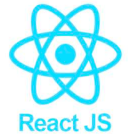


# Master Spring & Spring Boot with Hibernate & React

# Getting Started

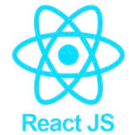
- **Top frameworks in the Java world today**
  - Spring Framework
  - Spring Boot
- **Beginners find the first steps very difficult:**
  - **Lot of terminology:** Dependency Injection, IOC, Auto wiring, Auto configuration, Starter Projects ..
  - **Variety of applications:** Web app, REST API, Full Stack
  - **Variety of other framework, tool and platform integrations:** Maven, Gradle, Spring Data, JPA, Hibernate, Docker and Cloud



# Simple Path - Learn Spring & Spring Boot & ...

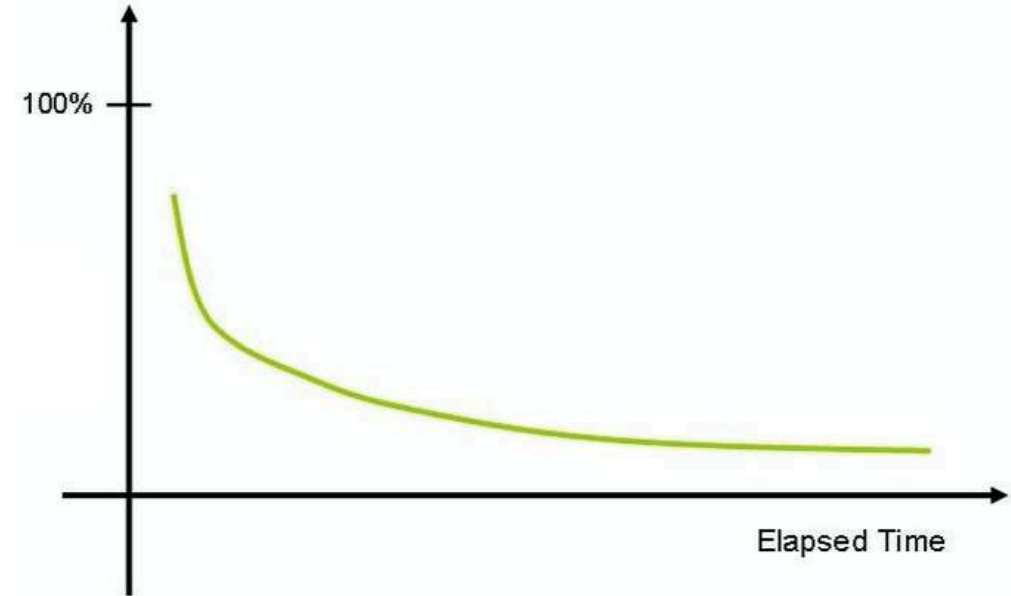
In 28  
Minutes

- We've created a **simple path** focusing on the **Fundamentals**
  - Using a Hands-on Approach
- You will build more than 10 Maven and Gradle projects during this course!
- This course is **designed** for absolute beginners to Spring & Spring Boot
- **Our Goal** : Help you start your journey with Spring & Spring Boot



# How do you put your best foot forward?

- Learning Spring & Spring Boot can be tricky:
  - Lots of new terminology, tools and frameworks
- As time passes, we forget things
- How do you **improve your chances** of remembering things?
  - Active learning - think & make notes
  - Review the presentation once in a while



# Our Approach

- **Videos with:**
  - Presentations &
  - Demos where we build projects
- **Quizzes:**
  - To Reinforce Concepts
- (Recommended) Take your time. Do not hesitate to replay videos!
- (Recommended) Have Fun!



# FASTEST ROADMAPS

in28minutes.com



In28  
Minutes



Google Cloud  
Certifications



Azure  
Certifications



AWS  
Certifications



DevOps



Java Full Stack



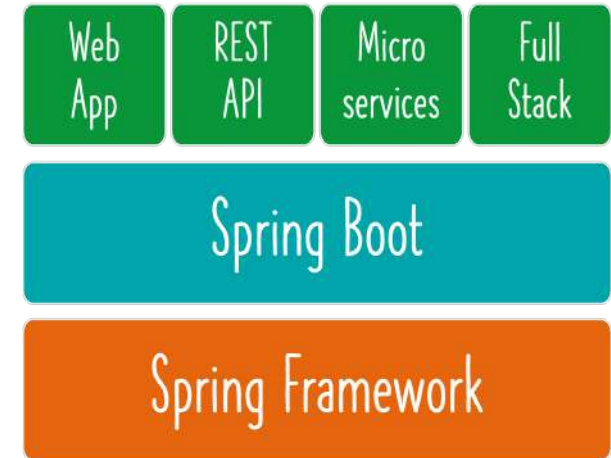
Java Microservices



# Spring Framework

# Getting Started with Spring Framework - Why?

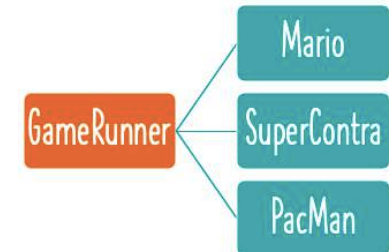
- You can build a **variety of applications** using Java, Spring and Spring Boot:
  - Web
  - REST API
  - Full Stack
  - Microservices
- Irrespective of the app you are building:
  - **Spring framework** provides all the **core features**
    - Understanding Spring helps you learn Spring Boot easily
    - Helps in debugging problems quickly
- In this module, we focus extensively on Spring Framework





# Getting Started with Spring Framework - Goals

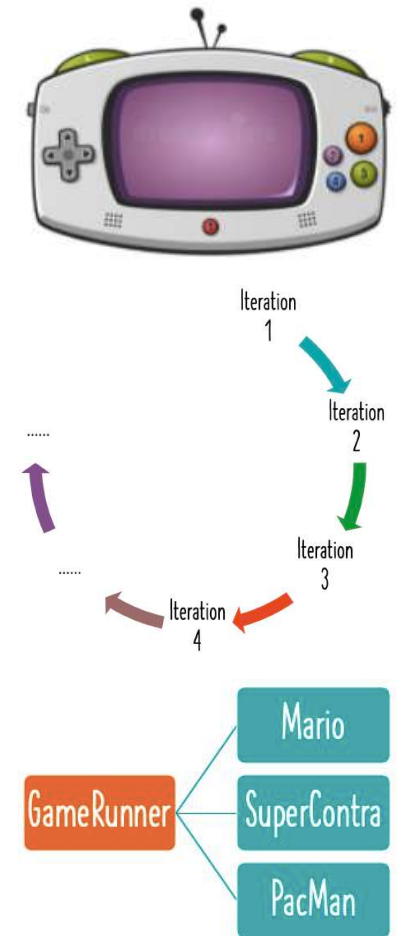
- **Goal:** Understand core features of Spring Framework
- **Approach:** Build a Loose Coupled Hello World Gaming App with **Modern Spring** Approach
  - Get **Hands-on** with Spring and understand:
    - Why Spring?
    - **Terminology**
      - Tight Coupling and Loose Coupling
      - IOC Container
      - Application Context
      - Component Scan
      - Dependency Injection
      - Spring Beans
      - Auto Wiring



# Getting Started with Spring Framework - Approach

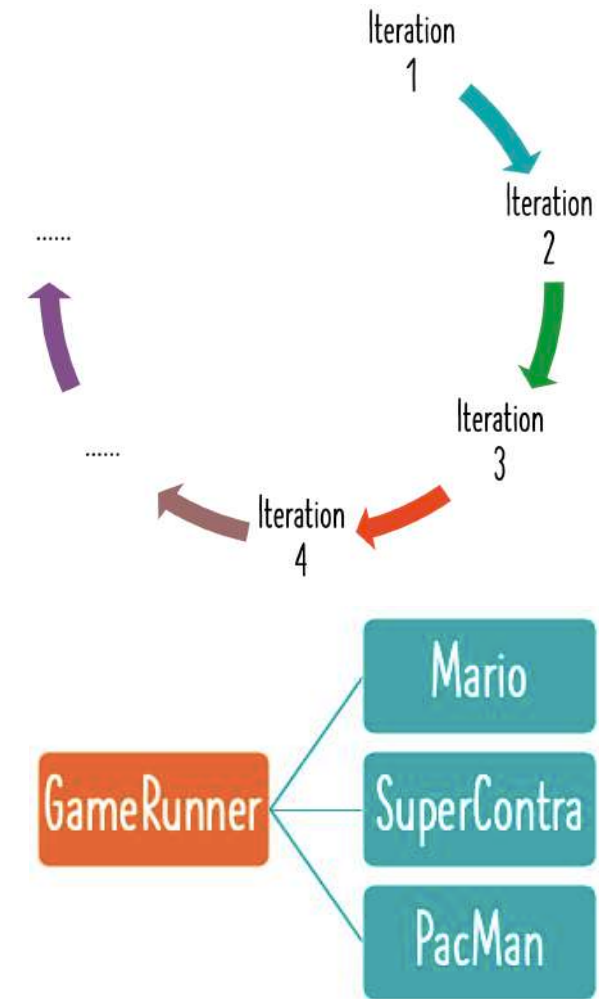
- Design Game Runner to run games (Mario, SuperContra, Pacman etc) in an **iterative approach**:

- **Iteration 1: Tightly Coupled Java Code**
  - GameRunner class
  - Game classes: Mario, SuperContra, Pacman etc
- **Iteration 2: Loose Coupling - Interfaces**
  - GameRunner class
  - GamingConsole interface
    - Game classes: Mario, SuperContra, Pacman etc
- **Iteration 3: Loose Coupling - Spring Level 1**
  - Spring Beans
  - Spring framework will manage objects and wiring
- **Iteration 4: Loose Coupling - Spring Level 2**
  - Spring Annotations
  - Spring framework will create, manage & auto-wire objects



# Do you want to do well at interviews?

- **Very Few Spring Framework** users understand fundamentals:
  - Spring Container vs Spring Context vs IOC Container vs Application Context
  - Java Bean vs Spring Bean
  - Auto Wiring vs Dependency Injection
  - How can you build loosely coupled applications?
  - Making good use of Java Interfaces along with Spring
- We spent **multiple weeks** designing this section to help you understand these fundamentals
- This is the **most important section** of the course:
  - Focus really well to understand the fundamentals
  - Good luck!



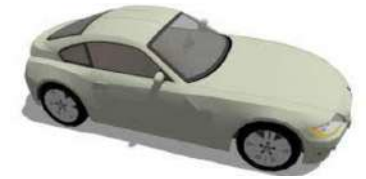
# Local Variable Type Inference

```
// List<String> numbers = new ArrayList<>(list);  
var numbers = new ArrayList<>(list);
```

- Java compiler infers the type of the variable at compile time
- Introduced in Java 10
- You can add final if you want
- var can also be used in loops
- Remember:
  - You cannot assign null
  - var is NOT a keyword
  - CANNOT be used for member variables, method parameters or return types
- Best Practices:
  - Good variable names
  - Minimize Scope

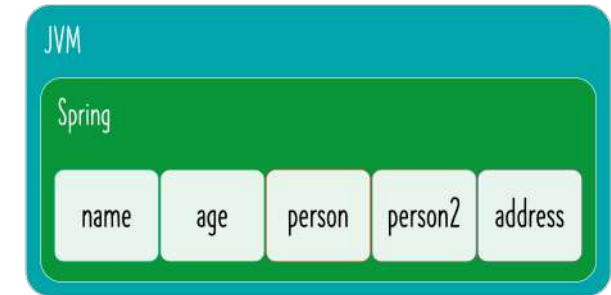
# Why is Coupling Important?

- **Coupling:** How much work is involved in changing something?
  - **Coupling** is important everywhere:
    - An engine is tightly coupled to a Car
    - A wheel is loosely coupled to a Car
    - You can take a laptop anywhere you go
    - A computer, on the other hand, is a little bit more difficult to move
  - Coupling is **even more important** in building great software
    - Only thing constant in technology is change
      - Business requirements change
      - Frameworks change
      - Code changes
    - We want Loose Coupling as much as possible
    - We want to make functional changes with as less code changes as possible
- Let's explore how Java Interfaces and Spring Framework help with Loose Coupling!



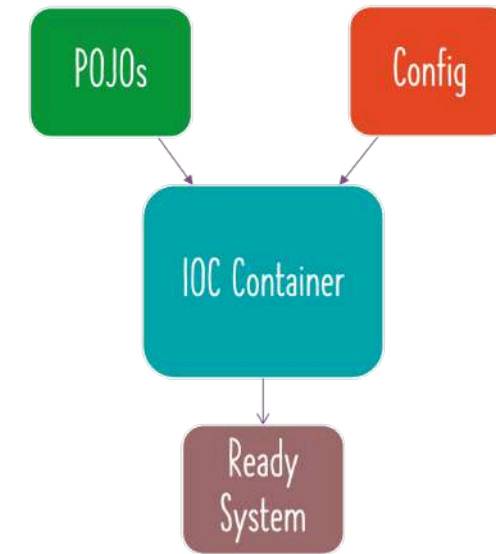
# Spring Questions You Might Be Thinking About

- **Question 1:** Spring Container vs Spring Context vs IOC Container vs Application Context
- **Question 2:** Java Bean vs Spring Bean
- **Question 3:** How can I list all beans managed by Spring Framework?
- **Question 4:** What if multiple matching beans are available?
- **Question 5:** Spring is managing objects and performing auto-wiring.
  - BUT aren't we writing the code to create objects?
  - How do we get Spring to create objects for us?
- **Question 6:** Is Spring really making things easy?



# What is Spring Container?

- **Spring Container:** Manages Spring beans & their lifecycle
- **1: Bean Factory:** Basic Spring Container
- **2: Application Context:** Advanced Spring Container with enterprise-specific features
  - Easy to use in web applications
  - Easy internationalization
  - Easy integration with Spring AOP
- **Which one to use?:** Most enterprise applications use Application Context
  - Recommended for web applications, web services - REST API and microservices



# Exploring Java Bean vs POJO vs Spring Bean

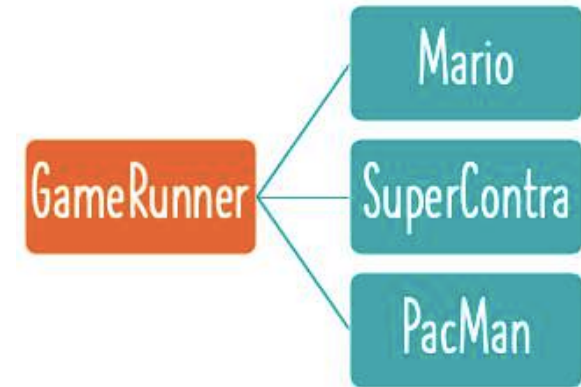
- **Java Bean:** Classes adhering to 3 constraints:
  - 1: Have public default (no argument) constructors
  - 2: Allow access to their properties using getter and setter methods
  - 3: Implement `java.io.Serializable`
- **POJO:** Plain Old Java Object
  - No constraints
  - Any Java Object is a POJO!
- **Spring Bean:** Any Java object that is managed by Spring
  - Spring uses IOC Container (Bean Factory or Application Context) to manage these objects

A teal rectangular box with the text "POJO" in white, sans-serif font, centered within the box.A green rectangular box with the text "Java Bean" in white, sans-serif font, centered within the box.An orange rectangular box with the text "Spring Bean" in white, sans-serif font, centered within the box.



# Exploring Spring - Dependency Injection Types

- **Constructor-based** : Dependencies are set by creating the Bean using its Constructor
- **Setter-based** : Dependencies are set by calling setter methods on your beans
- **Field**: No setter or constructor. Dependency is injected using reflection.
- **Question: Which one should you use?**
  - Spring team recommends Constructor-based injection as dependencies are automatically set when an object is created!



# Exploring auto-wiring in depth

- When a dependency needs to be @Autowired, IOC container looks for matches/candidates (by name and/or type)
  - **1: If no match is found**
    - **Result:** Exception is thrown
    - You need to help Spring Framework find a match
      - Typical problems:
        - @Component (or ..) missing
        - Class not in component scan
  - **2: One match is found**
    - **Result:** Autowiring is successful
  - **3: Multiple candidates**
    - **Result:** Exception is thrown
    - You need to help Spring Framework choose between the candidates
      - 1: Mark one of them as @Primary
        - If only one of the candidates is marked @Primary, it becomes the auto-wired value
      - 2: Use @Qualifier - Example: @Qualifier("myQualifierName")
        - Provides more specific control
        - Can be used on a class, member variables and method parameters



# @Primary vs @Qualifier - Which one to use?

```
@Component @Primary
class QuickSort implement SortingAlgorithm {}

@Component
class BubbleSort implement SortingAlgorithm {}

@Component @Qualifier("RadixSortQualifier")
class RadixSort implement SortingAlgorithm {}

@Component
class ComplexAlgorithm
    @Autowired
    private SortingAlgorithm algorithm;

@Component
class AnotherComplexAlgorithm
    @Autowired @Qualifier("RadixSortQualifier")
    private SortingAlgorithm iWantToUseRadixSortOnly;
```

- **@Primary** - A bean should be given preference when multiple candidates are qualified
- **@Qualifier** - A specific bean should be auto-wired (name of the bean can be used as qualifier)
- **ALWAYS** think from the perspective of the class using the SortingAlgorithm:
  - **1: Just @Autowired:** Give me (preferred) SortingAlgorithm
  - **2: @Autowired + @Qualifier:** I only want to use specific SortingAlgorithm - RadixSort
  - (REMEMBER) @Qualifier has higher priority then @Primary

# Spring Framework - Important Terminology

- **@Component** (..): An instance of class will be managed by Spring framework
- **Dependency**: GameRunner needs GamingConsole impl!
  - GamingConsole Impl (Ex: MarioGame) is a dependency of GameRunner
- **Component Scan**: How does Spring Framework find component classes?
  - It scans packages! (@ComponentScan("com.in28minutes"))
- **Dependency Injection**: Identify beans, their dependencies and wire them together (provides **IOC** - Inversion of Control)
  - **Spring Beans**: An object managed by Spring Framework
  - **IoC container**: Manages the lifecycle of beans and dependencies
    - **Types**: ApplicationContext (complex), BeanFactory (simpler features - rarely used)
  - **Autowiring**: Process of wiring in dependencies for a Spring Bean



# @Component vs @Bean

Heading	@Component	@Bean
Where?	Can be used on any Java class	Typically used on methods in Spring Configuration classes
Ease of use	Very easy. Just add an annotation.	You write all the code.
Autowiring	Yes - Field, Setter or Constructor Injection	Yes - method call or method parameters
Who creates beans?	Spring Framework	You write bean creation code
Recommended For	Instantiating Beans for Your Own Application Code: @Component	1: Custom Business Logic 2: Instantiating Beans for 3rd-party libraries: @Bean
Beans per class?	One (Singleton) or Many (Prototype)	One or Many - You can create as many as you want

# Why do we have a lot of Dependencies?

- In **Game Runner Hello World App**, we have very few classes
- BUT Real World applications **are much more complex**:
  - Multiple Layers (Web, Business, Data etc)
  - Each layer is **dependent** on the layer below it!
    - Example: Business Layer class talks to a Data Layer class
      - Data Layer class is a **dependency** of Business Layer class
    - There are thousands of such dependencies in every application!
- With Spring Framework:
  - **INSTEAD** of FOCUSING on objects, their dependencies and wiring
    - You can focus on the business logic of your application!
  - **Spring Framework manages the lifecycle** of objects:
    - Mark components using annotations: `@Component` (and others..)
    - Mark dependencies using `@Autowired`
    - Allow Spring Framework to do its magic!
- Ex: `BusinessCalculationService`



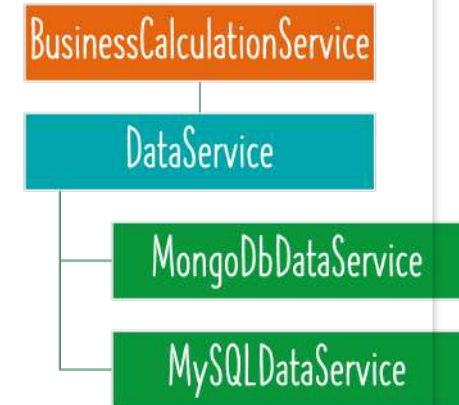
# Exercise - BusinessCalculationService

```
public interface DataService
    int[] retrieveData();

public class MongoDBDataService implements DataService
    public int[] retrieveData()
        return new int[] { 11, 22, 33, 44, 55 };

public class MySQLDataService implements DataService
    public int[] retrieveData()
        return new int[] { 1, 2, 3, 4, 5 };

public class BusinessCalculationService
    public int findMax()
        return Arrays.stream(dataService.retrieveData())
                        .max().orElse(0);
```



- Create **classes and interfaces as needed**
  - Use constructor injection to inject dependencies
  - Make **MongoDbDataService** as primary
  - Create a **Spring Context**
    - Prefer annotations
    - Retrieve **BusinessCalculationService** bean and run **findMax** method

