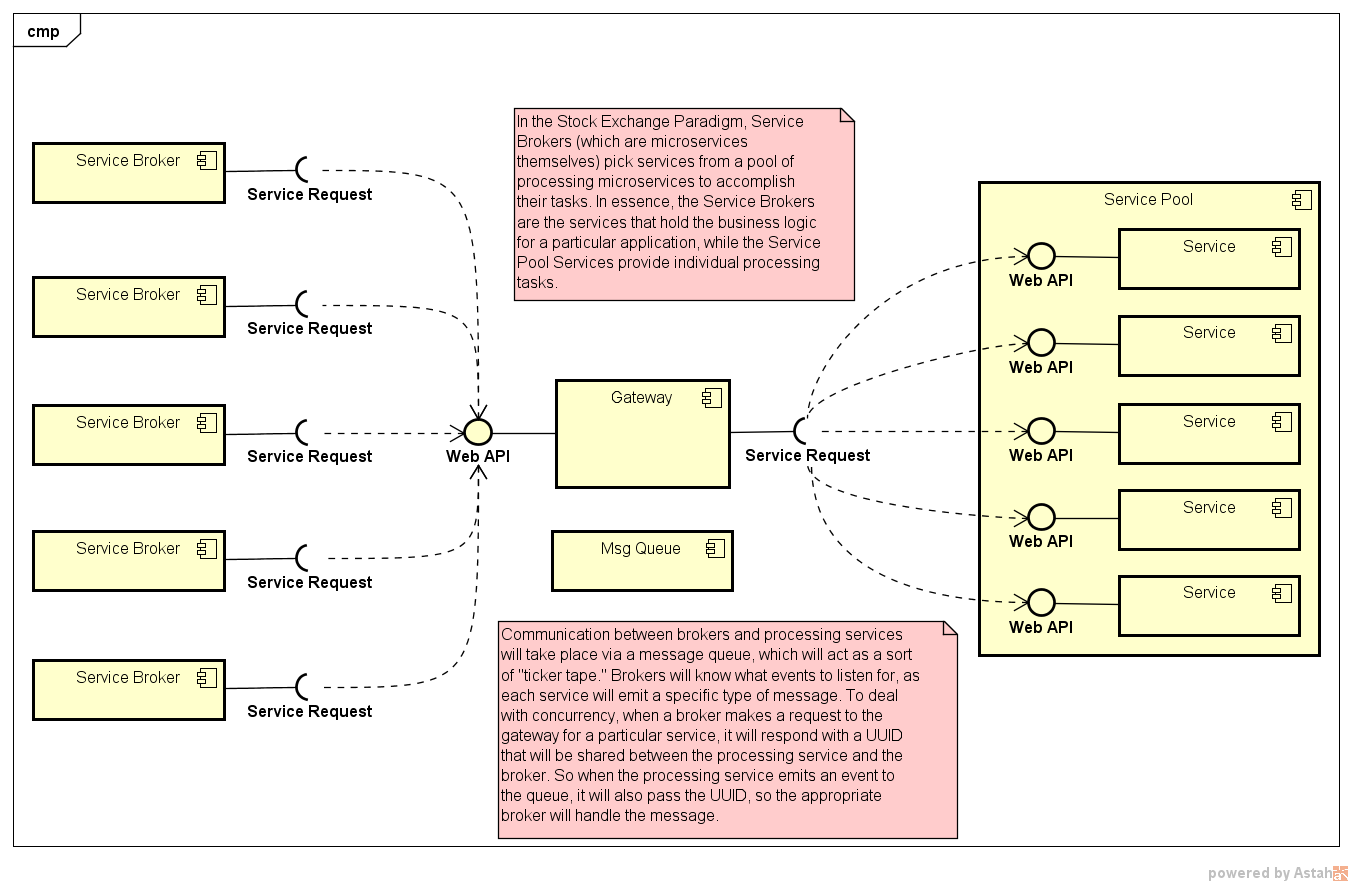
## Overview

One of the many challenges in moving to a new systems architecture – if not the most difficult challenge – is establishing an architectural **pattern** from which all teams in the engineering organization can follow, but not only that, be able to repeat. The idea of a “pattern” is an important one in that as opposed to a fully descriptive architecture that may express several concepts at once, an architectural pattern is much like a fractal. Repeating and arranging the pattern reveals the system layout, much like repeating and arranging a fractal reveals an image.

