Metallica: A full-stack microservices based web application

This document outlines an exercise to build a full-stack web application using cutting-edge front-end technologies and microservices. The intent is to implement a reasonably complex business domain that challenges your design and implementation skills. The domain is *physical metals trading,* which involves the following processes:

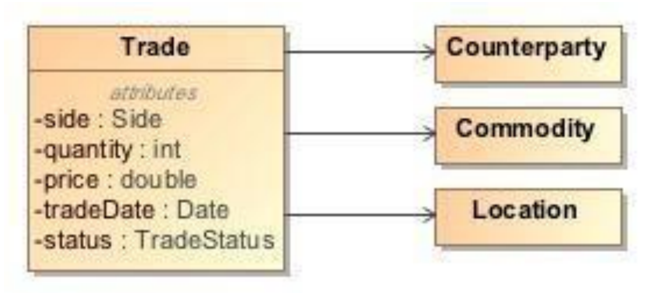
- Purchase and sale of metals such as aluminum, zinc and copper
- Live streaming of market prices

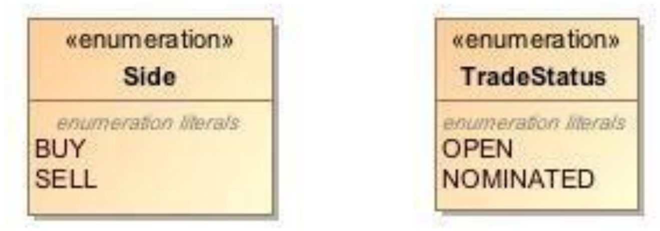
We will provide a high-level architecture for the desired system. You must follow this architecture in your detailed design and implementation. You must also make sure that each component of your system is well tested using automated tests. In addition, your system as a whole shou ld be integration tested.

[This project is named as a tribute to the heavy metal band Metallica, whose fast tempos, instrumentals, and aggressive musicianship placed them as one of the top heavy metal bands in the world.]

Business Domain

Trades





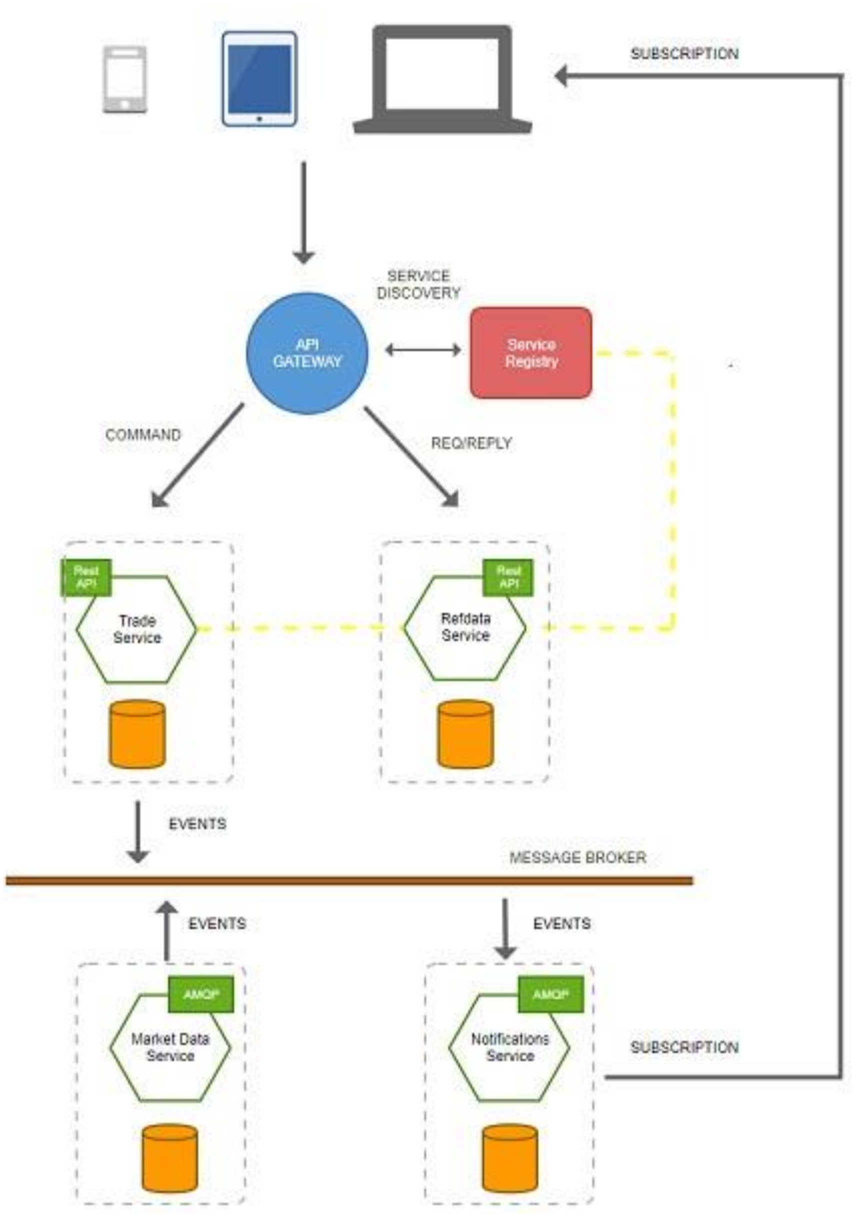
A *Trade* is a purchase or sale of a commodity from (or to) a counterparty at a specified location, date and price. For example, we could buy 100 MT (metric tons) of aluminium from a counterparty at Buenos Aires on a specified date at $1,860.75/MT and sell it to another counterparty in London on a later date at $2,010.20/MT.

