Spring Boot 3
Spring 6
Hibernate

S.A. Ruvini Rangathara

Spring Boot-3 Overview

- ☐ Develop Spring Boot Application
- ☐ Leverage Hibernate/JPA for database access
- ☐ Develop a REST API using Spring Boot
- ☐ Create a Spring MVC app with Spring Boot
- ☐ Connect Spring Boot app to a database for CRUD development
- ☐ Apply Spring Security to control application access
- ☐ Leverage all Java configuration(no XML) and maven

Spring Boot 3 requires JDK 17 or higher

Spring

- ➢ Very popular framework for building Java applications.
- ➢ Provides a large number of helper classes and annotations.

The problem

- Building a traditional Spring application is really HARD!!!

Q: Which JAR dependencies do I need?
Q: How do I set up configuration (xml or Java)?
Q: How do I install the server? (Tomcat, JBoss etc…)

Spring Boot Solution

- Make it easier to get started with Spring development
- Minimize the amount of manual configuration
    - Perform auto-configuration based on props files and JAR classpath
- Help to resolve dependency conflicts (Maven or Gradle)
- Provide an embedded HTTP server so you can get started quickly
    - Tomcat, Jetty, Undertow, …

Spring Boot and Spring

- Spring Boot uses Spring behind the scenes
- Spring Boot simply makes it easier to use Spring

Spring Framework :

- Spring is a comprehensive framework for building Java applications.
- It provides support for dependency injection, aspect-oriented programming, transaction management, data access, messaging, and more.

Spring Boot :

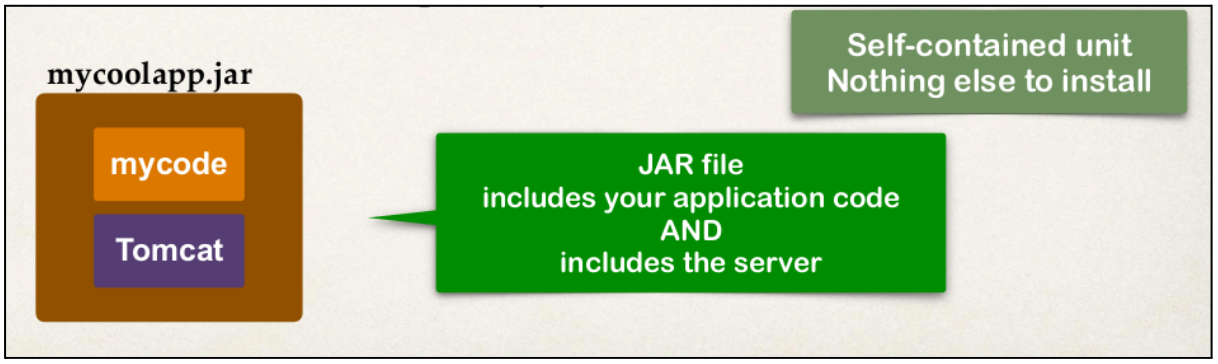
- Spring Boot is an opinionated framework built on top of the Spring Framework.
- It aims to simplify the process of building and deploying Spring-based applications.
- Spring Boot provides default configurations, embedded servers (like Tomcat, Jetty, or Undertow), and starter dependencies to jumpstart application development.
- It focuses on convention over configuration, allowing developers to get started quickly with minimal setup.
- Spring Boot also provides production-ready features like health checks, metrics, and externalized configuration out of the box.

In summary :

Spring is the broader framework providing a wide range of functionalities for Java application development.
Spring Boot is a tool that enhances the Spring Framework by simplifying the setup and development of Spring-based applications, focusing on convention over configuration and providing production-ready features.

So, while Spring Boot builds upon Spring, it's not a replacement for Spring. It's more of a complementary tool that makes it easier and faster to build Spring applications.

Spring Boot Embedded Server
Provide an embedded HTTP server so you can get started quickly
Tomcat, Jetty, Undertow, …
No need to install a server separately

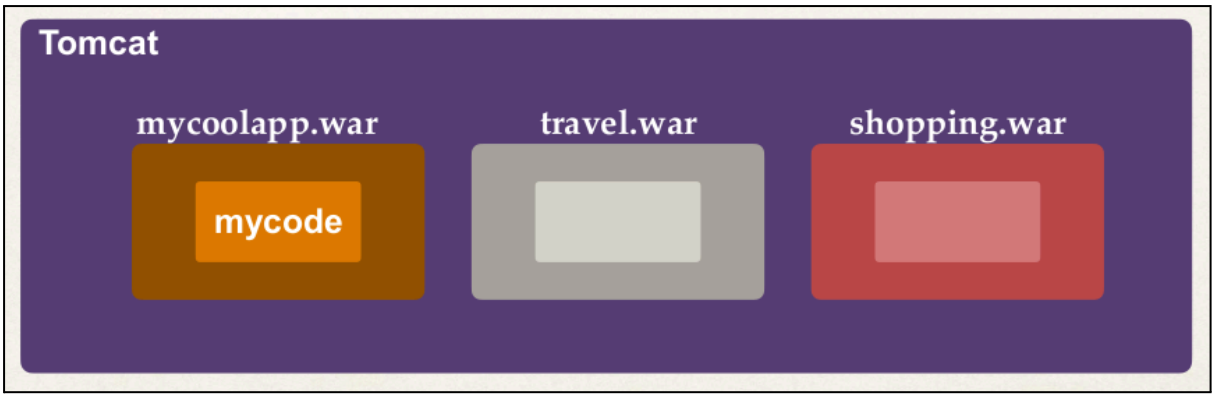


Running Spring Boot Apps
Spring Boot apps can be run standalone (includes embedded server)
Run the Spring Boot app from the IDE or command-line

```
java -jar mycoolapp.jar
```

Deploying Spring Boot Apps

- Spring Boot apps can also be deployed in the traditional way
- Deploy Web Application Archive (WAR) file to an external server:
    - Tomcat, JBoss, WebSphere etc …



Q: Does Spring Boot replace Spring MVC, Spring REST etc …?
Spring Boot simplifies the development of Spring applications, including Spring MVC-based web applications and RESTful services, by providing a convention-over-configuration approach and a range of additional features. It doesn't replace Spring MVC or Spring REST but rather enhances them and provides easier ways to work with them.

එය Spring MVC හෝ Spring REST replace නොකරන නමුත් ඒවා වැඩිදියුණු කර ඒවා සමඟ වැඩ කිරීමට පහසු ක්‍රම සපයයි.

Q: Does Spring Boot run code faster than regular Spring code?
No. Behind the scenes, Spring Boot uses the same code of Spring Framework.
Minimizing configuration etc …

Maven

- Apache Maven is a build automation tool primarily used for Java projects.
- It simplifies the process of managing dependencies, building, testing, and packaging Java applications.
- Maven uses a project object model (POM) file to describe the structure of a project, including its dependencies, build settings, and plugins configuration.

Java Plain Old Java Object (POJO)

A Java Plain Old Java Object (POJO) is a Java class that encapsulates only data and doesn't contain any business logic.

Here are some characteristics of a Java POJO:

1. **Simple Structure:** A POJO typically consists of private fields, public getter and setter methods (accessor methods) to access and modify the fields, and optionally constructors.

2. **No Dependencies:** A POJO shouldn't depend on any framework or base class other than the standard Java API. It means that a POJO shouldn't extend any specific class or implement any interface that's part of a framework or library.

3. **Serializable:** POJOs often implement the Serializable interface to enable serialization and deserialization, which allows objects to be converted into a stream of bytes for storage or transmission.

4. **JavaBean Conventions:** While not strictly necessary, POJOs often follow the conventions of JavaBeans, such as providing no-argument constructors and adhering to naming conventions for getter and setter methods.

5. **Used for Data Transfer:** POJOs are commonly used for transferring data between different layers of an application or between different systems. For example, in web development, POJOs are often used as data transfer objects (DTOs) to pass data between the frontend and backend of a web application.

Goals of Spring

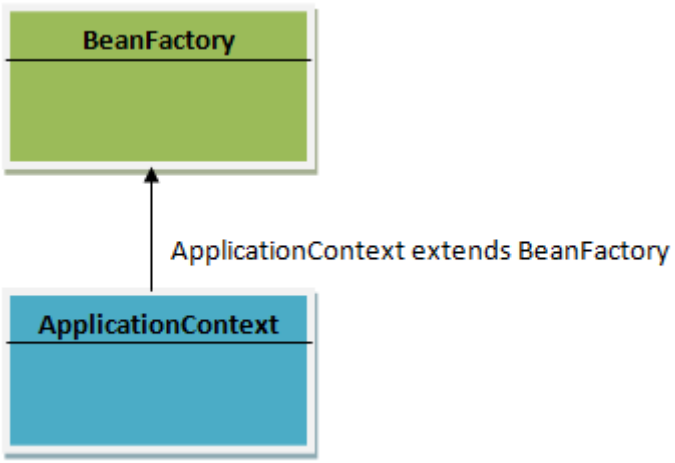
- Lightweight development with Java POJOs (Plain-Old-Java-Objects)
- Dependency injection to promote loose coupling
- Declarative programming with Aspect-Oriented-Programming (AOP)
- Minimize boilerplate Java code

Bean Factory :

- The Bean Factory is the fundamental interface in Spring for managing beans.
- It's the original container that provides basic support for dependency injection (DI) and lifecycle management of beans.
- Bean Factory loads bean definitions, instantiates beans, and manages their lifecycle.
- While it's a powerful mechanism for configuring and managing beans, it lacks some of the more advanced features provided by ApplicationContext.

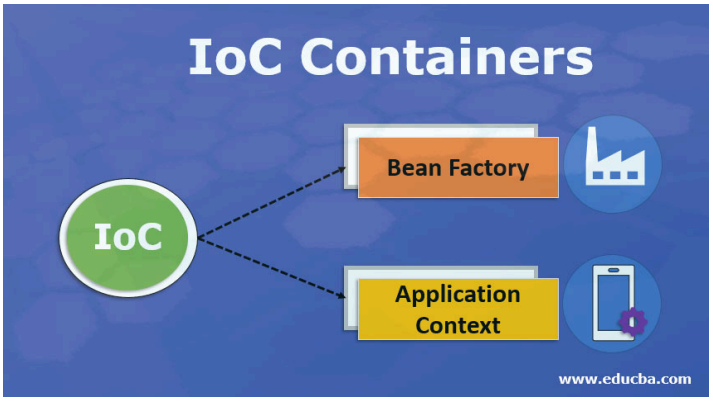
ApplicationContext :

- ApplicationContext is a more advanced container than the Bean Factory and is the preferred choice in most applications.
- It extends the functionality of the Bean Factory by providing additional features such as event propagation, internationalization support, resource loading, and application context hierarchy.
- ApplicationContext is typically used in Spring applications as it provides a more feature-rich environment for developing enterprise applications.



IoC Container :

- Inversion of Control (IoC) Container is that encompasses both Bean Factory and ApplicationContext.
- IoC Container is responsible for managing the lifecycle of objects, creating instances of objects, and injecting dependencies into those objects.



In Spring, both Bean Factory and ApplicationContext act as IoC Containers, providing IoC capabilities to Spring applications.

Both BeanFactory and Application Context are types of IoC (Inversion of Control) containers in the Spring Framework.

Servlet :

- Servlets are Java classes that extend the functionality of web servers to handle HTTP requests and generate HTTP responses.
- Spring Boot provides integration with servlet containers like Apache Tomcat, Jetty, and Undertow.

WebSocket :

- WebSocket is a communication protocol that provides full-duplex communication channels over a single TCP connection.
- It enables bidirectional communication between clients and servers, allowing real-time data exchange.
- In Spring Boot, WebSocket support is provided through the Spring WebSocket module, which allows developers to create WebSocket endpoints, handle WebSocket messages, and manage WebSocket sessions.

Spring Web :

- Spring Web is a module within the Spring Framework that provides support for building web applications.
- It includes features such as controllers, request mapping, data binding, form handling, validation, and view resolution.
- In Spring Boot, developers can leverage Spring Web MVC, a component of Spring Web, to build RESTful APIs and web applications using the MVC (Model-View-Controller) architectural pattern.

AOP - More About AOP

AOP breaks the program logic into distinct parts (called concerns). It is used to increase modularity by **cross-cutting concerns**.

A **cross-cutting concern** is a concern that can affect the whole application and should be centralized in one location in code as possible, such as transaction management, authentication, logging, security etc.

Spring Projects

Spring is a vast ecosystem with a wide range of projects aimed at simplifying various aspects of enterprise Java development.

Spring Framework: The core project of the Spring ecosystem, providing comprehensive infrastructure support for developing Java applications. It includes features such as dependency injection, aspect-oriented programming, transaction management, data access, and more.

Spring Boot: A framework that simplifies the process of building production-ready Spring-based applications. Spring Boot provides auto-configuration, embedded servers, opinionated defaults, and a range of starter dependencies to streamline application development.

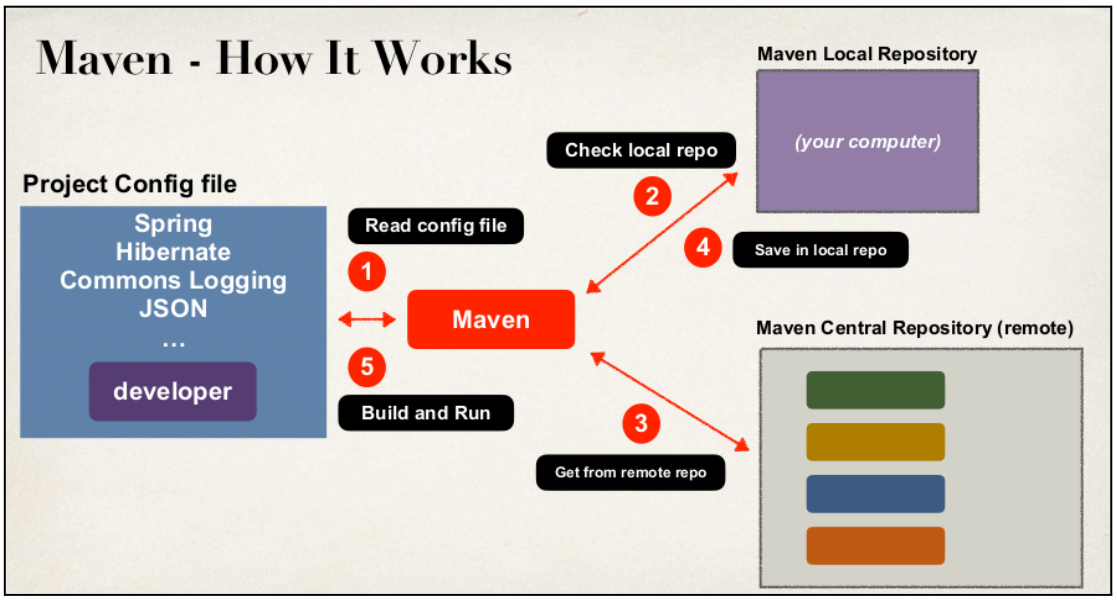
Spring Data: A set of projects that simplify data access in Spring applications. Spring Data provides abstractions and implementations for working with various data sources such as relational databases, NoSQL databases, key-value stores, and more. Examples include Spring Data JPA, Spring Data MongoDB, Spring Data Redis, etc.

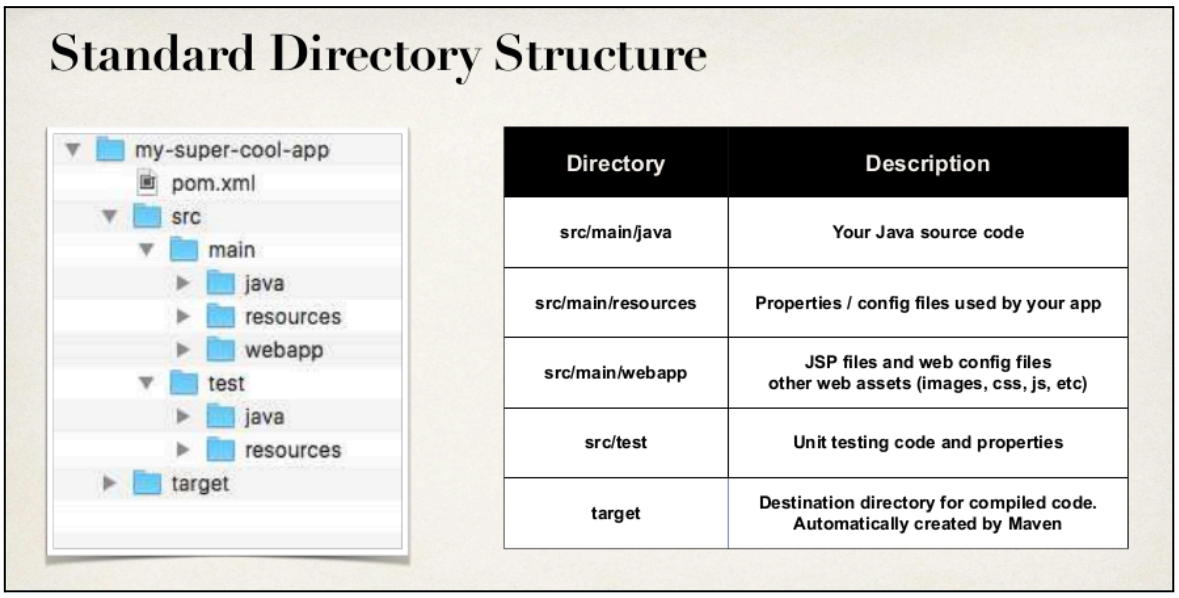
Spring Security: A powerful and customizable framework for authentication, authorization, and other security features in Spring applications. It provides robust security mechanisms for protecting web applications, REST APIs, and other resources.

==========

Maven

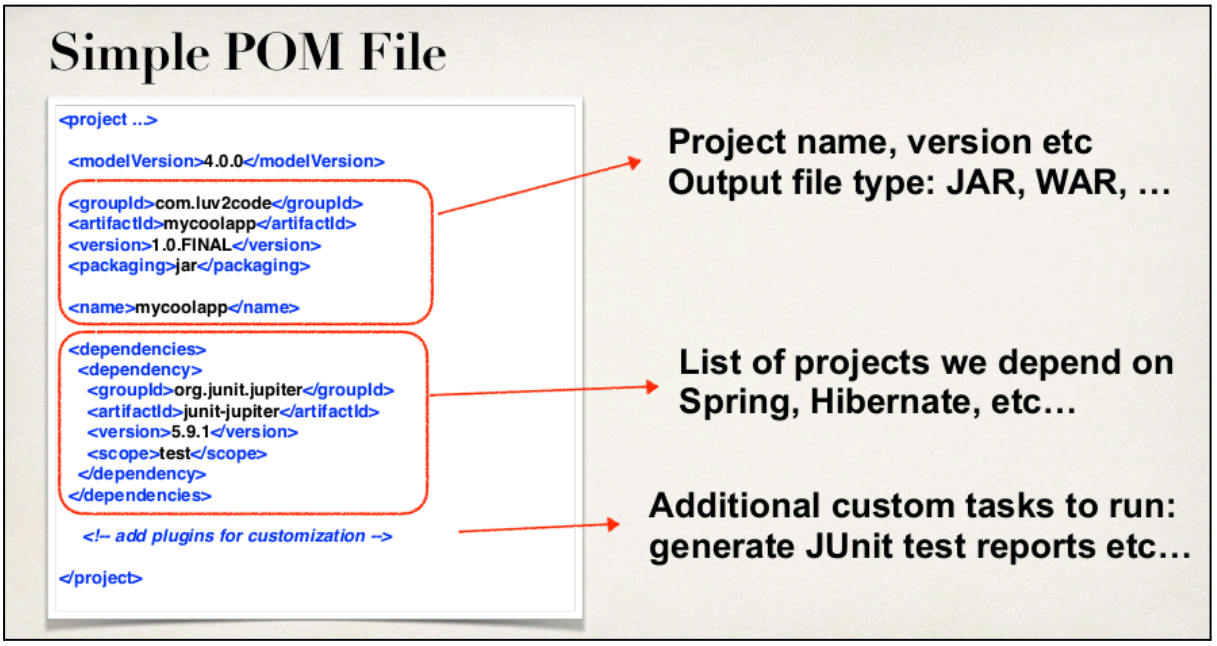
Maven is a powerful build automation and project management tool primarily used for Java projects. It simplifies the process of managing dependencies, building, testing, and packaging Java applications. Maven uses a project object model (POM) file to describe the structure of a project, including its dependencies, build settings, and plugins configuration.

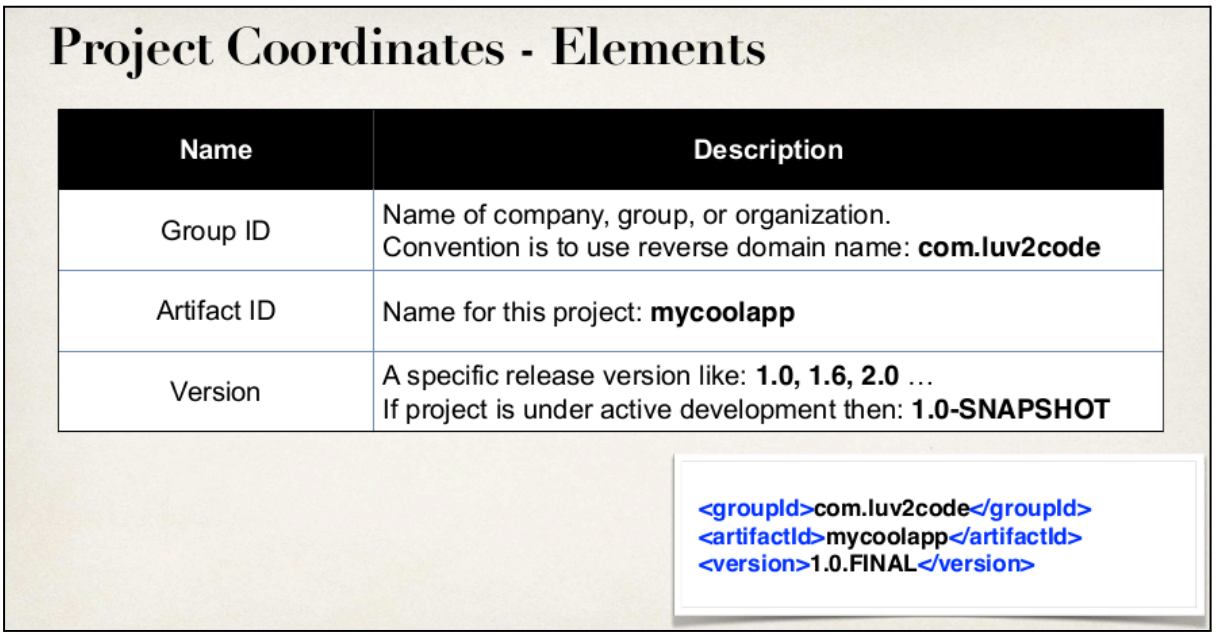




Maven Key Concepts

- POM File - pom.xml
- Project Coordinates

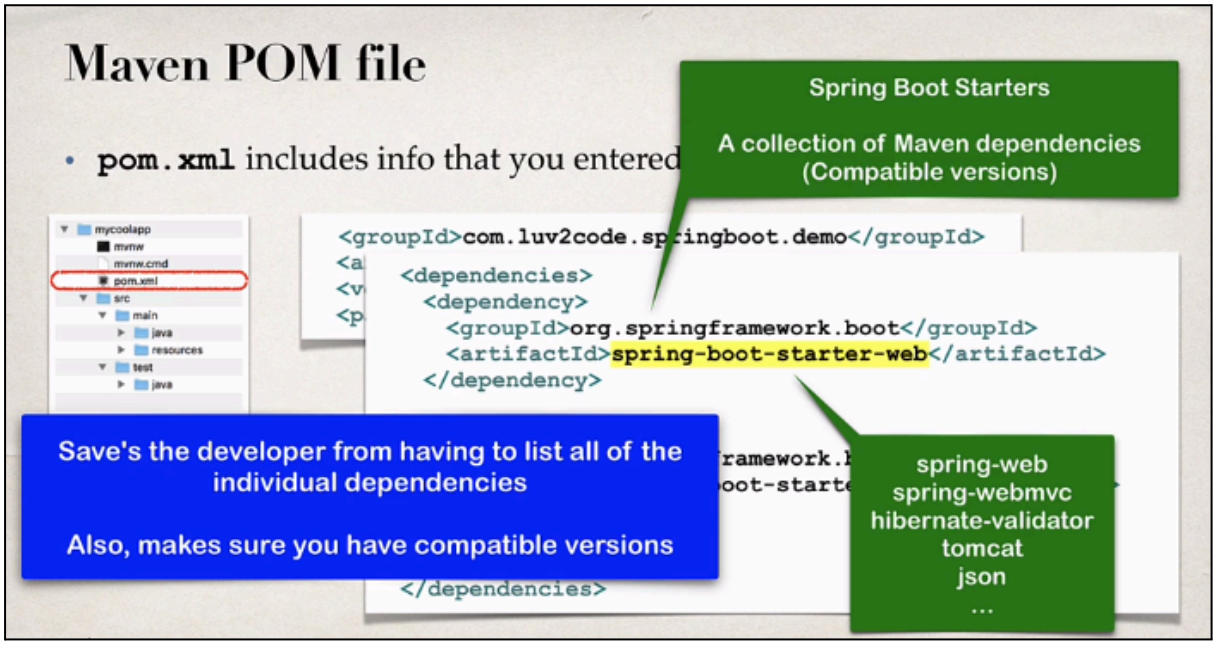


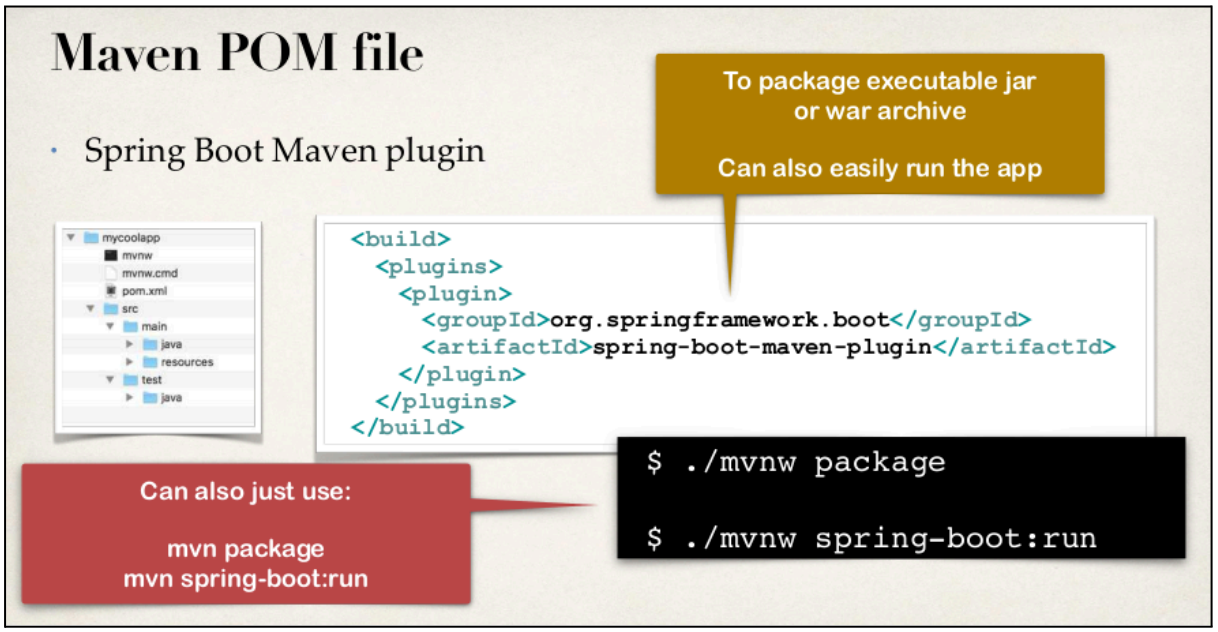


May see this referred to as: **GAV** - Group ID, Artifact ID and Version

A Maven wrapper file, often referred to as mvnw (Maven Wrapper), is a small shell script along with a JAR file that enables a Maven project to be built and executed without needing to have Maven installed on the system.

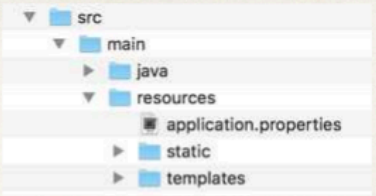
Spring Boot Project Structure





Application Properties

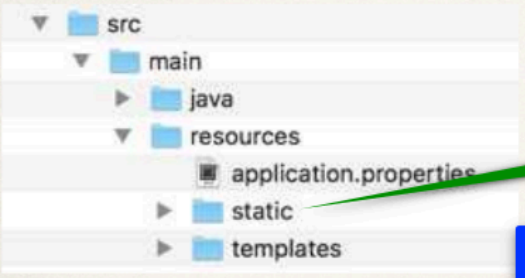
- Read data from: **application.properties**




```
# configure server port
server.port=8484

# configure my props
coach.name=Mickey Mouse
team.name=The Mouse Crew
```

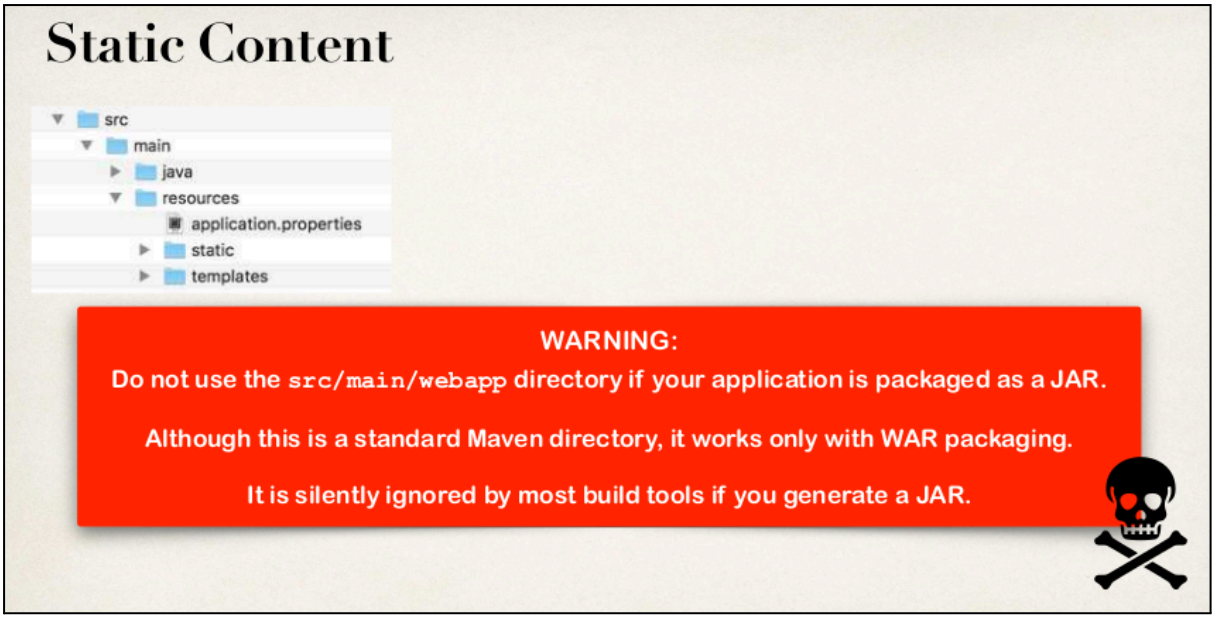
```
@RestController
public class FunRestController {

  @Value("${coach.name}")
  private String coachName;

  @Value("${team.name}")
  private String teamName;

  ...
}
```

Static Content



By default, Spring Boot will load static resources from "/static" directory

Examples of static resources
HTML files, CSS, JavaScript, images, etc ...