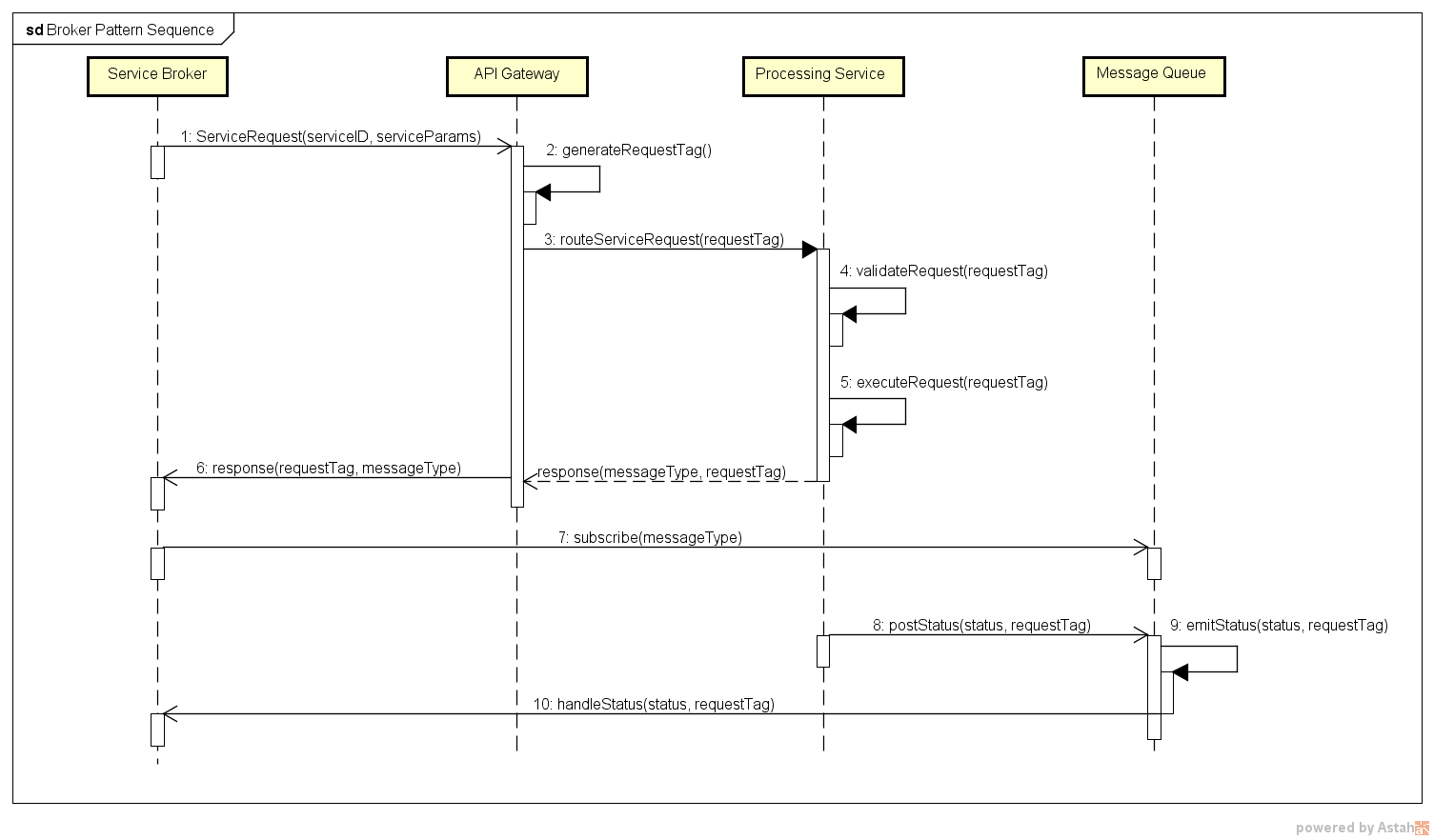
Furthermore, a pattern makes system management much easier as the actors within that pattern are limited. Of course, it is realized that the actors may have their own set of complexities, but from a management perspective should an incident occur, it is easy to identify and isolate the actor that is causing the issue.

