Concert Hall Booking System

CS5721 Team-Based Project Report

Lecturer: J.J. Collins

Submitted by: Team VonNeumann

Adarsh Ajit (24026492)
Anjana Thalappilly Anildas (24042714)
Anushree Vinod (24158178)
Mahika Jaguste (24016454)

Contents

1 Project Overview

The Concert Hall Booking System is a web-based platform that aims to provide a complete solution to manage shows and book tickets for the shows hosted by concert halls. The system caters to three key stakeholders namely, show producers, customers, and admins. Show producers can use the platform to raise requests for hosting shows and manage them. Customers can view scheduled shows, book tickets for them and purchase memberships for added benefits. Admins can supervise all aspects of the Concert Hall Booking platform. The platform features an approval engine, that processes show requests created by producers, and evaluates and approves or rejects them without manual intervention. Additionally, a recommendation system is integrated that suggests shows to a customer based on their previous booking history. A payment gateway handles all financial transactions on the platform including membership fees and ticket purchases. It is also responsible for initiating refunds to customers if a scheduled show has been cancelled.

The show producer can register on the platform for the first time and use these credentials to login to the system at any time. The Concert Hall Booking System, allows the show producer to raise a request by giving essential details such as the name of the show, category, required hall capacity, and time slot. A list of available halls will be displayed from which the producer can select one and submit the request. Show producer can view the current status of submitted requests, and update or cancel it if necessary. Additionally, the Concert Hall Booking System also allows the show producer to track ticket sales information of the requested shows.

A customer can register and login with the appropriate credentials into the Concert Hall booking platform. Once logged in, they can view shows recommended by the recommendation system and all the scheduled shows and book tickets for them. They can also select the seat type such as Regular and Premium each priced differently. Customers can view their upcoming tickets and payment history. The platform also allows them to cancel tickets.

The Concert Hall Booking system awards Loyalty points for each ticket booked depending on the seat type selected. Customers are awarded extra loyalty points for their first booking. These loyalty points can be redeemed as Food coupons in the future. The platform also offers membership deals namely Silver and Gold tiers that provide them with multiple benefits. Memberships will enable customers to avail discounts on ticket prices (10% for the Silver tier and 15% for the Gold tier). They also get loyalty point boosters that provide extra loyalty points for members and varied cancellation policies as per their membership tiers.

Admins can oversee the complete management of the concert hall. They can login to view show requests made by producers, scheduled shows and tickets booked by customers. Admins also have the option to cancel any shows due to technical reasons.

2 Software Development Life Cycle (SDLC)

The Concert Hall Booking System platform that focuses on managing ticket booking, user login and registrations, show scheduling, slot allocation and so on. Software Development Life Cycle (SDLC) can help in developing a software product in a time and cost efficient way. The software lifecycle helps us to build quality software in a systematic and effective way. There are many SDLC models such as waterfall, iterative, V-shaped, spiral, agile, etc. which can help in the development of quality software.

The waterfall model follows a linear and successive approach in software development. This approach does not suit us as it is not flexible since it does not allow the inclusion of changes once the respective phase is over. The iterative SDLC model describes iterative development, with each iteration/cycle including a small set of software requirements. This model is not suitable for our booking platform since it is complex and interdependent, and changing requirements may further complicate the iterations.

A V-shaped model is a verification and validation model where the testing phase is associated with each development stage. Since it is not flexible, it is not suitable for our project. The spiral model is an iterative method that puts emphasis on risk analysis. Spiral's extensive planning and high emphasis on risk analysis make it not ideal for our booking platform since it can increase the complexity and time constraints of our platform.

Agile SDLC combines iterative and incremental process models with focus on collaborations, flexibility and customer satisfaction. Agile is suitable for changing requirements and helps in concurrent development. As per the Agile Manifesto, the importance is given to individuals and interactions over processes and tools, working software over comprehensive documentation, customer collaboration over contract negotiation, and responding to change over following a set plan[2]. Agile believes that the best architectures, requirements, and designs emerge from self-organizing teams[2]. This is ideal for our software development since it can help us perform well on our small dynamic ticket booking platform, giving importance to collaborations, fast actions, and efficient software.

In our project, the main concern was to act upon the changes of requirements or business logic and to tackle the whole development within the time frame. This we addressed by following the Scrum Framework with a sprint-based structure, with strict monitoring of deliverables and frequent standup meetings where the planning and design of the booking platform are discussed and decided. Regular feedback sessions during meetings helped us stay aligned with our requirements and priorities. Breaking down the work into sprints offered a structured way to measure progress and make necessary adjustments. Documenting key takeaways and refining agendas for future meetings provided a comprehensive overview of the project's trajectory, ensuring its timely and successful completion.

3 Project Plan

3.1 Roles

	Role	Description	Designated Team Member
1	Project Manager	Sets up group meetings, gets agreement on the project plan, and tracks progress.	Anjana
2	Documentation Manager	Responsible for sourcing relevant supporting documentation from each team member and composing it in the report.	Anushree
3	Business Analyst / Reqs. Engineer	Responsible for section 4 - Requirements.	Anushree
4	Architect	Defines system architecture.	Mahika
5	Systems Analysts	Creates conceptual class model.	Mahika, Anjana
6	Designer	Responsible for recovering design time blueprints from implementation.	Anushree, Anjana
7	Technical Lead	Leads the implementation effort.	Adarsh
8	Programmers	Each team member to develop at least 1 package in the architecture.	ALL
9	Tester	Coding of automated test cases.	ALL
10	Dev Ops	Must ensure that each team member is competent with development infrastructure, e.g., GitHub, etc.	Adarsh

Table 1: Roles, Descriptions, and Designated Team Members

3.2 Project plan

Deliverable	Team Member	Due Week
Front Cover	Anushree	4
Business Scenario	Anushree	6
Lifecycle Model	Anjana	6
Project Plan and Roles	Anushree, Mahika	6
Requirements and Use Cases	Team	6
Use Case Diagram	Anushree	6
Use Case Description	Anushree, Anjana	6
Non-Functional Requirements	Mahika, Adarsh	6
Quality Attributes	Mahika, Adarsh	6
GUI Prototypes	Adarsh	6

Architecture Patterns Considered	Mahika, Adarsh	6
System Architecture	Mahika, Adarsh, Anjana	6
Analysis Sketches	Team	7
Sequence Diagram	Adarsh, Anushree	7
Entity Relationship Diagram	Mahika	7
State Chart Diagram	Anushree	7
Transparency and Traceability	Anjana	12
CI/CD	Adarsh	8
Docker	Mahika	8
Database Implementation	Anushree	8
Automated Testing	Anjana	9
Refactoring	Team	From Week 8
Code Review	Team	From Week 8
Design-time Package Diagram	Anjana	12
Design-time Sequence Diagram	Anushree	12
Design-time Component Diagram	Anjana	12
Design-time Class Diagram	Team	12
Software Metrics	Team	12
References & Bibliography	Team	From Week 1

Table 2: Project Plan

3.3 Team Member Experience

Name	**Student ID**	**Experience**	**Domain/ Programming Languages/ Frameworks**
Adarsh Ajit	24026492	3 years	Frontend Web Development, Nodejs, Typescript, Python, React
Anjana Thalappilly Anildas	24042714	2 years	c#, Cucumber, SpecFlow, BDD
Anushree Vinod	24158178	2 years	IT Service Management, Python, Javascript, SQL, ServiceNow
Mahika Jaguste	24016454	1.5 years	Backend web development, Nodejs, Typescript, Python

Table 3: Team member experience

4 Requirements

4.1 Functional Requirements

For Show Producer

1. Show producer can register/login to the Concert Hall Booking platform.
2. Show producer can request to organize a show at available time slot and hall.
3. Show producer can update the request before or after approval (time slot and hall cannot be updated after approval).
4. Show producer can cancel the request.
5. Show producer is able to see the ticket sales for the scheduled show.
6. Show producer should be able to view their requests.

For Admin

1. Admin can login to the Concert Hall Booking platform.
2. Admin can cancel a scheduled show and initiate refund to customers.
3. View all scheduled shows.
4. View all ticket sales for scheduled shows.

For Customer

1. Customer can register/login to the Concert Hall Booking platform.
2. Customer should be able to view all the scheduled shows.
3. Customer can purchase a membership.
4. Customers can book tickets for a scheduled show.
5. Customer can view booked tickets for upcoming scheduled shows.
6. Customer can view the payment history.
7. Customer can cancel booked tickets.

4.2 Use Case Diagram

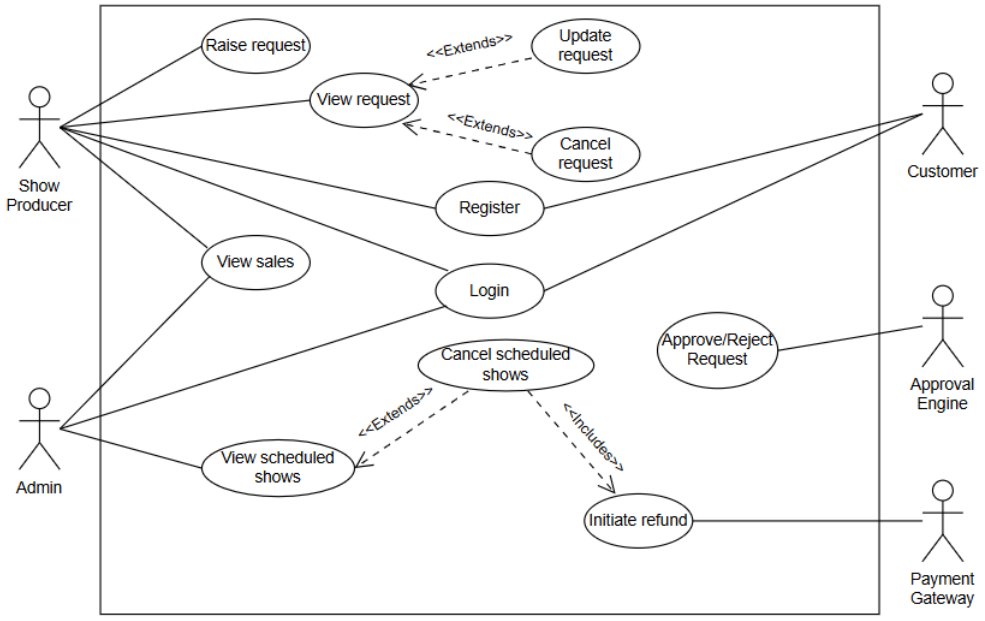


Figure 1: Use case diagram for Show producer and Admin

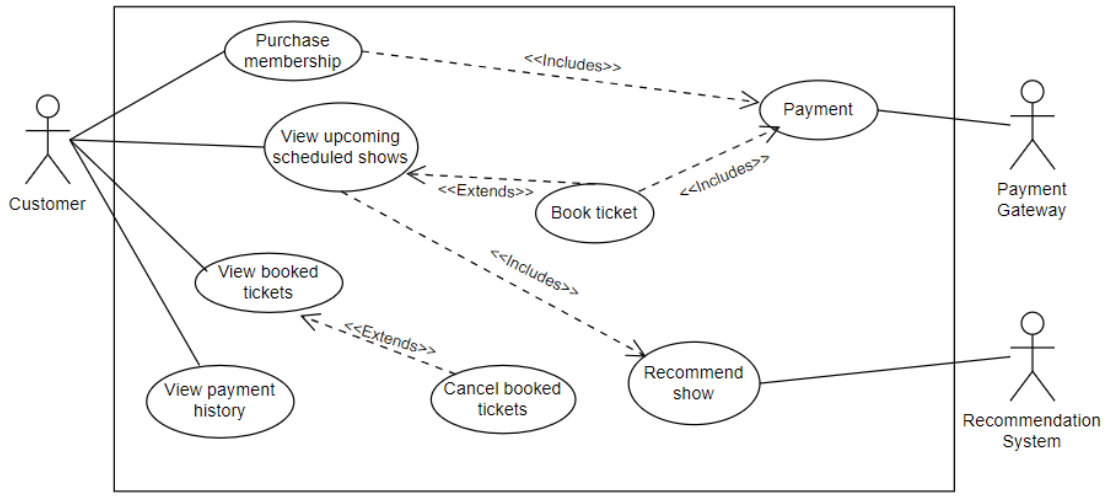


Figure 2: Use case diagram for Customer

4.3 Use Case Description

4.3.1 Raise Request

Use case description	**Raise request**	
Goal in Context	Show producer raises a request for showcasing a show in the concert hall by giving show details and number of seats.	
Scope & Level	System, Summary	
Preconditions	Show producer must be registered and logged into the system.	
Success End Condition	Request is raised for an available hall by a show producer, Approval engine should be able to approve or reject the request.	
Failed End Conditions	Request is not created, created request is not accessible to Approval engine to handle.	
Primary, Secondary Actors	Show producer, Approval engine	
Trigger	Show producer seeks to raise a request to showcase a show	
Description	**Step**	**Action**
	1	Show producer navigates to the request show section.
	2	Show producer gives details of the show such as name, category, date, slot, venue and number of seats.
	3	System shows the available halls.
	4	Show producer selects a preferred hall from the list of available halls and submits the request.
	5	System registers the request and makes it available for the approval engine to decide.
Extensions	3a	If no halls are available in the time, category, and number of seats, the system informs the producer that a hall is unavailable for the given specification.
Variations	None	
Priority	Top	
Performance	Less than two seconds for request creation.	
Frequency	10/day	
Open Issues	If a show producer wants to request multiple show timings or multiple show types?	
Due date	Week 9	

Table 4: Use Case: Raise Request

4.3.2 Book Ticket

Use case description	**Book Ticket**	
Goal in Context	Customer purchases tickets for seats from the available list of shows, including the perks and concessions.	
Scope & Level	System, Summary	
Preconditions	The customer has logged in.	
Success End Condition	Purchased tickets are added to the system and corresponding seats are marked sold. The tickets are successfully purchased, and the customer can view the ticket details including the show name, seat number, date, and time.	
Failed End Conditions	Payment fails during the checkout.	
Primary, Secondary Actors	Customer, Recommendation system, Payment gateway	
Trigger	Customer initiates the process of purchasing a ticket at the Concert Hall Booking Platform.	
Description	**Step**	**Action**
	1	Customer logs in to the system.
	2	The system displays a list of recommended and available shows at the Concert Halls.
	3	Customer selects a show and seats.
	4	The total cost is displayed after deducting concession if any, if the user falls in any of the membership packages.
	5	Customer proceeds to checkout, system processes the payment and confirms the purchase, and the ticket details are shown to the customer.
Extensions	3a	If seats are fully booked for the selected show, the system informs the user that tickets are booked.
	5a	If payment is failed, the user is given an option to retry.
Variations	None	
Priority	Top	
Performance	1 minute for booking and 5 minutes for payment	
Frequency	150/day	
Open Issues	What if the show or the chosen seat is no longer available during the checkout process? What happens if the payment fails and the seat is taken by other users?	
Due date	Week 9	

Table 5: Use Case: Book Ticket

4.4 Non Functional Requirements

1. During normal conditions with 200-page visits per day, the response time for any request should be at most 2 seconds.
2. During peak traffic, when a popular show goes live, around 1k people can book a show ticket simultaneously. The latency should not exceed 5 seconds.
3. The application must be easy to use with clarity on timings, show availability and ticket prices to facilitate a smooth ticket booking experience.
4. For the analytics dashboards, the system should refresh and display updated data every 10 minutes.
5. The system should operate 24*7 with a maximum downtime of 1 hour per month.

4.4.1 Quality Attributes

1. Availability

   It refers to the property of a system to be ready to carry out its task when you need it to; and the notion of recovery, by which the system can repair itself if it breaks [12]. Most enterprises use cloud computing instead of taking on the risk and expense of managing in-house servers. The cloud services provide service-level agreements/contracts which promise metrics such as uptime and response time.

   Apart from recovering from hardware faults, there are tactics to detect and recover from software upgrades. Monitoring systems can be used to analyze log messages so that we can identify potential issues early. Docker Orchestration tools is used to restart failed containers to minimize downtime and failures.

   When a message is consumed from the queue for processing and if it fails due to some network issues or any other issues in the database side, it will move to the retry phase. If it still fails it goes to the DLQ. These tactics ensures that the concert hall booking system will be able to withstand any unexpected issues.

2. Extensibility

   It is the quality attribute which provides for the future growth and expansion of the system, via the addition of new capabilities. One of the primary design principles of the Django stack is loose coupling and tight cohesion which enable extensibility - the different layers of the framework (model, template, view) need not know about each other.

   Event-driven architecture is a software architecture paradigm which uses events to publish changes and communicate amongst decoupled services, commonly built as microservices. They act like plugins - more functionalities can be easily added, and existing functionality can be substituted or deleted.

Following design principles such as Program to Interfaces, not Implementations, and Replace conditional with polymorphism, we strive to write extensible code so that extensions such as adding new Membership tiers, supporting more than one concert hall, modifications in loyalty point business rules can be supported easily.

3. Scalability

   Once our application becomes popular, we need to scale the resources running our application. It could be scaled horizontally by running instances of our stateless application on multiple servers. Introducing a new layer, the load balancer, will help to route huge volumes of incoming traffic equally among available servers.

   We might need to scale our database vertically to support efficient operations on huge volumes of data and further implement sharding. Caching responses is a popular technique to reduce load on our server.

   Cloud services provide options to set up auto-scaling which will scale or shrink your resources when certain thresholds are reached. Using microservices lets us decouple components such that we can scale only those services experiencing high traffic as opposed to scaling the entire system, thus resulting in cost reductions.

4. Portability

   When large teams across the globe are working on an enterprise project, it is essential that the software can run on any operating system. In our team, 2 members own devices running on the Windows operating system and 2 members own devices running on MacOS. When deploying the application, it is ideal that we are not restricted to a particular machine specification. Docker encapsulates the application code, its dependencies and everything needed to run it into a standard software unit called containers which can run cross-platforms, supporting application portability.

4.5 GUI Prototype

Figure 3: GUI Prototype of Concert Hall Booking System

5 System Architecture

5.1 Model-View-Controller (MVC) Pattern

The MVC pattern separates an application into three main components [1]:

1. Models handle the data representation and act as an interface for persistent storage.
2. Views that represent the user interface and what users see.
3. Controllers contain the programmed logic to change the view or update model data.

This separation of concerns facilitates maintenance and portability. It ensures that the core functionality and the application domain knowledge's internal representation exist independently of the user interface/s. In Django [7], the MVC pattern is implemented in a Model-Template-View+Controller form. The controller is the framework itself, as it sends a request to the appropriate view, according to the Django URL configuration. In Django, the view describes which data you see, not how you see it. The user presentation is delegated to the template.

For our application, the model will have the persistence of objects, mainly Customers, ShowProducers, Admins, Shows, Tickets, Seats, Memberships, etc. Django provides an inbuilt ORM to handle the associations amongst these models. The views will manage the application's response to the user's actions, such as raising a request for a hall. It will interact with Shows, Customers, and Membership models to handle ticket booking requests. When we extend our application beyond a Postman UI interface, we can leverage templates to present the data to users using HTML templates. This project structure enforces clarity in the development of different components.

Despite being a monolithic structure, Django offers the flexibility to decompose the system into smaller, self-contained modules called apps that serve a specific purpose [11]. A Django project is a collection of settings, configurations and apps that work together. Some apps that the concert hall booking system can be split into are ShowManagerApp, ApprovalEngineApp, TicketManagerApp, MembershipApp, PaymentApp, NotificationApp, etc. Each app can be relatively independent in functionality but will communicate with others. Each app will have its own models.py, views.py, urls.py (for routing), and any other necessary files like signals.py for handling events.

5.2 Event-Driven Architecture (EDA) Pattern

An event-driven system is designed to communicate and process updates to the application state amongst decoupled services using events. Typically, a producer emits an event when it encounters a state change and wants to propagate it to other services. The consumer services affected by that state change subscribe to these events and process the state change asynchronously to achieve eventual consistency.

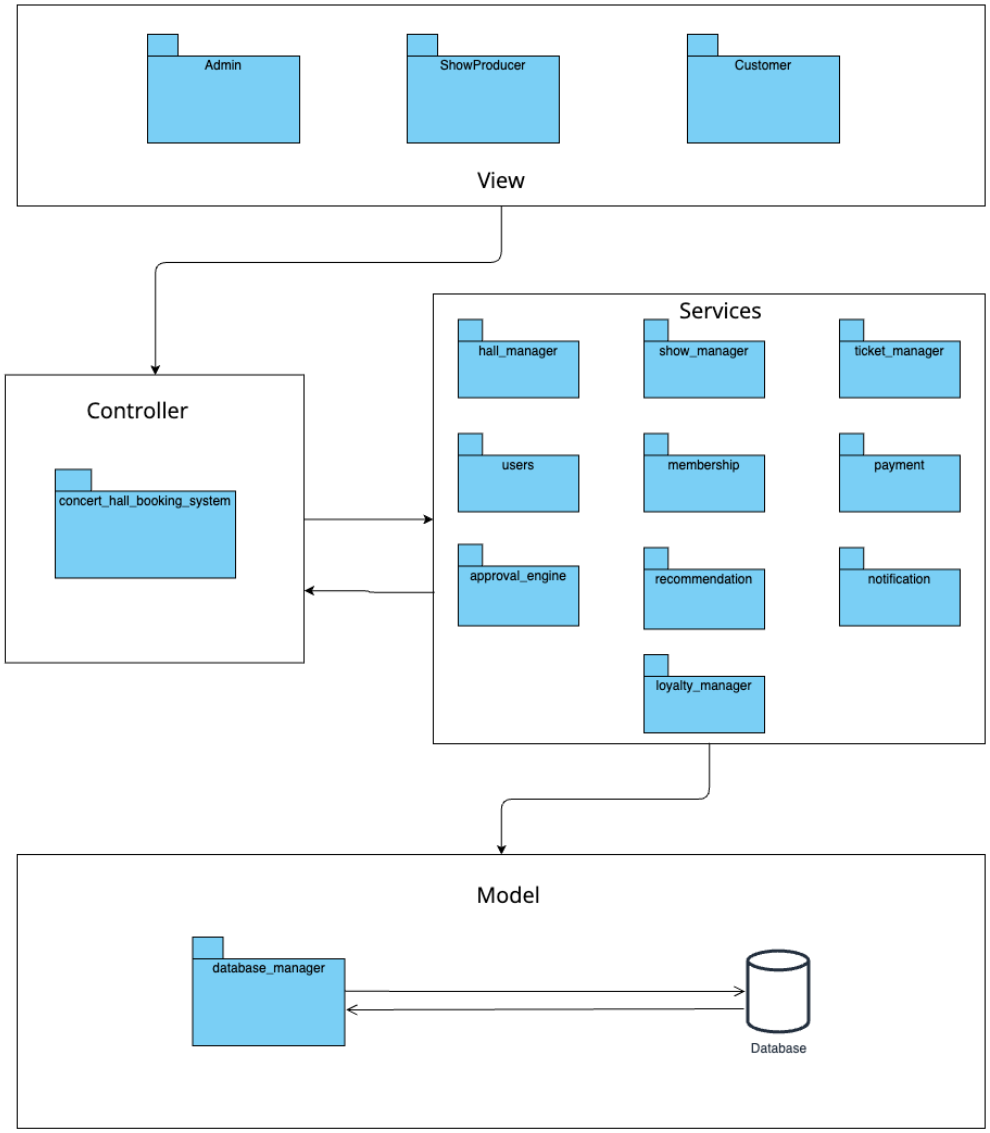


Figure 4: High-level package diagram

Django is traditionally used for monolith applications; however, some of our use cases like approving requests, processing refunds and sending out notifications bring up the need for different components that can communicate updates asynchronously. A fully microservices-based architecture, on the other hand, maybe too complex for our current needs. Using EDA within Django offers a middle ground; we can maintain the simplicity and development speed of Django's monolithic structure as well as handle certain workflows asynchronously, improving flexibility, performance, and fault tolerance without the overhead of full microservices.

