THE NATIONAL INSTITUTE OF ENGINEERING (North)
MYSURU-570008, KARNATAKA, INDIA
2024-25

DEPARTMENT OF INFORMATION SCIENCE & ENGINEERING
THE NATIONAL INSTITUTE OF ENGINEERING MYSURU –570008

Report on

"Hospital Management System"

Submitted in the partial fulfillment of the requirements of the

Experiential Learning Event for course

(Software Architecture and Design Patterns-BIS605)

Submitted by

Batch-2			
Name	***USN***	***Name***	***USN***
SATWIK R KINI	4NI22IS193	SUHAS JAVALI	4NI22IS219
SHREEDHAR	4NI22IS203	SHRAVAN L GOWDA	4NI22IS201
VISHWAS B	4NI22IS245	SUMUKH P B	4NI22IS222
VISHWAS JOSHI	4NI22IS234	VIJAY TEJAS WALISHETTAR	4NI22IS239
SUHAS N	4NI22IS220	SUPRIYA G	4NI22IS223

Submitted by
Under the guidance of

Nandini P

ASSISTANT PROFESSOR
Department of IS&E,
The National Institute of Engineering, Mysuru.

Name of the Examiners **Signature with Date**

1.

CONTENTS

Chapter 1: Intent

The Model-View-Template (MVT) architectural pattern, as implemented in Django, is designed to promote separation of concerns within a web application. It divides the application into three distinct layers:

- Model – handles the data and business logic
- View – manages the logic for processing user requests
- Template – focuses on how data is presented to the user

This separation facilitates parallel development, increases modularity, and ensures that updates or modifications to one layer have minimal impact on the others.

In the context of a Hospital Management System (HMS), where multiple stakeholders (Admins, Doctors, Patients) interact with complex datasets (appointments, diagnoses, billing), MVT helps maintain structure and clarity. This pattern enables robust scalability, faster development cycles, and easy integration of additional features without disturbing existing code.

Chapter 2: Problem

In large-scale applications like hospital management platforms, combining business logic, UI presentation, and database operations in a single layer leads to several issues:

- **Tightly Coupled Components:**

  If business logic is embedded within templates or view code directly manipulates database operations, any small change can trigger cascading modifications across the entire codebase.

- **Code Duplication:**

  Without structured patterns, similar logic (e.g., user data fetching or role verification) may be written repeatedly in various parts of the application.

- **Poor Testability:**

  Mixing logic and UI hinders the ability to test individual components in isolation, making automated testing harder.

- **Maintenance Bottlenecks:**

  Updating the user interface or modifying a data model may require touching multiple unrelated files.

- **Scalability Challenges:**

  Adding new modules or user types becomes increasingly complex, risking regression in unrelated parts of the application

Chapter 3: Solution

To address the operational, administrative, and security challenges inherent in traditional hospital workflows, the project proposes a multifaceted solution built upon the Django framework and the MVT (Model-View-Template) design pattern. The solution is structured into six core components that work synergistically to deliver a centralized, secure, and efficient Hospital Management System (HMS). The detailed components are as follows:

• **Centralized Patient and Doctor Records**:
A unified database is established to store all patient medical histories, prescriptions, billing details, and doctor profiles. This centralized approach enables real-time data access, enhances treatment accuracy, and minimizes paperwork, laying the foundation for coordinated patient care.

• **Automated Appointment Scheduling**:
The system facilitates appointment booking, modification, and cancellation by patients, while simultaneously providing doctors with a dynamic and up-to-date scheduling interface. Automated notifications are integrated to minimize missed appointments and improve time management for both patients and medical staff.

• **Integrated Billing System**:
Billing processes are seamlessly connected with healthcare services, including consultations, laboratory tests, and hospital admissions. Automatic invoice generation ensures precise billing and expedites payment workflows, reducing administrative overhead.

• **Real-Time Patient Monitoring**:
A real-time dashboard offers continuous insights into patient admissions, discharges, and room occupancy. This enables medical staff to access and update treatment status promptly, improving the coordination and quality of patient care.

• **Role-Based Access and Data Security**:
Robust access control mechanisms ensure that sensitive patient and operational data are accessible only to authorized personnel. User roles—such as admin, doctor, patient, and receptionist—are carefully managed through secure login systems and encryption, safeguarding data integrity and confidentiality.

• **Responsive and User-Friendly Interface**:
The platform is designed with a clean, intuitive interface optimized for both desktop and mobile environments. This responsive design enhances user experience for both hospital staff and patients, reducing training requirements and improving system adoption.