## The Broker Pattern

The architecture that will be presented here describes an architectural pattern that will be the basis of a completely decoupled system, loosely mimicking that of a trading floor of a stock exchange. A broker will receive an order, then invoke the financial instrument they wish to trade, then execute the order. The status of the order can be tracked via theticker tape to monitor changes in price. In this architecture, a broker represents a service that can invoke one or more processing services, then monitors a message queue for changes to the status of those services.

## Request Flow Sequence

The mechanics behind the broker pattern are described in the sequence diagram below, then discussed in detail.



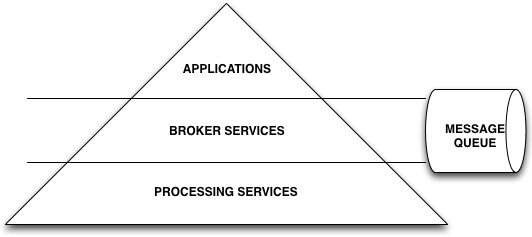
### Sequence Description

1. Broker makes a service request, passing the serviceID and service parameters to the API Gateway
2. The gateway immediately generates a requestTag
   1. Note that this could be a UUID
3. Once the tag is generated, the gateway routes the request to the appropriate processing service.
4. The service validates the request
5. The diagram illustrates a successful validation but if the validation fails, it will respond and the gateway will throw a 400 Bad Request.
   1. If valid, the service will begin executing, then respond to the gateway, sending the messageType it will be emitting and the requestTag
6. The gateway in turn will forward the response to the broker with the messageType and requestTag.
7. Once the broker receives a response, it will subscribe to the messageType on the Message Queue, and begin listening for that message.
8. During processing, the process service will post status messages including the status and the requestTag.
9. The message queue will emit the message as it receives it.
10. The broker will handle the status
    1. The assumption is that the correct messageType has been emitted.
    2. If the requestTag is the one in which the broker expects, it will process the status.

## Broker Pattern Actors

There are four primary actors in a broker “transaction.” These will be described in detail below. But note that these actors apply to everything underneath the user interface. The interaction between the UI and an application entity follows a more traditional MVC pattern that we will cover later in this document.

While technically everything underneath the UI is a service, they take on specific roles. How a developer approaches building their service will be influenced by the role they are trying to fill.



### Processing Service

This is the most elemental service in the system, and will most probably represent the greatest physical number of services. A processing service is responsible for processing a *specific* operation in the system; for instance, performing a file copy, or submitting and executing a Spark job. The idea behind this is that this service does one thing and one thing only. Granted, there may be several internal functions that it performs over the course of its operation, but its behavior is specifically defined, thus inputs and outputs are constant and predictable.

### Broker Service

The Broker Service - for which this pattern is named - is a special service that will invoke and encapsulate multiple processing services. You might consider a broker service to represent a workflow execution module. Whereas the processing service performs a single duty, the broker invokes several services to carry out a specific workflow. An example of this would be data ingestion (a diagram of this is shown below) where what is called a “Producer” receives a workflow configuration as input then invokes several services to ingest customer data into the system.

### Application

The application could be considered a “broker of brokers.” It represents a specific problem domain within the ecosystem; for instance, RV, or OSM. It has the ability to not only invoke brokers, but can also invoke services directly to fulfill its business requirements.

