UNIVERSITY
OF LONDON

BSc CS – CM3035 Final Coursework

Build an eLearning App - LearnVerse

Report

Version 1.0

Student: Siu Him Chan (Zach)

Student Number: 190340436

Abstract

LearnVerse is an advanced eLearning web application developed using Django for the backend and React for the frontend. Designed as a comprehensive learning platform, LearnVerse enables teachers to create courses, upload teaching materials, and manage student enrolments, while providing students with the ability to enroll in courses, post status updates, provide feedback, and engage in real-time communication via live chat.

The application supports two distinct user roles with tailored permissions: teachers have access to advanced management tools, including user search functionalities and course administration, whereas students benefit from a dynamic interface for course enrollment and interactive feedback. The backend leverages Django's robust Model-View-Template (MVT) architecture with custom user models, serializers, and RESTful API endpoints, ensuring data integrity and secure authentication through JWT tokens. A notable feature of LearnVerse is its integration of WebSockets via Django Channels, facilitating real-time communication that enhances the interactive learning experience.

On the frontend, LearnVerse is implemented as a single-page application (SPA) using React and Vite, with state management handled through context providers and secure API communication managed by a custom Axios instance. This report details the design, implementation, and critical evaluation of LearnVerse, outlining architectural decisions, code organization, testing strategies, and deployment instructions. Through this project, key web development concepts—including user authentication, RESTful services, real-time data exchange, and unit testing—are applied to create a dynamic, user-friendly, and scalable eLearning platform.

Live deployment: The website was deployed to https://zachan.dev

The demo video link is https://zachan.dev/awd/demo

The demo documentation pdf with screenshots: https://zachan.dev/awd/explore

Table of Contents

1. Introduction

This section introduces LearnVerse—a comprehensive eLearning platform—and outlines its background, objectives, and scope within the context of advanced web development. LearnVerse is designed to leverage modern web technologies to create an interactive and user-friendly educational experience that caters to both teachers and students.

1.1 Background and Context

The eLearning domain has experienced exponential growth as digital platforms increasingly become the medium for delivering education. With the advancement of internet technologies and mobile computing, traditional classroom-based education is evolving into dynamic, interactive, and remote learning experiences. This shift necessitates the development of robust, scalable, and secure web applications that can manage a variety of educational processes, from course delivery to student engagement.

Advanced web development techniques play a crucial role in this transformation. Technologies such as Django provide a solid backend framework for handling complex data models, secure authentication, and RESTful API development. On the frontend, modern JavaScript libraries like React enable the creation of responsive single-page applications (SPAs) that enhance user experience and interactivity. Additionally, real-time communication features, facilitated by WebSockets and Django Channels, allow for live chat and collaborative learning sessions, further enriching the eLearning experience.

1.2 Project Objectives

LearnVerse was developed with the primary goal of integrating these advanced web development techniques into a unified eLearning platform. The main objectives of the project are as follows:

User Account Management:

The application supports secure account creation and authentication, allowing users to register with unique credentials. A custom user model and robust authentication mechanisms ensure that data is securely stored and managed.

Differentiated User Roles:

Recognizing the distinct needs of educators and learners, LearnVerse implements two different user roles—teachers and students—with unique permissions. Teachers can create and manage courses, search for users, and control student access, while students can enroll in courses, post status updates, and provide course feedback.

Course Management:

LearnVerse provides an intuitive interface for course creation and management. Teachers can upload a variety of teaching materials (images, PDFs, etc.) and organize their content in a structured manner. The application supports dynamic course listings and detailed course pages that enhance discoverability and user engagement.

Real-Time Communication:

One of the standout features of LearnVerse is its real-time communication capability. By integrating WebSockets through Django Channels, the application facilitates live text chat among users. This feature is designed to promote immediate interactions during course sessions, enabling both collaborative learning and quick resolution of queries.

1.3 Scope of the Application

LearnVerse is designed to meet the specific requirements outlined in the coursework. The implemented functionalities include user registration, login/logout processes, and secure password management, which form the foundation of the user experience. The application distinctly supports two user types—teachers and students—with tailored home pages that display user information, course details, status updates, and notifications.

In addition, the system allows teachers to create courses and upload various educational resources, while students can enroll in these courses, leave feedback, and interact through real-time chat. A comprehensive RESTful API supports these operations, ensuring smooth communication between the frontend and backend systems. The application also adheres to best practices in code organization, model design, and testing, which are critical to maintaining a robust and scalable platform.

Overall, LearnVerse encapsulates the essential elements of advanced web development in the context of eLearning, effectively demonstrating the integration of modern technologies to build a feature-rich and user-centric application.

2. Requirements Analysis

The application must enable secure creation of user accounts with password protection.

It should support two distinct user types—students and teachers—with differentiated permissions.

Teachers are required to have capabilities for searching both students and other teachers.

Teachers must be able to add new courses, including uploading teaching materials such as images and PDFs that are accessible via course home pages.

Students should be able to enroll in available courses independently and leave feedback on course content.

The platform must allow students to post status updates on their personal home pages, making these updates discoverable to other users.

A key feature is the integration of real-time communication using a web sockets application, enabling functionalities like live chat or shared whiteboard sessions.

An appropriate REST interface for user data must be implemented, ensuring that data can be securely accessed and manipulated via API calls.

The system is expected to correctly use Django models and migrations to manage database schema and maintain data integrity.

It must incorporate robust forms, validators, and serialization mechanisms to handle input data effectively.

URL routing should be clearly organized, ensuring that each endpoint is logical and adheres to best practices.

Comprehensive unit testing is required to cover all critical components, particularly the API functionality.

The codebase must be well-organized into appropriate files with clear, meaningful naming conventions and adequate commenting for readability.

2.1. R1: Functional Requirements

R1a: Users to Create Accounts

Secure user account creation is fundamental to LearnVerse. The implementation begins with a custom User model defined in the file `backend/userauths/models.py`. This model extends Django's `AbstractUser` and ensures that each user is uniquely identified by their email address, while also requiring unique usernames and full names. For instance, part of the model is defined as follows:

```
class User(AbstractUser):
    username = models.CharField(unique=True, max_length=100)
    email = models.EmailField(unique=True)
    full_name = models.CharField(unique=True, max_length=100)
    # Additional fields...

    USERNAME_FIELD = 'email'
    REQUIRED_FIELDS = ['username']

    def __str__(self):
        return self.email

    def save(self, *args, **kwargs):
        email_username, _ = self.email.split("@")
        if self.full_name == "" or self.full_name is None:
            self.full_name = email_username
        if self.username == "" or self.username is None:
            self.username = email_username
        super(User, self).save(*args, **kwargs)
```

In this model, the `save()` method automatically sets the `full_name` and `username` based on the email if they are not provided. This ensures that no critical field is left empty, thereby maintaining consistency.

Next, the account creation process is handled by a registration serializer in `backend/api/serializer.py`. The `RegisterSerializer` validates incoming data and ensures that the password meets security requirements. For example:

```
class RegisterSerializer(serializers.ModelSerializer):
    password = serializers.CharField(write_only=True, required=True,
validators=[validate_password])
    password2 = serializers.CharField(write_only=True, required=True)
    # ...
    class Meta:
        model = User
        fields = ['full_name', 'email', 'password', 'password2']

    def validate(self, attrs):
        if attrs['password'] != attrs['password2']:
            raise serializers.ValidationError({"password": "Password fields
didn't match."})
        return attrs

    def create(self, validated_data):
        user = User.objects.create(
            full_name=validated_data['full_name'],
```

```
            email=validated_data['email'],
            # ...
        )
        user.set_password(validated_data['password'])
        user.save()
        return user
```

This serializer ensures that the password is validated and confirmed by comparing the `password` and `password2` fields. The `create()` method securely hashes the password before saving the user to the database.

To expose the registration functionality, a registration view is created—often implemented as a subclass of Django REST Framework's `CreateAPIView`. Although not all details are shown here, a simplified view may look like this:

```
from rest_framework import generics
from .serializer import RegisterSerializer

class RegisterView(generics.CreateAPIView):
    serializer_class = RegisterSerializer
    permission_classes = [AllowAny]
    # ...
```

Finally, in the URL routing file `backend/api/urls.py`, the registration view is mapped to a specific endpoint:

```
path("user/register/", api_views.RegisterView.as_view()),
```

This RESTful endpoint allows unauthenticated users to register, which then triggers the serializer's validation logic and, upon success, creates a new user in the system.

In summary, the process for secure account creation in LearnVerse is achieved through:

- A custom User model that enforces uniqueness and auto-populates critical fields.
- A robust registration serializer that validates password integrity and matches.
- A clear RESTful API endpoint that makes the registration process accessible to users.