The main advantage of EDA is decoupling and eliminating any direct dependencies between separate system components. When an admin cancels a show, he should not wait for refund initiations and notifications to be sent to register

that his show cancellation action has been processed successfully. The payment and notification modules responsible for these side-effects of show cancellation can instead be triggered by an event which contains the show ID and they can process it asynchronously and independently.

Events act as an abstraction or an interface between the producer and the consumer. When the show producer raises a request for a hall, that service need not know how this request will be handled. This helps to scale and update each part independently providing evolution support. In the request approval workflow, we can swap out a rules-based approval engine for a smarter AI agent without impacting the rest of the system. Another example would be the use case of booking a ticket; we can swap out the email notification service with an SMS notification service without much code change.

Other pros of EDA include system fault tolerance and flexibility. In case the notification service has some faults, the pending events can be processed once the system is restored thus improving system resilience. It provides an easier path if we want to migrate to a microservices-based architecture in the future. Although it introduces complexities in debugging, popular monitoring tools help to manage asynchronous workflows.

Using Django's MVT architecture ensures the system is maintainable and structured, while EDA provides scalability and flexibility for event-driven operations like approvals, refunds, and notifications. Together, they offer a balanced architecture for our concert hall booking system.

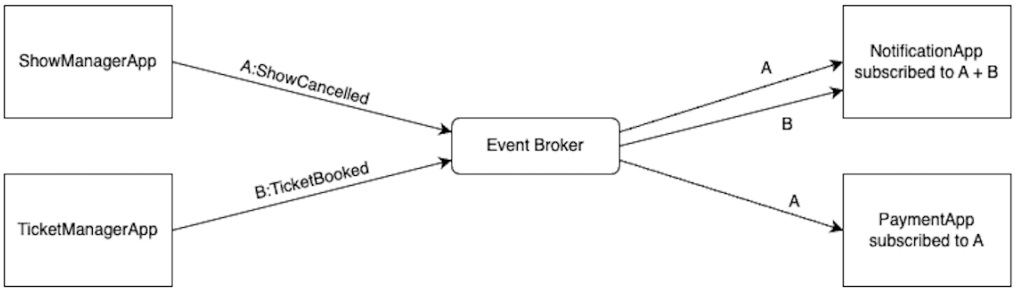


Figure 5: A use-case of EDA

5.3 Technology Pipeline

1. Django:

   We're using Django for our concert hall booking system to organise the application into modular components. This separation promotes easier maintenance, as each app can be developed and updated independently while remaining fully integrated. Django's URL routing simplifies directing user requests to appropriate functions, and its template system provides a flexible way to present data to users.

2. MySQL:

   MySQL follows the ACID principles which makes it suitable for features like buying a ticket as well as requesting a show. It makes sense to relate entities such as customers, shows, tickets, memberships, etc. and therefore, data organization and its retrieval are easier. MySQL can be integrated easily with the database layer of Django.

3. RabbitMQ:

   RabbitMQ is the message broker that facilitates the interaction between different parts of our application. The queues will be able to handle the tickets bookings, notifications etc. The most important feature of RabbitMQ is that no messages are lost when there are any interruptions. This plays well with our concert booking hall system as we deal with a lot of real-time data.

4. Jenkins:

   With Jenkins, we can automate the processes of building, testing, and deployment which means issues can be pinpointed and dealt with immediately after they are coded [10]. This is very important in a multi-application system such as ours where the change of one module will likely affect other modules.

5. PyTest:

   PyTest is straightforward to use and to maintain the test cases. It is especially helpful in our modular project structure, as we will be able to test every app in isolation and verify functionality [20]. Furthermore, automating test executions with Pytest can be integrated easily with Jenkins.

6. Docker:

   Docker is used to containerize our independant apps like ShowManager, TicketManager, Payment etc to make sure that all the dependencies, configurations are same through out. We will be able to withstand any issues that could arise in development and production environments. With docker we will also be able to deploy multiple instances based on the complexity of our booking system as these containers are lightweight and scalable. It also increases the portability and will be more easier to maintain. [6]

6 Conceptual Design

6.1 List of Candidate Objects

Objects are identified using Noun Identification Technique.

- Halls
- Venues
- Show Producer
- Requests
- Show
- Ticket
- Customer
- Booked Ticket
- Membership
- Loyalty points
- Admin
- Scheduled shows
- Seats
- Payment gateway
- Recommendation System
- Approver Engine
- Sales
- Refund

Heuristics Applied

- **Loyalty points** – attribute
- **Show, scheduled shows** – redundant
- **Requests** – same as show with only different status
- **Ticket, booked ticket** – redundant
- **Sales** – part of booked ticket data
- **Refund** – event

Classes Decided

- Hall
- Venue
- Show Producer
- Customer
- Ticket
- Membership
- Admin
- Shows
- Seats
- Payment gateway
- Recommendation System
- Approval Engine

6.2 Analysis Class Diagram

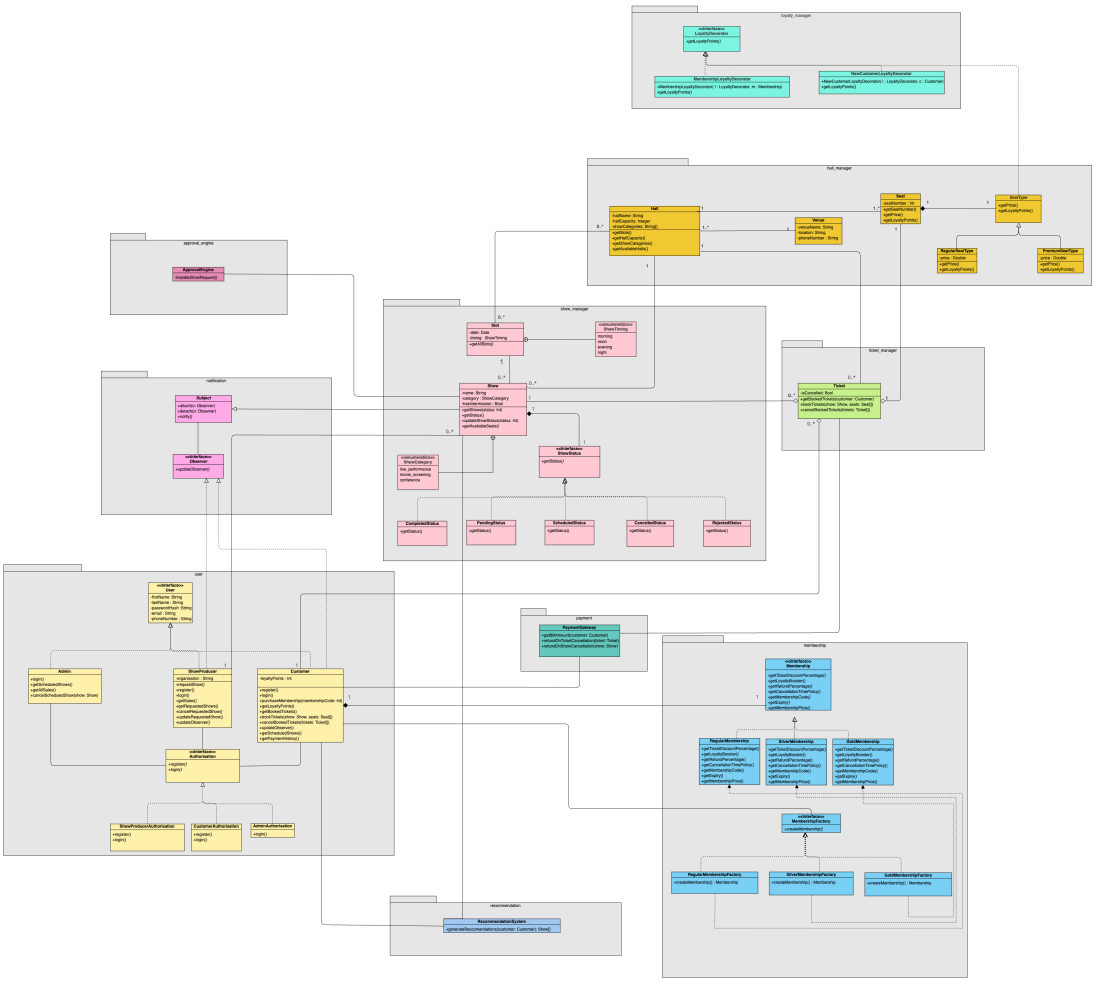


Figure 6: Analysis Class Diagram

6.3 Sequence Diagram

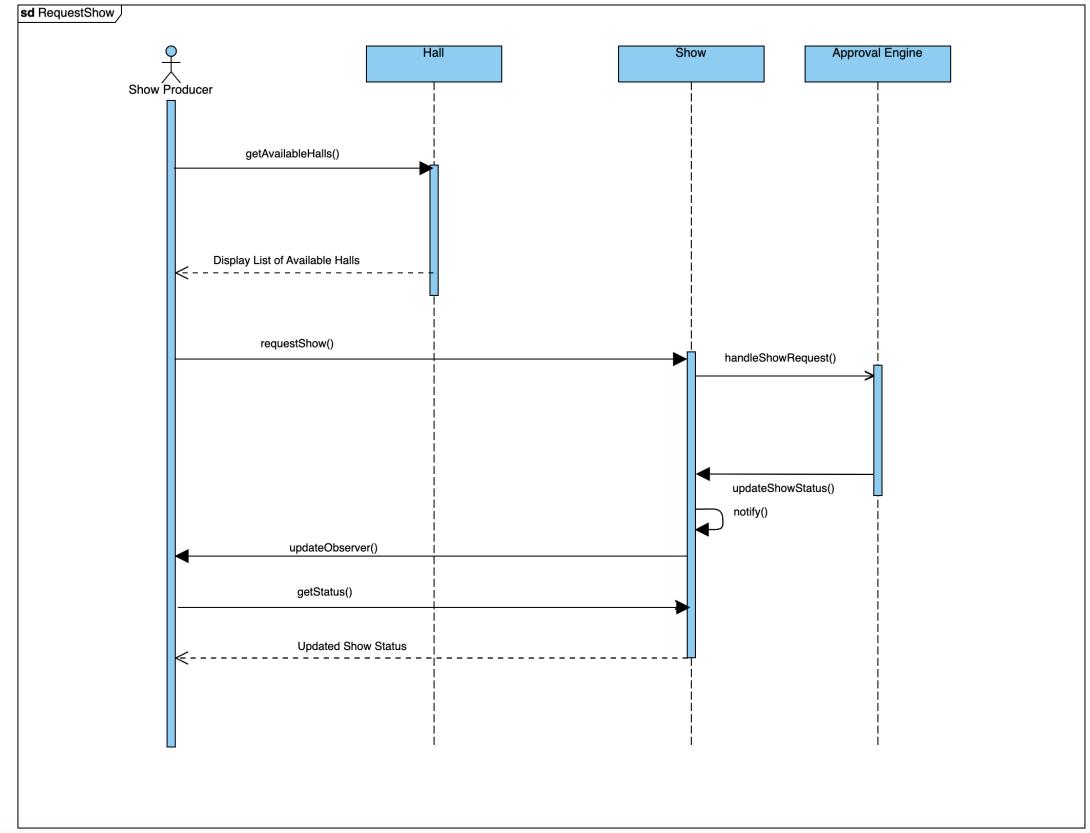


Figure 7: Sequence Diagram for Request Show

6.4 State Chart

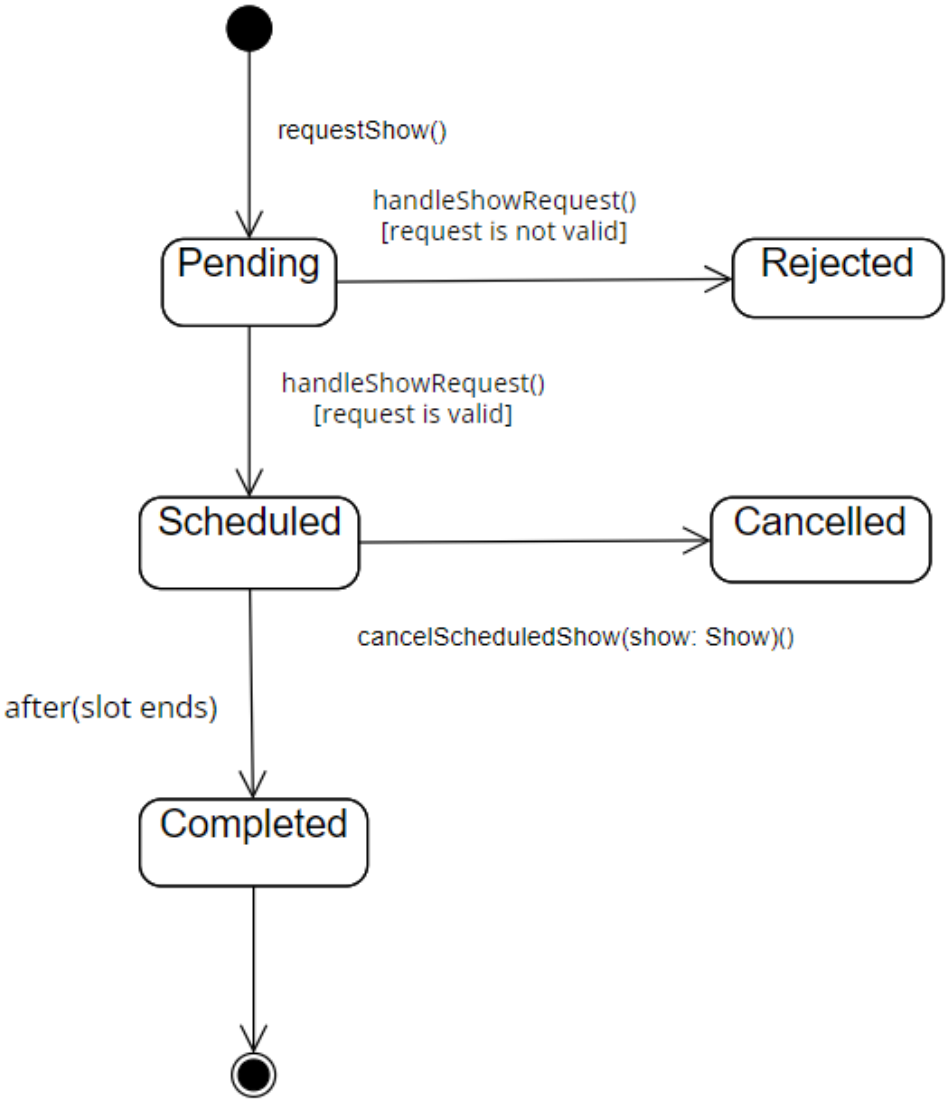


Figure 8: State chart: show

6.5 Entity Relationship (ER) Diagram

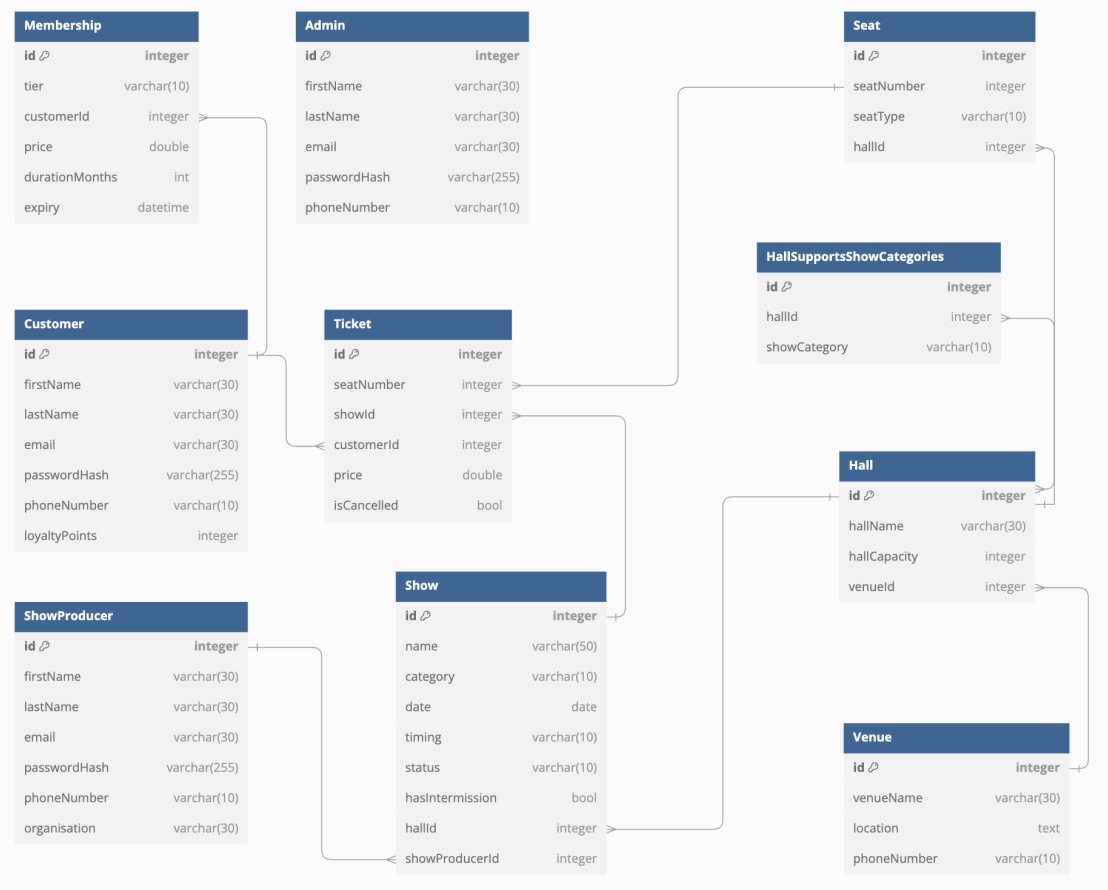


Figure 9: ER Diagram

7 Transparency and Traceability

Total number of packages	12
Total number of classes	97
Total lines of code	5787

Table 6: Project overview

Package	**Files**	**Author(s)**	**LOC**
approval_engine	engine_cor.py, tasks.py, tests.py	Anushree, Mahika	180
hall_manager	models.py, seattypes.py, serializers.py, views.py	Adarsh, Mahika	392
membership	factories.py, memberships.py, serializers.py, services.py, views.py	Adarsh, Anushree	430
notifications	models.py, tests.py, views.py	Adarsh, Mahika	123
payment_gateway	facade.py, models.py, services.py, views.py	Anushree	70
periodic_task_initializer	init_periodic_tasks .py	Mahika	30
shared	interfaces.py	Mahika, Anjana	32
show_manager	models.py, serializers.py, services.py, showstatuses.py, tasks.py, tests.py, views.py	Mahika, Adarsh	642
users	models.py, serializers.py, tests.py, views.py, middleware.py	Anjana, Adarsh, Mahika	486
ticket_manager	command.py, template.py, services.py, tests.py, serializers.py, views.py	Anushree, Anjana, Adarsh	860
loyalty_manager	decorators.py	Adarsh	39
recommendation_engine	recommendation_strategy.py, views.py, tests.py	Adarsh	203

Table 7: Package-wise code contribution

Team member	LOC
Adarsh Ajit (24026492)	1471
Anjana Thalappilly Anildas (24042714)	1746
Anushree Vinod (24158178)	1431
Mahika Om Jaguste (24016454)	1229

Table 8: Member-wise code contribution

Weekly Contributions, Issues, and Plans

Week	Contributions	Issues Arising	Plan for Next Week
Week 4 (30th Sept - 6th Oct)			
Adarsh	Discussed business scenario, Collaborated on use-case diagram.	Incorporating sufficient business logic.	Start state diagram and quality attributes
Anjana	Discussed business scenario, Collaborated on use-case diagram.	Incorporating sufficient business logic.	
Anushree	Discussed business scenario, worked on use-case diagram.	Incorporating sufficient business logic.	Start working on use-case descriptions.
Mahika	Discussed business scenario, Collaborated on use-case diagram.	Incorporating substantiality in business scenario.	Start class diagram and quality attributes.
Week 5 (7th Oct - 13th Oct)			
Adarsh	Collaborated on state diagram, wrote NFRs, read about quality attributes.		
Anjana	Collaborated on use-case description and finalizing business scenarios.		Work on class diagram and incorporate design patterns.
Anushree	Completed first draft of use-case diagram and descriptions.		Include NFR in use-case descriptions and start report.
Mahika	Collaborated on class diagram, wrote NFRs, read about quality attributes.	Not able to visualize all aspects of the class diagram.	Think about high-level system architecture.
Week 6 (14th Oct - 20th Oct)			

Week	Contributions	Issues Arising	Plan for Next Week
Adarsh	Finalized system architecture tools and pipeline, Read up on MVC and Design patterns.		Complete state diagram and finish the report
Anjana	Look into design patterns that can be included and work on class diagram.		Complete class diagram.
Anushree	Finalized use-cases, completed use-case diagram, and description; Started on the report.		Complete state diagram and finish the report.
Mahika	Read on architectural patterns (Event Driven), Finalized system architecture tools and pipeline.	Need to follow blogs to implement the pipeline.	Finalize report sections 1-6.
Week 7 (21st Oct - 27th Oct)			
Adarsh	Sequence Diagram, Working on report sections.		Follow Django Tutorials and Read Documentation.
Anjana	Class Diagram,Package diagram. Working on report sections 1-6.		Learn Django.
Anushree	State Diagram, Working on report sections 1-6.		Follow Django Tutorials and do initial setup.
Mahika	ER diagram, Working on report sections 1-6.		Follow Django tutorials to set up a repository.
Week 8 (28th Oct - 3rd Nov)			
Adarsh	Learning Django + initial setup + Branching Strategies + Documentation.		Implement Hall Manager APIs, Sonarqube integration + Docker.
Anjana	Learning Django + initial setup.		Implement authentication flow.
Anushree	Learning Django + initial setup.		Implement use-case - book ticket.

Week	Contributions	Issues Arising	Plan for Next Week
Mahika	Learning Django + boilerplate setup + Docker setup.	Learning curve for Django, MySQL, Docker.	Implement 1 use-case - raise request.
Week 9 (4th Nov - 10th Nov)			
Adarsh	Hall Manager APIs, Sonarqube integration + Docker.		Implement loyalty point calculation
Anjana	Basic authentication for admin, show producer and customer, middleware to fetch current user.	Setting up middleware, Auth using Django REST framework.	Implement command pattern.
Anushree	Ticket manager - Book ticket use-case, Factory method for membership types.		Implement Facade and CoR patterns.
Mahika	Show manager - Raise request use-case + Celery, Notifications - Observer, Show status - State.	Need to learn RabbitMQ to understand retry and DLQ.	Implement state design pattern for seats.
Week 10 (11th Nov - 17th Nov)			
Adarsh	Loyalty Points - Decorator Pattern, Recommendation Engine - Strategy Pattern.		Implement Admin cancel show use case
Anjana	Command pattern for ticket cancellation.		Implement view booked tickets and sales use cases.
Anushree	Complete Factory Method, Facade Pattern for payment gateway, CoR for approval engine.		Implement ticket and membership history.
Mahika	Seat type - State pattern, Show completion cron job - Celery Beat, Hall manager APIs.	Challenges to set up Celery Beat.	More extension use-cases, Retry mechanism.
Week 11 (18th Nov - 24th Nov)			

Week	Contributions	Issues Arising	Plan for Next Week
Adarsh	Admin cancels a show, Notifications - Observer Pattern, Hall Manager extension use-cases, Show manager extension use-cases.		Resolve code smells identified by SonarQube
Anjana	Template and strategy pattern for booked ticket view and ticket sales.		Write tests and include logging.
Anushree	Add views for booking and purchase history, Integrating Swagger for API documentation.		Write automation test and integrate Prometheus.
Mahika	Celery retry mechanism + RabbitMQ DLQ, Show manager extension use-cases.		Testing.
Week 12 (25th Nov - 5th Dec)			
Adarsh	Tests, Integrate Sonarqube with Jenkins pipeline, Resolve code smells detected by Sonarqube, Class Diagram.		Finish final report.
Anjana	Logging, Load balancing and tests, Component Diagram, Package Diagram.	Setting up load balancer.	Finish final report.
Anushree	Tests, Prometheus and Grafana for monitoring, Deployment Diagram, State Diagram.	Faced issues in integrating Prometheus to Django project.	Finish final report.
Mahika	Tests, Docker, Design-time sequence diagram.	Took time to figure out mocks and fixtures in pytest.	Finish final report.

