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

16th May 2017

Acknowledgements

I would like to thank the supervisor of the work, Ing. Jiří Chludil for help in writing of this thesis, valuable advice and suggestions for improvement, and Michal Maněna, supervisor of ElateMe project, for managing the practical part of the work. I would also like to thank development team of the ElateMe project namely Georgii Solovev, Maksym Balatsko, Yegor Terokhin and Gleb Arkhipov.

I want to express my special gratitude to my family, especially my parents, for their great support and help throughout the whole study and writing of this work.

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 16th 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

ElateMe je nová crowdfundingová platforma s elementy sociální sítě. Na rozdíl od jiných podobných projektů, jako jsou Kickstarter nebo Patreon, které podporují vývoj kreativních a komerčních projektů prostřednictvím zájemců, ElateMe je zaměřen na naplnění osobních přání s pomocí přátel uživatelů. V aplikaci ElateMe může uživatel sdílet své přání a jeho přátelé mu mohou pomoci tím, že finančně přispějí. Vývoj této platformy je týmovým projektem. Práce je rozdělena do vývoje Android a iOS aplikací, backend REST API a reklamního serveru.

Tato bakalářská práce se zaměřuje na vývoj backend REST API pro výše uvedený projekt. Cílem této práce bylo definovat a zdokumentovat funkční a nefunkční požadavky pro systém ve vývoji, analyzovat případy užití a celkovou strukturu projektu. Autor také analyzuje externí systémy používané touto aplikací, jmenovitě Facebook Graph API, rozhraní online platebních systémů, jako jsou FIO-banka a Bitcoin, služby push notifikace Apple(APNs) a Google(GCM). V rámci této práce byla navržena struktura databáze a architektura serverové aplikace a bylo realizováno backend rozhraní pro komunikaci s mobilními a webovými aplikacemi pomocí populárních nástrojů pro vývoj web serverů, jako jsou programovací jazyk Python, Django web framework a PostgreSQL DBMS. Nakonec aplikace byla podrobena jednotkovým (unit) a výkonovým testům.

Klíčová slova ElateMe, crowdfundingová platforma, sociální síť, backend API, RESTful, online platební systémy

Abstract

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 focuses on the fulfillment of personal wishes with the help of user's friends. In ElateMe application, the user can share his wish, and his friends can help him by contributing financially. The development of this platform is a team project. The work is divided into the development of Android and iOS applications, REST API server and an advertising server.

This thesis focuses on the development of the backend REST API for the project mentioned above. The aim of this work was to define and document the functional and non-functional requirements for the system under development, to analyze use cases and the overall structure of the project. The author also explains the external systems used by this application, such as the Facebook Graph API, the interfaces of the online payment systems like FIO-banka and Bitcoin, the operating system push notification services of Apple(APNs) and Google(GCM). In the framework of this work, the database structure and server-side application architecture were designed, and the backend interface for communication with mobile and web applications was implemented using modern web development tools such as Python programming language, Django web framework, and PostgreSQL DBMS. After that, the application has undergone unit and performance testing.

Keywords ElateMe, crowdfunding platform, social network, backend API, RESTful, online payments systems

Contents

List of Figures

Introduction

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 focuses on the 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 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 primary goal for the author of the thesis was 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 the analysis of the project as a part of a software development that connects customer's requirements to the system and its subsequent design and development.

Analysis of software project is intended to define the 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 *software team project* (BI-SP1 subject). Our development team divided into groups: Android, iOS, Backend developers, and architect. Our task was to define and document primary client's requirements, implement functioning prototypes of mobile applications and the server API. During BI-SP1 subject, Maksym Balatsko was working on the prototype of backend server, so the choice of used technologies was up to him. Then the technology stack was agreed with the supervisor of the project. Chosen technologies will be discussed in the next chapter.

Because of changes in requirements and the new interface design of the mobile applications, analysis and its documentation have undergone certain changes. And at the start of the BI-SP2 subject implementation of back-end API has begun.

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 [2].

Authorization

F1 Sign up via Facebook. User shall be able to sign up to the ElateMe application with his Facebook account. The 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 a list of user's friends that are already signed up in this application. These users shall be considered as friends in the ElateMe application.

Friendship management

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

Wish management

F7 Create wish. User shall be able to create a wish, set its 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 has donated money yet.

F9 Close wish. User shall be able to close his wish. Money that will have been gathered 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, the user, to whom the wish was addressed, shall not have access 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 the list of wishes he will have contributed.

Feed and notifications

F13 View user's feed. User shall receive the feed with the latest news of his friends.

F14 View user's notifications. User shall receive information about the state of his wishes, new donations, comments, etc. via push notifications.

Donation management

F15 Donate to wish. User shall be able to contribute to wishes of his friends financially.

F16 Refund. In the case of the closure of the wish, all gathered money shall be refunded to donators.

Comments management

F17 View wishes comments list. User shall be able to view the list of comments under the wish he will be browsing.

F18 Comment wish. User shall be able to leave a comment under the wish.

F19 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 [3].

Back-end API

N1 RESTful. Back-end API shall follow architectural constraints of REST architectural style.

N2 HTTPS. The server shall communicate with the client via HyperText Transfer Protocol Secure (HTTPS).

N3 PostgreSQL database. PostgreSQL shall be used as the primary DataBase Management System (DBMS).

N4 Performance. The server shall be able to serve 1500 requests per second.

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. The system shall ensure secure payments.

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

1.4 Use cases

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

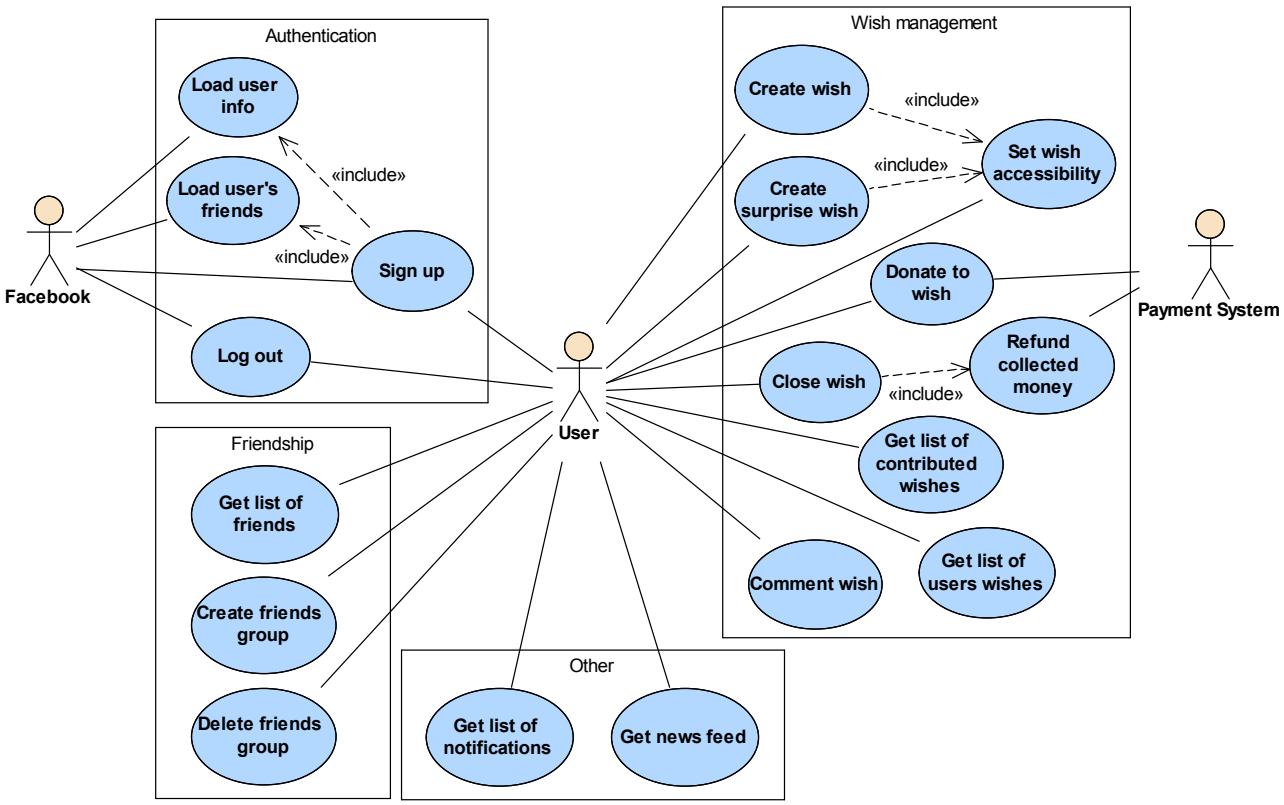


Figure 1.1: Use cases diagram

Use case model includes the following components:

- **Actors** represent people and external systems involved in a particular use 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 redundancies. 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 the subsequent refund.
  - **Other** includes the news feed and notifications.

