BIL 481 – Design Document

TOBB ETÜ

University of Economics & Technology

Group Members

Helen Parlar – Artificial Intelligence Engineering

Mete Kılıç – Artificial Intelligence Engineering

Ahmet Çakmak – Medicine & Artificial Intelligence Engineering

TASK MATRIX

Section	Description	Contributor
1. System Overview	**Brief project description, system architecture, and technology stack.**	**All Members**
2. Implementation Details	**Codebase structure, key implementations, component interfaces, and visual interfaces.**	**Mete Kılıç**
3. Use Case Support in Design	**Use case selection, requirement mapping, use case design, and demo requirement.**	**Helen Parlar**
4. Design Decisions	**Technology comparisons, AI model choice, backend/frontend framework choice.**	**Ahmet Çakmak**
5. GitHub Commit Requirement	**Code implementations, interfaces, and technology comparisons.**	**All Members**

Table of Contents

1. System Overview

1.1. Brief Project Description

The "Smart Clinical Appointment System" is an AI-driven web platform designed to optimize the patient journey in hospitals. By utilizing an OpenAI-powered symptom analysis module, the system accurately directs patients to the correct polyclinic, reducing incorrect bookings. The system features dedicated interfaces for Patients (booking), Doctors (schedule management), and Administrators.

1.2. System Architecture

The system follows a 3-Tier Layered Architecture, separating the presentation, application logic, and data storage layers for modularity and scalability.

1.3. Technology Stack

- Frontend: Java Script
- Backend: Django (Python)
- Database: MySQL
- AI Model: OpenAI GPT Models (via API)
- Authentication: SimpleJWT

2. Implementation Details

2.1. Codebase Structure

The project repository is structured to separate application components into dedicated Django apps and isolate setup/seeding logic from the core backend.

High-Level Separation:

- Database/: Contains the main initial_schema.sql file necessary for the MySQL setup.
- clinic-randevu-system-frontend/: Contains all React/JavaScript source code for the Presentation Layer.
- clinical_backend/: Contains the main Django project settings and configuration.

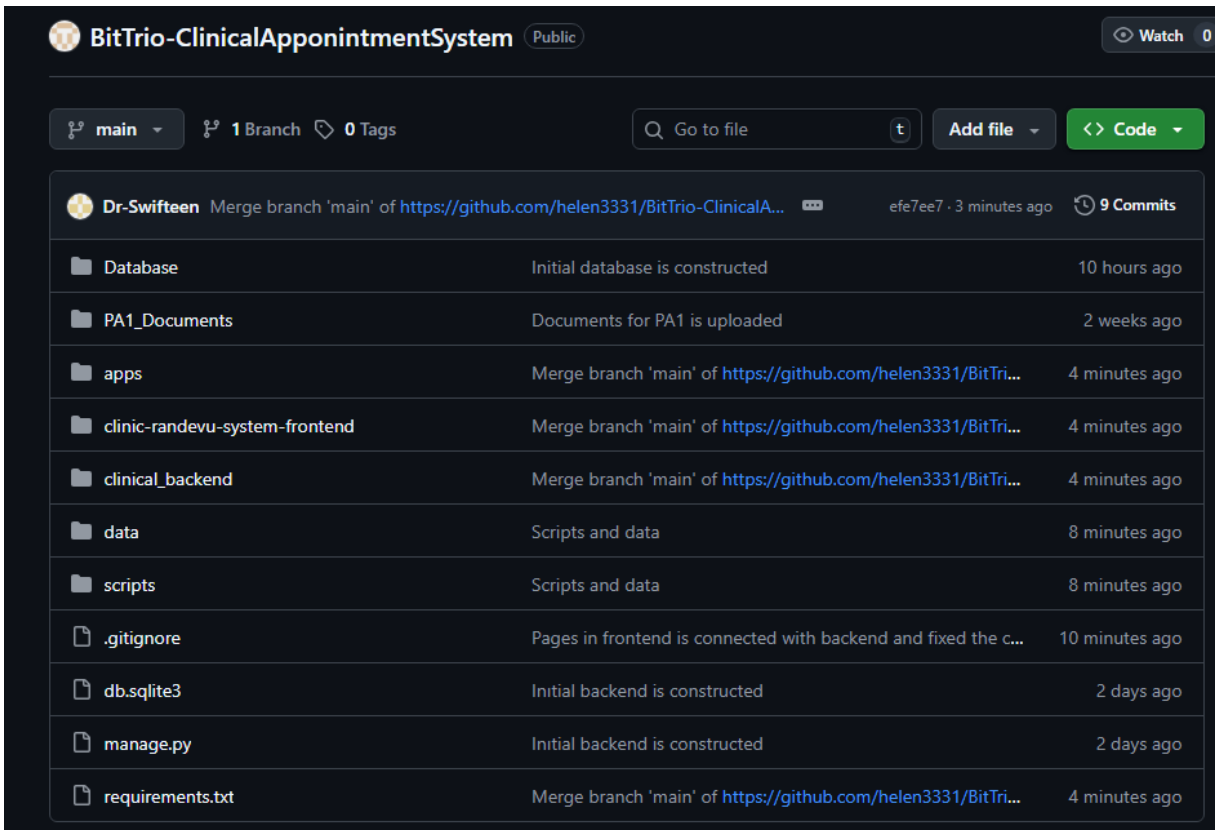
Application Logic (Django Apps):

