FACULTY
OF INFORMATION
TECHNOLOGY
CTU IN PRAGUE

ASSIGNMENT OF BACHELOR'S THESIS

Title:	ElateMe: Backend II.
Student:	Lukáš Hrachovina
Supervisor:	Ing. Petr Pauš, Ph.D.
Study Programme:	Informatics
Study Branch:	Web and Software Engineering
Department:	Department of Software Engineering
Validity:	Until the end of summer semester 2018/19

Instructions

The goal of this Bachelor thesis is to expand the ElateMe API service so that it can be used in a production environment. This thesis is an iteration upon the ElateMe - backend thesis by Bc. Yevhen Kuzmovych.
Analyze (Use UML diagrams, where appropriate):
- New endpoints and enhancements required, based on the expansion of the application's client side,
- API security standards,
- Payment Request API,
- User administration mechanism.

Design:
- API versioning mechanism,
- Credit payment,
- Product suggestions,
- User administration mechanism,
- Monitoring and alerting system.

Implement:
- Push notifications for mobile devices,
- API service expansion,
- Credit payment,
- User and users' wishes management,
- User administration mechanism,
- Monitoring and alerting system.

Test:
- Cover the codebase with tests.

References

Will be provided by the supervisor.

Ing. Michal Valenta, Ph.D.
Head of Department

doc. RNDr. Ing. Marcel Jiřina, Ph.D.
Dean

Prague February 2, 2018

FACULTY
OF INFORMATION
TECHNOLOGY
CTU IN PRAGUE

Bachelor's thesis

ElateMe: Backend II.

Lukáš Hrachovina

Department of Software Engineering
Supervisor: Ing. Petr Pauš Ph.D.

May 15, 2018

Acknowledgements

I would like to thank my supervisor, Ing. Petr Pauš Ph.D. for providing help and consultation in the process of writing this thesis. Also I owe big thanks to Michal Maněna for bringing ElateMe to us and providing technical supervision throughout the whole development process.

Declaration

I hereby declare that the presented thesis is my own work and that I have cited all sources of information in accordance with the Guideline for adhering to ethical principles when elaborating an academic final thesis.

I acknowledge that my thesis is subject to the rights and obligations stipulated by the Act No. 121/2000 Coll., the Copyright Act, as amended, in particular that the Czech Technical University in Prague has the right to conclude a license agreement on the utilization of this thesis as school work under the provisions of Article 60(1) of the Act.

In Prague on May 15, 2018 . . . .

Citation of this thesis

Hrachovina, Lukáš. *ElateMe: Backend II.*. Bachelor's thesis. Czech Technical University in Prague, Faculty of Information Technology, 2018.

Abstrakt

ElateMe je pracovní název nové crowdfundingové platformy, kde narozdíl od ostatních platforem, komerční nebo inovativní projekty jsou nahrazeny osobními přání uživatelů. Uživatelé mohou používat mobilní Android nebo iOS, ale také webovou verzi aplikace.

Mobilní i webové aplikace používají stejný backend, který je předmětem této práce. Konkrétně RESTové API, napsané s použitím Django REST frameworku nad Pythonem 3. Cílem je analyzovat, navrhnout a implementovat rozšíření funkcionality API. Tato práce navazuje na Bakalářskou práci Yevhena Kuzmovyče *ElateMe Backend I.*

Klíčová slova ElateMe, crowdfunding, REST API, Django, Python, PostgreSQL

Abstract

ElateMe is working name for a new crowdfunding platform, where typical commercial or innovative projects are replaced with personal wishes. The whole platform is delivered to the users in the form of Android and iOS mobile applications and a web version of the application.

Both mobile and web applications connect to the same backend, which is this thesis' focus. In particular the REST API written using Django REST framework on Python 3. The aim is to analyze, desing and implement expansion of the API's functionality. This thesis is a continuation of Yevhen Kuzmovych's Bachelor thesis *ElateMe Backend I.*

Keywords ElateMe, crowdfunding, REST API, Django, Python, PostgreSQL

Contents

List of Figures

Introduction

ElateMe

ElateMe is working name for a new crowdfunding platform, where one's potential benefactors are not strangers, but their friends. This causes the subjects of funding, simply called "Wishes", to be more personal rather than commercial projects as is often the case with other established crowdfunding platforms. Many elements of a typical social network are included, such as comments and friend groups. Users can access the platform using Android and iOS mobile applications or use a web version. As of now, there isn't any other widespread service on the market, providing this functionality.

Users can post wishes, publicly or only for their friends to see, in order to raise money and fund them. Or on the contrary a group of friends may want to fund a secret surprise present for their mutual friend, which is something ElateMe can help with.

A platform like this could not possibly work on frontend alone, since plenty of data need to be stored in database and accessed. Therefore with both mobile and web applications, a REST (Representational State Transfer) API (Application Programming Interface) is the best choice for providing communication between the frontend and a database.

Aim of the thesis

This thesis is a continuation of Yevhen Kuzmovych's Bachelor thesis *ElateMe Backend I.* It can be split into three major parts: Analysis, Design and Implementation, as is often the case with software engineering theses.

The aim is to move towards improving the API's functionality so that the product can be launched in early testing mode for potential users. In the Design and Analysis chapters these changes are documented alongside some foundation for the future expansion and full production usage.

Motivation

I have been working on the project for the past three semesters during Software project 1 and 2, starting on the frontend in ReactJS and for the second run moving on to work on the backend API.

It is a challenging project with real world application, using modern technologies and following the latest trends. Also it was a good opportunity to work on a complex backend system, which certainly is a good practice for professional life. I was given enough freedom, enabling me to come up with my own solutions.

Chapter 1

Analysis

This chapter focuses on analyzing the project current state in terms of its background and structure, new requirements and challenges related to improving the API's functionality.

1.1 Background

ElateMe is a brand new platform, developed here at FIT CTU in Prague, with Michal Maněna being the head of the project. Development started two years ago and now a second team of students is working on the platform. ElateMe has recently rebranded with a new name in place Wowee. Because this change is quite recent and in order to keep consistency, for the purpose of this thesis, the name ElateMe will be used.

The API service is specifically tailored to the needs of the frontend. The basic functionality is already in place, with plenty of endpoints covering wish creation and management, Facebook friends integration, comments etc. ElateMe API communicates over HTTPS, with OAuth2 user authentication, which is compliant with current standard [1].

Naturally many components of the application have been reworked over the past year. These changes will be pointed out further in this thesis.

1.1.1 Domain model

A domain model is an abstract model of the domain, where both data and behaviour are represented [2]. Its objects usually cannot be translated directly to programming language objects, as domain models tend to be simplified and not bound by the programming language constraints and conventions. It is more closely related to the database model, with the biggest difference that it combines data with processes [2].

The domain model for ElateMe can be seen in 1.1. Due to the sheer size of the project only the most important entities are displayed to help in understanding relations in the project.

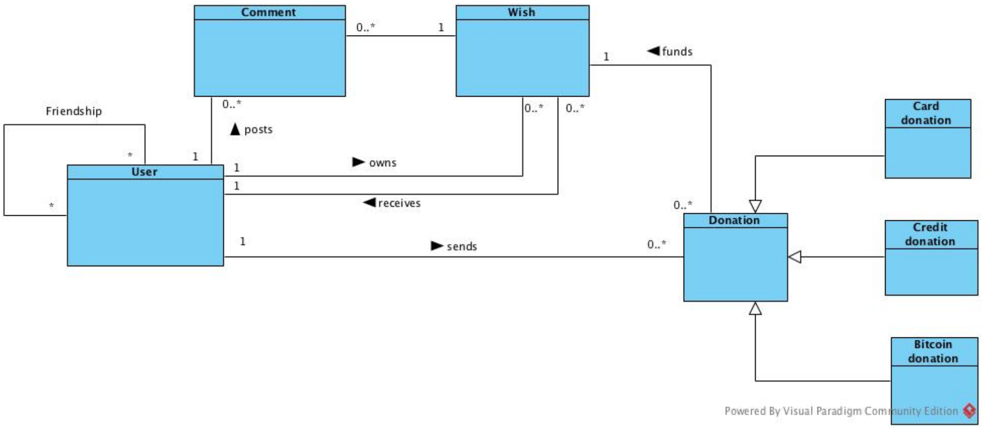


Figure 1.1: Domain model

1.1.2 Project structure

The project structure is visualized in the component diagram in 1.2.

The whole application stack is split into Android and iOS applications, a web application and the backend server. Backend consists of a PostgreSQL database keeping the data and REST API. The API itself is written in Django REST framework, which is a REST variant of the Django Web framework.

Django projects are often split into more modules, simply called applications. They are standalone, yet can easily interact and be combined together. This makes the project easier to navigate. Also they have separate database migration scripts, which helps separate the changes to the database structure and data. For example in the case of ElateMe, User management is separated from Wish related operations and models etc.

