Projektarbeit –

RedBlue Academy

**Thema: Web Entwicklung / KMS 2**

**Jaroslav Krc**

**Dauer der Projektarbeit: 14.10.2024 - 22.11.2024**

Trainer\*in: Sabrina Müller

**Content**

[1 Introduction 4](#_Toc182311774)

[1.1 Create .env and install requirements 7](#_Toc182311775)

[2 Backend 9](#_Toc182311776)

[2.1 Create Django project 9](#_Toc182311777)

[2.2 Create startapp 9](#_Toc182311778)

[2.3 Update settings.py 9](#_Toc182311779)

[2.4 Create serializer 12](#_Toc182311780)

[2.5 Create User view 13](#_Toc182311781)

[2.6 Create Urls 14](#_Toc182311782)

[2.7 Start Server 15](#_Toc182311783)

[2.8 Create MySQL database 16](#_Toc182311784)

[2.9 Create models Student hour, Reservation, Order and Control 18](#_Toc182311785)

[2.1 Create two more serializers 22](#_Toc182311786)

[2.10 Create another views 24](#_Toc182311787)

[2.11 Create urls in api 32](#_Toc182311788)

[2.13 Create superuser admin account 33](#_Toc182311789)

[2.14 Login to Django administration 33](#_Toc182311790)

[2.15 Create admin 34](#_Toc182311791)

[**3** Frontend 39](#_Toc182311792)

[3.1 Create Frontend 39](#_Toc182311793)

[3.2 Install packages 39](#_Toc182311794)

[3.3 Delete css 40](#_Toc182311795)

[3.4 Create new folders 41](#_Toc182311796)

[3.5 Create new files 42](#_Toc182311797)

[3.6 Create API 43](#_Toc182311798)

[3.7 Create Authentication 45](#_Toc182311799)

[3.8 Create Protected Route 55](#_Toc182311800)

[3.9 Create page files 57](#_Toc182311801)

[3.10 Rewrite App 58](#_Toc182311802)

[3.11 Create Component (Footer.jsx) 59](#_Toc182311803)

[3.11.1 Overview 60](#_Toc182311804)

[3.11.2 Code Breakdown 60](#_Toc182311805)

[3.12 Create Component (Layout.jsx) 62](#_Toc182311806)

[3.13 Update App with Layout 63](#_Toc182311807)

[3.14 Create first css 64](#_Toc182311808)

[3.15 Create page Home with style 65](#_Toc182311809)

[3.16 Create page Privacy Policy with style 69](#_Toc182311810)

[3.17 Create page Terms of Service with style 70](#_Toc182311811)

[3.18 Create page FAQ with style 71](#_Toc182311812)

[3.19 Create page Not Found 73](#_Toc182311813)

[3.20 Create page Price with styled 74](#_Toc182311814)

[3.21 Update index.html 76](#_Toc182311815)

[3.22 Create page Services with style 76](#_Toc182311816)

[3.23 Create component Header with style 78](#_Toc182311817)

[3.24 Update Layout and write there Header 82](#_Toc182311818)

[3.25 Create page Calendar 82](#_Toc182311819)

[3.26 Update App with new connection 83](#_Toc182311820)

[3.27 Run servers and check pages 84](#_Toc182311821)

[3.28 Create Component Form.jsx with style 85](#_Toc182311822)

[3.29 Update Login and Register Page 89](#_Toc182311823)

[3.30 Start backend and frontend server server and make test 90](#_Toc182311824)

[3.31 Open Django Administrator 91](#_Toc182311825)

[3.32 Install packeges for calendar 91](#_Toc182311826)

[3.33 Create OrderPage 92](#_Toc182311827)

[3.34 Create OrderPending 94](#_Toc182311828)

[3.35 Update Calendar 95](#_Toc182311829)

[4 GitHub 104](#_Toc182311830)

[4.1 Create .gitignore 104](#_Toc182311831)

[4.2 Create repository 105](#_Toc182311832)

# 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**

* Features
* Folder Structure
* Environment Variables
* Usage
* API Endpoints
* Technologies Used

**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.

# 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.



**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.

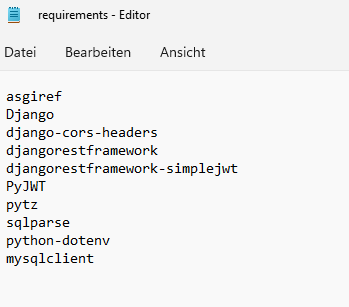
**PyJWT**  
PyJWT is a tool for working with JSON Web Tokens in Python. It is used for encoding and decoding JWT tokens and is a dependency of djangorestframework-simplejwt. This library provides secure token management for user authentication and authorization.

**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.



# Backend

# 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.



# 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).

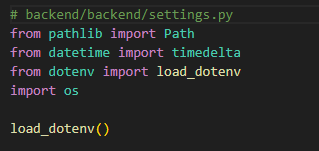


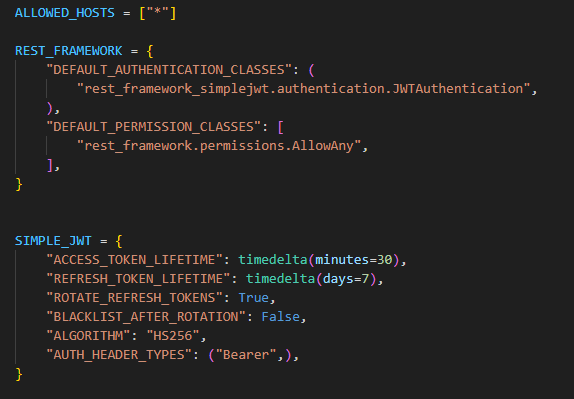
# Update settings.py

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.



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.

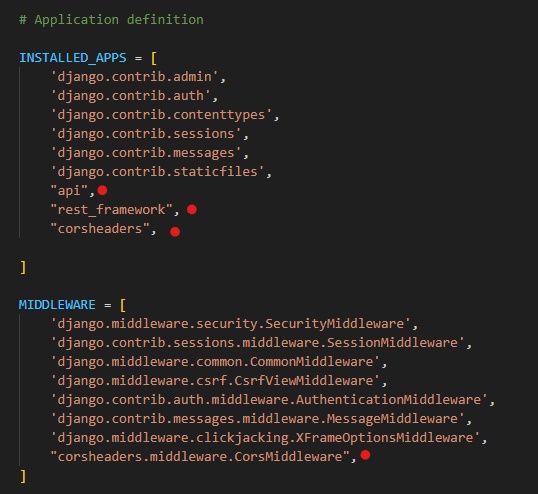
**REFRESH\_TOKEN\_LIFETIME**: Defines the lifespan of the refresh token, which is set to expire after 7 days. The refresh token allows the user to obtain a new access token without logging in again.

**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.





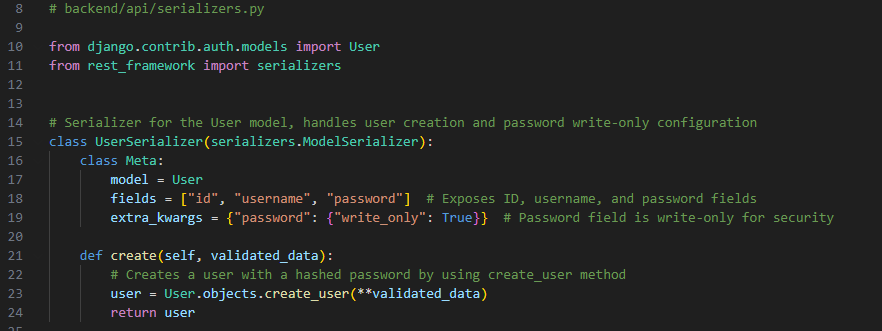
**CORS\_ALLOW\_ALL\_ORIGINS**: When set to True, this setting allows requests from any origin (domain). This means the server will permit access from all domains attempting to communicate with the backend. In a production environment, it’s safer to restrict access to specific domains to reduce security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

# 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.



**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**:

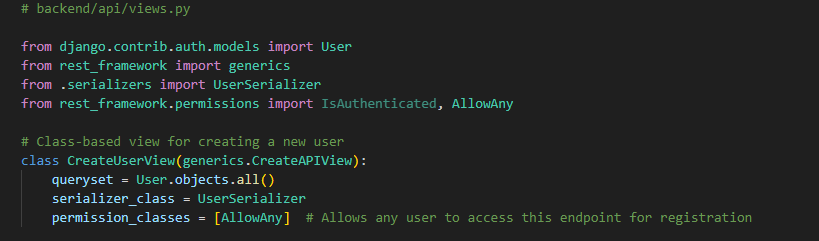
* **model**: Specifies that this serializer is for the User model.
* **fields**: Specifies the fields to include in the serialized data (id, username, and password).
* **extra\_kwargs**: Sets additional options for the fields. Here, password is marked as "write\_only": True, meaning it will only be used 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

# 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.



**Imports**:

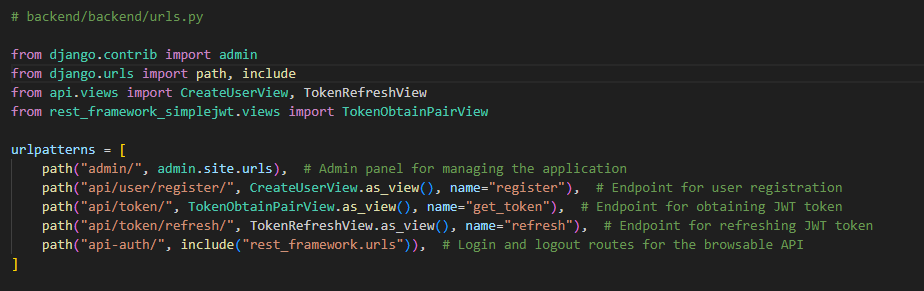
* **User**: The User model from Django's authentication system, representing users in the database.
* **generics**: Provides generic views from Django REST Framework, which simplify the process of creating common views like Create, Retrieve, Update, and Delete.
* **UserSerializer**: The custom serializer 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.

# 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.



**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.

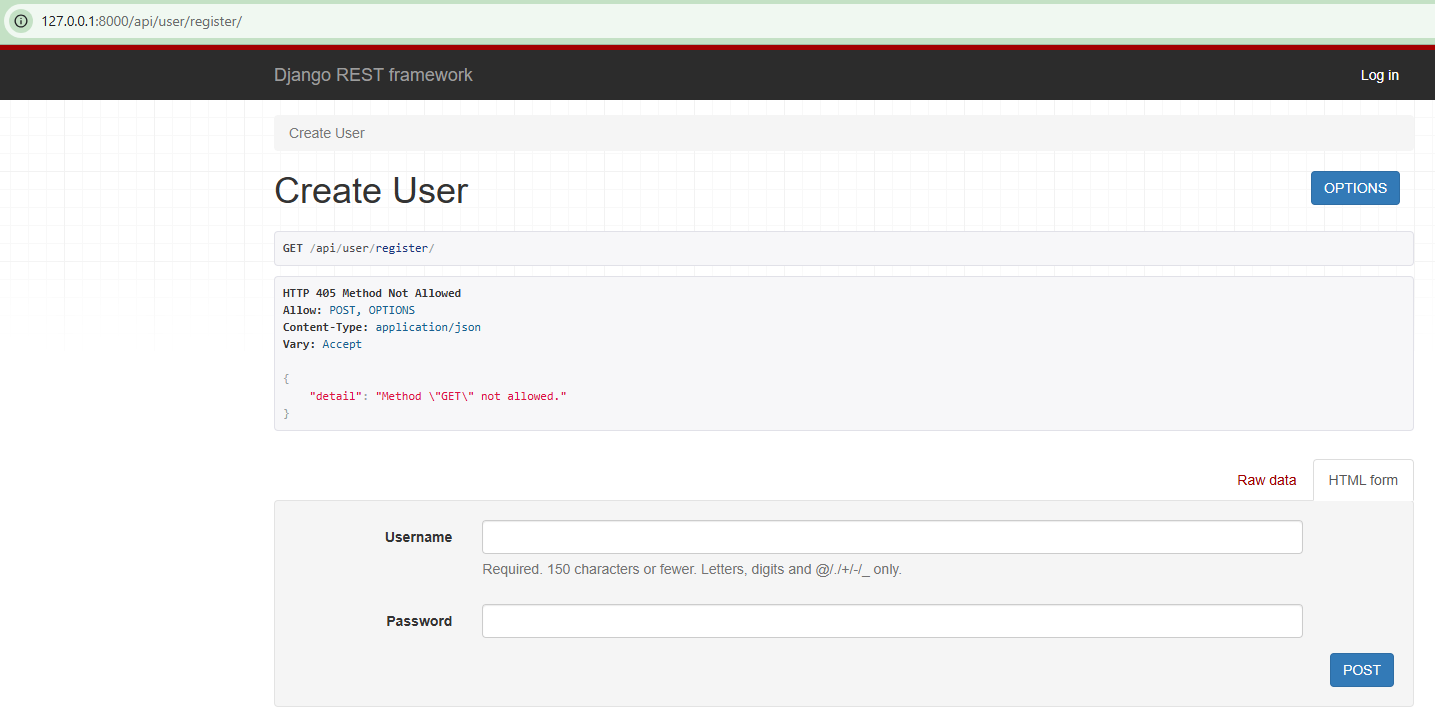
This command generates new migration files based on changes detected in Django models.

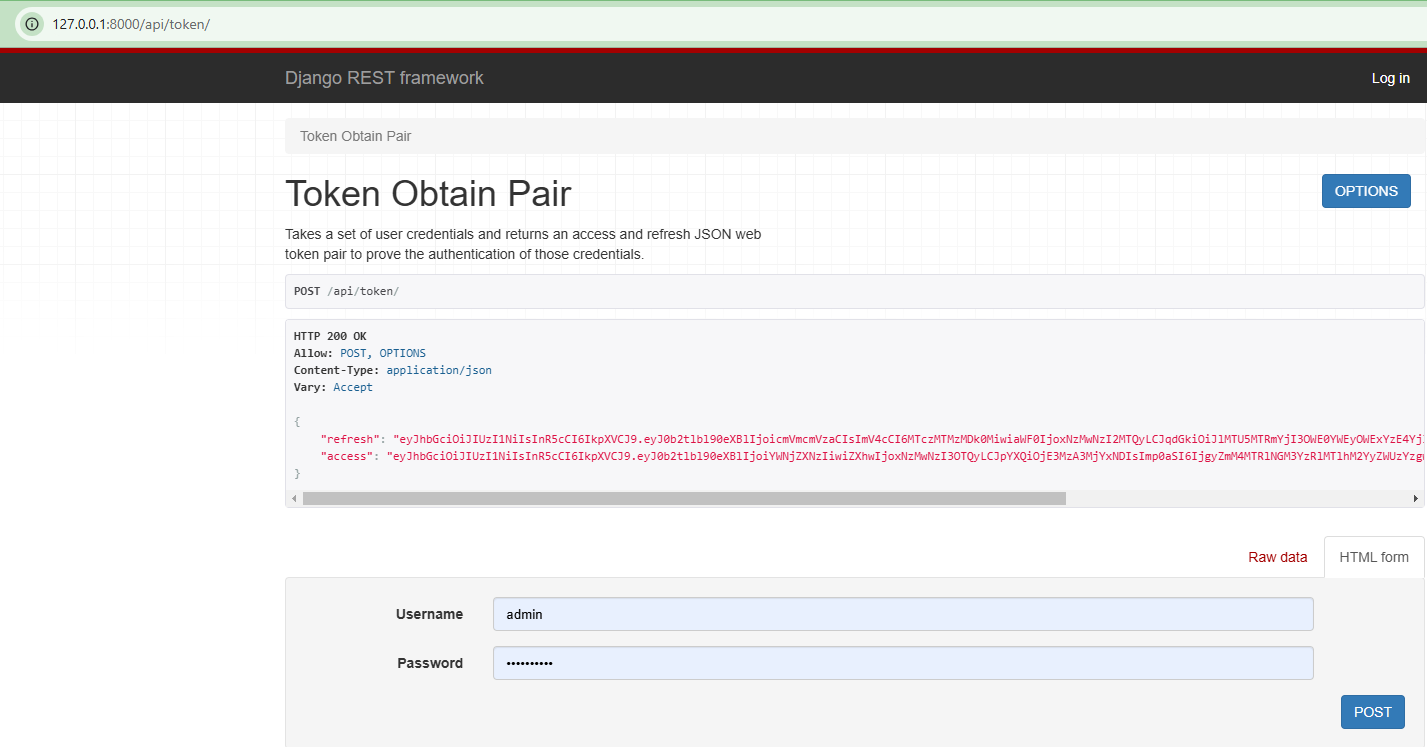
 This command applies all available migrations to the database. It ensures that the database structure is synchronized with the current model definitions in the project.

