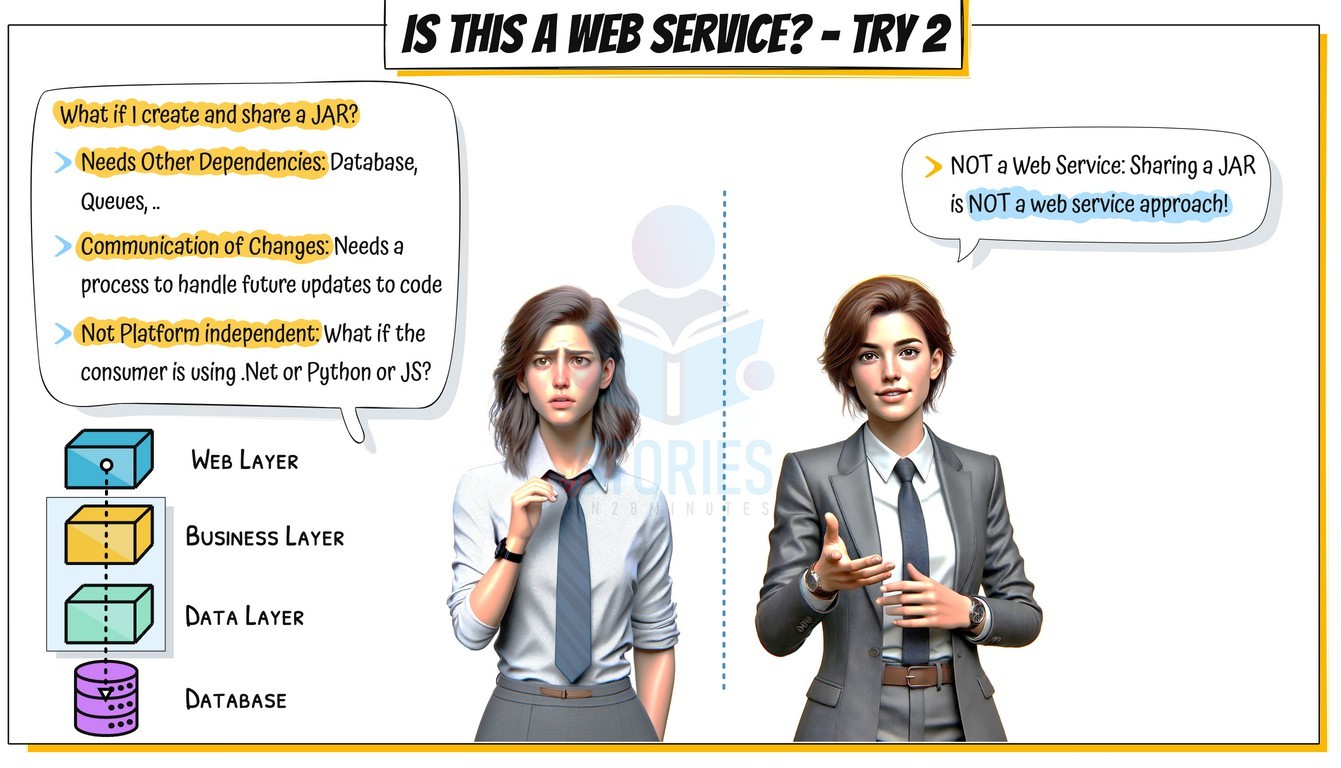




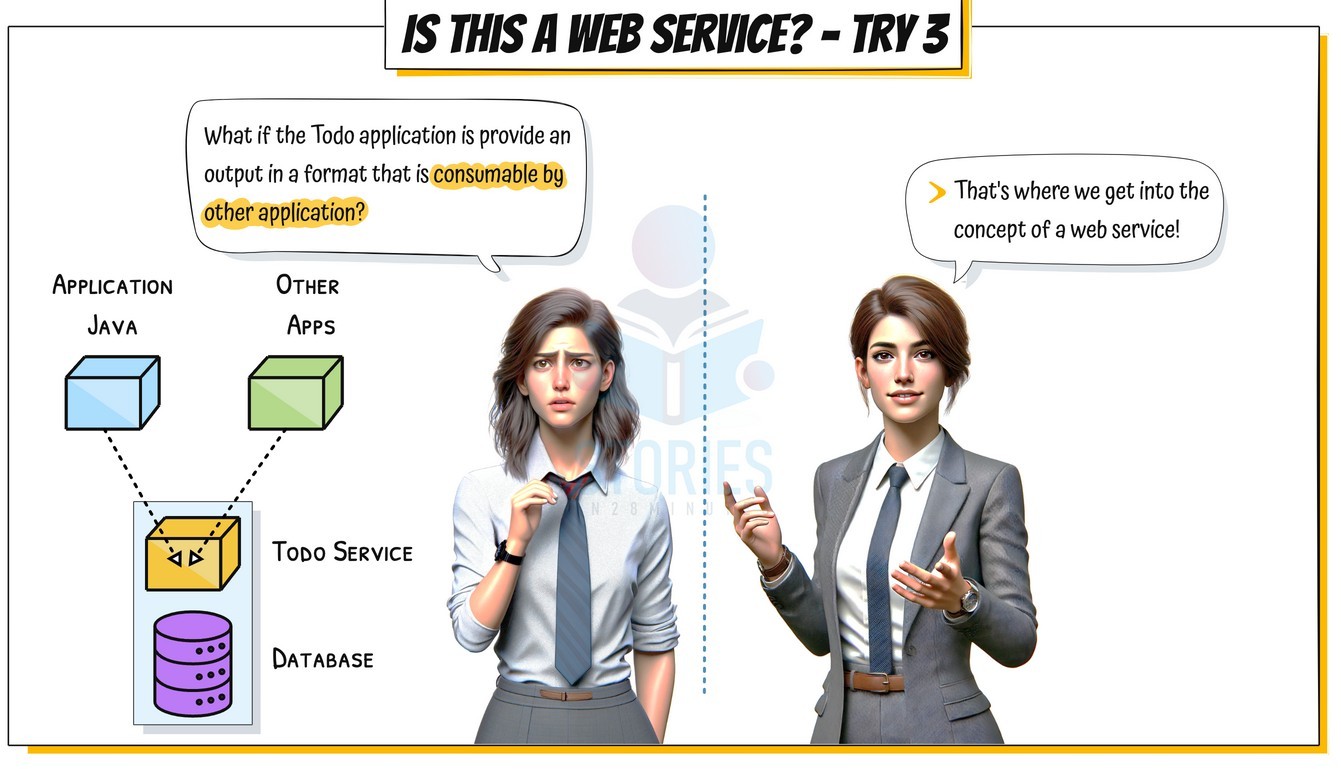
# Web Services



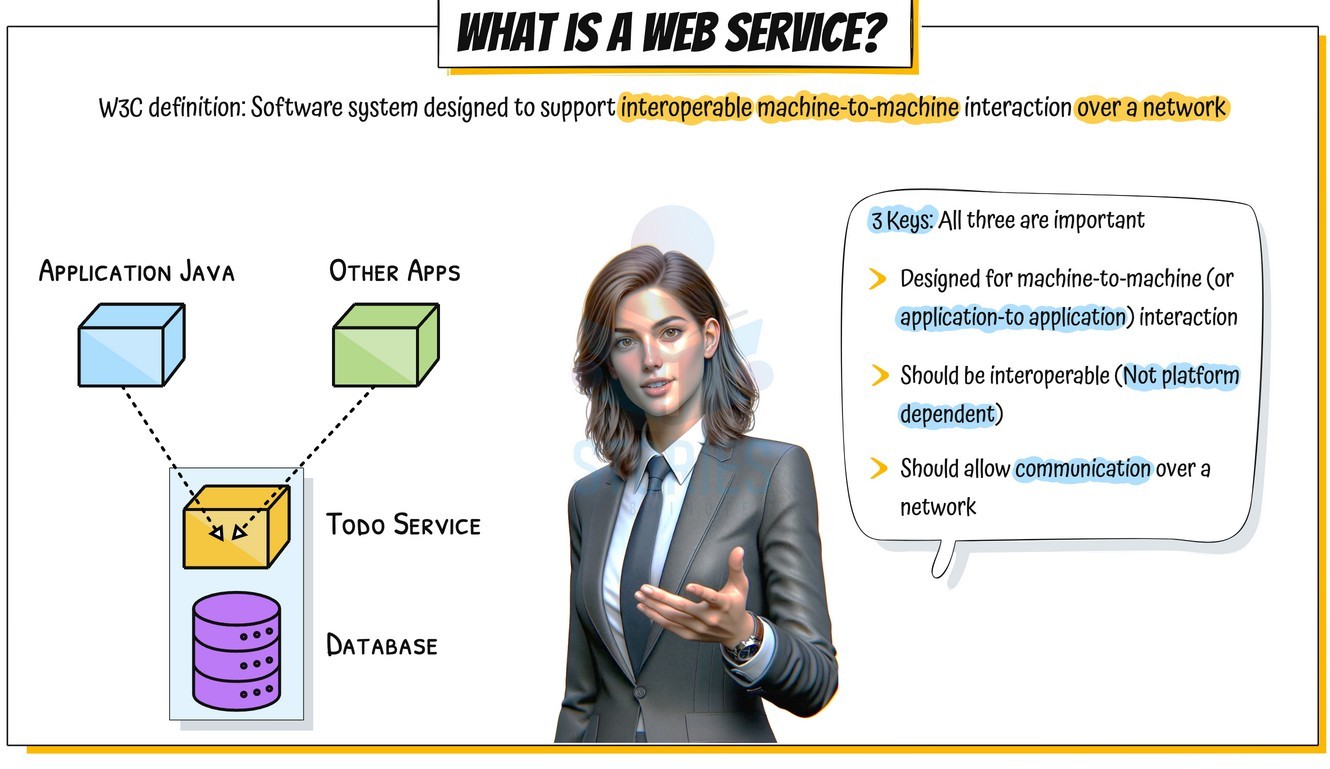




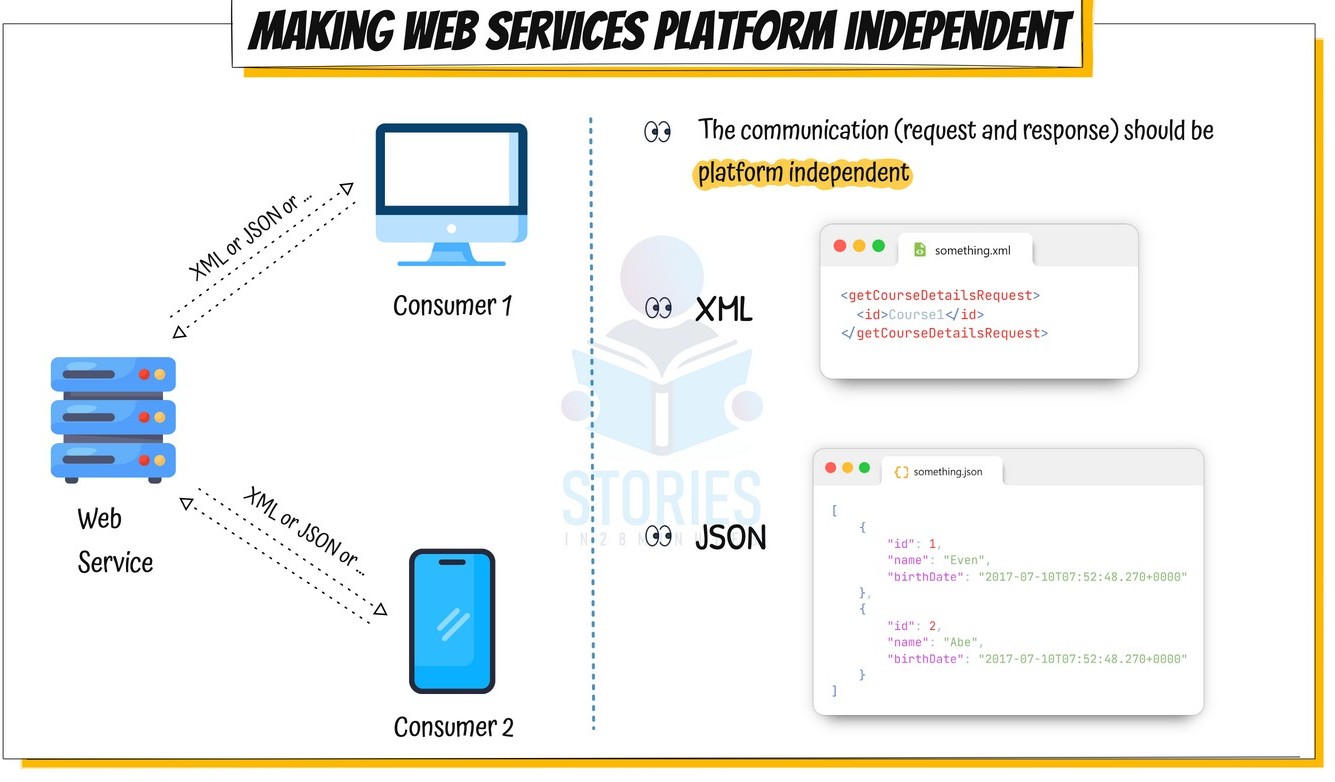








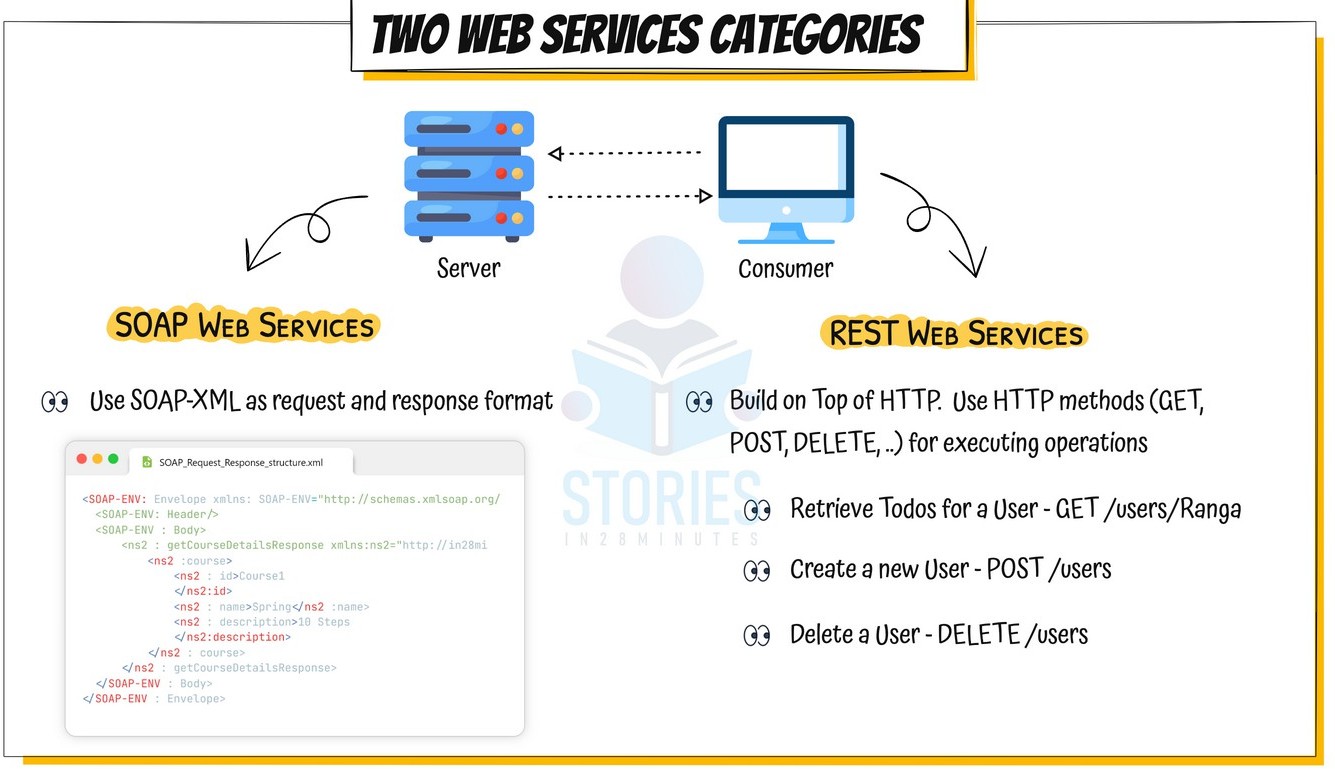














**Spring Boot in 10(ish) Steps**

### Getting Started with Spring Boot



 **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



 **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!



 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



<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?



