**Technical Specification**

**for**

**Prenetics Assessment**

Date: 09 Aug, 2021

Document Version: V0.1

**Document History**

|  |  |  |  |
| --- | --- | --- | --- |
| **Date** | **Version** | **From** | **Remark** |
| 2021-08-09 | V0.1 | Ashwini Pawale | Initial Draft |

**Table of Contents**

[1. Introduction 4](#_Toc79477982)

[1.1 Tech Stack used 4](#_Toc79477983)

[2. Part-A 4](#_Toc79477984)

[2.1 Database Design 4](#_Toc79477985)

[**2.1.1 Data Migration** 4](#_Toc79477986)

[**2.1.2 Data Type** 4](#_Toc79477987)

[**2.1.3 Concurrency** 4](#_Toc79477988)

[**2.1.3 Security** 4](#_Toc79477989)

[2.2 API Aspects 5](#_Toc79477990)

[**2.2.1 Functional aspect** 5](#_Toc79477991)

[**2.2.2 Concurrency** 5](#_Toc79477992)

[**2.2.3 Security** 5](#_Toc79477993)

[**2.2.4 Network Level Security** 6](#_Toc79477994)

[2.3 Handle Variable Traffic 6](#_Toc79477995)

[2.4 Handle secrets, configuration and environments 6](#_Toc79477996)

[2.5 Instructions to bring up the service 6](#_Toc79477997)

[2.6 Testing strategy to the service 6](#_Toc79477998)

[**2.6.1 Back-End testing** 6](#_Toc79477999)

[**2.6.2 Front-End testing** 7](#_Toc79478000)

[3. Part-B 7](#_Toc79478001)

[4. Appendix 8](#_Toc79478002)

# **Introduction**

This platform provides the features for customers to authenticate and retrieve their personal data and genetic results. This feature is implemented using the PERN stack

## Tech Stack used

|  |  |  |
| --- | --- | --- |
| **Item No.** | **Technology** | **Version** |
| 1 | PostgreSQL | 12.6 |
| 2 | Express | 4.17.1 |
| 3 | ReactJS | 16.12.0 |
| 4 | React-Redux | 7.1.3 |
| 5 | NodeJS | 12.18.2 |
| 6 | Typescript | 4.3.5 |

# **Part-A**

## Database Design

### **2.1.1 Data Migration**

* To recreate the PostgreSQL database for this platform, DDL(Data Definition Language), DML (Data Manipulation Language), database user/role creation queries are provided in a file. Please refer to prenetics\_assessment/back\_end/resources/db.sql.
* In the absence of specific requirements for Genetic Results, a simple database design has been followed for this implementation. For practical purposes there can be additional tables with schema designed with Normalization techniques. Please refer to prenetics\_assessment/documents/Data\_Dictionary.docx.

### **2.1.2 Data Type**

For the detailed information about the data types followed, please refer to prenetics\_assessment/documents/Data\_Dictionary.docx

### **2.1.3 Concurrency**

All the database call operations are performed asynchronously using async/await to avoid blocking of NodeJS main thread. In this current design, there is only a single database connection but for future to scale the additional load demand, it is advised to use a connection pool.

### **2.1.3 Security**

* SSL key/certificate was generated and stored on the PostgreSQL server to provide secured connection between client and the server. Please find all the SSL keys and certificates generated and used for this assessment in the folder prenetics\_assessment/keystore. Below are some of the commands used to generate the SSL key/cert on a Linux server.

# Create CA public key and private key.

openssl req -new -nodes -text -out **ca\_root.csr** -keyout **ca\_root\_key.pem** -subj "/CN=**PNTTEST**"

### Use the CA private key and public key to generate CA public certificate.

openssl x509 -req -in **ca\_root.csr** -text -extfile **/etc/pki/tls/openssl.cnf** -extensions v3\_ca -signkey **ca\_root\_key.pem** -out **ca\_root\_cert.pem**

* Apart from the super user **postgres**, two users/roles are created with encrypted password. One role with READ/WRITE rights to perform CRUD operations and the other user with READ-ONLY rights to only read data. More users can be allowed to allow access to certain tables in a schema.
* The PostgreSQL data folder is owned by postgres system user on the server. SSL key/cert folder is also owned by postgres user and the **file permissions** set to **600** to avoid any other system user from misusing these crucial files.
* pg\_hba.conf (PostgreSQL config file) has only specific IP/subnet entries to allow authorized IPs to access specific database.

