SCHULUNGS
ZENTRUM
FOHNSDORF

Projektarbeit - RedBlue Academy

Thema: Web Entwicklung / KMS 2

Author: Jaroslav Krc

Duration of the project work: 14.10.2024 - 22.11.2024

Trainer: Sabrina Müller

Im Auftrag und finanziert durch das AMS Arbeitsmarktservice Steiermark

Content

1 Introduction 4

1.1 Flowchart 7

1.2 Create .env and install requirements 8

2 Backend 10

2.1 Create Django project 10

2.2 Create startapp 10

2.3 Update settings.py 10

2.4 Create serializer 14

2.5 Create User view 15

2.6 Create Urls 16

2.7 Start Server 17

2.8 Create MySQL database 18

2.9 Create models Student hour, Reservation, Order and Control 20

2.10 Create two more serializers 24

2.11 Create another views 27

2.12 Create urls in api 38

2.13 Create superuser admin account 39

2.14 Login to Django administration 40

2.15 Create admin 40

3 Frontend 45

3.1 Create Frontend 45

3.2 Install packages 45

3.3 Delete css 46

3.4 Create new folders 47

3.5 Create new files 48

3.6 Create API 49

3.7 Create Authentication 51

3.8 Create Protected Route 61

3.9 Create page files 63

3.10 Rewrite App 64

3.11 Create Component (Footer.jsx) 65

3.12 Create Component (Layout.jsx) 67

3.13 Update App with Layout 66

3.14 Create first css 68

3.15 Create page Home with style 69

3.16 Create page Privacy Policy with style 73

3.17 Create page Terms of Service with style 74
3.18 Create page FAQ with style 75
3.19 Create page Not Found 77
3.20 Create page Price with styled 77
3.21 Update index.html 79
3.22 Create page Services with style 79
3.23 Create component Header with style 81
3.24 Update Layout and write there Header 85
3.25 Create page Calendar 85
3.26 Update App with new connection 86
3.27 Run servers and check pages 87
3.28 Create Component Form.jsx with style 88
3.29 Update Login and Register Page 92
3.30 Start backend and frontend server server and make test 93
3.31 Open Django Administrator 94
3.32 Install packeges for calendar 94
3.33 Create OrderPage 95
3.34 Create OrderPending 97
3.35 Update Calendar 98
4 GitHub 110
4.1 Create .gitignore 110
4.2 Create repository 111

1 Introduction

RedBlue Academy

RedBlue Academy is a comprehensive web application for managing lesson reservations, tracking study hours, and allowing users to order additional study hours. The platform provides a robust authentication system, integrates an interactive calendar, and allows admin management of orders and reservations. Built with a React frontend and a Django backend, this application offers a seamless experience for both users and administrators.

Table of Contents

- Key Features
- Folder Structure
- Environment Variables
- Usage
- API Endpoints
- Technologies Used

Key Features

1. **User Authentication**: Enables users to register, log in, and manage sessions securely with JWT tokens.
2. **Order Management**: Users can request additional study hours, which must be approved by an admin.
3. **Interactive Calendar**: Displays lesson reservations with color-coded statuses:
   - Green for approved lessons
   - Orange for pending lessons
   - Red for rejected lessons
4. **Admin Controls**: Admins can view and manage orders, approve/reject them, and adjust user study hours.
5. **Protected Routes**: Restricts access to certain pages based on user authentication and order status.
6. **Session Tracking**: Tracks active user sessions and refreshes access tokens periodically.

Folder Structure

The project is divided into two main parts: the `backend` (Django) and the `frontend` (React). Below is a detailed look at the folder structure:

Backend (Django)

- **backend/**: Contains the main Django project files and settings.
  - `settings.py`: Configures Django settings, including database, installed apps, middleware, etc.
  - `urls.py`: Main URL configurations for the Django project.
  - `wsgi.py` & `asgi.py`: Web server gateways for deploying the application.
- **api/**: This Django app within the project handles the core logic for user management, orders, reservations, and more.

- `admin.py`: Customizes the Django admin panel for managing orders and reservations with additional actions like approval.
- `apps.py`: Django app configuration for `api`.
- `models.py`: Defines the database models, including `Order`, `Reservation`, `UserProfile`, and `ActiveUser`.
- `serializers.py`: Serializes model data for API responses, ensuring controlled data exposure.
- `views.py`: Defines API views that handle user requests for managing orders, reservations, and other functionality.
- `urls.py`: API-specific URL routing, connecting view functions with specific endpoints.
- `tests.py`: Contains test cases to ensure API endpoints and functionalities work as expected.

Frontend (React)

- **frontend/src**: Contains the main codebase for the frontend, including components, pages, styling, and API configuration.
  - **components/**: Reusable components used across different pages.
    - `AuthContext.jsx`: Manages user authentication state, token handling, and provides context to other components.
    - `Footer.jsx`: Footer component displayed across all pages.
    - `Form.jsx`: A form component for handling login and registration.
    - `Header.jsx`: Header with navigation links and user options.
    - `Layout.jsx`: Layout component for consistent styling and structure.
    - `OrderPending.jsx`: Displays a message indicating an order is pending approval.
    - `ProtectedRoute.jsx`: Controls access to certain routes based on user authentication and order status.
  - **pages/**: Main pages of the application.
    - `Calendar.jsx`: Page that displays the interactive calendar for managing reservations, with color-coded statuses.
    - `CustomSolutions.jsx`: Custom solutions page (details not provided in the current README).
    - `Faq.jsx`: FAQ page for frequently asked questions.
    - `Home.jsx`: Home page for the application, introducing RedBlue Academy.
    - `Login.jsx`: Login page for user authentication.
    - `NotFound.jsx`: 404 page displayed when a route is not found.
    - `OrderPage.jsx`: Allows users to request additional study hours.
    - `PriceList.jsx`: Displays the price list for services.
    - `PrivacyPolicy.jsx`: Privacy policy page for GDPR compliance.
    - `Register.jsx`: Registration page for new users.
    - `Services.jsx`: Services overview page.
    - `TermsOfService.jsx`: Terms of Service page.
  - **styles/**: Contains CSS files for each page and component, allowing for modular styling.
    - Each file corresponds to a component or page, such as `Calendar.css` for the calendar styling and `Header.css` for the header.
  - **assets/**: This folder (not detailed in the images) would typically contain static files like images, fonts, and icons used across the application.
  - **api.js**: Configures Axios to handle HTTP requests to the Django backend.

- **constants.js**: Defines constants, including token names used for localStorage management.

- **public/**: Contains the HTML template and any static assets required at the root level of the frontend application.

Usage

1. **Register/Login**: Users can register and log in to access their profile and schedule lessons.
2. **Order Study Hours**: On the `OrderPage`, users can request additional study hours, which need to be approved by an admin.
3. **View Calendar**: Users can view and manage their reservations in the calendar. Approved lessons are green, pending lessons are orange, and rejected lessons are red.
4. **Admin Panel**: Admins can log in to the Django admin panel (`/admin`) to approve or reject orders and manage study hours.

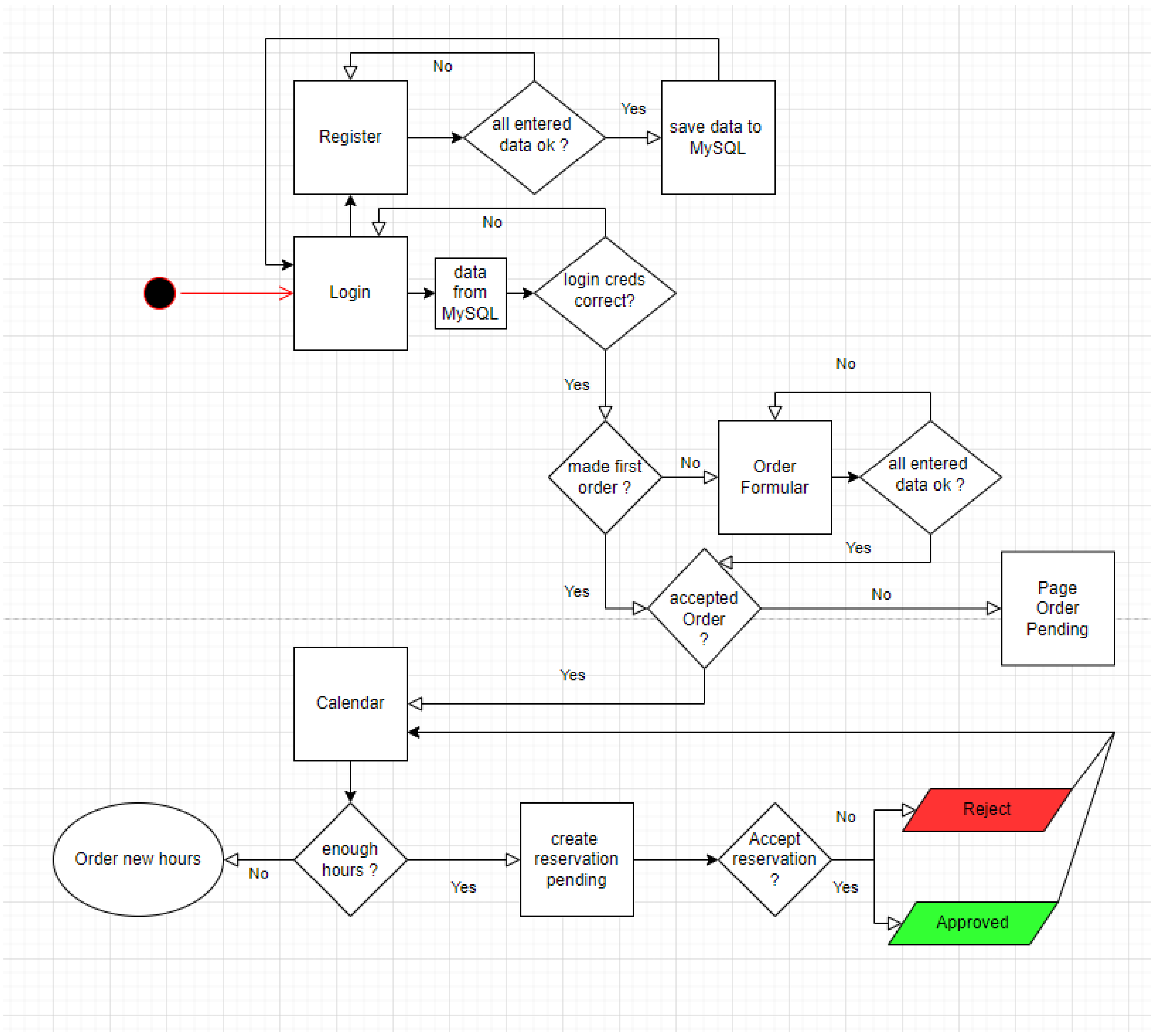
API Endpoints

Endpoint	Method	Description
`/api/user/login/track/`	POST	Track user login session
`/api/user/study_hours/`	GET	Retrieve available study hours for user
`/api/order/create/`	POST	Create a new order for study hours
`/api/reservations/`	GET	List reservations with status
`/api/reservation/create/`	POST	Create a reservation
`/api/reservation/<pk>/`	DELETE	Delete a pending reservation
`/api/reservations/hide_rejected/`	POST	Hide rejected reservations

Technologies Used

- **Backend**:
  - Django: Web framework for backend logic and API.
  - Django REST Framework: For building the API.
  - MySQL: database
- **Frontend**:
  - React: JavaScript library for building the user interface.
  - FullCalendar: Calendar component for managing reservations.
  - Axios: For making HTTP requests to the backend.
  - Bootstrap: Styling framework for responsive design.

1.1 Flowchart



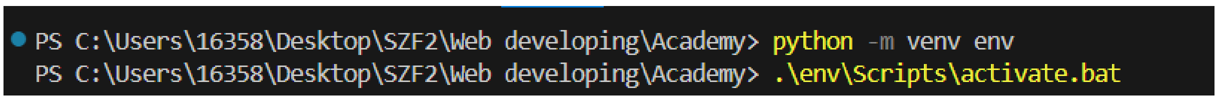
1.2 Create .env and install requirements

First command: python -m venv env

- This command creates a new virtual environment named "env" in the current directory. A virtual environment is an isolated environment that allows you to install and manage dependencies for a project independently of the system-wide Python libraries.

Second command: .env\Scripts\activate.bat

- This command activates the virtual environment "env." After activation, any installed Python packages and dependencies will be stored and used only within this virtual environment, without affecting other projects or the system-wide Python setup.



```
PS C:\Users\16358\Desktop\SZF2\Web developing\Academy> python -m venv env
PS C:\Users\16358\Desktop\SZF2\Web developing\Academy> .\env\Scripts\activate.bat
```

Creating a requirements.txt file to list the libraries needed for the project

asgiref
The asgiref library provides a reference implementation for ASGI (Asynchronous Server Gateway Interface), which is necessary for asynchronous support in Django. It enables efficient handling of asynchronous requests, useful for applications that require a higher degree of parallelism.

Django
Django is the main web framework for the project. It is a high-level Python framework that promotes rapid development and clean, pragmatic design. Django offers many built-in tools, including an ORM for database operations, authentication, and an admin interface, simplifying the development of complex web applications.

django-cors-headers
django-cors-headers provides support for Cross-Origin Resource Sharing (CORS) in Django. CORS is essential when the frontend application (e.g., a React app) runs on a different domain than the backend. This library allows control over which external domains have access to the server, enhancing security during cross-origin communication.

djangorestframework
djangorestframework is a toolkit for building RESTful APIs in Django. This package includes tools for creating API endpoints, managing authentication, and serializing data. It is used to create a flexible and extensible API for accessing the application's data.

djangorestframework-simplejwt
The djangorestframework-simplejwt library adds support for JWT (JSON Web Token)

authentication in Django REST Framework. JWT authentication is commonly used to secure API endpoints. This library enables the generation, validation, and management of JWT tokens, ensuring secure communication between the frontend and backend.

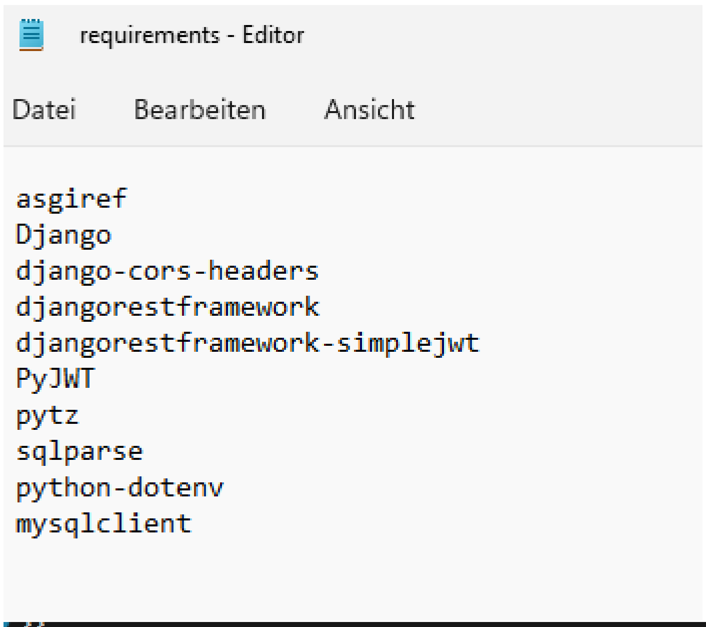
PyJWT
PyJWT is a tool for working with JSON Web Tokens in Python. It is used for encoding and decoding JWT tokens and is a dependency of djangorestframework-simplejwt. This library provides secure token management for user authentication and authorization.

pytz
pytz is a timezone library used for proper timezone support in Django. It allows the application to handle time-related data across different time zones, which is crucial for global applications.

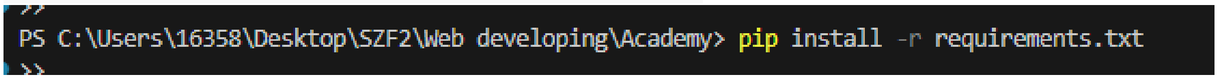
sqlparse
sqlparse is a non-validating SQL parser for Python, used by Django to format SQL queries. This library is primarily necessary for Django's ORM and admin interface, ensuring better readability of SQL queries.

python-dotenv
python-dotenv is a library that enables loading environment variables from a .env file. This approach helps to securely store sensitive information (such as database credentials or API keys) outside of the source code, enhancing the application's security.

mysqlclient
mysqlclient is a MySQL database adapter for Python that allows Django to interact with a MySQL database. It is used to configure MySQL as the database backend for the Django project, providing a reliable data storage solution for the application.



```
asgiref
Django
django-cors-headers
djangorestframework
djangorestframework-simplejwt
PyJWT
pytz
sqlparse
python-dotenv
mysqlclient
```



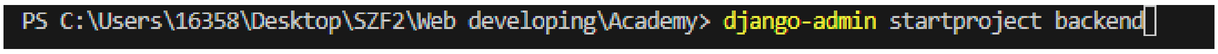
```
>>
PS C:\Users\16358\Desktop\SZF2\Web developing\Academy> pip install -r requirements.txt
>>
```

2 Backend

2.1 Create Django project

command: django-admin startproject backend

- This command initializes a new Django project named "backend" in the current directory. It creates the basic structure of the project, including a folder with the same name (backend) and essential files like settings.py, urls.py, and wsgi.py, which are necessary for configuring and running the Django project.

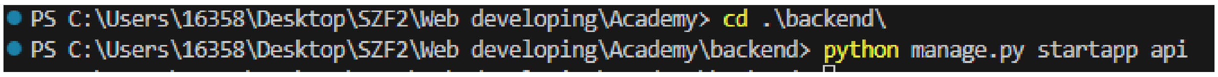


```
PS C:\Users\16358\Desktop\SZF2\Web developing\Academy> django-admin startproject backend
```

2.2 Create startapp

command: python manage.py startapp api

- This command creates a new Django app named "api" within the existing Django project (currently in the "backend" directory).



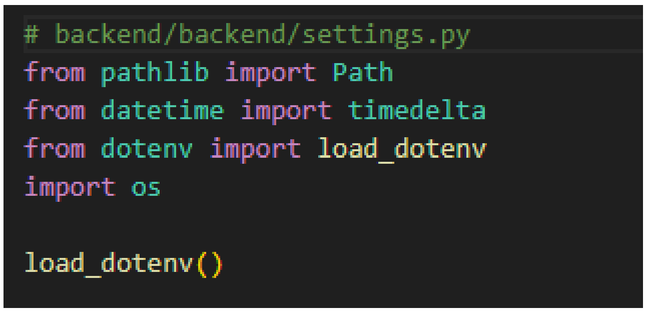
```
PS C:\Users\16358\Desktop\SZF2\Web developing\Academy> cd .\backend\
PS C:\Users\16358\Desktop\SZF2\Web developing\Academy\backend> python manage.py startapp api
```

2.3 Update settings.py

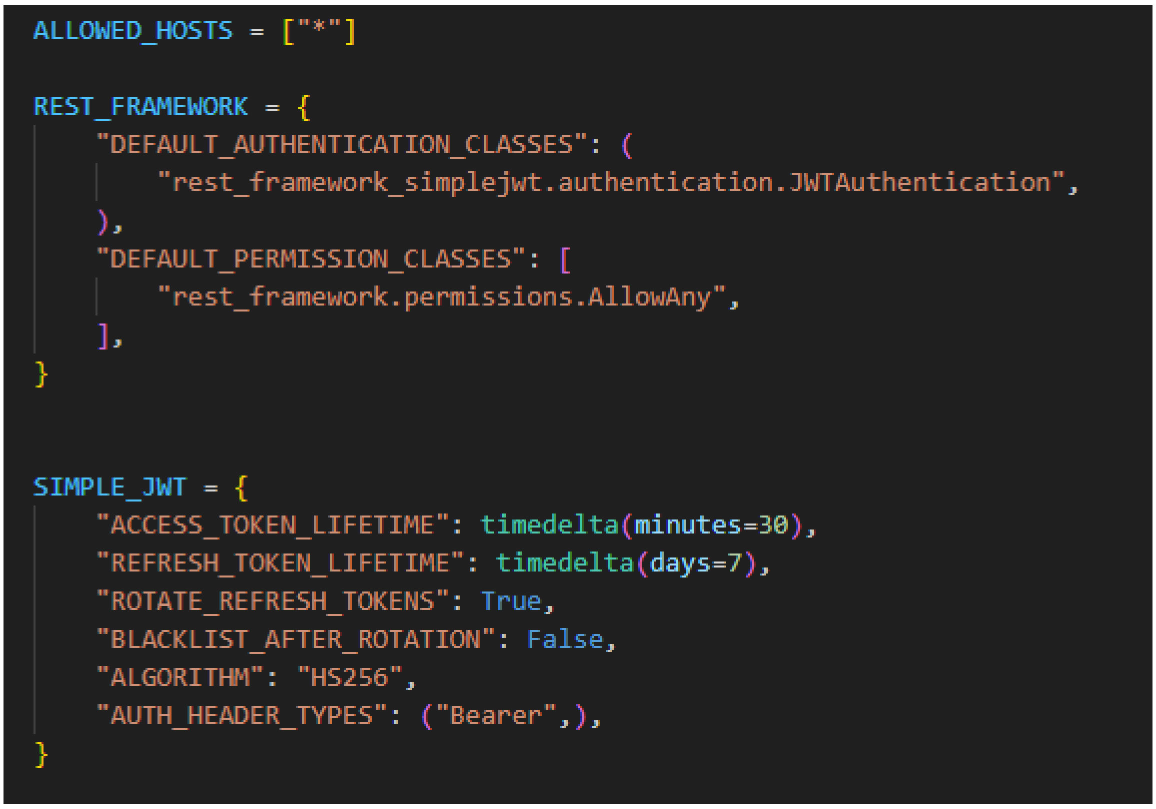
Add new libraries to settings.py, configure settings for tokens, set up the REST Framework, and include the used applications and middleware.

Loading Environment Variables

This code imports the load_dotenv function from the dotenv library and calls it. This function loads environment variables from a .env file into the application's environment. This is useful for securely storing sensitive information, such as API keys and database credentials, outside of the main codebase.



```
# backend/backend/settings.py
from pathlib import Path
from datetime import timedelta
from dotenv import load_dotenv
import os

load_dotenv()
```



```
ALLOWED_HOSTS = ["*"]

REST_FRAMEWORK = {
    "DEFAULT_AUTHENTICATION_CLASSES": (
        "rest_framework_simplejwt.authentication.JWTAuthentication",
    ),
    "DEFAULT_PERMISSION_CLASSES": [
        "rest_framework.permissions.AllowAny",
    ],
}


SIMPLE_JWT = {
    "ACCESS_TOKEN_LIFETIME": timedelta(minutes=30),
    "REFRESH_TOKEN_LIFETIME": timedelta(days=7),
    "ROTATE_REFRESH_TOKENS": True,
    "BLACKLIST_AFTER_ROTATION": False,
    "ALGORITHM": "HS256",
    "AUTH_HEADER_TYPES": ("Bearer",),
}
```

ALLOWED_HOSTS is a Django setting that specifies which hosts (domain names) are allowed to make requests to the Django server. The asterisk ("*"), in this case, allows requests from any host. In production, this setting should be restricted to specific domains for security reasons.

DEFAULT_AUTHENTICATION_CLASSES: This setting specifies the authentication methods to use for the Django REST Framework. In this case, it uses JWTAuthentication provided by djangorestframework-simplejwt for token-based authentication.

DEFAULT_PERMISSION_CLASSES: This setting defines the default permissions for accessing the API. AllowAny means that any user, regardless of authentication status, can access the API. In production, you might want to restrict access, such as using IsAuthenticated to require user authentication.

ACCESS_TOKEN_LIFETIME: Defines the lifespan of the access token, which is set to expire after 30 minutes. After expiration, the user will need a new token to access the API.

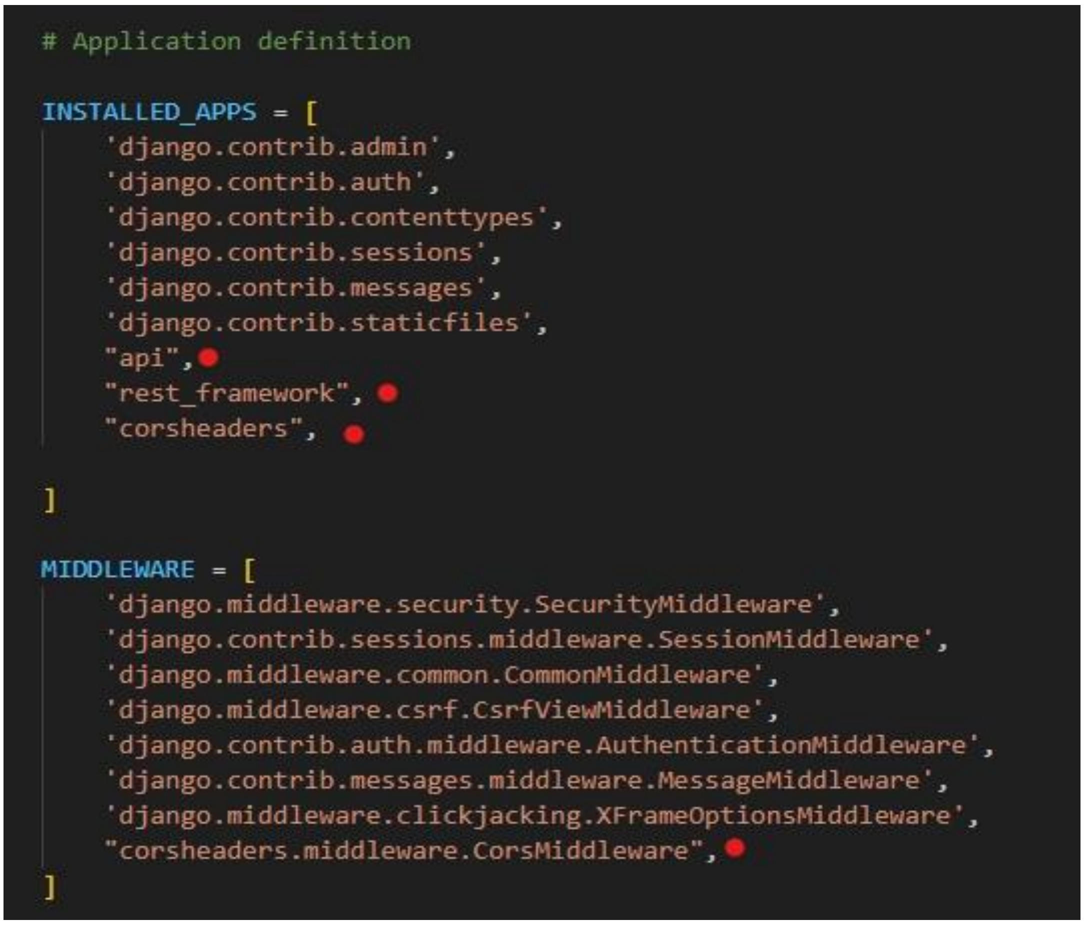
REFRESH_TOKEN_LIFETIME: Defines the lifespan of the refresh token, which is set to expire after 7 days. The refresh token allows the user to obtain a new access token without logging in again.

ROTATE_REFRESH_TOKENS: When set to True, this setting allows a new refresh token to be issued each time an access token is refreshed, improving security by rotating tokens.

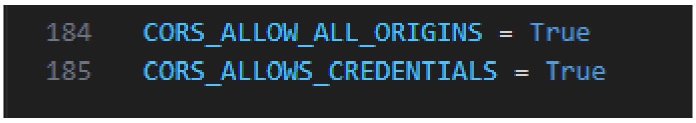
BLACKLIST_AFTER_ROTATION: This setting, when set to True, would blacklist old refresh tokens after rotation. It's set to False here, meaning old tokens will still be valid until they naturally expire.

ALGORITHM: Specifies the algorithm used to sign the JWT tokens. HS256 is a secure algorithm for token signing using a secret key.

AUTH_HEADER_TYPES: Defines the type of authentication header to expect. Here, it's set to Bearer, meaning tokens should be sent in the form of Authorization: Bearer <token> in the HTTP header.



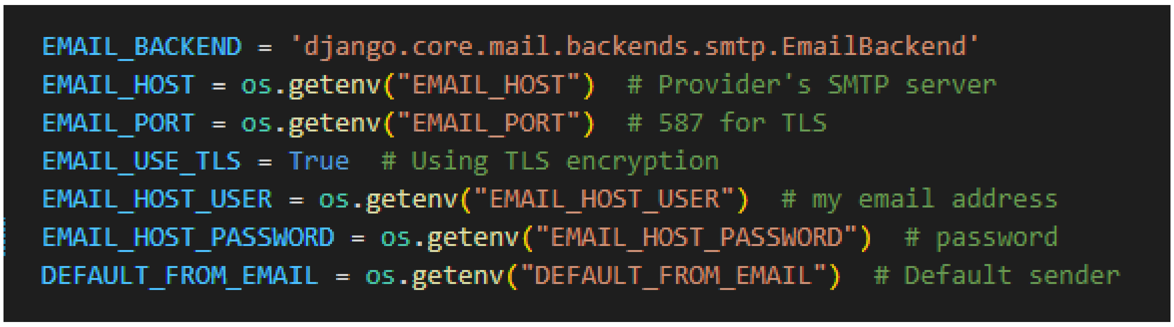
```
# Application definition

INSTALLED_APPS = [
    'django.contrib.admin',
    'django.contrib.auth',
    'django.contrib.contenttypes',
    'django.contrib.sessions',
    'django.contrib.messages',
    'django.contrib.staticfiles',
    "api",
    "rest_framework",
    "corsheaders",

]

MIDDLEWARE = [
    'django.middleware.security.SecurityMiddleware',
    'django.contrib.sessions.middleware.SessionMiddleware',
    'django.middleware.common.CommonMiddleware',
    'django.middleware.csrf.CsrfViewMiddleware',
    'django.contrib.auth.middleware.AuthenticationMiddleware',
    'django.contrib.messages.middleware.MessageMiddleware',
    'django.middleware.clickjacking.XFrameOptionsMiddleware',
    "corsheaders.middleware.CorsMiddleware",
]
```



```
CORS_ALLOW_ALL_ORIGINS = True
CORS_ALLOWS_CREDENTIALS = True
```

CORS_ALLOW_ALL_ORIGINS: When set to True, this setting allows requests from any origin (domain). This means the server will permit access from all domains attempting to communicate with the backend. In a production environment, it's safer to restrict access to specific domains to reduce security risks.

CORS_ALLOW_CREDENTIALS: When set to True, this setting allows authentication credentials (such as cookies, tokens) to be sent along with requests from different domains. This is useful for authentication purposes, where user credentials need to be passed with requests.



```python
EMAIL_BACKEND = 'django.core.mail.backends.smtp.EmailBackend'
EMAIL_HOST = os.getenv("EMAIL_HOST")  # Provider's SMTP server
EMAIL_PORT = os.getenv("EMAIL_PORT")  # 587 for TLS
EMAIL_USE_TLS = True  # Using TLS encryption
EMAIL_HOST_USER = os.getenv("EMAIL_HOST_USER")  # my email address
EMAIL_HOST_PASSWORD = os.getenv("EMAIL_HOST_PASSWORD")  # password
DEFAULT_FROM_EMAIL = os.getenv("DEFAULT_FROM_EMAIL")  # Default sender
```

EMAIL_BACKEND: Specifies the backend to be used for sending emails in Django. In this case, the `django.core.mail.backends.smtp.EmailBackend` is used, which enables email delivery through an SMTP server.

EMAIL_HOST: Defines the SMTP server's hostname or IP address. The value is fetched from the environment variable `EMAIL_HOST`, ensuring flexibility and security by not hardcoding the value.

EMAIL_PORT: Specifies the port number used by the SMTP server. For TLS encryption, port `587` is commonly used. The value is fetched from the `EMAIL_PORT` environment variable.

EMAIL_USE_TLS: Enables TLS (Transport Layer Security) encryption for secure communication with the SMTP server. This ensures that email messages and credentials are encrypted during transmission.

EMAIL_HOST_USER: Specifies the email address used to authenticate with the SMTP server. This value is fetched from the `EMAIL_HOST_USER` environment variable, ensuring sensitive information is not directly written in the code.

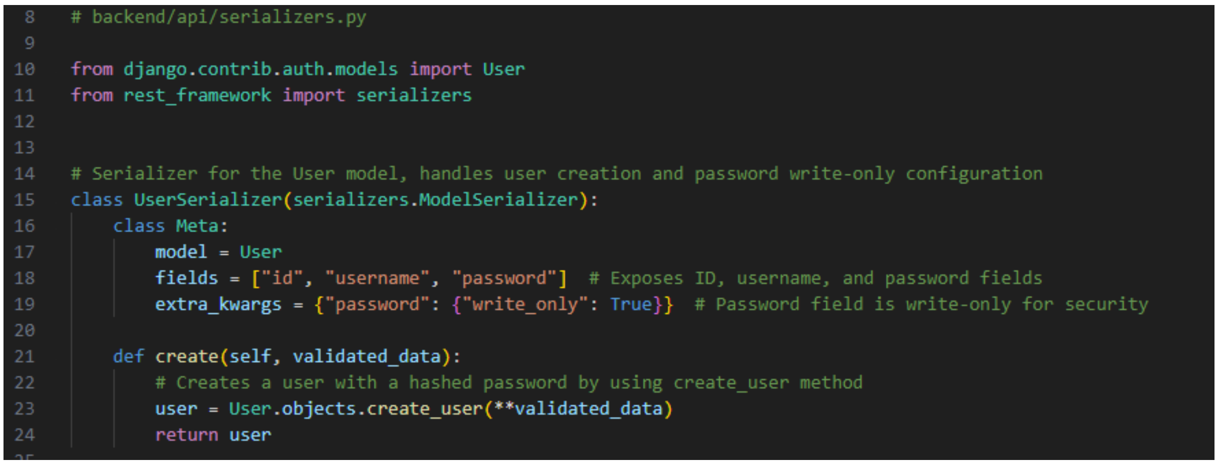
EMAIL_HOST_PASSWORD: Defines the password for the `EMAIL_HOST_USER` account. It is retrieved from the environment variable `EMAIL_HOST_PASSWORD`, maintaining security and preventing exposure in the codebase.

DEFAULT_FROM_EMAIL: Sets the default email address to appear in the "From" field of outgoing emails. This value is fetched from the `DEFAULT_FROM_EMAIL` environment variable, allowing customization without code changes.

2.4 Create serializer

Create first serializer UserSerializer.

This serializer handles the creation of a `User` instance with secure password handling. The password field is set as write-only for security, and the `create` method ensures that the password is hashed before saving the user to the database.



```python
# backend/api/serializers.py

from django.contrib.auth.models import User
from rest_framework import serializers


# Serializer for the User model, handles user creation and password write-only configuration
class UserSerializer(serializers.ModelSerializer):
    class Meta:
        model = User
        fields = ["id", "username", "password"]  # Exposes ID, username, and password fields
        extra_kwargs = {"password": {"write_only": True}}  # Password field is write-only for security

    def create(self, validated_data):
        # Creates a user with a hashed password by using create_user method
        user = User.objects.create_user(**validated_data)
        return user
```

Imports:

- Imports the User model from Django's built-in authentication system.
- Imports the serializers module from Django REST Framework to create and manage serializers.

UserSerializer Class:

- Defines a serializer class UserSerializer that inherits from serializers.ModelSerializer, a convenient way to create serializers for Django models.

Meta Class:

- **model**: Specifies that this serializer is for the `User` model.
- **fields**: Specifies the fields to include in the serialized data (`id`, `username`, and `password`).
- **extra_kwargs**: Sets additional options for the fields. Here, `password` is marked as `"write_only": True`, meaning it will only be used for writing (creating or updating a user) and will not be exposed in the serialized output for security reasons.

create Method:

- **Purpose**: This method overrides the default `create` method in `ModelSerializer` to handle password hashing securely.
- **Functionality**: Uses `User.objects.create_user()` instead of the default `create()` method to automatically hash the password before saving it to the database. This is important because passwords should not be stored as plain text.
- **Return**: Returns the newly created `user` instance

2.5 Create User view

This view allows any user to create a new account. It uses `UserSerializer` to handle the data validation and creation of new user instances, while `AllowAny` permissions ensure that the registration endpoint is open to everyone. The use of `CreateAPIView` simplifies the process, as it comes with built-in functionality for handling POST requests to create objects.

```python
# backend/api/views.py

from django.contrib.auth.models import User
from rest_framework import generics
from .serializers import UserSerializer
from rest_framework.permissions import IsAuthenticated, AllowAny

# Class-based view for creating a new user
class CreateUserView(generics.CreateAPIView):
    queryset = User.objects.all()
    serializer_class = UserSerializer
    permission_classes = [AllowAny]  # Allows any user to access this endpoint for registration
```

Imports:

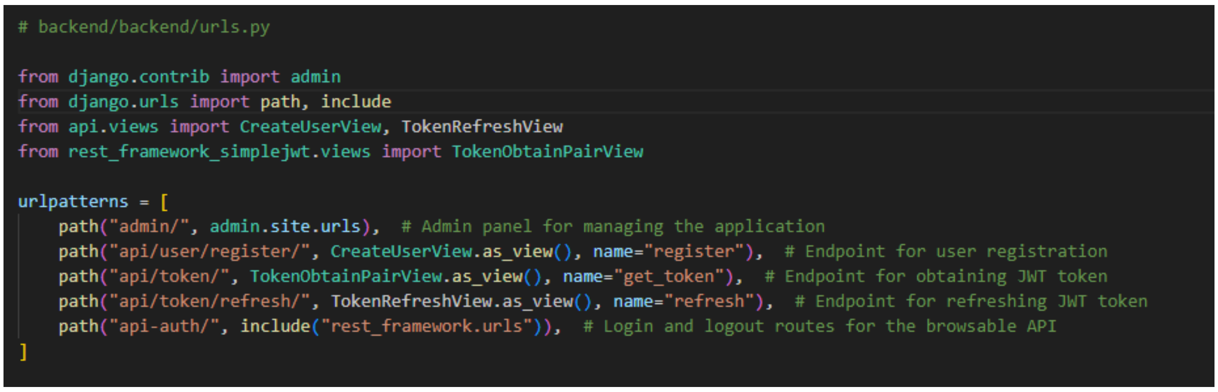
- **User**: The `User` model from Django's authentication system, representing users in the database.
- **generics**: Provides generic views from Django REST Framework, which simplify the process of creating common views like Create, Retrieve, Update, and Delete.
- **UserSerializer**: The custom serializer defined for the `User` model, responsible for handling user creation and password security.
- **IsAuthenticated** and **AllowAny**: Permission classes used to control access to views. `IsAuthenticated` requires authentication for access, while `AllowAny` allows unrestricted access.

CreateUserView Class:

- **Class Definition**: `CreateUserView` inherits from `generics.CreateAPIView`, which is a built-in view class that provides the functionality for creating new instances in the database (in this case, creating a new user).
- **queryset**: Defines the queryset used by the view, which is `User.objects.all()`. This specifies that the view will work with all user instances in the database.
- **serializer_class**: Specifies the serializer to use for this view, which is `UserSerializer`. This serializer is responsible for validating input data and creating new user instances.
- **permission_classes**: Sets the permissions for accessing this view. Here, `AllowAny` is used, meaning any user, authenticated or not, can access this endpoint. This is common for registration views, as users need to be able to create accounts without being logged in.

2.6 Create Urls

Configures the URL patterns for a Django project, including endpoints for user registration, obtaining and refreshing JSON Web Tokens (JWT), and the Django admin panel.



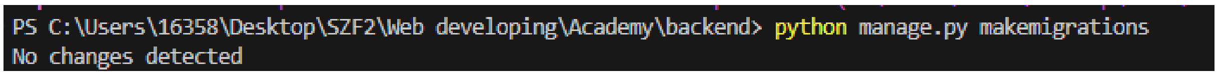
```python
# backend/backend/urls.py

from django.contrib import admin
from django.urls import path, include
from api.views import CreateUserView, TokenRefreshView
from rest_framework_simplejwt.views import TokenObtainPairView

urlpatterns = [
    path("admin/", admin.site.urls),  # Admin panel for managing the application
    path("api/user/register/", CreateUserView.as_view(), name="register"),  # Endpoint for user registration
    path("api/token/", TokenObtainPairView.as_view(), name="get_token"),  # Endpoint for obtaining JWT token
    path("api/token/refresh/", TokenRefreshView.as_view(), name="refresh"),  # Endpoint for refreshing JWT token
    path("api-auth/", include("rest_framework.urls")),  # Login and logout routes for the browsable API
]
```

Imports:

- **admin**: Imports Django's admin site functionality.
- **path** and **include**: Imports URL path functions to define URL patterns.
- **CreateUserView** and **TokenRefreshView**: Imports views from the `api` app. `CreateUserView` is a custom view for user registration, while `TokenRefreshView` handles refreshing JWT tokens.
- **TokenObtainPairView**: A view from `rest_framework_simplejwt` that provides a token upon user login. This token is used for user authentication in the API.

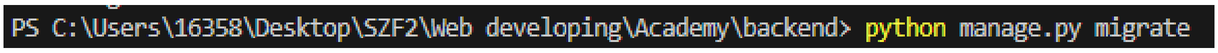
URL Patterns:

- **admin/**: Provides access to the Django admin panel. Visiting `/admin/` in the browser will bring up the admin interface for managing users, models, and other application data.
- **api/user/register/**: Points to `CreateUserView`, allowing new users to register by sending a request to this endpoint. The view will validate and create a new user in the system.
- **api/token/**: Points to `TokenObtainPairView`, which generates a new JWT token upon a successful login. This token can then be used to access protected API endpoints.
- **api/token/refresh/**: Points to `TokenRefreshView`, which allows users to refresh their JWT token before it expires, ensuring they stay authenticated without needing to log in again.
- **api-auth/**: Includes default login and logout routes provided by Django REST Framework for the browsable API interface, useful during development for testing API endpoints.



```
PS C:\Users\16358\Desktop\SZF2\Web developing\Academy\backend> python manage.py makemigrations
No changes detected
```

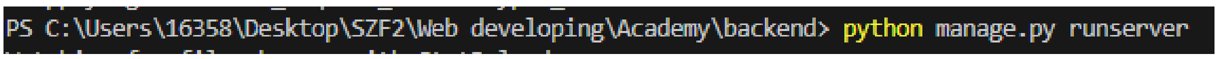
This command generates new migration files based on changes detected in Django models.



```
PS C:\Users\16358\Desktop\SZF2\Web developing\Academy\backend> python manage.py migrate
```

This command applies all available migrations to the database. It ensures that the database structure is synchronized with the current model definitions in the project.

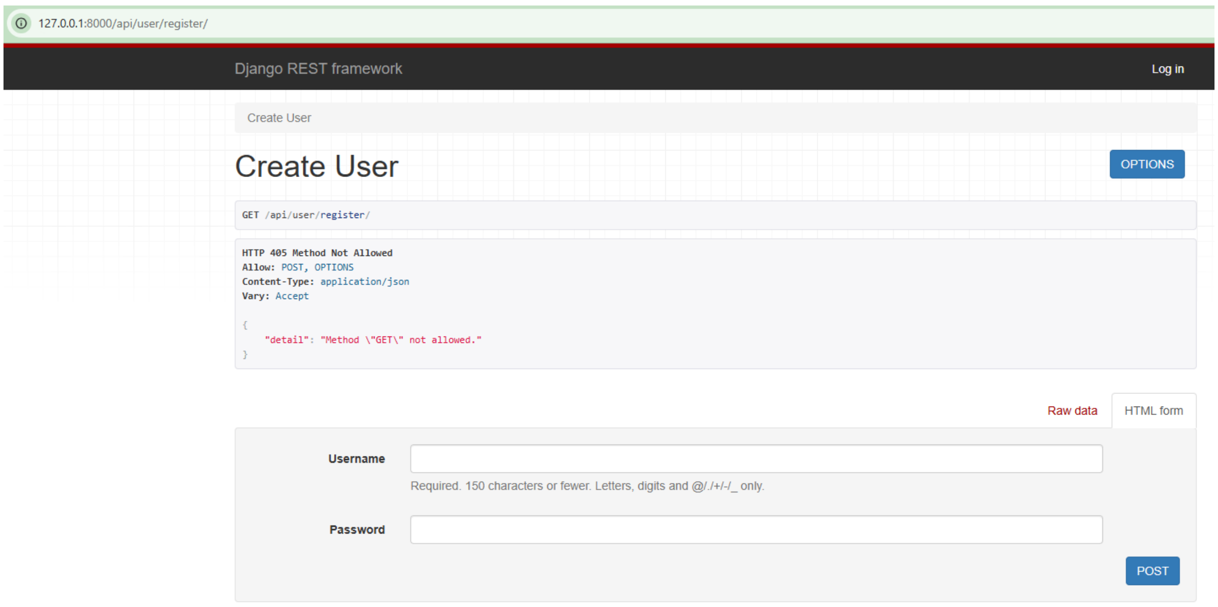
2.7 Start Server



```
PS C:\Users\16358\Desktop\SZF2\Web developing\Academy\backend> python manage.py runserver
```

This command starts Django’s built-in development server

Create first user and check if tokens works



127.0.0.1:8000/api/user/register/

Django REST framework

Log in

Create User

Create User

OPTIONS

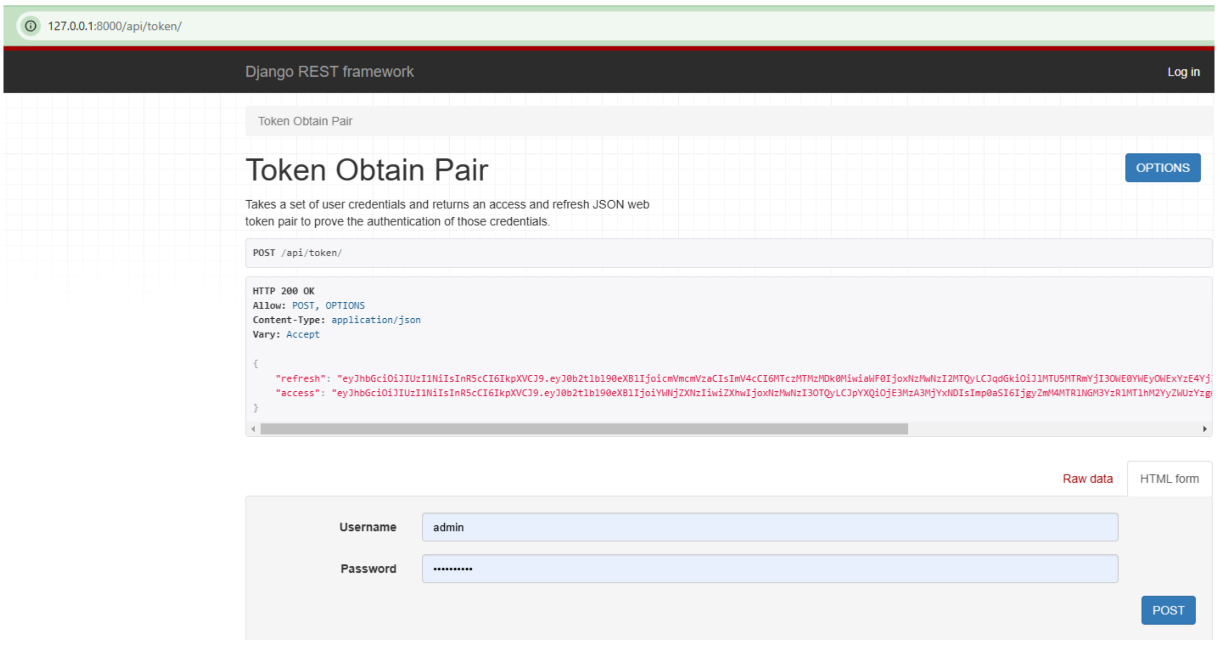
```
GET /api/user/register/
```

```
HTTP 405 Method Not Allowed
Allow: POST, OPTIONS
Content-Type: application/json
Vary: Accept

{
    "detail": "Method \"GET\" not allowed."
}
```

Raw data | HTML form

Username	
	Required. 150 characters or fewer. Letters, digits and @/./+/-/_ only.
Password	

POST



127.0.0.1:8000/api/token/

Django REST framework

Log in

Token Obtain Pair

Token Obtain Pair

OPTIONS

Takes a set of user credentials and returns an access and refresh JSON web token pair to prove the authentication of those credentials.

```
POST /api/token/
```

```
HTTP 200 OK
Allow: POST, OPTIONS
Content-Type: application/json
Vary: Accept

{
    "refresh": "eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJ0b2tlbl90eXBlIjoicmVmcmVzaCIsImV4cCI6MTczMTMzMDk0MiwiaWF0IjoxNzMwNzI2MTQyLCJqdGkiOiJ1MTU5MTRmYjI3OWE0YWEyOWExYzE4Yj
    "access": "eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJ0b2tlbl90eXBlIjoiYWNjZXNzIiwiZXhwIjoxNzMwNzI3OTQyLCJpYXQiOjE3MzA3MjYxNDIsImp0aSI6IjgyZmM4MTR1NGM3YzR1MTlhM2YyZWUzYzg
}
```

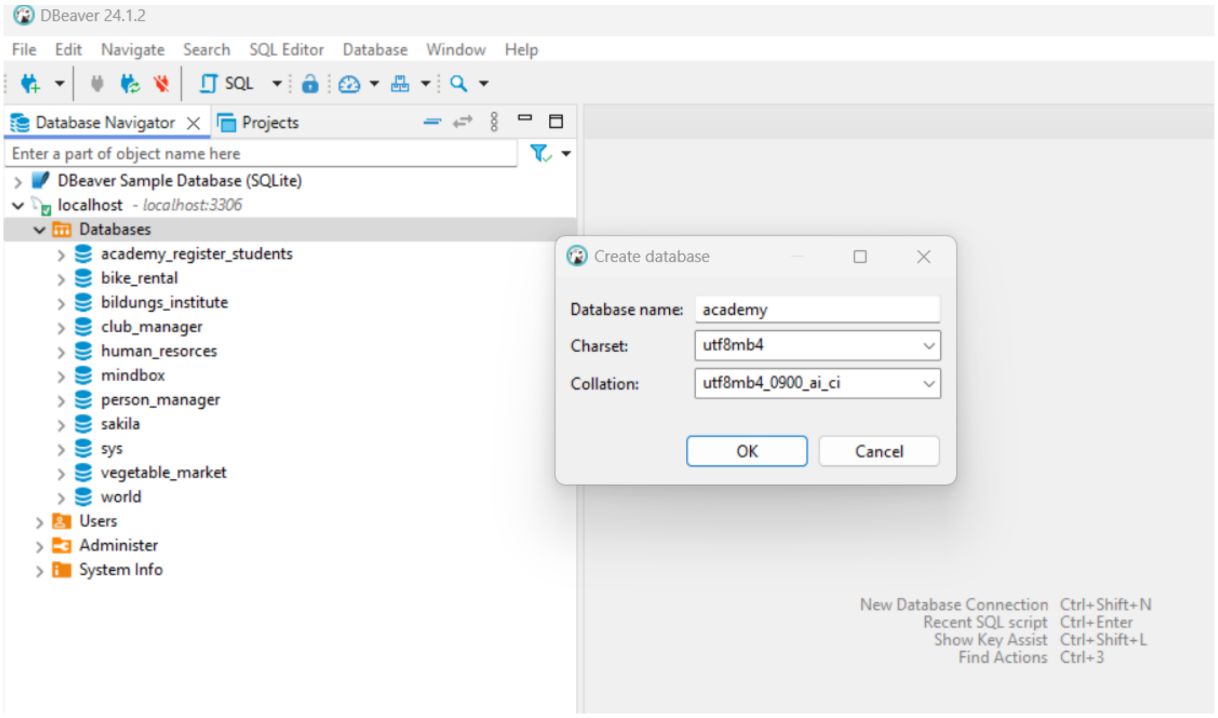
Raw data | HTML form

Username	admin
Password	••••••••••

POST

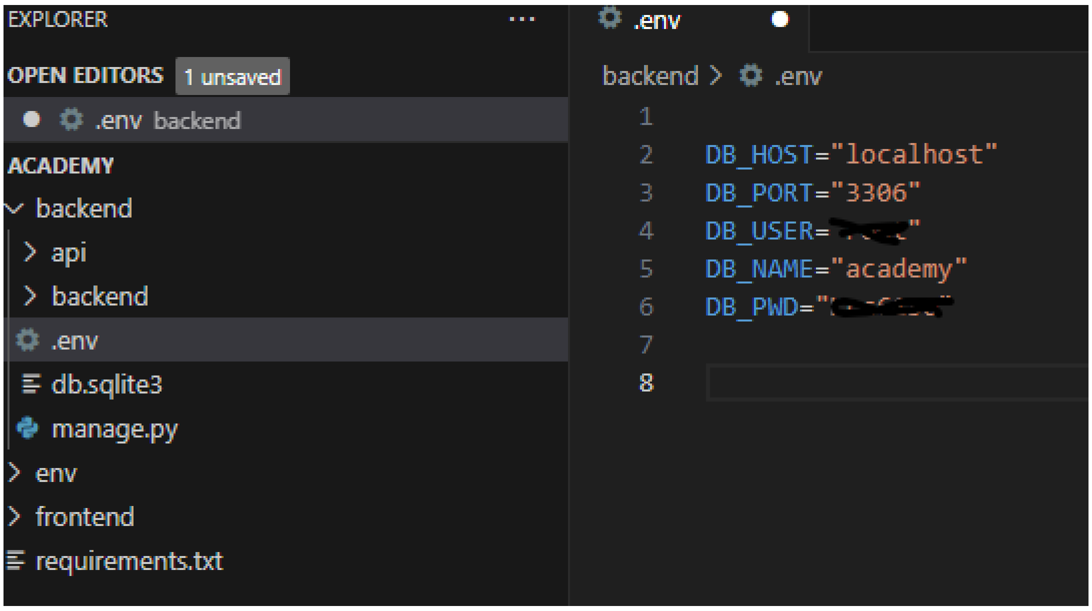
2.8 Create MySQL database

Using DBeaver and create a new database.


DBeaver 24.1.2
File Edit Navigate Search SQL Editor Database Window Help
Database Navigator
Projects
Enter a part of object name here
DBeaver Sample Database (SQLite)
localhost - localhost:3306
Databases
academy_register_students
bike_rental
bildungs_institute
club_manager
human_resorces
mindbox
person_manager
sakila
sys
vegetable_market
world
Users
Administer
System Info
Create database
OK
Cancel
New Database Connection Ctrl+Shift+N
Recent SQL script Ctrl+Enter
Show Key Assist Ctrl+Shift+L
Find Actions Ctrl+3

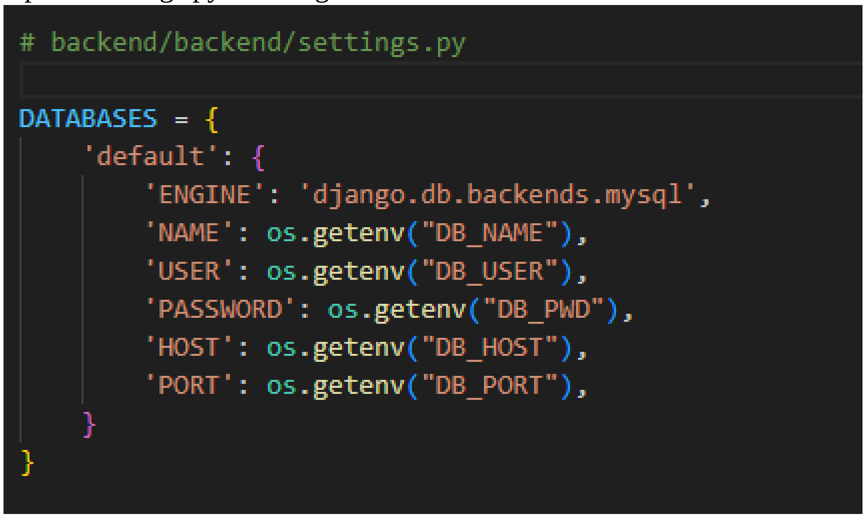
Database name: academy
Charset: utf8mb4
Collation: utf8mb4_0900_ai_ci

Create .env file and insert database access credentials into it.