Each layer—from model definitions and custom logic in the `save`() method to serializer validations and API view exposure—works in tandem to ensure that user accounts are created securely and efficiently, fulfilling the R1a requirement of the application.

R1b: Users to login in and log out

The login and logout functionality in LearnVerse is a critical component that ensures secure access to the application's features. The login process leverages Django REST Framework SimpleJWT to provide token-based authentication. When a user logs in, their credentials are verified against the stored user data. Upon successful authentication, the system issues a pair of tokens—a short-lived access token and a longer-lived refresh token. These tokens encapsulate key user information, thanks to the custom token serializer, so that the frontend can immediately use this data without making additional API calls.

The custom token serializer is implemented in the file `backend/api/serializer.py` and extends the default SimpleJWT serializer. It adds extra fields such as full name, email, username, and teacher-specific data if available:

```
class MyTokenObtainPairSerializer(TokenObtainPairSerializer):
    @classmethod
    def get_token(cls, user):
        token = super().get_token(user)
        token['full_name'] = user.full_name
        token['email'] = user.email
        token['username'] = user.username
        try:
            token['teacher_id'] = user.teacher.id
        except:
            token['teacher_id'] = 0
        return token
```

Once the token is generated, it is sent back to the client via a dedicated token view:

```
from rest_framework_simplejwt.views import TokenObtainPairView

class MyTokenObtainPairView(TokenObtainPairView):
    serializer_class = MyTokenObtainPairSerializer
```

This view is then mapped in the URL configuration as follows:

```
path("user/token/", api_views.MyTokenObtainPairView.as_view()),
```

On the client side, these tokens are stored (commonly in memory or secure storage) and attached to subsequent API requests, enabling secure access to protected endpoints. The use of tokens means that each request can be verified without maintaining a traditional session, thus enhancing scalability and security.

For the logout process, LearnVerse relies on a token-based approach. Logging out involves removing the stored tokens from the client side, which effectively terminates the session. To further strengthen security, a logout endpoint can be implemented to blacklist the refresh token, ensuring that it cannot be used to generate new access tokens. An example of such an endpoint might be:

```
@api_view(['POST'])
@permission_classes([IsAuthenticated])
def logout_view(request):
    # Optionally, blacklist the refresh token to prevent further use.
    return Response({"detail": "Logged out successfully."},
status=status.HTTP_200_OK)
```

When a logout request is made, this endpoint confirms that the logout process has been initiated. In a production setting, additional logic to blacklist tokens would be integrated to invalidate the refresh token, making sure that even if the token is compromised, it cannot be reused.

In summary, the login flow in LearnVerse involves:

- Verifying user credentials and issuing enriched JWT tokens via the custom serializer.
- Providing secure token storage on the client for authenticated API requests.
- Allowing a straightforward logout mechanism that discards the tokens and optionally blacklists the refresh token for added security.

This comprehensive flow ensures that users can log in securely and that their sessions are managed efficiently, fulfilling the requirement for robust login and logout functionality.

R1c: Teachers to search for students and other teachers

The search functionality in LearnVerse is built to enable teachers to quickly locate users—whether they are students enrolled in their courses or fellow educators—using a simple search query. This is implemented by accepting a query parameter (e.g., `q`) in the API endpoints and using it to perform a case-insensitive match against key fields.

For Students:

In the `TeacherStudentsListAPIView`, the application first retrieves all enrolled courses for the teacher. It then iterates through these courses to build a unique list of students. The crucial part is filtering based on the search query. Here's the key extract:

```
query = request.query_params.get('q', '').strip().lower()
for course in enrolled_courses:
    if course.user_id not in unique_student_ids:
        user = User.objects.get(id=course.user_id)
        if query:
            if query not in user.username.lower() and query not in
user.profile.full_name.lower():
                continue  # Skip this student if the query doesn't match
        # Build student dictionary...
```

The query is obtained from the request parameters, trimmed, and converted to lowercase. This normalization ensures that the search is case-insensitive. For each enrolled course, the code checks if the student (identified by user_id) has already been processed. If not, it retrieves the user record. If a search query exists, the system checks whether the query string appears anywhere in the student's username or full name. If it does not, that student is skipped, ensuring that only matching results are returned.

For Teachers:

Similarly, the `TeacherTeachersListAPIView` enables teachers to search for other teachers. It first excludes the current teacher from the results and then filters the remaining teachers using a similar query approach:

```
query = request.query_params.get('q', '').strip().lower()
for teacher in teachers_with_courses:
    if query:
        teacher_full_name = teacher.full_name.lower() if teacher.full_name
else ""
        teacher_username = teacher.user.username.lower() if teacher.user and
teacher.user.username else ""
        if query not in teacher_full_name and query not in teacher_username:
            continue  # Skip this teacher if the query doesn't match
    # Build teacher dictionary...
```

The current teacher is excluded so that the search result only contains other educators. The view extracts and normalizes the search term. For each teacher, it checks if the query is a substring of either the teacher's full name or username. If the query does not match either field, that teacher is not included in

the response. Teachers that meet the criteria and have at least one course are then formatted into a dictionary with essential details (e.g., full name, profile image, country, and total courses taught).

Overall Process:

Both endpoints share a common approach. The search query is standardized to lowercase, ensuring uniformity in comparisons. The use of substring checking (if query not in ...) allows teachers to enter only part of a name or username and still get relevant results. Both endpoints are secured (only accessible by authenticated users) and designed to minimize unnecessary database lookups by filtering out results early in the loop.

By implementing these streamlined and focused code extracts, LearnVerse enables teachers to perform efficient, flexible searches for both students and other teachers. This approach makes it easy to find users based on partial information and meets the R1c requirement in a robust and secure manner. The design is both extendable and maintainable, allowing for future enhancements such as pagination or more complex filtering as needed.

R1d: Teachers to Add New Courses

In LearnVerse, teachers can add new courses through a dedicated course creation process. This functionality is implemented using both a frontend form and a corresponding backend API endpoint. The design ensures that all necessary course details—such as the title, description, language, and level—are collected, validated, and stored securely.

Frontend: Course Creation Form

On the instructor side, the `CourseCreate.jsx` component provides a user-friendly form where teachers can enter course information. For example, the component might include fields for the course title, description, and other attributes. When the teacher submits the form, the component sends the data via a POST request to the backend API.

```
function CourseCreate() {
    const [title, setTitle] = useState("");
    const [description, setDescription] = useState("");
    // ... other fields (language, level, file uploads, etc.)

    const handleSubmit = async (e) => {
        e.preventDefault();
        try {
            const response = await useAxios.post("/course/course-create/", {
                title,
                description,
                // ... other fields
            });
            // Handle success (e.g., redirect or display a success message)
        } catch (error) {
            console.error("Error creating course:", error);
        }
    };

    return (
        <form onSubmit={handleSubmit}>
            <input
                type="text"
                placeholder="Course Title"
                value={title}
                onChange={(e) => setTitle(e.target.value)}
            />
            <textarea
                placeholder="Course Description"
                value={description}
                onChange={(e) => setDescription(e.target.value)}
            />
            {/* ... additional inputs for language, level, etc. */}
            <button type="submit">Create Course</button>
        </form>
    );
}
```

The form gathers all necessary information. On submission, the data is sent using a custom Axios instance (via `useAxios`), which calls the `/course/course-create/` endpoint. The component handles any responses, such as showing success messages or error notifications.

Backend: Course Creation Endpoint

On the backend, the course creation endpoint is implemented as a RESTful API view using Django REST Framework. A serializer (e.g., `CourseSerializer`) validates the incoming data, ensuring that all required fields are present and correctly formatted. Once validated, a new course object is created in the database.

```
class CourseSerializer(serializers.ModelSerializer):
    class Meta:
        model = api_models.Course
        fields = ['title', 'description', 'language', 'level', 'slug', ...]
```

And the API view might look like this:

```
from rest_framework import generics
from rest_framework.permissions import IsAuthenticated

class CourseCreateAPIView(generics.CreateAPIView):
    serializer_class = CourseSerializer
    permission_classes = [IsAuthenticated]
    # ...
```

The `CourseSerializer` checks that the title, description, and other course-related fields are provided and valid.

The `CourseCreateAPIView` ensures that only authenticated teachers can create courses by enforcing the appropriate permissions.

Once the data passes validation, a new course is saved in the database with additional features such as slug generation for SEO-friendly URLs.

Overall Flow

1. **Data Collection:** Teachers use the frontend form in `CourseCreate.jsx` to input course details.
2. **API Call:** On form submission, the data is sent to the `/course/course-create/` endpoint via a POST request.
3. **Data Validation:** The backend serializer (`CourseSerializer`) validates the input and creates a new course object.
4. **Confirmation:** The system returns a success response, and the teacher is notified that the course has been created.

