SOFTWARE ENGINEERING II

CKB – CodeKataBattle
Implementation and Test Document

Version 1.0

Polito Attilio

Rigione Pisone Raimondo

Soricelli Francesco

https://github.com/Attilioap/PolitoRigionePisoneSoricelli

February 4, 2024

CONTENTS

LIST OF TABLE

1. INTRODUCTION

1.1 PURPOSE

The objective of this document is to elucidate the implementation and testing of the CKB application. Following discussions on its design in the RASD and DD documents, we will present a detailed analysis of the application's core functionalities. We will delve into the decisions made regarding frameworks and programming languages, providing insights into the entire implementation process. Additionally, various test cases will be defined and executed to ensure the application's proper alignment with the requirements outlined in the RASD document.

1.2 DEFINITIONS, ACRONYMS, ABBREVIATIONS

1.2.1 ACRONYMS

Acronym	**Description**
DD	Design Document
RASD	Requirement Analysis and Specificatio Document
API	Application Programming Interface
CKB	CodeKataBattle
TDD	Test-Driven Development
DBMS	Database Management System
HTTP	Hypertext Transfer Protocol
MVC	Model-View-Controller
ORM	Object-Relational Mapping
JSON	JavaScript Object Notation
URL	Uniform Resource Locator
CSS	Cascading Style Sheets
HTML	HyperText Markup Language
SQL	Structured Query Language
DTO	Data Transfer Object
TCP-IP	Transmission control protocol – Internet protocol

Tabella 1:Table of acronyms

1.2.2 ABBREVIATIONS

Abbreviation	**Description**
Rn	n-th functional requirement

Tabella 2:Table of abbrevations

1.3 REVISION HISTORY

Version 1.0 - 04/02/2024

1.4 REFERENCE DOCUMENTS

1- I&T Assignment A.Y. 2022-2023;
2- Course slides;
3- ITDs of the previous academic years;
4- Django documentation: https://docs.djangoproject.com/en/5.0/;
5- Ngrock documentation: https://ngrok.com/docs/;
6- GitHub: https://docs.github.com/en;

2. DEVELOPMENT

2.1 IMPLEMENTED FUNCTIONALITIES

- **User Registration:** During the registration phase, users, whether they are students or educators, are prompted to provide essential information, including personal details and additional profile information. The registration process includes a user-friendly toggle that enables individuals to easily identify themselves as educators. It is crucial to use a unique email address that is not already associated with an existing account. Upon successful completion of the registration, students and educators are redirected to their respective main dashboards.

- **Login:** The login process for the CKB application is uniform for both students and educators, relying on the previously set username and password during registration. Users enter their credentials, and if correct, they gain access to the various functionalities offered by the platform. The login procedure is designed to be intuitive and secure, ensuring a consistent experience for both user categories. Upon successful login, users are directed to the main screen of the application.

➢ Educator Functionalities

- **Creation of New Tournaments:** Educators can make use of the feature to create new tournaments within the CKB platform to further customize programming competitions. During this procedure, it is required to provide parameters such as the tournament name, registration deadline, ending date, and a brief description of the tournament.
- **Manage Tournament:** Educators can create and manage programming tournaments on the CKB platform. When creating a tournament, they need to specify the name, description, registration deadline, and end date. It's also necessary to upload the specific CodeKata file for the tournament. Educators have the flexibility to set additional configurations for scoring, allowing them to tailor the evaluation criteria to the specific needs of the tournament.
- **Assign Permission:** This functionality enables educators to delegate permissions for the management of specific tournaments to other educators already registered on the CKB platform. To grant these permissions, simply mention the username of the educator to whom you want to assign the authorization. The system verifies if the provided username corresponds to a valid educator, ensuring accuracy and control in the permission assignment process. This feature streamlines the delegation of tournament management tasks among educators, enhancing overall efficiency in organizing programming competitions.
- **Visualize Tournament Status:** Educators can use the Tournament Status View function to monitor and analyze activities and performance during a programming competition. This view

provides real-time information on ongoing and completed battles within the tournament, along with the tournament leaderboard displaying scores.

- **Visualize Battle Status:** The "View Battle Status" function allows you to monitor the battles currently underway in a tournament, providing an overview of the team rankings and their respective individual students. This view offers a detailed insight into the performance during ongoing battles, enabling educators to assess team strategies and the individual contributions of students.
- **Close a Tournament:** Educators have the ability to conclude a tournament on the CKB platform. This feature empowers educators to make informed decisions about ending a programming competition, marking the conclusion of battles and determining the final results. By leveraging this function, educators can effectively oversee the lifecycle of a tournament, ensuring closure at the most suitable juncture.
- **Evaluation of Students' Performance:** This function allows educators to assess the work of students in concluded battles. After the conclusion of a programming competition, educators can provide a detailed evaluation of the individual performances of students. This evauation scores for each team enrolled in the Battle. The aim is to provide educators with an effective tool to evaluate students' performance after a competition.

