Metropolist

A Manuscript

Submitted to

the Department of Computer Science

and the Faculty of the

University of Wisconsin–La Crosse

La Crosse, Wisconsin

by

Shizheng Yang

in Partial Fulfillment of the

Requirements for the Degree of

Master of Software Engineering

May, 2019

Metropolist

By Shizheng Yang

We recommend acceptance of this manuscript in partial fulfillment of this candidate's requirements for the degree of Master of Software Engineering in Computer Science. The candidate has completed the oral examination requirement of the capstone project for the degree.

Prof. Kasi Periyasamy Examination Committee Chairperson	Date
Prof. Steven Senger Examination Committee Member	Date
Prof. Kenny Hunt Examination Committee Member	Date

Abstract

Yang, Shizheng, "Metropolist," Master of Software Engineering, May 2019, (Kenny Hunt, Ph.D.).

This manuscript describes the development of a web-based map generator, which allows the user to create randomly generated fantasy maps of cities and additionally allows the user to annotate and edit city elements at a fine granularity.

Acknowledgements

I would like to express my sincere appreciation to my project advisor Dr. Kenny Hunt for his invaluable guidance and untiring support. I would also like to express my thanks to the Department of Computer Science at the University of Wisconsin–La Crosse for providing the learning materials and computing environment for my project.

Table of Contents

List of Tables

List of Figures

Glossary

LaTeX

LaTeX is a document markup language and document preparation systems for the TeX typesetting program.

1. Introduction

1.1. Background

Generating content for computer games, CGI effects, feature films, print media, or Role-playing games is a significant bottleneck in terms of effort and resources. A typical game contains many thousands of audio files, images, textures, and 3D models. Procedurally generated content provides a cost-effective alternative to the manual creation of models, textures, images, and sound assets and can expand playability beyond what is otherwise possible. The video game "No Man's Sky," for example, was released in 2016 and relied on procedural asset generation to create over 18 quintillion planets each of which has a unique ecosystem composed of flora and fauna. Such a scale is, of course, beyond the reach of any manually created system. Figure 1 is a screenshot of a planetary terrain region of "No Man's Sky."

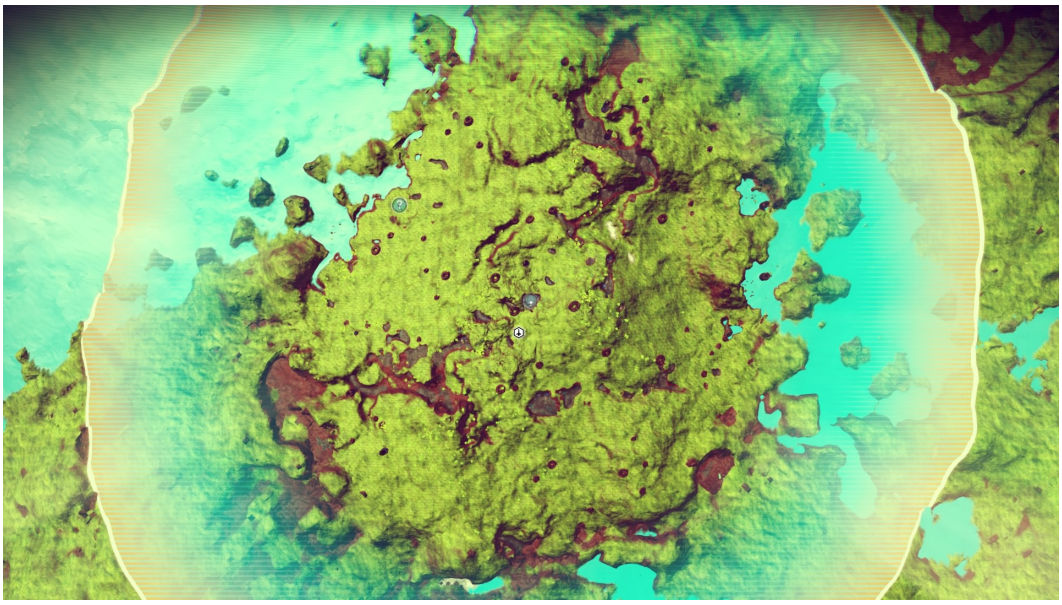

Figure 1. A screenshot of a planetary terrain region of "No Man's Sky"

1.2. Similar Systems

1.2.1. Minecraft

"Minecraft" is a computer game that can produce massive worlds composed of fine details, like elaborate cliff faces and waterfalls. Moreover, it relies on procedural generation, which automatically creates environments and objects that are at once random but guided by rules that maintain a consistent logic. Mountains are always rocky and sprinkled with snow, for example, while the low lands are typically full of grass and trees. Figure 2 is a screenshot of "Minecraft."

