Competency Based Learning Material (CBLM)

Web Application Development with Python

Level-4

Module: Creating Final project

Code: CBLM- OU-ICT-WADP-07-L4-V1

National Skills Development Authority
Chief Advisor's Office
Government of the People's Republic of Bangladesh

Copyright

National Skills Development Authority
Chief Advisor's Office
Level6-11, Biniyog Bhaban,
E-6 / B, Agargaon, Sher-E-Bangla Nagar Dhaka-1207, Bangladesh.
Emailec@nsda.gov.bd
Websitewww.nsda.gov.bd.
National Skills Portalhttp:\\skillsportal.gov.bd

Approved by the Authority ……..meeting held on .....

How to use this Competency Based Learning Material (CBLM)

The module contains training materials and activities for you to complete. These activities may be completed as part of structured classroom activities or you may be required you to work at your own pace. These activities will ask you to complete associated learning and practice activities in order to gain knowledge and skills you need to achieve the learning outcomes.

1. Review the **Learning Activity** page to understand the sequence of learning activities you will undergo. This page will serve as your road map towards the achievement of competence.

2. Read the **Information sheet s.** This will give you an understanding of the jobs or tasks you are going to learn how to do. Once you have finished reading the **Information sheet s** complete the questions in the **Self-Check.**

3. **Self-**Checks are found after each **Information sheet** . **Self-Checks** are designed to help you know how you are progressing. If you are unable to answer the questions in the **Self-Check** you will need to re-read the relevant **Information sheet** . Once you have completed all the questions check your answers by reading the relevant **Answer Keys** found at the end of this module.

4. Next move on to the **Job Sheets. Job Sheets** provide detailed information about *how to do the job* you are being trained in. Some **Job Sheets** will also have a series of **Activity Sheets**. These sheets have been designed to introduce you to the job step by step. This is where you will apply the new knowledge you gained by reading the Information sheet s. This is your opportunity to practise the job. You may need to practise the job or activity several times before you become competent.

Specification **sheets**, specifying the details of the job to be performed will be provided where appropriate.

A review of competency is provided on the last page to help remind if all the required assessment criteria have been met. This record is for your own information and guidance and is not an official record of competency

When working though this Module always be aware of your safety and the safety of others in the training room. Should you require assistance or clarification please consult your trainer or facilitator.

When you have satisfactorily completed all the Jobs and/or Activities outlined in this module, an assessment event will be scheduled to assess if you have achieved competency in the specified learning outcomes. You will then be ready to move onto the next Unit of Competency or Module

Table of Contents

Module Content

Unit of Competency	**Create Final project**
Unit Code	OU-ICT-WADP-08-L4-V1
Module Title	**Creating Final project**
Module Descriptor	This module covers the knowledge, skills and attitudes required to create Final project It includes the task of creating project, applying notification and activation processes and deploying project
Nominal Hours	60 Hours
Lerning Outcome	After completing the practice of the module, the trainees will be able to perform the following jobs 1. Create project 2. Apply notification and activation processes 3. Deploy project

Assessment Criteria

1. SRS is created from the assigned domain-specific scenario
2. Models are designed and class diagram is created
3. User Interface (UI) is designed
4. GitHub repo is managed
5. All functionalities are implemented and tested
6. Database and classes are designed following flow diagram
7. User Interface (UI) is designed
8. All functionalities are implemented and tested
9. Create a hosting account on preferred platform
10. Preparation for hosting is performed
11. Django project is uploaded

Project is enabled

Learning Outcome 1: Create project

Assessment Criteria	1. SRS is created from the assigned domain-specific scenario 2. Models are designed and class diagram is created 3. User Interface (UI) is designed 4. GitHub repo is managed 5. All functionalities are implemented and tested
Conditions and Resources	1. Real or simulated workplace 2. CBLM 3. Handouts 4. Laptop 5. Multimedia Projector 6. Paper, Pen, Pencil, Eraser 7. Internet facilities 8. White board and marker 9. Audio Video Device
Contents	1. SRS 2. Models and clss diagram 3. User Interface (UI) 4. GitHub repo 5. All functionalities
Activities/job/Task	1. Building a Django Quiz Application
Training Methods	1. Discussion 2. Presentation 3. Demonstration 4. Guided Practice 5. Individual Practice 6. Project Work 7. Problem Solving 8. Brainstorming
Assessment Methods	Assessment methods may include but not limited to 1. Written Test 2. Demonstration 3. Oral Questioning 4. Portfolio

Learning Experience 1: Create project

In order to achieve the objectives stated in this learning guide, you must perform the learning steps below. Beside each step are the resources or special instructions you will use to accomplish the corresponding activity.

Learning Activities	Recourses/Special Instructions
1. Trainee will ask the instructor about the learning materials	1. Instructor will provide the learning materials 'Create project'
2. Read the Information sheet and complete the Self Checks & Check answer sheets on "Create project"	2. Read Information sheet 1: Create project 3. Answer Self-check 1: Create project 4. Check your answer with Answer key 1: Create project
3. Read the Job/Task Sheet and Specification Sheet and perform job/Task	5. Job Sheet-1: Building a Django Quiz Application

Information sheet 1: Create project

Learning Objective:

After completion of this Information sheet , the learners will be able to explain, define and interpret the following contents:

1.1. SRS is created from the assigned domain-specific scenario
1.2. Models are designed and class diagram is created
1.3. User Interface (UI) is designed
1.4. GitHub repo is managed
1.5. All functionalities are implemented and tested

1.1. SRS (System Requirements Specification)

Introduction

This document outlines the System Requirements Specification (SRS) for a Django web application called "Quiz Master". It defines the functionalities, features, and user requirements of the application.

System Overview

Quiz Master is a web application that allows users to create, take, and manage quizzes. It caters to educators, trainers, or individuals who want to create interactive quizzes to assess knowledge.

Functional Requirements

A. User Roles

- User:
    - Can register and log in.
    - Can search for and browse existing quizzes.
    - Can take quizzes and view their results.
- Administrator (Optional):
    - (In addition to User functionalities)
    - Can create new quizzes.
    - Can add questions (multiple choice, true/false, open ended).
    - Can manage categories for quizzes.
    - Can view quiz statistics (e.g., average scores, user performance).

B. Core Functionalities

- User Management:
    - User registration and authentication (login/logout).
    - User profile management (optional).
- Quiz Management (Administrator):
    - Create new quizzes with titles, descriptions, and categories (optional).
    - Add questions (text, answer choices, correct answer for multiple choice and true/false, answer field for open ended).
    - Edit existing quizzes and questions.
    - Delete quizzes and questions.
- Quiz Taking (User):
    - Search for quizzes by title, category, or keyword.
    - View quiz details (title, description, creator, difficulty).
    - Take quizzes by answering questions (selecting options for multiple choice/true/false, typing answers for open ended).

- Submit quizzes and view immediate results (score and correct/incorrect answers).

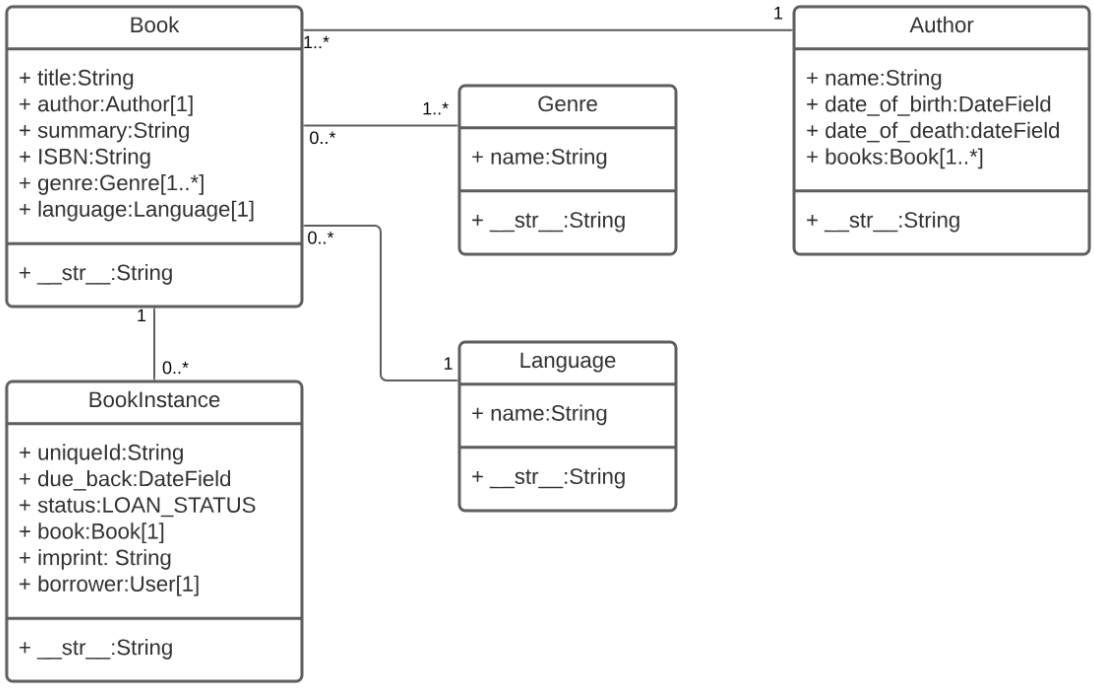
C. Additional Functionalities (Optional):

- Timer functionality for quizzes (optional).
- Leaderboard to display top-scoring users (optional).
- User progress tracking for attempted quizzes (optional).

D. Non-Functional Requirements

- **Performance:** The application should load pages and quizzes quickly, even with a moderate number of users.
- **Security:** User authentication and authorization should be implemented to prevent unauthorized access.
- **Scalability:** The application should be designed to accommodate a growing number of users and quizzes.
- **Usability:** The user interface (UI) should be intuitive and easy to navigate for both users and administrators.

1.2. Data Models and Class Diagrams



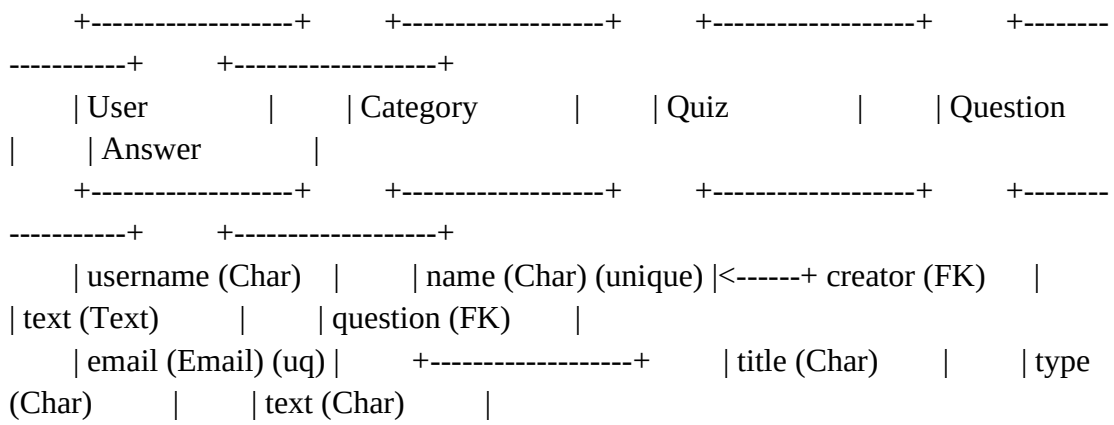
A. Data Models

- User:
    - username (CharField)
    - email (EmailField) (unique)
    - password (CharField)
    - (Optional: first_name, last_name)
- Category (Optional):
    - name (CharField) (unique)
- Quiz:
    - title (CharField)
    - description (TextField)
    - difficulty (CharField) (e.g., easy, medium, hard)
    - category (ForeignKey to Category, null=True, blank=True)
    - creator (ForeignKey to User)
    - created_at (DateTimeField, auto_now_add=True)
- Question:
    - quiz (ForeignKey to Quiz)
    - text (TextField)
    - type (CharField) (e.g., multiple_choice, true_false, open_ended)
- Answer (Multiple Choice and True/False):
    - question (ForeignKey to Question)
    - text (CharField)
    - is_correct (BooleanField)
- UserAnswer (Optional - for tracking user responses):
    - user (ForeignKey to User)
    - question (ForeignKey to Question)
    - chosen_answer (ForeignKey to Answer, null=True, blank=True) (for multiple choice/true/false)
    - answer_text (TextField) (for open ended)
    - submitted_at (DateTimeField, auto_now_add=True) (optional)
- Result (Optional - for storing quiz results):
    - user (ForeignKey to User)

- quiz (ForeignKey to Quiz)
- score (IntegerField)
- completed_at (DateTimeField, auto_now_add=True) (optional)

1.3. Class Diagram

Here's a basic UML class diagram representing the relationships between the models:



1.4. User Interface (UI) Design

The UI of your Django quiz project is crucial for providing a user-friendly and engaging experience. Here's a breakdown of the key elements to consider:

A. User Roles and Views

- **User:**
  - **Landing Page:** View a list of available quizzes (searchable/filtered by category, difficulty, etc.).
  - **Quiz Details:** See information about a specific quiz (title, description, creator, difficulty).
  - **Take Quiz:** Answer questions in the chosen quiz format (multiple choice, true/false, open ended).
  - **Results:** View their score and potentially correct/incorrect answers after completing a quiz (optional: leaderboard).
  - **Profile (Optional):** Manage user information (optional).
- **Administrator (Optional):**
  - **Dashboard:** View an overview of quizzes, users, and results (optional).

- **Create Quiz:** Define a new quiz with title, description, difficulty, category (optional).
- **Add Questions:** Create questions for a specific quiz, specifying the type (multiple choice, true/false, open ended) and answers.
- **Manage Quizzes:** Edit or delete existing quizzes.
- **View Results (Optional):** See users' performance on quizzes (optional).

B. Design Considerations

- **Responsiveness:** Ensure the UI adapts seamlessly to different screen sizes (desktops, tablets, mobiles).
- **Clarity and Readability:** Use clear fonts, appropriate colors, and spacing for optimal readability.
- **Navigation:** Provide clear navigation menus or buttons to allow users to easily access different functionalities.
- **Intuitive Interface:** Make the UI intuitive and easy to understand, minimizing user confusion.
- **User Feedback:** Provide feedback mechanisms (e.g., progress indicators, confirmation messages) to guide users.
- **Accessibility:** Consider accessibility best practices to make your UI usable for people with disabilities.

C. UI Technologies

- **Django Templates:** Utilize Django's templating system to generate dynamic HTML content based on your data models and logic.
- **HTML, CSS, and JavaScript:** These are the core building blocks for web UIs. You can use them to structure the layout, style the appearance, and add interactivity to your quiz project.
- **Front-End Frameworks (Optional):** Consider using a front-end framework like Bootstrap, Foundation, or Materialize to streamline your UI development process. These frameworks offer pre-built components, styles, and layouts that can save you time and effort.

