Criterion C - Development

Pharmacy Management Application

Technical Evaluation of Success Criteria....2
Frontend Connection....6
Interceptors.... 7
Usage of the Axios Instance in the Frontend....10
Backend Connection....11
Serializer.... 11
API Endpoints and Views.... 12
Authentication Classes....13
Folder Structure....15
Front end....17
Home Page for application....17
Home Page for profiles.... 19
Home Page for the other profiles.... 22
Menu Bar and Drawer....23
View Employees.... 27
Dialog Boxes and Notifications.... 29
Add New Medicines....31
Filled function....35
Add Stock for Existing Medicines....35
Add Prescription.... 36
Logout....39
References....40
Additional Logic in the Backend.... 41
Update Stock....41
Calculate the Cost of Unpaid Appointments.... 43
Logout....44
References....44

Technical Evaluation of Success Criteria

Success Criteria	**Pages that achieve the success criteria**	**Technical Components that have been used to achieve the success criteria**
Profiles for the Front Desk, Pharmacists, Doctors, and the Admin	Home page of the application in the frontend.	Carousel for viewing all the profiles in an orderly manner. Login and signup buttons that direct users to their respective profile pages.
Only the admin profile can verify and remove employees and edit doctor specializations.	Home page of the administrator profile in the frontend. View Employees and Verify Employees for viewing and verifying employees in the frontend. View Specialization for viewing, adding, and editing specializations in the frontend.	PrimeVue's Data Tables show a preview of each action on the administrator profile's home page. Each data table is accompanied by buttons to redirect the user to a page where the action can be performed, such as verifying employees. On the dedicated page, special data tables in which instances can be selected and edited are used. Buttons have been appropriately placed to perform the action. The action of verifying and removing employees, and editing doctor specializations has been only restricted to administrator credentials. From the backend, these views can only be accessed if the authorisation credentials in the request match the permission granted in the backend through the use of permission_classes and authentication_classes features provided by Django Rest Framework.
Admin and pharmacy profiles can view and edit medicines and inventory, and lab tests.	Home Page of the administrator and pharmacy profile in the frontend. View Medicines, Add New Medicines, and Add Existing Medicines pages for viewing and editing the details of medicines in the frontend. View Lab Tests and Add Lab Tests pages for viewing and adding lab tests in the frontend.	
The front desk profile can book appointments for new and existing patients.	Home Page of Front Desk Profile in the frontend. Add New Patient page in the Front Desk profile in the frontend.	Button in the Home Page of Front Desk Profile for redirecting the user to appropriate pages. For adding appointments for new patients, various PrimeVue input fields

	Add Existing page in the Front Desk profile in the frontend.	are placed in the centre of the page to input the details of the patient and the appointment. For existing appointments, the details of the patient need not be entered again. The patient can be selected from a data table. Only appointment details, like the appointment date, and doctor assigned need to be entered.
The doctor profile can write a prescription for booked appointments.	Home page of the doctor profile in the frontend. Add Prescription page in the doctor profile in the frontend.	Data Table is used in the Home Page of the doctor profile to showcase all the appointments that need to be evaluated by the doctor. A Button is placed beside each prescription to write a prescription that will be linked to that appointment. The button will redirect the user to the Add Prescription page, and it uses query parameters to identify each unique patient and their appointment. On the Prescription page, the doctor can prescribe medicines, lab tests and additional information if required.
The doctor's profile allows access to the prescription at all times, and the pharmacy profile allows access for some time, abiding by medical rules and regulations.	Home Page of the doctor profile in the frontend. View Prescription page in the doctor's profile in the frontend. Add Prescription page in the doctor's profile in the frontend. Home Page of the pharmacy profile in the frontend. Dispense Medicines Page of the Pharmacy profile in the frontend.	Sortable Data Tables have been placed in the home page of the doctor profile, one for showing previous prescriptions that are completed and paid, and one for prescriptions that are incomplete. Each of the rows in the data table has an accompanying button to view the previous prescription. Clicking the button redirects the user to the View Prescription page which shows the details of the previous prescription. The Pharmacy profile can only view profiles that are marked as incomplete. Each Prescription instance has two fields, one for keeping track of the status of dispensed medicines, and one for

		collecting data for lab tests. If the patient pays through the front desk profile, both of these fields are marked as complete and cannot be accessed again by the pharmacy profile.
The pharmacy profile can dispense medicines and enter the required data for lab tests.	Home Page of the pharmacy profile in the frontend. Dispense Medicines page of the pharmacy profile in the frontend. Update Lab Test page of the pharmacy profile in the frontend.	Sortable Data Tables that show outstanding medicines that need to be dispensed and the data for lab tests that need to be collected. Dispense Medicines page has input fields for the number of units that are dispensed for each medicine to accurately keep track of the stock. Similarly, the Update Lab Tests page employs similar Input fields to collect data for each lab test that is assigned.
Status trackers for prescribed medicines, lab tests and prescriptions.	Implemented in models.py file in the doctor profile in the backend. Dispense Medicines page of the pharmacy profile in the frontend. Update Lab Test page of the pharmacy profile in the frontend.	The status of all prescribed medicines is tracked using a single BooleanField called 'medicines_fulfilled' in the Prescription model. Each lab test has its own status, stored as a CharField in the PrescribedLabTest model. A BooleanField called 'lab_tests_completed' in the Prescription model tracks whether all lab tests are completed. The pharmacy profile can update these statuses based on progress.
All required fields must be filled out before submitting; to inform the user otherwise appropriately.	All pages in the front-end profile that require the submission of data using Input fields. helpers.js in the frontend profile.	helpers.js defines a method called 'filled' that checks if the data to be sent to the backend does not have empty values in any of the required fields.
Sessions will automatically expire after a set period, requiring re-login.	settings.py file in the root folder of the backend.	When configuring Simple JWT, the library responsible for generating and authorizing access and refresh tokens, the

		access token's validity is set to 30 minutes, while the refresh token lasts for one day. After one day, the user must log in again to obtain a new access token.
Structured way of generating tokens at the front desk, for an ordered way of writing prescriptions by the doctor.	views.py in the front desk profile of the backend. The home page of the doctor profile is on the front end.	The last appointment token is accessed from the backend using database querying and the next token is returned. If none is found, it indicates that there are no patients assigned for that day, and assigns 1 as the token. These tokens are highlighted in the doctor home page, so the doctor can decide an ordered way of evaluating his/her patients.
The system will calculate the total costs for an appointment based on consultations, lab tests, and medicines taken.	views.py in the front desk profile of the backend. Home Page of the front desk profile in the frontend.	The method getTotalCost defined in views.py accurately calculates the money due by the patient. The amount calculated by the above algorithm is reflected against each patient in the home page of the front desk profile, showing how much each patient needs to pay.
The system will display clear error messages and success notifications for user actions to keep them informed.	All pages of the front end profile.	Toasts have been used along with the warn function to serve as notifications pop-ups, informing the user of the status of an action, such as creating an appointment for a new patient.

Frontend Connection

Below are the helper functions for the interceptors (described below) for making the frontend connection.

```
import axios from 'axios';
import router from './router';

const instance = axios.create({
    baseURL: 'http://127.0.0.1:8000'
});

const getAccessToken = () => {
    return localStorage.getItem('accessToken');
};

const getRefreshToken = () => {
    return localStorage.getItem('refreshToken');
}

const refreshAccessToken = async () => {
    const refreshToken = getRefreshToken();

    if (!refreshToken) {
        return null;
    }

    try {
        const response = await instance.post('/api/v1/jwt/refresh', { refresh: refreshToken });
        const accessToken = response.data.access;
        localStorage.setItem('accessToken', accessToken);
        return accessToken;
    } catch (error) {
        return null; // The refresh token is invalid or expired.
    }
}
```

An instance whose properties will be modified to fit the needs of the application

Gets access token from local storage

Gets refresh token from local storage

Refresh token was not successfully accessed and thus it returns null

Makes a request to the backend to this URL that will return a new access token

Updates the access token when it expires, if the refresh token is valid. Otherwise, returns null.

Figure 1: *Helper Functions for Axios instance*

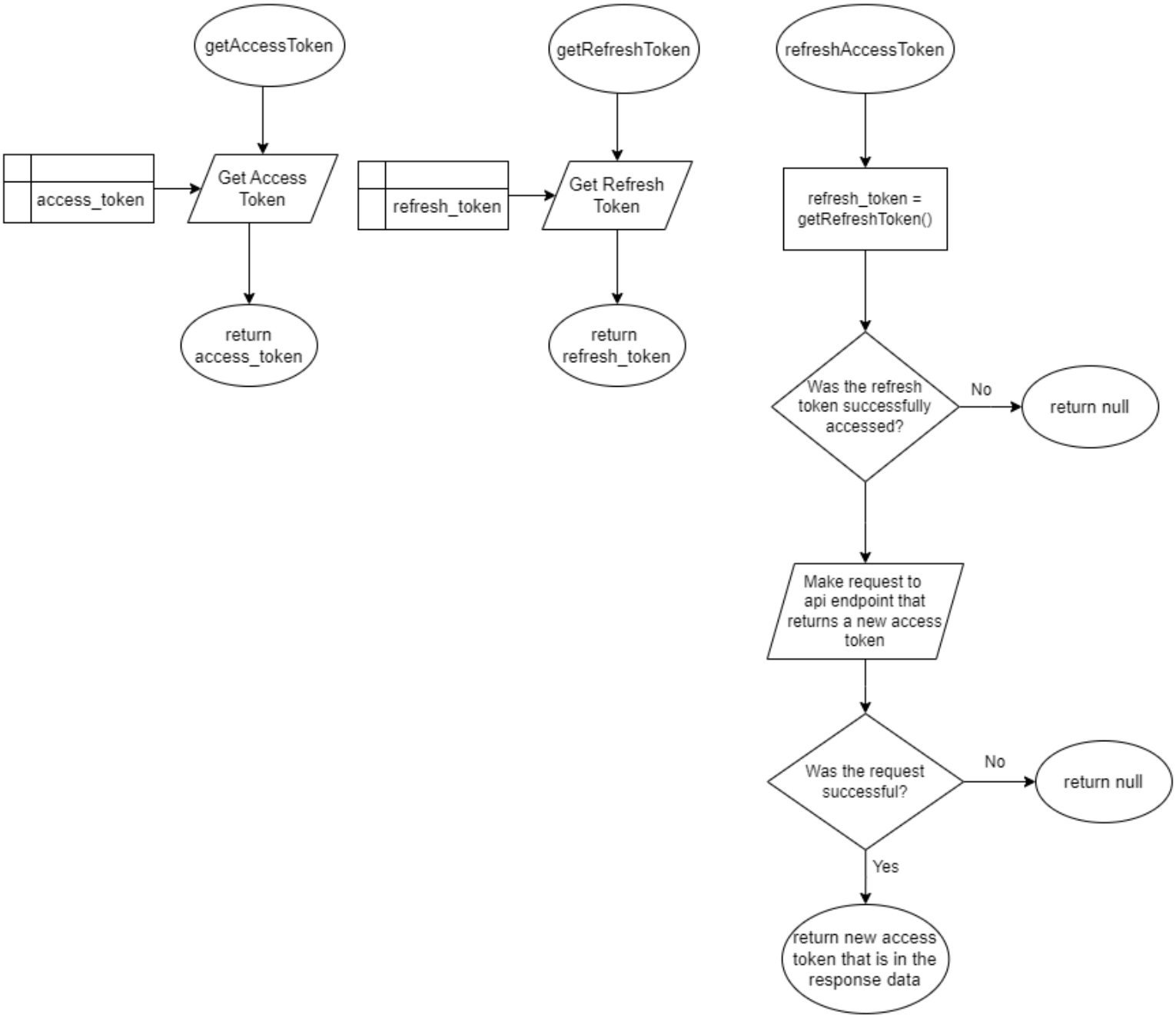

Figure 2: *Flowchart for logic of helper functions of axios instance (Created using draw.io)*

Interceptors

Interceptors manipulate requests and responses before they reach the main functions, which is particularly useful for handling JWT tokens.

```
instance.interceptors.request.use(
    async(config) => {
        let access_token = getAccessToken();

        if (access_token) {
            config.headers['Authorization'] = `JWT ${access_token}`
        }

        return config
    },
    (error) => {
        console.log(`Request Error: ${error}`)
        return Promise.reject(error)
    }
)
```

An Axios request interceptor modifies requests before they are sent, enabling authentication headers, logging, or request transformation to ensure consistency and security.

Attaches the access token to the HTTP request headers to authenticate the client for secure communication.

If an error is encountered when making the request, the request is cancelled, and the error is propagated down the promise chain. It has to be handled by the calling block.

***Figure 3**: Request Interceptor for axios instance*

```
instance.interceptors.response.use(
    (response) => {
        return response
    },
    async (error) => {
        console.log(error.response)
        const original_request = error.config

        if (error.response.status === 403 && error.response.data.detail === 'Authentication credentials were not provided.') {
            const new_access_token = await refreshAccessToken();
            if (new_access_token) {
                original_request.headers['Authorization'] = `JWT ${new_access_token}`;
                return instance(original_request);
            }
        }
        else if (error.response.status === 401 && error.response.data.detail === 'Token is invalid or expired') {
            const usertype = localStorage.getItem('usertype');
            localStorage.setItem('usertype', '');
            localStorage.setItem('is_registered', false);
            localStorage.setItem('refresh_token', '');
            localStorage.setItem('access_token', '');
            localStorage.setItem('first_name', '');
            localStorage.setItem('last_name', '');
            localStorage.setItem('user_id', '');
            if (!usertype && !window.location.href.includes('login')) {
                router.push('/');
            }
            else if (usertype  && !window.location.href.includes('login')) {
                router.push(`/${usertype}/login/`);
            }
        }
        return Promise.reject(error);
    }
)

export default instance;
```

If the error gotten is a 403 error and the error message is "Authentication credentials were not provided.", it is due to an expired access token.

Gets a new access token

Makes the original request again

Updates the access token that is sent on each request to the backend

If the error is a 403 error with the message "Token is invalid or expired", it means that the refresh token is expired

Deleting all information from the client side local storage

If somehow, the usertype could not be accessed from the local storage, the user is redirected to the home page of the application

If the usertype could be accessed, the user is redirected to their respective login pages.

***Figure 4**: Response Interceptors for axios instance*

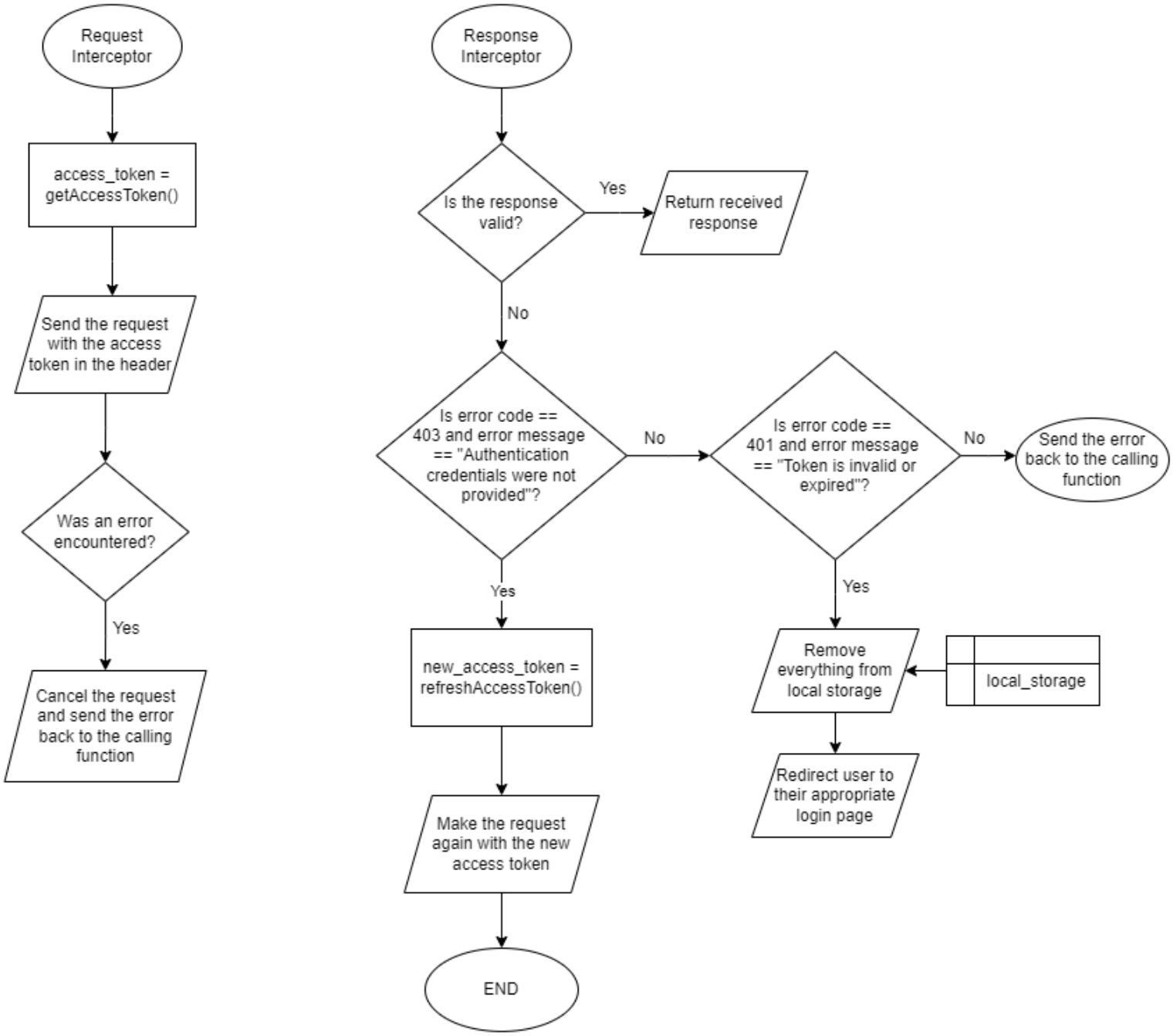

Figure 5: *Flowchart of Interceptor Logic for Axios instance*

Usage of the Axios Instance in the Frontend

An HTTP **GET** request retrieves data from a server without modifying any resources, commonly used for fetching web pages or API responses. An HTTP **POST** request sends data to a server to create or update resources, often used for submitting forms or uploading files.

```
axios.get('administrator/specializations/')
.then( (response) => {
    specializationsAvailable.value = response.data;
})
.catch( (error) => {
    warn("Error getting specializations.")
})
```

Makes a request to the backend to get information

Saves the response in a variable

If there's an error in getting the response from the backend, the user is informed appropriately.