In order to meet the sales terms, we would have to transport the material from Buenos Aires to London.

New trades are in *open* status.

High-Level Architecture

The diagram below illustrates the high-level architecture of Metallica.



Above architecture supports a mix of Restful and Event Driven microservices. Each of the restful microservice would register itself with the service registry for service discovery.

All the requests from the frontend would be routed through an API Gateway. API Gateway would look-up for service details in the service registry and then route the calls to the appropriate microservice. These microservice can then publish the events on the message broker which would be picked up by the notifications service. Notification service would push those notifications to the frontend via web sockets.

Microservices support following inter-process communication mechanisms:-

- **Commands** are actions that modify state. A Command asks a service to do something, e.g. Trade Service, please create a trade.

- **Events** are notifications that inform listeners when something has happened, e.g. a Trade was created. Events never modify state directly.
- **Request/Responses** ask services for information, e.g. Refdata Service, please give all the counterparties starting with 'ABN'.

Note that each microservice is autonomous and fully encapsulated. It contains its own persistence data store, e.g. a relational, aggregate or graph database. The only way to get data from a microservice is through its messaging API. In other word, a microservice does not expose its data store to the outside world - not even to other microservices. This ensures that microservices are resilient to internal implementation changes.

Exercise

Implement Metallica. You must support the following requirements:

Frontend

- Allow traders to enter trades as described in the domain model.
- Allow traders to receive notifications for changes in trades
- Allow traders to view the live update of the market price of various metals

Backend
Four micro -services, each responsible for a subset of the business domain
c ontext):

- **Trade Service** supports the management of *Trade* instances.
- **Notifications Service** supports pushing of trade notifications and live streaming of market prices.
- **Market Data Service** supports streaming of market price of various metals.
- **Refdata Service** supports read-only reference entities such as *Counterparties*, *Commodities* and *Locations*. Each reference entity should have an *identifier* that can be used to reference it from other entities and other required attributes. For example, a counterparty may have an *identifier* (e.g. "AAPL") and a *name* (e.g. "Apple, Inc.)