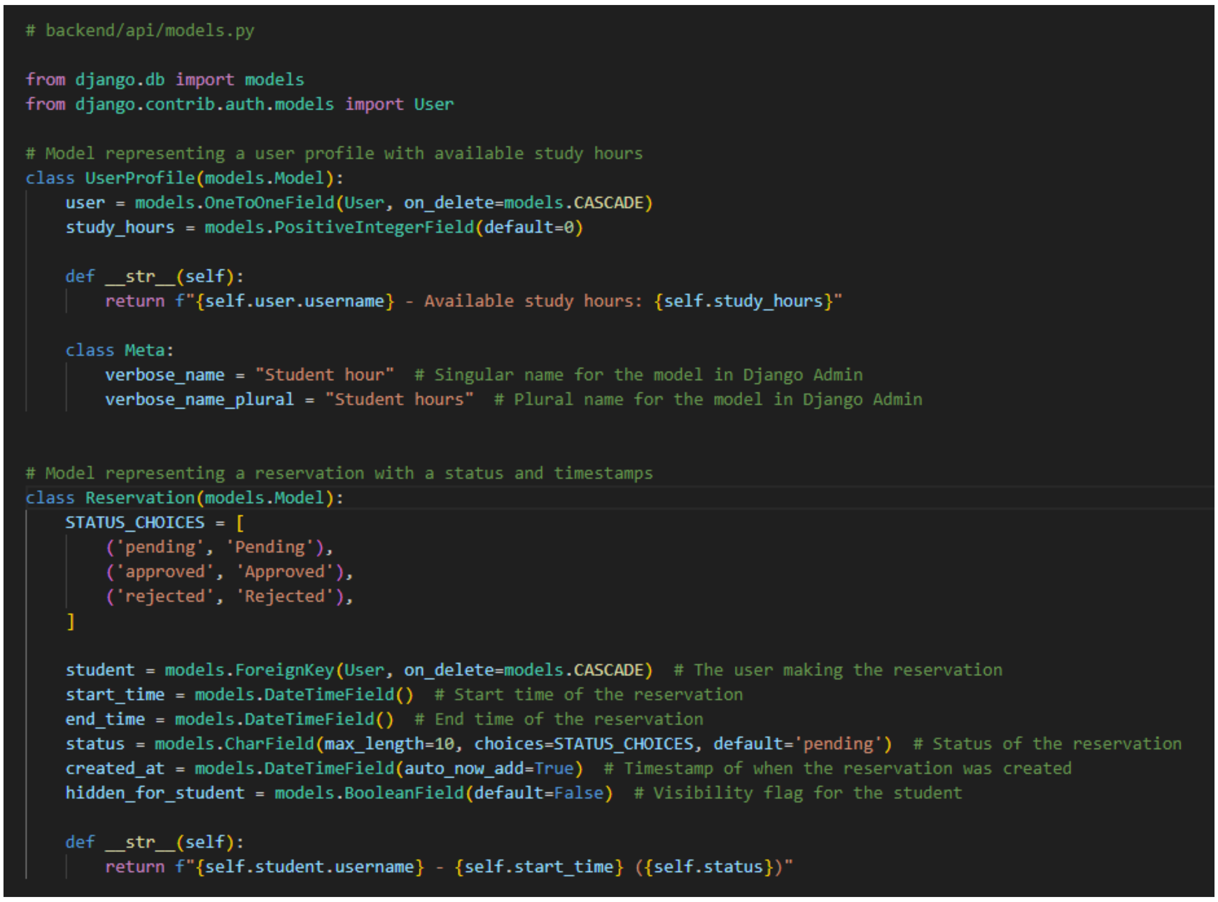
```

DB_HOST="localhost"
DB_PORT="3306"
DB_USER=[redacted]"
DB_NAME="academy"
DB_PWD="[redacted]"

```

Update settings.py to change the details to our database.



```python
# backend/backend/settings.py

DATABASES = {
    'default': {
        'ENGINE': 'django.db.backends.mysql',
        'NAME': os.getenv("DB_NAME"),
        'USER': os.getenv("DB_USER"),
        'PASSWORD': os.getenv("DB_PWD"),
        'HOST': os.getenv("DB_HOST"),
        'PORT': os.getenv("DB_PORT"),
    }
}
```

2.9 Create models Student hour, Reservation, Order and Control



```python
# backend/api/models.py

from django.db import models
from django.contrib.auth.models import User

# Model representing a user profile with available study hours
class UserProfile(models.Model):
    user = models.OneToOneField(User, on_delete=models.CASCADE)
    study_hours = models.PositiveIntegerField(default=0)

    def __str__(self):
        return f"{self.user.username} - Available study hours: {self.study_hours}"

    class Meta:
        verbose_name = "Student hour"  # Singular name for the model in Django Admin
        verbose_name_plural = "Student hours"  # Plural name for the model in Django Admin


# Model representing a reservation with a status and timestamps
class Reservation(models.Model):
    STATUS_CHOICES = [
        ('pending', 'Pending'),
        ('approved', 'Approved'),
        ('rejected', 'Rejected'),
    ]

    student = models.ForeignKey(User, on_delete=models.CASCADE)  # The user making the reservation
    start_time = models.DateTimeField()  # Start time of the reservation
    end_time = models.DateTimeField()  # End time of the reservation
    status = models.CharField(max_length=10, choices=STATUS_CHOICES, default='pending')  # Status of the reservation
    created_at = models.DateTimeField(auto_now_add=True)  # Timestamp of when the reservation was created
    hidden_for_student = models.BooleanField(default=False)  # Visibility flag for the student

    def __str__(self):
        return f"{self.student.username} - {self.start_time} ({self.status})"
```

Model: `UserProfile`

- This model represents a user profile with available study hours.

Fields:

- `user`: A one-to-one relationship with Django's built-in `User` model, linking each profile to a specific user. If the user is deleted, the profile is also deleted (`on_delete=models.CASCADE`).
- `study_hours`: A positive integer field representing the number of study hours available to the user. Default is set to 0.

Methods:

- `__str__`: Returns a string representation of the user profile, showing the username and available study hours.

Meta Class:

- `verbose_name`: Sets a human-readable singular name for this model in Django Admin.
- `verbose_name_plural`: Sets a human-readable plural name for this model in Django Admin.

Model: `Reservation`

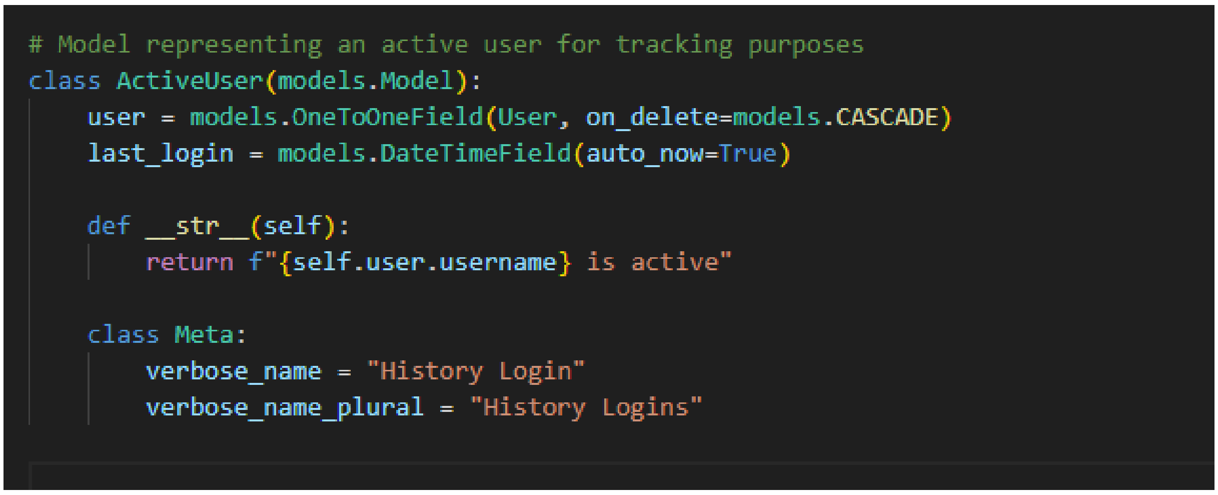
- This model represents a reservation with a status and timestamps.

Fields:

- `STATUS_CHOICES`: Defines the possible statuses for a reservation (`pending`, `approved`, and `rejected`).
- `student`: A foreign key linking the reservation to a user (student) who made it. If the student is deleted, the reservation is also deleted.
- `start_time`: The start time of the reservation.
- `end_time`: The end time of the reservation.
- `status`: A character field with a choice constraint. Default status is `pending`.
- `created_at`: A timestamp indicating when the reservation was created. Automatically set when a new reservation is created (`auto_now_add=True`).
- `hidden_for_student`: A boolean field indicating if the reservation is hidden from the student. Default is `False`.

Methods:

`__str__`: Returns a string representation of the reservation, showing the student's username, start time, and reservation status.


```python
# Model representing an active user for tracking purposes
class ActiveUser(models.Model):
    user = models.OneToOneField(User, on_delete=models.CASCADE)
    last_login = models.DateTimeField(auto_now=True)

    def __str__(self):
        return f"{self.user.username} is active"

    class Meta:
        verbose_name = "History Login"
        verbose_name_plural = "History Logins"
```

Model: `ActiveUser`

This model represents an active user for tracking purposes.

Fields:

- `user`: A one-to-one relationship with Django's `User` model, indicating the specific user being tracked. If the user is deleted, the tracking record is also deleted.
- `last_login`: A datetime field that stores the last login time of the user. Automatically updates to the current timestamp each time the record is saved (`auto_now=True`).

Methods:

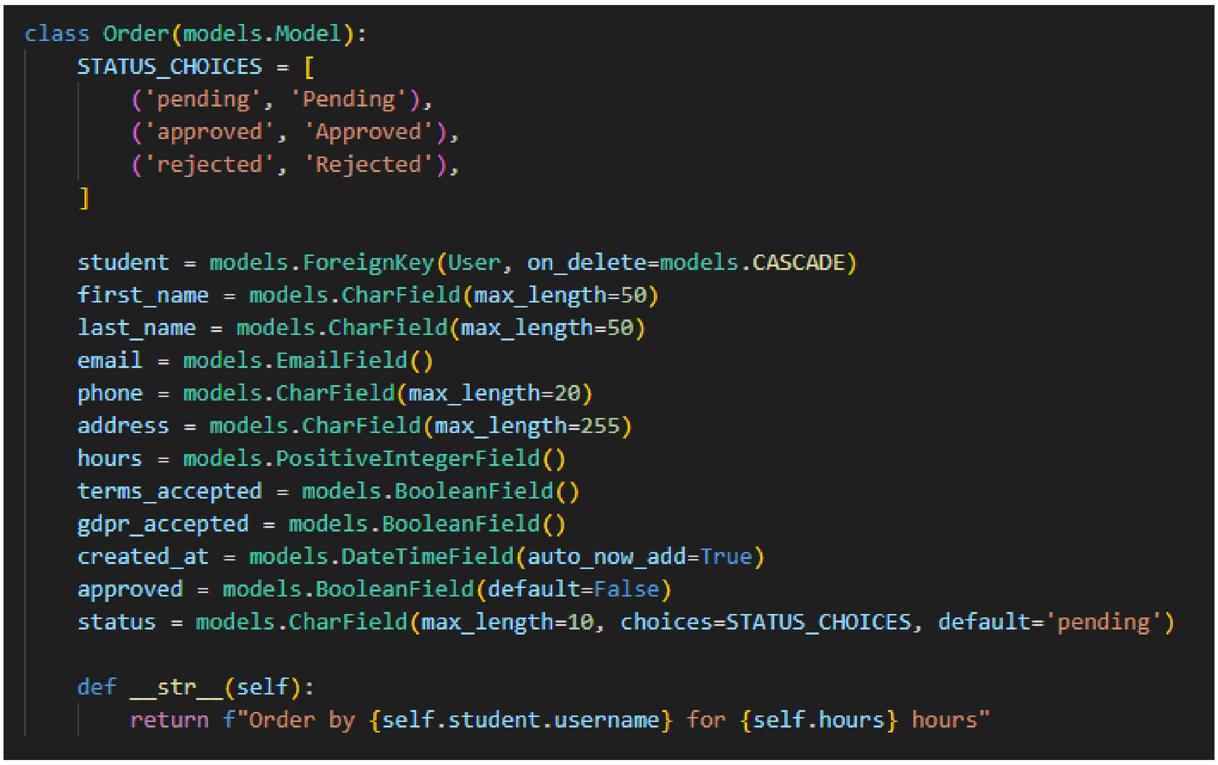
`__str__`: Returns a string indicating that the user is active, showing the username.

Meta Class:

- `verbose_name`: Sets a human-readable singular name for this model in Django Admin.
- `verbose_name_plural`: Sets a human-readable plural name for this model in Django Admin.

`Order` Model

This model represents an order placed by a user (student) for purchasing study hours, along with the associated details like contact information, status, and terms acceptance.



```python
class Order(models.Model):
    STATUS_CHOICES = [
        ('pending', 'Pending'),
        ('approved', 'Approved'),
        ('rejected', 'Rejected'),
    ]

    student = models.ForeignKey(User, on_delete=models.CASCADE)
    first_name = models.CharField(max_length=50)
    last_name = models.CharField(max_length=50)
    email = models.EmailField()
    phone = models.CharField(max_length=20)
    address = models.CharField(max_length=255)
    hours = models.PositiveIntegerField()
    terms_accepted = models.BooleanField()
    gdpr_accepted = models.BooleanField()
    created_at = models.DateTimeField(auto_now_add=True)
    approved = models.BooleanField(default=False)
    status = models.CharField(max_length=10, choices=STATUS_CHOICES, default='pending')

    def __str__(self):
        return f"Order by {self.student.username} for {self.hours} hours"
```

Fields:

- **`STATUS_CHOICES`**: Defines the possible statuses of an order (`pending`, `approved`, `rejected`), providing a way to categorize each order's current state.
- **`student`**: A foreign key that links the order to the specific user (student) who placed it. The order is deleted if the associated user is deleted (`on_delete=models.CASCADE`).
- **`first_name`** and **`last_name`**: CharFields that store the first and last name of the student, with a maximum character length of 50.
- **`email`**: An EmailField that stores the email address of the student.
- **`phone`**: A CharField that stores the student's phone number, with a maximum length of 20 characters.
- **`address`**: A CharField for the student's address, with a maximum character length of 255.
- **`hours`**: A PositiveIntegerField that records the number of study hours the student wants to purchase.
- **`terms_accepted`** and **`gdpr_accepted`**: BooleanFields that indicate whether the student has accepted the terms and GDPR policy, respectively. These fields are essential for compliance and legal requirements.
- **`created_at`**: A DateTimeField that stores the timestamp of when the order was created. This field is set automatically when a new order is created (`auto_now_add=True`).

- **approved**: A BooleanField that indicates whether the order has been approved. Default is set to `False`.
- **status**: A CharField with options defined in `STATUS_CHOICES`, allowing the order status to be set to `pending`, `approved`, or `rejected`. Default is `pending`.

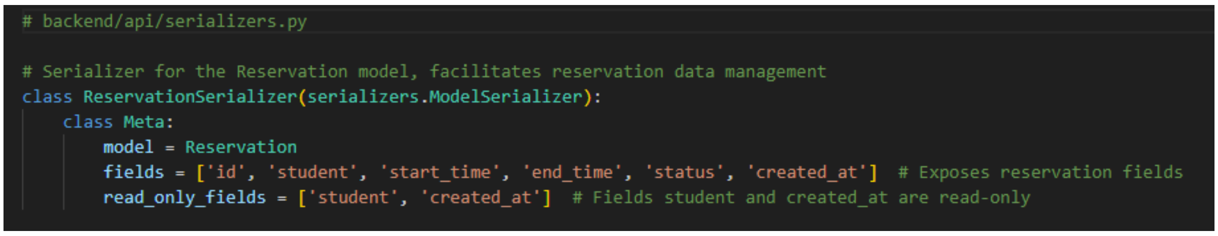
Methods:

- **__str__**: Returns a string representation of the order, including the username of the student and the number of hours purchased, making it easy to identify individual orders in the admin interface.

The Order model represents an individual order by a student to purchase study hours. It includes fields for personal information (name, email, phone, and address), order details (number of hours, status, approval), and legal compliance (terms and GDPR acceptance). The STATUS_CHOICES field allows for tracking the order's progress, and the __str__ method provides a readable format for each order instance.

2.10 Create two more serializers

The `ReservationSerializer` provides a structured way to expose the data of the `Reservation` model in an API, with certain fields (`student` and `created_at`) marked as read-only to prevent accidental modification. This serializer allows the creation and updating of reservations while ensuring important fields remain protected.



```
# backend/api/serializers.py

# Serializer for the Reservation model, facilitates reservation data management
class ReservationSerializer(serializers.ModelSerializer):
    class Meta:
        model = Reservation
        fields = ['id', 'student', 'start_time', 'end_time', 'status', 'created_at']  # Exposes reservation fields
        read_only_fields = ['student', 'created_at']  # Fields student and created_at are read-only
```

ReservationSerializer:

- This serializer class is designed to convert `Reservation` model instances into JSON format (and vice versa) for easier data transfer and management in an API.

Meta Class:

model: Specifies that this serializer is associated with the `Reservation` model.

fields: Defines the fields from the `Reservation` model that will be included in the serialized data. These fields are:

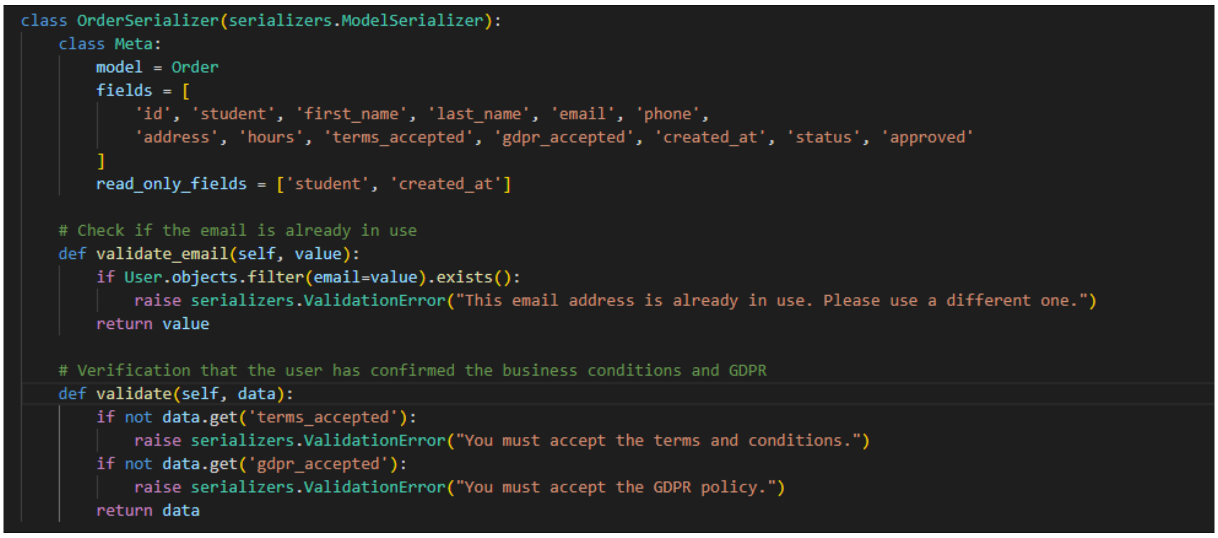
- `id`: Unique identifier for each reservation.
- `student`: The user who made the reservation.
- `start_time`: The start time of the reservation.
- `end_time`: The end time of the reservation.
- `status`: The status of the reservation (e.g., pending, approved, rejected).

- `created_at`: The timestamp of when the reservation was created.

read_only_fields: Specifies fields that are read-only, meaning they cannot be modified through the serializer. Here:

- `student`: This field is set as read-only, meaning the `student` cannot be changed through the API once it's set.
- `created_at`: This field is also read-only, as it is automatically set when the reservation is created and should not be modified.

The `OrderSerializer` is designed to transform `Order` model instances into JSON format for API interaction and vice versa, allowing easy data handling for order-related requests. It includes validations to ensure that users accept essential terms before creating an order.



```python
class OrderSerializer(serializers.ModelSerializer):
    class Meta:
        model = Order
        fields = [
            'id', 'student', 'first_name', 'last_name', 'email', 'phone',
            'address', 'hours', 'terms_accepted', 'gdpr_accepted', 'created_at', 'status', 'approved'
        ]
        read_only_fields = ['student', 'created_at']

    # Check if the email is already in use
    def validate_email(self, value):
        if User.objects.filter(email=value).exists():
            raise serializers.ValidationError("This email address is already in use. Please use a different one.")
        return value

    # Verification that the user has confirmed the business conditions and GDPR
    def validate(self, data):
        if not data.get('terms_accepted'):
            raise serializers.ValidationError("You must accept the terms and conditions.")
        if not data.get('gdpr_accepted'):
            raise serializers.ValidationError("You must accept the GDPR policy.")
        return data
```

Key Features of OrderSerializer:

- **Model Association**: Links directly to the `Order` model, so it knows which fields to serialize and deserialize.
- **Field Control**: Specifies which fields are included in the serialized data and marks certain fields as read-only.

Meta Class:

- **model**: Specifies the `Order` model as the source for this serializer.
- **fields**: Lists the fields from the `Order` model to include in the API representation:
    - `id`: Unique identifier for the order.
    - `student`: The user placing the order.
    - `first_name`: First name of the student.
    - `last_name`: Last name of the student.
    - `email`: Contact email of the student.
    - `phone`: Contact phone number of the student.
    - `address`: Address of the student.

        - `hours`: The number of study hours ordered.
        - `terms_accepted`: Boolean indicating if the student accepted the terms.
        - `gdpr_accepted`: Boolean indicating if the student accepted the GDPR policy.
        - `created_at`: Timestamp of when the order was created.
        - `status`: Current status of the order (pending, approved, rejected).
        - `approved`: Boolean indicating whether the order has been approved.
- **read_only_fields**: Marks fields as unchangeable once set:
    - `student`: Cannot be modified through the serializer to ensure the student placing the order remains consistent.
    - `created_at`: Automatically set when the order is created and should not be altered.

Custom Validation Method:

`validate_email(self, value)`:

- Ensures that the email address provided for the order is not already in use.

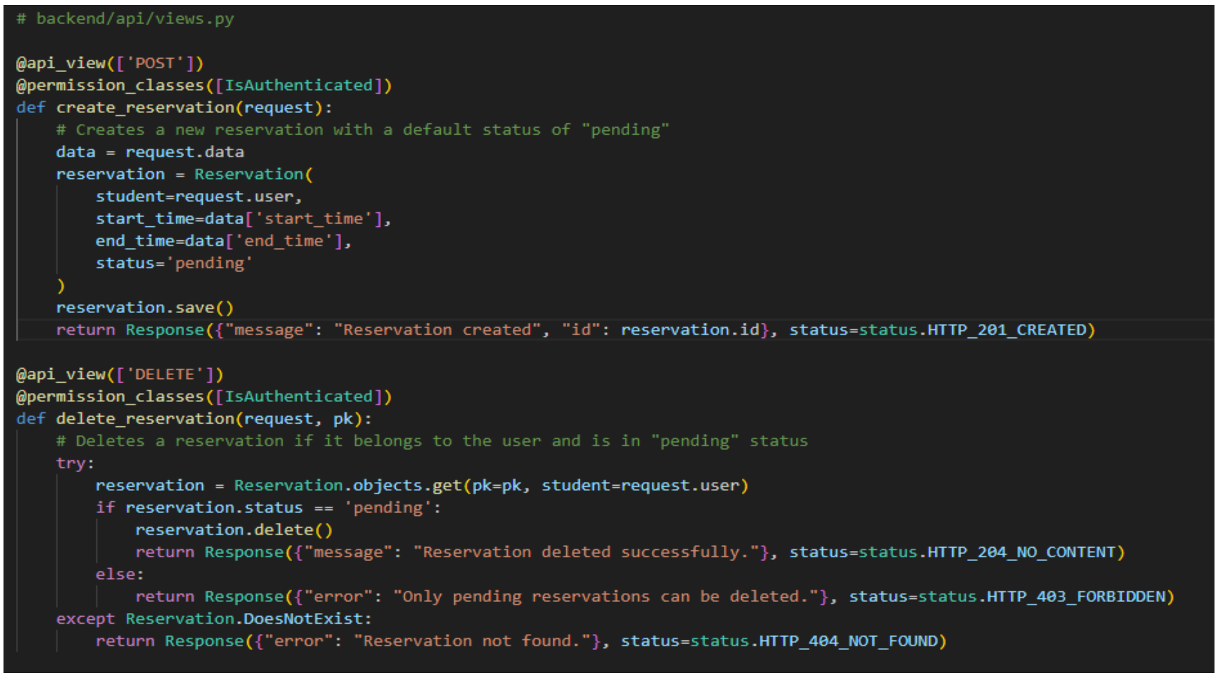
    - Checks if the provided email address exists in the `User` model.
    - If a match is found, raises a `ValidationError` with an appropriate message.
    - Returns the email value if it passes validation.

-
- **validate(self, data)**: Ensures that specific business rules are followed before saving the order:
    - Checks if the `terms_accepted` field is `True`. If not, raises a validation error with a message prompting the user to accept the terms and conditions.
    - Checks if the `gdpr_accepted` field is `True`. If not, raises a validation error asking the user to accept the GDPR policy.

2.11 Create another views

`create_reservation`: Allows an authenticated user to create a new reservation with a default status of "pending".

`delete_reservation`: Allows an authenticated user to delete their reservation only if it is in the "pending" status. Returns appropriate error messages if the reservation doesn’t exist or cannot be deleted due to its status.



```python
# backend/api/views.py

@api_view(['POST'])
@permission_classes([IsAuthenticated])
def create_reservation(request):
    # Creates a new reservation with a default status of "pending"
    data = request.data
    reservation = Reservation(
        student=request.user,
        start_time=data['start_time'],
        end_time=data['end_time'],
        status='pending'
    )
    reservation.save()
    return Response({"message": "Reservation created", "id": reservation.id}, status=status.HTTP_201_CREATED)

@api_view(['DELETE'])
@permission_classes([IsAuthenticated])
def delete_reservation(request, pk):
    # Deletes a reservation if it belongs to the user and is in "pending" status
    try:
        reservation = Reservation.objects.get(pk=pk, student=request.user)
        if reservation.status == 'pending':
            reservation.delete()
            return Response({"message": "Reservation deleted successfully."}, status=status.HTTP_204_NO_CONTENT)
        else:
            return Response({"error": "Only pending reservations can be deleted."}, status=status.HTTP_403_FORBIDDEN)
    except Reservation.DoesNotExist:
        return Response({"error": "Reservation not found."}, status=status.HTTP_404_NOT_FOUND)
```

`create_reservation` View:

Decorators:

- `@api_view(['POST'])`: Specifies that this view only accepts POST requests.
- `@permission_classes([IsAuthenticated])`: Requires the user to be authenticated to access this view.

Function:

- `create_reservation`: This function creates a new reservation for the authenticated user.

Steps:

- **Data Extraction**: Retrieves the reservation details (`start_time`, `end_time`) from the request data.
- **Reservation Creation**: Creates a new `Reservation` instance with:
    - `student`: Set to the authenticated user (`request.user`).
    - `start_time` and `end_time`: Set to the values provided in the request.
    - `status`: Defaulted to `'pending'`.
- **Save**: Saves the reservation to the database.
- **Response**: Returns a JSON response with a success message and the reservation ID, along with a status code `HTTP_201_CREATED` (created successfully).

`delete_reservation` View:

Decorators:

- `@api_view(['DELETE'])`: Specifies that this view only accepts DELETE requests.
- `@permission_classes([IsAuthenticated])`: Requires the user to be authenticated to access this view.

Function:

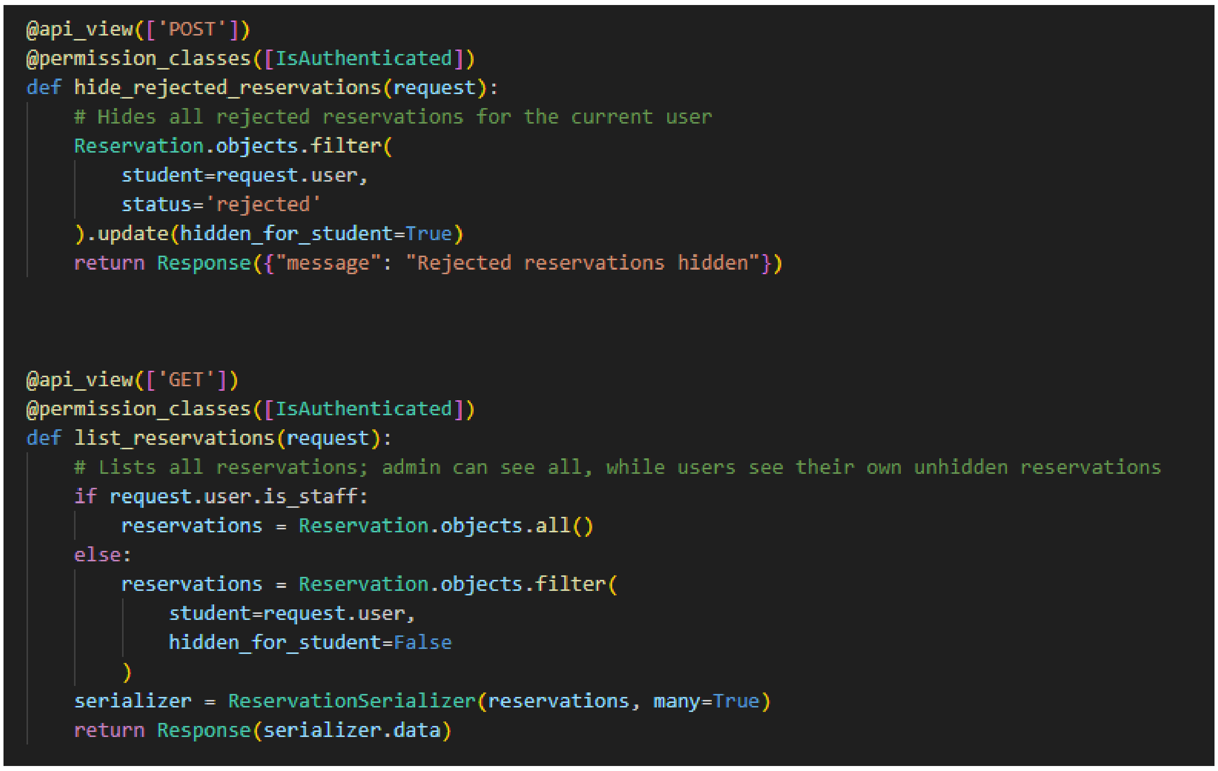
`delete_reservation`: This function deletes a reservation if it belongs to the authenticated user and is in a "pending" status.

Steps:

- **Reservation Retrieval**: Attempts to retrieve the reservation by primary key (`pk`) and ensure it belongs to the authenticated user (`student=request.user`).
- **Status Check**:
  - If the reservation status is `'pending'`, it proceeds to delete the reservation.
  - If the status is not `'pending'`, it returns a JSON response with an error message and a `HTTP_403_FORBIDDEN` status code, indicating that only pending reservations can be deleted.
- **Exception Handling**:
  - If the reservation with the given `pk` does not exist for the authenticated user, it raises a `Reservation.DoesNotExist` exception, and the function returns a JSON response with an error message and a `HTTP_404_NOT_FOUND` status code.

hide_rejected_reservations: This view allows an authenticated user to hide all their reservations that have a "rejected" status, setting `hidden_for_student=True` on these entries.
list_reservations: This view lists reservations, with different access depending on the user role:

- Admin users see all reservations.
- Regular users only see their own reservations that are not hidden.



```python
@api_view(['POST'])
@permission_classes([IsAuthenticated])
def hide_rejected_reservations(request):
    # Hides all rejected reservations for the current user
    Reservation.objects.filter(
        student=request.user,
        status='rejected'
    ).update(hidden_for_student=True)
    return Response({"message": "Rejected reservations hidden"})



@api_view(['GET'])
@permission_classes([IsAuthenticated])
def list_reservations(request):
    # Lists all reservations; admin can see all, while users see their own unhidden reservations
    if request.user.is_staff:
        reservations = Reservation.objects.all()
    else:
        reservations = Reservation.objects.filter(
            student=request.user,
            hidden_for_student=False
        )
    serializer = ReservationSerializer(reservations, many=True)
    return Response(serializer.data)
```

`hide_rejected_reservations` View:

Decorators:

- `@api_view(['POST'])`: Specifies that this view only accepts POST requests.
- `@permission_classes([IsAuthenticated])`: Requires the user to be authenticated to access this view.

Function:

- `hide_rejected_reservations`: This function hides all reservations with a "rejected" status for the authenticated user.

Steps:

- **Query**: Filters reservations belonging to the authenticated user (`student=request.user`) and with a `status` of `'rejected'`.
- **Update**: Sets `hidden_for_student=True` for all reservations that match the filter, effectively hiding these reservations from the user.
- **Response**: Returns a JSON response with a confirmation message indicating that rejected reservations have been hidden.

`list_reservations` View:

Decorators:

- `@api_view(['GET'])`: Specifies that this view only accepts GET requests.
- `@permission_classes([IsAuthenticated])`: Requires the user to be authenticated to access this view.

Function:

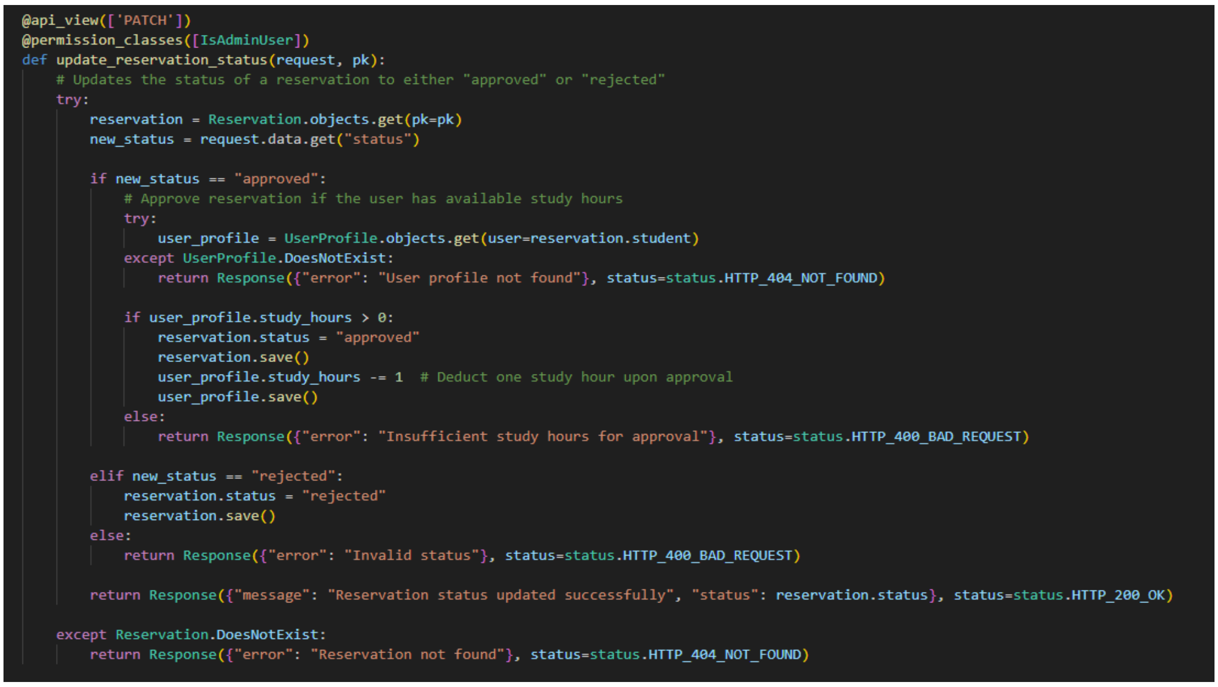
`list_reservations`: This function lists reservations with different access levels for admin and regular users.

Steps:

- **Admin Check**: Checks if the authenticated user is an admin (`is_staff=True`).
    - **If Admin**: Retrieves all reservations (`Reservation.objects.all()`).
    - **If Not Admin**: Filters reservations to only include those belonging to the user (`student=request.user`) and that are not hidden (`hidden_for_student=False`).
- **Serialization**: Serializes the list of reservations using `ReservationSerializer` to prepare the data for JSON format.
- **Response**: Returns the serialized data in a JSON response.

update_reservation_status:

- This view allows an admin to update the status of a reservation:
    - If setting the status to "approved," it checks if the user has available study hours, deducting one hour upon approval.
    - If setting the status to "rejected," it simply updates the status.
- The view validates that the new status is either "approved" or "rejected" and handles errors such as invalid status, missing user profiles, or reservations not found.



```python
@api_view(['PATCH'])
@permission_classes([IsAdminUser])
def update_reservation_status(request, pk):
    # Updates the status of a reservation to either "approved" or "rejected"
    try:
        reservation = Reservation.objects.get(pk=pk)
        new_status = request.data.get("status")

        if new_status == "approved":
            # Approve reservation if the user has available study hours
            try:
                user_profile = UserProfile.objects.get(user=reservation.student)
            except UserProfile.DoesNotExist:
                return Response({"error": "User profile not found"}, status=status.HTTP_404_NOT_FOUND)

            if user_profile.study_hours > 0:
                reservation.status = "approved"
                reservation.save()
                user_profile.study_hours -= 1  # Deduct one study hour upon approval
                user_profile.save()
            else:
                return Response({"error": "Insufficient study hours for approval"}, status=status.HTTP_400_BAD_REQUEST)

        elif new_status == "rejected":
            reservation.status = "rejected"
            reservation.save()
        else:
            return Response({"error": "Invalid status"}, status=status.HTTP_400_BAD_REQUEST)

        return Response({"message": "Reservation status updated successfully", "status": reservation.status}, status=status.HTTP_200_OK)

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

Decorators:

- `@api_view(['PATCH'])`: Specifies that this view only accepts PATCH requests, typically used for partial updates.
- `@permission_classes([IsAdminUser])`: Restricts access to admin users only.

Function:

`update_reservation_status`: This function updates the status of a reservation to either "approved" or "rejected" based on the provided input.

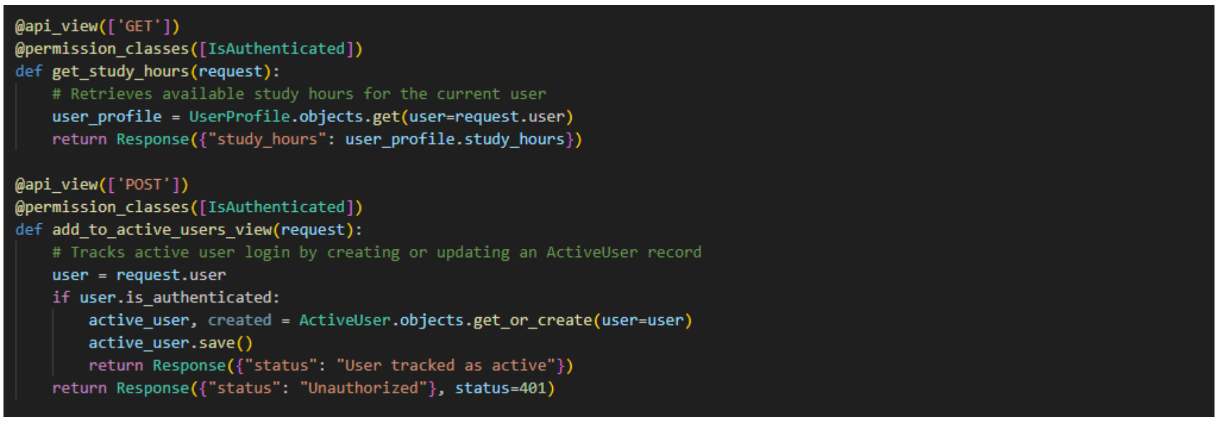
Steps:

- **Retrieve Reservation**:
    - Tries to retrieve the reservation with the primary key `pk`. If not found, it returns a `404 NOT FOUND` response.

- **Check New Status**:
    - **If `new_status` is "approved"**:
        - Attempts to retrieve the `UserProfile` of the student associated with the reservation. If the profile is not found, it returns a `404 NOT FOUND` error.
        - Checks if the user has available study hours (`study_hours > 0`).
            - If there are available hours, it:
                - Sets the reservation status to "approved."
                - Deducts one study hour from the user's profile.
                - Saves both the reservation and the updated `UserProfile`.
            - If there are no available hours, it returns a `400 BAD REQUEST` error with a message indicating "Insufficient study hours for approval."
    - **If `new_status` is "rejected"**:
        - Sets the reservation status to "rejected" and saves the reservation.
    - **If `new_status` is invalid**:
        - If the status is anything other than "approved" or "rejected," it returns a `400 BAD REQUEST` error with a message "Invalid status."
- **Success Response**:
    - If the status update is successful, it returns a success message with the updated status and a `200 OK` response.
- **Error Handling**:
    - If the reservation is not found, it catches the `Reservation.DoesNotExist` exception and returns a `404 NOT FOUND` error with a message "Reservation not found."

get_study_hours: Retrieves the available study hours for the authenticated user and returns them in a JSON response.

add_to_active_users_view: Tracks the authenticated user's activity by creating or updating an `ActiveUser` record, logging their login. If the user is not authenticated, it returns a `401 Unauthorized` response.


```
@api_view(['GET'])
@permission_classes([IsAuthenticated])
def get_study_hours(request):
    # Retrieves available study hours for the current user
    user_profile = UserProfile.objects.get(user=request.user)
    return Response({"study_hours": user_profile.study_hours})

@api_view(['POST'])
@permission_classes([IsAuthenticated])
def add_to_active_users_view(request):
    # Tracks active user login by creating or updating an ActiveUser record
    user = request.user
    if user.is_authenticated:
        active_user, created = ActiveUser.objects.get_or_create(user=user)
        active_user.save()
        return Response({"status": "User tracked as active"})
    return Response({"status": "Unauthorized"}, status=401)
```

`get_study_hours` View

Decorators:

- `@api_view(['GET'])`: Specifies that this view only accepts GET requests.
- `@permission_classes([IsAuthenticated])`: Requires the user to be authenticated to access this view.

Function:

- `get_study_hours`: This function retrieves the number of available study hours for the authenticated user.

Steps:

- **User Profile Retrieval**: Finds the `UserProfile` associated with the authenticated user (`user=request.user`).
- **Response**: Returns a JSON response containing the available `study_hours` from the user profile.

`add_to_active_users_view` View

Decorators:

- `@api_view(['POST'])`: Specifies that this view only accepts POST requests.
- `@permission_classes([IsAuthenticated])`: Requires the user to be authenticated to access this view.

Function:

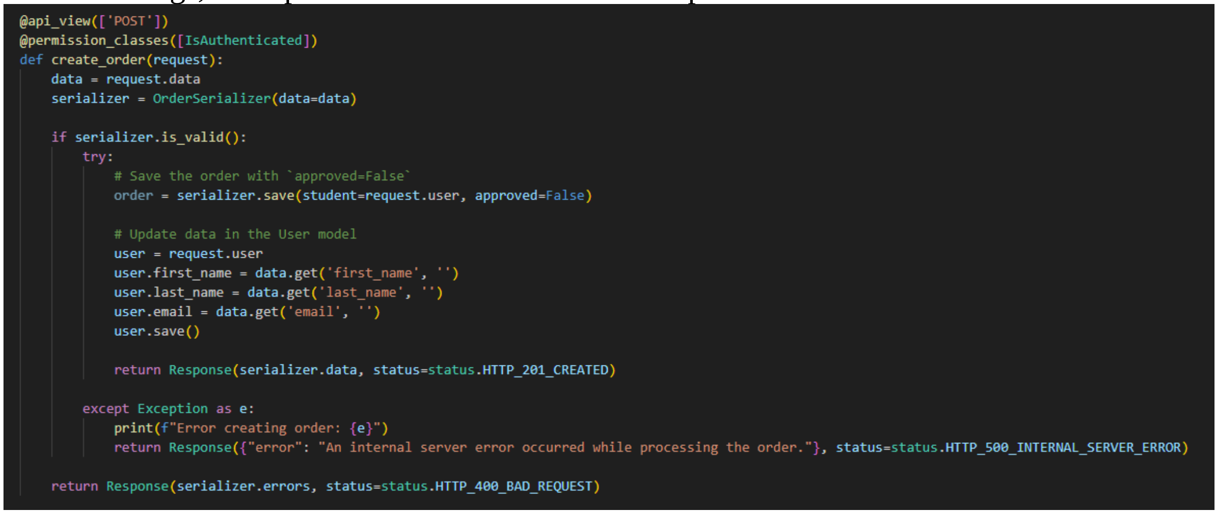
`add_to_active_users_view`: This function tracks the user's login status by creating or updating an `ActiveUser` record for the authenticated user.

Steps:

User Check: Checks if the user is authenticated.

- If authenticated:
    - Uses `get_or_create` to retrieve or create an `ActiveUser` record for the user.
    - Calls `save()` on the `active_user` record to update the timestamp if the record already exists.
    - Returns a success message indicating that the user is tracked as active.
- If not authenticated:
    - Returns a response with status `401 Unauthorized`.

create_order View: This view allows an authenticated user to create an order by sending a POST request with the necessary data. The view validates the data, saves the order with default settings, and updates some fields in the user's profile.



```python
@api_view(['POST'])
@permission_classes([IsAuthenticated])
def create_order(request):
    data = request.data
    serializer = OrderSerializer(data=data)

    if serializer.is_valid():
        try:
            # Save the order with `approved=False`
            order = serializer.save(student=request.user, approved=False)

            # Update data in the User model
            user = request.user
            user.first_name = data.get('first_name', '')
            user.last_name = data.get('last_name', '')
            user.email = data.get('email', '')
            user.save()

            return Response(serializer.data, status=status.HTTP_201_CREATED)

        except Exception as e:
            print(f"Error creating order: {e}")
            return Response({"error": "An internal server error occurred while processing the order."}, status=status.HTTP_500_INTERNAL_SERVER_ERROR)

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

Decorators:

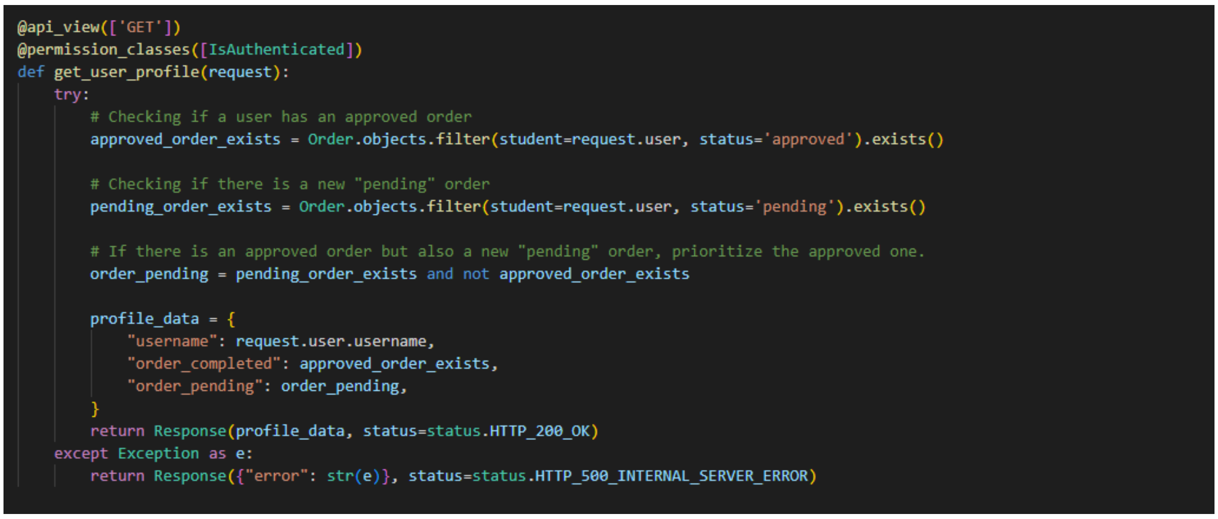
- **@api_view(['POST'])**: Restricts this view to accept only POST requests.
- **@permission_classes([IsAuthenticated])**: Requires the user to be authenticated to access this view.

Function:

- **create_order(request)**: This function handles the creation of a new order for the authenticated user.

Steps:

1. **Data Extraction**: Retrieves the order data from the request (`data = request.data`).
2. **Order Serialization**: Initializes the `OrderSerializer` with the provided data to handle validation and serialization.
3. **Validation Check**:
   - **if serializer.is_valid()**: Checks if the provided data meets all validation requirements.
   - If valid, the order is saved with:
     - **student**: The current authenticated user (`request.user`).
     - **approved**: Set to `False` by default (pending approval).
4. **User Profile Update**:
   - Updates specific fields in the `User` model for the authenticated user (like `first_name`, `last_name`, and `email`) based on the data provided in the order.
   - **user.save()**: Saves these changes to the user profile.
5. **Response**:
   - **Successful Creation**: If the order is successfully created and the data is valid, it returns a JSON response with the serialized order data and a status of `HTTP_201_CREATED`.
   - **Internal Server Error**: If an error occurs during saving, it catches the exception, logs it, and returns an error message with a `HTTP_500_INTERNAL_SERVER_ERROR` status.
   - **Validation Error**: If the data is invalid, it returns the serializer's errors with a `HTTP_400_BAD_REQUEST` status.



```python
@api_view(['GET'])
@permission_classes([IsAuthenticated])
def get_user_profile(request):
    try:
        # Checking if a user has an approved order
        approved_order_exists = Order.objects.filter(student=request.user, status='approved').exists()

        # Checking if there is a new "pending" order
        pending_order_exists = Order.objects.filter(student=request.user, status='pending').exists()

        # If there is an approved order but also a new "pending" order, prioritize the approved one.
        order_pending = pending_order_exists and not approved_order_exists

        profile_data = {
            "username": request.user.username,
            "order_completed": approved_order_exists,
            "order_pending": order_pending,
        }
        return Response(profile_data, status=status.HTTP_200_OK)
    except Exception as e:
        return Response({"error": str(e)}, status=status.HTTP_500_INTERNAL_SERVER_ERROR)
```

`get_user_profile` View

Decorators:

- **`@api_view(['GET'])`**:
  - Specifies that this view only accepts HTTP GET requests.
- **`@permission_classes([IsAuthenticated])`**:
  - Requires the user to be authenticated to access this view.

- If the user is not logged in, a 401 Unauthorized response will be returned.

Function:

- **`get_user_profile(request)`**:
    - This function retrieves the user's profile data, focusing on their order statuses, and returns a structured JSON response.

Steps:

1. **Check for Approved Orders**:

   ```
   approved_order_exists =
   Order.objects.filter(student=request.user,
   status='approved').exists()
   ```

    - Queries the database to check if the user has at least one approved order.
    - Uses `.exists()` for efficient checking, stopping at the first matching record.
2. **Check for Pending Orders**:

   ```
   pending_order_exists =
   Order.objects.filter(student=request.user,
   status='pending').exists()
   ```

    - Queries the database to check if the user has at least one pending order.
3. **Resolve Conflicts Between Approved and Pending Orders**:

   ```
   order_pending = pending_order_exists and not
   approved_order_exists
   ```

    - If both approved and pending orders exist, prioritizes the approved order:
        - Sets `order_pending` to `False` if an approved order exists.
        - Otherwise, `order_pending` remains `True`.
4. **Construct Profile Data**:

   ```
   profile_data = {
       "username": request.user.username,
       "order_completed": approved_order_exists,
       "order_pending": order_pending,
   }
   ```

    - Builds a JSON object containing:
        - `username`: The username of the authenticated user.
        - `order_completed`: Boolean indicating if the user has completed an order.
        - `order_pending`: Boolean indicating if the user has a pending order (only if there are no completed orders).

5. **Return Profile Data**:

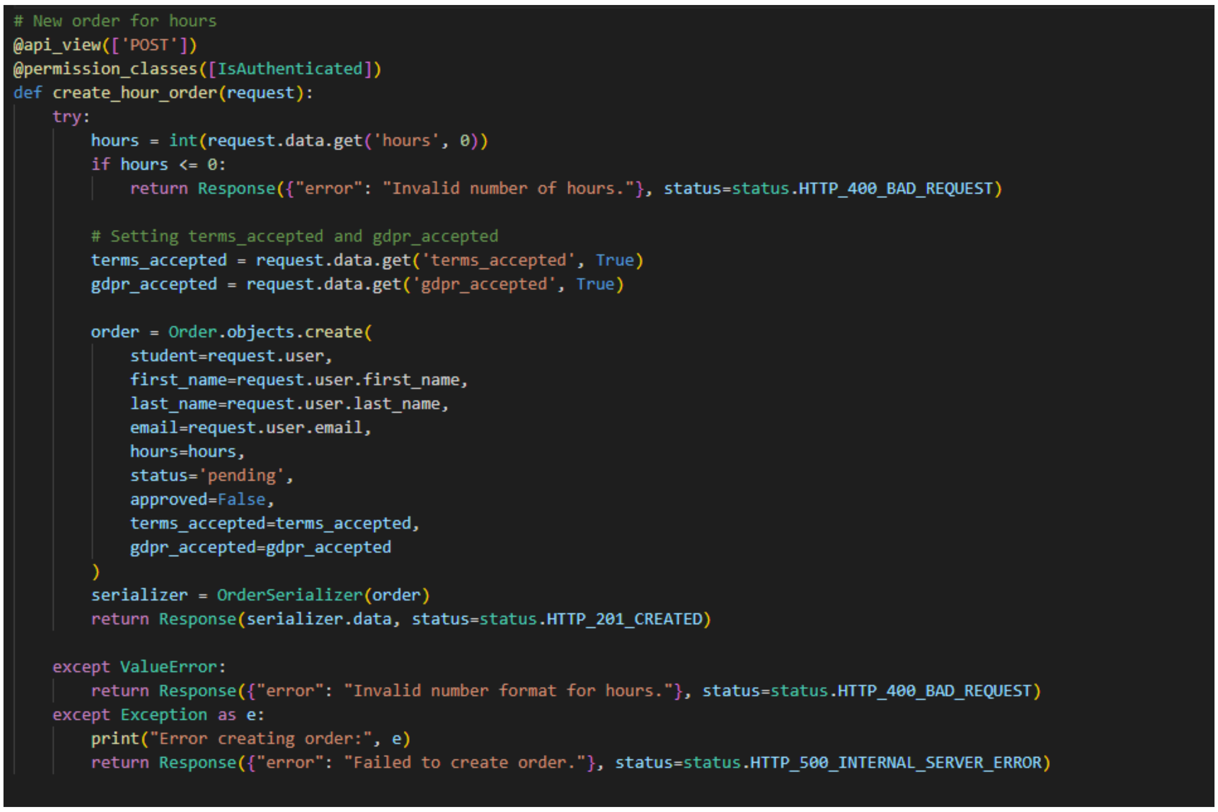
   ```
   return Response(profile_data, status=status.HTTP_200_OK)
   ```

   - Returns the constructed `profile_data` JSON object with an HTTP 200 OK status.

