Jam.py Design Tips

Jam.py Team

Jul 01, 2022

CONTENTS

CHAPTER

ONE

JAM.PY APPLICATION DESIGN TIPS

Welcome to Jam.py! If you are new to Jam.py or application development, this is the place to find some tips.

1.1 How the documentation is 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.

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 Python version

Since Jam.py is a 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 /howto/windows useful.

2.1.2 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:

```
source ~/Downloads/py3.9/bin/activate
pip install jam.py
pip install pandas
```

The above command will install jam.py and pandas packages into `~/Downloads/py3.9/lib/python3.9/site-packages/` directory. We can have as many virtual environments as needed.

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

2.1.3 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.

2.1.4 Unit Testing

Jam.py version higher than 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.

2.1.5 Continuous Integration (CI)

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

2.1.6 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 Documantation:

```
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.

2.1.7 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
```

2.1.8 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.

2.1.9 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.

2.1.10 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 and the Application can be shipped *as is*, with just *jam* folder from Jam.py distribution copied into the Application folder.

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 than 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.

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 and SQLite.

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, you'll need to make sure your Python database bindings are installed.

- If you're using PostgreSQL, you'll need the psycopg2 package.
- If you're using MySQL or MariaDB, you'll need a `mysqlclient` for Python 2.x or `pymysql` for Python 3.x.
- If you're using Oracle, you'll need a copy of cx_Oracle.

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. The same applies as for database triggers.

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.

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

3.1.1 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.

1. What is Jam.py?

Jam.py is a Rapid Application Development framework. The word rapid should be stressed.

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. Ne need for anything more than 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. Why not to use Jam.py?

If not using databases, then it probably doesn't make sense to use Jam.py.

4. Does it scale?

Yes. That is absolutely true with containers.

5. What can I use it for?

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

3.1.2 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.

3.1.3 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 it's 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 an 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 cant 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 enabled by default, no need for any utility.*

3.1.4 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.*

3.1.5 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 an 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 than 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.*

3.1.6 Reports

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

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

> *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.*

3.1.7 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.

3.1.8 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.

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.

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 a like, 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.

3.2 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.

3.2.1 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 a Application specific requirements. There might be also an requirement to store the two mentioned tables outside the admin.sqlite database for improved security by using some other database than 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.

3.2.2 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 *build in* method.

3.2.3 External Authentication

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 connection to LDAP or AD.

SAML or SSO Authentication

In addition to LDAP, SAML will be supported in newer major versions. 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: http://auth.jampyapplicationbuilder.com

OAuth2, OpenID or SiteMinder (CA) Authentication

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

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

3.2.4 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.

3.2.5 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

CHAPTER

FOUR

"HOW-TO" GUIDES

Here you'll find short answers to "How do I....?" types of questions.

CHAPTER

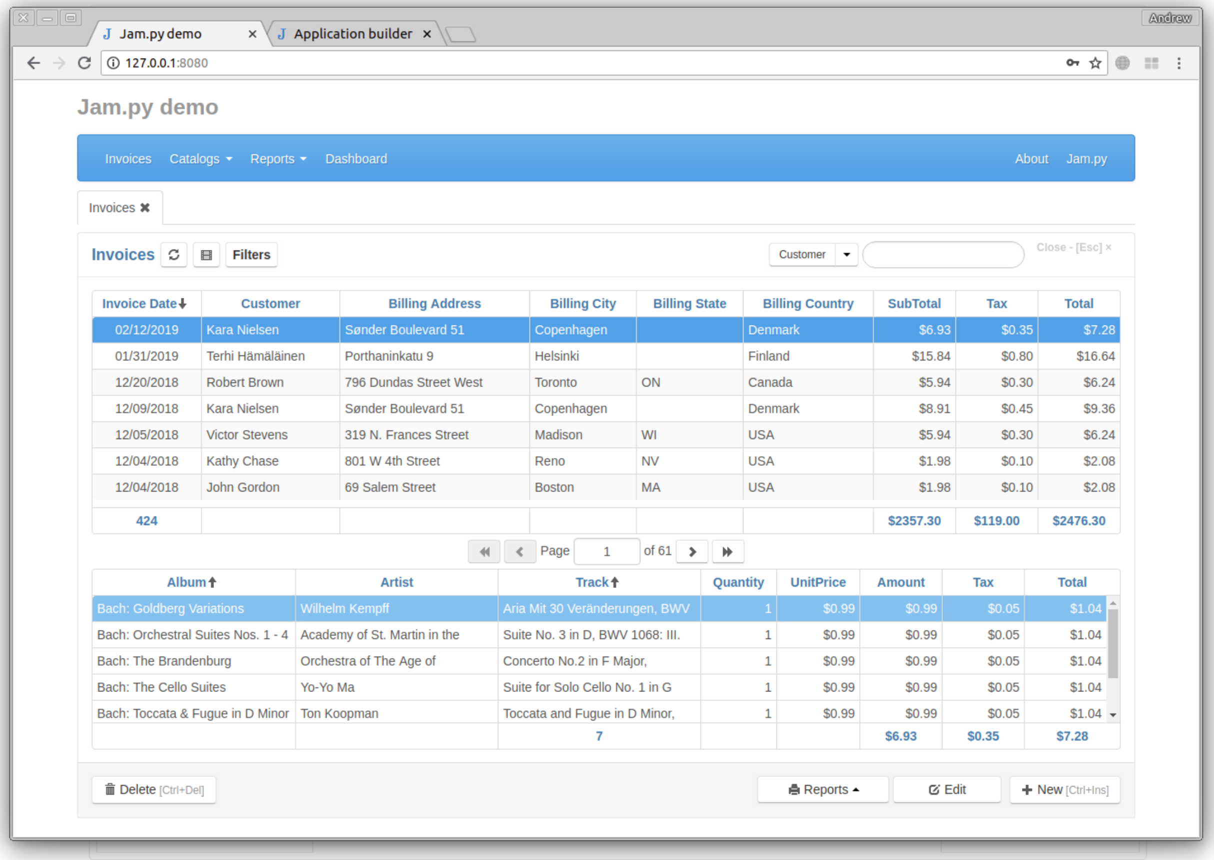
FIVE

HOW WAS DEMO BUILT?

So with no 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

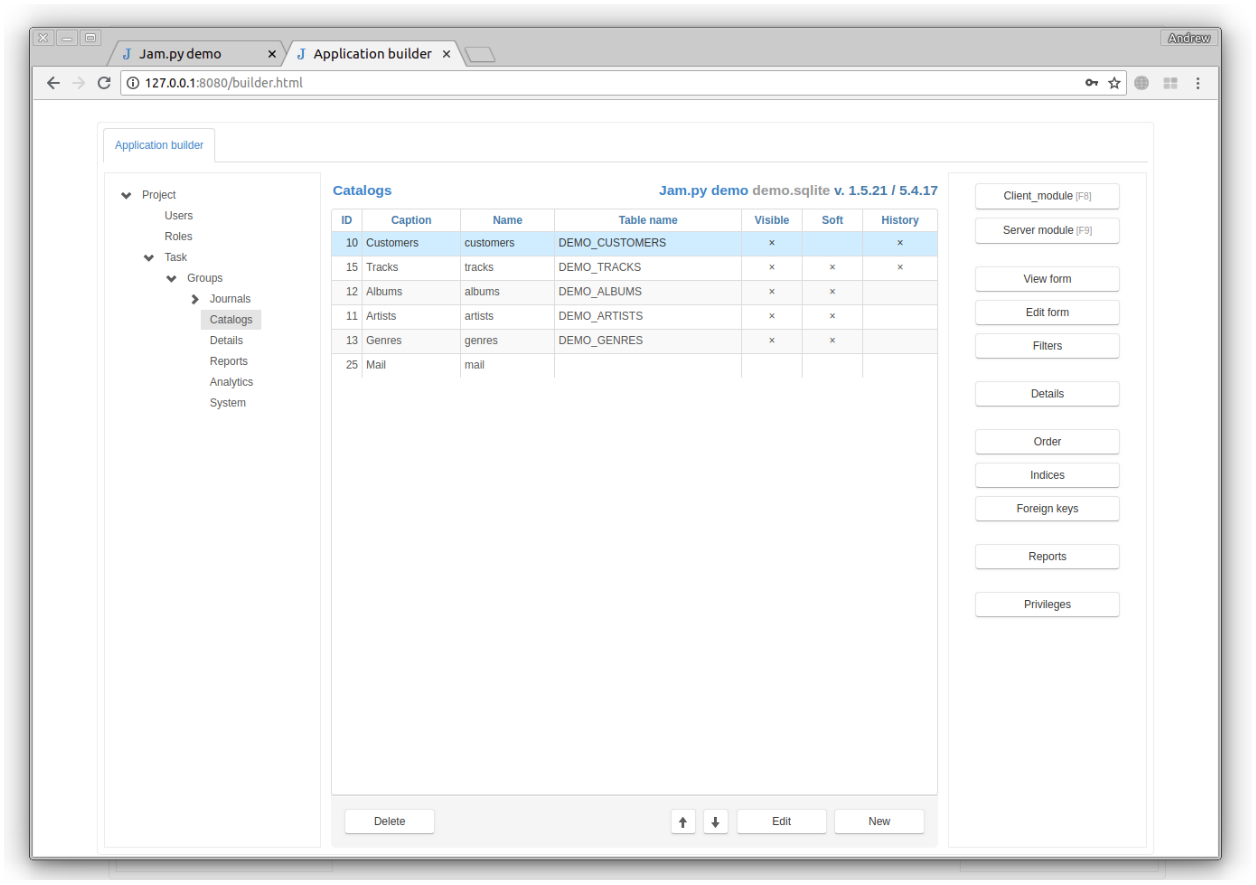
After downloading the Jam.py package, and starting Demo, the application is accessible with typing 127.0.0.1:8080/index.html or just 127.0.0.1:8080 in the Browser:



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:



Catalogs — Jam.py demo demo.sqlite v. 1.5.21 / 5.4.17

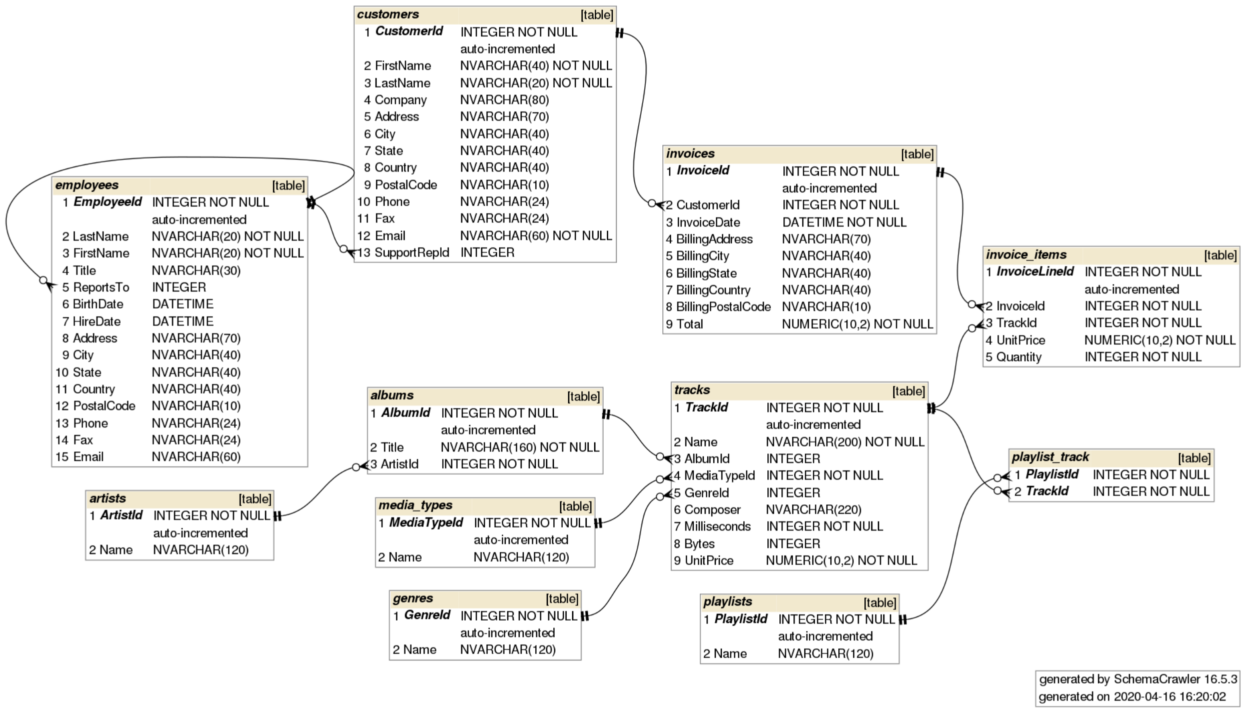
ID	Caption	Name	Table name	Visible	Soft	History
10	Customers	customers	DEMO_CUSTOMERS	×		×
15	Tracks	tracks	DEMO_TRACKS	×	×	×
12	Albums	albums	DEMO_ALBUMS	×	×	
11	Artists	artists	DEMO_ARTISTS	×	×	
13	Genres	genres	DEMO_GENRES	×	×	
25	Mail	mail				

The Application is derived from Open Source Chinook Database. It is an Invoicing example with music tracks as invoiced products, Customers, Albums, etc. Basically, the application creates a Customer details, product details, and raises Invoices for Customers. That is it. Cool little application, though.

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

5.1.1 Demo database

To better understand the actual database, here is the original Chinook database relational 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 tables Primary and Foreign Keys, which are essentially a table *Lookup fields* used within Jam.py. More about that latter.

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?

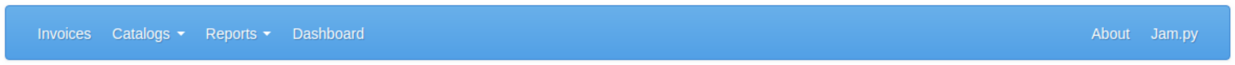
5.1.2 What to expect?

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 is 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 is providing many bells and whistles, or so called batteries, out of shelf. However, Jam.py is not "Out of Shelf 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 the 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 specialised in doing so. Jam.py is a specialised framework. It is the right tool for the right job. Let's see what can we expect from it.

DropDown Menu(s)

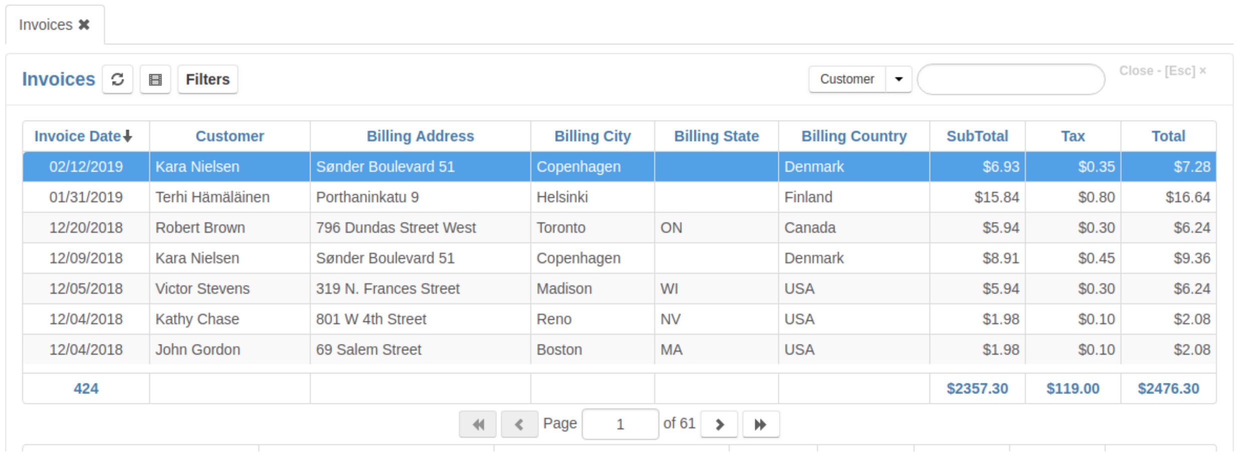
On the below menu all options on the left are created automatically. The “About” and “Jam.py” on the right side is a code. We can build complete menu manually, not to worry. Will get there soon.



Data Grid(s)

The Demo Invoices data grid is called a Journal. In Accounting, this are General Ledger tables. In databases, this is a Master Table. This grid is presented automatically, with pagination on the bottom and can contain History button, refresh and Filters button. This are the default options, and can be turned off. If some none default button is needed, it can be added with a code though. Did I mention 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 a number of records for other fields. As seen, the Invoices data grid is presented as Tab. As we open more grids, more Tabs will appear here automatically. Tabs Caption can be changed with a code. What we see initially is the default.