- accounts/: Handles user authentication (login, registration) and manages the Patient/Doctor/Admin user roles.
- ai/: Contains the AIAnalyzeSymptomsView and logic for integrating with the OpenAI API.

- appointments/: Manages the core booking logic, transactional integrity, and the Appointment model.
- clinics/: Manages Polyclinic and Symptom data structures and related views/APIs.
- doctors/: Manages Doctor profiles and Schedule creation/availability logic.

Support & Data Seeding:

- data/: Contains project-wide data files, configuration references, and potentially external data used by scripts.
- scripts/: Contains utility scripts, primarily for data seeding (e.g., load_slots.py, load_symptoms.py) and setup tasks.



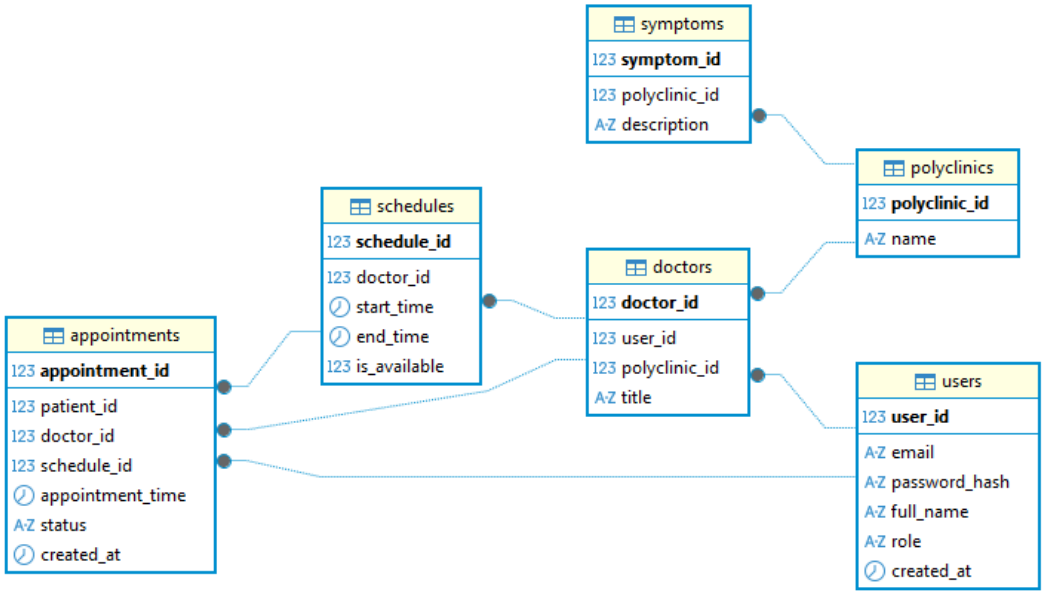
2.2. Key Implementations

- ### 2.2.1. Database Schema (Data Layer):

The database is the backbone of the application, designed with 6 core entities to ensure data integrity and support the booking flow.

  - **Users:** Stores role-based accounts with strict email validation constraints.
  - **Polyclinics & Symptoms:** Manages the medical departments and their associated symptoms.

- **Doctors, Schedules, Appointments:** Handles the transactional logic of booking.



SQL Implementation Snippet (User Constraints):

```
CONSTRAINT chk_email_role CHECK (
    (role = 'Doctor' AND email LIKE '%@doc.com') OR
    (role IN ('Patient', 'Administrator') AND email NOT LIKE '%@doc.com')
)
```

2.2.2 Custom User Model with Role-Based Access

```python
class User(AbstractUser):
    ROLE_CHOICES = (
        ("patient", "Patient"),
        ("doctor", "Doctor"),
        ("admin", "Administrator"),
    )

    email = models.CharField(max_length=255, unique=True)
    full_name = models.CharField(max_length=255, null=True, blank=True)
    role = models.CharField(max_length=20, choices=ROLE_CHOICES)
    created_at = models.DateTimeField(auto_now_add=True)

    USERNAME_FIELD = "email"          # Kullanıcı email ile giriş yapacak
    REQUIRED_FIELDS = ["username"]    # username yine zorunlu olsun (opsiyonel)

    def __str__(self):
        return self.full_name or self.email
```