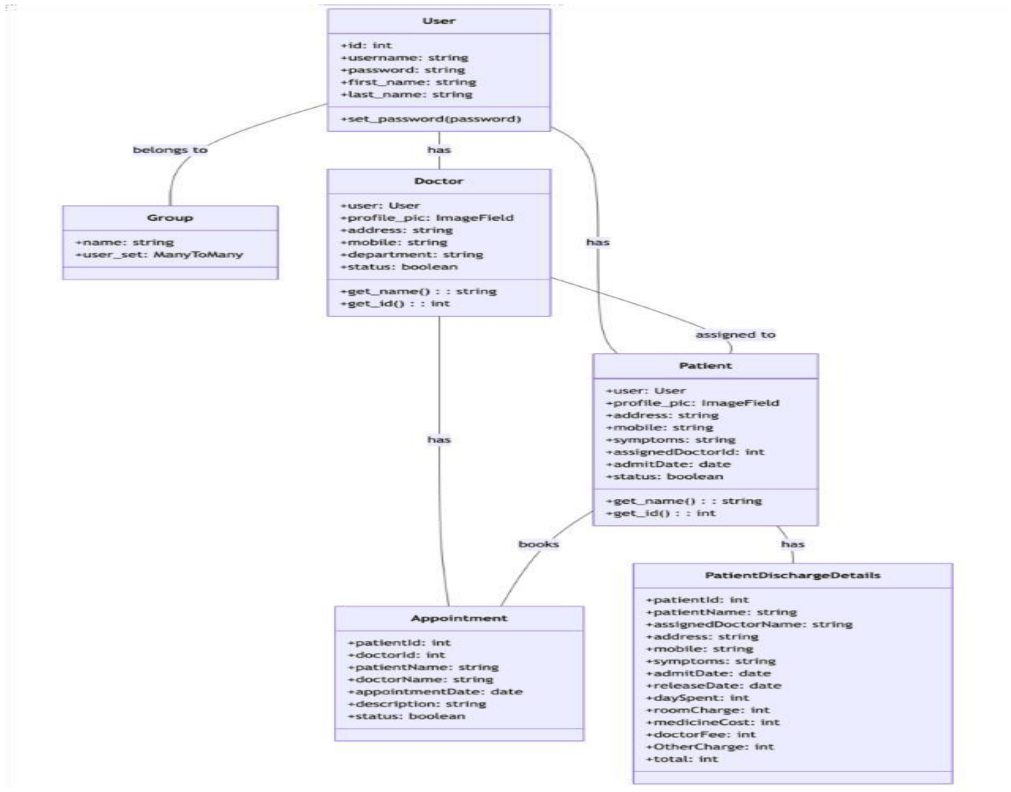
By leveraging these integrated components, the Hospital Management System offers a scalable, secure, and user-centric solution that transforms hospital operations and elevates patient care.

Chapter 4: Structure

Overall Architecture:

Layered (N-Tier) Architecture: The system can be organized into different layers where each layer has a distinct responsibility.

1. Presentation Layer: Handles user interaction and displays relevant information through the web interface. This includes interfaces for patients (e.g., appointment booking, viewing prescriptions), doctors (e.g., patient records, schedules).
2. Application Layer: Manages the core functionality of the HMS, such as appointment scheduling, billing operations, real-time monitoring, notification services, and role-based access control. It also coordinates interactions between the user interface and the underlying business logic
3. Domain Layer: Encapsulates the hospital's business logic, including patient record management, doctor profiles, appointment workflows, billing calculations, room occupancy management, and reporting.
4. Persistence Layer: Responsible for database interactions, including storing and retrieving patient records, doctor information, appointments, invoices, treatment histories, and system logs.



Chapter 5: Applicability

1) Facade Pattern:

The Facade Pattern is useful in the HMS when:
• You have a complex system with multiple interdependent modules (e.g., patient management, appointment scheduling, billing, reporting, and authentication).
• You want to simplify the interaction between the user interface and the underlying system.
• You need to encapsulate the internal workings of the hospital modules and provide a clean, unified interface for clients (such as frontend components or external systems).

Design Benefits:
• **Simplified Client Interaction**: The client (UI or API consumer) interacts with a single interface, such as a central controller or service layer, rather than directly dealing with multiple modules like patient services, doctor services, billing services, etc.
• **Reduced Coupling**: The client is decoupled from the complexity of individual subsystems (e.g., appointment logic, billing calculations, record management), making the system more modular and flexible.

2) Observer Pattern:

The Observer Pattern is beneficial in the HMS when:
• You want to notify multiple system components when key events occur (e.g., appointment bookings, patient admissions, billing completion, discharge updates).
• You need to decouple the components that trigger events (such as the appointment scheduler or billing processor) from the components that respond to those events (such as notification services, reporting modules, or dashboard updates).

Design Benefits:
• **Loose Coupling**: Observers (e.g., notification service, admin dashboard, reporting system) do not need to know the internal details of the modules that trigger events (e.g., appointment or billing services).
• **Scalability**: You can add more observers, such as SMS alerts, email notifications, or audit logging, without changing the core business logic.
• **Flexibility**: It’s easy to extend system behaviour when new events are introduced — for example, adding a new observer to track room occupancy or to send post-discharge surveys.

3) Decorator Pattern:

The Decorator Pattern is useful in the HMS when:
• You want to dynamically add additional features or responsibilities to objects (e.g., enhancing invoices, extending patient records, or customizing notifications) without modifying their core implementation.
• You need flexible and reusable ways to extend system behavior at runtime.

Design Benefits:
• **Enhanced Flexibility**: You can add features like discounts, taxes, or insurance details to billing invoices without changing the base billing class.
• **Single Responsibility Principle**: The core classes (like Patient, Invoice, Notification) stay focused on their main responsibilities, while decorators handle additional concerns.
• **Runtime Behavior Change**: You can layer multiple decorators — for example, sending an appointment reminder via both email and SMS by wrapping the notification object.

