Jam.py Design Tips

Jam.py Team

Apr 16, 2024

CONTENTS

CHAPTER

ONE

JAM.PY APPLICATION DESIGN TIPS

1.1 Introduction

Welcome to Jam.py! If you are new to Jam.py or no-code, low-code or more-code Web application development, this is the place to find some tips about the Jam.py.

Objectives

Installing Python and Jam.py, choosing the database and the Web server and making Application Design decisions.

Audience

Web development enthusiasts or developers, with a limited or no experience with the Web development, or deployment.

Prerequisites

Some Python and JavaScript knowledge is recommended. The general knowledge about the Command Line prompt, and typing the commands is required.

To download this document as a single PDF, please visit:

https://jampy-application-design-tips.readthedocs.io/_/downloads/en/latest/pdf/

The PDF is built instantly after every commit into the repository. Hence, some data on this site might be older than the PDF.

1.2 How is the documentation organised

This Documentation follows official Jam.py documentation, keeping similar concepts and looks:

Development Checklist topic touches on some development principles. We also mention on differences between Django and Jam.py.

Application Design topic discusses Jam.py design terminology, authentication thoughts, etc.

"How-to" guides, here you'll find short answers to "How do I….?" types of questions.

"How was Demo built?" guide, here you'll find my take on how was Demo built.

MS Access migration to Jam.py tips.

Acknowledgements to everyone involved in producing this document or helping with Jam.py.

CHAPTER

TWO

DEVELOPMENT CHECKLIST

Here we discuss some Jam.py Development environment issues and Python ecosystem.

2.1 Development Checklist

This list is inspired by the classic article by Joel Spolsky entitled The Joel Test 12 Steps to Better Code.

2.1.1 Built in Code Editor

For the JS and Python code, Jam.py has a built in Ace Code Editor. The best is to familiarise with the editor shortcuts from here: https://github.com/ajaxorg/ace/wiki/Default-Keyboard-Shortcuts

The most used shortcuts:

Windows/Linux	Mac	Action
Tab	Tab	Indent
Shift-Tab	Shift-Tab	Outdent
Ctrl-/	Command-/	Toggle comment
Ctrl-Shift-/	Command-Shift-/	Toggle block comment

It is absolutely possible to extend the ACE editor for AutoCompletition or Themes.

2.1.2 Python version

Since Jam.py is a JavaScript and Python Web Framework, we need Python installed on the target Operation System. Get the latest version of Python at https://www.python.org/downloads/ or with your operating system's package manager.

Python on Windows

If you are just starting with Jam.py and using Windows, you may find *How to install Jam.py on Windows* useful.

2.1.3 Using Python Virtual Environments

Virtual environment is strongly encouraged for any Python development. When compiling Python from source, we would normally install all binaries in some folder, so we can use different Python versions for different projects.

This is particularly true with the Apache Web server and mod_wsgi module, when the module is compiled against specific Python version, and loaded as per Apache procedure.

Standard Python library is used to create a virtual environment. After creation, the new environment must be "sourced" , and all packages installed with pip:

```
$ python3 -m venv project-name

$ source project-name/bin/activate

$ pip install jam.py
```

If installing from source, for example downloaded zip archive from Github, unzip the archive, navigate to the folder and install with pip:

```
$ pip install .
```

We can have as many virtual environments as needed.

Note: It is advisable to have Production and Development environments.

2.1.4 Using the Source Control

Using Source Control is encouraged with Jam.py Export functionality. Since all Jam.py code is stored in the internal database, it is possible to only Source Control the admin.sqlite database, or it' s content by exporting the tables.

However, for the static files, for example JavaScript libraries, or CSS files, the best approach would be using Export which fetches all objects into one compressed file. It is possible to create an automated Export, for example:

Automated config backup

The Application Builder Export file will not contain the Application database content or structure. Because everything in Jam.py is developed inside the Application Builder, it is not possible to create different branch for the Application only. This is the main difference comparing to Django.

The utility to completely track the application changes, even the database structure, is here:

Storing jam.py Application in GitLab

2.1.5 Unit Testing

Jam.py version higher then v5 is using pytest and Chai/Mocha. It is also possible to use Selenium IDE for the browser.

Please visit posted video Jam.py Application Builder automated testing with pytest and Mocha/Chai, and Simple CRM with jam.py and Selenium IDE in 2.45minutes!.

All necessary tools and libraries for testing should be installed in the same Python virtual environment where Jam.py libraries are.

2.1.6 Continuous Integration (CI)

It is possible to use Continuous Integration as we would with Django.

2.1.7 Generating Documentation

Sphinx is used as an defacto Python standard. It is very simple to start with using what is already provided with Jam.py.

As a bare minimum, the below files and folders are copied from Jam.py Docs directory into a new directory to create new Documentation:

```
conf.py       index.txt  Makefile    README.md
contents.txt  make.bat   prepare.py  set_scale.py

intro:
checklist.txt  index.txt

_static:
favicon.ico  jquery.js

_templates:
jamdocs
```

Modify all files for your preference and build the documentation. For example, the below command will build one **single** html file:

```
$  make singlehtml
```

For more professional look, the **latexpdf** option might be used.

```
$  make latexpdf
```

Note: **Latex** libraries are quite large and the installation is on the OS level. **Sphinx** can be installed inside Python virtual environment.

2.1.8 Limited introduction to the tool

Jam.py is using **jam-project.py** with no options to create a new project, and **server.py** to run the project on default port 8080, optionally providing the port number.

Django is using a few different commands, for example **manage.py** with options, or **django-admin** with options, etc.

Hence, there are no options to run management commands from the code in Jam.py as in Django.

After running the **server.py** command, everything is continued in the Web browser, e.g.:

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

2.1.9 Debugging

Since Jam.py is mostly JavaScript based, most of Debugging work is done via browser debug console. It is possible to debug the Server side Python code with **print** command as usual.

2.1.10 Profiling

It is strongly suggested not to run production Applications with the **server.py** command only. Instead, a proper Web server should be configured. Most of the Web server speed benefits are from compressed content sent from the Web server to the User browser, which is particularly true for CSS and JavaScript files. In addition, Jam.py has a special **static** folder where all images files and documents are remaining, and this files are served by the Web server bypassing the Apache mod_wsgi or similar Python interface for other Web servers. Serving the **static** might be possible with a separate Web server, similar to Django tactics. Jam.py has no utility as *collectstatic* Django command, and rsync might be used in this scenario.

2.1.11 Containers

To use a container with Jam.py is easy. However, the decision has to be made if the Application will use Reports based on LibreOffice (LO), since packaging complete LO might be prohibitive due to a container size. If not using LO, but plain CSV export or other mechanisms for reporting, the container size is small and fast to build. Special consideration needs to be made about hosting the Application database and separate Jam.py admin.sqlite and langs.sqlite internal database.

Please refer to more info in here: Use external database for admin e langs

It is also possible to run Jam.py Application as an Azure Web Application, or AWS Functions, hence server less.

Please refer to more info in here: Azure Deployment

2.2 Choosing the Web Server

Jam.py is providing a lightweight internal Web server for the Development/Testing, just like Django. This means Jam.py is extremely portable. Application can be shipped *as is*, or so called *click-and-run*, with just *jam* folder from Jam.py distribution copied into the Application folder. Of course, Python still needs to be installed locally.

2.2.1 Apache Web Server and `mod_wsgi`

Adapted from Django Docs

Django Docs: If you want to use Jam.py on a production site, use Apache with mod_wsgi. mod_wsgi operates in one of two modes: embedded mode or daemon mode. In embedded mode, mod_wsgi is similar to mod_perl –it embeds Python within Apache and loads Python code into memory when the server starts. Code stays in memory throughout the life of an Apache process, which leads to significant performance gains over other server arrangements. In daemon mode, mod_wsgi spawns an independent daemon process that handles requests. The daemon process can run as a different user then the web server, possibly leading to improved security. The daemon process can be restarted without restarting the entire Apache web server, possibly making refreshing your codebase more seamless. Consult the mod_wsgi documentation to determine which mode is right for your setup. Make sure you have Apache installed with the mod_wsgi module activated. Jam.py will work with any version of Apache that supports mod_wsgi.

If you can' t use mod_wsgi for some reason, fear not: Jam.py supports many other deployment options. It works very well with nginx. Additionally, Jam.py follows the WSGI spec (**PEP 3333**), which allows it to run on a variety of server platforms.

mod_wsgi is tightly coupled with Python and quite often shipped as the default Python version installed on the Operation System. When developing Application in an Python Virtual Environment, for example on the Developer' s computer, it is possible that Python version does not match the *mod_wsgi* version activated in Apache due to Python mismatch. To solve any problems with Python differences, it is suggested to install *mod_wsgi* for Apache from the Virtual Environment which matches the Development/Test environment.

Please visit Apache on Windows mailgroup thread for using Apache with Windows.

2.2.2 IIS Web Server

Using IIS Web with FastCGI is supported. Please visit JamPy deployment on Microsoft IIS for more information.

2.2.3 CPanel

Using CPanel is supported. Please visit Success with cPanel v82.0.12 and Jam.py for more information.

2.3 Choosing the Database

Similar to Django, Jam.py attempts to support as many features as possible for supported databases. However, not all database backends are alike, and Jam.py design decisions were made on which features to support.

As contrary to Django, Jam.py has no models, classes, subclasses and attributes. Since Jam.py is exclusively using Application Builder, there is no code to develop *model* to database table relationship, or table fields specified as *class* attributes.

Adapted from Django Docs

Django Docs: Jam.py supports many different database servers and is officially supported with PostgreSQL, MariaDB, MySQL, MSSQL, Oracle, Firebird, SQLite and SQLite with SQLCipher.

If you are developing a small project or something you don' t plan to deploy in a production environment, SQLite is generally the best option as it doesn' t require running a separate server. However, SQLite has many differences from other databases, so if you are working on something substantial, it' s recommended to develop with the same database that you plan on using in production.

In addition to a database backend, we need to make sure the Python database bindings are installed.

- If using PostgreSQL, the `psycopg2` package is needed.
- If using MySQL or MariaDB, the `MySQLdb` for Python 2.x or `mysqlclient` for Python 3.x is needed, as well as database development files.
- If using MSSQL, the `pymssql` is needed.
- If using Oracle, the cx_Oracle is needed.
- If using SQLCipher, `sqlcipher3-binary` is needed for Linux. There is a standalone DLL for Windows available.

Using MySQL on Windows is supported, please visit MySQL deployment on Windows.

Even though Jam.py supports all databases from the above, there is no guarantee that some specific and/or propriety database functionality is supported. Here we name a few.

2.3.1 Database triggers

Database triggers are specific to the database vendor and Jam.py does not support creation of triggers within the Application Builder. It is absolutely possible to use triggers, however, when moving the Application into the ie. Production, the Application Export file will not contain any information about the triggers and they need to be recreated manually.

2.3.2 Database views

The database views are specific to the vendor too, and Jam.py does not support it for now. That is, it does not support a View creation or modification. It does support the View access though. If the View is needed, we can create an table with the same View name and needed fields, providing the Builder Project Database option is set to "DB manual mode" when doing it.

Same as for database triggers, the Application Export file will not contain any information about the Views.

Very similar to Database Views are Jam.py Virtual Tables. More about the Virtual Table is covered in *Northwind Traders*.

2.3.3 Database indexes

Indexes creation/deletion is supported with Jam.py. The indexes information is stored in the Application Export file if the indexes were created by Application Builder interface. Not all vendor specific index functionality is supported, ie. Oracle *function based* index, etc. The Primary Key creation will result in creating an index. When working with *legacy* databases, meaning the tables were Imported and not created by the Application Builder, Jam.py does not import indexes information. This might lead into lost indexes if moving the Application to different environment by Jam.py Export/Import utility.

2.3.4 Database sequences

The database sequences are supported and Jam.py is providing an interface to use the *sequence generator*. Not all *sequence generators* can be used as this is specific to the database vendor. The Export file does not store the sequence definition, just the name of the sequence used.

2.4 admin.sqlite Database

The Mind Map for admin.sqlite database describes the Jam.py database engine schema. The intention was to quickly find the information needed, as well as the code.

CHAPTER

THREE

APPLICATION DESIGN

Here we discuss some Application Design terminology.

3.1 Getting Started

"I want to put a ding in the universe." Steve Jobs

3.2 Top 5 Questions

Before we dive into more details, let' s answer some simple questions. For sure the below is applicable to many similar products. However, the main difference is that some other products might be "selling" other services. It is a typical "hook, line and sinker" scenario. And fair enough, it' s business after all. The Jam.py does not do that. The source code is yours, and it is very well structured. A joy to dig in, and I think Steve would be pleased. Try it. Give it a go.

3.2.1 1. What is Jam.py?

- *Jam.py is a Rapid Application Development framework. The word rapid should be stressed.*
- Jam.py is a SPA (Single Page Application).

3.2.2 2. Why using Jam.py?

- *If you are already working with the databases, then Jam.py is a no-brainier.*
- *The development tool is free. No need for anything more then a browser.*
- *Leverage the existing skill-set. With some Python and JS experience, you' ll be productive in no time.*
- *Simple infrastructure, especially for developers.*
- *Deployment complexity is greatly reduced. Jam.py itself is only a few megabytes in size.*

3.2.3 3. Why not to use Jam.py?

- *If not using databases, then it probably doesn' t make sense to use Jam.py.*
- *Jam.py within a browser as all JavaScript. This means there are a limited number of html elements on a page.*

3.2.4 4. Does it scale?

Yes. That is absolutely true with containers as well.

3.2.5 5. What can I use it for?

Data-centric application –anything to do with the database, for example reporting, CRUD.

Hope the above five questions sparked some interest. Nothing wrong with comparing Jam.py with for example Oracle Apex.

3.3 Terminology

One of the first thing to understand is the Jam.py terminology: the difference between Catalogs (Catalogues), Journals and Details Groups.

Excellent Article by Marco Fioretti @ Linux Magazine:

https://www.linux-magazine.com/Issues/2020/241/Jam.py

We will reference the above article:

Every Jam.py interface, or project, is structured as a tree of tasks that are organized in groups. You may create custom groups if you like; however, at least for your first Jam.py projects, I strongly suggest that you stick with the default task tree, which consists of four groups called Catalogs, Journals, Details, and Reports.

3.3.1 Catalogs (Catalogues)

Note: *Catalogs are tables that store actual records, like the inventory of a store or the students of some school.*

Catalogs are groups of tables that can exist on its own, as for example in popular Spreadsheets software. Normally, the Spreadsheet is used for automated calculations. Since Jam.py is not a Spreadsheets software, all calculations for table rows is JavaScript code developed by the User/Developer within the Application Builder *Client Module* for a table. When some data is updated, deleted or added to the table, Jam.py can execute calculations or any other server task in the background by the code developed within the *Server Module*. Or, any calculations needed within the browser can be executed by the JavaScript within the *Client Module*.

Once tables are created in Catalogs, they can' t be moved to some other group that easily. There is a special *utility that does that. In a newer Jam.py version, moving the tables is enabled by default, no need for any utility.*

3.3.2 Journals

Note: *Journals store information about events or transactions pertaining to the records in the Catalogs –such as invoices or purchase orders.*

Journals are groups of tables that depend on some other tables, namely Catalogs and Details tables. This are the *Master tables* in database Master-Details scenario, or *General Ledger tables*, for example in Accounting. Journals table should contain at least one Details table, and if it does not, it probably is a Catalogs table and not a Journals table. One Journals table can contain as many Details tables as needed. Journals table is using Catalogs tables as a source for data lookups, for example a *Customer* data like *Name, Surname*, if this data exist in the *Catalogs Customers* table.

The same calculations and location principles apply as for Catalogs.

3.3.3 Details

Note: *Details* store detailed information pertaining to the records in the Journals –such as invoices items details or purchase orders items details.

Details are group of tables used only by Journals tables. This are special tables with extended functionality by the default, since extra fields are added every time a Details table is created. The extra fields are used to identify the **owner** record in the Journals tables. This enables Jam.py to find information much quicker then not using extra fields. It is possible to use Details with no extra fields to find exactly the same information by the code. For example, when Importing tables from some *Legacy system*, there would be no such extra field. The solution is to use code for some otherwise missing functionality, which is enabled automatically if used *native* Jam.py Details.

The same calculations and location principles apply as above.

3.3.4 Reports

Note: *Reports* are based on OpenDocument templates`, more specifically Calcs ods files.

Professional *look* of reports is developed visually in LibreOffice (LO) Calcs. It is similar *look and feel* to any other Spreadsheet software. The LO enables us to use graphics, for example as the Company Logo. It is even possible to insert images from the database itself (with watermark, if needed).

The LibreOffice must be installed on the server OS level which is hosting Jam.py. The containerised LO images do exist on the Internet, if needed to use for the Jam.py container.

3.3.5 Virtual Tables

Virtual Tables are in-memory datasets. There is no corresponding table in the project database for Virtual Table. The Demo application is using it for sending an email to *Customers*. IMO, can be used to replace the need for Database Views, for example when used with the *SELECT* SQL.

More about the Virtual Table is covered in *Northwind Traders*.

3.3.6 Need more Groups?

Any other Group item not covered by the above can be created. For instance, on official Jam.py Demo Application, there is an Item called *Dashboard*, which is a table within the *Analytics* Group. The Dashboard is used for exactly that, presenting some statistics as graphs. Quite similar to any Spreadsheet software, except the JavaScript code is needed to create those graphs. Analytics is just storing the Dashboard as a placeholder, since not related to any other Item Groups.

Note: It is possible to rename any existing Item, or delete or make them invisible when accessing the Application.

3.3.7 Wrapping up

Now that we know the Jam.py terminology, it is good to start thinking about what we want to do. We might completely ignore the default Item Groups and make some new ones. Or, we can have only one single Group and create everything in there. However, that would not be nicely organized, due to:

1. Item Groups are represented by the drop-down menu automatically. Hence, there might be a need for HR, Payroll, General Ledger and Assets Groups, to name a few for a Financial System. The drop-down menu would contain many items (tables), for each Group.
2. Item Groups are Ordered in any order. In each Group, the items are also ordered.
3. All of the above happens with no programming.

3.4 Journal/Detail (or Master/Child) scenario

As mentioned, Details Group is used to quickly identify what the Detail (Child), is for the Journal (Master), tables. However, there is more in that. Jam.py can automatically join two or more tables to Master as Details only if tables are available in Details Group. Hence, if there is no Details group, with no tables in it, Jam.py can not build the Master/Child relation automatically.

3.5 Forms, Buttons and other user interaction items

The Form is a basic type of interaction with the User. All User interaction is based on JavaScript. Which means some knowledge of JS programming might be needed if, and only if, there is a need to extend provided Jam.py functionality out of the box. Forms are automatically created for any table and default buttons are added. The Form *look* depends on the template created in the *index.html* file. However, since Jam.py v5 is a SPA (Single Page Application), all Forms will look alike, if there is only one template defined.

To add a button on some Form presenting the data, please consult the official documentation. The buttons can be disabled, enabled, not visible at all due to User Role, or some other conditions, etc.

3.6 Bulk updates, inserts or delete

Sometimes there might be a need for bulk update, insert or delete. The best approach for this task is to use a direct SQL. There are a few bulk insert and update examples on *Northwind Traders*.

3.7 Authentication Decision

There are a number of Design decisions to make, namely what is the Authentication method the Application will use, the need for Users Registration and password reset, or forgotten password mechanism, default Language or translations, Business Logic and what not.

3.7.1 Built in Authentication

Jam.py has a built in Authentication based on Users and Roles tables within **admin.sqlite** database. This method works fine for smaller Applications or Proof of Concept (PoC). However, for larger implementation it might be needed to extend the Users/Roles table for the Application specific requirements. There might be also a requirement to store the two mentioned tables outside the admin.sqlite database for improved security by using some other database other then SQLite3. To copy Users/Roles from a *built in* database to some other Application (e.g. from Development to Test and Prod), export/import of SYS_USERS and SYS_ROLES tables would be needed. All Administrator accounts accessing the Builder interface use *built in* database.

3.7.2 Non built in Authentication

To extend the Authentication with the Application database and Users/Roles tables is easy. Please consult the official Jam.py Documentation how to do that. The benefits for doing it is the ability to use the database backup or export, which will contain all information specific to the Application, in one or more backup/export files. There is no need to export/import SYS_USERS/SYS_ROLES tables, unless there are a large number of Administrator accounts who are accessing the Builder interface.

With non *built in* Authentication, the Application can, as an option, use a custom Python password hashing within the *Task/Server Module*. It is also possible to develop custom password complexity, or password lifetime, which does not exist with *built in* method.

3.7.3 External Authentication

3.7.4 LDAP (Active Directory) Authentication

LDAP is supported by Jam.py v5. LDAP or Active Directory Authentication is fairly straight forward, and it is possible to use a specific LDAP (AD), branch (or AD Groups), where the Jam.py Users exist. The Application will still use *built in* or *non built in* database for storing Users/Roles.

Note: The Python *Business Logic (BL)* can be developed within the *Task/Server Module*.

For example, if the User does not exist in the Application database, it can be created with Python BL and some Role can be assigned for the User. Otherwise, the User accessing the Application **should** be created first manually in the Application with the specific Role assigned. It is worth mentioning that LDAP or AD is using password to Authenticate the User, and it does not contain Roles information in the LDAP tree, unless LDAP is extended to contain Roles for Jam.py.

Please refer to more info in here: https://groups.google.com/g/jam-py/c/Q2iqvSA1zTM/m/o8HKxhvSCgAJ

Note: It is worth using SSL with all connections to LDAP or AD.

3.7.5 SAML or SSO Authentication

SAML will be supported in newer major versions other then v5. SAML depends on the DNS service and the SSL Certificates issued for the Application name. It is a large implementation for mostly commercial environment.

Please visit Demo Application in here: https://auth.jampyapplicationbuilder.com

The default username is: jam@jam.com with the same password. JumpCloud is used as the SSO provider.

To test the SAML attributes released from the provider, use https://demo.jampyapplicationbuilder.com application with the same username and password. It should be possible to cross authenticate for this two applications.

The SAML authentication code is not publicly available at this time of writing. The code is tested with MS Azure, Shibboleth and JumpCloud. MS Graph should also work.

3.7.6 OAuth2, OpenID or SiteMinder (CA) Authentication

Similar to SAML, it is possible to develop other Authentication mechanisms. SiteMinder is supported by Apache.

3.7.7 MFA or Two Factor Authentication

Instead of using LDAP or SAML, the Application might consume Multi Factor Authentication, with Google Authenticator, FreeOTP or similar software.

Please refer to more info in here: https://groups.google.com/g/jam-py/c/Nisyemcx6Vc/m/d6m7A4WSDQAJ

3.8 User Registration Form

Instead of manually creating the Users, regardless of Authentication method, the Application might use a Registration Form for initial User creation. the default Role might be assigned for the User by *Business Logic (BL)*.

Please consult the official Jam.py Documentation how to create a Registration Form.

3.9 Forgotten Password Method

For the Application Authenticated Users it might be worth to provide the Forgotten Password method by email. Jam.py already has *Change Password* method explained in the official documentation. However, Forgotten Password method depends on multitude dependencies, namely sending emails from the Application, and SSL, to name a few.

Please refer to more info in here: https://groups.google.com/g/jam-py/c/SYn2R0ILy74/m/Y5YMzUsSCAAJ

3.10 Using Python libraries

The true power of *Jam.py* is with using Python libraries. Just like some other Python frameworks.

However, the Python code or the libraries are used within the Jam.py *Server Module*. And only there. With the *Server Module*, the source code is logically stored in relation to each part of the application. Just like the JavaScript is within the *Client Module*. Of course, *Jam.py* can *call* any *external* Python script as usual.

The *Northwind Traders* migrated application from MS Access has a lot of code to start with. Most of it for is related the RFM Analytics and bulk insert/update.

When deploying the application, the Python libraries must be installed first. Or, the application might fail to function. To avoid this behaviour, please consult the best practises when importing the libraries.

3.11 Generated Images

As mentioned, the *Jam.py* application is pure JavaScript. Hence, there is a limited number of html elements on a Single Page Application (SPA). This means we need to *wrap* the file system image(s), which needs displaying.

Those images are usually created by the Python procedure or similar server code and/or not uploaded to the application. The uploaded images need no special attention, since displaying is fully supported by the framework.

For example, the *Northwind Traders* application has a generated image on Analytics/RFM Analysis. The image is created every time the user visits the *RFM Analysis* and *wrapped* into a JavaScript image for displaying purposes.

CHAPTER

FOUR

“HOW-TO” GUIDES

Here you’ ll find short answers to “How do I⋯.?” types of questions.

TBC

4.1 How to install Jam.py on Windows

Adapted from Django Docs

The below document is adopted from Django Docs.

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

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

4.1.1 Install Python

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

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

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

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

4.1.2 About `pip`

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

4.1.3 Setting up a virtual environment

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

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

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

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

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

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

4.1.4 Install Jam.py

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

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

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

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

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

4.1.5 Common pitfalls

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

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

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

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

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

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

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

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

The output might be similar to below:

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

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

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

The output might be similar to below:

```
...\> Directory of C:\Users\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
```

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

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

Run the new project:

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

CHAPTER

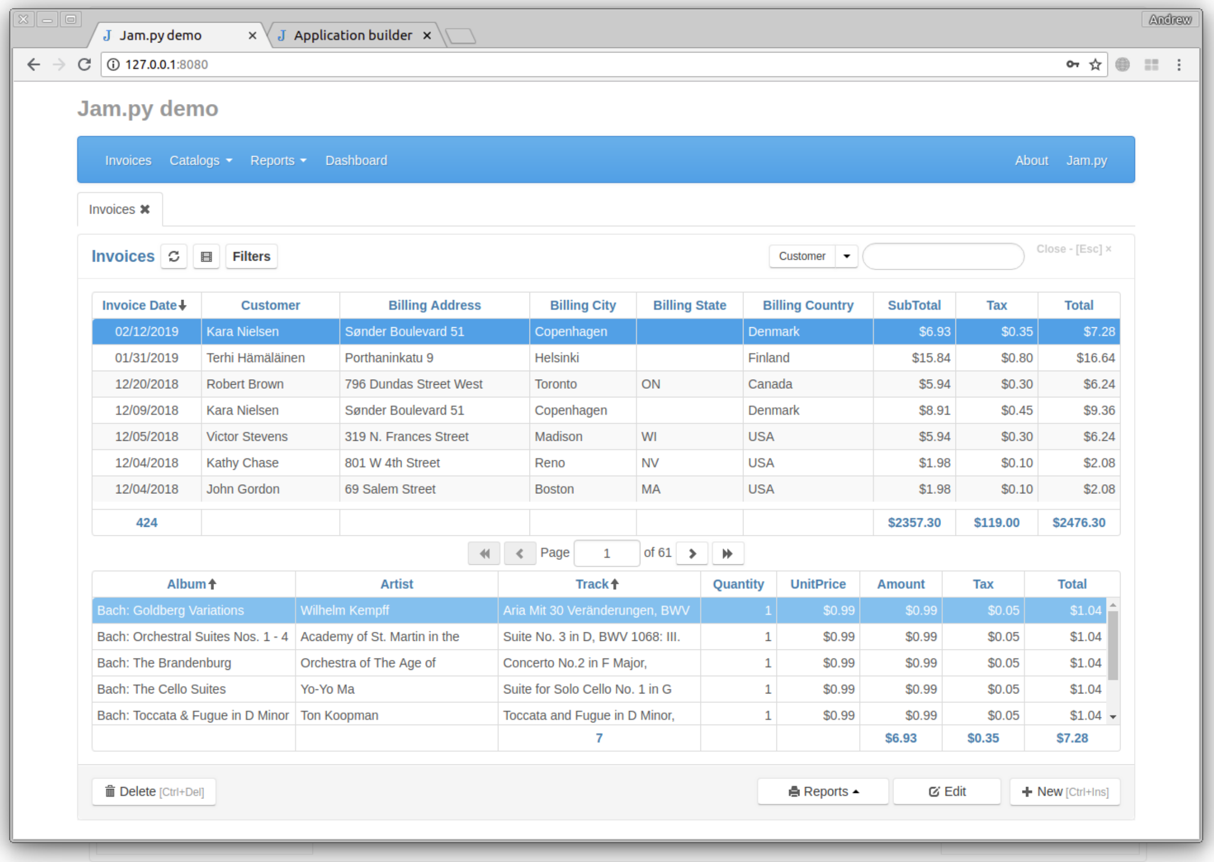
FIVE

HOW WAS DEMO BUILT?