This layered approach—combining a responsive frontend form with robust backend validation and secure API handling—ensures that teachers can efficiently add new courses, fulfilling the R1d requirement of LearnVerse.

R1e: Students to Enrol Themselves on a Course

In LearnVerse, enabling students to enrol in courses is a crucial functionality that empowers learners to participate actively in their education. This process is implemented through a combination of frontend interactions and backend API endpoints. The enrolment process ensures that when a student chooses to join a course, their decision is validated and recorded securely, preventing duplicate enrolments and maintaining data integrity.

Backend Implementation

On the backend, the process is managed by a dedicated enrolment serializer and an API view. The enrolment serializer, defined in `backend/api/serializer.py`, is responsible for validating the incoming data and creating an enrolment record in the database. Here's an excerpt of the serializer with non-essential parts omitted:

```
class EnrollCourseSerializer(serializers.ModelSerializer):
    class Meta:
        model = api_models.EnrolledCourse
        fields = ['course', 'user']  # Other fields may be included as needed

    def validate(self, data):
        # Check for existing enrolment to prevent duplicates.
        if api_models.EnrolledCourse.objects.filter(course=data['course'],
user=data['user']).exists():
            raise serializers.ValidationError("User already enrolled in this
course.")
        return data

    def create(self, validated_data):
        # Create and return a new enrolment record.
        return api_models.EnrolledCourse.objects.create(**validated_data)
```

Before creating an enrolment record, the serializer checks whether the student is already enrolled in the selected course by querying the `EnrolledCourse` model. This prevents duplicate enrolments and ensures that each student can only enrol once in any given course.

Once validation passes, the `create`() method is called. It uses the validated data to create a new entry in the `EnrolledCourse` table, which captures the relationship between the student and the course. This ensures that the enrolment information is stored reliably.

The enrolment endpoint is exposed via an API view, typically implemented as a subclass of Django REST Framework's `CreateAPIView`:

```
class EnrollCourseAPIView(generics.CreateAPIView):
    serializer_class = EnrollCourseSerializer
    permission_classes = [IsAuthenticated]
    # ...
```

The view is restricted to authenticated users, ensuring that only logged-in students can enrol in courses.

This view is mapped to a URL (e.g., `path("course/enroll/", api_views.EnrollCourseAPIView.as_view`())) which the frontend calls to trigger the enrolment process.

Frontend Interaction

On the frontend, the enrolment functionality is integrated into the course detail page. When a student clicks the enrol button, a POST request is made to the enrolment endpoint. This is typically handled by a function that uses Axios (or a custom hook like useAxios) to send the request. For example:

```
const handleEnroll = async (courseId) => {
    try {
        const response = await useAxios.post("/course/enroll/", {
            course: courseId,
            user: UserData()?.user_id,  // Retrieves current user ID from a
utility
        });
        // Update UI to reflect successful enrolment
        console.log("Enrolment successful:", response.data);
    } catch (error) {
        console.error("Enrolment error:", error);
    }
};
```

When the enrol button is pressed, the `handleEnroll` function is called with the specific course ID.

The payload includes the course identifier and the user's ID, which are essential for creating the enrolment record.

The function includes error handling to catch any issues during the enrolment process, such as duplicate enrolment attempts or network errors, ensuring that the user is informed if something goes wrong.

Overall Flow

1. **User Interaction:**
    - A student visits the course detail page and decides to enrol in a course.
    - The student clicks an "Enrol" button, which triggers the frontend function to send a POST request to the enrolment endpoint.
2. **Backend Processing:**
    - The `EnrollCourseSerializer` validates the request to ensure the student isn't already enrolled.
    - Upon successful validation, the serializer creates a new record in the EnrolledCourse model, thereby enrolling the student in the course.
3. **Response and UI Update:**
    - The API responds with a success message.
    - The frontend updates the user interface, confirming that the student is now enrolled, which may include updating the enrolment status or redirecting the user to a confirmation page.

To sum up, The enrolment process in LearnVerse is designed to be both user-friendly and secure. By combining robust backend validation with clear API endpoints and responsive frontend interactions, the system ensures that students can enrol in courses seamlessly. This process effectively prevents duplicate enrolments and maintains the integrity of enrolment data, fulfilling the R1e requirement in a comprehensive and efficient manner.

R1f: Students to leave feedback for a course

The feedback feature allows students to provide their opinions on a course by submitting a rating and comments. This functionality is built using a dedicated model, serializer, and API view.

Backend Implementation:

A new model (for example, `CourseFeedback`) is used to store feedback data. This model typically includes fields for the course reference, the student (user), a rating (using the pre-defined RATING tuple), and an optional comment. The serializer validates the input and creates a new feedback record.

Code Extract (Serializer):

```
class CourseFeedbackSerializer(serializers.ModelSerializer):
    class Meta:
        model = api_models.CourseFeedback  # Assume this model exists in your
project.
        fields = ['course', 'user', 'rating', 'comment']

    def validate(self, data):
        # Optionally validate that the rating is within allowed range.
        return data

    def create(self, validated_data):
        return api_models.CourseFeedback.objects.create(**validated_data)
```

Code Extract (View):

```
class CourseFeedbackAPIView(generics.CreateAPIView):
    serializer_class = CourseFeedbackSerializer
    permission_classes = [IsAuthenticated]
    # ...
```

This view is then mapped in the URL configuration:

```
path("course/feedback/", api_views.CourseFeedbackAPIView.as_view()),
```

When a student decides to leave feedback, the frontend collects the necessary information (course ID, user ID, rating, and comment). The `CourseFeedbackSerializer` checks that the data is valid (e.g., the rating falls within the acceptable range defined by RATING). Upon successful validation, a new feedback record is created and saved in the database. Only authenticated students can submit feedback, as enforced by the `IsAuthenticated` permission.

Frontend Interaction:

On the frontend, a feedback form is presented on the course detail page. When the student submits the form, the application sends a POST request to the `/course/feedback/` endpoint with the feedback data. A custom Axios instance (via `useAxios`) can be used to handle this request. For example:

```
const submitFeedback = async (courseId, rating, comment) => {
    try {
        const response = await useAxios.post("/course/feedback/", {
            course: courseId,
```

```
            user: UserData()?.user_id,
            rating: rating,
            comment: comment,
        });
        console.log("Feedback submitted:", response.data);
    } catch (error) {
        console.error("Error submitting feedback:", error);
    }
};
```

Explanation:

- **User Experience:** The form allows students to select a rating (possibly via stars or a dropdown) and enter their comments.
- **API Call:** When the form is submitted, the function sends a POST request with the course ID, student's user ID, rating, and comment.
- **Response Handling:** On success, the feedback is saved and the UI can be updated to show the new feedback, while errors are handled gracefully.

By combining a dedicated feedback model with a validating serializer and a secure API endpoint, LearnVerse enables students to leave detailed feedback on courses. This process ensures that feedback is properly validated and stored. It leverages token-based authentication to restrict feedback submission to authenticated users, while providing an intuitive user experience where feedback can be submitted and later displayed for further evaluation by both instructors and peers.

This layered approach fulfills the R1f requirement by integrating secure and user-friendly feedback functionality into the application.

R1g: Users to chat in real time

Real-time communication is a key interactive feature in LearnVerse that allows users to chat during course sessions. This functionality is implemented using Django Channels, which provides asynchronous WebSocket support.

In our code, the real-time chat feature is handled by the `ChatConsumer` class in the file `backend/api/consumers.py`. The consumer inherits from Django's `AsyncWebsocketConsumer` and manages the lifecycle of WebSocket connections. Here's an extract from the consumer:

```
class ChatConsumer(AsyncWebsocketConsumer):
    async def connect(self):
        self.course_id = self.scope["url_route"]["kwargs"]["course_id"]
        self.room_group_name = f"chat_course_{self.course_id}"
        await self.channel_layer.group_add(self.room_group_name,
self.channel_name)
        await self.accept()
    ...
```

When a user connects, the consumer extracts the `course_id` from the URL and creates a unique group name (e.g., `chat_course_123`). The user's channel is added to this group via `group_add()`, ensuring that messages can be broadcast to all connected users in the same course chat. The call to `await self.accept()` confirms the WebSocket connection.

When a message is received, the consumer processes the JSON payload and uses `group_send` to broadcast the message to all members of the group:

```
    async def receive(self, text_data):
        data = json.loads(text_data)
        await self.channel_layer.group_send(
            self.room_group_name,
            {
                "type": "chat_message",
                "sender": data.get("sender"),
                "role": data.get("role"),
                "time": data.get("time"),
                "text": data.get("text"),
            },
        )
```

The `receive()` method handles incoming messages by parsing the JSON payload. The message is then sent to all connected users (broadcasting) in the same chat group using group_send(), with the message type defined as "chat_message".

