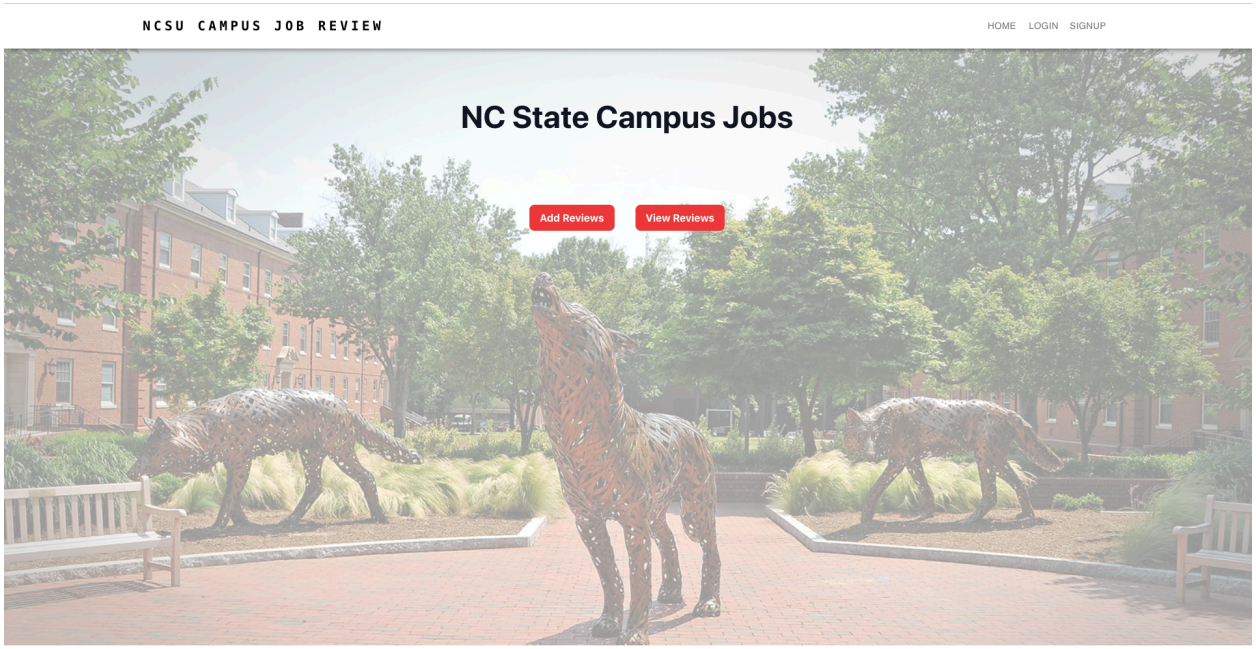
CSC Software Engineering Project 2

Campus Job Review System

Software Code Documentation



Find the perfect campus job with honest reviews from fellow students, so you know exactly what you're signing up for before you even apply!

Team Members

Tahreem Yasir
Shazia Muckram
Amay Gada

CSC Software Engineering Project 2

Introduction

This document provides a comprehensive overview of the CampusJobReview project, detailing its architecture, code structure, and intended use. It includes in-depth documentation of each module, class, and function, along with usage examples. This guide is designed to assist developers, contributors, and users by clarifying the project's purpose, architecture, and functionality, ultimately supporting maintenance, development, and expansion.

Project Description

CampusJobReview is a Django-based web application aimed at helping students find reliable information about on-campus jobs. By centralizing reviews and ratings from current and past student employees, the platform empowers users to make informed decisions based on real job experiences. Students can filter positions based on department, rating, pay, and other relevant criteria, providing a comprehensive view of job expectations and benefits.

The *CampusJobReview* project is a web platform that helps students share and explore reviews about on-campus jobs. Built with Django and MongoDB for backend management and React for frontend interactivity, it supports dynamic job review posting, filtering by criteria like pay and location, and uses JWT authentication for secure access. The system also includes a browser extension for easy access to job reviews.

Technologies Used

The following technologies were used to create a user-friendly platform for students seeking insights into campus employment opportunities.

1. **Django**: The core web framework used for handling backend logic, including API endpoints and authentication.
2. **MongoDB**: A NoSQL database that stores review data, providing scalability and dynamic querying capabilities.
3. **React**: A JavaScript library used to build the frontend interface, enabling a responsive and interactive user experience.
4. **JWT (JSON Web Tokens)**: Implements secure user authentication, allowing safe access to job reviews and data.
5. **Browser Extension**: Integrated for streamlined access to job reviews and automated data entry when browsing job listings.

