CZECH TECHNICAL UNIVERSITY IN PRAGUE
FACULTY OF INFORMATION TECHNOLOGY

ASSIGNMENT OF BACHELOR'S THESIS

Title:	RPG game with augmented reality features - server part
Student:	Jakub Čech
Supervisor:	Ing. Miroslav Balík, Ph.D.
Study Programme:	Informatics
Study Branch:	Software Engineering
Department:	Department of Software Engineering
Validity:	Until the end of winter semester 2018/19

Instructions

The aim of the thesis is to specify, design, and implement a functional prototype of the server part of a RPG game with features of augmented reality (AR).
1. Create a story line and rules for the game. Consider intensive usage of geolocation and AR features.
2. Formalize the following requirements for the implementation of the server part:
 - the data layer consists of a database engine and a caching,
 - users can use their Google accounts to login and play,
 - the server part provides an API,
 - the communication between client and server parts of the application must be secure.
3. Design the server part of the game.
4. Design a suitable front-end part for server administration.
5. Discuss and choose a suitable implementation platform and related technologies (databases etc.).
6. Implement the functional prototype, document it, and perform suitable testing.
7. Tightly cooperate with Tomáš Zahálka who works on the client part.

References

Will be provided by the supervisor.

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

prof. Ing. Pavel Tvrdík, CSc.
Dean

Prague March 6, 2017

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

Bachelor's thesis

Location-based Role Playing Game

Jakub Čech

Supervisor: Ing. Miroslav Balík, Ph.D.

June 15, 2017

Acknowledgements

TODO I would like to thank myself for doing this. I am an awesome and humble person. With great power comes great responsibility and no one else is as good or worthy as I am to be thanked. Ave Kuba!

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 June 15, 2017 ……

Czech Technical University in Prague
Faculty of Information Technology

Citation of this thesis

Čech, Jakub. *Location-based Role Playing Game*. Bachelor's thesis. Czech Technical University in Prague, Faculty of Information Technology, 2017.

Abstrakt

TODO Hvězdy jsou krásné, protože je na nich květina, kterou není vidět. Poušť je krásná právě tím, že někde skrývá studnu. Ať je to dům, hvězdy nebo poušť, to, co je dělá krásnými, je neviditelné!

Klíčová slova **TODO** #deep, #thoughtoftheday, #follow4follow

Abstract

TODO Place the 2 cups of crushed ice into a cocktail shaker. Pour the rum, lime juice, and simple syrup over the ice, cover, and shake well. Remove the ice from your serving glass and strain the drink into it. Serve immediately.

Keywords **TODO** Daiquiri, Coctail, Rum, Cuba

Contents

List of Figures

List of Tables

Introduction

The world of mobile devices is quickly evolving. Smartphones and tablets are becoming more powerful and not only in terms of computational power and available memory. Nowadays, mobile devices are packed with various sensors. It is possible to integrate data from GPS (Global Positioning System) to quickly determine device's position. This opens us door to augmented reality (AR) applications.

Surprisingly, there are not many existing augmented reality games on the market. Thanks to couple successful AR games in recent years, which I will explore later, public awareness of this genre rapidly rose.

In my bachelor thesis, I will create the server part of a role-playing augmented reality mobile game. I will work in cooperation with Tomáš Zahálka, who works on the client part. I will design and implement a prototype, test it and deploy it.

Chapter 1

Goals of the Thesis

The thesis is mainly focused on practical game server development. In the research part, I will analyze existing augmented reality games, research available database management systems and explore frameworks used for server development.

In the practical part, I will specify features and requirements of the server. The goal is to support operations the mobile client might need for seamless gaming experience. These operations include for example getting game objects near a player, or login. The main goal is to implement server's functionality to communicate with clients, parse their requests and respond in valid format. The underlying goals include the need to use a database, caching, and communication with external services. Lastly, the game server should be tested and deployed.

Chapter 2

Game Description

This is a role-playing mobile game with augmented reality features. Player becomes a character in an invisible alternative reality. He can explore the new magical world, using his phone which displays a map with objects around him. Player can discover monsters, like goblins or skeletons, and fight them to death for a reward. As he gains more experience and gold, he can buy himself more powerful weapons and armor in a shop. Tired of running in the outside world? Player can exchange his real money for in-game gold.

For prototyping purchases, some described features are limited. Complete list of all available actions is specified in *Features* section of Requirements 3.3.1.

Chapter 3

Analysis

3.1 Existing Games

3.1.1 Parallel Kingdom

The game was developed by PerBlue and released in October 2008. Parallel Kingdom is the most similar game to ours.

"Parallel Kingdom is a mobile, location based, massively multiplayer game that uses GPS location and Google Maps to place users in a virtual world. Parallel Kingdom is the first location based RPG for the iOS and Android platforms. The game is set in a virtual world or "Parallel Kingdom" where users claim their territories based on their GPS location or by making friends who invite them to travel to new places." [1]

The game gained on popularity and even reached 1 million player by the end of January 2012 [2]. Parallel Kingdom discontinued on November 1, 2016 for undisclosed reasons.

3.1.2 Ingress

Developed by Niantic, which was then part of Google, the game was released in December 2013 for Android, followed by an iOS version in June 2014. It is a location based, massively multiplayer game. A player joins one of two factions, Enlightened or Resistance, and then as a part of his team capture regions of the game map. Fate of the factions relies on players' cooperation. Thanks to that players meet in real life and coordinate their actions.

"Move through the real world using your Android device and the Ingress app to discover and tap sources of this mysterious energy. Acquire objects to aid in your quest, deploy tech to capture territory, and ally with other players to advance the cause of the Enlightened or the Resistance." [4]

Ingress is a very successful augmented reality game with tens of millions installs.

Figure 3.1: Screenshot of Parallel Kingdom [3]

3.1.3 Pokémon GO

After its success with Ingress, Niantic started working on a new game Pokemon GO. Released in July 2016, the game became an incredible hit. Even though the game faced many problems during its launch, mainly caused by the unexpected success and more active users than Pokémon GO servers were able to handle, in the first 80 days Pokémon GO reached about 550 millions downloads and earned about $470 million [5].

"Venusaur, Charizard, Blastoise, Pikachu, and many other Pokémon have been discovered! Now's your chance to discover and capture the Pokémon all around you—so get your shoes on, step outside, and explore the world. You'll join one of three teams and battle for the prestige and ownership of Gyms with your Pokémon at your side." [6]

Pokémon GO is very similar to Ingress and uses the same crowd-sourced geographical data.

3.2 Use Cases

3.2.1 Actors

Actor is a role played by a user or other system that interacts with the server. The most general role is *Client* and anyone who accesses the server through API is considered to play either this role or any of its children. From now on,

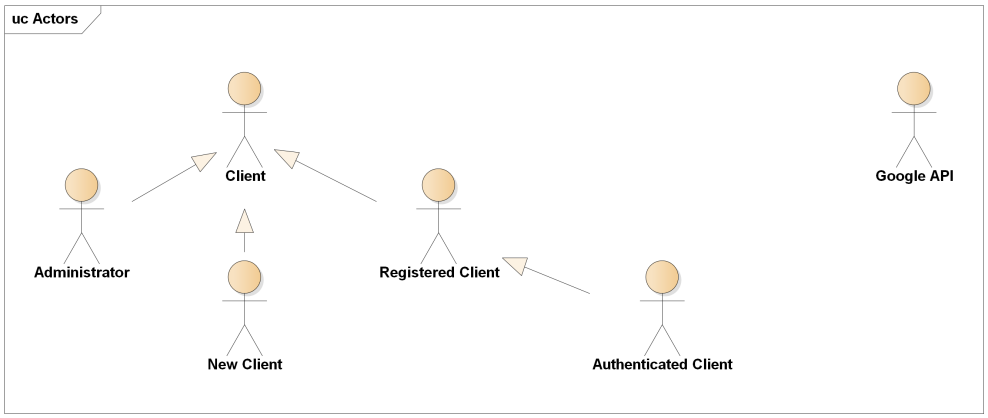

Figure 3.2: Use Case Diagram – Actors

the terms client and user will be used interchangeably. Refer to Figure 3.2 for the role hierarchy.

Administrator is a client who has privilege to create and maintain the functionality of the game. *New Client* is a user who is not yet registered and probably accesses the game for the first time. *Registered Client* is the default role for a user who already has a valid account but is not logged in. Lastly, *Authenticated Client* has all the required privileges to play the game. This client will be refereed to as *player*. The last main actor is Google API which provides the access to Google services.