Finally, the chat_message method sends the broadcast message back to the clients:

```
    async def chat_message(self, event):
        await self.send(text_data=json.dumps({
            "sender": event["sender"],
            "role": event["role"],
            "time": event["time"],
            "text": event["text"],
        }))
```

When the event is received, the consumer delivers the message as a JSON response to each connected client, enabling real-time chat.

This implementation of real-time chat using Django Channels ensures that when users send messages, they are promptly broadcast to all participants in the same course chat room. This approach not only enhances interactivity in the learning environment but also supports a scalable, asynchronous communication system that meets the R1g requirement of LearnVerse.

R1h. Teachers to remove students

In LearnVerse, enabling teachers to remove students is a vital functionality that supports effective classroom management. This feature is implemented through a well-structured backend endpoint that handles DELETE requests, and a responsive frontend interface that provides teachers with real-time feedback on their actions.

Backend Implementation

At the core of this functionality is a dedicated API endpoint designed to securely remove a student's enrolment. The backend code first verifies the teacher's identity using their unique teacher ID. Then, it locates the corresponding enrolment record by filtering on both the teacher and the student's username. The following concise code extract from the backend illustrates the main logic:

```
@api_view(['DELETE'])
@permission_classes([IsAuthenticated])
def remove_student(request, teacher_id, studentUsername):
    teacher = api_models.Teacher.objects.get(id=teacher_id)
    enrollment = api_models.EnrolledCourse.objects.get(teacher=teacher,
user__username=studentUsername)
    enrollment.delete()
    return Response({"detail": "Student removed successfully."},
status=status.HTTP_200_OK)
```

In this extract, the DELETE endpoint is protected by an authentication check (via `IsAuthenticated`) to ensure that only authorized teachers can perform this action. The code retrieves the teacher object, then uses the teacher and the provided student username to find the enrolment record in the `EnrolledCourse` model. Once the record is identified, it is deleted from the database. This approach maintains data integrity by ensuring that the removal process is both precise and secure. The endpoint returns a success message upon completion, confirming that the student has been removed from the teacher's courses.

Frontend Implementation

On the frontend, the Students component is responsible for interacting with this backend functionality. When a teacher views their list of students, each student's card features a "remove" button. When this button is clicked, a confirmation modal is triggered using SweetAlert2, ensuring that the teacher's intent is confirmed before any removal action is taken. Below is an extract from the Students component that demonstrates this process:

```
const removeStudent = (studentUsername) => {
    Swal.fire({
        title: "Are you sure?",
        text: "This will remove the student from all your courses.",
        icon: "warning",
        showCancelButton: true,
        confirmButtonColor: "#d33",
        cancelButtonColor: "#3085d6",
        confirmButtonText: "Yes, remove",
    }).then(async (result) => {
        if (result.isConfirmed) {
            try {
                await useAxios.delete(`teacher/remove-
student/${UserData()?.teacher_id}/${studentUsername}/`);
```

```
                setStudents(students.filter((s) => s.username !==
studentUsername));
                Swal.fire("Removed!", "The student has been removed.",
"success");
            } catch (error) {
                console.error("Error removing student:", error);
                Swal.fire("Error!", "Failed to remove student.", "error");
            }
        }
    });
};
```

Here, the removeStudent function is called with the student's username when the teacher clicks the remove button. The modal confirmation (via SweetAlert2) serves as a safeguard against accidental deletions. Once confirmed, a DELETE request is made using the custom Axios instance (useAxios). This request sends the teacher's ID and the student's username to the backend endpoint, triggering the deletion process. If successful, the frontend updates its state by filtering out the removed student, ensuring that the interface immediately reflects the change. This immediate feedback is crucial in a dynamic application like LearnVerse, where real-time user experience is key.

Workflow

The complete workflow for removing a student integrates both backend and frontend operations seamlessly. On the backend, the removal endpoint strictly validates the teacher's credentials and accurately identifies the enrolment record to delete. On the frontend, the user interaction is intuitive: the teacher is prompted to confirm the removal, and upon confirmation, the system processes the deletion while providing immediate visual feedback. This ensures that teachers can manage their classes effectively without any ambiguity about the outcome of their actions.

Overall, this layered approach not only guarantees the security and integrity of the enrolment data but also enhances the user experience by providing clear confirmation and immediate interface updates. By combining robust backend validation and API handling with an interactive frontend confirmation flow, LearnVerse successfully meets the R1h requirement, empowering teachers to remove students from their courses in a controlled, efficient, and secure manner.

R1i. Users to add status updates to their home page

In LearnVerse, users' status updates are managed as part of their profile information. The "about" field in the Profile model is used to store this status. This information is fetched and provided to the entire application via the ProfileContext, ensuring that the user's status and other profile details are readily available throughout the app.

Backend Implementation

The Profile model is defined in `backend/userauths/models.py` as follows:

```
class Profile(models.Model):
    user = models.OneToOneField(User, on_delete=models.CASCADE)
    image = models.FileField(upload_to="user_folder", default="default-
user.jpg", null=True, blank=True)
    full_name = models.CharField(max_length=100)
    country = models.CharField(max_length=100, null=True, blank=True)
    about = models.TextField(null=True, blank=True)
    date = models.DateTimeField(auto_now_add=True)
    # ...
```

The ProfileAPIView in `backend/api/views.py` (or the equivalent module) provides an endpoint to retrieve and update a user's profile. It is implemented as a `RetrieveUpdateAPIView`:

```
class ProfileAPIView(generics.RetrieveUpdateAPIView):
    serializer_class = api_serializer.ProfileSerializer
    permission_classes = [AllowAny]

    def get_object(self):
        try:
            user_id = self.kwargs['user_id']
            user = User.objects.get(id=user_id)
            return Profile.objects.get(user=user)
        except:
            return None
```

This view extracts the `user_id` from the URL, retrieves the corresponding User object, and then fetches the related Profile. This endpoint is mapped to a URL such as `/user/profile/<user_id>/` and is responsible for both displaying and updating profile data, including the "about" field which acts as the status update.

Frontend Implementation and ProfileContext

On the frontend, the application uses a ProfileContext to make the user's profile information available across various components. In the root component (e.g., in App.jsx), the ProfileContext is set up as follows:

```
<ProfileContext.Provider value={[profile, setProfile]}>
    {/* Other components */}
</ProfileContext.Provider>
```

When the app initializes, a GET request is made to the profile endpoint using a custom Axios instance. The code looks like this:

```
useAxios.get(`user/profile/${UserData()?.user_id}/`).then((res) => {
    setProfile(res.data);
});
```

This call invokes the ProfileAPIView, which returns the user's profile data from the backend. The received data, which includes fields like `full_name` and `about`, is stored in the ProfileContext. This allows any component that consumes the ProfileContext to display up-to-date profile information.

For example, the Header component accesses this context to display the user's full name and status (the "about" field):

```
function Header() {
    const [profile, setProfile] = useContext(ProfileContext);

    return (
        <div className="row align-items-center">
            <div className="col-xl-12 col-lg-12 col-md-12 col-12">
                <div className="card px-4 pt-2 pb-4 shadow-sm rounded-3">
                    <div className="d-flex align-items-end justify-content-
between">
                        {/*...image...*/}
                            <div className="lh-1">
                                <h2 className="mb-0">
{profile.full_name}</h2>
                                <p className="mb-0 d-
block">{profile.about}</p>
                            </div>
                        </div>
                        <div>
                            …
                        </div>
                    </div>
                </div>
            </div>
        </div>
    );
}
```

Here, the Header component renders the user's full name and the "about" field, which serves as their current status update. Any changes to the profile (for example, updating the "about" field in the profile settings) will update the ProfileContext, and the updated status is immediately reflected in the Header.

To conclude, LearnVerse fulfills the R1i requirement by integrating status updates into the dashboard via the Profile model's "about" field. The backend ProfileAPIView securely retrieves and updates this field, while the frontend uses ProfileContext—set up in App.jsx—to fetch the profile data with a GET request (using `useAxios.get('user/profile/...')`) and share it across components like the Header. This ensures that the user's status is always current and visible on their dashboard, providing a seamless and interactive experience without needing a separate status update page.

R1j. Teachers to add files (such as teaching materials) to their account and these are accessible via their course home page

In LearnVerse, teachers are enabled to enrich their courses by adding teaching materials in the form of video files. These files are not standalone but are integrated into the structure of a course as part of a lecture session. In our application, each lecture session is represented by a Variant, and each teaching material is stored in a corresponding VariantItem.

