



İSTİNYE UNIVERSITY

ENGINEERING AND NATURAL SCIENCES FACULTY

COMPUTER ENGINEERING DEPARTMENT

CAPSTONE PROJECT 1

PRELIMINARY REPORT

16 December 2022

PROJECT TITLE

Today's Web Application System Designs and Modern User
Interfaces

PROJECT MEMBERS

Eren Çam

Mehmetcan Emanet

ADVISOR

Dr. Buse Yılmaz

ABSTRACT

The project is going to cover the modern approaches of ultra-large-scale containerized backend systems and front-end designs with the implementations of those approaches on a certain business domain with best practices.

Microservices architecture will be used and several patterns & paradigms (CQRS, API Gateway Pattern, Event Sourcing, Caching First, etc.) are going to be implemented and explained where and why they should be used. The implementation will be done on a certain business domain, and services are going to be shaped around it.

TABLE OF CONTENTS

INTRODUCTION	1
LITERATURE REVIEW	2
MATERIAL AND METHODS.....	3
1. Project Structure	3
2. API Gateway Pattern and Communication of API Gateway Between Services and Client...4	
2.1 gRPC and Protocol Buffers	5
3. Authentication and Authorization with JSON Web Tokens (JWT)	6
3.1 Selecting Password Hashing Algorithm	7
4. Containerization	8
5. Backend System Tools.....	10
6. User Interface Tools.....	10
EXPECTED RESULTS UNTIL THE NEXT REPORT ON THE PROJECT.....	11
PROJECT TIMELINE	12
REFERENCES.....	13
LIST OF FIGURES	15

INTRODUCTION

Since the popularity of the internet has increased tremendously, today's applications have more problems than ever. Traditional monolithic architectures have problems with tight coupling, complexity, limited scalability, management of teams, and difficulty in deploying updates. The goal is showing what are the industry standard solutions using microservices architectures with comparing and investigating trade-offs of different patterns, paradigms, databases, and programming languages on a specific business domain.

In addition, the project is not just going to show you how to make decisions and choose the right technology stack, patterns and paradigms. The Project will also cover best practices for the implementation.

Moreover, user interfaces have evolved as well. In the past desktop applications were needed, however today most of the functions can be done via web browser. User interfaces are bigger, more complicated, and should be faster than ever. So, the project is also going to include best practices for the front-end too.

Briefly, the aim of the project is to help readers comprehend how today's systems are being built, and what are the motivations behind the choices of the solutions for modern architectural problems.

LITERATURE REVIEW

Design of Modern Distributed Systems based on Microservices Architecture [1] article also covers microservices benefits and what are the differences between microservice applications and monolith application but there is no implementation and details about how to build the system. Our project covers best practices and patterns that are applied today as well.

Microservice Architecture Practices and Experience a Focused Look on Docker Configuration Files [2] focuses on containerization part of the microservices. Our project also shows how to containerize the services and the components.

The scope of our project is much wider. We are aiming to show the industry standard microservice implementations with best practices, along with the modern user interface architectures.

MATERIAL AND METHODS

Since the project is going to be large-scale system, the backend is going to be designed with microservice architecture. Microservices scale better than monolith applications. Also it allows to separate the business domain into services. If it planned well, the services are going to be loosely coupled. Another advantage of microservices is that all services are independent programs that allow architects to make decisions based on those needs. We are planning to cover polyglot architecture [3], we are going to discuss which technology or language fits better for a certain service of the architecture and take action according to this discussion.

1. Project Structure

Project repository is mono-repo and all the service implementations can be found in *services*. Also *services* directory has several scripts. *generate-go-proto.sh* generates proto files which are required to communicate with gRPC and *services.sh* starts containers.

The app environment is separated into *development* and *production* for best practice. For example, environment files are separated *dev.env* and *prod.env*. This will help us to distinguish between environments and make some differences according the environment choices. Also, for the future it makes more easier to maintain the project.



Figure 1 Project Structure

2. API Gateway Pattern and Communication of API Gateway Between Services and Client

Since we want to control access to microservices at one endpoint, API Gateway pattern will be used. It allows control authorization, logging, and separation of communication between the client and the microservices. [4]

Client and the gateway is going to communicate via HTTP and the API Gateway and the microservices are going to communicate with gRPC (a framework for remote procedure call [5]). The reason we want to separate the internal and external communication is that gRPC is lightweight corresponding to HTTP, it makes payloads faster and smaller. So, it fits better than HTTP for internal communication in this case.