D. Tools and Resources

- **Django Documentation:** The official Django documentation provides detailed guidance on using Django templates and building web applications: https://docs.djangoproject.com/en/5.0/
- **HTML, CSS, and JavaScript Tutorials:** Numerous online tutorials and resources can help you learn these fundamental web development technologies.
- **Front-End Framework Documentation:** If you choose to use a front-end framework, refer to its official documentation for specific instructions and examples.

E. Wireframing and Mockups (Optional)

- **Wireframes:** Create low-fidelity sketches to define the basic layout and functionality of each UI page. This helps visualize the user flow without getting bogged down in visual details.
- **Mockups:** Develop higher-fidelity mockups with more refined visuals (colors, fonts) to represent the intended look and feel of your UI before coding.

By carefully considering these UI design aspects, you can create a user-friendly and engaging interface for your Django quiz project.

1.5. Version Control with GitHub

Here's a step-by-step guide on integrating Git and GitHub for version control in your Django quiz project:

A. Install Git (if not already installed):

- **Windows:** Download and install Git for Windows from the official website: https://git-scm.com/downloads
- **macOS/Linux:** Git is often pre-installed. Open a terminal and type git --version to check. If not installed, use your package manager (e.g., sudo apt install git on Ubuntu/Debian).

B. Initialize a Git Repository:

- Open a terminal window and navigate to your Django project's root directory.
- Run the command git init to create a new Git repository in your project directory. This initializes version control for your project.

C. Create a Remote Repository on GitHub:

- Go to https://github.com/ and create an account if you don't have one already.
- Click on "New repository" to create a new repository.
- Give your repository a descriptive name (e.g., django-quiz-project). Optionally, add a brief description and initialize with a README file.
- Click "Create repository" to finalize the remote repository on GitHub.

D. Link Your Local Repository to GitHub (Remote):

- In your terminal, run the following command, replacing <remote_repository_url> with the actual HTTPS URL of your newly created GitHub repository:

  Bash
  ```
  git remote add origin <remote_repository_url>
  ```
- You can retrieve the URL from your GitHub repository's "Clone or download" section.

E. Stage and Commit Changes:

- Git tracks changes to your files. Use git status to see the current status of your files (modified, untracked, etc.).
- To add specific files to the staging area for your next commit, use:

  Bash

  git add <filename>
- You can add multiple files or use git add . to add all modified files.
- To create a commit with a descriptive message, run:

  Bash

  git commit -m "<commit message>"
- The commit message summarizes the changes you're committing. Use clear and concise messages for each commit (e.g., "Added user model", "Implemented quiz creation form").

F. Push Your Local Commits to GitHub:

- After staging and committing your changes, you can push them to your remote repository on GitHub:

  Bash

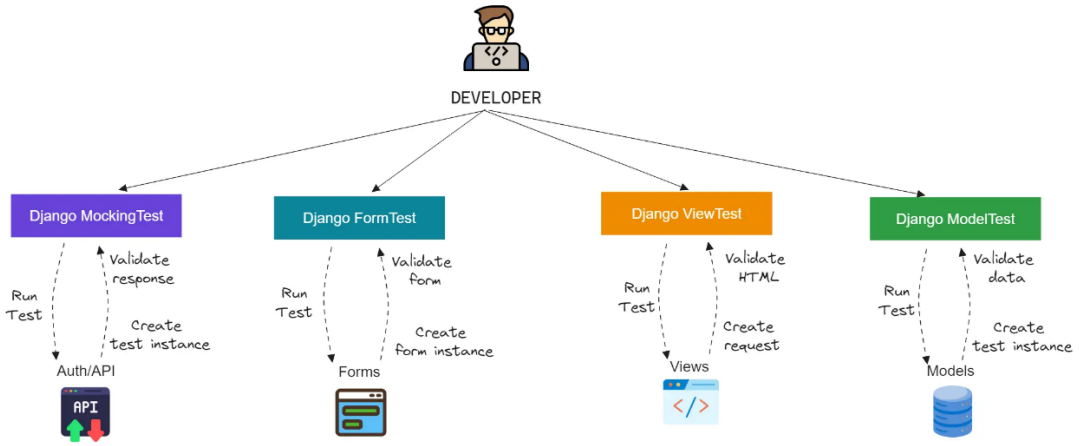
  git push origin main
- This command pushes your local commits to the main branch of your remote repository on GitHub.

G. Collaboration and Version Control:

- With your project version-controlled on GitHub, you can collaborate with others effectively. You can create branches for new features, make pull requests to merge changes, and track the version history visually on GitHub.
- Remember to commit your code regularly and push your changes to GitHub to maintain a centralized backup and facilitate collaboration.

1.6. Functionality Implementation and Testing

This stage involves bringing your Django quiz project to life by implementing the core functionalities and rigorously testing them. Here's a breakdown of the key aspects:



A. Functionality Implementation:

- This stage focuses on coding the core functionalities outlined in your SRS document (Section 1). Utilize Django's features, libraries, and tools to build the following features for your quiz application:

a. User Management

- Implement user registration using Django's built-in user model or a third-party authentication package if needed.
- Develop functionalities for user login/logout and manage user profiles (optional).

a. Quiz Management (Optional - Administrator functionality)

- Create views and forms to allow administrators to create quizzes with titles, descriptions, difficulty levels, and categories (optional).
- Develop functionalities for adding questions (multiple choice, true/false, open ended) and defining answers with the correct option marked.
- Implement logic to edit existing quizzes and questions.
- Allow administrators to delete quizzes and associated questions.

a. Quiz Taking (User functionality)

- Create views to display a list of available quizzes (searchable/filtered by category, difficulty, etc.).
- Implement functionalities for users to view quiz details (title, description, creator, difficulty) before taking them.
- Develop logic to handle the quiz-taking process, displaying questions, capturing user answers (selecting options or entering text), and submitting the quiz.
- Upon submission, calculate the user's score and provide immediate feedback (optional: display correct/incorrect answers).

B. Testing

- Thorouh testing is crucial to ensure your application functions as intended and to identify and fix bugs before deployment. Here are different types of testing to consider:

a. Unit Testing

Test individual functions and modules in isolation to verify their behavior. Utilize testing frameworks like `unittest` or `pytest` to write unit tests that focus on specific functionalities.

b. Functional Testing

Test the overall functionality of your application from a user's perspective. This involves testing user workflows, form submissions, database interactions, and expected outcomes. Create test cases that simulate user interactions and verify the application's behavior.

c. Integration Testing

Test how different parts of your application work together, ensuring smooth integration between models, views, forms, and other components. Write test cases that interact with multiple functionalities and verify their combined behavior.

d. UI Testing (Optional)

Test the user interface for responsiveness, usability, and functionality across different browsers and devices. Consider using browser automation tools for UI testing.

C. Tools and Resource

- Django provides built-in testing functionalities and integrates well with popular testing frameworks. Here are some helpful tools:
    - django.test.TestCase: Provides a base class for writing unit and functional tests.
    - django.test.Client: Simulates a web browser for testing views and forms.
- Leverage testing frameworks like pytest or unittest for more advanced testing features and functionalities.
- Refer to the official Django testing documentation for detailed guidance: https://docs.djangoproject.com/en/5.0/

Self-Check Sheet 1: Create project

1. Which functionality allows users to search for quizzes based on specific criteria?
Answer:

A. User registration
B. User profile management
C. Quiz searching
D. Creating new categories

2. What type of data model stores the relationship between a question and its corresponding answers (for multiple choice and true/false questions)?
Answer:

A. User
B. Category
C. Question
D. Answer

3. According to the UI Design Considerations, what element is NOT mentioned as crucial for the user interface?
Answer:

A. Complex and cluttered design
B. Responsiveness (adapting to different screen sizes)
C. Clarity and Readability
D. Intuitive Interface (easy to understand)

4. **What Git command is used to initialize a new Git repository in your project directory?**
Answer:

A. git remote add origin <remote_repository_url>
B. git add <filename>
C. git commit -m "<commit message>"
D. git init

5. **What is the purpose of a commit message in Git?**
Answer:

A. To download a specific version of the code from GitHub.
B. To create a new branch for a feature in your project.
C. To link your local repository to a remote repository on GitHub.
D. To summarize the changes you're making when committing your code.

Answer Key 1: Create project

1. Which functionality allows users to search for quizzes based on specific criteria?
Answer:

A. User registration
B. User profile management
C. Quiz searching
D. Creating new categories

2. What type of data model stores the relationship between a question and its corresponding answers (for multiple choice and true/false questions)?
Answer:

A. User
B. Category
C. Question
D. Answer

3. According to the UI Design Considerations, what element is NOT mentioned as crucial for the user interface?
Answer:

A. Complex and cluttered design
B. Responsiveness (adapting to different screen sizes)
C. Clarity and Readability
D. Intuitive Interface (easy to understand)

6. **What Git command is used to initialize a new Git repository in your project directory?**

Answer:

A. git remote add origin <remote_repository_url>
B. git add <filename>
C. git commit -m "<commit message>"
D. git init

7. **What is the purpose of a commit message in Git?**

Answer:

A. To download a specific version of the code from GitHub.
B. To create a new branch for a feature in your project.
C. To link your local repository to a remote repository on GitHub.
D. To summarize the changes you're making when committing your code.

Job Sheet-1: **Building a Django Quiz Application**

UoC Cover

OU-ICT-WADP-01- L4-V1 - Enable Django Framework Environment

OU-ICT-WADP-02- L4-V1 - Develop dynamic web pages

OU-ICT-WADP-03- L4-V1 - Use Django Model

OU-ICT-WADP-04- L4-V1- Process Forms

OU-ICT-WADP-05- L4-V1- Customize UI

OU-ICT-WADP-06- L4-V1 - Apply User Management

OU-ICT-WADP-07- L4-V1 - Create API using Django REST Framework

OU-ICT-WADP-08-L4- V1 - Create Final project

Working Procedure / Steps

1. Define Requirements (SRS):

- Create a System Requirements Specification (SRS) document outlining the functionalities, features, and user requirements for your quiz application.
- Refer to Section 1.1 of the learning objectives for guidance on what to include in the SRS.
- This document will serve as a blueprint for your development process.

2. Design Data Models (ER Diagrams):

- Design your data models using entity-relationship (ER) diagrams. These models represent the entities (data objects) involved in your application and their relationships.
- Section 1.2 of the learning objectives provides a list of models you might consider, including User, Category, Quiz, Question, Answer, UserAnswer (optional), and Result (optional).
- Define the attributes (fields) for each model and establish relationships between them (e.g., a Quiz can have many Questions, a User can take many Quizzes).

3. Plan User Interface (UI):

- Sketch out the user interface (UI) for your quiz application. Consider different user roles (user and optional administrator) and the functionalities they need to access.
- Section 1.3 of the learning objectives provides a breakdown of key UI elements to consider, including landing page, quiz details, quiz taking, results, profile (optional), dashboard (optional), and create/edit functionalities (optional for administrators).

- Focus on creating a user-friendly and intuitive interface that is responsive across different devices.

4. Version Control with Git and GitHub:

- Set up version control for your project using Git and GitHub. This allows you to track changes to your code, collaborate with others, and revert to previous versions if necessary.
- Section 1.4 of the learning objectives provides a step-by-step guide on installing Git, initializing a Git repository, creating a remote repository on GitHub, linking your local repository to GitHub, staging and committing changes, and pushing your commits to GitHub.

5. Development and Implementation:

- Start coding your Django application based on the defined requirements, data models, and UI design.
- Utilize Django's functionalities to implement user management, quiz management (optional for administrators), quiz taking, and result processing.
- Refer to Section 1.5 of the learning objectives for a breakdown of functionalities to implement and consider using tools like Django's built-in user model, forms, views, and generic class-based views.

6. Testing:

- Implement a comprehensive testing strategy to ensure your application works as expected and identify and fix bugs before deployment.
- Section 1.5 of the learning objectives outlines different types of testing to consider, including unit testing, functional testing, integration testing, and optional UI testing.
- Utilize Django's built-in testing functionalities and testing frameworks like `unittest` or `pytest` to write comprehensive test cases.

Specification Sheet 1: Building a Django Quiz Application

1. Define Requirements (SRS)

Purpose
To outline the functionalities, features, and user requirements for a quiz application.

Functionalities

- **User Management**: Registration, login, profile management.
- **Quiz Management**: Creating, editing, and deleting quizzes (admin).
- **Quiz Taking**: Users can start and complete quizzes.
- **Results**: Display results and scores to users after quiz completion.
- **Categories**: Organize quizzes into categories.
- **Questions and Answers**: Create multiple-choice questions with answers.

Features

- **Landing Page**: Overview of available quizzes.
- **Quiz Details Page**: Information about each quiz.
- **Quiz Interface**: User-friendly interface for answering questions.
- **Result Page**: Display user scores and correct answers.
- **Profile Page** (optional): Manage user information and view quiz history.
- **Dashboard** (optional for admins): Overview of quizzes, users, and performance metrics.
- **Create/Edit Functionalities** (optional for admins): Manage quizzes, questions, and categories.

User Requirements

- **End Users**: Access quizzes, take quizzes, view results.
- **Administrators** (optional): Manage quizzes, users, and view performance metrics.

2. Design Data Models (ER Diagrams)

Entities and Relationships

- **User**: UserID, Username, Email, PasswordHash.
- **Category**: CategoryID, Name, Description.
- **Quiz**: QuizID, Title, Description, CategoryID (FK), CreatedBy (FK).
- **Question**: QuestionID, QuizID (FK), Text.
- **Answer**: AnswerID, QuestionID (FK), Text, IsCorrect.

- **UserAnswer** (optional): UserAnswerID, UserID (FK), QuestionID (FK), AnswerID (FK).
- **Result** (optional): ResultID, UserID (FK), QuizID (FK), Score.

ER Diagram

1. **User** 1

Quiz (A User can create many Quizzes).

2. **Quiz** 1

Question (A Quiz can have many Questions).

3. **Question** 1

Answer (A Question can have many Answers).

4. **User** 1

UserAnswer (A User can have many UserAnswers).

5. **Question** 1

UserAnswer (A Question can have many UserAnswers).

6. **Quiz** 1

Result (A Quiz can have many Results).

7. **User** 1

Result (A User can have many Results).

3. Plan User Interface (UI)

UI Elements

- **Landing Page**: Display available quizzes and categories.
- **Quiz Details Page**: Show quiz description and start button.
- **Quiz Taking Page**: User interface for answering questions.
- **Results Page**: Display scores and correct answers.
- **Profile Page** (optional): User information and quiz history.
- **Dashboard** (optional for admins): Manage quizzes, users, and view statistics.
- **Create/Edit Pages** (optional for admins): Forms for creating and editing quizzes and questions.

Design Considerations

- **Responsiveness**: Ensure the UI is responsive and works well on various devices (desktop, tablet, mobile).
- **Accessibility**: Design for accessibility, including screen readers and keyboard navigation.