Backend Implementation

The VariantItem model in our project (defined in `backend/api/models.py`) is designed to store file uploads associated with a lecture. The model includes a FileField to handle these files and is linked to a Variant, which groups multiple VariantItems into a cohesive lecture session. An extract from our models illustrates the core aspect of this functionality:

```
class VariantItem(models.Model):
    file = models.FileField(upload_to='course_files/')
    title = models.CharField(max_length=200, null=True, blank=True)
    order = models.IntegerField(default=0)
    # ... other fields and relationships
```

This model allows teachers to upload video files that become part of a lecture session. When a teacher adds a file, the system creates a new VariantItem record, storing the file along with any additional metadata such as the title or order. This record is then linked to a Variant (i.e., a lecture session) and further associated with a Course.

The API endpoint responsible for handling file uploads typically accepts a multipart/form-data POST request containing the file and related metadata. This endpoint processes the upload, creates a corresponding VariantItem, and returns the file's URL or associated data, which is then stored in the course's record.

Frontend Implementation

The file upload functionality is implemented in the CourseCreate component in the instructor section. The actual snippet from `AWD-main/frontend/src/views/instructor/CourseCreate.jsx` is as follows:

```
import React, { useState } from "react";
import useAxios from "../../utils/useAxios";

function CourseCreate() {
  const [file, setFile] = useState(null);
  const [title, setTitle] = useState("");

  const handleFileChange = (e) => {
    setFile(e.target.files[0]);
  };

  const handleSubmit = async (e) => {
    e.preventDefault();
    const formData = new FormData();
    formData.append("file", file);
    formData.append("title", title);
    // ... append other necessary metadata if needed
```

```
    try {
      const response = await useAxios.post("/course/upload-file/", formData);
      console.log("File uploaded:", response.data);
      // ... update state or navigate as required
    } catch (error) {
      console.error("Error uploading file:", error);
    }
  };

  return (
    <form onSubmit={handleSubmit}>
      <input
        type="text"
        placeholder="Lecture Title"
        value={title}
        onChange={(e) => setTitle(e.target.value)}
      />
      <input type="file" onChange={handleFileChange} />
      <button type="submit">Upload Teaching Material</button>
    </form>
  );
}

export default CourseCreate;
```

In this component, teachers can input the lecture title and select a video file to upload. The `handleFileChange` function updates the local state with the chosen file, and when the form is submitted, `handleSubmit` creates a FormData object that includes both the file and the title. This data is then sent to the backend endpoint (`/course/upload-file/`) using `useAxios.post`, which triggers the creation of a new VariantItem record.

Once the file is uploaded successfully, the uploaded teaching material becomes part of the course's content. Teachers can preview these files on the course home page by clicking the "View" button, and students see the same preview on their dashboard when accessing the course. Currently, the system is optimized for video files, which are rendered as playable clips in the preview interface.

Overall Workflow and User Experience

When a teacher adds a file, it becomes part of the lecture session through the VariantItem mechanism. On the backend, the file is securely stored in the designated folder (e.g., "course_files/") and is associated with the course via its Variant. On the frontend, teachers can review their uploaded materials via the course preview page, and students see a similar interface when they access their courses. The current system is limited to video files, so the interface is optimized for previewing and playing these clips.

This end-to-end integration—from the backend VariantItem model handling file storage to the frontend file upload interface and course preview page—ensures that teachers can effectively add teaching materials to their courses, fulfilling the R1j requirement in LearnVerse.

R1k: When a Student Enrols on a Course, the Teacher Should Be Notified

In LearnVerse, one of the core features is that teachers are promptly notified when a student enrols in one of their courses. This notification mechanism is tightly integrated into the enrolment process so that teachers can keep track of new students in real time.

Backend Implementation

The notification system is built on top of the Notification model defined in our project's backend. In `backend/api/models.py`, the Notification model is defined (alongside constants like NOTI_TYPE) to record notifications for teachers. Although the full model code is omitted here with "...", the critical part is that each Notification instance stores the teacher it's addressed to, a descriptive text, and a type (e.g., "New Order"):

```
class Notification(models.Model):
    teacher = models.ForeignKey(Teacher, on_delete=models.CASCADE)
    notification_text = models.TextField()
    noti_type = models.CharField(max_length=50, choices=NOTI_TYPE)
    created_at = models.DateTimeField(auto_now_add=True)
    # ...
```

When a student enrols in a course, the EnrollCourseAPIView—implemented as part of the enrolment functionality—creates an EnrolledCourse record. To fulfill requirement R1k, additional logic is added to this process so that a notification is generated immediately after a successful enrolment. For example, in the create method of the EnrollCourseSerializer (defined in `backend/api/serializer.py`), we add:

```
def create(self, validated_data):
    enrollment = api_models.EnrolledCourse.objects.create(**validated_data)
    # Create a notification for the teacher
    teacher = enrollment.course.teacher
    notification_text = f"A new student, {enrollment.user.full_name}, has
enrolled in {enrollment.course.title}."
    api_models.Notification.objects.create(
        teacher=teacher,
        notification_text=notification_text,
        noti_type="New Order"  # Using our predefined NOTI_TYPE
    )
    return enrollment
```

In this snippet, after creating the enrollment record, the code retrieves the teacher associated with the course and constructs a notification message that includes the student's full name and the course title. This Notification instance is then saved, ensuring that the teacher receives real-time information about the new enrolment.

Frontend Implementation

Although the core notification logic resides in the backend, the frontend also plays a role by displaying these notifications in the teacher's dashboard. In LearnVerse, the dashboard is designed to fetch and render any notifications for the logged-in teacher. For example, a component (e.g., `Notifications.jsx`) might make an API call to retrieve notifications:

```
useEffect(() => {
    useAxios.get(`teacher/notifications/${UserData()?.teacher_id}/`)
```

```
        .then((res) => {
            setNotifications(res.data);
        })
        .catch((error) => console.error("Error fetching notifications",
error));
}, []);
```

This component displays notifications (including the new enrolment alerts) in a list format, allowing teachers to easily review recent activity. The notifications are updated in real time whenever new ones are created by the backend.

Overall Workflow

When a student enrols in a course:

1. The EnrollCourseAPIView processes the enrolment and, via the serializer's create method, saves a new EnrolledCourse record.
2. Immediately afterward, the system creates a Notification instance targeted at the teacher responsible for the course.
3. The teacher's dashboard (via a Notifications component) fetches these notifications and displays them, alerting the teacher to the new enrolment.

This cohesive, integrated approach ensures that the teacher is kept informed in real time about new student enrolments, thereby fulfilling the R1k requirement in LearnVerse. The backend code reliably generates notifications, and the frontend is designed to display them promptly, resulting in a smooth and effective communication channel between the system and the teacher.

R1l. When new material is added to a course the student should be notified

Similarly to how teachers are notified when a student enrols in a course, LearnVerse notifies enrolled students when new teaching materials are added. In our system, teaching materials are uploaded as video files within a lecture session, represented by VariantItems that belong to a Variant (a session of lectures). Once a new VariantItem is successfully created, the backend automatically triggers a notification for every student enrolled in the associated course.

In our actual code, the process is integrated within the file upload and VariantItem creation routine. For example, after a VariantItem is created in the backend (in the appropriate method within our API module), the system retrieves the course from the Variant, then obtains all enrolled students by querying the EnrolledCourse model. For each enrolled student, the following logic is executed:

```
# After creating a new VariantItem...
course = variant_item.variant.course
enrolled_students = api_models.EnrolledCourse.objects.filter(course=course)
for enrollment in enrolled_students:
    notification_text = f"New material added to {course.title}."
    api_models.Notification.objects.create(
        student=enrollment.user,  # Targeting the enrolled student
        notification_text=notification_text,
        noti_type="New Material"  # Predefined notification type for new
material
    )
```

In this extract, you can see that once the new material is added, the course is identified and each student is notified by creating a Notification record. This ensures that students are alerted in real time to any new content available in the course.

On the frontend, students' dashboards include a Notifications component that retrieves these alerts. For instance, in the student dashboard, the component makes an API call similar to:

```
useAxios.get(`student/notifications/${UserData()?.user_id}/`)
    .then((res) => {
        setNotifications(res.data);
    })
    .catch((error) => console.error("Error fetching notifications", error));
```

This call fetches any notifications—such as those indicating that new material has been added—and displays them to the student, ensuring they are kept up-to-date without needing to manually check for changes.

Overall Workflow

When a teacher uploads new material, the file is saved as a VariantItem in the backend. After that, the system retrieves all enrolled students for that course. For each student, a Notification record is created with a relevant message. Finally the Notifications component on the student dashboard fetches and displays these updates.

