

# Master Microservices



# Spring Boot in 10(ish) Steps

# Getting Started with Spring Boot

In **28**  
Minutes

- **WHY** Spring Boot?
  - You can build web apps & REST API WITHOUT Spring Boot
  - What is the need for Spring Boot?
- **WHAT** are the goals of Spring Boot?
- **HOW** does Spring Boot work?
- **COMPARE** Spring Boot vs Spring MVC vs Spring



# Getting Started with Spring Boot - Approach

In **28**  
Minutes

- **1:** Understand the world before Spring Boot (10000 Feet)
- **2:** Create a Spring Boot Project
- **3:** Build a simple REST API using Spring Boot
- **4:** Understand the MAGIC of Spring Boot
  - Spring Initializr
  - Starter Projects
  - Auto Configuration
  - Developer Tools
  - Actuator
  - ...



# World Before Spring Boot!

In **28**  
Minutes

- Setting up Spring Projects **before Spring Boot** was **NOT** easy!
- We needed to configure a **lot of things** before we have a **production-ready** application



# World Before Spring Boot - 1 - Dependency Management In 28 Minutes

```
<dependency>
  <groupId>org.springframework</groupId>
  <artifactId>spring-webmvc</artifactId>
  <version>6.2.2.RELEASE</version>
</dependency>
<dependency>
  <groupId>com.fasterxml.jackson.core</groupId>
  <artifactId>jackson-databind</artifactId>
  <version>2.13.3</version>
</dependency>
<dependency>
  <groupId>log4j</groupId>
  <artifactId>log4j</artifactId>
  <version>1.2.17</version>
</dependency>
```

- Manage frameworks and versions
  - **REST API** - Spring framework, Spring MVC framework, JSON binding framework, ..
  - **Unit Tests** - Spring Test, Mockito, JUnit, ...

# World Before Spring Boot - 2 - web.xml

```
<servlet>
  <servlet-name>dispatcher</servlet-name>
  <servlet-class>

    org.springframework.web.servlet.DispatcherServlet
  </servlet-class>
  <init-param>

    <param-name>contextConfigLocation</param-name>

    <param-value>/WEB-INF/todo-servlet.xml</param-value>
  </init-param>
  <load-on-startup>1</load-on-startup>
</servlet>

<servlet-mapping>
  <servlet-name>dispatcher</servlet-name>
  <url-pattern>/*</url-pattern>
</servlet-mapping>
```

- **Example: Configure DispatcherServlet for Spring MVC**



# World Before Spring Boot - 3 - Spring Configuration

```
<context:component-scan base-package="com.in28minutes" />

<bean
  class="org.springframework.web.servlet.view.InternalResourceViewResolver">
  <property name="prefix">

    <value>/WEB-INF/views/</value>
  </property>
  <property name="suffix">

    <value>.jsp</value>
  </property>
</bean>
```

- Define your **Spring Configuration**
  - Component Scan
  - View Resolver
  - ....

# World Before Spring Boot - 4 - NFRs

```
<plugin>
  <groupId>org.apache.tomcat.maven</groupId>
  <artifactId>tomcat7-maven-plugin</artifactId>
  <version>2.2</version>
  <configuration>
    <path>/</path>
    <contextReloadable>true</contextReloadable>
  </configuration>
</plugin>

<dependency>
  <groupId>log4j</groupId>
  <artifactId>log4j</artifactId>
  <version>1.2.17</version>
</dependency>
```

- Logging
- Error Handling
- Monitoring

# World Before Spring Boot!

- Setting up Spring Projects **before Spring Boot** was NOT easy!
  - 1: Dependency Management (**pom.xml**)
  - 2: Define Web App Configuration (**web.xml**)
  - 3: Manage Spring Beans (**context.xml**)
  - 4: Implement Non Functional Requirements (NFRs)
- AND repeat this for every new project!
- Typically takes a **few days** to setup for each project (and countless hours to maintain)



# Understanding Power of Spring Boot

```
// http://localhost:8080/courses
[
  {
    "id": 1,
    "name": "Learn AWS",
    "author": "in28minutes"
  }
]
```

- **1:** Create a Spring Boot Project
- **2:** Build a simple REST API using Spring Boot

# What's the Most Important Goal of Spring Boot?

In **28**  
Minutes

- Help you build **PRODUCTION-READY** apps **QUICKLY**
  - Build **QUICKLY**
    - Spring Initializr
    - Spring Boot Starter Projects
    - Spring Boot Auto Configuration
    - Spring Boot DevTools
  - Be **PRODUCTION-READY**
    - Logging
    - Different Configuration for Different Environments
      - Profiles, ConfigurationProperties
    - Monitoring (Spring Boot Actuator)
    - ...



# Spring Boot

## BUILD QUICKLY

# Exploring Spring Boot Starter Projects

In 28  
Minutes



- I need a lot of frameworks to build application features:
  - **Build a REST API:** I need Spring, Spring MVC, Tomcat, JSON conversion...
  - **Write Unit Tests:** I need Spring Test, JUnit, Mockito, ...
- How can I group them and make it easy to build applications?
  - **Starter:** Convenient **dependency descriptors** for diff. features
- **Spring Boot** provides variety of starter projects:
  - **Web Application & REST API** - Spring Boot Starter Web (spring-webmvc, spring-web, spring-boot-starter-tomcat, spring-boot-starter-json)
  - **Unit Tests** - Spring Boot Starter Test
  - **Talk to database using JPA** - Spring Boot Starter Data JPA
  - **Talk to database using JDBC** - Spring Boot Starter JDBC
  - **Secure your web application or REST API** - Spring Boot Starter Security
- (REMEMBER) **Starter:** Define all application dependencies

# Exploring Spring Boot Auto Configuration

- I need **lot of configuration** to build Spring app:
  - Component Scan, DispatcherServlet, Data Sources, JSON Conversion, ...
- How can I simplify this?
  - **Auto Configuration: Automated configuration** for your app
    - Decided based on:
      - Which frameworks are in the Class Path?
      - What is the existing configuration (Annotations etc)?
- **Example: Spring Boot Starter Web**
  - Dispatcher Servlet (DispatcherServletAutoConfiguration)
  - Embedded Servlet Container - Tomcat is the default (EmbeddedWebServerFactoryCustomizerAutoConfiguration)
  - Default Error Pages (ErrorMvcAutoConfiguration)
  - Bean<->JSON (JacksonHttpMessageConvertersConfiguration)

```
spring-boot-autoconfigure-2.4.4.jar - /Users/rangakaranam/.m2/re
└─ org.springframework.boot.autoconfigure
└─ org.springframework.boot.autoconfigure.admin
└─ org.springframework.boot.autoconfigure.amqp
└─ org.springframework.boot.autoconfigure.aop
└─ org.springframework.boot.autoconfigure.availability
└─ org.springframework.boot.autoconfigure.batch
└─ org.springframework.boot.autoconfigure.cache
└─ org.springframework.boot.autoconfigure.cassandra
└─ org.springframework.boot.autoconfigure.codec
└─ org.springframework.boot.autoconfigure.condition
└─ org.springframework.boot.autoconfigure.context
└─ org.springframework.boot.autoconfigure.couchbase
└─ org.springframework.boot.autoconfigure.dao
└─ org.springframework.boot.autoconfigure.data
└─ org.springframework.boot.autoconfigure.data.cassandra
└─ org.springframework.boot.autoconfigure.data.couchbase
└─ org.springframework.boot.autoconfigure.data.elasticsearch
└─ org.springframework.boot.autoconfigure.data.jdbc
└─ org.springframework.boot.autoconfigure.data.jpa
└─ org.springframework.boot.autoconfigure.data.ldap
└─ org.springframework.boot.autoconfigure.data.mongo
└─ org.springframework.boot.autoconfigure.data.neo4j
└─ org.springframework.boot.autoconfigure.data.r2dbc
└─ org.springframework.boot.autoconfigure.data.redis
└─ org.springframework.boot.autoconfigure.data.rest
└─ org.springframework.boot.autoconfigure.data.solr
└─ org.springframework.boot.autoconfigure.data.web
└─ org.springframework.boot.autoconfigure.diagnostics.analyzer
└─ org.springframework.boot.autoconfigure.domain
└─ org.springframework.boot.autoconfigure.elasticsearch
└─ org.springframework.boot.autoconfigure.elasticsearch.rest
└─ org.springframework.boot.autoconfigure.flyway
└─ org.springframework.boot.autoconfigure.freemarker
└─ org.springframework.boot.autoconfigure.groovy.template
└─ org.springframework.boot.autoconfigure.gson
└─ org.springframework.boot.autoconfigure.h2
└─ org.springframework.boot.autoconfigure.hateoas
└─ org.springframework.boot.autoconfigure.hazelcast
└─ org.springframework.boot.autoconfigure.http
└─ org.springframework.boot.autoconfigure.http.codec
```



# Understanding the Glue - @SpringBootApplication

- Questions:
  - Who is launching the Spring Context?
  - Who is triggering the component scan?
  - Who is enabling auto configuration?
- Answer: **@SpringBootApplication**
  - 1: **@SpringBootConfiguration**: Indicates that a class provides Spring Boot application @Configuration.
  - 2: **@EnableAutoConfiguration**: Enable auto-configuration of the Spring Application Context,
  - 3: **@ComponentScan**: Enable component scan (for current package, by default)

```
spring-boot-autoconfigure-2.4.4.jar - /Users/rangakaranam/.m2/re
> org.springframework.boot.autoconfigure
> org.springframework.boot.autoconfigure.admin
> org.springframework.boot.autoconfigure.amqp
> org.springframework.boot.autoconfigure.aop
> org.springframework.boot.autoconfigure.availability
> org.springframework.boot.autoconfigure.batch
> org.springframework.boot.autoconfigure.cache
> org.springframework.boot.autoconfigure.cassandra
> org.springframework.boot.autoconfigure.codec
> org.springframework.boot.autoconfigure.condition
> org.springframework.boot.autoconfigure.context
> org.springframework.boot.autoconfigure.couchbase
> org.springframework.boot.autoconfigure.dao
> org.springframework.boot.autoconfigure.data
> org.springframework.boot.autoconfigure.data.cassandra
> org.springframework.boot.autoconfigure.data.couchbase
> org.springframework.boot.autoconfigure.data.elasticsearch
> org.springframework.boot.autoconfigure.data.jdbc
> org.springframework.boot.autoconfigure.data.jpa
> org.springframework.boot.autoconfigure.data.ldap
> org.springframework.boot.autoconfigure.data.mongo
> org.springframework.boot.autoconfigure.data.neo4j
> org.springframework.boot.autoconfigure.data.r2dbc
> org.springframework.boot.autoconfigure.data.redis
> org.springframework.boot.autoconfigure.data.rest
> org.springframework.boot.autoconfigure.data.solr
> org.springframework.boot.autoconfigure.data.web
> org.springframework.boot.autoconfigure.diagnostics.analyzer
> org.springframework.boot.autoconfigure.domain
> org.springframework.boot.autoconfigure.elasticsearch
> org.springframework.boot.autoconfigure.elasticsearch.rest
> org.springframework.boot.autoconfigure.flyway
> org.springframework.boot.autoconfigure.freemarker
> org.springframework.boot.autoconfigure.groovy.template
> org.springframework.boot.autoconfigure.gson
> org.springframework.boot.autoconfigure.h2
> org.springframework.boot.autoconfigure.hateoas
> org.springframework.boot.autoconfigure.hazelcast
> org.springframework.boot.autoconfigure.http
> org.springframework.boot.autoconfigure.http.codec
```

# Build Faster with Spring Boot DevTools

In 28  
Minutes

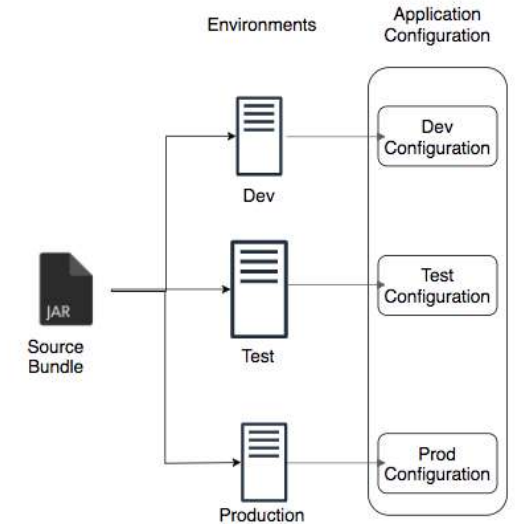
- Increase developer productivity
- Why do you need to restart the server **manually** for every code change?
- **Remember:** For pom.xml dependency changes, you will need to restart server **manually**



# Spring Boot PRODUCTION-READY

# Managing App. Configuration using Profiles

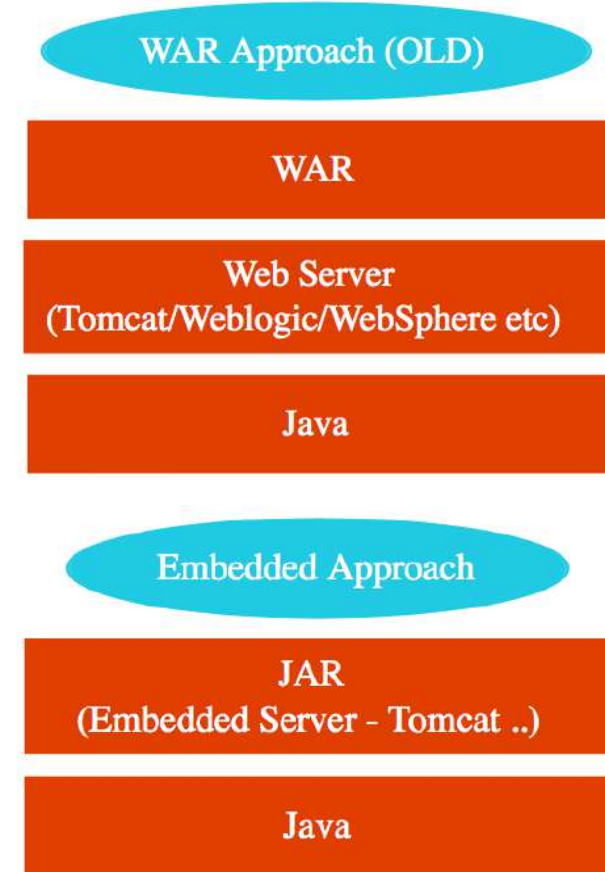
- Applications have different environments: **Dev, QA, Stage, Prod, ...**
- Different environments need **different configuration**:
  - Different Databases
  - Different Web Services
- How can you provide different configuration for different environments?
  - **Profiles**: Environment specific configuration
- How can you define externalized configuration for your application?
  - **ConfigurationProperties**: Define externalized configuration