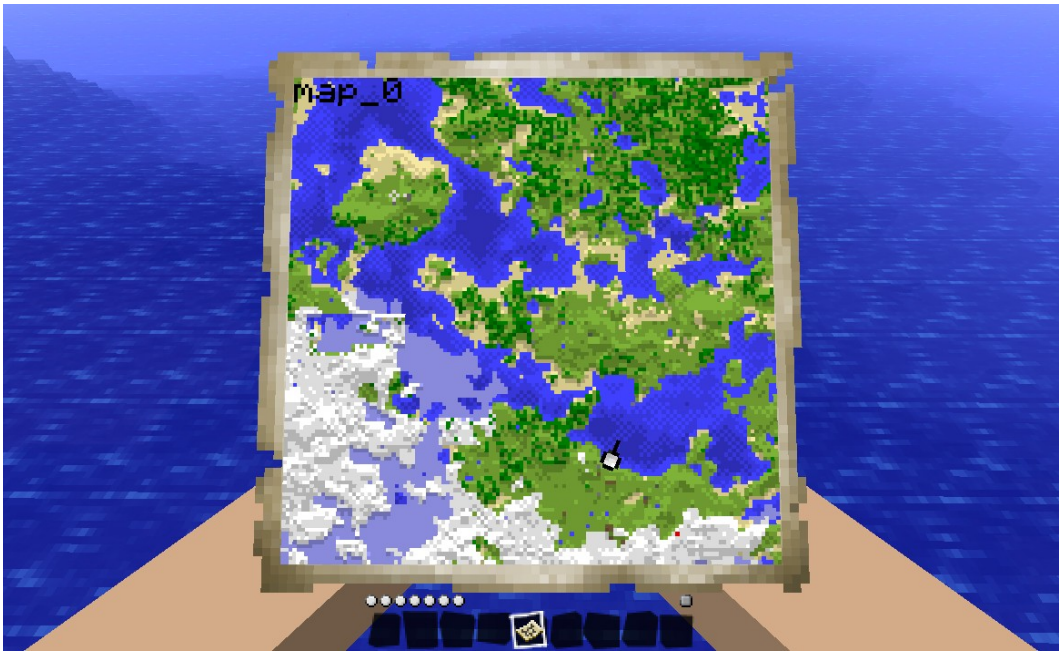

Figure 2. A screenshot of "Minecraft"

1.2.2. Medieval Fantasy City Generator(MFCG)

The "Medieval Fantasy City Generator(MFCG)" is a web application. This application generates a random medieval city layout of a requested size: small, medium or large, which is made up of different types of regions, and the generation method is stochastic. Furthermore, elements are provided for the user to add to the city, such as farm fields, citadel, plaza, temple, river, coast and so on. Because of the premise of medieval fantasy, the map always includes the walls and castle, but the user can decide whether to display them. It allows the user to edit the map to modify unsatisfying layers using a warping tool. The author also mentioned that the goal of the application is to produce a nice looking map, not an accurate model of a city. Finally, the user can save the map as an image in the "png" or "svg" format by using the export feature. Figure 3 is a screenshot of MFCG.

1.2.3. Azgaar's Fantasy Map Generator(FMG)

The "Azgaar's Fantasy Map Generator(FMG)" is another similar system, which is a larger scale world map. The size of the map made by FMG is not just an island, but a random fantasy map represents a pseudo-medieval world. Just like the real world, the map generated by FMG has constraints that the continent is always surrounded by the ocean and will never touch the border of the map. Although it is a randomly generated map, it is still based on real-world rules. While its most prominent feature is that the user can choose the type of map he likes, which provides the following 5 map types: political, cultural, height, biomes and pure landmass. It also supports user-defined map