So without reading the Doc' s (pun intended), jump on some Application building. From the official Jam.py Documentation, here is the Demo project:

5.1 Demo project

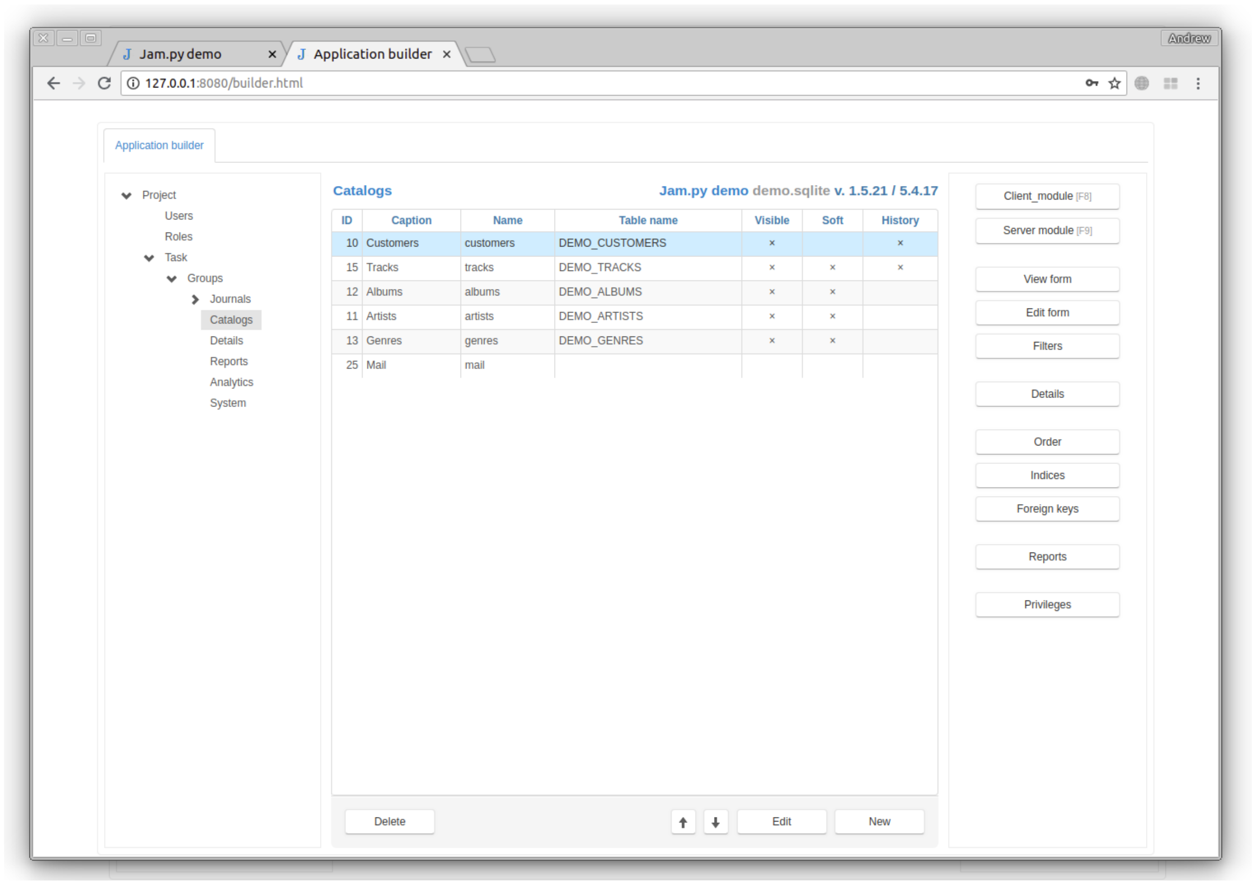
After downloading the Jam.py package, and starting Demo, the application is accessible with typing 127.0.0.1:8080/index.html or just 127.0.0.1:8080 in the Browser:



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

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

The Application Builder is accessible with typing 127.0.0.1:8080/builder.html in the Browser:



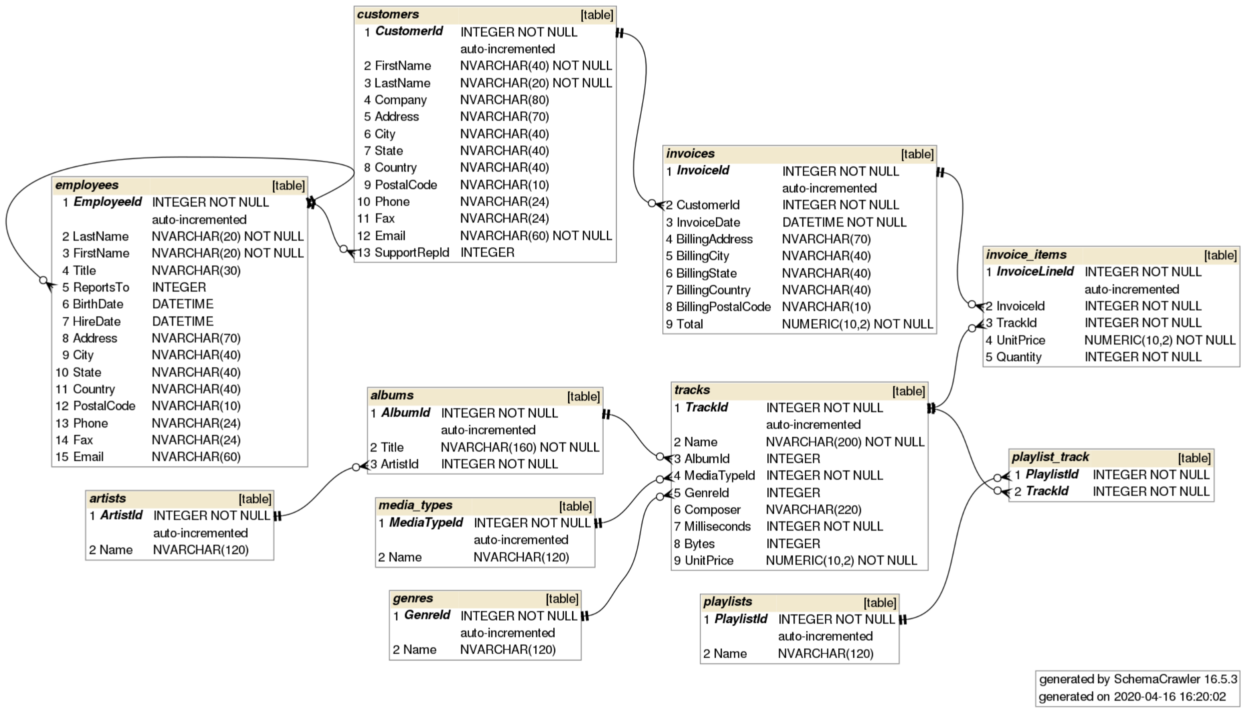
ID	Caption	Name	Table name	Visible	Soft	History
10	Customers	customers	DEMO_CUSTOMERS	×		×
15	Tracks	tracks	DEMO_TRACKS	×	×	×
12	Albums	albums	DEMO_ALBUMS	×	×	
11	Artists	artists	DEMO_ARTISTS	×	×	
13	Genres	genres	DEMO_GENRES	×	×	
25	Mail	mail				

The application is derived from the open-source Chinook database. It is an invoicing example with music tracks as invoiced products, customers, albums, etc. Basically, the application creates customer details, product details, and raises invoices for customers. That’ s it. Cool little application, though.

As mentioned, the demo application is derived from it. It is not a one-to-one mapping of all database fields and relations. We need to start somewhere, right?

5.2 Demo database

To better understand the actual database, here is the original Chinook relational database diagram:


customers [table]
1 CustomerId INTEGER NOT NULL auto-incremented
2 FirstName NVARCHAR(40) NOT NULL
3 LastName NVARCHAR(20) NOT NULL
4 Company NVARCHAR(80)
5 Address NVARCHAR(70)
6 City NVARCHAR(40)
7 State NVARCHAR(40)
8 Country NVARCHAR(40)
9 PostalCode NVARCHAR(10)
10 Phone NVARCHAR(24)
11 Fax NVARCHAR(24)
12 Email NVARCHAR(60) NOT NULL
13 SupportRepId INTEGER
invoices [table]
1 InvoiceId INTEGER NOT NULL auto-incremented
2 CustomerId INTEGER NOT NULL
3 InvoiceDate DATETIME NOT NULL
4 BillingAddress NVARCHAR(70)
5 BillingCity NVARCHAR(40)
6 BillingState NVARCHAR(40)
7 BillingCountry NVARCHAR(40)
8 BillingPostalCode NVARCHAR(10)
9 Total NUMERIC(10,2) NOT NULL
employees [table]
1 EmployeeId INTEGER NOT NULL auto-incremented
2 LastName NVARCHAR(20) NOT NULL
3 FirstName NVARCHAR(20) NOT NULL
4 Title NVARCHAR(30)
5 ReportsTo INTEGER
6 BirthDate DATETIME
7 HireDate DATETIME
8 Address NVARCHAR(70)
9 City NVARCHAR(40)
10 State NVARCHAR(40)
11 Country NVARCHAR(40)
12 PostalCode NVARCHAR(10)
13 Phone NVARCHAR(24)
14 Fax NVARCHAR(24)
15 Email NVARCHAR(60)
invoice_items [table]
1 InvoiceLineId INTEGER NOT NULL auto-incremented
2 InvoiceId INTEGER NOT NULL
3 TrackId INTEGER NOT NULL
4 UnitPrice NUMERIC(10,2) NOT NULL
5 Quantity INTEGER NOT NULL
albums [table]
1 AlbumId INTEGER NOT NULL auto-incremented
2 Title NVARCHAR(160) NOT NULL
3 ArtistId INTEGER NOT NULL
tracks [table]
1 TrackId INTEGER NOT NULL auto-incremented
2 Name NVARCHAR(200) NOT NULL
3 AlbumId INTEGER
4 MediaTypeId INTEGER NOT NULL
5 GenreId INTEGER
6 Composer NVARCHAR(220)
7 Milliseconds INTEGER NOT NULL
8 Bytes INTEGER
9 UnitPrice NUMERIC(10,2) NOT NULL
playlist_track [table]
1 PlaylistId INTEGER NOT NULL
2 TrackId INTEGER NOT NULL
artists [table]
1 ArtistId INTEGER NOT NULL auto-incremented
2 Name NVARCHAR(120)
media_types [table]
1 MediaTypeId INTEGER NOT NULL auto-incremented
2 Name NVARCHAR(120)
genres [table]
1 GenreId INTEGER NOT NULL auto-incremented
2 Name NVARCHAR(120)
playlists [table]
1 PlaylistId INTEGER NOT NULL auto-incremented
2 Name NVARCHAR(120)
generated by SchemaCrawler 16.5.3
generated on 2020-04-16 16:20:02

Why is the above diagram important? Because it is demonstrating the table’s Primary and Foreign Keys, which are essentially table *Lookup fields* used within Jam.py. More about that later.

So now that we know what the Demo application is all about, we can dive into more details. First, what can we actually expect from Jam.py?

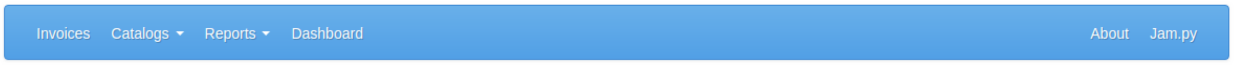
5.3 What to expect?

Low code is what everyone talks about. Below is what Jam.py provides with no coding needed at all, at least not for this Demo. The code does exist; it is there. Otherwise, the application would not work at all. However, this only means that no coding is required by us to build similar applications. Because the code is there and available to us, this also means we can adapt the existing code for our requirements. Mostly, it is a copy/paste from this Demo, with some minor changes for the application we are building. This is important to understand. Jam.py provides many bells and whistles or so-called batteries out of the box. However, Jam.py is not an “Out of the Box Application”! It is a framework, so we build applications within the constraints of the framework.

Just like Django’s MVC. When building applications with Django, we operate within similar constraints. Both frameworks are flexible enough for the job at hand. The main difference is the task or the application. Django could build the Jam.py application, but it would be overkill, since Jam.py is way more specialized in doing so. Jam.py is a specialized framework. It is the right tool for the right job. Let’s see what we can expect from it.

5.3.1 DropDown Menu(s)

On the below menu all options on the left are created automatically. The “About” and “Jam.py” on the right side is a code. We can build complete menu manually, not to worry. Will get there soon.



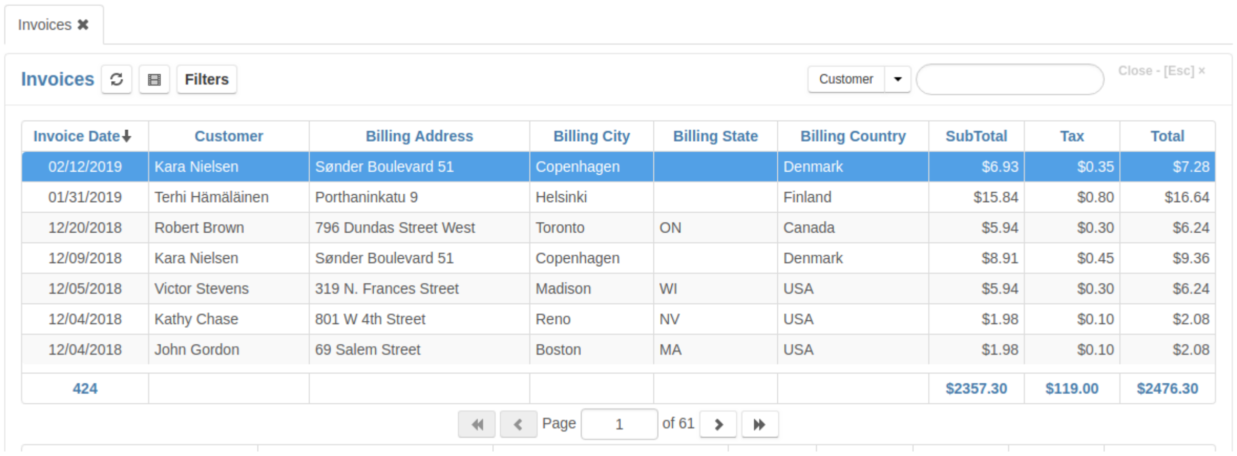
5.3.2 Data Grid(s)

The Demo Invoices data grid is called a Journal. In Accounting, these are General Ledger tables. In databases, this is a Master Table. This grid is presented automatically with pagination at the bottom and can contain a History button, refresh button, and Filters button. These are the default options and can be turned on/off. If a non-default button is needed, it can be added with code, though.

Did I mention the search for any field and sorting for any field? It’ s automated.

What is also done by Jam.py with no code is a summary for any numerical field or the number of records for other fields.

As seen, the Invoices data grid is presented as a tab. As we open more grids, more tabs will appear here automatically. Tab captions can be changed with code. What we see initially is the no-code default.



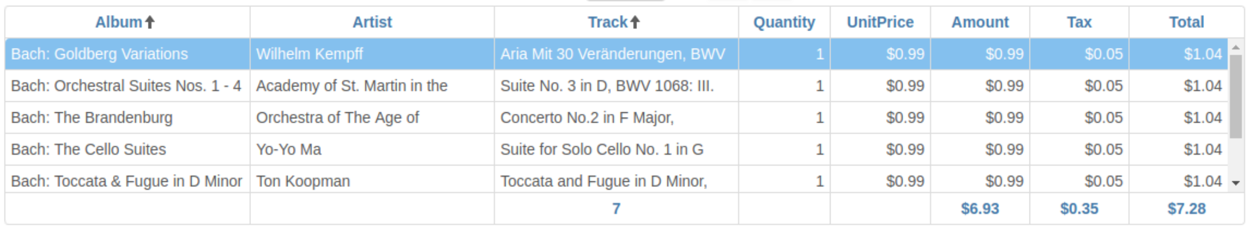
Invoice Date	Customer	Billing Address	Billing City	Billing State	Billing Country	SubTotal	Tax	Total
02/12/2019	Kara Nielsen	Sønder Boulevard 51	Copenhagen		Denmark	$6.93	$0.35	$7.28
01/31/2019	Terhi Hämäläinen	Porthaninkatu 9	Helsinki		Finland	$15.84	$0.80	$16.64
12/20/2018	Robert Brown	796 Dundas Street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

5.3.3 More Data Grid(s)

The Demo Invoices data has a Details grid. These are the *Invoice items* details. They are created automatically and can contain summary fields and sorting. The “Total” field visible is a code! What is not visible is editing directly in the grid!

There can be any number of details data grids. Now, that is actually pretty impressive. Any number, huh? We’ ll get there.

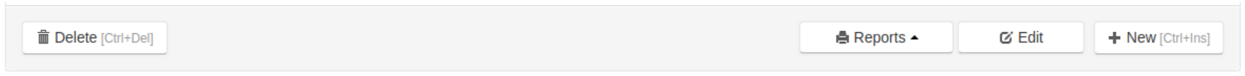
Like the above grid, each Details grid is presented in tabbed format. The more Details added to a journal, the more tabs we see here.



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

5.3.4 Data Grid Header/Footer

Automatically provided is a classic New/Delete/Edit option for any Data Grid. It can exist on the Top/Bottom of some data Grid, hence I call it Header/Footer. It can also contain other buttons or a menu, which is a code driven, so coding is needed for non default buttons.



Ok, do not want to go any further, just a minimum to get us going. Adding the Buttons, Multiple Details as Tabs, or changing the look and feel is not discussed yet.

5.3.5 Any questions?

So now that we’ ve covered the first thing we see on the Demo application, any questions?

All of the above is driven by the Builder GUI. We might be saying it’ s no big deal, but it actually is a big deal! Because to code this from the beginning, like with Django, one would need years and years of coding experience to cover all possible scenarios. Or a team of dedicated developers.

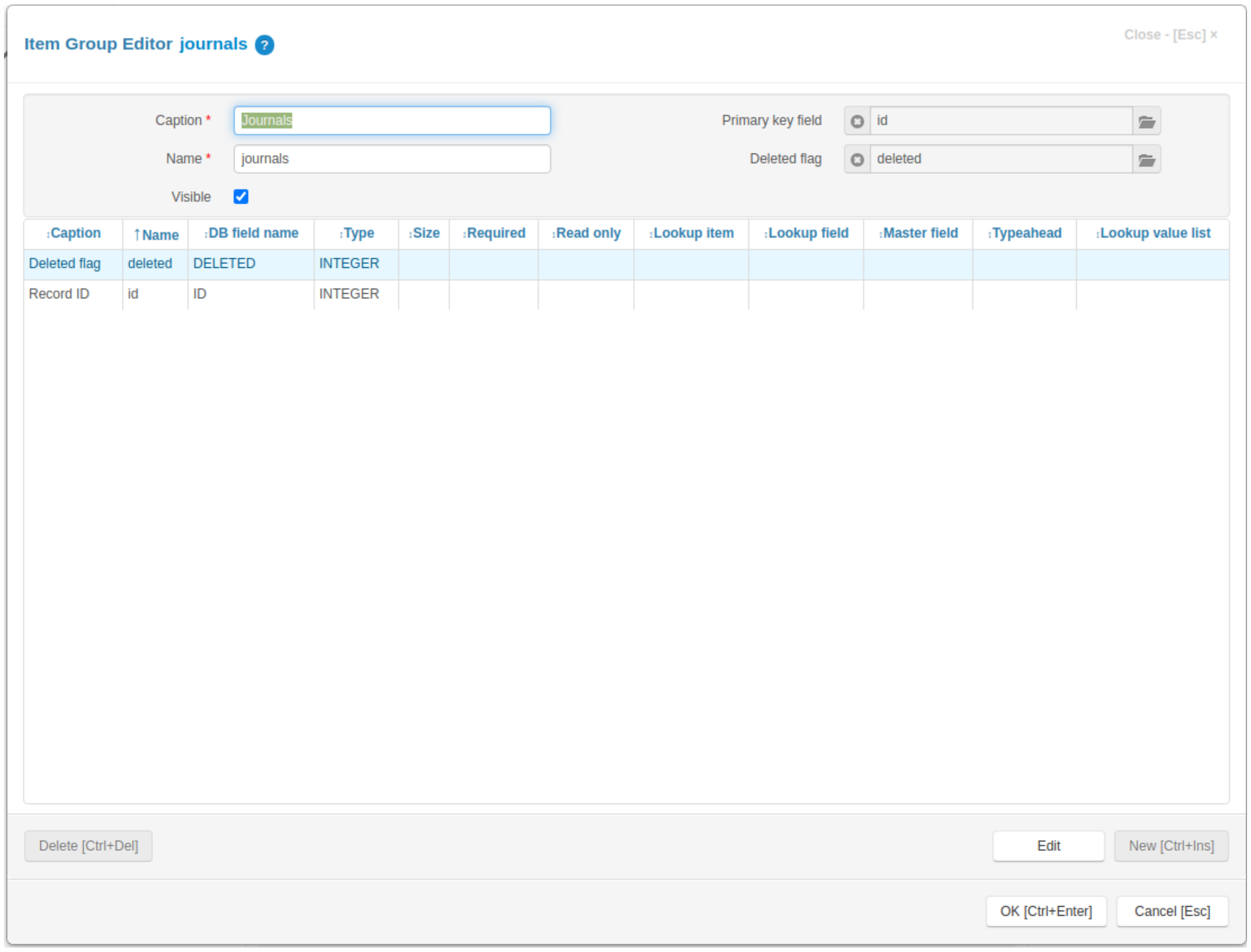
I am ignoring the Reports and Analytics for now. Only concentrating on no-code or low-code. The Reports and Analytics require some coding experience. Not much, but still needed.

5.4 Ok, how do I start?

If no questions (cough, cough), I think we should first start with building some tables. Since we are doing Invoices, lets start with this. It is a sort of top-to-bottom approach!

5.4.1 Invoices

For an invoice, we need a Customer, some Product and somewhere to store invoiced data, like a Journal. To be fair, Journal is just a name. We do not really need to stick to the Jam.py Demo application naming! Rename anything to whatever floats your boat!



Caption * Journals
Name * journals
Visible ✓
Primary key field id
Deleted flag deleted

Caption	Name	DB field name	Type	Size	Required	Read only	Lookup item	Lookup field	Master field	Typeahead	Lookup value list
Deleted flag	deleted	DELETED	INTEGER								
Record ID	id	ID	INTEGER								

We can rename the Journals caption to anything, since this is just a Caption. However, the Name can be used in the code somewhere, so it is always advisable to use the "Find" built-in function to search where some Name is used.

On above screenshot we see the "Deleted Flag" and "Record ID". This is just a bit of built-in functionality. When we create a new table in the Journals group, these two records will be created automatically. We can think of this as a table template.

The Master field field seen above is also a feature. In the Demo application Invoices table, it is used to identify which item belongs to which Invoice.

Note: *Master fields are not real fields in the database, they get populated when we select a value in the Lookup field.*

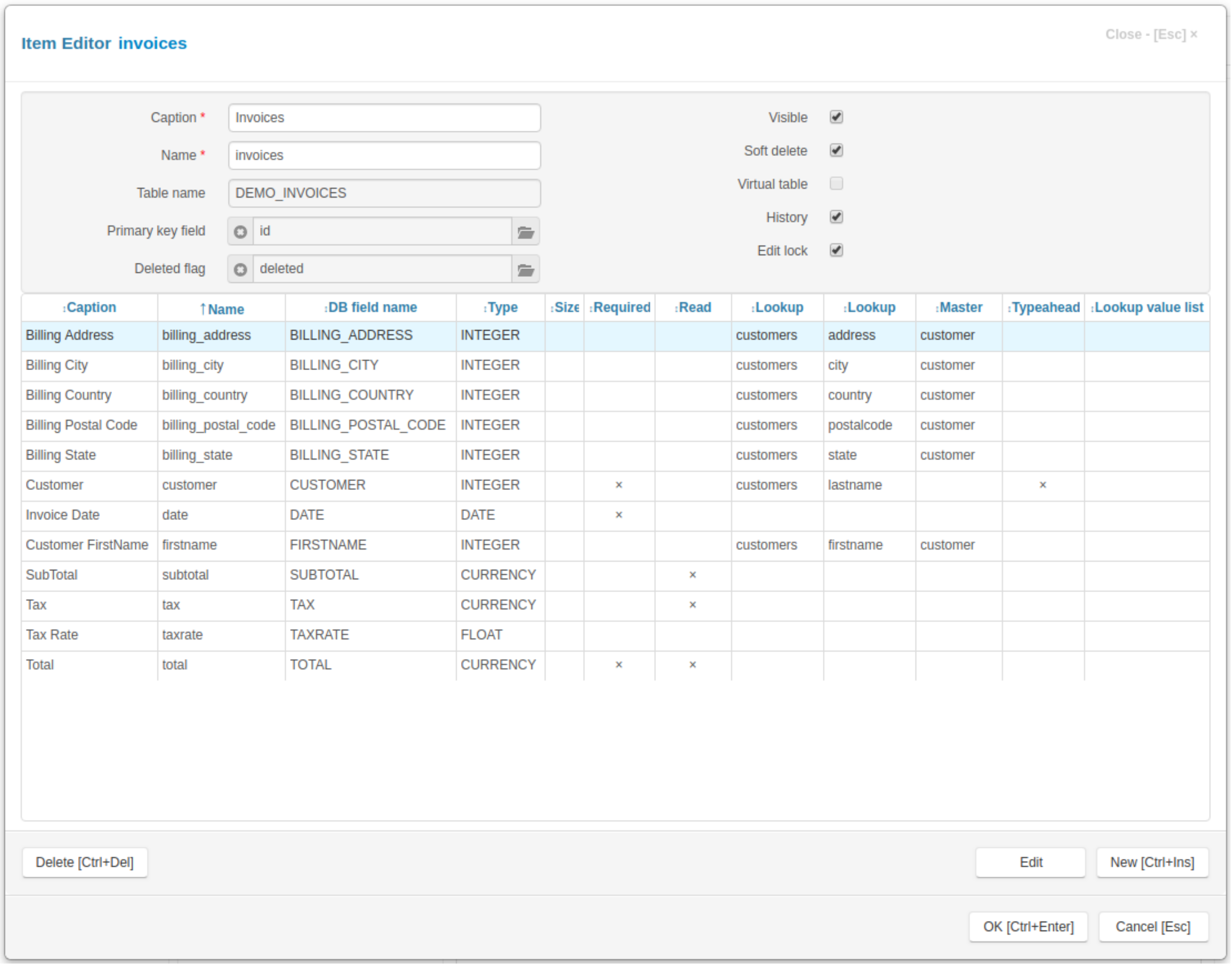
Master fields are virtual fields in Jam.py that get populated based on the value selected in the lookup field. They are not actual columns in the database table, but rather a way to display related information from another table in the UI. When a value is selected in the lookup field, the master field is automatically updated with the corresponding value from the related table. This allows for easy navigation and visualization of related data in the UI.

This is actually very powerful feature in my books, because it quickly identifies stuff. Which stuff you might ask? Anything built as a *Lookup fields*. By setting the *Master Field* to *Customer*, we can easily identify all data related to the *Customers* table that we are pulling information from.

I would leave it here for now, but just note that the *Master field* can also be achieved with code for imported tables. Usually, imported tables are from other systems like MSSQL or MySQL, and are not created by Jam.py. In this case, we can add the table as a detail to only one master. With the *Master field* feature in Jam.py, we can do much more. We will get to this later.

Only Invoices, Invoices Items, and Tracks are using the "Master field" in Demo application.