# Start Server

This command starts Django’s built-in development server

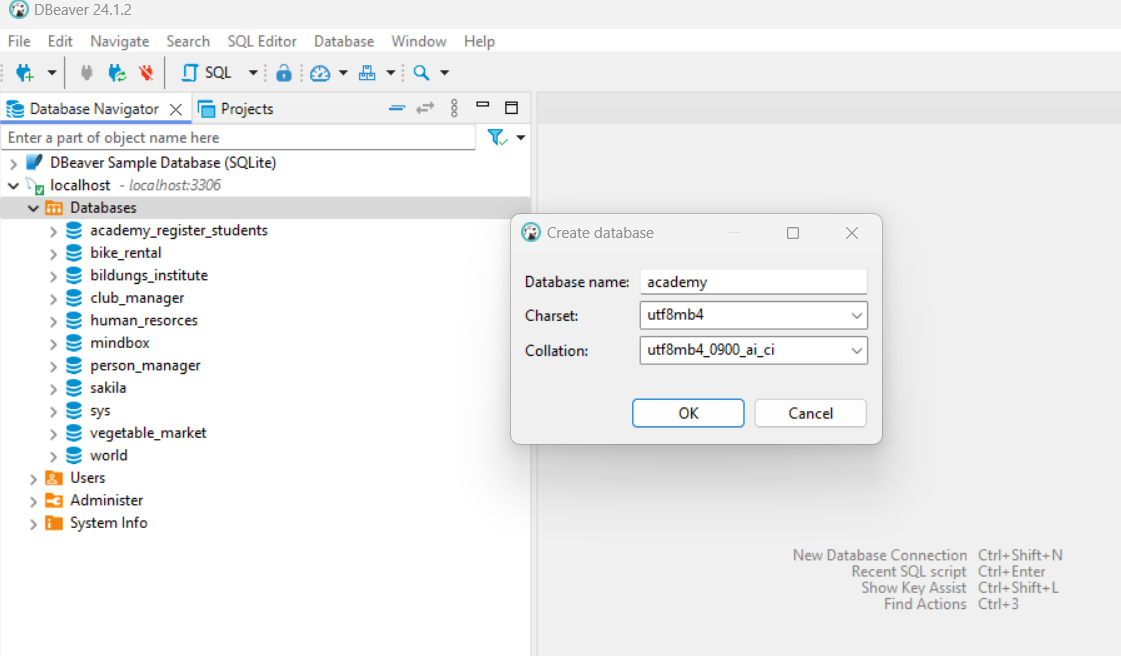
Create first user and check if tokens works

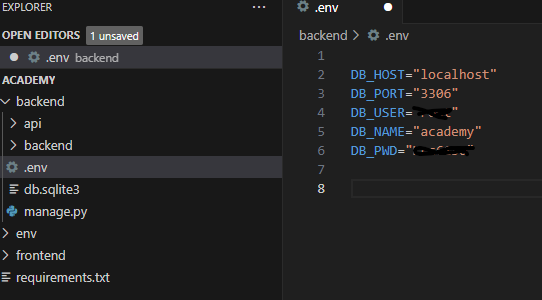


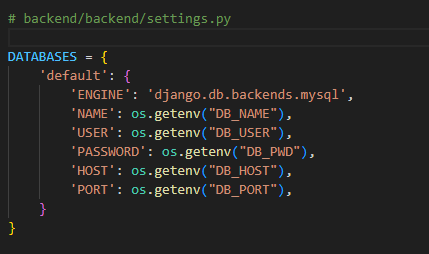


# Create MySQL database

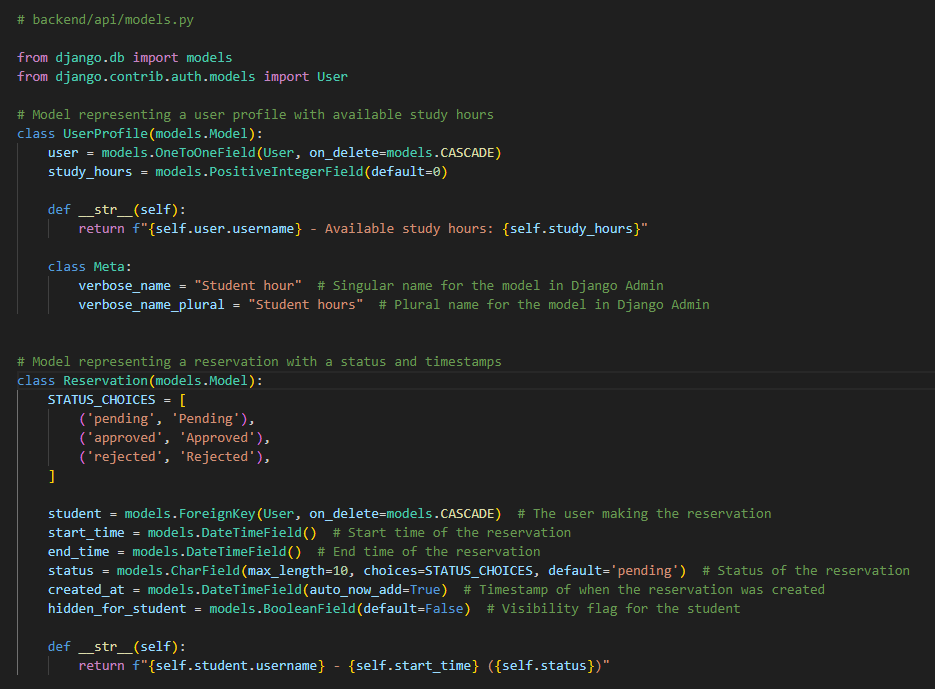
Using DBeaver and create a new database.



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

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

# Create models Student hour, Reservation, Order and Control



**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

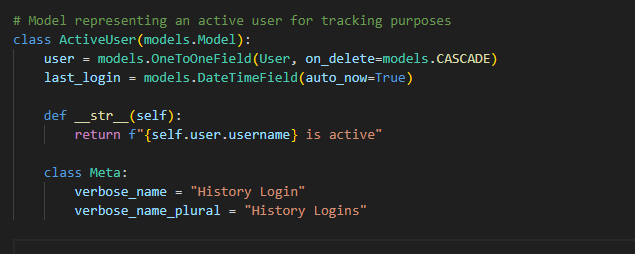
* This model represents a reservation with a status and timestamps.

**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.



**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**:

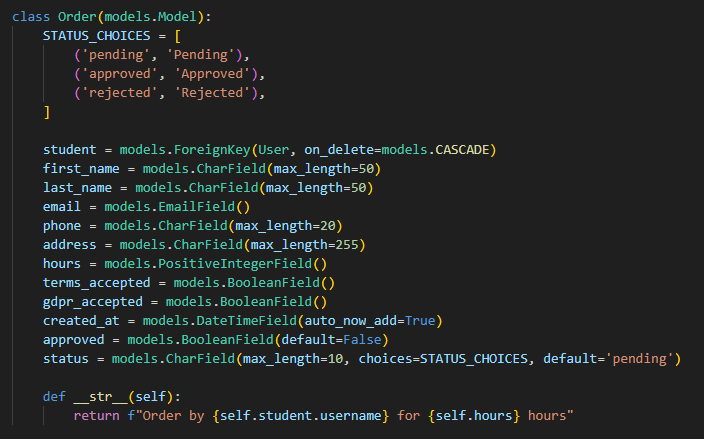
\_\_str\_\_: Returns a string indicating that the user is active, showing the username.

**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.



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.

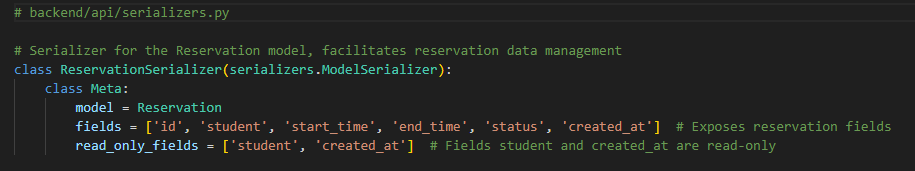
Methods:

* **\_\_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.

## 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.



**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.

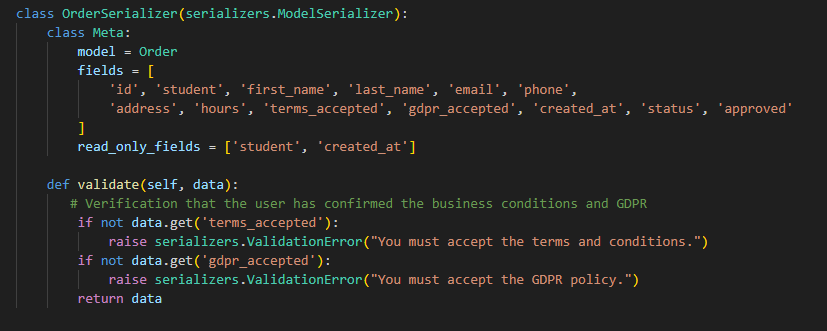
**fields**: Defines the fields from the Reservation model that will be included in the serialized data. These fields are:

* 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.



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.

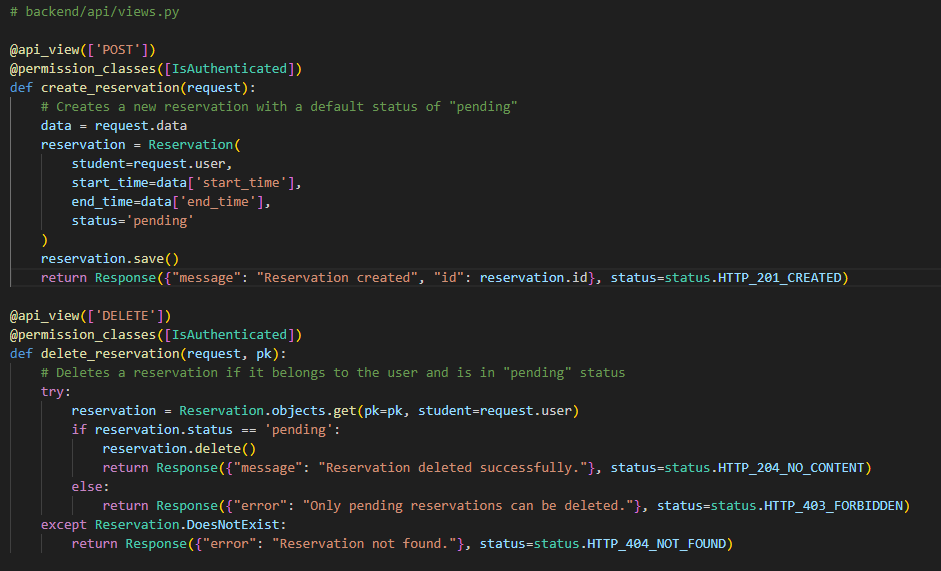
Custom Validation Method:

* **validate(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.

# 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.



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.

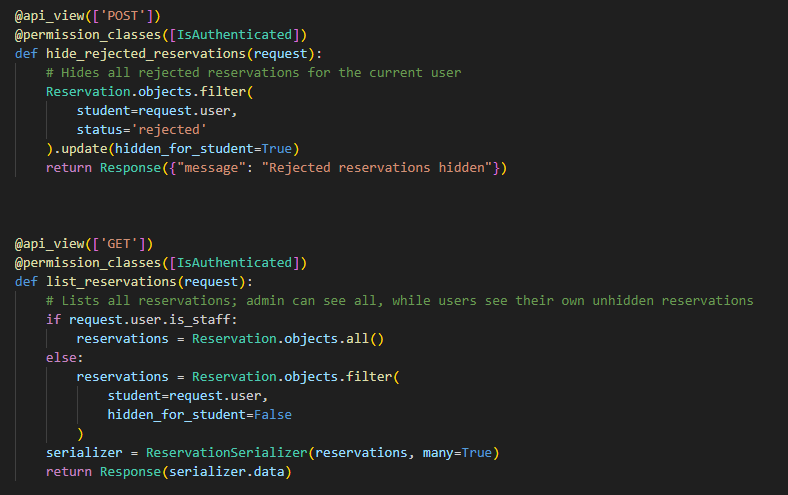
**Steps**:

* **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.



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**:

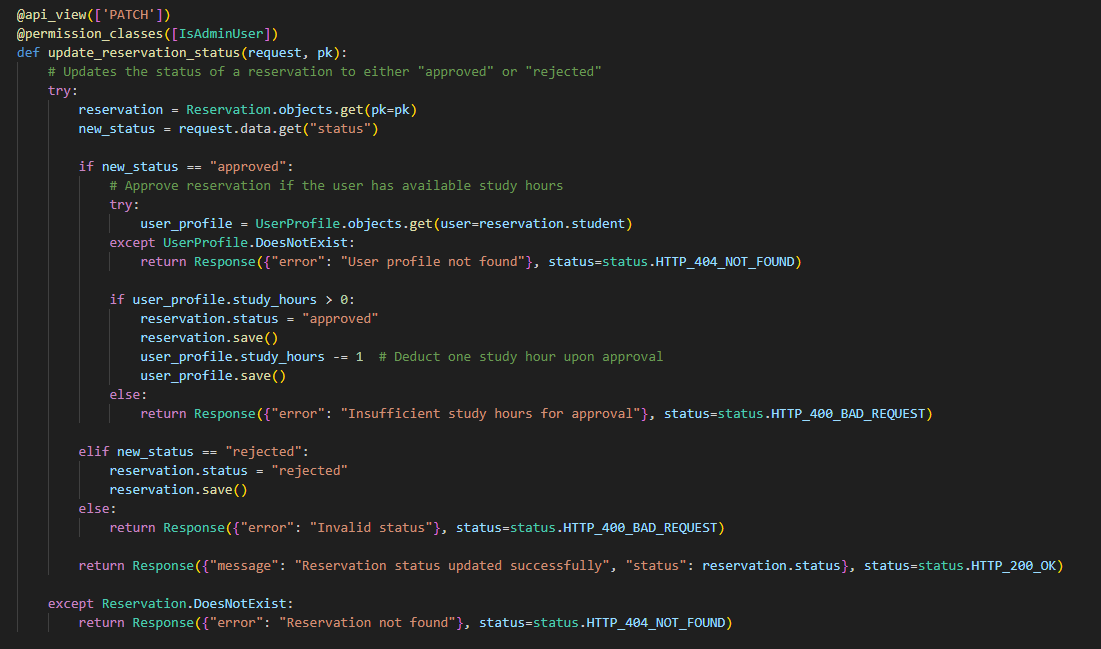
list\_reservations: This function lists reservations with different access levels for admin and regular users.

**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.



**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**:

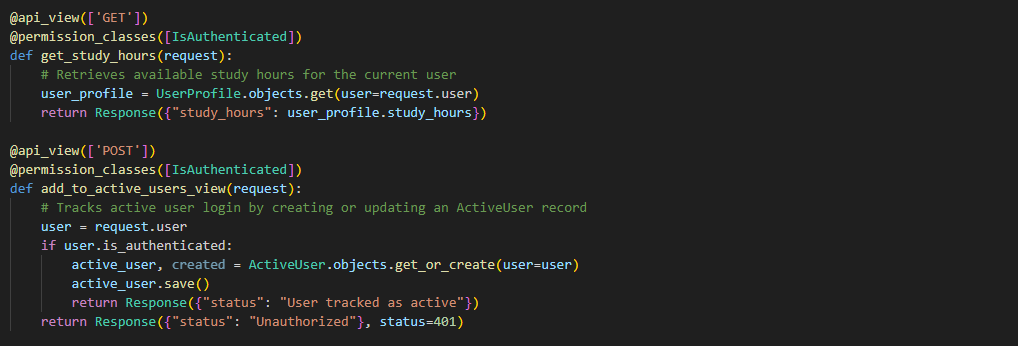
update\_reservation\_status: This function updates the status of a reservation to either "approved" or "rejected" based on the provided input.

**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.



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.

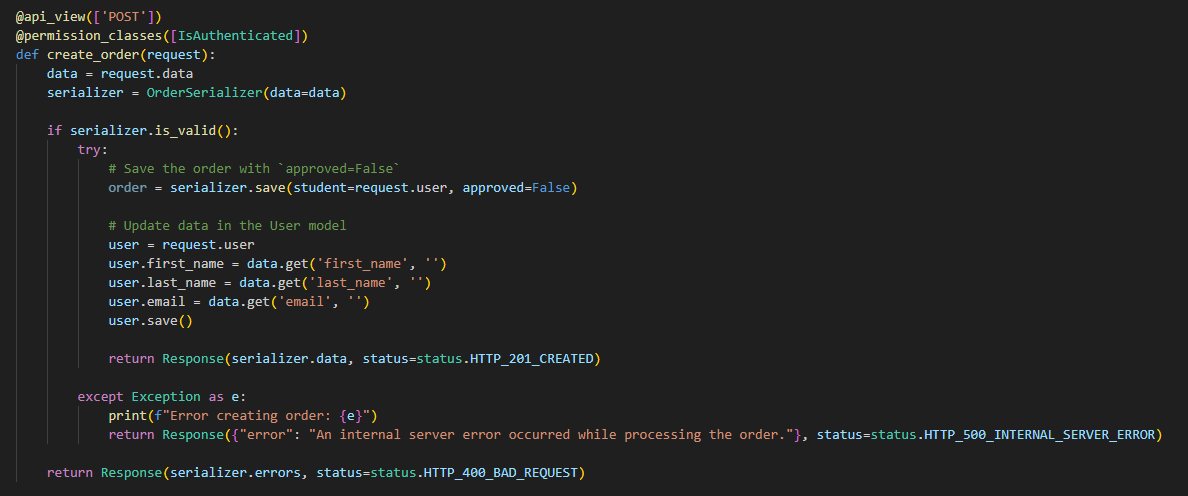
**Function**:

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.

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.

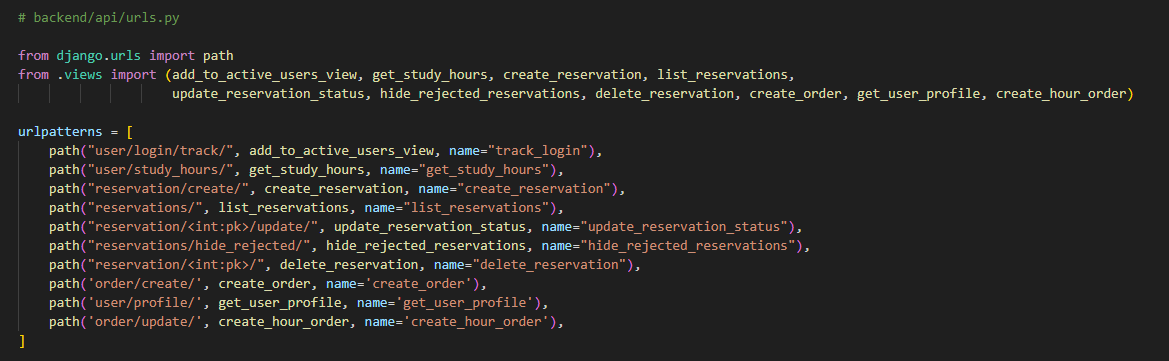
Function:

* **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.