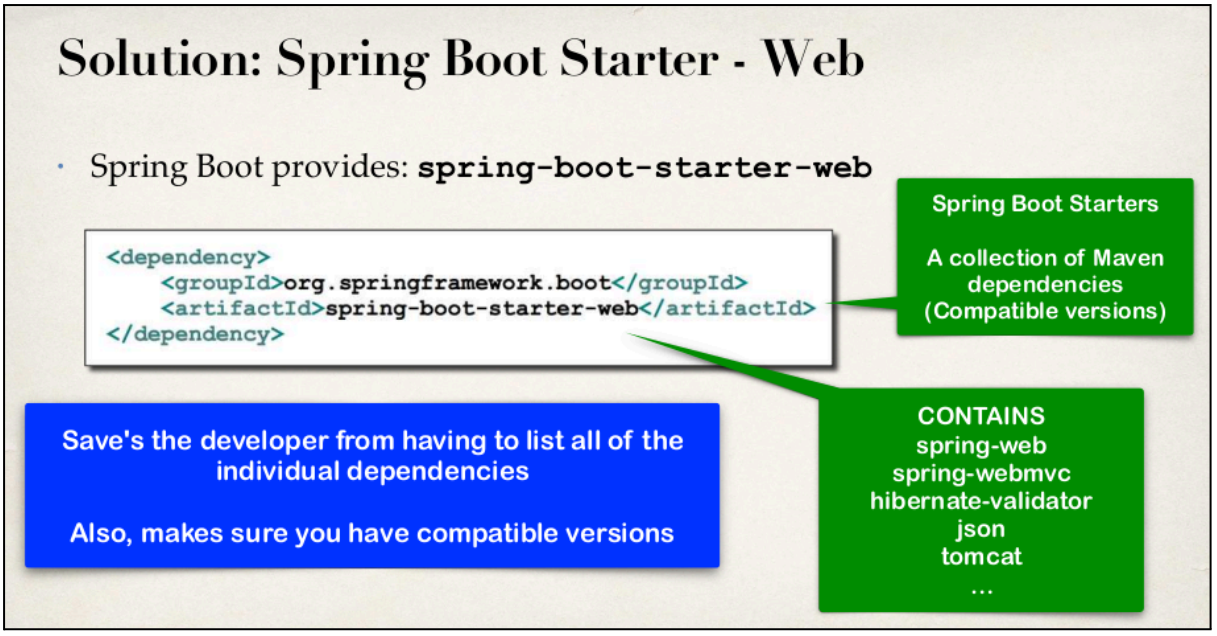


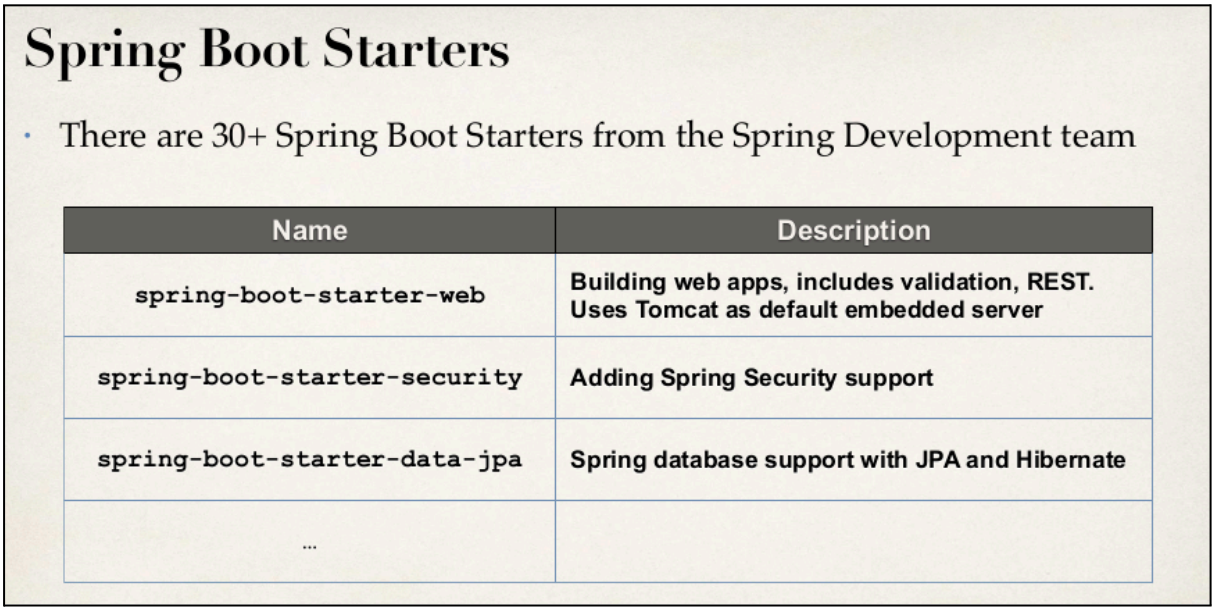
Spring Boot Starters

- A curated list of Maven dependencies
- A collection of dependencies grouped together
- Tested and verified by the Spring Development team
- Makes it much easier for the developer to get started with Spring
- Reduces the amount of Maven configuration

Spring MVC
For example, when building a Spring MVC app, you normally need

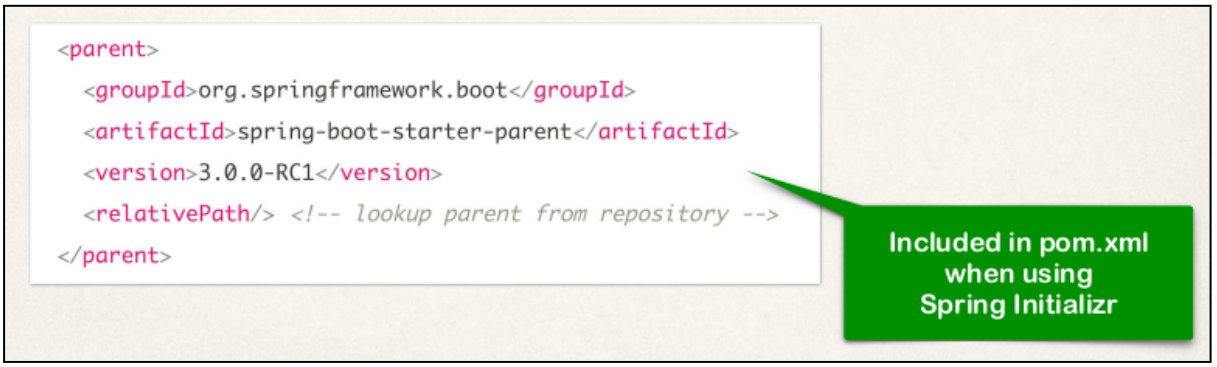
```
<!-- Spring support -->
<dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-webmvc</artifactId>
    <version>6.0.0-RC1</version>
</dependency>

<!-- Hibernate Validator -->
<dependency>
    <groupId>org.hibernate.validator</groupId>
    <artifactId>hibernate-validator</artifactId>
    <version>7.0.5.Final</version>
</dependency>

<!-- Web template: Thymeleaf -->
<dependency>
    <groupId>org.thymeleaf</groupId>
    <artifactId>thymeleaf</artifactId>
    <version>3.0.15.RELEASE</version>
</dependency>
```





Spring Boot Starter Parent

- Spring Boot provides a "Starter Parent"
- This is a special starter that provides Maven defaults



Benefits of the Spring Boot Starter Parent

- Default Maven configuration: Java version, UTF-encoding etc
- Dependency management
    - Use version on parent only
    - spring-boot-starter-* dependencies inherit version from parent
- Default configuration of Spring Boot plugin

The Problem

- When running Spring Boot applications
- If you make changes to your source code
- Then you have to manually restart your application

Solution: Spring Boot DevTools

- spring-boot-devtools to the rescue!
- Automatically restarts your application when code is updated
- Simply add the dependency to your POM file
- No need to write additional code :-)
- For IntelliJ, need to set additional configurations

IntelliJ Community Edition - DevTools

IntelliJ Community Edition does not support DevTools by default

- Select: Preferences > Build, Execution, Deployment > Compiler
    - Check box: Build project automatically

- Additional setting
    - Select: Preferences > Advanced Settings
    - Check box: Allow auto-make to ...

Spring Boot Actuator

Spring Boot Actuator is a feature of Spring Boot that provides built-in monitoring, metrics, and management capabilities for Spring Boot applications. It offers a set of production-ready endpoints and tools to monitor and manage your application at runtime.

/actuator/health : Provides information about the health status of the application. It typically includes details about the database connectivity, disk space, and other health indicators. The response is usually in JSON format.

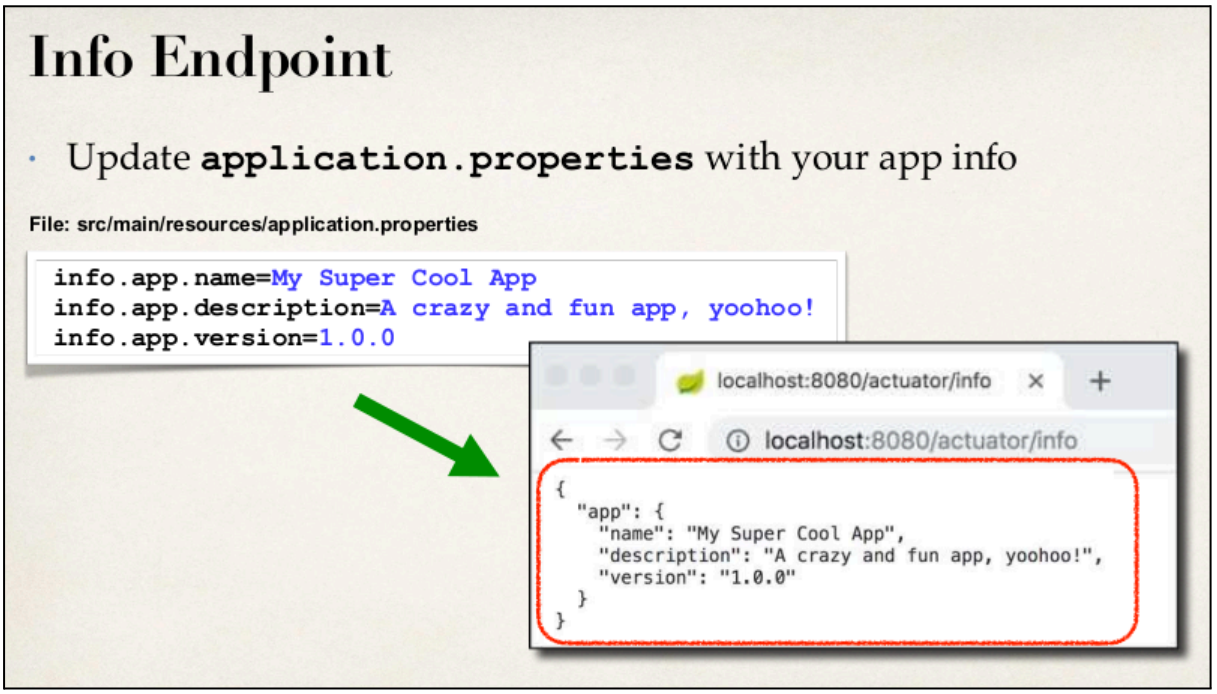
/actuator/info : Displays arbitrary application information. This endpoint can be customized to include additional metadata about the application, such as version information, build details, or environment-specific configuration.

/actuator/metrics : Exposes metrics about the application's behavior and performance. It includes JVM memory usage, garbage collection statistics, HTTP request metrics, and other custom metrics. Metrics can be retrieved in JSON format.

/actuator/env : Displays details about the application's environment, including active profiles, property sources, and configuration properties.

/actuator/beans : Displays a list of all Spring beans in the application context, including their names, types, and dependencies.

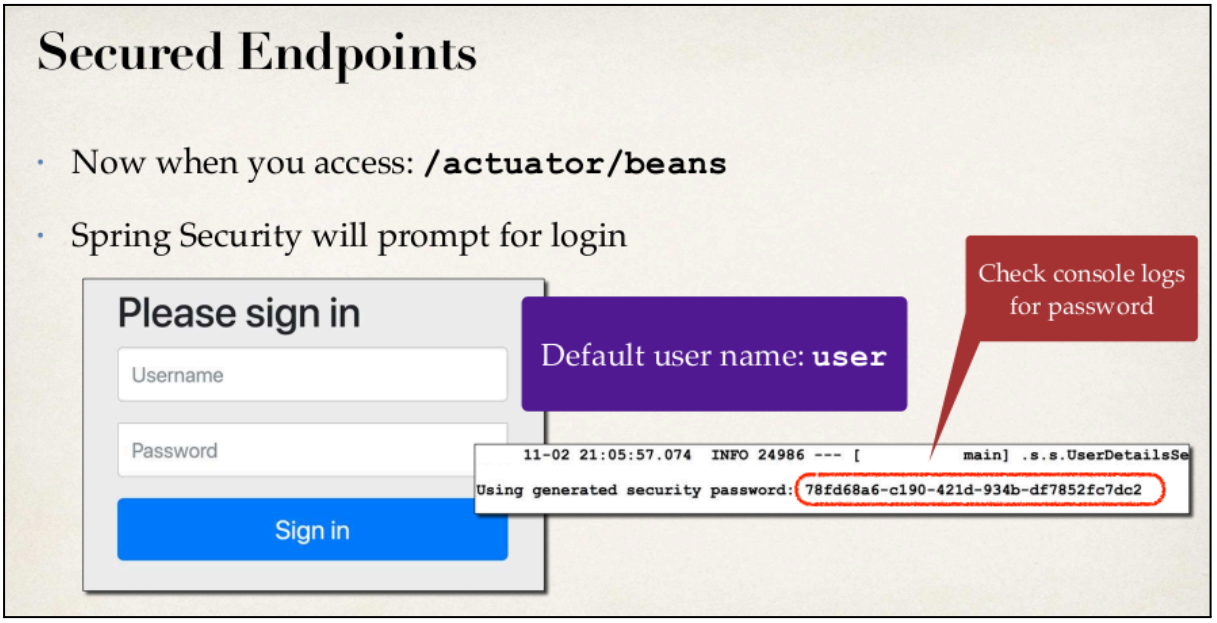
/actuator/loggers : Allows viewing and modifying the application's logging configuration. It displays a list of loggers and their current log levels, and allows changing the log levels at runtime.



Spring Boot Actuator Security

What about Security?

- You may NOT want to expose all of this information
- Add Spring Security to project and endpoints are secured



Customizing Spring Security

- You can customize Spring Security for Spring Boot Actuator
- Use a database for roles, encrypted passwords etc …
- We will cover details of Spring Security later in the course

Excluding Endpoints

- To exclude **/health**

File: src/main/resources/application.properties

```
# Exclude individual endpoints with a comma-delimited list
#
management.endpoints.web.exposure.exclude=health
```

Command Line Demo

Running from the Command-Line

- Two options for running the app
- Option 1: Use **java -jar**
- Option 2: Use Spring Boot Maven plugin
  - **mvnw spring-boot:run**

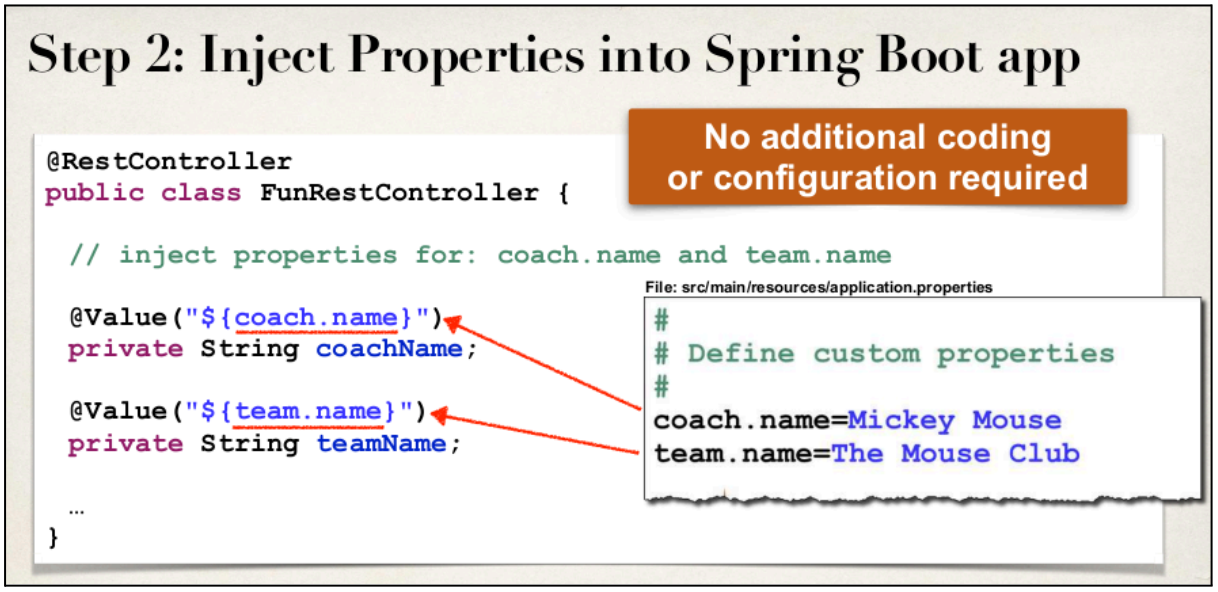
Development Process

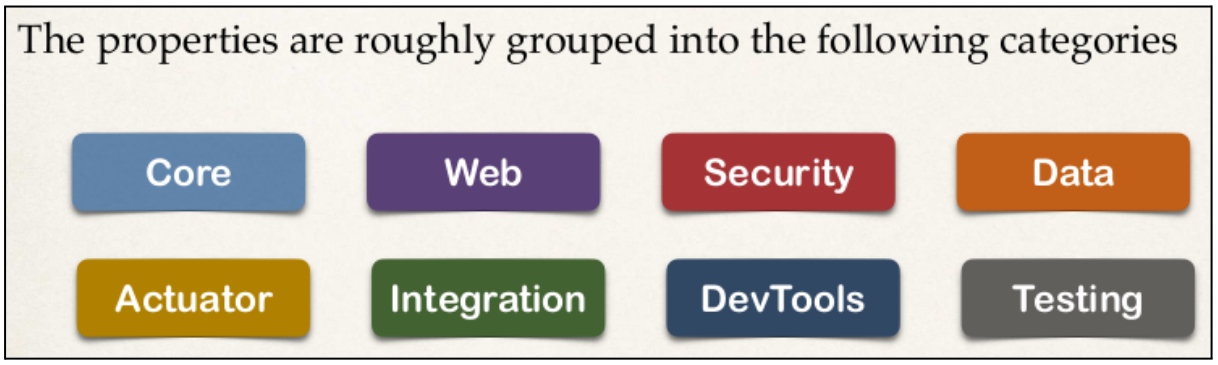
1. Exit the IDE
2. Package the app using (build jar file) - mvnw package
   (find jar file name - ls *.jar)
3. Run app using (in app target directory) - java -jar myapp.jar

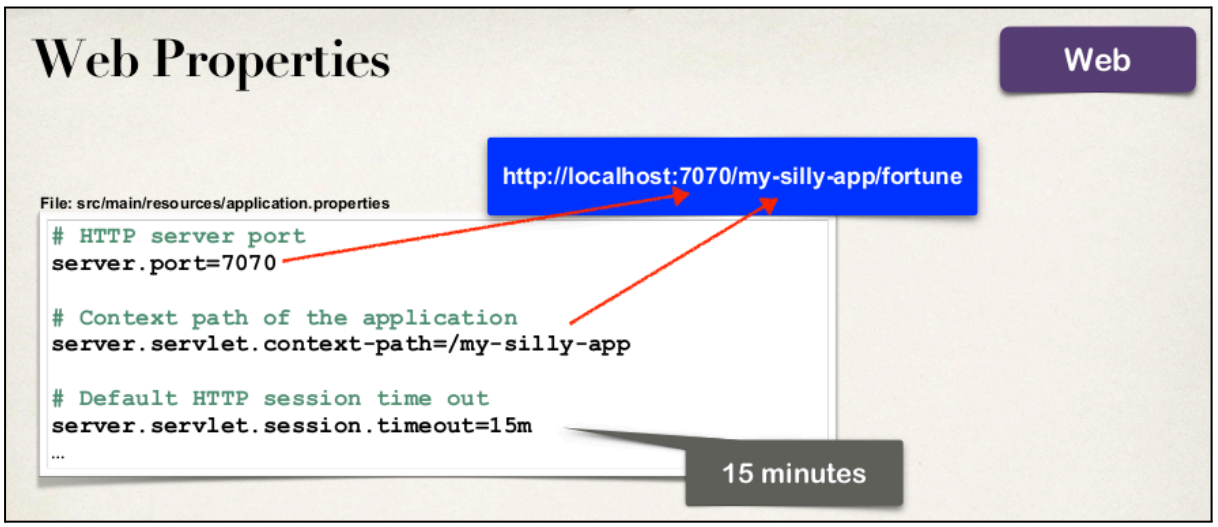
or

4. Run app using Spring Boot Maven plugin - mvnw spring-boot:run

Spring Boot Custom Application Properties







Actuator Properties

Actuator

File: src/main/resources/application.properties

```
# Endpoints to include by name or wildcard
management.endpoints.web.exposure.include=*

# Endpoints to exclude by name or wildcard
management.endpoints.web.exposure.exclude=beans,mapping

# Base path for actuator endpoints
management.endpoints.web.base-path=/actuator
...
```

http://localhost:7070/actuator/health

Security Properties

Security

File: src/main/resources/application.properties

```
# Default user name
spring.security.user.name=admin

# Password for default user
spring.security.user.password=topsecret
...
```

Data Properties

Data

File: src/main/resources/application.properties

```
# JDBC URL of the database
spring.datasource.url=jdbc:mysql://localhost:3306/ecommerce

# Login username of the database
spring.datasource.username=scott

# Login password of the database
spring.datasource.password=tiger
...
```

More on this in later videos

Spring Boot-3 Core

Inversion of Control

IoC is a principle in software engineering where the control of object creation and lifecycle management is inverted from the application code to a container or framework.

In the Spring Framework, the term "Spring container" refers to the core component responsible for managing the lifecycle of application objects (beans) and their dependencies.

The Spring container comes in different implementations, primarily:

- BeanFactory: This is the simplest container providing basic support for dependency injection. It is typically suitable for applications with lightweight configuration needs.

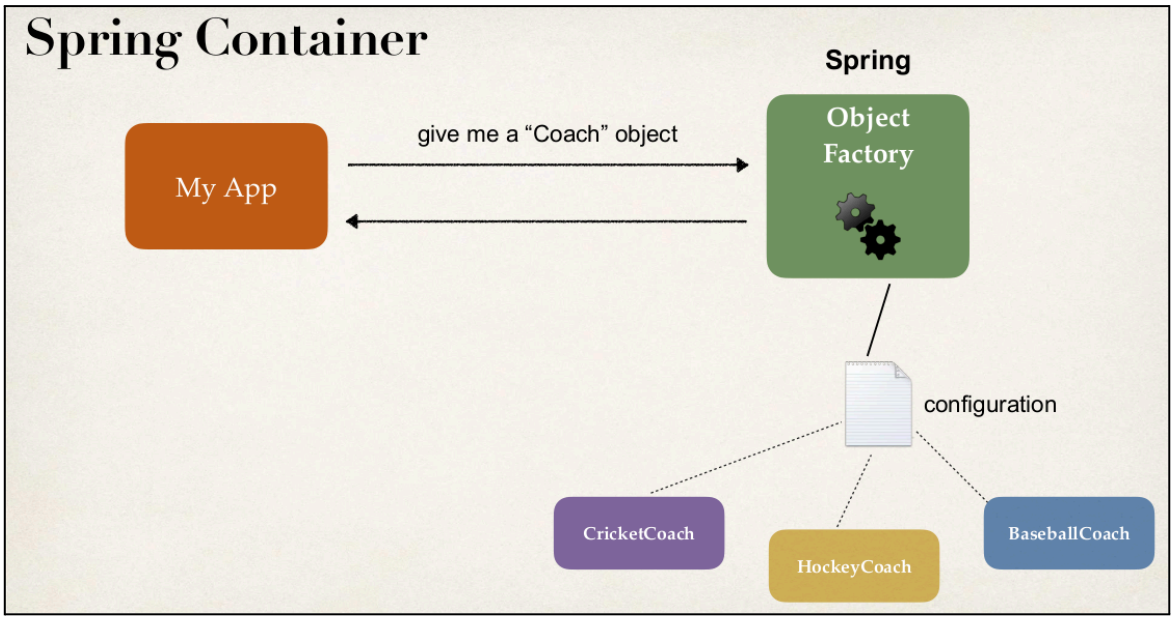
- ApplicationContext: This is a more feature-rich container that extends the functionality of the BeanFactory. It provides additional features such as internationalization support, event propagation, application-layer-specific contexts (e.g., WebApplicationContext for web applications), and more.

The Spring container plays a crucial role in enabling the Inversion of Control (IoC) and Dependency Injection (DI) principles within the Spring Framework, allowing developers to build flexible, modular, and maintainable applications.

Spring Container

Primary functions

- Create and manage objects (Inversion of Control)
- Inject object dependencies (Dependency Injection)



Configuring Spring Container

- XML configuration file (legacy - older method) ❌
- Java Annotations (modern) ✅
- Java Source Code (modern) ✅

Dependency Injection

Dependency Injection (DI) is a design pattern and a fundamental concept in software engineering. It's a technique where one object supplies the dependencies of another object, rather than the object itself creating those dependencies.

In Spring, Dependency Injection (DI) is a core principle and is widely used for managing dependencies between Spring-managed beans.

Here's a breakdown of Dependency Injection:

Dependency:
A dependency is an object that another object depends on to perform its work. For example, a UserService class might depend on a UserRepository to fetch user data from a database.

Injection:
Injection refers to the passing of a dependency (typically as a parameter) to the dependent object (usually through a constructor, setter method, or field). The dependent object doesn't create the dependency itself; instead, it receives it from an external source.

Injection Types

- There are multiple types of injection with Spring
- Constructor Injection
- Setter Injection

Constructor Injection

- Use this when you have required dependencies
- Generally recommended by the spring.io development team as first choice

Setter Injection

- Use this when you have optional dependencies
- If dependency is not provided, your app can provide reasonable default logic

Autowiring

Spring Autowiring is a feature that allows Spring to automatically inject dependencies into Spring-managed beans without explicit configuration. Autowiring eliminates the need for manual wiring of beans through XML configuration or Java annotations, reducing boilerplate code and making the configuration more concise.

```java
@Service
public class UserService {
    private final UserRepository userRepository;

    @Autowired
    public UserService(UserRepository userRepository) {
        this.userRepository = userRepository;
    }

    // Other methods
}
```

Replace this with @RequiredArgConstructor

```java
@Service
@RequiredArgsConstructor
public class UserService {
    private final UserRepository userRepository;

    // Other methods
}
```

@Component annotation

- @Component marks the class as a Spring Bean
- A Spring Bean is just a regular Java class that is managed by Spring
- @Component also makes the bean available for dependency injection

Spring Bean

In the Spring Framework, a "bean" is simply an object that is managed by the Spring IoC (Inversion of Control) container. Beans are the building blocks of a Spring application, and they represent the various components and services that make up the application.