4. Version Control with Git and GitHub

Setup

- **Install Git**: Follow instructions for installing Git on your operating system.
- **Initialize Repository**: git init in your project directory.
- **Create Remote Repository**: Set up a new repository on GitHub.
- **Link Local to Remote**: git remote add origin [URL].
- **Stage Changes**: git add ..
- **Commit Changes**: git commit -m "Initial commit".
- **Push Changes**: git push -u origin main.

5. Development and Implementation

Framework

- **Django**: Use Django for backend development.

Implementation

- **User Management**: Utilize Django's built-in authentication system.
- **Quiz Management**: Implement views and forms for creating, editing, and deleting quizzes (optional for admin).
- **Quiz Taking**: Create views and templates for quiz participation.
- **Results Processing**: Calculate and display results.

Tools

- **Django Models**: For database interactions.
- **Django Forms**: For user inputs and data validation.
- **Django Views**: For handling HTTP requests and responses.
- **Django Templates**: For rendering HTML pages.

6. Testing

Testing Types

- **Unit Testing**: Test individual components and methods.
- **Functional Testing**: Test specific functionalities from the user's perspective.
- **Integration Testing**: Test the interaction between different components.
- **UI Testing** (optional): Test the user interface for usability and responsiveness.

Tools

- **Django Testing Framework**: Utilize Django's built-in testing framework.
- **pytest**: Alternative testing framework for more advanced features.

Implementation

- **Write Test Cases**: Use unittest or pytest to write comprehensive test cases for all functionalities.

Learning Outcome 2: Apply notification and activation process

Assessment Criteria	1. Database and classes are designed following flow diagram 2. User Interface (UI) is designed 3. All functionalities are implemented and tested
Conditions and Resources	1. Real or simulated workplace 2. CBLM 3. Handouts 4. Laptop 5. Multimedia Projector 6. Paper, Pen, Pencil, Eraser 7. Internet facilities 8. White board and marker 9. Audio Video Device
Contents	1. Database and classes 2. User Interface (UI) of designed project 3. All functionalitie
Activities/job/Task	1. Design a Django Quiz Application 2. Creating the Blog with User Comments 3. Creating To-Do List application 4. Building the Recipe Sharing Site
Training Methods	1. Discussion 2. Presentation 3. Demonstration 4. Guided Practice 5. Individual Practice 6. Project Work 7. Problem Solving 8. Brainstorming
Assessment Methods	Assessment methods may include but not limited to 1. Written Test 2. Demonstration 3. Oral Questioning 4. Portfolio

Learning Experience 2: Apply notification and activation process

In order to achieve the objectives stated in this learning guide, you must perform the learning steps below. Beside each step are the resources or special instructions you will use to accomplish the corresponding activity.

Learning Activities	Recourses/Special Instructions
1. Trainee will ask the instructor about the learning materials	1. Instructor will provide the learning materials 'Apply notification and activation process'
2. Read the Information sheet and complete the Self Checks & Check answer sheets on "Apply notification and activation process"	2. Read Information sheet 2: Apply notification and activation process 3. Answer Self-check 2: Apply notification and activation process 4. Check your answer with Answer key 2: Apply notification and activation process
3. Read the Job/Task Sheet and Specification Sheet and perform job/Task	5. Job Sheet-2.1: Design a Django Quiz Application Job Sheet-2.2: Creating the Blog with User Comments Job Sheet-2.3: Creating To-Do List application Job Sheet-2.4: Building the Recipe Sharing Site

Information sheet 2: Apply notification and activation process

Learning Objective:

After completion of this Information sheet , the learners will be able to explain, define and interpret the following contents:

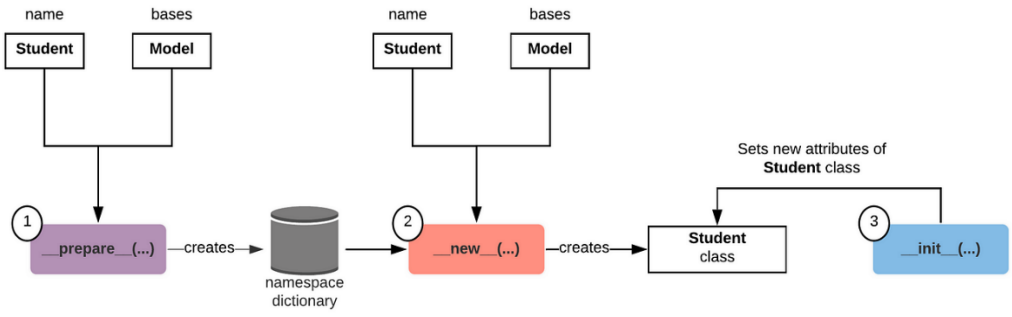
2.1. Database and classes are designed following flow diagram
2.2. User Interface (UI) is designed
2.3. All functionalities are implemented and tested

2.1. Design Database and classes following flow diagram

While a flow diagram can be helpful for visualizing the overall process of your quiz project, it's not the most suitable approach for designing the database and classes in Django. Here's why:

- **Flow diagrams focus on processes:** They represent the sequence of steps involved in a system, not the data structures and relationships.
- **Django uses object-oriented models:** Your database schema will be represented through classes in Python, reflecting real-world entities and their relationships.

However, the flow diagram can still be a valuable starting point for understanding the high-level interactions and data flow within your quiz project.



Here's how you can translate your flow diagram concepts into Django models and classes:

A. Identify Entities

From your flow diagram, identify the key entities involved in the quiz project. These entities typically translate to models in Django. Here are some potential entities based on common quiz functionalities:

- User
- Category (Optional)
- Quiz
- Question
- Answer (For multiple choice and true/false)
- UserAnswer (Optional - to track user responses)
- Result (Optional - to store user quiz results)

B. Define Model Attributes:

For each entity, determine the relevant attributes (fields) that represent its properties. These attributes become the fields of your Django models. Here's an example breakdown for some entities:

- **User:** username (CharField), email (EmailField, unique), password (CharField)
- **Category (Optional):** name (CharField, unique)
- **Quiz:** title (CharField), description (TextField), difficulty (CharField), category (ForeignKey to Category, null=True, blank=True), creator (ForeignKey to User), created_at (DateTimeField, auto_now_add=True)
- **Question:** quiz (ForeignKey to Quiz), text (TextField), type (CharField) # e.g., multiple_choice, true_false, open_ended

C. Define Relationships

Identify the relationships between entities. These relationships are represented using Django's ForeignKey or ManyToManyField fields within your models. Here are some examples:

- A Quiz belongs to a single User who created it (creator field with ForeignKey to User).
- A Quiz can optionally belong to a single Category (category field with ForeignKey to Category, null=True and blank=True to allow quizzes without a category).
- A Question belongs to a single Quiz (quiz field with ForeignKey to Quiz).
- An Answer belongs to a single Question (question field with ForeignKey to Question).

D. Additional Considerations

- You might need additional models depending on your specific functionalities (e.g., UserAnswer to track user responses, Result to store quiz results).
- Consider using Django's built-in user model or a third-party authentication package if you need more advanced user management features.
- Define appropriate data types for each field in your models (e.g., CharField for text, IntegerField for numbers).

Example

Connect To Database

Following Python code shows how to connect to an existing database. If the database does not exist, then it will be created and finally a database object will be returned.

```
#!/usr/bin/python

import sqlite3

conn = sqlite3.connect('test.db')

print "Opened database successfully";
```

Here, you can also supply database name as the special name **:memory:** to create a database in RAM. Now, let's run the above program to create our database **test.db** in the current directory. You can change your path as per your requirement. Keep the above code in sqlite.py file and execute it as shown below. If the database is successfully created, then it will display the following message.

```
$chmod +x sqlite.py
$./sqlite.py
Open database successfully
```

Create a Table

Following Python program will be used to create a table in the previously created database.

```
#!/usr/bin/python

import sqlite3

conn = sqlite3.connect('test.db')
print "Opened database successfully";

conn.execute('''CREATE TABLE COMPANY
         (ID INT PRIMARY KEY     NOT NULL,
         NAME           TEXT    NOT NULL,
         AGE            INT     NOT NULL,
         ADDRESS        CHAR(50),
         SALARY         REAL);''')
print "Table created successfully";

conn.close()
```

When the above program is executed, it will create the COMPANY table in your **test.db** and it will display the following messages −

```
Opened database successfully
Table created successfully
```

Insert Operation

Following Python program shows how to create records in the COMPANY table created in the above example.

```
#!/usr/bin/python

import sqlite3

conn = sqlite3.connect('test.db')
print "Opened database successfully";

conn.execute("INSERT INTO COMPANY (ID,NAME,AGE,ADDRESS,SALARY) \
      VALUES (1, 'Paul', 32, 'California', 20000.00 )");

conn.execute("INSERT INTO COMPANY (ID,NAME,AGE,ADDRESS,SALARY) \
      VALUES (2, 'Allen', 25, 'Texas', 15000.00 )");

conn.execute("INSERT INTO COMPANY (ID,NAME,AGE,ADDRESS,SALARY) \
      VALUES (3, 'Teddy', 23, 'Norway', 20000.00 )");

conn.execute("INSERT INTO COMPANY (ID,NAME,AGE,ADDRESS,SALARY) \
      VALUES (4, 'Mark', 25, 'Rich-Mond ', 65000.00 )");

conn.commit()
print "Records created successfully";
conn.close()
```

When the above program is executed, it will create the given records in the COMPANY table and it will display the following two lines –

```
Opened database successfully
Records created successfully
```

Select Operation

Following Python program shows how to fetch and display records from the COMPANY table created in the above example.

```
#!/usr/bin/python

import sqlite3

conn = sqlite3.connect('test.db')
print "Opened database successfully";

cursor = conn.execute("SELECT id, name, address, salary from COMPANY")
for row in cursor:
   print "ID = ", row[0]
   print "NAME = ", row[1]
   print "ADDRESS = ", row[2]
   print "SALARY = ", row[3], "\n"

print "Operation done successfully";
conn.close()
```

When the above program is executed, it will produce the following result.

```
Opened database successfully
ID = 1
NAME = Paul
ADDRESS = California
SALARY = 20000.0

ID = 2
NAME = Allen
ADDRESS = Texas
SALARY = 15000.0

ID = 3
NAME = Teddy
ADDRESS = Norway
SALARY = 20000.0

ID = 4
NAME = Mark
ADDRESS = Rich-Mond
SALARY = 65000.0

Operation done successfully
```

Update Operation

Following Python code shows how to use UPDATE statement to update any record and then fetch and display the updated records from the COMPANY table.

```
#!/usr/bin/python

import sqlite3

conn = sqlite3.connect('test.db')
print "Opened database successfully";

conn.execute("UPDATE COMPANY set SALARY = 25000.00 where ID = 1")
conn.commit
print "Total number of rows updated :", conn.total_changes

cursor = conn.execute("SELECT id, name, address, salary from COMPANY")
for row in cursor:
   print "ID = ", row[0]
   print "NAME = ", row[1]
   print "ADDRESS = ", row[2]
   print "SALARY = ", row[3], "\n"

print "Operation done successfully";
conn.close()
```

When the above program is executed, it will produce the following result.

```
Opened database successfully
Total number of rows updated : 1
ID = 1
NAME = Paul
ADDRESS = California
SALARY = 25000.0

ID = 2
NAME = Allen
ADDRESS = Texas
SALARY = 15000.0

ID = 3
NAME = Teddy
ADDRESS = Norway
SALARY = 20000.0

ID = 4
NAME = Mark
ADDRESS = Rich-Mond
SALARY = 65000.0

Operation done successfully
```

Delete Operation

Following Python code shows how to use DELETE statement to delete any record and then fetch and display the remaining records from the COMPANY table.

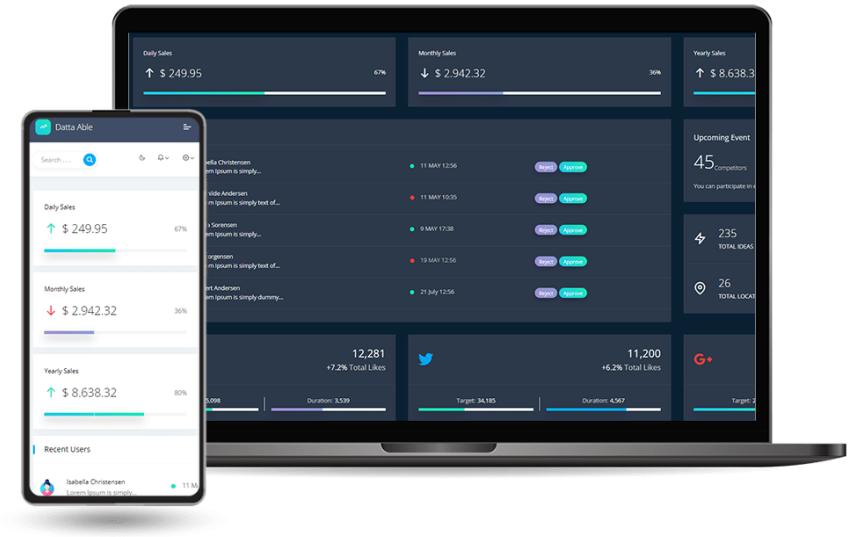
```
#!/usr/bin/python

import sqlite3

conn = sqlite3.connect('test.db')
print "Opened database successfully";

conn.execute("DELETE from COMPANY where ID = 2;")
conn.commit()
print "Total number of rows deleted :", conn.total_changes

cursor = conn.execute("SELECT id, name, address, salary from COMPANY")
for row in cursor:
   print "ID = ", row[0]
   print "NAME = ", row[1]
   print "ADDRESS = ", row[2]
   print "SALARY = ", row[3], "\n"

print "Operation done successfully";
conn.close()
```

When the above program is executed, it will produce the following result.

```
Opened database successfully
Total number of rows deleted : 1
ID = 1
NAME = Paul
ADDRESS = California
SALARY = 20000.0

ID = 3
NAME = Teddy
ADDRESS = Norway
SALARY = 20000.0

ID = 4
NAME = Mark
ADDRESS = Rich-Mond
SALARY = 65000.0

Operation done successfully
```

2.2. Design User Interface (UI)

We've covered the core concepts of UI design for your quiz project. Now, let's delve into specific elements and considerations:



A. User Roles and Views:

- **User:**
    - **Landing Page:**
        - List available quizzes with search/filter options (category, difficulty, keyword).
        - Display quiz title, description, creator, difficulty.
        - Optionally, show average score or number of attempts.
    - **Quiz Details:**
        - Provide detailed information about a specific quiz.
        - Include creator name, description, difficulty level, and category (if applicable).
    - **Take Quiz:**
        - Present questions in the appropriate format (multiple choice, true/false, open ended).
        - Clearly indicate the type of question and provide instructions (e.g., select one answer, enter a short answer).
        - Allow users to navigate between questions or review their answers before submission.
    - **Results Page:**
        - Display the user's score after completing the quiz.
        - Optionally, show correct/incorrect answers with explanations.

        - Consider including a leaderboard (optional) to display top performers.
    - **Profile (Optional):**
        - Allow users to manage their profile information (e.g., name, email).
        - Optionally, display quiz history or performance summary.