# Exploring Lazy Initialization of Spring Beans



- Default initialization for Spring Beans: **Eager**
- Eager initialization is recommended:
  - Errors in the configuration are discovered immediately at application startup
- However, you can configure beans to be lazily initialized using **Lazy** annotation:
  - NOT recommended (AND) Not frequently used
- **Lazy** annotation:
  - Can be used almost everywhere `@Component` and `@Bean` are used
  - Lazy-resolution proxy will be injected instead of actual dependency
  - Can be used on Configuration (`@Configuration`) class:
    - All `@Bean` methods within the `@Configuration` will be lazily initialized



# Comparing Lazy Initialization vs Eager Initialization

Heading	Lazy Initialization	Eager Initialization
Initialization time	Bean initialized when it is first made use of in the application	Bean initialized at startup of the application
Default	NOT Default	Default
Code Snippet	@Lazy OR @Lazy(value=true)	@Lazy(value=false) OR (Absence of @Lazy)
What happens if there are errors in initializing?	Errors will result in runtime exceptions	Errors will prevent application from starting up
Usage	Rarely used	Very frequently used
Memory Consumption	Less (until bean is initialized)	All beans are initialized at startup
Recommended Scenario	Beans very rarely used in your app	Most of your beans

# Spring Bean Scopes



- Spring Beans are defined to be used in a specific scope:
  - **Singleton** - One object instance per Spring IoC container
  - **Prototype** - Possibly many object instances per Spring IoC container
  - Scopes applicable ONLY for web-aware Spring ApplicationContext
    - **Request** - One object instance per single HTTP request
    - **Session** - One object instance per user HTTP Session
    - **Application** - One object instance per web application runtime
    - **Websocket** - One object instance per WebSocket instance
- **Java Singleton (GOF) vs Spring Singleton**
  - **Spring Singleton**: One object instance per Spring IoC container
  - **Java Singleton (GOF)**: One object instance per JVM

# Prototype vs Singleton Bean Scope

Heading	Prototype	Singleton
Instances	Possibly Many per Spring IOC Container	One per Spring IOC Container
Beans	New bean instance created every time the bean is referred to	Same bean instance reused
Default	NOT Default	Default
Code Snippet	<code>@Scope(value=ConfigurableBeanFactory.SCOPE_PROTOTYPE)</code>	<code>@Scope(value=ConfigurableBeanFactory.SCOPE_SINGLETON)</code> OR Default
Usage	Rarely used	Very frequently used
Recommended Scenario	Stateful beans	Stateless beans

# Evolution of Jakarta EE: vs J2EE vs Java EE

- Enterprise capabilities were **initially built into JDK**
- With time, they were separated out:
  - **J2EE** - Java 2 Platform Enterprise Edition
  - **Java EE** - Java Platform Enterprise Edition (Rebranding)
  - **Jakarta EE** (Oracle gave Java EE rights to the Eclipse Foundation)
    - **Important Specifications:**
      - Jakarta Server Pages (JSP)
      - Jakarta Standard Tag Library (JSTL)
      - Jakarta Enterprise Beans (EJB)
      - Jakarta RESTful Web Services (JAX-RS)
      - Jakarta Bean Validation
      - Jakarta Contexts and Dependency Injection (CDI)
      - Jakarta Persistence (JPA)
    - Supported by **Spring 6 and Spring Boot 3**
      - That's why we use `jakarta.*` packages (*instead of javax.\**)



# Jakarta Contexts & Dependency Injection (CDI)



- Spring Framework V1 was released in 2004
- **CDI specification** introduced into Java EE 6 platform in December 2009
- Now called **Jakarta Contexts and Dependency Injection (CDI)**
- CDI is a **specification (interface)**
  - Spring Framework implements CDI
- **Important Inject API Annotations:**
  - Inject (~Autowired in Spring)
  - Named (~Component in Spring)
  - Qualifier
  - Scope
  - Singleton

# Let's Compare: Annotations vs XML Configuration

Heading	Annotations	XML Configuration
Ease of use	Very Easy (defined close to source - class, method and/or variable)	Cumbersome
Short and concise	Yes	No
Clean POJOs	No. POJOs are polluted with Spring Annotations	Yes. No change in Java code.
Easy to Maintain	Yes	No
Usage Frequency	Almost all recent projects	Rarely
Recommendation	Either of them is fine BUT be consistent	Do NOT mix both
Debugging difficulty	Hard	Medium

# Spring Stereotype Annotations - @Component & more..

- **@Component** - Generic annotation applicable for any class
  - Base for all Spring Stereotype Annotations
  - Specializations of @Component:
    - **@Service** - Indicates that an annotated class has business logic
    - **@Controller** - Indicates that an annotated class is a "Controller" (e.g. a web controller)
      - Used to define controllers in your web applications and REST API
    - **@Repository** - Indicates that an annotated class is used to retrieve and/or manipulate data in a database
- **What should you use?**
  - (MY RECOMMENDATION) Use the most specific annotation possible
  - **Why?**
    - By using a specific annotation, you are giving more information to the framework about your intentions.
    - You can use AOP at a later point to add additional behavior
      - **Example:** For @Repository, Spring automatically wires in JDBC Exception translation features

Web

Business

Data

# Quick Review of Important Spring Annotations

Annotation	Description
<b>@Configuration</b>	Indicates that a class declares one or more @Bean methods and may be processed by the Spring container to generate bean definitions
<b>@ComponentScan</b>	Define specific packages to scan for components. If specific packages are not defined, scanning will occur from the package of the class that declares this annotation
<b>@Bean</b>	Indicates that a method produces a bean to be managed by the Spring container
<b>@Component</b>	Indicates that an annotated class is a "component"
<b>@Service</b>	Specialization of @Component indicating that an annotated class has business logic
<b>@Controller</b>	Specialization of @Component indicating that an annotated class is a "Controller" (e.g. a web controller). Used to define controllers in your web applications and REST API
<b>@Repository</b>	Specialization of @Component indicating that an annotated class is used to retrieve and/or manipulate data in a database



# Quick Review of Important Spring Annotations - 2

Annotation	Description
@Primary	Indicates that a bean should be given preference when multiple candidates are qualified to autowire a single-valued dependency
@Qualifier	Used on a field or parameter as a qualifier for candidate beans when autowiring
@Lazy	Indicates that a bean has to be lazily initialized. Absence of @Lazy annotation will lead to eager initialization.
@Scope (value = ConfigurableBeanFactory.SCOPE_PROTOTYPE)	Defines a bean to be a prototype - a new instance will be created every time you refer to the bean. Default scope is singleton - one instance per IOC container.

# Quick Review of Important Spring Annotations - 3

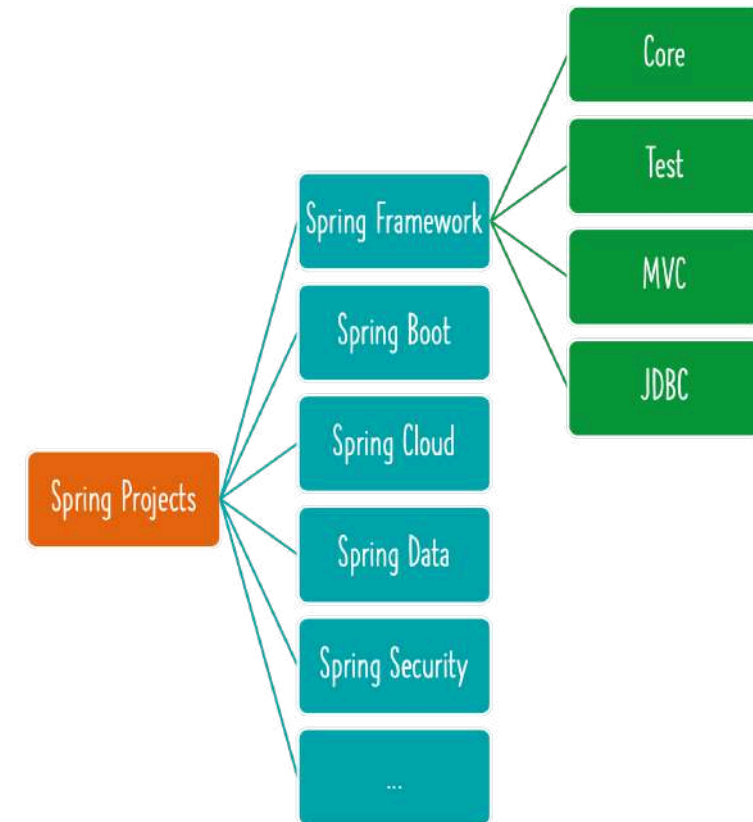
Annotation	Description
<b>@PostConstruct</b>	Identifies the method that will be executed after dependency injection is done to perform any initialization
<b>@PreDestroy</b>	Identifies the method that will receive the callback notification to signal that the instance is in the process of being removed by the container. Typically used to release resources that it has been holding.
<b>@Named</b>	Jakarta Contexts & Dependency Injection (CDI) Annotation similar to Component
<b>@Inject</b>	Jakarta Contexts & Dependency Injection (CDI) Annotation similar to Autowired

# Quick Review of Important Spring Concepts

Concept	Description
Dependency Injection	Identify beans, their dependencies and wire them together (provides <b>IOC</b> - Inversion of Control)
Constr. injection	Dependencies are set by creating the Bean using its Constructor
Setter injection	Dependencies are set by calling setter methods on your beans
Field injection	No setter or constructor. Dependency is injected using reflection.
IOC Container	Spring IOC Context that manages Spring beans & their lifecycle
Bean Factory	Basic Spring IOC Container
Application Context	Advanced Spring IOC Container with enterprise-specific features - Easy to use in web applications with internationalization features and good integration with Spring AOP
Spring Beans	Objects managed by Spring
Auto-wiring	Process of wiring in dependencies for a Spring Bean

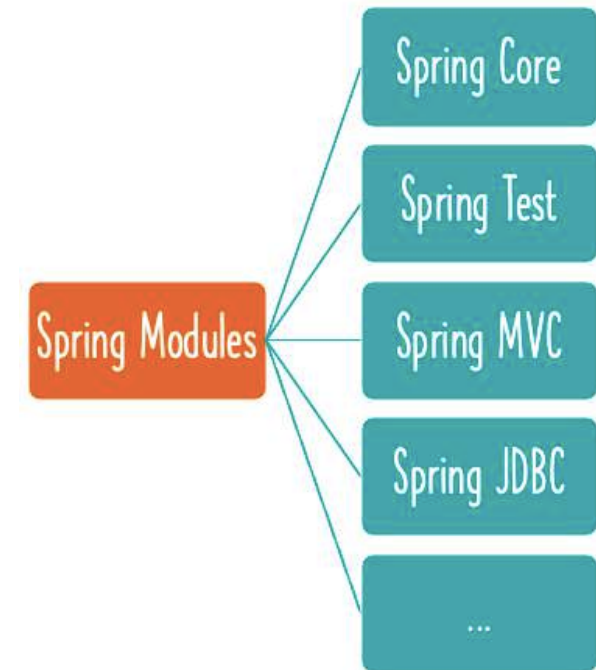
# Spring Big Picture - Framework, Modules and Projects

- **Spring Core** : IOC Container, Dependency Injection, Auto Wiring, ..
  - These are the fundamental building blocks to:
    - Building web applications
    - Creating REST API
    - Implementing authentication and authorization
    - Talking to a database
    - Integrating with other systems
    - Writing great unit tests
- Let's now get a Spring Big Picture:
  - Spring Framework
  - Spring Modules
  - Spring Projects



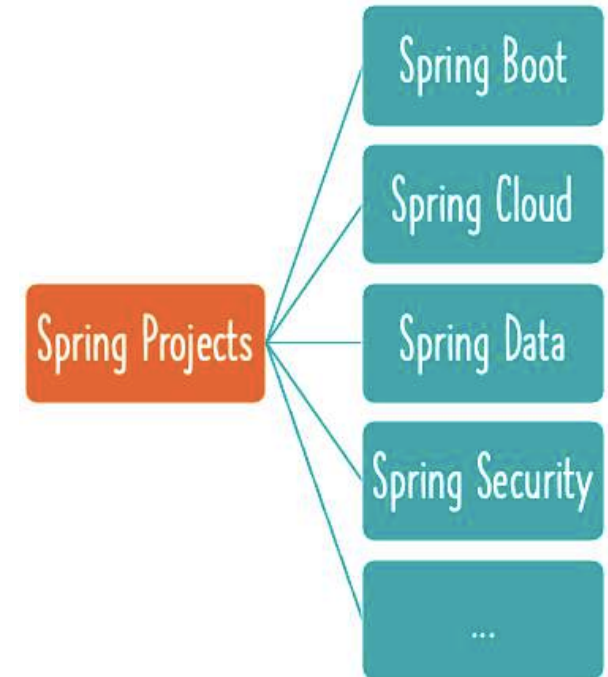
# Spring Big Picture - Framework and Modules

- Spring Framework contains multiple **Spring Modules**:
  - **Fundamental Features:** Core (IOC Container, Dependency Injection, Auto Wiring, ..)
  - **Web:** Spring MVC etc (Web applications, REST API)
  - **Web Reactive:** Spring WebFlux etc
  - **Data Access:** JDBC, JPA etc
  - **Integration:** JMS etc
  - **Testing:** Mock Objects, Spring MVC Test etc
- **No Dumb Question:** Why is Spring Framework divided into Modules?
  - Each application can choose modules they want to make use of
  - They do not need to make use of everything in Spring framework!



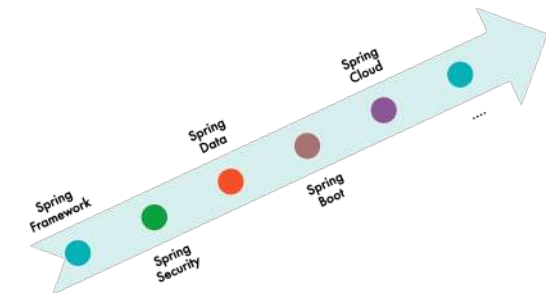
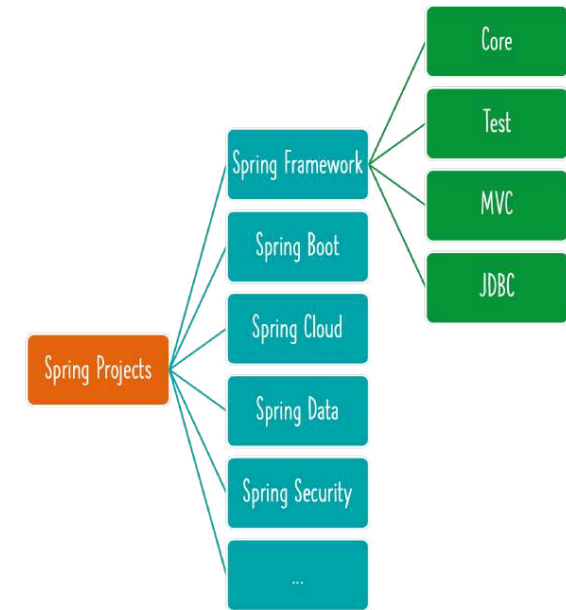
# Spring Big Picture - Spring Projects

- Application architectures evolve continuously
  - Web > REST API > Microservices > Cloud > ...
- Spring evolves through **Spring Projects**:
  - **First Project**: Spring Framework
  - **Spring Security**: Secure your web application or REST API or microservice
  - **Spring Data**: Integrate the same way with different types of databases : NoSQL and Relational
  - **Spring Integration**: Address challenges with integration with other applications
  - **Spring Boot**: Popular framework to build microservices
  - **Spring Cloud**: Build cloud native applications



# Spring Big Picture - Framework, Modules and Projects

- **Hierarchy:** Spring Projects > Spring Framework > Spring Modules
- **Why is Spring Eco system popular?**
  - **Loose Coupling:** Spring manages creation and wiring of beans and dependencies
    - Makes it easy to build loosely coupled applications
    - Make writing unit tests easy! (Spring Unit Testing)
  - **Reduced Boilerplate Code:** Focus on Business Logic
    - Example: No need for exception handling in each method!
      - All Checked Exceptions are converted to Runtime or Unchecked Exceptions
  - **Architectural Flexibility:** Spring Modules and Projects
    - You can pick and choose which ones to use (You DON'T need to use all of them!)
  - **Evolution with Time:** Microservices and Cloud
    - Spring Boot, Spring Cloud etc!



# 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?
  - **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

# 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/rangakaramam/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.jpa
  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/rangakaraman/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.idap
  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

- 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**

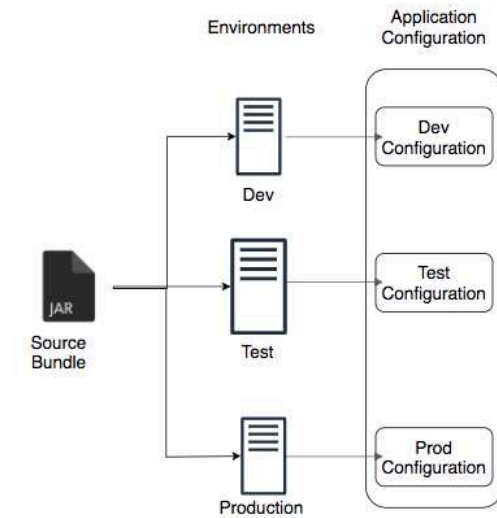


# Spring Boot PRODUCTION-READY



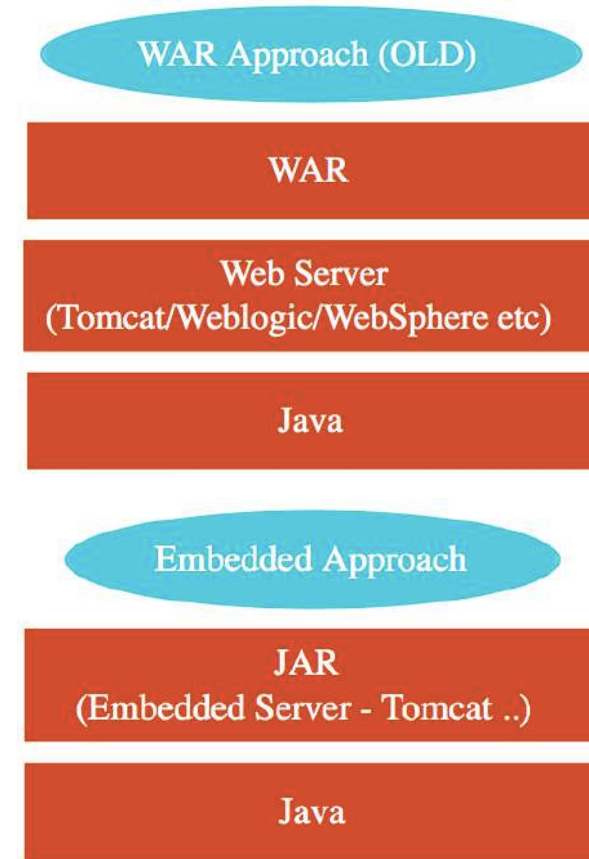
# Managing App. Configuration using Profiles

- 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

- 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



- **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!

# JPA and Hibernate in 10 Steps

# Getting Started with JPA and Hibernate

- Build a Simple JPA App using **Modern Spring Boot Approach**
- Get **Hands-on** with JPA, Hibernate and Spring Boot
  - World before JPA - JDBC, Spring JDBC
  - Why JPA? Why Hibernate? (JPA vs Hibernate)
  - Why Spring Boot and Spring Boot Data JPA?
  - **JPA Terminology: Entity and Mapping**

Spring Data JPA

JPA

Spring JDBC

JDBC

# Learning JPA and Hibernate - Approach

- 01: Create a **Spring Boot Project** with H2
- 02: Create **COURSE** table
- 03: Use **Spring JDBC** to play with COURSE table
- 04: Use **JPA and Hibernate** to play with COURSE table
- 05: Use **Spring Data JPA** to play with COURSE table

Spring Data JPA

JPA

Spring JDBC

JDBC



# Spring Boot Auto Configuration Magic

- We added Data JPA and H2 dependencies:
  - Spring Boot Auto Configuration does some magic:
    - Initialize JPA and Spring Data JPA frameworks
    - Launch an in memory database (H2)
    - Setup connection from App to in-memory database
    - Launch a few scripts at startup (example: `data.sql`, `schema.sql`)
- **Remember** - H2 is in memory database
  - Does NOT persist data
  - Great for learning
  - BUT NOT so great for production



# JDBC to Spring JDBC to JPA to Spring Data JPA

- **JDBC**
  - Write a lot of SQL queries! (*delete from todo where id=?*)
  - And write a lot of Java code
- **Spring JDBC**
  - Write a lot of SQL queries (*delete from todo where id=?*)
  - BUT lesser Java code
- **JPA**
  - Do NOT worry about queries
  - Just Map Entities to Tables!
- **Spring Data JPA**
  - Let's make JPA even more simple!
  - I will take care of everything!