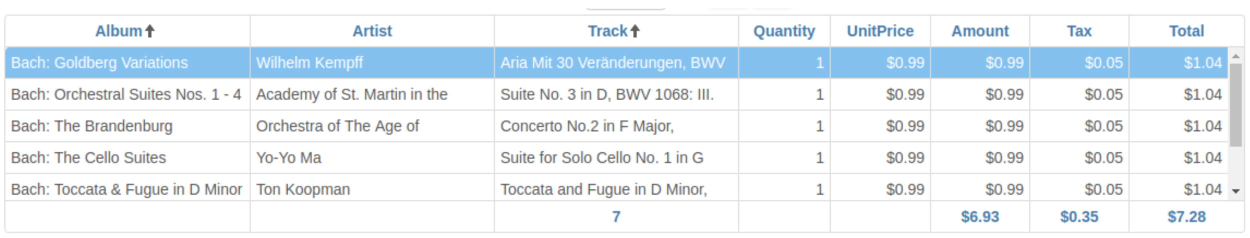


Invoice Date↓	Customer	Billing Address	Billing City	Billing State	Billing Country	SubTotal	Tax	Total
02/12/2019	Kara Nielsen	Sønder Boulevard 51	Copenhagen		Denmark	$6.93	$0.35	$7.28
01/31/2019	Terhi Hämäläinen	Porthaninkatu 9	Helsinki		Finland	$15.84	$0.80	$16.64
12/20/2018	Robert Brown	796 Dundas Street 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

More Data Grid(s)

The Demo Invoices data has a Details grid. This 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? Will get there.

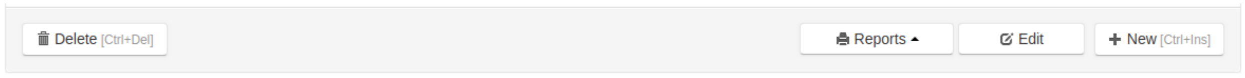
As above grid, each Details grid is presented in Tabbed format. More Details added to an Journal, more Tabs we see in here.



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

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.



Ok, do not want to go further, just a minimum to get us going.

Any questions?

So now that we 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 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 for all scenarios possible. Or a team of dedicated developers.

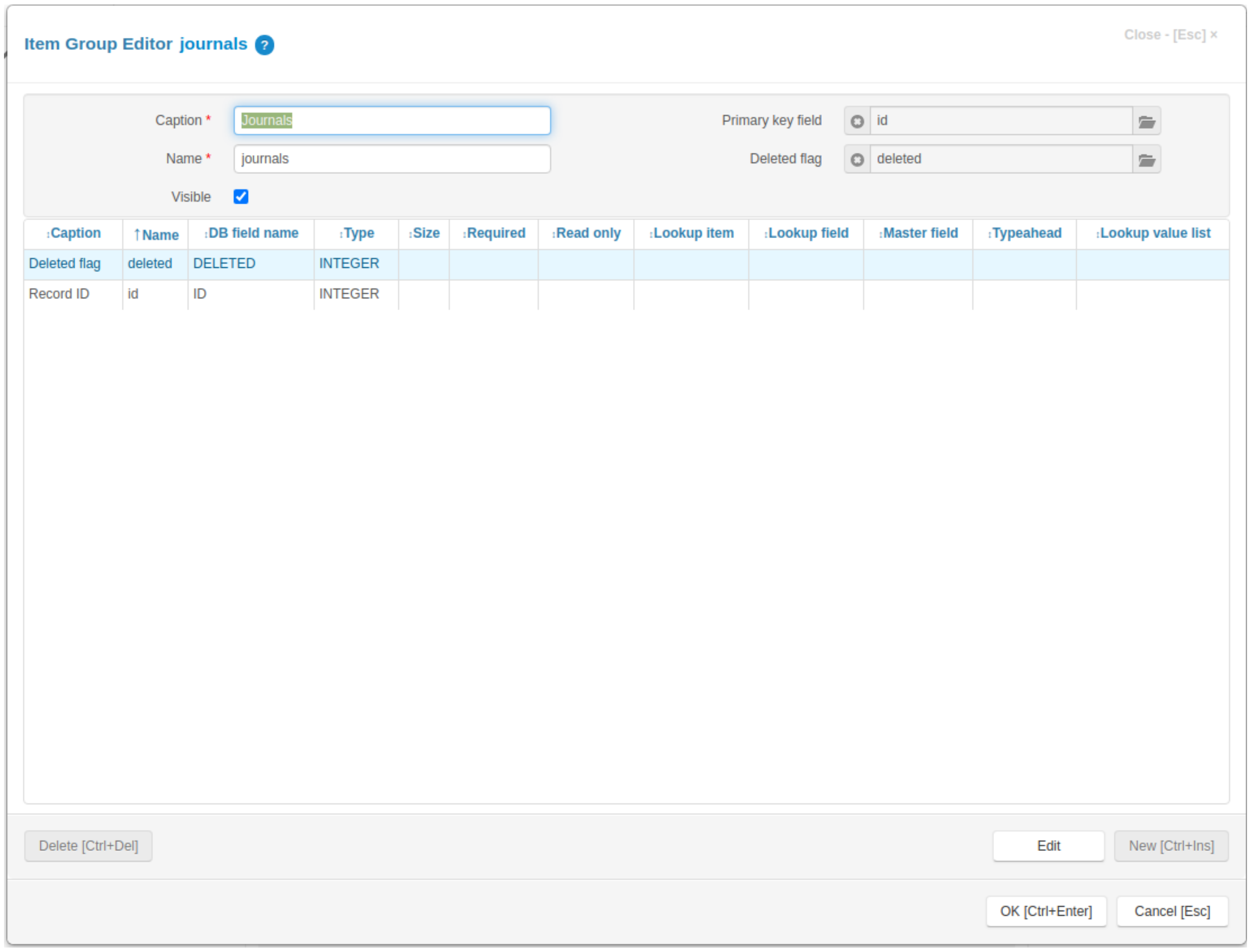
I am ignoring the Reports and Analytics for now, visible on Demo application. Concentrating only on no-code or low-code. Both mentioned features require some coding experience. Not much, but still needed.

5.1.3 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 sort of top to bottom approach!

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

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 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 "Find" built-in option to search where some Name is used.

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

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

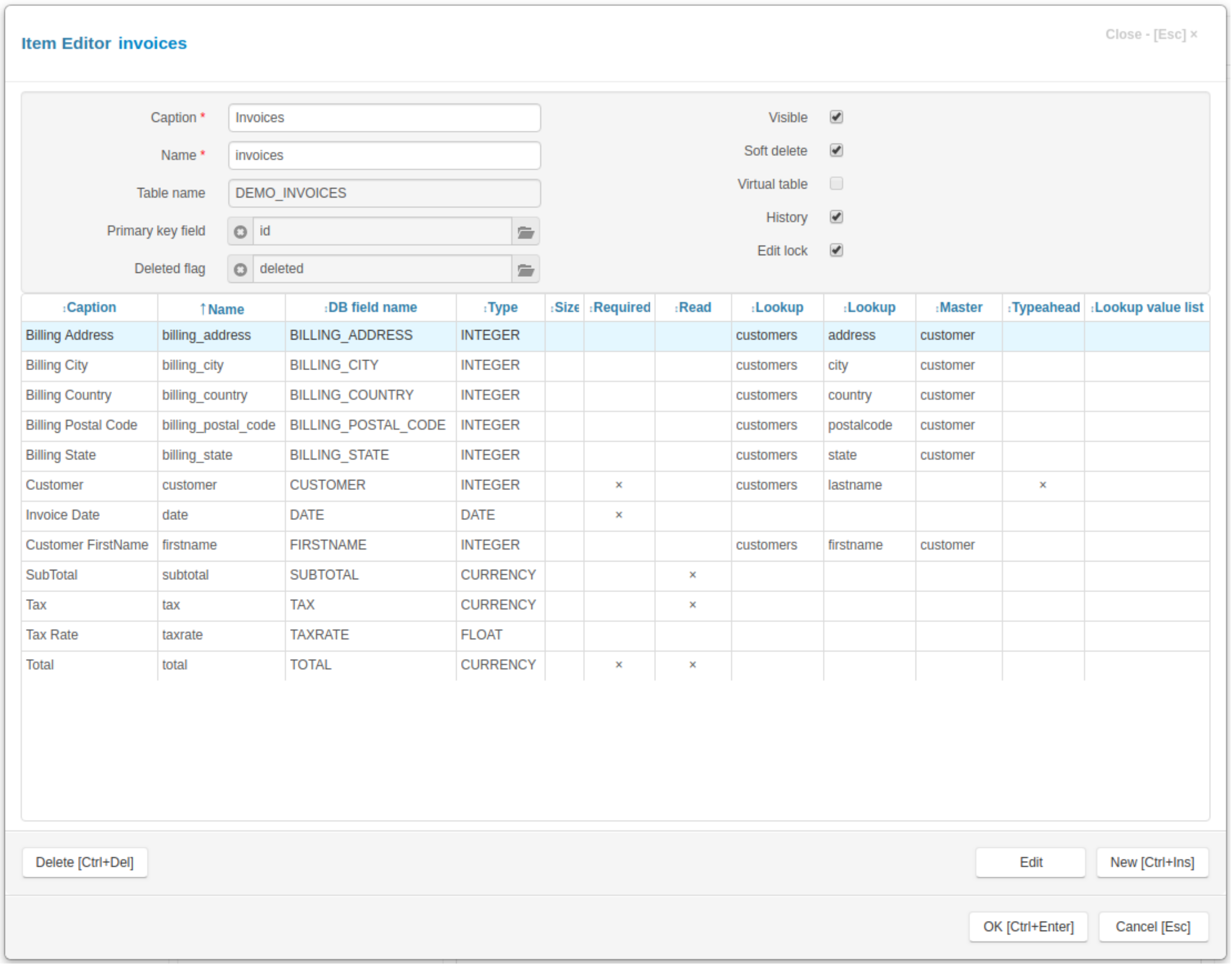
Note: *Master fields are not real fields in the database, they get populated when we select a value in the Lookup field.*

This is actually very powerful feature in my books, because it quickly identifies stuff. Which stuff we might ask? Anything built as a *Lookup fields*. So we set the "Master Field" to "Customer", for everything related to "Customers" table we are pulling the data from.