➢ Student Functionalities

- **Visualize Tournament Information:** The student can access details about ongoing tournaments they haven't enrolled in through the "View Tournament Information" function. This feature offers essential information such as the starting date, ending date, and a brief description of each tournament. By leveraging this function, students can make informed decisions about their participation in specific tournaments, considering their preferences, schedule, and interests.
- **Enroll in a tournament:** This function allows students to enroll in a specific tournament. By using this function, students can choose the tournament of their interest. Once the registration process is completed, students officially become participants in the selected tournament within the CKB platform.
- **Visualize Tournament Status:** Students can access to various information regarding the tournaments they are enrolled in. This function provides an overview of ongoing battles, battles that students can still join, the tournament leaderboard, and a dedicated section to accept or decline invitations to form teams for battles.
- **Join a Battle:** Students can actively participate in programming competitions by joining an ongoing battle. Using this feature, students can explore available battles, choose one of interest, and enroll to participate. This process encourages direct engagement of students with the programming challenges presented on the CKB platform.
- **Invite a Teammate:** Students can enhance their experience by collaborating with teammates. The "Invite a Teammate" function allows students to extend invitations to their peers to join a specific battle. This feature promotes teamwork and the building of connections among students involved in coding competitions on the CKB platform.

- **Visualize Tournament Status:** This function, on the other hand, allows users to view the overall ranking of the tournament, providing a detailed analysis of the performance of participating students and teams.
- **Visualize Battle Status:** This function allows users to view the ranking of students and their respective teams for a specific battle. Users can access detailed information about the performance of students and teams in this competition.

In order to provide all these features, the functions described in the DD document that are related to these functionalities have been implemented. Their description will be deepened later.

2.2 ADOPTED DEVELOPMENT FRAMEWORKS

For what concerns the application logic tier, it has been implemented through the Django framework, which relies on the Phyton programming language. The Python programming language is widely adopted by developers due to its advantageous features. One of its main qualities is its platform independence, allowing code to run on different operating systems without the need for significant modifications. Unlike other languages, Python makes use of code interpretation, simplifying the development process and making it highly portable. Finally, Django provides a powerful and intuitive ORM (Object-Relational Mapping) system that allows developers to interact with the database using Python objects instead of writing SQL queries directly. This greatly simplifies data management and makes the code more readable.

2.2.1 APPLICATION LOGIC TIER

Django Framework

Django is an open-source web framework for creating robust and scalable web applications using the Python programming language. Widely adopted by developers, Django simplifies the implementation of web applications by handling infrastructure concerns, allowing programmers to concentrate on the application logic. The framework consists of several modules organised into five main components:

- Main Framework: Django's core includes concepts such as models, views and templates. It uses an Object-Relational Mapping (ORM) system to manage the lifecycle of models and facilitate interactions with the database. Dependency injection is used to manage component life cycles.

- Data Access: Django provides a powerful module for data access, facilitating the mapping of Python objects to data retrieved from different sources. This simplifies interaction with databases, and repositories.

- Web: The web component includes the web-servlet module, characterised by a Model-View-Controller (MVC) architecture. This MVC pattern aids in the development of web applications, with a dispatcher handling incoming requests and directing them to the appropriate controller. The controllers, in turn, interact with business logic services to process requests and generate responses.

- AOP (Aspect-Oriented Programming): Django incorporates elements of AOP to provide declarative services for the enterprise, such as declarative transaction management. This allows programmers to define cross-cutting aspects and apply them to the application in a modular and reusable way.

- Testing Module: Django facilitates the testing of its components using popular testing framework. This ensures the reliability and correctness of the developed components.

Within the Django framework, the application logic is effectively managed through the Model-View-Controller architecture. Entities, representing database tables, can be mapped as Django models using the ORM system. These models simplify data retrieval and manipulation, and custom services, annotated as Services, process the data to provide the required functionality.
In the Django architecture, controllers handle HTTP requests from the client layer, identify them by URL paths and categorise them as GET, POST, PUT or DELETE requests using the @RequestMapping annotation. The parameters attached to these requests may be simple variables in the URL paths or data transfer objects (DTOs).