Explanation:

- Extends Django's AbstractUser to add custom fields
- Email is the primary authentication field (instead of username)
- Role field enables role-based access control (RBAC)
- Supports three user types: Patient, Doctor, and Administrator

2.2.3. AI-Powered Symptom Analysis

```python
class AIAnalyzeSymptomsView(APIView):
    def post(self, request):
        symptoms = request.data.get("symptoms", "")

        if not symptoms:
            return Response({"error": "Symptoms field is required."}, status=400)

        prompt = f"""
        Aşağıdaki semptomları değerlendir ve en olası 1-3 poliklinigi öner.
        Cevabı sadece JSON formatında döndür:

        Format:
        {{
            "clinics": ["KBB", "Dahiliye"],
            "reason": "Kullanıcının X ve Y semptomları bu kliniklere uygun."
        }}

        Semptomlar: {symptoms}
        """

        try:
            response = client.chat.completions.create(
                model="gpt-4o-mini",
                messages=[
                    {"role": "system", "content": "You are a medical triage assistant."},
                    {"role": "user", "content": prompt}
                ]
            )

            ai_output = response.choices[0].message.content

            return Response({"result": ai_output})

        except Exception as e:
            return Response({"error": str(e)}, status=500)
```

Algorithm Explanation:

1. Input Validation: Check if symptoms text is provided
2. Prompt Engineering: Structured prompt in Turkish for clinic recommendation
3. JSON Response Format: Ensures parseable, structured output
4. Error Handling: Catches API failures and returns appropriate error messages

Core Business Logic:

- Uses GPT-4 mini model for cost-efficiency
- Expects 1-3 clinic recommendations with reasoning
- Turkish language support for user-facing application

2.2.4 Appointment Booking with Slot Locking

```python
class AppointmentCreateSerializer(serializers.ModelSerializer):
    class Meta:
        model = Appointment
        fields = ["doctor", "schedule"]

    def validate(self, data):
        if data["schedule"].is_booked:
            raise serializers.ValidationError("Bu slot dolu.")
        return data

    def create(self, validated_data):
        schedule = validated_data["schedule"]
        schedule.is_booked = True
        schedule.save()

        return Appointment.objects.create(
            patient=self.context["request"].user,
            doctor=validated_data["doctor"],
            schedule=schedule
        )
```

Race Condition Prevention:

- Validation: Checks if slot is available before booking
- Atomic Operation: Marks slot as booked immediately after creation
- OneToOneField: Database-level constraint prevents double booking

2.3. Component Interfaces

The system exposes RESTful API endpoints organized by domain:

Authentication APIs:

- POST /api/accounts/register/patient/ - Patient registration

```
import { API_URL } from "./config";
// Kullanıcının Tam İsim (isim) alanını backend'deki full_name alanına map'liyoruz.
export async function register_patient({ isim, username, email, password }) {

    const res = await fetch(`${API_URL}/accounts/register/patient/`, {
        method: "POST", // 🔥 KRİTİK DÜZELTME: Mutlaka POST olmalı
        headers: {
            "Content-Type": "application/json",
        },
        // Backend'in PatientRegisterSerializer'ının beklediği alan adlarını kullanıyoruz.
        body: JSON.stringify({
            full_name: isim, // 👈 Senin 'isim' state'in, backend'in 'full_name' alanına gidiyor.
            username: username,
            email: email,
            password: password,
        })
    });
    // 4xx veya 5xx hatalarını (örneğin email/username zaten kayıtlı) yakalama
    if (!res.ok) {
        const errorData = await res.json();

        // Backend'den gelen spesifik hata mesajlarını okumaya çalış
        let errorMessage = "Kayıt sırasında bir hata oluştu.";
        if (errorData.email) {
            errorMessage = `Email: ${errorData.email[0]}`;
        } else if (errorData.username) {
            errorMessage = `Kullanıcı Adı: ${errorData.username[0]}`;
        } else if (errorData.password) {
            errorMessage = `Şifre: ${errorData.password[0]}`;
        } else if (errorData.detail) {
            errorMessage = errorData.detail;
        }

        throw new Error(errorMessage);
    }
    return res.json();
}
```

Clinic & Symptom APIs:

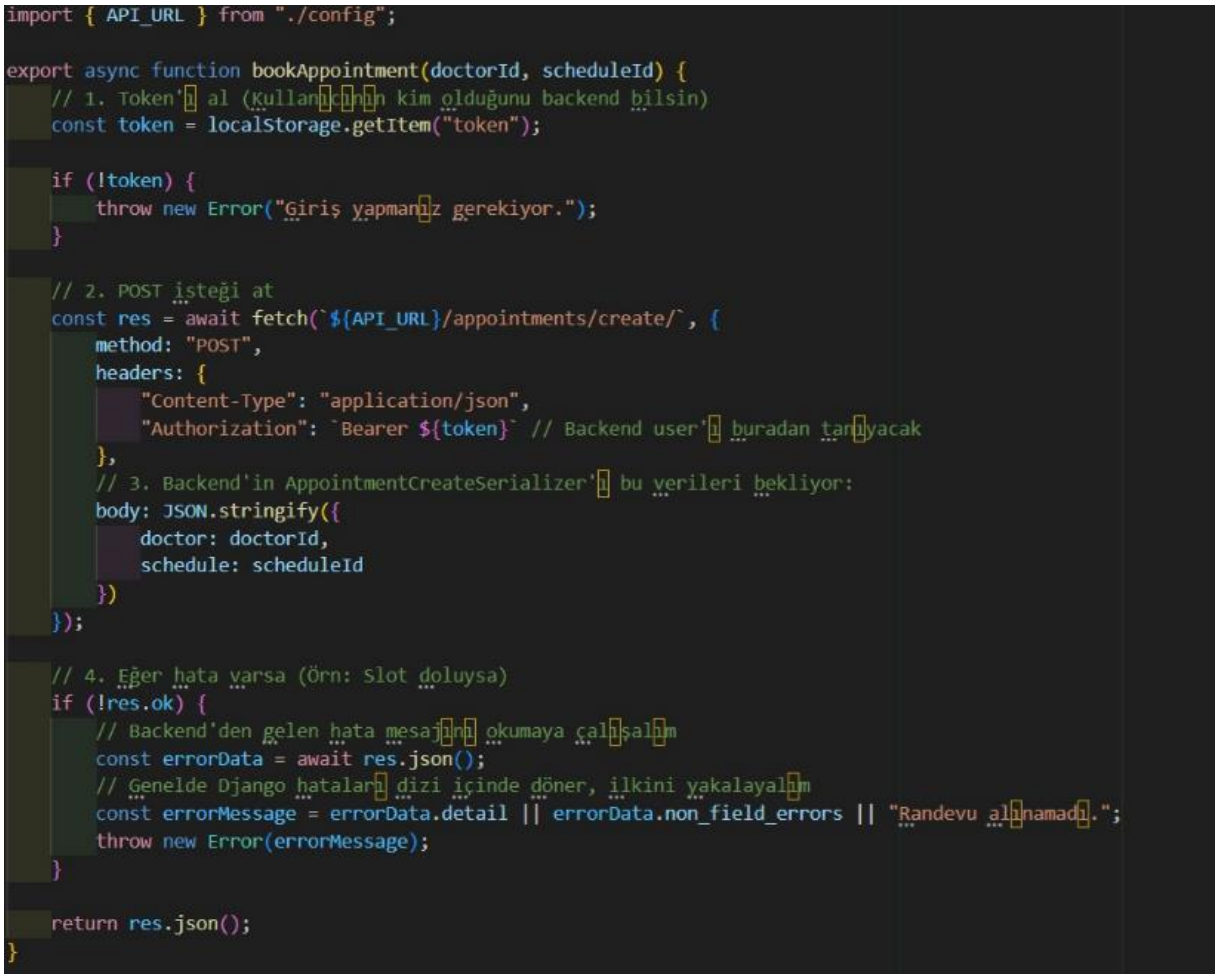
- GET /api/clinics/list/ - List all polyclinics

```
import { API_URL } from "./config";

export async function getClinics() {
    const res = await fetch(`${API_URL}/clinics/list/`);
    return res.json();
}
```

Appointment APIs:

- POST /api/appointments/create/ - Book appointment



```
import { API_URL } from "./config";

export async function bookAppointment(doctorId, scheduleId) {
    // 1. Token'ı al (Kullanıcının kim olduğunu backend bilsin)
    const token = localStorage.getItem("token");

    if (!token) {
        throw new Error("Giriş yapmanız gerekiyor.");
    }

    // 2. POST isteği at
    const res = await fetch(`${API_URL}/appointments/create/`, {
        method: "POST",
        headers: {
            "Content-Type": "application/json",
            "Authorization": `Bearer ${token}` // Backend user'ı buradan tanıyacak
        },
        // 3. Backend'in AppointmentCreateSerializer'ı bu verileri bekliyor:
        body: JSON.stringify({
            doctor: doctorId,
            schedule: scheduleId
        })
    });

    // 4. Eğer hata varsa (Örn: Slot doluysa)
    if (!res.ok) {
        // Backend'den gelen hata mesajını okumaya çalışalım
        const errorData = await res.json();
        // Genelde Django hataları dizi içinde döner, ilkini yakalayalım
        const errorMessage = errorData.detail || errorData.non_field_errors || "Randevu alınamadı.";
        throw new Error(errorMessage);
    }

    return res.json();
}
```