6. **Handle Errors**:

   ```
   except Exception as e:
       return Response({"error": str(e)},
   status=status.HTTP_500_INTERNAL_SERVER_ERROR)
   ```

   - Catches unexpected errors during the execution of the function.
   - Logs the error message and returns an HTTP 500 Internal Server Error response with details for debugging.



```
# New order for hours
@api_view(['POST'])
@permission_classes([IsAuthenticated])
def create_hour_order(request):
    try:
        hours = int(request.data.get('hours', 0))
        if hours <= 0:
            return Response({"error": "Invalid number of hours."}, status=status.HTTP_400_BAD_REQUEST)

        # Setting terms_accepted and gdpr_accepted
        terms_accepted = request.data.get('terms_accepted', True)
        gdpr_accepted = request.data.get('gdpr_accepted', True)

        order = Order.objects.create(
            student=request.user,
            first_name=request.user.first_name,
            last_name=request.user.last_name,
            email=request.user.email,
            hours=hours,
            status='pending',
            approved=False,
            terms_accepted=terms_accepted,
            gdpr_accepted=gdpr_accepted
        )
        serializer = OrderSerializer(order)
        return Response(serializer.data, status=status.HTTP_201_CREATED)

    except ValueError:
        return Response({"error": "Invalid number format for hours."}, status=status.HTTP_400_BAD_REQUEST)
    except Exception as e:
        print("Error creating order:", e)
        return Response({"error": "Failed to create order."}, status=status.HTTP_500_INTERNAL_SERVER_ERROR)
```

`create_hour_order` View

Decorators:

- **`@api_view(['POST'])`**:
  - Specifies that this view only accepts HTTP POST requests.
- **`@permission_classes([IsAuthenticated])`**:
  - Requires the user to be authenticated to access this view.
  - If the user is not logged in, a 401 Unauthorized response will be returned.

Function:

- **`create_hour_order(request)`**:
    - Handles the creation of a new order for additional study hours.

Steps:

1. **Retrieve and Validate Hours**:

    ```
    hours = int(request.data.get('hours', 0))
    if hours <= 0:
        return Response({"error": "Invalid number of hours."},
    status=status.HTTP_400_BAD_REQUEST)
    ```

    - Extracts the number of hours from the incoming request data.
    - Validates that the number of hours is greater than zero.
    - Returns a 400 Bad Request response if the validation fails.
2. **Set Terms and GDPR Acceptance Flags**:

    ```
    terms_accepted = request.data.get('terms_accepted', True)
    gdpr_accepted = request.data.get('gdpr_accepted', True)
    ```

    - Retrieves values for `terms_accepted` and `gdpr_accepted` from the request data.
    - Defaults both values to `True` if they are not explicitly provided in the request.
3. **Create Order**:

    ```
    order = Order.objects.create(
        student=request.user,
        first_name=request.user.first_name,
        last_name=request.user.last_name,
        email=request.user.email,
        hours=hours,
        status='pending',
        approved=False,
        terms_accepted=terms_accepted,
        gdpr_accepted=gdpr_accepted
    )
    ```

    - Creates a new `Order` object using the validated data.
    - Assigns the logged-in user (`request.user`) as the student associated with the order.
    - Sets the order status to "pending" and marks it as not approved.
4. **Serialize and Return Response**:

    ```
    serializer = OrderSerializer(order)
    return Response(serializer.data,
    status=status.HTTP_201_CREATED)
    ```

    - Serializes the newly created order using the `OrderSerializer`.
    - Returns the serialized order data with a 201 Created status.

5. **Handle Errors**:
   - **ValueError**:

     ```
     except ValueError:
         return Response({"error": "Invalid number format for
     hours."}, status=status.HTTP_400_BAD_REQUEST)
     ```

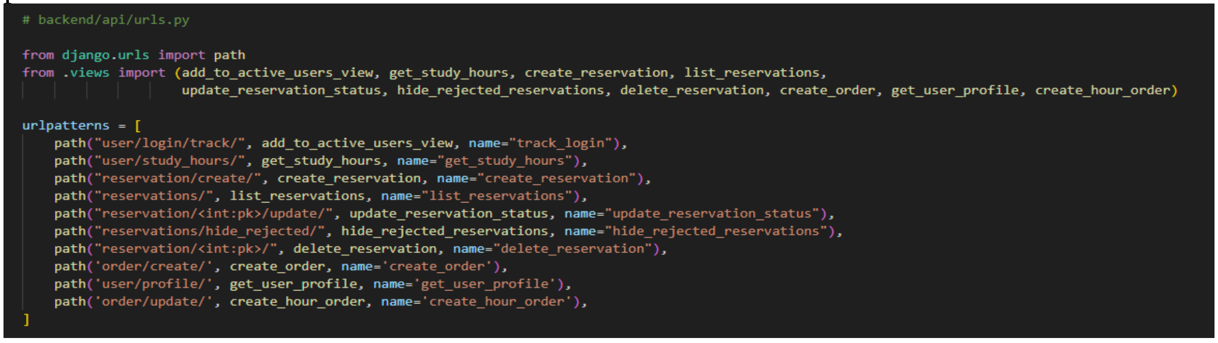
     - Catches errors where the `hours` value is not a valid integer.
     - Returns a 400 Bad Request response with a relevant error message.
   - **General Exceptions**:

     ```
     except Exception as e:
         print("Error creating order:", e)
         return Response({"error": "Failed to create order."},
     status=status.HTTP_500_INTERNAL_SERVER_ERROR)
     ```

     - Catches any other unexpected errors during the order creation process.
     - Logs the error and returns a 500 Internal Server Error response.

2.12 Create urls in api

This URL configuration enables organized access to various functionalities such as user tracking, study hours retrieval, reservation creation and management, and order processing. The setup ensures that users can interact with these resources while applying appropriate permissions and conditions for each route



```
# backend/api/urls.py

from django.urls import path
from .views import (add_to_active_users_view, get_study_hours, create_reservation, list_reservations,
                    update_reservation_status, hide_rejected_reservations, delete_reservation, create_order, get_user_profile, create_hour_order)

urlpatterns = [
    path("user/login/track/", add_to_active_users_view, name="track_login"),
    path("user/study_hours/", get_study_hours, name="get_study_hours"),
    path("reservation/create/", create_reservation, name="create_reservation"),
    path("reservations/", list_reservations, name="list_reservations"),
    path("reservation/<int:pk>/update/", update_reservation_status, name="update_reservation_status"),
    path("reservations/hide_rejected/", hide_rejected_reservations, name="hide_rejected_reservations"),
    path("reservation/<int:pk>/", delete_reservation, name="delete_reservation"),
    path('order/create/', create_order, name='create_order'),
    path('user/profile/', get_user_profile, name='get_user_profile'),
    path('order/update/', create_hour_order, name='create_hour_order'),
]
```

- **User Tracking and Profile**:

  - **`user/login/track/`**:
    - Tracks the login activity of the authenticated user by creating or updating an `ActiveUser` record.
  - **`user/study_hours/`**:
    - Retrieves the available study hours for the authenticated user.
  - **`user/profile/`**:
    - Retrieves the profile details of the authenticated user.

- **Reservations Management**:

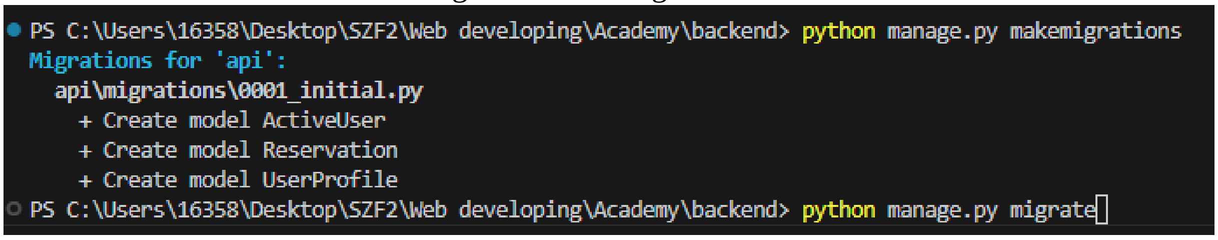
  - **`reservation/create/`**:
    - Allows the authenticated user to create a new reservation.
  - **`reservations/`**:

- Lists all reservations. Admin users can see all reservations, while regular users see only their own unhidden reservations.
- `reservation/<int:pk>/update/`:
  - Updates the status of a reservation (e.g., approve or reject) based on the provided primary key (`pk`).
- `reservations/hide_rejected/`:
  - Hides all reservations with a `"rejected"` status for the authenticated user.
- `reservation/<int:pk>/`:
  - Deletes a reservation identified by its primary key (`pk`) if it belongs to the authenticated user and is in a pending state.

- **Order Management**:

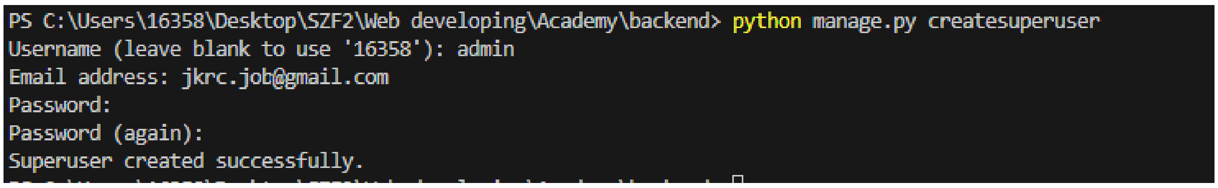
  - `order/create/`:
    - Allows the authenticated user to create a new order. This order is created with an initial status of not approved.
  - `order/update/`:
    - Updates the order details to add study hours for the user, handling order approval and updating user profile information accordingly.

Need make one more time makemigrations and migrate



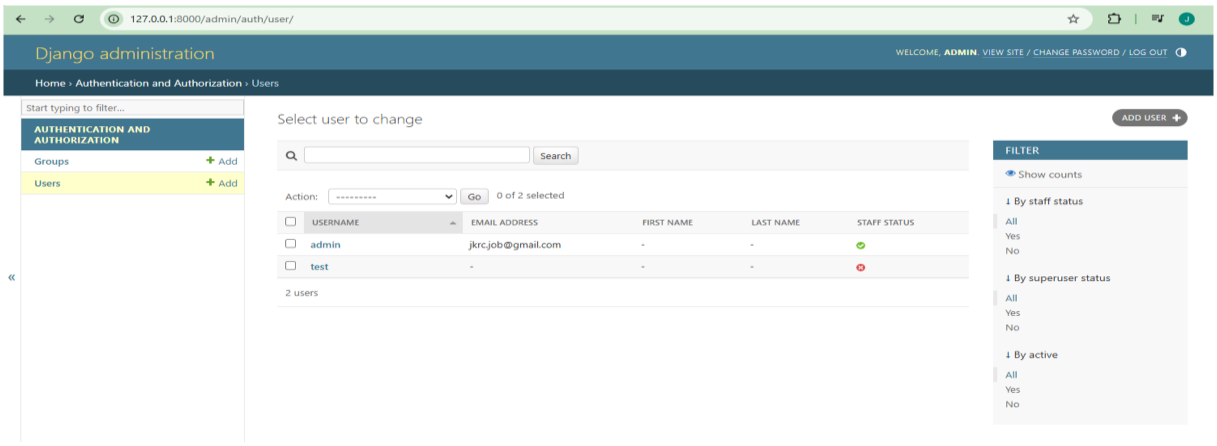
```
PS C:\Users\16358\Desktop\SZF2\Web developing\Academy\backend> python manage.py makemigrations
Migrations for 'api':
  api\migrations\0001_initial.py
    + Create model ActiveUser
    + Create model Reservation
    + Create model UserProfile
PS C:\Users\16358\Desktop\SZF2\Web developing\Academy\backend> python manage.py migrate
```

2.13 Create superuser admin account



```
PS C:\Users\16358\Desktop\SZF2\Web developing\Academy\backend> python manage.py createsuperuser
Username (leave blank to use '16358'): admin
Email address: jkrc.job@gmail.com
Password:
Password (again):
Superuser created successfully.
```

2.14 Login to Django administration



127.0.0.1:8000/admin/auth/user/

Django administration

WELCOME, ADMIN. VIEW SITE / CHANGE PASSWORD / LOG OUT

Home › Authentication and Authorization › Users

Select user to change

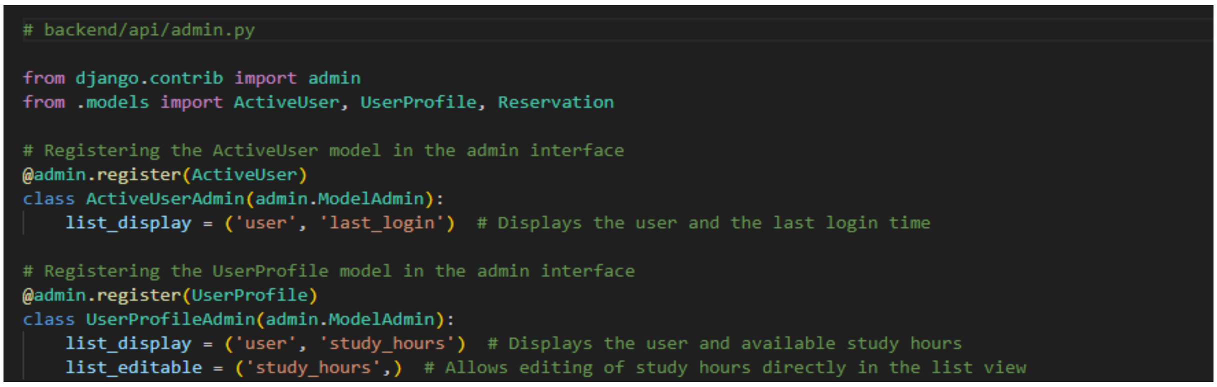
Action: --------- Go 0 of 2 selected

USERNAME	EMAIL ADDRESS	FIRST NAME	LAST NAME	STAFF STATUS
admin	jkrc.job@gmail.com	-	-	✓
test	-	-	-	✗

2 users

2.15 Create admin

ActiveUserAdmin: Customizes the admin interface for `ActiveUser` by displaying the `user` and `last_login` fields, making it easy to track when each user was last active.
UserProfileAdmin: Customizes the admin interface for `UserProfile` by displaying the `user` and `study_hours` fields and allowing inline editing of `study_hours` in the list view for quick adjustments.



```python
# backend/api/admin.py

from django.contrib import admin
from .models import ActiveUser, UserProfile, Reservation

# Registering the ActiveUser model in the admin interface
@admin.register(ActiveUser)
class ActiveUserAdmin(admin.ModelAdmin):
    list_display = ('user', 'last_login')  # Displays the user and the last login time

# Registering the UserProfile model in the admin interface
@admin.register(UserProfile)
class UserProfileAdmin(admin.ModelAdmin):
    list_display = ('user', 'study_hours')  # Displays the user and available study hours
    list_editable = ('study_hours',)  # Allows editing of study hours directly in the list view
```

`ActiveUser` Model:

Decorator:

- **`@admin.register(ActiveUser)`**: This decorator registers the ActiveUser model in the admin interface.

Class:

- **`class ActiveUserAdmin(admin.ModelAdmin)`**: Defines an admin class to customize how ActiveUser is displayed in the admin.

Step:

- **`list_display = ('user', 'last_login')`**: Specifies that the admin list view for `ActiveUser` should show the `user` and `last_login` fields, displaying each user along with their last login time.

`UserProfile` Model:

Decorator:

- **`@admin.register(UserProfile)`**: This decorator registers the `UserProfile` model in the admin interface.

Class:

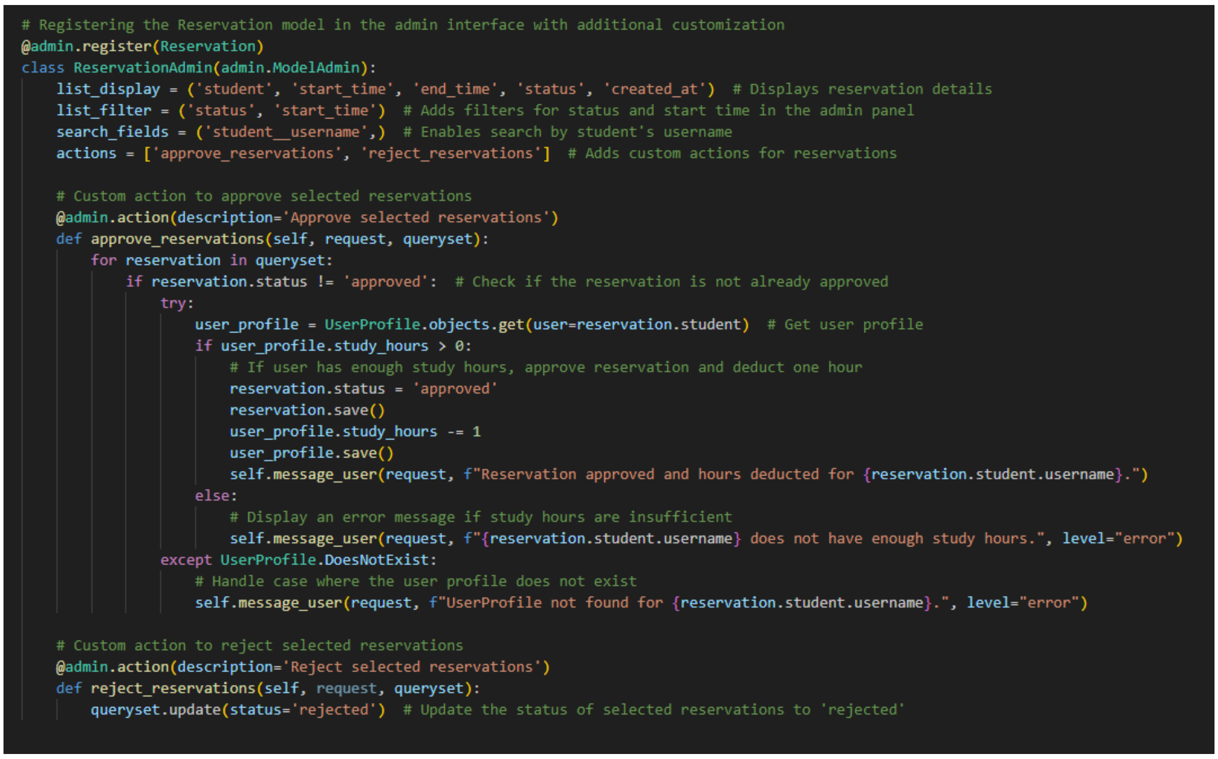
`class UserProfileAdmin(admin.ModelAdmin)`: Defines an admin class to customize how `UserProfile` is displayed in the admin.

Steps:

- **`list_display = ('user', 'study_hours')`**: Specifies that the admin list view for `UserProfile` should show the `user` and `study_hours` fields, displaying each user along with their available study hours.
- **`list_editable = ('study_hours',)`**: Allows the `study_hours` field to be edited directly in the list view, enabling quick adjustments to a user's study hours without needing to open the detail view for each entry.

ReservationAdmin:

- Customizes the list view of reservations in the Django admin interface with fields, filters, and search capabilities.
- Provides two custom actions:
  - **approve_reservations**: Approves reservations if the student has enough study hours, deducts one hour, and displays success or error messages as appropriate.
  - **reject_reservations**: Sets the status of selected reservations to `rejected` in bulk, simplifying the process of rejecting multiple reservations at once.



```python
# Registering the Reservation model in the admin interface with additional customization
@admin.register(Reservation)
class ReservationAdmin(admin.ModelAdmin):
    list_display = ('student', 'start_time', 'end_time', 'status', 'created_at')  # Displays reservation details
    list_filter = ('status', 'start_time')  # Adds filters for status and start time in the admin panel
    search_fields = ('student__username',)  # Enables search by student's username
    actions = ['approve_reservations', 'reject_reservations']  # Adds custom actions for reservations

    # Custom action to approve selected reservations
    @admin.action(description='Approve selected reservations')
    def approve_reservations(self, request, queryset):
        for reservation in queryset:
            if reservation.status != 'approved':  # Check if the reservation is not already approved
                try:
                    user_profile = UserProfile.objects.get(user=reservation.student)  # Get user profile
                    if user_profile.study_hours > 0:
                        # If user has enough study hours, approve reservation and deduct one hour
                        reservation.status = 'approved'
                        reservation.save()
                        user_profile.study_hours -= 1
                        user_profile.save()
                        self.message_user(request, f"Reservation approved and hours deducted for {reservation.student.username}.")
                    else:
                        # Display an error message if study hours are insufficient
                        self.message_user(request, f"{reservation.student.username} does not have enough study hours.", level="error")
                except UserProfile.DoesNotExist:
                    # Handle case where the user profile does not exist
                    self.message_user(request, f"UserProfile not found for {reservation.student.username}.", level="error")

    # Custom action to reject selected reservations
    @admin.action(description='Reject selected reservations')
    def reject_reservations(self, request, queryset):
        queryset.update(status='rejected')  # Update the status of selected reservations to 'rejected'
```

Decorator:

- **`@admin.register(Reservation)`**: Registers the `Reservation` model with custom configurations in the admin interface.

Class:

- **`class ReservationAdmin(admin.ModelAdmin)`**: Customizes how the `Reservation` model is displayed in the Django admin panel.

Steps:

- **`list_display`**: Specifies the fields to display in the list view of the admin panel, including `student`, `start_time`, `end_time`, `status`, and `created_at`.
- **`list_filter`**: Adds filtering options based on `status` and `start_time`, making it easier for admins to filter reservations.
- **`search_fields`**: Allows searching reservations by the `student__username`, enabling admins to quickly find reservations by a student's username.
- **`actions`**: Defines custom actions (`approve_reservations` and `reject_reservations`) that admins can apply to selected reservations in the admin panel.

Approve Selected Reservations

- **`@admin.action(description='Approve selected reservations')`**: Defines a custom admin action labeled "Approve selected reservations."

`approve_reservations`:

- Iterates through the selected reservations (`queryset`).
- Checks if the reservation's `status` is not already `approved`.
  - **Fetches User Profile**: Tries to retrieve the `UserProfile` of the student associated with the reservation.
  - **Check Study Hours**:
    - If the student has at least one study hour, it:
      - Sets the reservation `status` to `approved`.
      - Saves the reservation.
      - Deducts one hour from `user_profile.study_hours`.
      - Saves the updated `UserProfile`.
      - Displays a success message confirming the approval and hour deduction.
    - If the student lacks study hours, it displays an error message indicating insufficient study hours.
  - **Error Handling**: If the user profile doesn't exist, it displays an error message stating that the user profile was not found.

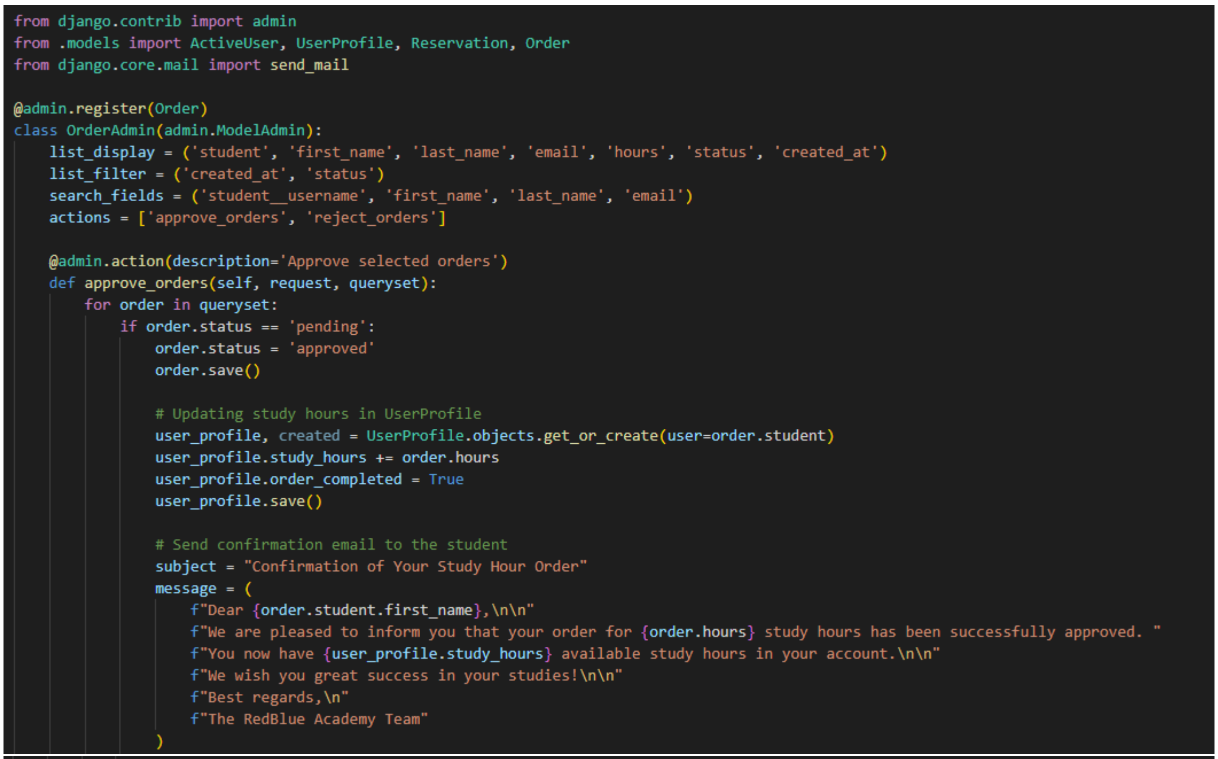
Reject Selected Reservations

- **`@admin.action(description='Reject selected reservations')`**: Defines a custom admin action labeled "Reject selected reservations."

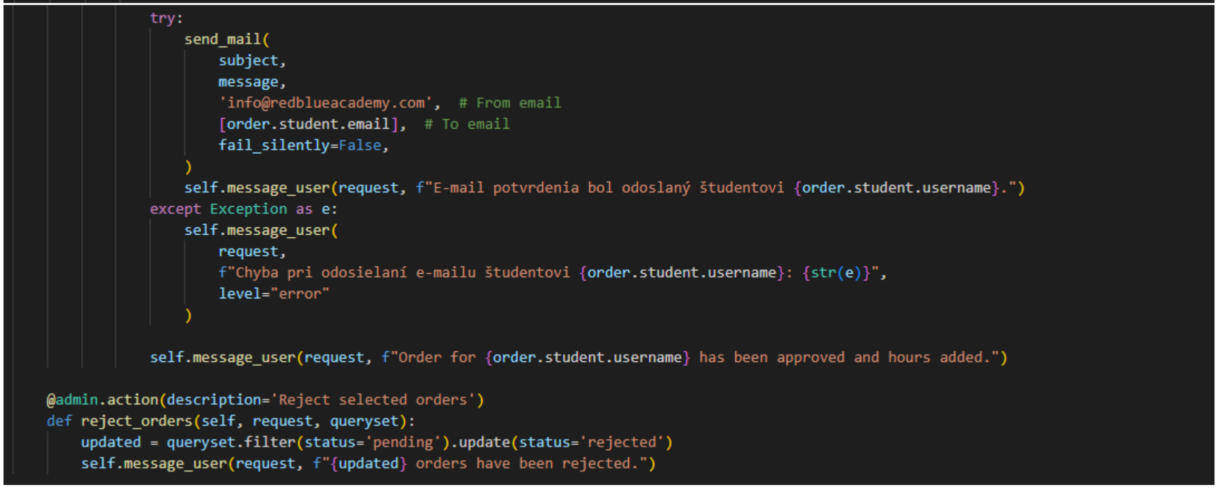
- **`reject_reservations`**:

  - Updates the `status` of all selected reservations in `queryset` to `rejected`.
  - This action allows the admin to reject multiple reservations in bulk with a single click.
- Admins can approve or reject orders.

- Upon approval, the hours from the order are added to the student’s profile.
- Upon rejection, the order status is updated to `rejected`.



```python
from django.contrib import admin
from .models import ActiveUser, UserProfile, Reservation, Order
from django.core.mail import send_mail

@admin.register(Order)
class OrderAdmin(admin.ModelAdmin):
    list_display = ('student', 'first_name', 'last_name', 'email', 'hours', 'status', 'created_at')
    list_filter = ('created_at', 'status')
    search_fields = ('student__username', 'first_name', 'last_name', 'email')
    actions = ['approve_orders', 'reject_orders']

    @admin.action(description='Approve selected orders')
    def approve_orders(self, request, queryset):
        for order in queryset:
            if order.status == 'pending':
                order.status = 'approved'
                order.save()

                # Updating study hours in UserProfile
                user_profile, created = UserProfile.objects.get_or_create(user=order.student)
                user_profile.study_hours += order.hours
                user_profile.order_completed = True
                user_profile.save()

                # Send confirmation email to the student
                subject = "Confirmation of Your Study Hour Order"
                message = (
                    f"Dear {order.student.first_name},\n\n"
                    f"We are pleased to inform you that your order for {order.hours} study hours has been successfully approved. "
                    f"You now have {user_profile.study_hours} available study hours in your account.\n\n"
                    f"We wish you great success in your studies!\n\n"
                    f"Best regards,\n"
                    f"The RedBlue Academy Team"
                )
```



```python
                try:
                    send_mail(
                        subject,
                        message,
                        'info@redblueacademy.com',  # From email
                        [order.student.email],  # To email
                        fail_silently=False,
                    )
                    self.message_user(request, f"E-mail potvrdenia bol odoslaný študentovi {order.student.username}.")
                except Exception as e:
                    self.message_user(
                        request,
                        f"Chyba pri odosielaní e-mailu študentovi {order.student.username}: {str(e)}",
                        level="error"
                    )

                self.message_user(request, f"Order for {order.student.username} has been approved and hours added.")

    @admin.action(description='Reject selected orders')
    def reject_orders(self, request, queryset):
        updated = queryset.filter(status='pending').update(status='rejected')
        self.message_user(request, f"{updated} orders have been rejected.")
```

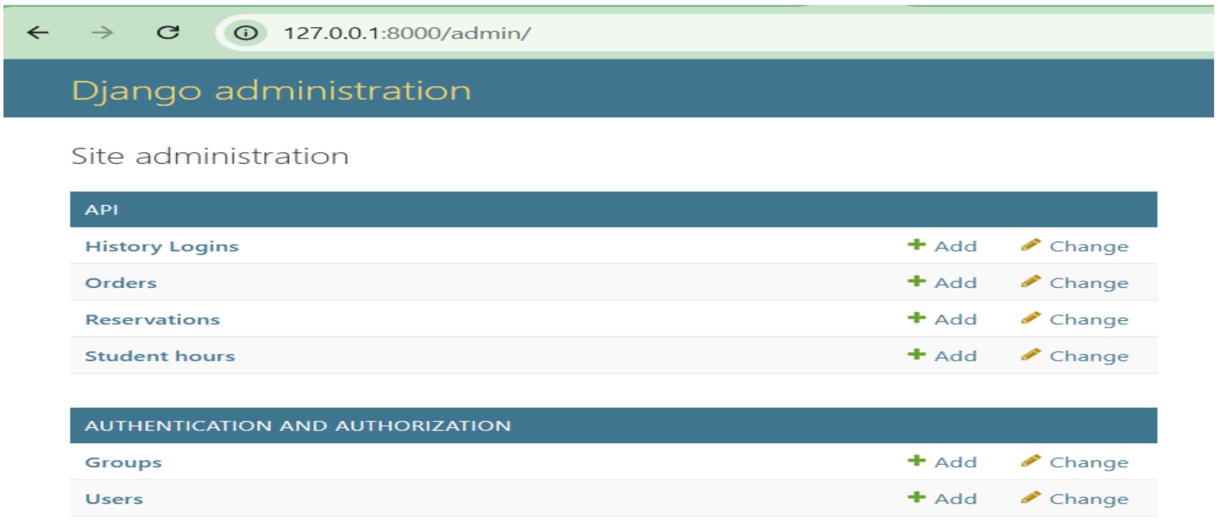
The `OrderAdmin` class provides the following functionalities for managing orders:

1. **Registering Orders:**
   - The `OrderAdmin` class is registered with the Django admin interface to allow management of the `Order` model.
2. **Display Configuration:**
   - **Fields Displayed:**
     - The `list_display` attribute ensures fields such as the student’s name, email, hours ordered, status, and creation time are shown in the admin interface.
   - **Filters and Search:**
     - The `list_filter` attribute adds filters for the `created_at` and `status` fields to streamline data filtering.

        - The `search_fields` attribute enables search functionality for fields like `username`, `first_name`, `last_name`, and `email`.

3. **Actions for Orders:**
    - Two custom actions, `approve_orders` and `reject_orders`, are provided for bulk processing of orders.
    - **Approving Orders:**
        - Updates the status of selected orders to `approved` if they are currently `pending`.
        - Adds the ordered study hours to the associated `UserProfile`.
        - Marks the `order_completed` flag in the `UserProfile` as `True`.
        - Sends a confirmation email to the student informing them about the approval and updated study hours.
        - Handles exceptions during email sending and displays appropriate messages in the admin panel.
    - **Rejecting Orders:**
        - Updates the status of selected orders to `rejected` for orders currently in the `pending` status.
        - Displays the number of orders rejected in the admin panel.
4. **Email Notifications:**
    - When an order is approved, an email is sent to the student's registered email address.
    - The email includes:
        - A subject line: **"Confirmation of Your Study Hour Order"**.
        - The body includes the student's first name, the number of hours ordered, and the updated total study hours in their account.
    - The email is sent using the configured email backend (`info@redblueacademy.com` as the sender).
5. **Error Handling:**
    - If the email fails to send, a detailed error message is displayed in the Django admin interface, ensuring transparency.
6. **Custom Messages in Admin Panel:**
    - After an action (approve/reject) is performed, a success message is displayed in the admin interface to inform the admin about the outcome.

Check administration if is all done



Site administration

API		
History Logins	Add	Change
Orders	Add	Change
Reservations	Add	Change
Student hours	Add	Change

AUTHENTICATION AND AUTHORIZATION		
Groups	Add	Change
Users	Add	Change

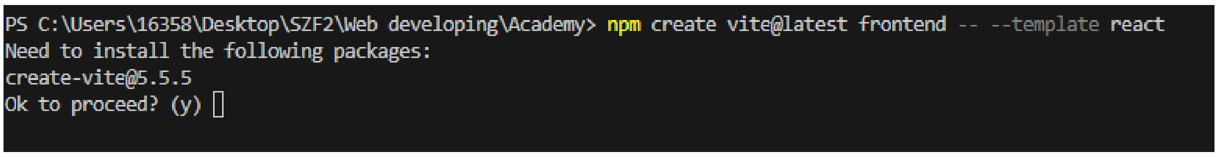
3 Frontend

3.1 Create Frontend

`npm create vite@latest`: This command creates a new project using the Vite tool. The `@latest` argument ensures that the latest version of Vite is used.

`frontend`: This is the name of the project or the folder where the project will be created. In this case, the project will be saved in a folder named **`frontend`**.

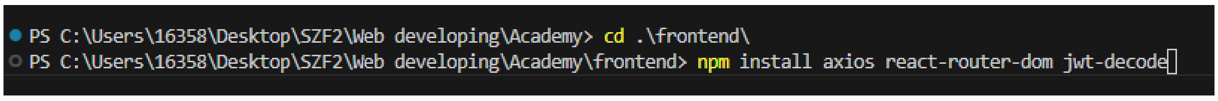
`-- --template react`: This specifies the `react` template, meaning Vite will configure the new project for React development. The template includes basic settings and packages necessary for a React application.



```
PS C:\Users\16358\Desktop\SZF2\Web developing\Academy> npm create vite@latest frontend -- --template react
Need to install the following packages:
create-vite@5.5.5
Ok to proceed? (y)
```

3.2 Install packages

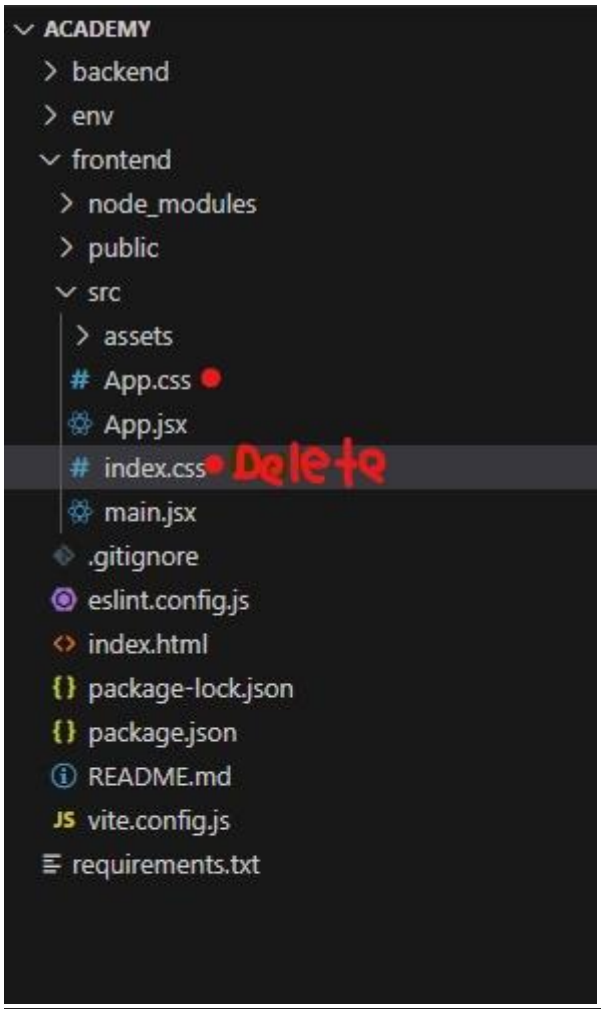
`cd .\frontend\`: Navigates into the `frontend` folder.
`npm install axios react-router-dom jwt-decode`: Installs `axios` for API requests, `react-router-dom` for handling routing, and `jwt-decode` for decoding JWT tokens, all essential tools for developing a React application with client-side routing and token-based authentication.



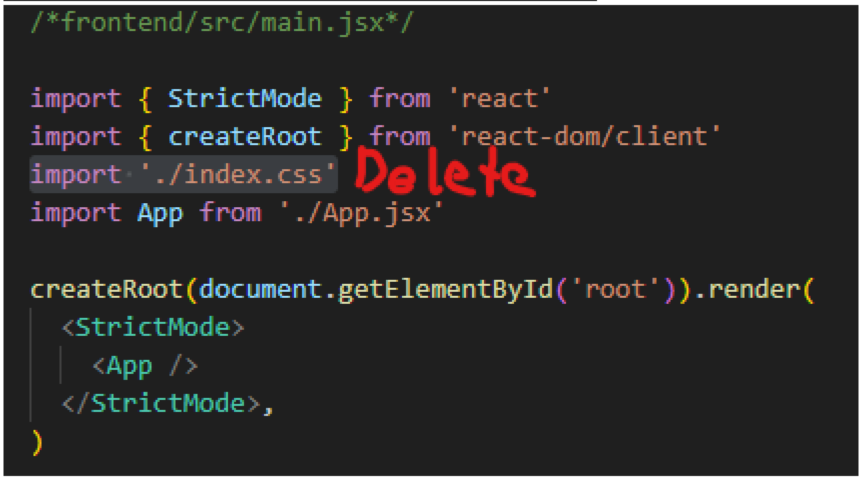
```
PS C:\Users\16358\Desktop\SZF2\Web developing\Academy> cd .\frontend\
PS C:\Users\16358\Desktop\SZF2\Web developing\Academy\frontend> npm install axios react-router-dom jwt-decode
```

3.3 Delete css

Unnecessary CSS files like index.css and App.css need to be deleted because styles will be added separately for each page.



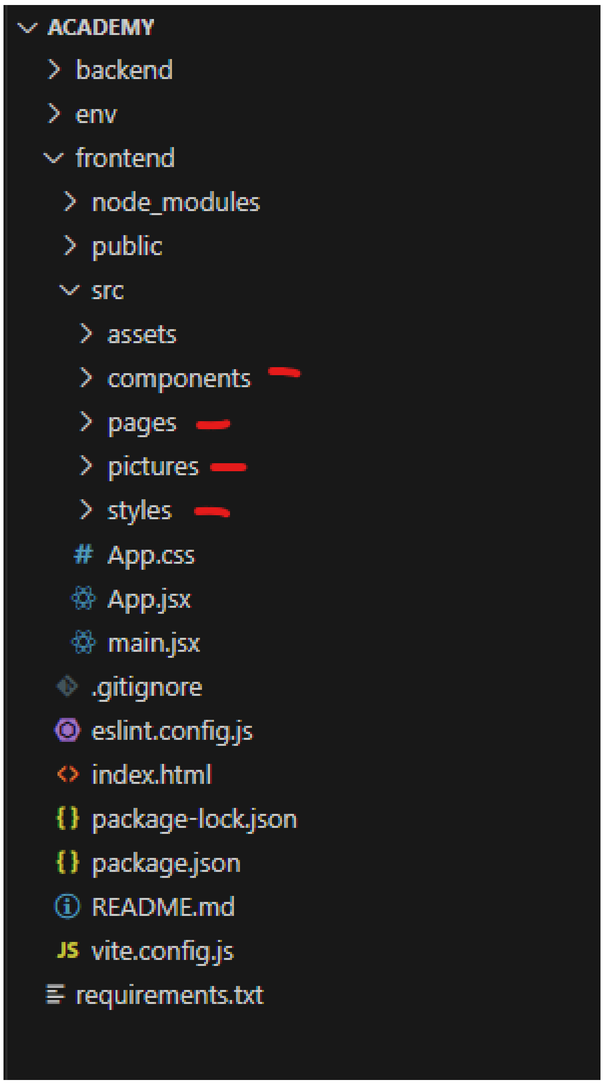
```
ACADEMY
  backend
  env
  frontend
    node_modules
    public
    src
      assets
      App.css
      App.jsx
      index.css
      main.jsx
    .gitignore
    eslint.config.js
    index.html
    package-lock.json
    package.json
    README.md
    vite.config.js
  requirements.txt
```



```
/*frontend/src/main.jsx*/

import { StrictMode } from 'react'
import { createRoot } from 'react-dom/client'
import './index.css'
import App from './App.jsx'

createRoot(document.getElementById('root')).render(
  <StrictMode>
    <App />
  </StrictMode>,
)
```

3.4 Create new folders

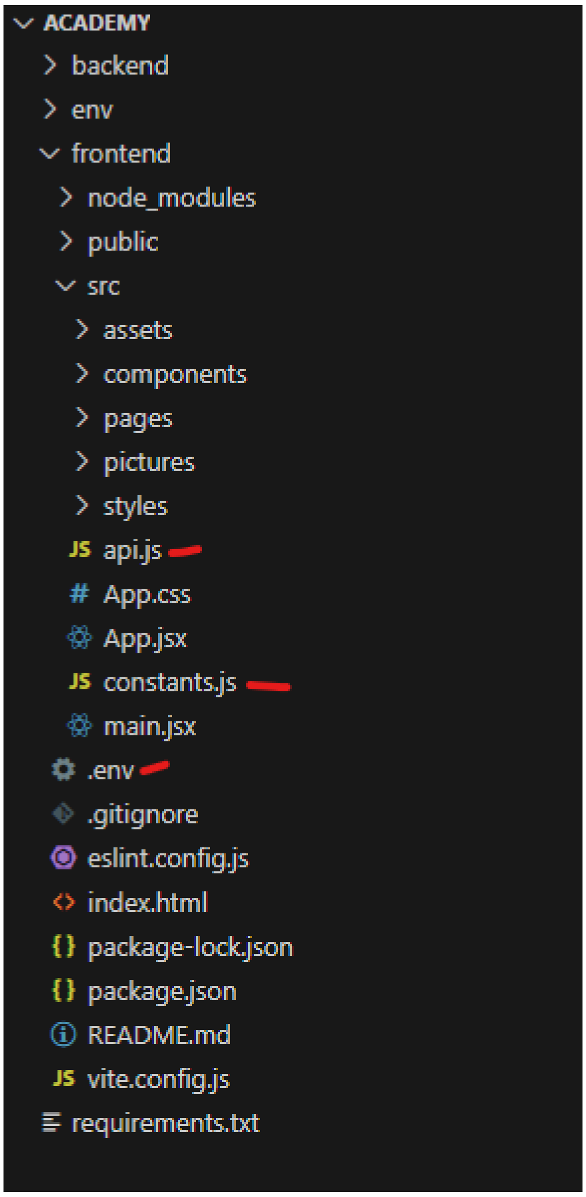
Create new folders in src.



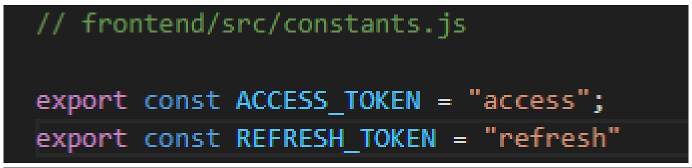
```
ACADEMY
  backend
  env
  frontend
    node_modules
    public
    src
      assets
      components
      pages
      pictures
      styles
      App.css
      App.jsx
      main.jsx
    .gitignore
    eslint.config.js
    index.html
    package-lock.json
    package.json
    README.md
    vite.config.js
  requirements.txt
```

3.5 Create new files

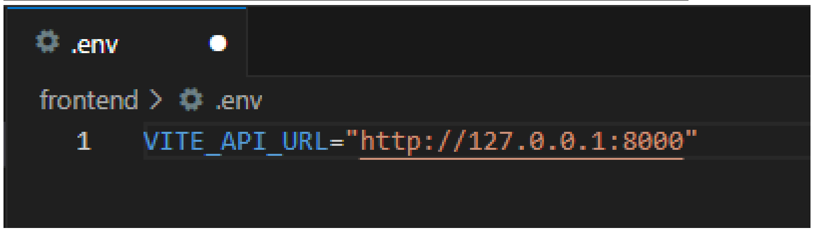
Create new files: api.js, constants.js, and .env, and in the constants.js file, add constants for storing and retrieving the access and refresh tokens. Also write URL for backendserver in .env



```
ACADEMY
  backend
  env
  frontend
    node_modules
    public
    src
      assets
      components
      pages
      pictures
      styles
      api.js
      App.css
      App.jsx
      constants.js
      main.jsx
    .env
    .gitignore
    eslint.config.js
    index.html
    package-lock.json
    package.json
    README.md
    vite.config.js
  requirements.txt
```



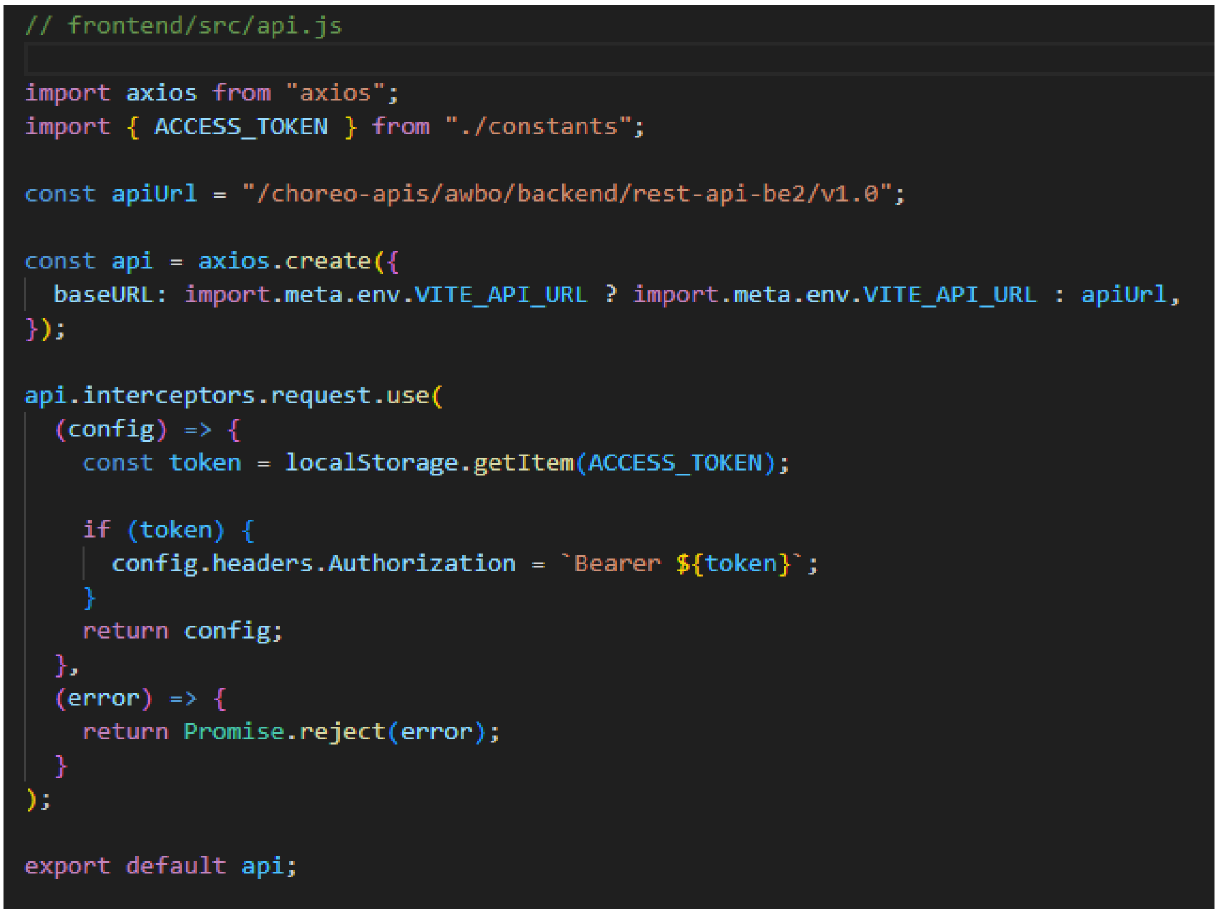
```js
// frontend/src/constants.js

export const ACCESS_TOKEN = "access";
export const REFRESH_TOKEN = "refresh"
```



```
VITE_API_URL="http://127.0.0.1:8000"
```

3.6 Create API

- This file sets up an Axios instance configured with a base URL for making requests to an API.
- An interceptor checks if an access token is available in `localStorage` and, if present, includes it in the `Authorization` header of each request.
- This setup enables secure communication with the backend API, as requests are authenticated when a token is provided.



```javascript
// frontend/src/api.js

import axios from "axios";
import { ACCESS_TOKEN } from "./constants";

const apiUrl = "/choreo-apis/awbo/backend/rest-api-be2/v1.0";

const api = axios.create({
  baseURL: import.meta.env.VITE_API_URL ? import.meta.env.VITE_API_URL : apiUrl,
});

api.interceptors.request.use(
  (config) => {
    const token = localStorage.getItem(ACCESS_TOKEN);

    if (token) {
      config.headers.Authorization = `Bearer ${token}`;
    }
    return config;
  },
  (error) => {
    return Promise.reject(error);
  }
);

export default api;
```

```javascript
import axios from "axios";
import { ACCESS_TOKEN } from "./constants";
```

- **`import axios from "axios";`**: Imports Axios, a library for making HTTP requests.
- **`import { ACCESS_TOKEN } from "./constants";`**: Imports the `ACCESS_TOKEN` constant, likely a key used to retrieve the access token from `localStorage`.

```javascript
const apiUrl = "/choreo-apis/awbo/backend/rest-api-be2/v1.0";
```

`const apiUrl`: Defines a default URL for the API endpoint. This URL is used if the environment variable (`VITE_API_URL`) is not set.

```
const api = axios.create({
  baseURL: import.meta.env.VITE_API_URL ? import.meta.env.VITE_API_URL : apiUrl,
});
```

`axios.create()`: Creates a new Axios instance with a specified base URL.

- **`baseURL`**: Checks if `VITE_API_URL` (an environment variable) is set. If so, it uses that as the `baseURL`. If not, it defaults to `apiUrl`.

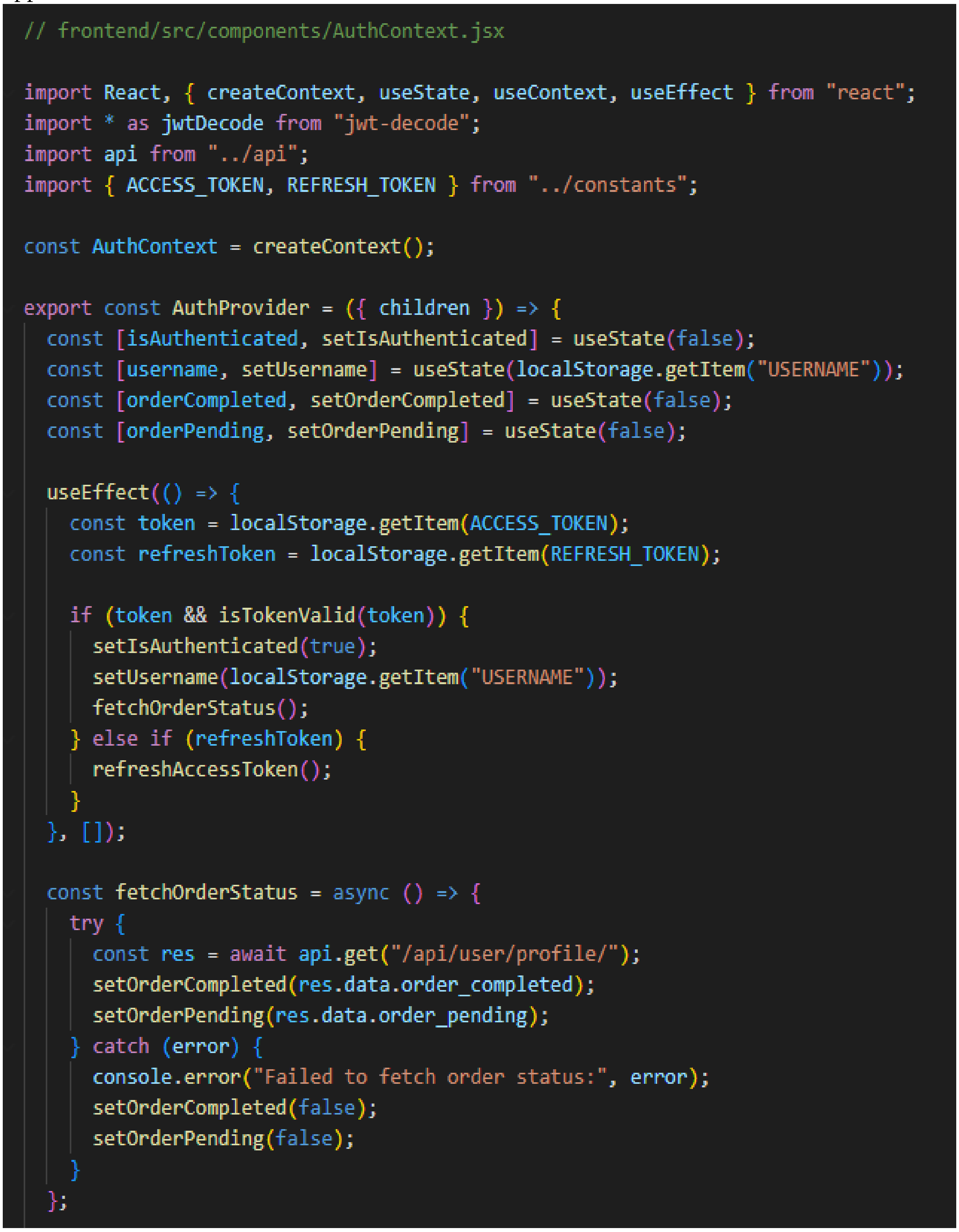
```
api.interceptors.request.use(
  (config) => {
    const token = localStorage.getItem(ACCESS_TOKEN);

    if (token) {
      config.headers.Authorization = `Bearer ${token}`;
    }
    return config;
  },
  (error) => {
    return Promise.reject(error);
  }
);
```

`api.interceptors.request.use`: Sets up an interceptor for requests made with the `api` instance.