Figure 6: *Get request made in the Doctor Sign In page to get the specializations available*

```
axios.post("/administrator/verifyEmployees/", { code: code, ids: idArray })
.then( (response) => {
    data.value = data.value.filter(val => !selected.value.includes(val));
    selected.value = null;

    if (code == 0){
        toast.add({severity:'success', summary: 'Successfully verified users!', life: 3000});
    } else {
        toast.add({severity:'success', summary: 'Successfully deleted users!', life: 3000});
    }
})
.catch( (error) => {
    console.log(error)
    if (code == 0){
        toast.add({severity:'warn', summary: 'Unsuccesful in verifying users.', message: 'Please try again.', life:3000});
    } else {
        toast.add({severity:'warn', summary: 'Unsuccesful in deleting users.', message: 'Please try again.', life:3000});
    }
})
```

Data to be posted

URL (API Endpoint) to be posted to

Logic for what happens if the post request was successful

Logic if an error was encountered during the request

Figure 7: *Post request made in Verify Employees page in Administrator profile to update the database on which employees to verify based on their ids*

Backend Connection

Serializer

Serializers are used to convert complex data types in Python to JSON data and vice versa which can be sent over HTTP requests. They are used to make new instances of a model, update an existing instance, and more. Different models might require different logic based on their fields and validation requirements.

```python
class DoctorSerializer(serializers.ModelSerializer):
    user = UserSerializer(many=False)

    class Meta:
        model = DoctorUser
        fields = '__all__'

    def create(self, validated_data):
        user_data = validated_data['user']
        specialization = validated_data['specialization']
        user = User.objects.create_user(**user_data)
        if not user:
            raise serializers.ValidationError("Failed to create user.")
        specialization = SpecializationAvailable.objects.get(id=specialization.id)
        validated_data['user'] = user
        validated_data['specialization'] = specialization
        try:
            doctor = DoctorUser.objects.create(**validated_data)
            return doctor
        except Exception:
            user.delete()
            return serializers.ValidationError("Failed to create doctor user.")
    def update(self, instance, validated_data):
        instance.user = validated_data.get('user', instance.user) # Make a user object

        if validated_data['user']:
            UserSerializer(instance.user, data=validated_data['user'])
        instance.consultation_fee = validated_data.get('consultation_fee', instance.consultation_fee)
        instance.experience = validated_data.get('experience', instance.experience)
        instance.registration_number = validated_data.get('registration_number', instance.registration_number)
        instance.specialization = validated_data.get('specialization', instance.specialization)
        instance.save()
        return instance
```

One To One Relationship between a DoctorUser and a User. The details for the user are nested in the request data.

Code that is executed when a new object of the model is created.

Gets the appropriate model references (creates them if required), and then replaces the JSON data with the reference to the model

Tries creating a user

If the doctor user creation is unsuccessful, the user created initially is deleted so that the new corrected data can be used to create one more user, which would otherwise have created a username clash. An error is also raised.

First argument is used if it is present in the request data

Falls back to the second argument if the first argument is not present

If en existing doctor user profile needs to be updated, this method is executed.

Saves the updated instance and returns it back to the calling code

Figure 8: *Serializer for DoctorUser Django model*

API Endpoints and Views

API Endpoints are the URLs that defines a point of interaction between the client and the server. Views are tied to these Endpoints and are used to define logic to handle the requests. All these URLs are defined in the urls.py file for each profile, used for accessing different endpoints.

```
class GetSpecializationAvailable(views.APIView):
    '''
        In a get request, returns all specializations available
        In a post request, filters the specializations returned according to the name given
    '''

    authentication_classes = (authentication.CustomAdminAuthentication, )
    permission_classes = (permissions.IsAuthenticatedOrReadOnly, )

    def get(self, request):

        specializations = SpecializationAvailable.objects.all()

        resp = []

        for specialization in specializations:
            number = DoctorUser.objects.filter(specialization=specialization).count()
            resp.append({'id': specialization.id, 'specialization': specialization.specialization, 'number': number})

        return response.Response(resp)

    def post(self, request):

        specializations = SpecializationAvailable.objects.filter(specialization__icontains=request.data['name'])

        resp = []

        for specialization in specializations:
            number = DoctorUser.objects.filter(specialization=specialization).count()
            resp.append({'id': specialization.id, 'specialization': specialization.specialization, 'number': number})

        return response.Response(resp)
```

Only Administrator credentials can be used to access this endpoint

When a request of type 'GET' is made to this url, this function is executed.

Gets the count of number of doctors that are associated with this specialization

Goes through each instance from all the instances returned

For each specialization instance, returns all its required fields that will be useful and displayed in front end

Attaches the access token to the HTTP request headers to authenticate the client for secure communication.

When a request of type 'POST' is made to this url, this function is executed.

Same logic as in the 'GET' request, but with filtered data

Figure 9: *API View for getting SpecializationAvailable objects*

```python
class AddSpecializationAvailable(views.APIView):

    authentication_classes = (authentication.CustomAdminAuthentication, )
    permission_classes = (permissions.IsAuthenticatedOrReadOnly, )

    def post(self, request):

        serializer = SpecializationSerializer(data=request.data)

        if serializer.is_valid():
            serializer.save()

            return response.Response(serializer.data, status=status.HTTP_201_CREATED)
        else:
            return response.Response(serializer.errors, status=status.HTTP_400_BAD_REQUEST)
```

The request data is sent directly to the appropriate serializer

If the request data was appropriate and the serializer could successfully create an instance of the model, it saves that instance in the database

Returns a message whether or not the creation of the instance was successful

```python
class EditSpecializationAvailable(views.APIView):
    authentication_classes = (authentication.CustomAdminAuthentication, )
    permission_classes = (permissions.IsAuthenticated, )

    def post(self, request):

        specializationAvailable = SpecializationAvailable.objects.get(id=request.data['id'])
        serializer = SpecializationSerializer(specializationAvailable, data=request.data)
        if serializer.is_valid():
            serializer.save()
        else:
            return response.Response(serializer.errors, status=status.HTTP_400_BAD_REQUEST)

        return response.Response(serializer.data, status=status.HTTP_202_ACCEPTED)
```

Gets the instance whose value has to be updated

The serializer updates the instance automatically based on the data that was sent from the front end

Saves the edited instance if the serializer could successfully edit it

***Figure 9**: API View for adding and editing SpecializationAvailable instances*

Authentication Classes

Authentication classes are a set of rules and methods used to verify the identity of a user from the access token provided in the request made. If multiple authentication classes are specified in the API Endpoint, all of them are checked against the token.

```python
class CustomAdminAuthentication(authentication.BaseAuthentication):

    def authenticate(self, request):

        JWT_authenticator = JWTAuthentication()

        try:
            response = JWT_authenticator.authenticate(request)
        except exceptions.AuthenticationFailed:
            return None

        if response is not None:
            user, token = response
        else:
            return None

        if user.role != 'Administrator':
            raise exceptions.PermissionDenied("Unauthorized access.")

        return (user, None)
```

Instance that decodes the access token that was provided

The access token is automatically accessed from the header of the request and validated

If authentication fails, the error is caught using a try-except block, and a None is returned. When None is returned from this function, an error is returned to the front end

Makes sure that the user accessing the view has the appropriate permission assigned by the role

Returns the user, indicating authentication success

Figure 9: *Authentication classes for views that is accessible for only Administrator*

```python
class CombinedPharmacyAndAdminAuthentication(authentication.BaseAuthentication):

    def authenticate(self, request):

        JWT_authenticator = JWTAuthentication()

        try:
            response = JWT_authenticator.authenticate(request)
        except exceptions.AuthenticationFailed:
            return None

        if response is not None:
            user, token = response
        else:
            return None

        # Perform additional checks for admin privileges.
        if user.role != 'Administrator' and user.role != 'Pharmacy':
            raise exceptions.PermissionDenied("Unauthorized access.")

        return (user, None)
```

Uses the same logic as an authentication class for a single role. However, in a combined authentication classes, multiple roles are checked for validity

If the user making the request has an administrator or pharmacy profile, access is granted

Figure 10: *Authentication classes for views that are accessible for both Administrator and Pharmacy profiles*

Folder Structure

The folder structure is organized to maintain modularity and scalability, ensuring a clear separation of concerns between components, views, assets, and backend interactions.

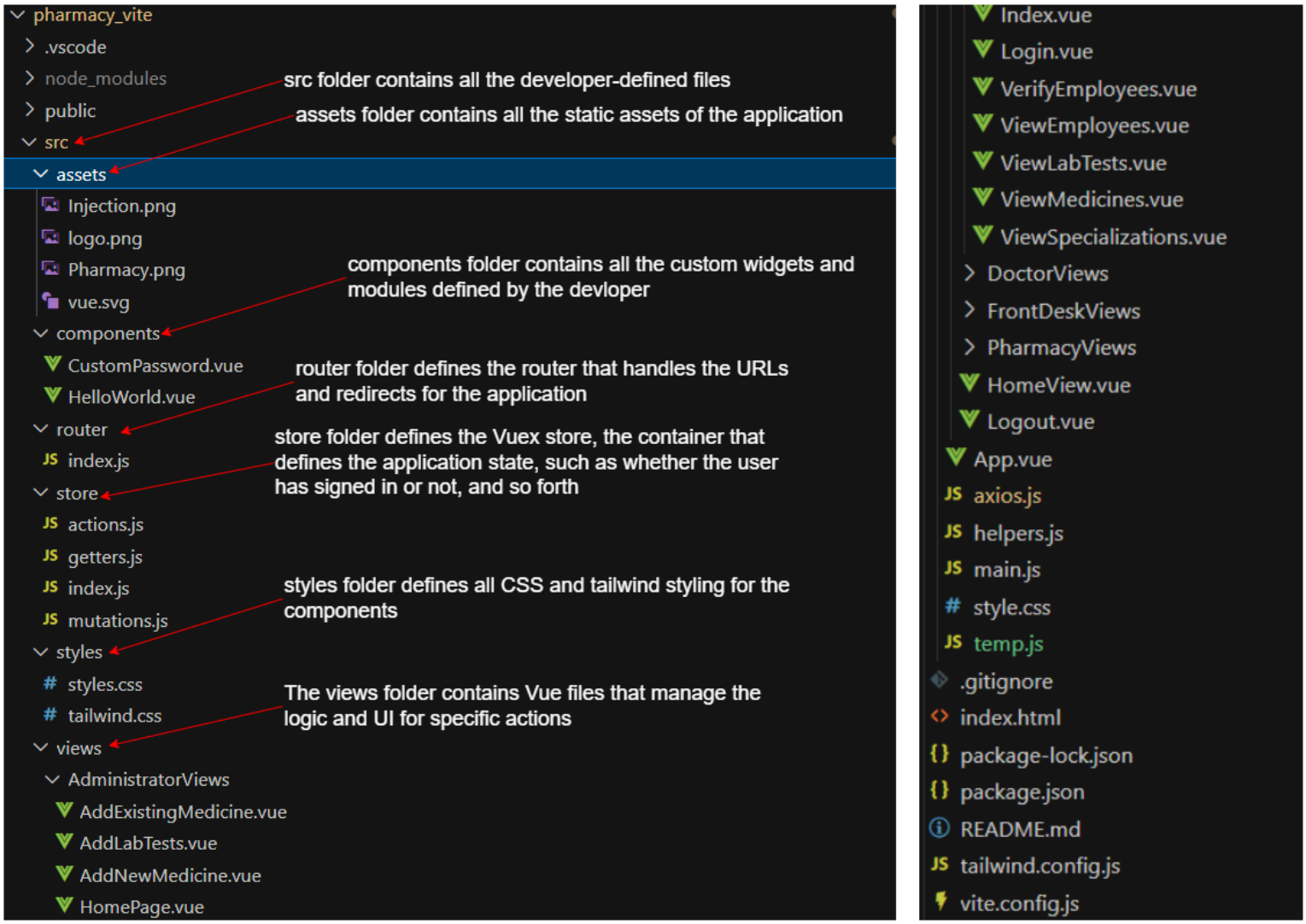

Figure 11: *Folder structure for the front end application*

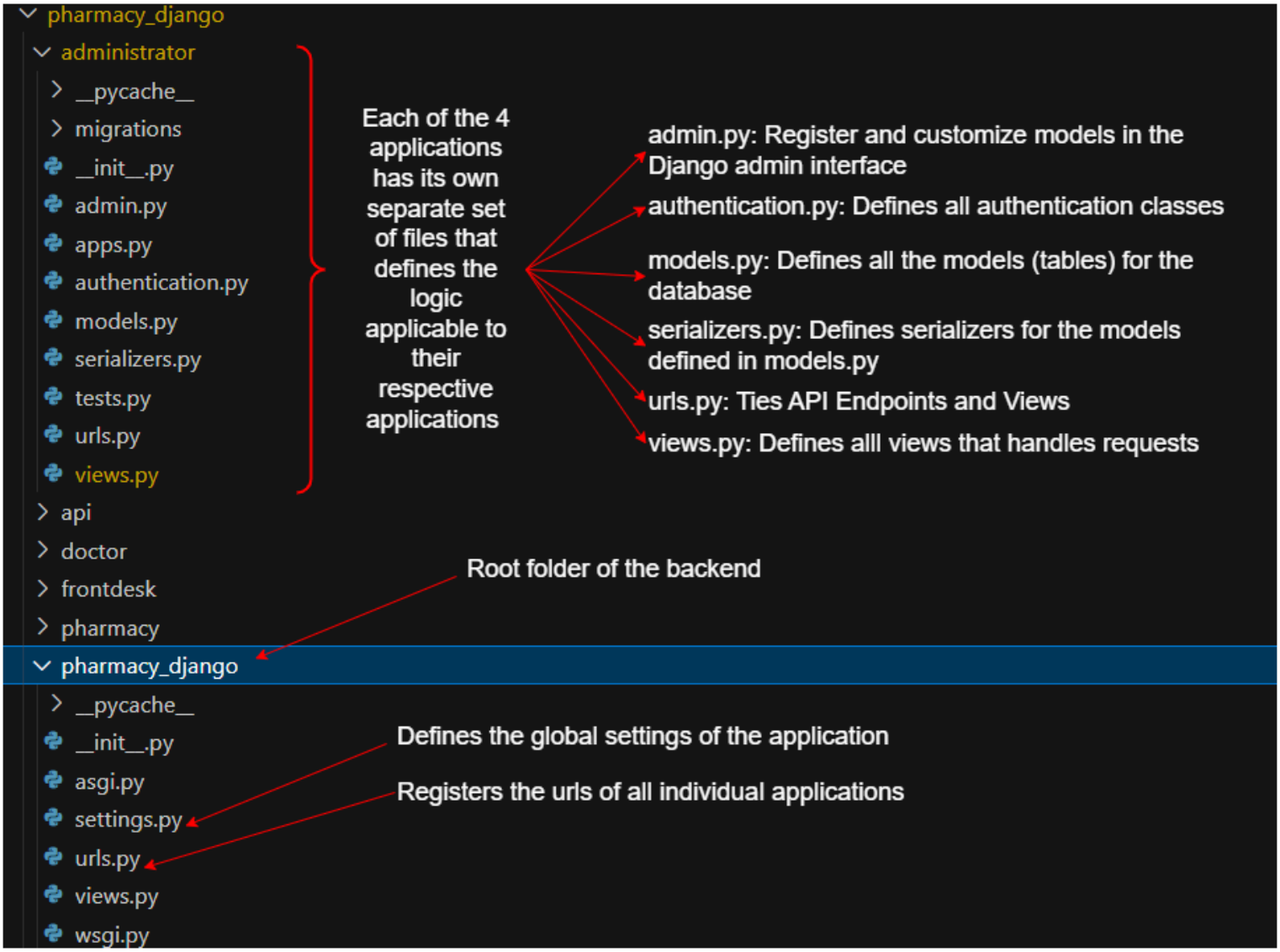

Figure 12: *Folder structure for the back end application*

Front end

Sign In

The Sign-in page allows the user to enter his/her details into the system. Until the user is verified by the administrator, the account created cannot be used for log in. Additionally, the administrator account can only be created in the View provided by the Rest Framework API in the backend, so that only the developer can create an administrator account. Unauthorized access to an Admin account can lead to security breaches, data manipulation, and potential loss of system integrity.

