在命令提示符旁边看到"(project-name)"以示区别正常环境。每次启动新的命令提示符时，您都需要再次激活虚拟环境。

5.1.4 安装 Jam.py

可以在虚拟环境中使用 pip 轻松安装 Jam.py 。

在命令提示符中，确保您的虚拟环境已激活，然后执行以下命

```
...\> python -m pip install jam.py-v7
```

这将下载并安装最新版本的 Jam.py 。

对于 Python 3.13 及以上版本，请同时安装

```
...\> python -m pip install standard-imghdr
```

安装完成后，您可以通过在命令提示符中执行 pip list 来验证您的 Jam.py 安装。

现在我们可以继续创建一个新项目 。

5.1.5 常见误区

- 如果您使用代理连接到互联网，运行命令 py -m pip install Jam.py 时可能会出现问题。解决此问题，可以在命令提示符中为代理配置设置环境变量，如下所示

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

- 如果您的管理员禁止设置虚拟环境，您仍然可以按以下方式安装 Jam.py

  ```
  ...\> python -m pip install jam.py-v7
  ```

  这将下载并安装最新版本的 Jam.py 。

  安装完成后，您可以通过在命令提示符中执行 pip list 来验证您的 Jam.py 安装。

  但是，运行 jam-project.py 可能会失败，因为它不在当前路径中。

  检查安装文件夹，请运行

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

输出可能类似于以下内容

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

将上面一行末尾的 `site-packages` 替换为 `Scripts`，运行以下命令

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

输出内容可能类似于

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

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

在某处创建新文件夹并从中运行 `jam-project`

```
...\> python C:\Users\youruser\AppData\Local\Packages\PythonSoftwareFoundation.Python.3.9_
↪qbz5n2kfra8p0\LocalCache\local-packages\Python39\Scripts\jam-project.py
```

运行新项目

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

5.1.6 安装 WLS

快速启用 WSL 并安装默认的 Linux 发行版（通常为 Ubuntu）

```
...\> wsl --install
```

输出可能类似于以下内容

```
Installing: Virtual Machine Platform
Virtual Machine Platform has been installed.
Installing: Windows Subsystem for Linux
Windows Subsystem for Linux has been installed.
Installing: Ubuntu
Ubuntu has been installed.
The requested operation is successful. Changes will not be effective until the system is␣
↪rebooted.
```

现在，我们有了一个 Ubuntu 系统下的开发环境，我们可以像在 Linux 上一样继续进行 Jam.py 安装。

5.2 如何实现多对多关系？

多对多关系在演示应用程序中通过业务台账下的 **发票 (Invoices)** 实现。每位客户可购买多首曲目，每首曲目可出售给多位客户。无需代码即可实现此功能。

Jam.py 提供的远不止于此。如“曲目 (Tracks)”表单所示，它还支持自动汇总，识别每位客户购买的具体曲目。需要编写一些代码来实现查找客户的功能。

5.3 如何从开发环境迁移到生产环境

由于 Jam.py 具有导出和导入元数据的能力，因此，将已完成的项目从开发环境迁移到生产环境非常简单。

要了解元数据的概念以及导出和导入元数据的过程，请阅读主题 导出/导入元数据。导入元数据的过程取决于项目数据库的类型。

5.3.1 迁移到新项目

- 先在生产环境中创建一个空数据库
- 运行脚本 *jam-project.py* ，创建一个新项目
- 设置服务器，请参考
  - *使用 Apache 和 mod_wsgi 部署 Jam.py*
  - *如何部署*
- 在浏览器中启动应用程序构建器，并使用空数据库完成项目的创建。
- 打开项目参数 对话框来设置项目。设置以下参数：
  - **生产环境 (Production)** 设为 true （勾选复选框）
  - **安全模式 (Safe mode)** 设为 true（勾选复选框）
  - **调试 (Debugging)** 设为 false（不勾选复选框）
- 在应用程序构建器中，通过点击导出 按钮，将开发环境中的项目的元数据导出为一个 zip 文件。
- 将元数据导入到新项目中

> **备注**
>
> 对于使用 **SQLite** 数据库的项目，您可以简单地将开发环境下的项目文件夹复制到生产环境中。

5.3.2 迁移到已有项目

- 将开发环境中的项目的元数据导出为一个 zip 文件。
- 将元数据导入到生产环境下的项目中。

> **备注**
>
> 对于 **SQLite** 数据库，Jam.py 不支持将元数据导入到现有项目（数据库中已有表的项目）中。您只能将元数据导入到新项目中。

5.3.3 关闭 HTTP 服务器进程时导入元数据

Stop the http server and copy the metadata zip file to *migration* folder in the project directory. If the folder doesn't exist, create it. 停止 HTTP 服务器并将元数据 zip 文件复制到项目目录中的 *migration* 文件夹。如果该文件夹不存在，请创建它。

启动 HTTP 服务器。Web 应用程序在初始化时会导入元数据文件。您可以在项目目录的 *logs* 文件夹下的日志文件中看到文件被导入的信息。如果导入成功，zip 文件将被删除。

5.3.4 不关闭 HTTP 服务器进程时导入元数据

点击应用程序构建器中的导入 按钮。

> **备注**
>
> 默认情况下，在导入元数据是，进程中的 Web 应用程序会等待 5 分钟，或直到 **此进程中**对应用程序的所有先前请求都被处理完毕，然后才开始更改数据库。
>
> 对于在多个进程上运行的项目，您可以在项目参数 中设置 **延迟导入 (Import delay)** 参数来延迟数据库更改，或者使用在关闭服务器时导入元数据的方法。

5.4 如何迁移到另一个数据库

待正式公布 - 从 Jam.py v5 起被更改。

您可以将数据迁移到另一个数据库。

例如，您使用 SQLite 数据库开发了项目，现在想要把数据库迁移到 Postgres 。

要执行此操作，请按照以下步骤操作：

1. 创建一个空的 Postgres 数据库
2. 使用此数据库创建一个新项目
3. 在 SQLite 项目的应用程序构建器中，通过点击导出 按钮，将元数据导出为一个 zip 文件。
4. 将元数据导入到新项目中。Web 应用程序将在 Postgres 数据库中创建数据库结构。
5. 使用任务的*copy_database* 方法将数据从 SQLite 复制到 Postgres 数据库：
   - 在 **项目\任务 (Project\Task)** 中创建以下服务器模块函数（调整下面的数据库路径为正确的路径）：

     ```
     from jam.db.db_modules import SQLITE

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

   - 然后，通过以下方式之一执行它：
     - 在*on_created* 事件处理程序中调用此函数：

       ```
       from jam.db.db_modules import SQLITE

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

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

     - 在某个表单中创建一个按钮，并使用任务的*server* 方法执行它：

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

– 或从浏览器的调试控制台运行：

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

6. 在此过程完成后立即删除使用的代码。

> **备注**
>
> 如果当前数据库有外键，则不能迁移到 SQLite 数据库。

5.5 如何部署

5.5.1 如何在 PythonAnywhere 上部署项目

- 使用 pip 安装 Jam.py。为此，请打开一个新的 **Bash** 命令行并运行：

```
mkvirtualenv --python=/usr/bin/python3.13 my-virtualenv  # use whichever python version↵
↪you prefer
pip install jam.py-v7
```

  上面的 **mkvirtualenv** 命令可能显示如下：

```
created virtual environment CPython3.10.5.final.0-64 in 8486ms
creator CPython3Posix(dest=/home/username/.virtualenvs/my-virtualenv, ...)
```

  dest 对应的路径将用于 **Virtualenv** 部分，这对于每个用户，各不相同。

- 将你的项目文件夹压缩为一个 zip 文件，然后在 **文件 (Files)** 选项卡将其上传并解压：

  假设你有一个 *username* 的用户，你的项目就会位于 */home/username/project_folder* 目录下。

- 打开 **Web** 选项卡，添加一个新的 web 应用程序。在 **代码 (Code)** 部分指定：
  – 源代码 (Source code): */home/username/project_folder*
  – 工作目录 (Working directory): */home/username/project_folder*
  – WSGI 配置文件: 打开 */var/www/username_pythonanywhere_com_wsgi.py* 文件, 删除已有内容，并只添加以下代码：

```
import os
import sys

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

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

  在 **静态文件 (Static files)** 部分，指定 Jam.py 的静态文件的访问位置和路径，例如：

  - URL: */static/*
  - 目录 (Directiory): */home/username/project_folder/static*

  在 **Virtualenv** 部分，指定使用上面 **mkvirtualenv** 命令创建的虚拟环境路径。例如：

  - /home/username/.virtualenvs/my-virtualenv/

  在 **强制 HTTPS(Force HTTPS)** 部分，启用 HTTPS。

- 在 **重启 (Reload)** 部分，重启服务器。
- 要进行调试，请在 **日志 (Logs)** 部分查看日志内容。

5.5.2 在 AWS 上部署 Jam.py 的逐步指南

本文改编自 https://devops.profitbricks.com/tutorials/install-and-configure-mod_wsgi-on-ubuntu-1604-1/

希望对大家有所帮助。

- 创建一个 AWS 账户并登录
- 进入 EC2，创建一个实例（本例中使用 Ubuntu 16.04 t2.micro）
- 按提示下载私钥
- 使用 Puttygen 将 pem 转换为 ppk (参加: https://stackoverflow.com/questions/3190667/convert-pem-to-ppk-file-format)
- 从 AWS 控制台获取 EC2 实例的公共 DNS
- 使用 Putty SSH 登录到 EC2 实例（指向公共 DNS 和您的 ppk）
- 用户名为 ubuntu
- 更新软件包信息库：

```
sudo apt-get update
```

- 安装 pip:

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

- 安装 jam.py:

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

- 安装 Apache:

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

- 安装 mod-wsgi:

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

- 重启 Apache:

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

- 切换到此处:

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

- 创建目录:

```
sudo mkdir [appname]
```

- 切换到此处:

```
cd [appname]
```

- 创建应用:

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

- 检查应用是否存在:

```
ls
```

- 创建配置:

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

- 粘贴以下内容:

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

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

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

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

- 退出并保存
- 给 apache 文件权限:

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

- 给 apache 所有权:

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

- 启用 wsgi:

```
sudo a2enconf wsgi
```

- 重启 apache:

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

- 在 AWS 上创建安全组，允许您在端口 80 上连接 HTTP
- 将实例分配到安全组

- 测试
- 如果不工作，检查错误日志以了解情况：

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

本文最初由 *Simon Cox* 发布于 https://groups.google.com/forum/#!msg/jam-py/Zv5JfkLRFy4/22tolZ-hAQAJ

5.5.3 如何在 Linux Apache HTTP 服务器上部署 Jam-py 应用？

抱歉，因为我对 MS 服务器完全不了解，因此，这里基本说的是直接部署到云服务器上，它们通常开放 22、80 和 443 端口。前提是拥有 DNS 服务器名称的签名证书（即，下面的 YOUR_SERVER DNS ）。也可以使用自签名证书等，这里不涵盖这些内容。此外，还需要安装 Python 和授予 sudo 访问权限（或 Linux 的 root 权限）。

应用处于只读模式。您可以访问 admin.html 页面，但无法更改任何内容。我在 Google Cloud 服务器上进行了一些调整，这是一个微型 Ubuntu 实例，使用 apt-get 进行普通的 apache2 安装。

- 为 Apache 安装 wsgi 模块:

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

- 为 apache 启用 ssl 和 wsgi 模块:

```
a2enmod ssl wsgi
```

- 为 jam-py 应用创建自定义配置文件，例如 /etc/apache2/sites-available/test.conf（仍在完善中）：

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

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

    DocumentRoot /var/www/html/simpleassets

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

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

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

```

(续下页)

(接上页)

```
      Order deny,allow
      Allow from all
      Require all granted

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

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

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

上面的文件使用带有 your.key 的签名证书 your.crt ，以及从 CA 获取的 CA 和链文件。请查看网络上关于证书和 DNS 的资源。您需要获取并将这些文件复制到 /etc/ssl/private 文件夹中。将 YOUR_xyz 更改为您的自己的内容。

/var/www/html 是 Ubuntu 用于提供网页的默认文件夹。

- 像往常一样安装 jam-py

  我创建了 /var/www/html/simpleassets 文件夹，在那里解压了 jam-py SimpleAssets 项目。按照这里解释的步骤来部署这些：

  简单来说，导出您的项目，保存好 zip 文件，并将其复制到您的网络托管服务器的所需文件夹中。同时复制 admin.sqlite 和您的数据库（假设您使用的是 sqlite3 数据库）。如果使用其他数据库，例如 mysql，您需要导出/导入数据库。

- 启用 test.conf (上面没有扩展名的文件名):

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

  就是这样。目前，我保留了 80 端口不变，jam-py 只在 https 端口上运行。要调试问题，我会从 SeLinux 或 apparmor 开始。在 Ubuntu 上，这可能会有所帮助：

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

现在，问题是如何在一个 https 服务器上运行两个 jam-py 实例？

这个问题的一个可能答案是 DNS。您可以将您的 DNS 设置为例如 second_instance.YOUR_SERVER 名称（上面的实际例子是 jam2.research...）。

因此，上面的 test.conf 文件几乎相同，只是 YOUR_SERVER 被称为 second_instance.YOUR_SERVER。

/etc/apache2/sites-available/test3.conf 文件内容如下：

```
<IfModule mod_ssl.c>
  <VirtualHost YOUR_IP:443>
    ServerName second_instance.YOUR_SERVER
    ServerAlias
    ServerAdmin YOUR_EMAIL
    ErrorLog ${APACHE_LOG_DIR}/test3-error-sec.log
```

(续下页)

(接上页)

```
    CustomLog ${APACHE_LOG_DIR}/test3-access-sec.log combined
    #below is for cx_Oracle
    SetEnv LD_LIBRARY_PATH /u01/app/oracle/product/11.2.0/xe/lib
    SetEnv ORACLE_SID XE
    SetEnv ORACLE_HOME /u01/app/oracle/product/11.2.0/xe
    #finish cx_Oracle
    DocumentRoot /var/www/html/simpleassets3
    SSLEngine on
    SSLCertificateFile  "/etc/ssl/private/your.crt"
    SSLCertificateKeyFile  "/etc/ssl/private/your.key"
    SSLCertificateChainFile "/etc/ssl/private/your_chain.crt"
    SSLCACertificateFile   "/etc/ssl/private/your_CA.crt"

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

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

      Order deny,allow
      Allow from all
      Require all granted

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

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

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

Jam-py 应用的 second_instance 现在位于例如 /var/www/html/simpleassets3 中，并且 WSGIDaemonProcess 已调整为新的守护进程，称为 assets3。其他一切几乎相同。

这是可能的，因为 SSL 证书是*（星号或通配符）证书，使您能够在一个 DNS 域上运行多个服务。

本文最初由 *Dražen Babić* 发布于 https://github.com/jam-py/jam-py/issues/35

5.5.4 Nginx 配合 Gunicorn 或 uvicorn 的使用，如何实现？

Green Unicorn (gunicorn) 是一个 HTTP/WSGI 服务器，设计用于为快速客户端或响应较慢的应用程序提供服务。也就是说，它通常在缓冲前端服务器（如 nginx 或 lighttpd）的后面运行。

默认情况下，`gunicorn` 会监听 127.0.0.1。切换到 jam 应用文件夹，或使用（例如，在脚本、cron 作业中）如下代码：

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

或从 /path/to/jam/App 目录运行:

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

要在所有接口和端口 8081 上启动 jam.python :

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

如果需要，可以使用 --workers=5 启动 5 个工作进程。

对于 `uvicorn` ，我们需要修改 wsgi.py 文件并安装 `asgiref` 。

我们将其命名为 asgi.py 文件，内容如下:

```
from jam.wsgi import create_application
from asgiref.wsgi import WsgiToAsgi
application = WsgiToAsgi(create_application(__file__))
```

要在 `localhost` 和端口 8000 上启动 jam.py :

```
uvicorn asgi:application
INFO:     Started server process [16576]
INFO:     Waiting for application startup.
INFO:     ASGI 'lifespan' protocol appears unsupported.
INFO:     Application startup complete.
INFO:     Uvicorn running on http://127.0.0.1:8000 (Press CTRL+C to quit)
```

Nginx 配置:

在 /etc/nginx/sites-enabled/default (Linux Mint) 中注释掉默认的 location 部分：:

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

并添加:

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

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

(续下页)

(接上页)

```
    proxy_set_header   X-Forwarded-For $proxy_add_x_forwarded_for;
    proxy_set_header   X-Forwarded-Host $server_name;

 }
```

在 /etc/nginx/nginx.conf 中添加 127.0.0.1:8081，如果这是你的 Gunicorn 或 uvicorn 服务器地址和端口:

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

这也使得在不同端口上运行不同的应用服务器成为可能:

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

重启 nginx 即可!

恭喜! 我们现在可以测试 Nginx 与 Jam.py 的配合使用了。

现在，关于 certificates:

在 /etc/nginx/sites-enabled/jam 中，我们可以添加如下配置，将所有 HTTP 请求重定向到 HTTPS 并转发到 8001 端口（或根据上述设置的任何其他端口）:

```
server {
      listen 80;
      server_name YOUR_SERVER;

      access_log off;

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

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

      return 301 https://$server_name$request_uri;
}
server {
      listen 443;
      server_name YOUR_SERVER_FQDN;

      access_log off;

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

(续下页)

(接上页)

```
        }

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


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

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

就是这样！

恭喜！我们现在可以在 HTTPS 端口上测试 Nginx 与 Jam.py 的配合使用了！

本文最初由 *Dražen Babić* 发布于 https://github.com/jam-py/jam-py/issues/67

5.6 如何编写具有全局作用域的函数？

在实体元素的服务端或客户端模块中定义的每个函数，都会成为该元素的一个属性。

因此，借助*任务树*，你可以在项目的任意模块中，访问在客户端或服务端模块内声明的任意函数。

例如，若我们在 **客户 (Customers)** 的客户端模块中声明了一个函数 `some_func`，就可以在项目的任意模块中调用它。需要注意的是，**task** 在客户端中是一个全局变量。

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

在服务端中，**task** 不是全局变量，但触发/调用它的实体项会被传入至每个由*server* 方法调用的事件处理程序和函数中。因此，如果 `some_func` 函数是在 Customers 的服务端模块中声明的，在函数或事件处理程序中就可以按如下方式调用：

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

需要注意，事件处理程序本质就是函数，同样也可以由其他模块调用。

5.7 如何验证字段的值

编写*on_field_validate* 事件处理程序来验证字段的值。

例如，当调用*post* 方法将记录保存时，会触发事件；当用户离开用于编辑单价字段值的输入框时，也会触发事件。

```
function on_field_validate(field) {
     if (field.field_name === 'unitprice' && field.value <= 0) {
```

(续下页)

(接上页)

```
        return 'Unit price must be greater that 0';
    }
}
```

举个例子，以下是不使用*on_field_validate* 方法的代码，它们检查单价字段的值，防止用户输入小于或等于零的值：

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

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

在*on_edit_form_shown* 事件处理程序中，使用*each_field* 方法来迭代访问所有字段，并找到每个字段对应的输入框（如果存在）。每个输入框都有一个包含字段名称的类（*field_name* ）。

然后，我们为它分配一个 jQuery blur 事件，在该事件中我们调用 `check_field_value` 函数，如果它返回包含错误信息的文本字符串，我们会警告用户并将焦点定位到输入框。在调用函数之前，我们检查是否按下了“取消”按钮。

我在实体的模块中定义的*on_edit_form_shown* 事件处理程序，所以只在定义它的模块中有效。

我们可以在任务客户端模块中声明以下事件处理程序，以便在我们需要启用此字段验证的任何模块中编写 `check_field_value` 函数。当编辑表单显示后，会首先为每个实体调用任务的

on_edit_form_shown 事件处理程序。参考表单窗体事件 。

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

(续下页)

(接上页)

```
                }
            });
        });
    }
}
```

在这个事件处理程序中，我们检查实体是否有 check_field_value 属性。在模块中声明的每个函数，都会成为其所属实体的一个属性。

5.8 如何给表单添加按钮

为编辑/视图表单添加按钮的最简单方法是使用对应的*add_edit_button* / *add_view_button* 方法。你可以在*on_edit_form_created* / *on_view_form_created* 事件处理程序中调用这些添加按钮的方法。

例如，客户 (Customers) 实体项使用这些代码，在客户端模块为一个视图表单添加了按钮：

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

在代码中，检查了实体的*lookup_field* 属性是否存在，然后（创建视图表单不是为了为查找字段选择值时）定义了两个按钮，并为它们的 JQuery 点击事件分配了模块中的 send_email 和 print 函数。

5.9 如何从客户端执行 Python 代码

虽然在操作系统层面通过 Popen 命令执行任意 Python 脚本是可行的，但我们首先演示 **服务端模块 [F8]** 代码的使用方法。

你可以调用*server* 方法向服务端发送请求，以执行在实体项的服务端模块中定义的函数。

在下方示例中，我们创建了一个 jQuery 对象类型的 btn 按钮，随后通过其点击事件绑定一个函数，该函数会调用实体项的*server* 方法，运行在实体项的服务端模块中定义的 calculate 函数。

客户端模块中的代码如下：

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

在服务端中的代码如下:

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

要执行操作系统脚本，我们可以使用带有 Popen 和类似于上面的按钮的服务器端模块代码:

```
build = Popen([make, 'html'] , cwd=build_path, stderr=STDOUT,stdout = PIPE, shell=shell)
result, err = build.communicate()
result = result.decode("utf-8")
```

要查看构建结果，我们可以使用带有按钮的 JavaScript 模态表单来显示它:

```
item.edit_form.find("#build-info-btn").hide().click(function() {
    show_build_info(item);
});

function show_build_info(item) {
    var i = 0,
        color,
        html = '<p>',
        info = item.build_result.split('\n');

    for (i = 0; i < info.length; i++) {
        color = '#333333';
        if (build_problems(item, info[i])) {
            color = 'red';
        }
        html += '<span style="color: ' + color + ';">' + info[i] + '</span><br>';
    }
    html += '</p>';
    html = $(html).css("margin", 20);
    task.message(html, {width: 700, height: 600,
        title: 'Build information', footer: false, print: true});
}
```

在这个例子中，用 Sphinx 的 make 命令来构建 Jam.py 的文档。

5.10 如何更改表单中元素的样式和属性

使用 jQuery ，能够访问表单上的任意 DOM 元素。

在下面的示例中，在实体项的客户端模块的 `on_edit_form_created` 事件处理程序中，我们找到 **确定 (OK)** 按钮，将其隐藏，并将编辑表单中 **取消 (Cancel)** 按钮的文本更改为 “关闭 (Close)”：

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

当应用程序创建输入控件 (input) 时，它会向每个输入控件 (input) 添加一个类 (class)，该类的名称是相应字段的*field_name* 属性值。

因此，使用 jQuery 选择器 ，我们就能向下面那样找到 customer 字段对应的输入控件 (input) （我们使用 “customer” 样式类选择在编辑表单里的输入控件元素）：

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

找到表单的元素后，就能使用 jQuery 方法对其更改。

由于字段的输入框是在*on_edit_form_created* 事件触发后，由*create_inputs* 方法自动创建的（可参考任务客户端模块中的 `on_edit_form_created` 事件处理器），因此你必须编写*on_edit_form_shown* 事件处理器来修改输入框样式和属性。

示例代码

```
.. code-block:: js
```

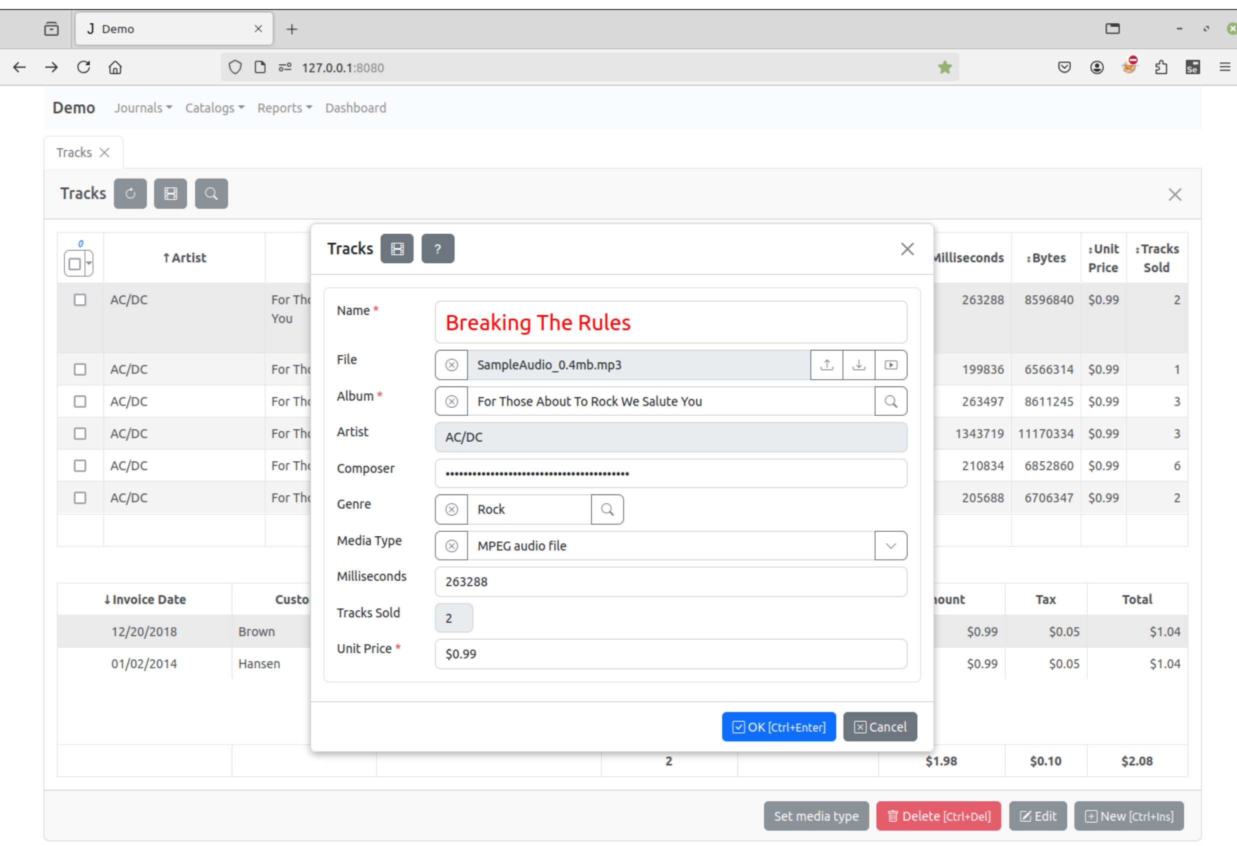
function on_edit_form_shown(item) {

item.edit_form.find('input.name').css('color', 'red'); item.edit_form.find('input.name').css('font-size',

'24px'); item.edit_form.find('input.tracks_sold').width(20); item.edit_form.find('input.genre').parent().width('40%'); item.edit_form.find('input.composer').prop('type', 'password');

}

将像下面一样修改表单的输入框:



请注意：若你需要修改带有前置按钮或后置按钮的输入框（日期、日期时间、查询字段的输入框）的宽度，请直接设置该输入框父元素的宽度：

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

更改 DOM 元素样式的另一种方法是使用 CSS。在应用程序构建器中选择任务节点后，"project.css" 按钮将出现在右侧窗格中。单击它，打开位于项目文件夹中的 *project.css* 文件。您可以使用在它里面输入定义项目中 DOM 元素样式的 CSS 。

项目中创建的每个实体项的表单都有 css 类，使开发人员能够识别定位表单。

每个表单都有一个类来标识它的类型："view-form"、"edit-form"、"filter-form" 或 "param-form"。

例如，以下代码将隐藏表单底部按钮中的图像：

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

更多编辑表单的例子：

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

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

同样，每个表单有个样式类，其名称是对应实体项的*item_name* 属性值。

The following code will remove images in the buttons only in the **Invoices** view form: 下面的代码将只隐藏 **发票 (Invoices)** 表单中按钮的图像

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

你可以更改表格的显示方式。由*create_table* 方法创建的表格，有一个"dbtable"样式和一个以实体项的*item_name* 属性值命名的样式。表格的每列也有一个样式，其名称是对应字段的*field_name* 属性值。

示例中，以下代码将以粗体显示 **Invoices** 表 **Customer** 列中单元格的内容：

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

另一种改变字段列显示方式的方法是编写*on_field_get_html* 事件处理程序。

例如：

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

5.11 如何创建自定义菜单

要创建一个自定义菜单，你必须在 任务的客户端模块中为 任务的*create_menu* 方法指定 custom_menu 选项的值。

5.12 如何在不打开视图表单的情况下使用编辑表单追加记录？

首先，你必须调用实体项的*open* 方法来初始化实体项使用的数据集。例如，按照下面的做法，你可以向 演示应用程序中的 invoices 添加一条新记录。

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

在这段代码中，我们使用实体项的*copy* 方法创建了它的一个副本。因此，即使已在一个选项卡中打开了 Invoices 的视图表单，这里的操作也不会影响到它。

您也可以更改记录，但在执行此操作之前，您必须从服务器获取它。下面是修改 id 为 411 的记录的代码。我们使用 rec_count 属性检查记录是否存在，若不存在则显示警告信息。

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

在上面的示例中，open 方法不是异步执行的。

下面的代码是异步执行的：

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

发票 *(Invoices)* 的编辑表单对话框已设置无模式（Modeless）属性，因此编辑表单会在标签页中打开。你可以通过设置*edit_options* 的无模式属性来修改它，使编辑表单变为模态对话框（模态窗口）：

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

5.13 如何禁止更改记录

假设我们有一个的数据表，它含有一个 boolean 类型的字段 "posted"。如果该字段的值是 真 *(true)* ，就必须禁止或删除记录。

我们可以通过编写*on_after_scroll* 事件处理程序，同时结合使用*permissions* 的属性来实现。

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

在上面的事件处理程序里，我们检查 "posted" 字段的值，然后设置*permissions* 的属性来实现。

我们也可以在服务端的模块中编写*on_apply* 事件处理程序来实现同样的目的：

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

5.14 如何关联两个数据表

以下步骤适用于 Jam.py V5 版本，适用于两个数据库表未在构建器中通过 **主/从关系 (Master/Detail)** 直接关联的场景。(参见教程第三部分：明细表)

在 Jam.py V7 版本中，演示应用程序的 曲目 *(Tracks)* 数据表已与 发票 *(invoice)* 明细表直接关联，因此无需执行以下步骤。

若数据表未在构建器中直接关联，在 Jam.py V7 中仍可使用以下步骤。

我们将以演示应用中的 曲目 *(Tracks)* 和 发票 *(invoice)* 实体项为例，说明如何关联两个实体项。我们会将 曲目 *(Tracks)* 表中的记录，与 发票 *(invoice)* 表中对应的已售出曲目列表关联（销售 *(invoice)* 表存储了所有已售出的曲目）。

在任务的客户端模块中声明的*on_view_form_created* 事件处理程序内定义*view_form* 的默认行为。

我们将在曲目的客户端模块的*on_view_form_created* 事件处理程序里修改其默认行为：

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

然后，我们将显示曲目数据的表格的高度减少 200 像素。

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

使用*copy* 方法创建 invoice_table 的一个副本，再将副本的*paginate* 属性设置为 false，并调用副本的*create_table* 方法来创建一个表格，用来显示已售出的曲目。

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

我们为表格设置了 200 像素的高度，并定义了汇总的字段。

如果我们不定义下面的*on_after_scroll* 事件处理程序，这个表将总是空的：

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

只要当前记录发生更改，就会触发*on_after_scroll* 事件。所以当曲目数据改变时，我们调用*open* 方法，预先设置过滤器和排序依据：

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

此方法向服务器发送请求，服务器生成 sql 查询，执行该查询并返回一个数据集，该数据集包含按 invoice_date 字段降序排列的此曲目的已售出记录。

如果曲目数据集是空的，将通过调用*close* 方法来清除已发票的数据集。

由于 Jam.py 中的控件都是数据感知型的，已售曲目数据集的每一次变更，都会自动显示在我们在*on_view_form_created* 事件处理程序中创建的表格中。

现在，每当曲目记录发生变化时，应用程序都会向服务器发送请求以刷新已售曲目列表。这种方式效率不高，有时还会导致延迟。为了解决这个问题，我们使用 JavaScript 的 setTimeout 函数：

```
var scroll_timeout;

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

该函数保证了数据的更新频率不会一次超过 100 毫秒。

由于 invoice_table 是一个明细表，它包含 **invoice** 字段，该字段存储了与当前记录相关的发票引用，因此我们可以向用户展示包含当前已售曲目记录的发票。为此，我们在调用*create_table* 方法时，传入一个函数，当用户双击表格中的记录时会执行该函数：

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

传入的函数的定义如下：

```
function show_invoice(invoice_table) {
    var invoices = task.invoices.copy();
    invoices.set_where({id: invoice_table.invoice.value});
    invoices.open(function(i) {
```

(续下页)

(接上页)

```
        i.edit_options.modeless = false;
        i.can_modify = false;
        i.invoice_table.on_after_open = function(t) {
            t.locate('id', invoice_table.id.value);
        };
        i.edit_record();
    });
}
```

在这个函数里，我们创建了 发票台账 *(invoices journal)* 的一个副本，并查找指定 id 的发票。当执行 open 方法时，我们将通过调用它的*edit_record* 方法来显示发票内容。但是，在调用前，我们对其属性进行设置，这样它将以模态形式显示，而且用户也不能对其修改。

此外，我们还会动态地为获取到的发票的 invoice_table 明细分配*on_after_open* 事件处理程序。在该事件处理程序中，我们通过调用*locate* 方法，在已售曲目记录中定位到当前记录。

最后，我们还会检查 曲目 *(tracks)* 的*lookup_field* 属性。当该属性为 true 时，表示该项是为选择查找字段的值而创建的（即用户点击查找字段输入框右侧按钮时创建）。我们会确保当用户为查找字段选择值时，不显示已售曲目的记录。

此外，我们添加了一个提醒，告知用户可以到发票。

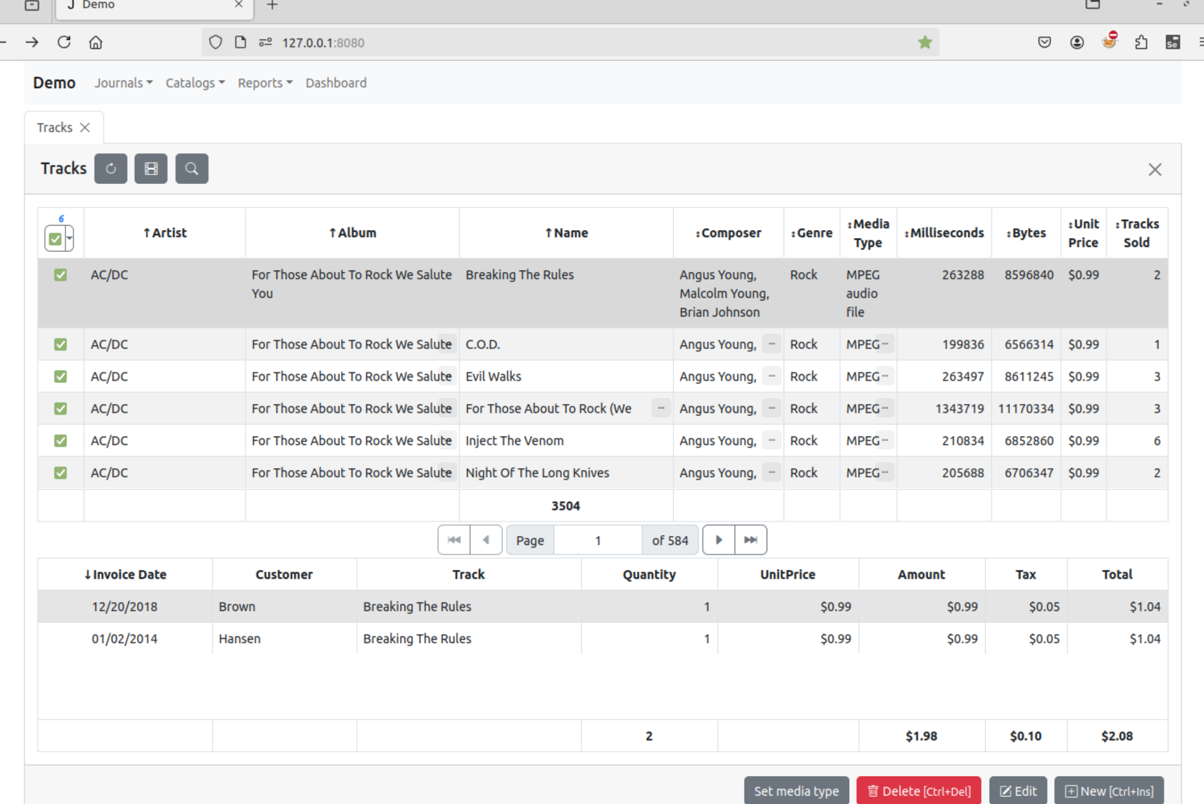
最终，*on_view_form_created* 的代码如下所示：

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

var scroll_timeout;

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

(续下页)

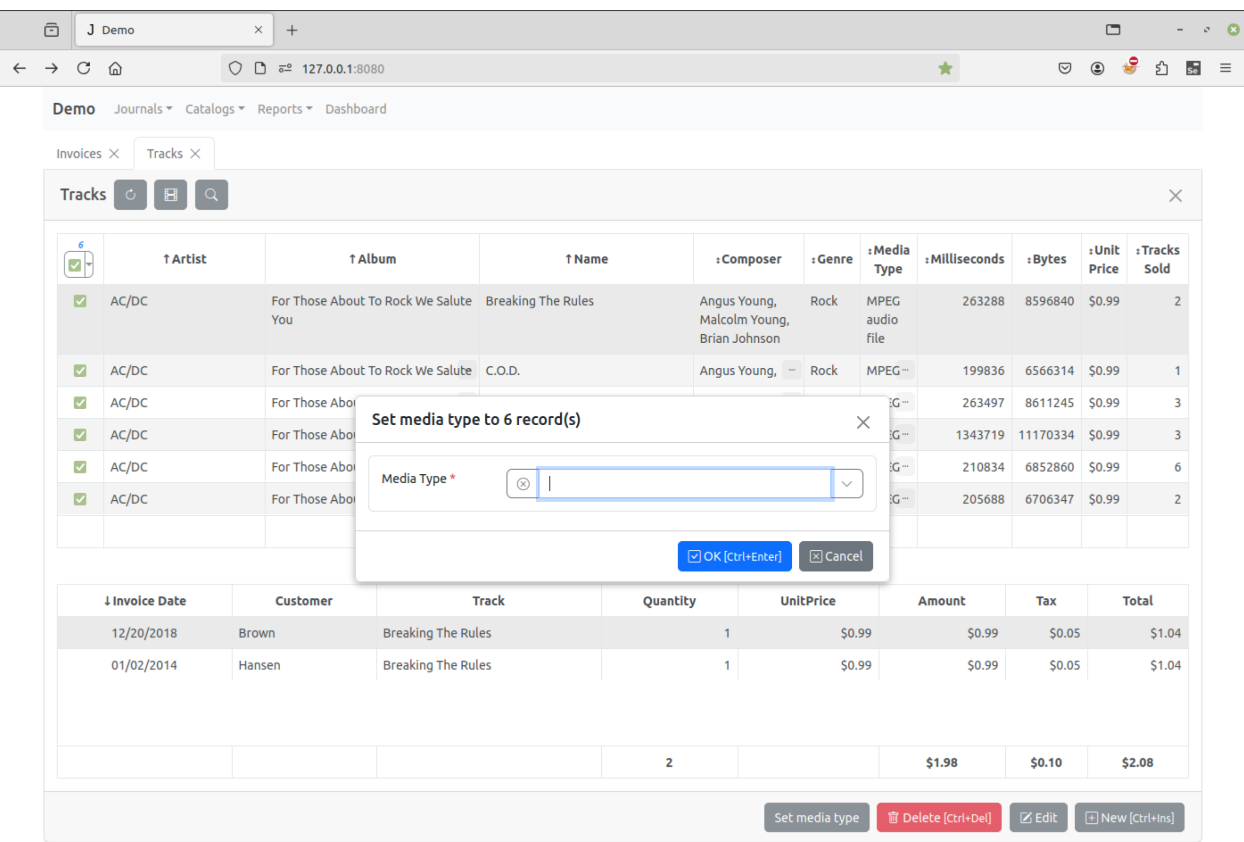
(接上页)

```
        );
    }
}

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



5.15 如何批量修改选中记录的字段值

本例将演示如何将“曲目 (Tracks)”主数据中选中记录的“Media Type”（媒体类型）字段批量设置为相同的值。



首先，需要将*table_options* 的 multiselect 属性设置为 true，这样在“曲目 (Tracks)”表格的最左侧会显示复选框，用户可以选择多条记录。然后在“曲目 (Tracks)”客户端模块的*on_view_form_created* 事件处理程序中，创建一个 **设置媒体类型 (Set media type)** 按钮。

```
function on_view_form_created(item) {
  item.table_options.multiselect = true;
  item.add_view_button('Set media type').click(function() {
      set_media_type(item);
  });
}
```

当点击该按钮时，会执行模块中定义的 `set_media_type` 函数。

在该函数中，我们创建了“曲目 (Tracks)”实体项的一个副本。调用*copy* 方法时，传入的 handlers 选项值为 false，表示副本不会继承应用构建器中表单对话框的设置，以及该实体项客户端模块中定义的所有函数和事件。

接着分析*selections* 属性，该属性是一个数组，包含用户选中的记录的主键字段值。

然后，通过调用*open* 方法并传入 open_empty 选项，初始化副本的数据集。我们还设置 fields 选项，使数据集只包含 media_type 字段，并将该字段的 required 属性设为 true。

最后，在调用*append_record* 方法前，动态分配*on_edit_form_created* 事件处理程序，以修改 **确定 (OK)** 按钮的点击事件（该事件原本定义在任务的客户端模块中）。

在新的点击事件处理程序中，首先调用*post* 方法，检查媒体类型值是否已设置。如果抛出异常，则调用*edit* 方法，允许用户对其进行重新设置。

```
function set_media_type(item) {
    var copy = item.copy({handlers: false}),
        selections = item.selections;
    if (selections.length > 1000) {
        item.alert('Too many records selected.');
    }
    else if (selections.length || item.rec_count) {
        if (selections.length === 0) {
            selections = [item.id.value];
        }

        copy.open({fields: ['media_type'], open_empty: true});

        copy.edit_options.title = 'Set media type to ' + selections.length +
          ' record(s)';
        copy.edit_options.history_button = false;
        copy.media_type.required = true;

        copy.on_edit_form_created = function(c) {
            c.edit_form.find('#ok-btn').off('click.task').on('click', function() {
                try {
                    c.post();
                    item.server('set_media_type', [c.media_type.value, selections],
                      function(res, error) {
                        if (error) {
                            item.alert_error(error);
                        }
                        if (res) {
                            item.selections = [];
                            item.refresh_page(true);
                            c.cancel_edit();
                            item.alert(selections.length + '
                              record(s) have been modified.');
                        }
                      }
                    );
                }
                finally {
                    c.edit();
                }
            });
        };
        copy.append_record();
    }
}
```

当用户点击 **确定 (OK)** 按钮时，实体项的*server* 方法会在服务器端执行 `set_media_type` 函数，从而批量修改选中记录的字段值。

服务器端修改完成后，在客户端会取消选中状态，刷新页面数据，调用*cancel_edit* 方法取消编辑状态，并提示用户操作的结果。

```
def set_media_type(item, media_type, selections):
    copy = item.copy()
    copy.set_where(id__in=selections)
    copy.open(fields=['id', 'media_type'])
    for c in copy:
```

(续下页)

(接上页)

```
        c.edit()
        c.media_type.value = media_type
        c.post()
    c.apply()
    return True
```

5.16 如何在不关闭编辑表单的情况下保存数据

你可以添加一个按钮来保存记录而不用关闭编辑表单。

下面是同步和异步场景下的示例:

```
function on_edit_form_created(item) {
    var save_btn = item.add_edit_button('Save and continue');
    save_btn.click(function() {
        if (item.is_changing()) {
            item.post();
            try {
              item.apply();
            }
            catch (e) {
              item.alert_error(error);
            }
            item.edit();
        }
    });
}
```

```
function on_edit_form_created(item) {
    var save_btn = item.add_edit_button('Save and continue');
    save_btn.click(function() {
        if (item.is_changing()) {
            item.disable_edit_form();
            item.post();
            item.apply(function(error){
                if (error) {
                    item.alert_error(error);
                }
                item.edit();
                item.enable_edit_form();
            });
        }
    });
}
```

5.17 如何在服务器端的同一个事务中保存两个表的变更

下面是两个示例。

在第一个示例中，每次调用*apply* 方法时，都会从连接池获取自己的连接，并将用户保存的更改提交到数据库。

在第二个示例中，先从连接池获取一个连接，并将其传递给每个*apply* 方法，这样所有更改会在最后一起提交到数据库。

```
import datetime

def change_invoice_date(item, invoice_id):
    now = datetime.datetime.now()

    invoices = item.task.invoices.copy(handlers=False)
    invoices.set_where(id=invoice_id)
    invoices.open()
    invoices.edit()
    invoices.invoice_date.value = now
    invoices.post()
    invoices.apply()

    customer_id = invoices.customer.value
    customers = item.task.customers.copy(handlers=False)
    customers.set_where(id=customer_id)
    customers.open()
    customers.edit()
    customers.last_modified.value = now
    customers.post()
    customers.apply()
```

```
import datetime

def change_invoice_date(item, invoice_id):
    now = datetime.datetime.now()

    con = item.task.connect()
    try:
        invoices = item.task.invoices.copy(handlers=False)
        invoices.set_where(id=invoice_id)
        invoices.open()
        invoices.edit()
        invoices.invoice_date.value = now
        invoices.post()
        invoices.apply(con)

        customer_id = invoices.customer.value
        customers = item.task.customers.copy(handlers=False)
        customers.set_where(id=customer_id)
        customers.open()
        customers.edit()
        customers.last_modified.value = now
        customers.post()
        customers.apply(con)

        con.commit()
    finally:
        con.close()
```

5.18 如何防止表的字段中出现重复值

其中一种方法是编写*on_apply* 事件处理程序。

在下面的示例中，delta 参数是一个数据集，包含将要存储到 users 表中的更改。

我们遍历所有记录的更改，如果该记录没有被删除，或者 login 字段发生了更改，则查找表中是否存在相同

登录名的记录，如果存在则抛出异常。如果用户在客户端使用编辑表单编辑记录，则无法保存该记录，并会看到相应的提示信息。

```
def on_apply(item, delta, params, connection):
    for d in delta:
        if not (d.rec_deleted() or d.rec_modified() and d.login.value == d.login.old_value):
            users = d.task.users.copy(handlers=False)
            users.set_where(login=d.login.value)
            users.open(fields=['login'])
            if users.rec_count:
                raise Exception('There is a user with this login - %s' % d.login.value)
```

5.19 如何实现基础的多用户？例如，让不同用户拥有各自独立的数据。

你可以通过 Jam.py 实现多用户。

例如，如果某个数据项有一个 user_id 字段（类型为 INT），可以在该项的服务端模块中添加如下代码即可实现。前提是必须启用认证功能：

```
def on_open(item, params):
    if item.session:
        user_id = item.session['user_info']['user_id']
        if user_id:
            params['__filters'].append(['user_id', item.task.consts.FILTER_EQ, user_id])

def on_apply(item, delta, params, connection):
    if item.session:
        user_id = item.session['user_info']['user_id']
        if user_id:
            for d in delta:
                if d.rec_inserted():
                    d.edit()
                    d.user_id.value = user_id
                    d.post()
                elif d.rec_modified():
                    if d.user_id.old_value != user_id:
                        raise Exception('You are not allowed to change record.')
                elif d.rec_deleted():
                    if d.user_id.old_value != user_id:
                        raise Exception('You are not allowed to delete record.')
```

上述代码，在*on_open* 和*on_apply* 事件处理程序中，利用了数据项的*session* 属性来获取唯一的用户 ID。

on_open 事件处理程序确保应用程序生成的 SQL 查询只会返回 user_id 字段等于当前请求用户 ID 的记录。

on_apply 事件处理程序会将 user_id 设置为新增或修改记录的用户 ID。

你还可以采用更通用的方式，将如下代码添加到任务的服务端模块中。这样，所有包含 user_id 字段的数据项都会自动应用多用户逻辑：

```
def on_open(item, params):
    if item.field_by_name('user_id'):
        if item.session:
            user_id = item.session['user_info']['user_id']
            if user_id:
                params['__filters'].append(['user_id', item.task.consts.FILTER_EQ, user_id])
```

(续下页)

(接上页)

```
def on_apply(item, delta, params, connection):
    if item.field_by_name('user_id'):
        if item.session:
            user_id = item.session['user_info']['user_id']
            if user_id:
                for d in delta:
                    if d.rec_inserted():
                        d.edit()
                        d.user_id.value = user_id
                        d.post()
                    elif d.rec_modified():
                        if d.user_id.old_value != user_id:
                            raise Exception('You are not allowed to change record.')
                    elif d.rec_deleted():
                        if d.user_id.old_value != user_id:
                            raise Exception('You are not allowed to delete record.')
```

用户还可以将上述方法与认证 结合使用。

5.20 我可以让 Jam.py 使用现有的数据库吗

请参见集成已有数据库

5.21 如何使用其他数据库中的数据

你可以使用其他数据库中的数据。

首先，你需要指定表名和字段信息。可以按照以下方式操作：

- 在任务树中选择 "项目 (Project)" 节点，点击页面右侧的 **数据库 (Database)** 按钮。
- 设置数据库为手动模式，并填写数据库连接的属性。
- 按照集成已有数据库 的说明导入表信息。
- 在任务树中再次选择 "项目 (Project)" 节点，点击 **数据库 (Database)** 按钮，恢复之前的设置。

然后，在新数据项的 服务端模块 (`Server module`) 中添加代码，实现对数据库中表的数据读写。

下面是 MySQL 数据库（自增主键字段）的代码示例：

```
import MySQLdb
from jam.db import mysql

def on_open(item, params):
    connection = item.task.create_connection_ex(mysql, database='demo', \
        user='root', password='111', host='localhost', encoding='UTF8')
    try:
        sql = item.get_select_query(params, mysql)
        rows = item.task.select(sql, connection, mysql)
    finally:
        connection.close()
    return rows, ''

def on_apply(item, delta, params):
    connection = item.task.create_connection_ex(mysql, database='demo', \
```

(续下页)

(接上页)

```
        user='root', password='111', host='localhost', encoding='UTF8')
    try:
        sql = delta.apply_sql(params, mysql)
        result = item.task.execute(sql, None, connection, mysql)
    finally:
        connection.close()
    return result
```

如果数据库使用生成器（generators）获取主键值（如 Firebird），则必须为新记录指定主键：

```
import fdb
from jam.db import firebird

def on_open(item, params):
    connection = item.task.create_connection_ex(firebird, database='demo.fdb', \
        user='SYSDBA', password='masterkey', encoding='UTF8')
    try:
        sql = item.get_select_query(params, firebird)
        rows = item.task.select(sql, connection, firebird)
    finally:
        connection.close()
    return rows, ''

def get_id(table_name, connection):
    cursor = connection.cursor()
    cursor.execute('SELECT NEXT VALUE FOR "%s" FROM RDB$DATABASE' % (table_name + '_SEQ'))
    r = cursor.fetchall()
    return r[0][0]

def on_apply(item, delta, params):
    connection = item.task.create_connection_ex(firebird, database='demo.fdb', \
        user='SYSDBA', password='masterkey', encoding='UTF8')
    for d in delta:
        if not d.id.value:
            d.edit()
            d.id.value = get_id(item.table_name, connection)
            for detail in d.details:
                for r in detail:
                    if not r.id.value:
                        r.edit()
                        r.id.value = get_id(r.table_name, connection)
                        r.post()
            d.post()
    try:
        sql = delta.apply_sql(params, firebird)
        result = item.task.execute(sql, None, connection, firebird)
    finally:
        connection.close()
    return result
```

你也可以在任务的 `on_open` 和 `on_apply` 事件中处理。下面是任务客户端模块的代码示例：

```
import MySQLdb
from jam.db import mysql

def on_open(item, params):
    if item.item_name in ['table1', 'table2']: # or
```

(续下页)

(接上页)

```
    #if item.table_name in ['table1', 'table2']:
      connection = item.task.create_connection_ex(mysql, database='demo', \
          user='root', password='111', host='localhost', encoding='UTF8')
      try:
          sql = item.get_select_query(params, mysql)
          rows = item.task.select(sql, connection, mysql)
      finally:
          connection.close()
      return rows, ''

def on_apply(item, delta, params):
    if item.item_name in ['table1', 'table2']:
      connection = item.task.create_connection_ex(mysql, database='demo', \
          user='root', password='111', host='localhost', encoding='UTF8')
      try:
          sql = delta.apply_sql(params, mysql)
          result = item.task.execute(sql, None, connection, mysql)
      finally:
          connection.close()
      return result
```

MSSQL 示例：

```
import pymssql
from jam.db import mssql

def on_open(item, params):
    connection = item.task.create_connection_ex(mssql, database='jam7', \
        user='sa', password='password', server='127.0.0.1', port='1433')
    try:
        sql = item.get_select_query(params, mssql)
        rows = item.task.select(sql, connection, mssql)
    finally:
        connection.close()
    return rows, ''

def on_apply(item, delta, params):
    connection = item.task.create_connection_ex(mssql, database='jam7', \
        user='sa', password='password', server='127.0.0.1', port='1433')
    try:
        sql = delta.apply_sql(params, mssql)
        result = item.task.execute(sql, None, connection, mssql)
    finally:
        connection.close()
    return result
```

备注

不要为这些表设置 History 属性为 True，否则会报错。历史表在项目中所有数据库只能有一个。你可以尝试在其他数据库中创建历史表，并为其编写 `on_open` 和 `on_apply` 事件处理程序。

备注

上述方法未在 Jam.py V7 下测试。

更简单的方案是使用数据库同义词（synonyms）。

5.22 如何处理来自其他应用或服务的请求，或获取其数据

你可以通过向应用发送带有“ext”的 post 请求来读写应用的数据。例如：

```
http://your_jampy_app.com/ext/something
```

当服务器上的 Web 应用收到这样的请求时，会触发*on_ext_request* 事件。

例如，Jam.py 应用程序的表 `account_transactions` 有一个字段 `actual_amount` 。应用程序的 Task 模块包含如下代码：

```
def on_ext_request(task, request, params):
    reqs = request.split('/')
    if reqs[2] == 'expenses':
        result = task.account_transactions.expenses(task, params)
        return result
```

表 `account_transactions` 的 Task 服务端模块包含：

```
from jam.common import cur_to_str

def expenses(item, params):
    inv = item.task.account_transactions.copy()
    inv.open()
    total = 0
    for i in inv:
        total += i.actual_amount.value

    total = cur_to_str(total)
    return(total)
```

使用 Curl 命令访问应用会返回如下结果：

```
...\> curl -k https://your_jampy_app.com/ext/expenses -d "[]" -H  "Content-Type: application/
↪json"
{"result": {"status": 9, "data": "-$2590.01", "modification": 99}, "error": null}
```

5.22.1 Curl 传递变量

在 Demo 应用中，如果在 Task 服务端模块中添加如下代码：

```
def on_ext_request(task, request, params):
    print(request, params)
    reqs = request.split('/')
    if reqs[2] == 'bla':
        users = task.customers.copy(handlers=False)
        users.set_where(id=params['id'])
        users.open()
        if users.rec_count == 1:
                return {
                    'id': users.firstname.value,
```

(续下页)

(接上页)

```
                    'firstname': users.firstname.value,
                }
```

用 Curl 传递参数会返回如下结果：

```
...\> curl  http://localhost:8080/ext/bla -d '{"id": "2", "firstname": "Leonie"}' -H "Content-
↪Type: application/json"
{"result": {"status": 9, "data": {"id": "Leonie", "firstname": "Leonie"}, "modification": 2014},
↪ "error": null}
```

5.22.2 处理请求中的数据

上面同样的应用可以被其他 Jam.py 应用通过服务端模块访问：

```
try:
    # For Python 3.0 and later
    from urllib.request import urlopen
except ImportError:
    # Fall back to Python 2's urllib2
    from urllib2 import urlopen
import json
import time

params = []

def api_fetch(url, request, params):
    try:
        a = urlopen(url + '/' + request, data=str.encode(json.dumps(params)))
        r = json.loads(a.read().decode())
        return r['result']['data']
    except:
        return False


def send(item):
    result= ''
    res = []
    request = 'expenses';
    endpoint = 'https://your_jampy_app.com/ext';

    try:
        # print('Req: ' + request)
        result = api_fetch(endpoint, request, [])
    except:
        return False
    if result:
        # print(result)
        res.append(
            {
                # 'id': 1,
                'request': request,
                'endpoint': endpoint,
                'value': result,
            }
        )
        return res
```

(续下页)

(接上页)

```
    else:
        raise Exception('Could not connect!')
```

某个虚拟表的客户端模块（包含 request、endpoint 和 value 字段）：

```
function on_view_form_created(item) {
    item.view_options.open_item = false;
    item.view_options.form_header = false;
    item.open({open_empty: true});
    item.paginate = false;
    item.view_form.find("#edit-btn").hide();
    item.view_form.find("#delete-btn").hide();
    item.view_form.find("#new-btn").hide();
    item.alert('Fetching!');
    item.server('send', function(records, err) {
        item.disable_controls();
        if (err) {
            item.warning('Failed to fetch data: ' + err);
        }
        else {
            if (records.length > 0) {
                records.forEach(function(rec) {
                    item.append();
                    // item.id.value = rec.id;
                    item.request.value = rec.request;
                    item.endpoint.value = rec.endpoint;
                    item.value.value = rec.value;
                    item.post();
                });
                item.first();
                item.enable_controls();
                item.alert('Successfully fetched from API!');
            }
        }
    });
}
```

最终结果会在表格中显示从 endpoint 获取的 value，以及对应的 request。

5.23 如何在后台执行计算

你可以在任务的服务端模块中使用如下代码，在 Web 应用中每隔 3 分钟（可通过 interval 设置修改）启动一个后台线程执行一些计算：

```
import threading
import time
import traceback

def background(task):
    interval = 3 * 60
    time.sleep(interval)
    while True:
        if not time:
            return
        with task.lock('background'):
```

(续下页)

(接上页)

```
            try:
                print('background')
                # some code to execute in background for example:
                # tracks = task.tracks.copy()
                # tracks.open()
                # for t in tracks:
                #     t.edit()
                #     t.sold.value = #some value
                #     t.post()
                # tracks.apply()
            except Exception as e:
                traceback.print_exc()
        time.sleep(interval)

def on_created(task):
    bg = threading.Thread(target=background, args=(task,))
    bg.daemon = True
    bg.start()
```

> **备注**
>
> 当多个 Web 应用以并行进程运行时，background 函数会在每个进程中执行。为防止该函数被同时执行，我们使用了 task 的 lock 方法。

> **备注**
>
> Jam.py V7 引入了 计算字段 *(calculated field)* 。现在可以在主/明细（Master/Detail）场景下，使用服务器端函数（SUM、COUNT、MIN、MAX、AVG）对其他表字段进行汇总。用户可以根据需要，将服务器端的计算代码替换为计算字段。

5.24 支持在明细表中嵌套明细表吗？

支持，你可以在明细表中再嵌套明细表。

假设我们有三个对象——“投票 (Polls)”、“问题 (Questions)”端和“答案 (Answers)”端。“答案 (Answers)”是“问题 (Questions)”的明细表，我们将“问题 (Questions)”作为“投票 (Polls)”的明细表。

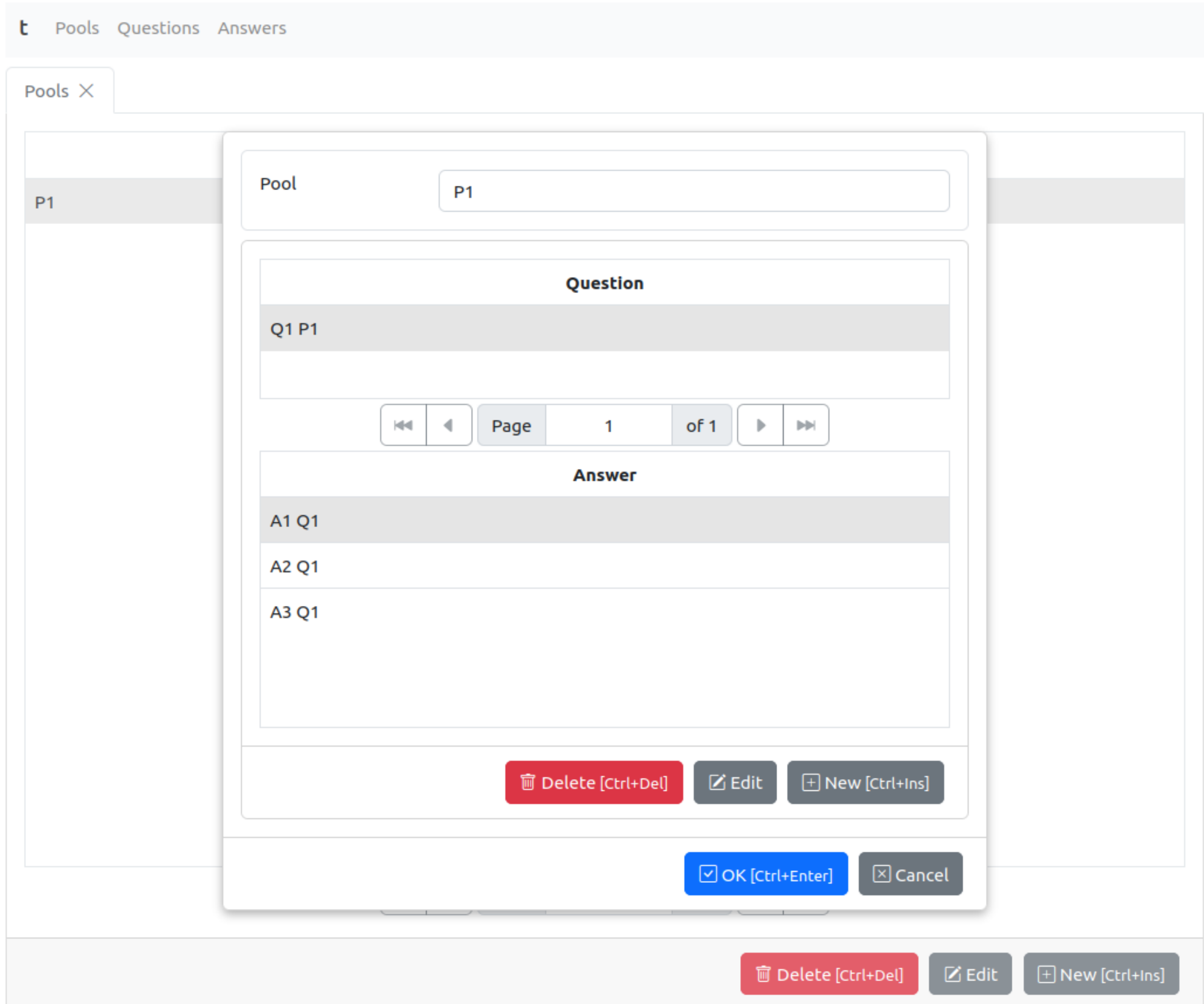
一种实现方式是在“Questions”表中添加一个整型字段“poll”，并在“投票 (Polls)”的客户端模块中添加如下代码：

```
function on_edit_form_created(item) {
    item.edit_options.form_header = false;
    var q = task.questions.copy();
    q.set_where({pool: item.id.value});
    q.view(item.edit_form.find('.edit-detail'));

    q.view_options.form_header = false;

    q.on_view_form_created = function(quest) {
        quest.paginate = false;
        quest.view_options.form_header = false;
```

(续下页)

(接上页)

```
    };

    q.on_before_append = function(quest) {
        if (!item.id.value) {
            quest.alert_error('Poll is not specified.');
            quest.abort();
        }
    };

    q.on_before_post = function(quest) {
        q.pool.value = item.id.value;
    };
}

function on_field_changed(field, lookup_item) {
    var item = field.owner;
    item.apply();
    item.edit();
}


function on_before_delete(item) {
    var q = task.questions.copy();
    q.set_where({id: item.id.value});
    q.open();
    while (!q.eof()) {
        q.delete();
    }
    q.apply();
}
```



5.25 CSV 文件的导入与导出

首先，在数据项的客户端模块中，我们创建两个按钮，点击后会执行相应的导入/导出函数：

```
function on_view_form_created(item) {
  var csv_import_btn = item.add_view_button('Import csv file'),
      csv_export_btn = item.add_view_button('Export csv file');
  csv_import_btn.click(function() { csv_import(item) });
  csv_export_btn.click(function() { csv_export(item) });
}

function  csv_export(item) {
    item.server('export_scv', function(file_name, error) {
        if (error) {
            item.alert_error(error);
        }
        else {
            var url = [location.protocol, '//', location.host, location.pathname].join('');
            url += 'static/files/' + file_name;
            window.open(encodeURI(url));
```

(续下页)

(接上页)

```
        }
    });
}

function csv_import(item) {
    task.upload('static/files', {accept: '.csv', callback: function(file_name) {
        item.server('import_scv', [file_name], function(error) {
            if (error) {
                item.warning(error);
            }
            item.refresh_page(true);
        });
    }});
}
```

这些函数会调用服务器模块中定义的相应函数。在该模块中我们使用了 Python 的 csv 模块。导出时不会包含系统字段——主键字段和删除标志字段。

下面是用 Pyton 3 实现的代码:

```
import os
import csv

def export_scv(item):
    copy = item.copy()
    copy.open()
    file_name = item.item_name + '.csv'
    path = os.path.join(item.task.work_dir, 'static', 'files', file_name)
    with open(path, 'w', encoding='utf-8') as csvfile:
        fieldnames = []
        for field in copy.fields:
            if not field.system_field():
                fieldnames.append(field.field_name)
        writer = csv.DictWriter(csvfile, fieldnames=fieldnames)
        writer.writeheader()
        for c in copy:
            dic = {}
            for field in copy.fields:
                if not field.system_field():
                    dic[field.field_name] = field.text
            writer.writerow(dic)
    return file_name

def import_scv(item, file_name):
    copy = item.copy()
    path = os.path.join(item.task.work_dir, 'static', 'files', file_name)
    with open(path, 'r', encoding='utf-8') as csvfile:
        copy.open(open_empty=True)
        reader = csv.DictReader(csvfile)
        for row in reader:
            print(row)
            copy.append()
            for field in copy.fields:
                if not field.system_field():
                    field.text = row[field.field_name]
            copy.post()
        copy.apply()
```

用 Python 2 实现的代码如下：

```
import os
import csv

def export_scv2(item):
    copy = item.copy()
    copy.open()
    file_name = item.item_name + '.csv'
    path = os.path.join(item.task.work_dir, 'static', 'files', file_name)
    with open(path, 'wb') as csvfile:
        fieldnames = []
        for field in copy.fields:
            if not field.system_field():
                fieldnames.append(field.field_name.encode('utf8'))
        writer = csv.DictWriter(csvfile, fieldnames=fieldnames)
        writer.writeheader()
        for c in copy:
            dic = {}
            for field in copy.fields:
                if not field.system_field():
                    dic[field.field_name.encode('utf8')] = field.text.encode('utf8')
            writer.writerow(dic)
    return file_name

def import_scv2(item, file_name):
    copy = item.copy()
    path = os.path.join(item.task.work_dir, 'static', 'files', file_name)
    with open(path, 'rb') as csvfile:
        item.task.execute('delete from %s' % item.table_name)
        copy.open(open_empty=True)
        reader = csv.DictReader(csvfile)
        for row in reader:
            print(row)
            copy.append()
            for field in copy.fields:
                if not field.system_field():
                    field.text = row[field.field_name.encode('utf8')].decode('utf8')
            copy.post()
        copy.apply()
```

5.26 认证

在 Jam.py 的代码仓库中有一个“身份验证 (Authentication)”项目。

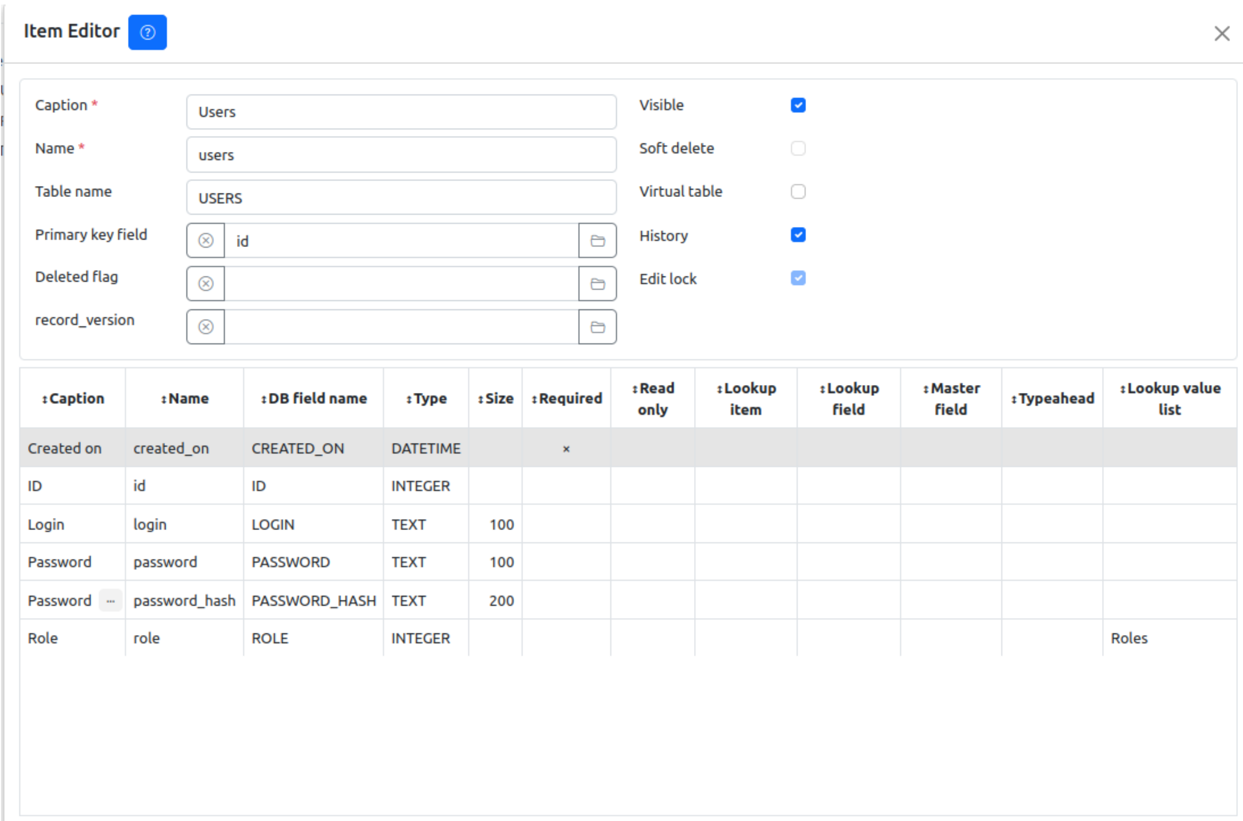
https://jam-py.com/repository/auth.zip

这个项目演示了本节的前三个主题。您可以将其下载，创建一个新项目，然后导入 这个文件。

5.26.1 如何通过自定义用户表进行认证

默认情况下，所有的用户信息都存储在 admin.sqlite 数据库中的一个表中。该表的结构是固定的，无法更改。本节介绍如何使用自定义用户表中的数据进行用户认证。

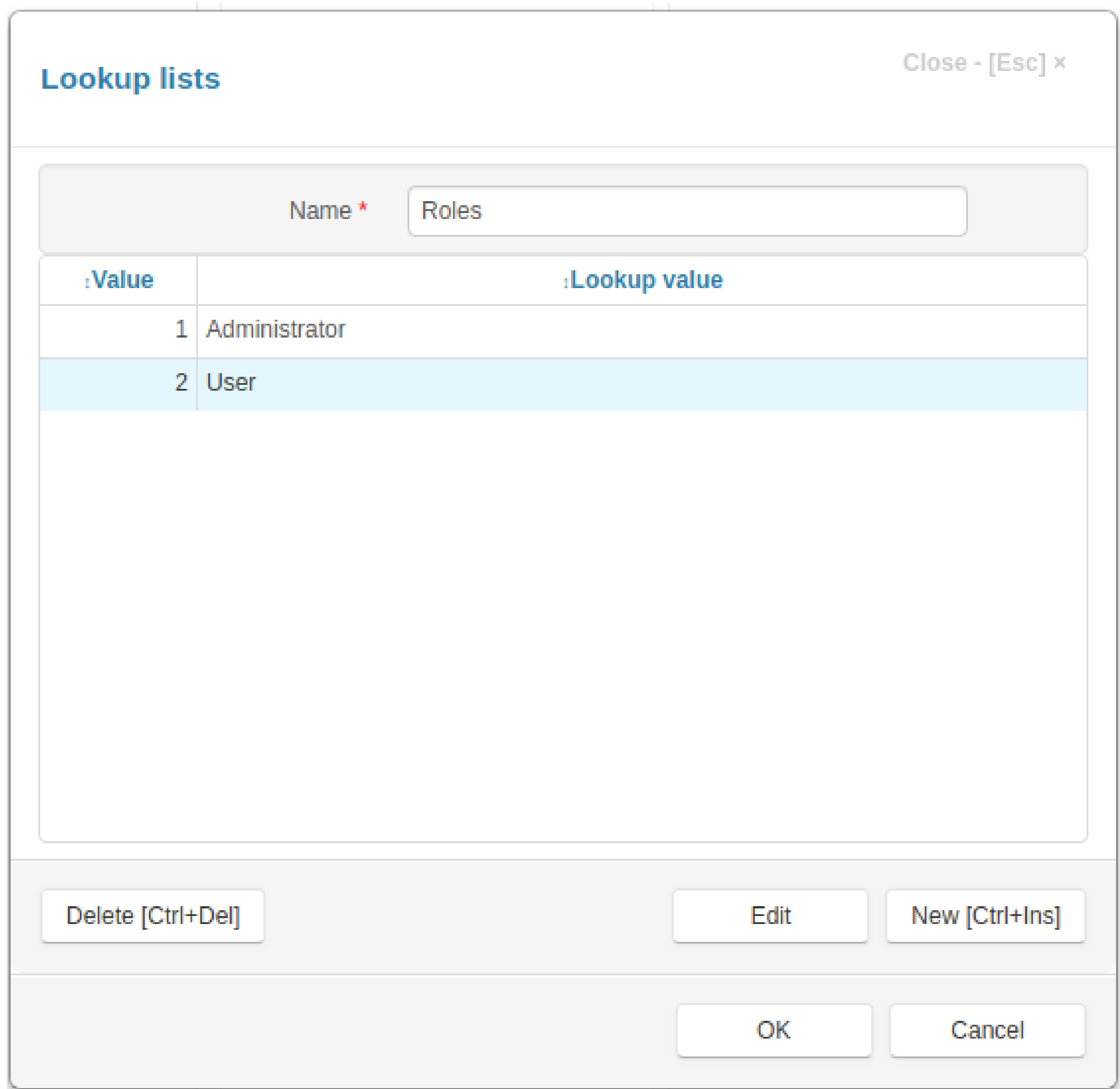
首先，创建一个分组 **认证 (Authentication)** ，选中它并添加一个名为 **用户 (Users)** 的实体项，包含如下字段：



我们不会在表中直接存储用户密码，而是在界面中使用该字段。实际存储的是加盐后的密码哈希值，保存在 password_hash 字段中。

我们还创建了名为“角色 (Roles)”的查找列表，并在“角色“(Roles)”字段定义中使用。

我们为其添加了与角色表中相同的角色（id 和名称）。后续需要保持这些角色的同步。



在角色 中，需要设置只有负责该项的用户才能查看 **用户 (Users)** 实体项。

我们在查看表单对话框 和编辑表单对话框 的字段列表中移除了 password_hash 字段。

在 **用户 (Users)** 服务端模块中，定义如下*on_apply* 事件处理程序：

```
def on_apply(item, delta, params, connection):
    for d in delta:
        if not (d.rec_deleted() or d.rec_modified() and d.login.value == d.login.old_value):
            users = d.task.users.copy(handlers=False)
            users.set_where(login=d.login.value)
            users.open(fields=['login'])
            if users.rec_count:
                raise Exception('There is a user with this login - %s' % d.login.value)
        if d.password.value:
            d.edit();
```

(续下页)

(接上页)

```
            d.password_hash.value = delta.task.generate_password_hash(d.password.value)
            d.password.value = None
            d.post();
```

在该事件处理程序中，我们检查是否有相同登录名的用户，如果有则抛出异常，否则用 task 的*generate_password_hash* 方法生成哈希，并将密码字段设为 None。

在客户端模块中，定义如下 on_field_get_text 事件处理程序。它会将密码显示为“******”：

```
function on_field_get_text(field) {
    var item = field.owner;
    if (field.field_name === 'password') {
        if (item.id.value || field.value) {
            return '**********';
        }
    }
}
```

最后，在 **任务 (Task)** 的服务端模块中定义*on_login* 事件处理程序：

```
def on_login(task, form_data, info):
    users = task.users.copy(handlers=False)
    users.set_where(login=form_data['login'])
    users.open()
    if users.rec_count == 1:
        if task.check_password_hash(users.password_hash.value, form_data['password']):
            return {
                'user_id': users.id.value,
                'user_name': users.name.value,
                'role_id': users.role.value,
                'role_name': users.role.display_text
            }
```

现在需要在 **用户 (Users)** 中添加一个有权限管理用户的管理员。之后即可在项目参数 中启用安全模式（Safe mode）。

5.26.2 如何创建用户注册表单

本部分内容适用于 Jam.py V5。如需 V7 相关内容，请访问：

https://groups.google.com/g/jam-py/c/DwALkbBsFcw/m/MujcOlgYAAAJ

或在此下载示例：

https://drive.google.com/file/d/1Gty1bC2I3srFo9XeQ0dXdAdnYeJbwoeB/view?usp=drive_link