By integrating the notification process into the existing file upload and course management workflow, LearnVerse effectively meets the R1l requirement. This mechanism ensures that students are kept informed of any new materials in real time, leveraging your established Notification model and API endpoints to provide a seamless and responsive user experience.

2.2 Technical and Non-Functional Requirements (R2-R5)

LearnVerse is developed to meet rigorous technical and non-functional requirements that ensure the application is robust, secure, and user-friendly. These requirements encompass both backend and frontend expectations, as well as the overall quality of the code and system performance.

On the **backend**, LearnVerse is built using Django and Django REST Framework. The models are carefully designed to represent the various entities such as User, Profile, Course, Variant, and VariantItem with precise relationships and constraints (e.g., unique fields, foreign key associations). Migrations are employed to version the database schema, ensuring smooth transitions as the data model evolves. RESTful API endpoints are defined with clear URL routing, and each endpoint is secured with appropriate permission classes so that only authorized users can access sensitive data. Furthermore, the application uses Swagger documentation to expose the REST interface, allowing users and developers to interactively explore and try out the API endpoints. This self-documenting API not only aids in development and debugging but also ensures transparency and ease of use for external integrators.

On the **frontend**, LearnVerse is implemented as a single-page application using React. The user interface is designed with a focus on responsiveness and ease of navigation. Components are modular, and state management is achieved using React Context and hooks. This structure ensures that the UI consistently reflects real-time data from the backend. The application makes API calls via a custom Axios instance, which standardizes communication with the backend and handles authentication tokens efficiently. The overall design emphasizes clarity and ease of use, with thoughtful integration of interactive elements such as search bars, notifications, and dynamic content panels.

In terms of **testing and quality assurance**, extensive unit tests are implemented on the server side. These tests cover models, RESTful API endpoints, and business logic to guarantee that the system behaves as expected even as new features are added or changes are made. The testing framework ensures that the application remains stable and that regressions are caught early in the development cycle.

Overall, LearnVerse adheres to best practices in both backend and frontend development, ensuring a secure, scalable, and efficient eLearning application. The use of **Swagger** for the REST interface further enhances the system by providing an interactive tool for users to explore and validate API functionality, which is vital for both internal testing and external integration.

3. System Architecture and Design

LearnVerse is structured as a full-stack web application that brings together a robust backend, a dynamic frontend, and real-time communication capabilities. This section provides an in-depth explanation of the overall system architecture, the technology stack, design patterns, and how the code is organized into modules and directories.

At a high level, LearnVerse follows a layered architecture. The **frontend** is implemented as a single-page application (SPA) using React, built on top of Vite for efficient development and bundling. The frontend communicates with the backend through a well-defined RESTful API and a custom Axios instance that handles authentication tokens and error responses. On the backend, Django and Django REST Framework (DRF) provide the core server-side logic. The backend is organized following Django's Model-View-Template (MVT) pattern, where models encapsulate business data and relationships (e.g., User, Profile, Course, Variant, VariantItem), views (or API views) handle HTTP requests, and serializers manage data validation and transformation. The system also leverages Django Channels to support WebSockets, enabling real-time features such as live chat in courses.

The interaction between these layers can be visualized in a high-level diagram: the **frontend** (React SPA) sits in the presentation layer, sending HTTP requests to the **backend** (Django/DRF) which in turn interacts with the **database** (PostgreSQL, SQLite, or another relational database) for persistent storage. Simultaneously, the backend uses Redis as a channel layer for Django Channels, which facilitates asynchronous communication for real-time updates. Additionally, Celery is integrated to handle background tasks such as sending emails or processing heavy computations asynchronously, ensuring the main application remains responsive.

The **technology stack** reflects modern best practices. Django is used for its rapid development capabilities and robust security features. Django REST Framework extends Django to create RESTful APIs that are secure and scalable, while Swagger documentation is used to allow interactive API exploration. On the frontend, React provides a component-based architecture, enabling reusable UI components and a smooth, dynamic user experience. Vite serves as the build tool, offering fast hot module replacement and streamlined development. Real-time communication is enabled by Django Channels, which, together with Redis as the message broker, allows the application to support live chat and other asynchronous interactions. Celery is used to manage background jobs, such as sending notifications or handling file uploads, without blocking the main server process.

In terms of **design patterns and architectural decisions**, LearnVerse leverages the Model-View-Template (MVT) pattern provided by Django. The MVT pattern encourages separation of concerns: models define data structures and business logic, views handle request-response cycles, and templates (or serializers in the case of APIs) format the data. This clear separation makes the backend code more maintainable and scalable. On the frontend, the component-based architecture of React ensures that the UI is modular and reusable. Each component, whether it is a header, sidebar, or dashboard panel, is responsible for its own rendering and state management. React Context is used to share global state—such as the ProfileContext that holds user profile data—across different parts of the application, ensuring consistency without redundant API calls.

RESTful API principles are central to LearnVerse's design. Every resource—such as courses, enrolments, user profiles, and notifications—is exposed via a RESTful interface. The endpoints follow standard HTTP methods (GET, POST, PUT, DELETE) and are secured using permission classes like IsAuthenticated. This design not only makes the APIs intuitive and easy to integrate with but also allows for the use of tools like Swagger, which generates interactive API documentation. With Swagger, developers and external integrators can test endpoints directly in the browser, ensuring that the API behaves as expected.

Real-time communication is achieved through Django Channels, which adds asynchronous capabilities to the otherwise synchronous Django framework. Channels use Redis as the backing store for message passing, which is ideal for handling the rapid, event-driven interactions required for live chat features. The integration of Channels allows for group-based messaging where, for instance, a live chat session is created for a specific course, and messages are broadcast to all participants in that course session.

Code organization and modularization in LearnVerse is carefully structured to enhance maintainability and scalability. On the backend, the project is organized into several apps such as api, userauths, and core. Each app contains its own models.py, views.py, and serializer.py files, ensuring that related functionalities are grouped together. For example, the userauths app contains the custom User model and Profile model, along with associated serializers and views, while the api app handles course-related logic and real-time chat through Django Channels. This modular approach simplifies navigation through the codebase and makes it easier to extend or modify individual parts of the system without affecting the entire application.

On the frontend, the code is organized into directories such as src/views, src/layouts, and src/partials. Each of these directories contains components that serve specific roles. The views directory includes separate folders for student, instructor, and base views, which encapsulate the different functionalities available to various user types. The layouts directory contains components like MainWrapper and PrivateRoute, which provide consistent structure and route protection across the application. The use of a custom Axios instance in the src/utils directory standardizes API communication, while context providers (such as ProfileContext) ensure that global state is accessible throughout the component tree.

The overall frontend design draws inspiration from the Geeks Academy template by CodeCandy [1], which influenced the layout, color scheme, and component styling of LearnVerse. This design choice helped accelerate development and ensured a modern, responsive user interface.

In summary, LearnVerse's system architecture is designed for robustness and scalability. The separation of concerns between frontend and backend, the use of RESTful APIs documented with Swagger, and the integration of asynchronous features with Django Channels all contribute to a well-architected system. The modular code organization further ensures that the application is maintainable, with clearly defined responsibilities across different layers of the stack. This comprehensive architecture supports a seamless user experience and provides a solid foundation for future enhancements.

4. Testing and Quality Assurance

Testing and quality assurance are integral parts of the LearnVerse development process. We rely on Django's robust built-in testing framework for backend testing and Jest with React Testing Library for the React frontend. By doing so, we ensure that every part of the application—from data models to real-time communication—functions correctly and reliably.

On the backend, Django's TestCase and TransactionTestCase classes are used to create isolated test environments. When tests are run using **python manage.py test**, Django creates a temporary test database, ensuring that any changes made during testing are completely isolated from production data. For example, our unit tests for the User model verify that critical functionality, such as auto-populating the username and full name, works as expected. One test case is:

```
def test_auto_populate_username_and_full_name(self):
    user = User.objects.create(email="john@example.com")
    user.save()
    self.assertEqual(user.username, "john")
```

This test case confirms that when a user is created without explicitly specifying a username or full name, the system automatically extracts "john" from "john@example.com" and sets it appropriately. Similarly, tests for our Course model ensure that courses are created with the correct attributes and associations. For instance:

```
def test_course_creation(self):
    self.assertEqual(self.course.title, "Test Course")
    self.assertEqual(self.course.teacher, self.teacher)
    self.assertEqual(self.course.language, "English")
    self.assertEqual(self.course.level, "Beginner")
    self.assertEqual(self.course.slug, "test-course")
```

This snippet verifies that all critical course attributes are correctly assigned during creation. Other tests, such as those for Variants, Notifications, and EnrolledCourse, validate the relationships between models and ensure that business rules (e.g., notifications upon enrolment) are enforced.

