**Problem statement**

XYZ wants to build an online movie ticket booking platform that caters to both B2B (theatre partners) and B2C (end customers) clients.

Key goals it wants accomplished as part of its solution:

* Enable theatre partners to onboard their theatres over this platform and get access to a bigger customer base while going digital.

* Enable end customers to browse the platform to get access to movies across different cities, languages, and genres, as well as book tickets in advance with a seamless experience.

**Technologies recommended**

* Language -Java and other add-on languages
* Frameworks- Any
* Database - Any
* Integration technologies- Any
* Cloud technologies- Any
* Preferred editor to build and present solution

*PS:* ***The given exercise will help you get prepared for technical discussion and demonstrate your current understanding on key architectural artifacts. It’s expected that you may not all sections hence can ignore them.***

**Evaluation criteria**

* Code artifacts (**APIs Contract, Design Patterns,** Any one Scenario Implementation)
* Design principles to address functional requirement and non-functional requirement
* DB & Data model
* Platform solutions detailing
* Solution completeness, presentation, and discussion.
* Solution coverage uniqueness and extensibility.

Note: Incomplete solution component would be discussed during discussion round. All sections are not mandatory.

*You can skip solution areas that you are not comfortable by making a note of it.*

**Functional features to implement** Good to have - Code Implementation (Read scenario)):

**Anyone of the following read scenarios:** (Only Service Implementation needed/ No UI required)

* Browse theatres currently running the show (movie selected) in the town, including show timing by a chosen date
* Booking platform offers in selected cities and theatres
  + 50% discount on the third ticket
  + Tickets booked for the afternoon show get a 20% discount

**Candidate Solution:**

**Please see the GitHub link below:**

[**https://github.com/tanwargaurav20/BMS**](https://github.com/tanwargaurav20/BMS)

* There is an API (REST API) which uses JPA backed by in-memory H2 database, as below.

GET /theatre/criteria?locationId=1&movieTitle=RRR&date=&date=2019-12-31

It returns all the theatres running the show/movie at the required location (depicted by locationId) and at /or after the required date.

* POST /bookings API while booking the tickets fetches the discount percentage from the discount table by discount code, location & theatres and applies it dynamically.

**Panel Feedback:**

**Select Outcome: Low/Med/High:**

**Anyone of the following write scenarios:** Good to have - Code Implementation (write scenario):

* Book movie tickets by selecting a theatre, timing, and preferred seats for the day
* Theatres can create, update, and delete shows for the day.
* Bulk booking and cancellation
* Theatres can allocate seat inventory and update them for the show

**Candidate Solution:**

**Please see the GitHub link below:**

[**https://github.com/tanwargaurav20/BMS**](https://github.com/tanwargaurav20/BMS)

When the user searches for a movie the following API returns the theatres with the relevant details.

GET /theatre/criteria?locationId=1&movieTitle=RRR&date=&date=2019-12-31

The selected movie/seat can be booked by sending the screenId, movieId, and array of seat IDs to the REST API as below:

POST /bookings

Request payload:

{

    "screenId": "1",

    "movieId": "1",

    "seats": [{"id": 1}, {"id":2}]

}

The above API will reserve the seats for 15 mins and the user needs to proceed with the payment by then.

The following API is used to make the payment and mark the booking as completed.

PUT /bookings/{bookingId}/pay

* For the other scenarios, the services & controllers can be updated/modified following the pattern in existing controllers & services.