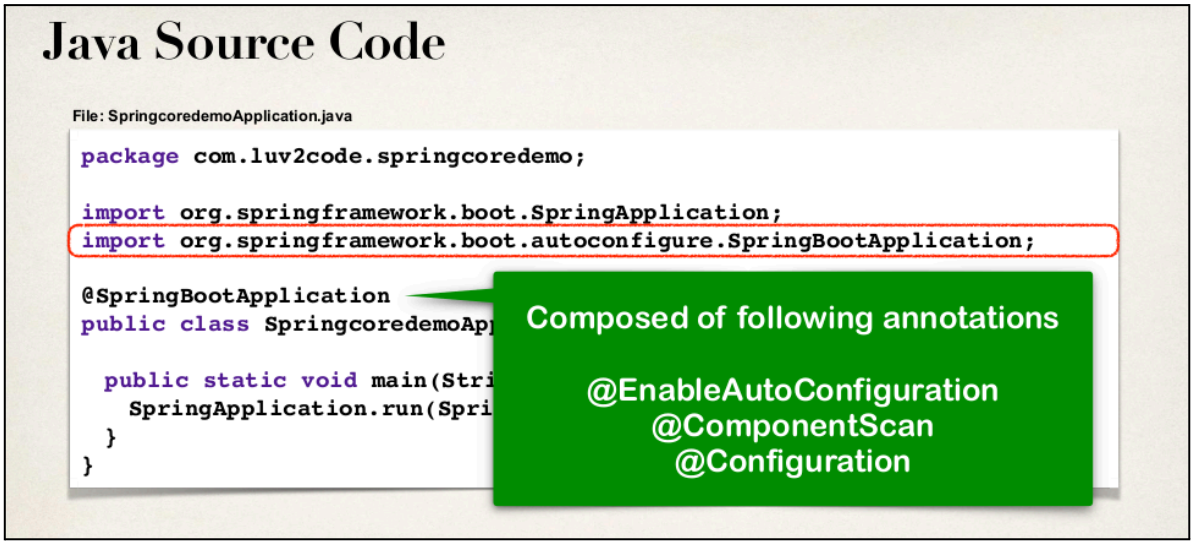
Spring for Enterprise applications

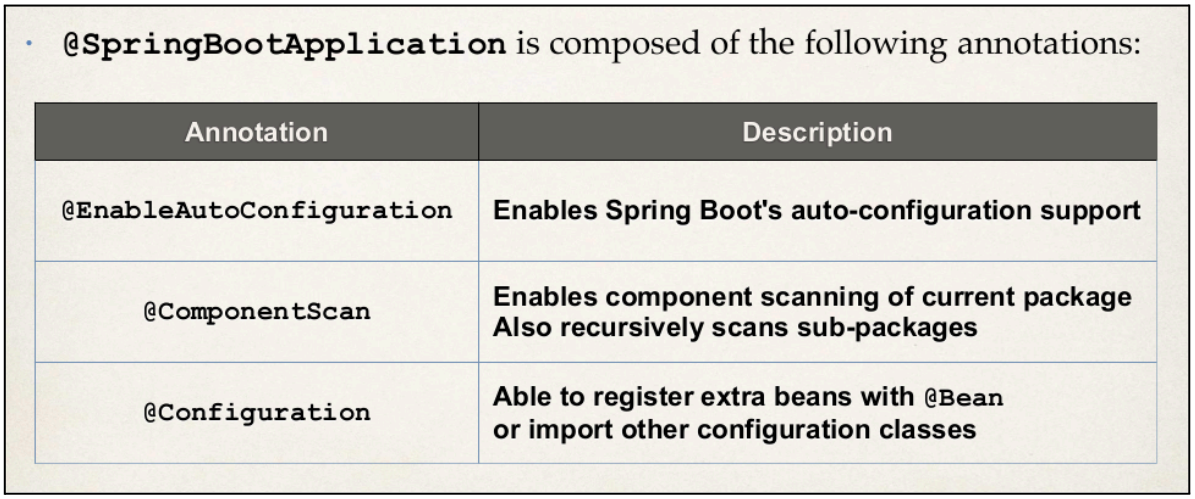
- Spring is targeted for enterprise, real-time / real-world applications
- Spring provides features such as
- Database access and Transactions
- REST APIs and Web MVC
- Security
- etc…

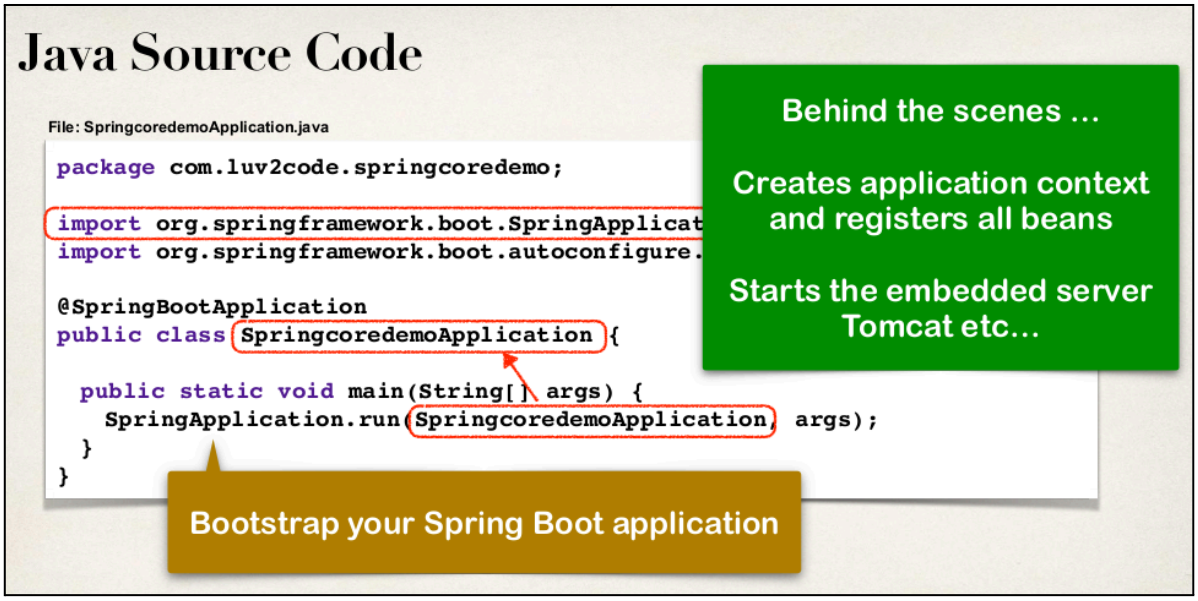
Component Scan

Scanning for Component Classes

- Spring will scan your Java classes for special annotations
    - @Component, etc …
- Automatically register the beans in the Spring container

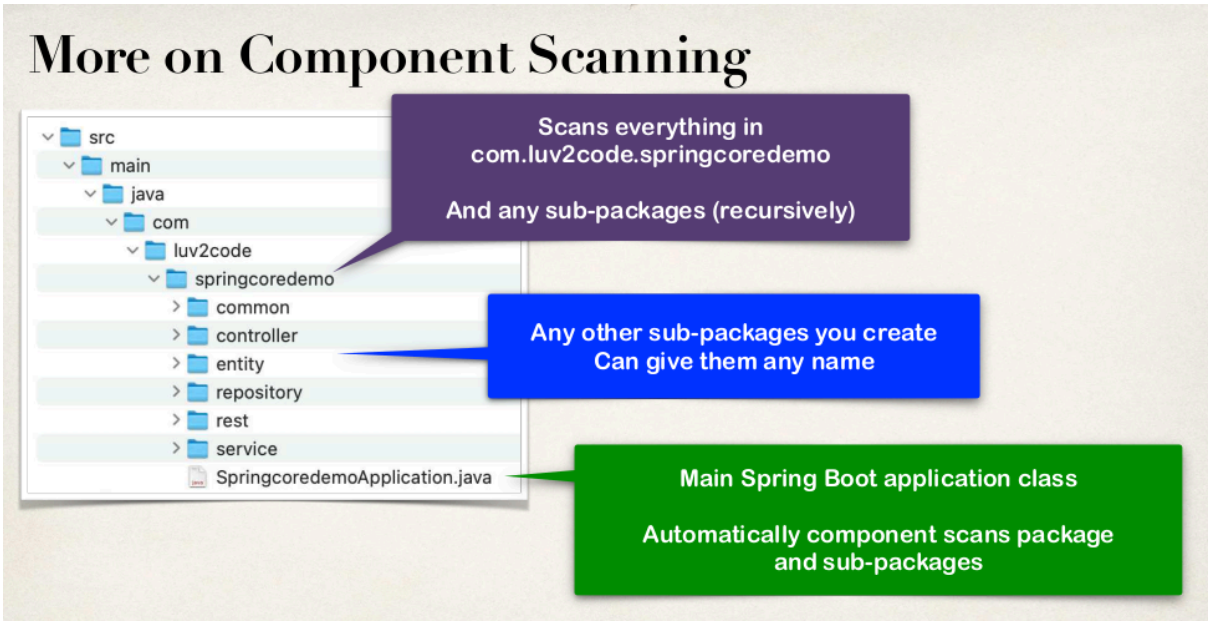


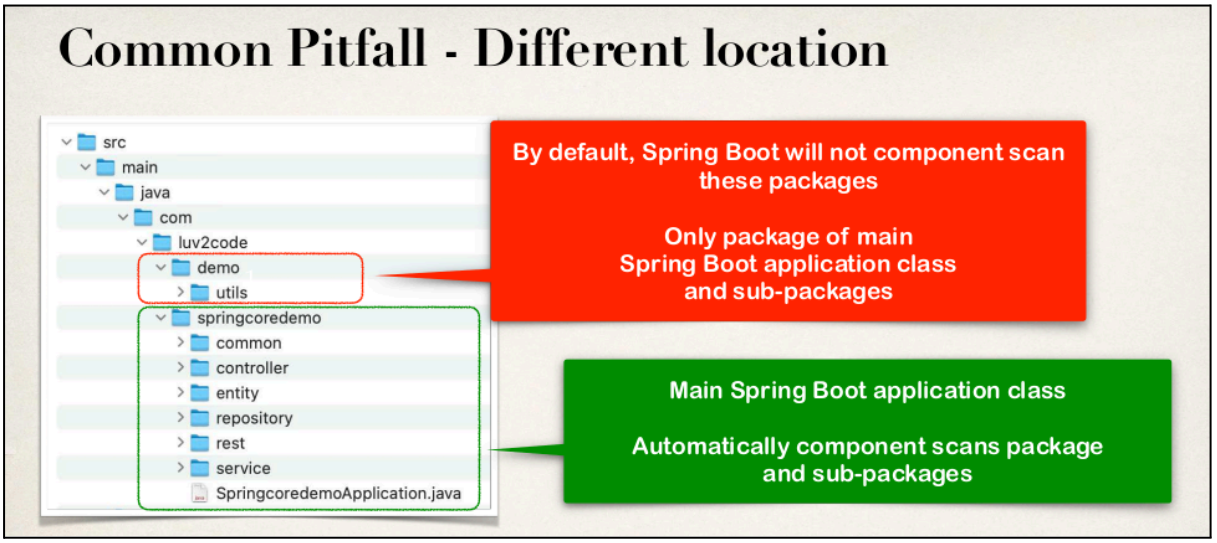


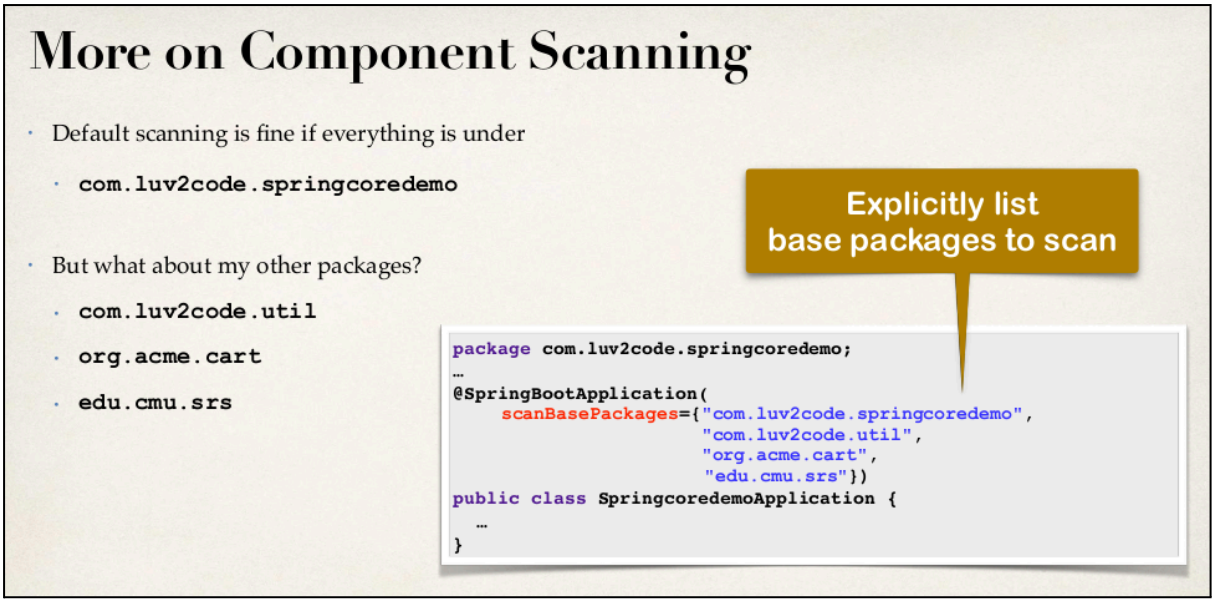


More on Component Scanning

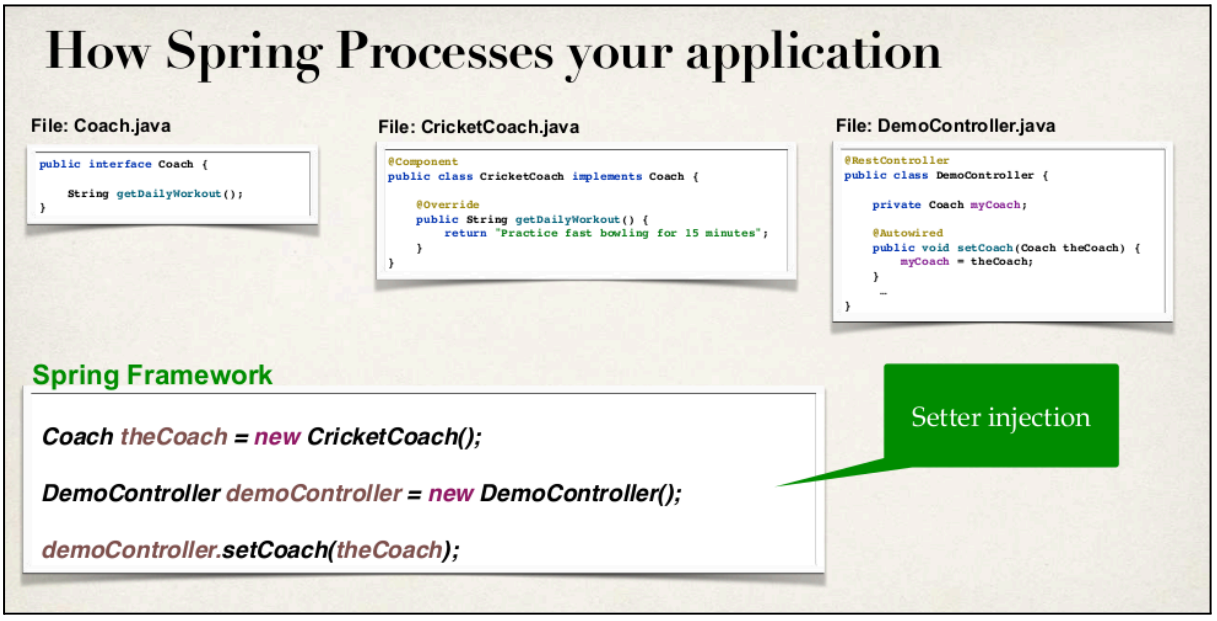
- By default, Spring Boot starts component scanning
    - From same package as your main Spring Boot application
    - Also scans sub-packages recursively
- This implicitly defines a base search package
    - Allows you to leverage default component scanning
    - No need to explicitly reference the base package name







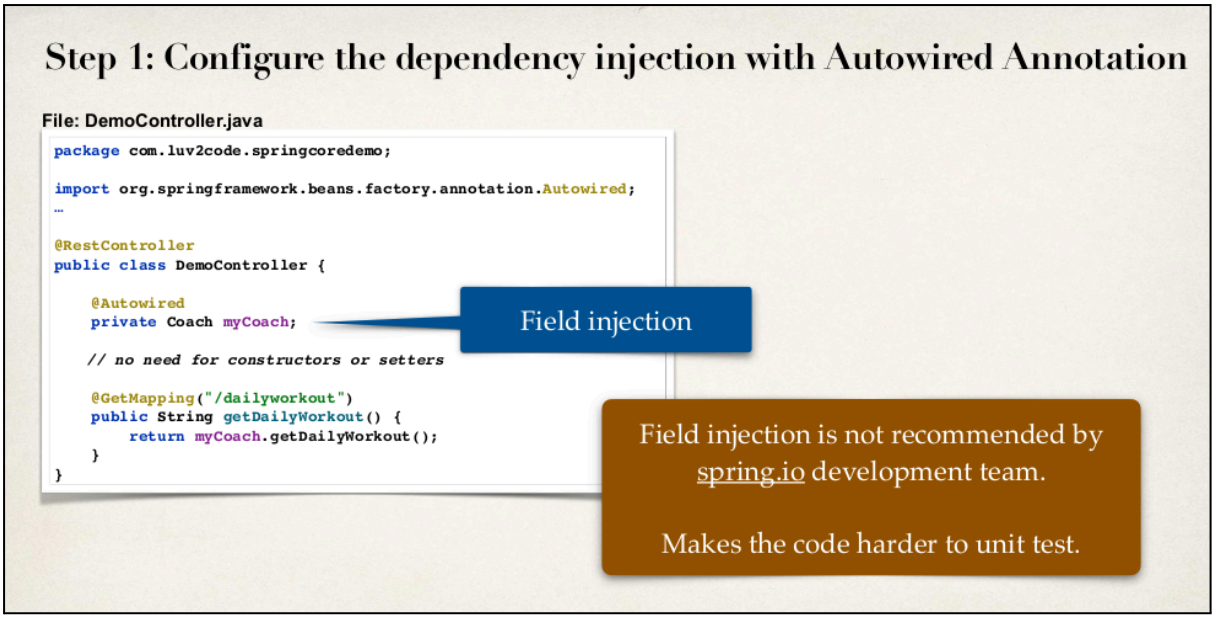
Setter Method injection



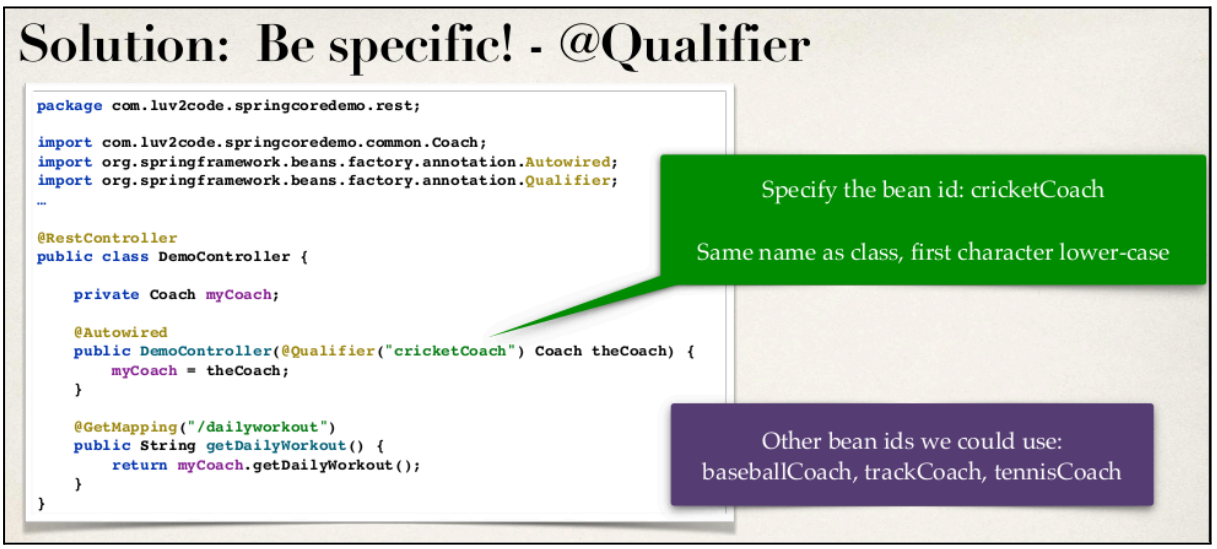
Field Injection with Annotations and Autowiring

Field Injection … no longer cool

- In the early days, field injection was popular on Spring projects
- In recent years, it has fallen out of favor
- In general, it makes the code harder to unit test
- As a result, the spring.io team does not recommend field injection
- However, you will still see it being used on legacy projects



Annotation Autowiring and Qualifiers



@Qualifier is an annotation used in Spring Framework for resolving autowiring conflicts when there are multiple beans of the same type available in the Spring container.

In a Spring application context, when you have multiple beans of the same type (for example, multiple implementations of the same interface), Spring might not know which bean to inject when using `@Autowired`. In such cases, you can use `@Qualifier` along with `@Autowired` to specify the exact bean to be injected.

1. Annotate the target bean definitions with `@Qualifier` and provide a unique name for each bean.
2. Use `@Autowired` along with `@Qualifier` specifying the bean's qualifier name to specify which bean to inject.

@Primary

@Primary is another annotation in the Spring Framework used to resolve bean injection ambiguity, similar to @Qualifier. However, it's used in a slightly different context.

When multiple beans of the same type are available in the Spring application context and you want one of them to be the default candidate for autowiring, you can annotate that bean with @Primary. This means that if Spring cannot determine which bean to autowire based on @Qualifier or other means, it will default to the bean marked with @Primary.

@Primary - Only one
When using @Primary, can have only one for multiple implementations
If you mark multiple classes with @Primary … umm, we have a problem

```
Unsatisfied dependency expressed through constructor parameter 0:
No qualifying bean of type 'com.luv2code.springcoredemo.common.Coach' available:
more than one 'primary' bean found among candidates:
[baseballCoach, cricketCoach, tennisCoach, trackCoach]
```

Mixing @Primary and @Qualifier

- If you mix @Primary and @Qualifier
- @Qualifier has higher priority

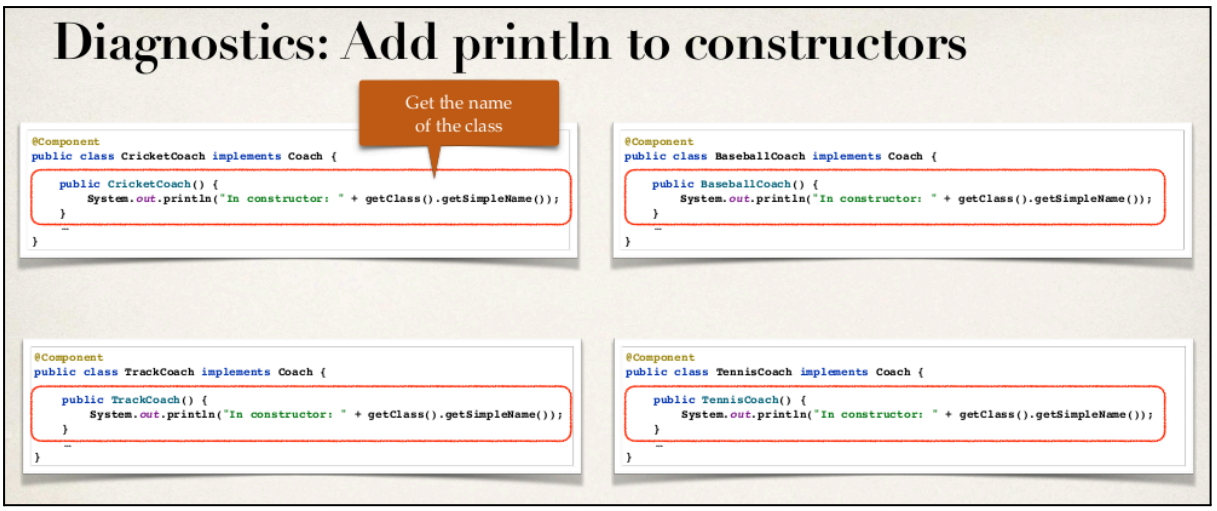
Lazy Initialization

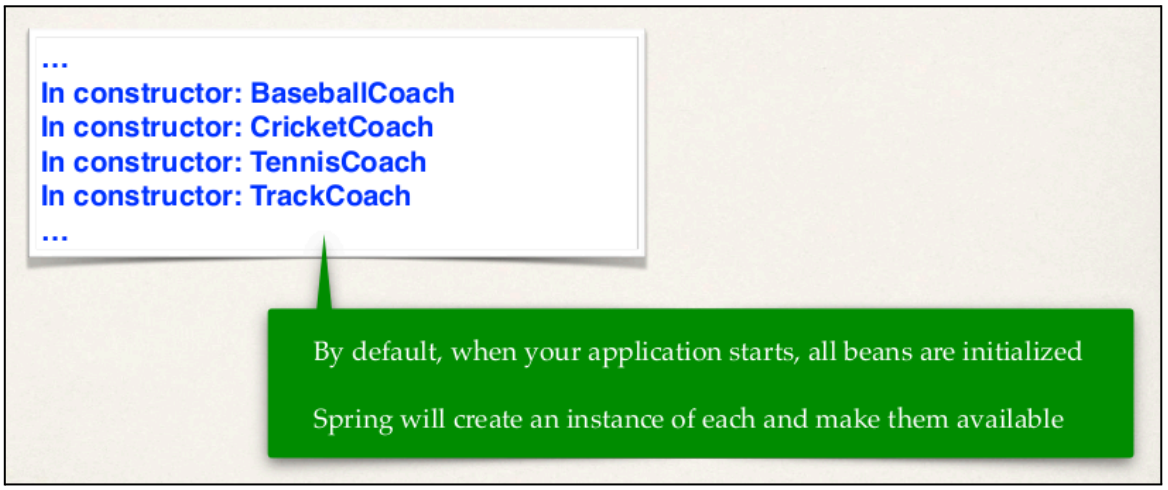
In Spring Framework, lazy initialization refers to the postponement of bean initialization until the bean is actually requested. This can help improve the startup time and memory consumption of an application by deferring the creation of beans until they are needed.

By default, Spring initializes all beans eagerly, meaning that they are instantiated as soon as the application context is created. However, by using lazy initialization, you can defer the instantiation of beans until they are actually needed, potentially improving the performance of your Spring application.

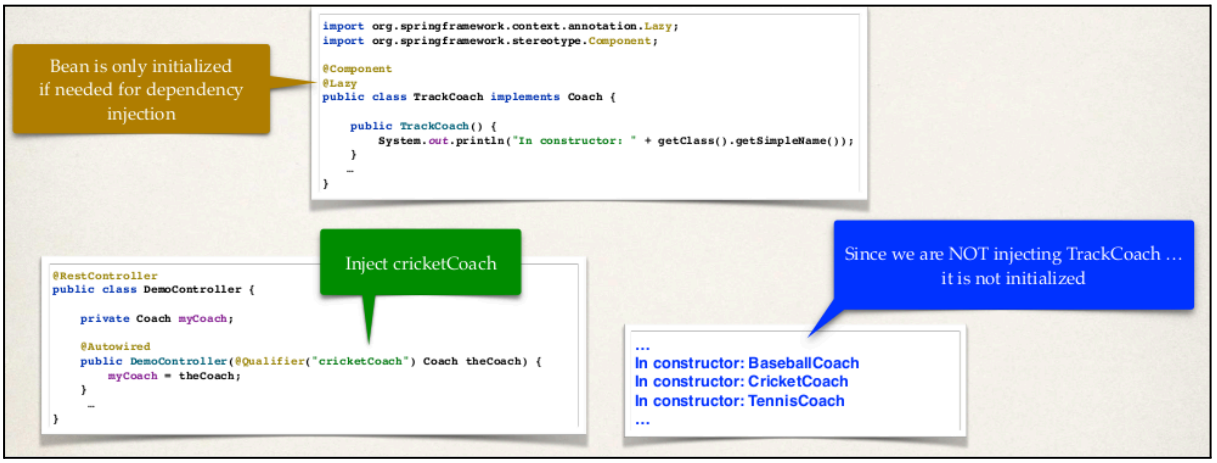
Initialization

- By default, when your application starts, all beans are initialized
    - @Component, etc …
- Spring will create an instance of each and make them available

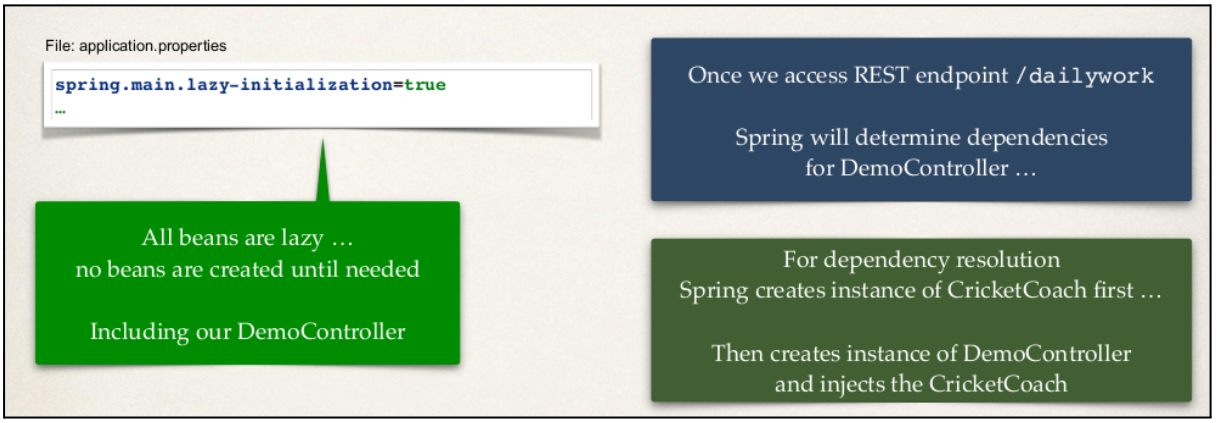




Lazy Initialization with @Lazy



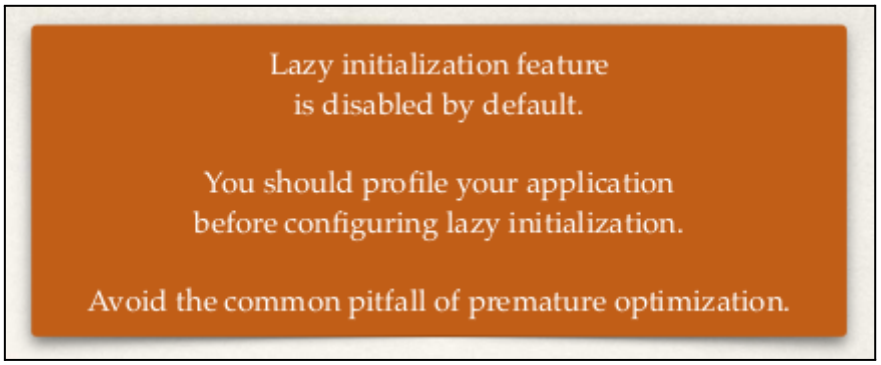
Lazy Initialization - Global configuration



Using Component Scanning: If you're using component scanning, you can enable lazy initialization globally in your Spring configuration.

java Copy code

```java
@Configuration
@ComponentScan(basePackages = "com.example", lazyInit = true)
public class AppConfig {
    // Configuration
}
```



Advantages

- Only create objects as needed
- May help with faster startup time if you have large number of components

Disadvantages

- If you have web related components like @RestController, not created until requested
- May not discover configuration issues until too late
- Need to make sure you have enough memory for all beans once created

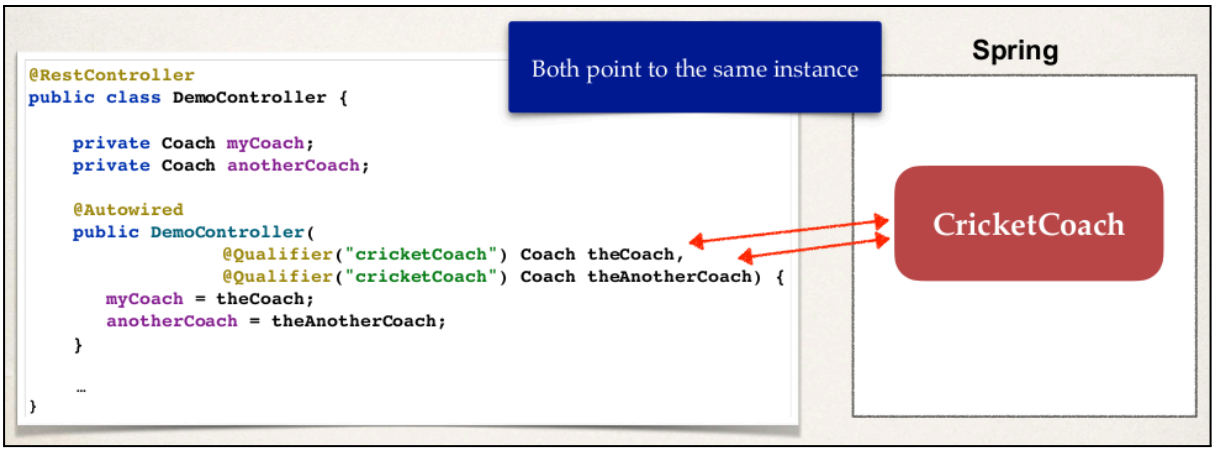
Bean Scopes

Default Scope is Singleton

Refresher: What Is a Singleton?