## API Aspects

### **2.2.1 Functional aspect**

* RESTful API architecture has been followed. This would allow load balancer to be used in future to infinitely replicate new instances of it.
* Customer logs in with Email Id and Password, then the credentials are encrypted and sent over the network to server. On successful authentication, a token is sent back to client. The client stores this token locally and for every subsequent request, this token is added with the header. The server verifies this token before serving any data to the client. Validity of this token is controlled through **expiresIn** parameter in the .env file on server side.
* For better user experience, based on the user activity the value of **expiresIn** parameter for the token expiry can be extended to avoid it from expiring while the customer is using the web application.

### **2.2.2 Concurrency**

Fetch API has been used on the Front-End side to handle asynchronous requests to the server

### **2.2.3 Security**

* JSON Web Token (**JWT**) has been exchanged between client and server to provide customer authentication and authorization to a resource.
* Customer **Password** has been encrypted using **bcryptjs** before saving to database and the same has been used to decrypt the password while authenticating the customer.

### **2.2.4 Network Level Security**

* **SSL** key/cert has been generated on Linux server and stored in local file system to allow **https** connection request and responses for both Front-End & Back-End.
* **CORS** has been handled using **express cors** on Back-End to allow specific IPs/Port to access resources.

**Note:** Please find the SSL key/cert for both Front-End & Back-End API in keystore folder in their respective source code folders.

## Handle Variable Traffic

* RESTful API with Load balancer can be configured to have distributed instances of servers to handle huge data traffic and multiple requests.
* Use a cluster of servers with replication to avoid single point of failure.

## Handle secrets, configuration and environments

JWT secret, database configuration and type of environment (dev/test/prod) is configured using **.env** and **config** file. **dotenv** module has been used to make use of these config variables.

## Instructions to bring up the service

* Front-End:
* Open Command Line
* Navigate to source folder prenetics\_assessment/front\_end
* Run command: **npm install**
* Run command: **npm start**
* Back-End:
* Open Command Line
* Navigate to source folder prenetics\_assessment/back\_end
* Update **.env** with database config parameters, secrets and environment
* Run command: **npm install**
* Run command: **npm start**

## Testing strategy to the service

### **2.6.1 Back-End testing**

* Insert test data in the database. Sample insert statements have been provided. Please refer to prenetics\_assessment/back\_end/resources/test\_data.sql
* Install **mocha, chai**
* Example test cases:
  + Write a test case to test **login** endpoint, check if the endpoint returns status code ‘200’ with token in response body and response error is false on successful authentication of a customer’s email Id and Password.
  + Write a test case to test **authorization** to get genetic results, check if the endpoint returns status code ‘401’ for client request for retrieving the Genetic Results or Personal profile with invalid token else ‘200’ OK for valid token
* Update package.json with the test command to use mocha
* Run the test cases using npm mocha
* **Postman** has been used with SSL enabled to test the end point **https** URLs.

### **2.6.2 Front-End testing**

* Create a mock service to return fake data.
* Install **Jest, Enzyme, react-test-renderer**
* Example test cases:
  + Write a test case to test after successful **login** a home page snapshot is rendered populated with the contents fetched from mock service.
  + Write a test case to test **logout,** when user clicks logout the state should be cleared and login snapshot is rendered.
* Update package.json with the test command to use jest
* Execute the test cases
* Check results in terminal window