**** 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



 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?

 **Starters**: 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

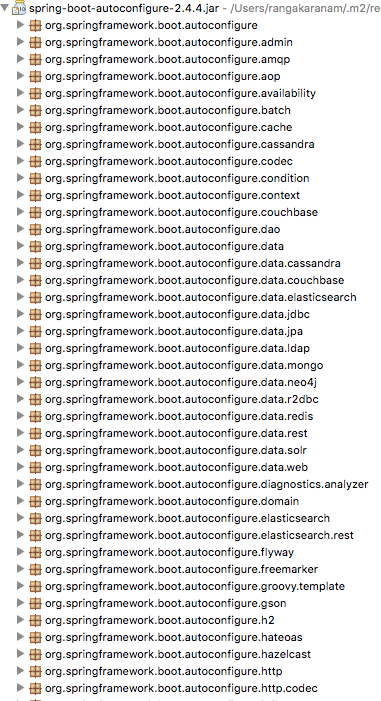
 **Talk to database using JPA** - Spring Boot Starter Data JPA

 **Talk to database using JDBC** - Spring Boot Starter JDBC

 **Secure your web application or REST API** - Spring Boot Starter Security

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)

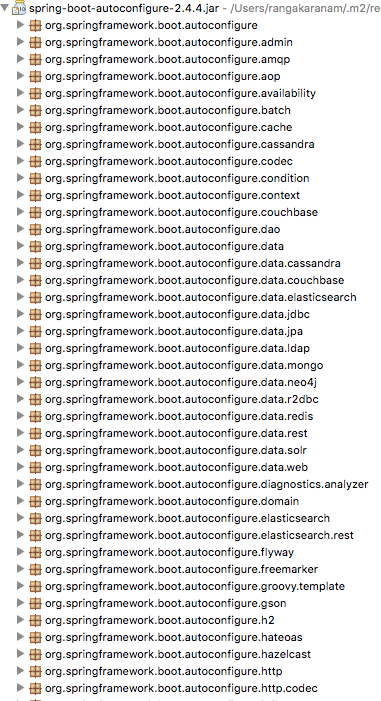
 Default Error Pages (ErrorMvcAutoConfiguration)

 Bean<->JSON

(JacksonHttpMessageConvertersConfiguration)

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)

## Build Faster with Spring Boot DevTools



 Increase developer productivity

 Why do you need to restart the server

**manually** for every code change?

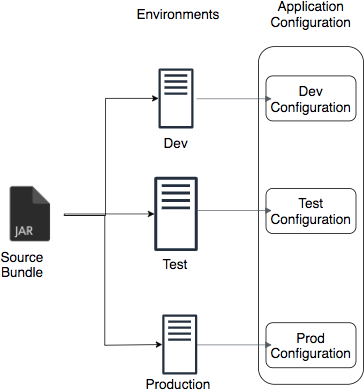
 **Remember**: For pom.xml dependency changes, you will need to restart server **manually**



# 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?

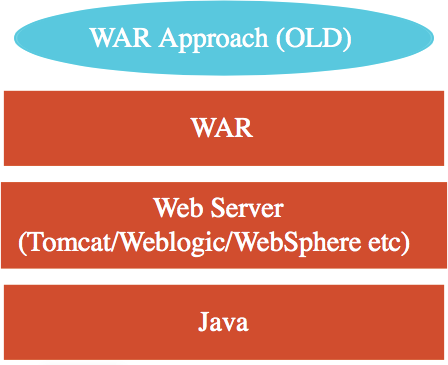
 **Profiles**: Environment specific configuration

 How can you define externalized configuration for your application?

 **ConfigurationProperites**: Define externalized configuration

Simplify Deployment with Spring Boot Embedded Servers



 How do you deploy your application?

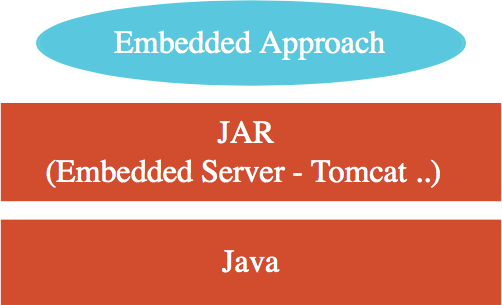
 Step 1 : Install Java

 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



 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



 **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



 **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!

Building REST API with Spring Boot - Approach



 **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

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

## 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

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}**

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

Advanced REST API Features



 Documentation

 Content Negotiation

 Internationalization - i18n  Versioning

 HATEOAS

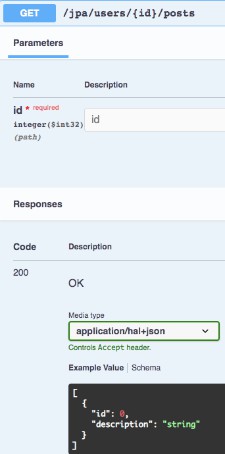
 Static Filtering

 Dynamic Filtering  Monitoring

 ....

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**:

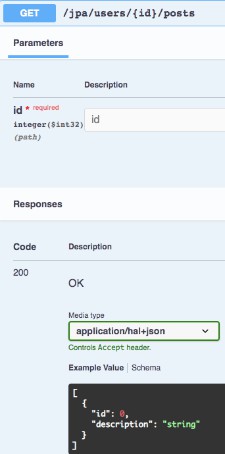
 **1**: Manually Maintain Documentation

 Additional effort to keep it in sync with code

 **2**: Generate from code

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

##### 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?

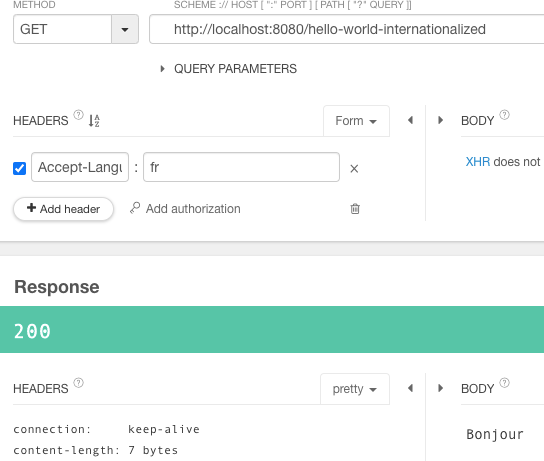
 **Content Negotiation**

 Example: Accept header (MIME types - application/xml, application/json, ..)

 Example: Accept-Language header (en, nl, fr, ..)

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

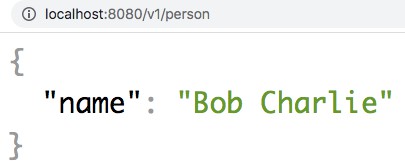
 Accept-Language - indicates natural language and locale that the consumer prefers

 Example: en - English (Good Morning)  Example: nl - Dutch (Goedemorgen)

 Example: fr - French (Bonjour)

#### 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!**

Versioning REST API - Options



 **URI Versioning** - Twitter

 [***http://localhost:8080/v1/person***](http://localhost:8080/v1/person) ****** [***http://localhost:8080/v2/person***](http://localhost:8080/v2/person)

 **Request Parameter versioning** - Amazon

 [***http://localhost:8080/person?version=1***](http://localhost:8080/person?version=1) ****** [***http://localhost:8080/person?version=2***](http://localhost:8080/person?version=2)

 **(Custom) headers versioning** - Microsoft

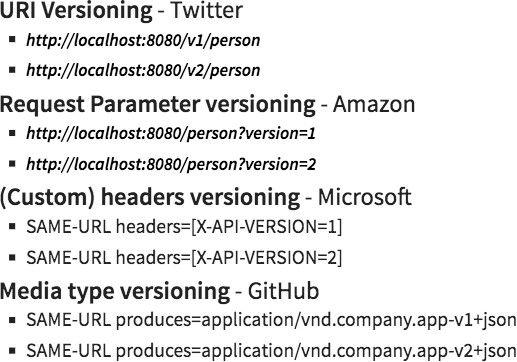
 SAME-URL headers=[X-API-VERSION=1]  SAME-URL headers=[X-API-VERSION=2]

 **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

##### Versioning REST API - Factors



**** **Factors to consider**

 URI Pollution

 Misuse of HTTP Headers  Caching

 Can we execute the request on the browser?  API Documentation

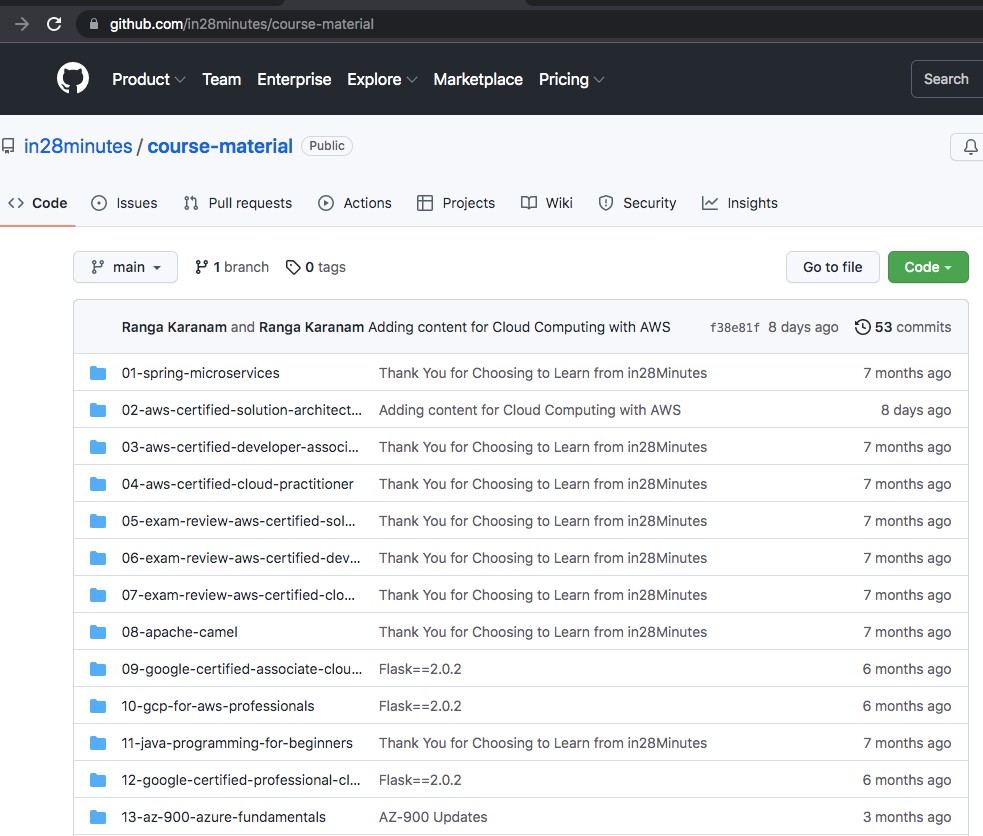
 **Summary**: No Perfect Solution

 **My Recommendations**

 Think about versioning even before you need it!  One Enterprise - One Versioning Approach

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

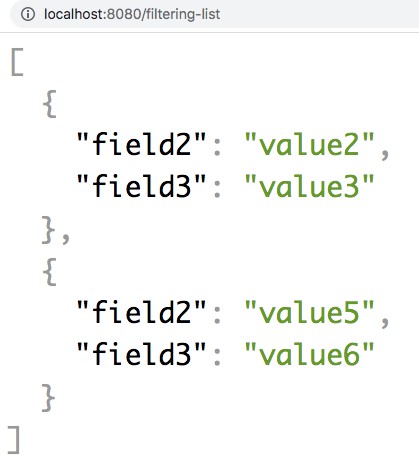
 **2:** Use Standard Implementation

 **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

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

Get Production-ready with Spring Boot Actuator



 **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



**** 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**

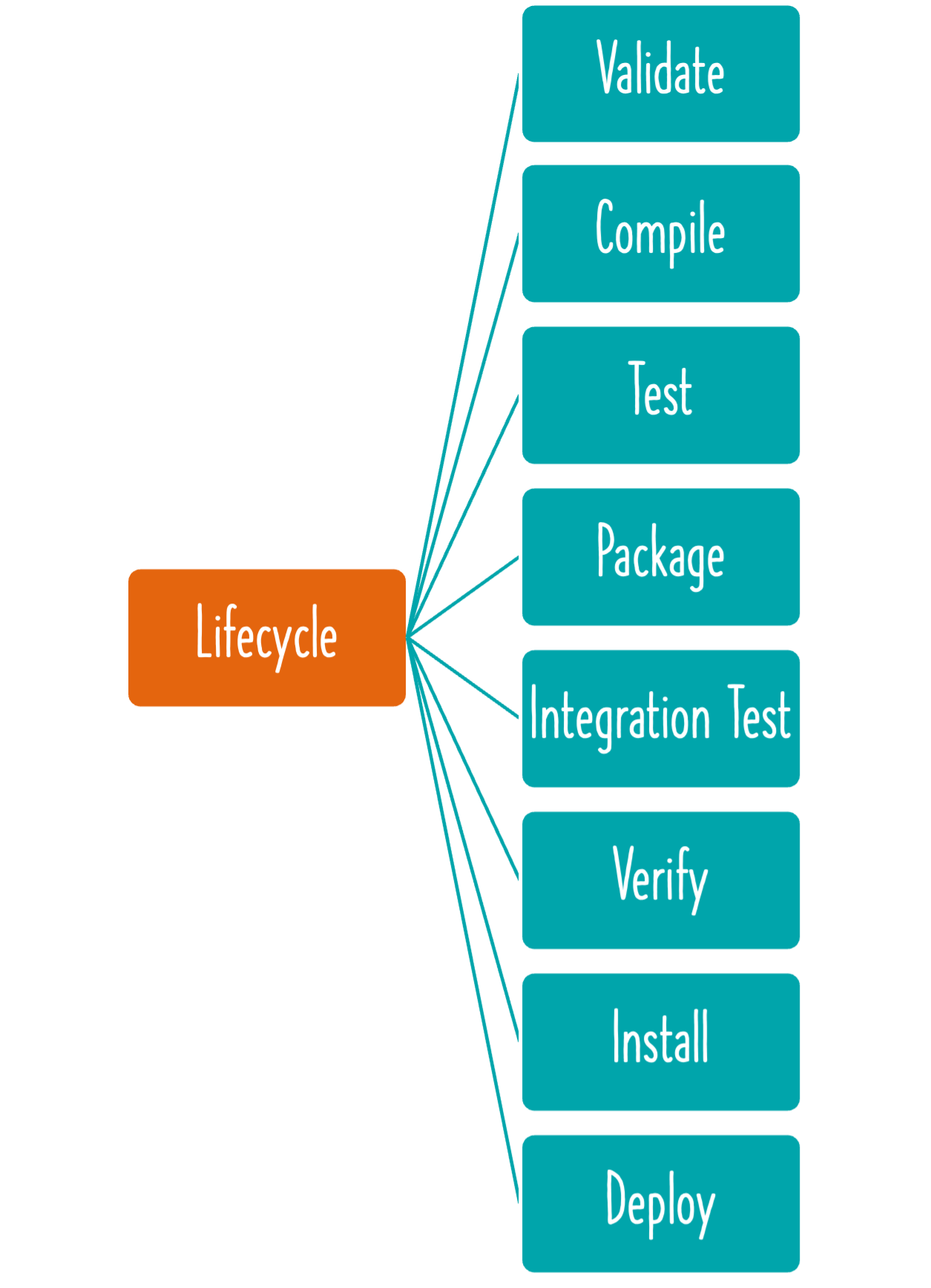
 Auto-configures HAL Explorer for Spring Boot Projects  spring-data-rest-hal-explorer