# Simplify Deployment with Spring Boot Embedded Servers

28  
Minutes

- How do you deploy your application?
  - Step 1 : Install Java
  - Step 2 : Install Web/Application Server
    - Tomcat/WebSphere/WebLogic etc
  - Step 3 : Deploy the application WAR (Web ARchive)
    - This is the OLD **WAR** Approach
    - Complex to setup!
- **Embedded Server** - Simpler alternative
  - Step 1 : Install Java
  - Step 2 : Run **JAR** file
  - **Make JAR not WAR** (Credit: Josh Long!)
  - **Embedded Server Examples:**
    - spring-boot-starter-tomcat
    - spring-boot-starter-jetty
    - spring-boot-starter-undertow



# Monitor Applications using Spring Boot Actuator

In **28**  
Minutes

- Monitor and manage your application in your production
- Provides a number of endpoints:
  - **beans** - Complete list of Spring beans in your app
  - **health** - Application health information
  - **metrics** - Application metrics
  - **mappings** - Details around Request Mappings



# Understanding Spring Boot vs Spring MVC vs Spring

- **Spring Boot vs Spring MVC vs Spring: What's in it?**
  - **Spring Framework: Dependency Injection**
    - @Component, @Autowired, Component Scan etc..
    - Just Dependency Injection is NOT sufficient (You need other frameworks to build apps)
      - **Spring Modules and Spring Projects:** Extend Spring Eco System
        - Provide good integration with other frameworks (Hibernate/JPA, JUnit & Mockito for Unit Testing)
  - **Spring MVC (Spring Module): Simplify building web apps and REST API**
    - Building web applications with Struts was very complex
    - @Controller, @RestController, @RequestMapping("/courses")
  - **Spring Boot (Spring Project): Build **PRODUCTION-READY** apps **QUICKLY****
    - **Starter Projects** - Make it easy to build variety of applications
    - **Auto configuration** - Eliminate configuration to setup Spring, Spring MVC and other frameworks!
    - Enable non functional requirements (NFRs):
      - **Actuator:** Enables Advanced Monitoring of applications
      - **Embedded Server:** No need for separate application servers!
      - Logging and Error Handling
      - Profiles and ConfigurationProperties

# Spring Boot - Review

In 28  
Minutes

- **Goal:** 10,000 Feet overview of Spring Boot
  - Help you understand the terminology!
    - Starter Projects
    - Auto Configuration
    - Actuator
    - DevTools
- **Advantages:** Get started quickly with production ready features!





# Building REST API with Spring Boot

# Building REST API with Spring Boot - Goals

In 28  
Minutes

- **WHY** Spring Boot?
  - You can build REST API WITHOUT Spring Boot
  - What is the need for Spring Boot?
- **HOW** to build a great REST API?
  - Identifying Resources (/users, /users/{id}/posts)
  - Identifying Actions (GET, POST, PUT, DELETE, ...)
  - Defining Request and Response structures
  - Using appropriate Response Status (200, 404, 500, ..)
  - Understanding REST API Best Practices
    - Thinking from the perspective of your consumer
    - Validation, Internationalization - i18n, Exception Handling, HATEOAS, Versioning, Documentation, Content Negotiation and a lot more!



```
localhost:8080/users
[
  {
    "id": 1,
    "name": "Adam",
    "birthDate": "2022-08-16"
  },
  {
    "id": 2,
    "name": "Eve",
    "birthDate": "2022-08-16"
  },
  {
    "id": 3,
    "name": "Jack",
    "birthDate": "2022-08-16"
  }
]
```

# Building REST API with Spring Boot - Approach

In 28  
Minutes

- **1: Build 3 Simple Hello World REST API**
  - Understand the magic of Spring Boot
  - Understand fundamentals of building REST API with Spring Boot
    - @RestController, @RequestMapping, @PathVariable, JSON conversion
- **2: Build a REST API for a Social Media Application**
  - Design and Build a Great REST API
    - Choosing the right URI for resources (/users, /users/{id}, /users/{id}/posts)
    - Choosing the right request method for actions (GET, POST, PUT, DELETE, ..)
    - Designing Request and Response structures
    - Implementing Security, Validation and Exception Handling
  - Build Advanced REST API Features
    - Internationalization, HATEOAS, Versioning, Documentation, Content Negotiation, ...
- **3: Connect your REST API to a Database**
  - Fundamentals of JPA and Hibernate
  - Use H2 and MySQL as databases



```
localhost:8080/users
[
  {
    "id": 1,
    "name": "Adam",
    "birthDate": "2022-08-16"
  },
  {
    "id": 2,
    "name": "Eve",
    "birthDate": "2022-08-16"
  },
  {
    "id": 3,
    "name": "Jack",
    "birthDate": "2022-08-16"
  }
]
```

# What's Happening in the Background?



- Let's explore some **Spring Boot Magic**: Enable Debug Logging
  - **WARNING**: Log change frequently!
- **1**: How are our requests handled?
  - **DispatcherServlet** - Front Controller Pattern
    - Mapping servlets: `dispatcherServlet urls= [/]`
    - Auto Configuration (`DispatcherServletAutoConfiguration`)
- **2**: How does **HelloWorldBean** object get converted to JSON?
  - `@ResponseBody` + `JacksonHttpMessageConverters`
    - Auto Configuration (`JacksonHttpMessageConvertersConfiguration`)
- **3**: Who is configuring error mapping?
  - Auto Configuration (`ErrorMvcAutoConfiguration`)
- **4**: How are all jars available(Spring, Spring MVC, Jackson, Tomcat)?
  - **Starter Projects** - Spring Boot Starter Web (`spring-webmvc`, `spring-web`, `spring-boot-starter-tomcat`, `spring-boot-starter-json`)

# Social Media Application REST API

- Build a REST API for a Social Media Application
- **Key Resources:**
  - Users
  - Posts
- **Key Details:**
  - User: id, name, birthDate
  - Post: id, description

```
localhost:8080/users
[
  {
    "id": 1,
    "name": "Adam",
    "birthDate": "2022-08-16"
  },
  {
    "id": 2,
    "name": "Eve",
    "birthDate": "2022-08-16"
  },
  {
    "id": 3,
    "name": "Jack",
    "birthDate": "2022-08-16"
  }
]
```

# Request Methods for REST API

- **GET** - Retrieve details of a resource
- **POST** - Create a new resource
- **PUT** - Update an existing resource
- **PATCH** - Update part of a resource
- **DELETE** - Delete a resource

```
localhost:8080/users
[
  {
    "id": 1,
    "name": "Adam",
    "birthDate": "2022-08-16"
  },
  {
    "id": 2,
    "name": "Eve",
    "birthDate": "2022-08-16"
  },
  {
    "id": 3,
    "name": "Jack",
    "birthDate": "2022-08-16"
  }
]
```

# Social Media Application - Resources & Methods

- **Users REST API**

- Retrieve all Users
  - GET /users
- Create a User
  - POST /users
- Retrieve one User
  - GET /users/{id} -> /users/1
- Delete a User
  - DELETE /users/{id} -> /users/1

- **Posts REST API**

- Retrieve all posts for a User
  - GET /users/{id}/posts
- Create a post for a User
  - POST /users/{id}/posts
- Retrieve details of a post
  - GET /users/{id}/posts/{post\_id}

```
localhost:8080/users
[
  {
    "id": 1,
    "name": "Adam",
    "birthDate": "2022-08-16"
  },
  {
    "id": 2,
    "name": "Eve",
    "birthDate": "2022-08-16"
  },
  {
    "id": 3,
    "name": "Jack",
    "birthDate": "2022-08-16"
  }
]
```

# Response Status for REST API

- Return the **correct response status**
  - Resource is not found => 404
  - Server exception => 500
  - Validation error => 400
- **Important Response Statuses**
  - 200 — Success
  - 201 — Created
  - 204 — No Content
  - 401 — Unauthorized (when authorization fails)
  - 400 — Bad Request (such as validation error)
  - 404 — Resource Not Found
  - 500 — Server Error

```
localhost:8080/users
[
  {
    "id": 1,
    "name": "Adam",
    "birthDate": "2022-08-16"
  },
  {
    "id": 2,
    "name": "Eve",
    "birthDate": "2022-08-16"
  },
  {
    "id": 3,
    "name": "Jack",
    "birthDate": "2022-08-16"
  }
]
```



# Advanced REST API Features

- Documentation
- Content Negotiation
- Internationalization - i18n
- Versioning
- HATEOAS
- Static Filtering
- Dynamic Filtering
- Monitoring
- ....

```
localhost:8080/users
[
  {
    "id": 1,
    "name": "Adam",
    "birthDate": "2022-08-16"
  },
  {
    "id": 2,
    "name": "Eve",
    "birthDate": "2022-08-16"
  },
  {
    "id": 3,
    "name": "Jack",
    "birthDate": "2022-08-16"
  }
]
```

# REST API Documentation

- Your REST API consumers need to understand your REST API:
  - Resources
  - Actions
  - Request/Response Structure (Constraints/Validations)
- **Challenges:**
  - Accuracy: How do you ensure that your documentation is upto date and correct?
  - Consistency: You might have 100s of REST API in an enterprise. How do you ensure consistency?
- **Options:**
  - 1: Manually Maintain Documentation
    - Additional effort to keep it in sync with code
  - 2: Generate from code

The screenshot shows a REST API documentation interface for a GET endpoint. At the top, there is a blue button labeled 'GET' and the endpoint path '/jpa/users/{id}/posts'. Below this, a 'Parameters' section is visible, containing a table with columns 'Name' and 'Description'. A single parameter is listed: 'id' with a red asterisk indicating it is required. Its type is 'integer(\$int32)' and its location is '(path)'. The value 'id' is shown in a text input field. Below the parameters section, a 'Responses' section is visible, containing a table with columns 'Code' and 'Description'. A single response is listed: '200' with the description 'OK'. Below the response table, there is a 'Media type' dropdown menu set to 'application/hal+json'. Below this, there is a note 'Controls Accept header.' and a section for 'Example Value' and 'Schema'. The 'Example Value' section shows a JSON object: { "id": 0, "description": "string" }.

Name	Description
id * required	
integer(\$int32)	id
(path)	

Code	Description
200	OK

Media type: application/hal+json

Controls Accept header.

Example Value | Schema

```
[
  {
    "id": 0,
    "description": "string"
  }
]
```

# REST API Documentation - Swagger and Open API

- **Quick overview:**

- **2011:** Swagger Specification and Swagger Tools were introduced
- **2016:** Open API Specification created based on Swagger Spec.
  - Swagger Tools (ex:Swagger UI) continue to exist
- **OpenAPI Specification:** Standard, language-agnostic interface
  - Discover and understand REST API
  - Earlier called Swagger Specification
- **Swagger UI:** Visualize and interact with your REST API
  - Can be generated from your OpenAPI Specification

The screenshot displays the Swagger UI interface. On the left, the OpenAPI specification is shown in a code editor. The specification includes the following details:

- openapi:** "3.0.1"
- info:** { }
- servers:** [ ]
- paths:** {
  - /posts:** {
    - get:** { }
    - post:** { }
  - /posts/{id}:** {
    - get:** { }
    - put:** { }
    - delete:** { }
    - patch:** { }

On the right, the details for the **GET /jpa/users/{id}/posts** endpoint are shown. The **Parameters** section includes a required path parameter **id** of type **integer(\$int32)**. The **Responses** section shows a **200 OK** response with a media type of **application/hal+json**. An example value for the response is shown as:

```
[  
  {  
    "id": 0,  
    "description": "string"  
  }  
]
```

# Content Negotiation

- **Same Resource - Same URI**
  - **HOWEVER Different Representations** are possible
    - Example: Different Content Type - XML or JSON or ..
    - Example: Different Language - English or Dutch or ..
- How can a consumer tell the REST API provider what they want?
  - Content Negotiation
- Example: Accept header (MIME types - application/xml, application/json, ..)
- Example: Accept-Language header (en, nl, fr, ..)

```
localhost:8080/users
[
  {
    "id": 1,
    "name": "Adam",
    "birthDate": "2022-08-16"
  },
  {
    "id": 2,
    "name": "Eve",
    "birthDate": "2022-08-16"
  },
  {
    "id": 3,
    "name": "Jack",
    "birthDate": "2022-08-16"
  }
]

<List>
  <item>
    <id>2</id>
    <name>Eve</name>
    <birthDate>1987-07-19</birthDate>
  </item>
  <item>
    <id>3</id>
    <name>Jack</name>
    <birthDate>1997-07-19</birthDate>
  </item>
  <item>
    <id>4</id>
    <name>Ranga</name>
    <birthDate>2007-07-19</birthDate>
  </item>
</List>
```

# Internationalization - i18n

- Your REST API might have consumers from around the world
- How do you customize it to users around the world?
  - Internationalization - i18n
- Typically **HTTP Request Header - Accept-Language** is used
  - Accept-Language - indicates natural language and locale that the consumer prefers
  - Example: en - English (Good Morning)
  - Example: nl - Dutch (Goedemorgen)
  - Example: fr - French (Bonjour)
  - Example: de - Deutsch (Guten Morgen)

The screenshot displays two instances of a REST client interface, likely Swagger Client, demonstrating how the 'Accept-Language' header affects the response body of a GET request to 'http://localhost:8080/hello-world-internationalized'.