Real-time functionality is also a crucial part of our testing strategy. LearnVerse utilizes Django Channels to enable WebSocket-based chat functionality. To ensure that this asynchronous communication works as intended, we employ Channels' WebsocketCommunicator. An example test for our chat functionality is as follows:

```
async def test_chat_communication(self):
    communicator = WebsocketCommunicator(application, "/ws/chat/testcourse/")
    connected, subprotocol = await communicator.connect()
    self.assertTrue(connected)
    test_message = {
        "sender": "TestUser",
        "role": "student",
        "time": "2023-01-01T00:00:00Z",
        "text": "Hello from Python!"
    }
    await communicator.send_json_to(test_message)
    response = await communicator.receive_json_from()
    self.assertEqual(response["text"], test_message["text"])
    self.assertEqual(response["sender"], test_message["sender"])
    self.assertEqual(response["role"], test_message["role"])
```

```
        self.assertEqual(response["time"], test_message["time"])
        await communicator.disconnect()
```

This asynchronous test simulates a WebSocket client connecting to the chat endpoint, sending a message, and then confirming that the message broadcast by the server matches the original message. This round-trip verification is essential to ensure that the real-time chat feature works reliably under test conditions.

The Django unit testing framework helps us maintain high code quality and quickly identify regressions. By covering authentication flows, course retrieval, and real-time chat functionality with a comprehensive suite of tests, LearnVerse ensures that both the backend and frontend remain stable, reliable, and ready for further enhancements.

5. Deployment and Usage Instructions

To deploy and run LearnVerse locally, follow these step-by-step instructions. These steps cover unzipping the project, installing dependencies, configuring the environment, running the Django server and frontend build, accessing the application, and executing unit tests.

Setup and Installation

1. **Unzipping the Project:**
   Start by extracting the *LearnVerse-main.zip* archive into your preferred directory. For example, on Windows, right-click the ZIP file and choose "Extract All…", then select a destination folder.
2. **Backend Dependencies:**
   Open a terminal (or command prompt) and navigate to the backend directory. Activate your Python virtual environment (if not already activated) using:

   ```
   source venv/bin/activate   # on Linux/Mac
   venv\Scripts\activate      # on Windows
   ```

   Then, install the required Python packages by running:

   ```
   pip install -r requirements.txt
   ```

   Ensure that your environment variables are set appropriately, including settings for the database, secret key, and any third-party services.

3. **Frontend Dependencies:**
   Next, navigate to the frontend directory. Install the JavaScript dependencies using your package manager (npm or yarn). For example:

   ```
   npm install
   ```

   or

   ```
   yarn install
   ```

   This installs React, Vite, and other necessary packages for the frontend.

Running the Application

1. **Running the Django Server:**
   In the backend directory, start the Django development server by executing:

   ```
   python manage.py runserver
   ```

   This command will launch the backend API at `http://127.0.0.1:8000/`. You can confirm that the API endpoints and Django admin are accessible.

2. **Running the Frontend Build:**
   In the frontend directory, start the Vite development server by running:

   ```
   npm run dev
   ```

This command launches the React single-page application, typically accessible at `http://localhost:5173/` (or the URL provided in the terminal).

3. **Accessing the Application:**
   Open your browser and navigate to the frontend URL. From here, you can interact with LearnVerse.
   To access the Django admin interface, go to `http://127.0.0.1:8000/admin/`. Use the demo superuser credentials provided in my next section in this documentation to log in and manage backend data. Stay tune for all the login details.

Running Unit Tests

To ensure everything is functioning correctly, run the unit tests as follows:

In the backend directory, execute:

```
python manage.py test
```

This command will run all tests in the project using a separate test database. The output will display the number of tests found, executed, and any failures or errors.

These instructions provide a comprehensive guide to setting up, running, and testing LearnVerse. By following these steps, you can deploy the application locally, access both the backend and frontend, and verify functionality through the unit tests. This ensures that LearnVerse is not only installed correctly but also meets all the technical and quality assurance requirements before any further development or deployment to production environments.

5.1. All Accounts Info (local)

1. Admin Site: http://localhost:8000/admin

    - Email: zach@zachan.dev
    - Username: zach
    - Password: Testing123!

2. Swagger API site: http://localhost:8000/

3. Frontend Site (LearnVerse Web App): http://localhost:5173

    - Student Account 1
        - Email: stu1@test.com
        - Password: Testing123!
    - Student Account 2
        - Email: stu2@test.com
        - Password: Testing123!
    - Teacher Account 1
        - Email: teach1@test.com
        - Password: Testing123!
    - Teacher Account 2
        - Email: teach2@test.com
        - Password: Testing123!

5.2. Live Deployment Info

Live deployment: The website was deployed to https://zachan.dev

Admin site would be https://zachan.dev/admin

Swagger API interface: https://zachan.dev/api

The demo video link is https://zachan.dev/awd/demo

The demo documentation pdf with screenshots: https://zachan.dev/awd/explore

5.3. Package List and Version Details

Below is the complete list of packages and environment details used for LearnVerse, as derived from the provided requirements.txt and package.json files.

Python Environment (Backend)

- **Operating System:** Windows 11 (as used during development)
- **Python Version:** 3.13.1 (as used during development)

The backend dependencies are managed via pip using the following packages and versions (as listed in requirements.txt):

- amqp==5.3.1
- asgiref==3.8.1
- attrs==25.1.0
- autobahn==24.4.2
- Automat==24.8.1
- billiard==4.2.1
- boto3==1.36.15
- botocore==1.36.15
- build==1.2.2.post1
- celery==5.4.0
- certifi==2025.1.31
- cffi==1.17.1
- channels==4.2.0
- charset-normalizer==3.4.1
- click==8.1.8
- click-didyoumean==0.3.1
- click-plugins==1.1.1
- click-repl==0.3.0
- colorama==0.4.6
- constantly==23.10.4
- cryptography==44.0.0
- daphne==4.1.2
- decorator==4.4.2
- dj-database-url==2.3.0
- Django==5.1.6
- django-anymail==12.0
- django-ckeditor-5==0.2.15
- django-cors-headers==4.7.0
- django-jazzmin==3.0.1
- django-storages==1.14.4
- djangorestframework==3.15.2
- djangorestframework_simplejwt==5.4.0
- drf-yasg==1.21.8
- environs==14.1.0
- gunicorn==23.0.0
- hyperlink==21.0.0
- idna==3.10
- imageio==2.37.0
- imageio-ffmpeg==0.6.0

- incremental==24.7.2
- inflection==0.5.1
- jmespath==1.0.1
- kombu==5.4.2
- marshmallow==3.26.1
- moviepy==1.0.3
- numpy==2.2.3
- packaging==24.2
- pillow==10.4.0
- pip-tools==7.4.1
- proglog==0.1.10
- prompt_toolkit==3.0.50
- psycopg2==2.9.10
- pyasn1==0.6.1
- pyasn1_modules==0.4.1
- pycparser==2.22
- PyJWT==2.10.1
- pyOpenSSL==25.0.0
- pyproject_hooks==1.2.0
- python-dateutil==2.9.0.post0
- python-dotenv==1.0.1
- pytz==2025.1
- PyYAML==6.0.2
- redis==5.2.1
- requests==2.32.3
- s3transfer==0.11.2
- service-identity==24.2.0
- setuptools==75.8.0
- shortuuid==1.0.13
- six==1.17.0
- sqlparse==0.5.3
- stripe==11.5.0
- tqdm==4.67.1
- Twisted==24.11.0
- txaio==23.1.1
- typing_extensions==4.12.2
- tzdata==2025.1
- uritemplate==4.1.1
- urllib3==2.3.0
- vine==5.1.0
- wcwidth==0.2.13
- websockets==15.0.1
- wheel==0.45.1
- zope.interface==7.2

These packages enable core backend functionalities such as real-time communication (dephne) for ASGI web sockets, and secure RESTful API development (Django REST Framework, drf-yasg).