Moreover, the gateway communicates with the client using JSON over HTTP. JSON is a lightweight data-interchange format. It is humanreadable and has key value structure. [6]

The API Gateway is implemented in Go language. Go is fast and has a strong standard library for network programming.

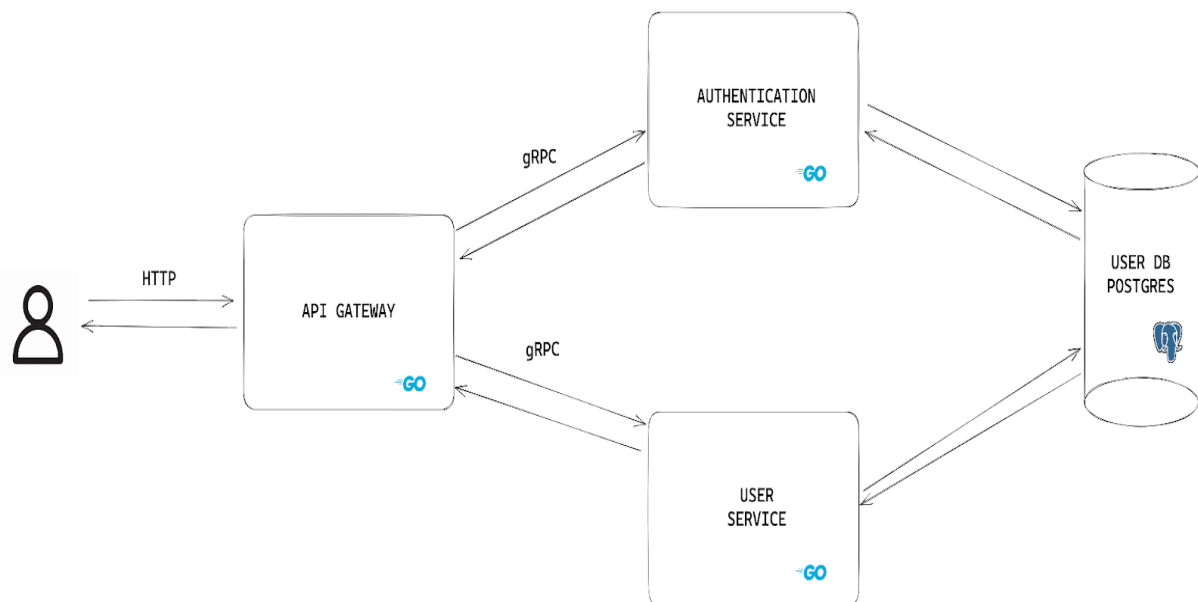


Figure 2 Example Communication of API Gateway Between Services and Client

2.1 gRPC and Protocol Buffers

gRPC supports many programming language [7], it can connect efficiently polyglot services. [8] They can be used with protocol buffers, which are like XML but smaller, faster, and simpler. So, that is why we chose gRPC along with Protocol Buffers for internal communication.

```
syntax = "proto3";

package auth;

option go_package = "pb/";

service AuthService {
  rpc Login(LoginRequest) returns (LoginResponse) {}
  rpc Validate(ValidateRequest) returns (ValidateResponse) {}
  rpc Register(RegisterRequest) returns (RegisterResponse) {}
}

message Response {
  int32 status = 1;
  string error = 2;
}

enum ROLES {
  ADMIN = 0;
  USER = 1;
}

// Login
message LoginRequest {
  string email = 1;
  string password = 2;
}

message LoginResponse {
  Response baseResponse = 1;
  string token = 2;
  string id = 3;
  ROLES role = 4;
}

message ValidateRequest {
  string token = 1;
}

message ValidateResponse {
  Response baseResponse = 1;
  string id = 2;
  ROLES role = 3;
}

message RegisterRequest {
  string email = 1;
  string password = 2;
  string name = 3;
  string surname = 4;
}

message RegisterResponse {
  Response baseResponse = 1;
}
```

Figure 3 Example Protocol Buffer File to Communicate Between API Gateway and Authentication Service

3. Authentication and Authorization with JSON Web Tokens (JWT)

The project has covered role-based authorization, and the authentication has been implemented with JWT (JSON Web Tokens). The claims can be transferred easily with JWT. [6]

An open standard called JSON Web Token (JWT) (RFC 7519) outlines a condensed and independent method for securely transferring data between parties as a JSON object. The fact that this information is digitally signed allows for verification and confidence. JWTs can be signed using a public/private key pair using RSA or ECDSA or a secret (with the HMAC algorithm).