2.2.2 CLIENT TIER

The implementation is integrated into a Django app, facilitating backend interactions and overall application functionality. The Django app encapsulates models representing entities, logic for handling HTTP requests and necessary URL routing for communication with the server. This structured approach enhances the maintainability and scalability of the client-side architecture.

Storyboard

In our application development using Django, the creation and management of user interfaces are streamlined through Django's versatile framework. Graphic elements like buttons and labels seamlessly integrate into Django-based view controllers, employing an intuitive design process. This methodology dynamically generates related variables within controllers or establishes handlers for specific actions tied to these elements.

Through Django's approach to user interface development, values of components easily update with data retrieved from the server or obtained from prior interfaces. Additionally, it simplifies the execution of various actions, such as sending requests to the server, presenting alerts for unexpected situations, or navigating smoothly between different interfaces.

Data Management with ORM

Django's strength in data management is further amplified by its ORM (Object-Relational Mapping) system. This system facilitates the seamless interaction between the application and the database by mapping database tables to Python objects and vice versa. It allows developers to work with database entities using high-level, Pythonic syntax instead of dealing directly with SQL queries. In the application, Django's ORM simplifies complex database operations, such as querying, inserting, updating, and deleting records. This abstraction layer enhances code readability and maintainability, providing a powerful and intuitive way to manage data within the application.

Security Features

Django places a strong emphasis on security, including robust password encryption mechanisms. Passwords are securely stored using strong hashing algorithms, ensuring the confidentiality of user credentials. This security measure significantly enhances protection against unauthorized access and data breaches. Moreover, Django incorporates various security features to safeguard the application against common web vulnerabilities, including cross-site scripting (XSS) and cross-site request forgery (CSRF). These built-in security measures contribute to creating a secure environment for data processing and user interactions.

Data Tier with SQLite

The data tier of the application is realized using Django's integrated system with SQLite for efficient data management. SQLite, a lightweight and serverless relational database, is seamlessly integrated into Django, providing a reliable solution for data storage and retrieval. The integration allows for easy configuration and ensures that all data interactions are handled efficiently within the application.

HTTP Requests

Django's robust architecture manages the entire communication with the server, ensuring secure sending and receiving of data. All data structures within the Django-based application adhere to the principles of DTO (Data Transfer Object), guaranteeing seamless transmission of data between the application and the application service model. Django-based HTTP request handling includes functions for various methods, such as GET, POST, PUT, and DELETE:

- **GET request:** In a GET request, a URL is created by concatenating a string with "http://," the IP address, and the port number, along with another string containing the path. After setting the timeout interval and method to "GET," the request is sent. Upon receiving a response, the data is processed and made available to the Django-based application controllers.

- **POST and PUT request:** Similar to GET requests, these actions involve setting timeout intervals and methods. Before sending the request, the body is encoded in JSON. Two types of functions exist for POST and PUT requests, one for cases requiring a response and one for cases where no response is needed.

- **DELETE request:** In a DELETE request, the URL is created, the timeout interval is set, the method is set to "DELETE," and the request is sent. No response is needed, as no data retrieval is required in this case.

This Django-centric approach, enriched with the ORM system for data management, security features for password encryption, and the integration of Django's system with SQLite for the data tier, ensures the seamless integration of graphical elements, secure data management, and efficient handling of HTTP requests within the application. Django's features, including its powerful framework for web applications, contribute to the overall effectiveness, security, and maintainability of the application.

3. SOURCE CODE

All the source code implemented for the DREAM project can be found in the github repository at the link: https://github.com/Attilioap/PolitoRigionePisoneSoricelli

3.1 BACKEND STRUCTURE

In the Django application, the backend structure comprises several essential components that collaborate to handle data processing, business logic, and communication with the frontend.

- **Models:** Models define the database structure, specifying fields and behaviors for data entities. They are typically found in the models.py file within each Django app.
- **Views:** Views serve as the control center for business logic, managing the flow of data between models and templates. They process HTTP requests and return responses, with code residing in the views.py file.
- **Templates:** Templates dictate how data is presented to users, utilizing HTML files with embedded Django template language. These files are stored in a templates directory within the Django app.
- **URLs:** URLs map specific endpoints to corresponding views, defining the paths for various functionalities. URL configurations are found in the urls.py file within the Django app and often included in the main project urls.py.
- **Settings:** Settings encompass configuration parameters for the entire Django project, including database settings, installed apps, and project-specific configurations. These settings are commonly stored in the settings.py file at the project level.
- **Middleware:** Middleware components process requests and responses globally before reaching or leaving views. They handle tasks such as authentication and security, configured in the MIDDLEWARE setting in settings.py.
- **Forms:** Forms manage user input validation and processing, aiding in creating, updating, or validating data. They are defined in the forms.py file within the Django app.
- **Static Files:** Static files, like CSS and JavaScript, are stored in the static directory. Django's {% static %} template tag is employed to reference these files, often residing in a static directory within each Django app.
- **Admin Interface:** Django offers a built-in admin interface for efficient management of models and data. Administrators can perform CRUD operations on database records through this interface, accessible at /admin by default.

