University of Groningen

Software Engineering: Invoice Engine

Architecture Document

Version 2.0

Authors:
Awin Gray *(s3073521)*
Xiaoyu Guan *(s3542807)*
Daniël Knap *(s4117743)*
Jesse Luchtenveld *(s3221245)*
Yao Xiao *(s3435679)*

Client:
A. Van der Deijl

Teaching Assistant:
I. Botez

June 13, 2021

1 Introduction

As periodic bookings are very prevalent nowadays, there ought to be a system which can efficiently track such bookings, and generate revenue from them by creating and sending out invoices. Such a system was developed by Arie van der Deijl, however this so-called Invoice Engine has an important shortcoming.

Namely, the application is written in an outdated programming language which makes it difficult and/or costly to maintain. It also makes it difficult to expand upon the application, should the need ever arise. The first goal of this project is therefore to address this problem by rewriting the application in a modern programming language.

In addition, the system as it stands is a regular application, which makes it harder to generate revenue from the Invoice Engine itself. A desired feature therefore is for the system to work in the cloud so it can be made to adhere to the software-as-a-service (SaaS) model. This way, organizations with a paid subscription can use the invoicing system to administer periodic bookings of one or more companies. Thus, incorporating the invoicing application in a web service is the second goal of this project.

To provide as much utility as possible, an organization should have a lot of freedom in defining the kind of periodic bookings offered by them or by the companies they represent. The main function, however, will be to periodically invoice these bookings. This invoicing process consists of writing an invoice for every contract registered with a certain company. Furthermore, every invoice will be linked to collections that specify the person(s) who should pay for the invoice, and general ledger posts that are used to track the financial transactions of the company.

This document provides an overview of the technology that is used to build, run, and test the project. It also describes the different components of the system and how they interact with each other. On occasion, this document will refer to the accompanying *Requirements Document* by mentioning 'see req. X.99'. Here, 'req.' is a requirement with ID 'X.99'.

2 Technology Stack

In this project, the cloud application is built in Django, with a PostgreSQL database. During development, Docker is used to containerize the web application and the database to emulate the application working on a server.

Django was chosen because it ensures a high development speed, as there are many things that Django can do out-of-the-box. These include user authentication (see req. C.2) and providing templates for modifying database entries. The Django admin page is also very useful to handle the user role of Administrator (see reqs. C.0, C.3). Moreover, direct compatibility with Python makes it possible to rewrite the invoicing application in Python, and to fully integrate it in the web application.

In addition, Django provides effective protection against most common threats that a typical application is exposed to. This is important because the invoicing application will work with sensitive data pertaining to both people and companies. In particular, using templates prevents cross-site scripting.

Lastly, Django provides an object-relational mapper (ORM) as a layer between the database and the developer, which makes handling data from the PostgreSQL database both faster and easier. This database system was chosen because of its good integration with Django, and because there is a lot of documentation on Django projects with a PostgreSQL database. Moreover, it has many features and offers good scalability.

As mentioned above, Docker is used to containerize the application. This offers two great advantages, namely that it is assured that the application will run exactly as tested since it can deploy anywhere where Docker is running; and that it is easier and more storage-efficient than a virtual machine because it does not need a full operating system. Over-all, using Docker makes for a more efficient and reliable development environment.

2.1 DevOps

In terms of development workflow, gitlab is used for version control, along with a continuous integration/continuous delivery (CI/CD) pipeline. During the project, we made use of the following basic repository structure:

1. master – the final branch used to submit deliverables. This branch has limited permissions and reviewers are needed to accept pull requests.
2. development – the working branch used as the secondary master branch. This is where all features under development are contained.

3 Architectural Overview

As mentioned in the previous section, the web application is built in Django, which follows the model-view-template principle: database tables are defined as classes in the model, the view handles input/output and interaction with the user, and the templates are seen by the user in their browser.

On top of this, it is also possible to use a front-end framework, such as React JS. Moreover, a REST API built in the Django REST Framework can be used to handle information transfer between the user and the server. It has however been decided that this is beyond the scope of this project.

In this section, an overview of architectural decisions is provided. This overview is structured as follows. First, the data model is described. Afterwards, Create, Read, Update and Delete (CRUD) operations and endpoints are discussed, as well as the user flow, the invoicing process, and options for the front-end of the application.

3.1 Data modeling

In this application, data is the key concept: what to store and how to store it. This section describes the database strategy that is used, and the design of the database itself. This section goes into the implementation of reqs. C.4-7, C.9-10, I.1, I.3-4, and to a lesser extent C.1.

3.1.1 Database tenancy pattern

One of the project requirements is that there should be a single database for all data of all organizations using the application (see req. C.1). This database therefore needs to be multi-tenant: able to serve multiple customers (tenants) with one instance of the application. There are two distinct strategies to achieve this: a shared database with a shared schema; and a shared database with an isolated schema.