Architecture

The **CampusJobReview** project consists of several key components and modules:

1. **review_backend**
2. **review_frontend**
3. **extension**
4. **database**
5. **authentication**

Below we have described the components in more details with respect to the functionality.

6. **review_backend**: The Django REST API responsible for handling job reviews, ratings, and user authentication. This component further consists of the following modules:
    a. **review_backend**: This is the core module that contains the main application logic, including models, serializers, views, and URLs for the job review application.
    b. **service**: This module likely handles business logic related to the reviews, such as managing user interactions and integrating external services for job postings or reviews.
    c. **auth_review**: This module is focused on authentication and authorization, managing user accounts, permissions, and ensuring secure access to the review functionalities.
    d. **database**: MongoDB is used for storing job reviews, user data, and other relevant information.
    e. **authentication**: Utilizes JWT (JSON Web Tokens) for secure user authentication.
7. **review_frontend**: A React application that provides the user interface for interacting with the job review system. This component further includes the following modules:
    a. **src**: This directory contains the main source code for the frontend application. It includes components, pages, and other essential files for building the user interface.
    b. **service**: This component is responsible for managing API calls and data handling. It serves as a bridge between the frontend and backend, facilitating communication and data retrieval.
    c. **node_modules**: This directory contains all the dependencies and libraries installed via npm (Node Package Manager). It is essential for the functioning of the frontend application but is usually not modified directly.
    d. **extension**: A browser extension for easy access to the review platform, enhancing user engagement.

These modules collectively create a comprehensive system for students to evaluate their on-campus job experiences.

CSC Software Engineering Project 2

The main Tree structure of the project is given below:

```
CampusJobReview/
├─ review_backend/
│  ├─ auth_review/
│  ├─ manage.py
│  ├─ requirements.txt
│  ├─ review_backend/
│  └─ service/
└─ review_frontend/
   ├─ src/
   │  ├─ api/
   └─ tailwind.config.js
```

```
CampusJobReview/
├─ review_backend/
│  ├─ auth_review/
│  │  ├─ admin.py
│  │  ├─ apps.py
│  │  ├─ models.py
│  │  ├─ serializers.py
│  │  ├─ tests_models.py
│  │  ├─ tests_serializers.py
│  │  ├─ tests_views.py
│  │  ├─ urls.py
│  │  └─ views.py
│  ├─ manage.py
│  ├─ requirements.txt
│  ├─ review_backend/
│  │  ├─ .env
│  │  ├─ asgi.py
│  │  ├─ settings.py
│  │  ├─ urls.py
│  │  └─ wsgi.py
│  └─ service/
│     ├─ admin.py
│     ├─ apps.py
│     ├─ models.py
│     ├─ serializers.py
│     ├─ tests.py
│     ├─ urls.py
│     └─ views.py
└─ review_frontend/
   ├─ package-lock.json
   ├─ package.json
   ├─ src/
   │  ├─ App.css
   │  ├─ App.js
   │  ├─ App.test.js
   │  ├─ Pages/
   │  │  ├─ AddReview.js
   │  │  ├─ Home.js
   │  │  ├─ Login.js
   │  │  ├─ Navbar.js
   │  │  ├─ SignUp.js
   │  │  └─ ViewReviews.js
   │  ├─ api/
   │  │  └─ api.js
   │  ├─ index.css
   │  ├─ index.js
   │  ├─ logo.svg
   │  ├─ reportWebVitals.js
   │  └─ setupTests.js
   └─ tailwind.config.js
```

Point Description of Class/Functions

In this section we will describe the point descriptions for all the classes and functions used inside the modules of this project. For this purpose we will go through each component one by one.

Review Backend

In this section we will describe the point description for the classes and functions used for the review backend module. Below is the file structure of the review backend.

```
review_backend/
├─ .env
├─ asgi.py
├─ settings.py
├─ urls.py
└─ wsgi.py
```

1. `asgi.py`

Class: None

- **Description**: This module does not define any classes.

Function: `get_asgi_application()`

- **Description**: This function retrieves the ASGI application instance configured in Django settings. It acts as an entry point for ASGI-compatible web servers to interface with the Django application.

Variable: `application`

- **Description**: This variable holds the ASGI application callable, which can be used by ASGI servers to handle requests. It is set up to use Django's settings specified in the environment variable `DJANGO_SETTINGS_MODULE`.