Technology Stack

Use the following technologies in your implementation:

Different implementation Options

- Angular 2 with Spring Boot
- Angular 2 with Node.js
- React / Redux with Node.js
- React / Redux with Spring Boot

Front-End

- ES6 (ES2015)
- React
- Material UI
- Redux
- Angular 2

API Gateway

- ZuuL
- Spring Cloud
- Java

Service Discovery

- Eureka
- Spring Cloud
- Java
- Node.js

Microservices

- Java 8
- Spring Boot
- Spring AMQP (for connecting to RabbitMQ)
- Hibernate (for Postgres)
- Spring Data (for connecting to MongoDB)
- Node.js

Messaging Infrastructure

- RabbitMQ

Streaming

- Spring Cloud
- WebSockets

Database

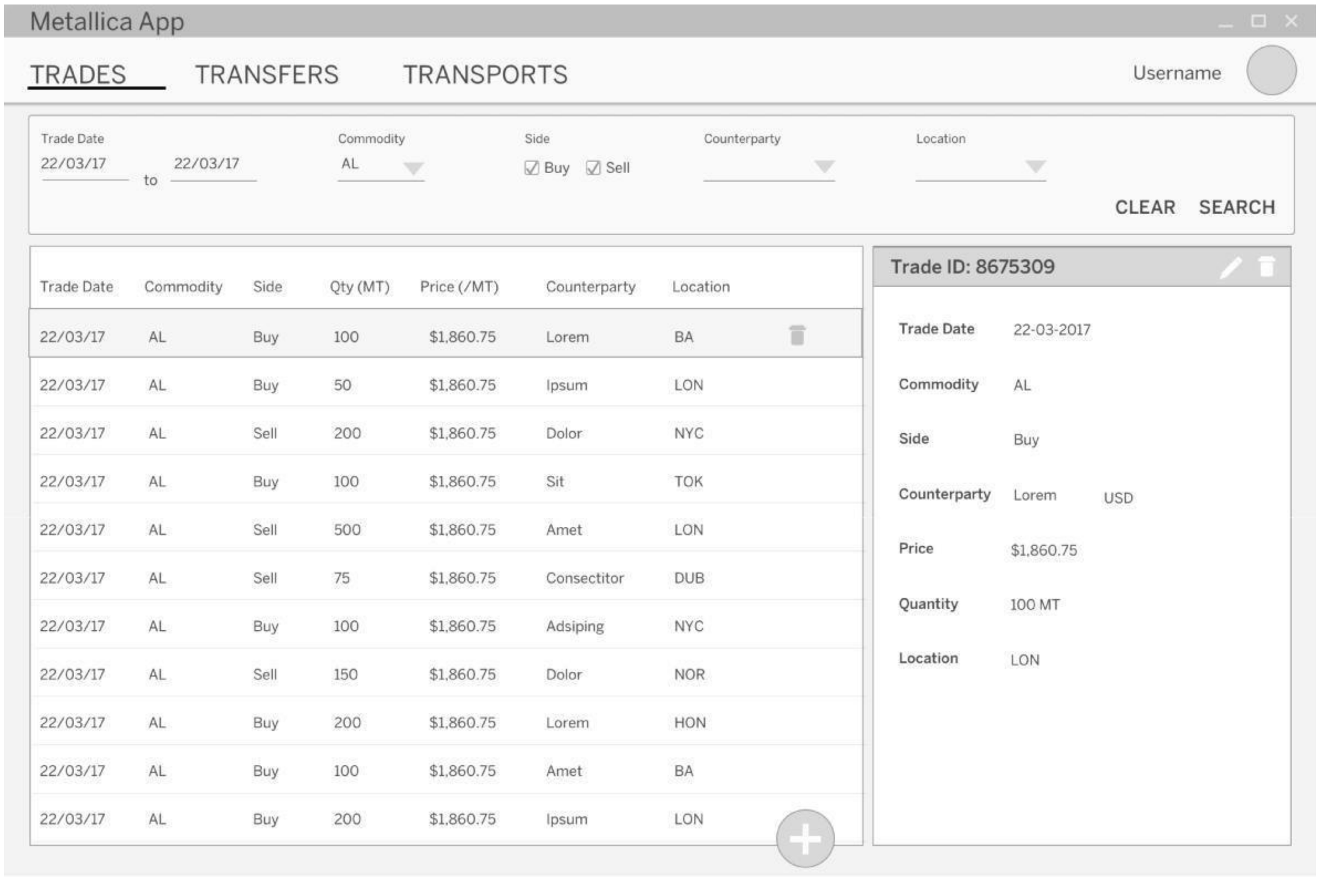
- Postgres
- MongoDB

Front-End Design

The front-end design for Metallica consists of three screens: Trades, Transfers and Transports.

