DALHOUSIE UNIVERSITY

CSCI 5308
Advanced Topics in Software Development

PROJECT

Dalhousie Multifaith Services (DMS)

Group 10

Aditya Deepak Mahale	B00867619
Anas Malvat	B00911636
Harshit Lakhani	B00887087
Jayasree Kulothungan	B00894354
Kalpit Machhi	B00911364

CLIENT TEAM

Group 22

INSTRUCTOR

Tushar Sharma

TA

Hari Ramesh

Index

1. Sign-up Feature

1.1 Feature Overview

The sign-up feature is added for the DMS Users to be able to register themselves to DMS. After registration, the user will be able to login into the system by entering their registered email and password. Two kinds of users can register into our system - Students of Dalhousie University and chaplains working for DMS.

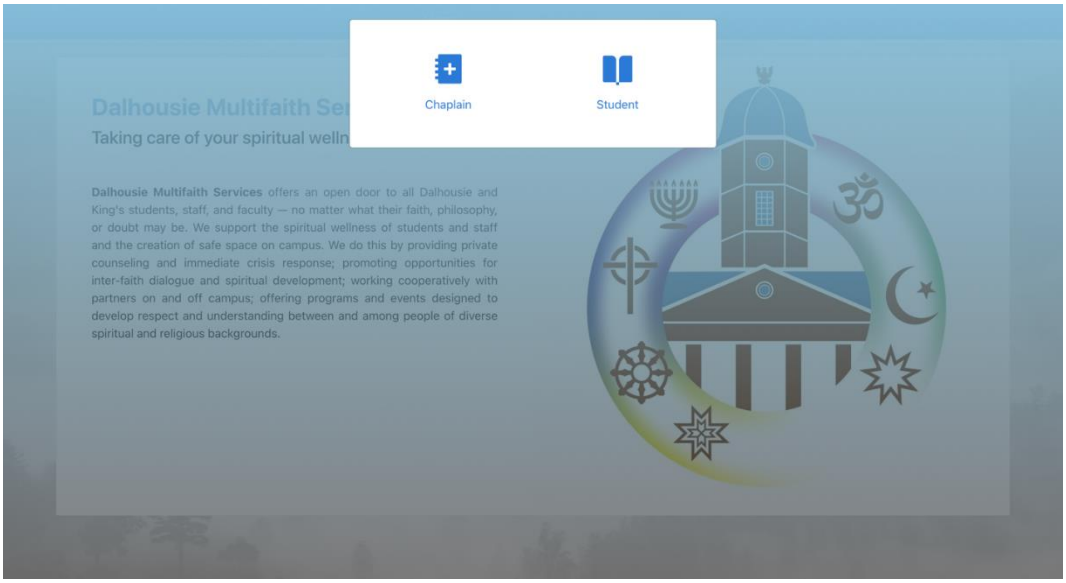

Figure 1.1.1 User Registration

The chaplain will enter his/her information such as First Name, Last Name, Religion, Description, Phone Number, Email, Password.

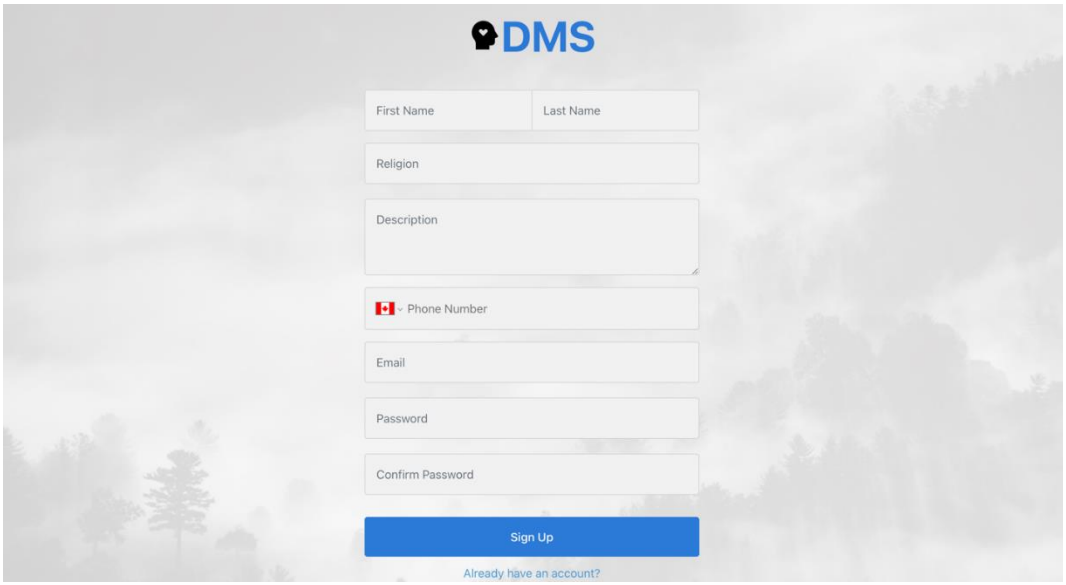

Figure 1.1.2 Chaplain Registration

The student will enter his/her information such as First Name, Last Name, Email, Password.

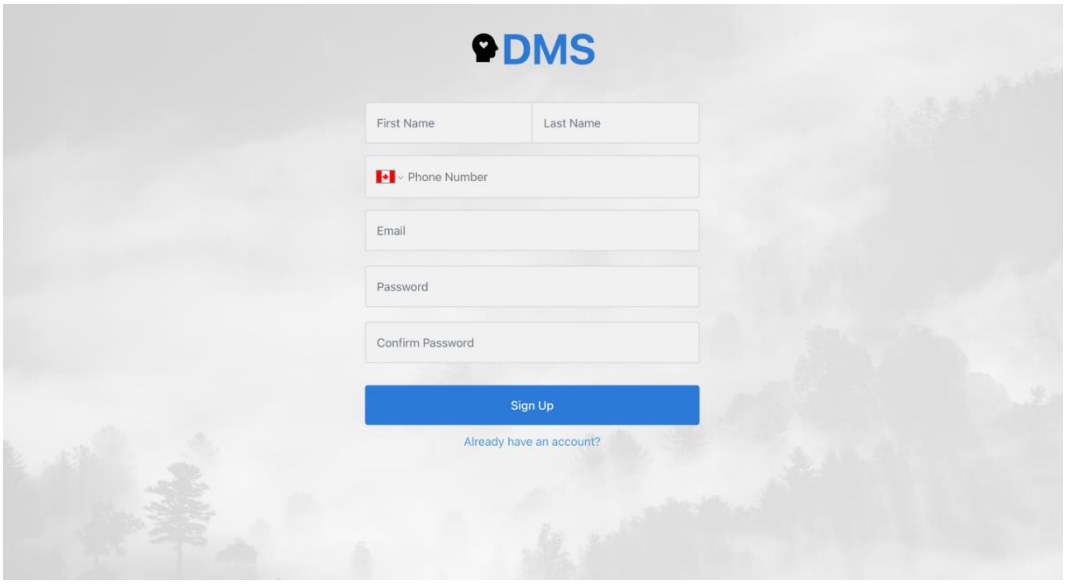

Figure 1.1.3 Student Registration

1.2 Back-end Data Management

Figure 1.2.1 and **Figure 1.2.2** show the models created to store the details of students and chaplains, respectively. Both models have a one-to-one relationship with the AUTH_USER_MODEL. The 'user' model has attributes like the first name, last name, mobile number, and email, typical for both models. The 'chaplain' model additionally has unique characteristics like phone, religion, and description.

```python
class DalUser(models.Model):
    user = models.OneToOneField(
        settings.AUTH_USER_MODEL, on_delete=models.CASCADE
    )
    phone = models.CharField(max_length=255)

    def __str__(self) -> str:
        return f"{self.user.first_name} {self.user.last_name}"

    class Meta:
        ordering = ["user__first_name", "user__last_name"]
```

Figure 1.2.1 Student - models.py

```python
class Chaplain(models.Model):
    user = models.OneToOneField(
        settings.AUTH_USER_MODEL, on_delete=models.CASCADE
    )
    phone = models.CharField(max_length=255)
    religion = models.CharField(max_length=255)
    description = models.TextField()

    def __str__(self) -> str:
        return f"{self.user.first_name} {self.user.last_name}"

    class Meta:
        ordering = ["user__first_name", "user__last_name"]
```

Figure 1.2.2 Chaplain - models.py

To show the details stored in these models, two classes are created in the views.py file: DalUserDetail and ChaplainDetail. They both have two methods:

(1) get()
(2) put()

The get() method is used to get the profile details from the database and view it on the website. At the same time, the put() method is used to update the elements stored in the database using the data provided in the front end.

```python
class DalUserDetail(APIView):
    def get(self, request, id):
        dal_user = get_object_or_404(DalUser, user_id=id)
        serializer = DalUserSerializer(dal_user)
        return Response(serializer.data)

    def put(self, request, id):
        dal_user = get_object_or_404(DalUser, user_id=id)
        serializer = DalUserSerializer(dal_user, data=request.data)
        serializer.is_valid(raise_exception=True)
        serializer.save()
        return Response(serializer.data)
```

Figure 1.2.3 Student - views.py

```python
class ChaplainDetail(APIView):
    def get(self, request, id):
        dal_user = get_object_or_404(Chaplain, user_id=id)
        serializer = ChaplainSerializer(dal_user)
        return Response(serializer.data)

    def put(self, request, id):
        dal_user = get_object_or_404(Chaplain, user_id=id)
        serializer = ChaplainSerializer(dal_user, data=request.data)
        serializer.is_valid(raise_exception=True)
        serializer.save()
        return Response(serializer.data)
```

Figure 1.2.4 Chaplain - views.py

2. Login Feature

2.1 Feature Overview

The login feature is added for the DMS Users to be able to login-in to DMS. This feature is developed using React-js. Aside from login, forgot password with O.T.P verification is also provided. The user must register himself to be able to login into the system. Two kinds of users can login into our system; Students of Dalhousie University and chaplains.

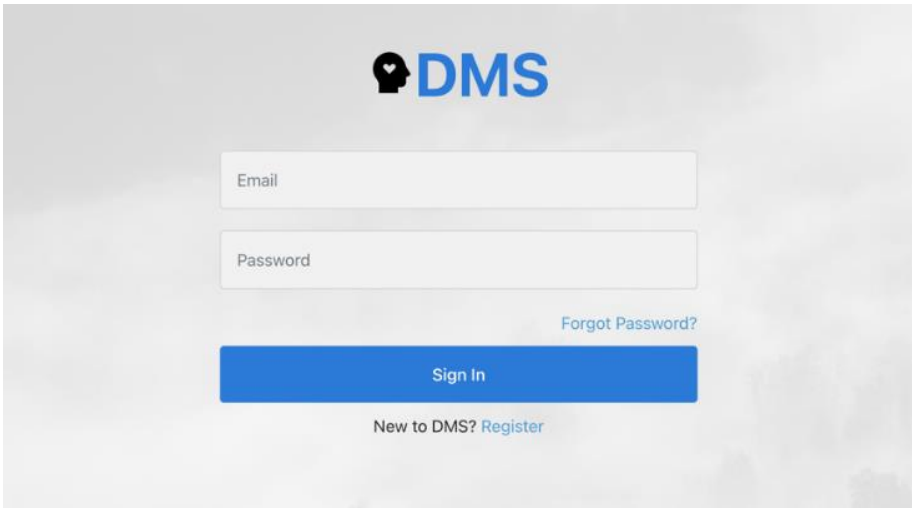

Figure 2.1.1 Login Page

If the user enters a valid registered email and password he/she will be logged in to the system depending on whether it's a chaplain login or student login

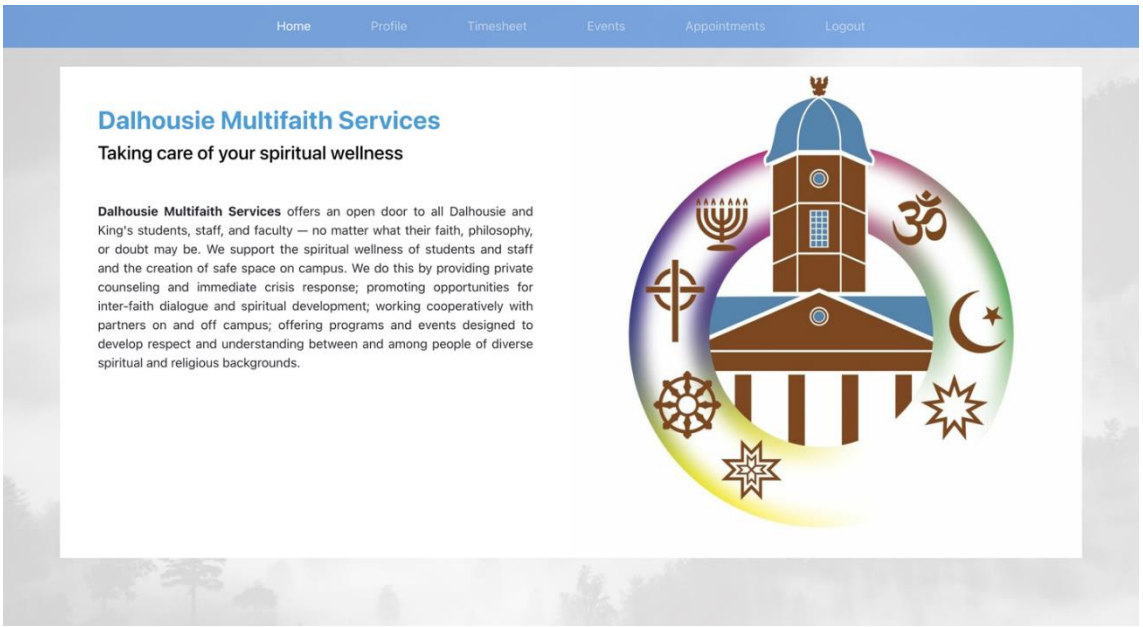

Figure 2.1.2 Chaplain Home Page

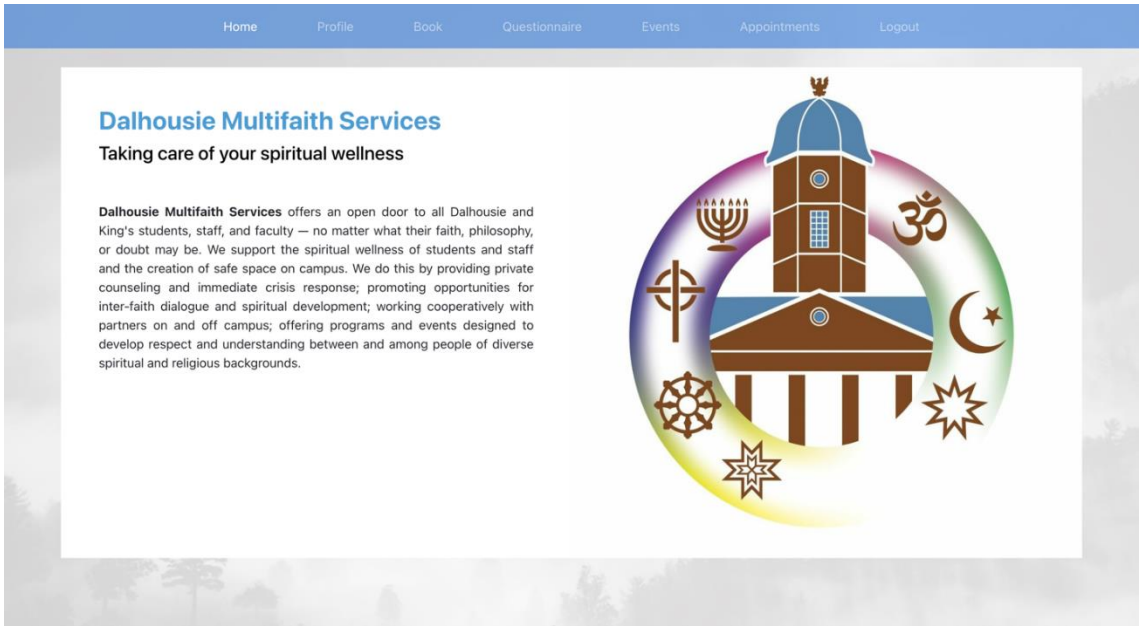

Figure 2.1.2 Student Home Page

2.2 Back-end Data Management

The data is stored on the back end and developed using Django Rest Framework. The table DalUser is defined. The authorization is performed to verify the user before logging in.

```
class DalUser(models.Model):
    user = models.OneToOneField(
        settings.AUTH_USER_MODEL, on_delete=models.CASCADE
    )
    phone = models.CharField(max_length=255)

    def __str__(self) -> str:
        return f"{self.user.first_name} {self.user.last_name}"

    class Meta:
        ordering = ["user__first_name", "user__last_name"]
```

Figure 2.2.1 Student – models.py

Then, the methods needed to perform the sending and retrieving data operations are defined in views. For login, we require two operations – Get and Post.

```
class DalUserDetail(APIView):
    def get(self, request, id):
        dal_user = get_object_or_404(DalUser, user_id=id)
        serializer = DalUserSerializer(dal_user)
        return Response(serializer.data)

    def put(self, request, id):
        dal_user = get_object_or_404(DalUser, user_id=id)
        serializer = DalUserSerializer(dal_user, data=request.data)
        serializer.is_valid(raise_exception=True)
        serializer.save()
        return Response(serializer.data)
```

Figure 2.2.2 Student Details – views.py

To verify if all the fields are defined and to validate the data received from the front-end, we use Serializers.

```python
class DalUserSerializer(serializers.ModelSerializer):
    user_id = serializers.IntegerField()
    user = UserCreateSerializer(read_only=True)

    class Meta:
        model = DalUser
        fields = ["id", "user_id", "phone", "user"]
```

Figure 2.2.3 Student – serializers.py

2.3 Integration

We integrated the backend and frontend by using AXIOS interceptors. After setting up the HTTP services to handle GET, POST, PUT and DELETE functions, the methods required for the login are specified in authService.js Service. Here, the endpoints are specified precisely. It is necessary to authorize the user that is logged in and to generate a token key in the local storage to know the currently logged-in user. JWT is used for secure data transfer.

```javascript
async function login(email, password) {
  const { data } = await http.post(apiEndpoint, { email, password });
  localStorage.setItem(tokenKey, data.access);
}

function logout() {
  localStorage.removeItem(tokenKey);
}

function getJwt() {
  return localStorage.getItem(tokenKey);
}

function loginWithJwt(jwt) {
  localStorage.setItem(tokenKey, jwt);
}
```

Figure 2.3.1 Specifying get and post methods

```
async function getCurrentUser() {
  try {
    const jwt = getJwt();
    if (jwt === null) return null;
    jwtDecode(jwt);

    http.setJwt(jwt);
    const { data } = await http.get(apiUserEndpoint);
    if (data.is_staff) {
      const { data: chaplain } = await http.get(
        `${apiChaplainEndpoint}${data.id}`
      );
      return chaplain;
    } else {
      const { data: daluser } = await http.get(
        `${apiDalUserEndpoint}${data.id}`
      );
      return daluser;
    }
  } catch (ex) {
    return null;
  }
}
```

Figure 2.3.2 Getting user details

The get function is called first in the useEffect() so that the user data loads before any other method.

```
useEffect(() => {
  const getData = async () => {
    const currentUser = await auth.getCurrentUser();
    setUser(currentUser);

    if (currentUser) {
      let user_id, chaplain_id;
      if (currentUser.user.is_staff) {
        user_id = 0;
        chaplain_id = currentUser.id;
      } else {
        user_id = currentUser.id;
        chaplain_id = 0;
      }
```

Figure 2.3.3 Getting the data from backend

2.4 Testing

The testing tool used is pytest.