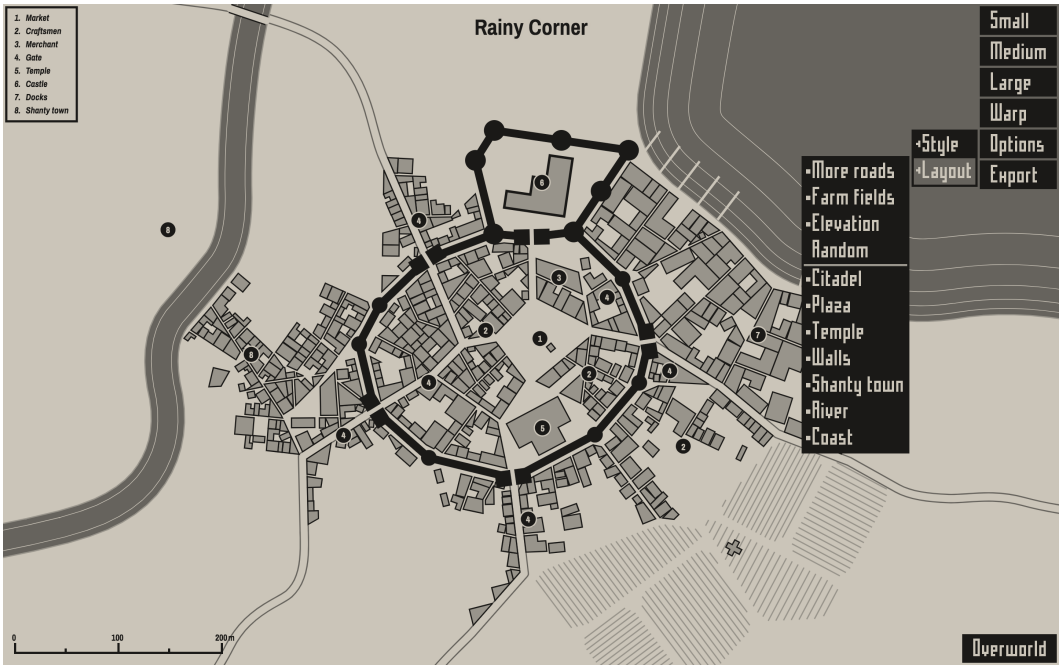

Figure 3. A screenshot of the Medieval Fantasy City Generator

type, and the user can add additional layers on existing map layers: rivers, temperature, and population. Also, it allows the user to annotate and edit the map using various such editors: layout, style, template, scale, countries, or cultural. It also supports exporting the map in the "png" or "svg" format, but unlike the previous application, if the user wants to come back to edit the map in the future, he can save it in the "map" format. Figure 4 is a screenshot of FMG.

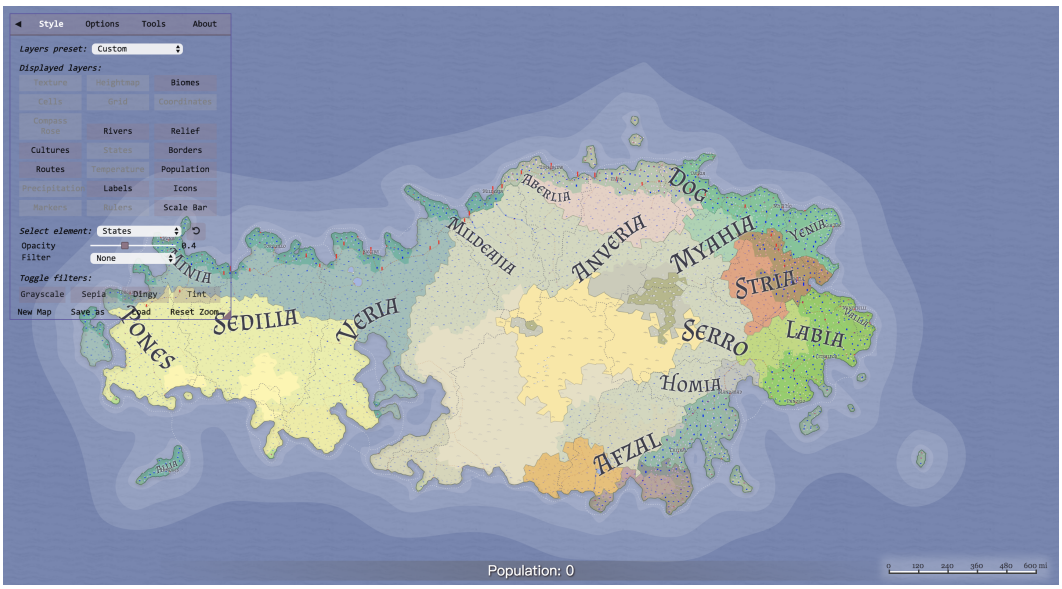

Figure 4. A screenshot of the Azgaar's Fantasy Map Generator

1.3. Project Goal

The goal of this project is to automatically generate city maps for use in Role-playing games (RPG) or Worldbuilding narratives. While the ultimate goal of this project is to procedurally replicate maps of a quality similar to the best cartographic hand-created maps by expert artists, we have obtained a modest approximation to the desired level of quality. Our system will have the essential features, such as: allowing the user to annotate or edit, allowing the user to export the map as an image in the "png" or "svg" format, allowing the user to save it to a database and retrieve it. Figure 5 is a screenshot of the current project.