Spring Data JPA

JPA

Spring JDBC

JDBC

# JDBC to Spring JDBC

## JDBC example

```
public void deleteTodo(int id) {  
    PreparedStatement st = null;  
    try {  
        st = db.conn.prepareStatement("delete from todo where id=?");  
        st.setInt(1, id);  
        st.execute();  
    } catch (SQLException e) {  
        logger.fatal("Query Failed : ", e);  
    } finally {  
        if (st != null) {  
            try {st.close();}  
            catch (SQLException e) {}  
        }  
    }  
}
```

## Spring JDBC example

```
public void deleteTodo(int id) {  
    jdbcTemplate.update("delete from todo where id=?", id);  
}
```

## JPA Example

```
@Repository
public class PersonJpaRepository {

    @PersistenceContext
    EntityManager entityManager;

    public Person findById(int id) {
        return entityManager.find(Person.class, id);
    }

    public Person update(Person person) {
        return entityManager.merge(person);
    }

    public Person insert(Person person) {
        return entityManager.merge(person);
    }

    public void deleteById(int id) {.....}
```

## Spring Data JPA Example

```
public interface TodoRepository extends JpaRepository<Todo, Integer>{
```

# Hibernate vs JPA

- **JPA** defines the specification. It is an API.
  - How do you define entities?
  - How do you map attributes?
  - Who manages the entities?
- Hibernate is one of the popular implementations of JPA
- Using Hibernate directly would result in a lock in to Hibernate
  - There are other JPA implementations (Toplink, for example)



# Web Application with Spring Boot

# Building Your First Web Application

- Building Your First Web Application can be complex:
  - Web App concepts (Browser, HTML, CSS, Request, Response, Form, Session, Authentication)
  - Spring MVC (Dispatcher Servlet, View Resolvers, Model, View, Controller, Validations ..)
  - Spring Boot (Starters, Auto Configuration, ..)
  - Frameworks/Tools (JSP, JSTL, JPA, Bootstrap, Spring Security, MySQL, H2)
- **Goal:** Build Todo Management Web App with a **Modern Spring Boot** Approach
  - **AND** explore all concepts in a **HANDS-ON** way

in28Minutes

Home

Todos

Logout

Your Todos

Description	Target Date	Is it Done?		
Learn AWS	10/06/2032	false	<button>Update</button>	<button>Delete</button>
Learn DevOps	10/06/2031	false	<button>Update</button>	<button>Delete</button>
Learn React	10/06/2030	false	<button>Update</button>	<button>Delete</button>
Learn Angular	10/06/2029	false	<button>Update</button>	<button>Delete</button>

# Spring Initializr

- My favorite place on the internet
- Easiest way to create Spring Boot Projects
- Remember:
  - 1: SpringBoot: Use **latest released** version
    - Avoid M1,M2,M3, SNAPSHOT!
  - 2: Java: Use **latest** Version
    - Java uses 6 month release patterns
    - Spring Boot 3.0+ works on Java 17+
  - 3: Use **latest** Eclipse Java EE IDE version



## Project

☒ Maven Project ☐ Gradle Project

## Language

☒ Java ☐ Kotlin ☐ Groovy



# Understanding Logging

```
logging.level.some.path=debug
logging.level.some.other.path=error
logging.file.name=logfile.log

private Logger logger = LoggerFactory.getLogger(this.getClass());
logger.info("postConstruct");
```

- **Knowing what to log** is an essential skill to be a great programmer
- Spring Boot makes logging easy
  - spring-boot-starter-logging
- **Default:** Logback with SLF4j
- Typical Log Levels: ERROR, WARN, INFO, DEBUG, or TRACE

# Session vs Request Scopes

- All requests from browser are handled by our web application deployed on a server
- **Request Scope:** Active for a single request **ONLY**
  - Once the response is sent back, the request attributes will be removed from memory
  - These cannot be used for future requests
  - Recommended for most use cases
- **Session Scope:** Details stored across multiple requests
  - Be careful about what you store in session (Takes additional memory as all details are stored on server)

Please sign in

You have been signed out

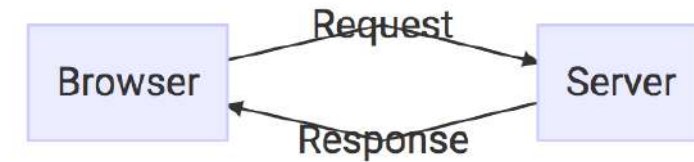
Username

Password

Sign in

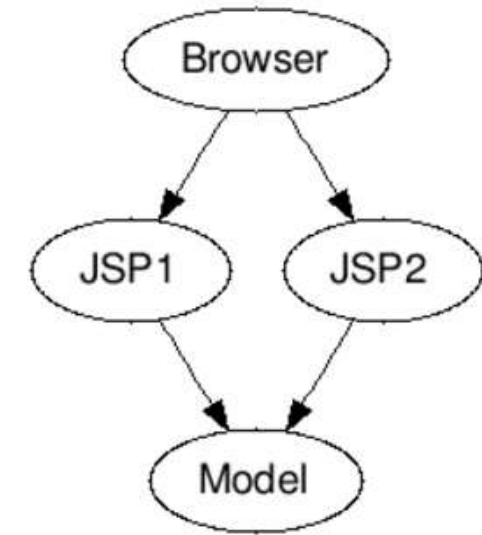
# How does Web work?

- **A:** Browser sends a request
  - HttpRequest
- **B:** Server handles the request
  - Your Spring Boot Web Application
- **C:** Server returns the response
  - HttpResponse



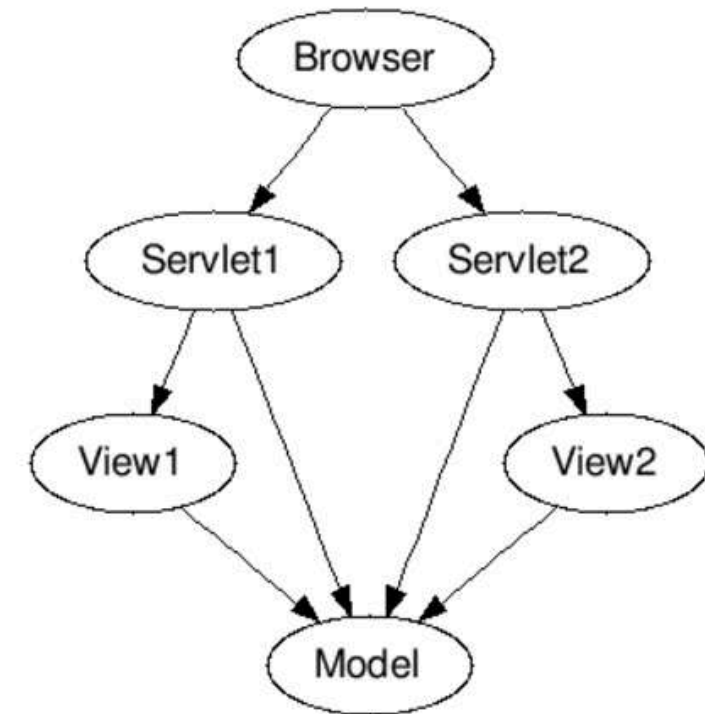
# Peek into History - Model 1 Arch.

- **ALL CODE** in Views (JSPs, ...)
  - View logic
  - Flow logic
  - Queries to databases
- **Disadvantages:**
  - VERY complex JSPs
  - ZERO separation of concerns
  - Difficult to maintain



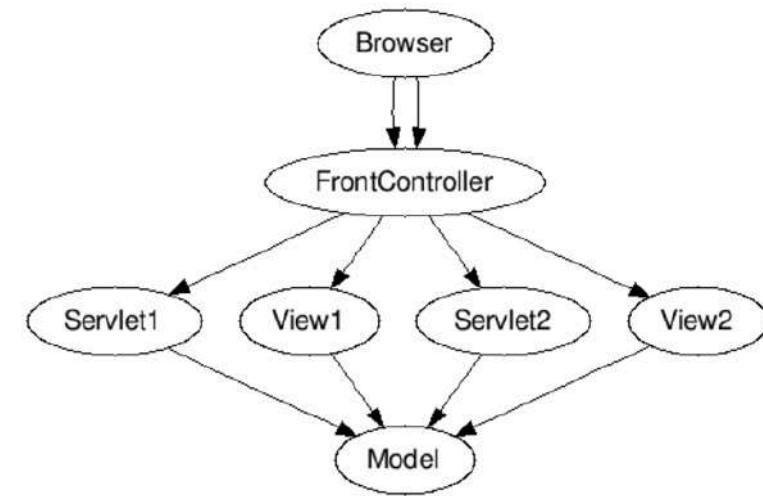
# Peek into History - Model 2 Arch.

- How about separating concerns?
  - **Model:** Data to generate the view
  - **View:** Show information to user
  - **Controller:** Controls the flow
- **Advantage:** Simpler to maintain
- **Concern:**
  - Where to implement common features to all controllers?



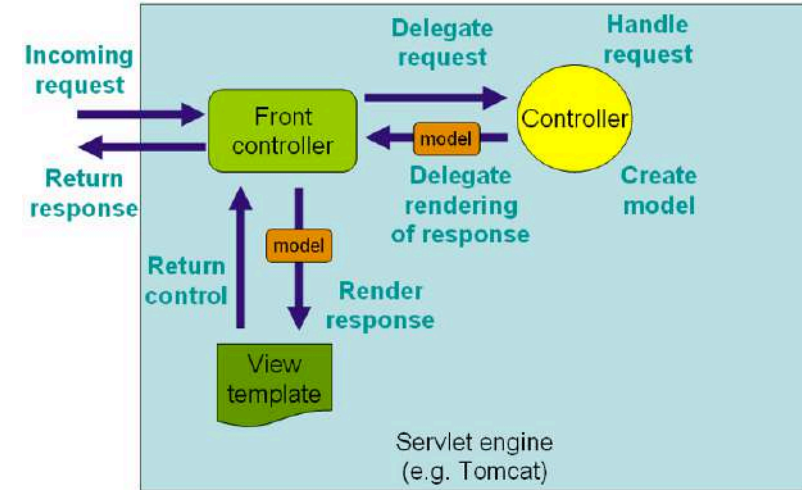
# Model 2 Architecture - Front Controller

- **Concept:** All requests flow into a central controller
  - Called as **Front Controller**
- Front Controller controls flow to Controller's and View's
  - Common features can be implemented in the Front Controller



# Spring MVC Front Controller - Dispatcher Servlet

- **A:** Receives HTTP Request
- **B:** Processes HTTP Request
  - **B1:** Identifies correct Controller method
    - Based on request URL
  - **B2:** Executes Controller method
    - Returns Model and View Name
  - **B3:** Identifies correct View
    - Using ViewResolver
  - **B4:** Executes view
- **C:** Returns HTTP Response



# Validations with Spring Boot

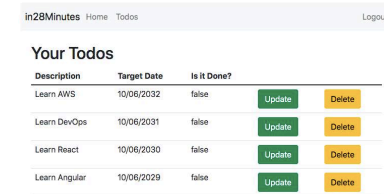
- 1: Spring Boot Starter Validation
  - pom.xml
- 2: Command Bean (Form Backing Object)
  - 2-way binding (todo.jsp & TodoController.java)
- 3: Add Validations to Bean
  - Todo.java
- 4: Display Validation Errors in the View
  - todo.jsp





# Quick Review : Web App with Spring Boot

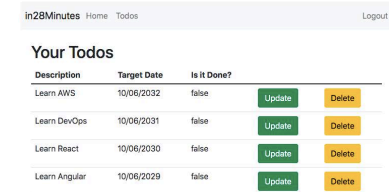
- **HTML:** Hyper Text Markup Language
  - Tags like html, head, body, table, link are part of HTML
- **CSS:** Cascading Style Sheets
  - Styling of your web page is done using CSS
  - We used Bootstrap CSS framework
- **JavaScript:** Do actions on a web page
  - Example: Display a Date Popup (Bootstrap Datepicker)
- **JSTL:** Display dynamic data from model
  - `<c:forEach items="${todos}" var="todo">`
- **Spring form tag library:** Data binding-aware tags for handling form elements
  - `<form:form` `method="post"`



in28Minutes		Home	Todos	Logout
Your Todos				
Description	Target Date	Is It Done?	Update	Delete
Learn AWS	10/06/2032	false	Update	Delete
Learn DevOps	10/06/2031	false	Update	Delete
Learn React	10/06/2030	false	Update	Delete
Learn Angular	10/06/2029	false	Update	Delete

# Quick Review : Web App with Spring Boot

- **DispatcherServlet:** All requests flow into a central controller (Front Controller)
  - **View:** Show information to user
  - **Controller:** Controls the flow
  - **Model:** Data to generate the view
- **Spring Boot Starters:** Fast track building apps
  - Spring Boot Starter Web
  - Spring Boot Starter Validation
  - Spring Boot Starter Security
  - Spring Boot Starter Data JPA



The screenshot shows a web application interface with a navigation bar at the top containing 'in28Minutes', 'Home', 'Todos', and 'Logout'. Below the navigation bar is a section titled 'Your Todos' which contains a table. The table has four columns: 'Description', 'Target Date', 'Is It Done?', and two buttons labeled 'Update' and 'Delete'. There are four rows of data in the table, each representing a todo item.

Description	Target Date	Is It Done?	Update	Delete
Learn AWS	10/06/2032	false	Update	Delete
Learn DevOps	10/06/2031	false	Update	Delete
Learn React	10/06/2030	false	Update	Delete
Learn Angular	10/06/2029	false	Update	Delete

# 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!



```
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

- **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

GET /jpa/users/{id}/posts

Parameters

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

Responses

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 JSON format. On the right, the details for the GET endpoint `/jpa/users/{id}/posts` are displayed.

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

**GET /jpa/users/{id}/posts**

**Parameters**

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

**Responses**

Code	Description
200	OK

Media type: **application/hal+json** (Controls Accept header)

Example Value | Schema

```
[
  {
    "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)

The screenshot displays two instances of a REST client interface, likely Swagger Client or similar, showing the configuration and results of HTTP requests.

**Top Instance (French):**

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

**Bottom Instance (Dutch):**

- 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/v1/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

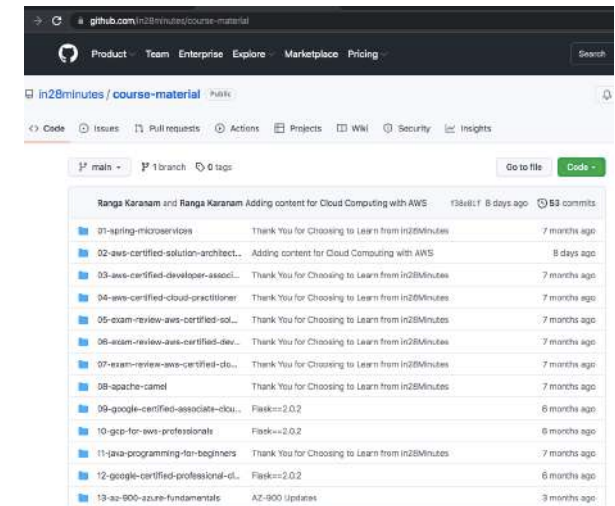
- SAME-URL headers=[X-API-VERSION=1]
- SAME-URL headers=[X-API-VERSION=2]

**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

- **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

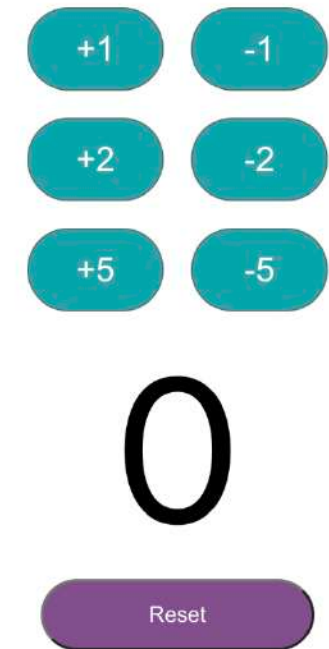
- **1: HAL (JSON Hypertext Application Language)**
  - Simple format that gives a consistent and easy way to hyperlink between resources in your API
- **2: HAL Explorer**
  - An API explorer for RESTful Hypermedia APIs using HAL
  - Enable your non-technical teams to play with APIs
- **3: Spring Boot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



# Full Stack Application with Spring Boot and React

# What will we build?

- Counter Application
  - Understand React Fundamentals
- A Full-Stack Todo Management Application
  - Add Todo
  - Delete Todo
  - Update Todo
  - Authentication (Login/Logout)
  - JWT



in28Minutes Home Todos Logout

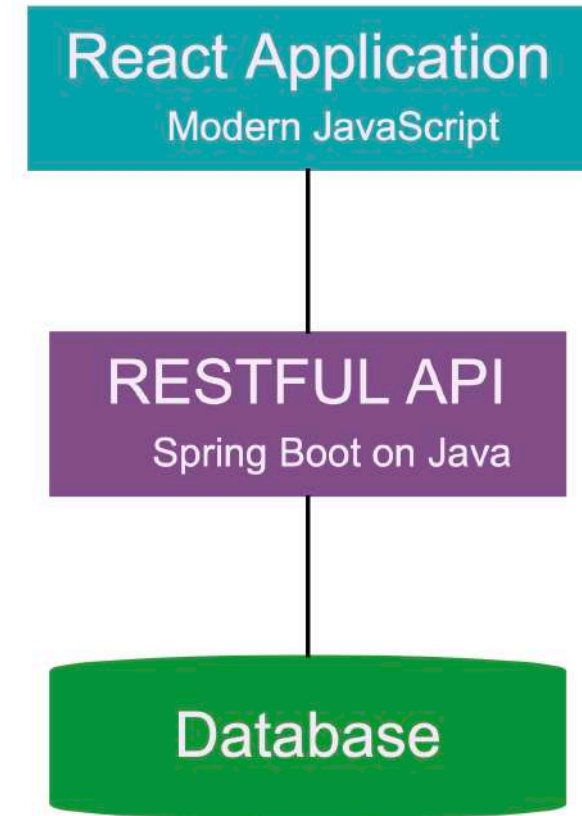
Your Todos

Description	Target Date	Is it Done?		
Learn AWS	10/06/2032	false	<button>Update</button>	<button>Delete</button>
Learn DevOps	10/06/2031	false	<button>Update</button>	<button>Delete</button>
Learn React	10/06/2030	false	<button>Update</button>	<button>Delete</button>
Learn Angular	10/06/2029	false	<button>Update</button>	<button>Delete</button>



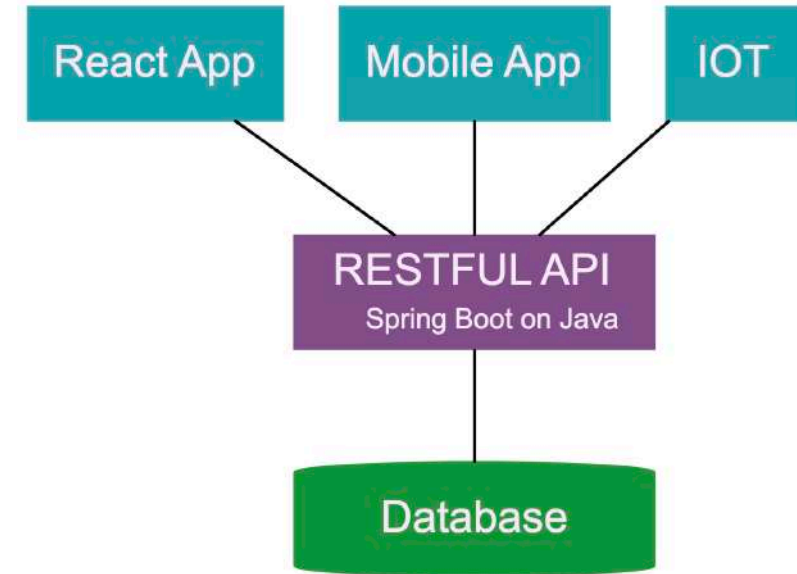
# Full Stack Architecture

- **Front-end: React Framework**
  - Modern JavaScript
- **Backend REST API: Spring Boot**
- **Database**
  - H2 > MySQL
- **Authentication**
  - Spring Security (Basic > JWT)



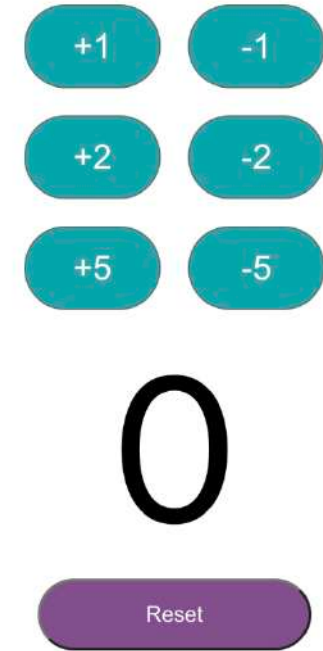
# Why Full-Stack Architecture?