# Maven

##### What is Maven?



 **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?**

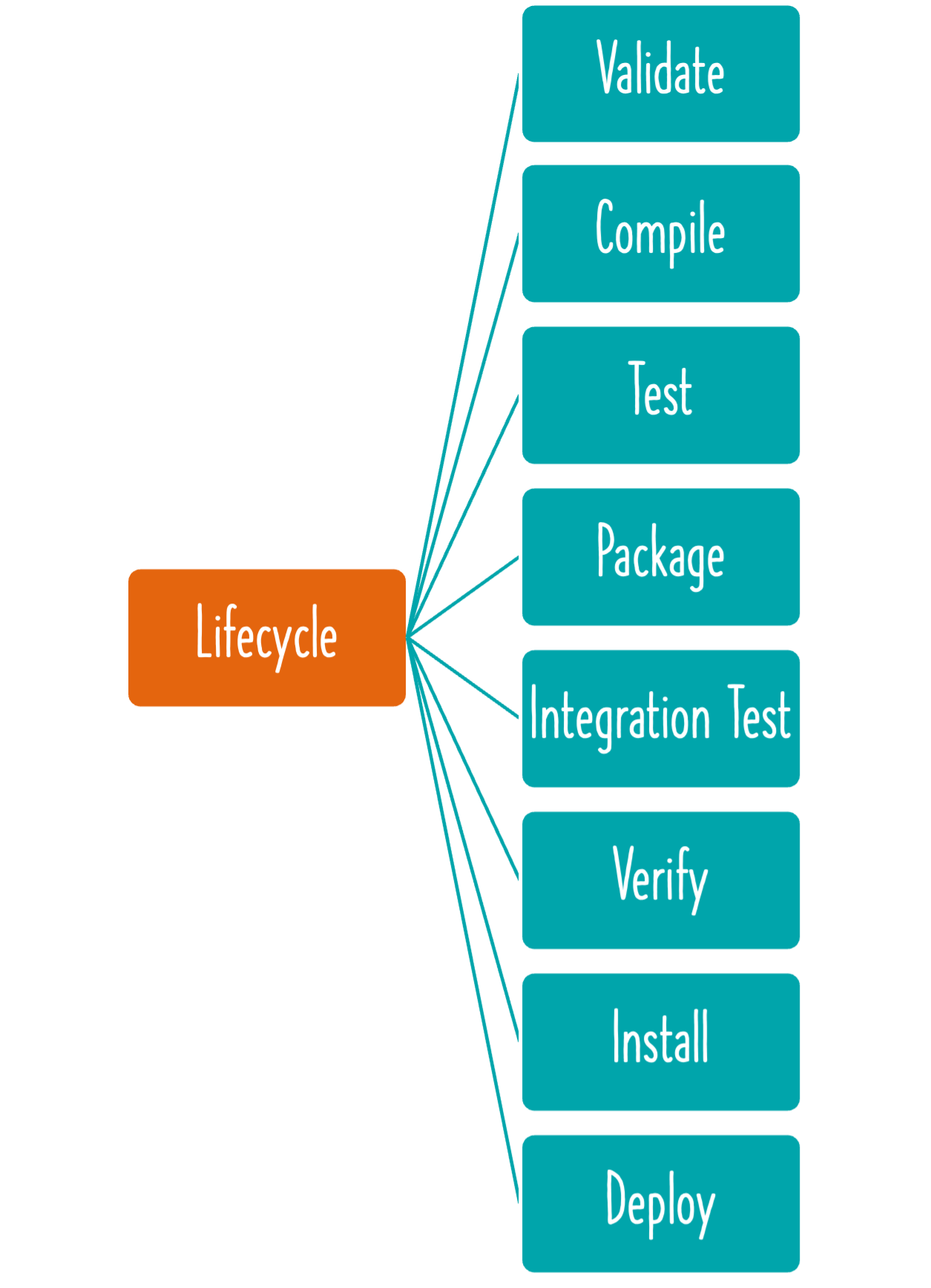
 Think about this: How can other projects use our new project?

 **Activity**: help:effective-pom, dependency:tree & Eclipse UI

 Let's add a new dependency: spring-boot-starter-web

Exploring Maven Build Life Cycle



 When we run a maven command, maven build life cycle is used

 Build LifeCycle is a sequence of steps

 Validate  Compile  Test

 Package

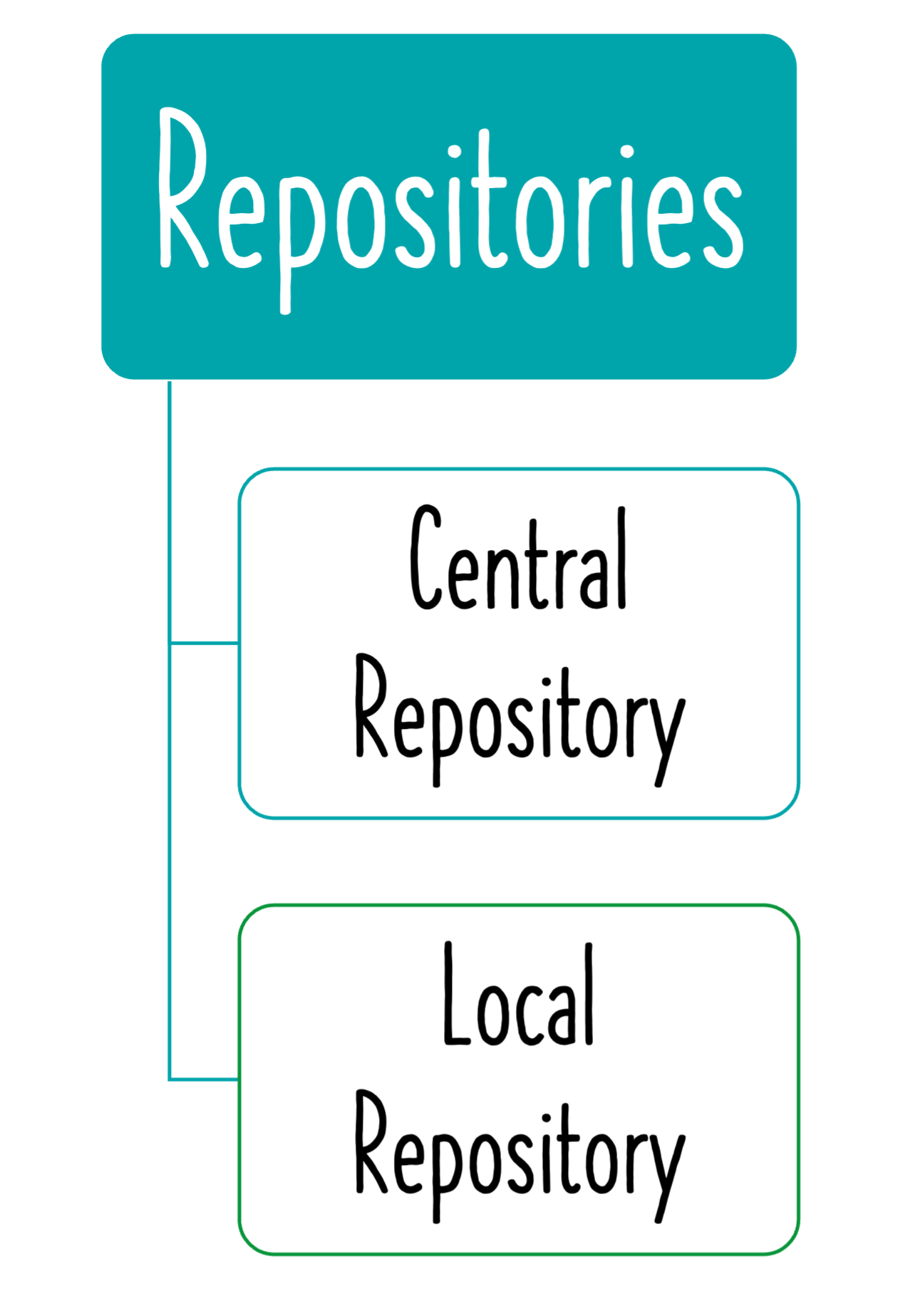
 Integration Test  Verify

 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

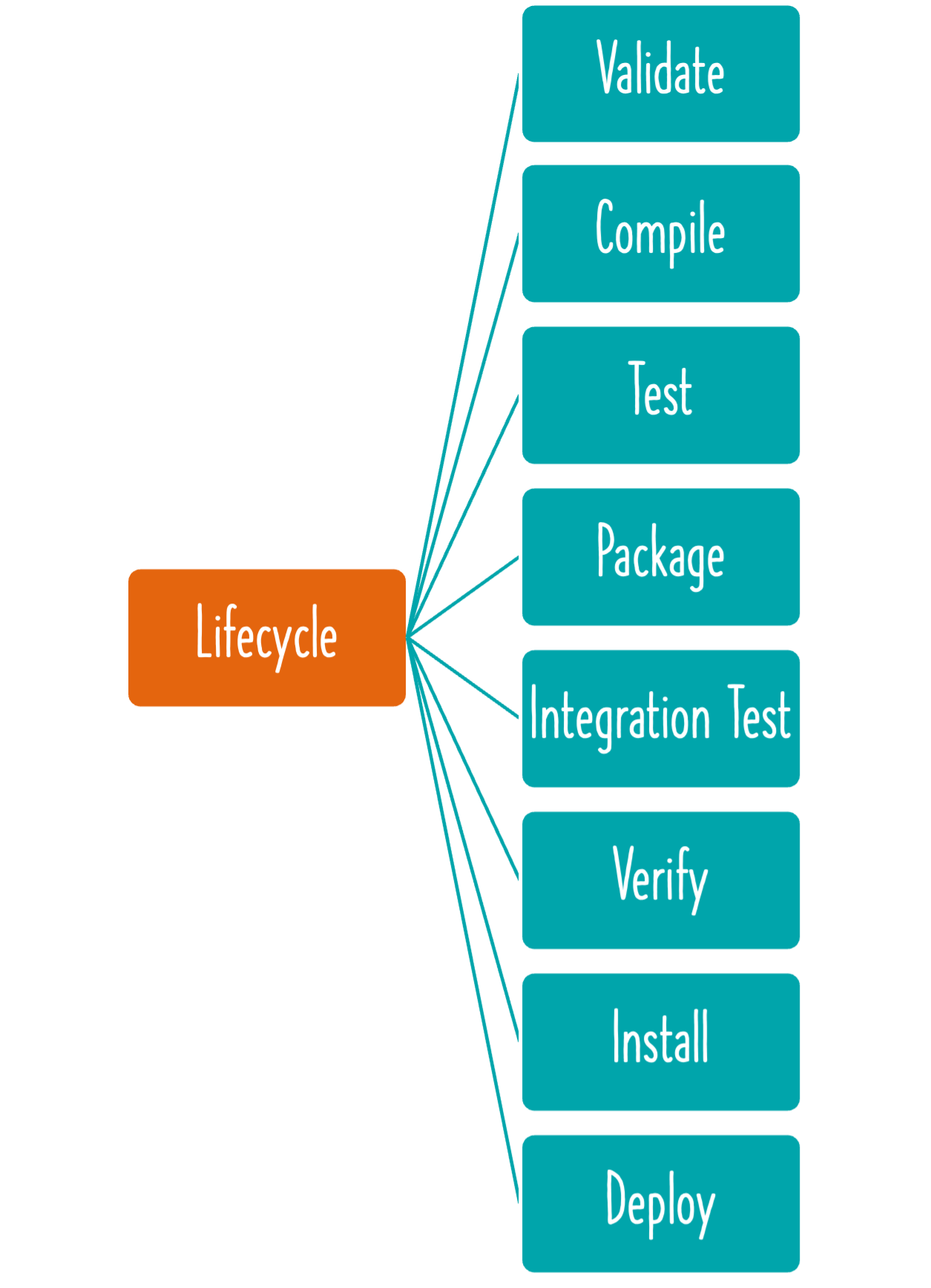
 When a dependency is added to pom.xml, Maven tries to download the dependency

 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

 OBSERVCE CAREFULLY: This will also compile source files

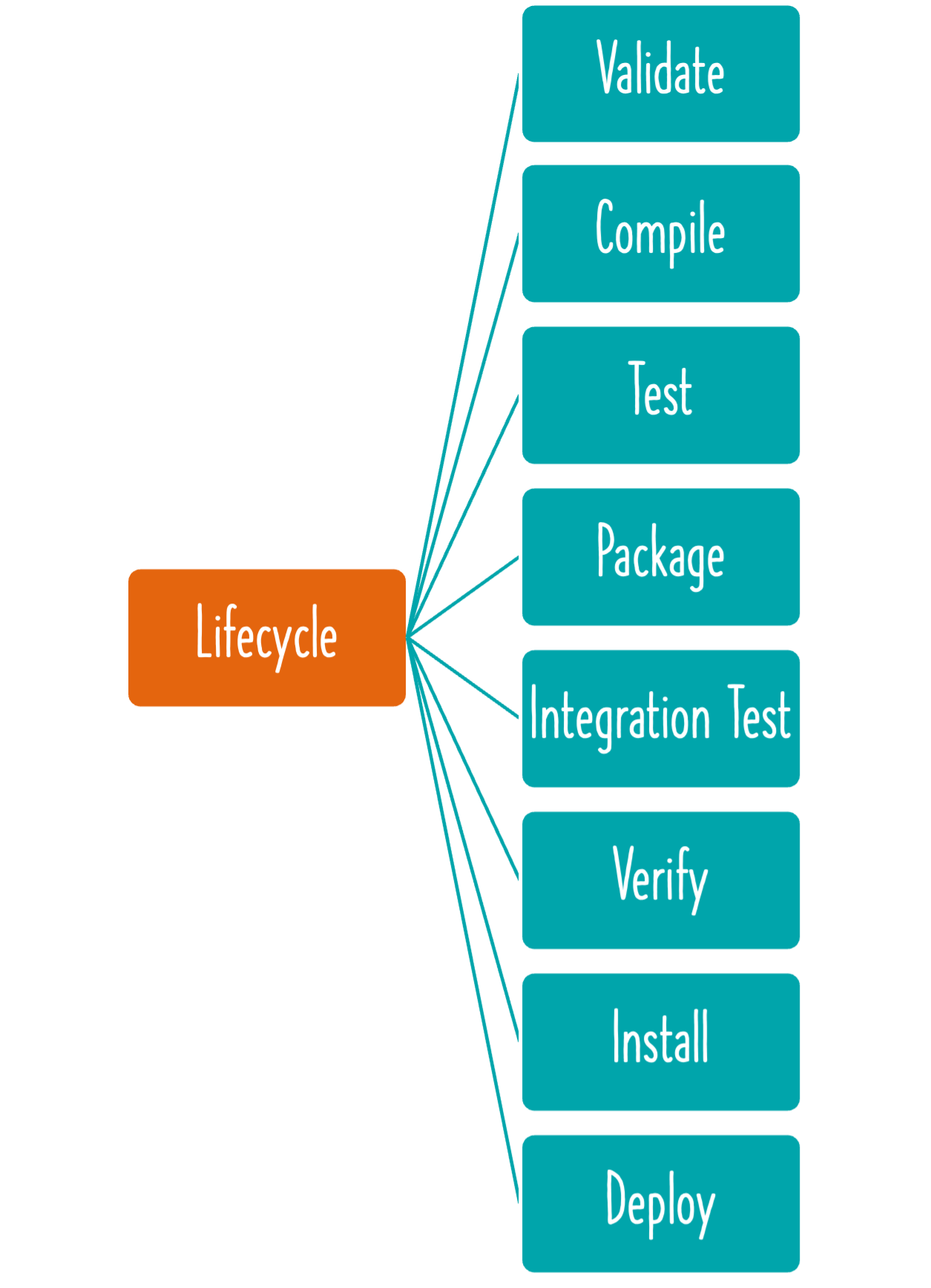
 mvn clean: Delete target directory  mvn test: Run unit tests

 mvn package: Create a jar  mvn help:effective-pom

 mvn dependency:tree

Spring Boot Maven Plugin



 **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 - Ex:integration tests)  mvn spring-boot:**stop** (Stop application)

 mvn spring-boot:**build-image** (Build a container image)

How are Spring Releases Versioned?