3.2.2 Authentication

Authentication use cases user performs to get promoted to more privileged roles. The basic transition flow is *New Client* → *Registered Client* → *Authenticated Client*. See Figure 3.3.

1. **Register**
   The only choice *New Client* has is to register a new account. He uses provides his Google identity and chooses an username. After his identity is verified, the server creates a new profile and logs the user in.

2. **Log in**
   The *Registered Client* must log in before he can access any of the game features. He provides his Google identity which is then verified and matched to an account. An access code is issued for the future identification within the session.

3. **Verify user's identity**
   Proper verification is needed when the game server receives a Google

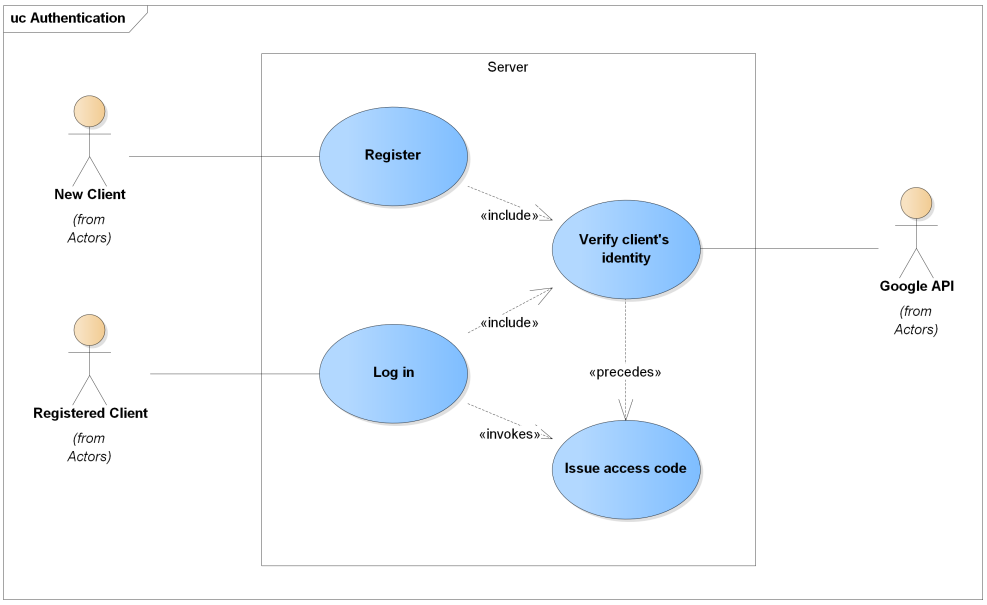

Figure 3.3: Use Case Diagram – Client's authentication

identity of a user. The server contacts the *Google API* which responds with user's personal information if the identity is valid.

4. **Issue access code**
   This use case is invoked during login process. Server issues a unique access code to the user. Only *Authenticated Client* has such code.

3.2.3 Actions

Action is an event triggered by player's interaction with a game object. See the diagram in Figure 3.4.

1. **Equip item**
   A player wants to equip an item from his inventory. The item will be assigned to a specific slot. For example the player equips a sword to his right hand.

2. **Buy item**
   A player wants to exchange gold in a shop for an item he chooses. When the purchase is finished, the player receives the item to his inventory.

3. **Kill monster**
   A player wants to kill monsters to progress in the game. If he successfully kills the monster, he is rewarded with gold and experience

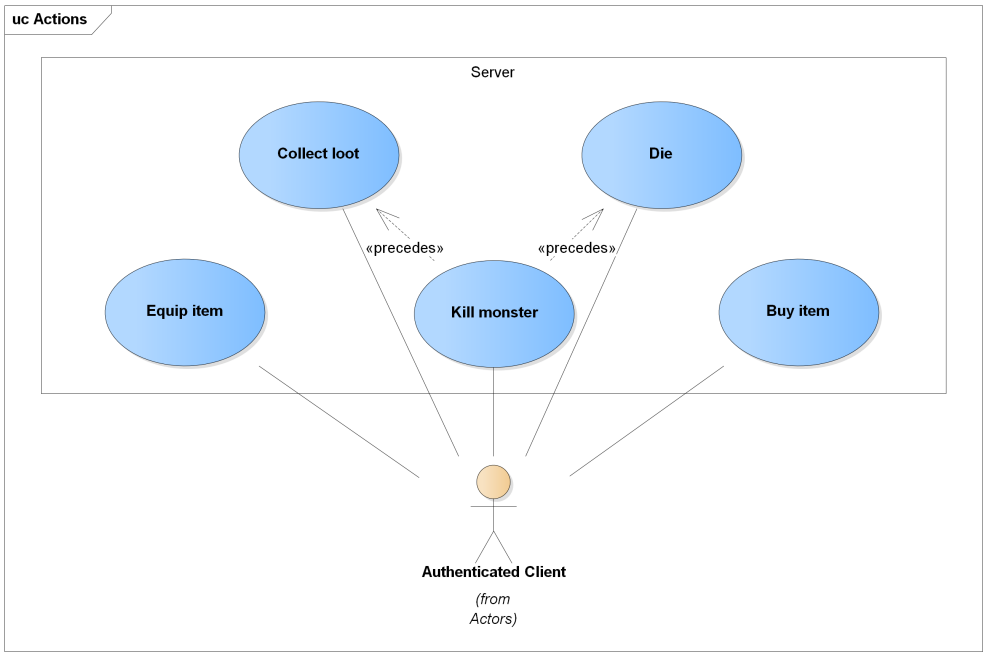

Figure 3.4: Use Case Diagram – Player's actions

4. **Collect loot**
   This action must be preceded by the *Kill monster* use case. A player can choose to collect loot from the monster he killed.

5. **Die**
   This action must be preceded by the *Kill monster* use case. A player who lost his fight against a monster dies and is punished with some penalty.

3.2.4 Miscellaneous

The following use cases mainly cover requests made by client application. See the diagram in Figure 3.5.

1. **Get inventory**
   The application requests all items the player owns. It is also possible to retrieve all the player's equipment with each item assigned to some slot.

2. **Get profile**
   A client may need to synchronize its internal state of the players profile with the server. The application is provided with complete state of the player's profile including attributes like health, gold, and experience.

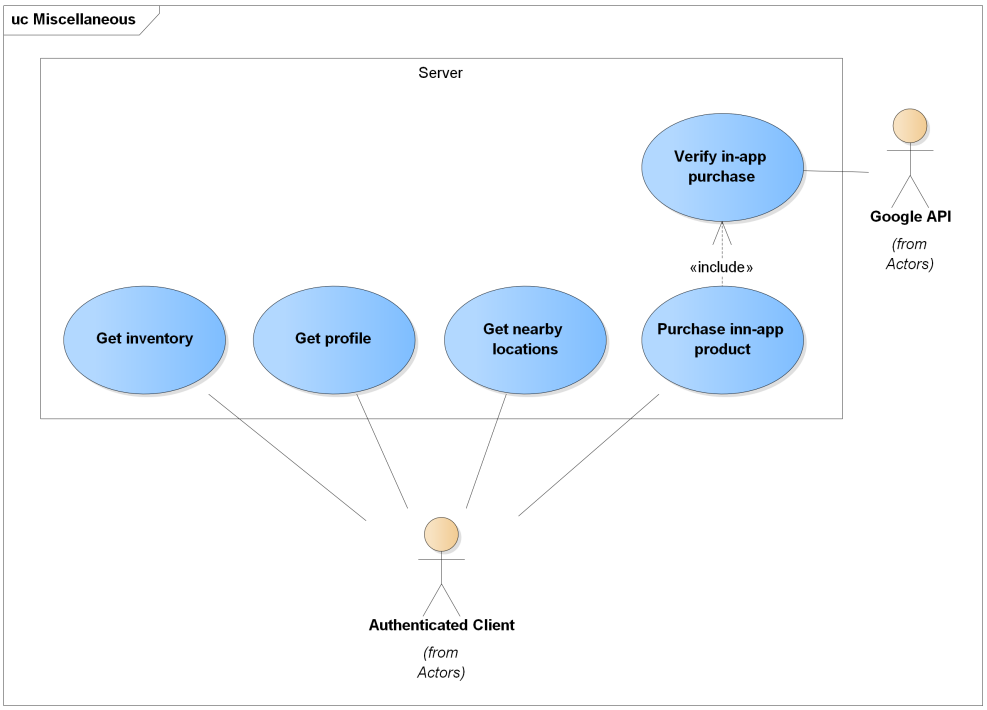

Figure 3.5: Use Cases: Miscellaneous

3. **Get nearby locations**
   This is a critical functionality of the server. A client asks for game locations near his coordinates. The server provides such locations along with their associated game objects.