- Full Stack Architectures are **complex** to build
  - You need to understand **different languages**
  - You need to understand a **variety of frameworks**
  - You need to use a **variety of tools**
- **Why Full-Stack?**
  - Because they give you **flexibility** and allow **reuse of REST API**
    - **OPTION:** Create a Mobile App talking to REST API
    - **OPTION:** Create an IOT App talking to REST API



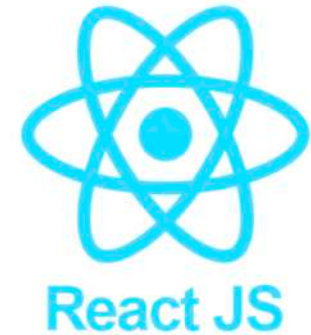
# Quick Look into JavaScript History

- **JavaScript** evolved considerably in the last decade or so
  - (EARLIER JS Versions) Very difficult to write maintainable JavaScript code
  - **Improved drastically** in the last decade
    - JAVASCRIPT VERSIONS
      - ES5 - 2009
      - ES6 - 2015 - ES2015
      - ES7 - 2016 - ES2016
      - ...
      - ES13 - 2022 - ES2022
      - ES14 - 2023 - ES2023
      - ...
  - **ES: ECMAScript**
    - EcmaScript is **standard**
    - JavaScript is **implementation**
- **GOOD NEWS:** Writing Good JavaScript code is not so difficult :)
  - Do NOT worry if you are new to JavaScript



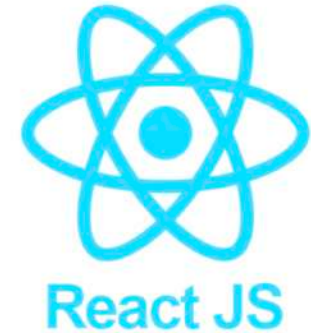
# What is React?

- **React:** One of the most popular JavaScript libraries to build SPA (Single Page Applications)
  - Popular alternatives: Angular, VueJS
- **Open-source project** created by Facebook
- **Component-Based**
- **Mostly** used to build front-end web SPA applications
  - Can also be used to create native apps for Android, iOS (React Native)



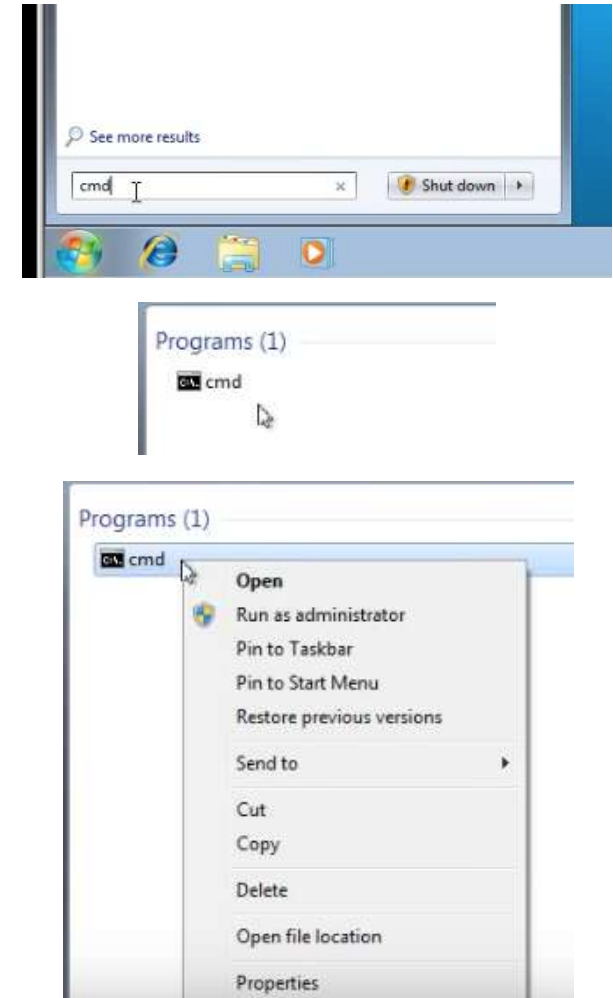
# Creating React App with Create React App

- **Create React App:** Recommended way to create a new single-page application (SPA) using React
  - Compatible with **macOS, Windows, and Linux**
  - **Prerequisite:** Latest version of Node JS
    - **NPM - package manager:** Install, delete, and update JS packages (npm --version)
    - **NPX - package executor:** Execute JS packages directly, without installing
  - **Let's get started:**
    - **DO NOT WORRY:** Troubleshooting instructions at the end of the video!
    - `cd YOUR_FOLDER`
    - `npx create-react-app todo-app`
    - `cd todo-app`
    - `npm start`



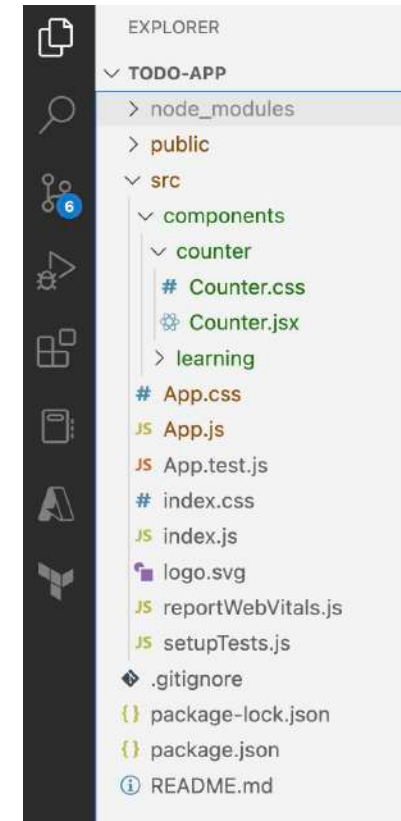
# Troubleshooting

- **Windows:** Launch command prompt as administrator
- **Mac or Linux:** Use sudo
  - `sudo npx create-react-app todo-app`
- **Other things you can try:**
  - `npm uninstall -g create-react-app`
  - `npx clear-npx-cache`
- **Complete troubleshooting guide:**
  - Google for "create react app troubleshooting"



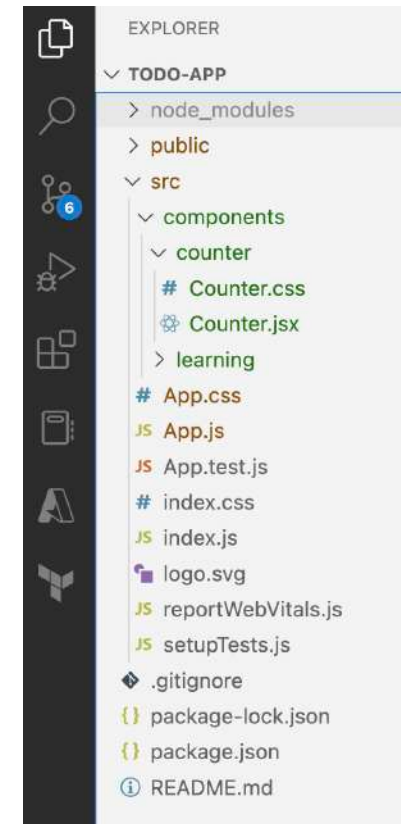
# Important Commands

- **npm start:** Runs the app in development mode
  - Recommendation: Use Google Chrome
- **npm test:** Run unit tests
- **npm run build:** Build a production deployable unit
  - Minified
  - Optimized for performance
- **npm install --save react-router-dom:** Add a dependency to your project



# Visual Studio Code - Tips

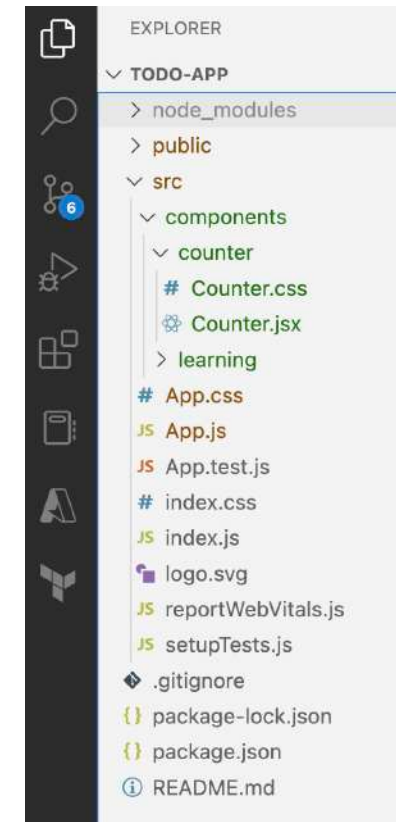
- **Toggle Explorer**
  - Ctrl + B or Cmd + B
- **Explore Left Hand Side Bar**
  - Search etc
- **Make a change to index.html**
  - Change Title
- **Make a change to App.js**
  - Remove everything in App div
  - Add My Todo Application
- **How is the magic happening?**
  - Create React App
  - Automatically builds and renders in the browser





# Exploring Create React App Folder Structure

- **Goal:** Get a **10,000 feet** overview of folder structure
  - **README.md:** Documentation
  - **package.json:** Define dependencies (similar to Maven pom.xml)
  - **node\_modules:** Folder where all the dependencies are downloaded to
  - **React Initialization**
    - **public/index.html:** Contains root div
    - **src/index.js:** Initializes React App. Loads App component.
      - **src/index.css** - Styling for entire application
    - **src/App.js:** Code for App component
      - **src/App.css** - Styling for App component
      - **src/App.test.js** - Unit tests for App component
        - Unit test is right along side production code (Different to Java approach)
  - **Remember:** Syntax might look little complex
    - Different from typical Java code (imports, ...)
    - We will focus on it a little later



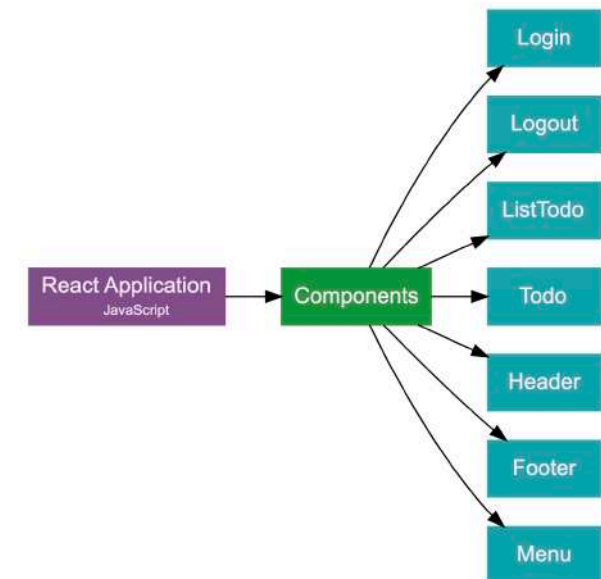
# Why do we need React Components?

- **Web applications** have complex structure
  - Menu, Header, Footer, Welcome Page, Login Page, Logout Page, Todo Page etc
- **Components** help you modularize React apps
  - **Create separate components** for each page element
    - Menu Component
    - Header Component
    - Footer Component
    - ..
  - **Why?**
    - Modularization
    - Reuse

in28Minutes Home Todos Logout

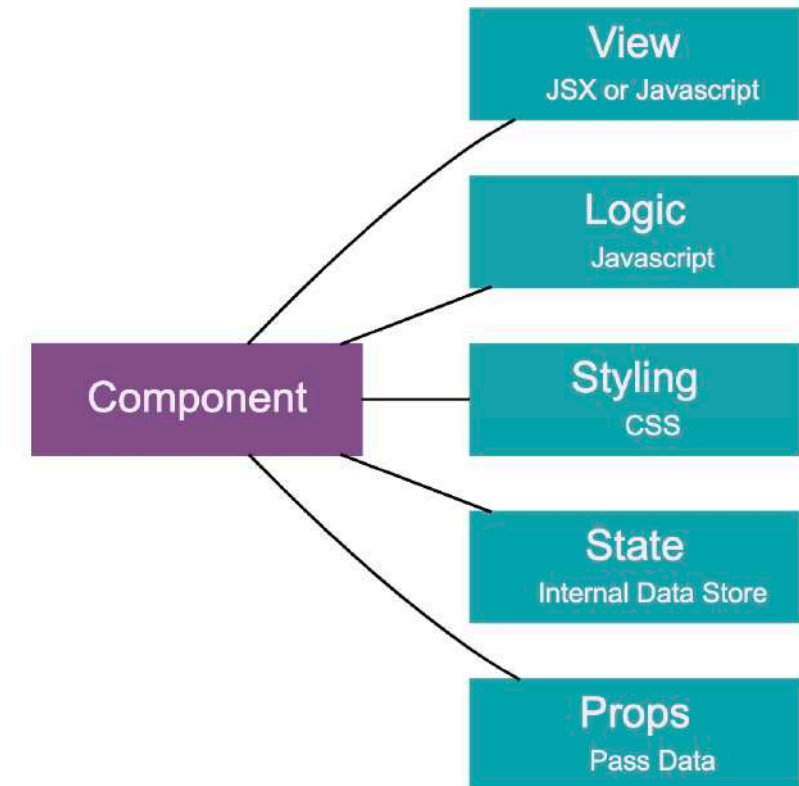
**Your Todos**

Description	Target Date	Is it Done?		
Learn AWS	10/06/2032	false	<button>Update</button>	<button>Delete</button>
Learn DevOps	10/06/2031	false	<button>Update</button>	<button>Delete</button>
Learn React	10/06/2030	false	<button>Update</button>	<button>Delete</button>
Learn Angular	10/06/2029	false	<button>Update</button>	<button>Delete</button>



# Understanding React Components

- **First component** typically loaded in React Apps: **App Component**
- **Parts of a Component**
  - View (JSX or JavaScript)
  - Logic (JavaScript)
  - Styling (CSS)
  - State (Internal Data Store)
  - Props (Pass Data)
- (Remember) React component names must always start with a capital letter

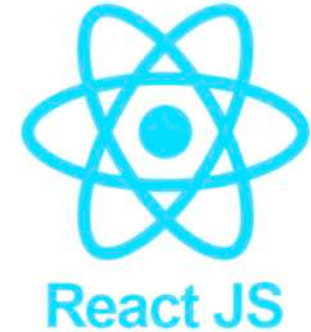


# Creating a React Component

```
function FirstComponent() {  
  return (  
    <div className="FirstComponent">FirstComponent</div>  
  );  
}  
  
class ThirdComponent extends Component {  
  render() {  
    return (  
      <div className="ThirdComponent">ThirdComponent</div>  
    );  
  }  
}
```

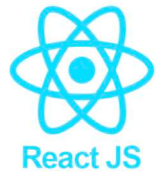
- For now, we will keep things simple:
  - We will write all code in one module
    - First Component
    - Exercise: Second Component
    - Third Component as Class Component
    - Exercise: Fourth Component

# Getting Started with JSX - Views with React



- React projects use **JSX** for presentation
- **Stricter than HTML**
  - Close tags are mandatory
  - Only one top-level tag allowed:
    - Cannot return multiple top-level JSX tags
    - Wrap into a shared parent
      - `<div>...</div>` or `<>...</>` (empty wrapper)
- **How is JSX enabled in a React project?**
  - Different browsers have different support levels modern JS features (ES2015,...,ES2022,...)
    - How to ensure backward compatibility for your JS code?
    - Solution: Babel
    - Babel also converts JSX to JS

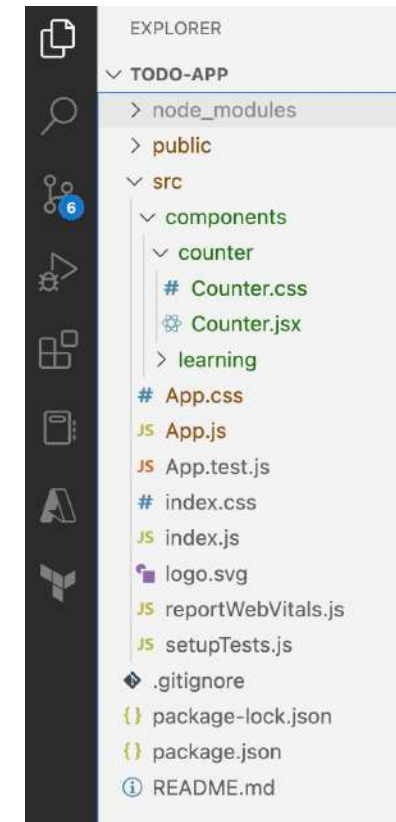
# Let's Play with Babel and JSX



- Let's try Babel: <https://babeljs.io/repl>
  - How does JSX get converted to JS?
    - Example 1: `<h1 className="something" attr="10">heading</h1>`
    - Example 2: `<Parent attr="1"><Child><AnotherChild></AnotherChild></Child></Parent>`
  - Following are examples of ERRORS
    - `<h1></h1><h2></h2>`
      - SOLUTION: wrap with `<div>...</div>` or `<>...</>` (empty wrapper)
    - Close tags are mandatory
- Let's try JSX in our components
  - Parentheses ( ) make returning complex JSX values easier
  - Custom Components should start with upper case letter
    - For HTML you should use small case
  - Specify CSS class - `className`
    - Similar to HTML `class` attribute

# Let's follow JavaScript Best Practices

- **1: Each component in its own file (or module)**
  - `src\components\learning-examples\FirstComponent.jsx`
  - Exercise: Move SecondComponent, ThirdComponent & FourthComponent to their own modules
  - To use a class from a different module, you need to import it
    - **Default import**
      - `import FirstComponent from './components/learning/FirstComponent.jsx';`
    - **Named import**
      - `import { FifthComponent } from './components/learning/FirstComponent.jsx';`
- **2: Exercise: Create LearningComponent and move all code in App component to it!**



# Quick JavaScript Tour For Java Developers

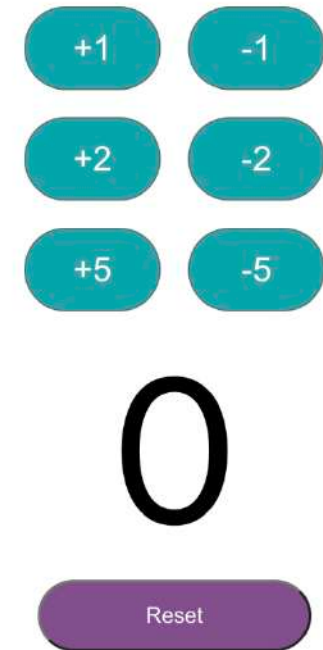
```
const person = {  
  name: 'Ranga Karanam',  
  address: {  
    line1: 'Baker Street',  
    city: 'London',  
    country: 'UK',  
  },  
  profiles: ['twitter', 'linkedin', 'instagram'],  
  printProfile: () => {  
    person.profiles.map(  
      profile => console.log(profile)  
    )  
  }  
}
```

- No need to use semicolon!
- Dynamic objects
- You can store a function in an object!



# Digging Deeper into Components - Counter

- **Parts of a Component**
  - View (JSX or JavaScript)
  - Logic (JavaScript)
  - Styling (CSS)
  - State (Internal Data Store)
  - Props (Pass Data)
- Let's learn more about each of these building another simple example
  - A Counter App
- Let's take a **hands-on step by step** approach



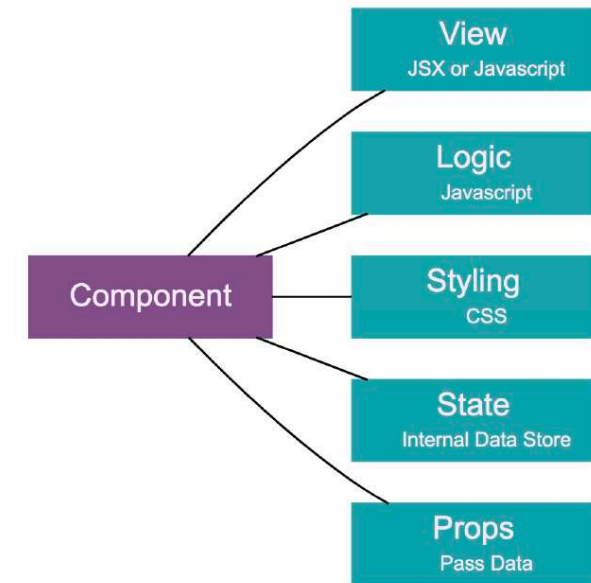
# Define CSS in JSX

```
const customStyle = {  
  backgroundColor: "green",  
  fontSize: "16px",  
  padding: "15px 30px",  
  color: "white",  
  width: "100px",  
  border: "1px solid #666666",  
  borderRadius: "30px",  
};  
  
<button style={customStyle}>+1</button>  
  
<button className="cssClass">+1</button>
```

- Options of styling your React components
  - 1: style
    - Error: `<button style={border-radius:30px}>`
    - Correct Syntax: `<button style={{borderRadius:"30px"}}>`
  - 2: className
    - Define the cssClass in your component CSS file

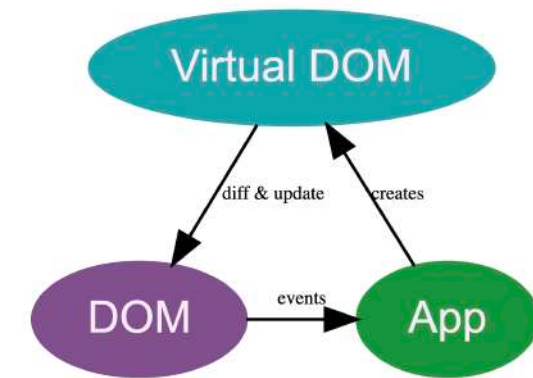
# Understanding State in React

- **State:** Built-in React object used to contain data or information about the component
- **(REMEMBER)** In earlier versions of React, ONLY Class Components can have state
  - AND implementing state was very complex!
- **Hooks were introduced in React 16.8**
  - Hooks are very easy to use
  - **useState** hook allows adding state to Function Components
    - **useState** returns two things
      - **1:** Current state
      - **2:** A function to update state
  - Each instance of component has it's own state
  - How to share state between components?
    - Move state “upwards” (to a parent component)



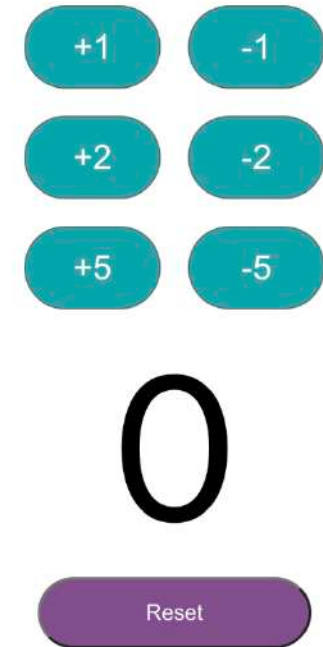
# What's happening in the background with React?