本主题假设你已根据前一节创建了用户 实体项。

现在我们创建一个 *register.html* 文件。

它包含一个注册表单：

```
<form id="login-form" target="dummy" class="form-horizontal" style="margin: 0;">
    <div class="control-group">
        <label class="control-label" for="name">Name</label>
        <div class="controls">
            <input type="text" id="name" placeholder="Login">
        </div>
```

(续下页)

(接上页)

```
    </div>
    <div class="control-group">
        <label class="control-label" for="login">Login</label>
        <div class="controls">
            <input type="text" id="login" placeholder="Login">
        </div>
    </div>
    <div class="control-group">
        <label class="control-label" for="password1">Password</label>
        <div class="controls">
            <input type="password" id="password1"
            placeholder="Password" autocomplete="on">
        </div>
    </div>
    <div class="control-group">
        <label class="control-label" for="password2">Repeat password</label>
        <div class="controls">
            <input type="password" id="password2"
            placeholder="Repeat password" autocomplete="on">
        </div>
    </div>
    <div class="alert alert-success" style="margin: 0; display: none">
        You have been successfuly registered.
    </div>
    <div class="alert alert-error" style="margin: 0; display: none">
    </div>
    <div class="form-footer">
        <input type="button" class="btn expanded-btn pull-right"
        id="register-btn" value="OK" tabindex="3">
    </div>
</form>
```

以及一段 JavaScript 代码：

```
$(document).ready(function(){

    function register(name, login, password) {
        $.ajax({
            url: "ext/register",
            type: "POST",
            contentType: "application/json;charset=utf-8",
            data: JSON.stringify([name, login, password]),
            success: function(response, textStatus, jQxhr) {
                if (response.result.data) {
                    show_alert(response.result.data);
                }
                else {
                    $("div.alert-success").show();
                    setTimeout(
                        function() {
                            window.location.href = "index.html";
                        },
                        1000
                    );
                }
            },
            error: function(jqXhr, textStatus, errorThrown) {
```

(续下页)

(接上页)

```
                console.log(errorThrown);
            }
        });
    }
    function show_alert(message) {
        $("div.alert-error")
            .text(message)
            .show();
    }

    $('input').focus(function() {
        $("div.alert").hide();
    });

    $("#register-btn").click(function() {
        var name = $("#name").val(),
            login = $("#login").val(),
            password1 = $("#password1").val(),
            password2 = $("#password2").val();
        if (!name) {
            show_alert('Name is not specified');
        }
        else if (!login) {
            show_alert('Login is not specified');
        }
        else if (!password1) {
            show_alert('Password is not specified');
        }
        else if (password1 !== password2) {
            show_alert('Passwords do not match');
        }
        else {
            register(name, login, password1)
        }
    })
})
```

当用户点击 **确定 (OK)** 按钮时，JavaScript 会向服务器发送 url 为 "ext/register " 的 ajax post 请求，参数为 "name, login, password "。

当服务器收到以 "ext/" 开头的请求时，会触发*on_ext_request* 事件。

任务 (Task) 的服务端模块包含如下 `on_ext_request` 事件处理程序：

```
def on_ext_request(task, request, params):
    reqs = request.split('/')
    if reqs[2] == 'register':
        name, login, password = params
        users = task.users.copy(handlers=False)
        users.set_where(login=login)
        users.open()
        if users.rec_count:
            return 'Existing login, please use different login'
        users.append()
        users.name.value = name
        users.login.value = login
        users.password_hash.value = task.generate_password_hash(password)
```

(续下页)

(接上页)

```
        users.role.value = 2
        users.post()
        users.apply()
```

它会检查 url 是否包含“register”，然后判断是否有相同登录名的用户，如果没有则注册新用户。

参见

on_ext_request

5.26.3 如何让用户能够修改密码

首先，创建一个“修改密码 (Change password)”实体项项。在创建时，在实体项编辑对话框 中将“虚拟表 (Virtual table)”和“可见 (Visible)”属性都设置为 false。添加两个字段：“旧密码 (Old password)”、“新密码 (New password)”。

我们将用这个数据项来显示“修改密码 (Change password)”对话框。

要打开该对话框，需要在 `index.html` 中添加一个 id 为”pass” 的“修改密码 (Change password)”菜单项：

```
<div class="container">
    <div id="taskmenu" class="navbar">
        <div class="navbar-inner">
            <ul id="menu" class="nav">
            </ul>
            <ul id="menu-right" class="nav pull-right">
                <li id="pass"><a href="#">Change password</a></li>
            </ul>
        </div>
    </div>
</div>
```

然后在 **任务 (Task)** 的客户端模块的*on_page_loaded* 事件处理程序中添加如下代码：

```
if (task.change_password.can_view()) {
    $("#menu-right #pass a").click(function(e) {
        e.preventDefault();
        task.change_password.open({open_empty: true});
        task.change_password.append_record();
    });
}
else {
    $("#menu-right #pass a").hide();
}
```

这段代码会检查用户是否有权限查看该数据项，如果有则打开空数据集并创建编辑表单，否则隐藏该菜单项。

在“修改密码 (Change password)”客户端模块中添加如下代码：

```
function on_edit_form_created(item) {
    item.edit_form.find("#ok-btn")
        .off('click.task')
        .on('click', function() {
            change_password(item);
        });
    item.edit_form.find("#cancel-btn")
```

(续下页)

(接上页)

```
        .off('click.task')
        .on('click', function() {
            item.close_edit_form();
        });
}

function change_password(item) {
    item.post();
    item.server('change_password', [item.old_password.value, item.new_password.value],␣
↪function(res) {
        if (res) {
            item.warning('Password has been changed. <br> The application will be reloaded.',
              function() {
                  task.logout();
                  location.reload();
              });
        }
        else {
            item.alert_error("Can't change the password.");
            item.edit();
        }
    });
}

function on_field_changed(field, lookup_item) {
    var item = field.owner;
    if (field.field_name === 'old_password') {
        item.server('check_old_password', [field.value], function(error) {
            if (error) {
                item.alert_error(error);
            }
        });
    }
}

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

在这里我们重写了 **确定 (OK)** 和 **取消 (Cancel)** 按钮的点击事件。默认情况下，这些事件在任务的客户端模块中定义，用于保存记录更改和取消编辑。在 `on_edit_form_close_query` 事件处理程序中返回 true，这样任务客户端模块中定义的"是/否/取消"对话框就不会弹出。

`on_field_changed` 事件处理程序会检查旧密码是否正确。它和 `change_password` 函数会向服务器发送请求，调用数据项服务器模块中定义的函数：

```
def change_password(item, old_password, new_password):
    user_id = item.session['user_info']['user_id']
    users = item.task.users.copy(handlers=False)
    users.set_where(id=user_id)
    users.open()
    same_password = item.task.check_password_hash(users.password_hash.value, old_password)
    if users.rec_count== 1 and same_password:
        users.edit()
        users.password_hash.value = item.task.generate_password_hash(new_password)
        users.post()
        users.apply()
```

(续下页)

(接上页)

```
        return True
    else:
        return False

def check_old_password(item, old_password):
    user_id = item.session['user_info']['user_id']
    users = item.task.users.copy(handlers=False)
    users.set_where(id=user_id)
    users.open()
    same_password = item.task.check_password_hash(users.password_hash.value, old_password)
    if users.rec_count == 1 and same_password:
        return
    else:
        return 'Invalid password'
```

session 属性用于获取当前用户的 `id`。

密码修改后，客户端会自动刷新。

5.27 如何将 v5 项目迁移到 v7

对于任何 v5 项目，首先备份 **index.html** 文件，然后将 v7 Jam.py `Demo` 应用程序中的 **index.html** 和 **template.html** 文件复制到你的应用目录下。或者，也可以从用 `jam-project.py` 创建的新 v7 项目中复制同名文件。

卸载 v5 版本的 Jam.py 并安装 v7 版本。或者可以同时运行两个虚拟 Python 环境，分别安装两个版本的 Jam.py。

用 v7 Jam.py 正常启动应用。

需要注意以下事项：

1. 迁移 Jam.py v5 应用程序的最大障碍是"编辑锁 (Edit Lock)"。如果表启用了该功能，Jam.py v7 期望每个启用编辑锁的表都包含一个 *record_version* （INT）字段。

   迁移到 v7 时有两个选择：一是在迁移前禁用"编辑锁"，二是在迁移前为相关表添加 *record_version* 字段。

   在数据库结构中添加该字段后，需要在应用构建器中手动添加该字段，并为每个启用"编辑锁"的表开启"数据库手动模式 (DB Manual Mode)"。"记录版本 (record version)"选项应设置为 *record_version* 字段。

   显然，如果禁用"编辑锁"，则无需任何其它额外操作。项目参数中也没有创建"锁定实体项 (Lock Item)"的选项。

2. 下一个问题是模板。Jam.py v7 期望应用程序目录下有 **templates.html** 文件。由于使用了 Bootstrap 5，从 Jam.py v5 的 **index.html** 文件迁移现有模板到 **templates.html** 文件时会有一些小的差异。

3. **index.html** 也需要调整，以适配 Bootstrap 5 依赖。请参考下方示例。

4. v7 版本不再支持外键（Foreign Key）。这意味着如果 v5 中创建了外键，v7 将无法管理外键。

5. Jam.py v7 引入了 **计算字段 (calculated field)** 。现在可以在主/明细（Master/Detail）场景下，使用服务端函数（SUM、COUNT、MIN、MAX、AVG）对表字段进行汇总。用户可以根据需要，将服务端的计算代码替换为计算字段。

6. Jam.py v7 现在采用 Python 依赖管理，初次安装时会自动安装所需库。这与 v5 不同，v5 所有依赖都"锁定"在 Jam.py 分发包中，可以通过单一的 `jam` 文件夹部署应用。

7. 对于 Jam.py 5.x 及以下版本的应用，将 **Task/Client** 模块代码替换为 `Demo` 应用或新 v7 项目的 **Task/Client** 模块代码。如果 v5 **Task/Client** 模块有自定义代码，再将其补充进去。

例如，Demo 应用的 **index.html** 文件内容如下：

```
<link href="jam/css/bootstrap-cerulean.css" rel="stylesheet">            <!--do not modify-->
<link href="jam/css/bootstrap-responsive.css" rel="stylesheet">
<link href="jam/css/bootstrap-modal.css" rel="stylesheet">
<link href="jam/css/datepicker.css" rel="stylesheet">
...
<script src="jam/js/bootstrap.js"></script>
<script src="jam/js/bootstrap-modal.js"></script>
<script src="jam/js/bootstrap-modalmanager.js"></script>
<script src="jam/js/bootstrap-datepicker.js"></script>
<script src="jam/js/jquery.maskedinput.js"></script>
<script src="jam/js/jam.js"></script>
```

需要修改为：

```
<link href="jam/css/bs5/bootstrap.css" rel="stylesheet">
<link href="jam/css/bs5/bootstrap-icons.css" rel="stylesheet">
<link href="jam/css/zebra_datepicker/bootstrap/zebra_datepicker.min.css" rel="stylesheet">
...
<script src="jam/js/bs5/bootstrap.bundle.js"></script>
<script src="jam/js/zebra_datepicker.js"></script>
<script src="jam/js/jquery.maskedinput.js"></script>
<script type="module" src="jam/js/jam.js"></script>
```

index.html 文件的其它部分只需做最小的调整。

5.28 如何编写测试

Jam.py 使用 Mocha/Chai 进行前端单元测试，使用 pytest 进行数据集集成测试。这些例子在 tests 文件夹中。

首先，从 Jam.py-v7 仓库创建自已的仓库副本（forks）；

然后，克隆你的仓库副本：

```
...\> git clone https://github.com/YourGitHubName/jam-py-v7.git
...\> cd jam-py-v7/tests/project
```

要添加一个新的前端测试，请在 project/js 文件夹中添加 JavaScript 文件。假如，我们想对 user 表的一个名为 username 的字段进行 CRUD 测试。

项目的文件夹有如下结构：

```
├── admin.sqlite
├── css
│   └── project.css
├── index.html
├── js
│   ├── test_dataset.js
│   ├── test_details.js
│   ├── test_edit_lock.js
│   ├── test_fields.js
│   ├── test.js
│   └── test_locale.js
├── langs.sqlite
├── server.py
├── templates.html
├── test.html
```

(续下页)

(接上页)

```
├── test.sqlite
└── wsgi.py
```

像往常一样启动项目：

```
...\> ./server.py
```

在构建器里，添加一个 Users 表，它有一个名为 Username 的字段，如图示：

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

在 index.html 中添加一个新的 js 文件：

```
<script src="js/test_users.js"></script>
```

index.html 内容应该如下所示：

```
<!DOCTYPE html>
<html>
    <head>
        <meta charset="utf-8">
        <title>Jam.py tests</title>
        <link rel="stylesheet" href="https://cdnjs.cloudflare.com/ajax/libs/mocha/6.1.4/mocha.
↪css">
    </head>
    <body>
        <div id="mocha"></div>
        <script src="https://cdnjs.cloudflare.com/ajax/libs/mocha/11.7.2/mocha.js"></script>
        <script src="https://cdn.jsdelivr.net/npm/chai@4.3.4/chai.js"></script>
        <script>mocha.setup('bdd')</script>
        <script src="jam/js/jquery.js"></script>
        <script src="jam/js/bs5/bootstrap.bundle.js"></script>
        <script src="jam/js/zebra_datepicker.js"></script>
        <script src="jam/js/jquery.maskedinput.js"></script>
        <script type="module" src="jam/js/jam.js"></script>


        <script src="js/test_dataset.js"></script>
        <script src="js/test_details.js"></script>
        <script src="js/test_fields.js"></script>
        <script src="js/test_edit_lock.js"></script>
        <script src="js/test_locale.js"></script>
        <script src="js/test_users.js"></script>
        <script>
            $(document).ready(function(){
                task.load(function() {
                    mocha.run();
                });
            });
        </script>
    </body>
</html>
```

创建带有测试内容的 **js/test_users.js** 文件，让后访问应用程序：

```
127.0.0.1:8080/index.html
```

所有单元测试都将运行并显示结果。数据库 tests.sqlite 将使用新用户进行更新。如果该表是使用“软删除”选项创建的，表中新行的 `DELETED` 字段将被设置为 1 。否则，新记录将被删除。

如果更改后一切顺利，创建一个 Github `pull` 请求。

5.29 在表单之间导航并保留数据的方法

本指南说明在创建新记录时，如何在表单之间导航并在表单间保持数据完整性。

该模式包括：

- 一个包含按钮以打开空表单（表单 2）的视图。
- 表单 2，带有用于返回到表单 1 的导航按钮。
- 表单 1，作为接收来自表单 2 数据的目标表单。

5.29.1 1. 设置初始视图

向视图中添加一个 **New** 按钮，用于打开表单 2：

```
// ===== VIEW CONFIGURATION =====

function on_view_form_created(item) {
    item.view_form.find('#new-btn')
        .text('New')
        .off('click.task')
        .on('click', function() {
            openForm2();
        });
    item.refresh_page(true);
}

function openForm2() {
    task.f2.open({open_empty: true});
    task.f2.append_record();
}
```

5.29.2 2. 配置表单 2（中间表单）

为表单 2 设置一个**下一表单 (Next Form)**（下一表单）按钮，用于导航回表单 1：

```
// ===== FORM 2 CONFIGURATION =====

function on_edit_form_created(item) {
    item.edit_form.find('#ok-btn')
        .text('Next Form')
        .off('click.task')
        .on('click', function() {
            item.close_edit_form();
            setTimeout(function() {
                openForm1(item);
            }, 300);
        });
}

function openForm1(item) {
```

(续下页)

(接上页)

```
    task.f1.open({open_empty: true});
    task.f1.append_record();
}

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

5.29.3 3. 配置表单 1（目标表单）

将表单 2 的数据映射到表单 1 字段：

```
// ===== FORM 1 CONFIGURATION =====

function on_edit_form_created(item) {
    var title = 'First Form value: ';

    if (item.is_new()) {
        // 将数据从表单2传到表单1
        item.f1t1.value = task.f2.f2t1.value;

        if (item.f1t1.value) {
            title += item.f1t1.value + ' value typed';
        }
        item.edit_options.title = title;
    } else {
        title = item.f1t1.value;
        item.edit_options.title = title;
    }
}
```

5.29.4 需要记住的关键点

`open_empty`: true：确保表单以无预加载数据的状态打开

`append_record()`：向表单添加一个新的空记录

`setTimeout()`：在打开下一个表单之前给当前表单适当的关闭时间

`on_edit_form_close_query`：返回 true 可绕过未保存更改的警告

5.29.5 字段映射参考

来源	目标	说明
task.f2.f2t1.value	item.f1t1.value	将表单 2 字段 f2t1 的数据传到表单 1 字段 f1t1

5.29.6 另见

on_edit_form_close_query

表单窗体

create_edit_form

edit_form

CHAPTER 6

应用程序构建器

应用程序构建器——是一个用于应用程序开发和数据库管理的 Jam.py Web 应用程序。

要运行应用程序构建器，请打开 Web 浏览器并在浏览器地址栏中输入

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

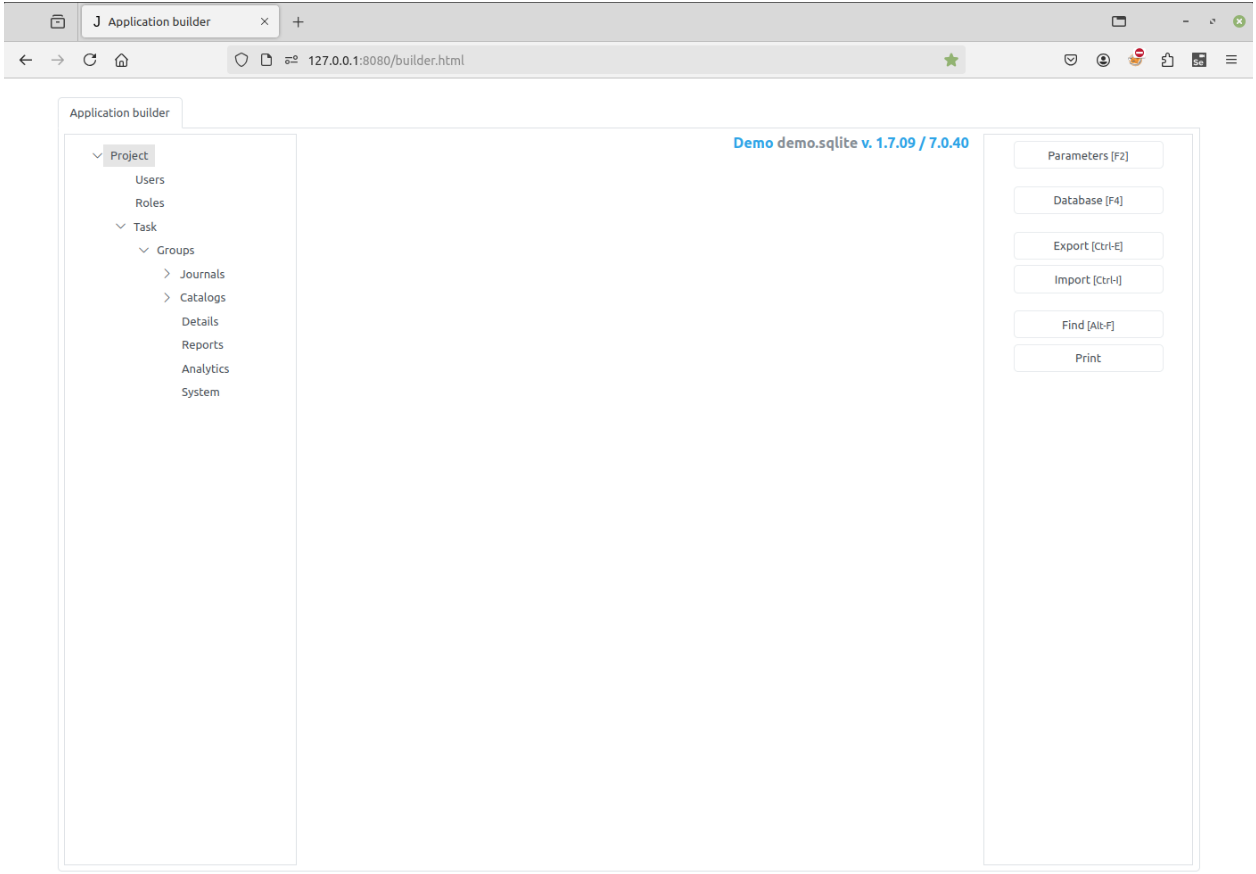
> **备注**
>
> 请注意，*server.py* 必须正在运行

在应用程序构建器页面的左侧是一个包含项目树的面板。当您选择项目树中的任何节点时，通常在页面的中央部分打开其内容，而页面的底部和右侧会出现允许您修改其内容的按钮。

要查看在应用程序构建器中所做的更改，请转到项目的客户端页面并重新加载它。

6.1 项目管理

应用程序构建器首次运行后，或在项目树中选择 **项目 (Project)** 节点时，应用程序构建器页面将如下所示：



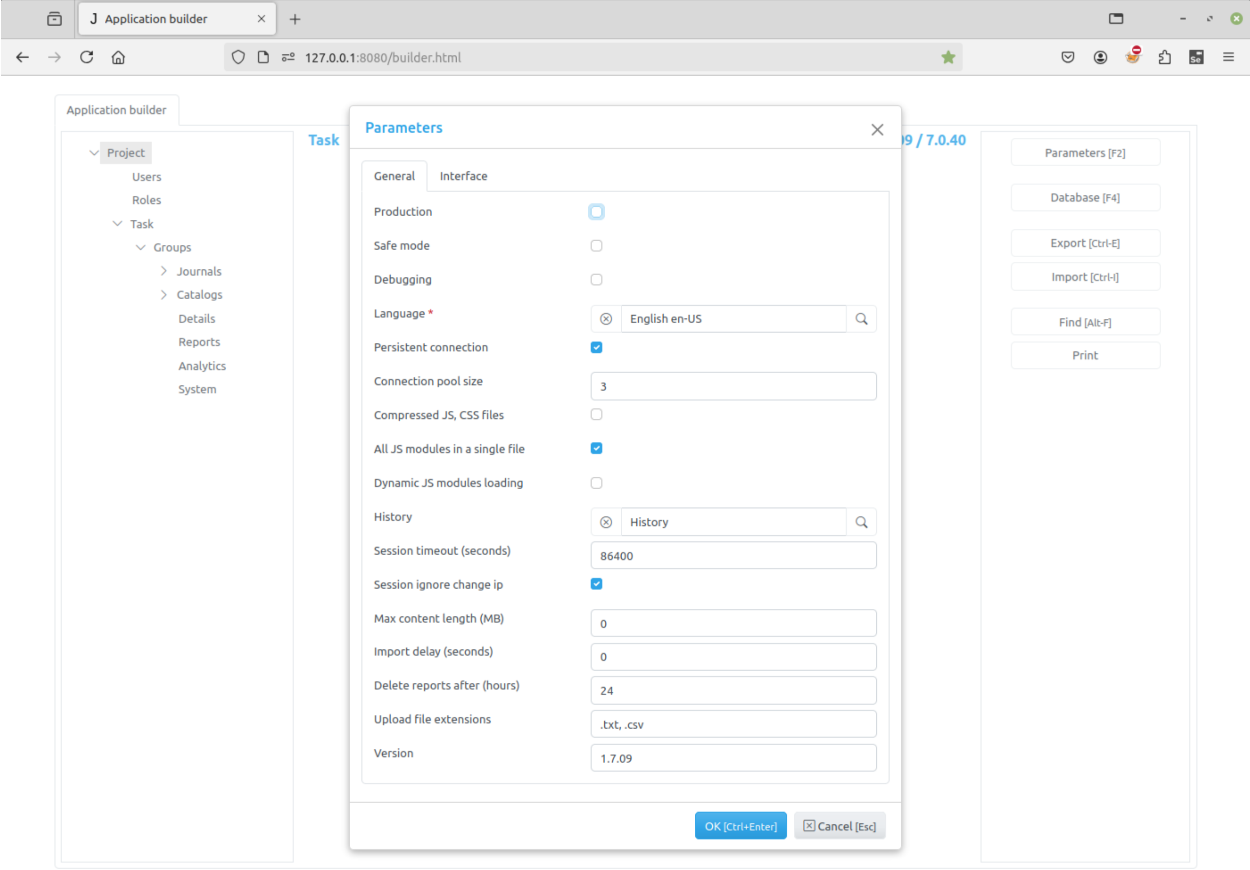
> **重要**
>
> 如右上角所示，显示应用程序的名称、使用的数据库名称、应用程序版本号以及 Jam.py 框架的 版本号。

点击下面的链接以了解页面右侧面板中按钮的用途。

6.1.1 参数

点击 **参数 (Parameters [F2])** 按钮后，将出现参数对话框。它有两个选项卡：**常规 (General)** 和 **界面 (Interface)** 。

常规选项卡



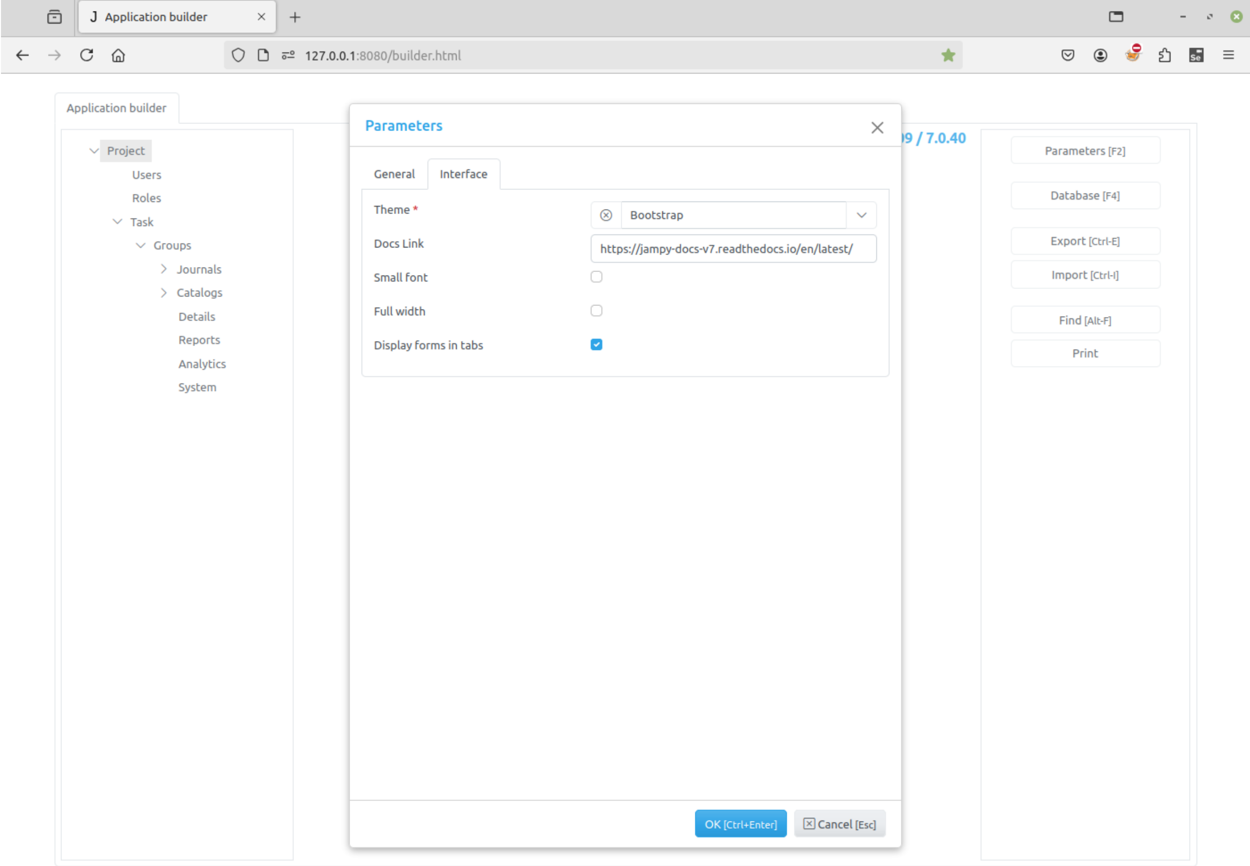
在常规选项卡上，您可以指定项目的常规参数：

- **生成环境 (Production)** - 如果选中此复选框，则禁用对应用程序构建器的访问。要启用访问，应从应用程序文件夹中删除 `builder.html` 文件。
- **安全模式 (Safe mode)** - 如果启用安全模式，用户需要身份验证才能在系统中工作。（请参阅用户 和角色 ）
- **调试 (Debugging)** - 如果选中此复选框，当服务器上发生错误时，将调用 Werkzeug 库调试器。
- **显示 SQL 语句 (Show SELECT SQL)** - 如果选中此复选框，将显示 SELECT SQL 语句。
- **语言 (Language)** - 使用它打开语言对话框。请参阅语言支持
- **持久连接 (Persistent connection)** - 如果选中此复选框，应用程序将创建连接池，否则在执行 sql 查询之前创建连接。
- **连接池的大小 (Connection pool size)** —服务器数据库连接池的大小。
- **压缩 JS，CSS 文件 (Compressed JS, CSS files)** - 如果选中此按钮，当加载 *index.html* 页面时，服务器将返回压缩后的 *js* 和 *css* 文件。
- **单文件的 JS 模块 (All JS modules in a single file)** - 如果未选中此复选框，应用程序将在项目的 *js* 文件夹中为任务树 中的每个实体项生成一个 javascript 文件，该文件包含对应实体项在客户端模块中的代码，其名称为 *item_name.js* ，其中 *item_name* 是对应的实体项的名称。否则，应用程序将生成一个名为 *task_name.js* 的 javascript 文件，其中 *task_name* 是实体项任务 的名称（例如 *demo.js* ），该文件将包含所有实体项的 javascript 代码，除了那些在实体编辑器对话框 中选中 **外部的 js 模块 (External js module)** 复选框的实体项（将为它们创建单独的文件）。

- **动态加载 JS 模块 (Dynamic JS modules loading)** - 如果未选中此复选框且应用程序生成多个 javascript 文件，则在运行应用程序时，只会加载名为 *task_name.js* 的文件。所有其他文件必须动态加载。请参阅使用模块 。
- **历史数据项 (History item)** - 指定将存储更改历史的数据项，请参阅保存用户创建的审计跟踪/更改历史记录
- **会话超时 (Session timeout (seconds))** - 允许会话在过期之前不活动的秒数。
- **会话忽略 ip(Session ignore change ip)** - 如果为 false，则会话仅在从创建会话的同一 ip 地址访问时有效。
- **最大内容长度 (Max content length (MB))** - 使用它限制请求服务器的总内容长度，以兆字节为单位。
- **导入延迟 (Import delay (seconds))** - 如果设置，在 导入元数据时，应用程序将在更改项目的数据集之前先等待参数中设置的秒数，否则它等待 5 分钟或直到当前进程中所有以前的服务器请求被处理之后，再导入数据。
- **多久后删除报表 (Delete reports after (hours))** - 如果指定了值，则位于 static/reports 文件夹中的已生成的报表将在指定的小时数之后被删除。
- **上传文件的扩展名 (Upload file extensions)** - 是一个可接受的字符串 ，它定义了可以通过任务的*upload* 方法上传到服务器的文件类型。系统禁止上传不匹配这些类型的文件。
- **版本 (Version)** —在此处指定项目的版本。

> **备注**
>
> 要使更改的 **连接池大小 (Connection pool size)** 或 **持久连接 (Persistent connection)** 参数生效，必须重新启动服务器应用程序。

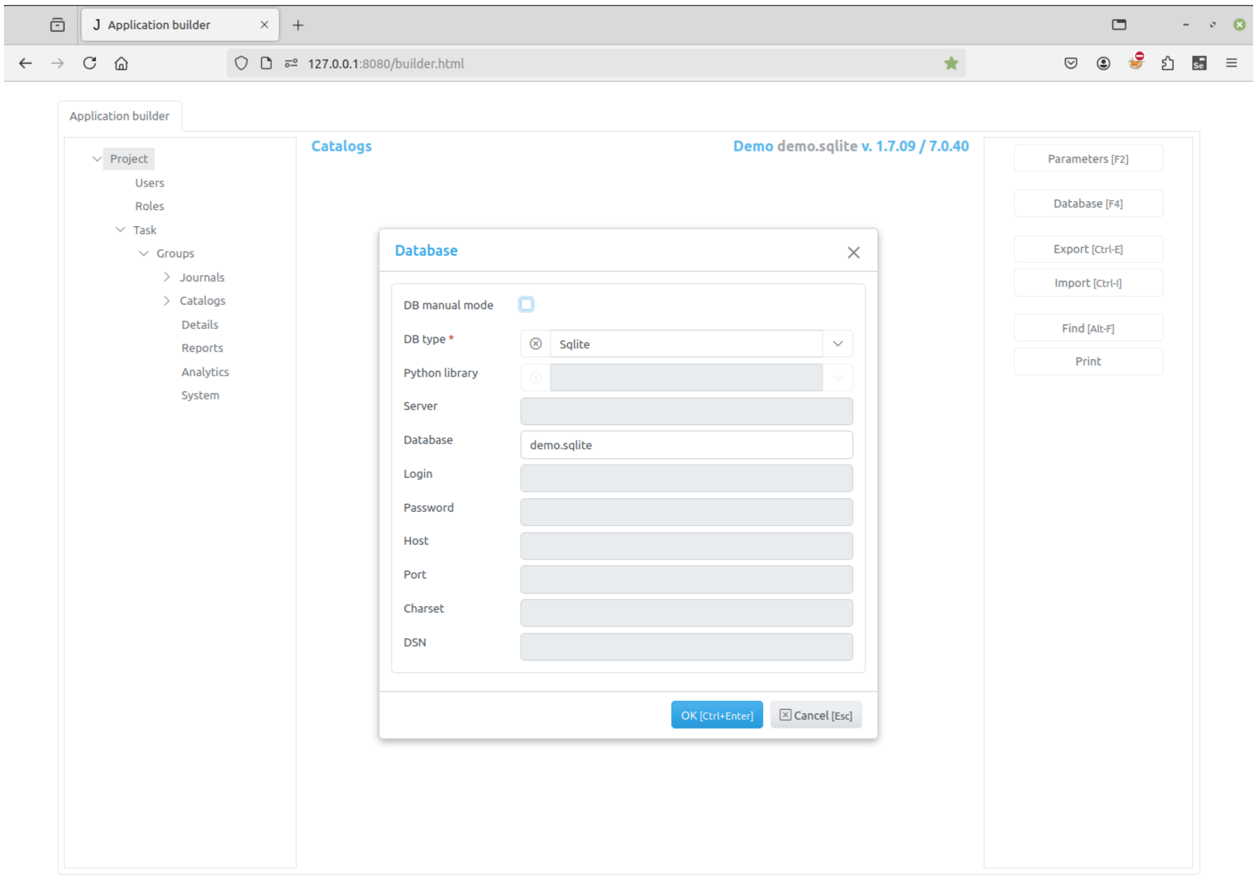
界面选项卡



在界面选项卡上，您可以指定项目的界面参数：

- **主题 (Theme)** - 使用此参数从预定义主题中选择项目使用的主题
- **文档链接 (Docs Link)** - 使用此参数指定文档位置（待定）
- **小字号 (Small font)** - 如果选中此按钮，默认字体大小将为 12px ，否则为 14px
- **全部宽度 (Full width)** - 如果选中此按钮，项目将填充页面宽度，没有左右边距
- **在选项卡中打开表单 (Display forms in tabs)** - 如果选中此按钮，表单将在选项卡中打开

6.1.2 数据库



在此对话框中显示项目的数据库参数。当它们被更改并点击"确定 (OK)"按钮时，应用程序构建器将检查与数据库的连接，如果连接失败，将显示错误。

> **备注**
>
> 当任何 **数据库 (Database)** 参数被更改时（除了 **数据库手动更新 (DB manual update)** ），必须重新启动服务器应用程序才能使更改生效。

如果 **数据库手动更新 (DB manual update)** 复选框未勾选（默认），那么当保存对有关联的数据库表的实体项所做的更改时，该数据库表会自动修改。例如，如果我们在实体项编辑器对话框 中为某个实体添加一个新字段，新字段将被添加到关联的数据库对应的表中。如果勾选此复选框，则不会对数据库中的数据表进行任何修改。

在更新的版本中，**数据库手动更新 (DB manual update)** 已重命名为 **数据库手动模式 (DB manual mode)** 。

> **备注**
>
> 使用此选项时请务必非常小心。

数据库设置示例

改编自 Jam.py 设计技巧

Jam.py 支持许多不同的数据库服务器。例如 PostgreSQL 、MariaDB 、MySQL 、MSSQL 、Oracle 、Firebird 、IBM 、SQLite 以及使用 SQLCipher 的 SQLite 。

如果您正在开发一个小型项目或不打算在生产环境中部署的项目，
SQLite 通常是最佳选择，因为它不需要运行单独的服务器。然而，SQLite 与其他数据库有许多不同之处，

因此如果您正在开发一个重要的项目，建议您使用与在生产环境中相同的数据库进行开发。

除了数据库后端，我们还需要确保安装了 Python 数据库绑定。

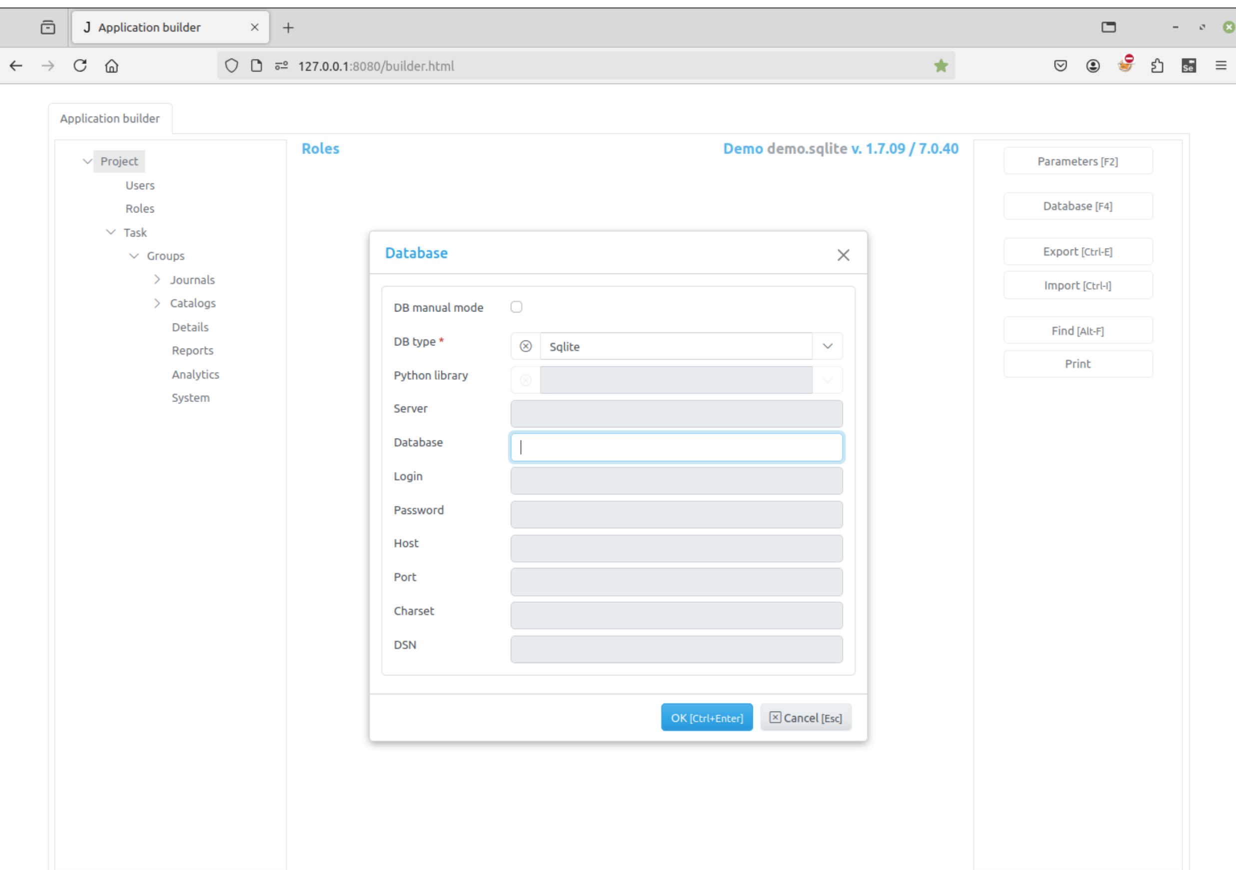
- 如果使用 PostgreSQL，需要 `psycopg2` 或 `psycopg2-binary` 包。
- 如果使用 MySQL 或 MariaDB，Python 2.x 需要 `MySQLdb`。对于 Python 3.x，需要 `mysql-connector-python` 和 `mysqlclient` 包，以及数据库客户端开发文件。

支持在 Windows 上使用 MySQL，请访问 在 Windows 上部署 MySQL （Windows 上的 MySQL 部署）。

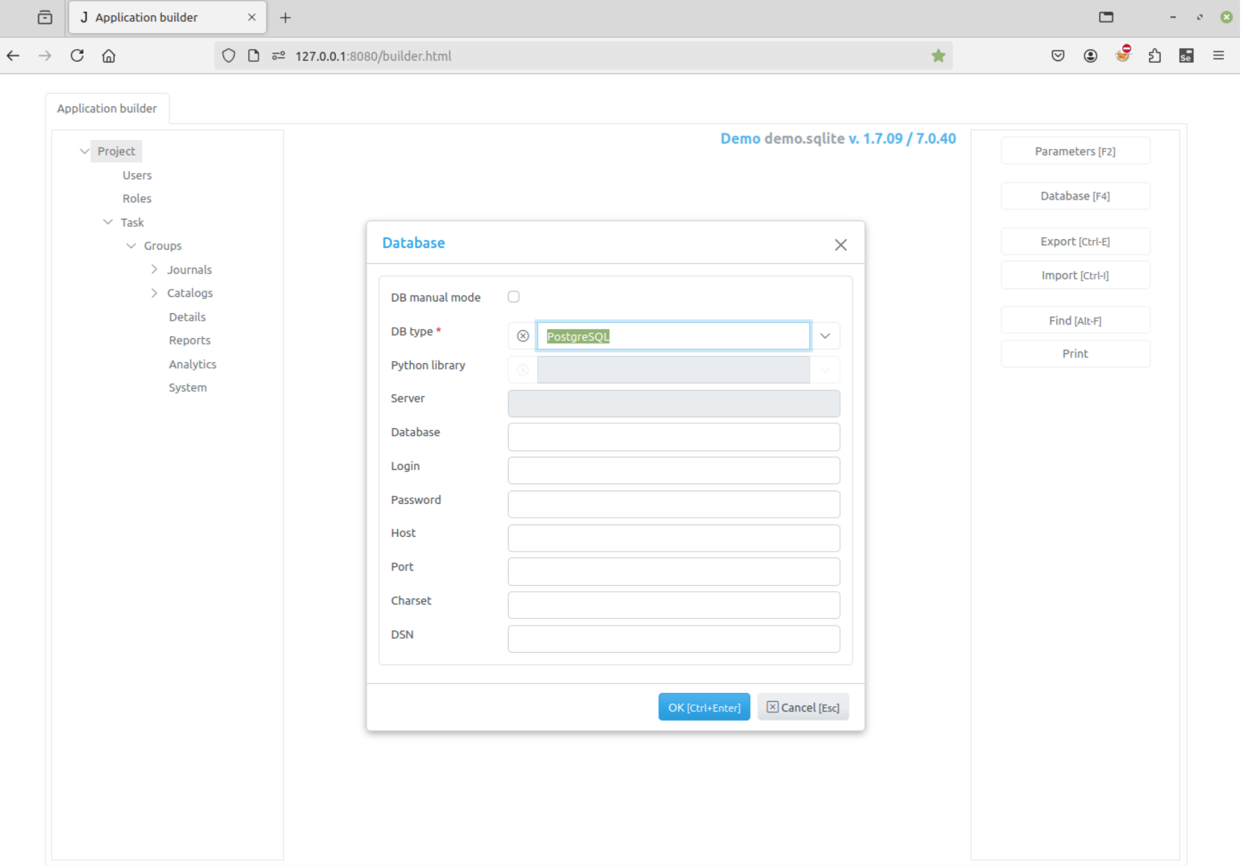
- 如果使用 MSSQL，需要 `pymssql`。对于 MS Windows 上的 ODBC，需要 `pyodbc`。像往常一样配置 ODBC，DSN 作为内容。
- 如果使用 Oracle，需要 cx_Oracle 以及 Python 头文件（开发文件）。
- 如果使用 SQLCipher ，Linux 需要 `sqlcipher3-binary` 包。Windows 有独立的 DLL 可用。
- 如果使用 IBM （待定），需要 `ibm_db` 和 `ibm_db_dbi` 包。
- 如果使用 Firebird，需要 `fdb` 包。
- 如果使用 Databricks ，需要 `databricks-sql-connector` 包。

尽管 Jam.py 支持上述所有数据库，但不能保证支持某些特定和/或专有的数据库功能。这里我们列出几个经过测试的数据库：

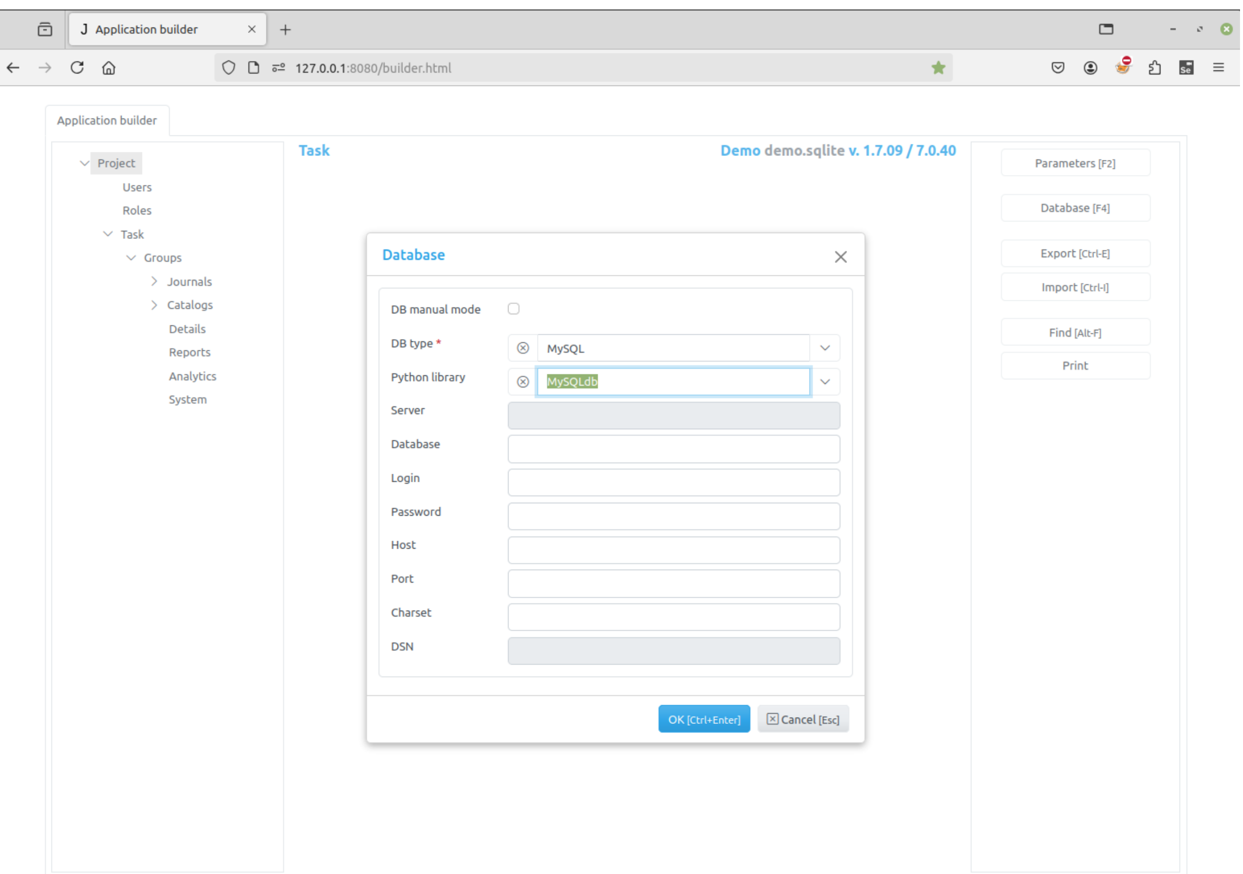
SQLite



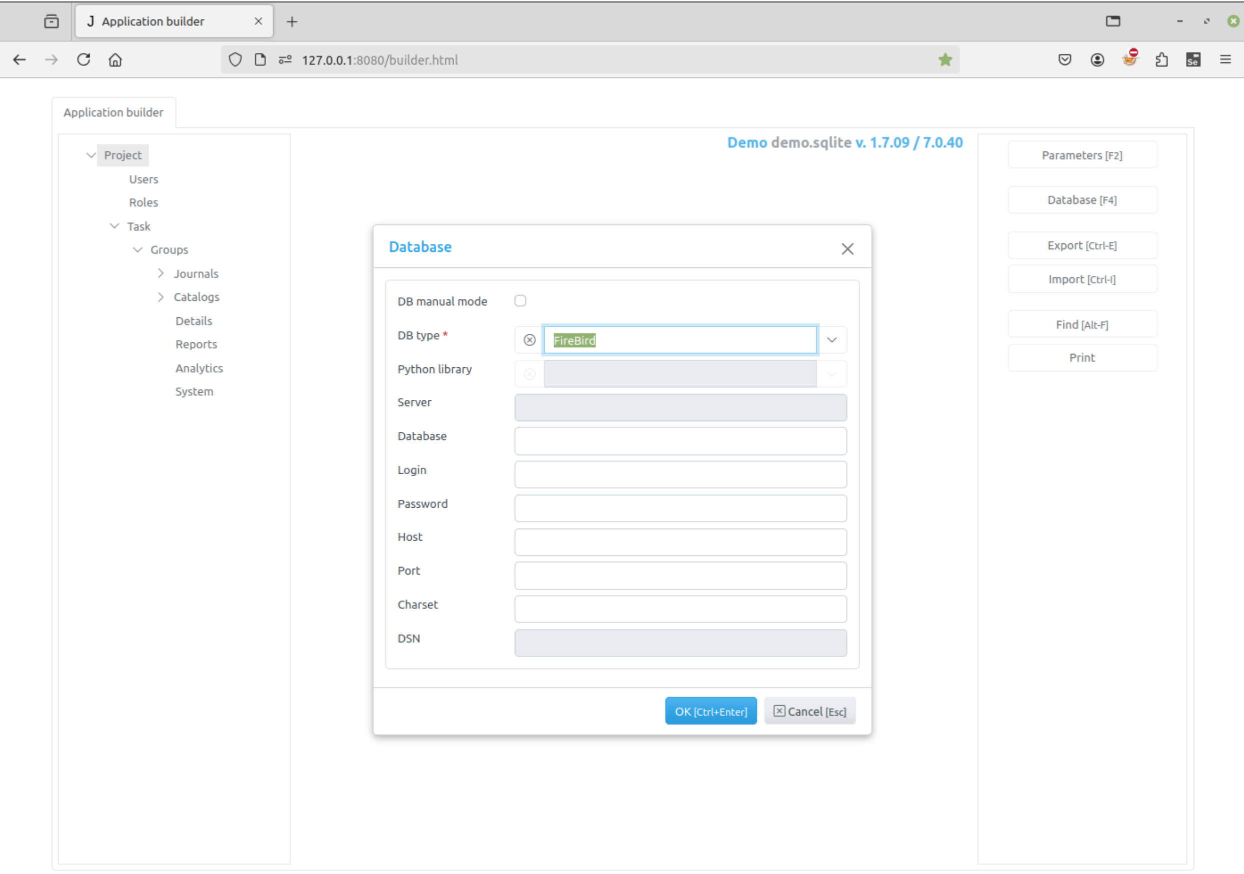
PostgreSQL



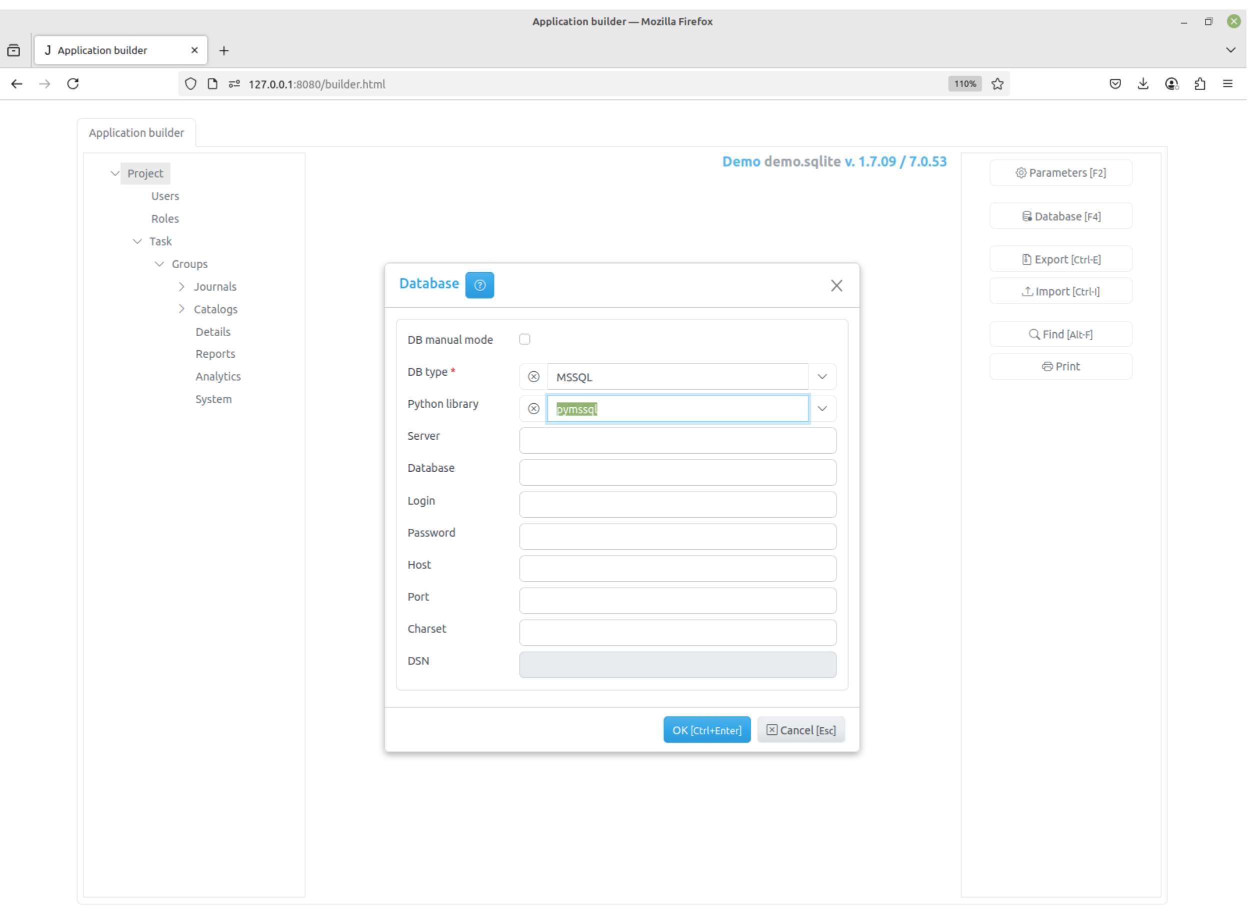
MySql



FireBird

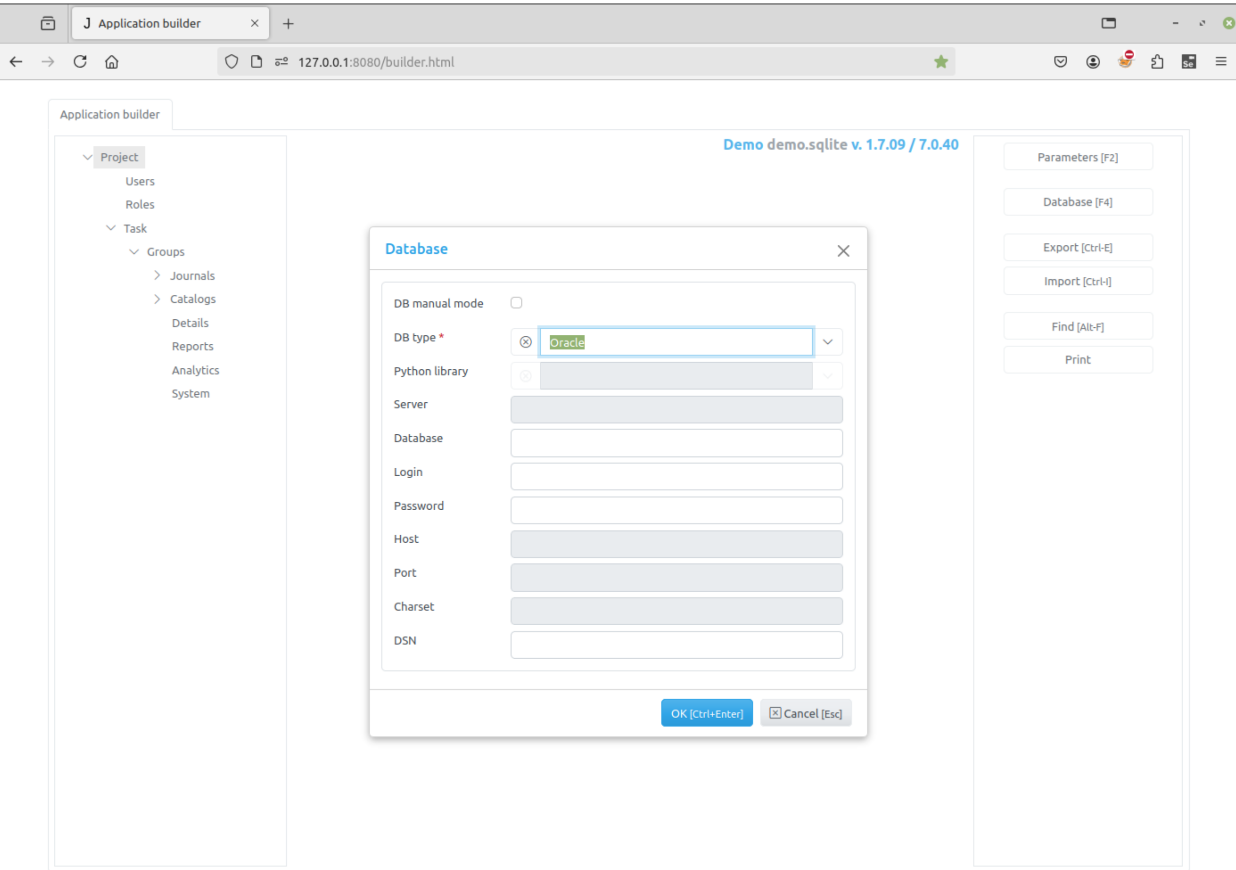


MSSQL

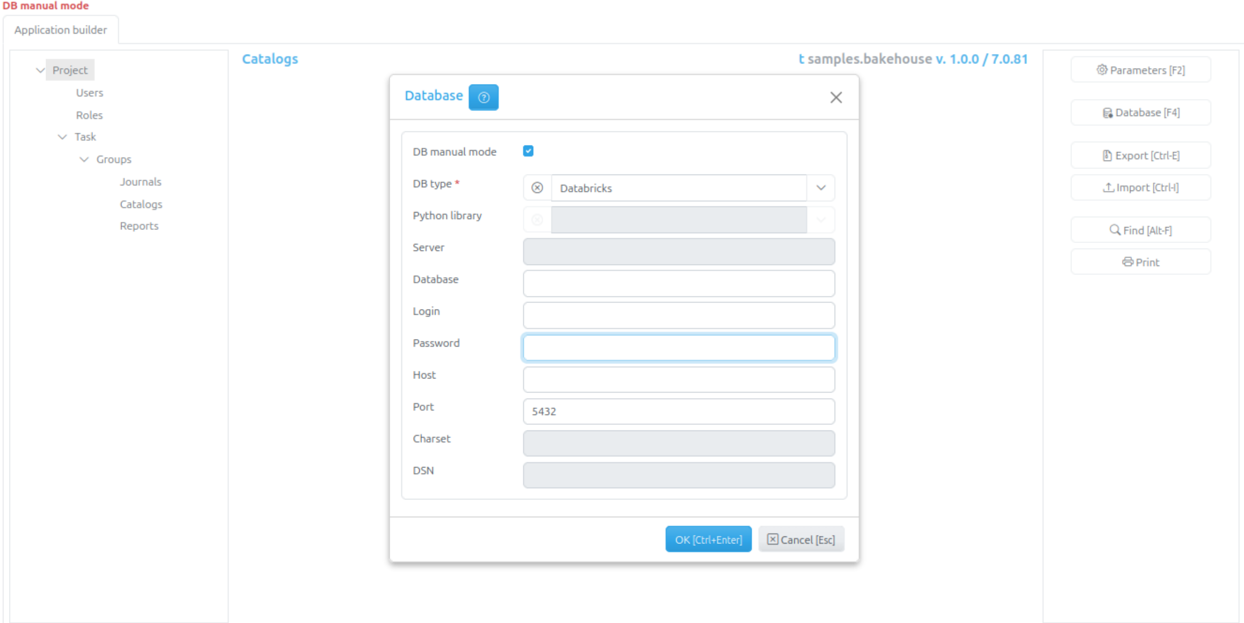


> **备注**
>
> DSN 示例：`DRIVER={SQL Server}; SERVER=localhost\MSSQLSERVER01; DATABASE=master; Trusted_Connection=yes;`

Oracle



Databricks



备注

`Database`（数据库）字段中，将 Databricks 数据库的工作区和数据库模式名称按“workspace.database”形式输入到 数据库 (`Database`) 字段中。这在屏幕截图上显示为右上角的“samples.bakehouse”。

并非所有 DDL 都受支持。

6.1.3 导出

点击 **导出 (Export [Ctrl-E])** 按钮将项目的 元数据导出到一个 zip 文件。

另请参阅

导入

元数据文件

如何将开发迁移到生产环境

6.1.4 导入

使用 **导出 (Import [Ctrl-I])** 按钮从 zip 文件导入项目的 元数据。

另请参阅

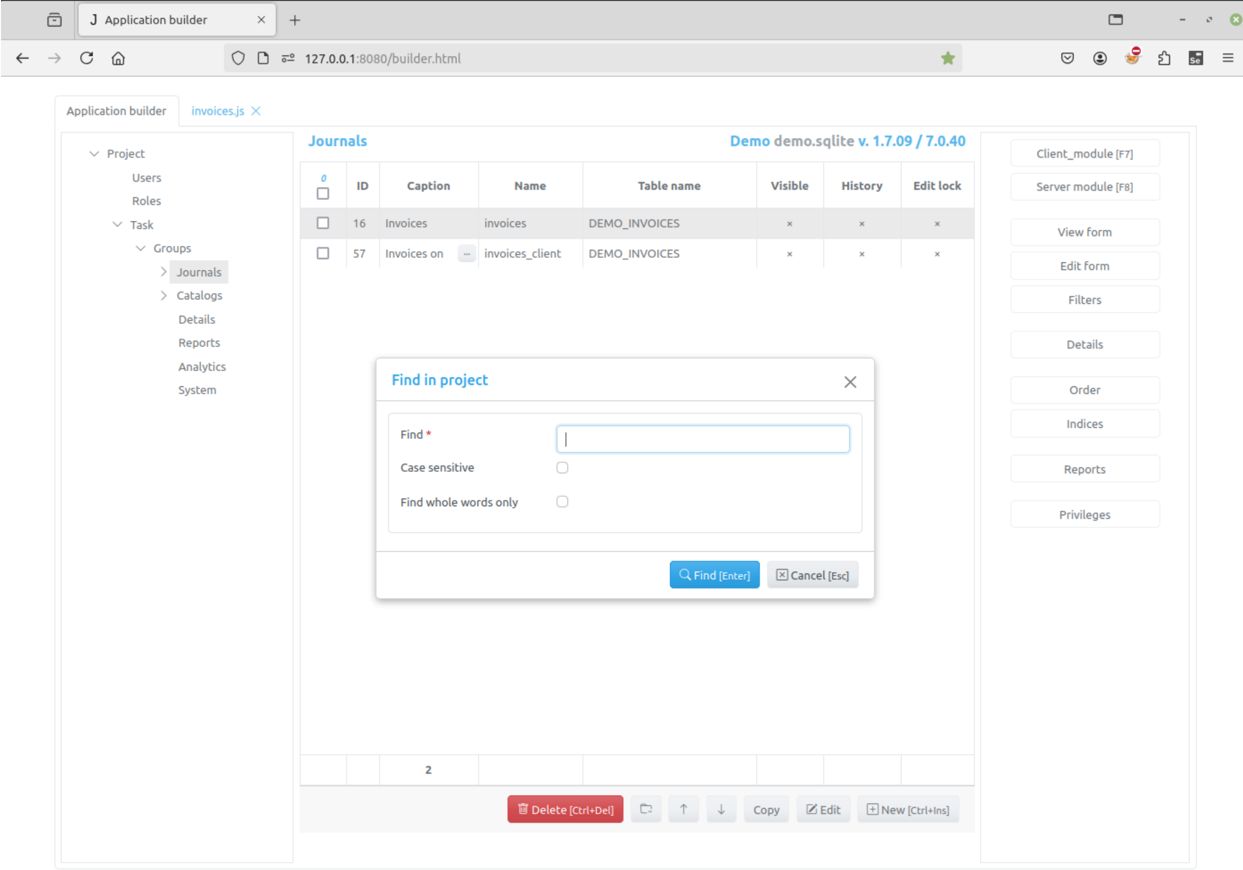
导出

元数据文件

如何将开发迁移到生产环境

6.1.5 查找

点击 **查找 (Find [Alt-F])** 按钮可以在项目的所有模块中搜索字符串。



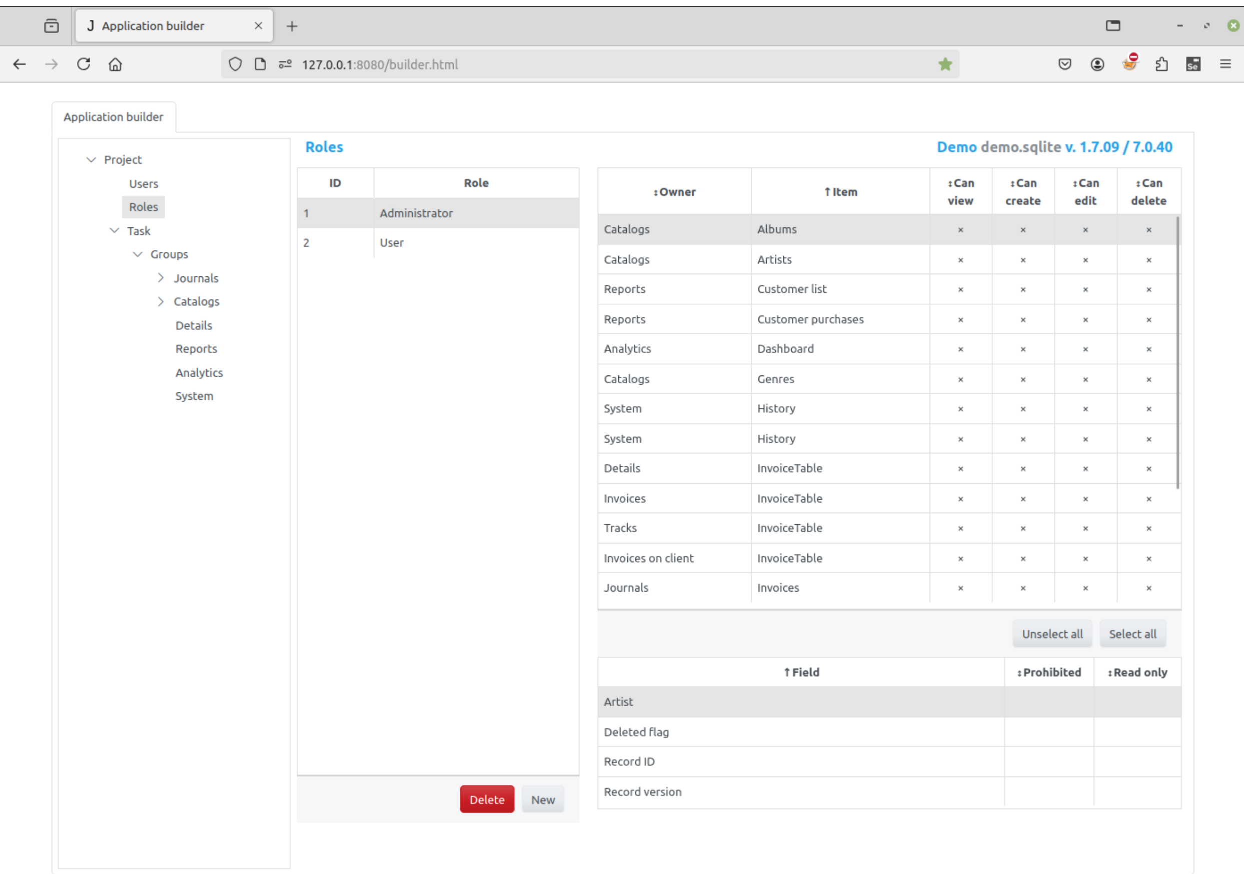
6.1.6 打印

按 **Print** 按钮打印项目的所有模块。

6.2 角色

在项目树中选择“角色 Roles”节点，以创建和修改定义用户权限的角色。必须为每个用户分配项目中定义的某个角色。角色定义了用户查看、创建、修改和删除数据的权利。

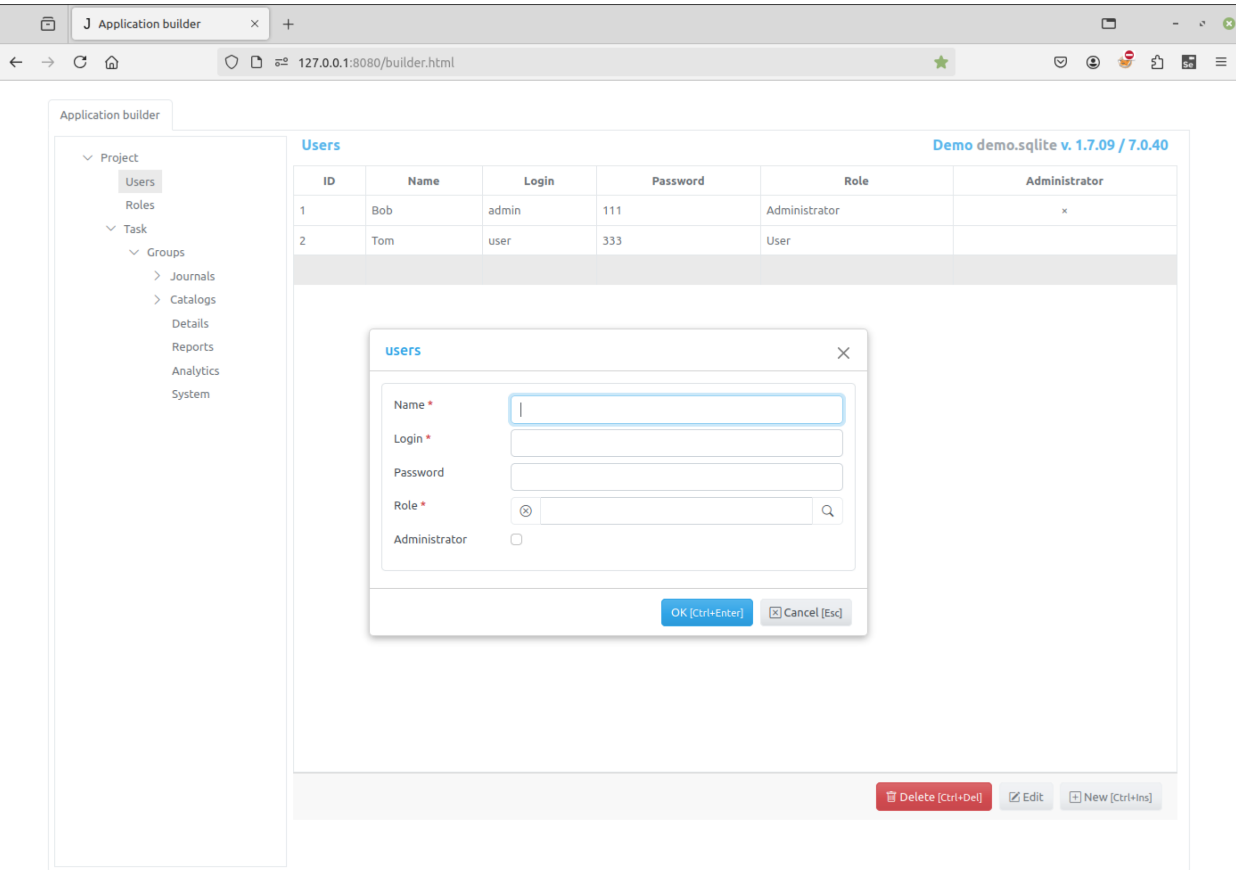
要添加或删除角色，请使用“新建 (New)”和“删除 (Delete)”按钮。要设置角色的权限，请在角色列表中选择角色，并在实体项相应列的单元格中用鼠标点击以添加或移除选中标记：**查看 (View)**、**创建 (Create)**、**编辑 (Edit)**、**删除 (Delete)**。



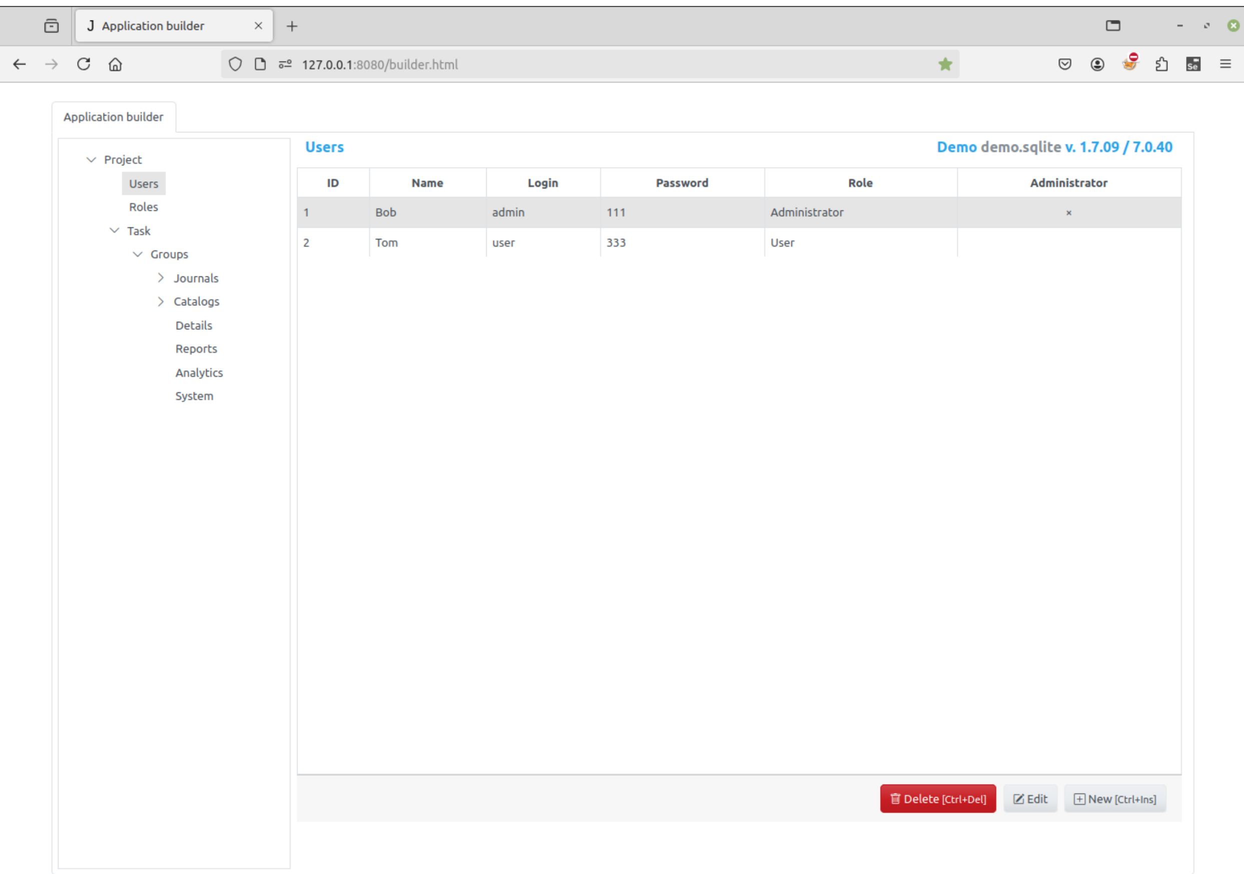
6.3 用户

如果在项目参数 中勾选了 **安全模式 (Safe mode)** 复选框，用户需要在系统中进行身份验证才能工作。

但在此之前，用户必须在框架中注册。要注册用户，请选择“用户 (Users)”节点，点击“新建 (New)”，并填写出现的表单：



- **姓名 (Name)** –用户名
- **登录名 (Login)** - 登录名
- **密码 (Password)** - 密码
- **角色 (Role)** –用户的角色
- **信息 (Information)** - 一些附加信息
- **管理员 (Admin)** - 如果设置了此标志，用户有权在应用程序构建器中工作。