2.4.1 Unit Testing

We have used mocking to perform unit testing. The response for both Get and Post methods are checked to see whether the connection is made, and the endpoints are working as they should.

```python
# DALUSER - mocked the get method from the DalUserDetails
def test_dal_user_detail_get(self):
    fake_response = Response()
    fake_response.status_code = 200
    mock_get = mock.Mock(
        name="DalUserDetail-get", return_value=fake_response
    )
    self.dal_user_detail.get = mock_get

    assert self.dal_user_detail.get().status_code == status.HTTP_200_OK

# DALUSER - mocked the put method from the DalUserDetails
def test_dal_user_detail_put(self):
    fake_response = Response()
    fake_response.status_code = status.HTTP_202_ACCEPTED
    mock_put = mock.Mock(
        name="DalUserDetail-put", return_value=fake_response
    )
    self.dal_user_detail.put = mock_put

    assert (
        self.dal_user_detail.put().status_code == status.HTTP_202_ACCEPTED
    )
```

Figure 2.4.1.1 Login - Unit Testing

2.4.2 Integration Testing

In integration testing, we are passing values to the methods and check if they are returning appropriate messages.

```python
# Daluser-test adding daluser with correct data
def test_dal_user_detail_get(self):
    # user added
    user_response = requests.post(
        self.url,
        {
            "email": "mvdfjne@lakha.ca",
            "password": "group@123",
            "first_name": "harshit",
            "last_name": "lakhani",
            "is_staff": False,
        },
    )
    user_id = user_response.json()["id"]
    data = {"user_id": user_id, "phone": "9024538293", "user": "harshit"}
    # daluser added
    daluser_response = requests.post(self.dal_user_url, data)
    response = requests.get(self.dal_user_url + str(user_id))
    assert response.json()["id"] == daluser_response.json()["id"]
```

Figure 2.4.2.1 Login - Integration Testing

3. Password Modification Feature

3.1 Feature Overview

Our system has 2 different types of users – Chaplin and Students. A user can register in DMS using an email address that has dal.ca in the end. If for some reason the user is not able to remember the password correctly, they have a one-time password-based recovery option. One needs to enter their dal email and they will get an email with a 6-digit numeric code on their registered email. In the next step, they can enter that OTP and set a new password for their account.

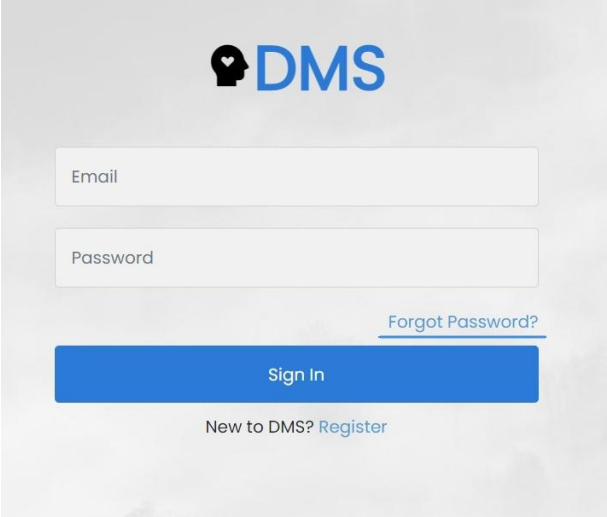

Figure 3.1.1 Forgot password option for Recovery

If a user clicks on the forget password link, he/she will be directed to the Password recovery page.

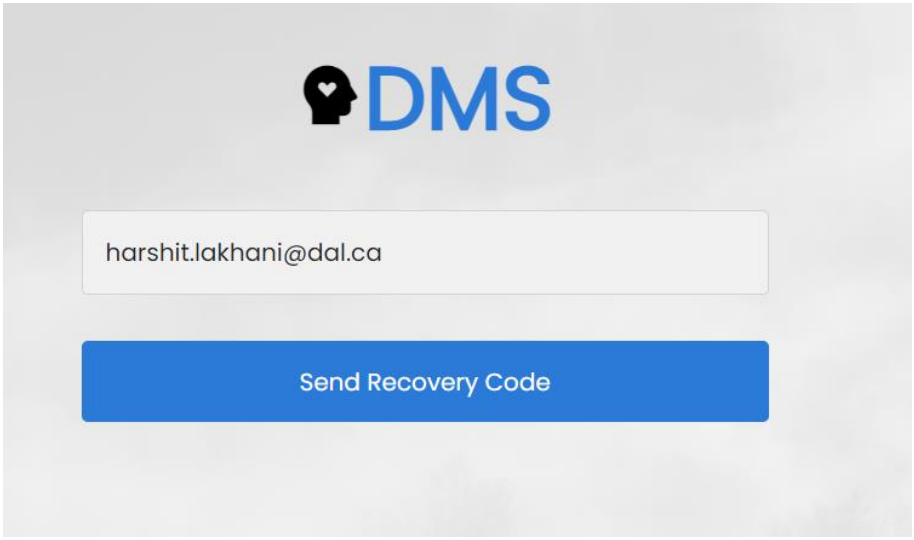

Figure 3.1.2 User enters DMS registered email

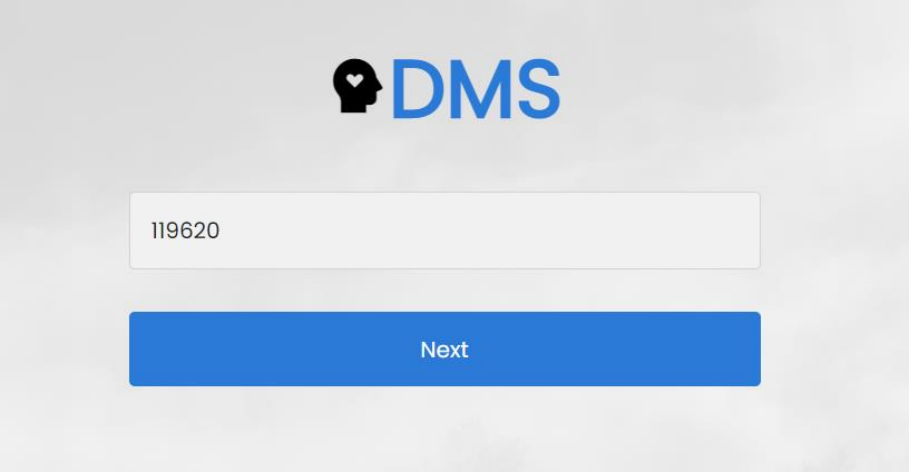

Figure 3.1.3 Enter One Time Password for verification

An email is received to the registered user, it has a 6-digit one-time password for authenticating the identity of the user.

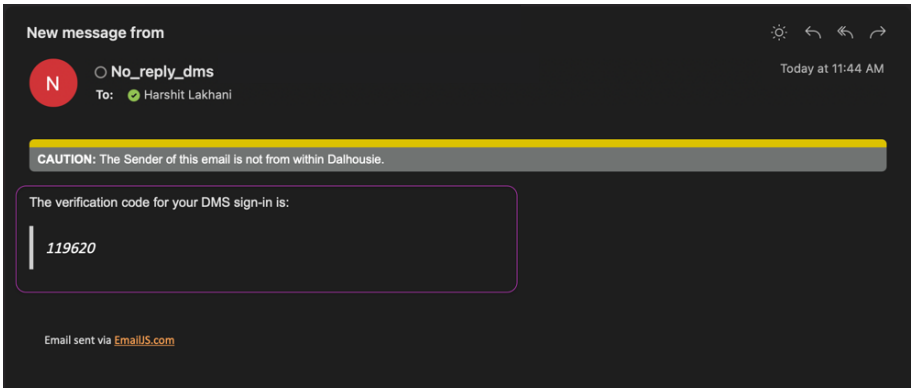

Figure 3.1.4 OTP Verification Mail

One has to set a new password and a confirmed password should match the new password to be accepted and updated.

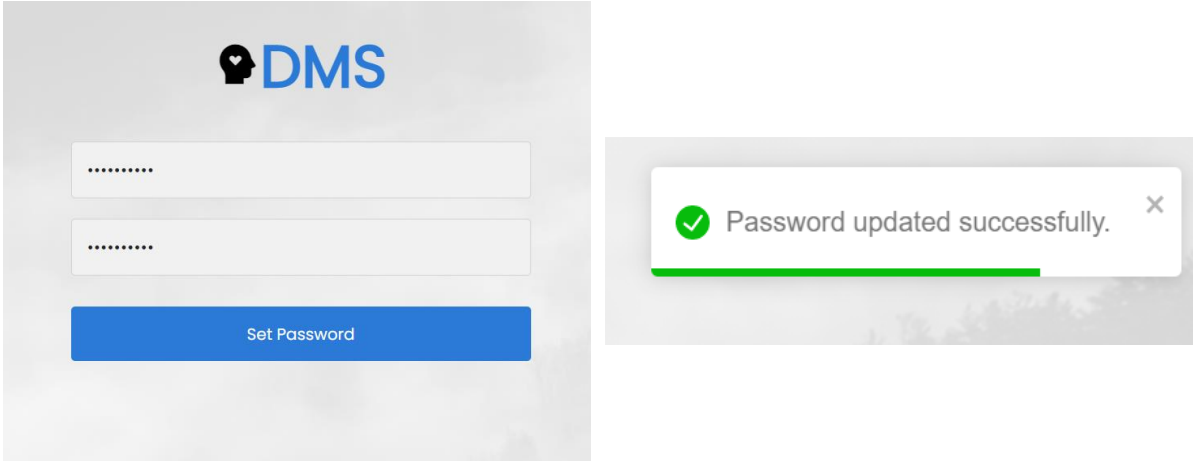

Figure 3.1.5 Password Updated

Once, the new password is updated, they are redirected to the login page so that they can easily login into their account.

4. Appointment Booking Feature

4.1 Feature Overview

The appointment booking feature is a complete booking flow that starts from selecting chaplains, booking slots, and then the chaplain confirming or rejecting the booked slots.
The list of chaplains can be viewed by clicking the Chaplains option in the header.

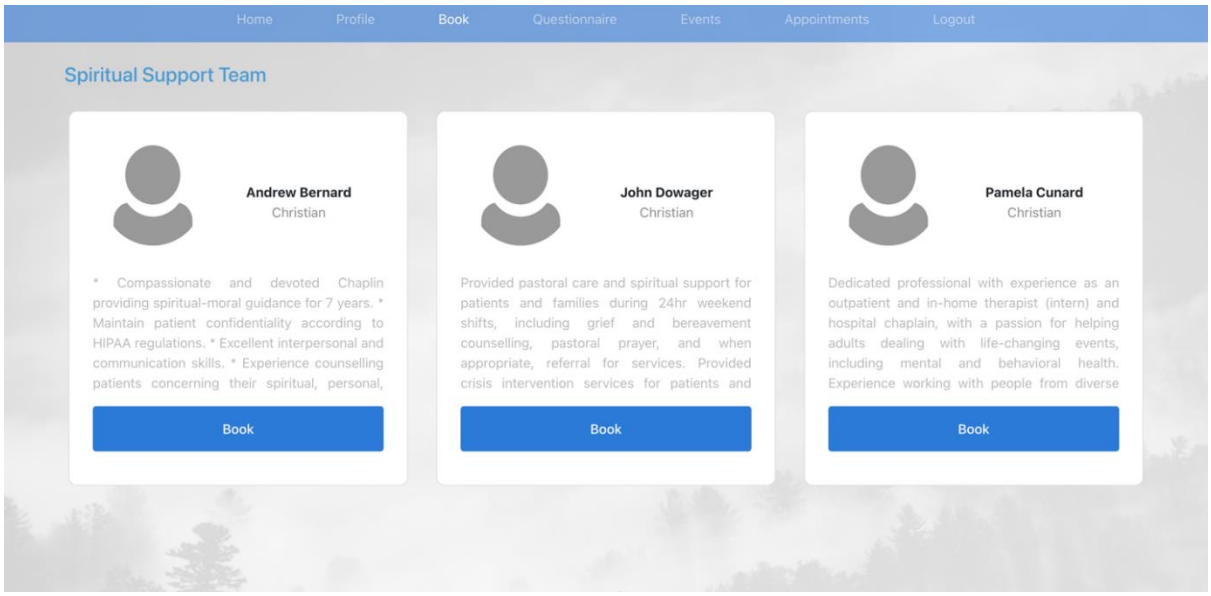

Figure 4.1.1 Chaplain List

Next, on clicking book, the time slots of each chaplain will be available.

Figure 4.1.2 Chaplain Slots

On selecting the slot, the book option will be enabled.

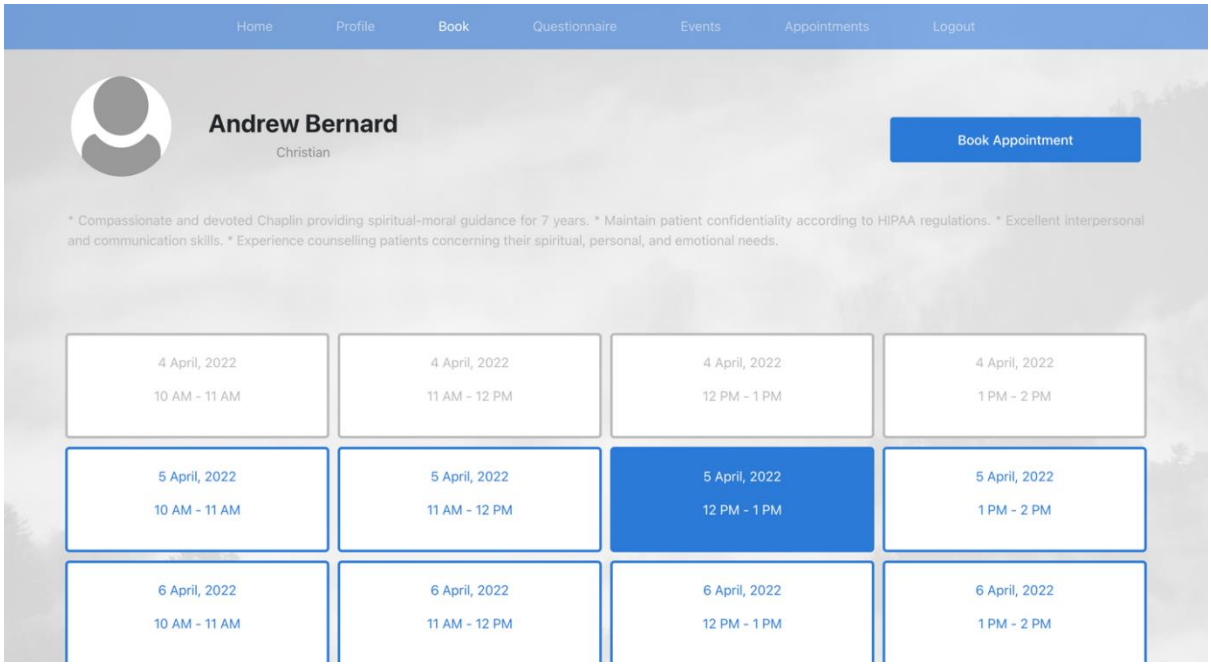

Figure 4.1.3 Appointment Slot Selection

On clicking on Book Appointment, the confirmation modal opens.