**Top Screenshot (French Request):**

- Method:** GET
- URL:** http://localhost:8080/hello-world-internationalized
- Headers:** Accept-Language: fr
- Response:** 200, Body: Bonjour

**Bottom Screenshot (Dutch Request):**

- Method:** GET
- URL:** http://localhost:8080/hello-world-internationalized
- Headers:** Accept-Language: nl
- Response:** 200, Body: Goede Morgen

# Versioning REST API

- You have built an amazing REST API
  - You have 100s of consumers
  - You need to implement a breaking change
    - Example: Split name into firstName and lastName
- **SOLUTION: Versioning REST API**
  - **Variety of options**
    - URL
    - Request Parameter
    - Header
    - Media Type
  - **No Clear Winner!**

localhost:8080/v1/person

```
{  
  "name": "Bob Charlie"  
}
```

localhost:8080/v2/person

```
{  
  "name": {  
    "firstName": "Bob",  
    "lastName": "Charlie"  
  }  
}
```

# Versioning REST API - Options

- **URI Versioning - Twitter**
  - *http://localhost:8080/v1/person*
  - *http://localhost:8080/v2/person*
- **Request Parameter versioning - Amazon**
  - *http://localhost:8080/person?version=1*
  - *http://localhost:8080/person?version=2*
- **(Custom) headers versioning - Microsoft**
  - SAME-URL headers=[X-API-VERSION=1]
  - SAME-URL headers=[X-API-VERSION=2]
- **Media type versioning (a.k.a “content negotiation” or “accept header”) - GitHub**
  - SAME-URL produces=application/vnd.company.app-v1+json
  - SAME-URL produces=application/vnd.company.app-v2+json

```
localhost:8080/v2/person
{
  "name": {
    "firstName": "Bob",
    "lastName": "Charlie"
  }
}
```

# Versioning REST API - Factors

- **Factors to consider**
  - URI Pollution
  - Misuse of HTTP Headers
  - Caching
  - Can we execute the request on the browser?
  - API Documentation
  - Summary: No Perfect Solution
- **My Recommendations**
  - Think about versioning even before you need it!
  - One Enterprise - One Versioning Approach

## URI Versioning - Twitter

- `http://localhost:8080/v1/person`
- `http://localhost:8080/v2/person`

## Request Parameter versioning - Amazon

- `http://localhost:8080/person?version=1`
- `http://localhost:8080/person?version=2`

## (Custom) headers versioning - Microsoft

- SAME-URL headers=[X-API-VERSION=1]
- SAME-URL headers=[X-API-VERSION=2]

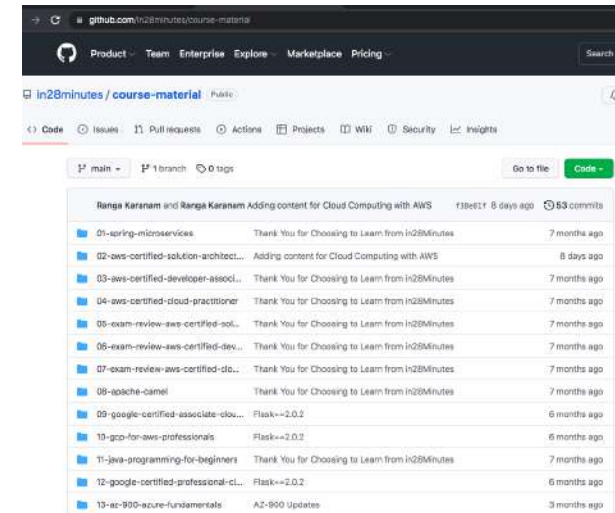
## Media type versioning - GitHub

- SAME-URL produces=application/vnd.company.app-v1+json
- SAME-URL produces=application/vnd.company.app-v2+json



# HATEOAS

- Hypermedia as the Engine of Application State (HATEOAS)
- Websites allow you to:
  - See **Data** AND Perform **Actions** (using links)
- How about enhancing your REST API to tell consumers how to perform subsequent actions?
  - HATEOAS
- **Implementation Options:**
  - **1: Custom Format and Implementation**
    - Difficult to maintain
  - **2: Use Standard Implementation**
    - **HAL (JSON Hypertext Application Language):** Simple format that gives a consistent and easy way to hyperlink between resources in your API
    - **Spring HATEOAS:** Generate HAL responses with hyperlinks to resources



```
{
  "name": "Adam",
  "birthDate": "2022-08-16",
  "_links": {
    "all-users": {
      "href": "http://localhost:8080/users"
    }
  }
}
```

# Customizing REST API Responses - Filtering and more..

- **Serialization:** Convert object to stream (example: JSON)
  - Most popular JSON Serialization in Java: Jackson
- How about customizing the REST API response returned by Jackson framework?
- **1:** Customize field names in response
  - @JsonProperty
- **2:** Return only selected fields
  - **Filtering**
  - Example: Filter out Passwords
  - **Two types:**
    - **Static Filtering:** Same filtering for a bean across different REST API
      - @JsonIgnoreProperties, @JsonIgnore
    - **Dynamic Filtering:** Customize filtering for a bean for specific REST API
      - @JsonFilter with FilterProvider

```
localhost:8080/filtering-list
[
  {
    "field2": "value2",
    "field3": "value3"
  },
  {
    "field2": "value5",
    "field3": "value6"
  }
]

localhost:8080/filtering
{
  "field1": "value1",
  "field3": "value3"
}
```

# Get Production-ready with Spring Boot Actuator

In 28  
Minutes

- **Spring Boot Actuator:** Provides Spring Boot's production-ready features
  - Monitor and manage your application in your production
- **Spring Boot Starter Actuator:** Starter to add Spring Boot Actuator to your application
  - `spring-boot-starter-actuator`
- Provides a number of endpoints:
  - **beans** - Complete list of Spring beans in your app
  - **health** - Application health information
  - **metrics** - Application metrics
  - **mappings** - Details around Request Mappings
  - and a lot more .....



# Explore REST API using HAL Explorer

In **28**  
Minutes

- **1: HAL (JSON Hypertext Application Language)**
  - Simple format that gives a consistent and easy way to hyperlink between resources in your API
- **2: HAL Explorer**
  - An API explorer for RESTful Hypermedia APIs using HAL
  - Enable your non-technical teams to play with APIs
- **3: Spring Boot HAL Explorer**
  - Auto-configures HAL Explorer for Spring Boot Projects
  - `spring-data-rest-hal-explorer`

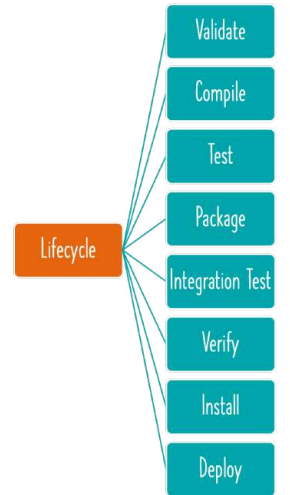


# Maven

# What is Maven?

In **28**  
Minutes

- **Things you do** when writing code each day:
  - Create new projects
  - Manages **dependencies** and their versions
  - Spring, Spring MVC, Hibernate,...
  - Add/modify dependencies
  - **Build** a JAR file
  - Run your application locally in Tomcat or Jetty or ..
  - Run **unit tests**
  - Deploy to a test environment
  - and a lot more..
- Maven helps you do all these and more...



# Exploring Project Object Model - pom.xml

- Let's explore Project Object Model - pom.xml
    - **1: Maven dependencies:** Frameworks & libraries used in a project
    - Ex: spring-boot-starter-web and spring-boot-starter-test
    - Why are there so many dependencies in the classpath?
      - Answer: Transitive Dependencies
- (REMEMBER) Spring dependencies are DIFFERENT
- **2: Parent Pom:** spring-boot-starter-parent
    - Dependency Management: spring-boot-dependencies
    - Properties: java.version, plugins and configurations
  - **3: Name of our project:** groupId + artifactId
- **1: groupId:** Similar to package name
  - **2: artifactId:** Similar to class name
  - **Why is it important?**

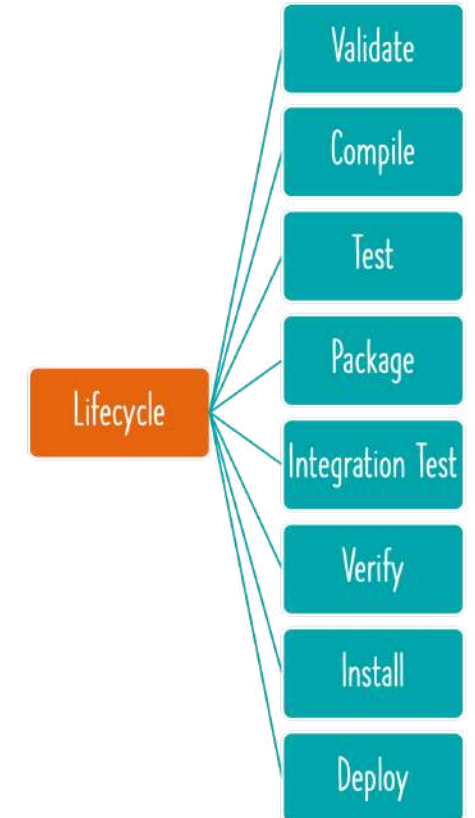
- Think about this: How can other projects use our new project?
  - **Activity:** help:effective-pom, dependency:tree & Eclipse UI
  - Let's add a new dependency: spring-boot-starter-web



# Exploring Maven Build Life Cycle

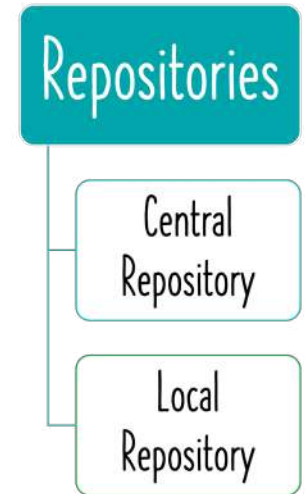
In 28  
Minutes

- When we run a maven command, maven build life cycle is used
- Build LifeCycle is a sequence of steps
  - Validate
  - Compile
  - Test
  - Package
  - Integration Test
  - Verify
  - Install
  - Deploy



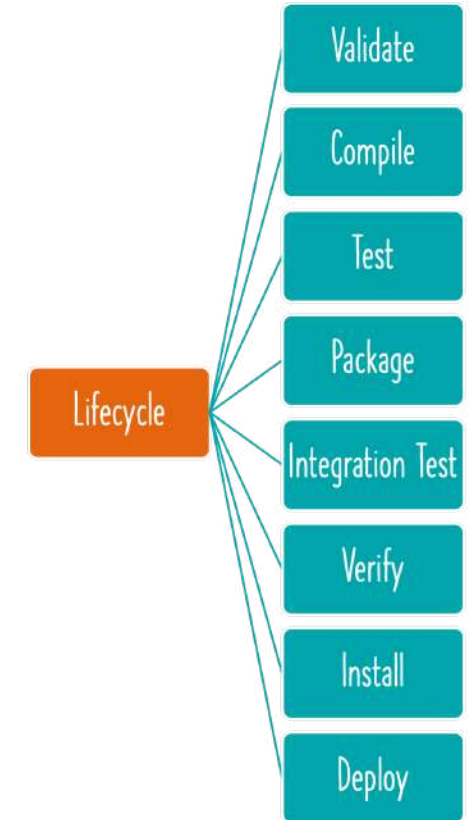
# How does Maven Work?

- Maven follows **Convention over Configuration**
  - Pre defined folder structure
  - Almost all Java projects follow **Maven structure** (Consistency)
- **Maven central repository** contains jars (and others) indexed by artifact id and group id
  - Stores all the versions of dependencies
  - repositories > repository
  - pluginRepositories > pluginRepository
- When a dependency is added to pom.xml, Maven tries to download the dependency
  - Downloaded dependencies are stored inside your maven local repository
  - **Local Repository** : a temp folder on your machine where maven stores the jar and dependency files that are downloaded from Maven Repository.



# Important Maven Commands

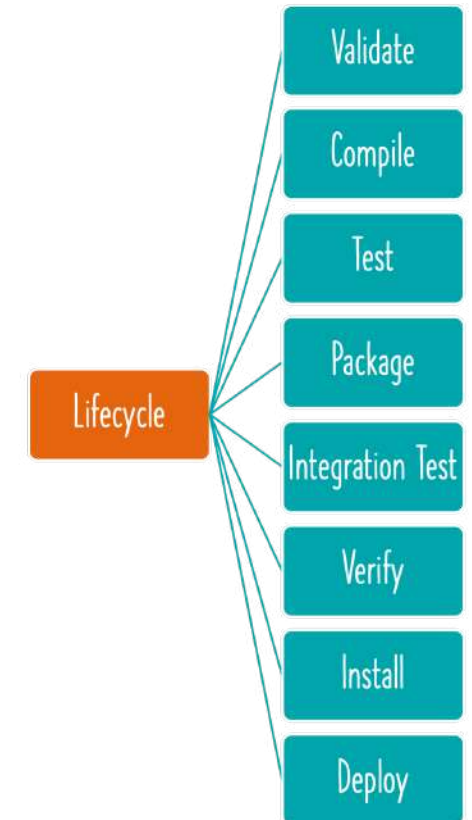
- `mvn --version`
- `mvn compile`: Compile source files
- `mvn test-compile`: Compile test files
  - OBSERVE CAREFULLY: This will also compile source files
- `mvn clean`: Delete target directory
- `mvn test`: Run unit tests
- `mvn package`: Create a jar
- `mvn help:effective-pom`
- `mvn dependency:tree`



# Spring Boot Maven Plugin

In **28**  
Minutes

- **Spring Boot Maven Plugin:** Provides Spring Boot support in Apache Maven
  - Example: Create executable jar package
  - Example: Run Spring Boot application
  - Example: Create a Container Image
  - **Commands:**
  - `mvn spring-boot:repackage` (create jar or war)
- **Run package using `java -jar`**
  - `mvn spring-boot:run` (Run application)
  - `mvn spring-boot:start` (Non-blocking. Use it to run integration tests.)