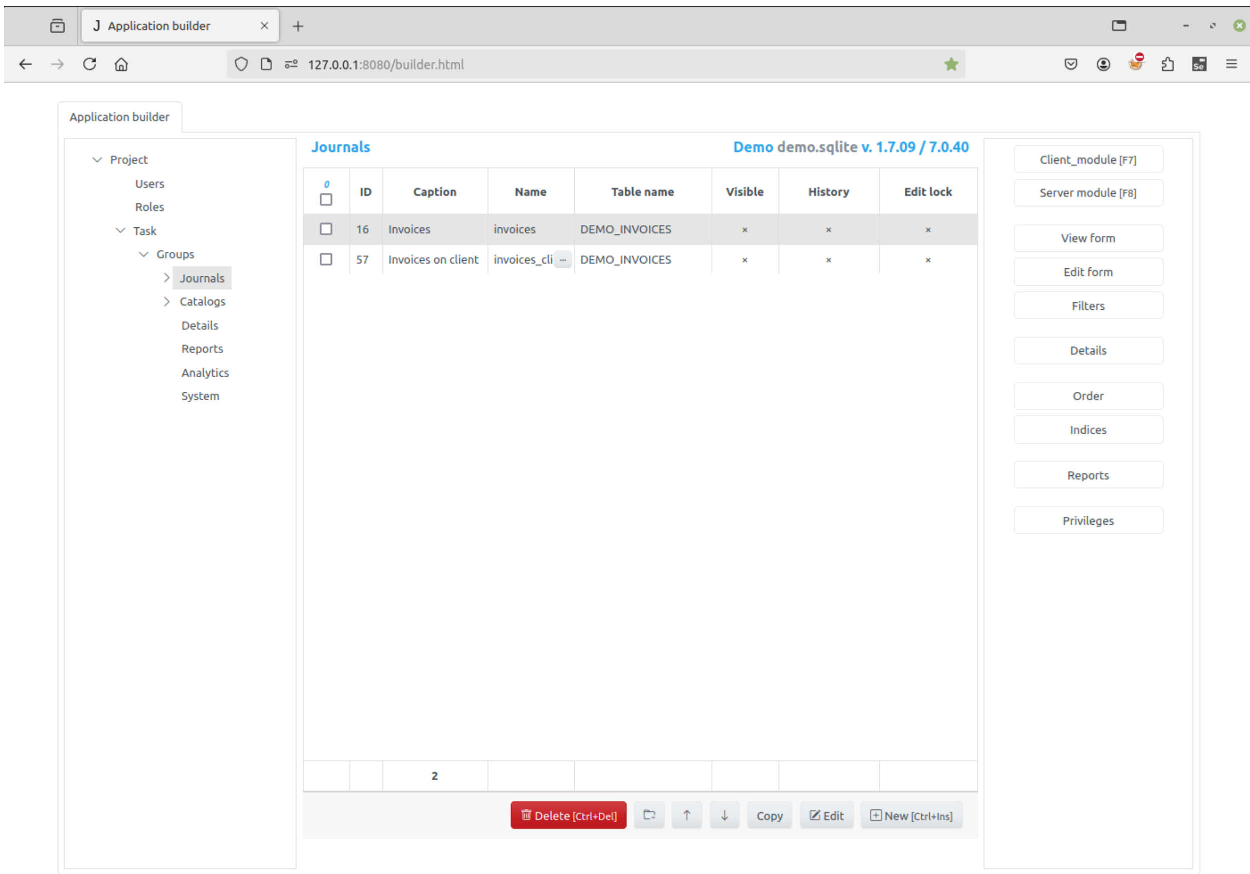


6.3.1 另请参阅

on_login 事件

6.4 代码编辑器

对于项目任务树 中的每个实体项，在应用程序构建器 的右上角有两个按钮：客户端模块 (Client module [F7]) 和 服务器模块 (Server module[F8]) 。



点击这些按钮将打开实体项的客户端模块或服务器模块的代码编辑器。（参见使用模块 ）

在 **编辑器**的左侧是一个包含四个选项卡的信息窗格：

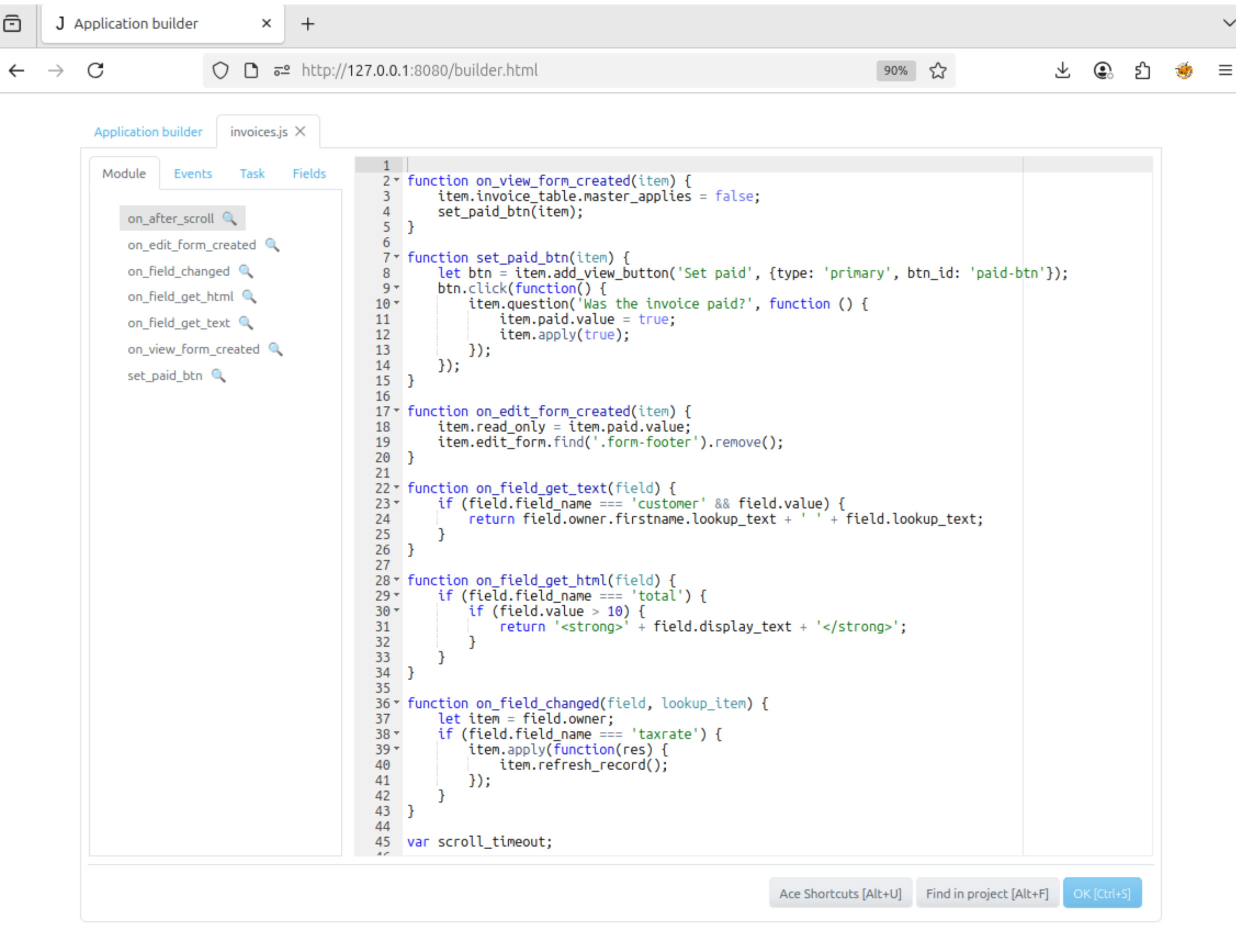
- **(Module)** - 此选项卡显示编辑器中定义的所有事件和函数，双击其中任意一项，即可将光标跳转到对应函数位置。
- **事件 (Events)** - 显示实体项的所有已发布事件，双击可在在光标当前位置为该事件添加包装代码（参见上图中的 on_before_post 事件）。
- **任务 (Task)** - 任务树 ，双击某个节点，可在当前光标位置插入其 item_name。
- **字段 (Fields)** - 当前实体项的字段列表，双击其中一个字段可在当前光标位置输入其 field_name。

要保存更改，请点击 **确定 (OK)** 按钮或按 Ctrl-S。

要搜索项目模块，请点击 **在项目中查找 (Find in project)** 按钮或按 Alt-F 以显示在项目中查找对话框

Jam.py 使用 ace 编辑器 来实现其代码编辑器。新版本的 Jam.py 将使用 Microsoft 的 Monaco 编辑器 。

这里是 ace 编辑器的键盘快捷键。



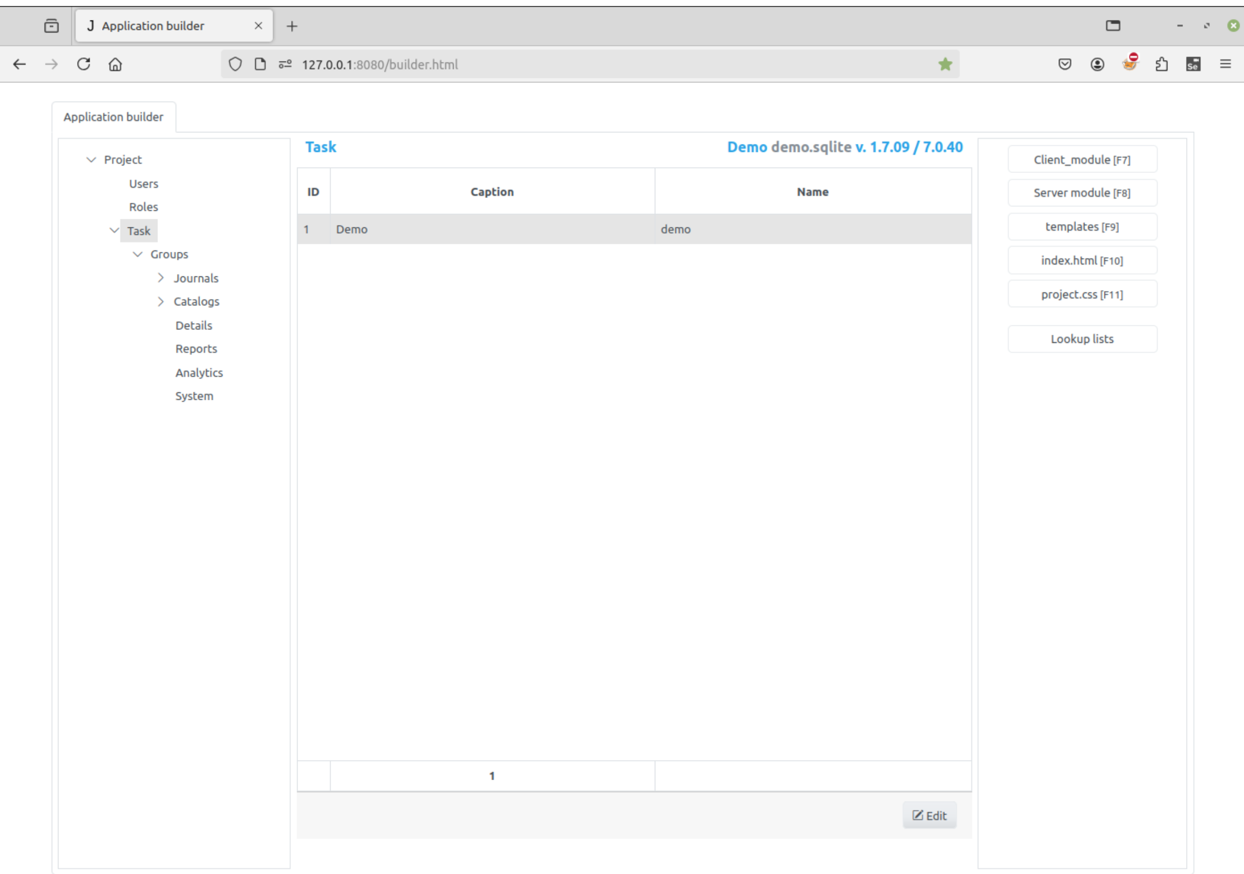
6.5 任务

选择 **任务 (Task)** 节点以访问项目的任务树 的根节点。

使用页面底部的 **编辑 (Edit)** 钮来更改任务的名称和标题。

使用页面右侧面板中的按钮来编辑:

- 任务的 客户端模块 (Client Module [F7]) 和 服务端模块 (Server Module [F8]) , 请参阅使用模块 和代码编辑器
- 项目根文件夹中包含项目页面的 index.html [F10] 文件
- 表单的 模板 (templates [F9]) 文件, 请参阅表单窗体 和代码编辑器
- 项目根文件夹中 **css** 目录下的 project.css [F11] 文件, 请参阅代码编辑器
- 查找列表 (Lookup lists) (查找列表) - 点击按钮打开查找列表 对话框

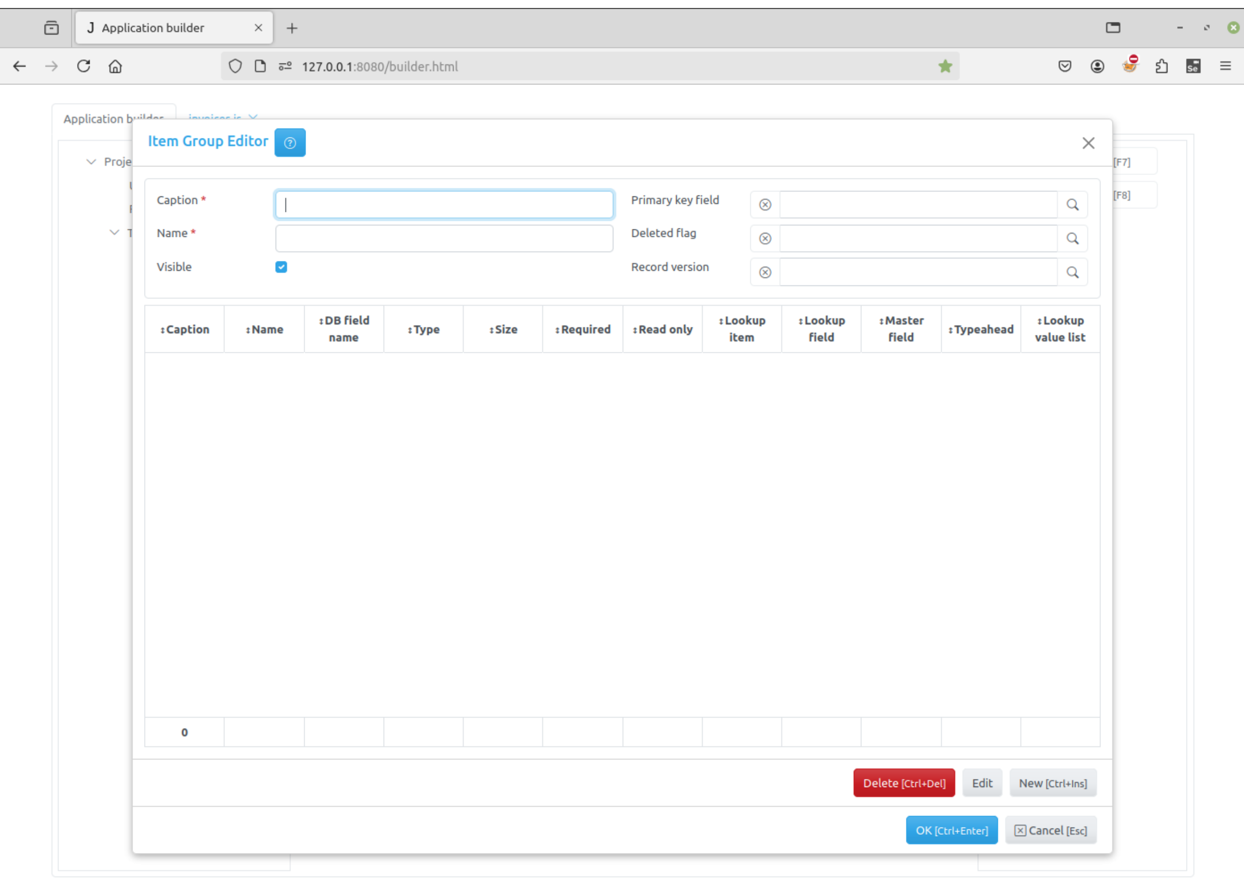


6.6 组

选择名为"任务 (task)"的节点以访问项目的任务树 组。

页面底部有 3 个按钮:

- **删除 (Delete)** - 点击此按钮删除一个空组。
- **编辑 (Edit)** - 点击此按钮修改选定的组，将出现相应的组编辑器对话框。
- **新建 (New)** - 使用此按钮创建新的组。
- **新建报表组 (New Report Group)** - 使用此按钮创建新的报表组。



对于每个组，将显示其自己的编辑器：

6.6.1 实体项的组编辑器

当开发人员想要创建新的实体项组或修改现有实体项组时，会打开 **实体项的组编辑器 (Item Group Editor)** 。请参阅任务树

实体项的组编辑器 (Item Group Editor) 的上半部分有以下字段：

- **标题 (Caption)** - 向用户显示的实体项组的标题。
- **名称 (Name)** - 实体项组的名称，将在编程代码中用于访问实体项组对象。它在项目中应该是唯一的，并且应该是有效的 Python 标识符。
- **主键字段 (Primary key field)** - 通过点击此属性右侧的按钮，您可以指定实体项的主键字段。如果主键字段已在拥有该实体项的组中被定义，则默认会显示在那里，否则您必须先创建此字段。
- **删除标志字段 (Deleted flag field)** - 通过点击此属性右侧的按钮，您可以将字段指定为实体项的删除标志。如果删除标志字段已在拥有该实体项的组中被定义，则默认会显示在那里，否则您必须先创建此字段。
- **记录版本 (Record version)** - 通过点击此属性右侧的按钮，您可以将字段指定为实体项的记录锁定 字段。
- **可见 (Visible)** - 使用此复选框设置实体项的可见属性。该属性的值可在客户端代码中用于创建菜单项等。

在 **实体项的组编辑器 (Item Group Editor)** 对话框的中心部分有一个表格，其中包含为该组内的实体项定义的字段列表。对于该组将拥有的所有实体项，这些字段都是公共的。

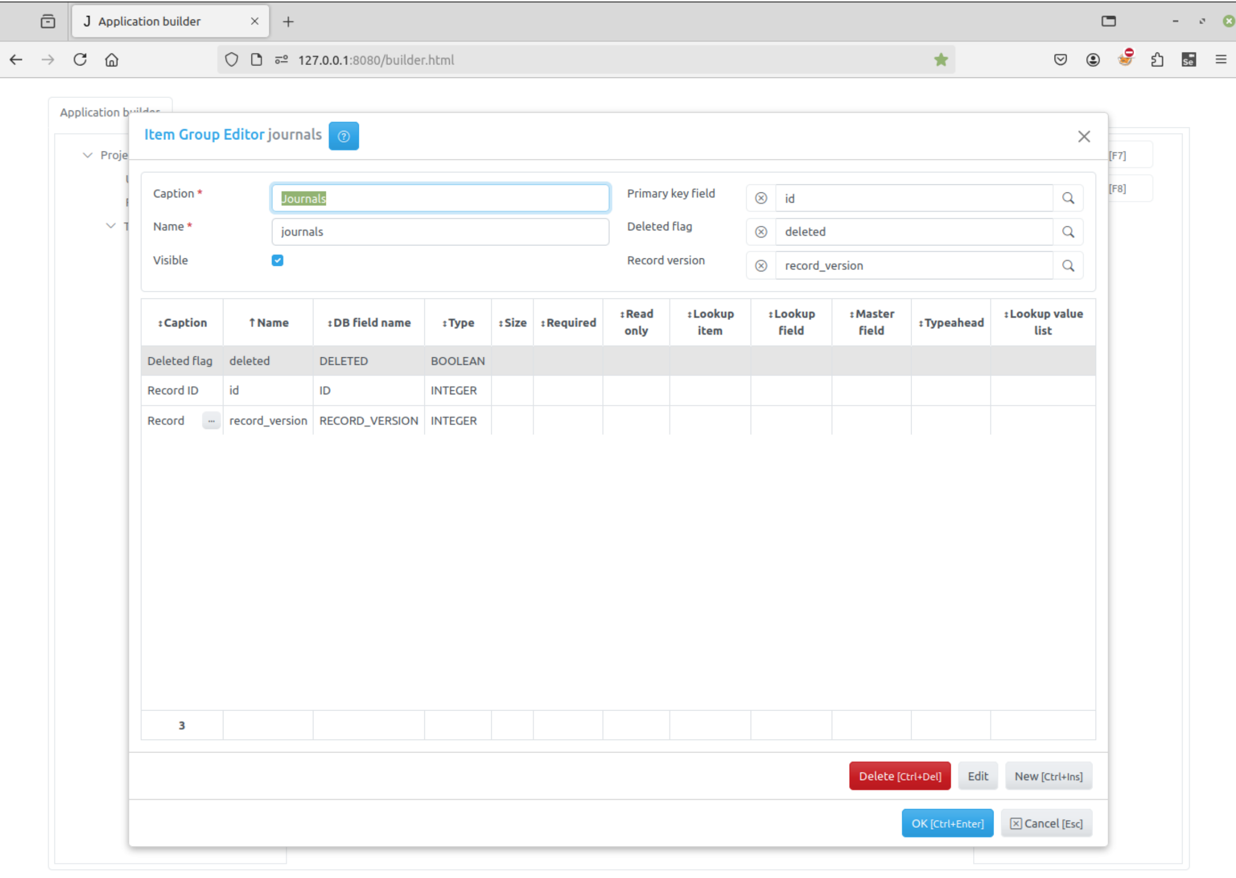
要添加、修改或删除字段，请使用以下按钮：

- **新建 (New)** - 点击此按钮调用字段编辑器对话框 以创建新字段。
- **编辑 (Edit)** （编辑）- 点击此按钮调用字段编辑器对话框 以修改选定的字段。
- **删除 (Delete)** - 点击此按钮删除字段列表中选定的字段。

在对话框表单的右下角有两个按钮：

- **确定 (OK)** - 点击此按钮保存您所做的更改。
- **取消 (Cancel)** - 点击此按钮取消操作。

> **备注**
>
> 您只能在创建新组或编辑空组时创建新字段或修改现有字段，以及设置 **主键字段 (Primary key field)** 和 **删除标志字段 (Deleted flag field)** 的属性。
>
> 对于已经拥有实体项的现有组，您只能更改 **标题 (Caption)** 、 **名称 (Name)** 和 **可见 (Visible)** 的属性。



6.6.2 报表的组编辑器

当开发人员想要创建新的报表组或更改现有报表组时，会打开 **报表的组编辑器 (Report Group Editor)** 。

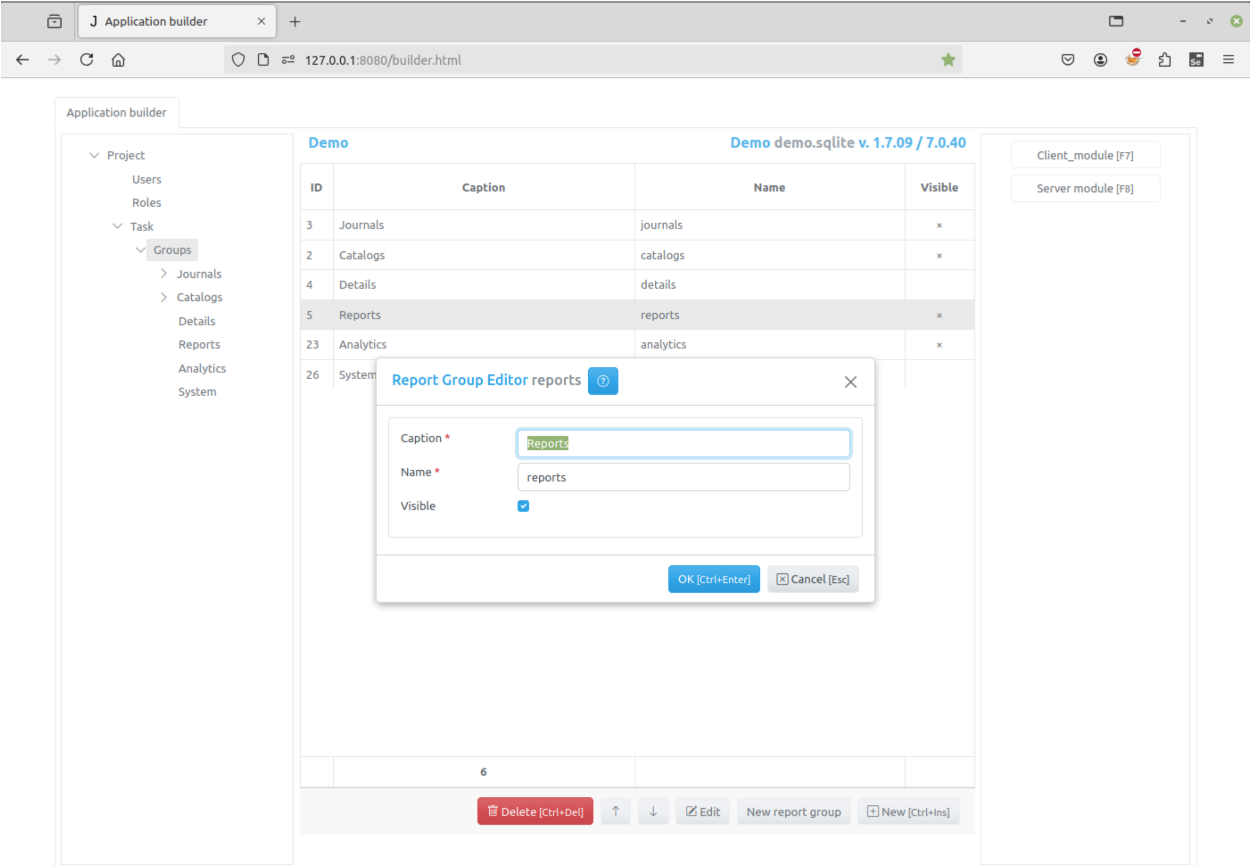
报表的组编辑器 (Report Group Editor) 的上半部分有以下字段：

- **标题 (Caption)** - 向用户显示的报表组的标题。
- **名称 (Name)** - 报表组的名称，将在编程代码中用于访问报表组对象。它在项目中应该是唯一的，并且应该是有效的 Python 标识符。

- **可见 (Visible)** - 使用此复选框设置组的可见属性。该属性的值可在客户端代码中用于创建菜单项等。

在对话框表单的右下角有两个按钮：

- **确定 (OK)** - 点击此按钮保存您所做的更改。
- **取消 (Cancel)** - 点击此按钮取消操作。



6.6.3 实体项编辑器

当开发人员想要创建新的明细表或修改现有明细表时，会打开 **实体项编辑器 (Item Editor)** 。请参阅任务树

* **实体项编辑器 (Item Editor)** 的上半部分有以下字段：

- **标题 (Caption)** - 向用户显示的明细表的标题。
- **名称 (Name)** - 明细表的名称，将在编程代码中用于访问明细表对象。它在项目中应该是唯一的，并且应该是有效的 Python 标识符。
- **数据表名 (Table name)** - 将在项目的数据库中创建的表的名称。在创建实体项时指定此名称，以后不能再更改。
- **主键字段 (Primary key field)** - 通过点击此属性右侧的按钮，您可以指定实体项的主键字段。如果主键字段已在拥有该实体项的组中被定义，则默认会显示在那里，否则您必须先创建此字段。
- **删除标志字段 (Deleted flag field)** - 通过点击此属性右侧的按钮，您可以将字段指定为实体项的删除标志。如果删除标志字段已在拥有该实体项的组中被定义，则默认会显示在那里，否则您必须先创建此字段。
- **记录版本 (Record version)** - 通过点击此属性右侧的按钮，您可以将字段指定为实体项的记录锁定 字段。

- **可见 (Visible)** - 使用此复选框设置项目的可见属性。该属性的值可在客户端代码中用于创建菜单项等。
- **软删除 (Soft delete)** - 当勾选此复选框时，删除方法不会真正地从表中物理删除记录，而是使用此字段将记录标记为已删除。请参阅公共字段 、服务端的 *delete* 方法、客户端的 *delete* 方法。
- **虚拟表 (Virtual table)** - 如果勾选此复选框，将不会创建数据库表。使用此选项可以创建具有内存数据集的数据项，或使用其模块编写代码。此复选框必须在创建项目时设置，以后不能再更改。
- **历史 (History)** - 如果勾选此复选框，应用程序将为此实体项保存用户所做的审计追踪/变更历史记录，请参阅保存用户所做的审计追踪/变更历史记录
- **编辑锁定 (Edit lock)** - 如果勾选此复选框，应用程序将在用户同时编辑记录时使用记录锁定，请参阅记录锁定

在 **实体项编辑器 (Item Editor)** 对话框的中心部分有一个表格，它包含为该实体项定义的字段列表。对于该组将拥有的所有实体项，这些字段都是公共的 。

要添加、修改或删除字段，请使用以下按钮：

- **新建 (New)** - 点击此按钮调用字段编辑器对话框 以创建新字段。
- **编辑 (Edit)** - 点击此按钮调用字段编辑器对话框 以修改选定的字段。
- **删除 (Delete)** - 点击此按钮删除字段列表中选定的字段。

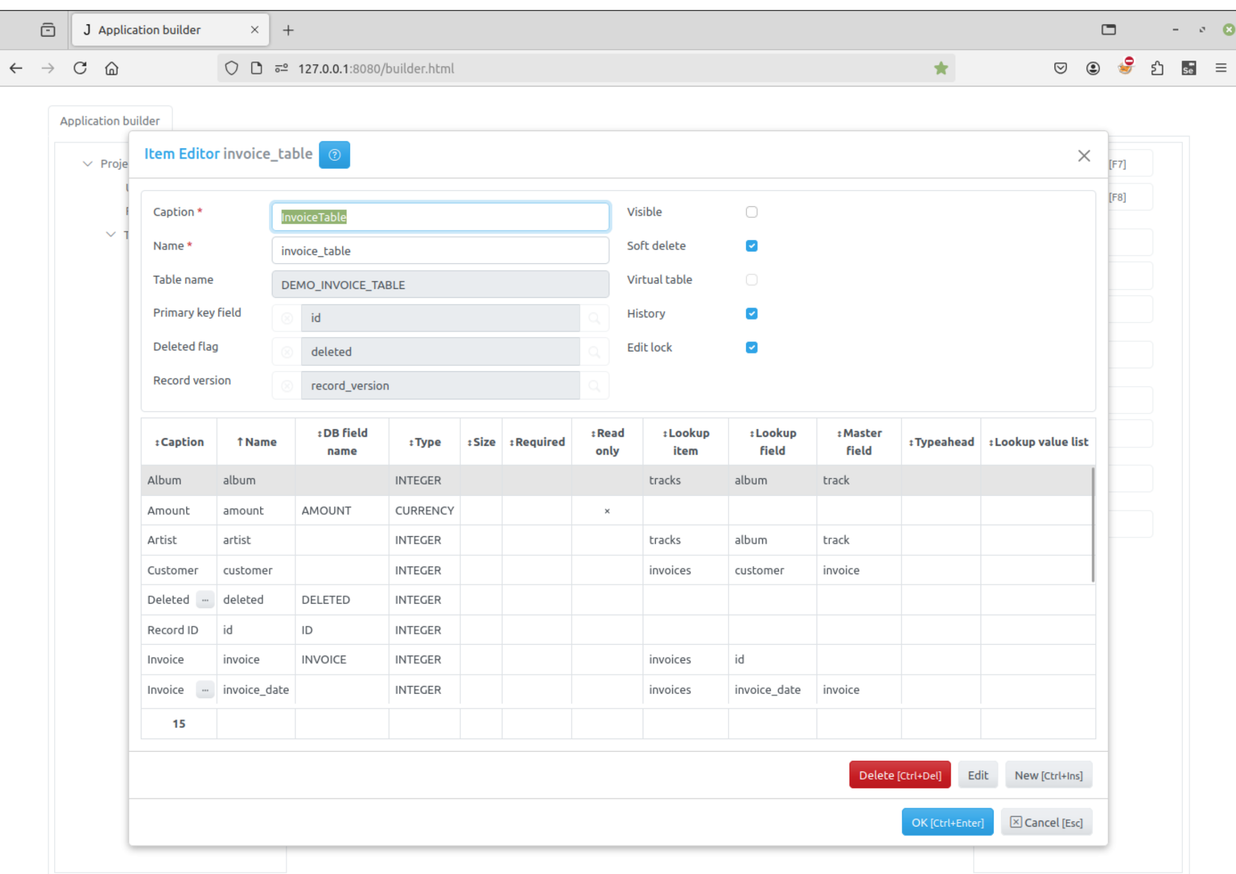
在对话框表单的右下角有两个按钮：

- **确定 (OK)** - 点击此按钮保存您所做的更改。
- **取消 (Cancel)** - 点击此按钮取消操作。

备注

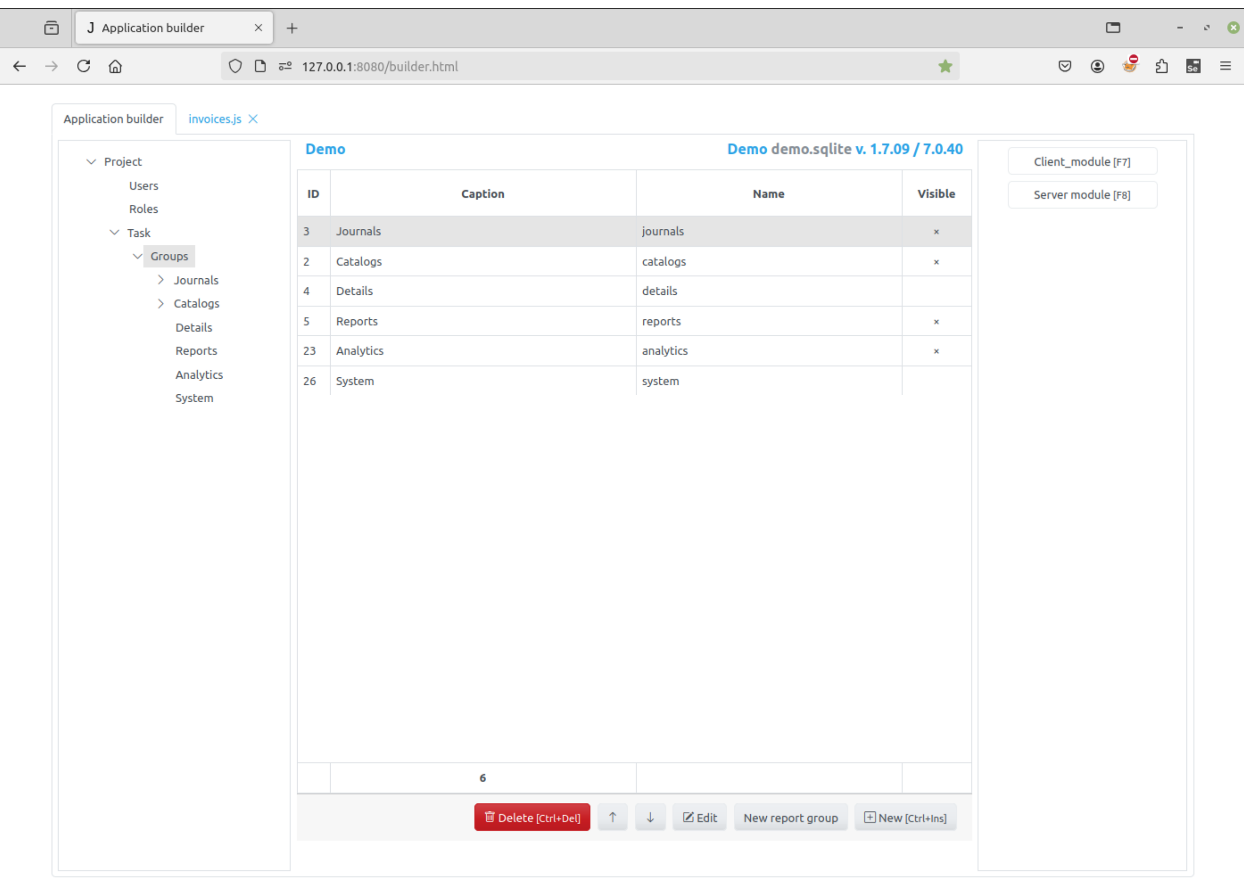
您只能在创建新明细项或编辑空明细项时创建新字段或修改现有字段，以及设置 **主键字段 (Primary key field)** 和 **删除标志字段 (Deleted flag field)** 的属性。

对于已经拥有实体项的现有明细表，您只能更改 **标题 (Caption)** 、**名称 (Name)** 和 **可见 (Visible)** 的属性。



使用页面右侧面板中的按钮编辑所选组的客户端和服务器模块，请参阅：

- 使用模块
- 代码编辑器



6.7 实体项

在项目树中选择一个组节点，以访问该组拥有的实体项，请参阅任务树。

页面底部有 3 个按钮：

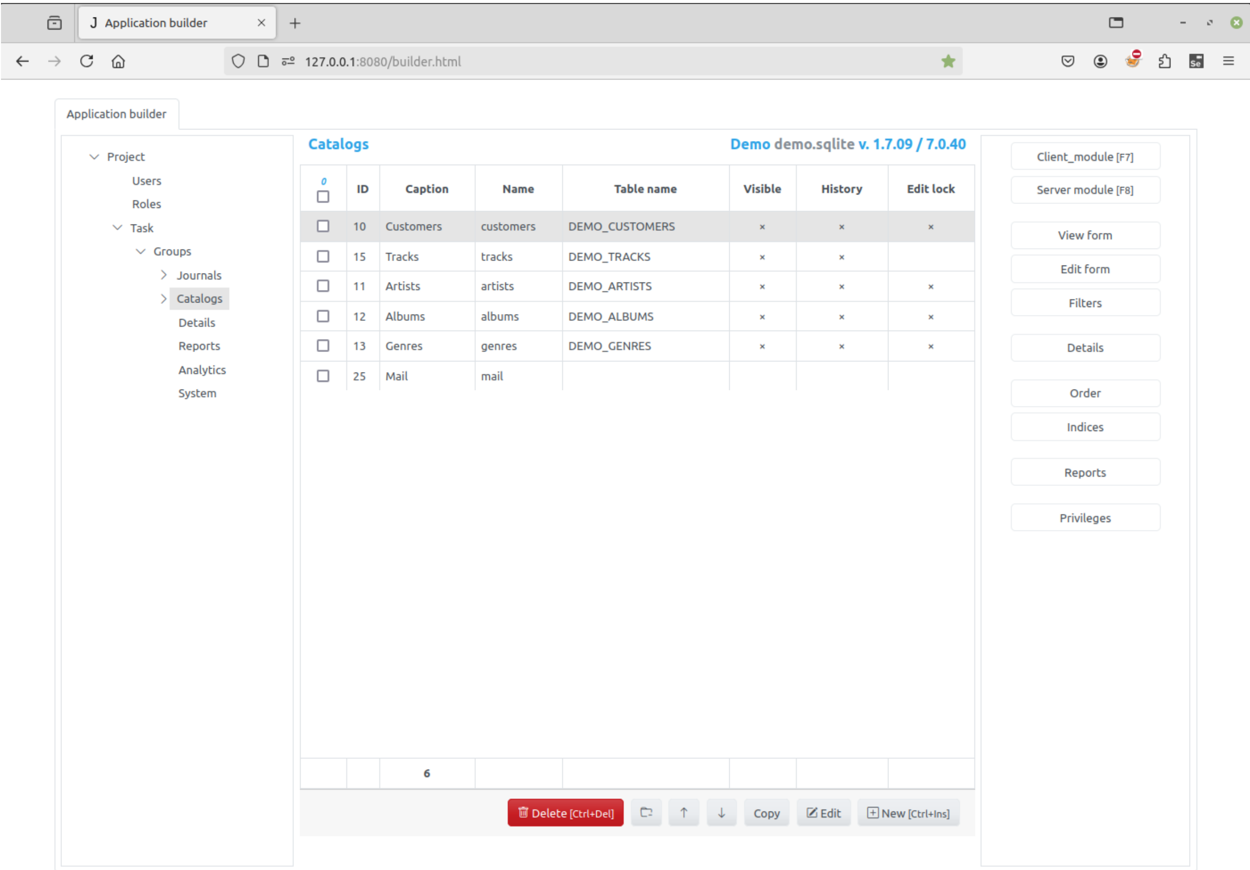
- **新建 (New)** - 点击 New（新建）在实体项编辑器对话框 中创建新的实体项。
- **编辑 (Edit)** - 使用此按钮修改实体项的属性，也可以在实体项编辑器对话框 中添加、更改或删除字段。
- **删除 (Delete)** - 点击此按钮删除实体项及其底层数据表。

您可以使用上下箭头在列表中排列实体项。这可能对创建菜单或以某种方式在网页上显示它们有用。

页面右侧面板有以下按钮：

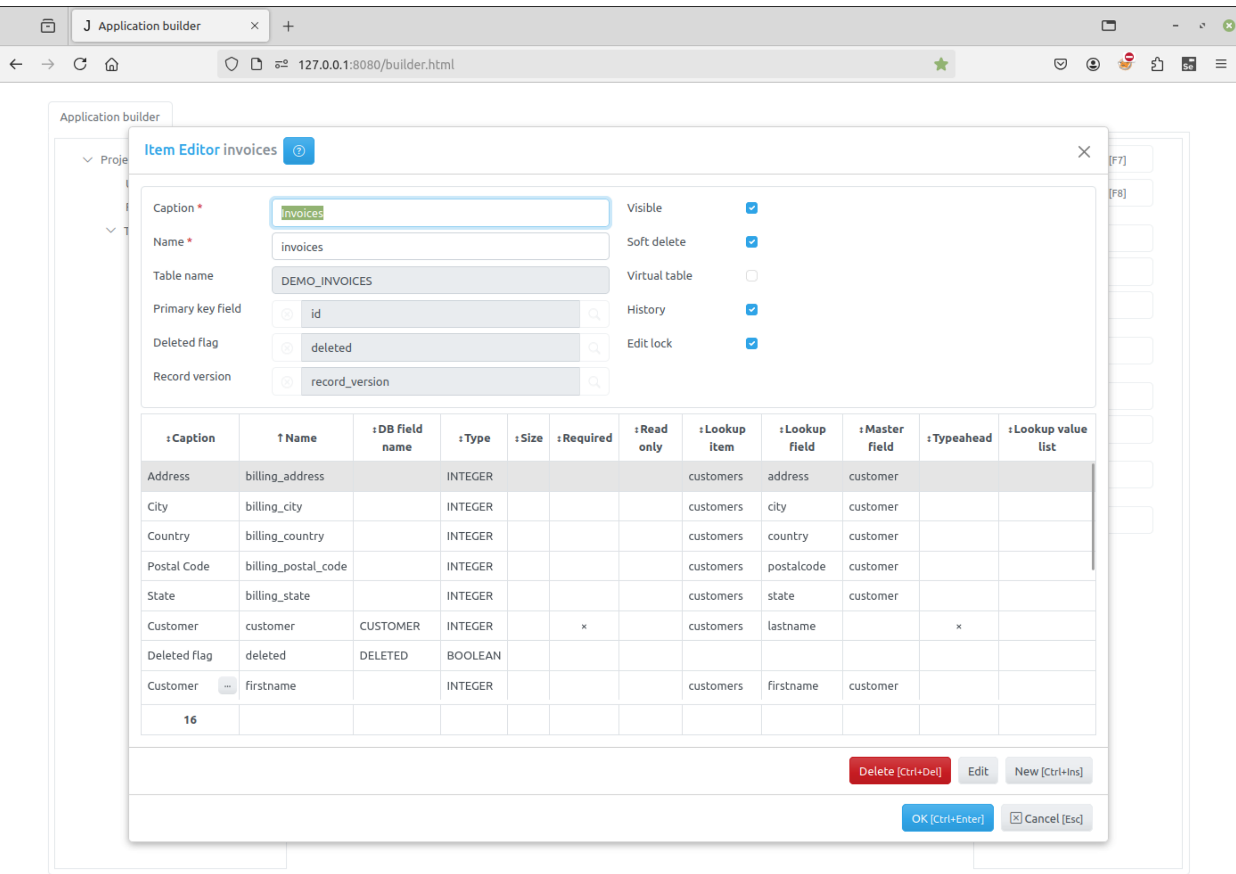
- **客户端模块 (Client module)** - 点击此按钮打开代码编辑器 以编辑实体项的客户端模块，请参阅使用模块 。
- **服务端模块 (Server module)** - 点击此按钮打开代码编辑器 以编辑实体项的服务器模块，请参阅使用模块 。
- **查看表单 (View Form)** - 使用此按钮调用查看表单对话框 以设置客户端的查看表单的显示方式。
- **编辑表单 (Edit Form)** - 使用此按钮调用编辑表单对话框 以设置客户端的编辑表单的显示方式。
- **过滤器 (Filters)** - 使用此按钮调用过滤器对话框 以创建、修改和删除实体项的过滤器。请参阅过滤器 。
- **明细表 (Details)** - 使用此按钮调用明细表对话框 以添加或删除链接到该项的明细表。
- **排序 (Order)** - 使用此按钮调用排序对话框 以指定记录在客户端的默认排序方式。请参阅*open* 方法。

- **(Indices)**（索引）- 点击此按钮打开索引对话框 为实体项的数据库表创建或删除索引。
  - **外键 (Foreign keys)** - 点击此按钮打开外键对话框 为数据库表创建外键。
- **报告 (Reports)** - 点击此按钮打开报告对话框 以指定可为实体项打印的报表。新项目有一个可用于创建下拉按钮以打印报表的功能。
- **权限 (Privileges)** - 点击此按钮打开一个对话框，配置用户的角色对此实体项的权限。



6.7.1 实体项编辑器对话框

开发人员在应用程序构建器的项目树中选择“分组 (Group)”节点，并点击 **新建 (New)** 或 **编辑 (Edit)** 按钮以创建新实体项或修改选定实体项时，会打开 **实体项编辑器对话框 (Item Editor dialog)** 。请参阅实体项 。



实体项编辑器对话框 (Item Editor dialog) 的上半部分有以下字段：

- **标题 (Caption)** - 向用户显示的实体项的标题。
- **名称 (Name)** - 实体项的名称，将在编程代码中用于访问实体项的对象。它在项目中应该是唯一的，并且应该是有效的 Python 标识符。
- **数据表名 (Table name)** - 将在项目数据库中创建的表的名称。此名称在创建实体项时指定，以后不能再更改。
- **主键字段 (Primary key field)** - 通过点击此属性右侧的按钮，您可以指定实体项的主键字段。如果主键字段已在拥有该实体项的组中被定义，则默认会显示在那里，否则您必须先创建此字段。
- **删除标志字段 (Deleted flag field)** - 通过点击此属性右侧的按钮，您可以将字段指定为实体项的删除标志。如果删除标志字段已在拥有该实体项的组中被定义，则默认会显示在那里，否则您必须先创建此字段。
- **记录版本 (Record version)** - 通过点击此属性右侧的按钮，您可以将字段指定为实体项的记录锁定 字段。
- **可见 (Visible)** - 使用此复选框设置实体项的可见属性。该属性的值可在客户端代码中用于创建菜单项等。
- **软删除 (Soft delete)** - 当勾选此复选框时，删除方法不会从表中真正地物理删除记录，而是使用此字段将记录标记为已删除。请参阅公共字段 、服务端的 *delete* 方法、客户端 *delete* 方法。
- **虚拟表 (Virtual table)** - 如果勾选此复选框，将不会创建数据库表。使用此选项可以创建具有内存数据集的数据项，或使用其模块编写代码。此复选框必须在创建实体项时设置，以后不能再更改。
- **历史 (History)** - 如果勾选此复选框，应用程序将为此实体项保存用户所做的审计追踪/变更历史记录，请参阅保存用户所做的审计追踪/变更历史记录

- **编辑锁定 (Edit lock)** - 如果勾选此复选框，应用程序将在用户编辑记录时使用记录锁定，请参阅记录锁定

在 **实体项编辑器对话框 (Item Editor dialog)** 的中心部分，有一个表格，里面包含为实体项定义的字段列表。要添加、修改或删除字段，请使用以下按钮：

- **新建 (New)** - 点击此按钮调用字段编辑器对话框 以创建新字段。
- **编辑 (Edit)** - 点击此按钮调用字段编辑器对话框 以修改选定的字段。
- **删除 (Delete)** - 点击此按钮删除字段列表中选定的字段。

在对话框表单的右下角有两个按钮：

- **确定 (OK)** - 点击此按钮保存您所做的更改。如果未勾选 **虚拟表 (Virtual table)** 复选框，且在项目的:doc:'数据库对话框 </admin/project/database>'中未设置 **数据库手动更新 (DB manual update)** 参数，应用程序将生成并执行 SQL 查询以更新项目数据库中的数据表（对字段所做的更改将应用到表中）。
- **取消 (Cancel)** - 点击此按钮取消操作。

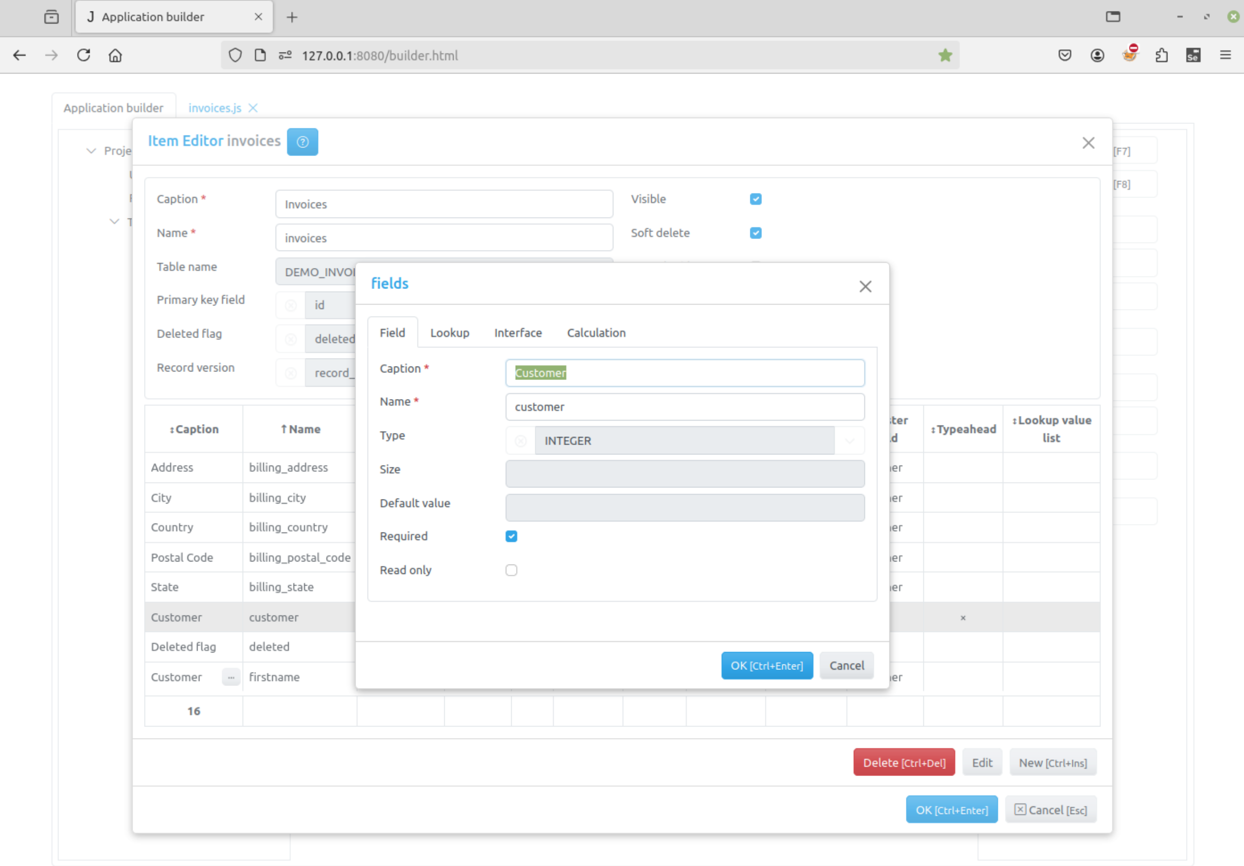
6.7.2 字段编辑器对话框

使用 **字段编辑器对话框 (Field Editor Dialog)** 创建新字段或修改现有字段。

> **备注**
>
> 对于某些操作，必须将数据库手动模式 设置为 true。例如，更改字段类型。由于数据库处于" 手动模式"，更改类型不会反映在数据库结构中。请谨慎使用。

该对话框有以下选项卡：**字段 (Field)** 、**(查找 Lookup)** 、**(界面 Interface)** 和 **(计算 Calculation)** 。

“字段 (Field)” 选项卡

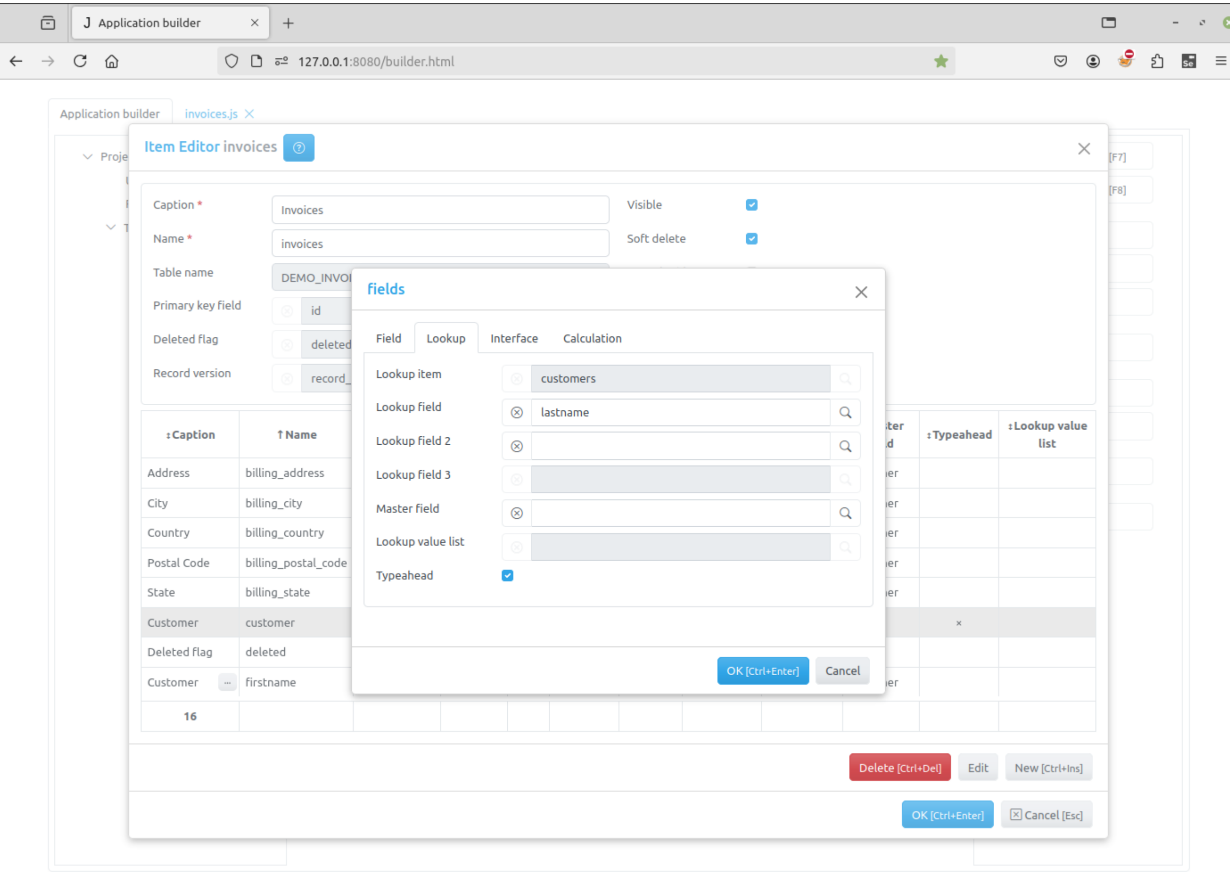


字段 (Field) 选项卡有以下字段：

- **标题 (Caption)** - 向用户显示的字段标题。
- **名称 (Name)** - 字段的名称，将在编程代码中用于访问字段对象。它应该是一个有效的 Python 标识符。
- **类型 (Type)** - 字段类型——以下值之一：
  - **TEXT** （文本）
  - **INTEGER** （整数）
  - **FLOAT** （浮点数）
  - **CURRENCY** （货币）
  - **DATE** （日期）
  - **DATETIME** （日期时间）
  - **BOOLEAN** （布尔值）
  - **LONGTEXT** （长文本）
  - **FILE** （文件）
  - **IMAGE** （图像）
- **大小 (Size)** - 文本字段的大小，即字符的个数
- **默认值 (Default value)** - 字段的默认值，对于布尔字段使用 0 或 1

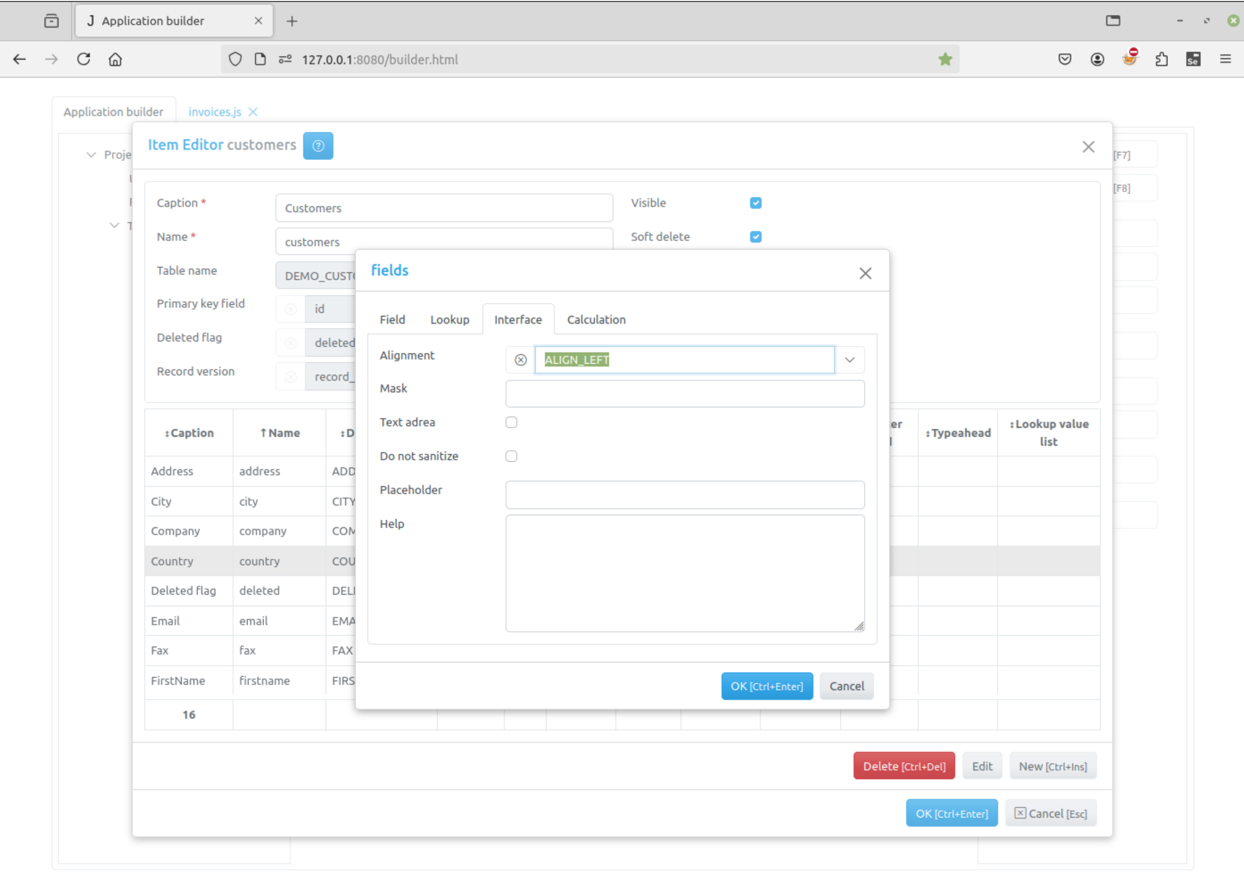
- **必填 (Required)** - 如果勾选此复选框，此字段为空时，post 方法将引发异常。请参阅修改数据集 。
- **只读 (Read only)** - 如果勾选此复选框，字段值无法在客户端通过*create_inputs* 方法创建的界面控件中更改。

"查找 (Look up)" 选项卡



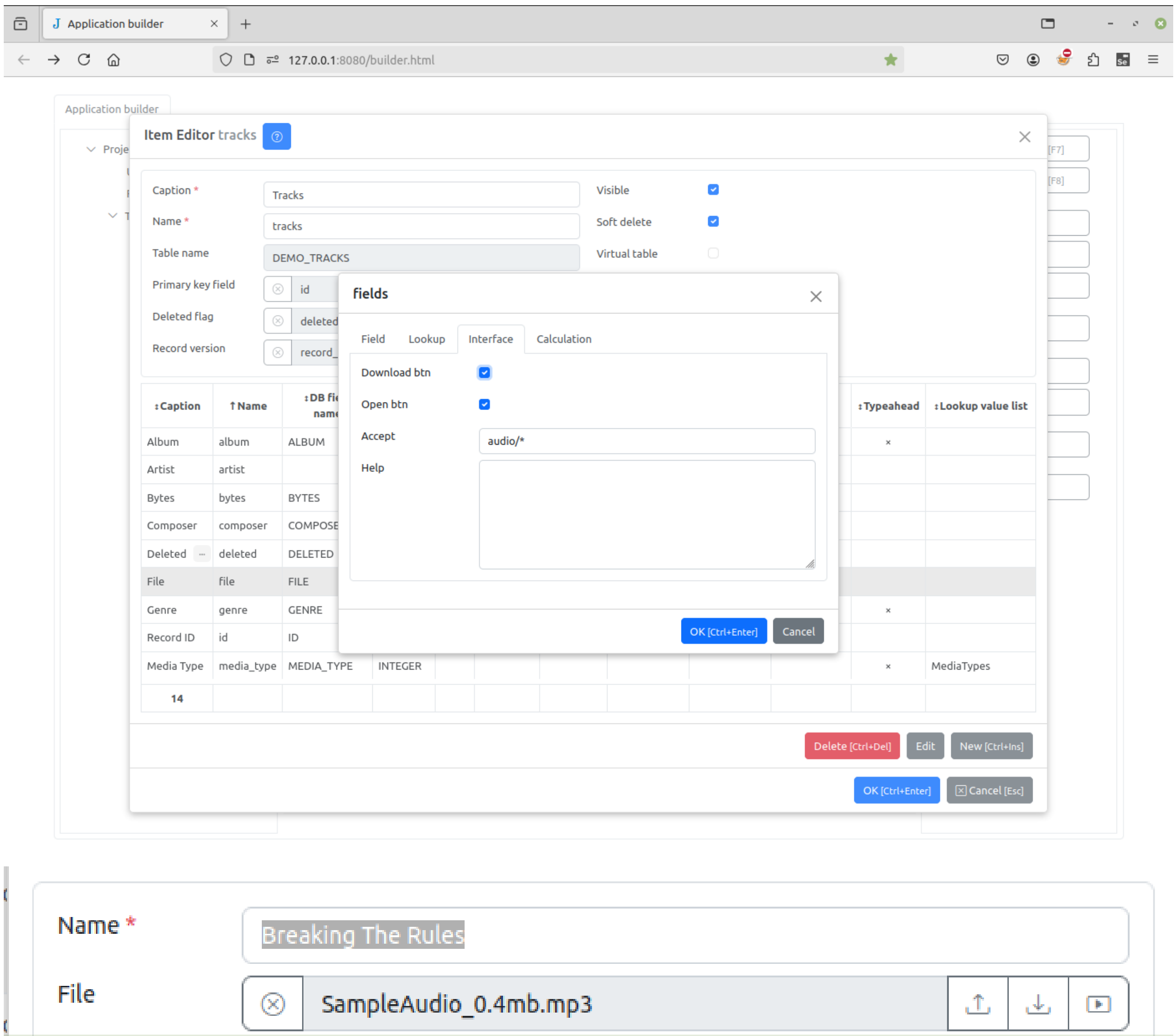
- **查找实体项 (Lookup item)** - 用于查找字段 引用的实体项
- **查找字段 (Lookup field)** - 用于查找字段 在用户界面显示内容的字段
- **查找字段 2(Lookup field 2)** - 用于查找字段 在用户界面显示内容的字段 2
- **查找字段 3(Lookup field 3)** - 用于查找字段 在用户界面显示内容的字段 3
- **主字段 (Master field)** - 用于查找字段 的主字段
- **预输入 (Typeahead)** - 如果勾选此复选框，则为查找字段启用预输入功能
- **(Lookup value list)** （查找值列表）- 用于为整数字段指定查找列表

“界面 (Interface)” 选项卡



- **掩码 (Mask)** - 使用此属性指定字段掩码
- **文本区域 (TextArea)** - 对于文本字段，如果设置了此属性，将在编辑表单对话框 中为这些字段创建 textarea 元素
- **不转义清理 (Do not sanitize)** - 设置此属性以防止默认的清理功能转义字段值，请参阅数据转义清理
- **对齐方式 (Alignment)** - 确定在显示此字段的控件中文本的对齐方式。
- **占位符 (Placeholder)** - 使用此属性指定字段输入框将显示的占位符。
- **帮助 (Help)** - 如果指定了任何“文本/HTML”消息内容，将在输入框右侧显示一个问号，当用户将鼠标指针移到该标记上时，将弹出一个窗口显示此消息。

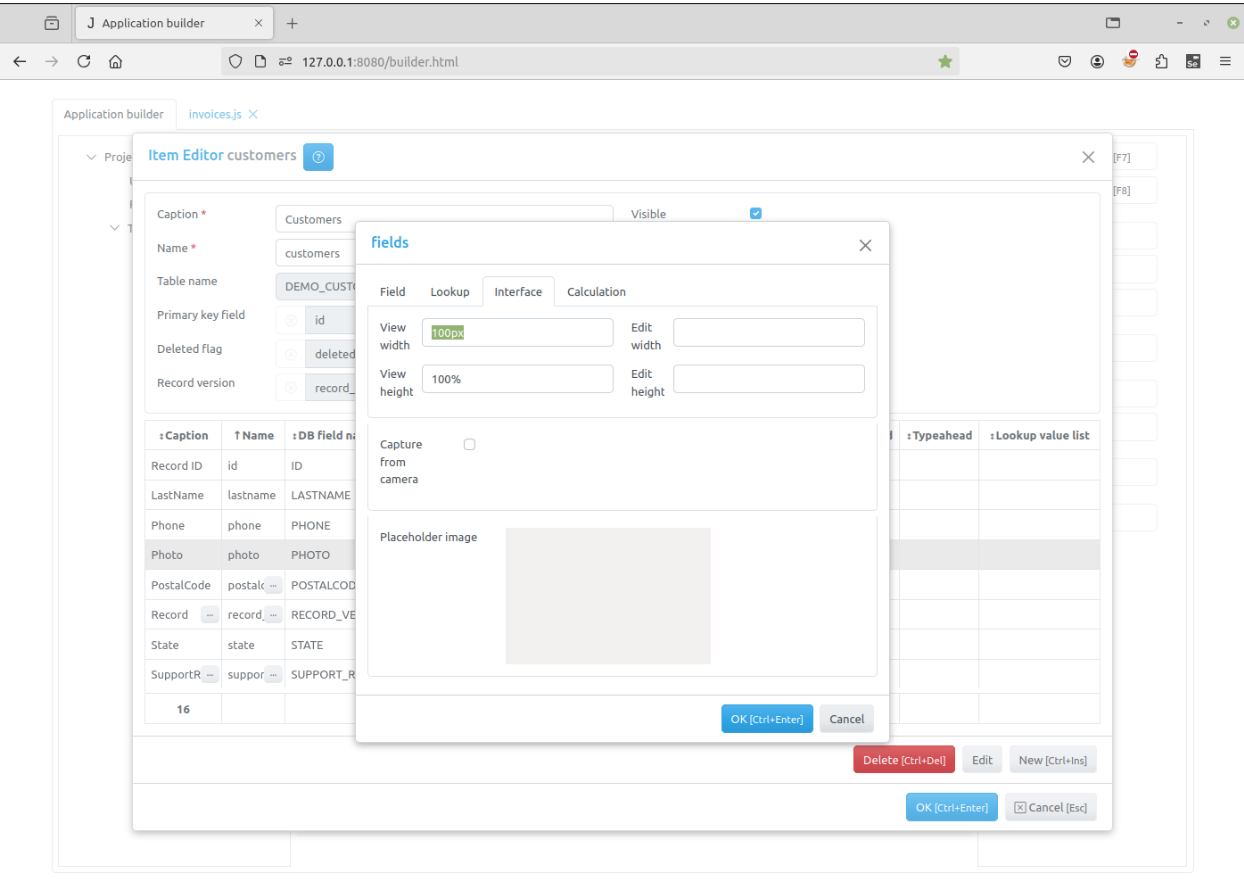
FILE 类型的字段的界面选项卡



- **下载按钮 (Download btn)** - 取消勾选以隐藏下载按钮（中间）
- **打开按钮 (Open btn)** - 取消勾选以隐藏打开按钮（右侧）
- **接受的文件类型 (Accept)** - 此属性指定可以加载的文件类型。这是一个可接受字符串。

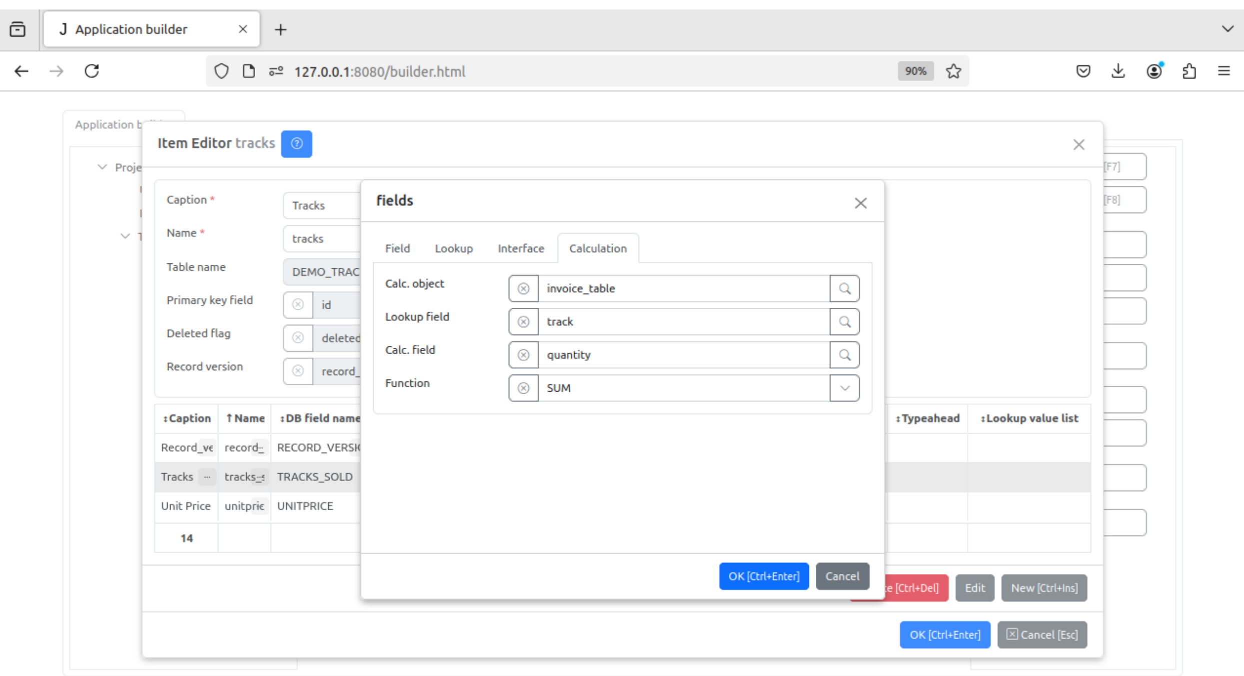
> **备注**
>
> 请注意，**接受的文件类型 (Accept)** 属性是必需填写的。在服务器端，会根据此属性检查上传的文件。

“图像 (IMAGE)”类型字段的界面选项卡



- **查看宽度 (View width)** - 指定图像在查看表单的表格中显示时的宽度（以像素为单位）。如果未指定，宽度为 auto（自动）
- **查看高度 (View height)** - 指定图像在查看表单的表格中显示时的高度（以像素为单位）。如果未指定，高度为 auto（自动）
- **编辑宽度 (Edit width)** - 指定图像在编辑表单中显示时的宽度（以像素为单位）。如果未指定，宽度为 auto（自动）
- **编辑高度 (Edit height)** - 指定图像在编辑表单中显示时的高度（以像素为单位）。如果未指定，高度为 auto（自动）
- **从摄像头捕获 (Capture from camera)** - 如果设置此复选框，用户将能够通过双击从摄像头捕获图像。图像会自动上传到服务器，前提是在参数接受字符串 中添加了”.png”。
- **占位符图像 (Placeholder image)** - 双击图像以设置占位符图像，当未设置字段图像时将显示该图像。按住 Ctrl 键并双击图像以清除占位符图像。

“计算 (Calculation)” 选项卡



- **计算对象 (Calc. object)** - 指定明细表 表。
- **查找字段 (Lookup field)** - 指定查找字段。例如，明细表 *invoice_table* 中的 *tracks* 字段，它是到 *Tracks* 表中 *Name* 字段的查找字段。
- **计算字段 (Calc. field)** - 指定在其上执行计算的字段。
- **函数 (Function)** - 指定计算使用的服务器端函数（SUM、COUNT、MIN、MAX、AVG 等）。

6.7.3 编辑表单对话框

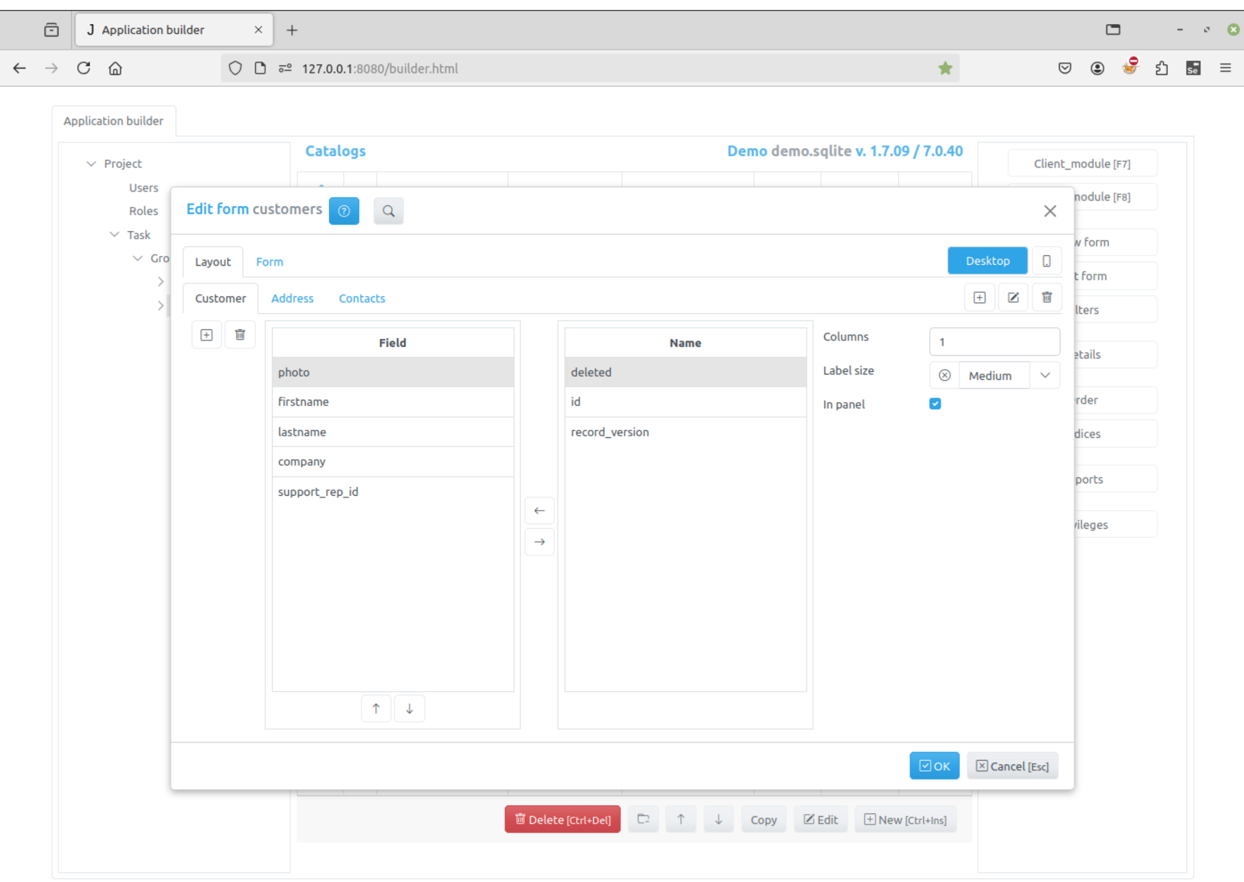
当开发人员在应用程序构建器中选择项目并点击 **编辑表单 (Edit Form)** 按钮时，会打开 **编辑字段对话框 (Edit Fields Dialog)** 。

它有两个选项卡：**布局 (Layout)** 和 **表单 (Form)** ，以及 `PC 桌面 (Desktop)` 和 `移动设备 (device)` 按钮。

`PC 桌面 (Desktop)` 按钮是默认选项，`移动设备 (device)` 可用于 **平板电脑 (tablet)** 、**手机 (mobile phone)** 。每个选项都是独立的。