1.5 Domain model

For simplification of understanding of the primary domain classes and their behavior, it was decided to define so-called Domain model. The domain model is the visual representation of conceptual classes in a domain of interests. Domain model is visualization of things in real-world, not of software components such as C++ or Python class [4].

Domain model 1.2 was created by Maksym Balatsko, team lead and advert server developer of ElateMe. In his bachelor's thesis [1] he describes in detail this diagram, classes specified in it, their attributes, and associations between them.

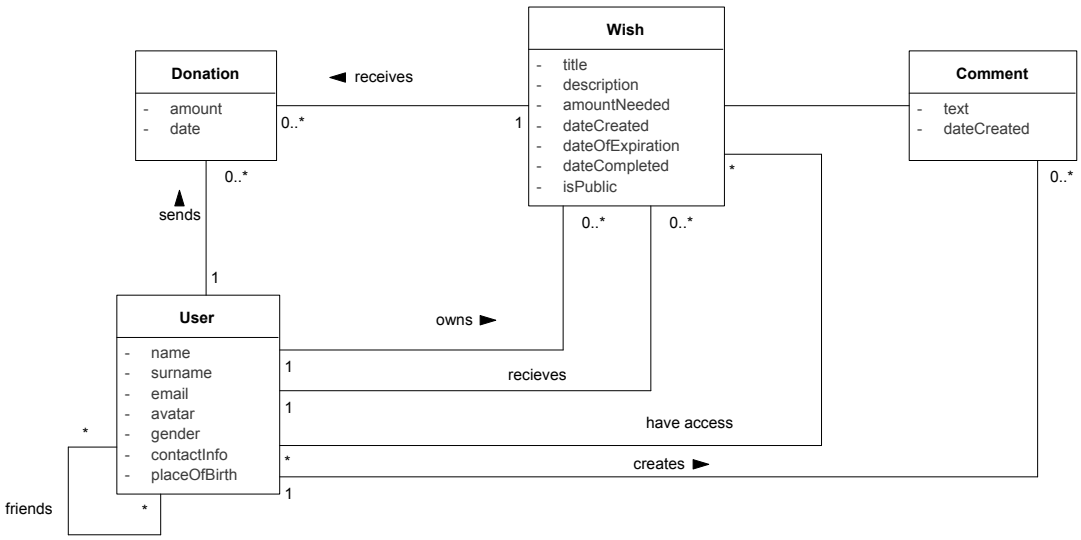


Figure 1.2: Domain model [1]

1.6 System structure

The whole ElateMe application system is divided into components. Principal components are the server, Android, and iOS clients.

The detailed structure of the server and its connection with external interfaces are presented at the component diagram 1.3. As seen in the diagram, the server provides interface for the mobile applications to communicate via Representational State Transfer (REST) API. The server also uses interfaces of Facebook (Graph API) to receive 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 the database via Django *models*. Models in Django is an interface designed to simplify querying to the database.

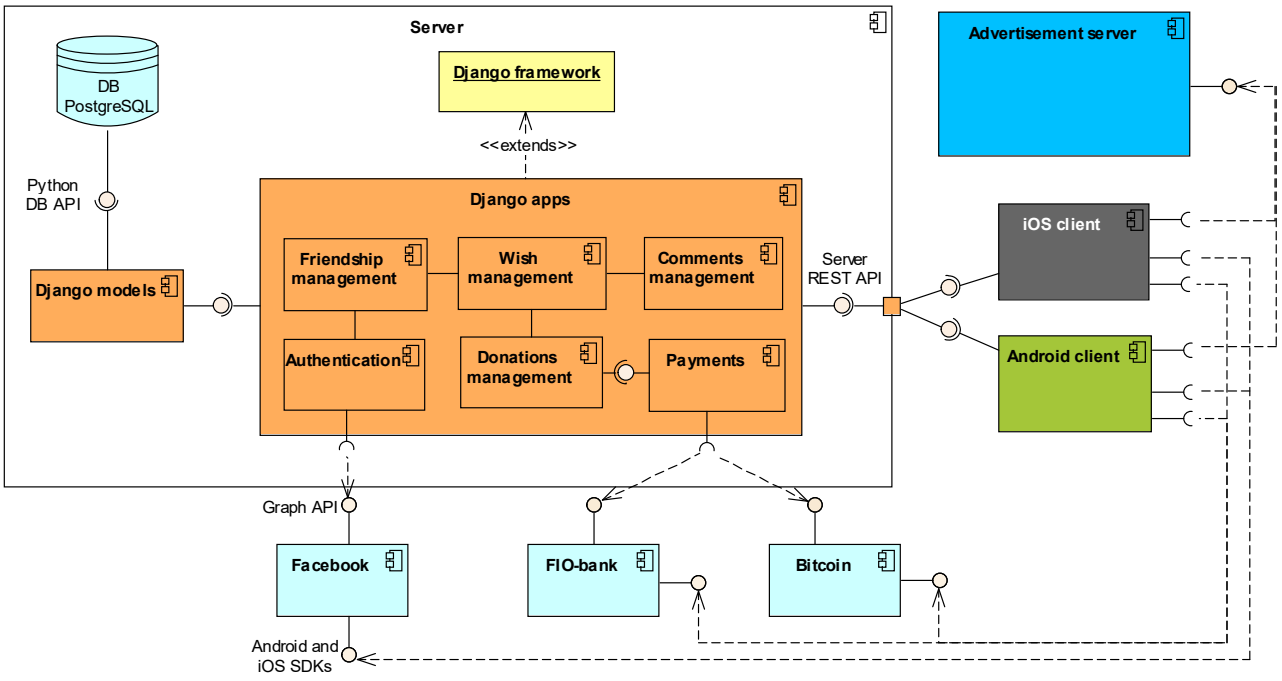


Figure 1.3: Component diagram

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

The user has to be authorized to use the application. The 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 the first login, the application will get from Facebook needed information about the user: first name, last name, email address, a 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 received from Facebook will be stored in ElateMe system as well, because a 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 the 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 the wish by its author and closing upon expiration of the 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 [5] does not provide sufficient functionality for the needs of this application, it was necessary to agree directly with the bank on the provision of required 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 1.4.