8 Implementation

8.1 Architectural Patterns

8.1.1 Model View Controller (MVC)

The MVC pattern separates an application into three main components:

i. Models handle the data representation and act as an interface for persistent storage.

ii. Views that represent the user interface and what users see.

iii. Controllers contain the programmed logic to change the view or update model data.

This separation of concerns facilitates maintenance and portability. It ensures that the core functionality and the application domain knowledge's internal representation exist independently of the user interface/s. In Django, the MVC pattern is implemented in a Model-Template-View+Controller form. The controller is the framework itself, as it sends a request to the appropriate view, according to the Django URL configuration. In Django, the view describes which data you see, not how you see it. The user presentation is delegated to the template.

For our application, the model will have the persistence of objects, mainly Customers, ShowProducers, Admins, Shows, Tickets, Seats, Memberships, etc. Django provides an inbuilt ORM to handle the associations amongst these models. The views will manage the application's response to the user's actions, such as raising a request for a hall. It will interact with Shows, Customers, and Membership models to handle ticket booking requests. When we extend our application beyond a Postman UI interface, we can leverage templates to present the data to users using HTML templates. This project structure enforces clarity in the development of different components.

Despite being a monolithic structure, Django offers the flexibility to decompose the system into smaller, self-contained modules called apps that serve a specific purpose. A Django project is a collection of settings, configurations and apps that work together. Some apps that the concert hall booking system can be split into ShowManagerApp, ApprovalEngineApp, TicketManagerApp, MembershipApp, PaymentApp, NotificationApp, etc. Each app can be relatively independent in functionality but will communicate with others. Each app will have its own models.py, views.py, urls.py (for routing), and any other necessary files like signals.py for handling event.

8.1.2 Event Driven Architecture (EDA)

In our implementation, we have utilised an event-driven architecture to handle key workflows asynchronously. For the use case of a show producer raising a show request, the approval or rejection of the request is performed asynchronously and the producer is sent a notification once his pending request is processed. The process begins when a show producer calls the raise show request API with details such as show name, category, desired hall and slot. This creates a Show entry in the Pending status and the show producer receives an instant response that his request has been received. Simultaneously, an event is triggered by sending a task to a Celery worker as shown in the second-last line in (Figure 10). The event for this use case is represented by a task named handle_show_request (Figure 11), which includes the show_id. Celery, with RabbitMQ as the message broker, handles the task queue and processes them asynchronously. For this use case, it makes the approval engine perform checks like the hall supports that show category, the hall supports that slot and there is no overlapping show at that hall and slot (Figure 12). The show status is updated and a notification is triggered to inform the show producer of the outcome of his request. We can monitor the tasks using Flower, an open-source web-application that shows real-time information about

Celery workers and tasks (Figure 13).

A significant benefit of using EDA for this use case is to handle when multiple requests for the same hall and slot come at the same time. The asynchronous flow will approve the request which arrives first and reject the later request, thus ensuring data integrity (Figure 14). Another advantage is the decoupling of the show producer raising the request and the approval engine; in future, we can swap in a more sophisticated approval engine without affecting any other component. In case of any faults in the approval engine, the tasks will persist in the message broker until the approval engine is up to consume the tasks. Such as architecture allows for scalability as more Celery workers can handle tasks in parallel when traffic increases.

```python
class ShowRequestService:
    def request_show(
        show_id: int,
        show_producer: ShowProducer,
        name: str,
        category: Category,
        has_intermission: bool,
        slot: Slot,
        hall: Hall
    ):
        """
        Creates a show request and triggers the asynchronous processing task.
        """
        try:
            if not show_id:
                show = Show.objects.create(
                    name=name,
                    category=category,
                    has_intermission=has_intermission,
                    slot=slot,
                    hall=hall,
                )
                show.attach(observer=show_producer, interest=0)
            else:
                Show.objects.filter(id=show_id).update(
                    name=name,
                    category=category,
                    has_intermission=has_intermission,
                    slot=slot,
                    hall=hall,
                )
                show = Show.objects.filter(id=show_id).first()

        except Exception as e:
            print(e)

        if not show_id:
            # synchronous: ApprovalEngine(show).handle_show_request()
            # asynchronous request to approval engine
            handle_show_request.delay(show.id)

        return show
```

Figure 10: Sending event for asynchronous processing

```python
from celery import shared_task
from show_manager.models import Show
from .engine_cor import ApprovalEngine
from config.celery import BaseTaskWithRetry

@shared_task(base=BaseTaskWithRetry, acks_late=True)
def handle_show_request(show_id):
    try:
        show = Show.objects.get(id=show_id)
    except Show.DoesNotExist:
        print(f"Show with ID {show_id} does not exist.")
        return

    approval_engine = ApprovalEngine(show)
    approval_engine.handle_show_request()      You, 7 mi
```

Figure 11: Celery shared task

8.2 Design Principles - SOLID

The Concert Hall Booking System adheres to SOLID principles making it robust, scalable, and easy-to-maintain design. Each principle plays a distinct role in keeping the system flexible, reliable, and well-organized.

The **Single Responsibility Principle (SRP)** states that a class or module should have only one main job. This is evident in how the system separates the responsibilities of show producers, customers, and admins. Each role focuses on its own tasks: show producers manage show requests, customers handle bookings and loyalty points, and admins oversee platform operations, such as approving or canceling shows. By keeping these responsibilities distinct, the system is easier to debug and extend. The project has been configured to have multiple packages for handling different responsibilities such as ticket_manager, hall_manager etc. Design patterns like the Command Pattern for ticket cancellations and the Facade Pattern for payment gateway interactions ensure that modules are focused and maintainable.

The **Open/Closed Principle (OCP)** means that software entities should be open to extension but closed to modification. This principle ensures the system can grow without changing its core. For example, the Factory Method for membership creation allows new tiers to be introduced without altering existing code. Similarly, the Decorator Pattern applied to loyalty points makes it easy to add new reward structures, like seasonal bonuses, without modifying the base logic. This approach keeps the system adaptable to future needs.

```python
class ApprovalEngine:
    """
    Engine for approving or rejecting show requests with CoR.
    """
    def __init__(self, show):
        self.show: Show = show
        # Assemble the chain of responsibility
        self.validation_chain = (
            OverlappingShowValidationHandler(
            CategorySupportValidationHandler(
            SlotSupportValidationHandler()
            ))
        )

    def handle_show_request(self):
        """
        Handles the show request by validating it through the chain.
        """
        try:
            if not isinstance(self.show.get_status_instance(), PendingStatus):
                print("Request not in pending status")
                return

            is_valid, message = self.validation_chain.handle(self.show)

            if is_valid:
                self.show.schedule()
            else:
                self.show.reject(message)
        except Exception as e:
            print(f"Error processing show request: {e}")
            self.show.reject("Error in validating request")
```

Figure 12: Approval engine performs validations

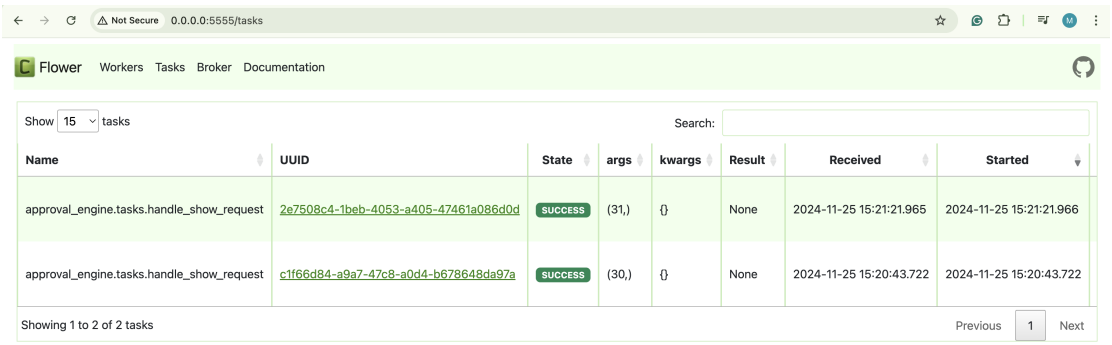


Figure 13: Flower dashboard to monitor Celery tasks and workers

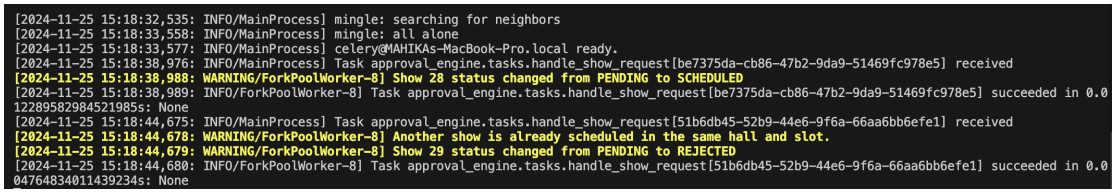


Figure 14: Celery worker consuming tasks from task queue

The **Liskov Substitution Principle (LSP)** ensures that objects of a superclass can be replaced with objects of a subclass without altering the correctness of the program. This is achieved in the system through components like seat types (Regular and Premium) implemented using the State Pattern. These allow for different pricing and reward systems without disrupting the booking process. Similarly, membership tiers (Silver and Gold) are designed using abstractions, enabling seamless upgrades or downgrades without impacting other parts of the system.

The **Interface Segregation Principle (ISP)** suggests that no client should be forced to depend on methods it does not use. This is exemplified by the system's role-based views and APIs. Each user—whether a show producer, customer, or admin—interacts only with the functionalities relevant to them. For instance, producers manage show requests and track ticket sales, customers handle bookings and loyalty points, and admins oversee approvals and cancellations. By segregating these interfaces, the system remains intuitive and avoids overloading users with irrelevant features.

The **Dependency Inversion Principle (DIP)** states that high-level modules should depend on abstractions, not concrete implementations. This principle is reflected in components like the approval engine, which uses the Chain of Responsibility Pattern to manage validations and decisions. This abstraction makes it easy to modify or extend the approval process without disrupting the rest of the system. Similarly, the payment gateway uses the Facade Pattern to hide complex integrations, allowing the system to switch payment providers without impacting the core functionality.

By incorporating these principles into its design and leveraging patterns like Factory, Decorator, and State, the Concert Hall Booking System ensures modularity, maintainability, and flexibility. Each component works independently yet integrates seamlessly, creating a platform that is ready to adapt and evolve with changing requirements.

8.3 Design Patterns

8.3.1 State Pattern

The State design pattern is a behavioural pattern that allows an objects to change its behaviour when its internal state changes. This is done by encapsulating that changing behaviour within instances of different states and switching between

these instances. For our Show object, it can have the 5 statuses - Pending, Scheduled, Rejected, Cancelled and Completed. According to the state chart, the show object changes its status throughout its lifetime. Hence it holds a ShowStatus instance via composition which can be changed when the show needs to update its status. (Figure 15) This aligns with the principle of 'Replace Conditional Logic with Polymorphism' as otherwise there would be the need for branching logic to handle the show status.

Another application of the state pattern was in defining seats and their types. We followed the refactoring principle 'Replace Type Code with State' to make the seat hold a SeatType instance via composition. The concrete SeatType, for now, RegularSeatType and PremiumSeatTypes, hold behaviour for the price of the seat and loyalty points gained from booking that seat (Figure 16). This code is easily extensible to add new seat types as well as upgrade or downgrade existing seats to other types.

The memberships feature also provides an opportunity to apply the state pattern where a customer can hold a Regular, Silver or Gold membership. The perks and concessions of each membership tier, primarily, the discount on tickets, loyalty booster, refund percentage on cancellation, etc. can be encapsulated within the concrete Membership classes (Figure 17). Also, customers can change their membership status easily during their lifetime.

8.3.2 Factory Design Pattern

The Factory Method Pattern is utilized in the concert hall booking system to manage the creation of different types of memberships dynamically and efficiently. This design pattern provides a way to encapsulate the instantiation logic for various membership classes, making the system more modular and easier to extend. By defining a common interface for creating memberships, the system ensures flexibility and consistency across all membership types.

At the core of this implementation is the abstract factory class (MembershipFactory), which declares a method create_membership to be implemented by its subclasses. This abstract definition allows the system to delegate the creation of specific memberships to specialized factory classes without needing to know the details of how those memberships are instantiated.

The system includes concrete factory classes such as RegularMembershipFactory, SilverMembershipFactory, and GoldMembershipFactory. Each of these factories is responsible for creating a specific membership type. For instance, the RegularMembershipFactory produces a RegularMembership, while the GoldMembershipFactory creates a GoldMembership. By isolating the instantiation logic within these factories, the system achieves a high degree of separation of concerns, making it easier to manage and extend.

The memberships themselves are defined through an abstract product class (Membership), which specifies the common functionalities that all memberships must

```python
class SeatType(ABC):
    """
    Abstract base class for seat types. Enforces the implementation of
    the following methods in subclasses.
    """

    @abstractmethod
    def get_seat_type(self):
        pass

    @abstractmethod
    def get_price(self):
        pass


Mahika Jaguste, 7 days ago | 1 author (Mahika Jaguste)
class RegularSeatType(SeatType):

    def get_seat_type(self):
        return SeatTypeEnum.REGULAR

    def get_price(self):
        return 10


Mahika Jaguste, 7 days ago | 1 author (Mahika Jaguste)
class PremiumSeatType(SeatType):

    def get_seat_type(self):
        return SeatTypeEnum.PREMIUM

    def get_price(self):
        return 20
```

Figure 15: Seat types

```python
class ShowStatus(ABC):
    """
    Abstract base class for show statuses. Enforces the implementation of
    the `get_status` in subclasses.
    """
    def __init__(self, show):
        self.show = show

    @abstractmethod
    def get_status(self):
        """Handle the current state logic."""
        pass


You, 1 hour ago | 2 authors (Mahika Jaguste and one other)
class PendingStatus(ShowStatus):
    def get_status(self):
        return ShowStatusEnum.PENDING

    def transition_to_scheduled(self):
        """Transition the status from 'Pending' to 'Scheduled'."""
        self.show.status = ShowStatusEnum.SCHEDULED.name
        self.show.save()
        print(f"Show {self.show.id} status changed from {ShowStatusEnum.PENDING.name} to {ShowStatusEnum.SCHEDULED.name}")

    def transition_to_rejected(self):
        """Transition the status from 'Pending' to 'Rejected'."""
        self.show.status = ShowStatusEnum.REJECTED.name
        self.show.save()
        print(f"Show {self.show.id} status changed from {ShowStatusEnum.PENDING.name} to {ShowStatusEnum.REJECTED.name}")

    def transition_to_cancelled(self):
        """Transition the status from 'Pending' to 'Cancelled'."""
        self.show.status = ShowStatusEnum.CANCELLED.name
        self.show.save()
        print(f"Show {self.show.id} status changed from {ShowStatusEnum.PENDING.name} to {ShowStatusEnum.REJECTED.name}")



Mahika Jaguste, 2 weeks ago | 1 author (Mahika Jaguste)
class ScheduledStatus(ShowStatus):
    def get_status(self):
        return ShowStatusEnum.SCHEDULED

Mahika Jaguste, 2 weeks ago | 1 author (Mahika Jaguste)
class CompletedStatus(ShowStatus):
   def get_status(self):
        return ShowStatusEnum.COMPLETED

Mahika Jaguste, 2 weeks ago | 1 author (Mahika Jaguste)
class RejectedStatus(ShowStatus):
    def get_status(self):
        return ShowStatusEnum.REJECTED

Mahika Jaguste, 2 weeks ago | 1 author (Mahika Jaguste)
class CancelledStatus(ShowStatus):
    def get_status(self):
        return ShowStatusEnum.CANCELLED
```

Figure 16: Show statuses

```python
# Defining Abstarct Product
Anushree Vinod, 5 days ago | 1 author (Anushree Vinod)
class Membership(ABC):

    @abstractmethod
    def get_membership_code(self):
        pass

    @abstractmethod
    def get_membership_price(self):
        pass

    @abstractmethod
    def get_ticket_discount_percentage(self):
        pass

    @abstractmethod
    def get_loyalty_booster(self):
        pass

    @abstractmethod
    def get_refund_percentage(self):
        pass

    @abstractmethod
    def get_cancellation_time_policy(self):
        pass

    @abstractmethod
    def get_expiry(self, membership_period):
        pass

# Defining Concrete product

Anushree Vinod, 4 days ago | 1 author (Anushree Vinod)
class RegularMembership(Membership):
    def get_membership_code(self):
        return MembershipTypeEnum.REGULAR.name

    def get_membership_price(self):
        return 0

    def get_ticket_discount_percentage(self):
        return 0

    def get_loyalty_booster(self):
        return 1.0  # No boost for regular

    def get_refund_percentage(self):
        return 50

    def get_cancellation_time_policy(self):
        return 24

    def get_expiry(self, membership_period):
        return datetime.now(timezone.utc) + relativedelta(months=membership_period)
```

Figure 17: Membership tiers

implement. These include methods like get_ticket_discount_percentage, get_loyalty_booster, and get_expiry. The concrete product classes, such as RegularMembership, SilverMembership, and GoldMembership, implement this interface to provide specific behavior for each membership type. For example, the GoldMembership class might offer the highest ticket discount and a zero-hour cancellation policy, while RegularMembership might not provide any discount.

When a customer requests to purchase a membership, the booking system uses the factory method to dynamically determine which membership factory to use based on the requested type. For example, if the customer selects a "SILVER" membership, the system retrieves the SilverMembershipFactory using a utility function like get_membership_factory. The factory's create_membership method then generates an instance of SilverMembership, which can be used by the system to apply discounts, calculate refunds, or determine the membership's expiry.

This approach offers several advantages. First, it makes the system highly scalable and flexible. Adding a new membership type only requires creating a new ConcreteMembership class and a corresponding factory without modifying the existing code. Second, it promotes separation of concerns by keeping the membership creation logic isolated from the rest of the booking system. Finally, it ensures encapsulation, as the system only interacts with the MembershipFactory interface, making the codebase easier to maintain and extend.

8.3.3 Facade Pattern

The Facade Pattern is utilized in the payment functionality to simplify and unify interactions with the underlying billing and refund services. The PaymentGatewayFacade acts as a single access point, consolidating complex operations like calculating ticket bills, processing refunds, and handling membership payments into straightforward methods. This abstraction allows external systems, such as booking modules, to use payment services without delving into their internal details.

Behind the facade, the BillService handles tasks such as retrieving seat prices, applying membership-specific discounts, and calculating total amounts. Similarly, the RefundService manages refund calculations by determining refund eligibility, percentages, and amounts based on the customer's membership. These operations are encapsulated within the facade, shielding clients from the intricate logic and ensuring a seamless workflow. This also allows a seamless integration of a third party payment system in the future if required.

This design reduces coupling between external systems and the payment subsystem, making the application more modular and easier to maintain. Clients interact with a clean, consistent API, while changes to the underlying services remain isolated from the facade users. When a third-party payment gateway is introduced in the future, the Facade Pattern allows easy integration by encapsulating the API logic within the facade without altering the facade users such as ticket_manager. The facade can route payment operations to the new API while

```
#define factory
You, 3 weeks ago | 1 author (You)
class MembershipFactory(ABC):
    @abstractmethod
    def create_membership(self):
        pass
    You, 3 weeks ago • added membership factories a
#define Concrete factory

You, 3 weeks ago | 1 author (You)
class RegularMembershipFactory(MembershipFactory):
    def create_membership(self):
        return RegularMembership()


You, 3 weeks ago | 1 author (You)
class SilverMembershipFactory(MembershipFactory):
    def create_membership(self):
        return SilverMembership()


You, 3 weeks ago | 1 author (You)
class GoldMembershipFactory(MembershipFactory):
    def create_membership(self):
        return GoldMembership()
```

Figure 18: Abstract and concrete factories

```python
from payment_gateway.services import BillService, RefundService

You, last week | 2 authors (Adarsh Ajit and one other)
class PaymentGatewayFacade:
    def __init__(self):
        self.bill_service = BillService()
        self.refund_service = RefundService()

    def get_ticket_bill_amount(self, customer_obj, seat_objs):
        price_per_ticket, bill_amount =  self.bill_service.get_ticket_bill_amount(customer_obj, seat_objs)
        return price_per_ticket, bill_amount

    def get_membership_bill_amount(self, membership_price):
        return self.bill_service.get_membership_bill_amount(membership_price)

    def get_ticket_refund(self, ticket_ids):      Adarsh Ajit, last week • Revert "Revert "Create payment
        return self.refund_service.get_ticket_refund(ticket_ids)

    def get_show_refund(self, show_id):
        return self.refund_service.get_show_refund(show_id)
```

Figure 19: Payment Gateway Facade

```python
class BillService:

    def get_ticket_bill_amount(self, customer_obj, seat_objs):
        try:
            prices = {seat.id : seat.get_seat_price() for seat in seat_objs}
            customer_membership_instance = (
                CustomerMembership.get_latest_valid_membership_instance(customer_obj)
                )

            discount_percentage = customer_membership_instance.get_ticket_discount_percentage()
            discounted_prices = {k: v * (1 - discount_percentage / 100) for k, v in prices.items()}
            return discounted_prices, sum(discounted_prices.values())
        except Exception as exc:
            return False, False

    def get_membership_bill_amount(self, membership_price):
        return membership_price


You, last week | 2 authors (Adarsh Ajit and one other)
class RefundService:      Adarsh Ajit, last week • Revert "Revert "Create payment gateway app""

    def get_ticket_refund(self, ticket_ids):
        try:
            tickets = Ticket.objects.filter(id__in = ticket_ids)
            customer_membership_instance = (
                CustomerMembership.get_latest_valid_membership_instance(
                    tickets.first().customer)
                )
```

Figure 20: Payment Gateway Services

maintaining the same interface for the booking module, ensuring minimal code changes in existing workflows.

8.3.4 Chain of Responsibility Pattern

The Chain of Responsibility (CoR) Pattern is effectively utilized in the approval engine to handle the sequential validation of show requests in a flexible and decoupled manner. This design organizes validation checks into independent handlers, allowing each to focus on a specific aspect of the approval process, such as overlapping schedules, category compatibility, or slot availability.

In this implementation, each validation step is encapsulated in a distinct handler class, such as OverlappingShowValidationHandler, CategorySupportValida-

Figure 21: Handlers defined in approval engine

tionHandler, and SlotSupportValidationHandler. These handlers are linked together in a chain within the ApprovalEngine. When a show request is processed, it is passed through the chain, with each handler performing its validation and either passing the request to the next handler or terminating the process if the validation fails. For example, if the show conflicts with an already scheduled one, the OverlappingShowValidationHandler returns a failure response without proceeding further.

This pattern offers following advantages. Firstly, it provides a modular structure where each handler operates independently, making it easy to add or modify validation logic without affecting other parts of the chain. Secondly, it promotes reusability, as each handler can be employed in other approval workflows if needed. Finally, it ensures maintainability, as the responsibility for validation is distributed among well-defined components rather than centralized in a monolithic block of code.

The CoR Pattern also enhances scalability. If additional validations are required in the future, such as checks for availability of facilities or other validation. New handlers can be seamlessly integrated into the chain without altering existing logic.

