Insert here your thesis' task.

Czech Technical University in Prague
Faculty of Information Technology
Department of Software Engineering

Bachelor's thesis

# ElateMe - Backend

***Yevhen Kuzmovych***

Supervisor: Ing. Jiří Chludil

8th May 2017

# Acknowledgements

THANKS (remove entirely in case you do not with to thank anyone)

# 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 8th May 2017 .....

Czech Technical University in Prague
Faculty of Information Technology


### Citation of this thesis

Kuzmovych, Yevhen. *ElateMe - Backend*. Bachelor's thesis. Czech Technical University in Prague, Faculty of Information Technology, 2017.

# Abstrakt

V několika větách shrňte obsah a přínos této práce v českém jazyce.

**Klíčová slova** Replace with comma-separated list of keywords in Czech.

# Abstract

Summarize the contents and contribution of your work in a few sentences in English language.

**Keywords** ElateMe, crowdfunding platform, social network, back-end API, RESTful, payments systems

# Contents

# List of Figures

# Introduction

!!!! START !!!!

## ElateMe

ElateMe is a new crowdfunding platform with elements of the social network. Unlike other similar projects like Kickstarter or Patreon that help bring creative, commercial projects to life by means of interested people, ElateMe is focused on fulfillment of personal wishes with the help of user's friends. The user can create a wish and set its cost, title, and a short description. His friends then will be able to contribute to his wish by donating money. When wish gathers needed amount, money will be transferred to the user bank account. The social part of the application is providing an ability for the user to connect with his friends, communicate with other users, rate and comment others' wishes.

## Aim of the thesis

The aim of this thesis is to analyze functional, non-functional requirements and use cases of the project, design database model and server architecture, implement back-end Application Programming Interface (API) and payments system for this service.

## Motivation

The main goal for the author of the thesis is to analyze and learn tools for web back-end development such as Python programming language and Django web framework, practice building complex systems using them, learn to design server architecture and explore various online payment systems.

Chapter 1

# Analysis

This chapter will focus on analysis of the project as a part of a software development that connects customer's requirements to the system and its following design and development.

Analysis of software project is intended to define detailed description of the product, break it down into requirements to the system, their systematization, detection of dependencies, and documentation.

## 1.1 BI-SP1 and BI-SP2 subjects

The work on the ElateMe project started within the framework of the BI-SP1 subject. Our development team devided into groups: Android, iOS and Back-end developers. Our task was to define and document main client's requirements, implement functioning prototypes of mobile applications and back-end server API. During BI-SP1 subject, Maksym Balatsko was working on prototype of back-end server, so choise of used technologies was up to him. Then the technology stack was agreed with supervisor of the project. Chosen technologies will be discussed in the next chapter.

Because of changes in requirements and provided a new interface design of the mobile applications, analysis and its documentation has undergone certain changes. And at the start of BI-SP2 subject implementation of back-end API, on which Maksym and I worked, has started.

## 1.2 Functional requirements

Functional requirements specify the behaviors the product will exhibit under specific conditions. They describe what the developers must implement to enable users to accomplish their task (user requirements), thereby satisfying the business requirements. [1]

### 1.2.1 Authorization

**F1: Sign up via Facebook** User shall be able to sign up to ElateMe application with his Facebook account. Application shall load user's data such as name, surname, email, date of birth, etc.

**F2: Logout** Authorized user shall be able to log out. In this case he shall also stop receiving any notifications from the application.

**F3: Load friends from social network** On initial login application shall load list of user's friends that are already signed up in this application. This users shall be considered as friends in ElateMe application.

### 1.2.2 Friendship management

**F4: View friends list** User shall be able to view list of his Facebook friends that are already signed up in application.

**F5: Create friends group** User shall be able to create friends group. Groups shall be used for simplification of friends management.

**F6: Delete friends group** User shall be able to delete friends group.

### 1.2.3 Wish management

**F7: Create wish** User shall be able to create wish, set it's title, description, price(amount of money that he(user) wants to gather), and deadline.

**F8: Delete wish** User shall be able to delete his wish if nobody shall have donated money yet.

**F9: Close wish** User shall be able to close his wish. Money that will have been gatherd on this wish shall be refunded to donators.

**F10: View users' wishes list** User shall be able to browse wishes lists of his friends.

**F11: Create surprise wish** User shall be able to create surprise wish for one of his friends. In this case user to whom the wish was addressed shall not have acces to it and shall not know about it until the whole amount is collected.

**F12: View contributed wishes list** User shall be able to view list of wishes he will have contributed to.

### 1.2.4 Feed

**F13: View user's feed** User shall recieve feed with latest wishes of his friends.

### 1.2.5 Donation management

**F14: Donate to wish** User shall be able to financially contribute to wishes of his friends.

**F15: Refund** In the case of the closure of the wish, all gatherd money shall be refunded to donators.

### 1.2.6 Comments management

**F16: View wishes comments list** User shall be able to view list of comments under the wish he will be browsing.

**F17: Comment wish** User shall be able to leave a comment under the wish.

**F18: Delete comment** User shall be able to delete his comment.

## 1.3 Non-functional requirement

Non-functional requirement is a software requirement that describes not what the software will do, but how the software will do it, for example, software performance requirements, software external interface requirements, software design constraints and software quality attributes. Non-functional requirements are difficult to test; therefore they are usually evaluated subjectively.[2]

### 1.3.1 Back-end API

**N1: Representational State Transfer (REST)ful** Back-end API shall follow architectural constraints of REST architectural style.

**N2: HyperText Transfer Protocol Secure (HTTPS)** Server shall comunicate with client via HTTPS.

**N3: PostgreSQL database** PostgreSQL shall be used as the main DBMS.

**N4: Performance** Server shall be able to serve 1500 requests per second.

### 1.3.2 Payments

**N5: FIO-bank** User shall be able to make payments via FIO-bank.

**N6: Bitcoin** User shall be able to make payments via Bitcoin.

**N7: Secure payments** System shall ensure secure payments.

**N8: Consistency** Servers data about payments shall be consistent with data in payments systems (FIO-bank, Bitcoin, etc.). System shall react accordingly to errors appeared during payments.

## 1.4 Use cases

Use cases were defined after analising of functional and non-functional requirements. Use cases documentation serves for better undestending of functionality required from the system. Use cases model of this application is presented on the diagram 1.1.

Use case model includes the following components:

- **Actors** represent people and external systems involved in a particular case cases. The diagram focuses specifically on the actors to insure that the system provides useful and usable functionality.

  The main actors of this system are:

  - **User** that uses mobile or web application.
  - **Facebook** participates in user authorization and provides an interface for obtaining the necessary information about him (user).
  - **Payment system** participates in payments, namely donations and refundances. In the role of payment systems in this application are Fio-Banka and Bitcoin.

- **Use cases** represent the functionality that the system provides for actors. In this diagram they are divided into logical groups namely

  - **Authentication** of the user through Facebook and obtaining the necessary information about the user from his Facebook account.
  - **Friendship** between users and the division of friends into groups.
  - **Wish management** includes the creation of wishes, donations and comments, as well as closure of the wishes with subsequent refandation, etc.
  - **Other** includes the news feed and notifications.