This overview emphasizes Django's convention over configuration philosophy, providing developers with a structured framework that encourages modular and reusable code. Each component plays a crucial role in creating a scalable, maintainable backend for web applications.

3.2 FRONTEND STRUCTURE

- **Templates:** Django employs a template system for the frontend, where HTML files contain embedded Django template language. Templates are responsible for rendering dynamic content and presenting data from the backend to the user.

- **Static Files:** Frontend assets, including CSS, JavaScript, images, and other static files, are stored in the static directory. The {% static %} template tag is used to reference these files, providing a centralized location for static resources.

- **Media Files:** User-uploaded files, such as images or documents, are stored in the media directory. Django's settings, specifically MEDIA_ROOT and MEDIA_URL, manage the storage and retrieval of these media files.

- **Django Forms:** Frontend interactions with user input are facilitated through Django forms. These forms generate HTML forms, validate user-submitted data, and play a crucial role in user interaction.

- **Django Views:** Django views on the frontend side handle HTTP requests and generate responses. They often render templates, process form submissions, and interact with models to retrieve data for display.

- **URLs and Routing:** URL patterns define how URLs map to views in Django. In the frontend, these patterns are configured in the urls.py file within each Django app, allowing for navigation through different sections of the application.

- **Django REST Framework:** For frontend applications communicating with the backend through a RESTful API, Django REST Framework can be integrated. It extends Django to handle API views, serializers, and authentication, facilitating smooth communication between the frontend and backend.

4. TESTING

4.1 APPLICATION SERVER TESTING

Testing in a Django application is a crucial aspect to ensure the functionality and reliability of the implemented features. In the case of the CKBApp, testing is organized into various units, focusing on different components of the backend functionality. The testing suite primarily leverages Django's built-in testing tools.

Authentication and User Management Testing: For the authentication and user management functionalities, tests cover the signup, login, and user role assignment processes. These tests verify the proper creation of user accounts, correct assignment to the 'Educators' or 'Students' group based on the user's choice during signup, and successful login authentication.

Tournament and Battle Management Testing: Tests related to tournament and battle management assess the creation, modification, and closure of tournaments and battles. The testing includes verifying that tournaments can be created with valid parameters, battles can be created within tournaments, and the closing of tournaments has the expected impact on associated battles.

Webhooks and Code Evaluation Testing: Webhooks play a crucial role in handling external events, such as GitHub push notifications triggering code evaluation. Tests in this category simulate GitHub webhook payloads and evaluate whether the code evaluation process functions correctly. Additionally, the testing ensures that the system handles JSON payloads appropriately and rejects invalid ones.

Invitation System Testing: The invitation system is thoroughly tested to validate the creation and response to invitations. This includes sending invitations to potential teammates, handling invitation responses (acceptance or decline), and ensuring that responses are appropriately processed, affecting team membership.

Scoring and Leaderboard Testing: Scoring mechanisms, both for battles and tournaments, are tested to guarantee accurate calculation and storage of scores. The testing covers scenarios such as evaluating team and student scores based on code quality, passed test cases (randomized), and timeliness. Leaderboards are then checked to confirm they reflect the correct rankings.

File Handling and Security Measures Testing: Tests related to file handling, including saving and retrieving code files, ensure that the application handles user-submitted code securely. This involves validating that files are stored appropriately, that file paths are constructed correctly.

Error Handling and Exception Testing: The application is tested for robust error handling and exception management. This involves simulating scenarios that could lead to errors such as attempts to create battles with conflicting parameters. The tests verify that the application gracefully handles such situations, providing informative error messages where necessary.

Overall Integration Testing: Integration tests assess the seamless interaction between different components of the system. This includes testing the end-to-end flow of creating a tournament, adding battles, inviting students, and evaluating code submissions. These tests ensure that the entire application functions cohesively as a unified system.

By adopting a comprehensive testing approach, the CKBApp can maintain reliability, identify potential issues early in the development process, and support ongoing enhancements with confidence in the existing functionality.