- Spring Container creates only one instance of the bean, by default
- It is cached in memory
- All dependency injections for the bean
- Will reference the SAME bean

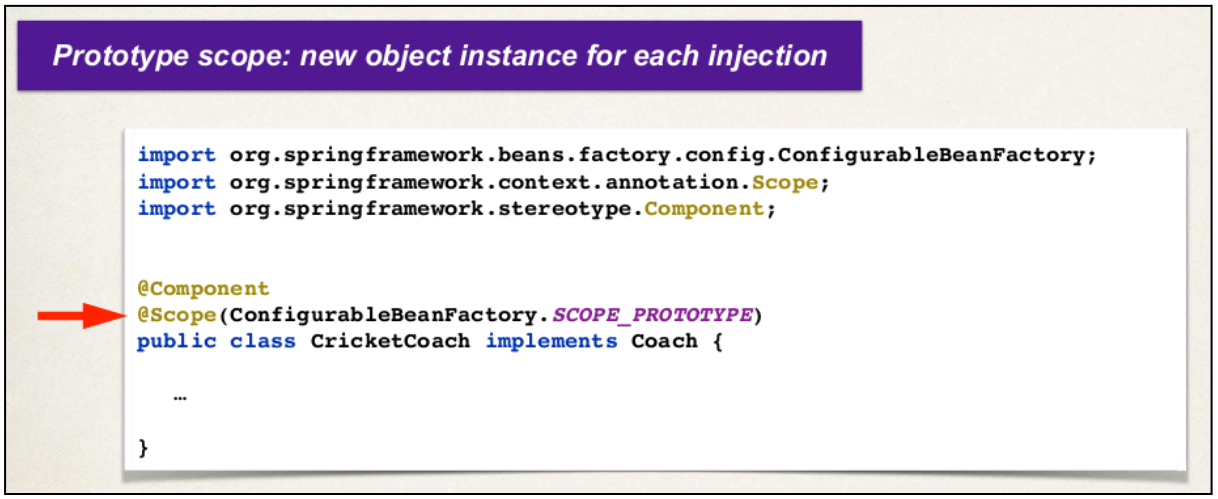
What is Singleton?

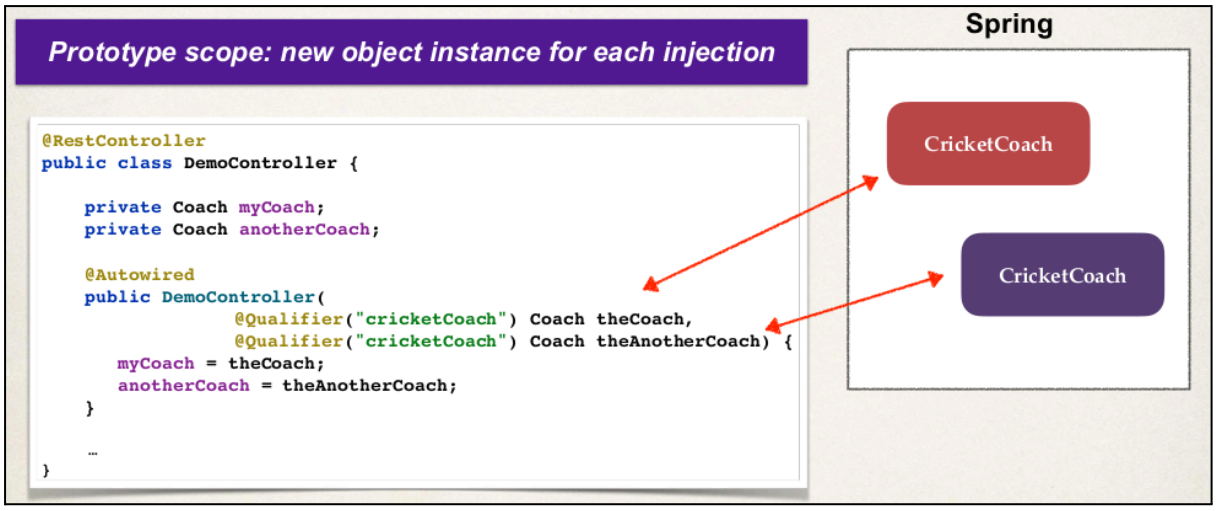


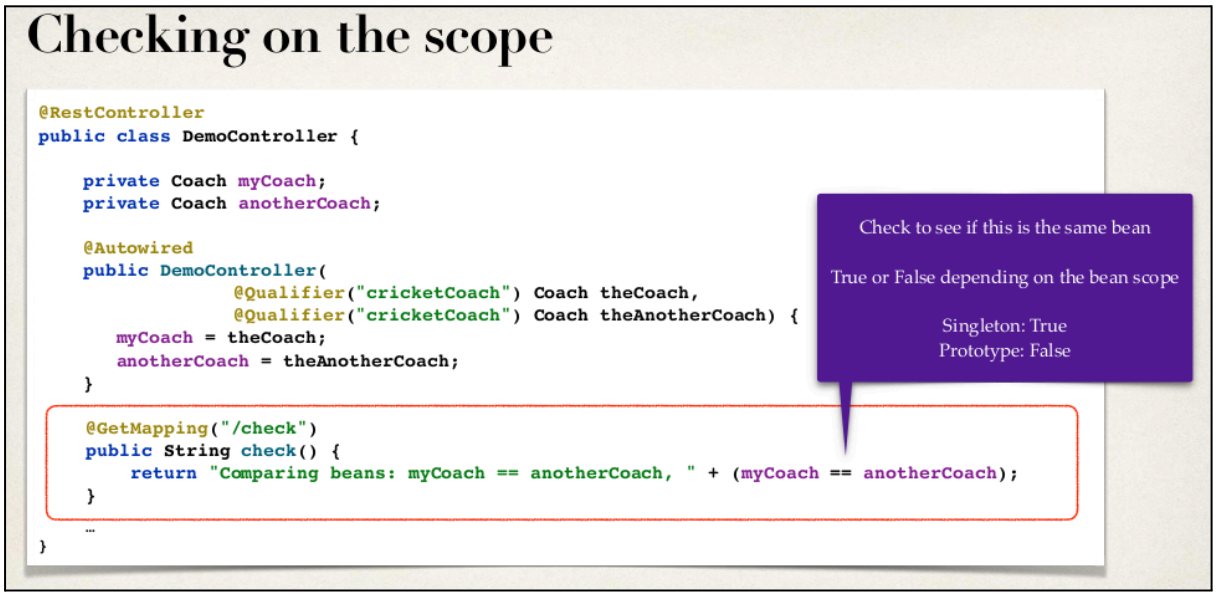
Bean Scopes

Scope	Description
singleton	Create a single shared instance of the bean. Default scope.
prototype	Creates a new bean instance for each container request.
request	Scoped to an HTTP web request. Only used for web apps.
session	Scoped to an HTTP web session. Only used for web apps.
global-session	Scoped to a global HTTP web session. Only used for web apps.

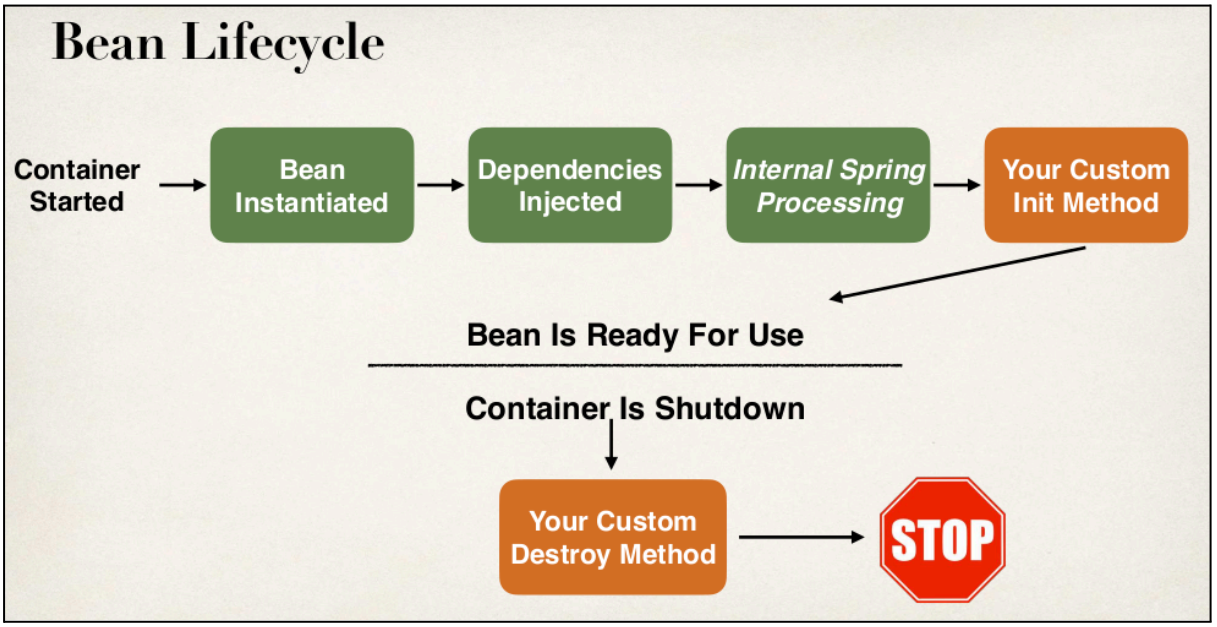
Prototype Scope Example

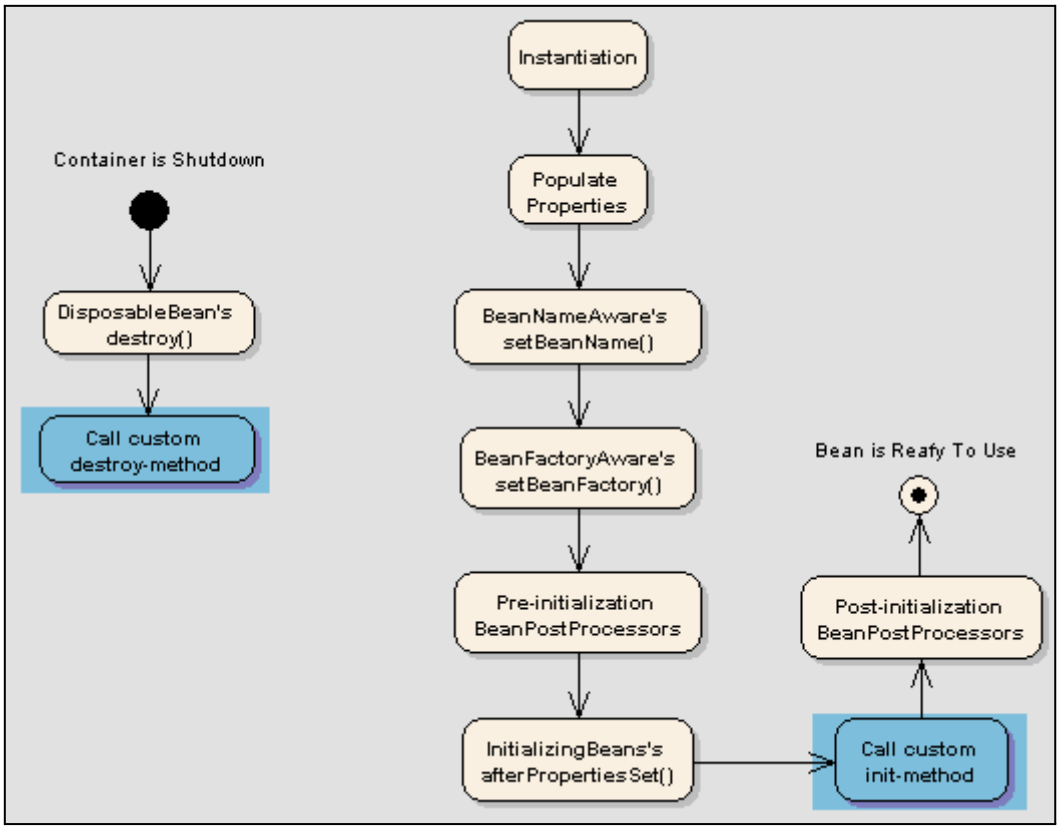






Bean Lifecycle Methods - Annotations





Bean Lifecycle Methods / Hooks

- You can add custom code during bean initialization
  - Calling custom business logic methods
  - Setting up handles to resources (db, sockets, file etc)
- You can add custom code during bean destruction
  - Calling custom business logic method
  - Clean up handles to resources (db, sockets, files etc)

Init: method configuration

```
@Component
public class CricketCoach implements Coach {

    public CricketCoach() {
        System.out.println("In constructor: " + getClass().getSimpleName());
    }

    @PostConstruct
    public void doMyStartupStuff() {
        System.out.println("In doMyStartupStuff(): " + getClass().getSimpleName());
    }

    …
}
```

Destroy: method configuration

```
@Component
public class CricketCoach implements Coach {

    public CricketCoach() {
        System.out.println("In constructor: " + getClass().getSimpleName());
    }

    @PostConstruct
    public void doMyStartupStuff() {
        System.out.println("In doMyStartupStuff(): " + getClass().getSimpleName());
    }

    @PreDestroy
    public void doMyCleanupStuff() {
        System.out.println("In doMyCleanupStuff(): " + getClass().getSimpleName());
    }
    …
}
```

Special Note about Prototype Scope - Destroy Lifecycle Method and Lazy Init

Prototype Beans and Destroy Lifecycle
There is a subtle point you need to be aware of with "prototype" scoped beans.

For "prototype" scoped beans, Spring does not call the destroy method. Gasp!

In contrast to the other scopes, Spring does not manage the complete lifecycle of a prototype bean*: the container instantiates, configures, and otherwise assembles a prototype object, and hands it to the client, with no further record of that prototype instance.*

Thus, although initialization lifecycle callback methods are called on all objects regardless of scope, ***in the case of prototypes, configured destruction lifecycle callbacks are not called****. The client code must clean up prototype-scoped objects and release expensive resources that the prototype bean(s) are holding.*

Prototype Beans and Lazy Initialization
Prototype beans are lazy by default. There is no need to use the @Lazy annotation for prototype scopes beans.

==========

When a bean is defined with prototype scope (`@Scope("prototype")`), Spring doesn't manage the complete lifecycle of the prototype bean. Instead, Spring creates a new instance of the prototype bean each time it is requested, and it's the responsibility of the application to manage the lifecycle of the instance.

Here's a brief overview of the lifecycle of prototype scoped beans in Spring:

1.Bean Creation : When a bean with prototype scope is requested from the Spring container, a new instance of the bean is created.

2.Dependency Injection : If the prototype bean has any dependencies, Spring injects those dependencies into the newly created instance.

3.Use by Client : The client code or other beans in the application context use the prototype bean instance.

4.No Lifecycle Management by Spring : Unlike singleton scoped beans, Spring does not manage the lifecycle of prototype scoped beans. This means Spring doesn't handle destruction of prototype beans. It's the responsibility of the application to manage the lifecycle, including destruction, if necessary.

5.Garbage Collection : When there are no more references to the prototype bean instance, it becomes eligible for garbage collection.

In summary, prototype scoped beans in Spring are created anew each time they are requested, and the application is responsible for managing their lifecycle and resources.

Configuring Beans with Java Code

In Spring Framework, `@Configuration` and `@Bean` are annotations used to define beans and their dependencies in a Spring application context.

1. @Configuration :

- `@Configuration` is used to indicate that a class declares one or more `@Bean` methods and may be processed by the Spring container to generate bean definitions and service requests for those beans at runtime.
- It's typically used on classes that contain `@Bean` methods to provide configuration to the Spring IoC container.
- Classes annotated with `@Configuration` can contain methods annotated with `@Bean`, which define the beans to be managed by the Spring container.

2. @Bean :

- `@Bean` is used to indicate that a method produces a bean to be managed by the Spring container.
- It works in conjunction with `@Configuration` or component scanning to register beans within the Spring application context.
- The method annotated with `@Bean` serves as a factory for creating and configuring the bean instance.

```java
@Configuration
public class AppConfig {
    @Bean
    public MyBean myBean() {
        return new MyBean();
    }
}
```

Use case for @Bean

- Make an existing third-party class available to Spring framework
- You may not have access to the source code of third-party class
- However, you would like to use the third-party class as a Spring bean

---

Real-World Project Example

- Our project used Amazon Web Service (AWS) to store documents
- Amazon Simple Storage Service (Amazon S3)
- Amazon S3 is a cloud-based storage system
- can store PDF documents, images etc
- We wanted to use the AWS S3 client as a Spring bean in our app
- The AWS S3 client code is part of AWS SDK
- We can’t modify the AWS SDK source code
- We can’t just add @Component
- However, we can configure it as a Spring bean using @Bean

Configure AWS S3 Client using @Bean

```
package com.luv2code.springcoredemo.config;

…
import software.amazon.awssdk.auth.credentials.ProfileCredentialsProvider;
import software.amazon.awssdk.regions.Region;
import software.amazon.awssdk.services.s3.S3Client;

@Configuration
public class DocumentsConfig {

    @Bean
    public S3Client remoteClient() {

        // Create an S3 client to connect to AWS S3
        ProfileCredentialsProvider credentialsProvider = ProfileCredentialsProvider.create();
        Region region = Region.US_EAST_1;
        S3Client s3Client = S3Client.builder()
                .region(region)
                .credentialsProvider(credentialsProvider)
                .build();

        return s3Client;
    }
}
```

Store our document in S3

```
package com.luv2code.springcoredemo.services;

import software.amazon.awssdk.services.s3.S3Client;
…

@Component
public class DocumentsService {

    private S3Client s3Client;

    @Autowired
    public DocumentsService(S3Client theS3Client) {
        s3Client = theS3Client;
    }

    public void processDocument(Document theDocument) {

       // get the document input stream and file size …

       // Store document in AWS S3
       // Create a put request for the object
       PutObjectRequest putObjectRequest = PutObjectRequest.builder()
               .bucket(bucketName)
               .key(subDirectory + "/" + fileName)
               .acl(ObjectCannedACL.BUCKET_OWNER_FULL_CONTROL).build();

       // perform the putObject operation to AWS S3 ... using our autowired bean
       s3Client.putObject(putObjectRequest, RequestBody.fromInputStream(fileInputStream, contentLength));
    }
}
```

- Bean Method Name should be bean id

Hibernate / JPA Overview

Hibernate is an open-source, object-relational mapping (ORM) framework for Java environments. It provides a framework for mapping an object-oriented domain model to a traditional relational database. Hibernate automates the process of object-relational mapping, freeing developers from manual SQL handling and allowing them to focus more on business logic.

Benefits of Hibernate:

1. Simplified Data Access : Hibernate abstracts away most of the complexities involved in database interactions, allowing developers to focus on business logic rather than SQL queries.

2. Object-Relational Mapping (ORM) : Hibernate simplifies the mapping between Java classes and database tables, making it easier to work with object-oriented code while persisting data in relational databases.

3. Automatic CRUD Operations : Hibernate provides automatic CRUD (Create, Read, Update, Delete) operations, reducing the need for boilerplate code for database interactions.

4. Caching : Hibernate supports various levels of caching, improving performance by reducing database round-trips.

5. Transaction Management : Hibernate provides built-in transaction management, making it easy to work with database transactions in a consistent and reliable manner.

6. Database Portability : Hibernate abstracts database-specific SQL dialects, allowing applications to be more easily ported across different database vendors without significant code changes.

- Hibernate handles all of the low-level SQL
- Minimizes the amount of JDBC code you have to develop
- Hibernate provides the Object-to-Relational Mapping (ORM)

What is JPA?

Java Persistence API (JPA) is a Java specification for accessing, persisting, and managing data between Java objects and relational databases. It provides a standardized way for developers to work with data persistence in Java applications.

Benefits of JPA:

1. Standardization : JPA provides a standardized way to work with data persistence in Java applications, allowing developers to write portable and vendor-independent code.

2. Annotation-based Mapping : JPA supports annotation-based mapping between Java objects and database tables, making it easier to define the mapping without relying on XML configuration.

3. Entity Relationships : JPA supports various types of entity relationships such as One-to-One, One-to-Many, and Many-to-Many, making it easy to model complex data relationships in object-oriented applications.

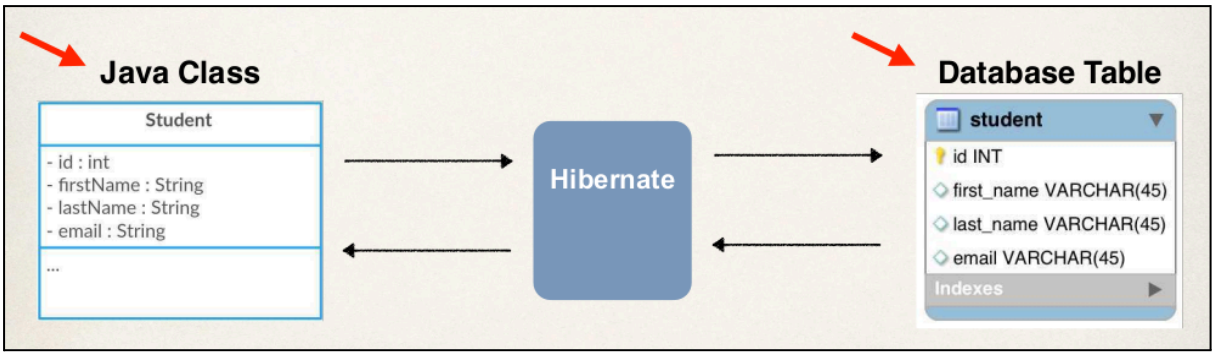
4. Query Language (JPQL) : JPA provides a powerful query language called JPQL (Java Persistence Query Language) for executing database queries in a database-independent manner.

5. Integration with Java EE : JPA is integrated with the Java EE platform, making it easy to use in Java EE applications.

- By having a standard API, you are not locked to vendor's implementation
- Maintain portable, flexible code by coding to JPA spec (interfaces)
- Can theoretically switch vendor implementations
    - For example, if Vendor ABC stops supporting their product
    - You could switch to Vendor XYZ without vendor lock in

Object-To-Relational Mapping (ORM)

- The developer defines mapping between Java class and database table



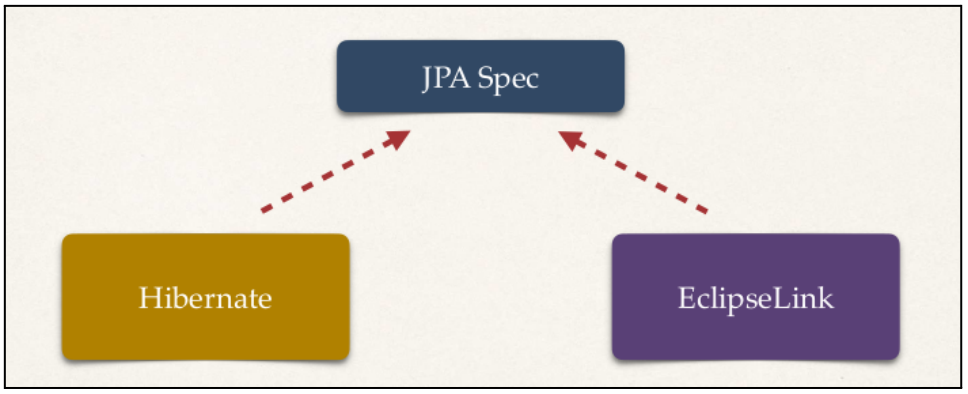
What is JPA?

- Jakarta Persistence API (JPA) … previously known as Java Persistence API

Standard API for Object-to-Relational-Mapping (ORM)

- Only a specification
- Defines a set of interfaces
- Requires an implementation to be usable

JPA - Vendor Implementations



Setting Up Spring Boot Project

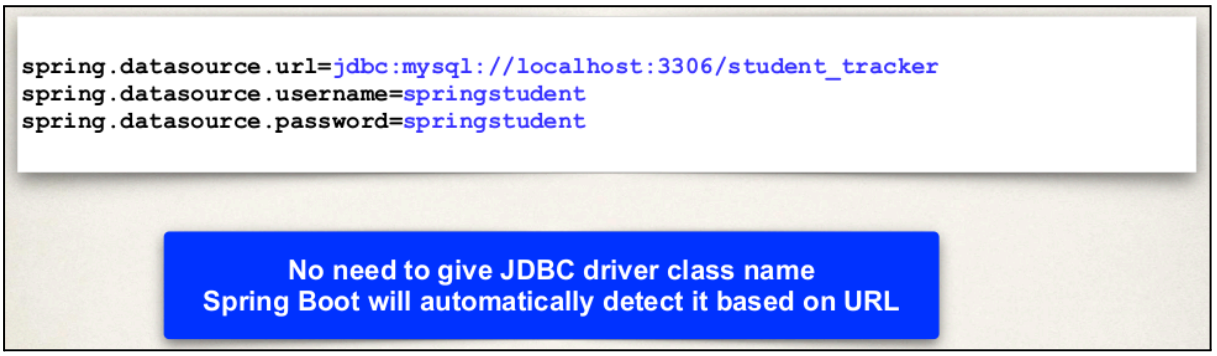
Automatic Data Source Configuration

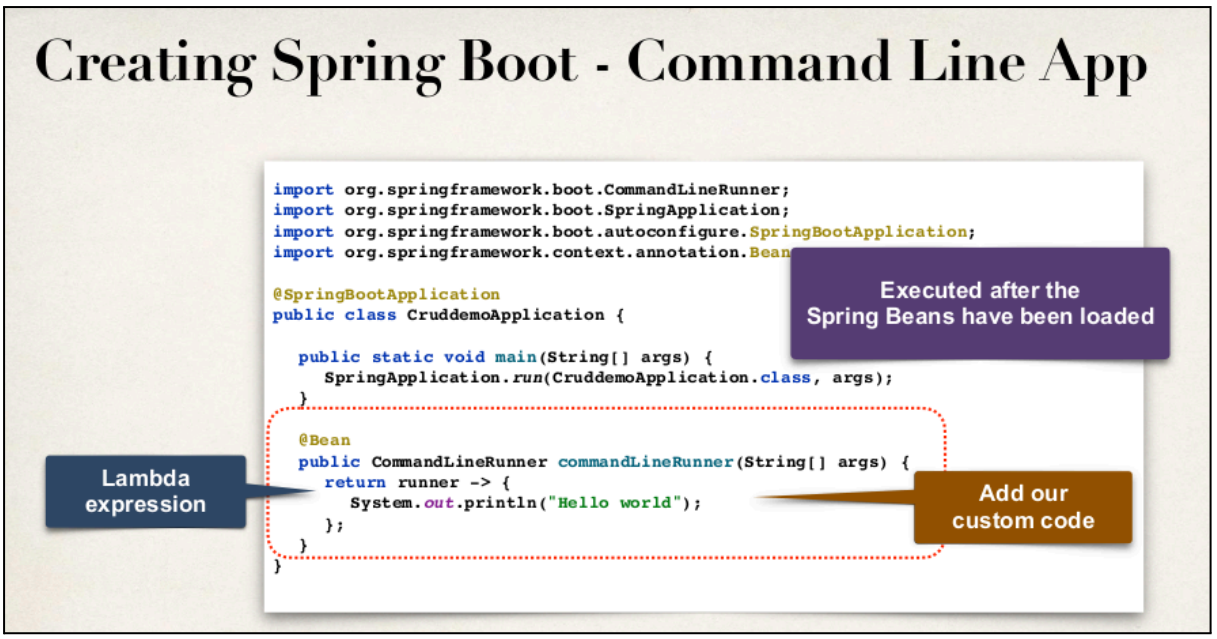
- In Spring Boot, Hibernate is the default implementation of JPA
- EntityManager is the main component for creating queries etc …
- EntityManager is from Jakarta Persistence API (JPA)
- Based on configs, Spring Boot will automatically create the beans:
- DataSource, EntityManager, …
- You can then inject these into your app, for example your DAO

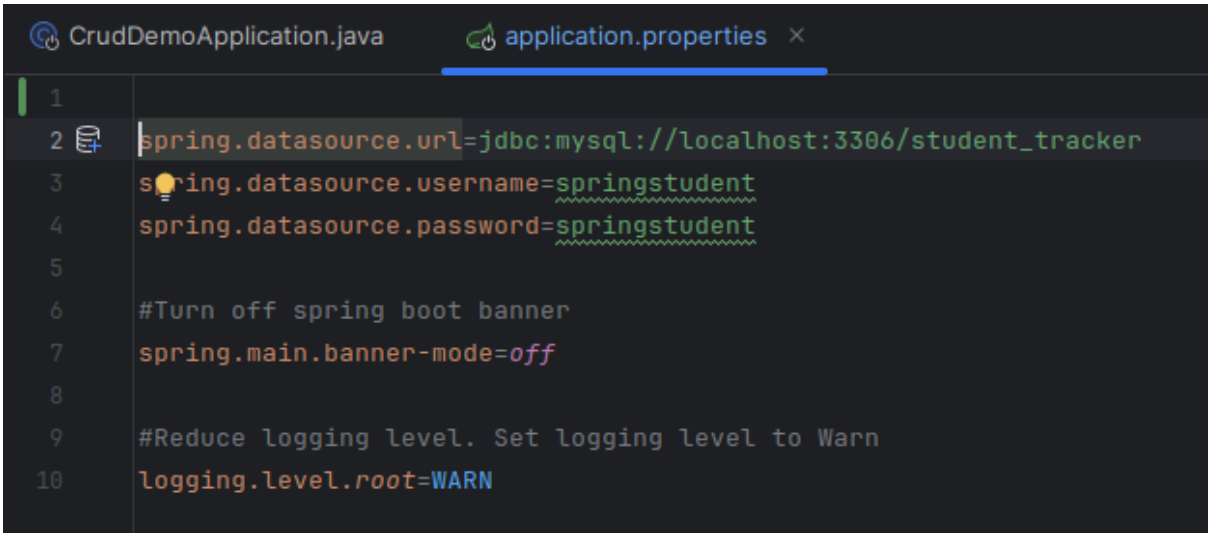
Setting up Project with Spring Initializr

- At Spring Initializr website, start.spring.io
- Add dependencies
    - MySQL Driver: mysql-connector-j
    - Spring Data JPA: spring-boot-starter-data-jpa

Application.properties







Below is a comprehensive list of annotations commonly used in JPA (Java Persistence API)

1. @Entity : Specifies that the class is an entity. This annotation is placed at the class level.