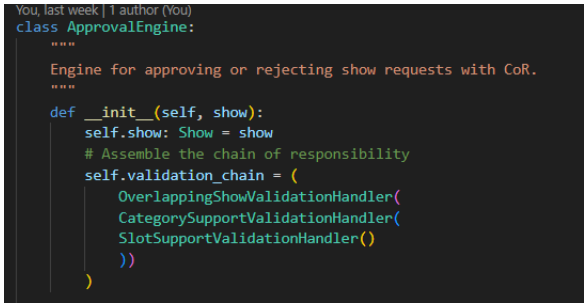


Figure 22: Defining the Chain

8.3.5 Strategy Pattern

The Strategy design pattern is a behavioral design pattern that enables selecting an algorithm's behavior at runtime. This is accomplished by defining a family of algorithms, encapsulating each one, and making them interchangeable. This eliminates the need for conditional logic to determine the appropriate algorithm, aligning with the principle of "Encapsulate What Varies".

In our concert hall booking system, we use this pattern to show the ticket sales based on the requesting user. Both admin and show producer have access to view the ticket sales which shows the tickets which are sold. While show producer only have access to sales of a particular slot of a show, admin have access to all slots of that show (Figure 23). Here we are following the principle of "Replace Type code with strategy".

In the code, we have an abstract class called TicketSalesStrategy that defines a common interface for all ticket sales view strategies. This interface includes an abstract method, fetch_ticket_sales, which all concrete strategies must implement. Two concrete classes, AdminTicketSalesStrategy and ShowProducerTicketSalesStrategy, implement this interface and provide their specific implementations of the fetch_ticket_sales method. The AdminTicketSalesStrategy fetches ticket sales for admins, while the ShowProducerTicketSalesStrategy ensures that the tickets belong to the show producer's shows and applies slot filtering as shown in Figure 25. The TicketSalesContext class as shown in Figure 24 is responsible for using a specific strategy to fetch ticket sales. It holds a reference to a TicketSalesStrategy object and delegates the fetch_ticket_sales call to the strategy object. When an instance of TicketSalesContext is created, it is initialized with a specific strategy, allowing for dynamic switching between different strategies at runtime based on the user role. Employing strategy pattern help in encapsulating different behaviour and provide flexibility in the implementation. The separation of concerns improves readability and makes the code easier to understand. Overall, the Strategy Pattern enhances the adaptability of our ticket sales view system, ensuring it can efficiently meet the diverse requirements of different user roles.

We apply the Strategy Pattern to also offer different options on how recommendations for shows are generated. So, for example, we can switch between different recommendation strategies easily. For example, there are two main strategies that we deploy: the Location Based Recommendation which recommends shows based on the location of the hall and the Trending Recommendation which focuses on

```python
@api_view(["POST"])
@permission_classes([IsAuthenticated])
def view_ticket_sales(request):
    user = get_current_user()
    serializer = TicketSalesRequestSerializer(data=request.data)
    if not serializer.is_valid():
        return JsonResponse({"error": serializer.errors}, status=400)

    data = serializer.validated_data
    show_name = data["show_name"]
    slot_id = data.get("slot_id", None)

    try:
        if user.is_superuser:
            strategy = AdminTicketSalesStrategy()
        elif hasattr(user, 'showproducer'):
            strategy = ShowProducerTicketSalesStrategy()
        else:
            return JsonResponse({"error": "Unauthorized access"}, status=403)

        context = TicketSalesContext(strategy)
        ticket_sales = context.fetch_sales(show_name, slot_id)
        serialized_sales = TicketSerializer(ticket_sales, many=True).data
        logger.info(f"{user} has viewed ticket sales")
        return JsonResponse(serialized_sales, safe=False)

    except Exception as e:
        return JsonResponse({"error": str(e)}, status=400)
```

Figure 23: View for ticket sales view

```python
class TicketSalesContext:
    def __init__(self, strategy):
        self.strategy = strategy

    def fetch_sales(self, show_name,slot_id):
        return self.strategy.fetch_ticket_sales(show_name,slot_id)
```

Figure 24: TicketSalesContext in ticket sales view strategy pattern

```python
class TicketSalesStrategy(ABC):

    @abstractmethod
    def fetch_ticket_sales(self, show_id, slot_id):
        pass


You, 2 days ago | 1 author (You)
class AdminTicketSalesStrategy(TicketSalesStrategy):

    def fetch_ticket_sales(self, show_name, slot_id=None):
        if slot_id:
            tickets = Ticket.objects.filter(
                show__name=show_name,
                isCancelled=False,
                show__slot_id=slot_id  )

        else:
            tickets = Ticket.objects.filter(
                show__name=show_name,
                isCancelled=False)
        return tickets


You, 2 days ago | 1 author (You)
class ShowProducerTicketSalesStrategy(TicketSalesStrategy):

    def fetch_ticket_sales(self, show_name,slot_id):
        user = get_current_user()
        user = ShowProducer.objects.get(user=user)
        tickets = Ticket.objects.filter(
                show__name=show_name,
                isCancelled=False,
                show__slot_id=slot_id,
                show__show_producer=user  )
        return tickets
```

Figure 25: TicketSalesStrategy in ticket sales view strategy pattern

the shows that are the most popular according to the ticket sales argument. The Location Based Recommendation strategy can be defined as one that recommends users shows that are relevant to them on the basis of the location of the venue. On the other hand, the Trending Recommendation strategy deals with recommending users the shows that have sold the most tickets meaning that it highlights the most popular shows.

Every strategy is provided in a distinct class that follows its interface so that the system will not be changed whenever new guidelines are added. The 'Global-RecommendationContext' class acts as a mediator and holds the active strategy and assigns the recommendation operations to that strategy.

```python
class GlobalRecommendationContext:
    """
    Context class that manages the global recommendation strategy.

    Allows setting and getting a strategy and retrieving recommendations based on the given strategy.
    """
    _strategy = None

    @classmethod
    def set_strategy(self, strategy):
        self._strategy = strategy

    @classmethod
    def get_strategy(self):
        if self._strategy is None:
            raise ValueError("Global recommendation strategy is not set.")
        return self._strategy

    @classmethod
    def get_recommendations(self):
        strategy = self.get_strategy()
        return strategy.recommend()


class RecommendationStrategy(ABC):
    """
    Abstract base class for recommendation strategies.
    """

    @abstractmethod
    def recommend(self, **kwargs):
        pass

class LocationBasedRecommendation(RecommendationStrategy):
    def recommend(self, **kwargs):
        """
        Fetch shows from the database based on the hall's location.
        """
        location = kwargs.get('location')
        try:
            halls_in_location = Hall.objects.filter(venue__location=location)

            shows = Show.objects.filter(hall__in=halls_in_location, status=ShowStatusEnum.SCHEDULED.value)
            serialized_shows = ShowSerializer(shows, many=True).data

            return list(serialized_shows)

        except Exception as e:
            print(f"Error fetching recommendations: {e}")
            return []


class TrendingRecommendation(RecommendationStrategy):
    def recommend(self, **kwargs):
        """
        Fetch shows from the database based on ticket sales.
        """
        try:
            tickets = Ticket.objects.filter(isCancelled=False, status=ShowStatusEnum.SCHEDULED.value).values('show') \
                .annotate(total_tickets_sold=Count('id')) \
                .order_by('-total_tickets_sold')[:5]

            ticket_ids = [ticket['show'] for ticket in tickets]
            trending_shows = Show.objects.filter(id__in=ticket_ids)

            serialized_shows = ShowSerializer(trending_shows, many=True).data

            return serialized_shows

        except Exception as e:
            print(f"Error fetching recommendations: {e}")
            return []
```

Figure 26: Context and abstract base class for recommendation strategies

```python
@api_view(['POST'])
@permission_classes([IsAuthenticated])
def set_recommendation_strategy(request):
    """
    API to manage and fetch the global recommendation strategy.
    """
    strategy_name = request.data.get("strategy")

    if not strategy_name:
        return Response({"error": "Strategy name is required."}, status=status.HTTP_400_BAD_REQUEST)

    strategy_class = STRATEGY_MAP.get(strategy_name)

    if not strategy_class:
        return Response({"error": (
                    f"Invalid strategy name. Valid options are: "
                    f"{', '.join(STRATEGY_MAP.keys())}."
                )
            },
            status=status.HTTP_400_BAD_REQUEST,
        )

    # Set the global strategy
    GlobalRecommendationContext.set_strategy(strategy_class())

    return Response({"message": f"Global recommendation strategy set to '{strategy_name}'."},
            status=status.HTTP_200_OK)

@api_view(['GET'])
@permission_classes([IsAuthenticated])
def get_recommendations(request):
    """
    View to fetch recommended shows based on the chosen strategy.
    """
    try:
        # Fetch the global strategy
        current_strategy = GlobalRecommendationContext.get_strategy()
        location = request.GET.get('location', '').strip("'\" ")

        # Call the recommend method of the strategy
        if isinstance(current_strategy, LocationBasedRecommendation):
            recommended_shows = current_strategy.recommend(location=location)
        elif isinstance(current_strategy, TrendingRecommendation):
            recommended_shows = current_strategy.recommend()
        else:
            recommended_shows = current_strategy.recommend()

        return Response(
            {"message": "Recommendations fetched successfully.", "recommendations":recommended_shows},
            status=status.HTTP_200_OK,
        )
    except ValueError as e:
        return Response({"error": str(e)}, status=status.HTTP_400_BAD_REQUEST)
```

Figure 27: Context and abstract base class for recommendation strategies

```python
class Subject():
    """
    The Subject class holds a list of observers and notifies them when there is a state change.
    """
    def __init__(self):
        self._observers = []

    def attach(self, observer):
        if observer not in self._observers:
            self._observers.append(observer)

    def detach(self, observer):
        if observer in self._observers:
            self._observers.remove(observer)

    def notify(self, message):
        raise NotImplementedError("Subclasses should implement this method")


class Observer():
    """
    The Observer class defines the method that will be called to update the observer when the subject changes.
    """
    def update(self, message):
        raise NotImplementedError("Subclasses should implement this method")
```

Figure 28: Subject and Observer Class definition

8.3.6 Observer Pattern

The Observer design pattern is a behavioral design pattern that establishes a one-to-many dependency between objects, ensuring that when one object (the subject) changes its state, all its dependents (observers) are notified and updated automatically. This is accomplished by defining a subject interface for attaching and detaching observer objects, promoting loose coupling between the subject and its observers. This pattern aligns with the principle of "Design to Interfaces, Not Implementations," enabling flexible and scalable communication.

In the concert hall booking system, the Observer Design Pattern is utilized to ensure that the show producers and customers, are kept informed of updates related to shows and tickets. The Subject class is responsible for maintaining a list of observers and notifying them when there is a state change. In this instance, the 'Show' and 'Ticket' models can be referred to as subjects while the show producer and customers are observers. As the shows and tickets undergo changes, the change of status of a show or ticket would prompt the respective observers to be notified as well.

For example, when the status of a show changes—whether it is scheduled, completed, cancelled, or rejected—the 'Show' model notifies the registered observer (typically the show producer) using the 'notify' method. The show producer, who is the observer, is then updated with the new status of the show. Similarly, in the case of ticket cancellations, the 'Ticket' model notifies the customer, alerting them about the ticket's cancellation and the impacted show's status.

By using the observer pattern, the system remains decoupled, where the subject does not need to know about the concrete observers. This promotes flexibility, allowing for easy addition of new observers (e.g., adding email notifications or alerts) without affecting the subject's core logic.

```python
class Show(models.Model, Subject):
    name = models.CharField(max_length=50)
    has_intermission = models.BooleanField()

    STATUS_CHOICES = [(showStatus.name, showStatus.name) for showStatus in ShowStatusEnum]
    status = models.CharField(max_length=10, choices=STATUS_CHOICES, default=ShowStatusEnum.PENDING.name)

    hall = models.ForeignKey(Hall, on_delete=models.CASCADE, null=True, related_name="shows")
    slot = models.ForeignKey(Slot, on_delete=models.CASCADE, null=True, related_name='shows')
    category = models.ForeignKey(Category, on_delete=models.CASCADE, null=True, related_name="shows")
    show_producer = models.ForeignKey(ShowProducer, on_delete=models.SET_NULL, null=True, related_name='shows')

    def attach(self, observer):
        # observer is interested in show approval / rejection
        # show producer is stored inside show table
        self.show_producer = observer
        self.save()

    def notify(self, message=""):
        # notify the registered observer (show producer) about the status change.
        self.show_producer.update(f"Show '{self.name}' status changed to {self.status}. {message}")

    def get_status_instance(self):
        status_class = STATUS_CLASSES.get(self.status)
        if status_class:
            return status_class(self)
        raise ValueError(f"Unknown status: {self.status}")

    def schedule(self):
        status_instance: ShowStatus = self.get_status_instance()
        if(status_instance and not isinstance(status_instance, PendingStatus)):
            raise ValueError(INVALID_PENDING_STATUS_ERROR)
        status_instance.transition_to_scheduled()
        self.notify()

    def reject(self, message):
        status_instance: ShowStatus = self.get_status_instance()
        if(not isinstance(status_instance, PendingStatus)):
            raise ValueError(INVALID_PENDING_STATUS_ERROR)
```

Figure 29: Show - Attach and Notify methods

```python
class Customer(models.Model, Observer):
    user = models.OneToOneField(User, on_delete=models.CASCADE, default="")
    phone = models.CharField(max_length=15, default="")
    loyalty_points = models.IntegerField(default=0)

    def __str__(self):
        return f"{self.user.first_name} {self.user.last_name} (Customer)"

    def update(self, message):
        """
        This method is called by the Subject (Show) when its status changes.
        """
        CustomerNotifications.objects.create(
            customer=self,
            message=message,
            isRead=False
        )

class ShowProducer(models.Model, Observer):
    user = models.OneToOneField(User, on_delete=models.CASCADE, default="")
    phone = models.CharField(max_length=15, default="")
    organisation = models.CharField(max_length=30, default="")
    def __str__(self):
        return f"{self.user.first_name} {self.user.last_name} (Show Producer)"

    def update(self, message):
        """
        This method is called by the Subject (Show) when its status changes.
        """
        ShowProducerNotifications.objects.create(
            show_producer=self,
            message=message,
            isRead=False
        )
```

Figure 30: Observer update method

```python
def calculate_loyalty_points(self, customer, latest_valid_membership):
    if not latest_valid_membership:
        return

    existing_loyalty_points = customer.loyalty_points or 0

    regular_loyalty = RegularLoyalty()
    new_customer_loyalty = NewCustomerLoyaltyDecorator(regular_loyalty, customer)
    loyalty_with_membership = MembershipLoyaltyDecorator(new_customer_loyalty, latest_valid_membership)

    calculated_loyalty_points = loyalty_with_membership.get_loyalty_points()

    total_loyalty_points = existing_loyalty_points + calculated_loyalty_points
    customer.loyalty_points = total_loyalty_points
    customer.save()
```

Figure 31: Calculation of loyalty points in Customer Membership Model

8.3.7 Decorator Pattern

The Decorator design pattern is a structural design pattern that allows behavior to be added to individual objects, dynamically, without modifying their class. This is accomplished by defining a set of decorator classes that are used to wrap concrete components, each adding its own behavior while delegating to the wrapped object. This pattern aligns with the principle of "Open-Closed Principle," enabling classes to be open for extension but closed for modification, promoting flexible and reusable designs.

In the concert hall booking system, the Decorator Design Pattern is used to dynamically add functionality to an object without altering its underlying structure. This is especially useful when enhancing loyalty point calculations for customers based on different conditions like their membership status or ticket purchase history.

The base class LoyaltyDecorator sets the ground for decorator classes by announcing the method get_loyalty_points that is called when the class has to adjust the calculations of points. The RegularLoyalty normal class is the first operational model which gives a thumb out for a specific and fixed number of points, which in this case is 10. The NewCustomerLoyaltyDecorator takes it a step further and looks at how many tickets the customer has purchased. If the customer has made their first ticket purchase and is new to our system, the loyalty points are increased by 25%. MembershipLoyaltyDecorator increases the loyalty points by applying the corresponding loyalty booster based on the membership type of the customer.

```python
class LoyaltyDecorator(ABC):
    @abstractmethod
    def get_loyalty_points(self):
        pass

class RegularLoyalty(LoyaltyDecorator):
    def get_loyalty_points(self):
        return 10

class NewCustomerLoyaltyDecorator(LoyaltyDecorator):
    def __init__(self, loyalty_decorator: LoyaltyDecorator, customer):
        self.loyalty_decorator = loyalty_decorator
        self.customer = customer

    def get_loyalty_points(self):
        loyalty_points = self.loyalty_decorator.get_loyalty_points()

        ticket_count = Ticket.objects.filter(customer=self.customer).count()

        if ticket_count == 1:
            return loyalty_points * 1.25

        return loyalty_points

class MembershipLoyaltyDecorator(LoyaltyDecorator):
    def __init__(self, loyalty_decorator: LoyaltyDecorator, customer_membership: Membership):
        self.loyalty_decorator = loyalty_decorator
        self.customer_membership = customer_membership

    def get_loyalty_points(self):
        loyalty_points = self.loyalty_decorator.get_loyalty_points()
        loyalty_booster = self.customer_membership.get_loyalty_booster()

        total_loyalty_points = loyalty_points * loyalty_booster
        return total_loyalty_points
```

Figure 32: Abstract and concrete classes for Decorator Pattern

```python
class TicketViewTemplate(ABC):
    def process_request(self, user):
        if not self.authenticate_user(user):
            raise PermissionError("User is not authorized to view tickets.")

        user = self.get_user(user)
        bookings = self.fetch_bookings(user)
        filtered_bookings = self.apply_filters(bookings)
        return filtered_bookings

    @abstractmethod
    def authenticate_user(self, user):
        """
        Must be implemented to define how a user is authenticated.
        """
        pass

    @abstractmethod
    def fetch_bookings(self, customer):
        """
        Must be implemented to fetch bookings for the given user.
        """
        pass

    @abstractmethod
    def get_user(self, user):
        """
        Override to customize how to retrieve the customer instance.
        """
        pass

    @abstractmethod
    def apply_filters(self, bookings):
        """
        Apply a chain of filters to bookings. Subclasses can override or add filters.
        """
        pass
```

Figure 33: TicketViewTemplate for Template Pattern

8.3.8 Template Pattern

The Template Method Pattern is a design approach where the main steps of an algorithm are outlined in a base class, while the details of those steps are left to be implemented by subclasses. This way, the overall structure of the algorithm stays the same, but the behavior of individual steps can vary as needed. This pattern is great for promoting code reuse because it keeps the shared workflow in one place while allowing flexibility for subclasses to handle specific details.

In the code, the ‘TicketViewTemplate‘ class as in Figure 33 serves as the base class that outlines the general process for handling ticket view requests (Figure 34). It defines the ‘process_request‘ method, which is the template method that organizes the sequence of operations: authenticating the user, retrieving the user instance, fetching the bookings, and applying filters to these bookings. The ‘CustomerTicketView‘ class as shown in Figure 35 extends ‘TicketViewTemplate‘ and provides concrete implementations for the abstract methods. For example, ‘authenticate_user‘ checks if the user has a customer attribute, ‘fetch_bookings‘ retrieves the bookings associated with the customer, and ‘apply_filters‘ processes

```python
@api_view(["GET"])
@permission_classes([IsAuthenticated])
def customer_view_tickets(request):
    try:
        customer_view = CustomerTicketView()
        response_data = customer_view.process_request(get_current_user())
        serialized_tickets = []
        for ticket in response_data:
            serialized_ticket = {
                "show": ticket.show.name,
                "venue":ticket.seat.hall.hall_name ,
                "date": ticket.getShowDate(),
                "time": ticket.getShowTimimg()
            }
            serialized_tickets.append(serialized_ticket)
        logger.info(f"{get_current_user()} viewed booked tickets")
        return JsonResponse(serialized_tickets, safe=False)

    except PermissionError as e:
        return JsonResponse({"error": str(e)}, status=403)
```

Figure 34: View for view booked tickets

```python
class CustomerTicketView(TicketViewTemplate):
    def authenticate_user(self, user):
        return hasattr(user, 'customer')

    def fetch_bookings(self, customer):
        return Ticket.objects.filter(customer=customer)

    def apply_filters(self, bookings):
        upcoming_bookings = self.filter_upcoming(bookings)
        bookings = self.filter_cancelled(upcoming_bookings)
        return bookings

    def filter_upcoming(self, bookings):
        upcoming_tickets = []
        for ticket in bookings:
            show_time = ticket.getShowTimimg()
            show_date = ticket.getShowDate()
            show_datetime = datetime.combine(show_date, TIMING_TO_TIME[show_time])
            if datetime.now() < show_datetime:
                upcoming_tickets.append(ticket)
        return upcoming_tickets

    def filter_cancelled(self, bookings):
        return [ticket for ticket in bookings if not ticket.isCancelled]

    def get_user(self, user):
        return Customer.objects.get(user=get_current_user())
```

Figure 35: CustomerTicketView for Template Pattern

the bookings to filter out only upcoming and non-cancelled tickets. This way, the ‘CustomerTicketView‘ class customizes the behavior for handling ticket views for customers, while still adhering to the general process defined by the template method in the base class.

By using the Template Method Pattern, the code maintains a clear and consistent structure for handling ticket view requests while allowing for flexibility in the details. This approach enhances code readability and maintainability by encapsulating common workflows in the base class and delegating specific behaviors to subclasses. It also makes it easy to introduce new user types with different requirements by simply creating new subclasses that implement the required methods, without altering the overall algorithm structure like if admin want to know the booked tickets of certain customer or want to get the whole list of who bought which tickets.

8.3.9 Command Pattern

The Command Pattern is a design pattern, which turns a request into a stand alone Command object. This approach takes a request and bundles it into an object containing all the details needed to execute the request, which makes the whole process more flexible and easier to handle.

In this implementation, the 'Command' class is the base abstract class for all other command objects as shown in Figure 37. It contains an execute method that will be overridden in subclasses to implement the desired behavior. In our code, the classes like CancelTicketCommand and RefundCommand are subclasses of Commands as shown in Figure 36. All of these classes have the execute method which is responsible for a specific task like ticket cancellation, or for performing specific business operations like refunding money. We achieve separation of concerns by concentrating on one operation within a command class, thereby enhancing the modularity of the code, making it easier to manage. The ‘CommandInvoker‘ class acts as invoker which maintains these commands. It holds references to the command objects and delegates their execution when the ‘execute‘ method is called. This ensures that the invoker doesn't need to know the specifics of how each action is performed, everything is handled by the respective command objects.

An example of this pattern in action is the ‘cancel_ticket‘ function. It starts by checking if the user has the right authorization. Then it creates the necessary command objects (‘CancelTicketCommand‘, ‘RefundCommand‘) and passes them to the ‘CommandInvoker‘. The control then executes the commands in sequence, ensuring the ticket cancellation process runs smoothly and consistently. This is demonstrated in Figure 38.