### Message Queue

At the heart of the pattern is the message queue. In a truly decoupled system, there must be a means by which services can communicate. The message queue provides that. As a service executes, it can post messages onto the message queue. Subscribers to those messages will then be notified and then respond accordingly. This represents the classic “pub-sub” pattern and is also the hallmark of “reactive programming.”

## \*\*\*THE MOST IMPORTANT RULE IN IMPLEMENTING THE BROKER PATTERN\*\*\*

*Thou shall not pass data sets on the message queue!*

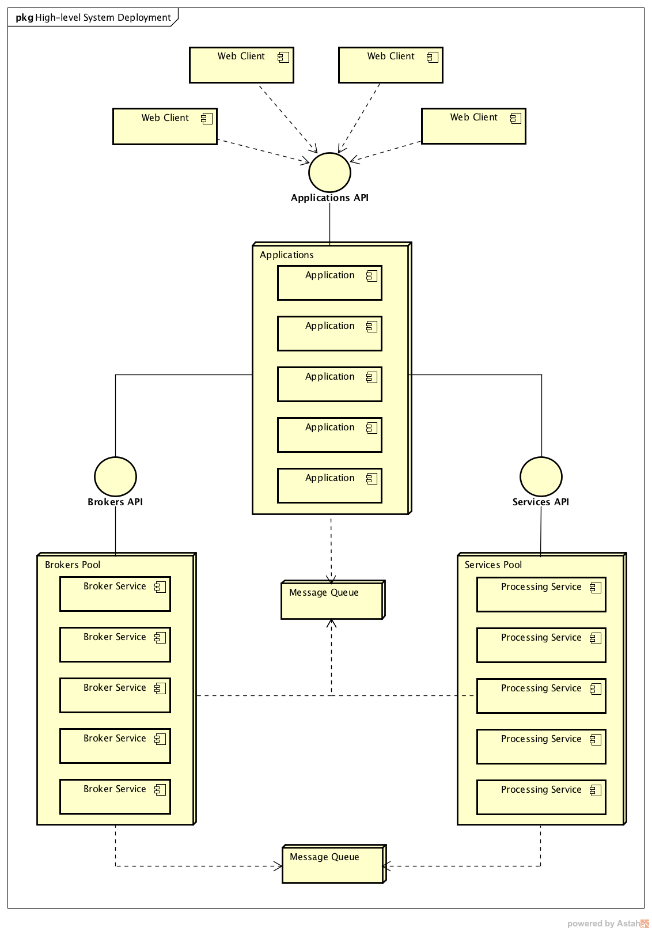
It will be tempting to pass the result set of an operation to a queue to make it easier for listeners to process. But a queue service can easily be overrun when there is a high velocity of messages that contain data posted to the queue concurrently. Furthermore, passing data onto the queue creates an implied dependency on the queue. You must resist this temptation!

A message queue, with respect to this pattern, is specifically designated to handle lightweight messaging only. The exception to this is a streaming queue like Kafka or Flink which are designed for data streaming. But streaming queues should be relegated to processing services to be used during their execution.

So how does a service get the data that it needs? There are a variety of ways this can be done, though most commonly a reference to a temporary data store or a database reference (id) is passed as part of the message.