 **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

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

Gradle Plugins



 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?

 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

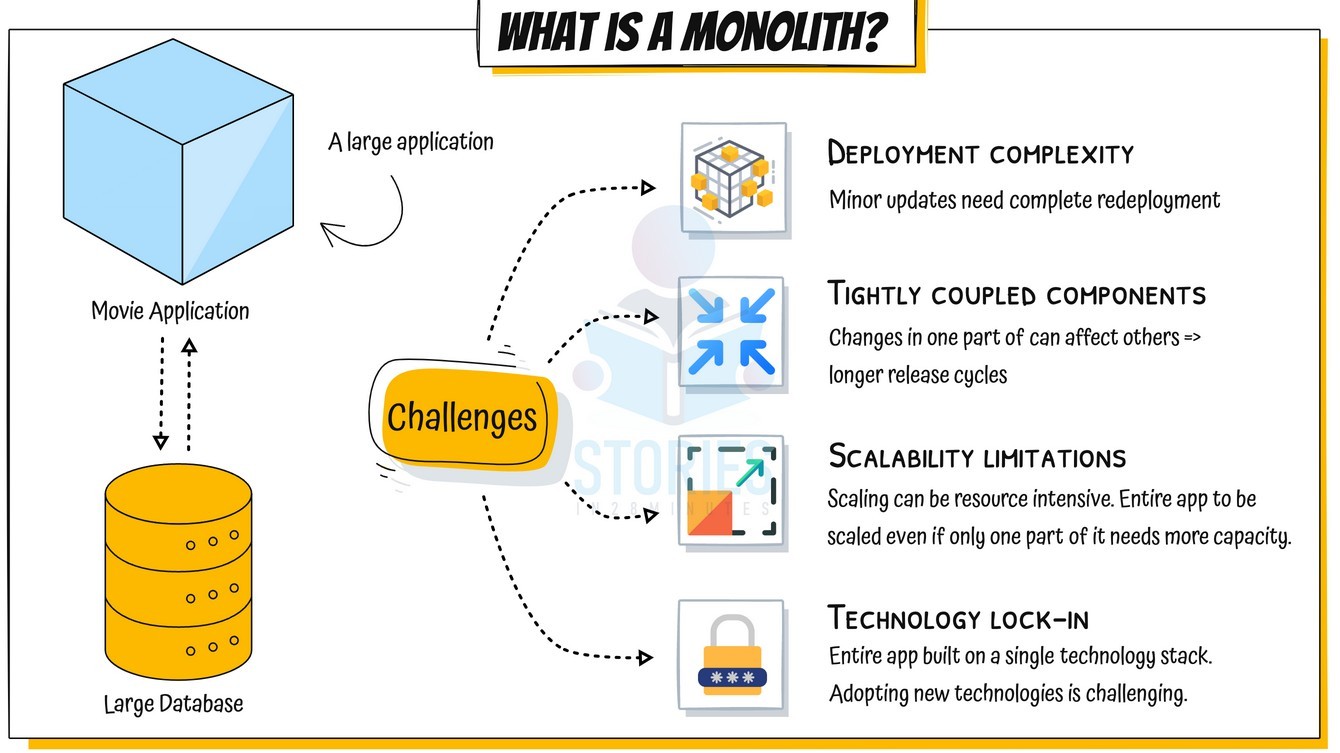
 **What Do I Recommend**: I'm sitting on the fence for now

 Choose whatever tool best meets your projects needs  If your builds are taking really long, go with Gradle

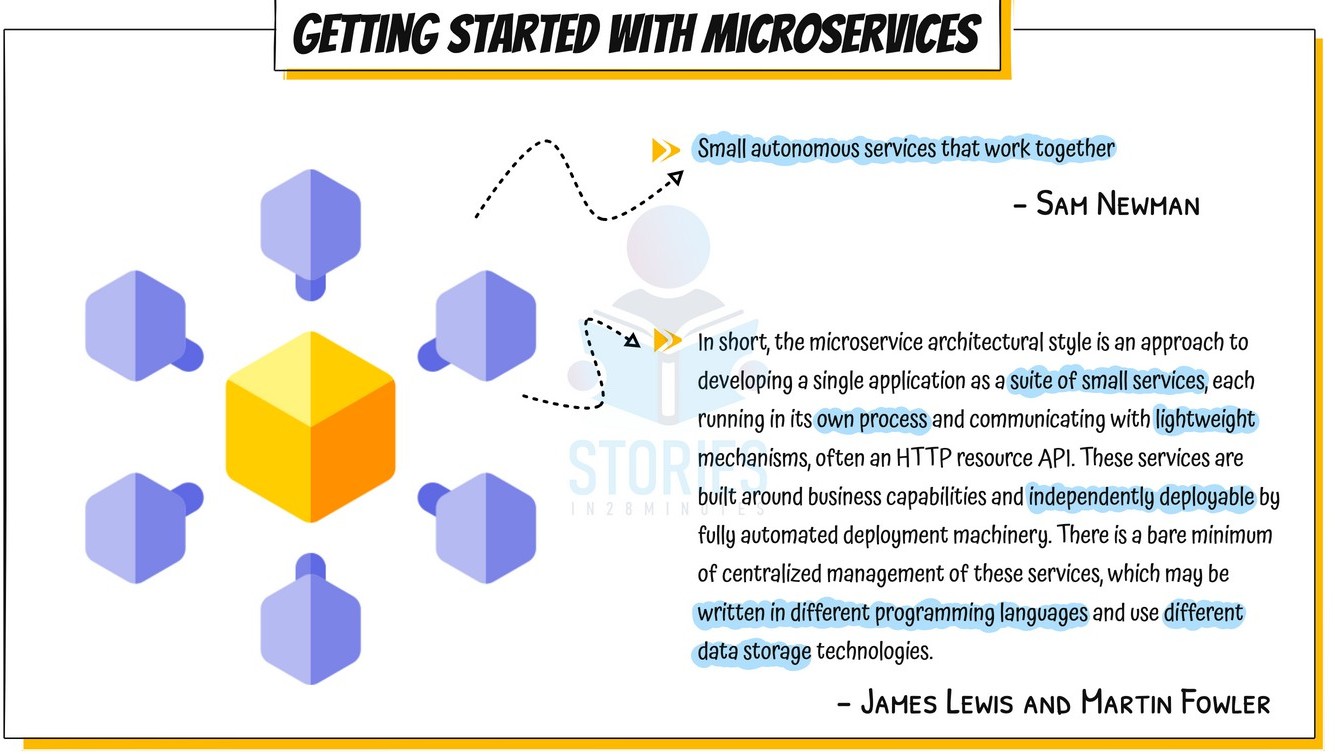
 If your builds are simple, stick with Maven