ORM (Object relational mapping) is achieved using Django models. These make use of the Django database API to communicate with the database.

The backend server provides an API for the client applications to connect to. Also the diagram 1.2 shows the Facebook Graph API used for User authentication and getting information about the user from Facebook. For payment processing Fio Banka is used to process card payments. Along with it the application will accept payments using Braintree and Bitcoin.

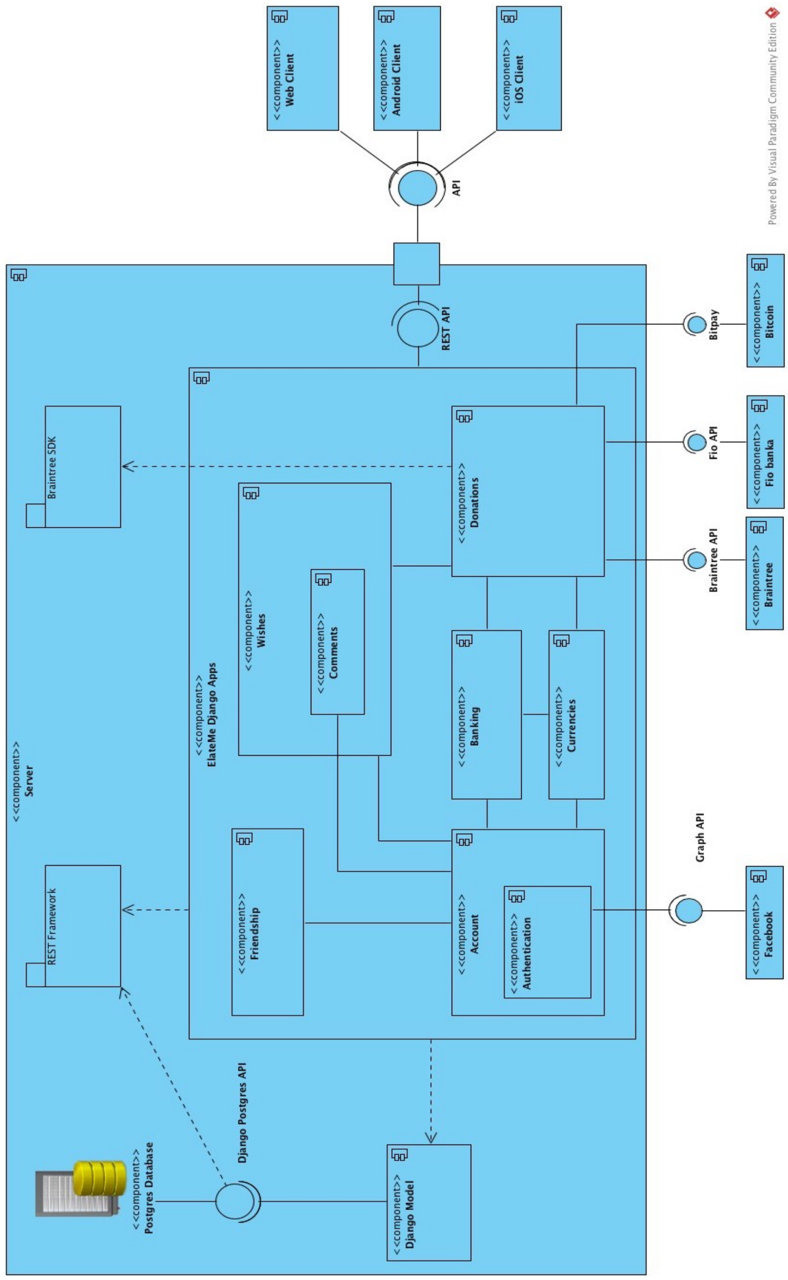


Figure 1.2: Component diagram

1.2 API Expansion

As the needs of the application evolve, the API has to reflect this. These necessary changes and additions to the backend are best documented as requirements. Requirements can functional or non-functional.

1.2.1 New functional requirements

In [3] functional requirements are described as statements of services the system should provide, how the system should react to particular inputs and how the system should behave in particular situations. It is also said that in some cases, the functional requirements may also define what the system in fact should not do.

1. **Banks and currencies**
   The application shall support several currencies for payments. Users should be able to choose their preferred currency out of the supported ones.

2. **User state**
   Users should be able to deactivate their account or request the administrator to delete it. Deactivated user's Wishes are no longer visible in the feed, but their comments on other Wishes still are. Deleted user's comments are no longer visible and their personal data is anonymised.

3. **Access to Wish**
   User outside of my friends should be able to ask the author to make the wish visible for them. If this permission is granted the Wish becomes visible and the user can post comments as well.

4. **User groups permissions**
   When creating a Wish a user should be able to select a group of his friends, that are granted access to the Wish. Should this group get updated, these changes should be reflected in the Wish visibility as well.

5. **User language**
   Users shall be able to select a preferred application language.

6. **Credit payment**
   As an incentive to use the application users can be given free credits, that they can then use to fund wishes. In order to enable this a credit payment system needs to be implemented. User will be given credits through the user administration.

7. **Push notifications**
   A modern push notification system needs to be implemented using Google Firebase messaging service.

8. **Long lived Facebook token**
   Due to the short term Facebook token, users were logged out after two hours and had to go through the whole login flow again, which is not a great example of good user experience. In order to fix that a long lived Facebook token, that expires in 60 days, needs to be generated. In Figure 1.3 the process of the generation is described.

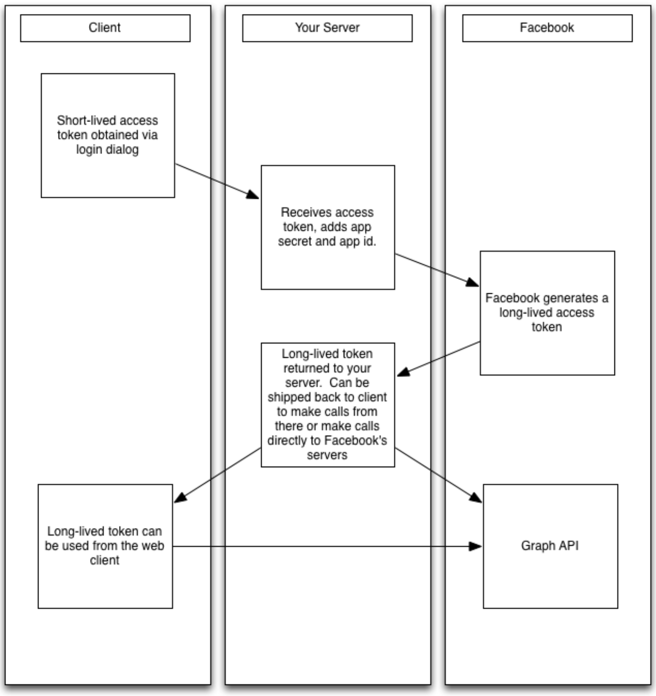


Figure 1.3: Long lived token generation [4]

9. **Birthday endpoint**
   In order to reduce the load on the API a new endpoint returning upcoming birthdays is needed.

10. **Administration**
    A complex user administration needs to be introduced, where users can be modified. Also lists of supported banks, currencies and languages need to be editable without having to directly interact with the database.

1.2.2 Non-functional requirements

Non-functional requirements include constraints set on the system, performance requirements, security requirements, implementation requirements, such as programming language etc. They usually apply to the whole system rather than just parts of it [3].

1. **Python3 Django REST framework**
   The author will use the same technology as his predecessor to expand and improve the API.

2. **REST**
   API will continue to follow REST API design style.

3. **PostgreSQL**
   A PostgreSQL database will be used for data storage.

4. **git**
   Project source code will be subject to version control using git. All new features will be on a separate feature branch.

5. **Zabbix**
   ElateMe backend server will be monitored using the Zabbix monitoring solution.

1.3 API security standards

In order to run a platform like ElateMe, one has ensure that it is secure. Especially because online payments are processed, therefore user confidential data is exchanged and some of it stored as well. Also plenty of users' private data is kept in the database.

In short, API security is vital, as it serves as a gateway to the data stored and also plays a role in payment processing.

1.3.1 Principles

There are several universal principles one should adhere to when designing a REST API service, well documented in [5].

- **Stateless authentication**
  REST service should by by principle stateless, therefore every request should include authentication credentials.

- **Least privilege**
  By default every user should have every resource denied unless they were granted permission for it. Also users should have only the permissions they truly need.

- **Separation of privilege**
  A combination of conditions should be used to grant access to resources, rather than single universal one. For example a user has to be authenticated and allowed to post comments on my Wishes.

1.3.2 HTTPS

HTTPS or HTTP Secure is a secure version of the HTTP (Hypertext Transfer Protocol). It uses TLS (Transport Layer Security) to encrypt the connection providing confidentiality, authenticity (user is connecting to the desired destination) and integrity (data has not been tampered with) [6].