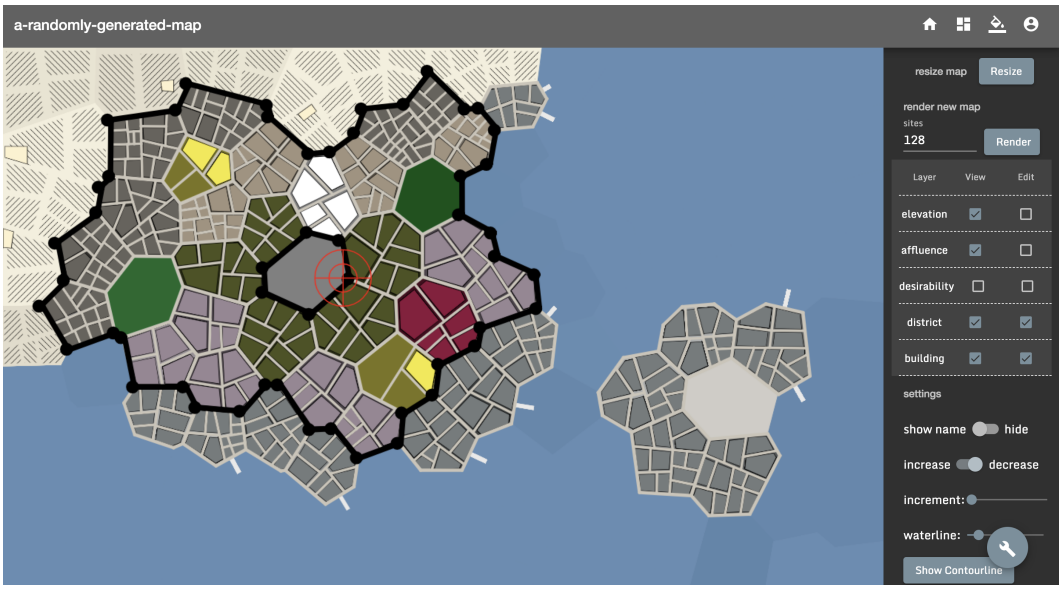

Figure 5. A screenshot of the current project

2. Requirements

Initially, this project was conceived by Dr. Kenny Hunt, who is served as the advisor, and almost all of the requirements were given by him, in an introductory meeting. Both the advisor, who is also the project sponsor, and the developer, who is also the author, are the stakeholders of the project.

After that, the advisor met with the developer almost every workday to gather informal functional requirements of this project, and he would also refine any previous requirements that were given. These requirements played a crucial role in the development of the map generator. On the one hand, the scope of development was clarified, and on the other hand, the necessary algorithms were compiled to maintain the continuation of development.

2.1. Functional Requirements

The functional requirements focus on developing a web application. There are two roles for this system: "admins" and "users". As a result, the following functional requirements were established for the project:

1. As an admin or a user, I need to be able to:
   (a) Log In: with unique email and password
   (b) Log Out: with ended session
   (c) Edit Personal Profile: change email, first name, last name, and password
2. As an admin, I want to be able to:
   (a) View All Users: on the "dashboard" page
   (b) Enable or Disable Users: allow me to log in or not
   (c) Search for Users: use any combination of email, first name, and last name; on the "dashboard" page
3. As a user, I want to be able to:
   (a) Sign Up: with unique email, first name, last name, long password, and confirmed password
   (b) View All Own Maps: on the "dashboard" page
   (c) View All Visible Maps: on the "community" page
   (d) Search for Maps: use any combination of the map name, created date, edited date, the owner's email, the owner's first name, and the owner's last name; on the "community" page
   (e) Download Maps: of either my own or other visible maps; in "png" or "svg" format

(f) Make Maps Public or Private: allow the map displaying on the "community" page or not

(g) Create Maps: with a name; on the "dashboard" page

(h) Delete Maps: on the "dashboard" page

(i) Edit Maps:

i. Open the map editor page, which shows an empty map with random map seeds by default, and then I can make maps by accessing the menu on the right

ii. Enter the number of map seeds to create a new empty map in the menu

iii. Select different types of layers for the map: elevation, affluence, desirability, district, and building, which can superimpose on each other

iv. Choose to display street names or not

v. Manipulate the "increase/decrease toggle," "increment sliding," and "waterline sliding" to edit the map: increase or decrease the value of elevation and affluence; build or remove the wall; make map cells as water or city

vi. If the waterline is changed, the continent will be changed accordingly

vii. Choose to display contour lines or not

viii. After selecting one or more than one layers under "edit" mode, I can change the size of the "soft brush" by using the mouse wheel, which determines the area where the map will be edited

ix. Edit the map by clicking and dragging the "soft brush."

x. The districts and buildings are procedurally generated

xi. I am allowed to change the type of the current district by right-clicking and selecting a new type from the context menu

xii. The map provides 13 different types of districts: rich, medium, poor, empty, plaza, park, farm, water, harbor, university, religious, castle, and military

xiii. Zoom in and zoom out the map by pressing on the alt key and using the mouse wheel

xiv. Use a specialized button to resize the map on the right top

xv. Save the map or download it by clicking the button on the right bottom

Figure 6 is the use case diagram that describes the functionalities to be implemented by this web application.