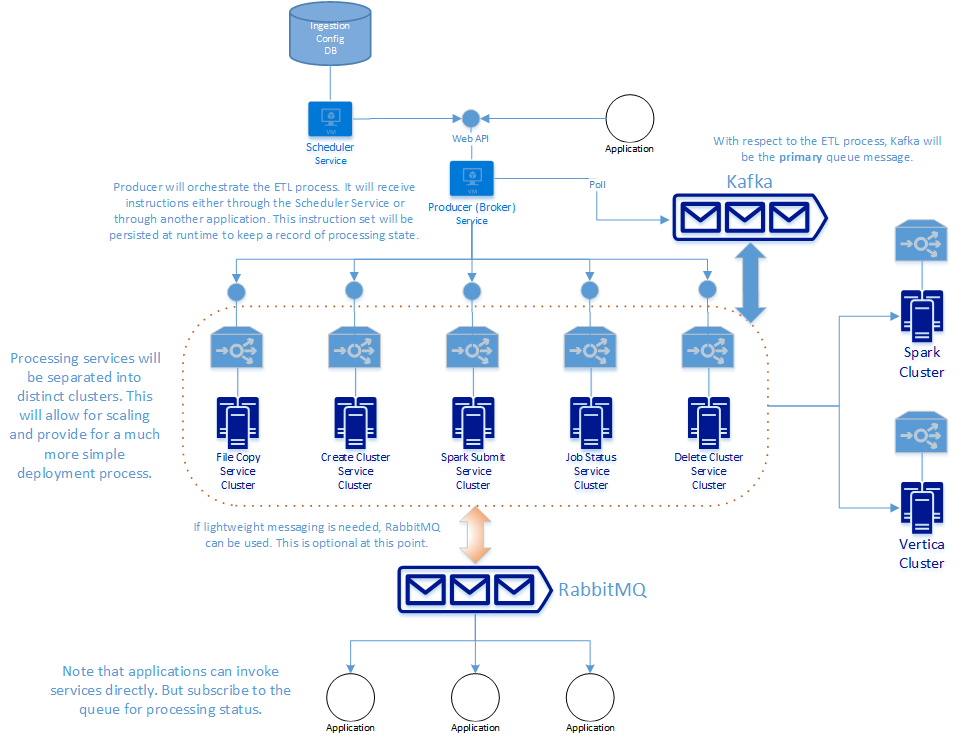
## System Overview

The high-level system layout can be described by the following diagram:



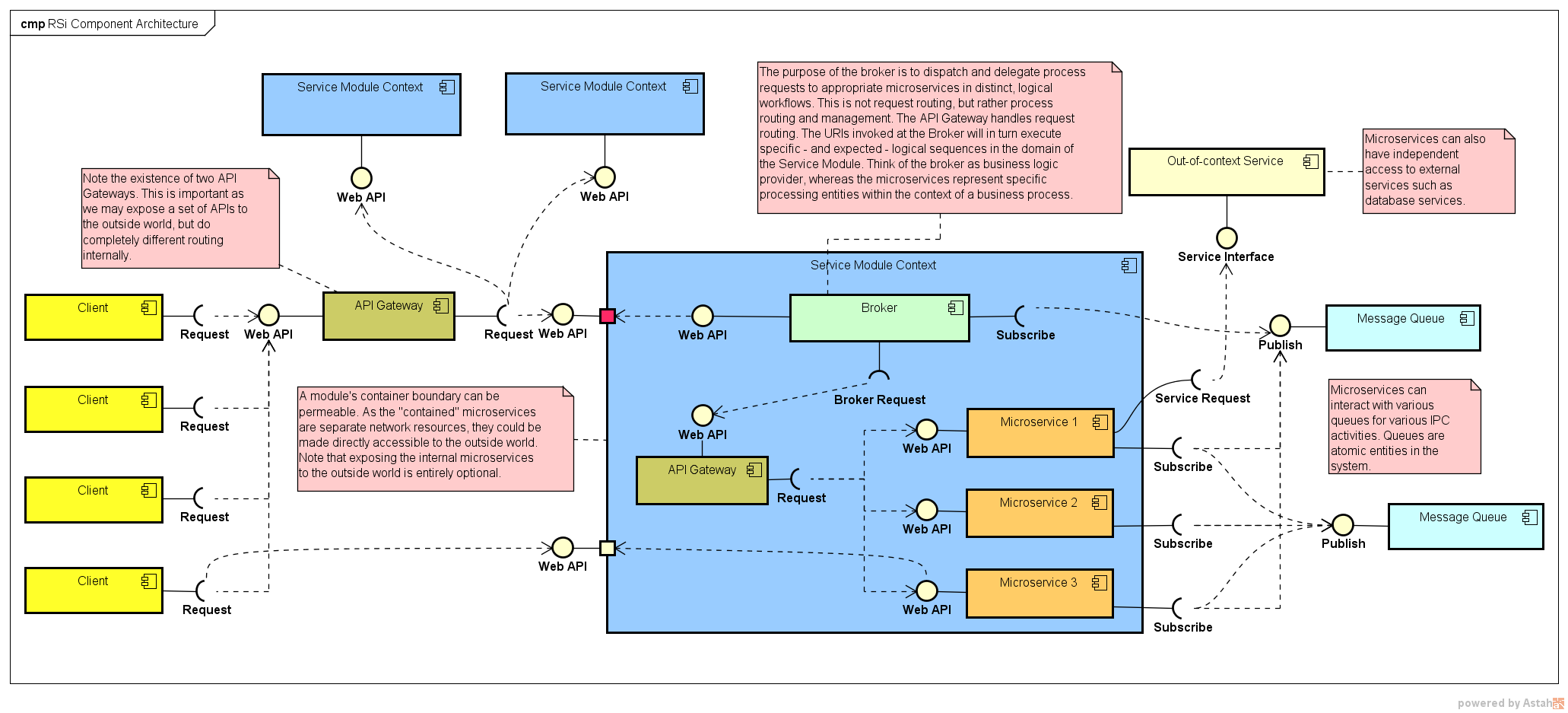
As described above, the system is separated into four distinct areas: Clients, Applications, Brokers, and Services. Applications, Brokers, and Services are in “pools,” simply to denote their function in the system. The diagram above provides a simple visual representation of the overall system.