We will concentrate on signed tokens even though JWTs can be encrypted to additionally offer confidentiality between parties. While encrypted tokens conceal those claims from outside parties, signed tokens can be used to confirm the validity of the claims they contain. When signing tokens with public/private key pairs, the signature additionally attests that only the party in possession of the private key signed the token.

Here is an example request to login endpoint from the API Gateway. API Gateway routes this request to Authenticate Service. As a client we make a request to API Gateway endpoint *localhost:3000/auth/login* and gateway route this request via gRPC using Protocol Buffers to *AuthContainerID:50000*. Then, auth service responds back with gRPC again. Finally, API Gateway checks the response and respond back to client with JSON over HTTP.

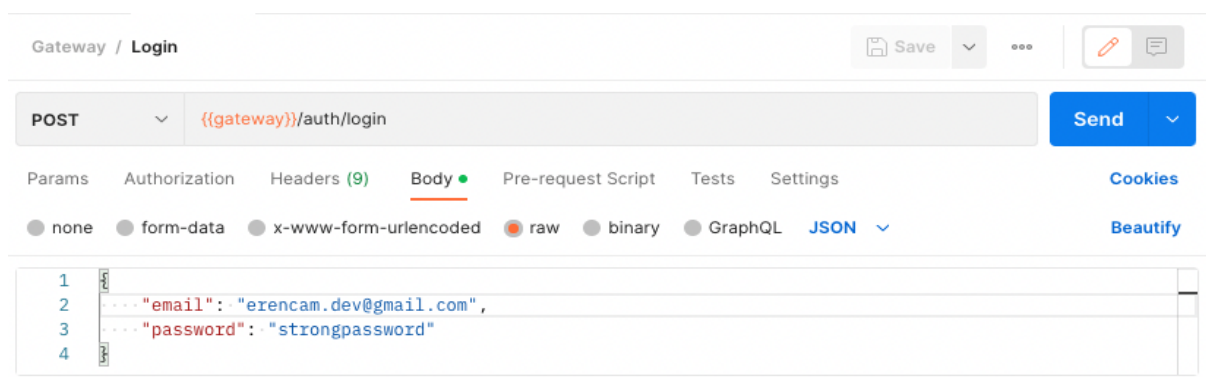


Figure 4 Example Request to Login endpoint with API Gateway

```
Body Cookies Headers (3) Test Results 200 OK 1208 ms 413 B Save Response
Pretty Raw Preview Visualize JSON
1
2   "id": "1f9fc3ab-c3b5-4446-8e7d-23b77796d2e9",
3   "role": 1,
4   "token": "eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJlbWFnbnCI6ImV5ZW5jYW0uZGV2QGdtYWlsLmNvbSIsIlVzZXJJRCI6IjFmOWZjM2FiLWMzYjUtNDQ0Ni04ZTkLTiZyYjc3Nzk2ZDJlOSIsImV4cCI6MTY3MTEwNjE5NSwiUm9sZSI6MX0.8a5hufAcSnxLh-5mx0eQbmyvtoZzxrdVZSI5h6ItBRk"
```

Figure 5 Example Response for Login Endpoint with JWT

When the token is encoded, you can see the structure of it. It has three parts: Header, Payload and the Verify Signature. For now, the payload (claims) is constructed with the user's email, id, role and token's expiration time. The role is used in the API Gateway for the authorization. That way routes can be secured based on roles.

Encoded PASTE A TOKEN HERE

```
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJlbWFnbnCI6ImV5ZW5jYW0uZGV2QGdtYWlsLmNvbSIsIlVzZXJJRCI6IjFmOWZjM2FiLWMzYjUtNDQ0Ni04ZTkLTiZyYjc3Nzk2ZDJlOSIsImV4cCI6MTY3MTEwNjE5NSwiUm9sZSI6MX0.R93i0Br21jHMxM0UKEQ4_ogVgbDXAwEmJRdUfKowc_c
```

Decoded EDIT THE PAYLOAD AND SECRET

HEADER: ALGORITHM & TOKEN TYPE
<pre>{ "alg": "HS256", "typ": "JWT"}</pre>
PAYLOAD: DATA
<pre>{ "email": "erencam.dev@gmail.com", "UserID": "1f9fc3ab-c3b5-4446-8e7d-23b77796d2e9", "exp": 1671106195, "Role": 1}</pre>
VERIFY SIGNATURE
<pre>HMACSHA256(base64UrlEncode(header) + "." + base64UrlEncode(payload), your-256-bit-secret)</pre> <input type="checkbox"/> secret base64 encoded