4. **Purchase in-app product**
   The application supports micro-transactions. Since the purchase is made client-side, the application has to notify the server about the purchase and give the bought product to the player.

5. **Verify in-app purchase**
   During the previous use case (*Purchase in-app product*), the server verifies if the purchase is valid and not fake, canceled, or already accounted. The verification process is done through *Google API*.

3.2.5 Administration

Since the game is not static during its lifetime, people in charge of the changes needs an easy way to add new locations and maintain game objects. These use cases are described in Figure 3.6.

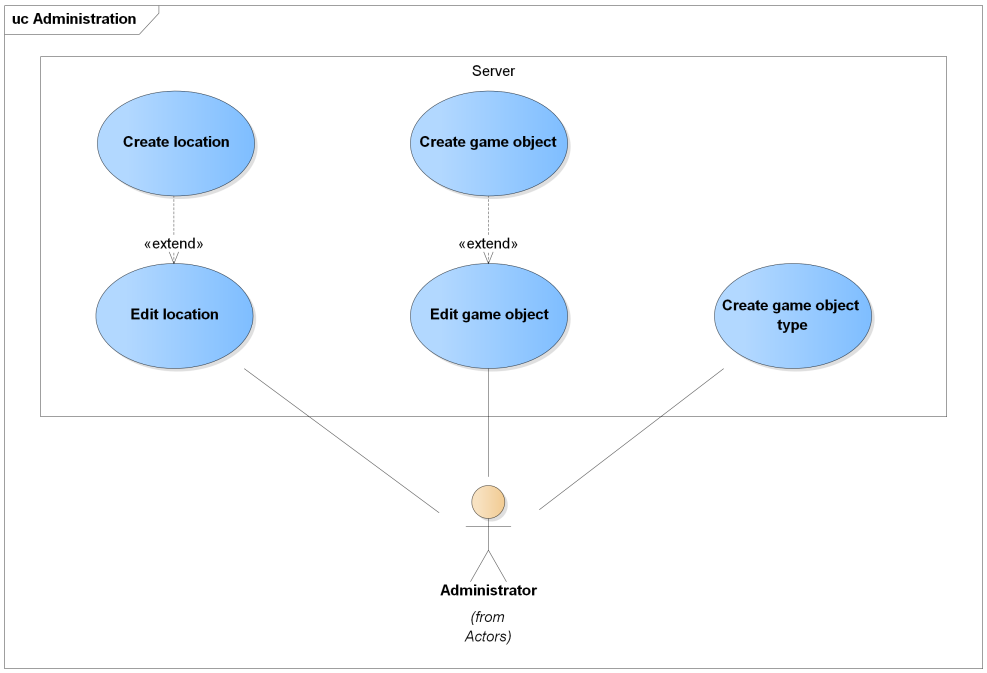

Figure 3.6: Use Cases: Administration

3.3 Requirements

After several discussions with my colleague, we agreed on the following requirements.

3.3.1 Functional Requirements

These requirements define functions of the server components.

Rules

These requirements specify the functionality not directly accessible to clients.

1. **The player's character has attributes**
   The character has a set of attributes, including health, experience, level, and owned gold. Maximum health increases with level. The experience is rewarded after certain actions, e.g. after killing a monster. The gold is primary in-game currency.

2. **A player can own items**
   A player has an inventory which can contain various types of items. The item can be for example sword, potion, armor etc.

3. **A game object has a type and inherits all its properties**
   The type of game object specifies allowed actions, its attributes, default name and description.

4. **A game object can be a monster**
   The monster can be killed but can also inflict damage to the player. It has its own inventory and there's a reward for killing the monster in a form of gold and experience.

5. **A game object can be a shop**
   The shop can contain several items with specified price.

6. **A game object can be an item**
   The item can be one of the many objects useful to a player. Examples of the items are health potion, sword, armor, necklace and similar.

7. **Each game object has its own inventory**
   The inventory contains other game objects. Example of this requirement is a monster with a potion and a sword in its inventory; both will be given to the player who kills the monster.

8. **The server stores a list of predefined locations**
   Real geographic locations for the game objects are stored on the server to ensure every player has the same location-object pair.

9. **A game object can independently exist at many locations**
   This requirement aims to help maintain the game objects efficiently by administrators. It allows creating small set of abstract game objects with predefined inventories and other attributes.

10. **If a player kills a monster at a location, the monster will be hidden for a period**
    To prevent the player from killing the same monster continuously without a need of moving somewhere else, the location should be hidden for a certain period after the kill.

11. **The server must persist player's profile between sessions**
    All the player's attributes, his inventory and equipment must be stored between sessions. Player will continue from the state in which he ended.

Features

These requirements contain the behavior directly visible to clients.

1. **The server must provide REST-like API to clients**
   The key requirement for the server is to allow receiving HTTP(S) requests. When processed, the server responds in JSON format.

2. **A player registers and logs in the game using Google account**
   For the player's convenience, a Google account is required to play. The server does not have to store or handle any password. Most of the authentication process is delegated to Google servers.

3. **A client can get nearby game objects based on his location**
   The major feature of this application is being location-aware. Server must provide a method to retrieve game locations near the requested latitude and longitude. The "near area" should be circular, defined by its radius; the size have to be carefully chosen so it's big enough to cover client's maps but also small to limit the response size and the spatial search overhead.

4. **A player can kill a monster**
   When the player wins the fight, he will be rewarded by experience and gold.

5. **A player can be killed be a monster**
   The player can lose health during the fight with a monster. If the health reaches zero, the player dies and loses an amount of gold based on his level.

6. **A player can collect items from the monster he killed**
   When the player wins the fight, he's offered to collect items from the monster's inventory. He can chose any subset of these items.

7. **A player can equip an item**
   Many items in the game can be equipped. These items have predefined equipment slot, for example a sword have to be held in hand, an armor worn on chest, shoes put on feet and so on.

8. **A player can buy object from a shop**
   Gold can be exchanged for various items in shops.

9. **A player can use an item from his inventory**
   Some items in the game are consumables. When used, an action defined by the item is executed. For example a health potion heals the player.

10. **A player can purchase in-app product**
    The application allow a user to exchange real-life currency for the in-game one. The server should verify such purchase and add the currency to his profile.

11. **The server should provide REST-like API for administration**
    Such API will be used to manage locations, create and edit game objects or to assign a game object to some locations. It is necessary to protect the administration endpoints from unauthorized access.

3.3.2 Non-functional Requirements

These requirements specify the criteria the application must meet.

1. **Database implements caching**
   This application is heavy database reliant. Additional cache provides optimization for faster reads.

2. **The communication between client and server parts of the application must be secure**
   All data sent from and to a client has to be encrypted. This should be achieved by connecting to the server via HTTPS.

3. **The server responsibilities are delegated to its components**
   The whole server consists of three components:

   — Connection server (*CS*)

   — Login server (*LS*)

   — Database server (*DS*)

   Only the Connection server is accessible to clients.

4. **The server components are scalable**
   Each component can run many instances if itself. These instances are mutually independent.

5. **The user is authenticated by Google**
   A user is authenticated using **Google Play Games** [7] on the client. The authentication is finished by verifying his ID token using **Google API Client Library**.

6. **The execution environment is Java 8**

7. **The operating system is Debian 8**
   Thanks to the portability of Java applications, other operating systems may be supported.

3.4 Technology

3.4.1 Database Management System

The database is a crucial part of the server. It handles most of the data persistence. Many commercial and open source solutions exist on market nowadays. In the following text, I will analyze three popular relational database management systems (DBMS) – MySQL, PostgreSQL, and Oracle. For a quick comparison, see Table 3.1. The described advantages and disadvantages are based on [8].