The Demo Invoices table:



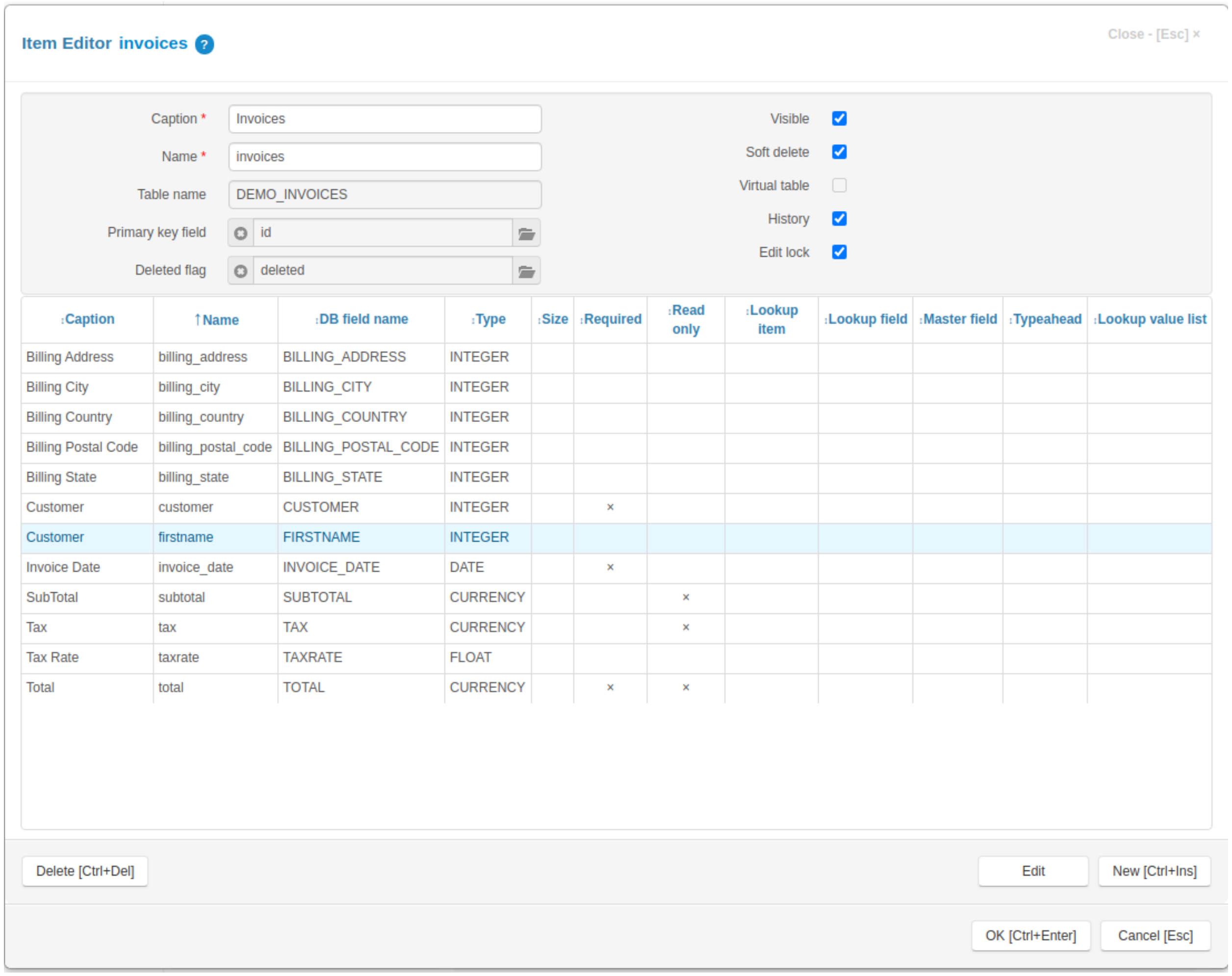
Caption *	Invoices
Name *	invoices
Table name	DEMO_INVOICES
Primary key field	id
Deleted flag	deleted
Visible	✓
Soft delete	✓
Virtual table	
History	✓
Edit lock	✓

Caption	Name	DB field name	Type	Size	Required	Read	Lookup	Lookup	Master	Typeahead	Lookup value list
Billing Address	billing_address	BILLING_ADDRESS	INTEGER				customers	address	customer		
Billing City	billing_city	BILLING_CITY	INTEGER				customers	city	customer		
Billing Country	billing_country	BILLING_COUNTRY	INTEGER				customers	country	customer		
Billing Postal Code	billing_postal_code	BILLING_POSTAL_CODE	INTEGER				customers	postalcode	customer		
Billing State	billing_state	BILLING_STATE	INTEGER				customers	state	customer		
Customer	customer	CUSTOMER	INTEGER		×		customers	lastname		×	
Invoice Date	date	DATE	DATE		×						
Customer FirstName	firstname	FIRSTNAME	INTEGER				customers	firstname	customer		
SubTotal	subtotal	SUBTOTAL	CURRENCY			×					
Tax	tax	TAX	CURRENCY			×					
Tax Rate	taxrate	TAXRATE	FLOAT								
Total	total	TOTAL	CURRENCY		×	×					

We see that Invoices table is referencing a lot from the Customers table! This is where we look at the *Chinook* database, we see the relation Customers - Invoices!

This means we can create the table but with no Customers one, we can't really achieve what is needed for Invoices to function correctly.

With no Customers table our Invoices table would look like this:



Caption *	Invoices
Name *	invoices
Table name	DEMO_INVOICES
Primary key field	id
Deleted flag	deleted
Visible	✓
Soft delete	✓
Virtual table	
History	✓
Edit lock	✓

Caption	Name	DB field name	Type	Size	Required	Read only	Lookup item	Lookup field	Master field	Typeahead	Lookup value list
Billing Address	billing_address	BILLING_ADDRESS	INTEGER								
Billing City	billing_city	BILLING_CITY	INTEGER								
Billing Country	billing_country	BILLING_COUNTRY	INTEGER								
Billing Postal Code	billing_postal_code	BILLING_POSTAL_CODE	INTEGER								
Billing State	billing_state	BILLING_STATE	INTEGER								
Customer	customer	CUSTOMER	INTEGER		×						
Customer	firstname	FIRSTNAME	INTEGER								
Invoice Date	invoice_date	INVOICE_DATE	DATE		×						
SubTotal	subtotal	SUBTOTAL	CURRENCY			×					
Tax	tax	TAX	CURRENCY			×					
Tax Rate	taxrate	TAXRATE	FLOAT								
Total	total	TOTAL	CURRENCY		×	×					

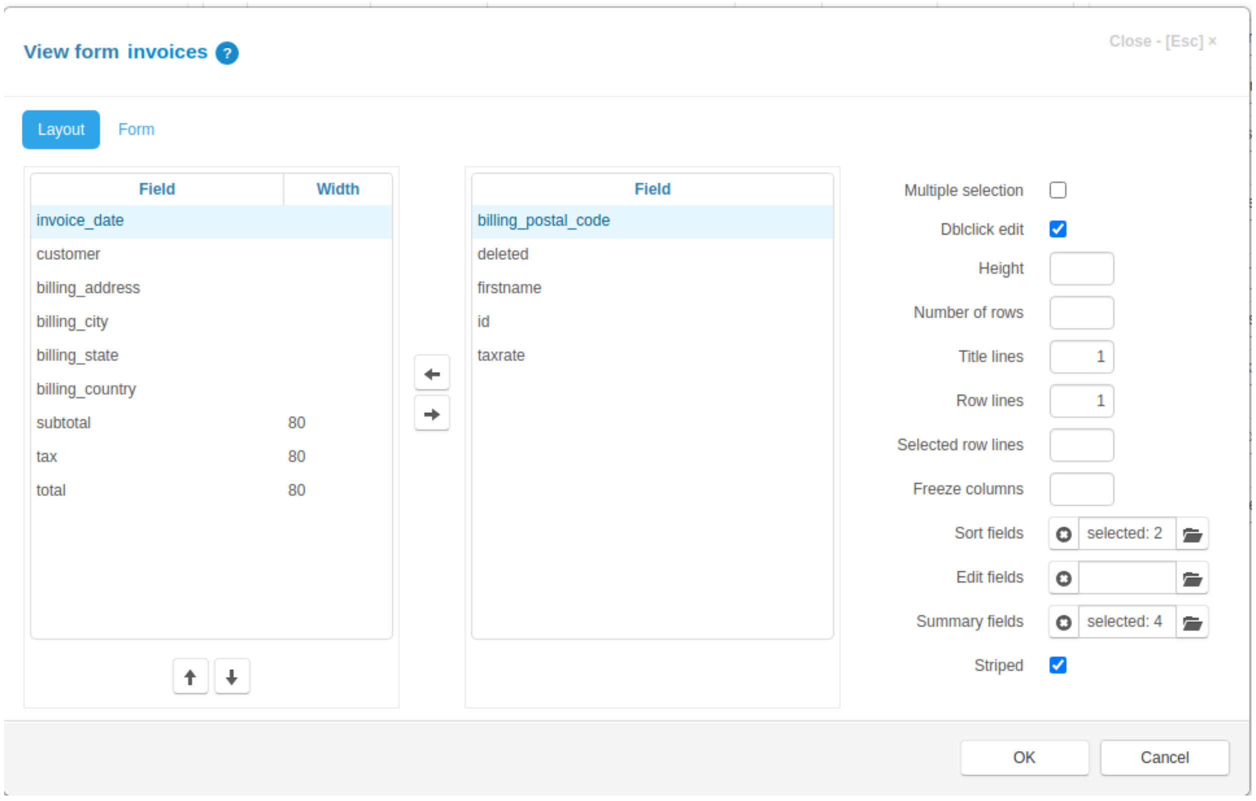
It is exactly the same thing, same as above one would think. Not really. It would not display any data because Jam.py would assume that all of Customers data is coming from one, and only one table, which is Invoices table. It is also true that all fields would have the Integer Type, which is not quite right. It is presented here just as an example and it is not the right way of doing it.

However, from the diagram, the Invoices is pulling some data from Customers and "Invoice Items" table. That is the Holy Grail of any application building. Pulling data from here and there, and showing the result with no code, or low code, is exactly what Jam.py does. We call that a *Lookup fields*.

As seen, on the right hand side checked options are "Visible" , "Soft Delete" , "History" and "Edit lock" . Refer to Item Editor dialog for more info.

Also, worth mentioning is that Invoices do contain a *tiny code* for calculating "Tax" and "Total" column. It also utilises Jam.py feature to alert the User with a custom message. It does that on Editing only.

Here we touch the View/Edit option on *builder.html*. When we click on Journals/Invoices, there is a "View Form" option on the right hand side.

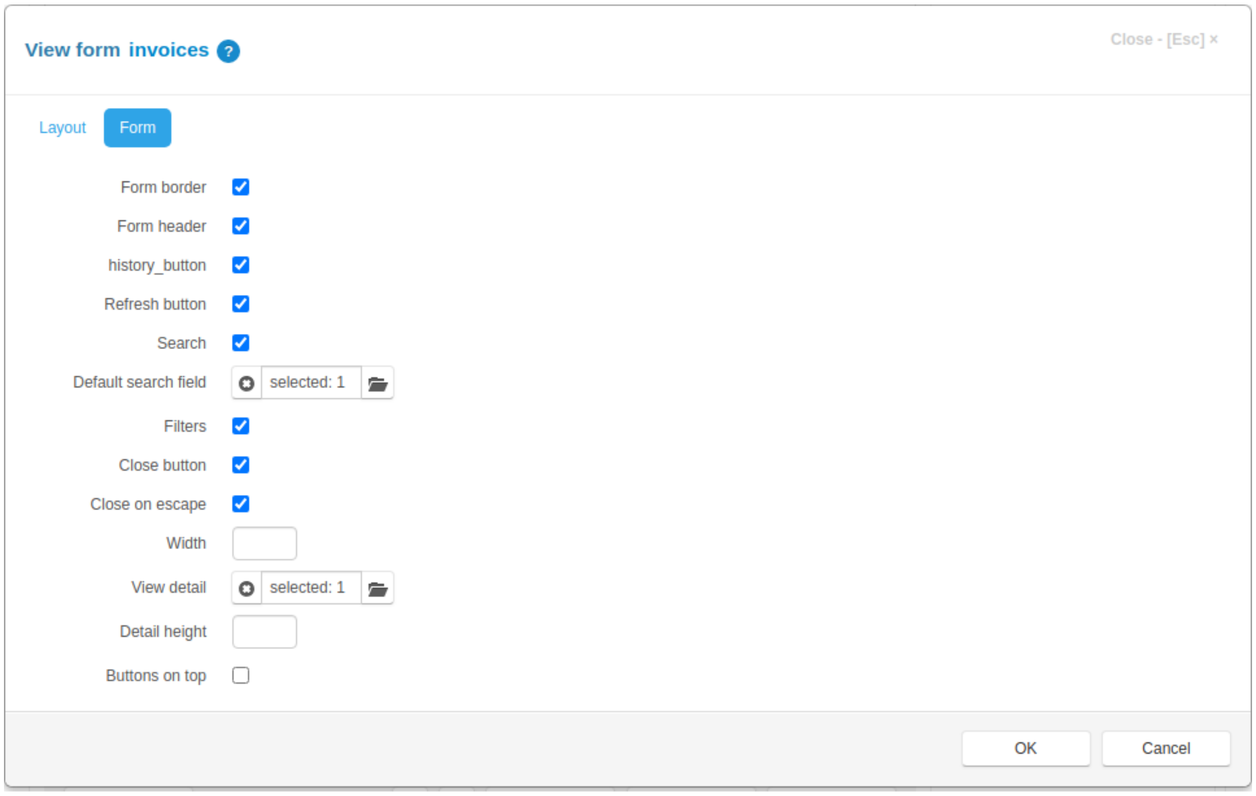


Field	Width
invoice_date	
customer	
billing_address	
billing_city	
billing_state	
billing_country	
subtotal	80
tax	80
total	80

Field
billing_postal_code
deleted
firstname
id
taxrate

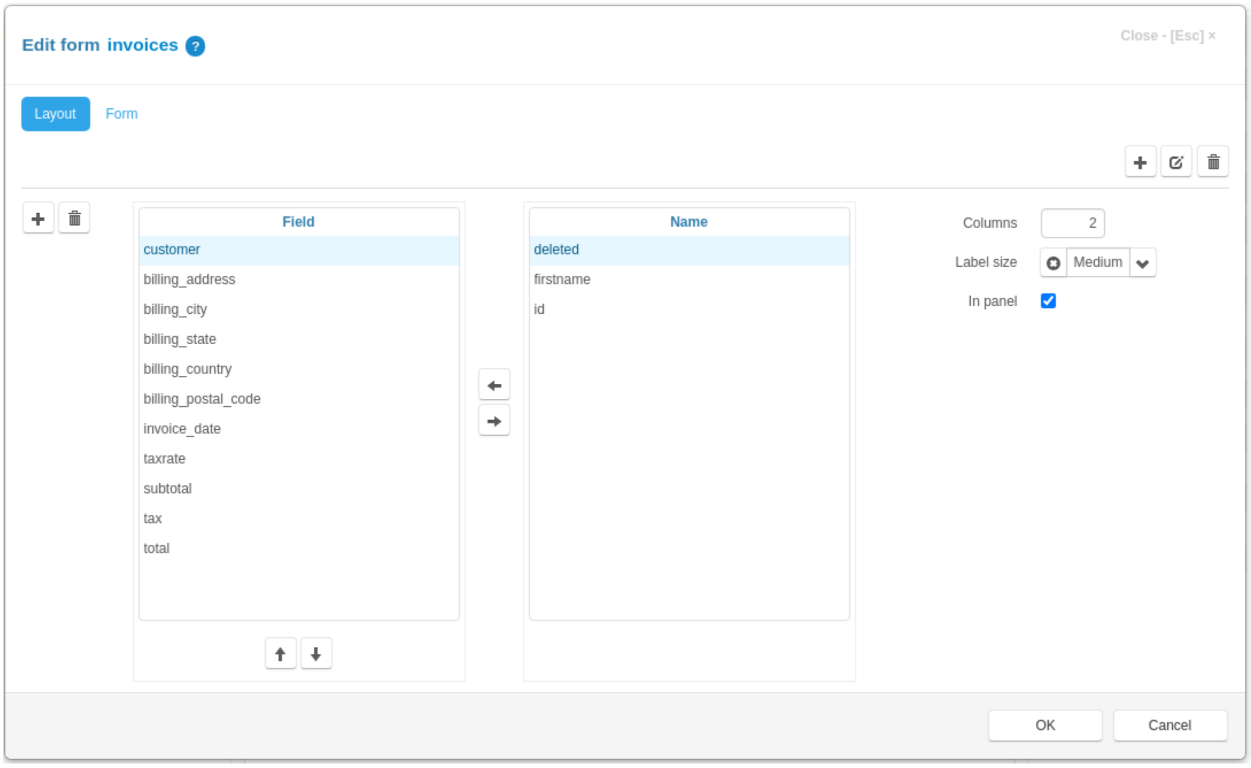
Multiple selection	☐
Dblclick edit	✓
Height	
Number of rows	
Title lines	1
Row lines	1
Selected row lines	
Freeze columns	
Sort fields	selected: 2
Edit fields	
Summary fields	selected: 4
Striped	✓

Here we control the layout, sorting and summary fields, as well as fields visibility on the Demo data grid. On the Form tab, we see a number of other options, but most importantly the "View detail" option, with selected "Invoice table". We can select as many Details as we like, providing they exist. If no detail is selected, there would be nothing to display. Somehow counter-intuitive on the first instance, however we are controlling the visibility with this option, not more then that.



Form border	✓
Form header	✓
history_button	✓
Refresh button	✓
Search	✓
Default search field	selected: 1
Filters	✓
Close button	✓
Close on escape	✓
Width	
View detail	selected: 1
Detail height	
Buttons on top	☐

Similar for Edit option. Exclude or select for editing whatever is necessary, as well as details needed for editing. While on it, we can also control how to present the edit form in the sense of tabs or bands. This is done on "Plus" icon on the left hand side. I would encourage to play with this options, it is quite subjective what and how something should be presented.



Field
customer
billing_address
billing_city
billing_state
billing_country
billing_postal_code
invoice_date
taxrate
subtotal
tax
total

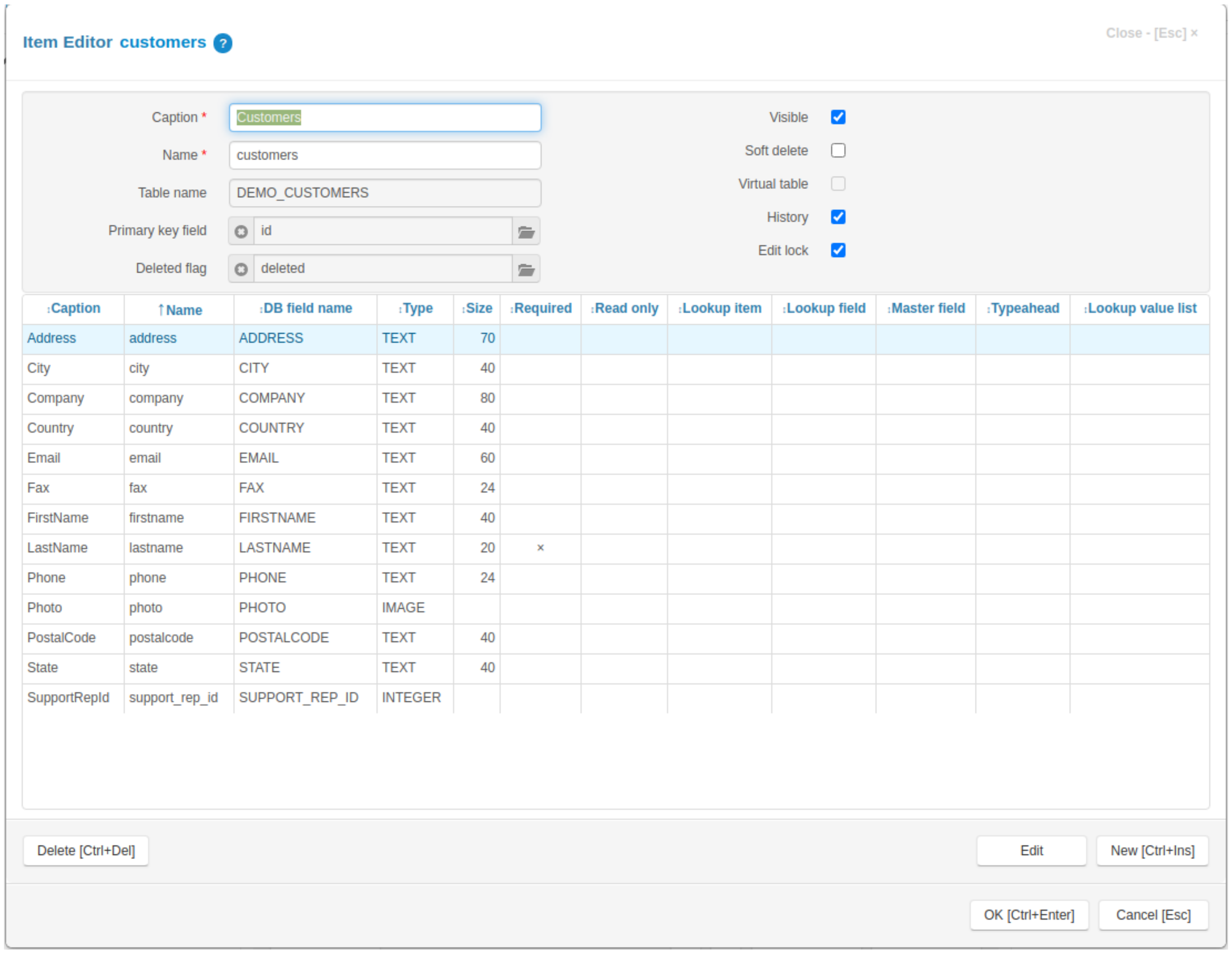
Name
deleted
firstname
id

Columns 2
Label size Medium
In panel ✓

5.4.2 Customers

Back to Customers, if we visited Tutorial. Part 1. First project in the Doc's, I hope it is pretty much clear how to create a simple CRM. Hence, Demo application has Customers table as well, which is consumed by the Invoices table.

Here is how Customers table looks like:


Item Editor customers

Close - [Esc] ×

Caption *	Customers	Visible	✓
Name *	customers	Soft delete	☐
Table name	DEMO_CUSTOMERS	Virtual table	☐
Primary key field	id	History	✓
Deleted flag	deleted	Edit lock	✓

Caption	Name	DB field name	Type	Size	Required	Read only	Lookup item	Lookup field	Master field	Typeahead	Lookup value list
Address	address	ADDRESS	TEXT	70							
City	city	CITY	TEXT	40							
Company	company	COMPANY	TEXT	80							
Country	country	COUNTRY	TEXT	40							
Email	email	EMAIL	TEXT	60							
Fax	fax	FAX	TEXT	24							
FirstName	firstname	FIRSTNAME	TEXT	40							
LastName	lastname	LASTNAME	TEXT	20	×						
Phone	phone	PHONE	TEXT	24							
Photo	photo	PHOTO	IMAGE								
PostalCode	postalcode	POSTALCODE	TEXT	40							
State	state	STATE	TEXT	40							
SupportRepId	support_rep_id	SUPPORT_REP_ID	INTEGER								

It is a simple table with no lookups to some other tables, like Invoices for example. Because it is derived from the *Chinook* database, we can see that Demo is missing the Employees table, so the SupportRepid is not used anywhere. That is fine. No harm done. We might argue that Customers table can be split in more tables, like Country, State or similar. While on it, please see Tutorial. Part 2. File and image fields for adding Image field. We did touch base with the *View/Edit* option on Invoices, no need for repeating.

Now that we have two tables in place, we can set the Lookup fields in Invoices in a similar fashion as in Tutorial. Part 1. First project.

We did not touch the third table yet, which is the “Invoice items” . This is the part of Tutorial. Part 3. Details

5.4.3 Invoice items

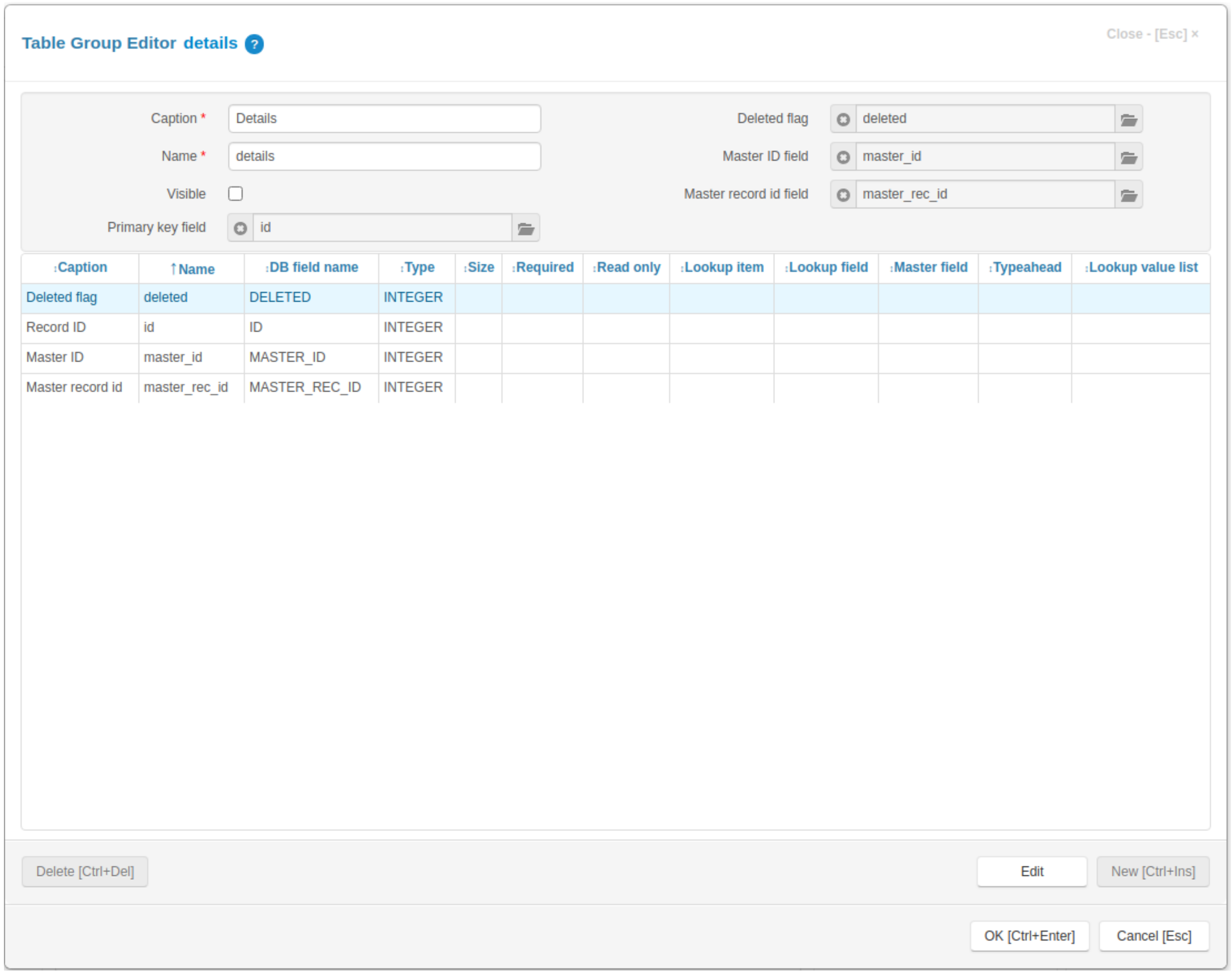
To automatically add some details to some other table, with no code what so ever, we need to use Jam.py Details feature.

Why would we do that? Well, we could opt for coding. It is available feature and used mostly for Imported tables, which do not contain additional fields Jam.py is using.

However, Demo application is demonstrating how Jam.py is functioning, so why not using it.

Basically, since the Invoice Table is a detail of the master table Invoices, it has the **master_rec_id** field that stores a reference to an invoice. As a master record, we can show an invoice that contains the current item. Clear as mud?

The Details Group differs from other default tables slightly:



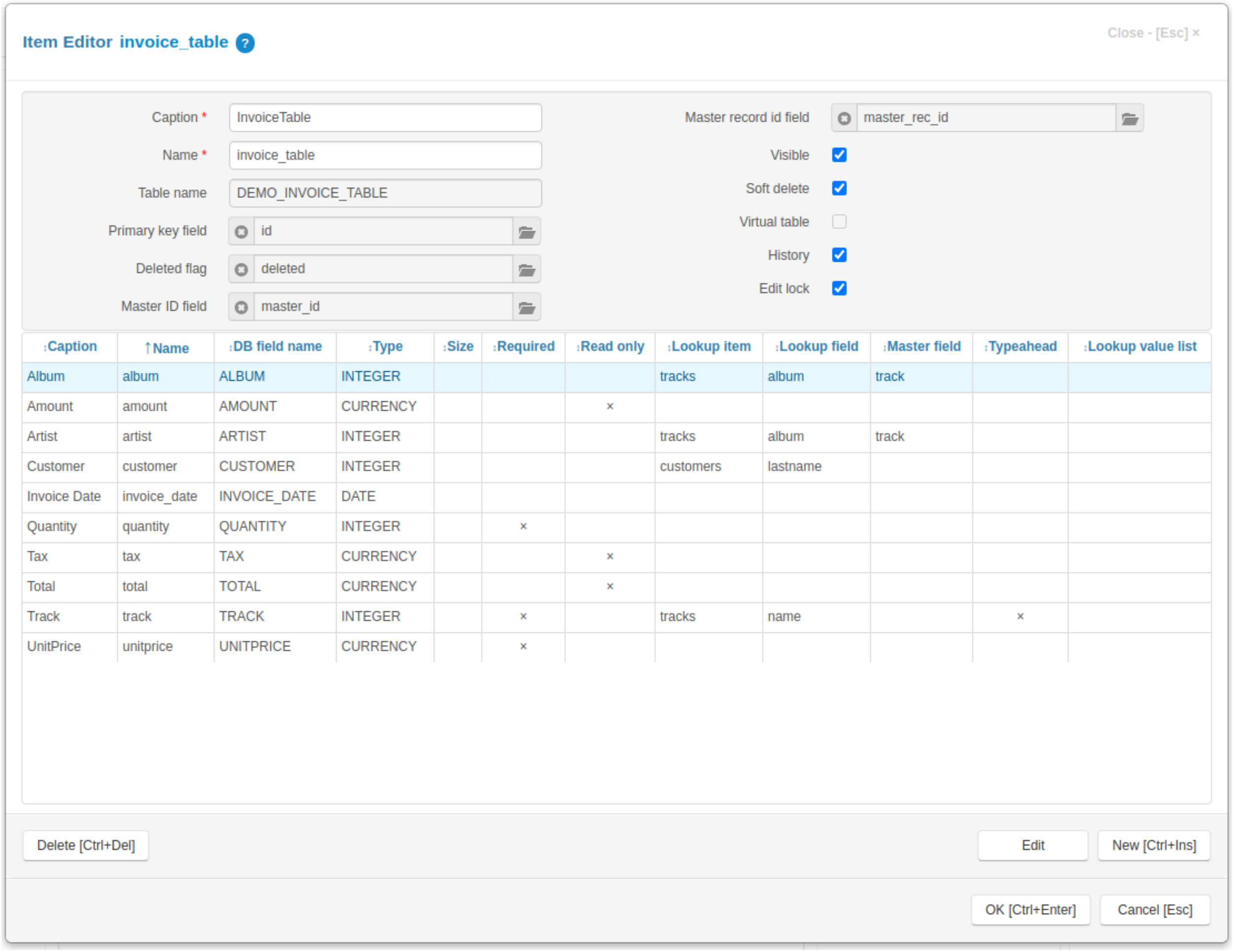
Caption *	Details	Deleted flag	deleted
Name *	details	Master ID field	master_id
Visible	☐	Master record id field	master_rec_id
Primary key field	id		

Caption	Name	DB field name	Type	Size	Required	Read only	Lookup item	Lookup field	Master field	Typeahead	Lookup value list
Deleted flag	deleted	DELETED	INTEGER								
Record ID	id	ID	INTEGER								
Master ID	master_id	MASTER_ID	INTEGER								
Master record id	master_rec_id	MASTER_REC_ID	INTEGER								

See the additional fields? The fields in question are master_rec_id and master_id. This fields do not exist in any legacy systems. None of the applications out there are using it. However, let' s not be afraid of this feature. As mentioned, we can and we will achieve the Jam.py functionality with a bit of code for legacy systems without this fields.