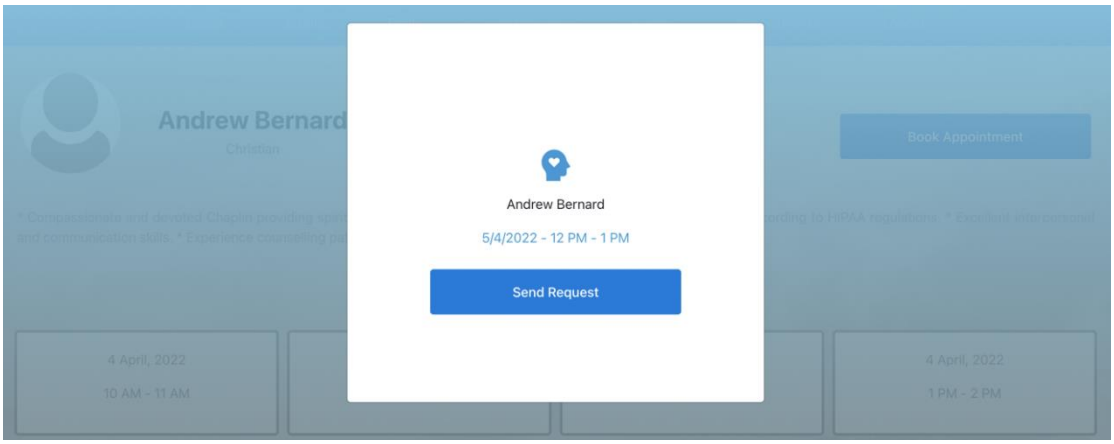

Figure 4.1.4 Appointment Confirmation Modal

A modal showing that an email for booking confirmation is sent will appear.

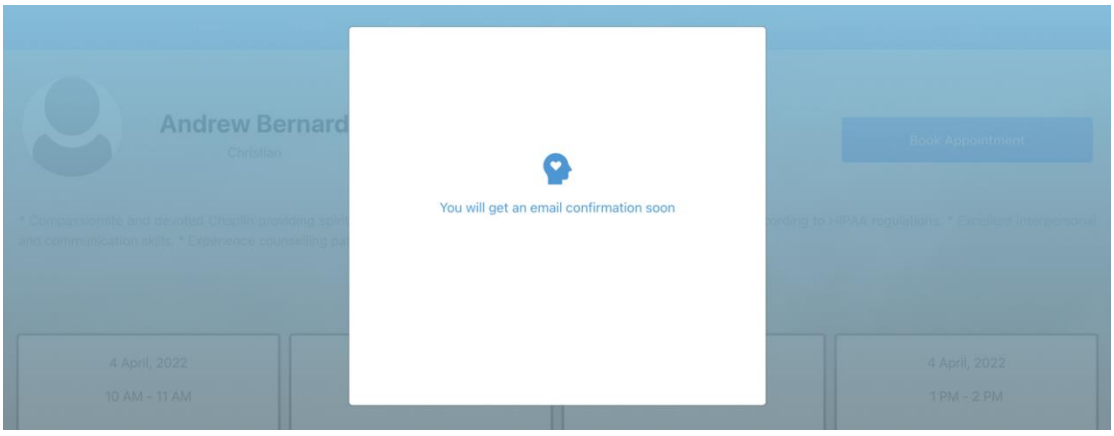

Figure 4.1.5 Email Confirmation Modal

The email is sent using Email.js.

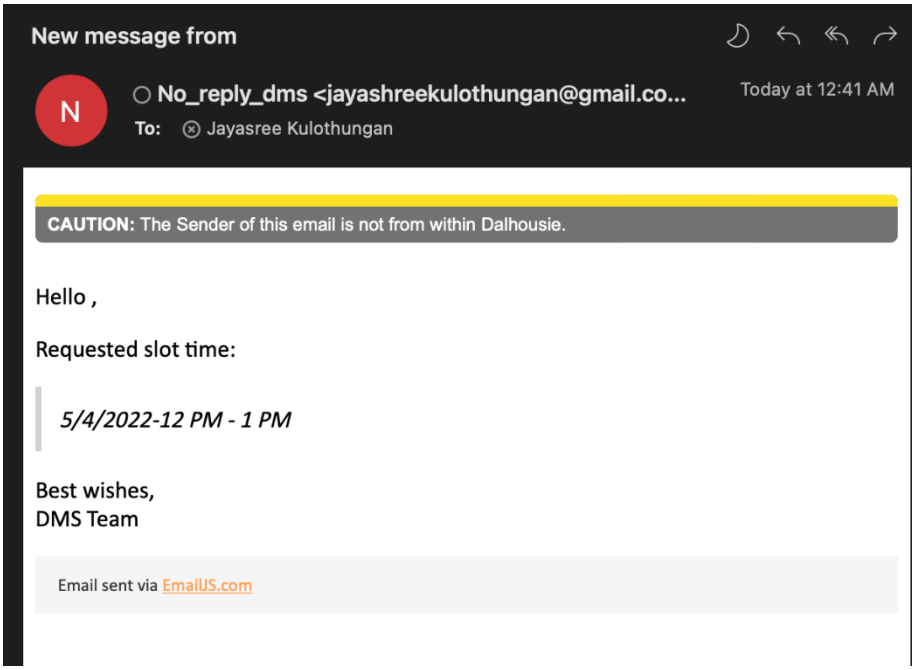

Figure 4.1.6 Email Sent to the Student

Once the slot is confirmed, it will reflect on the history of appointments in user and chaplain views. There are three different appointment statuses – pending, confirmed, and rejected.

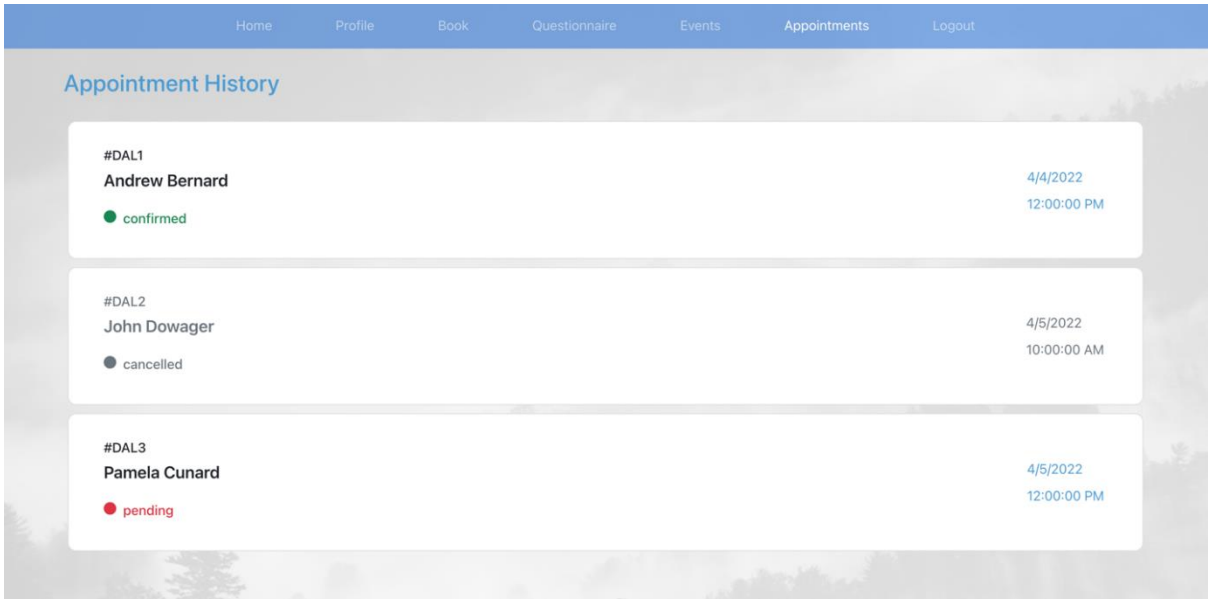

Figure 4.1.7 Appointment History Status Dashboard

The chaplain can accept or reject an appointment.

Accept:

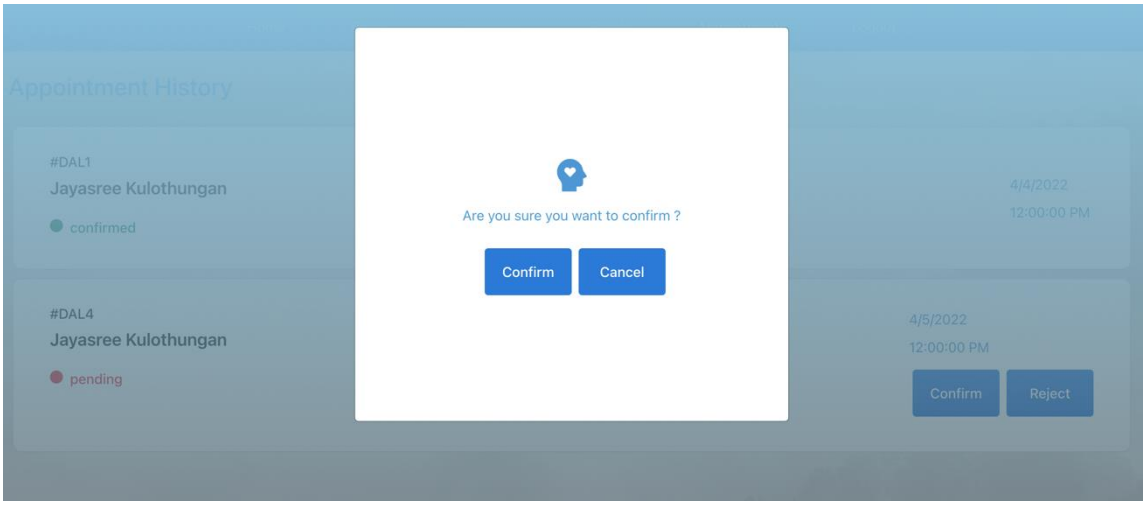

Figure 4.1.8 Confirmation Modal on Chaplain View

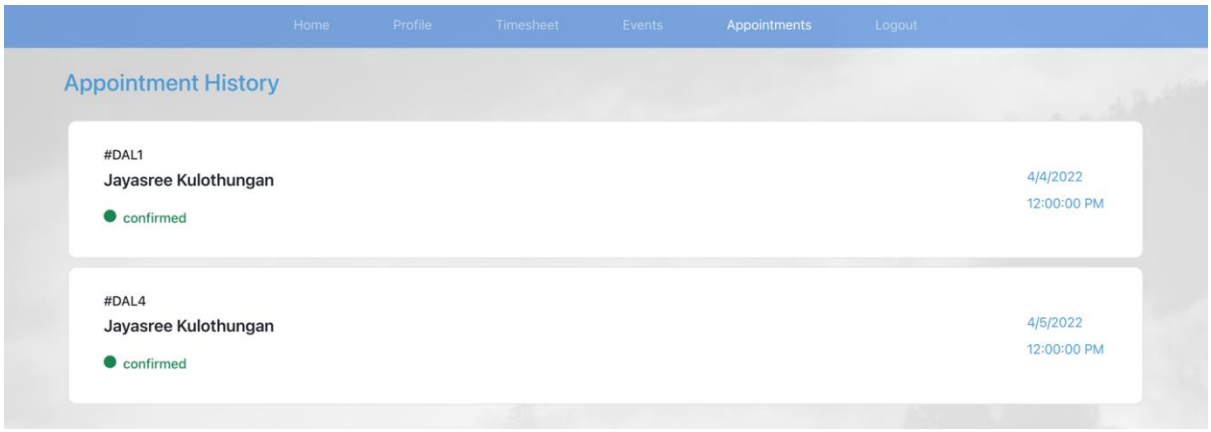

Figure 4.1.9 Appointment history on chaplain view – Confirmed

Reject:

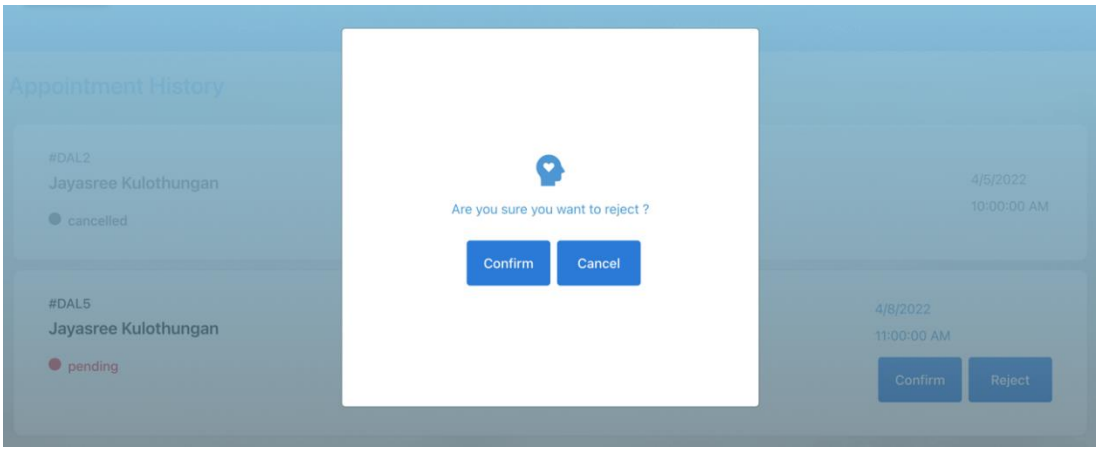

Figure 4.1.10 Rejection modal on chaplain view

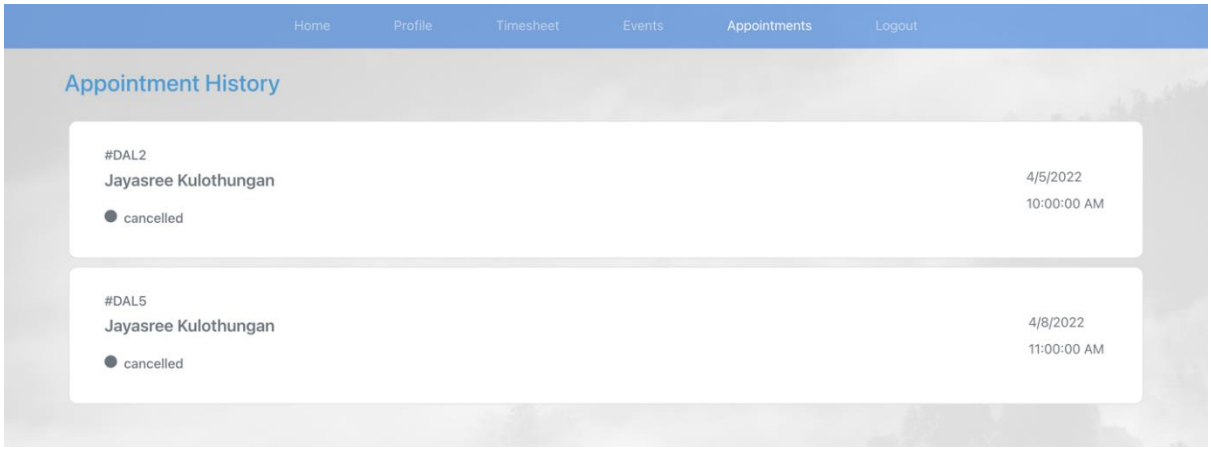

Figure 4.1.11 Appointment history on chaplain view – Cancelled

Once the chaplain confirms or rejects the appointment it will reflect on the student's view.