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

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

The Demo Invoices table:


Item Editor invoices

Close - [Esc] ×

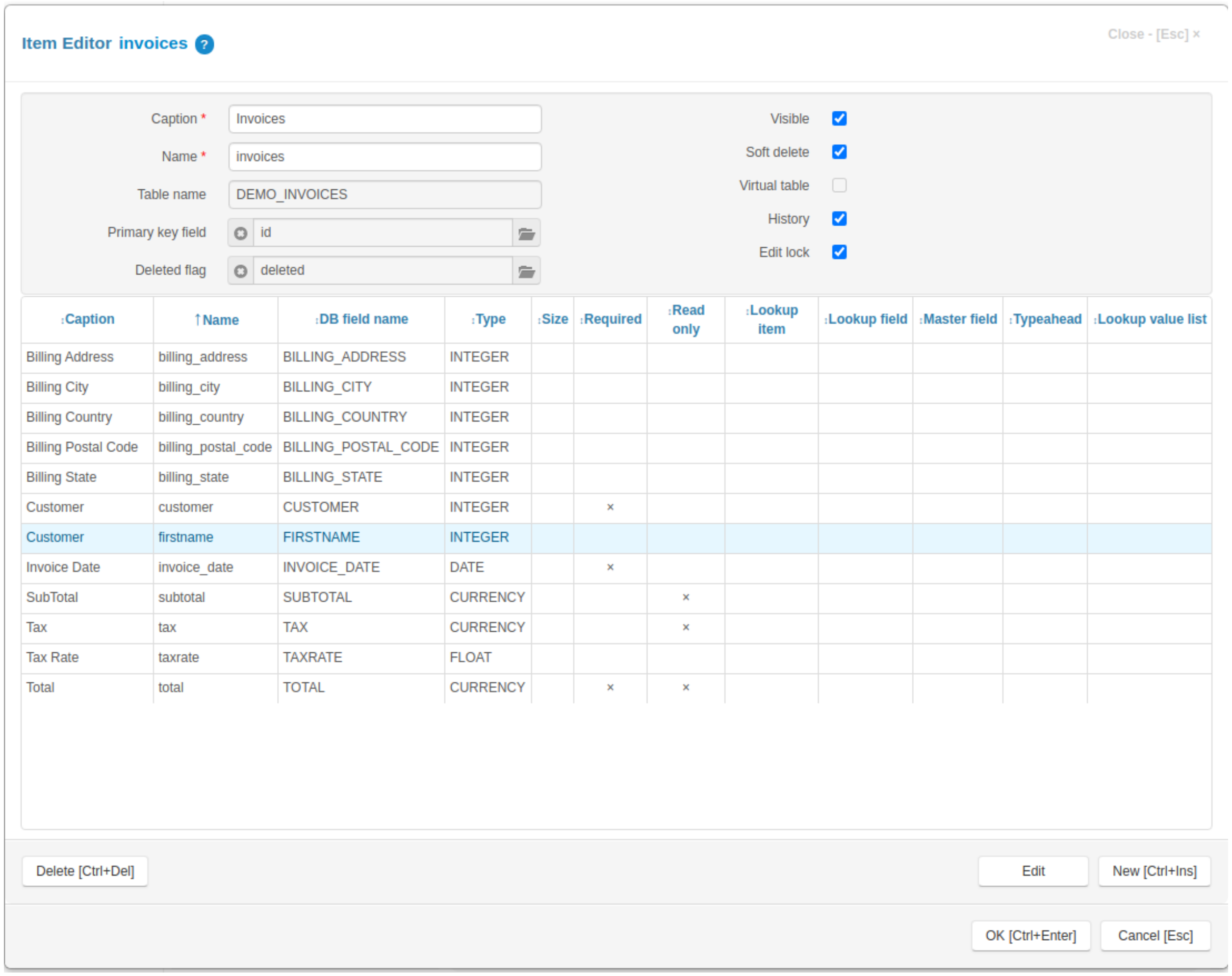
Caption *	Invoices
Name *	invoices
Table name	DEMO_INVOICES
Primary key field	id
Deleted flag	deleted
Visible	✓
Soft delete	✓
Virtual table	
History	✓
Edit lock	✓

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 cant really achieve what is needed for Invoices to function correctly.

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



Caption *	Invoices	Visible	✓
Name *	invoices	Soft delete	✓
Table name	DEMO_INVOICES	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
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		×	×					

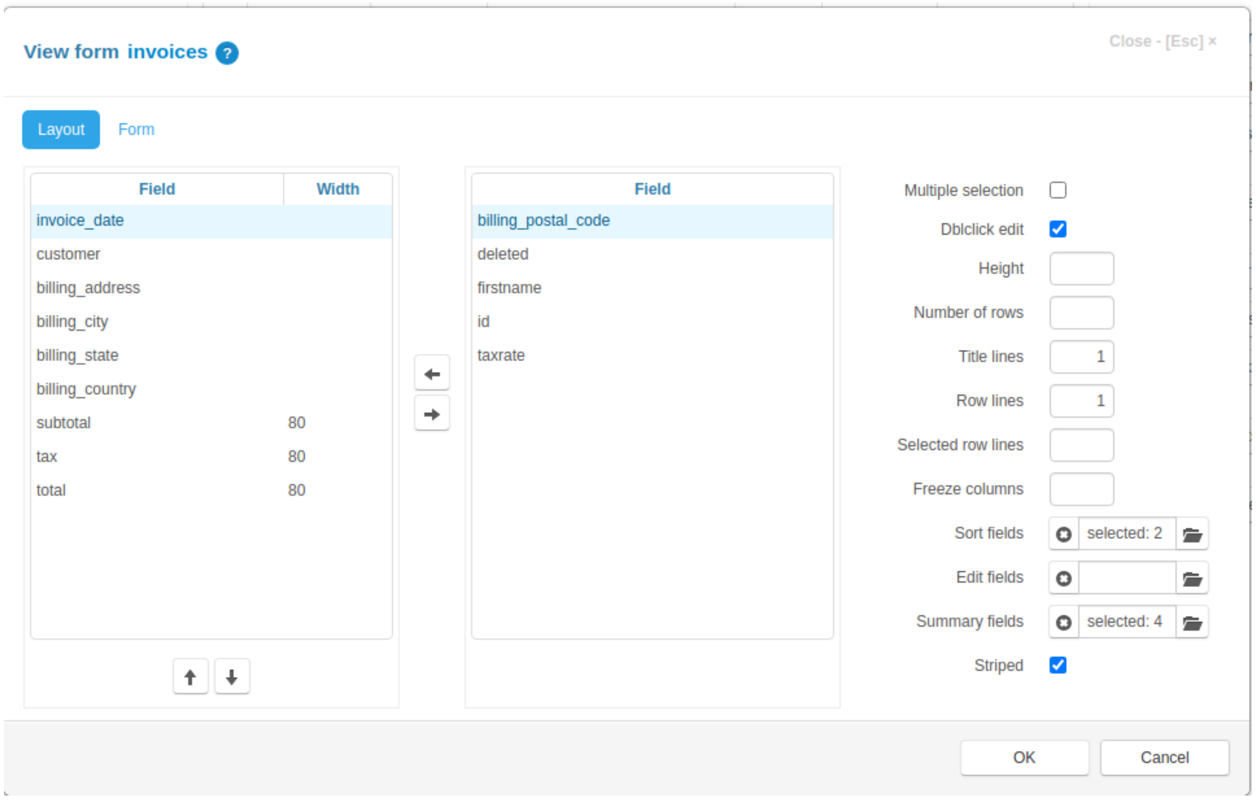
It is exactly the same thing, same as above one would think. Not really. It would not display any data because Jam.py would assume that all of Customers data is coming from one, and only one table, which is Invoices table. It is also true that all fields would have the Integer Type, which is not quite 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 an “View Form” option on the right hand side.