**Panel Feedback:**

**Select Outcome: Low/Med/High:**

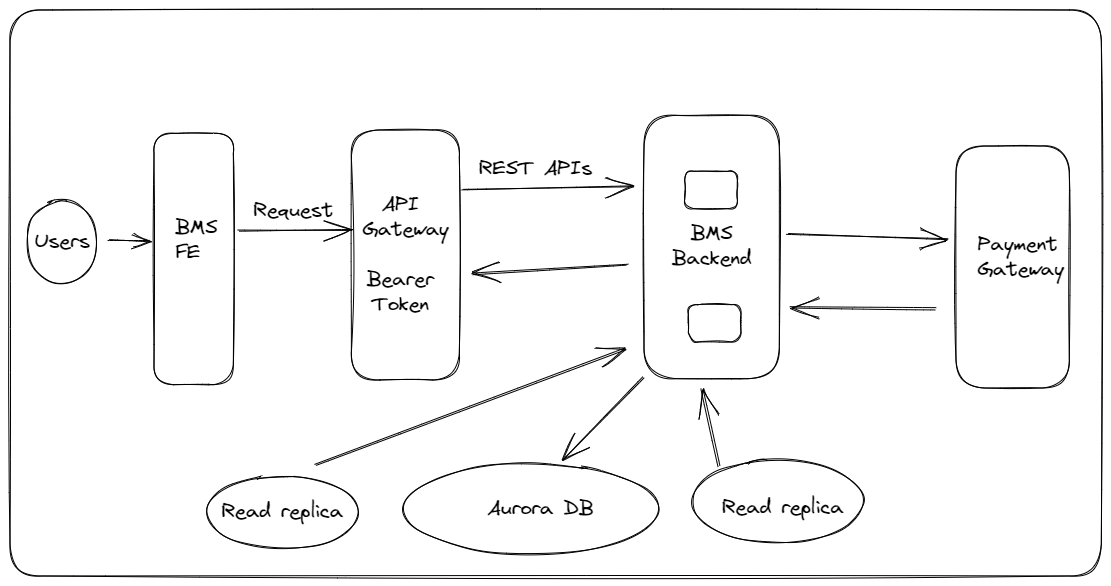
**Discussion topics & Logical View:**

**Non-functional requirements-(**Mandatory **-**Design/Arch solution & Optional Implementation**):**

* Describe transactional scenarios and design decisions to address the same.
* Integrate with theatres having existing IT system and new theatres and localization(movies)
* How will you scale to multiple cities, countries and guarantee platform availability of 99.99%?
* Integration with payment gateways
* How do you monetize platform?
* How to protect against OWASP top 10 threats.

**Candidate Solution:**

High level architecture as below:



* Describe transactional scenarios and design decisions to address the same.

When a theatre is on boarded, the screens, movies, seats, the location of theatre are also created at the same time as an atomic task/job. If there is a problem creating a theatre, then all the composition should rollback and it should not create dangling entities or data which alone doesn’t make any sense.

Like if a screen is created but the Theatre entity failed to be created due to any reason. Then there is no point of keeping the dangling screen object as it’ll be orphan and wouldn’t add any value by itself

Similarly, while booking / reserving a movie, if there is an issue allocating seats, then the booking shouldn’t go ahead and the entire transaction should roll back.

The transaction management is implemented using @Transactional annotation

on a service method which is critical section of the code which means it should either complete fully or in case of any issues should roll back the transaction.

* Integrate with theatres having existing IT system and new theatres and localization(movies)

Integration with theatres with their own IT system can be done by managing a flag while Theatre on boarding, which depicts whether Theatre has its own IT system.

That Theatre will expose their APIs for seats inventory, screens, movies etc. If they have their payment / booking system in place as well, they need to expose those APIs and our booking system can leverage those to go ahead with bookings.

The APIs exposed by that theatre will be secured by API key (as opposed to JWT token) as API key is light weight, would not introduce any latency to get authenticated /authorised and is well suited for system to system integration.

* How will you scale to multiple cities, countries and guarantee platform availability of 99.99%?

The application can make use of public cloud like AWS or a private cloud can be setup if there is an infrastructure already in place.

Let’s take the example of public cloud like AWS. Our application will be containerised using Docker during the build (CI/CD) time. The Docker image created can then be run as containers with replicas, leveraging AWS services like ECS (with or without fargate) or EKS (I wouldn’t recommend running application directly on managed EC2 instances as container orchestrators really automates automate spinning up of a new pod when load on the application increases or decreases and can scale up/down based on the needs of the application).

Using ECS or EKS will make the application highly available.

The entry point to the system APIs can be via API gateway like Apigee or custom built using Netflix zuul. The API gateway will take care of the Authentication and authorisation of the human users (admin or customer). In my code implementation I have added JWT based authentication/authorisation and it sits with the code currently. However, that piece of code can be moved out to our own custom API gateway or Apigee extension could be leveraged for auth.

The application will run in multiple regions & multiple AZs, which will make it highly available and 99.99% available.

* Integration with payment gateways

Integration with payment gateway is critical and should be treated a first class concern.

We can follow the below guidelines to make sure the integration is done without any room for errors:

1. Choose a reputable payment gateway provider: There are many payment gateway providers to choose from, so it's important to we research and select a provider that has a good reputation and a track record of secure and reliable transactions.