- We updated the state => React updated the view
  - **How can you update an HTML element?**
    - A HTML page is represented by DOM (Document Object Model)
    - Each element in a HTML page is a node in the DOM
    - You need to update the DOM to update the element
    - HOWEVER, writing code to update the DOM can be complex and slow!
  - **React takes a different approach:**
    - **Virtual DOM:** “virtual” representation of a UI (kept in memory)
      - React code updates Virtual DOM
    - React **identifies changes** and **synchronizes them** to HTML page
      - **1:** React creates Virtual DOM v1 on load of page
      - **2:** You perform an action
        - **3:** React creates Virtual DOM v2 as a result of your action
        - **4:** React performs a diff between v1 and v2
        - **5:** React synchronizes changes (updates HTML page)
- **Summary:** We are NOT updating the DOM directly!
  - React identifies changes and **efficiently** updates the DOM



# Enhancing Counter Example

- **1:** Let's create multiple counter buttons
- **2:** Let's have a different increment value for each button
- **3:** Let's have common state for all our buttons



# Exploring React props

```
<PlayingWithProps prop1="value1" prop2="value2" />

function PlayingWithProps({ prop1, prop2 }) {
  return (<{prop1} {prop2}</>)
}

function CounterButton({ incrementBy }) {...}

<CounterButton incrementBy={2}>

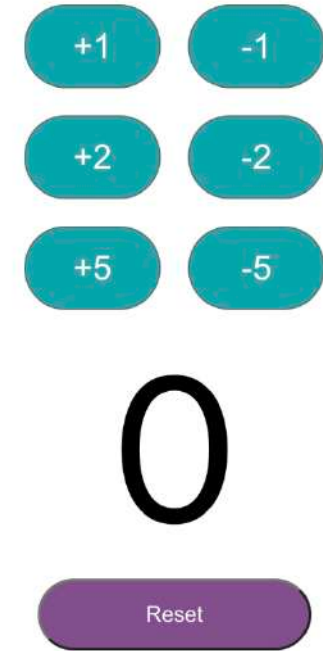
Counter.defaultProps = {
  incrementBy: 1,
};

Counter.propTypes = {
  incrementBy: PropTypes.number,
};
```

- You can pass “props” (short for properties) object to a React Component
- Used for things that remain a constant during lifetime of a component
  - Example increment value of a specific component

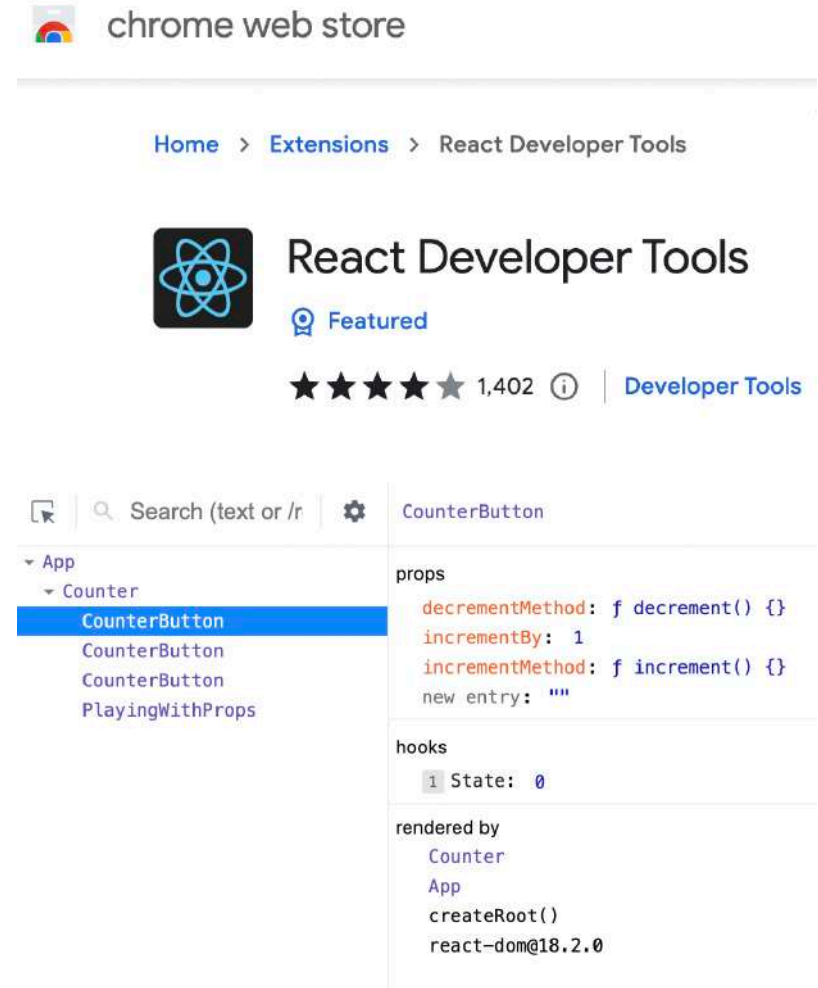
# Moving State Up and More...

- How can we have **one state** for all counters?
  - 1: Rename Counter to CounterButton
  - 2: Calling a parent component method
    - `<CounterButton incrementMethod={increment}>`
  - 3: Exercise: CounterButton as separate module
  - 4: Exercise: Adding Reset Button
  - 5: Remove State From Child
  - 6: Directly Call Parent Methods



# React Developer Tools - Chrome Extension

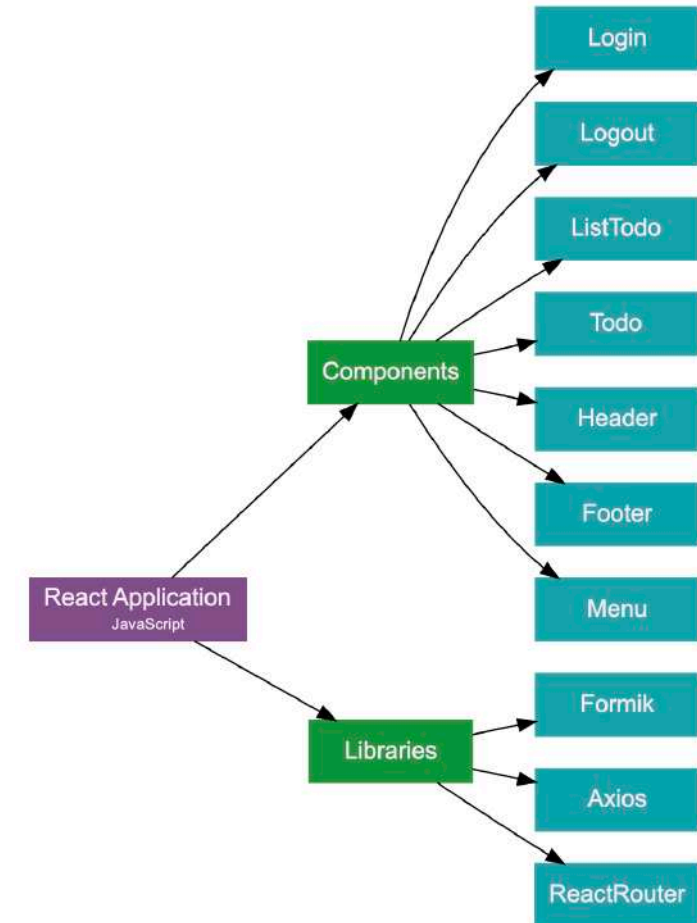
- Chrome Developer Tools extension for React
- **Goal:** Inspect React Component Hierarchies
- Components tab shows:
  - Root React components
  - Sub components that were rendered
- For each component, you can see and edit
  - props
  - state
- **Useful for:**
  - Understanding and Learning React
  - Debugging problems





# Todo Management React App - First Steps

- **1: Counter example** - What did we learn?
  - Basics of Components
    - View (JSX)
    - Styling (CSS)
    - State
    - Props
- **2: Todo Management App** - What will we learn?
  - Routing
  - Forms
  - Validation
  - REST API calls
  - Authentication
  - & a lot more...



# Getting Started with Todo App - Components

- Starting with your TodoApp
  - 1: LoginComponent
    - Make LoginComponent Controlled
      - Link form fields with state
    - Implement Hard-coded Authentication
    - Implement Conditional Rendering
  - 2: WelcomeComponent
    - Implement Routing
  - 3: ErrorComponent
  - 4: ListTodosComponent
  - 5: Add Bootstrap & style our pages
  - 6: HeaderComponent
  - 7: FooterComponent
  - 8: LogoutComponent

Name :  Password :

---

Welcome in28Minutes!! [Click here](#) to manage your todo's.

---

in28Minutes Home Todos Logout

---

© 2018 in28minutes. All rights reserved.

---

Description

Target Date

Your todos are

Description	Target Date	Is it Done?		
Default Desc	06/11/2018	false	<input type="button" value="Update"/>	<input type="button" value="Delete"/>
Learn Angular JS	05/11/2018	false	<input type="button" value="Update"/>	<input type="button" value="Delete"/>
Learn to Dance	05/11/2018	false	<input type="button" value="Update"/>	<input type="button" value="Delete"/>

[Add a Todo](#)

# Full Stack - Todo REST API - Resources and Methods

- REST API

- Hello World REST API:

- Hello World:
      - @GetMapping(path = "/hello-world")
    - Hello World Bean:
      - @GetMapping(path = "/hello-world-bean")
    - Hello World Path Variable:
      - @GetMapping(path = "/hello-world/path-variable/{name}")

- Todo REST API:

- Retrieve Todos
      - @GetMapping("/users/{username}/todos")
    - Retrieve Todo
      - @GetMapping("/users/{username}/todos/{id}")
    - Delete Todo
      - @DeleteMapping("/users/{username}/todos/{id}")
    - Update Todo
      - @PutMapping("/users/{username}/todos/{id}")
    - Create Todo
      - @PostMapping("/users/{username}/todos")

in28Minutes Home Todos

Logout

## Your Todos

Description	Target Date	Is it Done?		
Learn AWS	10/06/2032	false	<button>Update</button>	<button>Delete</button>
Learn DevOps	10/06/2031	false	<button>Update</button>	<button>Delete</button>
Learn React	10/06/2030	false	<button>Update</button>	<button>Delete</button>
Learn Angular	10/06/2029	false	<button>Update</button>	<button>Delete</button>

# Getting Started with JWT

- **Basic Authentication**
  - No Expiration Time
  - No User Details
  - Easily Decoded
- How about a **custom token system**?
  - Custom Structure
  - Possible Security Flaws
  - Service Provider & Service Consumer should understand
- **JWT (Json Web Token)**
  - Open, industry standard for representing claims securely between two parties
  - Can Contain User Details and Authorizations

```
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJzdWIiOiIxMjM0NTY3ODkwIiwibmFtZSI6IkpvaG4gRG9lIiwiaWF0IjoxNTE2MjM5MDIyfQ.SflKxwRJSMeKKF2QT4fwpMeJf36P0k6yJV_adQssw5c
```

# What does a JWT contain?

- **Header**
  - Type: JWT
  - Hashing Algorithm: HS512
- **Payload**
  - **Standard Attributes**
    - **iss**: The issuer
    - **sub**: The subject
    - **aud**: The audience
    - **exp**: When does token expire?
    - **iat**: When was token issued?
  - **Custom Attributes**
    - **youratt1**: Your custom attribute 1
- **Signature**
  - Includes a Secret

HEADER: ALGORITHM & TOKEN TYPE
<pre>{   "alg": "HS256",   "typ": "JWT" }</pre>
PAYLOAD: DATA
<pre>{   "sub": "1234567890",   "name": "John Doe",   "iat": 1516239022 }</pre>
VERIFY SIGNATURE
<pre>HMACSHA256(   base64UrlEncode(header) + "." +   base64UrlEncode(payload),   your-256-bit-secret ) <input type="checkbox"/> secret base64 encoded</pre>

# JWT Flow

```
Request
{
  "username": "in28minutes",
  "password": "dummy"
}

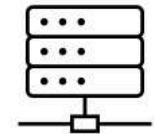
Response
{
  "token": "TOKEN_VALUE"
}
```

- Copy code for now
  - We will understand the code in the Spring Security section
- Send POST to ***http://localhost:8080/authenticate***
  - Get the TOKEN\_VALUE from response
- Use token in authorization header for future API calls:
  - Authorization : "Bearer TOKEN\_VALUE"

# Spring Security

# Understanding Security Fundamentals

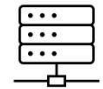
- In any system:
  - You have **resources**
    - A REST API, A Web Application, A Database, A resource in the cloud, ...
  - You have **identities**
    - Identities need to access to resources and perform actions
      - For example: Execute a REST API call, Read/modify data in a database
  - Key Questions:
    - How to **identify users**?
    - How to configure resources they can access & actions that are allowed?
- **Authentication** (is it the right user?)
  - UserId/password (What do you remember?)
  - Biometrics (What do you possess?)
- **Authorization** (do they have the right access?)
  - User XYZ can only read data
  - User ABC can read and update data





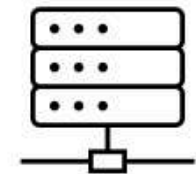
# Understanding Important Security Principles

- A chain is **only as strong** as its WEAKEST link
  - Small security flaw makes an app with robust architecture vulnerable
- **6 Principles Of Building Secure Systems**
  - **1: Trust Nothing**
    - Validate every request
    - Validate piece of data or information that comes into the system
  - **2: Assign Least Privileges**
    - Start the design of the system with security requirements in mind
    - Have a clear picture of the user roles and accesses
    - Assign Minimum Possible Privileges at all levels
      - Application
      - Infrastructure (database + server + ..)
  - **3: Have Complete Mediation**
    - How were Medieval Fort's protected?
      - Everyone had to pass through one main gate
    - Apply a well-implemented security filter. Test the role and access of each user.



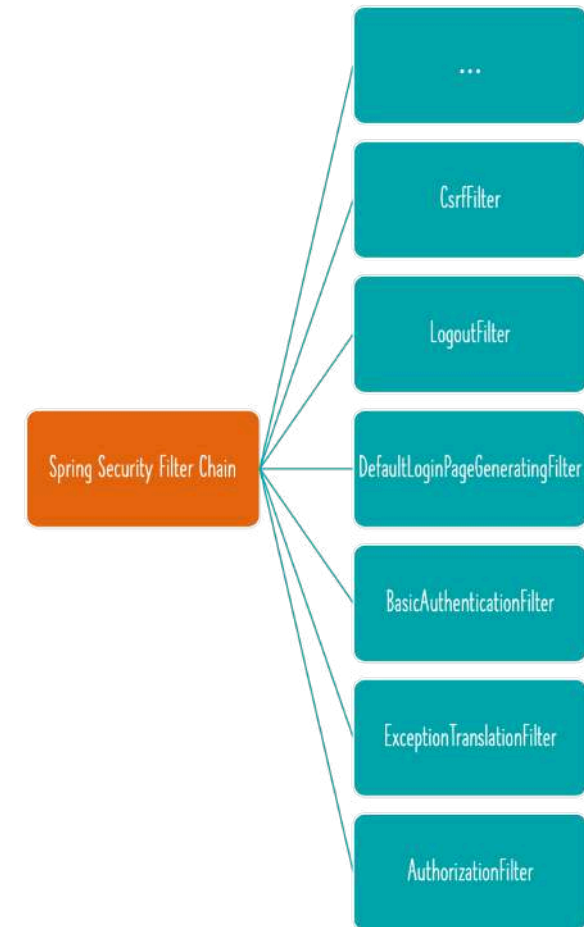
# Understanding Important Security Principles

- **4: Have Defense In Depth**
  - Multiple levels of security
    - Transport, Network, Infrastructure
    - Operating System, Application, ..
- **5: Have Economy Of Mechanism**
  - Security architecture should be simple
  - Simple systems are easier to protect
- **6: Ensure Openness Of Design**
  - Easier to identify and fix security flaws
  - Opposite of the misplaced idea of "Security Through Obscurity"



# Getting Started with Spring Security

- Security is the **NO 1 priority** for enterprises today!
- What is the **most popular security project** in the Spring eco-system?
  - **Spring Security:** Protect your web applications, REST API and microservices
  - Spring Security **can be difficult** to get started
    - Filter Chain
    - Authentication managers
    - Authentication providers
    - ...
  - **BUT** it provides a very flexible security system!
    - By default, everything is protected!
    - A chain of filters ensure proper authentication and authorization

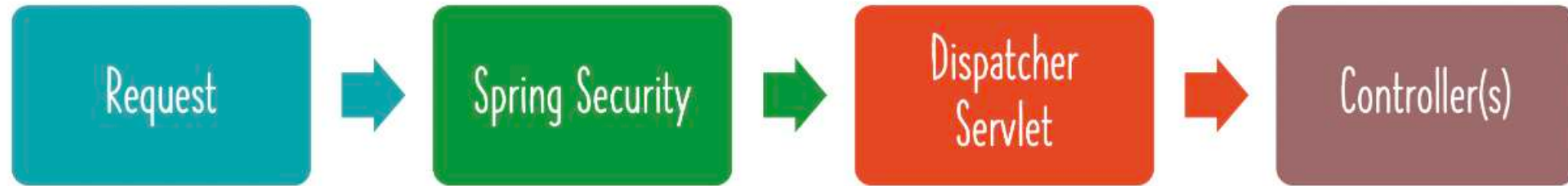


# How does Spring MVC Work?



- DispatcherServlet acts as the front controller
  - Intercepts all requests
  - Routes to the Right Controller

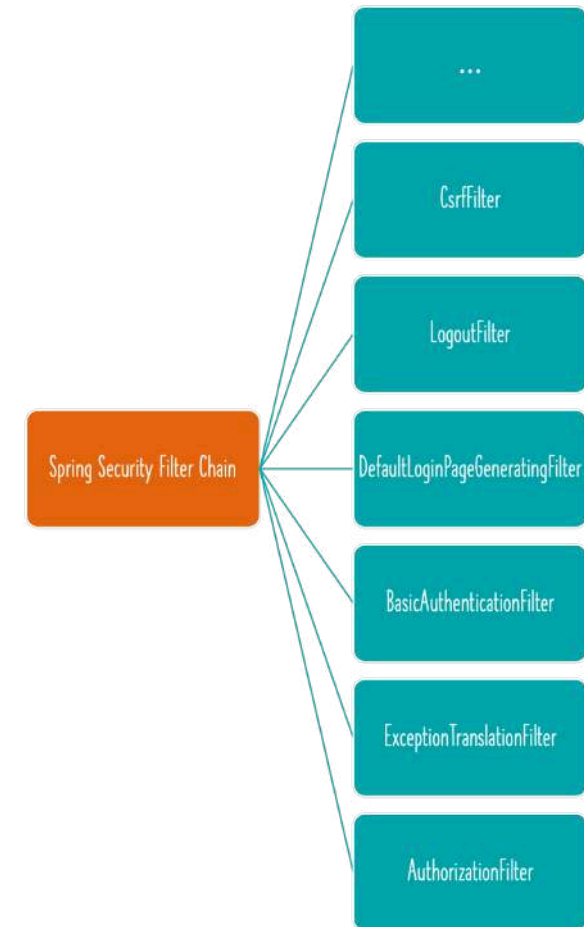
# How does Spring Security Work?




