InternetExplorers

Fareed Idris	Tom Furlong	John Engracio
21207373	18366263	18452452

Synopsis

The intention of this project was to create a distributed system of services using a mix of JMS (Java Messaging Service) and REST (REpresentational State Transfer) API. Each service represents a video game. A user submits a request to a REST endpoint containing a JSON that describes an object item (i.e., weapon, melee, ranged, etc…). Each of these items contain stats such as damage, damage type and grade. The system will then create a representation of that item in each of the services (Assuming that such item type exists in a video game/service). As a finished product, this would enable seamless interoperability of items between different distinct and unique games.

Technology Stack & Components

Backend

- Programming language: Java
  - Java is a high-level, class based, object-oriented programming language. Used particularly for client-server applications. This applies to our project as we use java for the backend development, and we will also have a client side. Java is beneficial as it can be executed on any platform adopting the Write Once Run Anywhere (WORA) methodology.
- Frameworks build system: Apache Maven
  - Maven is a software project management and comprehension tool. It is used as a build system for Java applications. It is based on the concept of the project object model (POM), the pom file is the key file that Maven can manage a project's build, reporting and documentation from a central piece of information. When using Maven in projects the folder structure is project-oriented and it provides a uniform build system.
- Docker
  - Docker is a lightweight platform as a service (PaaS) for deploying software artifacts. It is based on the concept of images and containers. It is useful for projects as it has a simple download and deploy model with minimal configuration. It is a popular tool as there is a massive amount of online repository images available. We run the ActiveMQ image as a part of our project for the JMS.

Front End

- Programming Language: JavaScript
  - JavaScript is similar to Java only in name's sake, it is a high-level, text-based programming language, predominantly used to make web pages interactive in client-side applications. We use this for our client aspect of our project to give some interactivity to the consumer/customer.
- Framework: VueJS
  - Vue.js is an open-source frontend JavaScript framework mainly used for building user interfaces. It is a popular framework as it is flexible, scalable and size-efficient. We are using this framework on the front end of our project to make prototyping the user interface faster and easier.

Message Oriented Middleware using JMS and ActiveMQ

Message-Oriented Middleware (MOM) is an infrastructure that allows communication and exchanges the data (messages). In our case for this project, we are communicating between each of the services we have.

- Java Messaging Service (JMS)

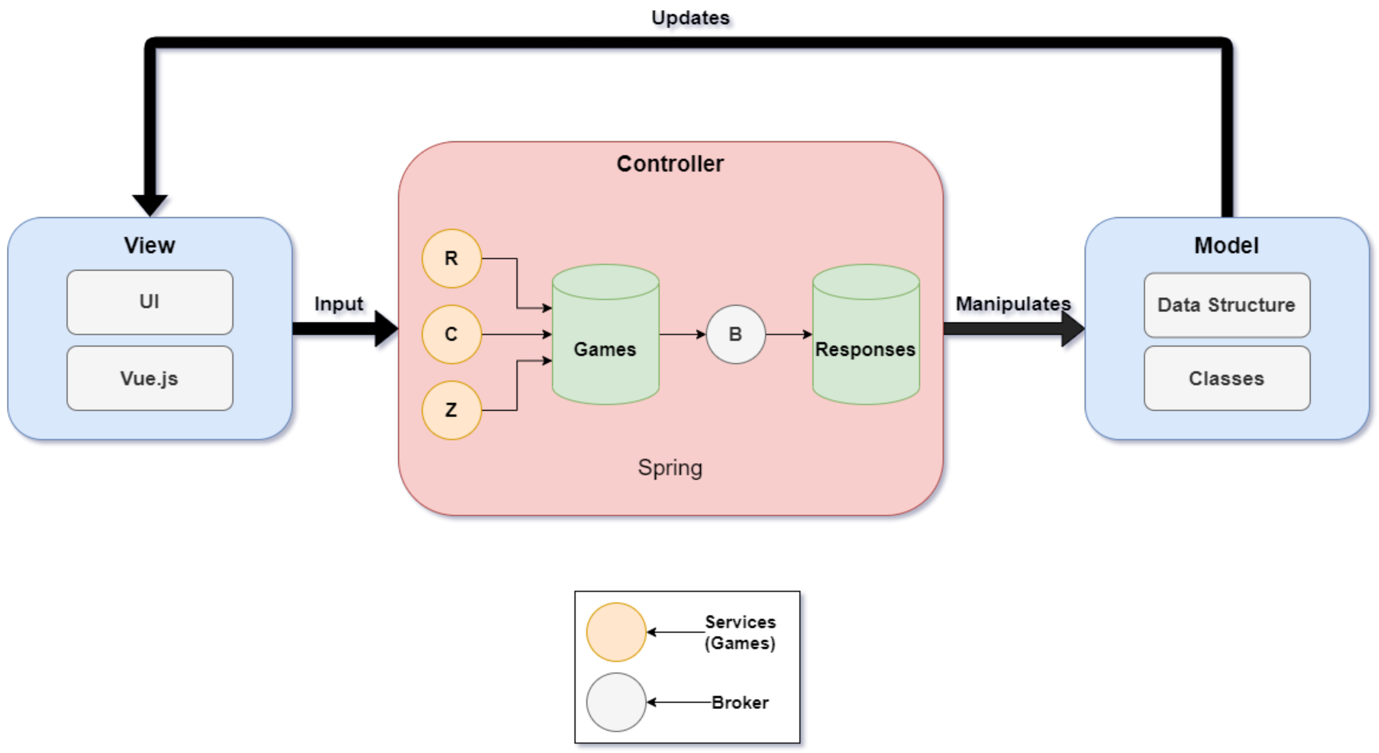
- We Use the Java Messaging Service API for implementing MOM into the project. JMS defines a standard client interface in the API that is simple and easy to learn. There are many free providers of the JMS API, about 90% of the code is the same for all providers. We use the Active MQ provider as it is the one used in labs and all team members are familiar with it.