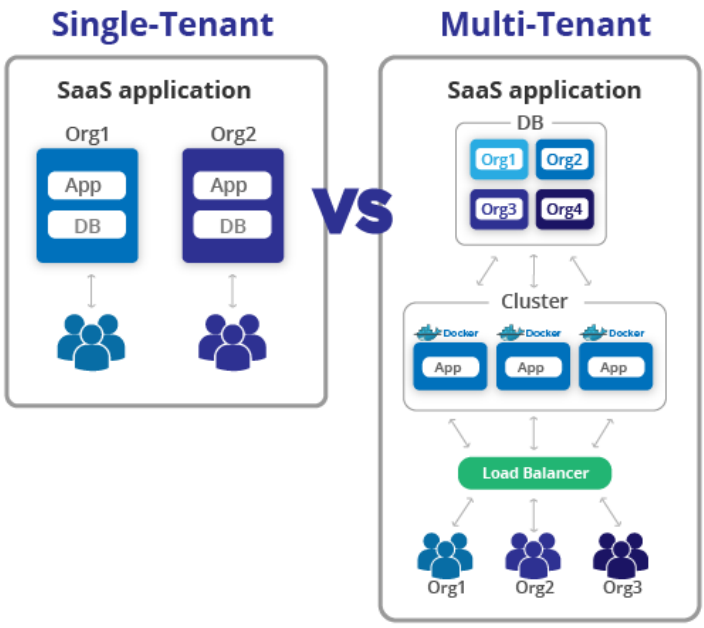


Figure 1: The difference between a single-tenant and multi-tenant application.

In a shared database with shared schema, there is a tenancy table with entries linked to users, or tenants. Entries in other database tables then link (directly or indirectly) to an entry in the tenancy table, which implies they belong to that tenancy and are to be accessed only by the user connected to it.

In contrast, in a shared database with isolated schema, every tenancy has their own set of tables which is only accessible by users connected to said tenancy. These schemas still exist in the same database.

Both approaches have their advantages and disadvantages. For instance, the shared schema is very scalable compared to the isolated schema. A disadvantage of the shared schema on the other hand is that there is weaker separation of tenants' data as access cannot be limited on a database level, which implies that it is vital to assure that the proper tenancy is provided as a filter when performing any database operation.

It was chosen to implement a shared database with a shared schema, because of the advantages in scalability (see req. NF.2). Section 3.2.3 will go into the methods used to assure that the proper data is selected when performing database operations.

3.1.2 Entity relationship diagram

An entity relationship diagram (ERD) was created and can be seen on its own page at the end of the document. Both the full model and the model split up into two parts are included for increased clarity. This section will provide an explanation of the data model.

To start with, there are two distinct sections to the ERD. The part above the purple line represents the tables which are filled before running the invoicing process (initial data); and the part below the purple line represents the tables which are filled during the invoicing process (generated data). This layout was chosen to increase the clarity for the reader with respect to what happens when bookings (or contracts) are invoiced.

The initial data consists of seven tables: Tenancy, ContractType, BaseComponent, VATRate, Contract, Component, and ContractPerson. Following is a brief description of each table. Trivial fields such as 'name' are omitted. In order to prevent repetition, some common fields include general_ledger_account and general_ledger_dimension. These are used to specify in which account certain financial transactions need to be booked once invoices have been created. general_ledger_debit and general_ledger_credit specify on which side of the balance these transactions should be posted.

Tenancy	This table is used to keep track of who can access what data, as all entries in other tables have a(n) (in)direct reference to an entry in the tenancy table. A user's username is their tenancy_id, which can also be found in the tenancy table. This makes it possible for the user to access only the tenancies that are linked to their username. This table also keeps track of the number of contracts belonging to this tenancy that are in the database, and of the next date to invoice the contracts.
ContractType	A contract defines a periodic booking, and always belongs to a contract type. This contract type specifies the kind of booking, for instance 'rental car' or 'rental van' for a company that rents out vehicles. This is used to group similar contracts together and thereby to prevent duplicate data.
Contract	A contract contains important information about a periodic booking, such as the invoicing period (month, quarter, etc.), and start and end dates (including the end date of invoicing). When the invoicing period is custom, the number of days for which the contract is valid needs to be provided, otherwise this field can remain empty. It always has a contract type. The contract table also contains amounts, which are the sums of the amounts from associated components (see below).
BaseComponent	A contract consists of components, which can be viewed as different lines on the contract specifying amounts for sub-categories. For instance, in keeping with the vehicle rental company example used earlier, there could be one contract of the type 'rental car' with three components of the types: 'car, up to 200 km', 'final cleaning', and 'insurance'. These types are represented by base components. This table therefore has a similar function to the contract type; it represents the type of component.
VATRate	The vast majority of countries employ a value added tax on products and services. There are many different rates used in different countries around the world, which is why the VAT rates can be specified by the user.

Component	A component contains important information about sub-categories of periodic bookings. It is always associated with a base component, a VAT rate, and a contract. All components associated with a single contract define the final price of that contract.
ContractPerson	Finally, a contract has associated contract persons. These are the people who pay for the contract. Every contract can have multiple people associated with it, for example, the parents of a student might pay half of the student's monthly rent. In such a case, two or three people can be linked to the contract. Personal details such as name and address are registered, as well as their preferred method of payment, their international bank account number (IBAN), their email address and phone number, and their day of payment (could be the first of the month, could be the fifth, etc.). Lastly, the percentage of the total amount is registered. In the example, this would be 50 for the student, 25 for their mother, and 25 for their father.