- **Administrator (Optional):**

    - **Dashboard (Optional):**
        - Overview of quizzes, users, and results (e.g., number of quizzes, active users, average scores).
        - Charts or graphs (optional) to visualize quiz statistics.
    - **Create Quiz:**
        - Form to define quiz title, description, difficulty level, and category (optional).
        - Option to add questions of various types (multiple choice, true/false, open ended).
        - Interface for defining answer options and marking the correct answer for multiple choice and true/false questions.
    - **Manage Quizzes:**
        - List existing quizzes with details (title, creator, difficulty).
        - Allow editing quiz details (title, description, difficulty, category).
        - Option to delete quizzes (along with associated questions).
    - **View Results (Optional):**
        - See user performance on quizzes (e.g., individual scores, average scores by quiz).
        - Optionally, filter results by quiz, user, or date range.

B. Design Considerations:

- **Responsiveness:** Ensure your UI adapts seamlessly to different screen sizes (desktops, tablets, mobiles). Use responsive design techniques or CSS frameworks like Bootstrap to achieve this.
- **Clarity and Readability:** Use clear fonts, appropriate colors, and sufficient spacing for optimal readability.
- **Navigation:** Provide clear navigation menus or buttons to allow users to easily access different functionalities. Consider a breadcrumb navigation system to show users their location within the application.
- **Intuitive Interface:** Strive for a user-friendly and intuitive interface. Minimize user confusion by using clear labels, instructions, and error messages.
- **User Feedback:** Provide feedback mechanisms (e.g., progress indicators, confirmation messages) to guide users through the quiz-taking process.

- **Accessibility:** Consider accessibility best practices to make your UI usable for people with disabilities. This includes using alt text for images, proper color contrast, and keyboard navigation support.

C. UI Technologies

- **Django Templates:** Utilize Django's template system to generate dynamic HTML content based on your data models and logic. This allows you to create reusable templates for different UI components (landing page, quiz details, etc.).
- **HTML, CSS, and JavaScript:** These are the core building blocks for web UIs. You can use them to structure the layout, style the appearance, and add interactivity to your quiz project.
- **Front-End Frameworks (Optional):** Consider using a front-end framework like Bootstrap, Foundation, or Materialize to streamline your UI development process. These frameworks offer pre-built components, styles, and layouts that can save you time and effort.

D. Tools and Resources

- **Django Documentation:** The official Django documentation provides detailed guidance on using Django templates and building web applications: https://docs.djangoproject.com/en/5.0/
- **HTML, CSS, and JavaScript Tutorials:** Numerous online tutorials and resources can help you learn these fundamental web development technologies.
- **Front-End Framework Documentation:** If you choose to use a front-end framework, refer to its official documentation for specific instructions and examples.

E. Quiz Taking Process

- **Clarity and Consistency:** Ensure question formats (multiple choice, true/false, open ended) are presented consistently with clear instructions.
- **Question Navigation:** Allow users to easily navigate between questions (e.g., using previous/next buttons or a progress bar).
- **Answer Selection:** Provide user-friendly controls for selecting answers (radio buttons for multiple choice, checkboxes for true/false, text input for open ended).
- **Review Functionality (Optional):** Allow users to review their answers before submitting the quiz.
- **Time Limit (Optional):** Implement a timer if you want to add a time pressure element to the quiz.
- **Submission and Feedback:** Upon submission, display the user's score and optionally provide correct/incorrect answers with explanations.

F. User Interface Components

- Landing Page:
    - Consider displaying a search bar for filtering quizzes by category, difficulty, keyword, or creator (optional).
    - Optionally, showcase popular or recently created quizzes.
- Quiz Details Page:
    - Clearly present quiz title, description, creator name, difficulty level, and category (if applicable).
    - Optionally, show average score or number of attempts (if data is available).
    - Include a button to start the quiz.
- Results Page:
    - Display the user's score prominently.
    - Consider using visuals like progress bars or charts to represent performance.
    - Optionally, show a leaderboard displaying top scorers (anonymously or with usernames, depending on your preference).

G. Considerations

- **Progress Tracking:** Implement a progress indicator or display the current question number to inform users of their position within the quiz.
- **Error Handling:** Provide clear error messages for user input mistakes or invalid actions.
- **Help and Documentation (Optional):** Consider including a help section with instructions on using the quiz platform or taking quizzes.

2.3. Implemented and Tested All functionalities

Starting the Project Folder

To start the project use this command

```
django-admin startproject quiz
cd quiz
```

To start the app use this command

python manage.py startapp home

Now add this app to the ‘settings.py’

File Structure
file-structure-
File Structure

then we register our app in settings.py file in the installed_apps sections like shown below

```
INSTALLED_APPS = [
    "django.contrib.admin",
    "django.contrib.auth",
    "django.contrib.contenttypes",
    "django.contrib.sessions",
    "django.contrib.messages",
    "django.contrib.staticfiles",
    "home",
    "django_extensions",
]
```

Setting up necessary Files
models.py

The code defines Django models for a quiz application. It includes models for categories, questions, and answers, along with a common base model for shared fields like timestamps. These models are used to organize and store quiz-related data in a Django project.

```
from django.db import models
import uuid
import random

class BaseModel(models.Model):
    uid = models.UUIDField(primary_key=True, default=uuid.uuid4, editable = True)
    created_at = models.DateField(auto_now_add = True)
    updated_at = models.DateField(auto_now = True)



    class Meta:
        abstract = True

class Types(BaseModel):
    gfg_name = models.CharField(max_length=100)
    def __str__(self) -> str:
        return self.gfg_name




class Question(BaseModel):
    gfg = models.ForeignKey(Types, related_name='gfg',on_delete=
models.CASCADE)
    question = models.CharField(max_length=100)
    marks = models.IntegerField(default = 5)

    def __str__(self) -> str:
        return self.question

    def get_answers(self):
        answer_objs =  list(Answer.objects.filter(question= self))
        data = []
        random.shuffle(answer_objs)

        for  answer_obj in answer_objs:
            data.append({
                'answer' :answer_obj.answer,
```

```
            'is_correct' : answer_obj.is_correct
        })
    return data

class Answer(BaseModel):
    question = models.ForeignKey(Question,related_name='question_answer',
on_delete =models.CASCADE)
    answer = models.CharField(max_length=100)
    is_correct = models.BooleanField(default = False)

    def __str__(self) -> str:
        return self.answer
    views.py
```

The code is a part of a Django web application for quizzes:

home Function: Displays a list of quiz categories. Redirects to the quiz page for a selected category.

quiz Function: Displays a quiz page for a specific category.

get_quiz Function: Retrieves random quiz questions, optionally filtered by category. Returns them as JSON. Handles exceptions with a “Something went wrong” response.

```
from django.shortcuts import render,  redirect
from django.http import JsonResponse
from .models import *

import random

def home(request):
    context = {'catgories': Types.objects.all()}

    if request.GET.get('gfg'):
        return redirect(f"/quiz/?gfg={request.GET.get('gfg')}")

    return render(request, 'home.html', context)

def quiz(request):
```

```
    context = {'gfg': request.GET.get('gfg')}
    return render(request, 'quiz.html', context)

def get_quiz(request):
    try:
        question_objs = Question.objects.all()

        if request.GET.get('gfg'):
            question_objs = question_objs.filter(gfg__gfg_name__icontains =
request.GET.get('gfg'))

        question_objs = list(question_objs)
        data = []
        random.shuffle(question_objs)



        for question_obj in question_objs:

            data.append({
                "uid" : question_obj.uid,
                "gfg": question_obj.gfg.gfg_name,
                "question": question_obj.question,
                "marks": question_obj.marks,
                "answer" : question_obj.get_answers(),
            })

        payload = {'status': True, 'data': data}

        return JsonResponse(payload)  # Return JsonResponse

    except Exception as e:
        print(e)
        return HttpResponse("Something went wrong")
```
admin.py

Here we are registering the models.

```
from django.contrib import admin
from .models import *

# Register your models here.

class AnswerAdmin(admin.StackedInline):
    model = Answer

class QuestionAdmin(admin.ModelAdmin):
    inlines = [AnswerAdmin]

admin.site.register(Types)
admin.site.register(Question, QuestionAdmin)
admin.site.register(Answer)
```

Creating GUI

home.html

This HTML template is designed for a Django quiz app, providing a category selection form for users and using Bootstrap for styling and layout.

```
<!doctype html>
<html lang="en">
<head>
    <!-- Required meta tags -->
    <meta charset="utf-8">
    <meta name="viewport" content="width=device-width, initial-scale=1, shrink-to-fit=no">

    <!-- Bootstrap CSS -->
    <link rel="stylesheet" href="https://cdn.jsdelivr.net/npm/bootstrap@4.3.1/dist/css/bootstrap.min.css"
```

```
        integrity="sha384-
ggOyR0iXCbMQv3Xipma34MD+dH/1fQ784/j6cY/iJTQUOhcWr7x9JvoRxT2MZw
1T"
        crossorigin="anonymous">

   <title>Django Quiz App</title>
</head>
<body>

<div class="container mt-5 pt-5">
   <!-- Added the Geeksforgeeks heading here -->
   <h1 style="color: green; text-align: center;">Geeksforgeeks</h1>

   <div class="col-md-6 mx-auto">
      <form action="">
         <div class="form-group">
            <label for="">Select Types</label>
            <select name="gfg" id="" class="form-control" value="gfg">
               <option value="Choose">Choose</option>
               {% for type in catgories %}
               <option value="{{type.gfg_name}}">{{type.gfg_name}}</option>
               {% endfor %}
            </select>
         </div>
         <button class="btn btn-danger mt-3">Submit</button>
      </form>
   </div>
</div>

</body>
</html>
```

quiz.html

This code integrates Vue.js into an HTML page to create a dynamic quiz interface where users can select answers, check correctness, and receive alerts. It fetches questions from a Django API based on the selected category.

```
<!doctype html>
<html lang="en">
  <head>
    <!-- Required meta tags -->
    <meta charset="utf-8">
```

```
    <meta name="viewport" content="width=device-width, initial-scale=1, shrink-to-fit=no">

    <!-- Bootstrap CSS -->
    <link rel="stylesheet" href="https://cdn.jsdelivr.net/npm/bootstrap@4.3.1/dist/css/bootstrap.min.css" integrity="sha384-ggOyR0iXCbMQv3Xipma34MD+dH/1fQ784/j6cY/iJTQUOhcWr7x9JvoRxT2MZw1T" crossorigin="anonymous">

    <title>Django Quiz App</title>
  </head>
  <body>

    <script src="https://unpkg.com/vue@3.0.0-rc.5/dist/vue.global.prod.js"></script>
<div id="app">
  <ul>
    <div class="container mt-5 pt-5">
       <div class="col-md-6 mx-auto">


              <h3>Give Quiz</h3>
              <div v-for="question in questions">
              <hr>
              <p>[[question.question]]</p>

           <div class="form-check" v-for="answer in question.answer">
              <input @change="checkAnswer($event, question.uid)" :value="answer.answer" class="form-check-input" type="radio" name="flexRadioDefault" id="flexRadioDefault1">
              <label class="form-check-label" for="flexRadioDefault1">
                [[answer.answer]]
              </label>
            </div>


          </div>
            </div>
       </div>
    <!-- <li v-for="task in tasks">[[task]]</li> -->
  </ul>
</div>
  <script>
    const app = Vue.createApp({
```

```
  el: '#app',
  delimiters: ['[[', ']]'],
  data() {
    return {
      gfg: '{{gfg}}',
      questions :[]
          }
  },
  methods : {
      getQuestions(){
          var _this = this
          fetch(`/api/get-quiz/?gfg=${this.gfg}`)
          .then(response => response.json())
          .then(result =>{
              console.log(result)
              _this.questions = result.data
          })
      },
      checkAnswer(event, uid){

          this.questions.map(question =>{

              answer = question.answer
              for(var i=0; i<answer.length; i++){
                  if(answer[i].answer==event.target.value){
                      if(answer[i].is_correct){
                          console.log('Your answer is correct!')
                          alert("Hurray ypur answer is correct !????")
                      }else{
                          console.log('Your answer is wrong!')
                          alert("Better luck next time !????")
                      }
                  }
              }

          })
          console.log(event.target.value  , uid)
      }
  },
  created() {
      this.getQuestions( )
    },
});
```

```
    app.mount('#app')
  </script>

</body>
</html>
```

quiz/urls.py

This is the urls.py file of our project quiz in this file we just map the other urls.py file of our home app for performing the some operations

```
from django.contrib import admin
from django.urls import path, include

urlpatterns = [
    path('', include('home.urls')),
    path("admin/", admin.site.urls),


]
```

home/urls.py

This is the urls.py file this is our app home urls.py

```
from django.contrib import admin
from django.urls import path
from .import views

urlpatterns = [
   path('', views.home , name = 'home'),
   path('quiz/', views.quiz, name= 'quiz'),
   path('api/get-quiz/', views.get_quiz, name='get_quiz')
 ]
```

Deployement of the Project

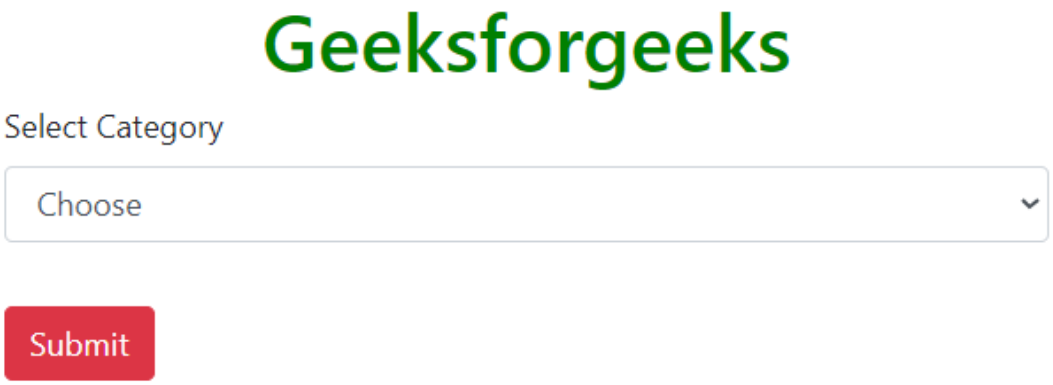
Run these commands to apply the migrations:

```
python3 manage.py makemigrations
python3 manage.py migrate
```

Run the server with the help of following command:

python3 manage.py runserver

Output



In conclusion, the Quiz Django app is a web application designed for creating and administering quizzes. It combines Django, a powerful Python web framework, with Vue.js, a JavaScript framework for building dynamic user interfaces. The app allows users to:

- Select a Category: Users can choose a quiz category from a dropdown menu.
- Answer Questions: The app presents users with a list of questions within the selected category. Users can select answers to these questions.
- Check Correctness: Upon selecting an answer, the app checks if it's correct and provides immediate feedback through alerts.
- Dynamic UI: Vue.js is used to create a dynamic user interface that updates in real-time as users interact with the quiz.
- API Integration: The app fetches quiz questions from a Django API based on the selected category.