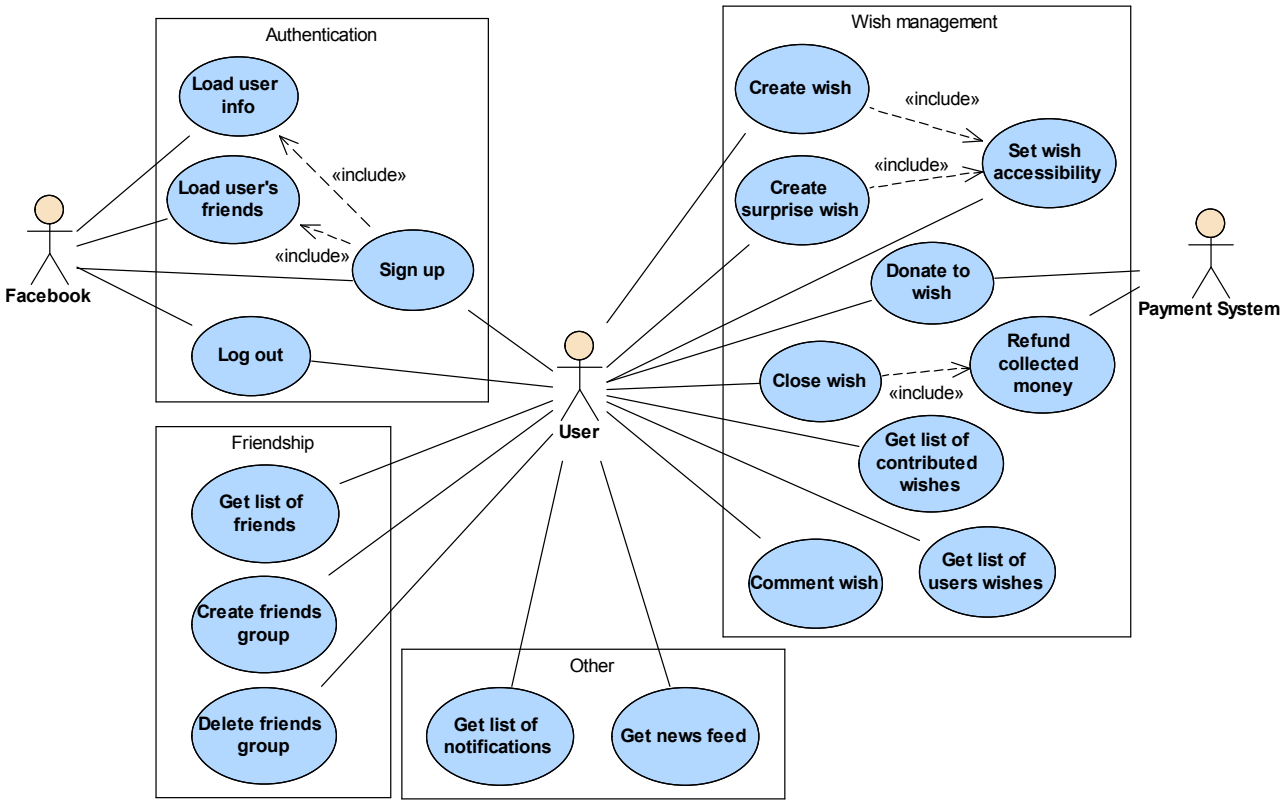



Figure 1.1: Use cases diagram

## 1.5 Domain model

** insert Domain model diagram **

## 1.6 System structure

The whole ElateMe application system is devided into components. Main components are server, Android and iOS clients.

Detailed structure of server and its connection with external interfaces are presented at the component diagram 1.2. As seen in the diagram, server provides interface for mobile applications to communicat via REST API. Server also uses interfaces of Facebook (Graph API) to recieve needed data about users and interfaces of payment systems (FIO-bank and Bitcoin) for payments processing.

Inside the server is divided into components that are responsible for storing and processing data of application entities. This components are called *apps* in Django. Apps communicate with database via Django *models*. Models in Django is an interface designed to simplify querying to database.

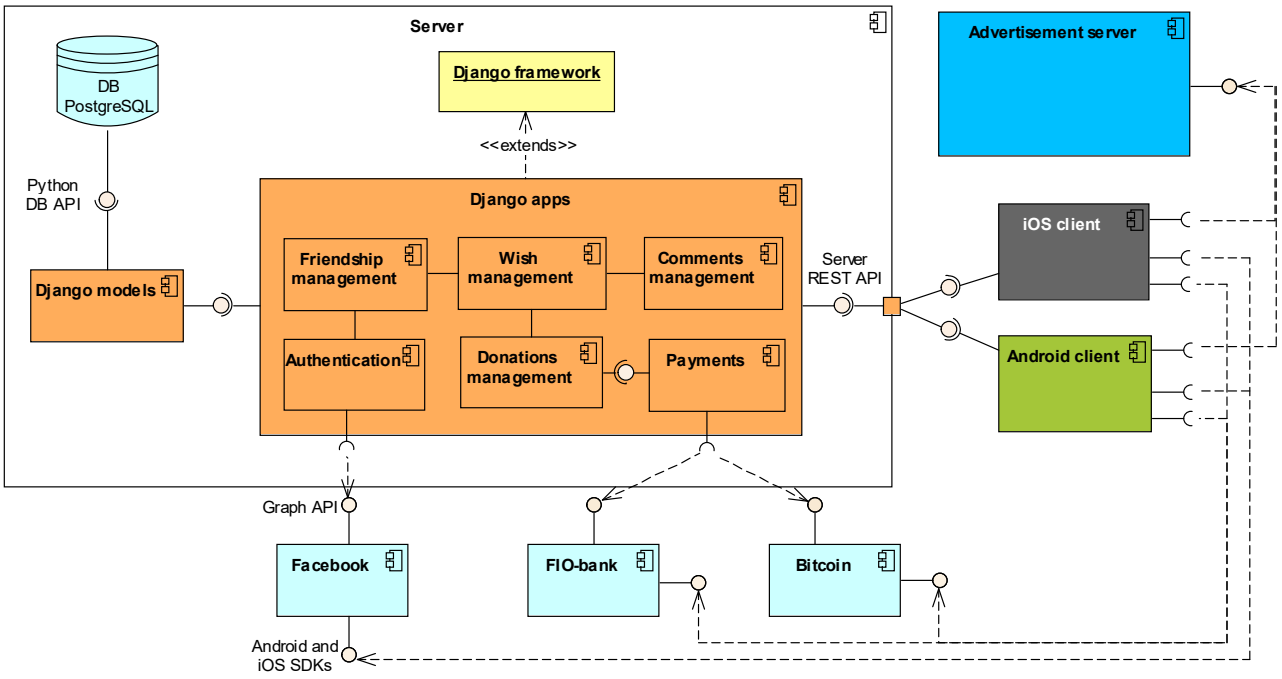



Figure 1.2: Component diagram

The diagram also shows the use of interfaces of Facebook and payment systems by mobile clients, but they are not a part of my work, so their design and implementation will not be described in this thesis.

## 1.7 Authentication

User has to be authorized to use the application. ElateMe application will not provide in-app registration. User authentication will be conducted exclusively through third-party systems. It is made to simplify the registration in the application.

### 1.7.1 Facebook

User authentication will be conducted through his Facebook account.

After first login, application will get from Facebook needed information about the user: first name, last name, email address, list of user's friends. User's Facebook friends who are already logged in to the application, automatically become his friends in the ElateMe.

Despite the lack of in-app registration, user's information recieved from Facebook will be stored in ElateMe system as well, because user will be able to add other users to his friend list, create friends groups independently from Facebook.

## 1.8 Payments system

The ElateMe project is based on crowdfunding. So application (and the server in particular) has to provide service for payments processing. According to the requirements, this service should use interfaces of FIO-banka and Bitcoin.

### 1.8.1 Use cases

In this application, the payment system participates in the following use cases:

- **Donation**

  During the donation, the money is transferred to the internal account of ElateMe, where it is stored until one of the following use cases.