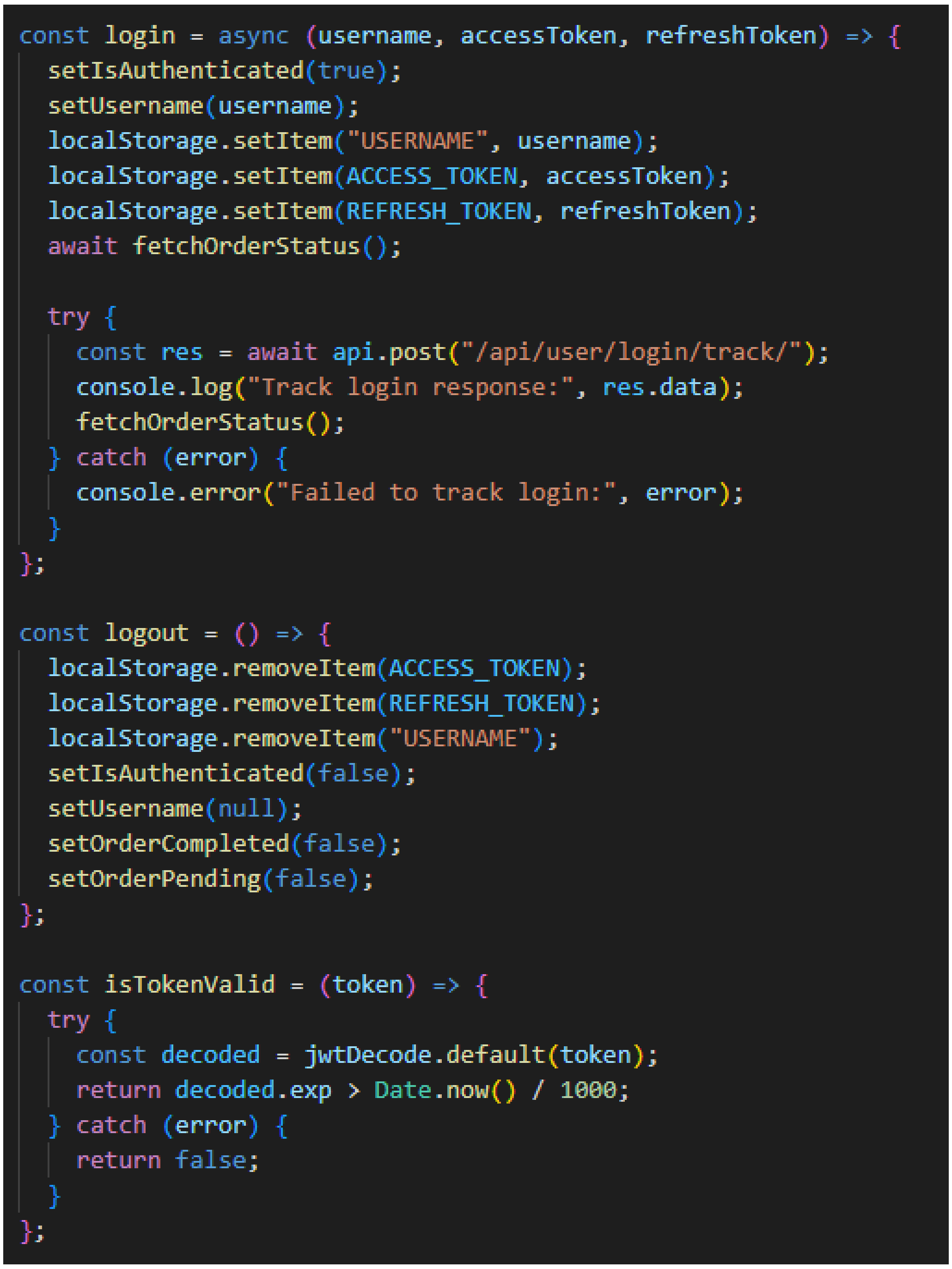
- **`config`**: The configuration of each request. This function is executed before each request is sent.
- **`const token = localStorage.getItem(ACCESS_TOKEN);`**: Retrieves the access token from `localStorage` using the `ACCESS_TOKEN` constant as the key.
- **`if (token)`**: Checks if a token is available.
  - **`config.headers.Authorization =` Bearer ${token}`;`**: If a token exists, it adds an `Authorization` header to the request with the format `Bearer <token>`. This header is typically required for authenticated requests.
- **`return config;`**: Returns the modified configuration to proceed with the request.
- **Error Handling**:
  - **`(error) => { return Promise.reject(error); }`**: If there is an error while setting up the request, it returns a rejected promise with the error.

3.7 Create Authentication

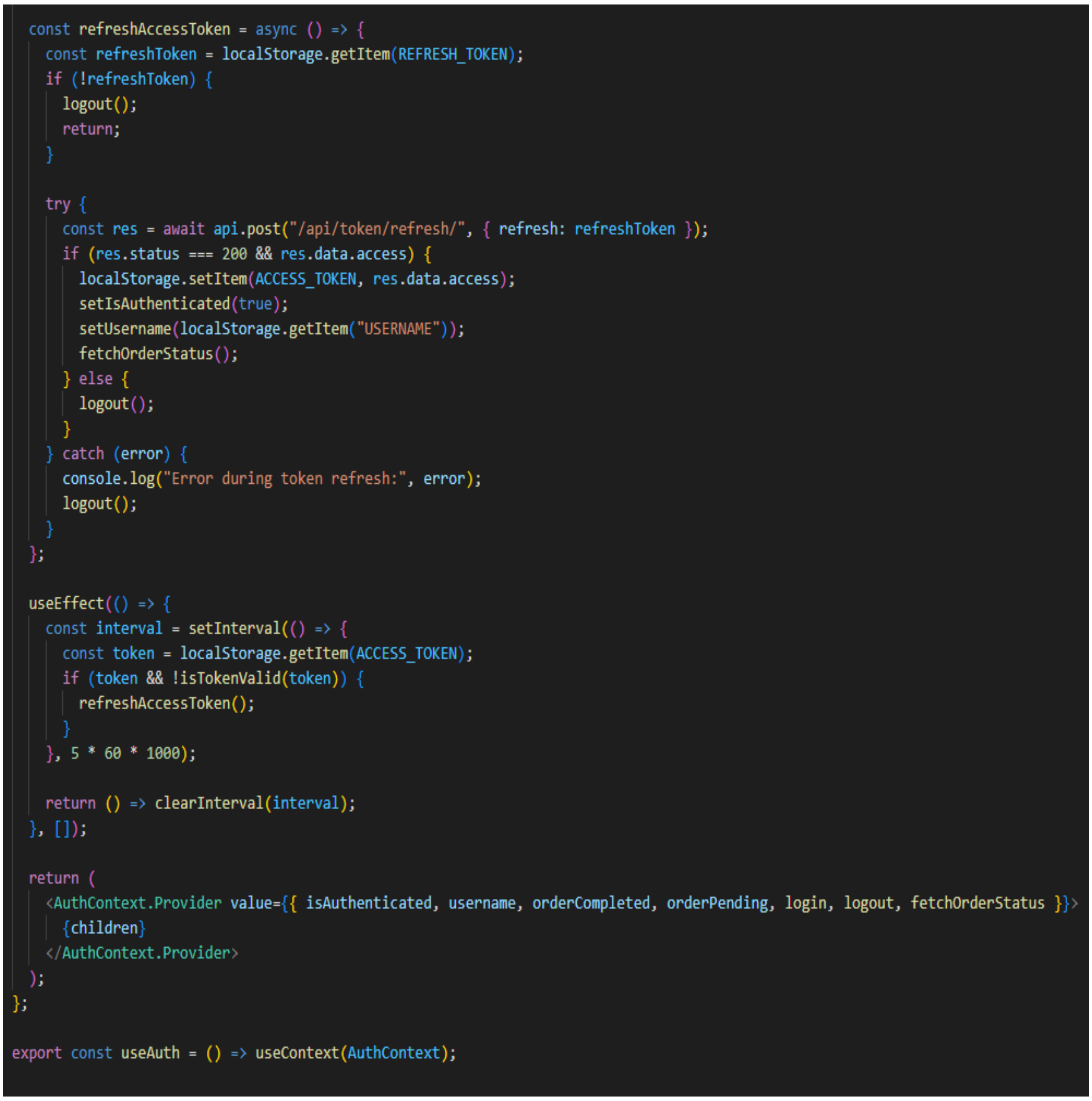
- **AuthContext**: Holds authentication state and functions for login, logout, and token management.
- **AuthProvider**: Wraps the app, enabling authentication context across components.
- **Login and Logout**: `login` saves authentication details and `logout` clears them.
- **Token Handling**: Checks token validity and refreshes it if necessary during the initial app load.



```jsx
// frontend/src/components/AuthContext.jsx

import React, { createContext, useState, useContext, useEffect } from "react";
import * as jwtDecode from "jwt-decode";
import api from "../api";
import { ACCESS_TOKEN, REFRESH_TOKEN } from "../constants";

const AuthContext = createContext();

export const AuthProvider = ({ children }) => {
  const [isAuthenticated, setIsAuthenticated] = useState(false);
  const [username, setUsername] = useState(localStorage.getItem("USERNAME"));
  const [orderCompleted, setOrderCompleted] = useState(false);
  const [orderPending, setOrderPending] = useState(false);

  useEffect(() => {
    const token = localStorage.getItem(ACCESS_TOKEN);
    const refreshToken = localStorage.getItem(REFRESH_TOKEN);

    if (token && isTokenValid(token)) {
      setIsAuthenticated(true);
      setUsername(localStorage.getItem("USERNAME"));
      fetchOrderStatus();
    } else if (refreshToken) {
      refreshAccessToken();
    }
  }, []);

  const fetchOrderStatus = async () => {
    try {
      const res = await api.get("/api/user/profile/");
      setOrderCompleted(res.data.order_completed);
      setOrderPending(res.data.order_pending);
    } catch (error) {
      console.error("Failed to fetch order status:", error);
      setOrderCompleted(false);
      setOrderPending(false);
    }
  };
```



```
const login = async (username, accessToken, refreshToken) => {
  setIsAuthenticated(true);
  setUsername(username);
  localStorage.setItem("USERNAME", username);
  localStorage.setItem(ACCESS_TOKEN, accessToken);
  localStorage.setItem(REFRESH_TOKEN, refreshToken);
  await fetchOrderStatus();

  try {
    const res = await api.post("/api/user/login/track/");
    console.log("Track login response:", res.data);
    fetchOrderStatus();
  } catch (error) {
    console.error("Failed to track login:", error);
  }
};

const logout = () => {
  localStorage.removeItem(ACCESS_TOKEN);
  localStorage.removeItem(REFRESH_TOKEN);
  localStorage.removeItem("USERNAME");
  setIsAuthenticated(false);
  setUsername(null);
  setOrderCompleted(false);
  setOrderPending(false);
};

const isTokenValid = (token) => {
  try {
    const decoded = jwtDecode.default(token);
    return decoded.exp > Date.now() / 1000;
  } catch (error) {
    return false;
  }
};
```



```
  const refreshAccessToken = async () => {
    const refreshToken = localStorage.getItem(REFRESH_TOKEN);
    if (!refreshToken) {
      logout();
      return;
    }

    try {
      const res = await api.post("/api/token/refresh/", { refresh: refreshToken });
      if (res.status === 200 && res.data.access) {
        localStorage.setItem(ACCESS_TOKEN, res.data.access);
        setIsAuthenticated(true);
        setUsername(localStorage.getItem("USERNAME"));
        fetchOrderStatus();
      } else {
        logout();
      }
    } catch (error) {
      console.log("Error during token refresh:", error);
      logout();
    }
  };

  useEffect(() => {
    const interval = setInterval(() => {
      const token = localStorage.getItem(ACCESS_TOKEN);
      if (token && !isTokenValid(token)) {
        refreshAccessToken();
      }
    }, 5 * 60 * 1000);

    return () => clearInterval(interval);
  }, []);

  return (
    <AuthContext.Provider value={{ isAuthenticated, username, orderCompleted, orderPending, login, logout, fetchOrderStatus }}>
      {children}
    </AuthContext.Provider>
  );
};

export const useAuth = () => useContext(AuthContext);
```

```
import React, { createContext, useState, useContext, useEffect } from "react";
import * as jwtDecode from "jwt-decode";
import api from "../api";
import { ACCESS_TOKEN, REFRESH_TOKEN } from "../constants";
```

- **`createContext`**: Creates a context object for managing global state in React.
- **`useState`** and **`useEffect`**: React hooks for managing state and side effects.
- **`useContext`**: Allows components to access the context's value.
- **`jwtDecode`**: Utility to decode JSON Web Tokens (JWT) to read their contents.
- **`api`**: Axios instance for API requests.
- **`ACCESS_TOKEN`** and **`REFRESH_TOKEN`**: Constants representing the keys to store tokens in `localStorage`.

```
const AuthContext = createContext();
```

- **`AuthContext`**: This is the context that holds the authentication state and functions, enabling access across the app.

```
export const AuthProvider = ({ children }) => {
  const [isAuthenticated, setIsAuthenticated] = useState(false);
  const [username, setUsername] = useState(localStorage.getItem("USERNAME"));
  const [orderCompleted, setOrderCompleted] = useState(false);
  const [orderPending, setOrderPending] = useState(false);
```

The `AuthProvider` component acts as a provider for `AuthContext`, offering access to authentication states and functions for all child components.

State Management

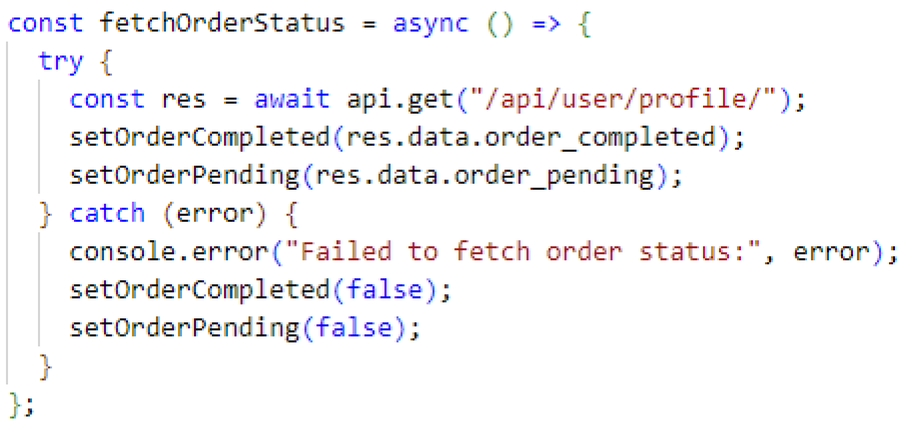
- **`isAuthenticated`**: Tracks if the user is authenticated. Initially set to `false`.
- **`username`**: Stores the authenticated user's username, retrieved from `localStorage` if available.
- **`orderCompleted`**: Tracks if a user's order is completed, initially set to `false`.
- **`orderPending`**: Tracks if a user's order is pending, initially set to `false`.

```
useEffect(() => {
  const token = localStorage.getItem(ACCESS_TOKEN);
  const refreshToken = localStorage.getItem(REFRESH_TOKEN);

  if (token && isTokenValid(token)) {
    setIsAuthenticated(true);
    setUsername(localStorage.getItem("USERNAME"));
    fetchOrderStatus();
  } else if (refreshToken) {
    refreshAccessToken();
  }
}, []);
```

The first `useEffect` hook executes on component mount. Its main role is to check for existing authentication tokens in `localStorage` and set the user's authentication state accordingly.

- **Retrieve Tokens**:
  - `token`: Retrieves the access token from `localStorage` using the `ACCESS_TOKEN` key.
  - `refreshToken`: Retrieves the refresh token using the `REFRESH_TOKEN` key.
- **Token Validation**:
  - If the `token` is found and is valid (checked via `isTokenValid(token)`), it updates the `isAuthenticated` state to `true`, retrieves the username from

localStorage, and calls fetchOrderStatus() to get the user's current order status.
   - If the token is invalid but a refreshToken is available, it calls refreshAccessToken() to attempt refreshing the access token.



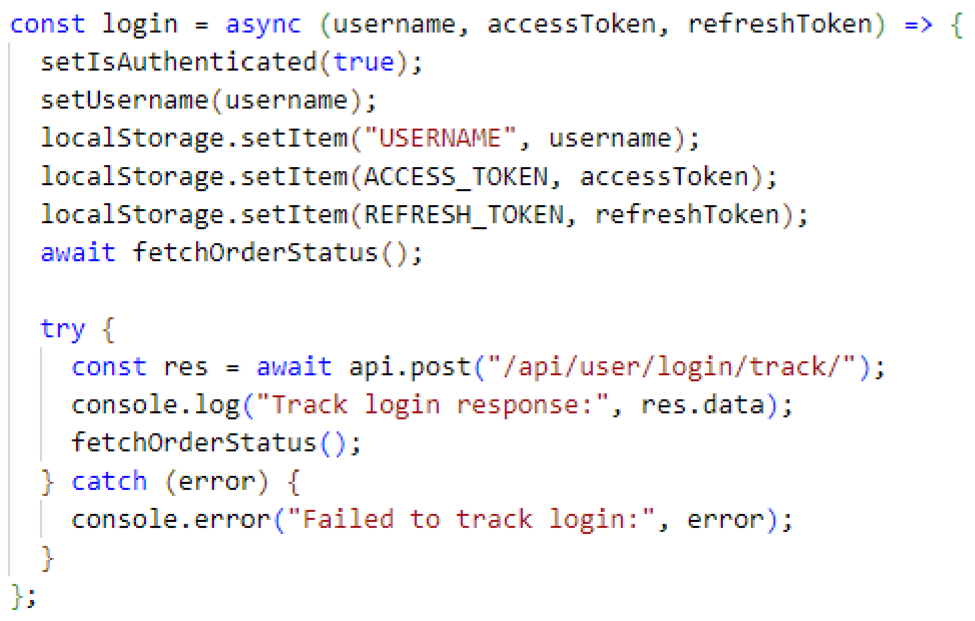
```
const fetchOrderStatus = async () => {
  try {
    const res = await api.get("/api/user/profile/");
    setOrderCompleted(res.data.order_completed);
    setOrderPending(res.data.order_pending);
  } catch (error) {
    console.error("Failed to fetch order status:", error);
    setOrderCompleted(false);
    setOrderPending(false);
  }
};
```

fetchOrderStatus Function

This function is responsible for retrieving the user's order status from the backend.

- **Asynchronous Call**: The function is asynchronous, using async and await to handle the API request.
- **API Request**: It sends a GET request to /api/user/profile/, which presumably returns data about the user's order status.
- **Set State Based on Response**:
  - setOrderCompleted: Sets the orderCompleted state to true or false, depending on the order_completed field in the response.
  - setOrderPending: Similarly, updates the orderPending state based on the order_pending field in the response.
- **Error Handling**: If the request fails, an error is logged to the console, and orderCompleted and orderPending are set to false as a fallback.

This function provides the current status of the user's orders, allowing the app to display information related to completed or pending orders accurately.



```
const login = async (username, accessToken, refreshToken) => {
  setIsAuthenticated(true);
  setUsername(username);
  localStorage.setItem("USERNAME", username);
  localStorage.setItem(ACCESS_TOKEN, accessToken);
  localStorage.setItem(REFRESH_TOKEN, refreshToken);
  await fetchOrderStatus();

  try {
    const res = await api.post("/api/user/login/track/");
    console.log("Track login response:", res.data);
    fetchOrderStatus();
  } catch (error) {
    console.error("Failed to track login:", error);
  }
};
```

`login` Function

The `login` function handles the user authentication process and token storage.

- **Parameters**:
  - `username`: The username of the user.
  - `accessToken` and `refreshToken`: Tokens received upon successful login, used for authentication and session management.
- **Authentication State Update**:
  - `setIsAuthenticated(true)`: Marks the user as authenticated.
  - `setUsername(username)`: Updates the `username` state with the provided username.
- **Store Tokens and Username**:
  - Stores `username`, `accessToken`, and `refreshToken` in `localStorage` to persist the user's session across page reloads.
- **Fetch Order Status**: Calls `fetchOrderStatus()` to update the `orderCompleted` and `orderPending` states immediately after login.
- **Track Login Activity**:
  - Sends a `POST` request to `/api/user/login/track/` to log the login activity on the server.
  - Logs the response or error to the console for debugging.

The `login` function establishes the user's authenticated session, stores session data, and immediately retrieves the user's order status.

```
const logout = () => {
  localStorage.removeItem(ACCESS_TOKEN);
  localStorage.removeItem(REFRESH_TOKEN);
  localStorage.removeItem("USERNAME");
  setIsAuthenticated(false);
  setUsername(null);
  setOrderCompleted(false);
  setOrderPending(false);
};
```

`logout` Function

The `logout` function clears the user's authentication and resets related states.

- **Clear Local Storage**: Removes `ACCESS_TOKEN`, `REFRESH_TOKEN`, and `USERNAME` from `localStorage` to invalidate the session.
- **Reset Authentication State**:
  - `setIsAuthenticated(false)`: Sets the authentication state to `false`.
  - `setUsername(null)`: Clears the username state.
  - `setOrderCompleted(false)` and `setOrderPending(false)`: Resets order-related states to indicate no active orders.

The `logout` function ensures that all session data is removed and the app's state reflects the user's logged-out status.

```
const isTokenValid = (token) => {
  try {
    const decoded = jwtDecode.default(token);
    return decoded.exp > Date.now() / 1000;
  } catch (error) {
    return false;
  }
};
```

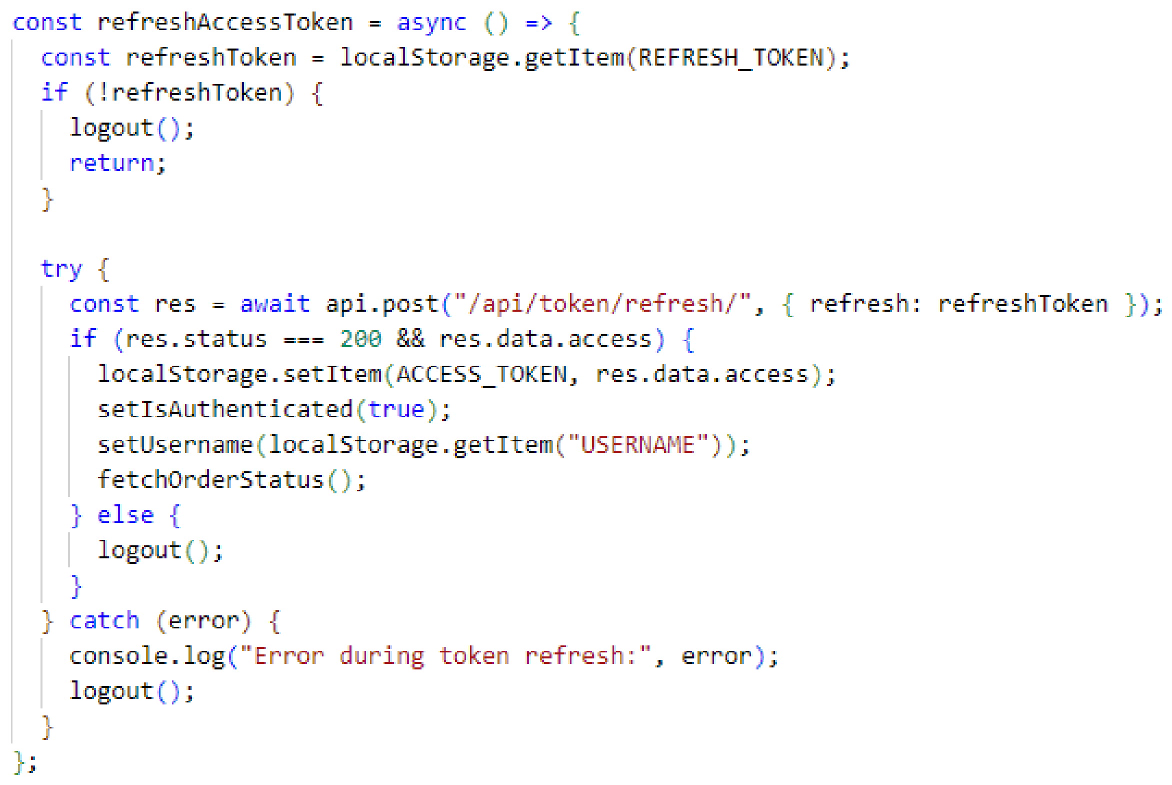
`isTokenValid` Function

This function checks whether a given token is valid based on its expiration time.

- **Parameter**:
  - `token`: The JWT (JSON Web Token) that needs validation.
- **Token Decoding**:
  - The function uses `jwtDecode` to decode the `token` and extract its payload, which includes an `exp` (expiration) timestamp.
- **Expiration Check**:
  - The function compares the `exp` timestamp to the current time (`Date.now() / 1000` to convert milliseconds to seconds). If the expiration time is greater than the current time, the token is still valid.
- **Error Handling**:

- If decoding fails (e.g., the token is malformed or missing), the function catches the error and returns `false`, marking the token as invalid.

This function helps determine if the current access token can still be used or if it needs to be refreshed.



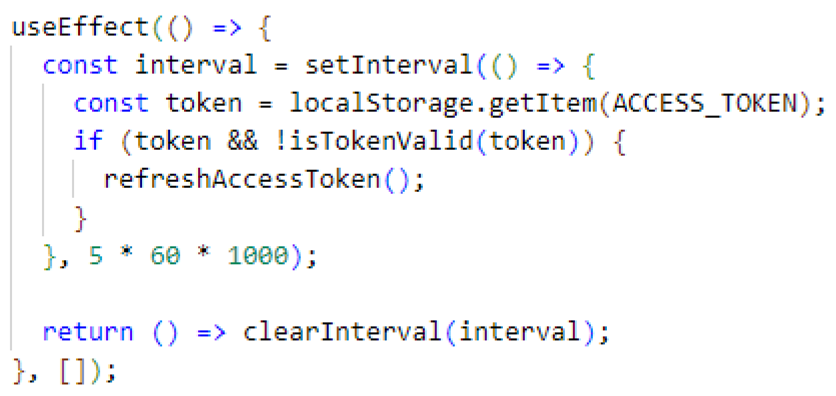
```
const refreshAccessToken = async () => {
  const refreshToken = localStorage.getItem(REFRESH_TOKEN);
  if (!refreshToken) {
    logout();
    return;
  }

  try {
    const res = await api.post("/api/token/refresh/", { refresh: refreshToken });
    if (res.status === 200 && res.data.access) {
      localStorage.setItem(ACCESS_TOKEN, res.data.access);
      setIsAuthenticated(true);
      setUsername(localStorage.getItem("USERNAME"));
      fetchOrderStatus();
    } else {
      logout();
    }
  } catch (error) {
    console.log("Error during token refresh:", error);
    logout();
  }
};
```

`refreshAccessToken` Function

This function attempts to refresh the user's access token using a stored refresh token.

- **Retrieve Refresh Token**:
  - The function retrieves the `refreshToken` from `localStorage`.
  - If the `refreshToken` is not found, the function calls `logout()` to end the session and returns.
- **Token Refresh Request**:
  - Sends a `POST` request to `/api/token/refresh/`, passing the `refreshToken` as payload. This endpoint is expected to return a new access token if the refresh token is valid.
- **Response Handling**:
  - If the response status is `200` and a new `access` token is present in the response:
    - The new access token is stored in `localStorage`.
    - `setIsAuthenticated(true)`: Sets the authentication state to `true`.
    - `setUsername(localStorage.getItem("USERNAME"))`: Ensures the username is updated from local storage.
    - `fetchOrderStatus()`: Calls this function to update the order-related states.
  - If the refresh attempt fails (e.g., due to an expired or invalid refresh token), the function calls `logout()` to clear session data and mark the user as logged out.
- **Error Handling**:

  - If the API request fails for any reason, an error message is logged, and the function calls `logout()` to clear any stale session data.



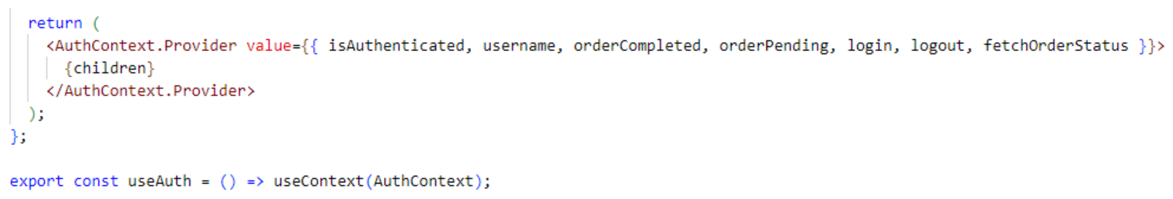
```
useEffect(() => {
  const interval = setInterval(() => {
    const token = localStorage.getItem(ACCESS_TOKEN);
    if (token && !isTokenValid(token)) {
      refreshAccessToken();
    }
  }, 5 * 60 * 1000);

  return () => clearInterval(interval);
}, []);
```

`useEffect` Hook for Token Refreshing

This `useEffect` hook sets up a periodic check to ensure that the access token remains valid and refreshes it if necessary.

- **Purpose**:
  - The purpose of this `useEffect` is to periodically check if the access token stored in `localStorage` is still valid. If it's invalid or expired, the function will attempt to refresh it by calling `refreshAccessToken`.
- **Set Interval**:
  - `setInterval(() => {...}, 5 * 60 * 1000)`: Sets up an interval that runs every 5 minutes (300,000 milliseconds).
- **Token Check and Refresh**:
  - Inside the interval, the function retrieves the `ACCESS_TOKEN` from `localStorage`.
  - It checks if the token exists and is invalid (using `isTokenValid`). If the token is invalid, it calls `refreshAccessToken` to try and get a new access token using the refresh token.
- **Cleanup**:
  - The `useEffect` hook returns a cleanup function that clears the interval when the component is unmounted or the hook is rerun. This ensures that the interval doesn't keep running in the background if the component is removed from the DOM.

This hook helps maintain the user's session by automatically refreshing the access token every 5 minutes, as long as the user remains active on the page.



```
  return (
    <AuthContext.Provider value={{ isAuthenticated, username, orderCompleted, orderPending, login, logout, fetchOrderStatus }}>
      {children}
    </AuthContext.Provider>
  );
};

export const useAuth = () => useContext(AuthContext);
```

`AuthContext.Provider`

This component returns an `AuthContext.Provider` that wraps around the children components, giving them access to the authentication context values.

- **Context Values Provided**:
  - `isAuthenticated`: A boolean that indicates if the user is logged in.
  - `username`: The username of the logged-in user.
  - `orderCompleted`: Boolean indicating if the order is completed.
  - `orderPending`: Boolean indicating if an order is pending.
  - `login`: Function to log in the user and set the authentication state.
  - `logout`: Function to log out the user and clear the authentication state.
  - `fetchOrderStatus`: Function to fetch the current order status.

By wrapping the children components in `AuthContext.Provider`, these values and functions are accessible to any component within the app that uses this context.

`useAuth` Hook

The `useAuth` hook is a custom hook that allows other components to easily access the values in `AuthContext` using `useContext`.

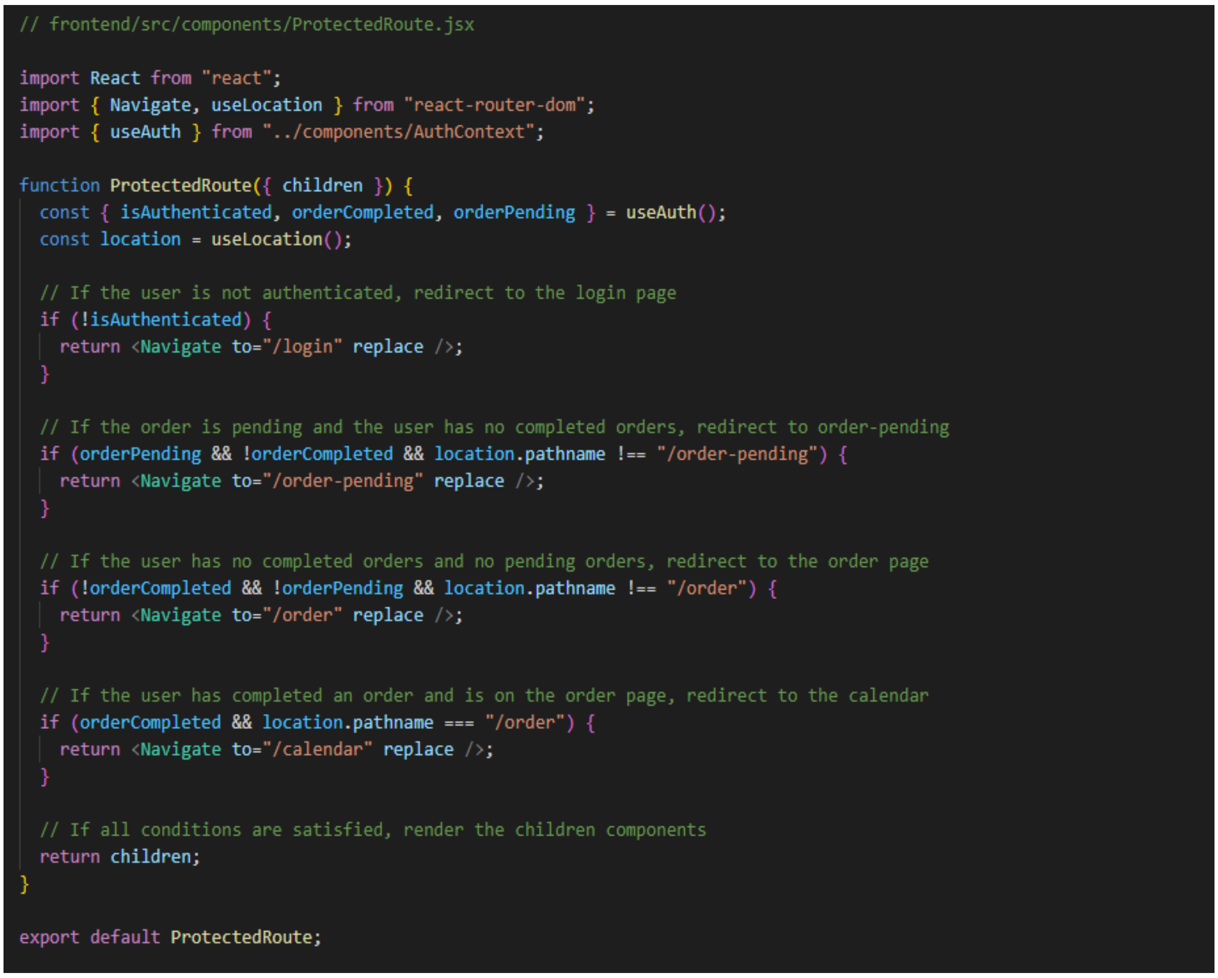
- **Purpose**:
  - This hook simplifies access to `AuthContext` by allowing components to call `useAuth()` directly instead of `useContext(AuthContext)`, improving code readability and encapsulating the context logic.

3.8 Create Protected Route

The `ProtectedRoute` component now not only controls access based on the user's authentication status but also incorporates logic related to the user's order status. It performs the following actions:

1. Redirects unauthenticated users to the login page.
2. Redirects users with pending orders to a pending order page.
3. Redirects users without completed orders to the order page if they do not have a pending order.
4. Redirects users with completed orders away from the order page to the calendar page.
5. Allows users with completed orders or valid pending orders to access protected routes.

This approach ensures that only users with the appropriate authentication and order status can access specific parts of the application.



```jsx
// frontend/src/components/ProtectedRoute.jsx

import React from "react";
import { Navigate, useLocation } from "react-router-dom";
import { useAuth } from "../components/AuthContext";

function ProtectedRoute({ children }) {
  const { isAuthenticated, orderCompleted, orderPending } = useAuth();
  const location = useLocation();

  // If the user is not authenticated, redirect to the login page
  if (!isAuthenticated) {
    return <Navigate to="/login" replace />;
  }

  // If the order is pending and the user has no completed orders, redirect to order-pending
  if (orderPending && !orderCompleted && location.pathname !== "/order-pending") {
    return <Navigate to="/order-pending" replace />;
  }

  // If the user has no completed orders and no pending orders, redirect to the order page
  if (!orderCompleted && !orderPending && location.pathname !== "/order") {
    return <Navigate to="/order" replace />;
  }

  // If the user has completed an order and is on the order page, redirect to the calendar
  if (orderCompleted && location.pathname === "/order") {
    return <Navigate to="/calendar" replace />;
  }

  // If all conditions are satisfied, render the children components
  return children;
}

export default ProtectedRoute;
```

Imports:

- `React`: Required to define the component.
- `{ Navigate, useLocation }` from `react-router-dom`: `Navigate` is used to redirect users to specific routes, and `useLocation` provides the current location/path, allowing for conditional redirects based on the user's current page.
- `{ useAuth }` from `../components/AuthContext`: This custom hook provides the authentication and order statuses (such as `isAuthenticated`, `orderCompleted`, and `orderPending`) from `AuthContext`.

ProtectedRoute Component

- **Function Definition**:
  - `function ProtectedRoute({ children })`: Defines the component that accepts `children` as a prop, representing the nested components that should be displayed if the access conditions are met.
- **Authentication Check**:
  - `if (!isAuthenticated)`: If the user is not authenticated, it redirects them to the login page (`/login`) using the `Navigate` component, with `replace` set to true to prevent users from returning to the protected route after logging in.
- **Order Status Checks**:

  1. **Pending Orders**:
     - **Condition**: `if (orderPending && !orderCompleted && location.pathname !== "/order-pending")`
     - **Action**: Redirects users with pending orders (and no completed orders) to the `/order-pending` page, ensuring they stay on the pending page until the order is resolved.
  2. **No Completed or Pending Orders**:
     - **Condition**: `if (!orderCompleted && !orderPending && location.pathname !== "/order")`
     - **Action**: Redirects users with no completed or pending orders to the `/order` page. Ensures users without a valid order cannot access protected routes.
  3. **Completed Orders on the Order Page**:
     - **Condition**: `if (orderCompleted && location.pathname === "/order")`
     - **Action**: Redirects users with completed orders away from the `/order` page to the `/calendar` page. This ensures users with valid orders do not remain on the ordering page unnecessarily.

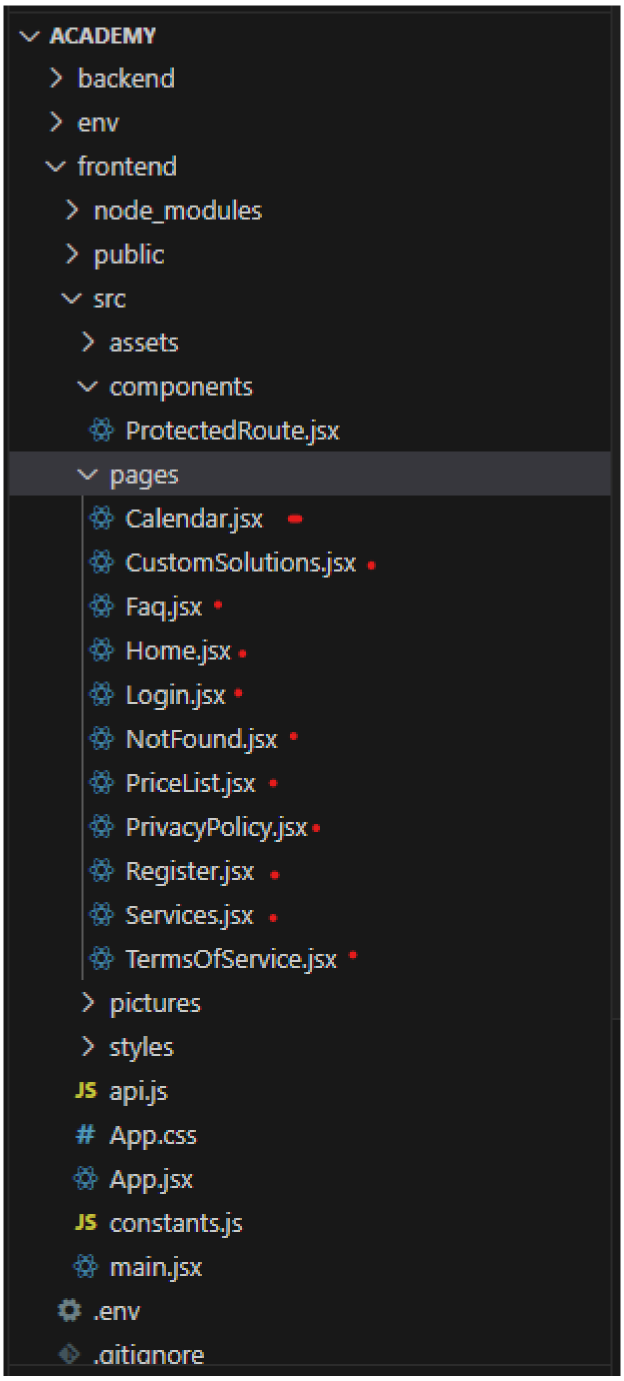
- **Rendering Children**:
  - `return children;`: If none of the above conditions trigger a redirect, the `children` components are rendered, meaning the user is allowed to access the intended protected content.

Export

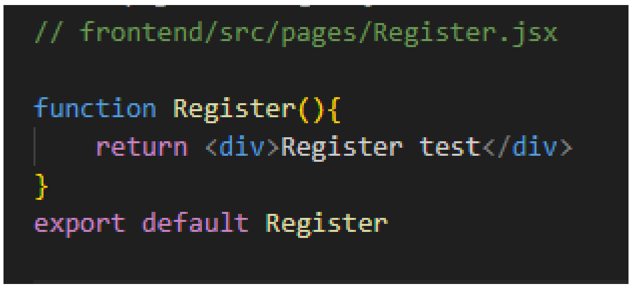
- `export default ProtectedRoute;`: Exports the `ProtectedRoute` component for use in other parts of the application.

3.9 Create page files

Create files for all the pages that we will use in the project. We will start by writing the code on the Register page to test the connection.



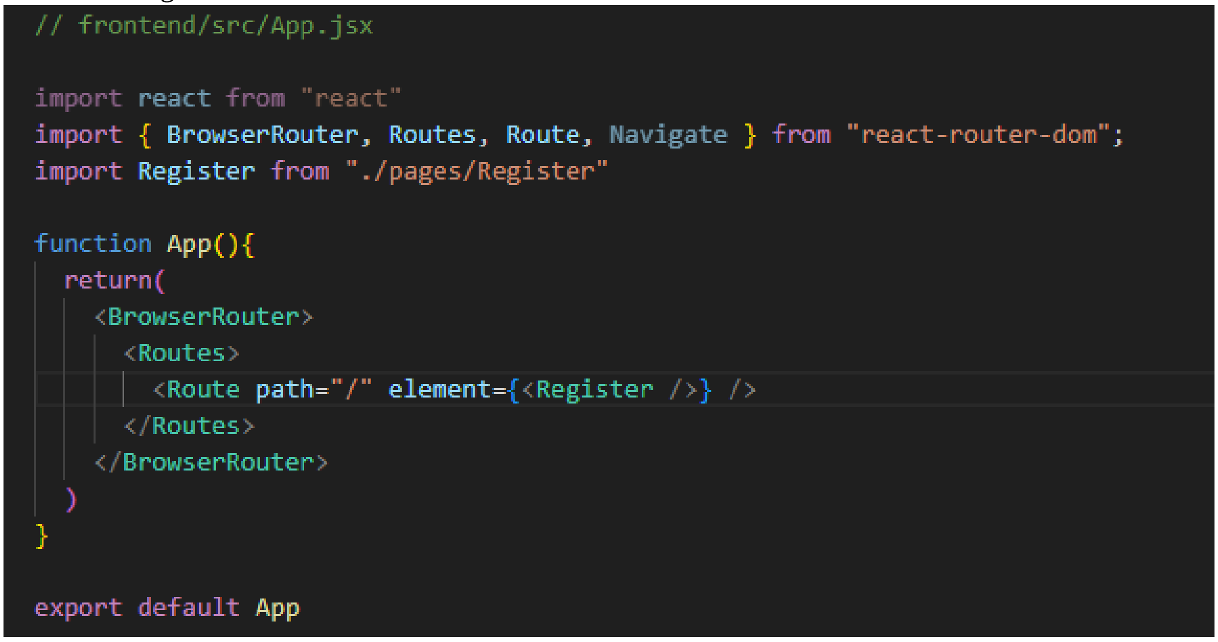
```
ACADEMY
  backend
  env
  frontend
    node_modules
    public
    src
      assets
      components
        ProtectedRoute.jsx
      pages
        Calendar.jsx
        CustomSolutions.jsx
        Faq.jsx
        Home.jsx
        Login.jsx
        NotFound.jsx
        PriceList.jsx
        PrivacyPolicy.jsx
        Register.jsx
        Services.jsx
        TermsOfService.jsx
      pictures
      styles
      api.js
      App.css
      App.jsx
      constants.js
      main.jsx
    .env
    .gitignore
```



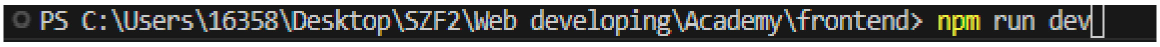
```jsx
// frontend/src/pages/Register.jsx

function Register(){
    return <div>Register test</div>
}
export default Register
```

3.10 Rewrite App

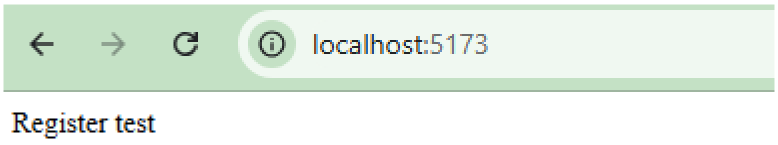
Rewrite the original App.jsx to create the first connection to the Register page. Start the server for the frontend and go to localhost:5173 in the browser.



```
// frontend/src/App.jsx

import react from "react"
import { BrowserRouter, Routes, Route, Navigate } from "react-router-dom";
import Register from "./pages/Register"

function App(){
  return(
    <BrowserRouter>
      <Routes>
        <Route path="/" element={<Register />} />
      </Routes>
    </BrowserRouter>
  )
}

export default App
```



```
PS C:\Users\16358\Desktop\SZF2\Web developing\Academy\frontend> npm run dev
```

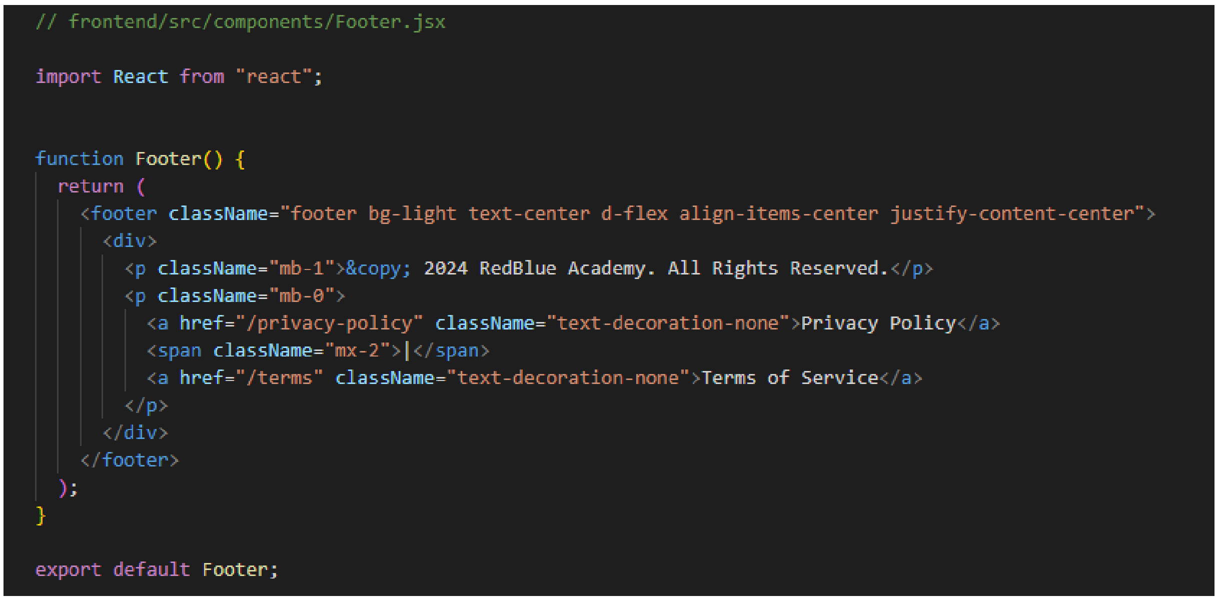


localhost:5173

Register test

3.11 Create Component (Footer.jsx)

The `Footer` component provides a standard footer layout with links to the Privacy Policy and Terms of Service, and displays copyright information.



```jsx
// frontend/src/components/Footer.jsx

import React from "react";


function Footer() {
  return (
    <footer className="footer bg-light text-center d-flex align-items-center justify-content-center">
      <div>
        <p className="mb-1">&copy; 2024 RedBlue Academy. All Rights Reserved.</p>
        <p className="mb-0">
          <a href="/privacy-policy" className="text-decoration-none">Privacy Policy</a>
          <span className="mx-2">|</span>
          <a href="/terms" className="text-decoration-none">Terms of Service</a>
        </p>
      </div>
    </footer>
  );
}

export default Footer;
```

- **Footer Component Definition**:

  - `function Footer() { ... }`: Defines the `Footer` component as a functional component.

- **JSX Structure**:

  - `<footer className="footer bg-light text-center d-flex align-items-center justify-content-center">`: This `footer` tag wraps the entire component. It has the following Bootstrap classes:
    - `bg-light`: Sets a light background color for the footer.
    - `text-center`: Centers the text within the footer.
    - `d-flex align-items-center justify-content-center`: Uses Flexbox to align the items both vertically (`align-items-center`) and horizontally (`justify-content-center`) in the center.
  - `<div>`: A container `div` inside the `footer` that holds two paragraphs of content.
  - `<p className="mb-1">&copy; 2024 RedBlue Academy. All Rights Reserved.</p>`:
    - Displays copyright information.
    - `mb-1`: A Bootstrap class that adds a small bottom margin to the paragraph.
  - `<p className="mb-0"> ... </p>`: A second paragraph containing links to the Privacy Policy and Terms of Service.
    - `mb-0`: Removes any bottom margin for the paragraph.
    - **Links**:
      - `<a href="/privacy-policy" className="text-decoration-none">Privacy Policy</a>`:
        - Links to the Privacy Policy page.
        - `text-decoration-none`: Removes the default underline from the link text.

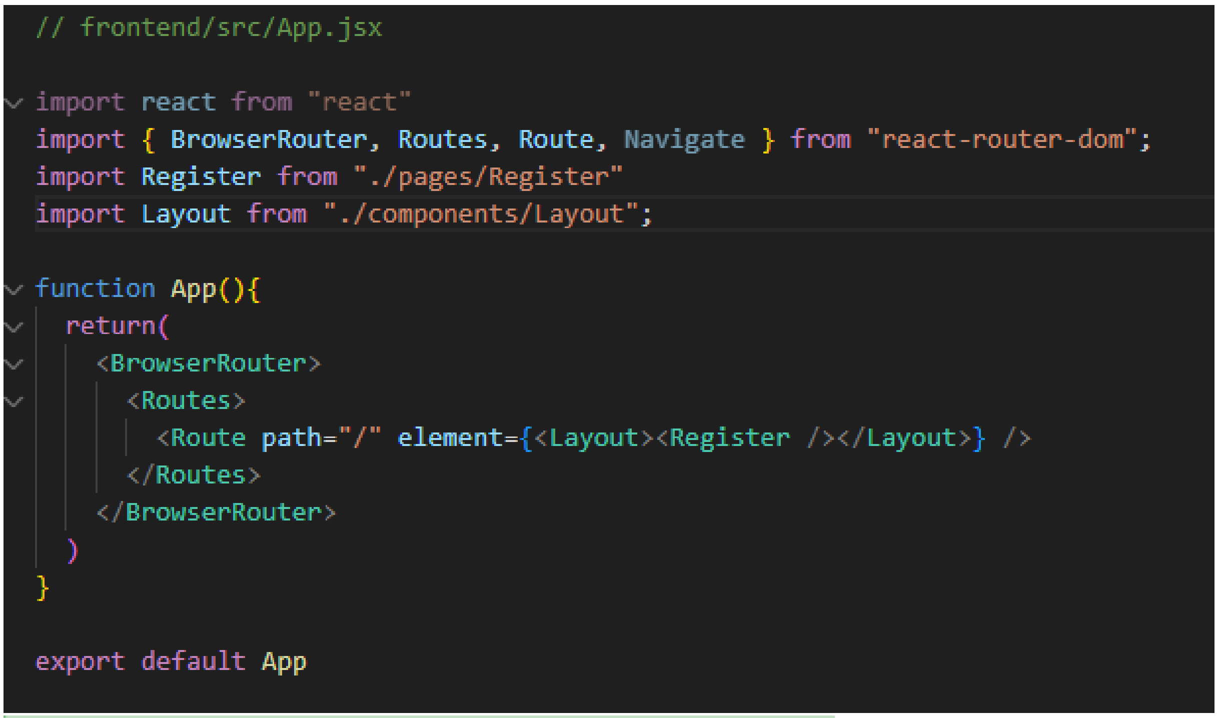
- `<span className="mx-2">|</span>`: A divider (pipe symbol) between the links.
  - `mx-2`: Adds horizontal margin on both sides of the divider.
- `<a href="/terms" className="text-decoration-none">Terms of Service</a>`:
  - Links to the Terms of Service page.
  - `text-decoration-none`: Removes the underline from the link text.

- **Export**:

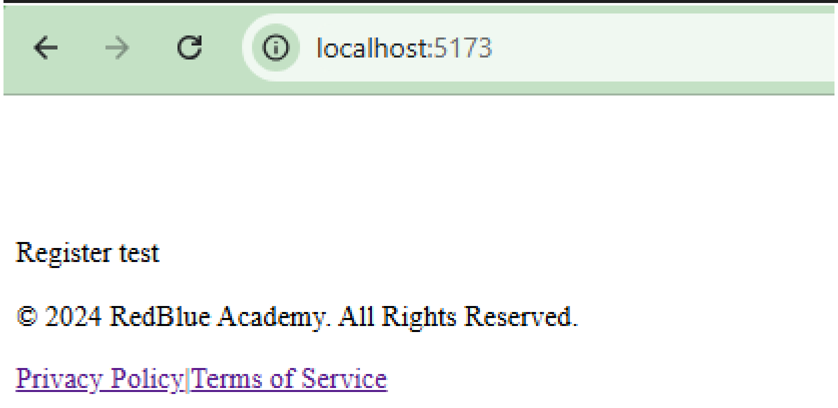
  - `export default Footer;`: Exports the `Footer` component, making it available for import and use in other parts of the application.