This is important because it means that even when the communication between the user and the API gets intercepted, it cannot be decrypted. Also users can be certain that the data they received from the API is truly sent by ElateMe API. This enables us to send sensitive data over the internet.

1.3.3 Authentication

Authentication is a process when user credentials are verified. These credentials are sent with each request using the "Authorization" header. There are several ways to authenticate users to an API, with the most simple one called Basic authentication [7].

It is a simple login and password pair sent along with the request. Because the credentials are sent in plain text, it is necessary to use basic authentication only with HTTPS connection. An improved version is the HMAC (Hash Based Message Authentication) which sends over the hashed version of the password [7].

1.3.3.1 Basic Authentication

Basic authentication is not very well suited to an application like ElateMe. There are two main reasons for that. Firstly user credentials have to be stored on the client to be sent along with each request, which is a security risk. Also there is the possibility of brute force attacks, which could, even when unsuccessful, heavily impair the service performance.

With the inclusion of Facebook into ElateMe the decision was to use OAuth2.0, as it is what the Facebook interface for third-party application uses [1]. This way the authentication and much of the burden coming with it was "outsourced" to Facebook.

1.3.3.2 OAuth2

OAuth offers a solution to sharing information with third party services, where the typical login-password based approach is risky due to the problems described above.

OAuth2 is an improved version of the original OAuth1 protocol. Innitially OAuth1 brought more security, but at a cost of difficult implementation. The second version moved towards easier implementation with some compromises at the security level. Today the trend is to use OAuth2 with cryptographic extensions for additional security [7].

One of the basic concepts in OAuth is separating the party that requests access to the data from the party that owns the data [8]. ElateMe uses Facebook not only to authenticate its users, but also to get their friends, their contact info, profile pictures etc. In order to achieve this once a user clicks on

"Login via Facebook" they authenticate to Facebook, get a token in return, which is sent over to ElateMe server. This user token is then used, along with the App secret to requests user data from Facebook.

1.3.4 Authorization

Authorization determines whether an authenticated user has permission to perform a given action [7]. In Django REST framework this is achieved using permission classes. The basic ones are provided by the framework, but it is easily possible to write custom functionality by inheriting the `BasePermission` class. By default every user has to be authenticated to use ElateMe API calls except for the login call.

1.3.5 Best practices

According to [9] REST API security can be broken down to following these best practices:

- **Use HTTPS**
  Always use HTTPS for API communication.

- **OAuth authentication**
  For large scale applications OAuth based authentication is better than basic auth.

- **No sensitive information in URLs**
  It is vital not to include any user sensitive information, such as passwords and login in the URL itself, use an "Authorization" header instead.

- **Hash IDs**
  Using a hash to identify entities, rather than sequential integer id prevents the potential attacker from guessing the next number in the sequence.

- **Administration security**
  If there is an administration page present, naturally it has to be adequately secured. Using two factor authentication and good, secure passwords.

- **Data validation**
  In order to prevent cross site scripting, it is advised to validate content types, validate method parameters and restrict them where possible.

- **Method restriction**
  API enpoints should accept only allowed methods (GET, POST, PUT, DELETE).

- **CORS**
  Cross-origin resource sharing is a standard to specify what cross domain resources are accepted. Default policy should be to dismiss any non-whitelisted domains.

1.4 Payment request API

As the main purpose of ElateMe is to raise money, to fund its users' wishes, a wide variety of payment methods is needed. The following payment methods will be supported by the application:

- Credit card
- Google Pay
- Braintree payment
- Bitcoin payment

In order to make payments as comfortable as possible, it was decided to opt for the most recent standard in online payments, the Payment request.

1.4.1 Concepts

Paying on the internet is often frustrating. Filling out long forms can be off-putting for many users, especially on mobile devices, which is the reason a purchase is often abandoned right before the checkout. Moreover for each payment method supported, the checkout form gets even more complicated and tedious to fill out [10].

The Payment Request API aims to make this process simpler and comes up with a new standard. It's goal is to minimize checkout forms and therefore improve user experience. Also it can be used regardless of the payment method, because it does not directly interact with the payment provider (a credit card vendor, PayPal etc.). Instead a web browser is used as an intermediary between the application and the payment provider. This way the necessary information can be stored in the browser, without the user having to type it in every time they want to pay [10].

Using Payment Requests is possible with Google Pay, Braintree and credit card payments.

1.4.2 Payment process

The merchant, in this case ElateMe application, creates a new `PaymentRequest` and passes this information to the browser. The browser checks the compatibility between supported payment methods and presents a payment UI to the user. Once authorized the information is then passed to the payment provider, with the result returned back to the application.

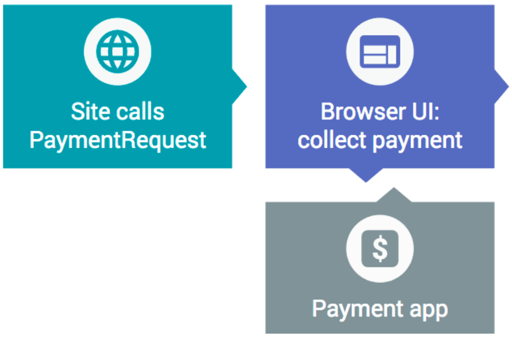


Figure 1.4: Payment request process [10]

1.4.3 Server side implementation

Once the payment process has been completed, it is necessary to pass on the information to ElateMe API. Then a Donation record is created in the database and thus the donation process is concluded.

1.5 User administration

A complex application like ElateMe needs an administration mechanism, where administrator users can modify database entities. Preferably without having to directly interact with the database, as was the case until now, which is much more prone to error. Also often times programming language objects can not be directly mapped to database entities, especially with M:N relations etc.

Django web framework has an administration application built-in, simply called "Admin". Admin makes use of the models defined to build a site, where one can easily create, update or delete database records. It is recommended to be used for internal data management, because a certain level of understanding of the underlying data is required [11].

This is exactly the case with ElateMe, where the major use cases are, adding credits to a user and managing supported currencies, languages and banks. Also with the current legislature, users can ask for their user data to be deleted or anonymized, which is another good use case for the administration.

CHAPTER 2

Design

This chapter begins with an overview of the technologies used in the project, followed by proposed designs of new functionality for the API. These designs make use of open source libraries, incorporating them into the ElateMe back-end application.

2.1 Used technologies

2.1.1 Python

ElateMe API is written in Python3, which is a modern, object-oriented language. It is easily readable and with plenty of users.

2.1.2 PostgreSQL

PostgreSQL is a powerful, open source object-relational database. It serves as a storage for ElateMe's data. Django supports many other database backends, however Postgres is well established as a stable and powerful database to use with Django projects.

2.1.3 uWSGI

uWSGI is a popular web server that implements the WSGI (Web Server Gateway Interface) standard, which is an interface between a web server and a Python application. Also it makes use of the uwsgi protocol to communicate with other web server software such as Nginx, which is often used in front of uWSGI to improve static content throughput [12].

2.1.4 Docker

Docker is an open source container platform, aimed at helping developers develop, deploy and run applications. Its key features are flexibility, scalability and portability. Containers are lightweight virtual machines created from a base image. These containers share kernel with the host machine, running as a process on it [13].

ElateMe's REST API is deployed in a Docker container. The definition of a Docker container is specified in a **Dockerfile**. Dockerfile contains a base image definition, a sequence of commands that are to be executed and which ports should be exposed. The base image usually contains a lightweight operating system image and the runtime environment for your application [13]. In our case that is Python 3. There is plenty of community created and maintained Docker images for all kinds of applications.

2.1.5 Docker compose

Docker compose is a tool for running a multi container Docker setup. A YAML (YAML Ain't Markup Language) file is used to define multiple different Docker applications and their interaction [13].

In the case of ElateMe there are two Docker applications, the API and the Postgres database. They each have a separate Dockerfile, with Docker compose being used to combine them together and deploy as a single stack.

2.1.6 Django REST framework

Django REST framework is an open source framework for REST APIs, built upon the popular Django Web framework. It contains additional tools, such as API endpoint URL definitions, resource permissions and serializers [14].

2.1.6.1 Architecture

However Django Web framework does not follow the MVC (Model View Controller) pattern fully, which is why it is often referred to as an MTV framework, which stands for Model, Template, View. In Django Models are the same as in MVC, but the what the user sees is described by Templates, with Views handling the business logic, thus filling the role of Controllers [15]. Because Django REST framework is built upon the web version, it follows the MTV pattern as well. However it needs not serve web pages to a browser, which makes Django templates obsolete.

2.1.6.2 Models

Models are the definitions of data handled by the application [16]. They are all a subclass of django.db.models.Model [16]. Each model attribute represents a database field, with some exceptions, such as M:N relationships, which are handled by creating an additional table.