## Example: Ingestion Service



The diagram above represents a high-level overview of the Ingestion Service. At the top of the diagram, a scheduler service and application invoke the Web API of the Producer (Broker), passing a workflow configuration to it. The Producer in turn invokes the services contained in the box demarked by the dotted orange line. While the services run, they will provide messages to either (or both) Kafka and RabbitMQ for status. The service will use Spark and Vertica over the course of their execution.

## Relationships in a Broker Pattern



The most important takeaway from this diagram is that ALL services are accessed via a Web API. That is their ONLY interface to the outside world. From that perspective, services become “black boxes.” Invokers of the services do not nor should they care about the internal mechanics of the invoked services. Their only concern is retrieving the result sets from those services.

## Implementing the Pattern: Technologies

The pattern presented here is quite easy to follow. But then the question becomes what technologies are available to implement a project that will use this pattern. To address this, let’s first establish one fundamental tenet:

*Use the best tool for the job…*

This does not mean we have the freedom to choose any tool. In a moment we discuss the tools, languages and/or frameworks which should be used in each area of the system. **NOTE: This is not an exhaustive list!** The final tools we will use will depend on our needs as we implement.

### UI

The accepted tools and frameworks for the UI are as follows:

* VueJS framework
* Axios for AJAX
* NodeJS for local development server
  + Node services
    - Grunt
    - Express
    - Webpack
    - Babel
* Bootstrap for CSS/presentation
* Font-awesome for web app icons
* JQuery if absolutely needed (but not recommended)

### Applications

* Java
  + Spring Framework
    - This includes several Spring Packages that will be useful for development including:
      * Spring Kafka
      * Spring AMQP
      * Spring Security
    - Note that Spring HATEOAS will be required to provide consistent response constructs
    - We will want to leverage as many of these projects as possible.
* Microservices will be discussed below.

### Message Queues and Processing

* Kafka
  + Flink is also available. The intent is to start with Flink, then when we start building out our stream capabilities, we will evaluate what the best solution will be.
* RabbitMQ for lightweight messaging
* Task Scheduling
  + Depending on the language a task scheduler can be used: Celery (Python), Quartz or Spring Scheduled Task (Java).

## Microservices

At the heart of the implementation of our architectural pattern are microservices. All applications, brokers and services will be microservices. We have identified that NGiNX will be used for load balancing and API Gateway. For API registry services, we can use Consul, etcd and Apache Zookeeper.

To serve applications, we will use Apache Tomcat. Applications and services will reside in Docker containers to provide for maximum mobility in the system as needed.

## Implementing the Architecture