Sign In Page for Administrator Application

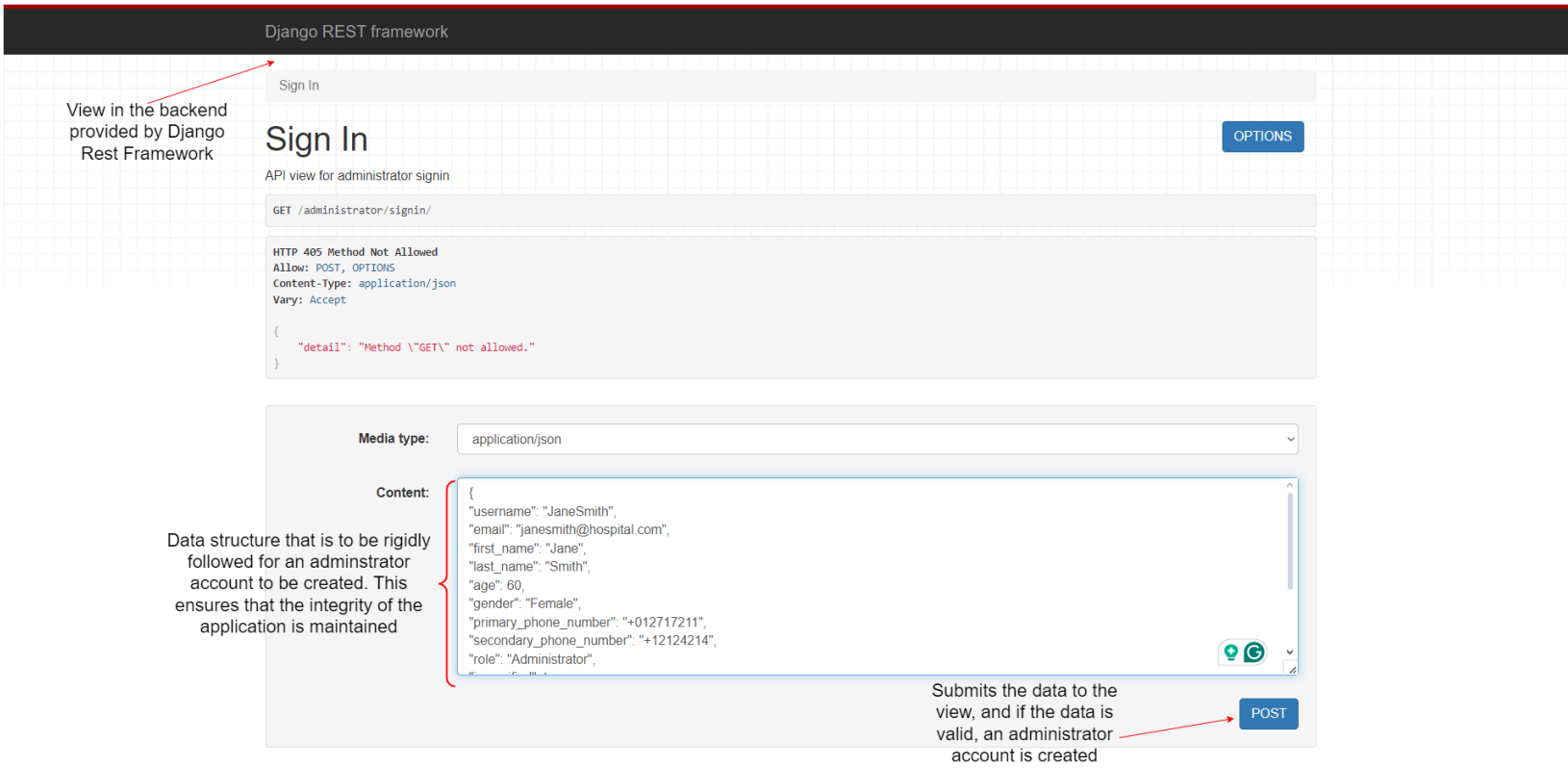

Figure 13: *Sign in page for creating an administrator user*

Sign In Page for Other Profiles

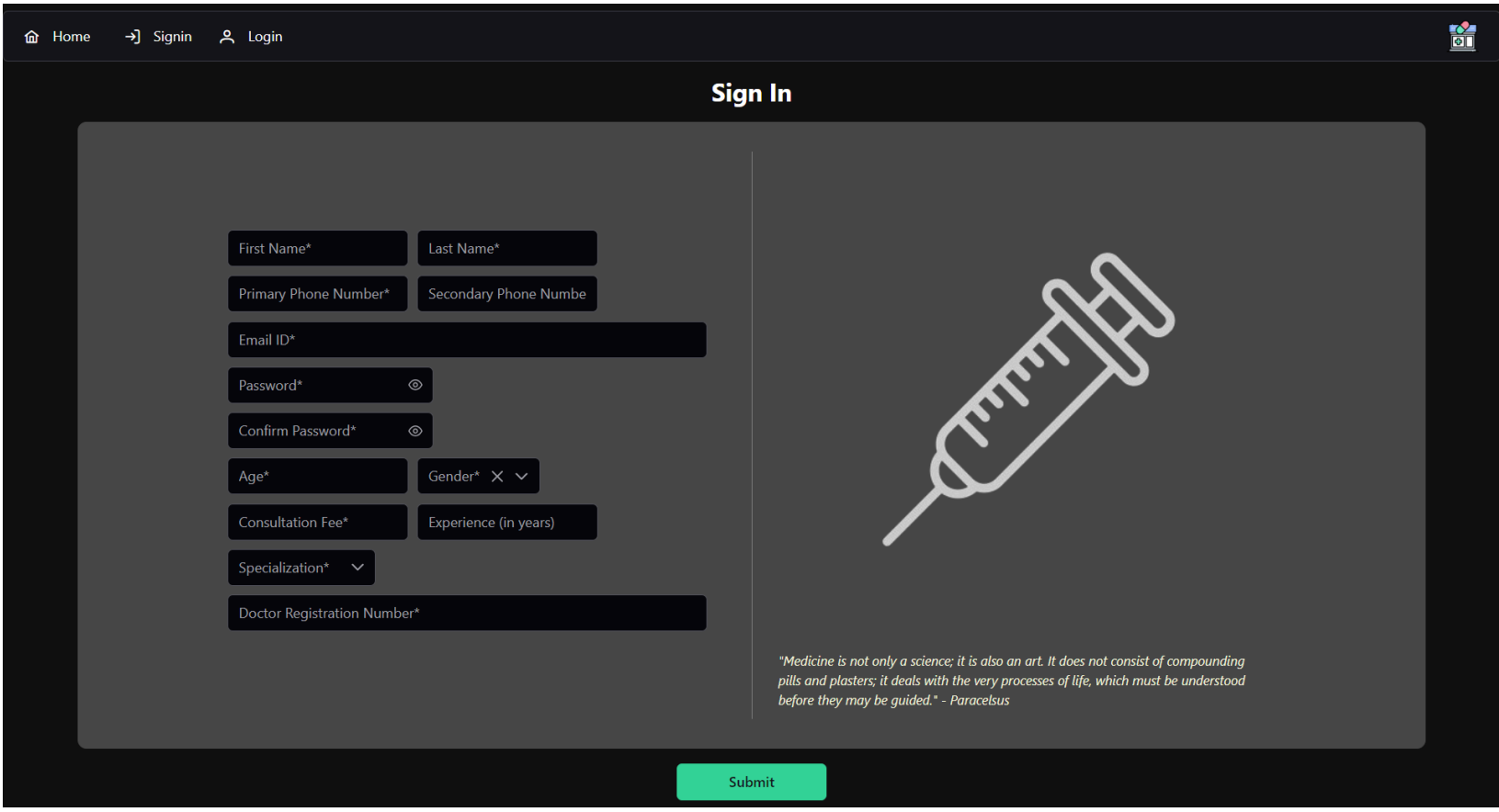

Figure 14: *Sign In page for creating a doctor user*

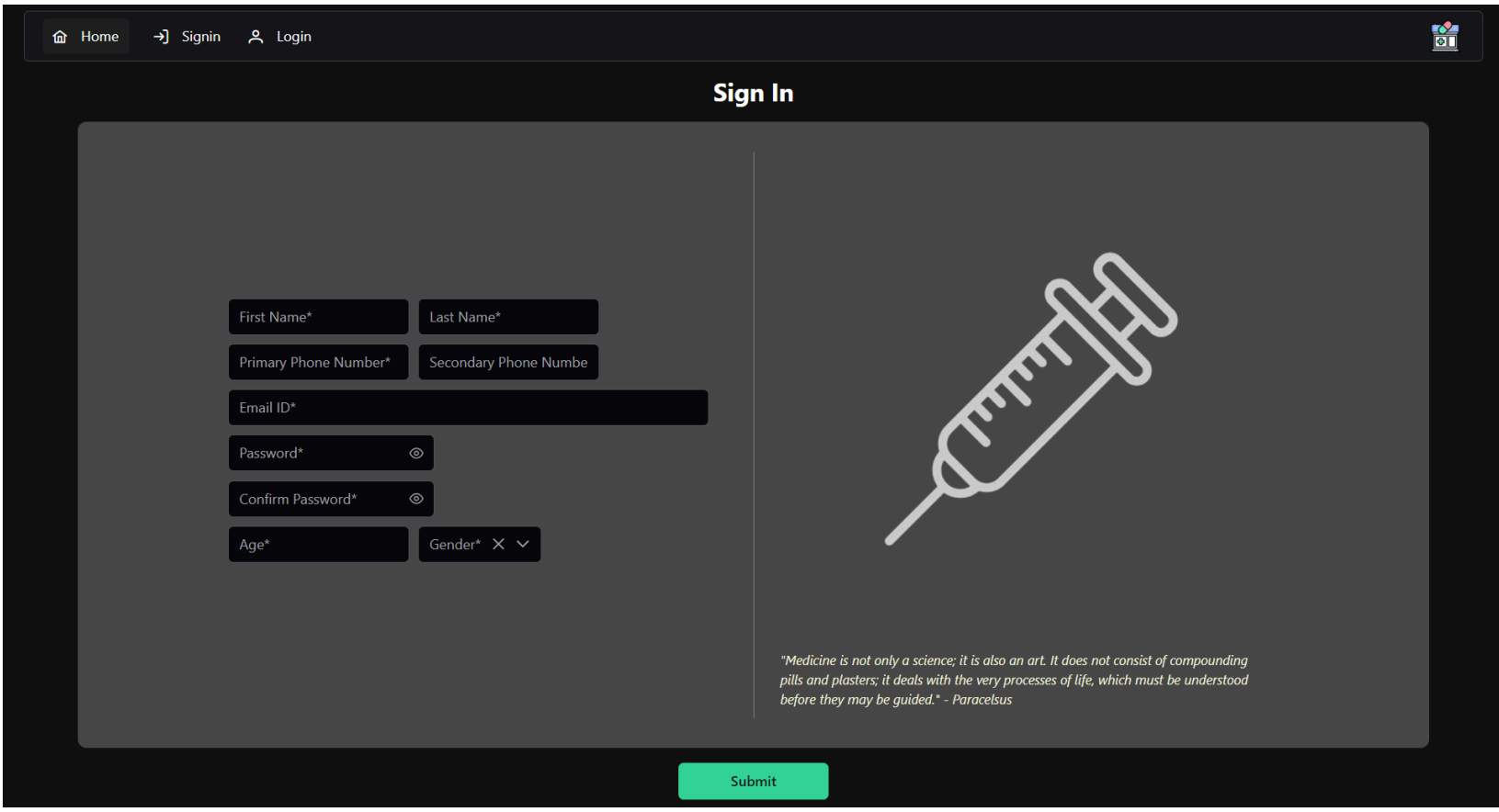

Figure 15: *Sign page for creating either a pharmacy or front desk user. Both have different pages but have the same fields*

Log In

Once the user is verified by the administrator, the account can be used to log in to their respective profiles. Except for the doctor profile which requires an additional registration number, the other profiles require only their name and the password they entered when creating the account.

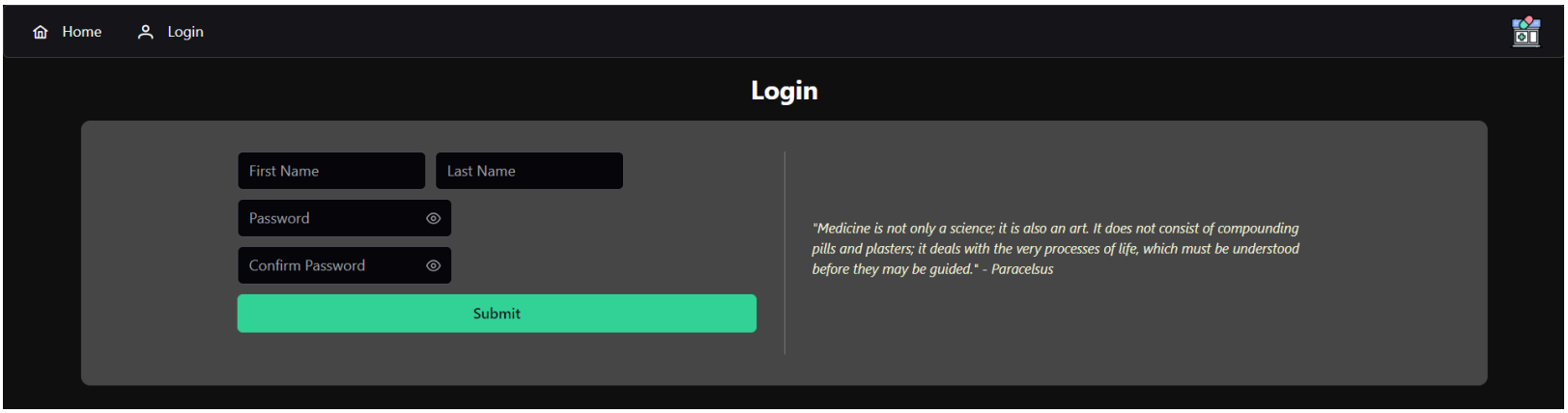

Figure 16: *Login page for administrator, pharmacy, and front desk profiles. They all have different pages to log in to their respective profiles*

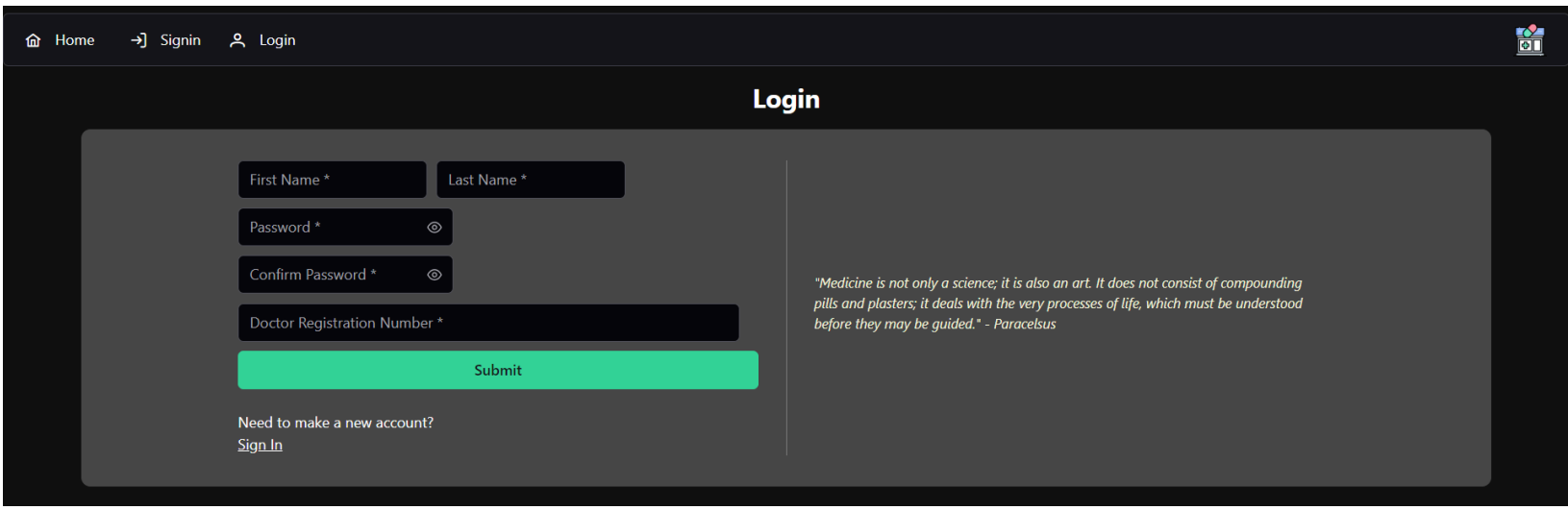

Figure 17: *Login page for doctor profile to gain access to their respective profile.*

Home Page for application

The pages are designed to maintain a consistent user experience while being reliable and easy to navigate. Modern elements from PrimeVue such as Carousel and DataTables have been used

throughout these homepages to easily display information and display all the options available to the user based on the profiles they have selected.

