User Microservice

The User Service is used to authenticate and authorize users through a token. This service is responsible for creating users and their corresponding roles and permission using Jwt authentication and authorization. Each of Administrator, Customer and Supplier are users.

To be a User, User Registration is required. Each User is assigned one or more User Roles. Each User Role has a set of Permissions. A Permission defines whether User can invoke a particular action or not.

User stories

- As a user, I want to register, so that I can get access to content which requires registration
- As a user, I want to confirm my email address after registration
- As a user, I want to login and log out
- As a user, I want to change my password
- As a user, I want to change my email address
- As a user, I want to be able to reset my password, so that I can login after I lost my password
- As a user, I want to update my profile, so that I can provide my correct contact data
- As a user, I want to close my account, so that I can close my relationship with that service I signed up for
- As an admin, I want to manage (create/delete/update) users manually, so that staff members wouldn't have to go over the registration process
- As an admin, I want to create users manually, so that staff members wouldn't have to go over the registration process
- As an admin, I want to list all users, even those once who closed their account
- As an admin, I want to be able to see users' activity (login, logout, password reset, confirmation, profile update), so that I can comply with external audit requirements

Non-functional requirements

User stories usually don't define non-functional requirements, such as security, development principles, technology stack, etc. So let's list them here separately.

- The authentication ideally needs to be managed by a third party that can run in the cloud and that could be easily tested out of the talent local machine, such as Auth0 (that

contains a Free tier up to 7000 logins a day). You are free to explore other third party services such as Keycloak
- When users log in, a JWT token is generated for them, which is valid for 24 hours. By providing this token for subsequent requests they can perform operation which require authentication
- Password reset tokens are valid for 10 minutes and email address confirmation tokens for a day

- Users can use gmail and/or facebook account to login/signup into the application
- Passwords should be managed by the third party authentication module in case of a signup not using SSO such as Gmail and/or Facebook
- A RESTful API is provided for interacting with the user registration service
- The application will have a modular design in order to be able to provide separate deployment artifacts for various scenarios
- Entity identifiers are generated in a database agnostic way, that is, no database specific mechanism (AUTO_INCREMENT or sequences) will be used to get next ID values.
- All API calls should be tracked with action being made and who made the action which can be consumed by an admin when using the Analytics endpoint. Analytics should work asynchronous and should use node.js events approach
- The API should be documented using swagger

REST API

Accessing most of the endpoints below require authentication, otherwise they return an UNAUTHORIZED status code.

They also return a client error (FORBIDDEN) if the user tries to query an entity which belongs to some other user, unless he has administrative privileges. If the specified entity doesn't exist the called endpoint returns NOT_FOUND.

Creating a JWT (POST /auth) and registering a new user (POST /users) are public and they don't require authentication.

User management

Minimum user required fields:

```
{
 "id": "auto-generated-id",
 "keycloakId": "auto-generated-from-keycloak",
 "permissionLevel": "Admin",
 "firstName": "Marcos",
 "lastName": "Silva",
 "gender": "male",
 "email": "foo.bar@toptal.com",
 "phone": "+123123123",
 "username": "somethingNice",
 "password": "9uQFF1Lh",
 "birthDate": "2010-12-25",
 "avatar": "img_url_such_as_gravatar",
 "addresses":[ {
   "address": "1745 T Street Southeast",
   "city": "Washington",
   "postalCode": "20020",
   "state": "DC",
   "primary":"true|false",
   "label":"home|..."
 }],
 "status": "active|closed",
 "createdAt": "timestamp",
 "modifiedAt": "timestamp",
}
```

GET /users/{user_id}

Finds a user with the given ID.

GET /users

List all users in the system. Should also include query params such as limit and offset and return the list of the users based on limit and offset. Max limit to be 1000. Only admin can query this endpoint

POST /users

Registers a new user. This route should be public. Email and username should be unique

DELETE /users/{user_id}

Deletes the given user. Only same user or admin can call this action

PUT /users/{user_id}

Updates the profile of a given user. Updates the entire user resource. Only the same user or admin can call this action. Only the admin can change user permission level. In case of password changes, it needs to hash it properly.