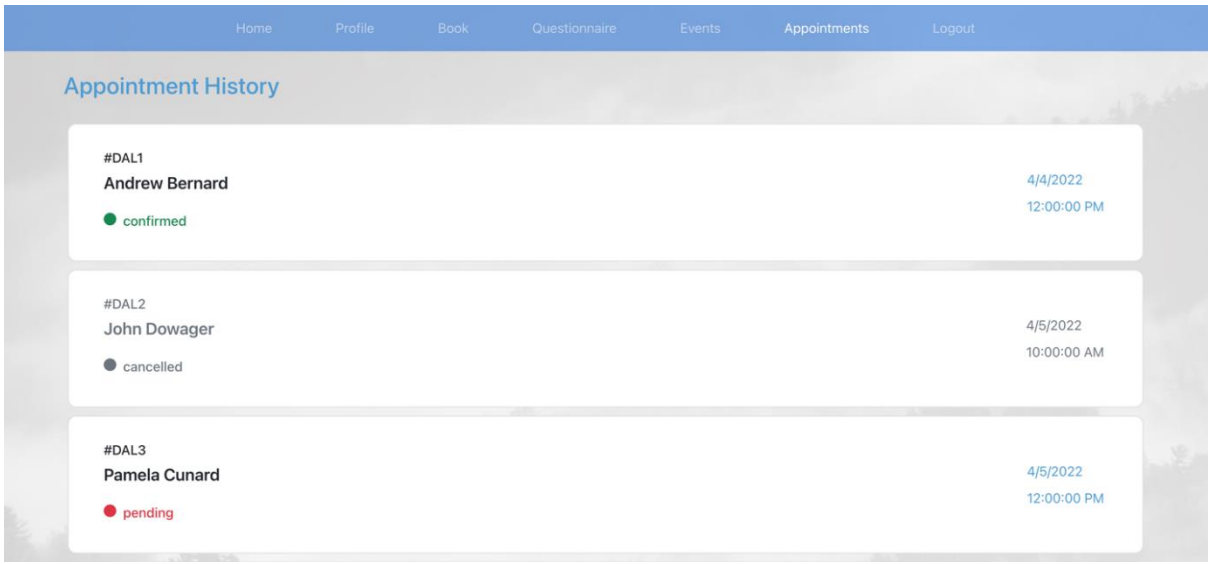

Figure 4.1.12 Appointment history on student view

4.2 Back-end Data Management

The data is stored on the back end and developed using Django Rest Framework. First, the model needed for the appointment was developed and then implemented. The table Appointment is defined as Class and the fields – chaplain_id, user_id, slot and status are specified. The field chaplain_id and user_id are foreign keys and is in relationship with the Chaplain table and Student table respectively.

```
class Appointment(models.Model):
    slot = models.DateTimeField(null=True)
    user_id = models.ForeignKey(DalUser, on_delete=models.CASCADE)
    chaplain_id = models.ForeignKey(Chaplain, on_delete=models.CASCADE)
    status = models.CharField(max_length=255)
```

Figure 4.2.1 Appointment Modal

Then, the methods needed to perform the sending and retrieving data operations are defined in views. For the appointment booking page, we require three operations – Get, Put and Post. For retrieving the data from the database, we require the chaplain_id and user_id.

```
class UserAppointmentList(APIView):
    def post(self, request):
        serializer = AppointmentSerializer(data=request.data)
        serializer.is_valid(raise_exception=True)
        serializer.save()
        return Response(serializer.data, status=status.HTTP_201_CREATED)


class AppointmentList(APIView):
    def get(self, request, user_id, chaplain_id):
        queryset = Appointment.objects.all()
        if user_id != 0:
            queryset = queryset.filter(user_id=user_id)
        if chaplain_id != 0:
            queryset = queryset.filter(chaplain_id=chaplain_id)
        serializer = AppointmentSerializer(queryset, many=True)
        return Response(serializer.data)


class UserAppointmentDetails(APIView):
    def put(self, request, id):
        appointment = get_object_or_404(Appointment, pk=id)
        serializer = AppointmentSerializer(appointment, data=request.data)
        serializer.is_valid(raise_exception=True)
        serializer.save()
        return Response(serializer.data)
```

Figure 4.2.2 Appointment View

To verify if all the fields are defined and to validate the data received from the front-end, we use Serializers.

```
class AppointmentSerializer(serializers.ModelSerializer):
    slot = serializers.DateTimeField()
    status = serializers.CharField(max_length=255)
    daluser = UserInfoSeriaizer(source="user_id", read_only=True)
    chaplain = ChaplinInfoSeriaizer(source="chaplain_id", read_only=True)

    class Meta:
        model = Appointment
        fields = [
            "id",
            "user_id",
            "chaplain_id",
            "slot",
            "status",
            "daluser",
            "chaplain",
        ]
```

Figure 4.2.3 Appointment Serializer

4.3 Integration

We integrated the backend and frontend by using AXIOS interceptors. After setting up the HTTP services to handle GET, POST, PUT and DELETE functions, the methods required for the appointment are specified in the appointment.js Service. Here, the endpoints are specified precisely. The chaplainId and user_id are passed to the get method as it is required by the Django views where GET. The put method is used to update the slot status.

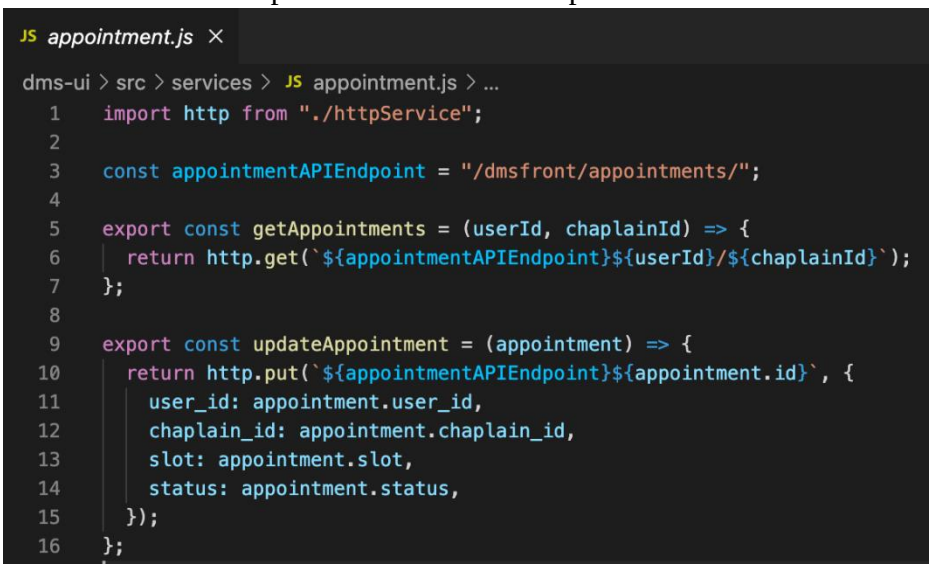

Figure 4.3.1 Specifying get and post methods

To bind the data, we use the concept of Context that React offers. This allows the passing of data to different components within the react app.

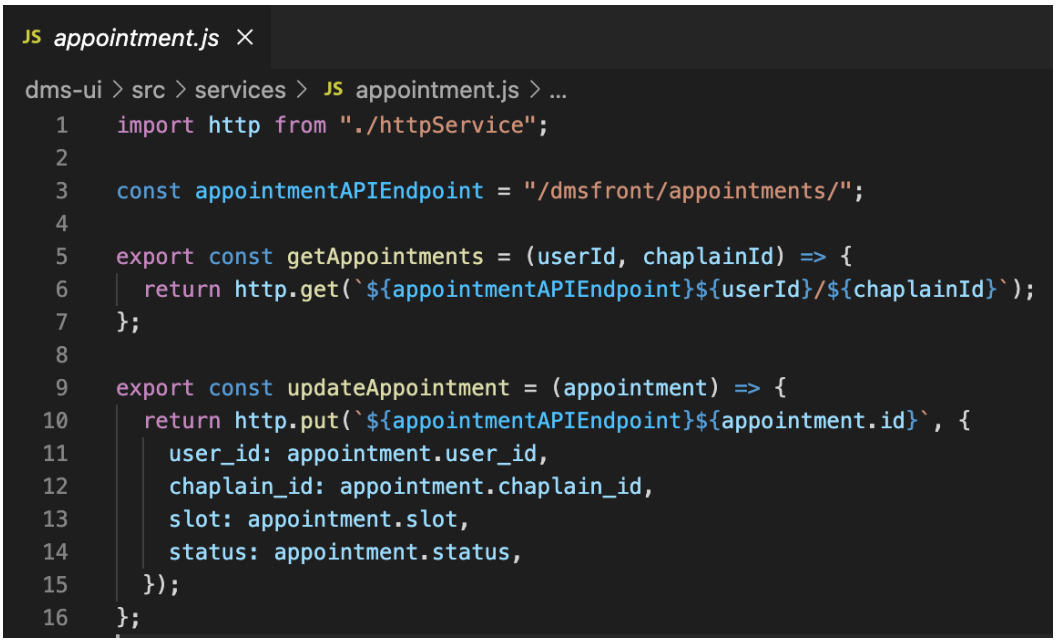

Figure 4.3.2 Defining appointment context

The handling of data is done in app.js. The get method is called in the useEffect hook of React. This allows the Get method to be called as soon as the app runs thereby retrieving the data immediately after the user logs in. according to the user logged in the data, views change.

```
useEffect(() => {
  const getData = async () => {
    const currentUser = await auth.getCurrentUser();
    setUser(currentUser);

    if (currentUser) {
      let user_id, chaplain_id;
      if (currentUser.user.is_staff) {
        user_id = 0;
        chaplain_id = currentUser.id;
      } else {
        user_id = currentUser.id;
        chaplain_id = 0;
      }
      const { data: dataAppointments } = await getAppointments(
        user_id,
        chaplain_id
      );
      setAppointments(dataAppointments);
```

Figure 4.3.3 Getting the data from backend

The put method is used to update the slot status.

```
const handleConfirmClick = async (action, selectedAppointment) => {
  const originalAppointments = appointments;
  const appointmentsData = [...appointments];
  const index = appointmentsData.indexOf(selectedAppointment);
  appointmentsData[index] = { ...selectedAppointment };
  if (action === "confirm") {
    appointmentsData[index].status = "confirmed";
  } else if (action === "reject") {
    appointmentsData[index].status = "cancelled";
  }
  setAppointments(appointmentsData);
  try {
    await updateAppointment(appointmentsData[index]);
  } catch (ex) {
    if (
      ex.response &&
      ex.response.status >= 400 &&
      ex.response.status < 500
    ) {
      setAppointments(originalAppointments);
      toast.error(<ListError errors={Object.values(ex.response.data)} />);
    }
  }
};
```

Figure 4.3.4 Updating the slots

Once the methods are specified, they need to be passed to the components using Context.

```
<EventContext.Provider value={{ events, handleAddEvent, handleBooking }}>
```

4.4 Testing

The testing tool used is pytest.

4.4.1 Unit Testing

We have used mocking to perform unit testing. The response for Get, Post and Put methods are checked to whether the connection is made, and the end pendpointsoints are working as they should.

```
# USERAPPOINTMENTLIST - mocked the post method from the UserAppointmentList
def test_user_appointment_list_put(self):
    fake_response = Response()
    fake_response.status_code = status.HTTP_202_ACCEPTED
    mock_put = mock.Mock(
        name="AppointmentList-put", return_value=fake_response
    )
    self.userappointment_list.put = mock_put

    assert (
        self.userappointment_list.put().status_code
        == status.HTTP_202_ACCEPTED
    )
```

Figure 4.4.1.1 User Appointment List – Unit Testing

```
# APPOINTMENTLIST - mocked the post method from the AppointmentList
def test_appointment_list_get(self):
    fake_response = Response()
    fake_response.status_code = 200
    mock_get = mock.Mock(
        name="AppointmentList-put", return_value=fake_response
    )
    self.appointment_list.get = mock_get

    assert self.appointment_list.get().status_code == status.HTTP_200_OK
```

Figure 4.4.1.2 Appointment List – Unit Testing

```python
# USERAPPOINTMENTDETAILS
# mocked the post method from the UserAppointmentDetails
def test_user_appointment_detail_put(self):
    fake_response = Response()
    fake_response.status_code = status.HTTP_202_ACCEPTED
    mock_put = mock.Mock(
        name="AppointmentList-put", return_value=fake_response
    )
    self.userappointment_detail.put = mock_put

    assert (
        self.userappointment_detail.put().status_code
        == status.HTTP_202_ACCEPTED
    )
```

Figure 4.4.1.3 Appointment Details – Unit Testing

4.4.2 Interesting Testing

In integration testing, we are passing values to the Get and Post methods and checking if they are returning appropriate messages.

```python
# APPOINTMNETS-test on appointment booking
def test_bookappointments_post(self):
    # daluser added
    daluser_response = requests.post(
        self.url,
        {
            "email": "hari@lakha.ca",
            "password": "group@123",
            "first_name": "harshit",
            "last_name": "lakhani",
            "is_staff": False,
        },
    )
    daluser_id = daluser_response.json()["id"]
    data = {"user_id": id, "phone": "9024538293", "user": "harshit"}
    requests.post(self.dal_user_url, data)
```

Figure 4.4.2.1 Book Appointment – Integration Testing

```
# chaplin added
chaplin_response = requests.post(
    self.url,
    {
        "email": "ramesh@lakha.ca",
        "password": "group@123",
        "first_name": "aditya",
        "last_name": "mahale",
        "is_staff": True,
    },
)
chaplin_id = chaplin_response.json()["id"]
data = {
    "user_id": id,
    "phone": "9024538293",
    "user": "harshit",
    "religion": "Hindu",
    "description": "i am a chaplin",
}
# chaplin added
requests.post(self.chaplin_url, data)

response = requests.get(
    self.appointment_url + str(daluser_id) + "/" + str(chaplin_id)
)
assert response.status_code == status.HTTP_200_OK
```

Figure 4.4.2.2 Appointments - Integration Testing

5. Upcoming Events Feature

5.1 Feature Overview

The events feature has been designed to let the users know about spiritual events that they can participate in. The events can be viewed and added by both students and chaplains. On clicking the Add Event button on the right corner a modal opens that allows the users to enter details about the event.

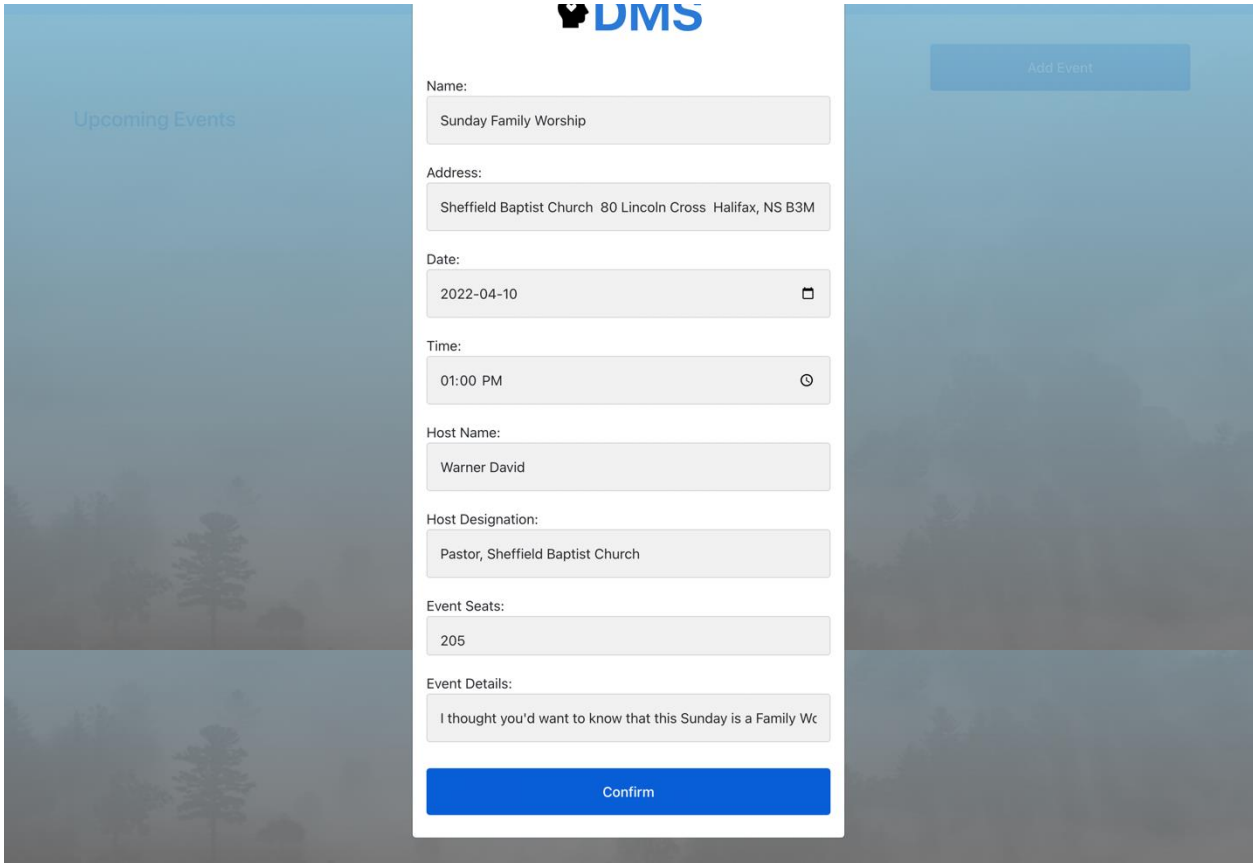

Figure 5.1.1 Adding events modal

The upcoming events can be seen by clicking the events tab on the header. A grid of event cards is displayed. The card contains details such as name and date.

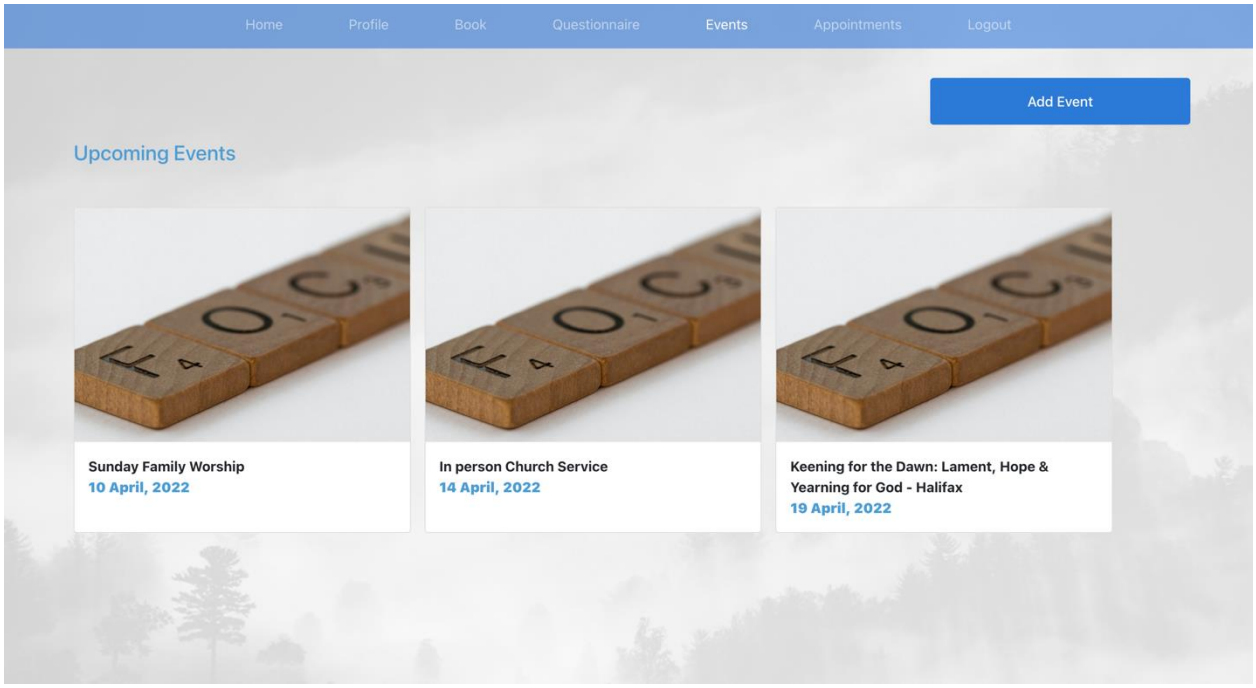

Figure 5.1.2 Events list

Clicking on a specific events card will open another page that gives a detailed description of the event. The details such as event name, address, date and time, the number of seats available and a description of the event. It also provides a button to allow booking.