- **Wish completion**

  In case of fulfillment of the wish, money is transferred to the account of the author of this wish.

- **Wish closing**

  There are two situations in which the wish is closed: closing of wish by its author and closing upon expiration of deadline. In both cases, already collected money is returned to the donators' accounts.

### 1.8.2 FIO-banka

Since the open API of the FIO-banka [3] does not provide sufficient functionality for the needs of this application, it was necessary to agree directly with bank on the provision of an needed interface. Due to prolonged communication with the bank, we still did not get the full documentation. We only got a description of the payment process, which can be seen on the diagram 1.3.

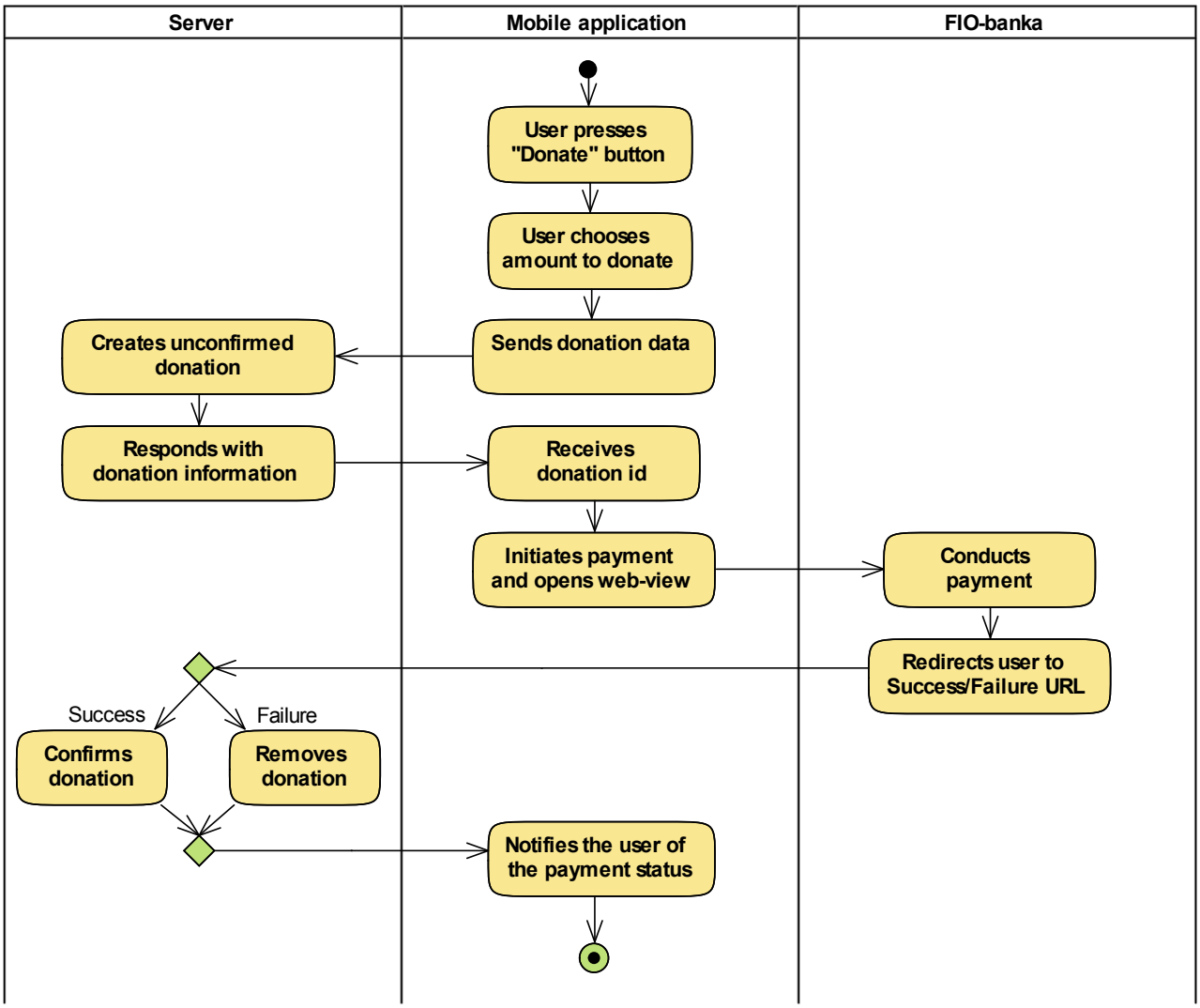



Figure 1.3: Payment via FIO-banka

As seen on the diagram, before initiating the payment, the mobile application will send a request to the server to create a donation, the server will create an unconfirmed donation and respond to the application with information about the newly created donation. From this information, the application will take the donation id, which will later serve as the identifier of the paid product. The application initializes the payment with the necessary information about the user and donation and redirects the user to the payment gateway of the FIO-banka in the web-view. After processing the payment, the FIO-Banka redirects the user to one of the two urls on the server, depending on the state of the payment (success/fail urls). In case of success, the server marks the payment as confirmed, alternatively not successful. The

mobile application will close the web-view and notify the user about the status of the payment.

### 1.8.3 Bitcoin

The second way to make payments in the application is Bitcoin.

Bitcoin is a collection of concepts and technologies that form the basis of a digital money ecosystem. Units of currency called bitcoins are used to store and transmit value among participants in the bitcoin network. Bitcoin users communicate with each other using the bitcoin protocol primarily via the Internet, although other transport networks can also be used. The bitcoin protocol stack, available as open source software, can be run on wide range of computing devices, including laptops and smartphones, making the technology easily accessible [4]. A detailed analysis of the system of Bitcoins was carried out in the bachelor work [5] of Yegor Terokhin, one of the iOS developers of ElateMe. In his work he describes in detail the principles of work of Bitcoins and the benefits of using this system in the framework of the ElateMe project.

To make payments using Bitcoins, it was decided to use Coinbase. Founded in June of 2012, Coinbase is a digital currency wallet and platform where merchants and consumers can transact with new digital currencies like bitcoin and ethereum [6]. The main benefit of using Coinbase is that it provides Software Development Kit (SDK) for both mobile platforms and a python used for the implementation of the backend in this project.

### 1.8.4 Refund mechanism

## 1.9 Push notifications

Push notifications are small important messages from the application or service, displayed by the operating system when the user does not directly work with the specified application or service. The advantage of such notifications is that there is no need to keep the program in memory, spending on it processor powers and memory.

In ElateMe application the user will receive information about the state of his wishes, new donations, comments, etc.