Module-Level Variable: `os.environ`

- **Description**: This variable is part of the `os` module, used to set environment variables. Here, it is used to define which settings module Django should use for configuration.

2. `settings.py`

Class: None

- **Description**: This module does not define any classes.

Function: `load_dotenv()`

- **Description**: This function loads environment variables from a `.env` file, allowing sensitive information (like database credentials) to be kept out of source control.

Variable: `BASE_DIR`

- **Description**: This variable defines the base directory of the project, used to construct paths for other files and directories within the project.

Variable: `SECRET_KEY`

- **Description**: This variable holds the secret key used for cryptographic signing in Django. It should be kept confidential and not shared in public repositories.

Variable: `DEBUG`

- **Description**: This variable indicates whether the Django application is in debug mode. It should be set to `False` in production.

Variable: `ALLOWED_HOSTS`

- **Description**: This variable is a list of strings representing the host/domain names that the Django site can serve. It's used for security purposes.

Variable: `INSTALLED_APPS`

- **Description**: This variable is a list of applications that are activated in this Django project. It includes both built-in Django applications and custom apps.

Variable: `AUTH_USER_MODEL`

- **Description**: This variable specifies the custom user model to be used for authentication, pointing to the `Client` model in the `auth_review` app.

Variable: `REST_FRAMEWORK`

- **Description**: This variable defines settings related to Django REST Framework, including authentication classes.

Variable: `SIMPLE_JWT`

- **Description**: This variable contains configuration for the Simple JWT library, which manages JSON Web Tokens for authentication.

Variable: `MIDDLEWARE`

- **Description**: This variable is a list of middleware classes that process requests and responses in the Django application.

Variable: `ROOT_URLCONF`

- **Description**: This variable specifies the URL configuration module for the project, defining the URL routing.

Variable: `TEMPLATES`

- **Description**: This variable is a list of settings related to template engines, including the backend to use and context processors.

Variable: `WSGI_APPLICATION`

- **Description**: This variable specifies the WSGI application callable, which is used for serving the Django application.

Variable: `DATABASES`

- **Description**: This variable defines the database configurations, including the engine (Djongo for MongoDB) and connection details.

Variable: `AUTH_PASSWORD_VALIDATORS`

- **Description**: This variable is a list of validators that check the strength of user passwords during registration.

Variable: `LANGUAGE_CODE`

- **Description**: This variable defines the language code for the project, determining the default language for translations.

Variable: `TIME_ZONE`

- **Description**: This variable specifies the time zone for the application, affecting time-related functions.

Variable: `USE_I18N`

- **Description**: This variable indicates whether Django's internationalization system is enabled.

Variable: `USE_TZ`

- **Description**: This variable indicates whether to use timezone-aware datetimes in the application.

Variable: `STATIC_URL`

- **Description**: This variable defines the URL prefix for serving static files.

Variable: `DEFAULT_AUTO_FIELD`

- **Description**: This variable sets the default field type for primary keys to `BigAutoField`, which allows for larger integer values.

3. `urls.py`

Class: None

- **Description**: This module does not define any classes.

Function: `path()`

- **Description**: This function is used to define URL patterns by associating a URL path with a view. It accepts parameters for the URL string, the view function or class, and an optional name for the URL.

Function: `include()`

- **Description**: This function allows including other URL configurations, effectively modularizing URL handling by delegating to other URLconf modules.

Variable: `urlpatterns`

- **Description**: This variable is a list of URL patterns that Django uses to route incoming requests to the appropriate view functions or classes. It includes:
  - `path("admin/", admin.site.urls)`: Routes requests for the admin site.
  - `path("auth/", include("auth_review.urls"))`: Routes requests for authentication-related URLs to the `auth_review` app.
  - `path("service/", include("service.urls"))`: Routes requests for service-related URLs to the `service` app.

4. `wsgi.py`

Class: None

- **Description**: This module does not define any classes.

Variable: `application`

- **Description**: This variable holds the WSGI callable that Django's built-in servers use to communicate with the application. It's initialized by calling `get_wsgi_application()`.

Function: `os.environ.setdefault()`

- **Description**: This function sets the default value for the environment variable `DJANGO_SETTINGS_MODULE`, which specifies the settings module for the Django project.

Function: `get_wsgi_application()`

- **Description**: This function returns the WSGI application callable for the project, which serves as the entry point for WSGI-compatible web servers to serve the Django application.