In addition, the option "Visible" is unchecked. It makes no sense in setting the Details as visible, since we edit/view them within the Invoices only. It is possible to set it as visible, though.

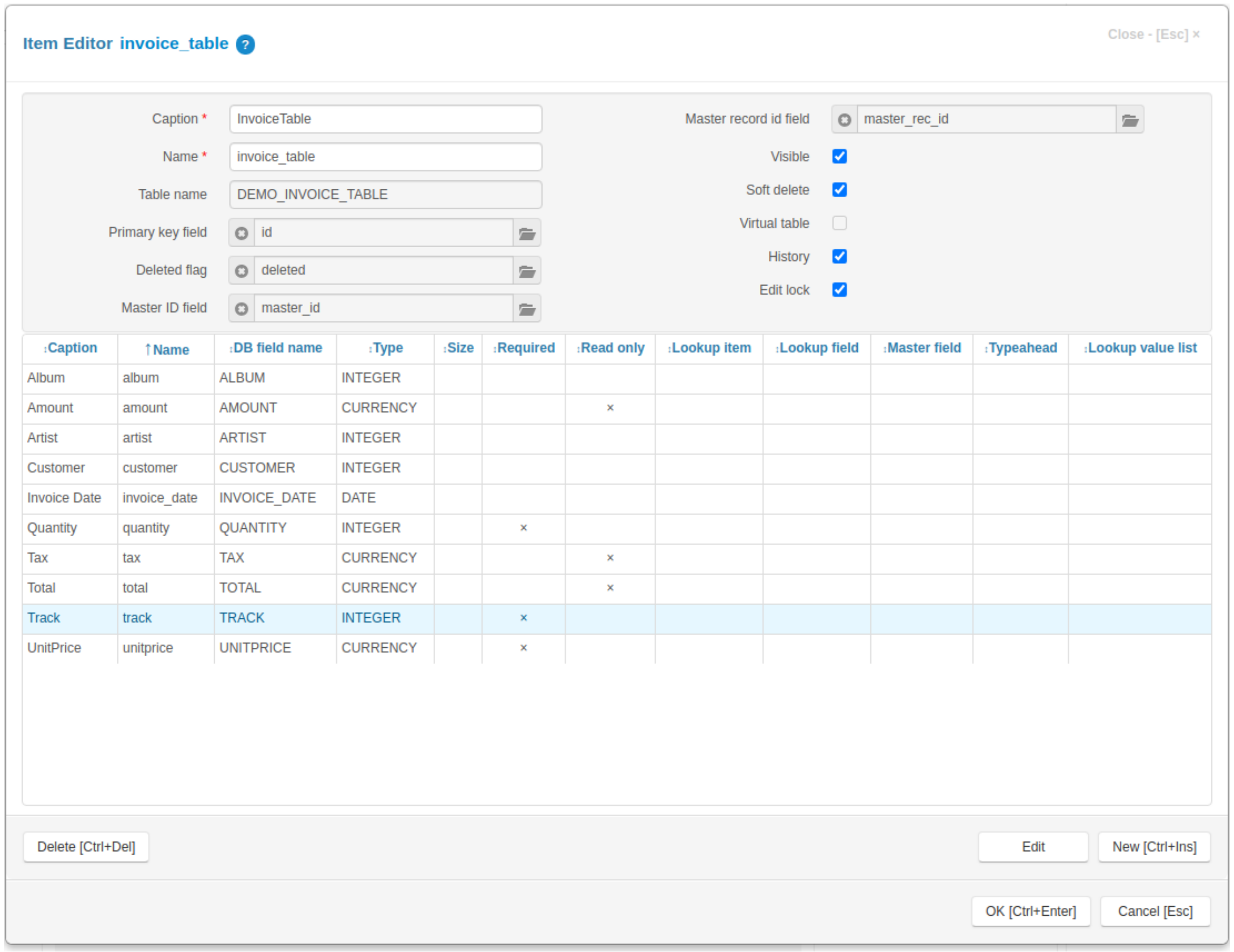
Just like before with Invoices table, but not with Customers, the "Invoice Table" is referencing two tables, "Tracks" and "Customers" .



Caption *	InvoiceTable
Name *	invoice_table
Table name	DEMO_INVOICE_TABLE
Primary key field	id
Deleted flag	deleted
Master ID field	master_id
Master record id field	master_rec_id
Visible	✓
Soft delete	✓
Virtual table	
History	✓
Edit lock	✓

Caption	Name	DB field name	Type	Size	Required	Read only	Lookup item	Lookup field	Master field	Typeahead	Lookup value list
Album	album	ALBUM	INTEGER				tracks	album	track		
Amount	amount	AMOUNT	CURRENCY			×					
Artist	artist	ARTIST	INTEGER				tracks	album	track		
Customer	customer	CUSTOMER	INTEGER				customers	lastname			
Invoice Date	invoice_date	INVOICE_DATE	DATE								
Quantity	quantity	QUANTITY	INTEGER		×						
Tax	tax	TAX	CURRENCY			×					
Total	total	TOTAL	CURRENCY			×					
Track	track	TRACK	INTEGER		×		tracks	name		×	
UnitPrice	unitprice	UNITPRICE	CURRENCY		×						

If below table screenshot was the "blank" Invoice Items table, with no *Lookup fields* to the above two tables, it would not show anything. The reason why we using "Customers" table and not the "Invoices" , InvoiceId from *Chinook* database diagram, is the presentation. It is more nicely presented with the customer last name on the screen, as compared to with some meaningless number. In this case it is InvoiceId (integer), as on the diagram, which ultimately identifies the Customer by the CustomerID. Similar with Tracks, we are using it to do the Lookup on a completely different tables, namely "Albums" and "Artists" .


Item Editor invoice_table

Caption *	InvoiceTable
Name *	invoice_table
Table name	DEMO_INVOICE_TABLE
Primary key field	id
Deleted flag	deleted
Master ID field	master_id
Master record id field	master_rec_id
Visible	✓
Soft delete	✓
Virtual table	
History	✓
Edit lock	✓

Caption	Name	DB field name	Type	Size	Required	Read only	Lookup item	Lookup field	Master field	Typeahead	Lookup value list
Album	album	ALBUM	INTEGER								
Amount	amount	AMOUNT	CURRENCY			×					
Artist	artist	ARTIST	INTEGER								
Customer	customer	CUSTOMER	INTEGER								
Invoice Date	invoice_date	INVOICE_DATE	DATE								
Quantity	quantity	QUANTITY	INTEGER		×						
Tax	tax	TAX	CURRENCY			×					
Total	total	TOTAL	CURRENCY			×					
Track	track	TRACK	INTEGER		×						
UnitPrice	unitprice	UNITPRICE	CURRENCY		×						

Same mantra, we look at the database diagram. We “dive” deep from “Invoice Items” to “Tracks” to “Albums” and finally to “Artists” .

We are using the power Jam.py feature - lookups, as much as we can to minimise coding. Just imagine the SQL needed to “join” the three tables! To provide the exact SQL, here is what Jam.py generates automatically as the last SQL query when we open Invoices:

```
SELECT "DEMO_INVOICE_TABLE"."ID", "DEMO_INVOICE_TABLE"."DELETED", "DEMO_INVOICE_TABLE
↪"."MASTER_ID",
"DEMO_INVOICE_TABLE"."MASTER_REC_ID", "DEMO_INVOICE_TABLE"."TRACK", "DEMO_INVOICE_
↪TABLE"."QUANTITY",
"DEMO_INVOICE_TABLE"."UNITPRICE", "DEMO_INVOICE_TABLE"."AMOUNT", "DEMO_INVOICE_TABLE".
↪"TAX",
"DEMO_INVOICE_TABLE"."TOTAL", "DEMO_INVOICE_TABLE"."INVOICE_DATE", "DEMO_INVOICE_TABLE
↪"."CUSTOMER",
DEMO_TRACKS_43."NAME" AS TRACK_LOOKUP, DEMO_TRACKS_43_ALBUM."TITLE" AS ALBUM_LOOKUP,␣
↪DEMO_TRACKS_43_ALBUM_ARTIST."NAME"
AS ARTIST_LOOKUP, DEMO_CUSTOMERS_329."LASTNAME" AS CUSTOMER_LOOKUP FROM "DEMO_INVOICE_
↪TABLE" AS "DEMO_INVOICE_TABLE"
```

(continues on next page)

(continued from previous page)

```
OUTER LEFT JOIN "DEMO_TRACKS" AS DEMO_TRACKS_43 ON "DEMO_INVOICE_TABLE"."TRACK" =␣
↪DEMO_TRACKS_43."ID" OUTER LEFT JOIN
"DEMO_ALBUMS" AS DEMO_TRACKS_43_ALBUM ON DEMO_TRACKS_43."ALBUM" = DEMO_TRACKS_43_
↪ALBUM."ID" OUTER LEFT JOIN
"DEMO_ARTISTS" AS DEMO_TRACKS_43_ALBUM_ARTIST ON DEMO_TRACKS_43_ALBUM."ARTIST" = DEMO_
↪TRACKS_43_ALBUM_ARTIST."ID" OUTER
LEFT JOIN "DEMO_CUSTOMERS" AS DEMO_CUSTOMERS_329 ON "DEMO_INVOICE_TABLE"."CUSTOMER" =␣
↪DEMO_CUSTOMERS_329."ID" WHERE
"DEMO_INVOICE_TABLE"."DELETED"=0 AND "DEMO_INVOICE_TABLE"."MASTER_ID"=16 AND "DEMO_
↪INVOICE_TABLE"."MASTER_REC_ID"=464
ORDER BY DEMO_TRACKS_43_ALBUM."TITLE", DEMO_TRACKS_43."NAME"
```

There we go! We can easily copy/paste the above SQL query into some utility to browse the data in demo.sqlite database. Just imagine coding this type of query for all possible combinations of tables. Btw, this SQL extract is possible by changing the Jam.py source code.

Now we execute Tutorial. Part 3. Details

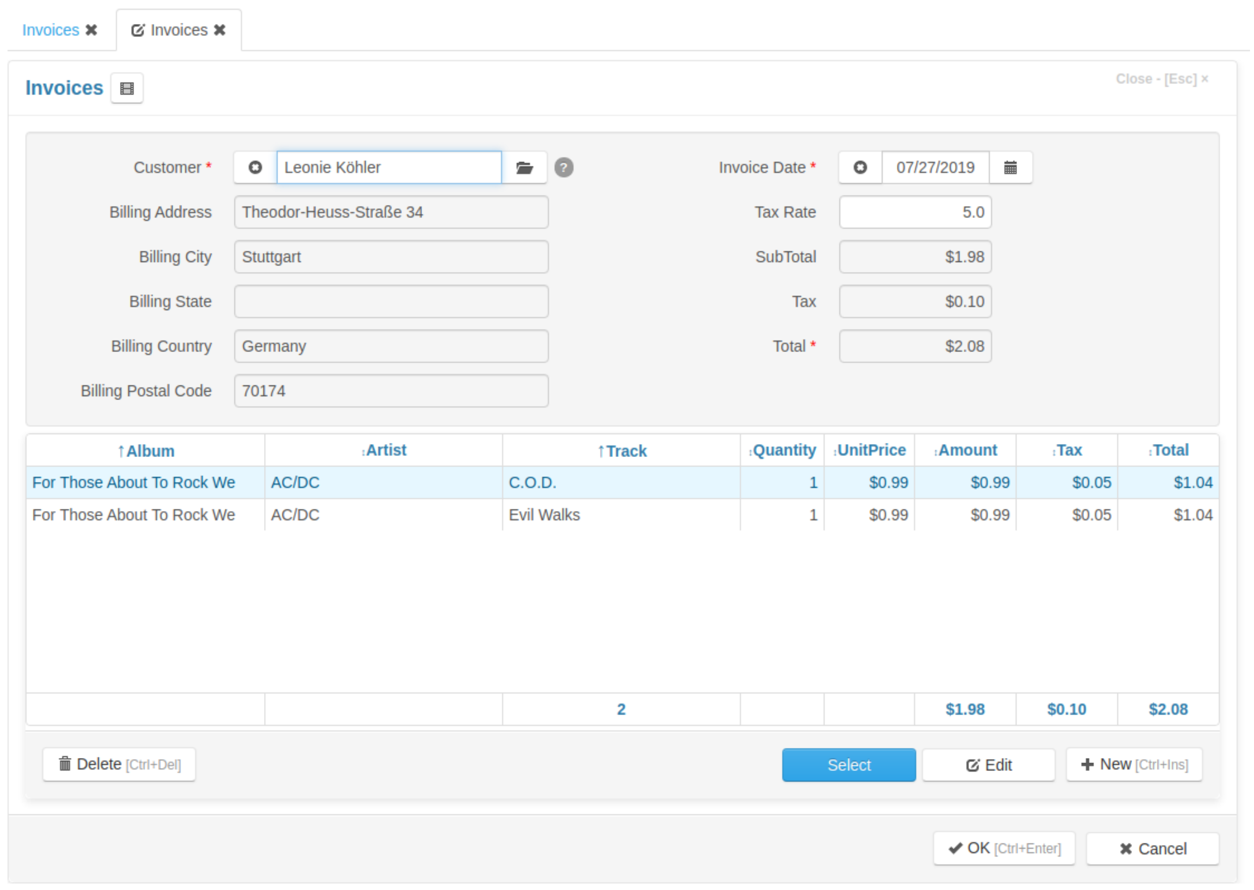
5.4.4 How did we go?

So, did we manage to get the Invoices with details up? It is quite common for the first timers to miss the "Details" button on the right hand side of Invoices in *builder.html*.

On the CRM example, the Details is a "to-do-list" . If all went ok, we should have a project page similar to Demo.

5.4.5 Click on!

With the expectations and basics covered, we can double click on any Invoices row. If all good, we see one invoice with the invoice items included.



Customer *	Leonie Köhler	Invoice Date *	07/27/2019
Billing Address	Theodor-Heuss-Straße 34	Tax Rate	5.0
Billing City	Stuttgart	SubTotal	$1.98
Billing State		Tax	$0.10
Billing Country	Germany	Total *	$2.08
Billing Postal Code	70174		

Album	Artist	Track	Quantity	UnitPrice	Amount	Tax	Total
For Those About To Rock We	AC/DC	C.O.D.	1	$0.99	$0.99	$0.05	$1.04
For Those About To Rock We	AC/DC	Evil Walks	1	$0.99	$0.99	$0.05	$1.04
		2			$1.98	$0.10	$2.08

If nothing was touched or changed, this is how it looks like. Job done. All created automatically, no code yet! Except for "Tax" and "Total" .

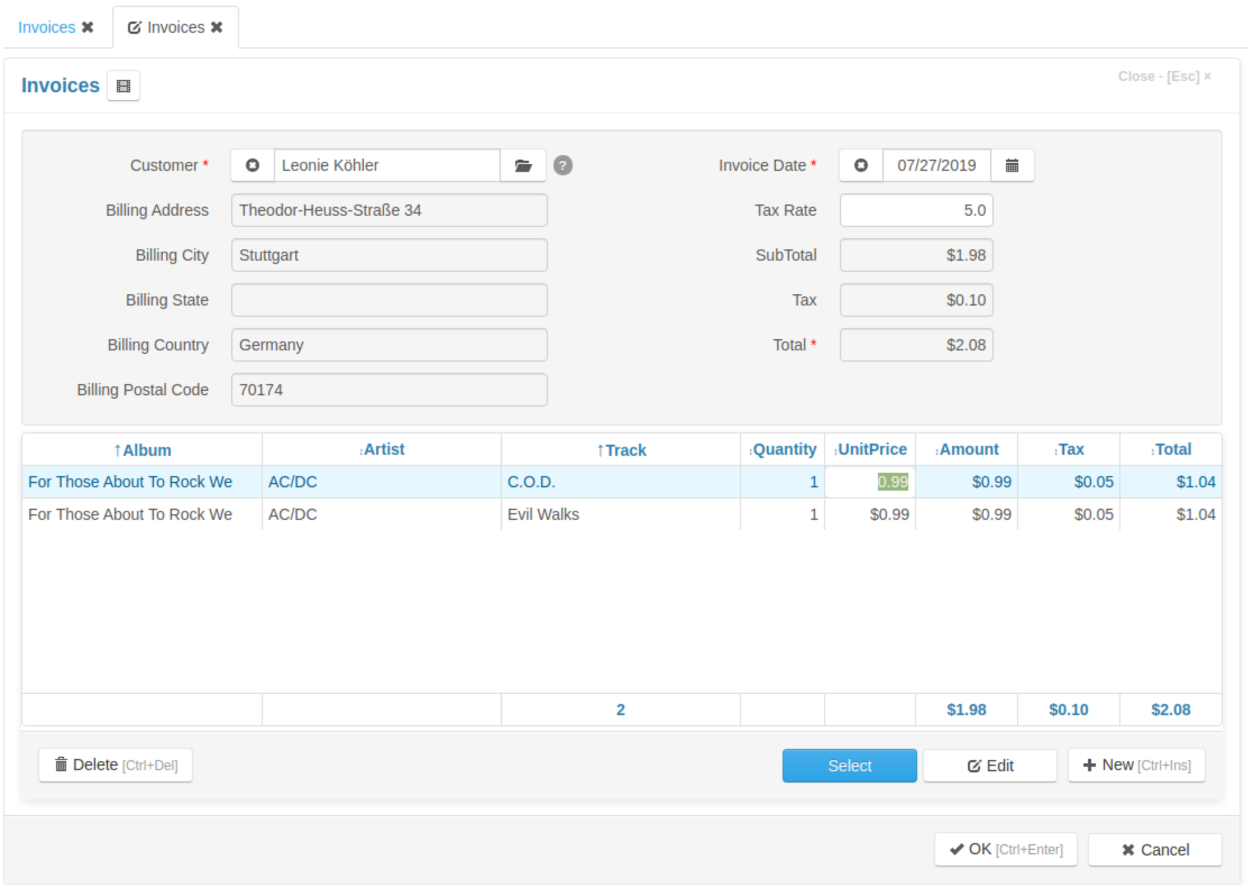
See how almost everything related to Customer is greyed out? This is because of the *Master field*! Only one field on the Invoices table is not using "Master field" and this is a Customer field. Hence, only that field won't be greyed out, because we are using it to define all other fields on the Form. For sure we are not updating the "Invoice Date" from anywhere, it is defined with a little code. Same with "Tax" , "SubTotal" and "Total" .

5.4.6 Your 1st task!

As mentioned, we can edit anything directly in the data grid! The changes will be picked immediately and saved in the database.

Your first task is to find the option which enables editing in the data grid.

It looks like this:

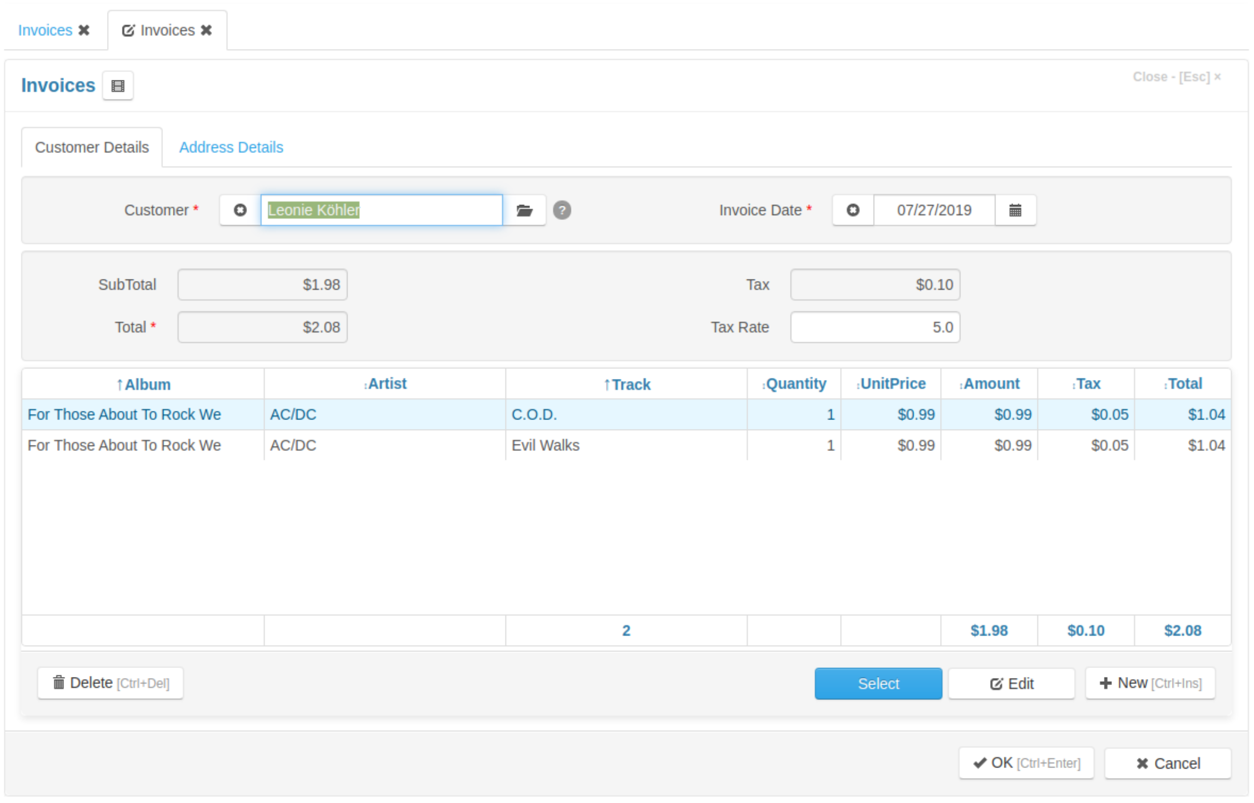


Customer *	Leonie Köhler	Invoice Date *	07/27/2019
Billing Address	Theodor-Heuss-Straße 34	Tax Rate	5.0
Billing City	Stuttgart	SubTotal	$1.98
Billing State		Tax	$0.10
Billing Country	Germany	Total *	$2.08
Billing Postal Code	70174		

Album	Artist	Track	Quantity	UnitPrice	Amount	Tax	Total
For Those About To Rock We	AC/DC	C.O.D.	1	0.99	$0.99	$0.05	$1.04
For Those About To Rock We	AC/DC	Evil Walks	1	$0.99	$0.99	$0.05	$1.04
		2			$1.98	$0.10	$2.08

And why not reshuffling the Invoices edit form a bit?

To look similar to this:



Customer *	Leonie Köhler	Invoice Date *	07/27/2019
SubTotal	$1.98	Tax	$0.10
Total *	$2.08	Tax Rate	5.0

Album	Artist	Track	Quantity	UnitPrice	Amount	Tax	Total
For Those About To Rock We	AC/DC	C.O.D.	1	$0.99	$0.99	$0.05	$1.04
For Those About To Rock We	AC/DC	Evil Walks	1	$0.99	$0.99	$0.05	$1.04
		2			$1.98	$0.10	$2.08

Very good, we are getting there!

I am not sure if more info is needed about the data grid and forms layout. Just note that the Menu is created from the *builder.html* Groups layout, so whatever is showing here first, it will be shown as first on the Menu.

If we quickly want to change the application starting view to ie Customers, we just change the Groups order.

Maybe we can go now through a code.

5.5 A little code

Before diving in to a code, remember the *ACE Code Editor shortcuts*!

If one is a beginner with any sort of coding, this might be a bit daunting. However, I can assure you, the developers out there are using copy/paste just like anyone else. So, with a bit of persistence, one can reuse the code from Demo application and get impressive results. This is particularly true for Dashboards, which is a flag Jam.py feature. We included more then 15-20 graphs for some applications, with a minimal change for each. For example, just changing "pie" to "bar" changes the graph appearance.

The official documentation has a heaps of code applicable to Demo application. Here will try to explain where exactly the code is used and a bit about why. Again, we touching only the additional code applicable for Demo, the default one which comes with a blank and empty project is not discussed.

5.5.1 Invoices

In Invoices, it is quite obvious that some calculations are happening "on the fly". At the moment, Jam.py v5 has no feature to calculate fields automatically, driven by visual application *builder.html*. The new major Jam.py version might include this option. This might be a put off for some users or a "would be" developers and the reasoning is that major players, like PowerApps, have that. Sure, they also have a bottomless financing, if you follow my drift.