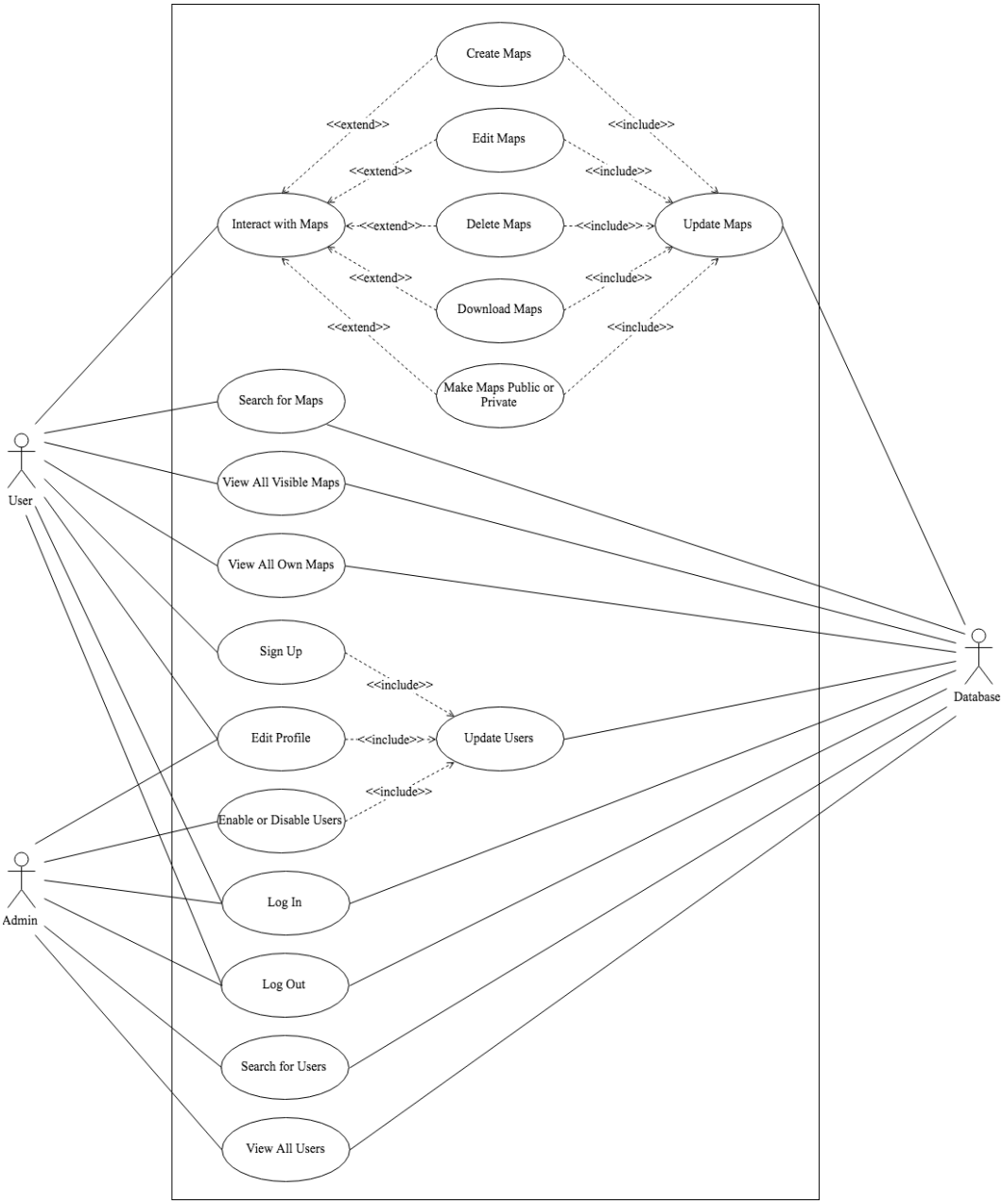

Figure 6. Use Case Diagram

2.2. Non-Functional Requirements

There are numerous non-functional requirements of a system: response time, availability, usability, security (authentication, authorization, integrity, privacy, etc.), and so on. For this application, we focused on the following requirements:

1. Security:
   (a) Input Validation:
      i. Add validations at both client and server sides
      ii. All methods should always validate all parameters in the server-side
   (b) Password Encryption:
      i. Shall always be encrypted
      ii. Not even the system administrators shall see clear text passwords
      iii. Applying a hashing algorithm to passwords
   (c) Prohibiting Cross Site Scripting (XSS):
      i. Never insert data anywhere in a "script" element
      ii. Never insert data in an HTML comment
      iii. Never insert data anywhere in CSS
      iv. Never insert data in an attribute name
      v. Never insert data in an attribute value
      vi. Never insert data in a tag name
   (d) Secure Session Cookie:
      i. Set the "Secure" cookie attribute and instruct web browsers to send the cookie only over encrypted, e.g., HTTPS, links.
      ii. Set the "HttpOnly" attribute true and prohibiting the JavaScript code from reading the cookie via the DOM "document.cookie" JavaScript object.
      iii. Always change the session id after login
2. Performance:
   (a) High performance of rendering map:
      i. Compare and select the most appropriate and the fastest way to render maps, e.g., canvas, SVG

2.3. Selection of Software Development Life Cycle Model

We analyzed and summarized the following possible risks:

1. Lack of experience developing in a map generation
2. The certainty whether a third-party library was available as the map rendering engine
3. The potential misunderstanding between the advisor and the developer with respect to the requirements

To reduce or avoid these risks, two life cycle models were considered: waterfall and agile. Because of our development model is flexible and the frequent communication between the sponsor and the developer, the waterfall model, is beyond our consideration. Unlike the waterfall model, we only need initial planning to start this project, and every time the new requirements are almost based on the previous one. Finally, the agile model was chosen, which is shown in Figure 7.