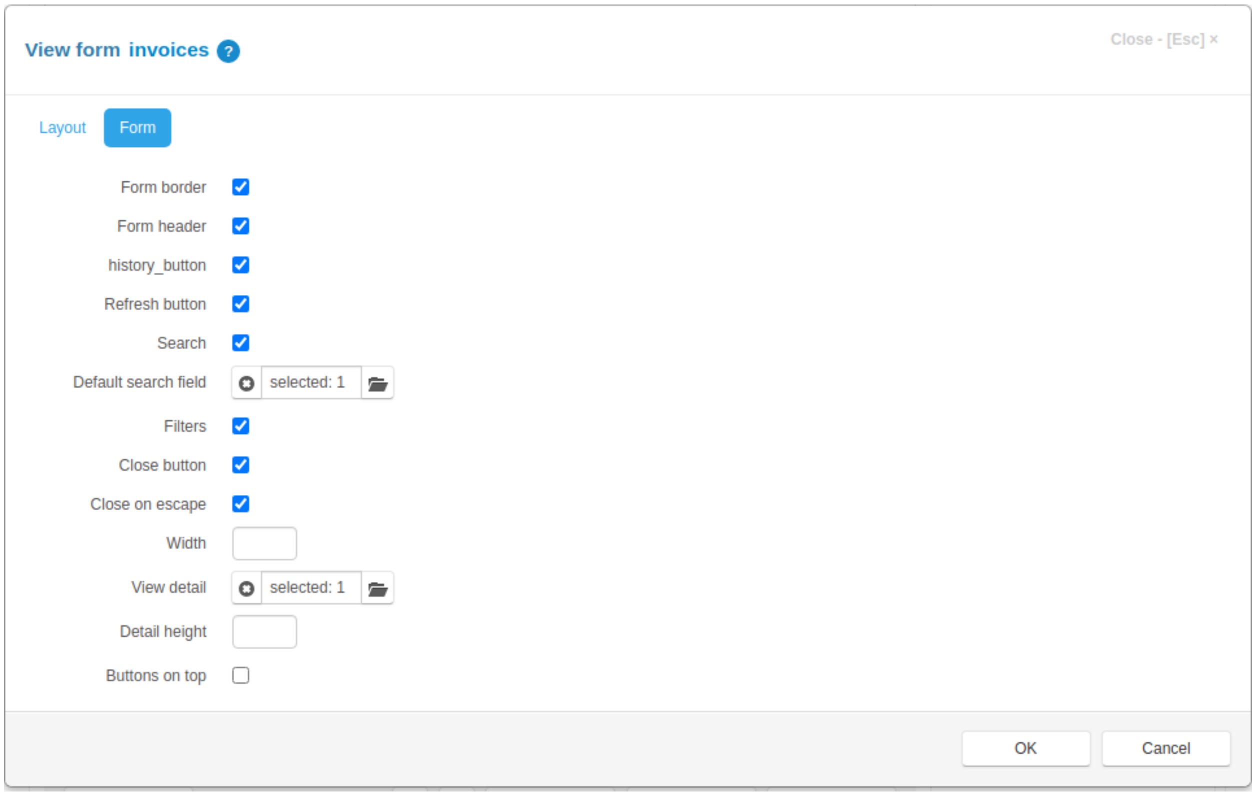


Field	Width
invoice_date	
customer	
billing_address	
billing_city	
billing_state	
billing_country	
subtotal	80
tax	80
total	80

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 than that.

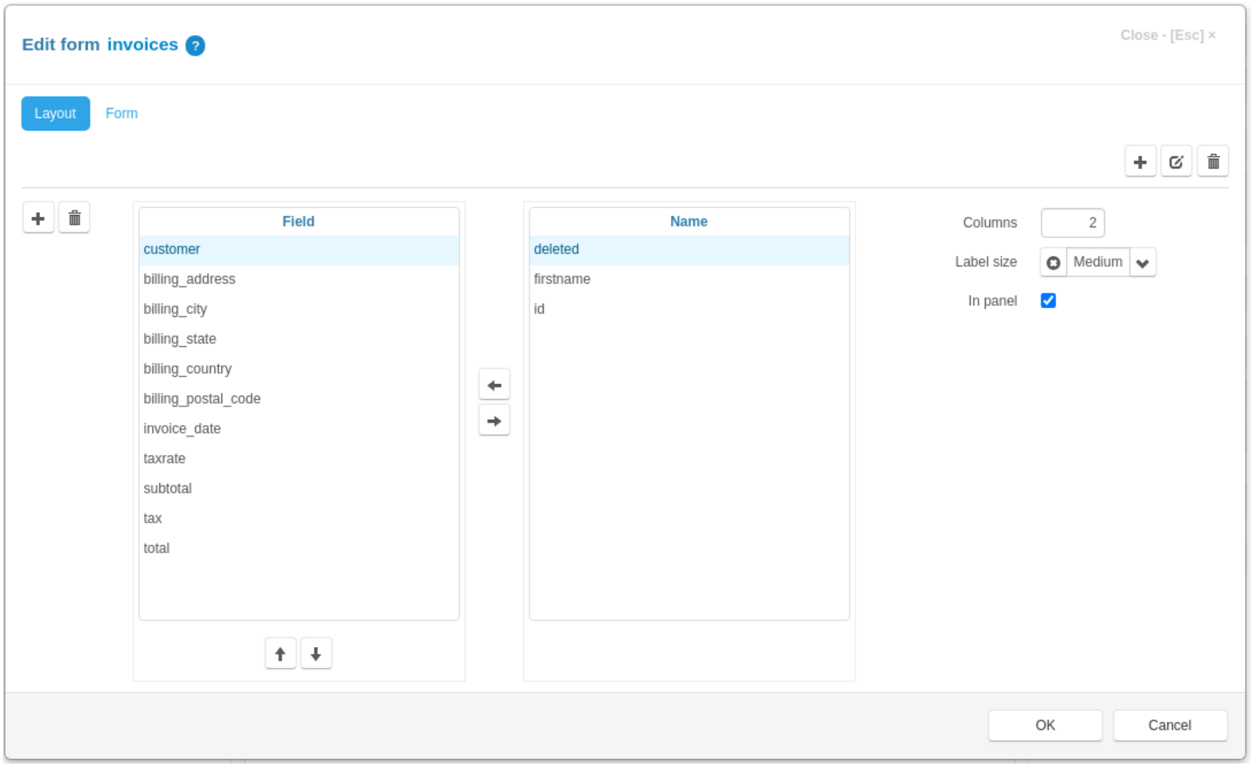


View form invoices

Layout | Form

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.