布局 (Layout) 选项卡

在 **布局 (Layout)** 选项卡中，你可以指定用户可编辑的字段及其排列顺序，创建用于对字段输入内容进行分组的选项卡和区域。



布局 (Layout) 选项卡有两个字段列表。左侧列表包含已选择的用于编辑表单的字段。右侧列表包含未选择的可用字段。

要选择一个字段，请在右侧列表中选择它，并使用中间的 **左箭头 (←)** 按钮，或按键盘上的 **空格**键。

要取消选择一个字段，请在左侧列表中选择它，并使用中间的 **右箭头 (→)** 按钮，或按键盘上的 **空格**键。

要对选定的字段进行排序，请使用位于左侧列表下方的按钮。

在"布局 (Layout)"选项卡的右侧是各类控件，可用于为表单中选定的待编辑字段指定显示选项。

- **Columns** （列数）- 将为字段输入创建的列数 (1,2,3,4,6,12)。
- **列数 (Columns)** - 将为字段输入创建的列数
- **标签大小 (Label size)** - 选择一个值，该值确定显示在字段输入右侧的标签的字号大小：
  - xSmall （特小）
  - Small （小）
  - Medium （中）
  - Large （大）
  - xLarge （特大）
- **嵌入面板 (In panel)** - 如果设置，包含输入框的 div 将具有内嵌效果。

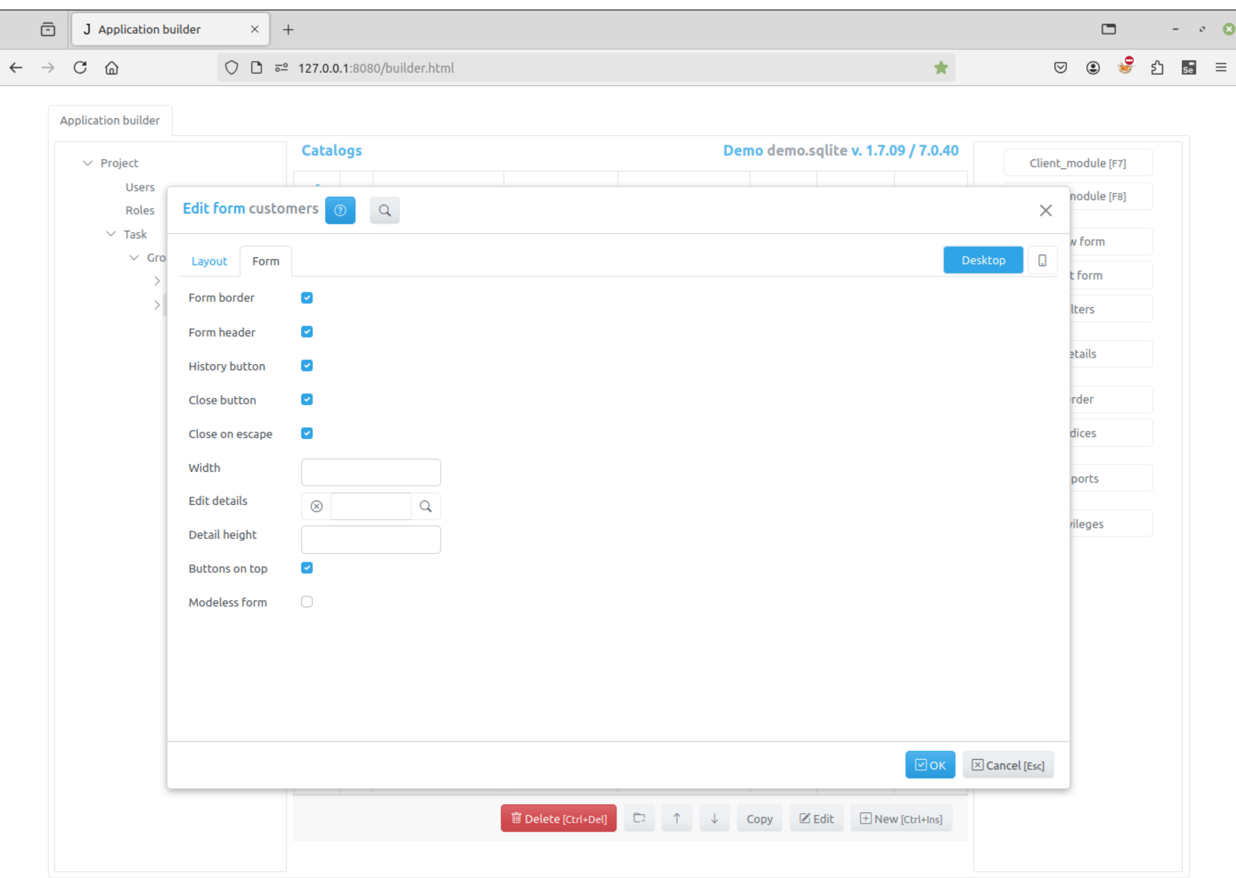
您可以为客户端编辑表创建 **选项卡 (tab)** 和 **区域 (band)** ，并自定义可以在每个选项卡或区域上编辑的字段。

布局选项卡的右侧有三个按钮，用于在客户端编辑表单的对话框中添加、编辑或删除 **选项卡**。

客户端编辑表单的每个选项卡都有一个默认区域，会占满整个选项卡。布局选项卡的左侧有两个按钮，可以用于在客户端编辑表单的选项卡内添加和删除 **区域**。

创建选项卡和区域后，您可以使用右侧的字段列表和控件来自定义将在每个选项卡和区域栏上编辑的字段。

表单 (Form) 选项卡



此选项卡上的控件可用于指定编辑表单的选项：

- **表单边框 (Form border)** - 如果设置，将在表单周围显示边框
- **表单页眉 (Form header)** - 如果设置，将创建并显示表单页眉，并包含表单的标题和各种按钮
- **记录历史 (History)** - 如果设置且启用了*保存更改历史*，记录操作的按钮将显示在表单标题中
- **关闭按钮 (Close button)** - 如果设置，将在表单的右上角创建关闭按钮
- **按 Escape 关闭 (Close on escape)** - 如果设置，按下 Escape 键将关闭表单
- **宽度 (Width)** - 一个整数，模态表单的宽度，如果未设置，值为 600 px
- **编辑明细表 (Edit details)** - 点击输入框右侧的按钮以选择将在编辑表单中可供编辑的明细表
- **明细表高度 (Detail height)** - 一个整数，编辑表单中显示的明细表的高度，如果未设置，明细表表格的高度为 262px
- **按钮置顶 (Buttons on top)** - 如果勾选此复选框，当表单具有默认表单模板时，按钮将显示在查看表单的顶部
- **非模态表单 (Modeless form)** - 如果勾选此复选框，表单将是非模态的，否则为模态。

点击 **确定 (OK)** 按钮保存结果，或点击 **取消 (Cancel)** 取消操作。

保存后，您可以通过刷新客户端项目页面查看更改。

6.7.4 查看表单对话框

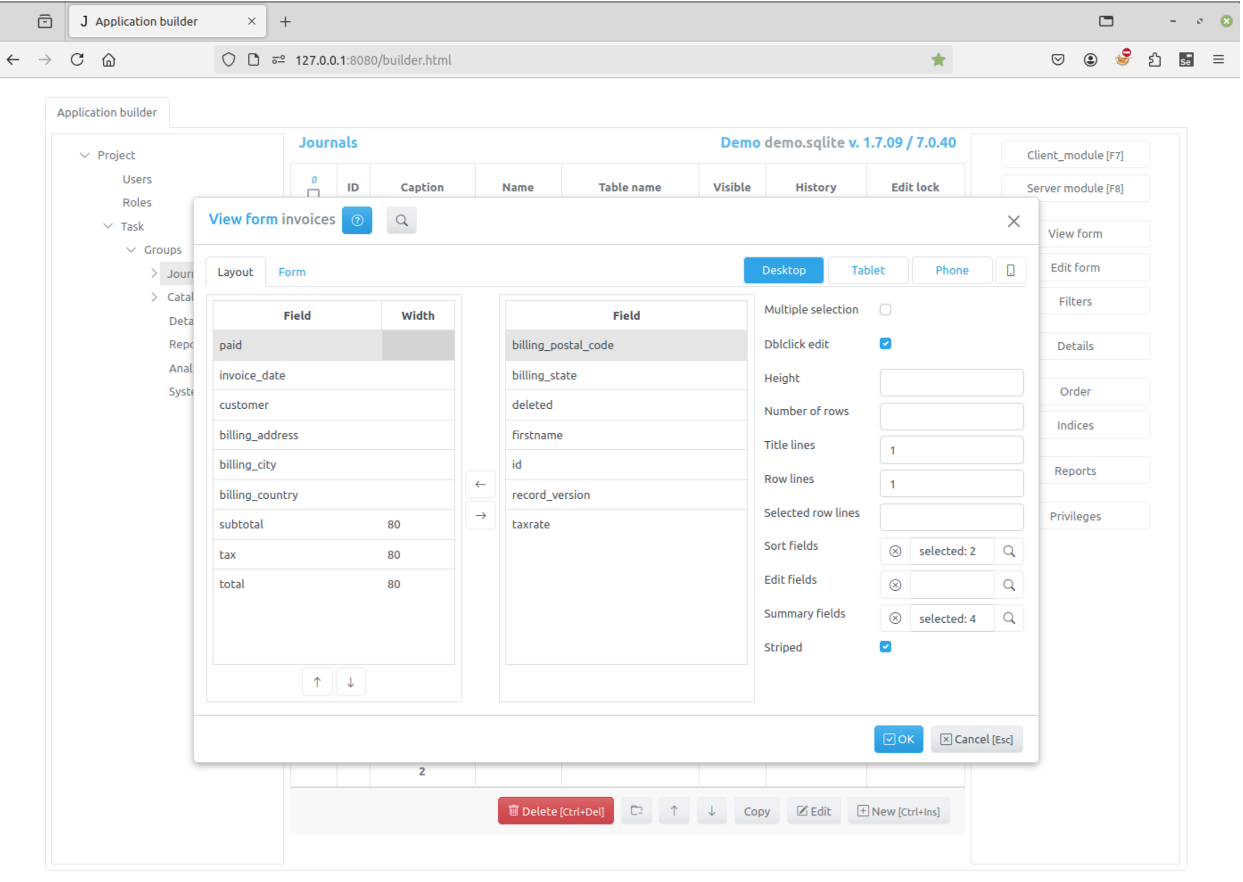
当开发人员在应用程序构建器中选择实体项并点击 **查看表单 (View Form)** 按钮时，会打开 **查看表单对话框 (View Form Dialog)** 。

它有两个选项卡：**布局 (Layout)** 和 **表单 (Form)** ，以及 `PC 桌面 (Desktop)` 和 `移动设备 (device)` 按钮。

`PC 桌面 (Desktop)` 按钮是默认选项，`移动设备 (device)` 可用于包含 **平板电脑 (tablet)** 和/或 **手机 (mobile phone)** 。每个选项都是独立的。

"布局 (Layout)" 选项卡

在布局选项卡上，您可以指定实体项的数据在客户端的查看表单中的显示方式。



设置表格显示的字段

(布局 Layout) 选项卡有两个字段列表。左侧列表包含已选择的在表格中显示的字段。右侧列表包含供您选择的可用字段。

要选择一个字段，请在右侧列表中选择它，并使用中间的 **左箭头 (←)** 按钮，或按键盘上的 **空格** 键。

要取消选择一个字段，请在左侧列表中选择它，并使用中间的 **右箭头 (→)** 按钮，或按键盘上的 **空格** 键。

要对选定的字段进行排序，请使用位于左侧列表下方的按钮。

您可以指定选定列的宽度。为此，请选择字段并在“宽度 (Width)”列中输入其宽度。该值可以用任何 CSS 支持的单位指定，例如，以像素为单位 - `px`，以相对于父元素的百分比为单位 - `%`。指定为整数值的宽度被解释为以像素为单位的宽度。

列宽度值示例：

- 100px
- 100
- 50%
- 2cm

设置表格选项

在“布局 (Layout)”选项卡的右侧是您可用于指定查看表单中显示的表格选项的控件：

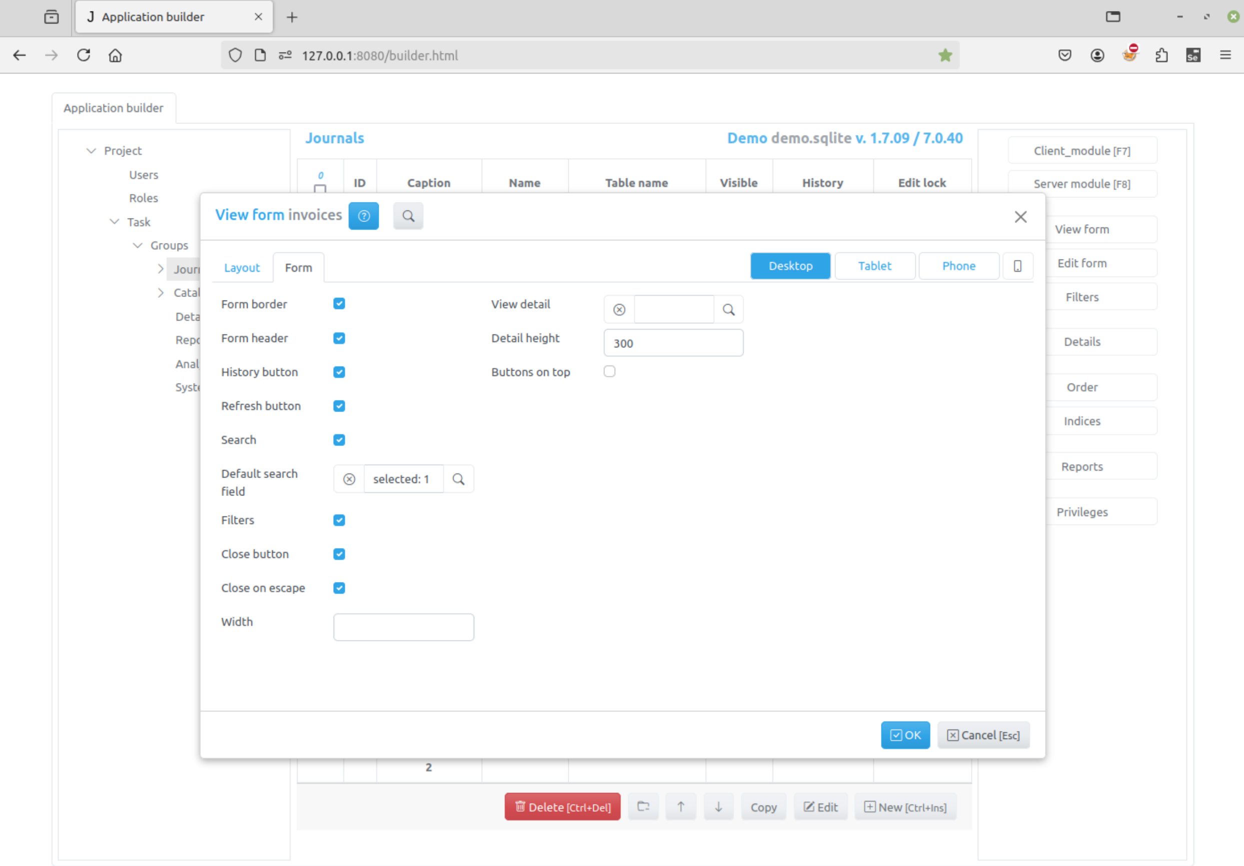
- **多选 (Multiple selection)** - 如果设置，将在最左侧创建一个复选框列以选择记录。

这样，当用户点击复选框时，记录的主键字段值将被添加到:doc:‘selections </refs/client/item/at_selections>‘（选择）属性中，或将其删除。

- **双击编辑 (Dblclick edit)** - 如果设置，当用户双击表格行时，将显示编辑表单。
- **记录的行数 (Number of rows)** - 一个整数，如果设置，指定表格中每页最多显示的行数；否则，如果未指定**(Height)**（高度），应用程序将根据页面高度计算表格的高度。
- **高度 (Height)** - 一个整数，如果设置，指定表格的高度（以像素为单位）；否则，如果未指定**行数 (Number of rows)** ()，应用程序将根据页面高度计算表格的高度。
- **文本行数 (Row lines)** - 一个整数，指定表格中每行显示的文本行数；如果为 0，则行的高度由行单元格的内容决定。
- **选定文本行数 (Selected row lines)** - 一个整数值，如果设置了**(Row lines)**（行数）且此值大于 0，它指定表格选定行中显示的最小文本行数。
- **冻结列 (Freeze columns)** - 一个整数，如果大于 0，它指定前几列将被冻结——当表格水平滚动时，这些列不会滚动。此选项在 V7 中是实体项的一个属性。
- **排序字段 (Sort fields)** - 点击输入框右侧的按钮以打开字段列表，并选择可以通过点击表格相应列标题来排序表格内容的字段。
- **汇总字段 (Summary fields)** - 点击输入框右侧的按钮以打开字段列表，并选择将为其计算汇总并在相应列脚注中显示的字段。对于数字字段，将计算总和；对于非数字字段，将计算记录数。

您可以通过使用实体项的:doc:‘table_options </refs/client/item/at_table_options>‘（表格选项）属性在客户端以编程方式获取或更改这些值。

“表单 (Form)”选项卡



此选项卡上的控件可用于指定查看表单的选项：

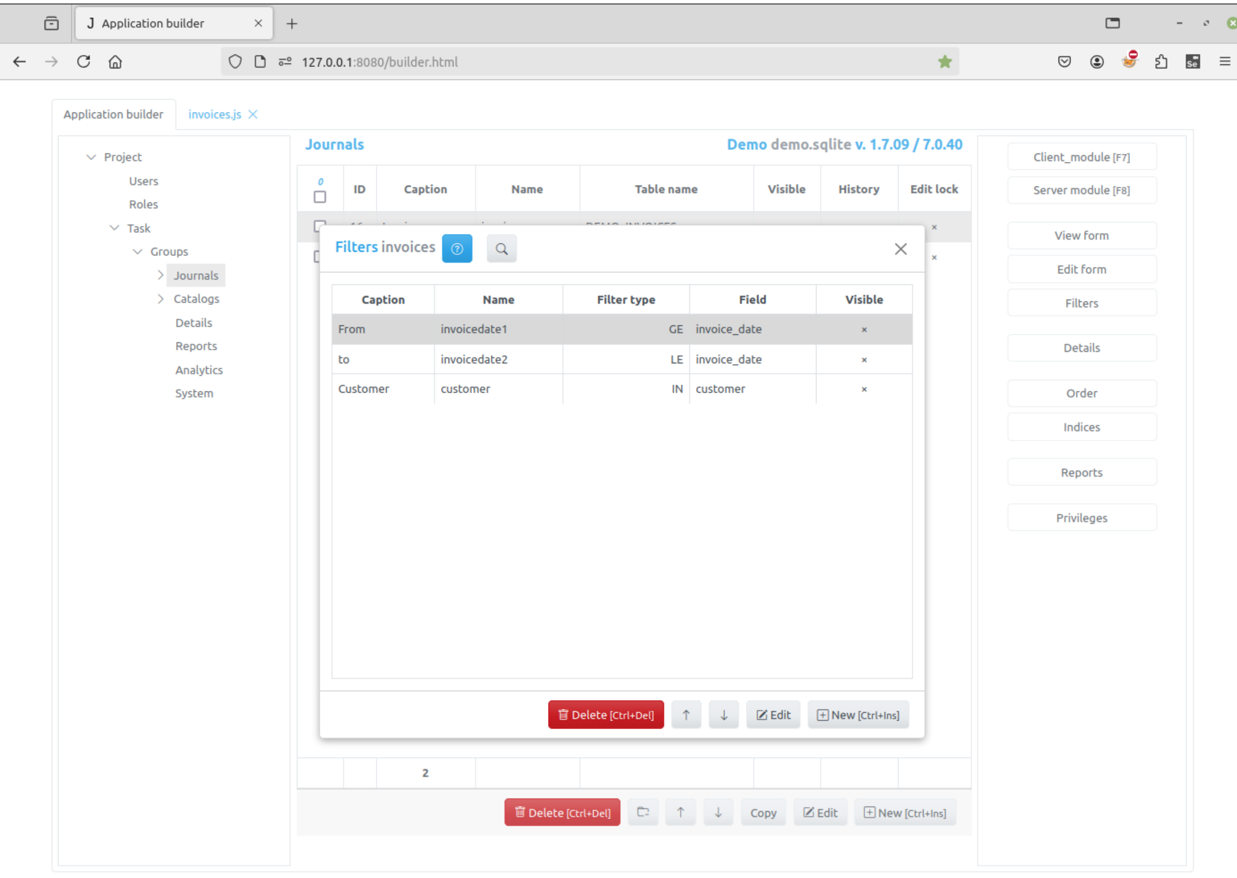
- **表单边框 (Form border)** - 如果设置，将在表单周围显示边框
- **表单表头 (Form header)** - 如果设置，将创建并显示表单标题，包含表单标题和各种按钮
- **历史 (History)** - 如果设置且启用了*保存更改历史*，历史按钮将显示在表单标题中
- **刷新按钮 (Refresh button)** - 如果设置，将在表单标题中创建刷新按钮，允许用户刷新页面
- **搜索 (Search)** - 如果设置，将在表单标题中创建搜索输入框
- **默认搜索字段 (Default search field)** - 点击输入框右侧的按钮以选择默认被搜索的字段
- **过滤器 (Filters)** - 如果设置且有可见的过滤器，将在表单标题中创建过滤器按钮
- **关闭按钮 (Close button)** - 如果设置，将在表单的右上角创建关闭按钮
- **按 Escape 关闭 (Close on escape)** - 如果设置，按下 Escape 键将关闭表单
- **宽度 (Width)** - 一个整数，模态表单的宽度，如果未设置，值为 600 px
- **查看明细表 (View details)** - 点击输入框右侧的按钮以选择将在查看表单中显示的明细表
- **明细表高度 (Detail height)** - 一个整数，查看表单中显示的明细表的高度，如果未设置，明细表表格的高度为 232px
- **按钮在顶部 (Buttons on top)** - 如果勾选此复选框，当表单具有默认表单模板时，按钮将显示在查看表单的顶部

您可以通过使用实体项的*view_options* 属性在客户端以编程方式获取或更改这些值。

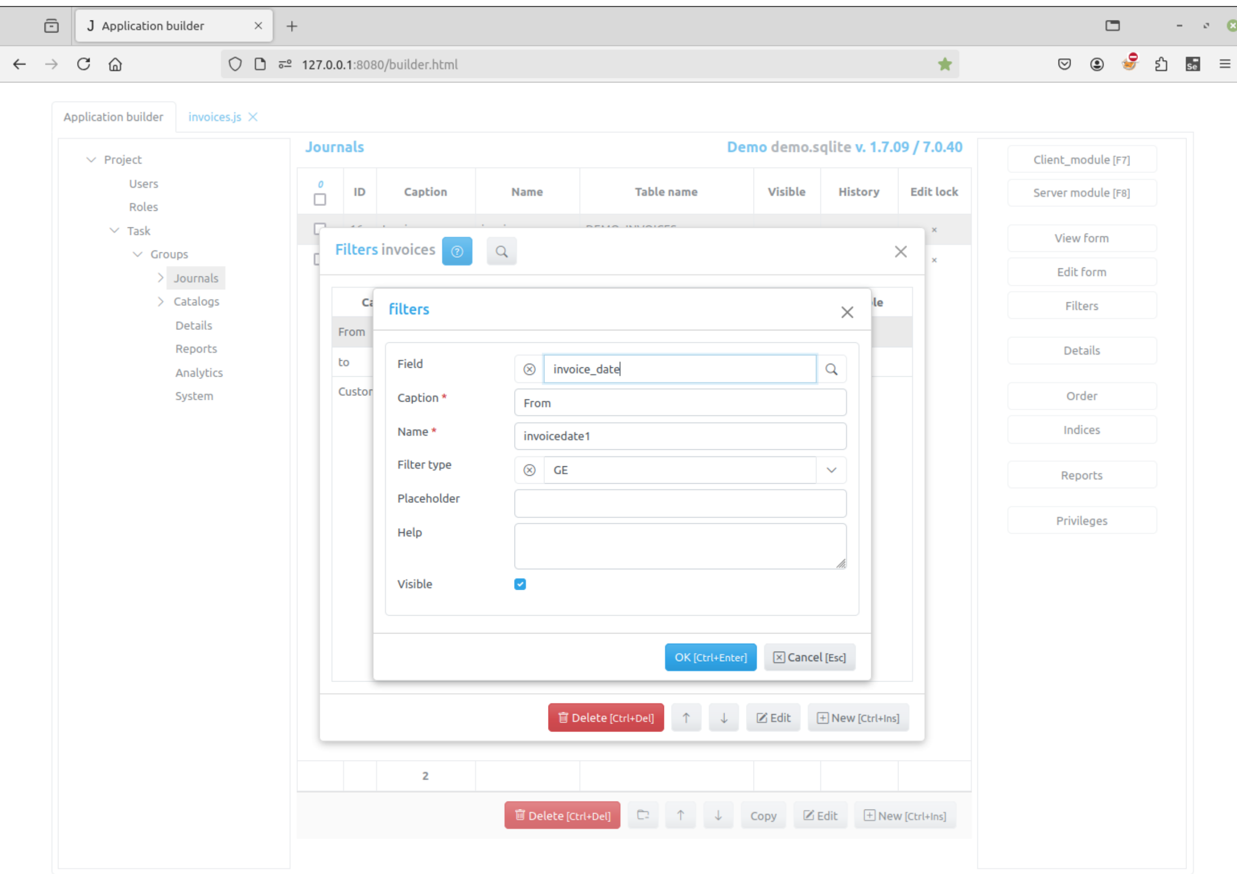
点击 **确定 (OK)** 按钮保存结果，或点击 **取消 (Cancel)** 取消操作。保存后，您可以通过刷新客户端项目页面查看更改。

6.7.5 过滤器对话框

使用 **过滤器对话框 (Filters Dialog)** 创建和修改实体项的过滤器。请参阅过滤器



要添加或编辑过滤器，请点击表单上的相应按钮。将出现以下表单：



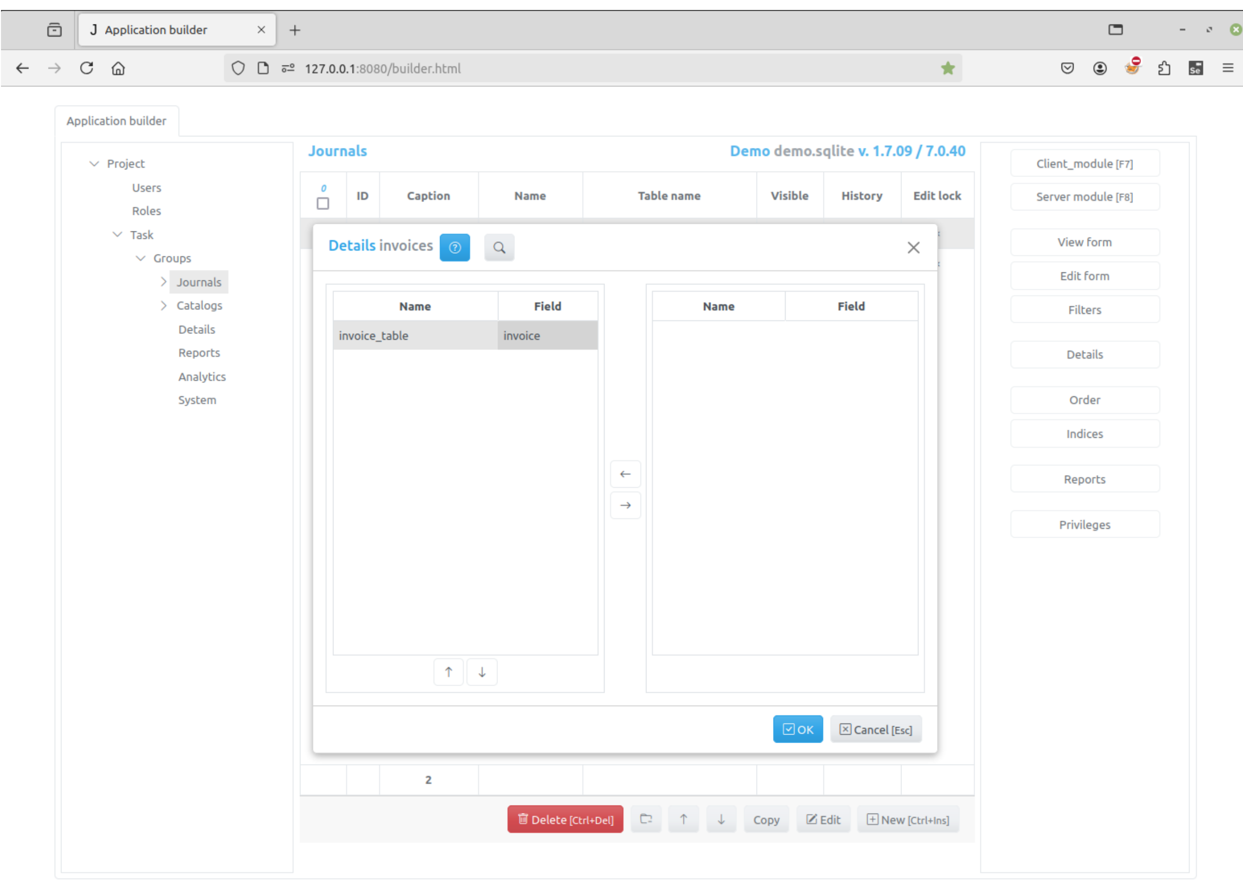
填写以下字段：

- **字段 (Field)** - 将用于筛选记录的字段。
- **标题 (Caption)** - 向用户显示的过滤器标题。
- **名称 (Name)** - 过滤器的名称，将在编程代码中用于访问过滤器对象。它应该是一个有效的 Python 标识符。
- **过滤器类型 (Filter type)** - 选择过滤器类型。
- **占位符 (Placeholder)** - 使用此属性指定字段输入框将显示的占位符。
- **帮助 (Help)** - 如果指定了任何文本/HTML 消息，将在输入框右侧显示一个问号，当用户将鼠标指针移到该标记上时，将弹出一个窗口显示此消息。
- **可见 (Visible)** - 如果未勾选此复选框，此过滤器将不会显示在实体项的过滤器对话框中。

使用上下箭头可以对过滤器按照显示的顺序进行排序。请参阅*create_filter_inputs*

6.7.6 明细表对话框

使用此对话框设置实体项的明细表。请参阅明细表 。



明细表对话框 (Details Dialog) 有两个面板。左侧面板列出了已添加到明细表的数据项。右侧面板列出了可供添加到明细表的数据项。

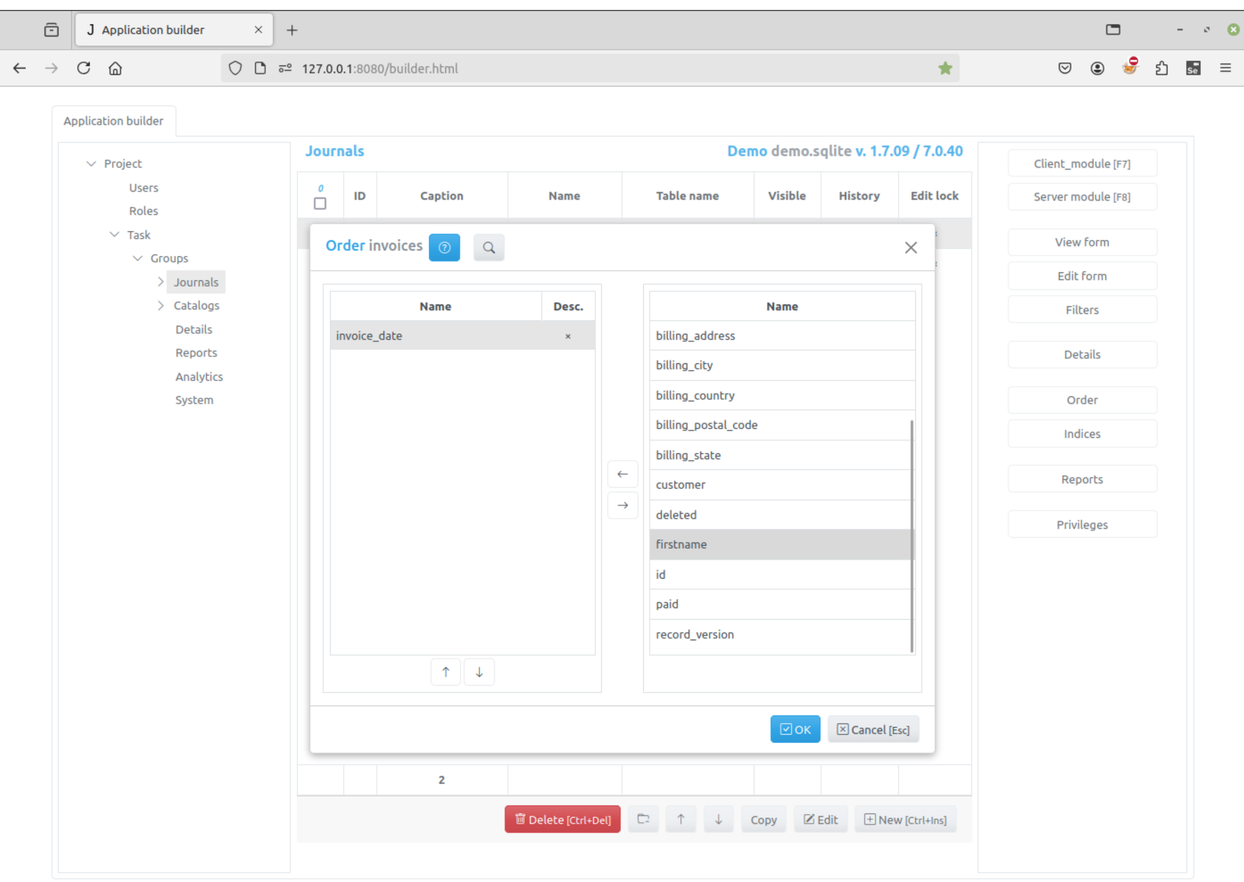
要将可用的数据项添加为明细表项，请在右侧面板中选择它，并使用中间的 **左箭头 (←)** 按钮，或按键盘上的 **空格**键。

要移除明细表中已有的项，请在左侧面板中选择它，并使用中间的 **右箭头 (→)** 按钮，或按键盘上的 **空格**键。

点击 **确定 (OK)** 按钮保存结果，或点击 **取消 (Cancel)** 取消操作。

6.7.7 排序对话框

当开发人员在应用程序构建器中选择实体项（请参阅实体项）并点击 **排序 (Order)** 按钮以指定记录的默认排序方式时，会打开 **排序对话框 (Order Dialog)** 。请参阅*open* 方法。



排序对话框 (Order Dialog) 有两个面板。左侧面板列出了已选择的字段。右侧面板列出了供选择的可用字段。

要选择一个字段，请在右侧面板中选择它，并使用中间的 **左箭头 (←)** 按钮，或按键盘上的 **空格**键。

要取消选择一个字段，请在左侧面板中选择它，并使用中间的 **右箭头 (→)** 按钮，或按键盘上的 **空格**键。

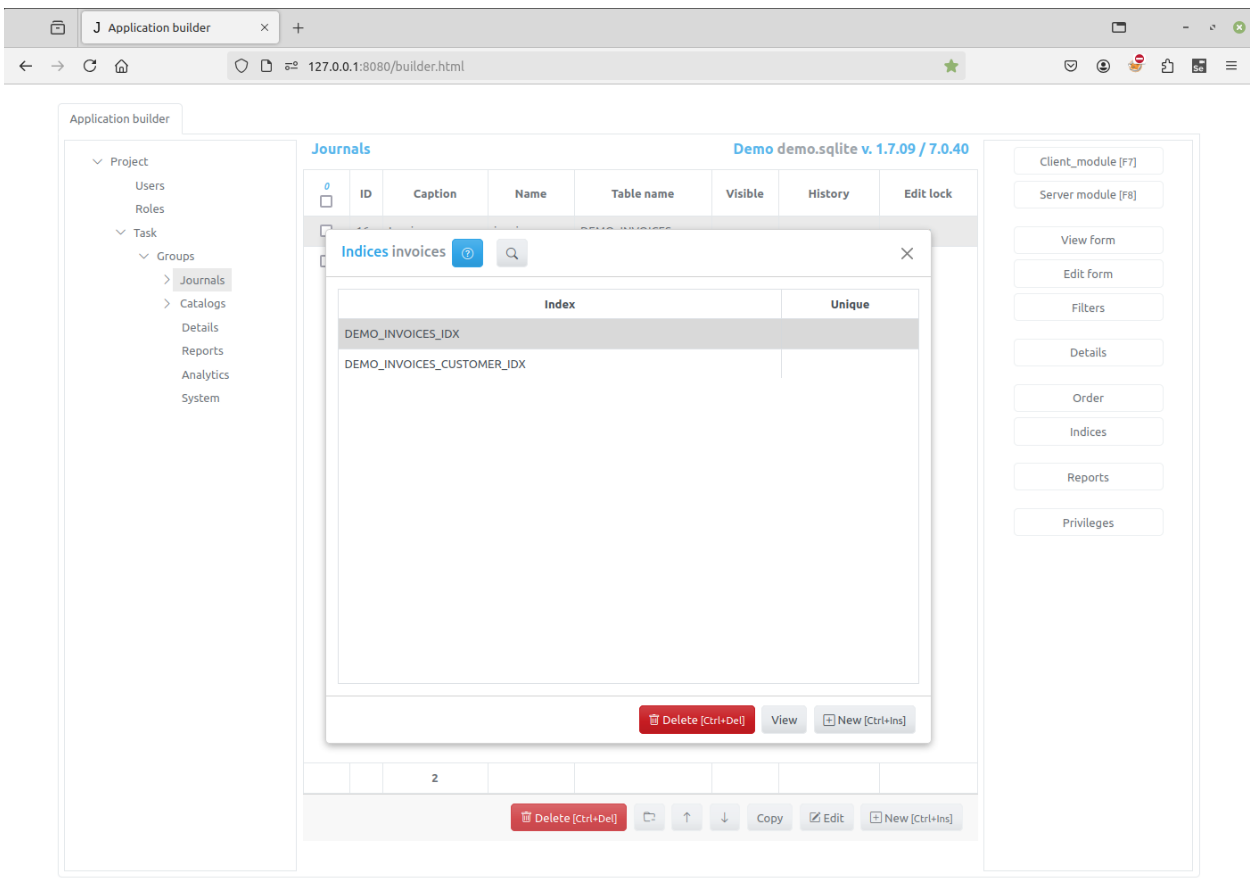
要对选定的字段进行排序，请使用位于左侧面板下方的按钮。

点击 **降序 (Desc)** 列，为字段设置降序/升序排序顺序。

点击 **确定 (OK)** 按钮保存结果，或点击 **取消 (Cancel)** 取消操作。

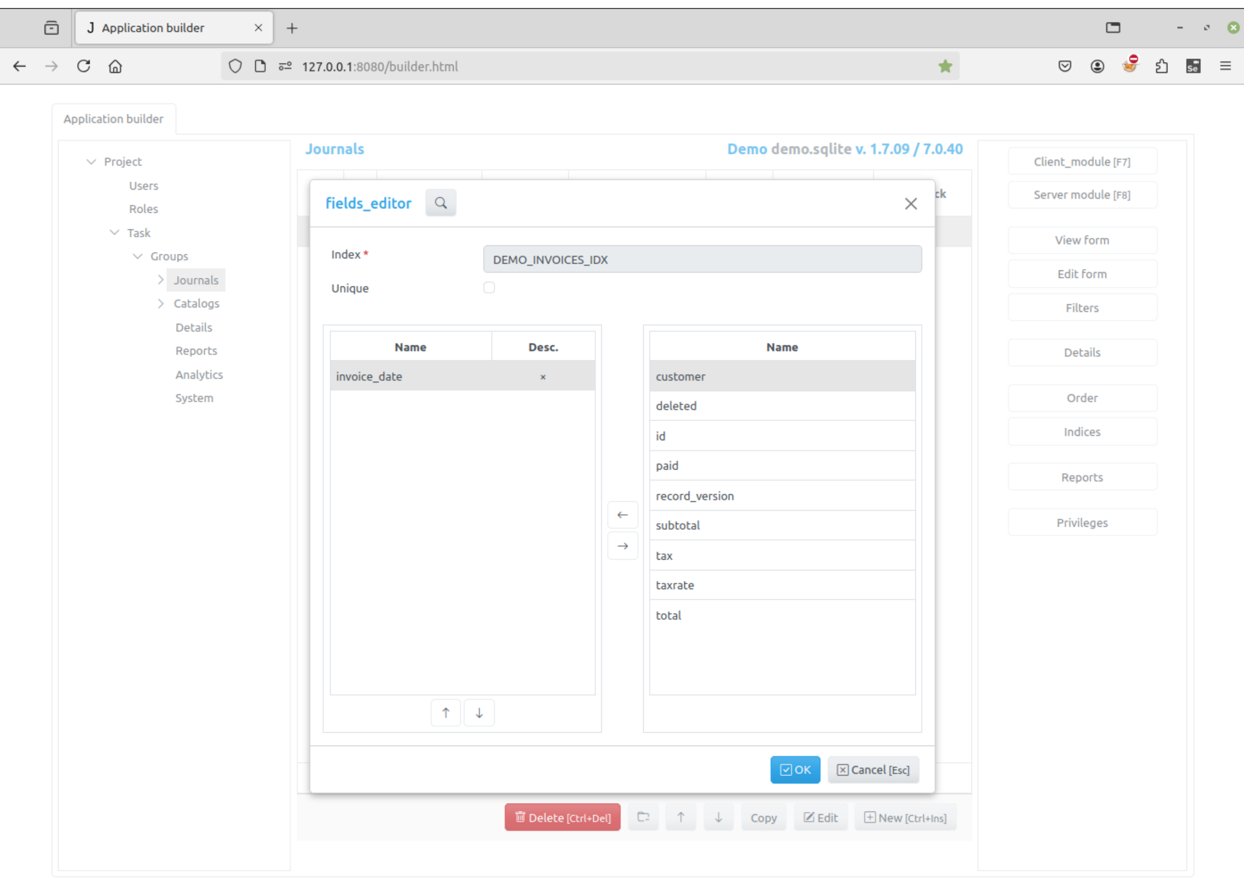
6.7.8 索引对话框

索引对话框 (Indices Dialog) 列出了为项目数据库中的数据表创建的索引。



要删除索引，请点击 **删除 (Delete)** 按钮。应用程序将生成删除索引的 SQL 查询并在服务器上执行。

要创建新索引，请点击 **新建 (New)** 按钮。将出现以下对话框：



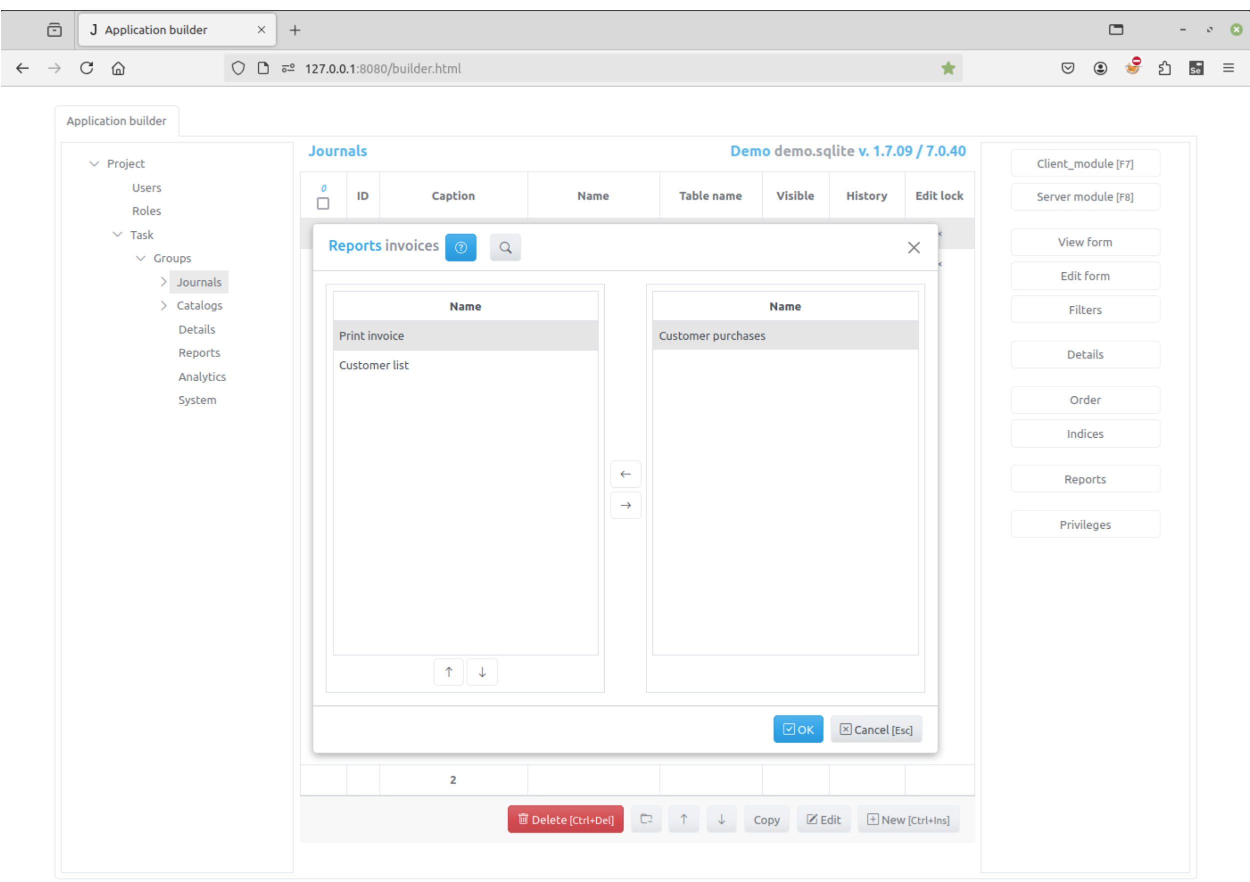
使用左右箭头按钮指定要创建索引的字段。如果要创建降序索引，请勾选 **降序 (Desc.)** 复选框。如有必要，请更改索引的名称。

点击 **确定 (OK)** 按钮创建索引。应用程序将生成创建索引的 SQL 查询并在服务器上执行。

点击 **取消 (Cancel)** 按钮取消操作。

6.7.9 报表对话框

当开发人员在应用程序构建器中选择项目（请参阅项目）并点击 **报表 (Reports)** 按钮以指定为实体项打印的报表时，会打开 **报表对话框 (Reports Dialog)** 。新项目的代码中有一个可用于打印报表的功能。



报表对话框 (Reports Dialog) 有两个面板。左侧面板列出了已选择的报表。右侧面板列出了供选择的可用报表。

要选择一个报表，请在右侧面板中选择它，并使用中间的 **左箭头 (←)** 按钮，或按键盘上的 **空格**键。

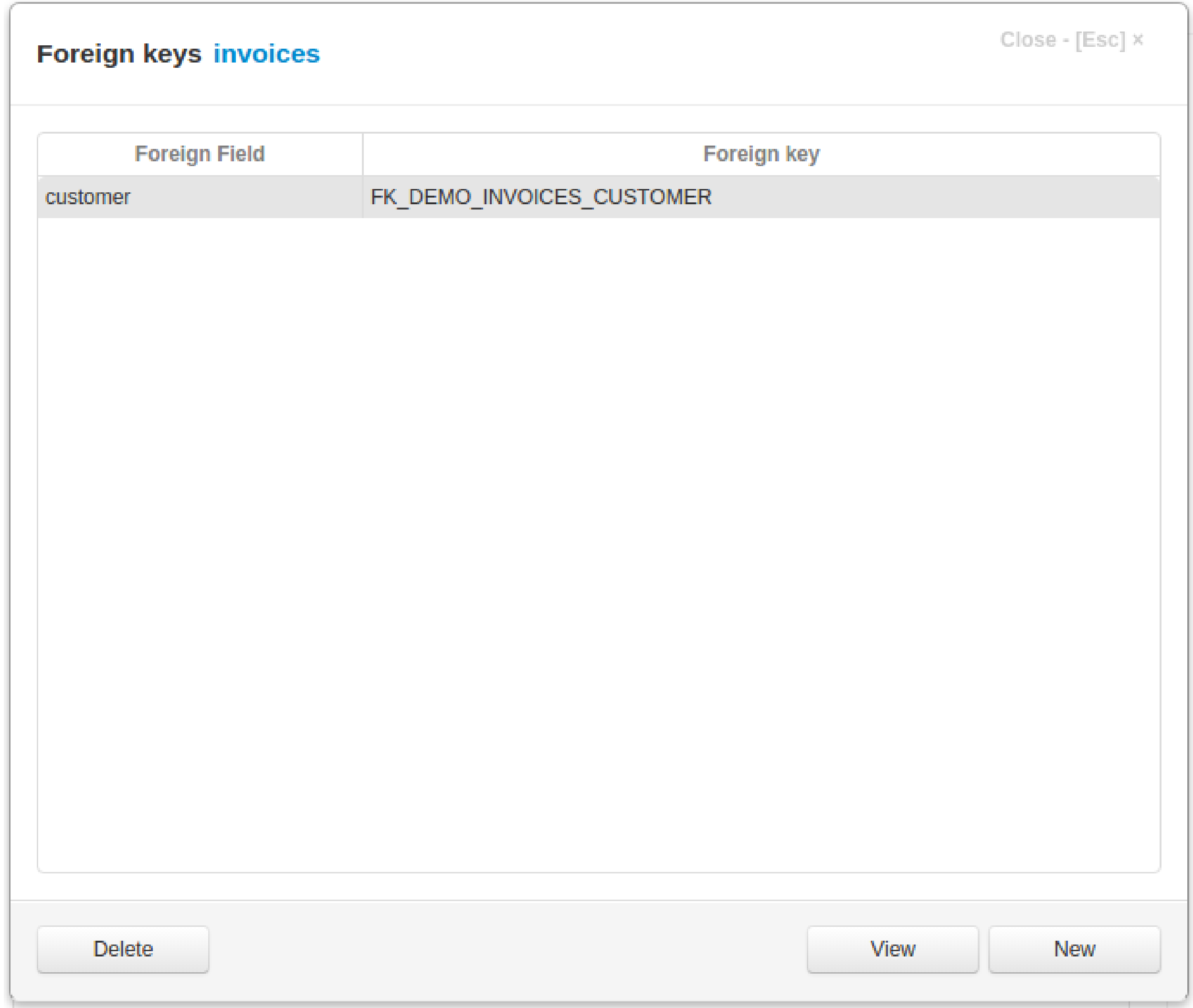
要取消选择一个报表，请在左侧面板中选择它，并使用中间的 **右箭头 (→)** 按钮，或按键盘上的 **空格**键。

要对选定的报表进行排序，请使用位于左侧面板下方的按钮。

点击 **确定 (OK)** 按钮保存结果，或点击 **取消 (Cancel)** 取消操作。

6.7.10 外键对话框

如果实体项有一个查找字段，并且在查找项的定义里没有设置 soft delete 属性，为了保证数据的完整性，我们可以创建一个外键。参考外键主题 in FAQ



为此，单击 **新建 (New)** 按钮，选择一个字段，并点击 **确定 (OK)** 。

6.8 明细表

Jam.py V7 中的明细表是指以主/明细关系链接的任何数据库表。为了逻辑上对数据库表进行分组，我们可以使用明细表组 来“存储”实体项的明细表，如下方截图所示。

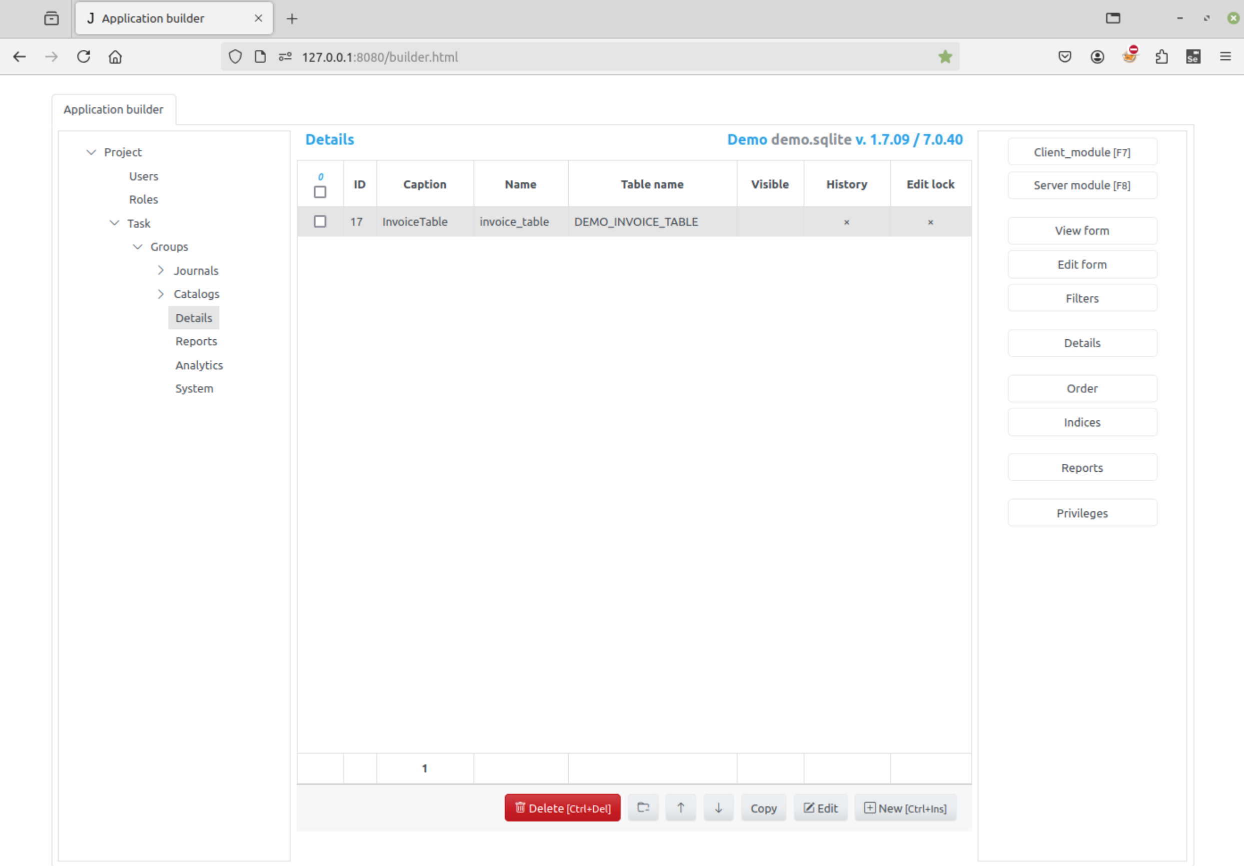
要处理实体项的明细表，请展开拥有该实体项的组节点，并在“任务”树中选择该实体项。在应用程序构建器的中心，将显示该实体项的所有明细表。

页面的右侧面板有以下按钮：

- **客户端模块 (Client module)** - 点击此按钮打开代码编辑器 以编辑明细表的客户端模块，请参阅使用模块 。
- **服务端模块 (Server module)** - 点击此按钮打开代码编辑器 以编辑明细表的服务端模块，请参阅使用模块 。
- **查看表单 (View Form)** - 使用此按钮调用查看表单对话框 以设置在客户端表格中默认显示的字段及其顺序。请参阅*create_table* 方法。

- **编辑表单 (Edit Form)** - 使用此按钮调用编辑表单对话框 以设置在客户端编辑表单中默认显示的字段及其顺序。请参阅*create_inputs* 方法。
- **过滤器 (Filters)** - 使用此按钮调用过滤器对话框 以创建、修改和删除实体项的过滤器。请参阅过滤器 。
- **明细表 (Details)** - 使用此按钮调用明细表对话框 以添加或删除链接到该实体项的明细表。注意，选择不是明细表的实体项，此此按钮打开的对话框才能正常使用。
- **排序 (Order)** - 使用此按钮调用排序对话框 以指定记录的默认排序方式。请参阅*open* 方法。
- **索引 (Indices)** - 点击此按钮打开索引对话框 为实体项的数据库表创建和删除索引。
- **外键 (Foreign keys)** - 点击此按钮打开外键对话框 为数据库表创建外键。
- **报表 (Reports)** - 点击此按钮打开报表对话框 以指定可为实体项打印的报表。新项目有一个可用于创建下拉按钮以打印报表的功能。
- **权限 (Privileges)** - 点击此按钮打开一个对话框，配置用户的角色对此实体项的权限。

使用页面底部的 **编辑 (Edit)** 按钮更改 明细表 *(detail)* 的 `item_name` 或 `caption` 。

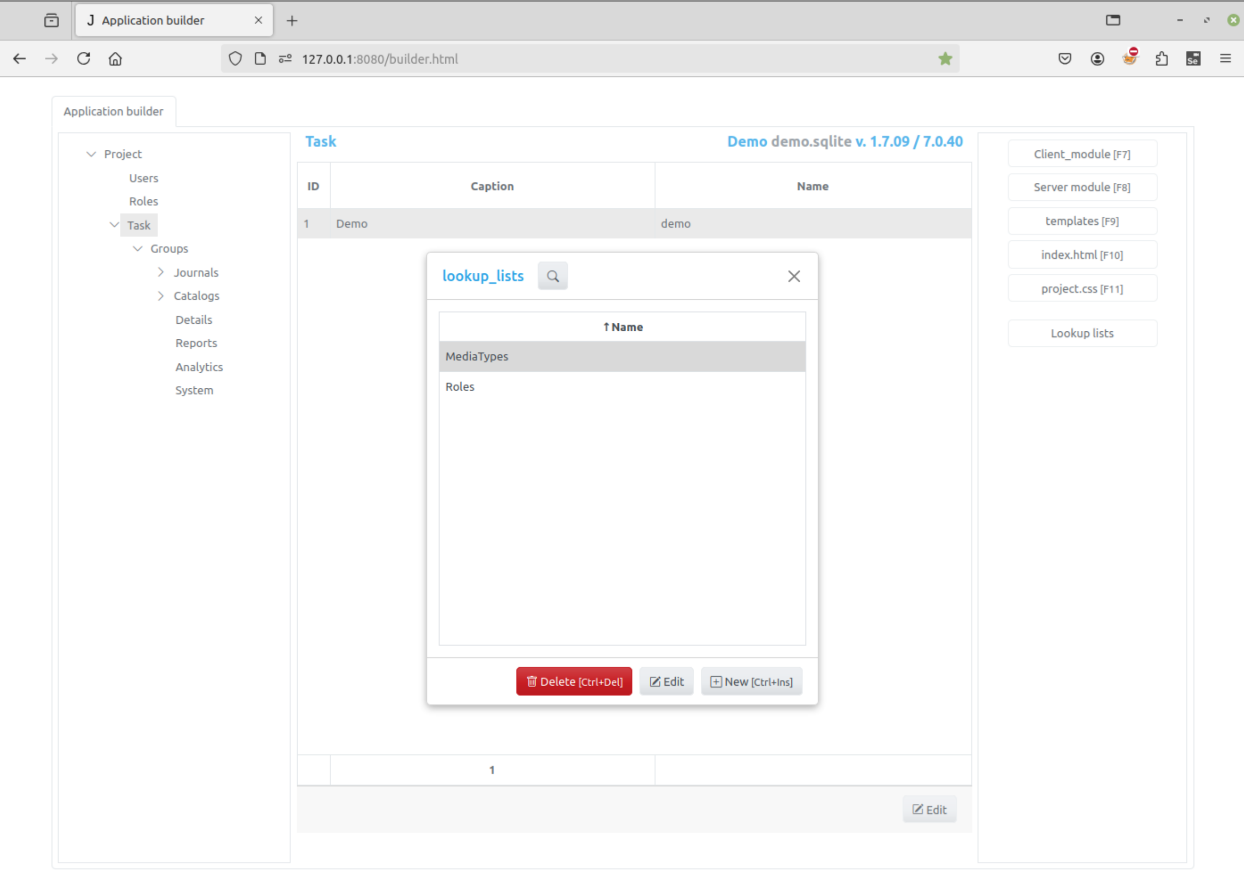


6.9 查找列表对话框

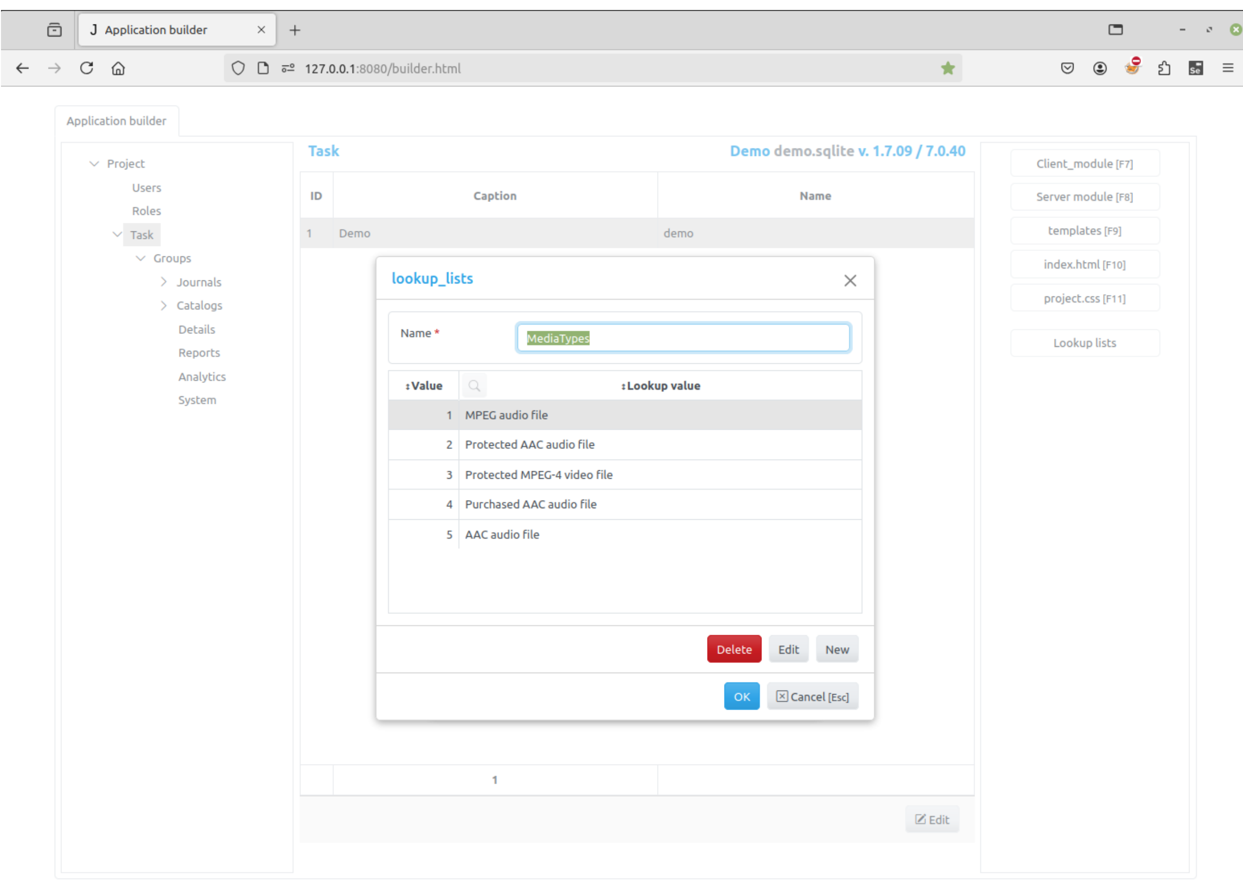
“查找列表 (Look list)”是一个“整数-文本”对的列表，可用作查找字段 的数据源。

> **备注**
>
> 查找列表的长度不应超过 10 条

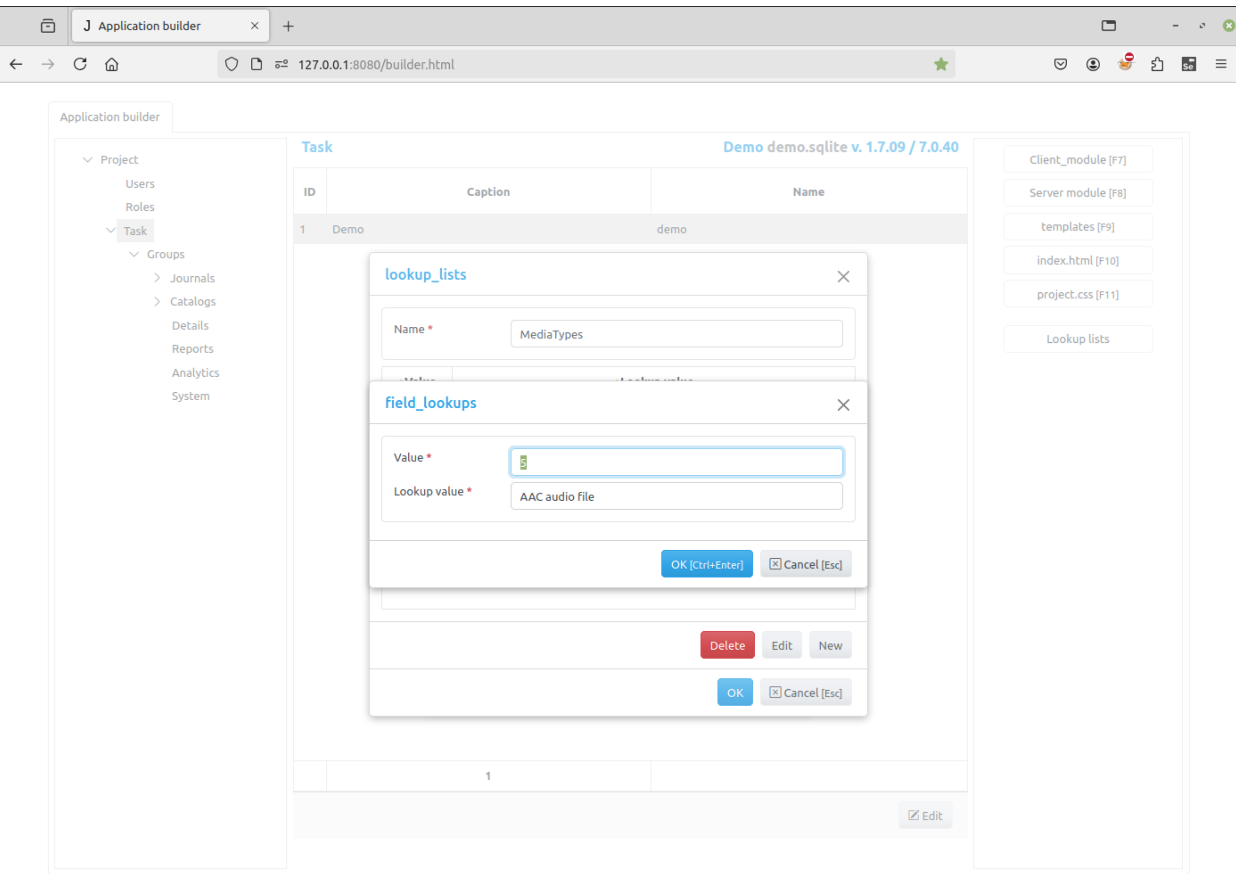
在左侧的项目树中，选择“任务 (Task)”节点，在页面右侧点击 **查找列表 (Look lists)** ，以打开 **查找列表对话框 (Look lists dialog)** 。



点击 **编辑 (Edit)** / **新建 (New)** 按钮以编辑/创建查找列表。



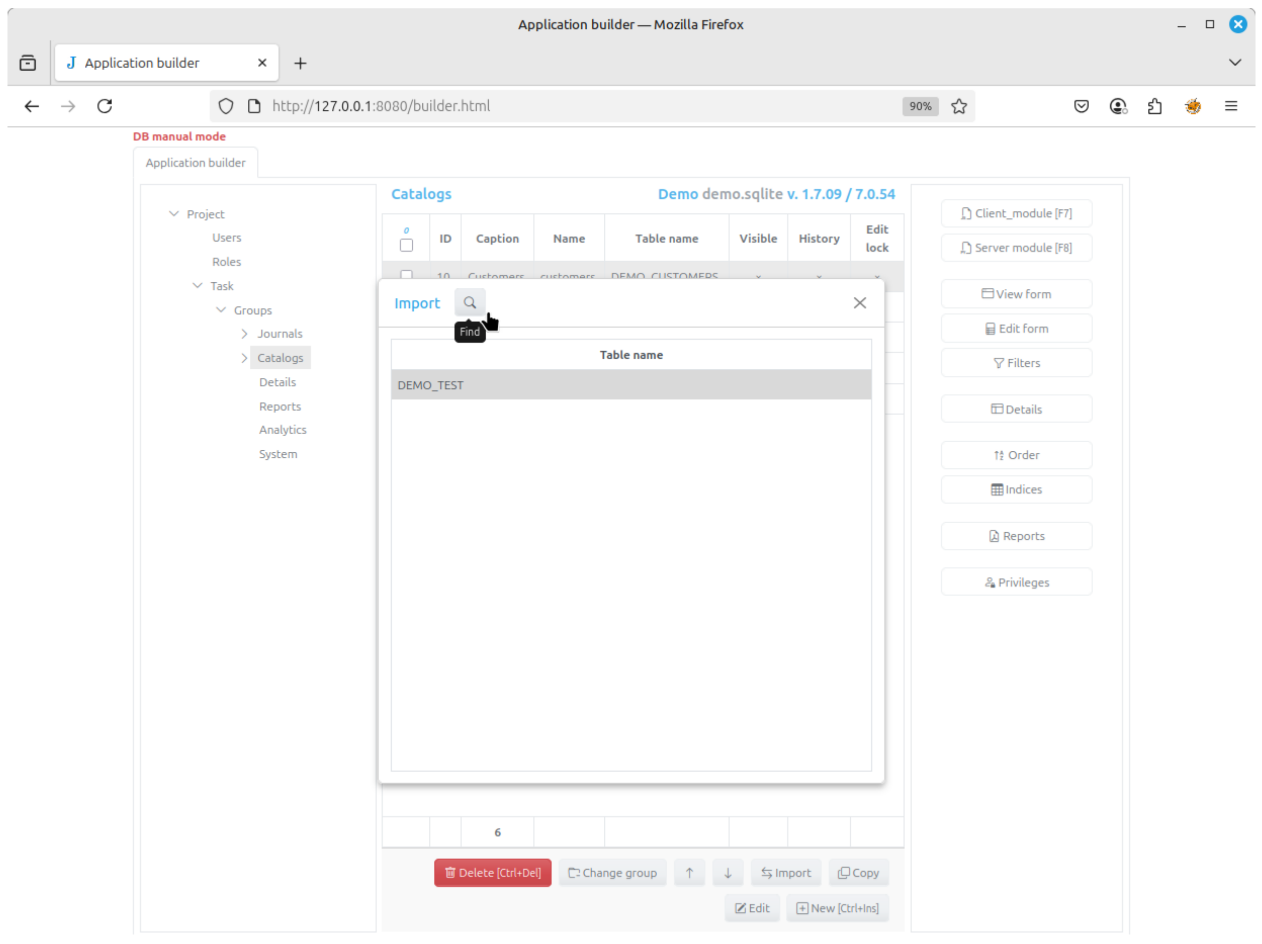
然后使用 **编辑 (Edit) / 新建 (New)** 按钮来编辑已有的查找对/向列表中添加新的查找对。



6.10 导入现有数据库表

要导入现有数据库表:

- 创建一个连接到现有数据库的新项目。
- 选择“项目 (Project)”节点并点击“数据库 (Database)”按钮。将数据库手动模式 设置为 true。
- 选择您要导入表到的组，然后点击“导入 (Import)”按钮。
- 在出现的表单中，双击要导入的表。