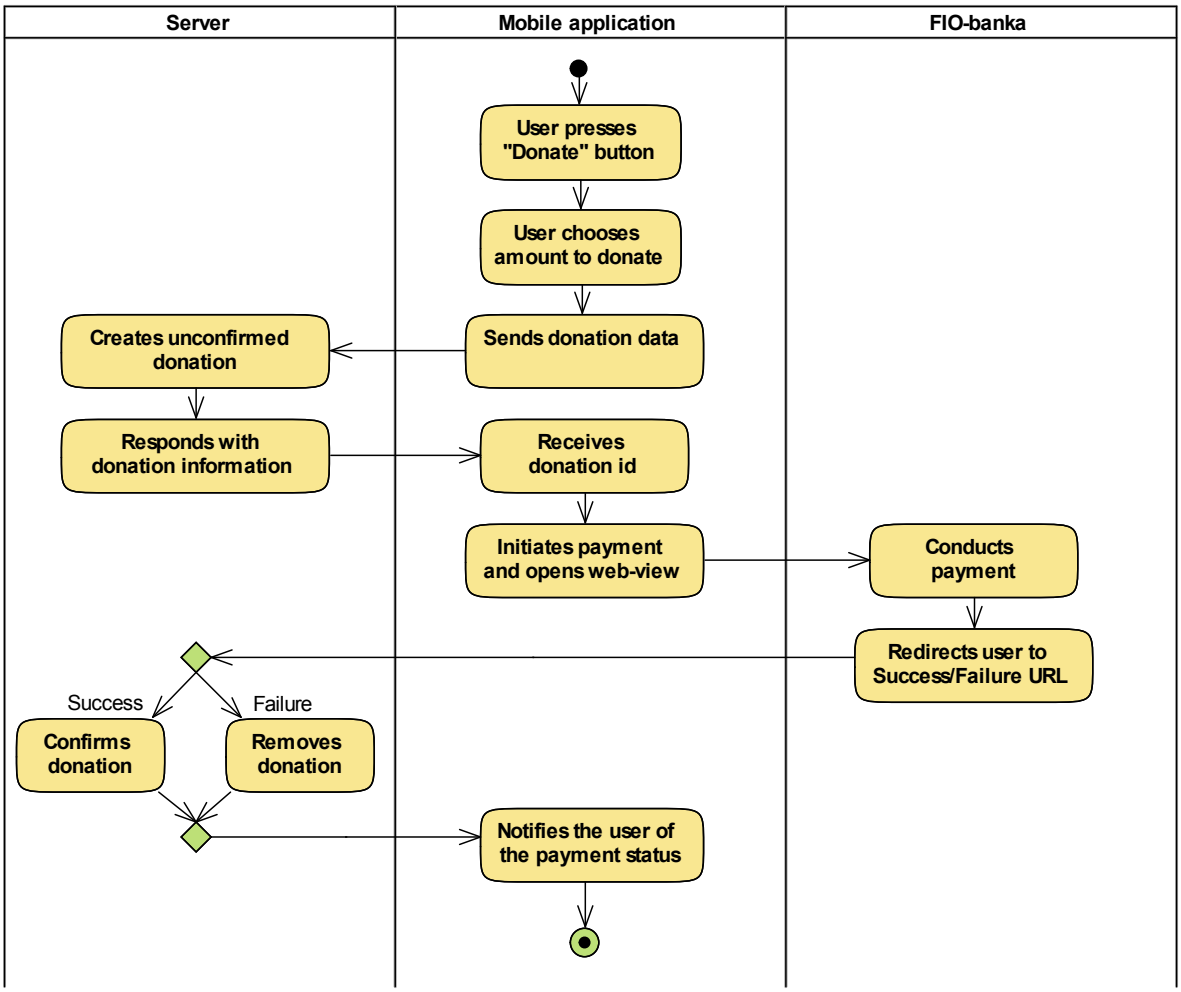


Figure 1.4: Payment via FIO-banka

As seen in 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 the case of success, the server marks the payment as confirmed, alternatively failed. 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 [6]. A detailed analysis of the system of Bitcoins was carried out in the bachelor's thesis of Yegor Terokhin [7], one of the iOS developers of ElateMe. In his work, he describes in detail the principles of work of Bitcoin and the benefits of using this system in the framework of the ElateMe project.

Yegor decided to use Coinbase, to process Bitcoin payments. 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 [8]. 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

As was mentioned before, after closing of the wish, collected money will be refunded to the donators' accounts. ElateMe will have a bank account or online wallet for every payment system that it will provide. As every payment system has a different interface for conducting payments, every donation and its refund will use a different way of payment processing, depending on the available interface for each particular system. There are three possible solutions on processing refunds:

- SDK (e.g. Coinbase SDK)
- HTTP API (e.g. FIO-banka API)
- Web user interface (e.g. mBank)

As long as payment system provides SDK or any other API, the implementation of donation and refunds processing does not require complicated systems to conduct payments. On the current stage of development, it was decided to use only FIO-banka and Bitcoin, which provide appropriate interfaces. But in the case of the addition of payment systems that don't have API for developers, payments, and refunds, in particular, need to be processed through other available interfaces.

Following solution was proposed by Michal Maněna, project supervisor of ElateMe.

User web interfaces of payment systems that don't provide proper API will be used for conducting refunds. It was decided to use Selenium for automatization of payments via user web interface. ***Selenium*** is a set of tools specifically for automating web browsers [9]. So it will simulate actions directly in the web browser to provide refunds to the users' bank accounts. Here it is necessary to reckon with the fact that many banks send a confirmation code via SMS. In this case, it is required to have a Global System for Mobile Communications (GSM) module with a SIM card physically connected to the server. Then there will be running daemon that will collect and save the SMS messages in the database. Then confirmation code will be entered in the corresponding field by Selenium and sent to process payment.

1.9 Push notifications

Push notifications are short 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 the ElateMe application, the user will receive information about the state of his wishes, new donations, comments, etc.

1.9.1 Mechanism of push notifications

For the 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 particular device. The token is generated by Operating system push notification service (OSPNS). After the application is installed on the device it registers itself for receiving of push notifications; OS requests token from OSPNS, the application receives token and sends it to the server.

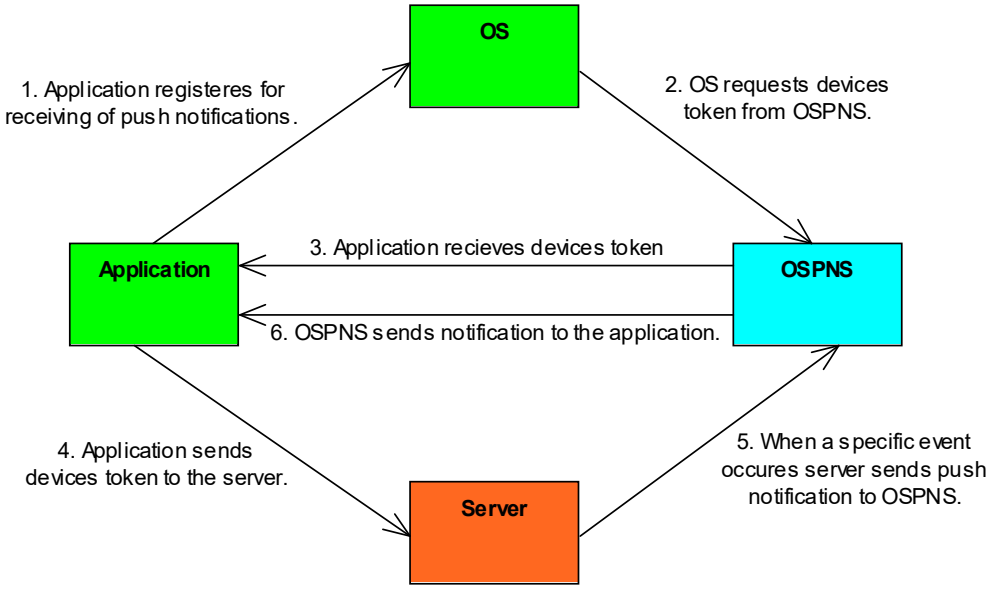


Figure 1.5: Mechanism of push notifications

1.9.2 Actors

- **OSPNS**

  Every operating system has its 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 1.5 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 device and sends the push notifications to OSPNS.

- **Client application**

  The application is registered to receive a 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 the 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 [10].

For the server to be able to get a list of friends and other information about the user, the mobile application needs to receive a 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 specific data of the certain user.