Each Django object, represented in the database, is an instance of its Model. Model instances are identified by a primary key field. If not specified.

Django will automatically create an id field. Naturally, Djago supports foreign keys, many-to-one and many-to-many relationships [16].

Changes to the Model are propagated into the database using the so called migration scripts. These can be automatically generated or written from scratch. Model metadata, such as database table name, string representation etc. can be edited using the inner Meta class.

2.1.6.3 Views

Views represent the business logic of the application. They take a request and return a response.

Django REST Views are different from Django Web Views in several ways. First off they handle Request instances instead of HttpRequest, similarly the response is a Response instance instead of a HttpResponse [14]. Each API endpoint (URL) has a Django View associated with it, that is handling the incoming requests. Django REST also provides so called generic APIViews, which serve to make the implementation of common actions and patterns easier [14]. For example a generic `ListCreateApiView` allows only the GET and POST methods.

Views, through their attributes, are also responsible for checking permissions, pagination and importantly selecting the Serializer class.

2.1.6.4 Serializers

Serializers convert Django querysets and model instances, often passed from View classes, to be converted to native Python datatypes and then rendered into JSON, XML or other [14]. The oposite process is called deserialization, where parsed data is converted back into Django objects [14].

Serializers also provide request data validation and often fill in information not included in the request.

2.2 API versioning

API versioning is running several versions of the same API simultaneously. This comes especially useful when one has multiple different clients, as is the case of ElateMe. As was said, there is a web application, an Android application and an iOS one. Oftentimes the development cannot progress on each of them at the same time, which is when different versioned API comes useful.

With API versioning in Django REST, the changes will be most often made to the business logic, represented by the `Views` and the data that gets sent or processed, which is managed by `Serializer` classes.

When it comes to making truly significant changes, where the database schema is heavily altered, one has to decide between keeping backwards compatibility, for the older versions to use, and deploying another database in parallel.

API versioning is disabled by default and currently there is no versioning on ElateMe API. The most suitable approach to API versioning boils down to the required scale. Both smaller and larger scale possible approaches to versioning are documented bellow.

2.2.1 Versioning schemes

There are several versioning schemes supported by the REST framework. In principle they can be broken down to two categories, URL based and request header based. In all cases the versioning scheme passes on the selected version as a request attribute.

- **Accept Header Versioning**
  The client has to specify the version as part of the media type in the Accept HTTP header. It is included as a media type parameter, additionally to the main media type. It is considered to be the best practice when it comes to versioning [14]. An example of the request header can be seen in 2.1, where the version is set to 1.0.

```
GET /wishes/ HTTP/1.1
Host: api.elateme.com
Accept: application/json; version=1.0
```

Listing 2.1: Accept header example

- **URL Path Versioning**
  URL path versioning is a process, where the client connects to a different version of the API depending on the version specified in the URL [14]. Usually version specification is the first part of the URL.

  There are two different approaches to implementation of URL versioning in Django. They are the same from the client's perspective, however in one case the version is matched as a keyword in Django application URL configuration file, as shown in 2.2. The other approach is namespace based, where the urls are namespaced by the selected version.

```
urlpatterns = [
    url(
        r'^(?P<version>(v1|v2))/wishes/$',
        wishes,
        name='wishes-list'
    )
]
```

Listing 2.2: URL configuration example

- **Query Base Versioning**
  Query based versioning is probably the simplest one, where the requested version is specified as a request query parameter, as shown in 2.3 [14].

```
GET /wishes/?version=1.0 HTTP/1.1
Host: api.elateme.com
Accept: application/json
```

Listing 2.3: Query versioning example

2.2.2 Smaller scale

In small scale projects or when not all endpoint are required to be subjected to versioning, the best approach would be to use the simple URL path versioning scheme. When the versioning scheme is selected, the request that is passed on to the view has an attribute named `version`. This attribute is matched based on the process described in listing and can be used to make decisions on how to handle the incoming request. An example of varying behaviour can be selecting a different serializer or permission class. An example of selecting a serializer by overriding the `get_serializer_class` method is in listing 2.4.

As of now this is an acceptable approach to ElateMe's API versioning, robust enough to handle client differences, while maintaining lower complexity and high readability. A great advantage with ElateMe is that all the clients are developed "in-house" along with the API. More problems arise, when one enables their API to be used by third parties.

```
class CurrentUserDonationsView(generics.ListAPIView):
    renderer_classes = (renderers.JSONRenderer,)
    permission_classes = [permissions.IsAuthenticated, ]

  def get_serializer_class(self):
        if self.request.version == '1.0':
            return DonationsSerializerVersion1
      return DonationsSerializerBase
```

Listing 2.4: Serializer selection based on version

2.2.3 Large scale

A complex versioning system is mostly used in large APIs, which are often publicly accessible. An example of this is Facebook's GraphAPI, which is versioned by URL and currently running 8 versions concurrently [17].

Regardless of which versioning scheme one opts for, when running more than two versions and multiple versioned endpoints, it is required to come up with a system to manage the overhead. There are some best practices related to running a large scale versioned API in Django REST framework.

The first thing is to set up a maintainable project structure, to make project easier to navigate. An example of a good structure is figure 2.1. Each Django application, subjected to versioning should have base classes, preferably in a directory named `base` or `default`. This directory should contain default views and serializers, with basic functionality that can be inherited from by the versioned ones.

```
default
├── init.py
├── router.py
├── views.py
└── serializers.py
versions
├── v1
│   ├── init.py
│   ├── router.py
│   ├── views.py
│   └── serializers.py
└── v2
```

Figure 2.1: Versioned API project structure

Also to make the routing (process of matching the right serializer/view to the desired version) easier `ViewSet` and `Router` classes can be used. These reduce the `if` branching of the code and enforce consistent URL style.

2.2.3.1 ViewSet

ViewSets can be used to group together related views in a single class [14]. They are often used alongside `Routers` to automatically generate URL configuration.

2.2.3.2 Router

Routers are used to automatically map views to URLs [14]. One just registers desired views and their prefix and the configuration gets generated .

2.2.4 Caveats

Supporting multiple API versions comes with a price. Especially when an API is publicly accessible and widely used, discontinuing outdated versions requires prior announcements, upgrade plans and proper documentation in general.

Also in every case it makes the code harder to understand with an added layer of branching. This effect should be mitigated by using inheritance wherever possible to make as much behaviour universal and predictable.

When maintaining multiple versions, say one for each client platform, there is a danger of increasing the differences between the platforms, which makes more and more of the API code platform specific. This usually causes problems down the line and requires extensive refactoring to solve.

2.3 Administration

ElateMe's administration is implemented using the built-in Django admin. Administrators can authenticate to the application using a login (email) and password pair. In order to be able to use application's users with the admin, it was necessary for ElateMe's `User` class to inherit from Django's `PermissionsMixin` class. This adds additional permissions, such as superuser permission, group membership and custom user permissions.

In each Django application, that needs to have an admin interface a file `admin.py` is created. The required models are then registered into the Django Admin. In order to customize the administration view, add searchable fields and filters a subclass of `admin.ModelAdmin` can be created, which is then registered alongside the model.

2.3.1 Admin security

Because the Django Admin gives direct access to the data stored in the database, it is important to secure the site properly. As an addition to the security best practices described above, a strong administrator password is required. Also it should be considered moving the admin to a less obvious URL, rather than standard `/admin`. It is possible to make the admin site accessible only from the local network and connect to it using a VPN from the outside.

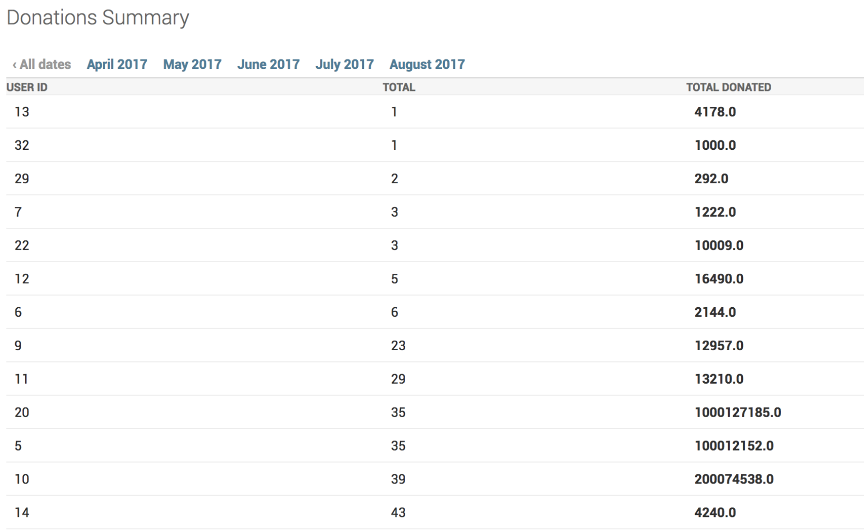
Additionally, as documented in [18], it is possible to set up a two-factor authentication using the `django_otp` package and overriding the default admin site with `OTPAdminSite`.