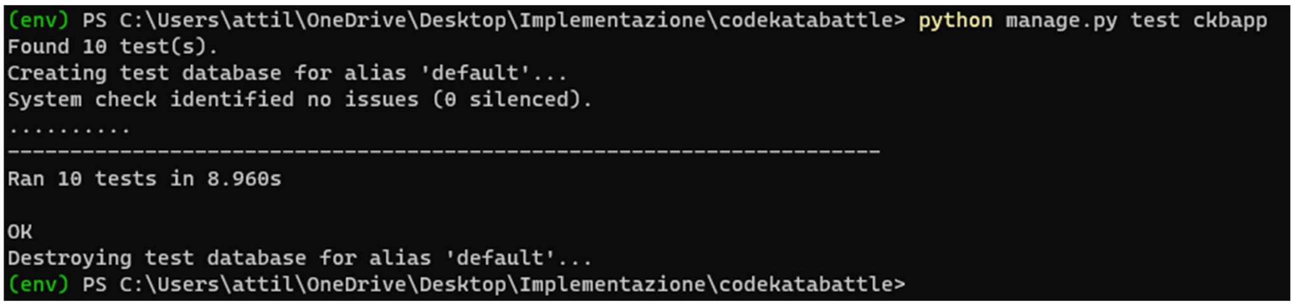
The tests were made thanks to Django unittest module. To perform the test correctly we configured the testing database in witch dummy instances are created during tests. After this we create the test into the "tests.py" file (check into the delivery folder for a better visualization):

```python
from django.urls import reverse
from django.test import TestCase
from django.contrib.auth.models import User, Group
from .models import Educator, Tournament, Student, Battle, Team, Invite, Repository


class CKBAppViewsTest(TestCase):
    def setUp(self):
        # Create Educator user
        self.educator_user = User.objects.create_user(username='educator',
password='test_password')
        self.educator_group = Group.objects.create(name='Educators')
        self.educator_user.groups.add(self.educator_group)
        self.educator = Educator.objects.create(user=self.educator_user)

        # Create Student user
        self.student_user = User.objects.create_user(username='student',
password='test_password')
        self.student_group = Group.objects.create(name='Students')
        self.student_user.groups.add(self.student_group)
        self.student = Student.objects.create(user=self.student_user)

        # Create a test tournament
        self.tournament = Tournament.objects.create(
            name='Test Tournament',
            registrationDeadline='2024-02-04',
            endingDate='2024-02-14',
            description='Test Description',
            creator=self.educator
        )
        # Create a test battle
        self.battle = Battle.objects.create(
            name='Test Battle',
            maxStudentsForTeam=3,
            registrationDeadline='2024-02-06',
            submissionDeadline='2024-02-10',
            creator=self.educator,
            tournament=self.tournament
        )

        # Create a test team
        self.team = Team.objects.create(
            name='Test Team',
            numTeammates=1,
            battle=self.battle
        )
        self.team.members.add(self.student)

    def test_signup_view(self):
        response = self.client.post(reverse('signup'), {
            'firstName': 'John',
            'lastName': 'Doe',
            'email': 'john.doe@example.com',
            'username': 'johndoe',
            'password1': 'test_password',
            'is_educator': False
        })
        self.assertEqual(response.status_code, 302)  # Check for a successful
redirect

    def test_user_login_view(self):
        response = self.client.post(reverse('login'), {'username': 'student',
'password': 'test_password'})
        self.assertEqual(response.status_code, 302)  # Check for a successful
redirect

    def test_educator_dashboard_view(self):
        self.client.login(username='educator', password='test_password')
        response = self.client.get(reverse('educator_dash'))
        self.assertEqual(response.status_code, 200)  # Check for a successful
response

    def test_tournament_info_view(self):
        tournament = Tournament.objects.create(name='Test Tournament',
registrationDeadline='2024-02-04', endingDate='2024-02-14', description='Test
Description', creator=self.educator)

        # Ensure the user is logged in (either educator or student)
        self.client.login(username='educator', password='test_password')
        response = self.client.get(reverse('tournament_info',
args=[tournament.id]))

        # Check for a successful response or redirect
        self.assertIn(response.status_code, [200, 302])

        if response.status_code == 200:
            # If status_code is 200, check for the content or any other specific
details
            self.assertContains(response, tournament.name)

    def test_student_dashboard_view(self):
        self.client.login(username='student', password='test_password')
        response = self.client.get(reverse('student_dash'))
        self.assertEqual(response.status_code, 200)  # Check for a successful
response

    def test_tournament_managment_view(self):
        # Test if the view is accessible by an educator
        self.client.force_login(self.educator_user)
        response = self.client.get(reverse('tournament_managment',
args=[self.tournament.id]))
        self.assertEqual(response.status_code, 200)

        # Test form submission for creating a battle
        response = self.client.post(reverse('tournament_managment',
args=[self.tournament.id]), {
            'battle_name': 'New Battle',
            'max_students_for_team': 2,
            'registration_deadline': '2024-02-07',
            'submission_deadline': '2024-02-11',
            'code_kata': '',
            'security': 'on',
            'reliability': 'on',
            'maintainability': 'on'
        })
        self.assertEqual(response.status_code, 302)  # Check for a successful
redirect after form submission

    def test_tournament_status_page_educator_view(self):
        # Test if the view is accessible by an educator
        self.client.force_login(self.educator_user)
        response = self.client.get(reverse('tournament_status_page_educator',
args=[self.tournament.id]))
        self.assertEqual(response.status_code, 200)

        # Test closing the tournament
        response = self.client.post(reverse('tournament_status_page_educator',
args=[self.tournament.id]), {'close_tournament': 'close'})
        self.assertEqual(response.status_code, 302)  # Check for a successful
redirect after closing the tournament

    def test_battle_status_page_view(self):
        # Test if the view is accessible by an educator
        self.client.force_login(self.educator_user)
        response = self.client.get(reverse('battle_status_page',
args=[self.battle.id]))
        self.assertEqual(response.status_code, 200)

    def test_tournament_status_page_student_view(self):
        # Test if the view is accessible by a student
        self.client.force_login(self.student_user)
        response = self.client.get(reverse('tournament_status_page_student',
args=[self.tournament.id]))
        self.assertEqual(response.status_code, 200)

    def test_battle_status_student_view(self):
        # Test if the view is accessible by a student
        self.client.force_login(self.student_user)
        response = self.client.get(reverse('battle_status_student',
args=[self.battle.id]))
        self.assertEqual(response.status_code, 200)

        # Test sending an invitation
        response = self.client.post(reverse('battle_status_student',
args=[self.battle.id]), {
            'action_type': 'invite_teammate',
            'teammate_username': 'new_teammate'
        })
        self.assertEqual(response.status_code, 200)  # Check for a successful
response after sending an invitation

    def tearDown(self):
        # Clean up objects created during setup
        User.objects.all().delete()
        Educator.objects.all().delete()
        Student.objects.all().delete()
        Tournament.objects.all().delete()
        Battle.objects.all().delete()
        Team.objects.all().delete()
        Invite.objects.all().delete()
        Repository.objects.all().delete()
```