- `mvn spring-boot:stop` (Stop application started with start command)
- `mvn spring-boot:build-image` (Build a container image)

# How are Spring Releases Versioned?

In 28  
Minutes

**Maven**™

 **spring**®

- **Version scheme - MAJOR.MINOR.PATCH[-MODIFIER]**
  - **MAJOR:** Significant amount of work to upgrade (10.0.0 to 11.0.0)
  - **MINOR:** Little to no work to upgrade (10.1.0 to 10.2.0)
  - **PATCH:** No work to upgrade (10.5.4 to 10.5.5)
  - **MODIFIER:** Optional modifier
  - **Milestones** - M1, M2, .. (10.3.0-M1, 10.3.0-M2)
  - **Release candidates** - RC1, RC2, .. (10.3.0-RC1, 10.3.0-RC2)
  - **Snapshots** - SNAPSHOT
  - **Release** - Modifier will be ABSENT (10.0.0, 10.1.0)
- **Example versions in order:**
  - 10.0.0-SNAPSHOT, 10.0.0-M1, 10.0.0-M2, 10.0.0-RC1, 10.0.0-RC2, 10.0.0, ...
- **MY RECOMMENDATIONS:**
  - Avoid SNAPSHOTs

- Use ONLY Released versions in PRODUCTION

# Gradle





- **Goal:** Build, automate and deliver better software, faster
  - **Build Anything:** Cross-Platform Tool
    - Java, C/C++, JavaScript, Python, ...
  - **Automate Everything:** Completely Programmable
    - Complete flexibility
  - **Uses a DSL**
    - Supports Groovy and Kotlin
  - **Deliver Faster:** Blazing-fast builds
  - **Compile avoidance to advanced caching**
  - **Can speed up Maven builds by up to 90%**
- **Incrementality** — Gradle runs only what is necessary
- **Example:** Compiles only changed files

- **Build Cache** — Reuses the build outputs of other Gradle builds with the same inputs
- Same project layout as Maven
- IDE support still evolving



- Top 3 Java Plugins for Gradle:
  - **1: Java Plugin:** Java compilation + testing + bundling capabilities
    - **Default Layout**
      - `src/main/java`: Production Java source
      - `src/main/resources`: Production resources, such as XML and properties files
      - `src/test/java`: Test Java source
      - `src/test/resources`: Test resources
    - **Key Task:** `build`
  - **2: Dependency Management:** Maven-like dependency management
    - `group: 'org.springframework',` `name: 'spring-core',`  
`version: '10.0.3.RELEASE'` OR
    - Shortcut: `org.springframework:spring-core:10.0.3.RELEASE`
  - **3: Spring Boot Gradle Plugin:** Spring Boot support in Gradle
    - Package executable Spring Boot jar, Container Image (`bootJar`, `bootBuildImage`)
    - Use dependency management enabled by `spring-boot-dependencies`
      - No need to specify dependency version
      - Ex: `implementation('org.springframework.boot:spring-boot-starter')`

# Maven vs Gradle - Which one to Use?

In 28  
Minutes

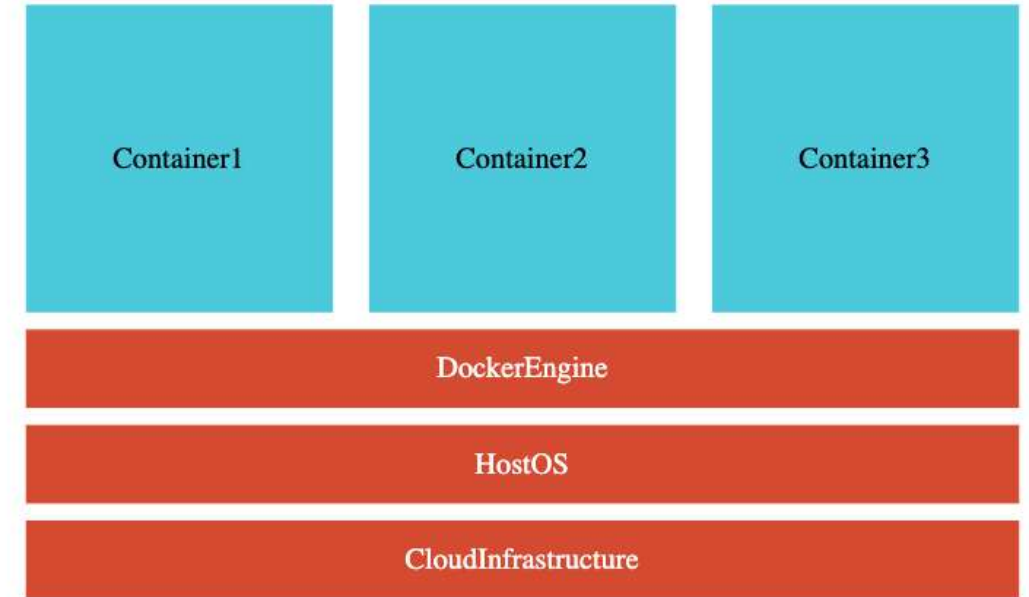
- Let's start with a few popular examples:
  - **Spring Framework** - Using Gradle since 2012 (Spring Framework v3.2.0)
  - **Spring Boot** - Using Gradle since 2020 (Spring Boot v2.3.0)
  - **Spring Cloud** - Continues to use Maven even today
  - Last update: Spring Cloud has no plans to switch
- **Top Maven Advantages:** Familiar, Simple and Restrictive
- **Top Gradle Advantages:** Faster build times and less verbose
- **What Do I Recommend:** I'm sitting on the fence for now
  - Choose whatever tool best meets your projects needs
    - If your builds are taking really long, go with Gradle
    - If your builds are simple, stick with Maven



# Microservices

# Microservices - V2

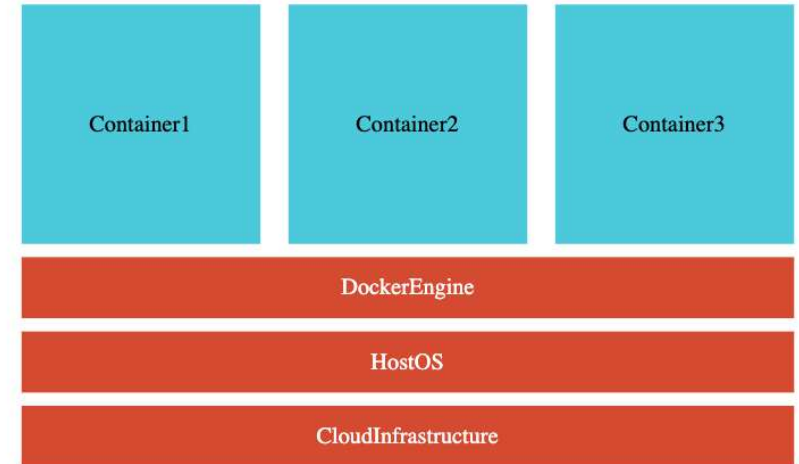
- **V2 (2.4+) - Latest Releases of**
  - Spring Boot
  - Spring Cloud
  - Docker and
  - Kubernetes
  - **Skip to Next Section :)**
- **V1 - Old Versions**
  - Spring Boot v2.3 and LOWER
  - **Continue on to next lecture :(**



# Microservices - Evolution

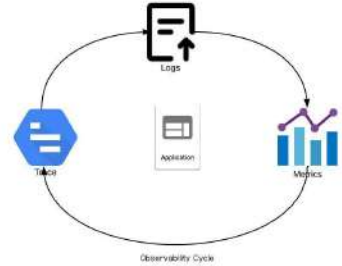
- **Goal: Evolve with Microservices**

- **V1** - Spring Boot 2.0.0 to 2.3.x
- **V2** - Spring Boot 2.4.0 to 3.0.0 to ...
  - Spring Cloud LoadBalancer (Ribbon)
  - Spring Cloud Gateway (Zuul)
  - Resilience4j (Hystrix)
  - NEW: Docker
  - NEW: Kubernetes
  - NEW: Observability
    - NEW: Micrometer (Spring Cloud Sleuth)
    - NEW: OpenTelemetry



# Microservices - Spring Boot 2 vs Spring Boot 3

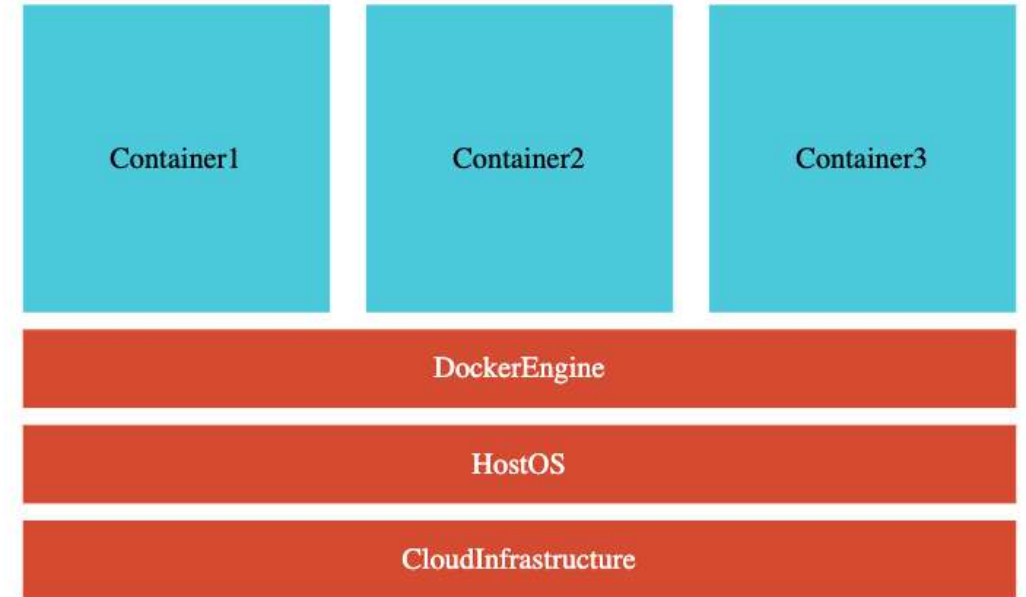
- V1(2.0.0 to 2.3.x)
- V2 (2.4.x to 3.0.0 to ..)
- Spring Boot 2.4.0+
  - <https://github.com/in28minutes/spring-microservices-v2>
- Spring Boot 3.0.0+
  - <https://github.com/in28minutes/spring-microservices-v3>
  - Notes: v3-upgrade.md
  - Key Changes:
    - **Observability** - Ability of a system to measure its current state based on the generated data
      - Monitoring is reactive while Observability is proactive
    - **OpenTelemetry**: One Standard for Logs + Traces + Metrics
      - Cross Language
      - Cross Platform
    - **Micrometer** (Replaces Spring Cloud Sleuth)
      - Collect (Logs + Traces + Metrics)





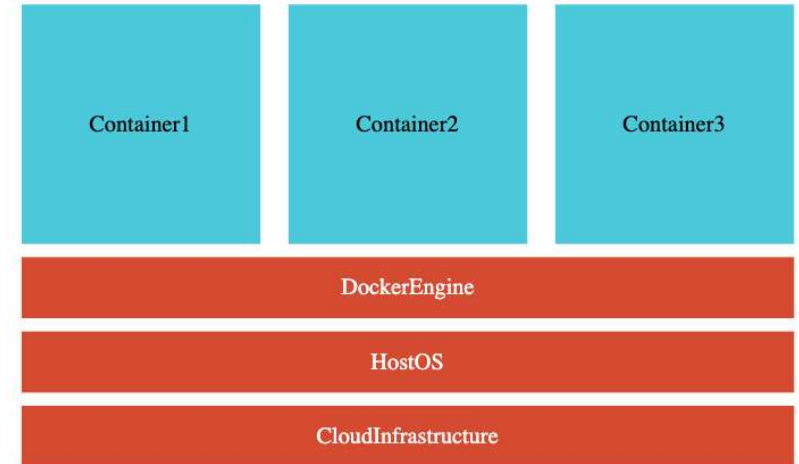
# Microservices - V2

- You have **skipped V1**
  - Go to next lecture!
- You have **completed V1**
  - Option 1: **Start from Zero Again:**
    - Go to the next lecture!
  - Option 2: **Get a Quick Start:**
    - Jump to "Step 21 - QuickStart by Importing Microservices"
      - Same microservices as V1: **Currency Exchange** and **Currency Conversion**
      - Very little changes in **Eureka Naming Server**
      - **Step 21** helps you set these up and get started quickly!



# Microservices - V2 - What's New

- Microservices Evolve Quickly
- V2 (Spring Boot - 2.4.x to 3.0.0 to LATEST)
  - Spring Cloud LoadBalancer instead of Ribbon
  - Spring Cloud Gateway instead of Zuul
  - Resilience4j instead of Hystrix
  - Docker: Containerize Microservices
    - Run microservices using Docker and Docker Compose
  - Kubernetes: Orchestrate all your Microservices with Kubernetes
  - OpenTelemetry: One Standard - Logs, Traces & Metrics
  - Micrometer (Replaces Spring Cloud Sleuth)

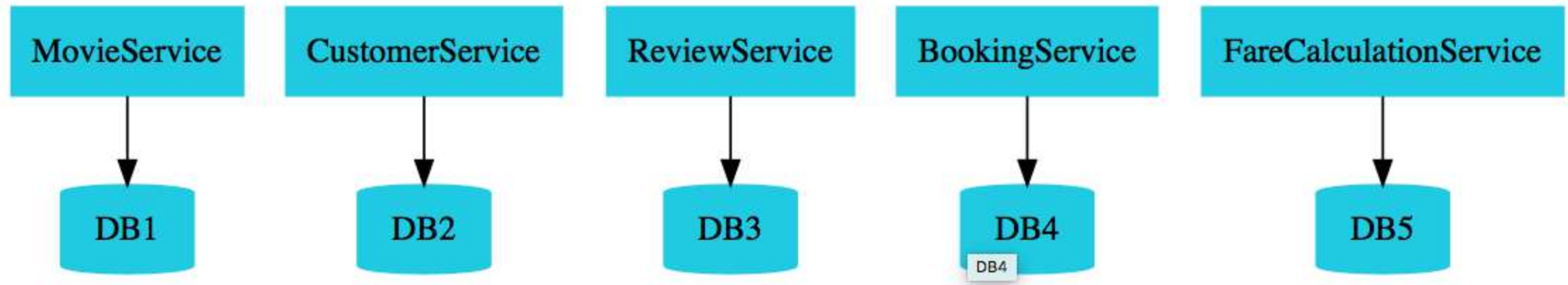


MovieApplication



LARGEDB

Monolith



## Microservices

# What is a Microservice?



*Small autonomous services that work together*

*Sam Newman*

# What is a Microservice?



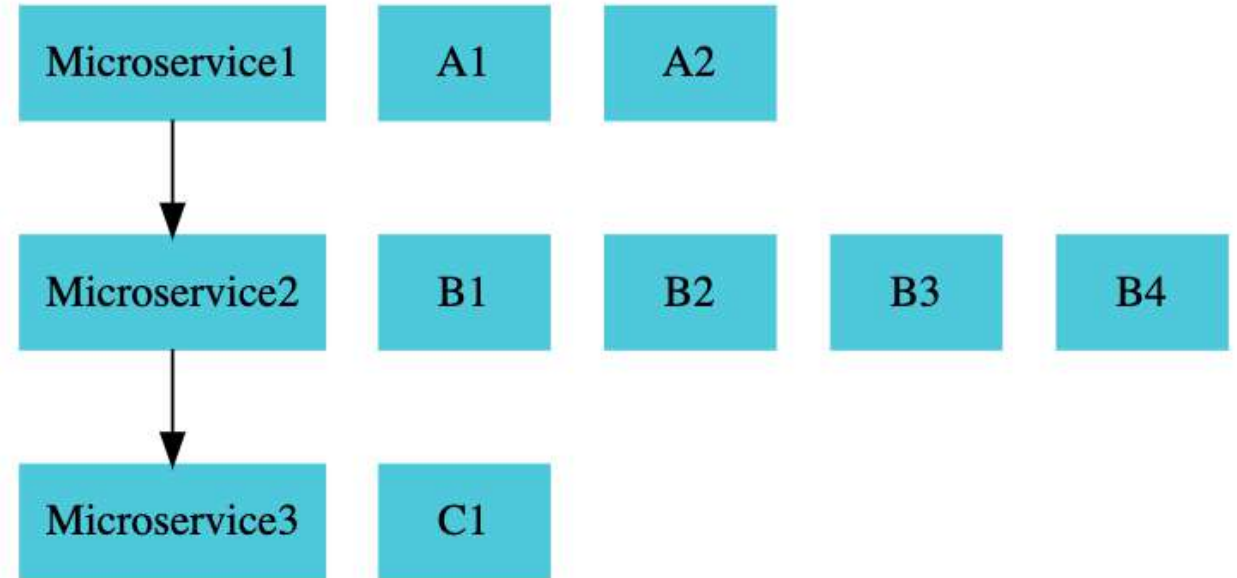
*Approach to developing a application as a suite of small services, each running in its own process and communicating with lightweight mechanisms often an HTTP resource API.*