2.3.2 Dashboard

Django admin site can be also used as a lightweight dashboard to display some metrics about the application. For example it could be convenient to have a visualization of the average donation amount, wishes in the funding phase etc.

This can be achieved by taking the model class of the entity to be used in the dashboard and creating a `proxy` subclass. Proxy model classes add functionality, however are not represented as new tables in the database [14]. This proxy class can be then used as a data source for a new `ModelAdmin` class. This is done so that there are two separate admin sites. One where we can actually edit the individual records and one for the dashboard purposes.

The whole process is well documented in [19]. By overriding the `changelist_view` method it is possible to edit the view that would normally be rendered in the admin site. A custom HTML template can be then created to visualize the data sent from the `changelist_view` method. A very simple table showing the amount of money donated by each user in different time periods can be seen in figure 2.2.


Donations Summary

‹ All dates April 2017 May 2017 June 2017 July 2017 August 2017

USER ID	TOTAL	TOTAL DONATED
13	1	4178.0
32	1	1000.0
29	2	292.0
7	3	1222.0
22	3	10009.0
12	5	16490.0
6	6	2144.0
9	23	12957.0
11	29	13210.0
20	35	1000127185.0
5	35	100012152.0
10	39	200074538.0
14	43	4240.0

Figure 2.2: Custom admin dashboard

2.4 Credit payment

Credits were implemented to ElateMe as an incentive for users to fund wishes. Credits are given to the users by the administrator using Django Admin.

With a credit account it is necessary to keep track of each user's credit balance, their transactions and which donations were executed using credit funding. Diagram 2.3 shows the model classes related to credit payments.

Users can later get a list of their own credit transactions using the endpoint `credit/transactions`. This is realized using the `UserTransactionsView` class, which automatically returns transactions of the same user that sent the request.

When posting a credit donation, user makes a POST request on `/credit/WISH_ID/` URL. With each payment a`CreditTransaction` is created, as long as the user has sufficient amount of credits in their account. It is also possible to get an overview of credit donations to a certain wish using the same `credit/WISH_ID/` endpoint, but the user requesting this information has to be the author of the wish. This functionality is handled by the `CreditTransaction View` class, which responds both to GET and POST requests.

Each view has different permission classes and a different serializer class. This is because the post request has to go through validation, to check payee's credit balance.

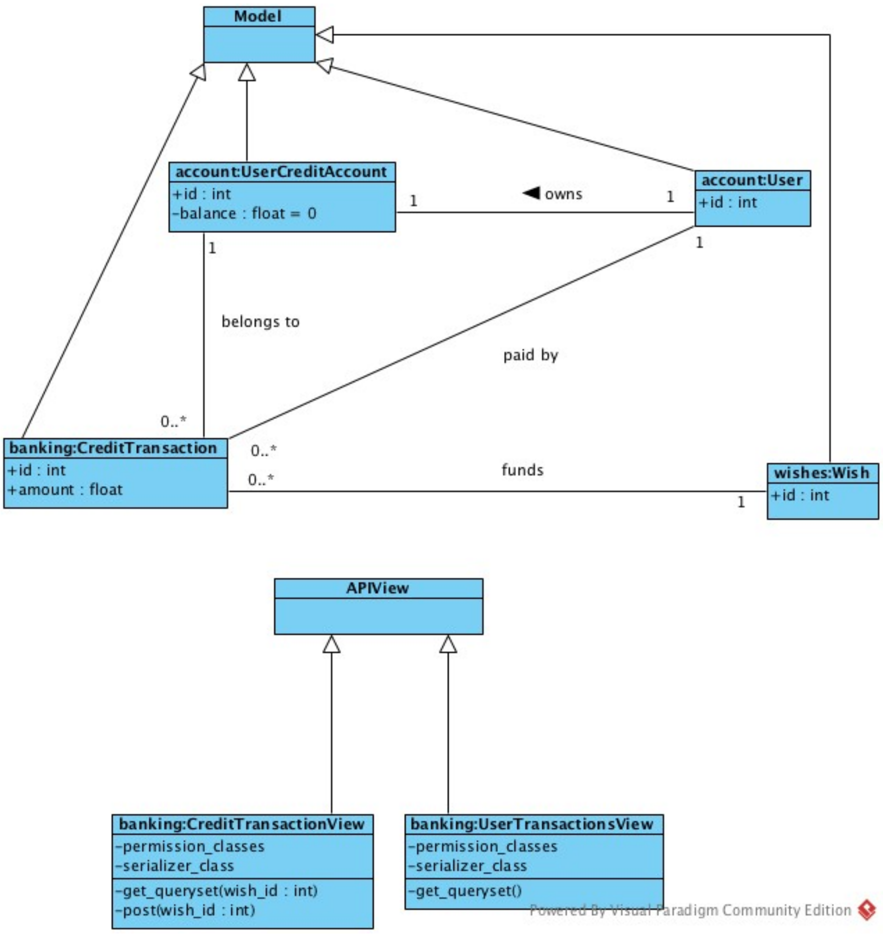


Figure 2.3: Credit class diagram

2.5 Product suggestions

During creation there are two ways a user can select the subject of their wish. Either create a custom wish, where they specify what the wish is, optionally upload a picture and set the amount of money they want to raise. The other option is that while typing in the wish title, they select one of the products suggested by the product drop down selection. This behaviour is demonstrated in figure 2.4.

While typing in the wish title the client issues a GET request on the `/wishes/suggested` endpoint sending along the user's input as a query string. The backend server then performs a search on Zbozi.cz and Alza.cz sites and returns a JSON object composed of possible products to the client. The backend first tries to search on Zbozi.cz and if unsuccessful queries Alza.

As ElateMe aims at the global market, Zbozi.cz and Alza are not ideal as the only stores to be present in the application. It was decided that a more widespread product marketplace search has to be implemented. Also the application could generate additional revenue using various affiliate programs.

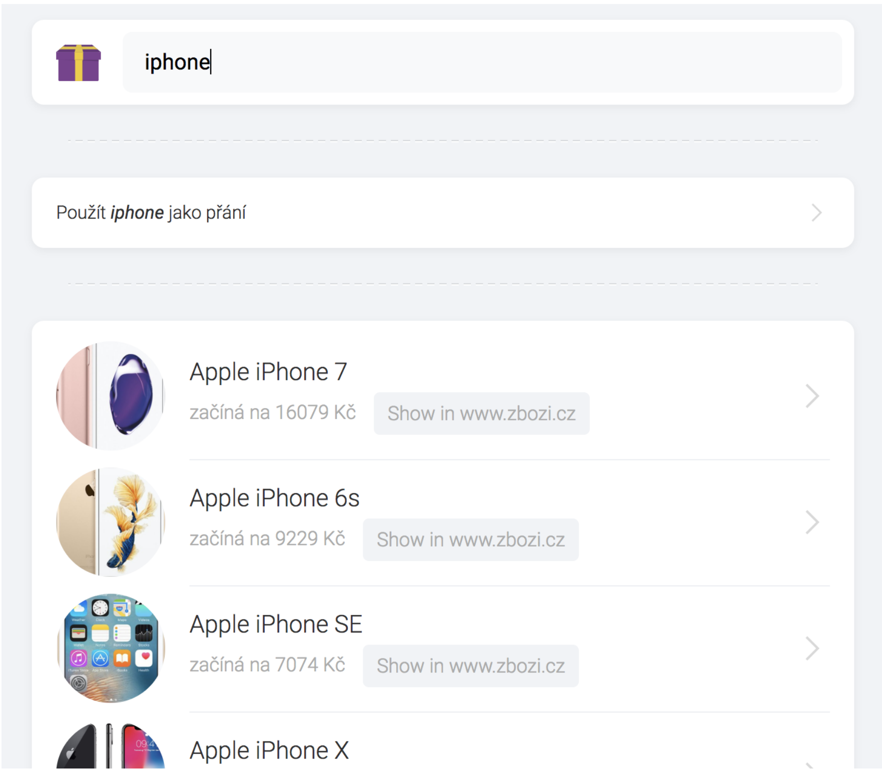


Figure 2.4: Product suggestion dropdown

It was decided to opt for inclusion of either Amazon or eBay in the product suggestions as they both have a global reach and provide an API for product lookup.

2.5.1 Amazon

In order to use Amazon's Product search API one needs an Amazon account. Each account has an associate tag assigned, which is used to identify look ups and get a commission. Each account is localized so each locale requires a separate registration [20].