We made the migration of the test database with:

```
python manage.py migrate --database=test
```

We run the “python manage.py test ckbapp” command in order to obtain the results of the test. Here it is the output retrieved by the test execution:



```
(env) PS C:\Users\attil\OneDrive\Desktop\Implementazione\codekatabattle> python manage.py test ckbapp
Found 10 test(s).
Creating test database for alias 'default'...
System check identified no issues (0 silenced).
..........
----------------------------------------------------------------------
Ran 10 tests in 8.960s

OK
Destroying test database for alias 'default'...
(env) PS C:\Users\attil\OneDrive\Desktop\Implementazione\codekatabattle>
```

Figura 1: Tests Results

4.2 HTTP REQUESTS TESTING

The process of testing this Django application involves a meticulous examination of its functionalities using the local server environment. The dynamic nature of the web application is explored by interacting with various buttons, forms, and other user interface elements. By leveraging the local server, we systematically navigated through the different views and functionalities provided by the application.

The urls.py file is a critical component in a Django application, responsible for mapping various URL patterns to their corresponding views. This file establishes the structure of the web application's API endpoints, dictating how different HTTP requests are handled and directing users to the appropriate views. Let's delve into some key URL patterns defined in this Django project.

- The /educator_dashboard/ URL, when accessed, directs users to the educator dashboard. This dashboard provides a comprehensive overview of ongoing tournaments, displaying relevant information for educators.

- User authentication is managed through the /signup/ and /login/ URLs. The former renders a user registration form, while the latter presents a login form. These URLs handle form submissions, ensuring secure user authentication and registration processes.

- Students interact with the application through the /student_dashboard/ URL. This endpoint offers insights into enrolled and upcoming tournaments, enhancing the student experience within the platform.

- Tournament management functionalities are facilitated by the /tournament_managment/<int:tournament_id>/ URL. This dynamic URL pattern enables educators to create and manage tournaments, including the initiation of new battles.