4) Factory Method Pattern:

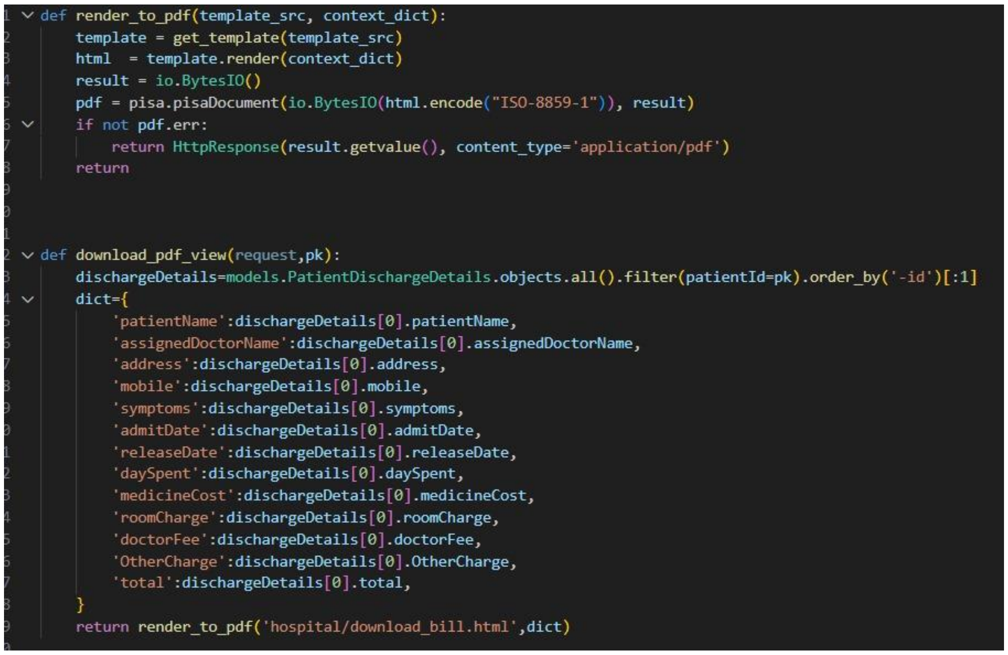
The Factory Method Pattern is useful in the HMS when:
• You need to create objects such as user accounts, appointments, or reports, but want to delegate the instantiation process to specialized factory methods.
• You want to encapsulate object creation logic, especially when creating families of related objects (e.g., different types of users or reports).

Design Benefits:
• **Encapsulation of Object Creation**: The logic for creating objects like Admin, Doctor, Patient, or Receptionist accounts is moved into factories, keeping client code clean.
• **Scalability**: Adding new user roles, report types, or appointment types requires minimal changes to the factory, without touching the rest of the system.
• **Improved Maintainability**: Changes to how objects are instantiated (for example, adding default values or complex initialization) can be handled in one place — the factory.

Chapter 6: Implementation

1) Facade Pattern

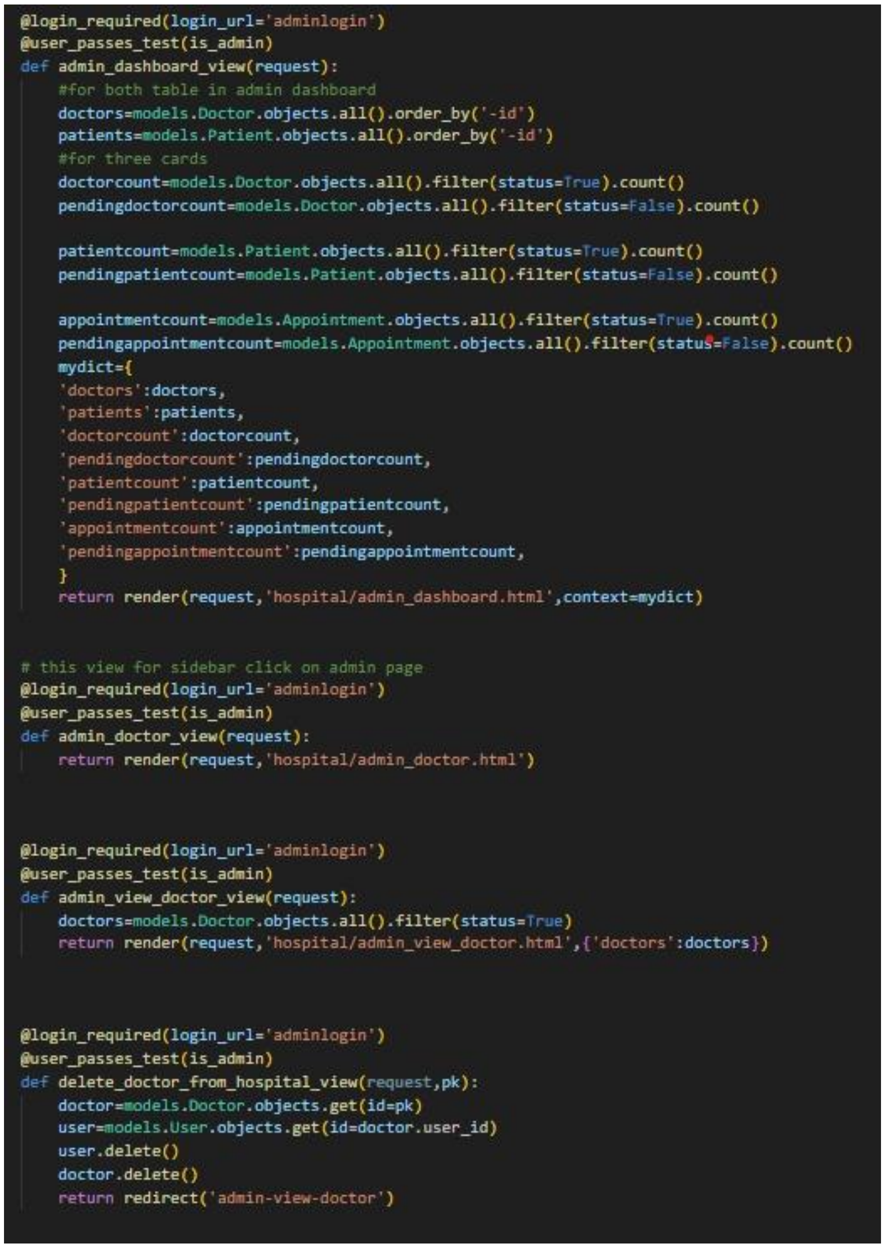
The Facade Pattern is useful in the HMS when:
• You have a complex system with multiple interdependent modules (e.g., patient management, appointment scheduling, billing, reporting, and authentication).
• You want to simplify the interaction between the user interface and the underlying system.
• You need to encapsulate the internal workings of the hospital modules and provide a clean, unified interface for clients (such as frontend components or external systems).



```
def render_to_pdf(template_src, context_dict):
    template = get_template(template_src)
    html  = template.render(context_dict)
    result = io.BytesIO()
    pdf = pisa.pisaDocument(io.BytesIO(html.encode("ISO-8859-1")), result)
    if not pdf.err:
        return HttpResponse(result.getvalue(), content_type='application/pdf')
    return



def download_pdf_view(request,pk):
    dischargeDetails=models.PatientDischargeDetails.objects.all().filter(patientId=pk).order_by('-id')[:1]
    dict={
        'patientName':dischargeDetails[0].patientName,
        'assignedDoctorName':dischargeDetails[0].assignedDoctorName,
        'address':dischargeDetails[0].address,
        'mobile':dischargeDetails[0].mobile,
        'symptoms':dischargeDetails[0].symptoms,
        'admitDate':dischargeDetails[0].admitDate,
        'releaseDate':dischargeDetails[0].releaseDate,
        'daySpent':dischargeDetails[0].daySpent,
        'medicineCost':dischargeDetails[0].medicineCost,
        'roomCharge':dischargeDetails[0].roomCharge,
        'doctorFee':dischargeDetails[0].doctorFee,
        'OtherCharge':dischargeDetails[0].OtherCharge,
        'total':dischargeDetails[0].total,
    }
    return render_to_pdf('hospital/download_bill.html',dict)
```