Here is the tiny code for Demo application version 1.5.30. The code can be found on "Client module", after selecting Invoices:

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

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

function on_field_changed(field, lookup_item) {
    var item = field.owner,
        rec;
    if (field.field_name === 'taxrate') {
        rec = item.invoice_table.rec_no;
        item.invoice_table.disable_controls();
        try {
            item.invoice_table.each(function(t) {
                t.edit();
                t.calc(t);
                t.post();
            });
        }
        finally {
            item.invoice_table.rec_no = rec;
            item.invoice_table.enable_controls();
        }
    }
}

function on_detail_changed(item, detail) {
```

(continues on next page)

(continued from previous page)

```
    var fields = [
        {"total": "total"},
        {"tax": "tax"},
        {"subtotal": "amount"}
    ];
    item.calc_summary(detail, fields);
}

function on_before_post(item) {
    var rec = item.invoice_table.rec_no;
    item.invoice_table.disable_controls();
    try {
        item.invoice_table.each(function(t) {
            t.edit();
            t.customer.value = item.customer.value;
            t.post();
        });
    }
    finally {
        item.invoice_table.rec_no = rec;
        item.invoice_table.enable_controls();
    }
}
```

As seen, the Client module contains five JavaScript functions. First two functions deal with text formatting. The *on_detail_changed* is using Jam.py built in function calc_summary.

Lastly, one of the most important function and the most commonly used is:

on_field_changed()

```
function on_field_changed(field, lookup_item) {      <- 1.
    var item = field.owner,                          <- 2.
        rec;
    if (field.field_name === 'taxrate') {            <- 3.
        rec = item.invoice_table.rec_no;
        item.invoice_table.disable_controls();       <- 4.
        try {                                        <- 5.
            item.invoice_table.each(function(t) {
                t.edit();
                t.calc(t);
                t.post();
            });
        }
```

(continues on next page)

(continued from previous page)

```
        finally {                                           <- 6.
            item.invoice_table.rec_no = rec;
            item.invoice_table.enable_controls();   <- 7.
        }
    }                                                   <- 8.
}
```

Understanding this function is quite important. It provides a mechanism to control what happens when some **input** on the application is **changed**.

Steps:

1. In this case we are looking to "monitor" the "taxrate" field, because it will affect all Items tax after changing it. We need to change the relevant *Lookup fields*. Which is a lookup to a different table, right? We are changing "Tax Rate" in Invoice table, but at the same time expect the changes in "Invoices Items" .

2. We define some "shortcuts" here. See how *item* is repeating in the code? Hence, a "shortcut" .

Note: When copy/paste the code, it is not obvious that "item" is not there as it should be. So the best is to look at the code we *know* that is working. Like Demo/Customers Client Module, etc.

Consider this example:

```
function on_field_changed(field) {
        if (item.field_name === 'pattern_type') {
        .
        .
```

That is not going to work. Simply because it is missing **"var item = field.owner"** .

3. This is where the magic happens. We "test" the field name if is the required one. If not, nothing happens and goes straight to Step 6, which is "the end" . Because this is typical "if" clause, better use "try" and "finally" in it, steps 4. and 5. respectively.

With **"rec = …"** we define all records needing changing, after the "taxrate" changes. With **"…disable_controls"** we disable all buttons (control items) temporary, as we do not want something actioned on while working on changing records.

4. Disable DOM controls temporarily. We are doing this to significantly speed up the displaying of data.

5. Then we "try" to update all records with a function **".each(…)"** . Which does "edit" , "calc" and finally "post" . We use "edit" to open every single record for editing, and "post" to post everything back via API. This is the "POST" part. Without it, the data would not be saved.

Note: The function - "calc" , does the actual calculation on records. The function is in "Journals/Invoices/InvoiceTable"

Client Module.

---

6. Enable DOM controls.

7. And "finally", we show the results back with **"item…= rec"** and enable the buttons, etc. with **"…enable_controls"** .

8. If the field name was not the one we are after, exit the "if" clause here.

That is it. With above steps, we created the JavaScript calculations.

Note: No calculation will happen with a field valued as null.

It is absolutely needed to check the condition before expecting that the calculation will go through.

5.5.2 Error handling!

As with above *null*, when doing something, it is always advisable to use *best practises*:

Note: One of the main obstacles with using **"on_field_changed"** is a infinite loop happening. For example one field changing the field we are expected to "monitor" .

The following algorithm can be used to avoid this situation:

```
let calculating;

function on_field_changed(field, lookup_item) {
     if (!calculating) {
        calculating = true;
        try {
              // some calculations
        }
        finally {
            calculating = false;
        }
    }
}
```

For more information please visit the mailgroup error thread.

5.5.3 So, how was it?

Is the above too much? Or easy to follow and apply in some other scenario?

To expand a bit on **"on_field_changed"** , please visit Conditional formatting mailgroup thread or search the mailgroup for the same. There are a number of scenarios where to apply **"on_field_changed"** , and this is just a touch of the surface. One possible scenario is changing the password. Again, the best is to search the mailgroup.

5.6 More code

As mentioned, the *calc* function exists outside the code used in Invoices Journal. Why? Because there might be many Details in some Journals, and placing them in one single place is not flexible enough. Simply put, an function, for example **on_field_changed**, would overwrite the same function if there is a need for two same functions.

5.6.1 Invoices Details

Here is what is in the Journals/Invoices/InvoiceTable, note the *on_field_changed* function again:

```
function calc(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;
}

function on_field_changed(field, lookup_item) {
    var item = field.owner;
    if (field.field_name === 'track' && lookup_item) {
        item.unitprice.value = lookup_item.unitprice.value;
    }
    else if (field.field_name === 'quantity' || field.field_name === 'unitprice') {
        calc(item);
    }
}

function on_view_form_created(item) {
    var btn = item.add_view_button('Select', {type: 'primary', btn_id: 'select-btn'});
    btn.click(function() {
        item.alert('Select the records to add to the invoice and close the from');
        item.select_records('track');
    });
}

```

(continues on next page)

(continued from previous page)

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

It needs to be stressed that *details* can “live” with absolutely no code on Details tree within the *builder.html*! However, when we *attach* the detail to some Journal (Tutorial. Part 3. Details), probably some code is needed there. And that is exactly the above example.

Because the details store *Invoices Items*, so basically products, the *calc* code exist here and only here. It is only relevant to Items, right?

The new functions presented here are:

`on_view_form_created``()`

`on_after_append``()`

It is quite obvious what the function **on_view_form_created** does. It creates a “Button” and assigns a function to it so when the button is clicked, it shows the JavaScript message and presents a form to select some products. All of this in just five lines of code.

Function **on_after_append** just adds the current date on a form.

We are almost there with the final Demo Invoices code! Hold on!

5.7 Server code

This is where Python programming language kicks in. Everything in Jam.py as the Server code is Python. Just like the VBA for Access!

5.7.1 Invoices

The last code remaining for Invoices is the Server Module code.

If we click on Journals/Invoices “Server module” as on *builder.html*, this is the code:

```
def on_apply(item, delta, params, connection):
    tracks = item.task.tracks.copy(handlers=False)
    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():
```

(continues on next page)

(continued from previous page)

```
                changes[t.track.value] += t.quantity.value
            elif t.rec_deleted():
                changes[t.track.value] -= t.quantity.value
            elif t.rec_modified():
                changes[t.track.value] += t.quantity.value - t.quantity.old_value
    ids = list(changes.keys())
    tracks.set_where(id__in=ids)
    tracks.open()
    for t in tracks:
        q = changes.get(t.id.value)
        if q:
            t.edit()
            t.tracks_sold.value += q
            t.post()
    tracks.apply(connection)
```

Here is where quite important function is introduced:

on_apply()

There are many on_apply occurrences in the official documentation since it is the same function name for the Client and Server operations, JavaScript and Python, respectively.

It is fairly small chunk of code, which mainly deals with changes, inserts or deletion of Items. The code is relevant for *Demo application version* at the time of writing. It can be used for many scenarios if slightly modified.

Not quite sure how to dive into the above code with the explanations with keeping it simple. The official documentation might be sufficient. However, why doing it we might ask?

5.7.2 Why Server Code?

Note: To answer the above question, the *Server Code* is needed because it is happening in the background. It is almost exactly as the `stored procedure`, the relational databases are using. It is absolutely possible for an application not to use the *Server Code* at all, hence no need for Python. This is the application design choice. If needed, Jam.py can also use the database *stored procedures*.

The relational database engines are quite efficient with the stored procedures. But, and there is a but, is the stored procedure portable to a different database products? Of course not. The good news is, Python is portable. Hence, with using the *Server Code* Jam.py can successfully run on any platform.

With some other database engines, or even the Front End Applications like MS Access, for the background operations the Visual Basic might be used. With MSSQL Server, the database stored procedure can be executed from Access. Either way, the code is needed and it would not be portable to a different platform. It is "locked in" for one and only one database product.

Jam.py completely eliminates the above. There is no "lock in" with any specific database product. The application can be developed on any supported database, and can be moved to any supported database.

There is also a question of security. It is quite simple to implement the code which controls the permissions on what can be executed by the user. This is almost the Desktop Application territory!

And there are even more good news! Debugging.

5.7.3 Debugging

To debug the *stored procedure* within the relational database, is not for faint hearted. Every developer or the DBA would agree. Of course, the more experienced professionals would protect their turf. Again, it is business after all.

However, for a "would be" developer, Python is way easier to learn and debug. And for the professionals, it is a breeze.

How to debug is completely down to Python experience, and leaving it to the reader to consult official Python resources.

5.7.4 And finally…

Below SQL will check for any code in the Application Server Modules (using sqlite3 command line utility):

```
sqlite3 admin.sqlite "select id, f_name,f_server_module  from sys_items where  f_
↪server_module!='' and f_server_module is not null"
```

Below sql will check for any code in the Client Modules:

```
sqlite3 admin.sqlite "select id, f_name, f_web_client_module from sys_items where  f_
↪web_client_module!='' and f_web_client_module is not null"
```

Of course, the Developer would need access to the admin.sqlite database to run the SQL against it. The SQL does not take into account commented code.

Here is the list where all *Server code* is on Demo application, hence it can be easily identified within the *builder.html*:

```
1|Jam.py Demo
5|Reports
13|Genres
15|Tracks
16|Invoices
19|Print invoice
20|Customer purchases
22|Customer list
25|Mail
```

Note: Note the *ID*! That is exactly the same ID as seen on *builder.html*! The *Genres* has ID=13, *Tracks* has ID=15, and so on. Hence, we know how many are there and exact location for each Server Module used within the application.

The commented out code, meaning not used at all for Demo, is code with ID=1 (Jam.py Demo). The code should be uncommented when the built in Authentication is turned on.

5.7.5 …the End of Code

Hope you've enjoyed this Code part! The intention was to explain some bits and pieces which were exclusively developed for Demo application. For sure there is more code as seen on the above SQL output! However, half of that number is related to Reports. And we are not going to Reports just yet.

5.8 index.html

Here we are introducing the basic index.html structure.

5.8.1 Templates

Demo application has the "About" and "Jam.py" on the right hand side of the Menu bar. This is a Code as well.

However, it is a bit different to what was discussed above, since the code is related to the index.html file. Btw, this is true for everything else. Everything is based on it. It is the **root** file on any application in Jam.py release 5 or older. In the newer release of Jam.py, the file template.html is introduced. More about that latter on.

If we open index.html file in *builder.html* (Task/Project), the below is visible:

```html
<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="admin"><a href="#">Application builder</a></li>-->
            <li id="about"><a href="#">About</a></li>
            <li id="jam"><a href="http://jam-py.com/" target="_blank">Jam.py</a></li>
        </ul>
    </div>
</div>
```

Jam.py calls this html block a Template.

The "About" and "Jam.py" is simply a link, just as "Application builder" is. Which is commented out.

Hence, everything we want to see here is just a matter of adding it as extra lines. It is quite clear that this Template is related to a "menu", and it will display on the right hand side. It clearly indicates *menu-right*! The id= is the most important one, here we see **id admin**, **about** and **jam**.

However, this itself is not enough for the proper application functioning. Why? Because the *Template* does not exist on it's own. For example, in Django or Flask, the Template might be using a "built in" functionality, like "for loop" clause. Jam.py does not do that. It does not contain any code for Template in the *index.html* file.

This must the stressed, in Jam.py, there is no need to "program" the Template. It is a simple html. Ingenious.

5.8.2 Template Code

The Template Code for "About" and "Jam.py" is in demo.js (Project/Task/Client Module). It is the default code available for any new project:

```
$("#menu-right #admin a").click(function(e) {
    var admin = [location.protocol, '//', location.host, location.pathname, 'builder.
↪html'].join('');
    e.preventDefault();
    window.open(admin, '_blank');
});
$("#menu-right #about a").click(function(e) {
    e.preventDefault();
    task.message(
        task.templates.find('.about'),
        {title: 'Jam.py framework', margin: 0, text_center: true,
            buttons: {"OK": undefined}, center_buttons: true}
    );
});
```

The **admin** link will just open the new tab in the browser, with the *builder.html* page. Cool. So this is how we open any page, ie a "Help" page for the application. Of course, we would need a help.html file. Try it. Create help.html within the application folder and replace builder.html with help.html! Obviously, the *commented* code prevents this from happening. Uncomment it first.

For **about** link, we also see:

```
task.templates.find()
```

The function is used to find the **correct** *Template* id. In this case the below "about" block in *index.html*, since the *id=* about, which is the **class** name "about":

```
<div class="about">
    <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␣
```

(continues on next page)

(continued from previous page)

```
↪Database</a>
    <p>by Andrew Yushev</p>
    <p>2015-2017</p>
</div>
```

That is all to get us going. It should be easy to change the Menu content, and open some pages or a dialogue.

5.8.3 Wrapping up

Congrats! Almost everything is covered regarding the Demo functionality! Hope it was not that difficult.

5.9 Dashboard

Now that we' ve covered the Data Grids functioning, maybe it' s a time to have a look at the Demo Dashboard. First to understand is the *Template* in *index.html* for the Dashboard, and Dashboard only.

5.9.1 Dashboard template

The *Template* for Demo Dashboard is a simple two columns table:

```
<div class="dashboard-view">
    <div class="row-fluid">                                             <-- the table   -->
        <div class="span6">                                             <-- column #one  -->
            <canvas id="cutomers-canvas"></canvas>                      <-- graphics one  -->
            <div id="customer-table"></div>                             <-- table under it -->
        </div>
        <div class="span6">                                             <-- column two  -->
            <canvas id="tracks-canvas"></canvas>                        <-- graphics #two  -->
            <div id="tracks-table"></div>                               <-- table under it  -->
        </div>
    </div>                                                              <-- end of table-->
</div>
```

This obviously means one thing: **we need a heaps bigger table if we are going to implement many Pie charts or similar!**

Hence, the below Template would create two rows with two columns table:

```
<div class="dashboard-view">
    <div class="row-fluid">
        <div class="span6">
```

(continues on next page)

(continued from previous page)

```
            <canvas id="assets-canvas"></canvas>
            <div id="assets-table"></div>
        </div>
        <div class="span6">
            <canvas id="parts-canvas"></canvas>
            <div id="parts-table"></div>
        </div>
    </div>
    <div class="row-fluid">
        <div class="span6">
            <canvas id="parts_table-canvas"></canvas>
            <div id="parts_table-table"></div>
        </div>
        <div class="span6">
            <canvas id="parts_suppliers-canvas"></canvas>
            <div id="parts_suppliers-table"></div>
        </div>
    </div>
</div>
```

The above example is from here second Demo.

The *Demo Template* above has an typo: **cutomers-canvas** should be **customers-canvas**. We can observe that this is related to a **customer-table**. Same with **tracks-canvas**, it is related to a **tracks-table**.

With this two pieces of information, we can build the JavaScript code to show the data. We just need to remember that *index.html* file must include:

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

The file Chart.min.js can exist elsewhere and not in static/js, however, this is where it normally resides.

5.9.2 Dashboard Menu Item

Before we dive into the JS code, it is visible on Demo that the Dashboard exist as an Item on the Menu, called Analytics. This is just an Project/Task/Groups Item on *builder.html*, where Analytics was created as an place holder for Dashboard virtual table! Now, what is a virtual table we will cover latter on, but just think of it as virtual - it does not exist in the database.

So, when creating a Dashboard from scratch, just replicate what Demo has, one Analytics (or whatever we need to call it), Group Item and one Dashboard virtual table within that Item. Simple. Takes 30 seconds to create!

5.9.3 Dashboard Code

As we know, Jam.py uses ChartJS as the Dashboard engine. However, that does not mean much when working with the Jam.py. It is really easy, though.

If we look at the Demo *builder.html*, and click on Analytics/Dashboard/Client Code, the code has approx 100 lines. But only half of it is actually related to the application!

The rest is copy/paste! Huh, that is assuring. Cough, cough!

First, remember the above Dashboard Template, **customers-canvas** and **tracks-canvas**?

To start working on the Dashboard, we need to define this two canvases:

```
function on_view_form_created(item) {
    show_cusomers(item, item.view_form.find('#cutomers-canvas').get(0).getContext('2d
↪'));
    show_tracks(item, item.view_form.find('#tracks-canvas').get(0).getContext('2d'));
}
```

The exact *typo* exist here as well! The code tells the Application to look at the *index.html* for this two canvases, and when found, the JS function executes the **show_cusomers** and **show_tracks**, respectively.

Hence, when building more graphics, we would need more functions. From above second Demo, the functions to show four graphics are:

```
function on_view_form_created(item) {
    show_assets(item, item.view$('#assets-canvas')[0].getContext('2d')),
    show_parts(item, item.view$('#parts-canvas')[0].getContext('2d'));
    show_parts_table(item,item.view$('#parts_table-canvas')[0].getContext('2d'));
    show_parts_suppliers(item,item.view$('#parts_suppliers-canvas')[0].getContext('2d
↪'));
}
```

Each JS function normally pulls data from one table only. However, the ChartJS is quite effective and does *Lookup fields* too! So no wonder Jam.py is using this library when it nicely utilises the power of lookups!

```
function show_cusomers(item, ctx) {                                     <-- step 1
    var inv = item.task.invoices.copy({handlers: false});               <-- step 2
    inv.open(                                                           <-- step 3
        {
            fields: ['customer', 'total'],
            funcs: {total: 'sum'},
            group_by: ['customer'],
            order_by: ['-total'],
            limit: 10
        },
```

(continues on next page)

(continued from previous page)

```
        function() {                                                   <-- step 4
            var labels = [],
                data = [],
                colors = [];
            inv.each(function(i) {                                     <-- step 5
                labels.push(i.customer.display_text);
                data.push(i.total.value.toFixed(2));
                colors.push(lighten('#006bb3', (i.rec_no - 1) / 10));
            });
            inv.first();                                               <-- step 6
            draw_chart(item, ctx, labels, data, colors, 'Ten .....'); <-- step 7
            inv.create_table(item.view_form.find('#customer-table'),  <-- step 8
                {row_count: 10, dblclick_edit: false});
        }
    );
    return inv;                                                        <-- step 9
}
```

Steps:

1. This is where we define the function name. It must be exact name as in the above function: on_view_form_created Yes, the same *typo* in the name! We try to logically name the functions as the name of the database tables.
2. The magic happens here. We define the **var inv** because the table is **invoices** table! Very important, almost all can be used with copy/paste if we just rename a few instances of **inv** and **invoices**. Taking into account that the table fields will possibly change as well! Do not care for handlers: false just yet.
3. Remember, the **var inv** is just a "shortcut" (I call it like that, because I like it). Hence, **inv.open** will open the table **invoices** for access. By opening, we are actually fetching a few fields (*Chinook* database fields) :

    - customer and total

    We also using a "built in" ChartJS Function, funcs:

    - to summarise total with function 'sum'

    Can we read the rest what is happening? Order and limit? I hope so. It is quite simple, right?
4. This code is copy/paste, it is needed for ChartJS.
5. Remember to adjust **inv**, **customer** and **total** for your needs. This is also where *Lookup fields* kicks in, the **customer.display_text** will display the Customer Name and not the CustomerId as an Integer, which is visible on the Invoices table. Also worth mentioning is **total.value.toFixed(2)**, which is obviously a value of Total, rounded to two decimals. Since we dealing with 1/10 of data, some calculations is needed with **(i.rec_no - 1) / 10)**.
6. Start from first. Remember **inv**!
7. Draw graphics with 'Ten most active customers' Label!

8. Create table with some data under the graphics within the **customer-table** Template! Remember to adjust for the rest of Templates!
9. And finally pass back the Invoices data.

So there you go! With one single block of code we can create indefinite number of graphics utilising *Lookup fields* within the ChartJS!

5.9.4 Cosmetic code

Now that we know how to create a graphics for one table, the rest is a copy/paste! Functions **draw_chart** and **lighten** is just cosmetic. Play with it.

Note: Changing a code in function draw_chart from a ‘pie’ to ‘bar’, will change the graphics!

5.9.5 End of Dashs

Once we get a handle of it, creating a Dashboards is a breeze! It is just a matter of adjusting the names of tables and variables. Plus, how much data we are presenting, one tenth of it, or one third, etcetera. Of course, there is a bit more about it, but we do not want the information overload! Hope that this is just enough to get going!

Note: To Be Continued…

CHAPTER SIX

MS ACCESS MIGRATION

Here we touch base on Access migration to Jam.py.

6.1 MS Access migration

Congratulations on the decision to *migrate* MS Access to the Web!

After all, it is moving to the modern JavaScript with Bootstrap and jQuery technology Front End! Which is powered by Python.

Jam.py is not exclusively developed for Access migration though. Just like Django (Python) or CodeIgniter (PHP), is not. Both mentioned products are not specialised for the databases oriented applications, even though one could migrate Access to any of those.

However, Jam.py is specialised. Just like Access is.

Before considering migrating Access (or any other proprietary software), to the Web with the help of Jam.py, please review some answers below. Which might help with understanding what to expect from migration to Jam.py.

6.2 Top Migration Questions

The MS Access might be a *bread and butter* for many developers out there. Particularly in some specific industries like the Law practices or similar. And fair enough. It is the right tool for the job.

However, the **risk** is that in some years to come, the developers will retire. I know, I' m one of them. Hence, where will the business find people to maintain their expensive application? Maybe they already hired the MSSQL database administrator, since the Access developers told them to do so. The DBA' s might be around in years to come, but the Access developers? Or better, THE Access itself?

Hence, the faster we move away from proprietary database AND Front End application, the better. Simply because the technology will be around way after the MS Access is gone. This is business after all, let' s put aside the emotions. The HTML is here to stay. Who remembers the Netscape any more? Or Mosaic?

6.2.1 1. Complacency

Success breeds complacency. Complacency breeds failure. Only the paranoid survive. Andy Grove

About being paranoid, many MS Access Front End applications are not encrypted! This means the IP (Intellectual Property), is not protected. Even worse! The VBA code might contain email passwords or similar. For example: https://wellsr.com/vba/2020/excel/vba-send-email-with-gmail/

Is your application cyber safety compliant?

The application might be compliant for some business, however does it run on the Web? Apple Mac? Tablet? That is exactly what we are also addressing with the migration. The *Future* with the *IP and Cyber Safety*.

6.2.2 2. VBA?

Does your application rely on heaps of VBA for Business Rules (BR)?

The VBA can’ t be migrated to the Web. The question is what is it used for? 5000 lines of VBA code might be a dinosaur code! Is it possible to use Python with 200 lines of code? How about 10 lines?

In addition, what might not be recognised is the speed. The VBA just does not compare with Python. The Developers will argue that this is due to the quality of code. Not quite true. The code is just as important as garbage collection, or utilising memory, etc. And this is where Python shines. It replaces massive technological debt with a few lines of code.

For example, the BR might be a way the application authenticates users (ie. a table with username, password, role). Jam.py can reuse the Access tables used for users, so no issues with that. When the database is migrated, the users authentication table is there and we can use exactly twelve lines of Python code, as seen on Demo, to authenticate users with the roles.

Also worth mentioning VBA to Python –10 Simple Python vs VBA Examples. The article touches on some differences and interesting points from a seasoned VBA developer. However, for building the Jam.py application, JavaScript is used as a Front End. Hence, the real difference is between JS and VBA. For example, showing the MessageBox or conditional formatting and functions.

On that note, the above article touches on VBA code for MessageBox:

```
Sub MessageBox()
    'Information box
    MsgBox "Hello", vbInformation, "Information"
    'Yes / No question
    If MsgBox("Do you like this tutorial?", vbYesNo, "Question") =␣
↪vbYes Then
        Debug.Print "They like it!"
    Else
        Debug.Print "They don't like it!"
```

(continues on next page)

(continued from previous page)

```
    End If
End Sub
```

In JavaScript, Jam.py does that in a similar way:

```
var btn = item.add_view_button('Set invoice paid', {type: 'primary',␣
↪btn_id: 'paid-btn'});
    btn.click(function() {
    item.question('Was the invoice paid?', function () {
            item.edit();
            item.paid.value = true;
            item.post();
            item.apply(true);
        });
    });
```

The code above actually does more then displaying the MsgBox! It will update all selected records in one go. Imagine writing the VBA code for the same task.

The MessageBox with JS in Jam.py is something like this:

```
item.alert('Successfully sent the mail');
```

The button in Jam.py is something like this:

```
var btn = item.add_view_button('Set invoice paid', {type: 'primary',␣
↪btn_id: 'paid-btn'});
    btn.click(function() {
            -- do something here --
        });
```

6.2.3 3. Excel?

Does your application depends on Excel or some other Office products?

This might be a show stopper, simply because the organisation will not let go of Excel. This might be particularly true within some Financial institution. The good news is Python can read/write Excel document formats. For sure Access has better interoperability with those products. It is the Desktop application after all. However, some operations within Access using Excel, might take a long time, sucking up the resources. Not so with Python and Excel. It is super easy to integrate Excel with Jam.py application using DataTables.

Good resource for using Excel with Python.

6.2.4 4. Queries

Most of the “applications” consist of a single hard-coded query or a single linked table.

The above is a quote from Front-End for MS Access migration? It is an old thread with all points still valid. With Jam.py, one does not write queries. It is absolutely possible to do so, if needed.

It is a no-code, low-code or more-code RAD framework after all. And that can open the whole new world for a group of developers! Because each of them can work simultaneously on some other part of application, being the Forms or the Server procedures, everything is instantly accessible to everyone with the right privileges.

Hence, everything is simplified. The Jam.py framework will build complex queries just like Access does, with a few clicks of the mouse. Contrary to the Access, this query, in fact the complete application, will work on any supported database. Develop the application and deploy it to preferred database. Simple. No need to rewrite the queries.

6.2.5 5. Primary Keys

Be assured, there are a number of applications out there with no Primary Keys in Access.

The Primary Key (PK), is a must with all relational databases. With no PK, Jam.py cannot reference the Foreign Keys (FK). Not only relevant to Jam.py, Django does not do that either. Since both are sharing similar ORM. So no matter how we migrate Access to the Web, with which technology, the PK simply must exist. This is even true with Access web applications and web databases.

6.2.6 6. Deployment

If possible, have the front-ends copied locally on each workstation, for performance reasons.

Again, the same source Front-End for MS Access migration? See how nothing really changed from year 2008?

Some Access Developers would argue that RDP or Citrix or even VPN is the way to deploy the Front End application to the Web. All this technology was around in 2008. Nothing new.

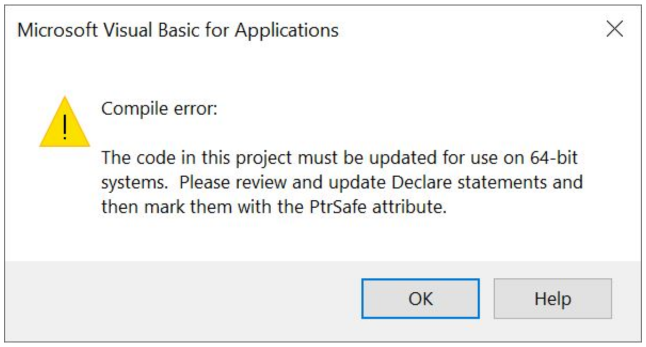
What is new is the Cloud.

Back to topic. Regarding the Access packaging, some people are using one distributable file, sometimes protected with the encryption. Jam.py can do exactly the same. Even more, one can use SQLCipher to protect the IP. Combining both within a portable application, which can run from, for example Windows x64 with no installation at all, is a powerful deployment and secure method. With such approach, the application database might be off-site or even embedded and encrypted with SQLCipher. This method can provide safe and secure access even from a portable drive in an off-line environment, for example with no Internet access at all. Only the local area network is needed for more then one user.

Regarding Web servers deployment, Jam.py application can be deployed in a few minutes on almost any popular Web server. Which can serve thousands of users simultaneously. Jam.py flagship application is supporting 2000 remote locations, each with 10-20 users. There is no need for installation of anything and costs almost nothing.

6.2.7 7. Compile Error

The code in this project must be updated for use on 64-bit systems…


Microsoft Visual Basic for Applications

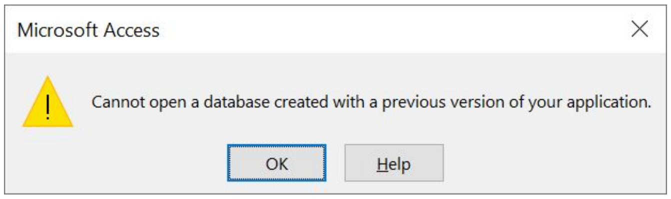
Compile error:

The code in this project must be updated for use on 64-bit systems. Please review and update Declare statements and then mark them with the PtrSafe attribute.

Self explanatory error.

6.2.8 8. Cannot open the database

Cannot open a database created with a previous version of your application.


Microsoft Access

Cannot open a database created with a previous version of your application.

Not so self explanatory error.

6.2.9 Conclusion

Hope the above sparked some interest. Even though there are probably more reasons why one should or should not try to migrate Access application to the Web. With or without using the migration tool as below is providing.

6.3 Where from here?

To successfully migrate MS Access to Jam.py, the assumption is that at least some knowledge exist with using the Jam.py Application Builder interface. Or, if feeling confident, one can start creating everything from scratch, and then load the data into the tables manually. We think that the below procedure is way more faster method. And, it does not cost anything.

If agreed, please visit:

https://jampyapplicationbuilder.com/conv/

After the Access file is uploaded and converted with no errors:

- link to the Application Builder will be emailed, which is using the sqlite3 data converted from Access instantly available on the Web! It is super easy and takes no time! **If not interested at all in building the FrontEnd Application**, please click on *Export* to download the DB as zip file which contains the sqlite3 file. The converted sqlite3 database has a long name exactly the same as in provided the link. This is to increase safety and security for all downloads.

Note: If interested in instantly showing data on the Web, meaning using Jam.py, please continue with the below:

The Application Builder from the provided link will set the Database to "DB manual mode". This means your new database is safe from changing the structure, like incidentally deleting the table or similar. If you would like to continue migrating Access to Web with the help of Jam.py framework, import all tables into the "Test Import" Group (click on Test Import/Import button), taking care of all needed DB columns correction or missing info. All fields must have a datatype if not already recognized. The "DB field name" should not be touched, the "Caption" is normally MS Access field caption, and the "Name" must be valid JavaScript naming standard, in short, no spaces, special characters, etc. To help out with the process, here are some videos:

Video overview - Jam.py VS MS Access

Video tutorial - How to publish MS Access Application to Web using Jam.py and SQLite3

Video tutorial - Simple CRM built automatically with Selenium IDE in 2.45minutes, demonstrates the Tutorial. Part 1. First project

New style Demo - Jam.py Demo Application

Personal Account Ledger - Migrated from MS Access template

Northwind Traders - Northwind Traders Migrated from MS Access template

After Importing some tables, just replace **builder.html** with **index.html** from the link to access your Application. Sometimes the index.html page will not show anything due to Load Balancer issues. Please try after some time. Or start over. One can even set the App password or create users within the Builder. The default App/Builder username/password is admin/111, change it on Project/Users and set Project/Parameters/Safe Mode to restrict access, etc. One can set all lookups to different tables as usual. It should be a fully functional App with all the bells and whistles!

It is important to note that once the Access file is converted and all tables Imported through the builder.html, it is not needed to use this service at all. Immediately use the **Export [CTRL+E]** option and download the Export file. Then, install the Python Jam.py framework locally or on the server, create the New application, Import the same downloaded file, and point the Application to sqlite database as per creating the New application. That is all.

6.3.1 Notes/Issues

This tool is for the database tables only, it does not convert MS Access file with no tables in it. Hence, **no need to upload a Front End application**, which probably connects to SQL Server or some other database(s). Output will be empty, or it will contain only a few tables. **The ODBC linked tables are supported.**

The process is expecting that the Primary Key exists on all tables. The simple INTEGER is not a Primary Key. The Log File will indicate this. Please address and fix this issue before attempting the conversion. Use only one *Numeric Field* as the Primary Key, ie. **AutoNumber field**.

MS Access might have a **special character, space, slash/backslash, dash (-),quote(‘)**, etc for the **table field, as well as table name**. The conversion will try to replace those, however, we would advise to fix the issues in MS Access, before attempting the conversion.

The conversion will fail if there are **non-ODBC linked tables in MS Access**. Please remove any linked tables before attempting the conversion. The ODBC linked tables are supported.

The conversion will fail if Access is using utf-16-li, no need to try again. No log or link will be produced and we suggest fixing Access.

There might be an #Error in the Access field, please fix it if conversion fails, and attempt again. **The best is to inspect the Log file, sometimes clearly showing the error**. If the log file is empty, something went wrong. No need to try again if Access wasn’t fixed.

Please do not upload confidential, private or **password protected/encrypted database**. If conversion to SQLite3 is needed for such data, the tool can be provided on demand. Password protected database conversion might fail, no need to try again.

6.3.2 Terms and Conditions

“This site and its components are offered for informational purposes only; this site shall not be responsible or liable for the accuracy, usefulness or availability of any information transmitted or made available via the site, and shall not be responsible or liable for any error or omissions in that information.”

6.4 Now what?

First of all, thanks for trying the migration!

Now that the database is converted to SQLite3, let' s see what we can do with it. The migration utility just speeds up the process of getting the data on the net. It does not build the Front End Application, even though we could easily automate this step.

But then, the Front End **look and feel** is subjective. If replicating the exact MS Access form as close as possible, it really takes no time to do it manually.

Before doing that, the best is to have a look at the MS Access Relational Diagram. It should clearly show the relations between the tables, as seen on the Chinook database.

For Access application with no Relation Diagram within it, it is still possible to look at the queries design. Each query would probably mean some sort of relation between two or more tables. Hence, in Jam.py, it is possible to create the Lookup fields to other tables just like MS Access does.

6.4.1 Tables

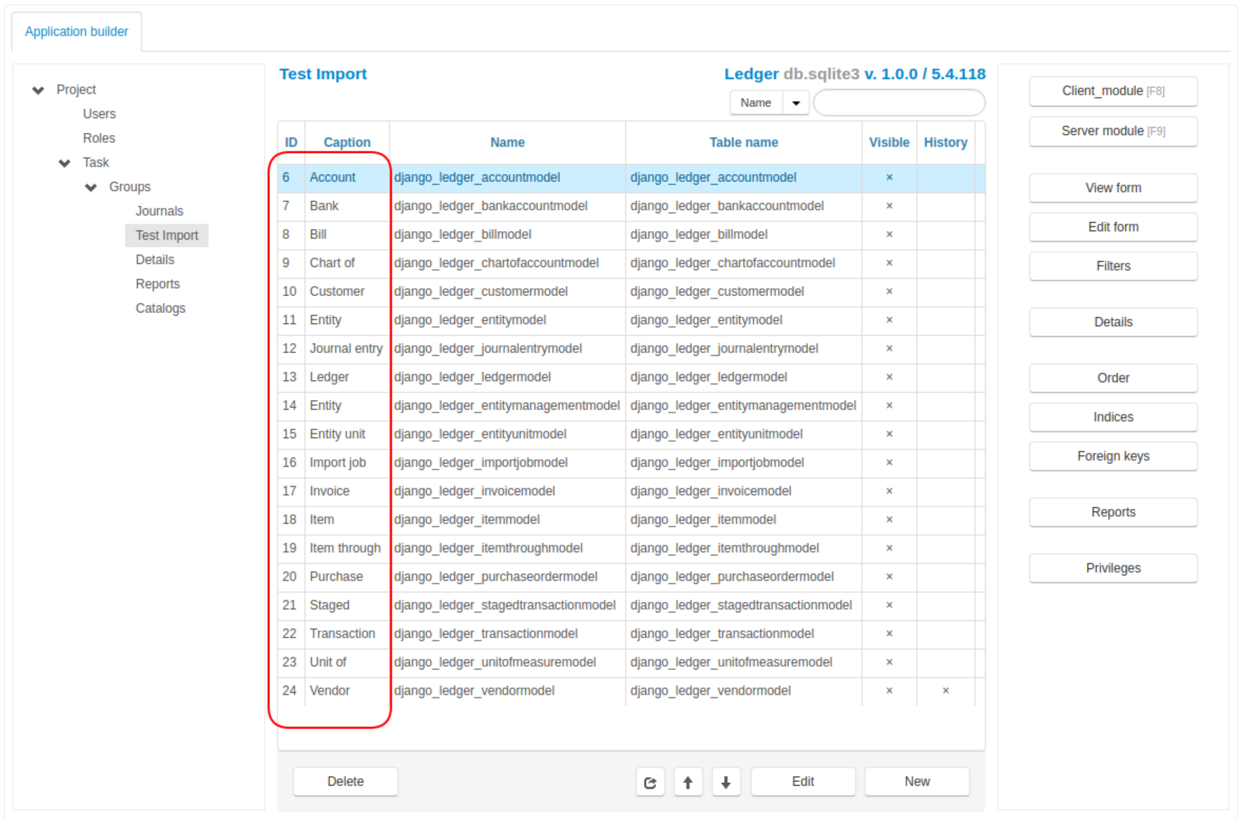
In order to use the automatic displaying of data (automatically with CRUD), the table name is used as table Caption! That is, if the Caption was not changed during the table Import.

Hence, if the database table name is `django_ledger_accountmodel`, then the JavaScript variable Name is also `django_ledger_accountmodel`! And more importantly, the visible table Caption on the Application is `django_ledger_accountmodel` as well.

So maybe better to set the table `Caption` to `Account`, `Bank`, etc. See how nicely everything looks like on the screenshot below?

This is also true for any table column during the table Import.

Finally, the application might look like this:



Test Import

Ledger db.sqlite3 v. 1.0.0 / 5.4.118

ID	Caption	Name	Table name	Visible	History
6	Account	django_ledger_accountmodel	django_ledger_accountmodel	×	
7	Bank	django_ledger_bankaccountmodel	django_ledger_bankaccountmodel	×	
8	Bill	django_ledger_billmodel	django_ledger_billmodel	×	
9	Chart of	django_ledger_chartofaccountmodel	django_ledger_chartofaccountmodel	×	
10	Customer	django_ledger_customermodel	django_ledger_customermodel	×	
11	Entity	django_ledger_entitymodel	django_ledger_entitymodel	×	
12	Journal entry	django_ledger_journalentrymodel	django_ledger_journalentrymodel	×	
13	Ledger	django_ledger_ledgermodel	django_ledger_ledgermodel	×	
14	Entity	django_ledger_entitymanagementmodel	django_ledger_entitymanagementmodel	×	
15	Entity unit	django_ledger_entityunitmodel	django_ledger_entityunitmodel	×	
16	Import job	django_ledger_importjobmodel	django_ledger_importjobmodel	×	
17	Invoice	django_ledger_invoicemodel	django_ledger_invoicemodel	×	
18	Item	django_ledger_itemmodel	django_ledger_itemmodel	×	
19	Item through	django_ledger_itemthroughmodel	django_ledger_itemthroughmodel	×	
20	Purchase	django_ledger_purchaseordermodel	django_ledger_purchaseordermodel	×	
21	Staged	django_ledger_stagedtransactionmodel	django_ledger_stagedtransactionmodel	×	
22	Transaction	django_ledger_transactionmodel	django_ledger_transactionmodel	×	
23	Unit of	django_ledger_unitofmeasuremodel	django_ledger_unitofmeasuremodel	×	
24	Vendor	django_ledger_vendormodel	django_ledger_vendormodel	×	×

The MS Access linked tables (by ODBC at.al), are now possible to import. There are some constraints though, not all datatypes can be recognised due to the ODBC connection. Of course, the MS Access linked tables are just the issue when using the conversion tool provided above. One can always connect to MSSQL server directly from Jam.py Application Builder and Import the tables.

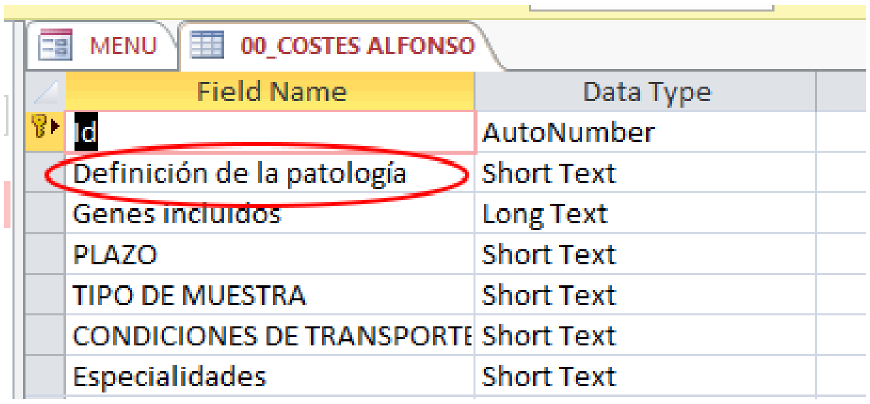
6.4.2 Using UTF8

The above is all well and true for the English language. Or the Latin alphabet. However, using UTF8 in MS Access is supported, but not supported in JavaScript naming standard. Unfortunately, this is directly impacting Jam.py tables Import usability.

Automatic translation to comply with JavaScript naming standard

Because of that reason, when using our site for the migration, all non-Latin alphabet is translated to JavaScript naming standard automatically. The below notes are valid for official Jam.py installation downloaded from the Internet, which does not support the automatic translation yet.

Consider this scenario:

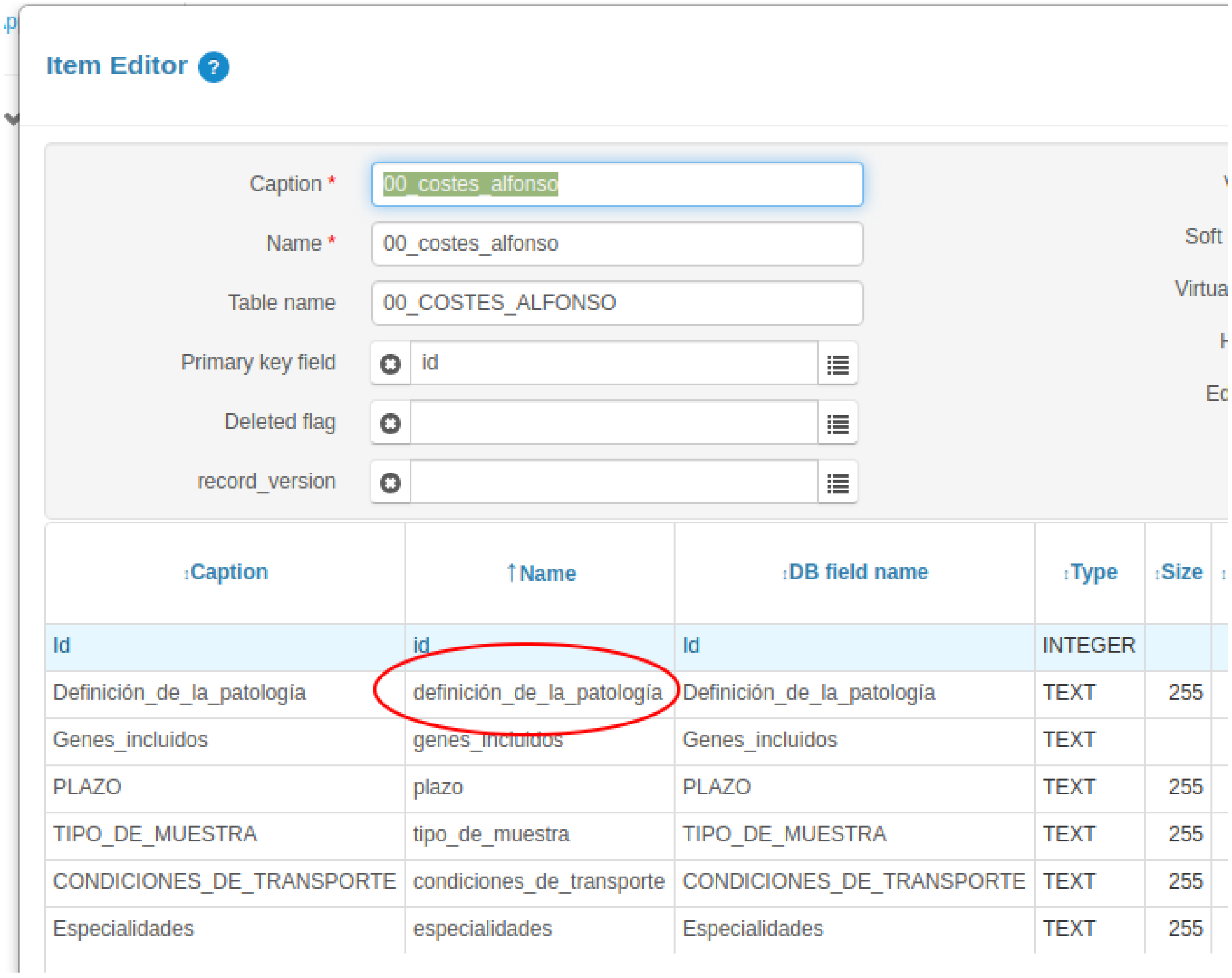


Field Name	Data Type
Id	AutoNumber
Definición de la patología	Short Text
Genes incluidos	Long Text
PLAZO	Short Text
TIPO DE MUESTRA	Short Text
CONDICIONES DE TRANSPORTE	Short Text
Especialidades	Short Text

The Table name is **00_COSTES ALFONSO**. And the list of the tables continues with **01_, 02_, 03_**, etc. Unfortunately, it is also using this as a table name: **00_PRECIOS ICM > GENOCOL_IMAT**.

Which just proves that MS Access will accept anything we throw on it. Is this a good practice? It is for the Access though. However, any modern framework will reject such a practise. Including Jam.py.

Importing the above table will produce the below:



Caption *	00_costes_alfonso
Name *	00_costes_alfonso
Table name	00_COSTES_ALFONSO
Primary key field	id
Deleted flag	
record_version	

Caption	Name	DB field name	Type	Size
Id	id	Id	INTEGER	
Definición_de_la_patología	definición_de_la_patología	Definición_de_la_patología	TEXT	255
Genes_incluidos	genes_incluidos	Genes_incluidos	TEXT	
PLAZO	plazo	PLAZO	TEXT	255
TIPO_DE_MUESTRA	tipo_de_muestra	TIPO_DE_MUESTRA	TEXT	255
CONDICIONES_DE_TRANSPORTE	condiciones_de_transporte	CONDICIONES_DE_TRANSPORTE	TEXT	255
Especialidades	especialidades	Especialidades	TEXT	255

Such a table Import might fail. It cannot be imported, and the problems need to be fixed first.

Note: The first issue is the table name starting with **00_, 01_, 02_…**, due to a JavaScript naming standard. The solution is to rename the tables to remove **00_, 01_, 02_…** in Access.

The migration utility will replace the starting numbers if they exist.

Or, we might use Python for this task:

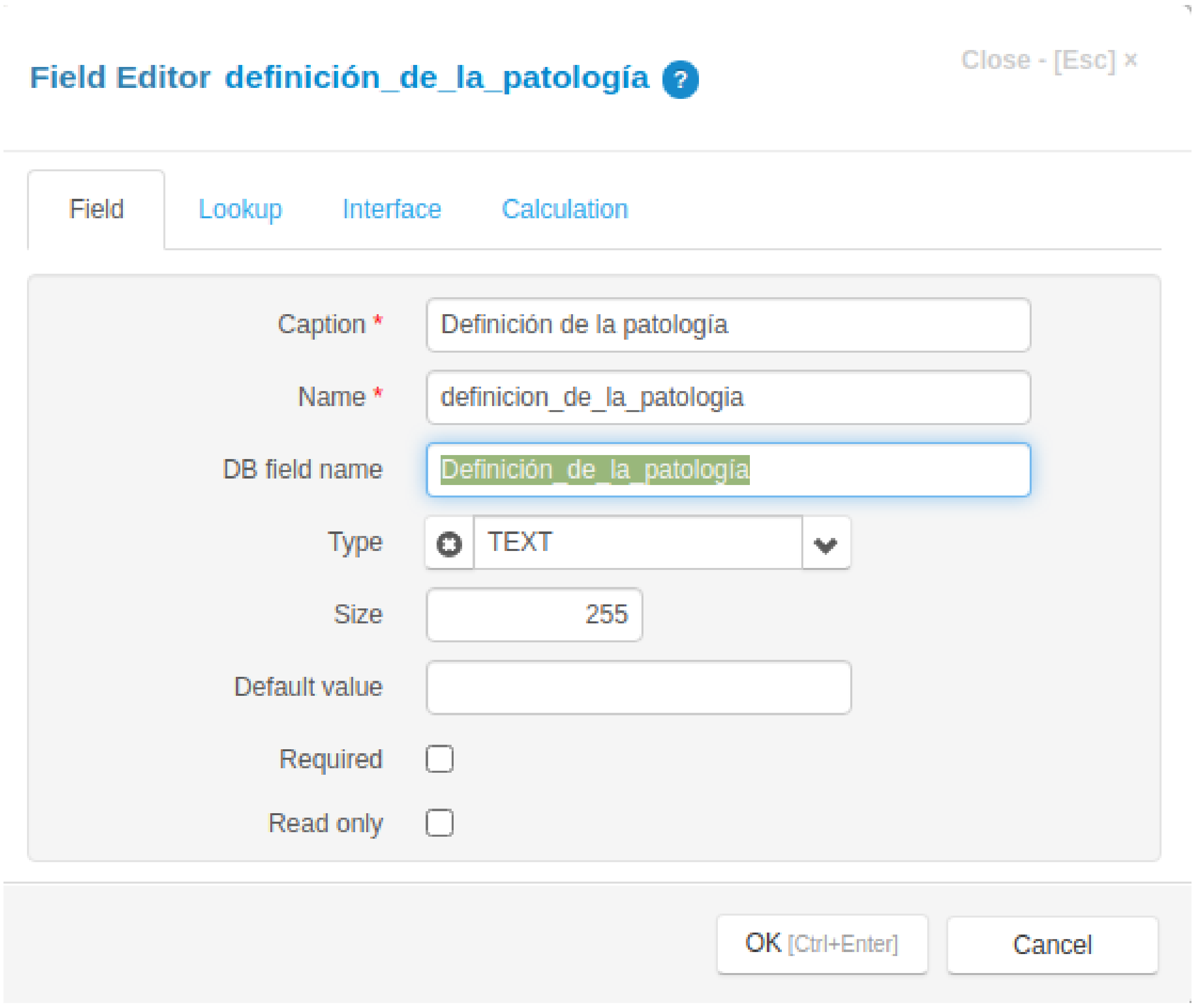
```python
import sqlite3

connection  = sqlite3.connect("database.sqlite3")
cursor      = connection.cursor()
tableQuery = "select * from sqlite_master where type = 'table' and name != 'Order'␣
↪and name != 'sqlite_sequence'"
cursor.execute(tableQuery)
tableList = cursor.fetchall()
for table in tableList:
    t_orig = table[1]
    t_fix = t_orig[3:]
    renameTable = 'ALTER TABLE "'+t_orig+"\" RENAME TO "+t_fix+""
    cursor.execute(renameTable)
connection.close()
```

Secondly, the table field used with UTF8: **definición_de_la_patología**. Note how the original MS Access field used is **Definición de la patología** with spaces? The conversion replaced spaces with the underscore automatically. It does that for the table name as well. However, read on.

The Import as such is going to fail, due to JavaScript naming standard for the column Name.

The solution is to rename **definición_de_la_patología** to **definicion_de_la_patologia**, or similar valid Name. The **DB Field name** can stay as is: Definición_de_la_patología.

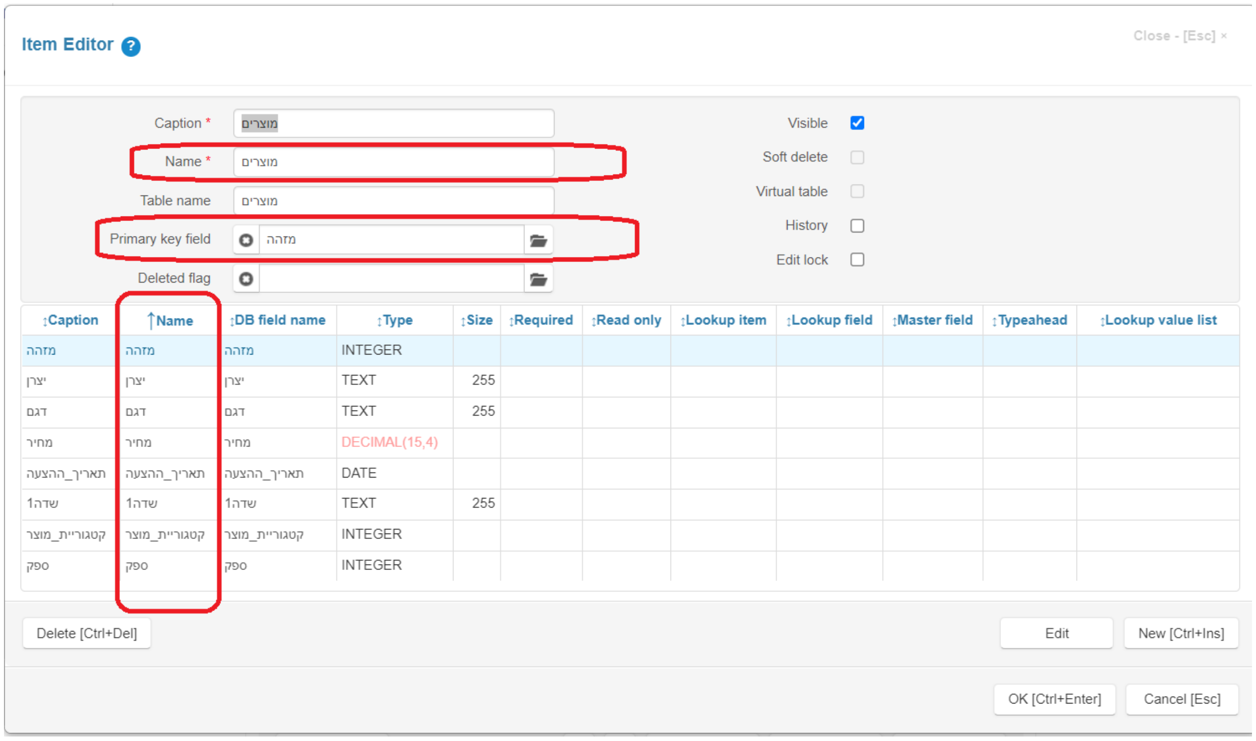
Finally, the Italian language field from MS Access might look like this:



Caption *	Definición de la patología
Name *	definicion_de_la_patologia
DB field name	Definición_de_la_patología
Type	TEXT
Size	255
Default value	
Required	☐
Read only	☐

As seen, Jam.py will happily use the UTF8 for the *Caption*. As well as the SQLite3 for database table Name and the DB field name. JavaScript naming standard is now fixed for the Name *.

In short, for any non-English language, or non-Latin alphabet, like the below Hebrew language, the Name * must be changed to Latin alphabet, and the rest can stay the same:

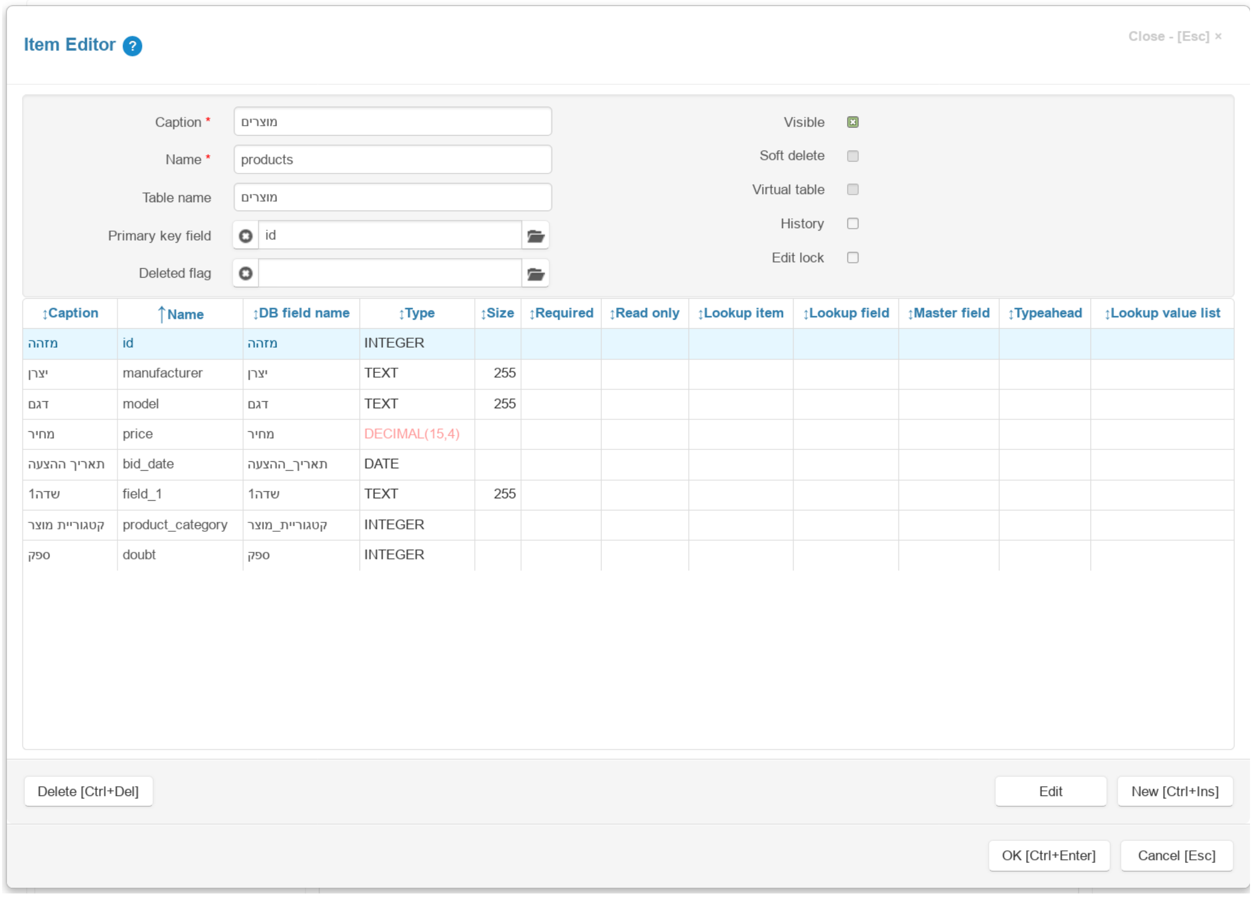


Caption *	מוצרים
Name *	מוצרים
Table name	מוצרים
Primary key field	מזהה
Deleted flag	
Visible	✓
Soft delete	
Virtual table	
History	
Edit lock	

Caption	Name	DB field name	Type	Size	Required	Read only	Lookup item	Lookup field	Master field	Typeahead	Lookup value list
מזהה	מזהה	מזהה	INTEGER								
יצרן	יצרן	יצרן	TEXT	255							
דגם	דגם	דגם	TEXT	255							
מחיר	מחיר	מחיר	DECIMAL(15,4)								
תאריך_ההצעה	תאריך_ההצעה	תאריך_ההצעה	DATE								
שדה1	שדה1	שדה1	TEXT	255							
קטגוריית_מוצר	קטגוריית_מוצר	קטגוריית_מוצר	INTEGER								
ספק	ספק	ספק	INTEGER								

Please note the Primary key field as well. Because the Name is in Hebrew language, the Primary Key field is also in Hebrew and Import will fail.

For that reason, we developed the way to translate all necessary items automatically, with no manual intervention needed. The above example with Hebrew language would automatically appear as per below. Due to translation inconsistencies, it is always advisable to double check the translated string. In this example, the "doubt" word probably is not correct, hence leaving to the readers to decide.



Caption *	מוצרים	Visible	☒
Name *	products	Soft delete	☐
Table name	מוצרים	Virtual table	☐
Primary key field	id	History	☐
Deleted flag		Edit lock	☐

Caption	Name	DB field name	Type	Size	Required	Read only	Lookup item	Lookup field	Master field	Typeahead	Lookup value list
מזהה	id	מזהה	INTEGER								
יצרן	manufacturer	יצרן	TEXT	255							
דגם	model	דגם	TEXT	255							
מחיר	price	מחיר	DECIMAL(15,4)								
תאריך ההצעה	bid_date	תאריך_ההצעה	DATE								
שדה1	field_1	שדה1	TEXT	255							
קטגוריית מוצר	product_category	קטגוריית_מוצר	INTEGER								
ספק	doubt	ספק	INTEGER								

If happy with the way the table looks like and DECIMAL(15,4) is changed to CURRENCY data type, we can proceed with the Import. It is possible to immediately check the result on the Internet, by removing builder.html from the browser link.

For this particular example with Hebrew language, the Application needs the support for right-to-left language.

This can be achieved by finding below code in index.html file (on Project/Task):