Trades Screen

This screen is used to search, create, edit, view and delete trades.



Trade Date 22/03/17 to 22/03/17

Commodity AL

Side ☑ Buy ☑ Sell

Counterparty

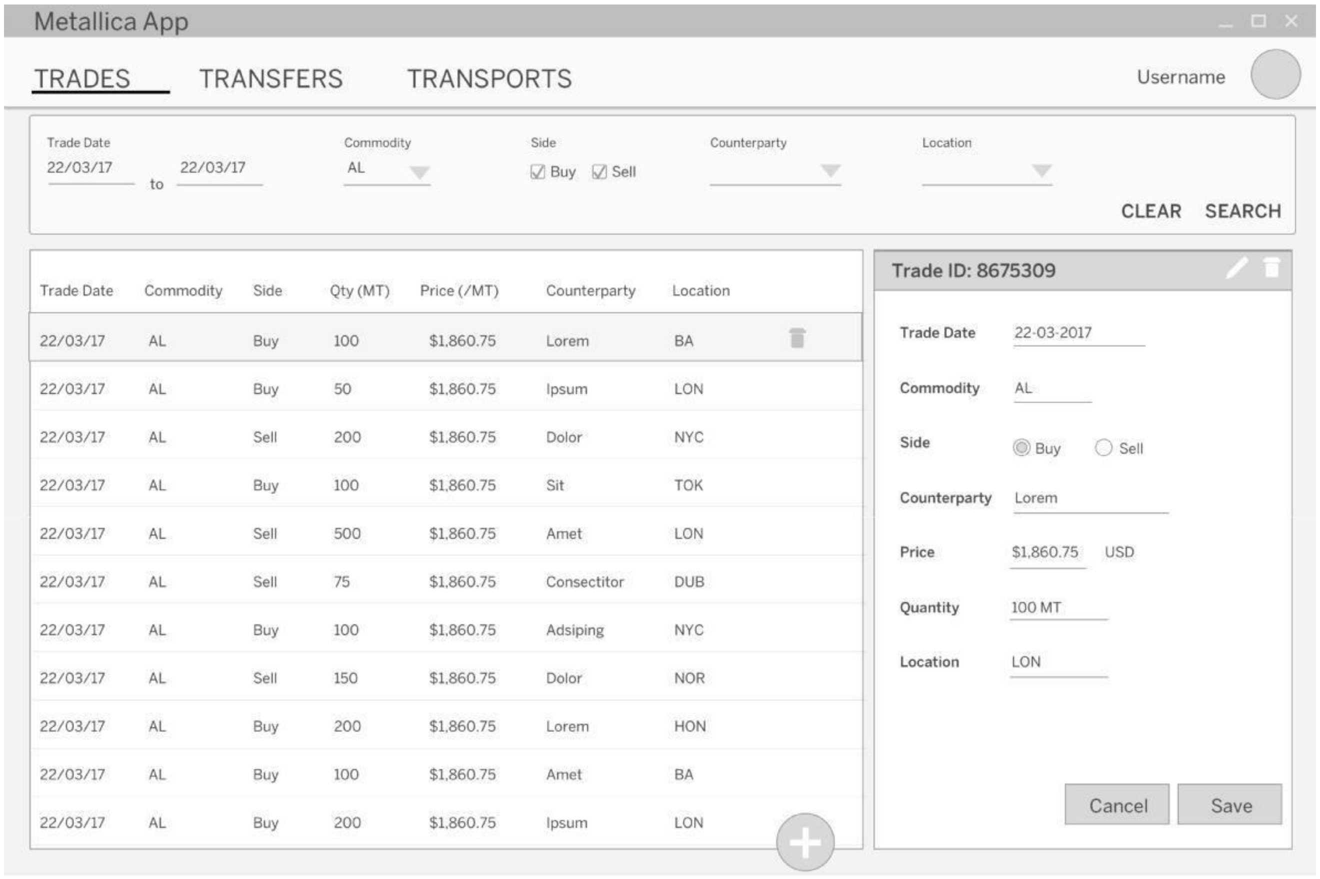
Location

Trade Date	Commodity	Side	Qty (MT)	Price (/MT)	Counterparty	Location
22/03/17	AL	Buy	100	$1,860.75	Lorem	BA
22/03/17	AL	Buy	50	$1,860.75	Ipsum	LON
22/03/17	AL	Sell	200	$1,860.75	Dolor	NYC
22/03/17	AL	Buy	100	$1,860.75	Sit	TOK
22/03/17	AL	Sell	500	$1,860.75	Amet	LON
22/03/17	AL	Sell	75	$1,860.75	Consectitor	DUB