2) Decorator Pattern

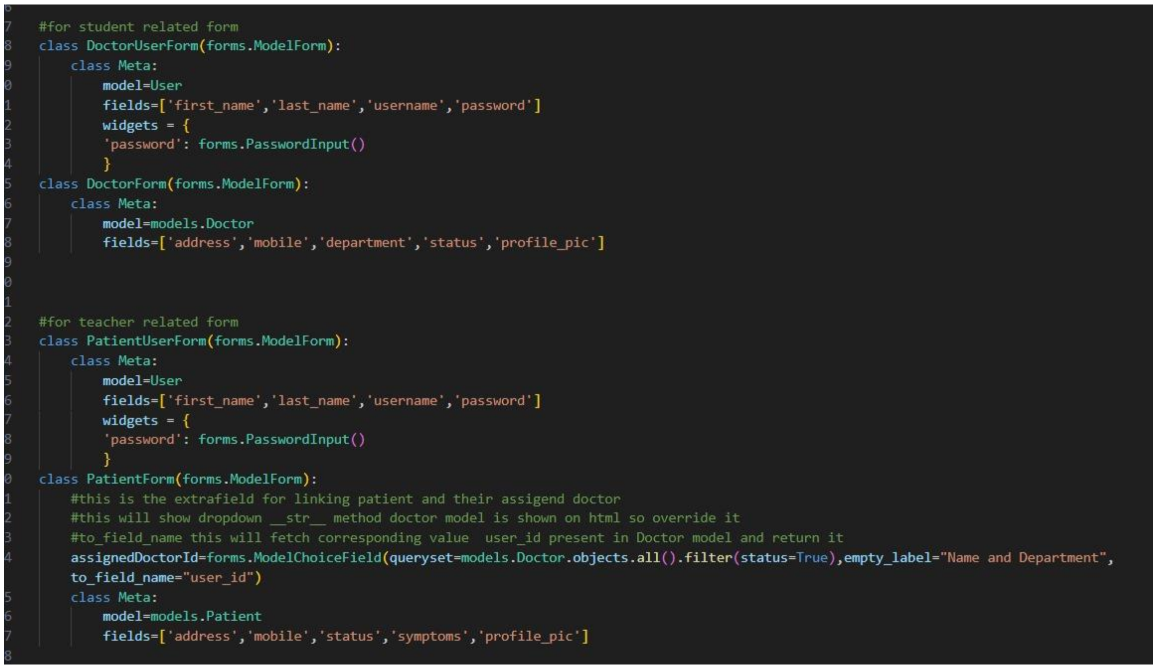
The Decorator Pattern is useful in the HMS when:
• You want to dynamically add additional features or responsibilities to objects (e.g., enhancing invoices, extending patient records, or customizing notifications) without modifying their core implementation.
• You need flexible and reusable ways to extend system behavior at runtime.



```python
@login_required(login_url='adminlogin')
@user_passes_test(is_admin)
def admin_dashboard_view(request):
    #for both table in admin dashboard
    doctors=models.Doctor.objects.all().order_by('-id')
    patients=models.Patient.objects.all().order_by('-id')
    #for three cards
    doctorcount=models.Doctor.objects.all().filter(status=True).count()
    pendingdoctorcount=models.Doctor.objects.all().filter(status=False).count()

    patientcount=models.Patient.objects.all().filter(status=True).count()
    pendingpatientcount=models.Patient.objects.all().filter(status=False).count()

    appointmentcount=models.Appointment.objects.all().filter(status=True).count()
    pendingappointmentcount=models.Appointment.objects.all().filter(status=False).count()
    mydict={
    'doctors':doctors,
    'patients':patients,
    'doctorcount':doctorcount,
    'pendingdoctorcount':pendingdoctorcount,
    'patientcount':patientcount,
    'pendingpatientcount':pendingpatientcount,
    'appointmentcount':appointmentcount,
    'pendingappointmentcount':pendingappointmentcount,
    }
    return render(request,'hospital/admin_dashboard.html',context=mydict)


# this view for sidebar click on admin page
@login_required(login_url='adminlogin')
@user_passes_test(is_admin)
def admin_doctor_view(request):
    return render(request,'hospital/admin_doctor.html')



@login_required(login_url='adminlogin')
@user_passes_test(is_admin)
def admin_view_doctor_view(request):
    doctors=models.Doctor.objects.all().filter(status=True)
    return render(request,'hospital/admin_view_doctor.html',{'doctors':doctors})



@login_required(login_url='adminlogin')
@user_passes_test(is_admin)
def delete_doctor_from_hospital_view(request,pk):
    doctor=models.Doctor.objects.get(id=pk)
    user=models.User.objects.get(id=doctor.user_id)
    user.delete()
    doctor.delete()
    return redirect('admin-view-doctor')
```