Representational State Transfer (Rest) using Spring Boot

Rest adopts the web model for distributed systems, applications using Rest consist of resources (services) that have global identifiers. Computation is based on interaction with those resources, interaction is based on the way humans interact with the web. Rest offers a data-oriented view of distributed systems. We are using Rest to represent the different weapons and stats they have in the three different games we chose as our services.

- Spring
  - Spring is a framework for building Enterprise Java-based applications. At its core Spring is a dependency injection framework. We use Spring Boot, a simplified version of Spring that uses the Java annotations and reflection API to remove the need for XML configurations which makes it easy to create stand-alone Spring applications. It is used to work with the build systems of Maven. Rest for Spring Boot is built around the concept of REST Controllers.

System Overview



The system architecture follows a typical Model-View-Controller (MVC) architecture. The view consists of a user interface that is built using Vue.js. It sends an input from a user using said interface. The controller then receives this input from the view which then creates a queue of items from each service (video games) constrained based on what the user sent. This then interacts with the broker which will then be sent through as a response. After, the controller sends the responses as output to the model which will create data and classes that are used to update the view model (i.e., an output).

Scalability and Fault Tolerance.

In terms of scalability, the system is designed in such a way that it is easier to add additional services if needed. For example, if we were to add a new service (in this instance, a game), it is as simple as just plugging it in. This is because the architecture within the controller is based on service-oriented architecture, which is an architectural design style where services are provided to the other components by application components available in a network. Its primary principles are independent of vendors and other technologies.

Regarding the fault tolerance, there is only one single point of failure in the system which is in the *broker*. The broker is responsible for getting items in each of the available services. If for some reason the broker breaks down, then there is no way to get the items as responses to output because the services do not communicate with each other and only with the broker. As the services do not communicate with each other (i.e., they are unaware of one another), if one service were to fail, it would not impact the other services in any way detrimentally. In terms of suggestions, it is possible to try to add redundancy by creating a backup server.

Contributions

Provide a sub section for each team member that describes their contribution to the project. Descriptions should be short and to the point.

- Fareed
    - Created RuneScape game service
    - Developed Broker service
    - Developed Client service
    - Developed REST API
    - Developed UI
    - Recorded demo video
    - Reflections section of report
- Tom
    - Created Zelda game services
    - Technologies and Components section of report
- John
    - Created Cyberpunk game service
    - Developed core classes
    - Synopsis and System Architecture part of report

Reflections

There were two main key challenges that came up during the development of the project. The first came when we began integrating the REST API into the project. Due to the asynchronous nature of how requests were made to the REST API and how JMS works, we could not add a wait timer to the request handler to pull generated items from the RESPONSES queue. As a workaround, a request to create an item simply adds the items details to the correct queue and returns a response indicating an item creation job had been submitted. What happens following this is that the broker handles the generating of the item's details by each game service and puts the newly created items into a RESPONSES queue.

We then utilised Spring Boots JMS support to setup a listener on the RESPONSES queue that would pick up any created items and add them to the data store. The problem was that Spring Boots JMS integration only supported text messages in the queues while we were passing Java objects. After spending a while trying to find a workaround, we eventually settled on serialising the items to a JSON string in the broker and adding a TextMessage as opposed to an ObjectMessage. In the Spring JMS listener, we then parsed the JSON string and added the object to the save file.

The Second main challenge was regarding Cross Origin Requests. Given that we had implemented a UI that ran as a separate service and as such on a separate port, requests made between the REST API and the UI were considered cross origin. This seems to be a common problem when building a new REST API and although it is intended to increase the security of services running in production environments, it can be a bit annoying in a dev environment. To get around this, we simply added the Spring Boot function decorator @CrossOrigin and specified the URL that cross origin requests should be accepted from.

If we could start again, we would first experiment with using relational databases in a distributed manner as the main data storage method. Currently the system is great for single player games that run entirely on the client's machine, but online games and services would require a source of truth for creating items and in a production environment, a way of relating items to individual users in a clear way. Along with this, we would try using Spring Boots JMS support and integration better. It was only towards the end of the project as we were trying to fix some bugs that we realised that Spring Boot had extensive and well documented JMS support. We ended up using parts of it to enable listening to our ActiveMQ queues, which demonstrated how much easier the integration made JMS to setup and use.

Regarding things we learnt, JMS turned out to not be the best solution for real time applications. In our case, a user might want an item they created to be available for use instantly but with JMS, this cannot be guaranteed. Given that the queues are in a first-in-first-out delivery mode, a high volume of item creation request could mean a user would not have access to the items for a while. However, the main benefit of JMS for our use case was that it enabled automated addition and provisioning of new services i.e., games. If support for a new game was to be added, it could be done without interfering with the functioning of any other services. As long as the service is publishing to the right queues, it is fully setup to be a part of the system. Other distributed architectures and tools might require manual provisioning or explicit declarations of where a service is located.

A REST API is great in that upon setting up is instantly available from almost any client. When developing the UI, the only configuration needed to interact with our REST API was to specify the URL it was located at. This makes development seamless while also making the addition and deletion of endpoints very straightforward.