	MySQL	**Oracle**	**PostgreSQL**
Rank[1]	2	1	4
Initial Release	1995	1980	1989
License	GPLv2 Commercial	Commercial	BSD-like
Version[2]	5.7	12c	9.6
Replication methods	Master-master Master-slave	Master-master Master-slave	Master-slave
SQL Support	yes	yes	yes
Foreign keys	yes	yes	yes
JDBC Support	yes	yes	yes
In-memory capabilities	yes	yes	no

Table 3.1: Comparison of popular relational DBMS [9]

3.4.1.1 MySQL

MySQL [10] is an open source relational database management system. It is written mostly in C++ and C [11]. Currently developed by Oracle, MySQL is available to consumers in several editions. Free Community Edition is released under GPLv2 license[3]. Oracle also offers commercial Standard and Enterprise editions which include additional technical support, and monitoring and management tools [12]. This DBMS is very widely used and has a strong community base.

Community Edition includes graphic interface for database management. It also contains a user-friendly modeling tool which allows fast development of the database. The model can be converted to an SQL script and easily deployed on other MySQL servers.

The main advantages: open source, great community support, lightweight, good replication support, powerful management tools.

The main disadvantages: little performance optimization, limited security, issues with reliability.

[1]Description of the ranking methodology available at `https://db-engines.com/en/ranking_definition`

[2]Current stable version as of 2017/06/05

[3]GPLv2 license text is available at `https://www.gnu.org/licenses/gpl-2.0.html`

3.4.1.2 Oracle

Oracle Database [13] is an object-relational database management system. Developed by Oracle, it aims on enterprise-scale applications and is well suitable for large businesses. Offered editions include Standard Edition 2 and Enterprise Edition (EE) which can be further expanded by some additional services. The price of EE start at $950 per user[4]. This DBMS is very widely used in corporate environment.

The main advantages: suitable for large databases, well scalable, extensive backup mechanisms. The main disadvantages: closed source, free version has very limited feature set, expensive.

3.4.1.3 PostgreSQL

PostgreSQL [14], also known as Postgres, is an object-relational database management system. Developed by PostgreSQL Global Development Group, a community of many companies and individuals, PostgreSQL is an open source project licensed under permissive PostgreSQL License [5].

The main advantages: open source, advanced business/location analytics features, good reliability and data-integrity.

The main disadvantages: ill-suited for read-heavy application, poor replication.

3.4.1.4 Chosen Solution

All the presented solutions are similar and are well-supported by Java frameworks. I chose MySQL 5.7 Community Edition to be the database system for this server. I have good experience with MySQL environment and the cost of game development can be lower thanks to the free Community Edition.

3.4.2 Frameworks and Libraries

Frameworks provide developer with a powerful tool-box. The tools in this box help in many areas of application development. They provide essential design patterns and structure to the development project and also provide the backbone and container for the components to operate within [15].

I chose to use a Java framework which provide a support for Representational State Transfer (REST) API. Furthermore, I added an Object/Relational Mapping (ORM) framework which helps me to interconnect Java application and MySQL server.

[4] Oracle Technology Global Price List, June 1, 2017

[5] PostgreSQL License text is available at `https://opensource.org/licenses/PostgreSQL`

3.4.2.1 Dropwizard

Dropwizard [16] glues together many mature libraries which helps with developing a powerful web-application. The libaries most useful for this project include:

Jetty Powerful open source web server. Jetty is flexible and extensible. It is a lightweight server with small footprint [17].

Jersey An open source framework for developing RESTful Web Services in Java. It supports JAX-RS[6] APIs [18].

Jackson A JSON parser and generator. It supports a conversion from JSON to Plain Old Java Object (POJO) and vice versa [19].

3.4.2.2 Hibernate ORM

Hibernate ORM enables to more easily develop applications whose data outlives the application process. It is an Object/Relational Mapping framework and handles data persistence as it applies to relational database. Hibernate supports many powerful features like lazy initialization and numerous fetching strategies. It requires not special database tables and much of the SQL is generated at system initialization [20].

3.4.2.3 Google API Client Libraries

A library developed by Google to help developers integrate Google API to their applications [21]. The library is included in this project to handle user authentication and to verify in-app purchases.

3.4.3 Index and Cache

Redis is an open source in-memory data structure store [22]. I chose to use Redis as the second level cache provider for Hibernate. The additional cache should result in more optimized database calls and thus performance boost.

Since our application relies heavily on working with geo-spatial data, I decided to implement an additional index. Redis provides a native support for indexing and searching points near the selected spatial location.

[6] Specification of JAX-RS is available at `https://jcp.org/en/jsr/detail?id=311`

Chapter 4

Design

4.1 Components

The game server is divided into separate components. Many instances of a component can run at the same time. Component, and even instances, can be distributed among many machines; this feature renders the whole server easily scalable. Refer to Figure 4.1 for a component diagram.

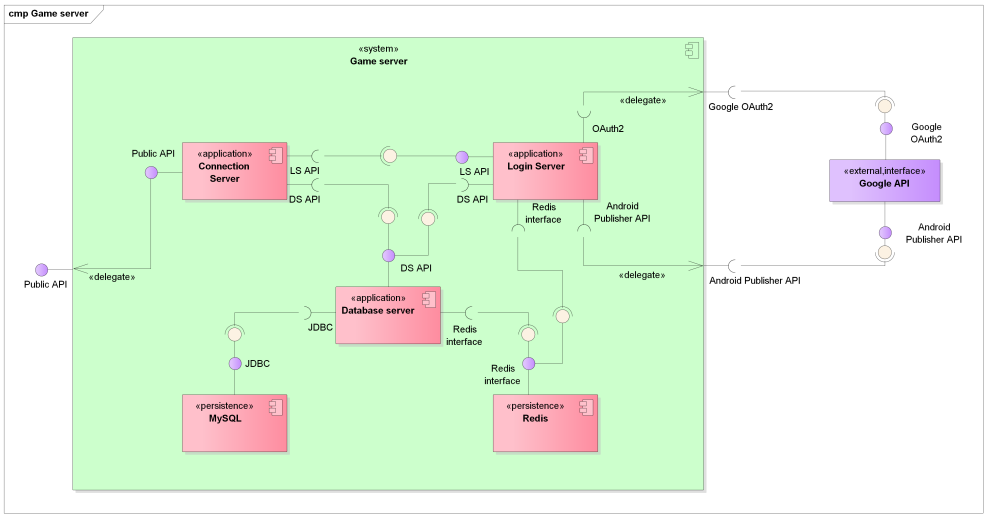

Figure 4.1: Component diagram of the game server

4.1.1 Connection Server

This is the public entry point and the only component exposed to clients. Reducing the number of publicly accessible components increases security of

the server. CS handles all incoming traffic and delegates work to other components.

4.1.2 Login Server

The main responsibility of the LS is to handle user authentication and authorization. It connects to Redis where users' access codes are stored. The LS is also used for in-app purchase verification. The component is connected to Google API.

4.1.3 Database Server

The most important component is Database server. Most of the game logic happens here. Since the DS is responsible for data persistence, it is connected to MySQL and Redis.

4.2 Activities

4.2.1 Authentication

Authentication process is visualized in Figure 4.2.

4.2.1.1 Access Code

A client is identified by his access code during a session. The code is random and unique. It is generated each time the client finishes login process; the old code, previously issued to the user, is invalidated.

4.2.1.2 Registration

Every user must register an account to gain access to the game. The user is asked to choose a unique username. If the username is already taken, the whole registration process has to be repeated. LS retrieves user's UID and his e-mail address and passes the information to the DS, which then creates and initializes a new user profile. Client is issued an access code and the authentication process is completed.

4.2.1.3 Login

A registered user can simply login using only his ID Token, which is provided to client during login process by Google Play Games. LS exchanges the token for UID which is then used to retrieve user's profile. Client is issued an access code and the authentication process is completed.