```
<html lang="en">
```

and replacing it with:

```
<html lang="he" dir="rtl">
```

6.4.3 Field Captions

To expand on UTF8 and field Captions, there is a utility for MS Access to dump all table fields Captions (MS Access optional Descriptions), to the CVS file. Why would we do that? Well, almost all third-party database conversion utilities do not write the Captions to the new database. Hence, the Access information might be lost. This is particularly true for a non English Access Front End Application, where the developers might utilise Captions from the Language table. Hence, the table name would always be the same, only the Captions would dynamically change.

In the above Italian language *scenario*, the Access Caption might be **Definición de la patología**. The MS Access Developer decided not to use Captions, as they are optional. Hence, the table Import will use the table name for the Caption, which is fine for this example as there are no many non-Latin names.

However, please consider the Access application **with** the field captions! This is what we are trying to resolve, so we reuse all the hard work already done in Access.

For example, if the MS Access table `DEMO_TRACKS` has a field `NAME1` and the Caption for the field `Track title 1`, the TEST.CSV file might look like below and the Python code to update it:

DEMO_TRACKS,	NAME1,	Track title 1
DEMO_TRACKS,	NAME2,	Track title 2
DEMO_TRACKS,	NAME3,	Track title 3

```python
import sqlite3
import numpy as np
captions = np.loadtxt('test.csv', dtype=str, delimiter=',')
print(captions)
# this will output:
#captions = [
#         ['DEMO_TRACKS', 'NAME1', 'Track title 1']
#         ['DEMO_TRACKS', 'NAME2', 'Track title 2']
# .
# .
#    ]
con = sqlite3.connect('admin.sqlite')
cursor = con.cursor()
for table_name, field_name, caption in captions:
    cursor.execute("SELECT ID FROM SYS_ITEMS WHERE DELETED=0 AND F_TABLE_NAME='%s'" %␣
↪table_name)
    res = cursor.fetchone()
    if res:
        item_id = res[0]
        cursor.execute("SELECT ID FROM SYS_FIELDS WHERE DELETED=0 AND OWNER_REC_ID=%s␣
↪AND F_DB_FIELD_NAME='%s'" % (item_id, field_name))
        res = cursor.fetchone()
```

(continues on next page)

(continued from previous page)

```
        if res:
            field_id = res[0]
            cursor.execute("UPDATE SYS_FIELDS SET F_NAME='%s' WHERE ID=%s" % (caption,
↪ field_id))
con.commit()
con.close()
```

Note: To prepare the CSV file, use Excel as "Save As" - "CSV UTF" - very important!

Also very important to run dos2unix command on the file in Linux, since it will otherwise do this at the output:

[[' ufeffDEMO_TRACKS, ..]

Note: The good news is, our conversion utility now supports Access optional Caption. They are written directly into the SQLite3 database as comments. All popular SQLite3 utilities, like "DB Browser for SQLite" or "SQLiteman", can read the database schema with the comments.

The below is an example for Arabic language where MS Access utilises the optional Description, which Jam.py might use for Captions. Each Caption starts with - - , which is a comment in SQLite3:

```
CREATE TABLE `Location` -- منطقه (`ID` INTEGER PRIMARY KEY AUTOINCREMENT -- ردیف
,
`location` VARCHAR(255) DEFAULT '' -- منطقه
,
`Edare` VARCHAR(255) DEFAULT '' -- اداره
,
`Address` VARCHAR(255) DEFAULT '' -- آدرس
)
```

We are working towards enabling Jam.py to read the table and field comments and interpret them as Captions while importing the table. As seen, the JavaScript naming standard does not apply for the above table, since the table is using Latin language for everything except Captions.

6.4.4 Reserved Words

In JavaScript and Python, there are a number of reserved words that can not be used. However, MS Access might happily use those. Simply add some character to the Name *, and leave the database Table Name as is. For example:

Caption *	Privileges	Active	Order
Name *	privileges_f	active_f	order_f
Table Name	privileges	active	order

6.4.5 Deleted Flag

In order to use a full Jam.py functionality with so called *Deleted Flag*, which basically prevents data to be permanently deleted, we need to add DELETED field to the tables. However, this is not required for Jam.py or the Application to function. It is an nice feature to have, specially if the *Foreign Key* enforcing is enabled. Meaning, some data cannot be deleted if some other data exist elsewhere as a reference. *Deleted Flag* totally eliminates the hustle of enforcing Foreign Keys deletion.

We could easily automate adding the `DELETED` field during the conversion, as it is not required for all tables probably. This is what we can do:

- we can add a field to any table with a Python code, as per below. The db.sqlite3 is the database converted from Access.

Note: "Order" word is a reserved word for Python, so we exclude it. There are other reserved words, please consult the Python or Jam.py documentation.

```python
import sqlite3
connection  = sqlite3.connect("db.sqlite3")
cursor      = connection.cursor()
tableQuery = "select * from sqlite_master where type = 'table' and name != 'Order' "
cursor.execute(tableQuery)
tableList = cursor.fetchall()
for table in tableList:
    renameTable = "ALTER TABLE %s"%(table[1]) + " ADD COLUMN DELETED integer"
    print(renameTable)
    cursor.execute(renameTable)
connection.close()
```

Now, we should be able to select the *Deleted Flag* as `deleted` field during the table Import.

6.4.6 Primary Keys

As noted in the *Top Migration Questions*, the Primary Key is a must. However, a lot of MS Access tables use the Indexed key only, and not an AutoNumber. If the key is an Integer, we might use it as an unique identifier in Jam.py.

Note: This only means that we need to manually update the identifier for a new record. More about that later.

For the tables with no indexed Integer, for example the plain indexed text values, the best is to add the Integer identifier.

Consider the scenario where table column `grupo` is referenced in table `material`, with MS Access functionality:

```
CREATE TABLE `agru`
(`grupo` VARCHAR(255) DEFAULT '')
```

In MS Access, the field `grupo` is indexed (no duplicates), however SQLite3 interprets this as above after the migration.

The solution is to add `ID` as identifier and populate the record with `rowid`, which is built-in SQLite3 function:

```
alter table agru ADD COLUMN ID integer;
update agru set id = rowid+1;
```

Now we can create `agru_id` in the table `material` and populate records with the `ID` value from table `agru`:

```
update material set agru_id = ( select id from agru where material.agru = agru.grupo)
```

As mentioned, because it is impossible to add a Primary Key with *AutoIncrement* to an existing table, the database has no functionality to increase the `ID` on-the-fly. Hence, we do that manually.

To do that, we just add below code to table `grupo` on *Client Module*:

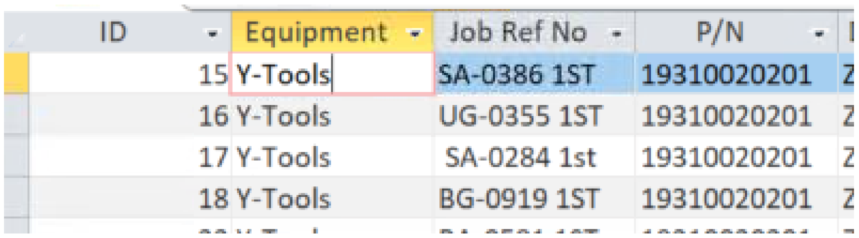
```
function on_before_post(item) {
    if (item.is_new()) {
        let copy = item.copy();
            copy.open();
            if (copy.rec_count ===0) {
                item.id.value = 1;
            }   else {
                item.id.value = copy.rec_count + 1;
            }
    }
}
```

The preferred method is to always use a *Primary Key* with AutoIncrement within the database. However, this would mean redesigning the application or starting from scratch.

6.4.7 Foreign Keys

Similar scenario, after the data is migrated, we can create a Foreign Keys within the Builder. Again, the Jam.py utilises *lookups* for looking up for data.

However, if the Access tables did not use a Foreign Keys, this will be an issue. Consider this scenario:



ID	Equipment	Job Ref No	P/N	I
15	Y-Tools	SA-0386 1ST	19310020201	Z
16	Y-Tools	UG-0355 1ST	19310020201	Z
17	Y-Tools	SA-0284 1st	19310020201	Z
18	Y-Tools	BG-0919 1ST	19310020201	Z

All columns in this Access table do not utilise a Foreign Key. This is clearly visible when we click on the field. If the Foreign Key is used, the Access Lookup would open. In addition, if we utilise MS Access *Database Tools/Analyse Table* option, the Wizard would suggest to split the table in a number of new ones. Hence, the migration to Web of such table will not produce the expected result, since the values are *hard coded*. The solution is to split the table with MS Access Wizard first, then use the migration tool.

6.4.8 Indexes

As stated, the migration does not import indexes. This is due to a number of constraints. The good news is once the data is converted, the indexes can be created within the Builder. The good practise is to index all *lookups* on some table.

Why is this important? Because doing it within the Builder, all indexes are kept and can be recreated on any supported database engine. If the indexes exist within the database only, the Builder would not know about it and they would be lost when moving Application to the preferred database.

6.4.9 Users table

If already using some MS Access database table for users authentication, the only thing missing is the Role *lookup*. It is quite possible that the role is hard-coded in Access, and this is what we are trying to avoid. Use *lookups* as much as possible.

The process is described in here: How to authenticate from custom users table.

Obviously, it is the best practise to use password hash for storing the passwords. This simply means adding a column with a **Password hash** text to the *Users* or similar table.

Note: To Be Continued···

6.5 Some migrated examples

To make it easier for a would be Jam.py Users/Developers, we started to build the showcase repository with the Applications directly migrated from MS Access. It is important to understand that one can start completely from scratch and build the application without any conversion or migration tools. We just think it' s easier to use the tools on our disposal.

Please find the list of some applications available as Templates in Access and migrated to Jam.py. All source code is available for download on Export tab.

For some mentioned applications, the Export file is available. Download the Export, *install* the `Jam.py` framework locally or on the server, create the *New Project*, *Import* the same downloaded file, and point the Application to sqlite database as per creating the New Project.

6.5.1 Personal Account Ledger

The application lives here:

https://msaccess.pythonanywhere.com

Note: If the Application Export is downloaded from the above link, ie. to try the application by yourself, the first thing needed is to install Python libraries.

Providing the Jam.py is already installed and a new application created, please execute below for Windows before Importing the downloaded file:

```
py -m pip install password_strength
```

Or Linux:

```
pip install password_strength
```

Overview

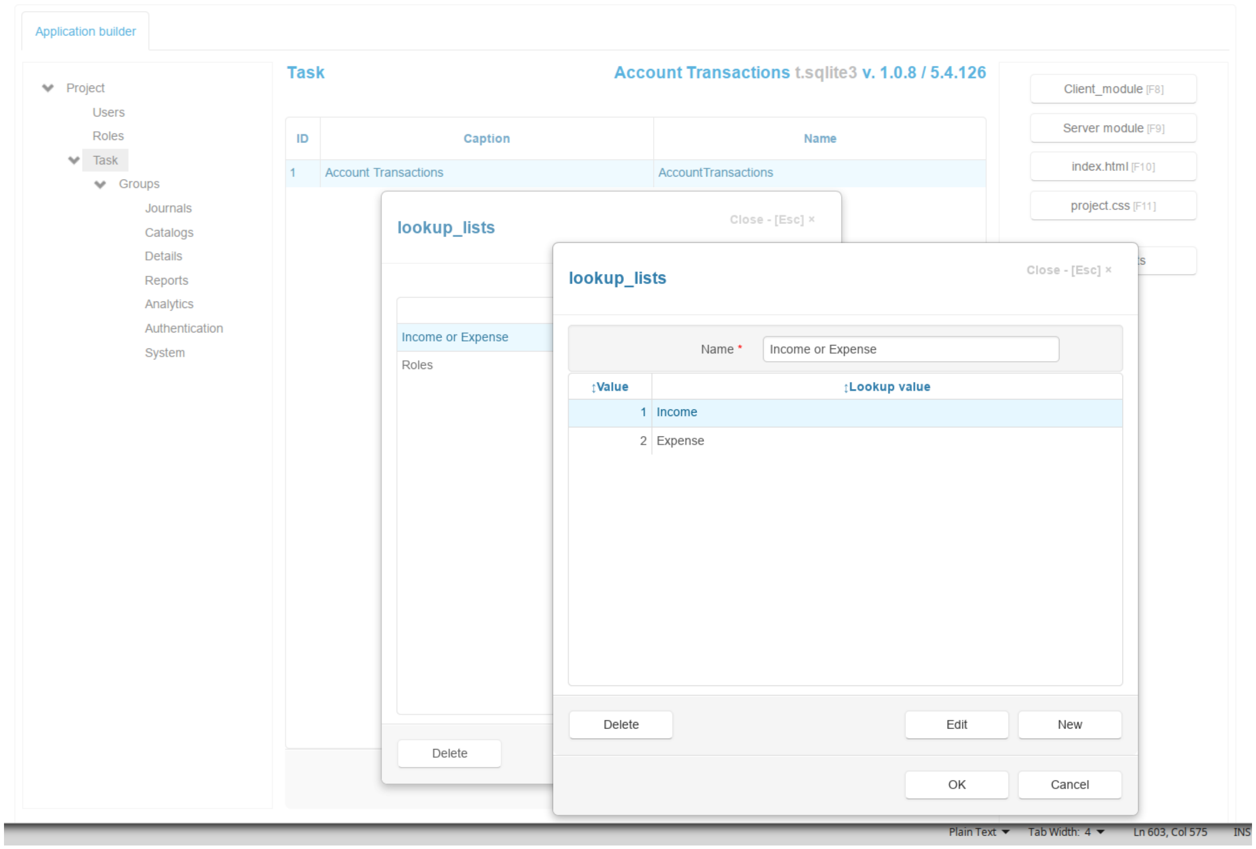
The Access Template Personal Account Ledger has a "feature" which enables typing the negative value for currency. Then the actual Expense is automatically converted to Income, since the negative currency becomes a positive value! Our migrated project does not allow for this due to *validation*.

The complete application was developed in a few hours` time. Below is the approximate process:

- The template was uploaded to https://jampyapplicationbuilder.com/conv/ to give us all tables and data needed for the project.

- All tables were imported into the Jam.py Application Builder from the provided link after the upload. Since the Access Categories Form has a Drop-down list with the Income and Expense, the Lookup List was created with the same.

Lookup List

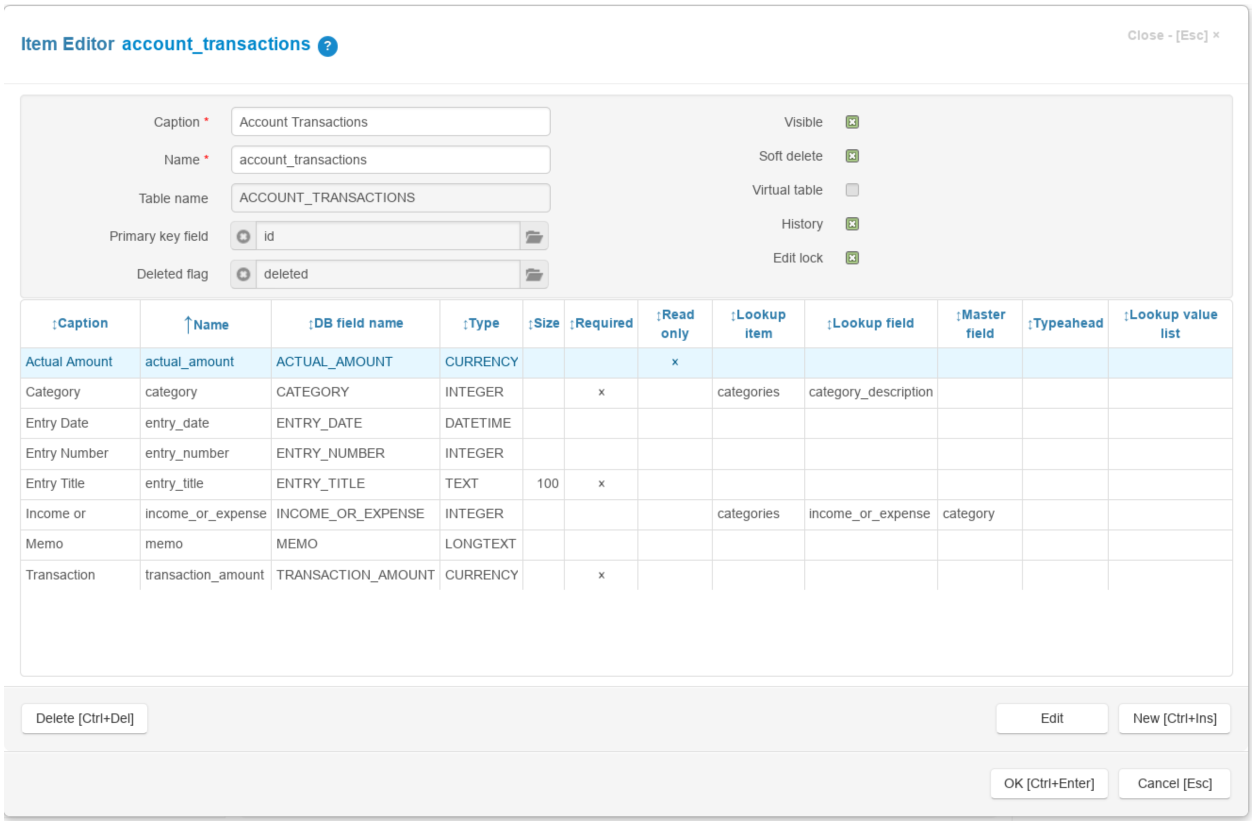


ID	Caption	Name
1	Account Transactions	AccountTransactions

Name * Income or Expense

Value	Lookup value
1	Income
2	Expense

- By looking at the Access Database Tools Relationships, the Account Transactions table *Category* field was linked by the Lookup Item to Categories Description field. The *Income or Expense* field was created in the *Transaction table* and linked to the Categories same field. Also, the *actual_amount* field was added, since in Access this is a query field.

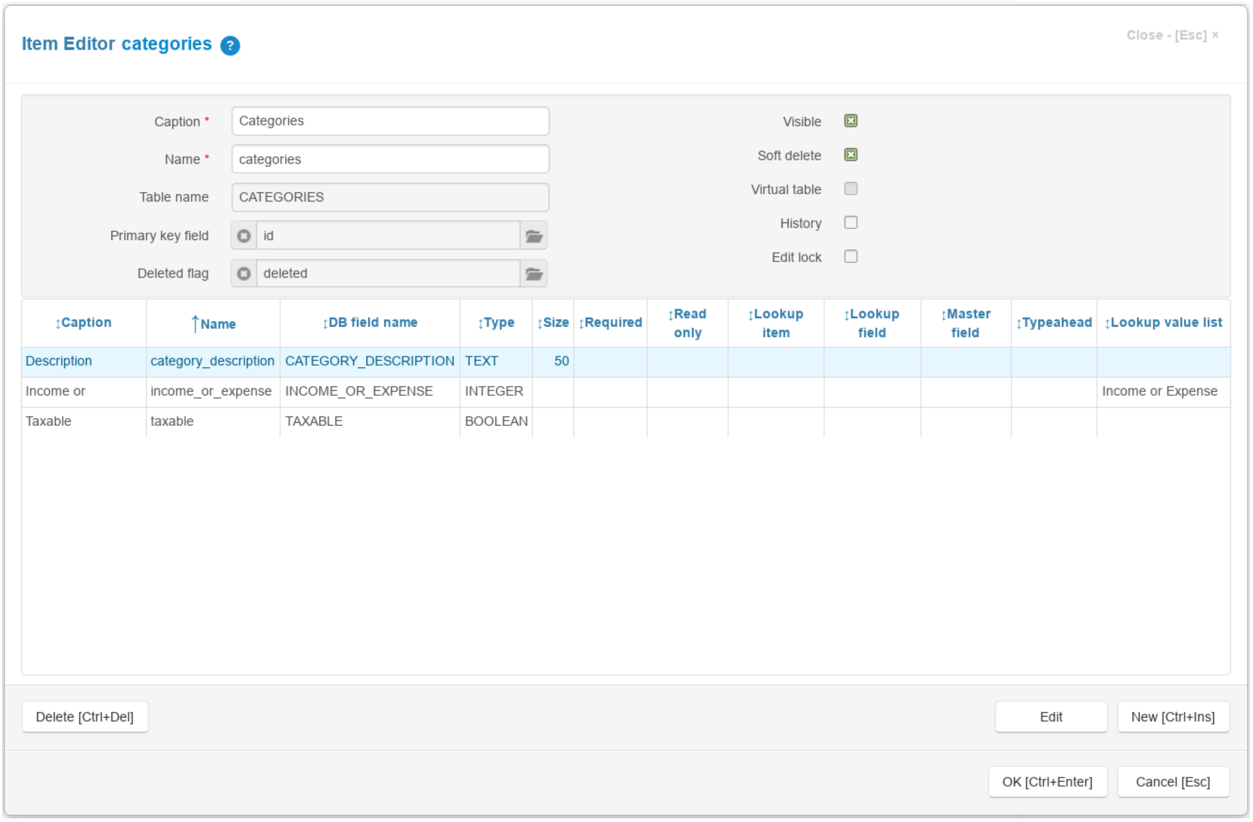


Item Editor account_transactions

Caption *	Account Transactions	Visible	☒
Name *	account_transactions	Soft delete	☒
Table name	ACCOUNT_TRANSACTIONS	Virtual table	☐
Primary key field	id	History	☒
Deleted flag	deleted	Edit lock	☒

Caption	Name	DB field name	Type	Size	Required	Read only	Lookup item	Lookup field	Master field	Typeahead	Lookup value list
Actual Amount	actual_amount	ACTUAL_AMOUNT	CURRENCY			×					
Category	category	CATEGORY	INTEGER		×		categories	category_description			
Entry Date	entry_date	ENTRY_DATE	DATETIME								
Entry Number	entry_number	ENTRY_NUMBER	INTEGER								
Entry Title	entry_title	ENTRY_TITLE	TEXT	100	×						
Income or	income_or_expense	INCOME_OR_EXPENSE	INTEGER				categories	income_or_expense	category		
Memo	memo	MEMO	LONGTEXT								
Transaction	transaction_amount	TRANSACTION_AMOUNT	CURRENCY		×						

- The Categories *Income or Expense* was linked to Lookup Value List.



Item Editor categories

Caption *	Categories	Visible	☒
Name *	categories	Soft delete	☒
Table name	CATEGORIES	Virtual table	☐
Primary key field	id	History	☐
Deleted flag	deleted	Edit lock	☐

Caption	Name	DB field name	Type	Size	Required	Read only	Lookup item	Lookup field	Master field	Typeahead	Lookup value list
Description	category_description	CATEGORY_DESCRIPTION	TEXT	50							
Income or	income_or_expense	INCOME_OR_EXPENSE	INTEGER								Income or Expense
Taxable	taxable	TAXABLE	BOOLEAN								

A little code

- The first code was added to restrict adding a negative currency with the MessageBox displaying the Alert.

```
function on_field_validate(field) {
    if (field.field_name === 'transaction_amount' && field.value < 1) {
            return 'The amount cant be negative!';
    }
}
```

- Then, we observed how is Access displaying the Expense/Income formatting in red and green color on the Data Grid. The decision has been made to display the actual Expense/Income text in a coloured way as well as the value.
- The code was added to accommodate for the above:

```
function on_field_get_html(field) {
    let item = field.owner;
    if (field.field_name === 'category') {
        let color = 'green';
        if (item.income_or_expense.display_text === 'Expense') {
            color = 'red';
        }
        return '<span style="color: ' + color + ';">' + field.display_text + '/' +␣
↪field.owner.income_or_expense.display_text + '</span>';
    }
    if (field.field_name === 'actual_amount' || field.field_name === 'category') {
        let color = 'green';
        if (item.income_or_expense.display_text === 'Expense') {
            color = 'red';
        }
        return '<span style="color: ' + color + ';">' + field.display_text + '</span>
↪';
    }

}
```

- Because Access is triggering the displaying of Actual Amount relative to Expense option in red color, or Income as green colour, the code was added to trigger the similar behaviour:

```
function on_field_changed(field, lookup_item) {
    var item = field.owner;
    if (field.field_name === 'transaction_amount' || field.field_name === 'category')
↪{
        calculate(item);
    }
}
```

- The JS function is needed to actually do the displaying of positive to negative values and vice versa:

```
function calculate(item) {
    if (item.income_or_expense.value) {
        item.actual_amount.value = item.transaction_amount.value;
        if (item.income_or_expense.display_text === 'Expense') {
            item.actual_amount.value = -item.actual_amount.value;
        }
    }
    else {
        item.actual_amount.value = 0;
    }
}
```

- Some formatting was added to make the Entry Title in bold. The below function was added at the end of *on_field_get_html* function from above:

```
if (field.field_name==='entry_title') {
    return '<strong>' + field.display_text + '</strong>';
}
```

- Then, the Dashboard/Reports was added. Some tlc is still needed for those. The *Dashboard* was covered in "How was Demo built?" topic

---

Note:

- The above is all from the actual application Front End point of view. It looks pretty similar to Access, correct? At this moment it is absolutely possible to turn on the "Safe Mode", which is the Authentication.

- All of the above was added to Account Transactions *Client module*. No more code is needed for Access basic Front End look and feel. The additional code for Authentication was copy/paste from the https://jampyapp.pythonanywhere.com/ project. Please observe the code and needed tables within the Builder/Authentication tab. Also, the index.html file was modified and added simple-line-icons support with the Task/project.css. Same copy/paste from the mentioned project.
- The *Clone* button was developed latter to emulate the support for highlighted row copy/paste.
- The *Delete* button was disabled with the *MessageBox* displaying the Alert, since the deletion is disabled for Publicly accessible application. This is controlled in Task/Account Transaction/Client module, please search for *deleted*, since deletion is actioned on CTRL+DEL as well.
- After two weeks running and some feedback from the Users, the Account *Transaction/Filter* was added. Also, some formatting was developed, particularly for displaying red and green *Actual Amount* on the Form. This is not the *Data Grid* formatting, which was added from the beginning. Hence, consider the above *positive to negative* function. The function was missing the CSS needed for the Form. The new function is below:

```
function calculate(item) {
    if (item.income_or_expense.value) {
        item.actual_amount.value = item.transaction_amount.value;
        if (item.income_or_expense.display_text === 'Expense') {
            item.actual_amount.value = -item.actual_amount.value;
            item.edit_form.find('input.actual_amount').css('color', 'red');
        }
        else {
            item.edit_form.find('input.actual_amount').css('color', 'green');
        }
    }
    else {
        item.actual_amount.value = 0;
    }
}
```

- More formatting was added to resize the fields. Here, we introduced a new Jam.py function for changing the CSS when the Edit/New Form is active. Which is as well changing the CSS for the field, but this time on the Edit/New Form:

on_edit_form_shown()

```
function on_edit_form_shown(item) {
    if (item.income_or_expense.value) {
        //item.actual_amount.value = item.transaction_amount.value;
        if (item.actual_amount.value < 1) {
            item.edit_form.find('input.actual_amount').css('color', 'red');
            //item.actual_amount.value = -item.actual_amount.value;
        }
        else {
            item.edit_form.find('input.actual_amount').css('color', 'green');
        }


    }

  item.edit_form.find('input.entry_number').parent().width('40%');
  item.edit_form.find('input.transaction_amount').parent().width('40%');
  item.edit_form.find('input.category').parent().width('60%');
  item.edit_form.find('input.categories').parent().width('60%');
  item.edit_form.find('input.income_or_expense').width('60%');
}
```

- History/Edit Lock was added. This is a no code operation done on Project/Parameters.
- The Report was missing the *Income/Expense* Parameter. Added. The Reports design and code might be covered

latter on.

- The Filters was added for *Account Transaction* table. Because of the "combined" Category with the "/" sign, some tweaking was needed. Here is the code added to the *Account Transactions* table, and the code to add three buttons. The buttons "Clone", "Filter by type" and "Clear filters", respectively:

```
function on_view_form_created(item) {
    if (!item.lookup_field) {
        var clone_btn = item.add_view_button('Clone', {image: 'icon-magic-wand'});
        clone_btn.click(function() { clone_record(item) });
    }
    //here is setting for filter by type of transaction (income or expense)
    let filter_type_btn = item.add_view_button('Filter by type', {type: 'primary',␣
↪image: 'icon-filter'});
        filter_type_btn.click(function() {
            filter_by_type(item);
        });

    let clear_filters_btn = item.add_view_button('Clear filters', {type: 'danger',␣
↪image: 'icon-remove'});
        clear_filters_btn.click(function() {
            item.clear_filters();
            item.refresh_page();
        });

}

function filter_by_type(item) {
    let copy = task.categories.copy({handlers: false});
    copy.open({fields: ['income_or_expense'], open_empty: true});

    copy.edit_options.title = 'Filter by type of transaction';
    copy.edit_options.history_button = false;

    copy.on_edit_form_created = function(c) {
        c.edit_form.find('#ok-btn')
            .text('Select type')
            .off('click.task')
            .on('click', function() {
                try {
                    c.post();
                    let categories_by_type = task.categories.copy({handlers: false}),
                        categories_list = [];
                        categories_by_type.set_where({income_or_expense: c.income_or_
↪expense.value});
```

(continues on next page)

(continued from previous page)

```
                        categories_by_type.open({fields: ['id']});

                        categories_by_type.each(function(s){
                            categories_list.push(s.id.value);
                        });

                        item.filters.category.value = categories_list;
                        item.refresh_page();
                    c.cancel_edit();
                }
                finally {
                    c.edit();
                }
            });
    };
    copy.on_edit_form_keyup= function(c, e) {
        if (e.keyCode === 13 && e.ctrlKey) {
            e.preventDefault();
            return true;
        }
    };
    copy.append_record();
}
```

- More summary was added for *Account Transaction* table. This is a no code operation done on View Form.

Validation

This is the JS code for the transaction amount validation. We also added on the table `account_transactions`, field *transaction_amount*, on the the Interface Help: *Value should be less then 10000*.