### 1.9.1 Mechanism of push notifications

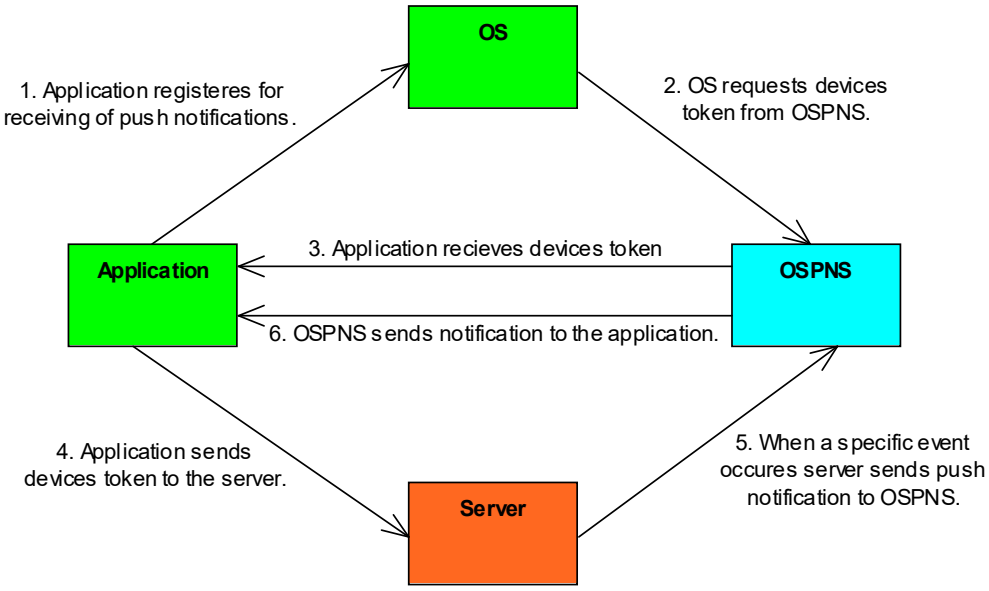



Figure 1.4: Mechanism of push notifications

For server to be able to send push notifications it needs to store a token. Token is a line of characters that serves as an address of specific application on the specific device. Token is generated by Operating system push notification service (OSPNS). After application is installed on the device it registers itself for receiving of push notifications, OS requests token from OSPNS, application receives token and sends it to the server.

### 1.9.2 Actors

- **OSPNS**

  Every operating system has its own service for processing push notifications. They are Google Cloud Messaging (GCM) for Android and Apple Push Notification service (APNs) for iOS. As shown on the diagram 1.4 OSPNS sends a token to the application when it registers in the service and sends the push notifications to the application itself.

- **Server**

  The server stores the tokens of each individual device and sends the push notifications to OSPNS.

- **Client application**

The application is registered to receive push notification, receives a token from OSPNS and sends it to the server.

Chapter 2

# Design

## 2.1 Authentication

As was mentioned earlier in this thesis, authentication of the user will be conducted through his Facebook account. Facebook provides interface for user authentication in third-party applications. This interface uses OAuth 2.0 protocol.

### 2.1.1 OAuth 2.0

OAuth 2.0 is the industry-standard protocol for authorization. OAuth 2.0 supersedes the work done on the original OAuth protocol created in 2006. OAuth 2.0 focuses on client developer simplicity while providing specific authorization flows for web applications, desktop applications, mobile phones, and living room devices. [7]

For server to be able to get list of friends and other information about user, mobile application needs to recieve token from Facebook with appropriate permissions and send it to the server. Token is a line generated by Facebook and by which Facebook provides access to certain data of certain user.

Diagram 2.1 shows mechanism of successful authentication via user's Facebook account.

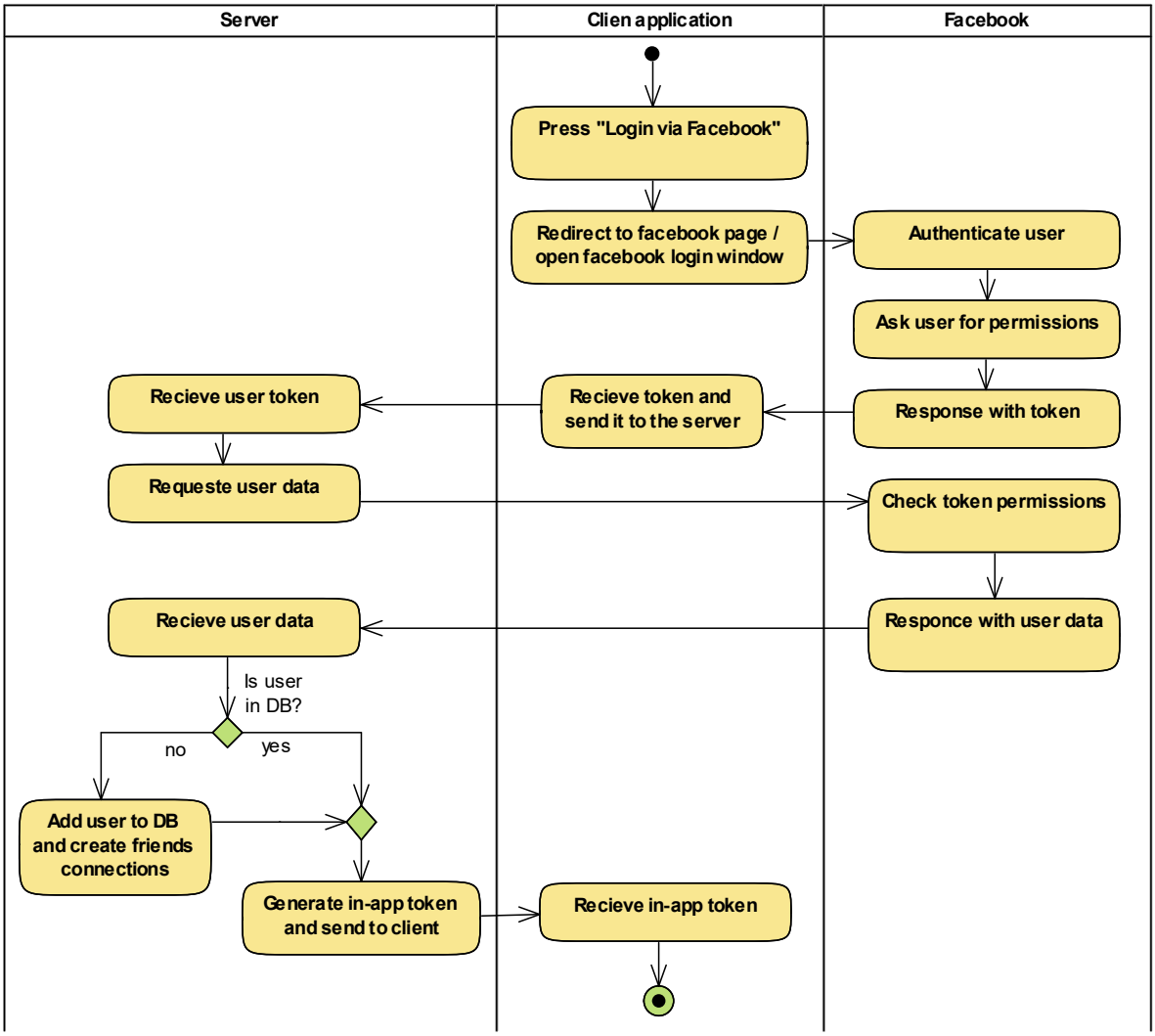



Figure 2.1: Authentication activity diagram

## 2.2 REST API

Server API will be built on the basis of REST. REST is the architectural solution for the transfer of structural data between server and client [8]. API is considered RESTful if it follows certain rules [9]:

- **Client-Server**

  Client-Server defines a clear separation between a service and its consumers. Service (in this case server) offers one or more capabilities and listens for requests on these capabilities. A consumer (in this case mobile

client) invokes a capability by sending the corresponding request message, and the service either rejects the request or performs the requested task before sending a response message back to the consumer.

- **Stateless**

  Statelessness ensures that each service consumer request can be treated independently by the service. The communication between service consumer (client) and service (server) must be stateless between requests. This means that each request from a service consumer should contain all the necessary information for the service to understand the meaning of the request, and all session state data should then be returned to the service consumer at the end of each request.

- **Cache**

  Responses may be cached by the consumer to avoid resubmitting the same requests to the service. Response messages are explicitly labeled as cacheable or non-cacheable. This way, the service and/or the consumer can cache the response for reuse in later requests.

