TASK MANAGEMENT WEB APPLICATION Flowingo

Arda KAYA
Software Engineering
Kocaeli University
Kocaeli, Turkiye
230229018@kocaeli.edu.tr

Osman Anıl YALÇIN
Software Engineering
Kocaeli University
Kocaeli, Turkiye
230229055@kocaeli.edu.tr

***Abstract*— This document serves as the final technical report for the "Flowingo" Task Management Web Application, completing the software development lifecycle initiated in the mid-term phase. While the initial phase established the core architecture using Python Django and MySQL, this final phase focuses on the implementation of advanced enterprise requirements. Key developments include a Task-Based File Attachment Module utilizing Django signals for storage optimization, a strict Role-Based Access Control (RBAC) system, and a Data Visualization Dashboard powered by Chart.js. Furthermore, the system's reliability and security have been validated through a comprehensive suite of automated unit tests covering API integrity and data privacy. This report details the architectural decisions, code implementations, and testing outcomes that demonstrate the project's compliance with all specified requirements.**

Keywords— Task Management, Web Application, Django, Python, MySQL, Bootstrap, Django REST Framework, Role-Based Access Control, Data Visualization, Unit Testing, Software Engineering.

I. INTRODUCTION

This project aims to develop a web-based "Task Management Web Application" designed to help users manage their daily or professional tasks. The core objective of the system is to create a platform that allows users to add, edit, categorize, and monitor the completion status of their tasks.

The project is also intended to practically enhance skills in both frontend and backend development, as well as database management proficiency.

Efficient task management is a cornerstone of professional productivity. The "Flowingo" project aims to digitize this process through a secure, scalable web application. Following the successful deployment of authentication and basic CRUD (Create, Read, Update, Delete) functionalities reported in the mid-term review, the project has evolved to meet complex user needs.

This final report documents the integration of the "Extended Requirements" defined in the project curriculum. Specifically, it addresses the technical implementation of file management systems, hierarchical user authorization, real-time visual analytics, and automated system verification. The following sections detail the technologies employed (Section II), the backend and frontend implementations (Section III & IV), and the verification results (Section V).

II. TECHNOLOGIES USED AND REASONS FOR SELECTION

The technologies selected to meet the project requirements and the reasons for their selection are detailed below.

A. Backend (Server Side)

The Python programming language and the Django web framework were chosen for the system's backend infrastructure. Django's "batteries-included" philosophy allowed for the rapid and secure development of the API endpoints required by the project, such as /api/auth/register, /api/auth/login, and /api/tasks.

B. Frontend (User Interface)

The user interface is built upon the foundations of HTML5, CSS3, and JavaScript. The Bootstrap CSS library was utilized to ensure the development of a modern, responsive, and quickly-implemented interface. This choice facilitated the efficient creation of the "Login/Registration Page" and "Task List (Dashboard)" drafts specified in the project's page flow requirements.

C. Database

MySQL, a relational database management system, was selected for storing user information (users) and task records (tasks). MySQL was identified as the optimal solution for the project's database needs due to its widespread use, reliability, and high compatibility with the Django framework.

D. Security and Encryption

Django's built-in security tools were used to fulfill the "Security and Encryption" requirement and to ensure passwords are stored encrypted. Django's native authentication system automatically handles and stores user passwords in the database using secure hashing methods as required. This allowed for the successful implementation of "Invalid login" tests and secure authentication.

E. Testing and Quality Assurance

To ensure the reliability and stability of the backend infrastructure, a comprehensive automated testing strategy was implemented using the django.test framework and the Django REST Framework's APIClient. This architecture allows for the simulation of HTTP requests (GET, POST, DELETE) within an isolated environment, verifying the integrity of API endpoints without affecting the production database. The testing suite, encapsulated within the TaskAPITest class, specifically targets critical security mechanisms, such as Role-Based Access Control (RBAC) verification in test_privacy_check and the handling of multipart binary data in test_file_upload. This automated approach guarantees that business logic—such as task creation validations and data privacy rules—remains robust against regressions during the development lifecycle.

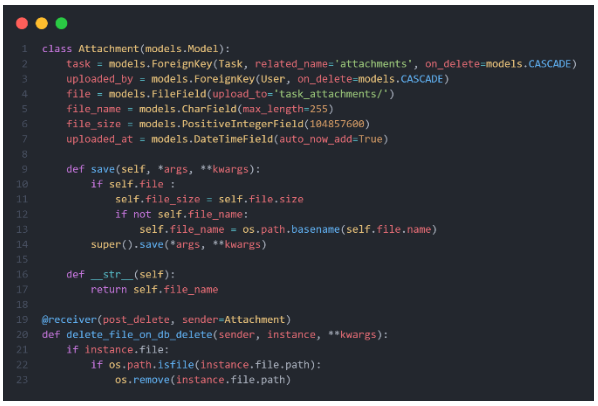
III. EXTENDED BACKEND INTEGRATION

The backend logic was significantly expanded to support file operations and security protocols.

