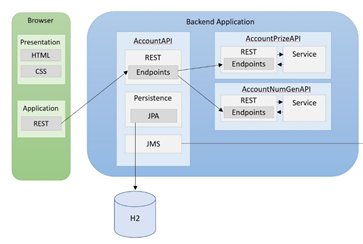
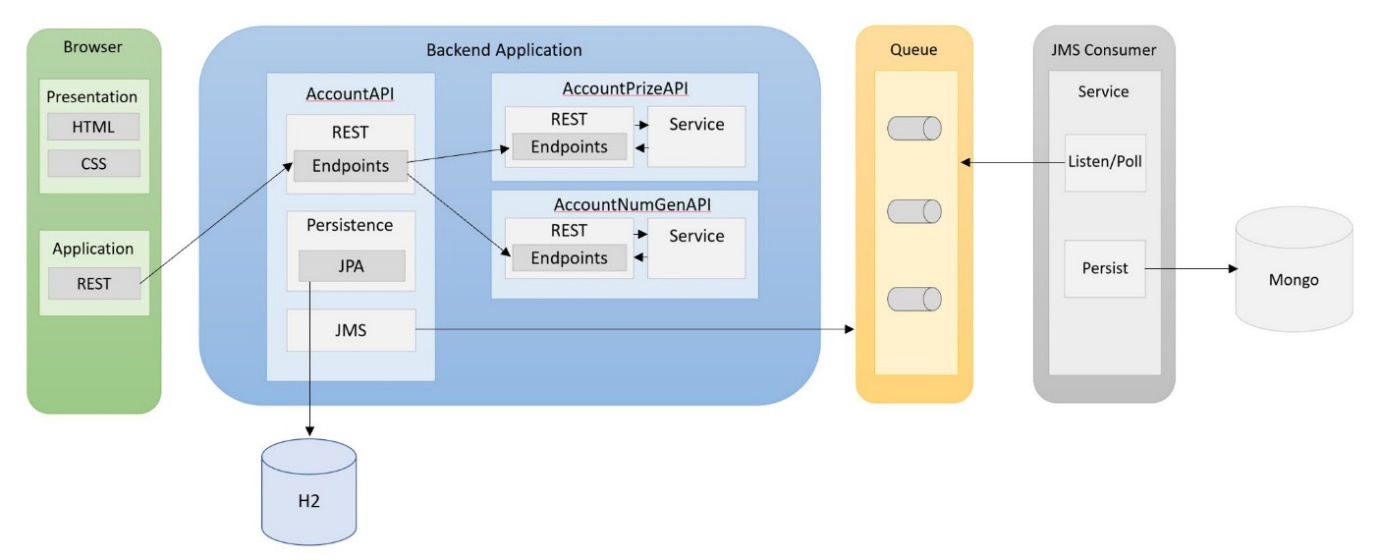
JMS

Assuming you have been following along with the handouts the current state of the application should look something like this.



Here we are going to add a JMS, or a Java Messaging Service, the service we are going to add is ActiveMQ.

We are also going to add a **consumer** for this queue, as well as a new type of database, specifically MongoDB.



# Synchronous Communication

Before we jump into the technical description we need to discuss the concept of synchronous communication.

A conversation is synchronous, one person talks whilst the other person listens and then that person waits until it is there turn to speak.

Meanwhile an email conversation would be an asynchronous communication, you send an email and then continue with your day-to-day business until you read the reply, most times you do not sit there waiting for the reply before you continue.

So from a technical perspective, http is typically a synchronous communication, we send a request and wait for the response until we continue. With an asynchronous process the response will come at some time in the future.

# Coupling

Coupling is the concept of how dependant a particular module in your system is dependent on others, for example a class that requires 5 other classes in a small system to operate might be **tightly coupled**. If a module is tightly coupled it is more likely that a different class or module breaking impacts the initial module.

# Let’s talk about Amazon

When you buy your something, this \*simple\* process triggers a lot of processes in the backend, including but not limited to:

* A stock checker or IMS
* A payment processor service
* A Delivery service – to work out where to go
* A dispatch service – to organise the delivery itself

Imagine if these services were synchronous, and the user had to sit and wait for the chain of responses to come back before they could continue!

If we called each service directly, we would also be tightly coupled to each service, if for some reason the Dispatcher service were to fail, no order would be able to be placed and no money taken because of this module failing. Realistically Amazon should still be able to operate even if this module failed for a short amount of time.

Another issue is that if for example the dispatcher service was bad - and we wanted to swap out - we would have to do in the client code - move through all the code that is talking to the dispatcher and change it to hook to the new service we wanted to implement because the 2 modules are directly linked.

strongly coupled means we have to change a lot of code because we have no resiliently, asynchronous communication helps us solve some of these issues.

# JMS

JMS is the solution.

JMS gives us a way to send async message in our system, it also provides us with loose coupling in our system as we are not tied into waiting for a response to come back.

A broker (here we are using ActiveMQ) is a technology that acts as a mechanism to have multiple **queues.** A message will be sent onto the queue by one of our microservices and that’s it, the microservice just sends the message onto the queue where it waits, it does not expect a response from the queue. It also knows nothing about service on the other end of the queue, which can be swapped out as needed.

The other end of the queue is something called a **consumer**, this is effectively another microservice that **listens** to the queue, when it has the capacity to handle a message on the queue it will remove the message from the queue and execute some instructions. This could be for example:

* Storing the message in a database
* Sending some data to an API
* Sending an email

# Types of Message

| ***Message Type*** | ***Description*** |
| --- | --- |
| StreamMessage | This message type consists of a serialized stream of objects. The objects must be read from the stream in the order they were written. |
| MapMessage | A message consisting of name/value pairs. Like a hash table, these are unordered, and each name must be unique within the map. |
| TextMessage | A message type to hold a string. |
| ObjectMessage | A message that holds a serialized object. |
| BytesMessage | A raw stream of bytes. Clients who need complete control over the raw message format use this message type. |
| XMLMessage | The WebLogic JMS implementation extends the TextMessage type with the XMLMessage to provide optimized delivery and selection of XML messages. |

# Examples of JMS

* RabbitMQ
* ActiveMQ
* Amazon Simple Queue Service
* Kafka

# Key points of a JMS

* Loosely couple components so that they can communicate with each other.
* Common workflow of a queuing service is producer -> intermediate device (broker) -> consumer
* The producer creates the message - simplest is a text message - you can put whatever in here as per the message type e.g. json or binary
* On envelope we put where we are sending it – the name of the queue
* We then send our message to a queue
* The consumer is listening all the time to the queues for something new
* Consumer takes something off the queue and does something with it - process it for instance
* **FIRE AND FORGET -** you don’t care what is consuming it you just fire off the message.