3.12 Update App with Layout

Update App to add Layout. Then test and launch the website.



```jsx
// frontend/src/App.jsx

import react from "react"
import { BrowserRouter, Routes, Route, Navigate } from "react-router-dom";
import Register from "./pages/Register"
import Layout from "./components/Layout";

function App(){
  return(
    <BrowserRouter>
      <Routes>
        <Route path="/" element={<Layout><Register /></Layout>} />
      </Routes>
    </BrowserRouter>
  )
}

export default App
```



localhost:5173

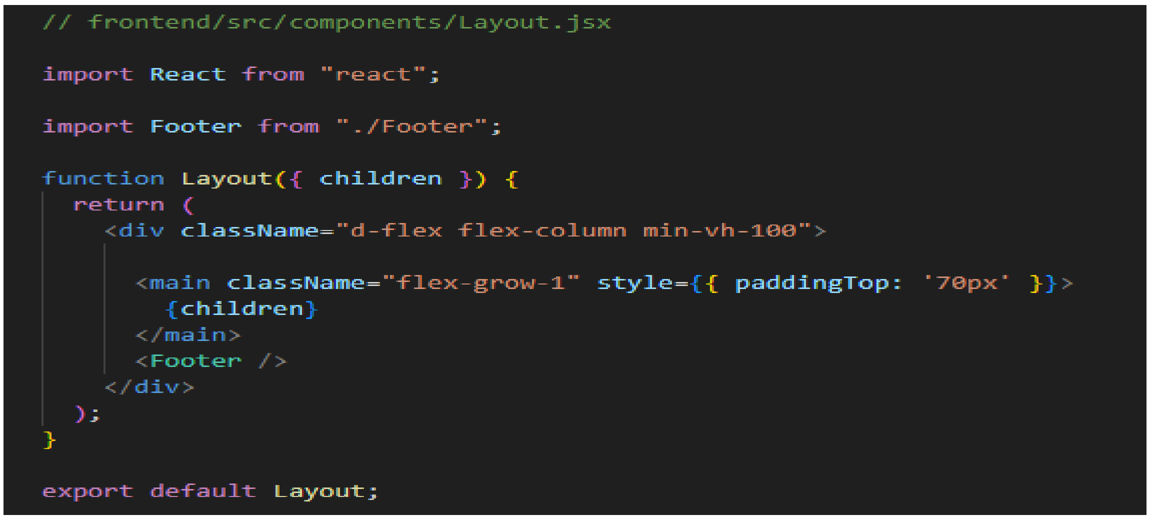
Register test

© 2024 RedBlue Academy. All Rights Reserved.

Privacy Policy|Terms of Service

3.13 Create Component (Layout.jsx)

The `Layout` component provides a consistent page structure. It wraps the main content with a flexible layout that keeps the footer at the bottom of the page. This ensures that the footer is always visible at the bottom, even if the page content is short, creating a cohesive structure for the app.



```jsx
// frontend/src/components/Layout.jsx

import React from "react";

import Footer from "./Footer";

function Layout({ children }) {
  return (
    <div className="d-flex flex-column min-vh-100">

      <main className="flex-grow-1" style={{ paddingTop: '70px' }}>
        {children}
      </main>
      <Footer />
    </div>
  );
}

export default Layout;
```

- **`import Footer from "./Footer";`**: Imports the `Footer` component so it can be used within `Layout`.

- **`function Layout({ children })`**: Defines a functional component named `Layout`, which accepts `children` as a prop. `children` represents the content that will be nested inside this layout component.

- **`<div className="d-flex flex-column min-vh-100">`**: Defines a flexbox container with Bootstrap classes for layout.

  - **`d-flex flex-column`**: Arranges items vertically in a column.
  - **`min-vh-100`**: Ensures the container takes up at least the full viewport height.

- **`<main className="flex-grow-1" style={{ paddingTop: '70px' }}`**: Defines the main content area.

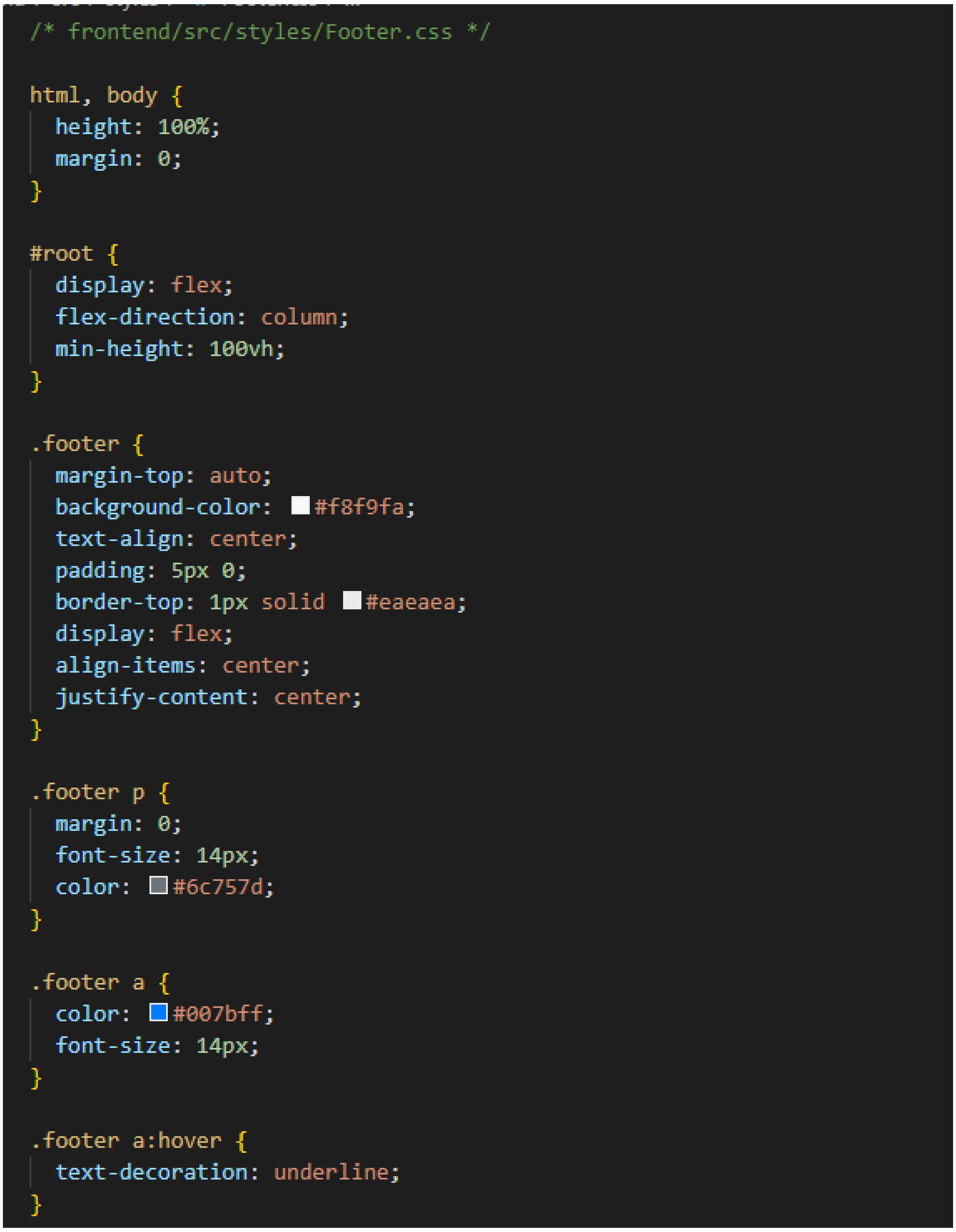
  - **`flex-grow-1`**: Allows the main content to expand and fill available space.
  - **`style={{ paddingTop: '70px' }}`**: Adds padding to the top, likely to create space for a header or navbar.
  - **`{children}`**: Renders any child components passed to the `Layout`.

- **`<Footer />`**: Renders the `Footer` component at the bottom of the layout.

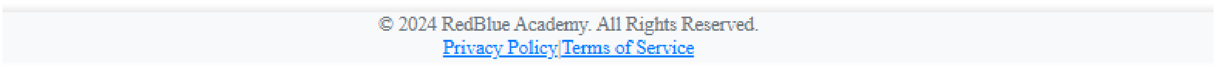
- **`export default Layout;`**: Exports the `Layout` component so it can be used in other parts of the application.

3.14 Create first css

Create first css for Footer and do test



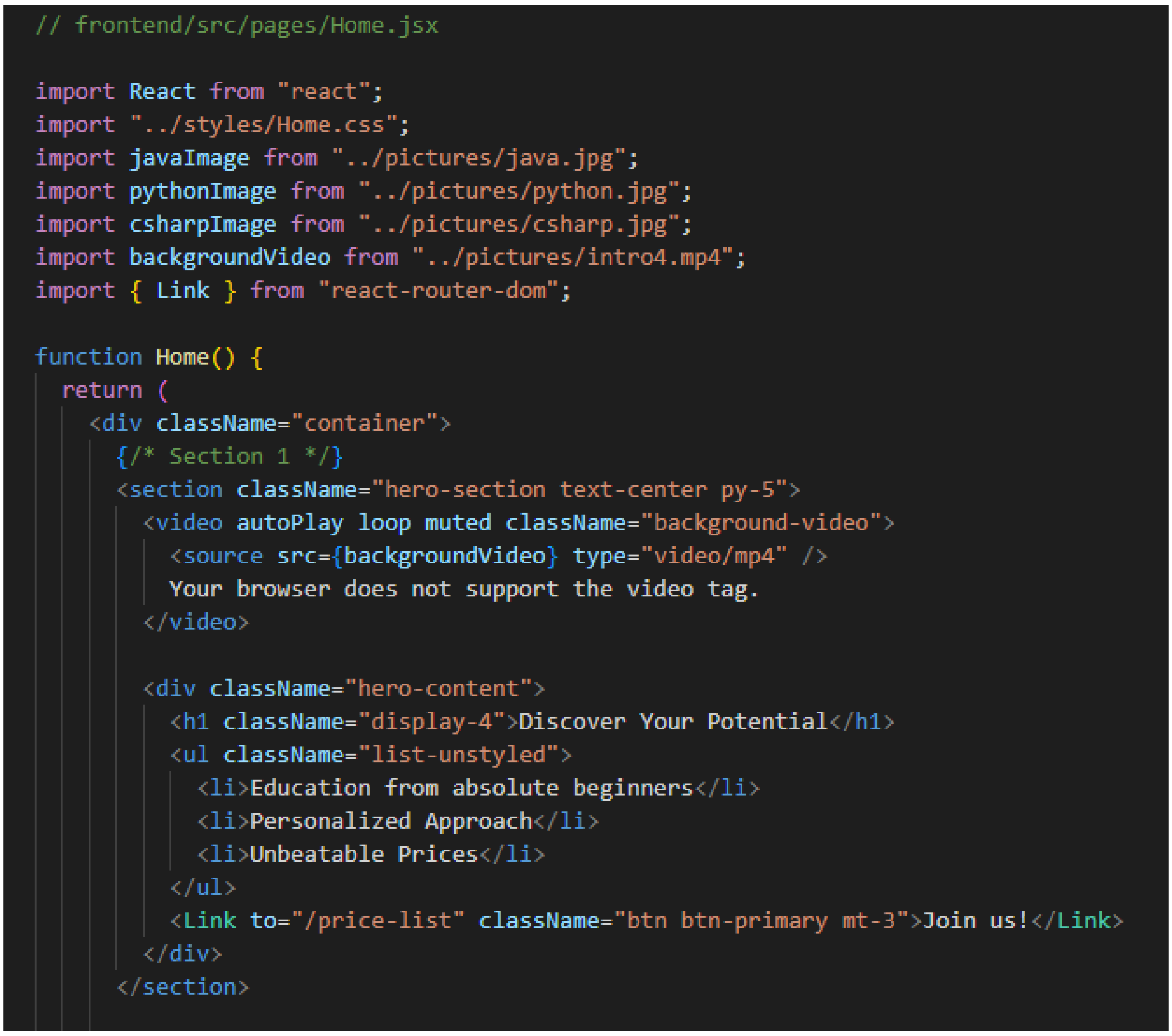
```css
/* frontend/src/styles/Footer.css */

html, body {
  height: 100%;
  margin: 0;
}

#root {
  display: flex;
  flex-direction: column;
  min-height: 100vh;
}

.footer {
  margin-top: auto;
  background-color: #f8f9fa;
  text-align: center;
  padding: 5px 0;
  border-top: 1px solid #eaeaea;
  display: flex;
  align-items: center;
  justify-content: center;
}

.footer p {
  margin: 0;
  font-size: 14px;
  color: #6c757d;
}

.footer a {
  color: #007bff;
  font-size: 14px;
}

.footer a:hover {
  text-decoration: underline;
}
```



© 2024 RedBlue Academy. All Rights Reserved.
Privacy Policy|Terms of Service

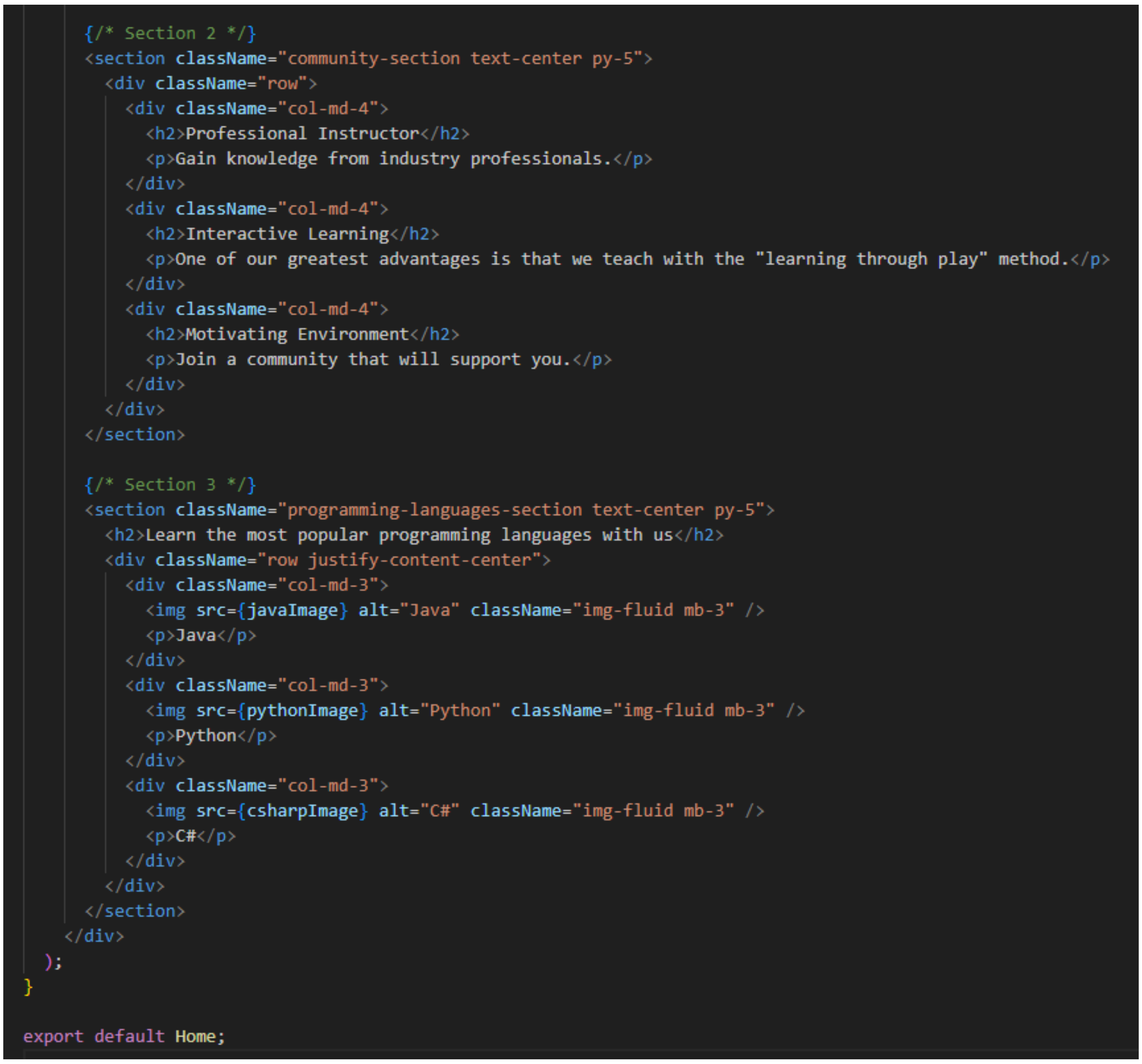
3.15 Create page Home with style

Home component in a React application that represents the homepage. The component is structured into three main sections:



```jsx
// frontend/src/pages/Home.jsx

import React from "react";
import "../styles/Home.css";
import javaImage from "../pictures/java.jpg";
import pythonImage from "../pictures/python.jpg";
import csharpImage from "../pictures/csharp.jpg";
import backgroundVideo from "../pictures/intro4.mp4";
import { Link } from "react-router-dom";

function Home() {
  return (
    <div className="container">
      {/* Section 1 */}
      <section className="hero-section text-center py-5">
        <video autoPlay loop muted className="background-video">
          <source src={backgroundVideo} type="video/mp4" />
          Your browser does not support the video tag.
        </video>

        <div className="hero-content">
          <h1 className="display-4">Discover Your Potential</h1>
          <ul className="list-unstyled">
            <li>Education from absolute beginners</li>
            <li>Personalized Approach</li>
            <li>Unbeatable Prices</li>
          </ul>
          <Link to="/price-list" className="btn btn-primary mt-3">Join us!</Link>
        </div>
      </section>
```

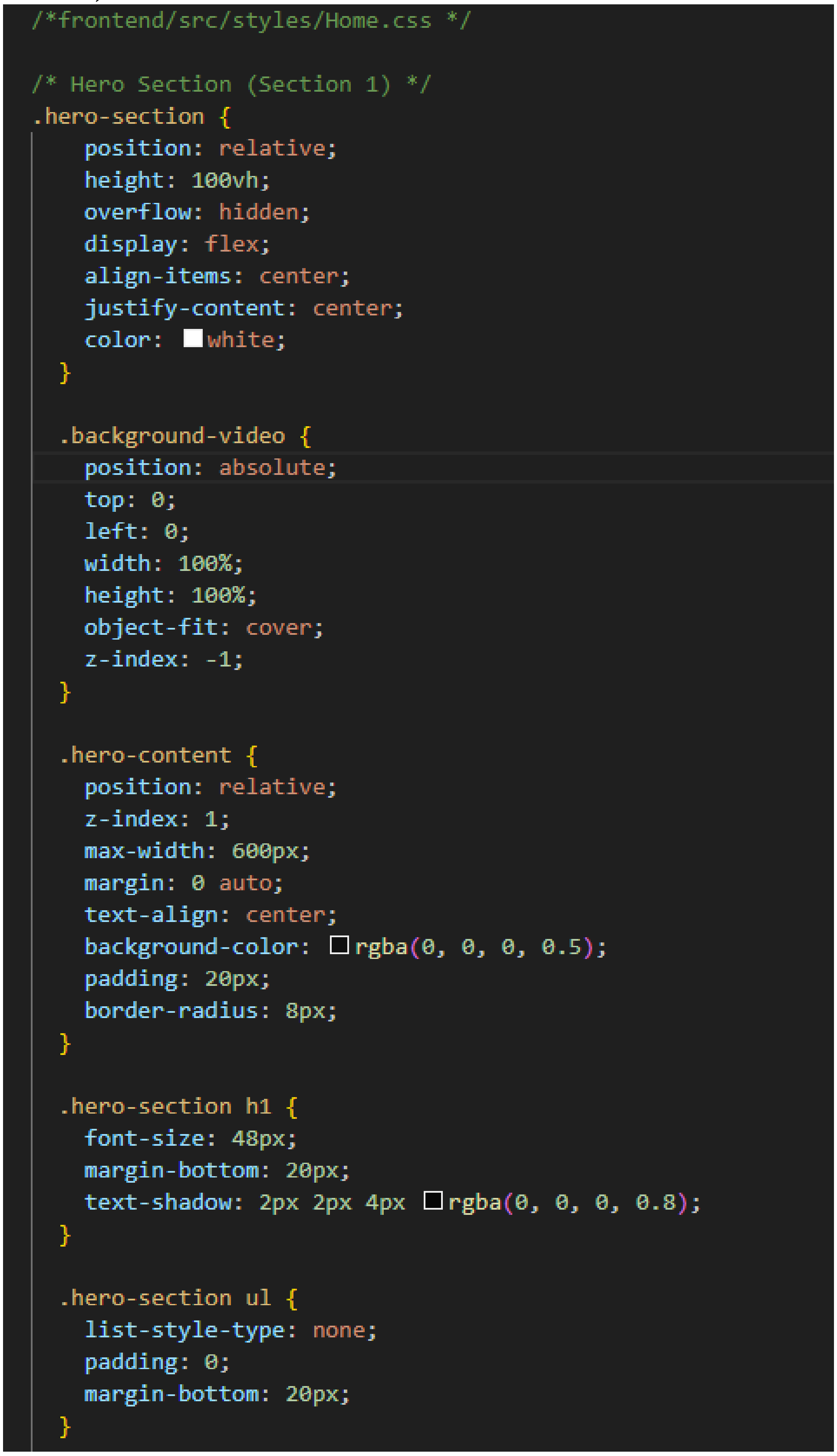
1. **Hero Section**:
   - Displays a background video with overlaid text and a button.
   - Contains motivational text such as "Discover Your Potential" and benefits of the academy like "Education for absolute beginners," "Personalized Approach," and "Unbeatable Prices."
   - Button link to Price page



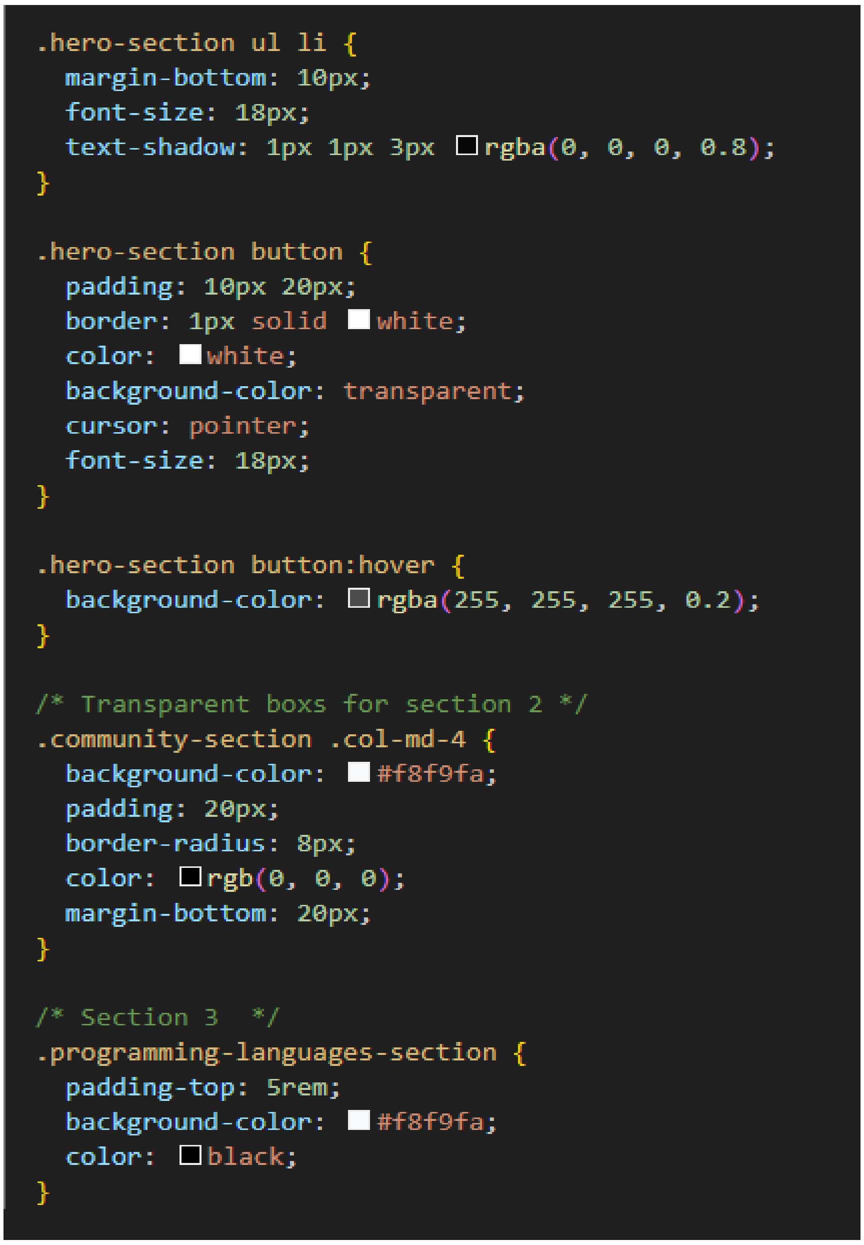
```
      {/* Section 2 */}
      <section className="community-section text-center py-5">
        <div className="row">
          <div className="col-md-4">
            <h2>Professional Instructor</h2>
            <p>Gain knowledge from industry professionals.</p>
          </div>
          <div className="col-md-4">
            <h2>Interactive Learning</h2>
            <p>One of our greatest advantages is that we teach with the "learning through play" method.</p>
          </div>
          <div className="col-md-4">
            <h2>Motivating Environment</h2>
            <p>Join a community that will support you.</p>
          </div>
        </div>
      </section>

      {/* Section 3 */}
      <section className="programming-languages-section text-center py-5">
        <h2>Learn the most popular programming languages with us</h2>
        <div className="row justify-content-center">
          <div className="col-md-3">
            <img src={javaImage} alt="Java" className="img-fluid mb-3" />
            <p>Java</p>
          </div>
          <div className="col-md-3">
            <img src={pythonImage} alt="Python" className="img-fluid mb-3" />
            <p>Python</p>
          </div>
          <div className="col-md-3">
            <img src={csharpImage} alt="C#" className="img-fluid mb-3" />
            <p>C#</p>
          </div>
        </div>
      </section>
    </div>
  );
}

export default Home;
```

2. **Community Section**:
   - Displays three columns with information about the academy's learning environment:
     - **Professional Instructor**
     - **Interactive Learning** (with a "learning through play" approach)
     - **Motivating Environment**
3. **Programming Languages Section**:
   - Highlights popular programming languages (Java, Python, C#) taught in the academy.
   - Each language has an image and a title displayed in a grid layout.

CSS (Home.css)



```css
/*frontend/src/styles/Home.css */

/* Hero Section (Section 1) */
.hero-section {
    position: relative;
    height: 100vh;
    overflow: hidden;
    display: flex;
    align-items: center;
    justify-content: center;
    color: ■white;
  }

  .background-video {
    position: absolute;
    top: 0;
    left: 0;
    width: 100%;
    height: 100%;
    object-fit: cover;
    z-index: -1;
  }

  .hero-content {
    position: relative;
    z-index: 1;
    max-width: 600px;
    margin: 0 auto;
    text-align: center;
    background-color: □rgba(0, 0, 0, 0.5);
    padding: 20px;
    border-radius: 8px;
  }

  .hero-section h1 {
    font-size: 48px;
    margin-bottom: 20px;
    text-shadow: 2px 2px 4px □rgba(0, 0, 0, 0.8);
  }

  .hero-section ul {
    list-style-type: none;
    padding: 0;
    margin-bottom: 20px;
  }
```

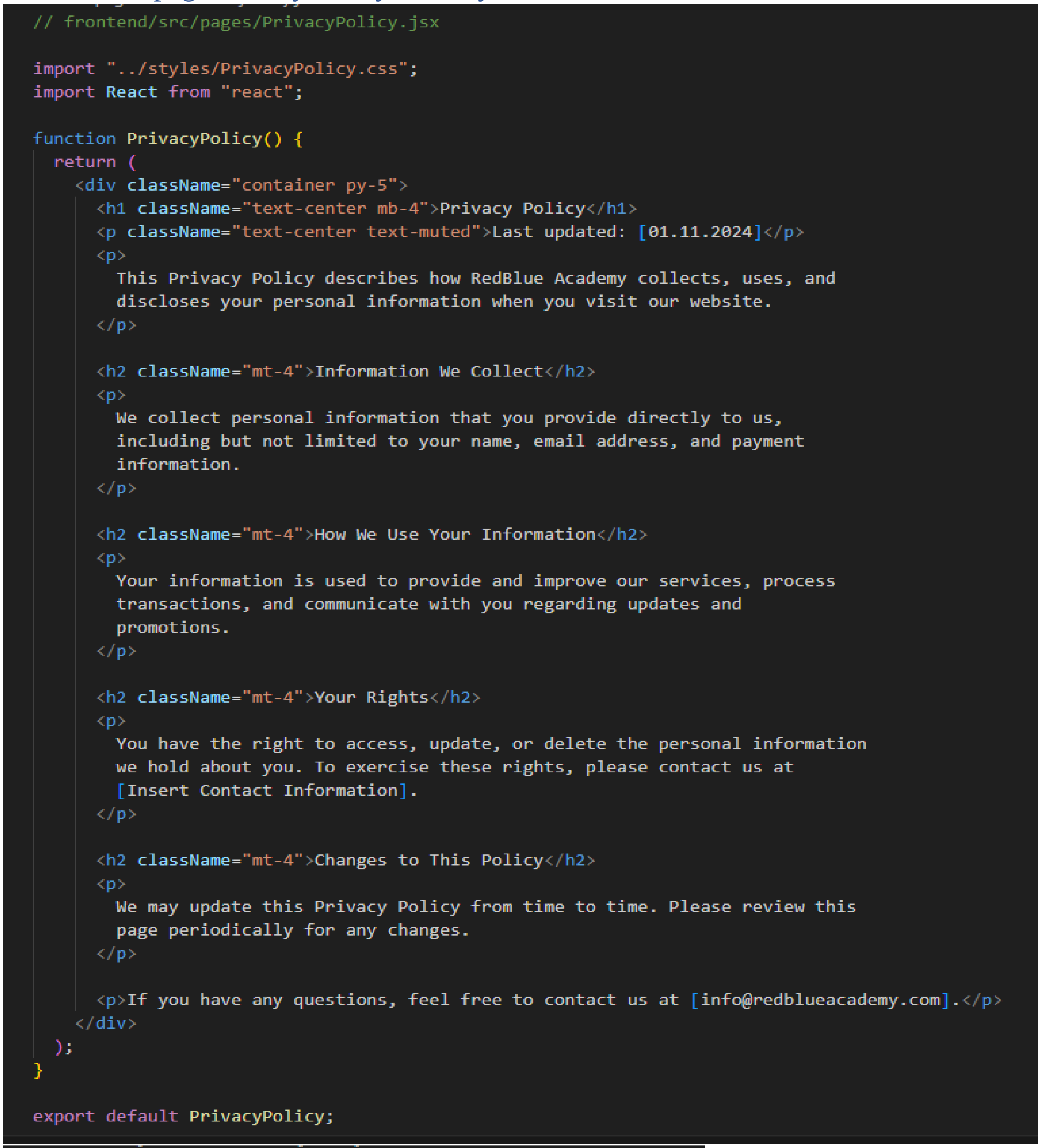


```css
.hero-section ul li {
  margin-bottom: 10px;
  font-size: 18px;
  text-shadow: 1px 1px 3px rgba(0, 0, 0, 0.8);
}

.hero-section button {
  padding: 10px 20px;
  border: 1px solid white;
  color: white;
  background-color: transparent;
  cursor: pointer;
  font-size: 18px;
}

.hero-section button:hover {
  background-color: rgba(255, 255, 255, 0.2);
}

/* Transparent boxs for section 2 */
.community-section .col-md-4 {
  background-color: #f8f9fa;
  padding: 20px;
  border-radius: 8px;
  color: rgb(0, 0, 0);
  margin-bottom: 20px;
}

/* Section 3  */
.programming-languages-section {
  padding-top: 5rem;
  background-color: #f8f9fa;
  color: black;
}
```

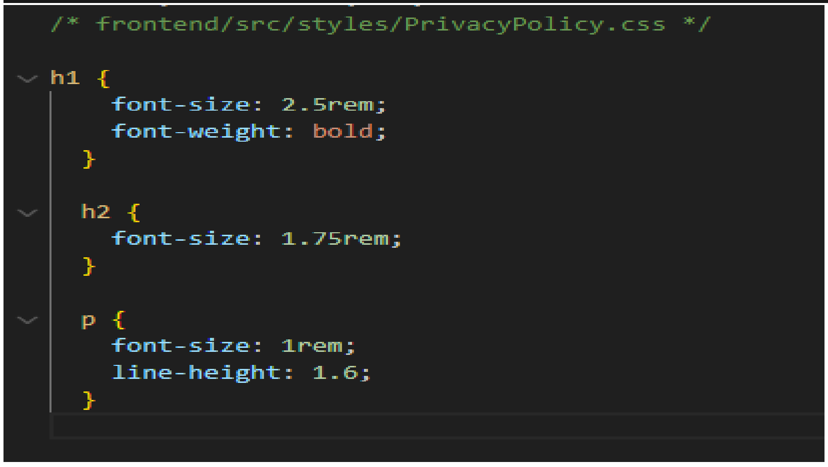
The CSS styles these three sections:

1. **Hero Section Styles**:
   - `hero-section` sets the section's background video to fill the viewport and positions text in the center.
   - `background-video` sets the video to cover the entire background and adjusts its positioning.
   - `hero-content` styles the overlay text container with a semi-transparent background, rounded corners, and centered alignment.
2. **Community Section Styles**:
   - `community-section` styles the text boxes for each item (like "Professional Instructor") with a light background, padding, and rounded corners.
3. **Programming Languages Section Styles**:
   - `programming-languages-section` sets padding and background color for this section, adding visual distinction.

3.16 Create page Privacy Policy with style

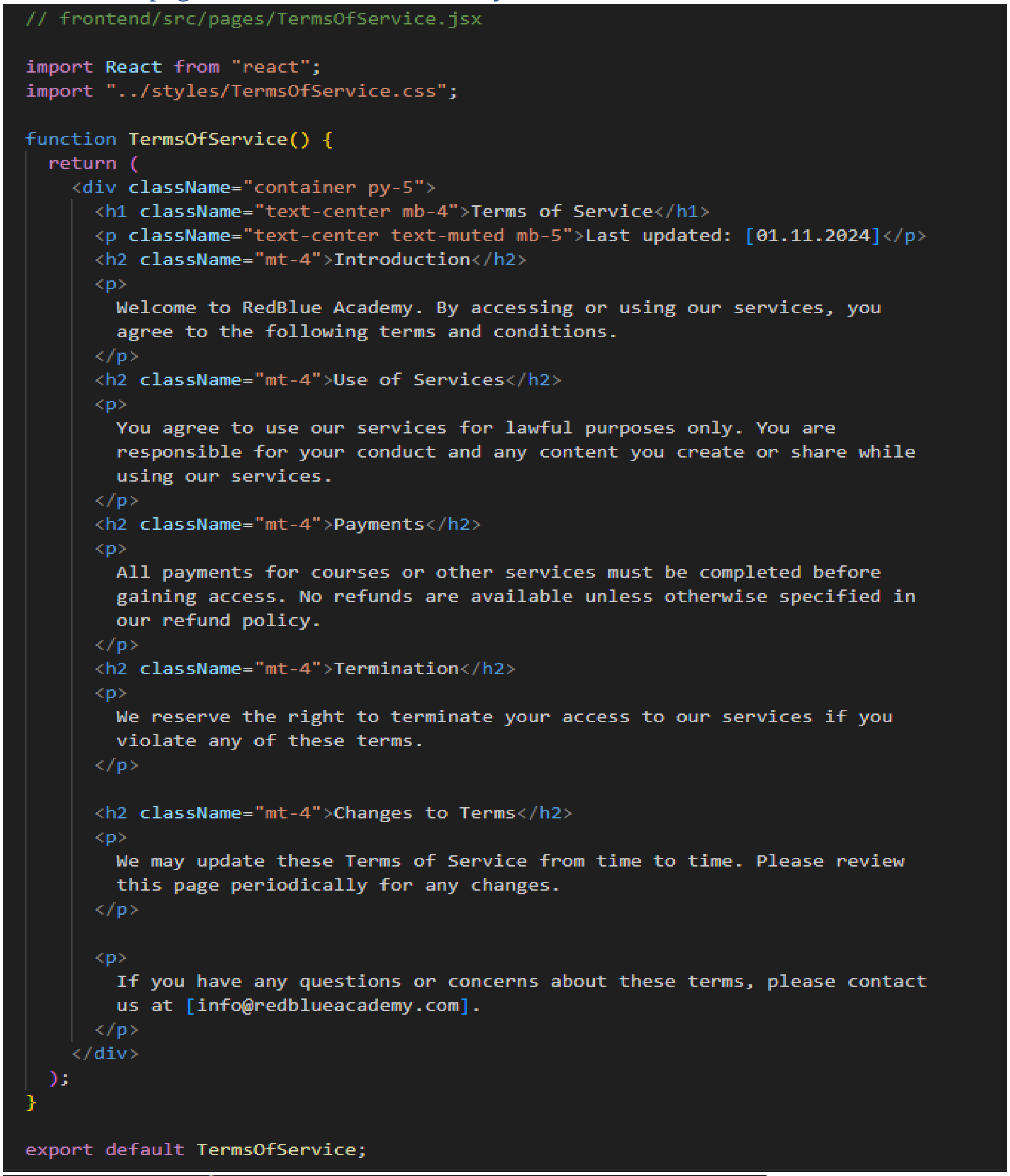


```
// frontend/src/pages/PrivacyPolicy.jsx

import "../styles/PrivacyPolicy.css";
import React from "react";

function PrivacyPolicy() {
  return (
    <div className="container py-5">
      <h1 className="text-center mb-4">Privacy Policy</h1>
      <p className="text-center text-muted">Last updated: [01.11.2024]</p>
      <p>
        This Privacy Policy describes how RedBlue Academy collects, uses, and
        discloses your personal information when you visit our website.
      </p>

      <h2 className="mt-4">Information We Collect</h2>
      <p>
        We collect personal information that you provide directly to us,
        including but not limited to your name, email address, and payment
        information.
      </p>

      <h2 className="mt-4">How We Use Your Information</h2>
      <p>
        Your information is used to provide and improve our services, process
        transactions, and communicate with you regarding updates and
        promotions.
      </p>

      <h2 className="mt-4">Your Rights</h2>
      <p>
        You have the right to access, update, or delete the personal information
        we hold about you. To exercise these rights, please contact us at
        [Insert Contact Information].
      </p>

      <h2 className="mt-4">Changes to This Policy</h2>
      <p>
        We may update this Privacy Policy from time to time. Please review this
        page periodically for any changes.
      </p>

      <p>If you have any questions, feel free to contact us at [info@redblueacademy.com].</p>
    </div>
  );
}

export default PrivacyPolicy;
```

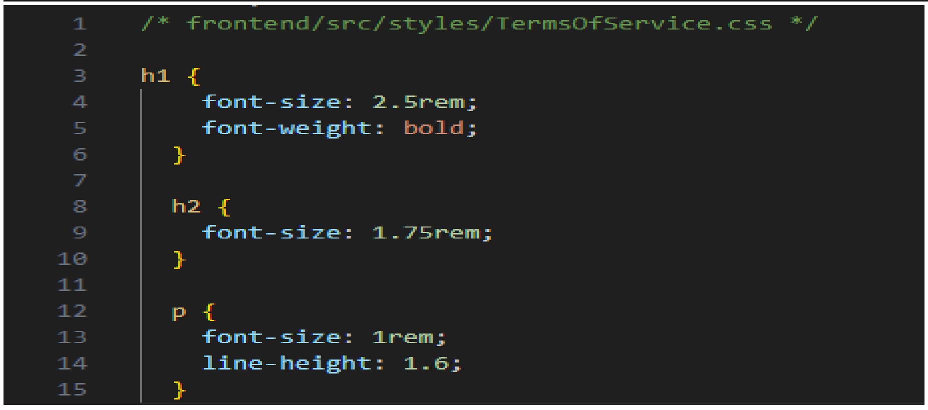


```
/* frontend/src/styles/PrivacyPolicy.css */

h1 {
    font-size: 2.5rem;
    font-weight: bold;
  }

  h2 {
    font-size: 1.75rem;
  }

  p {
    font-size: 1rem;
    line-height: 1.6;
  }
```

3.17 Create page Terms of Service with style



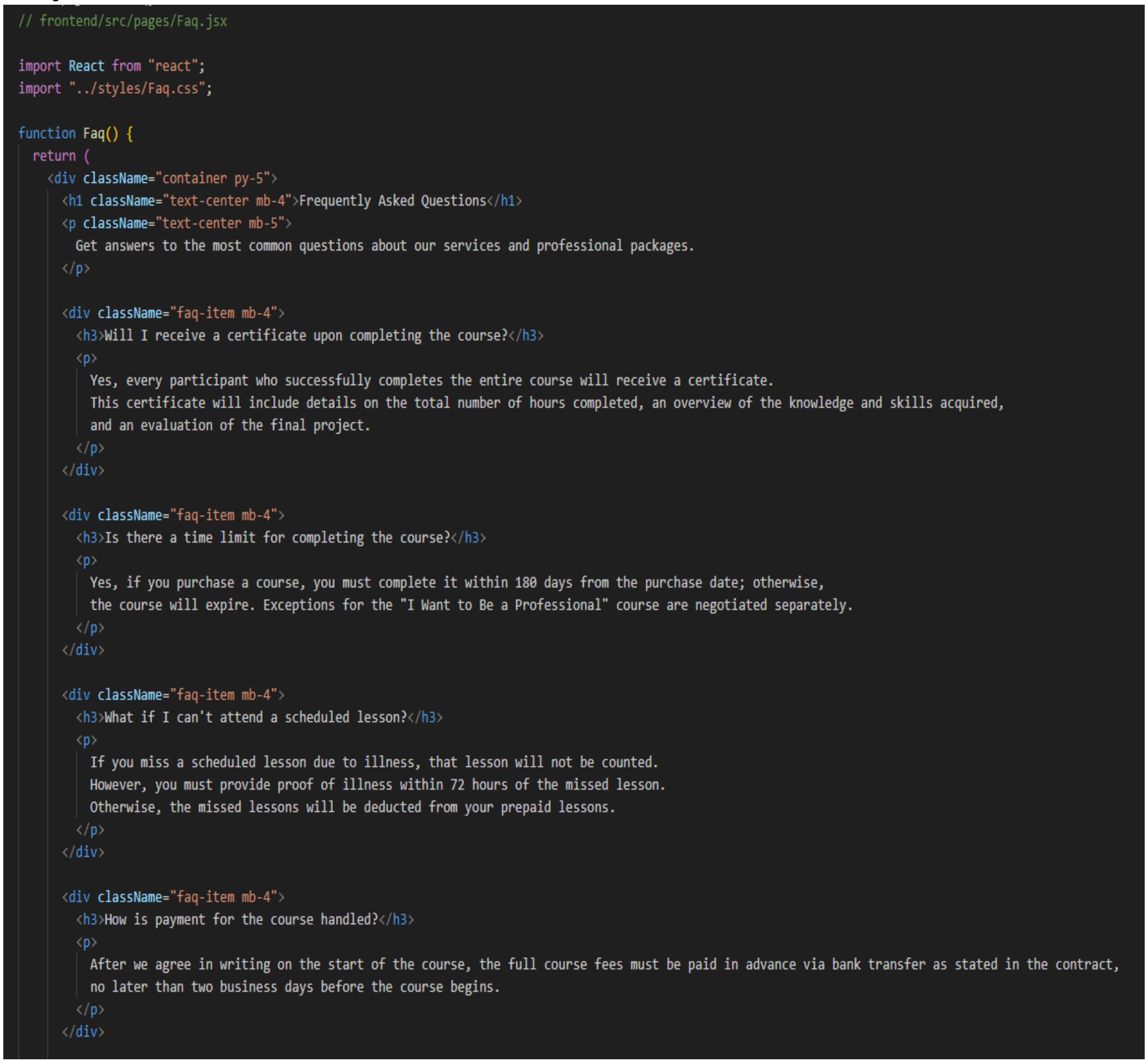
```jsx
// frontend/src/pages/TermsOfService.jsx

import React from "react";
import "../styles/TermsOfService.css";

function TermsOfService() {
  return (
    <div className="container py-5">
      <h1 className="text-center mb-4">Terms of Service</h1>
      <p className="text-center text-muted mb-5">Last updated: [01.11.2024]</p>
      <h2 className="mt-4">Introduction</h2>
      <p>
        Welcome to RedBlue Academy. By accessing or using our services, you
        agree to the following terms and conditions.
      </p>
      <h2 className="mt-4">Use of Services</h2>
      <p>
        You agree to use our services for lawful purposes only. You are
        responsible for your conduct and any content you create or share while
        using our services.
      </p>
      <h2 className="mt-4">Payments</h2>
      <p>
        All payments for courses or other services must be completed before
        gaining access. No refunds are available unless otherwise specified in
        our refund policy.
      </p>
      <h2 className="mt-4">Termination</h2>
      <p>
        We reserve the right to terminate your access to our services if you
        violate any of these terms.
      </p>

      <h2 className="mt-4">Changes to Terms</h2>
      <p>
        We may update these Terms of Service from time to time. Please review
        this page periodically for any changes.
      </p>

      <p>
        If you have any questions or concerns about these terms, please contact
        us at [info@redblueacademy.com].
      </p>
    </div>
  );
}

export default TermsOfService;
```



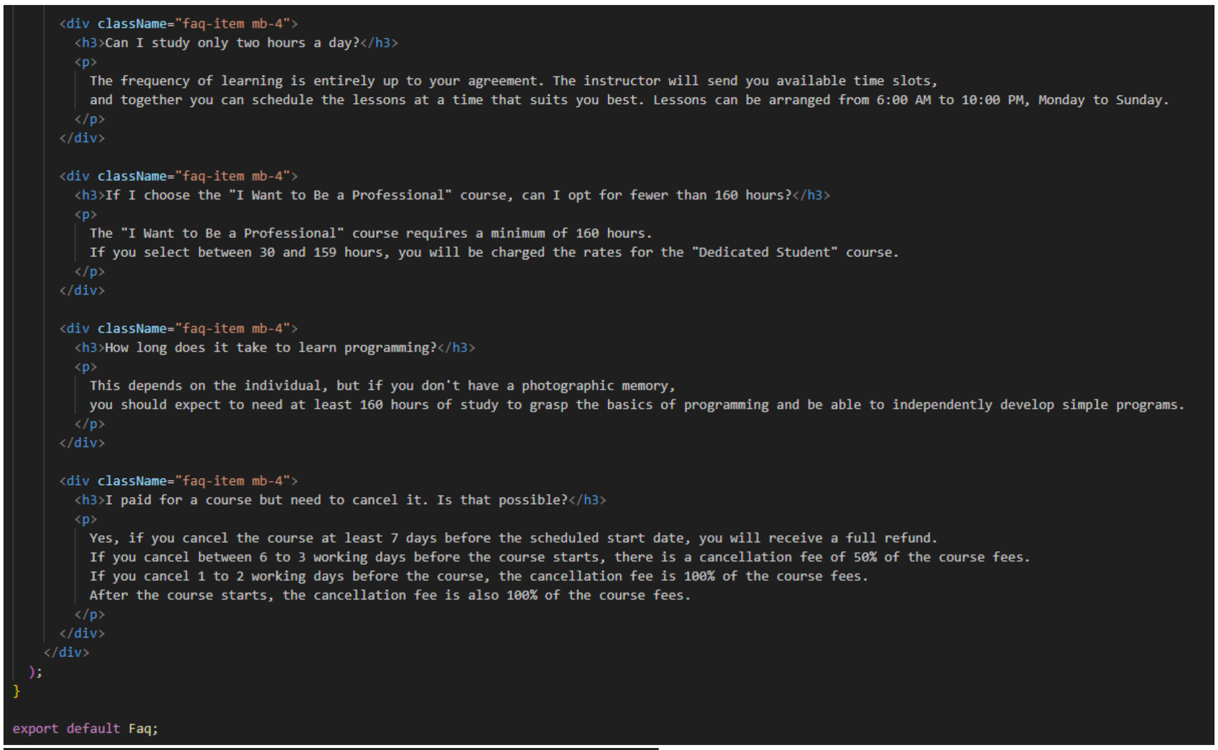
```css
/* frontend/src/styles/TermsOfService.css */

h1 {
    font-size: 2.5rem;
    font-weight: bold;
  }

  h2 {
    font-size: 1.75rem;
  }

  p {
    font-size: 1rem;
    line-height: 1.6;
  }
```

3.18 Create page FAQ with style

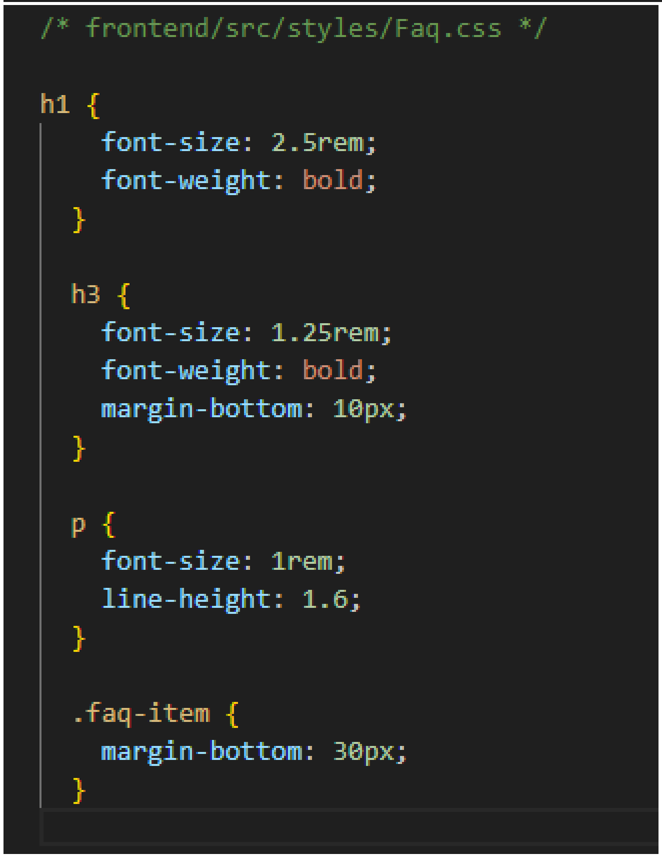
FAQ



```
// frontend/src/pages/Faq.jsx

import React from "react";
import "../styles/Faq.css";

function Faq() {
  return (
    <div className="container py-5">
      <h1 className="text-center mb-4">Frequently Asked Questions</h1>
      <p className="text-center mb-5">
        Get answers to the most common questions about our services and professional packages.
      </p>

      <div className="faq-item mb-4">
        <h3>Will I receive a certificate upon completing the course?</h3>
        <p>
          Yes, every participant who successfully completes the entire course will receive a certificate.
          This certificate will include details on the total number of hours completed, an overview of the knowledge and skills acquired,
          and an evaluation of the final project.
        </p>
      </div>

      <div className="faq-item mb-4">
        <h3>Is there a time limit for completing the course?</h3>
        <p>
          Yes, if you purchase a course, you must complete it within 180 days from the purchase date; otherwise,
          the course will expire. Exceptions for the "I Want to Be a Professional" course are negotiated separately.
        </p>
      </div>

      <div className="faq-item mb-4">
        <h3>What if I can't attend a scheduled lesson?</h3>
        <p>
          If you miss a scheduled lesson due to illness, that lesson will not be counted.
          However, you must provide proof of illness within 72 hours of the missed lesson.
          Otherwise, the missed lessons will be deducted from your prepaid lessons.
        </p>
      </div>

      <div className="faq-item mb-4">
        <h3>How is payment for the course handled?</h3>
        <p>
          After we agree in writing on the start of the course, the full course fees must be paid in advance via bank transfer as stated in the contract,
          no later than two business days before the course begins.
        </p>
      </div>
```

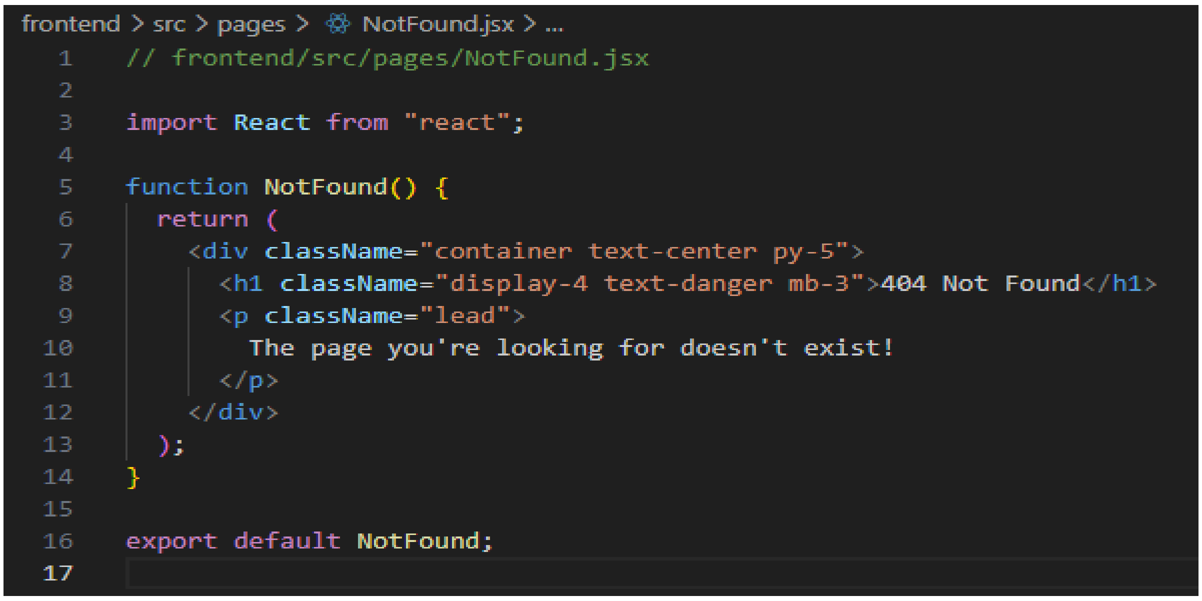


```
      <div className="faq-item mb-4">
        <h3>Can I study only two hours a day?</h3>
        <p>
          The frequency of learning is entirely up to your agreement. The instructor will send you available time slots,
          and together you can schedule the lessons at a time that suits you best. Lessons can be arranged from 6:00 AM to 10:00 PM, Monday to Sunday.
        </p>
      </div>

      <div className="faq-item mb-4">
        <h3>If I choose the "I Want to Be a Professional" course, can I opt for fewer than 160 hours?</h3>
        <p>
          The "I Want to Be a Professional" course requires a minimum of 160 hours.
          If you select between 30 and 159 hours, you will be charged the rates for the "Dedicated Student" course.
        </p>
      </div>

      <div className="faq-item mb-4">
        <h3>How long does it take to learn programming?</h3>
        <p>
          This depends on the individual, but if you don't have a photographic memory,
          you should expect to need at least 160 hours of study to grasp the basics of programming and be able to independently develop simple programs.
        </p>
      </div>

      <div className="faq-item mb-4">
        <h3>I paid for a course but need to cancel it. Is that possible?</h3>
        <p>
          Yes, if you cancel the course at least 7 days before the scheduled start date, you will receive a full refund.
          If you cancel between 6 to 3 working days before the course starts, there is a cancellation fee of 50% of the course fees.
          If you cancel 1 to 2 working days before the course, the cancellation fee is 100% of the course fees.
          After the course starts, the cancellation fee is also 100% of the course fees.
        </p>
      </div>
    </div>
  );
}

export default Faq;
```