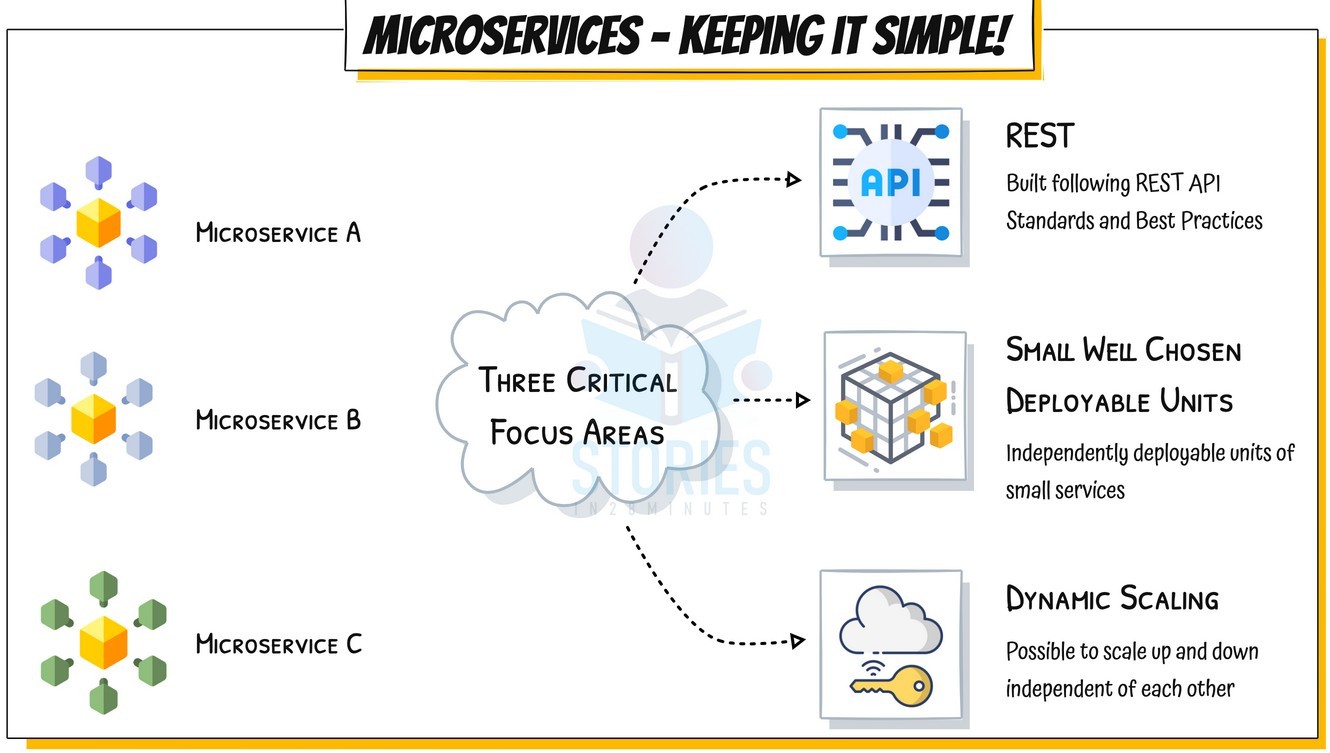
# Microservices



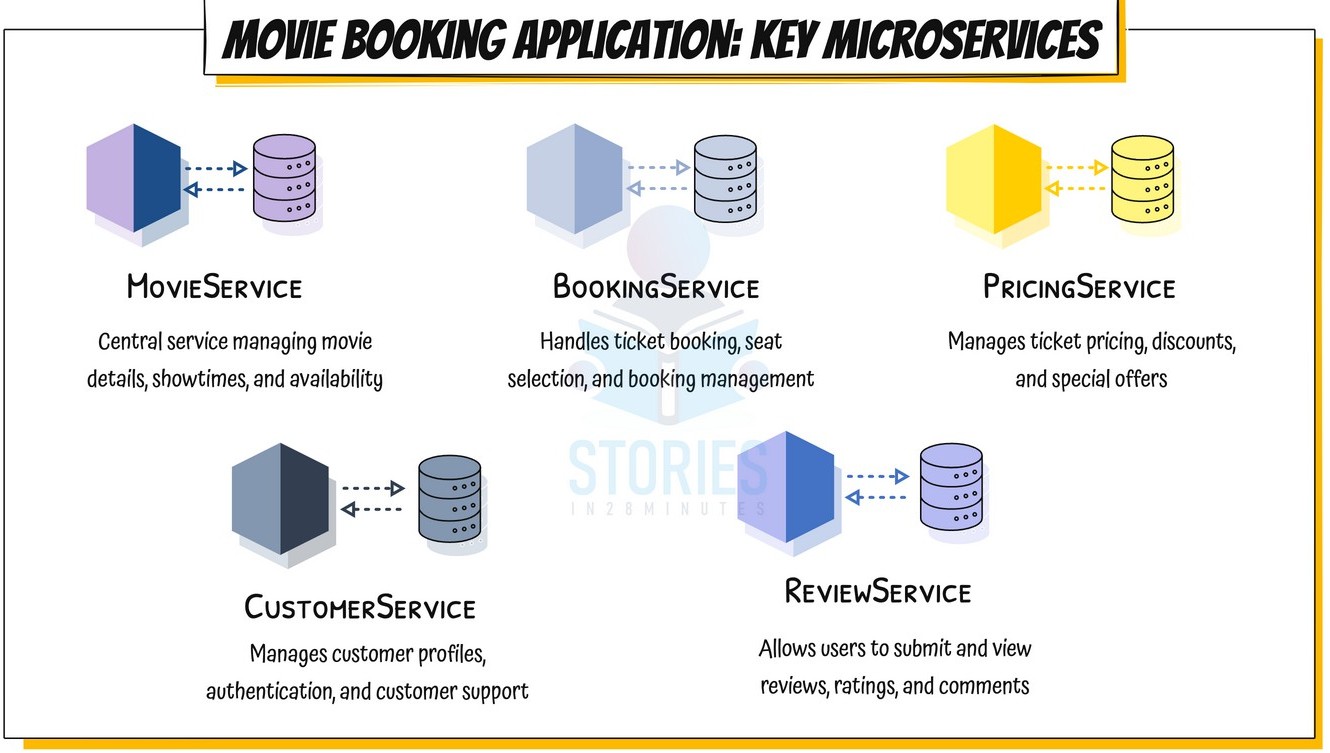




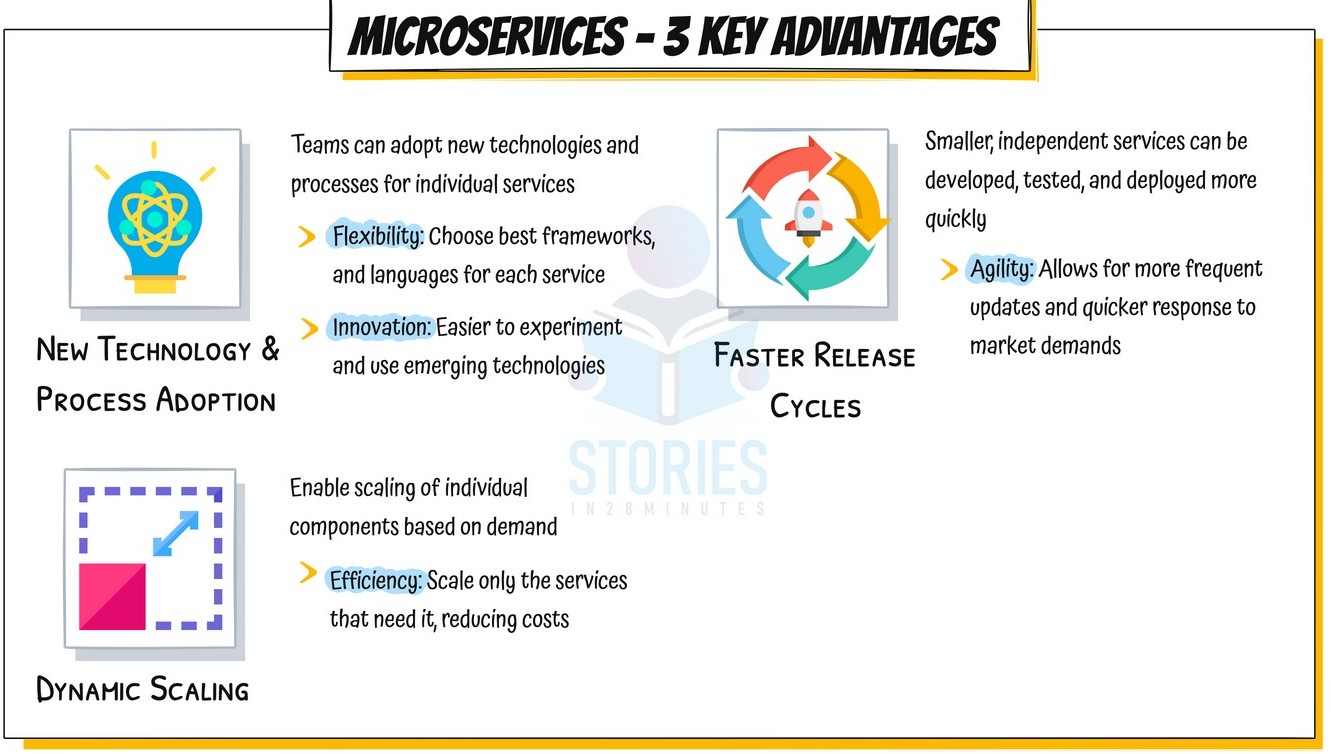




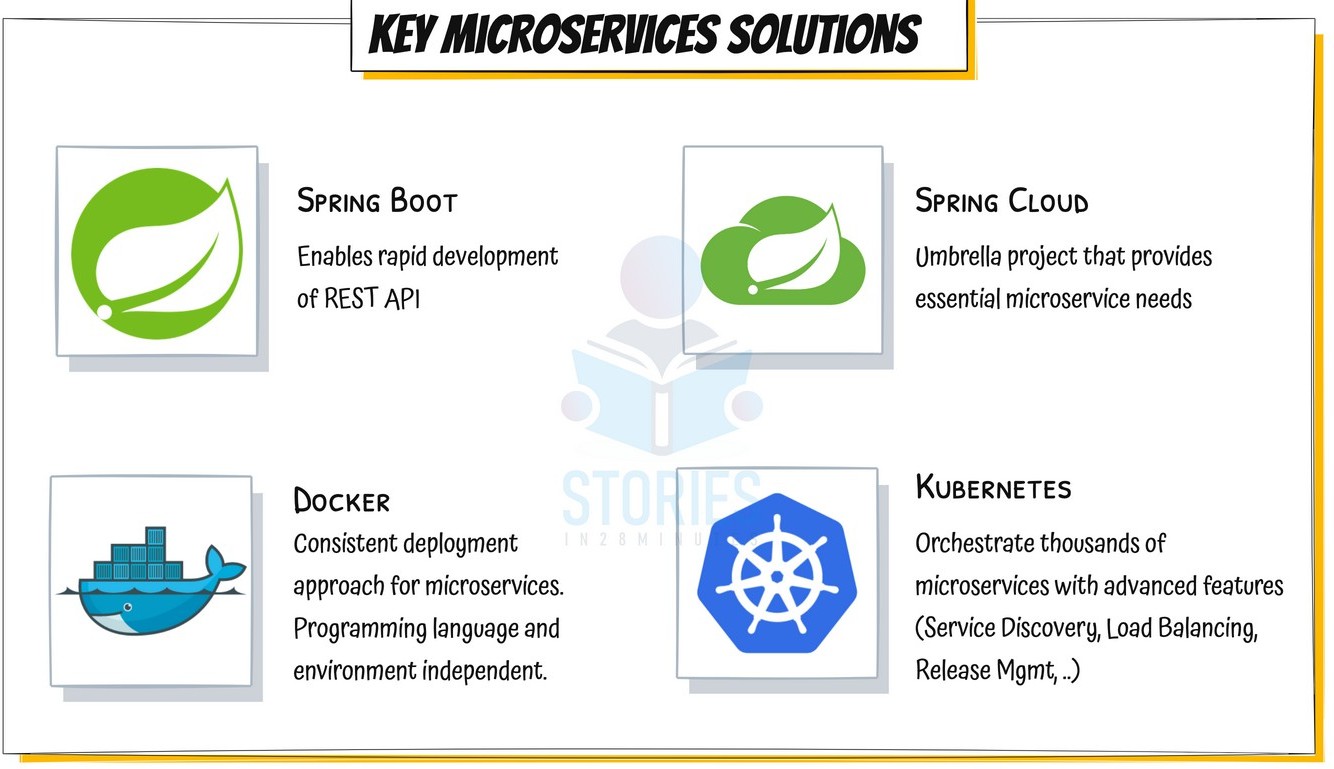




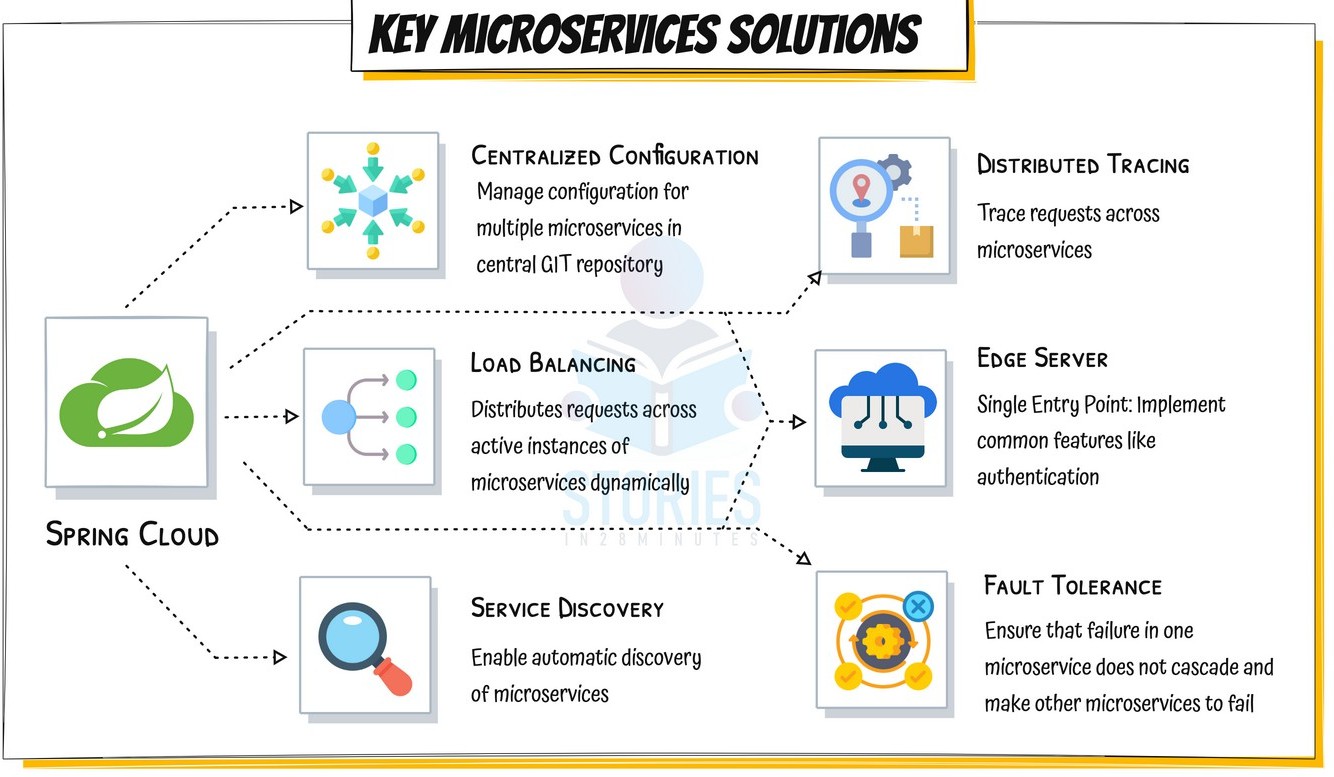














Microservices - Evolution



 **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)

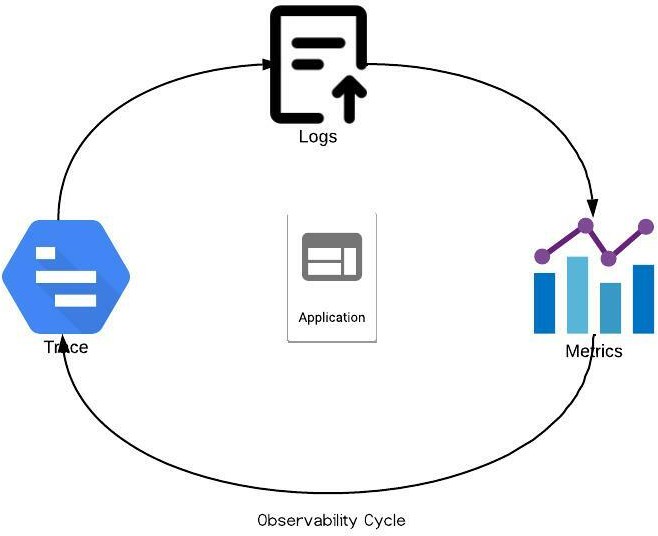
 Resilience4j (Hystrix)  NEW: Docker

 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***](https://github.com/in28minutes/spring-microservices-v2)

 Spring Boot 3.0.0+

 [***https://github.com/in28minutes/spring-microservices-v3***](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

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

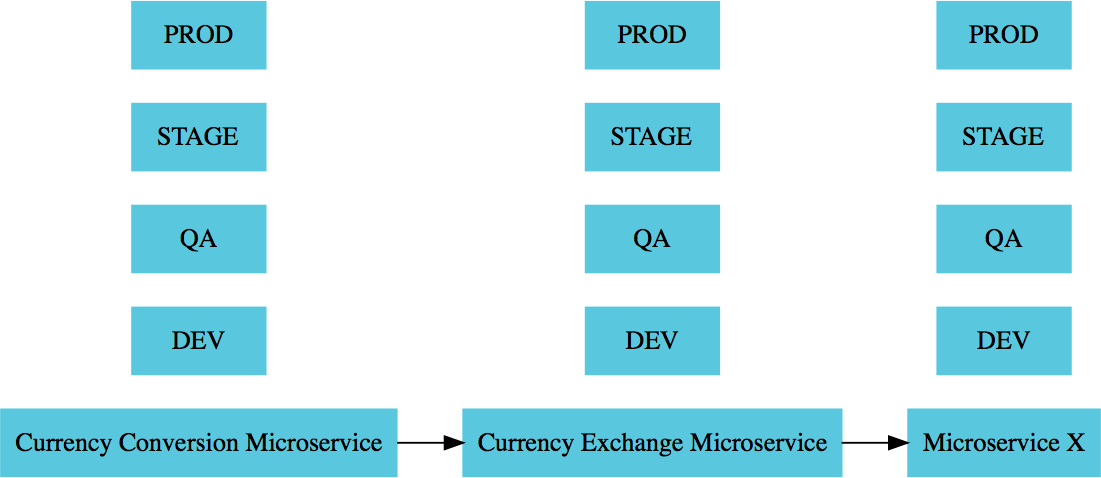
 **OpenTelemetry**: One Standard - Logs, Traces & Metrics

 **Micrometer** (Replaces Spring Cloud Sleuth)

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**

 Lot of configuration:

 External Services  Database

 Queue

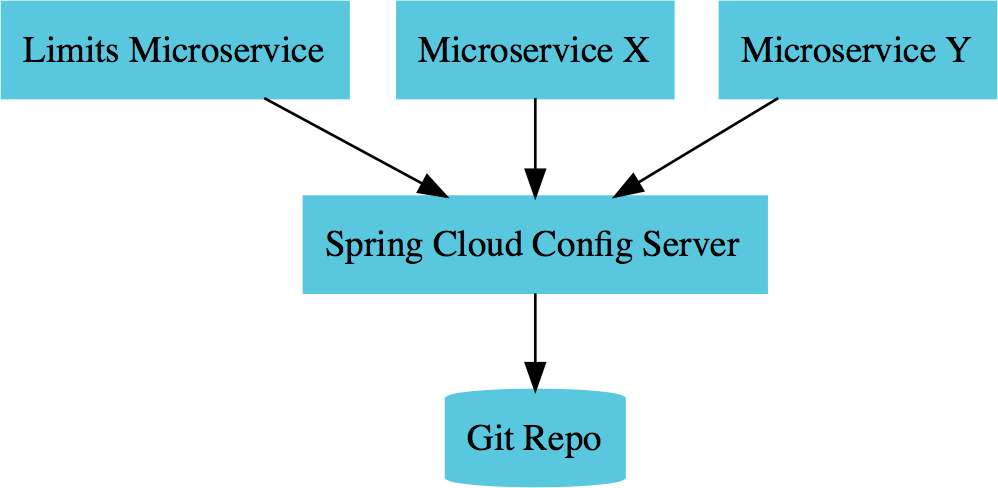
 Typical Application Configuration

 Configuration variations:

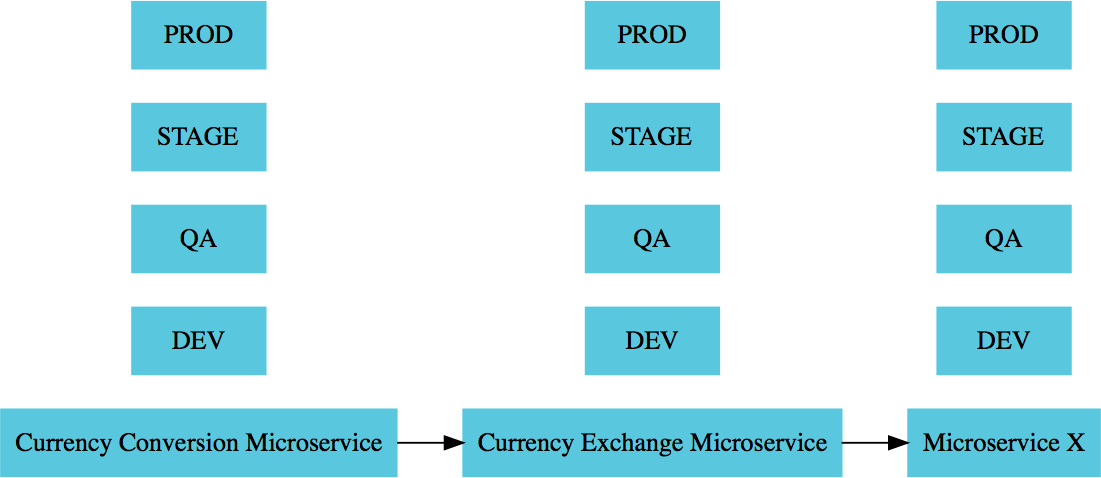
 1000s of Microservices  Multiple Environments

 Multiple instances in each Environment

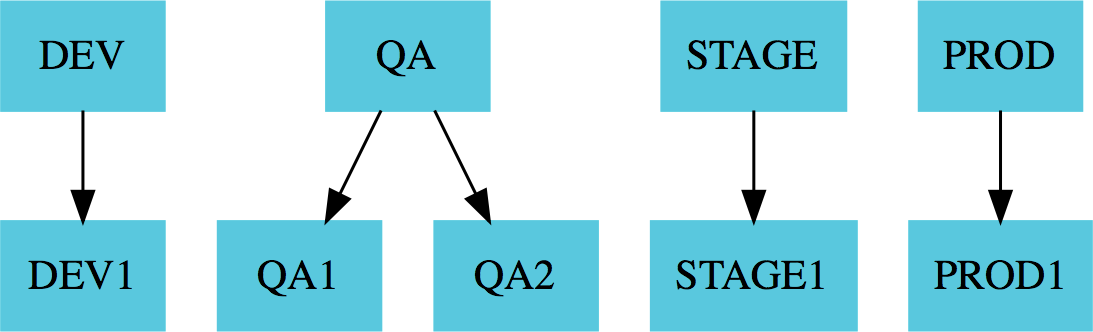
 How do you manage all this configuration?