22/03/17	AL	Buy	100	$1,860.75	Adsiping	NYC
22/03/17	AL	Sell	150	$1,860.75	Dolor	NOR
22/03/17	AL	Buy	200	$1,860.75	Lorem	HON
22/03/17	AL	Buy	100	$1,860.75	Amet	BA
22/03/17	AL	Buy	200	$1,860.75	Ipsum	LON

Trade ID: 8675309

Trade Date	22-03-2017
Commodity	AL
Side	Buy
Counterparty	Lorem USD
Price	$1,860.75
Quantity	100 MT
Location	LON

Below is an example of a trade being edited:



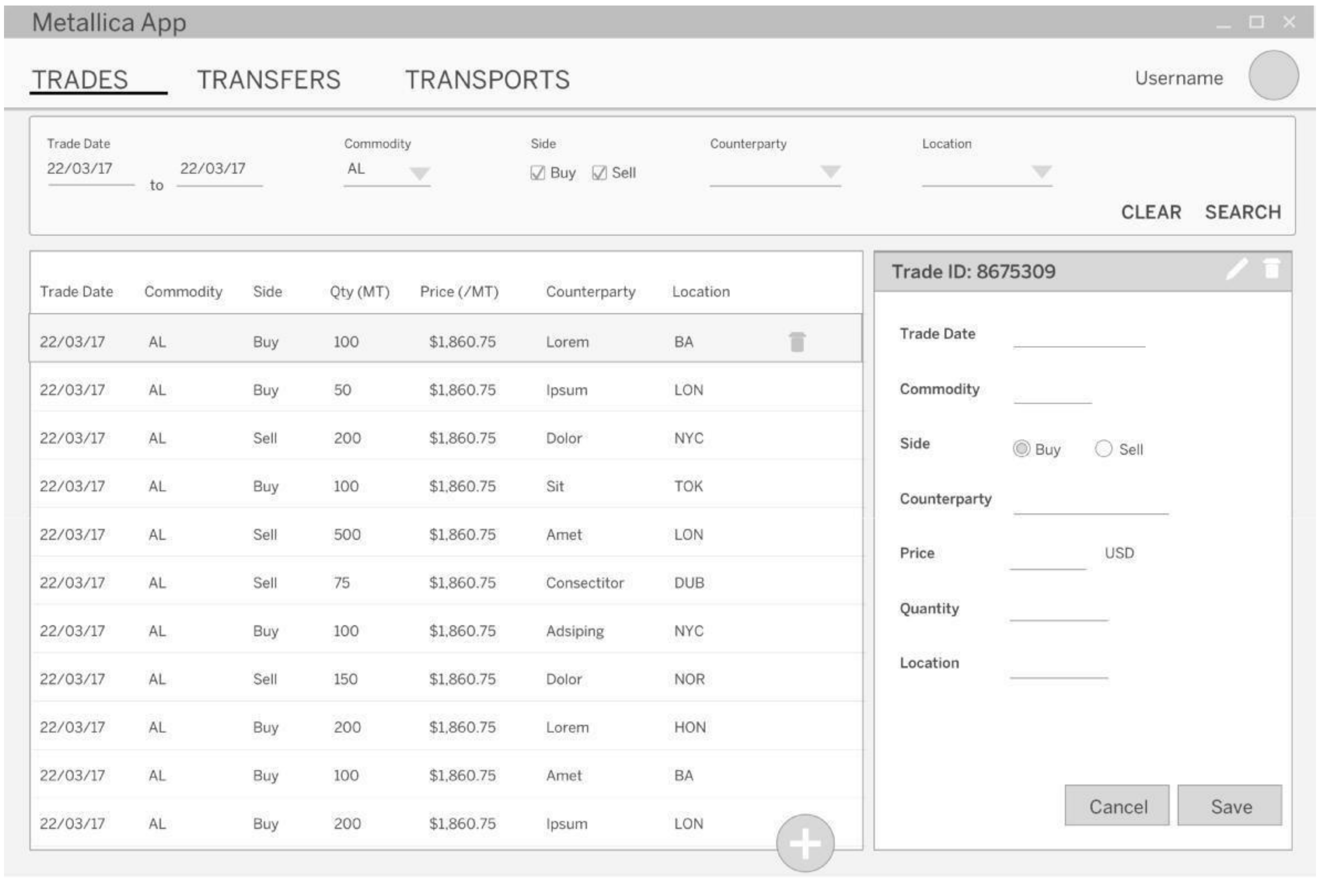
Trade Date 22/03/17 to 22/03/17
Commodity AL
Side [✓ Buy] [✓ Sell]
Counterparty
Location