2. Review the documentation: Before we start coding, reviewing the payment gateway provider's documentation to understand their APIs, endpoints, and other technical requirements. Make sure we understand the data that needs to be sent and received, as well as any security considerations.

3. Implement security measures: Security is crucial when it comes to handling financial transactions. Ensure that all sensitive data is encrypted and stored securely, and implement measures such as tokenization to further protect customer data.

4. Use a robust testing environment: Before going live, thoroughly test your integration in a sandbox or testing environment provided by the payment gateway provider. This will help identify and resolve any issues before real transactions take place.

5. Provide clear error messaging: When a payment fails, provide clear error messaging to the user so they understand what went wrong and how to resolve the issue. This can help reduce customer frustration and improve the user experience.

6. Monitor for fraud: Set up fraud monitoring tools to detect and prevent fraudulent transactions. These tools can analyze transaction data in real-time and alert you to suspicious activity. This is however, will be managed at the payment gateway system end.

Eg: Stripe

Stripe is a popular payment gateway that supports credit card payments, digital wallets, and other payment methods. They offer flexible APIs and customizable checkout options, as well as fraud protection and support for recurring payments.

* How do you monetize platform?

There are several ways to monetize our application target audience. Below are some common methods:

In-App Advertising - We can include ads within the application to generate revenue. The ads can be displayed as banner ads, native ads and can earn revenue each time a user clicks on or views an ad.

In-App Purchases - We can offer additional features or content within your app that users can purchase. This can include online food orders to be provided to the customer during the movie. Users can pay for these purchases using various payment methods, such as credit cards, wallets, or mobile payments.

Subscription Model - We can offer your app on a subscription basis, where users pay a monthly or annual fee to get discounts on the luxury / premium seats. This can be a good option if the theatres premium seats are mostly vacant and it can become a good way to monetize the app further.

* How to protect against OWASP top 10 threats.

Protecting the app against the OWASP threats involves implementing various security measures throughout the software development lifecycle. A few considerations as below:

Secure Coding Practices: Use secure coding practices, such as input validation and output encoding, to prevent common vulnerabilities such as injection attacks (e.g. SQL injection, command injection).

Authentication and Authorization: Implement strong authentication and authorization controls to prevent unauthorized access to sensitive data or functionality. This includes password policies, multi-factor authentication, and role-based access control.

Secure Communications: Use encryption and secure protocols (e.g. HTTPS, SSL/TLS) to protect data in transit, and avoid using vulnerable or deprecated cryptographic algorithms.

Secure Configuration: Ensure that the application is configured securely, such as by disabling unnecessary features and services, and using secure default settings.

Error Handling: Implement appropriate error handling and logging to detect and respond to security incidents, and avoid exposing sensitive information in error messages.

Security Testing: Perform regular security testing, including vulnerability scanning and penetration testing, to identify and remediate vulnerabilities.

Secure Deployment: Implement secure deployment practices, such as using secure file permissions and avoiding hardcoding credentials or sensitive data in code or configuration files.

**Panel Feedback:**

**Select Outcome: Low/Med/High:**

**Discussion topics & Logical View:**

**Platform provisioning, sizing & Release requirements:** (Mandatory-Architecture artifacts)

* Discuss your technology choices and decisions through key drivers
* Discuss database, transactions, and data modelling.
* Discuss enterprise systems that you may need to manage specific areas.
* Discuss hosting solution and sizing (Cloud / Hybrid/ Multi cloud)- Any
* Discuss release management across cities, languages etc
* Provide details on monitoring solution
* Discuss overall KPIs
* Create a high-level project plan and estimates breakup.