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

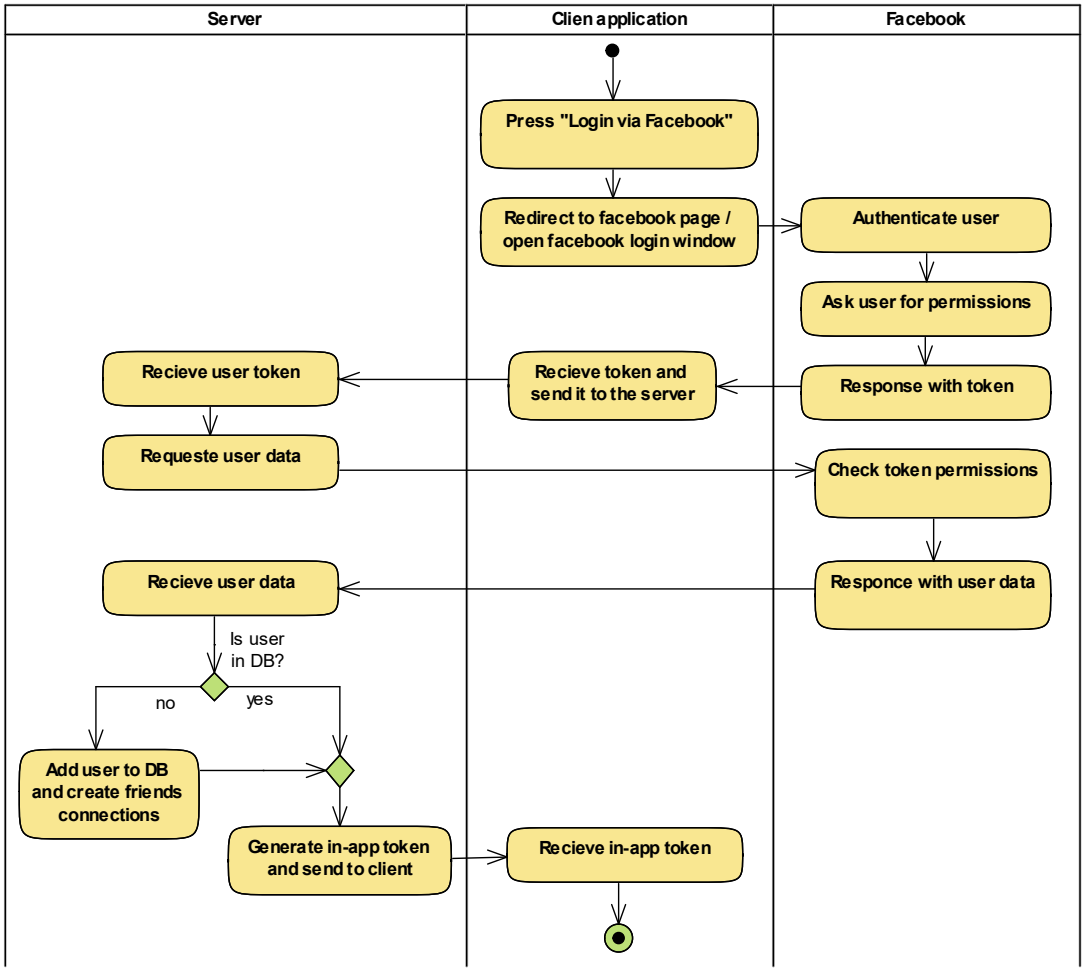


Figure 2.1: Authentication activity diagram

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 [11]. API is considered RESTful if it follows certain rules [12]:

- **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 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 the client rather than the API [11].

2.2.1 Apiary

The apiary service will be used for the server API documentation. ***Apiary*** is a powerful API design stack [13]. The Blueprint API is used to describe the structure of the APIs in the apiary. ***API Blueprint*** is a powerful high-level API description language for web APIs [14].

The following screenshot shows an example of the documented API endpoint, specifically GET request on receiving comments list of the specified wish.

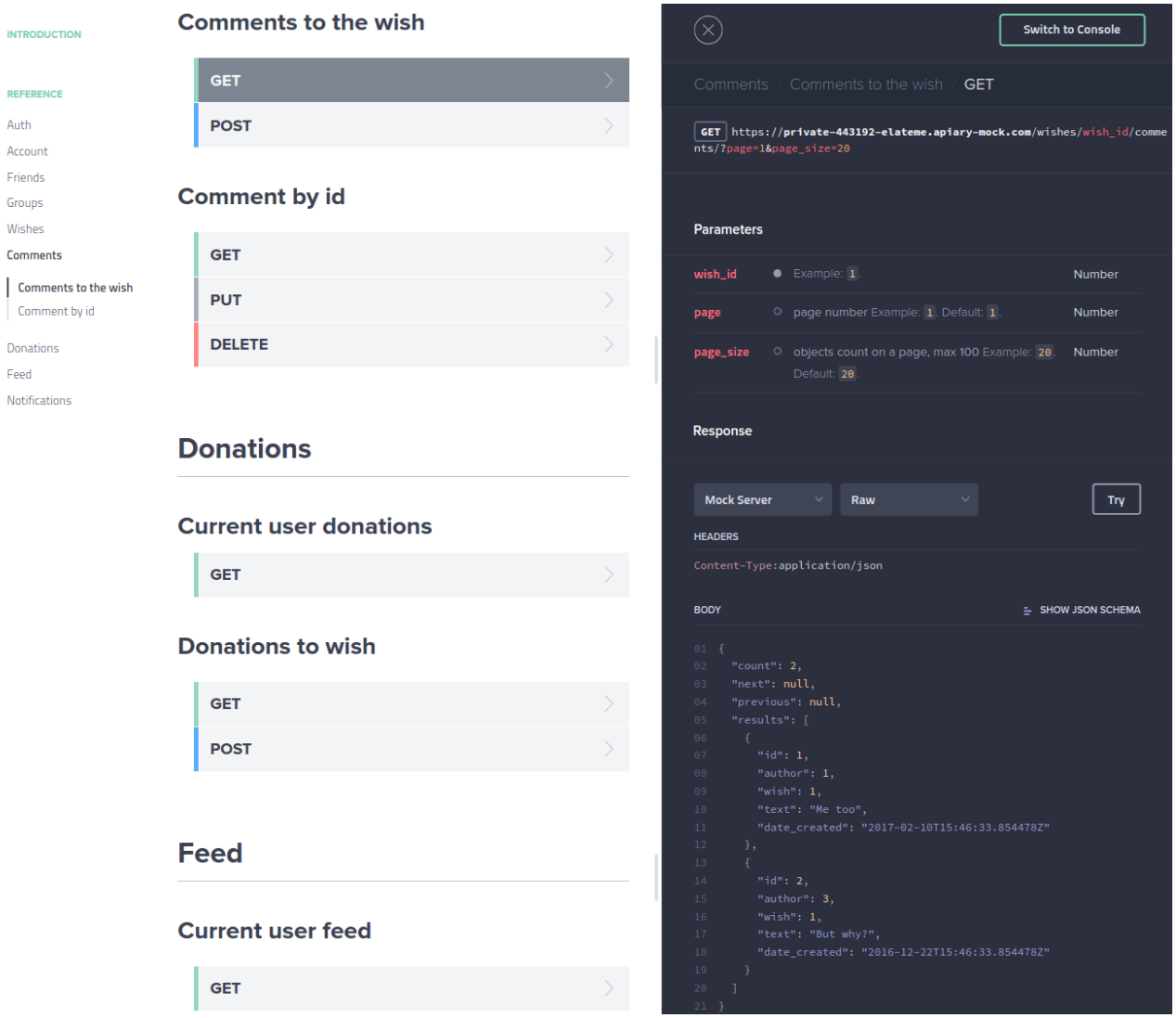


Figure 2.2: Apiary documentation

Documentation of every endpoint contains Uniform Resource Locator (URL), mandatory URL parameters (filled circle), optional URL parameters (hollow circle), request headers and body format if required, response headers and body format.

The corresponding Blueprint to the documentation on the screenshot looks like this:

```
## Comments to the wish [/wishes/{wish_id}/comments/{?page,page_size}]

### GET [GET]

+ Parameters
    + wish_id: `1` (required, number)
    + page: `1` (optional, number) - page number
        + Default: `1`
    + page_size: `20` (optional, number) - objects count on apage
        + Default: `20`

+ Request (application/json)
    + Headers

            Authorization: Token In_APP_Token

+ Response 200 (application/json)
    + Attributes
        + count: 2 (number) - total number of results
        + next (string, optional, nullable) - next page url
        + previous (string, optional, nullable) - previous page url
        + results (array)
            + (object)
                + id: 1 (number)
                + author: 1 (number)
                + wish: 1 (number)
                + text: `Me too`
                + date_created: `2017-02-10T15:46:33.854478Z`
            + (object)
                + id: 2 (number)
                + author: 3 (number)
                + wish: 1 (number)
                + text: `But why?`
                + date_created: `2016-12-22T15:46:33.854478Z`
```

API Blueprint detailed documentation can be found on the official API Blueprint website [14].

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 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 crucial for large-scale projects. Its syntax and standard library simplify and speed up 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. Django framework follows Model-View-Controller (MVC) design pattern. It 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 [15].

2.3.3 Django REST

Django REST framework is an open source project built on Django framework. It contains needed tools for implementation of the 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 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 [16]. Django framework provides great API for working with PostgreSQL databases.