# 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  **User Tracking and Profile**:

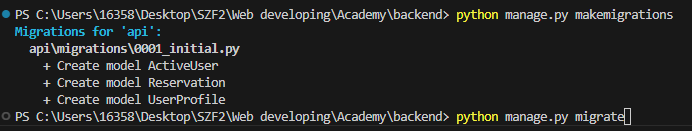
* **user/login/track/**:
  + Tracks the login activity of the authenticated user by creating or updating an ActiveUser record.
* **user/study\_hours/**:
  + Retrieves the available study hours for the authenticated user.
* **user/profile/**:
  + Retrieves the profile details of the authenticated user.

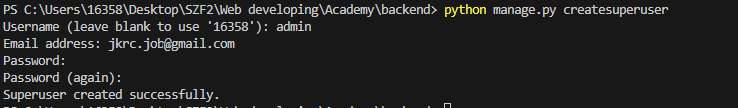
 **Reservations Management**:

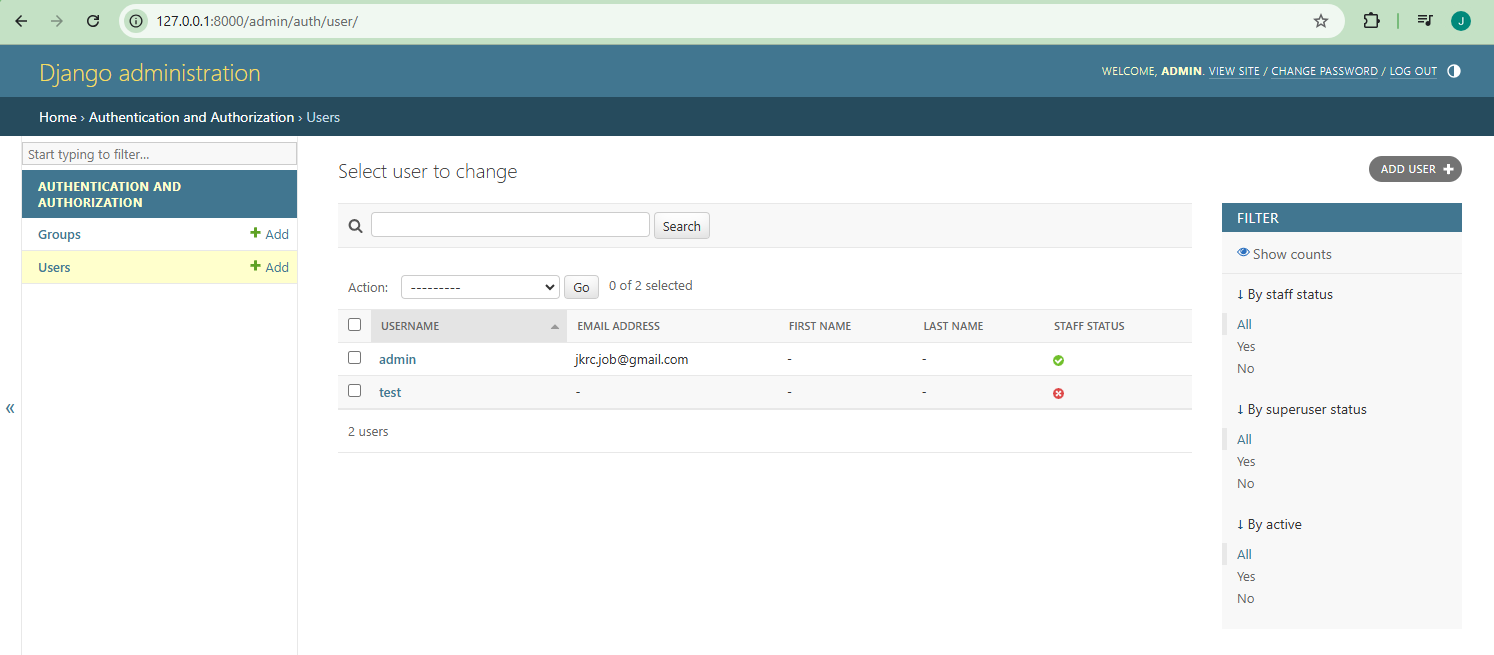
* **reservation/create/**:
  + Allows the authenticated user to create a new reservation.
* **reservations/**:
  + Lists all reservations. Admin users can see all reservations, while regular users see only their own unhidden reservations.
* **reservation/<int:pk>/update/**:
  + Updates the status of 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 

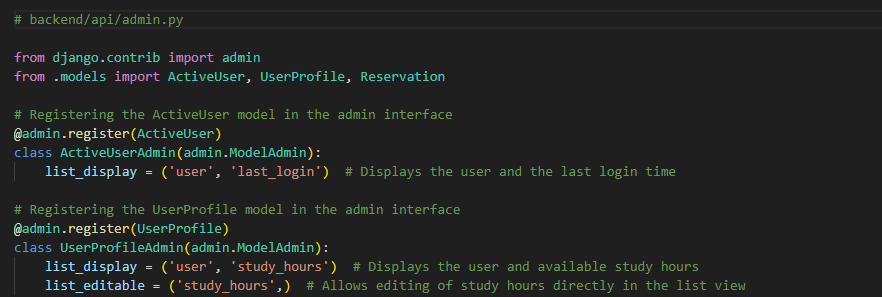
2.13 Create superuser admin account 

2.14 Login to Django administration 

# 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.



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**:

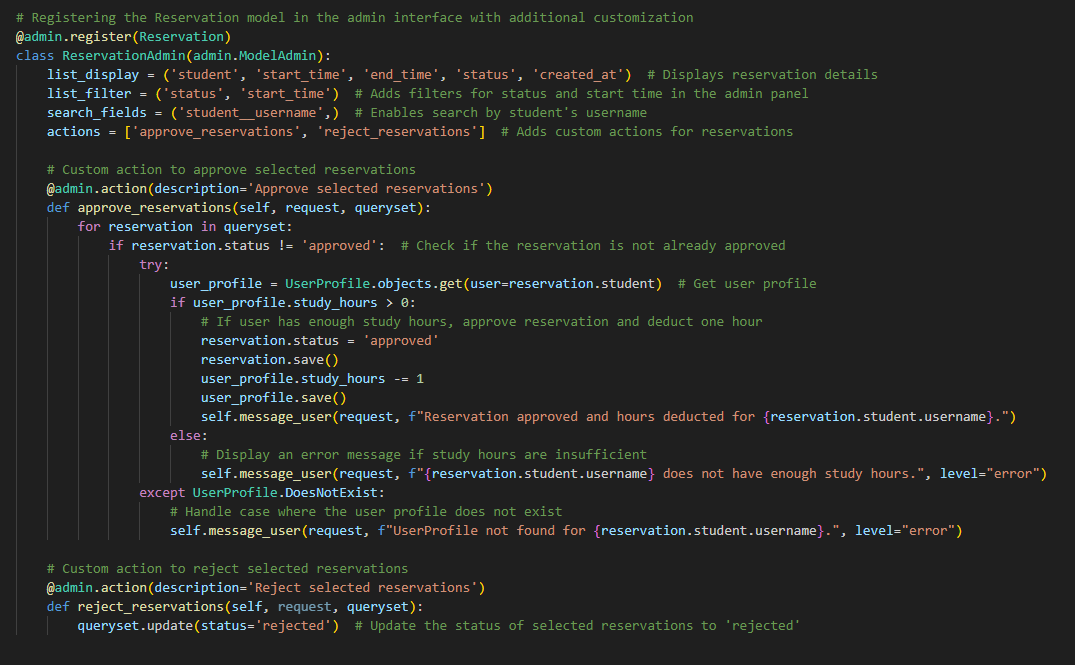
**class UserProfileAdmin(admin.ModelAdmin)**: Defines an admin class to customize how UserProfile is displayed in the admin.

**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.



**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.

****

 **Registering the Order Model in the Admin Interface**:

* **@admin.register(Order)**: Registers the Order model with the admin panel.
* **class OrderAdmin(admin.ModelAdmin)**: Defines a custom admin class for managing the Order model within the admin interface.

 **Admin Display Configuration**:

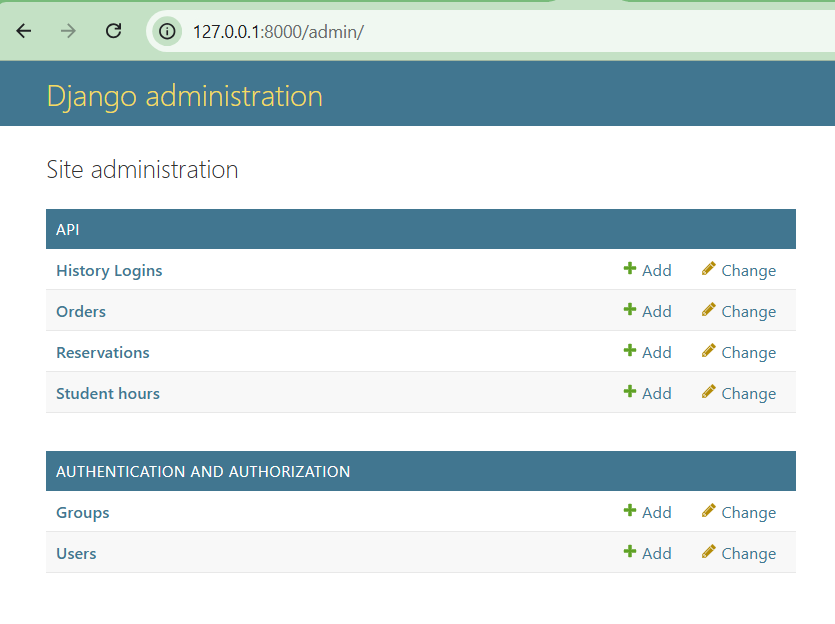
* **list\_display**: Specifies fields to be displayed in the list view of orders in the admin interface (student, first name, last name, email, hours, status, creation date).
* **list\_filter**: Adds filters for created\_at and status to allow quick filtering by these fields.
* **search\_fields**: Enables search functionality by student username, first name, last name, and email.
* **actions**: Defines custom actions available for orders: approve\_orders (approve selected orders) and reject\_orders (reject selected orders).

 **Approving Orders**:

* **@admin.action(description='Approve selected orders')**: This decorator provides a description for the custom admin action.
* **def approve\_orders(self, request, queryset):**: This function iterates over the selected orders in the queryset.
* **Updating Order Status**: If an order’s status is pending, it is changed to approved, and the order is saved.
* **Updating Study Hours in UserProfile**:
  + Retrieves or creates a UserProfile for the student associated with the order.
  + Adds the number of hours from the order to the student’s study\_hours.
  + Sets order\_completed to True, indicating that the order is completed.
  + Saves the updated profile.
* **Admin Message**: After approving an order, a confirmation message is displayed, showing that the order for the specific student has been approved and hours have been added.

 **Rejecting Orders**:

* **@admin.action(description='Reject selected orders')**: This decorator provides a description for the rejection action.
* **def reject\_orders(self, request, queryset):**: This function iterates over the selected orders in the queryset and sets the status to rejected for each.
* **Admin Message**: After rejecting orders, a message displays the number of orders that have been rejected.

Check administration if is all done 

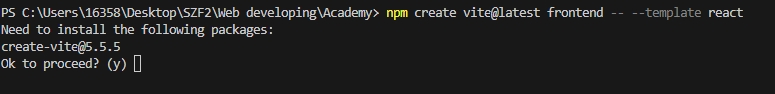
# **Frontend**

## **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.



## Install packages

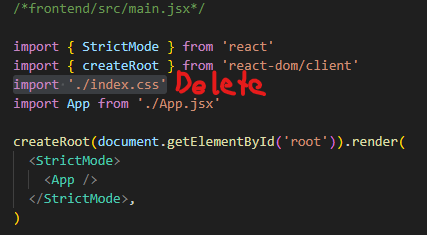
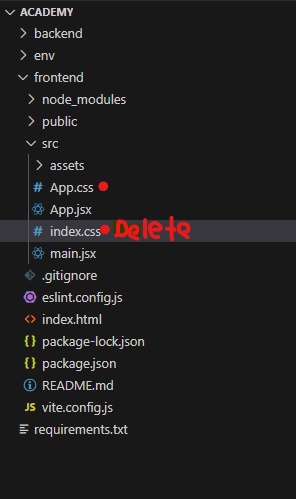
**cd .\frontend\**: Navigates into the frontend folder.

**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.



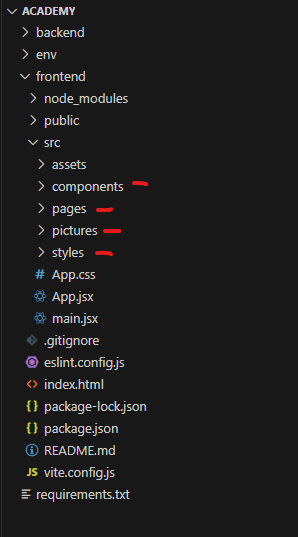
## Delete css

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



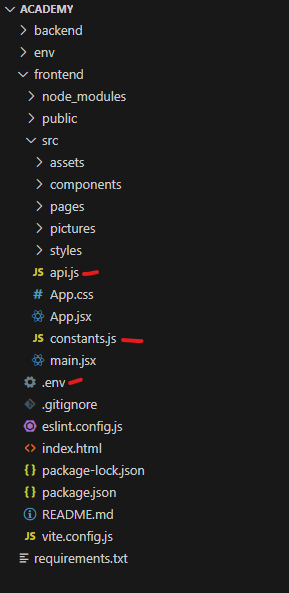
## Create new folders

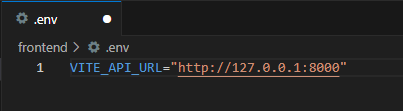
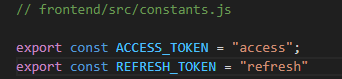
Create new folders in src.



## Create new files

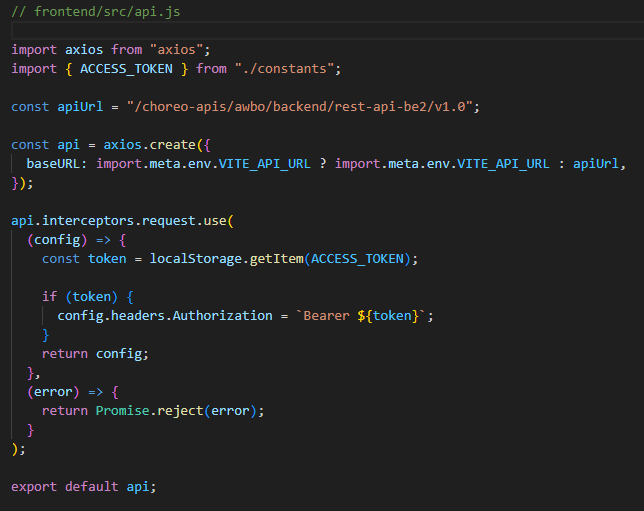
Create new files: api.js, constants.js, and .env, and in the constants.js file, add constants for storing and retrieving the access and refresh tokens. Also write URL for backendserver in .env





## 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.





* **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.

**const apiUrl**: Defines a default URL for the API endpoint. This URL is used if the environment variable (VITE\_API\_URL) is not set.

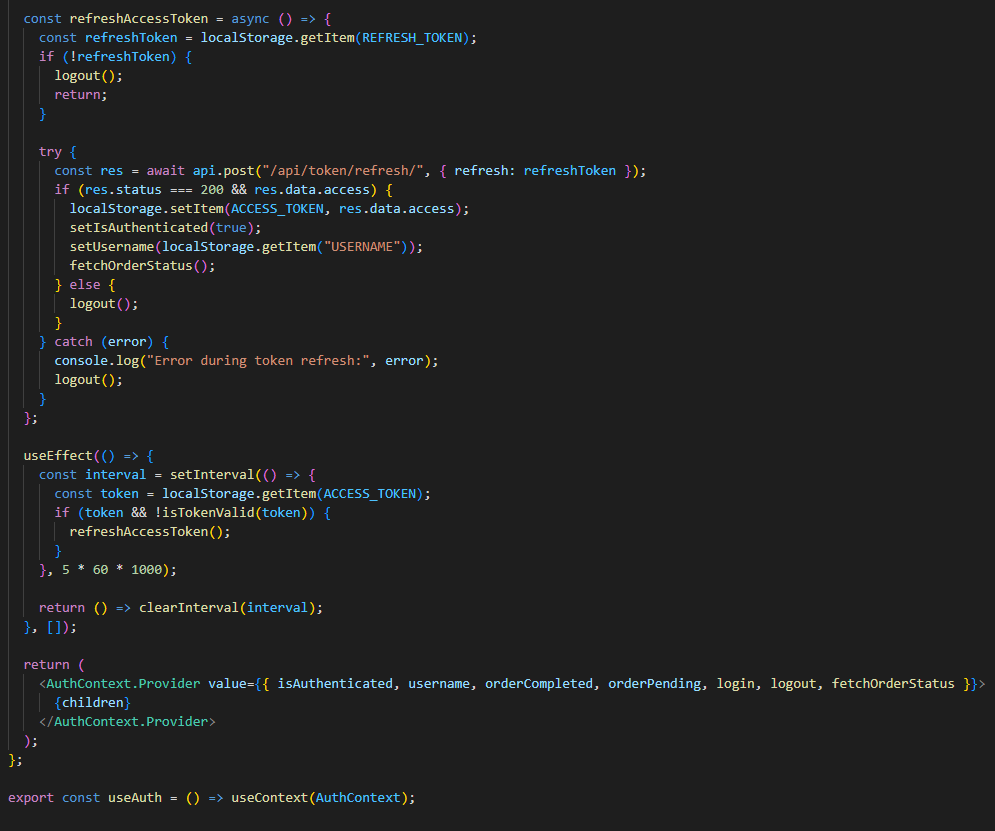
**axios.create()**: Creates a new Axios instance with a specified base URL.