```
function on_field_validate(field) {
    if (field.field_name === 'transaction_amount' && field.value < 1) {
            return 'The amount cant be negative!';
    }
    if (field.field_name === 'transaction_amount' && field.value > 10000) {
            return 'The value is be less then 10000!';
    }
}
```

In June 2023, added date validation to above code, and the Order by *Entry Date* as DESC:

```
if (field.field_name === 'entry_date' && field.value > Date.now()) {
```

(continues on next page)

(continued from previous page)

```
        return 'The date can not be in the future!';
}
```

In Jan 2024, added Master/Detail for Categories:

Master/Detail

- For Master/Detail View only, there is no need to do anything special other than to add a simple code to display the Detail for a Master table. Please see the *Northwind Traders Master/Detail View* example.

In February 2024, added `Read Only` on Editing for *Categories* table, *Income or Expense* field. Also added formatting.

```
function on_edit_form_created(item) {
    if (item.is_new()) {
        item.income_or_expense.read_only = false;
    }   else {
        item.income_or_expense.read_only = true;
    }


}
```

API

In November 2023, added API endpoint on Task/Server:

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

The table `account_transactions` has the Server Module, which will return Total for the `Actual Amount`:

```
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)
```

The API can be accessed with:

```
curl -k https://msaccess.pythonanywhere.com/ext/expenses -d "[]" -H  "Content-Type:␣
↪application/json"
```

Now that we have API in place, it is trivial to use it in any other `Jam.py` application within the Server Module:

```
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

url = 'https://msaccess.pythonanywhere.com/ext'

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']

result = send(url, 'expenses', [])
print(result)
```

For example, on *NorthwindTraders*, we created a table `api_call`, with the field `Value`. The value of this field is populated by the *Server Code* within the same table. The code is executed by the button *Fetch from API*.

index.html

- Added the Export tab to download the latest source code. The Export feature is available by default on the Application Builder. The Builder is not accessible publicly due to the security reasons. Here is the code added to index.html file below “about” and *Account Transaction* table “function on_page_loaded(task)” :

```
<li id="about"><a href="#">About</a></li>
<li id="export"><a href="#">Export</a></li>
```

```
$("#menu-right #export a").click(function(e) {
            var url = [location.protocol, '//', location.host, location.pathname].
↪join('');
            url += 'static/internal/AccountTransactions.zip';
            window.open(encodeURI(url));
});
```

- Maybe the last feature needed is the CVS (spreadsheet) export/import. This step is fully documented in the official documentation.

Note: To Be Continued when/if more features were added …

6.5.2 Northwind Traders

The application lives here:

https://northwind.pythonanywhere.com

Note: If the Application Export is downloaded from the above link, ie. to try the application by yourself, the first thing needed is to install Python libraries.

Providing the Jam.py is already installed and a new application created, please execute below for Windows before Importing the downloaded file:

```
py -m pip install rfm pandas matplotlib numpy password_strength faker faker_food
```

Or Linux:

```
pip install rfm pandas matplotlib numpy password_strength faker faker_food
```

Overview

The Northwind template has some VBA attached to it. It is a fairly simple code from the migration point of view. Below is the approximate process:

- The template was uploaded to https://jampyapplicationbuilder.com/conv/ to give us all tables and data needed for the project.
- All tables were imported into the Jam.py Application Builder from the provided link after the upload. Since there are a few tables with no *Primary Key*, this are the candidates for the Lookup List or tables:

```
Employee_Privileges
Inventory_Transaction_Types
Order_Details_Status
Orders_Status
Orders_Tax_Status
Purchase_Order_Status
```

- For this exercise, we decided to add the *Primary Key* for the above tables, instead of making the *Lookup Lists*. The only *Lookup List* created in the beginning of migration was for *Purchase orders* and *Orders* tables, *Payment Method* field. Which in Access has hard coded values as *Cash*, *Check* and *Credit Card* information (in the sense of exporting the data into a csv):

```
Payment Method
```

The *Payment Method* is the VARCHAR Type in MS Access. Hence, to be able to use a *Lookup List*, it was changed to INTEGER after the Import, and pointed to a created *Lookup List*.

Note: Which opens a question: how to migrate the hard coded text to a *Lookup List* or even a table? A bit latter about that.

Virtual Table

- The biggest challenge was implementing the *Inventory List* MS Access SQL query as a Virtual Table, since the query is quite large:

```
SELECT Products.ID AS [Product ID], Products.[Product Name], Products.[Product Code],␣
↪Nz([Quantity Purchased],0) AS [Qty Purchased], Nz([Quantity Sold],0) AS [Qty Sold],␣
↪Nz([Quantity On Hold],0) AS [Qty On Hold], [Qty Purchased]-[Qty Sold] AS [Qty On␣
↪Hand], [Qty Purchased]-[Qty Sold]-[Qty On Hold] AS [Qty Available], Nz([Quantity On␣
↪Order],0) AS [Qty On Order], Nz([Quantity On Back Order],0) AS [Qty On Back Order],␣
↪Products.[Reorder Level], Products.[Target Level], [Target Level]-[Current Level]␣
↪AS [Qty Below Target Level], [Qty Available]+[Qty On Order]-[Qty On Back Order] AS␣
↪[Current Level], IIf([Qty Below Target Level]>0,IIf([Qty Below Target Level]
↪<[Minimum ReOrder Quantity],[Minimum Reorder Quantity],[Qty Below Target Level]),0)␣
↪AS [Qty To Reorder]
FROM ((((Products LEFT JOIN [Inventory Sold] ON Products.ID = [Inventory Sold].
↪[Product ID]) LEFT JOIN [Inventory Purchased] ON Products.ID = [Inventory␣
↪Purchased].[Product ID]) LEFT JOIN [Inventory On Hold] ON Products.ID = [Inventory␣
↪On Hold].[Product ID]) LEFT JOIN [Inventory On Order] ON Products.ID = [Inventory␣
↪On Order].[Product ID]) LEFT JOIN [Products On Back Order] ON Products.ID =␣
↪[Products On Back Order].[Product ID];


SELECT [Inventory Transactions].[Product ID], Sum([Inventory Transactions].Quantity)␣
↪AS [Quantity Sold]
FROM [Inventory Transactions]
WHERE ((([Inventory Transactions].[Transaction Type])=2))
GROUP BY [Inventory Transactions].[Product ID];
```

(continues on next page)

(continued from previous page)

```
SELECT [Inventory Transactions].[Product ID], Sum([Inventory Transactions].Quantity)␣
↪AS [Quantity Purchased]
FROM [Inventory Transactions]
WHERE ((([Inventory Transactions].[Transaction Type])=1))
GROUP BY [Inventory Transactions].[Product ID];

SELECT [Purchase Order Details].[Product ID] AS [Product ID], Sum([Purchase Order␣
↪Details].Quantity) AS [Quantity On Order]
FROM [Purchase Order Details]
WHERE ((([Purchase Order Details].[Posted To Inventory])=False))
GROUP BY [Purchase Order Details].[Product ID];

SELECT [Inventory Transactions].[Product ID], Sum([Inventory Transactions].Quantity)␣
↪AS [Quantity On Hold]
FROM [Inventory Transactions]
WHERE ((([Inventory Transactions].[Transaction Type])=3))
GROUP BY [Inventory Transactions].[Product ID];

SELECT [Order Details].[Product ID] AS [Product ID], Sum([Order Details].Quantity) AS␣
↪[Quantity On Back Order]
FROM [Order Details]
WHERE ((([Order Details].[Status ID])=4))
GROUP BY [Order Details].[Product ID];
```

Note: The above SQL will work only in MS Access. Of course, developing a completely new SQL is possible for any database provider. Jam can use any SQL within the Server Module, however we will demonstrate a pure Jam way of doing it.

Here is the JS and Python code, respectively, replacing the need for SQL. The button *Purchase* was added just like it exists on MS Access.

The important JS function is:

on_after_open()

As this JS function is taking the result of *get_rows* Server Module function, it is placed in `on_after_open`. This is important to understand, because in formal Jam.py Documentation it is sparsely mentioned. Most of the time, the `on_after_open` is used with Master/Detail views. We will touch base more about this function letter on, since it is extremely important to understand how to use it properly.

There is a topic How to link two tables in the Docs, which I would also encourage to visit.

```
function on_view_form_created(item) {
```

(continues on next page)

(continued from previous page)

```
    item.paginate = false;
    item.table_options.new = false;
    if (!item.lookup_field) {
        var email_btn = item.add_view_button('Purchase', {image: 'icon-pencil'});
        email_btn.click(function() { purchase() });
    }
            item.view_form.find("#edit-btn").hide();
            item.view_form.find("#delete-btn").hide();
            item.view_form.find("#new-btn").hide();
}

function on_after_open(item) {
    item.server('get_records', function(records) {
        records.forEach(function(rec) {
            item.append();
            item.product_name.value = rec.product_name;
            item.target_level.value = rec.target_level;
            item.quantity_on_hold.value = rec.quantity_on_hold;
            item.quantity_on_order.value = rec.quantity_on_order;
            item.quantity_on_hand.value = rec.quantity_on_hand;
            item.quantity_purchased.value = rec.quantity_purchased;
            item.quantity_sold.value = rec.quantity_sold;
            item.post();
        });
    });
}
function on_edit_form_created(item) {
    var title = 'Purchase ';
    item.edit_options.title = title;
    item.edit_form.find('#ok-btn')
        .text('Purchase')
        .off('click.task')
        .on('click', function() {
            purchase(item);
        });
}
function on_field_get_html(field) {
    let item = field.owner;
    if (field.field_name === 'quantity_on_hand') {
        let color = 'green';
        if (item.quantity_on_hand.display_text < 30) {
            color = 'red';
        }
        return '<span style="color: ' + color + ';">' + field.display_text + '</span>
```

(continues on next page)

(continued from previous page)

```
↪';
    }
}
```

Next is the Server Module code:

The Python code needs a lot of explanation for a *would be* Python developer. In short, we are looping through three SQL statements, and not six as in MS Access. All of this is to populate the list `res` and to return the information back to JavaScript function. Each SQL statement can be recognised by `set_where` function:

set_where()

```
def get_records(item):
    res, inventory = [], product = [], ''

    product = item.task.products.copy()
    #product.set_where(id=41)                              # here we can specify␣
↪just one item for testing
    product.open(fields=['id', 'product_name', 'target_level'],
        group_by=['id'], order_by=['id'])


    for p in product:
        allocated_inventory = 0
        target_level = 0
        product_id = product.id.value
        target_level += product.target_level.value
        print(product_id)

        order_details = item.task.order_details.copy()
        order_details.set_fields('product_id', 'status_id', 'quantity')
        #order_details.set_where(product_id=product_id, status_id=4); # here we can␣
↪specify just one item for testing
        order_details.set_where(product_id=product_id);

        order_details.open(fields=['product_id', 'quantity', 'status_id'],
            funcs={'quantity': 'sum'},
            group_by=['product_id'], order_by=['product_id'])
        for i in order_details:
            quantity = 0;
            transaction_type = 0
            quantity_sold = 0
            quantity_purchased = 0
            quantity_on_hold = 0
```

(continues on next page)

(continued from previous page)

```
            quantity_on_hand = 0
            quantity_on_order = 0
            quantity_on_back_order = 0

            #print()

            inv_transactions = item.task.inventory_transactions.copy()
            inv_transactions.set_fields('product_id', 'transaction_type', 'quantity')
            inv_transactions.set_where(product_id=product_id, transaction_type__in=[1,
↪2,3])
            inv_transactions.open(fields=['product_id', 'transaction_type', 'quantity
↪'],
                funcs={'quantity': 'sum'},
                group_by=['transaction_type'], order_by=['product_id'])

            for i in inv_transactions:
                if inv_transactions.transaction_type.value == 1:
                    quantity_purchased += inv_transactions.quantity.value
                if inv_transactions.transaction_type.value == 2:
                    quantity_sold += inv_transactions.quantity.value
                if inv_transactions.transaction_type.value == 3:
                    quantity_on_hold += inv_transactions.quantity.value
                quantity_on_hand = quantity_purchased - quantity_sold

                purchase_order_details = item.task.purchase_order_details.copy()
                purchase_order_details.set_fields('product_id', 'posted_to_inventory',
↪ 'quantity')
                purchase_order_details.set_where(product_id=product_id, posted_to_
↪inventory=0)
                purchase_order_details.open(fields=['product_id', 'quantity'],
                    funcs={'id': 'count'},
                    group_by=['product_id'], order_by=['product_id'])
                for i in purchase_order_details:
                    quantity_on_order = purchase_order_details.quantity.value

            res.append(
                {
                    'product_name': i.product_id.display_text,
                    'target_level': target_level,
                    'quantity_on_hold': quantity_on_hold,
                    'quantity_on_hand': quantity_on_hand,
                    'quantity_purchased': quantity_purchased,
                    'quantity_sold': quantity_sold,
                    'quantity_on_order': quantity_on_order
```

(continues on next page)

(continued from previous page)

```
                }
            )
            print(res)

    return res
```

Note: What was achieved with the above Python code is portability. It will execute against any database provider. However, the performance was not great. Hence, it was decided to develop the SQL instead of the above code.

- The *set_where* function deserves the separate topic though. For now, it is similar to specifying *WHERE* clause in SQL.

The SQL developed is much faster than the above code and runs on all supported databases as well:

```
SELECT
  p1.ID AS ID,
  p1.Product_Name AS Name,
  COALESCE(
    (SELECT SUM(od.quantity)
     FROM order_details od
     WHERE p1.ID = od.product_id AND od.status_id = 4),
    0) AS quantity_on_back_order,
  COALESCE(
    (SELECT SUM(it1.quantity)
     FROM inventory_transactions it1
     WHERE p1.ID = it1.product_id AND it1.transaction_type = 1),
    0) AS quantity_purchased,
  COALESCE(
    (SELECT SUM(it1.quantity)
     FROM inventory_transactions it1
     WHERE p1.ID = it1.product_id AND it1.transaction_type = 2),
    0) AS quantity_sold,
  COALESCE(
    (SELECT SUM(it1.quantity)
     FROM inventory_transactions it1
     WHERE p1.ID = it1.product_id AND it1.transaction_type = 3),
    0) AS quantity_on_hold,
  COALESCE(
    (SELECT SUM(po1.quantity)
     FROM purchase_order_details po1
     WHERE p1.ID = po1.product_id AND po1.posted_to_inventory = 0),
    0) AS quantity_on_order,
```

(continues on next page)

(continued from previous page)

```
  p1.target_level AS target_level
FROM products p1
GROUP BY p1.ID, p1.Product_Name, p1.target_level ORDER BY p1.ID
```

As seen, this was the iteration process from a quite *large* Access SQL, to a much smaller SQL.

Hence, we replace the Server Module code from the above with a new one containing the above SQL:

```
def get_records(item):
    res = []
    rows = []
    err = ''
    sql =   """SELECT
                  p1.ID AS ID,
                  p1.Product_Name AS Name,
                  COALESCE(
                    (SELECT SUM(od.quantity)
                     FROM order_details od
                     WHERE p1.ID = od.product_id AND od.status_id = 4),
                    0) AS quantity_on_back_order,
                  COALESCE(
                    (SELECT SUM(it1.quantity)
                     FROM inventory_transactions it1
                     WHERE p1.ID = it1.product_id AND it1.transaction_type = 1),
                    0) AS quantity_purchased,
                  COALESCE(
                    (SELECT SUM(it1.quantity)
                     FROM inventory_transactions it1
                     WHERE p1.ID = it1.product_id AND it1.transaction_type = 2),
                    0) AS quantity_sold,
                  COALESCE(
                    (SELECT SUM(it1.quantity)
                     FROM inventory_transactions it1
                     WHERE p1.ID = it1.product_id AND it1.transaction_type = 3),
                    0) AS quantity_on_hold,
                  COALESCE(
                    (SELECT SUM(po1.quantity)
                     FROM purchase_order_details po1
                     WHERE p1.ID = po1.product_id AND po1.posted_to_inventory = 0),
                    0) AS quantity_on_order,
                  p1.target_level AS target_level
                FROM products p1
                GROUP BY p1.ID, p1.Product_Name, p1.target_level ORDER BY p1.ID"""
```

(continues on next page)

(continued from previous page)

```
    rows = item.task.execute_select(sql)

    for r in rows:
        quantity_on_hand = 0

        if r[3] and r[4]:
            quantity_on_hand = r[3] - r[4]
        else:
            quantity_on_hand = 0

        res.append(
            {
                'product_id': r[0],
                'product_name': r[1],
                'quantity_on_back_order': r[2],
                'quantity_purchased': r[3],
                'quantity_on_hand': int(quantity_on_hand),
                #'quantity_on_hand': str(quantity_on_hand),
                'quantity_sold': r[4],
                'quantity_on_hold': r[5],
                'quantity_on_order': r[6],
                'target_level': r[7]
            }
        )
    return res
```

This is it. We now have a fully functional replacement for MS Access SQL. The effort in writing the Python code and the SQL was pretty much the same, except SQL is much faster.

Note: Or is it?

Slow Virtual Table?

The SQL is fast. However, the `on_after_open`, or simply put, displaying the data is not. This is because for every single record added, deleted, or modified, the controls (DOM elements that display the data), are data aware! Which means checked and appropriately updated for every-single-row. And that is adding a significant delay.

So what do we do? We "disable" the controls temporary, while we're "filling" the table. We "enable" the controls after finished.

We do that with:

`disable_controls``()`

and:

enable_controls()

In the official *Demo*, the same principle is used when `Tax` field is modified to recalculate all Items tax on *Invoice*.

The final `on_after_open` looks like this:

```
function on_after_open(item) {
    item.alert('Working!');
    item.server('get_records', function(records) {
        item.disable_controls();                     // <-- disable DOM controls
        try {                                        // <-- try is a bast practise␣
↪with JS
            records.forEach(function(rec) {
                item.append();
                item.id.value = rec.product_id;
                item.product_name.value = rec.product_name;
                item.target_level.value = rec.target_level;
                item.quantity_on_hold.value = rec.quantity_on_hold;
                item.quantity_on_order.value = rec.quantity_on_order;
                item.quantity_on_back_order.value = rec.quantity_on_back_order;
                item.quantity_on_hand.value = rec.quantity_on_hand;
                item.quantity_purchased.value = rec.quantity_purchased;
                item.quantity_sold.value = rec.quantity_sold;
                item.post();
            });
            item.first();
        }
        finally {
            item.enable_controls();                  // <-- enable DOM controls
        }
    });
}
```

This concludes the Virtual Table used for the *Inventory List*. We should be able to build a Dashboard as seen on the MS Access template *Startup*:

```
- Inventory to Reorder
- Active Orders
```

Using a DB Views

It is absolutely possible to use the *DB Views* instead of Virtual Tables. The *smaller* SQL would need to be modified for `quantity_on_hand` calculation. However, because the SQL is executing on the server side, we did not see any benefits with using the SQL view for this example.

Master/Details Edit

- In order to implement Master/Details functionality, the *PO Detail* and *Order Detail* Group Item was created, and the related tables were imported into this Item Group. This enables the functionality to add Detail to any Master table for viewing and editing.

Master/Details View

- For Master/Details View only, with no need for editing, there is no need to do anything special other than to add a code to display the Details for a Master table.

  Below is the code which will display for each *Supplier* the relevant *Purchase Order* as Details:

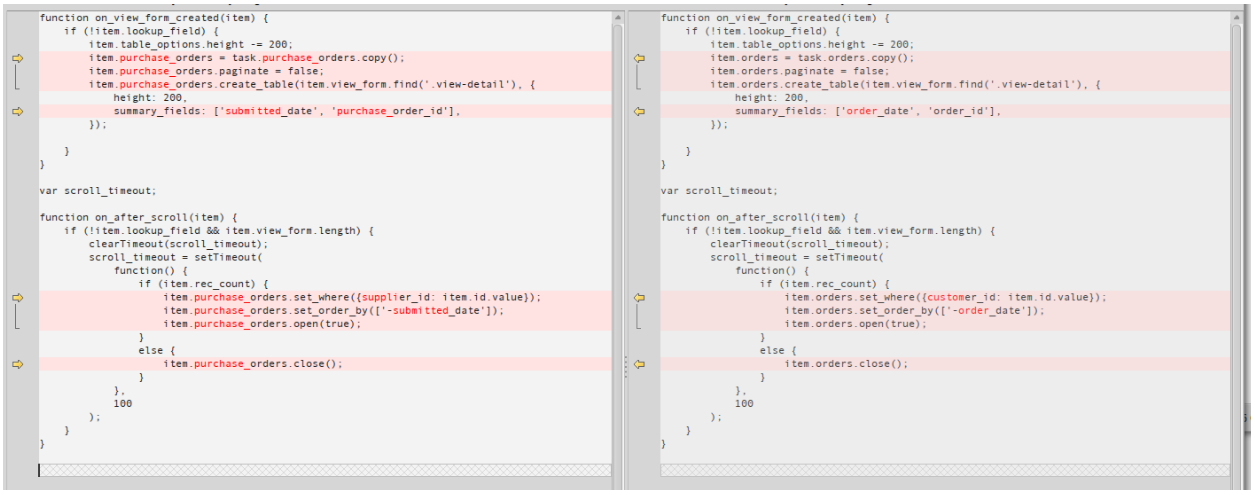
```
function on_view_form_created(item) {
    if (!item.lookup_field) {
        item.table_options.height -= 200;
        item.purchase_orders = task.purchase_orders.copy();
        item.purchase_orders.paginate = false;
        item.purchase_orders.create_table(item.view_form.find('.view-detail'), {
            height: 200,
            summary_fields: ['submitted_date', 'purchase_order_id'],
        });

    }
}

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.purchase_orders.set_where({supplier_id: item.id.value});
                    item.purchase_orders.set_order_by(['-submitted_date']);
                    item.purchase_orders.open(true);
                }
```

(continues on next page)

(continued from previous page)

```
                else {
                    item.purchase_orders.close();
                }
            },
            100
        );
    }
}
```

Here we showing the difference with code used for ie *Suppliers* and *Customers*. It is the similar functionality to display *Orders* as Details for each Customer:



```
function on_view_form_created(item) {
    if (!item.lookup_field) {
        item.table_options.height -= 200;
        item.purchase_orders = task.purchase_orders.copy();
        item.purchase_orders.paginate = false;
        item.purchase_orders.create_table(item.view_form.find('.view-detail'), {
            height: 200,
            summary_fields: ['submitted_date', 'purchase_order_id'],
        });

    }
}

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.purchase_orders.set_where({supplier_id: item.id.value});
                    item.purchase_orders.set_order_by(['-submitted_date']);
                    item.purchase_orders.open(true);
                }
                else {
                    item.purchase_orders.close();
                }
            },
            100
        );
    }
}
```

```
function on_view_form_created(item) {
    if (!item.lookup_field) {
        item.table_options.height -= 200;
        item.orders = task.orders.copy();
        item.orders.paginate = false;
        item.orders.create_table(item.view_form.find('.view-detail'), {
            height: 200,
            summary_fields: ['order_date', 'order_id'],
        });

    }
}

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.orders.set_where({customer_id: item.id.value});
                    item.orders.set_order_by(['-order_date']);
                    item.orders.open(true);
                }
                else {
                    item.orders.close();
                }
            },
            100
        );
    }
}
```

Note: As seen, almost the same code is used for all Master/Details Views for *Products*, *Employees* and *Shippers*. It is really simple to add an Detail table to a Master table when knowing the Primary Key.

Lookup Lists

There are many *Drop down* lists we could move to a Lookup List, for example for *Order Details* table *Status ID* field has:

```
None
Allocated
Invoiced
Shipped
On Order
No Stock
```

Or, as already mentioned, *Purchase orders* and *Orders* table *Payment method* field:

```
Payment Method
```

However, this usually means splitting the table in two tables, *Orders* and *Payment_method* for the Lookups. Or adding a Lookup List *Payment Method* and pointing *payment_method* to it. Which changes TEXT to INT in the Builder, and stores INT in the database.

In legacy applications, ie. Imported tables from live system, first option is not possible due to table changes. However, second option is not possible either, because it will store the INT in a table!

The solution is to store the TEXT in tables, as this is acceptable for Imported tables:

```
function on_field_changed(field, lookup_item) {
    let item = field.owner;
    if (field.field_name === 'payment_type') {
        item.payment_type.value = field.display_text;
    }
}
function on_edit_form_created(item) {
    if (item.payment_type.value === 'Credit Card') {
        item.payment_type.value = "Credit Card";
    }
    else if (item.payment_type.value === 'Cash') {
        item.payment_type.value = "Cash";
    }
    else if (item.payment_type.value === 'Check') {
        item.payment_type.value = "Check";
    }
}
```

We still need a Lookup List ie. “Payment Type” with the values, but the stored value is TEXT, and not INT.

Analytics

This is where Python and JavaScript shines and the chapter almost needs a separate article, due to Python and JavaScript libraries introduction. To name a few, the application is using Python libraries pandas, matplotlib, numpy, RFM and JavaScript is using PivotJS and ChartJS.

Bulk update, insert or delete

The *Orders* table has a *Data Pump* option which will insert a number of records, and update the relevant rows in the database. In addition, the *Purchase Orders* table has the same as well as *Products* table.

Note: To Be Continued⋯

6.5.3 Inventory transactions

The application lives here:

https://assetinventory.pythonanywhere.com

This is interesting template due to a number of reports. There is no VBA at all. However, there is a logic within the reports to reorder the Inventory based on *Reorder Level*, *Current Stock*, and *Target Stock Level*.

The complete application was developed in a few hours` time. Below is the approximate process:

- The template was uploaded to https://jampyapplicationbuilder.com/conv/ to give us all tables and data needed for the project.
- All tables were imported into the Jam.py Application Builder from the provided link after the upload.
- The Lookup Lists are within the forms, namely:

```
Category
Location
TransactionType
```

- Because how the Access application works, it might be a good decision to convert the Lookup Lists into tables. That way we might utilise the Jam.py reports and Dashboard features with no additional code needed. The problem with any Lookup Lists is the table relationship, which can not be built with the SQL, and must be addressed with code. This is how *Personal Account Ledger* was built.
- A code from previous examples was applied in the same or similar manner.

Note: To Be Continued when/if more features were added ⋯

CHAPTER

SEVEN

ACKNOWLEDGEMENTS

Here we acknowledge people who influenced `Jam.py` development or made a contribution themselves.

Especially to:

- Andrew Yushev, Ph.D, author of `Jam.py`.
- Maxwell Morais, developer and contributor, who created fascinating applications (ERP POC, POS, etc) with `Jam.py` and shared his work, and supported me when having issues with the CSS.
- Fabio Lenzarini, developer and contributor, who created a multitude of changes regarding database views, languages translations with google translator engine, to name a few. Also shared ERP POC application.
- Alistair Bates, developer and contributor, who helped a number of people over the years and shared his knowledge.
- Marko Pandžić, developer who helped with MS Access to SQLite conversion.
- Danijel Kaurin, developer and contributor, who also helped a number of people over the years and shared his knowledge.

7.1 About the author

Dean Dražen Babić, BSc, is `Jam.py` enthusiast, developer and promoter. Dean was a frequent public speaker at the Stirling Toastmasters Club as CC (ex VPE, ex VPPR), and currently residing in WA.

7.1.1 Copyright

INDEX