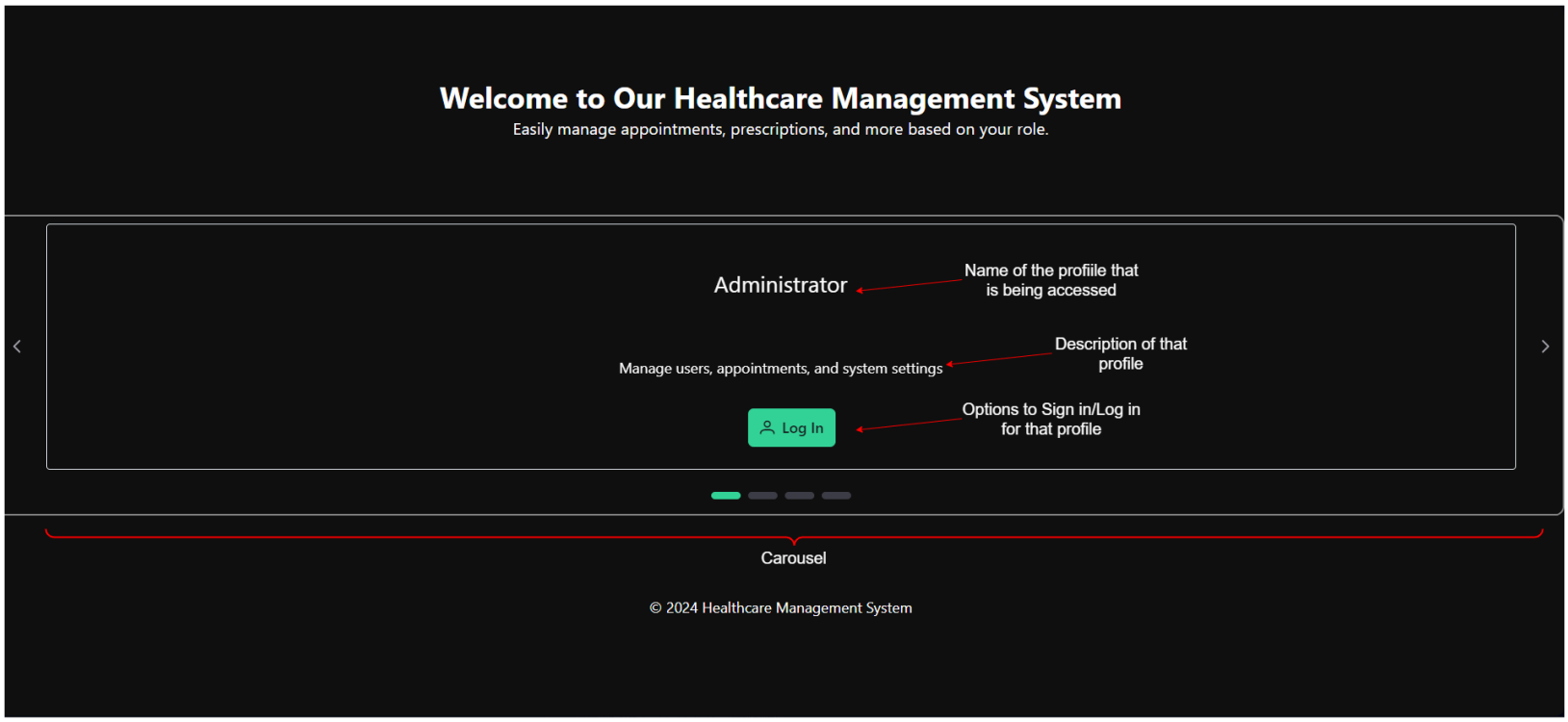

Figure 13: *Home Page of the application*

```
const options = ref([
    {
        'name': 'Administrator',
        'description': 'Manage users, appointments, and system settings',
        'loginroute': 'AdministratorLogin'
    },
    {
        'name': 'Doctor',
        'description': 'View and manage patient appointments',
        'loginroute': 'DoctorLogin',
        'siginroute': 'DoctorSignin'
    },
    {
        'name': 'Front Desk',
        'description': 'Handle front desk tasks like scheduling and patient check-ins',
        'loginroute': 'FrontDeskLogin',
        'siginroute': 'FrontDeskSignin'
    },
    {
        'name': 'Pharmacy',
        'description': 'Manage prescriptions, medicine inventory, and dispense medications to patients',
        'loginroute': 'PharmacyLogin',
        'siginroute': 'PharmacySignin'
    }
])
```

The header that is displayed in each slide of the carousel

The route that is used when the login button is clicked

The description that is displayed with the header in each slide of the carousel

The route that is used when the Signin button is clicked. The Admin does not have a signin page in the front end of the application

Figure 14: *Variable that stores the data to be used by the carousel*

```
<Carousel :value="options" :numVisible="1" :numScroll="1" circular :autoplayInterval="10000">
    <template #item="slotProps">
        <div class="relative border border-surface-200 dark:border-surface-700 rounded m-2 p-4 bg-cover bg-center">
            <div class="flex flex-column items-center justify-center h-full">
                <div class="text-white text-2xl p-4 rounded">{{ slotProps.data.name }}</div>
                <div class="mt-10 text-l">{{ slotProps.data.description }}</div>
                <span class="mt-5">
                    <Button icon="pi pi-sign-in" severity="secondary" label="Sign In"
                    @click.prevent="$router.push({ name: slotProps.data.siginroute })" v-if="slotProps.data.siginroute"/>
                    <Button icon="pi pi-user" label="Log In" class="ml-4"
                    @click.prevent="$router.push({ name: slotProps.data.loginroute })"/>
                </span>
            </div>
        </div>
    </template>
</Carousel>
```

Gets through each index of the array passed in the :value parameter.

Defines the content displayed in each slide of the Carousel. The data accessed through slotProps is defined above under the <script setup> tag.

Figure 15: *Carousel in the home page of the application for displaying different profiles*

Home Page for profiles

This section describes the home pages that are displayed for each profile. While only the Administrator profile is elaborated in detail, the rest of the profiles follow similar structure and logic.

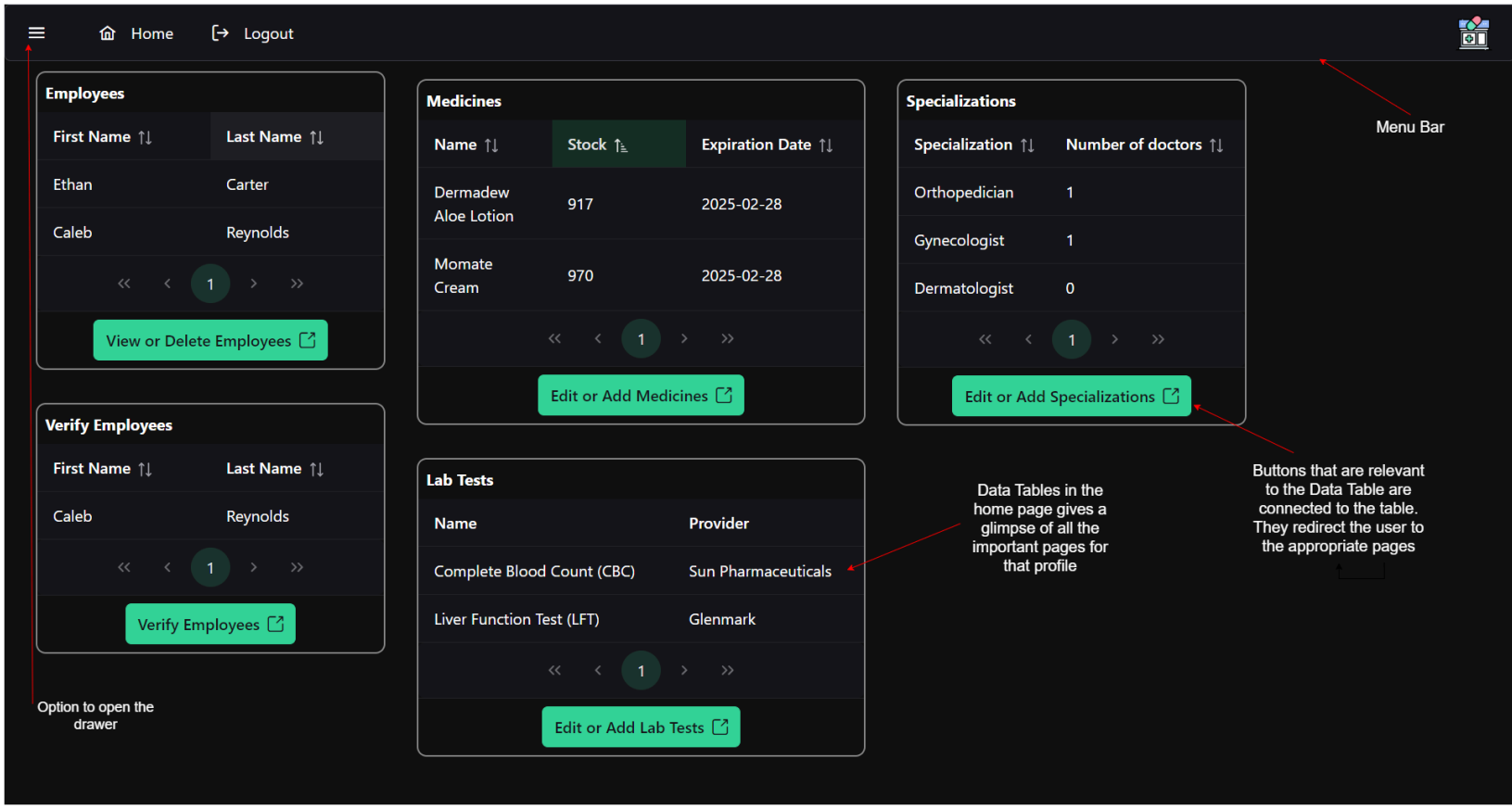

Figure 16: *Home Page of the administrator profile*

```
<template>
    <Toast/>
    <div class="flex flex-row space-y-2" v-if="is_registered === 'true'">
        <div class="flex flex-column">
            <div class="mb-4">
                <div class="card ml-5">
                    <h1 class="text-l font-bold m-2">Employees</h1>
                    <DataTable v-if="is_loaded[0] & employees_data.length > 0" :value="employees_data"
                        removableSort :rows="3"
                        paginator tableStyle="min-width: 22rem">
                        <Column field="first_name" header="First Name" style="width: 20%" sortable></Column>
                        <Column field="last_name" header="Last Name" style="width: 20%" sortable></Column>
                    </DataTable>
                    <div v-else-if="employees_data.length == 0 && is_loaded[0]" class="centered placeholder-table" style="min-width: 20rem; padding:1rem">
                        There are no employees in this system.
                    </div>

                    <div class="centered" v-else>
                        <ProgressSpinner/>
                    </div>

                    <div class="centered">
                        <Button label="View or Delete Employees" icon="pi pi-external-link"  iconPos="right"
                        @click="$router.push({ name: 'AdministratorViewEmployees' })" style="margin: 0.5rem"/>
                    </div>
                </div>
            </div>
```

The elements directly below this div are arrange horizontally

The entire page is only loaded if the user is logged in

This div is visible only if the requests have been made and there are values to be showcased in the table

Enables the table to be sorted based on alphabetical order of one of the columns

PrimeVue's DataTables are used to display data in a tabular format

Each column of the DataTable that directly processes the input data into a tabular format

This div is visible only if the requests have been made and there are no values in the response data.

This div (a spinner) is visible if the data is being fetched from the backend

Button that redirects the user to the action that is relevant to the data table. In this case, viewing the employees details or deleting them

Figure 17: *Code for the first data table in the Administrator profile*

```
<script setup>
    import axios from '../../axios';
    import { useStore } from 'vuex';
    import { ref } from 'vue';
    import { useToast } from 'primevue/usetoast';
    import '../../styles/styles.css';

    const unverified_users_data = ref([]);
    const medicine_inventory_data = ref([]);
    const employees_data = ref([]);
    const lab_tests_data = ref([]);
    const specializations_data = ref([]);

    // Ordered this way: unverified users, employees, medicines, labtests
    const is_loaded = ref([false, false, false, false, false]);

    const store = useStore();
    store.dispatch('initializeStore');

    const toast = useToast();

    const warn = (warn, summary, detailed) => {…
    }

    const is_registered = ref(store.state.is_registered)

    if (is_registered.value === 'true') {
        axios.get('administrator/verifyEmployees/')
        .then( (response) => {…
        })
        .catch( (error) => {…
        })

        axios.get('administrator/getEmployees/')
        .then( (response) => {…
        })
        .catch( (error) => {…
        })

        // To get data for inventory
        axios.get('/administrator/getMedicines/')
        .then( (response) => {…
        })
        .catch( (error) => {…
        })

        axios.get('/administrator/getLabTests/')
        .then( (response) => {…
        })
        .catch( (error) => {…
        })

        axios.get('/administrator/getSpecializations/')
        .then( (response) => {…
        })
        .catch( (error) => {…
        })
    } else {
        warn('warn', "Log in using an admin account to access this page.", '');
        router.push({ name: 'Home' })
    }
</script>
```