* **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**: 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.

## Create Authentication

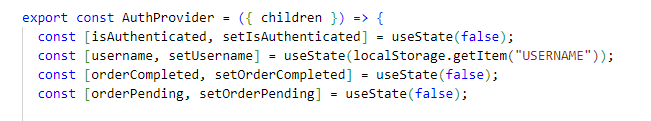
* **AuthContext**: Holds authentication state and functions for login, logout, and token management.
* **AuthProvider**: Wraps the app, enabling authentication context across components.
* **Login and Logout**: login saves authentication details and logout clears them.
* **Token Handling**: Checks token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



* **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.



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



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

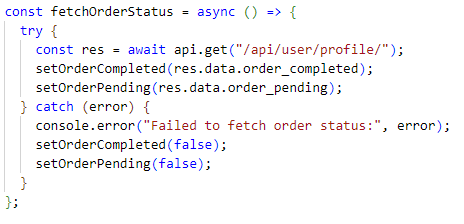
State Management

* **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.



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.

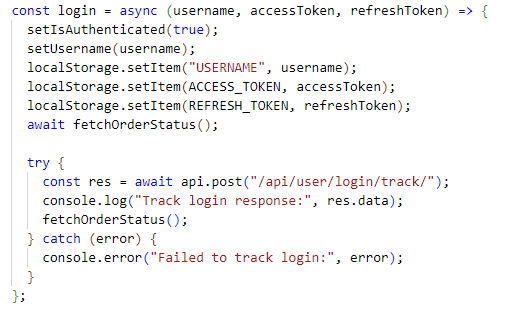


fetchOrderStatus Function

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.

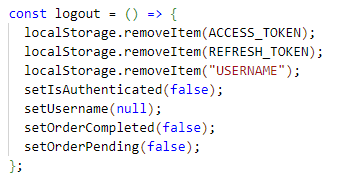


login Function

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.

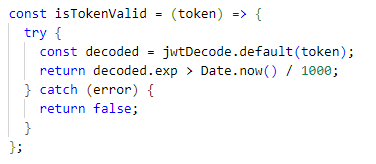


logout Function

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.

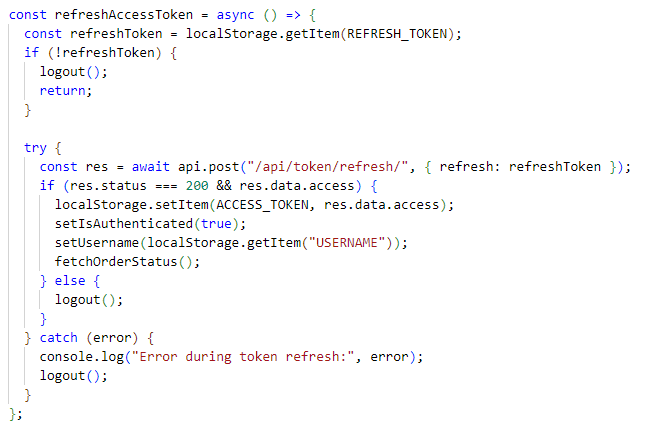


isTokenValid Function

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.

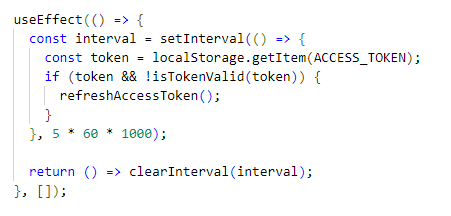
This function helps determine if the current access token can still be used or if it needs to be refreshed.



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.

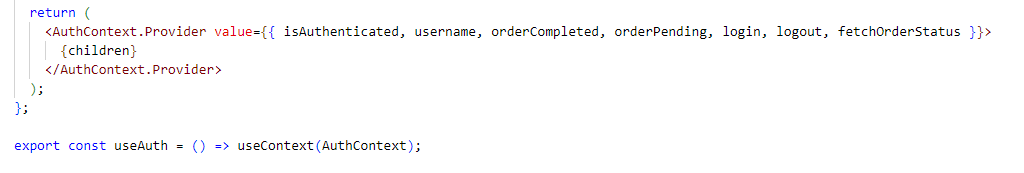


useEffect Hook for Token Refreshing

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.



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.

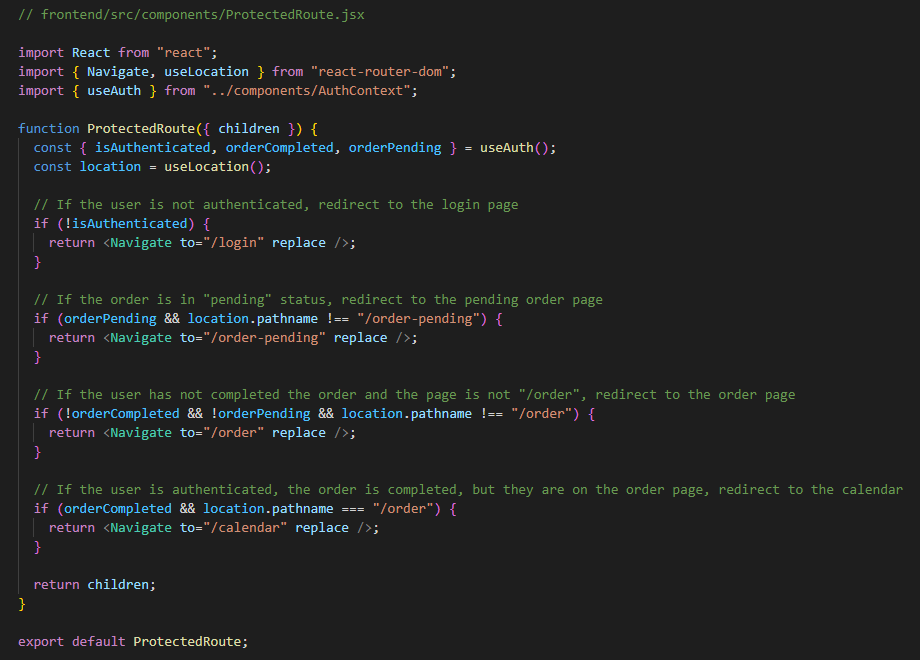
* **Purpose**:
  + This hook simplifies access to AuthContext by allowing components to call useAuth() directly instead of useContext(AuthContext), improving code readability and encapsulating the context logic.

## 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.

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





**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**

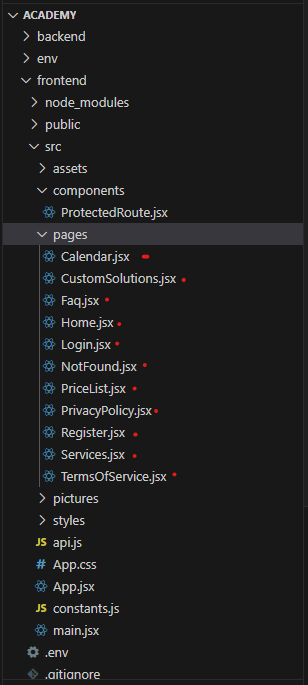
* **Function Definition**:
  + function ProtectedRoute({ children }): Defines the component that accepts children as a prop, representing the nested components that should be displayed if the access conditions are met.
* **Authentication Check**:
  + if (!isAuthenticated): If the user is not authenticated, 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**:
  + if (orderPending && location.pathname !== "/order-pending"): If the user has a pending order and is not already on the /order-pending page, they are redirected there. This ensures they stay on the pending page until the order is resolved.
  + if (!orderCompleted && !orderPending && location.pathname !== "/order"): If the user’s order is not completed and they do not have a pending order, they are redirected to the /order page unless they are already on it. This check ensures that users without completed orders cannot access protected routes until the order is completed.
  + if (orderCompleted && location.pathname === "/order"): If the user has completed the order but is currently on the /order page, they are redirected to the /calendar page. This check prevents users with completed orders from staying on the order page.
* **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.

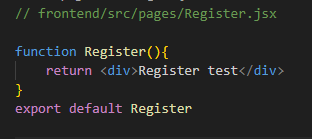
**Export**

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

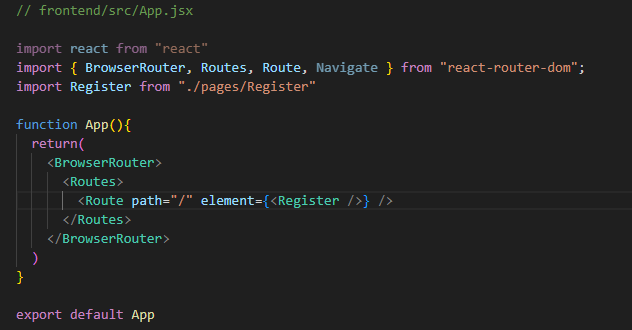
## Create page files

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.

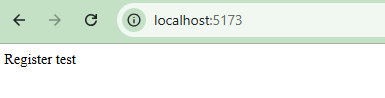




## Rewrite App

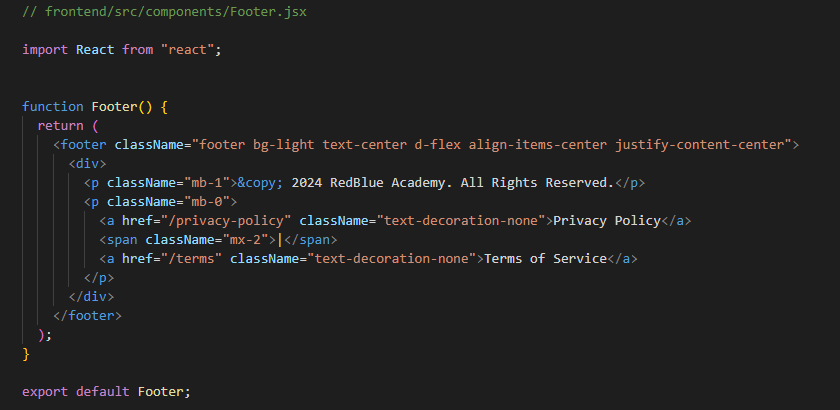
Rewrite the original App.jsx to create the first connection to the Register page. Start the server for the frontend and go to localhost:5173 in the browser.





## 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.



 **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.

Here’s a breakdown of the Footer component code:

### Overview

The Footer component provides a simple footer for the webpage with centered text and links to the Privacy Policy and Terms of Service pages. It uses Bootstrap classes to style and align the content.

### Code Breakdown

* **Imports**:
  + import React from "react";: This imports the React library, which is necessary for defining components.
* **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.
* **Summary**

This Footer component:

* Displays copyright information for RedBlue Academy.
* Provides links to the Privacy Policy and Terms of Service pages.
* Uses Bootstrap classes to create a clean, centered, and responsive footer layout.

## 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.



 **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.

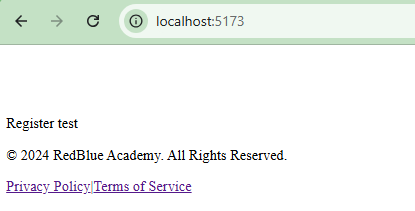
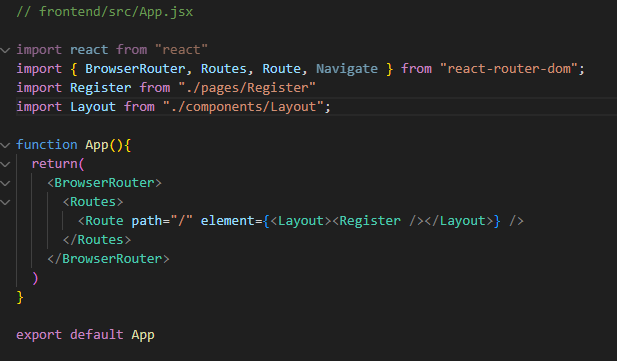
* **flex-grow-1**: Allows the main content to expand and fill available space.
* **style={{ paddingTop: '70px' }}**: Adds padding to the top, likely to create space for a header or navbar.
* **{children}**: Renders any child components passed to the Layout.

 **<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.

## Update App with Layout

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



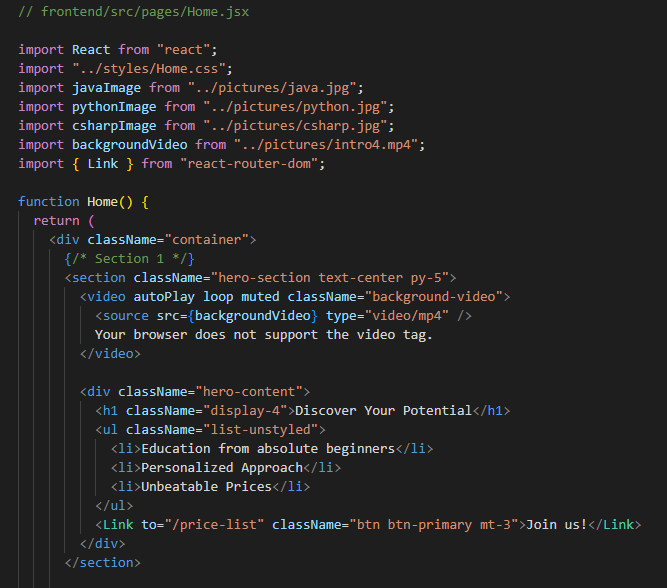
## Create first css

Create first css for Footer and do test

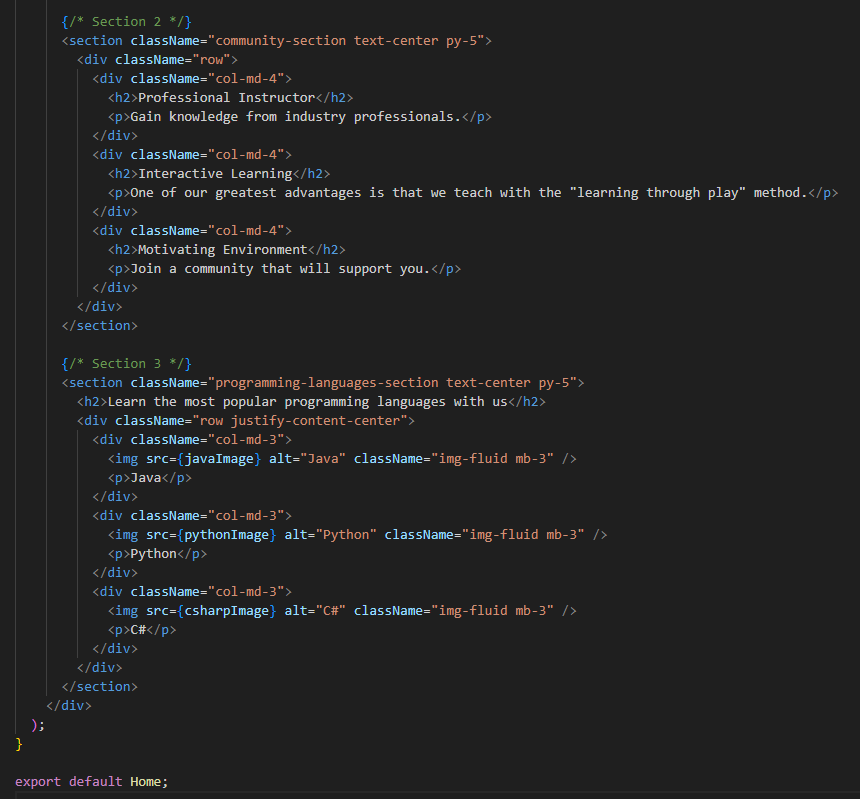


## Create page Home with style

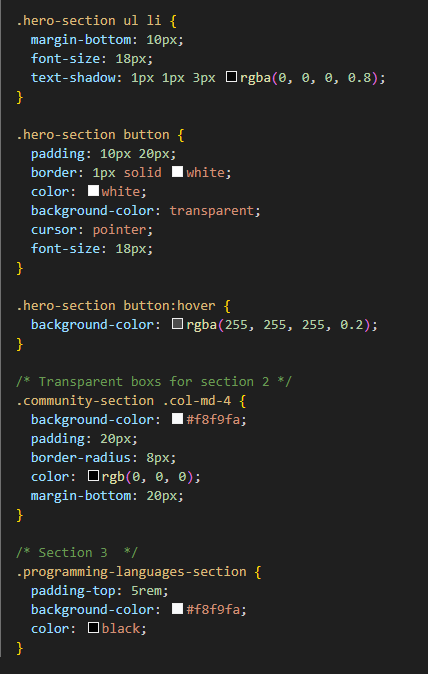
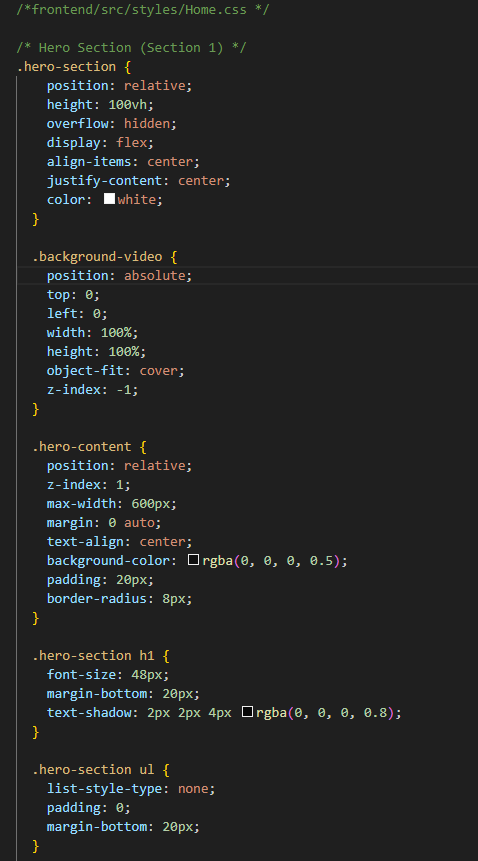
Home component in a React application that represents the homepage. The component is structured into three main sections:



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



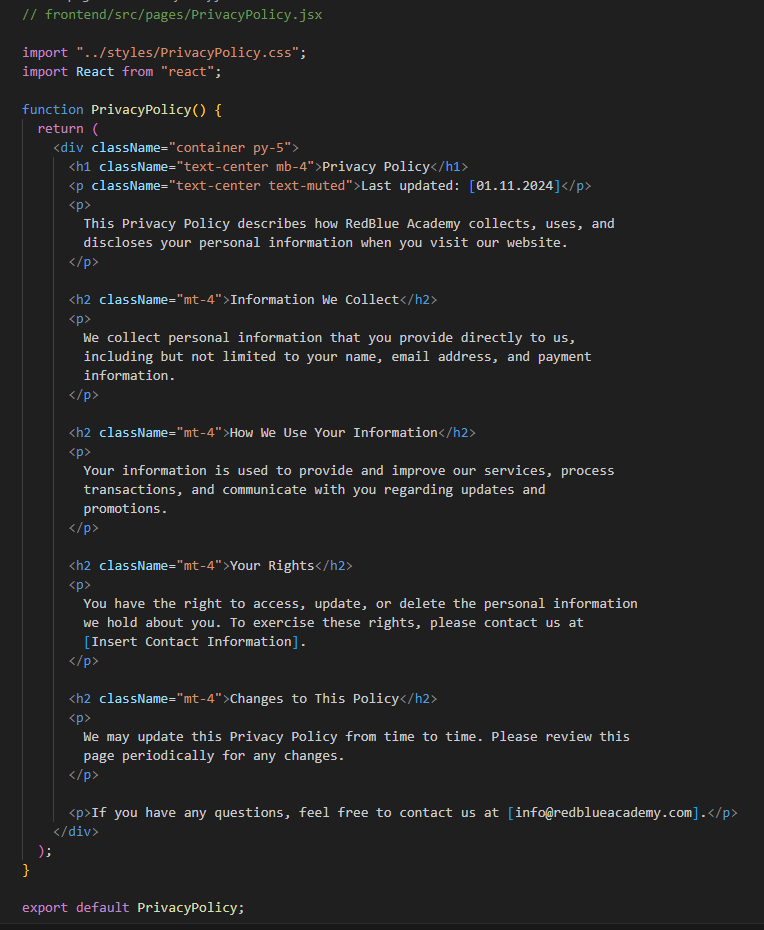
1. **Community Section**:
   * Displays three columns with information about the academy’s learning environment:
     + **Professional Instructor**
     + **Interactive Learning** (with a "learning through play" approach)
     + **Motivating Environment**
2. **Programming Languages Section**:
   * Highlights 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) 

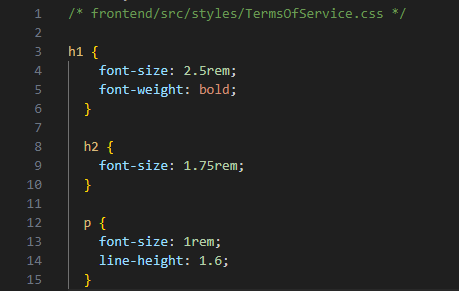
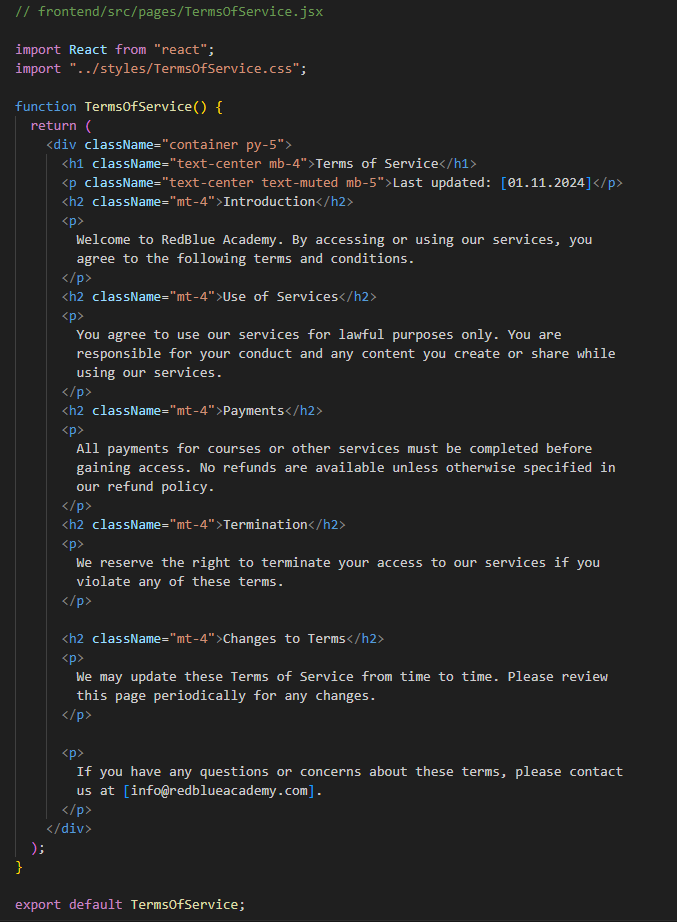
The CSS styles these three sections:

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.

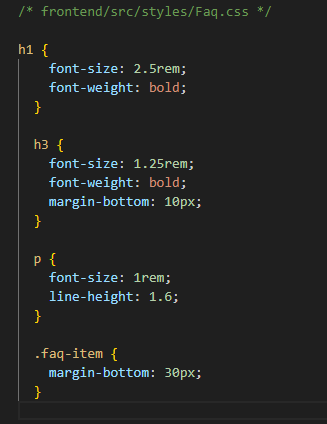
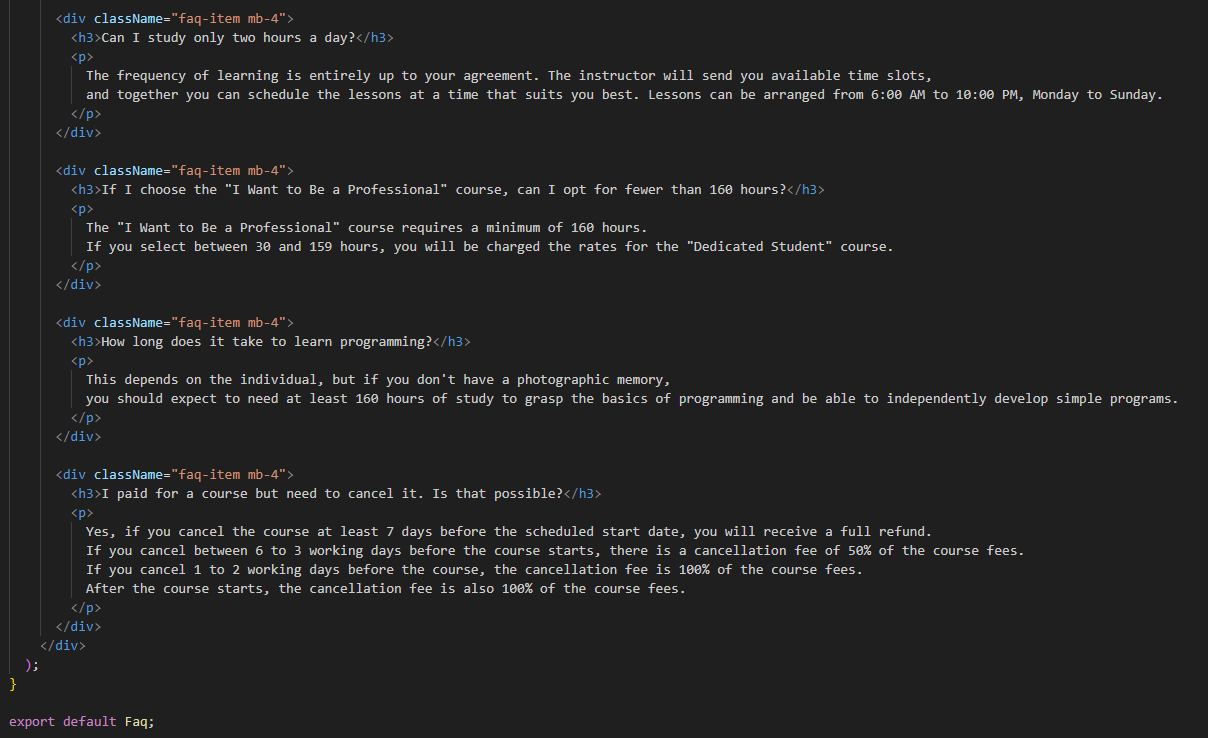
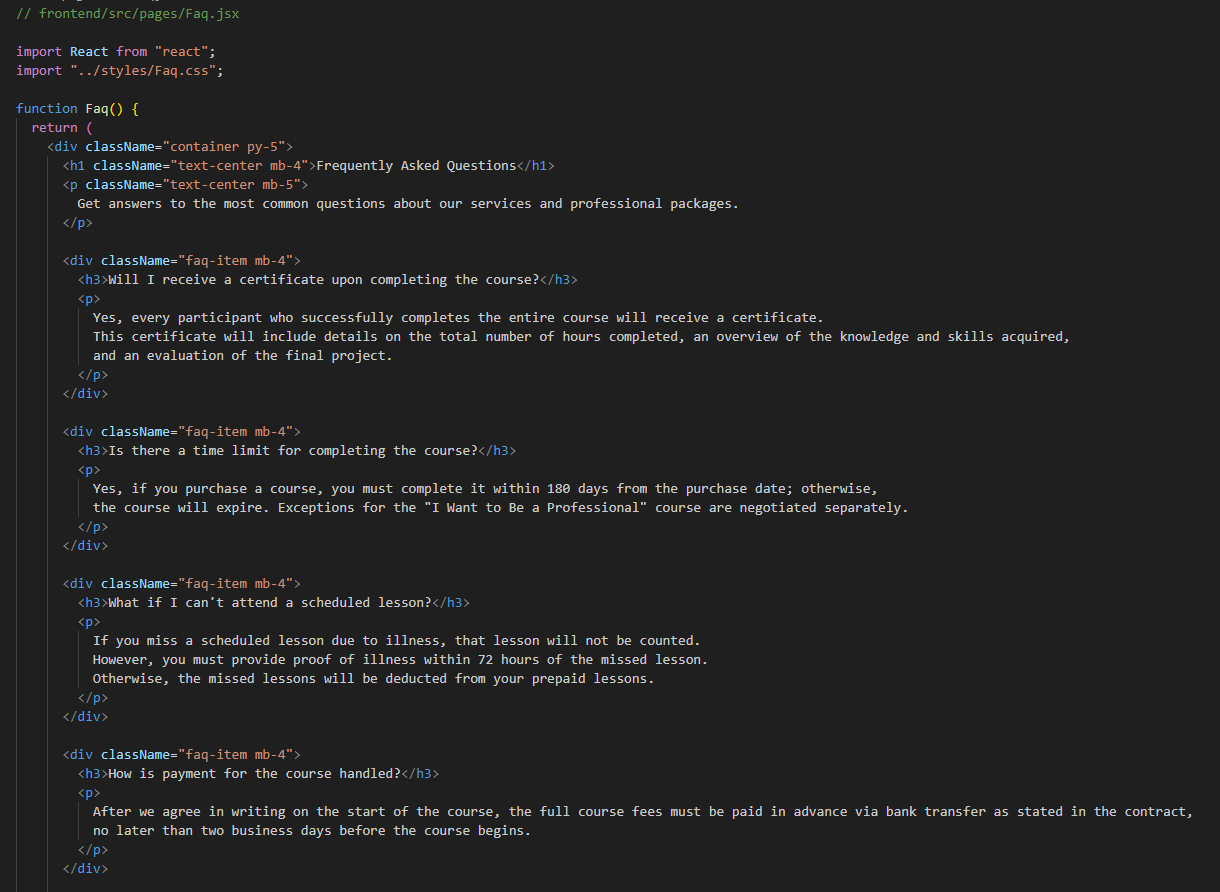
## Create page Privacy Policy with style



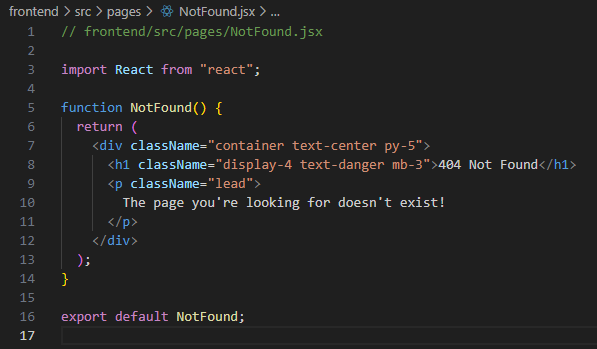
## Create page Terms of Service with style