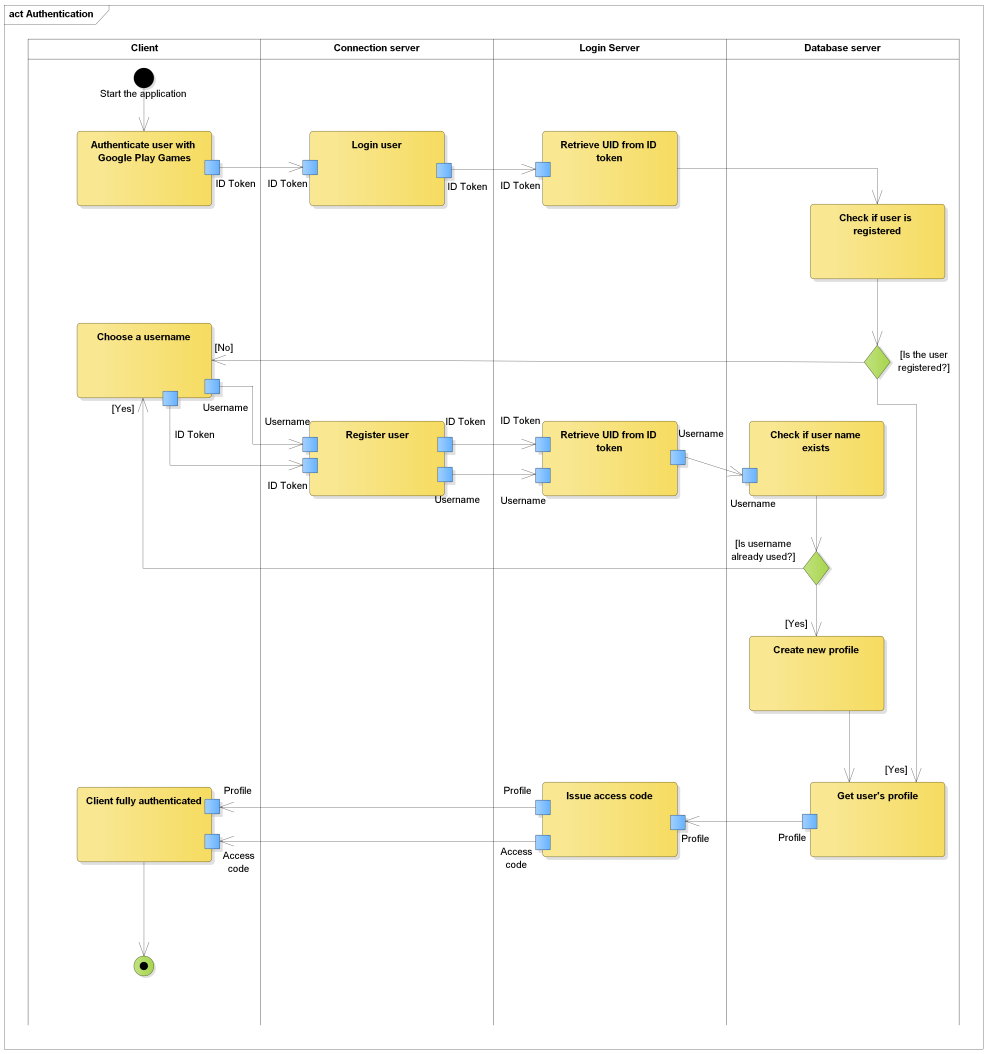

Figure 4.2: Activity diagram of the authentication process

4.2.2 Killing a Monster

A user decides to fight a *monster*. In this prototype, the client iteratively deals damage to the user and the *monster*. The damage is calculated as *attackDamage* attribute multiplied by a random value 0.5-1.5. If *health* of the *monster* drops to below zero before the user's one does, the *monster* is killed. Server is notified of the result and rewards the user with *gold* and *experience*. The kill is logged, the *monster* is removed from the game map and the client is provided with a one-time *killConfirmationCode* which allows him to collect loot from the *monster*.

4.2.3 Killing a User

If user's *health* drops to or below before the *monster*'s one, the user dies. Client notifies the server of user's death. His *health* is fully restored and he's punished with *deathPenalty* which is deducted from his gold:

$$deathPenalty = 200 + 100 * (userLevel - 1)$$

4.2.4 Collecting Loot after Kill

User is presented with an option to collect loot from a *monster* he killed. Clients sends a *killConfirmationCode* along with a list of items, he wants to collect, to the server. DS consumes the *killConfirmationCode* and adds the selected items to user's inventory. Client then receives the updated inventory.

4.2.5 Buying an Item

Client shows its user a *shop* and lets him choose what item he wants to buy. The selected item is sent to the server. DS verifies the item is in the specified *shop*. Price of the item is then deducted from user's account and the item is added to his inventory. If the user does not have enough *gold*, the purchase is rejected and an error message is returned. Otherwise, client receives updated user's inventory.

4.2.6 Equipping an Item

User selects a *slot* and an appropriate item from his inventory. The client sends the item along with the slot to the server. DS checks if the item-slot pair is correct and assigns the item to the *slot*. The successful result is then confirmed to the client.

4.2.7 Using an Item

User selects a usable item from his inventory. Client sends the selected item to the server. DS decides what the item does by looking at attributes *addHealth*, *addExperience*, and *addGold*. Based o the values set for the item, health, experience, and/or gold is added to user's account. The updated profile is then sent back to the client.

4.2.8 Purchasing In-app Product

In-app purchases are handled on client which performs the transaction using a Google service. The prototype currently supports only buying *gold*. If the transaction is successful, client sends a Google's *Purchase token* to the server. LS verifies the status of the purchase using **Android Publisher API** [23]

and adds the *gold* to user's account. The purchase is then confirmed to the client.

4.2.9 Retrieving Nearby Game Objects

Client visualizes nearby *game objects* on the map. Since the user moves, the client frequently retrieves new *game objects* based on the actual location, making high demands on the speed of the retrieval process.

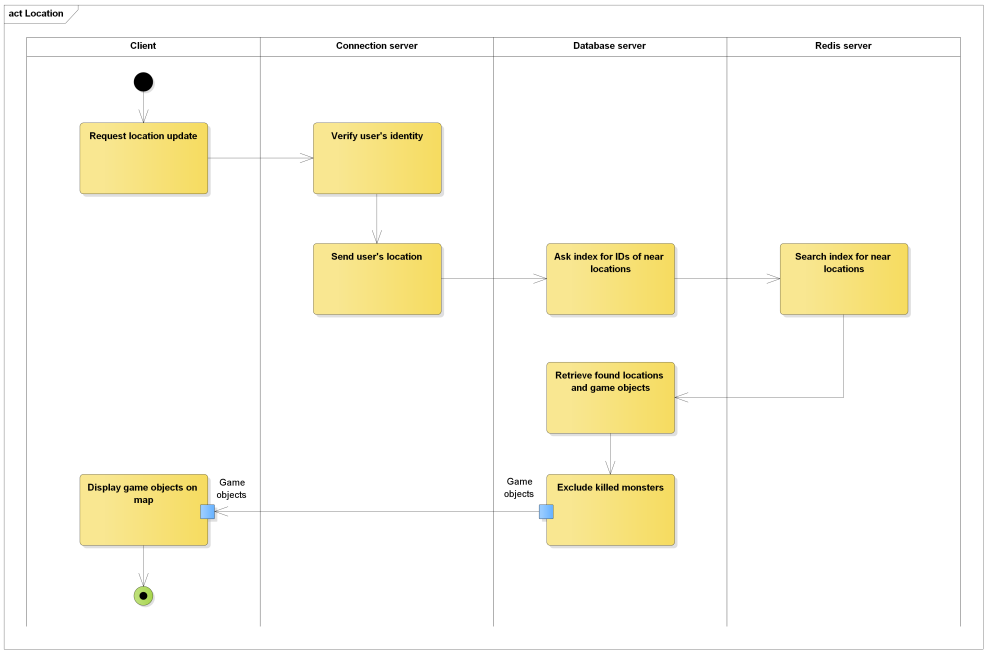

Figure 4.3: Activity diagram of how the server provides nearby game objects

Client sends its coordinates to the server. DS queries Redis which contains an index of all available *locations*. The index responds with IDs of *locations* in 200 m radius from the provided coordinates. The *locations* and their assigned *game objects* are then retrieved from the database. Server excludes already killed *monsters*. The list of *locations* and their *game objects* is sent to the client which presents them on the map. See an activity diagram of the described process in Figure 4.3.