2.3.5 Nginx

Nginx [engine x] is an HyperText Transfer Protocol (HTTP) and reverse proxy server, a mail proxy server, and a generic TCP[1]/UDP[2] proxy server, originally written by Igor Sysoev [17]. According to Netcraft [18], nginx served or proxied 28.50% busiest sites in March 2017.

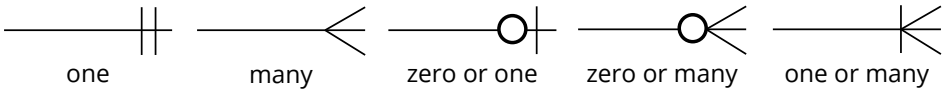
2.4 Database model

One of the primary parts of the web server is the database. A database design, the data model, is created before the implementation of the physical database model.

A data model is a combination of three components [19]:

- **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 to query and update parts of the database.

Thus, the structure of the ElateMe database was defined and documented. The full database model is in the attachment C. 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 the database, which themselves create tables and connections between them.

[1]Transmission Control Protocol (TCP)
[2]User Datagram Protocol (UDP)

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 the electronic attachments.

2.5.1 Authentication

As was said before, in frameworks of this work user authentication will be conducted through his Facebook account. But in the future, other social networks and authorization methods may be added. Since most of the social networks support OAuth 2.0 [10], this allows to make a universal interface for user authorization, that will process user authentication via a social network using token provided by the client (mobile or web application).

Thus, as shown in the diagram 2.3, classes that extend *AbstractSocialAPI* will provide an interface for token processing. *AbstractSocialAPI* defines method *process* that receives token from the client, requests information about the user from the corresponding social network, decides if the user is already registered in the application, registers him (adds to the database) if necessary, and authorizes the user in the application. Information about user is obtained using abstract methods like *request_data*, *get_social_id*, *get_friends*, etc.

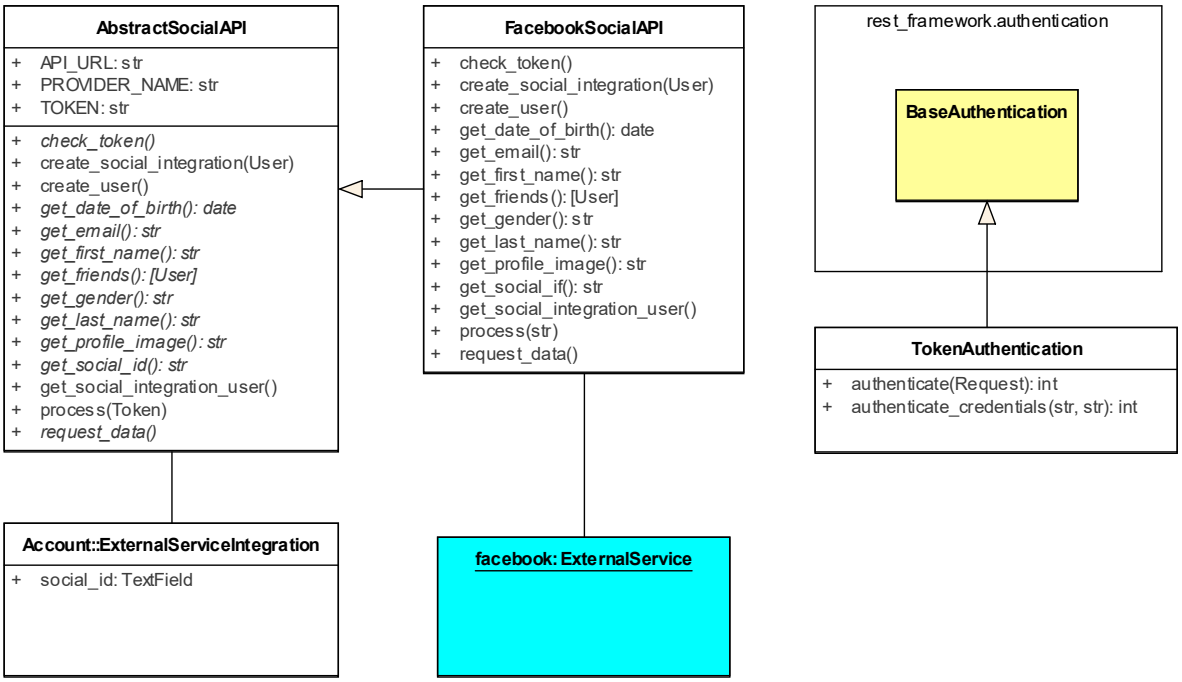


Figure 2.3: Social integration class model

2.5.2 Payments

Payment information will be stored in the models that inherit the *Payment* model. Currently, 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 handling of donations, refunds, 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 and implement the *PaymentsHandler* interface for this particular system.

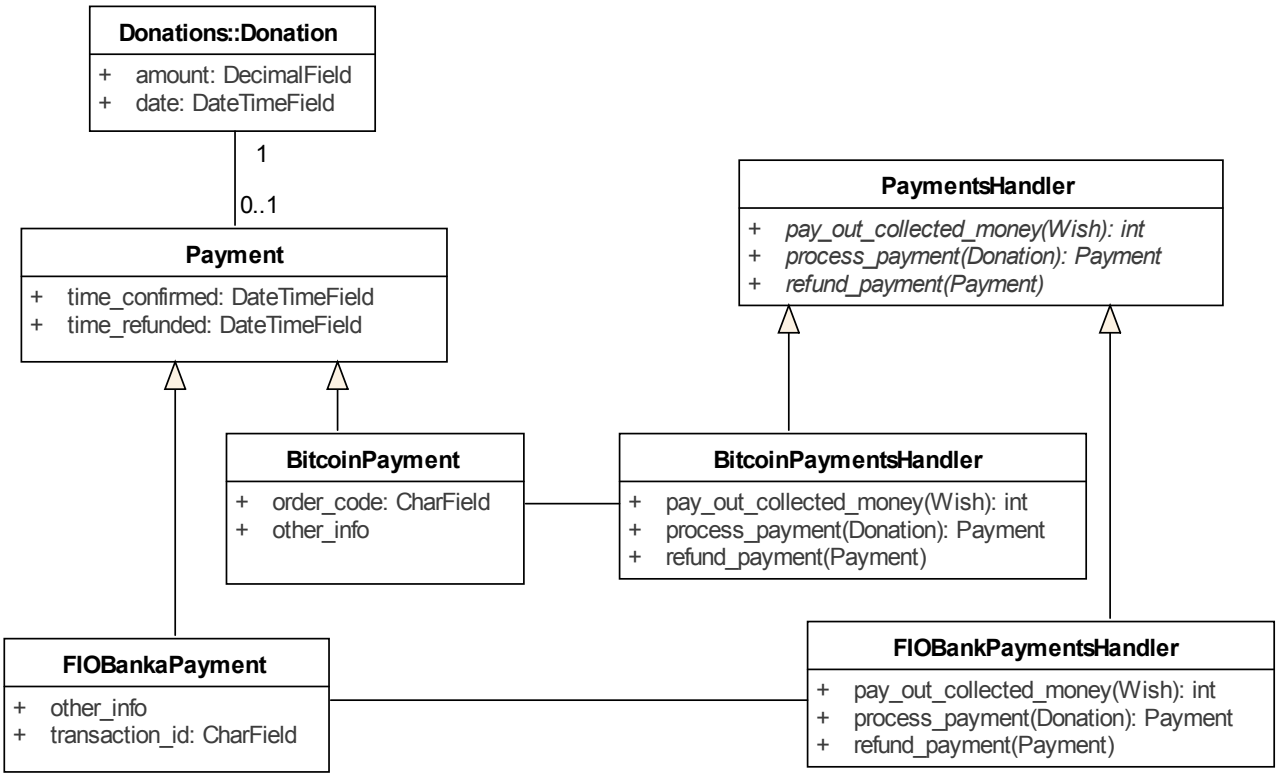


Figure 2.4: Payments class model

2.5.3 Push notifications

As shown in the diagram 2.5, 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 corresponding 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 different OSPNSs. Classes that extend *AbstractNotificationService* and implement method *notify* will be responsible for sending push notifications to the specific OSPNS.