- Educators gain detailed perspectives on their tournaments through the /tournament_status_page_educator/<int:tournament_id>/ URL. This endpoint allows educators to perform actions such as closing tournaments, providing a centralized hub for overseeing tournament-related activities.

- Individual battles are explored through the /battle/<int:battle_id>/ URL, offering information on teams, submissions, and leaderboards. This URL is integral to the application's core functionality, providing insights into specific battle details.

- Students gain insights into tournaments and battles through the /tournament/<int:tournament_id>/status/ and /battle_status_student/<int:battle_id>/ URLs, respectively. These endpoints offer a student's perspective on tournament progress and battle specifics.

- User logout functionality is managed by the /logout/ URL, ensuring a smooth and secure logout process for application users.

- Comprehensive details about a specific tournament are presented through the /tournament/<int:tournament_id>/info/ URL, offering a centralized source of information for users.

- Lastly, the /github_webhook/ URL handles incoming GitHub webhook events. This integration fosters seamless communication between the Django application and version control on GitHub, supporting collaborative development practices.

Testing HTTP requests to these diverse endpoints is crucial for verifying the robustness and reliability of the application's API.

5. INSTALLATION INSTRUCTIONS

Before starting the configuration of the web application it is important to notice that you cannot install the web application on a MacOS or Linux device. It would be possible with the “requirements” file but, since the fact that we do not have a device to test this feature, we’ll release only the indications to run the web application on Windows.

Step 1: Set up Python

Install Python:

Download Python (version 3.9 or newer) from the official website.

During installation, make sure to check the option that says "Add Python to PATH."

Open a Command Prompt to verify the Python installation:

```
python --version
```

Step 2: Navigate to the project directory

Open a Command Prompt and navigate to the directory where your Django project is located:

```
cd path\to\project\Implementazione
```

Step 3: Activate the virtual environment and install the libraries to run the web application (ensure to have “pip” installed):

```
python -m venv env

env\Scripts\activate

pip install django

pip install django-crispy-forms

pip install github

pip install PyGithub

pip install flake8
```

Step 4: Apply Migrations

Run database migrations:

```
python manage.py migrate
```

Step 5: Run the Development Server

Start the Django development server:

python manage.py runserver

Open your web browser and go to http://127.0.0.1:8000/ to see the Django app.

Step 6: Create an Admin Superuser (Optional)

To access the Django admin interface, create a superuser:

python manage.py createsuperuser

Follow the prompts to set up the superuser account.

Step 7: Run the App

python manage.py runserver

Your Django app should now be running locally on your Windows machine. Access the admin interface at http://127.0.0.1:8000/admin/ if you created a superuser, or access to login interface at http://127.0.0.1:8000/ckbapp/login.

Remember to keep the Command Prompt open with the virtual environment activated while running the development server. To deactivate the virtual environment, use the command:

deactivate

Now you have all the necessaries requirements to run the web application locally and execute all the functions such as signup/login, create/enroll in tournaments/battles, see tournament and battle's information such us leaderboards, created repositories…

To access to functionalities that trigger through GitHub actions, such as automatic evaluation after a student's push on the main branch of the created GitHub repository, you'll need some further step. To configure GitHub webhooks, in order to let GitHub notify the platform with a POST after every Push, you'll need a public endpoint URL, since GitHub cannot send requests to local endpoints.

First of all you need to install ngrok

Go to the Ngrok website https://ngrok.com/

Sign up for a free Ngrok account by providing the required information.

Find the appropriate download link for windows

Log in to your Ngrok account on the Ngrok dashboard.

Copy the authentication token provided on the page.

Open a terminal or command prompt.

Run the following command in the terminal or command prompt, replacing <auth_token> with the token you copied:

```
ngrok authtoken <auth_token>
```

Now you can use Ngrok to expose your local server to the internet:

Start your Django development server

```
python manage.py runserver
```

Run the following command:

```
ngrok http http://localhost:8000
```

The Prompt will display ngrok parameters, in the section "Forwarding", you'll have your public endpoint, here's an example: https://c36e-5-90-122-8.ngrok-free.app (before "-> http://localhost:8000")

To make the whole thing work correctly you'll just need some few steps such us, in "settings.py" you need to modify:

```
DJANGO_APP_WEBHOOK_URL= 'https://be51-158-47-247-42.ngrok-free.app/ckbapp/github_webhook/'
```

And

```
CSRF_TRUSTED_ORIGINS= ['https://be51-158-47-247-42.ngrok-free.app']
```

With your public endpoint. Remember to do this every time you restart ngrok.