The generated data consists of four tables that have been implemented, and one table which has not been implemented (indicated in blue). It is important to note that these tables are read-only to the user. Entries will be generated by running the invoicing process (see Section 3.4). Similarly to the initial data, following is a brief description of each table.

Invoice	Recall that a contract represents a periodic booking. This means that the contract person has to pay for their booking every period. A new invoice is created every time the company is to receive payment for a contract. The rate of invoicing depends on the defined period in the contract, for instance every month. The invoice contains information about the total amount due, the invoicing date, and the expiration date of the invoice.
InvoiceLine	Similarly to a contract consisting of components, an invoice consists of invoice lines. Every time a contract is invoiced, one invoice line is created for every component. The invoice line is linked to the invoice created from the contract that the component belongs to. It contains information about the amounts that are due, and about how to book these amounts on the general ledger.
Collection	A collection represents what a contract person needs to pay, and how they will do this. It is linked to a contract person and an invoice, and the amount of all collections connected to one invoice add up to the total amount as specified in the invoice.
GeneralLedgerPost	The general ledger is used to keep track of all financial transactions of a company. Since invoices and invoice lines represent (future) financial transactions, they need to be represented in the ledger as well. This is why a general ledger post is created for every invoice and for every invoice line. The general ledger post groups all accounts, dimensions, and debit and credit amounts together so it can be properly registered in the actual ledger.
Cumulative	A cumulative represents the total amount that was paid for one contract, over a certain time period. It had low priority in this project, as is it mainly applicable to the housing market and the application is intended to be used more broadly. As such, it has not been implemented.

3.2 CRUD operations & endpoints

In this section, the structure of the web application in terms of URLs will be laid out. There is a separate URL for each type of CRUD operation: create, read (list), update, and delete.

3.2.1 Registration

The home page has a link to a log-in page (/login/), where a user can log in with their id and password. Afterwards, they are redirected to a profile page (/profile/) where they can choose to change their details (password, email address). From here, they can also access their tenancies, i.e. the companies for which they handle periodic bookings.

It is only possible to add a new User via the admin site (/admin/), because a username is chosen for the User by the Administrator. This username must be a positive integer, as that is the data type of the tenancy id in the tenancy table.

3.2.2 Database tables

Because this is a very data-driven application, all endpoints will have some link to database tables. In the following URLs, the words in italics represent numbers. Starting with tenancy:

- /profile/tenancies/ – shows all available tenancies of this User. The User can also update an existing one from this page. For each available tenancy, the user can go to the list of associated objects (contract types, base components, VAT rates, contracts, invoices). They can also start the invoicing process for a tenancy, which will generate invoices and invoice lines based on the contracts and contract lines associated with that tenancy (see Section 3.4).
- /profile/tenancies/*company_id*/ – to view details of the tenancy identified by *company_id*.
- /profile/tenancies/*company_id*/update/ (or .../delete/) – contain pages to update or to delete the tenancy identified by *company_id*.

The other URLs are very much in keeping with the same theme, although details pages only exist for contracts and invoices. Other models show their details on their respective list pages. Only VAT rate and component will be shown here as additional examples. Vat rate:

- /profile/tenancies/*company_id*/vat_rates/
- /profile/tenancies/*company_id*/vat_rates/create/
- /profile/tenancies/*company_id*/vat_rates/*vat_rate_id*/update/ (or .../delete/)

For component, there is no list view. All components are shown on the details page for their respective contract (first URL below). Of course they can be created, updated, and deleted:

- /profile/tenancies/*company_id*/contracts/*contract_id*
- /profile/tenancies/*company_id*/contracts/*contract_id*/components/create
- /profile/tenancies/*company_id*/contracts/*contract_id*/components/*component_id*/update/
- /profile/tenancies/*company_id*/contracts/*contract_id*/components/*component_id*/delete/

For everything underneath the purple line in the data model (see Section 3.1.2 and the ERD at the end of the document), there will be no update or delete pages. These tables are read-only. Furthermore, the invoice lines will work similarly to the components, in that there will not be a separate list view for them. They will be grouped with their respective invoices.

3.2.3 Ensuring proper data access

Because of the 'shared database, shared schema' database tenancy pattern (see Section 3.1.1), it is of vital importance to make sure that every new entry gets linked to the proper tenancy, and that the user can only request database entries corresponding to the selected tenancy.

This is why the URL matters. Once a tenancy is selected, the URL contains the primary key of that tenancy. This means that whenever a new object is created, the company_id can be retrieved from the URL and the new entry can be linked with the tenancy corresponding to that primary key.

When a user attempts to retrieve data from the database, it is checked that their username matches the tenancy_id as stored in the associated tenancy object. If this is not the case, the user is not be allowed to retrieve the data. This satisfies req. C.1.

3.2.4 Database constraints

Utilizing database constraints is an important way to guarantee information integrity. An example is ensuring that the start and end date of a VAT rate are in the correct order. In order to prevent such problems, a constraint can be imposed on the two dates. It is checked whether the start date is earlier than the end date, because it would not make sense otherwise. There are two approaches to achieve this goal.