3) Factory-Method Pattern

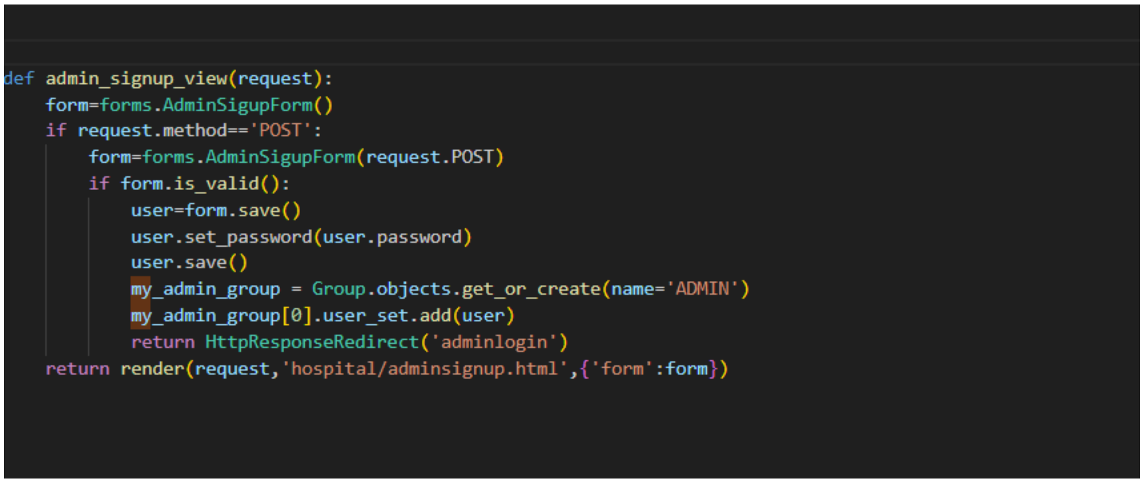
The Factory Method Pattern is useful in the HMS when:
• You need to create objects such as user accounts, appointments, or reports, but want to delegate the instantiation process to specialized factory methods.



```python
#for student related form
class DoctorUserForm(forms.ModelForm):
    class Meta:
        model=User
        fields=['first_name','last_name','username','password']
        widgets = {
        'password': forms.PasswordInput()
        }
class DoctorForm(forms.ModelForm):
    class Meta:
        model=models.Doctor
        fields=['address','mobile','department','status','profile_pic']



#for teacher related form
class PatientUserForm(forms.ModelForm):
    class Meta:
        model=User
        fields=['first_name','last_name','username','password']
        widgets = {
        'password': forms.PasswordInput()
        }
class PatientForm(forms.ModelForm):
    #this is the extrafield for linking patient and their assigend doctor
    #this will show dropdown __str__ method doctor model is shown on html so override it
    #to_field_name this will fetch corresponding value  user_id present in Doctor model and return it
    assignedDoctorId=forms.ModelChoiceField(queryset=models.Doctor.objects.all().filter(status=True),empty_label="Name and Department",
    to_field_name="user_id")
    class Meta:
        model=models.Patient
        fields=['address','mobile','status','symptoms','profile_pic']
```

4) Observer Pattern

The Observer Pattern is beneficial in the HMS when:
• You want to notify multiple system components when key events occur (e.g., appointment bookings, patient admissions, billing completion, discharge updates).



```python
def admin_signup_view(request):
    form=forms.AdminSigupForm()
    if request.method=='POST':
        form=forms.AdminSigupForm(request.POST)
        if form.is_valid():
            user=form.save()
            user.set_password(user.password)
            user.save()
            my_admin_group = Group.objects.get_or_create(name='ADMIN')
            my_admin_group[0].user_set.add(user)
            return HttpResponseRedirect('adminlogin')
    return render(request,'hospital/adminsignup.html',{'form':form})
```

Chapter 7: Result

The implementation of the Hospital Management System using the MVT pattern in Django has produced the following key outcomes:

1. **Separation of Concerns**: Clear separation between models, views, and templates enhances maintainability and readability.
2. **Improved Development Efficiency**: Frontend and backend developers can work in parallel, accelerating development.
3. **Role-Based Accessibility**: Secure and appropriate data access is ensured for Admins, Doctors, and Patients.
4. **Enhanced User Experience**: Responsive dashboards and real-time updates offer a smoother, more intuitive interface.
5. **Scalability**: Modular design allows new features and functionalities to be added with minimal code disruption.
6. **Robust Data Management**: Efficient handling of appointments, discharges, invoices, and user information with relational database integrity.
7. **Administrative Ease**: The Django admin panel streamlines hospital operations through simple yet powerful controls.
8. **Security Assurance**: Built-in Django features and custom validations ensure protection against common web threats.