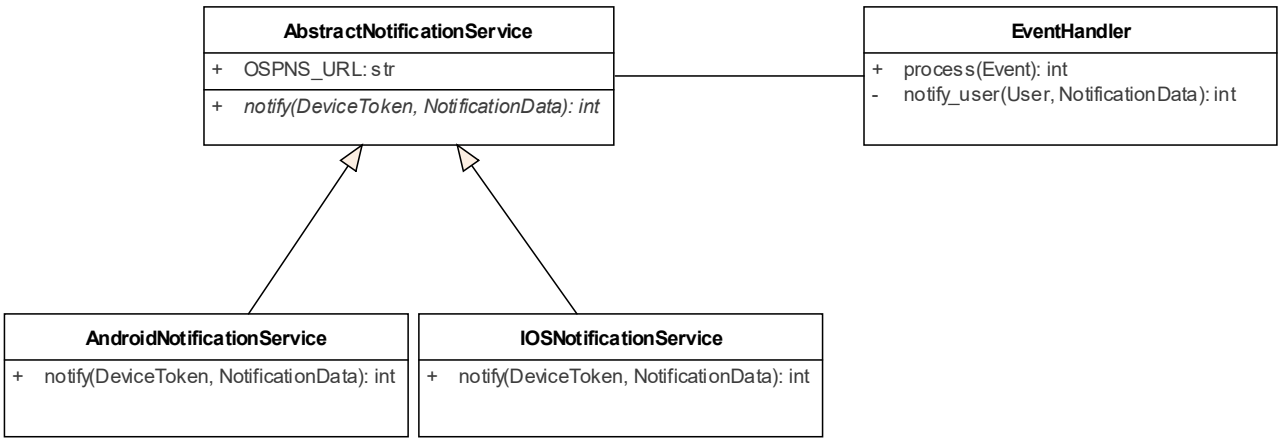


Figure 2.5: Push notification class model

2.6 Deployment model

ElateMe implies an enterprise system. That means that its server application needs to be able to cope with a large number of simultaneous requests, has to be scalable, secure and reliable.

Nginx will be used as a reverse proxy server. ***Reverse proxy server*** is a type of proxy server that typically sits behind the firewall in a private network and directs client requests to the appropriate backend server. Using of such proxy server makes backend application run faster, reduces downtime, consumes less server resources, and improves security [20]. Nginx will also be used as a load balancer for multiple application server instances. ***Load balancing*** means distributing client requests across a group of servers in a manner that maximizes speed and capacity utilization while ensuring no one server is overloaded, which can degrade performance. If some server goes down, the load balancer redirects traffic to the remaining online servers [21].

Design of the ElateMe deplyment model is presented on the diagram 2.6.

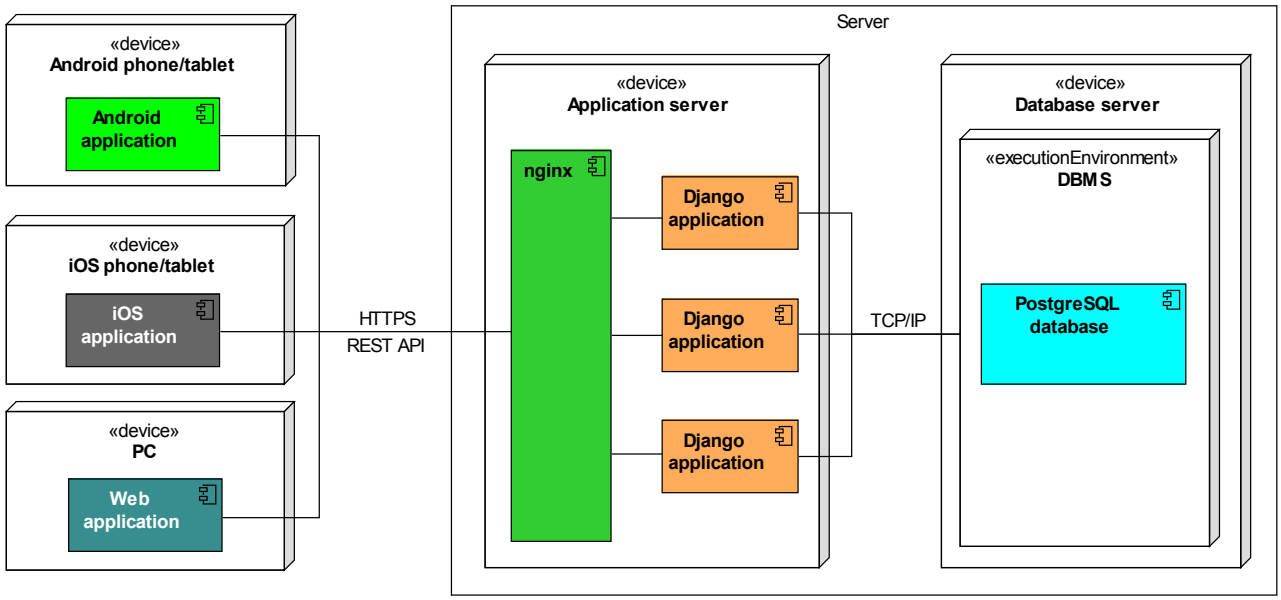


Figure 2.6: Deployment model

As seen in the diagram, backend application and database will be running on separate servers. Pros of such approach are

- Application and database don't use the same server resources (CPU, Memory, I/O, etc.)
- It allows vertical scaling, by adding more resources to whichever server needs increased capacity.
- It increases security by removing the database from the Demilitarized Zone (DMZ)[3].

Nginx in this model is used as a load balancer which improves performance and reliability by distributing the workload across multiple Django application instances. It also allows horizontal scaling, i.e. environment capacity can be scaled by adding more servers to it.

Currently, the server is deployed on the Virtual Private Server (VPS) with the database and Django application on the same machine for testing purposes.

[3]DMZ is a host or small network that is a neutral zone between a local network and the Internet.

Chapter 3

Implementation

This chapter contains a description of the implementation of the project's server side. This part will describe the structure of the project. It 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.

Note that the installation guide for this application is in the attachment D.

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, which implement interfaces and extend classes, which 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 the 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 the database. All models extend class *model* from *django.db* module and usually represent single table in a database [22].

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 querying 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 URL. This function takes a Web request and returns a Web response, in this case, JavaScript Object Notation (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)
```

Such 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 JSON format. It was decided to use the Django REST

framework, to simplify the implementation of such requests and the corresponding auxiliary classes (serializers, paginations, etc.)

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 deserialization 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

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

```
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 the 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**. 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 provides user notifications. At the moment, it provides the REST interface for getting news list. Later this application will work with push notifications.
- **wishes**. Application provides the interface for processing user wishes. It also contains sub-application *comments*.

3.1.7 Django settings

Django ***settings*** is a module that contains all the configuration of the Django project. The main configurations to notice are:

- **INSTALLED_APPS**. A list of all apps in a project.
- **ALLOWED_HOSTS**. A list of the host/domain names that this Django site can serve.
- **DATABASES**. A dictionary containing the settings for all databases to be used with Django.
- **DEBUG**. A boolean that turns on/off debug mode.

A complete list of settings available in Django can be found in the official documentation [23].

As for development and deployment it is needed to have different configurations for allowed hosts, databases and debugging, in the *server_api* folder, alongside with *settings* module there were created two modules: *prod_settings* and *dev_settings*. They contain specific configurations for the production and development respectively. For example *dev_settings* have configured SQLite database and set debug mode on, while *prod_settings* defines settings for PostgreSQL without debug mode. To enable *prod_settings* it is needed to set environment variable *PRODUCTION*. Otherwise *dev_settings* will be used.

Following lines were added to the main *settings* module to make it work:

```
if os.environ.get('PRODUCTION', False):
    from .prod_settings import *
else:
    from .dev_settings import *
```

3.2 Python Virtual Environment

Since this application uses a set of dependencies that don't come as a 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 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 global site-packages directory clean and manageable [24].

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 attachment D.

Chapter 4

Testing

4.1 Unit tests

Automatic testing was performed using unit tests, alongside with the development. 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 [25].

4.1.1 Django REST tests

Native tools were used for testing, 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 primary database. SQLite DBMS is used for testing in this project.

4.1.2 Auxiliary methods

I wrote a set of auxiliary methods that simulate HTTP requests to the server. These methods take URL, to which the request is sent, and, optionally, 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 a 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 the 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 an open source, single-threaded command line program for benchmarking a web server.