The first one is on a database level. This is implemented by adding check conditions in the model. The problem with this approach is that only the developer can see the error from the database, and the user will not be able to see the issue when they are giving the input although this is nice for the developer to debug in the future.

The second approach is on a form level. Conditions are checked in the form class right before saving the form. The error message can then be passed on to the user. This method also has a drawback in that only the user will get the message that their input is insufficient, which does not help the developer to make the form more intuitive in the future.

Overall, for storing the users' input, the second approach is sufficient.

3.3 User flow

This section will describe various actions a user can perform on the website. Most of the actions are rather similar due to the data-driven nature of the application. This section also ties in closely with Section 3.2.2: here the paths to reach the different URLs are described.

A subscriber to the invoicing service (a User) can log on to the website with their tenancy id and password. They will then be directed to a profile page, where they can change, for instance, their password or contact details. From here, the user can also log off. From this page, they also have access to a list of their tenancies.

On the tenancy list page, each tenancy links to pages containing lists of contract types, base components, VAT rates, contracts, and invoices. There are also buttons to invoice the contracts in the database, and to export the contents of the generated data tables. Moreover, the user can choose to view, update or delete a tenancy; or to create a new one.

When the User visits a list page, they can create new instances of the object depicted in this list. They can also choose to update or delete existing objects.

When creating or updating an object, the User is redirected to a page with a form to fill in all accessible fields. If the user has made a mistake, for instance if they inserted a wrong data type or they violated a constraint, they will get an error message.

When deleting an object, the User is prompted whether they are sure that this object should be deleted. If they choose yes, they will be returned to the list page.

Finally, every page also contains a link to return to the page that linked to the current one. Crucially, this is not simply a 'back-button' as found in a web-browser, because such a button would return Users to forms as well, which is not desired.

3.4 Invoicing process

This section will describe the invoicing process, during which invoices are created based on the contracts that are in the database already. This section describes the implementation of req. C.8. Reqs. I.2 & U.1 are implemented in much the same way. When a User triggers the invoicing process for a certain tenancy, the following will occur:

- The method `tenancy.invoice_contracts()` is called.
- All components linking to tenancy with an invoicing date before or on today are loaded into memory, ordered by contract.
- The same applies to all active contract persons (those that do not have an end date).
- Loop over the components:
  - When a new contract is reached:
    - call `contract.invoice()` to create a new Invoice object and to update the contract.
  - Call `component.invoice()` to create invoice lines and general ledger posts.
  - When a new contract is reached:
    - Finalize the previous invoice by calling `invoice.create_gl_posts()`.
    - Loop over the contract persons for this contract:
      - Call `contract_person.invoice()` to create Collection objects.
- Use a single database transaction to add all new Invoice, InvoiceLine, GeneralLedgerPost, and Collection objects to the database.

It is important that this process is as fast as possible, which is why components are loaded into memory beforehand. This prevents many slow database hits. It was also found that using the optimized creation functions helps speed up the process considerably.

To test this, a number of functions to assess the speed of the invoicing process were written, most notably a function that automatically populates the database with a large number of contracts with randomized dummy data; and a function which times the invoicing process. In addition to this, utility functions to fully clear the database have also been written. These functions are of course completely separate from the application and can only be called from the development environment.

3.5 Front-end

For the front-end of the project, Django's built-in class-based views, which are rendered as templates, are employed. Bootstrap 5 is used as the main framework for styling. The reason for this is that Bootstrap is widely used, easy to implement and free. More importantly, it was assessed that the front-end component of the project would not require a lot of state managements and client-side processing. This is ultimately due to the project being developed as a SaaS that is data-intensive.

4 Team Organization

Because the components of this project were very interconnected, we thought it better to not have sub-teams within our project group. Instead, it was decided that every group member ought to know about every aspect of the program.

We therefore used Trello to keep track of all project tasks. This way everyone has a clear overview of what needs to be done, and each person could choose any task to complete. Communication within the team was handled over Slack or WhatsApp, and there were twice-weekly meetings via Google Meet. In addition to this, a client meeting was scheduled every two weeks.

5 Future Ideas

This section will describe some ideas to extend and improve the application beyond the scope of this project.

The most obvious extensions of the application would of course be to implement the will-not-haves from the *Requirements Document*. This would then result in the ability to book a void contract, and the ability to calculate cumulatives of contracts. The addition of these things would make the application more suitable for the housing market.

Aside from this, a welcome extension would be to implement a REST API in the Django REST Framework to serve as an abstraction layer between the server and the client. This would result in greatly increased flexibility in future development of front-end or back-end, as they would no longer depend on each other to a certain extent.

Moreover, a REST API would make it easier to use a dedicated front-end framework to make the application look better and to make it more user-friendly.

Some other details include firstly a rework of the contract person interface. When one wants to add two persons to a contract, they have to be added one at a time. It would be better to have the possibility to add all persons in one operation. Secondly, the possibility to export more tables to CSV-files would be beneficial, in case a customer would like to stop using the service without losing their data.