The code organization is the key factor of Command Pattern which makes it manageable. Each operation is neatly encapsulated in its own command class, making the system modular and easier to maintain. Adding new commands or modifying existing ones doesn't disrupt the rest of the code. This pattern is especially useful for managing complex workflows where multiple actions need to happen in a specific order.

```python
class CancelTicketCommand(Command):
    def __init__(self, ticket_ids, customer):
        self.ticket_ids = ticket_ids
        self.customer = customer

    def execute(self):
        # Fetch the ticket from the database
        tickets=[]
        for t_id in self.ticket_ids:
            ticket = Ticket.objects.get(id=t_id, customer=self.customer)
            # Cancel the ticket
            ticket.cancel()
            tickets.append(ticket)
        return tickets

class RefundCommand(Command):
    def __init__(self, customer, ticket_ids):
        self.customer = customer
        self.ticket_ids = ticket_ids

    def execute(self):
        try:
            # Utilize the RefundService for calculating refund
            refund_service = RefundService()
            refund_amount=0
            refund = refund_service.get_ticket_refund(self.ticket_ids)
            if refund!=None:
                refund_amount+=refund
            if refund_amount is False:
                raise Exception("Refund calculation failed.")
            # Mock actual refund processing
            return f"Refund Initiated."
        except Exception as e:
            raise Exception(f"Refund processing failed: {str(e)}")

class CommandInvoker:
    def __init__(self, cancel_command, refund_command):
        self.cancel_command = cancel_command
        self.refund_command = refund_command

    def commandExecute(self):
        cancel_message = self.cancel_command.execute()
        refund_message = self.refund_command.execute()
        return cancel_message, refund_message
```

Figure 36: Concrete commands and Invoker in Command Pattern

```python
from abc import ABC, abstractmethod

class Command(ABC):

    @abstractmethod
    def execute(self):
        pass
```

Figure 37: Command interface

```python
@api_view(["POST"])
@permission_classes([IsAuthenticated])
def cancel_ticket(request):
    user = get_current_user()
    if not hasattr(user, 'customer'):
        return Response({"error": "The logged-in user is not a customer."}, status=403)

    customer = user.customer
    print(customer)
    serializer = TicketCancellationSerializer(data=request.data, context={"request": request})
    if serializer.is_valid():
        tickets = serializer.context["validated_tickets"]
        try:
            # Create the command objects
            cancel_command = CancelTicketCommand(ticket_ids=tickets, customer=customer)
            refund_command = RefundCommand(ticket_ids=tickets, customer=customer)

            # Execute the service
            service = CommandInvoker(
                cancel_command=cancel_command,
                refund_command=refund_command
            )
            canceled_tickets = service.commandExecute()
            logger.info(f"Tickets {tickets} Cancelled by {customer.user.email}")

            # Return a success response
            return Response({
                "status": "success",
                "tickets": [ticket.id for ticket in canceled_tickets[0]],
                "message": f"Successfully canceled {len(canceled_tickets[0])} ticket(s).{canceled_tickets[1]}"
            }, status=200)

        except Exception as e:
            logger.error(str(e))
            return Response({
                "status": "error",
                "message": str(e)
            }, status=400)
```

Figure 38: View for Cancel booked ticket

```python
@api_view(["GET"])
@permission_classes([AllowAny])
def get_halls(request: HttpRequest):
    try:
        category_id = request.GET.get('category_id')
        category = get_object_or_404(Category, id=category_id)

        slot_id = request.GET.get('slot_id')
        slot = get_object_or_404(Slot, id=slot_id)

        supporting_halls = Hall.get_halls_by_category_and_slot(category=category, slot=slot)
        available_supporting_halls = []

        for hall in supporting_halls:
            if not Show.is_overlapping_show_exists(hall=hall, slot=slot):
                available_supporting_halls.append(model_to_dict(hall))

        return JsonResponse({
            'halls_response': available_supporting_halls
        }, status=200)
    except Exception as e:
        print(e)
```

Figure 39: View for get halls API

8.4 Key Use-cases

8.4.1 Raise show request

One of the primary use cases was the show producer browsing the available halls and raising a show request for a particular hall and slot. The API in Figure 39 provides the list of halls that support the category and slot selected by the producer. There are checks to ensure that no other overlapping shows are scheduled. Figure 40 shows the use of a Serializer to validate the inputs provided by the show producer. The API view (Figure 41) handles the validations and passes the request to the ShowRequestService (Figure 10) which creates or updates the show object in pending status and emits an event for asynchronous processing of the request.

8.4.2 Book Ticket

This usecase enables logged in customers to book tickets for a show that is scheduled in the concert hall. The BookTicket API begins by validating the incoming request data, including the selected seats and the show by using serializer ().(Figure 42) After validation, it checks if the requested seats are available for booking using the return_available_seats function. If the seats are available, the API proceeds to calculate the total ticket price through the PaymentGatewayFacade, which includes any discounts based on the customer's membership.

Once the payment details are confirmed, the API creates the tickets for the customer using the create_ticket function. If the ticket creation is successful, the API returns the ticket IDs and the total amount. If any step fails—such as seat unavailability, payment failure, or ticket creation errors—the API responds with appropriate error messages.

```python
class CreateShowRequestSerializer(serializers.Serializer):
    show_id = serializers.IntegerField(required=False)
    name = serializers.CharField()
    category_id = serializers.IntegerField()
    has_intermission = serializers.BooleanField()
    slot_id = serializers.IntegerField()
    hall_id = serializers.IntegerField()

    def validate(self, data):
        # Validate category existence
        category_id = data.get("category_id")
        if not Category.objects.filter(id=category_id).exists():
            raise serializers.ValidationError("Invalid category ID.")

        data['category'] = Category.objects.filter(id=category_id).first()

        # Validate slot existence
        slot_id = data.get("slot_id")
        if not Slot.objects.filter(id=slot_id).exists():
            raise serializers.ValidationError("Invalid slot ID.")

        data['slot'] = Slot.objects.filter(id=slot_id).first()

        # Validate hall existence
        hall_id = data.get("hall_id")
        if not Hall.objects.filter(id=hall_id).exists():
            raise serializers.ValidationError("Invalid hall ID.")

        data['hall'] = Hall.objects.filter(id=hall_id).first()

        show_id = data.get('show_id')
        if show_id:
            if not Show.objects.filter(id=show_id).exists():
                raise serializers.ValidationError("Invalid show ID.")
            show = Show.objects.filter(id=show_id).first()
            if not isinstance(show.get_status_instance(), PendingStatus):
                raise serializers.ValidationError("Show is not in pending status.")

        return data
```

Figure 40: Create show request serializer performs data validations

```python
@api_view(["POST"])
@permission_classes([IsAuthenticated])
def create_update_show_request(request: HttpRequest):
    show_producer = ShowProducer.objects.get(user=get_current_user())

    body = json.loads(request.body)
    # Validate input with serializer
    serializer = CreateShowRequestSerializer(data=body)
    serializer.is_valid(raise_exception=True)
    validated_data = serializer.validated_data

    name = validated_data['name']
    category = validated_data['category']
    has_intermission = validated_data['has_intermission']
    slot = validated_data['slot']
    hall = validated_data['hall']

    show_id = validated_data.get('show_id', None)

    # Use the service to request a show
    show = ShowRequestService.request_show(
        show_id,
        show_producer,
        name,
        category,
        has_intermission,
        slot,
        hall,
    )

    return JsonResponse({
        'show_response': model_to_dict(show)
    })
```

Figure 41: View for raise show request API

```python
class BookTicketSerializer(serializers.Serializer):

    show_id = serializers.IntegerField(required=True)
    seats = serializers.ListField(
        child=serializers.IntegerField(),
        required=True
        )       Adarsh Ajit, last week • Revert "Revert "Create payment gateway app""

    def validate(self, attrs):
        attrs = super().validate(attrs)
        show_id = attrs.get('show_id')
        seat_list = attrs.get('seats')

        show_obj = get_object_or_404(Show, id=show_id)

        if any(seat > show_obj.hall.hall_capacity for seat in seat_list):
            raise serializers.ValidationError({"seats":"Some of the requested seats do not exist in the hall."})

        attrs['show_obj'] = show_obj

        return attrs
```

Figure 42: Serializer with validations for ticket

```python
@swagger_auto_schema(
    request_body=BookTicketSerializer,
    method='POST'
)
@api_view(["POST"])
@permission_classes([IsAuthenticated])
def book_tickets(request: HttpRequest):
    customer = Customer.objects.get(user=get_current_user())
    customer_membership = CustomerMembership.objects.filter(customer=customer).first()
    latest_valid_membership = customer_membership.get_latest_valid_membership_instance(customer=customer)

    data = json.loads(request.body)

    serializer = BookTicketSerializer(data=data)
    serializer.is_valid(raise_exception=True)

    validated_data = serializer.validated_data

    if seat_objs := return_available_seats(validated_data['show_obj'], validated_data['seats']):

        payment_gateway = PaymentGatewayFacade()
        price_per_ticket, bill_amount = payment_gateway.get_ticket_bill_amount(
            customer, seat_objs)

        if bill_amount:

            if tickets := create_ticket(customer, validated_data['show_obj'], seat_objs, price_per_ticket):

                customer_membership.calculate_loyalty_points(customer=customer, latest_valid_membership=latest_valid_membership)

                return JsonResponse({
                    "ticket_ids": list(tickets),
                    "total_amount": bill_amount
                }, status=HTTP_200_OK)

            return JsonResponse({"error": "Something went wrong creating tickets."}, status=HTTP_404_NOT_FOUND)

        return JsonResponse({"error": "Payment Failed."}, status=HTTP_404_NOT_FOUND)

    return JsonResponse({"error": "Seat not available."}, status=HTTP_404_NOT_FOUND)
```

Figure 43: View for book ticket API

```python
def cancel(self, message=""):
    status_instance: ShowStatus = self.get_status_instance()
    # producer is cancelling a pending show, no need for notifications
    if(status_instance and isinstance(status_instance, PendingStatus)):
        status_instance.transition_to_cancelled()

    if(status_instance and isinstance(status_instance, ScheduledStatus)):
        status_instance.transition_to_cancelled()
        self.notify(message=message)
```

Figure 44: Show model - cancel method

```python
@staticmethod
def cancelled_show(show, message=""):
    """
    Notify all customers who have tickets for the show that the show has been
    cancelled.
    """
    tickets = Ticket.objects.filter(show=show)

    for ticket in tickets:
        ticket.notify(message=f"We regret to inform you that the show has been
        cancelled, and your ticket with id:{ticket.id} is no longer valid.")
```

Figure 45: Ticket Model - cancelled_show method

8.4.3 Admin cancels show

When an admin cancels a show, the system ensures that both the show producer and customers are notified. The cancellation request is first validated through the cancel_show API endpoint, which checks if the user is an authenticated admin. Once the request is verified, the cancel method in the Show model transitions the show status to "Cancelled" if it was previously scheduled. A notification is sent to the show producer, informing them of the cancellation.

Simultaneously, the cancelled_show method Figure 45 in the Ticket model notifies all customers who purchased tickets for the show, informing them that their tickets are no longer valid. This ensures timely communication to all relevant stakeholders, keeping both the producer and customers up-to-date on the status change, while maintaining a clear flow of information in the system.

8.4.4 Cancel Booked Ticket

Customer have the ability to cancel booked tickets within a certain time period which is determined by their membership. The CancelTicket API (Figure 38) begins by checking if the current user is a customer. A series of validations is carried out by the TicketCancellationSerializer which check if the ticket id is valid, if the logged in customer have bought the ticket, is the ticket already already cancelled and is is permissible to cancel the ticket with the customer's current membership cancellation policy.

```python
@api_view(["PUT"])
@permission_classes([IsAuthenticated])
def cancel_show(request):
    admin_user = get_current_user().is_superuser

    if not admin_user:
        return JsonResponse({"error": PERMISSION_DENIED_ERROR}, status=403)

    body = json.loads(request.body)
    # Validate input with serializer
    serializer = CancelShowSerializer(data=body, admin_user=admin_user)
    serializer.is_valid(raise_exception=True)
    validated_data = serializer.validated_data

    show: Show = validated_data['show']

    message = f"The Show: {show.name} has been cancelled. Sorry for the inconvenience"
    show.cancel(message=message)
    Ticket.cancelled_show(show=show, message=message)

    return Response({"message": "Show canceled successfully."}, status=status.HTTP_200_OK)
```

Figure 46: Show view - cancel_show method

After the validations, the operations are carried out by the commands, namely CancelTicketCommand and RefundCommand (Figure 36) which will encapsulate each actions. First the tickets are cancelled and the refund action is initiated and then the corresponding loyalty points are deducted from the customer based on the membership features. This is executed with the help of Command pattern.

8.5 Testing

Pytest is a robust testing framework for Python, known for its ease of use and scalability. It's designed to make writing and running tests simple, offering a wide range of features that cater to both beginners and experienced developers. Pytest supports fixtures for setting up testing environments, parameterized testing for running the same test with different data, and a variety of plugins that extend its functionality. Its user-friendly syntax and powerful capabilities make it a popular choice for developers looking to ensure their code is reliable and bug-free.

Running tests with pytest is straightforward. We can simply write our test functions using standard Python, prepending them with "test_" so pytest can identify them. When we execute the 'pytest' command, the framework automatically discovers these test functions, runs them, and reports the results. Pytest's output is clean and easy to understand, highlighting both the successful and failed tests. If a test fails, pytest provides detailed information on the failure, including the specific line of code that caused the issue, making it easier to debug and fix problems.

One of the standout features of pytest is its flexibility in running tests. We can run all tests in our project, specific test files, or even individual test functions, offering a high degree of control over the testing process. Whether we are working on a small project or a large application, pytest's powerful features and ease of use make it an excellent choice for managing our testing needs. Figure 47 shows

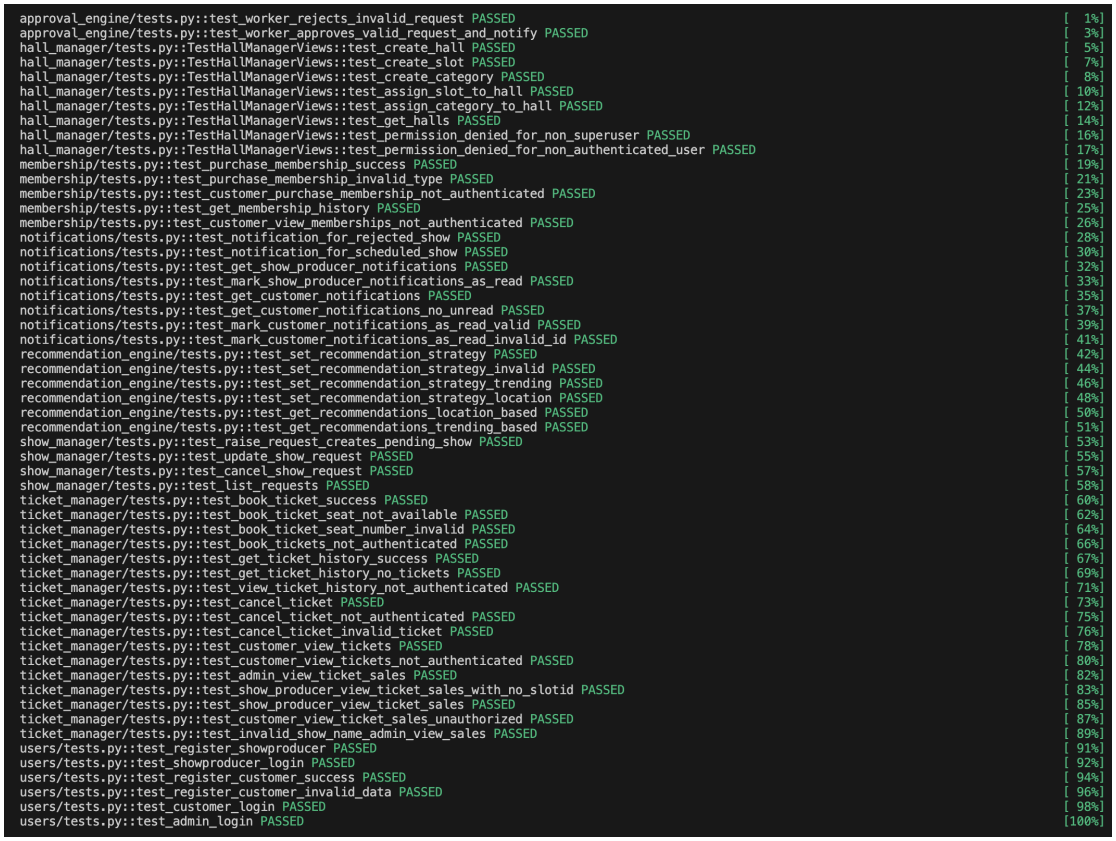


Figure 47: Sample result of Pytest

a sample of tests implemented in our project.

8.6 Github - Branching Strategy

In our concert hall booking system project, we implemented version control using GitHub to ensure smooth collaboration and track changes in the codebase. Developers followed a clear branching strategy to maintain organization and minimize conflicts. Each developer created branches using a consistent naming convention: ‘feature/name-of-the-developer-feature-name‘ for new features and ‘fix/name-of-the-developer-fix-name‘ for bug fixes. This naming convention ensured that the purpose of each branch was clear, and it facilitated easier navigation through the project history.

Once the changes were made, developers raised pull requests (PRs) to merge their branches into the main codebase. The PRs were then reviewed and approved by the technical lead to ensure code quality and consistency. This process helped maintain the stability of the application while allowing for rapid development. The team was able to collaborate effectively, avoid code conflicts, and ensure that all features and fixes were properly integrated into the project.

Link to GitHub repository: https://github.com/CS5721-VonNeumann/concert-hall-booking-system

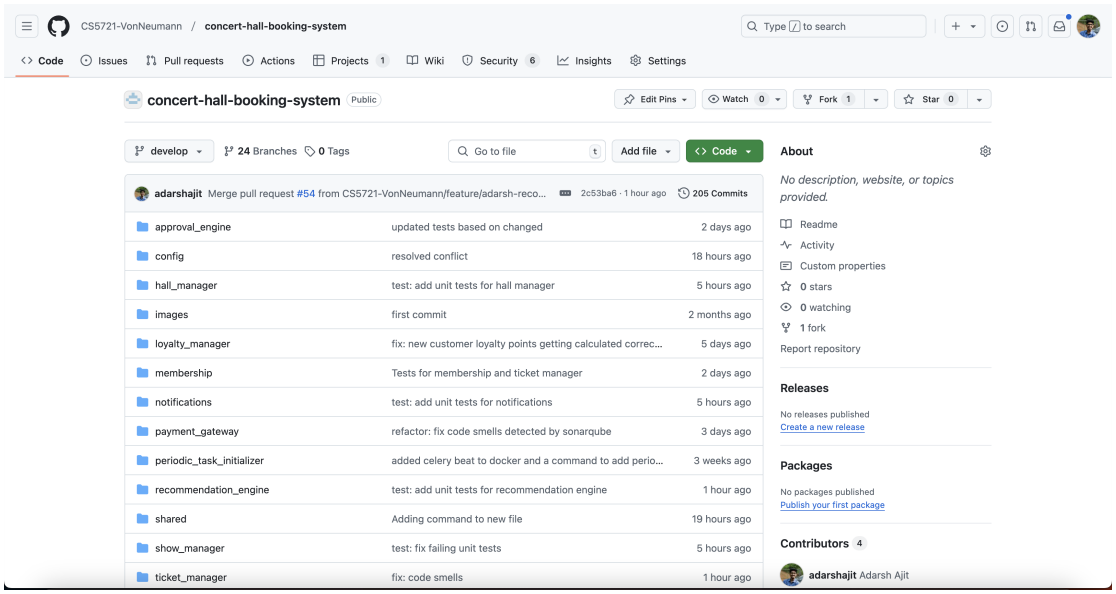


Figure 48: Github repository of Concert Hall Booking System

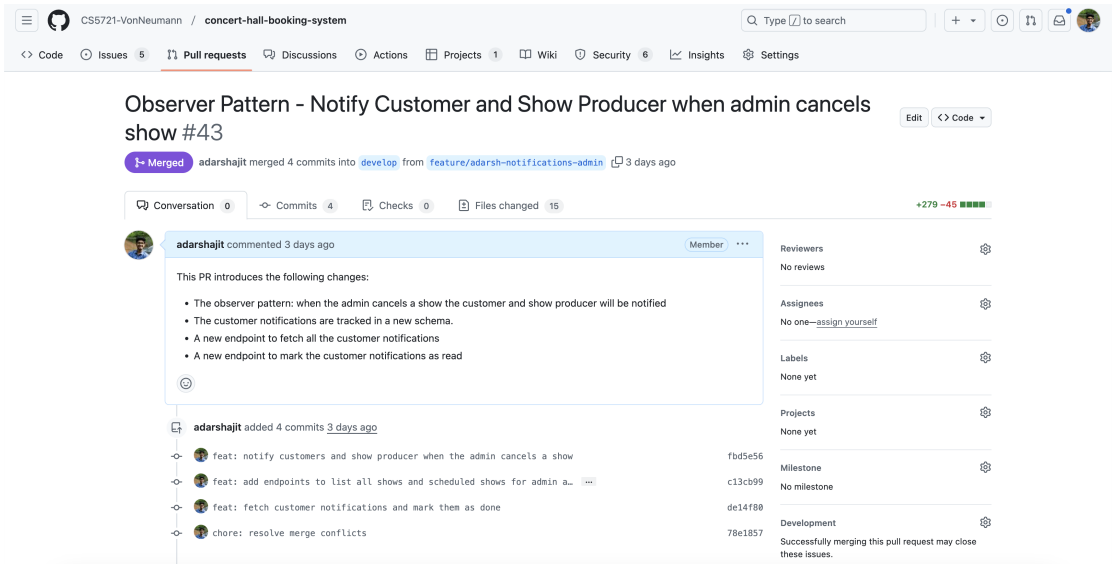


Figure 49: Pull Request raised to develop branch

```
SWAGGER_SETTINGS = {
    'SECURITY_DEFINITIONS': {
        'api_key': {
            'type': 'apiKey',
            'in': 'header',
            'name': 'Authorization'
        }
    }
}
```

Figure 50: Swagger configuration

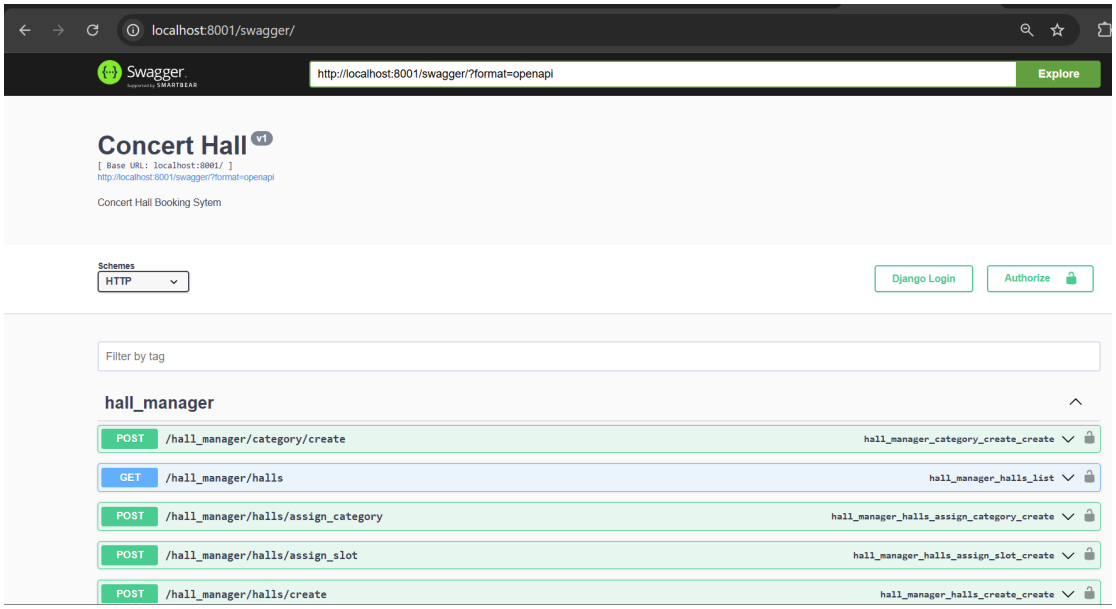


Figure 51: Swagger Setup

9 Added Value

9.1 Swagger UI

Swagger UI has been integrated into this project as a user interface for interacting with the major APIs, following the REST architectural pattern. By leveraging the Django REST Framework and the drf-yasg library, Swagger UI automatically generates interactive documentation for the available API endpoints. It provides a visual interface where we can view and test API operations, such as booking tickets or raising show requests, without the need to write custom requests manually.