2. @Table : Specifies the name of the database table to be used for mapping the entity. This annotation can be placed at the class level.

3. @Id : Specifies the primary key of the entity. This annotation is placed at the field level.

4. @GeneratedValue : Specifies the strategy used to generate primary key values. This annotation can be used in conjunction with @Id.

5. @Column : Specifies the mapping of a column in the database to a field or property in the entity. This annotation can be placed at the field level.

6. @Embedded : Specifies that an embedded class is to be mapped to the database table. This annotation is placed at the field level.

7. @Embeddable : Specifies that a class is an embeddable class and is mapped as part of the containing entity. This annotation is placed at the class level.

8. @EmbeddedId : Specifies a composite primary key that is mapped to multiple fields or properties of the entity. This annotation is placed at the field level.

9. @OneToOne : Specifies a one-to-one relationship between two entities. This annotation is placed at the field level.

10. @OneToMany : Specifies a one-to-many relationship between two entities. This annotation is placed at the field level.

11. @ManyToOne : Specifies a many-to-one relationship between two entities. This annotation is placed at the field level.

12. @ManyToMany : Specifies a many-to-many relationship between two entities. This annotation is placed at the field level.

13. @JoinTable : Specifies the join table for a many-to-many relationship. This annotation is placed at the field level.

14. @JoinColumn : Specifies the foreign key column used for joining an entity association. This annotation is placed at the field level.

15. @NamedQuery : Specifies a named query for an entity. This annotation is placed at the class level.

16. @NamedQueries : Specifies multiple named queries for an entity. This annotation is placed at the class level.

17. @NamedQuery : Specifies a named native query for an entity. This annotation is placed at the class level.

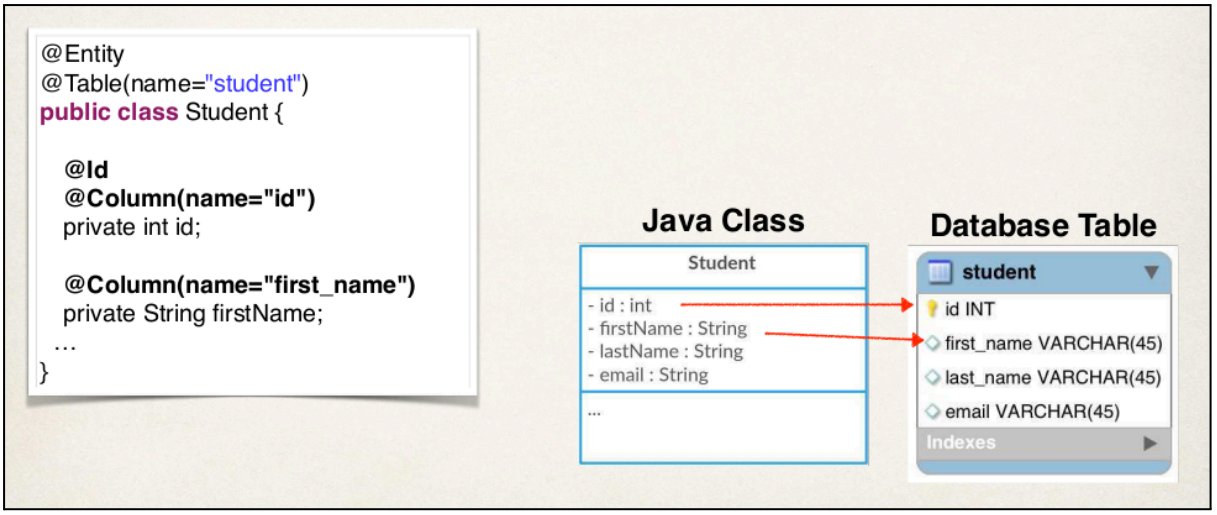
18. @NamedQueries : Specifies multiple named native queries for an entity. This annotation is placed at the class level.

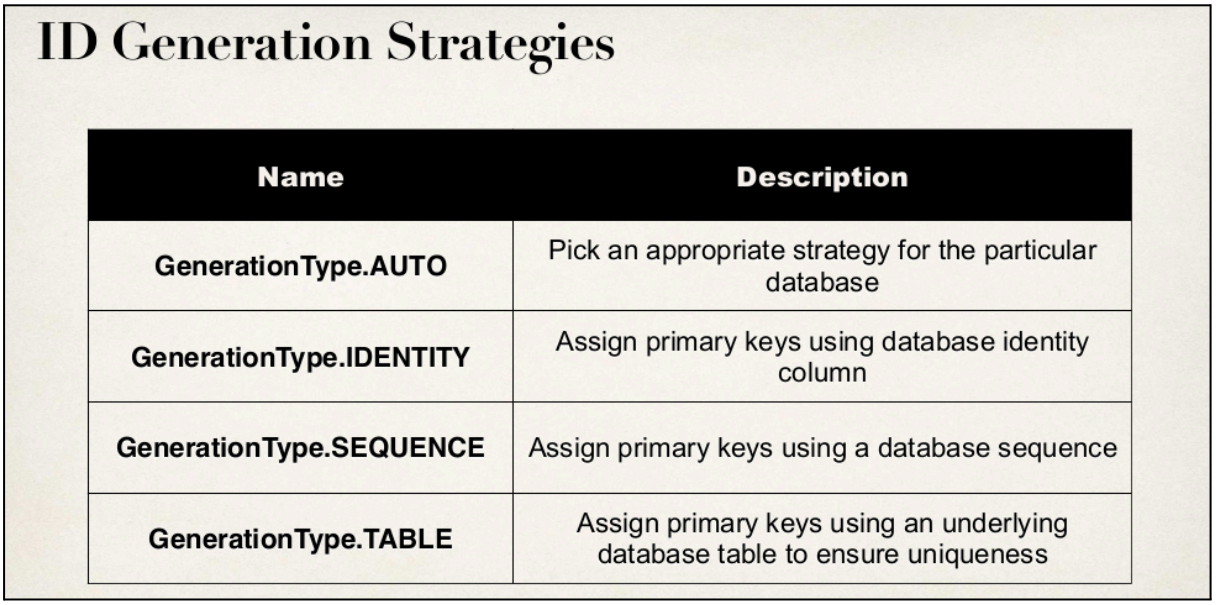
19. @Version : Specifies a version field for optimistic locking. This annotation is placed at the field level.

20. @Transient : Specifies that a field or property is not to be persisted to the database. This annotation is placed at the field level.

21. @SequenceGenerator : Specifies a sequence generator for generating primary key values. This annotation is placed at the class level.

22. @GeneratedValue(strategy = GenerationType.SEQUENCE) : Specifies that the primary key value should be generated using a sequence. This annotation can be used in conjunction with @Id.





Bonus Bonus

- You can define your own CUSTOM generation strategy :-)
- Create implementation of org.hibernate.id.IdentifierGenerator
- Override the method: public Serializable generate(…)

DataSource:

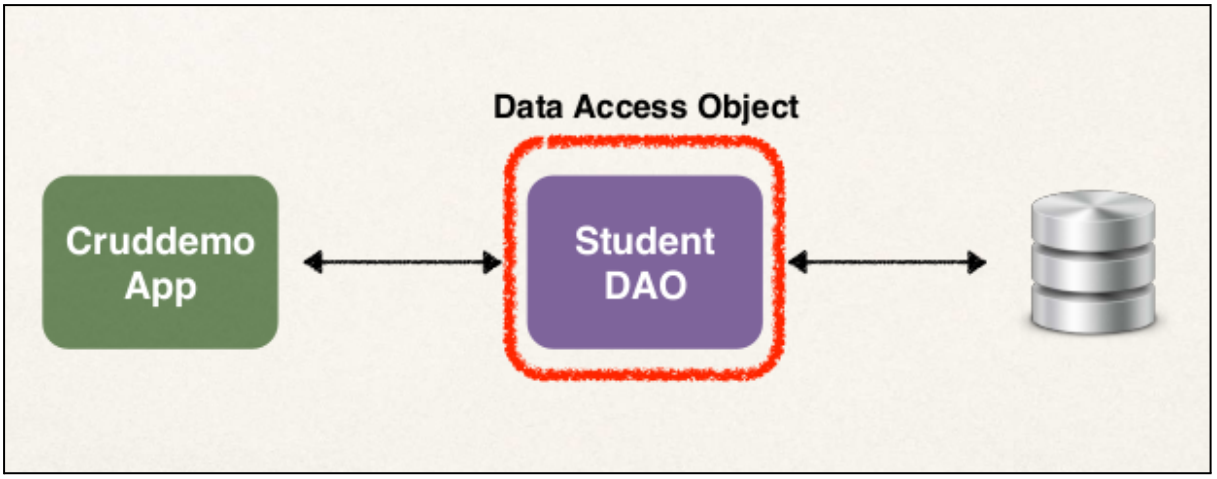
- The DataSource is an interface provided by JDBC (Java Database Connectivity) for creating and managing database connections.
- In a Spring Boot application, the DataSource is typically configured in the application.properties or application.yml file to define the connection properties such as URL, username, and password for the database.

EntityManager:

- The EntityManager is an interface provided by JPA for performing CRUD (Create, Read, Update, Delete) operations on entities (Java objects mapped to database tables).
- EntityManager is responsible for managing the lifecycle of entities, including persisting, updating, and removing them from the database.
- In a Spring Boot application, EntityManager is typically obtained from the EntityManagerFactory, which is configured by Spring to manage JPA entities.
- EntityManager uses the DataSource configured in the application context to establish a connection to the database.

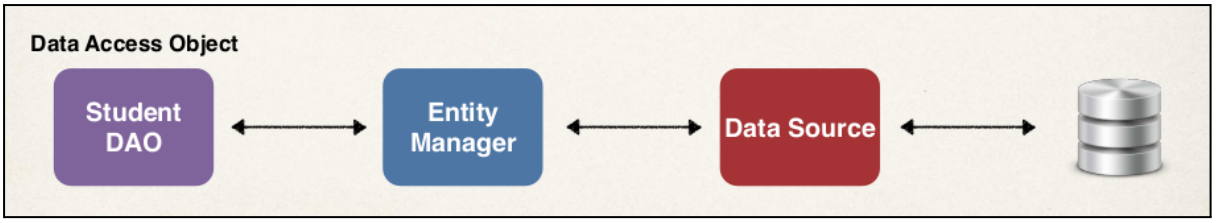
Data Access Object

- Responsible for interfacing with the database
- This is a common **design pattern** : Data Access Object (DAO)



JPA Entity Manager

- Our JPA Entity Manager needs a Data Source
- The Data Source defines database connection info
- JPA Entity Manager and Data Source are automatically created by Spring Boot
- Based on the file: application.properties (JDBC URL, user id, password, etc …)
- We can autowire/inject the JPA Entity Manager into our Student DAO



EntityManager vs JpaRepository

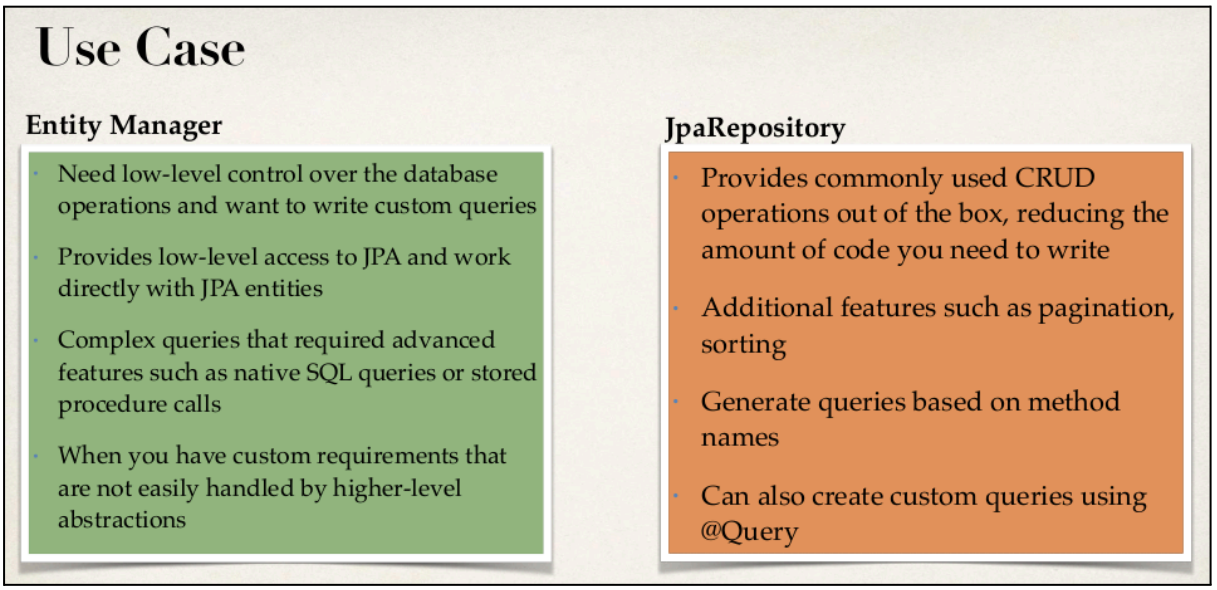
1. EntityManager :
   - The `EntityManager` is part of the JPA specification and is responsible for managing entities, their lifecycle, and executing queries.
   - It provides methods for persisting, merging, removing, and finding entities, as well as creating and executing JPQL (Java Persistence Query Language) queries.
   - EntityManager is typically obtained from the `EntityManagerFactory`, which is configured in the Spring application context.
   - Developers have full control over SQL queries and can write custom JPQL queries as needed.

```
@PersistenceContext
private EntityManager entityManager;
```

2. JpaRepository :
   - `JpaRepository` is part of Spring Data JPA, a higher-level abstraction built on top of JPA.
   - It provides a set of generic CRUD (Create, Read, Update, Delete) methods for working with entities, eliminating the need to write boilerplate code for basic data operations.
   - `JpaRepository` extends `PagingAndSortingRepository`, which in turn extends `CrudRepository`, providing additional functionality for pagination and sorting of query results.
   - Spring Boot automatically generates implementations of `JpaRepository` interfaces at runtime, based on the provided entity types.
   - `JpaRepository` also supports query methods, which allow developers to define custom queries by method naming conventions, without writing JPQL queries explicitly.

```
public interface UserRepository extends JpaRepository<User, Long> {
    List<User> findByLastName(String lastName);
}
```

In summary, while both `EntityManager` and `JpaRepository` can be used for database interactions in a Spring Boot application, `EntityManager` provides low-level control over entities and queries, whereas `JpaRepository` provides higher-level abstractions and simplifies common data access tasks with its predefined methods and query derivation mechanisms. The choice between them depends on the requirements and complexity of your application.



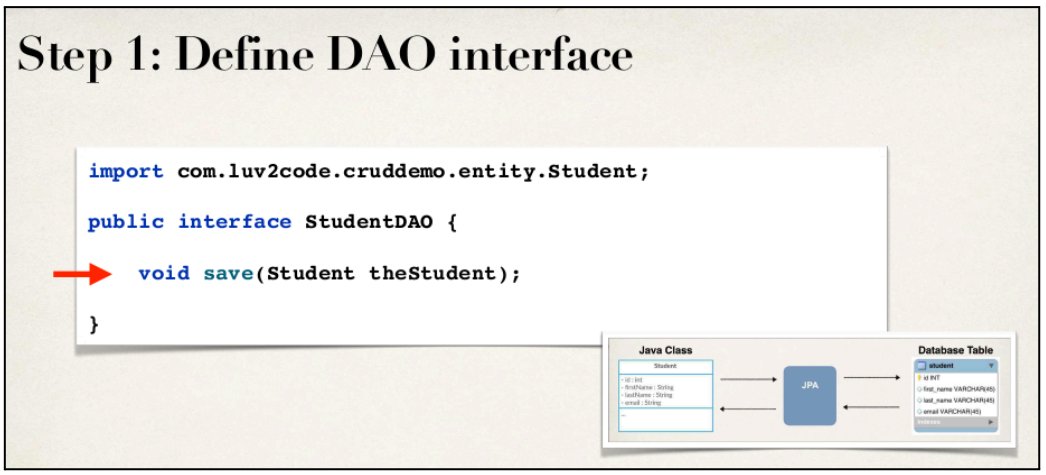
@Transactional

@Transactional is an annotation used in Spring Framework to indicate that a method or class should be wrapped in a transactional context. Transactions ensure that a group of operations are performed atomically, meaning that either all of them succeed or none of them do, preserving data integrity.

@Repository

@Repository is an annotation in the Spring Framework that is used to indicate that a particular class is a Data Access Object (DAO). It's a specialization of the @Component annotation, meant to be used for persistence layer classes.

Save a Java Object

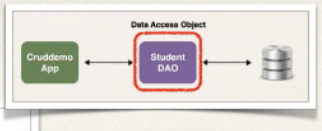


Step 2: Define DAO implementation

```java
import com.luv2code.cruddemo.entity.Student;
import jakarta.persistence.EntityManager;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.transaction.annotation.Transactional;

public class StudentDAOImpl implements StudentDAO {

    private EntityManager entityManager;

    @Autowired
    public StudentDAOImpl(EntityManager theEntityManager) {
        entityManager = theEntityManager;
    }

    @Override
    @Transactional
    public void save(Student theStudent) {
        entityManager.persist(theStudent);
    }

}
```

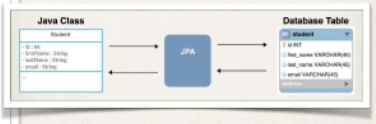
Handles transaction management

Step 3: Update main app




```java
@SpringBootApplication
public class CruddemoApplication {

    public static void main(String[] args) {
        SpringApplication.run(CruddemoApplication.class, args);
    }

    @Bean
    public CommandLineRunner commandLineRunner(StudentDAO studentDAO) {
        return runner -> {

            createStudent(studentDAO);
        };
    }

    private void createStudent(StudentDAO studentDAO) {

        // create the student object
        System.out.println("Creating new student object...");
        Student tempStudent = new Student("Paul", "Doe", "paul@luv2code.com");

        // save the student object
        System.out.println("Saving the student...");
        studentDAO.save(tempStudent);

        // display id of the saved student
        System.out.println("Saved student. Generated id: " + tempStudent.getId());
    }

}
```

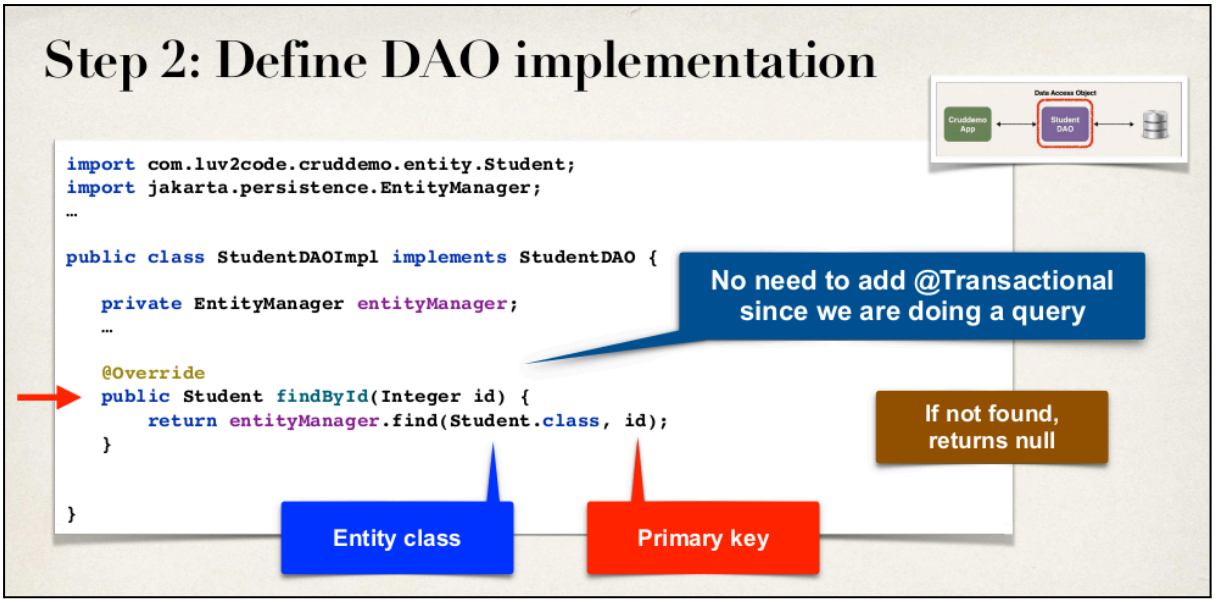
Inject the StudentDAO

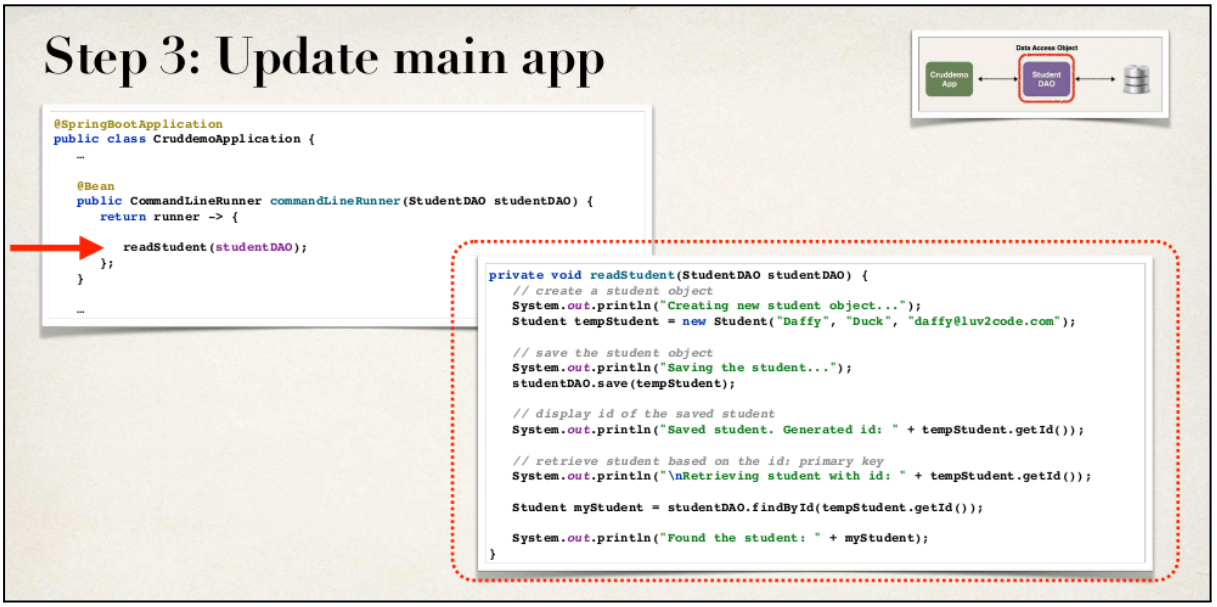


Retrieving an Object

Step 1: Add new method to DAO interface


```java
import com.luv2code.cruddemo.entity.Student;

public interface StudentDAO {
    ...

    Student findById(Integer id);

}
```



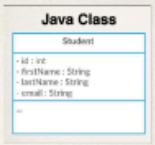


JPA Query Language (JPQL)

JPQL (Java Persistence Query Language): JPQL is a query language provided by JPA for performing database operations on entities. It allows developers to write queries in terms of Java objects and their attributes, rather than directly interacting with database tables and columns.

- Query language for retrieving objects
- Similar in concept to SQL
- where, like, order by, join, in, etc…
- However, JPQL is based on entity name and entity fields

Retrieving all Students



Name of JPA Entity ... the class name

```
TypedQuery<Student> theQuery = entityManager.createQuery("FROM Student", Student.class);

List<Student> students = theQuery.getResultList();
```

Note: this is NOT the name of the database table

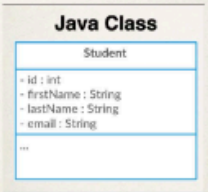
All JPQL syntax is based on entity name and entity fields

Retrieving Students using OR predicate:

Field of JPA Entity

```
TypedQuery<Student> theQuery = entityManager.createQuery(
                    "FROM Student WHERE lastName='Doe' OR firstName='Daffy'", Student.class);

List<Student> students = theQuery.getResultList();
```

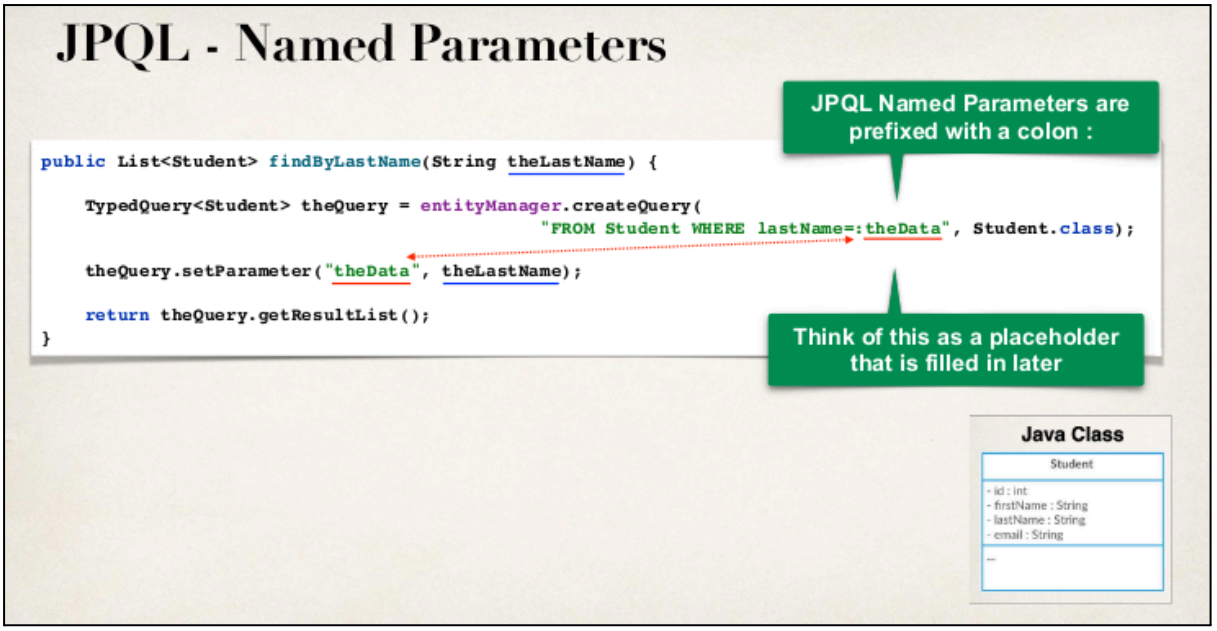
Field of JPA Entity



Retrieving Students using LIKE predicate:

```
TypedQuery<Student> theQuery = entityManager.createQuery(
                        "FROM Student WHERE email LIKE '%luv2code.com'", Student.class);

List<Student> students = theQuery.getResultList();
```

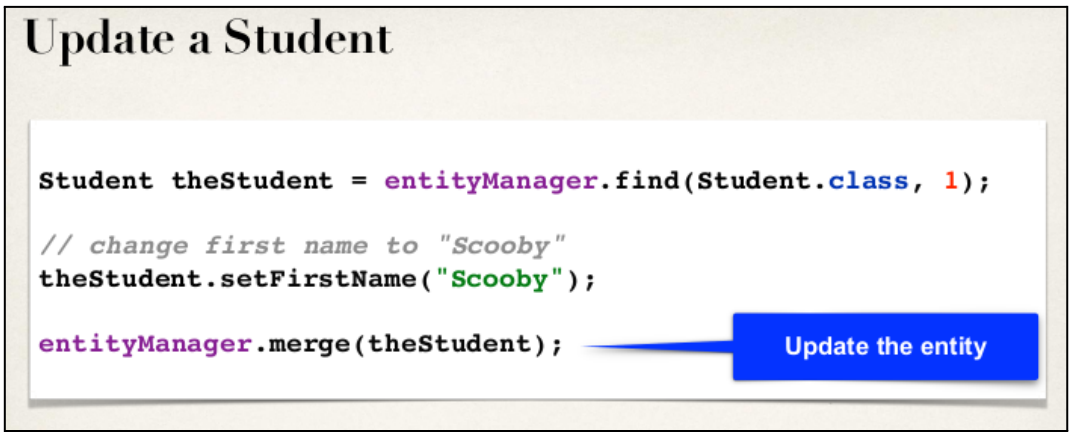
Match of email addresses that ends with `luv2code.com`




```
private void findAllStudents(StudentDao studentDAO) {
    System.out.println("Retrieving all students ...");
    List<Student> all = studentDAO.findAll();

    //using lambda expression
    all.forEach(student -> System.out.println(student));

    //using method reference
    all.forEach(System.out::println);
}
```

For each student entity in the list, the System.out::println method reference is used to print the student details to the console. This shorthand notation is equivalent to using a lambda expression like student -> System.out.println(student).