The last idea for future improvement is to either use a job-scheduling service such as Celery to manage invoicing runs triggered by different Users, or to have a daily scheduled invoicing run of all contracts of all Users. The latter is thought to be better, because it ensures that no subscriber to the service will ever miss an invoicing date.

6 Testing & maintenance

To verify the functionality of the application, we have written tests for a variety of situations. Most notably, we have tests for all create, update, and delete operations (read is tested implicitly with those operations). Moreover, we have a number of logic tests for the standard invoicing cycle in a variety of scenarios (standard, using units, without VAT, starting during the invoicing period, ending during the invoicing period), and for creating corrections when an end date is changed, for instance.

For page access, there is no automated test. Nevertheless, this is a very important aspect of the application, so this was extensively tested manually. The procedure here was to create two users, to register objects to user #1, and to then manually visit the URLs belonging to user #1 while being logged on as user #2. We were unable to access any restricted data, which shows that the system works as intended.

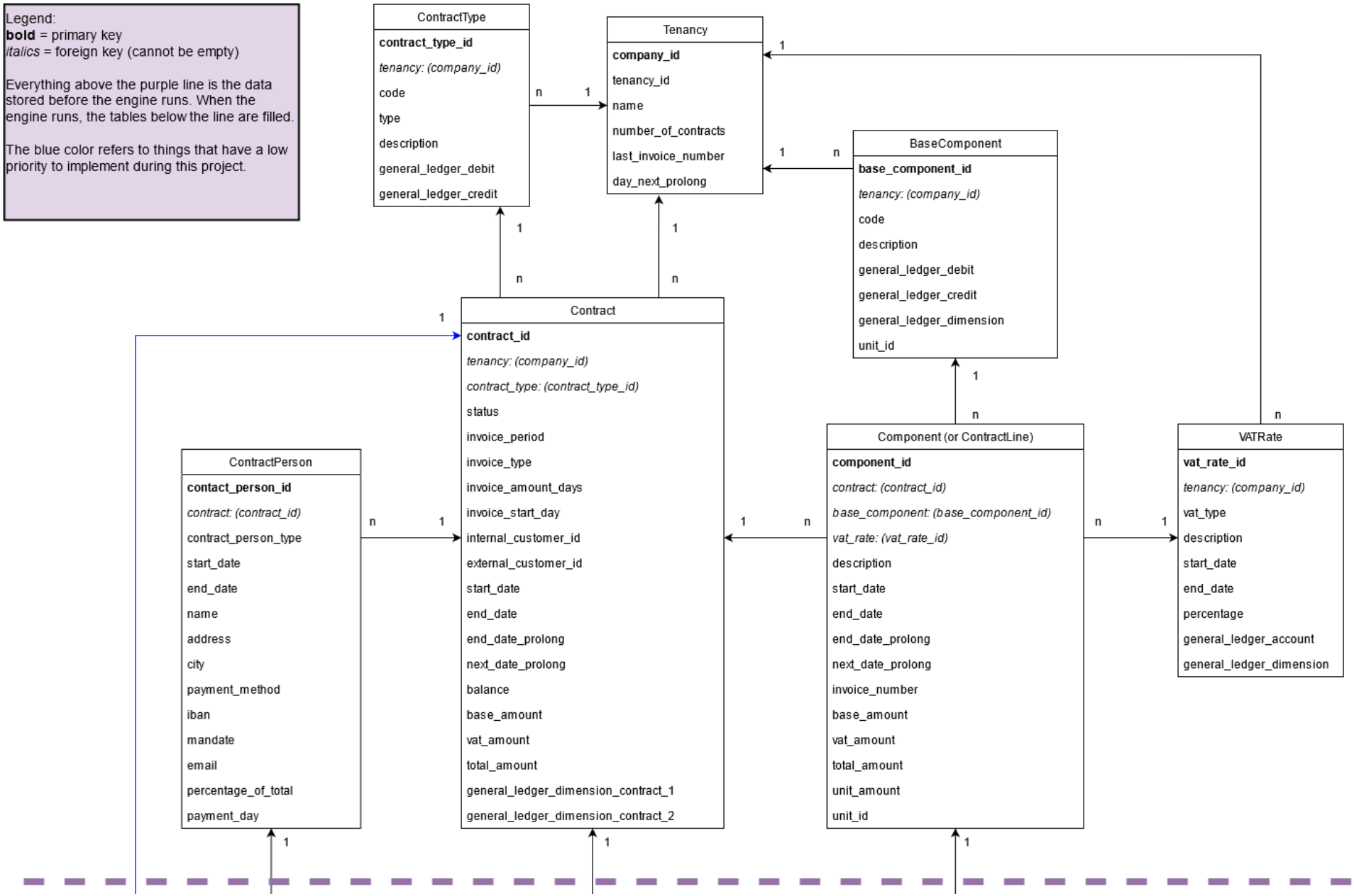
This works so well, because restrictions to object access are written in one place and inherited by other objects. This makes maintenance with respect to this issue very easy as well. In fact, maintenance of the view/controller section of the application is very easy in general, because it was chosen to make the views and forms as 'dumb' as possible. Nearly all business logic is located in the models, which does imply that the models file is very large in comparison.