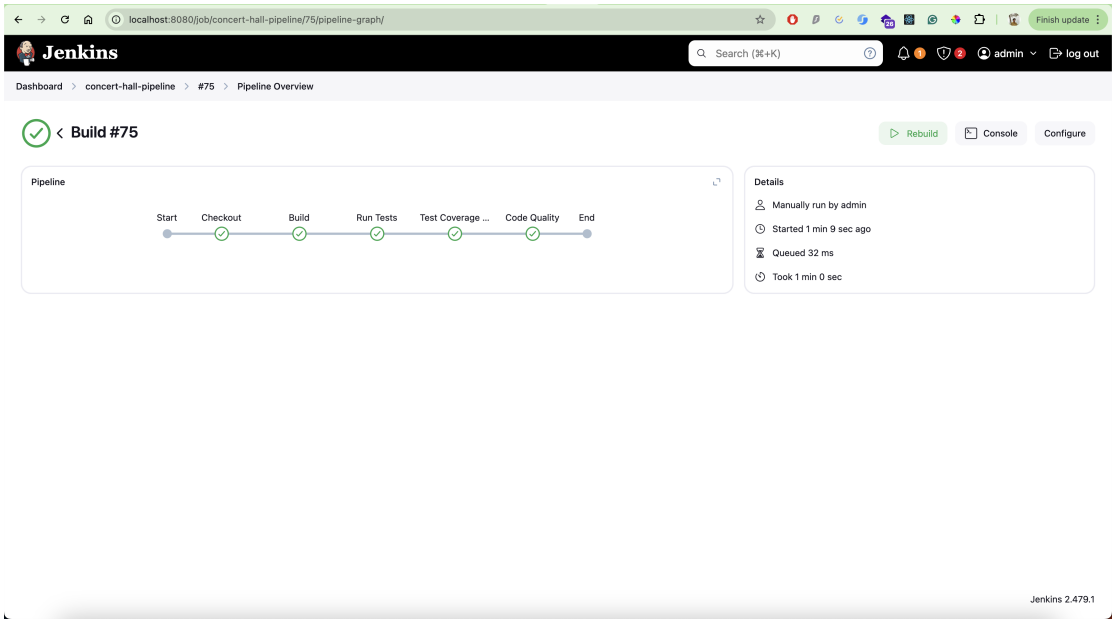


Figure 52: Jenkins Pipeline Overview for Build No: 75

9.2 Jenkins

In modern software development, Continuous Integration and Continuous Deployment (CI/CD) pipelines are crucial for automating the process of integrating code changes and deploying applications. We have set up a Jenkins pipeline to automate these processes for our Django-based concert hall booking system. The pipeline, defined in the Jenkinsfile, includes multiple stages such as checkout, build, run tests, generate test coverage reports, and perform code quality analysis. By utilizing Jenkins, we ensure that each code change is thoroughly tested, integrated, and prepared for deployment without manual intervention. Additionally, we enforce strict rules for branch naming conventions to maintain a structured and consistent workflow. Branch names must start with 'feature/', 'fix/' etc, ensuring that feature, and bug-fix tasks are clearly identified and appropriately handled within the pipeline.

The Jenkins pipeline defines environment variables essential for the application, such as database credentials and RabbitMQ configurations. During the build stage, it uses Docker Compose to build the application's services, ensuring that each component, including the database, RabbitMQ, and Celery, is correctly set up. The pipeline also runs unit tests and generates code coverage reports. Additionally, the pipeline integrates SonarQube for static code analysis which helps to identify potential issues early in the development cycle. This CI/CD setup ensures that our application is always in a deployable state, improving reliability and accelerating development while adhering to a clear and organized branch management strategy.

```
pipeline {
    agent any

    environment {
        ENVIRONMENT = 'development'
        DEBUG = '1'
        SECRET_KEY = 'django-insecure-+$ap&pghgqprwiio*-*m!el$9(h(i8znd^-90bj+akflpwhcz1'
        DJANGO_ALLOWED_HOSTS = 'localhost,127.0.0.1,0.0.0.0,[::1]'

        DATABASE_ROOT_PASSWORD = 'password@123'
        DATABASE_ENGINE = 'django.db.backends.mysql'
        DATABASE_NAME = 'djangoexperiments'
        DATABASE_USER = 'user'
        DATABASE_USER_PASSWORD = 'password@123'
        DATABASE_HOST = 'db'
        DATABASE_PORT = '3306'

        RABBITMQ_DEFAULT_USER = 'guest'
        RABBITMQ_DEFAULT_PASS = 'guest'
        CELERY_BROKER_URL = 'amqp://guest:guest@rabbitmq:5672//'
    }

    stages {
        stage('Checkout') {
            steps {
                checkout scmGit(
                    branches: [[name: '*/develop'], [name: 'feature/*'], [name: 'fix/*']],
                    extensions: [],
                    userRemoteConfigs: [[url: 'https://github.com/CS5721-VonNeumann/concert-hall-booking-system.git']]
                )
            }
        }

        stage('Build') {
            steps {
                sh 'docker-compose build'
            }
        }

        stage('Run Tests') {
            steps {
                sh '/Users/adarshajit/.local/share/virtualenvs/concert-hall-booking-system-fdD0L9Df/bin/pytest -v'
            }
        }

        stage('Test Coverage Report') {
            steps {
                sh '/Users/adarshajit/.local/share/virtualenvs/concert-hall-booking-system-fdD0L9Df/bin/pytest --cov=. --cov-report=xml'
            }
        }

        stage('Code Quality') {
            steps {
                sh '/opt/homebrew/bin/sonar-scanner -Dproject.settings=/Users/adarshajit/Documents/concert-hall-booking-system/sonar-scanner.properties'
            }
        }
    }
}
```

Figure 53: Jenkinsfile for the application

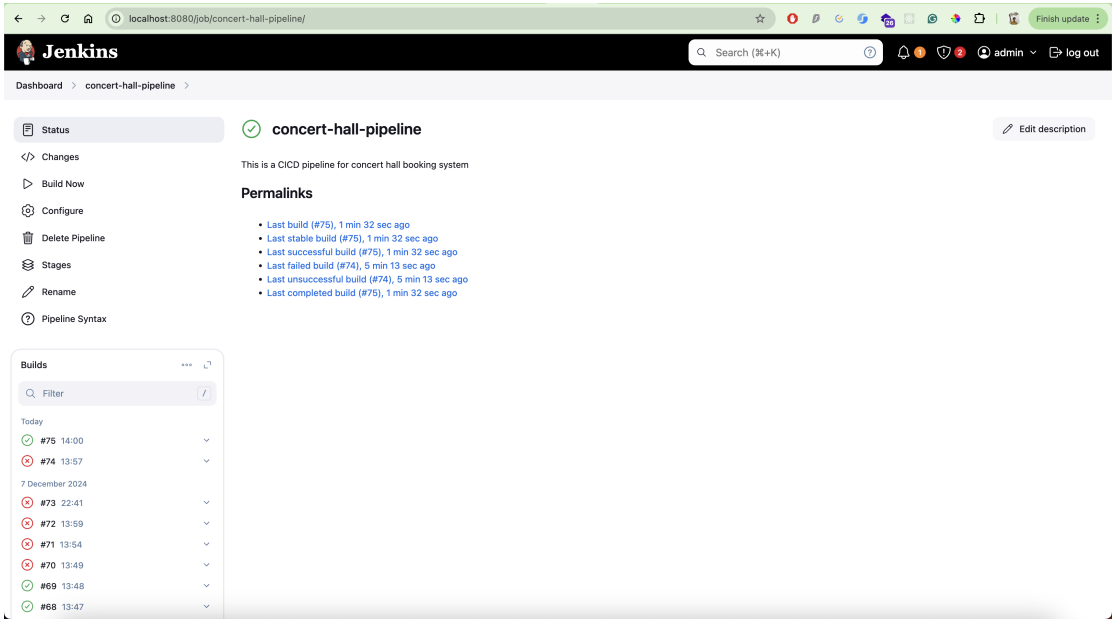


Figure 54: Jenkins Dashboard

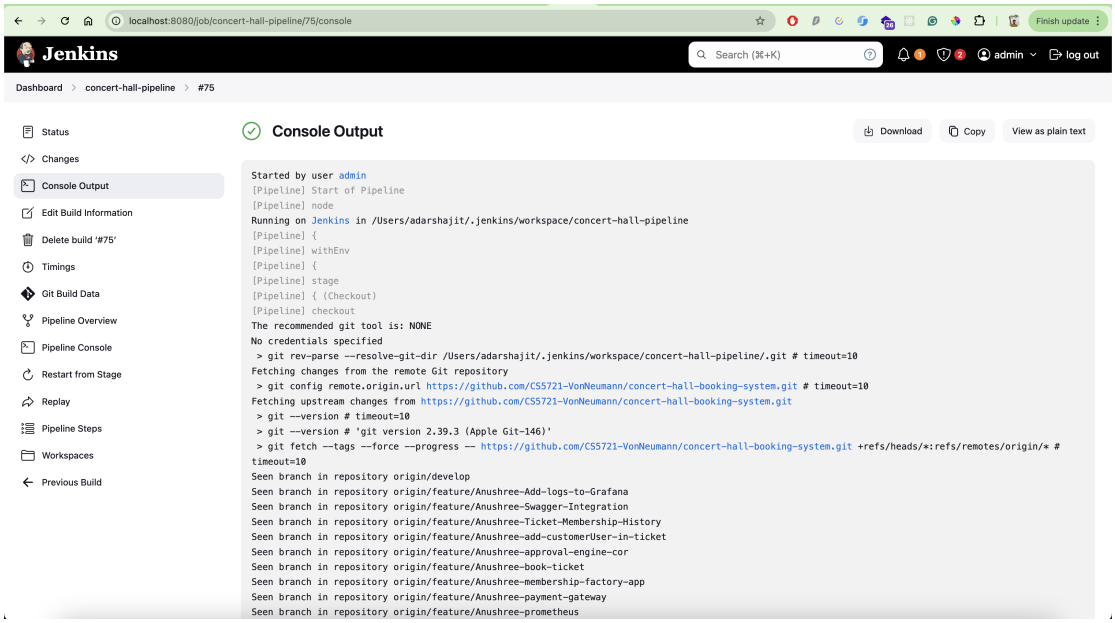


Figure 55: Console output for the build logs

9.3 SonarQube

To ensure high code quality and maintainability in our concert hall booking system, we have integrated SonarQube into our development workflow. SonarQube automatically analyzes our codebase for potential issues, such as bugs, vulnerabilities, and code smells, providing detailed feedback to developers. This integration helps maintain a high standard of code quality throughout the development lifecycle. By running SonarQube on every code commit, we can quickly identify and resolve problems, ensuring that the system remains robust, secure, and efficient. The feedback from SonarQube also promotes best practices in code writing and fosters a culture of continuous improvement within the team.

SonarQube identified a total of 43 code smells in our codebase, including unused variables (app, request, env, etc.), duplicated literals (e.g., "Invalid request method.", "hall_manager.hall"), improper exception handling (Figure 59), and unnecessary imports. Additionally, some function names were in camelCase instead of snake case (Figure 60), violating Python's PEP 8 naming conventions. To resolve these issues, we removed unused variables and parameters, replaced duplicated literals with constants, renamed functions to align with snake case naming conventions, refined exception handling by specifying appropriate exception classes, and optimized imports by including only the necessary modules or members.

9.4 Celery and Celery Beat

The rise of microservices and event-driven architectural patterns has led to the evolution of various asynchronous task-processing technologies. We utilise Celery to handle asynchronous workflows in our application. The message broker utilised to act as a task queue runs on RabbitMQ (Figure 61). Figure 63 shows the setup

```
# SonarQube project configuration
sonar.projectKey=concert-hall-booking-system
sonar.projectName=Concert Hall Booking System
sonar.projectVersion=1.0

# Path to sources and tests
sonar.sources=.
sonar.tests=.

# Include test files in the analysis
sonar.test.inclusions=**/tests.py

# Python coverage report
sonar.python.coverage.reportPaths=coverage.xml

# Python-specific settings (if needed)
sonar.python.version=3.12

# Additional configurations for SonarQube
sonar.host.url=http://localhost:9001
sonar.login=sqp_c29bad0740d36d8524fac64204170f8b5cf6b0c0
```

Figure 56: SonarQube properties file

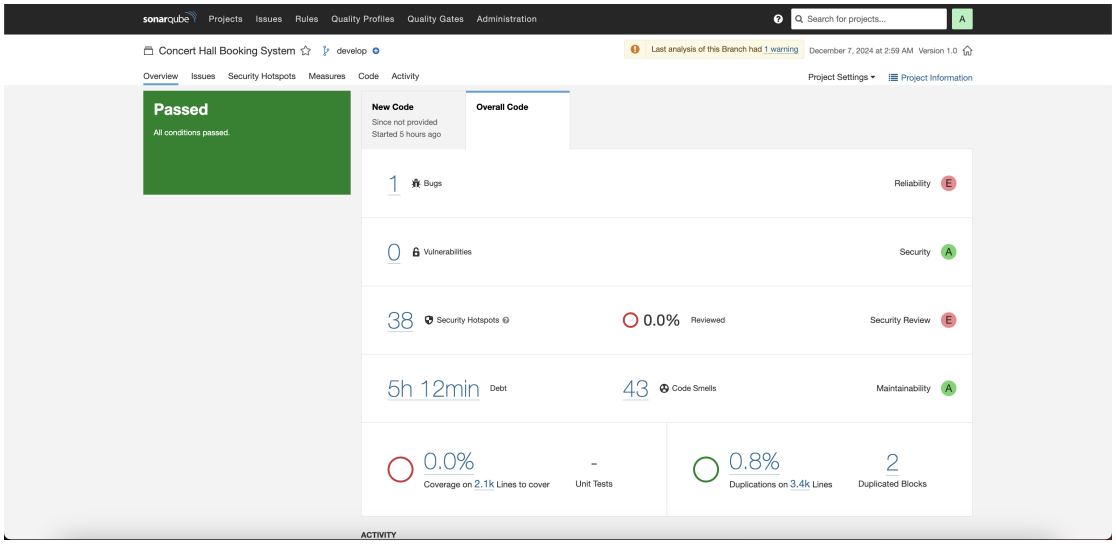


Figure 57: SonarQube Dashboard - Software Metrics before refactoring

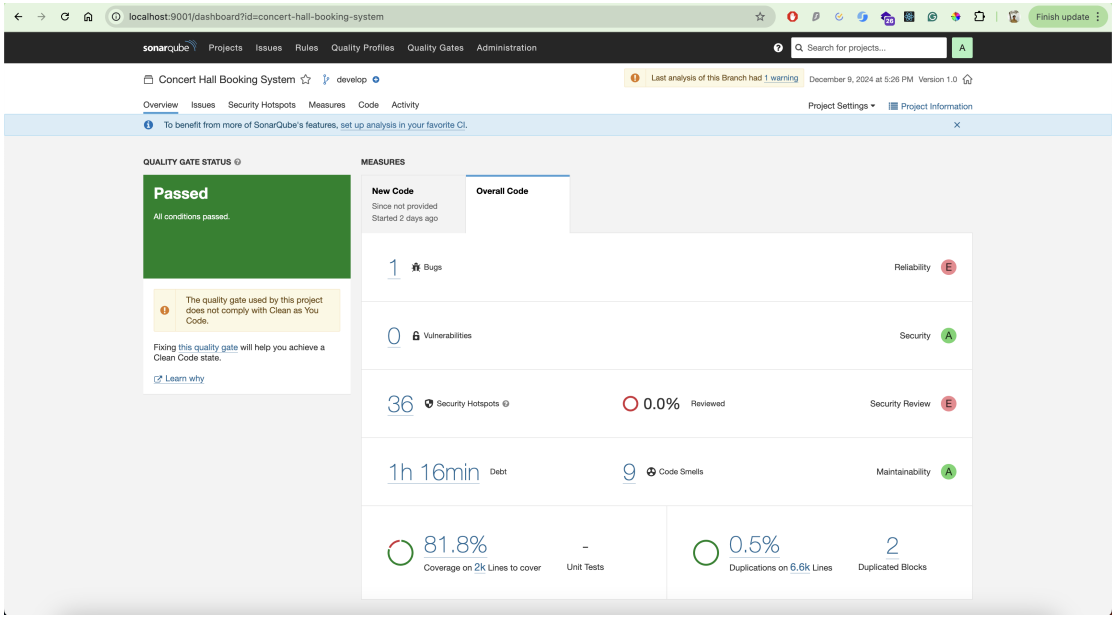


Figure 58: SonarQube Dashboard - Software Metrics after refactoring

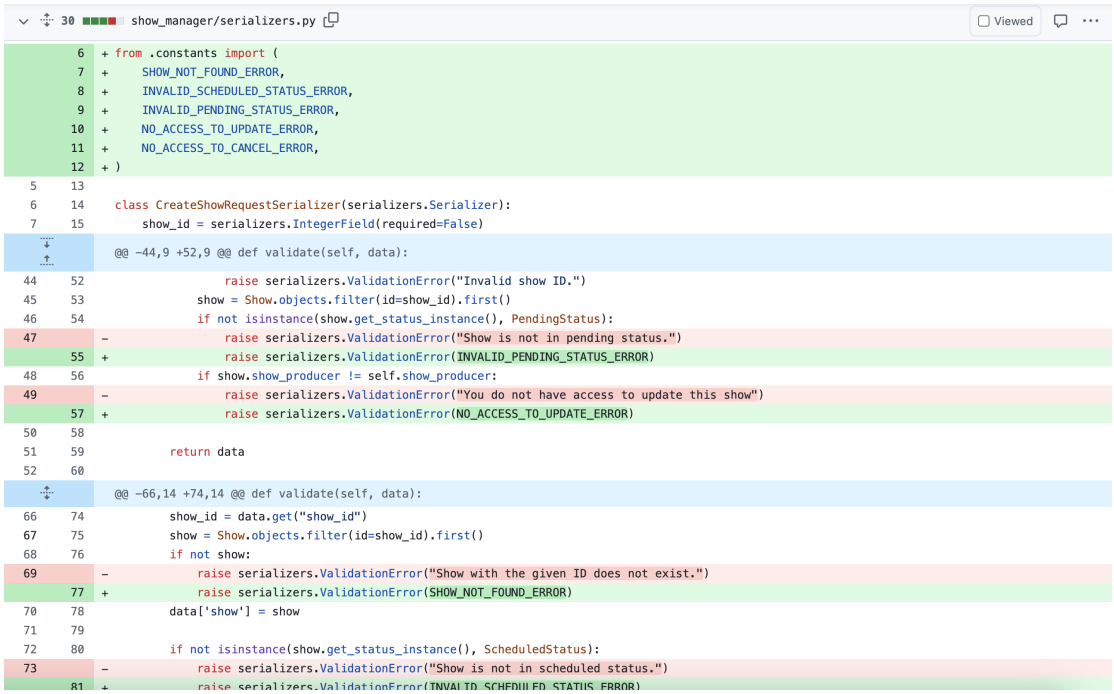


Figure 59: SonarQube - Fix for code smell - Defining a constant instead of repeating the literal in error handling

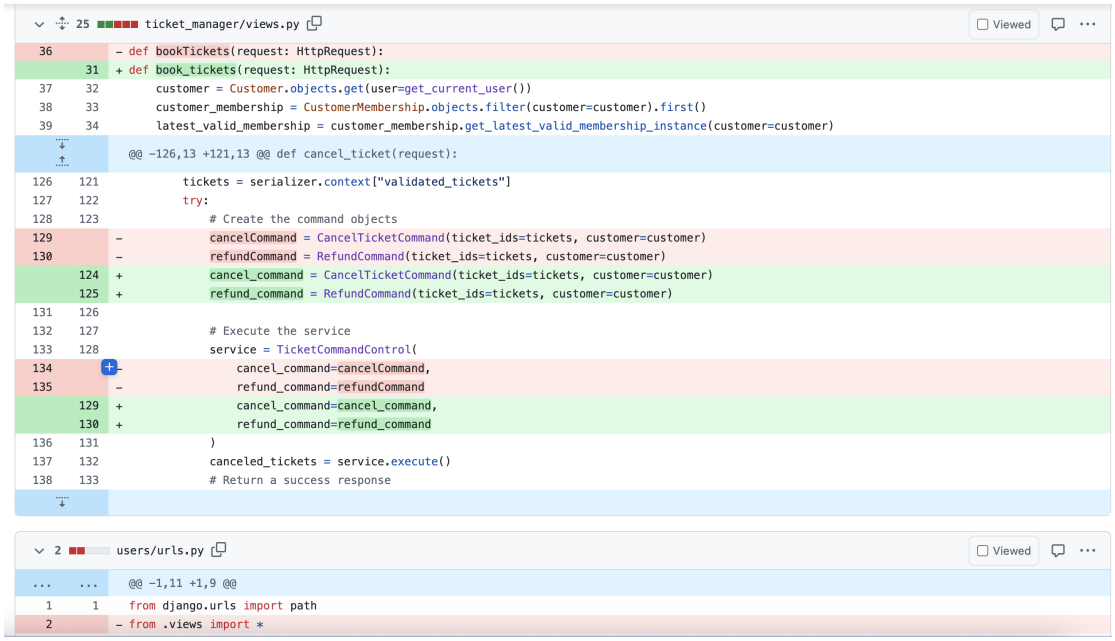


Figure 60: SonarQube - Fix for code smell related to improper naming convention, changed from camelCase to snake_case, and removing unwanted imports

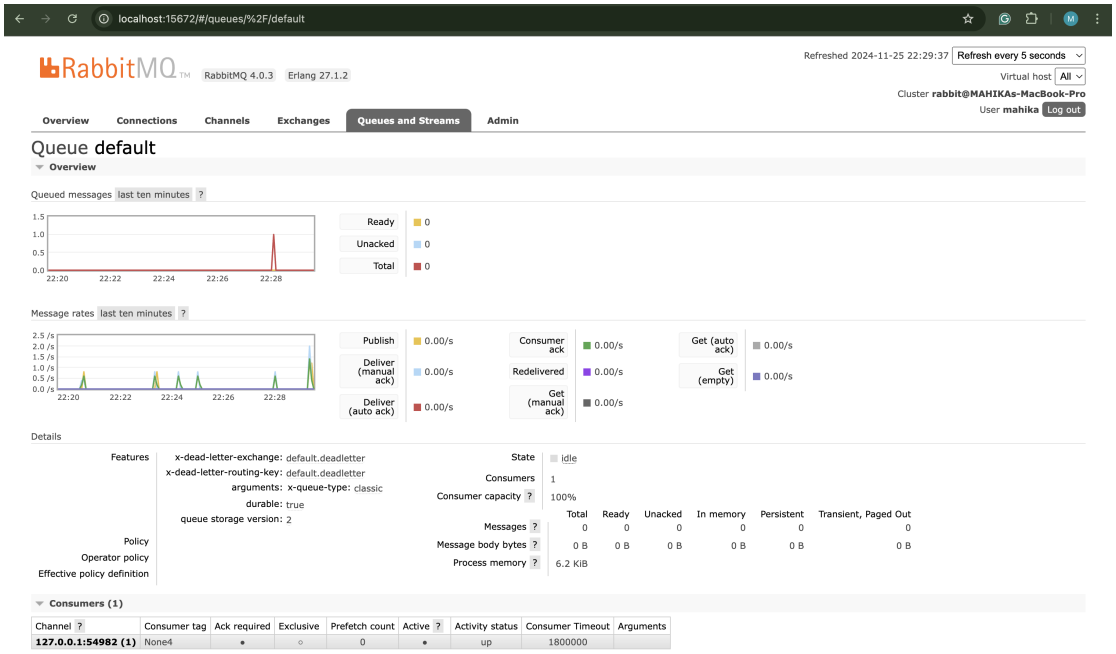


Figure 61: RabbitMQ acts as message broker

configuration needed to integrate Celery in our Django application. As discussed in EDA, Figure 11 defines the task which will be pulled by Celery from the task queue and processed asynchronously.