- 在项目编辑器对话框中，检查所有字段是否具有有效的类型。如果字段类型显示为红色，请尝试选择适当的类型。

  您可以导入表中的字段子集。

  在保存之前，如有必要，请指定实体项的主键字段和生成器名称。

- 保存导入的项目后，转到项目客户端页面，检查其显示情况。
- 导入多个表后，您可以指定查找字段（在数据库手动模式下）。

备注

请注意，执行此操作时请务必小心。

当移除数据库手动模式后，对实体项所做的任何更改都将反映在相应的数据库表中。如果您删除实体项，该表将从数据库中删除。

备注

要导入的数据库表必须有具有一个字段的主键。

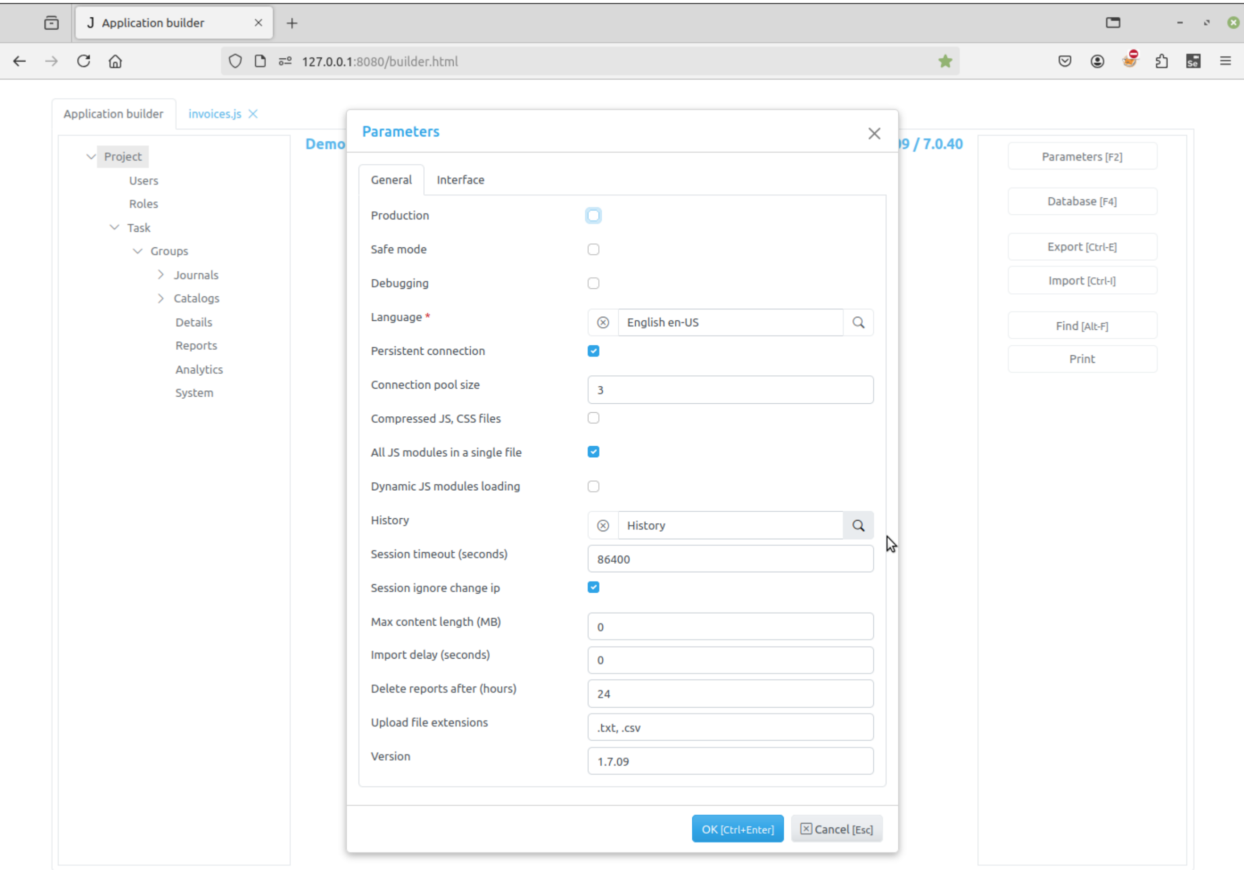
> **备注**
>
> 不得导入二进制字段。

> **备注**
>
> 不会导入索引。

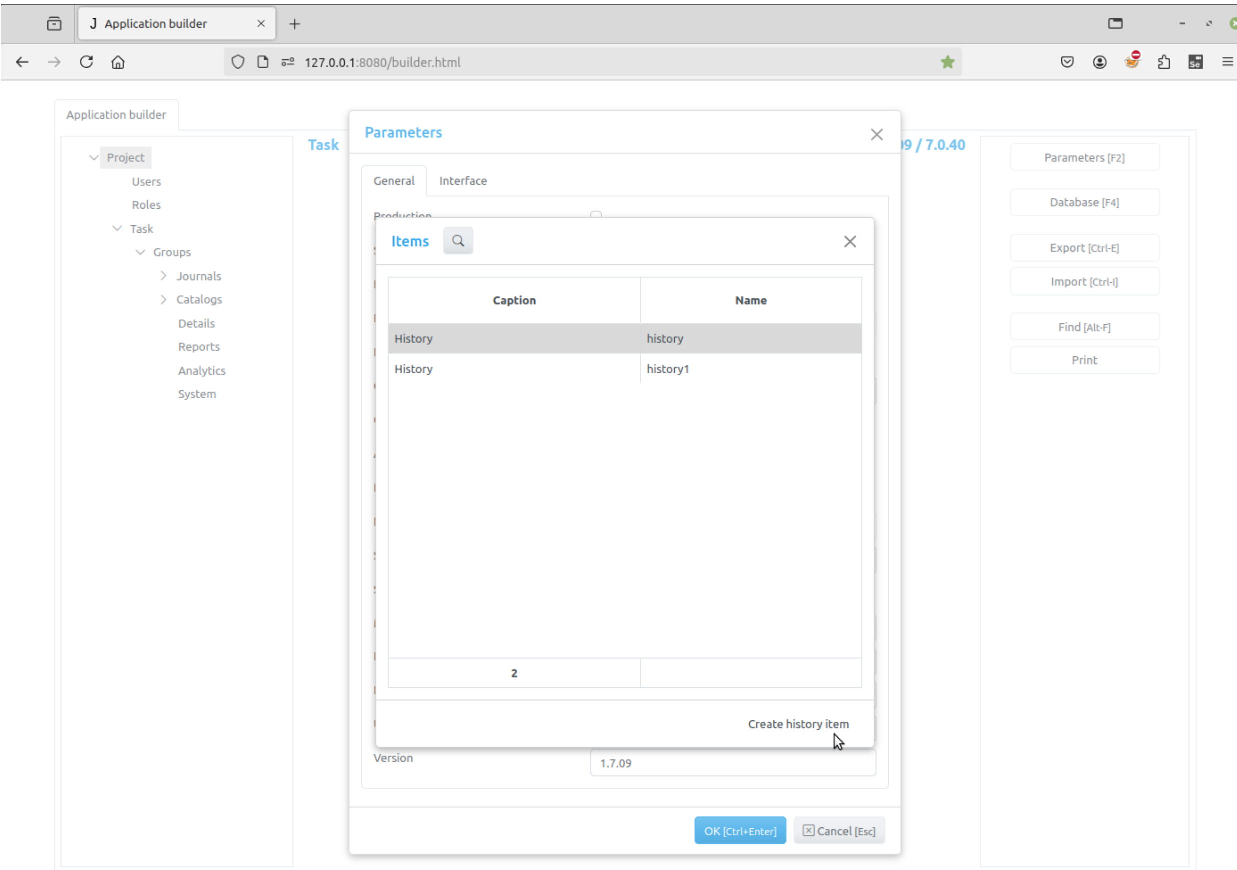
6.11 保存用户所做的审计追踪/变更历史记录

要保存用户所做的变更历史记录，您必须指定要存储这些记录的实体项。

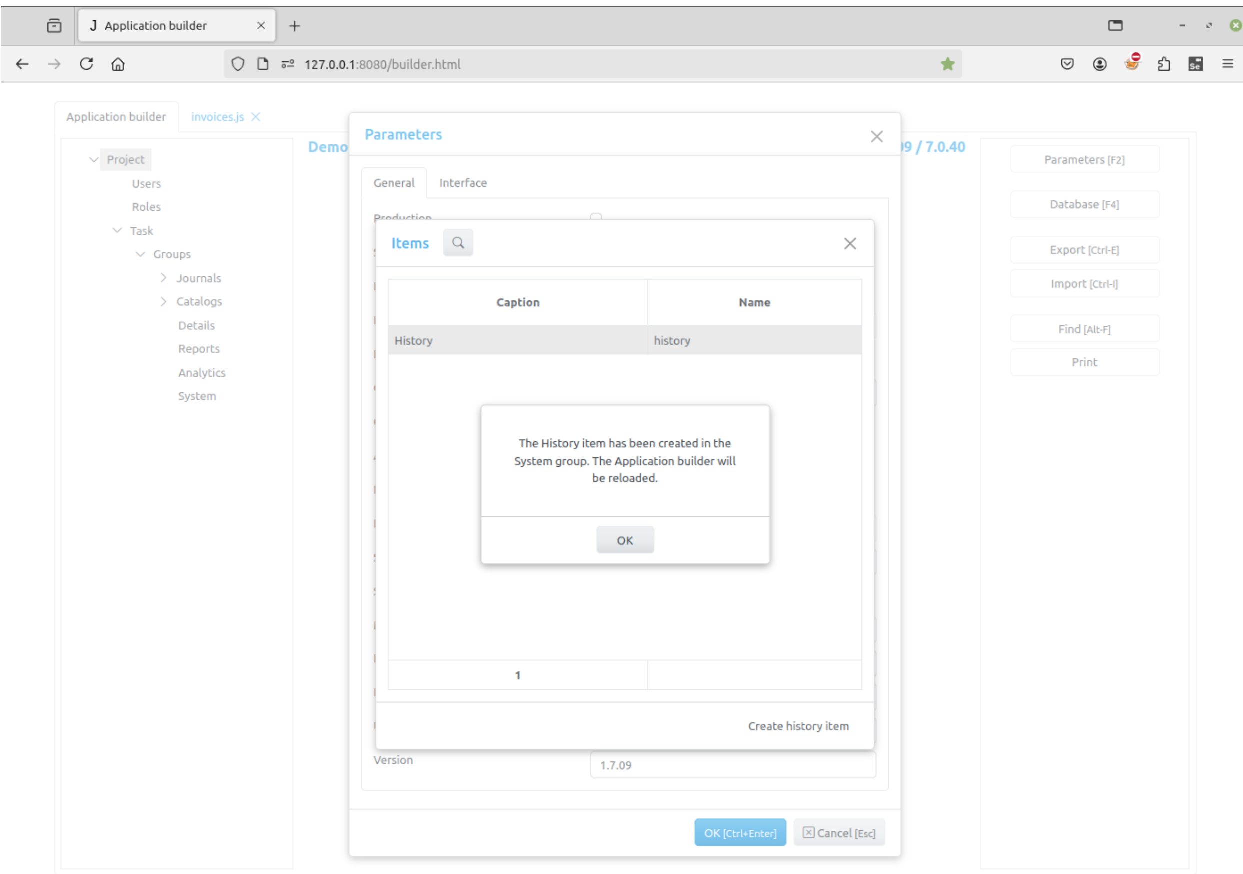
为此，请打开项目参数，并点击“历史 (History)”输入框右侧的按钮：



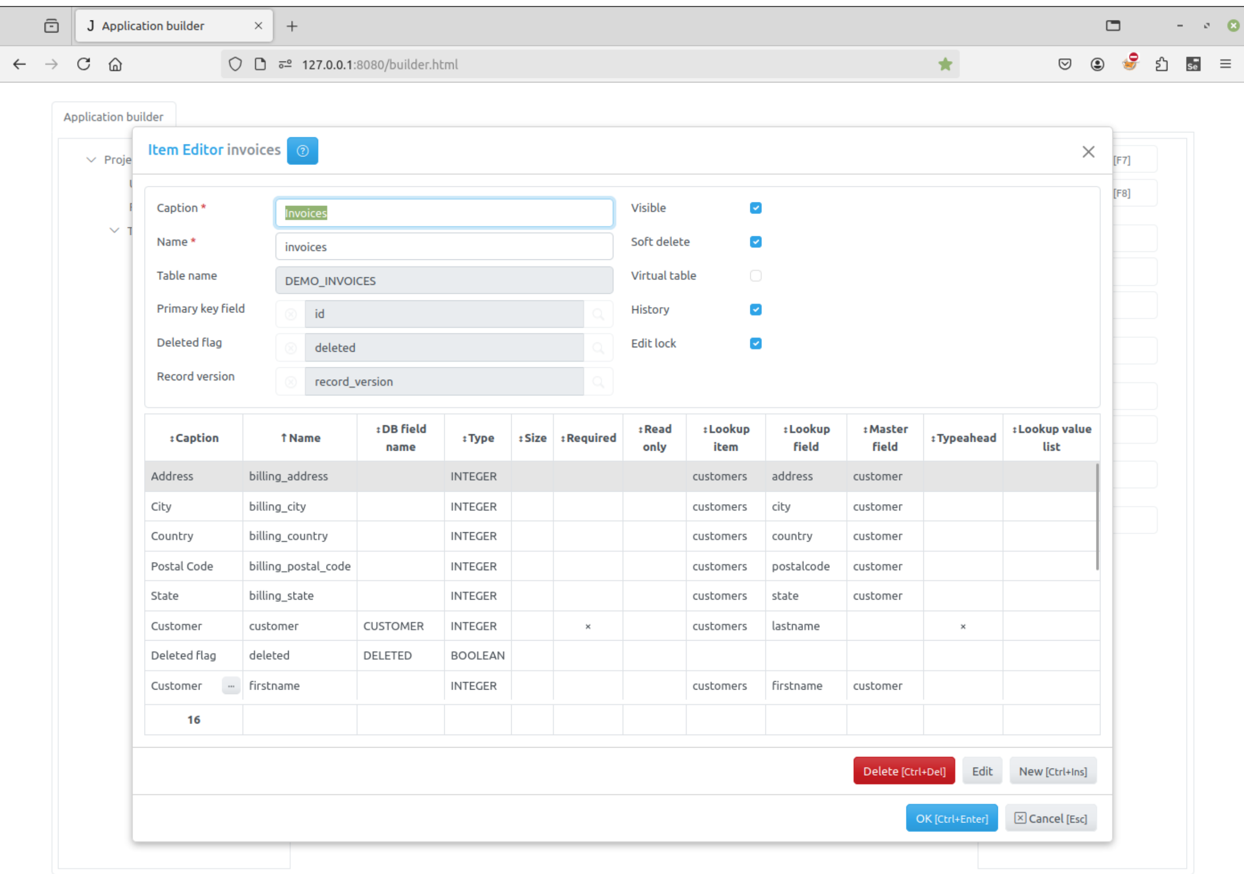
在出现的对话框中，点击“创建历史项 (Create history item)”按钮



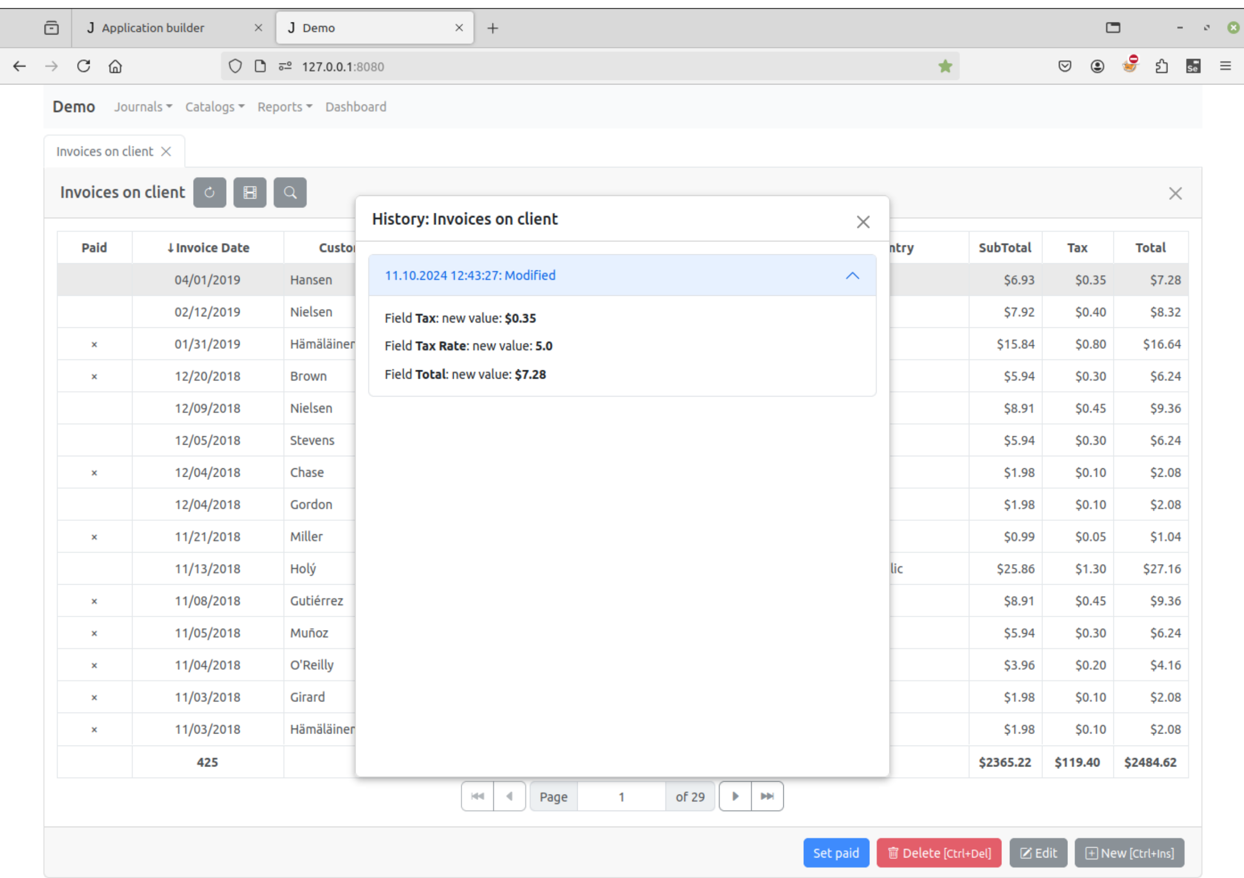
创建历史项时将显示以下消息:



之后，您必须设置要更改的实体项的 **保留历史 (Keep history)** 属性以保存其更改历史：



要查看记录的变更历史，请点击编辑表单标题栏上相应的的图标。



或者，您可以使用*show_history* 方法来实现。

> **备注**
>
> 当使用客户端 / 服务端 的 apply 方法将对数据集的更改应用到数据库时，会保存更改历史。
>
> 通过自定义 SQL 请求对数据库所做的更改不会保存在历史记录中。

> **备注**
>
> 这些更改可能会显著增加数据库的大小。请务必小心。

6.12 记录锁定

在 Jam.py 应用程序中，您可以在用户编辑记录时实现记录锁定，以防止多个用户同时编辑同一条数据项。

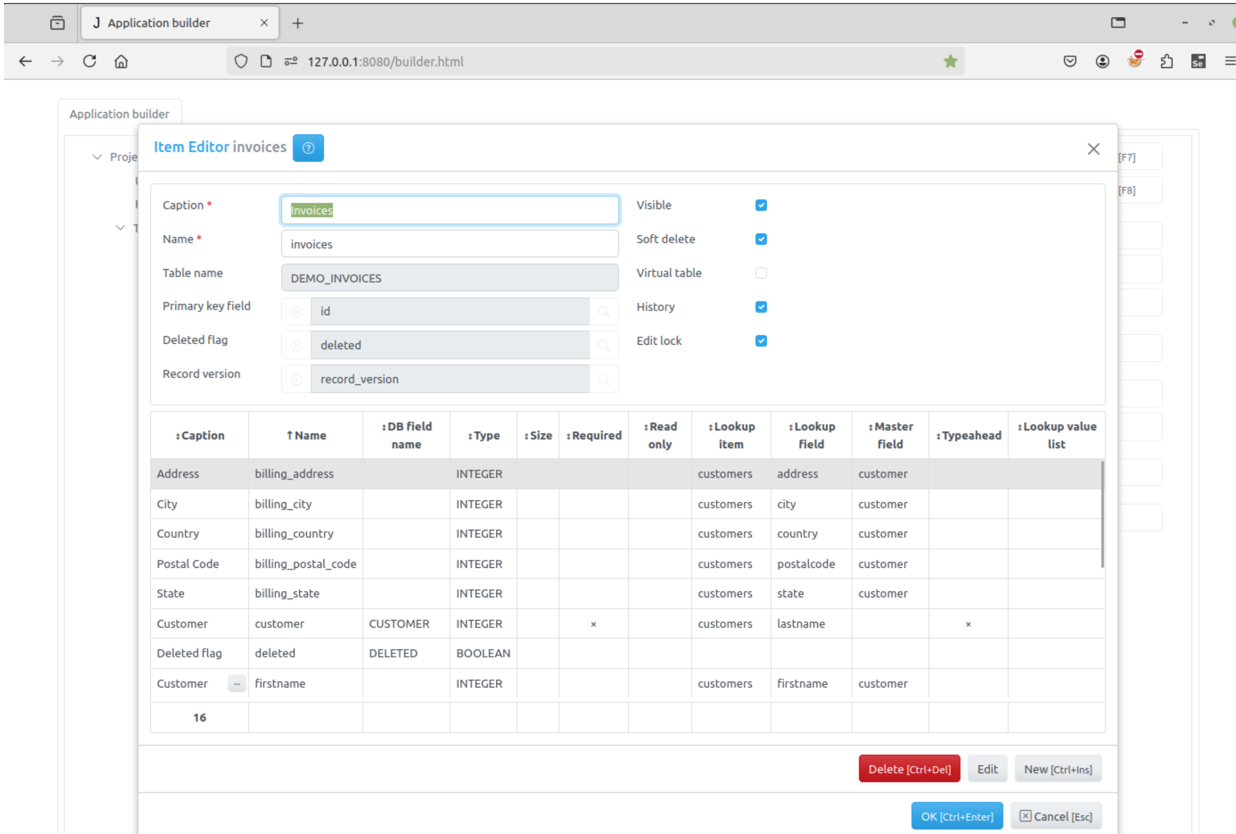
Jam.py 使用乐观锁定模型，也称为乐观并发控制。

当应用程序执行*edit_record* 方法时，它会从服务器接收记录的当前版本并保存它。当用户开始保存记录时，服务器应用程序会检查记录的当前版本。如果它与存储的值不同（另一个用户在编辑记录时更改了它），应用程序会警告用户并禁止保存。

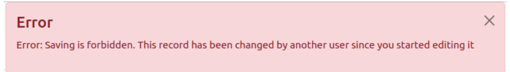
这个记录锁定机制非常容易实现。

为此，请创建一个用于存储记录版本的表字段。字段类型为 整数 (`integer`) 。

之后，我们可以在实体项编辑器对话框 中设置 **编辑锁定 (Edit lock)** 属性：

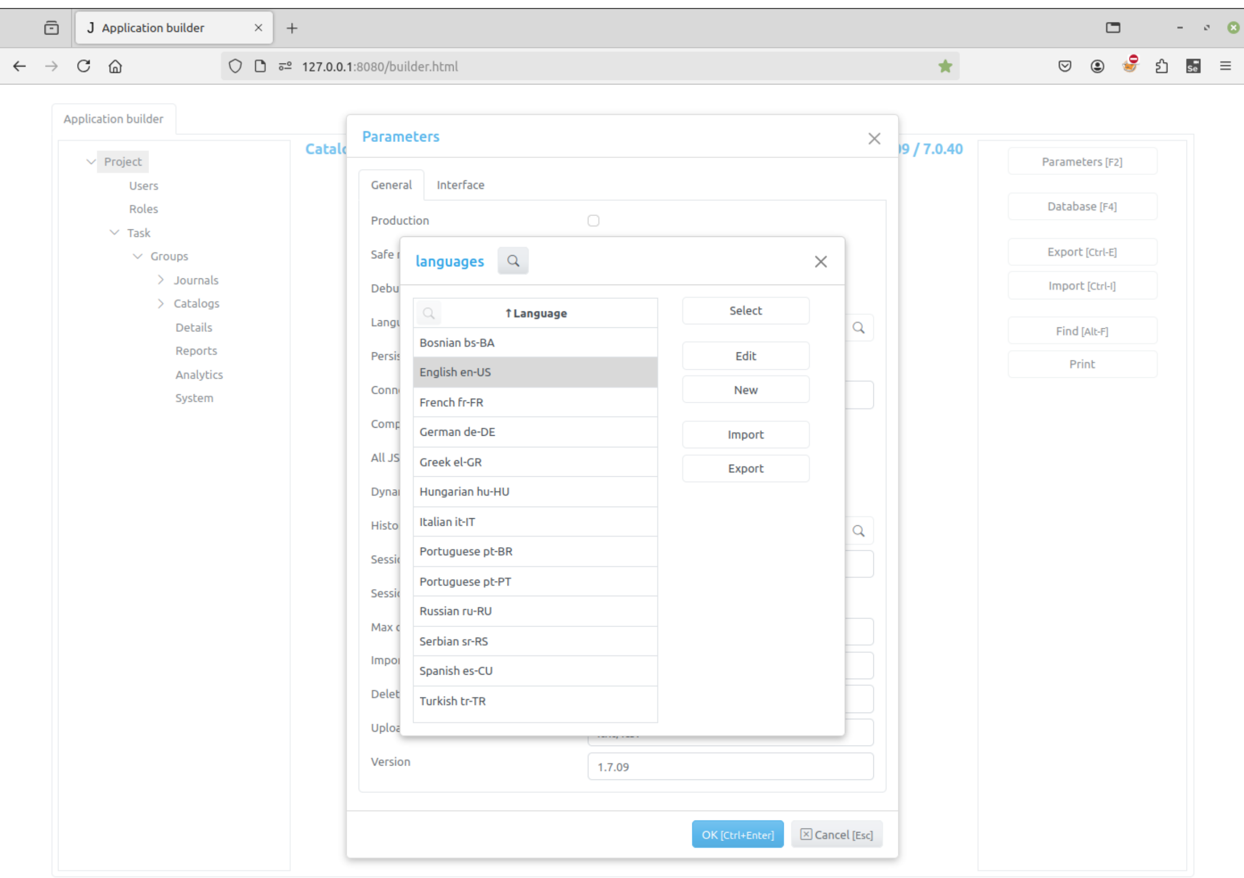


为锁定记录显示的消息：



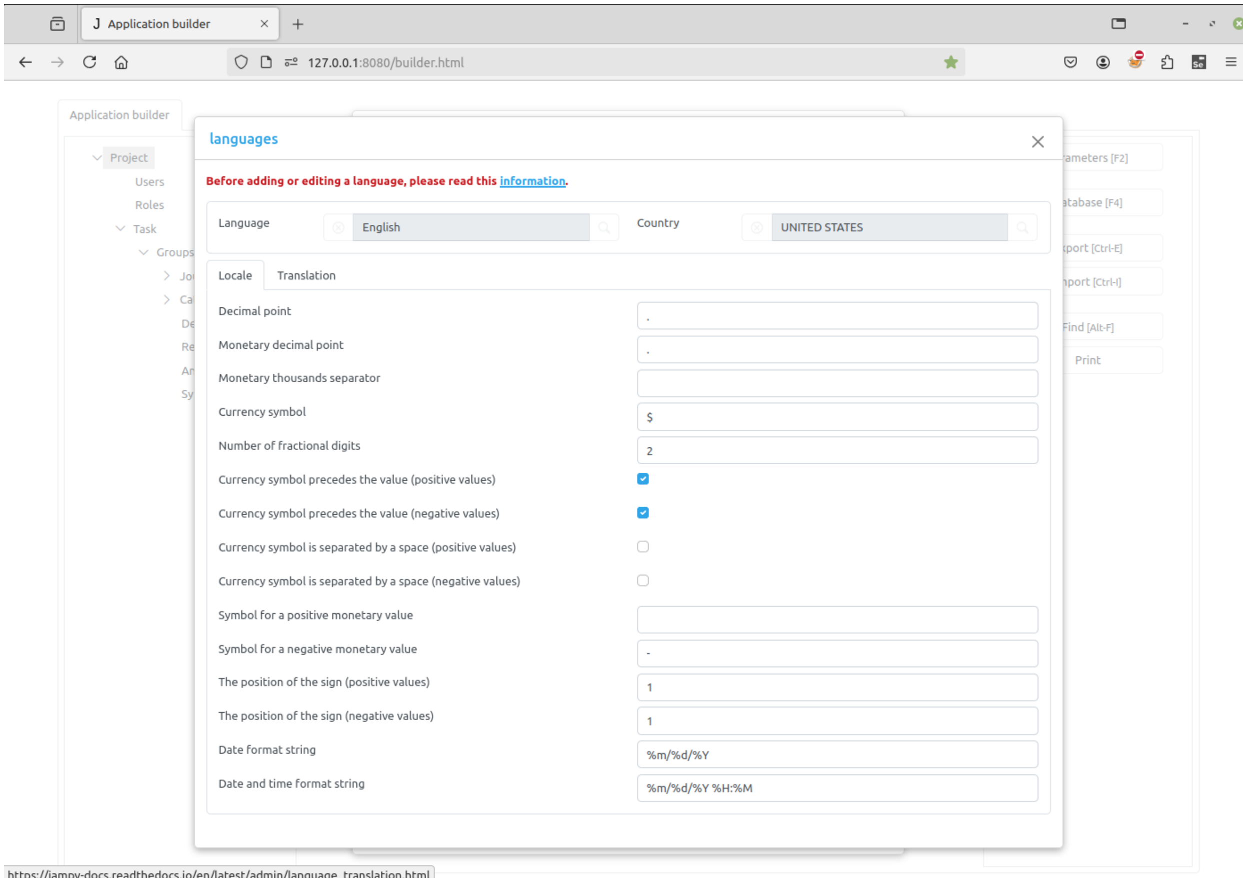
6.13 语言支持

使用语言对话框来添加、选择和更改您的语言。



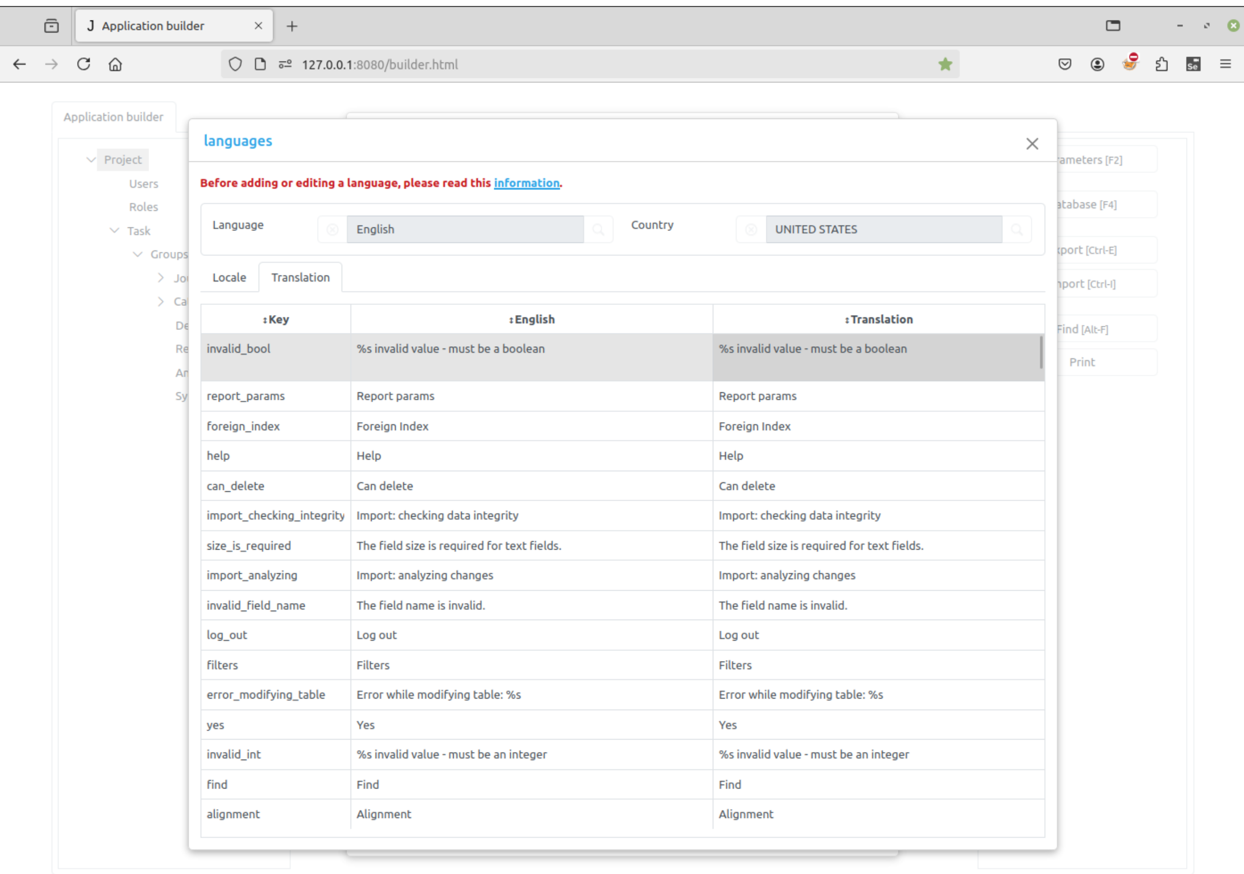
6.13.1 语言区域设置

使用语言区域设置来配置字段值的显示方式。请参阅*display_text*



6.13.2 语言翻译

请参阅语言翻译



6.14 语言翻译

所有语言翻译都存储在“jam”软件包项目下的的 langs.sqlite 数据库中。

> **备注**
>
> 因此，如果您对翻译数据库进行了一些更改，并安装了新版本的软件包，您将使用新软件包的翻译数据库，其中 **将不包含您所做的更改**。
>
> **请务必将您的翻译导出到文件中!!!**

如果您希望您的语言翻译被包含在 Jam.py 软件包中，请将其导出到文件，并在 GitHub 上联系软件包维护者以将其包含进去。或者，将文件发送到 Jam.py 邮件组。

请注意，Jam.py 在不断发展，提交您的翻译后，您可能需要在将来进行必要的更改。如果您不介意，您将被列入贡献者名单。

> **备注**
>
> 请勿更改以下符号 **%**、**%(item)s**、**%(field)s**、**%(filters)s**
>
> 例如
>
> **英文**:

Can’t delete the field %(field)s. It’s used in field definitions:%(fields)s

哈萨克语翻译：

Нельзя удалить поле %(field)s. Используется в определении полей:%(fields)s

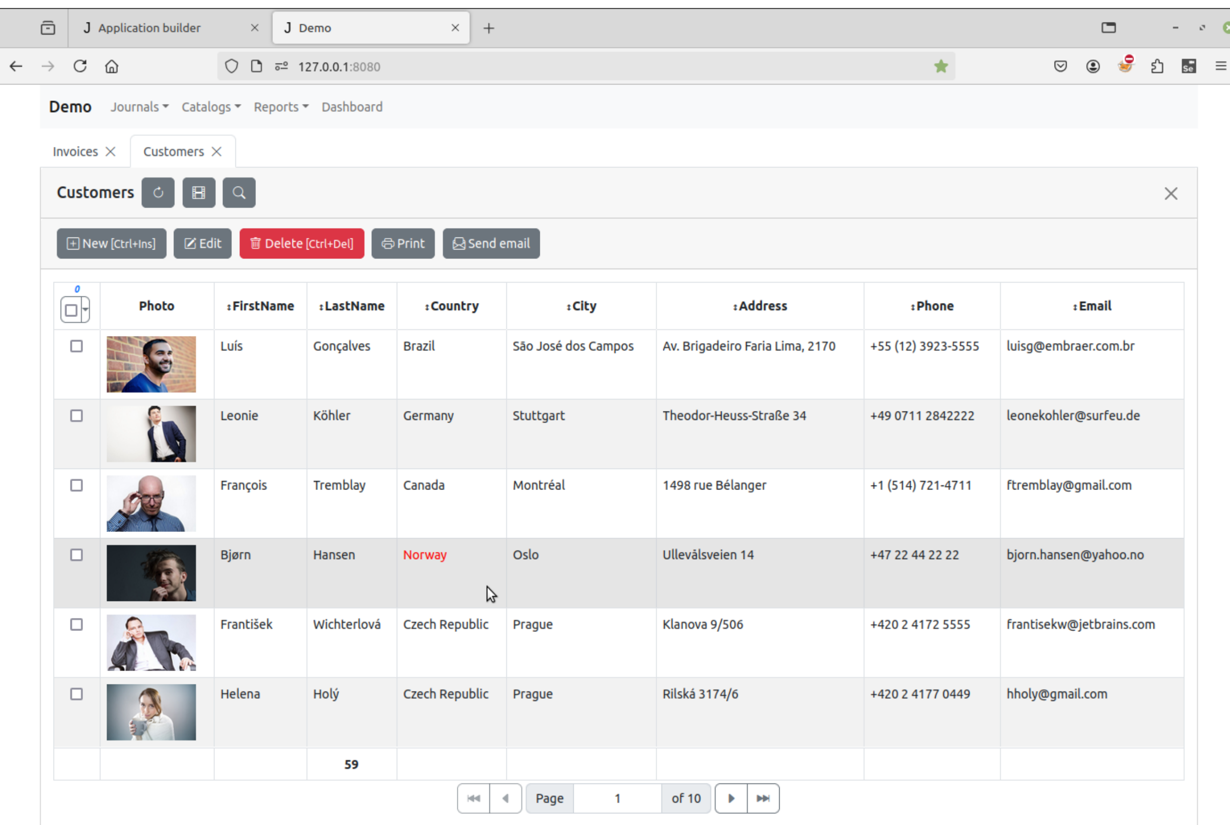
6.15 数据转义清理

为了防止跨站脚本（XSS）攻击，Jam.py 会转义清理在表格列中显示的字段值。

例如，如果字段包含以下文本：

```
<span style='color: red'>Norway</span>
```

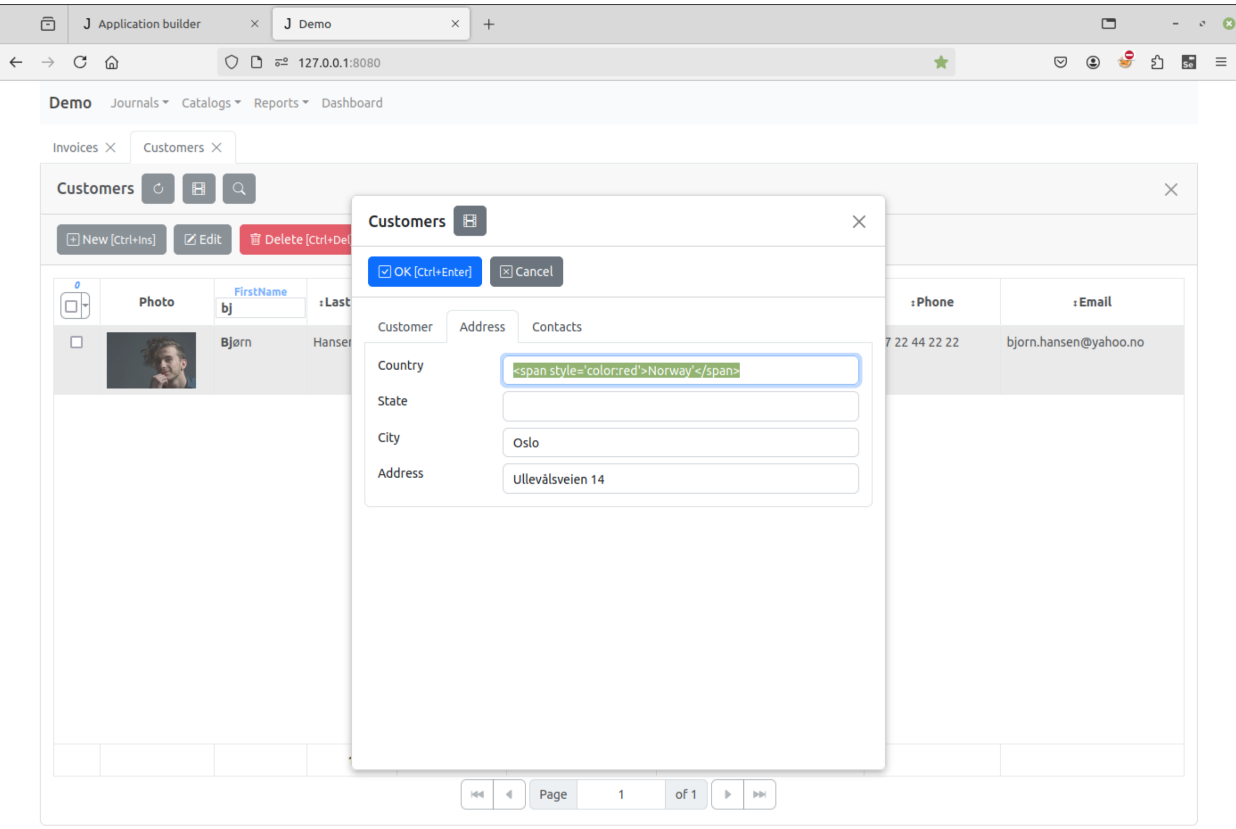
当未经过转义清理时，它将在表格列中显示如下：



当字段文本经过转义清理后，它会转换为以下形式：

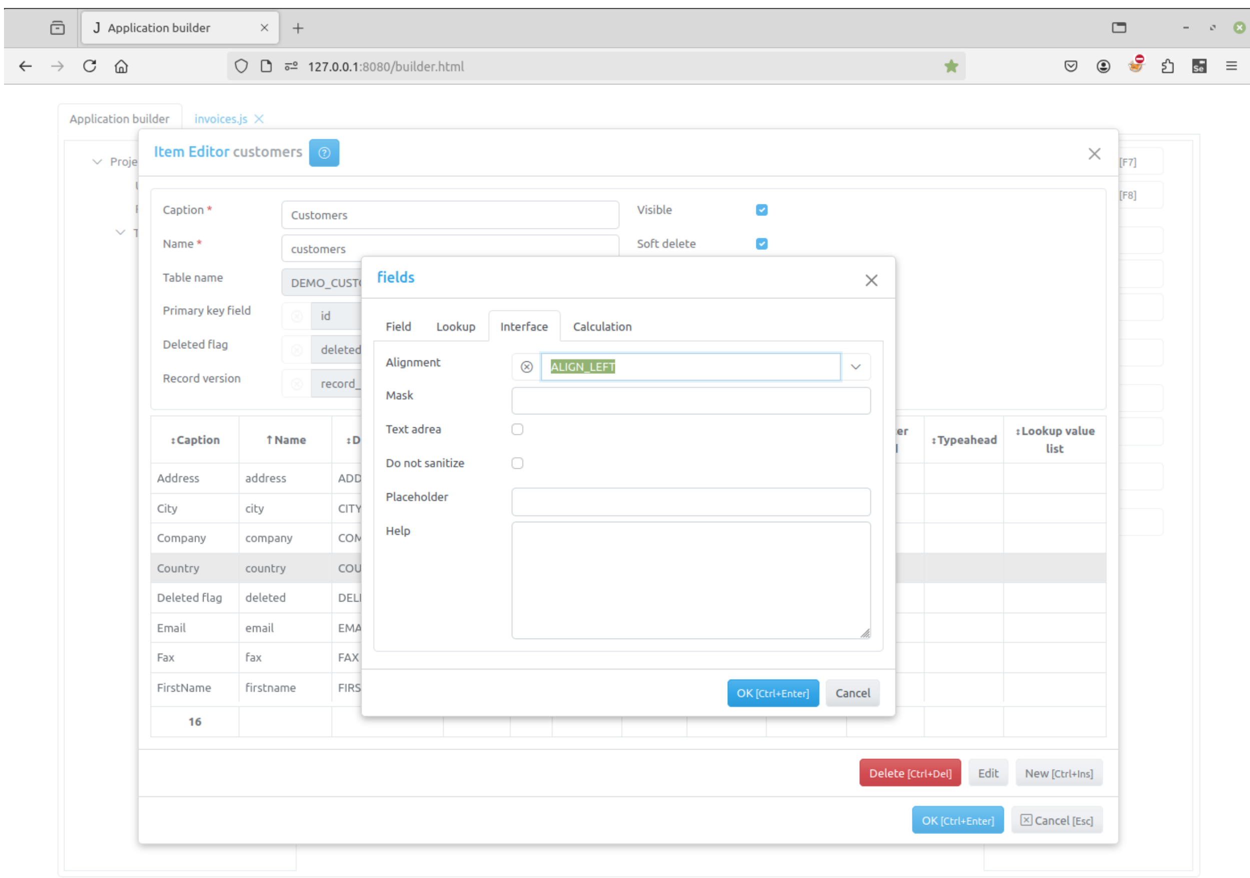
```
"&lt;span style='color: red'&gt;Norway&lt;/span&gt;"
```

如您所见，符号“<”和“>”被替换为“<”和“>”表格列将按以下列方式显示：



有两种方法可以防止转义清理。

第一种是在字段编辑器对话框 的“界面 (Interface)”选项卡中设置 **不转义清理 (Do not sanitize)** 属性。



第二种是编写*on_field_get_html* 事件处理程序。如果此事件处理程序返回一个值，则字段的值不会被转义清理。

6.16 可接受的字符串

接受字符串可以是以下值的组合，用逗号分隔。

值	描述
file_extension	指定文件扩展名（例如：.gif、.jpg、.png、.doc）
audio/*	所有音频文件
video/*	所有视频文件
image/*	所有图像文件

例如

```
.pdf, .xls

image/*, .pdf, .xls

audio/*

audio/*, video/*
```

6.17 路由

Jam.py v7 引入了路由功能。由于 Jam.py v5 是 SPA（单页应用程序），因此不需要路由。另一方面，例如，需要在 Jam.py v5 中实现用户注册页面。

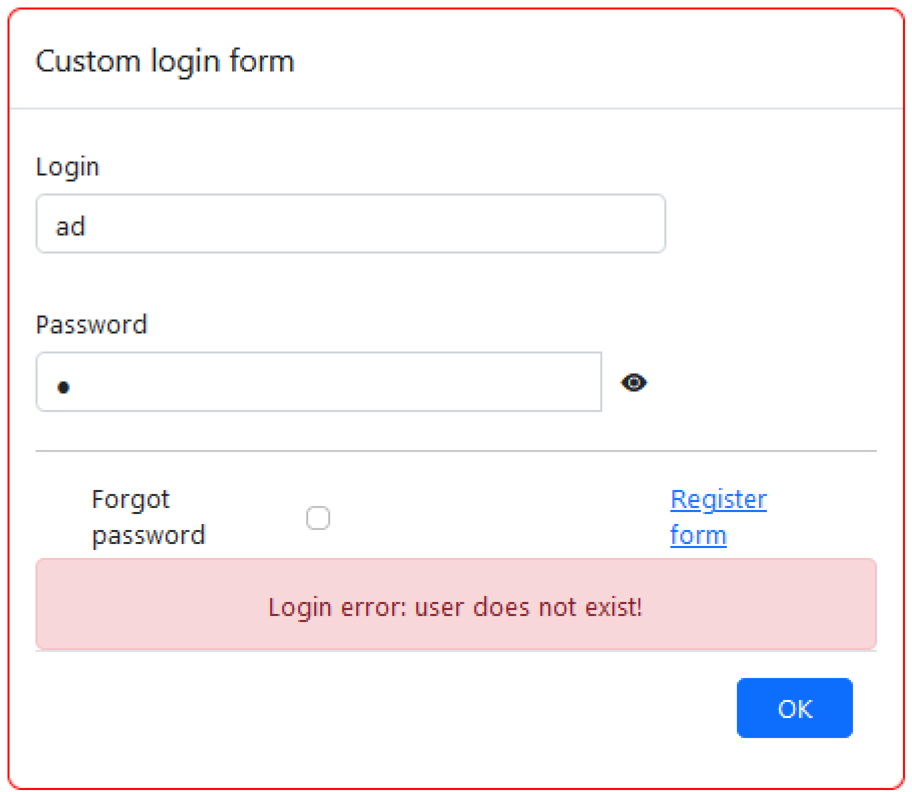
此问题的解决方案类似于创建注册表单 。

如您所见，**register.html** 文件使用了 JavaScript AJAX 代码。因此，任何需要与数据库交互的附加页面都需要类似的方法。

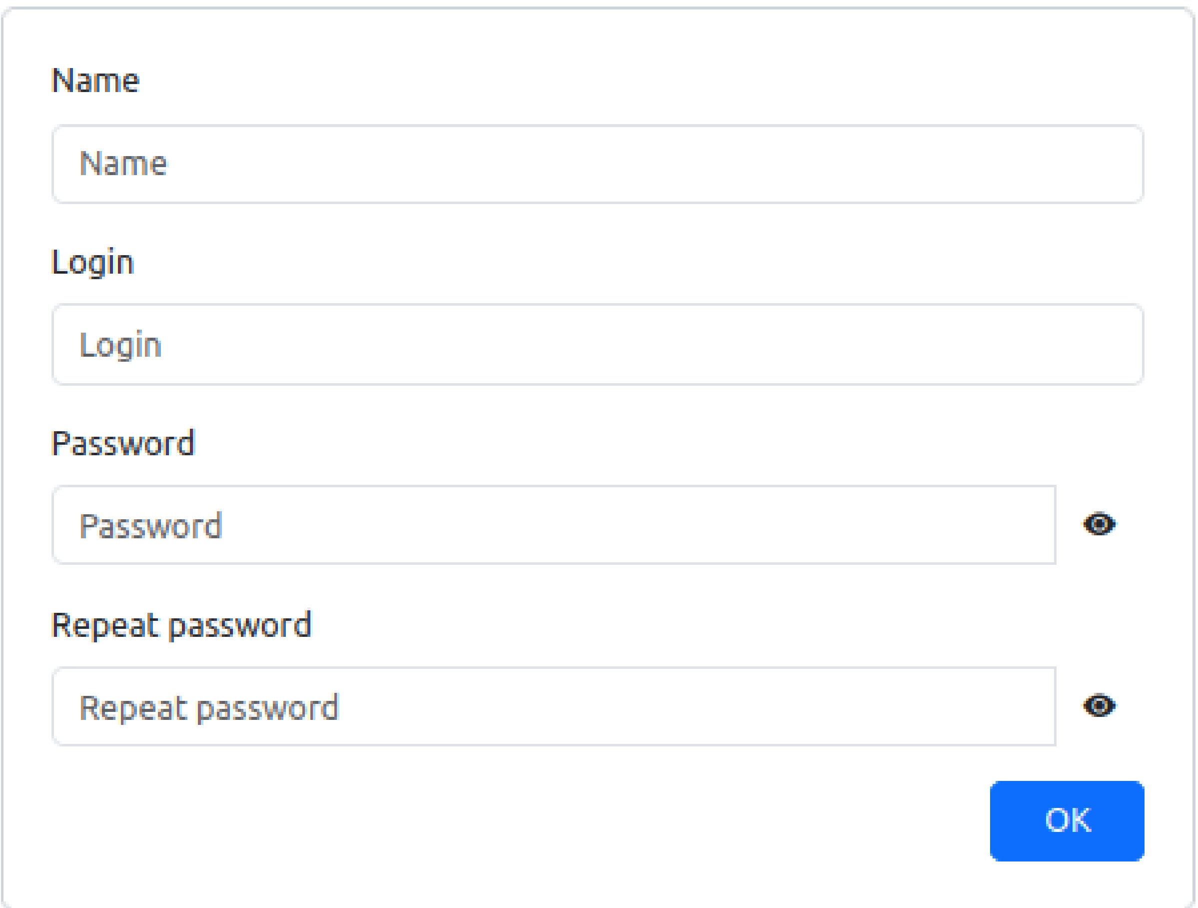
在 Jam.py v7 中，解决方案是在 `on_request` 事件处理程序中处理每个请求，并对有效请求定义响应。

这使我们能够创建自定义的登录和注册表单，其中可以创建自定义错误信息。例如：" 用户不存在！"、" 密码错误" 等。

新的 **login.html** 页面显示的错误消息：



新的 **register.html** 页面（无需 AJAX）：



6.17.1 路由代码

如前所述，使用 on_request 事件。

以下代码应存在于“任务/服务器模块”中。为便于阅读，以下未包含 Python 逻辑：

```
def on_request(task, request):
        parts = request.path.strip('/').split('/')
        if not parts[0]:
            if task.logged_in(request):
                return task.serve_page('index.html')
            else:
                return task.redirect('/login.html')
        elif parts[0] == 'login.html':
                                .
                                .

        elif parts[0] == 'register.html':
                                .
                                .
                    return task.serve_page('register.html')
                                .
                                .
```

(续下页)

(接上页)

```
                    return task.redirect('/login.html')
```

提供 robots.txt 文件的工作示例:

```
def on_request(task, request):
    parts = request.path.strip('/').split('/')
    if not parts[0]:
        if task.logged_in(request):
            return task.serve_page('index.html')
        else:
            return task.redirect('/login.html')
    elif parts[0] == 'robots.txt':
        if task.logged_in(request):
            return task.serve_page('robots.txt')
```

`robots.txt` 文件应存在于应用程序文件夹内。

6.17.2 另请参阅

serve_page

redirect

on_request

6.18 自定义构建器

以下是 Andrew 的笔记:

自定义应用程序构建器

应用程序构建器项目位于发行版的"builder"文件夹中。该项目用于使用旧版本创建新版本的应用程序构建器。该项目包含了进一步开发构建器所需的一切。应用程序像任何其他 Jam.py 应用程序一样启动。开发过程类似于常规 Jam.py 应用程序的开发,只是所有服务器代码必须位于任务服务器模块中。通过 server 方法从客户端代码调用的服务器模块中的所有函数,都应使用 register 方法在模块末尾的 register_events 函数中注册,并且服务器实体项的事件应在此函数中定义。在进行更改并测试后,使用"准备文件 (Prepare files)"按钮。应用程序将在 jam_files 文件夹中创建必要的文件。如果该文件夹不存在,将创建一个。此文件夹的内容应复制到 Jam.py 发行版软件包的 jam 文件夹中。

访问应用程序的构建器应用程序

当使用 wsgi.py 创建 Web 应用程序时,会创建构建器任务树——admin 对象。admin 使用 get_info 方法从点击"准备文件 (Prepare files)"按钮时保存的 builder_structure.info 文件加载任务。这是在软件包的 admin 文件夹中的 admin.py 模块中完成的。当 Web 应用程序收到第一个请求时,会创建项目的任务树。软件包 admin 文件夹中的 task.py 模块包含创建项目任务树的代码。它使用管理员用户从 admin.sqlite 数据库读取数据。为了加快过程,对应表中的信息被加载到字典中。

首先,请从 fork Jam.py-v7 开始。

接下来,克隆你的 fork:

```
...\> git clone https://github.com/YourGitHubName/jam-py-v7.git
```

并启动服务器

```
...\> cd jam-py-v7/builder
...\> ./server.py
```

打开 Web 浏览器，并在地址栏中输入

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

这是应用程序构建器的源代码。因此，在构建应用程序时，`builder.html` 页面内显示的所有内容，都是修改此处内容后的直接结果。

打开 Web 浏览器，并在地址栏中输入

```
127.0.0.1:8080
```

这是应用程序构建器的内容，像任何 Jam.py 应用程序一样显示。但是，它有一个额外的功能，即 `Prepare files`（准备文件）按钮。

应用程序的功能和外观直接取决于已安装的 Jam.py 版本。如果在未安装较新的运行时文件的情况下，在构建器中所做的任何更改都不会在此处显示。

自定义完成后，点击"准备文件 (Prepare files)"，我们将所有文件从 `jam_files` 文件夹复制到发行版文件夹中：

```
...\> cp -fr jam_files/* ../jam/.
```

在 MS Windows 上：

```
xcopy jam_files\* ..\jam\.  /E /H /C /I /Y
```

要增加 Jam.py 的版本号，我们编辑以下文件

```
...\> vi ../jam/__init__.py
```

现在，我们可以像往常一样安装更新后的版本，并使用新的版本号。

如果一切顺利，提交并创建一个包含更改的 Github `pull` 请求。

对于项目维护者，要启动 Github Actions，请提供以下标签号

```
...\> git push && git tag 7.0.XX && git push --tags
```

CHAPTER 7

Jam.py 类的参考

服务端采用 Python 实现，使用 Werkzeug 库；客户端则采用 JavaScript 编写，利用 JQuery 和 Bootstrap 框架.

7.1 客户端的 (javascript) 类参考

框架的所有对象组成一个任务树 。

下面是每种任务树对象的类:

7.1.1 AbstractItem 类

`class AbstractItem()`

使用范围: client

编程语言: javascript

AbstractItem 类是任务树 中所有实体项对象的祖先。

下面列出了这个类的属性和方法。

属性

ID

`ID`

使用范围: client

语言: javascript

所属类*AbstractItem*

描述说明

ID 属性在项目的框架里的所有 id 中是唯一的。

ID 属性在按数字而不是按名称引用项目时最有用。它也在框架内部使用。

item_caption

item_caption

使用范围: client

编程语言: javascript

父类: *AbstractItem*

描述

item_caption 属性指定实体项在界面中显示给用户的名称

item_name

item_name

使用范围: client

编程语言: javascript

父类: *AbstractItem*

描述

指定实体项在代码中被引用的名称，在代码中，使用 item_name 属性来引用实体项。

item_type

item_type

使用范围: client

编程语言: javascript

父类: *AbstractItem*

描述说明

指定实现项的类型。使用 type 属性来获取实现项的类型。

实现项的类型可以是以下值:

- "task"
- "items"
- "details"
- "reports"
- "item"
- "detail_item"

- “report”
- “detail”

items

items

使用范围: client

编程语言: javascript

父类: *AbstractItem*

描述说明

列举出（包含）某个实体项的所有子项。

使用 items 属性来访问实体项所拥有某个的子项。

owner

表示拥有此某个实体项的实体项（即父项）。

owner

使用范围: client

编程语言: javascript

父类: *AbstractItem*

描述说明

使用 owner 属性来引用某个实体项的拥有者（即父项）。

task

表示拥有该实体项的任务树 的根节点。

task

使用范围: client

编程语言: javascript

父类: *AbstractItem*

描述说明

使用 task 属性来引用拥有该实体项的任务树 的根节点。

方法

abort

abort (*message*)

使用范围: client

编程语言: javascript

父类: *AbstractItem*

描述说明

使用 **abort** 方法来抛出异常。

当你需要中止某些"on_before"事件的执行时，它会很有用。

示例

下面的代码会抛出带有文本内容的异常：

执行被中止：invoice_table - 需要一个数量值

```
function on_before_post(item) {
    if (item.quantity.value === 0) {
        item.abort('a quantity value is required');
    }
}
```

alert

alert (*mess, options*)

使用范围: client

编程语言: javascript

父类: *AbstractItem*

描述说明

> 使用 `alert` 方法，创建一个显示在应用程序右上角的消息弹出框，用鼠标在页面上任意地方单击后，它会消失。

The `mess` parameter specifies the text that will be displayed. 使用 `mess` 参数指定将要显示的文本。

The `options` parameter is an object with the following attributes: `options` 参数是一个带有下列属性的对象：

- `type` - 指示消息的类型——字体、背景颜色和标题文本，如果未指定 `header` 参数。必须是以下之一：
  - 'info',
  - 'error',
  - 'success'

  默认值是'info'
- `header` - 指定消息弹出框的标题
- `pulsate` - 如果为 true ， 标题将会闪动，默认为 true
- `show_header` - 如果为 false, 将不显示标题

`alert_error` 和 `alert_success` 和 `alert` 相同，它们自身就有相应的 `type` 类型。

示例

```
item.alert_error('Failed to send the mail: ' + err);
item.alert('Successfully sent the mail');
```

can_view

can_view()

使用范围: client

编程语言: javascript

父类: *AbstractItem*

描述说明

在勾选了项目的安全模式参数 后，使用 **can_view** 方法就能确定某个用户是否有权限访问实体项 的数据集，或生成的报表 结果。如果为勾选项目的安全模式参数 ，该方法总是返回 true。

用户权限是项目树中角色节点 的子集。

示例

```
if (item.visible && item.can_view()) {
    $("#submenu")
        .append($('<li></li>')
        .append(
            $('<a href=""></a>')
                .text(item.item_caption)
                .data('item', item);
        )
    );
}
```

each_item

each_item(*function(item)*)

使用范围: client

编程语言: javascript

父类: *AbstractItem*

描述说明

使用 each_item 方法来迭代访问某个实体项的*items* 属性中的实体项。

each_item() 可以指定一个函数来对每个子项进行处理（子项作为参数传递给该函数）。

通过使回调函数返回 false，你能在某个特定的迭代内中断 each_item 方法的循环，

示例

以下代码将在浏览器的命令行中输出项目的全部主表目录（catalogs）:

```
function on_page_loaded(task) {
    task.catalogs.each_item(function(item) {
        console.log(item.item_name);
    })
}
```

hide_message

hide_message(*form*)

使用范围: client

编程语言: javascript

父类: *AbstractItem*

描述说明

使用 **hide_message** 方法来关闭一个由*message* 方法创建的模态对话框。

form 参数是一个由*message* 方法返回的 JQuery 对象。

item_by_ID

item_by_ID(*ID*)

使用范围: client

编程语言: javascript

父类: *AbstractItem*

描述说明

item_by_ID 在项目的任务树 的所有实体项中进行搜索，从当前条目开始，查找其*ID* 属性等于 ID 参数的实体项。

load_module

load_module(*callback*)

使用范围: client

编程语言: javascript

父类: *AbstractItem*

描述说明

在执行回调前，**load_module** 方法将动态加载一个实体项的 javascript 文件。

该方法检查模块是否已被加载。如果没有模块，该方法会从服务端加载模块并初始化实体项，然后执行 **callback** 函数，否则仅执行 **callback** 函数。实体项会被作为参数被传递给回调函数。

发送到服务端的请求使被异步执行的。

示例

下面，只有实体项模块被加载后，才执行 do_some_work 函数:

```
function some_work(item) {
    item.load_module(do_some_work);
}

function do_some_work(item) {
    // some code
}
```

另请参见

使用模块

load_modules

load_script

load_modules

load_modules(*module_array*, *callback*)

使用范围: client

编程语言: javascript

父类: *AbstractItem*

描述说明

在执行 **callback** 前，**load_modules** 方法会动态加载指定的多个模块。该方法与*load_module* 的作用类似，不同处是会加载并初始化 **module_array** 中指定的实体项的所有模块。

示例

下面，只有该实体项和它父项所拥有的模块都被加载后，才执行 do_some_work 函数:

```
function some_work(item) {
    item.load_modules([item, item.owner], do_some_work);
}

function do_some_work(item) {
    // some code
}
```

另请参见

使用模块

load_module

load_script

load_script

load_script(*js_filename*, *callback*, *onload*)

使用范围: client

编程语言: javascript

父类: *AbstractItem*

描述说明

在执行回调前，**load_script** 方法将动态地从服务端加载一个实体项的 javascript 文件。

该方法检查文件是否已被加载。如果没有模块，该方法会从服务端加载，再执行指定地 onload 方法，然后执行 **callback** 函数，否则仅执行 **callback** 函数。实体项会被作为参数被传递给回调函数。

js_filename 应该指明 javascript 在服务端项目的目录中的相对路径。

发送到服务端的请求使被异步执行的。

示例

下面，只有当服务端从 *js* 目录中加载 *lib.js* 后，才会执行 do_some_work 函数：

```
function some_work(item) {
    item.load_script('js/lib.js', do_some_work);
}

function do_some_work(item) {
    // some code
}
```

另请参见

使用模块

load_module

load_modules

message

message(*mess*, *options*)

使用范围: client

编程语言: javascript

父类: *AbstractItem*

描述说明

使用 **message** 方法创建一个模态窗体表单。

用 **mess** 参数指定显示在表单中的文本或 html 内容

options 参数是一个对象，它有以下属性：

- **title** - 表单的标题

- **width** - 表单的宽度，默认值为 400px
- **height** - 表单的高度
- **margin** - 使用此属性来定义表单主体的外边距
- **text_center** - 如果为 true，表单主体中的文本元素将居中显示，默认为 false 。
- **buttons** - 一个定义了将在表单底部创建的按钮的对象，对象的键是按钮名称，值是按钮被点击时将执行的函数。
- **button_min_width** - 按钮的宽度的最小值，默认值为 100px 。
- **center_buttons** - 如果为 true，按钮将居中显示，默认为 false 。
- **close_button** - 如果为 true，应用程序将在表单窗体的右上角创建一个关闭按钮，默认为 true 。
- **close_on_escape** - 如果为 true，当用户按下 Escape 按键时，会关闭表单窗体，默认为 true 。
- **print** - 如果为 true，应用程序将在表单窗体的右上角创建一个打印按钮，以便打印表单的主体内容，默认为 false 。

该方法返回表单的一个 jquery 对象。要通过编程的方式关闭表单，请将返回的对象传递给*hide_message* 方法。

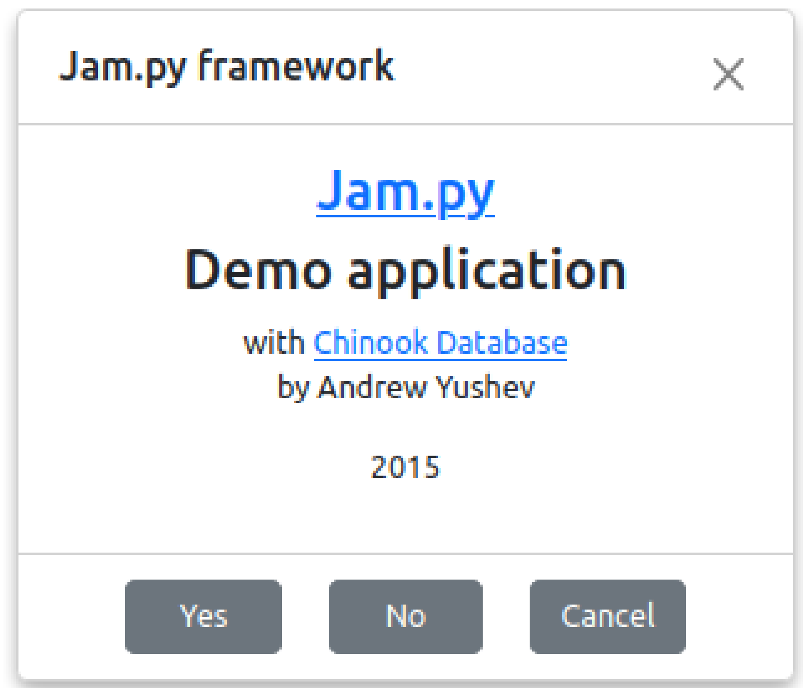
Examples

下面的代码将创建一个有“是、否、取消”的对话框：

```
function yes_no_cancel(item, mess, yesCallback, noCallback, cancelCallback) {
    var buttons = {
        Yes: yesCallback,
        No: noCallback,
        Cancel: cancelCallback
    };
    item.message(mess, {buttons: buttons, margin: "20px",
        text_center: true, width: 500, center_buttons: true});
}
```

```
task.message(
    '<a href="http://jam-py.com/" target="_blank"><h3>Jam.py</h3></a>' +
    '<h3>Demo application</h3>' +
    ' with <a href="http://chinookdatabase.codeplex.com/" target="_blank">Chinook Database</a>'
↪+
    '<p>by Andrew Yushev</p>' +
    '<p>2015</p>',
    {title: 'Jam.py framework', margin: 0, text_center: true, buttons: {"Yes": undefined, "No":
↪undefined, "Cancel": undefined},
        center_buttons: true}
);
```

上面代码的结果如下：



question

创建一个带有 **是 (Yes)** 和 **否 (No)** 按钮的模态表单窗体

question (*mess*, *yes_callback*, *no_callback*, *options*)

使用范围: client

编程语言: javascript

父类: *AbstractItem*

描述说明

使用 **question** 方法创建一个带有 **是 (Yes)** 和 **否 (No)** 按钮的模态表单窗体

mess 参数指定要在表单窗体主题显示的文本或 html 内容。

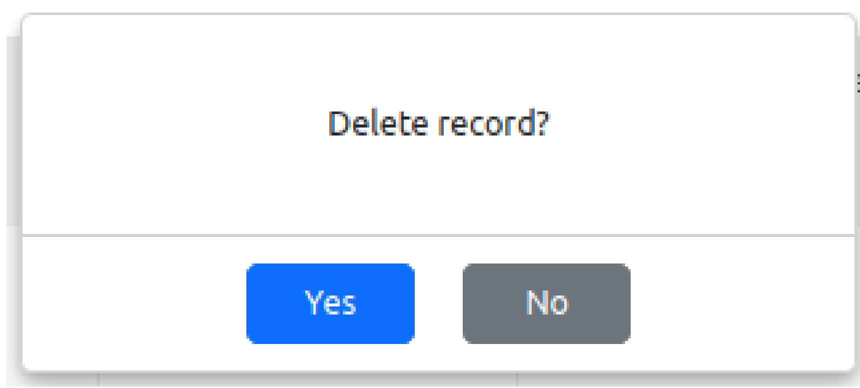
如果指定了 **yes_callback** ， **no_callback** 函数，它们会在用户点击了 **是 (Yes)** 或 **否 (No)** 按钮后被执行，然后会自动关闭表单窗体。

示例

下面的代码将创建一个模态表单窗体，当用户用鼠标点击 "是 (Yes)" 按钮时，将删除已选择的记录。

```
item.question('Delete record?',
    function() {
        item.delete();
    }
);
```

上面代码的结果如下:



server

server (*func_name*, *params*, *callback*)

使用范围: client

编程语言: javascript

父类: *AbstractItem*

描述说明

使用 sever 方法在客户端可以执行一个在实体项的服务端模块中定义的函数。

sever 方法会执行一个在实体项的服务端模块中定义的函数，函数名称形式为 func_name ，而且可以在 sever 方法的 参数 (params) 中指定函数需要的参数。

如果指定了回调，服务器上的函数将异步执行，随后使用服务器函数执行结果作为参数执行 callback ，否则函数将同步执行并返回服务器函数的结果。

如果函数在服务端上的操作抛出异常，并且未传递回调参数（同步执行），客户端将抛出异常。如果存在回调参数，它将作为参数传递给回调。

当服务端函数执行期间引发异常时，客户端的应用程序会抛出包含服务端异常文本的异常。

服务器上函数的第一个参数必须是 item，它必须跟随客户端函数中指定的参数。

参数 (params) 是一个参数列表，如果没有参数，可以省略 参数 (params) 。

示例

在 发票 *(Invoices)** 业务台账的服务端模块中，定义函数：

```
def get_total(item, id_value):
    result = 0;
    copy = item.copy()
    copy.set_where(id=id_value)
    copy.open()
    if copy.record_count():
        result = copy.total.value
    else:
        raise Exception, 'Journal "invoices" does not have a record with id  %s' % id_value
    return result;
```

下面在 发票 *(Invoices)** 业务台账的客户端模块的代码中，将调用这个服务端的函数：

```
task.invoices.server('get_total', [17], function(total, err) {
    if (err) {
        throw err;
    }
    else {
        console.log(total);
    }
});
```

warning

warning(*mess*, *callback*)

使用范围: client

编程语言: javascript

父类: *AbstractItem*

描述说明

使用 **warning** 方法创建一个带有 **确定 (Ok)** 按钮的模态表单窗体

mess 参数指定要在表单窗体主题显示的文本或 html 内容。

如果指定了 **callback** 函数，它们会在用户点击了按钮后被执行，然后会自动关闭表单窗体。

示例

```
item.warning('No record selected.');
```

yes_no_cancel

yes_no_cancel(*mess*, *yes_callback*, *no_callback*, *cancel_callback*)

使用范围: client

编程语言: javascript

父类: *AbstractItem*

描述说明

使用 **yes_no_cancel** 方法创建一个带有 **是 (Yes)** 、**否 (No)** 和 **取消 (Cancel)** 按钮的模态表单窗体

mess 参数指定要在表单窗体主题显示的文本或 html 内容。

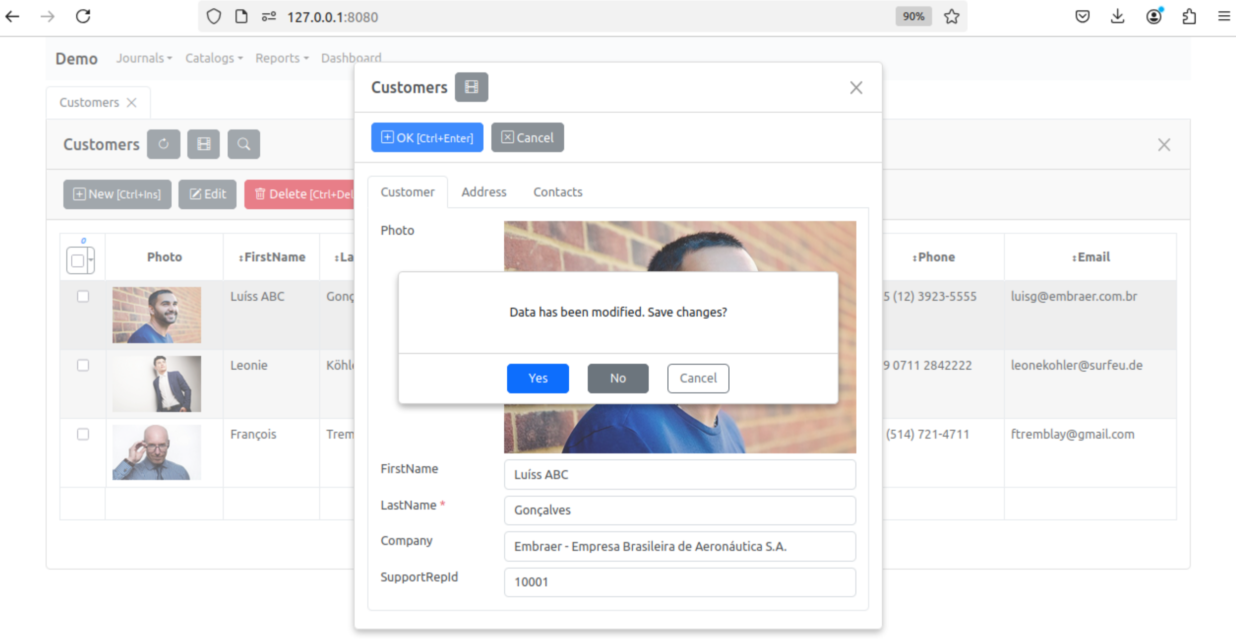
如果指定了 **yes_callback** ，**no_callback** ，**cancel_callback** 函数，它们会在用户点击了 **是 (Yes)** 、**否 (No)** 或 **取消 (Cancel)** 按钮后被执行，然后会自动关闭表单窗体。

示例

当用户点击实体项编辑窗体右上角的关闭按钮时，会执行下面的代码；

```
function on_edit_form_close_query(item) {
    var result = true;
    if (item.is_changing()) {
        if (item.is_modified()) {
            item.yes_no_cancel('Data has been modified. Save changes?',
                function() {
                    item.apply_record();
                },
                function() {
                    item.cancel_edit();
                }
            );
            result = false;
        }
        else {
            item.cancel();
        }
    }
    return result;
}
```

上面的代码结果如下：



7.1.2 Task Task 类

class Task()

使用范围: client

编程语言: javascript

Task 类用于创建项目的任务树 。

下面列出了的这个类的属性、方法和事件。

它同样继承了其祖先类*AbstractItem* 类 的属性和方法。

属性

forms_container

forms_container

使用范围: client

编程语言: javascript

父类: *Task*

描述说明

forms_container 是一个 JQuery 对象，应用程序会在它里面创建表单窗体。

要初始化 forms_container 属性，请使用*set_forms_container* 方法或*create_menu* 方法。

默认代码使用了*create_menu* 方法。

另请参见

forms_in_tabs

create_menu

set_forms_container

forms_in_tabs

forms_in_tabs

使用范围: client

编程语言: javascript

父类: *Task*

描述说明

如果设置了 forms_in_tabs 属性，并且指定了*forms_container* 属性，应用程序将在选项卡中创建表单窗体。

可以在参数 的 **界面 (Interface)** 选项卡中设置这个属性的值。

safe_mode

safe_mode

使用范围: client

编程语言: javascript

父类: *Task*

描述说明

检查 safe_mode 属性可以判断是否设置了项目的安全模式 参数

示例

```
function on_page_loaded(task) {

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

    task.tasks.view($("#content"));
}
```

另请参见

参数

user_info

on_page_loaded

templates

templates

使用范围: client

编程语言: javascript

父类: *Task*

描述说明

`templates` 属性中存储了项目的表单窗体的模板。

另请参见

表单窗体模板

表单窗体

user_info

user_info

使用范围: client

编程语言: javascript

父类: *Task*

描述说明

当设置了项目的安全模式参数，可以使用 user_info 属性来获取用户信息。

user_info 一个有下列属性的对象:

- user_id - 用户 id
- user_name - 用户名
- role_id - 用户所属角色的 id
- role_name - 分配给用户的角色
- admin - 如果为 true，那么用户能使用应用程序构建器.