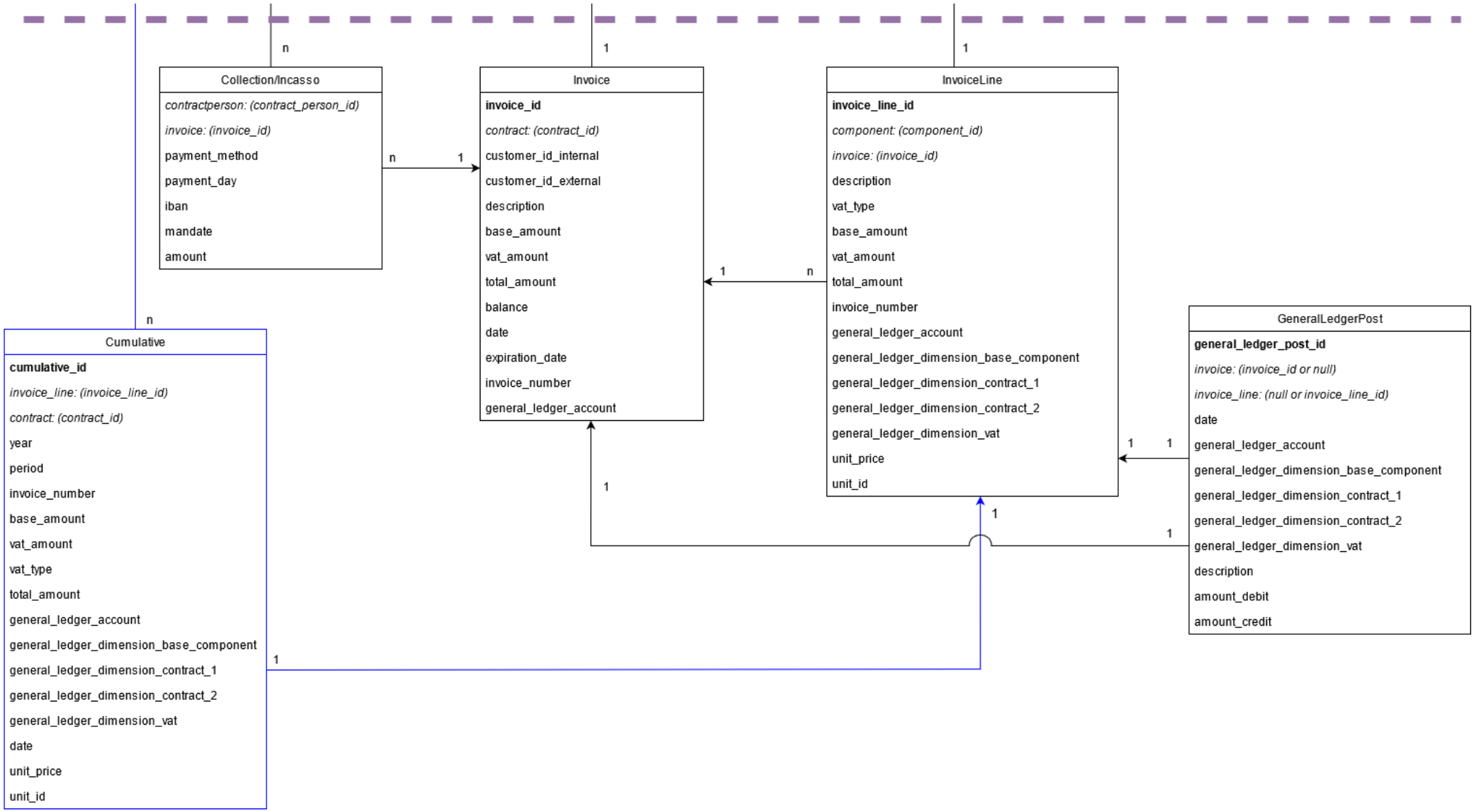
Such large files are never good from the standpoint of maintenance, however this is how Django applications are structured. The models are also rather interconnected and some methods are quite long. Given more time, this would be something to improve upon, because it does negatively impact maintainability as well.