**Candidate Solution:**

* Technology choice:

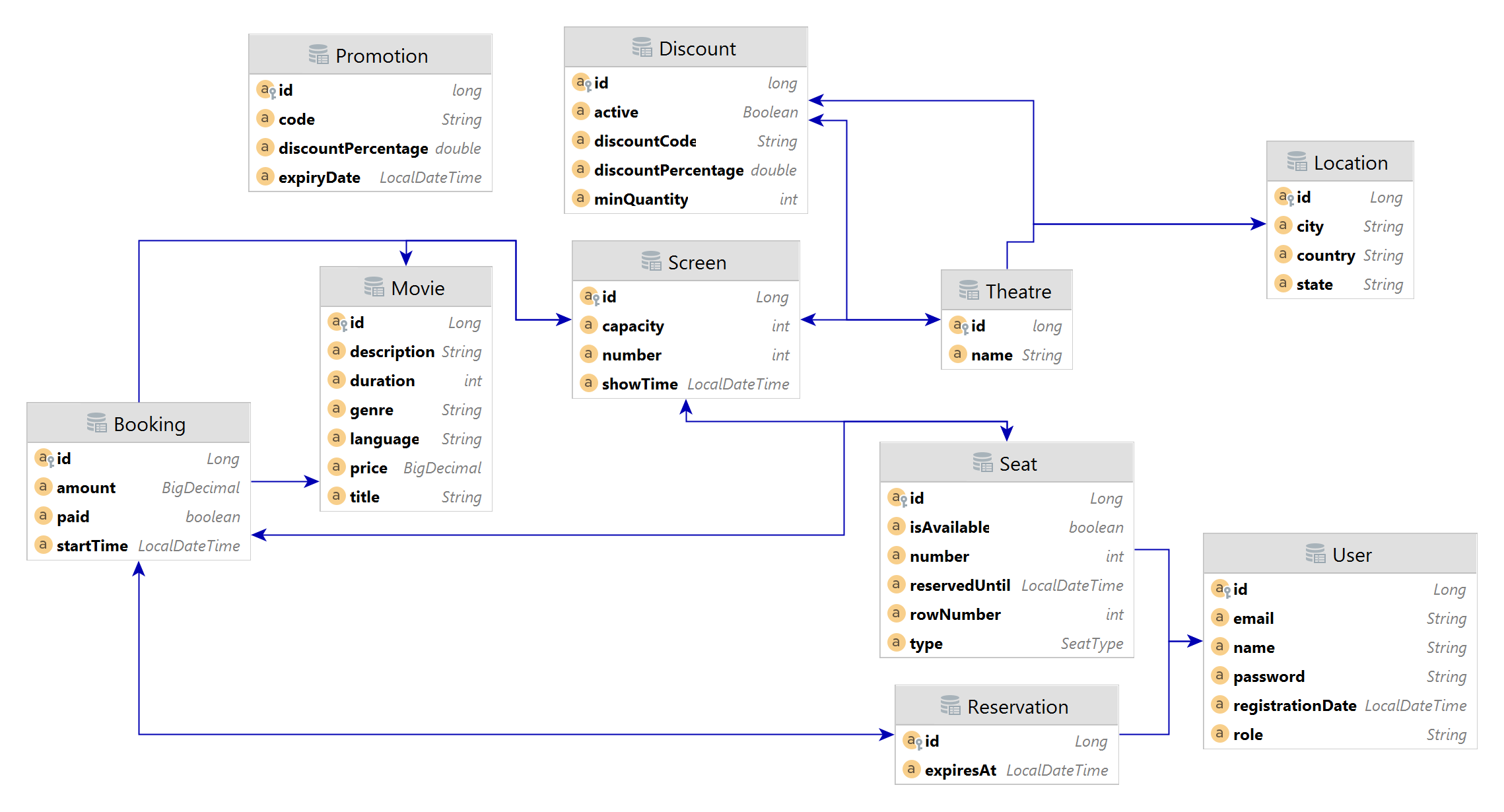
Java/springboot/JPA; key drivers: scalability, development time, community support, security, and performance.

Cloud AWS: Managed database service like aurora postgres or dynamo DB. EKS for container orchestration. SNS for notifications.

Frontend: Reactjs – as it lets you put application as compoents and hence, makes the FE modularised, highly maintainable & follow DRY principles.

* Database & data modelling:

For the MVP we’ll need the following tables. The ER diagram depicting the data model and the entity relationships as below:

****

* When it comes to hosting a movie booking system on could / AWS, there are a few key considerations to keep in mind:

Compute: We will need to choose an appropriate compute instance type based on the size and complexity of your movie booking system. For example, if we expect to have a large number of users making bookings simultaneously, we may need to choose an instance type with high CPU and memory resources.

Storage: Depending on the amount of data we need to store, we may need to choose an appropriate storage solution on AWS. This might include Amazon S3 for storing large media files, Amazon EBS for storing data that needs to be accessed quickly, or Amazon Glacier for long-term data storage.