PATCH /users/{user_id}

Partially updates user fields. Only the same user or admin can use this action. In case of password changes, it needs to hash it properly

Admin Analytics

Minimum user required fields:


```
{
 "id": "auto-generated-id",
 "action": "user.create",
 "requestUserId": "user_id_who_requested_action",
 "payload": "relevant json data based on action",
 "createdAt": "timestamp"
}
```

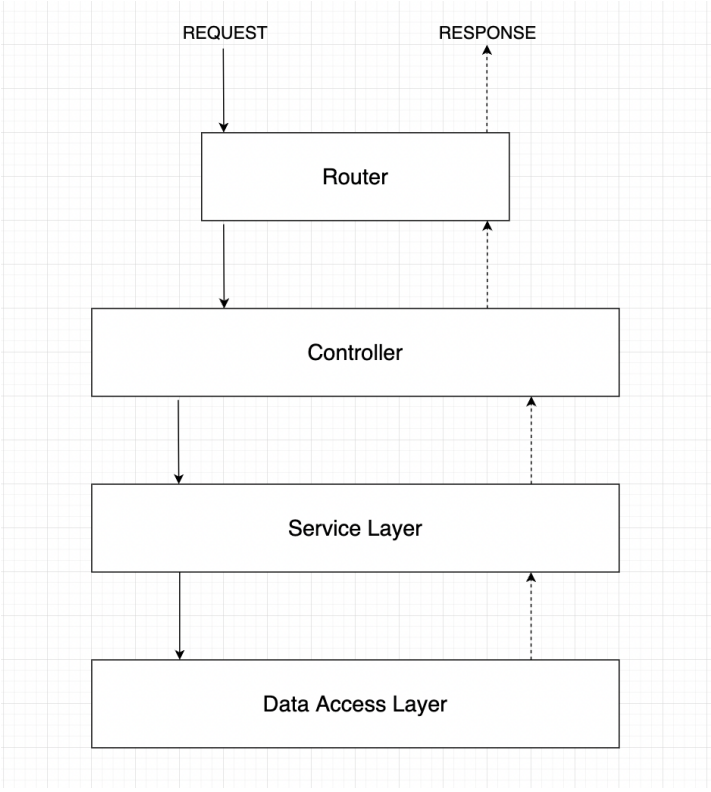

GET /analytics

Finds a list of all users activity. Can receive parameters to filter the list such as limit, offset, userId and action (login, logout, password reset, confirmation, profile update)

Authentication

An application asks the third party Auth server to authenticate a user for them. After a successful login, the application will receive an identity token and an access token. The identity token contains information about the user such as username, email, and other profile information. The access token is digitally signed by the realm and contains access information (like user role mappings) that the application can use to determine what resources the user is allowed to access on the application.

Architecture (3 Layer Architecture):



The Controller layer will be handling all stuff that is related to HTTP. Above that layer is also a Router from Express that passes requests to the corresponding controller.

The business logic will reside in the Service Layer that exports certain services (methods) which are used by the controller.

Data Access Layer will have all the needful logic to connect with the DB.

Things you can consider while developing solution for this requirement

1. Versioning

Consider organizing the code in such a way to address the need of versioning. There can be future improvements, new features, and stuff like that. So it's important to consider appropriate arrangements for versioning while implementing the solution.

We also don't force the clients to use the new version straight away. They can use the current version and migrate on their own when the new version is stable.

The current and new versions can be running in parallel and don't affect each other.

2. Naming Resources

Consider naming the resources appropriately as your API is going to be used by humans and should be precise

3. Accept and respond with data in JSON format

When interacting with an API, you always send specific data with your request or you receive data with the response. There are no obligation to use camelCase or underscore_case for your JSON object, but once you defined it then be consistent in the entire API

4. Respond with standard HTTP Error Codes

When something goes wrong (either from the request or inside our API) send appropriate HTTP Error codes / data with appropriate messages. Also think about filtering error codes in production to not expose too much relevant information that could put your API at risk

5. Well compiled documentation

Consider using swagger or any alternative to implement appropriate documentation for the REST API to include relevant information such as the endpoint and compatible methods, different parameters, type of data and so on

6. E2E tests for endpoints

Write appropriate e2e tests for all the endpoints you are exposing in your API. Having this set in your API will facilitate automated tests and will avoid having issues going to Production. Make sure to test the good and bad scenarios

7. Unit tests

Define an ideal unit test scenario for your API as well as the minimum acceptable code coverage for it. In some cases you don't need to unit test libraries that you are using that were already tested, as well as in some scenarios you don't need to focus too much in some other cases that could be tested in the e2e (routes as an example)

Also, if you can do unit tests while coding (like using test driven development) then you will tend to have a naturally testable application being developed.

8. Validate input data

Consider having appropriate validation for user-send data and ensure consistent and secure database updates.