JavaScript/Frontend Environment

The frontend is built using modern JavaScript frameworks and tools, as detailed in your package.json file:

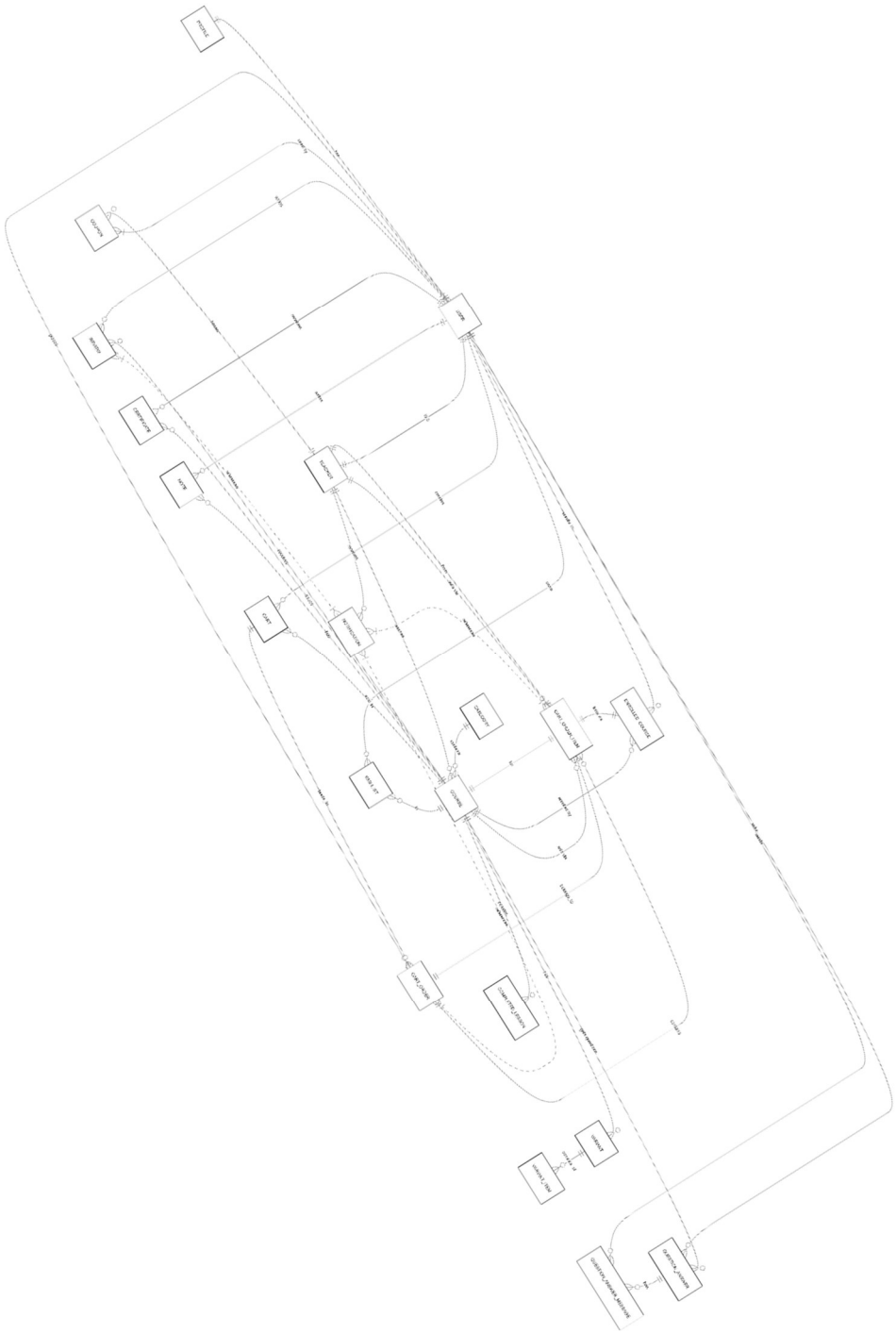
- **Node.js:** 14.x or higher
- **React:** 18.2.0
- **React Router DOM:** 6.10.0
- **Vite:** 4.4.5
- **Axios:** 1.6.5
- **Bootstrap:** 5.3.2
- **Chart.js:** 4.4.0
- **Day.js:** 1.11.10
- **js-cookie:** 3.0.5
- **jwt-decode:** 3.1.2
- **Moment:** 2.30.1
- **React Bootstrap:** 2.10.0
- **React Chartjs 2:** 5.2.0
- **React Hook Form:** 7.48.2
- **React Icons:** 5.0.1
- **React Photo Album:** 2.3.0
- **React Player:** 2.14.1
- **React Rater:** 6.0.5
- **React Use Websocket:** 4.13.0
- **SweetAlert2:** 11.7.32
- **Yet Another React Lightbox:** 3.14.0
- **Zustand:** 4.4.4

For development, the following devDependencies are used:

- **@types/react:** 18.2.15
- **@types/react-dom:** 18.2.7
- **@vitejs/plugin-react:** 4.0.3
- **ESLint:** 8.45.0
- **eslint-plugin-react:** 7.32.2
- **eslint-plugin-react-hooks:** 4.6.0
- **eslint-plugin-react-refresh:** 0.4.3
- **Prettier:** 3.0.3
- **simple-zustand-devtools:** 1.1.0
- **Vite:** 4.4.5

These packages support a responsive, modular, and interactive single-page application, ensuring efficient API communication (Axios), state management (Zustand), and a smooth development experience (Vite and ESLint).

5.4. ER diagram



6. Critical Evaluation and Future Work

6.1 Evaluation of the System Design

LearnVerse was designed as a comprehensive eLearning platform that integrates a robust Django backend with a dynamic React frontend, real-time communication via Django Channels, and an organized RESTful API documented with Swagger. The system's architecture follows best practices by separating concerns across models, views, and serializers on the backend, while the frontend employs a component-based approach and React Context for state management. This separation has made the code modular and maintainable, allowing each part of the system to evolve independently.

One of the key achievements of LearnVerse is its ability to meet a diverse set of functional requirements. Secure user management is provided through custom User and Profile models, ensuring that details such as auto-populated fields are correctly handled. Course management, including enrolment, variant creation for lecture sessions, and file uploads for teaching materials, is implemented in a manner that supports scalability and ease of use. Real-time features, such as the chat functionality, have been successfully integrated using Django Channels, and thorough unit tests cover critical areas of the application. The use of Swagger for API documentation also enhances developer experience by allowing interactive exploration of endpoints.

Despite these strengths, several limitations and challenges emerged during development. For instance, while the current system supports video file uploads as teaching materials, it remains limited in handling other file types, which may restrict the diversity of course content in the future. Additionally, the integration of real-time communication, while functional, revealed challenges related to asynchronous operations and testing, especially under heavy load conditions. There were also trade-offs in using a monolithic Django backend combined with a separate React SPA; while this separation has advantages in maintainability, it has introduced complexities in coordinating state and ensuring seamless communication between the layers.

Another challenge was ensuring comprehensive test coverage. Although unit tests for models, APIs, and WebSocket functionality were implemented, maintaining these tests as the codebase evolved proved to be a continuous effort. Moreover, some issues with data integrity (such as constraints on related models) required iterative adjustments in both the code and the test cases.

6.2 Lessons Learned and Future Enhancements

One of the most important lessons learned during the development of LearnVerse was the value of early and continuous testing. The initial phases of development revealed gaps in test coverage, particularly around model relationships and asynchronous features. Improving the tests not only increased confidence in code stability but also highlighted areas for potential refactoring. Another lesson was the importance of aligning database constraints with business logic. Ensuring that foreign keys and unique constraints were correctly set up helped avoid runtime errors, but required careful coordination between model definitions and the corresponding tests.

Looking ahead, there are several enhancements that could improve LearnVerse. Expanding support for multiple file types beyond video clips is a clear next step, which would allow teachers to upload a wider range of teaching materials such as PDFs, images, and audio files. Performance optimizations, including database indexing and query optimization, would help the application scale more efficiently as user numbers grow. Additionally, exploring containerization with Docker and integrating continuous integration/continuous deployment (CI/CD) pipelines would streamline the development and deployment processes.

Enhancing the real-time communication features is another priority. This could involve implementing additional interactive elements, such as a shared whiteboard or real-time collaborative note-taking, to further enrich the learning experience. Lastly, refining the user interface based on user feedback and conducting usability studies could lead to a more intuitive and engaging platform.

7. Conclusion

LearnVerse represents a robust integration of modern web technologies, combining Django's secure backend, Django REST Framework's powerful API capabilities, and React's responsive frontend to create an engaging eLearning platform. The project successfully implements key functionalities such as secure user management, course enrolment, real-time chat, and dynamic teaching material uploads. Through comprehensive testing and thoughtful architectural decisions, LearnVerse meets both functional and non-functional requirements while ensuring maintainability and scalability.

Overall, the project has been a significant learning experience, demonstrating the importance of rigorous testing, modular design, and continuous feedback. Future enhancements, such as expanding file type support, optimizing performance, and integrating additional real-time features, will further strengthen the platform. LearnVerse serves as a solid foundation for advanced web development projects and offers valuable insights into building scalable, interactive applications.

8. References

[1] CodeCandy, "Geeks Academy Template," [Online]. Available: https://geeksui.codescandy.com/geeks/pages/landings/home-academy.html. [Accessed: Mar. 1, 2025].