Variables used throughout the program

a variable that keeps track of the status of all requests

Initializes the vuex store to gain access to the application state

Function that generates the notifications

If the user at least has the access token and refresh token to access the views

Makes all the requests to the backend to get the required data to be displayed

If the user does not have the necessary credentials

Figure 18: *Javascript code for the Homepage of the administrator profile*

Home Page for the other profiles

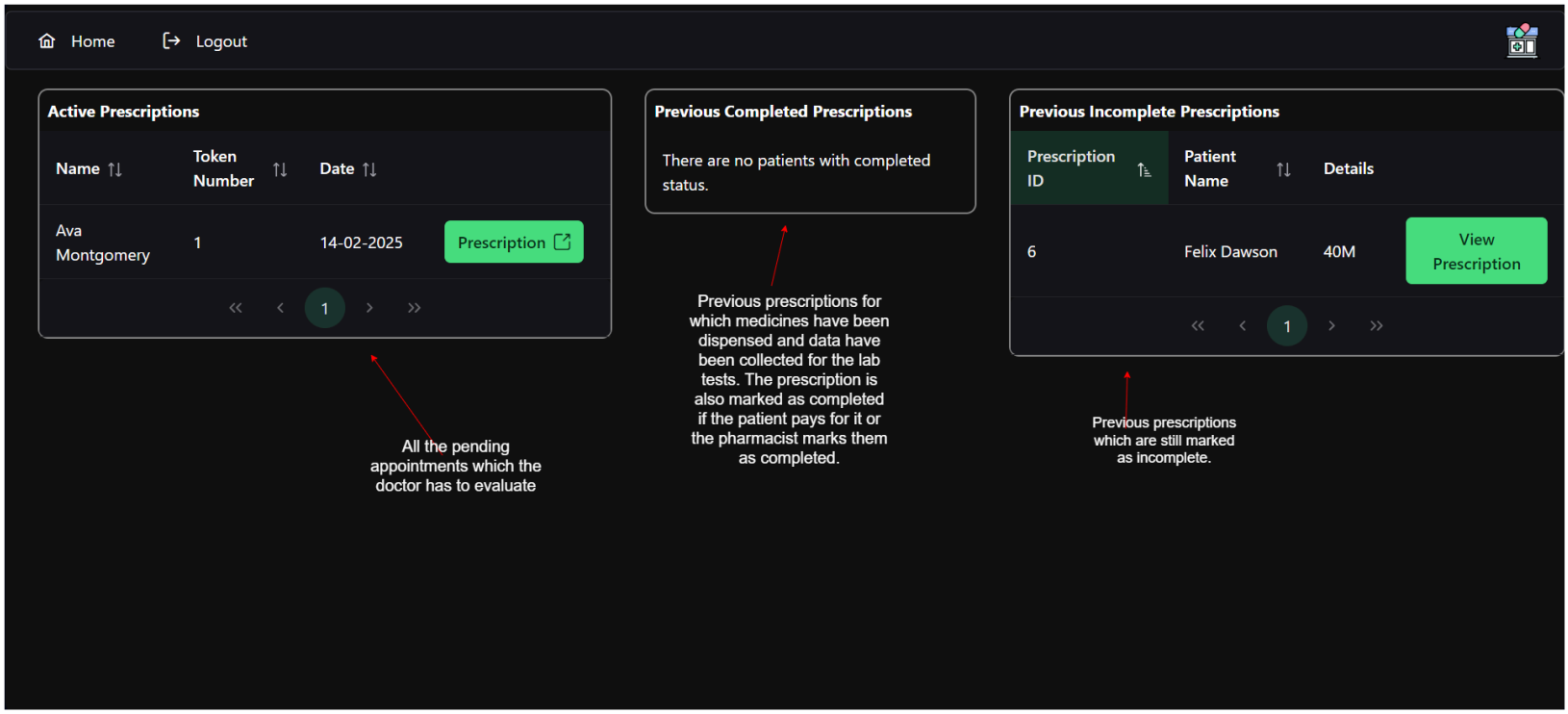

Figure 19: *Home Page for the doctor profile*

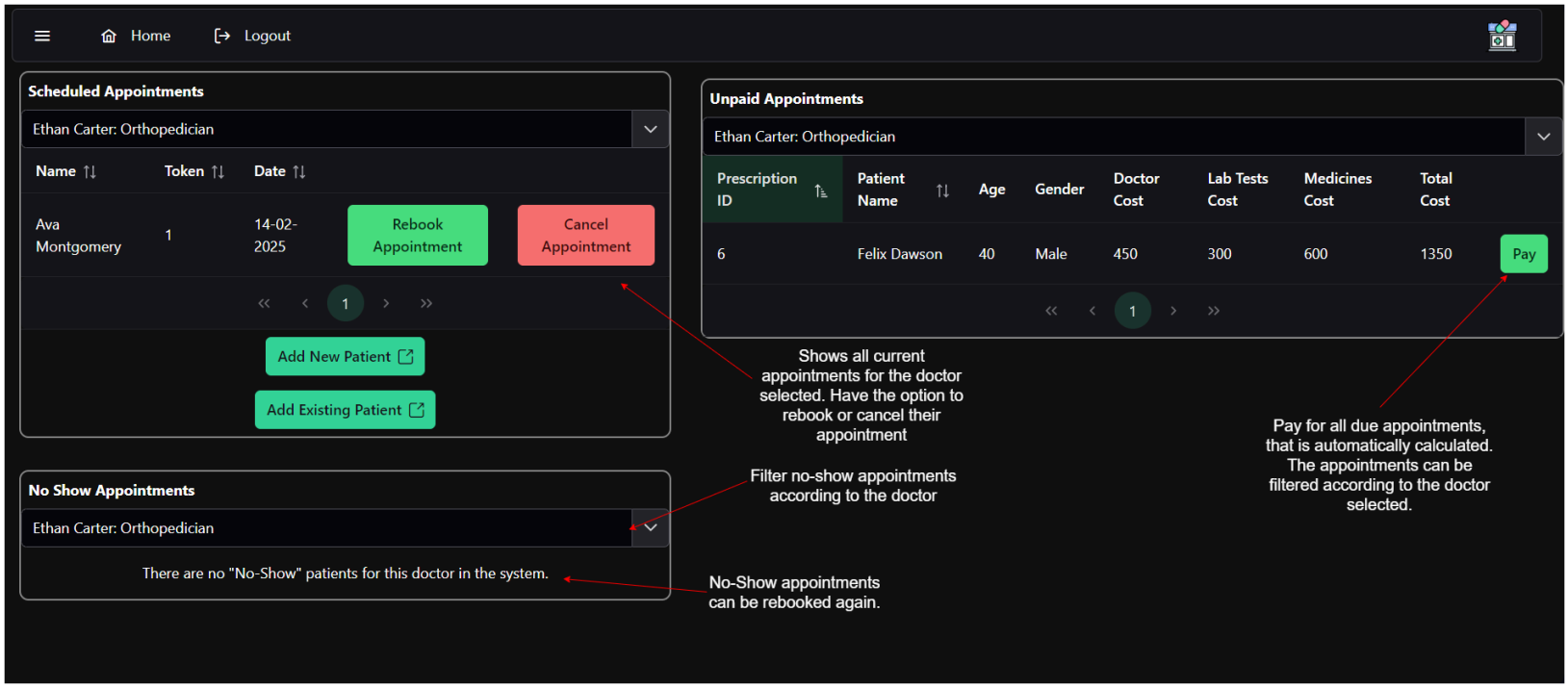

Figure 20: *Home page for the front desk profile*

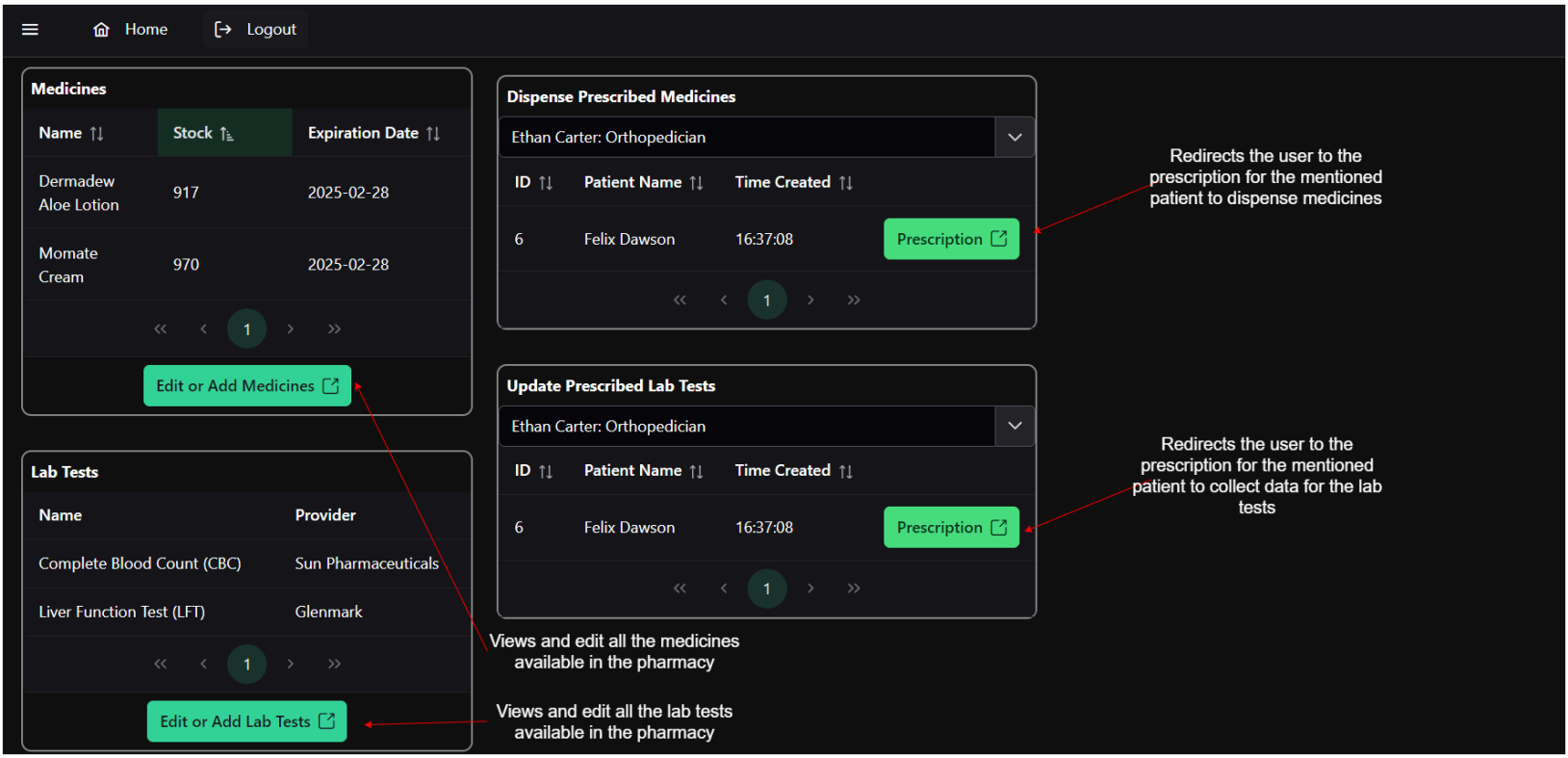

Figure 21: *Home page for the pharmacy home page*

Menu Bar and Drawer

The menu Bar is the top bar located at the page, useful for accessing all the important pages at a glance. The Drawer can be opened from the Menu Bar and pops up from the left side of the page, giving access to all views for the profile at once.

Menu Bar and Drawer are present for all profiles and are modified according to their needs. However, the Doctor's profile does not have a Drawer since all the views can be accessed from the Home page itself. Both of these components are defined in the **Index.vue** file

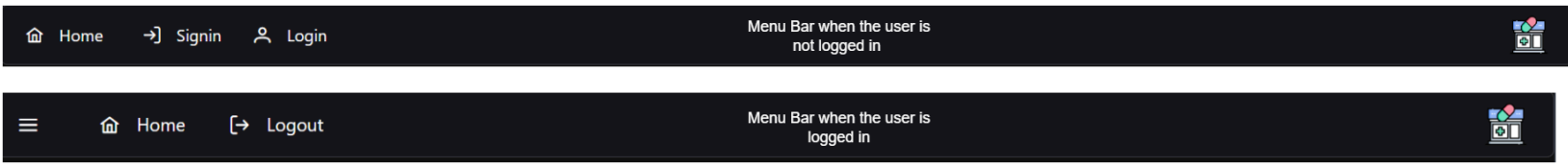

Figure 22: *Menu Bar when the user is logged out (top) and the user is logged in (bottom)*

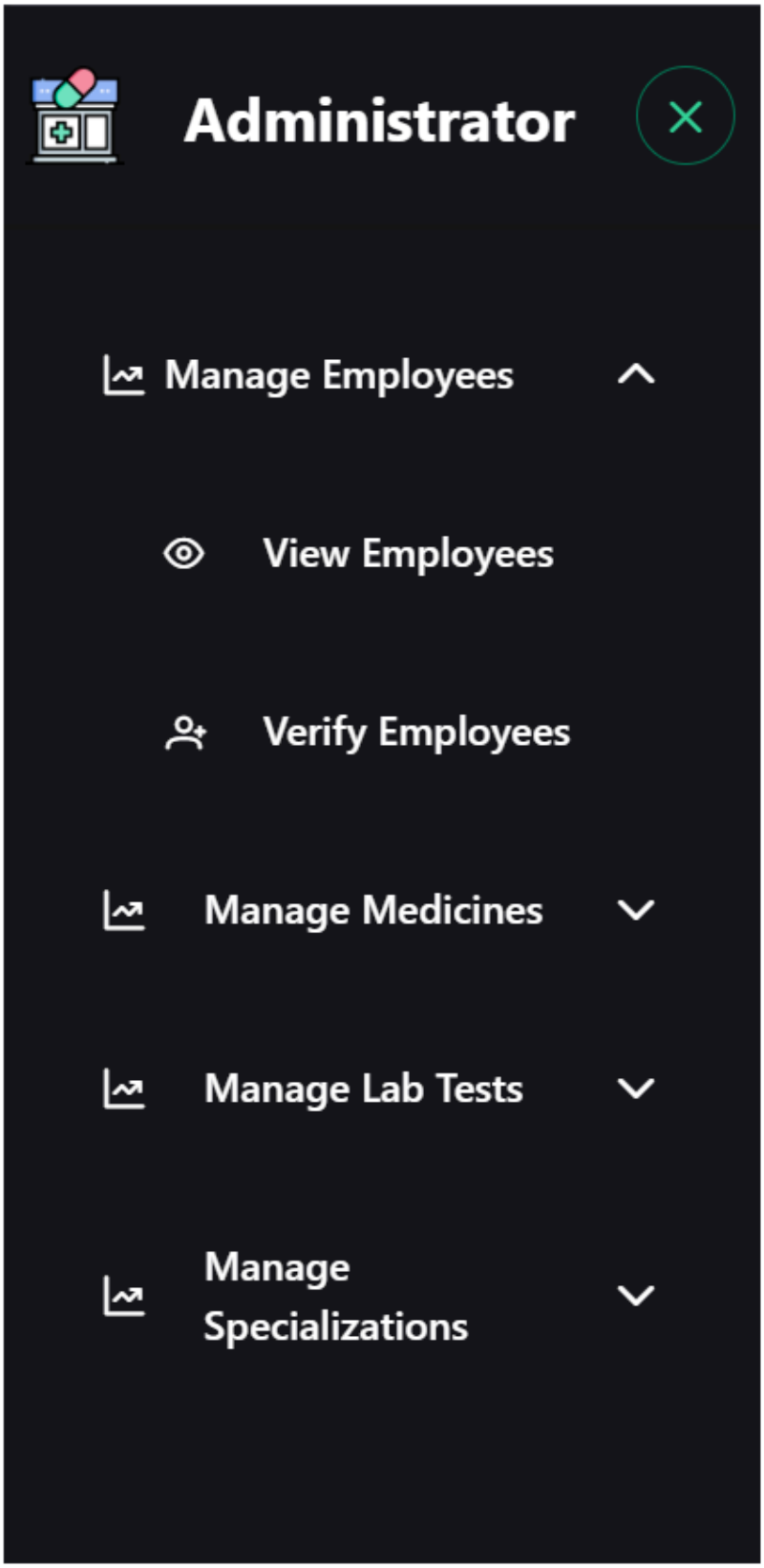

Figure 23: *Drawer for the administrator profile with the Manage Employees dropdown open*

```
const store = useStore();
const drawer_visible = ref(false);
const dropdown_open_employees = ref(false);
const dropdown_open_medicines = ref(false);
const dropdown_open_lab_tests = ref(false);
const dropdown_open_specializations = ref(false);

store.dispatch('initializeStore');

const items = ref([
    {
        label: '',
        icon: 'pi pi-bars',
        loggedIn: true,
        loggedOut: false,
        command: () => {
            drawer_visible.value = true;
        }
    },
    {
        label: 'Home',
        icon: 'pi pi-home',
        loggedIn: false,
        loggedOut: true,
        command: () => {
            router.push({ name: 'Home' })
        }
    },
```

Variables that control whether the drawer is open or not, and whether each dropdown in the drawer is open or not

The definition of one item that is shown in the Menu Bar

Name of the button that is shown

Whether the button is shown when the user is logged in

Whether the button is shown when the user is logged out

The action to execute when the button is clicked

Figure 24: *Variables and Data structures used for the Drawer and Menu Bar for the administrator profile*

```
const logged_in = computed(() => store.state.is_registered === 'true');

const toggleDropdownEmployees = () => {
    dropdown_open_employees.value = !dropdown_open_employees.value;
}

const toggleDropdownMedicines = () => {
    dropdown_open_medicines.value = !dropdown_open_medicines.value;
}

const toggleDropdownLabTests = () => {
    dropdown_open_lab_tests.value = !dropdown_open_lab_tests.value
}

const toggleDropdownSpecializations = () => {
    dropdown_open_specializations.value = !dropdown_open_specializations.value
}

const redirect = (url_name) => {
    router.push({ name: url_name})
    drawer_visible.value = false
}
```

If the user logs in, the variable refreshes, thus refreshing the menu bar to make the updates to the button

Toggles the dropwdowns that are available in the Drawer

This function is used for the items in the drawer and the menu bar, to redirect the user based on a click and close the drawer

Figure 25: *Methods used for the Drawer and Menu Bar for the administrator profile*

```
<Drawer v-model:visible="drawer_visible">
    <template #container="{ closeCallback }">
        <div class="flex flex-col h-full">
            <div class="flex items-center justify-between px-3 py-4 shadow-md">
                <div class="inline-flex items-center gap-4">
                    <img src="../../assets/Pharmacy.png" alt="Pharmacy Logo" class="h-10 w-10" />
                    <h1 class="font-bold text-2xl text-primary">Pharmacy</h1>
                </div>
                <Button type="button" @click="closeCallback" icon="pi pi-times" rounded outlined></Button>
            </div>

            <div class="overflow-y-auto">
                <ul class="list-none p-4 m-0">
                    <li>
                        <a
                            @click="toggleDropdownMedicines"
                            class="flex items-center cursor-pointer p-4 rounded text-surface-700 hover:bg-surface-100 dark:text-surfac
                        >
                            <i class="pi pi-chart-line mr-4"></i>
                            <span class="font-medium">Manage Medicines</span>
                            <i class="pi pi-chevron-down ml-auto transform transition-transform duration-600" :class="{ 'rotate-180':
                        </a>
                        <ul v-show="dropdown_open_medicines" class="list-none py-0 pl-4 pr-0 m-0 overflow-y-hidden transition-all dura
                            <li>
                                <a
                                    class="flex items-center cursor-pointer p-4"
                                    @click="redirect('PharmacyViewMedicines')"
                                >

                                    <i class="pi pi-eye mr-4"></i>
                                    <span class="font-medium">View Medicines Inventory</span>
                                </a>
                            </li>
                        </ul>
```

Button to close the drawer

Toggles the dropdown for all the medicines features in the administrator page

This redirect is performed on the click of the hyperlink

This option is shown when the dropdown is opened, and can be clicked on to be redirected to the page

Figure 26: *Code for the drawer instance of the administrator profile*

```
<Menubar :model="items">
    <template #item="{ item, props }">
        <a v-if="item.loggedIn == logged_in || item.loggedOut == !logged_in" :target="item.target" v-bind="props.action">
            <span :class="item.icon" />
            <span class="ml-2">{{ item.label }}</span>
        </a>
    </template>
    <template #end>
        <svg width="50" height="40">
            <image href="../../assets/Pharmacy.png" x="2" y="2" height="36" width="36"/>
        </svg>
    </template>
</Menubar>
```

Shows the item and its icon

Goes through all the items and checks which items to display and which items to not based on whether the user is logged in

Shows the picture of a phrmacy at the end of the menu bar

Figure 27: *Code for the Menu Bar instance of the administrator profile*

View Employees

Throughout the application, most of the data can be viewed and edited through "View" pages, as shown in Figure 28. The foundational logic for all of these pages are roughly the same, with different requests and additional logic for extra features that are required by that page. For example, there is an option to edit the values in "View Specializations" view for the administrator.