Chapter 8: Snapshots

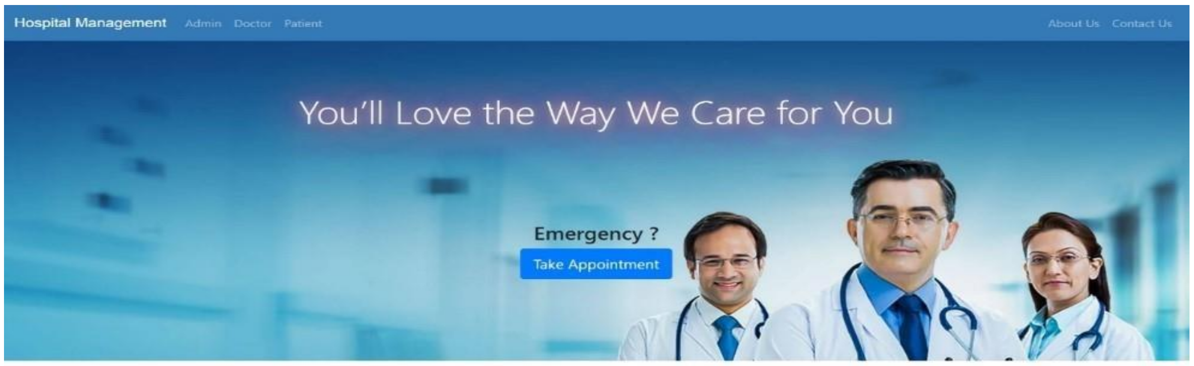
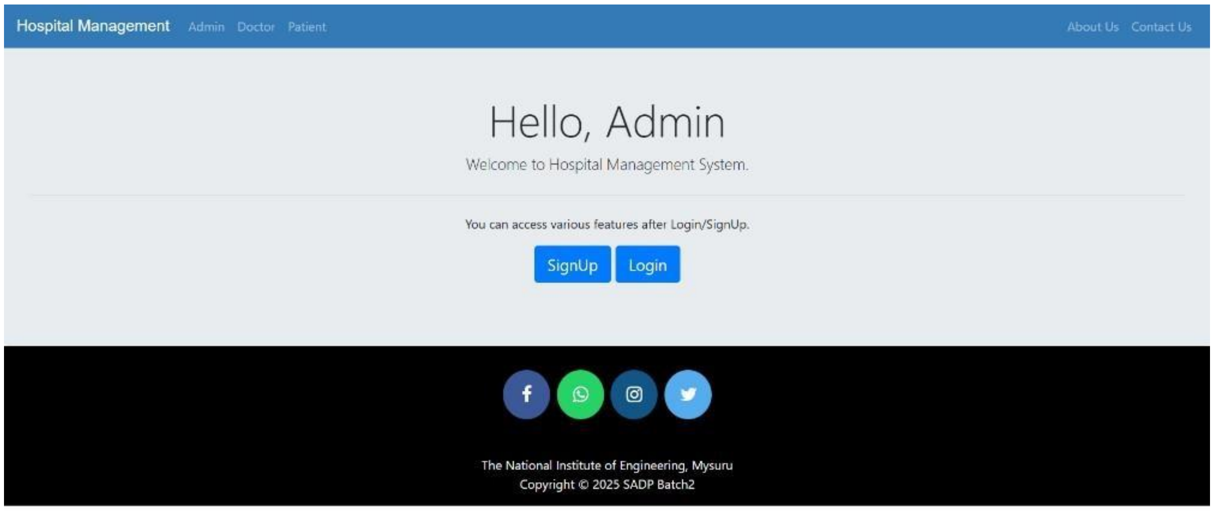