- **Uniform Interface**

  All services and service consumers within a REST-compliant architecture must share a single, overarching technical interface. As the primary constraint that distinguishes REST from other architecture types, Interface is generally applied using the methods and media types provided by HTTP.

- **Layered System**

  A REST-based solution can be comprised of multiple architectural layers, and no one layer can "see past" the next. Layers can be added, removed, modified, or reordered in response to how the solution needs to evolve.

There is also an optional constraint **Code-On-Demand**. This constraint states that client application can be extended if they are allowed to download and execute scripts or plug-ins that support the media type provided by the server. Adherence to this constraint is therefore determined by client rather than the API [8].

### 2.2.1 Apiary

The apiary service will be used for the server API documentation. It is a powerful API design stack [10]. In the apiary, the Blueprint API is used to describe the structure of the APIs. API Blueprint is a powerful high-level API description language for web APIs [11].

## 2.3 Chosen technologies


As I mentioned before, the choice of used technology was not up to me so in this section I will not describe why certain technologies were chosen, but will describe their advantages (alternatively disadvantages) for this project.

### 2.3.1 Python

Python is a base of the server. It was chosen as a primary programming language because it was designed to be simple and highly readable which is very important for large-scale projects. Its syntax and standard library simplify and speed up a development.

### 2.3.2 Django

Django is an open source web framework for python. It provides a high level abstraction of common web development patterns. It follows Model-View-Controller (MVC) design pattern. Django uses MVC to separate model as a data and a business logic of the application, view as a representation of the information for the user, in this case, the client side of the application and controller as an interface of the application, in this case, set of URLs to communicate with front-end [12].

### 2.3.3 Django REST

Django REST framework is an open source project built on Django framework. It contains needed tools for implementation of RESTful API such as serializers, pagination, permissions, etc.

### 2.3.4 PostgreSQL and SQLite

On initial stage of the development, SQLite will be used as a DataBase Management System (DBMS), because it does not require a standalone database server and is simple to set up. The database will be changed and migrated to PostgreSQL later.

PostgreSQL is powerful, open source relational DBMS. It has advanced features such as full atomicity, consistency, isolation, durability [13]. Django framework provides great API for working with PostgreSQL databases.

### 2.3.5 nginx

nginx [engine x] is an HTTP and reverse proxy server, a mail proxy server, and a generic TCP/UDP proxy server, originally written by Igor Sysoev [14]. According to Netcraft [15], nginx served or proxied 28.50% busiest sites in March 2017.

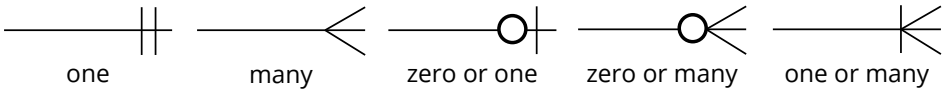
## 2.4 Database model

One of the main parts of the web server is the database. Before the implementation of the physical database model, a database design is created, the data model.

A data model is a combination of three components [16]:

- **The structural part**: A collection of data structures (or entity types, or object types) which define the set of allowable databases.
- **The integrity part**: A collection of general integrity constraints, which specify the set of consistent databases or the set of allowable changes to a database.
- **The manipulative part**: A collection of operators or inference rules, which can be applied to an allowable database in any required combination in order to query and update parts of the database.

Thus, the structure of the ElateMe database was defined and documented. Full datatabase model is in the attachments. It is not included in the text of this thesis because of its large volume. The documentation is divided into logical parts containing the corresponding database tables, their columns, and connections. Diagrams include the following connections.




This documentation, however, is not an accurate representation of the physical database, since django models are used to work with database, which themselves create tables and connections between them.

## 2.5 Class model

Class model of the project was built based on Django project structure. Most of the classes extend Django classes following certain rules and format. Therefore I will describe only parts that don't depend on Django structure. A complete model of classes can be found in attachments.

### 2.5.1 Authentication

As I said before, in frameworks of this work user authentication will be conducted through his Facebook account. But in the future, other social networks and/or authorization methods may be added. Since most of social networks support OAuth 2.0 [7], this allows you to make a universal interface for user authorization, that will process user authentication via social network using token provided by client (mobile or web application).

Thus, as shown in the diagram 2.2, classes that extend *AbstractSocialAPI* will provide interface for token processing. *AbstractSocialAPI* defines method *process* that receives token from client, requests information about the user from the corresponding social network, decides if user is already registered in the application, registers him (adds to database) if necessary, and authorizes user in application. Information about user is obtained using abstract methods like *request_data*, *get_social_id*, *get_friends*, etc.

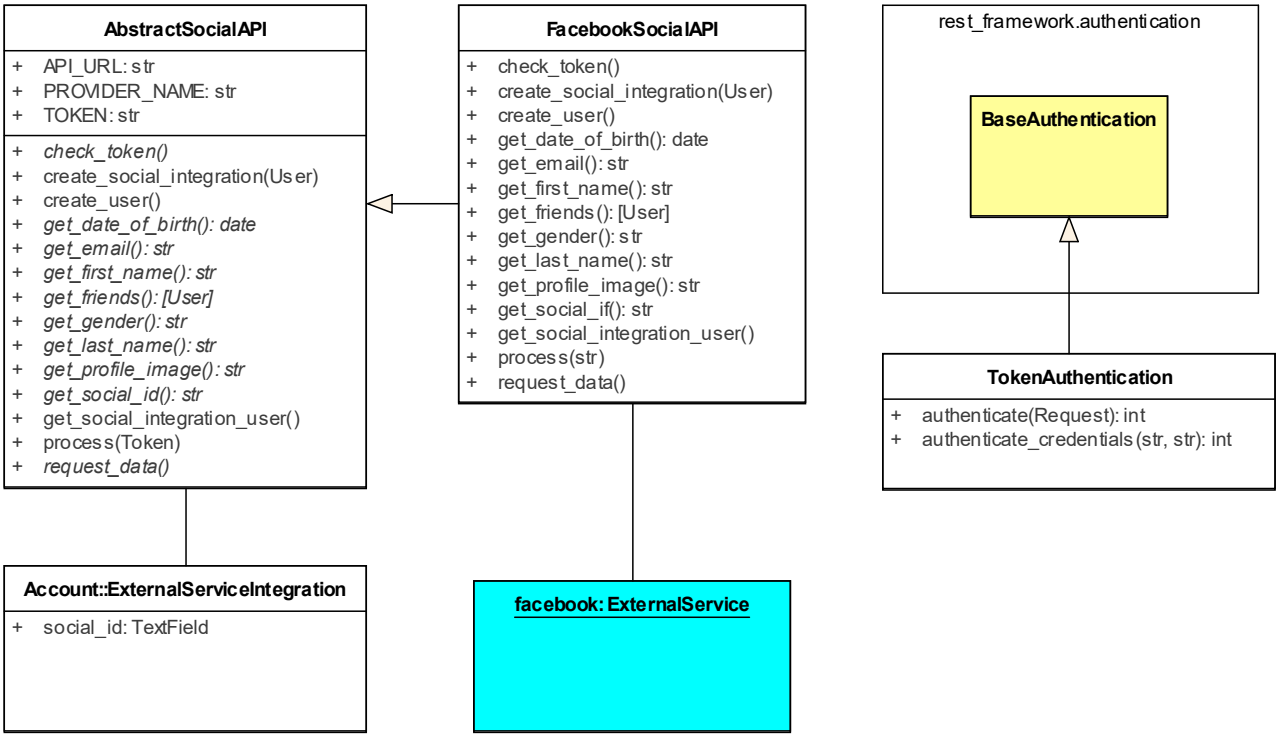



Figure 2.2: Social integration class model

### 2.5.2 Push notifications

As shown in the diagram 2.3, the *EventHandler* class will be responsible for processing of events which require user notification. This class uses interface of *AbstractNotificationService* for sending of push notifications to the correspond-

ing OSPNSs. Since the user can have several devices with the installed application, there may be situations when one notification will be sent to several tokens and/or different OSPNSs. Classes that extend *AbstractNotificationService* and implement method *notify* will be responsible for sending push notifications to the specific OSPNSs.