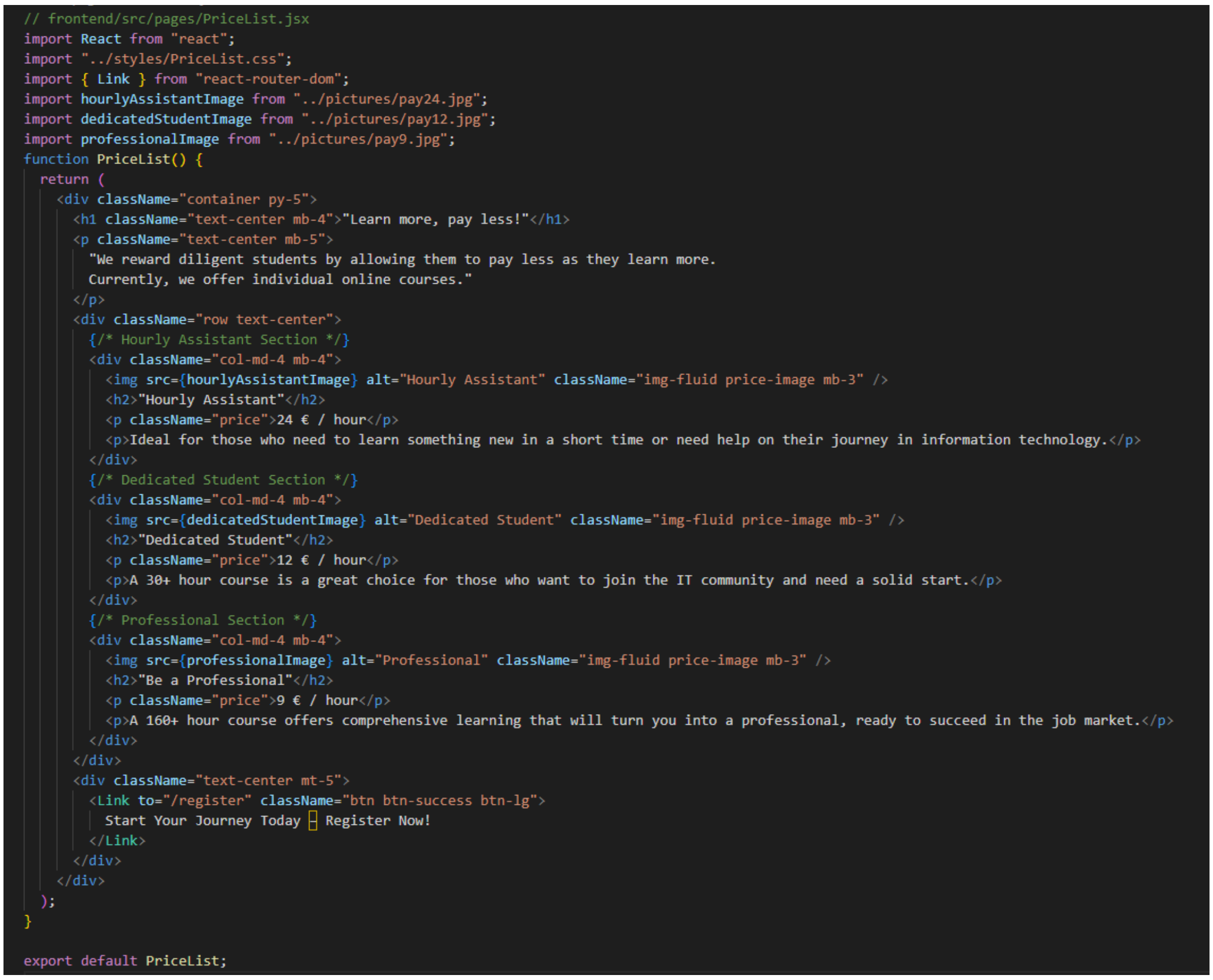
```
/* frontend/src/styles/Faq.css */

h1 {
    font-size: 2.5rem;
    font-weight: bold;
  }

  h3 {
    font-size: 1.25rem;
    font-weight: bold;
    margin-bottom: 10px;
  }

  p {
    font-size: 1rem;
    line-height: 1.6;
  }

  .faq-item {
    margin-bottom: 30px;
  }
```

3.19 Create page Not Found

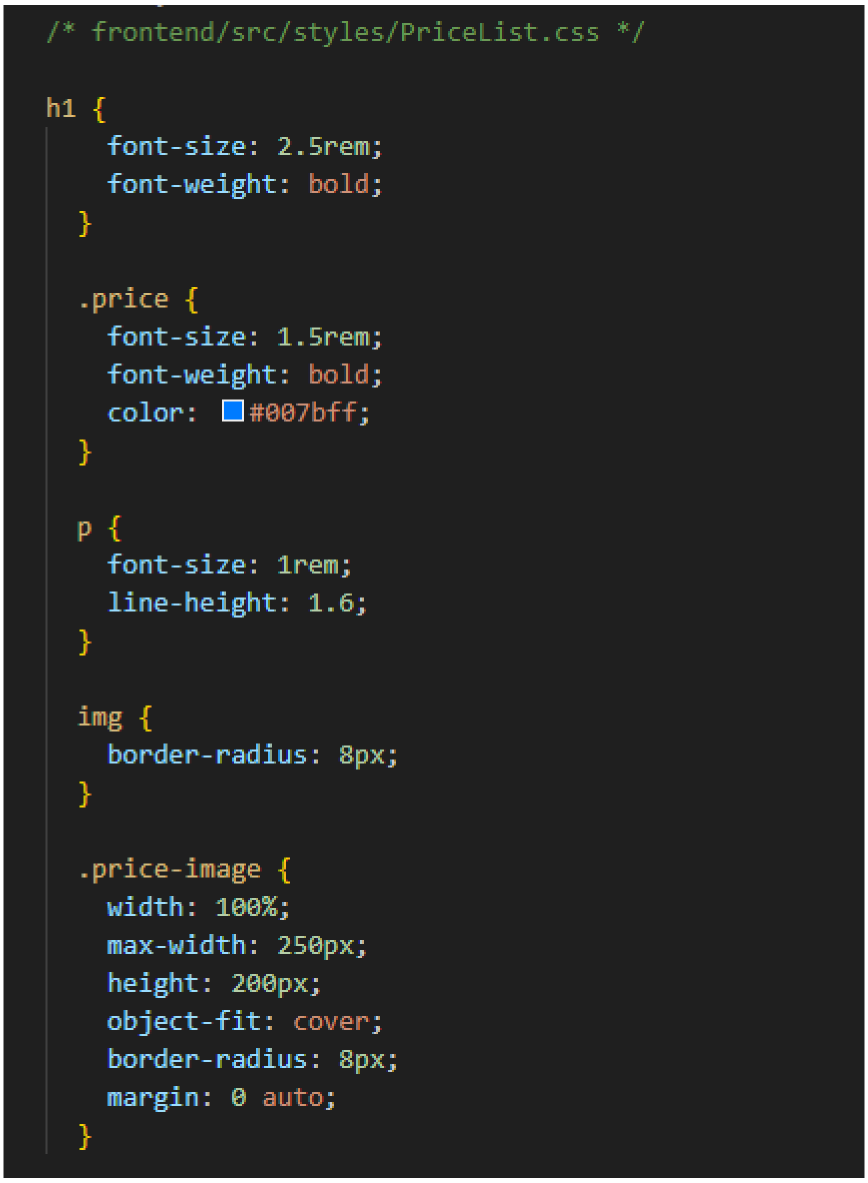

frontend > src > pages > NotFound.jsx > ...

```
// frontend/src/pages/NotFound.jsx

import React from "react";

function NotFound() {
  return (
    <div className="container text-center py-5">
      <h1 className="display-4 text-danger mb-3">404 Not Found</h1>
      <p className="lead">
        The page you're looking for doesn't exist!
      </p>
    </div>
  );
}

export default NotFound;

```

3.20 Create page Price with styled



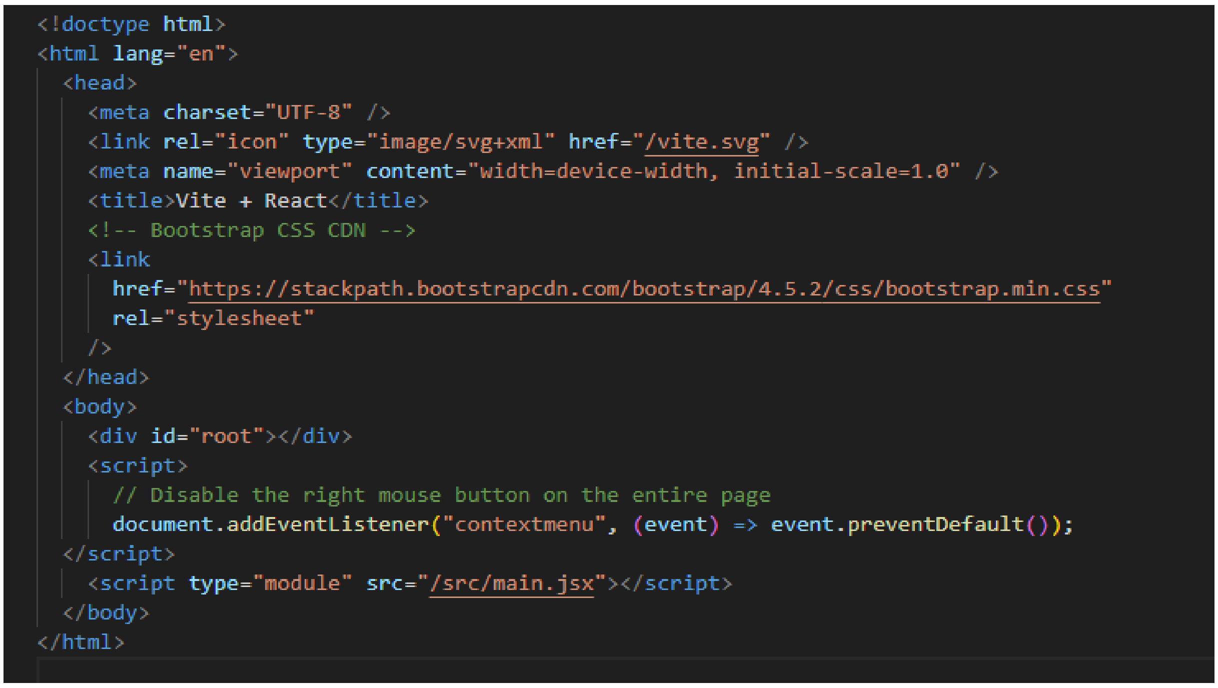
```
// frontend/src/pages/PriceList.jsx
import React from "react";
import "../styles/PriceList.css";
import { Link } from "react-router-dom";
import hourlyAssistantImage from "../pictures/pay24.jpg";
import dedicatedStudentImage from "../pictures/pay12.jpg";
import professionalImage from "../pictures/pay9.jpg";
function PriceList() {
  return (
    <div className="container py-5">
      <h1 className="text-center mb-4">"Learn more, pay less!"</h1>
      <p className="text-center mb-5">
        "We reward diligent students by allowing them to pay less as they learn more.
        Currently, we offer individual online courses."
      </p>
      <div className="row text-center">
        {/* Hourly Assistant Section */}
        <div className="col-md-4 mb-4">
          <img src={hourlyAssistantImage} alt="Hourly Assistant" className="img-fluid price-image mb-3" />
          <h2>"Hourly Assistant"</h2>
          <p className="price">24 € / hour</p>
          <p>Ideal for those who need to learn something new in a short time or need help on their journey in information technology.</p>
        </div>
        {/* Dedicated Student Section */}
        <div className="col-md-4 mb-4">
          <img src={dedicatedStudentImage} alt="Dedicated Student" className="img-fluid price-image mb-3" />
          <h2>"Dedicated Student"</h2>
          <p className="price">12 € / hour</p>
          <p>A 30+ hour course is a great choice for those who want to join the IT community and need a solid start.</p>
        </div>
        {/* Professional Section */}
        <div className="col-md-4 mb-4">
          <img src={professionalImage} alt="Professional" className="img-fluid price-image mb-3" />
          <h2>"Be a Professional"</h2>
          <p className="price">9 € / hour</p>
          <p>A 160+ hour course offers comprehensive learning that will turn you into a professional, ready to succeed in the job market.</p>
        </div>
      </div>
      <div className="text-center mt-5">
        <Link to="/register" className="btn btn-success btn-lg">
          Start Your Journey Today – Register Now!
        </Link>
      </div>
    </div>
  );
}

export default PriceList;
```



```css
/* frontend/src/styles/PriceList.css */

h1 {
    font-size: 2.5rem;
    font-weight: bold;
  }

  .price {
    font-size: 1.5rem;
    font-weight: bold;
    color: 🟦#007bff;
  }

  p {
    font-size: 1rem;
    line-height: 1.6;
  }

  img {
    border-radius: 8px;
  }

  .price-image {
    width: 100%;
    max-width: 250px;
    height: 200px;
    object-fit: cover;
    border-radius: 8px;
    margin: 0 auto;
  }
```

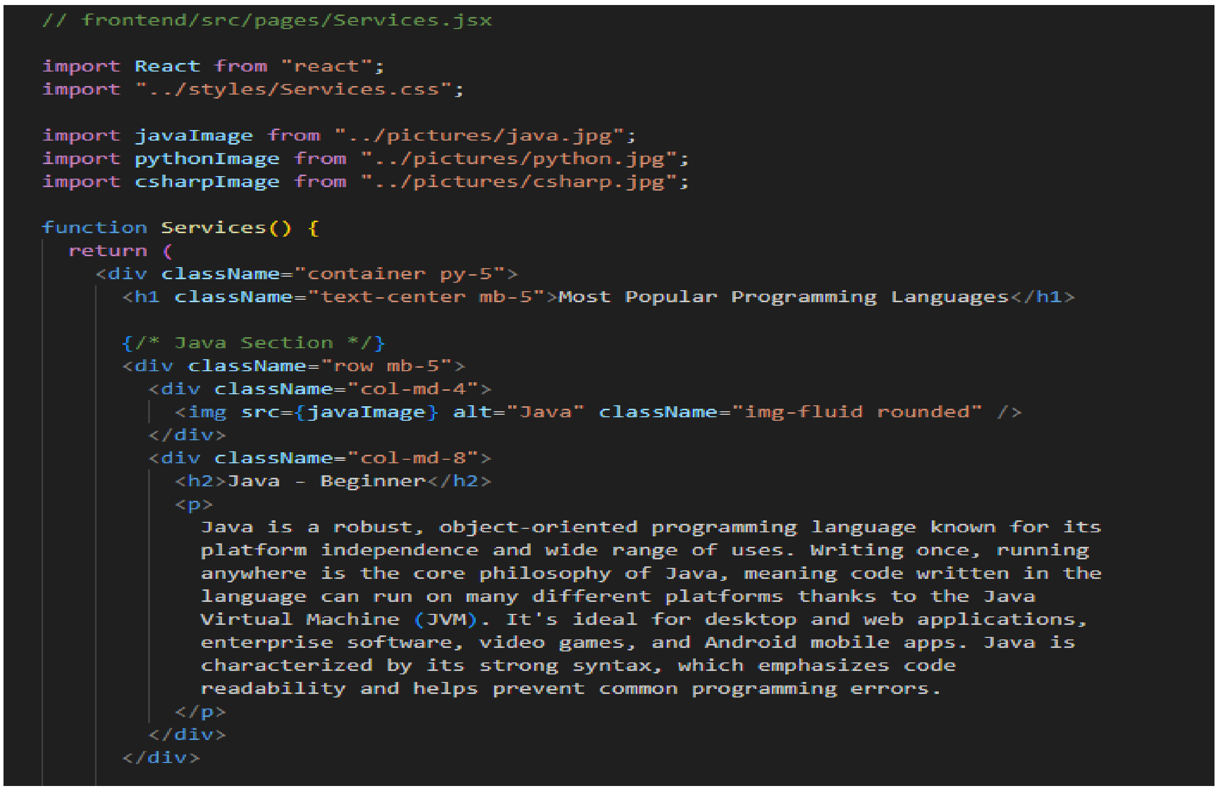
3.21 Update index.html

Update index.html to add the Bootstrap link and a script to disable the right-click on the page, preventing people from copying text.

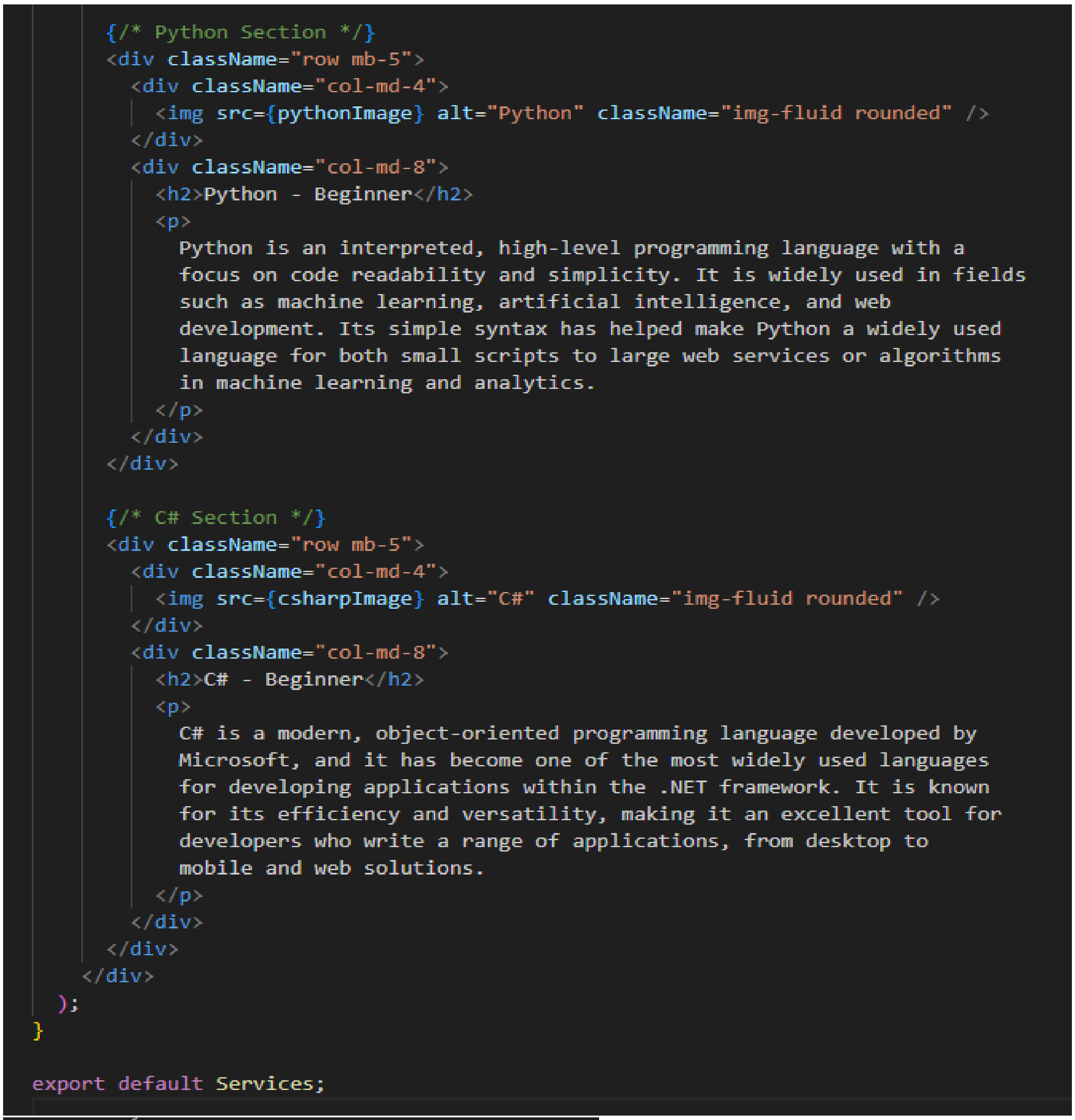


```html
<!doctype html>
<html lang="en">
  <head>
    <meta charset="UTF-8" />
    <link rel="icon" type="image/svg+xml" href="/vite.svg" />
    <meta name="viewport" content="width=device-width, initial-scale=1.0" />
    <title>Vite + React</title>
    <!-- Bootstrap CSS CDN -->
    <link
      href="https://stackpath.bootstrapcdn.com/bootstrap/4.5.2/css/bootstrap.min.css"
      rel="stylesheet"
    />
  </head>
  <body>
    <div id="root"></div>
    <script>
      // Disable the right mouse button on the entire page
      document.addEventListener("contextmenu", (event) => event.preventDefault());
  </script>
    <script type="module" src="/src/main.jsx"></script>
  </body>
</html>
```

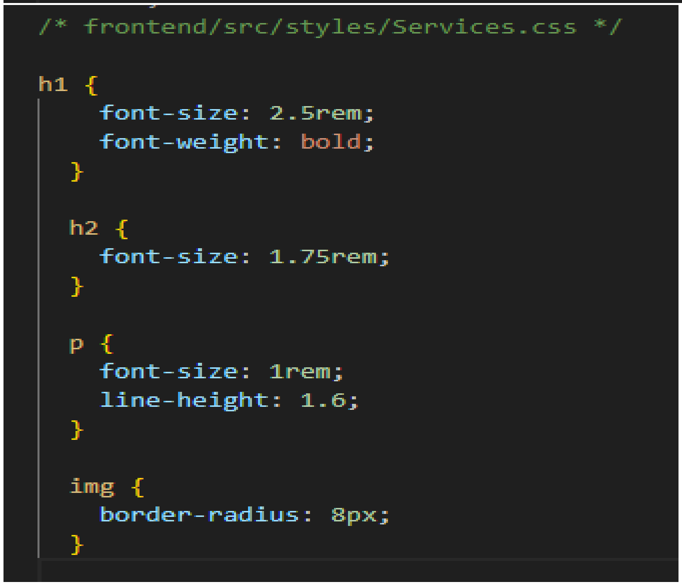
3.22 Create page Services with style



```jsx
// frontend/src/pages/Services.jsx

import React from "react";
import "../styles/Services.css";

import javaImage from "../pictures/java.jpg";
import pythonImage from "../pictures/python.jpg";
import csharpImage from "../pictures/csharp.jpg";

function Services() {
  return (
    <div className="container py-5">
      <h1 className="text-center mb-5">Most Popular Programming Languages</h1>

      {/* Java Section */}
      <div className="row mb-5">
        <div className="col-md-4">
          <img src={javaImage} alt="Java" className="img-fluid rounded" />
        </div>
        <div className="col-md-8">
          <h2>Java - Beginner</h2>
          <p>
            Java is a robust, object-oriented programming language known for its
            platform independence and wide range of uses. Writing once, running
            anywhere is the core philosophy of Java, meaning code written in the
            language can run on many different platforms thanks to the Java
            Virtual Machine (JVM). It's ideal for desktop and web applications,
            enterprise software, video games, and Android mobile apps. Java is
            characterized by its strong syntax, which emphasizes code
            readability and helps prevent common programming errors.
          </p>
        </div>
      </div>
```



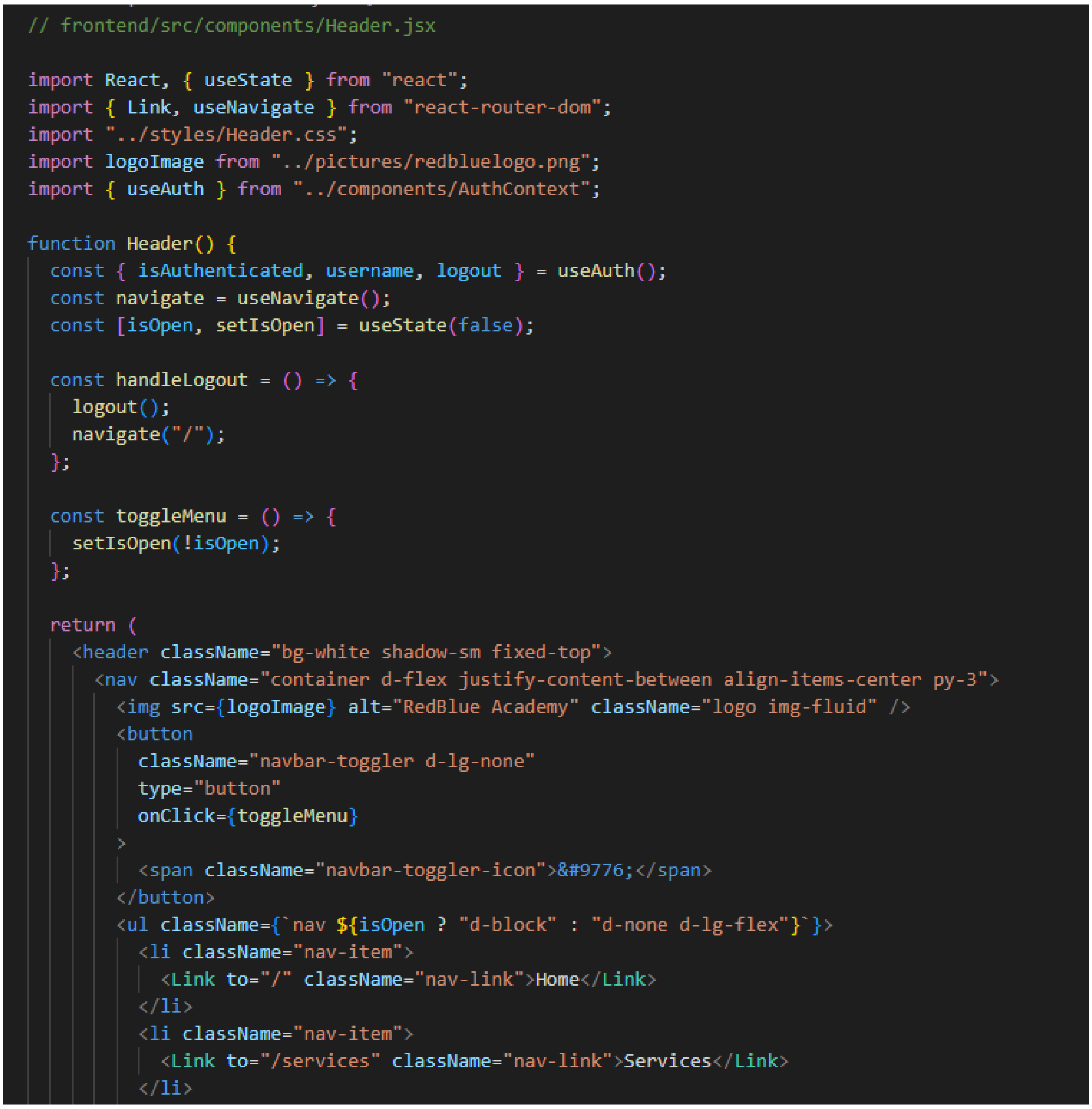
```
        {/* Python Section */}
        <div className="row mb-5">
          <div className="col-md-4">
            <img src={pythonImage} alt="Python" className="img-fluid rounded" />
          </div>
          <div className="col-md-8">
            <h2>Python - Beginner</h2>
            <p>
              Python is an interpreted, high-level programming language with a
              focus on code readability and simplicity. It is widely used in fields
              such as machine learning, artificial intelligence, and web
              development. Its simple syntax has helped make Python a widely used
              language for both small scripts to large web services or algorithms
              in machine learning and analytics.
            </p>
          </div>
        </div>

        {/* C# Section */}
        <div className="row mb-5">
          <div className="col-md-4">
            <img src={csharpImage} alt="C#" className="img-fluid rounded" />
          </div>
          <div className="col-md-8">
            <h2>C# - Beginner</h2>
            <p>
              C# is a modern, object-oriented programming language developed by
              Microsoft, and it has become one of the most widely used languages
              for developing applications within the .NET framework. It is known
              for its efficiency and versatility, making it an excellent tool for
              developers who write a range of applications, from desktop to
              mobile and web solutions.
            </p>
          </div>
        </div>
      </div>
    );
}

export default Services;
```



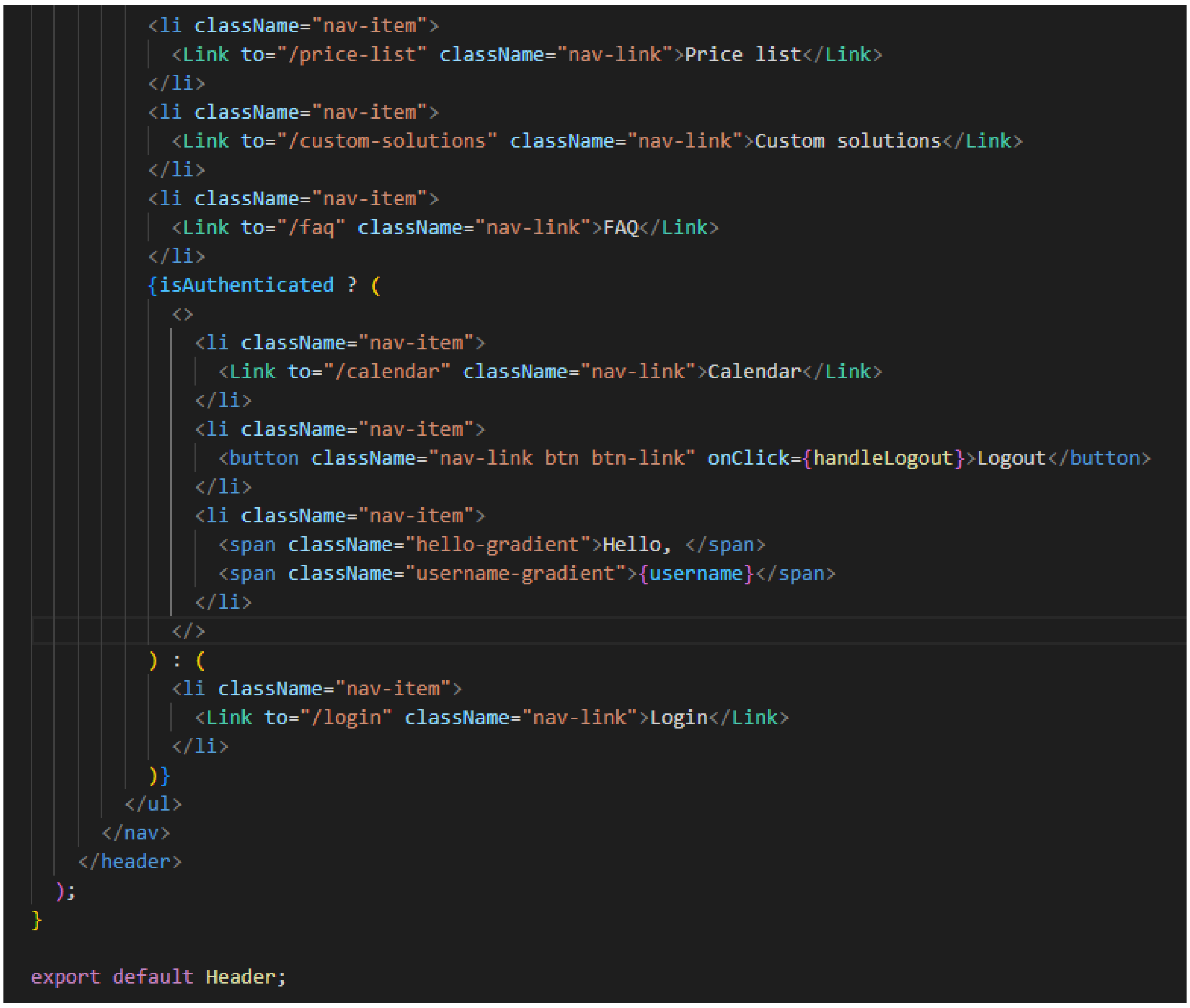
```
/* frontend/src/styles/Services.css */

h1 {
    font-size: 2.5rem;
    font-weight: bold;
  }

  h2 {
    font-size: 1.75rem;
  }

  p {
    font-size: 1rem;
    line-height: 1.6;
  }

  img {
    border-radius: 8px;
  }
```

3.23 Create component Header with style

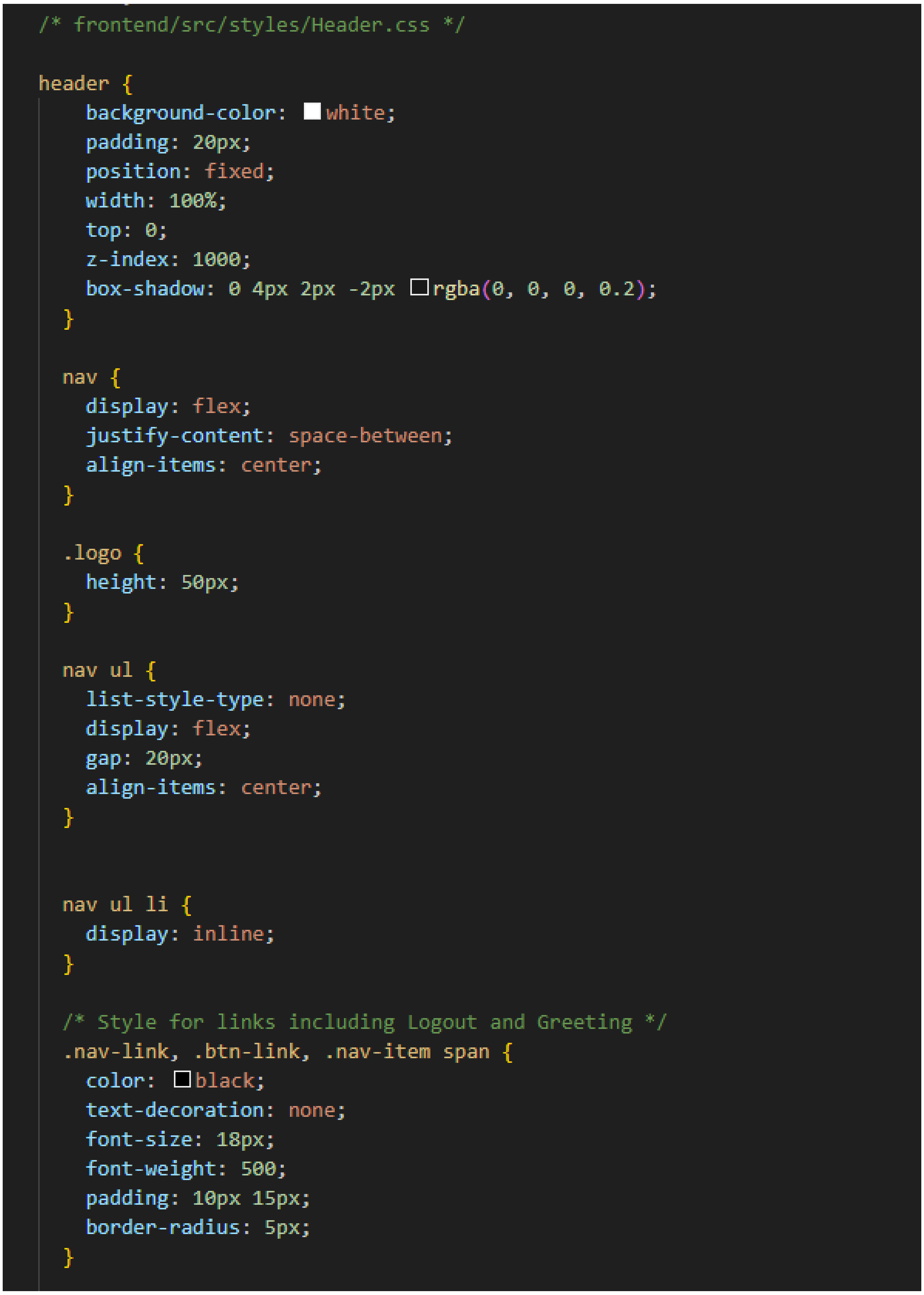
This component enables a dynamic and responsive header that displays different options based on the user's authentication status. The `toggleMenu` function makes it adaptive to different screen sizes by showing or hiding the navigation items accordingly.



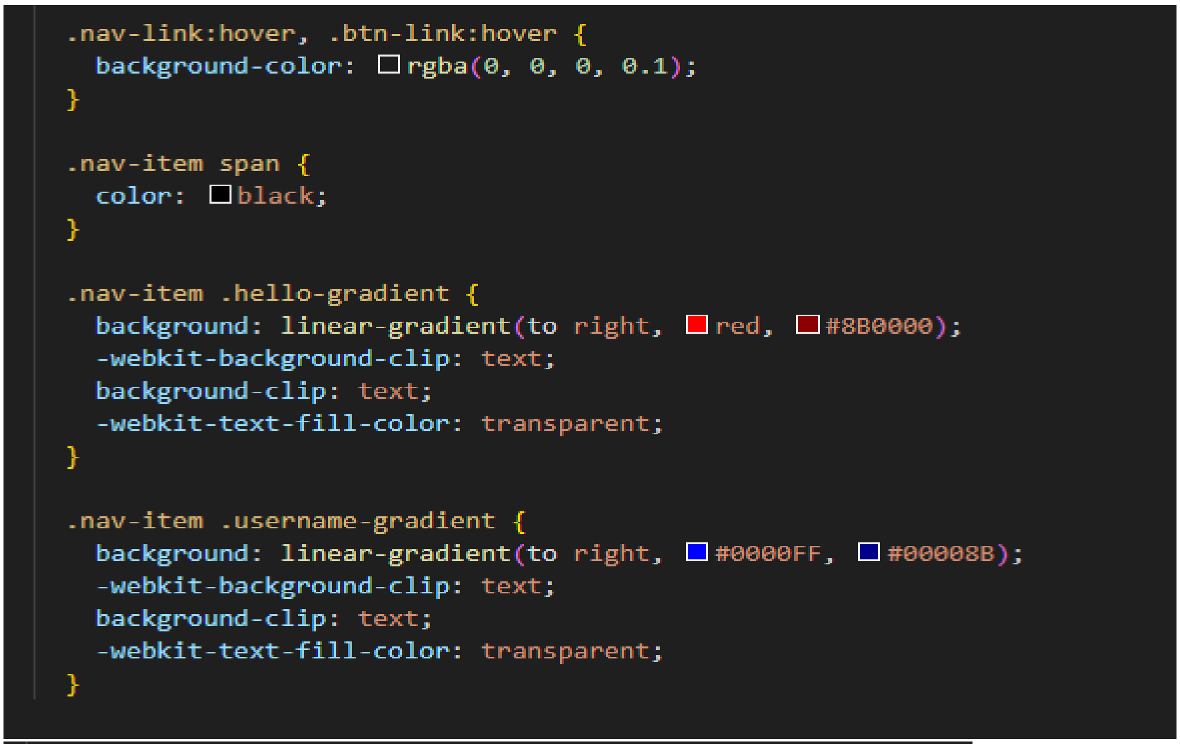
```jsx
// frontend/src/components/Header.jsx

import React, { useState } from "react";
import { Link, useNavigate } from "react-router-dom";
import "../styles/Header.css";
import logoImage from "../pictures/redbluelogo.png";
import { useAuth } from "../components/AuthContext";

function Header() {
  const { isAuthenticated, username, logout } = useAuth();
  const navigate = useNavigate();
  const [isOpen, setIsOpen] = useState(false);

  const handleLogout = () => {
    logout();
    navigate("/");
  };

  const toggleMenu = () => {
    setIsOpen(!isOpen);
  };

  return (
    <header className="bg-white shadow-sm fixed-top">
      <nav className="container d-flex justify-content-between align-items-center py-3">
        <img src={logoImage} alt="RedBlue Academy" className="logo img-fluid" />
        <button
          className="navbar-toggler d-lg-none"
          type="button"
          onClick={toggleMenu}
        >
          <span className="navbar-toggler-icon">&#9776;</span>
        </button>
        <ul className={`nav ${isOpen ? "d-block" : "d-none d-lg-flex"}`}>
          <li className="nav-item">
            <Link to="/" className="nav-link">Home</Link>
          </li>
          <li className="nav-item">
            <Link to="/services" className="nav-link">Services</Link>
          </li>
```



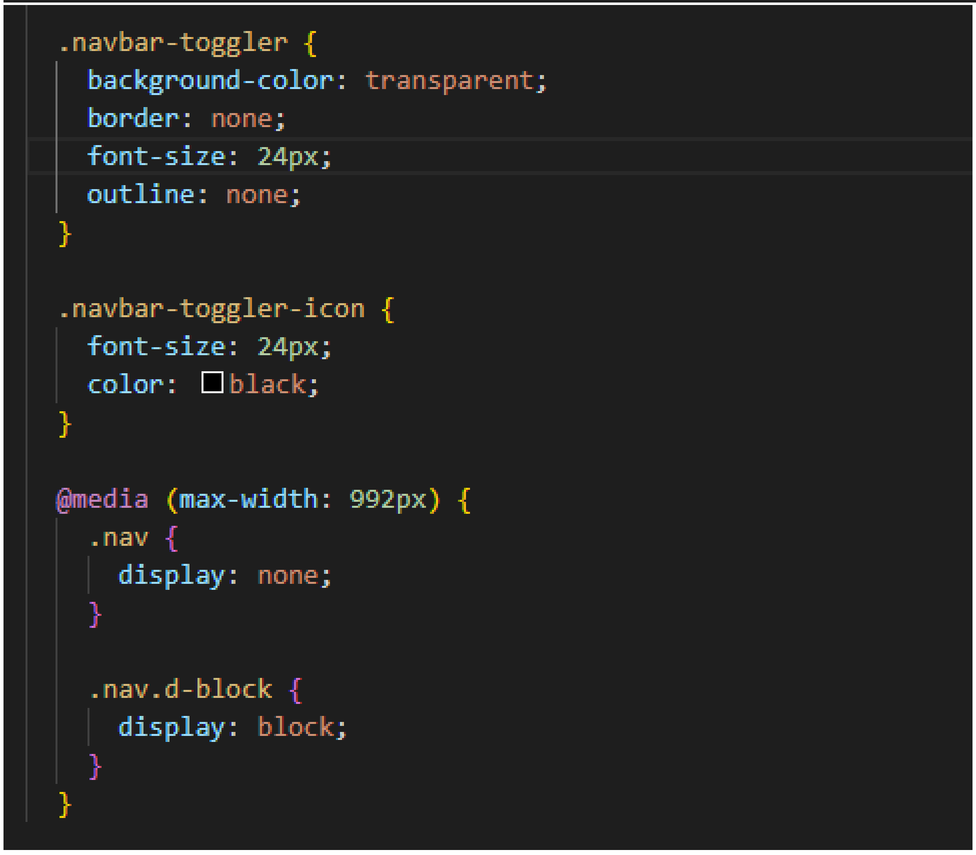
```
          <li className="nav-item">
            <Link to="/price-list" className="nav-link">Price list</Link>
          </li>
          <li className="nav-item">
            <Link to="/custom-solutions" className="nav-link">Custom solutions</Link>
          </li>
          <li className="nav-item">
            <Link to="/faq" className="nav-link">FAQ</Link>
          </li>
          {isAuthenticated ? (
            <>
              <li className="nav-item">
                <Link to="/calendar" className="nav-link">Calendar</Link>
              </li>
              <li className="nav-item">
                <button className="nav-link btn btn-link" onClick={handleLogout}>Logout</button>
              </li>
              <li className="nav-item">
                <span className="hello-gradient">Hello, </span>
                <span className="username-gradient">{username}</span>
              </li>
            </>
          ) : (
            <li className="nav-item">
              <Link to="/login" className="nav-link">Login</Link>
            </li>
          )}
        </ul>
      </nav>
    </header>
  );
}

export default Header;
```



```css
/* frontend/src/styles/Header.css */

header {
    background-color: white;
    padding: 20px;
    position: fixed;
    width: 100%;
    top: 0;
    z-index: 1000;
    box-shadow: 0 4px 2px -2px rgba(0, 0, 0, 0.2);
  }

  nav {
    display: flex;
    justify-content: space-between;
    align-items: center;
  }

  .logo {
    height: 50px;
  }

  nav ul {
    list-style-type: none;
    display: flex;
    gap: 20px;
    align-items: center;
  }


  nav ul li {
    display: inline;
  }

  /* Style for links including Logout and Greeting */
  .nav-link, .btn-link, .nav-item span {
    color: black;
    text-decoration: none;
    font-size: 18px;
    font-weight: 500;
    padding: 10px 15px;
    border-radius: 5px;
  }
```

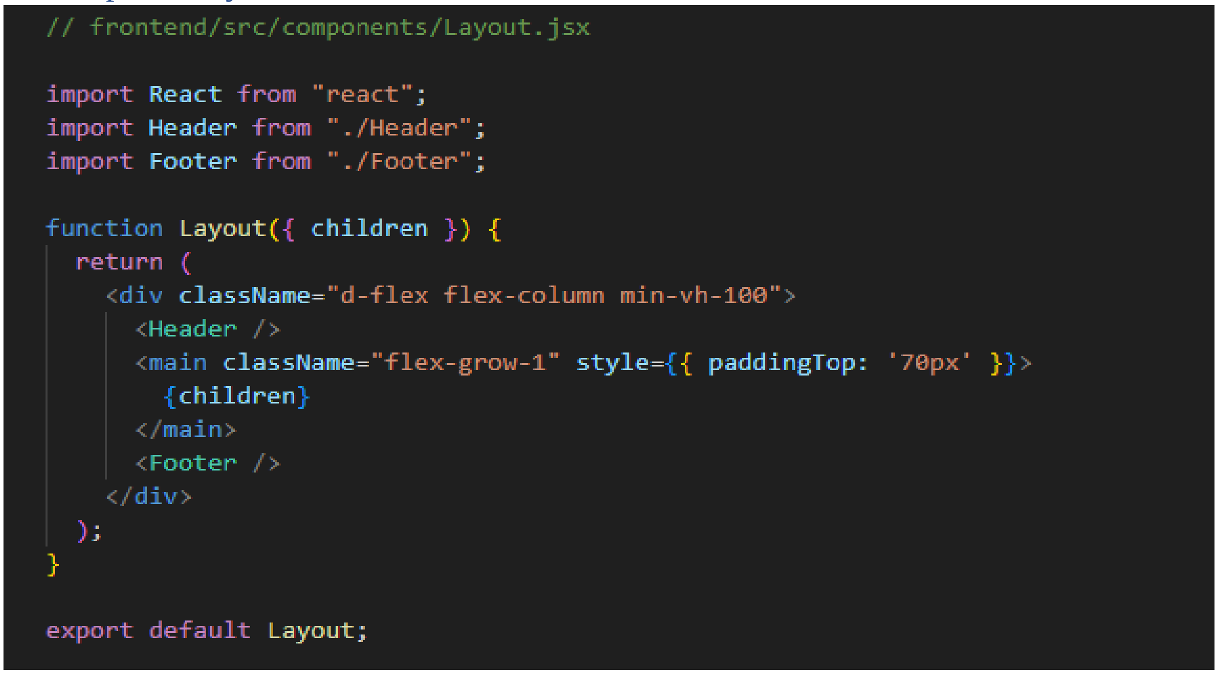


```css
.nav-link:hover, .btn-link:hover {
  background-color: rgba(0, 0, 0, 0.1);
}

.nav-item span {
  color: black;
}

.nav-item .hello-gradient {
  background: linear-gradient(to right, red, #8B0000);
  -webkit-background-clip: text;
  background-clip: text;
  -webkit-text-fill-color: transparent;
}

.nav-item .username-gradient {
  background: linear-gradient(to right, #0000FF, #00008B);
  -webkit-background-clip: text;
  background-clip: text;
  -webkit-text-fill-color: transparent;
}
```



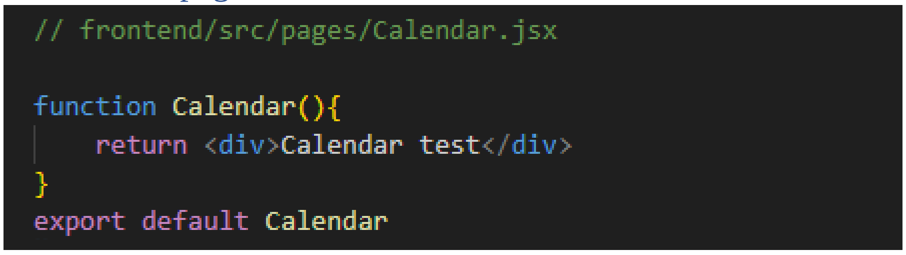
```css
.navbar-toggler {
  background-color: transparent;
  border: none;
  font-size: 24px;
  outline: none;
}

.navbar-toggler-icon {
  font-size: 24px;
  color: black;
}

@media (max-width: 992px) {
  .nav {
    display: none;
  }

  .nav.d-block {
    display: block;
  }
}
```

3.24 Update Layout and write there Header



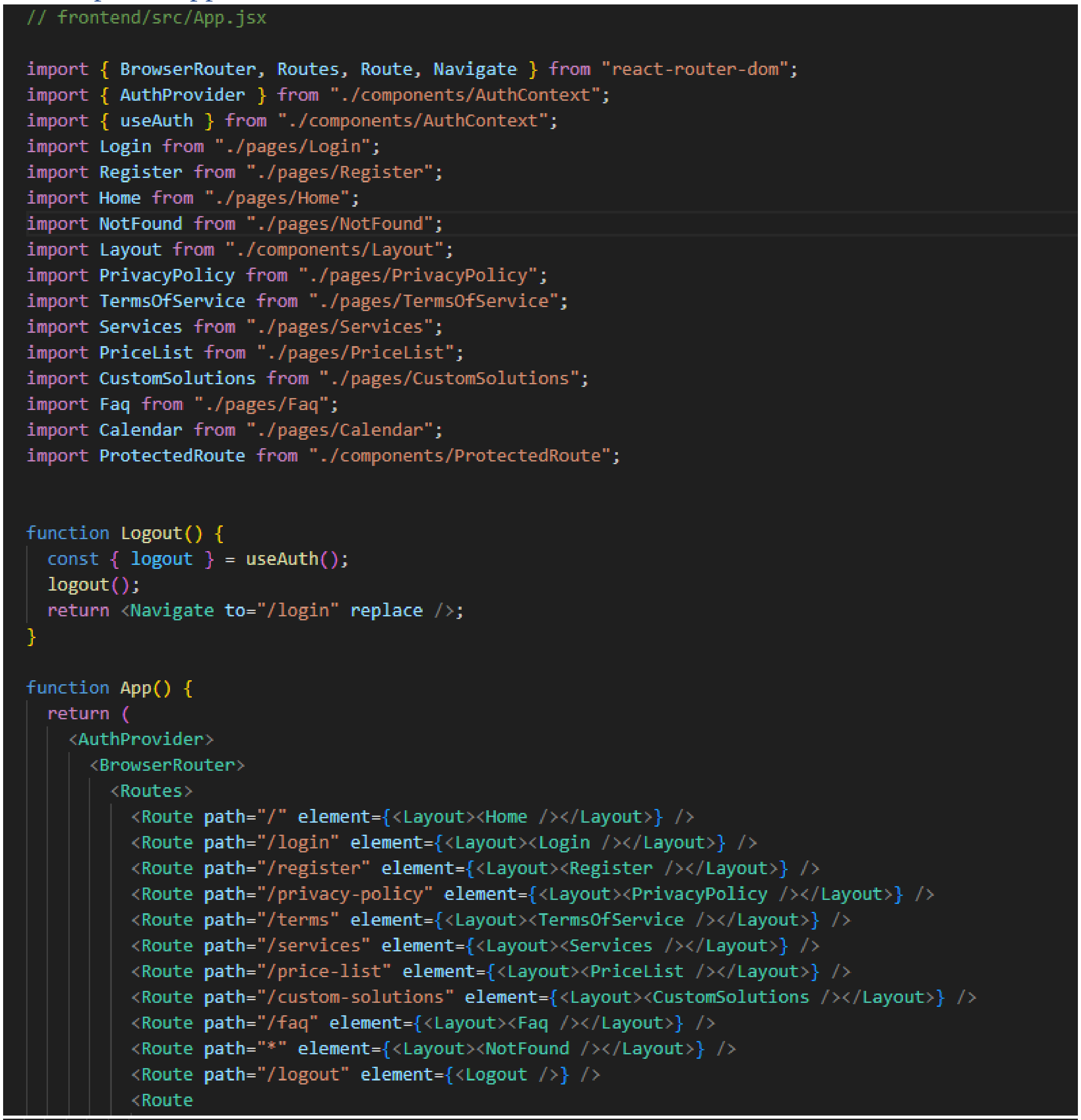
```
// frontend/src/components/Layout.jsx

import React from "react";
import Header from "./Header";
import Footer from "./Footer";

function Layout({ children }) {
  return (
    <div className="d-flex flex-column min-vh-100">
      <Header />
      <main className="flex-grow-1" style={{ paddingTop: '70px' }}>
        {children}
      </main>
      <Footer />
    </div>
  );
}

export default Layout;
```

3.25 Create page Calendar

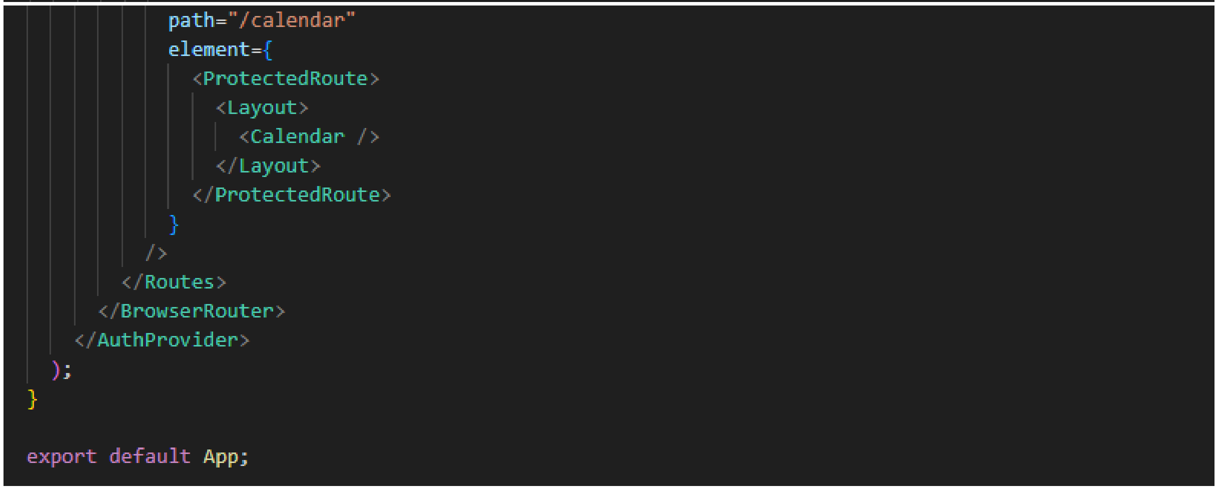


```
// frontend/src/pages/Calendar.jsx

function Calendar(){
    return <div>Calendar test</div>
}
export default Calendar
```

3.26 Update App with new connection

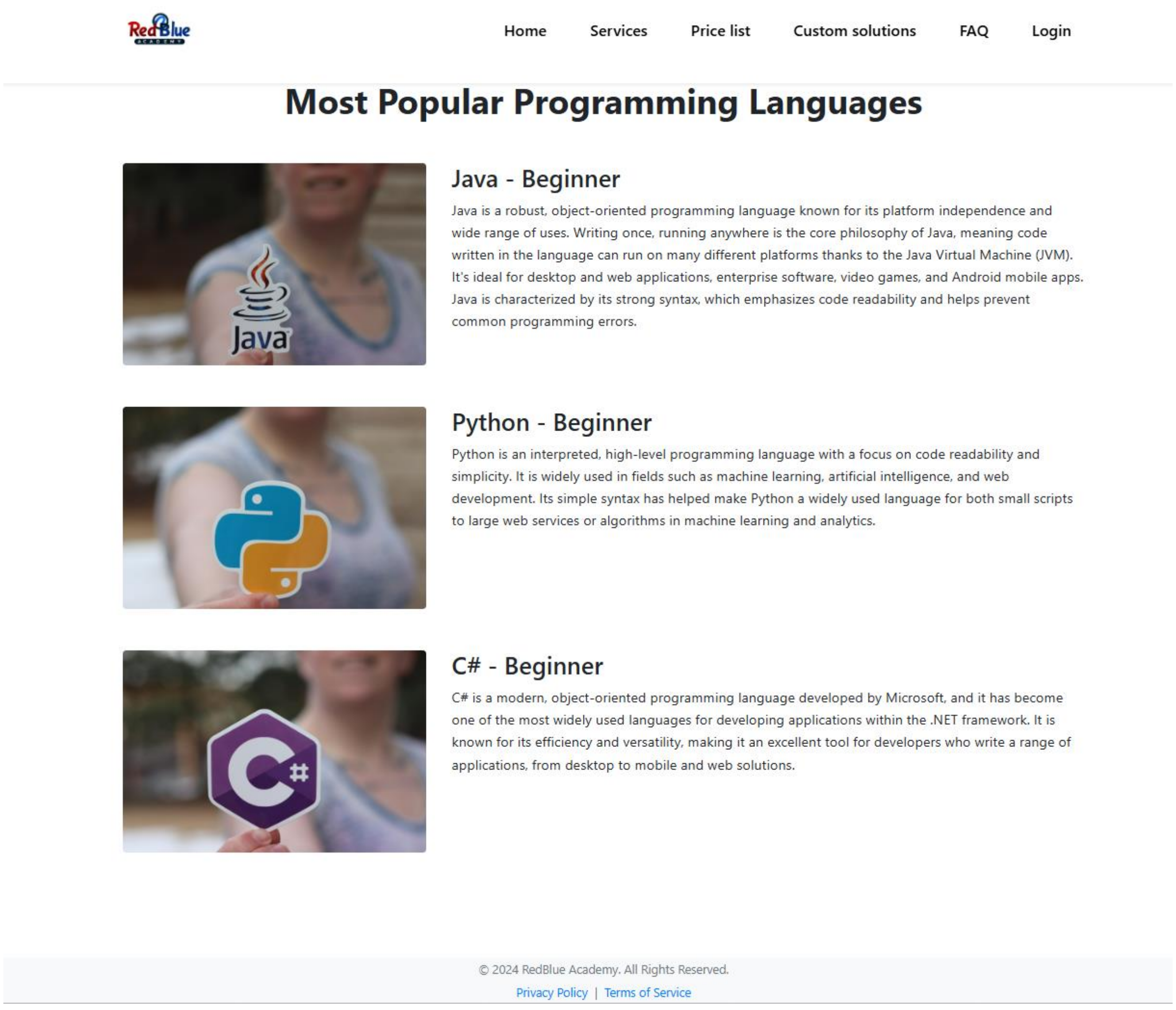


```jsx
// frontend/src/App.jsx

import { BrowserRouter, Routes, Route, Navigate } from "react-router-dom";
import { AuthProvider } from "./components/AuthContext";
import { useAuth } from "./components/AuthContext";
import Login from "./pages/Login";
import Register from "./pages/Register";
import Home from "./pages/Home";
import NotFound from "./pages/NotFound";
import Layout from "./components/Layout";
import PrivacyPolicy from "./pages/PrivacyPolicy";
import TermsOfService from "./pages/TermsOfService";
import Services from "./pages/Services";
import PriceList from "./pages/PriceList";
import CustomSolutions from "./pages/CustomSolutions";
import Faq from "./pages/Faq";
import Calendar from "./pages/Calendar";
import ProtectedRoute from "./components/ProtectedRoute";


function Logout() {
  const { logout } = useAuth();
  logout();
  return <Navigate to="/login" replace />;
}

function App() {
  return (
    <AuthProvider>
      <BrowserRouter>
        <Routes>
          <Route path="/" element={<Layout><Home /></Layout>} />
          <Route path="/login" element={<Layout><Login /></Layout>} />
          <Route path="/register" element={<Layout><Register /></Layout>} />
          <Route path="/privacy-policy" element={<Layout><PrivacyPolicy /></Layout>} />
          <Route path="/terms" element={<Layout><TermsOfService /></Layout>} />
          <Route path="/services" element={<Layout><Services /></Layout>} />
          <Route path="/price-list" element={<Layout><PriceList /></Layout>} />
          <Route path="/custom-solutions" element={<Layout><CustomSolutions /></Layout>} />
          <Route path="/faq" element={<Layout><Faq /></Layout>} />
          <Route path="*" element={<Layout><NotFound /></Layout>} />
          <Route path="/logout" element={<Logout />} />
          <Route
```



```jsx
            path="/calendar"
            element={
              <ProtectedRoute>
                <Layout>
                  <Calendar />
                </Layout>
              </ProtectedRoute>
            }
          />
        </Routes>
      </BrowserRouter>
    </AuthProvider>
  );
}

export default App;
```

3.27 Run servers and check pages



Most Popular Programming Languages

Java - Beginner

Java is a robust, object-oriented programming language known for its platform independence and wide range of uses. Writing once, running anywhere is the core philosophy of Java, meaning code written in the language can run on many different platforms thanks to the Java Virtual Machine (JVM). It's ideal for desktop and web applications, enterprise software, video games, and Android mobile apps. Java is characterized by its strong syntax, which emphasizes code readability and helps prevent common programming errors.