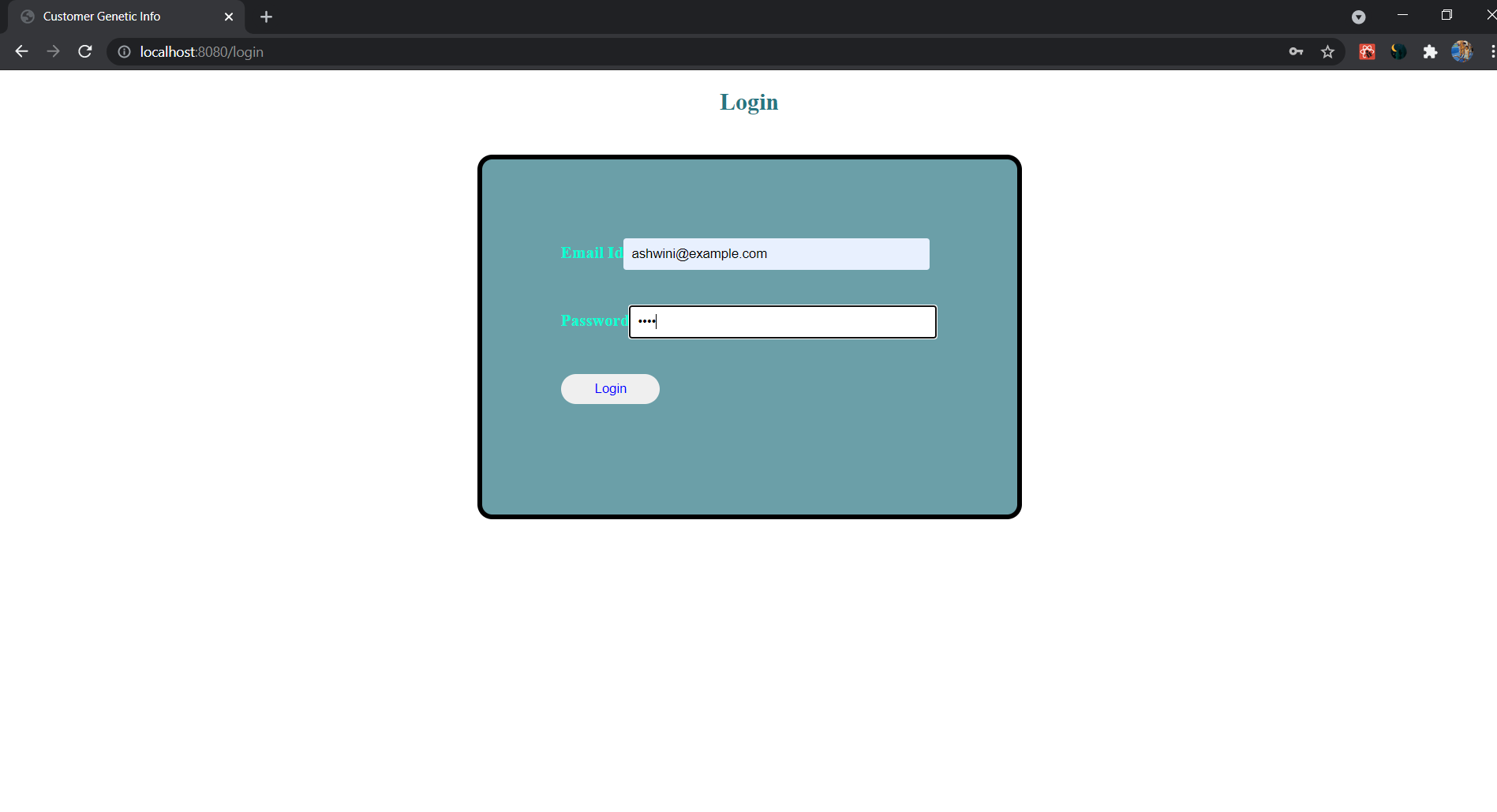
# **Part-B**

We have a centralised authentication service that provides password based authentication for all our users across the platform. Everytime a user logs in, this service is invoked and a security context is passed back to the client. This context is then passed to the services which get back to the authentication service to verify the context. What are the options to scale this service?

A distributed database system can be used to scale the centralized authentication service. The database system should be a cluster of master and standby nodes with streaming replication set up to avoid single point of failure.

# **Appendix**

Login Page



Results Page