Auth_review

In this section we will describe the point description for the classes and functions used for the auth review module. Below is the file structure of the review backend.

```
auth_review/
├─ admin.py
├─ apps.py
├─ models.py
├─ serializers.py
├─ tests_models.py
├─ tests_serializers.py
├─ tests_views.py
├─ urls.py
└─ views.py
```

5. admin.py

Description: This module integrates the models from the `auth_review` application into the Django admin interface, enabling administrators to manage user authentication and review data effectively.

Functions and Classes:

- **Admin Registration**:
  - **Purpose**: To register models with the Django admin site.
  - **Functionality**: It enhances the usability of the admin interface for managing authentication and review-related data.

6. apps.py

1. Class: AuthReviewConfig

- **Description**: Configures the `auth_review` application within a Django project.
- **Attributes**:
  - **`default_auto_field`**: Defines the type of field used for auto-created primary keys, set to `BigAutoField`.
  - **`name`**: Specifies the application name (`auth_review`), used in other parts of the Django framework.
- **Purpose**: Handles user authentication, registration, and review management within the `auth_review` app, establishing its configuration in the Django ecosystem.

7. models.py

1. Class: `ClientManager`

This module defines a custom user model and its management for a Django application, implementing the `Client` model and a corresponding `ClientManager`.

- **Description**: Custom manager for the `Client` model to handle user creation.
  - **Method: `create_user(username, password=None)`**
    - **Description**: Creates and returns a regular user with an encrypted password.
    - **Args**:
      - `username (str)`: The username for the user.
      - `password (str, optional)`: The password for the user.
    - **Returns**: `Client`: The created user instance.
    - **Raises**: `TypeError`: If username is None.
  - **Method: `create_superuser(username, password=None)`**
    - **Description**: Creates and returns a superuser with an encrypted password.
    - **Args**:
      - `username (str)`: The username for the superuser.
      - `password (str, optional)`: The password for the superuser.
    - **Returns**: `Client`: The created superuser instance.
    - **Raises**: `TypeError`: If password is None.

2. Class: `Client`

- **Description**: Custom user model allowing for customization of user fields and permissions.
  - **Attributes**:
    - `is_active (bool)`: Indicates whether the account is active.
    - `username (str)`: Unique username for the user, serves as the primary key.
    - `is_admin (bool)`: Indicates if the user has admin privileges.
    - `is_staff (bool)`: Indicates if the user has staff privileges.
  - **Constant**:
    - `USERNAME_FIELD`: The field used for authentication.
  - **Methods**:
    - `__str__()`
      - **Description**: Returns the string representation of the `Client` instance.
      - **Returns**: `str`: The username of the client.

4. serializers.py

1. Class: MyTokenObtainPairSerializer

- This module defines custom serializers for user authentication and registration, handling JWT token generation and user validation.
- Extends `TokenObtainPairSerializer` to customize JWT claims during user authentication.
- **Methods**:
    - `get_token(user)`:
        - **Description**: Generates a JWT for the specified user and adds custom claims (e.g., username).
        - **Args**:
            - `cls`: The class itself.
            - `user`: The user instance.
        - **Returns**: An instance of the token with custom claims.

2. Class: RegisterSerializer

- **Description**: Handles user registration, including validation and user creation.
- **Attributes**:
    - `username` (str): Required username for registration, validated for uniqueness.
    - `password` (str): Required password for the user account, validated for security.
- **Methods**:
    - `create(validated_data)`:
        - **Description**: Creates a new user instance with the provided validated data.
        - **Args**:
            - `validated_data` (dict): Contains username and password.
        - **Returns**: The created user instance.
- **Meta**:
    - **Description**: Provides metadata options for the serializer.
    - **Attributes**:
        - `model`: The associated user model.
        - `fields`: List of field names included in the serialized output (`username`, `password`).

5. Test_models.py

Class: ClientModelTests

Description: Unit tests for the User model, verifying functionality related to user creation, attribute handling, and error cases.

1. `test_create_regular_user`

Description: Tests the creation of a regular user and checks default attributes such as `is_active`, `is_staff`, `is_admin`, and `is_superuser`.

2. `Test_create_superuser`

Description: Tests the creation of a superuser and verifies superuser-specific attributes.

3. `test_create_user_without_username`

Description: Ensures that attempting to create a user without a username raises a `TypeError`.

4. `test_create_superuser_without_password`

Description: Ensures that attempting to create a superuser without a password raises a `TypeError`.