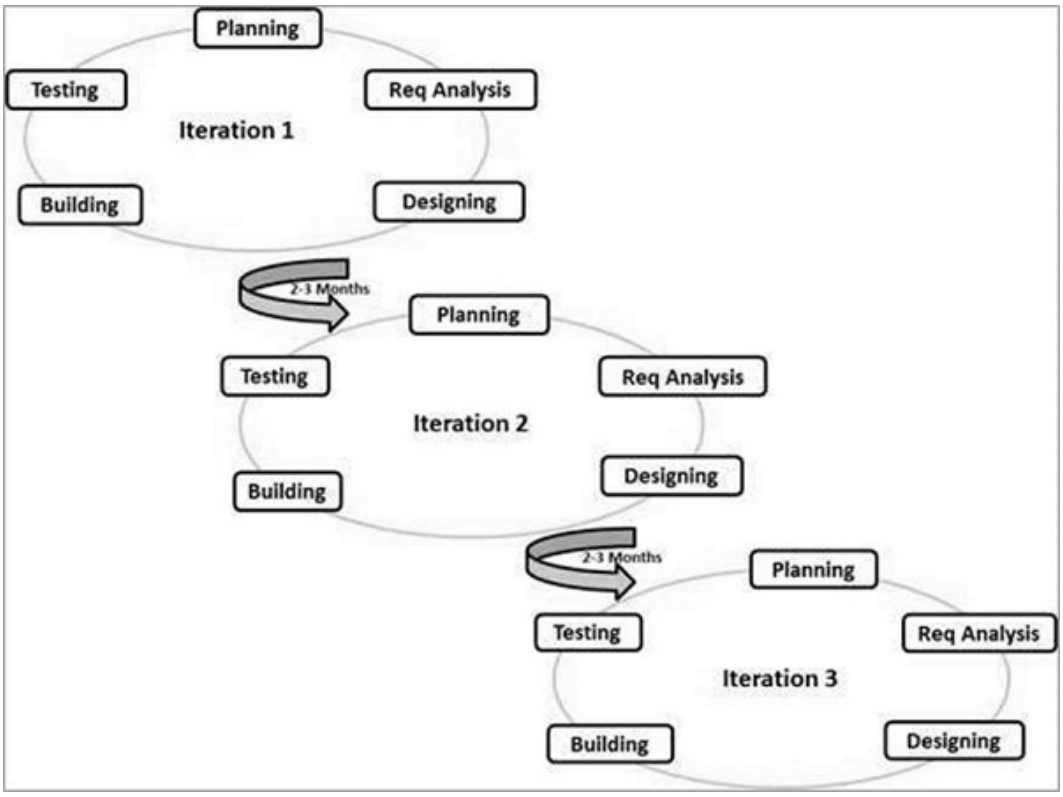

Figure 7. Agile Development Life Cycle Model

An agile methodology is a practice that helps continuous iterations of development and testing in the software development process. In this model, development and testing activities are concurrent. Furthermore, the agile model is a combination of iterative and incremental process models, which breaks the

product into small incremental builds. These builds are provided in iterations. Each iteration typically lasts from about one to three weeks. The iterations that have occurred in this project are listed below:

1. Iteration 1: Made the map generator based on the force-directed graph
2. Iteration 2: Abandoned the previous map generator based on force-directed simulation, and transformed the map generator that completely procedurally generates maps into a map generator that requires partial manual editing

At the end of each iteration, a working product demo is displayed to stakeholders. There are several known advantages and disadvantages of using the agile model in this project:

1. Advantages:
   (a) Easy to manage
   (b) Little or no planning required
   (c) Gives flexibility to developers
   (d) Resource requirements are minimum
   (e) Suitable for fixed or changing requirements
2. Disadvantages:
   (a) More risks of sustainability, maintainability, and extensibility
   (b) An overall plan, an agile leader (the advisor) is a must without which it will not work
   (c) There is a very high individual dependency since there is minimum documentation generated

3. Design

3.1. Architecture Design

Because we selected the web application model from the beginning, so the classic client-server architecture was quickly adopted, which is a distributed application structure that partitions tasks or workloads between the providers of a resource or service, called servers, and service requesters, called clients. In theory, any device connected to the Internet can try to access and obtain the resources or services of the server while the server is running normally. Figure 8 shows how it works.