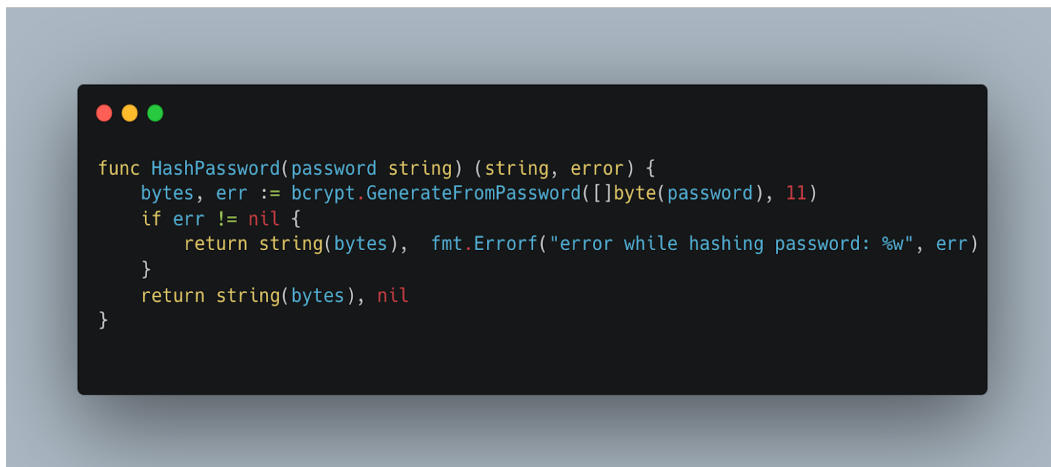
Figure 6 Example Encoded and Decoded JWT Token

3.1 Selecting Password Hashing Algorithm

Increased computation makes this selection more important. To secure our passwords from brute-force attacks from this increased computation, we chose bcrypt algorithm.

MD5 is not secure at all and SHA family algorithms are not good at hashing passwords. Passwords are short and it makes hashed passwords with SHA family algorithms easy to break in the future.

Bcrypt has a *cost factor* and it makes the algorithm slower which means that more secure for brute-force attacks. That way the cost factor can be increased according to increased computation.

A screenshot of a code editor window with a dark background and light-colored text. The code is a Go function named `HashPassword` that takes a `password string` as input and returns a `(string, error)` tuple. The function uses `bcrypt.GenerateFromPassword` to generate a password hash with a cost factor of 11. It includes an error handling block that returns an error message if the generation fails.

```
func HashPassword(password string) (string, error) {
    bytes, err := bcrypt.GenerateFromPassword([]byte(password), 11)
    if err != nil {
        return string(bytes), fmt.Errorf("error while hashing password: %w", err)
    }
    return string(bytes), nil
}
```

Figure 7 Hashing Function with Cost of 11

4. Containerization

Container is isolated process which is the feature of Linux namespaces. [9] Containers are lightweight when comparing with virtual machines. They are measured in megabytes not gigabytes. In terms of what we built, containers are way better in terms of scaling, management and etc. More importantly, they give us freedom to work platform indepent.

All the services are going to be containerized and should be managed effectively in order speed up to development. Each service has its *docker-compose.yml* file in itself and the *services.sh* script makes the defined services in the compose files, up and running.

```

version: "3.8"
services:
  users_db:
    image: postgres:15.1-alpine
    container_name: users_db
    restart: always
    env_file:
      - ./dev.env
    volumes:
      - ./data/db:/var/lib/postgresql/data
      - ./config/db/schema.sql:/docker-entrypoint-initdb.d/schema.sql
    environment:
      - POSTGRES_USER=${POSTGRES_USER}
      - POSTGRES_PASSWORD=${POSTGRES_PASSWORD}
      - POSTGRES_DB=${POSTGRES_DB}

  redis_cache:
    image: redis:7.0.5-alpine
    container_name: redis_cache
    restart: always
    volumes:
      - ./data/cache:/data

  users_service:
    container_name: users_go
    depends_on:
      - redis_cache
      - users_db
    restart: always
    build:
      dockerfile: dev.Dockerfile
    ports:
      - 50001:8081
    volumes:
      - ./:/app

networks:
  default:
    name: cp-backend
    external: true

volumes:
  data:
  config:

```

Figure 8 Example docker-compose File of Users Service

Above, three services are defined users a postgres database, a redis cache, and users service which is written in go. Users service image is built from the Dockerfile. As mentioned on the above, there is *services.sh* that controls those defined services.

```
~/Desktop/backend/services on [?]develop! 17:01:11
$ ./services.sh up
[+] Running 3/0
  :: Container users_db      Running
  :: Container redis_cache   Running
  :: Container users_go      Running
[+] Running 1/0
  :: Container auth_go       Running
[+] Running 1/0
  :: Container gateway       Running

-----
CONTAINERS ARE READY
-----
```

Figure 9 Example of Command `services.sh up`

```
~/Desktop/backend/services on [?]develop! 17:03:07
$ docker ps
```

CONTAINER ID	IMAGE	COMMAND	CREATED	STATUS	PORTS	NAMES
962336eea538	gateway_gateway	"/bin/sh -c 'Compile..."	4 seconds ago	Up 3 seconds	0.0.0.0:3000->8082/tcp	gateway
ce511a777b8c	auth_auth_service	"/bin/sh -c 'Compile..."	4 seconds ago	Up 3 seconds	0.0.0.0:50000->8080/tcp	auth_go
cbdd4d96eb0c	users_users_service	"/bin/sh -c 'Compile..."	5 seconds ago	Up 3 seconds	0.0.0.0:50001->8081/tcp	users_go
582e2403cd2a	postgres:15.1-alpine	"docker-entrypoint.s..."	5 seconds ago	Up 4 seconds	5432/tcp	users_db
bf0768692521	redis:7.0.5-alpine	"docker-entrypoint.s..."	5 seconds ago	Up 4 seconds	6379/tcp	redis_cache

Figure 10 Example of Running Containers

5. Backend System Tools

- Microservices: Go [10], Java [11], Nodejs [12], Typescript [13]
- Databases: PostgreSQL [14], MongoDB [15], Elasticsearch [16], Redis [17]
- Cache: Redis [17]
- Message Brokers: RabbitMQ [18], Kafka [19]
- Containerization: Docker [20]
- External Communication: HTTP [21]
- Internal Communication: gRPC [22]

6. User Interface Tools

On the front-end components are going to implemented according to composition over inheritance principle. We chose React to handle renders. It has powerful structure when it comes to build large applications and also supports several rendering patterns.

- Language and Main Libraries: Typescript [13], React [23], Redux-Toolkit [24] and Redux-Saga [25]

EXPECTED RESULTS UNTIL THE NEXT REPORT ON THE PROJECT

Designing the rest of the services. Currently, all microservices are written in Go language because it fits well with the current design but for the rest of the services, several additional technologies, patterns, languages will be applied. Also, the internal communication is going to be more complicated and master-slave databases may be needed in order to satisfy data consistency between the services.

Project will also cover the choice about the database like where and why non-relational (noSQL) or relational (SQL) databases should be used and we are going to take a deep dive into how to handle distributed data.

Moreover, how to scale the containers according to load. We are planning to use Kubernetes in order to orchestrate containers.