5. `Test_user_activation`

Description: Tests the ability to deactivate a user and verifies that the user's `is_active` status is updated correctly.

6. `test_user_string_representation`

Description: Tests the string representation of the user model to ensure it returns the username.

6. `test_serializers.py`

This module contains unit tests for the serializers in the `auth_review` application, specifically testing user registration and token generation functionality.

1. Class: `RegisterSerializerTest`

Description: A test case for validating the `RegisterSerializer` behavior, ensuring proper handling of user registration.

Method: `test_valid_registration_data`

- **Description**: Validates that a user can be successfully created with valid registration data.

Method: `test_registration_missing_username`

- **Description**: Checks that registration fails when the username field is missing.

Method: `test_registration_missing_password`

- **Description**: Ensures registration fails when the password field is not provided.

Method: `test_registration_weak_password`

- **Description**: Tests that registration fails when the password does not meet the specified strength criteria.

2. Class: `MyTokenObtainPairSerializerTest`

Description: A test case for validating the `MyTokenObtainPairSerializer` behavior, focusing on token generation for users.

Method: `test_get_token_with_valid_user`

- **Description**: Validates that a token can be successfully obtained for an existing user.

Method: (Commented Out) `test_get_token_for_nonexistent_user`

- **Description**: Intended to verify that attempting to get a token for a nonexistent user raises a `ValidationError`. (Currently commented out and not executed in tests.)

7. `test_views.py`

1. Class: `AuthTests`

- **Description**: Contains test cases for user authentication and registration in the `auth_review` application.

Method: `__init__`

- **Description**: Initializes the test class, setting up placeholders for test user and URL variables.

Method: `setUp`

- **Description**: Prepares the test environment by creating a test user and defining URLs for registration and token operations.

Method: `test_register_new_user_success`

- **Description**: Tests that a new user can successfully register and receive a confirmation message.

Method: `test_register_existing_user_error`

- **Description**: Validates that attempting to register with an existing username fails and returns an error message.

Method: `test_token_obtain_pair_success`

- **Description**: Confirms that a valid user can obtain a JWT token with correct credentials.

Method: `test_token_obtain_pair_invalid_credentials`

- **Description**: Ensures that invalid credentials do not allow token generation and return an unauthorized error.

Method: `test_token_refresh_success`

- **Description**: Tests successful token refresh using a valid refresh token.

Method: `test_register_user_invalid_password`

- **Description**: Verifies that registration fails with a weak password, returning an appropriate error message.

Method: `test_token_obtain_inactive_user`

- **Description**: Confirms that an inactive user cannot obtain a JWT token, resulting in an unauthorized error.

Method: `test_token_obtain_with_empty_payload`

- **Description**: Checks that an empty payload sent to the token endpoint results in a bad request error.

Method: `test_get_not_allowed_on_token_endpoint`

- **Description**: Tests that sending a GET request to the token endpoint is not allowed, returning a method not allowed error.

8. urls.py

Module: `auth_review.urls`

This module defines the URL routing for the `auth_review` application, managing user authentication and registration through Django REST Framework.

URL Patterns

- `urlpatterns`:
  - A list of URL patterns that link specific endpoints to corresponding views.
- `path("token/", MyTokenObtainPairView.as_view(), name="token_obtain_pair")`:
  - Endpoint for user authentication that returns a JWT token upon successful login.
- `path("token/refresh/", TokenRefreshView.as_view(), name="token_refresh")`:
  - Endpoint for refreshing an expired JWT token.
- `path("register/", RegisterView.as_view(), name="register")`:
  - Endpoint for user registration, allowing new users to create accounts.
- `format_suffix_patterns(urlpatterns)`:
  - Enables the API to respond to requests in various formats (e.g., JSON, XML).

9. views.py

1. RegisterView (APIView)

- **Purpose**: Handles user registration, ensuring unique usernames.
- **Methods**:
  - **post(request)**:
    - **Description**: Processes registration requests, validates username uniqueness, and creates a new user.
    - **Args**:
      - `request`: The HTTP request containing registration data.
    - **Returns**: A response with registration status and message.

2. MyTokenObtainPairView (TokenObtainPairView)

- **Purpose**: Manages user authentication and generates JWT tokens.