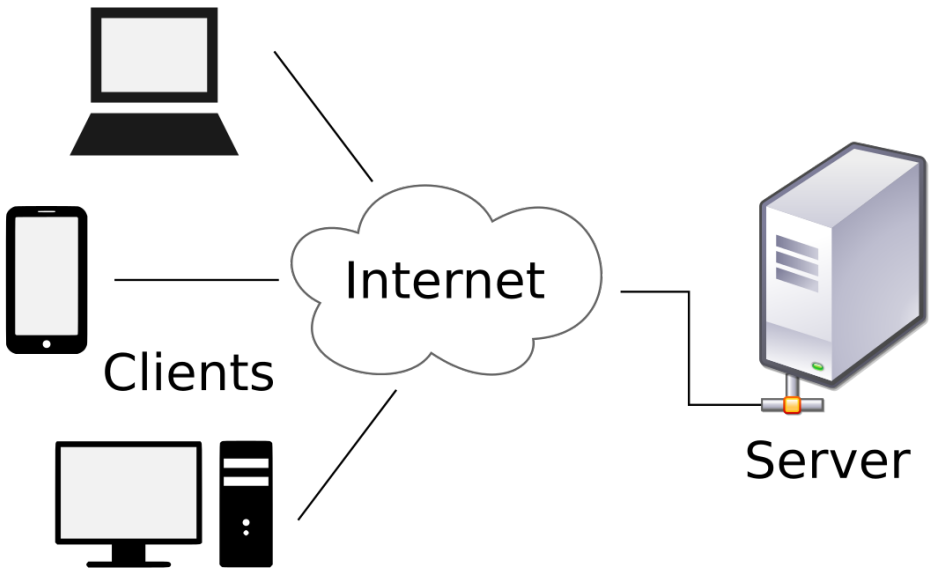

Figure 8. the Architecture of Client-Server Model

Based on the client-server structure, we have detailed its component structures below:

3.1.1. Web Browser or Client

The web browser or client is the interface rendition of a web app functionality, with which the user interacts with. This content delivered to the client can be developed using HTML, JavaScript, and CSS and doesn't require operating system related adaptations. In essence, the web browser or client manages how end users interact with the application.

There are three, well-known Web Application Architecture types available in the modern tech landscape: Single Page Applications (SPA), Microservices, and Serverless Architectures. Considered the needs of the project and the choice of the Software Development Life Cycle (SDLC) model, we chose the SPA architecture.

The SPA model interacts with the user in a more dynamic fashion by providing updated content within the current page, rather than loading entirely new pages from the server with each action from the user. It helps prevent interruptions in the user experience, transforming the behavior of the application such that it resembles a traditional desktop application.

3.1.2. Web Application Server

The web application server manages business logic and data persistence and can be built using PHP, Python, Java, Ruby, .NET, Node.js, among other languages. It's comprised of at least a centralized hub or control center to support multi-layer applications. The essential purpose of a web server architecture is to complete requests made by clients for a website. The clients are typically browsers and mobile apps that make requests using secure HTTPs protocol, either for page resources or a REST API.

Since the focus of this project is on the client side (frontend), which is the map generator (written in JavaScript), so we chose the Node.js Express framework. Moreover, the Node.js is written using JavaScript and is the same technology as frontend components, which makes it easier for the developer to program backend services and frontend user interfaces. It also provides consistency, code sharing and reusability, simple knowledge-transfer, and a large number of free tools. These benefits bring flexibility and efficiency when building this project.

3.1.3. Database Server

The database server provides and stores relevant data for the application. Additionally, it may also supply the business logic and other information that is managed by the web application server. There are multiple popular database systems available: Oracle, MySQL, Microsoft SQL Server, PostgreSQL, MongoDB, MariaDB, DB2, and SAP HANA.

Because of the project involves a small number of entities and the relationships among entities are not complicated, and the choice of the Software Development Life Cycle (SDLC) model, we chose the MongoDB, which is dynamic, flexible and easy to get started. It also provides high performance, high availability, and high scalability. What's more, our main purpose is to save maps and implement basic CRUD (create, retrieve, update, delete) operations of the database. As MongoDB is a schema-less database (written in C++), we can serialize the map data to JSON, send it to MongoDB and then save it.

3.2. Database Design

NoSQL, which stands for "not only SQL," is an approach to database design that can accommodate a wide variety of data models, including key-value, document, columnar and graph formats. MongoDB is a type of NoSQL

database. A record in MongoDB is a document, which is a data structure composed of field and value pairs. MongoDB documents are similar to JSON objects, the values of fields may include other documents, arrays, and arrays of documents.

Firstly, we thought there should be two entities: the user and the map. A user can have many maps, but a map can only belong to one user. Thus, the relationship between the user and the map should be One-to-Many.