Once registered one can use the Product search API which has two relevant methods `ItemSearch` and `ItemLookup`. There is a nice open source pacakge `python-amazon-simple-product-api` developed by Yoav Aviram providing an interface around Amazon's API [21]. Once installed it provides two important methods, `lookup` and `search` [21]. These match the Amazon's texttItemSearch and `ItemLookup` with lookup being able to look up products based on their Amazon `ItemID`, whereas search working as a keyword based search engine.

Both methods return an iterbale object, which can be then passed to a parser method to be turned into a JSON. This is then returned as a response to the client.

This process can be either incorporated under the current `/wisges/suggested/` endpoint and `SuggestedWishesView` or a new endpoint specific to Amazon will be created. This is advantageous as clients could connect to different endpoints based on their location.

2.5.2 eBay

With eBay the process is quite similar. After registering the application in eBay developers account, one is able to make use of the powerful eBay product API. There is an official open source Python SDK maintained by Tim Keefer [22]. It provides a thorough interface to eBay API functionality, while being fairly easy to use.

After authentication and establishing a `Connection` to the eBay api, one can use methods like `findItemsAdvanced` to find products listed on eBay [22].

It would make sense to combine results both from Amazon and eBay into one endpoint, as they are both international. A suggestion of the final result is visualized in diagram 2.5. Using the Python SDK it is not needed to parse the data ourselves, as it has a built in methon `json()` which converts the data to JSON automatically.

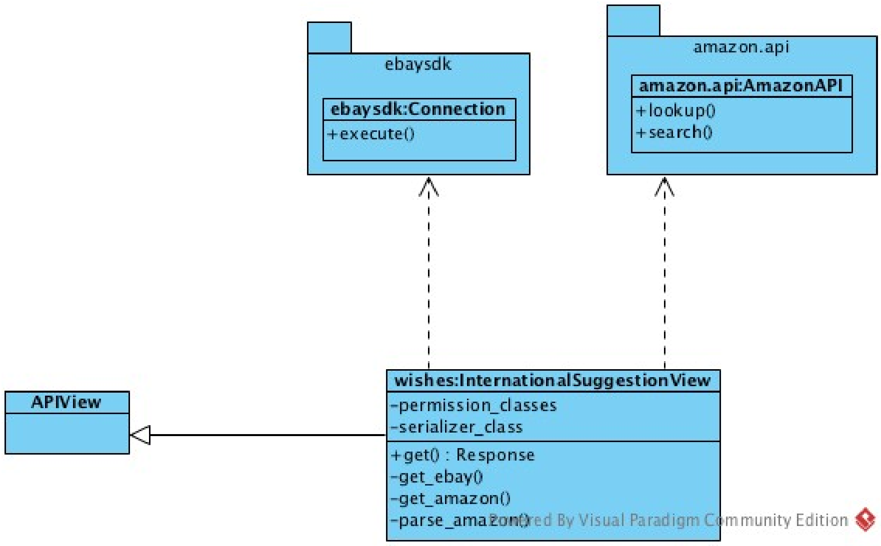


Figure 2.5: Product suggestion structure

2.6 Monitoring and alerting

When running a platform like ElateMe in production it is absolutely vital to detect problems with the platform as soon as possible. This is solved by a monitoring system, that can monitor all the parts of the backend server, which in our case is the PostgreSQL database and the API. Zabbix was chosen as the monitoring system for ElateMe.

2.6.1 Zabbix

Zabbix is an open source distributed monitoring system. It works on a client–server principle, with clients collecting data and reporting to the central server. It allows for email alerts to be sent based on user defined criteria. All the information is accessible using a web frontend, which can be accessed from anywhere.

Instances monitored by the Zabbix agents are called `hosts` which can be either polled for information or they actively send it themselves.

2.6.2 Server deployment

As was already said, both the Postgres server and the API run inside Docker containers on a single Virtual Machine. Therefore it makes sense to have the monitoring in Docker as well. The main reasons for this is there isn't a production environment for ElateMe set up yet. Docker containers are quick to set up and tear down, so there won't be too much work wasted, should ElateMe move to a different server.

First a Zabbix server is needed. It will be deployed using the `Dockbix` docker image. It is a preconfigured Docker image for Zabbix server deployment, maintained by Monitoring Artist. It uses a mySQL database for data storage, which can be deployed inside a Docker container as well.

As for the monitoring itself a `Dockbix-agent` will be used, which is yet another Docker-deployed application. It contains Docker monitoring for all containers running on the host as well as host monitoring. This agent will be configured to send data to the Zabbix server, port 10051.

Chapter 3

Implementation

This chapter contains a description of the API's implementation. First the project implementation will be described in general and secondly the newly implemented parts will be covered.

3.1 Project apps

The project is split into the following Django applications:

- **Account**
  Account handles Users, their data and related objects such as user's credit account, profile pictures etc. Also it has two nested applications, Authorization and Social. Authorization handles user login and authentication, with Social providing the social API connected to Facebook's GraphAPI [1].
- **Banking**
  Banking takes care of credit transactions and provides models for supported countries, card transactions and data transactions that are then used in the Donation app.
- **Currencies**
  Currencies is a very small, simple application which handles only supported currencies.
- **Donations**
  All the payments and donation overview is handled by the Donations app. It covers Braintree payments and card payments, with Bitcoin payments coming in the future.
- **Feed**
  Feed is another simpler application with the main functionality being returning a list of wishes to be displayed in the frontend feed.

- **Friendship**
  Friendship application processes the Friendship relations between users.

- **Notifications** Currently this application provides a list of user's notification and provides an interface called `NotificationManager` for sending notifications.

- **Wishes**
  The Wishes app handles wish creation, product suggestion, access to wishes, wish image upload and various wish views. Also the Comments application is nested within, handling wish comments.

- **Server API**
  This is the central application tying all the others together.

Interaction between different applications is handled mostly by importing Models in order to create ForeignKey relationships and create database entries belonging to a different application. Each application contains its own migration scripts, which are used to make changes to the database model or seed it with new data.

3.2 Settings

Application settings are handled in the `server_api` application. In the central `urls.py` file the main routing decisions are defined, where the top-level URLs are listed as in listing 3.1.

The universal settings are defined in the `settings.py` file, where all the applications are tied together, along with other applications used, such as the `rest_framework` itself and all the middleware used. Based on the environment the application is running in, additional settings are imported from `ENV_settings.py` files. The runtime environment is defined at application startup.

```
urlpatterns = [
    url(r'^account/',\
        include('account.urls', namespace="account")),
    url(r'^wishes/',\
        ('wishes.urls', namespace="wishes")),
    url(r'^donations/',\
        include('donations.urls', namespace="donations")),
```

Listing 3.1: Top level URL config

3.3 API service expansion

In this section some of the new more interesting minor additions to the API will be described.

3.3.1 Banks and Currencies

In order to be able to match Account transactions to particular bank carrying out the payment, the supported banks had to be added to the application. Also because payments can be a carried out in any of the supported currencies, which are currently: Czech Crown, United States Dollar and Euro, models representing these had to be added.

3.3.2 Wish access

A situation could arise, when someone who is not my friend on Facebook, but uses ElateMe as well, might want to donate to one of my wishes. For example our mutual friend sent him a link to it, but he cannot see the wish contents. In order to enable users outside of friends to see certain wishes a wish access request has been implemented.

When a user comes upon a wish they are not authorized to see, a button "Request access to Wish" is displayed to them. Upon clicking on it a POST request is sent to the API server on endpoint `/access/request/` with the wish ID in the payload. This is handled by the `WishAccessRequestView` seen in listing 3.2. In the serializer handling this request a `WishAccessRequest` object is created, where the status of the request is recorded. The view then sends a notification to the wish author, that a user has requested access, which can be either granted or denied using a POST request to the `/access/WISH_ID/` endpoint. Once granted the petitioner is given notification that they can visit the wish.

```python
# Handles Wish access grant/deny
class WishAccessView(generics.UpdateAPIView):
    permission_classes = [permissions.IsAuthenticated,\
        custom_permissions.IsWishAuthor]
    serializer_class = serializers.WishAccessGrantSerializer
    queryset = WishAccessRequest.objects.all()
    lookup_url_kwarg = 'request_id'

    def perform_update(self, serializer):
        serializer.save()
        if serializer.data['status'] == 'granted':
            request_id = self.kwargs['request_id']
            request = WishAccessRequest.objects.get(id=request_id)
            wish = Wish.objects.get(id=request.wish_id)
            wish.allowed_users.add(request.petitioner)
            Manager.add_reveal_granted(request.petitioner, wish)
```

```

# Handles Wish access request
class WishAccessRequestView(generics.ListCreateAPIView):
    permission_classes = [permissions.IsAuthenticated]
    serializer_class = serializers.WishAccessRequestSerializer

    def perform_create(self, serializer):
        serializer.save()
        petitioner = get_object_or_404(User, \
            id=serializer.data['petitioner'])
        wish = get_object_or_404(Wish, id=serializer.data['wish'])
        influencer = get_object_or_404(User, \
            id=serializer.data['author'])
        Manager.add_reveal_request(petitioner, wish, influencer)

    def get_queryset(self):
        current_user = self.request.user
        return WishAccessRequest.objects.filter(petitioner=
    current_user)
```

Listing 3.2: Wish access handling

3.3.3 Group permissions

Because users can group their friends into groups it would be practical to be able to use the groups for giving access to wishes. During creation a user can choose between making his wish public, visible to all friends or just to certain people. And the last case is where groups come in, as it could be quite tedious having to manually select many friends. With groups one can just select one or more groups.