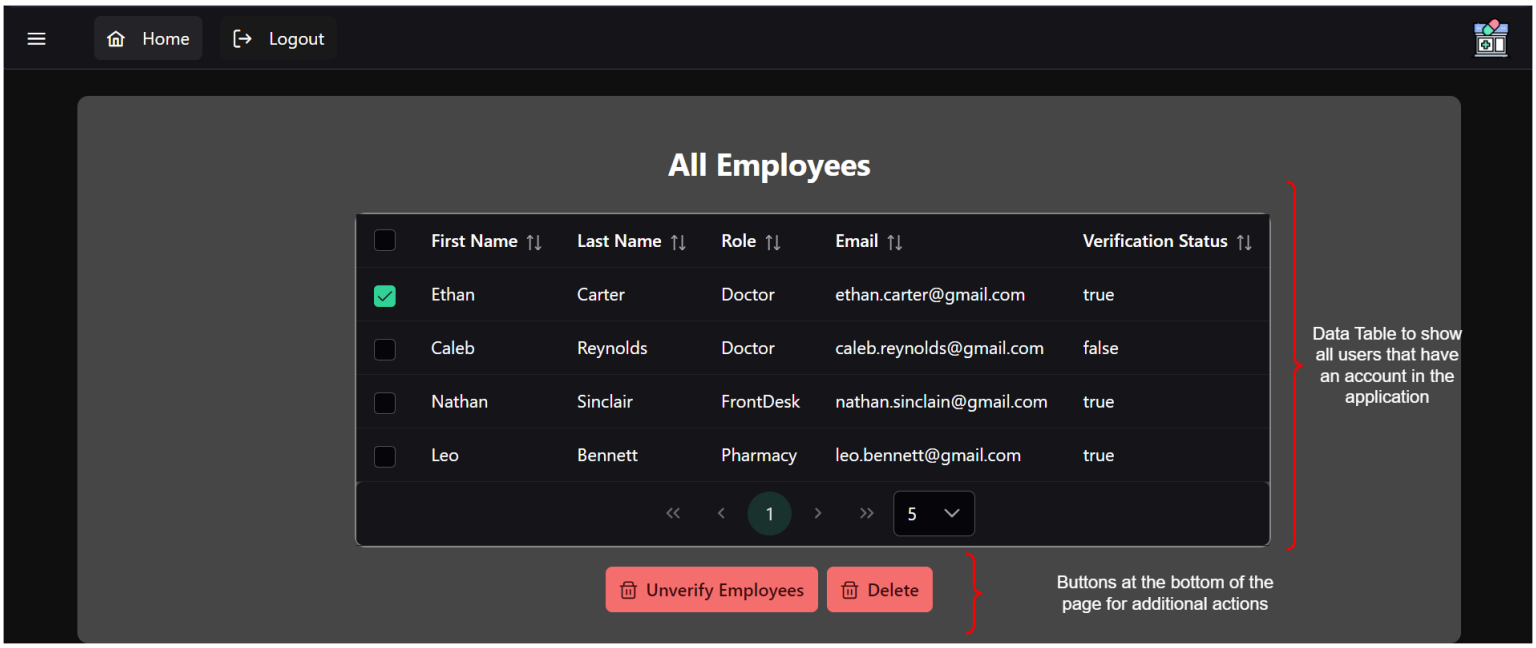

Figure 28: *GUI for viewing the employees in the administrator profile*

```
<div class="top-container mt-4" v-if="data.length >= 0">
    <div class="container">
        <div class="centered">
            <h1 class="text-3xl font-bold m-3">All Employees</h1>
        </div>

        <div class="sub-container" style="margin-left:7rem;" v-if="data.length > 0">
            <div class="card">
                <DataTable :value="data" v-model:selection="selected" datakey="id" removableSort paginator :rows="5" :rowsPerPageOptio
                    <Column selectionMode="multiple" style="width: 3rem"></Column>
                    <Column field="first_name" header="First Name" style="width: 20%" sortable></Column>
                    <Column field="last_name" header="Last Name" style="width: 10%" sortable></Column>
                    <Column field="role" header="Role" style="width: 20%" sortable></Column>
                    <Column field="email" header="Email" style="width:40%" sortable></Column>
                    <Column field="is_verified" header="Verification Status" style="width: 10%" sortable></Column>
                </DataTable>
            </div>
        </div>

        <div class="centered" v-if="data.length == 0">
            <h1 class="text-l font-bold m-2">There are no users in the system.</h1>
        </div>

        <div class="flex flex-row align-items-center justify-content-center mt-2">
            <div v-if="data.length > 0">
                <Button label="Unverify Employees" icon="pi pi-trash" severity="danger" @click="confirmUnverification" :disabled="!sel
            </div>

            <div v-if="data.length > 0">
                <Button label="Delete" icon="pi pi-trash" severity="danger" @click="confirmDeletion" class="ml-2" :disabled="!selected
            </div>
        </div>
    </div>
</div>
```

Heading on the top of the page

Row to allow the administrator user to select user instances from the table

Data Table to show the details of the users

If the get request was successful, but there were no instances in the database

Buttons at the bottom of the page for additional actions

Figure 29: *GUI Code for building the View Employees page of the administrator profile*

```
const confirmDeletion = () => {
    deletion_dialog.value = true;
}

const confirmUnverification = () => {
    unverification_dialog.value = true;
}

const sendRequest = (code) => {
    deletion_dialog.value = false;
    unverification_dialog.value = false;
    let idArray = []

    for (let i = 0; i < selected.value.length; i++) {
        idArray.push(selected.value[i]['id'])
    }

    axios.post('/administrator/editEmployees/', { code: code, ids: idArray })
    .then( (response) => {
        warn('success', 'Successfully edited users!', '')

        setTimeout(() => {
            window.location.reload();
        }, 1000);
    })
    .catch( (error) => {
        warn('warn', 'Unsuccesful in editing users.', error)
    })
}
```

These methods are used to open the dialog box to confirm the user's action

Closes the dialog boxes

Gets all the ids of the selected users from the Data Table to be processed in the backend

Sends the request with the appropriate data to the backend for the data and informs the user accordingly, based on whether the request was successful or not.

Figure 30: *Important Methods used in the View Employees page of the administrator profile*

Dialog Boxes and Notifications

Dialog Boxes are an efficient way to confirm the user's actions, making sure it was not a misclick. If the yes button is clicked, the request is sent to the backend, and through the help of notifications, the user is kept informed about the status.

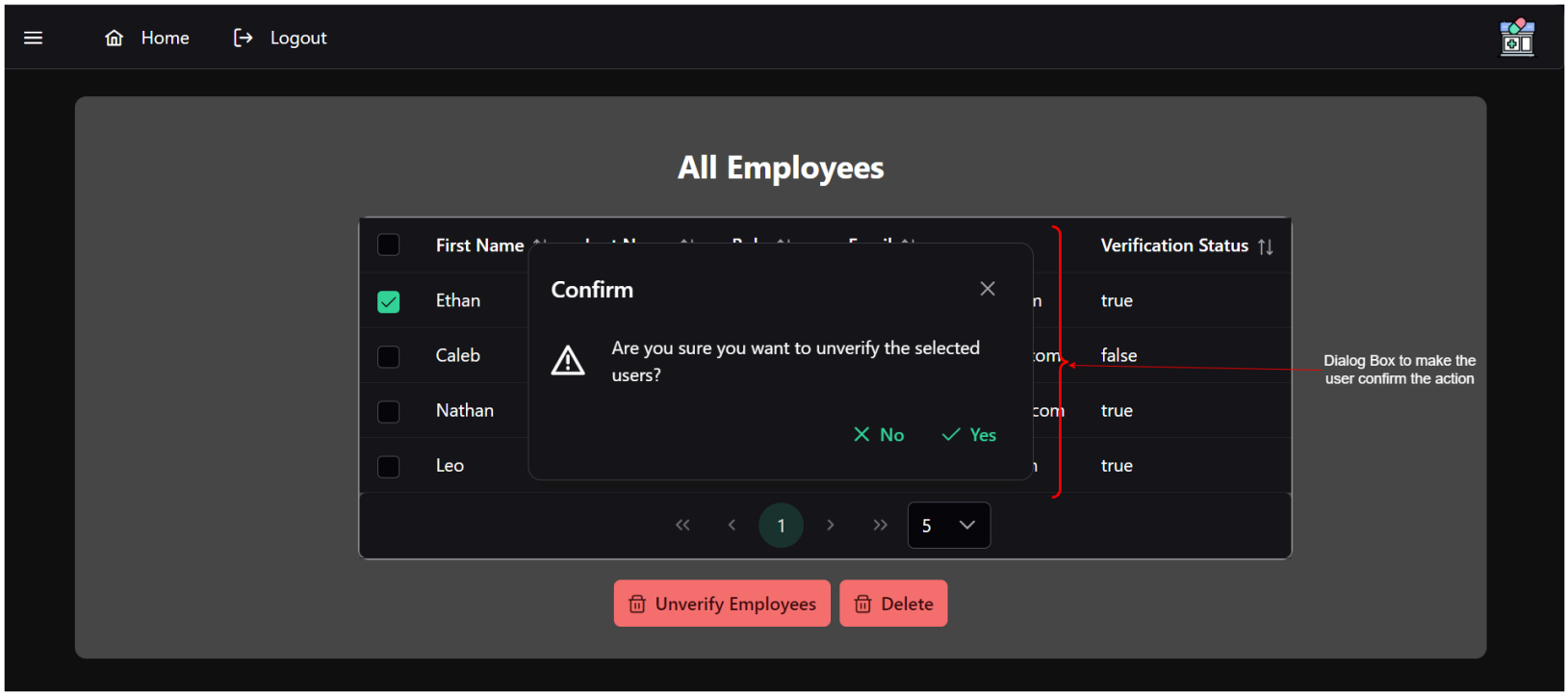

Figure 31: *Dialog Box in the administrator profile to confirm the user's actions*

```
<Dialog v-model:visible="deletion_dialog" :style="{ width: '450px' }" header="Confirm">
    <div class="flex items-center gap-4">
        <i class="pi pi-exclamation-triangle !text-3xl" />
        <span>Are you sure you want to delete the selected users?</span>
    </div>
    <template #footer>
        <Button label="No" icon="pi pi-times" text @click="deletion_dialog = false"/>
        <Button label="Yes" icon="pi pi-check" text @click="sendRequest(0)"/>
    </template>
</Dialog>

<Dialog v-model:visible="unverification_dialog" :style="{ width: '450px' }" header="Confirm">
    <div class="flex items-center gap-4">
        <i class="pi pi-exclamation-triangle !text-3xl" />
        <span>Are you sure you want to unverify the selected users?</span>
    </div>
    <template #footer>
        <Button label="No" icon="pi pi-times" text @click="unverification_dialog = false"/>
        <Button label="Yes" icon="pi pi-check" text @click="sendRequest(1)"/>
    </template>
</Dialog>
```

If no is selected, the dialog box is simply closed without sending a request

If yes is selected, a request is sent to the backend (here to unverify the selected users) and the dialog box is closed

Another dialog box that achieves a different purpose

Figure 32: *The code for the dialog boxes*

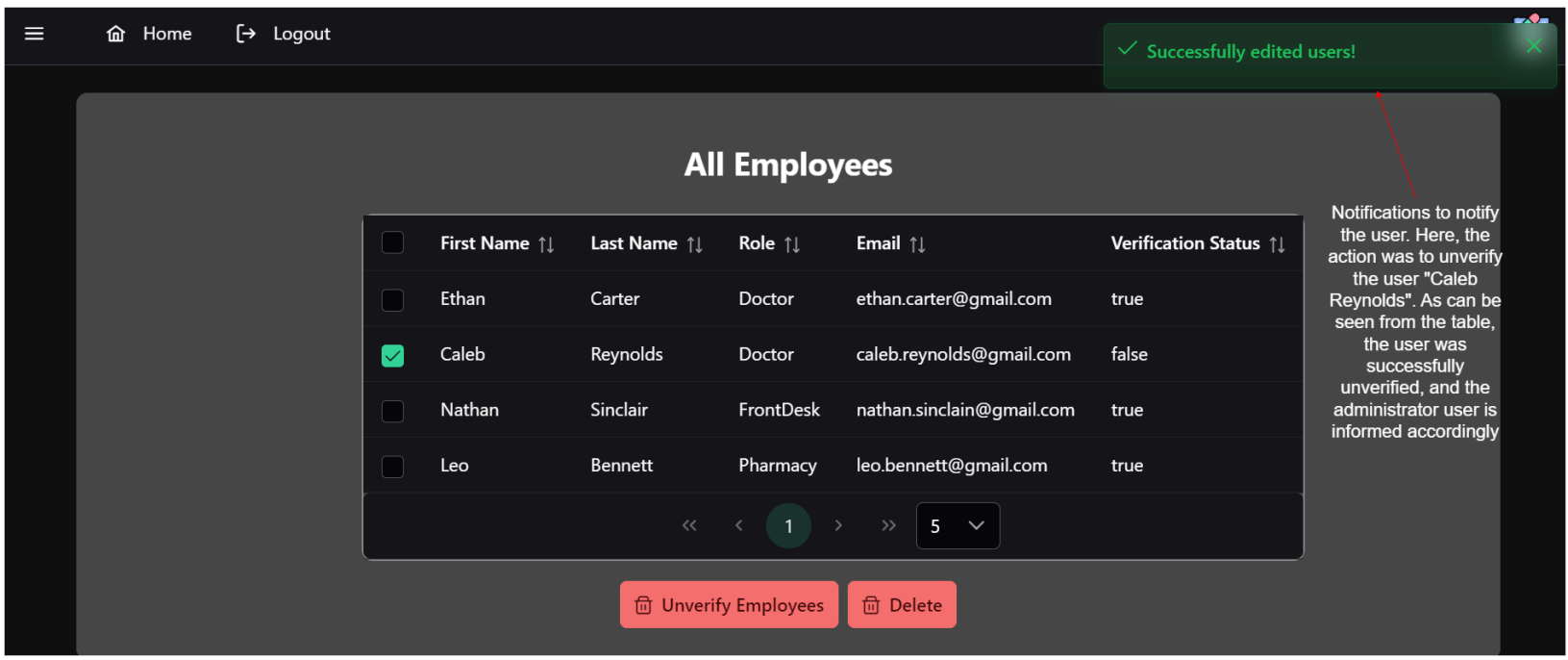

Figure 33: *The notification generated informing them of the status of the request*

```
const toast = useToast();
const warn = (severity, summary, detailed) => {
    toast.add({ severity: severity, summary: summary, detail: detailed, life: 3000 });
}

axios.post('/administrator/editEmployees/', { code: code, ids: idArray })
.then( (response) => {
    warn('success', 'Successfully edited users!', '')

    setTimeout(() => {
        window.location.reload();
    }, 1000);
})
.catch( (error) => {
    warn('warn', 'Unsuccesful in editing users.', error)
})

<Toast/>
```

A Toast object from Prime Vue is used to generate notification

Defines the settings of the toast instance. In this case, a notifcation will last for 3000 milliseconds

Generates the notification that the request was succesful

If the request was not successful, the notification informs the user with the error message that was generated from the backend

The instance under the <template> tag that generates the actual notification

Figure 34: *The code involved in generating the notification*

Add New Medicines

"Add New Medicines" page can be accessed by both pharmacy and administrator profiles to add details for a new medicine and add stock for it. Other pages in the application that create new instances entirely, such as Add New Patient for the front desk profile, use the same logic to validate and send data to the backend for processing. They differ slightly according to the special requirements of the page.