2.4. Visual Interfaces

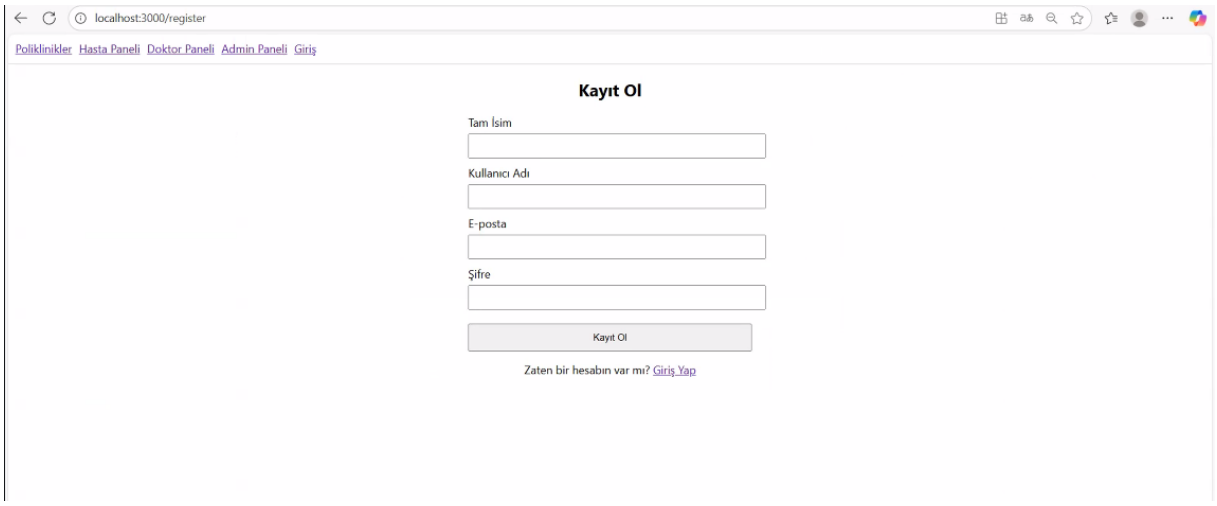
2.4.1. Patient Registration Page

- Form fields: İsim, Soyisim, E-posta, Şifre

- Validation: Real-time error messages

- Success message: "Kayıt başarılı! Giriş sayfasına yönlendiriliyorsunuz..."

- Navigation: Link to login page



2.4.2. Login Page

- Email and password fields

- Role-based routing after login:

  - Patient → /patient/dashboard

  - Doctor → /doctor/dashboard

  - Admin → /admin/dashboard

- "Kayıt Ol" button in header

2.4.3. Symptom Checker

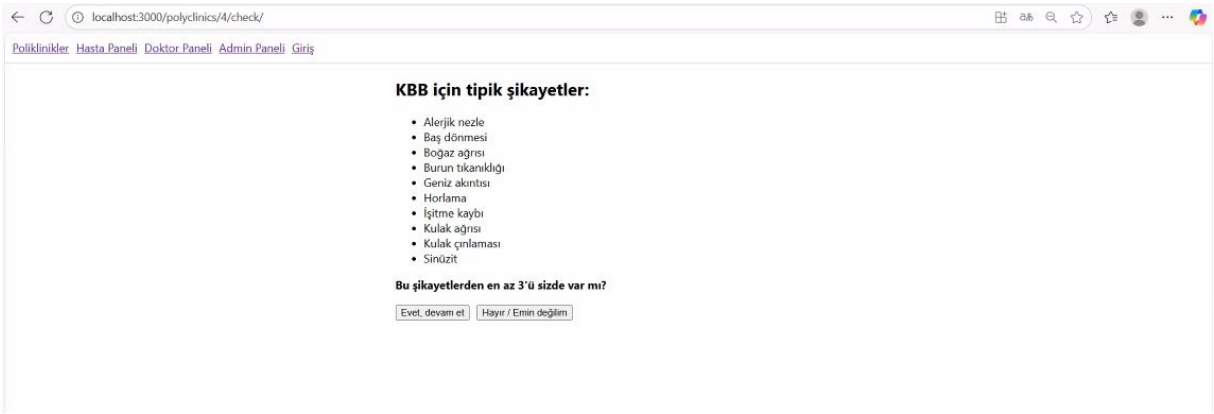
Two-stage verification:

1. Display common symptoms for selected clinic

2. Ask: "Bu şikayetlerden en az 3'ü sizde var mı?"

   - "Evet, devam et" → Doctor selection

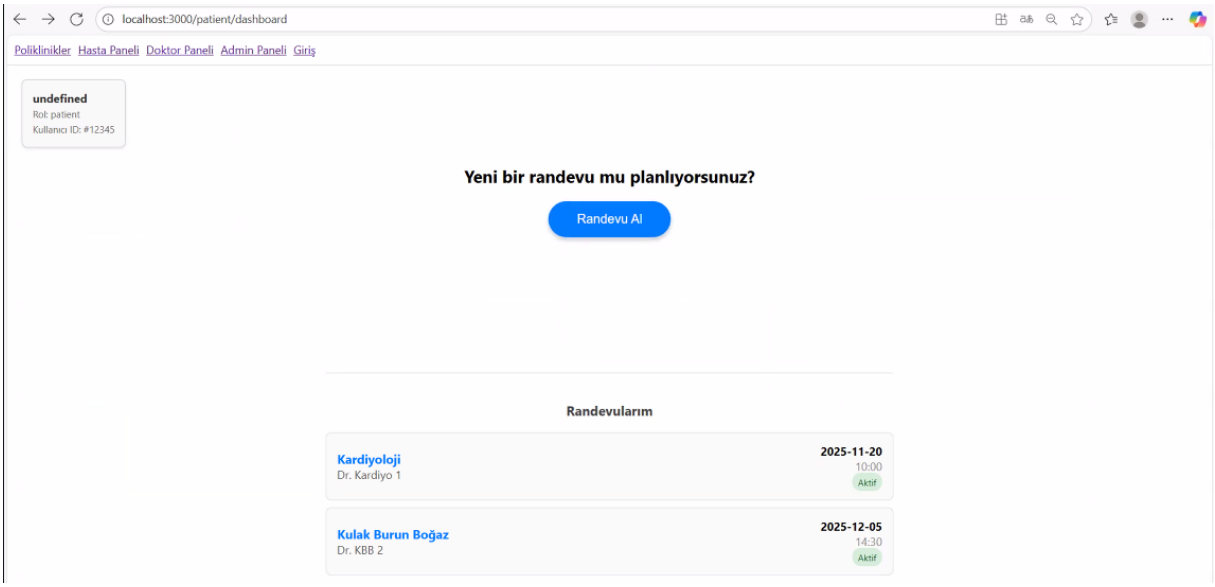
   - "Hayır / Emin değilim" → Free-text symptom entry



2.4.4. Patient Dashboard

This screen serves as the central hub for authenticated patients. It provides a quick overview of the user's status and current/upcoming appointments.

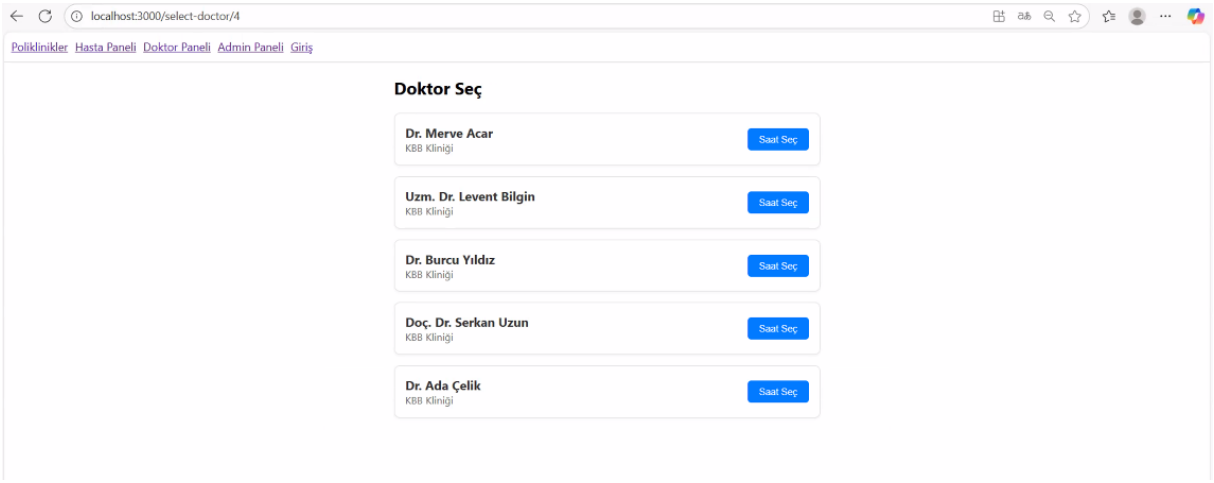
- **Functionality:** The dashboard facilitates the start of the booking process via the main **"Randevu Al"** (Book Appointment) call-to-action button, which navigates the user to the Polyclinic List.
- **User Status:** The screen displays the user's current role (e.g., 'patient') and ID.
- **Appointment List:** The **"Randevularım"** section lists all scheduled bookings, clearly showing the Polyclinic (e.g., Kardiyoloji, Kulak Burun Boğaz), Doctor, Date, and Time.