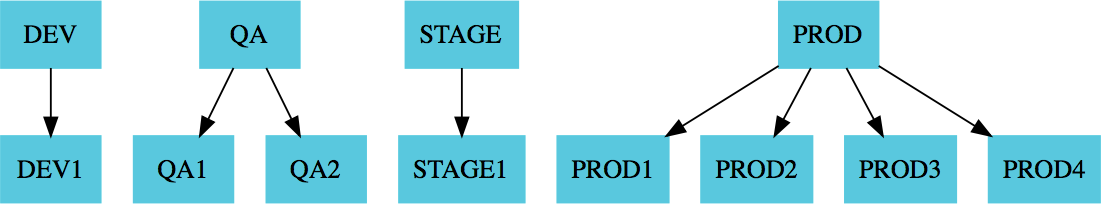
**Config Server**



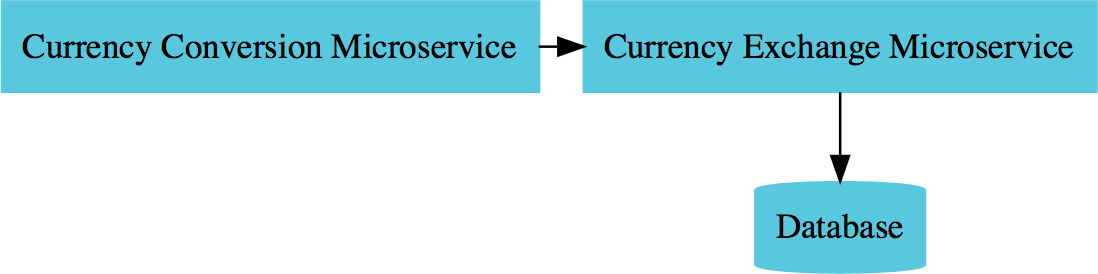
**Environments**



**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",

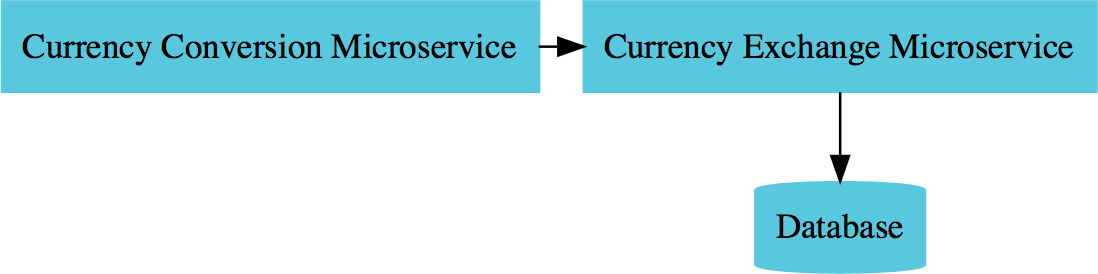
"conversionMultiple": 65.00,

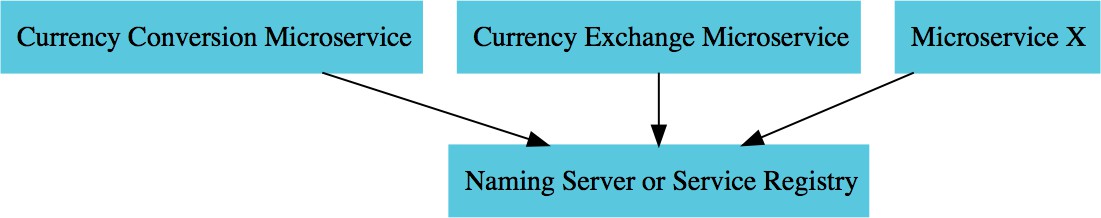
"quantity": 10,

"totalCalculatedAmount": 650.00,

"environment": "8000 instance-id"

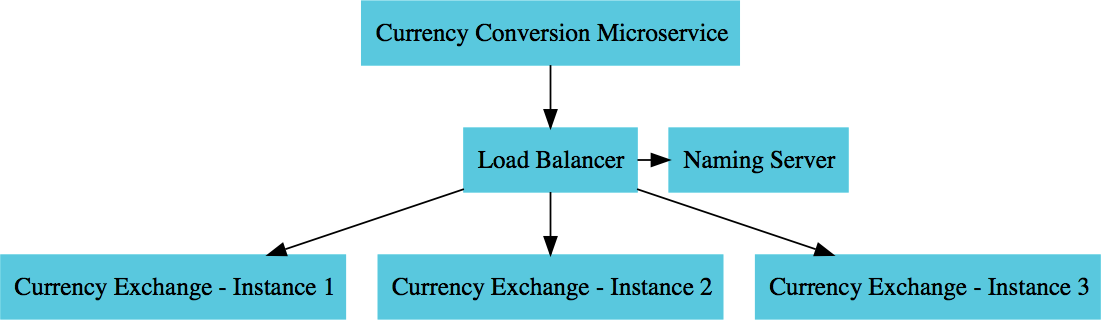
}

****



**Naming Server**





**Load Balancing**

Spring Cloud Gateway



 Simple, yet effective way to route to APIs  Provide cross cutting concerns:

 Security

 Monitoring/metrics

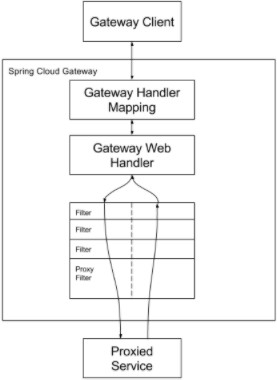
 Built on top of Spring WebFlux (Reactive Approach)

 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***](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

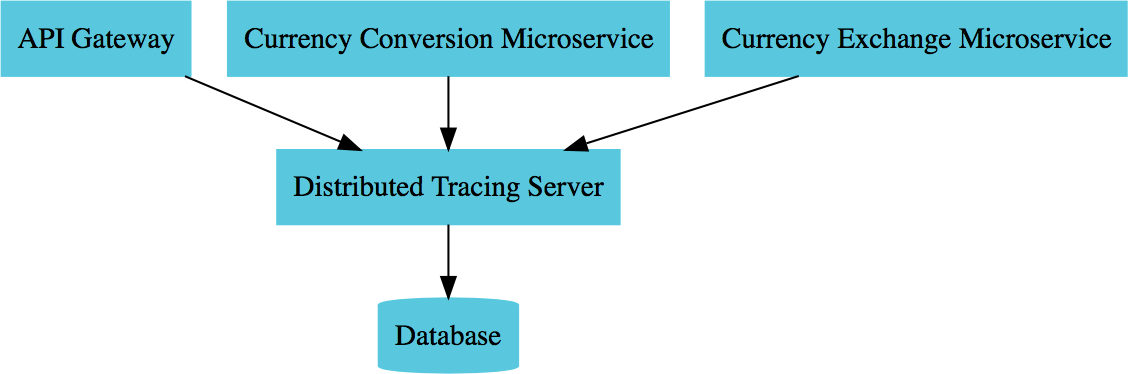
Distributed Tracing

****

 Complex call chain

 How do you debug problems?

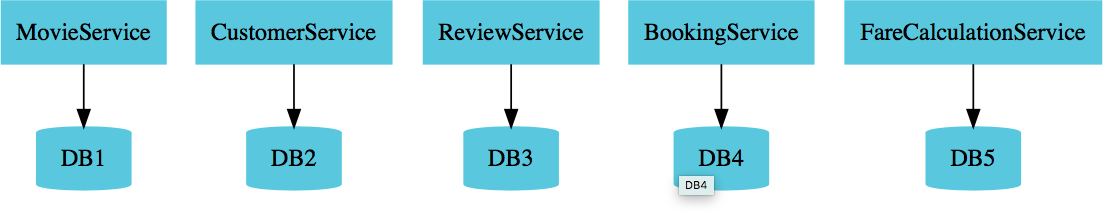
 How do you trace requests across microservices?  Enter Distributed Tracing





**Distributed Tracing**

Microservices

****

 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**!

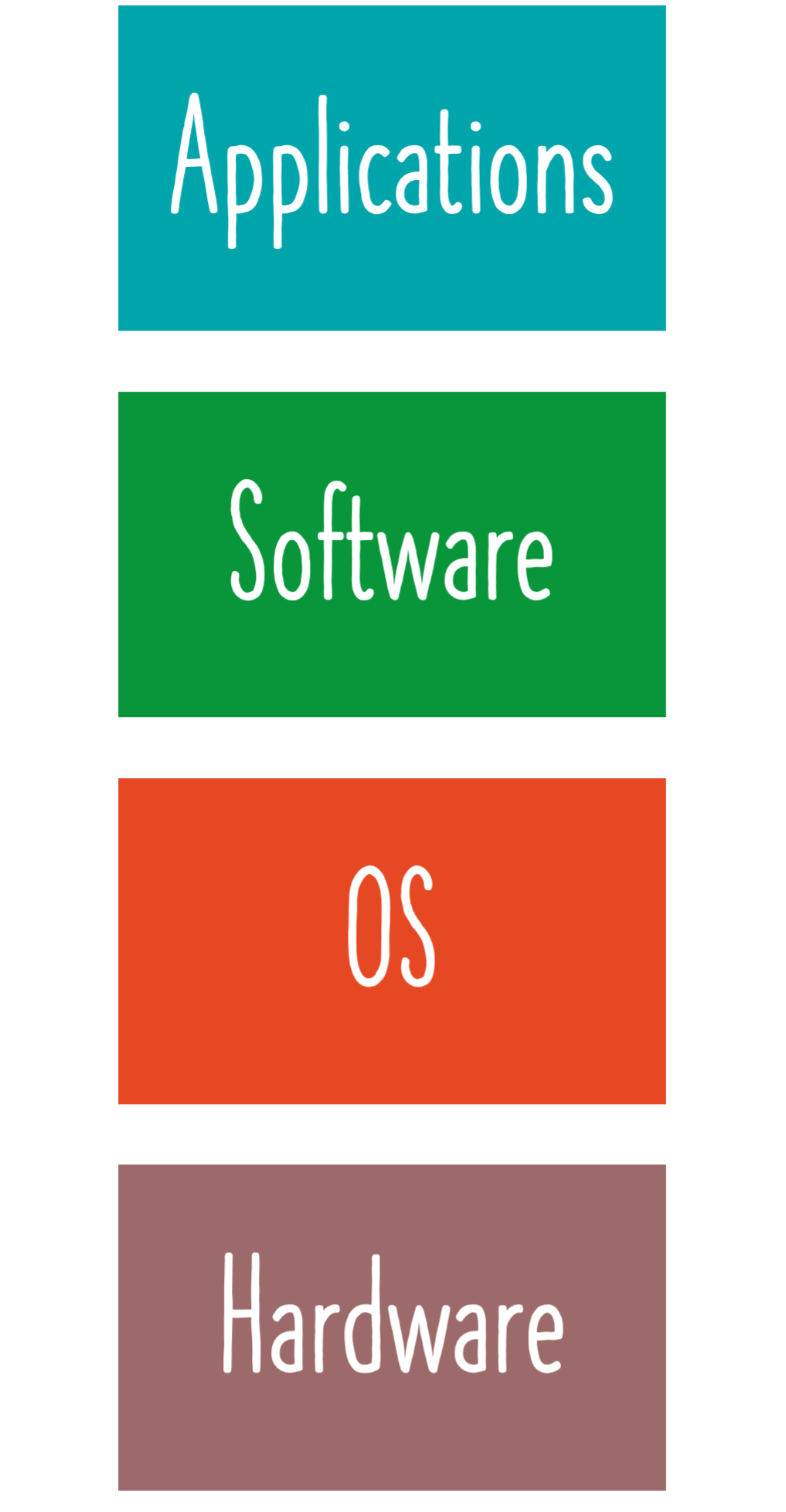
 How can we have **one way of deploying** Go, Java, Python or JavaScript .. microservices?

 Enter **containers**!

# Docker

**Getting Started**

##### How does Traditional Deployment work?

**** Deployment process **described in a document**

 Operations team **follows steps** to:

 Setup Hardware

 Setup OS (Linux, Windows, Mac, ...)