Overall, the Quiz Django app combines the power of Django for backend development and Vue.js for frontend interactivity to create an engaging and educational quiz-taking experience.

Self-Check Sheet 2: Apply notification and activation process

1. What is NOT a suitable approach for designing the database and classes in Django?
Answer:

a) User registration
b) User profile management
c) Flow diagrams
d) Defining model attributes

2. Which Django model represents the relationship between a question and its corresponding answers (for multiple choice and true/false questions)?
Answer:

a) User
b) Category
c) Question
d) Answer

3. What is an example of a UI Design Consideration for the Django quiz project?
Answer:

a) Complex and cluttered design
b) Responsiveness (adapting to different screen sizes)
c) Using a programming language other than Python
d) Implementing unnecessary functionalities

4. Which command is used to start a new Django project?
Answer:

a) python manage.py startapp home
b) File Structure
c) INSTALLED_APPS = [...]
d) django-admin startproject quiz

5. What is the purpose of the get_answers() method in the Question model (models.py)?
Answer:

a) To create a new question object.
b) To retrieve and shuffle the answer objects associated with a question.
c) To define the correct answer for a question.
d) To update the question text.

Answer Key 2: Apply notification and activation process

1. What is NOT a suitable approach for designing the database and classes in Django?
Answer:

a) User registration (Correct)
b) User profile management
c) Flow diagrams (focusing on processes)
d) Defining model attributes

2. Which Django model represents the relationship between a question and its corresponding answers (for multiple choice and true/false questions)?
Answer:

a) User
b) Category
c) Question
d) Answer **(Correct)**

3. What is an example of a UI Design Consideration for the Django quiz project?
Answer:

a) Complex and cluttered design
b) Responsiveness (adapting to different screen sizes) **(Correct)**
c) Using a programming language other than Python
d) Implementing unnecessary functionalities

4. Which command is used to start a new Django project?
Answer:

a) python manage.py startapp home
b) File Structure
c) INSTALLED_APPS = [...]
d) django-admin startproject quiz **(Correct)**

5. What is the purpose of the get_answers() method in the Question model (models.py)?
Answer:

a) To create a new question object.
b) To retrieve and shuffle the answer objects associated with a question. **(Correct)**
c) To define the correct answer for a question.
d) To update the question text.

Job Sheet-2.1: **Design a Django Quiz Application**

UoC Cover

OU-ICT-WADP-01- L4-V1 - Enable Django Framework Environment

OU-ICT-WADP-02- L4-V1 - Develop dynamic web pages

OU-ICT-WADP-03- L4-V1 - Use Django Model

OU-ICT-WADP-04- L4-V1- Process Forms

OU-ICT-WADP-05- L4-V1- Customize UI

OU-ICT-WADP-06- L4-V1 - Apply User Management

OU-ICT-WADP-07- L4-V1 - Create API using Django REST Framework

OU-ICT-WADP-08-L4- V1 - Create Final project

Steps:

1. Project Setup:
    - Install Django and set up a new project using django-admin startproject.
    - Create a new app for your e-commerce store with python manage.py startapp store.
    - Set up database (SQLite or PostgreSQL) and configure settings for the project.
2. User Authentication:
    - Set up Django's built-in authentication for user registration, login, and logout.
    - Implement email verification for account activation after registration using django-allauth or django-registration.
3. Product Model:
    - Create a Product model with attributes like name, description, price, stock quantity, and image.
    - Add a Category model to categorize products.
4. Shopping Cart and Orders:

- Implement a shopping cart using Django sessions to store products temporarily before checkout.
- Create an Order model with fields like user, product, quantity, order date, shipping address, and status.

5. Checkout and Payment:
    - Integrate payment gateways (like Stripe or PayPal) for payment processing.
    - Ensure users can enter shipping details during the checkout process.
6. Admin Panel:
    - Customize the Django admin panel to manage products, categories, and orders.
    - Set up low stock alerts for administrators.
7. Notifications:
    - Use Django's email backend or third-party services (like SendGrid or Mailgun) to send:
        - Order confirmation emails.
        - Shipping updates emails.
        - Low stock alerts to admins.
        - Optional product announcements.
8. UI/UX Design:
    - Implement front-end templates for the homepage, product listings, product detail pages, cart, and checkout.
    - Use CSS and JavaScript for a better user experience.
9. Testing & Debugging:
    - Test all functionalities (account registration, cart, checkout, notifications).
    - Use Django’s testing framework to ensure the application works correctly.
10. Deployment:
    - Deploy the site using platforms like Heroku, AWS, or DigitalOcean.
    - Set up environment variables, database configuration, and static file management.

Specification Sheet 2.1: Design a Django Quiz Application

Necessary Tools

Sl. No	Name of Tools	Specification	Unit	Quantity
1	Django	Web framework for Python	N/A	1
2	Python	Programming language for development	N/A	1
3	Visual Studio Code / PyCharm	Code editor/IDE	N/A	1
4	Git	Version control tool	N/A	1
5	Postman	API testing tool (if using APIs for payments)	N/A	1
6	Stripe API / PayPal API	Payment gateway integration	N/A	1

Necessary Equipment

Sl. No	Name of Equipment	Specification	Unit	Quantity
1	Laptop / PC	For development	N/A	1
2	Server (optional)	For deployment and hosting	N/A	1

Necessary Materials

Sl. No	Name of Materials	Specification	Unit	Quantity
1	Internet Connection	For research, API calls, deployment	N/A	1
2	Email Service (Mailgun/SendGrid)	For sending notifications via email	N/A	1
3	Database (SQLite/PostgreSQL)	Database for storing user and product data	N/A	1

Job Sheet-2.2: **Creating the Blog with User Comments**

UoC Cover

OU-ICT-WADP-01- L4-V1 - Enable Django Framework Environment

OU-ICT-WADP-02- L4-V1 - Develop dynamic web pages

OU-ICT-WADP-03- L4-V1 - Use Django Model

OU-ICT-WADP-04- L4-V1- Process Forms

OU-ICT-WADP-05- L4-V1- Customize UI

OU-ICT-WADP-06- L4-V1 - Apply User Management

OU-ICT-WADP-07- L4-V1 - Create API using Django REST Framework

OU-ICT-WADP-08-L4- V1 - Create Final project

Steps:

1. **Project Setup:**
    - Create a Django project using django-admin startproject and a new app for the blog using python manage.py startapp blog.
    - Configure the project settings for email (for email verification) and database.
2. **User Authentication:**
    - Use Django's built-in authentication system to allow users to register, log in, and log out.
    - Implement email verification upon user registration to ensure that comments can only be posted by verified users.
3. **Blog Post Model:**
    - Create a Post model with attributes such as title, content, author (foreign key to the user), date created, and status (published/draft).
    - Include a field to store the post's category (optional).
4. **Comment Model:**
    - Create a Comment model with attributes like user (foreign key to the User model), post (foreign key to the Post model), content, and creation date.

- Optionally, add a parent_comment field to support threaded (replies to) comments.
- Implement a status field to track if the comment is pending, approved, or rejected.

5. **Email Verification for Commenting:**
   - Enable email verification for users upon registration.
   - Modify the commenting system so that users must be logged in and have a verified email before posting a comment.
6. **Comment Moderation (Optional):**
   - Add a flag to allow comment moderation. If enabled, comments will be placed in a "pending" status until an admin approves them.
   - Create an admin interface to approve/reject comments.
7. **Notifications:**
   - **New Comment Notifications:** Use Django's email backend (like SendGrid or Mailgun) to notify the post author when a new comment is posted.
   - **Replies to User Comments:** Notify the user when someone replies to their comment.
   - **Comment Approval/Rejection Notifications:** If comments are moderated, notify users when their comment is approved or rejected.
8. **UI/UX Design:**
   - Create templates for the blog homepage (listing posts), individual post pages (with the comment section), and user registration/login.
   - Use Django's template system to render comments and replies under each post.
9. **Admin Panel:**
   - Customize the Django admin panel to manage blog posts, comments, and user moderation.
   - Ensure the admin can approve, reject, or delete comments.
10. **Testing & Debugging:**
    - Test all features (email verification, comment posting, notifications, comment moderation).

- Ensure the comment system works correctly with moderation, replies, and notifications.

11. **Deployment:**
    - Deploy the project on platforms such as Heroku, AWS, or DigitalOcean.
    - Set up the necessary configurations for production, including email handling, static files, and database settings.

Specification Sheet 2.2: Creating the Blog with User Comments

Necessary Tools

Sl. No	Name of Tools	Specification	Unit	Quantity
1	Django	Web framework for Python	N/A	1
2	Python	Programming language for development	N/A	1
3	Visual Studio Code / PyCharm	Code editor/IDE	N/A	1
4	Git	Version control tool	N/A	1
5	Postman	API testing tool (if using APIs for payments)	N/A	1
6	Stripe API / PayPal API	Payment gateway integration	N/A	1

Necessary Equipment

Sl. No	Name of Equipment	Specification	Unit	Quantity
1	Laptop / PC	For development	N/A	1
2	Server (optional)	For deployment and hosting	N/A	1

Necessary Materials

Sl. No	Name of Materials	Specification	Unit	Quantity
1	Internet Connection	For research, API calls, deployment	N/A	1
2	Email Service (Mailgun/SendGrid)	For sending notifications via email	N/A	1
3	Database (SQLite/PostgreSQL)	Database for storing user and product data	N/A	1

Job Sheet-2.3: **Creating To-Do List application**

UoC Cover

OU-ICT-WADP-01- L4-V1 - Enable Django Framework Environment

OU-ICT-WADP-02- L4-V1 - Develop dynamic web pages

OU-ICT-WADP-03- L4-V1 - Use Django Model

OU-ICT-WADP-04- L4-V1- Process Forms

OU-ICT-WADP-05- L4-V1- Customize UI

OU-ICT-WADP-06- L4-V1 - Apply User Management

OU-ICT-WADP-07- L4-V1 - Create API using Django REST Framework

OU-ICT-WADP-08-L4- V1 - Create Final project

Steps:

1. **Project Setup:**
    - Create a Django project and app for the To-Do List application using the following commands:
        - django-admin startproject task_manager
        - python manage.py startapp tasks
    - Set up database configurations (SQLite for local development, PostgreSQL for production).
    - Install any necessary third-party packages, such as django-crispy-forms for better form styling and django-background-tasks for handling notifications.
2. **User Authentication:**
    - Implement Django's built-in authentication system to manage user registration, login, and logout.
    - Enable email activation for user registration. You can use Django's EmailValidator and the django-allauth or django-registration package to handle the email activation process.
3. **Task Model:**
    - Create a Task model that includes fields such as:

        - title (Task name)
        - description (Details of the task)
        - assigned_user (ForeignKey to User model)
        - due_date (DateTimeField)
        - completed (Boolean, default to False)
        - priority (Choice field: low, medium, high)
        - created_at (DateTimeField)
    - Optionally, you can add a reminder_time field to store when the reminder should be triggered.

4. **Task Management:**
    - Allow users to create, edit, and delete tasks.
    - Allow users to assign tasks to other users.
    - Add functionality for users to mark tasks as completed.
    - Implement filtering of tasks by due date, priority, or assigned user.
5. **Reminders and Notifications:**
    - Implement functionality to send email reminders for upcoming tasks.
    - Use Django's EmailMessage or third-party services (e.g., SendGrid or Mailgun) to send reminders to users.
    - Use django-background-tasks or Celery for background processing of notifications and reminders.
    - Send a notification email when tasks are assigned to a user.
    - Send notifications when a task is close to its due date.
6. **UI/UX Design:**
    - Create templates to display tasks in a list view and a detailed view for each task.
    - Use Django forms to create and edit tasks.
    - Add form validation for required fields (like task title, due date, etc.).
    - Implement filtering options for users to view tasks based on status, priority, and user.
    - Use CSS for styling and make the application mobile-friendly.

7. **Admin Panel:**
    - Customize Django's admin panel to manage tasks, users, and task assignments.
    - Add filtering options in the admin panel for easy task management.
    - Provide an option to assign tasks to users directly from the admin interface.
8. **Testing and Debugging:**
    - Test all features to ensure tasks can be created, assigned, and completed correctly.
    - Ensure email notifications are sent properly.
    - Test the task assignment functionality and email reminders.
    - Implement unit tests for the Task model and related views.
9. **Deployment:**
    - Deploy the project to a cloud platform such as Heroku, AWS, or DigitalOcean.
    - Set up email configuration (like Mailgun or SendGrid) in the production environment.
    - Configure static and media files for deployment.

Specification Sheet 2.3: Creating To-Do List application

Necessary Tools

Sl. No	Name of Tools	Specification	Unit	Quantity
1	Django	Web framework for Python	N/A	1
2	Python	Programming language for development	N/A	1
3	Visual Studio Code / PyCharm	Code editor/IDE	N/A	1
4	Git	Version control tool	N/A	1
5	Postman	API testing tool (if using APIs for payments)	N/A	1
6	Stripe API / PayPal API	Payment gateway integration	N/A	1

Necessary Equipment

Sl. No	Name of Equipment	Specification	Unit	Quantity
1	Laptop / PC	For development	N/A	1
2	Server (optional)	For deployment and hosting	N/A	1

Necessary Materials

Sl. No	Name of Materials	Specification	Unit	Quantity
1	Internet Connection	For research, API calls, deployment	N/A	1
2	Email Service (Mailgun/SendGrid)	For sending notifications via email	N/A	1
3	Database (SQLite/PostgreSQL)	Database for storing user and product data	N/A	1

Job Sheet-2.4: **Building the Recipe Sharing Site**

UoC Cover

OU-ICT-WADP-01- L4-V1 - Enable Django Framework Environment

OU-ICT-WADP-02- L4-V1 - Develop dynamic web pages

OU-ICT-WADP-03- L4-V1 - Use Django Model

OU-ICT-WADP-04- L4-V1- Process Forms

OU-ICT-WADP-05- L4-V1- Customize UI

OU-ICT-WADP-06- L4-V1 - Apply User Management

OU-ICT-WADP-07- L4-V1 - Create API using Django REST Framework

OU-ICT-WADP-08-L4- V1 - Create Final project

Steps:

1. **Project Setup:**
    - Start by creating a Django project with django-admin startproject recipe_sharing.
    - Create a new app within the project: python manage.py startapp recipes.
    - Set up a database for the project (SQLite for development, PostgreSQL for production).
2. **User Authentication:**
    - Implement user registration and login using Django's built-in authentication system.
    - Set up email verification to confirm the user's email address upon registration, using packages like django-allauth or django-registration.
3. **Recipe Model:**
    - Create a Recipe model to store recipe details. It should include:
        - title: A string for the recipe name.
        - description: A detailed description of the recipe.
        - ingredients: A text field listing the ingredients.
        - instructions: A text field for the step-by-step instructions.

        - author: A foreign key to the User model.
        - created_at: A DateTimeField indicating when the recipe was created.
        - likes: IntegerField to track the number of likes for each recipe.