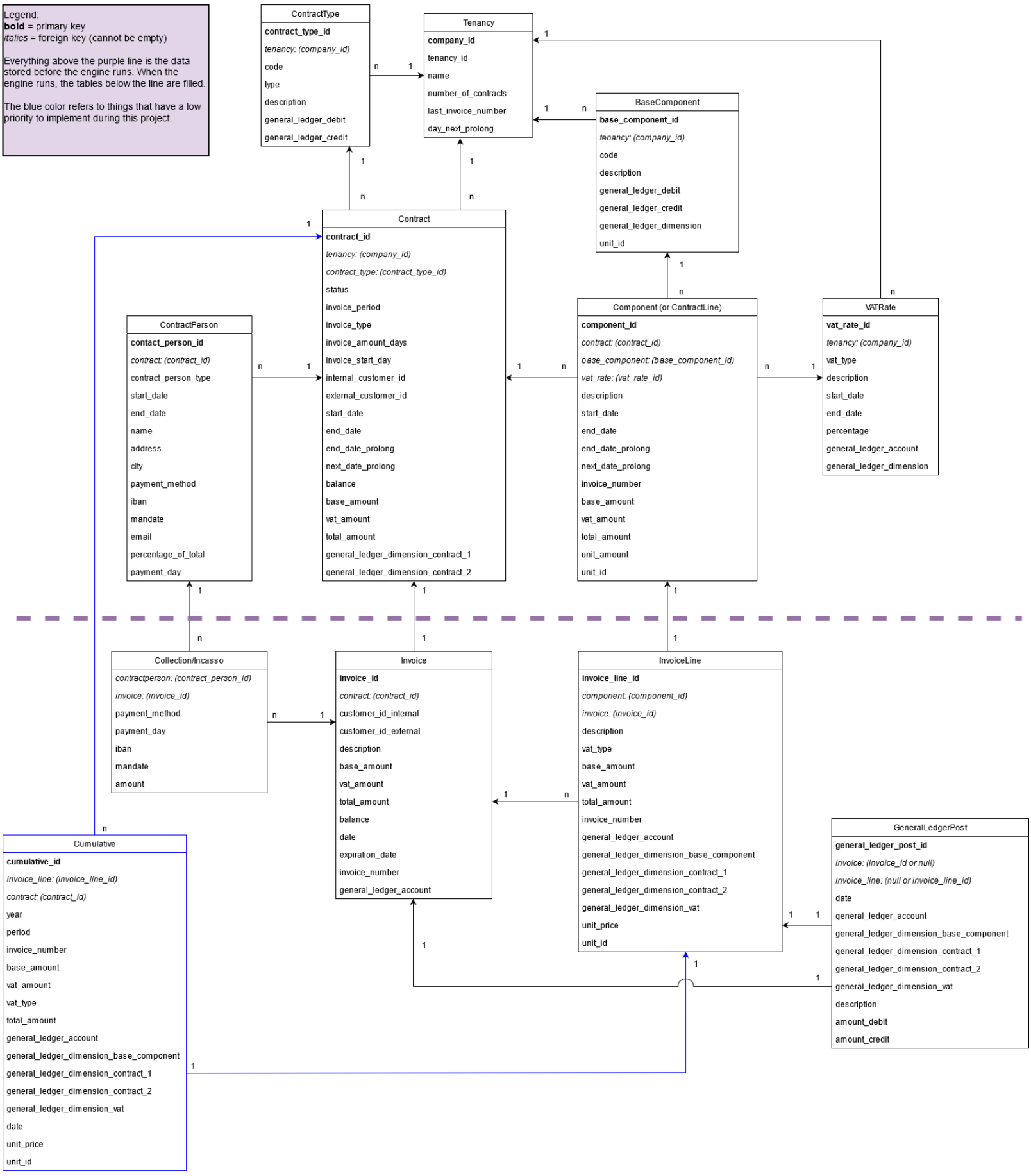
7 Document Changelog

In this section, the changelog for this document is given.

Date	**Author**	**Description**
Feb. 23, 2021	J. Luchtenveld	Created the document structure, and made a title page.
Mar. 21, 2021	A. Gray	Added architectural descriptions and justifications for some design choices. (Iteration I)
Mar. 23, 2021	J. Luchtenveld	Added the ERD, and wrote a description of the 'initial data' side of the ERD.
Mar. 28, 2021	J. Luchtenveld	Updated the ERD, and wrote a description of the 'generated data' side of the ERD. Added small 'setups' for others to use in different sections.
Mar. 28, 2021	A. Gray	Added front-end descriptions and DevOps specifications.
Mar. 29, 2021	A. Gray	Added endpoints, cruds and updated previous sections.
Mar. 29, 2021	J. Luchtenveld	Finished introduction, section 2, section 3.1. Worked on section 3.2.
Mar. 30, 2021	X.Guan	Worked on database constraints (section 3.2.4).
Mar. 30, 2021	J. Luchtenveld	Finished User flow and made a final pass.
May 27, 2021	J. Luchtenveld	Minor changes to incorporate teacher feedback.
May 31, 2021	J. Luchtenveld	Added future ideas, included more references to requirements, updated invoicing process, and over-all finished the document.
Jun. 13, 2021	J. Luchtenveld	Added a few paragraphs on testing & maintainability.


Legend:
bold = primary key
italics = foreign key (cannot be empty)
Everything above the purple line is the data stored before the engine runs. When the engine runs, the tables below the line are filled.
The blue color refers to things that have a low priority to implement during this project.
ContractType
contract_type_id
tenancy: (company_id)
code
type
description
general_ledger_debit
general_ledger_credit
Tenancy
company_id
tenancy_id
name
number_of_contracts
last_invoice_number
day_next_prolong
BaseComponent
base_component_id
tenancy: (company_id)
code
description
general_ledger_debit
general_ledger_credit
general_ledger_dimension
unit_id
Contract
contract_id
tenancy: (company_id)
contract_type: (contract_type_id)
status
invoice_period
invoice_type
invoice_amount_days
invoice_start_day
internal_customer_id
external_customer_id
start_date
end_date
end_date_prolong
next_date_prolong
balance
base_amount
vat_amount
total_amount
general_ledger_dimension_contract_1
general_ledger_dimension_contract_2
ContractPerson
contact_person_id
contract: (contract_id)
contract_person_type
start_date
end_date
name
address
city
payment_method
iban
mandate
email
percentage_of_total
payment_day
Component (or ContractLine)
component_id
contract: (contract_id)
base_component: (base_component_id)
vat_rate: (vat_rate_id)
description
start_date
end_date
end_date_prolong
next_date_prolong
invoice_number
base_amount
vat_amount
total_amount
unit_amount
unit_id
VATRate
vat_rate_id
tenancy: (company_id)
vat_type
description
start_date
end_date
percentage
general_ledger_account
general_ledger_dimension