Edit form invoices

Layout | Form

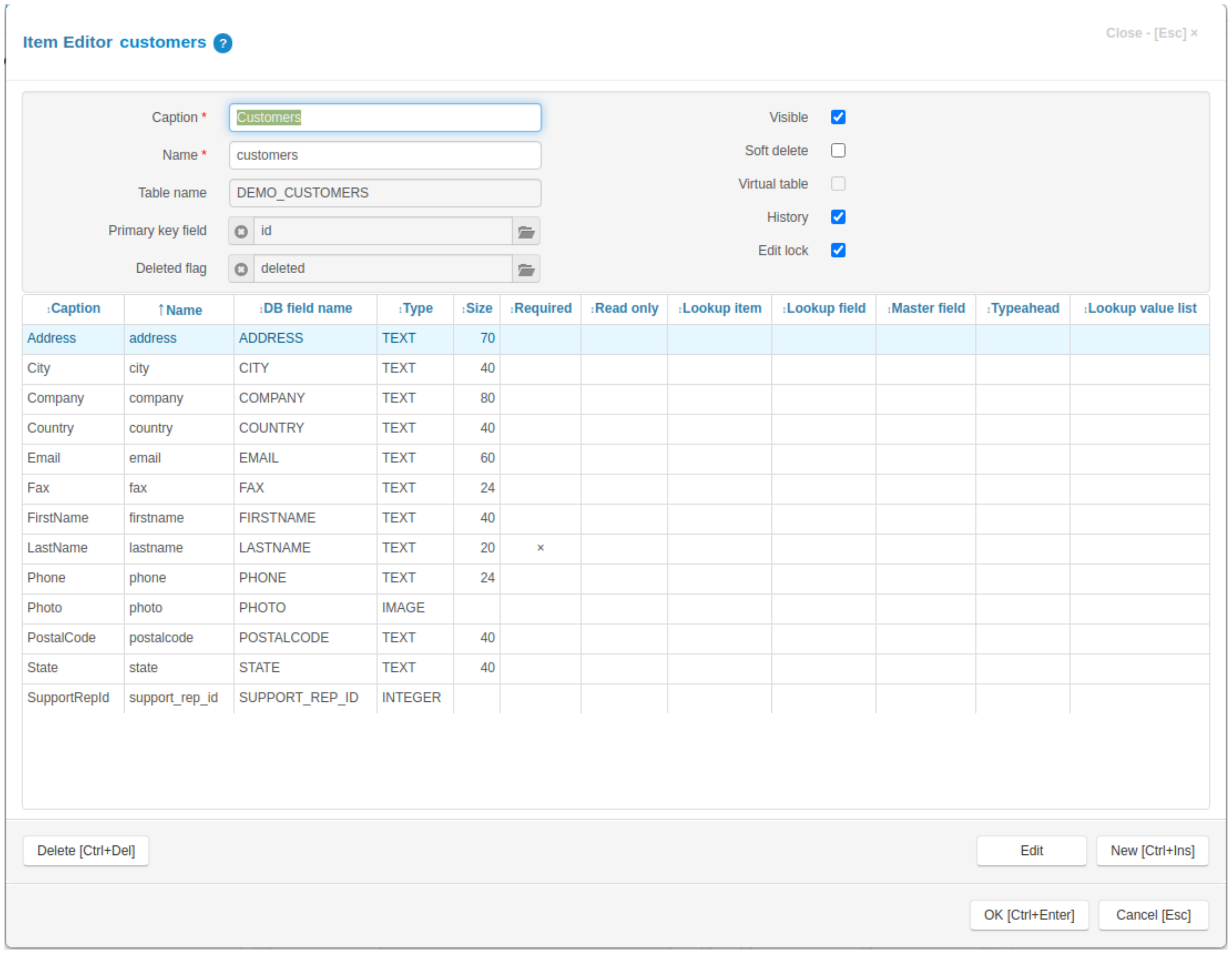
Field	Name
customer	deleted
billing_address	firstname
billing_city	id
billing_state	
billing_country	
billing_postal_code	
invoice_date	
taxrate	
subtotal	
tax	
total	

Columns	2
Label size	Medium
In panel	✓

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:



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

Invoice items

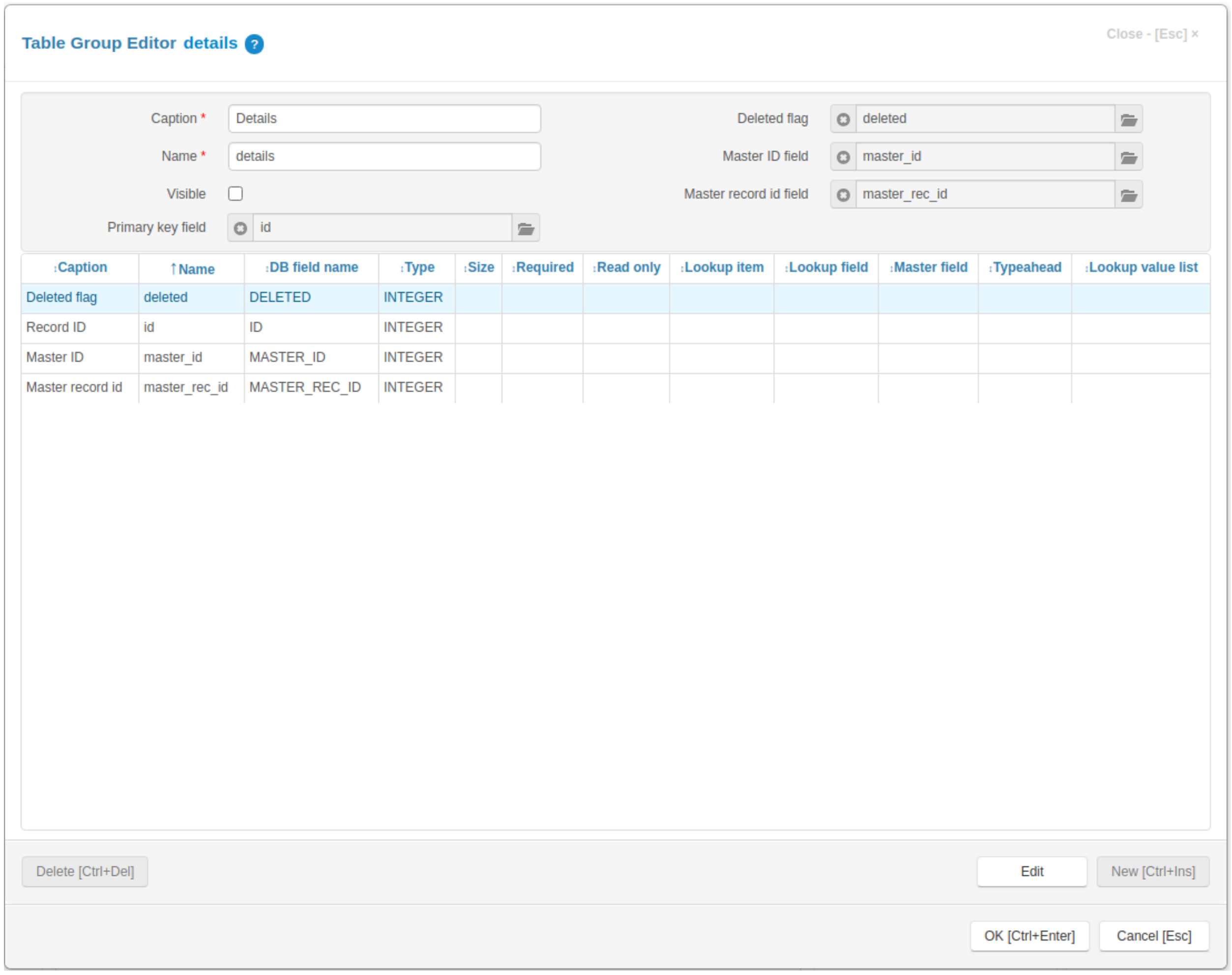
To automatically add some details to some other table, with no code what so ever, we need to use Jam.py Details feature.

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, it has the master_rec_id field that stores a reference to invoice that has this record, we can show the user an invoice that contains the current sold record. Clear as mud?

The Details Group differs from other default tables slightly:



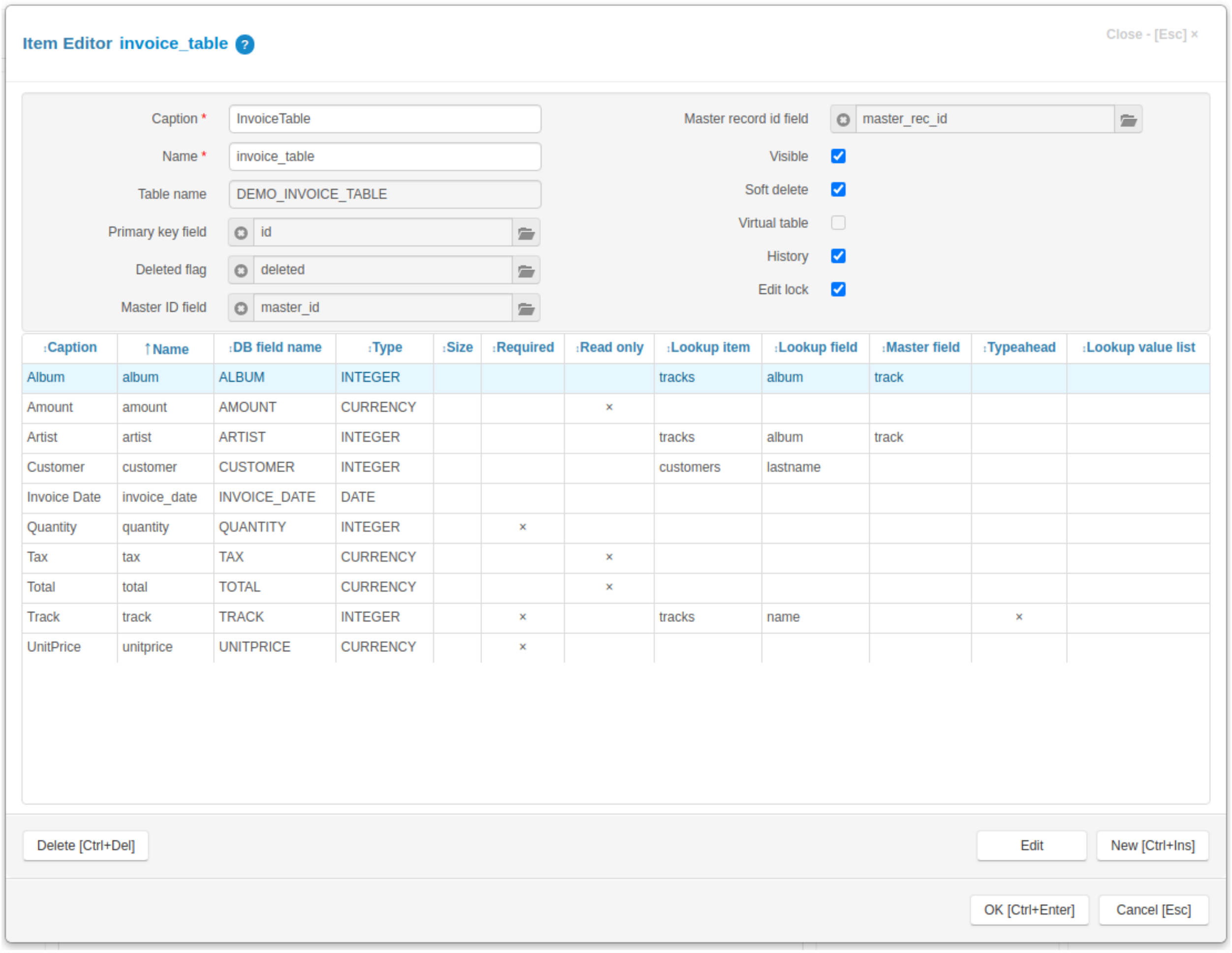
Caption *	Details	Deleted flag	deleted
Name *	details	Master ID field	master_id
Visible	☐	Master record id field	master_rec_id
Primary key field	id		