4. **Follow System:**
    - Create a Follow model to allow users to follow other users.
    - The model should contain two foreign keys to the User model: follower and following.
    - When a user follows another, they will be notified when the followed user posts a new recipe.
5. **Comment Model:**
    - Create a Comment model to allow users to comment on recipes.
    - Each comment should be linked to a specific Recipe and User, with a content field for the text of the comment and a created_at timestamp.
6. **Like System:**
    - Create a Like model to allow users to like recipes.
    - Each Like should be associated with a specific Recipe and User to ensure a user can only like a recipe once.
7. **Notifications System:**
    - **New Recipe Notifications:** When a user they follow posts a new recipe, notify the followers via email.
    - **Comment Notifications:** Notify the author of a recipe when someone comments on their recipe.
    - **Like Notifications:** Notify users when someone likes their recipe.
8. **UI/UX Design:**
    - Design views for displaying a list of recipes, detailed views for each recipe, and forms for submitting new recipes, comments, and likes.
    - Use Django's templating system to display recipes, comments, and allow users to interact with the site.
    - Design a follow button on user profiles and recipe pages to allow users to follow or unfollow others.
    - Ensure the interface is mobile-responsive and user-friendly.

9. **Admin Panel Customization:**
    - Customize Django's admin panel to allow easy management of users, recipes, comments, and likes.
    - Provide filtering options to easily manage content based on categories or user actions.
10. **Testing and Debugging:**
    - Test the user authentication, email verification, and recipe posting functionalities.
    - Ensure that notifications (for new recipes, comments, and likes) are triggered correctly.
    - Test the follow/unfollow system and verify that notifications are sent to the correct users.
    - Implement unit tests for models (e.g., ensuring like counting and comment association).
11. **Deployment:**
    - Deploy the project to a hosting platform like Heroku, AWS, or DigitalOcean.
    - Configure the production environment with proper email settings (e.g., using SendGrid or Mailgun).
    - Set up static files and media file handling for deployment.

Specification Sheet 2.4: Building the Recipe Sharing Site

Necessary Tools

Sl. No	Name of Tools	Specification	Unit	Quantity
1	Django	Web framework for Python	N/A	1
2	Python	Programming language for development	N/A	1
3	Visual Studio Code / PyCharm	Code editor/IDE	N/A	1
4	Git	Version control tool	N/A	1
5	Postman	API testing tool (if using APIs for payments)	N/A	1
6	Stripe API / PayPal API	Payment gateway integration	N/A	1

Necessary Equipment

Sl. No	Name of Equipment	Specification	Unit	Quantity
1	Laptop / PC	For development	N/A	1
2	Server (optional)	For deployment and hosting	N/A	1

Necessary Materials

Sl. No	Name of Materials	Specification	Unit	Quantity
1	Internet Connection	For research, API calls, deployment	N/A	1
2	Email Service (Mailgun/SendGrid)	For sending notifications via email	N/A	1
3	Database (SQLite/PostgreSQL)	Database for storing user and product data	N/A	1

Learning Outcome 3: Deploy project

Assessment Criteria	1. Create a hosting account on preferred platform 2. Preparation for hosting is performed 3. Django project is uploaded 4. Project is enabled
Conditions and Resources	1. Real or simulated workplace 2. CBLM 3. Handouts 4. Laptop 5. Multimedia Projector 6. Paper, Pen, Pencil, Eraser 7. Internet facilities 8. White board and marker 9. Audio Video Device
Contents	1. Preferred platform a. Pythonanywhere b. Linode c. Heroku d. GitHub 2. Preparation for hosting a. Hosting environment is setup b. Settings are changed on Local settings.py c. Change local settings.py d. Connect via SSH e. Install pip, Apache, mod-wsgi f. Create software repository g. Create virtual environment h. Install dependencies i. Change remote settings.py j. Apache config file k. Upload database l. Change ownership/permissions databas
Activities/job/Task	1. Django Quiz Project Deployment
Training Methods	1. Discussion 2. Presentation 3. Demonstration 4. Guided Practice 5. Individual Practice 6. Project Work 7. Problem Solving 8. Brainstorming

Assessment Methods	Assessment methods may include but not limited to 1. Written Test 2. Demonstration 3. Oral Questioning 4. Portfolio

Learning Experience 3: Deploy project

In order to achieve the objectives stated in this learning guide, you must perform the learning steps below. Beside each step are the resources or special instructions you will use to accomplish the corresponding activity.

Learning Activities	Recourses/Special Instructions
1. Trainee will ask the instructor about the learning materials	1. Instructor will provide the learning materials 'Deploy project'
2. Read the Information sheet and complete the Self Checks & Check answer sheets on "Deploy project"	2. Read Information sheet 3: Deploy project 3. Answer Self-check 3: Deploy project 4. Check your answer with Answer key 3: Deploy project
3. Read the Job/Task Sheet and Specification Sheet and perform job/Task	5. Job Sheet-3: Django Quiz Project Deployment

Information sheet 3: Deploy project

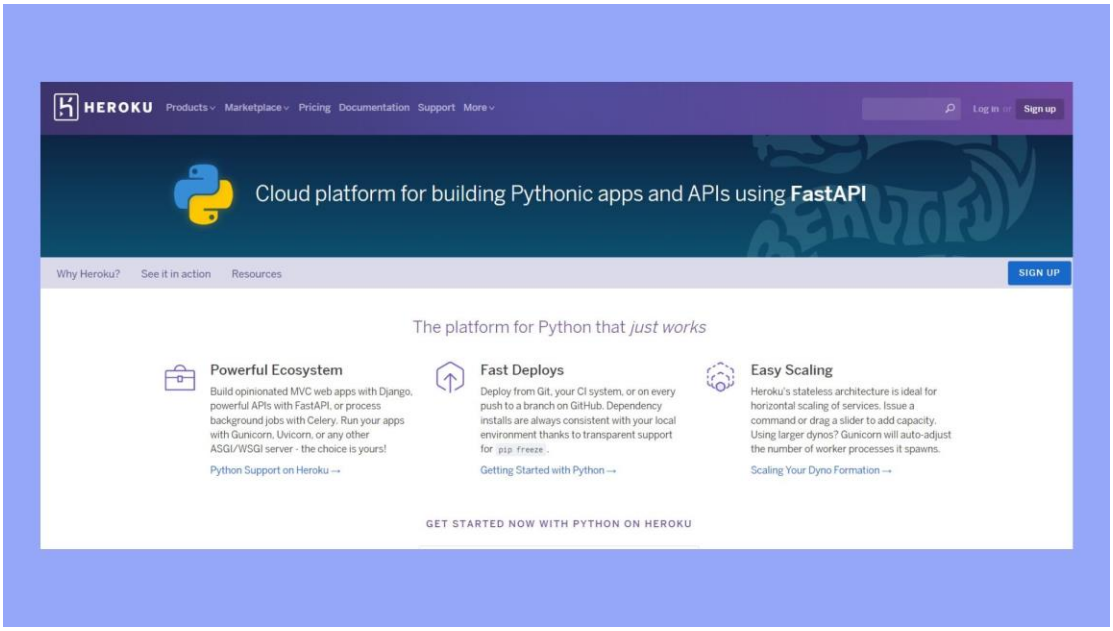
Learning Objective:

After completion of this Information sheet , the learners will be able to explain, define and interpret the following contents:

3.1. Create a hosting account on preferred platform
3.2. Preparation for hosting is performed
3.3. Django project is uploaded
3.4. Project is enabled

3.1. Create a hosting account on preferred platform

There are several popular platforms suitable for hosting Django applications. Here's a breakdown of some key factors to consider when making your choice:



Factors to Consider:

- **Ease of Use:** How easy is it to set up and deploy your Django project on the platform?
- **Features:** Does the platform offer features like Python support, database integration, and web server management?
- **Scalability:** Can the platform handle your project's growth in terms of traffic and data storage?

- **Cost:** What are the pricing plans and fees associated with hosting your Django project?
- **Support:** Does the platform offer reliable customer support in case you encounter any issues?

Here are some popular Django-friendly hosting platforms to consider:

A. PythonAnywhere

- **Pros:** Beginner-friendly, free tier available, easy deployment process, integrated web development environment.
- **Cons:** Limited resources in the free tier, might not be suitable for high-traffic applications.

B. Heroku

- **Pros:** Popular choice for Django projects, good scalability, easy deployment using Git.
- **Cons:** Can be more expensive for resource-intensive applications, can have limitations on custom configurations.

C. AWS Elastic Beanstalk

- **Pros:** Highly scalable, integrates with other AWS services, offers various pricing models.
- **Cons:** Steeper learning curve compared to other options, requires some AWS knowledge.

D. DigitalOcean:

- **Pros:** Affordable cloud hosting platform, offers good control and flexibility, good for developers familiar with Linux.
- **Cons:** Requires more technical expertise to manage server configuration compared to managed platforms.

E. Render:

- **Pros:** Simple and developer-friendly platform, focused on Python web applications, good for beginners.
- **Cons:** Might be less suitable for complex projects with specific needs.

Choosing the Right Platform:

There's no single "best" platform for everyone. Consider your project's requirements (traffic, complexity), your technical expertise, and budget when making your decision.

Here's a quick recommendation based on your preferences

- **For beginners:** Start with PythonAnywhere or Render for their ease of use and clear Django support.
- **For more experienced users:** Consider Heroku or DigitalOcean for their scalability and flexibility.
- **For enterprise-grade applications:** AWS Elastic Beanstalk provides extensive features and control, but requires more technical knowledge.

Tips:

- Most platforms offer free trials or tiers, allowing you to experiment before committing.
- Research user reviews and comparisons of Django hosting platforms to get a broader perspective.
- Consider factors like uptime guarantees, security features, and data backups offered by the platform.

By carefully evaluating your needs and the available options, you can choose the best hosting platform for your Django quiz project.

3.2. Preparation for hosting

Deployment checklist

The internet is a hostile environment. Before deploying your Django project, you should take some time to review your settings, with security, performance, and operations in mind.

Django includes many security features. Some are built-in and always enabled. Others are optional because they aren't always appropriate, or because they're inconvenient for development. For example, forcing HTTPS may not be suitable for all websites, and it's impractical for local development.

Performance optimizations are another category of trade-offs with convenience. For instance, caching is useful in production, less so for local development. Error reporting needs are also widely different.

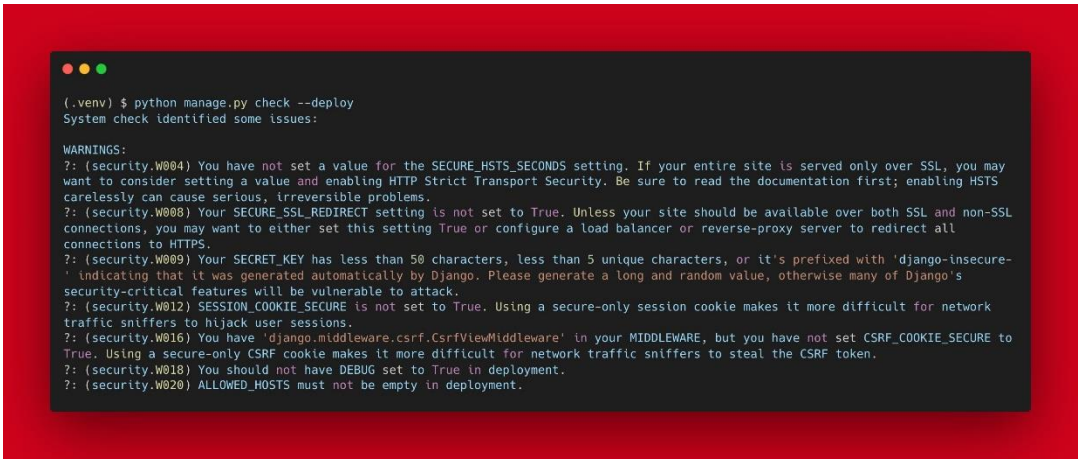
The following checklist includes settings that:

- must be set properly for Django to provide the expected level of security;
- are expected to be different in each environment;

- enable optional security features;
- enable performance optimizations;
- provide error reporting.

Many of these settings are sensitive and should be treated as confidential. If you're releasing the source code for your project, a common practice is to publish suitable settings for development, and to use a private settings module for production.

Run **manage.py check --deploy**



Some of the checks described below can be automated using the **check --deploy** option. Be sure to run it against your production settings file as described in the option's documentation.

Critical settings

SECRET_KEY

The secret key must be a large random value and it must be kept secret.

Make sure that the key used in production isn't used anywhere else and avoid committing it to source control. This reduces the number of vectors from which an attacker may acquire the key.

Instead of hardcoding the secret key in your settings module, consider loading it from an environment variable:

```
import os
SECRET_KEY = os.environ["SECRET_KEY"]
```

or from a file:

```
with open("/etc/secret_key.txt") as f:
    SECRET_KEY = f.read().strip()
```

If rotating secret keys, you may use **SECRET_KEY_FALLBACKS**:

```
import os
SECRET_KEY = os.environ["CURRENT_SECRET_KEY"]
SECRET_KEY_FALLBACKS = [
    os.environ["OLD_SECRET_KEY"],
]
```

Ensure that old secret keys are removed from **SECRET_KEY_FALLBACKS** in a timely manner.

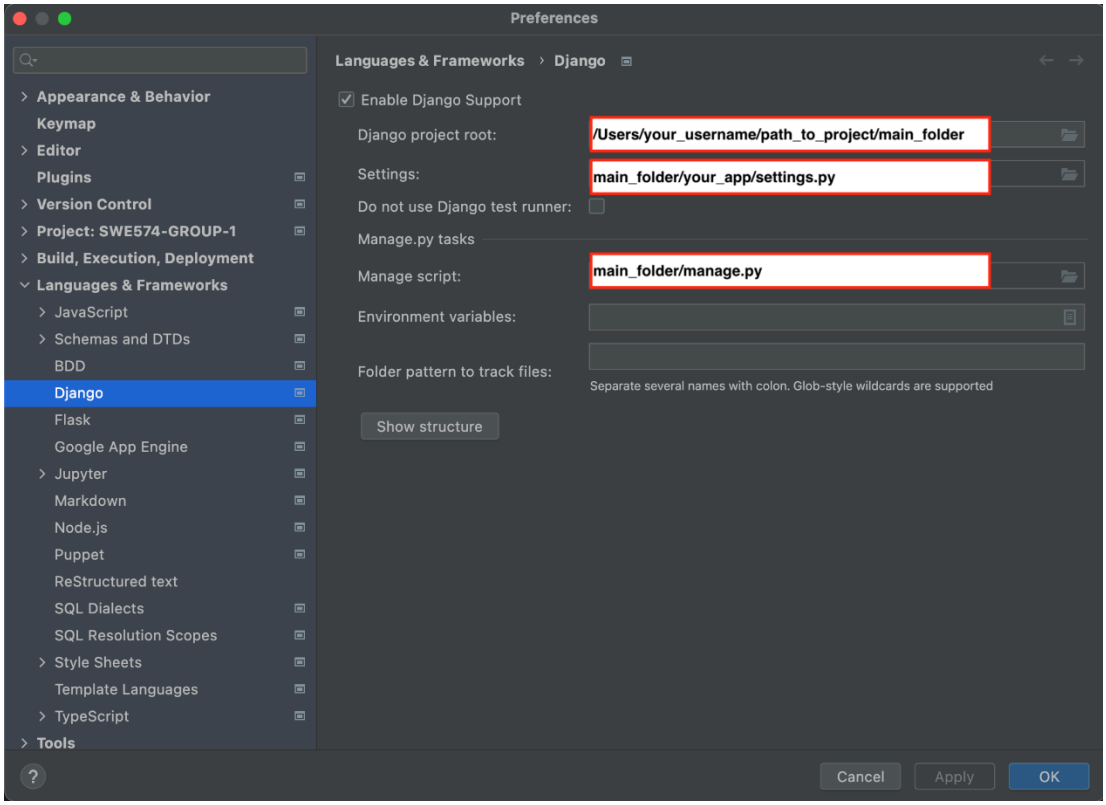
DEBUG

You must never enable debug in production.