*These services are built around business capabilities and independently deployable by fully automated deployment machinery. There is a bare minimum of centralized management of these services, which may be written in different programming languages and use different data storage technologies.*

***James Lewis and Martin Fowler***

# Microservices for me

- REST
- Small Well Chosen Deployable Units
- Cloud Enabled



# Microservices - Challenges



- Bounded Context
- Configuration Management
- Dynamic Scale Up and Scale Down
- Visibility
- Pack of Cards
- Zero Downtime Deployments



# Microservice - Solutions



- **Spring Cloud Umbrella Projects**
  - Centralized Configuration Management (Spring Cloud Config Server)
  - Location Transparency - Naming Server (Eureka)
  - Load Distribution (Ribbon, Spring Cloud Load Balancer)
  - Visibility and Monitoring (Zipkin)
  - API Gateway (Zuul, Spring Cloud Gateway)
  - Fault Tolerance (Hystrix, Resilience4j)
- **Docker:** Language Neutral, Cloud Neutral deployable units
- **Kubernetes:** Orchestrate Thousands of Microservices

# Microservices - 3 Key Advantages



- New Technology & Process Adoption
- Dynamic Scaling
- Faster Release Cycles

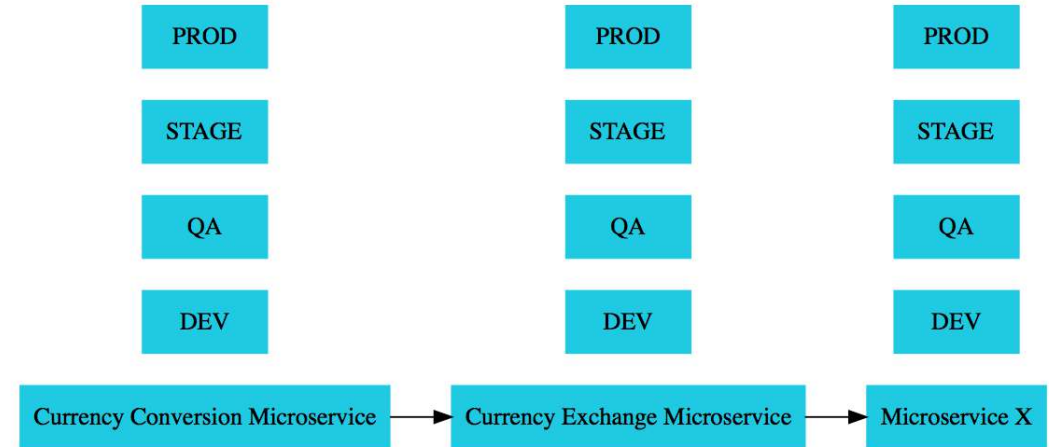
# Ports Standardization

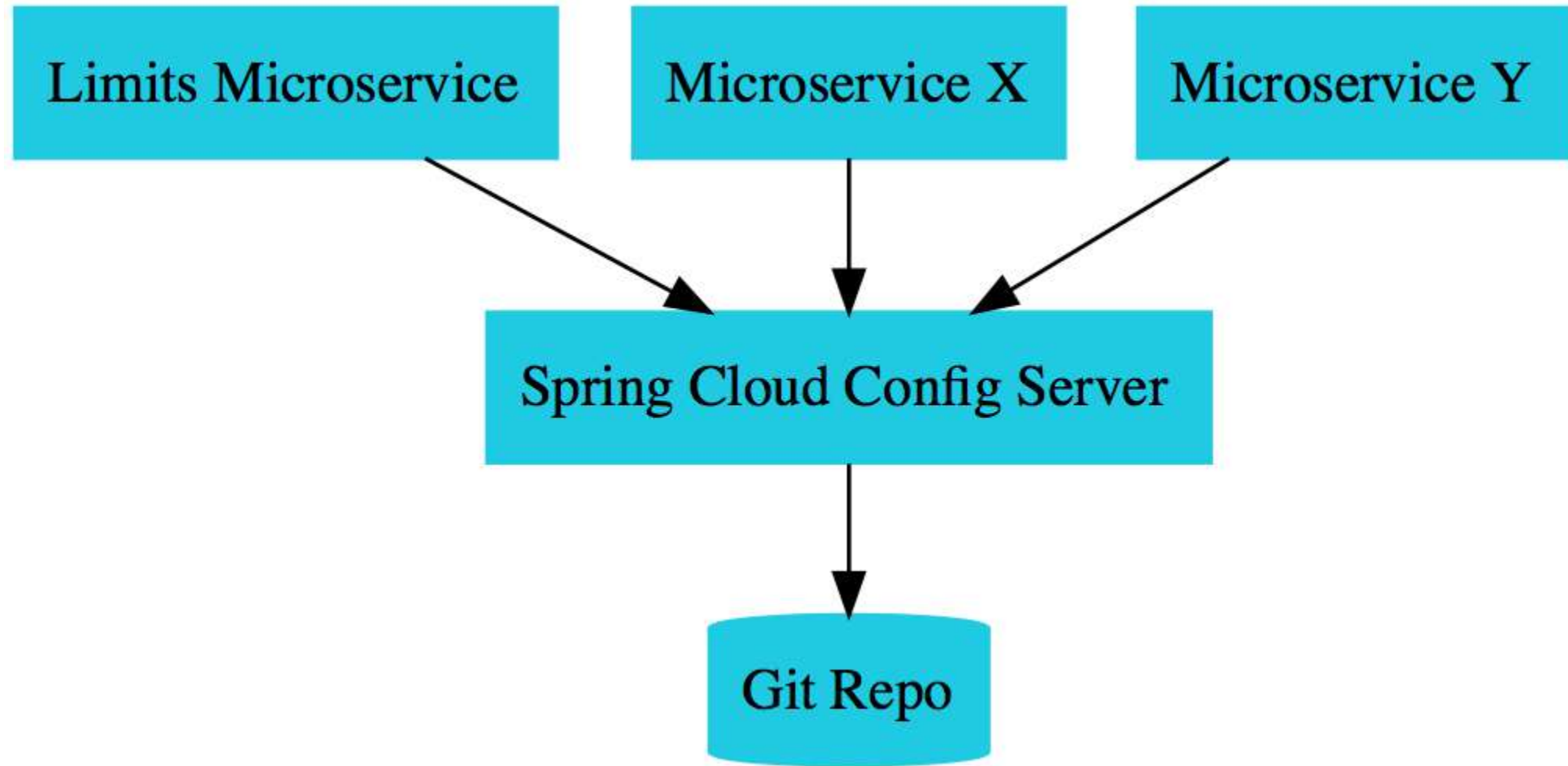
Application	Port
Limits Microservice	8080, 8081, ...
Spring Cloud Config Server	8888
Currency Exchange Microservice	8000, 8001, 8002, ..
Currency Conversion Microservice	8100, 8101, 8102, ...
Netflix Eureka Naming Server	8761
API Gateway	8765
Zipkin Distributed Tracing Server	9411

# Need for Centralized Configuration

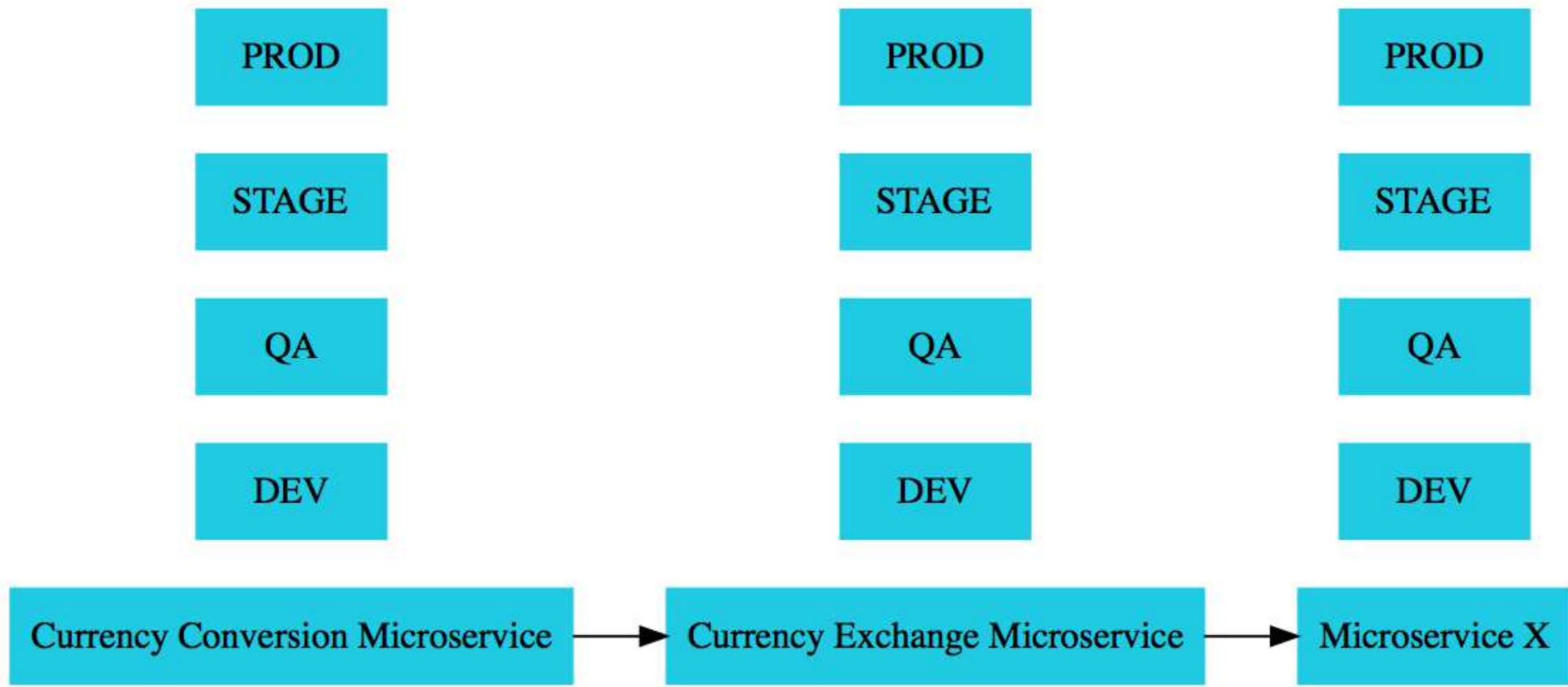
In 28  
Minutes

- Lot of configuration:
  - External Services
  - Database
  - Queue
  - Typical Application Configuration
- Configuration variations:
  - 1000s of Microservices
  - Multiple Environments
  - Multiple instances in each Environment
- How do you manage all this configuration?