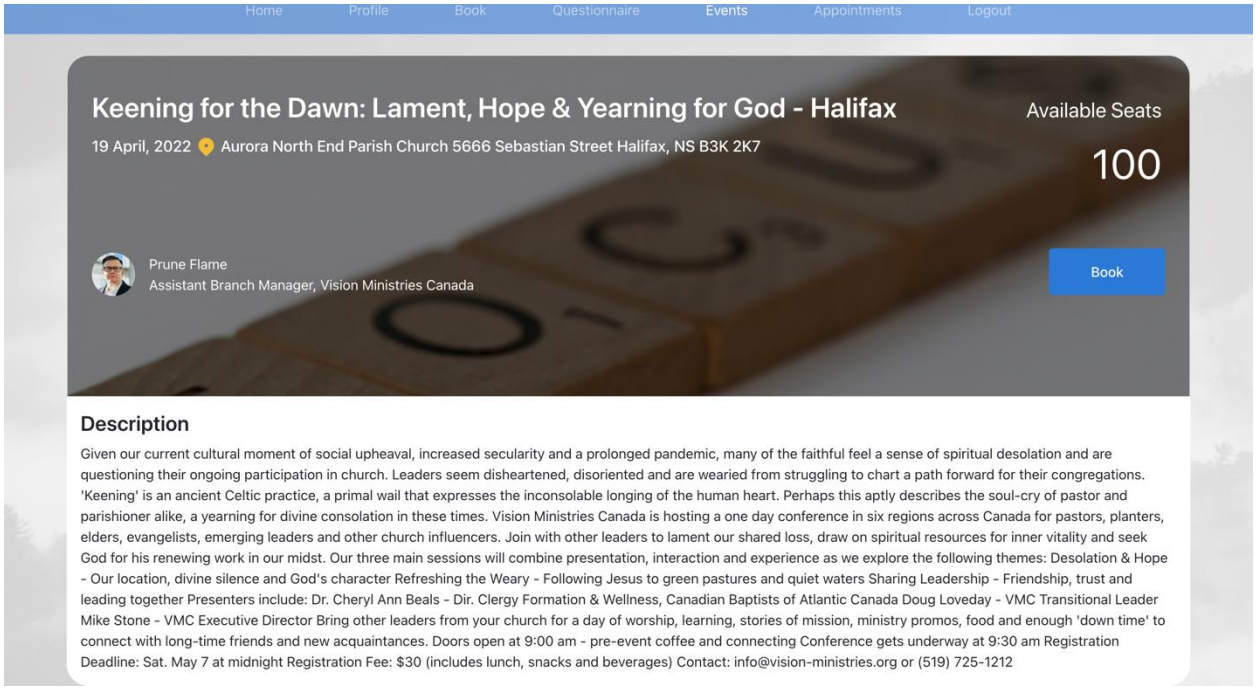

Figure 5.1.3 Details of each event

5.2 Back-end Data Management

The data is stored on the back end developed using Django Rest Framework. First, the model needed for the events was developed and then implemented. The table Event is defined as Class and the fields – event_title, event_date, event_location, event_description, available_seats, host_name, host_details is specified.

```
class Event(models.Model):
    event_title = models.CharField(max_length=150)
    event_date = models.DateTimeField()
    event_location = models.TextField()
    event_description = models.TextField()
    available_seats = models.PositiveSmallIntegerField()
    host_name = models.CharField(max_length=100)
    host_details = models.CharField(max_length=250)

    def __str__(self):
        return self.event_title
```

Figure 5.2.1 Event model

Then, the methods needed to perform the sending and retrieving data operations are defined in views. For the events, we require four operations – Get all events, Get details of a particular event, Post and Put.

1. The first operation is getting data of all events. This method returns the entire events data from the backend.
2. The second operation is adding events. The data got from a form is passed from the front end and is stored.
3. The third operation is to get the details of a particular event. For this, the event id is passed to specify the correct event details to be returned.
4. The final operation is put or update operation where the available seats get updated each time someone books a seat for an event.

```python
@api_view(["GET"])
def get_events(request):
    events = Event.objects.all()
    serializer = EventSerializer(events, many=True)
    return Response(serializer.data)


@api_view(["POST"])
def add_event(request):
    serializer = EventSerializer(data=request.data)
    if serializer.is_valid():
        serializer.save()
    return Response(serializer.data)


@api_view(["GET"])
def get_event(request, id):
    event = Event.objects.filter(id=id)
    serializer = EventSerializer(event, many=True)
    return Response(serializer.data)


@api_view(["PUT"])
def book_event(request, id):
    event = get_object_or_404(Event, id=id)
    data = {"available_seats": event.available_seats - 1}
    serializer = EventSerializer(event, data=data, partial=True)

    if serializer.is_valid():
        serializer.save()
        return Response(serializer.data)

    return Response(serializer.errors, status=status.HTTP_400_BAD_REQUEST)
```

Figure 5.2.2 Event get, put, and post methods

To verify if all the fields are defined and to validate the data received from the front-end, we use Serializers. All the fields are mandated.

```python
class EventSerializer(serializers.ModelSerializer):
    class Meta:
        model = Event
        fields = "__all__"
```

Figure 5.2.3 Event serializer

5.3 Integration

We integrated the backend and frontend by using AXIOS interceptors. After setting up the HTTP services to handle GET, POST, PUT and DELETE functions, the methods required for the events page are specified in events.js Service. Here, the endpoints are specified precisely.

The form data is passed to the fields in the post method as required by the django view.

```
import http from "./httpService";

const eventsAPIEndpoint = "/dmsfront/events/";
const addEventAPIEndpoint = "/dmsfront/addevent/";

export const getEvents = () => {
  return http.get(`${eventsAPIEndpoint}`);
};

export const addEvent = (event) => {
  const date = `${event.date}T${event.time}:00Z`;
  return http.post(`${addEventAPIEndpoint}`, {
    event_title: event.eventName,
    event_date: date,
    event_location: event.address,
    event_description: event.eventDetails,
    available_seats: event.seats,
    host_name: event.hostname,
    host_details: event.hostDesignation,
  });
};

export const updateEvent = (id) => {
  return http.put(`${eventsAPIEndpoint}${id}/book`);
};
```

Figure 5.3.1 Specifying the get and post methods of events service

To bind the data, we use the concept of Context that React offers. This allows the passing of data to different components within the react app.

JS eventContext.js ×

dms-ui > src > context > JS eventContext.js > ...

```
1   import React from "react";
2
3   const EventContext = React.createContext();
4
5   export default EventContext;
```

Figure 5.3.2 Specifying event context

The handling of data is done in app.js. The get method is called in the useEffect hook of React. This allows Get method to be called as soon as the app runs thereby retrieving the data immediately after the user logs in.

```
const { data: dataEvents } = await getEvents();
setEvents(dataEvents);
```

The post method sends form data to the backend and is specified in app.js.

```
const handleAddEvent = async (e, event) => {
  e.preventDefault();
  try {
    await addEvent(event);
    window.location = "/";
  } catch (ex) {
    if (
      ex.response &&
      ex.response.status >= 400 &&
      ex.response.status < 500
    ) {
      toast.error(<ListError errors={Object.values(ex.response.data)} />);
    }
  }
};
```

Figure 5.3.3 Sending the form data

The put method is also handled in app.js where the slots are updated using events_id.

```
const handleBooking = async (event) => {
  try {
    await updateEvent(event.id);
    window.location = "/";
  } catch (ex) {
    if (
      ex.response &&
      ex.response.status >= 400 &&
      ex.response.status < 500
    ) {
      toast.error(<ListError errors={Object.values(ex.response.data)} />);
    }
  }
};
```

Figure 5.3.4 Requesting the HTTP server for Post method

Once the methods are specified, they need to be passed to the components using Context.

```
<EventContext.Provider value={{ events, handleAddEvent, handleBooking }}>
```

Finally, the data is sent from and received in events.jsx where it is bound.

5.4 Testing

The testing tool used is pytest.

5.4.1 Unit Testing

We have used mocking to perform unit testing. The response for Get, Post and Put methods are checked to whether the connection is made, and the endpoints are working as they should.

Post:

```
@pytest.mark.unit
class TestEventsUnit:
    dal_user = DalUserList()
    dal_user_detail = DalUserDetail()
    chaplin_detail = ChaplainDetail()
    chaplin_list = ChaplainList()
    userappointment_list = UserAppointmentList()
    appointment_list = AppointmentList()
    userappointment_detail = UserAppointmentDetails()

    # DALUSER - mocked the post method from the DalUserList
    def test_dal_user_list_post(self):
        fake_response = Response()
        fake_response.status_code = status.HTTP_201_CREATED
        mock_post = mock.Mock(
            name="DalUserList-post", return_value=fake_response
        )
        self.dal_user.post = mock_post

        assert self.dal_user.post().status_code == status.HTTP_201_CREATED
```

Figure 5.4.1.1 Unit testing for post method

Get:

```
    # DALUSER - mocked the get method from the DalUserDetails
    def test_dal_user_detail_get(self):
        fake_response = Response()
        fake_response.status_code = 200
        mock_get = mock.Mock(
            name="DalUserDetail-get", return_value=fake_response
        )
        self.dal_user_detail.get = mock_get

        assert self.dal_user_detail.get().status_code == status.HTTP_200_OK
```

Figure 5.4.1.2 Unit testing for get method

Put:

```
# DALUSER - mocked the put method from the DalUserDetails
def test_dal_user_detail_put(self):
    fake_response = Response()
    fake_response.status_code = status.HTTP_202_ACCEPTED
    mock_put = mock.Mock(
        name="DalUserDetail-put", return_value=fake_response
    )
    self.dal_user_detail.put = mock_put

    assert (
        self.dal_user_detail.put().status_code == status.HTTP_202_ACCEPTED
    )
```

Figure 5.4.1.3 Unit testing for put method

5.4.2 Integration Testing

In integration testing, we are passing values to the Get and Post methods and checking if they are returning appropriate messages.

Getting all events:

```
@pytest.mark.integration
class TestEventsAuth:

    url = "http://localhost:8000/auth/users/"
    dal_user_url = "http://localhost:8000/dmsfront/dalusers/"
    chaplin_url = "http://localhost:8000/dmsfront/chaplains/"
    events_url = "http://localhost:8000/dmsfront/"
    appointment_url = "http://localhost:8000/dmsfront/appointments/"

    # EVENTS-test on getting all the events
    def test_events_get(self):
        # events
        response = requests.get(self.events_url + "events/")
        assert response.status_code == status.HTTP_200_OK
```

Figure 5.4.2.1 Testing the endpoint

Getting a particular event:

```
# EVENTS-test on getting an events
def test_particular_event_get(self):
    # events
    date_time = models.DateTimeField(
        default=datetime.datetime.now, blank=True
    )
    response = requests.post(
        (self.events_url + "addevent/"),
        {
            "event_title": "abcdbd",
            "event_date": date_time,
            "event_location": "svhsjnv",
            "event_description": "sdcjdshvn",
            "available_seats": 5,
            "host_name": "dvhdfnvf",
            "host_details": "dchdbfchfd",
        },
    )
```

Figure 5.4.2.2 Get Event - Integration testing by passing values

Post:

```
# EVENTS-test on adding an events
def test_add_events_post(self):
    # events
    date_time = models.DateTimeField(
        default=datetime.datetime.now, blank=True
    )
    response = requests.post(
        (self.events_url + "addevent/"),
        {
            "event_title": "abcdbd",
            "event_date": date_time,
            "event_location": "xyanascaasc3",
            "event_description": "sdcjdshvn",
            "available_seats": 10,
            "host_name": "acas^7anc",
            "host_details": "dchdbfchfd",
        },
    )

    assert response.status_code == status.HTTP_200_OK
```

Figure 5.4.2.3 Add Event - Integration testing by passing values

6. Timesheet Feature

6.1 Feature Overview

The timesheet feature is added for the DMS staff members to document the hours they have worked. This feature is developed using React-big-calendar. It is a calendar component for managing events and dates. The moment is used as a localizer to format dates and times to the appropriate time zones. The calendar offers different views such as – Month, Week, Work Week, Day, and Agenda. The timesheet can only be viewed by the staff and cannot be edited once created.

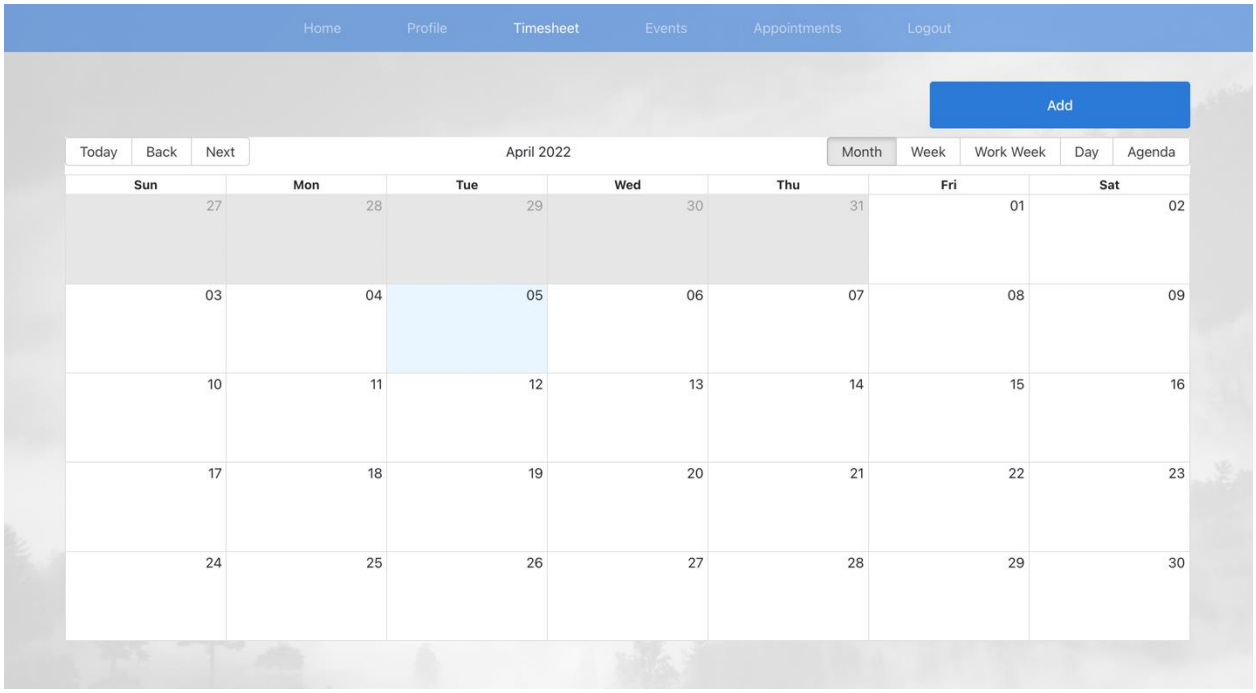

Figure 6.1.1 View of timesheet

On clicking the Add Event button, the chaplains can add the details of the student they have met, and the date and time slots.

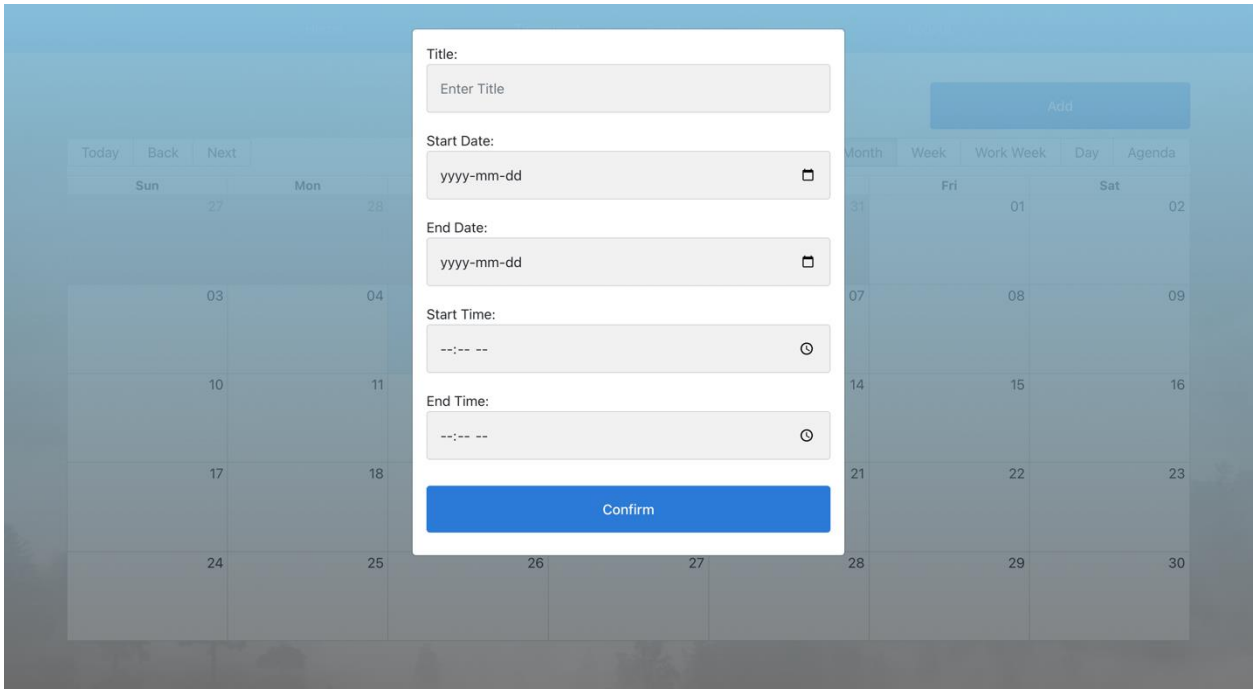

Figure 6.1.2 Modal to add timesheet event

Once the slot is added, it can be viewed in the views mentioned previously.