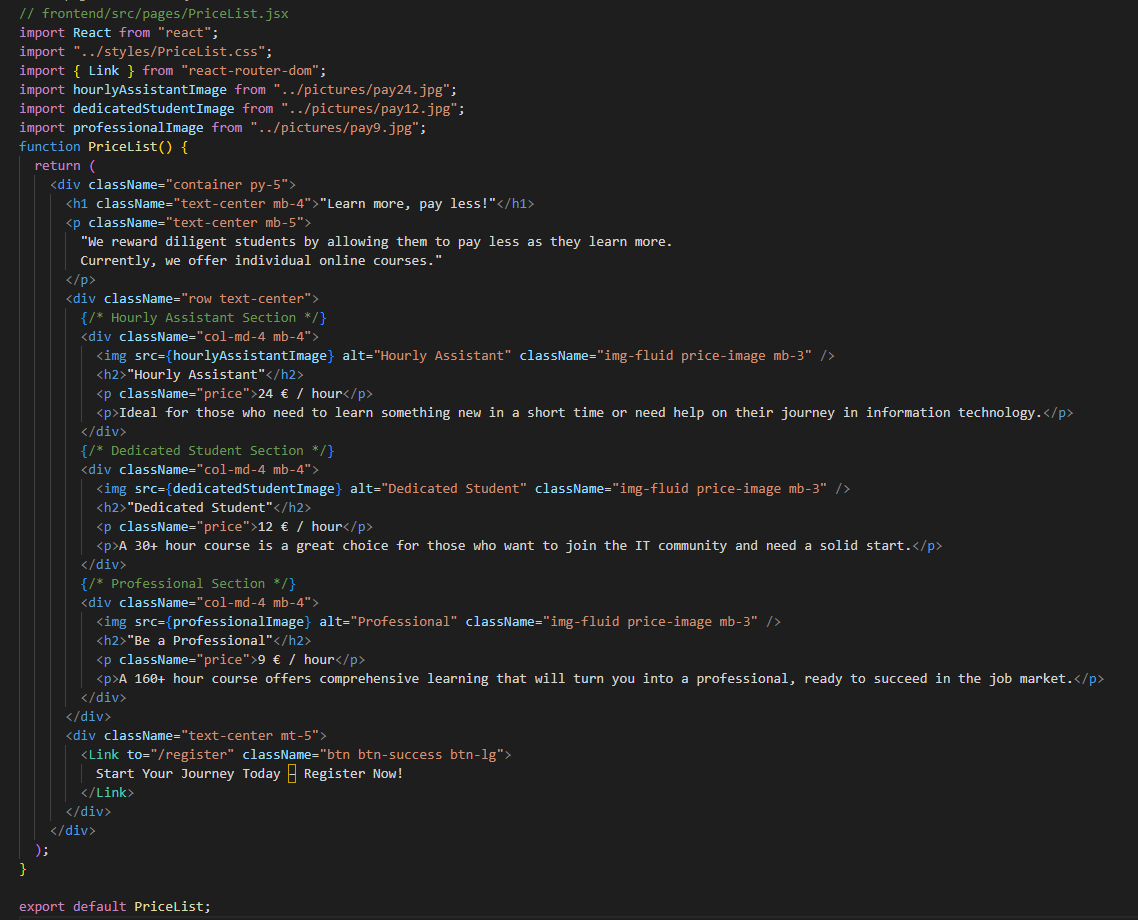
## Create page FAQ with style

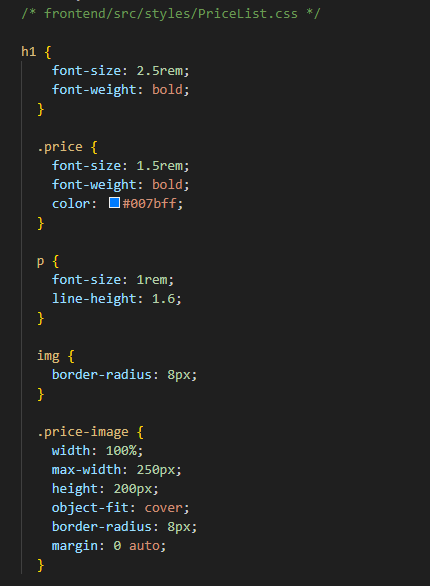
FAQ

## Create page Not Found



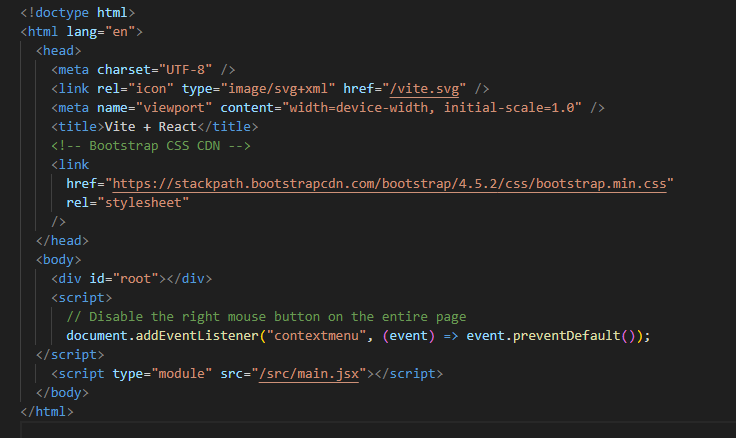
## Create page Price with styled



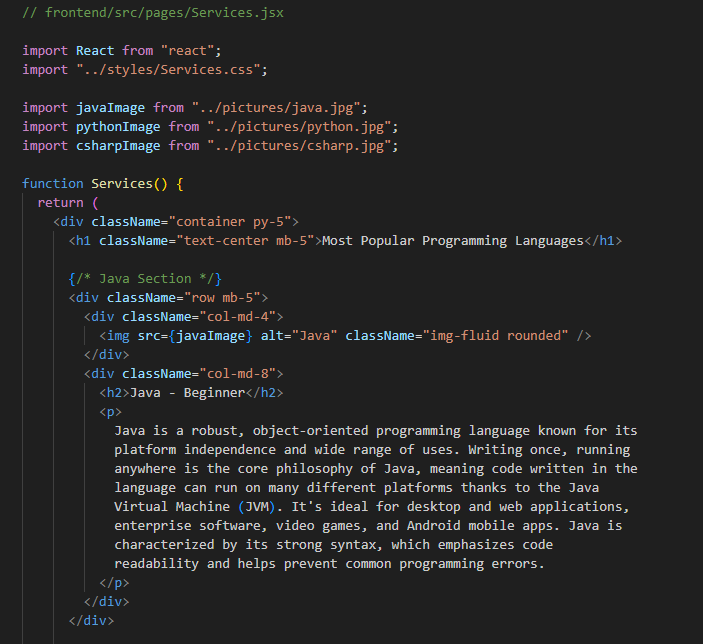


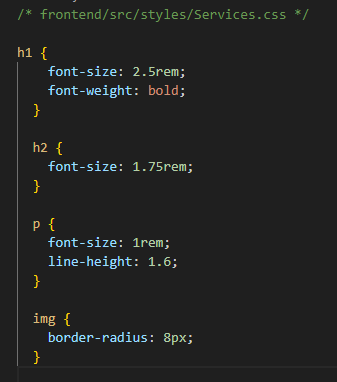
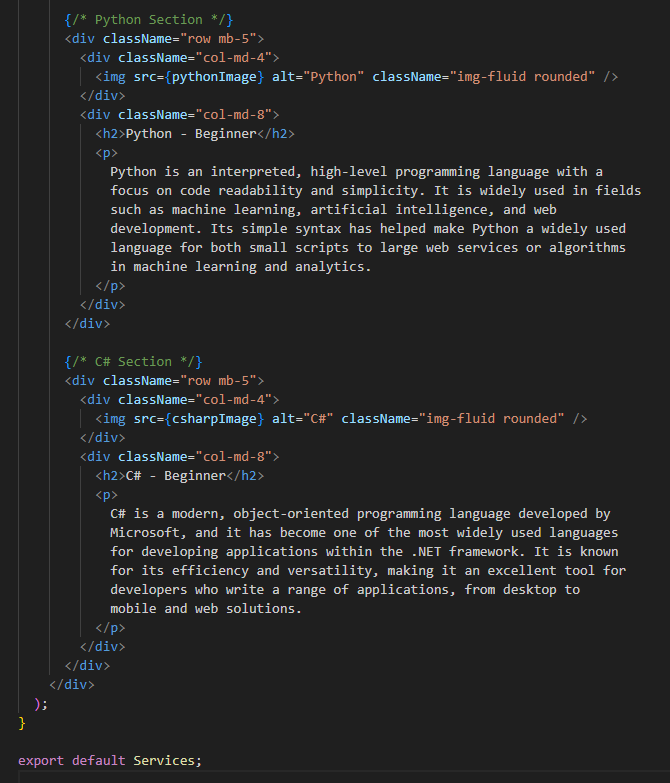
## Update index.html

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



## Create page Services with style

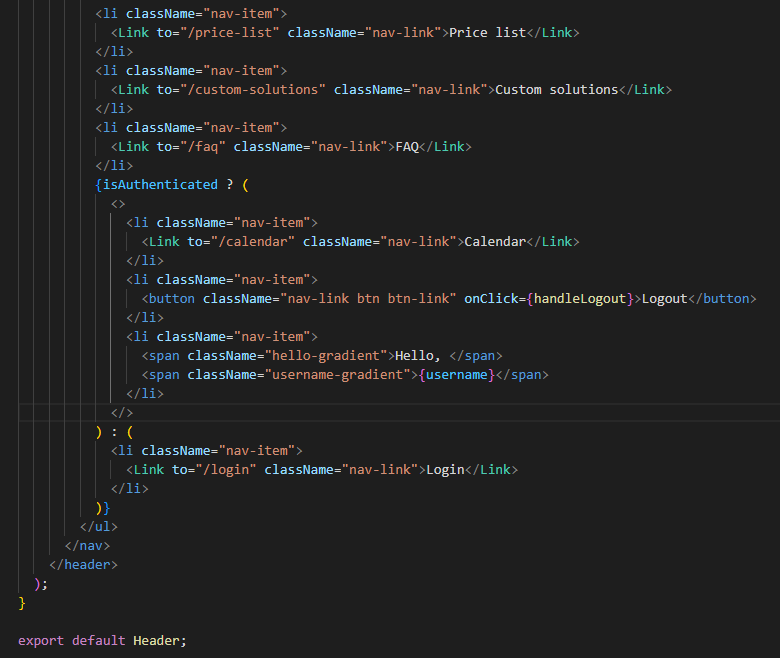


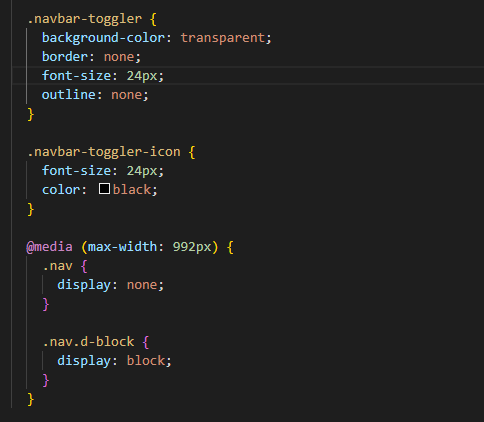
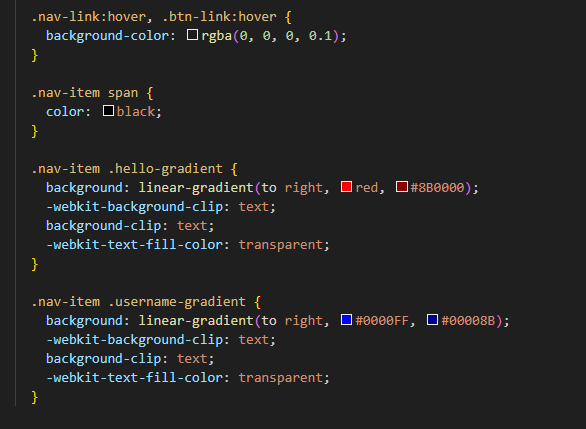
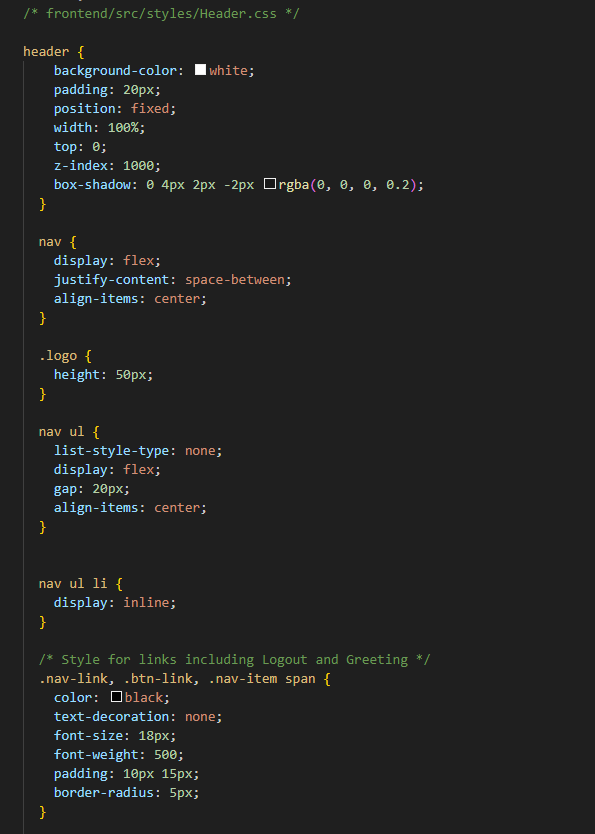


## Create component Header with style

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.



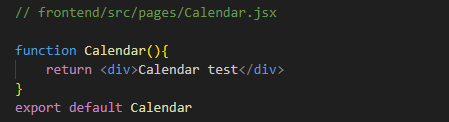




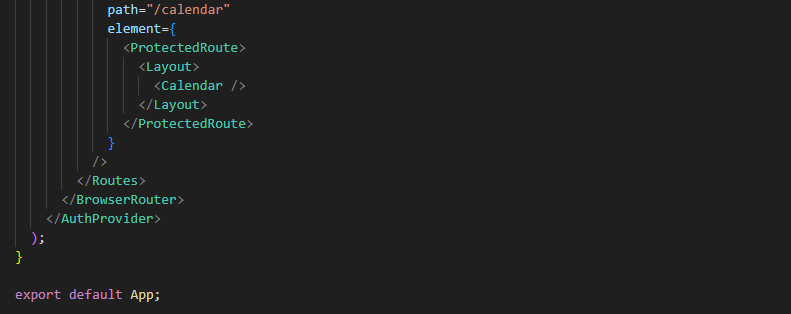
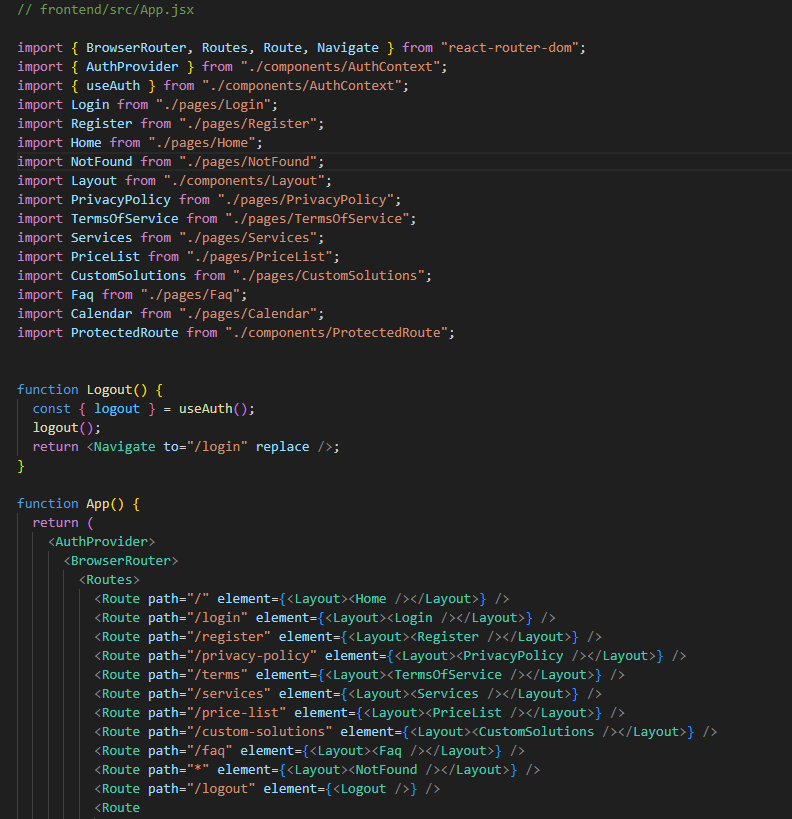
## Update Layout and write there Header



## Create page Calendar



## Update App with new connection



## Run servers and check pages

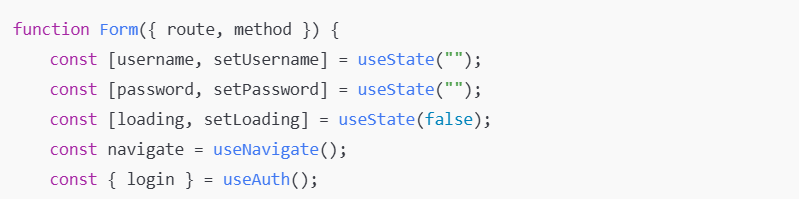


## Create Component Form.jsx with style

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



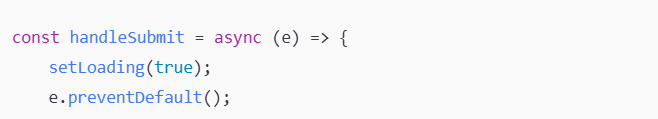
 **username, password**: States to manage the input values for username and password.

 **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.

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

**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).

 **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)**:

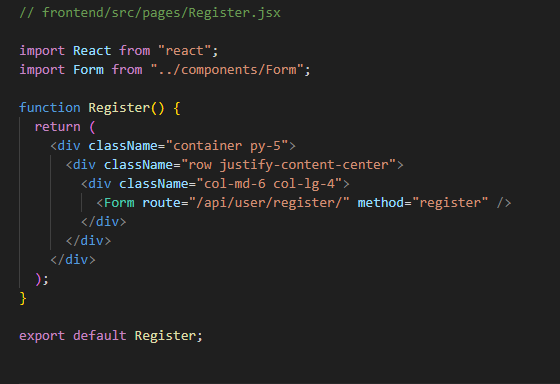
* If the method is not "login," the user is redirected to the login page (/login) after registration.

 **Error Handling**: Shows an alert if login or registration fails.

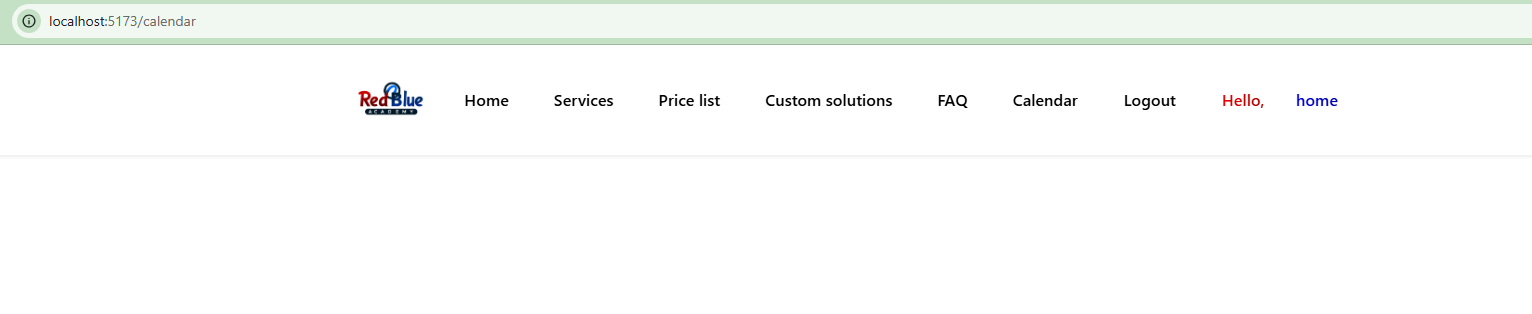
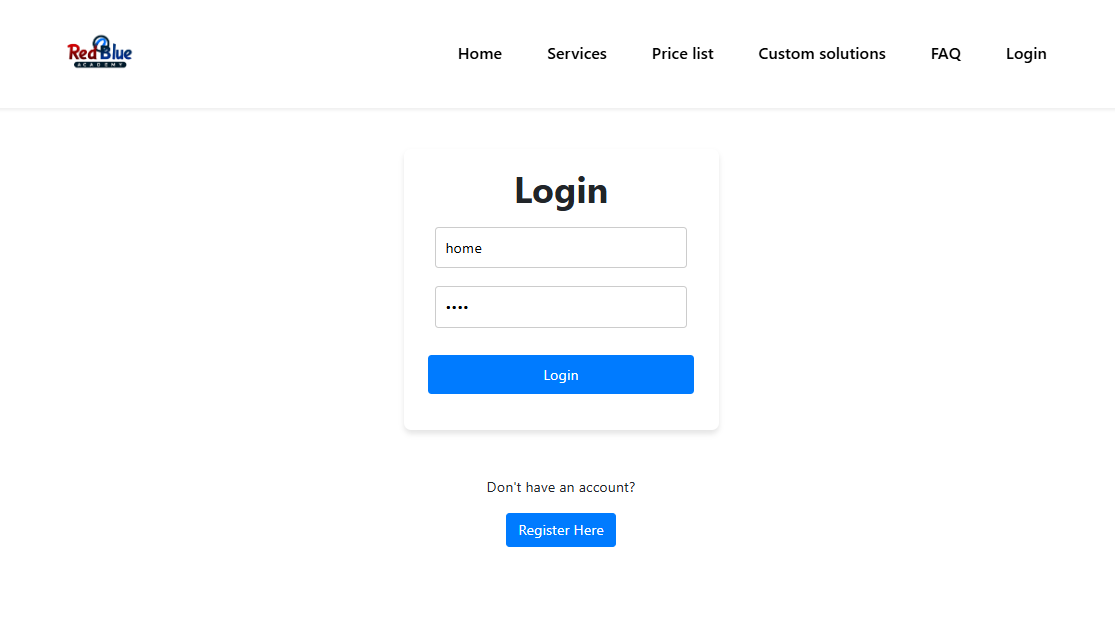
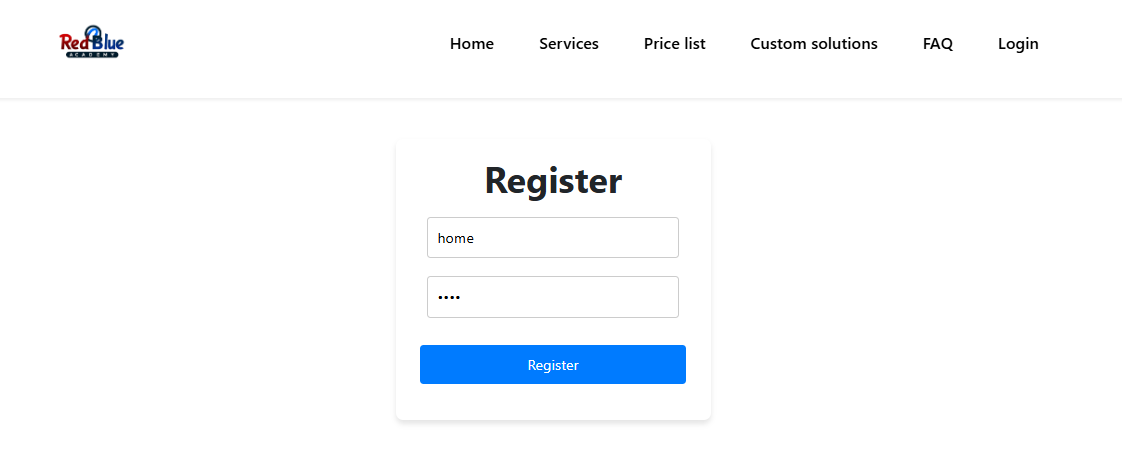
 **finally**: Always sets loading back to false after processing, to disable the loading indicator.



