Organ_Management_Backend

Backend for the Organ Management System built as a part of the Software Engineering course

Made by -

1. IMT2021019 - Siddharth Kothari
2. IMT2021028 - Sankalp Kothari
3. IMT2021002 - Kolipakula Charan Sri Sai
4. IMT2021034 - Prachoday Davuluri

Instructions to run

1. You need to have MySQL, Java installed in your systems. For installation of those kindly refer to

respective documentation.

   MySQL-Shell

   MySQL-WorkBench

   Java

2. Download the softwares suitable for your operating system.
3. Next clone this repository.
4. Then run the follwing command in your terminal source organ_donation.sql. This shall create the necessary database in you local computer. This can also be done by opening the script in workbench, and running it.
5. Now go to the OrganManagementSystem folder and then to src/main/resources. cd src/main/resources/ it's equivalent in your OS.
6. Go to application.properties file and then *comment in the first three lines*.
7. Make sure to add your username and password in those fields.
8. Next run the spring boot application using any IDE (we recommend IntelliJ), or type the following command in your terminal mvn spring-boot:run
9. Now, you can use any service like Postman for testing out the variuos functionalities given in the controller classes.

Application Details

Entities and Classes

1. User - denotes all types of users having different roles and stores login details for different types of users (Doctor, Admin, User) - (Users are essentially patients)

2. PatientInformation - stores details like name, age, gender blood group etc of the patients. Patient is simply a broad term being used for anyone registered in the system who is not an admin user / doctor.
3. DoctorInformation - stores info and contact details of the doctor.
4. Donor - Any patient wanting to donate organs.
5. Recipient - any patient in need of an organ.
6. DonorRecipientMatch - stores pair of donor and recipient that have been matched (matching happens by checking the organ and blood group for compatibility).

The application architecture is as follows: -

REST Contollers -

They contain the necessary endpoints for the respective entities and they are responsible for interacting with the web and the application. Respective functions from services are called and values are for the same are returned.

**Services*- They act as bridge between the *REST Controllers and the Repositories. They shall call the respective functions from the Repositories/DAOs and the values are returned to the REST Controller.

Repositories/DAOs -

They are responsible for communicating with the database. All the operations to the database are headed from here.

Endpoints -

- */register_admin /register_doctor* - Only admin has access to these endpoints. They allow the admin to register a doctor / a new admin user and store their details in the database.
- */register_user* - All kind of users have access to this endpoint. Everyone can register themselves as a patient on the application.
- */authenticate* - Any person who has been registered with the database can use this endpoint to login with valid credentials. If the user is logged in successfully, they recieve a jwt token which is stored locally and auto logs in the user until it expires, after which user must log in again.
- */admin/doctors/*\ /admin/patients/** - Only admin has access to these endpoint, these are for viewing all / specific doctors and patients registered in the system.
- */doctor/viewPatients/*** - Only doctor can use this endpoint, for viewing all / specific patients registered in the system.
- */doctor/viewMyInfo* - For doctors to view their details like name, contact info etc.
- */doctor/addMyInfo /doctor/updateMyInfo* - For doctors to add / edit the aforementioned details about them.
- */user/viewMyInfo /user/updateMyInfo / user/addPatientInfo* - these endpoints serve similar purpose to the ones that the doctor had, with extra information like blood group etc. Now, we have only implemeted matching on blood group and organ requested, but the actual parameters for matching are way more complex in real life, hence in future iterations, we can add more parameters which the doctor will have access to, not the user, which would help in more accurate and realistic matching.

- */recipent/viewInfo /donor/viewInfo* - donors and recipient patients can view the matching status here.
- *recipient/addInfo/*\ /recipient/updateInfo/\ donor/addInfo/** - doctors add the organ to be donated/ that has been requested with the priority here and only they have access to add / update any of this information. Also, when adding a donor or recipient, we also check if there exists someone that can be a potential match for them (with someone of preferably les spriority but not already matched), so that waiting time is reduced.
- */recipient/getAll /donor/getAll* - for admin and doctors to see all the requests made, and the people who have registered for donating.
- */match/donor/*\ /match/recipient/** - Each request made / organ being donated has a particular id, this endpoint finds the match (if it exists) for that organ.
- */match/patient/donor/*\ /match/patient/recipient/** - for checking all organs donated / requested by someone registered in the system.

Testing -

- *Security Testing* -

* We made sure that any user can use the app only if they are signed in, and the jwt token helps to determine what all features they are allowed to access.

- We used postman to make sure each of the features works.

The report for the same can be found here.

- We used JUnit and Mockito to test the following :-

unit tests for -

- *Controller Testing* - we checked that all endpoints are accessible and serve the purpose they were designed for.

We designed specific test cases to show what each function does, and the expected outputs, in both cases where we either have a set of arguments which give successful outputs and ones where there is some Exception thrown because of incorrect input or the user being unaithorized etc. Here is a sample piece of code showing how the doctor gets a paticular donor record -

```
@Test
@WithMockUser(username = "patient1",
    password = "test123", roles = "USER")
public void testGetByDonorId() throws Exception {
    when(matchService.getMatchByDonorId(any()))
            .thenReturn(donorRecipientMatch);
```

```
    mockMvc.perform(MockMvcRequestBuilders.get("/match/donor/1")
            .contentType(MediaType.APPLICATION_JSON)
            .header("Authorization", "Bearer
            eyJhbGciOiJIUzI1NiJ9.eyJzdWIiOiJwYXRpZW50MSI
            sImlhdCI6MTcwMTM2NzI0NiwiZXhwIjoxNzAyNTc
            2ODQ2fQ.i5dJkbNKzifod6q9HzoGUV35ngx
            IprgCYIxf_vIvI4I"))
        .andExpect(MockMvcResultMatchers.status().isOk())
        .andExpect(MockMvcResultMatchers
        .jsonPath("$.donor.organName").value("Kidney"))
        .andExpect(MockMvcResultMatchers
        .jsonPath("$.recipient.organName").value("Kidney"));
  }
```

- *Service Testing* - we checked that all the service class methods were calling the underlying DAO operations correctly using carefully written unit tests to check all possible scenarios.

Here also, we made sure to include all the various possible scenarios (successful execution and exceptions being thrown). An example of adding a donor record is shown below -

```
@Test
public void givenDonorToAddShouldReturnAddedDonorInfo(){
    when(donorDAO.save(any())).thenReturn(donor);
    Donor donor1 = this.donorService.addInfo(this.donor);
    assertThat(donor1).isEqualTo(donor);
    verify(donorDAO, times(1)).save(any());
}
```

The following code snippet shows how we throw an exception if there does exist a patient record for the id entered -

```
@Test
public void GivenPatientNullWillThrowException(){
```

```
    when(patientInfoDAO.findById(1)).
        thenReturn(Optional.empty());
    assertThrows(PatientNotFoundException.class,
        () → patientService.viewPatientInfo(1));
    verify(patientInfoDAO, times(1)).findById(1);
}
```

We have similarly done for all controllers and services and controllers.