Updating an Object



Update last name for all students

```
int numRowsUpdated = entityManager.createQuery(
                        "UPDATE Student SET lastName='Tester'")
                        .executeUpdate();
```

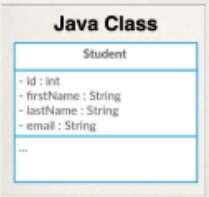
Field of JPA Entity

Name of JPA Entity ... the class name

Return the number of rows updated

Execute this statement

Java Class



Step 2: Define DAO implementation

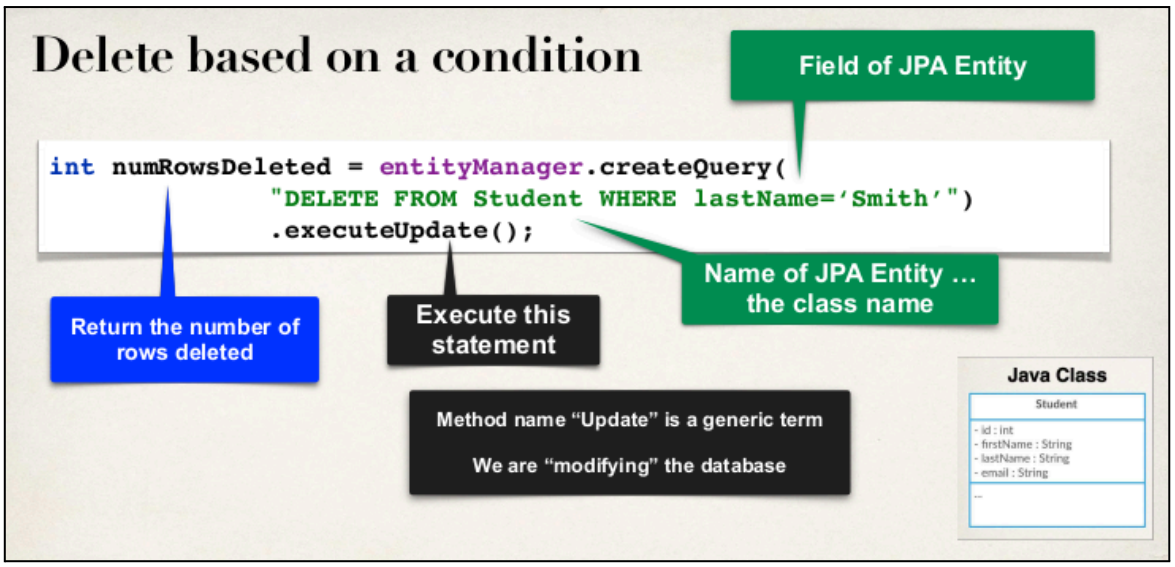
```
import com.luv2code.cruddemo.entity.Student;
import jakarta.persistence.EntityManager;
import org.springframework.transaction.annotation.Transactional;
…

public class StudentDAOImpl implements StudentDAO {

   private EntityManager entityManager;
   …

   @Override
   @Transactional
   public void update(Student theStudent) {
       entityManager.merge(theStudent);
   }

}
```

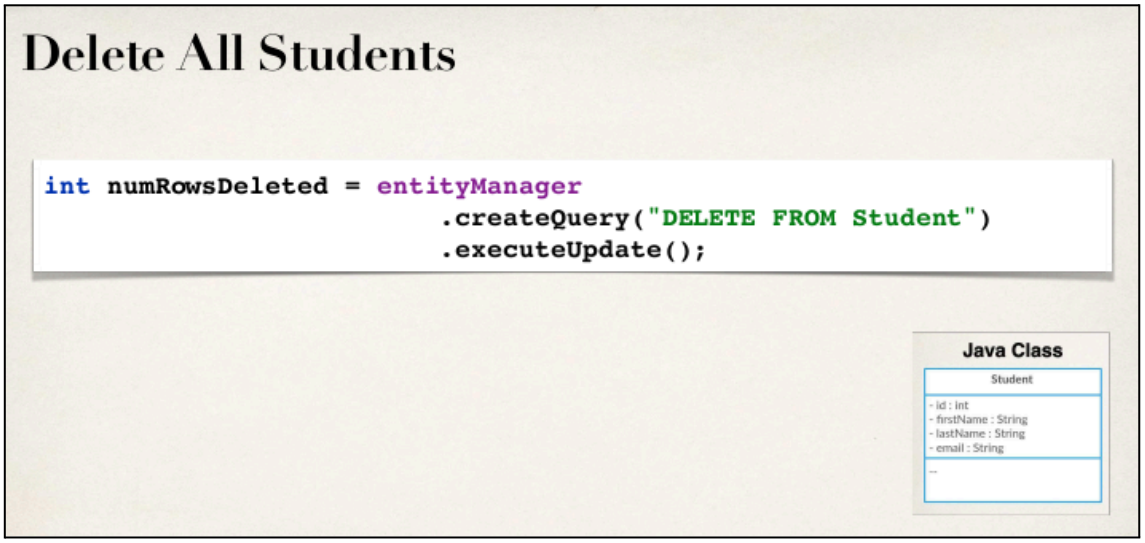
Add @Transactional since we are performing an update

Deleting an Object

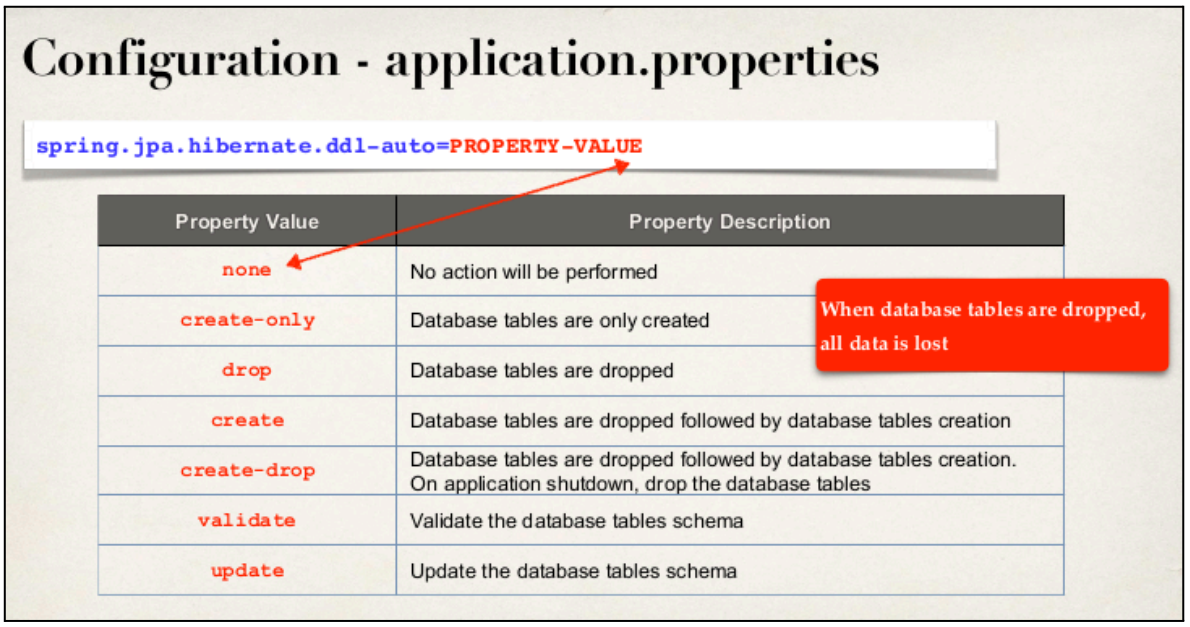
Delete a Student

```
// retrieve the student
int id = 1;
Student theStudent = entityManager.find(Student.class, id);

// delete the student
entityManager.remove(theStudent);
```





Create Database Tables from Java Code



Property Value	Property Description
none	No action will be performed
create-only	Database tables are only created
drop	Database tables are dropped
create	Database tables are dropped followed by database tables creation
create-drop	Database tables are dropped followed by database tables creation. On application shutdown, drop the database tables
validate	Validate the database tables schema
update	Update the database tables schema

Recommendation

- In general, I don't recommend auto generation for enterprise, real-time projects
  - You can VERY easily drop PRODUCTION data if you are not careful 
- I recommend SQL scripts ✓
  - Corporate DBAs prefer SQL scripts for governance and code review
  - The SQL scripts can be customized and fine-tuned for complex database designs
  - The SQL scripts can be version-controlled
  - Can also work with schema migration tools such as Liquibase and Flyway

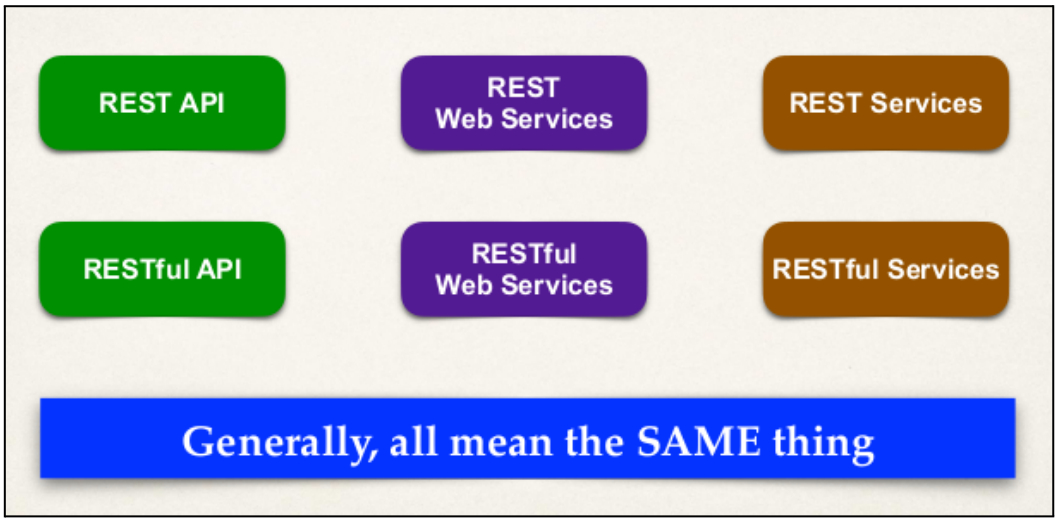
REST APIs - REST Web Services

- [ ] Create REST APIs / Web Services with Spring
- [ ] Discuss REST concepts, JSON and HTTP messaging
- [ ] Install REST client tool: Postman
- [ ] Develop REST APIs / Web Services with @RestController
- [ ] Build a CRUD interface to the database with Spring REST

- We can make REST API calls over HTTP
- REST : REpresentational State Transfer
- Lightweight approach for communicating between applications
- REST is language independent
- The client application can use ANY programming language
- The server application can use ANY programming language
- REST applications can use any data format
- Commonly see XML and JSON
- JSON is most popular and modern - JavaScript Object Notation

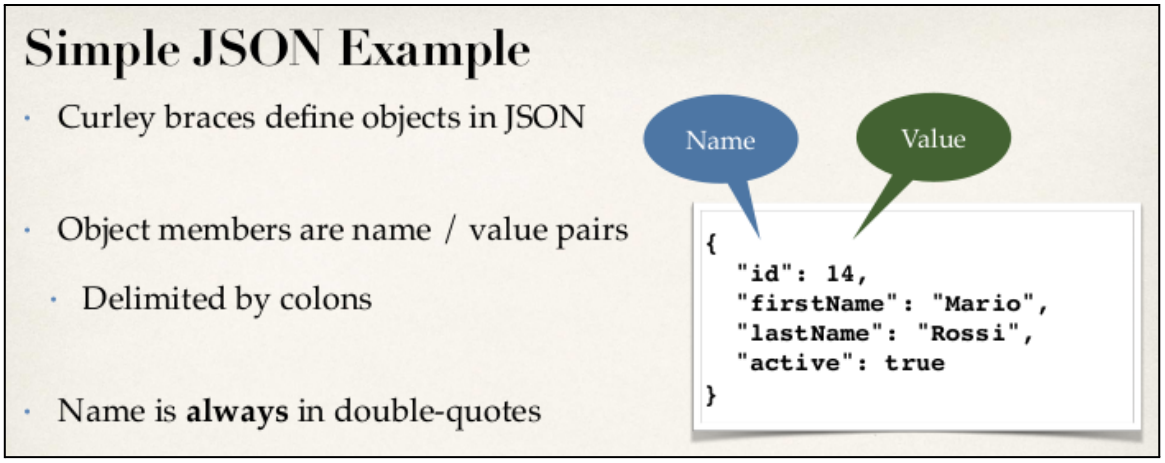
A REST API (Representational State Transfer Application Programming Interface) is a set of rules and conventions used for building web services that allow communication between different systems over the internet. RESTful APIs are designed to be stateless, meaning each request from a client to the server contains all the information needed to process the request.

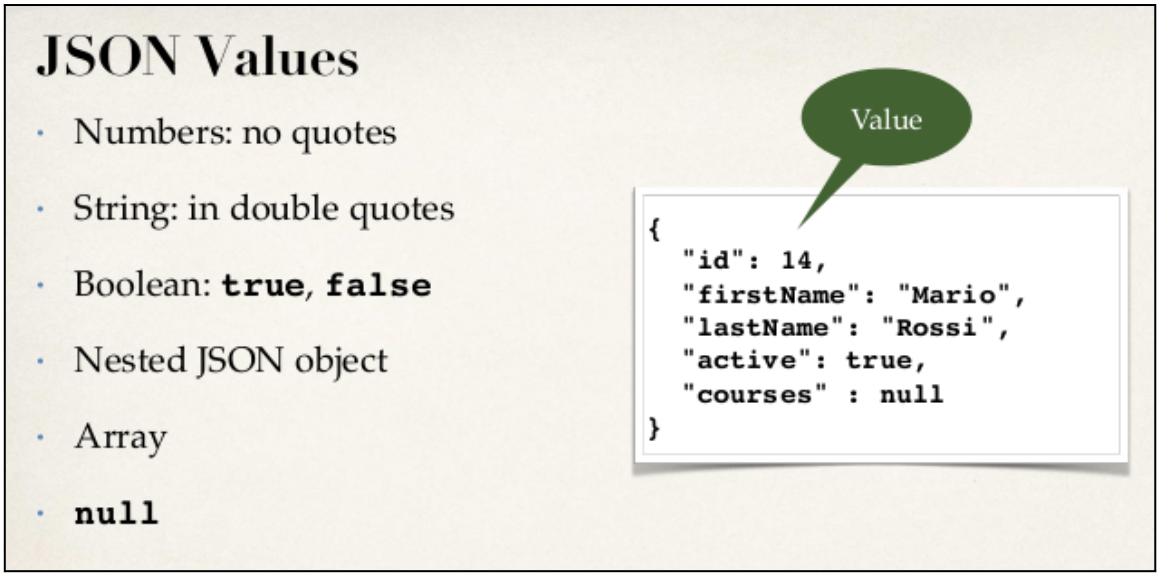
JSON (JavaScript Object Notation) is a lightweight data interchange format that is easy for humans to read and write, and easy for machines to parse and generate. It is based on a subset of the JavaScript programming language, but is language-independent, making it widely used across different programming languages and platforms. JSON is commonly used for transmitting data between a server and a web application as an alternative to XML.



What is JSON?

- JavaScript Object Notation
- Lightweight data format for storing and exchanging data … plain text
- Language independent … not just for JavaScript
- JSON is just plain text data
- Can use with any programming language: Java, C#, Python etc …





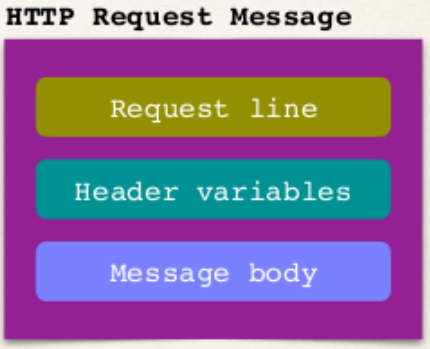
REST HTTP Basics

HTTP (Hypertext Transfer Protocol) defines a set of request methods, also known as HTTP verbs, that indicate the desired action to be performed on a resource. These methods are used to communicate between clients (such as web browsers) and servers, specifying how the server should process the request.

HTTP Method	CRUD Operation
POST	Create a new entity
GET	Read a list of entities or single entity
PUT	Update an existing entity
DELETE	Delete an existing entity

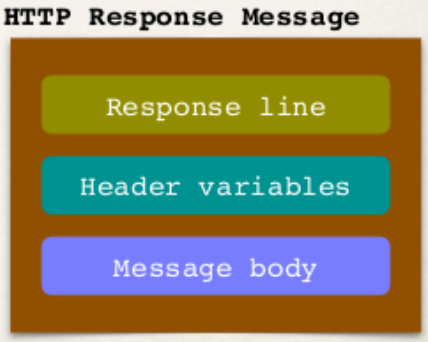
HTTP Request Message

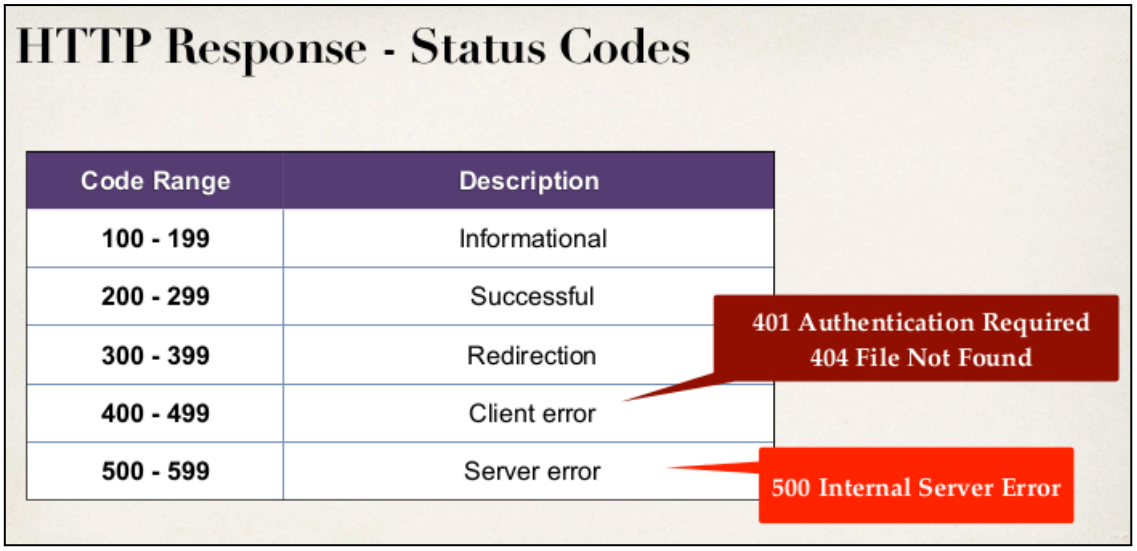
- Request line: the HTTP command
- Header variables: request metadata
- Message body: contents of message



HTTP Response Message

- Response line: server protocol and status code
- Header variables: response metadata
- Message body: contents of message





MIME Content Types

- The message format is described by MIME content type
- Multipurpose Internet Mail-Extension
- Basic Syntax: type/sub-type
- Examples
- text/html, text/plain
- application/json, application/xml, …

MIME (Multipurpose Internet Mail Extensions) content types, also known as MIME types or media types, are standardized labels used to indicate the type and format of data being transmitted over the internet. They are used in HTTP headers and email messages to specify how the data should be interpreted by the recipient.

A MIME content type consists of two parts:

1. Type : The general category or type of the data. Examples include text, image, audio, video, application, etc.

2. Subtype : A more specific identifier for the type of data. Subtypes provide additional information about the content, such as the file format or encoding.

MIME content types are typically expressed as a string with the type and subtype separated by a forward slash (/). For example:

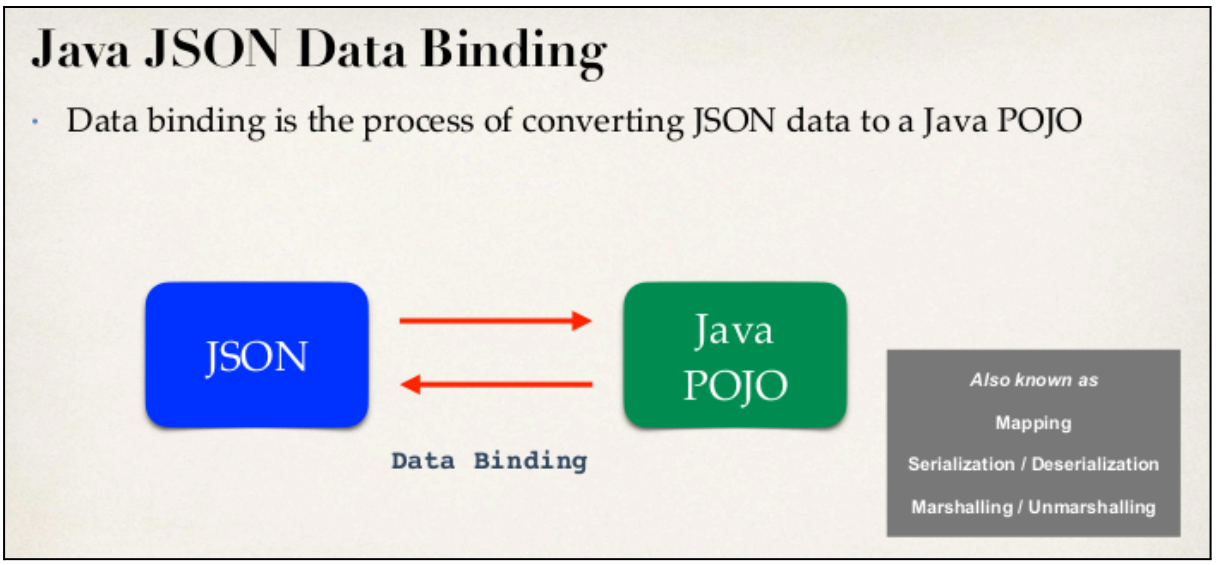
- - `text/plain`: Plain text document
- - `image/jpeg`: JPEG image file
- - `audio/mp3`: MP3 audio file
- - `video/mp4`: MP4 video file
- - `application/json`: JSON data

Additionally, MIME types may include parameters to provide further details about the content. For example:

- - `text/html; charset=UTF-8`: HTML document encoded using UTF-8 character encoding
- - `application/json; charset=UTF-8`: JSON data encoded using UTF-8 character encoding

MIME content types are important for web servers and browsers to properly handle different types of data. They allow servers to specify the type of content being served, and browsers to interpret and display the content accordingly. Incorrect or missing MIME types can result in data being displayed incorrectly or not at all by web browsers. Therefore, it is essential for developers to set appropriate MIME types when serving content over the web.

Java JSON Data Binding



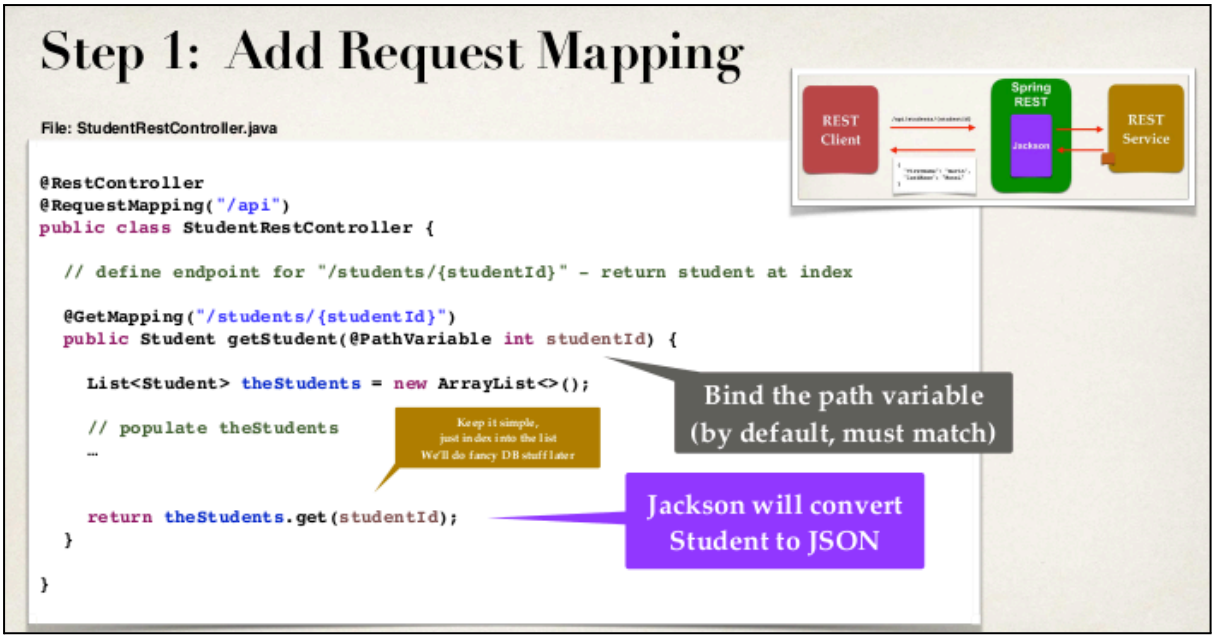
JSON Data Binding with Jackson

- Spring uses the Jackson Project behind the scenes (Available at spring starter web)
- Jackson handles data binding between JSON and Java POJO

Java POJO stands for Plain Old Java Object. It is a term used to describe a simple Java object that does not depend on any framework or external library. POJOs are often used to represent data in applications and are typically used as DTOs (Data Transfer Objects) or model objects.

@PathaVariable

@PathVariable is an annotation used in Spring Framework to extract values from the URI template and map them to method parameters in Spring MVC controllers. It is commonly used to capture dynamic values (such as IDs or names) from the URL path and pass them to controller methods for processing.

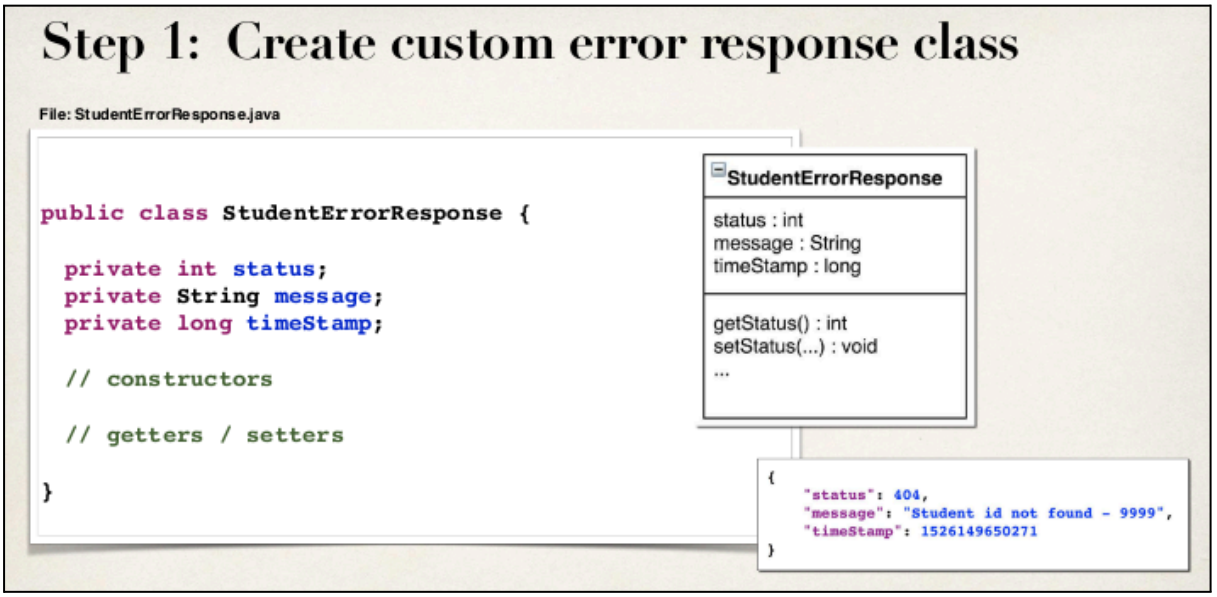


Spring REST - Exception Handling

@ExceptionHandler is an annotation commonly used in Java Spring Framework for handling exceptions in Spring MVC controllers. When applied to a method within a controller, it allows developers to specify how to handle specific exceptions that may be thrown during the execution of that method. This annotation helps in centralizing error handling logic within the application.