Each `Wish` instance has an attribute `allowed_users`, which could be used to propagate users from groups during wish creation. However this would not reflect later changes made to the groups. In order to solve this a new attribute `allowed_groups` has been added and users' permissions are cross-checked against the allowed group as seen in listing 3.3.

```
def UserInGroup(user, wish):
    for w in wish.allowed_groups.all():
        if user in w.members.all():
            return True
    return False
def has_permission(self, request, view):
    ...
     if wish.is_public:
         return True
     return wish is None or \
         request.user == wish.author or \
         (request.method in permissions.SAFE_METHODS and \
         (request.user in wish.allowed_users.all()) or\
         UserInGroup(request.user, wish))
```

Listing 3.3: Group permissions check

3.4 Administration

ElateMe administration is realized using the built-in Django administration. Admin provides the administration of Users, Wishes, Banks, Currencies, Countries and Donations. A two factor authentication into the admin site is implemented using the `django_opt` package, yet currently disabled.

The administration can be further adjusted to the evolving needs of the frontend by adding models and other functionality. Each registered model can be customized using a `admin.ModelAdmin` subclass, as shown in listing 3.4. In this example user's credit account is inserted into user administration, despite being a separate model using the `inlines` attribute.

This can be also used to create additional dashboards and graphs by providing a HTML template to render the data sent from the `ModelAdmin` class.

```
class UserCreditAdmin(admin.TabularInline):
    model = UserCreditAccount

class UserAdmin(admin.ModelAdmin):
    inlines = (UserCreditAdmin,)
    list_display = ('email', 'last_name', 'first_name', 'state')
    list_filter = ('state', 'language', 'currency')
    readonly_fields = ('first_name', 'last_name', 'date_of_birth',
     'date_created')
    search_fields = ('last_name', 'first_name', 'email')
```

Listing 3.4: User administration customization

3.5 Credit payment

Credit payment is mostly handled by `CreditTransactionView` with most of the logic happening in the serializer as seen in listing 3.5. Data that wasn't part of the request is filled in, user's credit balance is checked, a transaction record object is created and the balance is adjusted accordingly.

In order to make Donations tracking easier, a Donation object ties to either an `AccountMovement` or `CreditTransaction` using a foreign key. This way all the donations are kept track of in a single place.

```
    def create(self, validated_data):
        payee = self.context['request'].user
        validated_data['payee'] = payee
        kwargs = self.context['view'].kwargs
        wish_id = kwargs['wish_id']
        account = payee.credit_account
        validated_data['account'] = account
        if float(abs(validated_data['amount'])) > account.balance:
            raise serializers\
                .ValidationError('Insufficient credit balance.')
        wish = get_object_or_404(Wish, pk=wish_id)
```

```
        transaction = wish.credit_transactions\
            .create(**validated_data)
        account.balance -= float(abs(validated_data['amount']))
        account.save()
        # Manager.add_thank_you_for_donation(payee, wish)
        return transaction
```

Listing 3.5: Credit payment serializer

3.6 Push notifications

Mobile devices push notifications will be handled using the Google Firebase Messaging (GFM). GFM is a continuation of the Google Cloud Messaging service with expanded functionality.

Firebase is a complex service providing not only messaging, but also authentication, a simple backend service, analytics and others. ElateMe needs only the messaging part, therefore it wasn't necessary to include the complete Python SDK for FCM and implement everything ourselves [23]. Rather than that an open source package `fcm-django` [24], maintained by Mojca Rojko, provides all the required functionality. It makes use of `FCMDevice` objects which can in our case matched with users we want to send messages to using the `send_message` method [25].

Currently the client side of Firebase cloud messaging is not implemented, so the functionality hasn't been tested yet. However there can be some preliminary functionality implemented on the backend. For example it would be needed to have an endpoint where user devices can be registered and the respective `FCMDevice` objects created. They also need to be paired up with users, to enable message sending. For this purpose and endpoint `/notifications/fcmdevices` has been created, which either gives a list of current user's devices or enables them to register a new one.

Messages can be then sent out either to singular devices or multiple using the `send_message` method. Both the registration view and messaging is showcased in listing 3.6

```
class FCMDeviceView(generics.ListCreateAPIView):
    permission_classes = [permissions.IsAuthenticated, ]
    serializer_class = FCMDeviceSerializer

    def get_queryset(self):
        return FCMDevice.objects.filter(user=self.request.user)

# sending messages
devices = FCMDevice.objects.filter(user=self.user.id)
devices.send_message(title="Title", body="Message")
```

Listing 3.6: Firebase cloud messaging support

3.7 Monitoring and alerting

Monitoring shall be implemented using the Dockbix system described in section 2.6. As described in [26], the Zabbix environment can be started up using the commands in 3.7. The first initiates the server, while the second starts the actual monitoring agents, which then report to the server. In order for the server to run, it needs a connection to a database. A current database can be used or a new one deployed.

The server accepts connections from agents on port 10051, while the frontend runs on port 8080. This is due to the fact that 443 is used for the api itself. The frontend can be accessed on `api.elateme.com:8080` with credentials `admin` : `zabbix`.

```
docker run \
    -d \
    --name dockbix \
    -p 8080:443 \
    -p 10051:10051 \
    -v /etc/localtime:/etc/localtime:ro \
    -v /etc/letsencrypt/live/api.elateme.com/cert.pem:/etc/nginx/ssl/dummy.crt:ro \
    -v /etc/letsencrypt/live/api.elateme.com/privkey.pem:/etc/nginx/ssl/dummy.key:ro \
    --link dockbix-db:dockbix.db \
    --env="ZS_DBHost=dockbix.db" \
    --env="ZS_DBUser=zabbix" \
    --env="ZS_DBPassword=elateM3Rules" \
    --env="XXL_zapix=true" \
    --env="XXL_grapher=true" \
     monitoringartist/dockbix-xxl:latest

docker run \
  --name=dockbix-agent-xxl \
  --net=host \
  --privileged \
  -v /:/rootfs \
  -v /var/run:/var/run \
  --restart unless-stopped \
  -e "ZA_Server=api.elateme.com" \
  -e "ZA_ServerActive=api.elateme.com" \
  -d monitoringartist/dockbix-agent-xxl-limited:latest
```

Listing 3.7: Dockbix startup

Chapter 4

Testing

Application functionality is ensured by thorough testing. It can be split into two parts: unit testing and acceptance tests.

Unit tests should be always written by the code authors. They are so called “white-box” tests, in the sense that the person writing the test is familiar with the inner workings of the tested software. Acceptance tests are usually testing the application like a black box, with the outcome the only important thing. ElateMe quality assurance (QA) is more in-depth covered in my colleague’s thesis [27].

4.1 Unit tests

Django REST unit tests are realized using the `APITestCase` class [14]. Using a production database to run tests would be impractical and potentially dangerous. Therefore the framework defines a new database, creates it, applies all migrations and runs the tests defined in `tests.py` files. This requires all data seeds to be present in the migrations, for the application to function.

Each test case is split up into a `setUp` phase, defined in a method of the same name, where all the prerequisite data is created [14]. This is then used in the testing methods, followed by a `tearDown` method which deletes it all.

It was necessary to adjust the tests to the changed database structure, remove unused methods and add new ones. In listing 4.1 is an example of a test using `APITestCase`.

```
cclass CreditTest(APITestCase):
    def setUp(self):
        self.user1 = UserManager().create_user('test1@test.com',\
            'test')
        self.user2 = UserManager().create_user('test2@test.com',\
            'test')
        self.user2.credit_account.balance = 100
        self.user2.save()
        wish = {
```

```
                'title': "iPhone X",
                'description': "I love cracked glass",
                'amount_needed': 29999,
                'author': self.user1,
                'user_money_receiver': self.user1,
            }
            self.wish = Wish.objects.create(**wish)

        def test_donation(self):
            url = reverse('banking:credit-transaction',\
                kwargs={'wish_id': self.wish.id})
            data = {
                'amount': 50,
            }
            status_code, data = post(url=url, user=self.user2,\
                data=data)
            self.assertEqual(status_code, status.HTTP_201_CREATED)
            self.assertEqual(data['wish'], self.wish.id)

        def test_insufficient_credits(self):
            url = reverse('banking:credit-transaction',\
                kwargs={'wish_id': self.wish.id})
            data = {
                'amount': 150,
            }
            status_code, data = post(url=url, user=self.user2,\
                data=data)
            self.assertEqual(status_code, status.HTTP_400_BAD_REQUEST)
```

Listing 4.1: API Test case example

There were plenty of unit tests already written as described in [1], however virtually all had to be rewritten to reflect changes to the models and permissions. Not all of them are currently passing as there are some errors in the application. Some functionality cannot really be tested, especially payment related methods, which calls for manual testing.