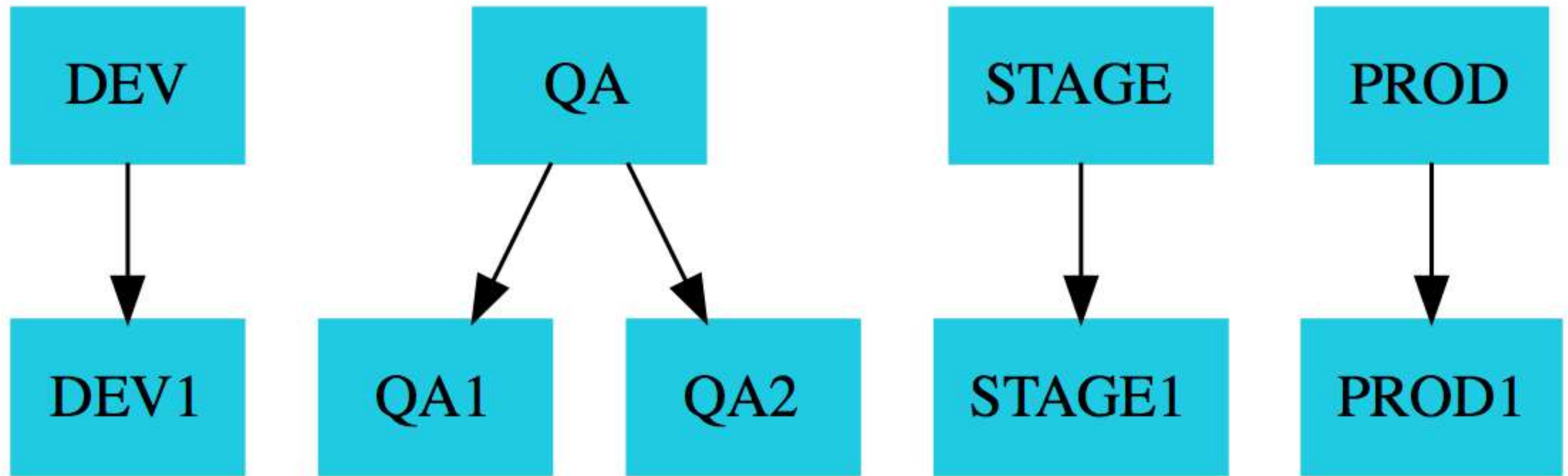




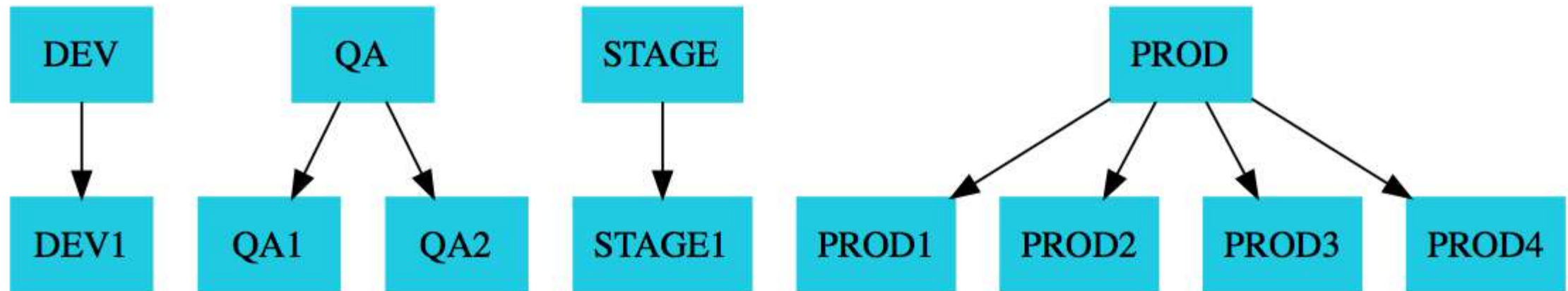
Config Server



Environments

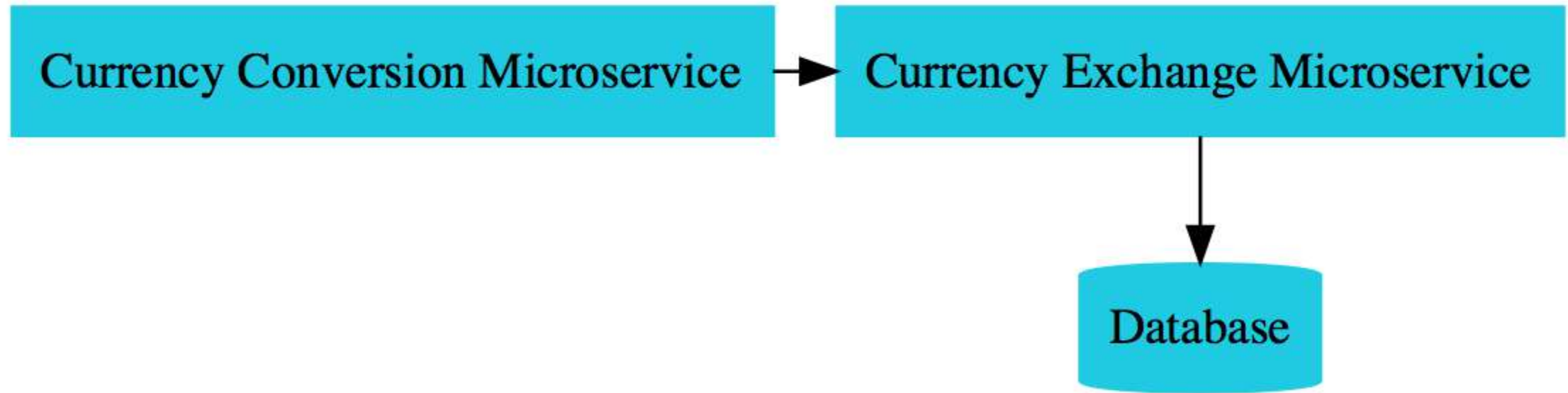


Environments



## Environments





## Microservices Overview

# Currency Exchange Microservice

*What is the exchange rate of one currency in another?*

```
http://localhost:8000/currency-exchange/from/USD/to/INR
```

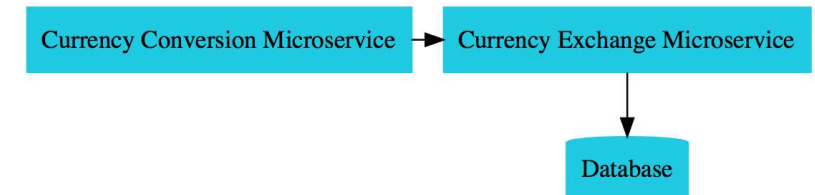
```
{  
  "id":10001,  
  "from":"USD",  
  "to":"INR",  
  "conversionMultiple":65.00,  
  "environment":"8000 instance-id"  
}
```

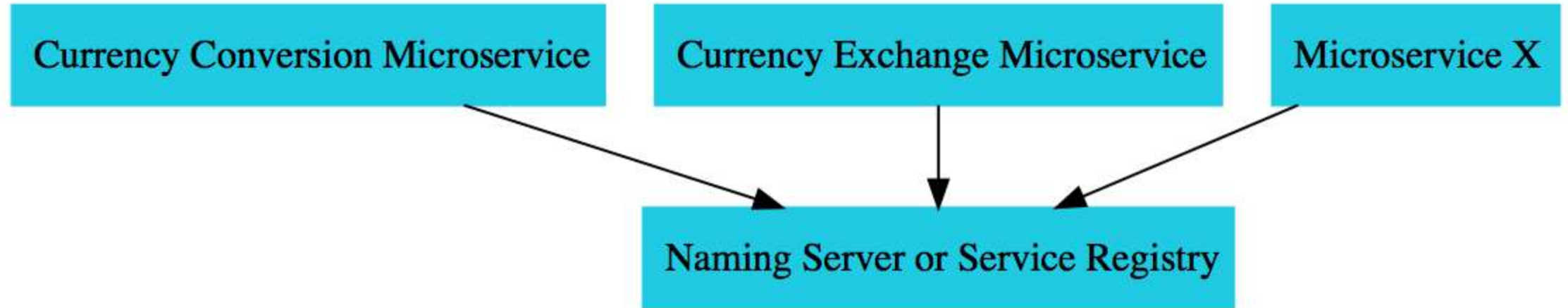
# Currency Conversion Microservice

*Convert 10 USD into INR*

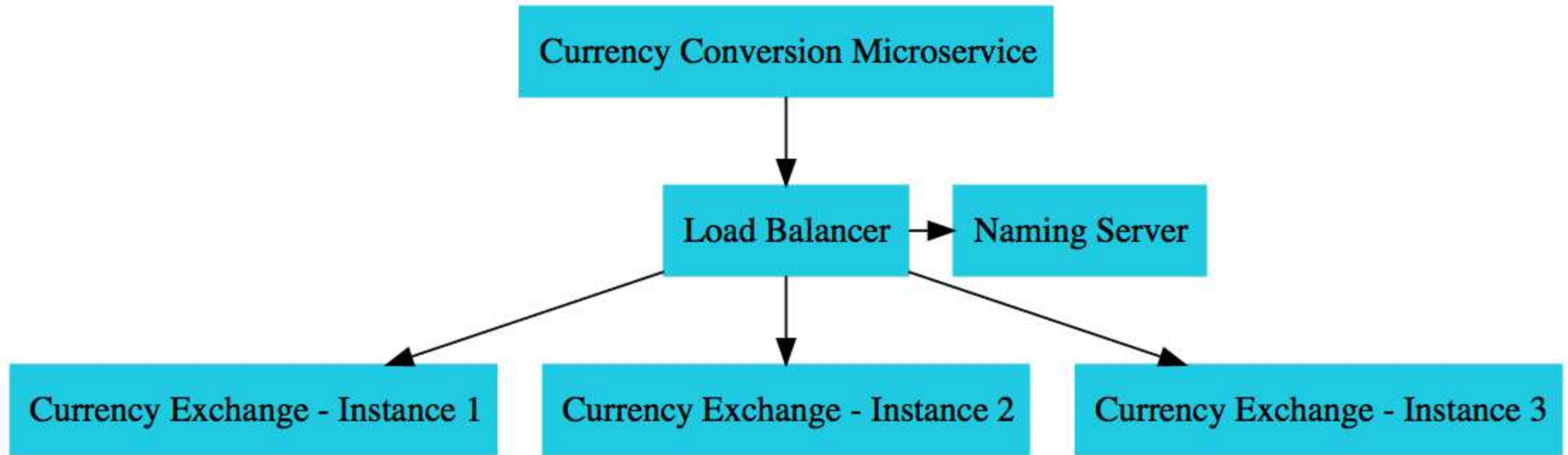
```
http://localhost:8100/currency-conversion/from/USD/to/INR/quantity/10
```

```
{  
  "id": 10001,  
  "from": "USD",  
  "to": "INR",  
  "conversionMultiple": 65.00,  
  "quantity": 10,  
  "totalCalculatedAmount": 650.00,  
  "environment": "8000 instance-id"  
}
```





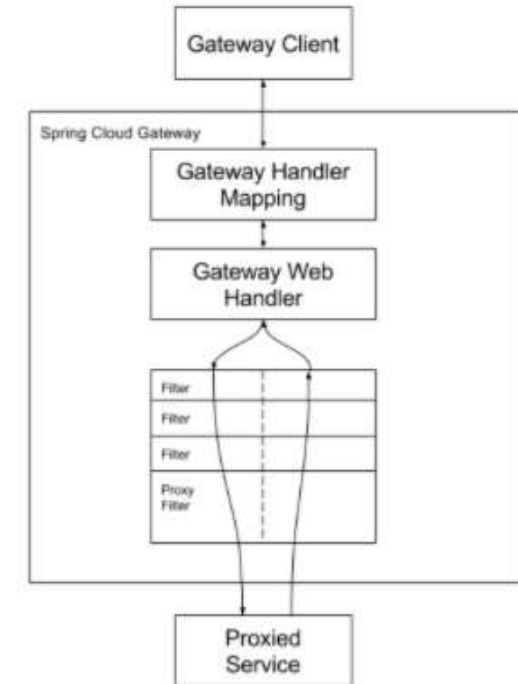
Naming Server



Load Balancing

# Spring Cloud Gateway

- Simple, yet effective way to route to APIs
- Provide cross cutting concerns:
  - Security
  - Monitoring/metrics
- Built on top of Spring WebFlux (Reactive Approach)
- Features:
  - Match routes on any request attribute
  - Define Predicates and Filters
  - Integrates with Spring Cloud Discovery Client (Load Balancing)
  - Path Rewriting



From <https://docs.spring.io>

# Circuit Breaker



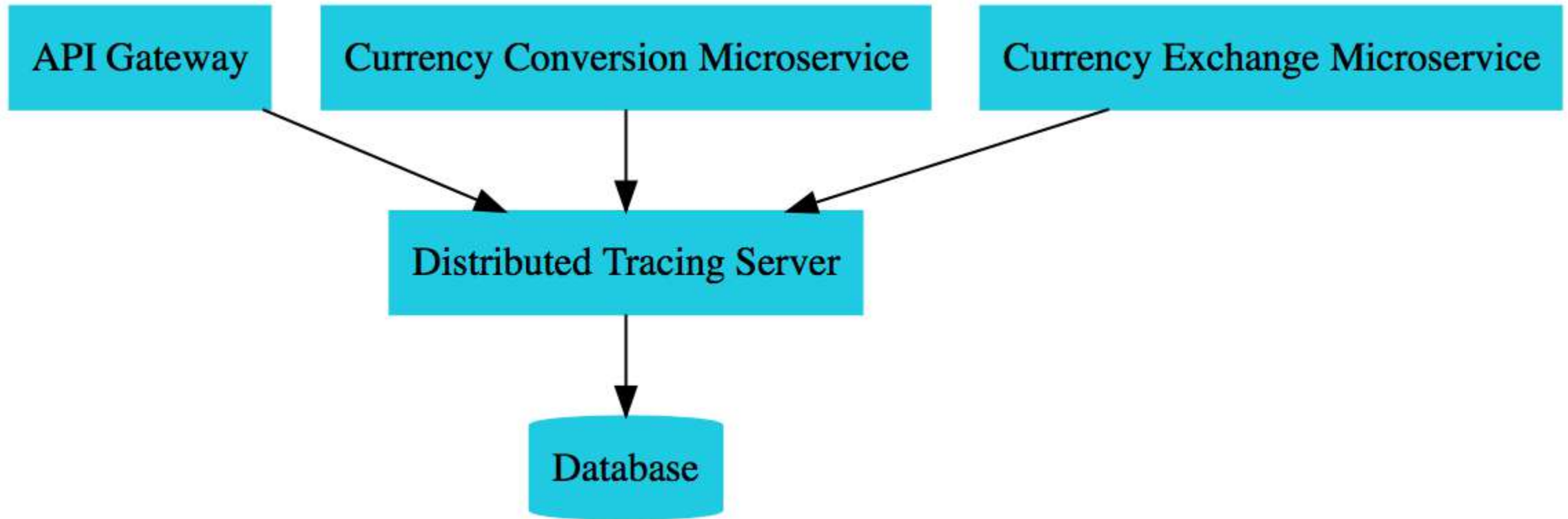
- What if one of the services is down or is slow?
  - Impacts entire chain!
- Questions:
  - Can we return a fallback response if a service is down?
  - Can we implement a Circuit Breaker pattern to reduce load?
  - Can we retry requests in case of temporary failures?
  - Can we implement rate limiting?
- Solution: Circuit Breaker Framework - Resilience4j

# Distributed Tracing

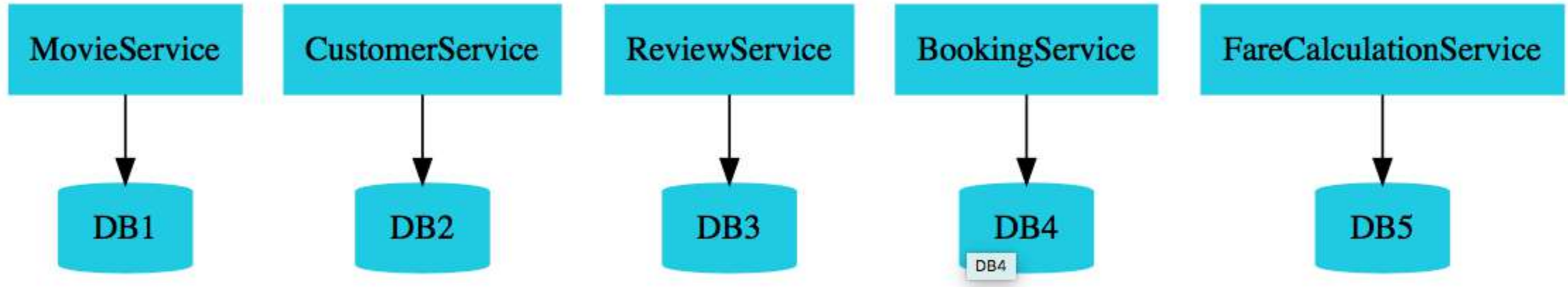


- Complex call chain
- How do you debug problems?
- How do you trace requests across microservices?
- Enter Distributed Tracing