To handle faults and keep the system available, we use a retry mechanism with exponential backoff to retry processing the task for a maximum of 3 retries [23]. The BaseTaskWithRetry class (Figure 62) holds these configurations which can be extended and re-used by all tasks. Once the maximum retries is reached, the task is Rejected and delivered to the dead letter queue (DLQ) [15] as seen in Figure 64.

Apart from asynchronous processing, we utilise Celery Beat to schedule tasks at periodic intervals. This is to update the status of a show from scheduled to completed once its slot has passed. We create a task to perform this show completion (Figure 65). For setting up this periodic task, we have an init_periodic_tasks command that sets the interval to 1 day and maps the task to this interval (Figure 66). This automates the show completion process and keeps the data clean without any manual intervention.

9.5 Monitoring with Prometheus and Grafana

Monitoring, using Prometheus for metrics collection and Grafana[13] for visualization, tracks key system metrics such as API performance, resource utilization, and error rates in real-time. Prometheus framework collects metrics from various components of the project, such as API performance, database queries, and server health, through exposed endpoints (often /metrics) or instrumented libraries. These metrics include details like response time and request rates. These metrics are scraped periodically by Prometheus and stored in a time-series database,

```python
class BaseTaskWithRetry(Task):
    autoretry_for = (Exception,)  # Exceptions to retry on
    retry_kwargs = {'max_retries': 3}
    retry_backoff = True  # Enables exponential backoff
    retry_backoff_max = 60  # Maximum backoff in seconds
    retry_jitter = True  # Adds randomness to prevent stampedes

    def on_failure(self, exc, task_id, args, kwargs, einfo):
        """
        Handle task failure. If max retries exceeded, reject the task.
        """
        print(f"Task {self.name} exceeded max retries. Rejecting...")
        # Reject the task to send it to DLQ
        raise Reject(requeue=False)
```

Figure 62: Base task retry configuration class

```python
from __future__ import absolute_import, unicode_literals
import os
from celery import Celery

import environ

# Initialize environment reading
env = environ.Env()

# Load environment file if it exists
env_name = os.getenv('ENV', 'development')
env_file = f".env.{env_name}"
if os.path.exists(env_file):
    environ.Env.read_env(env_file)
    print(f"Loading environment from {env_file}")
else:
    print(f"Warning: {env_file} not found. Using default settings.")

# Set the default Django settings module for the 'celery' program.
os.environ.setdefault('DJANGO_SETTINGS_MODULE', 'config.settings')

app = Celery('config')

# Load task modules from all registered Django app configs.
app.config_from_object('django.conf:settings', namespace='CELERY')

# Auto-discover tasks in installed apps
app.autodiscover_tasks()

@app.task(bind=True)
def debug_task(self):
    print(f'Request: {self.request!r}')
```

Figure 63: Configuring Celery in Django application

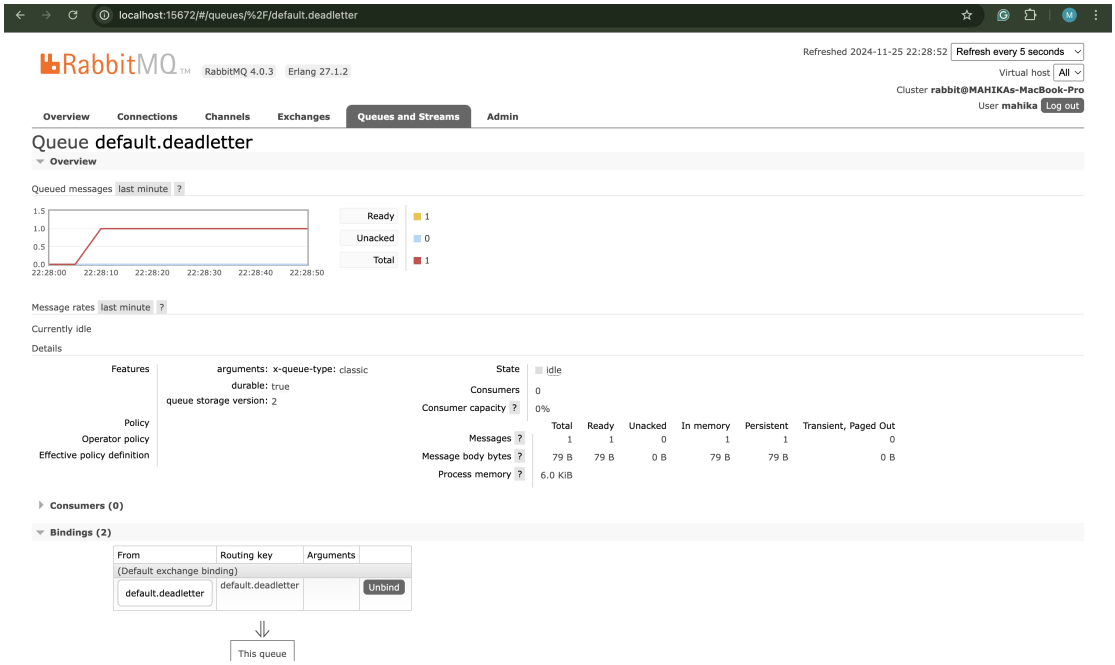


Figure 64: Dead letter queue to hold failed tasks

```python
from celery import shared_task
from django.utils.timezone import now
from datetime import timedelta
from show_manager.models import Show, ShowStatusEnum

@shared_task      Mahika Jaguste, 7 days ago • install django ce
def mark_shows_as_completed():
    print("Running mark_shows_as_completed cron job")
    yesterday = (now() - timedelta(1)).strftime('%Y-%m-%d')
    shows_to_complete = Show.objects.filter(
        slot__date__lte=yesterday,
        status = ShowStatusEnum.SCHEDULED.name
    )

    for show in shows_to_complete:
        show.complete()
        print(f"Show '{show.name}' marked as completed.")
```

Figure 65: Task to mark show as completed

```python
from django.core.management.base import BaseCommand
from django_celery_beat.models import PeriodicTask, IntervalSchedule
import json
from django.utils import timezone

Mahika Jaguste, 6 days ago | 1 author (Mahika Jaguste)
class Command(BaseCommand):
    help = "Initialize periodic tasks"

    def handle(self, *args, **kwargs):
        # Create or fetch an interval schedule
        schedule, created = IntervalSchedule.objects.get_or_create(
            every=1,  # Every 1 day
            period=IntervalSchedule.DAYS,
        )

        # Define or update a periodic task
        task_name = "mark_shows_as_completed"
        PeriodicTask.objects.get_or_create(
            name=task_name,
            defaults={
                "interval": schedule,
                "task": "show_manager.tasks.mark_shows_as_completed",  # Full path to the task
                "start_time": timezone.now(),
                "args": json.dumps([]),
                "kwargs": json.dumps({}),
            },
        )

        self.stdout.write(self.style.SUCCESS(f"Periodic task '{task_name}' initialized."))
```

Figure 66: Command to map task with its interval (every day)

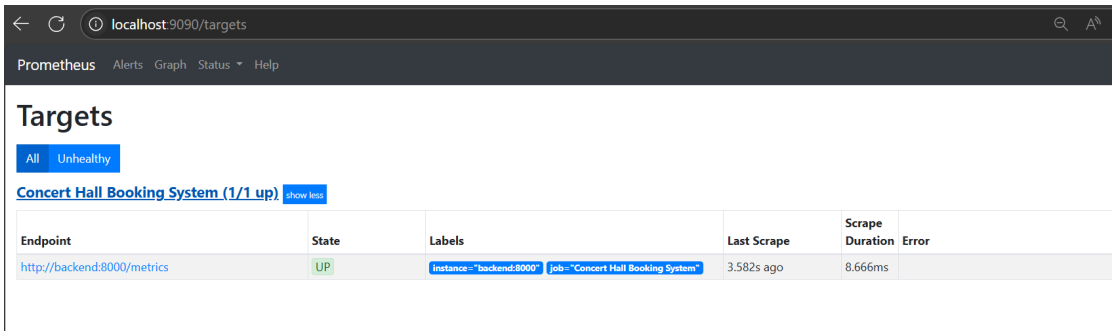


Figure 67: Prometheus

allowing custom alert rules to be defined. Grafana acts as the visualization layer, connecting to Prometheus as a data source. It displays the collected metrics on customizable dashboards, making it easier to understand trends, pinpoint issues, and gain insights into the system's behavior. This setup improves system availability by enabling quick detection of issues, proactive alerting for critical failures, and faster resolution of problems. It ensures reliability, supports capacity planning and helps maintain seamless operations for end-users. Figure 68 gives a screenshot of the dashboard setup for this application.

9.6 Logging

We have implemented a logging mechanism for our Django application, ensuring that different components of the application log messages appropriately. We have implemented a file based logger [22] as shown in Figure 69, which will capture the different logs from the system. It also allows the developer to create custom logs

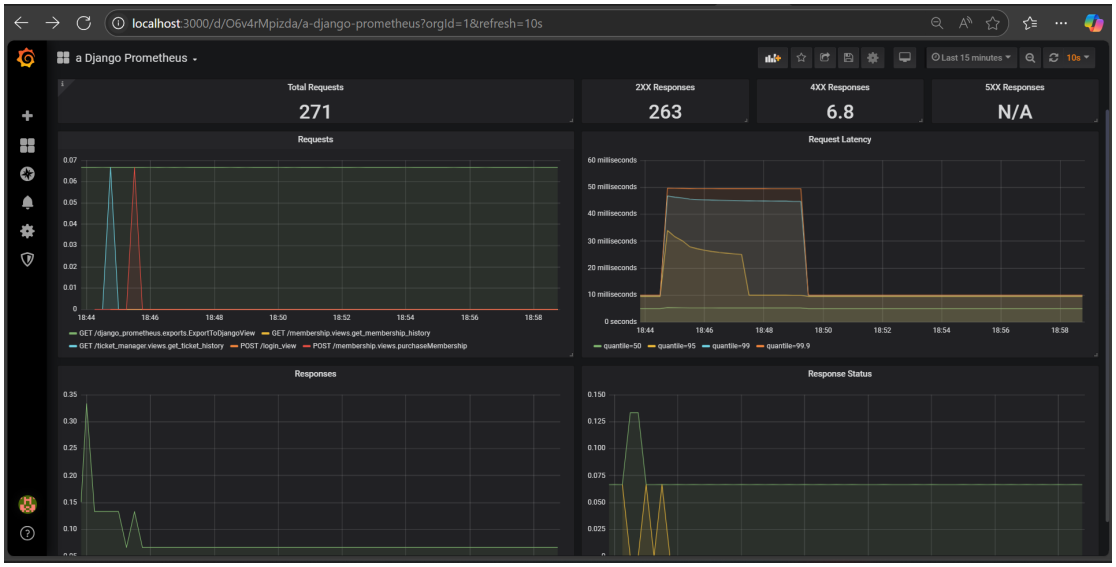


Figure 68: Garfana Dashboard

```
2024-12-07 07:03:04,392 - django.utils.autoreload - INFO - /Users/anjana/Desktop/Semester 1/Software Design/Project/temp/con
2024-12-07 07:03:04,954 - django.utils.autoreload - INFO - Watching for file changes with StatReloader
2024-12-08 19:44:46,174 - django.utils.autoreload - INFO - Watching for file changes with StatReloader
2024-12-08 19:44:56,426 - concert_hall_booking - ERROR - {'email': [ErrorDetail(string='A user with this email already exist
2024-12-08 19:44:56,427 - django.request - WARNING - Bad Request: /users/customer/register
2024-12-08 19:44:56,428 - django.server - WARNING - "POST /users/customer/register HTTP/1.1" 400 110
2024-12-08 19:45:02,092 - concert_hall_booking - INFO - User with ID customer111@gmail.com signed in
2024-12-08 19:45:02,093 - django.server - INFO - "POST /users/login HTTP/1.1" 200 122
2024-12-08 19:45:11,059 - concert_hall_booking - INFO - Admin logged in
2024-12-08 19:45:11,060 - django.server - INFO - "POST /users/admin/login HTTP/1.1" 200 80
2024-12-08 19:45:14,180 - concert_hall_booking - INFO - User with ID max1@gmail.com signed in
2024-12-08 19:45:14,180 - django.server - INFO - "POST //users/login HTTP/1.1" 200 119
2024-12-08 19:45:18,057 - django.utils.autoreload - INFO - /Users/anjana/Desktop/Semester 1/Software Design/Project/temp/use
2024-12-08 19:45:18,613 - django.utils.autoreload - INFO - Watching for file changes with StatReloader
```

Figure 69: Logs

for the context of different types like info, error, debug and warning. The logging configurations are implemented in settings.py as in Figure 70. The formatters section defines how log messages should be formatted, with a detailed formatter that includes the timestamp, logger name, log level, and message. The handlers section specifies where the log messages should go; in this case, a file handler is defined to write log messages to logs/application.log with the detailed formatter and a log level of DEBUG.

The loggers section configures different loggers for various parts of the application. The django logger handles general Django logs and the django.db.backends logger is configured to handle database backend logs. Finally, a custom logger named concert_hall_booking is set up for the application, logging messages at the DEBUG level. This setup ensures that different parts of the application log messages at appropriate levels, helping developers monitor and debug the application effectively. The logs are scrapped by Loki into the Grafana dashboard as well[14].

```python
LOGGING = {
    'version': 1,
    'disable_existing_loggers': False,
    'formatters': {
        'detailed': {
            'format': '{asctime} - {name} - {levelname} - {message}',
            'style': '{',
        },
    },
    'handlers': {
        'file': {
            'level': 'DEBUG',  # Logs DEBUG, INFO, and ERROR
            'class': 'logging.FileHandler',
            'filename': 'logs/application.log',
            'formatter': 'detailed',
        },
    },
    'loggers': {
        # General Django logger
        'django': {
            'handlers': ['file'],
            'level': 'INFO',  # Only log INFO and above
            'propagate': True,
        },
        # Suppress excessive debug messages from database backends
        'django.db.backends': {
            'handlers': ['file'],
            'level': 'WARNING',
            'propagate': False,
        },
        # Custom logger for your application
        'concert_hall_booking': {
            'handlers': ['file'],
            'level': 'DEBUG',
        },
    },
}
```

Figure 70: Logging configurations in settings.py

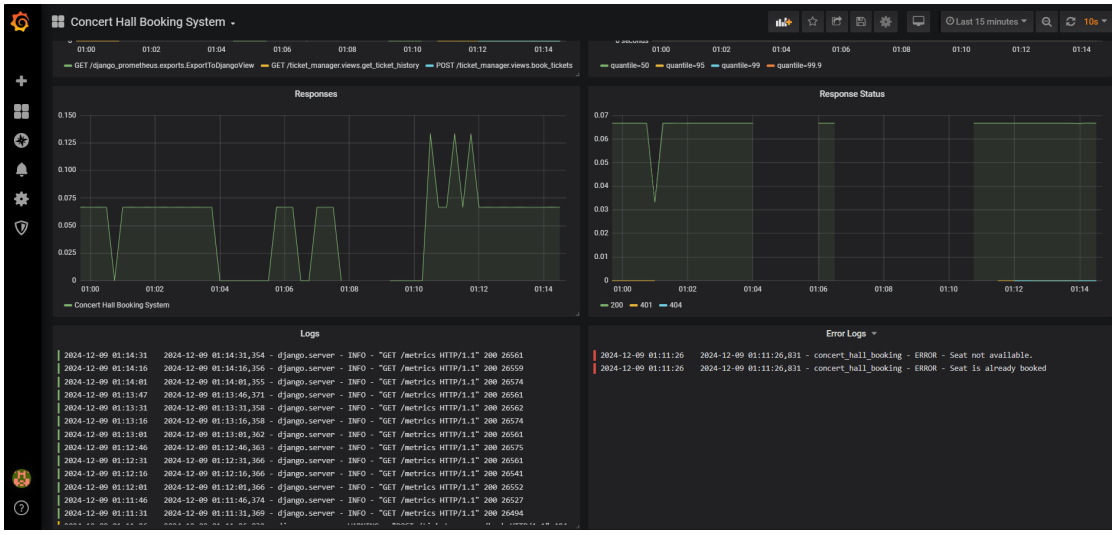


Figure 71: Logging dashboard in Grafana

```
# Use an official Python runtime as a parent image
FROM python:3.12

# Set environment variables
ENV PYTHONDONTWRITEBYTECODE 1
ENV PYTHONUNBUFFERED 1

# Set the working directory
WORKDIR /app

# Install dependencies
COPY requirements.txt /app/
RUN pip install -r requirements.txt

# Copy the project code into the container
COPY . /app/      Mahika Jaguste, last month • enviro
```

Figure 72: Dockerfile to build Django application server

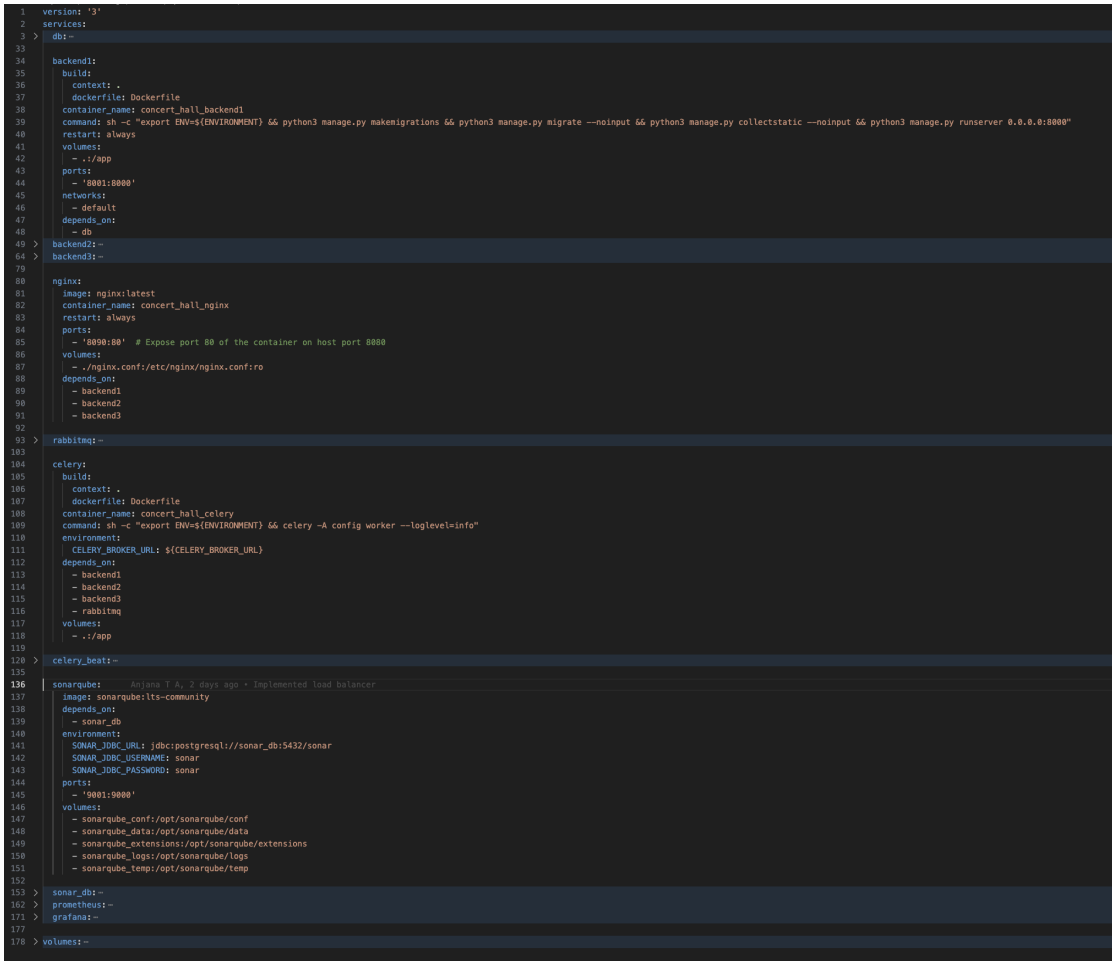
9.7 Docker

With global teams and consumer bases, the portability of modern applications is extremely important. We have dockerised our application to ensure it runs on any operating system [8]. It will also help to run multiple instances of these lightweight containers when we want to scale our system. The Dockerfile (Figure 72) is used to show how to install dependencies and build our Django application server. The Docker compose file (Figure 73) is used to list all the services our application will be using, which are MySQL database, RabbitMQ task queue, Celery worker and Celery Beat for cron jobs. Docker uses these files to pull the required images for these services and compose our application and get it running seamlessly.

9.8 Load Balancing via Nginx

Nginx allows load balancing through distribution of incoming requests among the available backend servers for maximum resource usage and reliability of the system. In out implementation, Nginx works as a reverse proxy whereby it receives the client's request and forwards it to one of the backend servers (Figure 74). This technique [18] makes it possible for each server to receive effortless traffic since it sends it to the one with the least number of connections in the network. By balancing requests this way, the application will be able to serve more concurrent users and thus better performance and user experience would be achieved.

In addition, this setup improves fault tolerance. When one of the backend servers is off, there's no need for somebody to keep sending requests to it, Nginx will do that automatically and will go on sending requests to the healthy backend servers. This means that there's continuity of service. Moreover, the settings provide for optimal timeouts for connecting with backends and communicating with them so as to reduce the swapping time ensuring that the application is responsive.



```yaml
version: '3'
services:
  db: ⋯

  backend1:
    build:
      context: .
      dockerfile: Dockerfile
    container_name: concert_hall_backend1
    command: sh -c "export ENV=${ENVIRONMENT} && python3 manage.py makemigrations && python3 manage.py migrate --noinput && python3 manage.py collectstatic --noinput && python3 manage.py runserver 0.0.0.0:8000"
    restart: always
    volumes:
      - .:/app
    ports:
      - '8001:8000'
    networks:
      - default
    depends_on:
      - db
  backend2: ⋯
  backend3: ⋯

  nginx:
    image: nginx:latest
    container_name: concert_hall_nginx
    restart: always
    ports:
      - '8090:80'  # Expose port 80 of the container on host port 8080
    volumes:
      - ./nginx.conf:/etc/nginx/nginx.conf:ro
    depends_on:
      - backend1
      - backend2
      - backend3

  rabbitmq: ⋯

  celery:
    build:
      context: .
      dockerfile: Dockerfile
    container_name: concert_hall_celery
    command: sh -c "export ENV=${ENVIRONMENT} && celery -A config worker --loglevel=info"
    environment:
      CELERY_BROKER_URL: ${CELERY_BROKER_URL}
    depends_on:
      - backend1
      - backend2
      - backend3
      - rabbitmq
    volumes:
      - .:/app

  celery_beat: ⋯

  sonarqube:      Anjana T A, 2 days ago • Implemented load balancer
    image: sonarqube:lts-community
    depends_on:
      - sonar_db
    environment:
      SONAR_JDBC_URL: jdbc:postgresql://sonar_db:5432/sonar
      SONAR_JDBC_USERNAME: sonar
      SONAR_JDBC_PASSWORD: sonar
    ports:
      - '9001:9000'
    volumes:
      - sonarqube_conf:/opt/sonarqube/conf
      - sonarqube_data:/opt/sonarqube/data
      - sonarqube_extensions:/opt/sonarqube/extensions
      - sonarqube_logs:/opt/sonarqube/logs
      - sonarqube_temp:/opt/sonarqube/temp

  sonar_db: ⋯
  prometheus: ⋯
  grafana: ⋯

volumes: ⋯
```

Figure 73: Docker Compose lists all services

```yaml
backend1:
  build:
    context: .
    dockerfile: Dockerfile
  container_name: concert_hall_backend1
  command: sh -c "export ENV=${ENVIRONMENT} && python3 manage.py makemigrations && python3 manage.py migra
  restart: always
  volumes:
    - .:/app
  ports:
    - '8001:8000'
  networks:
    - default
  depends_on:
    - db
backend2:
  build:
    context: .
    dockerfile: Dockerfile
  container_name: concert_hall_backend2
  command: sh -c "export ENV=${ENVIRONMENT} && python3 manage.py makemigrations && python3 manage.py migra
  restart: always
  volumes:
    - .:/app
  ports:
    - '8002:8000'
  networks:
    - default
  depends_on:
    - db
backend3:
  build:
    context: .
    dockerfile: Dockerfile
  container_name: concert_hall_backend3
  command: sh -c "export ENV=${ENVIRONMENT} && python3 manage.py makemigrations && python3 manage.py migra
  restart: always
  volumes:
    - .:/app
  ports:
    - '8003:8000'
  networks:
    - default
  depends_on:
    - db

nginx:
  image: nginx:latest
  container_name: concert_hall_nginx
  restart: always
  ports:
    - '8090:80'  # Expose port 80 of the container on host port 8080
  volumes:
    - ./nginx.conf:/etc/nginx/nginx.conf:ro
  depends_on:
    - backend1
    - backend2
    - backend3
```

Figure 74: Nginx configuration with 3 backends

10 Design Blueprints

The following diagrams were created using [19] and [5]. This tool offers an easy-to-use interface and lets us collaborate remotely to create and ideate on UML diagrams.