Collection/Incasso
contractperson: (contract_person_id)
invoice: (invoice_id)
payment_method
payment_day
iban
mandate
amount
Invoice
invoice_id
contract: (contract_id)
customer_id_internal
customer_id_external
description
base_amount
vat_amount
total_amount
balance
date
expiration_date
invoice_number
general_ledger_account
InvoiceLine
invoice_line_id
component: (component_id)
invoice: (invoice_id)
description
vat_type
base_amount
vat_amount
total_amount
invoice_number
general_ledger_account
general_ledger_dimension_base_component
general_ledger_dimension_contract_1
general_ledger_dimension_contract_2
general_ledger_dimension_vat
unit_price
unit_id
GeneralLedgerPost
general_ledger_post_id
invoice: (invoice_id or null)
invoice_line: (null or invoice_line_id)
date
general_ledger_account
general_ledger_dimension_base_component
general_ledger_dimension_contract_1
general_ledger_dimension_contract_2
general_ledger_dimension_vat
description
amount_debit
amount_credit
Cumulative
cumulative_id
invoice_line: (invoice_line_id)
contract: (contract_id)
year
period
invoice_number
base_amount
vat_amount
vat_type
total_amount
general_ledger_account
general_ledger_dimension_base_component
general_ledger_dimension_contract_1
general_ledger_dimension_contract_2
general_ledger_dimension_vat
date
unit_price
unit_id
n
1
n
1
n
1
1
n
1
1
1
1
1
1
n
1


Legend:
bold = primary key
italics = foreign key (cannot be empty)
Everything above the purple line is the data stored before the engine runs. When the engine runs, the tables below the line are filled.
The blue color refers to things that have a low priority to implement during this project.
ContractType
contract_type_id
tenancy: (company_id)
code
type
description
general_ledger_debit
general_ledger_credit
Tenancy
company_id
tenancy_id
name
number_of_contracts
last_invoice_number
day_next_prolong
BaseComponent
base_component_id
tenancy: (company_id)
code
description
general_ledger_debit
general_ledger_credit
general_ledger_dimension
unit_id
Contract
contract_id
tenancy: (company_id)
contract_type: (contract_type_id)
status
invoice_period
invoice_type
invoice_amount_days
invoice_start_day
internal_customer_id
external_customer_id
start_date
end_date
end_date_prolong
next_date_prolong
balance
base_amount
vat_amount
total_amount
general_ledger_dimension_contract_1
general_ledger_dimension_contract_2
Component (or ContractLine)
component_id
contract: (contract_id)
base_component: (base_component_id)
vat_rate: (vat_rate_id)
description
start_date
end_date
end_date_prolong
next_date_prolong
invoice_number
base_amount
vat_amount
total_amount
unit_amount
unit_id
VATRate
vat_rate_id
tenancy: (company_id)
vat_type
description
start_date
end_date
percentage
general_ledger_account
general_ledger_dimension
ContractPerson
contact_person_id
contract: (contract_id)
contract_person_type
start_date
end_date
name
address
city
payment_method
iban
mandate
email
percentage_of_total
payment_day
Collection/Incasso
contractperson: (contract_person_id)
invoice: (invoice_id)
payment_method
payment_day
iban
mandate
amount
Invoice
invoice_id
contract: (contract_id)
customer_id_internal
customer_id_external
description
base_amount
vat_amount
total_amount
balance
date
expiration_date
invoice_number
general_ledger_account
InvoiceLine
invoice_line_id
component: (component_id)
invoice: (invoice_id)
description
vat_type
base_amount
vat_amount
total_amount
invoice_number
general_ledger_account
general_ledger_dimension_base_component
general_ledger_dimension_contract_1
general_ledger_dimension_contract_2
general_ledger_dimension_vat
unit_price
unit_id
GeneralLedgerPost
general_ledger_post_id
invoice: (invoice_id or null)
invoice_line: (null or invoice_line_id)
date
general_ledger_account
general_ledger_dimension_base_component
general_ledger_dimension_contract_1
general_ledger_dimension_contract_2
general_ledger_dimension_vat
description
amount_debit
amount_credit
Cumulative
cumulative_id
invoice_line: (invoice_line_id)
contract: (contract_id)
year
period
invoice_number
base_amount
vat_amount
vat_type
total_amount
general_ledger_account
general_ledger_dimension_base_component
general_ledger_dimension_contract_1
general_ledger_dimension_contract_2
general_ledger_dimension_vat
date
unit_price
unit_id