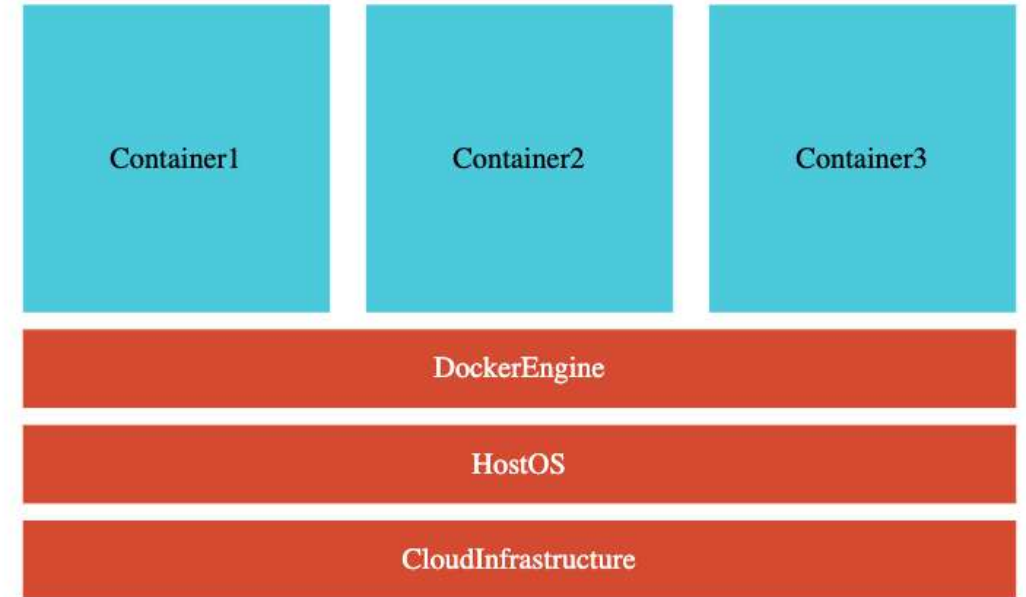
## Distributed Tracing



- Enterprises are heading towards microservices architectures
  - Build small focused microservices
  - **Flexibility to innovate** and build applications in different programming languages (Go, Java, Python, JavaScript, etc)
  - **BUT deployments become complex!**
  - How can we have **one way of deploying** Go, Java, Python or JavaScript .. microservices?
    - Enter **containers!**

# Docker

- Create **Docker images** for each microservice
- Docker image **contains everything a microservice needs** to run:
  - Application Runtime (JDK or Python or NodeJS)
  - Application code
  - Dependencies
- You can run these docker containers **the same way** on any infrastructure
  - Your local machine
  - Corporate data center
  - Cloud



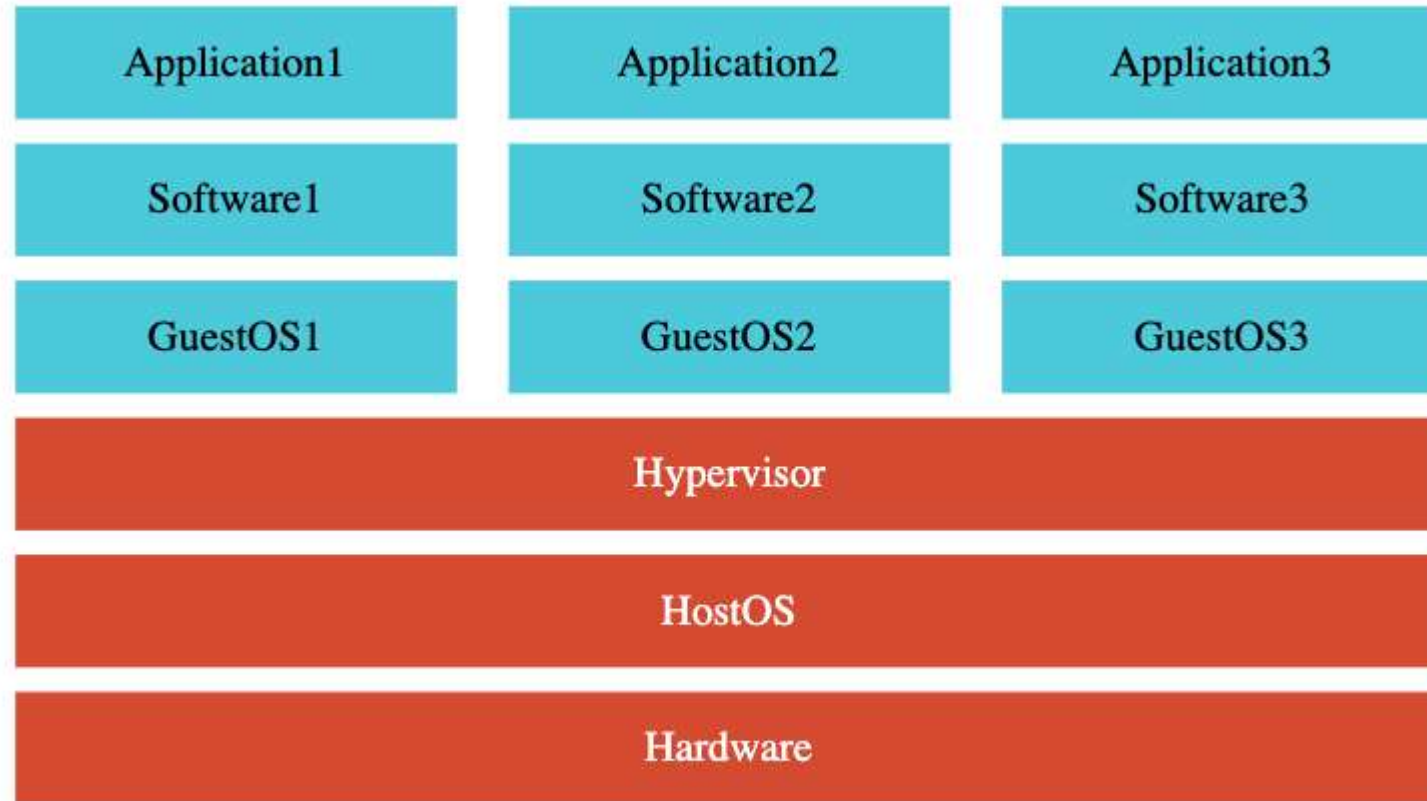
Applications

Software

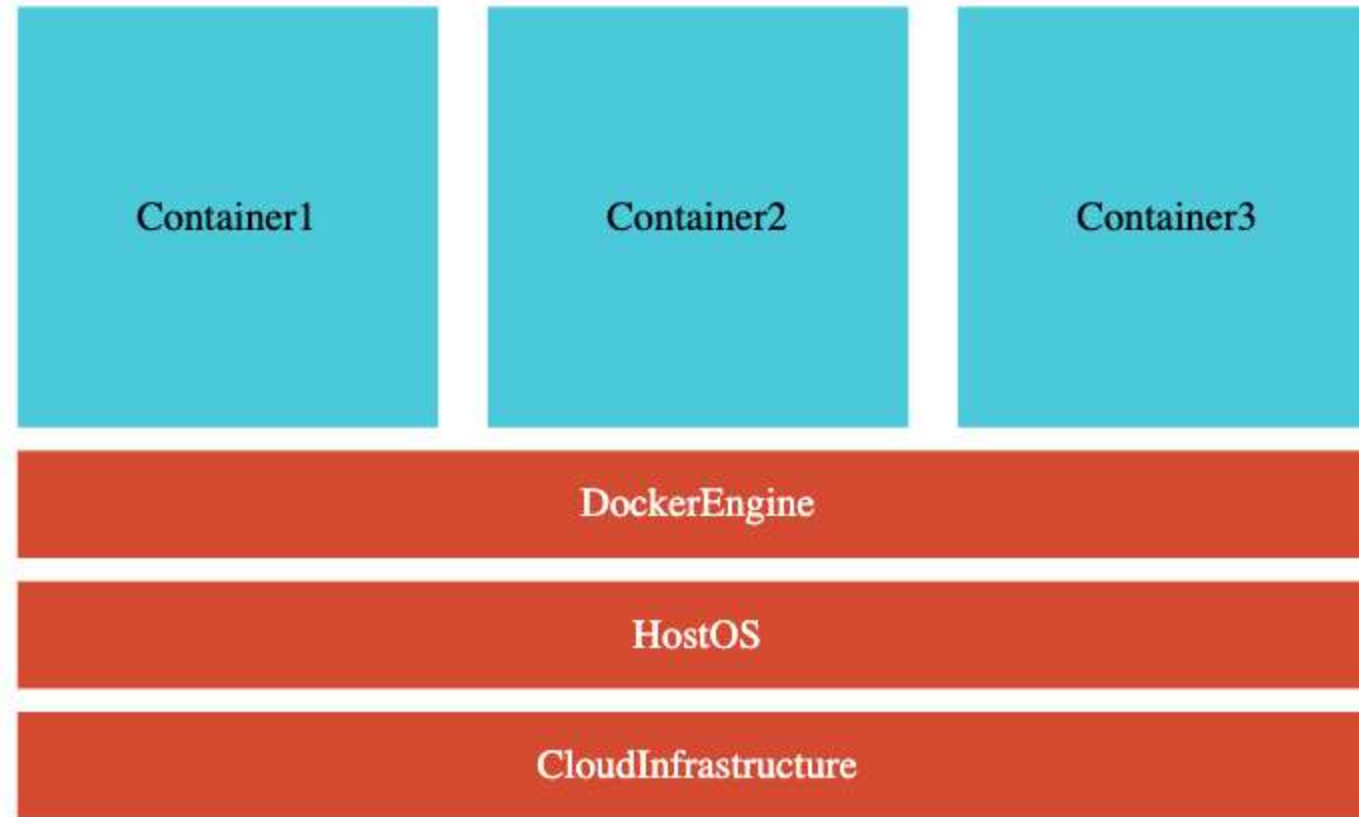
OS

Hardware

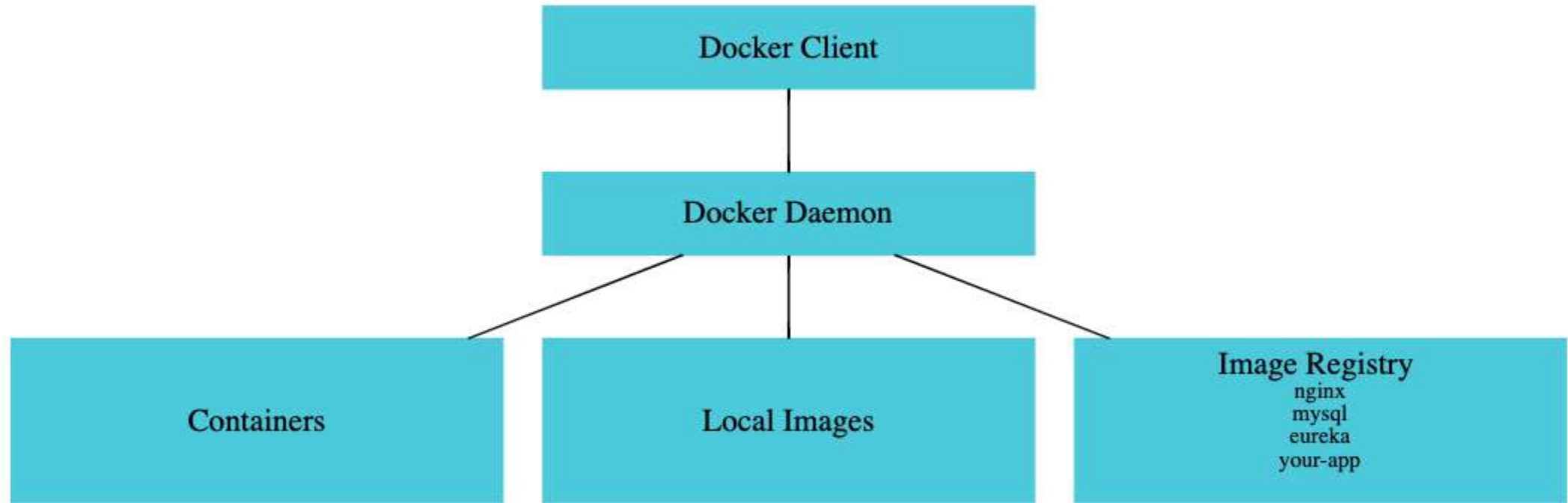
*Traditional Deployment*



*Deployments using Virtual Machines*



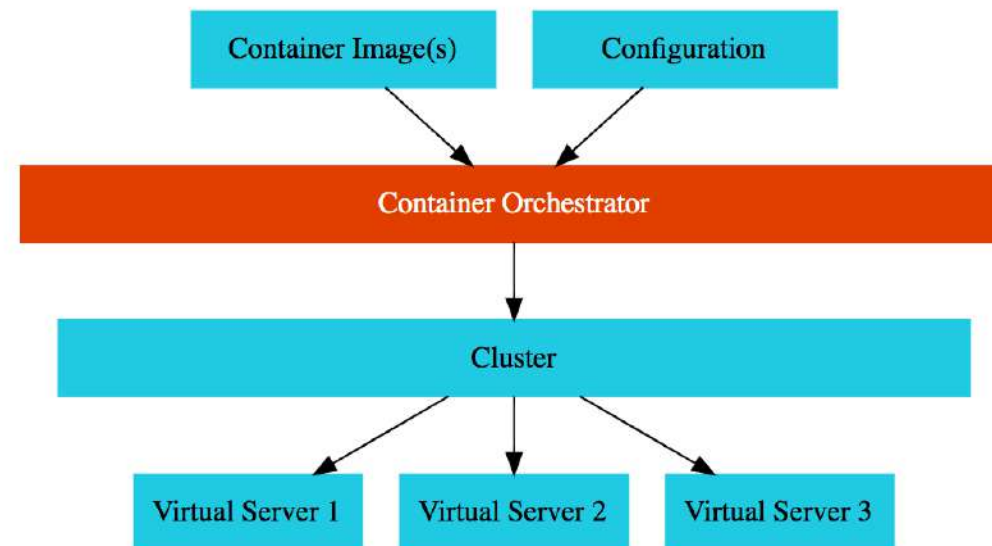
*Deployments using Docker*



## *Docker Architecture*

# Container Orchestration

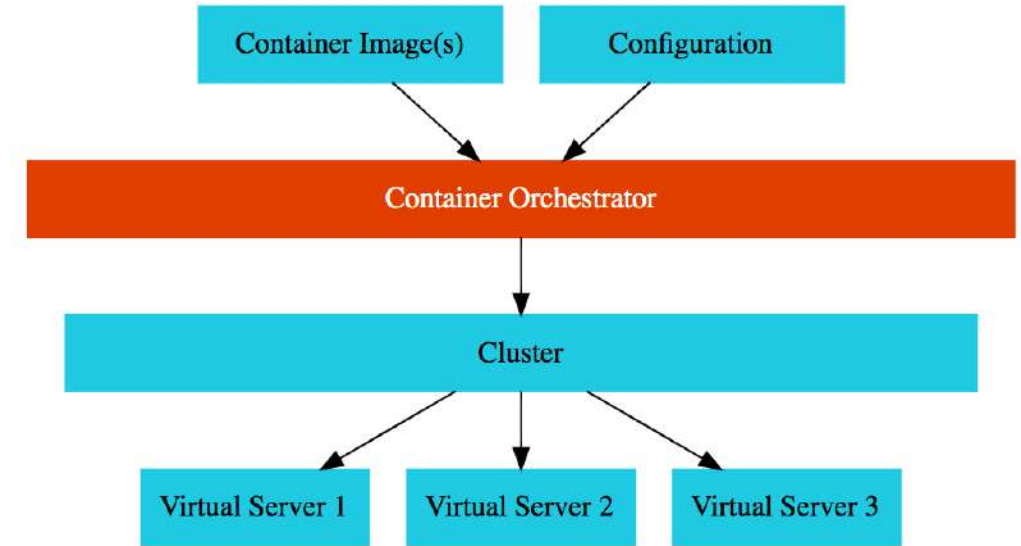
- **Requirement** : I want 10 instances of Microservice A container, 15 instances of Microservice B container and ....
- **Typical Features**:
  - **Auto Scaling** - Scale containers based on demand
  - **Service Discovery** - Help microservices find one another
  - **Load Balancer** - Distribute load among multiple instances of a microservice
  - **Self Healing** - Do health checks and replace failing instances
  - **Zero Downtime Deployments** - Release new versions without downtime