如果 safe mode 为 false，那么 user_info 属性是一个空对象。

示例

```
function on_page_loaded(task) {
    $("#title").html('Jam.py demo application');
    if (task.safe_mode) {
        $("#user-info").text(task.user_info.role_name + ' ' + task.user_info.user_name);
        $('#log-out')
        .show()
        .click(function(e) {
            e.preventDefault();
            task.logout();
        });
    }
    // some initalization code
}
```

另请参见

load

login

logout

用户

角色

方法

add_tab

add_tab(*container*, *tab_name*, *options*)

使用范围: client

编程语言: javascript

父类: *Task*

描述说明

add_tab 方法为 container 创建一个选项卡。

container 是一个容器（container）元素的 JQuery 对象。

“tab_name“要创建的选项卡的名称。

用户能使用 options 指定可选参数。options 是一个有下列属性的对象：

- tab_id - 标识选项卡的唯一字符串
- show_close_btn - 如果将其设置为 true ，将出现能够用来关闭这个选项卡的“关闭按钮”。
- set_active - 如果将其设置为 true ，那么将激活新创建的选项卡（显示其内容）。
- on_close - 一个可以在单击关闭按钮时调用的回调函数。

该函数返回 JQuery 对象，它是带有 tab-pane 样式类的 div ，当选项卡被激活时，将显示其内容。

示例

下面的代码将为编辑“客户主表 (Customers catalog)”创建一个选项卡。代码中使用了*create_inputs* 方法。

```
function on_edit_form_created(item) {
    var container = item.edit_form.find('.tabs');
    task.init_tabs(container);
    item.create_inputs(task.add_tab(container, 'Customer'),
        {fields: ['firstname', 'lastname', 'company', 'support_rep_id']}
    );
    item.create_inputs(task.add_tab(container, 'Address'),
        {fields: ['country', 'state', 'address', 'postalcode']}
    );
    item.create_inputs(task.add_tab(container, 'Contact'),
        {fields: ['phone', 'fax', 'email']}
    );
}
```

下面是“客户主表 (Customers catalog)”的编辑 html 的模板

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

另请参见

init_tabs

close_tab

close_tab

close_tab(*container*, *tab_id*)

使用范围: client

编程语言: javascript

父类: *Task*

描述说明

使用 close_tab 方法关闭 container 中用 tab_id 标识的选项卡。

另请参见

init_tabs

add_tab

create_menu

create_menu: function(*menu*, *forms_container*, *options*)

使用范围: client

编程语言: javascript

父类: *Task*

描述说明

create_menu 方法将基于项目的*task 树* 创建一个菜单。

如果设置了*project parameters* 的在选项卡中显示表单窗体的属性，将初始化创建后被用于显示表单窗体的选项卡。

该方法将迭代*task 树* 的实体项，并将可见属性设置为 true 且用户有权限查看的实体项添加到菜单中。

该方法用于将点击事件分配给菜单项，当用户点击时，对于报表将执行*print* 方法，而对其他项目将执行*view* 方法。

可以把下列参数传递给该方法：

- menu - 来自 index.html 文件中的菜单元素的 JQuery 对象
- forms_container 一个 JQuery 对象，其元素包含由*view* 常见的表单窗体。
- options - 可以有下列属性的对象：
  - custom_menu - 使用这个选项来创建一个自定义菜单，请见下文
  - view_first - 如果为 " true"，则创建菜单后将显示菜单中第一个实体项的视图表单窗体，默认值为 false 。
  - create_single_group - 如果为 true，并且任务树中只有一个组有实体项，则将创建该组的菜单项，该菜单项具有组实体项的下拉菜单，否则将创建每个实体项的菜单项。默认值为 false" 。
  - splash_screen - 一个 html ，当关闭全部选项卡是，它将显示在 forms_container 中。

自定义菜单选项

要创建你自己的自定义菜单，你必须设置 custom_menu 选项。

这个选项是一个含有多个菜单对象的列表，其中每个菜单对象可以是：

- Jam.py 实体项或 or 实体项组
- array: 数组的第一个元素是菜单项的名称，第二个是菜单对象的列表
- 有一个属性的对象: 属性的“键”是菜单项的名称，而“值”是菜单对象的列表
- 有一个属性的对象: 属性的“键”是菜单项的名称，而“值”是一个点击菜单项将要执行的函数。

要添加一个分隔线，可以将一个空字符串（”）添加到菜单对象的列表中。

示例

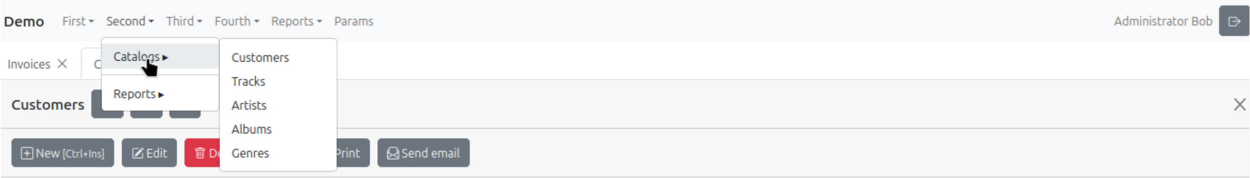
修改后“on_page_loloaded”如下所示：

```
task.create_menu($("#menu"), $("#content"), {
    custom_menu: menu,
    splash_screen: '<h1 class="text-center">Jam.py Demo Application</h1>',
    view_first: true
});
```

在同一个“on_page_looaded”中使用自定义菜单的示例：

```
let menu = [
        ['First',  [task.invoices, task.customers]],
        {'Second': [task.catalogs, '', task.reports]},
        {Third: [task.tracks, {Params: function() {alert('params clicked')}}]},
        {Fourth: [task.task.analytics, {'Artists list': [task.artists]}]},
        task.reports,
        {Params: function() {alert('params clicked')}},
    ];
task.create_menu($("#menu"), $("#content"), {
    custom_menu: menu,
    splash_screen: '<h1 class="text-center">Jam.py Demo Application</h1>',
    view_first: true
});
```

对于演示应用程序，上面的代码会有如下结果：



init_tabs

init_tabs(*container*, *tabs_position*)

使用范围: client

编程语言: javascript

父类: *Task*

描述说明

init_tabs 方法为“容器 (container)”参数初始化选项卡。

container 是“容器 (container)”元素的一个 JQuery 对象。

tabs_position 参数指定由*add_tab* 创建的选项卡的位置。它是字符串，其值是下列值中的一个：

- tabs-below
- tabs-left
- tabs-right

If this parameter is omitted tabs will be positioned at the top of the container. 如果省略这个参数，选项卡将被放置在“容器”的顶部。

调用此方法后，你就能使用*add_tab* 方法来创建选项卡。

另请参见

add_tab

close_tab

load

load(*callback*)

使用范围: client

编程语言: javascript

父类: *Task*

描述说明

当浏览器加载 index.html 文件中的 jam.js 库时，jam.js 会创建一个空的“任务”对象。Load 方法从服务器加载项目的任务树，并对其进行初始化。之后，应用程序会触发*on_page_loaded* 事件。(参考工作流程)

示例

以下代码来自项目的 index.html 文件。

```
<script src="/jam/js/jam.js"></script>
<script src="/js/events.js"></script>

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

另请参见

login

logout

user_info

用户

角色

login

login(*callback*)

使用范围: client

编程语言: javascript

父类: *Task*

描述说明

`login` 方法使用在 index.html 文件模板中定义的登录表单窗体 div 创建一个登录的表单窗体。当设置了项目的安全模式参数 后，该方法由 doc:'load <m_load>'方法调用。

另请参见

load

logout

user_info

用户

角色

logout

logout()

使用范围: client

编程语言: javascript

父类: *Task*

描述说明

调用 `logout` 方法使一个用户退出。

示例

```
function on_page_loaded(task) {
    $("#title").html('Jam.py demo application');
    if (task.safe_mode) {
        $("#user-info").text(task.user_info.role_name + ' ' + task.user_info.user_name);
        $('#log-out')
        .show()
        .click(function(e) {
            e.preventDefault();
            task.logout();
        });
    }
```

(续下页)

(接上页)

```
    // some initalization code
}
```

另请参见

load

login

user_info

用户

角色

set_forms_container

set_forms_container (*container*, *options*)

使用范围: client

编程语言: javascript

父类: *Task*

描述说明

`set_forms_container` 方法被用来初始化*forms_container* 属性，该属性将包含应用程序的表单窗体。

如果设置了*forms_in_tabs* 属性，应用程序也初始化用来显示表单窗体的选项卡。

`container` 是一个 JQuery 对象，它将被用作应用程序表单窗体的一个容器。

`options` 参数可以有如下属性:

- `splash_screen` - 当所有选项卡都关闭时，将在 forms_container 中显示的 html .

示例

```
task.set_forms_container($("#content"), {
    splash_screen: '<h1 class="text-center">Jam.py Demo Application</h1>'
});
```

另请参见

forms_container

forms_in_tabs

create_menu

upload

upload (*options*)

使用范围: client

编程语言: javascript

父类: *Task*

描述说明

使用 `upload` 方法在“打开文件对话框”中选择一个文件，并将其上传到服务端的 *static/files* 目录中。

当在服务端保存文件时，Werkzeug secure_filename 函数会更改该文件的名称，并将当前日期添加到名称中。请参考 http://werkzeug.pocoo.org/docs/0.14/utils/

`options` 参数是一个可以有下列属性的对象：

- `callback` - 是一个当文件被下载时执行的回调函数，指定了该参数，文件的名称被存储在服务端，它将存储在服务器上的文件名、下载文件的名称以及保存文件的文件夹的路径作为参数传递。
- `show_progress` - 如果为 true ，且上传的文件很大，将显示进度条。默认值是 true 。
- `accept` - 该属性指定通过文件上传可以提交的文件类型。参考可接受字符串

> **备注**
>
> 请注意，`accept` 属性仅仅是指定用户在浏览器中弹出的文件对话框里能够选择的文件类型。
>
> 服务端会检查所有上传的文件是否符合项目参数 的 **上传文件扩展名**属性。

事件

on_edit_form_close_query

on_edit_form_created(item)

使用范围: client

编程语言: javascript

父类: *Task 类*

描述说明

`on_edit_form_close_query` 事件由实体项的*close_edit_form* 方法触发。

`item` 参数是那个触发事件的实体项。

另请参见

表单窗体

close_edit_form

on_edit_form_created

on_edit_form_created(item)

使用范围: client

编程语言: javascript

父类: *Task 类*

描述说明

当窗体已被创建当未显示时，由实体项的*create_edit_form* 方法触发 `on_edit_form_created` 事件。

`item` 参数是那个触发事件的实体项

如果定义了该事件，会为任务中调用了*create_edit_form* 方法的每个实体项触发此事件。

另请参见

表单窗体

create_edit_form

on_edit_form_keydown

on_edit_form_keydown(item, event)

使用范围: client

编程语言: javascript

父类: *Task 类*

描述说明

当实体项的*edit_form* 中发生键盘的 keydown 事件时，触发 `on_edit_form_keydown` 事件。

`item` 参数是那个触发事件的实体项。

`event` 是 JQuery 事件对象。

另请参见

表单窗体

create_edit_form

on_edit_form_keyup

on_edit_form_keyup(item, event)

使用范围: client

编程语言: javascript

父类: *Task 类*

描述说明

当实体项的*edit_form* 中发生键盘的 keyup 事件时，触发 `on_edit_form_keyup` 事件。

`item` 参数是那个触发事件的实体项。

`event` 是 JQuery 事件对象。

例如 `CTR+Enter`：

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

另请参见

表单窗体

create_edit_form

on_edit_form_shown

on_edit_form_shown(item)

使用范围: client

编程语言: javascript

父类: *Task 类*

描述说明

当表单窗体显示后，实体项的*create_edit_form* 的方法会触发 `on_edit_form_shown` 事件。

`item` 参数是那个触发事件的实体项。

如果定义了该事件，会为任务中调用了*create_edit_form* 方法的每个实体项触发此事件。

另请参见

表单窗体

create_edit_form

on_filter_form_close_query

on_filter_form_close_query(item)

使用范围: client

编程语言: javascript

父类: *Task 类*

描述说明

`on_filter_form_close_query` 事件由实体项的*close_filter_form* 方法触发。

`item` 参数是那个触发事件的实体项。

另请参见

表单窗体

create_filter_form

close_filter_form

on_filter_form_created

on_filter_form_created(item)

使用范围: client

编程语言: javascript

父类: *Task 类*

描述说明

当窗体已被创建当未显示时，由实体项的*create_filter_form* 方法触发 `on_filter_form_created` 事件。

`item` 参数是那个触发事件的实体项

如果定义了该事件，会为任务中调用了*create_filter_form* 方法的每个实体项触发此事件。

另请参见

表单窗体

create_filter_form

on_filter_form_shown

on_filter_form_shown(item)

使用范围: client

编程语言: javascript

父类: *Task 类*

描述说明

当表单窗体显示后，实体项的*create_filter_form* 的方法会触发 `on_filter_form_shown` 事件。

`item` 参数是那个触发事件的实体项。

如果定义了该事件，会为任务中调用了*create_filter_form* 方法的每个实体项触发此事件。

另请参见

表单窗体

create_filter_form

on_page_loaded

on_page_loaded(task)

使用范围: client

编程语言: javascript

父类: *Task 类*

描述说明

在客户端，`on_page_loaded` 事件是第一个被触发的事件。参考工作流

使用该事件来初始化客户端。

`task` 参数时客户端中任务树 的根节点。

另请参见

工作流

Task 树

on_param_form_close_query

on_param_form_close_query(item)

使用范围: client

编程语言: javascript

父类: *Task* 类

描述说明

`on_param_form_close_query` 事件由*close_param_form* 方法触发。

`report` 参数是那个触发该事件的报表。

另请参见

表单窗体

客户端报表编程

close_param_form

on_param_form_created

on_param_form_created(item)

使用范围: client

编程语言: javascript

父类: *Task* 类

描述说明

`on_param_form_created` 由在*print* 方法中经常调用的*create_param_form* 方法触发。

`report` 参数是那个触发该事件的报表。

另请参见

表单窗体

客户端报表编程

print

create_param_form

on_param_form_shown

on_param_form_shown(item)

使用范围: client

编程语言: javascript

父类: *Task 类*

描述说明

`on_param_form_shown` 由在*print* 方法中经常调用的*create_param_form* 方法触发。

`report` 参数是那个触发该事件的报表。

另请参见

表单窗体

客户端报表编程

print

create_param_form

on_view_form_close_query

on_view_form_close_query(item)

使用范围: client

编程语言: javascript

父类: *Task 类*

描述说明

`on_view_form_close_query` 事件由实体项的*close_view_form* 方法触发。

`item` 参数是那个触发事件的实体项。

另请参见

表单窗体

close_view_form

on_view_form_created

on_view_form_created(item)

使用范围: client

编程语言: javascript

父类: *Task 类*

描述说明

当窗体已被创建当未显示时，由实体项的*view* 方法触发 `on_view_form_created` 事件。

`item` 参数是那个触发事件的实体项。

如果定义了该事件，会为任务中调用了*view* 方法的每个实体项触发此事件。

另请参见

表单窗体

view

on_view_form_keydown

on_view_form_keydown(item, event)

使用范围: client

编程语言: javascript

父类: *Task* 类

描述说明

当实体项的*edit_form* 中发生键盘的 keydown 事件时，触发 `on_edit_form_keydown` 事件。

`item` 参数是那个触发事件的实体项。

`event` 是 JQuery 事件对象。

另请参见

表单窗体

view

on_view_form_keyup

on_view_form_keyup(item, event)

使用范围: client

编程语言: javascript

父类: *Task* 类

描述说明

当实体项的*view_form* 中发生键盘的 keyup 事件时，触发 `on_view_form_keyup` 事件。

`item` 参数是那个触发事件的实体项。

`event` 是 JQuery 事件对象。

例如 `CTRL+Ins` 和 `CTRL+Del`:

```
function on_view_form_keyup(item, event) {
    if (event.keyCode === 45 && event.ctrlKey === true){
        if (item.master) {
            item.append_record();
        }
        else {
            item.insert_record();
        }
    }
    else if (event.keyCode === 46 && event.ctrlKey === true){
        // item.delete_record();
        item.alert('Cannot be deleted on Demo!');
    }
}
```

另请参见

表单窗体

view

on_view_form_shown

on_view_form_shown(item)

使用范围: client

编程语言: javascript

父类: *Task 类*

描述说明

当表单窗体显示后，实体项的*view* 的方法会触发 `on_view_form_shown` 事件。

`item` 参数是那个触发事件的实体项。

如果定义了该事件，会为任务中调用了*view* 方法的每个实体项触发此事件。

另请参见

表单窗体

view

7.1.3 Group Group 类

class Group()

使用范围: client

编程语言: javascript

Group 类用于创建任务树 中的组对象。

下面列出了的这个类的属性、方法和事件。

它同样继承了其祖先类*AbstractItem 类* 的属性和方法

事件

on_edit_form_close_query

on_edit_form_close_query(item)

使用范围: client

编程语言: javascript

父类: *Group 类*

描述说明

`on_edit_form_close_query` 事件由实体项的*close_edit_form* 方法触发。

`item` 参数是触发事件的实体项。

另请参见

表单窗体

close_edit_form

on_edit_form_created

on_edit_form_created(item)

使用范围: client

编程语言: javascript

父类: *Group 类*

描述说明

在表单被创建后但没显示出来的时候，实体项的*create_edit_form* 方法会触发 `on_edit_form_created` 事件。

`item` 参数是触发事件的实体项。

如果定义了这个事件，会为分组里的每个调用了*create_edit_form* 方法的实体项触发此事件，

另请参见

表单窗体

create_edit_form

on_edit_form_keydown

on_edit_form_keydown(item, event)

使用范围: client

编程语言: javascript

父类: *Group 类*

描述说明

当按键的 keydown 事件发生在实体项的*edit_form* 表单上时，会触发 `on_edit_form_keydown` 事件。

`item` 参数是触发事件的实体项。

`event` 是 JQuery 事件对象。

另请参见

表单窗体

create_edit_form

on_edit_form_keyup

on_edit_form_keyup(item, event)

使用范围: client

编程语言: javascript

父类: *Group 类*

描述说明

当按键的 keyup 事件发生在实体项的*edit_form* 表单上时，会触发 `on_edit_form_keyup` 事件。

`item` 参数是触发事件的实体项。

`event` 是 JQuery 事件对象。

另请参见

表单窗体

create_edit_form

on_edit_form_shown

on_edit_form_shown(item)

使用范围: client

编程语言: javascript

父类: *Group 类*

描述说明

在表单显示后，`on_edit_form_shown` 事件由实体项的*create_edit_form* 方法触发。

`item` 参数是触发事件的实体项。

如果定义了这个事件，会为分组里的每个调用了*create_edit_form* 方法的实体项触发此事件，

另请参见

表单窗体

create_edit_form

on_filter_form_close_query

on_filter_form_close_query(item)

使用范围: client

编程语言: javascript

父类: *Group 类*

描述说明

`on_filter_form_close_query` 事件由实体项的:doc:'close_filter_form </refs/client/item/m_close_filter_form>'方法触发。

`item` 参数是触发事件的实体项。

另请参见

表单窗体

close_filter_form

on_filter_form_created

on_filter_form_created(item)

使用范围: client

编程语言: javascript

父类: *Group 类*

描述说明

在表单被创建后但没显示出来的时候，实体项的*create_filter_form* 方法会触发 `on_filter_form_created` 事件。

`item` 参数是触发事件的实体项。

如果定义了这个事件，会为分组里的每个调用了*create_filter_form* 方法的实体项触发此事件，

另请参见

表单窗体

create_filter_form

on_filter_form_shown

on_filter_form_shown(item)

使用范围: client

编程语言: javascript

父类: *Group 类*

描述说明

The `on_filter_form_shown` event is triggered by the *create_filter_form* method of the item when the form has been shown. 在表单显示后，`on_filter_form_shown` 事件由实体项的*create_filter_form* 方法触发。

`item` 参数是触发事件的实体项。

如果定义了这个事件，会为分组里的每个调用了*create_filter_form* 方法的实体项触发此事件，

另请参见

表单窗体

create_filter_form

on_view_form_close_query

on_view_form_close_query(item)

使用范围: client

编程语言: javascript

父类: *Group 类*

描述说明

`on_view_form_close_query` 事件由实体项的*close_view_form* 方法触发。

`item` 参数是触发事件的实体项。

另请参见

表单窗体

close_view_form

on_view_form_created

on_view_form_created(item)

使用范围: client

编程语言: javascript

父类: *Group 类*

描述说明

在表单被创建后但没显示出来的时候，实体项的*view* 方法会触发 `on_view_form_created` 事件。

`item` 参数是触发事件的实体项。

如果定义了这个事件，会为分组里的每个调用了*view* 方法的实体项触发此事件，

另请参见

表单窗体

view

on_view_form_keydown

on_view_form_keydown(item, event)

使用范围: client

编程语言: javascript

父类: *Group 类*

描述说明

当按键的 keydown 事件发生在实体项的*view_form* 表单上时，会触发 `on_view_form_keydown` 事件。

`item` 参数是触发事件的实体项。

`event` 是 JQuery 事件对象。

另请参见

表单窗体

view

on_view_form_keyup

on_view_form_keyup(item, event)

使用范围: client

编程语言: javascript

父类: *Group 类*

描述说明

当按键的 keyup 事件发生在实体项的*view_form* 表单上时，会触发 `on_view_form_keyup` 事件。

`item` 参数是触发事件的实体项。

`event` 是 JQuery 事件对象。

另请参见

表单窗体

view

on_view_form_shown

on_view_form_shown(item)

使用范围: client

编程语言: javascript

父类: *Group 类*

描述说明

在表单显示后，`on_view_form_shown` 事件由实体项的*view* 方法触发。

`item` 参数是触发事件的实体项。

如果定义了这个事件，会为分组里的每个调用了*view* 方法的实体项触发此事件，

另请参见

表单窗体

view

7.1.4 Item class

class Item()

使用范围: client

编程语言: javascript

Item 类用于创建任务树 中的实体项对象，这些对象可能有一个关联的数据库表。

下面列出了的这个类的属性、方法和事件。

它同样继承了其祖先类*AbstractItem 类* 的属性和方法

属性与特性

active

active

使用范围: client

编程语言: javascript

父类: *Item 类*

描述说明

指明实体项的数据集是否已经被打开。

使用 `active` 只读属性来确定实体项的数据集是否已经打开。

open 方法将 `active` 的值更改为 `true` ，*close* 方法将 `active` 的值更改为 `false` ，

当数据集打开时，可以浏览其记录，修改其数据，并将更改保存在实体项的数据库表中。

另请参见

数据集

导航数据集

修改数据集

can_modify

active

使用范围: client

编程语言: javascript

父类: *Item 类*

描述说明

把 can_modify 属性设置为 false ， 可以禁止在可视化控件中修改实体项。

当 can_modify 是 true 时，*can_create* 、*can_edit* 和*can_delete* 方法返回 false。

can_modify 属性的默认值是 true 。

details

details

使用范围: client

编程语言: javascript

父类: *Item 类*

描述说明

列举出实体项的全部明细 对象。

另请参见

明细

each_detail

edit_form

edit_form

使用范围: client

编程语言: javascript

父类: *Item 类*

描述说明

使用 edit_form 属性来访问对一个呈现实体项的编辑表单的 JQuery 对象。

它由*create_edit_form* 方法创建。

close_edit_form 方法将 edit_form 的值设置为 undefined。

示例

在以下示例中，实体项编辑表单的 html 模板中定义的按钮被分配了一个点击事件：

```
item.edit_form.find("#ok-btn").on('click.task',
    function() {
        item.apply_record();
    }
);
```

另请参见

表单窗体

create_edit_form

close_edit_form

edit_options

edit_options

使用范围: client

编程语言: javascript

父类: *Item 类*

描述说明

`edit_options` 属性是一些选项的集合，这些选项决定着编辑表单将在浏览器页面中如何显示。

这些选项在应用程序构建器里的*标记表达对话框* 里面进行设置。

你可以在实体项的*on_edit_form_created* 事件处理程序中修改 `edit_options` 。

`edit_options` 是一个有下列属性的对象：

选项	描述
width	模态表单窗体的宽度，默认值是 600 px 。
title	窗体的标题，其默认值是*item_caption* 属性的值。
form_border	如果为 true ，窗体周围将显示边框。
form_header	如果为 true ，将创建并显示窗体的包含标题和各种按钮的标题栏。
history_button	如果为 true ，并且*启用了保存变更历史* ，历史按钮将显示在窗体的标题栏中。
close_button	如果为 true ，关闭按钮将显示在窗体的右上角。
close_on_esc	如果为 true ，按下 Escape 键，将执行*close_edit_form* 方法以关闭窗体。
edit_details	明细名称的列表，如果编辑表单模板包含类为“edit-detail”的 div （默认编辑表单模板具有此 div），则可以在编辑表单中对其进行编辑。
detail_height	显示在视图表单中的明细内容的高度，如果不指定，明细表格的高度则是 200 px。
fields	如果未指定其 options 参数的 fields 属性，则指定*create_inputs* 方法将使用的字段名列表
template_class	如果指定，则将在任务*templates* 属性中搜索具有此类的 div，并在创建表单时用作表单 html 模板。必须在创建表单之前设置此属性
modeless	如果设置，则编辑表单将以非模态方式创建，否则将以模态方式创建

示例

```
function on_edit_form_created(item) {
    item.edit_options.width = 800;
    item.edit_options.close_on_escape = false;
}
```

另请参见

表单窗体

create_edit_form

close_edit_form

fields

fields

使用范围: client

编程语言: javascript

父类: *Item 类*

描述说明

列举出了实体项的所有的字段 对象。

示例

```
function customer_fields(customers) {
    customers.open({limit: 1});
    for (var i = 0; i < customers.fields.length; i++) {
        console.log(customers.fields[i].field_caption, customers.fields[i].display_text);
    }
}
```

另请参见

字段

Field 类

each_field

filter_form

filter_form

使用范围: client

编程语言: javascript

父类: *Item 类*

描述说明

Use `filter_form` attribute to get access to a Jquery object representing the filter form of the item. 使用 `filter_form` 属性获取对一个表示为实体项的过滤器窗体的 JQuery 对象的访问权限。

它由*create_filter_form* 方法创建。

close_filter_form 方法将 `filter_form` 的值设置为 undefined 。

示例

In the following example the button defined in the item filter html template is assigned a click event:

在以下示例中，实体项过滤器的 html 模板中定义的按钮被分配了一个点击事件：

```
item.filter_form.find("#cancel-btn").on('click',
    function() {
        item.close_filter()
    }
);
```

另请参见

表单窗体

create_filter_form

close_filter_form

filter_options

filter_options

使用范围: client

编程语言: javascript

父类: *Item 类*

描述说明

使用 `filter_options` 属性指定过滤器模态表单窗体的参数。

`filter_options` 是一个有下列属性对象：

- `width` - 模态表单窗体的宽度，默认值是 560 px 。
- `title` - 使用这个属性来获取或设置过滤器表单窗体的标题。
- `close_button` - 如果为 true ，关闭按钮将显示在窗体的右上角，默认为 true。
- `close_caption` - 如果为 true ，同时 close_button 也为 true ，在按钮旁显示 "Close - [Esc]"
- `close_on_escape` 如果为 true ，按下 Escape 键，将触发*close_filter_form* 方法。
- `close_focusout` - 如果为 true ，当表单窗体失去焦点时，会调用*close_filter_form* 方法。
- `template_class` - 如果指定，则将在任务*templates* 属性中搜索具有此类的 div，并在创建表单时用作表单 html 模板。

示例

```
function on_filter_form_created(item) {
    item.filter_options.width = 700;
}
```

另请参见

表单窗体

create_filter_form

close_filter_form

Filtered

filtered

使用范围: client

编程语言: javascript

父类: *Item 类*

描述说明

指定一个数据集是否启用了过滤（筛选）。

检查 filtered 来确定本地的数据集筛选是否有效。如果 filtered 为 true ，那么筛选是激活的。要应用*on_filter_record* 事件处理程序中指定的筛选条件，请将 filtered 设置为 true 。

另请参见

on_filter_record

filters

filters

使用范围: client

编程语言: javascript

父类: *Item 类*

描述说明

列举出了实体项的所有的过滤器 。

示例

```
function invoices_filters(invoices) {
    for (var i = 0; i < invoices.filters.length; i++) {
        console.log(invoices.filters[i].filter_caption, invoices.filters[i].value);
    }
}
```

另请参见

过滤器

Filter 类

each_filter

item_state

item_state

使用范围: client

编程语言: javascript

父类: 实体项

描述说明

检查 `item_state` 以确定实体项的当前操作模式。Item_state 决定了可以对一个实体项中的数据集进行哪些操作，例如编辑已存在额记录，或者插入新记录。`item_state` 在应用程序处理数据时不断变化。Opening a item changes state from inactive to browse. 打开一个实体项会将数据集的状态由不活动状态变为浏览状态。应用程序可以调用*edit* 将实体项置于编辑状态，或者调用*insert* 或*append* 将实体项置于插入状态。

提交或取消编辑、插入或删除，将使 `item_state` 从实体项的当前状态变为浏览状态。关闭一个数据集，将使其处于不活动状态。

要检查 item_state 的值，请使用下面的方法：

- *is_new* - 指明实体项是否处于插入状态
- *is_edited* - 指明实体项是否处于编辑状态
- *is_changing* - 指明实体项是否处于编辑或插入状态

item_state 的值，可以是：

- 0 - 不活动状态
- 1 - 浏览状态
- 2 - 插入状态
- 3 - 编辑状态
- 4 - 删除状态

item *task* attribute have consts object that defines following attributes: 实体项的*task* 属性有一个常量对象，它有下列属性：

- "STATE_INACTIVE": 0,
- "STATE_BROWSE": 1,
- "STATE_INSERT": 2,
- "STATE_EDIT": 3,
- "STATE_DELETE": 4

因此，可以使用下列方法检查实体项是否处于插入状态：

```
item.item_state === 2
```

or:

```
item.item_state === item.task.consts.STATE_INSERT
```

or:

```
item.is_new()
```

另请参见

修改数据集

log_changes

log_changes

使用范围: client

编程语言: javascript

父类: *Item* 类

描述说明

指明是否记录数据变更的日志。

使用 `log_changes` 来控制对一个实体项数据集中的数据做作的更改是否进行记录。当 `log_changes` 为 `true`（这是默认值），所由更改都被记录下来。以后，可以通过调用*apply* 方法，在应用程序服务器上使用它们。当 `log_changes` 为 *false*‘，所做的更改不被记录，也不可以在应用程序服务器上使用它们。

另请参见

修改数据集

apply

lookup_field

lookup_field

使用范围: client

编程语言: javascript

父类: *Item* 类

描述说明

使用 `lookup_field` 检查是否创建了实体项来为查找字段选择值。请参阅：查找字段

示例

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

(续下页)

(接上页)

```
        print_btn.click(function() { print(item) });
        item.table_options.multiselect = true;
    }
}
```

paginate

paginate

使用范围: client

编程语言: javascript

父类: *Item 类*

描述说明

`paginate` 属性决定了由*create_table* 方法创建的表格的分页行为。

当设置 `paginate` 为 `true` 时，将创建一个分页器，表格将基于自身的高度计算每页显示的行数。

该表将根据自身的高度和当前显示的页面，在内部操纵*open* 方法的 `limit` 和 `offset` 参数，在页面更改时重新打开数据集。

如果 `paginate` 为 `false` 时，表格将显示出数据集中所有可用的记录。

另请参见

create_table

open

permissions

permissions

使用范围: client

编程语言: javascript

父类: *Item 类*

描述说明

设置 `permissions` 属性以禁止更改可视化控件中的实体项。

`permissions` 属性是一个对象，它有下列属性:

- can_create
- can_edit
- can_delete

By default theses attributes are set to true. 这些属性的默认值是 true。

当这些属性被设置为 false 时，下来相应的方法返回 false：

- *can_create*,
- *can_edit*,

- *can_delete*

另请参见

如何禁止修改记录

read_only

read_only

使用范围: client

编程语言: javascript

父类: *Item 类*

描述说明

`read_only` 属性决定了是否可以修改数据感知控件中的数据。

Set `read_only` property to `true` to prevent data from being modified in data-aware controls. 设置 `read_only` 属性为 `true` 以保护数据在数据感知控件中不被修改。

当为 read_only 属性赋值时，应用程序会将所有明细的 read_only 属性和字段的*read_only* 属性的都设置为相同的值。

如果用户角色禁止编辑记录，那么 `read_only` 返回 `true` 。

另请参见

read_only

示例

在这个例子中，我们首先将发票实体项的 `read_only` 属性设置为 `true` 。这将使所有字段和 invoice_table 明细都是只读的。然后，我们运行一个用户编辑 customer 字段和 invoice_table 明细表。

```
function on_edit_form_created(item) {
    item.read_only = true;
    item.customer.read_only = false;
    item.invoice_table.read_only = false;
}
```

rec_count

rec_count

使用范围: client

编程语言: javascript

父类: *Item 类*

描述说明

读取 `rec_count` 属性以获取实体项数据集所拥有的记录的数量。

如果在模块中声明了一个*on_filter_record* 事件处理程序，并且设置了*Filtered* 属性，`rec_count` 属性计算经过过滤后的记录的数量，否则使用*record_count* 方法计算记录的数量。

另请参见

record_count

示例

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

rec_no

rec_no

使用范围: client

编程语言: javascript

父类: *Item 类*

描述说明

检查 rec_no 属性以确定实体项数据集中当前记录的记录号。rec_no 可以设置为特定的记录号，以将光标定位在该记录上。

示例

```
function calculate(item) {
    var subtotal,
        tax,
        total,
        rec;
    if (!item.calculating) {
        item.calculating = true;
        try {
            subtotal = 0;
            tax = 0;
            total = 0;
            item.invoice_table.disable_controls();
            rec = item.invoice_table.rec_no;
            try {
                item.invoice_table.each(function(d) {
                    subtotal += d.amount.value;
                    tax += d.tax.value;
                    total += d.total.value;
                });
            }
            finally {
```

(续下页)

(接上页)

```
                item.invoice_table.rec_no = rec;
                item.invoice_table.enable_controls();
            }
            item.subtotal.value = subtotal;
            item.tax.value = tax;
            item.total.value = total;
        }
        finally {
            item.calculating = false;
        }
    }
}
```

另请参见

数据集

导航数据集

selections

selections

使用范围: client

编程语言: javascript

父类: *Item 类*

描述说明

`selections` 属性存储主键字段值的列表。

当在视图表单对话框 的 **布局**选项卡上选中 **多选**复选框，或者以编程方式设置*table_options* 的多选属性时，表格的最左侧列就会出现复选框，每次用户单击复选框时，`selections` 属性都会发生变化。

也可以通过使用 `add` 或 `remove` 方法或分配数组以编程方式对其进行更改。

示例

在这个例子中，客户端上的 `send_email` 函数使用 **Customers** 的 selections 属性获取用户选择的主键字段值数组，并使用*server* 方法将它们发送到实体项服务端模块中定义的"send_email"功能

```
function send_email(subject, message) {
    var selected = task.customers.selections;
    if (!selected.length) {
        selected.add(task.customers.id.value);
    }

    item.server('send_email', [selected, subject, message],
        function(result, err) {
            if (err) {
                item.alert('Failed to send the mail: ' + err);
            }
            else {
                item.alert('Successfully sent the mail');
```

(续下页)

(接上页)

```
            }
        }
    );
}
```

在服务端，使用*open* 方法来检索已选择的客户的相关信息。

```
import smtplib

def send_email(item, selected, subject, mess):
    cust = item.task.customers.copy()
    cust.set_where(id__in=selected)
    cust.open()
    to = []
    for c in cust:
        to.append(c.email.value)

    # code that sends email
```

table_options

table_options

使用范围: client

编程语言: javascript

父类: *Item 类*

描述说明

`table_options` 属性一个选项的集合，它们决定了视图表单中的表格将如何显示。如果*create_table* 的可选参数没有覆盖相关选项，那么它会使用定义在 `table_options` 里面的选项。

在应用程序构建器中，在视图表单对话框 的 **布局 (Layout)** 选显卡中设置这些选项。

你可以在实体项的*on_view_form_created* 事件处理程序中更改 `table_options` ，请见示例。

`table_options` 参数是一个有下列属性的对象：

选项	描述
row_count	指定表格显示的行数。
height	若未指定 row_count，则由该参数决定表格高度，默认值为 480。表格创建时会根据此参数值计算显示的行数（row_count）。
fields	字段名称的列表。若指定该参数，会为列表中的每个字段创建一列；若未指定（默认情况），则使用 *view_options* 的 fields 属性。
ti-tle_line_cou	指定标题行显示的文本行数。若值为 0，行高由标题单元格的内容决定。
row_line_cc	指定表格行显示的文本行数。若值为 0，行高由单元格的内容决定。
ex-pand_select	若已设置 row_line_count 且 expand_selected_row 大于 0，则指定表格选中行显示的最小文本行数。
ti-tle_word_w	指定列标题文本是否允许自动换行。
col-umn_width	列宽默认由浏览器自动计算。可使用该选项强制指定列宽。该选项为一个对象，其键名为字段名，值为 CSS 单位格式的列宽。
ed-itable_fields	可在表格中编辑的字段名称的列表。
se-lected_field	若设置了 editable_fields，指定当选中行发生变化时，会自动选中其对应列的字段名。
sortable	若指定该选项，可通过点击表格列标题对数据记录进行排序。若未指定 sort_fields 选项（默认），用户可按任意字段排序；否则，仅可按该选项中列出的字段排序。
sort_fields	可通过点击对应列标题进行排序的字段名称列表。若实体项为明细项，排序操作在客户端执行；否则在服务端执行（内部使用 *open* 方法）
sum-mary_fields	字段名称列表。指定后，表格会为数值型字段计算合计值并显示在表尾，非数值型字段则显示记录总数
freeze_cour	一个整数值。若大于 0，指定前几列固定显示——表格水平滚动时，这些列不会跟随滚动。
show_hints	若为 true，当鼠标悬停在文本内容超出单元格尺寸的表格单元格上时，会显示提示框。默认值为 true。
hint_fields	字段名称列表。若指定该参数，无论 show_hints 的值如何，仅会为列表中的字段显示提示框
on_click	指定用户点击表格行时执行的函数。函数会将当前数据项作为参数传入。
on_dblclick	指定用户双击表格行时执行的函数。函数会将当前数据项作为参数传入。
dblclick_ed	若该选项值为 true 且未设置 on_dblclick，用户双击表格行时会显示编辑表单。
multise-lect	若设置该选项，会在最左侧生成一列复选框用于选择记录。用户点击复选框时，记录的主键字段值会添加至 *selections* 属性，或从中移除。
select_all	若为 true，表格表头最左侧会显示一个菜单，允许用户选中所有匹配当前筛选和搜索条件的记录
row_callbac	记录字段每次发生变化时调用的回调函数。函数接收两个参数：发生变化的数据项、对应表格行的 jQuery 对象。注意：传入函数的数据项可能不是原数据项本身，而是共享同一数据集的克隆对象。

示例

```
function on_view_form_created(item) {
    item.table_options.row_line_count = 2;
    item.table_options.expand_selected_row = 3;
}
```

下来两个示例的代码作用一样：

```
item.invoice_table.create_table(item.view_form.find('.view-detail'), {
    height: 200,
    summary_fields: ['date', 'total'],
```

(续下页)

(接上页)

```
});
```

```
item.invoice_table.table_options.height = 200;
item.invoice_table.table_options.summary_fields = ['date', 'total'];
item.invoice_table.create_table(item.view_form.find('.view-detail'));
```

另请参见

视图表单对话框

on_view_form_created

create_table

view_form

view_form

使用范围: client

编程语言: javascript

父类: *Item* 类

描述说明

使用 `view_form` 属性访问表示实体项视图表单的 Jquery 对象。

它由*view* 方法创建。

close_view_form 方法会将 `view_form` 的值设置为 undefined 。

示例

在下面的示例中，在实体项的 html 模板中定义的按钮上分配了一个 click 事件：

```
item.view_form.find("#new-btn").on('click',
    function() {
        item.insert_record();
    }
);
```

另请参见

表单窗体

view

create_view_form

close_view_form

view_options

view_options

使用范围: client

编程语言: javascript

父类: *Item 类*

描述说明

`view_options` 属性是一个选项的集合，它们决定视图表单在浏览器页面上将如何显示。

在应用程序构建器的视图表单对话框 里面对这些选项进行设置。

你可以在实体项的*on_view_form_created* 事件处理程序历更改 `view_options` ，请见示例。

`view_options` 是一个有下列属性的对象：

Option	Description
width	模态表单窗体的宽度，默认值是 600 px 。
title	表单窗体的标题，其默认值是*item_caption* 属性的值。
form_border	如果为 true，表单窗体周围会显示边框。
form_header	如果为 true，将创建包含表单窗体的标题和各种按钮的标题栏。
history_button	如果为 true 并且启用保存变更历史 ，那么将在表单标题栏内显示“保存变更历史按钮”。
refresh_button	如果为 true，那么将在表单标题栏内创建“刷新按钮”，这允许用户向服务端发送请求来刷新页面。
enable_search	if true, the search input will be created in the form header 如果为 true，那么将在表单标题栏内创建“搜索输入框”。
search_field	默认的搜索字段的名称
enable_filters	如果为 true 并且有可用的可见的过滤器，那么将在表单标题栏内创建“搜过滤器按钮”。
close_button	如果为 true，那么将在表单右上角创建“关闭按钮”。
close_on_esc	如果为 true，按下 Escape 键会执行*close_view_form* 来关闭表单窗体。
view_details	明细项名称的列表，如果视图表单模板包含样式类为'view-detail' 的 div （默认的视图表单模板已包含），它们将显示在视图表单中，
detail_height	在视图表单中显示的明细表的高度，如果没有指定，那么明细表的高度是 200px。
modeless	如果为 true，表单窗体将以非模态显示。
template_class	如果指定，将在“任务”的*templates* 属性中搜索具有此类的 div ，并在创建表单时用作表单 html 模板。必须在创建表单之前设置此属性

示例

```javascript
function on_view_form_created(item) {
    item.view_options.width = 800;
    item.view_options.close_button = false;
    item.view_options.close_on_escape = false;
}
```

另请参见

表单窗体

view

virtual_table

virtual_table

使用范围: client

编程语言: javascript

父类: *Item 类*

描述说明

使用只读的 `virtual_table` 属性找出对应的实体项在项目数据库中是否有相应的数据表。

如果 `virtual_table` 属性是 `True`，那么在项目数据库中与之对应的数据表。你可以使用这样的实体项来处理内存中的数据集，也可以使用其模块来编写代码。调用*open* 方法会创建一个空的数据集，而调用*apply* 方法什么也不会发生。

方法

add_edit_button

add_edit_button(*text*, *options*)

使用范围: client

编程语言: javascript

描述说明

使用 `add_edit_button`，可以在编辑表单窗体中动态地添加一个按钮。

这个方法与*add_view_button* 方法有相同地参数。

add_view_button

add_view_button(*text*, *options*)

使用范围: client

编程语言: javascript

描述说明

使用 `add_view_button`，可以在视图表单窗体上动态地添加一个按钮。

在 `on_view_form_created` 事件中经常使用这个方法。

下列参数被传给这个方法:

- `text` - 将在按钮上显示的文本内容。
- `options` - 指定按钮额外属性的选项。

`options` 参数是一个对象，他又下列属性:

- `parent_class_name` 是其父元素的一个样式类的名称，默认值是 “form-footer”。
- `btn_id` - 按钮的 ID 属性。
- `btn_class` - 按钮的样式类。
- `type` - 指定按钮的类型（颜色），其值在下列文本值中：
  - primary
  - success
  - info
  - warning
  - danger
- `image` - 一个图标样式类, 取值来自 Glyphicons 的图标样式库 http://getbootstrap.com/2.3.2/base-css.html
- `secondary`: 如果设置这个属性为 true ，那么在设置了视图表单窗体 的 **按钮置顶 (Buttons on top)** 属性时按钮将右对齐显示，否则将左对齐显示。
- `expanded` - 如果设置为 true ， 按钮将具有类 “expanded btn”， 并将其最小宽度定义为 120px，默认为 true .

该方法返回按钮的一个 JQuery 对象。

Examples

```
function on_view_form_created(item) {
    var btn = item.add_view_button('Select', {type: 'primary'});
    btn.click(function() {
        item.select_records('track');
    });
}

function on_view_form_created(item) {
    if (!item.view_form.hasClass('modal')) {
        var print_btn = item.add_view_button('Print', {image: 'icon-print'}),
            email_btn = item.add_view_button('Send email', {image: 'icon-pencil'});
        email_btn.click(function() { send_email() });
        print_btn.click(function() { print(item) });
    }
}
```

append

append()

使用范围: client

编程语言: javascript

父类: *Item 类*

描述说明

在数据集末尾，打开一个新的空记录。

在调用 append 之后，应用程序能使用户在记录的字段中输入数据，然后使用*post* 方法将用户的输入提交到实体项的数据集中，最后，使用*apply* 方法，将修改的内容保存到实体项在数据库中对应的数据表中。

append 方法

- 检查实体项的数据集是否处于*active* 状态，如果不是，会抛出异常
- 如果实体项是一个明细项 *(detail)*，那么会检查主实体是否处于“编辑”或“插入”*active* 状态，如果不是，会抛出异常
- 如果实体项不是一个明细项 *(detail)*，那么会检查它是否处于“浏览”*active* 状态，如果不是，会抛出异常
- 如果为实体项定义了*on_before_append* 事件处理程序，那么会触发它。
- 在数据集末尾，打开一个新的空记录。
- 将实体项置于“插入”状态 。
- 如果为实体项定义了*on_after_append* 事件处理程序，那么会触发它。
- 更新数据感知控件

另请参见

修改数据集

append_record

append_record(*container*)

使用范围: client

编程语言: javascript

父类: *Item* 类

描述说明

在数据集末尾，打开一个新的空记录，并创建一个*edit_form* 用于记录的可视化编辑。

如果指定了 container 参数（DOM 元素的 JQuery 对象），那么编辑表单的 html 模板会被添加到容器（container）里面。

如果未指定 container 参数，但在编辑表单对话框中设置了 **无模式表单 (Modeless form)** 属性，或者以编程方式设置了*edit_options* 的无模式属性，并且设置了任务的*forms_in_tabs* 属性，而应用程序没有模式表单，则将在任务的*forms_container* 对象的新选项卡中创建无模式编辑表单。

在其它情况下，将创建模态表单窗体。

如果在非模态模式下允许添加记录，应用程序会调用*copy* 方法创建实体项的一个副本。这个副本用来添加新记录。

append_record 方法：

- 调用*can_create* 方法检查一个用户是否有权限添加记录，如果没有权限，则
- 检查实体项是否处于编辑或插入状态 ，如果不是，则调用*append* 方法添加一个新记录，
- 调用*create_edit_form* 方法创建一个用于记录的可视化编辑的表单窗体。

另请参见

修改数据集

append

can_create

apply

apply(*callback*, *params*, *async*)

使用范围: client

编程语言: javascript

父类: *Item 类*

描述说明

将所有已更新的、已插入的、已删除的记录从实体项数据集发送到应用程序服务端，以写入数据库。

`apply` 方法有如下参数:

- `callback`: 如果参数不存在，并且 `async` 参数是 `false` 或是 `undefined`，发送到服务端的请求是同步的；否则，将异步执行请求，在收到响应后，将执行回调。
- `params` - 一个指定用户定义参数的对象，可以在服务器上的*on_apply* 事件处理程序中使用，以进行一些额外的处理
- `async`: 如果值是 true ，并且没有回调参数，将异步执行请求

参数的先后顺序无关紧要。

`apply` 方法

- 检查实体项是否是一个明细项，如果它是，返回 and if it is, returns (the master saves the details changes)
- 检查实体项是否处于"编辑"或"插入"状体 ，如果是，则提交记录
- 检查变更记录是否有变化，如果没有，则执行回调函数（如果传递了），然后返回
- 如果为实体项定义了*on_before_apply* 事件处理程序，则触发
- 将变更发送到服务端。
- 服务器在接收到求情后，检查是否为实体项定义了*on_apply* 事件处理程序，如果定义了，就执行它。否则，生成 SQL 查询语句并执行，已将变更写入数据库。请参考*on_apply events* 话题。
- 在生成一个 SQL 查询语句时，会检查发送请求的用户是否有权限进行更改，如果没有，则抛出一个异常。
- 将变更写入数据库。
- 在将变更写入数据库之后，服务器将执行的结果发送到客户端。
- 如果在服务端上的操作引发了异常，那么在抛出异常之前，客户端也会抛出异常。如果传了回调参数，将调用它，将错误信息作为回调函数的参数进行传递。
- 客户端依据执行结果更新变更日志。
- 如果为实体项定义了*on_after_apply* 事件处理程序，则触发它。
- 如果传递了回调参数，则调用它。

> **备注**
>
> 服务端在将新记录写入数据库表之前，会为主字段生成值。客户端根据从服务端接收到的信息更新这些字段。如果你在*on_apply* 事件处理程序中更改了其他字段的值，这些更改不会反映在客户端上。您可以自己使用像:doc:'refresh_record <m_refresh_record>'这样的方法更新它们。

示例

```
var self = this;
this.apply(function(err) {
    if (err) {
        self.alert_error(err);
    }
    else {
        //some code to execute after appling changes
    }
});
```

另请参见

修改数据集

apply_record

apply_record()

使用范围: client

编程语言: javascript

父类: *Item* 类

描述说明

将更改写入应用程序的数据集。

`apply_record` 方法

- 调用*apply* 方法将更改写入数据集。
- 调用*close_edit_form* 方法销毁 edit_form

另请参见

修改数据集

close_edit_form

apply

assign_filters

assign_filters(*item*)

使用范围: client

编程语言: javascript

父类: *Item* 类

描述说明

使用 `assign_filters` 将 `item` 参数的过滤器的值设置为实体项的过滤器的值。

示例

```
function calc_footer(item) {
    var copy = item.copy({handlers: false, details: false});
    copy.assign_filters(item);
    copy.open(
        {fields: ['subtotal', 'tax', 'total'],
        funcs: {subtotal: 'sum', tax: 'sum', total: 'sum'}},
        function() {
            var footer = item.view_form.find('.dbtable.' + item.item_name + ' tfoot');
            copy.each_field(function(f) {
                footer.find('div.' + f.field_name)
                    .css('text-align', 'right')
                    .css('color', 'black')
                    .text(f.display_text);
            });
        }
    );
}
```

另请参见

过滤记录

过滤器

bof

bof()

使用范围: client

编程语言: javascript

父类: *Item* 类

描述说明

调用 bof (beginning of file) 方法，以确定数据库的游标是否在一个实体项数据集的第一条记录上。

bof 返回 true 时，游标就明确地位于数据集中的第一行。

bof 返回 true ，当应用程序：

- 打开一个实体项数据集；
- 调用一个实体项的*first* 方法；
- 调用一个实体项的*prior* 方法, 且方法执行失败 (因为游标已经就在数据集的第一行)。

在其它所有情况下，bof 返回 false。

> **备注**
>
> 如果*eof* 和 bof 都返回 true ，实体项数据集是空的。

另请参见

数据集

导航数据集

calc_summary

calc_summary(*detail*, *fields*)

使用范围: client

编程语言: javascript

父类: *Item* 类

描述说明

使用 calc_summary 方法计算明细项的字段的总和，并在 on_detail_changed 事件处理程序中将这些值保存明细项的主表的字段中。

`detail` 参数是将对其求和的字段的明细。

`fields` 参数是一个对象，它定义了主表字段和明细项字段之间的对应。对象的键是主表字段，值是对应的明细项的字段。如果明细项字段是一个数值类型的字段，将对其求和，否则，计算的结果是记录的数量。这个对象的值可以是一个函数，它返回对明细项的记录的计算结果。

示例

```
function on_detail_changed(item, detail) {
    var fields = [
        {"total": "total"},
        {"tax": "tax"},
        {"subtotal": function(d) {return d.quantity.value * d.unitprice.value}}
    ];
    item.calc_summary(detail, fields);
}
```

另请参见

on_detail_changed 明细

can_create

can_create()

使用范围: client

编程语言: javascript

父类: 实体项

描述说明

使用 `can_create` 方法确定一个用户是否有权限创建一个新记录。

当设置了项目安全模式参数 以及*permissions* 属性和*can_modify* 属性的值时，此方法考虑了在应用程序生成器的角色节点 中设置的用户权限。

示例

```
if (item.can_create()) {
    item.view_form.find("#new-btn").on('click',
        function() {
            item.append_record();
        }
    );
}
else {
    item.view_form.find("#new-btn").prop("disabled", true);
}
```

另请参见

参数

can_delete

can_delete()

使用范围: client

编程语言: javascript

父类: 实体项

描述说明

使用 `can_delete` 方法确定一个用户是否有权限删除一个属于实体项数据集的记录。

当设置了项目安全模式参数 以及*permissions* 属性和*can_modify* 属性的值时，此方法考虑了在应用程序生成器的角色节点 中设置的用户权限。

示例

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

can_edit

can_edit()

使用范围: client

编程语言: javascript

父类: 实体项

描述说明

使用 can_edit 方法确定一个用户是否有权限编辑一个实体项数据集中的记录。

当设置了项目安全模式参数 以及*permissions* 属性和*can_modify* 属性的值时，此方法考虑了在应用程序生成器的角色节点 中设置的用户权限。

示例

```
if (item.can_edit()) {
    item.view_form.find("#edit-btn").on('click',
        function() {
            item.edit_record();
        }
    );
}
else {
    item.view_form.find("#edit-btn").prop("disabled", true);
}
```

cancel

cancel()

使用范围: client

编程语言: javascript

父类: *Item 类*

描述说明

调用 cancel 方法以撤销对当前记录一个或多个字段所作的修改，只要变更还没有被提交到实体项数据集。

Cancel

- 如果为实体项定义了*on_before_cancel* 事件处理程序，就触发它
- 如果当前已有的记录被编辑，则撤销当前对记录和与之关联的明细项的修改；如果新记录被添加或插入，则移除新记录。。
- 将实体项置于“浏览”状态
- 如果为实体项定义了*on_after_cancel* 事件处理程序，就触发它
- 更新数据感知控件

另请参见

修改数据集

cancel_edit

cancel_edit()

使用范围: client

编程语言: javascript

父类: *Item 类*

描述说明

取消记录的可视化编辑

`cancel_edit` 方法

- 调用*close_edit_form* 方法以销毁 edit_form
- 调用*cancel* 方法以撤销对记录所做的修改。

另请参见

修改数据集

close_edit_form

cancel

clear_filters

clear_filters()

使用范围: client

编程语言: javascript

父类: *Item 类*

描述说明

使用 `clear_filters` 方法以将实体项的过滤器的值设置为 `null` 。

另请参见

过滤记录

过滤器

clone

clone(*keep_filtered*)

使用范围: client

编程语言: javascript

父类: *Item 类*

描述说明

使用 `clone` 方法来创建一个与其共享数据集的实体项的副本。

这个副本有它自己的游标，因此你可以导航游标，而实体项的有标位置不会受影响。

如果你希望副本有与实体项一样的本地过滤器，请将 `keep_filtered` 参数设置为 true 。

示例

```
function calc_sum(item) {
    var clone = item.clone(),
        result = 0;
    clone.each(function(c) {
        result += c.sum.value;
    })
    return result;
}
```

另请参见

on_filter_record

close

close()

使用范围: client

编程语言: javascript

父类: *Item 类*

描述说明

调用 `close` 方法关闭实体项的数据集。在数据集被关闭后，*active* 属性是 `false`。

另请参见

数据集

open

close_edit_form

close_edit_form()

使用范围: client

编程语言: javascript

父类: *Item 类*

描述说明

使用 `close_edit_form` 方法关闭实体项的编辑表单窗体。

如果定义了*on_edit_form_close_query* 事件处理程序，那么 `close_edit_form` 方法会触发它。如果定义了事件处理程序，并且

- 返回 `true` - 表单窗体被销毁，实体项的*edit_form* 属性被设置为 undefined ，然后退出方法。
- 返回 `false` - 操作被终止，然后退出方法。

如果它不返回一个值（undefined），方法触发实体项所属的组的*on_edit_form_close_query* 事件处理程序（如果为组定义了它）。如果定义了这个事件处理程序，并且

- 返回 `true` - 表单窗体被销毁，实体项的*edit_form* 属性被设置为 undefined ，然后退出方法。
- 返回 `false` - 操作被终止，然后退出方法。

如果它不返回一个值（undefined），则触发"任务"的*on_edit_form_close_query* 事件处理程序。如果定义了这个事件处理程序，并且

- 返回 `true` - 表单窗体被销毁，实体项的*edit_form* 属性被设置为 undefined ，然后退出方法。
- 返回 `false` - 操作被终止，然后退出方法。

如果未定义任何事件处理程序，或者这些事件处理程序都没有返回 `false` ，表单窗体将被销毁，并且实体项的*edit_form* 属性被设置为 undefined 。

另请参见

表单窗体

create_edit_form

edit_form

close_filter_form

close_filter_form()

使用范围: client

编程语言: javascript

父类: *Item 类*

描述说明

使用 `close_filter_form` 方法关闭实体项的过滤器表单窗体。

如果定义了*on_filter_form_close_query* 事件处理程序，那么 `close_filter_form` 方法会触发它。如果定义了事件处理程序，并且

- 返回 `true` - 表单窗体被销毁，实体项的*filter_form* 属性被设置为 undefined ，然后退出方法。
- 返回 `false` - 操作被终止，然后退出方法。

如果它不返回一个值（undefined），方法触发实体项所属的组的*on_filter_form_close_query* 事件处理程序（如果为组定义了它）。如果定义了这个事件处理程序，并且

- 返回 `true` - 表单窗体被销毁，实体项的*filter_form* 属性被设置为 undefined ，然后退出方法。
- 返回 `false` - 操作被终止，然后退出方法。

如果它不返回一个值（undefined），则触发"任务"的*on_filter_form_close_query* 事件处理程序。如果定义了这个事件处理程序，并且

- 返回 `true` - 表单窗体被销毁，实体项的*filter_form* 属性被设置为 undefined ，然后退出方法。
- 返回 `false` - 操作被终止，然后退出方法。

如果未定义任何事件处理程序，或者这些事件处理程序都没有返回 `false` ，表单窗体将被销毁，并且实体项的*filter_form* 属性被设置为 undefined 。

另请参见

表单窗体

create_filter_form

filter_form

close_view_form

close_view_form()

使用范围: client

编程语言: javascript

父类: *Item 类*

描述说明

使用 `close_view_form` 方法关闭实体项的视图表单窗体。

如果定义了*on_view_form_close_query* 事件处理程序，那么 `close_view_form` 方法会触发它。如果定义了事件处理程序，并且

- 返回 `true` - 表单窗体被销毁，实体项的*view_form* 属性被设置为 undefined ，然后退出方法。
- 返回 `false` - 操作被终止，然后退出方法。

如果它不返回一个值（undefined），方法触发实体项所属的组的*on_view_form_close_query* 事件处理程序（如果为组定义了它）。如果定义了这个事件处理程序，并且

- 返回 `true` - 表单窗体被销毁，实体项的*view_form* 属性被设置为 undefined ，然后退出方法。
- 返回 `false` - 操作被终止，然后退出方法。

如果它不返回一个值（undefined），则触发"任务"的*on_view_form_close_query* 事件处理程序。如果定义了这个事件处理程序，并且

- 返回 `true` - 表单窗体被销毁，实体项的*view_form* 属性被设置为 undefined ，然后退出方法。
- 返回 `false` - 操作被终止，然后退出方法。

如果未定义任何事件处理程序，或者这些事件处理程序都没有返回 `false` ，表单窗体将被销毁，并且实体项的*view_form* 属性被设置为 undefined 。

另请参见

表单窗体

view

view_form

copy

copy(*options*)

使用范围: client

编程语言: javascript

父类: *Item 类*

描述说明

使用 `copy` 方法创建一个实体项的副本。创建的副本不会被添加到任务树 中，而且不再需要时，它会被 JavaScript 垃圾处理程序回收。

副本对象的所有属性都是在应用程序启动时加载任务树时被定义的。请见工作流

`Options` parameter further specifies the created copy. It can have the following attributes: `Options` 参数进一步指定了创建的副本，它可以有下列属性：

- `handlers` - 如果这个属性的值是 `true` ，那么，在应用程序构建器的表单窗体对话框中对实体项做的所有设置和在实体项客户端模块中定义的所有函数和事件也可以在副本里生效和使用。其默认值是 `true` 。
- `filters` - 如果这个属性的值是 `true` ，将为副本创建设置的过滤器，否则，副本没有过滤器。其默认值是 `true` 。
- `details` -如果这个属性的值是 `true` ，将为副本创建设置的明细项，否则，副本没有明细项。其默认值是 `true` 。

示例

在演示项目 中，使用下面的代码来异步计算字段值的总和，其结果显示在 **发票 (Invoice)** 业务表的页脚位置：

```
function on_filter_applied(item) {
    var copy = item.copy({handlers: false, details: false});
    copy.assign_filters(item);
    copy.open(
        {fields: ['subtotal', 'tax', 'total'],
        funcs: {subtotal: 'sum', tax: 'sum', total: 'sum'}},
        function() {
            var footer = item.view_form.find('.dbtable.' + item.item_name + ' tfoot');
            copy.each_field(function(f) {
                footer.find('div.' + f.field_name)
                    .css('text-align', 'right')
                    .css('color', 'black')
                    .text(f.display_text);
            });
        }
    );
}
```

另请参见

任务树

工作流

create_detail_views

create_detail_views (*container*)

使用范围: client

编程语言: javascript

描述说明

使用 create_detail_views 方法创建实体项的明细项的视图表单窗体。可以在编辑表单对话框 的 **编辑明细 (Edit details)** 属性里指定明细项，或是在*edit_options* 的 edit_details 属性里设置明细项。

这个方法通常用在 on_edit_form_created 事件处理程序里。

The following parameters are passed to the method: 下列参数传递给方法：

- container - 一个 JQuery 对象，它包含明细项的视图表单窗体。如果没有指定此参数，方法将返回。

如果有多个明细项，该方法将在窗体的不同选项卡里创建视图表单。

如果明细项没处于*active* 状体，该方法会调用它们的*open* 方法来激活明细项。

示例

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

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

    if (item.on_edit_form_created) {
        item.on_edit_form_created(item);
    }

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

    return true;
}
```

create_edit_form

create_edit_form(*container*)

使用范围: client

编程语言: javascript

父类: *Item 类*

描述说明

使用 create_edit_form 方法创建一个实体项的编辑表单窗体，以对记录可视化编辑。

该方法在“任务”的*templates* 属性里搜索实体项的 html 模板，然后创建模板的一个副本并将其分配给实体项的*edit_form* 属性。

如果指定了 container 参数，则该方法将其清空并将 html 模板附加到该参数上。否则，它将创建一个模态表单并将 html 附加到该表单上。

触发“任务”的*on_edit_form_created* 事件。

如果为实体项所属的组定义了*on_edit_form_created* 事件, 那么触发定义的事件。

如果为实体定义了*on_edit_form_created* 事件, 那么触发定义的事件。

将 JQuery 的 keyup 和 keydown 事件分配给 edit_form 后，当窗体的 JQuer 事件发生时，会触发 `on_edit_form_keyup` 和 `on_edit_form_keydown` 事件。会以相同的方法触发已定义的事件：首先是“任务”的事件处理程序，然后是“组”的事件处理程序，最后是实体项自己的事件处理程序。之后，调用事件的 JQuery stopPropagation 方法。

如果表单是模态形式，则显示它。在显示表单之前，该方法会先启用在*edit_options* 属性里指定的选项。

触发“任务”的*on_edit_form_shown* 事件。

如果为实体项所属的组定义了*on_edit_form_shown* 事件, 那么触发定义的事件。

如果为实体定义了*on_edit_form_shown* 事件, 那么触发定义的事件。

另请参见

表单窗体

edit_form

edit_options

create_edit_form

close_edit_form

create_filter_form

create_filter_form(*container*)

使用范围: client

编程语言: javascript

父类: *Item* 类

描述说明

使用 `create_filter_form` 方法创建一个实体项的过滤器表单窗体，以对记录过滤进行可视化编辑。

该方法在“任务”的*templates* 属性里搜索实体项的 html 模板（请见表单窗体 ），然后创建模板的一个副本并将其分配给实体项的*filter_form* 属性。

如果指定了 `container` 参数，则该方法将其清空并将 html 模板附加到该参数上。否则，它将创建一个模态表单并将 html 附加到该表单上。

触发“任务”的*on_filter_form_created* 事件。

如果为实体项所属的组定义了*on_filter_form_created* 事件, 那么触发定义的事件。

如果为实体定义了*on_filter_form_created* 事件, 那么触发定义的事件

将 JQuery 的 keyup 和 keydown 事件分配给 filter_form 后，当窗体的 JQuer 事件发生时，会触发 `on_filter_form_keyup` 和“on_filter_form_keydown‘ 事件。会以相同的方法触发已定义的事件：首先是“任务”的事件处理程序，然后是“组”的事件处理程序，最后是实体项自己的事件处理程序。之后，调用事件的 JQuery stopPropagation 方法。

如果表单是模态形式，则显示它。在显示表单之前，该方法会先启用在*filter_options* 属性里指定的选项。

触发“任务”的*on_filter_form_shown* 事件。

如果为实体项所属的组定义了*on_filter_form_shown* 事件, 那么触发定义的事件。

如果为实体定义了*on_filter_form_shown* 事件, 那么触发定义的事件。

另请参见

表单窗体

filter_form

filter_options

close_filter_form

create_filter_inputs

create_filter_inputs(*container*, *options*)

使用范围: client

编程语言: javascript

描述说明

使用 `create_filter_inputs` 方法创建数据感知控件（输入框、复选框等）以编辑实体项的*filters* 。

通常，在由*create_filter_form* 方法触发的 `on_filter_form_created` 事件中使用该方法。

下列参数可以传递给该方法:

- `container` - 一个包含可视化控件的 JQuery 对象，如果该对象的长度为 0 （没有），该方法将直接返回。
- `options` - 指定如何显示控件的选项。

`options` 参数是一个可以有下列属性的对象:

- `filters` - 过滤器的名称列表。如果指定了该属性，那么将为该列表中的每个过滤器创建一个可视化的控件；如果没有指定该属性（默认不指定），那么将使用*filter_options* 的字段属性（默认情况下，它会列出在应用程序构建器中指定的所有可见过滤器）。
- `col_count` - 为可视化控件创建的列数，其默认值是 1 。
- `label_on_top`: 其默认值是 false 。如果值为 false，标签将被放置于控件的左测；否则，在控件的上面创建标签。
- `tabindex` - 如果指定了 `tabindex` ，它将是第一个可视控件的选项卡索引，所有后续控件的 tabindex 值将依次增加 1 。
- `autocomplete` - 其默认值是 false 。如果将此属性设置为 true ，控件的自动补全属性将被设置为 "on "。

在创建空间之前，应用程序将清空 `container` 。

示例

```
function on_filter_form_created(item) {
    item.filter_options.title = item.item_caption + ' - filter';
    item.create_filter_inputs(item.filter_form.find(".edit-body"));
    item.filter_form.find("#cancel-btn").on('click.task', function() {
        item.close_filter()
    });
    item.filter_form.find("#ok-btn").on('click.task', function() {
        item.apply_filter()
    });
}
```

另请参见

filters

create_filter_form

filter_form

filter_options

create_inputs

create_inputs(*container*, *options*)

使用范围: client

编程语言: javascript

描述说明

使用 `create_inputs` 方法创建数据感知控件（输入框、复选框等）以编辑实体项的*filters* 。

通常，在 `on_edit_form_created` 事件中使用该方法。

下列参数可以传递给该方法：

- `container` - 一个包含可视化控件的 JQuery 对象，如果该对象的长度为 0 （没有），该方法将直接返回。
- `options` - 指定如何显示控件的选项。

`options` 参数是一个可以有下列属性的对象：

- `fields` - 字段的名称列表。如果指定了该属性，那么将为该列表中的每个字段创建一个可视化的控件；如果没有指定该属性，那么将使用*edit_options* 中已定义的 `fields` 属性，否则，将创建在应用程序构建器的编辑表单对话框 中的设置布局。
- `col_count` - 为可视化控件创建的列数，其默认值是 1 (1,2,3,4,6,12)。

在创建空间之前，应用程序将清空 `container` 。

示例

```
function on_edit_form_created(item) {
    item.create_inputs(item.edit_form.find(".left-div"),
        {fields: ['firstname', 'lastname', 'company', 'support_rep_id']}
    );
}
```

另请参见

fields

Data-aware controls

create_edit_form

create_table

create_table(*container*, *options*)

使用范围: client

编程语言: javascript

描述说明

使用 create_table 方法创建一个表格来显示实体项数据集的记录。

表格的行为由实体项的*paginate* 属性决定。

当*paginate* 为 true 时，将创建一个分页器，在页面切换时，它将在内部更新实体项数据集。

当*paginate* 为 false 时，实体项数据集的所有可用记录都将同时显示在表格里。

这个方法创建的是数据感知类型的表格，当你变更数据集时，这些更改会立即反映在表格中。因此，你可以先创建一个表格，然后再调用*open* 方法。

下列参数可以传递给该方法：

- container - 一个包含可视化控件的 JQuery 对象，如果该对象的长度为 0（没有），该方法将直接返回。在创建空间之前，应用程序将清空 container 。
- options - 指定表格显示方法的选项。在创建表格时，该方法默认使用*table_options* ，这些属性是在应用程序构建器的视图表单对话框 中设置的属性。options 属性优先于*table_options* 属性。

options 参数一个与*table_options* 有相同属性的对象。

Examples

```
function on_edit_form_created(item) {
    item.edit_options.width = 1050;
    item.invoice_table.create_table(item.edit_form.find(".edit-detail"),
        {
            height: 400,
            editable_fields: ['quantity'],
            column_width: {"track": "60%"}
        });
}
```

另请参见

表单窗体

数据感知控件

create_view_form

create_view_form(*container*)

使用范围: client

编程语言: javascript

父类: *Item 类*

描述说明

使用 `create_view_form` 方法创建实体项的视图表单。

该方法在“任务”的*templates* 属性里搜索实体项的 html 模板（请见表单窗体 ），然后创建模板的一个副本并将其分配给实体项的*view_form* 属性。

如果指定了 `container` 参数，则该方法将其清空并将 html 模板附加到该参数上。否则，它将创建一个模态表单并将 html 附加到该表单上。

触发“任务”的*on_view_form_created* 事件。

如果为实体项所属的组定义了*on_view_form_created* 事件, 那么触发定义的事件。

如果为实体定义了*on_view_form_created* 事件, 那么触发定义的事件。

将 JQuery 的 keyup 和 keydown 事件分配给 edit_form 后，当窗体的 JQuer 事件发生时，会触发 `on_view_form_keyup` 和 `on_view_form_keydown` 事件。会以相同的方法触发已定义的事件：首先是“任务”的事件处理程序，然后是“组”的事件处理程序，最后是实体项自己的事件处理程序。之后，调用事件的 JQuery stopPropagation 方法。

如果表单是模态形式，则显示它。在显示表单之前，该方法会先启用在*view_options* 属性里指定的选项。

触发“任务”的*on_view_form_shown* 事件。

如果为实体项所属的组定义了*on_view_form_shown* 事件, 那么触发定义的事件。

如果为实体定义了*on_view_form_shown* 事件, 那么触发定义的事件。

另请参见

表单窗体

view_form

view_options

close_view_form

delete

delete()

使用范围: client

编程语言: javascript

父类: *Item 类*

描述说明

删除活动的记录，并将游标移到下一条记录上。

`delete` 方法

- 检查实体项数据集是否处于:doc:*active <at_active>* 状态。如果不是，则引发异常。
- 检查实体项数据集是否不为空。如果为空，则引发异常。
- 如果实体项是一个明细项 ，那么检查其主实体项是否处于“编辑”或“插入”状态 ，如果不是，则引发异常。
- 如果实体项不是一个明细项 ，那么检查其是否处于“浏览”状态 ，如果不是，则引发异常。
- 触发为实体项定义的*on_before_delete* 事件处理程序。

- 将实体项置于“删除”状态 。
- 删除活动的记录，并将游标移到下一条记录。
- 将实体项置于“浏览”状态
- 触发为实体项定义的*on_after_delete* 事件处理程序。
- 更新数据感知控件

另请参见

修改数据集

delete_record

delete_record()

使用范围: client

编程语言: javascript

父类: *Item* 类

描述说明

- 调用*can_delete* 方法检查一个用户是否有权限删除操作的记录，如果没有权限，则直接返回。
- 如果用户有权限，则要求用户是否确认操作。
- 调用*delete* 方法删除操作的记录。
- 调用*apply* 方法将变更写入应用程序数据库。

另请参见

修改数据集

delete

disable_controls

disable_controls()

使用范围: client

编程语言: javascript

父类: *Item* 类

描述说明

调用 `disable_controls` 方法“关闭”控件的数据感知功能。因此，控件不再实时反应实体项数据集中数据的变更。

调用*enable_controls* 方法，以重新启用与数据集关联的数据感知控件中的数据显示，同时更新它们显示的数据值。

示例

```
function calculate(item) {
    var subtotal,
        tax,
        total,
        rec;
    if (!item.calculating) {
        item.calculating = true;
        try {
            subtotal = 0;
            tax = 0;
            total = 0;
            item.invoice_table.disable_controls();
            rec = item.invoice_table.rec_no;
            try {
                item.invoice_table.each(function(d) {
                    subtotal += d.amount.value;
                    tax += d.tax.value;
                    total += d.total.value;
                });
            }
            finally {
                item.invoice_table.rec_no = rec;
                item.invoice_table.enable_controls();
            }
            item.subtotal.value = subtotal;
            item.tax.value = tax;
            item.total.value = total;
        }
        finally {
            item.calculating = false;
        }
    }
}
```

另请参见

数据感知控件

enable_controls

disable_edit_form

disable_edit_form()

使用范围: client

编程语言: javascript

父类: *Item 类*

描述说明

当*edit_form* 可见时，调用 `disable_edit_form` 方法可以阻止任何用户的操作。

调用*enable_edit_form* 方法重新启用编辑表单对窗体。

示例

```
function on_edit_form_created(item) {
    var save_btn = item.add_edit_button('Save and continue');
    save_btn.click(function() {
        if (item.is_changing()) {
            item.disable_edit_form();
            item.post();
            item.apply(function(error){
                if (error) {
                    item.alert_error(error);
                }
                item.edit();
                item.enable_edit_form();
            });
        }
    });
}
```

另请参见

enable_edit_form

each

each (*function(item)*)

使用范围: client

编程语言: javascript

父类: *Item 类*

描述说明

Use each method to iterate over records of an item dataset. 使用 `each` 方法来迭代访问一个实体项所拥有的记录。

`each` 方法指定了一个函数，这个函数会为每条记录执行。

通过使回调函数返回 `false` ，你能在一个特定的迭代中终止 each 循环。

示例

下面的示例中，**t** 和 **item.invoice_table** 都指向了同一个对象。

```
var subtotal = 0,
    tax = 0,
    total = 0;
item.invoice_table.each(function(t) {
    subtotal += t.amount.value;
    tax += t.tax.value;
    total += t.total.value;
});
```

另请参见

导航数据集

each_detail

each_detail (*function(detail)*)

使用范围: client

编程语言: javascript

父类: *Item 类*

描述说明

使用 `each_detail` 方法来迭代访问一个实体项的所有明细项。

`each_detail` 方法指定了一个函数，这个函数会为实体项的每个明细项执行。（当前明细项作为参数传递给该方法）

通过使回调函数返回 `false` ，你能在一个特定的迭代中终止 each_detail 循环。

另请参见

明细

each_field

each_field (*function(field)*)

使用范围: client

编程语言: javascript

父类: *Item 类*

描述说明

使用 `each_field` 方法来迭代访问一个实体项所拥有的*fields* 。

`each_field` 方法指定了一个函数，这个函数会为实体项的每个字段执行。（当前字段作为参数传递给该方法）

通过使回调函数返回 `false` ，你能在一个特定的迭代中终止 each_field 循环。

示例

```
function customer_fields(customers) {
    customers.open({limit: 1});
    customers.each_field(function(f) {
        console.log(f.field_caption, f.display_text);
    });
}
```

字段

Field 类

each_filter

each_filter (*function(filter)*)

使用范围: client

编程语言: javascript

父类: *Item 类*

描述说明

使用 each_filter 方法来迭代访问一个实体项所拥有的*filters* 。

each_filter 方法指定了一个函数，这个函数会为实体项的每个过滤器执行。(当前过滤器作为参数传递给该方法)

通过使回调函数返回 false ，你能在一个特定的迭代中终止 each_filter 循环。

示例

```
function customer_filters(customers) {
    customers.each_filter(function(f) {
        console.log(f.filter_caption, f.value);
    });
}
```

过滤器

Filter 类

edit

edit ()

使用范围: client

编程语言: javascript

父类: *Item 类*

描述说明

在数据集中，启用数据的编辑。

调用 edit 之后，应用程序将允许用户变更记录的字段中的数据，稍后能使用*post* 方法将这些变更提交到实体项数据集，最后使用*apply* 方法将变更保存到数据库。

edit 方法

- 检查实体项数据集是否处于**活动 (active)* 状态。如果不是，则引发异常。
- 检查实体项数据集是否不为空。如果为空，则引发异常。
- 检查实体项数据集是否已处于“编辑”状态。如果是，则直接返回。
- 如果实体项是一个明细项 ，那么检查其主实体项是否处于“编辑”或“插入”状态 。如果不是，则引发异常。
- 如果实体项不是:doc:“明细项 </programming/data/details>‘，那么检查它是否处于“浏览”状态 。如果部不是，则引发异常。

- 触发已为实体项定义的*on_before_edit* 事件处理程序。
- 将实体项置于“编辑”状态，允许应用程序或用户修改记录中的字段的值。
- 触发已为实体项定义的*on_after_edit* 事件处理程序。

另请参见

修改数据集

edit_record

edit_record(*container*)

使用范围: client

编程语言: javascript

父类: *Item* 类

描述说明

将当前记录置于“编辑”状态，并且创建一个*edit_form* 用于记录的可视化编辑。

如果指定了 container 参数（DOM 元素的 JQuery 对象），编辑表单窗体的 html 模板将被插入到 container 中。

如果未指定 container 参数，但在编辑表单对话框中设置了 **无模式表单 (Modeless form)** 属性，或者以编程方式设置了*edit_options* 的无模式属性，并且设置了任务的*forms_in_tabs* 属性，而应用程序没有模式表单，则将在任务的*forms_container* 对象的新选项卡中创建无模式编辑表单。

在其它情况下，将创建模态表单窗体。

如果在非模态模式下允许编辑记录，那么用户能同时（在提交保存之前）编辑多条记录。在这种情况下，应用程序会调用*copy* 方法创建实体项的一个副本。这个副本用来编辑记录。应用程序将调用*open* 方法，以主键字段的值作为筛选条件，从服务器中获取相应的记录。

在模态编辑的情况下，应用程序执行*refresh_record* 方法从服务器获取记录的最新数据。

如果为实体项启用了*record locking*，并且从服务器接收记录数据，则应用程序将接收记录的版本。

然后 edit_record 方法

- 调用*can_edit* 方法检查一个用户是否有权限编辑此记录
- 如果用户有权限编辑此记录，检查实体项是否处于“编辑”或“插入”状态，如果不是，则调用:doc:‘edit <m_edit>‘方法类编辑记录。
- 调用*create_edit_form* 方法创建一个表单窗体以用于记录的可视化编辑。

另请参见

表单窗体

修改数据集

edit

can_create

记录锁定

enable_controls

enable_controls()

使用范围: client

编程语言: javascript

父类: *Item 类*

描述说明

调用 `enable_controls` 方法以允许在数据感知控件中显示数据，并在之前调用*disable_controls* 后再重新绘制它们。

另请参见

数据感知控件

disable_controls.

enable_edit_form

enable_edit_form()

使用范围: client

编程语言: javascript

父类: *Item 类*

描述说明

在调用*disable_edit_form* 方法之后，调用 `enable_edit_form` 方法以重新启用编辑表单窗体，

示例

```
function on_edit_form_created(item) {
    var save_btn = item.add_edit_button('Save and continue');
    save_btn.click(function() {
        if (item.is_changing()) {
            item.disable_edit_form();
            item.post();
            item.apply(function(error){
                if (error) {
                    item.alert_error(error);
                }
                item.edit();
                item.enable_edit_form();
            });
        }
    });
}
```

另请参见

disable_edit_form

eof

eof()

使用范围: client

编程语言: javascript

父类: *Item 类*

描述说明

调用 `eof` (end-of-file) 来确定游标是否指向一个实体项数据集的最后一条记录。如果 eof 返回 `true`, 游标就明确地位于数据集中的最后一行上。

eof 返回 true ，当应用程序:

- 打开一个空数据集。.
- 调用了实体项的*last* 方法。
- 调用了实体项的*next* 方法，并且执行失败（因为游标已经在数据集的最后一行上）。

在其它情况下，eof 返回 false 。

> **备注**
>
> 如果 eof 和*bof* 都返回 `true`, 则实体项数据集是空的。

另请参见

数据集

导航数据集

field_by_name

field_by_name(*field_name*)

使用范围: client

编程语言: javascript

父类: *Item 类*

描述说明

当只知道字段的名称时，调用 `field_by_name` 方法来检索字段的信息。

`field_name` 参数是一个已存在的字段的名称。

`field_by_name` 返回指定字段的字段对象。如果指定的字段不存在，那么 `field_by_name` 返回 `null` 。

filter_by_name

filter_by_name (*filter_name*)

使用范围: client

编程语言: javascript

父类: *Item 类*

描述说明

当只知道过滤器的名称时，调用 filter_by_name 方法来检索过滤器的信息。

filter_name 参数是一个已存在的过滤器的名称。

filter_by_name 返回指定过滤器的过滤器对象。如果指定的过滤器不存在，那么 filter_by_name 返回 null 。

first

first ()

使用范围: client

编程语言: javascript

父类: *Item 类*

描述说明

调用 first 将光标定位在实体项数据集中的第一条记录上，并使其成为活动记录。first 会将任何更改提交到活动的记录中。

另请参见

数据集

导航数据集

insert

insert ()

使用范围: client

编程语言: javascript

父类: *Item 类*

描述说明

在实体项的数据集中插入一个新的空记录。

在调用 insert 后，应用程序让用户在记录的字段中输入数据，然后使用*post* 方法将这些更改提交到实体项数据集，并使用*apply* 方法将它们应用到实体项数据库表。

insert 方法

- 检查实体项数据集是否处于*active* 状态，否则引发异常

- 如果实体项是明细项，则检查主实体项是否处于编辑或插入*state*，否则引发异常
- 如果实体项不是明细项，则检查它是否处于“浏览”*state*，否则引发异常
- 如果为实体项定义了*on_before_append* 事件处理程序，则触发
- 在实体项数据集中插入一个新的空记录。
- 将实体项置于插入*state*
- 如果为项目定义了*on_after_append* 事件处理程序，则触发。
- 更新数据感知控件。

另请参见

修改数据集

insert_record

insert_record(*container*)

使用范围: client

编程语言: javascript

父类: *Item* 类

描述说明

打开数据集开头的一个新的空记录，并创建一个*edit_form* 用于可视化编辑记录。

如果指定了 `container` 参数（DOM 元素的 Jquery 对象），则编辑表单的 HTML 模板将插入到该容器中。

如果未指定 `container` 参数，但在编辑表单对话框中设置了 **Modeless form** 属性，或者通过编程设置了*edit_options* 的 modeless 属性，并且任务具有*forms_in_tabs* 属性设置，且应用程序没有模态表单，则将在任务的*forms_container* 对象的新选项卡中创建无模式编辑表单。

在所有其他情况下，将创建模态表单。

如果允许在无模式下插入记录，应用程序将调用*copy* 方法来创建项目的副本。此副本将用于插入记录。

`insert_record` 方法

- 调用*can_create* 方法检查用户是否有权插入记录，如果没有，则返回
- 检查项目是否处于编辑或插入*state*，如果不是，则调用*insert* 方法插入记录
- 调用*create_edit_form* 方法创建用于记录可视化编辑的表单。

另请参见

表单窗体

修改数据集

insert

can_create

is_changing

is_changing()

使用范围: client

编程语言: javascript

父类: *Item 类*

描述说明

检查一个实体项是否处于“编辑”或“插入”状体。如果是，则返回 `true` 。

应用程序通过调用*edit* 方法可以将一个实体项置于“编辑”状态，通过调用*append* 或者*insert* 方法可以将一个实体项置于“插入”状态。

另请参见

修改数据集

is_edited

is_edited()

使用范围: client

编程语言: javascript

父类: *Item 类*

描述说明

检查一个实体项是否处于“编辑”状体。如果是，则返回 `true` 。

应用程序通过调用*edit* 方法可以将一个实体项置于“编辑”状态，

另请参见

修改数据集

is_modified

is_modified()