- Spring security intercepts all requests
- Follows following security principle
  - 3: Have Complete Mediation
- Spring security executes a series of filters
  - Also called Spring Security Filter Chain

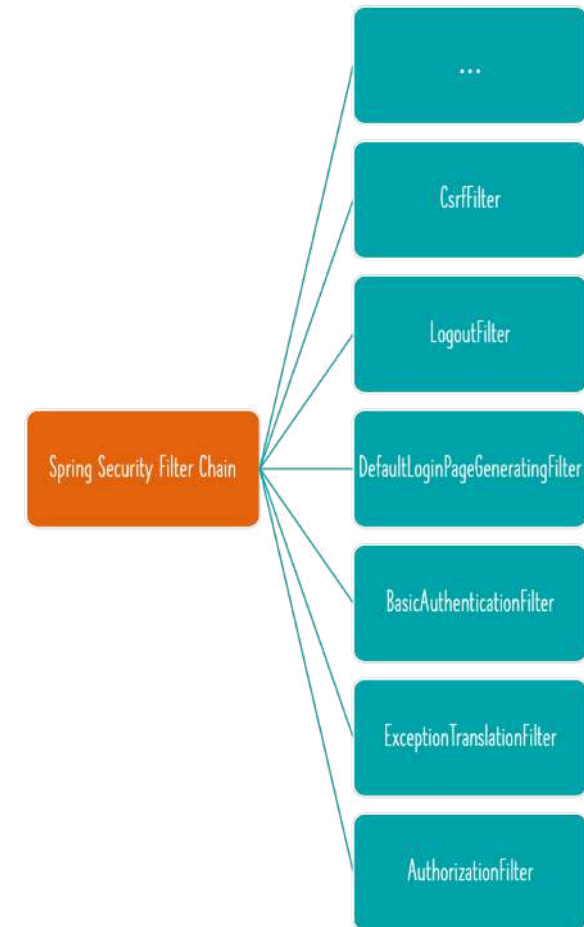
# How does Spring Security Work? (2)

- Spring Security executes a series of filters
  - Filters **provide these features**:
    - **Authentication**: Is it a valid user? (Ex: BasicAuthenticationFilter)
    - **Authorization**: Does the user have right access?(Ex: AuthorizationFilter)
    - **Other Features**:
      - Cross-Origin Resource Sharing (CORS) - CorsFilter
        - Should you allow AJAX calls from other domains?
      - Cross Site Request Forgery (CSRF) - CsrfFilter
        - A malicious website making use of previous authentication on your website
        - Default: CSRF protection enabled for update requests - POST, PUT etc..
      - Login Page, Logout Page
        - LogoutFilter, DefaultLoginPageGeneratingFilter, DefaultLogoutPageGeneratingFilter
      - Translating exceptions into proper Http Responses (ExceptionTranslationFilter)
  - **Order of filters** is important (typical order shown below)
    - **1**: Basic Check Filters - CORS, CSRF, ..
    - **2**: Authentication Filters
    - **3**: Authorization Filters



# Default Spring Security Configuration

- **Everything** is authenticated 
  - You can customize it further
- **Form authentication** is enabled (with default form and logout features)
- **Basic authentication** is enabled
- **Test user** is created
  - Credentials printed in log (Username is **user**)
- **CSRF protection** is enabled
- **CORS requests** are denied
- **X-Frame-Options** is set to 0 (Frames are disabled)
- And a lot of others...



# Exploring Form Based Authentication

- Used by most web applications
- Uses a **Session Cookie**
  - JSESSIONID: E2E693A57F6F7E4AC112A1BF4D40890A
- Spring security enables form based authentication **by default**
- Provides a default **Login Page**
- Provides a default **Logout Page**
- Provides a `/logout` URL
- You can add a change password page
  - `(http.passwordManagement(Customizer.withDefaults()))`

Please sign in

You have been signed out

in28minutes

.....

Sign in



# Exploring Basic Authentication

- **Most basic option** for Securing REST API
  - BUT has many flaws
  - NOT recommended for production use
- Base 64 encoded username and password is sent as **request header**
  - Authorization: Basic aW4yOG1pbnV0ZXM6ZHVtbXk=
  - (DISADVANTAGE) Easy Decoding
- **Basic Auth Authorization Header:**
  - Does NOT contain authorization information (user access, roles,...)
  - Does NOT have Expiry Date

The screenshot shows a REST client interface. At the top, the METHOD is set to GET and the URL is http://localhost:8080/users/in28minutes/todos. Below the URL bar, there is a section for HEADERS. The 'Authorization' header is checked and its value is 'Basic aW4yOG1pbnV0ZXM6ZHVtbXk='. There are buttons for '+ Add header' and 'Add authorization'. On the right side, there is a 'Send' button and a 'Form' dropdown menu. The 'BODY' tab is visible on the far right.

# Getting started with Cross-Site Request Forgery (CSRF)

- **1:** You are logged-in to your bank website
  - A cookie `Cookie-A` is saved in the your web browser
- **2:** You go to a malicious website without logging out
- **3:** Malicious website executes a bank transfer without your knowledge using `Cookie-A`
- How can you **protect from CSRF?**
  - **1:** Synchronizer token pattern
    - A token created for each request
    - To make an update (POST, PUT, ..), you need a CSRF token from the previous request
  - **2:** SameSite cookie (`Set-Cookie: SameSite=Strict`)
    - `application.properties`
      - `server.servlet.session.cookie.same-site=strict`
    - Depends on browser support



# Getting Started with CORS

```
@Bean
public WebMvcConfigurer corsConfigurer() {
    return new WebMvcConfigurer() {
        public void addCorsMappings(CorsRegistry registry) {
            registry.addMapping("/**")
                .allowedMethods("*")
                .allowedOrigins("http://localhost:3000");
        }
    };
}
```

- Browsers do NOT allow AJAX calls to resources outside current origin
- Cross-Origin Resource Sharing (CORS): Specification that allows you to configure which cross-domain requests are allowed
  - **Global Configuration**
    - Configure addCorsMappings callback method in WebMvcConfigurer
  - **Local Configuration**
    - @CrossOrigin - Allow from all origins
    - @CrossOrigin(origins = "https://www.in28minutes.com") - Allow from specific origin

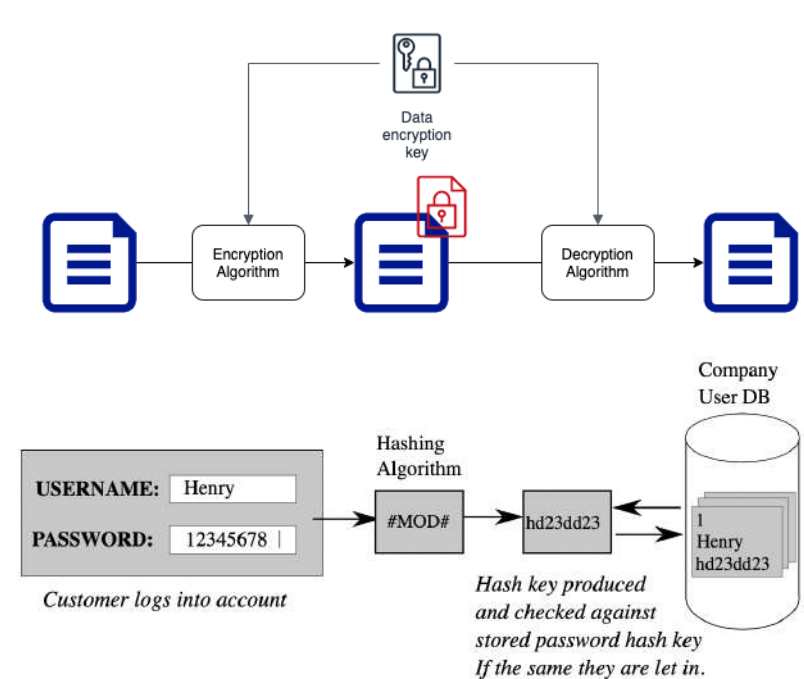
# Storing User Credentials

```
@Bean
public UserDetailsService userDetailsService(DataSource dataSource) {
    UserDetails user = User.builder()
        .username("in28minutes")
        //password("{noop}dummy")
        .password("dummy")
        .roles("USER")
        .passwordEncoder(str -> passwordEncoder().encode(str))
        .build();
    JdbcUserDetailsManager users = new JdbcUserDetailsManager(dataSource);
    users.createUser(user);
    return users;
    //return new InMemoryUserDetailsManager(user);
}
```

- User credentials can be stored in:
  - **In Memory** - For test purposes. Not recommended for production.
  - **Database** - You can use JDBC/JPA to access the credentials.
  - **LDAP** - Lightweight Directory Access Protocol
    - Open protocol for directory services and authentication

# Encoding vs Hashing vs Encryption

- **Encoding:** Transform data - one form to another
  - Does NOT use a key or password
  - Is reversible
  - Typically NOT used for securing data
  - Usecases: Compression, Streaming
  - Example: Base 64, Wav, MP3
- **Hashing:** Convert data into a Hash (a String)
  - One-way process
  - NOT reversible
    - You CANNOT get the original data back!
  - Usecases: Validate integrity of data
  - Example: bcrypt, scrypt
- **Encryption:** Encoding data using a key or password
  - You need to key or password to decrypt
  - Example: RSA

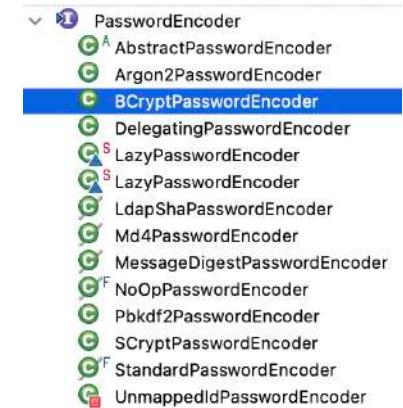


<https://upload.wikimedia.org/wikipedia/commons/5/5e/CPT->

*Hashing-Password-Login.svg*

# Spring Security - Storing Passwords

- Hashes like SHA-256 are no longer secure
- Modern systems can perform billions of hash calculations a second
  - AND systems improve with time!
- Recommended: Use adaptive one way functions with Work factor of 1 second
  - It should take at least 1 second to verify a password on your system
  - Examples: bcrypt, scrypt, argon2, ..
- PasswordEncoder - interface for performing one way transformation of a password
  - (REMEMBER) Confusingly Named!
  - BCryptPasswordEncoder



# Getting Started with JWT

- **Basic Authentication**
  - No Expiration Time
  - No User Details
  - Easily Decoded
- How about a **custom token system**?
  - Custom Structure
  - Possible Security Flaws
  - Service Provider & Service Consumer should understand
- **JWT (Json Web Token)**
  - Open, industry standard for representing claims securely between two parties
  - Can Contain User Details and Authorizations

```
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJzdWIiOiIxMjM0NTY3ODkwIiwibmFtZSI6IkpvaG4gRG9lIiwiaWF0IjoxNTE2MjM5MDIyfQ.SflKxwRJSMeKKF2QT4fwpMeJf36P0k6yJV_adQssw5c
```

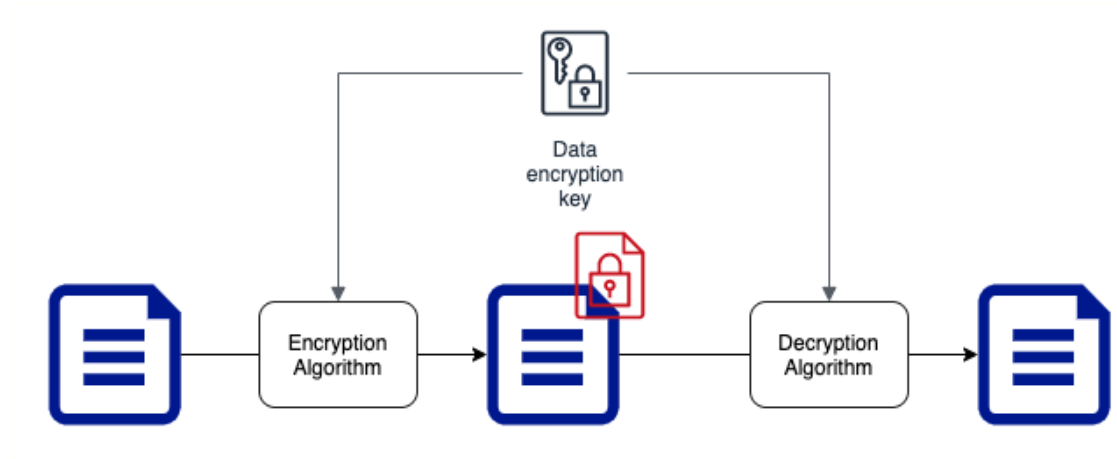
# What does a JWT contain?

- **Header**
  - Type: JWT
  - Hashing Algorithm: HS512
- **Payload**
  - **Standard Attributes**
    - **iss**: The issuer
    - **sub**: The subject
    - **aud**: The audience
    - **exp**: When does token expire?
    - **iat**: When was token issued?
  - **Custom Attributes**
    - **youratt1**: Your custom attribute 1
- **Signature**
  - Includes a Secret

HEADER: ALGORITHM & TOKEN TYPE
<pre>{   "alg": "HS256",   "typ": "JWT" }</pre>
PAYLOAD: DATA
<pre>{   "sub": "1234567890",   "name": "John Doe",   "iat": 1516239022 }</pre>
VERIFY SIGNATURE
<pre>HMACSHA256(   base64UrlEncode(header) + "." +   base64UrlEncode(payload),   your-256-bit-secret ) <input type="checkbox"/> secret base64 encoded</pre>



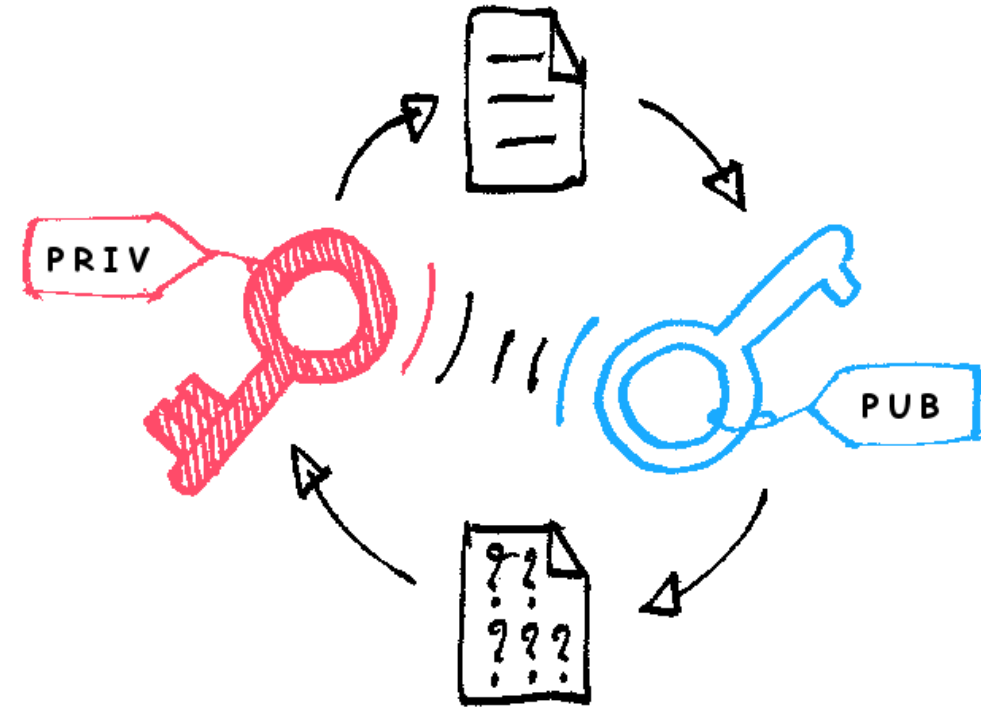
# Symmetric Key Encryption



- Symmetric encryption algorithms use the **same key for encryption and decryption**
- Key Factor 1: Choose the **right encryption algorithm**
- Key Factor 2: How do we **secure the encryption key**?
- Key Factor 3: How do we **share the encryption key**?

# Asymmetric Key Encryption

- **Two Keys** : Public Key and Private Key
- Also called **Public Key Cryptography**
- Encrypt data with Public Key and decrypt with Private Key
- Share Public Key with everybody and keep the Private Key with you(YEAH, ITS PRIVATE!)
- No crazy questions:
  - Will somebody not figure out private key using the public key?
- **Best Practice**: Use Asymmetric Keys



[https://commons.wikimedia.org/wiki/File:Asymmetric\\_encryption\\_\(colored\).p](https://commons.wikimedia.org/wiki/File:Asymmetric_encryption_(colored).p)

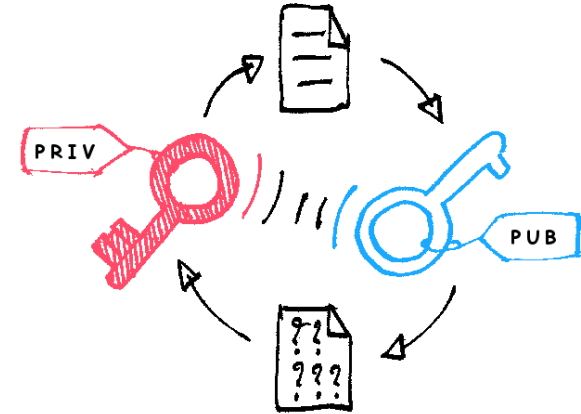
# Understanding High Level JWT Flow

- 1: Create a JWT
  - Needs Encoding
    - 1: User credentials
    - 2: User data (payload)
    - 3: RSA key pair
  - We will create a JWT Resource for creating JWT later
- 2: Send JWT as part of request header
  - Authorization Header
  - Bearer Token
  - `Authorization: Bearer ${JWT_TOKEN}`
- 3: JWT is verified
  - Needs Decoding
  - RSA key pair (Public Key)



# Getting Started with JWT Security Configuration

- JWT Authentication using Spring Boot's OAuth2 Resource Server
  - **1: Create Key Pair**
    - We will use `java.security.KeyPairGenerator`
    - You can use `openssl` as well
  - **2: Create RSA Key object using Key Pair**
    - `com.nimbusds.jose.jwk.RSAKey`
  - **3: Create JWKSSource (JSON Web Key source)**
    - Create `JWKSet` (a new JSON Web Key set) with the RSA Key
    - Create `JWKSSource` using the `JWKSet`
  - **4: Use RSA Public Key for Decoding**
    - `NimbusJwtDecoder.withPublicKey(rsaKey().toRSAPublicKey()).build()`
  - **5: Use JWKSSource for Encoding**
    - `return new NimbusJwtEncoder(jwkSource());`
    - We will use this later in the JWT Resource



# Getting Started with JWT Resource

```
username:"in28minutes",  
password:"dummy"
```

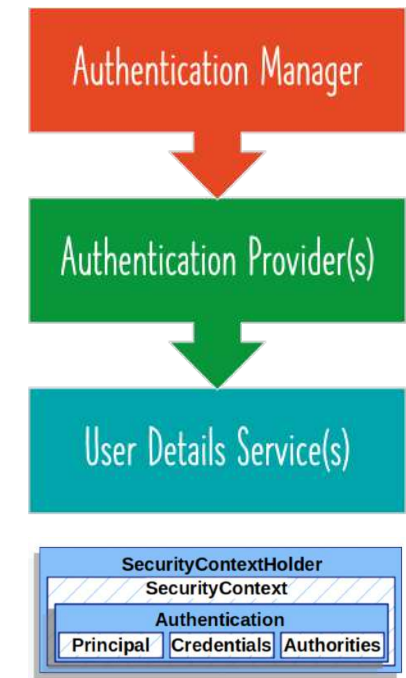
Response

```
{  
  "token": "TOKEN_VALUE"  
}
```

- Step 1: Use Basic Auth for getting the JWT Token
- Step 2-n: Use JWT token as Bearer Token for authenticating requests

# Understanding Spring Security Authentication

- Authentication is done as part of the Spring Security Filter Chain!
- **1: AuthenticationManager** - Responsible for authentication
  - Can interact with multiple authentication providers
- **2: AuthenticationProvider** - Perform specific authentication type
  - `JwtAuthenticationProvider` - JWT Authentication
- **3: UserDetailsService** - Core interface to load user data
- How is authentication result stored?
  - `SecurityContextHolder` > `SecurityContext` > `Authentication` > `GrantedAuthority`
    - **Authentication** - (After authentication) Holds user (**Principal**) details
    - **GrantedAuthority** - An authority granted to principal (roles, scopes,..)



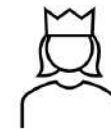
# Exploring Spring Security Authorization

- 1: Global Security: `authorizeHttpRequests`
  - `.requestMatchers("/users").hasRole("USER")`
    - `hasRole`, `hasAuthority`, `hasAnyAuthority`, `isAuthenticated`
- 2: Method Security (`@EnableMethodSecurity`)
  - `@Pre` and `@Post` Annotations
    - `@PreAuthorize("hasRole('USER') and #username == authentication.name")`
    - `@PostAuthorize("returnObject.username == 'in28minutes'")`
  - JSR-250 annotations
    - `@EnableMethodSecurity(jsr250Enabled = true)`
    - `@RolesAllowed({"ADMIN", "USER"})`
  - `@Secured` annotation
    - `@EnableMethodSecurity(securedEnabled = true)`
    - `@Secured({"ADMIN", "USER"})`
- (REMEMBER) JWT: Use  
`hasAuthority('SCOPE ROLE USER')`



# Getting Started with OAuth

- How can you give an application access to files present on your google drive?
  - You don't want to provide your credentials (NOT SECURE)
- OAuth: Industry-standard protocol for authorization
  - Also supports authentication now!
- Let's say you want to provide access to your Google Drive files to the Todo management application!
  - Important Concepts:
    - **Resource owner:** You (Person owning the google drive files)
    - **Client application:** Todo management application
    - **Resource server:** Contains the resources that are being accessed - Google Drive
    - **Authorization server:** Google OAuth Server





# Spring AOP

# What is Aspect Oriented Programming?

- A layered approach is typically used to build applications:
  - **Web Layer** - View logic for web apps OR JSON conversion for REST API
  - **Business Layer** - Business Logic
  - **Data Layer** - Persistence Logic
- Each layer has **different responsibilities**
  - HOWEVER, there are a **few common aspects** that apply to all layers
    - Security
    - Performance
    - Logging
  - These common aspects are called **Cross Cutting Concerns**
  - **Aspect Oriented Programming** can be used to implement Cross Cutting Concerns



# What is Aspect Oriented Programming? - 2

- **1:** Implement the cross cutting concern as an aspect
- **2:** Define point cuts to indicate where the aspect should be applied
- **TWO Popular AOP Frameworks**
  - **Spring AOP**
    - NOT a complete AOP solution BUT very popular
    - Only works with Spring Beans
    - Example: Intercept method calls to Spring Beans
  - **AspectJ**
    - Complete AOP solution BUT rarely used
    - Example: Intercept any method call on any Java class
    - Example: Intercept change of values in a field
- We will be focusing on **Spring AOP** in this section



# Aspect Oriented Programming - Important Terminology

- **Compile Time**

- **Advice** - What code to execute?
  - Example: Logging, Authentication
- **Pointcut** - Expression that identifies method calls to be intercepted
  - Example: `execution( com.in28minutes.aop.data.*(..) )`
- **Aspect** - A combination of
  - 1: Advice - what to do AND
  - 2: Pointcut - when to intercept a method call
- **Weaver** - Weaver is the framework that implements AOP
  - AspectJ or Spring AOP

- **Runtime**

- **Join Point** - When pointcut condition is true, the advice is executed. A specific execution instance of an advice is called a Join Point.