MongoDB provides two ways of data modeling: Embedded Data Modeling and Normalized Data Modeling. Using Embedded Data Modeling, we may embed related data in a single structure or document, which means we should store maps in the user schema. But the data of the map is relatively large, in general, the CRUD operations will be slower, and the map cannot exist as a separate entity, so this one is beyond our consideration.

Normalized Data Modeling provides One-to-One relationship and One-to-Many relationship, here we chose the Normalized One-to-Many structure.

Figure 9 shows the relationship between the "user" and the "map".

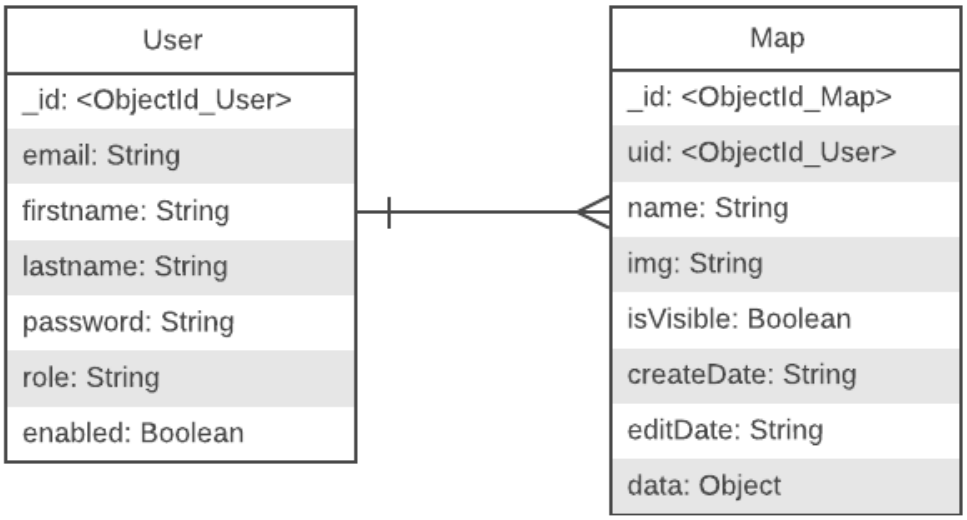

Figure 9. ER Diagram

3.3. REST API Design

REST is the acronym for Representational State Transfer. It is an architectural style for distributed hypermedia systems and was first presented by Roy Fielding in 2000 in his famous dissertation "https://www.ics.uci.edu/ fielding/pubs/dissertation/rest

Like any other architectural style, REST also does have it's own 6 guiding constraints which must be satisfied if an interface needs to be referred as RESTful. These principles are listed below:

1. Client–server - By separating the user interface concerns from the data storage concerns, we improve the portability of the user interface across multiple platforms and improve scalability by simplifying the server components

2. Stateless - Each request from a client to server must contain all of the information necessary to understand the request, and cannot take advantage of any stored context on the server. Session state is therefore kept entirely on the client

3. Cacheable – Cache constraints require that the data within a response to a request be implicitly or explicitly labeled as cacheable or non-cacheable. If a response is cacheable, then a client cache is given the right to reuse that response data for later, equivalent requests

4. Uniform interface – By applying the software engineering principle of generality to the component interface, the overall system architecture is simplified and the visibility of interactions is improved.

5. Layered system – The layered system style allows an architecture to be composed of hierarchical layers by constraining component behavior such that each component cannot "see" beyond the immediate layer with which they are interacting

6. Code on demand (optional) – REST allows client functionality to be extended by downloading and executing code in the form of applets or scripts. This simplifies clients by reducing the number of features required to be pre-implemented

Based on these principles, we have designed the APIs shown in Table 1.

3.4. Map Generator Design

HTTP Verb	URI	Description
GET	/metro/api/v1/users	Get a list of all users
GET	/metro/api/v1/users/{id}	Get the user with {id}
GET	/metro/api/v1/users/{uid}/ maps	Get a list of all maps of the user with {uid}
PATCH	/metro/api/v1/users/{id}/ password	Verify the password of the user with {id}
PUT	/metro/api/v1/users/{id}/ password	Update the password of the user with {id}
PUT	/metro/api/v1/users/{id}/ email	Update the email of the user with {id}
PUT	/metro/api/v1/users/{id}/ name	Update the name of the user with {id}
PUT	/metro/api/v1/users/{id}/ enabled	Update the enabled of the user with {id}
GET	/metro/api/v1/maps/{id}	Get the map with {id}
GET	/metro/api/v1/maps/ ?page={page}&limit={limit}	Get a list of {limit} maps on page {page}
POST	/metro/api/v1/maps	Add a new map to the database
PUT	/metro/api/v1/maps/{id}	Update the map with {id}
DELETE	/metro/api/v1/maps/{id}	Delete the map with {id}

Table 1. REST API Design Table

4. Bibliography

5. Appendices