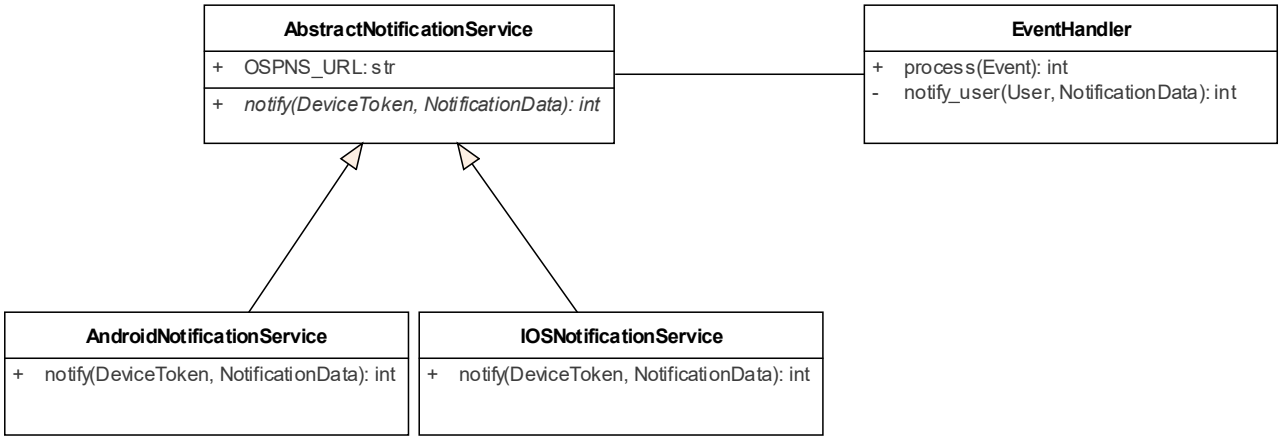



Figure 2.3: Push notification class model

### 2.5.3 Payments

Payment information will be stored in the models that are inherited from the *Payment* model. At the moment these classes are *FIOBankaPayment* and *BitcoinPayment*. They store information for conducting and refunding payments needed for interfaces of FIO-banka and Bitcoins respectively. Payment processing will be performed by *PaymentsHandler*s. They provide interfaces for the processing of donations, refunations and payments for the completed wishes. For each *Payment* model there will be a corresponding *PaymentsHandler*. Which means if a new payment system arrives, it will be enough to override the *Payments* class with the necessary information for this payment system and implement the *PaymentsHandler* interface for this particular system.

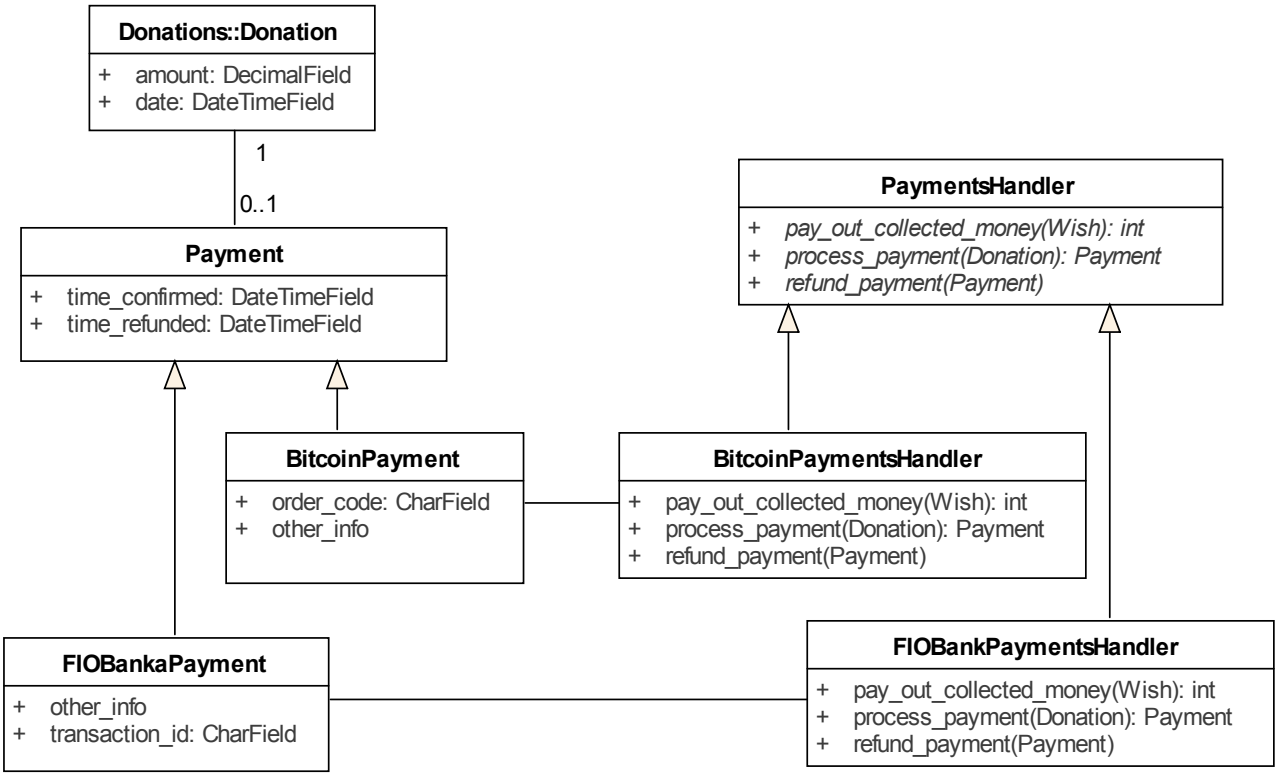



Figure 2.4: Payments class model

## 2.6 Deployment model

Chapter 3

# Implementation

This chapter contains a description of the implementation of the project's server side. This chapter will describe the structure of the project, the problems arose during implementation and ways to solve them. There will also be described guide for the installation, launch and deployment of this project.

This chapter is intended to familiarize the reader with the implementation of this application and to simplify the understanding of the structure of the project for future developers.

## 3.1 Project structure

Django as a framework determines the structure of the whole system. Django project is divided into logical parts - apps. Apps contain a set of modules with classes, that implement interfaces and extend classes, that are provided by Django.

Later in this section, the main modules of the apps in the Django project will be described. The parts involved in processing of the request for the receiving of the user's wishes, will be taken as an example.

### 3.1.1 Django models

Django models is an interface for simplified querying to database. All models extend class *model* from *django.db* module and usually represent single table in a database [17].

Thus, each app except the feed and notifications contains a module *models*. In this module there are models that completely describe the database. Despite what kind of DBMS is used (PostgreSQL or SQLite) the Django models and quering through them does not change, which simplifies development, testing and deploy.