You're certainly developing your project with **DEBUG = True**, since this enables handy features like full tracebacks in your browser.

For a production environment, though, this is a really bad idea, because it leaks lots of information about your project: excerpts of your source code, local variables, settings, libraries used, etc.

Environment-specific settings



ALLOWED_HOSTS

When **DEBUG = False**, Django doesn't work at all without a suitable value for **ALLOWED_HOSTS**.

This setting is required to protect your site against some CSRF attacks. If you use a wildcard, you must perform your own validation of the **Host** HTTP header, or otherwise ensure that you aren't vulnerable to this category of attacks.

You should also configure the web server that sits in front of Django to validate the host. It should respond with a static error page or ignore requests for incorrect hosts instead of forwarding the request to Django. This way you'll avoid spurious errors in your Django logs (or emails if you have error reporting configured that way). For example, on nginx you might set up a default server to return "444 No Response" on an unrecognized host

```
server {
    listen 80 default_server;
    return 444;
}
```

CACHES

If you're using a cache, connection parameters may be different in development and in production. Django defaults to per-process local-memory caching which may not be desirable.

Cache servers often have weak authentication. Make sure they only accept connections from your application servers.

DATABASES

Database connection parameters are probably different in development and in production.

```
SECRET_KEY = 'your-django-secret-key'
```

```
DATABASES = {
      'default': {
        'ENGINE': 'your-database-engine name',
        'NAME': 'database-name',
        'USER': 'database-username',
        'PASSWORD': 'database-password',
        'HOST': 'database-host'
        'PORT': '5432',
    }
}
```

Database passwords are very sensitive. You should protect them exactly like **SECRET_KEY**.

For maximum security, make sure database servers only accept connections from your application servers.

If you haven't set up backups for your database, do it right now!

EMAIL_BACKEND and related settings

If your site sends emails, these values need to be set correctly.

By default, Django sends email from webmaster@localhost and root@localhost. However, some mail providers reject email from these addresses. To use different sender addresses, modify the **DEFAULT_FROM_EMAIL** and **SERVER_EMAIL** settings.

STATIC_ROOT and STATIC_URL

Static files are automatically served by the development server. In production, you must define a **STATIC_ROOT** directory where **collectstatic** will copy them.

MEDIA_ROOT and MEDIA_URL

Media files are uploaded by your users. They're untrusted! Make sure your web server never attempts to interpret them. For instance, if a user uploads a **.php** file, the web server shouldn't execute it.

Now is a good time to check your backup strategy for these files.

HTTPS

Any website which allows users to log in should enforce site-wide HTTPS to avoid transmitting access tokens in clear. In Django, access tokens include the login/password, the session cookie, and password reset tokens. (You can't do much to protect password reset tokens if you're sending them by email.)

Protecting sensitive areas such as the user account or the admin isn't sufficient, because the same session cookie is used for HTTP and HTTPS. Your web server must redirect all HTTP traffic to HTTPS, and only transmit HTTPS requests to Django.

Once you've set up HTTPS, enable the following settings.

CSRF_COOKIE_SECURE

Set this to **True** to avoid transmitting the CSRF cookie over HTTP accidentally.

SESSION_COOKIE_SECURE

Set this to **True** to avoid transmitting the session cookie over HTTP accidentally.

Performance optimizations

Setting **DEBUG = False** disables several features that are only useful in development. In addition, you can tune the following settings.

Sessions

Consider using cached sessions to improve performance.

If using database-backed sessions, regularly clear old sessions to avoid storing unnecessary data.

CONN_MAX_AGE

Enabling persistent database connections can result in a nice speed-up when connecting to the database accounts for a significant part of the request processing time.

This helps a lot on virtualized hosts with limited network performance.

TEMPLATES

Enabling the cached template loader often improves performance drastically, as it avoids compiling each template every time it needs to be rendered.
When **DEBUG = False**, the cached template loader is enabled automatically.

Error reporting

By the time you push your code to production, it's hopefully robust, but you can't rule out unexpected errors. Thankfully, Django can capture errors and notify you accordingly.

LOGGING

Review your logging configuration before putting your website in production, and check that it works as expected as soon as you have received some traffic.

See Logging for details on logging.

ADMINS and MANAGERS

ADMINS will be notified of 500 errors by email.

MANAGERS will be notified of 404 errors. **IGNORABLE_404_URLS** can help filter out spurious reports.

See How to manage error reporting for details on error reporting by email.

Error reporting by email doesn't scale very well

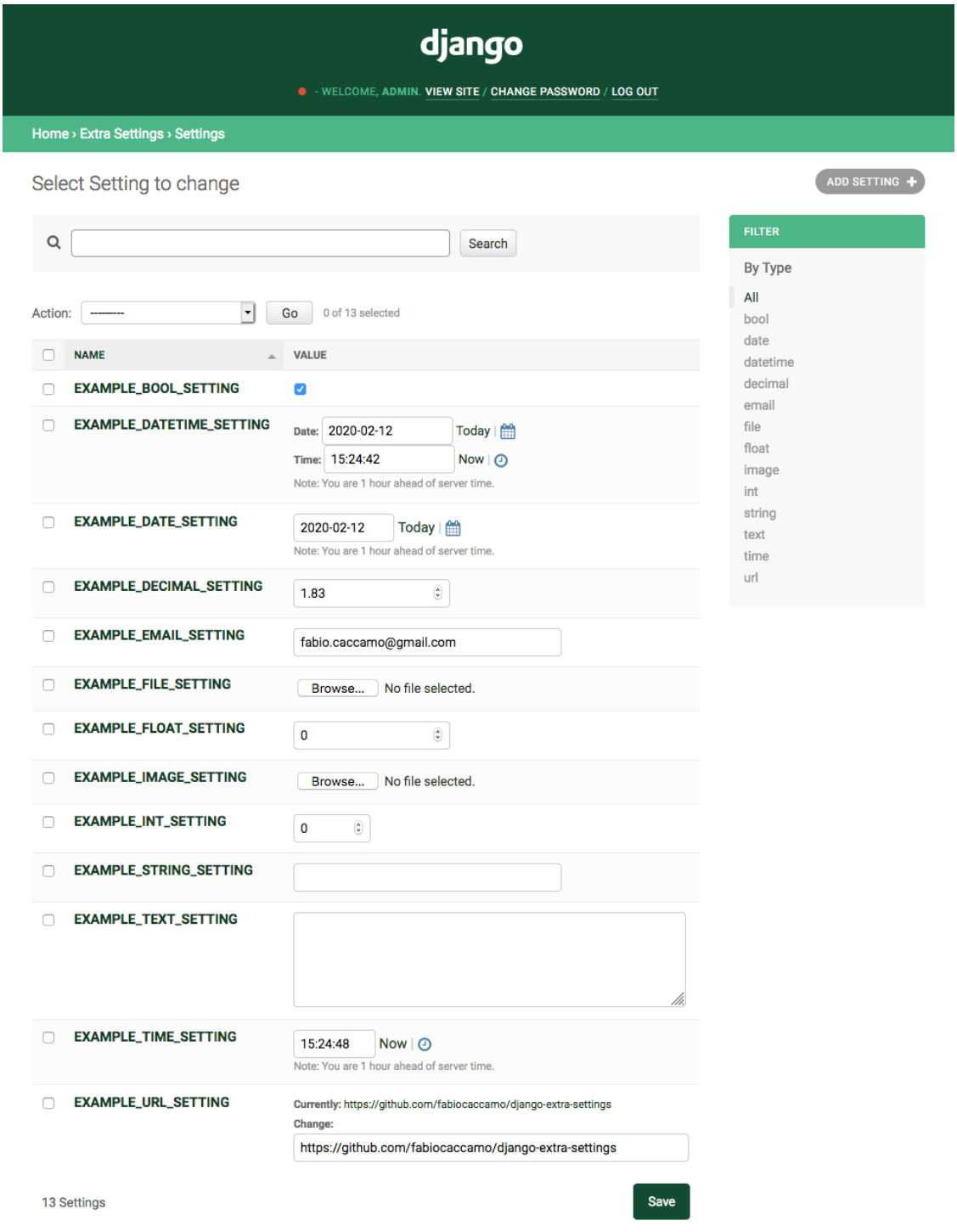
Consider using an error monitoring system such as Sentry before your inbox is flooded by reports. Sentry can also aggregate logs.

Customize the default error views

Django includes default views and templates for several HTTP error codes. You may want to override the default templates by creating the following templates in your root template directory: **404.html**, **500.html**, **403.html**, and **400.html**. The default error views that use these templates should suffice for 99% of web applications, but you can customize them as well.

Once you've chosen a hosting platform and created an account, here are the key steps involved in preparing your Django project for deployment:

A. Configure Django Settings:



- Update your project's settings.py file with production-specific settings. This might include:

- **Database credentials:** Replace development database settings with the actual credentials provided by your hosting platform.
- **Secret key:** Use a strong and secure secret key for production (different from your development key).
- **Static files and media:** Configure how static files (CSS, JavaScript) and media files (uploaded content) are served in production. You might need to use the platform's specific settings or commands to manage these files.
- **Debug mode:** Set DEBUG to False for production to disable debugging features.

B. Collect Static Files:

- Use the python manage.py collectstatic command to collect all static files from your project's static folders into a single directory (usually static). This ensures your static files are served correctly when deployed.

C. Create a Production Requirements File:

- Create a requirements file (e.g., requirements.txt) that lists all the Python packages your project depends on. Use pip freeze to generate a starting point and then review and adjust the listed packages. This file ensures your project has the necessary dependencies installed on the hosting platform.

D. Virtual Environment (Optional):

- If you've been using a virtual environment for development, you might need to recreate it on the hosting platform or use the platform's virtual environment management features. Ensure all dependencies are installed within the chosen virtual environment.

F. Codebase Preparation:

- Double-check your codebase for any development-specific configurations or debugging statements that might not be suitable for production.
- Consider using a code linter or static code analysis tool to identify potential errors or vulnerabilities before deployment.

G. Version Control (Optional):

- If you're using Git for version control, push your latest code to your remote repository (e.g., GitHub) before deployment. This creates a backup and allows you to revert to previous versions if needed.

H. Deployment Process:

- The specific deployment process will vary depending on your chosen hosting platform. Most platforms offer deployment options like Git integration, manual file uploads, or command-line tools. Refer to the platform's documentation for detailed instructions on deploying your Django project.

Tips:

- Consider using a deployment tool like Fabric or Ansible to automate the deployment process for easier management, especially if you plan on frequent updates.
- Most platforms offer logging and monitoring features. Utilize these tools to track your application's performance and identify any issues after deployment.
- Set up backups of your database and codebase on the hosting platform or your own server for disaster recovery purposes.

By following these preparation steps, you can ensure your Django quiz project is ready for a smooth and successful deployment on your chosen hosting platform.

3.3. Upload Django project

The exact process for uploading your Django project depends on the hosting platform you've chosen. Here's a general outline with some platform-specific examples:

Common Methods for Uploading Django Projects

- **Git Integration:** Many platforms integrate with Git repositories like GitHub. You can link your project to a remote repository and push your code directly to deploy it on the platform. (e.g., Heroku, Render)
- **Manual Upload:** Some platforms allow you to upload your project files (usually a compressed archive) through a web interface or FTP (File Transfer Protocol). (e.g., DigitalOcean with manual configuration)
- **Command-Line Tools:** Certain platforms offer command-line tools or frameworks that automate the deployment process. These tools might require configuration and knowledge of command-line interfaces. (e.g., AWS Elastic Beanstalk with tools like EB CLI)

Here are some specific examples for popular platforms

- **PythonAnywhere:** You can deploy using Git integration or by uploading a zipped project directory.
- **Heroku:** Heroku integrates seamlessly with Git. Push your code to your remote repository, and Heroku automatically detects and deploys the changes.
- **AWS Elastic Beanstalk:** This platform requires more configuration. You can use the EB CLI tool to package your project, configure settings, and deploy it to an Elastic Beanstalk environment.

- **DigitalOcean:** DigitalOcean offers a more manual approach. You can upload your project files via FTP or the web interface and then configure settings like web server and database manually.

Important Considerations

- Always refer to your hosting platform's specific documentation for detailed instructions on the deployment process.
- Make sure you've prepared your project for deployment as mentioned earlier (configure settings, collect static files, create requirements file).
- Double-check your database credentials and other sensitive information in your settings file before deployment.

Once Uploaded, What's Next?

- After uploading your project, the hosting platform will typically perform some setup tasks and launch your Django application.
- You'll be provided with a URL or IP address to access your deployed application.
- Test your application thoroughly to ensure everything is functioning correctly in the production environment.

By following these steps and consulting your platform's documentation, you can successfully upload your Django quiz project and make it accessible online.

3.4. Enable Project

The term "enabled" might have different meanings depending on your hosting platform. Here are two interpretations and how they might apply to your Django project:

A. Application Startup:

On some platforms, deploying your Django project might not automatically start the application server. You might need to take an extra step to "enable" or launch the application.

Here's a breakdown of possibilities:

- **Platform-Managed Startup:** Some platforms like Heroku or Render automatically handle application startup after deployment. Once you upload your project, the platform takes care of starting the web server and making your application accessible.
- **Manual Startup:** Certain platforms, especially those offering more server control (e.g., DigitalOcean), might require you to manually start the web server (e.g., using commands like sudo service gunicorn start) after uploading your project.

How to Check and Enable (if necessary):

- Consult your hosting platform's documentation for specific instructions on starting the application server.
- Look for sections related to deployment or application management. They might mention commands or steps required to launch your Django project.
- If you're unsure, reach out to your platform's customer support for clarification.

B. Web Server Configuration (Optional):

In some less managed hosting environments, you might need to configure the web server (e.g., Apache, Nginx) to recognize and serve your Django application. This configuration typically involves creating or modifying configuration files on the server.

This is less common with platforms offering managed Django deployment. However, if you're using a platform with more manual server control, consult their documentation to see if any web server configuration is required for your Django project.