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.

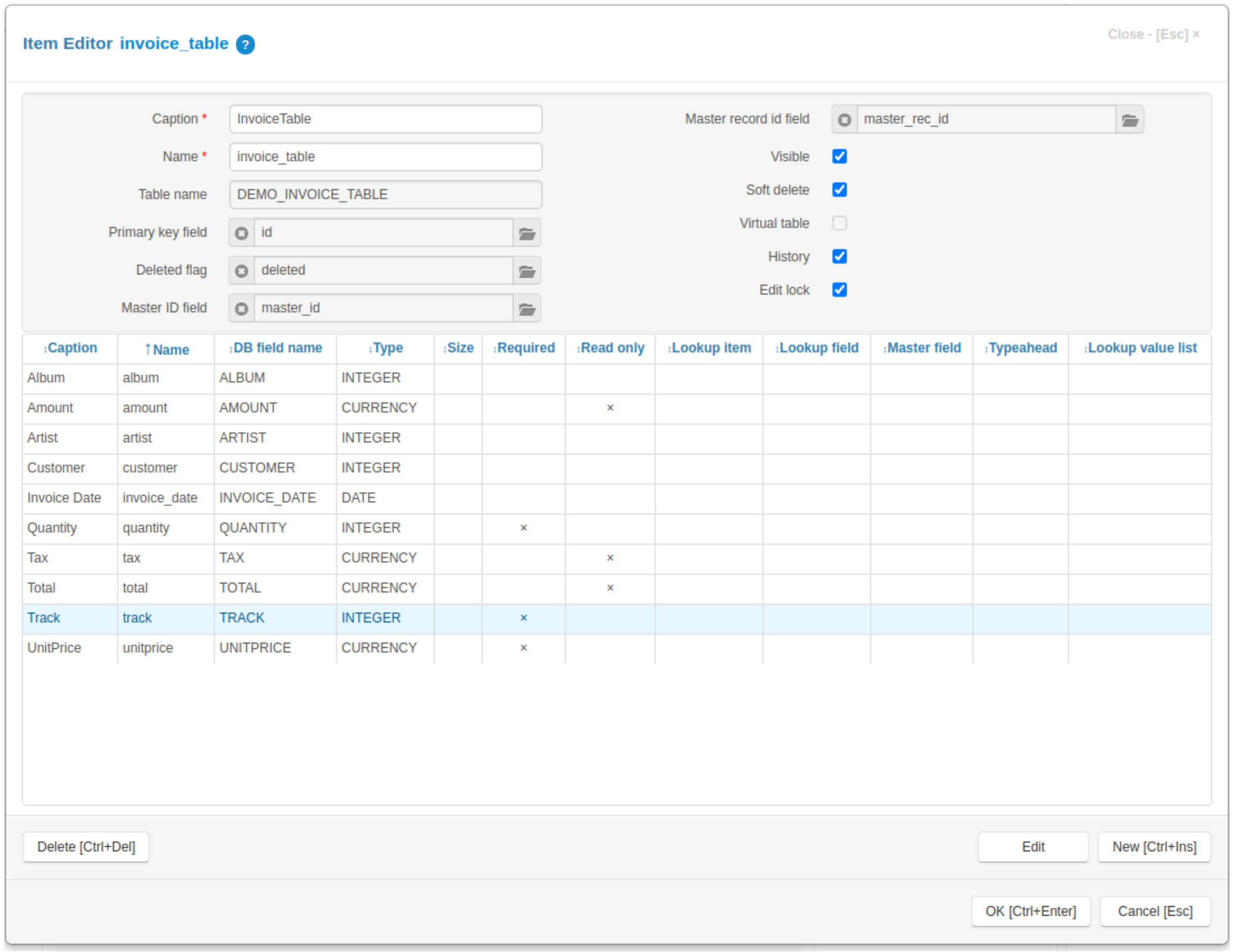
Just like before with Incoices table, but not with Customers, the "Invoice Table" is referencing two tables, "Tracks" and "Customers".



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 than 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".



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"
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
```

(continues on next page)

(continued from previous page)

```
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

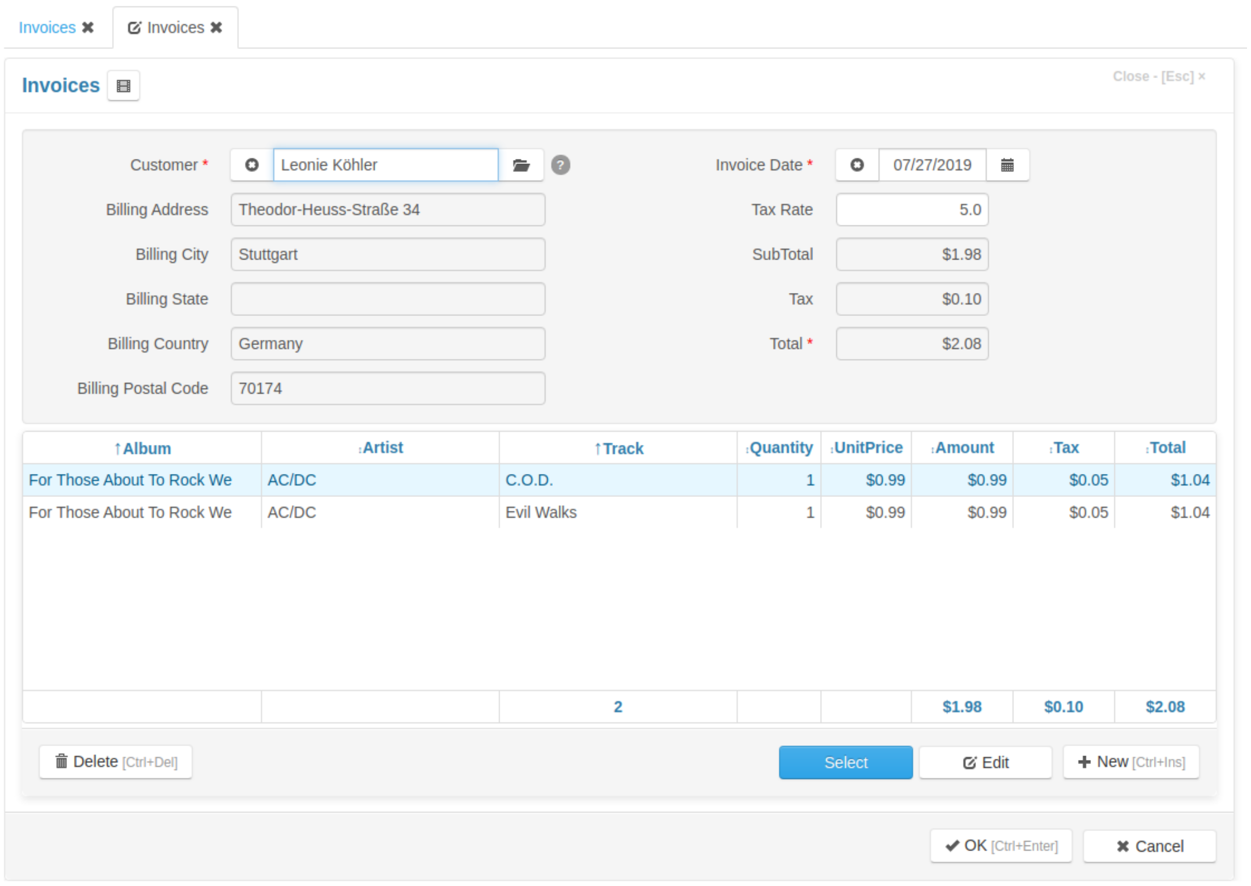
How did we go?