Trade Date	Commodity	Side	Qty (MT)	Price (/MT)	Counterparty	Location
22/03/17	AL	Buy	100	$1,860.75	Lorem	BA
22/03/17	AL	Buy	50	$1,860.75	Ipsum	LON
22/03/17	AL	Sell	200	$1,860.75	Dolor	NYC
22/03/17	AL	Buy	100	$1,860.75	Sit	TOK
22/03/17	AL	Sell	500	$1,860.75	Amet	LON
22/03/17	AL	Sell	75	$1,860.75	Consectitor	DUB
22/03/17	AL	Buy	100	$1,860.75	Adsiping	NYC
22/03/17	AL	Sell	150	$1,860.75	Dolor	NOR
22/03/17	AL	Buy	200	$1,860.75	Lorem	HON
22/03/17	AL	Buy	100	$1,860.75	Amet	BA
22/03/17	AL	Buy	200	$1,860.75	Ipsum	LON

Trade ID: 8675309

Trade Date	22-03-2017
Commodity	AL
Side	[✓ Buy] Sell
Counterparty	Lorem
Price	$1,860.75 USD
Quantity	100 MT
Location	LON

Finally, here's an example of a new trade being created:



Trade Date		Commodity	Side	Counterparty	Location
22/03/17 to 22/03/17		AL	☑ Buy ☑ Sell		

Trade Date	Commodity	Side	Qty (MT)	Price (/MT)	Counterparty	Location
22/03/17	AL	Buy	100	$1,860.75	Lorem	BA
22/03/17	AL	Buy	50	$1,860.75	Ipsum	LON
22/03/17	AL	Sell	200	$1,860.75	Dolor	NYC
22/03/17	AL	Buy	100	$1,860.75	Sit	TOK
22/03/17	AL	Sell	500	$1,860.75	Amet	LON
22/03/17	AL	Sell	75	$1,860.75	Consectitor	DUB
22/03/17	AL	Buy	100	$1,860.75	Adsiping	NYC
22/03/17	AL	Sell	150	$1,860.75	Dolor	NOR
22/03/17	AL	Buy	200	$1,860.75	Lorem	HON
22/03/17	AL	Buy	100	$1,860.75	Amet	BA
22/03/17	AL	Buy	200	$1,860.75	Ipsum	LON

Trade ID: 8675309

Trade Date	
Commodity	
Side	◉ Buy ○ Sell
Counterparty	
Price	USD
Quantity	
Location	

Testing Approach

Be sure to define *Behavioral Unit Tests* for the front-end and each microservice. If you don't know what that means, watch this video:

Ian Cooper: TDD, where did it all go wron g

The most important take-away from this should be the understanding that ***our testing strategy leverages the Ports and Adapters Architecture to allow the units under test to be tested directly, but in terms of the behavior we are actually interested in***. Once you understand this concept, you will be well equipped to test your system.

Happy Implementation!