# Aspect Oriented Programming - Important Annotations

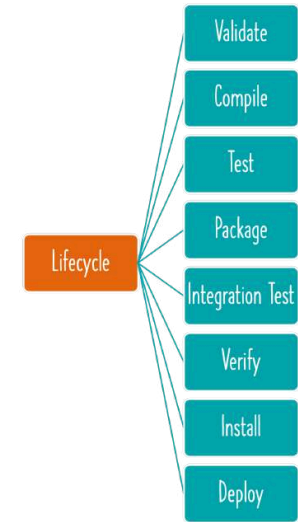
- **@Before** - Do something before a method is called
- **@After** - Do something after a method is executed irrespective of whether:
  - 1: Method executes successfully OR
  - 2: Method throws an exception
- **@AfterReturning** - Do something ONLY when a method executes successfully
- **@AfterThrowing** - Do something ONLY when a method throws an exception
- **@Around** - Do something before and after a method execution
  - Do something AROUND a method execution



# 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

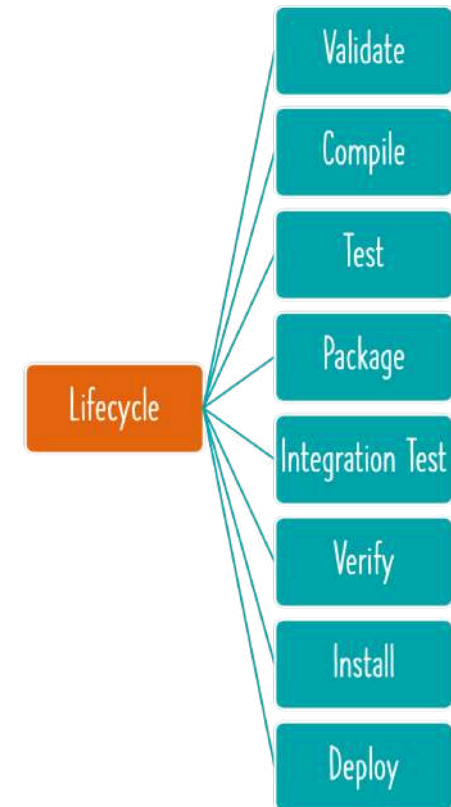


- Let's explore Project Object Model - pom.xml
  - **1: Maven dependencies:** Frameworks & libraries used in a project
    - Ex: `spring-boot-starter-web` and `spring-boot-starter-test`
    - Why are there so many dependencies in the classpath?
      - Answer: Transitive Dependencies
      - (REMEMBER) 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`



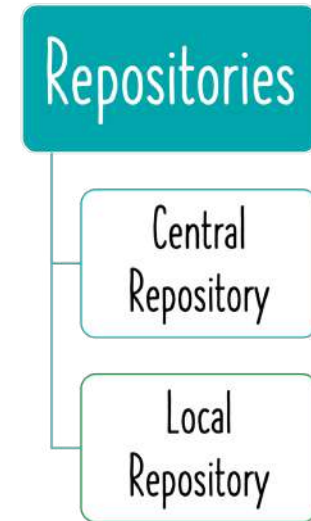
# Exploring Maven Build Life Cycle

- 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



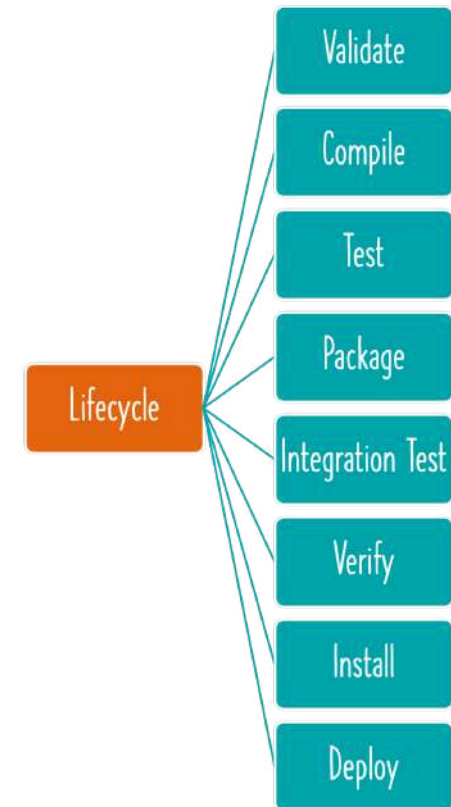
# How does Maven Work?

- 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.



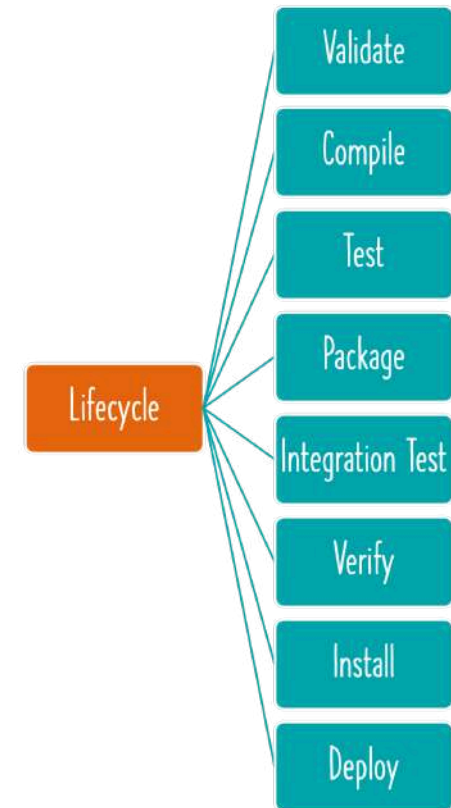
# Important Maven Commands

- mvn --version
- mvn compile: Compile source files
- mvn test-compile: Compile test files
  - 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

- **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?



- **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



# 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

A vertical stack of four colored rectangles representing the layers of a Docker container. From top to bottom: a teal rectangle labeled 'Applications', a green rectangle labeled 'Software', an orange rectangle labeled 'OS', and a brown rectangle labeled 'Hardware'.

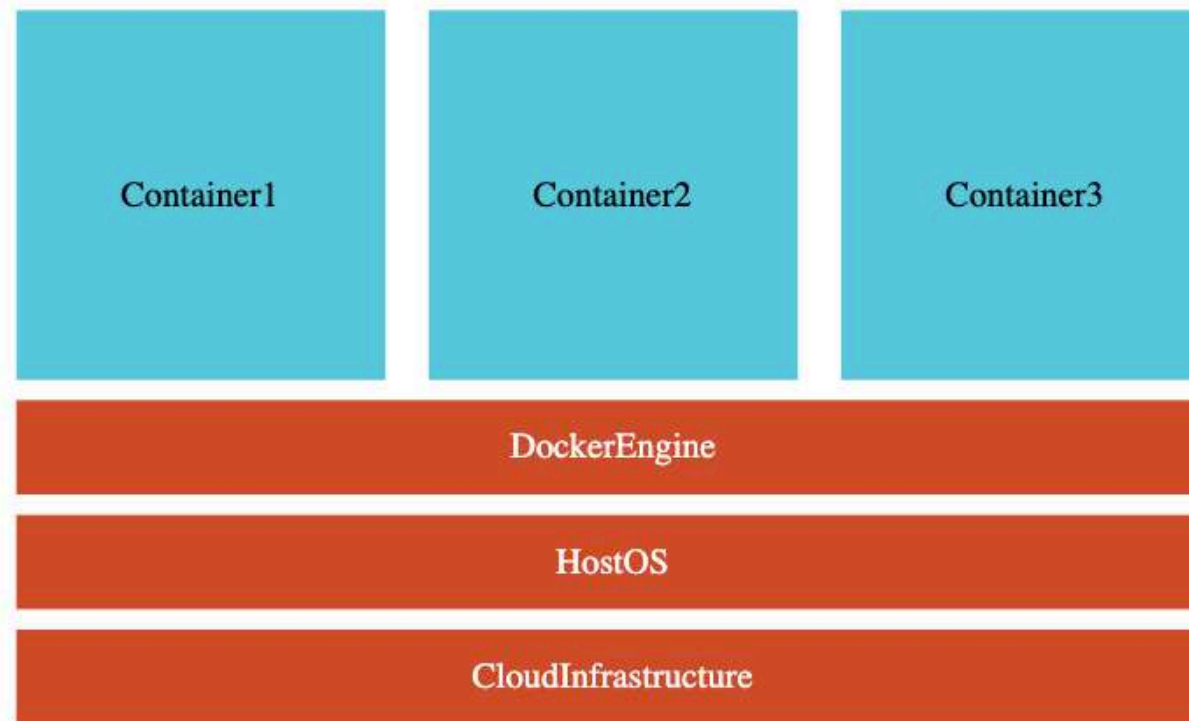
Applications

Software

OS

Hardware

# Run Docker Containers Anywhere



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

# Why is Docker Popular?

## Standardized Application Packaging

Same packaging for  
all types of applications

- Java, Python or JS

## Multi Platform Support

Local Machine  
Data Center  
Cloud - AWS, Azure and GCP

## Isolation

Containers have isolation  
from one another

Docker

# What's happening in the Background?

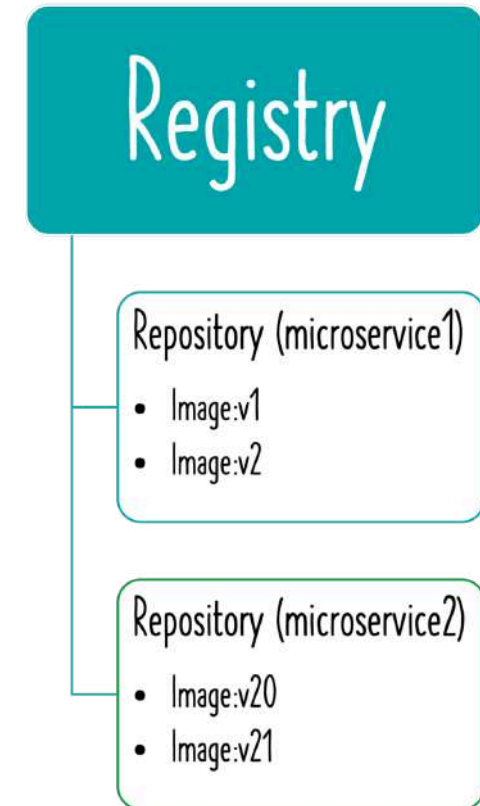
```
docker container run -d -p 5000:5000 in28min/hello-world-nodejs:0.0.1.RELEASE
```

- **Docker image** is downloaded from Docker Registry (Default: Docker Hub)
  - *<https://hub.docker.com/r/in28min/hello-world-nodejs>*
  - 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:** Detached 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/DemoDockApplication.java /
    /home/app/src/main/java/com/example/demodocker/DemoDockApplication.java

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

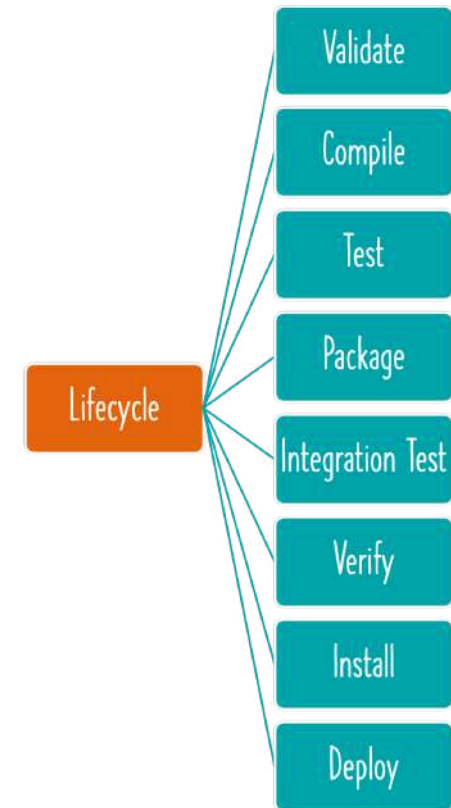
COPY . /home/app
RUN mvn -f /home/app/pom.xml clean package

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)