Database: We will need to choose a suitable database solution for our movie booking system. This might include Amazon RDS for a relational database or Amazon DynamoDB for a NoSQL database. We can opt for a max of both RDBMS & no SQL. (no SQL primarily for analytics)

Load Balancing: To ensure that the movie booking system can handle a large number of users, we may need to use a load balancing solution on AWS. This can help distribute traffic across multiple instances and improve system availability and scalability.

Security: It's important to ensure that the movie booking system is properly secured on AWS. This might include using network security groups to control access to the instances, encrypting sensitive data, and setting up access control policies for AWS services.

Multi-Availability Zone: To ensure high availability and minimize downtime, you may want to consider hosting your movie booking system across multiple availability zones (AZs) within a given region. This can help ensure that your system remains accessible even if one AZ experiences an outage.

Content Delivery Network (CDN): If we have users located in different parts of the world, you may want to consider using a CDN to improve the performance of your system. A CDN can help cache content closer to users, reducing latency and improving the user experience.

Monitoring and Logging: To ensure that the movie booking system is running smoothly, we will need to set up appropriate monitoring and logging solutions on AWS. This might include using Amazon CloudWatch to monitor system metrics and set alarms, or using AWS CloudTrail to log user activity and changes to your AWS resources.

Backup and Disaster Recovery: To ensure that the data is safe and secure, you will need to set up appropriate backup and disaster recovery solutions on AWS. This might include using Amazon S3 for backup and recovery, or setting up cross-region replication to ensure that your data is always available, even in the event of a major outage.

* When it comes to release management for a movie booking system that operates across multiple cities and languages, there are a few key considerations to keep in mind:

Localization: The first step in release management across multiple cities and languages is to ensure that the system is properly localized for each city and language. This means that the user interface and content should be adapted to meet the needs of each market, including language, currency, and cultural differences.

Rollout Strategy: Once the system has been properly localized, you will need to plan a rollout strategy that takes into account the specific needs of each market. This might involve staggered rollouts across different cities or languages to ensure that the system is properly tested and any issues are identified and addressed before the system is fully deployed.

Communication: Effective communication is key to successful release management across multiple cities and languages. You will need to communicate clearly and effectively with stakeholders in each market to ensure that they are aware of the release schedule, any changes or updates to the system, and any other relevant information.

Version Control: Another important consideration in release management is version control. You will need to ensure that the correct version of the system is deployed in each market, and that any updates or changes are properly managed and tracked to avoid version conflicts.

Support and Maintenance: Finally, it's important to have a clear plan for ongoing support and maintenance of the system across multiple cities and languages. This might involve setting up dedicated support teams for each market or working with local partners to provide support and maintenance services.

Overall, effective release management across multiple cities and languages requires careful planning, clear communication, and a thorough understanding of the specific needs and requirements of each market. By taking a strategic and proactive approach to release management, you can ensure that the movie booking system is properly localized and effectively deployed to users in all of your target markets.

* Here are some details on monitoring solutions for a Java-based movie booking system:

Logging: One of the most basic monitoring solutions is logging. Logging can help developers and operations teams identify issues that occur in the system by collecting information about errors, exceptions, and events that occur within the application. With logging, you can easily see what happened in the application and diagnose the root cause of issues.

Metrics: In addition to logging, collecting metrics from the system is also important. Metrics can help us understand the performance of the system by providing quantitative data about key aspects of our application such as the response time, throughput, and error rates. By monitoring these metrics, we can detect problems and quickly take action to resolve them.

Tracing: Tracing is a powerful tool that helps you identify the path of a user's request through the system. By collecting and analyzing traces, we can identify bottlenecks and issues that are impacting the user experience. This is especially important in a movie booking system where users need to quickly and easily book tickets.

Alerts: Alerts are notifications that are triggered when certain conditions are met in the system. For example, if the response time of the system exceeds a certain threshold or if the error rate exceeds a certain percentage, an alert can be sent to the operations team. Alerts can help us detect issues before they impact users and take proactive measures to resolve them.

APM (Application Performance Monitoring): APM solutions can provide more advanced monitoring capabilities, including real-time performance monitoring and end-to-end visibility into the application. With APM, we can identify issues in the code, database, or other parts of the system that impact the user experience.