Now you'll able to access to the app from any PC-device you want through https://c36e-5-90-122-8.ngrok-free.app/ckbapp/login (replace https://c36e-5-90-122-8.ngrok-free.app with your actual ngrok endpoint).

Since the fact that the webhook URL is automatically set for repositories at creation, the entire application will work only for the repositories created after you made the before steps, unless you manually change the webhook URL for repositories created before. To make this you can access to GitHub account of the application with username:"codeKata3", password: "Politorigione3" (contact attilio.polito@mail.polimi.it for any problem you have accessing GitHub), and go to the repository's settings; here, in the webhooks section, edit the payload URL setting it as the one you provided for DJANGO_APP_WEBHOOK_URL.

It is important to state that repositories are created when the battle registration deadline is passed, after a student log into the platform. State also that all the pushes make after the battle submission deadline will not trigger any automatic update of the leaderboards (since the fact that battle is ended).

Now a student will just need to setup the GitHub workflow after the fork of the repository. In the main branch of the Repository, he must create and commit ".github/workflows/workflow1.yml"(you can find the workflow in the project directory as "yaml.yml"):

```yaml
name: Invia a piattaforma

on:
  push:
    paths:
      - 'code_katas/code_kata.py'

jobs:
  send-to-platform:
    runs-on: ubuntu-latest

    steps:
      - name: Checkout repository
        uses: actions/checkout@v2

      - name: Visualizza il percorso del file
        run: |
          ls -la
          echo "Contenuto della cartella:"
          ls -laR

      - name: Visualizza il contenuto del file
        run: |
          cat code_katas/code_kata.py

      - name: Ottieni l'email del committer
        id: get_commit_email
        run: echo "::set-output name=email::$(git log -1 --pretty=format:'%ae')"

      - name: Invia richiesta POST alla piattaforma
        run: |
                     python3   -c   'import   json,sys;   content=sys.stdin.read();
print(json.dumps({"file_content":      content,      "repository_name":      "'${{
github.event.repository.name         }}'",         "user_email":          "'${{
steps.get_commit_email.outputs.email       }}'",       "user_username":       "'${{
github.repository_owner  }}'"  }  ))'  <  code_katas/code_kata.py  |  curl  -X  POST  -H
"Content-Type:    application/json"    -d    @-    https://be51-158-47-247-42.ngrok-
free.app/ckbapp/github_webhook/
```

At the end of the workflow the student must change the endpoint with the one provided for DJANGO_APP_WEBHOOK_URL.

Remember that, as stated in DA (domain assumptions) the student GitHub username should be the same of the one used for the platform.

Now the student can make commits for the code. A commit will trigger the workflow, so the POST request will be sent to the endpoint URL, that will trigger the “github_webhook” function in the application. The function will handle the submitted code, making the static analysis of the code, evaluating the student work automatically due to quality of the code, timeliness of the commit and number of passed test cases. It is important to notice that the static analysis tool <u>will accept only python codes</u>, and the last one of the three parameters (number of passed test cases), has been simulated (random number between 1 and 10, multiplied by ten).

With the steps described before you’ll able to use the platform and all its functionalities.

6. EFFORT SPENT

Topic	**Hours**
DB configuration and development	8:00h
Installation	3:30h
Templates and CSS	15:00h
Views	15:00h
Django urls configurations	8:30h
GitHub configuration/reaction to pushes handling	10:00h
Overall Tests	5:00h
I&T Document development	10:00h
Document organization	5:00h
Overall system report	5:00h
Total effort spent	85:00h

Tabella 3:Effort spent by student 1

Topic	**Hours**
DB configuration and development	8:00h
Installation	3:30h
Templates and CSS	15:00h
Views	15:00h
Django urls configurations	8:30h
GitHub configuration/reaction to pushes handling	10:00h
Overall Tests	5:00h
I&T Document development	10:00h
Document organization	5:00h
Overall system report	5:00h
Total effort spent	85:00h

Tabella 4:Effort spent by student 2

Topic	**Hours**
DB configuration and development	8:00h
Installation	3:30h
Templates and CSS	15:00h
Views	15:00h
Django urls configurations	8:30h
GitHub configuration/reaction to pushes handling	10:00h
Overall Tests	5:00h
I&T Document development	10:00h
Document organization	5:00h
Overall system report	5:00h
Total effort spent	85:00h

Tabella 5:Effort spent by student 3

7. REFERENCES

• All the Python Scripts and user interfaces have been made with the VisualStudioCode using Python, HTML and CSS.