PROJECT TIMELINE

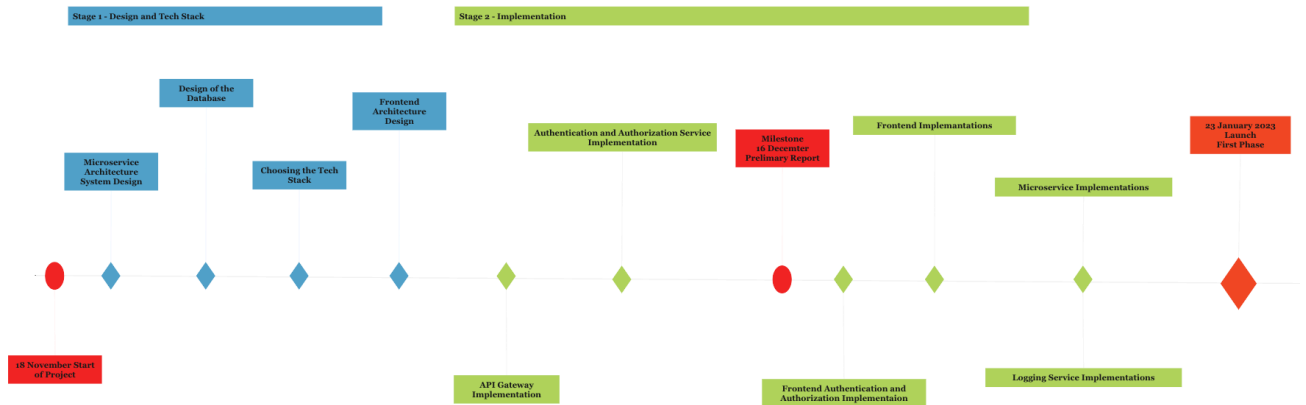


Figure 11 Project Timeline

REFERENCES

- [1] E. M. B. B. T. B. Isak Shabani, "Design of Modern Distributed Systems based on Microservices Architecture," January 2021. [Online]. Available: https://www.researchgate.net/publication/349749418_Design_of_Modern_Distributed_Systems_based_on_Microservices_Architecture.
- [2] G. Q. Luciano Baresi, "Microservice Architecture Practices and Experience: a Focused Look on Docker Configuration Files," December 2022. [Online]. Available: https://www.researchgate.net/publication/366063465_Microservice_Architecture_Practices_and_Experience_a_Focused_Look_on_Docker_Configuration_Files.
- [3] R. R. V, "Microservices with polyglot architecture," in *Spring 5.0 Microservices - Second Edition*, Packt Publishing, 2017.
- [4] C. Richardson, "Pattern: API Gateway / Backends for Frontends," 2018. [Online]. Available: <https://microservices.io/patterns/apigateway.html>.
- [5] M. v. Steen, "Remote Procedure Call," in *Distributed Systems: Principles and Paradigms*, Pearson, 2014, pp. 125-145.
- [6] J. B. N. S. M. Jones, "JSON Web Token (JWT)," May 2015. [Online]. Available: <https://www.rfc-editor.org/rfc/rfc7519>.
- [7] g. Authors, "Supported languages," Google Inc, 11 August 2021. [Online]. Available: <https://grpc.io/docs/languages/>.
- [8] g. Authors, "Who is using gRPC and why," Google INC, [Online]. Available: <https://grpc.io/about/>.
- [9] M. Kerrisk, "Linux Programmer's Manual," 27 August 2021. [Online]. Available: <https://man7.org/linux/man-pages/man7/namespaces.7.html>.
- [10] [Online]. Available: <https://go.dev/>.
- [11] [Online]. Available: https://www.java.com/en/download/help/whatis_java.html.
- [12] [Online]. Available: <https://nodejs.org/en/>.
- [13] [Online]. Available: <https://www.typescriptlang.org/>.
- [14] [Online]. Available: <https://www.postgresql.org/>.
- [15] [Online]. Available: <https://www.mongodb.com/home>.
- [16] [Online]. Available: <https://www.elastic.co/>.
- [17] [Online]. Available: <https://redis.io/>.
- [18] [Online]. Available: <https://www.rabbitmq.com/>.
- [19] "<https://kafka.apache.org/documentation/>," [Online].
- [20] <https://docker.com>. [Online].
- [21] [Online]. Available: <https://developer.mozilla.org/en-US/docs/Web/HTTP>.
- [22] [Online]. Available: <https://grpc.io/>.

- [23] [Online]. Available: <https://reactjs.org/>.
- [24] [Online]. Available: <https://redux-toolkit.js.org/>.
- [25] [Online]. Available: <https://redux-saga.js.org/>.

LIST OF FIGURES

FIGURE 1 PROJECT STRUCTURE	3
FIGURE 2 EXAMPLE COMMUNICATION OF API GATEWAY BETWEEN SERVICES AND CLIENT	4
FIGURE 3 EXAMPLE PROTOCOL BUFFER FILE TO COMMUNICATE BETWEEN API GATEWAY AND AUTHENTICATION SERVICE	5
FIGURE 4 EXAMPLE REQUEST TO LOGIN ENDPOINT WITH API GATEWAY	6
FIGURE 5 EXAMPLE RESPONSE FOR LOGIN ENDPOINT WITH JWT	7
FIGURE 6 EXAMPLE ENCODED AND DECODED JWT TOKEN	7
FIGURE 7 HASHING FUNCTION WITH COST OF 11	8
FIGURE 8 EXAMPLE DOCKER-COMPOSE FILE OF USERS SERVICE	9
FIGURE 9 EXAMPLE OF COMMAND SERVICES.SH UP	10
FIGURE 10 EXAMPLE OF RUNNING CONTAINERS	10
FIGURE 11PROJECT TIMELINE	12

APPENDIX LIST

All the related codes can be found on this GitHub repository:

<https://github.com/capstone-project-bunker/backend>. This repository is going to be private until the project is finished. If you want to access, you can contact us.