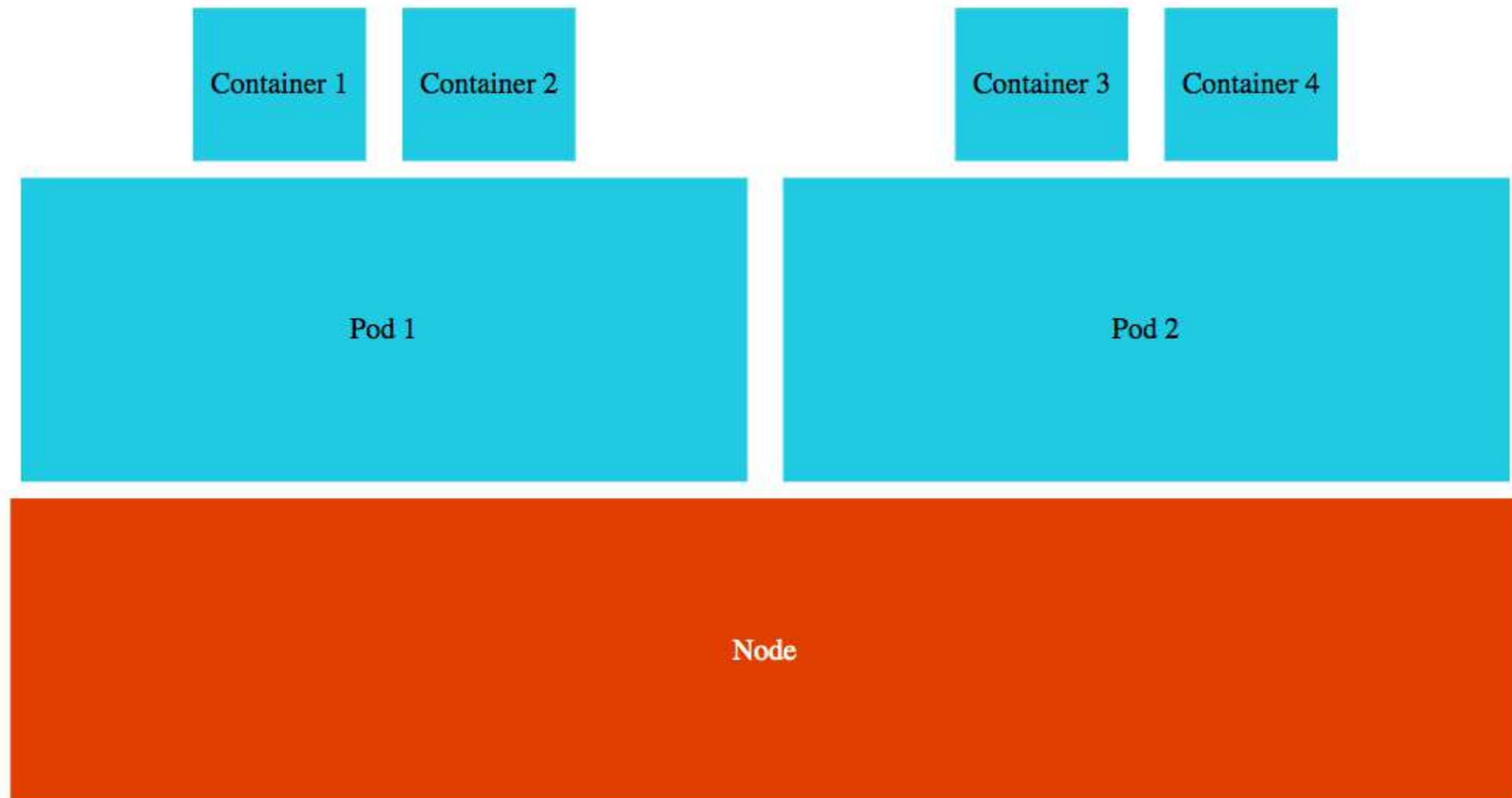




# Container Orchestration Options

- **AWS Specific**
  - AWS Elastic Container Service (ECS)
  - AWS Fargate : Serverless version of AWS ECS
- **Cloud Neutral - Kubernetes**
  - AWS - Elastic Kubernetes Service (EKS)
  - Azure - Azure Kubernetes Service (AKS)
  - GCP - Google Kubernetes Engine (GKE)
  - EKS/AKS does not have a free tier!
    - We use GCP and GKE!





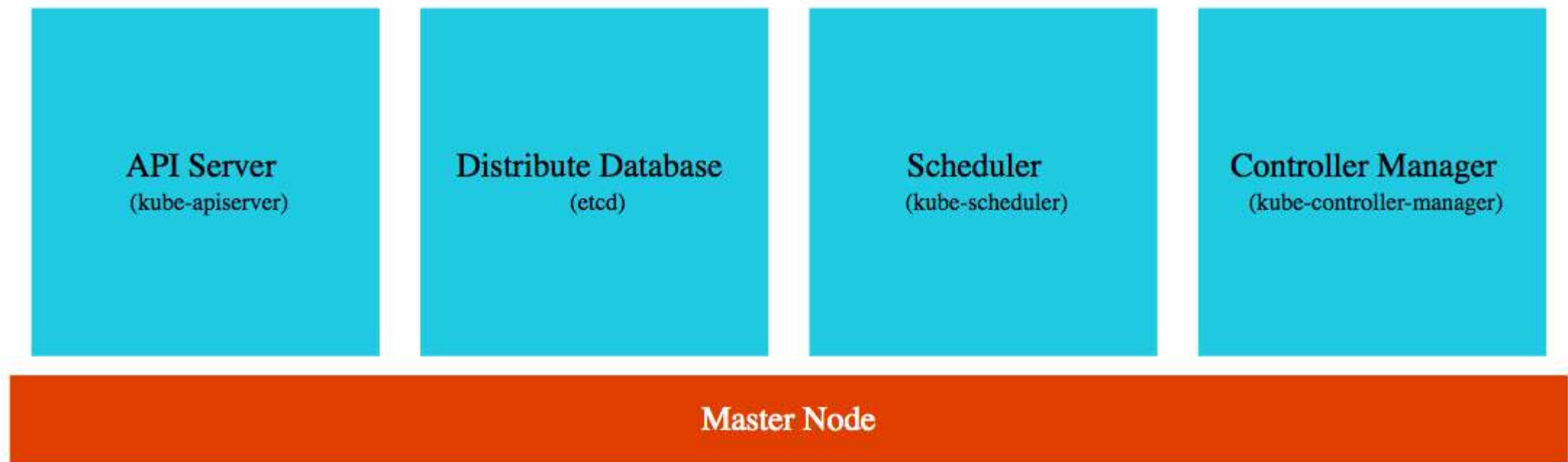
# *Kubernetes Architecture*

**Master Node(s)**  
Manages Cluster

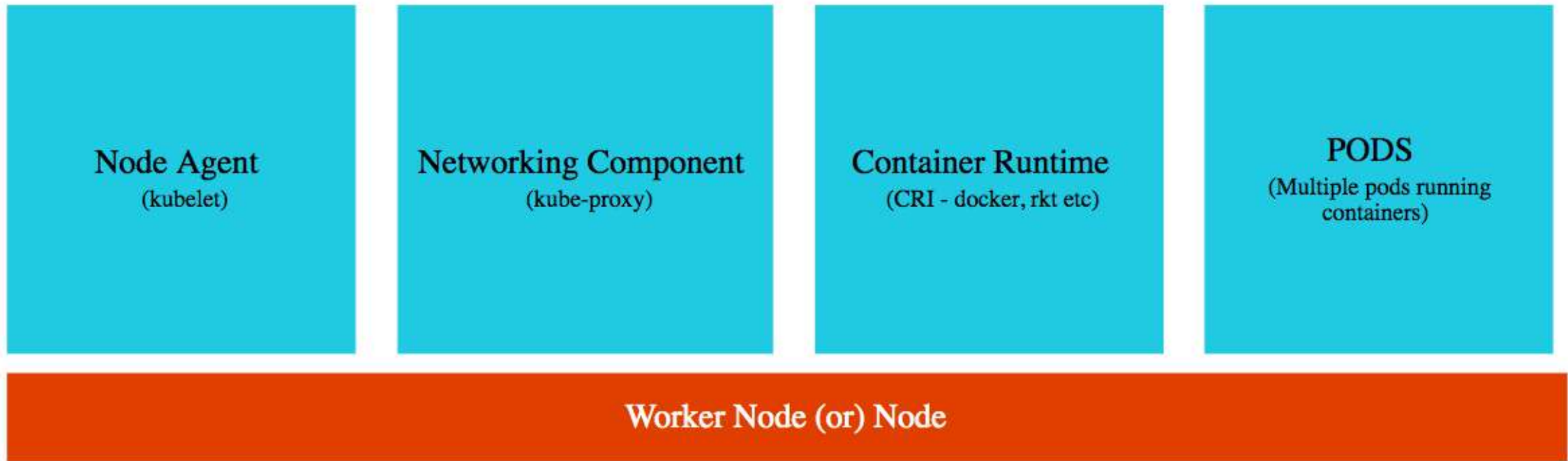
**Worker Node(s)**  
Run Your Applications

**Cluster**

*Kubernetes Architecture*



*Kubernetes Architecture*



*Kubernetes Architecture*



*Kubernetes Deployments*



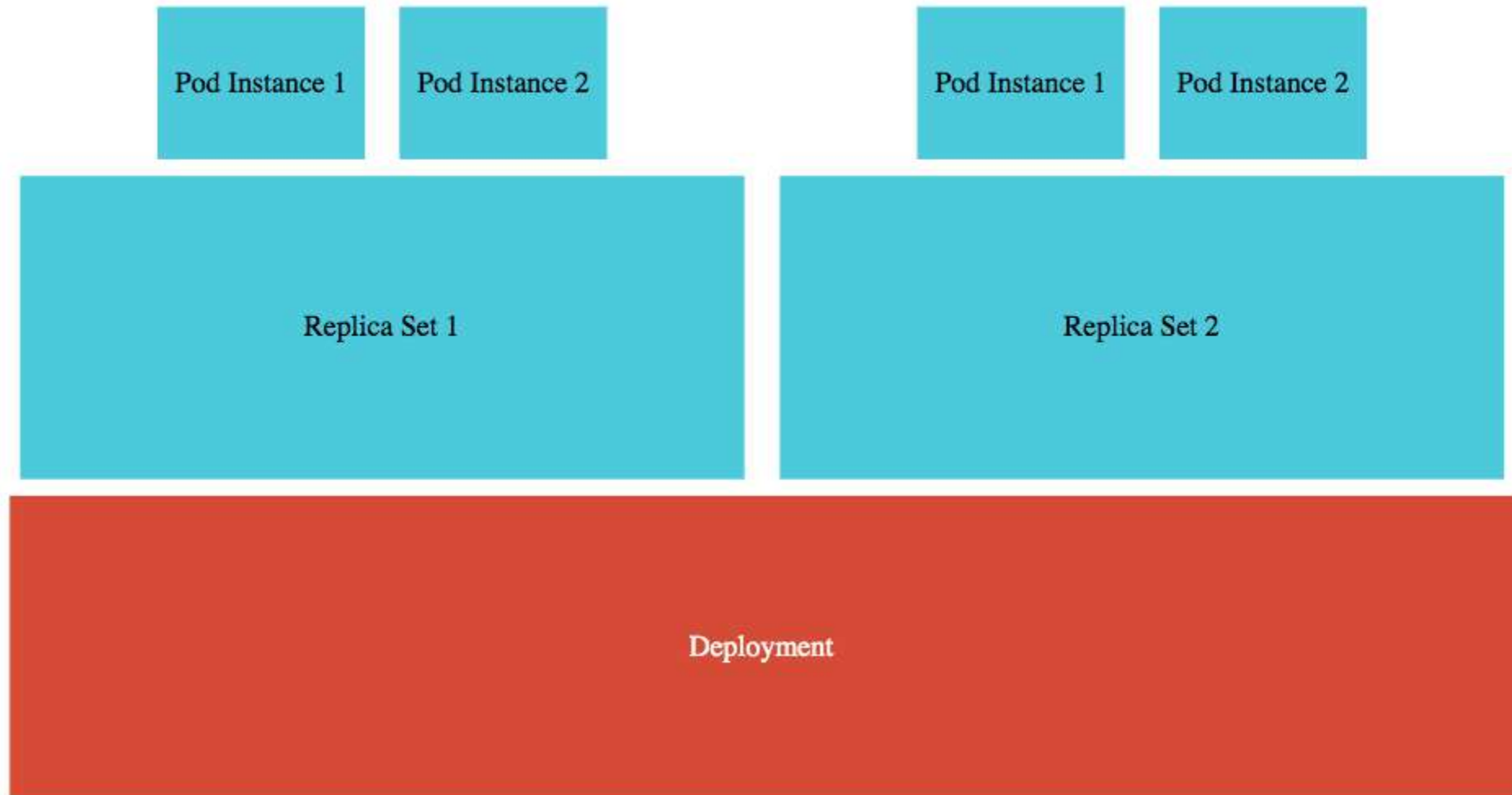
Create Cluster

Create Deployment

Docker Repository

*Kubernetes Deployments*





## *Kubernetes Deployments*



*Kubernetes Service*

# Kubernetes - Liveness and Readiness Probes



- Kubernetes uses probes to check the health of a microservice:
  - If readiness probe is not successful, no traffic is sent
  - If liveness probe is not successful, pod is restarted
- Spring Boot Actuator ( $\geq 2.3$ ) provides inbuilt readiness and liveness probes:
  - `/health/readiness`
  - `/health/liveness`

# What Next?

# 7 Roadmaps, 1 Million+ Learners

In **28**  
Minutes

- *AWS*
- *Azure*
- *Google Cloud*
- *Spring Boot*
- *Java Microservices*
- *Java Full Stack*
- *DevOps*