- **Methods**:
  - **get(request)**:
    - **Description**: Handles GET requests, returning an error message indicating that GET requests are not allowed.
    - **Args**:
      - `request`: The HTTP request.
    - **Returns**: A response with an error message.
  - **post(requests)**:
    - **Description**: Authenticates user and generates JWT tokens, providing additional user details.
    - **Args**:
      - `requests`: The HTTP request containing authentication data.
    - **Returns**: A response with authentication status, tokens, and user details.

Service

In this section we will describe the point description for the classes and functions used for the service module. Below is the file structure of the review backend.

```
service/
├─ admin.py
├─ apps.py
├─ models.py
├─ serializers.py
├─ tests.py
├─ urls.py
└─ views.py
```

1. apps.py

Class: `ServiceConfig`

- **Description**: This class configures the `service` application within the Django project. It defines application-specific settings, including the default primary key field type.

Attributes:

- **`default_auto_field`**: (str) Specifies the field type used for auto-created primary keys, set to `BigAutoField`.
- **`name`**: (str) Represents the name of the application, which is used to refer to the app throughout the Django project.

2. models.py

Classes

1. **Reviews**
    - **Description**: Represents a job review submitted by a user.
    - **Attributes**:
        - `department`: The department related to the job.
        - `locations`: Job location, indexed for performance.
        - `job_title`: Title of the job, indexed.
        - `job_description`: Description of the job, indexed.
        - `hourly_pay`: Pay rate for the job.
        - `benefits`: Benefits offered, indexed.
        - `review`: Text of the review, indexed and optional.
        - `rating`: Rating from 1 to 5.
        - `reviewed_by`: User who submitted the review, indexed.
        - `recommendation`: Indicates recommendation status.
    - **Method**:
        - `clean()`: Validates the model fields and raises `ValidationError` for invalid entries.
2. **Vacancies**
    - **Description**: Represents job vacancies available in the application.
    - **Attributes**:
        - `jobTitle`: Title of the job, indexed.
        - `jobDescription`: Description of the job, indexed.
        - `jobLocation`: Location of the job, indexed.
        - `jobPayRate`: Pay rate for the job, indexed.
        - `maxHoursAllowed`: Maximum hours allowed for the job.

Meta Classes

- **Meta (Reviews)**
    - **Description**: Contains options for the `Reviews` model.
    - **Attribute**:
        - `verbose_name_plural`: Display name for multiple instances of this model.
- **Meta (Vacancies)**
    - **Description**: Contains options for the `Vacancies` model.
    - **Attribute**:
        - `verbose_name_plural`: Display name for multiple instances of this model.

3. serializers.py

Classes

1. **ReviewsSerializer**
    - **Description**: Serializes the `Reviews` model for validation and conversion to/from JSON.
    - **Meta**:
        - `model`: Specifies the `Reviews` model.
        - `fields`: Includes all fields of the `Reviews` model.
        - `extra_kwargs`: Enforces that certain fields (`department`, `job_title`, `hourly_pay`, `review`, `rating`) are required, while others are optional.
    - **Method**:
        - `validate(self, attrs)`: Custom validation logic to ensure the rating is between 1 and 5 and that the review is not empty. Raises `ValidationError` if conditions are not met.
2. **VacanciesSerializer**
    - **Description**: Serializes the `Vacancies` model for validation and conversion to/from JSON.
    - **Meta**:
        - `model`: Specifies the `Vacancies` model.
        - `fields`: Includes all fields of the `Vacancies` model.
    - **Method**:
        - `validate(self, attrs)`: Custom validation logic to ensure the job title and description are not empty and that the maximum hours allowed is greater than 0. Raises `ValidationError` if conditions are not met.

4. tests.py

Class: `ReviewsTests`

A test case class that inherits from `TestCase`, designed to test the `Reviews` model in the 'service' application.

Method: `setUp`

Prepares the test data by creating three review instances, ensuring each test runs with isolated data.

Validation Tests (8)

1. **test_department_not_empty**: Validates that the department field cannot be empty.
2. **test_locations_not_empty**: Validates that the locations field cannot be empty.
3. **test_job_title_not_empty**: Validates that the job title field cannot be empty.
4. **test_hourly_pay_not_empty**: Validates that the hourly pay field cannot be empty.
5. **test_benefits_not_empty**: Validates that the benefits field cannot be empty.
6. **test_review_not_empty**: Validates that the review field cannot be empty.
7. **test_rating_not_null**: Validates that the rating cannot be null.
8. **test_invalid_rating_above_max**: Validates that the rating cannot exceed 5.
9. **test_invalid_rating_below_min**: Validates that the rating cannot be less than 1.