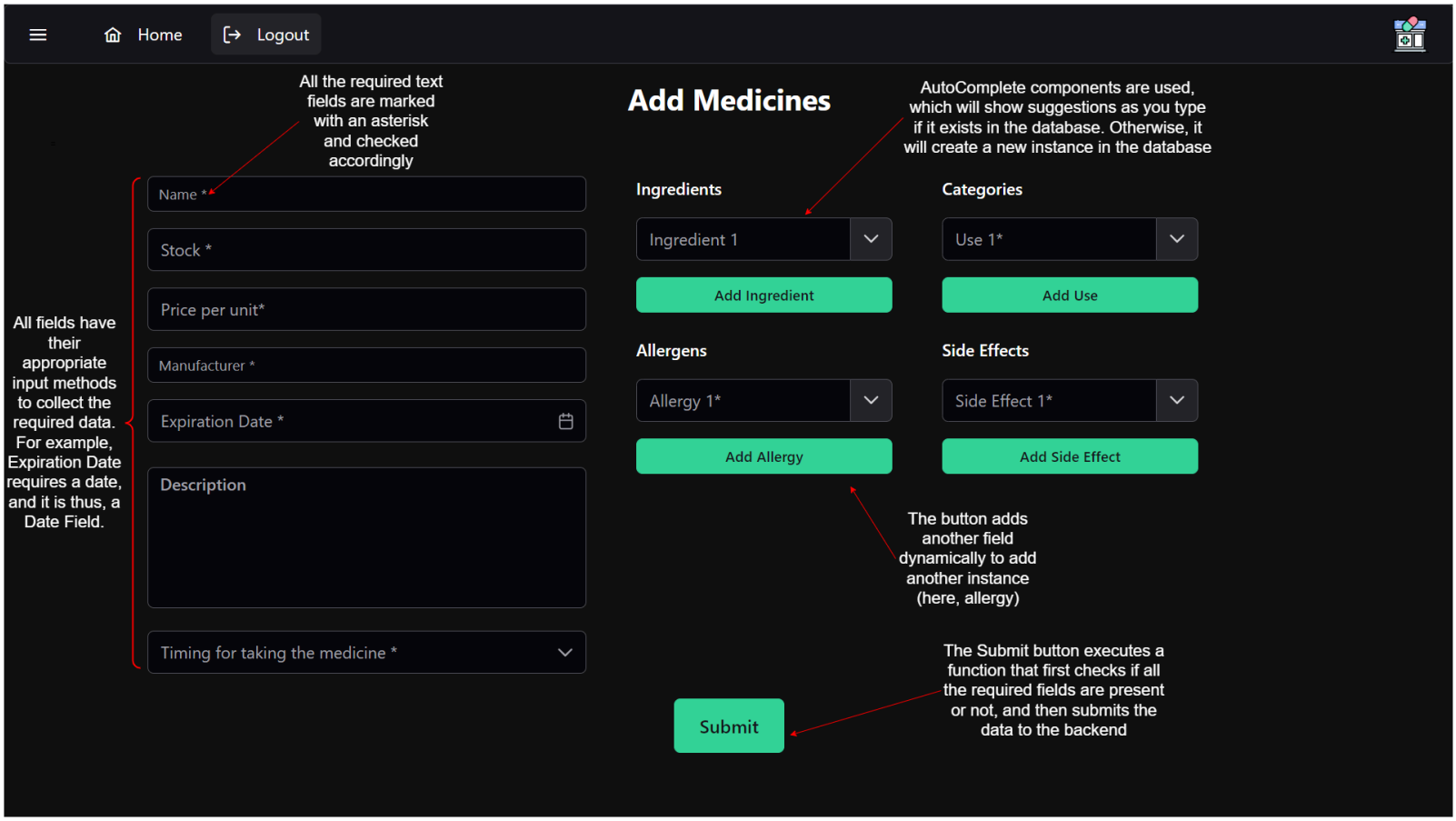

Figure 35: *GUI for Add New Medicines page in the Administrator and Pharmacy profiles*

```
<template>
    <Toast />

    <div class="centered" v-if="is_registered === 'true'">
        <h1 class="text-3xl font-bold m-3">Add Medicines</h1>
    </div>

    <div class="container mx-auto p-6 bg-grey shadow-md rounded-lg">
        <div class="grid grid-cols-1 gap-6 md:grid-cols-2">
            <div class="flex flex-col space-y-4">
                <InputText id="name" placeholder="Name *" v-model.trim="name" class="p-inputtext-sm w-full"/>
                <InputNumber id="stock" placeholder="Stock *" inputId="withoutgrouping" :useGrouping="false" v-model.number="stock" :min="0"
                class="p-inputnumber-sm w-full"/>
                <InputNumber id="price" placeholder="Price per unit*" inputId="currency-india" mode="currency" currency="INR" currencyDisplay="code"
                locale="en-IN" v-model.number="price" :min="0" class="p-inputnumber-sm w-full"/>
                <InputText id="manufacturer" placeholder="Manufacturer *" v-model.trim="manufacturer" class="p-inputtext-sm w-full"/>
                <DatePicker v-model="expiration_date" dateFormat="dd/mm/yy" placeholder="Expiration Date *" class="p-datepicker-sm w-full"
                showIcon fluid iconDisplay="input"/>
                <FloatLabel class="mt-4">
                    <Textarea v-model="description" autoResize rows="5" cols="54" class="w-full" />
                    <label>Description</label>
                </FloatLabel>
                <Select v-model="timings" :options="timings_choices" placeholder="Timing for taking the medicine *"/>
                <InputText id="customTiming" placeholder="Custom Timing Description *" v-model.trim="custom_timing_description" v-if="timings === 'Custom'"
                class="p-inputtext-sm w-full"/>
            </div>

            <div class="flex flex-col space-y-6">
                <div class="flex flex-col space-y-4">
                    <label class="font-semibold">Ingredients</label>
                    <div v-for="n in ingredients_count" :key="n" class="flex items-center space-x-4">
                        <AutoComplete :placeholder="`Ingredient ${n}`" v-model="selected_ingredients[n - 1]" optionLabel="name" dropdown
                        :suggestions="filtered_array" @complete="(event) => search(event, ingredients_data)" class="w-full" />
                    </div>
                    <Button label="Add Ingredient" @click.prevent="ingredients_count += 1" class="p-button-sm" />
                </div>
```

All the input fields are connected to a variable which is updated if the user types in information in that field

All the fields required to make a Medicine object and add stock for it.

A for loop that executes based on the number of times the button has been clicked, initially starting off with 1. With each execution, a new Input field is created.

The search function is used to filter the available options based on what the user inputs in the field

The button dynamically adds new input fields by incrementing the variable controlling the for loop.

Figure 36: *GUI Code for Add New Medicines page in the Administrator and Pharmacy profiles*

```
const search = (event, fullArray) => {
    setTimeout(() => {
        if (!event.query.trim().length) {
            filtered_array.value = [...fullArray]
        } else {
            filtered_array.value = fullArray.filter((element) => {
                return element.name.toLowerCase().startsWith(event.query.toLowerCase());
            });
        }
    }, 50);
}
```

An event is passed containing important details, including the string that the user has already typed in the field

If the user has not searched for anything yet (the length of the search query is 0), the full array is returned

If the user has inputted a string, the function returns the instances that match the inputted string

Figure 37: *Search function used by AutoComplete fields in Add New Medicines page*

```
const submit = () => {

    let medicineToSend = {}

    try {
        medicineToSend = {
            'name': name.value,
            'price': price.value,
            'description': description.value ? description.value : '',
            'manufacturer': manufacturer.value,
            'timings': timings.value,
            'side_effects' : selected_side_effects.value.map(sideEffect => ({ name: sideEffect.name ? sideEffect.name : sideEffect })),
            'allergens' : selected_allergens.value.map(allergy => ({ name: allergy.name ? allergy.name : allergy })),
            'timings': timings.value,
            'custom_timing_description': custom_timing_description.value ? custom_timing_description.value : ''
        }
    } catch (error) {
        warn("warn", "Required fields are not filled", "Please fill in all the required fields with appropriate values.")
        return;
    }

    if (selected_ingredients.value.length != 0) {
        medicineToSend.ingredients = selected_ingredients.value.map(ingredient => ({ name: ingredient.name ? ingredient.name : ingredient }));
    } else {
        medicineToSend.ingredients = []
    }
```

The function that is executed when the submit button is clicked.

The data that is sent to the backend, but is first checked

If the required fields are not filled, it generates an error because it is accessing values that does not exist yet as they have not been filled.

Goes through all the selected ingredients, checks if it is a value already existing in the database (ingredient.name) or if it is a new entry (ingredient). Gets the name of the ingredient accordingly.

As this is an optional field, the value is simply replaced with an empty array

Figure 38: *Function executed when the submit button is clicked (Part 1)*

```
if (selected_categories.value.length != 0){
    const categoriesArray = [];

    for (let i = 0; i < categories_count.value; i++) {
        categoriesArray[i] = {
            "name": selected_categories.value[i].name ? selected_categories.value[i].name : selected_categories.value[i],
            "usage_priority": (i + 1)
        };
    }

    medicineToSend.categories = categoriesArray
} else {
    warn('warn', 'Categories should be filled.', '');
    return;
}

let completed = filled(medicineToSend, ['ingredients', 'description', 'custom_timing_description'])

if (completed !== 'success') {
    warn('warn', completed + " required field is not filled", "Please fill in all the required fields with appropriate values.")
    return;
}

const requestToSend = {
    "medicine" : medicineToSend,
    "stock": stock.value,
    "expiration_date" : format(new Date(expiration_date.value), 'yyyy-MM-dd')
}

axios.post(`/${usertype}/addMedicines/`, requestToSend)
.then( (response) => {
    warn('success', 'Successfully added medicine.', '')
    // Reload the page
    setTimeout(() => {
        window.location.reload();
    }, 1000);
```

All categories are associated with a priority in which the medicine should be taken. The data is structured accordingly.

Since this is a required field, having empty fields will generate a notification, informing the user what to do

This function checks if the variable that is passed has all the required fields filled or not. The optional fields are mentioned in the second parameter, the array.

The function returns the field that is not filled, if all required fields are not fileld. The user is informed accordingly.

Sends the request to the backend after all checks have been made

Figure 38: *Function executed when the submit button is clicked (Part 2)*

Filled function

Pages that add new instances to the database undergo strict validation checks both on the front end and back end. Before submission, the front end utilizes the **filled** function from **helpers.js** to ensure all required fields are completed. Once submitted, the backend enforces rigorous validation to guarantee that only valid and useful instances are stored in the database.

```
export const filled = (data, optional_fields) => {

    for (const key in data) {
        if (!optional_fields.includes(key)) {
            const value = data[key];
            if (typeof value === 'string' && value.trim() === '') {
                return key; // Exit the loop early if an empty field is found
            } else if (typeof value === 'number' && value === 0) {
                return key; // Exit the loop early if a zero value is found
            } else if (Array.isArray(value) && value.length === 0) {
                return key;
            }
        }
    }

    return 'success';
}
```

This parameter requires an array of key values that do not need to be checked if they are filled

This parameter holds all the data that needs to be checked

Excludes the optional fields from the being checked

Depending on the data type, different validation checks are made to ensure they are not left blank

If a field is not filled, returns which field was not filled, so that the user can be informed appropriately.

returns the string 'success' if the data passes all validation checks

Figure 39: *Definition of filled function in helpers.js*

Add Stock for Existing Medicines

This page allows the user to add stock for medicines already existing in the database, preventing unnecessary repetition. The user just needs to select the medicine for which the stock has to be added, without inputting the details of the medicine.

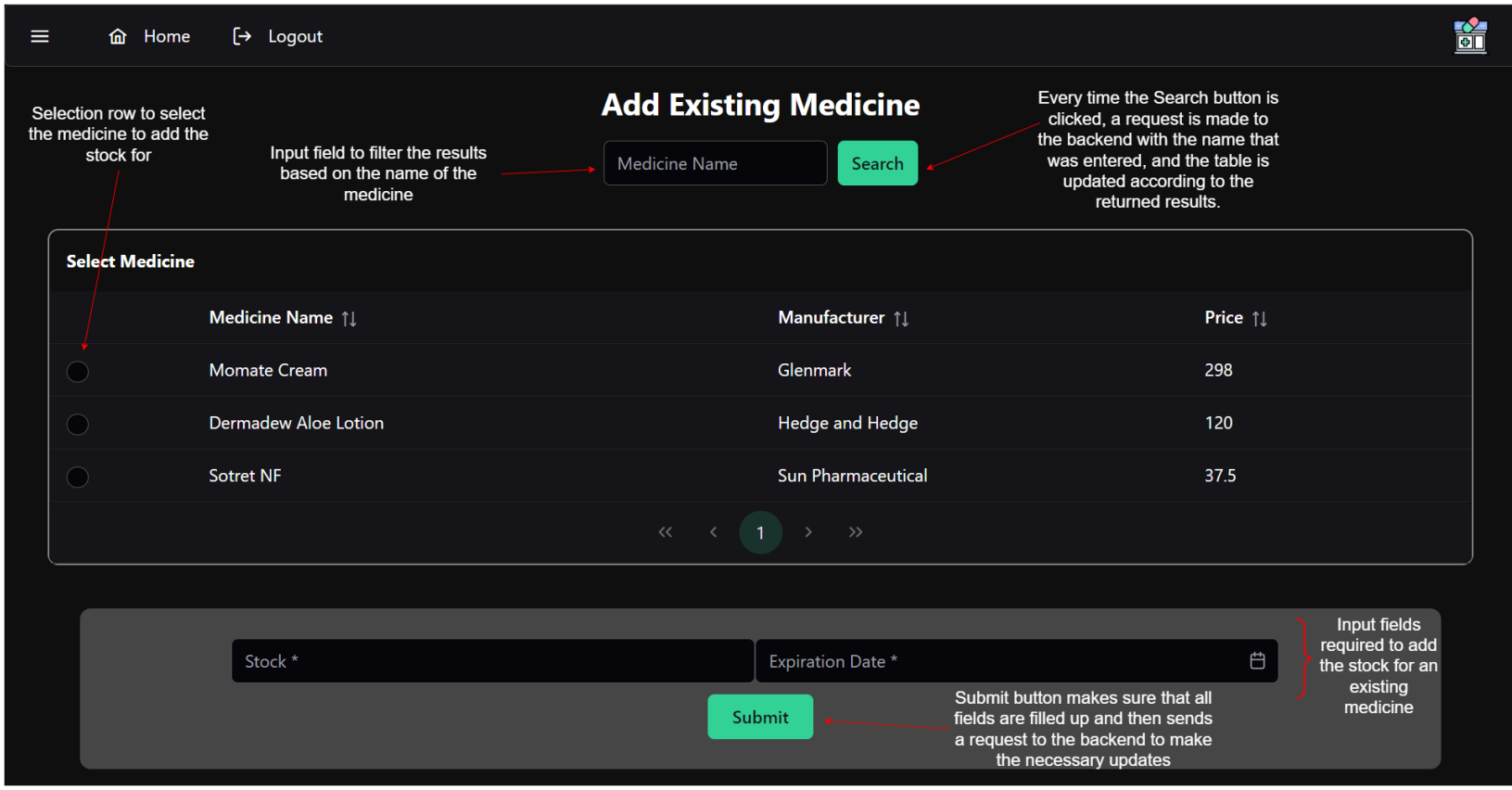

Figure 40: *GUI for Add Existing Medicines page in the Administrator and Pharmacy profiles*

Add Prescription

This page allows the doctor user to write a prescription for patients that have booked an appointment with that doctor. The page has various features to maximise the efficiency by the doctors while giving them fewer opportunities for making mistakes while prescribing medicines.