Fig 8.1 Home Page



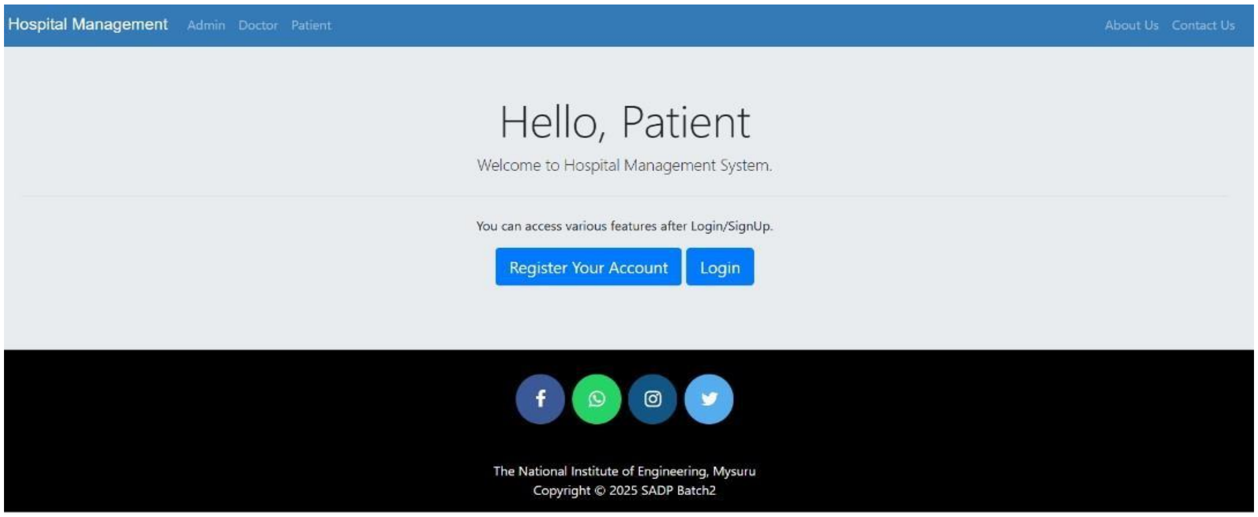
Hello, Admin

Welcome to Hospital Management System.

You can access various features after Login/SignUp.

The National Institute of Engineering, Mysuru
Copyright © 2025 SADP Batch2

Fig 8.2 Admin Signup


Hospital Management Admin Doctor Patient About Us Contact Us

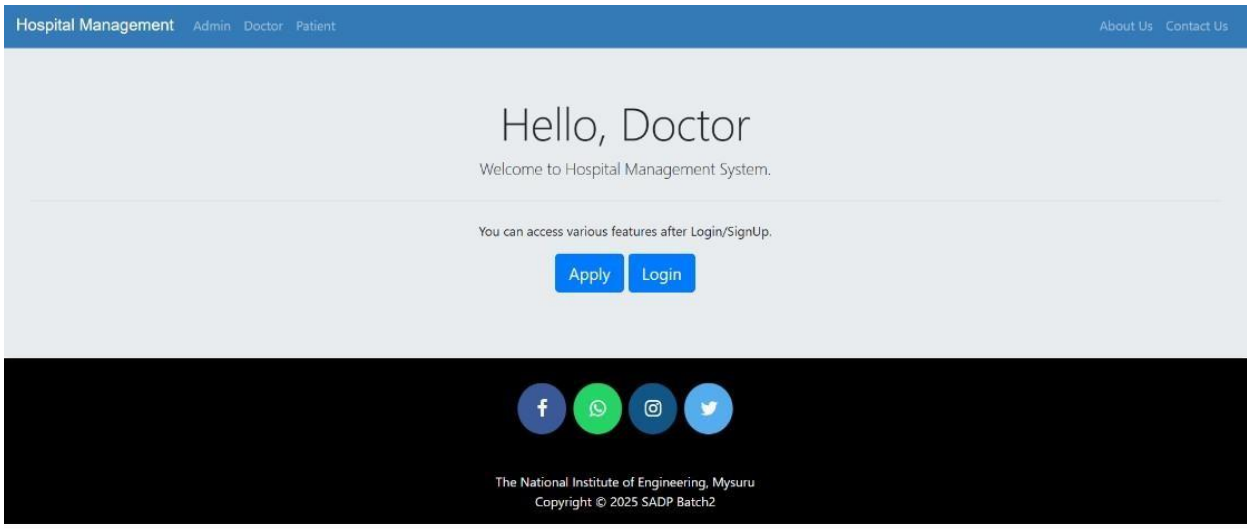
Hello, Patient

Welcome to Hospital Management System.

You can access various features after Login/SignUp.

The National Institute of Engineering, Mysuru
Copyright © 2025 SADP Batch2

Fig 8.3 Register Patient


Hospital Management Admin Doctor Patient About Us Contact Us

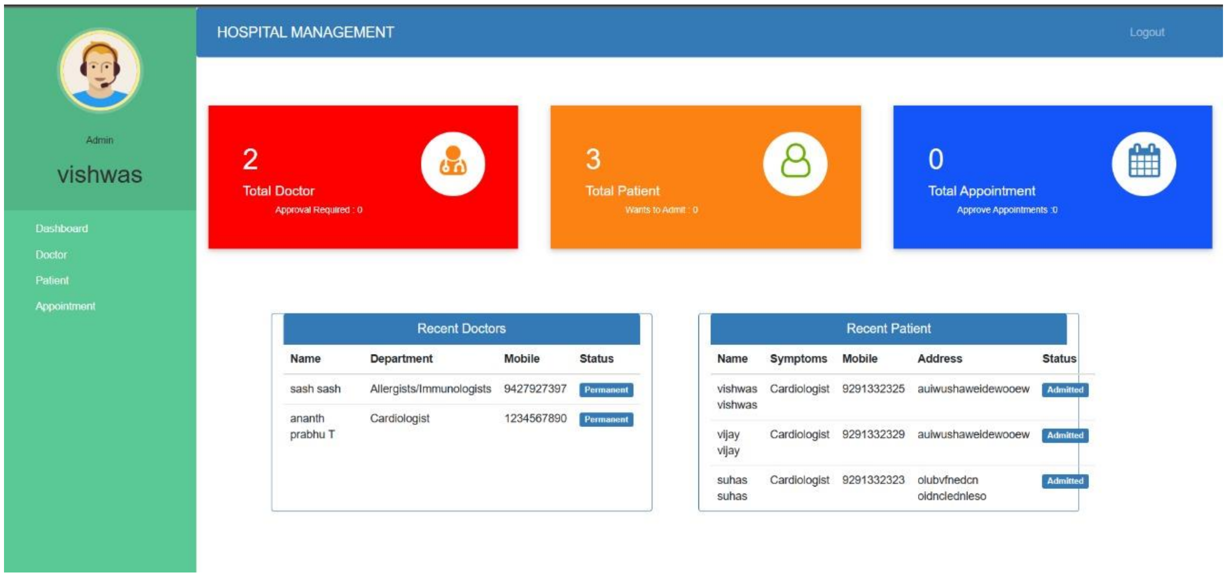
Hello, Doctor

Welcome to Hospital Management System.