2.4.5. Doctor Selection and Slot Navigation

This screen is reached after the user has completed the symptom verification step. It lists available doctors within the specific polyclinic chosen by the patient.

- **Filtering:** The list is automatically filtered to show only doctors affiliated with the previously selected polyclinic (e.g., KBB kliniği).
- **Doctor Display:** Each entry shows the doctor's name and academic title (e.g., Dr., Uzm. Dr., Doç. Dr.).
- **Call to Action:** Clicking the **"Saat Seç"** button next to a doctor's name navigates the patient to the final available slot selection screen.



3. Use Case Support in Design

3.1. Use Case Selection

We have selected 4 critical use cases that define the system's core value:

1. Patient Registration & Login
2. Direct Polyclinic Selection (with Symptom Verification)
3. AI-Driven Symptom Analysis & Recommendation
4. Appointment Booking

3.2. Requirement Mapping

Use Case	Functional Requirement	Description
Registration & Login	FR1, FR2	Users create accounts and authenticate securely.
Direct Selection	FR4, FR5	Users select a clinic and confirm symptoms manually.
AI Analysis	FR6, FR7	Free-text symptoms are analyzed by AI to suggest clinics.
Booking	FR8	User selects a doctor and time slot; system creates appointment.

3.3. Use Case Design

Use Case	Data Flow	State Changes
1. Patient Registration & Login	1. [RegisterPage] Input → POST /api/accounts/register/ 2. Serializer Validates & Hashes Password 3. Users Table → Record Created 4. [LoginPage] Input → POST /api/accounts/login/ 5. Backend generates JWT Tokens 6. Frontend stores token in localStorage	• Initial: Unauthenticated (Guest) • Registered: User record created in DB • Authenticated: Session active, Token stored • Final: Redirected to Patient Dashboard
2. Direct Polyclinic Selection	1. User clicks Clinic → GET /api/clinics/{id}/symptoms/	• Selection: Clinic ID captured • Verification: User confirms symptom match

Use Case	Data Flow	State Changes
	2. [CheckSymptomsPage] displays list **3. User verifies symptoms (Yes/No)** **4. If Yes: Redirect to [DoctorListPage]** **5. If No: Redirect to [SymptomInputPage]**	**• Routing: Decision point triggers navigation to either Booking Flow or AI Analysis**
3. AI-Driven Symptom Analysis	**1. User types text → POST /api/ai/analyze/** **2. Backend prompts gpt-4o-mini (JSON Mode)** **3. AI returns {"clinics": ["KBB"]}** **4. Backend parses JSON & fetches Clinic ID** **5. Frontend redirects to [DoctorListPage]**	**• Input: Raw text entered** **• Processing: Loading state (External API Call)** **• Output: structured JSON response** **• Action: Auto-navigation to correct clinic**
4. Appointment Booking	**1. Select Slot → POST /api/appointments/create/** **2. Serializer checks schedule.is_booked** **3. DB Transaction: Locks row via OneToOneField** **4. Appointments Table → Record Inserted** **5. Schedules Table → is_booked = True**	**• Available: Slot visible, is_booked=False** **• Validating: Transaction initiated** **• Booked: is_booked=True, Appointment link created** **• Final: Slot hidden from other users**

3.4. Demo Requirement

We confirm that the following four core use cases are fully implemented and ready for live demonstration during the final project presentation. The demonstration will showcase the complete user journey from registration to booking, highlighting the system's robustness and data integrity features.

Demo Scenarios:

1. **Patient Onboarding:** Registering a new patient account and logging in to view the dashboard.
2. **Symptom-Based Routing:**
    - *Scenario A (Direct):* Selecting a polyclinic and verifying symptoms manually.
    - *Scenario B (AI):* Entering free-text symptoms (e.g., *"Şiddetli kulak ağrısı"*) and receiving an AI-generated clinic recommendation.
3. **Transactional Booking:** Successfully booking an appointment slot and verifying that the slot immediately becomes unavailable for other users.

Technical Verification during Demo: The demonstration will specifically validate the system's **concurrency handling** and **data integrity**:

- **Race Condition Prevention:** We will demonstrate that the database constraint (unique_together) and Django's OneToOneField logic successfully prevent double-booking, even if requested simultaneously.
- **Slot Locking:** The system will be shown to transition slot states instantly from Available (is_booked=False) to Booked (is_booked=True) upon successful transaction.