A. *Task-Based File Management (Attachment Module)*

To allow users to attach documents to tasks, a new Attachment model was developed. This model utilizes a One-to-Many relationship with the Task model.

Storage Optimization via Signals: A critical engineering challenge was ensuring that deleted database records did not leave orphaned files in the server's storage. To solve this, a post_delete signal was implemented. When an Attachment object is deleted from the database, this signal automatically triggers the removal of the physical file from the file system.



```python
class Attachment(models.Model):
    task = models.ForeignKey(Task, related_name='attachments', on_delete=models.CASCADE)
    uploaded_by = models.ForeignKey(User, on_delete=models.CASCADE)
    file = models.FileField(upload_to='task_attachments/')
    file_name = models.CharField(max_length=255)
    file_size = models.PositiveIntegerField(104857600)
    uploaded_at = models.DateTimeField(auto_now_add=True)

    def save(self, *args, **kwargs):
        if self.file :
            self.file_size = self.file.size
            if not self.file_name:
                self.file_name = os.path.basename(self.file.name)
        super().save(*args, **kwargs)

    def __str__(self):
        return self.file_name

@receiver(post_delete, sender=Attachment)
def delete_file_on_db_delete(sender, instance, **kwargs):
    if instance.file:
        if os.path.isfile(instance.file.path):
            os.remove(instance.file.path)
```

Figure 1. The Attachment model implementation and the post_delete signal configuration ensuring automated file cleanup and storage optimization.

B. *Role Based Authorization System*

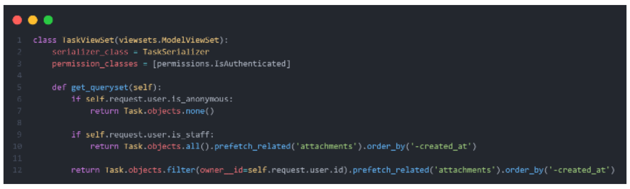
The application security was significantly improved by implementing a strict Role-Based Access Control (RBAC) system. Instead of relying on frontend restrictions, the authorization logic is enforced directly in the backend within the TaskViewSet class in views.py.

The system overrides the get_queryset method to filter data dynamically based on the user's role:

- *Admins*: Users with is_staff=True have unrestricted access to all tasks (Task.objects.all()), allowing for system-wide management.

- *Standard Users*: Regular users are restricted to accessing only the tasks they created. The system applies a mandatory filter: Task.objects.filter(owner=self.request.user).

By enforcing these rules at the database query level, the system ensures data isolation and effectively prevents Horizontal Privilege Escalation attacks, meaning a user cannot access another user's data even by manipulating ID parameters.



```python
class TaskViewSet(viewsets.ModelViewSet):
    serializer_class = TaskSerializer
    permission_classes = [permissions.IsAuthenticated]

    def get_queryset(self):
        if self.request.user.is_anonymous:
            return Task.objects.none()

        if self.request.user.is_staff:
            return Task.objects.all().prefetch_related('attachments').order_by('-created_at')

        return Task.objects.filter(owner__id=self.request.user.id).prefetch_related('attachments').order_by('-created_at')
```

Figure 2. Code snippet from views.py showing the get_queryset method override to enforce role-based data isolation.

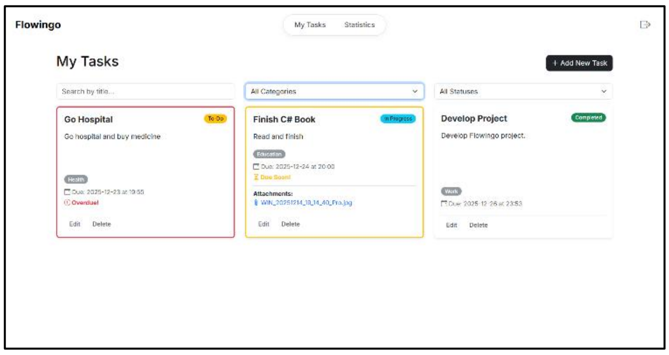
IV. FRONTEND ENHANCEMENTS AND VISUALIZATION

The user interface was upgraded to provide real-time feedback and actionable insights.

A. *Dynamic Filtering and Alert System*

The Dashboard (dashboard.js) now features a client-side filtering mechanism that allows users to sort tasks by "Category" and "Status" instantaneously without server round-trips. Furthermore, a visual alert system based on the dueDate was implemented:

- Overdue Tasks: Highlighted with a Red border.
- Urgent Tasks (<24h): Highlighted with a Yellow border.
- Completed Tasks: Marked with a Green badge.