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

Testing command looks like this:

```
HEADERS=(
        "Authorization:Token b0edca023c283518f20b36894708" \
        "User-Agent:test-agent"\
        )

URL="https://api.elateme.com/wishes/"

curl -sL "${HEADERS[@]/#/-H}" "$URL"

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

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

In this case, *curl* should print **200**, which means a successful request. Further testing with the same headers and the 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 primary 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 result.

Finding a bottleneck point is necessary to optimize the performance of the server. There are several possible problematic places:

- **Database**. Slow connection, long requests processing.
- **Django**. Unsuitable Django configuration.
- **Nginx**. Incorrect proxy configuration, wrong number of workers, logging, caching, static files, etc.
- **Hardware**. Low hardware performance.

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

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. 23 users were stored in the database with 5 to 30 wishes each, at the time of testing.

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

It was enough to run the Apachebench locally on the port on which Django server is running to test Django separately from Nginx (without a proxy):

```
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 or hardware performance. Same tests of this Django project on authors local machine showed much better results, about 350 requests per second.

4.2.4 Nginx test

It was enough to make virtualhost that served a static page to test Nginx separately from Django application. Here is a testing of this page:

```
ab -c 100 -n 5000 "https://api.elateme.com/test.html"
```

Results of this test showed similar rate as requests to the URLs of server API. This indicates that the problem is in Nginx or hardware performance.

4.2.5 Results

Taking into consideration everything mentioned above the problem is presumably in server's hardware. Currently, the server is running on the free VPS, which is not designed for enterprise projects, so testing on current server is not an accurate indicator of project performance. Therefore, performance tests will be conducted again after backend application is deployed on the full-fledged server.

Conclusion

The aim of this work was to learn how to develop a complex backend system from the analysis of the requirements and the design of the future system to implementation, deploy and testing. Functional and non-functional requirements, use cases and business processes were documented. The structure of the project, the scheme of the database and the class model, the payment and refund systems were designed, REST API for communication with mobile and web applications was implemented and the implemented application was tested by the unit and performance tests.

In the framework of this thesis, the author studied the use of such web development tools like Python, Django and Django REST frameworks, Nginx, and PostgreSQL. The author also learned about ways to integrate user authorization via Facebook using OAuth 2.0 protocol and payment systems like FIO-Bank and Bitcoins.

Work contribution

The reader of this work can learn for himself how to implement the authorization system through the OAuth 2.0 protocol, the basics of using the interfaces of the FIO-banks and the Bitcoin payment systems. Also, the work explains the structure of the implementation of this project, which will be useful for future developers of this platform.

Future outlook

The next step in developing of the backend of the application will be the completion of the REST API and the integration with the advert server. Also, it will be necessary to study in detail the configurations of the Nginx and ways to optimize the application and complete performance testing on the production server.

Bibliography

[1] Balatsko, M. *ElateMe - Project management and Advert server. Bachelor's thesis*. Czech Technical University in Prague, Faculty of Information Technology, 2017.

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

[3] 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`

[4] Larman, C. *Applying UML and Patterns: An Introduction to Object-oriented Analysis and Design and the Unified Process*. Safari electronic books, Prentice Hall PTR, 2002, ISBN 9780130925695. Available from: `https://books.google.cz/books?id=r8i-4En_aa4C`

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

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

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

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

[9] Reference. Selenium Documentation [online]. [Cited 2017-05-11]. Available from: `http://www.seleniumhq.org/docs/`

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

[11] 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.

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

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

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

[15] 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.

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

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

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

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

[20] Smith, F. Maximizing Python Performance with NGINX, Part 1: Web Serving and Caching [online]. March 2016, [Cited 2017-05-12]. Available from: https://www.nginx.com/blog/maximizing-python-performance-with-nginx-parti-web-serving-and-caching/

[21] Reference. What is a reverse proxy server? [online]. [Cited 2017-05-12]. Available from: https://www.nginx.com/resources/glossary/reverse-proxy-server/

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

[23] Reference. Django settings [online]. [Cited 2017-05-12]. Available from: https://docs.djangoproject.com/en/1.11/ref/settings

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

[25] 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

APNs Apple Push Notification service

DBMS DataBase Management System

DMZ Demilitarized Zone

GCM Google Cloud Messaging

GSM Global System for Mobile Communications

HTTPS HyperText Transfer Protocol Secure

HTTP HyperText Transfer Protocol

JSON JavaScript Object Notation

MVC Model-View-Controller

OSPNS Operating system push notification service

REST Representational State Transfer

SDK Software Development Kit

TCP Transmission Control Protocol

UDP User Datagram Protocol

URL Uniform Resource Locator

VPS Virtual Private Server

Appendix **B**

Contents of enclosed CD

- `readme.txt` ..... the file with CD contents description
- `src` ..... the directory of source codes
  - `server_api` ..... implementation sources
  - `thesis` ..... the directory of LaTeX source codes of the thesis
- `text` ..... the thesis text directory
  - `BP_Kuzmovych_Yevhen_2017.pdf` ..... the thesis text in PDF format
  - `Assignment.pdf` ..... the thesis assignment
- `documentation` ..... the generated documentation of the analysis
  - `Class model.pdf` ..... class model documentation
  - `Database model.pdf` ..... database model documentation
  - `Requirements.pdf` ..... requirements documentation
  - `Use cases.pdf` ..... use cases documentation
  - `Component diagram.pdf` ..... component diagram documentation
  - `Deployment model.pdf` ..... deployment model documentation