 Install Software (Java, Python, NodeJs, ...)  Setup Application Dependencies

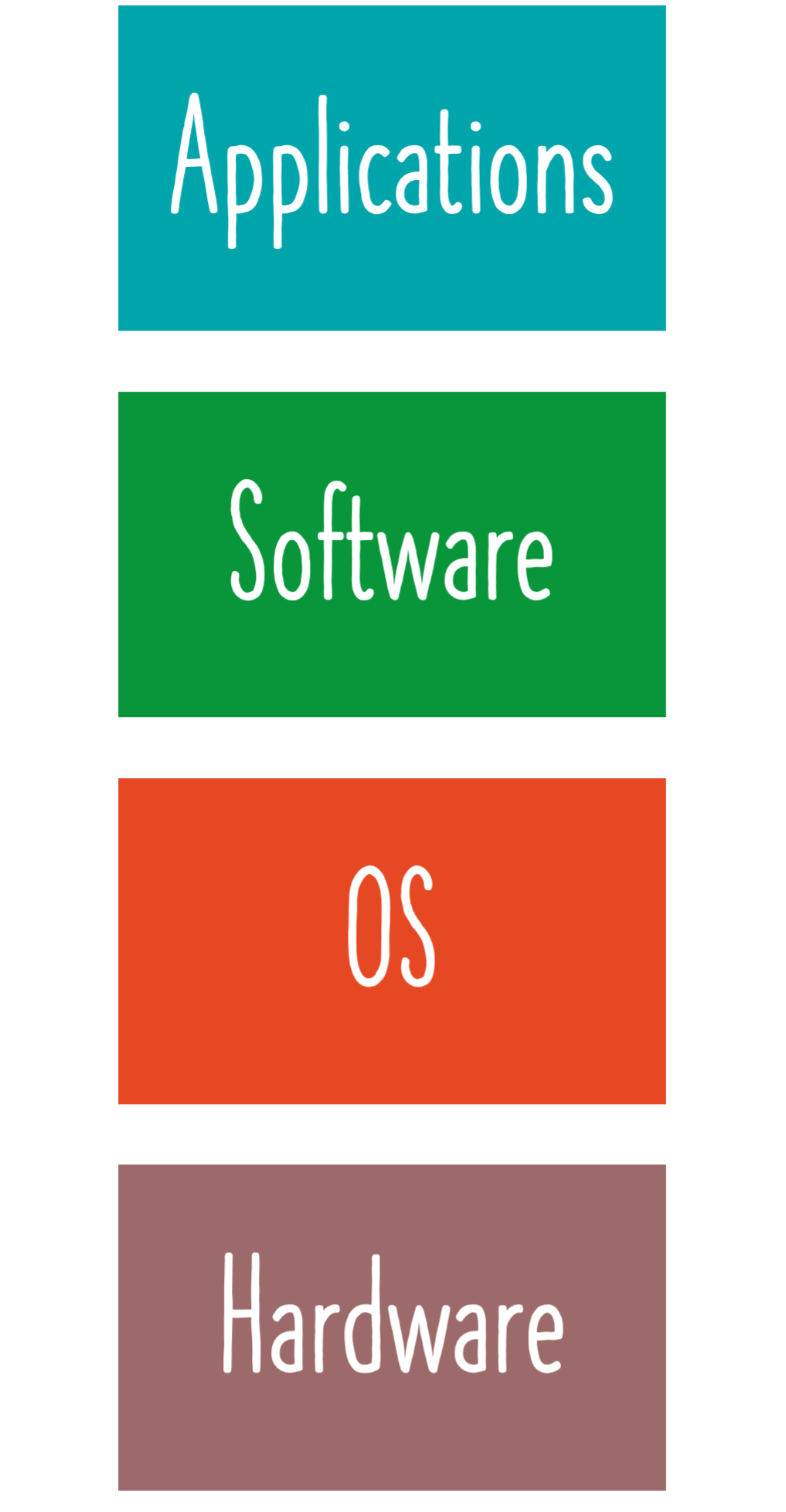
 Install Application

 **Manual approach**:

 Takes a lot of time

 High chance of making mistakes

Understanding Deployment Process with Docker

 **Simplified** Deployment Process:

 OS doesn't matter

 Programming Language does not matter  Hardware does not matter

 **01:** Developer creates a Docker Image

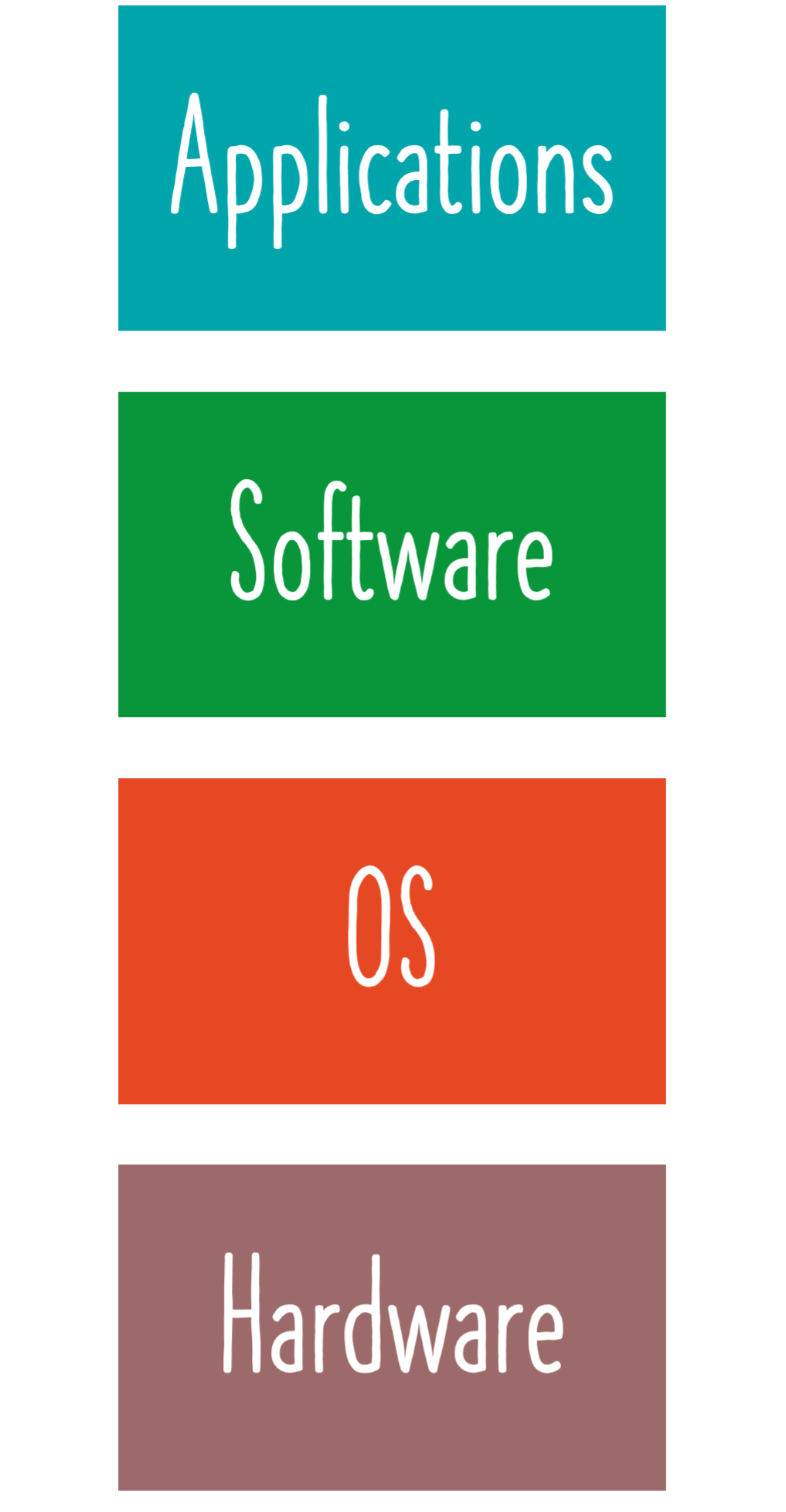
 **02:** Operations run the Docker Image

 Using a very simple command

 **Takeaway**: Once you have a Docker Image, irrespective of what the docker image contains, you run it the same way!

 Make your operations team happy

##### How does Docker Make it Easy?

 Docker image **has everything you need to run your application**:

 Operating System

 Application Runtime (JDK or Python or NodeJS)  Application code and dependencies

 You can run a Docker container **the same way**

everywhere:

 Your local machine

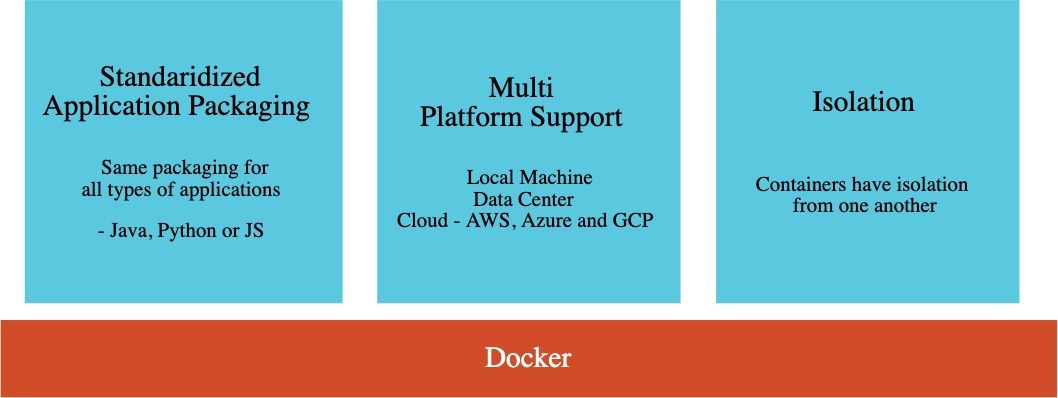
 Corporate data center  Cloud

Run Docker Containers Anywhere

****

 All that you need is a Docker Runtime (like Docker Engine)

Why is Docker Popular?

****

**What's happening in the Background?**

docker container run -d -p 5000:5000 in28min/hello-world-nodejs:0.0.1.RELEASE

 **Docker image** is downloaded from Docker Registry (Default: Docker Hub)

 [***https://hub.docker.com/r/in28min/hello-world-nodejs***](https://hub.docker.com/r/in28min/hello-world-nodejs)

 **Image** is a set of bytes

 **Container**: Running Image

 **in28min/hello-world-nodejs**: Repository Name

**0.0.1.RELEASE**: Tag (or version)

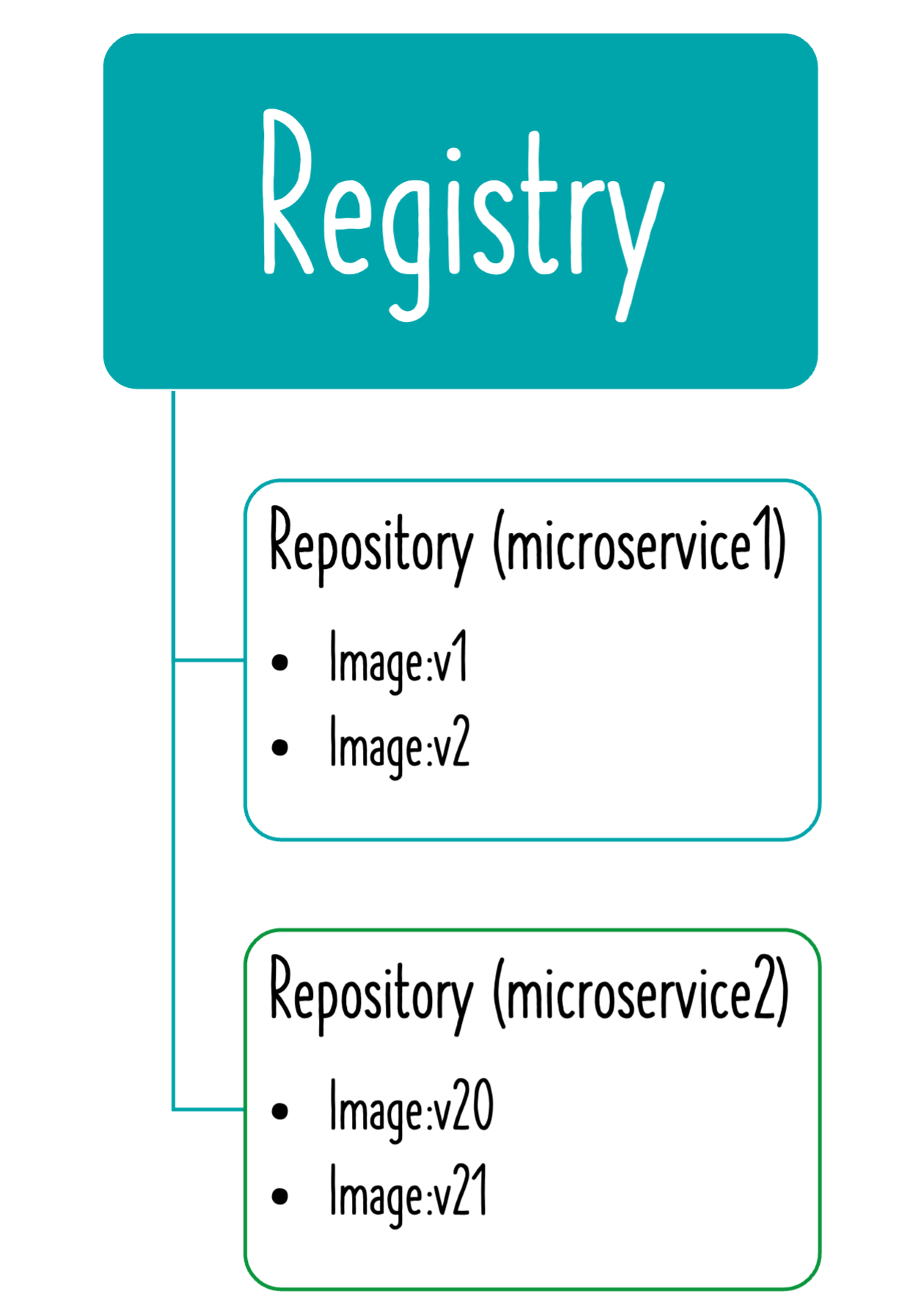
 **-p hostPort:containerPort**: Maps internal docker port (container port) to a port on the host (host port)

 By default, Docker uses its own internal network called bridge network  We are mapping a host port so that users can access your application

 **-d**: Detatched Mode (Don't tie up the terminal)

Understanding Docker Terminology



 **Docker Image**: A package representing specific version of your application (or software)

 Contains everything your app needs

 OS, software, code, dependencies

 **Docker Registry**: A place to store your docker images

 **Docker Hub**: A registry to host Docker images

 **Docker Repository**: Docker images for a specific app (tags are used to differentiate different images)

 **Docker Container**: Runtime instance of a docker image

 **Dockerfile**: File with instructions to create a Docker image

Dockerfile - 1 - Creating Docker Images

FROM openjdk:18.0-slim COPY target/\*.jar app.jar EXPOSE 5000

ENTRYPOINT ["java","-jar","/app.jar"]

 Dockerfile contains instruction to create Docker images

 **FROM** - Sets a base image

 **COPY** - Copies new files or directories into image

 **EXPOSE** - Informs Docker about the port that the container listens on at runtime

 **ENTRYPOINT** - Configure a command that will be run at container launch

 docker build -t in28min/hello-world:v1 .

Dockerfile - 2 - Build Jar File - Multi Stage

FROM maven:3.8.6-openjdk-18-slim AS build WORKDIR /home/app

COPY . /home/app

RUN mvn -f /home/app/pom.xml clean package

FROM openjdk:18.0-slim EXPOSE 5000

COPY --from=build /home/app/target/\*.jar app.jar ENTRYPOINT [ "sh", "-c", "java -jar /app.jar" ]

 Let build the jar file as part of creation of Docker Image

 Your build does NOT make use of anything built on your local machine

Dockerfile - 3 - Improve Layer Caching

FROM maven:3.8.6-openjdk-18-slim AS build WORKDIR /home/app

COPY ./pom.xml /home/app/pom.xml

COPY ./src/main/java/com/example/demodocker/DemoDockerApplication.java /

/home/app/src/main/java/com/example/demodocker/DemoDockerApplication.java RUN mvn -f /home/app/pom.xml clean package

COPY . /home/app

RUN mvn -f /home/app/pom.xml clean package

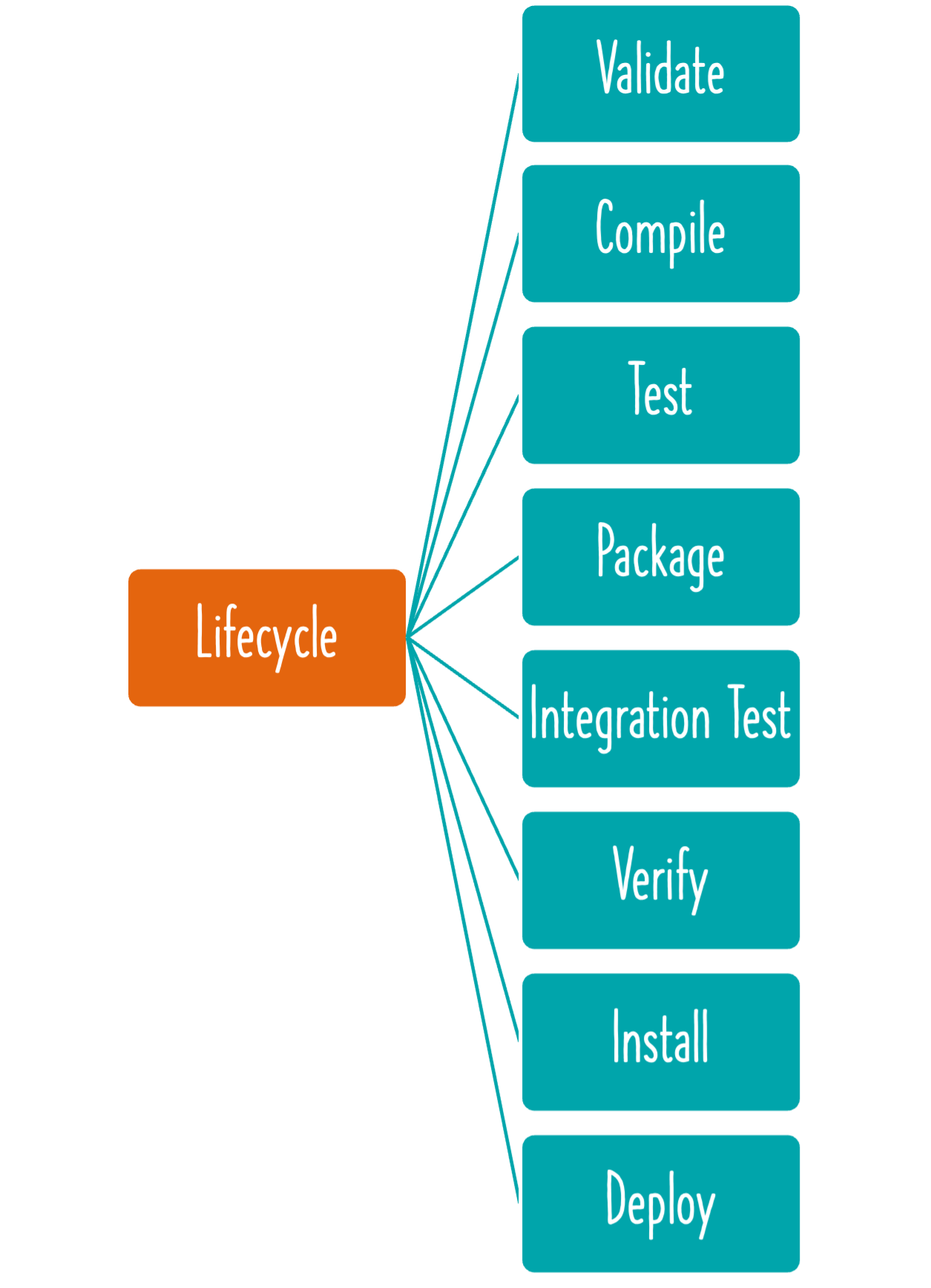
FROM openjdk:18.0-slim EXPOSE 5000

COPY --from=build /home/app/target/\*.jar app.jar ENTRYPOINT [ "sh", "-c", "java -jar /app.jar" ]