4. Design Decisions

4.1. Technology Comparisons

4.1.1 Backend Framework: Django vs. Flask

Feature	Django	Flask
ORM	**Built-in (powerful and robust)**	**Requires external libraries (e.g., SQLAlchemy)**
Admin Panel	**Automatic generation (ready-to-use)**	**Manual setup and coding required**
Authentication	**Built-in system + JWT libraries**	**Custom implementation required**
REST API	**Django REST Framework (mature ecosystem)**	**Flask-RESTful (lightweight extension)**
Learning Curve	**Steeper (opinionated structure)**	**Gentler (flexible structure)**

Feature	Django	Flask
Scalability	Excellent for medium-large monolithic apps	Better suited for microservices architectures

4.1.2 Database: MySQL vs. PostgreSQL

Aspect	MySQL (Selected)	PostgreSQL
Architecture	Relational (RDBMS)	Object-Relational (ORDBMS)
Read Speed	Highly optimized for read-heavy workloads (e.g., fetching schedules).	Better for complex queries and write-heavy loads.
Complexity	Simpler to set up and maintain.	Steeper learning curve, more advanced features.
Team Skill	High familiarity within the team.	Moderate familiarity.

4.2. Decision Justifications

- ### 4.2.1 Database Choice (MySQL)
  We selected MySQL as our primary relational database management system because our application's core data structure—consisting of Users, Doctors, and Appointments—is highly structured and requires strict transactional integrity. MySQL is renowned for its speed and reliability in read-heavy environments, which perfectly matches our system's usage pattern where checking doctor schedules is far more frequent than booking appointments. Furthermore, its ACID compliance guarantees that critical operations, such as preventing double-booking of the same time slot, are handled securely, while its seamless integration with Django's ORM and our team's existing proficiency with SQL syntax minimized the learning curve and accelerated deployment.

- ### 4.2.2. Backend Choice (Django)

We selected Django as the backend framework for the Clinical Appointment System due to its "batteries-included" philosophy, which accelerates development without sacrificing robustness. Its powerful Object-Relational Mapping (ORM) system abstracts complex SQL queries to ensure data integrity, while the built-in, production-ready administration interface allows hospital administrators to manage doctors and schedules immediately without extra development effort. Furthermore, Django meets strict health data security standards by providing default protection against common vulnerabilities such as CSRF, XSS, and SQL injection, and the Django REST Framework (DRF) enables the rapid development of scalable APIs for our frontend and AI modules.

- ### 4.2.3. External API Choice (OpenAI API)

  We chose OpenAI's API over rule-based systems or custom trained models because of its superior Natural Language Understanding capabilities, which effectively handle varied symptom descriptions (e.g., distinguishing between "my head hurts a lot" and "I have a headache"). Its context awareness allows it to analyze symptom combinations intelligently—for instance, directing a patient with "fever and cough" to a different clinic than one with "fever alone." This approach offers a maintenance-free solution that requires no manual rule updates for new patterns, significantly enhancing the user experience by allowing patients to describe their symptoms naturally without being forced into rigid keyword constraints.

- ### 4.2.4. Authentication Choice (JWT)

  We opted for JWT (JSON Web Tokens) over traditional session-based authentication to ensure a stateless and scalable architecture. Unlike sessions, which require server-side storage lookup for every request, JWTs are self-contained tokens that carry necessary user claims, reducing server load and latency. This method is particularly advantageous for our decoupled architecture (React frontend + Django backend), as it simplifies authentication across different domains and allows for seamless future integration with mobile applications without complex session management.

- ### 4.2.5. Frontend Choice (JavaScript / React)

  We selected JavaScript (utilizing the React library) for the frontend to deliver a highly responsive and dynamic user interface, which is essential for a smooth appointment booking experience. React's component-based architecture allows us to build reusable UI elements such as the dynamic appointment calendar and interactive symptom form ensuring code maintainability and visual consistency. Additionally, its Virtual DOM implementation ensures high performance by efficiently updating only the changed parts of the page, while the vast JavaScript ecosystem provides robust libraries for state management and API integration, enabling seamless real-time communication with our Django backend.

5. GitHub Commit Requirement

This section confirms that all code implementations, interfaces, and documentation required for Programming Assignment 2 are uploaded and traceable in the GitHub repository.

5.1 Code Implementations & Interfaces

All source code (Frontend, Backend applications, and SQL schema) and visual interface designs (screenshots) have been uploaded to the designated project repository. This includes:

- The final MySQL Database Schema as defined in Section 2.2.1
- Component Interfaces (API endpoints) as defined in Section 2.3.
- Key Implementations (AI prompt logic, Slot Locking logic) as defined in Section 2.2.

6. References

- BIL 481 - Programming Assignment 2
- IEEE 830-1998 / ISO/IEC/IEEE 29148:2018 (Software Requirements Standard)
- MySQL Official Documentation
- OpenAI API Documentation