Data Integrity and Creation Tests (3)

1. **test_create_review**: Confirms that a review can be created successfully.
2. **test_delete_review**: Ensures that a review can be deleted successfully.
3. **test_review_fields**: Checks that all fields of a review are set correctly.

Filtering/Query Tests (7)

1. **test_filter_by_location**: Tests filtering reviews by location.
2. **test_filter_by_department**: Tests filtering reviews by department.
3. **test_filter_by_hourly_pay**: Tests filtering reviews by hourly pay.
4. **test_filter_by_rating**: Tests filtering reviews by rating.
5. **test_filter_by_benefits**: Tests filtering reviews by benefits.
6. **test_filter_by_job_title**: Tests filtering reviews by job title.
7. **test_filter_by_recommendation**: Tests filtering reviews by recommendation.

Edge Cases Tests (5)

1. **test_empty_string_department**: Validates that the department cannot be an empty string.
2. **test_max_length_exceeded**: Ensures that a field cannot exceed its maximum length.
3. **test_review_with_special_characters**: Tests that special characters are handled correctly in reviews.
4. **test_review_creation_without_recommendation**: Confirms that a review can be created without a recommendation value.
5. **test_review_creation_with_zero_rating**: Validates that a review cannot be created with a rating of zero.

Method: `tearDownClass`

Cleans up the database by deleting all review instances after tests have completed.

5. urls.py

URL Configuration Overview

- **Purpose**: This module sets up the URL patterns for the 'service' app, mapping URLs to their corresponding views using Django's URL routing system.

Components

1. **Imports**: Includes necessary modules from Django and Django REST framework.
2. **Router**: Uses `DefaultRouter` to register two view sets:
    - `ReviewsViewSet` for handling reviews.
    - `VacanciesViewSet` for handling job vacancies.
3. **URL Patterns**:
    - The base path includes all routes defined by the router.
    - A separate path for filtering reviews (`FilterReviewsView`) is defined.

This setup helps in organizing and managing the API endpoints efficiently.

5. views.py

Views Overview

- **Purpose**: This module manages HTTP requests for the 'service' app, interacting with models and serializers.

Key Classes

1. **ReviewsViewSet**:
    - Inherits from `ModelViewSet`.
    - Allows CRUD operations on `Reviews` with authentication.
    - Overrides `create` method to add `reviewed_by` field before saving.
2. **FilterReviewsView**:
    - Inherits from `ListAPIView`.
    - Filters reviews based on query parameters (department, location, job title, and ratings).
3. **VacanciesViewSet**:
    - Similar to `ReviewsViewSet`, manages CRUD operations for `Vacancies`.

This structure effectively handles review and vacancy management with user authentication and query filtering.

Review_frontend

In this section we will describe the point description for the classes and functions used for the service module. Below is the file structure of the review backend.

```
review_frontend/
├─ package-lock.json
├─ package.json
├─ src/
│  ├─ App.css
│  ├─ App.js
│  ├─ App.test.js
│  ├─ Pages/
│  │  ├─ AddReview.js
│  │  ├─ Home.js
│  │  ├─ Login.js
│  │  ├─ Navbar.js
│  │  ├─ SignUp.js
│  │  └─ ViewReviews.js
│  ├─ api/
│  │  └─ api.js
```

1. api.js

1. **unprotected_api_call**: Handles requests to endpoints that do not require authentication, allowing for GET and POST requests.
2. **protected_api_call**: Similar to the first, but includes an authorization header with a Bearer token retrieved from local storage, enabling access to protected endpoints.

2. addReview.js

Class: `AddReview`

- **Purpose**: Represents a component for adding a review, managing form state, and submitting review data to the backend.

State: `formData`

- **Type**: Object
- **Description**: Holds the input values for the review form, including job title, department, locations, job description, hourly pay, benefits, rating, recommendation, and review text.

Method: `handleChange`

- **Parameters**: `e` (event)
- **Purpose**: Updates the `formData` state based on user input in the form fields.

Method: `handleSubmit`

- **Parameters**: `e` (event)
- **Purpose**: Prevents the default form submission, calls the `protected_api_call` function to submit the review data, and alerts the user upon success or failure of the submission.

Method: `render`

- **Purpose**: Renders the component's UI, including a styled background and a form for inputting review details. It also includes a navigation bar at the top.