4.3 Database Model

The database was designed to comply with the requirements specified in section 3.3. The entire database model is shown in Figure 4.4. In the following text, I will describe important tables.

user User's profile. It contains *e-mail*, *UID*, and *username*. Quality attributes of the profile are also stored here - current *health*, *experience*, and *gold*.

user_inventory Items a user own.

user_equipment Information about what items a user has equipped. Is in 1:1 relationship with *user*. Each column represents an equipment *slot*. When an item is equipped, the item's entry from *user_inventory* to its *slot*.

game_object_type A "recipe" for every object in the game. Name and description of an object is stored here. The *game_object_type* has linked attributes and actions.

action Defines all allowed *actions*. Each *game object type* implements their subset.

game_object_type_attribute Defines all *attributes* of a *game object*. The attribute is identified by its *name* and can have a *value*.

game_object An implementation of *game object type* which can be assigned to a *location*.

game_object_content Inventory of a *game object*

location Predefined real-world locations. Each one must have *latitude* and *longitude* defined; *location* can have a *game object* assigned.

action_log Log of user's actions. In prototype, it is used only for kills to allow collecting loot.

user_exclude_location Locations at which a user killed a monster. The table is periodically flushed.

4.4 Administration

The *Database server* supports several API endpoints through which authorized administrators can manage the game data. The *Admin* section is protected using **HTTP Basic Authentication** and thus the administrator has to know a valid username-password combination. For prototyping purposes, I've decided to provide only basic functionality for the administration.

4.4.1 Game Object Type Management

Administrator can create new *game object types* by sending a POST request to the endpoint */admin/gameObjectType*. A unique *name* has to be specified for the new type. Optionally, the type can include a *description*, a set of allowed *actions* and *attributes*.

Existing *game object types* can be updated. It is possible to change all their properties like *name* and *description*. Administrator can also add new *attributes* and *actions*. This is done using PUT request to the endpoint */admin/gameObjectType*

All existing *game object types* can be retrieved along with their *actions* and *attributes* by sending a GET request to the endpoint */admin/gameObjectType*

4.4.2 Game Object Management

Administrator can create new *game objects* by sending a POST request to the endpoint */admin/gameObject*. *Game object type* have to be specified for the new *game object*. Optionally, the new object can have a set of children which can be later updated by calling PUT */admin/gameObject*.

All existing *game objects* can be retrieved along with their children by sending a GET request to the endpoint */admin/gameObject*

4.4.3 Location Import

New locations can be imported from a file in OSM XML [24]. Administrator can do so be sending a POST request to */admin/importLocations*. The file is parsed for latitude and longitude data nad the new locations are inserted into the database.

4.4.4 Game Object To a Location Assignment

A location serves no purpose without a*game object* assigned to it. This can be achieved by using PUT endpoint */admin/assignGameObjectToLocation*.

4.4.5 Cache Clearing

During the development process, developers might need to change game data by accessing the database directly. Since Hibernate won't be aware of such changes, its cache must be cleared via DELETE endpoint*/admin/clearCache*.

4.5 Calculations

4.5.1 User's Level

$$level = \left\lfloor \left(\frac{xp}{1024} \right)^{0.62} + 1 \right\rfloor,$$

where xp is user's experience.

4.5.2 Maximum Health

Player's cannot exceed a certain value which linearly scales with his level.

$$maxHealth = 200 + 50 * (userLevel - 1)$$

4.5.3 Death Penalty

A player is punished by losing gold when he dies. The total amount of the lost gold scales with user's level.

$$deathPenalty = 200 + 100 * (userLevel - 1)$$

4.6 Public API

Even though every component has its own API, only the Connection Server API is available to clients. HTTP methods correspond to REST principles.

The public API for the prototype has been specified in cooperation with my colleague Tomáš Zahálka. Below are shortly described selected endpoints. For the full list of publicly available API enpoints, please refer to Appendix B. The mentioned appendix also contains description of API provided by LS and DS.

4.6.1 GET /login

The Login endpoint verifies the Google ID token and generates an *Access Code* for identification. When successfully authenticated, the user's profile and the *Access Code* are returned.

Parameters

token the *Google ID token*

Response

200 Successfully logged in, player's profile and the *Access Code* are returned.

403 Invalid token.

404 User not found, registration needed.

4.6.2 POST /purchase

The Purchase endpoint offers support for in-app purchases. The purchase is verified and then assigned to the player. To be accepted, it cannot be cancelled or consumed.

Parameters

accessCode the *Access Code*

productId the id of the product to buy

token the purchase token

Response

200 Player's profile.

400 Invalid data.

403 Invalid *Access Code* or the purchase is not valid.

404 User not found, registration needed.

500 Unexpected error.

4.6.3 GET /location

This retrieves all nearby locations in a 200 m radius from the provided coordinates. The locations are returned along with their associated objects.

Parameters

lat the latitude

lon the longitude

accessCode the *Access Code*

Response

200 List of nearby locations with game objects assigned to them.

500 Unexpected error.

4.6.4 POST /action/kill

The kill action is performed on the selected object and location. The location is temporarily excluded from future requests to /location. The player's health is updated, experience and gold are added. It returns a *killConfirmedCode* which is needed to perform the collect action.

Parameters

accessCode the *Access Code*

locationid the id of the location

gameObjectId the id of the game object

health the new player's health

Response

200 One-time code for kill confirmation.

400 Invalid data.

403 Invalid *Access Code*.

404 User not found, registration needed.

500 Unexpected error.

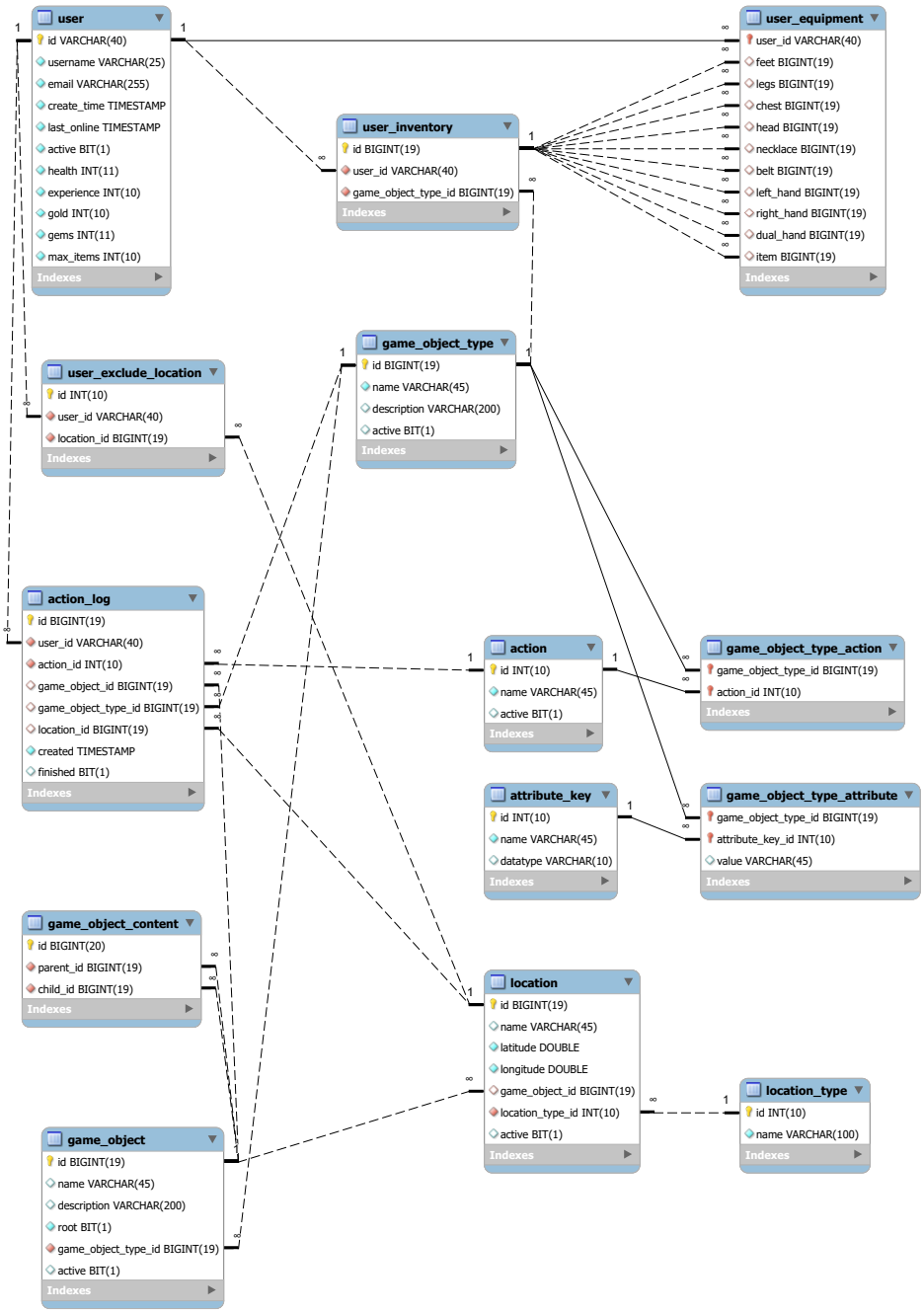

Figure 4.4: Database model

Chapter 5

Implementation

5.1 Development Environment

I chose to use IntelliJ IDEA Ultimate 2017 [25] as my IDE. It is very user-friendly and powerful tool, which packs almost everything needed to develop a Java application. Build management is handled by Apache Maven [26]. This tool is extremely useful as it takes care of all application's dependencies and completely manages the build process. I also used GIT [27] as my version control system.