10.1 Design-time Package Diagram

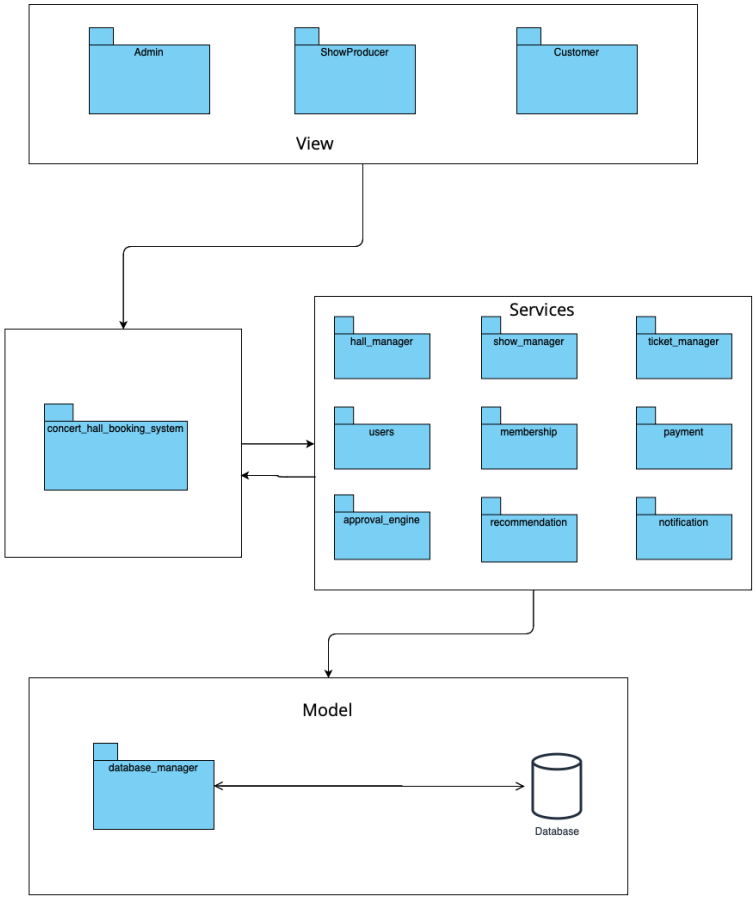


Figure 75: Design-time Package Diagram

10.2 Design-Time Class Diagram

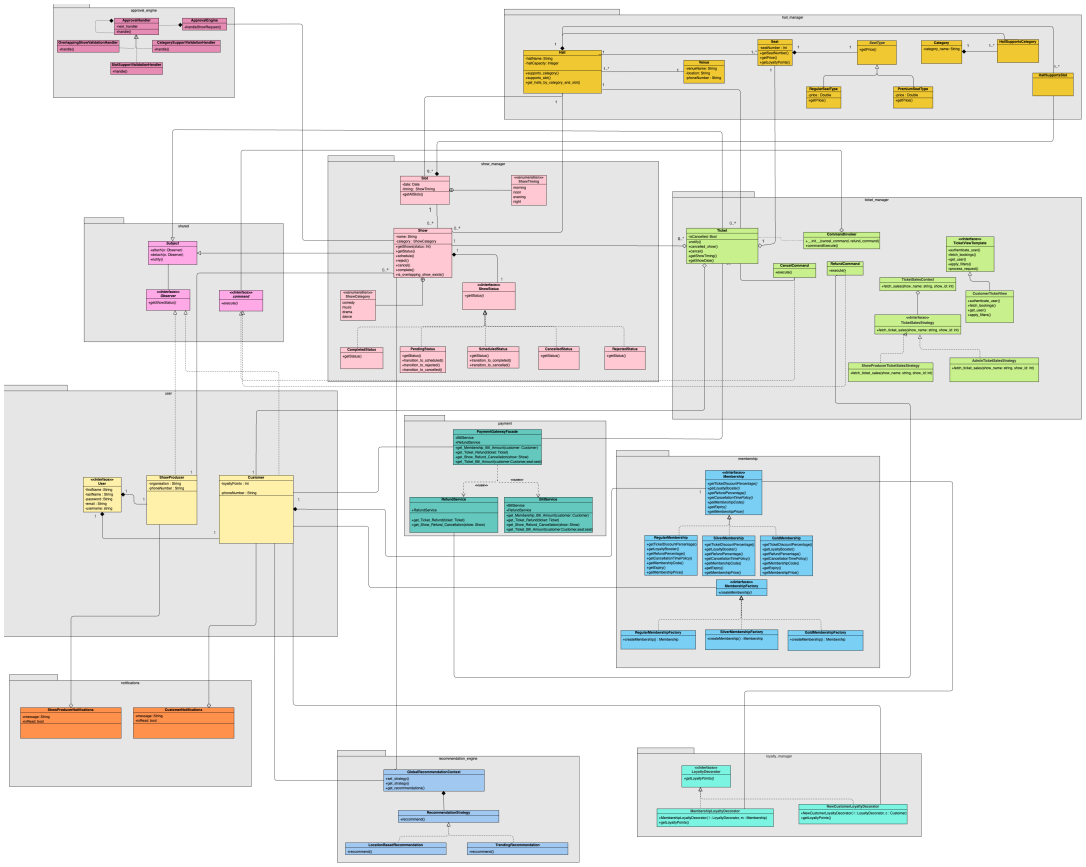


Figure 76: Design-Time Class Diagram

10.3 Design-Time Sequence Diagram

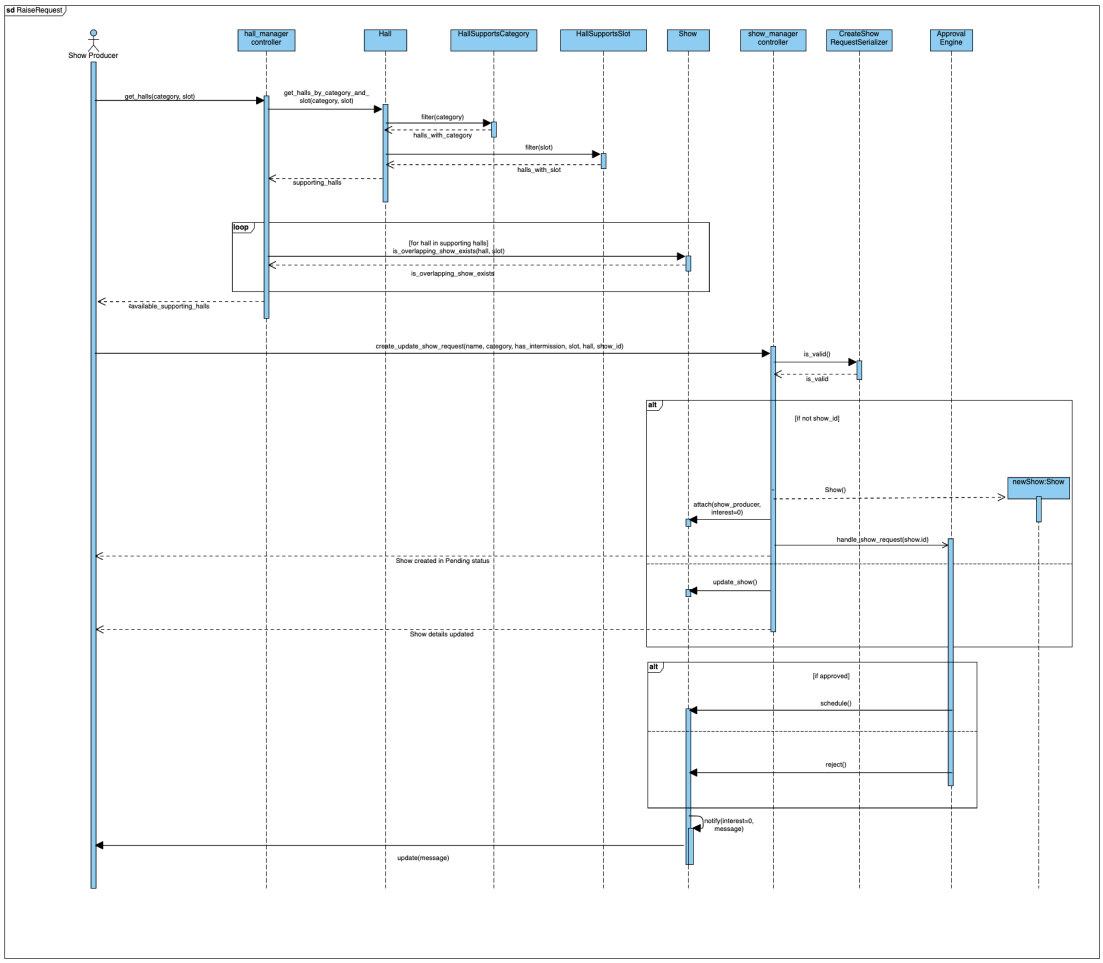


Figure 77: Design-Time Sequence Diagram for Raise Show Request

10.4 Component Diagram

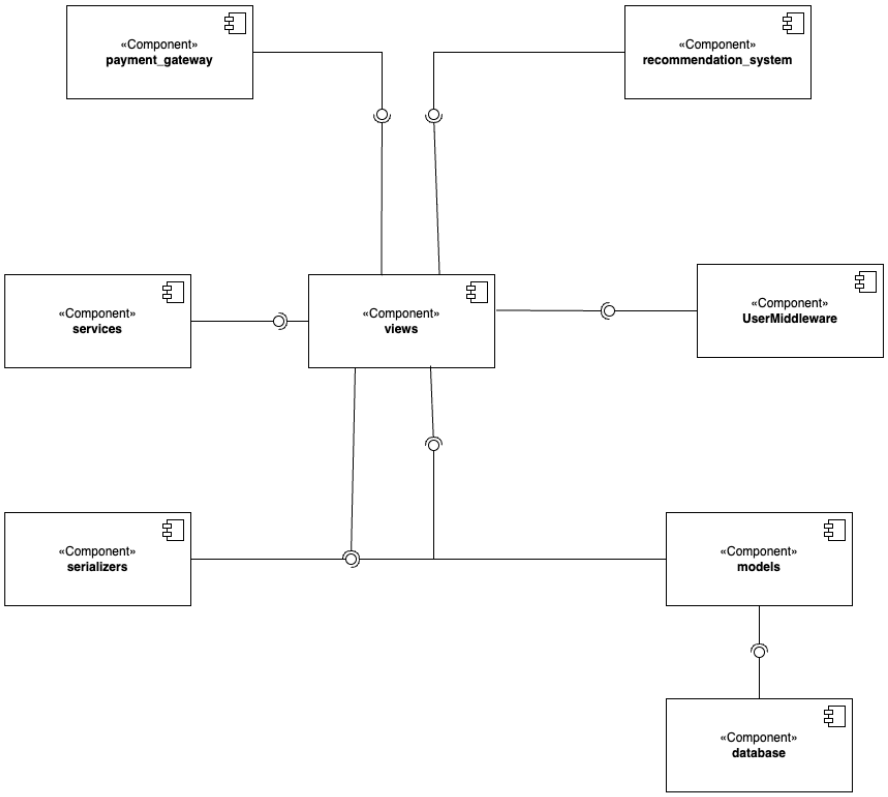


Figure 78: Component Diagram

10.5 Deployment Diagram

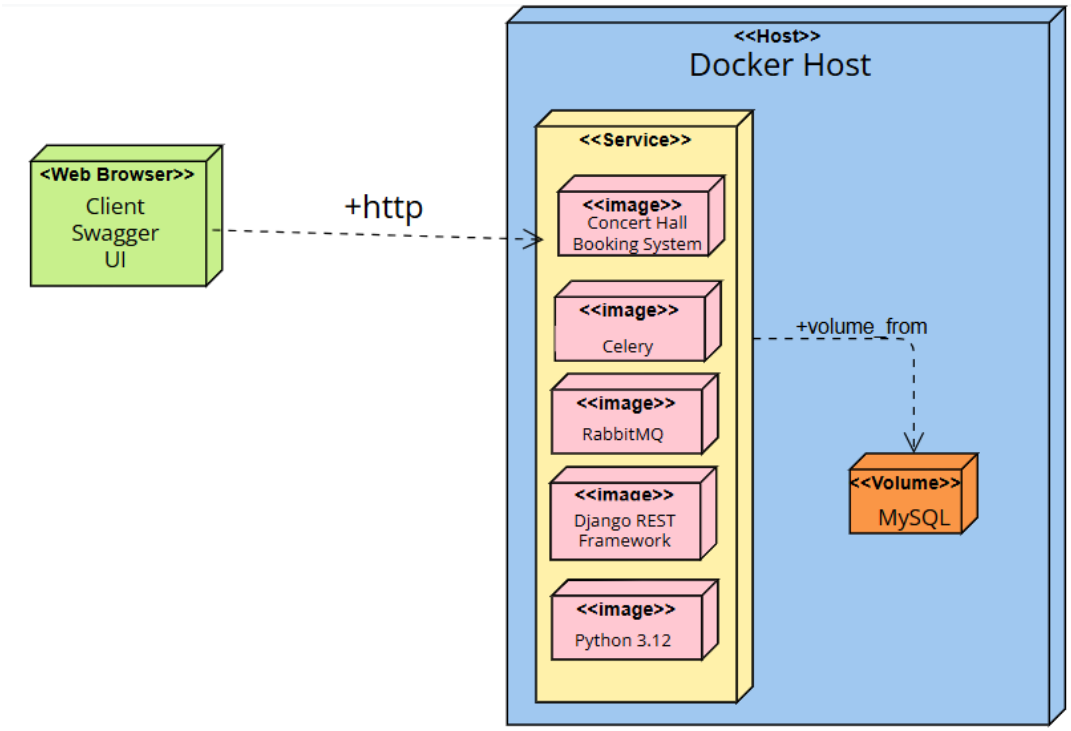


Figure 79: Deployment Diagram

11 Critique

One of the most challenging and engaging parts of the project was determining where and how to implement design patterns effectively. We had to identify areas to apply the Command, Decorator, and other patterns which could improve the code quality and modularity required. Codebase cleanliness and maintainability were significantly improved due to our design pattern implementations.

The team's unfamiliarity with Django introduced a learning curve. It took us significant effort to set up the server application [17] and understand the patterns and best practices of using Django. We leveraged the Django REST framework along with serializers for efficient data validation and transformation. The serializers in a way resemble the preconditions of the API endpoints. Swagger proved to be an efficient and easy-to-implement documentation of our system endpoints. With minimal overhead, its integration provides a neat interface for communication with other stakeholders.

The UML diagrams, especially the class diagram were invaluable to clarify ambiguities in our mental model and provided a shared visual reference for discussing and refining the design. We were able to document relationships, dependencies and workflows systematically at a very early stage in the project which avoided

any significant logic gaps or inconsistencies at a later stage. Each developer heavily referred to the diagrams to understand the requirements on their end as well as to depend on code fragments developed by other members. These diagrams will also act as documentation for further developing this project. The diagrams recovered after implementation are much more detailed and cover the gaps in the analysis diagrams. However, there are two flaws with the UML diagrams. Firstly, in our planning, we had allocated specific weeks to design the diagrams, and they took up significant time upfront which may not be possible in an industrial setting. Secondly, while they provide clarity on the part of the code they represent, it is not possible to cover all aspects of the codebase via diagrams. The code is ultimately the best source of documentation, however using diagrams for the most critical aspects can be a good compromise.

While Django's architecture is monolithic in nature which is considered a legacy in modern industry, we tried to overcome this limitation by experimenting with event-driven architecture for certain asynchronous parts of the system. This approach allowed us to decouple certain components and keep the architecture flexible to support evolution and maintainability.

One shortcoming was the lack of test-driven development (TDD). Tests were added after the initial implementation rather than being part of the development cycle. For better quality assurance and test coverage, the tests could have been integrated during the development phase.

12 Reflections

Adarsh's Experience
This project was a valuable learning experience for me in multiple ways. It was satisfying to see all these design patterns we learned in class come to life in our solutions. As the technical lead, I took on responsibilities like identifying best practices, reviewing pull requests, and thinking of ways we can incorporate various design patterns, which was an exciting challenge. One of the highlights was working with CI/CD pipelines and tools like Jenkins, Docker, and SonarQube. It was my first time delving into these technologies at such depth, and it gave me a practical understanding of streamlining development workflows and maintaining code quality. Collaborating with my team to implement these patterns also reinforced the importance of clear communication and teamwork in overcoming challenges and meeting deadlines. This project not only strengthened my technical skills but also honed my leadership abilities.

Anushree's Experience
Working on this project was a rewarding experience, as it allowed me to move from my previous work with low-code platforms into a more hands-on, code-driven development process. I got to dive deep into design patterns and understand how they make code more maintainable and reusable, showing me how cool and powerful they can be when properly implemented. I also had the chance to learn new technologies like Django, REST APIs, and Git giving me a better grasp of the

tools that are commonly used in the industry.

While integrating tools like Prometheus for monitoring and Swagger for API documentation was a bit challenging, it was also a great learning opportunity. It pushed me to think critically and enhanced my ability to bring different technologies together. This project really helped me to move from the simplicity of low-code platforms to designing and coding a complex application from scratch teaching me the value of structured design, adaptability, and writing clear, understandable code.

Anjana's Experience
Coming from a testing background to development space have been challenging and quite meaningful. It gave me the opportunity to transition from focusing on quality assurance to actively designing and coding application. I got to dive into design patterns and see firsthand how they help make code more maintainable and reusable. Interacting with new technologies like Django, Docker, Nginx have been rewarding. Participating in authentication, load balancing and logging gave me more hands-on experience with essential aspects of application development. This was a great learning curve, as it helped me understand the importance of security in real-world applications. Implementing the different design patterns based on the necessity improved my analytical skills.

Mahika's Experience
There are 2 aspects of this project that I enjoyed the most. Firstly, thinking of ways in which we could integrate design patterns into our solutions was a great learning experience. It was fun to get hands-on and implement the patterns that we talked about in class. I think this will help me in my professional life to think of clean neat solutions. The other aspect was experimenting with event-driven architecture and integrating Celery and RabbitMQ. I got to go in-depth on how these tools help in asynchronous flows. I got to consider non-functional attributes like fault tolerance by introducing a retry mechanism and dead-letter-queues for the failed messages. These tools and architecture are an industry standard and using them in a project was a good way to get introduced to them. I feel as a team we paced the project well so that we could keep our deadlines.

Link to GitHub repository: https://github.com/CS5721-VonNeumann/concert-hall-booking-system

References

[1] Nur: The She Code Africa Blog Ada Nduka Oyom. *Understanding the MVC pattern in Django*. URL: `https://medium.com/shecodeafrica/understanding-the-mvc-pattern-in-django-edda05b9f43f`.

[2] Kent Beck et al. *Manifesto for Agile Software Development*. 2001. URL: `https://agilemanifesto.org`.

[3] Saurabh Chodvadiya. *Creating Interactive API Documentation with Swagger UI in Django*. Accessed: 2024-12-09. 2023. URL: `https://medium.`

com/@chodvadiyasaurabh/creating-interactive-api-documentation-with-swagger-ui-in-django-53fa9e9898dc.

[4] J.J. Collins. "Lecture slides". University of Limerick.

[5] Diagrams.net. *Draw.io - Diagram Software and Flowchart Maker*. Accessed: 2024-12-08. 2024. URL: https://www.drawio.com.

[6] Docker. *Docker Documentation*. 2013. URL: https://docs.djangoproject.com/en/5.1/.

[7] Django Software Foundation. *Django Documentation*. 2005. URL: https://docs.djangoproject.com/en/5.1/.

[8] Akshat Gadodia. *Dockerizing a Django and MySQL Application: A Step-by-Step Guide*. Accessed: 2024-12-5. URL: https://medium.com/django-unleashed/django-project-structure-a-comprehensive-guide-4b2ddbf2b6b8.

[9] Geek for geeks. *Use Case Diagram – Unified Modeling Language (UML)*. URL: https://www.geeksforgeeks.org/use-case-diagram/.

[10] Jenkins. *Jenkins Documentation*. URL: https://www.jenkins.io/doc/pipeline/tour/getting-started/.

[11] Django Unleashed Julio Cesar Siles Perez. *Django Project Structure: A Comprehensive Guide*. URL: https://medium.com/django-unleashed/django-project-structure-a-comprehensive-guide-4b2ddbf2b6b8.

[12] P. Clements L. Bass and R. Kazman. *Software architecture in practice, 4th Edition*. Addison-Wesley, 2021.

[13] Grafana Labs. *Django Prometheus Dashboard*. Accessed: 2024-12-09. 2023. URL: https://grafana.com/grafana/dashboards/9528-django-prometheus/.

[14] Grafana Labs. *Loki Datasource Documentation*. Accessed: 2024-12-09. 2023. URL: https://grafana.com/docs/grafana/latest/datasources/loki/.

[15] Hengfeng Li. *How to create a dead letter queue in Celery + RabbitMQ*. Accessed: 2024-12-5. URL: https://medium.com/@hengfeng/how-to-create-a-dead-letter-queue-in-celery-rabbitmq-401b17c72cd3.

[16] Miguel Morejón. *Deployment Diagram for Docker and Django*. Accessed: 2024-12-09. 2023. URL: https://mmorejon.io/en/blog/deployment-diagram-docker-django/.

[17] Programming with Mosh. YouTube. *Django tutorial for beginners - build powerful backends*. Accessed: 2024-12-5. URL: https://www.youtube.com/watch?v=rHux0gMZ3Eg.

[18] Nandagopal. *Deploy Django Project with Nginx, Gunicorn, Docker to Production*. Accessed: 2024-12-08. 2023. URL: https://medium.com/@nandagopal05/deploy-django-project-with-nginx-gunicorn-docker-to-production-b4368a2fefff.

[19] Visual Paradigm. *Ideal modeling & diagramming tool for Agile team collaboration*. Accessed: 2024-12-5. URL: https://www.visual-paradigm.com/.

[20] PyTest. *pytest Documentation*. URL: https://docs.pytest.org/en/stable/.

[21] Sipios. *Get Started to Monitor Your Django Application with Prometheus & Grafana in 10 Minutes*. Accessed: 2024-12-09. 2023. URL: https://medium.com/sipios/get-started-to-monitor-your-django-application-with-prometheus-grafana-in-10-minutes-dac9c0fdcf58.

[22] Mahdi Torkashvand. *A Comprehensive Guide to Logging in Django*. Accessed: 2024-12-08. 2023. URL: https://medium.com/@torkashvand/a-comprehensive-guide-to-logging-in-django-e041f311bcb7.

[23] Michael Yin. *Automatically Retrying Failed Celery Tasks*. Accessed: 2024-12-5. URL: https://testdriven.io/blog/retrying-failed-celery-tasks/.