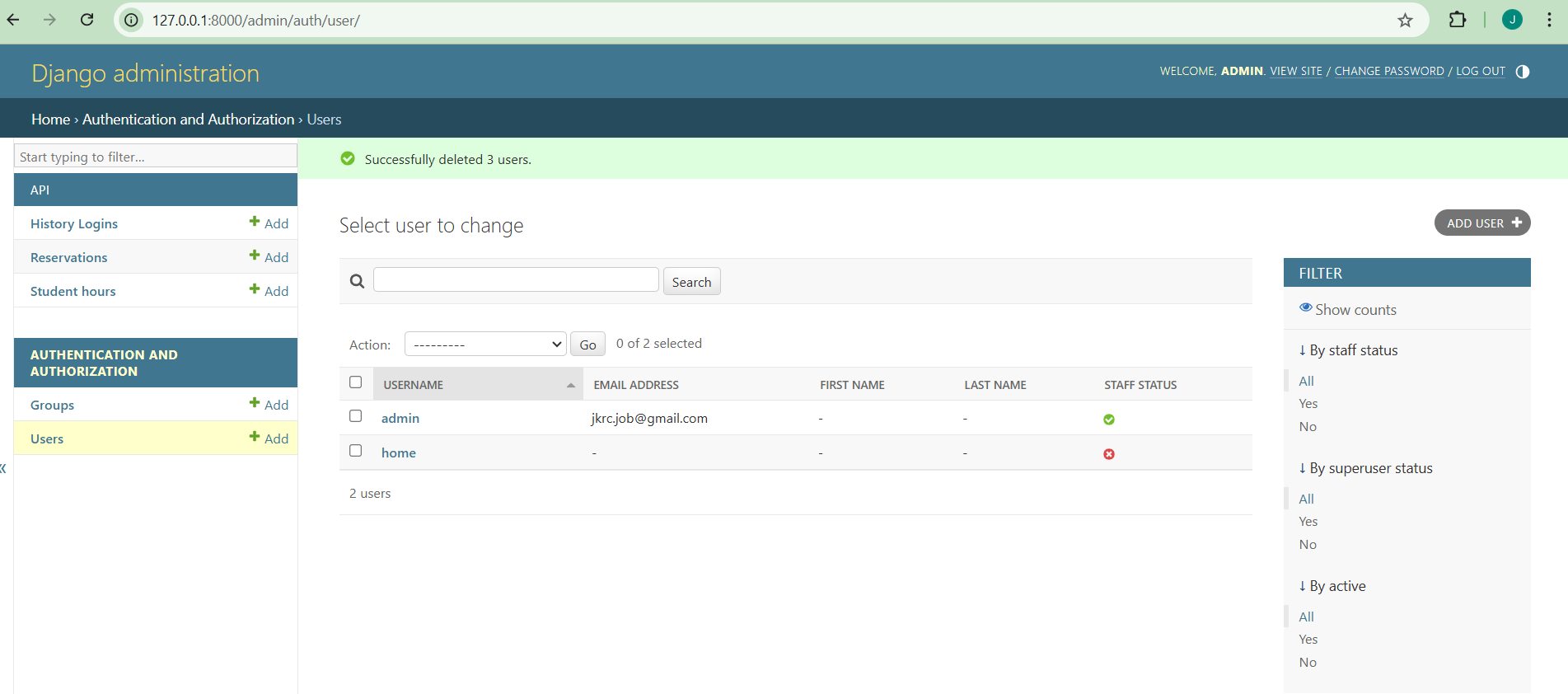
## Update Login and Register Page



## Start backend and frontend server server and make test



## Open Django Administrator



## 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.



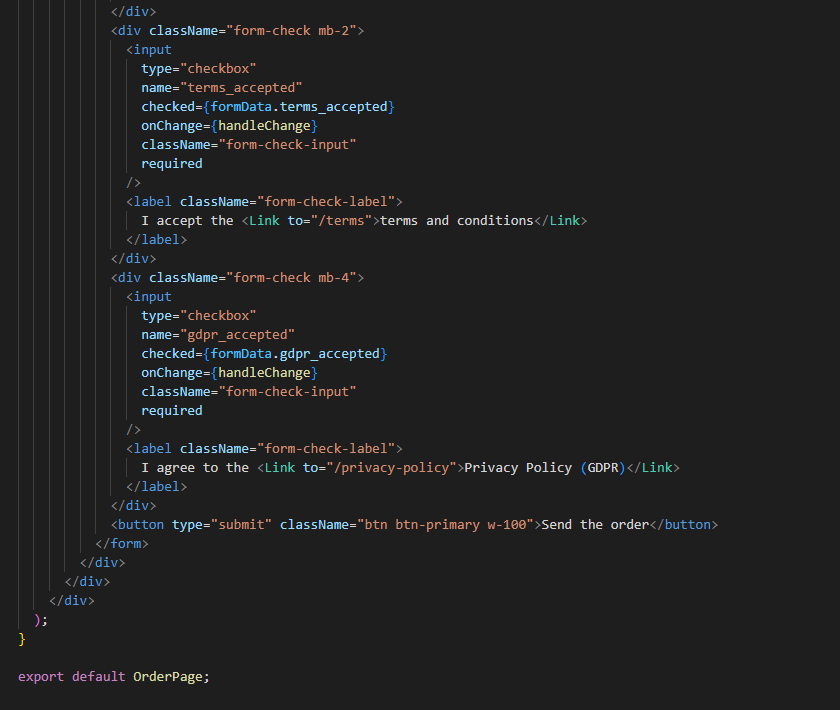
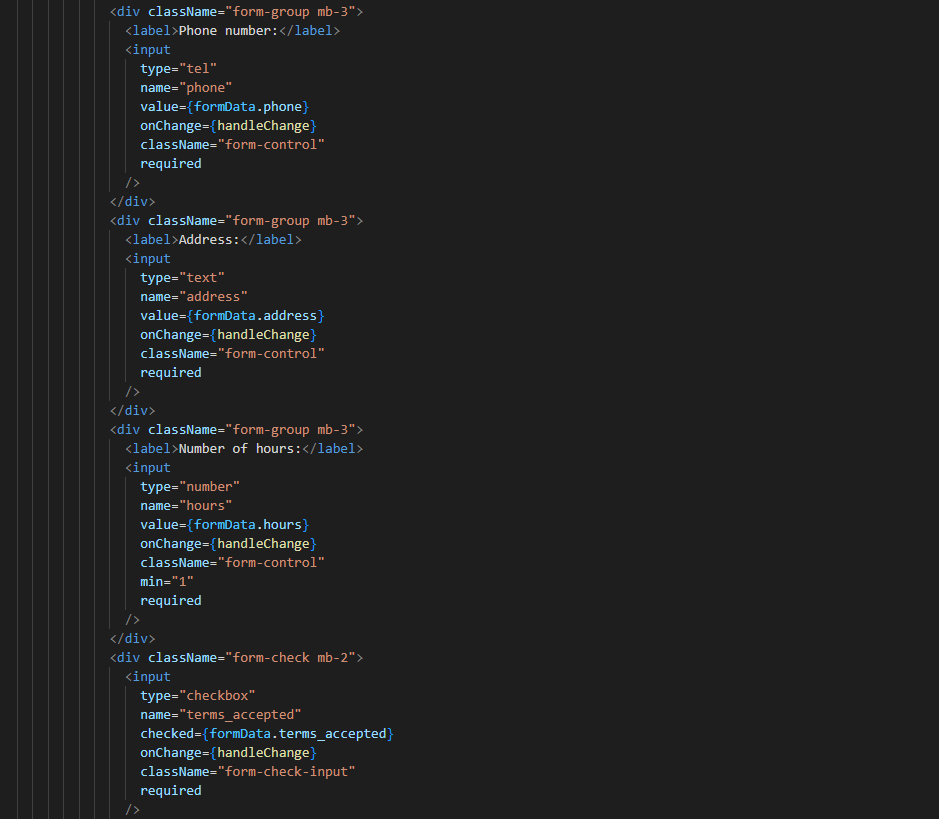
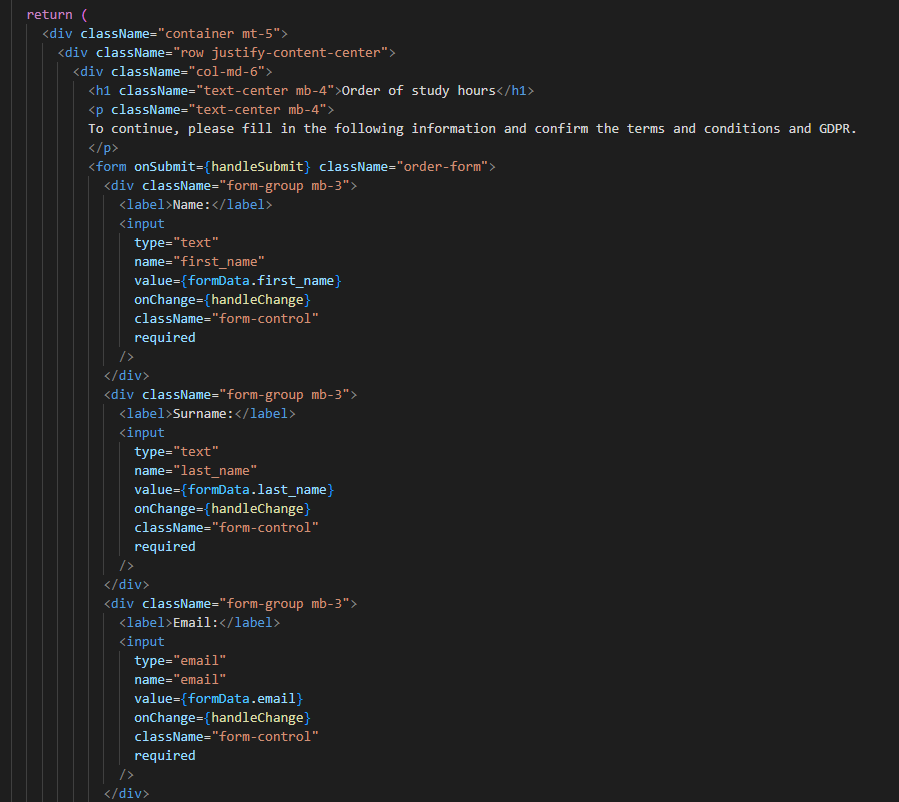
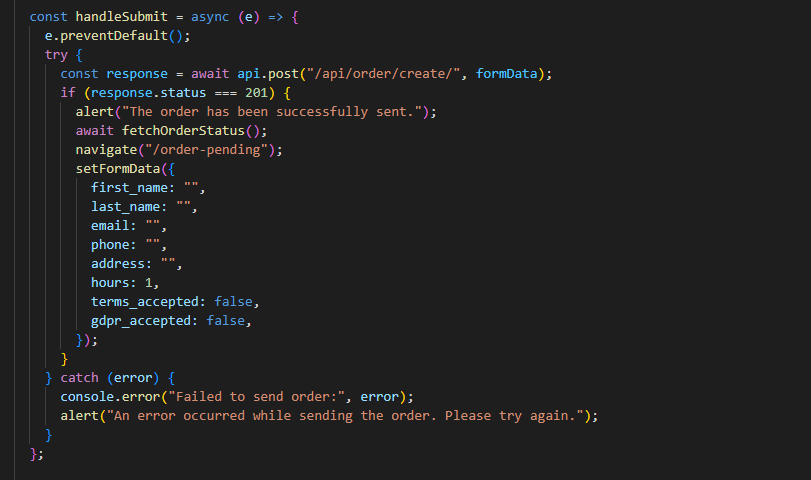
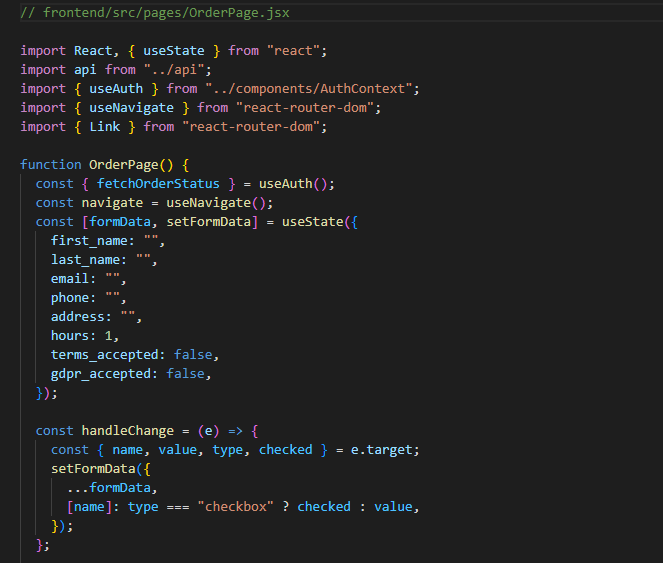
 **@fullcalendar/react**: This is the core FullCalendar package for React, which provides the main calendar component compatible with React applications.

 **@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.

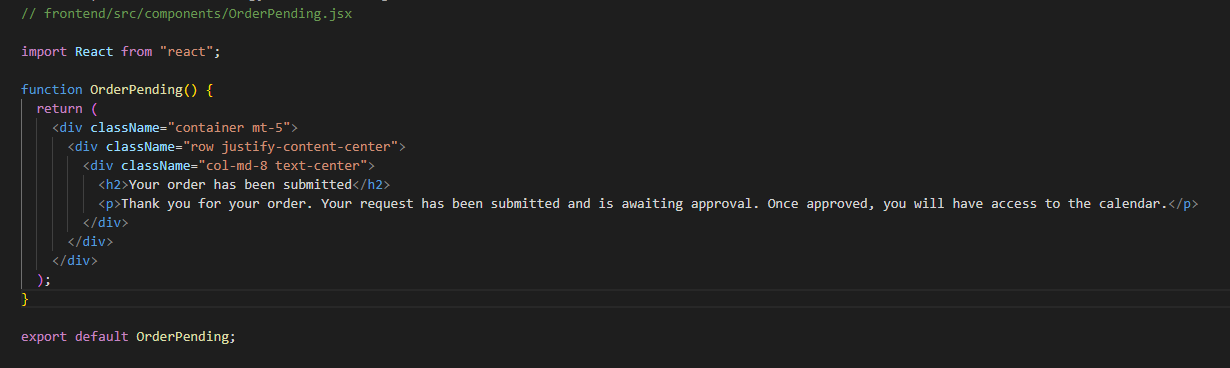
 **@fullcalendar/interaction**: Enables interactivity features, including drag-and-drop functionality for moving events and click handling for selecting or editing events directly in the calendar.