 Docker caches every layer and tries to reuse it

 Let's make use of this feature to make our build efficient

Spring Boot Maven Plugin - Create Docker Image

 **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)

Creating Docker Images - Dockerfile

FROM node:8.16.1-alpine WORKDIR /app

COPY . /app RUN npm install EXPOSE 5000

CMD node index.js

 Dockerfile contains instruction to create Docker images

 **FROM** - Sets a base image

 **WORKDIR** - sets the working directory

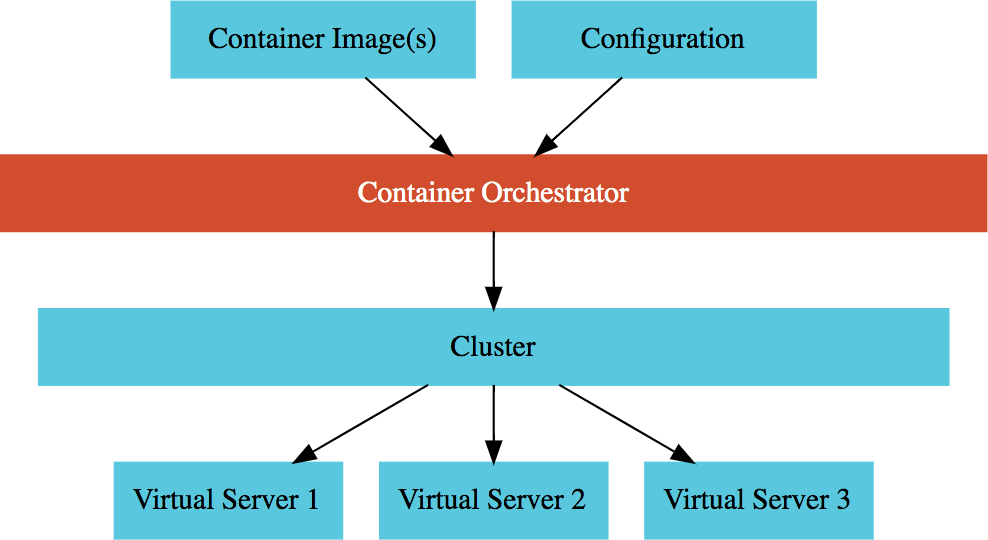
 **RUN** - execute a command

 **EXPOSE** - Informs Docker about the port that the container listens on at runtime

 **COPY** - Copies new files or directories into image

 **CMD** - Default command for an executing container

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

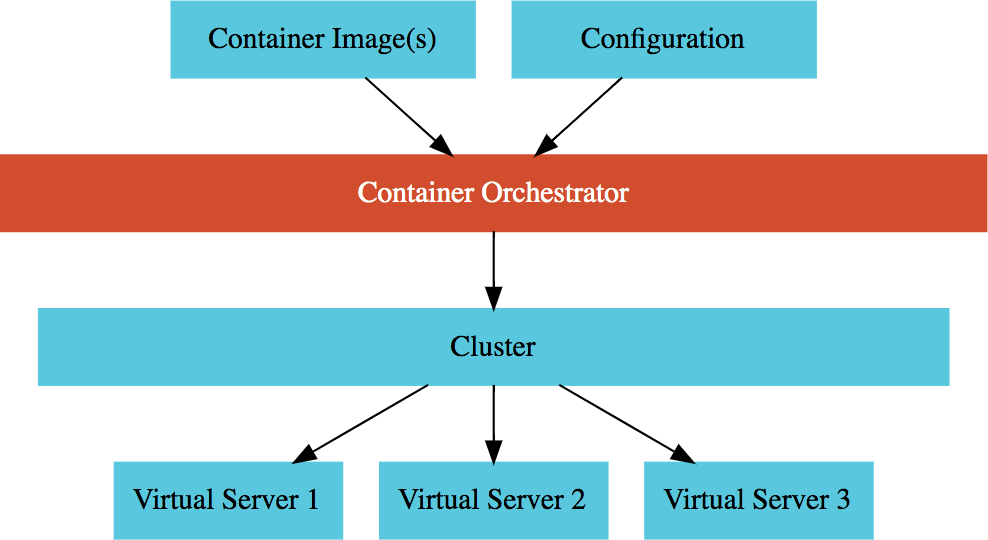
 **Service Discovery** - Help microservices find one another

 **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

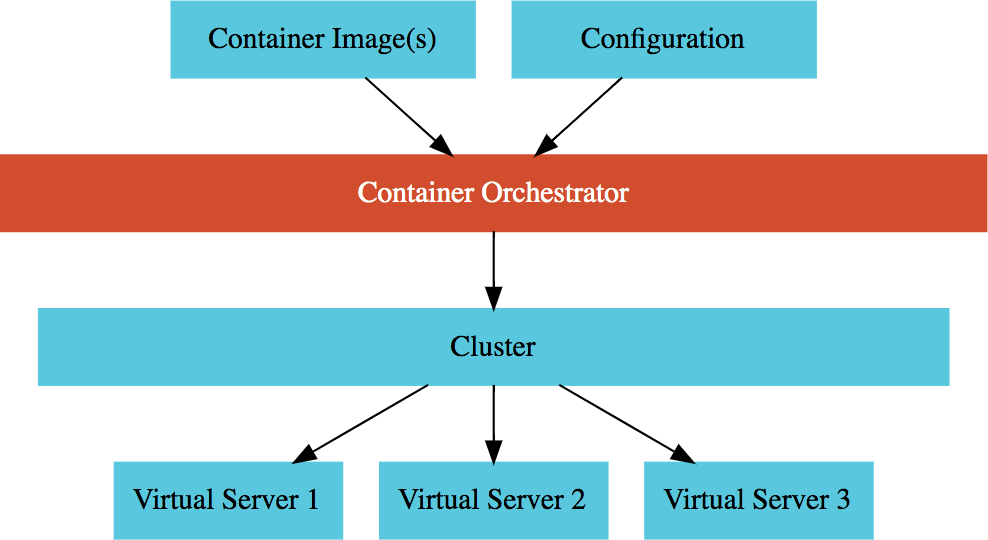
 AWS - Elastic Kubernetes Service (EKS)  Azure - Azure Kubernetes Service (AKS)  GCP - Google Kubernetes Engine (GKE)

 EKS/AKS does not have a free tier!

 We use GCP and GKE!

# Kubernetes

Kubernetes

 Most popular open source container orchestration solution

 Provides Cluster Management (including upgrades)

 Each cluster can have different types of

virtual machines

 Provides all important container orchestration features:

 **Auto Scaling**

 **Service Discovery ** **Load Balancer**

 **Self Healing**

 **Zero Downtime Deployments**

Google Kubernetes Engine (GKE)

 **Managed** Kubernetes service

 Minimize operations with **auto-repair** (repair failed nodes) and

**auto-upgrade** (use latest version of K8S always) features  Provides **Pod and Cluster Autoscaling**

 Enable **Cloud Logging** and **Cloud Monitoring** with simple configuration

 Uses **Container-Optimized OS**, a hardened OS built by Google  Provides support for **Persistent disks** and **Local SSD**

Kubernetes - A Microservice Journey - Getting Started

 **Let's Have Some Fun**: Let's get on a journey with Kubernetes:

 Let's create a cluster, deploy a microservice and play with it in **13 steps**!

 **1:** Create a Kubernetes cluster with the default node pool

 *gcloud container* ***clusters create*** or use cloud console

 **2:** Login to Cloud Shell

 **3:** Connect to the Kubernetes Cluster

 *gcloud container* ***clusters get-credentials*** *my-cluster --zone us-central1-a --project solid-course-258105*

Kubernetes - A Microservice Journey - Deploy Microservice

 **4:** Deploy Microservice to Kubernetes:

 Create deployment & service using kubectl commands

 *kubectl* ***create deployment*** *hello-world-rest-api --image=in28min/hello-world-rest-api:0.0.1.RELEASE * *kubectl* ***expose deployment*** *hello-world-rest-api --type=LoadBalancer --port=8080*

 **5:** Increase number of instances of your microservice:

 *kubectl* ***scale deployment*** *hello-world-rest-api --replicas=2*

 **6:** Increase number of nodes in your Kubernetes cluster:

 *gcloud container clusters resize* ***my-cluster --node-pool my-node-pool --num-nodes 5***

 You are NOT happy about manually increasing number of instances and nodes!

Kubernetes - A Microservice Journey - Auto Scaling and ..

 **7:** Setup auto scaling for your microservice:

 *kubectl* ***autoscale deployment*** *hello-world-rest-api --max=10 --cpu-percent=70*

 Also called horizontal pod autoscaling - HPA - *kubectl get hpa*

 **8:** Setup auto scaling for your Kubernetes Cluster

 *gcloud* ***container clusters update*** *cluster-name --enable-autoscaling --min-nodes=1 -- max-nodes=10*

 **9:** Add some application configuration for your microservice

 Config Map - *kubectl* ***create configmap*** *todo-web-application-config --from- literal=RDS\_DB\_NAME=todos*

 **10:** Add password configuration for your microservice

 Kubernetes Secrets - *kubectl* ***create secret*** *generic todo-web-application-secrets-1 -- from-literal=RDS\_PASSWORD=dummytodos*

**Kubernetes Deployment YAML - Deployment**

apiVersion: apps/v1 kind: Deployment metadata:

labels:

app: hello-world-rest-api name: hello-world-rest-api namespace: default

spec:

replicas: 3 selector:

matchLabels:

app: hello-world-rest-api template:

metadata: labels:

app: hello-world-rest-api spec:

containers:

- image: in28min/hello-world-rest-api:0.0.3.RELEASE name: hello-world-rest-api

**Kubernetes Deployment YAML - Service**

apiVersion: v1 kind: Service metadata:

labels:

app: hello-world-rest-api name: hello-world-rest-api namespace: default

spec:

ports:

- port: 8080 protocol: TCP targetPort: 8080

selector:

app: hello-world-rest-api sessionAffinity: None type: LoadBalancer

**Kubernetes - A Microservice Journey - The End!**

 **11:** Deploy a new microservice which needs nodes with a GPU attached

 Attach a new node pool with GPU instances to your cluster

 gcloud **container node-pools create** POOL\_NAME --cluster CLUSTER\_NAME  gcloud container node-pools list --cluster CLUSTER\_NAME

 Deploy the new microservice to the new pool by setting up nodeSelector in the

deployment.yaml

 nodeSelector: cloud.google.com/gke-nodepool: POOL\_NAME

 **12:** Delete the Microservices

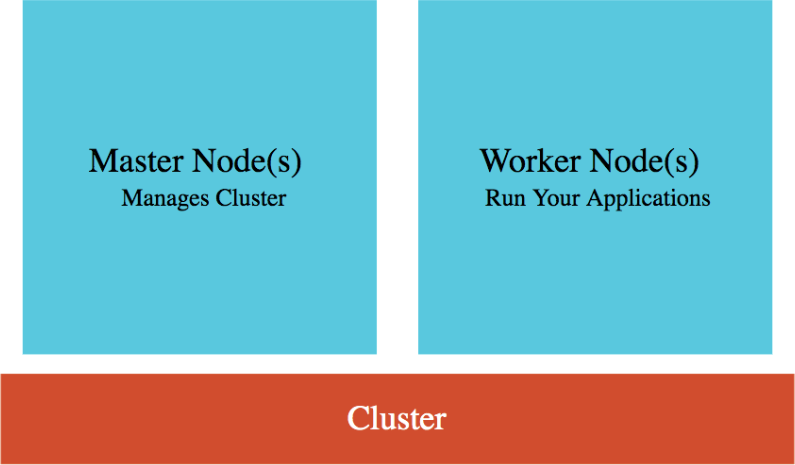
 Delete service - *kubectl* ***delete service***

 Delete deployment - *kubectl* ***delete deployment***

 **13:** Delete the Cluster

 *gcloud container* ***clusters delete***

Google Kubernetes Engine (GKE) Cluster

 **Cluster** : Group of Compute Engine instances:

 **Master Node(s)** - Manages the cluster

 **Worker Node(s)** - Run your workloads (pods)

 **Master Node** (Control plane) components:

 **API Server** - Handles all communication for a K8S cluster (from nodes and outside)

 **Scheduler** - Decides placement of pods

 **Control Manager** - Manages deployments & replicasets

 **etcd** - Distributed database storing the cluster state

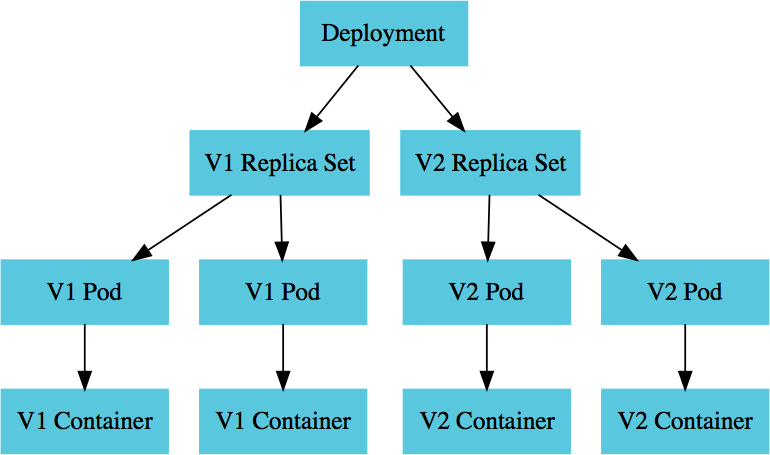
 **Worker Node** components:

 Runs your pods

 **Kubelet** - Manages communication with master node(s)

Kubernetes - Pods



**** Smallest deployable unit in Kubernetes  A Pod contains **one or more containers**

 Each Pod is assigned an ephemeral **IP address**

 All containers in a pod share:

 Network  Storage

 IP Address

 Ports and

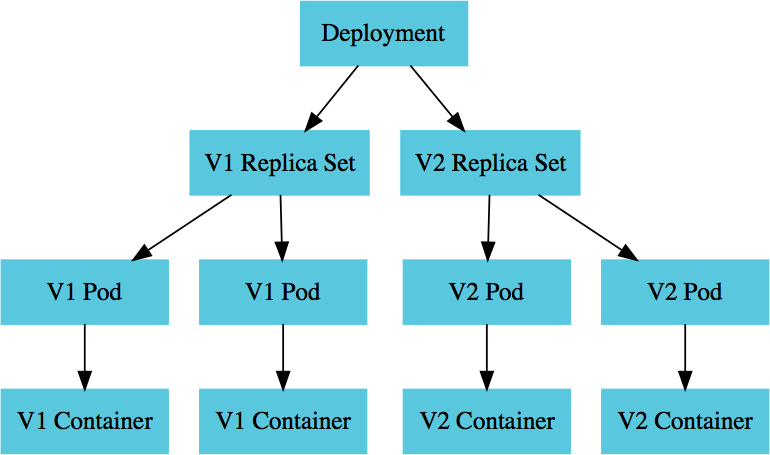
 Volumes (Shared persistent disks)

 POD statuses : Running /Pending /Succeeded

/Failed /Unknown

Kubernetes - Deployment vs Replica Set



 A **deployment** is created for each microservice:

 kubectl create deployment m1 --image=m1:v1

 Deployment represents a microservice (with all its releases)  Deployment manages new releases ensuring zero downtime

 **Replica set** ensures that a specific number of pods are running for a specific microservice version

 kubectl scale deployment m2 --replicas=2

 Even if one of the pods is killed, replica set will launch a new one

 Deploy V2 of microservice - Creates a new replica set

 kubectl set image deployment m1 m1=m1:v2

 V2 Replica Set is created

 Deployment updates V1 Replica Set and V2 Replica Set based on the release strategies

Kubernetes - Service



 Each Pod has its **own IP address**:

 How do you ensure that external users are not impacted when:

 A pod fails and is replaced by replica set

 A new release happens and all existing pods of old release are replaced by ones of new release

 Create **Service**

 *kubectl expose deployment name --type=LoadBalancer --port=80*

 Expose PODs to outside world using a stable IP Address

 Ensures that the external world does not get impacted as pods go down and come up

 Three Types:

 **ClusterIP**: Exposes Service on a cluster-internal IP

 Use case: You want your microservice only to be available inside the cluster (Intra cluster communication)

 **LoadBalancer**: Exposes Service externally using a cloud provider's load balancer

 Use case: You want to create individual Load Balancer's for each microservice

 **NodePort**: Exposes Service on each Node's IP at a static port (the NodePort)

 Use case: You DO not want to create an external Load Balancer for each microservice (You can create one Ingress

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 (>=2.3) provides inbuilt readiness and liveness probes:

 /health/readiness  /health/liveness

**What Next?**