Function: `AddReviewWithNavigate`

- **Parameters**: `props`
- **Purpose**: Wraps the `AddReview` component with the `useNavigate` hook from React Router to enable navigation after submitting the review.

3. home.js

Class: `Home`

- **Purpose**: Renders the main home page of the application, providing options to add or view reviews based on user authentication status.

Method: `add_review`

- **Purpose**: Checks if the user is logged in (by retrieving the login status from local storage). If logged in, navigates to the add review page; otherwise, alerts the user to log in.

Method: `view_reviews`

- **Purpose**: Similar to `add_review`, this method checks the user's login status. If the user is logged in, it navigates to the view reviews page; if not, it alerts the user to log in.

Method: `render`

- **Purpose**: Renders the component's UI, including a styled background image and buttons for adding and viewing reviews, along with the navigation bar.

Function: `HomeWithNavigate`

- **Purpose**: A higher-order component that wraps the `Home` component, providing it with the `navigate` function from the React Router for programmatic navigation.

Export Statement

- **Purpose**: Exports the `HomeWithNavigate` component as the default export for use in other parts of the application.

capabilities.

4. login.js

Class: `Login`

- **Description**: Manages the login functionality for the application, including form state, input handling, and submission logic.

Method: `handleChange`

- **Description**: Updates the component state with the values entered in the username and password fields. It uses the event object to identify which input field has changed.

Method: `handleSubmit`

- **Description**: Prevents the default form submission, retrieves the form data from the state, and makes an API call to the login endpoint. If successful, it stores the user data in local storage and navigates to the home page; otherwise, it alerts the user of invalid credentials.

Function: `LoginWithNavigate`

- **Description**: A wrapper component that uses the `useNavigate` hook from `react-router-dom` to provide navigation capabilities to the `Login` component.

Render Method

- **Description**: Constructs the UI for the login page, including the background image, form fields for username and password, and the submit button. It also integrates the `NavBar` component for navigation.

5. navbar.js

Component: `NavBar`

- **Description**: A navigation bar component that provides links for user actions, dynamically adjusting based on the user's authentication status.

Function: `NavBar`

- **Parameters**: `props` - Contains properties passed to the component, including navigation methods.
- **Description**: Initializes the navigation menu and handles rendering of navigation items based on user login status.

Variable: `pages`

- **Description**: An array of navigation pages that adjusts based on whether the user is logged in or not.

Function: `handleOpenNavMenu`

- **Parameters**: `event` - The event triggered by opening the navigation menu.
- **Description**: Sets the state for opening the mobile navigation menu.

Function: `handleCloseNavMenu`

- **Parameters**: `e` - The event triggered by closing the navigation menu.
- **Description**: Placeholder function for handling close actions of the mobile menu.

Function: `goto`

- **Parameters**: `page` - The name of the page to navigate to.
- **Description**: Handles navigation to different routes based on the selected page.

Return Statement

- **Description**: Renders the AppBar with the title, navigation menu, and buttons for different routes, ensuring responsive design for mobile and desktop views.

6. signUp.js

Class: `Signup`

- **Purpose**: Manages the user registration process, including handling form data and API calls to register a new user.

State:

- **formData**: An object containing user input for `username` and `password`.

Method: `handleChange`

- **Description**: Updates the `formData` state with user input from the registration form fields.

Method: `handleSubmit`

- **Description**: Handles form submission, preventing the default behavior, calling the API to register the user, and providing feedback based on the API response.

Method: `render`

- **Description**: Renders the signup form, including a background image, form fields for username and password, and a submit button.

Function: `SignupWithNavigate`

- **Purpose**: Wraps the `Signup` component with `useNavigate` to provide navigation functionality, allowing redirection after successful registration.

Export Statement:

- **`export default SignupWithNavigate`**: Exports the `SignupWithNavigate` function as the default export of the module.

6. viewReviews.js

Class: `Signup`

- **Description**: Represents a signup form for user registration. It manages the form state and handles user input and submission.

State

- `formData`: Holds the username and password entered by the user.

Method: `handleChange`

- **Description**: Updates the `formData` state when the user types into the input fields.

Method: `handleSubmit`

- **Description**: Prevents the default form submission behavior, calls the API to register the user, and handles the response (success or error).

Method: `render`

- **Description**: Renders the signup form and background style, including input fields for username and password, along with a submit button.

Function: `SignupWithNavigate`

- **Description**: A higher-order component that wraps the `Signup` component with the `useNavigate` hook, providing navigation capabilities to the `Signup` component.