## Create 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.

## 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.

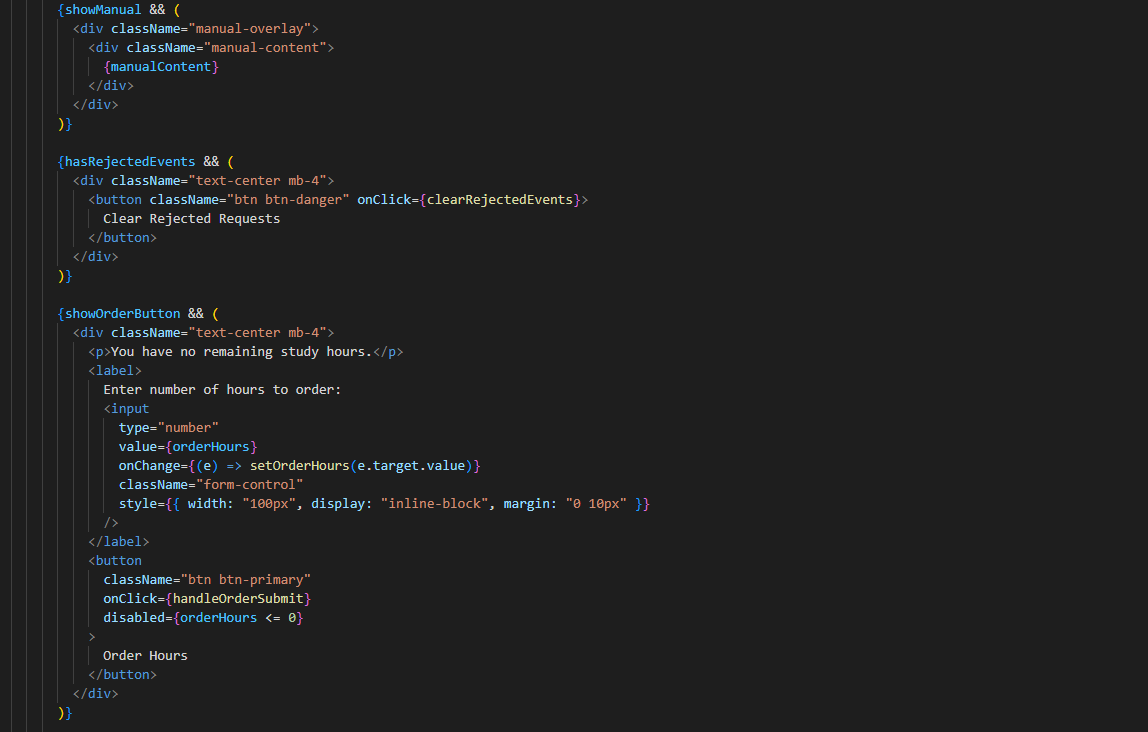
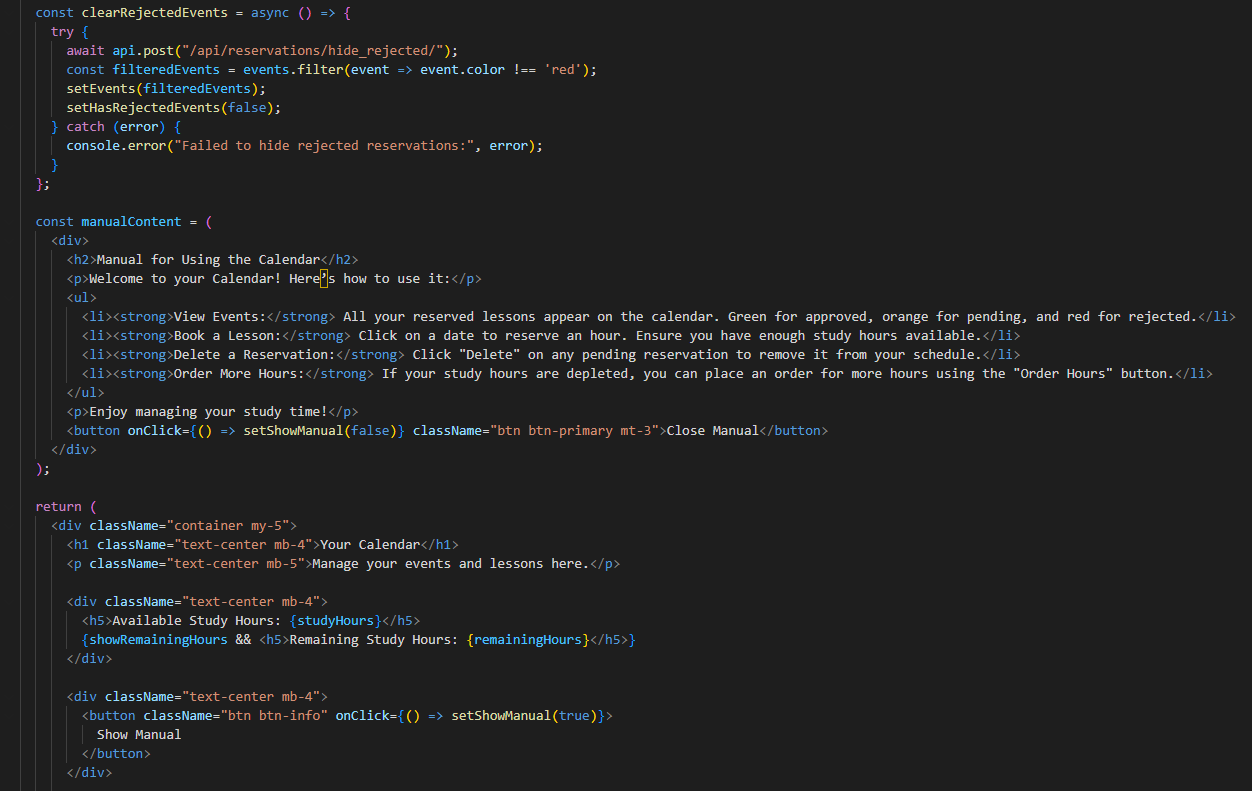
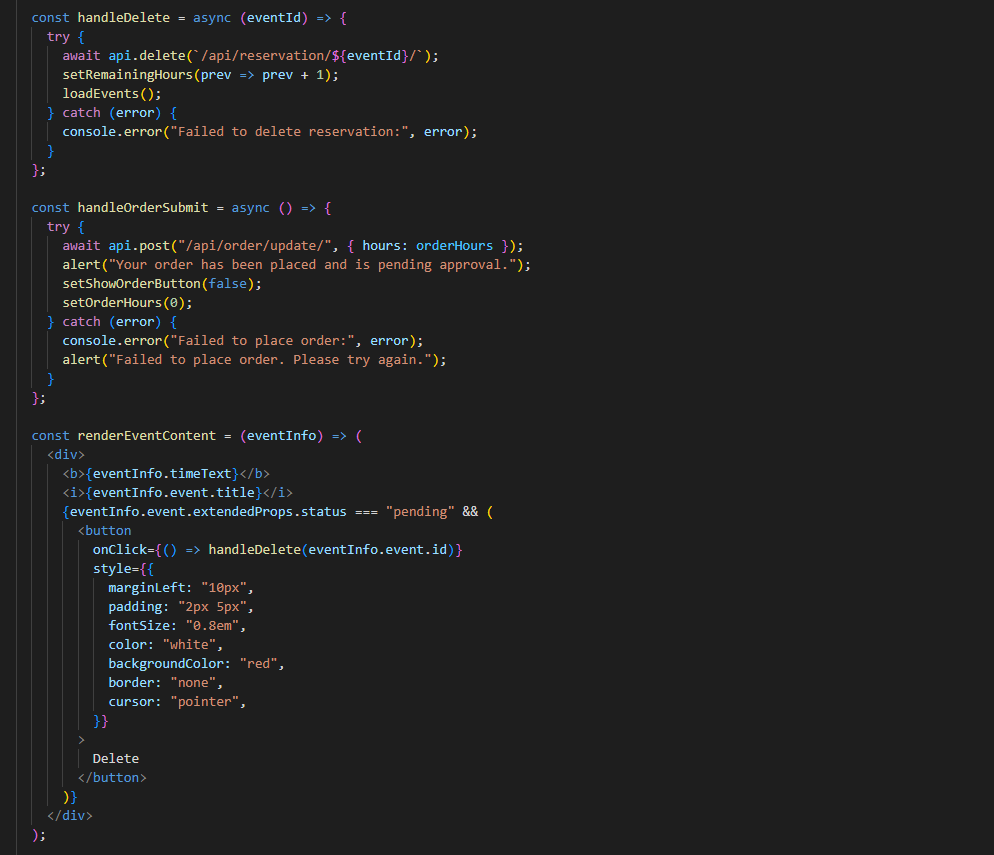
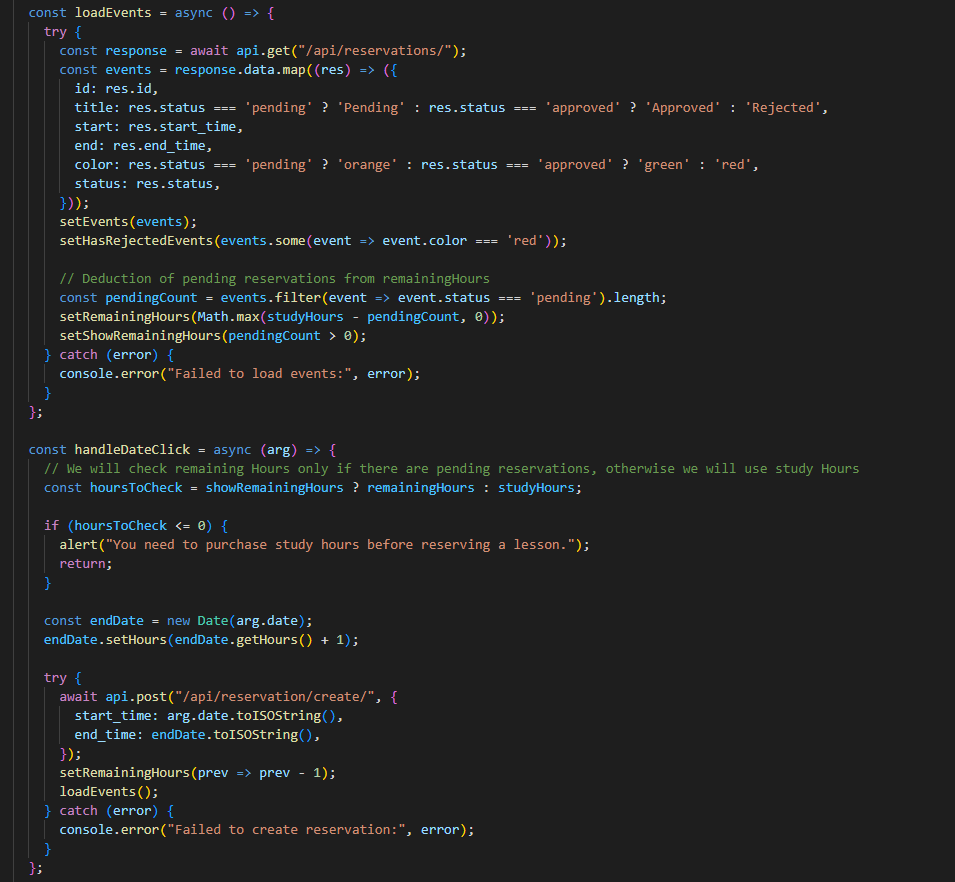
## Update Calendar

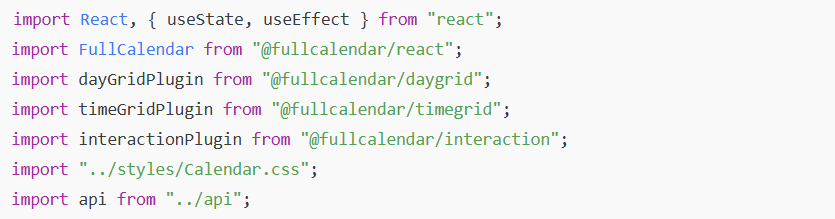
The Calendar component:

* **Displays available study hours** and calculates remaining hours after accounting for pending reservations.
* **Loads existing reservations** from the backend, color-coding them based on their status:
  + **Green** for approved reservations.
  + **Orange** for pending reservations.
  + **Red** for rejected reservations.
* **Allows users to**:
  + **Reserve new lessons** by clicking on available time slots.
  + **Delete pending reservations**.
  + **Order additional study hours** if they have none remaining.
* **Provides a user manual** explaining how to use the calendar.
* **Clears rejected reservations** upon user request.

The component renders a structured layout containing:

* **Header**: Title and description.
* **Study Hours Display**:
  + Shows available and remaining study hours.
* **Buttons**:
  + "Show Manual" button to display the user manual.
  + "Clear Rejected Requests" button if there are rejected events.
  + "Order Hours" section if the user has no remaining hours.
* **Calendar**:
  + The FullCalendar component is configured with:
    - Plugins: dayGridPlugin, timeGridPlugin, interactionPlugin.
    - Initial view: "timeGridWeek".
    - Time range: "07:00:00" to "22:00:00".
    - Events: Loaded from the events state.
    - dateClick: Handler to create new reservations.
    - eventContent: Custom renderer for event content.
    - Time formatting settings.



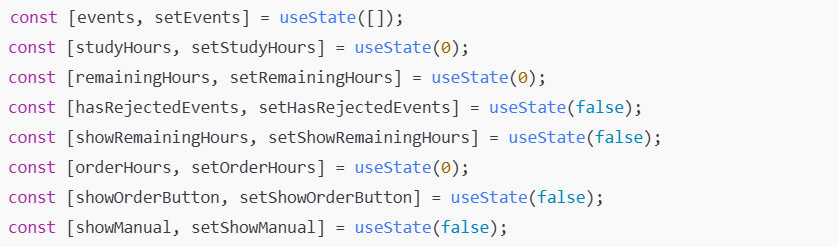
 **React and Hooks**: useState and useEffect for state management and side effects.

 **FullCalendar and Plugins**: Used to render the calendar with various functionalities.

* FullCalendar: The main calendar component.
* dayGridPlugin, timeGridPlugin, interactionPlugin: Plugins to enable different views and interactivity.

 **Stylesheet**: Calendar.css for custom styling.

 **API Module**: api for making HTTP requests to the backend.

 **events**: Stores the calendar events (reservations).

 **studyHours**: Total study hours available to the user.

 **remainingHours**: Study hours remaining after accounting for pending reservations.

 **hasRejectedEvents**: Indicates if there are any rejected reservations.

 **showRemainingHours**: Controls the display of remaining hours.

 **orderHours**: Number of additional hours the user wants to order.

 **showOrderButton**: Determines whether to show the "Order Hours" button.

 **showManual**: Controls the visibility of the user manual.

**Purpose**: Runs once on component mount to fetch study hours and load existing reservations.

 **Purpose**: Retrieves the user's available study hours from the backend.

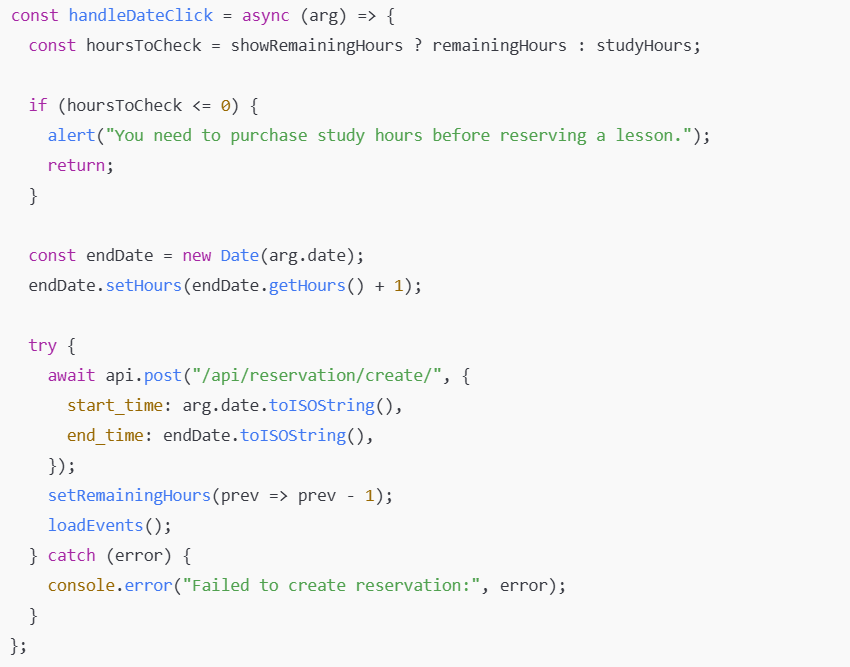
 **Actions**:

* Updates studyHours and remainingHours state variables.
* Determines whether to display the "Order Hours" button based on available hours.

 **Purpose**: Fetches reservations from the backend and formats them for the calendar.

 **Actions**:

* Maps the response data to the events state, adding color coding based on reservation status.
* Checks for any rejected events to control the display of the "Clear Rejected Requests" button.
* Updates remainingHours by subtracting the number of pending reservations.
* Determines whether to display remainingHours.

 **Purpose**: Handles user clicks on the calendar to create new reservations.

 **Actions**:

* Checks if the user has enough study hours.
* Sets the end time one hour after the start time.
* Sends a POST request to create a new reservation.
* Updates remainingHours and reloads events upon success.

 **Purpose**: Allows users to delete pending reservations.

 **Actions**:

* Sends a DELETE request to remove the reservation.
* Updates remainingHours and reloads events upon success.

 **Purpose**: Handles the submission of additional study hour orders.

 **Actions**:

* Sends a POST request with the number of hours the user wants to order.
* Displays a success or error message based on the response.

 **Purpose**: Customizes how events are displayed on the calendar.

 **Actions**:

* Shows the time and title of the event.
* If the event is pending, displays a "Delete" button to allow cancellation.

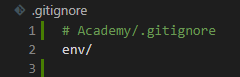
 **Purpose**: Clears rejected reservations from the calendar.

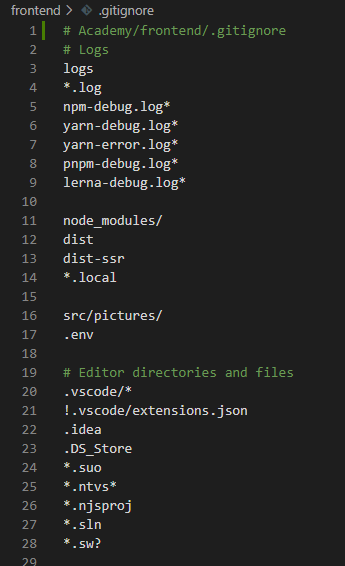
 **Actions**:

* Sends a POST request to hide rejected reservations.
* Filters out rejected events from the events state.
* Updates hasRejectedEvents accordingly.

# GitHub

## Create .gitignore



## 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