Flowingo

My Tasks | Statistics

My Tasks

+ Add New Task

Search by title... | All Categories | All Statuses

Go Hospital — To Do
Go hospital and buy medicine
Health
Due: 2025-12-23 at 19:55
Overdue!
Edit Delete

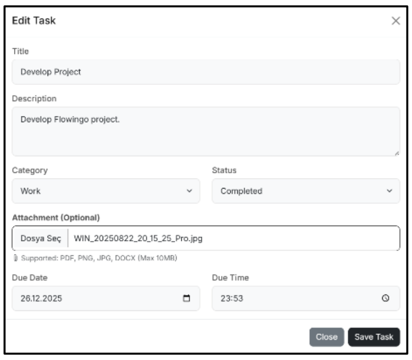
Finish C# Book — In Progress
Read and finish
Education
Due: 2025-12-24 at 20:00
Due Soon!
Attachments:
WIN_20251214_19_14_40_Pro.jpg
Edit Delete

Develop Project — Completed
Develop Flowingo project.
Work
Due: 2025-12-26 at 23:53
Edit Delete

Figure 3. The Dashboard interface displaying color-coded alerts for overdue (Red) and approaching (Yellow) deadlines, alongside the active filtering and search mechanisms.

B. *File Attachment Interface*

The task creation modal was updated to support multipart/form-data submissions. Users can upload files (PDF, DOCX, IMG) up to 10MB. Attached files are rendered as downloadable links within the task card.



Edit Task

Title: Develop Project

Description: Develop Flowingo project.

Category: Work

Status: Completed

Attachment (Optional): Dosya Seç | WIN_20250822_20_15_25_Pro.jpg

Supported: PDF, PNG, JPG, DOCX (Max 10MB)

Due Date: 26.12.2025

Due Time: 23:53

Close | Save Task

Figure 4. A Task Card showing the "Attachments" section with a downloadable file link, demonstrating the successful integration of the file module.

C. *Statistical Data Visualization*

A dedicated "Statistics" page was developed to visualize user productivity. Using Chart.js, the system fetches real-time data from the /api/tasks/ endpoint and renders:

- Status Distribution: A Doughnut Chart showing the ratio of To-Do, In Progress, and Completed tasks.
- Category Analysis: A Bar Chart displaying the distribution of tasks across different categories (Work, Personal, etc.).

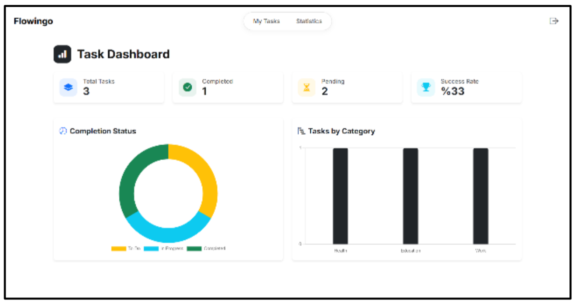


Figure 5. The Statistics Dashboard featuring interactive charts for task distribution and completion status.

V. TESTING AND VERIFICATION

To guarantee system stability, automated unit tests were written and executed using Django's testing tools.

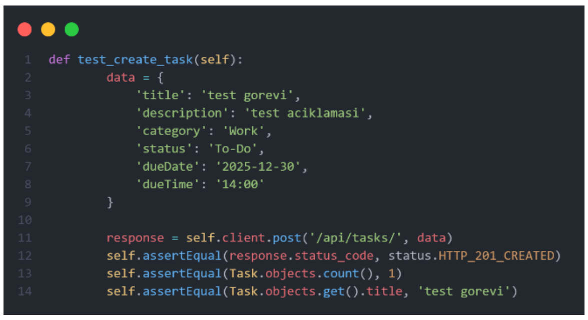
A. Automated Backend Test Scenarios

The tests.py suite includes three critical verification scenarios:

1. Task Creation: Validates that tasks are correctly saved with all mandatory fields.

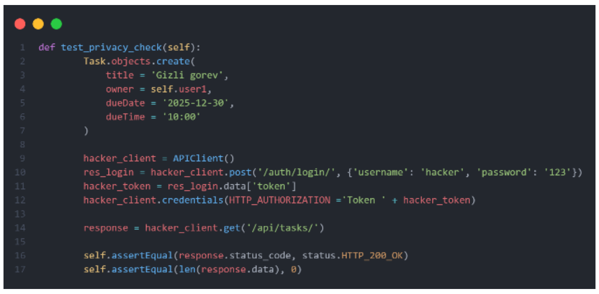
2. Privacy Check (test_privacy_check): Simulates a malicious user attempting to access another user's private data. The test asserts that the returned dataset is empty, confirming the RBAC logic.