Overall the test coverage should improve as the application will near production release. Tests should also be run on every application build as part of Continuous Integration.

Conclusion

This thesis is very practical and industry oriented. The aim was to iterate on the work laid out by my predecessor and improve the API functionality.

The first part was analyzing the topic, new functional requirements and some additional challenges coming with running such a complex backend system in production environment. It was necessary for me to familiarize with the Django framework and the code already written.

In the design chapter the application structure and used technologies were covered, along with laying a foundation for the future expansion and improvement. It contains a description of a possible solutions to various components of the system, some which are already implemented or will be in the future. I put an emphasis on trying to integrate existing open source solutions, as they provide the desired functionality, while being easier to maintain as the project further expands. The topics covered include API versioning problems both in smaller and larger scale, product suggestions using Amazon an eBay APIs etc.

The implementation chapter covers the new functionality implemented as part of the thesis, including code snippets. I strove to include the reasoning behind my approach to these problems as well as the solution itself. The way unit tests are handled, using built in REST frameworks tools, is covered in the Testing chapter.

Contribution

Potential readers are given an insight into the structure and challenges of a larger scale REST API. They can familiarize themselves with recent best practices regarding API security and payment handling.

This thesis can also serve as a valuable document for future developers working on ElateMe.

Future outlook

In my opinion in order to close in on production readiness it is required to gradually put more effort into unifying and testing the current functionality, while reducing the amount of new to-be-implemented functionality.

However a very important step is the inclusion of international product suggestors, using Amazon and eBay APIs. This is a vital step in making sure the platform has a true global reach.

Bibliography

[1] Kuzmovych, Y. *ElateMe Backend I.* Bachelor's thesis, Czech Technical University in Prague, Faculty of Information Technology, 2017.

[2] Fowler, M.; Rice, D. *Patterns of Enterprise Application Architecture.* A Martin Fowler signature book, Addison-Wesley, 2003, ISBN 9780321127426. Available from: `https://books.google.cz/books?id=Jl5rkQnbfAIC`

[3] Sommerville, I. *Software Engineering.* International computer science series Software engineering, Addison-Wesley, 2007, ISBN 9780321313799. Available from: `https://books.google.cz/books?id=B7idKfL0H64C`

[4] Facebook. Refreshing User Access Tokens. *Docs*, 2018, [online], [cit. 2018-04-23]. Available from: `https://developers.facebook.com/docs/facebook-login/access-tokens/refreshing/`

[5] Gupta, L. REST API Security Essentials. *REST API Tutorial*, [online], [cit. 2018-04-29]. Available from: `https://restfulapi.net/security-essentials/`

[6] Google. Secure your site with HTTPS. *Google Search Console*, 2018, [online], [cit. 2018-04-29]. Available from: `https://support.google.com/webmasters/answer/6073543?hl=en`

[7] Levin, G. RESTful API Authentication Basics. *RestCase Blog*, 2016, [online], [cit. 2018-04-29]. Available from: `https://blog.restcase.com/restful-api-authentication-basics/`

[8] Reference. The OAuth 2.0 Authorization Framework. *OAuth*, 2012, [online], [cit. 2018-04-30]. Available from: `https://tools.ietf.org/html/rfc6749`

[9] Reference. REST Security Cheat Sheet. *OWASP*, 2018, [online], [cit. 2018-04-30]. Available from: https://www.owasp.org/index.php/REST_Security_Cheat_Sheet

[10] Kitamura, E.; Gash, D.; et al. Introducing the Payment Request API. *Web Fundamentals*, 2018, [online], [cit. 2018-04-23]. Available from: https://developers.google.com/web/fundamentals/payments/

[11] Mozilla; individual contributors. Django admin site. *MDN web docs*, 2018, [online], [cit. 2018-04-23]. Available from: https://developer.mozilla.org/en-US/docs/Learn/Server-side/Django

[12] Land, D. An introduction into the WSGI ecosystem. *Ultraviolet software studios blog*, 2017, [online], [cit. 2018-05-01]. Available from: http://www.ultravioletsoftware.com/single-post/2017/03/23/An-introduction-into-the-WSGI-ecosystem

[13] Reference. Docker overview. *Docker docs*, 2018, [online], [cit. 2018-05-04]. Available from: https://docs.docker.com/

[14] Reference. API Guide. *Django REST framework*, 2018, [online], [cit. 2018-05-10]. Available from: http://www.django-rest-framework.org/api-guide/

[15] Oyom, A. N. Understanding the MVC pattern in Django. *She Code Africa*, 2017, [online], [cit. 2018-03-29]. Available from: https://medium.com/shecodeafrica/understanding-the-mvc-pattern-in-django-edda05b9f43f

[16] Reference. Models. *Django documentation*, 2018, [online], [cit. 2018-05-01]. Available from: https://docs.djangoproject.com/en/2.0/topics/db/models/

[17] Facebook. Graph API. *Docs*, 2018, [online], [cit. 2018-05-12]. Available from: https://developers.facebook.com/docs/graph-api

[18] Reference. django-otp. 2017, [online], [cit. 2018-05-01]. Available from: http://django-otp-official.readthedocs.io/en/latest/

[19] Benita, H. How to turn Django Admin into a lightweight dashboard. *Medium*, 2017, [online], [cit. 2018-05-02]. Available from: https://medium.com/@hakibenita/how-to-turn-django-admin-into-a-lightweight-dashboard-a0e0bbf609ad

[20] Reference. Product Advertising API. *Developer Guide*, 2018, [online], [cit. 2018-05-04]. Available from: https://docs.aws.amazon.com/AWSECommerceService/latest/DG/Welcome.html

[21] Aviram, Y. Amazon Simple Product API. 2017, v2.2.11. Available from: https://github.com/yoavaviram/python-amazon-simple-product-api

[22] Keefer, T. python ebaysdk. 2017, v2.1.4. Available from: https://github.com/timotheus/ebaysdk-python/releases

[23] Google. Add the Firebase Admin SDK to Your Server. *Firebase Guides*, 2018, [online], [cit. 2018-05-04]. Available from: https://firebase.google.com/docs/admin/setup

[24] Rojko, M. fcm-django. 2018, v0.2.18. Available from: https://github.com/xtrinch/fcm-django

[25] Reference. fcm-django. *Docs*, 2018, [online], [cit. 2018-05-12]. Available from: http://fcm-django.readthedocs.io/en/latest/

[26] monitoringartist. Dockbix XXL. 2018, [online], [cit. 2018-05-01]. Available from: https://hub.docker.com/r/monitoringartist/dockbix-xxl/

[27] Grofek, T. *ElateMe - QA v multiplatformních aplikacích.* Bachelor's thesis, Czech Technical University in Prague, Faculty of Information Technology, 2018.

Appendix A

Acronyms

API Application Programming Interface

REST Representational State Transfer

ORM Object Relational Mapping

WSGI Web Server Gateway Interface

MVC Model View Controller

MTV Model Template View

URL Uniform Resource Locator

JSON JavaScript Object Notation

CRUD Create Read Update Delete

UI User Interface

HMAC Hash Based Message Authentication

CORS Cross-origin resource sharing

YAML YAML Ain't Markup Language

QA Quality assurance

Appendix B

Contents of enclosed SD card

```
readme.txt ........................ the file with CD contents description
setup.md ................................................... setup guide
src ..................................... the directory of source codes
    server-api ......... the directory containing application source code
    thesis .............. the directory of LaTeX source codes of the thesis
text ......................................... the thesis text directory
    BP_Hrachovina_2018.pdf ............. the thesis text in PDF format
```

Appendix C

Installation guide

Project setup guide is also in the setup.md file on the enclosed SD card.

C.1 Requirements

- Python 3
- Python pip
- PostgreSQL database server version 9.6.x

C.2 Setup

1. Create database
2. `cd src/server_api`
3. Fill in connection details in `server_api/dev_settings.py`
4. Make sure database is accessible
5. Install dependencies using:
   `pip install -r requirements.txt`
6. Either run migrations using:
   `python manage.py migrate`
   or import database data from dump.sql using:
   `psql -h HOST -u USER -d DATABASE -p PASSWORD < dump.sql`
7. Run API server by:
   `python manage.py runserver`
8. Run Tests by :
   `python manage.py test`