Month:

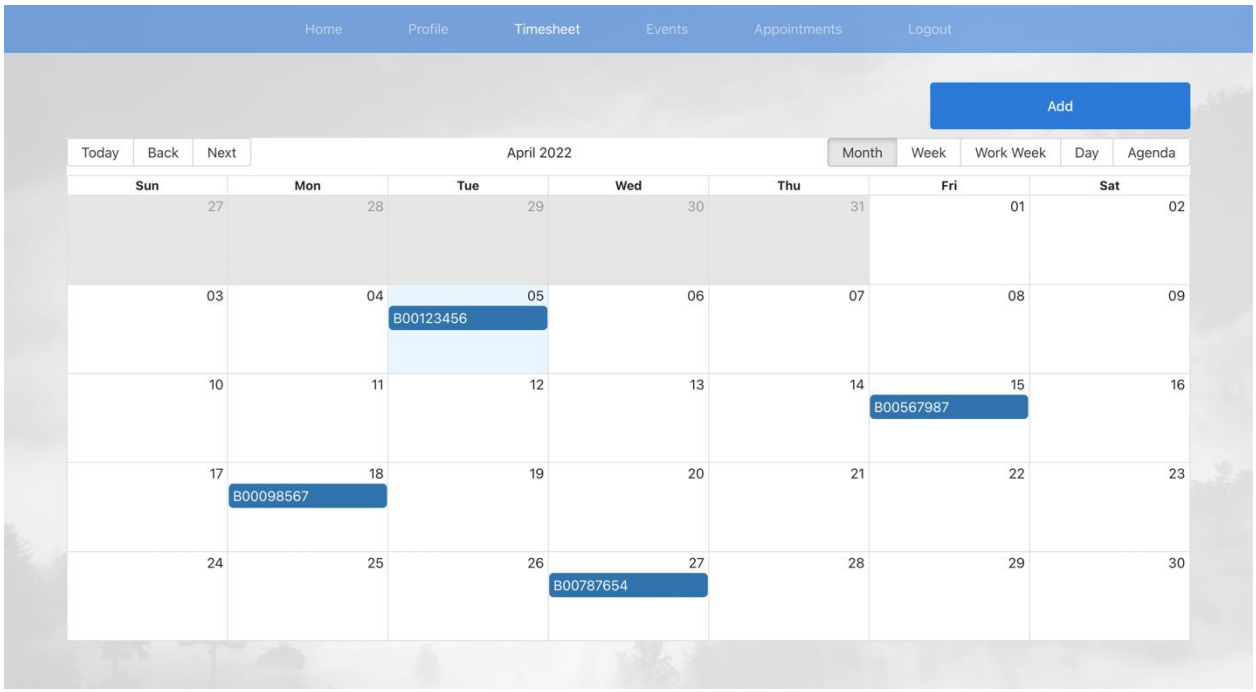

Figure 6.1.3 Month view

Week:

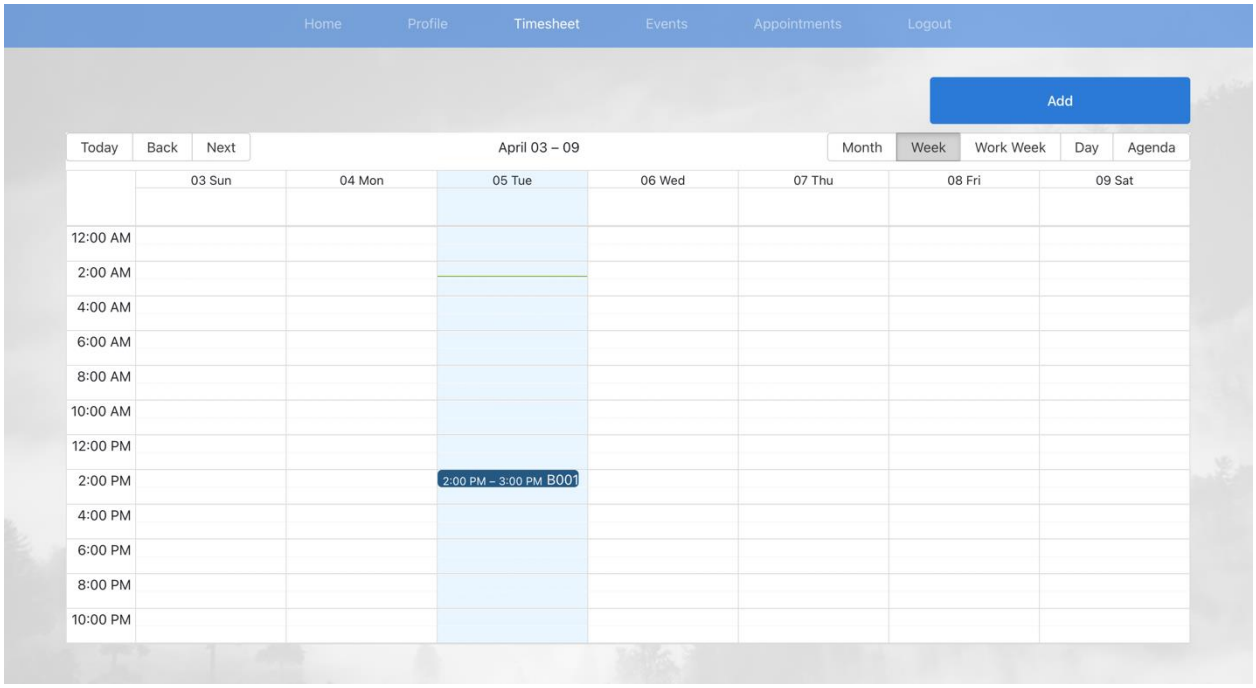

Figure 6.1.4 Week view

Work Week:

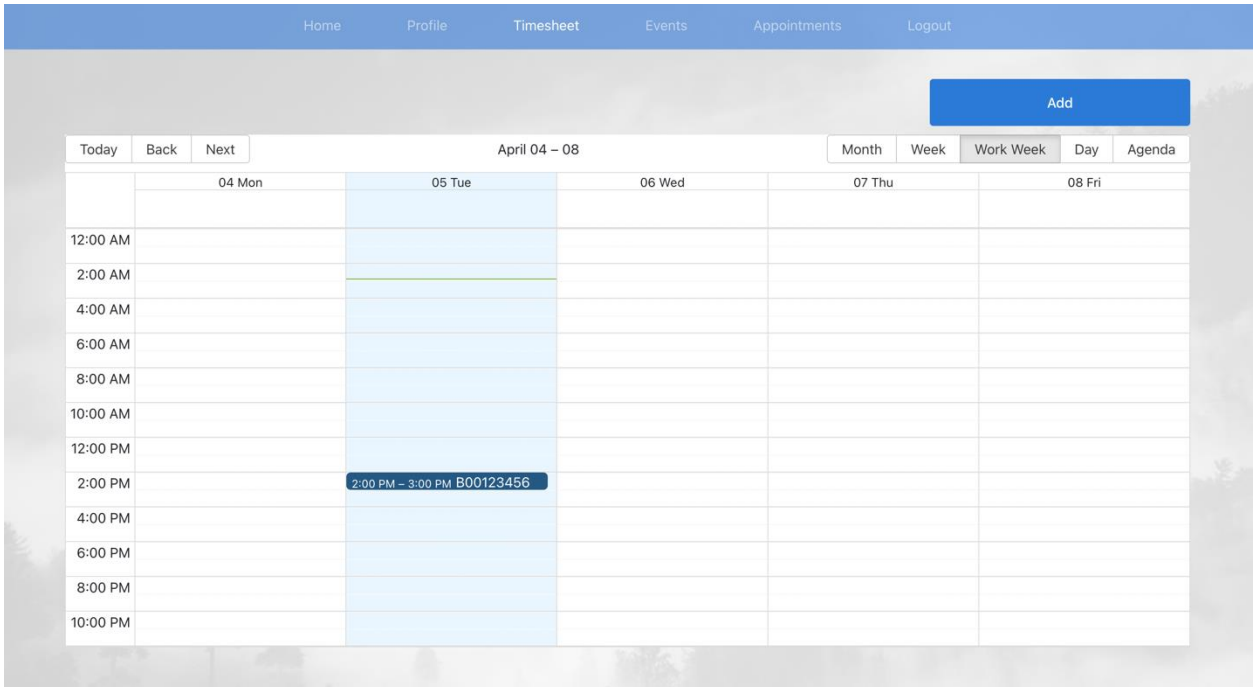

Figure 6.1.5 Workweek view

Day:

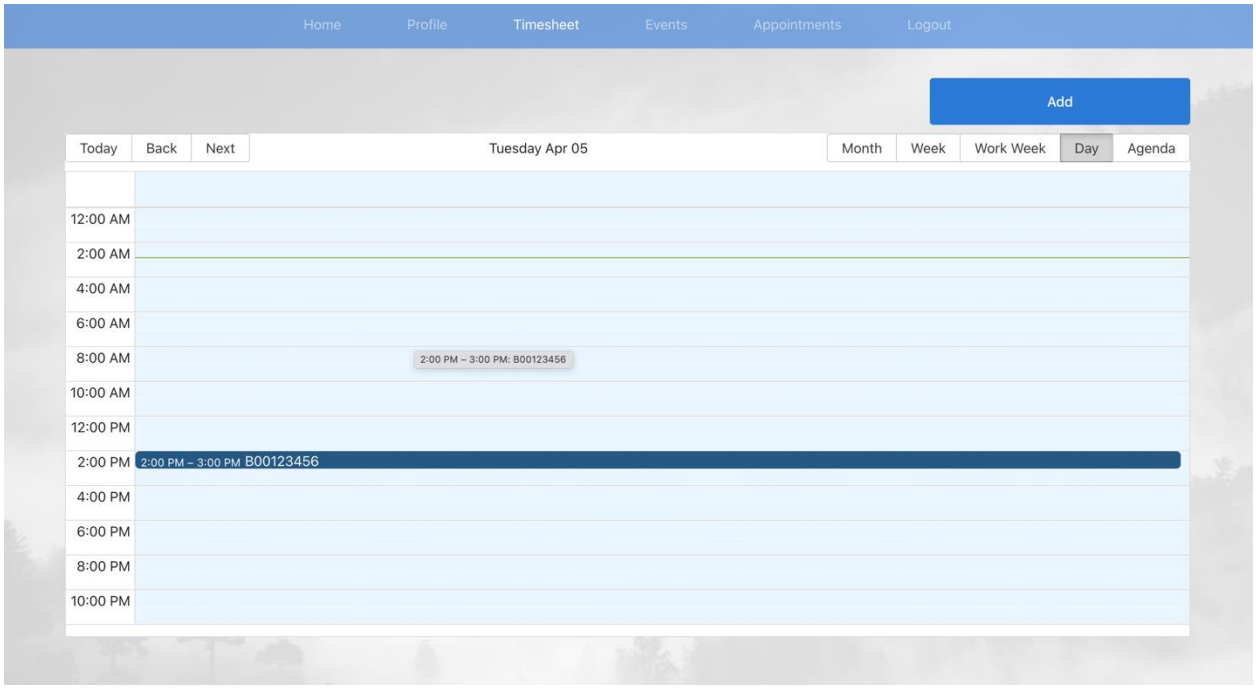

Figure 6.1.6 Day view

Agenda:

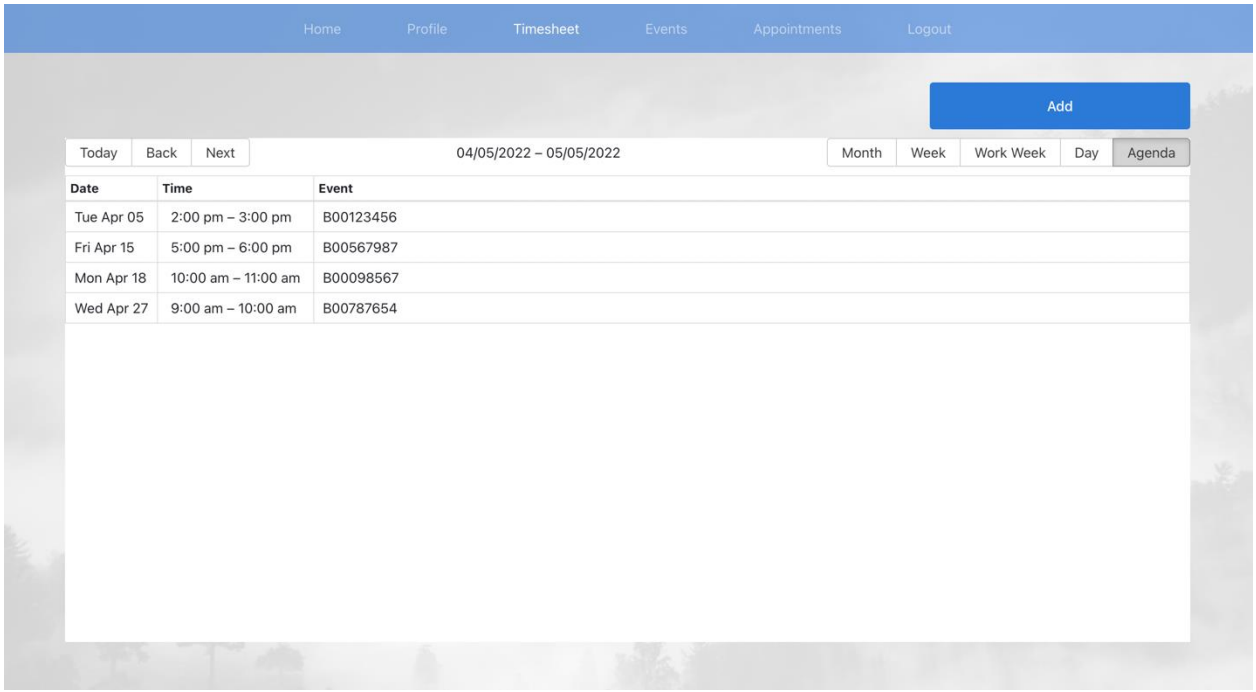

Figure 6.1.7 Agenda view

6.2 Back-end Data Management

The data is stored on the back end developed using Django Rest Framework. First, the model needed for the timesheet was developed and then implemented. The table Timesheet is defined as Class and the fields – chaplain_id, title, start, end is specified. The field chaplain_id is a foreign key and is in relationship with the Chaplain table. The changes made to the chaplain reflects on the timesheet.

```
class TimeSheet(models.Model):
    chaplain_id = models.ForeignKey(Chaplain, on_delete=models.CASCADE)
    title = models.CharField(max_length=150)
    start = models.TextField(blank=False)
    end = models.TextField(blank=False)
```

Figure 6.2.1 Timesheet modal

Then, the methods needed to perform the sending and retrieving data operations are defined in views. For the timesheet, we require two operations – Get and Post. For retrieving the data from the database, we require the chaplain_id that is currently logged in.

```
class TimeSheetList(APIView):
    def post(self, request):
        serializer = TimeSheetSerializer(data=request.data)
        serializer.is_valid(raise_exception=True)
        serializer.save()
        return Response(serializer.data, status=status.HTTP_201_CREATED)


class TimeSheetDetail(APIView):
    def get(self, request, chaplain_id):
        queryset = TimeSheet.objects.filter(chaplain_id=chaplain_id)
        serializer = TimeSheetSerializer(queryset, many=True)
        return Response(serializer.data)
```

Figure 6.2.2 Timesheet views

To verify if all the fields are defined and to validate the data received from the front-end, we use Serializers.

```python
class TimeSheetSerializer(serializers.ModelSerializer):
    title = serializers.CharField(max_length=150)
    start = serializers.CharField(max_length=150)
    end = serializers.CharField(max_length=150)

    class Meta:
        model = TimeSheet
        fields = [
            "chaplain_id",
            "title",
            "start",
            "end",
        ]
```

Figure 6.2.3 Timesheet serializer

6.3 Integration

We integrated the backend and frontend by using AXIOS interceptors. After setting up the HTTP services to handle GET, POST, PUT and DELETE functions, the methods required for the timesheet are specified in timesheet.js Service. Here, the endpoints are specified precisely. The chaplainId is passed to the get method as it is required by the Django views where GET is defined so that only the details of the currently logged-in user are retrieved. In the post method, the form data is passed.