Development Process

1. Create a custom error response class
2. Create a custom exception class
3. Update REST service to throw exception if student not found
4. Add an exception handler method using @ExceptionHandler



Step 2: Create custom student exception

File: StudentNotFoundException.java

```
public class StudentNotFoundException extends RuntimeException {

  public StudentNotFoundException(String message) {
    super(message);
  }

}
```

Call super class constructor

Step 3: Update REST service to throw exception

File: StudentRestController.java

```
@RestController
@RequestMapping("/api")
public class StudentRestController {

  @GetMapping("/students/{studentId}")
  public Student getStudent(@PathVariable int studentId) {

    // check the studentId against list size

    if ( (studentId >= theStudents.size()) || (studentId < 0) ) {
      throw new StudentNotFoundException("Student id not found - " + studentId);
    }

    return theStudents.get(studentId);

  }
  ...
}
```

Could also check results from DB

Throw exception

Happy path

Step 4: Add exception handler method

Exception handler method

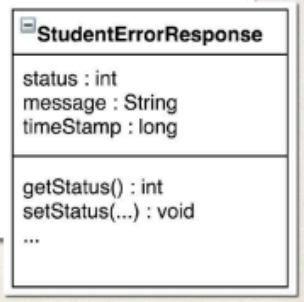
Type of the response body

Exception type to handle / catch

```
java

pi")
public class StudentRestController {
  ...

  @ExceptionHandler
  public ResponseEntity<StudentErrorResponse> handleException(StudentNotFoundException exc) {

    StudentErrorResponse error = new StudentErrorResponse();

    error.setStatus(HttpStatus.NOT_FOUND.value());
    error.setMessage(exc.getMessage());
    error.setTimeStamp(System.currentTimeMillis());

    return new ResponseEntity<>(error, HttpStatus.NOT_FOUND);
  }
}
```

Body

Status code



Spring REST - Global Exception Handling

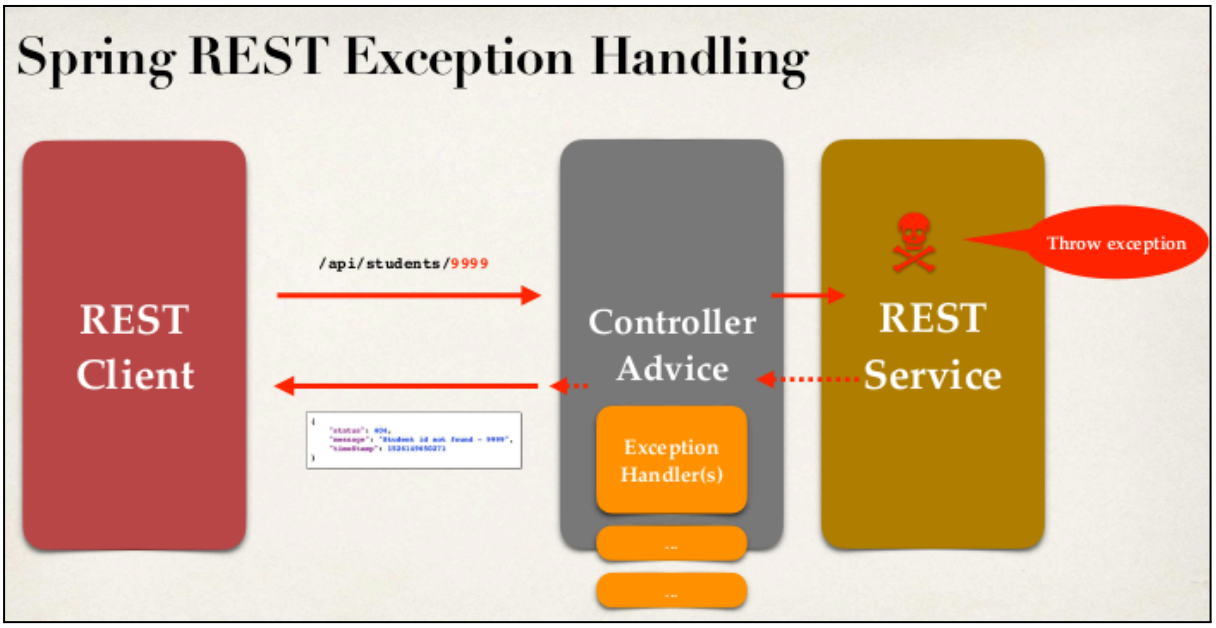
@ControllerAdvice

@ControllerAdvice is an annotation in the Spring Framework of Java. It's used to define global or common exception handling logic across multiple controllers in a Spring MVC application.

By using @ControllerAdvice, you can centralize exception handling logic in one place rather than duplicating it across multiple controllers. This improves code maintainability and makes it easier to manage error responses in your application.

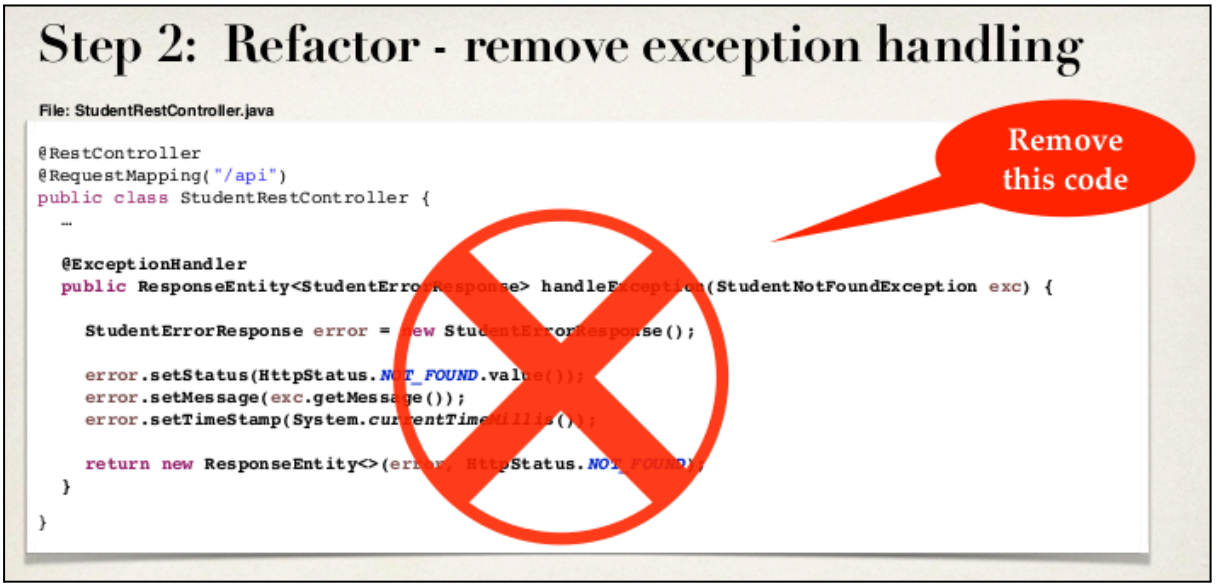
- Exception handler code is only for the specific REST controller
- Can't be reused by other controllers :-(
- We need global exception handlers
- Promotes reuse
- Centralized exception handling
- Spring @ControllerAdvice
- @ControllerAdvice is similar to an interceptor / filter
- Pre-process requests to controllers
- Post-process responses to handle exceptions
- Perfect for global exception handling

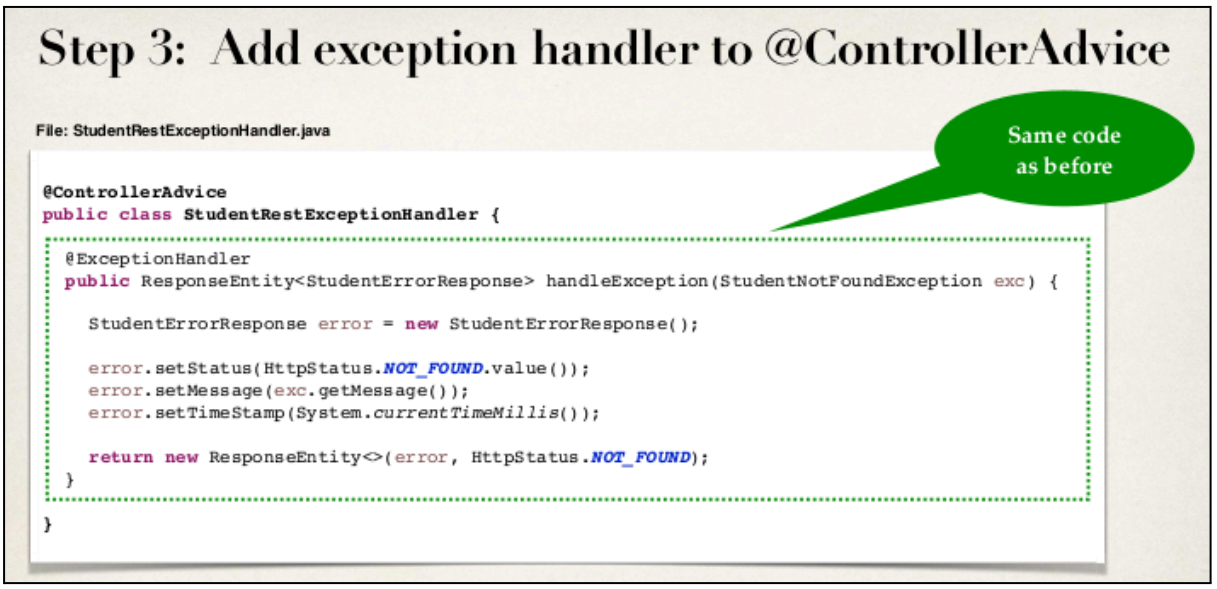
Real-time use of AOP.



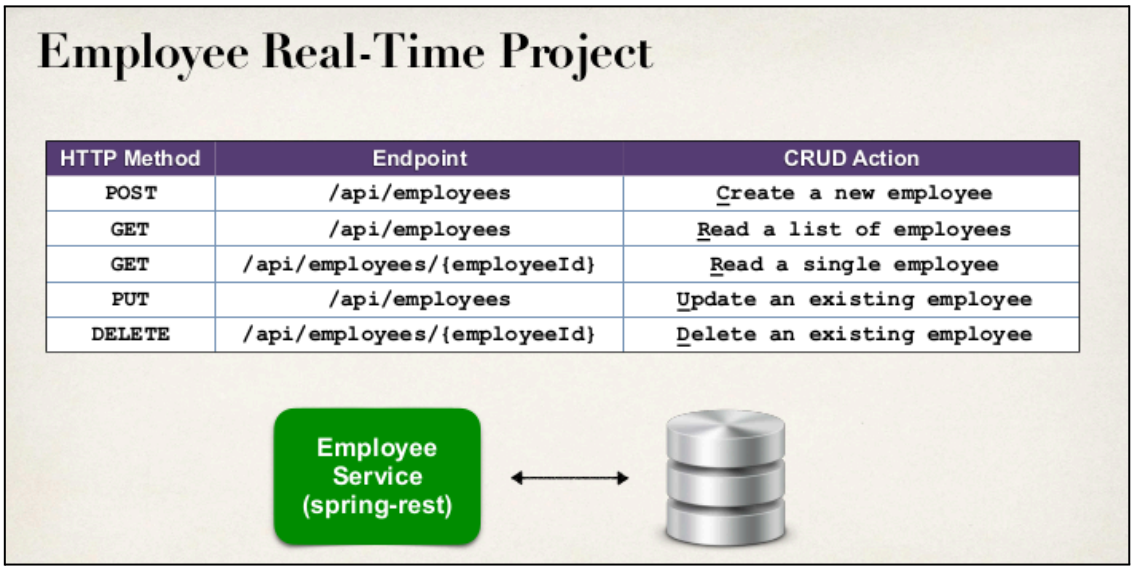
Development Process

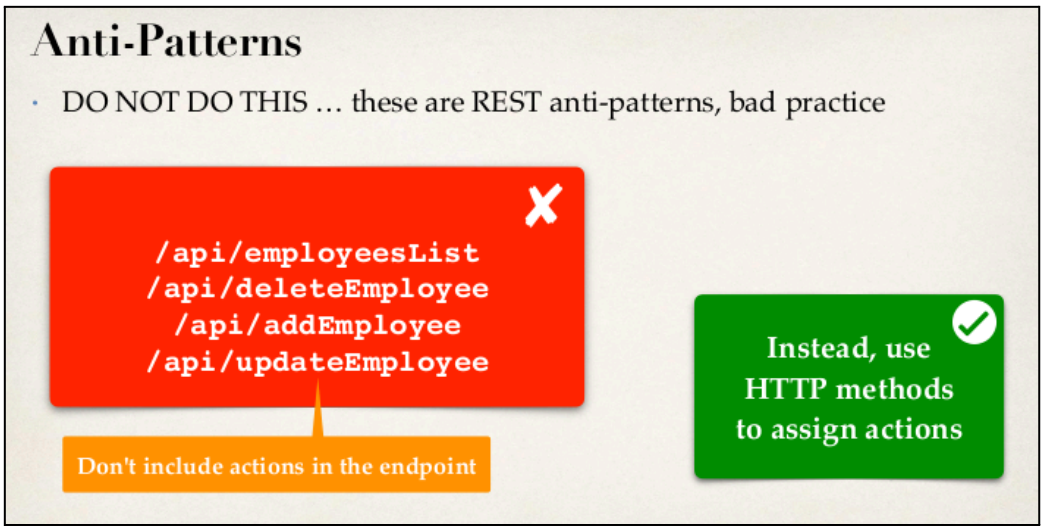
1. Create new @ControllerAdvice
2.Refactor REST service … remove exception handling code
3. Add exception handling code to @ControllerAdvice



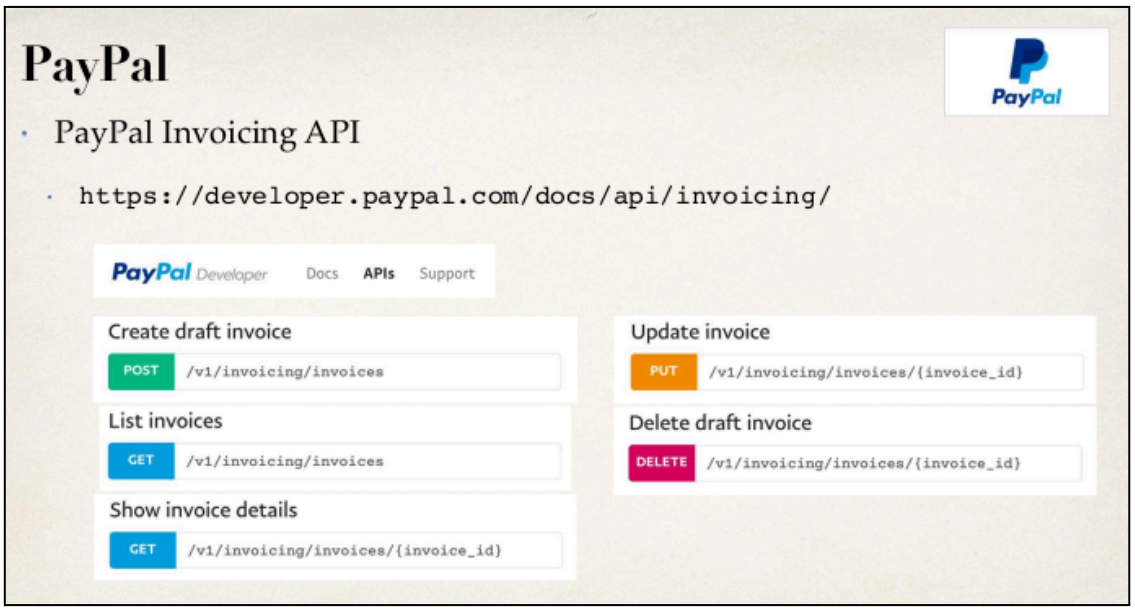


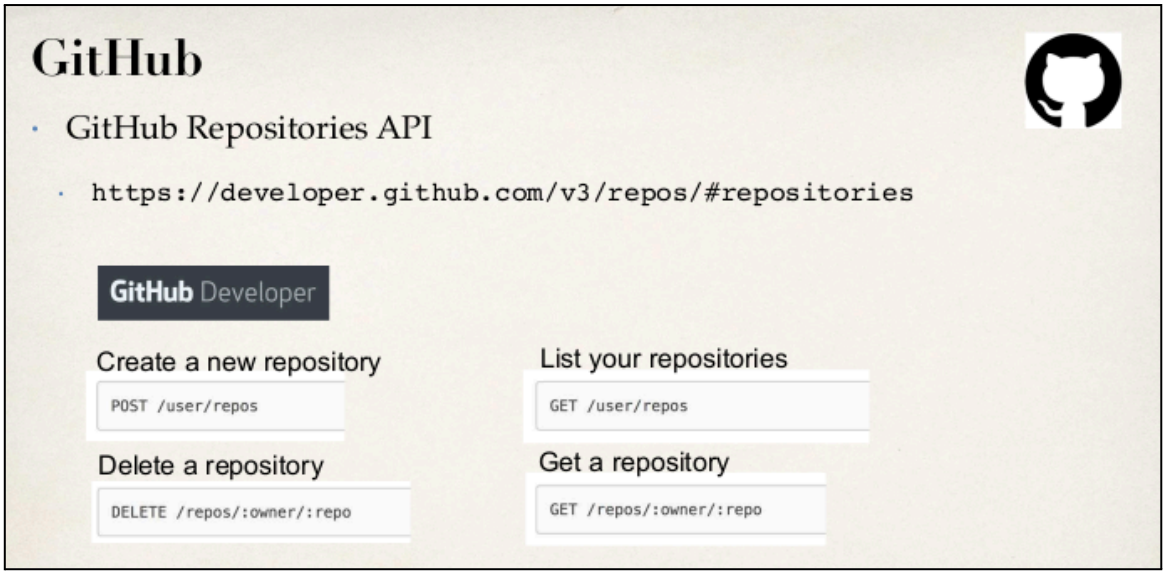
Spring REST API Design

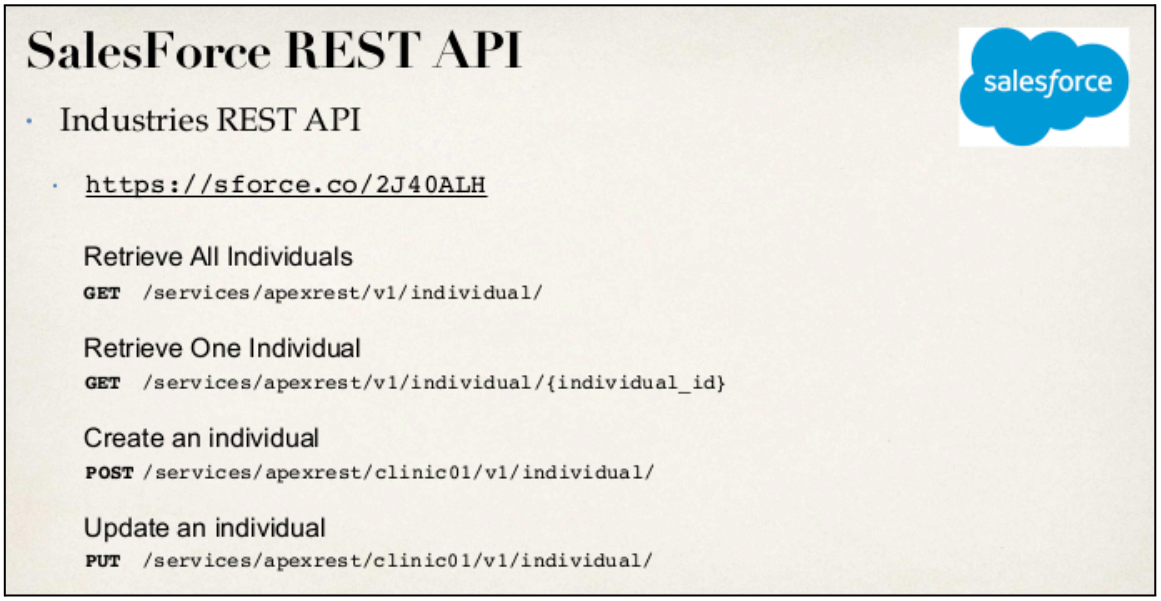




APIs from real-time projects

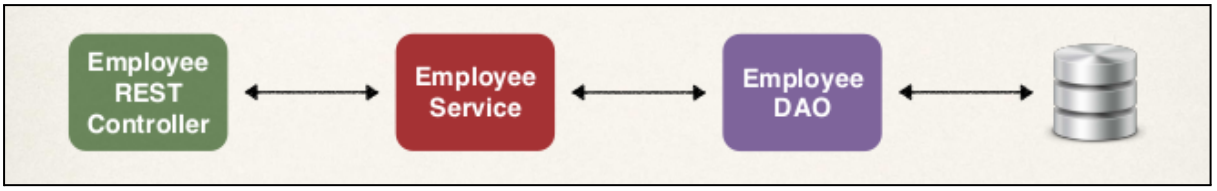






Spring Boot REST API - Real Time Project

Application Architecture



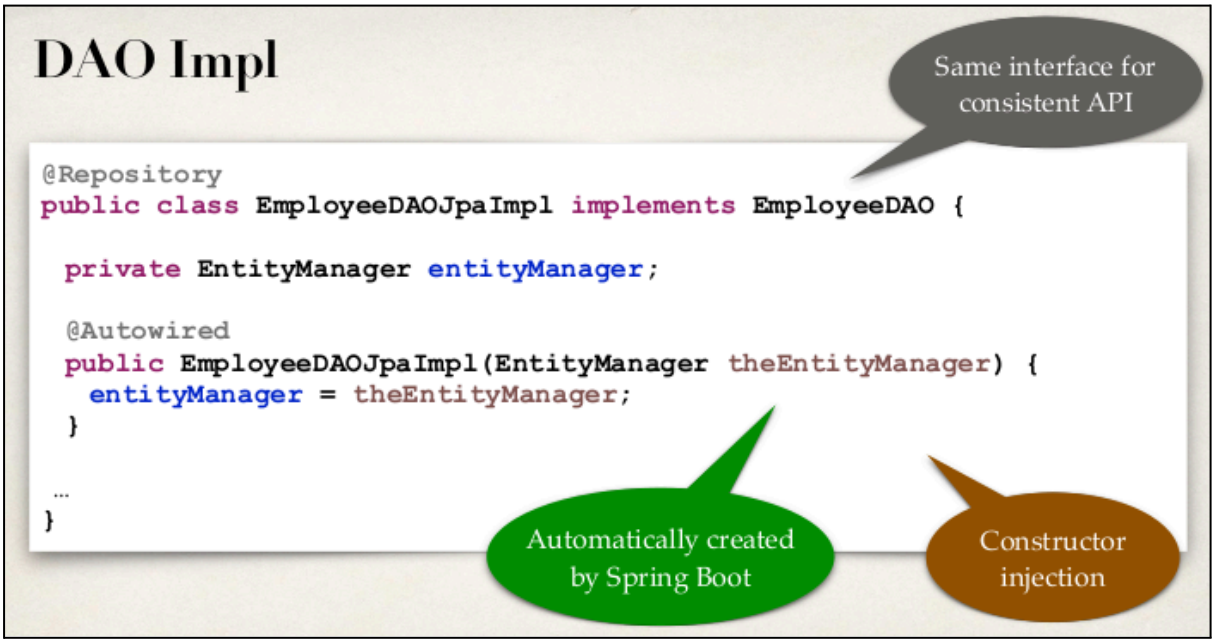
Development Process

1. Update db configs in application.properties
2. Create Employee entity
3. Create DAO interface
4. Create DAO implementation
5. Create REST controller to use DAO

@Repository

In the Spring Framework for Java, @Repository is an annotation used to indicate that a particular class is a repository, typically used for database access or any form of data storage and retrieval.

When you annotate a class with @Repository, Spring automatically detects it during component scanning and creates a bean of that class, making it available for autowiring into other Spring-managed components, such as services or controllers.



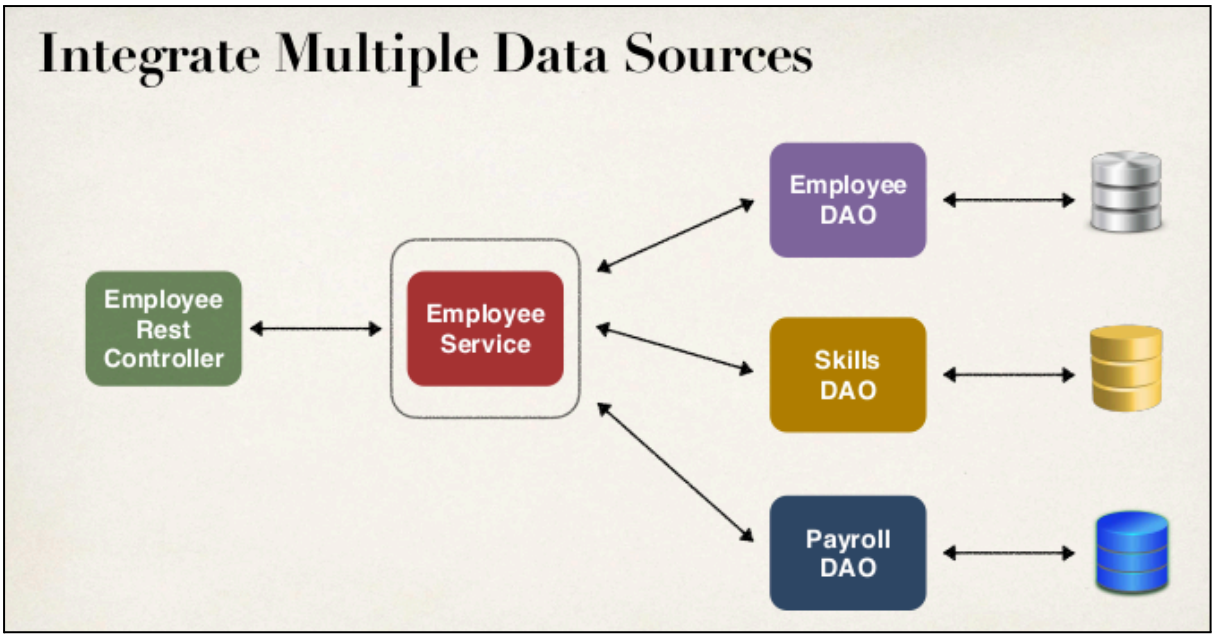
Get a list of employees

```
@Override
public List<Employee> findAll() {

  // create a query
  TypedQuery<Employee> theQuery =
     entityManager.createQuery("from Employee", Employee.class);

  // execute query and get result list
  List<Employee> employees = theQuery.getResultList();

  // return the results
  return employees;
}
```

Define Services with @Service

Purpose of Service Layer

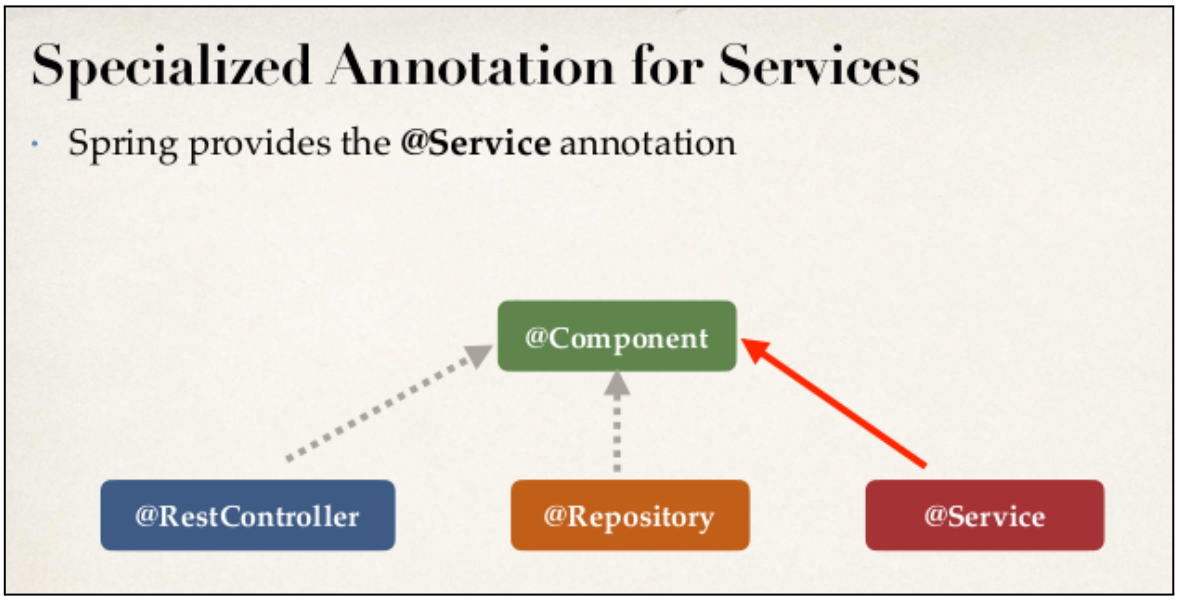
- ✤ Service Facade design pattern
- ✤ Intermediate layer for custom business logic
- ✤ Integrate data from multiple sources (DAO/repositories)



@Service

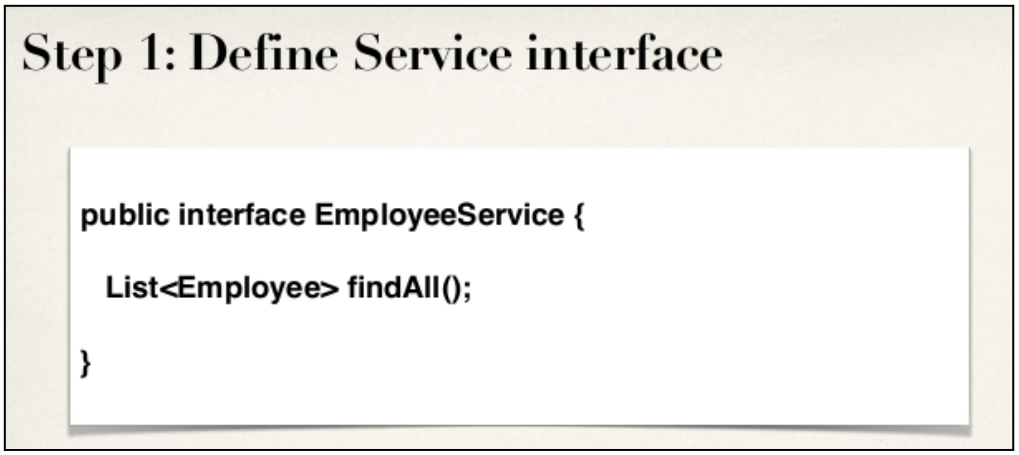
In Spring Framework for Java, @Service is an annotation used to indicate that a particular class is a service component in the application. It's typically used to denote a class that performs some business logic, transaction management, or other operations in the service layer of an application.

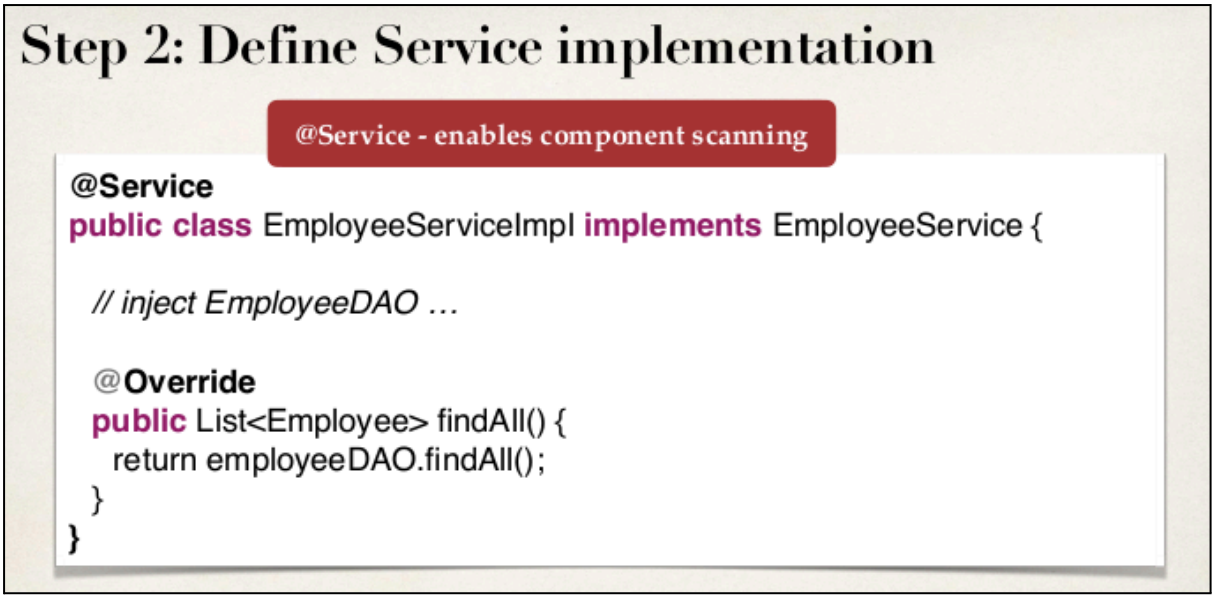
When you annotate a class with @Service, Spring automatically detects it during component scanning and creates a bean of that class. This makes the service class available for autowiring into other Spring-managed components, such as controllers, other services, or repositories.