Model that represent table *Wish* in the database:

```
class Wish(models.Model):
```

```
    title = models.CharField(max_length=100)
    description = models.CharField(max_length=512)
    amount_needed = models.FloatField()
    date_created = models.DateTimeField(auto_now_add=True)
    date_of_expiration = models.DateTimeField(null=True)
    date_completed = models.DateTimeField(null=True)
    is_public = models.BooleanField(default=False)
    author = models.ForeignKey(User,
                               on_delete=models.CASCADE,
                               related_name='wishes')

    class Meta:
        db_table = 'Wish'
```

### 3.1.2 Django views

Django view is a method that is called during request on certain Uniform Resource Locator (URL). This function takes a Web request and returns a Web response, in this case JSON. The main logic of processing requests is in the views.

Before we started using Django REST framework, the request on receiving the wish list of the current user looked like this:

```
def current_user_wishes_view(request):
    current_user = request.user
    if not user.is_authenticated():
        return HttpResponse('Unauthorized', status=401)

    current_user_wishes = current.user.wishes. \
                                    order_by('-date_created')

    response_data = []
    for wish in current_user_wishes:
        serialized_wish = WishSerializer(wish).data
        response_data.append(serialized_wish)

    paginated_response = WishPagination(). \
                get_paginated_response(request, response_data)

    return JsonResponse(paginated_response)
```

Similar requests to obtain a list of objects of a certain model (wishes, donations, comments, etc.) look very similar. It is checked if the user is authenticated, the data queryset is obtained, the data is serialized, paginated (divided into pages), returned in the JavaScript Object Notation (JSON) format. To simplify the implementation of such requests and the corresponding auxiliary classes (serializers, paginations, etc.), it was decided to use the Django REST framework.

Thus, in Django REST framework view on getting the current user's wish list looks like this:

```
class CurrentUserWishesView(generics.ListCreateAPIView):
    renderer_classes = (renderers.JSONRenderer,)
    permission_classes = (permissions.IsAuthenticated,)
    serializer_class = serializers.WishSerializer
    pagination_class = pagination.WishPagination

    def get_queryset(self):
        user = self.request.user
        return user.wishes.order_by('-date_created')
```

This approach simplifies implementation and improves the readability of the code.

### 3.1.3 Django urls

The *urls* module in Django is responsible for linking the URL endpoints to their corresponding views. It contains a list of objects *url*. In *wishes* app it looks like this:

```
urlpatterns = [
    url(r'wishes/', CurrentUserWishesView.as_view()),
    # other urls
]
```

### 3.1.4 Django REST serializers

Django REST serializers is an interface that provides the Django REST framework for simplifying the serialization and desirilization of instances of Django models. The simplest wish serializer looks like this:

```
class WishSerializer(serializers.ModelSerializer):
    class Meta:
        model = Wish
        fields = '__all__'
        read_only_fields = ('id', 'author', 'date_created',
                            'date_completed', 'amount_gathered',
                            'donators_count')
```

### 3.1.5 Django REST pagination

To avoid large responses in the case of a large number of objects in the queryset, it was decided to use pagination. Pagination is the partitioning of the response into so-called pages of the same size. To create a class responsible for the pagination of the list of data, it is enough to extend the *PageNumberPagination* class from the module *rest_framework.pagination*:

```
class WishPagination(PageNumberPagination):
    page_size = 10
    page_size_query_param = 'page_size'
    max_page_size = 50
```

If this class is used as *pagination_class* in the view, *page_size* and *page* are used in the URL as optional parameters. So on the request "/wishes?page_size=5&page=2" server will respond with JSON in following format:

```
{
  "count": 13,
  "next": "https://api.elateme.com/wishes?page_size=5&page=3",
  "previous": "https://api.elateme.com/wishes?page_size=5&page=1",
  "results": [
    # 5 serialized wishes from the second page
  ]
}
```

### 3.1.6 Django apps

The project was divided into the following apps:

- **account**. This app includes modules for storing and processing information about the user. *account* is divided into sub-applications *authorization* and *social* that are responsible for user authorization and integration with social networks respectively.
- **donations**. This app is designed to process donations. It will also contain the logic of the payment and refund systems.
- **feed**. App for the arrangement of a user's news feed.
- **friendship**. Application for the processing of friendly relationships between users.
- **notifications**. App provids user notifications. At the moment, it provides the REST interface for getting news list. Later this application will work with push-notifications.
- **wishes**. Application provids interface for processing user wishes. It also contains sub-application *comments*.

### 3.1.7 Django settings?

## 3.2 Integration with Facebook

## 3.3 Python Virtual Environment

Since this application uses a set of dependencies that don't come as part of the python standard library they must be installed for all instances of the application, namely the development and testing on the local machines of developers and on the production server. Python Virtual Environment was used for these purposes.

A Virtual Environment is a tool to keep the dependencies required by different projects in separate places, by creating virtual Python environments for them. It solves the "Project X depends on version 1.x but, Project Y needs 4.x" dilemma, and keeps your global site-packages directory clean and manageable.[18]

A list of all the dependencies that are installed in the virtual environment can be found in the *requirements.txt* file in the project's root directory. The guide for installing and configuring the virtual environment is described in the installation guide in the attachments.

## 3.4 Payments

Chapter 4

# Testing

## 4.1 Unit tests

Alongside with the development automatic testing was performed using unit tests. Unit tests are designed to verify the correct functioning of the parts of the application.

Unit testing code means validation or performing the sanity check of code. Sanity check is a basic test to quickly evaluate whether the result of calculation can possibly be true. It is a simple check to see whether the produced material is coherent. [19]

### 4.1.1 Django REST tests

For testing, native tools were used, namely the Django REST framework tests. Similar to java JUnit tests Django tests are class-based. Every test case is a method of the class, that extends *APITestCase* from the module *rest_framework.test*. Classes can also contain the following methods:

- setUp: Method called to prepare the test fixture.
- tearDown: Method called immediately after the test method has been called and the result recorded.

For testing, Django creates a separate empty database independent of the main database. Sqlite DBMS is used for testing in this project.

### 4.1.2 Auxiliary methods

For testing of the server API, I wrote a set of auxiliary methods that simulate HTTP requests to the server. This methods take URL to which the requeste is sent and optionaly information (JSON) which is sent as the body of the request. Methods use *APIRequestFactory* to perform requests to the server.

Methods also use *force_authenticate* function that allows to authenticate user (in this case test user) in the system without involvement of Facebook. This function is used for testing of requests that require authorization.

### 4.1.3 Test cases

As an example of a test, I'll take the creation of the wish by the user.

Initially in the method *setUp* I create test user, after that I make a POST request to the server with information about the wish in the body of the request. After the server responded, I check status code of the response, compare the information between the body of the request and the body of the response (body of the response contains the newly created wish) and check that wish is added to the database.

```
from django.urls import reverse
from rest_framework.test import APITestCase
from rest_framework import status
from account.models import User, UserManager
from wishes.models import Wish

# auxiliary methods for http requests
from util.test_requests import post, get, put, patch, delete

class WishesTest(APITestCase):

  def setUp(self):
    self.url = reverse('wishes:wishes')
    self.user = UserManager().create_user('test1@test.com', 'test')

  def test_create_wish(self):
    wish_data = {
      'title': "iPhone7",
      'description': "I don't need no jack",
      'amount': 19999
    }
    status_code, response_data = post(url=self.url,
                                      user=self.user,
                                      data=wish_data)

    self.assertEqual(status_code, status.HTTP_201_CREATED)
    self.assertEqual(response_data['title'], wish_data['title'])
    self.assertEqual(response_data['amount'], wish_data['amount'])
    self.assertEqual(Wish.objects.get().title, wish_data['title'])
```

This is a positive test, so the status code must be 201 (created), wish should be created and added to the database.

## 4.2 Apachebench

Apachebench tool was used to test server performance. Apachebench is a open source, single-threaded command line program for benchmarking a web server.