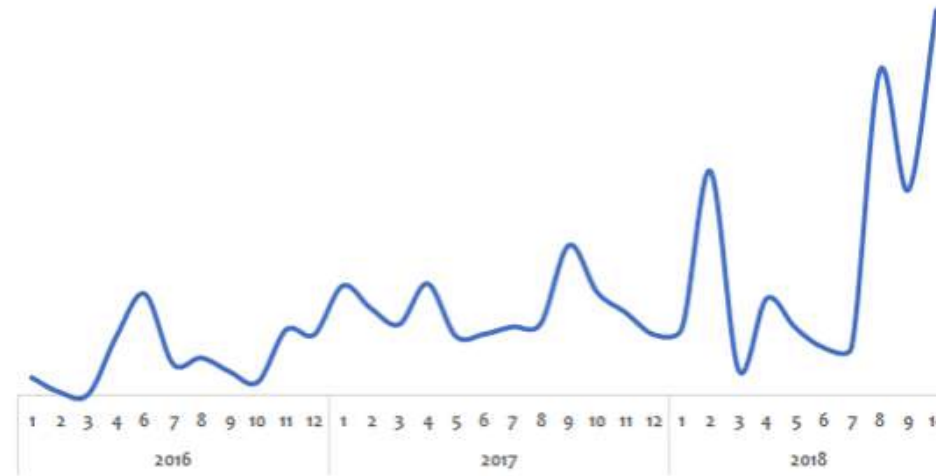
# Creating Docker Images - Dockerfile

```
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

# Learning AWS

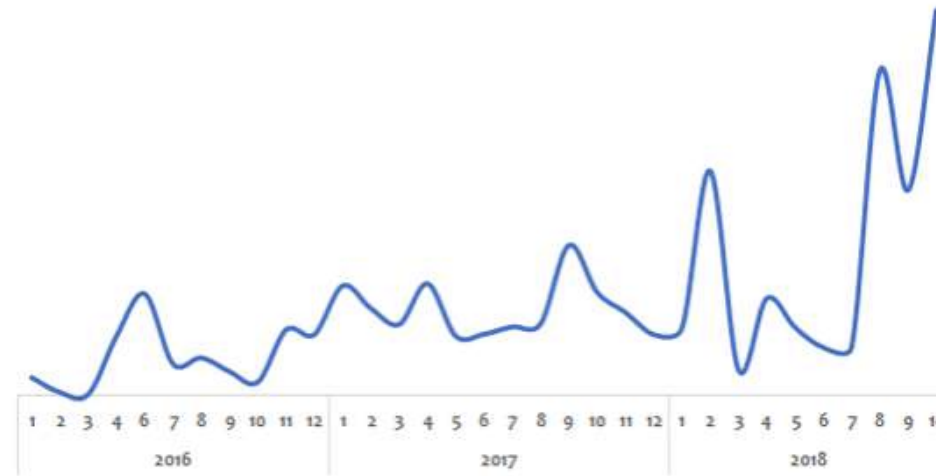
# Before the Cloud - Example 1 - Online Shopping App



- **Challenge:**
  - Peak usage during holidays and weekends
  - Less load during rest of the time
- **Solution (before the Cloud):**
  - **Procure (Buy) infrastructure for peak load**
    - QUESTION: What would the infrastructure be doing during periods of low loads?

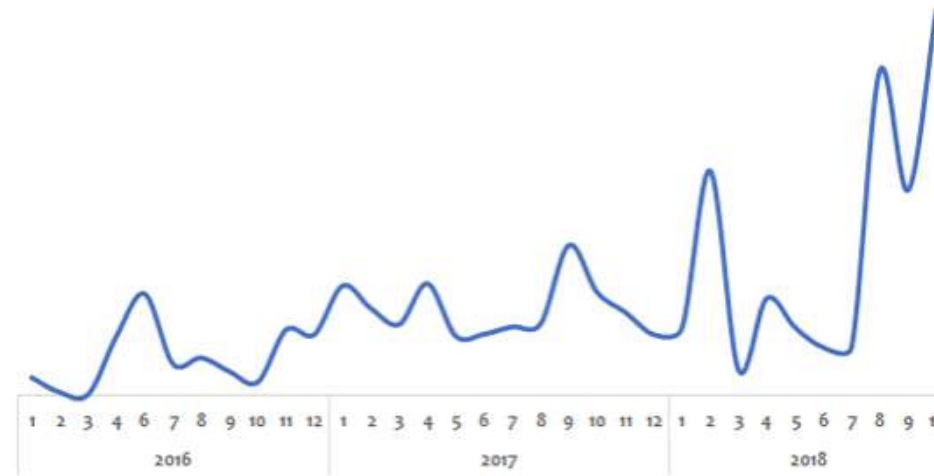


# Before the Cloud - Example 2 - Startup



- **Challenge:**
  - It suddenly becomes popular.
  - How to handle the **sudden increase** in load?
- **Solution** (before the Cloud):
  - **Procure** (Buy) infrastructure assuming they would be successful
    - QUESTION: What if they are not successful?

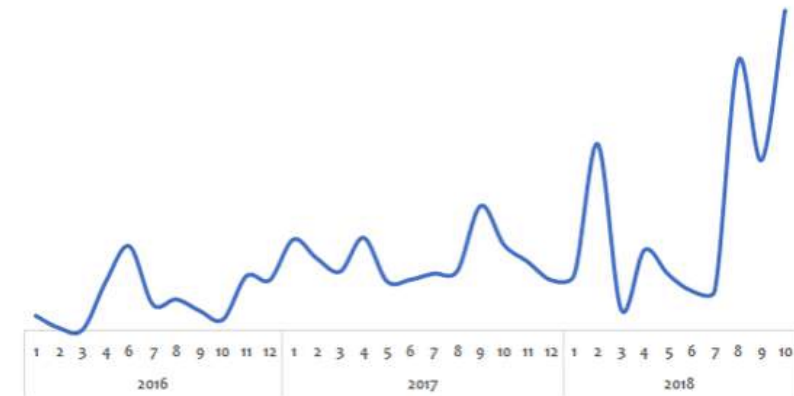
# Before the Cloud - Challenges



- Low infrastructure utilization (**PEAK LOAD** provisioning)
- Needs ahead of time planning (**Can you guess the future?**)
- High cost of procuring infrastructure
- Dedicated infrastructure maintenance team (**Can a startup afford it?**)

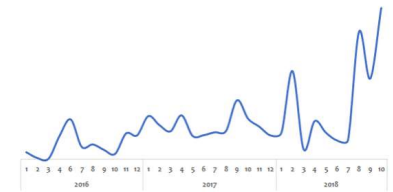
# Silver Lining in the Cloud

- How about **provisioning (renting) resources** when you want them and releasing them back when you do not need them?
  - On-demand resource provisioning
  - Also called **Elasticity**



# Cloud - Advantages

- Trade "capital expense" for "variable expense"
- Benefit from massive economies of scale
- Stop guessing capacity
- "Go global" in minutes
- Avoid undifferentiated heavy lifting
- Stop spending money running and maintaining data centers

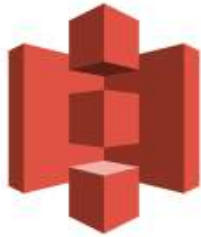


# Amazon Web Services (AWS)

- Leading cloud service provider
  - Competitors: Microsoft Azure and Google Cloud
- Provides **MOST** (200+) services
- Reliable, secure and cost-effective
- You will learn more about AWS as we go further in the course!



# Best path to learn AWS!



Amazon S3



EC2



Amazon EBS



ELB



ECS

- Cloud applications make use of multiple AWS services.
- There is **no single path** to learn these services independently.
- HOWEVER, we've worked out a simple path!

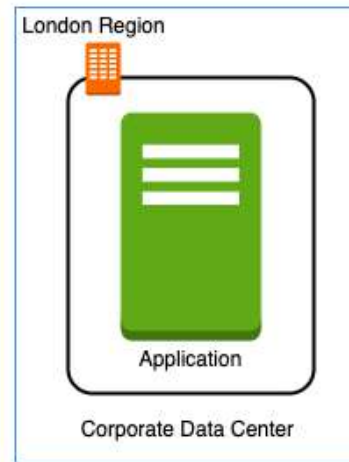
# Setting up AWS Account

- Create an AWS Account
- Setup an IAM user

# Regions and Zones

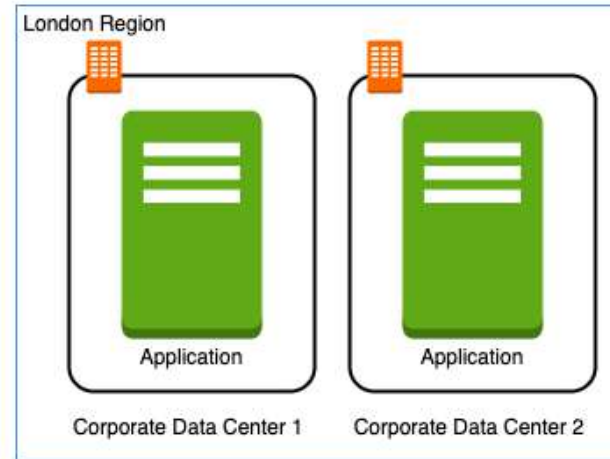


# Regions and Zones



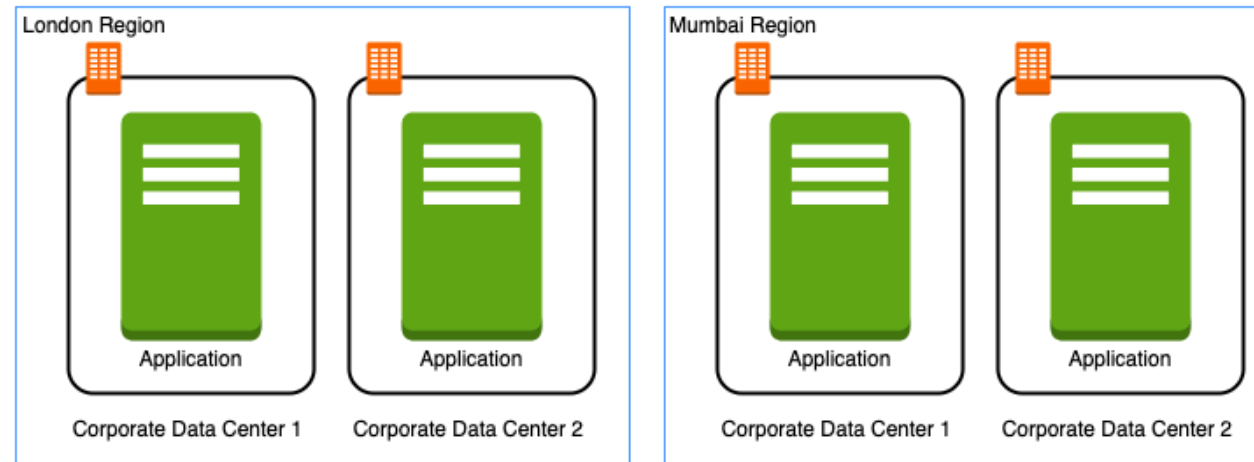
- Imagine that your application is deployed in a data center in London
- What would be the challenges?
  - Challenge 1 : Slow access for users from other parts of the world (**high latency**)
  - Challenge 2 : What if the data center crashes?
    - Your application goes down (**low availability**)

# Multiple data centers



- Let's add in one more data center in London
- What would be the challenges?
  - Challenge 1 : Slow access for users from other parts of the world
  - Challenge 2 (**SOLVED**) : What if one data center crashes?
    - Your application is **still available** from the other data center
  - Challenge 3 : What if **entire region** of London is unavailable?
    - Your application goes down

# Multiple regions



- Let's add a new region : Mumbai
- What would be the challenges?
  - Challenge 1 (**PARTLY SOLVED**) : Slow access for users from other parts of the world
    - You can solve this by adding deployments for your applications in other regions
  - Challenge 2 (**SOLVED**) : What if one data center crashes?
    - Your application is still live from the other data centers
  - Challenge 3 (**SOLVED**) : What if entire region of London is unavailable?
    - Your application is served from Mumbai

# Regions



- Imagine setting up data centers in different regions around the world
  - Would that be easy?
- (Solution) AWS provides **25+ regions** around the world (expanding every year)

# Regions - Advantages

- High Availability
- Low Latency
- Global Footprint
- Adhere to government **regulations**

# Availability Zones

- Each AWS Region consists of multiple AZ's
- Each Availability Zone:
  - Can have **One or more discrete data centers**
  - has **independent & redundant** power, networking & connectivity
- AZs in a Region are connected through **low-latency** links
- (Advantage) **Increase availability and fault tolerance** of applications in the same region



# Regions and Availability Zones examples

*New Regions and AZs are constantly added*

Region Code	Region	Availability Zones	Availability Zones List
us-east-1	US East (N. Virginia)	6	us-east-1a us-east-1b us-east-1c us-east-1d us-east-1e us-east-1f
eu-west-2	Europe (London)	3	eu-west-2a eu-west-2b eu-west-2c
ap-south-1	Asia Pacific(Mumbai)	3	ap-south-1a ap-south-1b ap-south-1c

# EC2 Fundamentals



# Introduction to EC2 (Elastic Compute Cloud)



EC2



EC2 Instances

- In corporate data centers, applications are deployed to physical servers
- Where do you deploy applications in the cloud?
  - Rent virtual machines
  - **EC2 instances** - Virtual machines in AWS
  - **EC2 service** - Provision EC2 instances or virtual machines

# Understanding Important Features of EC2



EC2 Instances



ELB



Amazon EBS

- Create and manage lifecycle of EC2 instances
- **Attach storage** to your EC2 instances
- **Load balancing** for multiple EC2 instances
- **Our Goal:** Play with EC2 instances!

# Reviewing Important EC2 Concepts

Feature	Explanation
Amazon Machine Image (AMI)	What operating system and what software do you want on the instance?
Instance Families	Choose the right family of hardware (General purpose or Compute/Storage/Memory optimized or GPU)
Instance Size (t2.nano, t2.micro, t2.small, t2.medium ...)	Choose the right quantity of hardware (2 vCPUs, 4GB of memory)
Elastic Block Store	Attach Disks to EC2 instances (Block Storage)
Security Group	Virtual firewall to control <b>incoming and outgoing</b> traffic to/from AWS resources (EC2 instances, databases etc)
Key pair	Public key and a private key Public key is stored in EC2 instance Private key is stored by customer

# IAM & Best Practices



- **IAM: Identity and Access Management**
  - **Authentication** (the right user?) and
  - **Authorization** (the right access?)
  - **Root User:** User we created our AWS account with
    - Credentials: Email address and password
    - DO NOT use Root User for day to day activities
    - Create a new IAM User and use the IAM user for regular activities
- **Things we will do now:**
  - 1: Create an IAM Group - Developers - with admin access
  - 2: Create an IAM user - in28minutes\_dev - with group Developers
  - 3: Login with IAM user - in28minutes\_dev
- (Remember) Bookmark Your Account Specific AWS URL

# Cloud Best Practices - Managing Costs



- **With Great Power comes Great Responsibility**
  - Cloud provides you with ability to create powerful resources
  - HOWEVER its important to understand the associated costs
- **5 Best Practices**
  - **1:** For the first week, monitor the billing dashboard everyday
  - **2:** Set Budget Alerts
    - 1: Enable Billing Alerts - My Billing Dashboard > Billing preferences
    - 2: Create Budget Alert - Budgets > Create a Budget > Cost Budget > Alert
  - **3:** STOP resources when you are not using them
  - **4:** Understand FREE Tier and 12 Month Limits (HARD TO DO)
  - **5:** Understand how pricing works for diff. resources (HARD TO DO)

# Cloud Services

# Cloud Services

- Do you want to continue running applications in the cloud, the same way you run them in your data center?
- OR are there OTHER approaches?
- You should understand some terminology:
  - IaaS (Infrastructure as a Service)
  - PaaS (Platform as a Service)
  - ....
- Let's get on a quick **journey** to understand these!



# IAAS (Infrastructure as a Service)

- Use **only infrastructure** from cloud provider
  - Ex: Using VM service to deploy your apps/databases
- **Cloud provider** is responsible for:
  - Hardware, Networking & Virtualization
- You are responsible for:
  - OS upgrades and patches
  - Application Code and Runtime
  - Configuring load balancing
  - Auto scaling
  - Availability
  - etc.. ( and a lot of things!)





# PAAS (Platform as a Service)

- Use a platform provided by the cloud
  - **Cloud provider** is responsible for:
    - Hardware, Networking & Virtualization
    - OS (incl. upgrades and patches)
    - Application Runtime
    - Auto scaling, Availability & Load balancing etc..
  - **You** are responsible for:
    - Configuration (of Application and Services)
    - Application code (if needed)
- **Examples:**
  - **Compute:** AWS Elastic Beanstalk, Azure App Service, Google App Engine
  - **Databases:** Relational & NoSQL (Amazon RDS, Google Cloud SQL, Azure SQL Database etc)
  - Queues, AI, ML, Operations etc!



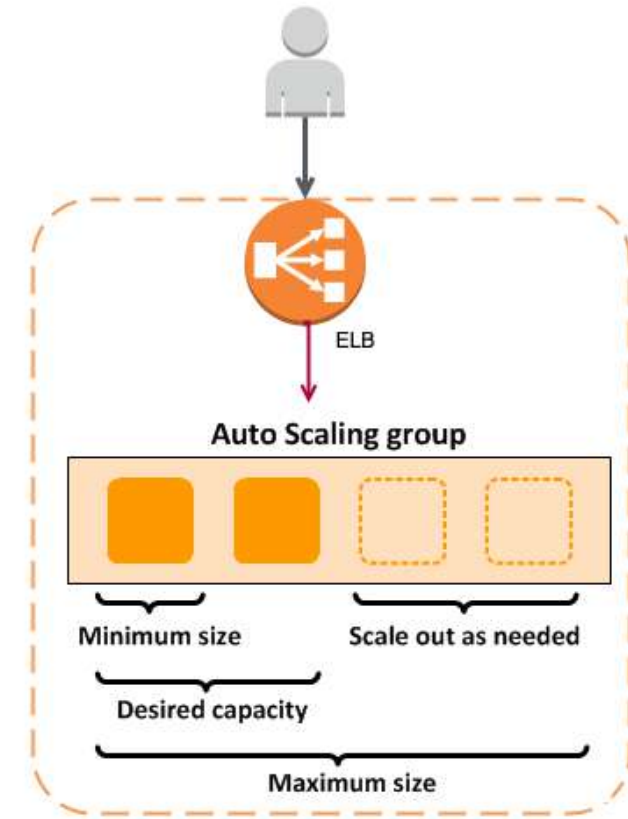
# AWS Elastic BeanStalk

- **Simplest way** to deploy and scale your web applications in AWS
  - Provides end-to-end web application management
  - Supports Java, .NET, Node.js, PHP, Ruby, Python, Go, and Docker applications
  - **No usage charges** - Pay for AWS resources provisioned
- **Features:**
  - Automatic load balancing
  - Auto scaling
  - Managed platform updates

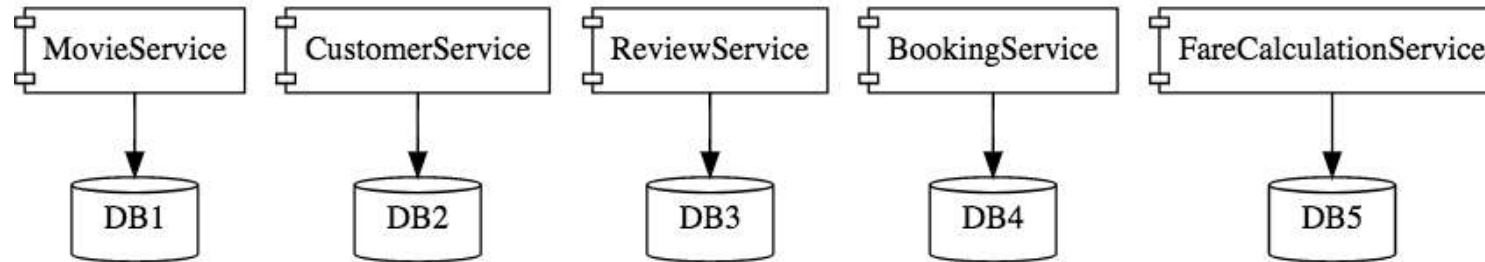


# Auto Scaling Group and Elastic Load Balancing

- Applications have millions of users:
  - Same application is deployed to multiple VMs
- How do you simplify creation and management of **multiple VMs**?
  - **Auto Scaling Groups**
  - Allow you to create and manage a group of EC2 instances
- How do you distribute traffic across multiple EC2 instances?
  - **Elastic Load Balancing**



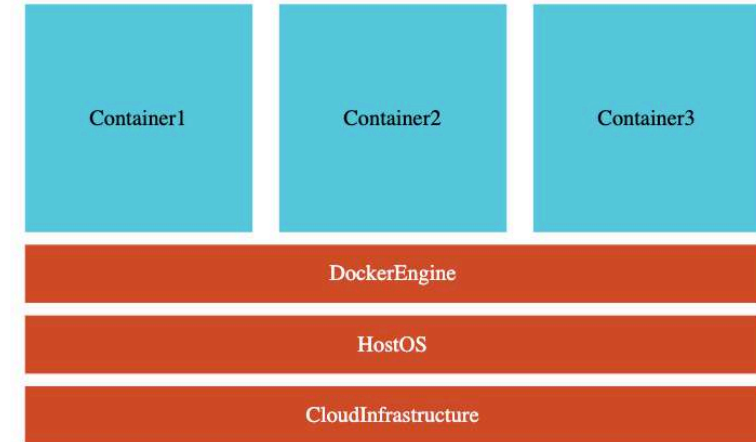
# Microservices



- 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!**

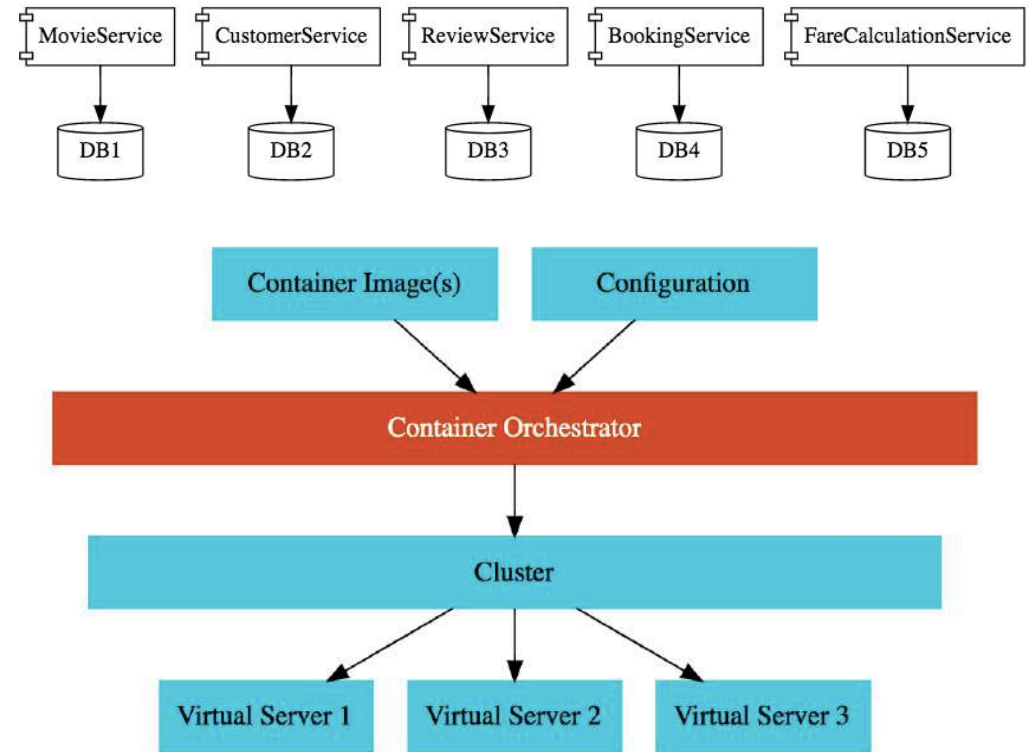
# Containers - Docker

- Create **Docker images** for each microservice
- Docker image **has all needs of a microservice**:
  - Application Runtime (JDK or Python or NodeJS)
  - Application code and Dependencies
  - Runs **the same way** on any infrastructure:
    - Your local machine, Corporate data center or in the Cloud
- **Advantages**
  - Docker is **cloud neutral**
  - Standardization: Simplified Operations
    - Consistent deployment, monitoring, logging ...
  - Docker containers are **light weight**
    - Compared to Virtual Machines as they do not have a Guest OS
  - Docker provides **isolation** for containers



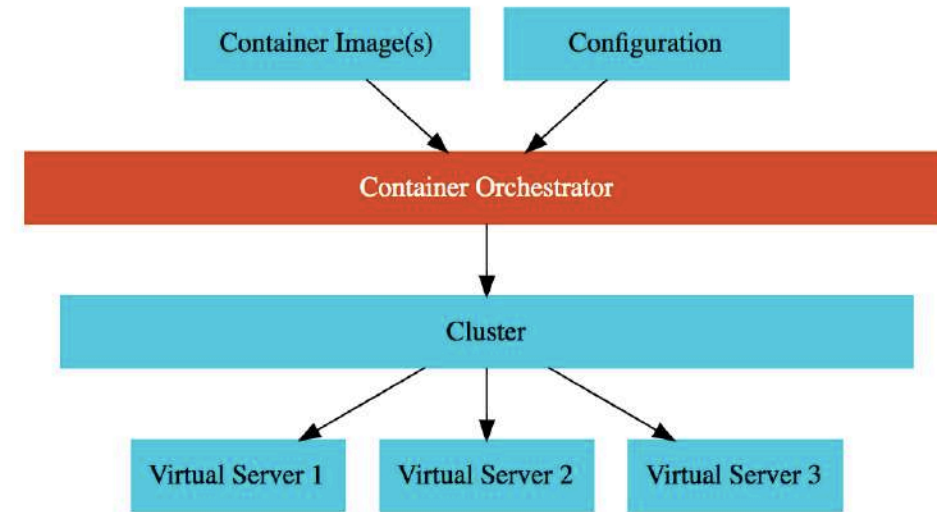
# Container Orchestration

- **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

- **Cloud Neutral: Amazon EKS**
  - Kubernetes: Open source container orchestration
  - Managed service: Amazon Elastic Kubernetes Service
  - EKS does not have a free tier
- **AWS Specific: Amazon ECS**
  - Amazon Elastic Container Service
- **Fargate: Serverless ECS/EKS**
  - AWS Fargate does not have a free tier



# Serverless



- What do **we think about** when we develop an application?
  - Where to deploy? What kind of server? What OS?
  - How do we take care of scaling and availability of the application?
- What if **you don't worry about servers and focus ONLY on code**?
  - Enter **Serverless**
    - Remember: **Serverless does NOT mean "No Servers"**
- **Serverless for me:**
  - You **don't worry** about infrastructure (ZERO visibility into infrastructure)
    - Flexible scaling and automated high availability
  - Most Important: **Pay for use**
    - Ideally ZERO REQUESTS => ZERO COST
- **You focus on code** and the cloud managed service takes care of all that is needed to scale your code to serve millions of requests!
  - And you pay for requests and NOT servers!



# AWS Lambda



AWS Lambda



Lambda Fn

- Truly serverless
- You don't worry about servers or scaling or availability
- You only worry about your code
- **You pay for what you use**
  - Number of requests
  - Duration of requests
  - Memory

# AWS Lambda - Supported Languages

- Java
- Go
- PowerShell
- Node.js
- C#
- Python,
- Ruby
- and a lot more...

# Review - AWS Services for Compute

AWS Service Name	Description
Amazon EC2 + ELB	Traditional Approach (Virtual Servers + Load Balancing) Use when you need control over OS OR you want to run custom software
AWS Elastic Beanstalk	Simplify management of web applications and batch applications Automatically creates EC2 + ELB(load balancing and auto scaling)
Amazon Elastic Container Service (Amazon ECS)	Simplify running of microservices with Docker containers Run containers in EC2 based ECS Clusters
Amazon Elastic Kubernetes Service (Amazon EKS)	Run and scale Kubernetes clusters
AWS Fargate	Serverless version of ECS and EKS
AWS Lambda	Serverless - Do NOT worry about servers

# You are all set!

# Let's clap for you!

- You have a lot of patience!  
**Congratulations**
- You have put your best foot forward **to be a great developer!**
- Don't stop your learning journey!
  - Keep Learning Every Day!
- Good Luck!



# Do Not Forget!

- **Recommend the course to your friends!**
  - Do not forget to **review!**
- **Your Success = My Success**
  - Share your success story with me on LinkedIn (Ranga Karanam)
  - Share your success story and lessons learnt in Q&A with other learners!



# FASTEST ROADMAPS

in28minutes.com



In28  
Minutes



Google Cloud  
Certifications



Azure  
Certifications



AWS  
Certifications



DevOps



Java Full Stack



Java Microservices