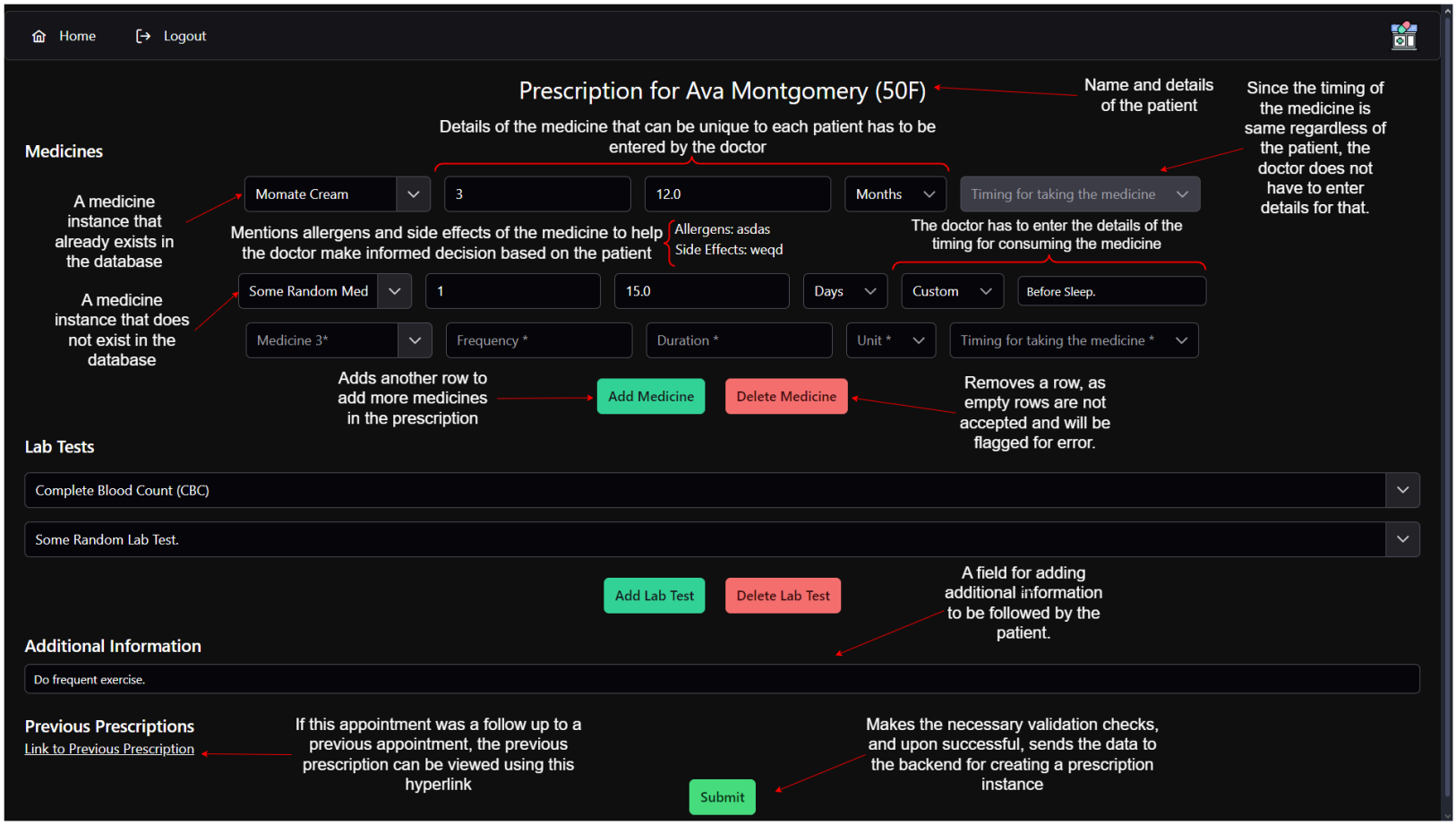

Figure 41: *Add Prescription Page in the doctor profile*

```
<div class="m-4 flex flex-column" v-if="is_registered === 'true'">
    <div class="flex flex-col space-y-4">
        <label class="text-xl font-semibold">Medicines</label>
        <div v-for="n in medicine_count" :key="n" class="flex flex-column items-center space-y-2">
            <div class="flex flex-row align-items-center justify-content-center space-x-4">
                <AutoComplete :placeholder="`Medicine ${n}*`" v-model="prescribed_medicines[n - 1]" optionLabel="name"
                 dropdown :suggestions="filtered_array" @complete="(event) => search(event, medicines_data )" class="w-full" />
                <InputNumber id="frequency" placeholder="Frequency *" inputId="withoutgrouping" :useGrouping="false"
                 v-model.number="frequency_medicines[n-1]" :min="0" class="p-inputnumber-sm w-full"/>
                <InputNumber id="duration" placeholder="Duration *" inputId="minmaxfraction" :minFractionDigits="1"
                 :maxFractionDigits="2" :useGrouping="false" v-model.number="duration_medicines[n-1]" :min="0"
                 class="p-inputnumber-sm w-full"/>
                <Select v-model="duration_unit[n - 1]" :options="duration_unit_choices" placeholder="Unit *"/>
                <Select v-model="timings[n - 1]" :options="timings_choices" placeholder="Timing for taking the medicine *"
                 v-if="isUnlistedMedicine(n - 1).value"/>
                <Select v-model="timings[n - 1]" :options="timings_choices" placeholder="Timing for taking the medicine"
                 v-else disabled/>
                <InputText v-model.trim="custom_timing_descriptions[n - 1]" id="customTiming"
                 placeholder="Custom Timing Description *" v-if="timings[n - 1] === 'Custom'" class="p-inputtext-sm w-full"/>
            </div>

            <div class="mt-2 flex flex-column align-items-center justify-content-center space-x-4"
             v-if="!isUnlistedMedicine(n - 1).value">
                <div class="flex flex-row space-x-1">
                    <label>Allergens: </label>
                    <label>{{ prescribed_medicines[n - 1].allergens.map(allergen => allergen.name).join(', ') }}</label>
                </div>

                <div class="flex flex-row space-x-1">
                    <label>Side Effects: </label>
                    <label>{{ prescribed_medicines[n - 1].side_effects.map(side_effect => side_effect.name).join(', ') }}</label>
                </div>
            </div>
```

This text box is enabled when the doctor prescribes a medicine that is not listed in the database. The doctor has to mention the timing for taking the medicine then

Disabled if the medicine is selected from the database

This Input Text is only triggerd if the timings value for that medicine instance is "Custom"

Checks if the medicine entered is listed in the database or not.

If the medicine is listed in the database, the allergens and side effects of the medicine will be listed below the medicine, so that the patient can then be informed accordingly.

Figure 42: *Code (partial) for making the GUI*

```
// Computed function to check if a specific medicine is unlisted
const isUnlistedMedicine = (index) => {
    return computed(() => {
        const selected_medicine = prescribed_medicines.value[index];
        console.log(typeof selected_medicine)
        return (typeof selected_medicine === 'string')
    });
};

// Watch for changes in prescribedMedicines and update indexArrayMedicine
watch(prescribed_medicines, (new_medicines) => {
    unlisted_index_array_medicine.value = new_medicines
        .map((medicine, index) => ({
            isUnlisted: medicine && !medicines_data.value.some(med => med.name === medicine.name),
            index,
        }))
        .filter(med => med.isUnlisted)
        .map(med => med.index);
}, { deep: true });

watch(prescribed_lab_tests, (new_lab_tests) => {
    unlisted_index_array_lab_tests.value = new_lab_tests
        .map((lab_test, index) => ({
            isUnlisted: lab_test && !lab_tests_data.value.some(lab => lab.name === lab_test.name),
            index,
        }))
        .filter(med => med.isUnlisted)
        .map(med => med.index);
}, { deep: true });
```

Computed property is reactive i.e. whenever there's a change in a property that is associated with it, this function is triggered. In this case, whenever a medicine is entered, it checks if it is a medicine taken from the database, or if it is an unlisted medicine

watch property looks out for changes to the variable (prescribed_medicines).

Goes through the entire prescribed medicines list and checks if a medicine is listed or not

Filters out the medicine that are listed

Extracts only the index, removing isUnlisted variable from the data

This entire process gives an array of indexes in the list prescribed_ medicines which contains unlisted medicines. This is useful when making the request to the backend.

The same logic is applied but this filter the lab tests.

Figure 43: *Watch and computed properties used to enable reactive feature used in the Add Prescription page of the doctor profile*

Logout

The Logout page deletes the tokens from the user side, so that the user has to re login again to access the desired profile.

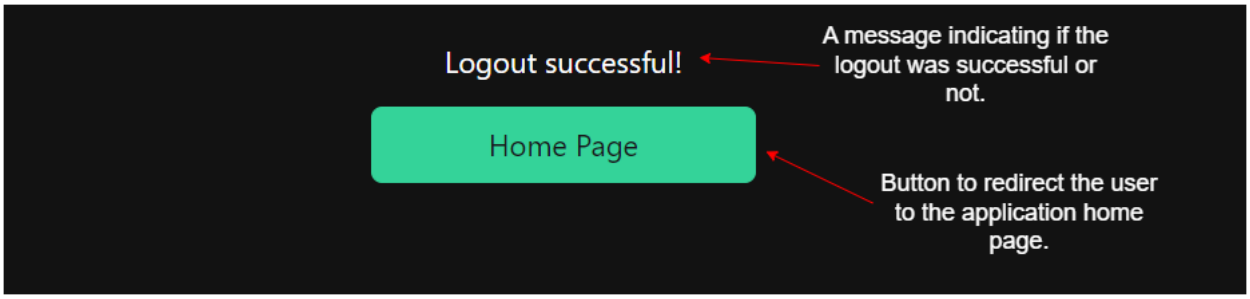

Figure 44: *Logout GUI*

```
<script setup>
    import axios from '../axios';
    import { useStore } from 'vuex';
    import { ref } from 'vue';

    const message = ref('');

    const store = useStore()
    store.dispatch('initializeStore')

    if (store.state.is_registered === "true") {
        axios.post('/logout/',
        {
            "refresh_token": localStorage.getItem('refresh_token')
        }).then( (response) => {
            store.dispatch('logout')
            message.value = "Logout successful!"
        }).catch( (error) => {
            message.value = "Logout unsuccessful!"
        })
    } else {
        message.value = "Logout unsuccesful! Log in first to log out."
    }
</script>

<template>
    <div class="flex flex-column align-items-center justify-content-center mt-4">
        <h1>{{ message }}</h1>
        <Button label="small" class="routerlink" @click="$router.push({ name: 'Home' })">Home Page</Button>
    </div>
</template>
```

First, checks if the user is logged in or not.

Sends the refresh token to the backend to be blacklisted, so that the refresh token cannot be used again

Based on the response, shows the message on the screen.

Figure 45: *Code for logging out in the front end*

References

The code was written by referring to numerous secondary resources enlisted below:

Serial	Hyperlink
1	https://primevue.org
2	https://vuejs.org
3	https://vuex.vuejs.org/guide/
4	https://router.vuejs.org

Additional Logic in the Backend

Update Stock

Update Stock is a view in the pharmacy application that updates the stock for medicines when they are dispensed for completing prescriptions. The following code demonstrates the logic for the code:

```python
class UpdateStock(views.APIView):
    '''
        On a successful post request, updates the stock
    '''
    authentication_classes = (authentication.CustomPharmacyAuthentication, )
    permission_classes = (permissions.IsAuthenticated, )

    def post(self, request):

        try:
            prescription = Prescription.objects.get(id=request.data['prescription_id'])
            update_stock = request.data.get('update_stock', [])  # Extract update_stock list safely

            resp = []

            medicines_cost = prescription.medicines_cost

            for row in update_stock:
                try:
                    medicine = Medicines.objects.get(id=row['id'])
                    quantity = row['quantity']
                    medicine_stocks = MedicineStock.objects.filter(medicine=medicine).order_by('expiration_date')

                    while quantity > 0 and medicine_stocks.exists():
                        stock_entry = medicine_stocks.first()  # Get the earliest expiring stock
                        stock = stock_entry.stock

                        if stock <= quantity:
                            quantity -= stock
                            medicines_cost += medicine.price * stock
                            stock_entry.delete()  # Delete depleted stock
                        else:
                            stock_entry.stock -= quantity
                            medicines_cost += medicine.price * quantity
                            stock_entry.save()
                            quantity = 0  # Fully reduced, break loop
```

Only pharmacy users are allowed to access this endpoint.

This variable holds the id of the medicine and the number of medicines dispensed for that medicine id

Retrieves the cost of medicines dispensed prior to this point.

Updates the stock for each medicine individually

Gets the medicine that needs to be dispensed.

Quantity to be dispensedd

The loop runs until the required quantity are dispensed or there is no medicine stock left to dispense.

Dispenses the earliest expiring stock first.

Stock available for the earliest expiring stock.

Calculates the cost of dispensed medicines

If the available stock for a medicine is less than or equal to the required quantity, the dispensed quantity is adjusted to match the available stock, and that stock instance is removed upon depletion.

If the available stock for a medicine is greater than the required quantity, the stock for that medicine is reduced by the amount dispensed, and the loop breaks as all medicines are dispensed

Figure 46: *Update Stock view of the pharmacy application in the backend (Part 1)*

```python
                    if quantity == 0:
                        resp.append({"id": row['id'], "medicine_name": medicine.name, "status": "updated"})
                    else:
                        resp.append({"id": row['id'], "medicine_name": medicine.name, "status": "unavailable", "quantity": quantity})

                except Medicines.DoesNotExist:
                    resp.append({"id": row['id'], "medicine_name": f"ID {row['id']}", "status": "Medicine Not Found."})

                except Exception as e:
                    resp.append({"id": row['id'], "medicine_name": f"ID {row['id']}", "status": f"Error: {str(e)}"})

            prescription.medicines_cost = medicines_cost
            prescription.save()

            return response.Response(resp, status=status.HTTP_200_OK)

        except Prescription.DoesNotExist:
            return response.Response({"error": "Prescription not found."}, status=status.HTTP_404_NOT_FOUND)

        except Exception as e:
            return response.Response({"error": str(e)}, status=status.HTTP_400_BAD_REQUEST)
```

If the required stock for the medicine is dispensed successfully, the status is set to "updated"

Otherwise, the status is sent as "unavailable"

In case of an error such as the Medicine could not be found from the id, the execution does not stop but the error is sent in the response to notify the pharmacy

The cost of medicines is updated for the prescription and saved

Sends the response with status of all medicines, indicating success

If the prescription was not found or any other error was encountered, the error is sent back to the front end

Figure 47: *Update Stock view of the pharmacy application in the backend (Part 2)*

Calculate the Cost of Unpaid Appointments

The view calculates the cost of an appointment for a patient based on the consultation fees of the doctor, medicines and the lab tests taken. It is displayed at the front desk of the application, where the front desk has the option to mark a prescription as paid upon collection of payment.

```python
class GetUnpaidAppointments(views.APIView):
    '''
        On a successful post request, returns unpaid appointments for a doctor
    '''
    authentication_classes = (authentication.CustomFrontDeskAuthentication,)
    permission_classes = (permissions.IsAuthenticated,)

    def getTotalCost(self, prescription):
        total_cost = 0

        # Doctor Cost
        doctor_cost = prescription.appointment.doctor.consultation_fee
        total_cost += doctor_cost

        # Lab Tests costs
        lab_tests_cost = sum(
            lab_test.lab_test.test_cost for lab_test in prescription.lab_tests.all() if lab_test.status != 'Prescribed'
        )
        total_cost += lab_tests_cost

        # Medicines Cost
        medicines_cost = prescription.medicines_cost
        total_cost += medicines_cost

        return {
            'doctor_cost': doctor_cost,
            'lab_tests_cost': lab_tests_cost,
            'medicines_cost': medicines_cost,
            'total_cost': total_cost
        }
```

First adds the consultation fees of the doctor.

If the status of the lab test is not prescribed, indicating that the lab test has been conducted, the fees are added accordingly.

Adds the medicine cost that was calculated when the medicines were dispensed, in Update Stock view.

Sends a breakdown of the cost back to the front end.

Figure 48: *Code to calculate the cost of appointment for Get Unpaid Appointments view in the Front Desk application.*

Logout

When logging out from the front end, the refresh and access tokens are deleted from local storage. However, to ensure maximum security, the refresh token is blacklisted so that it cannot be used to generate new access tokens, preventing unauthorized reauthentication.

```
class LogoutView(APIView):
    permission_classes = (permissions.IsAuthenticated, )

    def post(self, request):
        try:
            refresh_token = request.data["refresh_token"]
            token = RefreshToken(refresh_token)
            token.blacklist()

            return Response('Logout successful', status=status.HTTP_205_RESET_CONTENT)
        except Exception as e:
            return Response({"error": str(e)}, status=status.HTTP_400_BAD_REQUEST)
```

Gets refresh token from request

Gets the instance from the refresh token string

Blacklists the refresh token from further use

Figure 49: *Logout view, defined in view.py of the root folder of the backend application*

References

The code was written by referring to numerous secondary resources enlisted below:

Serial	Hyperlink
1	https://docs.djangoproject.com/en/5.1/
2	https://www.django-rest-framework.org/api-guide/serializers/
3	https://www.django-rest-framework.org/api-guide/views/
4	https://www.django-rest-framework.org/api-guide/authentication/
5	https://django-rest-framework-simplejwt.readthedocs.io/en/latest/