Tests were conducted on different URLs, with different methods including GET and POST. Example of a testing will be GET request on "wishes/" URL, that returns list of wishes of the current user. It is one of the most popular requests. During the GET request on this URL, the server makes one SELECT-WHERE request to the database.

Testing command looks like this:

```
HEADERS=(
        "Authorization: Token b0edca023c283518f20b368947085bc4a82da8a8" \
        "User-Agent: test-agent"\
        )
URL="https://api.elateme.com/wishes/"

curl -sL -w "%{http_code}" "${HEADERS[@]/#/-H}" "$URL" -o /dev/null

ab -c 100 -n 5000 "${HEADERS[@]/#/-H}" "$URL"
```

Before testing itself, it is checked, with the *curl* utility, if the headers and URL are valid and it is possible to get a successfull responce with them.

In this case, *curl* should print "200", which means a successful request. Further testing with the same headers and URL is conducted. The ab (Apachebench) utility offers two main flags:

**-c** Number of multiple requests to perform at a time.

**-n** Number of requests to perform for the benchmarking session.

This test sends 5000 requests to the server with 100 simultaneous connections.

After testing, ab writes out the statistics, which includes time taken for tests, requests per second, average per request, etc. The main analyzed indicator was "requests per second".

### 4.2.1 Testing results and optimisation

After the first test, the request per second rate was about 25, which is a very low rate.

To optimize the performance of the server, it is necessary to find a bottleneck point. There are several possible problematic places:

- **Database**. Slow connection, long requests processing.

- **Django**. Unsuitable Django configuration.
- **Gunicorn**. Unsuitable configuration of Gunikorn, wrong number of workers, slow logging, etc.
- **Nginx**. Incorrect proxy configuration, wrong number of workers, logging, caching, static files, etc.
- **Hardware**. Low hardware performance.

To find a problematic place, it is necessary to test each of the above-mentioned parts separately.

### 4.2.2 Database test

Database testing is quite simple: sending a large number of requests and timing duration of execution. This was done directly through Django to test all the parts involved in connecting to the database at once (Django, python, PostgreSQL).

The test looks like this:

```
def test_db(requests_per_user):
    start = datetime.now()
    users = User.objects.all()
    for i in range(requests_per_user):
        for u in users:
            wishes = u.wishes.all()
    time = (datetime.now() - start).total_seconds()
    total_requests = requests_per_user * users.count()
    print(total_requests, 'requests_per', time, 'seconds')
    print(total_requests/time, 'req/sec')

test_db(1000)
```

The test checks how long it takes to get each user's wishes separately from the database 1000 times. At the time of testing, 23 users were stored in the database with 5 to 30 wishes each.

Output of the test:

```
23000 requests per 7.23 seconds
3178.34 req/sec
```

As seen, the database is capable of serving more than three thousand requests per second, so the problem is not in it.

### 4.2.3 Django test

To test Django separately from nginx (without a proxy), it was enough to run the Apachebench locally on the port on which Django server is running:

```
URL="127.0.0.1:8888/wishes/"
ab -c 20 -n 1500 "${HEADERS[@]/#/-H}" "$URL"
```

This test showed that one instance of the Django server itself serves about 11 requests per second. This indicates that the problem is in Django.

Unfortunately, solutions to this problem was not found. With the help of changing server configurations, optimizing Gunicorn and nginx, it was possible to increase the performance almost twice, up to 50 requests per second. But the main problem was not solved.

# Conclusion

# Bibliography


[1] Wiegers, K.; Beatty, J. *Software Requirements*. Best practices, Microsoft Press, 2013, ISBN 9780735679665. Available from: `https://books.google.cz/books?id=40lDmAEACAAJ`

[2] Chung, L.; Nixon, B.; et al. *Non-Functional Requirements in Software Engineering*. International Series in Software Engineering, Springer US, 2012, ISBN 9781461552697. Available from: `https://books.google.cz/books?id=MNrcBwAAQBAJ`

[3] Reference. FIO API BANKOVNICTVÍ [online]. [Cited 2017-04-28]. Available from: `https://www.fio.cz/docs/cz/API_Bankovnictvi.pdf`

[4] Antonopoulos, A. *Mastering Bitcoin*. O'Reilly Media, Incorporated, 2014, ISBN 9781449374044.

[5] Terokhin, Y. *ElateMe - iOS klient I: bakalářská práce.* Praha: České vysoké učení technické v Praze, Fakulta informačních technologií, 2017.

[6] Reference. ABOUT COINBASE [online]. [Cited 2017-05-7]. Available from: `https://www.coinbase.com/about`

[7] Reference. OAuth 2.0 [online]. [Cited 2017-04-02]. Available from: `https://oauth.net/2/`

[8] DAIGNEAU, Robert. *Service design patterns: fundamental design solutions for SOAP/WSDL and RESTful web services*. Upper Saddle River: Addison-Wesley, c2012. Addison-Wesley signature series, 2012, ISBN 978-0-321-54420-9.

[9] Reference. What is REST? [online]. [Cited 2017-04-02]. Available from: `http://whatisrest.com/`

[10] Reference. Apiary [online]. [Cited 2017-04-22]. Available from: `https://apiary.io/`

[11] Reference. API Blueprint [online]. [Cited 2017-04-22]. Available from: https://apiblueprint.org/

[12] HOLOVATY, Adrian and Jacob. KAPLAN-MOSS. *The definitive guide to Django: Web development done right. 2nd ed.* Berkeley: Apress, c2009. Expert's voice in Web development., 2009, ISBN 978-1-4302-1936-1.

[13] OBE, Regina Obe and Leo Hsu. *PostgreSQL: up and running.* Sebastopol: O'Reilly, c2012, 2012, ISBN 978-1-449-32633-3.

[14] Reference. nginx [online]. [Cited 2017-04-08]. Available from: https://nginx.org/en/

[15] Reference. Netcraft [online]. [Cited 2017-04-08]. Available from: https://news.netcraft.com/archives/2017/03/24/march-2017-web-server-survey.html

[16] Levene, M.; Loizou, G. *A Guided Tour of Relational Databases and Beyond.* Springer London, 2012, ISBN 9780857293497.

[17] Reference. Django documentation [online]. [Cited 2017-04-18]. Available from: https://docs.djangoproject.com/

[18] Reference. Virtual Environments [online]. [Cited 2017-05-05]. Available from: http://python-guide-pt-br.readthedocs.io/en/latest/dev/virtualenvs/

[19] ACHARYA, Sujoy. *Mastering unit testing using Mockito and JUnit: an advanced guide to mastering unit testing using Mockito and JUnit.* Birmingham, England: Packt Publishing, 2014. Community experience distilled., 2014, ISBN 978-1-78398-250-9.

Appendix A

# Acronyms

**API** Application Programming Interface

**MVC** Model-View-Controller

**SDK** Software Development Kit

**HTTPS** HyperText Transfer Protocol Secure

**URL** Uniform Resource Locator

**DBMS** DataBase Management System

**REST** Representational State Transfer

**OSPNS** Operating system push notification service

**GCM** Google Cloud Messaging

**APNs** Apple Push Notification service

**JSON** JavaScript Object Notation

Appendix B

# Contents of enclosed CD

- `readme.txt` ..... the file with CD contents description
- `exe` ..... the directory with executables
- `src` ..... the directory of source codes
  - `wbdcm` ..... implementation sources
  - `thesis` ..... the directory of LaTeX source codes of the thesis
- `text` ..... the thesis text directory
  - `thesis.pdf` ..... the thesis text in PDF format
  - `thesis.ps` ..... the thesis text in PS format