Overall, a combination of logging, metrics, tracing, alerts, and APM can provide a comprehensive monitoring solution for our booking system. By monitoring these key aspects of the system, we can ensure that the application is running smoothly and provide a seamless experience for users.

* KPIs

Conversion Rate: This measures the percentage of website visitors who actually book a movie ticket after visiting the website. A high conversion rate indicates that the system is easy to use and the booking process is smooth.

Average Order Value: This measures the average amount spent by customers on movie tickets per transaction. An increase in the average order value could indicate that customers are willing to spend more, or that the system is upselling customers with additional services.

Customer Satisfaction Score: This measures the satisfaction of customers with the movie booking system, which can be gathered through surveys, feedback forms or online reviews. A high customer satisfaction score indicates that the system is meeting the needs and expectations of users.

Website Traffic: This measures the number of visitors to the website over time. An increase in website traffic could indicate that the system is attracting more customers and generating more revenue.

Time to Complete a Booking: This measures the time it takes for a customer to complete the booking process. A shorter time to complete a booking could indicate that the system is user-friendly and efficient.

Abandoned Booking Rate: This measures the percentage of customers who start the booking process but do not complete it. A high abandoned booking rate could indicate that the system has usability issues or is not offering the right services or pricing.

* High-level project plan and estimated effort breakdown for a movie booking system using Agile methodology

Project Planning and Inception (2-4 weeks): This phase involves creating a product vision and gathering requirements from stakeholders, such as the movie theatre owners, customers, and other relevant parties. This phase also includes creating a project plan, identifying the team roles and responsibilities, and setting up the development environment.

Iterative Development and Delivery (16-20 weeks): This phase involves multiple iterations of development and delivery, with each iteration producing a working increment of the system. This phase includes developing, testing, and delivering functional increments, as well as refining the backlog based on feedback from stakeholders.

Sprint Planning (1-2 weeks per sprint): This phase involves planning each sprint, defining the sprint goals, and selecting the backlog items to be worked on during the sprint.

Daily Stand-ups (15 minutes per day): This phase involves daily stand-up meetings where the team discusses progress, roadblocks, and plans for the day.

Sprint Review (1-2 weeks per sprint): This phase involves reviewing the work completed during the sprint, demonstrating the working increment to stakeholders, and gathering feedback for further refinement of the backlog.

Sprint Retrospective (1-2 weeks per sprint): This phase involves reflecting on the previous sprint, identifying areas for improvement, and creating a plan to implement changes in the next sprint.

Estimated Breakup of Effort:

Project Planning and Inception: 10%

Iterative Development and Delivery: 70%

Sprint Planning: 5%

Daily Stand-ups: 5%

Sprint Review: 5%

Sprint Retrospective: 5%

During the project planning and inception phase, the team will focus on gathering requirements from stakeholders, creating a project plan, and setting up the development environment. This phase may also include market research and competitive analysis to ensure the system meets the needs of the target audience.

During the iterative development and delivery phase, the team will focus on developing and delivering functional increments of the system. Each increment will be tested and refined based on feedback from stakeholders. This phase may include user interface design, database schema design, and integration with third-party APIs.

During the sprint planning phase, the team will plan the upcoming sprint by selecting backlog items to work on and defining the sprint goals. Daily stand-up meetings will be held to keep the team on track and address any roadblocks that arise.

During the sprint review phase, the team will review the work completed during the sprint and gather feedback from stakeholders to refine the backlog. The sprint retrospective phase will focus on identifying areas for improvement and implementing changes in the next sprint.

**Panel Feedback:**

**Select Outcome: Low/Med/High:**

**Discussion topics & Logical View:**

**Product management and Stakeholder management**

* Please talk about stakeholder management instances
  + What decisions and actions were taken for decision closure?
* Overall technology management
* Enabling team and introducing efficiencies
* Delivery planning and estimates

**Candidate Solution:**

The only stakeholder at this stage was the HR and I did not have any pertinent questions regarding the requirements of the system.

**Panel Feedback:**

**Panel Recommendation:**

1. **Key Strengths:**

1. **Key Growth areas:**

1. **Career stage:**

1. **Title:**

Disclaimer:

*This document is meant to assess your technical skills and is classified as "Sapient confidential". This document by any means shall not be used/shared without permission from Sapient, non-adherence to this can get your candidature blocked for employment with Sapient.*