5.2 Game Locations Source

I have obtained all game locations from open-source project OpenStreetMaps (OSM) [28]. I downloaded complete map data of the Czech Republic. All features in OSM have one or more tags which specify a type of the feature, for example *amenity.college* is a college or a campus building, *historic.castle* is a castle and so on. Since it would be unreasonable to use all available features[7], I chose only several types, mostly from categories *amenity* and *historic*. I used a tool *Osmosis* [29] to extract selected map features from the data. The selection resulted in 99 037 locations for the entire Czech Republic.

5.3 Database

The initial database structure was created from the database model in Figure 4.4. I used MySQL Workbench [30] to generate a creation script.

The database includes an event which is triggered every 8 hours and wipes a table containing list of monsters killed by users. It causes the monsters to "re-spawn".

[7] The list of the feature types is available at `http://wiki.openstreetmap.org/wiki/Map_Features`

5.4 Project Structure

The entire project is organized by its components (also called *modules* in the IDEA's terminology). The project name is *BachelorsServer* and consists of three modules – *ConnectionServer*, *LoginServer*, and *DatabaseServer*. Each module follows Maven's Standard Directory Layout[8]. Top-level directory contains important configuration file. First one is *config.yml* which stores server setting, such as listening ports, used protocol (HTTP/HTTPS), or URLs of other components. Path to this file must be specified as run argument of the server. Second file is *pom.xml* (*Project Object Model* file). It contains project-specific definitions for Maven. It specifies project version and name, its dependencies, and build strategies.

Common package organization is[9]:

bachelors.*module* Main class and server configuration classes.

bachelors.*module*.api Classes used for JSON serialization and deserialization.

bachelors.*module*.resources Definitions of API endpoints.

5.5 Connection Server

A component which mostly serves as a proxy. It's implementation should be effective, since every client must connect to this component. Connection Server is the least complex module of the three. To satisfy the requirement for secured communication, this component allows incoming connection only using HTTPS.

5.5.1 Resources

Classes which handle API requests. I described their main responsibilities and examples of their usage in the following text.

BaseResource Abstract super-class for all resources. It contains commonly used methods, such as *putRequest()*, or *getRequest()*. These methods verify client's access code and delegates the request to a supplied URL.

UserResource The class handles user-oriented requests. It is used to get user's profile, or his inventory.

LoginResource The resource handles login and registration requests.

[8]Described at `https://maven.apache.org/guides/introduction/introduction-to-the-standard-directory-layout.html`

[9]The package name *module* is used as a placeholder for module-specific name.

LocationResource It is responsible for retrieving nearby game locations.

PurchaseResource It is responsible for in-app purchases.

ActionResource Important resource which processes user's action. It is used to kill a monster, to buy an item from a shop, or to use a health potion.

5.6 Login Server

A component responsible for authentication and authorization of clients and in-app purchase verification. I introduced several dependencies to help me fulfill the requirements.

5.6.1 Access Key Store

All client's requests after login are authorized using an *access code*. The storage for these codes has to be fast, reliable and shared among all instances of Login Server component. The codes are stored in Redis and accessed from the component using a Redis java client – Jedis [31].

5.6.2 Google API

During the authentication process user sends a Google ID token. The component uses a Google API Client library [21] to access the API and exchange the token for user ID and e-mail. Similar process is the in-app purchase verification. The task is done using Google Play Developer API Client Library for Java [32].

5.7 Database Server

The most complex and important component of the three. It is responsible for game logic and data persistence.

5.7.1 Configuration Files

A folder *DatabaseServer/src/main/resources* contains configuration files for various dependencies of Database Server.

hibernate.cfg.xml Hibernate's configuraton. It defines second-level cache settings, type of the database engine, and lists database entities.

hibernate-redis.properties Setup of Hibernate's second-level cache provider.

logback.xml Setting of logger. Specifies what log level to show and where to output the logs.

redisson.yml Configuration for Redisson which describes Redis server connection settings.

5.7.2 Hibernate

The component uses Hibernate to access the database.

5.7.2.1 Entities

Every table in the database (except decomposed Many-to-Many relationships) has to be defined in an entity class. The specification contains not only table columns but also other entities in relationship with the class. It means the developer can for example simply call *getGameObjectType()* on *GameObjectEntity* and Hibernate automatically fetches associated *GameObjectType* from the database.

5.7.2.2 Second-level cache

I've decided to use a second-level cache to optimize database interactions. Hibernate supports many cache providers and I chose to use hibernate-redis [33]. When the cache is configured, each entity can be annotated as *@Cacheable* and have caching strategy specified. Hibernate then automatically caches the annotated entities.

5.7.3 Models

Package *bachelors.database.db* contains model in which database operations are implemented. Each model extends abstract super-class *BaseModel* and operates in its logical scope, e.g. *UserModel* handles User-related operations, *GamObjectObjectModel* handles GameObject-related operations and so on. The *BaseModel* implements generic methods for simple database operations, like select all objects, or get object by id. Please refer to class diagram in Figure XXX or to project documentation for more detailed information about models.

5.7.4 HTTP Basic Authentication

Admin endpoints require a user to authenticate. Database Server use Dropwizard's authentication support. Only three additional files are needed to implement HTTP Basic Authentication. They are all located in package *bachelors.database.security*.

AdminUser Type of the user which extends *java.security.Principal*.

AdminAuthenticator The file defines how the application verifies user's username and password.

AdminAuthorizer A method to verify if the user has sufficient privileges to access the requested resource.

5.8 Deployment Environment

The prototype currently runs on a virtual private server provided by WEDOS internet, a.s. The server runs on Debian 8. It uses the lowest available server specification[10]:

CPU	1 thread Xeon 1.80 GHz
RAM	2 GB
SSD	15 GB
SLA	99.99%

The specifications are sufficient for testing and prototyping but in unused state, the server has only about 200 MB free RAM. Additionally, the application will have to support many concurrent users which requires more CPU threads.

[10] Offered server `https://hosting.wedos.com/cs/virtualni-servery.html`

Chapter 6

Testing

6.1 Unit Testing

The project use JUnit [34]. Unit tests are located in *src/test* directory. All components use the test to verify proper serialization and deserialization of JSON requests. Each tested JSON object is defined in *src/test/resources/fixtures*.

The tests are packed in a test suite. This allows running all test by executing *bachelors.connection.api.AllTests*. All tests are passing in the current build.

6.2 Static Code Analysis

I use SonarLint [35] for on-the fly static code analysis. It offers many useful rules with specified severity and also a description how to fix the issues. Static code analysis discovered many issues, the most notable were:

— Hide a private constructor to hide the implicit one in a static class.

— Use constant instead of duplicating string literal.

— Replace use of System.out by a logger.

— Make final constants also static.

The static analysis proved to be useful to maintain a good quality of the code and prevent bugs.

6.3 System Testing

I developed a Python script to test most of the endpoints in the real environment. It skips tests of */login/registration* and */purchase* as they need valid

Google tokens which can't be reused and are not easily obtainable. All tests are passing in the current build.

The script requires Python 3 or newer to run and is located in *BachelorsServer/SystemTest/TestBachelorsPrototype.py*. The tester must initialize the script with values of a tests account in the current database. The script then sends HTTP request to each endpoint testing valid and invalid data.

For example, test *ACTION – Kill* verifies the client cannot request setting his health to negative value or more than his level allows. Script also tries to kill a monster at a wrong location and vice versa. A successful kill follows a second attempt to kill the same monster which must fail.