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.

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".

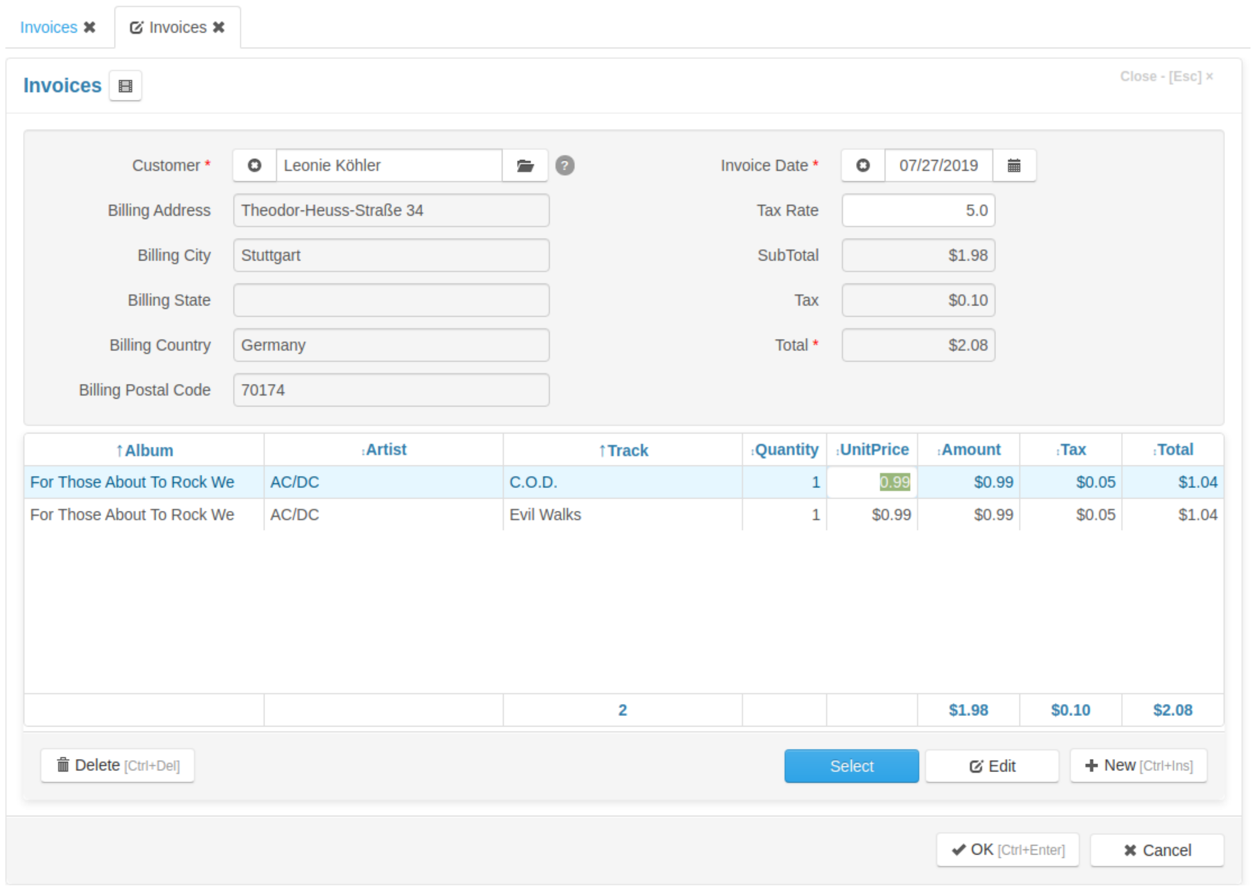
See how almost everything related to Customer is greyed out? This is because of the *Master field*! Only one field on the Invoices table is not using "Master field" and this is a Customer field. Hence, only that field won't be greyed out, because we are using it to define all other fields on the Form.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

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:



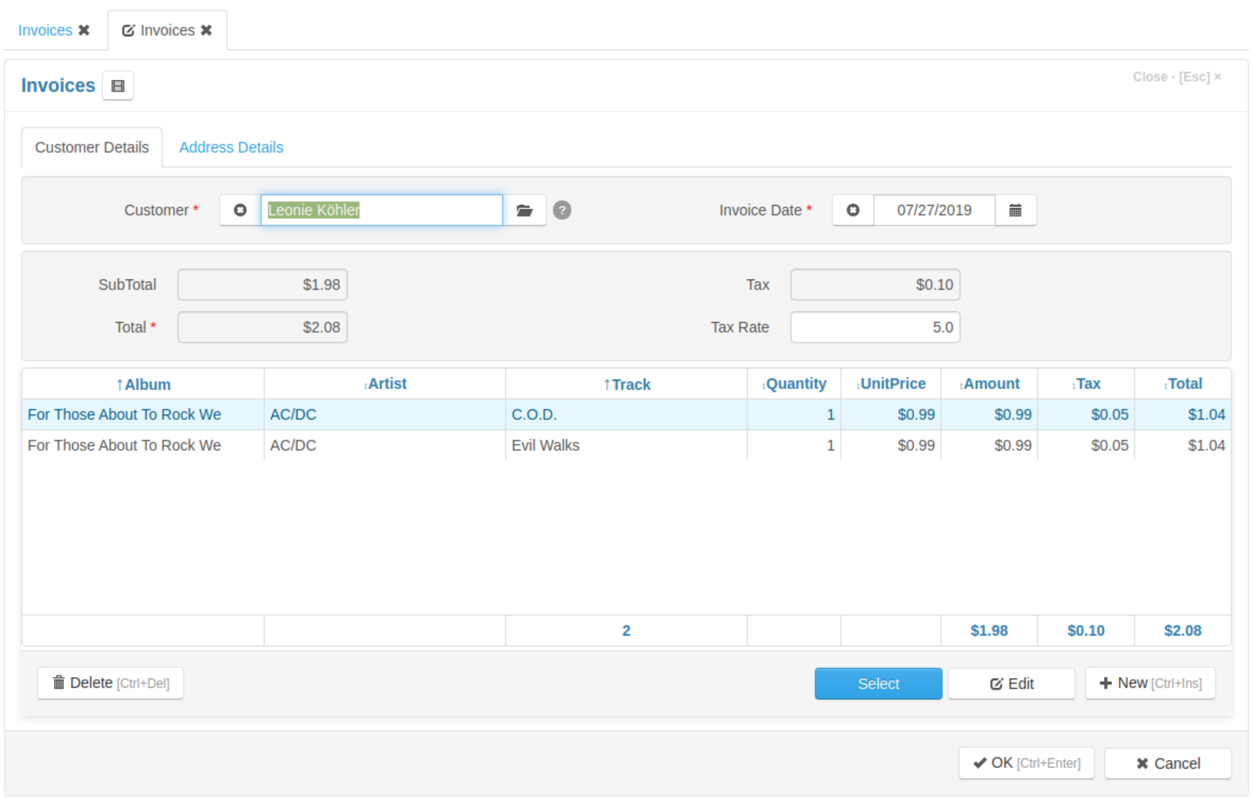
Invoices

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

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.

5.1.4 A little code

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 than 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 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.

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 version 1.5.30 which 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) {
    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();
```

(continues on next page)

(continued from previous page)

```
            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 formating. 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** from the above:

```
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();
        try {                                        <- 4.
            item.invoice_table.each(function(t) {
                t.edit();
                t.calc(t);
                t.post();
            });
        }
        finally {                                    <- 5.
            item.invoice_table.rec_no = rec;
            item.invoice_table.enable_controls();
        }
    }                                                <- 6.
}
```

Understanding the **on_field_changed** is hugely 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, let's say because it is important (no better explanation for now), and we are hoping to change the relevant *Lookup fields*. Which is a lookup to a different table, right? We are changing "Tax Rate" in Invoices table, but at the same time expect the changes in "Invoices Items".

2. In plain, down to Earth language, 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 customers.js@Demo, etc. . .

Consider this example:

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

That is not going to work. Simple 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 needed changing, after the “taxrate” changes. With **“…disable_controls”** we disable all buttons, etc. temporary, as we do not want something actioned on while working on changing records.

4. Then we “try” to update all records with a nice function **“.each(…)”**. Which does “edit”, “calc” and finally “post”, everything back via API. This is the “POST” part, correct? Without it, the data would not be saved.

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

6. If the field name was not the one we are after, exit the “if” clause here.

So, how was it?!

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

INDEX