A Baseball Game Ticket Booking Web Portal

In this lesson, we'll discuss the case study of a baseball game online ticket booking application.

We'll cover the following

In this lesson, we'll have an understanding of the architecture & the key points to consider when designing an application like a baseball game online ticket booking portal.

Let's get started.

Database

Starting with the *database*, one key thing in this particular use case is the sale of tickets online. We need to set up a foolproof payment system for the fans to buy tickets to their most awaited baseball game.

For setting up payments, what do you think what database should we pick & why?

Implementing an online payment system makes *transactions* & *strong consistency* vital. The database needs to be *ACID* compliant. This makes a *relational database* like *MySQL* an obvious pick for us.

Handling Concurrency

Handling Concurrency

Another important thing to note is that the application should be designed to handle a high number of *concurrent* connections. There will be a surge of fans on the portal, to buy tickets for the baseball game as soon as they are made available.

Also, the number of requests will be a lot more than the number of tickets available.

At one point in time, there will be n requests to buy one ticket. We need to make sure the system handles this concurrent scenario well.

How will you implement this scenario? Think about it

Message Queue

One way is to *queue* all the ticket buy requests using a *message queue*. Apply the *FIFO First In First Out* principle. We've talked about this in the message queue lesson, handling concurrent requests with the help of message queue.

Database Locks

Another approach is to use *database locks*. Use the right *Transaction Isolation Level*.

A *transaction isolation level* ensures *consistency* in a database transaction. It ensures that at one point in time only one transaction has access to a resource in the database.

This is a good read on it. Also, read snapshot isolation

Transaction isolation levels can be implemented only with a *transactional ACID* compliant database like *MySQL*.

Generally, in e-commerce sites or when booking travel tickets, the number of tickets shown on the website are not accurate, they are the cached values. When a user moves on to buy a particular ticket, checks out the cart, then the system polls the database for the accurate value & locks the resource for the transaction.

Caching

Speaking of *caching*. Pick any of the popular caches like *Redis*, *Memcache* or *Hazelcast* to implement caching. There are a lot number of user events on the portal where the users just browse the website to look at the current price of the tickets and not buy them. Caching averts the load on the database in this scenario.

Backend Tech

Speaking of the backend technology, we can take a pick from *Java*, *Scala*, *Python*, *Go* etc.

To send notifications to the users we can pick a message queue like *RabbitMQ* or *Kafka*.

Let's move to the UI

User Interface

We don't really need to establish a *persistent connection* with the server as the application is kind of a *CRUD Create Read Update Delete* based app. Simple *Ajax* queries will work good.

It's a good idea to make the *UI responsive*, as fans will access it via devices having different screen sizes. The *UI* should be smart enough to adjust itself based on the screen size.

We can either design the responsive behaviour from the ground up using *CSS3* or leverage a popular *open-source* responsive framework like *Bootstrap JS*.

If you are fond of *JavaScript* frameworks you can use a framework like *React*, *Angular*, *Vue* etc. These frameworks are pretty popular in the industry & businesses prefer to use them to standardize the behaviour & the implementation of their applications.

Well, this pretty much sums the case study on a baseball ticket booking web portal.