6.4 Client Testing

I tested the application with my colleague using his client-part of the game. We didn't discover any bugs which would affect the player in any way.

6.5 Stress Testing

I performed a stress test using Apache JMeter [36]. This testing framework is used to simulate interactions with a web server. Results of this type of test offer useful insight into how many concurrent users can server handle. Hardware specifications of the testing environment are described in Section 5.8.

I created a test plan in a way, that one thread simulates about 10 users. Each thread operates independently for 10 minutes and performs following series of actions:

1. Wait 1 second[11]
2. Get nearby locations around location A (*/location*)
3. Wait 1 second
4. Get nearby locations around location B (*/location*)
5. Wait 1 second
6. Get nearby locations around location C (*/location*)
7. Get profile (*/user*)
8. Get inventory (*/user/inventory*)
9. Go back to step 1

[11]The locations are fetched from the server once every 10 seconds in real game client

Users	500	1 500	2 300	3 000	5 000
Latency median [ms]	108	123	166	473	1 737
Throughput [request/s]	16.0	46.4	67.9	79.1	99.8

Table 6.1: Result of the server stress test.

As expected, calls to the endpoint */location* took the longest time out of the three tested, the difference in latency was about 30% on average for 500 and 1 500 users. As you can see in Table 6.1, latency significantly increased at 3 000 users and about 2% request resulted in an error. At 5 000 users, the server was far beyond its limits and responded with very high latency and about 8% error rate. This behavior is expected due to the very limited resources of the server. The results might be affected by the performance issues of the test machine, since 500 threads had to be run concurrently to simulate 5 000 users.

Based on the test, server is currently able to handle more than 2 300 concurrent users with no performance issues that would affect clients.

Conclusion

This thesis aimed to create a prototype of the server part of a role-playing game with features of augmented reality. I explored and analyzed exiting similar games. After discussion with my colleague Tomáš Zahálka, I defined the rules and features of the game prototype.

I analyzed use cases and specified requirements to clarify expected server functionality. I explored available solutions for *database management system* and chose t o use *MySQL*. I decided to use several Java frameworks and libraries to handle database communication, JSON serialization, and manage API endpoints. I designed the structure of the server and described actions a user can perform. Based on these actions, I created a specification for public API as well as private one for internal communication among components. I designed and implemented a database model. I obtained and imported initial game locations to the database. In implementation phase, I created all specified API endpoints, implemented game logic a database communication. Administration section was secured and requires authentication. I created an index of location and configured a cache to improve database performance. Lastly I performed unit, system and stress tests as well as static code analysis.

Since the server is currently in prototype version, many features and further improvements are planed for future development. I plan to improve security, error handling and increase coverage of the unit tests before production release. One of the planned features is a quest system. My design allow easy scalability to support many concurrent users and high extensibility which enables me to create full, market-ready server.

Bibliography

1. *About PerBlue and Parallel Kingdom* [online] [visited on 2017-06-06]. Available from: http://www.parallelkingdom.com/Downloader.aspx?dk=PRESSPACK.
2. *Parallel Kingdom Reaches One Million Players* [online] [visited on 2017-06-06]. Available from: http://www.parallelkingdom.com/Media/PR/PR%5F013012%5FOne%5FMillion.pdf.
3. *Parallel Kingdom - Android game screenshot* [online] [visited on 2017-06-06]. Available from: http://www.parallelkingdom.com/img/theme/pages/pk_media/screenshots_page/pieces/screenshot_android_2_full.png.
4. *Ingress* [online] [visited on 2017-06-06]. Available from: https://play.google.com/store/apps/details?id=com.nianticproject.ingress.
5. *Pokémon GO hype is slowing but still making $2 million a day* [online] [visited on 2017-06-06]. Available from: http://www.ign.com/articles/2016/10/03/pokemon-go-hype-is-slowing-but-still-making-2-million-a-day.
6. *Pokémon GO* [online] [visited on 2017-06-06]. Available from: https://play.google.com/store/apps/details?id=com.nianticlabs.pokemongo.
7. *Google Play Games Services* [online] [visited on 2017-06-07]. Available from: https://developers.google.com/games/services/.
8. *Oracle vs. MySQL vs. SQL Server vs. PostGRESQL* [online] [visited on 2017-06-05]. Available from: https://www.libertycenterone.com/blog/oracle-vs-mysql-vs-sql-server-vs-postgresql-which-dbms-is-the-best-choice-for-you/.
9. *Knowledge Base of Relational and NoSQL Database Management Systems* [online] [visited on 2017-06-04]. Available from: https://db-engines.com/en/system/MySQL%3BOracle%3BPostgreSQL.

10. *MySQL* [online] [visited on 2017-06-05]. Available from: https://www.mysql.com/.
11. *MySQL Project Summary* [online] [visited on 2017-06-05]. Available from: https://www.openhub.net/p/mysql.
12. *MySQL Editions* [online] [visited on 2017-06-05]. Available from: https://www.mysql.com/products/.
13. *Oracle Database 12c* [online] [visited on 2017-06-05]. Available from: https://www.oracle.com/database/index.html.
14. *PostgreSQL* [online] [visited on 2017-06-05]. Available from: https://www.postgresql.org/.
15. NASH, Michael. *Java Frameworks and Components*. Cambridge University Press, 2003. ISBN 9780521520591.
16. *Dropwizard* [online] [visited on 2017-06-05]. Available from: http://www.dropwizard.io.
17. *Jetty* [online] [visited on 2017-06-05]. Available from: http://www.eclipse.org/jetty/.
18. *Jersey* [online] [visited on 2017-06-05]. Available from: https://jersey.github.io/.
19. *Jackson* [online] [visited on 2017-06-05]. Available from: https://github.com/FasterXML/jackson.
20. *Hibernate ORM* [online] [visited on 2017-06-05]. Available from: http://hibernate.org/orm/.
21. *Google API Client Library for Java* [online] [visited on 2017-06-05]. Available from: https://developers.google.com/api-client-library/java/.
22. *Redis* [online] [visited on 2017-06-05]. Available from: https://redis.io/.
23. *Google Android Publisher API* [online] [visited on 2017-06-07]. Available from: https://developers.google.com/android-publisher/api-ref/purchases/products.
24. *OpenStreetmMaps XML format* [online] [visited on 2017-06-15]. Available from: http://wiki.openstreetmap.org/wiki/OSM_XML.
25. *IntelliJ IDEA* [online] [visited on 2017-06-15]. Available from: https://www.jetbrains.com/idea/.
26. *Apache Maven* [online] [visited on 2017-06-15]. Available from: https://maven.apache.org/.
27. *GIT* [online] [visited on 2017-06-15]. Available from: https://git-scm.com/.
28. *OpenStreetMap* [online] [visited on 2017-06-15]. Available from: https://www.openstreetmap.org.

29. *Osmosis* [online] [visited on 2017-06-15]. Available from: http://wiki.openstreetmap.org/wiki/Osmosis.

30. *MySQL Workbench* [online] [visited on 2017-06-15]. Available from: https://www.mysql.com/products/workbench/.

31. *Jedis* [online] [visited on 2017-06-15]. Available from: https://github.com/xetorthio/jedis.

32. *Google Play Developer API Client Library for Java* [online] [visited on 2017-06-15]. Available from: https://developers.google.com/api-client-library/java/apis/androidpublisher/v2.

33. *hibernate-redis* [online] [visited on 2017-06-15]. Available from: https://github.com/debop/hibernate-redis.

34. *JUnit* [online] [visited on 2017-06-15]. Available from: http://junit.org/junit4/.

35. *SonarLint for IntelliJ IDEA* [online] [visited on 2017-06-15]. Available from: http://www.sonarlint.org/intellij/.

36. *Apache JMeter* [online] [visited on 2017-06-15]. Available from: http://jmeter.apache.org/.

Appendix A

Acronyms

CS Connection Server

LS Login Server

DS Database Server

ID Identifier

UID Unique Identifier

HTTP Hypertext Transfer Protocol

HTTPS Hypertext Transfer Protocol Secure

JSON JavaScript Object Notation

API Application Programming Interface

DBMS Database Management System

REST Representational State Transfer

IDE Integrated Development Environment

CPU Central Processing Unit

RAM Random-Access Memory

SLA Service Level Agreement

Appendix B

API

Appendix C

Class Diagrams

Appendix D

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