JS *timesheet.js* ×

dms-ui > src > services > JS timesheet.js > ...

```javascript
import http from "./httpService";

const timesheetAPIEndpoint = "/dmsfront/timesheet";

export const getTimesheet = async (chaplainId) => {
    return await http.get(`${timesheetAPIEndpoint}/${chaplainId}`);
};

export const createTimesheet = (timesheet) => {
    return http.post(`${timesheetAPIEndpoint}/`, {
        chaplain_id: timesheet.chaplain_id,
        title: timesheet.title,
        start: timesheet.start,
        end: timesheet.end,
    });
};

```

Figure 6.3.1 Timesheet.js

To bind the data, we use the concept of Context that React offers. This allows the passing of data to different components within the react app.

timesheetContext.jsx

dms-ui > src > context > timesheetContext.jsx > ...

```
1  import React from "react";
2
3  const TimesheetContext = React.createContext();
4
5  export default TimesheetContext;
6
```

Figure 6.3.2 Specifying context for timesheet

The handling of data is done in app.js. The get method is called in the useEffect hook of React. This allows Get method to be called as soon as the app runs thereby retrieving the data immediately after the user logs in. Another constraint that is checked is that the user logged in needs to be exclusively a chaplain.

```
useEffect(() => {
    const getData = async () => {
        const currentUser = await auth.getCurrentUser();
        setUser(currentUser);

        if (currentUser) {
            let user_id, chaplain_id;
            if (currentUser.user.is_staff) {
                user_id = 0;
                chaplain_id = currentUser.id;
            } else {
                user_id = currentUser.id;
                chaplain_id = 0;
            }
            if (currentUser.user.is_staff) {
                const { data: dataTimesheet } = await getTimesheet(chaplain_id);
                setTimesheet(dataTimesheet);
            }
        }
    };
```

Figure 6.3.3 Sending request to the server to get data

The post method sends a form data to the backend and is specified in app.js

```
const handleTimesheetSubmit = async (createTimeEvent) => {
    try {
        await createTimesheet(createTimeEvent);
        window.location = "/";
    } catch (ex) {
        if (
            ex.response &&
            ex.response.status >= 400 &&
            ex.response.status < 500
        ) {
            toast.error(<ListError errors={Object.values(ex.response.data)} />);
        }
    }
};
```

Figure 6.3.4 Handling post request

Once the methods are specified, they need to be passed to the components using Context.

```
<TimesheetContext.Provider value={{ timesheet, handleTimesheetSubmit }}>
```

Finally, the data is sent from timesheet.jsx where it is got as form data and is displayed on the react-big-calendar component.

6.4 Testing

The testing tool used is pytest.

6.4.1 Unit Testing

We have used mocking to perform unit testing. The response for both Get and Post methods are checked to whether the connection is made and the endpoints are working as they should.

```python
@pytest.mark.unit
class TestTimesheetUnit:

    timesheet_list = TimeSheetList()
    timesheet_detail = TimeSheetDetail()

    # TIMESHEETLIST - mocked the post method from the TimeSheetList
    def test_timesheet_list_post(self):
        fake_response = Response()
        fake_response.status_code = status.HTTP_202_ACCEPTED
        mock_post = mock.Mock(
            name="TimeSheetList-post", return_value=fake_response
        )
        self.timesheet_list.post = mock_post

        assert (
            self.timesheet_list.post().status_code == status.HTTP_202_ACCEPTED
        )

    # TIMESHEETDETAIL - mocked the get method from the TimeSheetDetail
    def test_timesheet_detail_get(self):
        fake_response = Response()
        fake_response.status_code = status.HTTP_200_OK
        mock_post = mock.Mock(
            name="TimeSheetDetail-get", return_value=fake_response
        )
        self.timesheet_detail.get = mock_post

        assert self.timesheet_detail.get().status_code == status.HTTP_200_OK
```

Figure 6.4.1.1 Unit testing endpoints

6.4.2 Integration Testing

In integration testing, we are passing values to the Get and Post methods and checking if they are returning appropriate messages.

Post:

```python
def test_timesheetlist_post(self):

    # Adding chaplain
    chaplin_response = requests.post(
        self.url,
        {
            "email": "kalpit@dal.ca",
            "password": "group@10",
            "first_name": "Kalpit",
            "last_name": "Machhi",
            "is_staff": True,
        },
    )

    chaplin_id = chaplin_response.json()["id"]
    chaplain_data = {
        "user_id": id,
        "phone": "9024832051",
        "user": "kalpit",
        "religion": "Hindu",
        "description": "Certified clergy member.",
    }

    requests.post(self.chaplin_url, chaplain_data)

    response = requests.get(self.timesheet_url + str(chaplin_id))
    assert response.status_code == status.HTTP_200_OK
```

Figure 6.4.2.1 Passing values to the post method and checking the response

Get:

```
def test_timesheetdetail_get(self):

    chaplin_response = requests.post(
        self.url,
        {
            "email": "anas@dal.ca",
            "password": "group@10",
            "first_name": "Anas",
            "last_name": "Malvat",
            "is_staff": True,
        },
    )

    chaplin_id = chaplin_response.json()["id"]
    chaplain_data = {
        "user_id": id,
        "phone": "9024832051",
        "user": "Anas",
        "religion": "Hindu",
        "description": "Member",
    }

    requests.post(self.chaplin_url, chaplain_data)

    response = requests.get(self.timesheet_url + str(chaplin_id))
    assert response.status_code == status.HTTP_200_OK
```

Figure 6.4.2.2 Passing values to the post method and checking the response

7. Spiritual Wellness Score Feature

7.1 Feature Overview

"Spiritual Wellness Score" feature is provided for students. The goal of this feature is to achieve balanced wellness. It helps measure spiritual wellness by asking several questions related to mental health. A user will be asked nine questions. Each question has three options:

(1) Very seldom
(2) Sometimes/occasionally
(3) Almost always

Each option has different points. Based on these points, the final score is calculated, and the result is given to the user. A score between 13 and 18 points is considered an excellent strength in the dimension. A score between 7 and 12 points depicts room for improvement. And the score below 7 points suggests a lot of work is required.

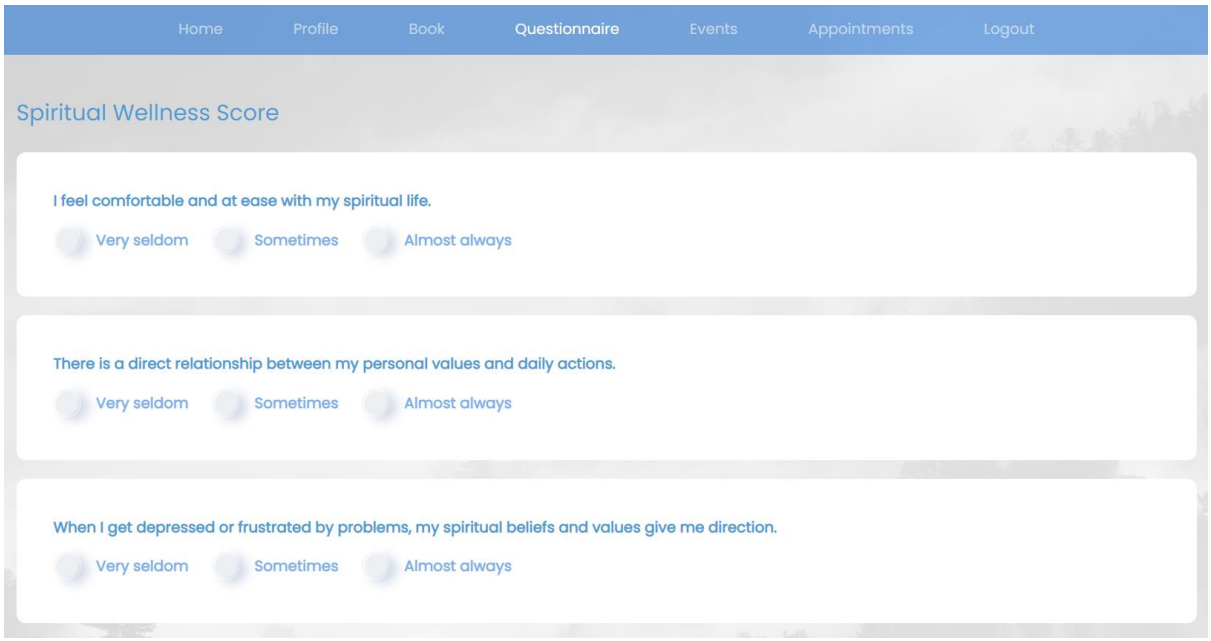

Figure 7.1.1 Spiritual Wellness Score UI

The student needs to answer all the questions, and the result will be shown in the modal based on the points scored. The resulting modal contains a progressive bar showing the points scored.

Figure 7.1.2 Result Modal - Score < 7 points

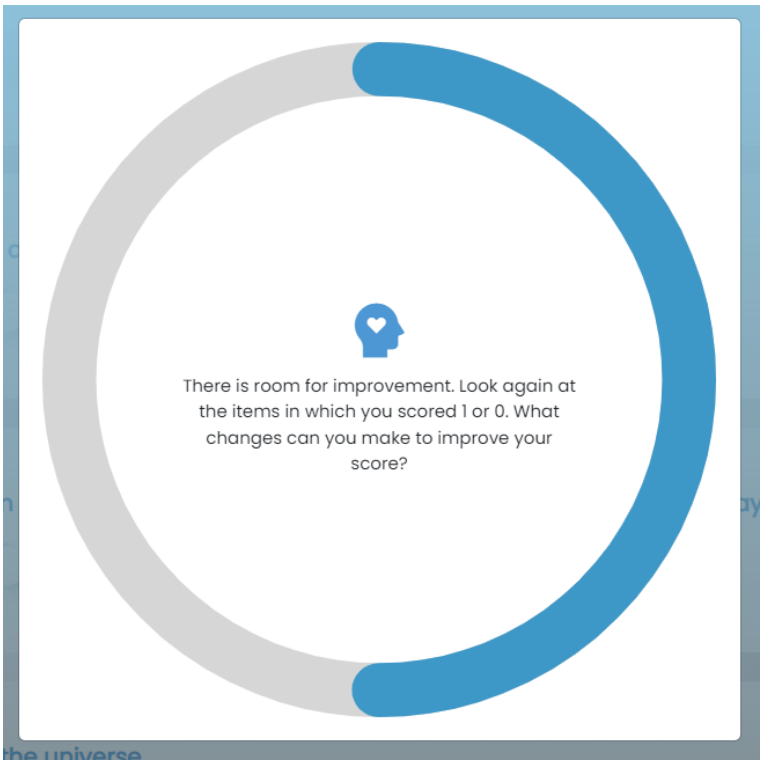

Figure 7.1.3 Result Modal - Score between 7 - 12 points

Figure 7.1.4 Result Modal - Score > 12 points

If the user tries to calculate the score without attempting all the questions, then the website will prompt a message to complete all the questions before clicking on calculate button.

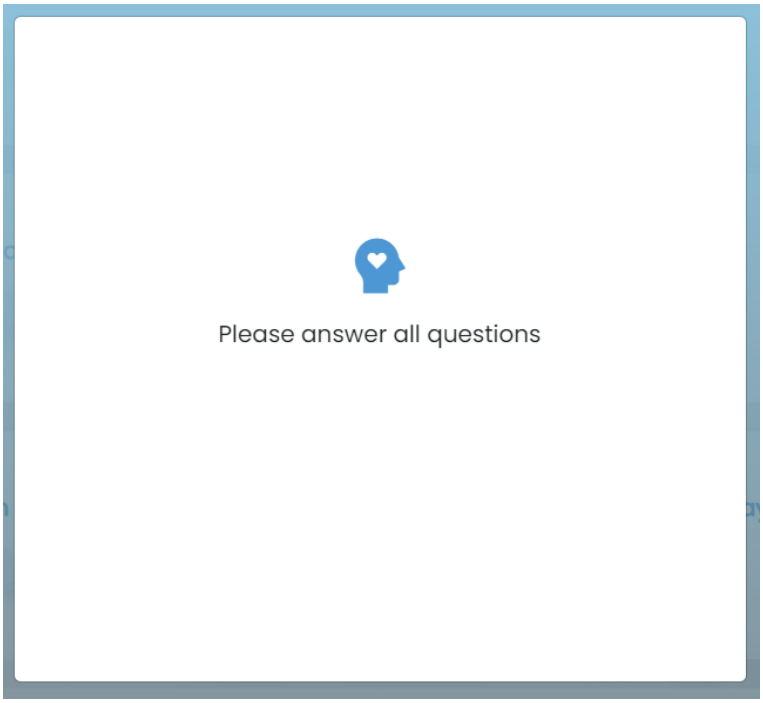

Figure 7.1.5 Complete All Questions - Modal

8. Edit Profile Feature

8.1 Feature Overview

'Edit Profile' feature lets users change their personal details like their first name, last name, mobile number, password, etc. Dalhousie Multifaith Services (DMS) provides a separate profile page where users can see their details and edit them.

8.1.1 Student Profile

Students need to provide their first name, last name, mobile number, institutional email, and password while registering on DMS. These details can be seen on their profile page.

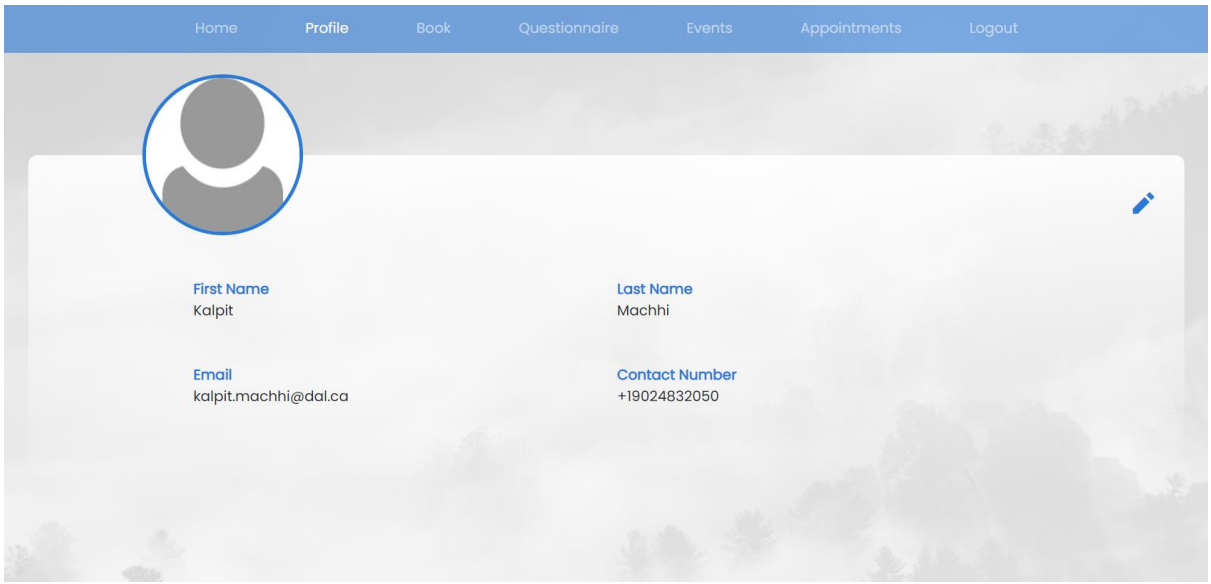

Figure 8.1.1.1 Student - Profile page

On clicking the edit icon, a modal will pop up, letting the users (students) edit their first name, last name, mobile number, and password. Edit profile modal for students can be seen in Figure 1.2.

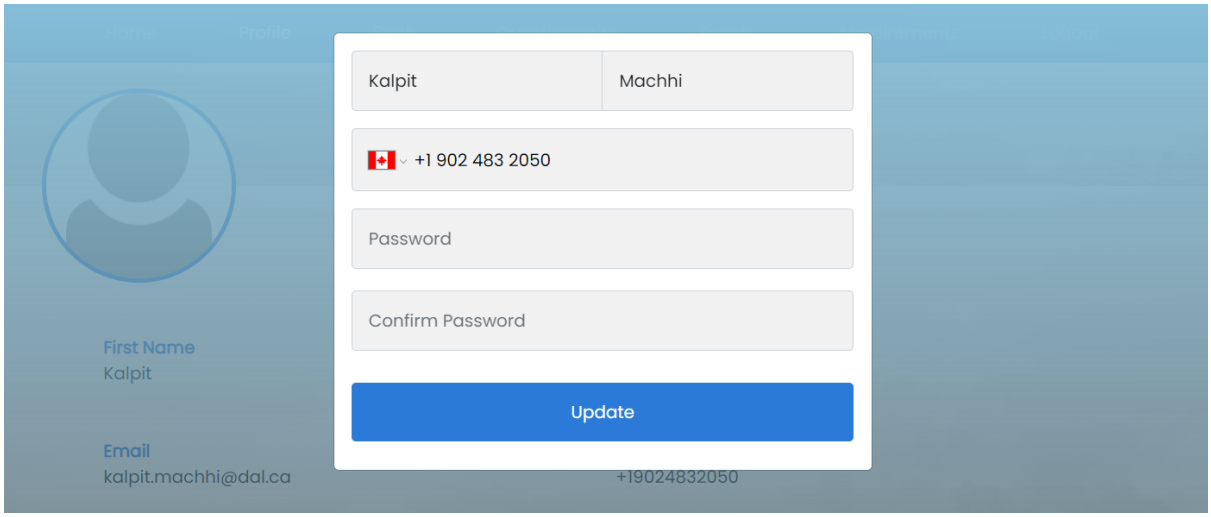

Figure 8.1.1.2 Student - Edit profile modal

Users can change any specific detail or all the details simultaneously. If users want to change their name or mobile number, they need to confirm the changes by providing their existing password. For updating the password, the users need to enter a new password and re-enter it for confirmation.