Tips:

- If you're unsure about the specific steps for enabling your project, it's always best to refer to your hosting platform's documentation.
- Search the platform's knowledge base or community forums for guidance related to Django deployment.
- Consider using a managed Django hosting platform for a simpler and more automated deployment experience, especially if you're new to server administration.

By understanding these different interpretations and checking your platform's documentation, you can ensure your Django quiz project is properly enabled and accessible online.

How to upgrade Django to a newer version

While it can be a complex process at times, upgrading to the latest Django version has several benefits:

- New features and improvements are added.
- Bugs are fixed.
- Older version of Django will eventually no longer receive security updates. (see Supported versions).

- Upgrading as each new Django release is available makes future upgrades less painful by keeping your code base up to date.

Here are some things to consider to help make your upgrade process as smooth as possible.

Required Reading

If it's your first time doing an upgrade, it is useful to read the guide on the different release processes.

Afterward, you should familiarize yourself with the changes that were made in the new Django version(s):

- Read the release notes for each 'final' release from the one after your current Django version, up to and including the version to which you plan to upgrade.
- Look at the deprecation timeline for the relevant versions.

Pay particular attention to backwards incompatible changes to get a clear idea of what will be needed for a successful upgrade.

If you're upgrading through more than one feature version (e.g. 2.0 to 2.2), it's usually easier to upgrade through each feature release incrementally (2.0 to 2.1 to 2.2) rather than to make all the changes for each feature release at once. For each feature release, use the latest patch release (e.g. for 2.1, use 2.1.15).

The same incremental upgrade approach is recommended when upgrading from one LTS to the next.

Dependencies

In most cases it will be necessary to upgrade to the latest version of your Django-related dependencies as well. If the Django version was recently released or if some of your dependencies are not well-maintained, some of your dependencies may not yet support the new Django version. In these cases you may have to wait until new versions of your dependencies are released.

Resolving deprecation warnings

Before upgrading, it's a good idea to resolve any deprecation warnings raised by your project while using your current version of Django. Fixing these warnings before upgrading ensures that you're informed about areas of the code that need altering.

In Python, deprecation warnings are silenced by default. You must turn them on using the **-Wa** Python command line option or the **PYTHONWARNINGS** environment variable. For example, to show warnings while running tests:

□/□ □

$ python -Wa manage.py test

If you're not using the Django test runner, you may need to also ensure that any console output is not captured which would hide deprecation warnings. For example, if you use pytest:

$ PYTHONWARNINGS=always pytest tests --capture=no

Resolve any deprecation warnings with your current version of Django before continuing the upgrade process.

Third party applications might use deprecated APIs in order to support multiple versions of Django, so deprecation warnings in packages you've installed don't necessarily indicate a problem. If a package doesn't support the latest version of Django, consider raising an issue or sending a pull request for it.

Installation

Once you're ready, it is time to install the new Django version. If you are using a **virtual environment** and it is a major upgrade, you might want to set up a new environment with all the dependencies first.

If you installed Django with pip, you can use the **--upgrade** or **-U** flag:

□/□ □

$ python -m pip install -U Django

Testing

When the new environment is set up, run the full test suite for your application. Again, it's useful to turn on deprecation warnings on so they're shown in the test output (you can also use the flag if you test your app manually using **manage.py runserver**):

□/□ □

$ python -Wa manage.py test

After you have run the tests, fix any failures. While you have the release notes fresh in your mind, it may also be a good time to take advantage of new features in Django by refactoring your code to eliminate any deprecation warnings.

Deployment

When you are sufficiently confident your app works with the new version of Django, you're ready to go ahead and deploy your upgraded Django project.

If you are using caching provided by Django, you should consider clearing your cache after upgrading. Otherwise you may run into problems, for example, if you are caching pickled objects as these objects are not guaranteed to be pickle-compatible across Django versions. A past instance of incompatibility was caching pickled **HttpResponse** objects, either directly or indirectly via the **cache_page()** decorator.

Self-Check Sheet 3: Deploy project

1. What factors are considered when choosing a hosting platform for a Django project? (Choose ALL that apply)
Answer:

a) Ease of Use
b) Features (Python support, database integration, web server management)
c) Scalability (handling traffic and data storage growth)
d) Cost
e) Security

2. What is the purpose of running manage.py check --deploy before deploying a Django project?
Answer:

a) To collect static files.
b) To identify potential security vulnerabilities and configuration issues for production deployment.
c) To create a virtual environment.
d) To start the development server.

3. Which of the following statements is TRUE about the SECRET_KEY setting in Django?
Answer:

a) It can be the same value used in development and production.
b) It should be a strong random value and kept secret.
c) It is used to control caching behavior.
d) It defines the database connection information.

4. Why is it important to set DEBUG to False in the production settings of a Django project?
Answer:

a) To enable caching for improved performance.
b) To disable error reporting features.
c) To prevent unauthorized access to sensitive information (like source code and settings).
d) To simplify debugging of production issues.

5. What does "enabling" a Django project typically mean on a hosting platform? (Choose the MOST likely interpretation)
Answer:
a) Installing all required Python libraries.
b) Starting the application server to make the project accessible.
c) Upgrading the Django version to the latest release.

Answer Key 3: Deploy project

1. What factors are considered when choosing a hosting platform for a Django project? (Choose ALL that apply)
Answer:

a) Ease of Use
b) Features (Python support, database integration, web server management)
c) Scalability (handling traffic and data storage growth)
d) Cost
e) Security **(Not mentioned in the excerpt, but likely a significant factor)**

2. What is the purpose of running manage.py check --deploy before deploying a Django project?
Answer:

a) To collect static files.
b) To identify potential security vulnerabilities and configuration issues for production deployment. **(Correct)**
c) To create a virtual environment.
d) To start the development server.

3. Which of the following statements is TRUE about the SECRET_KEY setting in Django?
Answer:

a) It can be the same value used in development and production. **(Incorrect)**
b) It should be a strong random value and kept secret. **(Correct)**
c) It is used to control caching behavior.
d) It defines the database connection information.

4. Why is it important to set DEBUG to False in the production settings of a Django project?
Answer:

a) To enable caching for improved performance.
b) To disable error reporting features.
c) To prevent unauthorized access to sensitive information (like source code and settings). **(Correct)**
d) To simplify debugging of production issues.

5. What does "enabling" a Django project typically mean on a hosting platform? (Choose the MOST likely interpretation)
Answer:
a) Installing all required Python libraries.
b) Starting the application server to make the project accessible. **(Correct)**
c) Upgrading the Django version to the latest release.

Job Sheet-3: **Django Quiz Project Deployment**

UoC Cover

OU-ICT-WADP-01- L4-V1 - Enable Django Framework Environment

OU-ICT-WADP-02- L4-V1 - Develop dynamic web pages

OU-ICT-WADP-03- L4-V1 - Use Django Model

OU-ICT-WADP-04- L4-V1- Process Forms

OU-ICT-WADP-05- L4-V1- Customize UI

OU-ICT-WADP-06- L4-V1 - Apply User Management

OU-ICT-WADP-07- L4-V1 - Create API using Django REST Framework

OU-ICT-WADP-08-L4- V1 - Create Final project

Working Procedure / Steps

Preparation Checklist

1. Choose a Hosting Platform:

- Consider factors like ease of use, features, scalability, cost, and support.
- Popular options include:
    - PythonAnywhere (beginner-friendly, free tier)
    - Heroku (popular, good scalability)
    - AWS Elastic Beanstalk (highly scalable, complex)
    - DigitalOcean (affordable, requires technical expertise)
    - Render (simple, beginner-friendly)
- Research and choose the platform based on your project needs and skill level.

2. Secure Your Django Project:

- Review security settings in your settings.py file.
- Set a strong and unique SECRET_KEY.
- Disable DEBUG mode for production.
- Configure ALLOWED_HOSTS appropriately.

3. Environment-Specific Settings:

- Update database credentials, cache settings, and static/media file handling for production.
- Consider HTTPS for secure logins and data transmission.

4. Performance Optimizations:

- Set DEBUG to False to disable development features.

- Use cached sessions and templates for improved performance.
- Configure database connection pooling (optional).

5. Error Reporting:

- Review logging configuration and ensure it works.
- Set up email notifications for errors (consider using a service like Sentry).
- Customize error views for a user-friendly experience.

Deployment Process

1. Configure Django Settings:

- Update settings.py with production-specific settings.

2. Collect Static Files:

- Use python manage.py collectstatic to gather static files.

3. Create Requirements File:

- Generate a requirements.txt file listing project dependencies using pip freeze.

4. Virtual Environment (Optional):

- Recreate your virtual environment on the hosting platform if needed.

5. Codebase Preparation:

- Double-check code for development-specific configurations or debugging statements.
- Consider using a linter or static code analysis tool to identify potential issues.

6. Version Control (Optional):

- Push your latest code to a remote repository (e.g., GitHub) before deployment.

7. Upload Your Project:

- Follow your platform's specific instructions for uploading your project files.
- Common methods include Git integration, manual upload, and command-line tools.

8. Deployment Process:

- Refer to your hosting platform's documentation for detailed deployment steps.
- Consider using deployment tools like Fabric or Ansible for automation.

9. Post-Deployment Tasks:

- Test your application thoroughly in the production environment.
- Set up backups of your codebase and database for disaster recovery.
- Monitor application performance and logs for any issues.

Specification Sheet 3: Django Quiz Project Deployment

Preparation Checklist

1. Choose a Hosting Platform

Considerations:

- **Ease of Use**: How straightforward is the platform to set up and manage?
- **Features**: Does it support the required technologies and services?
- **Scalability**: Can it handle increased traffic and data load?
- **Cost**: What are the pricing options and tiers?
- **Support**: What kind of support and documentation is available?

Popular Options:

- **PythonAnywhere**: Beginner-friendly with a free tier.
- **Heroku**: Popular, scalable, with various add-ons.
- **AWS Elastic Beanstalk**: Highly scalable but complex.
- **DigitalOcean**: Affordable with flexible options but requires technical expertise.
- **Render**: Simple and beginner-friendly.

Action:

- Research and select the platform that best fits your project needs and skill level.

2. Secure Your Django Project

Actions:

- **Review Security Settings**: Ensure your settings.py file is configured for security.
- **SECRET_KEY**: Set a strong, unique key.
- **DEBUG Mode**: Set DEBUG = False for production.
- **ALLOWED_HOSTS**: Configure this list with the domain names or IP addresses allowed to access your application.

3. Environment-Specific Settings

Actions:

- **Database Credentials**: Update with production database details.
- **Cache Settings**: Configure caching settings appropriate for production.
- **Static/Media Files**: Set up proper handling for static and media files.
- **HTTPS**: Configure HTTPS for secure data transmission.

4. Performance Optimizations

Actions:

- **Disable Debug Mode**: Ensure DEBUG = False.
- **Caching**: Use cached sessions and templates to enhance performance.
- **Database Connection Pooling**: Optional, but consider configuring it for high-performance applications.

5. Error Reporting

Actions:

- **Logging Configuration**: Review and test logging settings to ensure error capture.
- **Email Notifications**: Set up email notifications for error reporting (consider services like Sentry).
- **Custom Error Views**: Create user-friendly error pages (e.g., 404, 500 pages).

Deployment Process

1. Configure Django Settings

Actions:

- **Update settings.py**: Add production-specific settings such as database configurations, security settings, and static file configurations.

2. Collect Static Files

Actions:

- **Command**: Run python manage.py collectstatic to gather static files in the STATIC_ROOT directory.

3. Create Requirements File

Actions:

- **Generate**: Run pip freeze > requirements.txt to create a file listing all dependencies.

4. Virtual Environment (Optional)

Actions:

- **Recreate**: Set up a new virtual environment on the hosting platform if needed.

5. Codebase Preparation

Actions:

- **Review Code**: Check for any development-specific configurations or debugging statements.
- **Linter/Static Code Analysis**: Use tools to identify potential issues.

6. Version Control (Optional)

Actions:

- **Push Code**: Commit and push your latest changes to a remote repository (e.g., GitHub) before deployment.

7. Upload Your Project

Actions:

- **Follow Platform Instructions**: Upload project files as per the hosting platform's guidelines.
- **Common Methods**: Git integration, manual upload, command-line tools.

8. Deployment Process

Actions:

- **Platform Documentation**: Refer to specific deployment instructions provided by your hosting platform.
- **Automation Tools**: Consider using tools like Fabric or Ansible for deployment automation.

9. Post-Deployment Tasks

Actions:

- **Testing**: Conduct thorough testing in the production environment.
- **Backups**: Set up regular backups for your codebase and database.
- **Monitoring**: Monitor application performance and review logs for any issues.

Tools and Equipment

Hosting Platforms

- **PythonAnywhere**
- **Heroku**
- **AWS Elastic Beanstalk**
- **DigitalOcean**
- **Render**

Development and Deployment Tools

- **Django Documentation**: Django Documentation
- **Git**: For version control.
- **Virtual Environment Tools**: venv or virtualenv.
- **CI/CD Tools** (Optional): Jenkins, GitHub Actions, GitLab CI.
- **Error Tracking**: Sentry or similar services.
- **Automation Tools** (Optional): Fabric, Ansible.

Performance and Security Tools

- **Caching Libraries**: Redis, Memcached.
- **Database Connection Pooling**: Use appropriate libraries and configurations.
- **HTTPS Configuration**: SSL certificates from providers like Let's Encrypt.

Reference

1. https://forum.djangoproject.com/
2. https://www.webforefront.com/django/

Review of Competency

Below is yourself assessment rating for module “Create Final project”

Assessment of performance Criteria	Yes	No
SRS is created from the assigned domain-specific scenario	☐	☐
Models are designed and class diagram is created	☐	☐
User Interface (UI) is designed	☐	☐
GitHub repo is managed	☐	☐
All functionalities are implemented and tested	☐	☐
Database and classes are designed following flow diagram	☐	☐
User Interface (UI) is designed	☐	☐
All functionalities are implemented and tested	☐	☐
Create a hosting account on preferred platform	☐	☐
Preparation for hosting is performed	☐	☐
Django project is uploaded	☐	☐
Project is enabled	☐	☐

I now feel ready to undertake my formal competency assessment.

Signed:

Date:

Development of CBLM

The Competency based Learning Material (CBLM) of 'Create Final project' (Occupation: Web Application Development with Python , Level-4) for National Skills Certificate is developed by NSDA with the assistance of SAMAHAR Consultants Ltd.in the month of June, 2024 under the contract number of package SD-9C dated 15th January 2024.

SL No.	Name and Address	Designation	Contact Number
1	Khan Mohammad Mahmud Hasan	Writer	Cell: 01714087897 Email: kmmhasan@gmail.com
2	A K M Mashuqur Rahman Mazumder	Editor	Cell: 01676323576 Email : mashuq.odelltech@odell.com.bd
3	Khan Mohammad Mahmud Hasan	Co-Ordinator	Cell: 01714087897 Email: kmmhasan@gmail.com
4	Md. Saif Uddin	Reviewer	Cell:01723004419 Email: engrbd.saif@gmail.com