Python - Beginner

Python is an interpreted, high-level programming language with a focus on code readability and simplicity. It is widely used in fields such as machine learning, artificial intelligence, and web development. Its simple syntax has helped make Python a widely used language for both small scripts to large web services or algorithms in machine learning and analytics.

C# - Beginner

C# is a modern, object-oriented programming language developed by Microsoft, and it has become one of the most widely used languages for developing applications within the .NET framework. It is known for its efficiency and versatility, making it an excellent tool for developers who write a range of applications, from desktop to mobile and web solutions.

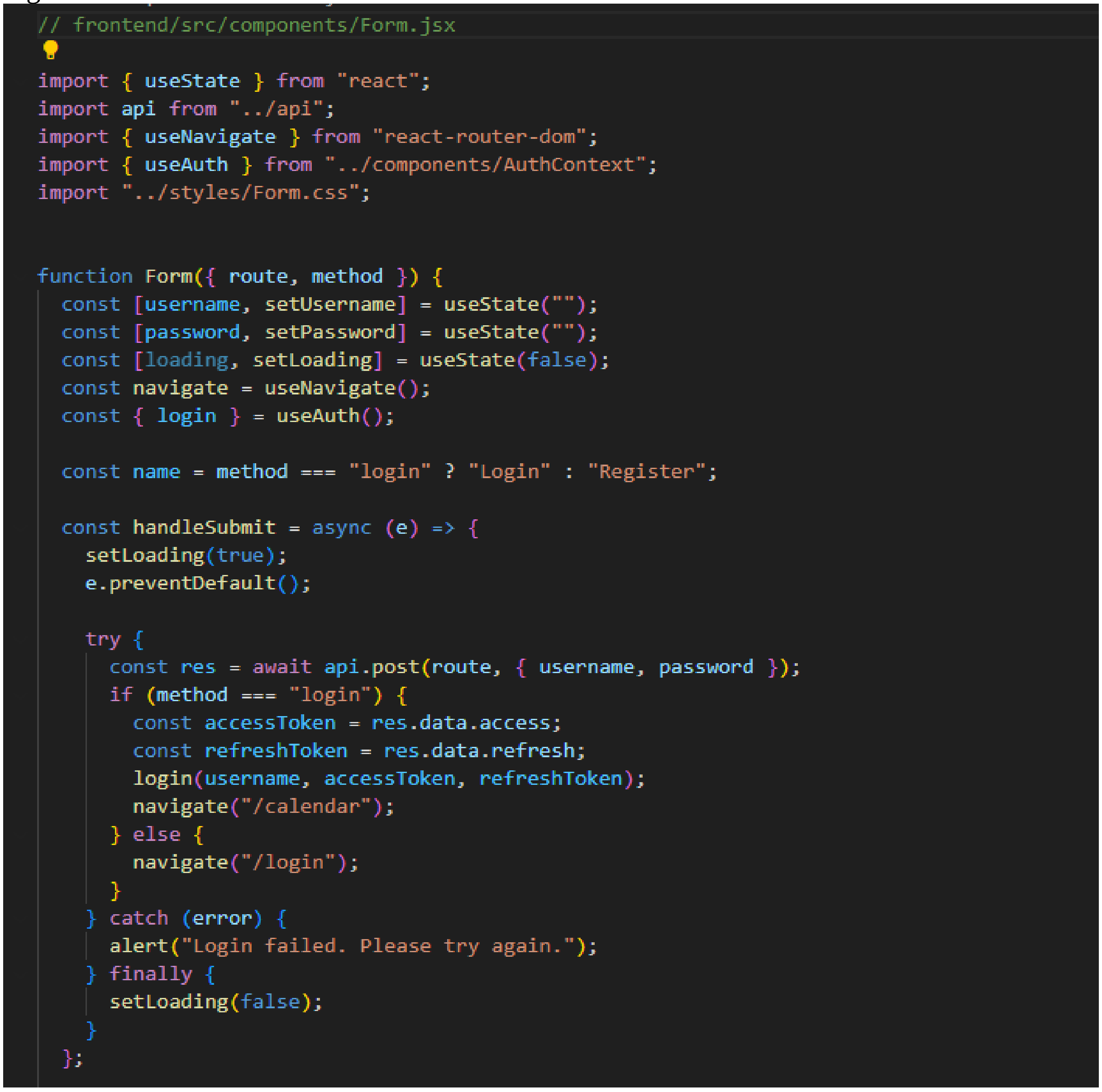
© 2024 RedBlue Academy. All Rights Reserved.

Privacy Policy | Terms of Service

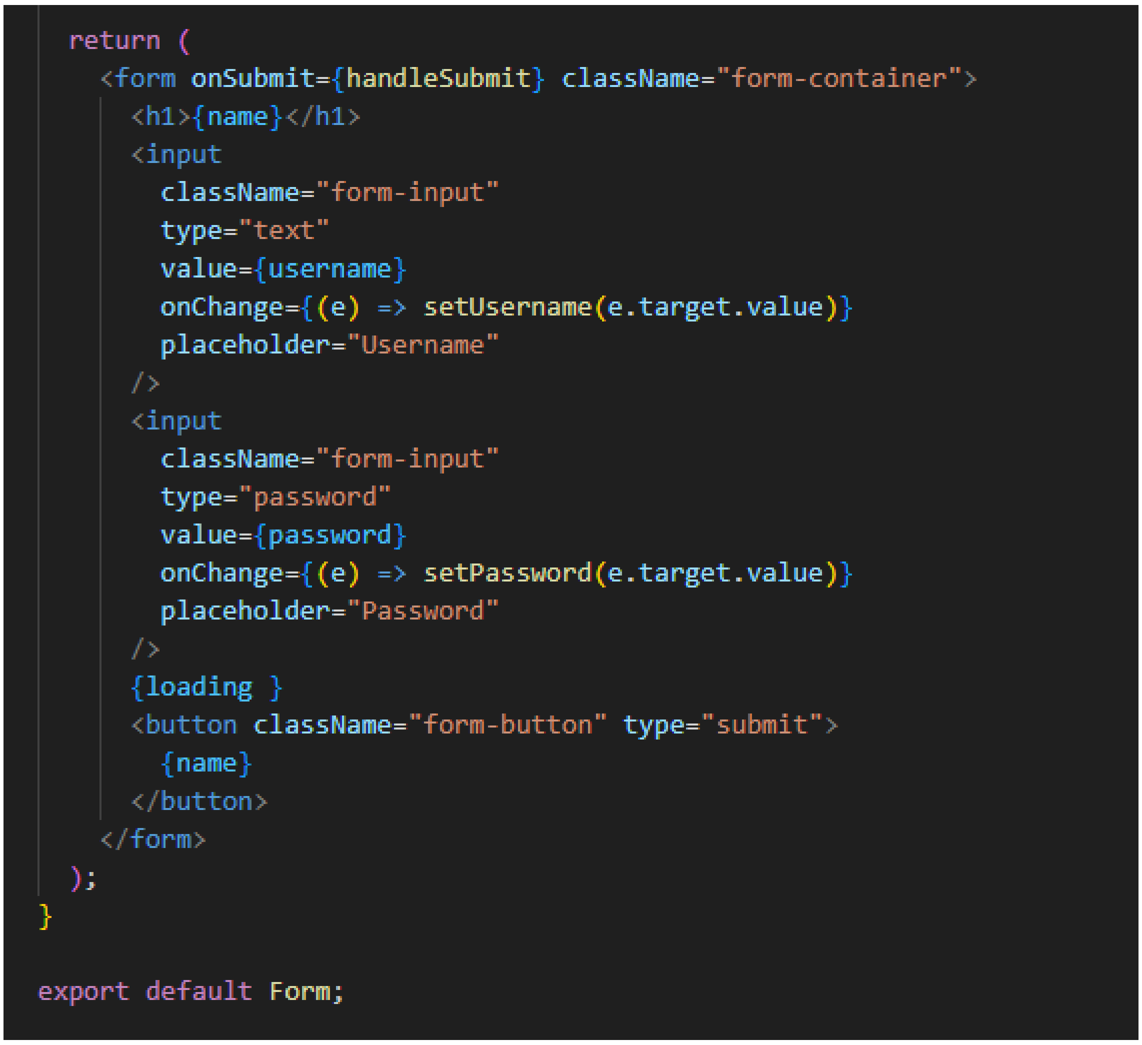
3.28 Create Component Form.jsx with style

The `Form` component provides a flexible login or registration form based on the `method` prop. Key functionalities include:

- Submitting the form to an API endpoint and handling login (with token storage) or registration.
- Managing loading and error states during submission.
- Dynamically adjusting the form's title and button text based on the action (login or register



```
// frontend/src/components/Form.jsx

import { useState } from "react";
import api from "../api";
import { useNavigate } from "react-router-dom";
import { useAuth } from "../components/AuthContext";
import "../styles/Form.css";


function Form({ route, method }) {
  const [username, setUsername] = useState("");
  const [password, setPassword] = useState("");
  const [loading, setLoading] = useState(false);
  const navigate = useNavigate();
  const { login } = useAuth();

  const name = method === "login" ? "Login" : "Register";

  const handleSubmit = async (e) => {
    setLoading(true);
    e.preventDefault();

    try {
      const res = await api.post(route, { username, password });
      if (method === "login") {
        const accessToken = res.data.access;
        const refreshToken = res.data.refresh;
        login(username, accessToken, refreshToken);
        navigate("/calendar");
      } else {
        navigate("/login");
      }
    } catch (error) {
      alert("Login failed. Please try again.");
    } finally {
      setLoading(false);
    }
  };
```



```
  return (
    <form onSubmit={handleSubmit} className="form-container">
      <h1>{name}</h1>
      <input
        className="form-input"
        type="text"
        value={username}
        onChange={(e) => setUsername(e.target.value)}
        placeholder="Username"
      />
      <input
        className="form-input"
        type="password"
        value={password}
        onChange={(e) => setPassword(e.target.value)}
        placeholder="Password"
      />
      {loading }
      <button className="form-button" type="submit">
        {name}
      </button>
    </form>
  );
}

export default Form;
```

```
function Form({ route, method }) {
    const [username, setUsername] = useState("");
    const [password, setPassword] = useState("");
    const [loading, setLoading] = useState(false);
    const navigate = useNavigate();
    const { login } = useAuth();
```

- **`username, password`**: States to manage the input values for username and password.
- **`loading`**: Indicates whether the form submission is in progress.
- **`navigate`**: A function for navigation between pages.
- **`login`**: A function from the authentication context to handle login.

```
const name = method === "login" ? "Login" : "Register";
```

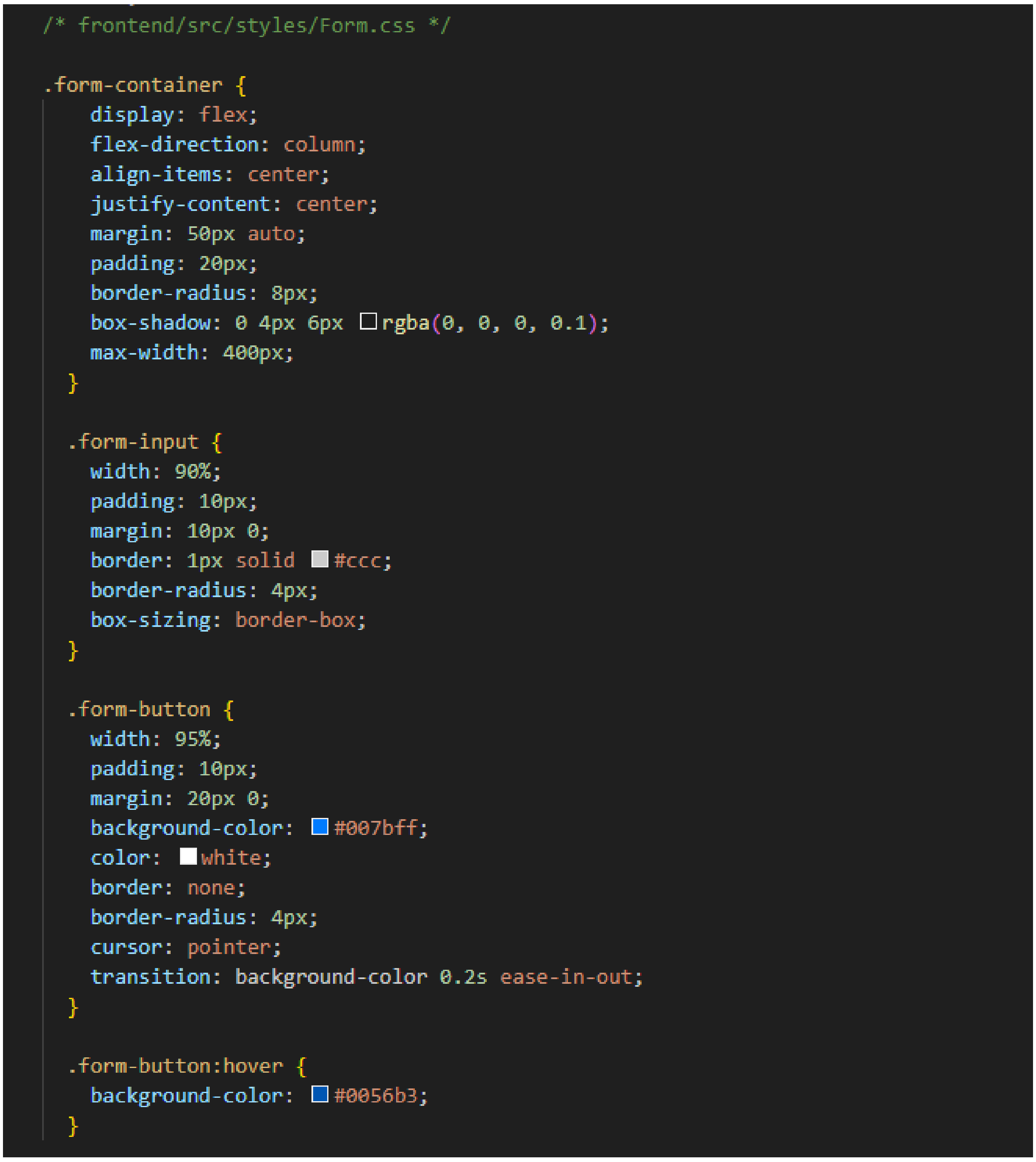
Purpose: Sets the form title and button text dynamically based on the `method` (either "login" or "register").

```
const handleSubmit = async (e) => {
    setLoading(true);
    e.preventDefault();
```

Purpose: The main function called on form submission. It first sets `loading` to `true` and prevents the default form submission behavior (e.g., page refresh).

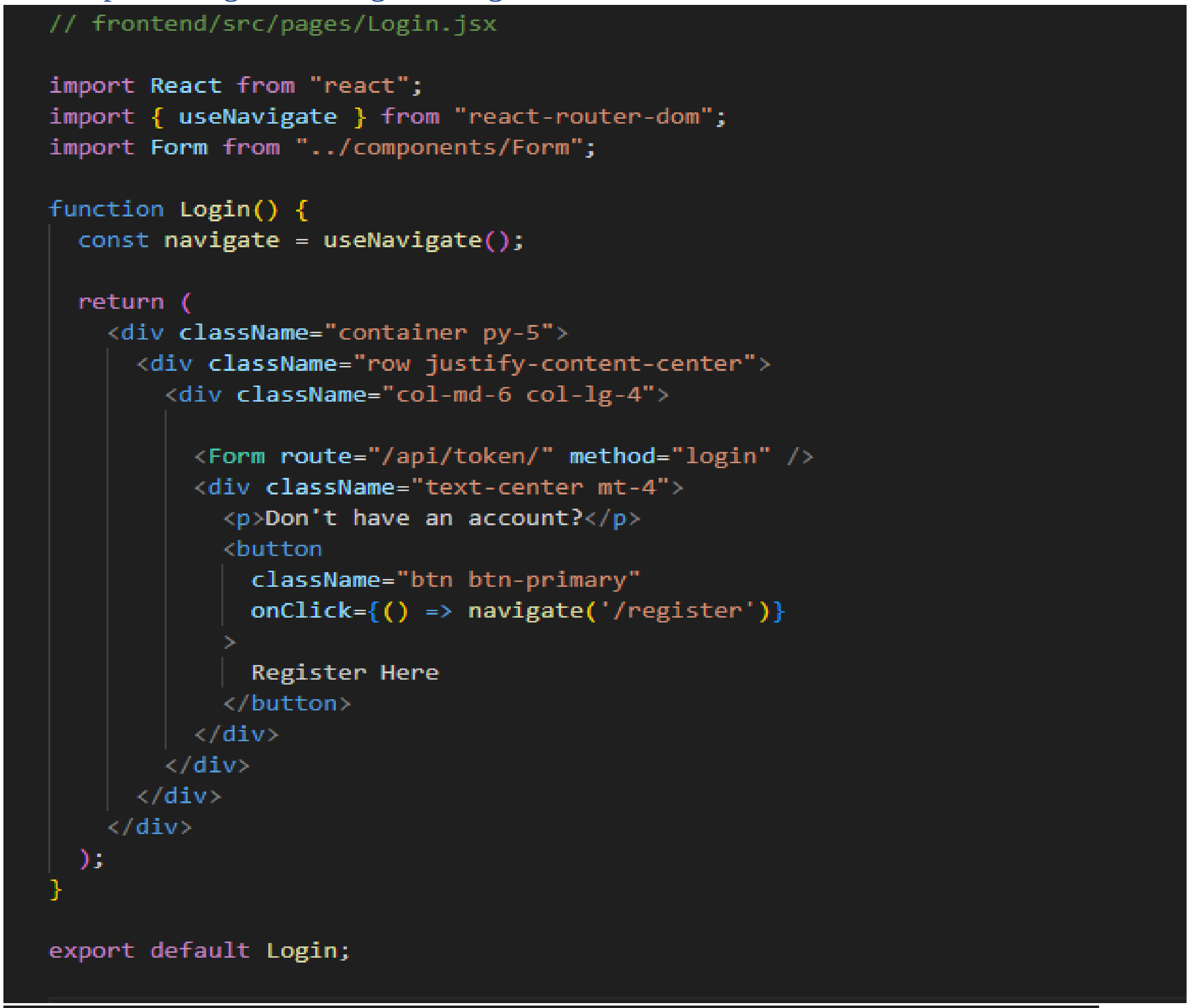
```
try {
    const res = await api.post(route, { username, password });
    if (method === "login") {
        const accessToken = res.data.access;
        const refreshToken = res.data.refresh;
        login(username, accessToken, refreshToken);
        navigate("/calendar");
    } else {
        navigate("/login");
    }
} catch (error) {
    alert("Login failed. Please try again.");
} finally {
    setLoading(false);
}
```

- **`api.post(route, { username, password })`**: Sends a POST request to the API with login or registration data.
- **Login (`if method === "login"`)**:
  - If the method is "login," it retrieves `accessToken` and `refreshToken` from the response, then calls the `login` function to store these tokens.
  - Redirects to the `/calendar` page.
- **Registration (`else`)**:
  - If the method is not "login," the user is redirected to the login page (`/login`) after registration.
- **Error Handling**: Shows an alert if login or registration fails.
- **`finally`**: Always sets `loading` back to `false` after processing, to disable the loading indicator.

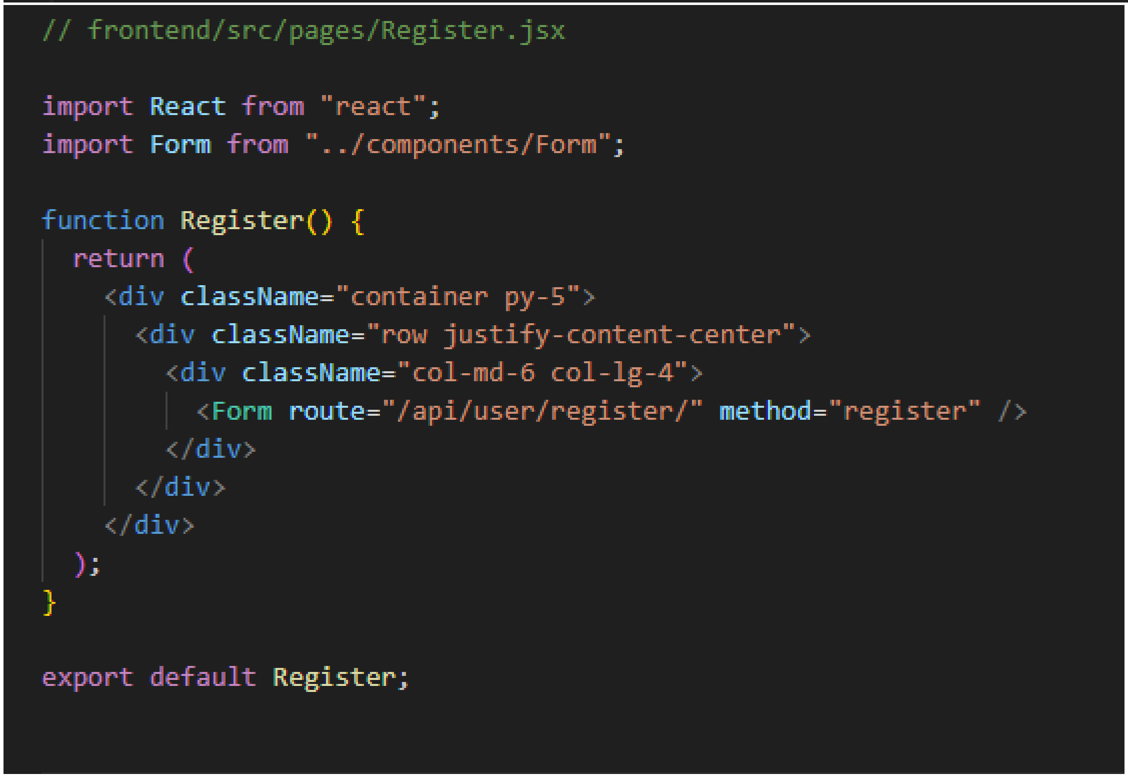


```css
/* frontend/src/styles/Form.css */

.form-container {
    display: flex;
    flex-direction: column;
    align-items: center;
    justify-content: center;
    margin: 50px auto;
    padding: 20px;
    border-radius: 8px;
    box-shadow: 0 4px 6px rgba(0, 0, 0, 0.1);
    max-width: 400px;
  }

  .form-input {
    width: 90%;
    padding: 10px;
    margin: 10px 0;
    border: 1px solid #ccc;
    border-radius: 4px;
    box-sizing: border-box;
  }

  .form-button {
    width: 95%;
    padding: 10px;
    margin: 20px 0;
    background-color: #007bff;
    color: white;
    border: none;
    border-radius: 4px;
    cursor: pointer;
    transition: background-color 0.2s ease-in-out;
  }

  .form-button:hover {
    background-color: #0056b3;
  }
```

3.29 Update Login and Register Page

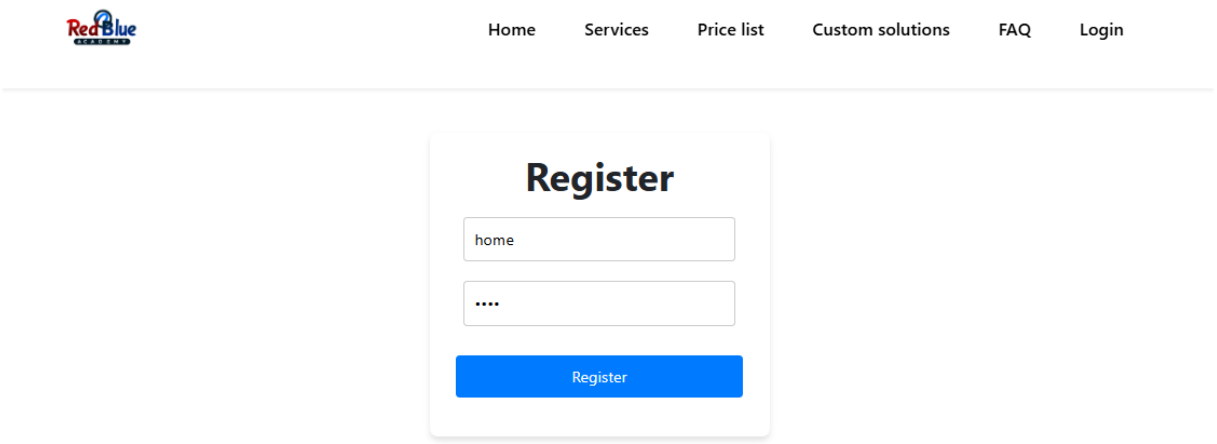


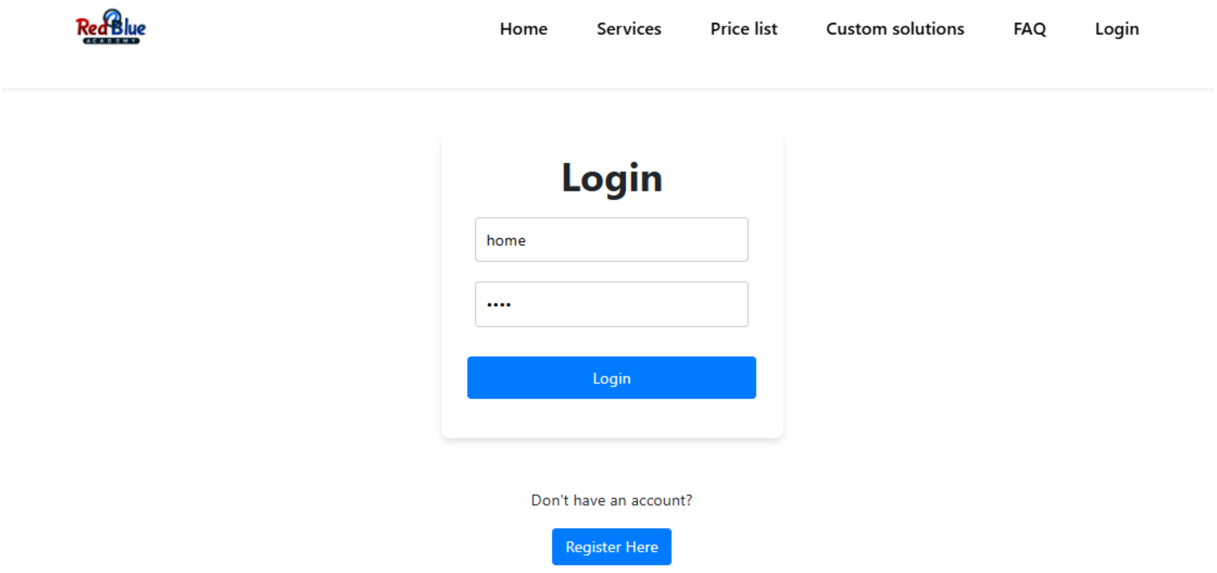
```
// frontend/src/pages/Login.jsx

import React from "react";
import { useNavigate } from "react-router-dom";
import Form from "../components/Form";

function Login() {
  const navigate = useNavigate();

  return (
    <div className="container py-5">
      <div className="row justify-content-center">
        <div className="col-md-6 col-lg-4">

          <Form route="/api/token/" method="login" />
          <div className="text-center mt-4">
            <p>Don't have an account?</p>
            <button
              className="btn btn-primary"
              onClick={() => navigate('/register')}
            >
              Register Here
            </button>
          </div>
        </div>
      </div>
    </div>
  );
}

export default Login;
```

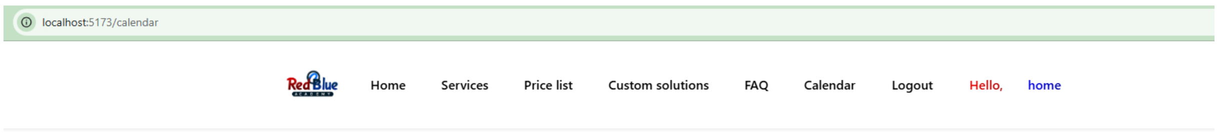


```
// frontend/src/pages/Register.jsx

import React from "react";
import Form from "../components/Form";

function Register() {
  return (
    <div className="container py-5">
      <div className="row justify-content-center">
        <div className="col-md-6 col-lg-4">
          <Form route="/api/user/register/" method="register" />
        </div>
      </div>
    </div>
  );
}

export default Register;
```

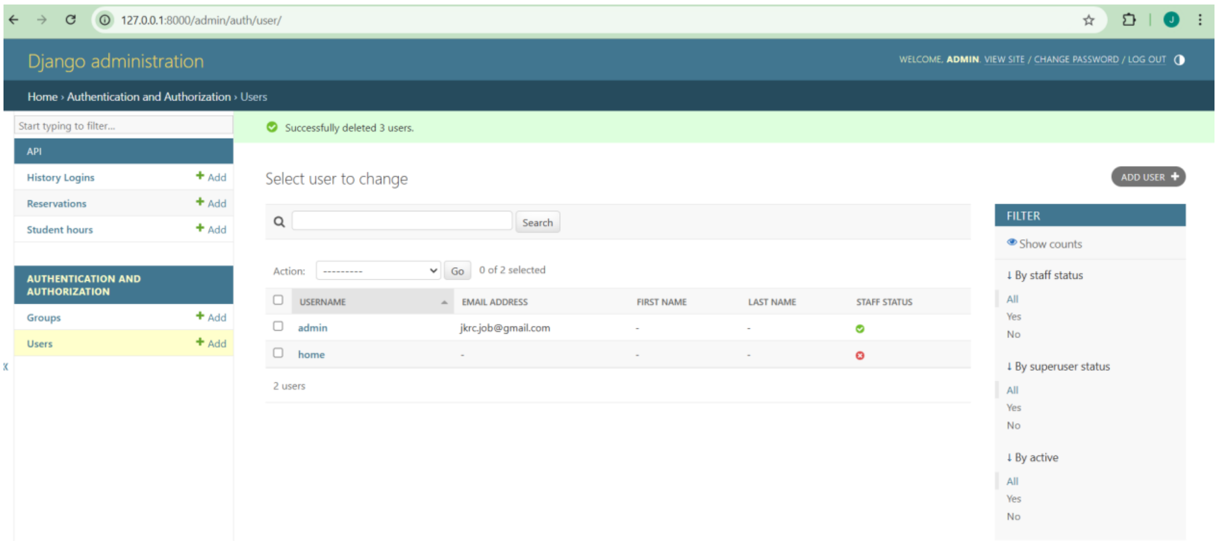
3.30 Start backend and frontend server server and make test







3.31 Open Django Administrator


127.0.0.1:8000/admin/auth/user/

Django administration

WELCOME, ADMIN. VIEW SITE / CHANGE PASSWORD / LOG OUT

Home › Authentication and Authorization › Users

Successfully deleted 3 users.

Select user to change

Action: --------- Go 0 of 2 selected

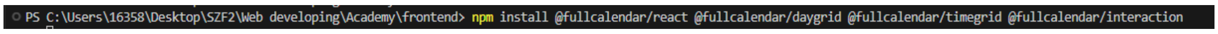
USERNAME	EMAIL ADDRESS	FIRST NAME	LAST NAME	STAFF STATUS
admin	jkrc.job@gmail.com	-	-	✓
home	-	-	-	✗

2 users

3.32 Install packeges for calendar

With these packages installed, your application can now display a dynamic calendar component with:

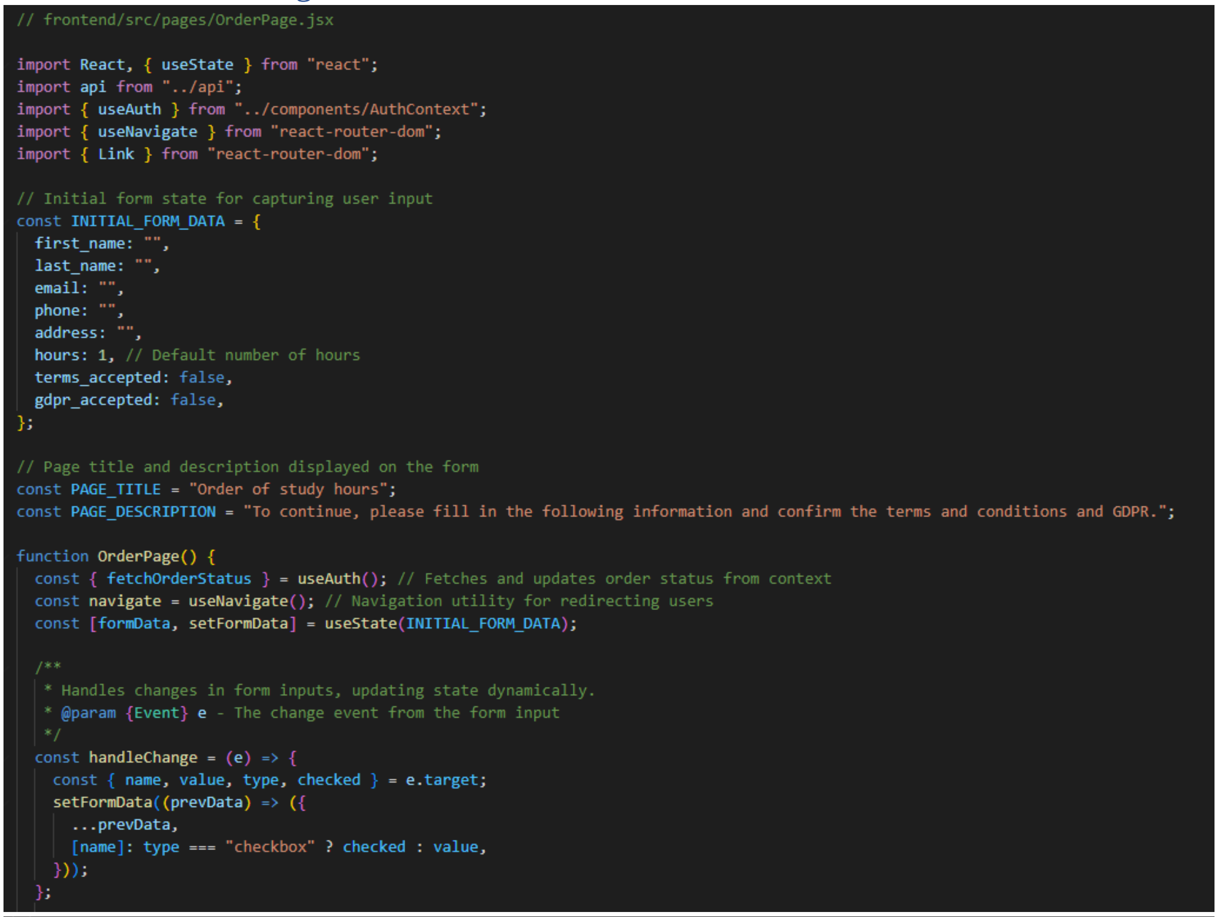
- **Different Views**: Support for monthly, weekly, and daily views.
- **Interactive Features**: Users can interact with events through actions like dragging to reschedule or clicking to view details.



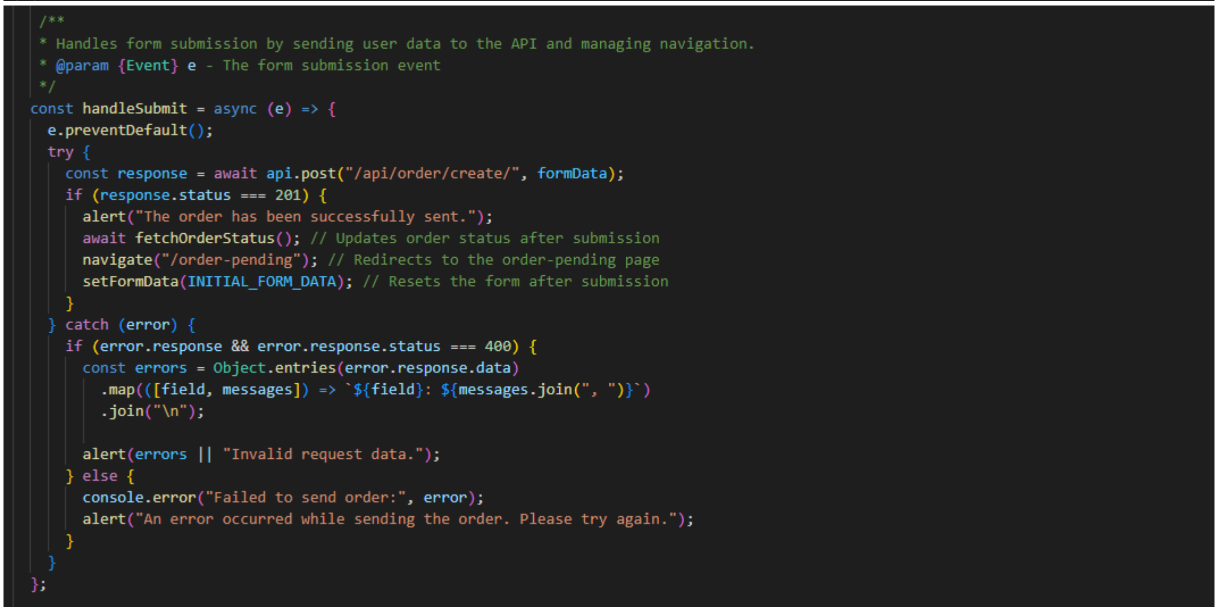
```
PS C:\Users\16358\Desktop\SZF2\Web developing\Academy\frontend> npm install @fullcalendar/react @fullcalendar/daygrid @fullcalendar/timegrid @fullcalendar/interaction
```

- **@fullcalendar/react**: This is the core FullCalendar package for React, which provides the main calendar component compatible with React applications.
- **@fullcalendar/daygrid**: This module allows the calendar to display a "day grid" view, which is typically used for monthly views where each day is shown as a grid cell.
- **@fullcalendar/timegrid**: Adds support for "time grid" views, such as daily or weekly views where each hour of the day is displayed. This is especially useful for scheduling events with specific start and end times.
- **@fullcalendar/interaction**: Enables interactivity features, including drag-and-drop functionality for moving events and click handling for selecting or editing events directly in the calendar.

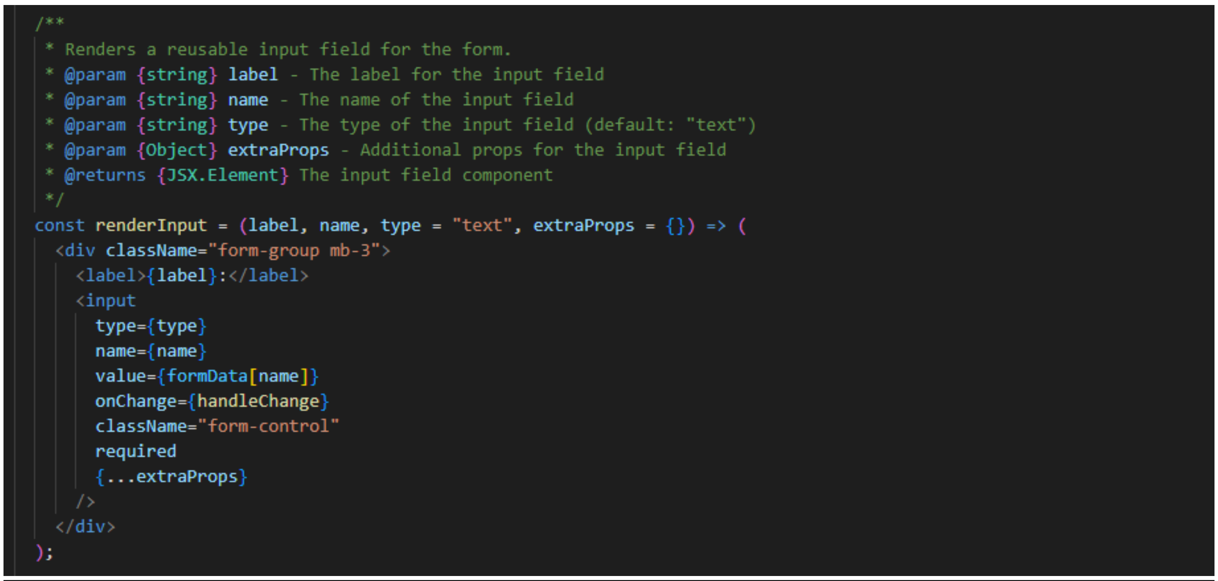
3.33 Create OrderPage



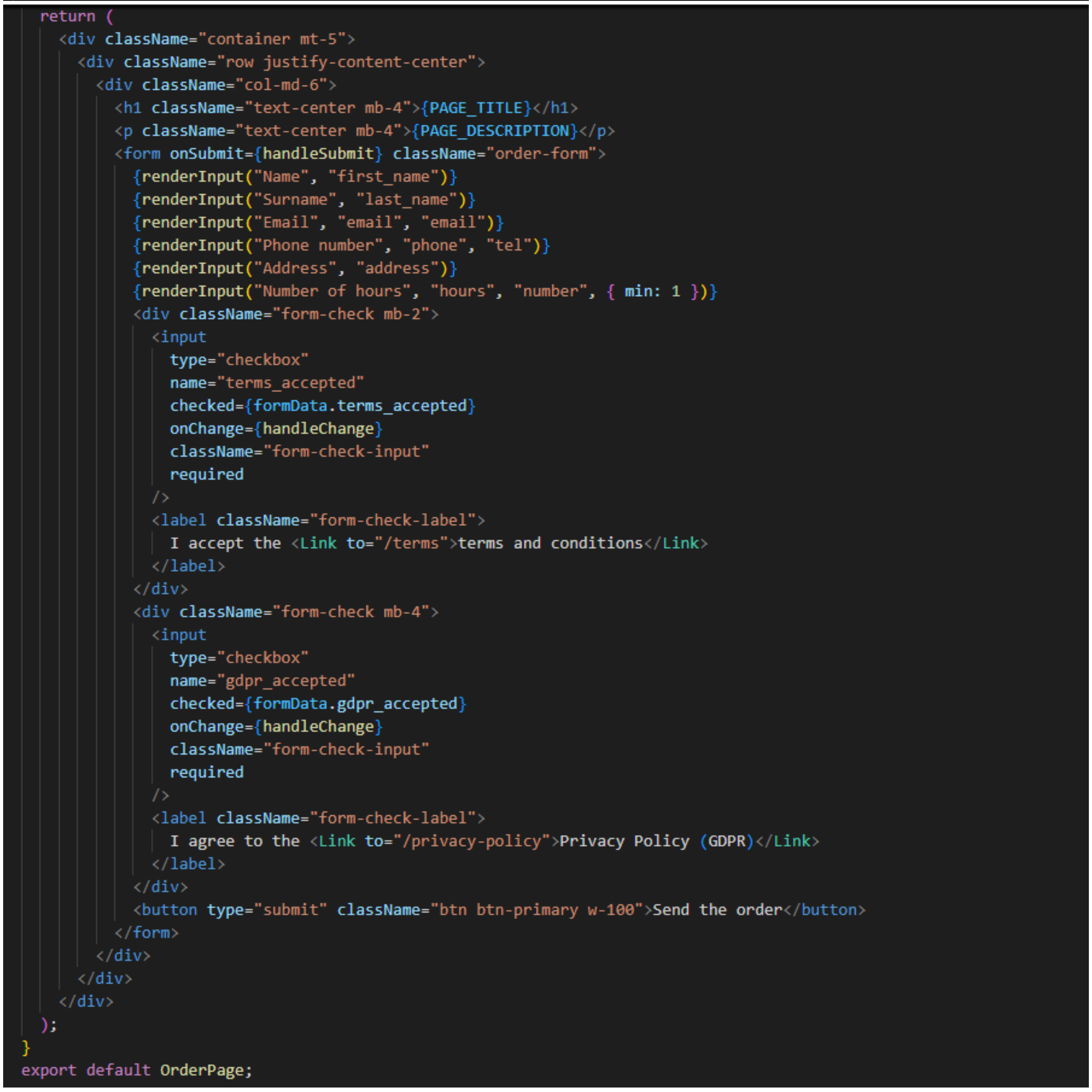
```jsx
// frontend/src/pages/OrderPage.jsx

import React, { useState } from "react";
import api from "../api";
import { useAuth } from "../components/AuthContext";
import { useNavigate } from "react-router-dom";
import { Link } from "react-router-dom";

// Initial form state for capturing user input
const INITIAL_FORM_DATA = {
  first_name: "",
  last_name: "",
  email: "",
  phone: "",
  address: "",
  hours: 1, // Default number of hours
  terms_accepted: false,
  gdpr_accepted: false,
};

// Page title and description displayed on the form
const PAGE_TITLE = "Order of study hours";
const PAGE_DESCRIPTION = "To continue, please fill in the following information and confirm the terms and conditions and GDPR.";

function OrderPage() {
  const { fetchOrderStatus } = useAuth(); // Fetches and updates order status from context
  const navigate = useNavigate(); // Navigation utility for redirecting users
  const [formData, setFormData] = useState(INITIAL_FORM_DATA);

  /**
   * Handles changes in form inputs, updating state dynamically.
   * @param {Event} e - The change event from the form input
   */
  const handleChange = (e) => {
    const { name, value, type, checked } = e.target;
    setFormData((prevData) => ({
      ...prevData,
      [name]: type === "checkbox" ? checked : value,
    }));
  };
```



```jsx
  /**
   * Handles form submission by sending user data to the API and managing navigation.
   * @param {Event} e - The form submission event
   */
  const handleSubmit = async (e) => {
    e.preventDefault();
    try {
      const response = await api.post("/api/order/create/", formData);
      if (response.status === 201) {
        alert("The order has been successfully sent.");
        await fetchOrderStatus(); // Updates order status after submission
        navigate("/order-pending"); // Redirects to the order-pending page
        setFormData(INITIAL_FORM_DATA); // Resets the form after submission
      }
    } catch (error) {
      if (error.response && error.response.status === 400) {
        const errors = Object.entries(error.response.data)
          .map(([field, messages]) => `${field}: ${messages.join(", ")}`)
          .join("\n");

        alert(errors || "Invalid request data.");
      } else {
        console.error("Failed to send order:", error);
        alert("An error occurred while sending the order. Please try again.");
      }
    }
  };
```



```jsx
/**
 * Renders a reusable input field for the form.
 * @param {string} label - The label for the input field
 * @param {string} name - The name of the input field
 * @param {string} type - The type of the input field (default: "text")
 * @param {Object} extraProps - Additional props for the input field
 * @returns {JSX.Element} The input field component
 */
const renderInput = (label, name, type = "text", extraProps = {}) => (
  <div className="form-group mb-3">
    <label>{label}:</label>
    <input
      type={type}
      name={name}
      value={formData[name]}
      onChange={handleChange}
      className="form-control"
      required
      {...extraProps}
    />
  </div>
);
```

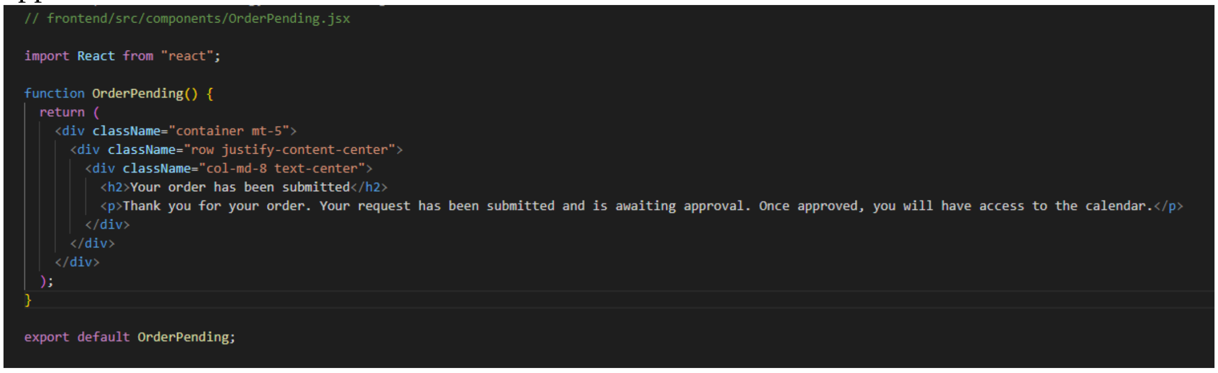


```jsx
  return (
    <div className="container mt-5">
      <div className="row justify-content-center">
        <div className="col-md-6">
          <h1 className="text-center mb-4">{PAGE_TITLE}</h1>
          <p className="text-center mb-4">{PAGE_DESCRIPTION}</p>
          <form onSubmit={handleSubmit} className="order-form">
            {renderInput("Name", "first_name")}
            {renderInput("Surname", "last_name")}
            {renderInput("Email", "email", "email")}
            {renderInput("Phone number", "phone", "tel")}
            {renderInput("Address", "address")}
            {renderInput("Number of hours", "hours", "number", { min: 1 })}
            <div className="form-check mb-2">
              <input
                type="checkbox"
                name="terms_accepted"
                checked={formData.terms_accepted}
                onChange={handleChange}
                className="form-check-input"
                required
              />
              <label className="form-check-label">
                I accept the <Link to="/terms">terms and conditions</Link>
              </label>
            </div>
            <div className="form-check mb-4">
              <input
                type="checkbox"
                name="gdpr_accepted"
                checked={formData.gdpr_accepted}
                onChange={handleChange}
                className="form-check-input"
                required
              />
              <label className="form-check-label">
                I agree to the <Link to="/privacy-policy">Privacy Policy (GDPR)</Link>
              </label>
            </div>
            <button type="submit" className="btn btn-primary w-100">Send the order</button>
          </form>
        </div>
      </div>
    </div>
  );
}
export default OrderPage;
```

This OrderPage component captures user details and validates necessary fields before submitting the order. It requires users to accept the terms and GDPR policy, ensuring compliance. Upon successful submission, it updates the order status and redirects the user to the order pending page. This component provides a streamlined interface for ordering study hours with essential validations and error handling.