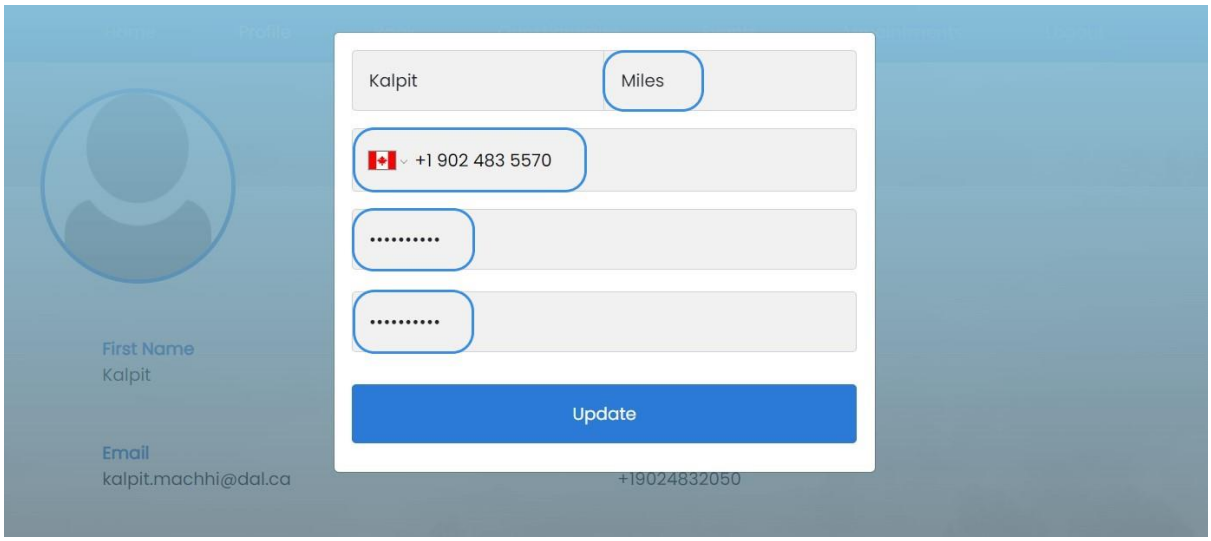

Figure 8.1.1.3 Student - Updating profile details

After making the required changes to the profile details, clicking on the update button will validate the new information and password. Once the validation is successful, it will update the information. Figure 1.4 shows the profile page after the update.

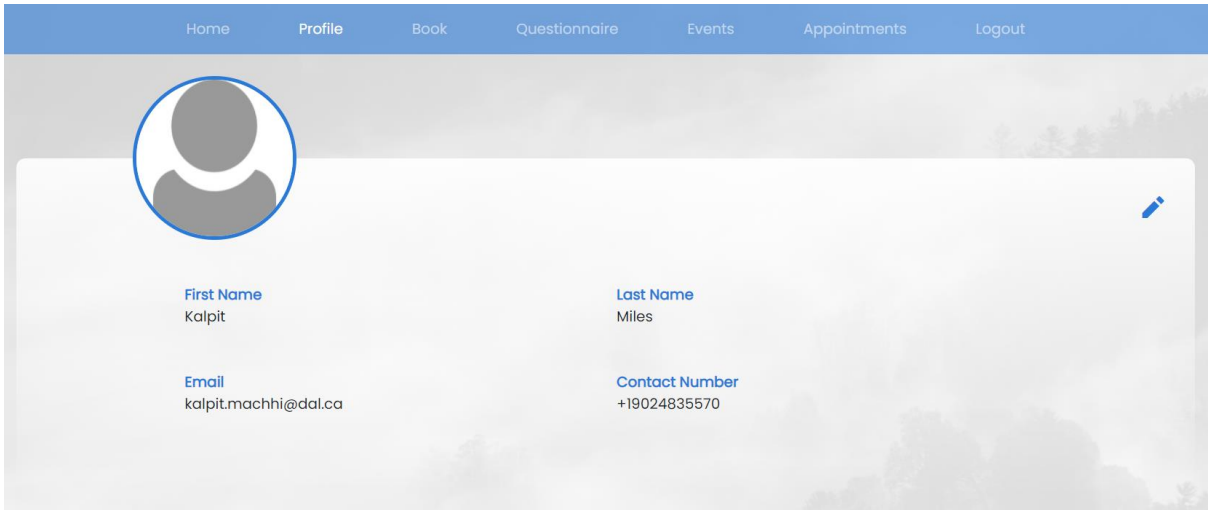

Figure 8.1.1.4 Student - Updated profile page

8.1.2 Chaplain Profile

Chaplains need to provide their first name, last name, mobile number, institutional email, Religion, Description, and password while registering on DMS. These details can be seen on their profile page.

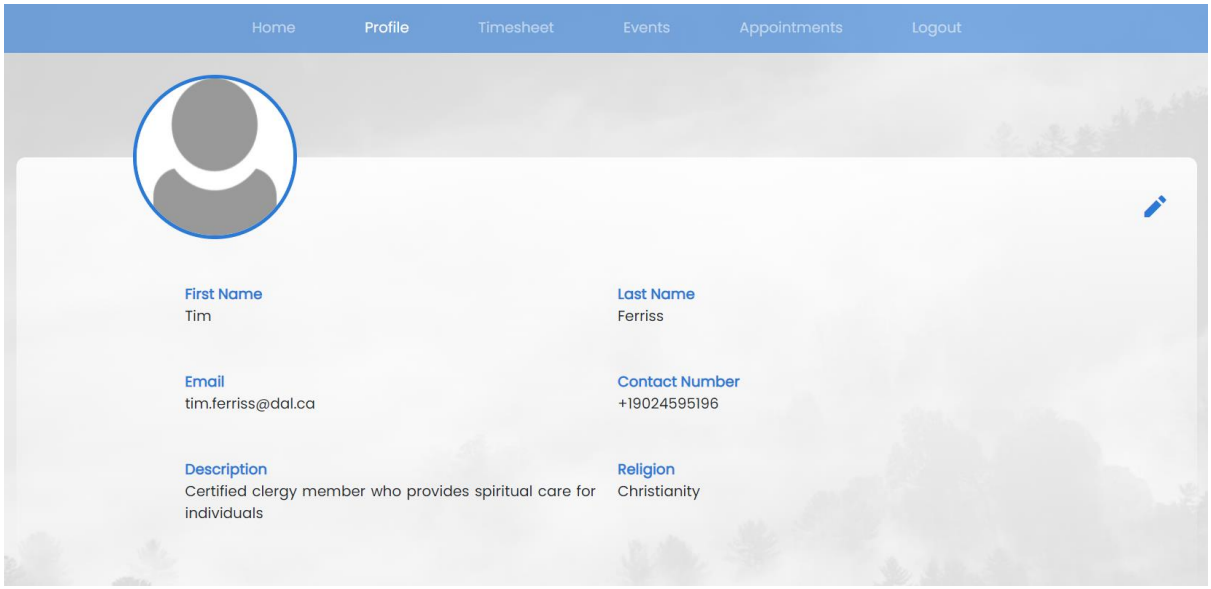

Figure 8.1.2.1 Chaplain - Profile page

On clicking the edit icon, a modal will pop up, letting the users (chaplains) edit their first name, last name, mobile number, religion, description, and password. Edit profile modal for chaplains can be seen in Figure 1.6.

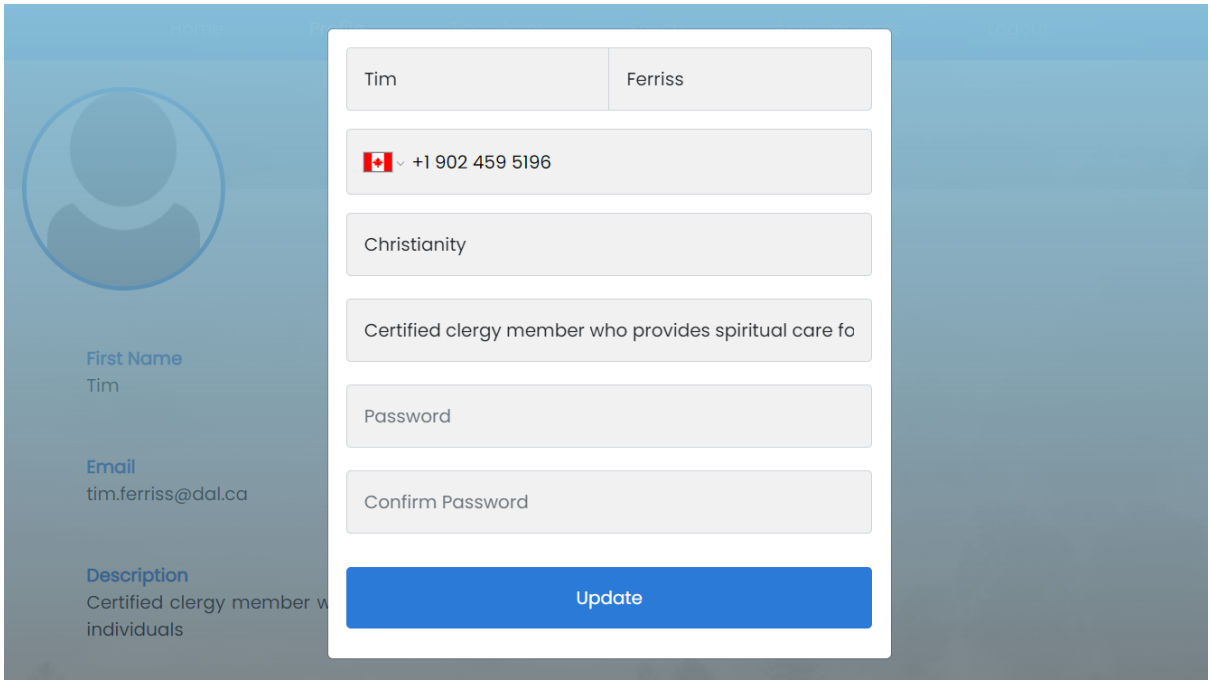

Figure 8.1.2.2 Chaplain - Edit profile modal

Like students, chaplains can also change any specific detail or all the details simultaneously. If chaplains want to change their name, religion, or mobile number, they need to confirm the changes by providing their existing password. For updating the password, the chaplains need to enter a new password and re-enter it for confirmation.

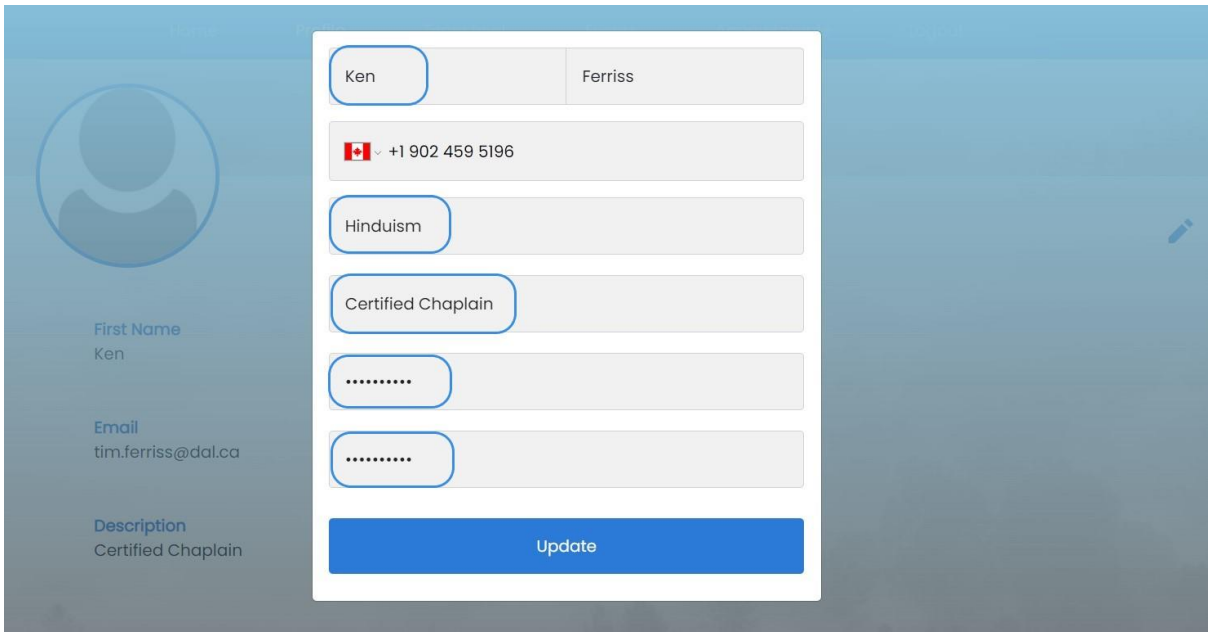

Figure 8.1.2.3 Chaplain - Updating profile details

After making the required changes to the profile details, clicking on the update button will validate the new information and password. Once the validation is successful, it will update the information. Figure 1.8 shows the profile page after the update.

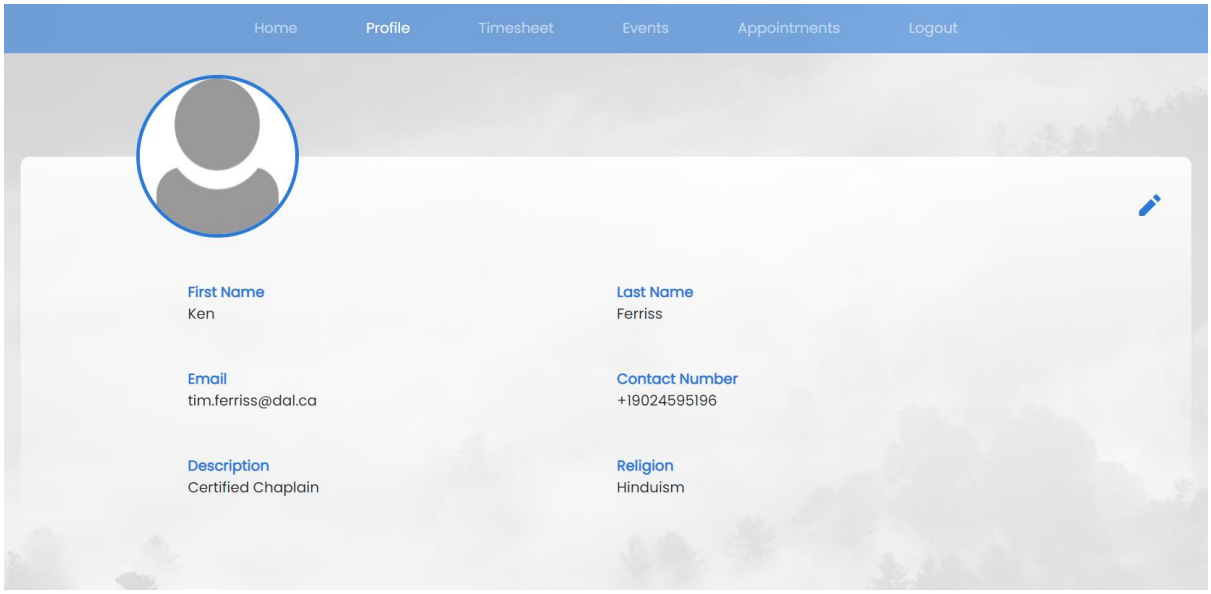

Figure 8.1.2.4 Chaplain - Updated profile page

8.2 Back-end Data Management

Figure 1.9 and Figure 1.10 show the models created to store the details of students and chaplains, respectively. Both models have a one-to-one relationship with the AUTH_USER_MODEL. The 'user' model has attributes like the first name, last name, mobile number, and email, typical for both models. The 'chaplain' model additionally has unique characteristics like phone, religion, and description.

```
class DalUser(models.Model):
    user = models.OneToOneField(
        settings.AUTH_USER_MODEL, on_delete=models.CASCADE
    )
    phone = models.CharField(max_length=255)

    def __str__(self) -> str:
        return f"{self.user.first_name} {self.user.last_name}"

    class Meta:
        ordering = ["user__first_name", "user__last_name"]
```

Figure 8.2.1 Student - models.py

```python
class Chaplain(models.Model):
    user = models.OneToOneField(
        settings.AUTH_USER_MODEL, on_delete=models.CASCADE
    )
    phone = models.CharField(max_length=255)
    religion = models.CharField(max_length=255)
    description = models.TextField()

    def __str__(self) -> str:
        return f"{self.user.first_name} {self.user.last_name}"

    class Meta:
        ordering = ["user__first_name", "user__last_name"]
```

Figure 8.2.2 Chaplain - models.py

To show the details stored in these models, two classes are created in the views.py file: DalUserDetail and ChaplainDetail. They both have two methods:

(1) get()
(2) put()

The get() method is used to get the profile details from the database and view it on the website. At the same time, the put() method is used to update the elements stored in the database using the data provided in the front end.

```python
class DalUserDetail(APIView):
    def get(self, request, id):
        dal_user = get_object_or_404(DalUser, user_id=id)
        serializer = DalUserSerializer(dal_user)
        return Response(serializer.data)

    def put(self, request, id):
        dal_user = get_object_or_404(DalUser, user_id=id)
        serializer = DalUserSerializer(dal_user, data=request.data)
        serializer.is_valid(raise_exception=True)
        serializer.save()
        return Response(serializer.data)
```

Figure 8.2.3 Student - views.py

```python
class ChaplainDetail(APIView):
    def get(self, request, id):
        dal_user = get_object_or_404(Chaplain, user_id=id)
        serializer = ChaplainSerializer(dal_user)
        return Response(serializer.data)

    def put(self, request, id):
        dal_user = get_object_or_404(Chaplain, user_id=id)
        serializer = ChaplainSerializer(dal_user, data=request.data)
        serializer.is_valid(raise_exception=True)
        serializer.save()
        return Response(serializer.data)
```

Figure 8.2.4 Chaplain - views.py