Appendix C

Database model

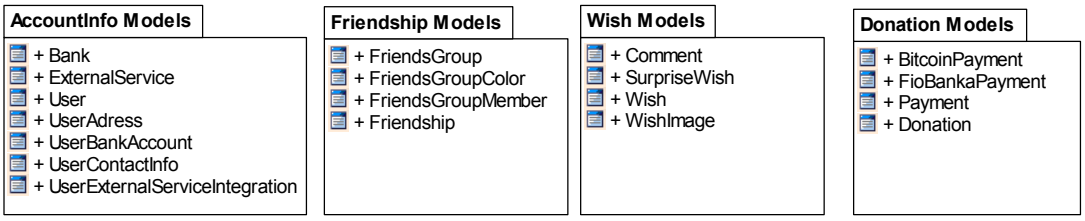


Figure C.1: Database packages

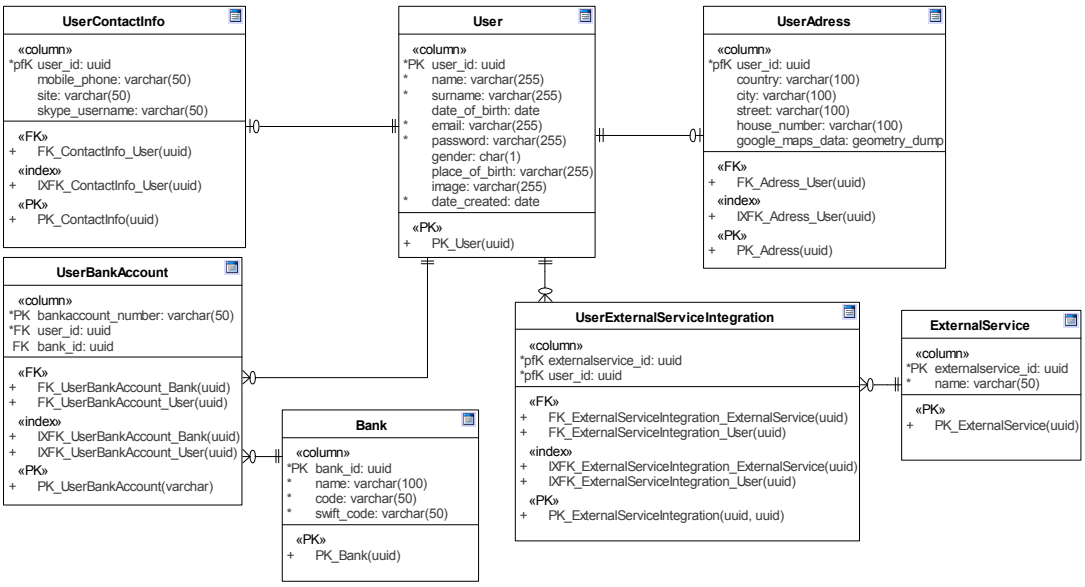


Figure C.2: Account models

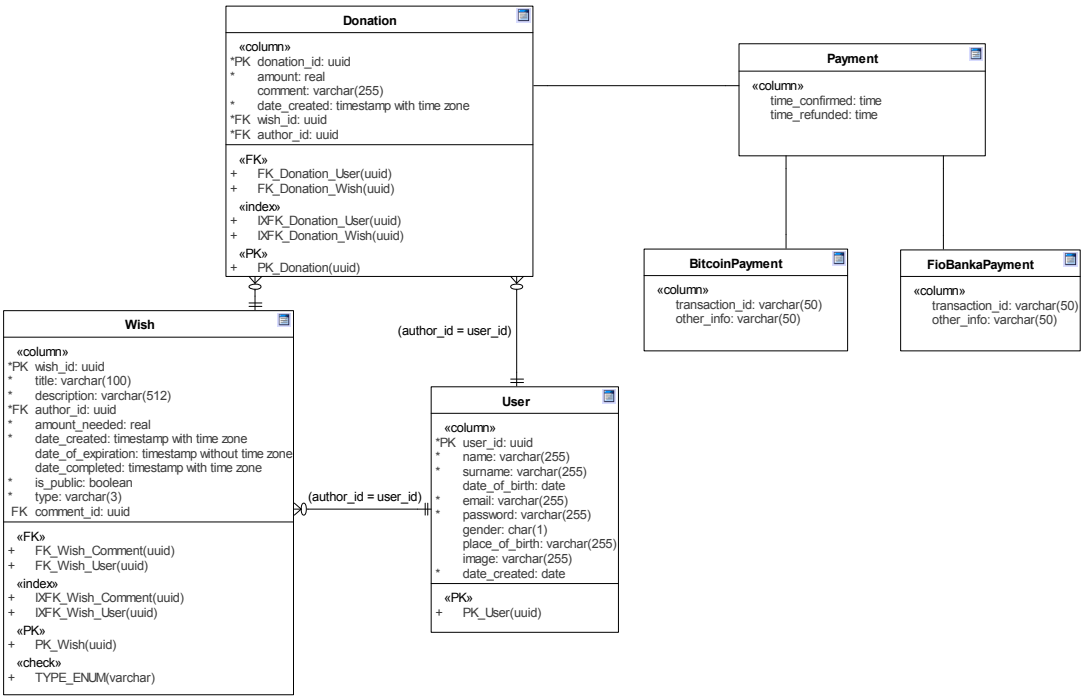


Figure C.3: Donation models

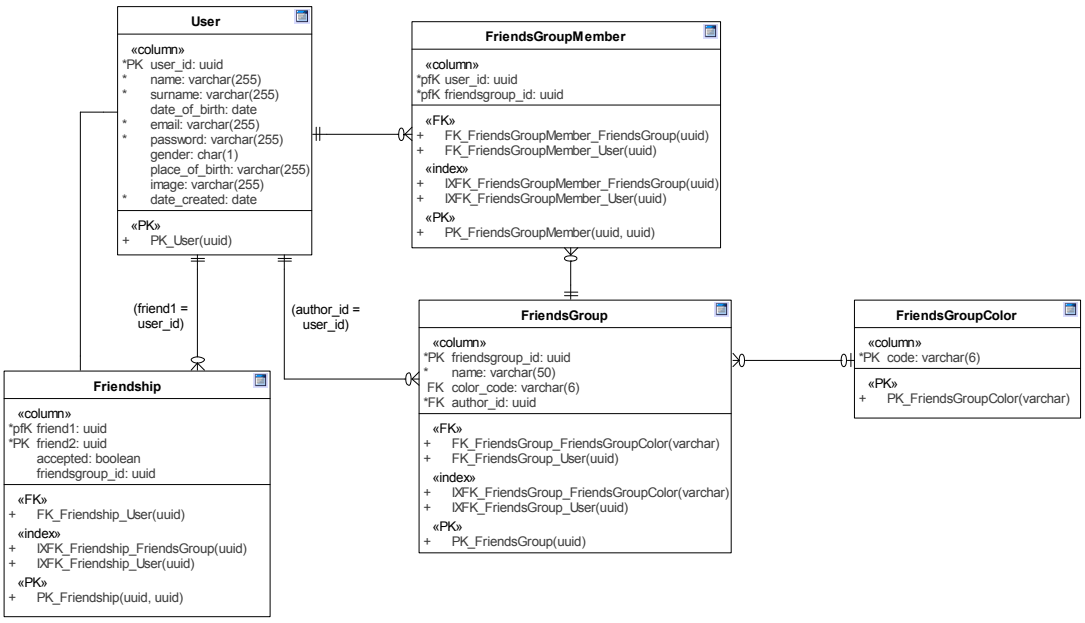


Figure C.4: Friendship models

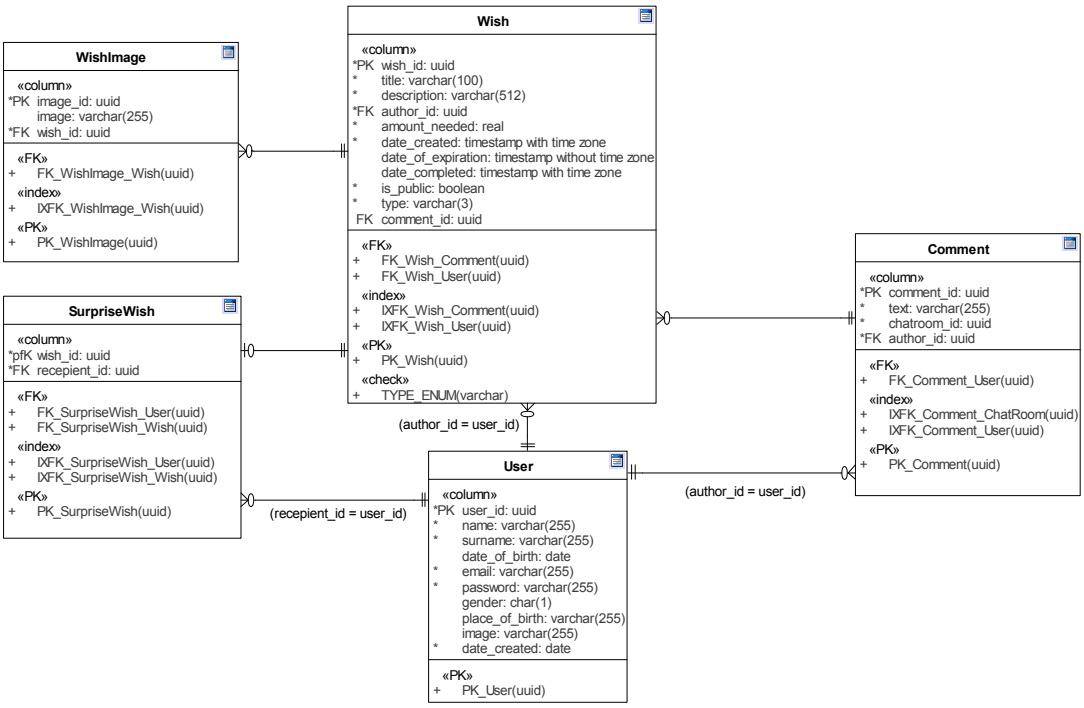


Figure C.5: Wish models

Appendix D

Installation guide

This is the guide on how to setup, run and deploy ElateMe back-end server on Ubuntu OS.

Requirements

To be able to setup and run project you need

- python3
- pip
- virtualenv

To install run:

```
sudo apt-get update
sudo apt-get install python3
sudo apt-get install python-pip
pip install virtualenv
```

Setup

Clone repository:

```
git clone git@repo.micman.cz:allmywishes/server-api.git ElateMe
```

Go to `ElateMe` folder and create virtual environment:

```
cd ElateMe
virtualenv -p /usr/bin/python3.5 venv
```

To begin using the virtual environment, it needs to be activated:

```
source venv/bin/activate
```

Install requirements inside virtual environment:

```
pip install -r requirements.txt
```

Migrate Django models:

```
python manage.py makemigrations
python manage.py migrate
```

By default project runs with `DEBUG=True` and SQLite database.
Run tests:

```
python manage.py test
```

Now you should be able to run project locally:

```
python manage.py runserver
```

Server should be running on localhost:8000