3. File Upload (test_file_upload): Verifies that binary file data is correctly parsed and stored by the API.



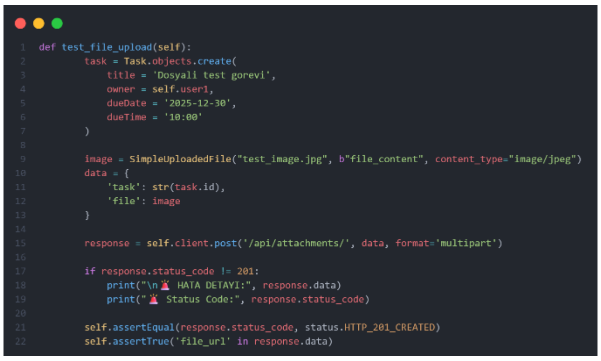
```python
def test_create_task(self):
        data = {
            'title': 'test gorevi',
            'description': 'test aciklamasi',
            'category': 'Work',
            'status': 'To-Do',
            'dueDate': '2025-12-30',
            'dueTime': '14:00'
        }

        response = self.client.post('/api/tasks/', data)
        self.assertEqual(response.status_code, status.HTTP_201_CREATED)
        self.assertEqual(Task.objects.count(), 1)
        self.assertEqual(Task.objects.get().title, 'test gorevi')
```

Figure 6. Code snippet from tests.py showing the test_create_task method.



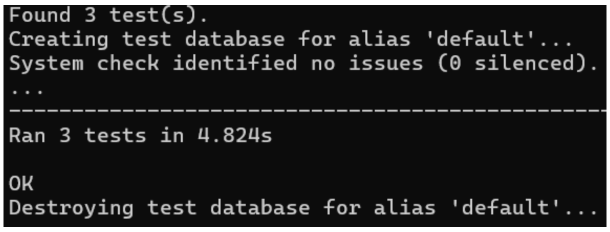
```python
def test_privacy_check(self):
        Task.objects.create(
            title = 'Gizli gorev',
            owner = self.user1,
            dueDate = '2025-12-30',
            dueTime = '10:00'
        )

        hacker_client = APIClient()
        res_login = hacker_client.post('/auth/login/', {'username': 'hacker', 'password': '123'})
        hacker_token = res_login.data['token']
        hacker_client.credentials(HTTP_AUTHORIZATION ='Token ' + hacker_token)

        response = hacker_client.get('/api/tasks/')

        self.assertEqual(response.status_code, status.HTTP_200_OK)
        self.assertEqual(len(response.data), 0)
```

Figure 7. Code snippet from tests.py showing the test_privacy_check method.



```python
def test_file_upload(self):
        task = Task.objects.create(
            title = 'Dosyali test gorevi',
            owner = self.user1,
            dueDate = '2025-12-30',
            dueTime = '10:00'
        )

        image = SimpleUploadedFile("test_image.jpg", b"file_content", content_type="image/jpeg")
        data = {
            'task': str(task.id),
            'file': image
        }

        response = self.client.post('/api/attachments/', data, format='multipart')

        if response.status_code != 201:
            print("\n🚨 HATA DETAYI:", response.data)
            print("🚨 Status Code:", response.status_code)

        self.assertEqual(response.status_code, status.HTTP_201_CREATED)
        self.assertTrue('file_url' in response.data)
```

Figure 8. Code snippet from tests.py showing the test_file_upload method.



```
Found 3 test(s).
Creating test database for alias 'default'...
System check identified no issues (0 silenced).
...
----------------------------------------------------------------------
Ran 3 tests in 4.824s

OK
Destroying test database for alias 'default'...
```

Figure 9. Terminal output showing the successful execution of automated tests (Ran 3 tests... OK).

VI. CONCLUSION

The "Flowingo" project has successfully transitioned from a prototype to a feature-rich web application. The final phase requirements—specifically the secure file handling system, hierarchical user permissions, and graphical data analysis—have been fully implemented and verified. The inclusion of automated testing ensures the long-term maintainability of the codebase. This project provided the team with extensive experience in full-stack development, database optimization, and software security practices.

REFERENCES

a. Django Software Foundation, "Django Documentation," 2025. [Online]. Available: https://www.djangoproject.com/
b. Python Software Foundation, "Python 3 Documentation," 2025. [Online]. Available: https://docs.python.org/3/
c. Oracle Corporation, "MySQL Documentation," 2025. [Online]. Available: https://dev.mysql.com/doc/
d. The Bootstrap Authors, "Bootstrap Documentation," 2025. [Online]. Available: https://getbootstrap.com/