使用范围: client

编程语言: javascript

父类: *Item 类*

描述说明

Checks if the current record of an item dataset has been modified during edit or insert opertaions. The method returns `false` after the *post* method is executed. 检查实体项数据集的当前记录在“编辑”或“插入”操作时是否被修改了。在执行*post* 方法之后，is_modified 方法返回 `false` 。

另请参见

修改数据集

is_new

is_new()

使用范围: client

编程语言: javascript

父类: *Item 类*

描述说明

检查一个实体项是否处于“插入”状体。如果是，则返回 `true` 。

应用程序通过调用*append* 或者*insert* 方法可以将一个实体项置于“插入”状态。

另请参见

修改数据集

last

last()

使用范围: client

编程语言: javascript

父类: *Item 类*

描述说明

调用 `last` 方法将游标移到实体项数据集的最后一条记录上，并使之成为“活动 (active)”记录。

另请参见

数据集

导航数据集

locate

locate(*fields*, *values*)

使用范围: client

编程语言: javascript

父类: *Item 类*

描述说明

实现一种方法，用于在实体数据集中搜索指定的记录，并使该记录成为“活动”记录。

参数：

- `fields`：字段名称，或字段名称列表
- `values`：字段值，或字段值列表

此方法定位记录，其中 `fields` 参数指定的字段具有 `values` 参数指定的值。

如果找到匹配指定条件的记录，并将光标重新定位到该记录，则 `Locate` 返回 true 。

如果未找到匹配的记录或光标未重新定位，则此方法返回 false 。

另请参见

数据集

导航数据集

next

next ()

使用范围: client

编程语言: javascript

父类: *Item 类*

描述说明

调用 `next` 方法将游标移到实体项数据集的下一条记录上，并使之成为“活动 (active)”记录。`next` 会提交当前“活动”记录的变更。

另请参见

数据集

导航数据集

open

open (*options*, *callback*, *async*)

使用范围: client

编程语言: javascript

父类: *Item 类*

描述说明

调用 `open` 向服务端发送请求以获取实体项的数据集。

`open` 方法可以有以下参数：

- `options` - 一个对象，指定发送到服务端的请求参数

- callback：如果参数不存在，则同步向服务端发送请求，否则，异步执行请求，并在收到数据集后执行回调
- async：如果其值为 true，且缺少 callback 参数，则异步执行请求

参数的顺序无关紧要。

该方法根据 options 初始化实体项的*fields* 并制定请求参数，如果为实体项定义了*on_before_open* 事件处理程序，则触发。

之后，它向服务端发送请求。如果指定了 callback 参数对应的函数，则异步执行请求，否则同步执行。

服务端在收到请求后，检查服务端上相应的实体项（具有相同 ID 属性的*task 树* 中的实体项）是否具有*on_open* 事件处理程序。如果有，它执行此事件处理程序并将执行结果返回给客户端；否则，根据请求参数生成 SELECT SQL 查询语句，执行此查询并将结果返回给客户端。

客户端在收到请求结果后，更改其数据集，将*active* 设置为 true，将*item_state* 设置为“浏览”模式，转到数据集的第一条记录，触发*on_after_open* 和*on_filters_applied* 事件处理程序（如果为实体项定义了），并更新控件。

然后，如果指定了 callback 参数，则调用对应的函数。

Options

options 对象参数可以具有以下属性：

- expanded - 如果此属性的值为 true，则在服务端生成的 SELECT 查询语句将具有 JOIN 子句以获取查找字段 的查找值，否则不会返回查找值。默认值为 true。
- fields - 使用此参数指定 SELECT 查询的 WHERE 子句。此参数是字段名称列表。如果省略，则使用*set_fields* 方法定义的字段。如果在 open 方法执行之前未调用*set_fields* 方法，则使用开发者创建的所有字段。
- where - 使用此参数指定如何在 SQL 查询中过滤记录。此参数是“键值对”对象，其中键是字段名称，后跟双下划线，然后是过滤符号（请参阅过滤记录 ）。如果省略，则使用*set_where* 方法定义的值。如果在 open 方法执行之前未调用*set_where* 方法，且省略了 where 参数，则使用为项目定义的过滤器 的值来过滤记录。
- order_by - 使用 order_by 指定记录的排序顺序。此参数是字段名称列表。如果字段名称前有'-' 符号，则在此字段上按降序排序记录。如果省略，则使用*set_order_by* 方法定义的列表。
- offset - 使用 offset 指定要返回的第一行记录的偏移量。
- limit - 使用 limit 限制 SQL 查询语句的输出“前几行”。
- funcs - 此参数可以是“键值对”对象，其中键是字段名称，值是将在 SELECT 查询中应用于字段的函数名称。
- group_by - 使用 group_by 指定用于对查询结果进行分组的字段。此参数必须是字段名称列表。
- open_empty - 如果此参数设置为 true，应用程序不会向服务端发送请求，而是仅仅初始化一个空的数据集。默认值为 false。
- params - 使用此参数传递一些用户定义的选项，以在服务端的*on_open* 事件处理程序中使用。此参数必须是“键值对”对象。

备注

当实体项的*paginate* 属性设置为 true 且通过*create_table* 方法创建表时，limit 和 offset 参数由表根据其行数和当前页面在内部设置。

Examples

```
function get_customer_sales(task, customer_id) {
    var date1 = new Date(new Date().setYear(new Date().getFullYear() - 5)),
        date2 = new Date(),
        invoices = task.invoices.copy();

    invoices.open({
        fields: ['customer', 'invoicedate', 'total'],
        where: {customer: customer_id, invoicedate__ge: date1, invoicedate__le: date2},
        order_by: ['invoicedate']
    });
}
```

```
function get_customer_sales(task, customer_id) {
    var date1 = new Date(new Date().setYear(new Date().getFullYear() - 5)),
        date2 = new Date(),
        invoices = task.invoices.copy();

    invoices.set_fields(['customer', 'invoicedate', 'total']);
    invoices.set_where({customer: customer_id, invoicedate__ge: date1, invoicedate__le: date2});
    invoices.set_order_by(['invoicedate']);
    invoices.open();
}
```

```
function get_sales(task) {
    var sales = task.invoices.copy();

    sales.open({
        fields: ['customer', 'id', 'total'],
        funcs: {'id': 'count', 'total': 'sum'},
        group_by: ['customer'],
        order_by: ['customer']
    });
}
```

post

post()

使用范围: client

编程语言: javascript

父类: *Item 类*

描述说明

将修改后的记录写入实体项的数据集。调用 post 以保存在调用*append* ， *insert* 或*edit* 方法之后对记录所做的更改。

post 方法：

- 检查一个实体项是否处于“编辑”或“插入”*状态 (state)* ，如果不是，则抛出异常。
- 触发为实体项已经定义的*on_before_post* 事件处理程序。
- 检查那条记录是否合法有效，如果不合法，则抛出异常。

- 如果实体项有明细项，则将当前记录提交到明细项。
- 将变更添加到实体项日志中
- 将实体项置于“浏览”状态 *(state)*
- 触发为实体项已经定义的*on_after_post* 事件处理程序。

另请参见

修改数据集

prior

prior()

使用范围: client

编程语言: javascript

父类: *Item* 类

描述说明

调用 `prior` 将游标定位在实体项数据集中的前一条记录上，并使其成为活动记录。最后，提交对活动记录的任何更改。

另请参见

数据集

导航数据集

record_count

record_count()

使用范围: client

编程语言: javascript

父类: *Item* 类

描述说明

Call `record_count` to get the total number of records owned by the item's dataset. 调用 `record_count` 方法获取实体项的数据集拥有的记录的总数。

示例

```
item.open()
if (item.record_count()) {
    // some code
}
```

另请参见

数据集

open

refresh_page

refresh_page(*callback*, *async*)

使用范围: client

编程语言: javascript

父类: *Item 类*

描述说明

调用 `refresh_page` 向服务器发送请求，以获取当前页面的当前数据并刷新现有的可视化控件。

`refresh_page` 方法有以下参数：

- `callback`: 如果参数不存在，则同步向服务器发送请求，否则异步执行请求，然后执行回调
- `async`: 如果其值为 true ，并且缺少回调参数，则异步执行请求

refresh_record

refresh_record(*options*, *callback*, *async*)

使用范围: client

编程语言: javascript

父类: *Item 类*

描述说明

调用 `refresh_record` 向服务器发送请求，以获取当前记录的当前数据并刷新现有的可视化控件。

`refresh_record` 方法有以下参数：

- `callback`: 如果参数不存在，则同步向服务器发送请求，否则异步执行请求，然后执行回调
- `async`: 如果其值为 true ，并且缺少回调参数，则异步执行请求
- `options` - 一个包含 `details` 属性的对象，`details` 是实体项对应的明细项的名称列表，它也会随之被刷新。

参数的顺序无关紧要。

search

search(*field_name*, *value*, *search_type*, *callback*)

使用范围: client

编程语言: javascript

父类: *Item 类*

描述说明

调用 search 方法，向服务器发送一个请求，生成并执行一个 SQL 查询语句，以获取满足字段查询条件的所有记录。

查询也将满足当前设置的过滤器或实体项的 where 条件。

将使用返回的数据集更新已有的可视化控件中的内容。

参数如下:

- field_name - 字段的名称
- value - 满足条件的字段的值
- search_type - 字符串格式的搜索类型，参见过滤记录 中的过滤符号
- callback - 一个回调函数，将在执行搜索后被执行

另请参见

数据集

过滤记录

select_records

select_records (*field_name*, *all_records*)

使用范围: client

编程语言: javascript

父类: *Item 类*

描述说明

使用 select_records 方法，通过从字段的查找项中选择记录，将记录添加到实体项中。

例如，“Demo 应用程序”使用这个方法从“曲目主表 (Tracks catalog)”中查询“曲目 (track)”，并将其添加到“发票 (invoice)”中。

参数:

- field_name : 此参数是一个实体项的查找字段的字段名称。
- all_records : 如果设置为 true ，将添加对应数据表中的所有记录。否则，该方法会忽略实体项中已经存在的记录（那些记录早已添加到了实体项）。

示例

```
function on_view_form_created(item) {
    var btn = item.add_view_button('Select', {type: 'primary'});
    btn.click(function() {
        item.select_records('track');
    });
}
```

set_fields

set_fields (*field_list*)

使用范围: client

编程语言: javascript

父类: *Item 类*

描述说明

当没指定*open* 方法的可选参数 fields 时，使用 set_fields 在内部定义并存储*open* 方法使用的 fields 参数。

在执行*open* 方法之后，它将清除这在些内部存储的值。

field_list 参数一个由字段名组成的列表。

示例

以下代码片段的执行结果相同:

```
item.open({fields: ['id', 'invoicedate']});
```

```
item.set_fields(['id', 'invoicedate']);
item.open();
```

另请参见

数据集

open

set_order_by

set_order_by (*field_list*)

使用范围: client

编程语言: javascript

父类: *Item 类*

描述说明

当没指定*open* 方法的可选参数 order_by 时，使用 set_order_by 在内部定义并存储*open* 方法使用的 order_by 参数。

在执行*open* 方法之后，它将清除这些在内部存储的值。

field_list 参数一个由字段名组成的列表。如果在一个字段名称前添加了符号“-”，那么将存储按这个字段降序排序后的记录。

示例

以下代码片段的执行结果相同:

```
item.open({order_by: ['-invoicedate']});
```

```
item.set_order_by(['-invoicedate']);
item.open();
```

另请参见

数据集

open

set_where

set_where(*where*)

使用范围: client

编程语言: javascript

父类: *Item* 类

描述说明

当没指定*open* 方法的可选参数 _where 时，使用 set__where 在内部定义并存储*open* 方法使用的 _where 参数。

在执行*open* 方法之后，它将清除这些在内部存储的值。

where 参数是一个由"键-值"对组成的对象。"键"是字段的名称，紧跟冒号":"，之后是过滤器符号。(参见过滤记录)。

示例

以下代码片段的执行结果相同:

```
item.open({where: {id: 100}});
```

```
item.set_where({id: 100});
item.open();
```

另请参见

数据集

open

show_history

show_history()

使用范围: client

编程语言: javascript

父类: *Item 类*

描述说明

实体项的 `show_history` 方法打开一个对话框，显示所选记录的更改历史。

另请参见

保存用户所做更改的历史记录

update_controls

update_controls()

使用范围: client

编程语言: javascript

父类: *Item 类*

描述说明

调用*update_controls* 告诉相关联的控件刷新显示以反应当前的数据。

另请参见

Data-aware controls

disable_controls

enable_controls

view

view(*container*)

使用范围: client

编程语言: javascript

父类: *Item 类*

描述说明

使用 `view` 方法创建实体项的一个视图表单窗体。

该方法会检查实体项及其父项的 javascript 模块是否已被加载，如果没有（*项目的参数* 中设置了“动态加载 JS 模块”），就加载它们。

如果指定了 `container` 参数（DOM 元素的 Jquery 对象），视图表单窗体的 html 模板将被插入到容器（conainer）中。

如果调用了“任务 (task)”的*init_tabs* 方法，将为窗体创建选项卡。

以上完成以后，将调用*create_view_form* 方法。

示例

In the following code the view for of the **Tasks** journal is created in the on_page_loaded event handler:

```
function on_page_loaded(task) {

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

    task.init_tabs($("#content"));
    task.tasks.view($("#content"));

    $(window).on('resize', function() {
        resize(task);
    });
}
```

另请参见

表单窗体

view_form

view_options

create_view_form

close_view_form

master_applies

master_applies

使用范围: client

编程语言: javascript

父类: *Item 类*

描述说明

如果在 master/Detail 场景中，将 `master_applies` 属性设置为 **true** ，则 **master** 将应用对明细项的“添加/删除”操作所做的更改。

如果该属性设置为 **false** ，则 **Detail** 将在创建新记录时立即应用更改以保证 Details 中的数据集的完整。

示例

```
function on_view_form_created(item) {
    item.invoice_table.master_applies = true;
}
```

另请参见

明细项

事件

on_after_append

on_after_append(item)

使用范围: client

编程语言: javascript

父类: *Item 类*

描述说明

在应用程序插入或追加一条记录之后发生此事件。

`item` 参数是触发了这个事件的实体项。

编写一个 `on_after_append` 事件处理程序，应用程序在实体项中插入或追加一条记录之后，立即执行指定的操作。

`on_after_append` 方法由*insert* 或*append* 方法调用。

另请参见

修改数据集

on_after_apply

on_after_apply(item)

使用范围: client

编程语言: javascript

父类: *Item 类*

描述说明

应用程序在项目数据库中保存变更日志之后发生此事件。

`item` 参数是触发了这个事件的实体项。

编写一个 `on_after_apply` 事件处理程序，应用程序在项目数据库中保存变更日志之后，立即执行指定的操作。

`on_after_apply` 方法由*apply* 方法触发。

另请参见

修改数据集

on_after_cancel

on_after_cancel(item)

使用范围: client

编程语言: javascript

父类: *Item 类*

描述说明

应用程序取消对实体项数据集的修改之后发生此事件。

`item` 参数是触发了这个事件的实体项。

编写一个 `an on_after_cancel` 事件处理程序，应用程序在取消对实体项数据集的修改之后，立即执行指定的操作。

另请参见

修改数据集

on_after_delete

on_after_delete(item)

使用范围: client

编程语言: javascript

父类: *Item 类*

描述说明

应用程序删除一条记录之后发生此事件。

`item` 参数是触发了这个事件的实体项。

编写一个 `on_after_delete` 事件处理程序，应用程序删除实体项的活动记录之后，立即执行指定的操作。

`on_after_delete` 在删除记录后由*delete* 调用，并将光标重新定位在刚刚删除的记录之前的记录上。

另请参见

修改数据集

on_after_edit

on_after_edit(item)

使用范围: client

编程语言: javascript

父类: *Item 类*

描述说明

应用程序编辑一条记录之后发生此事件。

item 参数是触发了这个事件的实体项。

编写一个 on_after_edit 事件处理程序，应用程序在编辑一条记录之后，立即执行指定的操作。

on_after_edit 由 by *edit* 调用。

另请参见

修改数据集

on_after_open

on_after_open(item)

使用范围: client

编程语言: javascript

父类: *Item 类*

描述说明

应用程序接收到来自服务端获取数据集的响应之后发生此事件。

item 参数是触发了这个事件的实体项。

编写一个 on_after_open 事件处理程序，应用程序在接收到来自服务端获取数据集的响应之后，立即执行指定的操作。

on_after_open 由*open* method 调用。

在演示应用程序中，此事件用于标识所购曲目的发票:

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

另请参见

数据集

on_after_post

on_after_post(item)

使用范围: client

编程语言: javascript

父类: *Item 类*

描述说明

应用程序将一条记录提交到数据集之后发生此事件。

`item` 参数是触发了这个事件的实体项。

编写一个 `on_after_post` 事件处理程序，应用程序在将一条记录提交到数据集之后，立即执行指定的操作。

`on_after_post` 由*post* method 调用。

另请参见

修改数据集

on_after_scroll

on_after_scroll(item)

使用范围: client

编程语言: javascript

父类: *Item 类*

描述说明

应用程序由一条记录滚动到另一条记录之后发生此事件。

`item` 参数是触发了这个事件的实体项。

编写一个 `on_after_scroll` 事件处理程序，以便在应用程序因调用以下方法而滚动到另一条记录之后立即采取特定操作：*first* , *last* , *next* , *prior* , 和*locate*。

`on_after_scroll` 在这些方法触发的所有其他事件以及在实体项数据集中从一个记录切换到另一个记录的任何其他方法之后调用。

示例

在演示项目 中使用以下代码，在 **invoice** 业务台账记录更改后异步打开 **invoice_table** 明细数据集。它还禁用按钮：

```
var scroll_timeout;

function on_after_scroll(item) {
    clearTimeout(scroll_timeout);
    scroll_timeout = setTimeout(
        function() {
            if (item.view_form && item.rec_count) {
                item.view_form.find("#delete-btn, #paid-btn").prop("disabled", item.paid.value);
            }
        }, 50
    );
}
```

另请参见

导航数据集

on_before_scroll

on_before_append

on_before_append(item)

使用范围: client

编程语言: javascript

父类: *Item* 类

描述说明

在应用程序插入或追加一条记录之前发生此事件。

`item` 参数是触发了这个事件的实体项。

编写一个 `on_before_append` 事件处理程序，应用程序在实体项中插入或追加一条记录之前，立即执行指定的操作。

`on_before_append` 方法由*insert* 或*append* 方法调用。

另请参见

修改数据集

on_before_apply

on_before_apply(item, params)

使用范围: client

编程语言: javascript

父类: *Item* 类

描述说明

应用程序在项目数据库中保存数据集的变之前发生此事件。

`item` 参数是触发了这个事件的实体项。

`params` 参数是一个传递给*apply* 方法的对象。如果未指定，则传递一个空对象。这个对象被传递到服务端，在*on_apply* 事件处理程序中使用它，当在数据库中保存变更时，能够执行一些操作。

编写一个 `an on_before_apply` 事件处理程序，应用程序在项目数据库中保存变更日志之前，立即执行指定的操作。

`on_before_apply` 方法由*apply* 方法触发。

另请参见

修改数据集

on_before_cancel

on_before_cancel(item)

使用范围: client

编程语言: javascript

父类: *Item 类*

描述说明

应用程序取消对实体项数据集的修改之前发生此事件。

`item` 参数是触发了这个事件的实体项。

编写一个 `on_before_cancel` 事件处理程序，应用程序在取消对实体项数据集的修改之前，立即执行指定的操作。

另请参见

修改数据集

on_before_delete

on_before_delete(item)

使用范围: client

编程语言: javascript

父类: *Item 类*

描述说明

应用程序删除一条记录之前发生此事件。

`item` 参数是触发了这个事件的实体项。

编写一个 `on_before_delete` 事件处理程序，应用程序删除实体项的活动记录之前，立即执行指定的操作。

`on_before_delete` 在删除记录后由*delete* 调用。

另请参见

修改数据集

on_before_edit

on_before_edit(item)

使用范围: client

编程语言: javascript

父类: *Item 类*

描述说明

在应用程序启用编辑记录功能之前发生此事件。

item 参数是触发了这个事件的实体项。

编写一个 on_before_edit 事件处理程序，应用程序在启用编辑记录功能之前，立即执行指定的操作。on_before_edit 由 by *edit* 调用。

另请参见

修改数据集

on_before_field_changed

on_before_field_changed(field)

使用范围: client

编程语言: javascript

父类: *Item 类*

描述说明

编写 on_before_field_changed 事件处理程序，在字段数据更改之前执行任何特殊处理。

field 参数是其数据设计到变更的字段。

获取拥有字段的实体项，使用字段的*owner* 属性。

在触发此事件处理程序之前，应用程序将要设置的新值分配给字段的"new_value"属性。您可以更改此属性的值。此值将用于更改字段的数据。

示例

```
function on_before_field_changed(field) {
    if (field.field_name === 'quantity' && field.new_value < 0) {
        field.new_value = 0;
    }
}
```

另请参见

字段

value

on_before_field_changed

on_before_open

on_before_open(item, params)

使用范围: client

编程语言: javascript

父类: *Item 类*

描述说明

应用程序向服务端发送请求以获取数据集之前发生此事件。

`item` 参数是触发了这个事件的实体项。

`params` 参数是一个传递给*open* 方法的对象。如果未指定，则传递一个空对象。这个对象被传递到服务端，在*on_open* 事件处理程序中使用它，当获取数据集时，能够执行一些操作。

编写一个 `on_before_open` 事件处理程序，在应用程序从服务端获取数据之前，立即执行指定的操作。

`on_before_open` 由*open* method 调用。

另请参见

数据集

on_before_post

on_before_post(item)

使用范围: client

编程语言: javascript

父类: *Item 类*

描述说明

应用程序将一条记录提交到数据集之前发生此事件。

`item` 参数是触发了这个事件的实体项。

编写一个 `on_before_post` 事件处理程序，应用程序在将一条记录提交到数据集之前，立即执行指定的操作。

`on_before_post` 由*post* method 调用。

另请参见

修改数据集

on_before_print_report

请见

Report 类

on_before_scroll

on_before_scroll(item)

使用范围: client

编程语言: javascript

父类: *Item 类*

描述说明

应用程序由一条记录滚动到另一条记录之前发生此事件。

`item` 参数是触发了这个事件的实体项。

编写一个 `on_before_scroll` 事件处理程序，以便在应用程序因调用以下方法而滚动到另一条记录之前立即采取特定操作： *first* , *last* , *next* , *prior* , 和*locate*。

`on_before_scroll` 在这些方法触发的所有其他事件以及在实体项数据集中从一个记录切换到另一个记录的任何其他方法之前调用。

另请参见

导航数据集

on_after_scroll

on_detail_changed

on_detail_changed(item, detail)

使用范围: client

编程语言: javascript

父类: *Item 类*

描述说明

在明细项记录的更改提交之后发生。它使用 clearTimeout 和 setTimeout 这两个 Javascript 函数，因此如果记录在一个周期内发生了更改，则只有在最后一次记录更改发生时才会触发。

`item` 参数是触发了这个事件的实体项。`detail` 参数是那个已经被更改的明细项。

编写一个 `on_detail_changed` 事件处理程序，通过使用:doc:'calc_summary <m_calc_summary>'方法计算明细项的字段的总和，并将这些值保存在其主表的字段中。

示例

```
function on_detail_changed(item, detail) {
    var fields;
    if (detail.item_name === 'invoice_table') {
        fields = [
            {"total": "total"},
            {"tax": "tax"},
            {"subtotal": "amount"}
        ];
        item.calc_summary(detail, fields);
    }
}
```

另请参见

明细项

calc_summary

master_applies

on_edit_form_close_query

on_edit_form_close_query(item)

使用范围: client

编程语言: javascript

父类: *Item 类*

描述说明

`on_edit_form_close_query` 事件由实体项的*close_edit_form* 方法触发。

`item` 参数是触发了这个事件的实体项。

另请参见

表单窗体

create_edit_form

edit_form

close_edit_form

on_edit_form_created

on_edit_form_created(item)

使用范围: client

编程语言: javascript

父类: *Item 类*

描述说明

`on_edit_form_created` 事件，在表单窗体被创建但还未显示之前，由实体项的*create_edit_form* 方法触发。

`item` 参数是触发了这个事件的实体项。

演示项目 使用下面的代码将 "发票 (Invoices)" 数据表的 read_only 字段设置为 `paid` 字段的值，并删除页脚：

```
function on_edit_form_created(item) {
    item.read_only = item.paid.value;
    item.edit_form.find('.form-footer').remove();
}
```

另请参见

表单窗体

create_edit_form

edit_form

on_edit_form_keydown

on_edit_form_keydown(item, event)

使用范围: client

编程语言: javascript

父类: *Item 类*

描述说明

`on_edit_form_keydown` 事件在实体项的编辑表单窗体 上发生 keydown 事件时被触发。

`item` 参数是触发了这个事件的实体项。

`event` 是一个 JQuery 事件对象。

另请参见

表单窗体

create_edit_form

edit_form

on_edit_form_keyup

on_edit_form_keyup(item, event)

使用范围: client

编程语言: javascript

父类: *Item 类*

描述说明

`on_edit_form_keyup` 事件在实体项的编辑表单窗体 上发生 keyup 事件时被触发。

`item` 参数是触发了这个事件的实体项。

`event` 是一个 JQuery 事件对象。

另请参见

表单窗体

create_edit_form

edit_form

on_edit_form_shown

on_edit_form_shown(item)

使用范围: client

编程语言: javascript

父类: *Item 类*

描述说明

on_edit_form_shown 事件在表单窗体已经显示后由*create_edit_form* 方法触发。

item 参数是触发了这个事件的实体项。

另请参见

表单窗体

create_edit_form

edit_form

on_field_changed

on_field_changed(field, lookup_item)

使用范围: client

编程语言: javascript

父类: *Item 类*

描述说明

编写一个 on_field_changed 事件处理程序，以响应字段数据中的任何更改。

field 参数时那个数据已经被更改的字段。要获取拥有这个字段的实体项，请使用过滤器的*owner* 属性。

当字段是查找字段 并且当用户从查找项数据集中选择记录后发生了变化时，lookup_item 参数不是 undefined 。

示例

```
function on_field_changed(field, lookup_item) {
    var item = field.owner;
    if (field.field_name === 'quantity' || field.field_name === 'unitprice') {
        item.owner.calc_total(item);
    }
    else if (field.field_name === 'track' && lookup_item) {
        item.quantity.value = 1;
        item.unitprice.value = lookup_item.unitprice.value;
    }
}
```

另请参见

字段

value

on_before_field_changed

on_field_get_html

on_field_get_html(field)

使用范围: client

编程语言: javascript

父类: *Item 类*

描述说明

编写 `on_field_get_html` 事件处理程序来指定在表格中将插入到该字段的单元格中的 `html` ，

如果事件处理程序没有返回任何值，应用程序检查是否定义了*on_field_get_text* 事件处理程序；如果返回了一个值，将使用*display_text* 属性的值在单元格中来显示该字段的值。

`field` 参数是一个字段，它的*display_text* 已经被处理。要获取拥有此字段的实体项，请使用字段的*owner* 属性。

示例

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

另请参见

字段

on_field_get_text

on_field_get_summary

on_field_get_summary

on_field_get_summary(field, value)

使用范围: client

编程语言: javascript

父类: *Item 类*

描述说明

编写 `on_field_get_summary` 事件处理程序，来指定将被插入到字段的汇总单元格中的 `summary` 。

`field` 参数是一个字段，它的*on_field_get_summary* 已经被处理。要获取拥有此字段的实体项，请使用字段的*owner* 属性。

示例

```
function on_field_get_summary(field, value) {
    let item = field.owner;
    if (field.field_name === 'customer') {
        return item.rec_count;
    }
    // to use with Virtual Table
    if (field.field_name === 'total') {
        let sum = 0;
        item.each(function(row) {
            sum += parseFloat(row.total.value) || 0;
        });
        return '$' + sum.toFixed(2);
    }
}
```

另请参见

字段

on_field_get_text

on_field_get_html

on_field_get_text

on_field_get_text(field)

使用范围: client

编程语言: javascript

父类: *Item* 类

描述说明

编写 `on_field_get_text` 事件处理程序，来指定将被插入到字段的单元格中的 `text` 。

`field` 参数是一个字段，它的*display_text* 已经被处理。要获取拥有此字段的实体项，请使用字段的*owner* 属性。

该函数用于连接两个或多个字段进行显示。这些字段可以是其他表中的任何可以访问的字段。

示例

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

另请参见

字段

on_field_get_html

on_field_select_value

on_field_select_value(field, lookup_item)

使用范围: client

编程语言: javascript

父类: *Item 类*

描述说明

当用户点击字段输入框右侧的按钮或使用与输入时，应用程序将创建该字段查找项的一个副本，并触发 `on_field_select_value` 事件。使用 `on_field_select_value` 指定将要现实的字段，在数据被打开之前为其查找项设置过滤器。

简而言之，当我们点击查找图标时，`on_field_select_value` 用于定位或标识查找表上的记录。

`field` 参数是其数据将被选择的字段。

`lookup_item` 参数是字段的查找项的副本。

示例

```
function on_field_select_value(field, lookup_item) {
    if (field.field_name === 'customer') {
        lookup_item.set_where({lastname__startwith: 'B'});
        lookup_item.view_options.fields = ['firstname', 'lastname', 'address', 'phone'];
    }
}
```

Or, generic code with different ID for each table:

```
function on_field_select_value(field, lookup_item) {
    const field_map = {
        album: 'id',
        genre: 'genreid',
        artist: 'artist_id'
    };

    const target_field = field_map[field.field_name];
    if (target_field && field.value) {
        const where = {};
        where[target_field + '__eq'] = field.value;
        lookup_item.set_where(where);
    }
}
```

另请参见

字段

查找字段

on_field_validate

on_field_validate(field)

使用范围: client

编程语言: javascript

父类: *Item 类*

描述说明

编写 `on_field_validate` 事件处理程序来验证对字段数据所作的更改。

`field` 参数是一个数据已经被更改的字段。要获取拥有该字段的实体项，请使用字段的*owner* 属性。

如果字段值非法，该事件处理程序必须返回字符串。

当调用*post* 方法或者用户离开用于编辑字段值的输入框时，触发此事件。

示例

```
function on_field_validate(field) {
    if (field.field_name === 'sum' && field.value > 10000000) {
            return 'The sum is too big.';
    }
}
```

另请参见

字段

value

如何验证字段的值

on_filter_changed

on_filter_changed(filter, lookup_item)

使用范围: client

编程语言: javascript

父类: *Item 类*

描述说明

编写一个 `on_filter_changed` 事件处理程序，以响应过滤器数据中的任何更改。

`filter` 参数时那个数据已经被更改的过滤器。要获取拥有这个过滤器的实体项，请使用过滤器的*owner* 属性。

另请参见

过滤器

value

on_filter_form_close_query

on_filter_form_close_query(item)

使用范围: client

编程语言: javascript

描述说明

`on_filter_form_close_query` 事件由实体项的*close_filter_form* 方法触发。

`item` 参数是触发了这个事件的实体项。

另请参见

表单窗体

create_filter_form

filter_form

close_filter_form

on_filter_form_created

on_filter_form_created(item)

The item parameter is the item that triggered the event.

使用范围: client

编程语言: javascript

描述说明

`on_filter_form_created` 事件，在表单窗体被创建但还未显示之前，由实体项的*create_filter_form* 方法触发。

`item` 参数是触发了这个事件的实体项。

另请参见

表单窗体

create_filter_form

filter_form

on_filter_form_shown

on_filter_form_shown(item)

The item parameter is the item that triggered the event.

使用范围: client

编程语言: javascript

描述说明

on_filter_form_shown 事件在表单窗体已经显示后由*create_filter_form* 方法触发。

`item` 参数是触发了这个事件的实体项。

另请参见

表单窗体

create_filter_form

filter_form

on_filter_record

on_filter_record(item)

The item parameter is the item that triggered the event.

使用范围: client

编程语言: javascript

描述说明

使用 `on_filter_record` 事件在本地过滤数据集记录。

当游标移动到另一个记录上时，并且*Filtered* 属性已被设置为 `true` ，会触发该事件。

编写 `on_filter_record` 事件处理程序，为数据集中的每条记录指定它是否应该对应用程序可见。为了表示记录通过了筛选条件，`on_filter_record` 事件处理程序必须返回 true 。

`item` 参数是触发了这个事件的实体项。

示例

```
function on_filter_record(item) {
    if (item.type.value === 2) {
        return true;
    }
}

function enable_filtering(item) {
    item.filtered = true;
}

function disable_filtering(item) {
    item.filtered = false;
}
```

另请参见

Filtered

on_filters_applied

on_filters_applied(item)

使用范围: client

编程语言: javascript

父类: *Item 类*

描述说明

编写 `on_filters_applied` 事件处理成，在过滤器已经被应用到实体项数据集时，进行特殊的处理。

另请参见

过滤器

on_open_report

请见

报表事件

on_param_select_value

待公布 (TBA)

on_param_form_close_query

请见

Report 类

on_param_form_created

请见

Task 类

on_param_form_shown

请见

Task 类

on_param_select_value

待公布 (TBA)

on_view_form_close_query

on_view_form_close_query(item)

使用范围: client

编程语言: javascript

父类: *Item 类*

描述说明

`on_view_form_close_query` 事件由实体项的*close_view_form* 方法触发。

`item` 参数是触发了这个事件的实体项。

另请参见

表单窗体

view

view_form

close_view_form

on_view_form_created

on_view_form_created(item)

使用范围: client

编程语言: javascript

父类: *Item 类*

描述说明

`on_view_form_created` 事件，在表单窗体被创建但还未显示之前，由*view* 方法触发。

`item` 参数是触发了这个事件的实体项。

在演示项目 中，有使用了 **on_view_form_created** 的代码。

另请参见

表单窗体

view

view_form

on_view_form_keydown

on_view_form_keydown(item, event)

使用范围: client

编程语言: javascript

父类: *Item 类*

描述说明

"on_view_form_keydown' 事件在实体项的*view form* 上发生 keydown 事件时被触发。

`item` 参数是触发了这个事件的实体项。

`event` 是一个 JQuery 事件对象。

另请参见

表单窗体

view

view_form

on_view_form_keyup

on_view_form_keyup(item, event)

使用范围: client

编程语言: javascript

父类: *Item 类*

描述说明

`on_view_form_keyup` 事件在实体项的*view form* 上发生 keyup 事件时被触发。

`item` 参数是触发了这个事件的实体项。

`event` 是一个 JQuery 事件对象。

另请参见

表单窗体

view

view_form

on_view_form_shown

on_view_form_shown(item)

使用范围: client

编程语言: javascript

父类: *Item 类*

描述说明

on_view_form_shown 事件在表单窗体已经显示后由*view* 方法触发。

`item` 参数是触发了这个事件的实体项。

另请参见

表单窗体

view

view_form

7.1.5 Detail Detail 类

`class Detail()`

使用范围: client

编程语言: javascript

Detail class 继承了*Item 类* 的属性、方法和事件。

属性

master

`master`

使用范围: client

编程语言: javascript

父类: *Detail 类*

描述说明

使用 `master` 属性来获取对明细项的 *主表 (master)* 的引用。

另请参见

明细项

7.1.6 Reports Reports 类

`class Reports()`

使用范围: client

编程语言: javascript

Reports 类用于创建拥有项目的报表的任务树 中的组对象。

下面列出了的这个类的属性、方法和事件。

它同样继承了其祖先类*AbstractItem 类* 的属性和方法

事件

on_before_print_report

on_before_print_report(item)

使用范围: client

编程语言: javascript

父类: *Reports 类*

描述说明

`on_before_print_report` 事件由*process_report* 方法触发。

`report` 参数是触发该事件的报表。

另请参见

表单窗体

客户端报表编程

process_report

on_open_report

on_open_report(report)

使用范围: client

编程语言: javascript

父类: *Reports 类*

描述说明

`on_open_report` 事件由*process_report* 方法触发。

`report` 参数是触发该事件的报表。

另请参见

客户端报表编程

process_report

on_param_form_close_query

on_param_form_close_query(item)

使用范围: client

编程语言: javascript

父类: *Reports 类*

描述说明

`on_param_form_close_query` 事件由*close_param_form* 方法触发。

`report` 参数是触发该事件的报表。

另请参见

表单窗体

客户端报表编程

print

create_param_form

on_param_form_created

on_param_form_created(item)

使用范围: client

编程语言: javascript

父类: *Reports 类*

描述说明

`on_param_form_created` 事件由在*print* 内经常调用的*create_param_form* 方法触发。

`report` 参数是触发该事件的报表。

另请参见

表单窗体

客户端报表编程

print

create_param_form

on_param_form_shown

on_param_form_shown(item)

使用范围: client

编程语言: javascript

父类: *Reports 类*

描述说明

`on_param_form_shown` 事件由在*print* 内经常调用的*create_param_form* 方法触发。

method.

另请参见

表单窗体

客户端报表编程

print

create_param_form

7.1.7 Report Report 类

class Report()

使用范围: client

编程语言: javascript

下面列出了的这个类的属性、方法和事件。

它同样继承了其祖先类*AbstractItem 类* 的属性和方法

属性

extension

extension

使用范围: client

编程语言: javascript

父类: *Report 类*

描述说明

使用 `extension` 属性来指定一个报表类型。服务端基于报表模板先生成 **ods** 文件。如果报表的扩展名不是 **ods** , , 则使用 Libre Office 执行转换。

该属性值可以是任何 Libre Office 支持转换为的任何扩展名。

示例

```
function on_before_print_report(report) {
    report.extension = 'html';
}
```

另请参见

客户端报表编程

报表的服务器端编程

print

create_param_form

on_before_print_report

param_form

param_form

使用范围: client

编程语言: javascript

父类: *Report 类*

描述说明

使用 `param_form` 属性访问表示报表表单窗体的 JQuery 对象。

它由经常被*print* 调用的*create_param_form* 方法创建。

close_param_form 方法将 `param_form` 的值设置为 undefined 。

示例

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

另请参见

表单窗体

print

create_param_form

close_param_form

param_options

param_options

使用范围: client

编程语言: javascript

父类: *Report 类*

描述说明

使用 `param_options` 属性指定模态参数表单的参数。

`param_options` 是一个对象，具有以下属性：

- `width` - 模态表单的宽度，默认值为 560 px 。
- `title` - 模态表单的标题，默认值为*report_caption* 属性的值。
- `close_button` - 如果为 true，则在表单右上角创建关闭按钮，默认值为 true，
- `close_caption` - 如果为 true 且 `close_button` 为 true，则在按钮旁显示 "Close - [Esc]"。
- `close_on_escape` - 如果为 true，按下 Escape 键将触发*close_param_form* 方法。
- `close_focusout` - 如果为 true，当表单失去焦点时将调用*close_param_form* 方法。
- `template_class` - 如果指定，则将在任务的*templates* 属性中查找具有此类的 div，并在创建表单时将其用作表单 HTML 模板。

示例

```
function on_param_form_created(report) {
    report.param_options.width = 800;
    report.param_options.close_button = false;
    report.param_options.close_on_escape = false;
}
```

另请参见

表单窗体

print

create_param_form

close_param_form

方法

close_param_form

close_param_form()

使用范围: client

编程语言: javascript

父类: *Report 类*

描述说明

使用 `close_param_form` 方法来关闭报表的参数表单窗体。

`close_param_form` 方法会触发以为报表定义的*on_param_form_close_query* 事件处理程序。如果定义了事件处理程序，并且

- 返回 true - 销毁表单窗体，将报表的*param_form* 属性设置为 undefined ，该方法退出。
- 返回 false - 操作中止，该方法退出。

如果它不返回值（undefined)，该方法会触发以为所属报表所属组定义的*on_param_form_close_query* 事件。如果此事件处理程序已定义，并且

- 返回 true - 表单被表单窗体，报表的*param_form* 属性被设置为 undefined，该方法退出
- 返回 false - 操作中止，该方法退出

如果它不返回值（undefined)，该方法会触发任务的*on_param_form_close_query*。如果此事件处理程序已定义，并且

- 返回 true - 表单被销毁，报表的*param_form* 属性被设置为 undefined，方法退出
- 返回 false - 操作中止，方法退出

如果没有定义事件处理程序，或者这些事件处理程序都未返回 false，则表单被销毁，并且报表的*param_form* 属性被设置为 undefined 。

另请参见

表单窗体

客户端报表编程

print

create_param_form

create_param_form()

使用范围: client

编程语言: javascript

父类: *Report 类*

描述说明

create_param_form 方法由*print* 方法调用，用于创建一个表单以在通过*process_report* 方法将请求发送到服务端之前设置报表参数。

该方法检查报表及其所有者的 javascript 模块是否已加载，如果未加载（且设置了*JS 模块动态加载参数*），则加载它们。

然后它在任务*templates* 属性中搜索报表的 html 模板（参见*表单窗体*），并创建该模板的副本，将其分配给报表*param_form* 属性。

创建一个表单窗体，并为其添加 html 。

触发“任务”的*on_param_form_created* 事件。

触发已为报表组定义的*on_param_form_created* 事件。

触发已为报表定义的*on_param_form_created* 事件。

显示表单窗体。在显示窗体之前，方法会应用在*param_options* 属性中指定的参数选项。

触发“任务”的*on_param_form_shown* 事件。

触发已为报表组定义的*on_param_form_created* 事件。

触发已为报表定义的*on_param_form_shown* 事件。

另请参见

表单窗体

客户端报表编程

print

create_param_inputs

create_param_inputs(*container*, *options*)

使用范围: client

编程语言: javascript

描述说明

使用 create_param_inputs 方法，为编辑报表参数创建数据感知控件（输入框、复选框等）。

该方法经常用在 on_param_form_created 事件中，这个事件由*print* 经常调用的*create_param_form* 方法触发。

下面的参数将传递给该方法：

- container - 一个包含可视化控件的 JQuery 对象。如果该参数长度是 0 （即，没有容器），那么该方法将直接返回。
- options - 指定控件如何显示的选项参数。

options‘ 参数是一个有下列属性的对象：

- `params` - 参数名称的列表。如果指定了此参数，那么将为这个列表中的每个参数创建一个可视化控件。如果未指定（默认值），那么将为应用程序构建器中指定的所有可视化参数创建控件。
- `col_count` - 为可视化控件创建的列的数量，默认是 1 。
- `label_on_top`: 默认值是 false 。如果值为 false ，用于提示内容的标签将位于控件的左边，否则位于控件的上边。
- `tabindex` - 如果指定了此参数，它将是首个可视化控件的 tab 按键的索引值。所有后续控件的索引值将依次加 1。
- `autocomplete` - 默认值是 false 。如果将其设置为 true，控件的自动完成属性将被设置为 “on”。

应用程序会在创建控件之前清空指定的容器。

示例

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

另请参见

create_param_form

param_form

param_options

print

print(*create_form*)

使用范围: client

编程语言: javascript

父类: *Report 类*

描述说明

使用 `print` 方法来打印报表。

如果 `create_form` 参数被省略或等于 `true` ，则该方法调用:doc:*create_param_form* <*m_create_param_form*> 方法，以基于在 index.html 文件中定义的 html 模板创建表单。

如果 `create_form` 参数设置为 `false` ，并且报告没有可见参数，则它调用*process_report* 向服务端发送请求以生成报告，否则它调用*create_param_form* 方法

另请参见

表单窗体

报表参数

客户端报表编程

create_param_form

process_report

process_report

process_report()

使用范围: client

编程语言: javascript

父类: *Report 类*

描述说明

process_report 方法向服务端发送报表来生成报表的内容，并访问服务端返回给客户端的报表文件，然后将其打开或保存。

如果 `create_form` 参数等于 false 并且没有设置可见的参数。它由*print* 方法直接调用。如果有可见参数，*print* 方法创建一个表单窗体来指定参数值，并调用它（例如，通过一些按钮点击事件）。

检查参数值是否合法有效，并触发以下事件：

- 报表组的*on_before_print_report* 事件处理程序
- 报表的*on_before_print_report* 事件处理程序

在这个事件处理程序中，开发人员能为报表定义一些通用（报表组事件处理程序）的或特定（报表事件处理程序）的属性。

之后，*process_report* 方法向服务端发送异步请求来生成一个报表的内容。(参考*报表的服务器端编程*)

服务端为该方法返回一个指向文件的 url ，那个文件包含为报表已生成的内容。

然后，该方法检查是否定义了报表组的*on_open_report* 事件处理程序。如果定义了，就调用它。否则，检查是否定义了报表的*on_open_report* 事件处理程序，如果定义了，就调用它。

如果没定义任何一个事件，将在浏览器中打开报表，或者将报表保存到磁盘（是打开还是保存，将依赖于报表的:doc:‘extension </refs/client/report/at_extension>‘属性）。

示例

在下面的事件处理程序中，设置了报表的 **id** 参数的值，这些代码的定义在演示应用程序中的 **发票 (invoice)** 报表的客户端模块中。

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

事件

on_before_print_report

on_before_print_report(report)

使用范围: client

编程语言: javascript

父类: *Report 类*

描述说明

`on_before_print_report` 事件由*process_report* 方法触发。在向服务器发送生成报表的请求之前，使用 `on_before_print_report` 来执行指定的操作。

`report` 参数是触发该事件的报表。

另请参见

客户端报表编程

process_report

on_open_report

on_open_report(report)

使用范围: client

编程语言: javascript

父类: *Report 类*

描述说明

`on_open_report` 事件由*process_report* 方法触发。

`report` 参数是触发该事件的报表。

另请参见

客户端报表编程

process_report

on_param_form_close_query

on_param_form_close_query(report)

使用范围: client

编程语言: javascript

父类: *Report 类*

描述说明

`on_param_form_close_query` 事件由*close_param_form* 方法触发。

`report` 参数是触发该事件的报表。

另请参见

表单窗体

客户端报表编程

close_param_form

on_param_form_created

on_param_form_created(report)

使用范围: client

编程语言: javascript

父类: *Report 类*

描述说明

`on_param_form_created` 事件通常由*print* 调用的*create_param_form* 方法触发。

`report` 参数是触发该事件的报表。

另请参见

表单窗体

客户端报表编程

print

create_param_form

on_param_form_shown

on_param_form_shown(report)

使用范围: client

编程语言: javascript

父类: *Report 类*

描述说明

`on_param_form_shown` 事件通常由*print* 调用的*create_param_form* 方法触发。

`report` 参数是触发该事件的报表。

另请参见

表单窗体

客户端报表编程

print

create_param_form

7.1.8 Field Field 类

class Field()

使用范围: client

编程语言: javascript

属性与特性

display_text

display_text

使用范围: client

编程语言: javascript

父类: *Field* 类

描述说明

将字段的值表示为字符串。

Display_text 属性是字段值的只读形式的字符串表示，用于在数据感知控件中显示。

如果指定了*on_field_get_text* 事件处理程序，**display_text** 将是这个事件处理程序返回的值。否则，**display_text** 是查找字段 的*lookup_text* 属性的值；其他情况下，**display_text** 是字段的*text* 属性值，会根据语言区域设置 的设置对其进行转换。

当不是编辑状态时，**display_text** 是字段值的字符串形式的表示。当字段处于编辑状态时，使用*text* 属性的值。

示例

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

另请参见

字段

查找字段

on_field_get_text

text

lookup_text

field_caption

field_caption

使用范围: client

编程语言: javascript

父类: *Field 类*

描述说明

Field_caption 属性指定要显示给用户的字段的标题。另请参见 ========

数据集

字段

field_name

field_mask

field_mask

使用范围: client

编程语言: javascript

父类: *Field 类*

描述说明

您可以使用 **field_mask** 属性来指定要显示的字段名称。

当您希望他们以特定格式输入数据时（日期、电话号码等），掩码允许用户更容易地输入固定宽度的内容。

掩码由掩码字面值和掩码规则组成的格式来定义。任何不在以下定义列表中的字符都被视为掩码字面值。掩码字面值将在用户输入时自动输入，并且用户无法删除。以下掩码规则是预定义的：

- a - 表示一个字母字符 (A-Z,a-z)
- 9 - 表示一个数字字符 (0-9)
- * - 表示一个字母或数字字符 (A-Z,a-z,0-9)

示例

```
function on_edit_form_created(item) {
    item.phone.field_mask = '999-99-99';
}
```

field_name

field_name

使用范围: client

编程语言: javascript

父类: *Field 类*

描述说明

指定字段在代码中被引用的名称。在代码中使用 **field_name** 来引用字段。

另请参见

数据集

字段

field_caption

field_size

field_size

使用范围: client

编程语言: javascript

父类: *Field 类*

描述说明

指定文本字段的大小（字符个数)。

另请参见

数据集

字段

field_type

field_type

使用范围: client

编程语言: javascript

父类: *Field 类*

描述说明

Identifies the data type of the field object. 指定字段对象的数据类型。

使用 **field_type** 属性来获取字段所包含的数据的类型，其值是下列值之一:

- ”text”,
- ”integer”,
- ”float”,
- ”currency”,
- ”date”,
- ”datetime”,
- ”boolean”,
- ”blob”

另请参见

数据集

字段

lookup_text

lookup_text

使用范围: client

编程语言: javascript

父类: *Field 类*

描述说明

使用 **lookup_text** 属性来获取查找字段 中对应的被转换为字符串的值.

如果字段是查找字段 ，那么 **lookup_text** 属性会获取字段对应的查找文本；否则，获取的是 doc:*text* <*at_text*> 属性的值。

另请参见

字段

查找字段

lookup_value

text

lookup_type

lookup_type

使用范围: client

编程语言: javascript

父类: *Field 类*

描述说明

对于查找字段，获取的是*lookup_value* 的类型；否则，返回*field_type* 属性的值。

另请参见

数据集

字段

lookup_value

lookup_value

使用范围: client

编程语言: javascript

父类: *Field 类*

描述说明

如果字段是查找字段，获取到的是它的查找的值；否则，获取到的是*value* 属性的值。

另请参见

字段

查找字段

lookup_value

lookup_text

owner

owner

使用范围: client

编程语言: javascript

父类: *Field 类*

描述说明

获取字段对象所属的实体项。

检查字段的 owner 属性的值，以确定将该字段对象作为其自身来显示的实体项。

示例

```
function calculate(item) {

}

function on_field_changed(field, lookup_item) {
    if (field.field_name === 'taxrate') {
        calculate(field.owner);
    }
}
```

另请参见

数据集

字段

raw_value

raw_value

使用范围: client

编程语言: javascript

父类: *Field 类*

描述说明

表示一个字段对象中的数据。

使用 **raw_value** 只读属性直接从实体项的数据集中读取数据。其他属性例如*value* 和*text* 会对字段数据进行转换。因此，*value* 属性将数值字段的 **null** 值转换为 **0**。

另请参见

字段

value

text

Field read_only

read_only

使用范围: client

编程语言: javascript

父类: *Field 类*

描述说明

决定着字段在数据感知控件里是否可以被修改。

将 **read_only** 设置为 **true** ，可以防止字段的值在数据感知控件里被修改。

另请参见

字段

required

required

required

使用范围: client

编程语言: javascript

父类: *Field 类*

描述说明

指定一个字段的值是否必须是非空的。

使用 **required** 来判断一个字段是必须有一个值，还是可以为空。当将 **required** 设置为 true ，试图保存 null 时，将会触发一个异常。

另请参见

字段

read_only

text

text

使用范围: client

编程语言: javascript

父类: *Field 类*

描述说明

使用 **text** 属性来设置或获取字段的文本值。

获取 text 属性的值

获取*value* 属性的值，并将其转换为文本。

设置 text 属性的值

按字段的类型对文本进行转换，并将字段的*value* 属性指定为转换后的值。

另请参见

字段

查找字段

lookup_value

text

lookup_text

value

value

使用范围: client

编程语言: javascript

父类: *Field 类*

描述说明

使用 **value** 属性来获取或设置字段的值。

获取 value

当字段的类型是 "integer"、"float" 或 "currency" 时，数据 **null** 将被转换为 **0** 。如果字段类型是 "text"，那么会转换为 空字符串。

对于查找字段 ，value 属性是一个整数，它是查找实体项中相应记录的 id 字段的值。要获取字段的查找的值，请使用*lookup_value* 属性。

设置 value

当指定了一个新值，字段会检查当前值是否不等于新指定的值。如果不等于，则

- 将 **new_value** 属性设置为新指定的值。
- 如果为字段定义了*on_before_field_changed* 事件处理程序，则会被触发。
- 将字段数据改为 **new_value** 属性的值，并将 **new_value** 设置为 **null** 。
- 将实体项标记为已被修改。因此，*is_modified* 方法将返回 **true** 。
- 如果为字段定义了*on_field_changed* 事件处理程序，则会被触发。
- 更新数据感知控件

示例

```
function calc_total(item) {
    item.amount.value = item.round(item.quantity.value * item.unitprice.value, 2);
    item.tax.value = item.round(item.amount.value * item.owner.taxrate.value / 100, 2);
    item.total.value = item.amount.value + item.tax.value;
}
```

另请参见

字段

查找字段

lookup_value

text

lookup_text

方法

download

download()

使用范围: client

编程语言: javascript

父类: *Field 类*

描述说明

在类型为 FILE 的字段上调用 `download` 函数来下载字段存储的文件。

示例

```
function on_view_form_created(item) {
  item.add_view_button('Download').click(function() {
     item.attachment.download();
  });
}
```

open

open()

使用范围: client

编程语言: javascript

父类: *Field 类*

描述说明

在类型为 FILE 的字段上调用 `open` 函数，它会通过 `window.open` 方法来打开指向字段存储的文件的地址。

示例

```
function on_view_form_created(item) {
  item.add_view_button('Open').click(function() {
     item.attachment.open();
  });
}
```

7.1.9 Filter Filter 类

class Filter()

使用范围: client

编程语言: javascript

属性与特性

filter_caption

filter_caption

使用范围: client

编程语言: javascript

父类: *Filter 类*

描述说明

用 **Filter_caption** 属性指定过滤器显示给用户的标题。

另请参见

过滤器

filter_name

数据集

filter_name

filter_name

使用范围: client

编程语言: javascript

父类: *Filter* 类

描述说明

指定在代码中引用过滤器的名称。在代码中，使用 **filter_name** 来引用字段的过滤器。

另请参见

过滤器

filter_caption

数据集

owner

owner

使用范围: client

编程语言: javascript

Filter 类

描述说明

获取一个过滤器对象属于哪个实体项。

检查 owner 属性的值来判断使用过滤器来显示其自身的实体项。

value

value

使用范围: client

编程语言: javascript

父类: *Filter* 类

描述说明

使用 **value** 属性来获取或设置过滤器的值。

示例

```
function on_view_form_created(item) {
    item.filters.invoicedate1.value = new Date(new Date().setYear(new Date().getFullYear() -␣
↪1));
}
```

另请参见

过滤器

visible

数据集

visible

visible

使用范围: client

编程语言: javascript

父类: *Filter 类*

描述说明

如果未指定 **filters** 选项，那么此属性为 **true** 时，过滤器的输入控件会被*create_filter_inputs* 方法创建。

另请参见

过滤器

value

数据集

7.2 服务端 (python) 的类参考

框架中的所有对象呈现为一个*task 树* 。

下面是每个任务树对象的类。

7.2.1 App 类

class App

使用范围: server

编程语言: python

App 类被用于创建一个 WSGI 应用程序。

下面列出了该类的属性。

admin

admin

使用范围: server

编程语言: python

父类: *App 类*

描述说明

该属性返回对应用程序构建器的任务树的引用。

另请参见

工作流

Task 树

task

task

使用范围: server

编程语言: python

父类: *App 类*

描述说明

该属性返回对项目中任务树的引用。

另请参见

工作流

Task 树

7.2.2 AbstractItem 类

class AbstractItem

使用范围: server

编程语言: python

AbstractItem 类是*task 树* 的所有实体项对象的祖先。

下面列出了该类的属性和方法。

属性

environ

environ

使用范围: server

编程语言: python

父类: *AbstractItem 类*

描述说明

指定来自客户端的当前请求的 WSGI 环境字典。

另请参见

服务器端编程

session

ID

ID

使用范围: server

编程语言: python

父类: *AbstractItem 类*

描述说明

`ID` 属性在实体项的框架 ID 中是唯一的、

当按数字而不是名称引用项目时，`ID` 属性最有用。它也在内部使用。

另请参见

Task 树

item_caption

item_caption

使用范围: server

编程语言: python

父类: *AbstractItem 类*

描述说明

指定向用户显示的实体项的名称。

另请参见

Task 树

item_name

item_name

使用范围: server

编程语言: python

父类: *AbstractItem 类*

描述说明

指定在代码中引用的实体项的名称。

使用 "item_name" 在代码中引用该实体项。

另请参见

Task 树

item_type

item_type

使用范围: server

编程语言: python

父类: *AbstractItem 类*

描述说明

指定实体项的类型。

使用 `item_type` 属性类获取实体项的类型。

它可以是下列值之一:

- "task"
- "items"
- "details"
- "reports"
- "item"
- "detail_item"
- "report"
- "detail"

另请参见

Task 树

items

items

使用范围: server

编程语言: python

父类: *AbstractItem 类*

描述说明

列举出该实体项拥有的所有实体项。

使用 `items` 属性来访问当前对象拥有的任何一个实体项。

另请参见

Task 树

owner

Indicates the item that owns this item.

owner

使用范围: server

编程语言: python

父类: *AbstractItem 类*

描述说明

使用 `owner` 属性来找到一个实体项的拥有者 (父项)。

另请参见

Task 树

session

session

使用范围: server

编程语言: python

父类: *AbstractItem 类*

描述说明

使用 `session` 属性来访问来自客户端的当前请求的 session 对象。

session 对象是一个字典，它有下列子项：

- `ip` - 用户的 ip 地址
- `user_info` - 包含用户信息的字典
  - `user_id` - 标识用户的 id

- user_name - 用户名
- role_id - 用户角色的 id
- role_name - 用户所属角色的名称

示例

```
def on_open(item, params):
    user_id = item.session['user_info']['user_id']
    if user_id:
        params['__filters'].append(['user_id', item.task.consts.FILTER_EQ, user_id])

def on_apply(item, delta, params):
    user_id = item.session['user_info']['user_id']
    if user_id:
      for d in delta:
          d.edit()
          d.user_id.value = user_id
          d.post()
```

另请参见

服务器端编程

environ

task

task

使用范围: server

编程语言: python

父类: *AbstractItem 类*

描述说明

该属性表示拥有当前实体项的*task 树* 的根节点。

使用 `task` 属性来查找当前实体项所属的*task 树* 的根节点。

另请参见

Task 树

方法

can_view

can_view(*self*)

使用范围: server

编程语言: python

父类: *AbstractItem 类*

描述说明

使用 can_view 方法来判断当前会话中的用户是否能浏览一个数据项的记录，或打印一个报表。

另请参见

角色

session

can_create

can_edit

can_delete

item_by_ID

item_by_ID(*self*, *ID*)

使用范围: server

编程语言: python

父类: *AbstractItem 类*

描述说明

item_by_ID 方法，从当前项开始，在项目的*task 树* 中搜索所有实体项，以查找其*ID* 属性等于 “ID” 参数的实体项。

另请参见

Task 树

7.2.3 Task 类

class Task

使用范围: server

编程语言: python

Task 类被用于创建项目的*Task 树* 的根节点。

下面列举了该类的属性、方法和事件。

该类同样继承了祖先类*AbstractItem 类* 的属性、方法。

属性

app

app

使用范围: server

编程语言: python

父类: *Task 类*

描述说明

返回对 WSGI 应用程序对象 的引用

该框架使用 Werkzeug WSGI Utility Library 。

另请参见

工作流

work_dir

work_dir

使用范围: server

编程语言: python

父类: *Task* 类

描述说明

返回指向项目目录的真实的绝对路径。

另请参见

工作流

方法

check_password_hash

check_password_hash (*self*, *pwhash*, *password*)

使用范围: server

编程语言: python

父类: *Task* 类

描述说明

使用 `check_password_hash` 方法，将用户输入的密码与预先加盐哈希处理后的密码值进行校验比对。

该方法是对 Werkzeug **check_password_hash** 函数的重新封装:: https://werkzeug.palletsprojects.com/en/0.15.x/utils/

示例

```
def on_login(task, login, password, ip, session_uuid):
  users = task.users.copy(handlers=False)
  users.set_where(login=login)
  users.open()
  for u in users:
      if task.check_password_hash(u.password_hash.value, password):
          return {
              'user_id': users.id.value,
```

(续下页)

(接上页)

```
            'user_name': users.name.value,
            'role_id': users.role.value,
            'role_name': users.role.display_text
        }
```

另请参见

generate_password_hash

connect

connect (*self*)

使用范围: server

编程语言: python

父类: *Task 类*

描述说明

使用 `connect` 方法从 SQLAlchemy 连接池中取出一个连接。

该方法的返回值是一个 DBAPI 连接。

当不再需要连接时，开发者必须通过调用连接的 `close` 方法将它放回连接池中。

示例

```
def delete_rec(item, item_id):
  conection = item.task.connect()
  try:
    cursor = conection.cursor()
    cursor.execute('delete from %s where id=%s' % (item.table_name, item_id))
    conection.commit()
  finally:
    conection.close()
```

copy_database

待公布 (TBA) - 自 Jam.py v5 后，有更改

copy_database (*self, dbtype, database=None, user=None, password=None, host=None, port=None, encoding=None, server=None*)

使用范围: server

编程语言: python

父类: *Task 类*

描述说明

当迁移到另一个数据库时，请使用 `copy_database` 方法来复制数据库中的数据。

请参阅如何迁移到另一个数据库

示例

在下面的代码中，当创建项目的任务树后，应用程序从 demo.sqlite 数据库中复制数据到项目的数据库里：

```
from jam.db.db_modules import SQLITE

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

create_connection

create_connection(*self*)

使用范围: server

编程语言: python

父类: *Task 类*

描述说明

使用 `create_connection` 来创建一个到项目数据库的连接。

该方法返回一个新连接。

不再需要一个连接后，开发者必须关闭它。

另请参见

execute

select

create_connection_ex

待公布 (TBA) - 自 Jam.py v5 后，有更改

create_connection_ex(*self*, *db_module*, *database*, *user=None*, *password=None*, *host=None*, *port=None*, *encoding=None*, *server=None*)

使用范围: server

编程语言: python

父类: *Task 类*

描述说明

使用 `create_connection_ex` 来创建一个到其它数据库的连接。

该方法返回一个新连接。

不再需要一个连接后，开发者必须关闭它。

另请参见

如何使用来自其他数据库的数据表

execute

execute(*self*, *sql*)

使用范围: server

编程语言: python

父类: *Task 类*

描述说明

通过多进程连接池，使用 `execute` 方法来执行一条 SQL 查询语句（不包括 SELECT 查询）。*select* 方法适用于 SELECT 查询语句。

`sql` 参数可以是一个查询字符串、一个由查询字符串组成的列表、一个由列表组成的列表等。

在一个事务中执行所有查询。如果都执行成功，则调用 `COMMIT` 命令；否者，执行 `ROLLBACK` 命令

示例

```
sql = []
for i in ids:
    sql.append('UPDATE DEMO_CUSTOMERS SET QUANTITY=2 WHERE ID=%s' % i)
item.task.execute(sql)
```

另请参见

select

generate_password_hash

generate_password_hash(*self*, *password*, *method='pbkdf2:sha256'*, *salt_length=8*)

使用范围: server

编程语言: python

父类: *Task 类*

描述说明

该方法使用指定的加密算法对密码进行哈希处理，并使用指定长度的随机字符串作为盐值进行加密。返回的字符串格式中包含了所使用的加密算法，以便*check_password_hash* 方法能够校验该哈希值。

该方法是对 Werkzeug **generate_password_hash** 函数的重新封装: https://werkzeug.palletsprojects.com/en/0.15.x/utils/

示例

```
def on_apply(item, delta, params, connection):
    for d in delta:
        if d.password.value:
            d.edit();
            d.password_hash.value = delta.task.generate_password_hash(d.password.value)
            d.password.value = None
            d.post();
```

另请参见

check_password_hash

lock

lock (*self*, *lock_name*, *timeout=-1*)

使用范围: server

编程语言: python

父类: *Task 类*

描述说明

使用 `lock` 在 Python 中实现独立于平台的文件锁，这提供了一种简单的进程间通信方式。

此方法对 Python 文件锁库的重新封装：https://github.com/benediktschmitt/py-filelock

一旦获取了锁，后续获取它的尝试就会被阻止执行，直到它被释放。

`lock_name` 参数是"锁 (lock)"的名称，在应用程序内它必须是唯一的。filelock 库在 *locks* 文件夹中创建一个具有此名称和 *.lock* 扩展名的文件，用于实现"锁 (lock)"。

`timeout` 参数 - 如果在超时秒内无法获取锁，则会引发 Timeout 异常。

示例

```
def calculate(item):
    lock = item.task.lock('calculation'):
    lock.acquire()
    try:
        #some code
    finally:
        lock.release()
```

上面的代码与下面的效果一样：

```
def calculate(item):
    with item.task.lock('calculation'):
        #some code
```

带有 timeout 的示例:

```
from jam.third_party.filelock import Timeout

def calculate(item):
  try
    with item.task.lock('calculation', timeout=10):
        #some code
  except Timeout:
    print("Another instance of this application currently holds the lock.")
```

在以下示例中，保存发票时，应用程序会计算已售出的曲目。在执行此操作之前，它会获取一个锁：

```
def on_apply(item, delta, params):
    with item.task.lock('invoice_saved'):
```

(续下页)

(接上页)

```
        tracks_sql = []
        delta.update_deleted()
        for d in delta:
            for t in d.invoice_table:
                if t.rec_inserted():
                    sql = "UPDATE DEMO_TRACKS SET TRACKS_SOLD = COALESCE(TRACKS_SOLD, 0) + \
                    %s WHERE ID = %s" % \
                    (t.quantity.value, t.track.value)
                elif t.rec_deleted():
                    sql = "UPDATE DEMO_TRACKS SET TRACKS_SOLD = COALESCE(TRACKS_SOLD, 0) - \
                    (SELECT QUANTITY FROM DEMO_INVOICE_TABLE WHERE ID=%s) WHERE ID = %s" % \
                    (t.id.value, t.track.value)
                elif t.rec_modified():
                    sql = "UPDATE DEMO_TRACKS SET TRACKS_SOLD = COALESCE(TRACKS_SOLD, 0) - \
                    (SELECT QUANTITY FROM DEMO_INVOICE_TABLE WHERE ID=%s) + %s WHERE ID = %s" %
↪\

                    (t.id.value, t.quantity.value, t.track.value)
                tracks_sql.append(sql)
        sql = delta.apply_sql()
        return item.task.execute(tracks_sql + [sql])
```

redirect

redirect (*self*, *location*, *code=302*, *Response=None*)

使用范围: server

编程语言: python

父类: *Task* 类

描述说明

使用 Werkzeug `redirect` 对象将客户端重定向到目标位置。

示例

```
task.redirect('/login.html')
```

另请参见

serve_page

select

select (*self*, *sql*)

使用范围: server

编程语言: python

父类: *Task* 类

描述说明

使用 select 方法来执行 SELECT SQL 查询语句。为了执行查询，使用了连接池。

sql 参数是一个要执行的查询语句。

该方法返回由记录组成的一个列表。

示例

```
recs = item.task.execute_select("SELECT * FROM DEMO_CUSTOMERS WHERE ID=41")
for r in rec:
    print(r)
```

另请参见

execute

serve_page

serve_page(*self*, *file_name*, *dic=None*)

使用范围: server

编程语言: python

父类: *Task* 类

描述说明

使用 serve_page 方法向客户端发送一个 html 页面。当前，serve_page 方法使用 Werkzeug Response 对象。

示例

```
task.serve_page('index.html')
```

另请参见

redirect

事件

on_created

on_created(task)

使用范围: server

编程语言: python

父类: *Task* 类

描述说明

在服务端，使用 on_created 事件初始化应用程序。

当项目的*task* 树 刚刚被创建后，会触发该事件。请见工作流

task 参数是指向*task 树* 的一个引用。

备注

在这个事件处理程序里的代码的执行时间必须非常短，因为这对最终用户的体验有很大影响。

示例

```
def on_created(task):
    # some code
```

另请参见

工作流

Task 树

on_ext_request

on_ext_request(task, request, params)

使用范围: server

编程语言: python

父类: *Task 类*

描述说明

使用 on_ext_request 事件将请求发送到服务器中进行处理。

task 参数是指向*task 树* 的一个引用。

request 参数是一个必须以 "/ext" 开始的字符串。可以有一个参数列表。

示例

下面，应用程序会每隔 60 秒向 Demo 应用程序的服务端发送一个请求。

```
#!/usr/bin/env python

try:
    # For Python 3.0 and later
    from urllib.request import urlopen
except ImportError:
    # Fall back to Python 2's urllib2
    from urllib2 import urlopen
import json
import time

def send(url, request, params):
    a = urlopen(url + '/' + request, data=str.encode(json.dumps(params)))
    r = json.loads(a.read().decode())
    return r['result']['data']

if __name__ == '__main__':
```

(续下页)

(接上页)

```
    url = 'http://127.0.0.1:8080/ext'
    while True:
        result = send(url, 'get_sum', [1, 2, 3])
        print(result)
        time.sleep(60)
```

服务端将处理这个请求并返回参数的和。必须在任务服务端模块中声明 `on_ext_request` 方法：

```
def on_ext_request(task, request, params):
    #print request, params
    reqs = request.split('/')
    if reqs[2] == 'get_sum':
        return params[0] + params[1] + params[2]
```

另请参见

on_request

如何创建注册表单窗体

如何处理请求或从其他应用程序及服务获取数据

on_login

on_login(task, form_data, info)

使用范围: server

编程语言: python

父类: *Task 类*

描述说明

通过应用程序构建器的“用户表 (Users table)”，使用 `on_login` 事件重写默认的登录过程，

`task` 参数是指向*task 树* 的一个引用。

`form_data` 是一个字典，里面包含用户在登录表单窗体的输入框里输入的数据。字典的“键”是输入框的 name 属性值。

`info` 参数一个字典，它有下列属性：

- `ip` 是请求的 ip 地址。
- `session_uuid` 是将被创建的会话的唯一标识符（uuid)。

该事件处理程序必须返回有下列属性的字典：

- `user_id` - 用户的唯一表示
- `user_name` - 用户名
- `role_id` - 在角色 中定义的莫格角色的 ID
- `role_name` - 角色名

作为模板的登录表单窗体位于在 Jam.py 的 `login.html` 文件中。你能添加你自己定义的含有输入框的文件，并用 form_data 参数获取它们的值。

在项目文件夹里的自定义 `login.html` 会覆盖默认的登录模板文件。

```
<form action="" method="post">
    <div class="modal-body">
        <div class="my-1 mx-4 row">
            <label for="login" class="col-sm-3 col-form-label">%(login_text)s</label>
            <div class="col-sm-9">
                <input type="text" class="form-control" name="login" id="login" placeholder="
↪%(login_text)s" value="%(login)s" required>
            </div>
        </div>

        <div class="my-1 mx-4 row">
            <label for="password" class="col-sm-3 col-form-label">%(password_text)s</label>
            <div class="col-sm-9">
            <div class="input-group">
                <input type="password" class="form-control" name="password" id="password"␣
↪placeholder="%(password_text)s" value="%(password)s">
                <button class="btn btn-outline" type="button" id="togglePassword">
                    <i class="bi bi-eye-fill"></i>
                </button>
            </div>
            </div>
        </div>
    </div>
    <div class="modal-footer">
        <input class="btn btn-primary px-4" type="submit" value="OK">
    </div>
</form>
```

示例

在此示例中，用户信息存储在项目数据库中的 **Users** 实体项对应的表中：

```
def on_login(task, form_data, info):
    users = task.users.copy(handlers=False)
    users.set_where(login=form_data['login'])
    users.open()
    if users.rec_count == 1:
        if task.check_password_hash(users.password_hash.value, form_data['password']):
            return {
                'user_id': users.id.value,
                'user_name': users.name.value,
                'role_id': users.role.value,
                'role_name': users.role.display_text
            }
```

添加"重置按钮"

要创建一个重置密码选项，像下面那样创建一个自定义的 `login.html` 文件，增加一个自定义的 `reset_pass.html` 文件，并开发 `on_ext_request` 事件逻辑和 `on_request` 路由：

```
<div class="modal-footer">
    <a href="/reset_pass.html" class="button" target="_blank">Forgotten password?</a>
    <button class="btn btn-lg btn-primary w-100 mt-3" type="submit">Login</button>
</div>
```

```
def on_request(task, request):
    parts = request.path.strip('/').split('/')
    if not parts[0]:
        if task.logged_in(request):
            return task.serve_page('index.html')
        else:
            return task.redirect('/login.html')
    .
    .
    elif parts[0] == 'reset_pass.html':
        return task.serve_page('reset_pass.html')

def on_ext_request(task, request, params):
    reqs = request.split('/')

    #reset password
    if reqs[2] == 'reset_pass':
    .
    .
```

另请参见

session

environ

generate_password_hash

check_password_hash

on_logout

on_logout(task, request)

使用范围: server

编程语言: python

父类: *Task 类*

on_request

on_request(task, request)

使用范围: server

编程语言: python

父类: *Task 类*

描述说明

使用 `on_request` 将请求发送到服务端进行处理。

`task` 参数是指向*task 树* 的一个引用。

示例

添加到项目的任务/服务端模块的代码：

```
def on_request(task, request):
    parts = request.path.strip('/').split('/')
    if parts[0] == 'register.html':
        return task.serve_page('register.html')
```

register.html 文件应该存在于应用程序文件夹下。

请求头的使用示例

```
def on_request(task, request):
    from jam.wsgi import Response
    from werkzeug.exceptions import MethodNotAllowed

    parts = request.path.strip('/').split('/')

    if not parts[0]:
        if task.logged_in(request):
            return task.serve_page('index.html')
        else:
            return task.redirect('/login.html')

    elif parts[0] == 'test_test':
        if request.method == 'POST':
            print(request.method )
            print(request.headers)
            api_key = request.headers['Api-key']

            print(api_key)
            return Response('Succesfull test!')
        if request.method == 'GET':
            raise MethodNotAllowed()
```

使用 Curl 命令访问应用程序将返回以下结果：

> **...> curl -X POST "http://127.0.0.1:8080/test_test" -H "api-key: 122448"**
> Successful test!

服务端命令行显示

```
127.0.0.1 - - [10/Dec/2025 11:00:27] "POST /api HTTP/1.1" 200 -
POST
Host: 127.0.0.1:8080
User-Agent: curl/8.16.0
Accept: */*
Api-Key: 122448
Content-Length: 77
Content-Type: application/x-www-form-urlencoded

122448
```

另请参见

serve_page

路由

register.html

7.2.4 Group 类

`class Group`

使用范围: server

编程语言: python

Group 类被用于创建*task 树* 的“组 (group)”对象。

该类同样继承了祖先类*AbstractItem 类* 的属性、方法。

7.2.5 Item 类

`class Item`

使用范围: server

编程语言: python

Item 类被用于创建*task 树* 的实体项对象，它可能有一个与之相关联的数据库数据表。

下面列出了该类的属性、方法和事件。

该类同样继承了祖先类*AbstractItem 类* 的属性、方法。

属性和特性

active

`active`

使用范围: server

编程语言: python

父类: *Item 类*

描述说明

由该属性能确定一个实体项的数据集是否已打开。

使用只读的 `active` 属性来确定一个实体项的数据集是否已打开。

open 方法会将 `active` 的值改为 `true` 。*close* 方法会将 `active` 的值改为 `false` 。

当数据集处于打开状态时，可以导航它的记录，也可以修改记录并将变更保存到实体项对应的数据表中。

另请参见

数据集

导航数据集

修改数据集

details

details

使用范围: server

编程语言: python

父类: *Item 类*

描述说明

该属性列举出一个实体项的所有明细项 对象。

另请参见

明细项

fields

fields

使用范围: server

编程语言: python

父类: *Item 类*

描述说明

该属性列举出一个实体项数据集的所有字段 对象。

示例

```
def customer_fields(customers):
    customers.open(limit=1)
    for f in customers.fields:
        print f.field_caption, f.display_text
```

另请参见

字段

Field 类

filters

filters

使用范围: server

编程语言: python

父类: *Item 类*

描述说明

该属性列举出一个实体项数据集的所有过滤器 对象。

示例

```
def invoices_filters(invoices):
    for f in invoices.filters:
        print f.filter_name, f.value
```

另请参见

过滤器

Filter 类

item_state

item_state

使用范围: server

编程语言: python

父类: *Item* 类

描述说明

检查 `item_state` 属性来确定实体项当前所处的操作模式。Item_state 决定了可以多实体项数据集进行那些操作，例如编辑已有的记录或者插入新记录。`item_state` 在应用程序处理数据时不断变化。

打开一个实体项将使其由"(不活动)inactive"状态变为"浏览 (browse)"状态。应用程序通过调用*edit* 方法可将一个实体置于"编辑 (edit)"状态，调用*insert* 或*append* 方法将一个实体项置于"插入 (insert)"状态。

提交或取消编辑、插入、删除操作，会使实体项的 `item_state` 从当前状体变为"浏览 (browse)"状态。关闭一个数据集，会将其状态置于"(不活动)inactive"状体。

要检查 item_state 的值，请使用下列方法：

- *is_new* - 指明实体项是否处于插入状体
- *is_edited* - 指明实体项是否处于编辑状体
- *is_changing* - 指明实体项是否处于插入或编辑状体

item_state 的值可以是：

- 0 - 非活动状态 (inactive state)
- 1 - 浏览状态 (browse state)
- 2 - 插入状态 (insert state)
- 3 - 编辑状态 (edit state)
- 4 - 删除状态 (delete state)

实体项的*task* 属性的 consts 属性有下列常量对象：

- "STATE_INACTIVE": 0,
- "STATE_BROWSE": 1,

- ”STATE_INSERT”: 2,
- ”STATE_EDIT”: 3,
- ”STATE_DELETE”: 4

so if the item is in edit state can be checked the following way: 因此，可以通过以下方法检查实体项是否处于编辑状态：

```
item.item_state == 2
```

或者:

```
item.item_state == item.task.consts.STATE_INSERT
```

或者:

```
item.is_new()
```

另请参见

修改数据集

log_changes

log_changes

使用范围: server

编程语言: python

父类: *Item* 类

描述说明

指明是否记录数据变更。

使用 log_changes 来控制是否记录对实体项数据集中数据的更改。当 log_changes 是 true （默认值）时，将记录所有变更。稍后，可通过调用*apply* 方法，可将这些变更应用到程序的服务端。

当 log_changes 是 false 时，不会记录数据的变更，也不能将变更应用到程序的服务端。

另请参见

修改数据集

apply

rec_no

rec_no

使用范围: server

编程语言: python

父类: *Item* 类

描述说明

检查 `rec_no` 属性来确定实体项数据集中当前记录的记录号。

可以将 `rec_no` 设置为一个指定的记录号，以将数据游标定位到那条记录上。

另请参见

数据集

导航数据集

table_name

table_name

使用范围: server

编程语言: python

父类: *Item 类*

描述说明

读取这个属性，以获得项目数据库中相应数据表的名称。

virtual_table

virtual_table

使用范围: server

编程语言: python

父类: *Item 类*

描述说明

使用只读的 `virtual_table` 属性，来确定实体项在项目的数据库中是否有一个相应的数据表。

如果 `virtual_table` 的值是 `True`，那么在项目的数据库中，就没有与之对应的数据表。你可以使用这样的实体项来处理内存中的数据集，或者使用它的模块编写代码。调用*open* 方法会创建一个空的数据集，而调用*apply* 方法不起任何作用。

方法

append

append (*self*)

使用范围: server

编程语言: python

父类: *Item 类*

描述说明

在数据集的末尾，打开一个新的空记录。

调用 `append` 方法之后，应用程序允许用户在记录的字段中输入数据，也允许用户稍后使用*post* 方法将这些变更提交到实体项的数据集中，允许使用*apply* 方法将变更应用到实体项在数据库中对应的数据表中。

`append` 方法：

- 检查实体项数据集是否处于*active* 状态。如果不是，则抛出异常。
- 如果实体项是一个明细项 ，就会检查其主实体项是否处于"编辑"或"插入"*state* ，如果不是，则抛出异常。
- 如果实体项不是一个明细项 ，就会检查其主实体项是否处于"浏览"*state* ，如果不是，则抛出异常。
- 在数据集的末尾，打开一个新的空记录。
- 将实体项置于"插入"*state* 。

另请参见

修改数据集

apply

apply(self, connection=None, params=None, safe=False):

使用范围: server

编程语言: python

父类: *Item 类*

描述说明

将所有已更新、已插入、已删除的记录，从实体项数据集中写入到服务端的数据库中。

`apply` 方法：

- 检查实体项是否是明细项，如果是，则返回（明细项的主控表将会保存其更改）
- 检查实体项是否处于"编辑"或"插入"*state* ，如果是，则提交记录。
- 检查变更日志是否有变更，如果没有，则返回。
- 触发为实体项已定义的 `on_before_apply` 事件处理程序。
- 如果 `connection` 参数是 `None` ，将调用任务的*connect* 方法从任务连接池中获取一个连接。
- 如果为任务定义了 `on_apply` 事件处理程序，则执行它。
- 如果为实体项定义了*on_apply* 事件处理程序，则执行它。
- 使用数据库连接，生成并执行 SQL 查询语句，以向数据库写入更改。
- 如果未指定 `connection` 参数，那么将变更提交到数据库，并将连接归还至连接池。
- 在将变更写入数据库之后，更新变更日志和实体项数据集 - 更新新记录的主键值
- 触发为实体项已定义的 `on_after_apply` 事件处理程序。

参数

- connection - 如果指定了该参数，应用程序将使用它来执行生成的 SQL 查询（不会提交变更，也不会关闭连接）；否则，应用程序会从任务连接池获取一个连接，并在提交变更后将该连接归还给连接池。
- params - 使用该参数传递用户的自定义选项，供*on_apply* 事件处理器使用。此参数必须是一个键值对对象。
- safe - 如果设置为 True，该方法会检查调用此方法的用户是否具备在当前项的数据库表中创建、编辑或删除记录的权限（即将执行此类操作之前）；若用户无此权限，则抛出异常。该参数的默认值为 False 。参考角色

Examples

下面的第二个示例中，在一个事务中保存变更。

```
def change_invoice_date(item, item_id):
    inv = item.copy()
    cust = item.task.customers.copy()
    inv.set_where(id=item_id)
    inv.open()
    if inv.record_count():
        now = datetime.datetime.now()
        cust.set_where(id=inv.customer.value)
        cust.open()

        inv.edit()
        inv.invoice_datetime.value = now
        inv.post()
        inv.apply()

        cust.edit()
        cust.last_action_date.value = now
        cust.post()
        cust.apply()
```

```
def change_invoice_date(item, item_id):
    con = item.task.connect()
    try:
        inv = item.copy()
        cust = item.task.customers.copy()
        inv.set_where(id=item_id)
        inv.open()
        if inv.record_count():
            now = datetime.datetime.now()
            cust.set_where(id=inv.customer.value)
            cust.open()

            inv.edit()
            inv.invoice_datetime.value = now
            inv.post()
            inv.apply(con)

            cust.edit()
            cust.last_action_date.value = now
            cust.post()
            cust.apply(con)
    finally:
```

(续下页)

(接上页)

```
        con.commit()
        con.close()
```

另请参见

修改数据集

bof

bof(*self*)

使用范围: server

编程语言: python

父类: *Item* 类

描述说明

使用 `bof` (beginning of file) 方法来判断游标是否指向了实体项数据集中的第一条记录。如果 `bof` 返回 true，那么明确地说明光标就位于数据集中的第一行。

当应用程序执行以下操作时，`bof` 方法返回 true：

- 打开一个实体项数据集
- 调用实体项的*first* 方法
- 调用实体项的*prior* 方法，且方法执行失败（因为光标已经位于数据集中的第一行上）

在其它情况下，bof 返回 false 。

> **备注**
>
> 如果*eof* 和 `bof` 都返回 true，那么说明实体项数据集是空的。

另请参见

数据集

导航数据集

can_create

can_create(*self*)

使用范围: server

编程语言: python

父类: *AbstractItem* 类

描述说明

使用 `can_create` 方法以确定当前会话的用户是否有权限创建新记录。

示例

```
def send_email(item, selected, subject, mess):
    if not item.can_create():
        raise Exception('You are not allowed to send emails.')
    #code sending email
```

另请参见

角色

session

can_view

can_create

can_edit

can_delete

can_delete

can_delete(*self*)

使用范围: server

编程语言: python

父类: *AbstractItem 类*

描述说明

使用 `can_delete` 方法以确定当前会话的用户是否有权限删除记录。

另请参见

角色

session

can_view

can_create

can_edit

can_edit

can_edit(*self*)

使用范围: server

编程语言: python

父类: *AbstractItem 类*

描述说明

使用 can_edit 方法以确定当前会话的用户是否有权限编辑记录。

另请参见

角色

session

can_view

can_create

can_delete

cancel

cancel(*self*)

使用范围: server

编程语言: python

父类: *Item* 类

描述说明

只要更改没有被提交到实体项的数据集中，就能调用 cancel 方法以撤销对当前记录中一个或多个字段的修改。

Cancel 方法：

- 触发为实体项定义的 on_before_cancel 事件处理程序。
- 如果对编辑了记录，则撤销对当前记录及其明细项的更改；如果添加或插入了新记录，则移除新记录。
- 将实体项置于“浏览”状态
- 触发为实体项定义的 on_after_cancel 事件处理程序。

另请参见

修改数据集

clear_filters

clear_filters(*self*)

使用范围: server

编程语言: python

父类: *Item* 类

描述说明

使用 clear_filters 方法，将实体项的过滤器的值设置为 None 。

另请参见

过滤记录

过滤器

close

close (*self*)

使用范围: server

编程语言: python

父类: *Item* 类

描述说明

调用 `close` 方法以关闭实体项数据集。关闭数据集后，*active* 属性的值是 `false` .

另请参见

数据集

open

copy

copy (*self*, *filters=True*, *details=True*, *handlers=True*)

使用范围: server

编程语言: python

父类: *Item* 类

描述说明

使用 `copy` 方法以创建实体项的一个副本（复制品）。创建的副本不会被添加到*task* 树 中，而且不再需要后，会被 Python 的垃圾回收器销毁。

副本对象的所有属性都按照创建任务树时的状态定义。请见工作流程

该方法可以有下列参数:

- `handlers` - 如果这个参数的值是 `true` ，该实体项在服务端模块中的所有函数和事件处理程序在副本中依然可用。默认值是 `true` 。
- `filters` - 如果这个参数的值是 `true` ，那么将为副本创建过滤器。

  否则，副本没有过滤器。默认值是 `true` 。
- `details` - 如果这个参数的值是 `true` ，那么将为副本创建明细项。

  否则，副本没有明细项。默认值是 `true` 。

示例

```
def on_generate(report):
    cust = report.task.customers.copy()
    cust.open()

    report.print_band('title')

    for c in cust:
        firstname = c.firstname.display_text
        lastname = c.lastname.display_text
        company = c.company.display_text
        country = c.country.display_text
        address = c.address.display_text
        phone = c.phone.display_text
        email = c.email.display_text
        report.print_band('detail', locals())
```

另请参见

Task 树

工作流

delete

delete(*self*)

使用范围: server

编程语言: python

父类: *Item* 类

描述说明

该方法删除当前记录，并将游标定位到下一条记录上。

“delete“方法：

- 检查实体项数据集是否处于*active* 状态。如果不是，则抛出异常。
- 检查实体项数据集是否不为空。如果是空的，则抛出异常。
- 如果实体项是明细项 ，那么检查其主控实体项是否处于“编辑”或“插入”状体 。如果不是，则抛出异常。
- 如果实体项不是明细项 ，那么它否处于“浏览”状体 。如果不是，则抛出异常。
- 将实体项置于“删除”状体 。
- 删除当前活动的记录，并将游标定位到下一条记录上。
- 将实体项置于“浏览”状体

另请参见

修改数据集

edit

edit (*self*)

使用范围: server

编程语言: python

父类: *Item 类*

描述说明

允许数据集中的数据编辑功能。

调用一个 `edit` 方法之后，应用程序就允许用户更改记录的字段中的数据，然后使用*post* 方法将这些更改提交到实体项的数据集中，然后使用*apply* 方法将更改应用到数据库中。

`edit` 方法：

- 检查实体项数据集是否处于“活动”状体。如果不是，则抛出异常。
- 检查实体项数据集是否不为空。如果为空，则抛出异常。
- 检查实体项数据集是否已经处于“编辑”状体。如果是，则返回。
- 如果实体是一个明细项 ，检查其主控项是否处于“编辑”或“插入”状体 。如果不是，则抛出异常。
- 如果实体不是一个明细项 ，检查其主控项是否处于“浏览”状体 。如果不是，则抛出异常。
- 将实体置于“编辑”状体 ，允许应用程序或用户修改记录中的字段。

另请参见

修改数据集

eof

eof (*self*)

使用范围: server

编程语言: python

父类: *Item 类*

描述说明

使用 `eof` (end-of-file) 方法来判断游标是否指向了实体项数据集中的最后一条记录。如果 `eof` 返回 true，那么明确地说明光标就位于数据集中的最后一行。

当应用程序执行以下操作时，`eof` 方法返回 true ：

- 打开一个空的数据集
- 调用实体项的*last* 方法
- 调用实体项的*next* 方法，且方法执行失败（因为光标已经位于数据集中的最后一行上）

在其它情况下，eof 返回 false 。

> **备注**
>
> 如果 `eof` 和*bof* 都返回 true，那么说明实体项数据集是空的。

另请参见

数据集

导航数据集

field_by_name

field_by_name(*self*, *field_name*)

使用范围: server

编程语言: python

父类: *Item 类*

描述说明

当只知道字段名称时，可调用 `field_by_name` 方法来分析字段的信息。

`field_name` 参数是已存在的字段的名称。

`field_by_name` 方法返回通过名称指定的字段的字段对象。如果指定的字段不存在，那么 `field_by_name` 方法返回 `None` 。

filter_by_name

filter_by_name(*self*, *filter_name*)

使用范围: server

编程语言: python

父类: *Item 类*

描述说明

当只知道过滤器名称时，可调用“filter_by_name“方法来分析过滤器的信息。

`filter_by_name` 参数是已存在的过滤器的名称。

`filter_by_name`` 方法返回通过名称指定的过滤器的过滤器对象。如果指定的过滤器不存在，那么 `filter_by_name` 方法返回 `None` 。

first

first(*self*)

使用范围: server

编程语言: python

父类: *Item 类*

描述说明

调用 `first` 方法将游标定位到实体项数据集中的第一条记录上，并使其成为活动记录。`first` 方法会提交对活动记录的任何更改。

另请参见

数据集

导航数据集

insert

insert(*self*)

使用范围: server

编程语言: python

父类: *Item* 类

描述说明

在实体项数据集中插入一条新的空记录。

调用 `insert` 方法之后，应用程序允许用户在记录的字段中输入数据，也允许用户稍后使用*post* 方法将这些变更提交到实体项的数据集中，允许使用*apply* 方法将变更应用到实体项在数据库中对应的数据表中。

`insert` 方法：

- 检查实体项数据集是否处于*active* 状态。如果不是，则抛出异常。
- 如果实体项是一个明细项 ，就会检查其主实体项是否处于“编辑”或“插入”*state* ，如果不是，则抛出异常。
- 如果实体项不是一个明细项 ，就会检查其主实体项是否处于“浏览”*state* ，如果不是，则抛出异常。
- 在数据集中，插入一个新的空记录。
- 将实体项置于“插入”*state* 。

另请参见

修改数据集

is_changing

is_changing(*self*)

使用范围: server

编程语言: python

父类: *Item* 类

描述说明

检查实体项是否处于“编辑”或“插入”状态，如果是，则返回 true 。

应用程序调用*edit* 方式会将实体项置于“编辑”状态，调用*append* 或*insert* 方法会将实体项置于“插入”状态。

另请参见

修改数据集

is_edited

is_edited(*self*)

使用范围: server

编程语言: python

父类: *Item 类*

描述说明

该方法检查实体项是否处于“编辑”状体，如果是，则返回 true 。应用程序可调用*edit* 方法将实体项置于“编辑”状体。

另请参见

修改数据集

is_modified

is_modified(*self*)

使用范围: server

编程语言: python

父类: *Item 类*

描述说明

该方法检查实体项数据集的当前记录在编辑或插入操作时是否已被修改。在执行*post* 方法后，该方法返回 `false` 。

另请参见

修改数据集

is_new

is_new(*self*)

使用范围: server

编程语言: python

父类: *Item 类*

描述说明

该方法检查实体项是否处于“插入”状体，如果是，则返回 true 。应用程序可调用*append* 或*insert* 方法将实体项置于“插入辑”状体。

另请参见

修改数据集

last

last (*self*)

使用范围: server

编程语言: python

父类: *Item* 类

描述说明

调用 `last` 方法将游标置于实体项数据集中的最后一条记录上，并使之成为“活动”记录。

另请参见

数据集

导航数据集

locate

locate (*self*, *fields*, *values*)

使用范围: server

编程语言: python

父类: *Item* 类

描述说明

实现一种在实体项数据集中搜索指定记录的方法，并使该记录成为活动记录。

参数：

- `fields`: 一个字段的名称，或多个字段名称组成的列表
- `values`: 字段列表中的字段的值

该方法定位通过 `fields` 参数指定的字段且具有 `values` 参数指定字段对应的值的记录。

如果找到匹配指定条件的记录，那么 `locate` 返回 true，并使游标指向那条记录。

如果未找到匹配的记录，或游标定位失败，那么该方法返回 false 。If a matching record was not found and the cursor is not repositioned, this method returns false.

另请参见

数据集

导航数据集

next

next (*self*)

使用范围: server

编程语言: python

父类: *Item 类*

描述说明

调用 next 方法使游标定位到实体项数据集中的下一条记录上，并使其成为“活动”记录。next 会提交对活动记录的任何更改。

open

open(self, options=None, expanded=None, fields=None, where=None, order_by=None, open_empty=False, params=None, offset=None, limit=None, funcs=None, group_by=None, safe=False)

使用范围: server

编程语言: python

父类: *Item 类*

描述说明

调用 open 方法来生成并执行针对实体项数据库表的 SELECT SQL 查询语句，以获取数据集。

该方法初始化实体项的*fields* 属性, 设置请求的参数，并触发为实体项已定义的 on_before_open 事件处理程序。

如果为实体项定义了*on_open* 事件处理程序，那么 open 方法执行此事件处理程序并将获得的数据集分配给返回的结果。否则，将根据请求的参数生成 SELECT SQL 查询语句，再执行生成的查询语句并将执行的结果分配给数据集。

稍后，将*active* 设置为 true ，将*item_state* 设置为浏览模式，将游标定位到数据集的第一条记录，触发已定义的 on_after_open 事件处理程序。

参数

你可以传入 “options“ 字典，以与客户端*open* 方法完全相同的格式来指定请求参数：

```
invoices.open({
    'fields': ['customer', 'invoicedate', 'total'],
    'where': {customer: customer_id, invoicedate__ge: date1, invoicedate__le: date2},
    'order_by': ['invoicedate']
})
```

或者传递关键字参数:

```
invoices.open(
    fields=['customer', 'invoicedate', 'total'],
    where={customer: customer_id, invoicedate__ge: date1, invoicedate__le: date2},
    order_by=['invoicedate']
)
```

- `expanded` - 如果该参数的值是 true ，那么 SELECT 查询语句将含有获取查找字段 的查找值的 JOIN 子语句，否则结果将不包含查找值。该参数值默认是 `true` 。
- `fields` - 使用该参数来指定 SELECT 查询语句中的可用字段。该参数是一个字段名称的列表。如果省略该参数，将使用*set_fields* 方法定义的字段。如果在"open"方法执行之前，未调用*set_fields* 方法，将使用所有可用字段。
- `where` - 使用该参数指定在 SQL 查询语句中如何过滤记录。该参数是一个字典，其"键"是字段的名称，紧跟双下划线，再加过滤器符号。(请见过滤记录) 如果省略该参数，将使用*set_where* 方法定义的值。如果在 `open` 方法执行之前，未调用*set_where* 方法，且省略了 `where` 参数，那么将使用为实体项定义的过滤器 的值来过滤记录。
- `order_by` - 使用 `order_by` 参数来指定记录的排序顺序。该参数是一个字段名称的列表。如果字段名称前有"-"符号，那么在该字段上会对记录按降序排序。如果省略了该参数，将使用*set_order_by* 方法定义的列表。
- `offset` - 使用 `offset` 参数来指定要获取的第一行记录的偏移。
- `limit` - 使用"limit"参数来限制 SQL 查询语句的输出，只保留输出的前 n 行记录。。
- `funcs` - 该参数可以是一个字典，其"键"是字段的名称，其"值"是函数名称，在 SELECT 查询语句中，将使用这些函数处理对应的字段。
- `group_by` - 使用 `group_by` 参数来指定如何对查询的结果按字段分组。该参数必须是字段名称的列表。
- `open_empty` - 如果将此参数设置为 `true` ，那么应用程序不会向服务端方请求，而是仅仅初始化一个空的数据集。其默认值是 `false` 。
- `params` - 使用此参数传递用户自定义的一些选项，在*on_open* 事件处理程序中将使用这些自定义选项。该参数必须是一个"键-值"对组成的对象。
- `safe` - 如果此参数为 `True` ，将检查调用该方法的用户是否有权限查看实体项的数据。如果用户没有权限，则抛出一个异常。请见角色

Examples

在这个例子中，请求的参数是一个字典：

```
import datetime

def get_sales(item):
    date1 = datetime.datetime.now() - datetime.timedelta(days=3*365)
    date2 = datetime.datetime.now()
    invoices = item.task.invoices.copy()

    invoices.open({
        'fields': ['customer', 'date', 'total'],
        'where': {'date__ge': date1, 'date__le': date2},
        'order_by': ['customer', 'date']
    })
```

在下面，传递的参数是一个关键字列表：

```
import datetime

def get_sales(item):
    date1 = datetime.datetime.now() - datetime.timedelta(days=3*365)
    date2 = datetime.datetime.now()
    invoices = item.task.invoices.copy()

```

(续下页)

(接上页)

```
    invoices.open(
        fields=['customer', 'date', 'total'],
        where={'date__ge': date1, 'date__le': date2},
        order_by=['customer', 'date']
    )
```

分别使用 set_fields、set_where、set_order_by 方法，可以获得与上面相同的结果：

```
import datetime

def get_sales(item):
    date1 = datetime.datetime.now() - datetime.timedelta(days=3*365)
    date2 = datetime.datetime.now()
    invoices = item.task.invoices.copy()

    invoices.set_fields('customer', 'date', 'total')
    invoices.set_where(date__ge=date1, date__le=date2);
    invoices.set_order_by('customer', 'date');
    invoices.open();
```

```
def get_sales(task) {
    sales = task.invoices.copy()

    sales.open(fields=['customer', 'id', 'total'],
        funcs={'id': 'count', 'total': 'sum'},
        group_by=['customer'],
        order_by=['customer'])
}
```

另请参见

数据集

过滤记录

set_fields

set_order_by

set_where

post

post (*self*)

使用范围: server

编程语言: python

父类: *Item* 类

描述说明

将修改后的记录写入到实体项数据集中。在调用*append* 、*insert* 或*edit* 方法之后，调用 post 方法保存对记录的变更。

post 方法：

- 检查实体项是否处于“编辑”或“插入”状体 。如果不是，则抛出异常。
- 触发为实体项定义的 `on_before_post` 事件处理程序。
- 检查对应的记录是否有效，如果无效，则抛出异常。
- 如果实体项有*details* ，那么提交明细项中的当前记录。
- 将变更添加到实体项的变更日志中。
- 将实体项置于“浏览”*state* 。
- 触发为实体项定义的 `on_after_post` 事件处理程序。

另请参见

修改数据集

prior

prior(*self*)

使用范围: server

编程语言: python

父类: *Item* 类

描述说明

调用 `prior` 方法使游标定位到实体项数据集中的前一条记录上，并使其成为“活动”记录。`prior` 会提交对活动记录的任何更改。

另请参见

数据集

导航数据集

record_count

record_count()

使用范围: server

编程语言: python

父类: *Item* 类

描述说明

调用 `record_count` 方法以获取实体项数据集拥有的记录的总数

示例

```
item.open()
if item.record_count():
    # some code
```

另请参见

数据集

open

set_fields

set_fields(*self*, *lst=None*, **fields*)

使用范围: server

编程语言: python

父类: *Item* 类

描述说明

使用 `set_fields` 方法定义并存储内部存储 `fields` 参数，未指定 `fields` 参数的*open* 方法将使用这些内部参数。*open* 方法会清除内部存储的该参数值。

`fields` 是字段名的任意参数列表

参数

你可以像客户端的*set_fields* 方法那样，以列表形式指定字段，或者以非关键字参数的形式指定字段。

示例

下列代码片段的执行结果相同：

```
item.open(fields=['id', 'invoicedate'])
```

```
item.set_fields('id', 'invoicedate')
item.open()
```

```
item.set_fields(['id', 'invoicedate'])
item.open()
```

另请参见

数据集

open

set_order_by

set_order_by(*self*, *lst=None*, **fields*)

使用范围: server

编程语言: python

父类: *Item* 类

描述说明

使用 `set_order_by` 方法定义并存储内部存储 `order_by` 参数，未指定 `order_by` 参数的*open* 方法将使用这些内部参数。*open* 方法会清除内部存储的该参数值。

参数

你可以像客户端的*set_order_by* 方法那样，以列表形式指定字段，或者以非关键字参数的形式指定字段。

如果在一个字段名前有 “-”，那么对记录在该字段上按降序排序。

示例

下列代码片段的执行结果相同：

```
item.open(order_by=['customer', '-invoicedate'])
```

```
item.set_order_by('customer', '-invoicedate')
item.open();
```

```
item.set_order_by(['customer', '-invoicedate'])
item.open();
```

另请参见

数据集

open

set_where

set_where (*self*, *dic=None*, ***fields*)

使用范围: server

编程语言: python

父类: *Item 类*

描述说明

Use the `set_where` method to define and store internally the where filters that will be used by the *open* method, when its own where parameter is not specified. The *open* method clears internally stored parameter value.

参数

你能够以字典的形式指定过滤器，就像客户端上在*set_where* 一样，或是使用关键字参数指定过滤器。

示例

下列代码片段的执行结果相同：

```
import datetime

date = datetime.datetime.now() - datetime.timedelta(days=3*365)
item.open(where={'customer': 44, 'invoicedate__gt': date})
```

```
import datetime

date = datetime.datetime.now() - datetime.timedelta(days=3*365)
item.set_where({'customer': 44, 'invoicedate__gt': date})
item.open()
```

```
import datetime

date = datetime.datetime.now() - datetime.timedelta(days=3*365)
item.set_where(customer=44, invoicedate__gt=date)
item.open()
```

另请参见

数据集

open

事件

on_after_apply_record

on_after_apply_record(self, delta, params, connection)

使用范围: server

编程语言: python

父类: *Item* 类

描述说明

当你需要在*client* 或*server* 上的 apply 方法执行之后重写标准的数据保存过程时，请编写 `on_after_apply_record` 事件处理程序。

在演示应用程序中，使用该事件来重写计算“发票 (Invoices)”。

“发票 (Invoice)”在“服务器模块”中的代码如下：

```
def on_after_apply_record(item, delta, params, connection):
    if not delta.rec_deleted():
        calc_invoice(delta, connection)
```

另请参见

服务器端编程

on_after_before_record events

on_apply events

修改数据集

on_after_generate

请见:

Task 类

on_after_open

on_after_open(self, delta, params, connection)

使用范围: server

编程语言: python

父类: *Item 类*

描述说明

`on_after_open` 事件处理器会在 SQL 请求执行完毕后触发，并以数据集作为参数。你可以在将数据集发送给客户端之前对其进行修改。

另请参见

服务器端编程

on_apply events

修改数据集

on_apply

on_apply(self, delta, params, connection)

使用范围: server

编程语言: python

父类: *Item 类*

描述说明

在 on_apply 事件前后的事件，请参阅 doc:*verson 7 </releases/version_7/index>* 中的 `on_before_apply_record` 和 `on_after_apply_record` 。

当你需要在*client* 或*server* 上的 apply 方法执行中重写标准的数据保存过程时，请编写 `on_apply` 事件处理程序。

请参阅*on_apply events* 来理解如何触发 `on_apply` 事件。

`on_apply` 事件处理程序有下列参数：

- `delta` - 一个包含数据项变更日志的增量（delta）数据（下文将详细讨论）
- `params` - 通过 apply 方法传递到服务端的参数
- `connection` - 用于将更改保存到数据库的连接

必须将 `delta` 参数包含的变更保存在数据库里。

就其本身而言，此选项是实体项的副本，其数据集是项目的更改日志。记录更改的性质可以通过以下方法获得：

- `rec_inserted`
- `rec_modified`
- `rec_deleted`

如果添加了、修改了、删除了记录，它们将分别返回 True

如果实体项有一个明细项，那么 delta 参数也有一个与之对应的明细项，用来存储实体明细项的更改。

备注

请注意，当从实体项中删除一个含有明细项记录的记录时，变更日志将仅仅保存那个被删除的主记录，不会存储明细项中被删除的记录信息。要添加这些被删除的明细项的记录，请调用 delta 的 update_deleted 方法。

光标移动到另一条记录后，您不需要打开增量（delta）明细信息。

增量（delta）数据集的字段含有一个 on_apply 属性，此属性能用来获取发生变更之前时字段的值。

如果没有定义 on_apply 事件处理程序，那么将执行 apply_delta 方法来生成并执行 SQL 查询语句。之后，将返回相关的处理结果信息，其中也存储着新记录的 id 。客户端根据这些信息更新实体项的变更日志和新记录的主键字段的值。

当 on_apply 事件处理程序返回 None 时，就会执行 apply_delta 方法。

你能对 delta 进行一些额外的处理过程。在下面的代码中，在变更应用到数据库表中之前，date 字段的值被设置为当前的日期。

```
import datetime

def on_apply(item, delta, params, connection):
    for d in delta:
        d.edit()
        d.date.value = datetime.datetime.now()
        d.post()
```

备注

请注意，使用这种方法产生的变更不会反映在客户端的实体项数据集中。你能够使用客户端的*refresh_record* 或*refresh_page* 方法来显示这些变更。

在 Jam.py V5 的以下代码中，在保存对发票所做的更改时，应用程序会同时更新曲目（tracks）的 tracks_sold 字段值以及当前发票的相关值。所有这些都是在一个事务中完成的。

```
def on_apply(item, delta, params, connection):
    tracks = item.task.tracks.copy()
    changes = {}
    delta.update_deleted()
    for d in delta:
        for t in d.invoice_table:
            if not changes.get(t.track.value):
                changes[t.track.value] = 0
            if t.rec_inserted():
                changes[t.track.value] += t.quantity.value
            elif t.rec_deleted():
                changes[t.track.value] -= t.quantity.value
            elif t.rec_modified():
                changes[t.track.value] += t.quantity.value - t.quantity.old_value
    ids = list(changes.keys())
    tracks.set_where(id__in=ids)
```

（续下页）

(接上页)

```
    tracks.open()
    for t in tracks:
        q = changes.get(t.id.value)
        if q:
            t.edit()
            t.tracks_sold.value += q
            t.post()
    tracks.apply(connection)
```

在前面的示例中，`on_apply` 事件处理程序返回 `None` 。在此之后，应用程序将执行 `apply_delta` 方法。

在 Jam.py V7 的以下代码中，我们执行了与服务器端计算相同且更多的操作。注意明细项的 `calc_track` 函数：

```
def on_apply(item, delta, params, connection):
    if not delta.rec_deleted():
        calc_invoice(delta, connection)

def calc_invoice(delta, connection):
    invoice = task.invoices.copy(handlers=False);
    invoice.set_where(id=delta.id.value)
    invoice.open(connection=connection)
    invoice.subtotal.value = 0
    invoice.tax.value = 0
    invoice.total.value = 0
    invoice.invoice_table.open(connection=connection)
    for t in invoice.invoice_table:
        task.invoice_table.calc_track(t)
        invoice.subtotal.value += t.amount.value
        invoice.tax.value += t.tax.value
        invoice.total.value += t.total.value
    invoice.apply(connection=connection)
    delta.copy_record_fields(invoice, copy_system_fields=True)
```

在发票详细项的 `Server Module` 上，有：

```
def calc_track(item):
    item.amount.value = item.quantity.value * item.unitprice.value
    item.tax.value = item.amount.value * item.master.taxrate.value / 100
    item.total.value = item.amount.value + item.tax.value
    item.paid.value = item.master.paid.value
```

更一般的情况是：

```
def on_apply(item, delta, params, connection):

  # execute some code before changes are written to the database

  result = item.apply_delta(delta, params, connection)

  # execute some code after changes are written to the database

  return result
```

另请参见

服务器端编程

on_apply events

修改数据集

on_before_apply_record

on_before_apply_record(self, delta, params, connection)

使用范围: server

编程语言: python

父类: *Item* 类

描述说明

当你需要在*client* 或*server* 上的 apply 方法执行之后重写标准的数据保存过程时，请编写 `on_before_apply_record` 事件处理程序。

在演示应用程序中，使用该事件检查发票是否已支付，以及是否在发票之外更改了曲目。

“发票 (Invoice)”在“服务器模块”中的代码：

```
def on_before_apply_record(item, delta, params, connection):
    if delta.paid.cur_value:
        item.abort('Operation prohibited. The invoice is paid')
```

请检查演示 `Details\invoice_table\Server module` 中的曲目。

另请参见

服务器端编程

on_after_apply_record events

on_apply events

修改数据集

on_before_generate

请见:

Reports 类

on_before_open

on_before_open(item, query, params, connection)

使用范围: server

编程语言: python

父类: *Item* 类

描述说明

`on_before_open` 事件处理器会在 SQL 请求执行完毕后触发。可以用它来验证请求和添加其他过滤器。

另请参见

服务器端编程

on_apply events

修改数据集

on_convert_report

请见:

Reports 类

on_created

请见:

Task 类

on_generate

请见:

Reports 类

on_login

请见:

Task 类

on_logout

请见:

Task 类

on_open

on_open(item, params)

使用范围: server

编程语言: python

父类: *Item 类*

描述说明

请参考*版本 7* 中的 `on_before_open` 和 `on_after_open` 。

当您需要重写在*client* 或*server* 上执行 open 方法期间从数据集中获取记录的标准过程时,

请编写 `on_open` 、`on_bforee_open` 和 `on_after_open` 事件处理程序。

参阅*on_open_events* 以理解如何触发 on_open 事件。

`on_open` 事件处理程序有如下参数：

- `item` - 实体项的引用
- `params` - 字典，它包含使用 open 方法传递给服务端的参数：
  - `__expanded` - 对应服务端*open* 方法的 `expanded` 参数，或是客户端:doc:*open </refs/client/item/m_open>* 方法的 options 参数中的"expanded"属性。
  - `__fields` - 字段名称的列表
  - `__filters` - 实体项列表。每个实体项都是一个包含以下成员的列表：
    - 字段名称
    - 来自过滤记录 中的过滤器常量
    - 过滤器的值
  - `__funcs` - 函数字典
  - `__order` - 实体项的列表。每个实体项都是一个包含以下成员的列表：
    - 字段名称
    - 逻辑值（ture 或 false ）, 如果为 ture， 则顺序为降序
  - `__offset` - 对应于 open 方法的 offset 参数
  - `__limit` - 对应于 open 方法的 limit 参数
  - `__client_request` - 当请求来自客户端时，其值为 true

  `params` 还可以包括传递给 open 方法的用户定义参数。

以下是发票实体项的客户端 open 方法发送到服务端的参数示例：

```
{
    '__fields': [u'id', u'deleted', u'customer', u'firstname', u'date',
        u'subtotal', u'taxrate', u'tax', u'total',
        u'billing_address', u'billing_city', u'billing_country',
        u'billing_postal_code', u'billing_state'],
    '__filters': [[u'customer', 7, [6]]],
    '__expanded': True,
    '__limit': 11,
    '__offset': 0,
    '__order': [[u'date', True]]
}

{
    '__fields': [u'id'],
    '__funcs': {u'id': u'count'},
    '__filters': [],
    '__expanded': False,
    '__offset': 0,
    '__order': [],
    '__summary': True
}
```

服务端应用程序基于参数生成 SQL 查询并执行它们。

如果在执行过程中发生错误，服务端会向客户端返回结果记录和错误消息。

一个如何使用服务端事件的示例

另请参见

on_open_events

服务器端编程

数据集

on_parsed

请见:

Task 类

on_request

请见:

Task 类

7.2.6 Detail 类

class Detail

使用范围: server

编程语言: python

Detail 类继承了*Item 类* 的属性、方法和事件。

属性

master

master

使用范围: server

编程语言: python

父类: *Detail 类*

描述说明

使用 `master` 属性来获取对明细项的主表的引用。

另请参见

明细项

7.2.7 Reports 类

class Reports

使用范围: server

编程语言: python

Reports 类被用来创建*task 树* 的 "组 (group)" 对象，该对象拥有项目的报告。

下面列举了该类的事件。

该类同样继承了祖先类*AbstractItem 类* 的属性、方法。

事件

on_convert_report

on_convert_report(*report*)

使用范围: server

编程语言: python

父类: *Reports 类*

描述说明

框架在内部使用 LibreOffice 来转换报表。如果像使用其它转换服务或修改报表转换的某些参数，请使用 **on_convert_report** 事件。

report 参数是触发该事件的报表。

示例

```
import os
from subprocess import Popen, STDOUT, PIPE

def on_convert_report(report):
    try:
        if os.name == "nt":
            import _winreg
            regpath = "SOFTWARE\\Microsoft\\Windows\\CurrentVersion\\App Paths\\soffice.exe"
            root = _winreg.OpenKey(_winreg.HKEY_LOCAL_MACHINE, regpath)
            s_office = _winreg.QueryValue(root, "")
        else:
            s_office = "soffice"
        convertion = Popen([s_office, '--headless', '--convert-to', report.ext,
            report.report_filename, '--outdir', os.path.join(report.task.work_dir, 'static',
↪'reports') ],
            stderr=STDOUT,stdout=PIPE)
        out, err = convertion.communicate()
        converted = True
    except Exception as e:
        print(e)
```

7.2.8 Report 类

class Report

使用范围: server

编程语言: python

下面列出了该类的属性、方法和事件。

该类同样继承了祖先类*AbstractItem 类* 的属性、方法。

属性

report_filename

report_filename

使用范围: client

编程语言: python

父类: *Report 类*

描述说明

当指定了报表的*template* 属性时，*generate* 方法就会将生成的报表的内容保存到“static”目录的报表文件夹中的一个文件里，并将 **report_filename** 属性设置为已保存文件的文件名。

该属性的值可用在*on_after_generate* 事件处理程序中。

另请参见

报表的服务器端编程

generate

report_url

report_url

使用范围: client

编程语言: python

父类: *Report 类*

描述说明

generate 方法向客户端发送存储在该属性里的值，这个值是已生成文件的 url 。

当指定了报表的*template* 属性时，*generate* 方法会，在保存已生成的内容后，设置该属性的值。否则，开发人员必须自己设置该属性的值。

另请参见

报表的服务器端编程

generate

template

template

使用范围: client

编程语言: python

父类: *Report 类*

描述说明

report_filename 属性里存储了报表模板的一个文件名。通常，在生成报表时，会在应用程序构建器里对其进行设置。但是，在服务端的*on_before_generate* 事件处理程序里，能进行动态更改，或者清空，比如必须创建一些像 **txt** 文件的时候。

另请参见

报表模板

创建报表

报表的服务器端编程

方法

generate

generate(*self*)

使用范围: client

编程语言: python

父类: *Report 类*

描述说明

该方法用于在内部生成报表的内容。

在服务端得到来自客户端生成报表的请求时，首先创建报表的副本，然后副本再调用该方法。

该方法触发*on_before_generate* 事件。

如果定义了报表*template* ，解析模板并触发*on_parsed* 和*on_generate* 事件。

在*on_generate* 事件处理程序里，开发人员应该编写代码来生成报表的内容，并将报表的内容保存在一个 **ods** 文件里，同时使用*print_band* 方法来打印数据区域。

当报表已生成且在客户端设置的报表的*extension* 属性的值不等于 **ods** 时，服务端尝试转换那个 **ods** 文件。

为了转换文件，首先检查报表组（该报表的拥有者）是否有*on_convert_report* 事件处理程序。如果定义了该处理程序，将使用它来转换报表。否则，使用在服务端安装的 **LibreOffice** 以 **无头 (headless)** 模式进行转换。

转换之后，应用程序将生成的报表保存到一个文件中，其位置在 "static" 目录的一个报表文件夹中，将*report_filename* 的值设置为已保存的文件的文件名，生成*report_url* 属性的值，并触发*on_after_generate* 事件。

报表生成后，就会将它保存在 **static** 目录下的 **report** 或 **reports** 文件夹中，然后服务端将报表文件的 url 发送到客户端。

如果没有设置*template* 属性，服务端会触发*on_generate* 事件，然后触发*on_after_generate* 事件。在这种情况下，你应该自己编写代码把已生成的内容保存到一个文件里，并设置*report_url* 属性的值。。

另请参见

报表编程

服务器端编程

hide_columns

hide_columns(*self*, *col_list*)

使用范围: client

编程语言: python

父类: *Report 类*

描述说明

使用 **hide_columns** 方法来隐藏在报表的*template* 中定义的一些列。

col_list 参数指定了应该隐藏哪些列。这是由定义报表列的位置的整数或字母组成的一个列表。

请在*on_parsed* 事件处理程序里使用该方法。

示例

```
def on_parsed(report):
    report.hide_columns(['A', 'C'])
#     report.hide_columns([1, 3])
```

另请参见

报表编程

报表模板

报表的服务器端编程

on_parsed

on_generate

print_band

print_band(*self*, *band*, *dic=None*)

使用范围: client

编程语言: python

父类: *Report 类*

描述说明

使用 `print_band` 方法设置在报表*template* 中定义的区域中的可编程单元格的值，并将该区域添加到报表的内容中。

它有下列参数:

- **band** - 指定要打印的区域的名称
- **dic** - 字典, 包含要分配给区域中的可编程单元格的值。

示例

以下代码生成演示应用程序的 **客户列表 (Customer list)** 报表的内容：

```
def on_generate(report):
    cust = report.task.customers.copy()
    cust.open()

    report.print_band('title')

    for c in cust:
        firstname = c.firstname.display_text
        lastname = c.lastname.display_text
        company = c.company.display_text
        country = c.country.display_text
        address = c.address.display_text
        phone = c.phone.display_text
        email = c.email.display_text
        report.print_band('detail', locals())
```

另请参见

报表编程

报表模板

报表的服务器端编程

generate

on_generate

事件

on_after_generate

on_after_generate(*report*)

使用范围: client

编程语言: python

父类: *Report 类*

描述说明

在报表已被生成且被保存到一个以*report_filename* 属性值命名的文件里时，*generate* 方法触发 **on_after_generate** 事件。

report 参数是触发该事件的报表。

另请参见

报表编程

generate

on_before_generate

`on_before_generate`(*report*)

使用范围: client

编程语言: python

父类: *Report 类*

描述说明

在生成报表之前，*generate* 方法会触发 **on_before_generate** 事件。

report 参数是触发该事件的报表。

另请参见

报表编程

generate

on_generate

`on_generate`(*report*)

使用范围: client

编程语言: python

父类: *Report 类*

描述说明

on_generate 事件由*generate* 方法触发。

编写 **on_generate** 事件处理程序来生成报表的内容。使用*print_band* 方法来打印在报表的模板 中定义的区域。

report 参数是触发该事件的报表。

另请参见

报表编程

报表的服务器端编程

报表模板

generate

on_parsed

`on_parsed`(*report*)

使用范围: client

编程语言: python

父类: *Report 类*

描述说明

在报表的模板 被解析之后，*generate* 方法会触发 **on_parsed** 事件。

使用该事件处理程序，通过调用*hide_columns* 方法来隐藏在报表模板中定义的一些列。

report 参数是触发该事件的报表。

另请参见

报表编程

报表的服务器端编程

报表模板

hide_columns

7.2.9 Field 类

class Field

使用范围: server

编程语言: python

属性和特性

display_text

display_text

使用范围: server

编程语言: python

Field 类

描述说明

将字段的值显示为字符串。

`Display_text` 属性是一个只读字符串，它是字段的值面向用户的显示形式。如果指派了一个*on_field_get_text* 事件，那么 **display_text** 是这个事件处理程序的返回值。否则，display_text 的值是查找字段 的*lookup_text* 属性和为其它字段设置的语言区域设置 的*text* 属性。

当字段不处于编状态时，`Display_text` 是字段的值属性的字符串表示形式。当字段处于编状态时，将使用*text* 属性表示字段。

示例

```
def on_generate(report):
    cust = report.task.customers.copy()
    cust.open()

    report.print_band('title')

    for c in cust:
        firstname = c.firstname.display_text
```

(续下页)

(接上页)

```
        lastname = c.lastname.display_text
        company = c.company.display_text
        country = c.country.display_text
        address = c.address.display_text
        phone = c.phone.display_text
        email = c.email.display_text
        report.print_band('detail', locals())
```

另请参见

字段

查找字段

on_field_get_text

text

lookup_text

field_caption

field_caption

使用范围: server

编程语言: python

Field 类

描述说明

`Field_caption` 属性指定了向用户显示的字段的名称。

另请参见

数据集

字段

field_name

field_name

field_name

使用范围: server

编程语言: python

Field 类

描述说明

指定在代码中引用的字段的名称。使用 `field_name` 来引用代码中的字段。

另请参见

数据集

字段

field_caption

field_size

field_size

使用范围: server

编程语言: python

Field 类

描述说明

指明文本字段对象的大小。

另请参见

数据集

字段

field_type

field_type

使用范围: server

编程语言: python

Field 类

描述说明

指定字段对象的数据类型。

使用 `field_type` 属性了解字段包含的数据类型。

它是下列值之一:

- ”text”,
- ”integer”,
- ”float”,
- ”currency”,
- ”date”,
- ”datetime”,
- ”boolean”,
- ”blob”

另请参见

数据集

字段

lookup_text

lookup_text

使用范围: server

编程语言: python

Field 类

描述说明

使用 `lookup_text` 属性来获取转换为字符串的查找字段 的查找文本。

如果字段是查找字段 ，则得到是它的查找文本（lookup text），否则，得到的是*text* 的属性值。

另请参见

字段

查找字段

lookup_value

text

lookup_value

lookup_value

使用范围: server

编程语言: python

Field 类

描述说明

使用 `lookup_value` 属性来获取的查找字段 的查找值。

如果字段是查找字段 ，则得到是它的查找值（lookup value），否则，得到的是*value* 的属性值。

另请参见

字段

查找字段

lookup_value

lookup_text

owner

owner

使用范围: server

编程语言: python

Field 类

描述说明

指明一个字段对象所属的实体项。

检查 owner 属性的值，以确定使用当前字段对象表示其字段之一的实体项。

另请参见

数据集

字段

raw_value

raw_value

使用范围: server

编程语言: python

Field 类

描述说明

表示一个字读对象中的数据。

使用 `raw_value` 只读属性直接从实体项数据集读取数据。

像*value* 和*text* 等其它属性将对字段的数据进行转换。

因此，数值字段的*value* 属性，会将 `null` 值转换为 `0` 。

另请参见

字段

value

text

read_only

read_only

使用范围: server

编程语言: python

Field 类

描述说明

确定是否允许在数据感知控件中修改字段。

将 `read_only` 设置为 `true` ，以阻止在数据感知控件中对字段的修改。

另请参见

字段

required

required

required

使用范围: server

编程语言: python

Field 类

描述说明

指定一个字段是否允许空值。

使用 `required` 来确定一个字段是需要一个值，还是可以为空。当设置 `required` 属性为 true ，若试图提交一个 null 值将会抛出一个异常。

另请参见

字段

read_only

text

text

使用范围: server

编程语言: python

Field 类

描述说明

使用 `text` 属性来获取或色湖之字段的文本值（text value）。

获取文本属性值

获取*value* 属性的值，并将其转换为文本。

获取文本属性值

将文本值转换为字段的类型，并将其*value* 属性设置为转换后的值。

另请参见

字段

查找字段

lookup_value

text

lookup_text

value

value

使用范围: server

编程语言: python

Field 类

描述说明

使用 `value` 属性来获取或设置字段的值。

获取值

如果字段的类型是 "integer", "float" 或 "currency" , 那么字段将 `null` 转换为 `0` 作为 `value` 的值；而字段类型是"text" 时，会将 `null``转换为空字符串 ``value` 的值。

对于查找字段，这个属性的值是一个整数，它是查找实体项中的对应记录的 id 字段的值。要获取字段的查找值，请使用*lookup_value* 属性。

设置值

When a new value is assigned, the field checks if the current value is not equal to the new one. If so it 当指定了一个新值，字段将检查其当前值是否不等于新值。如果当前值不等于新值：

- 将新值分配给 `new_value` 属性
- 触发为字段定义的*on_before_field_changed* 事件
- 将 `new_value` 属性的值分配给字段数据，再将其设为 `null`
- 将实体项标记为已修改，所以*is_modified* 方法会返回 `true`
- 触发为字段定义的*on_field_changed* 事件
- 更新数据感知控件的内容。

另请参见

字段

查找字段

lookup_value

text

lookup_text

7.2.10 Filter 类

class Filter

使用范围: server

编程语言: python

属性和特性

filter_name

filter_name

使用范围: server

编程语言: python

父类: *Filter 类*

描述说明

指定在代码中引用的过滤器的名称。使用 `filter_name` 来引用代码中的过滤器。

另请参见

过滤器

数据集

owner

owner

使用范围: server

编程语言: python

父类: *Filter 类*

描述说明

指明一个过滤器对象所属的实体项。

检查 owner 属性的值，以确定使用当前筛选器对象表示其筛选器之一的实体项。

value

value

使用范围: server

编程语言: python

父类: *Filter 类*

描述说明

使用 `value` 属性来获取或设置过滤器的值。

示例

```
function on_view_form_created(item) {
    item.filters.invoicedate1.value = new Date(new Date().setYear(new Date().getFullYear() -
↪1));
}
```

另请参见

过滤器

数据集

7.3 Jam.py 异常

待定（TBA）

Jam.py 会抛出一些自己的异常以及标准的 Python 异常。

7.3.1 Jam.py 核心异常

Language with id %s is not found

exception LanguageNotFound

CHAPTER 8

发行说明

8.1 版本 1

版本 1 旨在基于 GTK 工具包开发数据库桌面应用程序。

8.2 版本 2

在版本 2 中，添加了对使用 web 界面开发数据库应用程序的支持。

8.3 版本 3

在版本 3 中，删除了对基于 GTK Toolkit 的数据库桌面应用程序开发的支持。

8.4 版本 4

In Version 4 the server side was reworked. Web.py library was replaced with werkzeug. Session support was added. 在版本 4 中，服务器端进行了重新设计。Web.py 库被替换为 werkzeug 。已添加会话支持。.. toctree:

```
:maxdepth: 1

version_4_0_70
version_4_0_71
version_4_0_74
version_4_0_78
version_4_0_79
version_4_0_81
version_4_0_84
version_4_0_88
```

8.5 版本 5

8.5.1 版本 5.0.1

库:

- 在 index.html 中，默认字体大小是 14px，现在您可以通过替换来将其更改为 12px 字体，将

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

  为

  ```
  <link href="jam/css/jam12.css" rel="stylesheet">
  ```

- Administrator 已重命名为 Application builder 您可以通过输入来运行它 127.0.0.1:8080/builder.html，127.0.0.1:8080/admin.html 也受支持
- 现在为必填字段添加了星号

  要取消它，请在 project.css 文件中添加

  ```
  .control-label.required:after {
      content: "";
  }
  ```

- 修复了报表参数中查找列表值的选择。

文档:

- 文档的第一个版本已完成
- 已添加新主题:

  refresh_record

  refresh_page

  search

- 已添加 Jam.py 路线图

演示:

- 小字体菜单项已添加到主题菜单

8.5.2 版本 5.1.1

库:

- 现在可以存储用户所做的更改的历史记录。参见保存用户所做更改的历史记录
- 数据集记录的本地过滤已重新设计并发布。参见*Filtered* , *on_filter_record*
- 发布*clone* 方法
- 当数据集为空时，尝试获取或设置字段值时，应用程序现在会抛出异常，。

应用程序构建器:

- **Delete reports after** 属性已添加到项目参数
- 对界面进行了一些更改。

8.5.3 版本 5.2.1

库:

- DBtable 类在 jam.js 中已重新设计。分页器 div 已从表中移除，当表滚动时分页器不会滚动。对于有分页的表，已移除垂直滚动。您可以传递*create_table* 方法两个新选项：
  - summary_fields - 字段名称列表。当指定且实体项分页属性为 true 时，表为数值字段计算总和并在表页脚中显示，对于非数值字段显示记录数。
  - freeze_count - 一个整数值。如果大于 0，它指定第一个列的序号，序号小的那些列将成为冻结的 - 当表滚动时它们不会滚动。
- 修复了插入新记录并按 Escape 键时不取消操作的错误
- 修复了历史记录不保存用户名的错误

演示项目:

- 移除了发票客户端模块中计算表摘要的代码
- 演示应用程序和新项目的任务客户端模块中的 on_view_form_created 事件处理程序代码已更改，以便在删除记录后调用*refresh_page* 方法

8.5.4 版本 5.3.1

库:

- 开发了一套任务的客户端方法，用于处理标签页
  - *init_tabs*
  - *add_tab*
  - *close_tab*
- 表单已重新设计。每个表单现在都有一个在 index.html 文件中声明的带有 modal-header 类的 div。搜索输入和过滤文本的元素已从表单模板中移除，并放置在表单头部。
- *view*, *append_record*, *insert_record* 和*edit_record* 方法已重新设计。如果向这些方法传递了容器参数，并且为容器调用了*init_tabs* 方法，则会创建包含表单的标签页。

  对于现有项目，在任务客户端模块的 on_page_loaded 事件处理程序开头添加行

  ```
  task.init_tabs($("#content"));
  ```

  以便表单在标签页中显示，并为 append_record、insert_record 和 edit_record 方法添加 $("#content") 容器参数。

  如果您不想更改表单的 html 模板，可以添加行

  ```
  task.add_form_borders = false;
  ```

  否则，从表单模板中移除搜索输入和过滤文本的元素（在带有 form-header 类的 div 中，移除它），并将带有 modal-header 类的 div 添加到模板。

演示:

- 演示已重写，以在标签页和无模式编辑表单中显示表单

文档:

- *on_ext_request* 示例已针对 Python 3 进行更正

8.5.5 版本 5.3.3

库:

- 修复了安全模式 (版本 5.3.1 之后)
- 已修复 Postgres 的导入错误已修复
- 在新项目和演示应用程序的任务客户端模块的处理程序 on_page_looaded 事件中添加了任务属性 "edit_form_container" 的定义

```
task.edit_form_container = $("#content"); // 注释此行以具有模态编辑表单
```

8.5.6 版本 5.4.1

应用程序构建器

- 语言属性已添加到项目参数中，以选择项目中使用的语言，并允许添加或编辑语言。
- 界面选项卡已添加到项目参数对话框中
- 按钮"View" 和"Edit" 已重命名为"View form" 和"Edit form"
- "View form" 对话框现在允许设置，除了用于创建表的字段之外，还可以设置数据表格的选项，如设置排序顺序和摘要字段的列。使用"Form" 选项卡设置表单选项，包括将在视图表单中显示的详细信息
- "Edit form" 对话框允许创建选项卡和区域以在编辑表单中显示字段输入。您可以在"Form" 选项卡中设置将在编辑表单中显示和编辑的详细信息

库

- 国际化第一阶段已完成。开发人员可以添加他们的语言
- 表单事件处理已重新设计。请参见 jam.js 中的 _process_event 方法
- 为避免并发问题和内存泄漏，服务器端的任务树现在是不可变的，除非执行 on_created 事件。当您需要在事件处理程序或服务器上的函数中调用 open 方法或更改实体的属性时，必须使用 copy 方法
- 添加了 create_detail_views 方法，允许在编辑表单中编辑详细信息
- Item 类中：添加了 table_options 属性（包含在应用程序中设置的表选项中）
- DBAbstructInput 类：复制、粘贴、Escape 键处理已重新设计
- DBTable 类：提示已重新设计
- 主题已更正
- 许多小更改

演示应用程序

- 主题已移除。您可以在项目参数界面选项卡中设置主题。
- 动态菜单已重新设计

> **备注**
>
> 如果您使用库版本低于 4.3.1 创建项目，请在任务客户端模块的 on_page_loaded 事件处理程序中添加以下行：
>
> ```
> task.old_forms = true;
> ```

对于版本 4.3 的库，请清除目录和日志的客户端模块代码，并将任务的客户端模块替换为演示应用程序或新项目的相应代码。在执行之前制作，请存档备份项目。

8.5.7 版本 5.4.11

库:

- 元数据导入/导出和服务器任务的 copy_database 方法已重新设计，当项目移动到不同类型的数据库时，以兼容不同数据库
- *on_detail_changed* 事件和*calc_summary* 方法已添加
- *alert*, alert_info 和 alert_success 方法已添加
- 修复了 python 3 对 MYSQL、Postgres、Oracle 数据库支持的错误
- 修复了一些与 SQL 查询生成相关的错误
- 修复了 on_login 事件的错误
- 为客户端字段添加了*field_mask* 属性
- 日期输入现在使用掩码
- 添加了客户端任务的 create_menu 方法。
- 尽可能多的代码已从默认代码（和演示项目）移动到库中
- 修复了项目路径中非 ASCII 字符相关的错误

应用程序构建器:

- 修复了键盘快捷键的错误
- 修复了角色的错误
- 可以为详细信息设置权限
- 掩码属性已添加到字段对话框
- 摘要字段属性已添加到视图表单对话框中的详细信息
- 默认搜索字段、详细信息高度属性已添加到视图表单对话框
- 详细信息高度属性已添加到编辑表单对话框
- 修复了一些小错误

备注

要在现有项目中使用掩码，包更新后必须在 index.html 中，在 <script src="jam/js/jam.js"></script> 之前添加以下行:

<script src="jam/js/jquery.maskedinput.js"></script>

8.5.8 版本 5.4.14

Library 库 * add_button 方法已添加

- *select_records* 方法已添加
- *alert* 方法已修复

- bootstrap 主题按钮已更改
- 元数据导入错误已修复 - 数据的更改提交到数据库有异常时不显示错误信息

8.5.9 版本 5.4.15

库:

- 增加对 MS SQL SERVER 的支持
- Jam.py 现在支持删除和更改字段，以及支持 SQLITE 数据库的外部索引。由于 SQLITE 不支持对现有表进行列更改、删除和添加外部索引，你需要使用 Jam.py 创建一个新表，并将旧表中的记录复制到其中。
- 对于 SQLITE 数据库，Jam.py 现在不支持将元数据导入现有项目（其项目项在数据库中有相应的表）。您可以将元数据导入到新项目中。
- BLOB 字段类型重命名为 LONGTEXT，相应的 DB 字段尽可能从 BLOB（如果是）更改为 Long text 类型

应用程序构建器:

- 记录用户操作的历史数据项创建错误已修复
- 修复了创建外部索引的错误

8.5.10 版本 5.4.21

库:

- SQLITE - 已实现不区分大小写的搜索
- 修复 MSSQL 相关错误
- 搜索已重构
- 字段对话框-您可以指定 DATE、DATETIME、BOOLEAN 字段和基于查找列表的查找字段的默认值。当在客户端或服务器上调用元素的 append 或 insert 方法时，这些默认值被分配给字段。当您使用直接 SQL 查询更改表记录时，会使用这些不默认值。
- select_records 已重构
- add_view_button, add_edit_button 方法已添加
- 当用户试图关闭或重新加载页面，并且有一个正在编辑的实体项且其数据已被修改时，应用程序会警告用户。
- 修复了许多杂项错误
- 常见问题解答、应用程序构建器、文档信息已重新编写

8.5.11 版本 5.4.22

库:

- *upload* 方法已重新设计
- 添加图像和文件字段类型 - 教程第二部分文件和图像字段
- 为视图表单对话框 表单选项卡添加了 **Buttons on top** 属性
- *refresh_record* 方法已重新设计, 它可以刷新数据项的明细表
- on_field_get_html 事件已添加

演示应用程序 n:

- 发货单 (Invoices): on_field_get_html 处理程序已添加
- 用户 (Customer): 图像字段"Photo" 已添加
- 唱片 (Tracks): 文件字段"File" 已添加

8.5.12 版本 5.4.23

库:

- *refresh_record* 的相关错误已修复
- 现在，图像和文件字段可以作为查找字段

应用程序构建器:

- 修复了创建新组有关的错误

8.5.13 版本 5.4.24

库:

- 语言支持已重新设计
- 存储在 static/builder 文件夹下的应用程序构建器实体项的图像字段的图片可以导入到元数据文件，也可以导出为元数据文件。
- MSSQL ALTER TABLE 错误已修复

8.5.14 版本 5.4.27

库:

- 现在可以使用从相机捕获图像的选项。请参见字段编辑器对话框 中的 **Capturing image from camera**
- 修复了 Chrome 7 中报表参数顺序的错误。
- **Buttons on top** 属性已添加到编辑表单对话框 的表单选项卡适用于新项目，对于现有项目，从新项目的 index.html 复制带有'default-top-edit' 类的 div 到您的 index.html
- *read_only* 已重新设计
- *on_login* 事件参数已更改，以前的参数在日志中支持警告
- 视图表单中可以有多个详细表
- 详详细表顺序现在可以更改
- 现在使用 Esprima-python 库在服务器上解析 javascript
- 添加了德语翻译
- 修复了各种小错误
- Readme 文件已更改

演示应用程序:

- 唱片目录视图表单显示已售出的唱片。

8.5.15 版本 5.4.29

库:

- Jam.py 现在使用 JQuery 3
- 添加了*lock* 方法
- 添加了*create_connection_ex* 方法
- *edit_record* 方法已重新设计，编辑表单事件在从服务器获取所有数据后触发
- 连接池的连接在不活动一小时后重新创建
- 修复了小错误

文档:

- 在后台，我如何执行某些计算
- 如何使用来自其他数据表的数据

8.5.16 版本 5.4.30

- 修复了在某些系统上创建新项目时与编码相关的错误
- 添加了希腊语语言
- 对于 longtext 类型的字段，当值为 null 时，value 属性现在返回空字符串。
- *select_records* 方法已重新设计。添加了 all_record 参数。如果 all_records 参数设置为 true，则添加所有选定的记录，否则该方法省略现有记录（它们之前已选中）。
- view_form_created 和 edit_form_created 方法已添加到 Task 类（为未来使用保留）
- 用于在任务的 on_view_form_created 事件处理程序中创建表和详细信息表的代码已移动到 jam.js 模块中 Item 类的 create_view_tables 方法
- table_container_class 和 detail_container_class 属性已添加到实体项的 view_options 中，以允许开发人员在实体项的 on_view_form_created 事件处理程序中更改
- inputs_container_class 和 detail_container_class 属性已添加到实体项的 edit_options 中，以允许开发人员在实体项的 on_edit_form_created 事件处理程序中更改
- 在 jam.css 和 jam12.css 中修复了 form-header 和 form-footer 类中 btn 组的左边距
- 修复了一些小错误

8.5.17 版本 5.4.31

- 修复了读取 index.html 文件时的错误。index.html 必须使用 Unicode 编码。
- 更新了德语翻译。
- 修复了演示中打开仪表板时的错误。
- 向演示中添加了 Users 实体项。
- 任务服务器模块中的 on_login 事件处理程序已编写（已注释），该处理程序使用 Users 实体项进行登录，并实现了密码修改功能。取消注释 on_login 以查看其工作方式。说明见 https://groups.google.com/forum/#!topic/jam-py/Obkv5d3yT8A

8.5.18 版本 5.4.36

库:

- 表格经过重新设计，现在支持虚拟滚动
- 修复了一些错误

应用程序构建器:

- 为实体添加了查询

演示应用程序:

- 已实现用户注册

8.5.19 版本 5.4.37

库:

- 现在可以将 Jam.py 部署到 PythonAnywhere 。参见*如何在 PythonAnywhere 上部署项目*
- 现在，可以将项目目录传递给 create_application 函数（jam/wsgi.py 模块）。
- 已从项目*参数* 中移除多进程连接池参数
- 修复了与表单键盘事件处理相关的错误
- 修复了一些错误

文档:

- 已创建*如何...* 部分。该部分将包含可用于快速完成常见任务的代码示例。

8.5.20 版本 5.4.40

库:

- Jam.py 现在使用 SQLAlchemy 连接池
- 当图像字段的 read_only 属性设置时，用户不能通过双击更改图像
- 修复了一些错误

文档:

已向*如何...* 添加部署章节

已添加“如何锁定记录以防止用户同时编辑”的主题

8.5.21 版本 5.4.53

库:

- 更改了*on_login* 事件
- 添加了*generate_password_hash* 和*check_password_hash* 方法
- 已得修复与迁移到 SQLAlchemy 和具有虚拟滚动的表相关的错误
- 修复了含有 numneric 类型字段的表大小调整错误
- 已修复带有冻结的列的表格的错误
- 已修复重命名副本时的详细表错误
- 修复了一些小错误

文档:

- 更改到最新文档
- 已修复如何实现切片
- other algorithm will be used 已删除“如何锁定记录，使用户无法同时编辑”主题 - 将使用其他算法

8.5.22 版本 5.4.54

库:

- 修复了选择数据表导入数据时，MSSQL 出现的 L 错误
- 修改了 delta old_value 属性代码（尚无文档记录）

文档:

- 将认证 添加到了如何... 中

8.5.23 版本 5.4.56

库:

- 记录锁定 已可用
- 修改了 wsgi.py 中的任务创建，以避免出现“project have not been created yet”信息
- 修复了报表参数 display_text 的错误
- `show_hints` 和 `hint_fields` 属性现在可以添加到*table_options* 或*create_table* 方法的 options 参数中。
- *refresh_record* 方法恢复明细记录的位置

文档:

- 表单窗体事件 已重写
- 一些主题已从*Jam.py* 常见问题 移动到如何...

演示应用程序

- 修改了 Invoices 服务器模块中的 on_apply 事件处理程序

8.5.24 版本 5.4.57

库:

- 修复了使用 PostgreSQL、MSSQL 或 Firebird 数据库时的记录锁定 错误
- 要为定义了*on_apply* 事件处理器的项目使用记录锁定，您必须改为：添加连接参数，创建游标并使用游标执行 sql 查询。否则记录锁定将不起作用。

例如，代码

```
def on_apply(item, delta, params):
    tracks_sql = []
    delta.update_deleted()
    for d in delta:
        for t in d.invoice_table:
            if t.rec_inserted():
                sql = "UPDATE DEMO_TRACKS SET TRACKS_SOLD = COALESCE(TRACKS_SOLD, 0) + \
                %s WHERE ID = %s" % \
                (t.quantity.value, t.track.value)
```

(续下页)

(接上页)

```
            elif t.rec_deleted():
                sql = "UPDATE DEMO_TRACKS SET TRACKS_SOLD = COALESCE(TRACKS_SOLD, 0) - \
                (SELECT QUANTITY FROM DEMO_INVOICE_TABLE WHERE ID=%s) WHERE ID = %s" % \
                (t.id.value, t.track.value)
            elif t.rec_modified():
                sql = "UPDATE DEMO_TRACKS SET TRACKS_SOLD = COALESCE(TRACKS_SOLD, 0) - \
                (SELECT QUANTITY FROM DEMO_INVOICE_TABLE WHERE ID=%s) + %s WHERE ID = %s" % \
                (t.id.value, t.quantity.value, t.track.value)
            tracks_sql.append(sql)
    sql = delta.apply_sql()
    return item.task.execute(tracks_sql + [sql])
```

必须更改为

```
def on_apply(item, delta, params, connection):
    with item.task.lock('invoice_saved'):
        cursor = connection.cursor()
        delta.update_deleted()
        for d in delta:
            for t in d.invoice_table:
                if t.rec_inserted():
                    sql = "UPDATE DEMO_TRACKS SET TRACKS_SOLD = COALESCE(TRACKS_SOLD, 0) + \
                    %s WHERE ID = %s" % \
                    (t.quantity.value, t.track.value)
                elif t.rec_deleted():
                    sql = "UPDATE DEMO_TRACKS SET TRACKS_SOLD = COALESCE(TRACKS_SOLD, 0) - \
                    (SELECT QUANTITY FROM DEMO_INVOICE_TABLE WHERE ID=%s) WHERE ID = %s" % \
                    (t.id.value, t.track.value)
                elif t.rec_modified():
                    sql = "UPDATE DEMO_TRACKS SET TRACKS_SOLD = COALESCE(TRACKS_SOLD, 0) - \
                    (SELECT QUANTITY FROM DEMO_INVOICE_TABLE WHERE ID=%s) + %s WHERE ID = %s" %
↪\
                    (t.id.value, t.quantity.value, t.track.value)
                cursor.execute(sql)
```

8.5.25 版本 5.4.60

库：

- 重做了并行进程中运行的 Web 应用程序在元数据更改时的参数同步和任务树重新加载。
- 重做了元数据导入。请参阅 导出/导入元数据
- 创建了在服务器重启时从迁移文件夹导入元数据的能力。请参阅如何将开发结果迁移到生产环境
- 现在可以迁移到另一个数据库。请参阅如何迁移到另一个数据库
- 现在客户端*virtual_table* 和服务器*virtual_table* 上，virtual_table 是只读属性。对于 virtual_table 属性为 true 的实体项，调用 open 方法创建空数据集，调用 apply 方法不执行任何操作。
- 现在，导入表时，virtual_table 属性是只读的。
- 为*table_options* 添加了 title_line_count 选项，指定标题行中显示的文本行数，如果为 0，则行高由标题单元格的内容决定。它可以在应用程序构建器中设置。

8.5.26 版本 5.4.69

库：

- Werkzeug 库升级到版本 0.15.4
- 重写了 common.py 模块，创建了 consts 对象
- 移除了 adm_server.py 模块
- 创建了 admin 文件夹，其中包含模块
  - admin.py - 应用程序构建器服务器端模块
  - task.py - 从 admin.sqlite 数据库加载任务
  - export_metadata.py
  - import_metadata.py
- 添加了 builder 文件夹到包中，该文件夹包含用于创建 Jam.py 应用程序构建器的应用程序构建器项目，请参阅文件夹中的 read.me 文件。
- 任务加载加速
- 重写了元数据导入
- 元数据导入加速
- 添加了*permissions* 属性
- 创建了日志记录（目前正在开发中，尚无文档）
- 客户端和服务器上的 edit 方法现在检查项目状态是否处于编辑模式，如果是则不执行任何操作
- 客户端和服务器上的 round 方法已更正，货币字段的值在分配之前被四舍五入
- 内联编辑现在可用于任何数据项（不仅仅是明细）
- 重做了查找字段、列表字段、日期和日期时间输入的内联编辑，修复了错误
- 修复了表格固定列的错误
- 添加了表格条纹选项
- 现在通过 Ctrl-F 使搜索输入框获取输入焦点，而 Escape 会将焦点返回到表格
- 现在已重绘*enable_controls* 控件，无需调用 update_controls 方法
- 修复了很多错误

应用程序构建器：

- 在应用程序构建器表单标题中添加了指向表单相关文档页面的链接

文档：

- 修复了搜索错误
- 重写了与服务器 on_apply 和 on_open 事件相关的主题
- 添加了新主题如何禁止更改记录

8.5.27 版本 5.4.109

- 字段值的转义清理工作已完成。请参阅清理转义
- 为 TEXT 字段在编辑表单对话框 的 **界面选项卡**中添加了 **TextArea** 属性
- 在编辑表单对话框 的 **Interface Tab** 中添加了 **不转义清理**属性。请参阅清理转义

- 现在，在*编辑表单对话框* 的 **界面选项卡**中，用于 FILE 字段的 **Accept** 属性是必需的。上传的文件在服务器上根据此属性进行检查。
- 在*项目参数* 中添加了 **Upload file extensions** 属性，该属性定义了任务*upload* 方法允许上传的文件类型。
- 为*add_edit_button* 和*add_view_button* 方法添加了 `expanded` 选项。

8.5.28 版本 5.5.4

- 版本号已提升，用以标识从 jam.py 分支为 jam.py-v5。分叉的原因是 Andrew 退出了这个项目。

8.6 版本 7

8.6.1 版本 7.0.39

首次创建了版本 7 的文档

8.6.2 版本 7.0.50

新日期选择器已实现 - https://github.com/stefangabos/Zebra_Datepicker

使用该插件的文档处于 TBA。

或者，请访问上面的 GitHub 页面。

8.6.3 版本 7.0.53

修复相机拍摄错误

8.6.4 版本 7.0.64

添加了事件和模块的链接，以及 ACE 的快捷键按钮。

8.6.5 版本 7.0.69

修复 Boolean 类型的摘要错误

8.6.6 版本 7.0.7x

版本 7.0.70 ，第一个带有 Monaco 编辑器的版本。

版本 7.0.71 ，修复了模态表单关闭的错误。

版本 7.0.72 ，添加了以下 Monaco 编辑器快捷键：

Monaco 快捷键

`CTRL+ALT+SHIFT+W` 关闭编辑器

`CTRL+ALT+←/→` 在编辑器之间移动

使用 Apple 键盘：

> **Monaco 快捷键**
>
> `control+Option+Shift+W` 关闭编辑器。
>
> `Cmd(⌘)+Option+←/→` 在编辑器之间移动。

> **Monaco 快捷键**
>
> 在某些 Monaco 编辑器的实现中，您可以在编辑器的上下文（右键单击）菜单中找到命令面板。

在版本 7.0.81 中，修复了新项目的回溯。演示"view_first: true" 更改为"view_first: false"。

在版本 7.0.82 中，添加了 Databricks 支持。

在版本 7.0.86 中，修复了 Monaco 选项卡关闭的错误。

版本 7 仍在开发中，并接受拉取请求。与 V5 的最大区别是路由支持、Bootstrap 5 和处理详细表，从而实现对移动设备的现代支持。此外，Monaco 编辑器将在应用程序构建器中引入。

以下是与 Jam.py V5 的主要差异，由 Andrew Yushev 于 2022 年 11 月撰写：

> **Jam.py V7**
>
> 现在字段有权限。您可以为角色声明某些字段为隐藏或只读。
>
> 隐藏字段在客户端不可用，并且具有此角色的用户无法从浏览器更改它们。
>
> 只读字段在浏览器中被禁用，如果更改，将不会保存到服务器。
>
> **详细表不再作为特殊类型**。如果项目通过查找字段链接，任何项目都可以是另一个项目的详细表。
>
> 例如，发货单可以是客户的详细表，因为发货单有客户查找字段，其查找项是客户。
>
> 这样，您不需要编写代码来使 invoice_table 成为当前演示中唱片的详细表。
>
> 有基于查找字段的计算字段。
>
> 例如，您可以有一个字段显示唱片记录的已售唱片数量，而无需编写代码。
>
> 支持无限级别的嵌套详细表。
>
> 数据的读取和写入已更改。
>
> `on_open` 和 `on_apply` 事件已弃用。
>
> 取而代之的是，引入了 `on_before_open`、`on_after_open` 和 `on_before_apply_record`、`on_after_apply_record` 事件。
>
> `on_before_open` 事件在执行 sql 请求之前触发，可用于验证请求并添加额外过滤器。`on_after_open` 在执行 sql 请求后触发，并有一个数据集作为参数，可以在发送到客户端之前修改。
>
> `on_before_apply_record` 在执行将记录更改保存到数据库表的 sql 查询之前触发，可用于数据验证和计算，几乎与现在在客户端完成的方式相同。
>
> `on_after_apply_record` 在执行保存记录更改的 sql 查询后触发，主键字段已设置，可用于在同一连接中对数据库进行其他额外更改。
>
> 数据保存后，增量发送到客户端，并在客户端更新服务器上对其所做的所有更改。

对具有详细表的记录的更改在服务器上按以下方式处理：
首先触发主记录的 `on_before_apply_record` 事件。然后对于每个已更改的详细表，触发 `on_before_apply_record` 事件。然后为子详细表更改触发该事件，依此类推。

之后，按相反顺序为子详细表、详细表和主记录触发 `on_after_apply_record` 事件。即使仅对详细表进行了更改，也是如此。也就是说，对文档（记录及其详细信息）的更改作为一个整体保存。处理数据库数据的代码已重写。对于 MSSQL 和 MYSQL，支持替代驱动程序。对于具有指定长度的文本字段的数据库，可以更改文本字段大小。如果新大小更大，则字段长度会更改。否则，字段长度保持不变，但应用程序检查文本长度是否小于大小值。更改记录时，可以省略编辑和 post 方法。它们在内部实现。在构建器中，可以创建实体项的副本（克隆）。可以将实体项移动到其他组。

Jam.py 的概念现在如下： 您可以使用 Jam.py 开发应用程序加载的界面。这些应用程序可以像桌面应用程序一样开发，无论客户端（浏览器调试器）中代码的可用性如何。这是可能的，因为完全控制用户在客户端可以访问哪些数据以及可以对服务器上更新的数据进行哪些更改。这种控制是通过为项目的元素和字段设置角色来实现的，这是使用服务器事件处理程序实现限制的非常简单的方式。

8.7 Jam.py 路线图

我们计划为 Jam.py 添加以下功能：

- 支持可以表示为按钮面板、导航栏、弹出菜单的操作，并简化对键盘事件、国际化以及移动设备的支持。
- 语言支持阶段 2：国际化，支持项目中的多种语言。
- 支持 Bootstrap 4。
- 支持移动设备。
- 开发报表向导，简化报表创建
- 重新设计导入/导出工具：可视化界面，控制更改的合并。
- DBTree 组件修订和文档创建

8.7.1 Andrew 离开 Jam.py

当前 Jam.py 维护者 D. Babic 的笔记：

Q: 由于 Andrew 因健康原因决定离开这个项目，问题是路线图将如何影响项目？

A: 这是一个好问题。Andrew 在 Jam.py 上投入了很多时间和精力，这很棒。我非常感谢他。我会说 Jam.py v5 非常稳定、生产就绪，并且核心代码维护将需要最小的努力。另一方面，v7 已经反映了上述一些路线图要点，主要是在 BS4 和移动设备上。这又是 Andrew 的出色努力。报表向导的想法很棒，基本上是为了消除 LibreOffice Calc 依赖。这里的问题是，有多少用户真正利用了报表的真正力量？不太确定有多少。因此，目前 LibreOffice 将保持原样。

Q: 是否会有新的 Jam.py v5 版本发布？

A: 我不确定 Andrew 是否会参与新的 v5 版本发布。拉取请求将根据需求进行处理。然而，Jam.py 将作为分支发布，这意味着合并将同步到分支，然后新版本将由 GitHub Actions 自动构建为 jam.py-v5

Q: 为什么分支为 jam.py-v5？

A: 因为 GitHub 组织。我们需要项目维护者，他们可以管理 GitHub Actions 和 PyPi。当前的 Jam.py github 页面由 Andrew 拥有，恐怕不支持这一点。因此，新版本将在 PyPi 上作为 jam.py-v5 存在。类似地，v7 分支作为 jam.py-v7，也可在 PyPi 上获得。

Q: 那么 v7 也可用吗？

A: 是的。从版本 7.0.50 开始，由于几乎已完成的文档，它可以用户生产环境。如果 Andrew 能够在 2024 年或以后进行一些调整，那就更好了。我们拭目以待。

Q: v7 文档准备好了吗？

A: 我们还决定将文档从 PyPi 分发中移除，因为工作正在进行中。我们相信用户很少自己构建文档，因此最好将用户指向任何在线文档存储库，这同样由 GitHub Actions 构建。当 v7 文档准备好时，我们也将通过 Actions 发布它们。

Q: Jam.py 的未来是什么？

A: 我看到 Jam.py 有非常光明的未来！最终，我们希望看到 Jam.py 通过 GitHub Actions 构建为 Windows App Store 应用程序。这将为 Windows 用户带来巨大的曝光，并带来更光明的未来。

感谢阅读！

2025 年 6 月