You can access various features after Login/SignUp.

The National Institute of Engineering, Mysuru
Copyright © 2025 SADP Batch2

Fig 8.4 Docter Signup



2
Total Doctor
Approval Required : 0

3
Total Patient
Wants to Admit : 0

0
Total Appointment
Approve Appointments :0

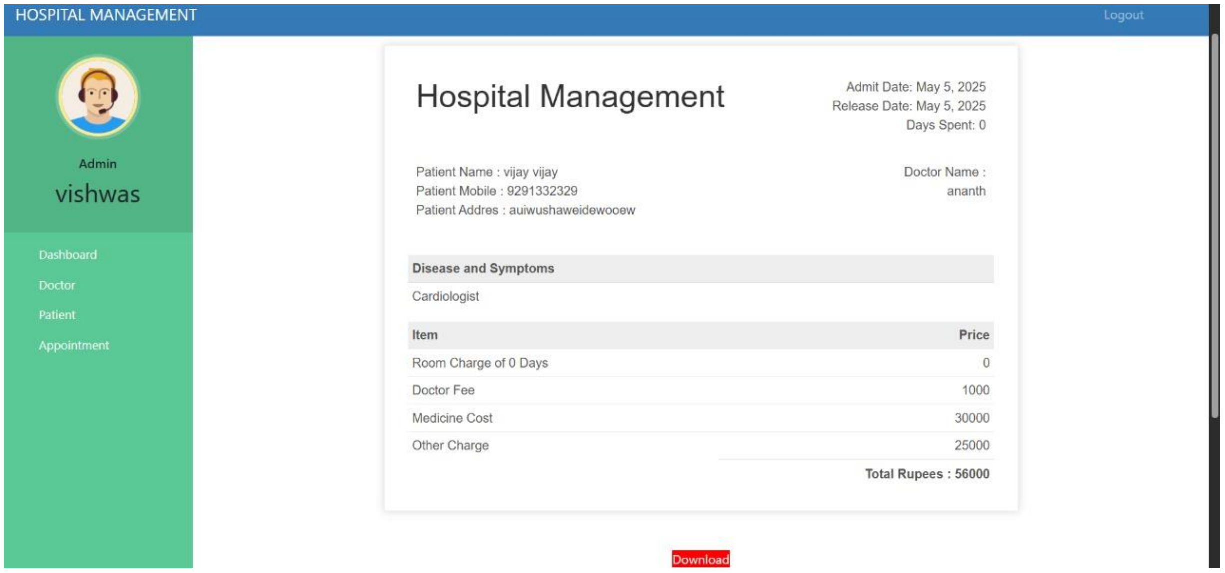
Recent Doctors

Name	Department	Mobile	Status
sash sash	Allergists/Immunologists	9427927397	Permanent
ananth prabhu T	Cardiologist	1234567890	Permanent

Recent Patient

Name	Symptoms	Mobile	Address	Status
vishwas vishwas	Cardiologist	9291332325	auiwushaweidewooew	Admitted
vijay vijay	Cardiologist	9291332329	auiwushaweidewooew	Admitted
suhas suhas	Cardiologist	9291332323	olubvfnedcn oidncledнleso	Admitted

Fig 8.5 Admin DashBoard



Hospital Management

Admit Date: May 5, 2025
Release Date: May 5, 2025
Days Spent: 0

Patient Name : vijay vijay
Patient Mobile : 9291332329
Patient Addres : auiwushaweidewooew

Doctor Name :
ananth

Disease and Symptoms
Cardiologist

Item	Price
Room Charge of 0 Days	0
Doctor Fee	1000
Medicine Cost	30000
Other Charge	25000
	Total Rupees : 56000

Fig 8.6 Billing

Chapter 9: Conclusion

In conclusion, the Hospital Management System successfully fulfills its core objectives of centralized patient management, efficient scheduling, secure billing, and real-time hospital operations. The choice of a modular MVT architecture, supported by proven design patterns such as Decorator, Observer, Factory, and Facade, ensures the system is both scalable and maintainable. By integrating role-based access and secure authentication, the system provides a reliable and user-friendly solution for administrators, doctors, and patients.

Future Directions:

1) **Cloud Integration**: For large-scale deployments, integrating with cloud platforms such as AWS or Azure could enable dynamic scalability, data backup, and remote system access across multiple hospital branches.
2) **Real-Time Monitoring with IoT**: Connecting the system to IoT-enabled medical devices (e.g., patient monitors, smart beds) could enhance real-time patient monitoring and automate alerts for critical conditions.
3) **AI-Driven Analytics**: Incorporating machine learning models for predictive analytics could help forecast patient trends, optimize resource allocation, and improve clinical decision-making.
4) **Mobile App Integration:** Developing mobile applications for doctors, patients, and staff would improve accessibility, allowing appointment management, notifications, and record access on the go.

References

1. Design Patterns: Elements of Reusable Object-Oriented Software by Erich Gamma, Helm, Johnson, and Vlissides
2. Django Documentation: https://docs.djangoproject.com/en/stable/
3. Python Design Patterns: https://python-patterns.guide/
4. Software Architecture Patterns by Mark Richards 5. Django Models and Databases:https://docs.djangoproject.com/en/stable/topics/db/models/