3.34 Create OrderPending

The OrderPending component displays a message to inform the user that their order has been successfully submitted and is awaiting approval. This component is designed to be rendered when the user is logged in but does not yet have an approved order for study hours. The ProtectedRoute component likely controls access, ensuring that users without an approved order are redirected here.



```
// frontend/src/components/OrderPending.jsx

import React from "react";

function OrderPending() {
  return (
    <div className="container mt-5">
      <div className="row justify-content-center">
        <div className="col-md-8 text-center">
          <h2>Your order has been submitted</h2>
          <p>Thank you for your order. Your request has been submitted and is awaiting approval. Once approved, you will have access to the calendar.</p>
        </div>
      </div>
    </div>
  );
}

export default OrderPending;
```

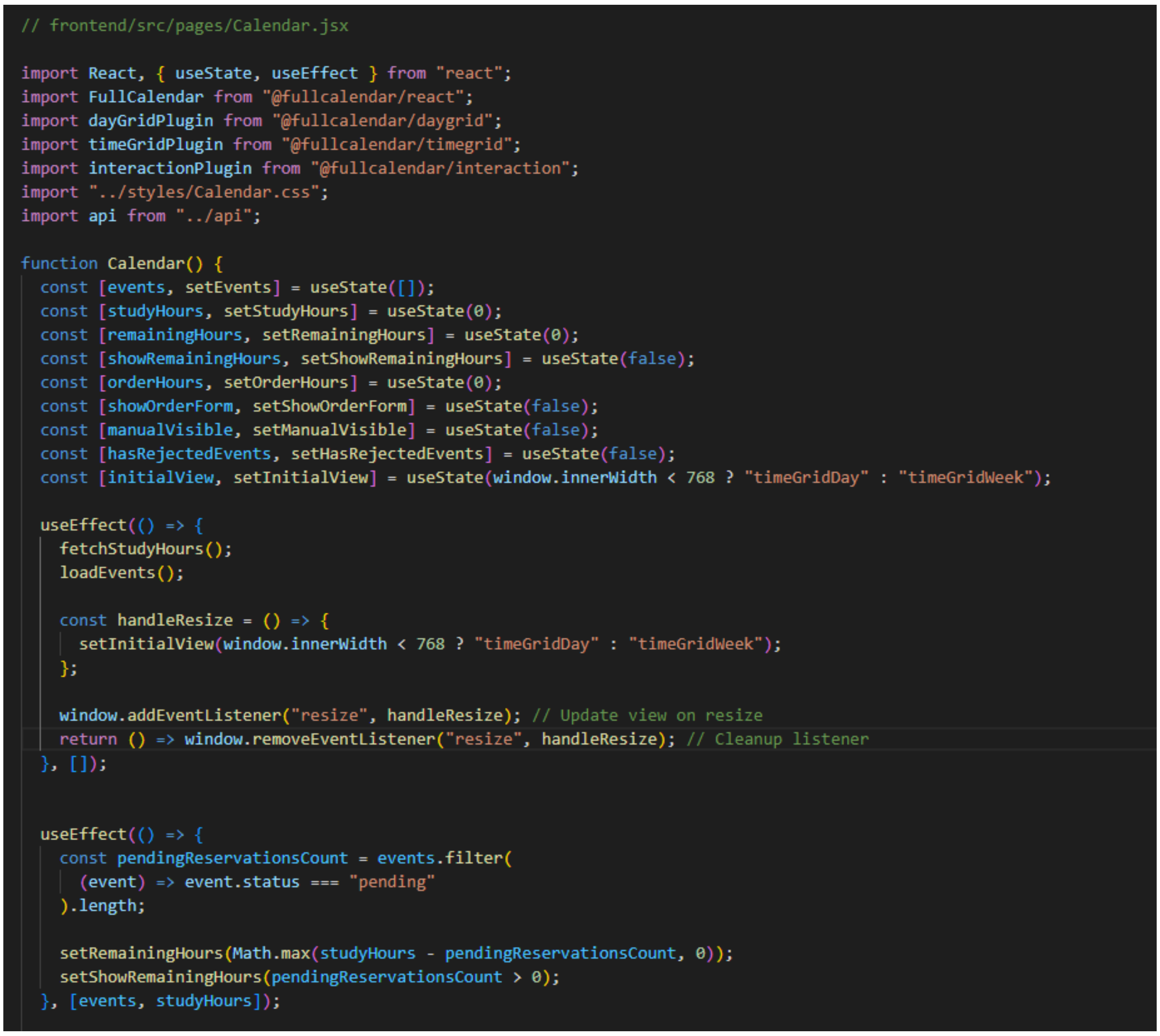
3.35 Update Calendar

The Calendar component:

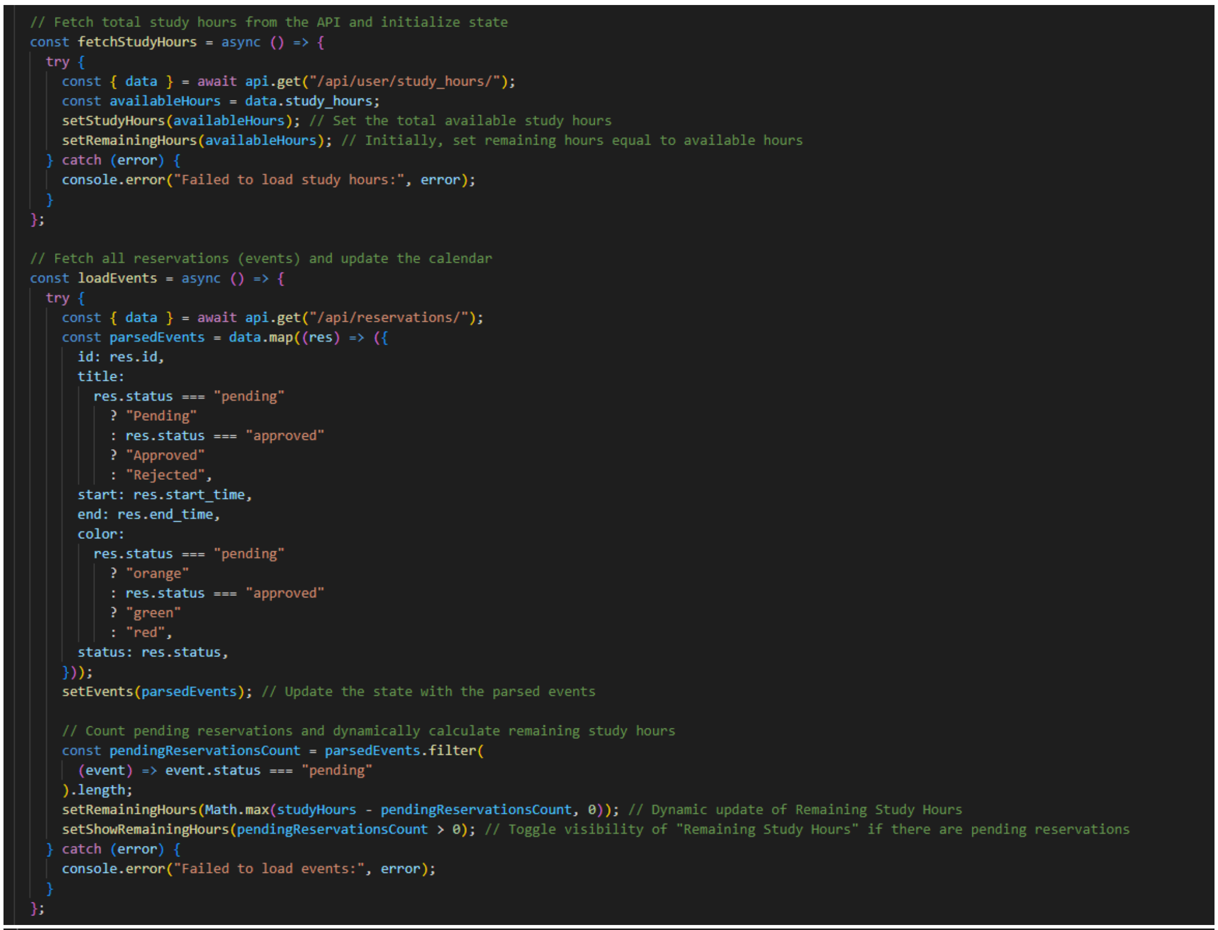
- Dynamically displays **available study hours** and **remaining hours**, recalculating them after considering pending reservations.
- Loads reservations from the backend and color-codes them based on their status:
    - **Green**: Approved reservations.
    - **Orange**: Pending reservations.
    - **Red**: Rejected reservations.
- Allows users to:
    - Reserve new lessons for **future dates** by clicking on available time slots.
    - Delete pending reservations directly from the calendar.
    - Order additional study hours when the remaining hours reach zero.
- Provides a **user manual** that guides users on how to manage their calendar and reservations.
- Includes functionality to **clear rejected reservations** from the calendar upon user request.
- Responsively adjusts the calendar view:
    - Displays **week view** (`timeGridWeek`) for larger screens.
    - Switches to **day view** (`timeGridDay`) for smaller screens.

The component renders a structured layout containing:

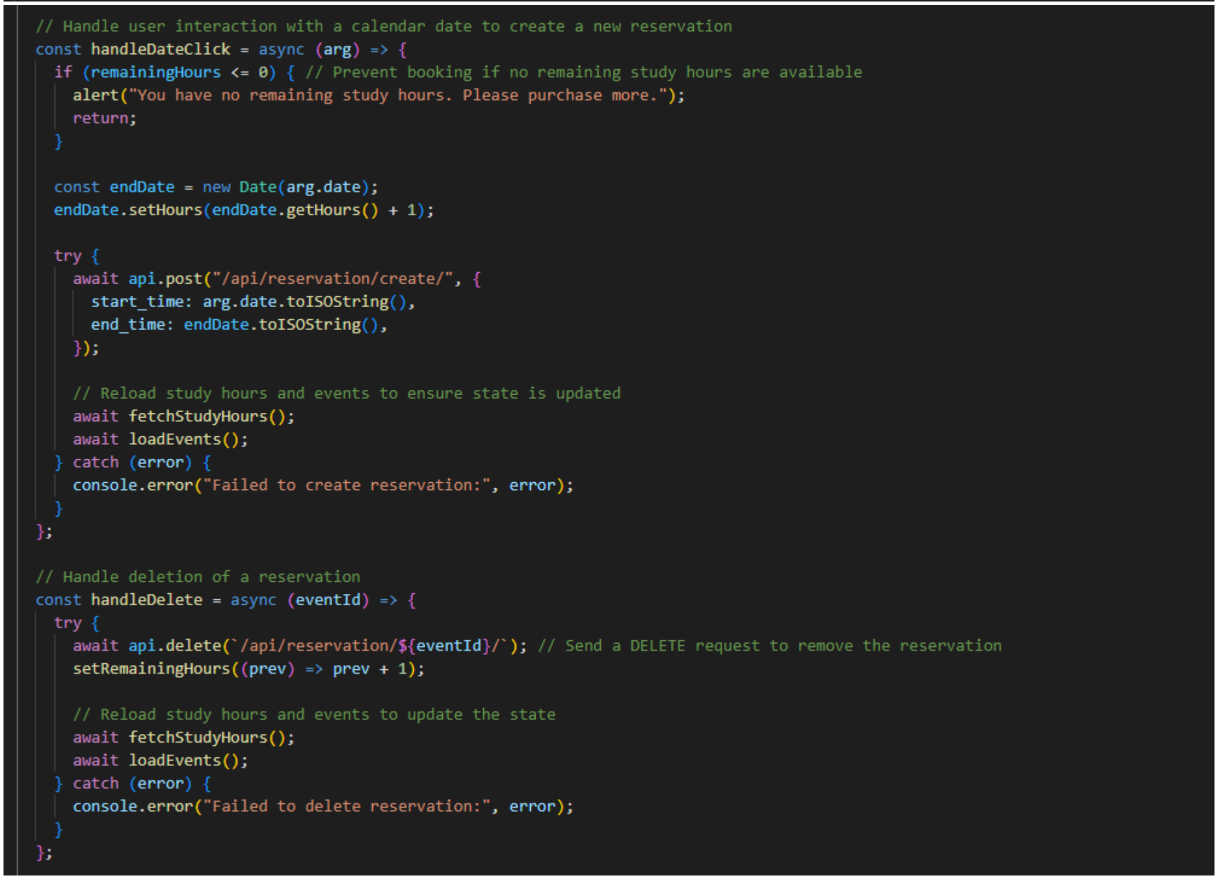
1. **Header**:
    - Displays the **title** ("Your Calendar") and a brief description of the page.
2. **Study Hours Display**:
    - Shows **available study hours**.
    - Dynamically updates to reflect **remaining hours** after considering pending reservations.
3. **Interactive Buttons**:
    - **"Show Manual"**: Displays the user manual in an overlay with step-by-step guidance.
    - **"Order More Hours"**: Opens a form to request additional study hours.
    - **"Clear Rejected Requests"**: Available when rejected reservations exist and clears them upon user action.
4. **Order Form Overlay**:
    - Collects the number of study hours the user wants to purchase.
    - Validates input and submits the order to the backend.
5. **Calendar**:
    - Built using the **FullCalendar** library and configured with:
        - **Plugins**: Includes `dayGridPlugin`, `timeGridPlugin`, and `interactionPlugin` for enhanced calendar functionality.
        - **Initial View**: Dynamically set to `timeGridDay` for small screens or `timeGridWeek` for larger screens.
        - **Time Range**: Restricted to the working hours of `07:00:00` to `22:00:00`.
        - **Valid Date Range**: Ensures users cannot select past dates for reservations.
        - **Events**: Loaded dynamically from the backend and updated in real-time.
        - **Handlers**:
            - `dateClick`: Enables users to create new reservations by clicking on an available time slot.
            - `eventContent`: Custom renderer for displaying event details with a **delete button** for pending reservations.
        - **Responsive Adjustments**:
            - Adjusts the calendar aspect ratio and view type dynamically based on the screen size.



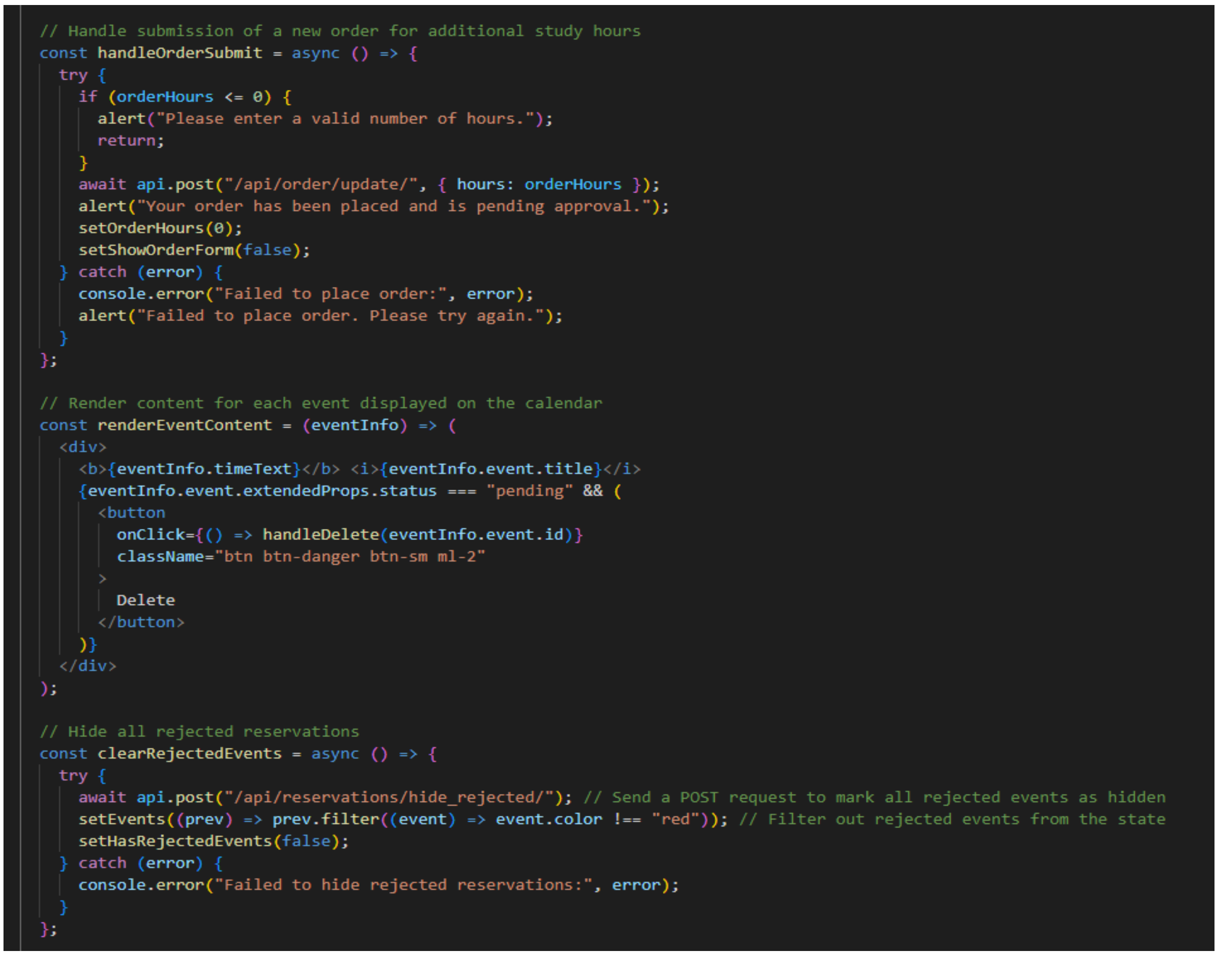
```jsx
// frontend/src/pages/Calendar.jsx

import React, { useState, useEffect } from "react";
import FullCalendar from "@fullcalendar/react";
import dayGridPlugin from "@fullcalendar/daygrid";
import timeGridPlugin from "@fullcalendar/timegrid";
import interactionPlugin from "@fullcalendar/interaction";
import "../styles/Calendar.css";
import api from "../api";

function Calendar() {
  const [events, setEvents] = useState([]);
  const [studyHours, setStudyHours] = useState(0);
  const [remainingHours, setRemainingHours] = useState(0);
  const [showRemainingHours, setShowRemainingHours] = useState(false);
  const [orderHours, setOrderHours] = useState(0);
  const [showOrderForm, setShowOrderForm] = useState(false);
  const [manualVisible, setManualVisible] = useState(false);
  const [hasRejectedEvents, setHasRejectedEvents] = useState(false);
  const [initialView, setInitialView] = useState(window.innerWidth < 768 ? "timeGridDay" : "timeGridWeek");

  useEffect(() => {
    fetchStudyHours();
    loadEvents();

    const handleResize = () => {
      setInitialView(window.innerWidth < 768 ? "timeGridDay" : "timeGridWeek");
    };

    window.addEventListener("resize", handleResize); // Update view on resize
    return () => window.removeEventListener("resize", handleResize); // Cleanup listener
  }, []);


  useEffect(() => {
    const pendingReservationsCount = events.filter(
      (event) => event.status === "pending"
    ).length;

    setRemainingHours(Math.max(studyHours - pendingReservationsCount, 0));
    setShowRemainingHours(pendingReservationsCount > 0);
  }, [events, studyHours]);
```



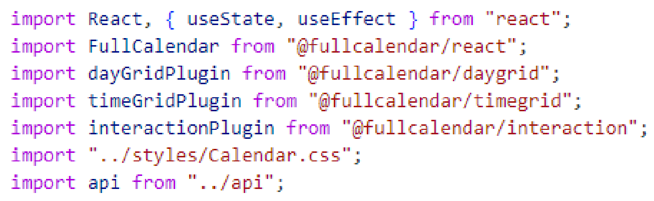
```
// Fetch total study hours from the API and initialize state
const fetchStudyHours = async () => {
  try {
    const { data } = await api.get("/api/user/study_hours/");
    const availableHours = data.study_hours;
    setStudyHours(availableHours); // Set the total available study hours
    setRemainingHours(availableHours); // Initially, set remaining hours equal to available hours
  } catch (error) {
    console.error("Failed to load study hours:", error);
  }
};

// Fetch all reservations (events) and update the calendar
const loadEvents = async () => {
  try {
    const { data } = await api.get("/api/reservations/");
    const parsedEvents = data.map((res) => ({
      id: res.id,
      title:
        res.status === "pending"
          ? "Pending"
          : res.status === "approved"
          ? "Approved"
          : "Rejected",
      start: res.start_time,
      end: res.end_time,
      color:
        res.status === "pending"
          ? "orange"
          : res.status === "approved"
          ? "green"
          : "red",
      status: res.status,
    }));
    setEvents(parsedEvents); // Update the state with the parsed events

    // Count pending reservations and dynamically calculate remaining study hours
    const pendingReservationsCount = parsedEvents.filter(
      (event) => event.status === "pending"
    ).length;
    setRemainingHours(Math.max(studyHours - pendingReservationsCount, 0)); // Dynamic update of Remaining Study Hours
    setShowRemainingHours(pendingReservationsCount > 0); // Toggle visibility of "Remaining Study Hours" if there are pending reservations
  } catch (error) {
    console.error("Failed to load events:", error);
  }
};
```



```
// Handle user interaction with a calendar date to create a new reservation
const handleDateClick = async (arg) => {
  if (remainingHours <= 0) { // Prevent booking if no remaining study hours are available
    alert("You have no remaining study hours. Please purchase more.");
    return;
  }

  const endDate = new Date(arg.date);
  endDate.setHours(endDate.getHours() + 1);

  try {
    await api.post("/api/reservation/create/", {
      start_time: arg.date.toISOString(),
      end_time: endDate.toISOString(),
    });

    // Reload study hours and events to ensure state is updated
    await fetchStudyHours();
    await loadEvents();
  } catch (error) {
    console.error("Failed to create reservation:", error);
  }
};

// Handle deletion of a reservation
const handleDelete = async (eventId) => {
  try {
    await api.delete(`/api/reservation/${eventId}/`); // Send a DELETE request to remove the reservation
    setRemainingHours((prev) => prev + 1);

    // Reload study hours and events to update the state
    await fetchStudyHours();
    await loadEvents();
  } catch (error) {
    console.error("Failed to delete reservation:", error);
  }
};
```



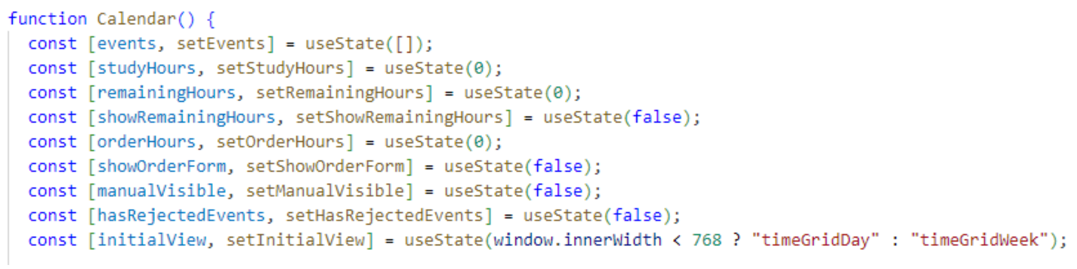
```
// Handle submission of a new order for additional study hours
const handleOrderSubmit = async () => {
  try {
    if (orderHours <= 0) {
      alert("Please enter a valid number of hours.");
      return;
    }
    await api.post("/api/order/update/", { hours: orderHours });
    alert("Your order has been placed and is pending approval.");
    setOrderHours(0);
    setShowOrderForm(false);
  } catch (error) {
    console.error("Failed to place order:", error);
    alert("Failed to place order. Please try again.");
  }
};

// Render content for each event displayed on the calendar
const renderEventContent = (eventInfo) => (
  <div>
    <b>{eventInfo.timeText}</b> <i>{eventInfo.event.title}</i>
    {eventInfo.event.extendedProps.status === "pending" && (
      <button
        onClick={() => handleDelete(eventInfo.event.id)}
        className="btn btn-danger btn-sm ml-2"
      >
        Delete
      </button>
    )}
  </div>
);

// Hide all rejected reservations
const clearRejectedEvents = async () => {
  try {
    await api.post("/api/reservations/hide_rejected/"); // Send a POST request to mark all rejected events as hidden
    setEvents((prev) => prev.filter((event) => event.color !== "red")); // Filter out rejected events from the state
    setHasRejectedEvents(false);
  } catch (error) {
    console.error("Failed to hide rejected reservations:", error);
  }
};
```



```
import React, { useState, useEffect } from "react";
import FullCalendar from "@fullcalendar/react";
import dayGridPlugin from "@fullcalendar/daygrid";
import timeGridPlugin from "@fullcalendar/timegrid";
import interactionPlugin from "@fullcalendar/interaction";
import "../styles/Calendar.css";
import api from "../api";
```

Imports

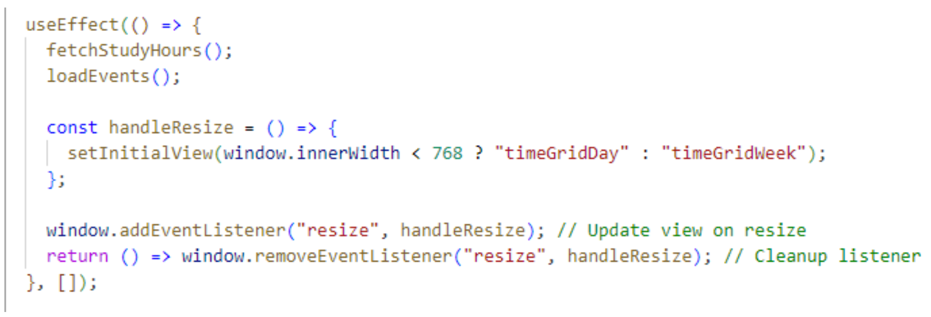
1. **React and Hooks:**
   - **`React`**: Imports the main React library for creating components.
   - **`useState`**: A hook for managing component state.
   - **`useEffect`**: A hook for handling side effects, such as data fetching or updating UI based on state changes.
2. **FullCalendar Plugins:**
   - **`FullCalendar`**: Main calendar component for rendering events.
   - **`dayGridPlugin`**: Plugin for displaying the calendar in a day grid format.
   - **`timeGridPlugin`**: Plugin for displaying the calendar in a time grid format by the hour.
   - **`interactionPlugin`**: Plugin for handling interactions like date clicks.
3. **Styles:**
   - **`../styles/Calendar.css`**: CSS file for styling the calendar.
4. **API:**
   - **`api`**: Imports an API object for server communication (likely containing predefined methods for making HTTP requests).



```
function Calendar() {
  const [events, setEvents] = useState([]);
  const [studyHours, setStudyHours] = useState(0);
  const [remainingHours, setRemainingHours] = useState(0);
  const [showRemainingHours, setShowRemainingHours] = useState(false);
  const [orderHours, setOrderHours] = useState(0);
  const [showOrderForm, setShowOrderForm] = useState(false);
  const [manualVisible, setManualVisible] = useState(false);
  const [hasRejectedEvents, setHasRejectedEvents] = useState(false);
  const [initialView, setInitialView] = useState(window.innerWidth < 768 ? "timeGridDay" : "timeGridWeek");
```

State Management (`useState`)

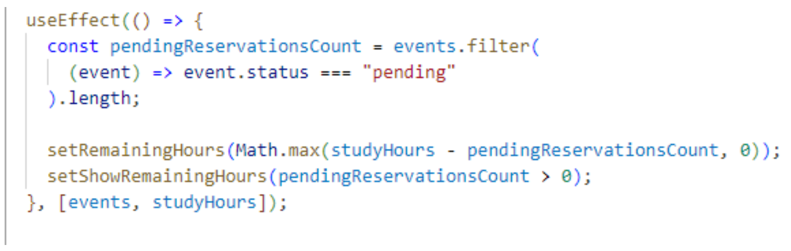
1. **`events`**: Stores an array of events to be displayed in the calendar.
2. **`studyHours`**: Tracks the total number of study hours retrieved from the server.
3. **`remainingHours`**: Stores the number of remaining hours after deducting pending reservations.
4. **`showRemainingHours`**: Boolean to control whether the remaining hours are displayed.
5. **`orderHours`**: Number of hours the user plans to order.
6. **`showOrderForm`**: Boolean to control the visibility of the order form for study hours.
7. **`manualVisible`**: Boolean to control the visibility of the user manual.
8. **`hasRejectedEvents`**: Boolean indicating if the user has rejected reservations.
9. **`initialView`**: Determines the initial calendar view based on screen size:
    - **`"timeGridDay"`** for small screens (< 768px).
    - **`"timeGridWeek"`** for larger screens.



```
useEffect(() => {
  fetchStudyHours();
  loadEvents();

  const handleResize = () => {
    setInitialView(window.innerWidth < 768 ? "timeGridDay" : "timeGridWeek");
  };

  window.addEventListener("resize", handleResize); // Update view on resize
  return () => window.removeEventListener("resize", handleResize); // Cleanup listener
}, []);
```

First `useEffect`

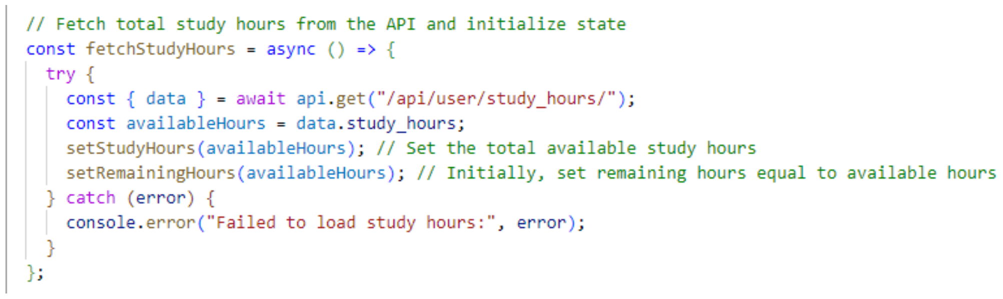
- **Functionality:**
    - Executes once when the component mounts (empty dependency array `[]`).
    - Fetches study hours (`fetchStudyHours`) and calendar events (`loadEvents`).
    - Sets up a resize event listener to adjust the calendar view dynamically based on screen size.
- **`handleResize`:**
    - Adjusts the `initialView` of the calendar based on the screen width:
        - **`"timeGridDay"`** for screen widths less than 768px.
        - **`"timeGridWeek"`** otherwise.
- **Adding and Removing `resize` Listener:**
    - Adds a listener for the `resize` event to invoke `handleResize`.
    - Removes the listener during cleanup to prevent memory leaks.



```
useEffect(() => {
  const pendingReservationsCount = events.filter(
    (event) => event.status === "pending"
  ).length;

  setRemainingHours(Math.max(studyHours - pendingReservationsCount, 0));
  setShowRemainingHours(pendingReservationsCount > 0);
}, [events, studyHours]);
```

Second `useEffect`

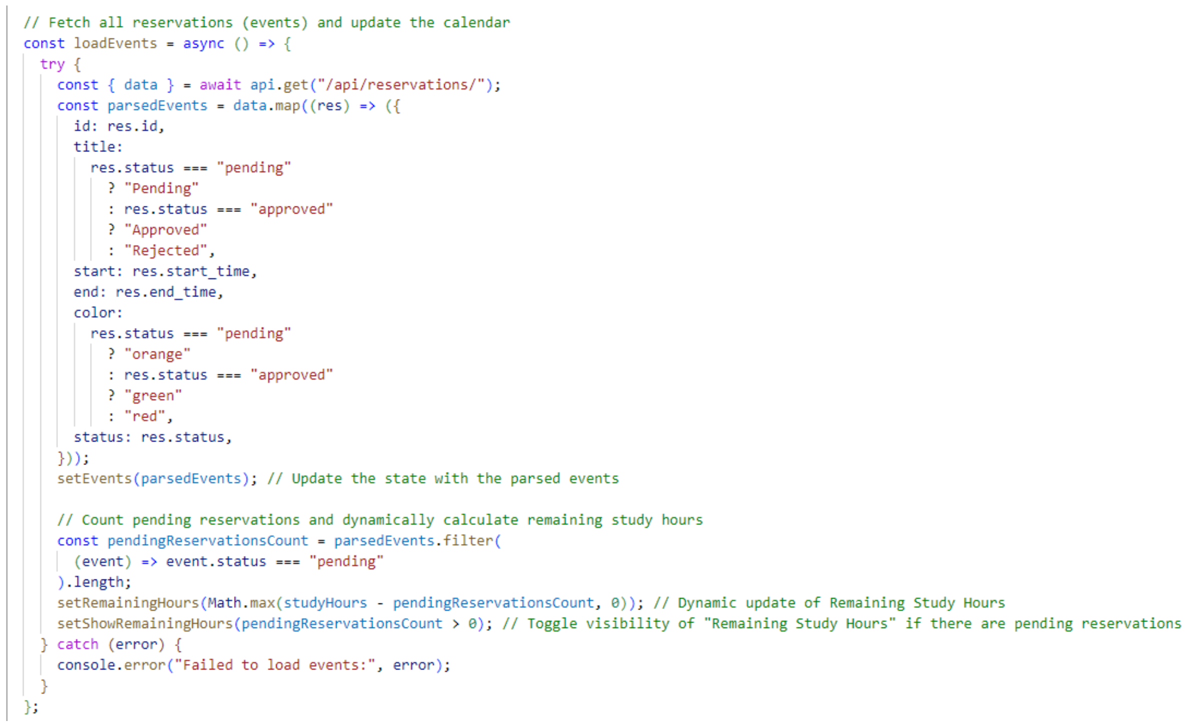
- **Functionality:**
  - Runs whenever `events` or `studyHours` changes.
  - Calculates the number of pending reservations.
  - Dynamically updates:
    - **`remainingHours`**: Remaining hours after accounting for pending reservations.
    - **`showRemainingHours`**: Whether to display remaining hours.
- **Logic:**
  - **`events.filter`**: Filters events with a `status === "pending"`.
  - **`Math.max`**: Ensures the `remainingHours` value does not drop below 0.
  - **`setShowRemainingHours`**: Toggles visibility of remaining hours if there are pending reservations.



```javascript
// Fetch total study hours from the API and initialize state
const fetchStudyHours = async () => {
  try {
    const { data } = await api.get("/api/user/study_hours/");
    const availableHours = data.study_hours;
    setStudyHours(availableHours); // Set the total available study hours
    setRemainingHours(availableHours); // Initially, set remaining hours equal to available hours
  } catch (error) {
    console.error("Failed to load study hours:", error);
  }
};
```

`fetchStudyHours` Function

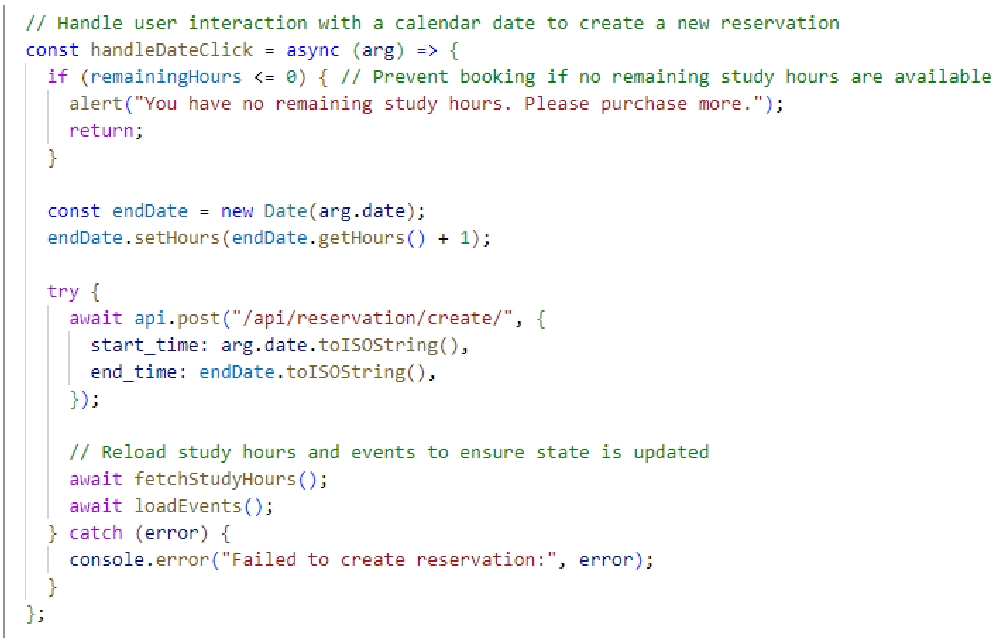
1. **Purpose:**
   - Fetches the total available study hours for the logged-in user from the API.
   - Updates the component state to store the fetched study hours and initialize the remaining hours.
2. **Implementation Details:**
   - **API Call:**
     - Sends a GET request to the `/api/user/study_hours/` endpoint.
     - Extracts the `study_hours` field from the response data.
   - **State Updates:**
     - **`setStudyHours(availableHours)`**: Stores the total study hours in the `studyHours` state.
     - **`setRemainingHours(availableHours)`**: Initializes the `remainingHours` state to match the total available hours.
   - **Error Handling:**
     - Catches any errors during the API request and logs an error message to the console.
3. **Error Logging:**
   - **`console.error`**: Provides details about failures in loading study hours for debugging.



```javascript
// Fetch all reservations (events) and update the calendar
const loadEvents = async () => {
  try {
    const { data } = await api.get("/api/reservations/");
    const parsedEvents = data.map((res) => ({
      id: res.id,
      title:
        res.status === "pending"
          ? "Pending"
          : res.status === "approved"
          ? "Approved"
          : "Rejected",
      start: res.start_time,
      end: res.end_time,
      color:
        res.status === "pending"
          ? "orange"
          : res.status === "approved"
          ? "green"
          : "red",
      status: res.status,
    }));
    setEvents(parsedEvents); // Update the state with the parsed events

    // Count pending reservations and dynamically calculate remaining study hours
    const pendingReservationsCount = parsedEvents.filter(
      (event) => event.status === "pending"
    ).length;
    setRemainingHours(Math.max(studyHours - pendingReservationsCount, 0)); // Dynamic update of Remaining Study Hours
    setShowRemainingHours(pendingReservationsCount > 0); // Toggle visibility of "Remaining Study Hours" if there are pending reservations
  } catch (error) {
    console.error("Failed to load events:", error);
  }
};
```

`loadEvents` Function

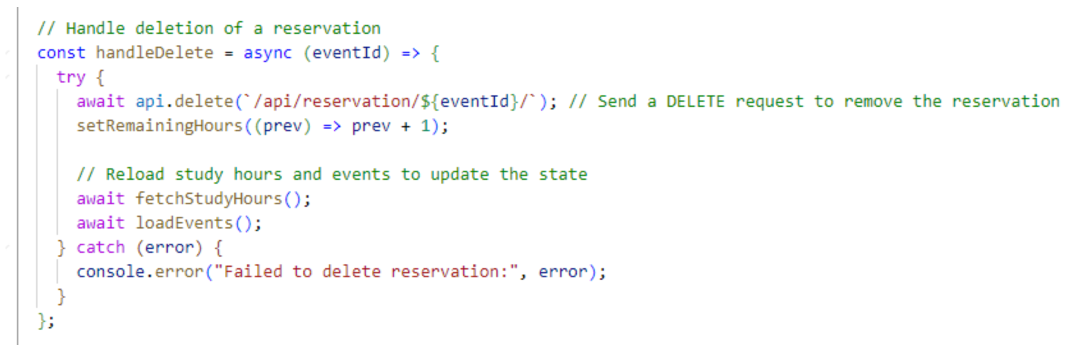
1. **Purpose:**
   - Fetches all reservations (events) for the user from the API.
   - Processes the reservations into a structured format for display in the calendar.
   - Updates the component state with the processed events and calculates the remaining study hours.
2. **Implementation Details:**
   - **API Call:**
     - Sends a GET request to the `/api/reservations/` endpoint.
     - Extracts the list of reservations from the response data.
   - **Mapping Events:**
     - Transforms each reservation into a structured format suitable for the calendar:
       - **`id`**: The unique identifier of the reservation.
       - **`title`**: The label displayed on the calendar:
         - "Pending" for pending reservations.
         - "Approved" for approved reservations.
         - "Rejected" for rejected reservations.
       - **`start`** and **`end`**: Start and end times of the reservation.
       - **`color`**: Color-coded based on reservation status:
         - Orange for pending.
         - Green for approved.
         - Red for rejected.
       - **`status`**: Stores the reservation's status for additional logic.
   - **State Update:**
     - **`setEvents(parsedEvents)`**: Updates the `events` state with the transformed data.
3. **Pending Reservations Count:**
   - **`filter`**: Filters the parsed events to find reservations with `status === "pending"`.
   - **`length`**: Counts the total number of pending reservations.
4. **Remaining Study Hours:**
   - **`Math.max`**: Ensures the `remainingHours` does not drop below 0 when subtracting pending reservations.
   - **`setRemainingHours`**: Updates the `remainingHours` state with the calculated value.
5. **Show Remaining Hours:**
   - **`setShowRemainingHours`**: Toggles visibility of the "Remaining Study Hours" section:
     - Visible only if there are pending reservations.
6. **Error Handling:**
   - Catches and logs any errors during the API request for reservations.



```
// Handle user interaction with a calendar date to create a new reservation
const handleDateClick = async (arg) => {
  if (remainingHours <= 0) { // Prevent booking if no remaining study hours are available
    alert("You have no remaining study hours. Please purchase more.");
    return;
  }

  const endDate = new Date(arg.date);
  endDate.setHours(endDate.getHours() + 1);

  try {
    await api.post("/api/reservation/create/", {
      start_time: arg.date.toISOString(),
      end_time: endDate.toISOString(),
    });

    // Reload study hours and events to ensure state is updated
    await fetchStudyHours();
    await loadEvents();
  } catch (error) {
    console.error("Failed to create reservation:", error);
  }
};
```

`handleDateClick` Function

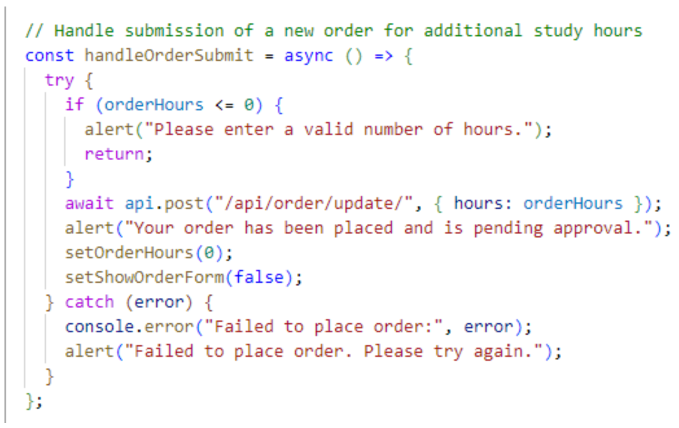
1. **Purpose:**
   - Handles user interaction when clicking on a calendar date to create a new reservation.
2. **Implementation Details:**
   - **Remaining Hours Check:**
     - If the user has no remaining study hours (`remainingHours <= 0`), an alert is shown, and the reservation creation is prevented by returning early.
     - **Alert Message:** "You have no remaining study hours. Please purchase more."
   - **End Date Calculation:**
     - Creates a new date object based on the clicked date (`arg.date`) and sets the end time to 1 hour after the start time.
     - Uses the `setHours` method to increment the hour of the start date by 1.
   - **API Call:**
     - Sends a POST request to `/api/reservation/create/` with the start and end times of the reservation in ISO string format.
   - **State Updates:**
     - Calls `fetchStudyHours` and `loadEvents` to reload the study hours and events, ensuring the state is updated after creating the reservation.
   - **Error Handling:**
     - Catches and logs any errors during the reservation creation process to the console.



```
// Handle deletion of a reservation
const handleDelete = async (eventId) => {
  try {
    await api.delete(`/api/reservation/${eventId}/`); // Send a DELETE request to remove the reservation
    setRemainingHours((prev) => prev + 1);

    // Reload study hours and events to update the state
    await fetchStudyHours();
    await loadEvents();
  } catch (error) {
    console.error("Failed to delete reservation:", error);
  }
};
```

`handleDelete` Function

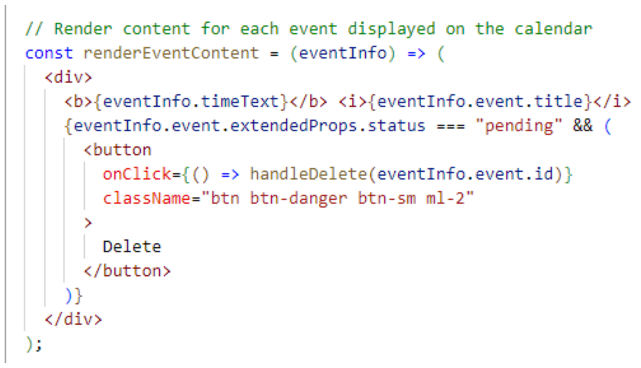
1. **Purpose:**
   - Handles the deletion of a specific reservation when requested by the user.
2. **Implementation Details:**
   - **API Call:**
     - Sends a DELETE request to `/api/reservation/{eventId}/`, where `{eventId}` is the unique identifier of the reservation to be deleted.
   - **State Updates:**
     - **Increment Remaining Hours:**
       - Increments the `remainingHours` state by 1 using the `setRemainingHours` method and a functional state update (`prev => prev + 1`).
     - Calls `fetchStudyHours` and `loadEvents` to reload the study hours and events, ensuring the state reflects the updated reservations.
   - **Error Handling:**
     - Catches and logs any errors during the deletion process to the console.



```
// Handle submission of a new order for additional study hours
const handleOrderSubmit = async () => {
  try {
    if (orderHours <= 0) {
      alert("Please enter a valid number of hours.");
      return;
    }
    await api.post("/api/order/update/", { hours: orderHours });
    alert("Your order has been placed and is pending approval.");
    setOrderHours(0);
    setShowOrderForm(false);
  } catch (error) {
    console.error("Failed to place order:", error);
    alert("Failed to place order. Please try again.");
  }
};
```

`handleOrderSubmit` Function

1. **Purpose:**
   - Handles the submission of a new order for additional study hours.
2. **Implementation Details:**
   - **Input Validation:**
     - Checks if the user has entered a valid number of hours (`orderHours > 0`).
     - If the input is invalid, an alert is displayed with the message: "Please enter a valid number of hours."
   - **API Call:**
     - Sends a POST request to `/api/order/update/` with the number of hours as the payload.
     - Displays an alert indicating the order has been successfully placed and is pending approval.
   - **State Updates:**
     - Resets the `orderHours` state to `0`.
     - Hides the order form by setting `showOrderForm` to `false`.
   - **Error Handling:**
     - Logs errors in the console and displays an alert if the API call fails.

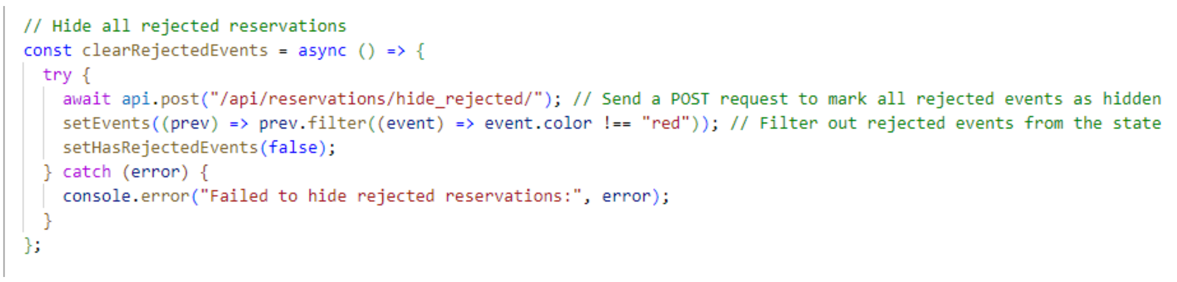


```
// Render content for each event displayed on the calendar
const renderEventContent = (eventInfo) => (
  <div>
    <b>{eventInfo.timeText}</b> <i>{eventInfo.event.title}</i>
    {eventInfo.event.extendedProps.status === "pending" && (
      <button
        onClick={() => handleDelete(eventInfo.event.id)}
        className="btn btn-danger btn-sm ml-2"
      >
        Delete
      </button>
    )}
  </div>
);
```

`renderEventContent` Function

1. **Purpose:**
   - Customizes the content rendered for each event on the calendar.
2. **Implementation Details:**
   - **Event Details Display:**

        - Displays the event's time (`eventInfo.timeText`) and title (`eventInfo.event.title`) in bold and italic formatting, respectively.
    - **Delete Button:**
        - If the event has a status of `"pending"`, a delete button is rendered alongside the event.
        - The button triggers the `handleDelete` function with the event's ID when clicked.
    - **Styling:**
        - Applies CSS classes (`btn btn-danger btn-sm ml-2`) to style the delete button.



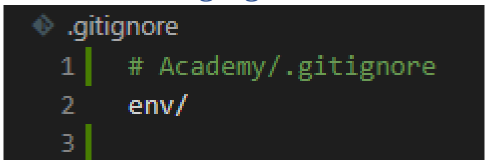
```
// Hide all rejected reservations
const clearRejectedEvents = async () => {
  try {
    await api.post("/api/reservations/hide_rejected/"); // Send a POST request to mark all rejected events as hidden
    setEvents((prev) => prev.filter((event) => event.color !== "red")); // Filter out rejected events from the state
    setHasRejectedEvents(false);
  } catch (error) {
    console.error("Failed to hide rejected reservations:", error);
  }
};
```

`clearRejectedEvents` Function

1. **Purpose:**
    - Clears all rejected reservations from the calendar.
2. **Implementation Details:**
    - **API Call:**
        - Sends a POST request to `/api/reservations/hide_rejected/` to mark all rejected events as hidden.
    - **State Updates:**
        - Filters out events with the color `"red"` from the `events` state, effectively removing rejected events.
        - Resets the `hasRejectedEvents` state to `false`.
    - **Error Handling:**
        - Logs errors in the console if the API call fails.

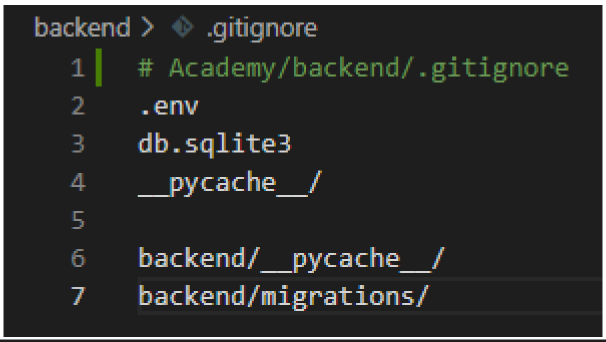
4 GitHub

4.1 Create .gitignore



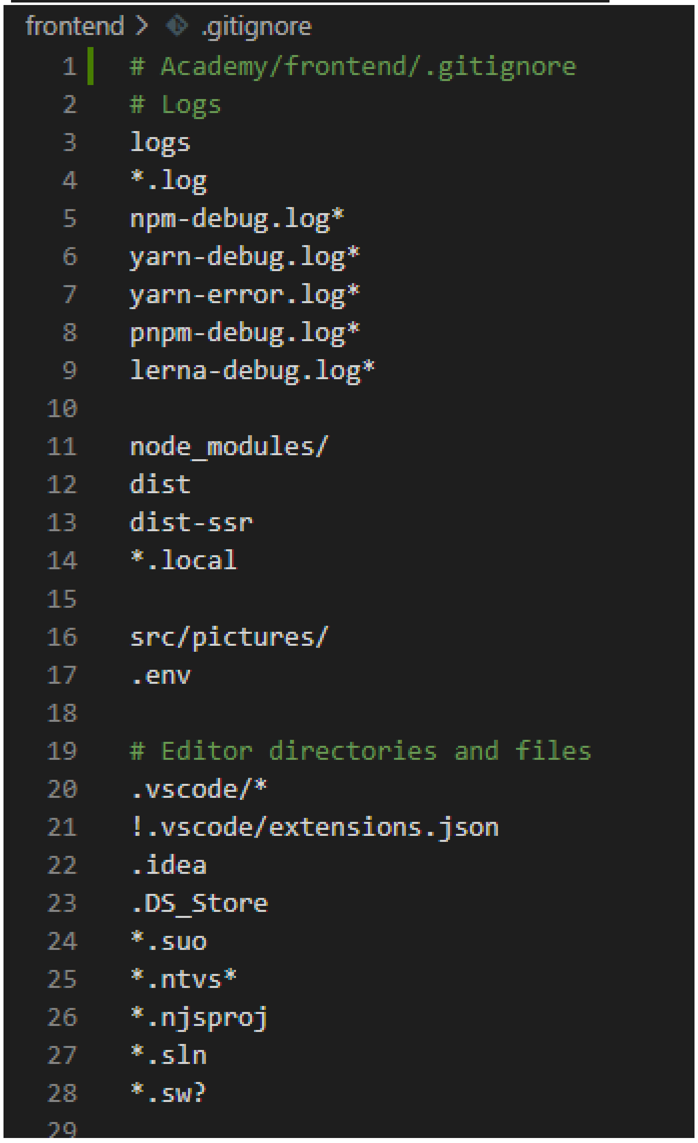
.gitignore

```
# Academy/.gitignore
env/

```



backend > .gitignore

```
# Academy/backend/.gitignore
.env
db.sqlite3
__pycache__/

backend/__pycache__/
backend/migrations/
```

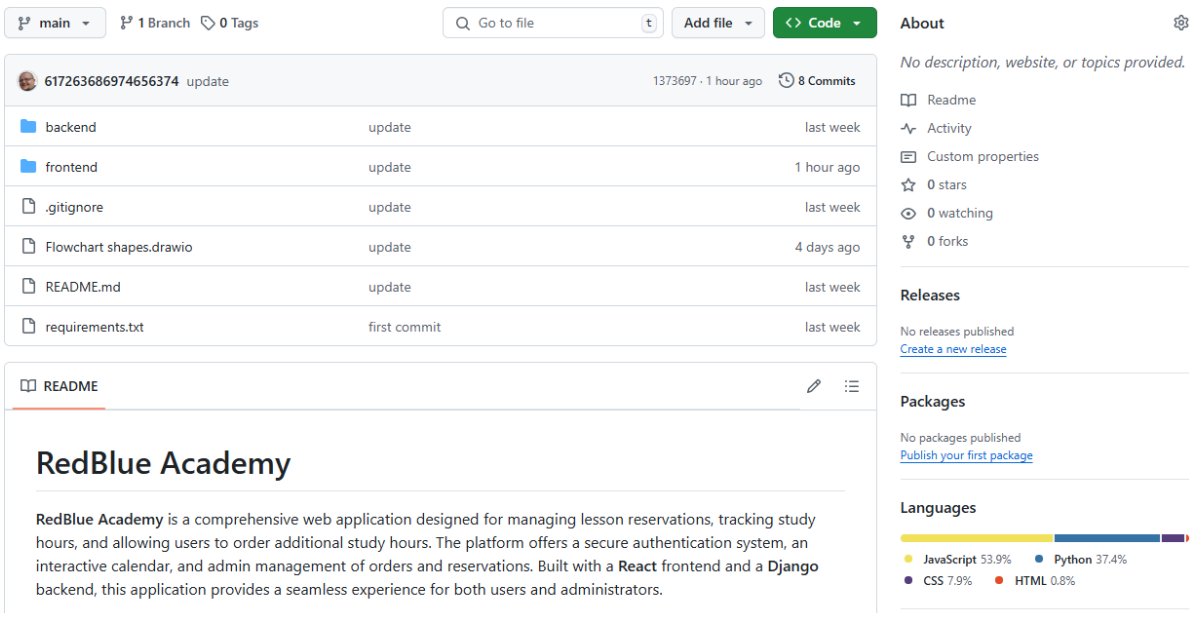


frontend > .gitignore

```
# Academy/frontend/.gitignore
# Logs
logs
*.log
npm-debug.log*
yarn-debug.log*
yarn-error.log*
pnpm-debug.log*
lerna-debug.log*

node_modules/
dist
dist-ssr
*.local

src/pictures/
.env

# Editor directories and files
.vscode/*
!.vscode/extensions.json
.idea
.DS_Store
*.suo
*.ntvs*
*.njsproj
*.sln
*.sw?

```

4.2 Create repository

```
git init
git add .
git commit -m "first commit"
git branch -M main
git remote add origin https://github.com/SZF-KRC/RedBlue.git
git push -u origin main
```



617263686974656374 update 1373697 · 1 hour ago 8 Commits

backend	update	last week
frontend	update	1 hour ago
.gitignore	update	last week
Flowchart shapes.drawio	update	4 days ago
README.md	update	last week
requirements.txt	first commit	last week

README

RedBlue Academy

RedBlue Academy is a comprehensive web application designed for managing lesson reservations, tracking study hours, and allowing users to order additional study hours. The platform offers a secure authentication system, an interactive calendar, and admin management of orders and reservations. Built with a **React** frontend and a **Django** backend, this application provides a seamless experience for both users and administrators.

About

No description, website, or topics provided.

Readme
Activity
Custom properties
0 stars
0 watching
0 forks

Releases

No releases published
Create a new release

Packages

No packages published
Publish your first package

Languages

● JavaScript 53.9% ● Python 37.4%
● CSS 7.9% ● HTML 0.8%