Service Layer - Best Practice

✤Best practice is to apply transactional boundaries at the service layer
✤It is the service layer's responsibility to manage transaction boundaries
✤For implementation code
✤Apply @Transactional on service methods
✤Remove @Transactional on DAO methods if they already exist





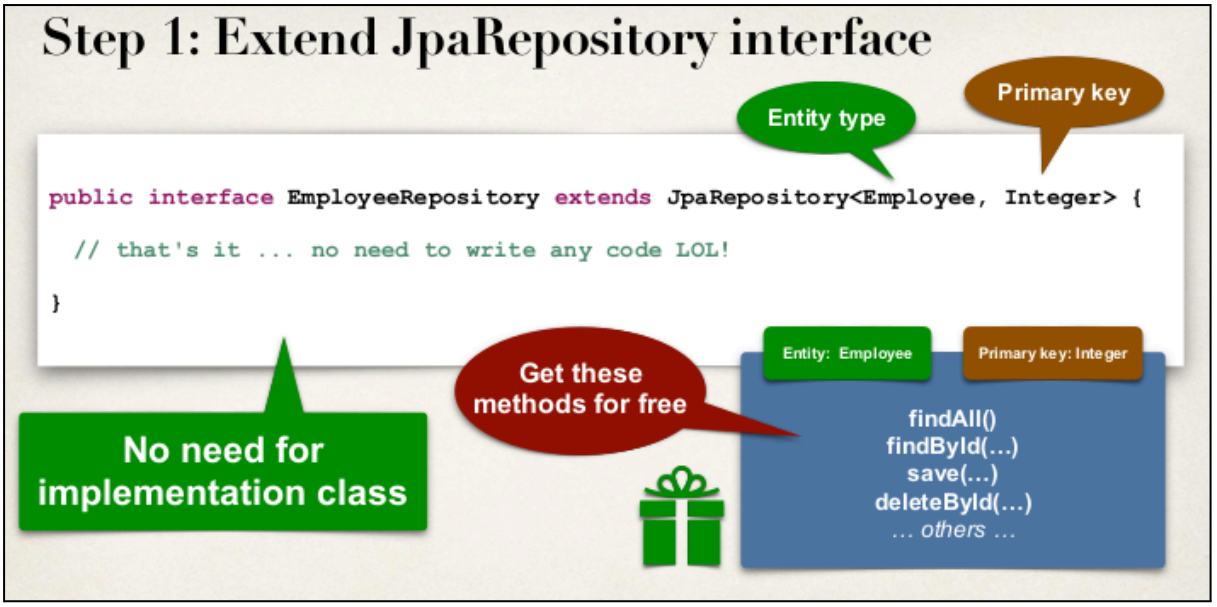
Spring Data JPA in Spring Boot

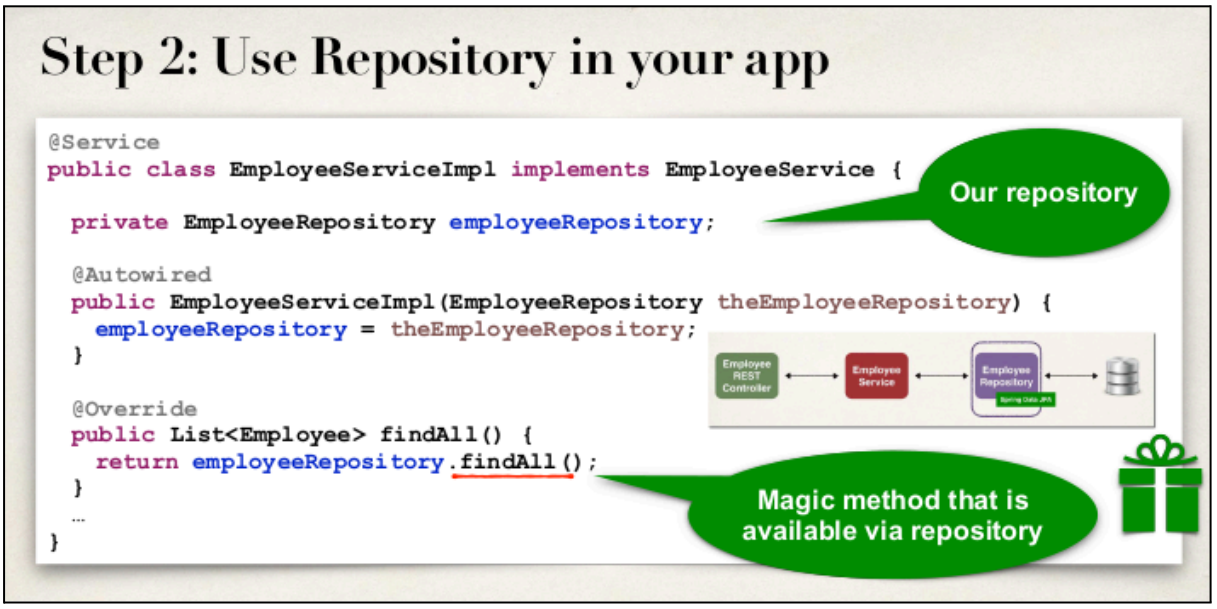
Spring Data JPA is a part of the larger Spring Data project that aims to simplify data access in Spring applications. It provides a layer of abstraction on top of the standard JPA (Java Persistence API) specifications, making it easier to interact with relational databases.

Spring Data JPA vs Entity Manager
Spring Data JPA is a higher-level abstraction that simplifies data access by providing ready-to-use repositories and query methods, whereas EntityManager offers more control and flexibility but requires more manual coding.

The EntityManager interface is part of the Java Persistence API (JPA), which is primarily designed for relational databases. Therefore, EntityManager itself is not suitable for directly interacting with non-relational databases like MongoDB, Redis, or Cassandra.

Spring Data JPA, as its name implies, is primarily designed for relational databases and is built on top of the Java Persistence API (JPA), which is tailored for relational databases. Therefore, Spring Data JPA itself does not directly support non-relational databases.





Advanced Features

- Extending and adding custom queries with JPQL
- Query Domain Specific Language (Query DSL)
- Defining custom methods (low-level coding)

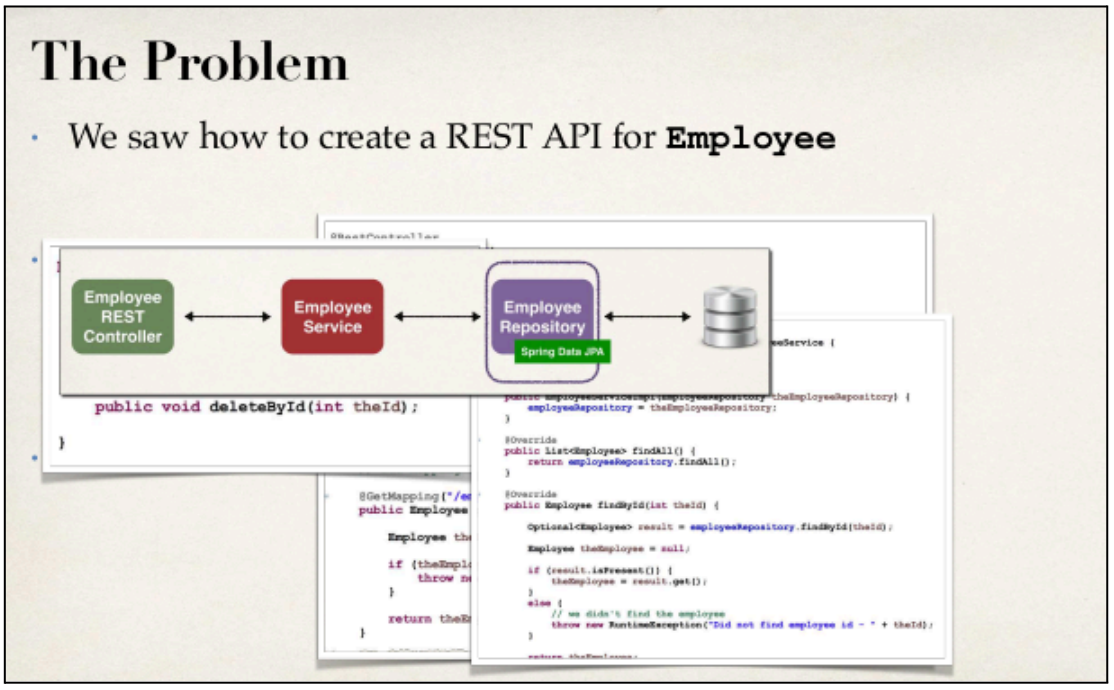
```java
2 usages
@Override
public Employee findById(int theId) {
    Optional<Employee> result = employeeRepository.findById(theId);

    Employee theEmployee = null;

    if (result.isPresent()) {
        theEmployee = result.get();
    }
    else {
        // we didn't find the employee
        throw new RuntimeException("Did not find employee id - " + theId);
    }

    return theEmployee;
}
```

Short ->

```java
2 usages
@Override
public Employee findById(int theId) {
    return employeeRepository.findById(theId).orElse( other: null);
}
```

orElse(null): This is a method of the Optional class. It returns the value contained in the Optional if it is present, or the specified default value (null in this case) if the Optional is empty.

Spring Data REST in Spring Boot

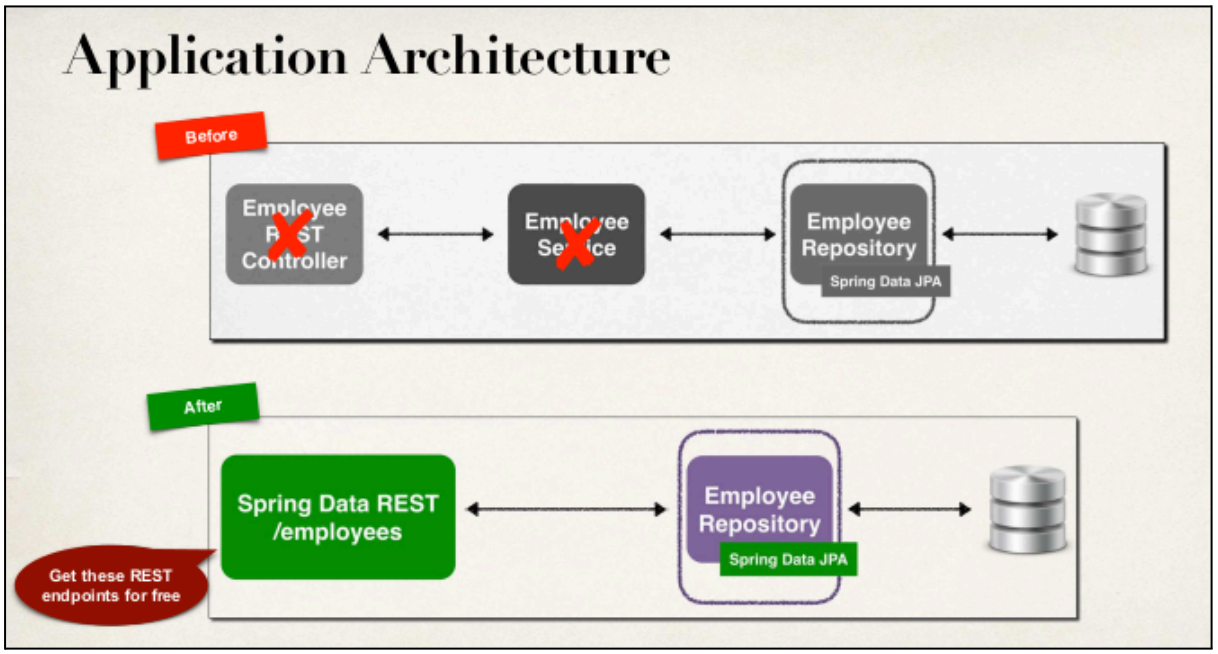


Spring Data REST - Solution

- Leverages your existing JpaRepository
- Spring will give you a REST CRUD implementation for FREE …. Like MAGIC!!
- Helps to minimize boiler-plate REST code!!!
- No new coding required!!!

Add Spring Data REST to your Maven POM file
For Spring Data REST, you only need 3 items,

1. Your entity: Employee
2. JpaRepository: EmployeeRepository extends JpaRepository
3. Maven POM dependency for: spring-boot-starter-data-rest



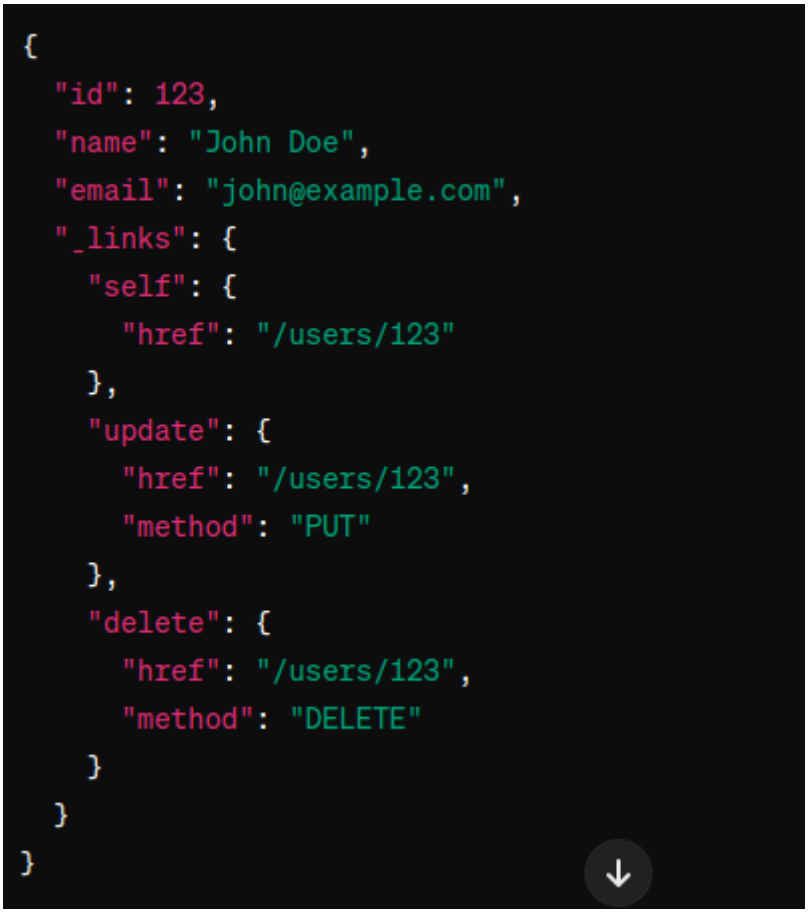
HATEOAS

HATEOAS stands for "Hypermedia as the Engine of Application State." It's a principle in RESTful API design that aims to enable clients to navigate a web API's resources dynamically by including hypermedia links in the responses.

In simpler terms, with HATEOAS, a server provides not only the data requested by the client but also additional information about related resources and possible actions that can be taken next. This additional information is typically provided in the form of hyperlinks embedded within the response.

Here's an example to illustrate how HATEOAS works:

Suppose you have an API for managing users. When a client requests information about a specific user, instead of just returning the user's data, the server response might include hyperlinks to related resources or actions, such as:



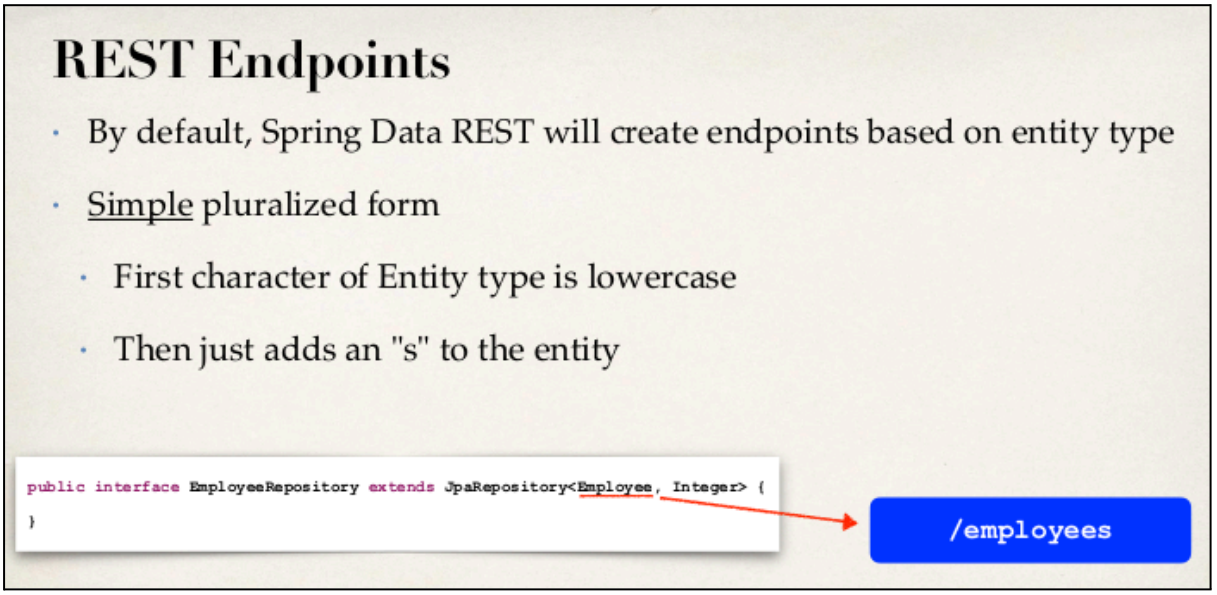
In this example:

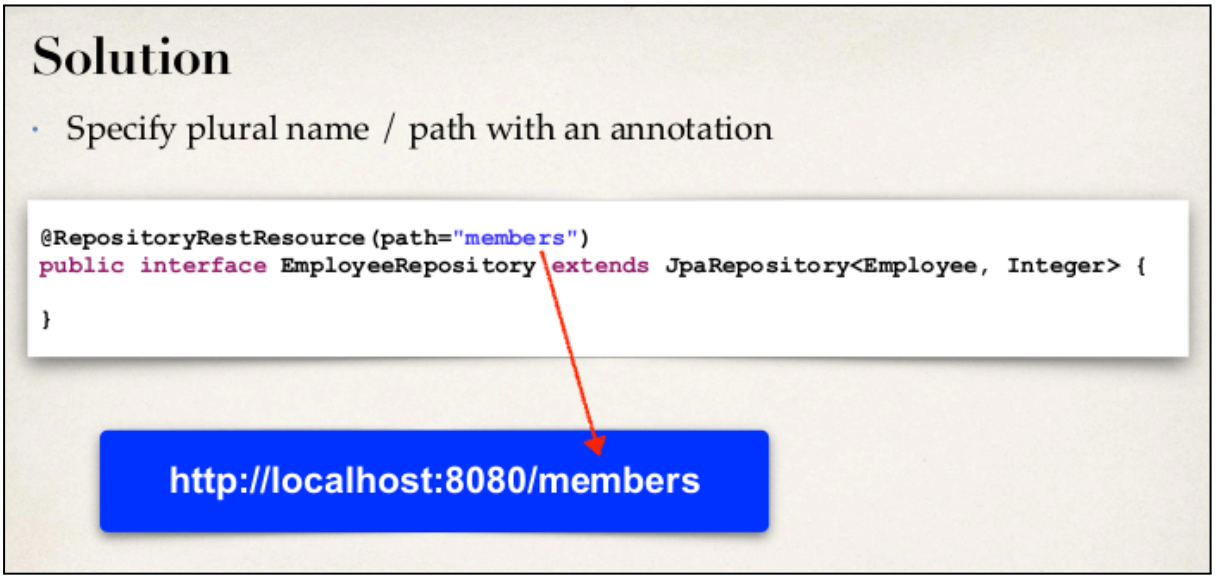
- - The `"self"` link provides the URL to retrieve the current user's details.
- - The `"update"` link provides the URL and HTTP method (`PUT`) to update the user's information.
- - The `"delete"` link provides the URL and HTTP method (`DELETE`) to delete the user.

By providing these hypermedia links dynamically in the API responses, clients can navigate the API more easily without relying on fixed URLs or predefined endpoints. This makes the API more flexible, discoverable, and self-descriptive, leading to improved client-server decoupling and scalability.

HATEOAS is considered one of the constraints of RESTful APIs and is emphasized in the Richardson Maturity Model, which defines different levels of RESTfulness. APIs that fully implement HATEOAS are often referred to as "Level 3 RESTful" or "HATEOAS-driven" APIs.

Spring Data REST Configuration, Pagination and Sorting

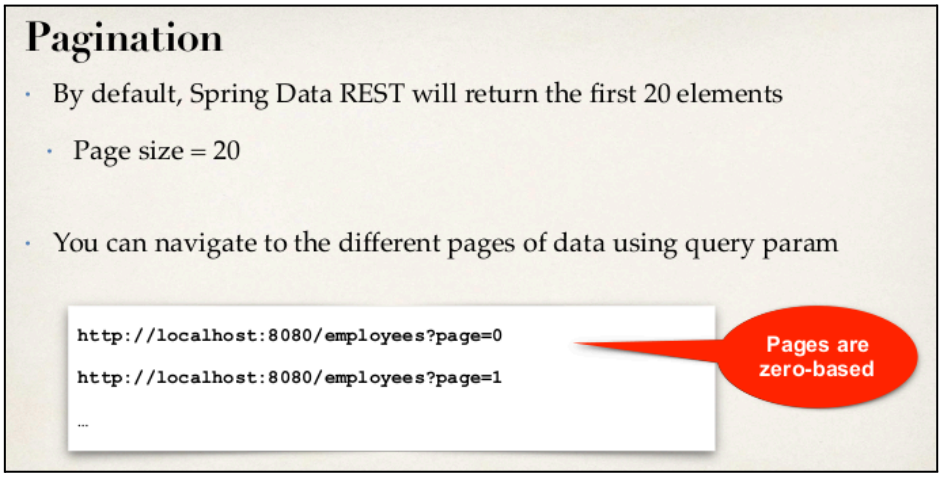


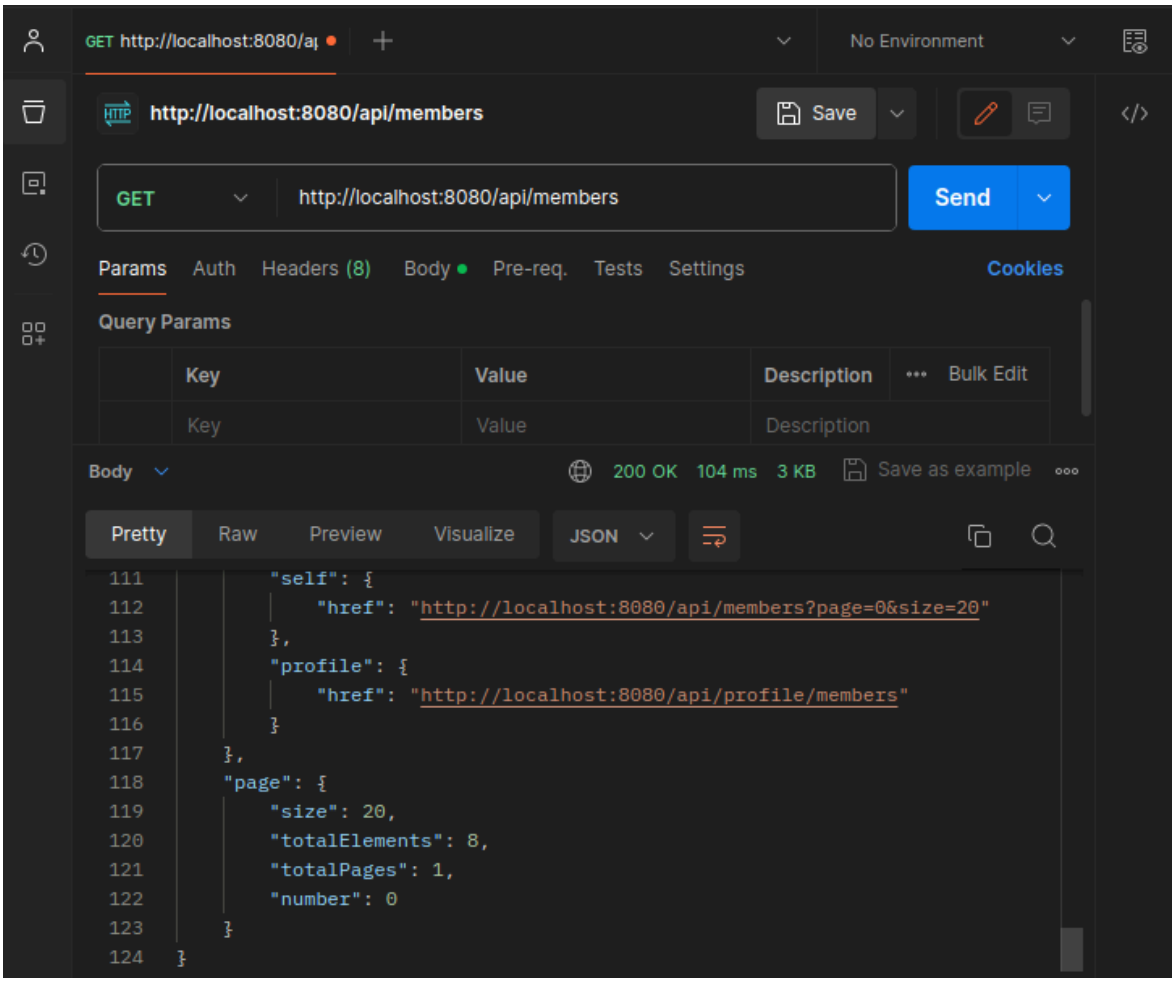


@RepositoryRestResource

@RepositoryRestResource is an annotation provided by Spring Data REST, which is a part of the larger Spring Data project. This annotation is used to expose Spring Data repositories as RESTful endpoints automatically.

When you annotate a Spring Data repository interface with @RepositoryRestResource, Spring Data REST automatically creates RESTful endpoints for that repository, allowing clients to perform CRUD (Create, Read, Update, Delete) operations on the underlying entities over HTTP.





Spring Data REST Configuration

- Following properties available: application.properties

Name	Description
`spring.data.rest.base-path`	Base path used to expose repository resources
`spring.data.rest.default-page-size`	Default size of pages
`spring.data.rest.max-page-size`	Maximum size of pages
...	...

More properties available

www.luv2code.com/spring-boot-props

`spring.data.rest.*`

Sample Configuration

http://localhost:8080/magic-api/employees

File: application.properties

```
spring.data.rest.base-path=/magic-api

spring.data.rest.default-page-size=50
```

Returns 50 elements per page

Sorting

- You can sort by the property names of your entity
  - In our Employee example, we have: **`firstName`**, **`lastName`** and **`email`**
- Sort by last name (ascending is default) `http://localhost:8080/employees?sort=lastName`
- Sort by first name, descending `http://localhost:8080/employees?sort=firstName,desc`
- Sort by last name, then first name, ascending `http://localhost:8080/employees?sort=lastName,firstName,asc`

Spring Boot REST API Security Overview

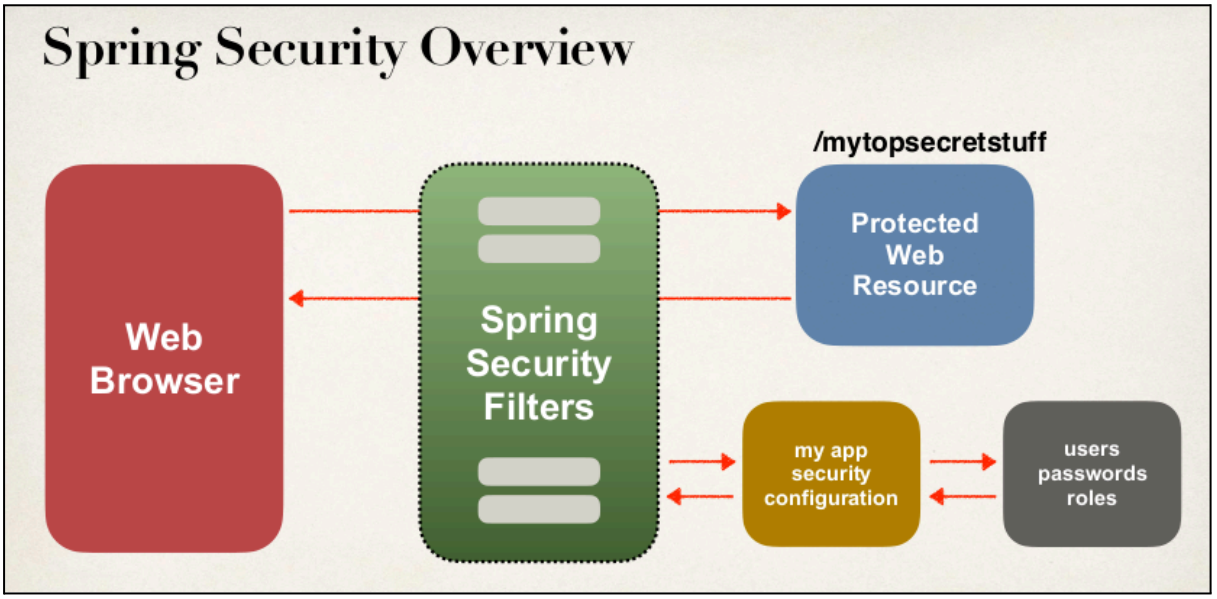
- [ ] Secure Spring Boot REST APIs
- [ ] Define users and roles
- [ ] Protect URLs based on role
- [ ] Store users, passwords and roles in DB (plain-text -> encrypted)

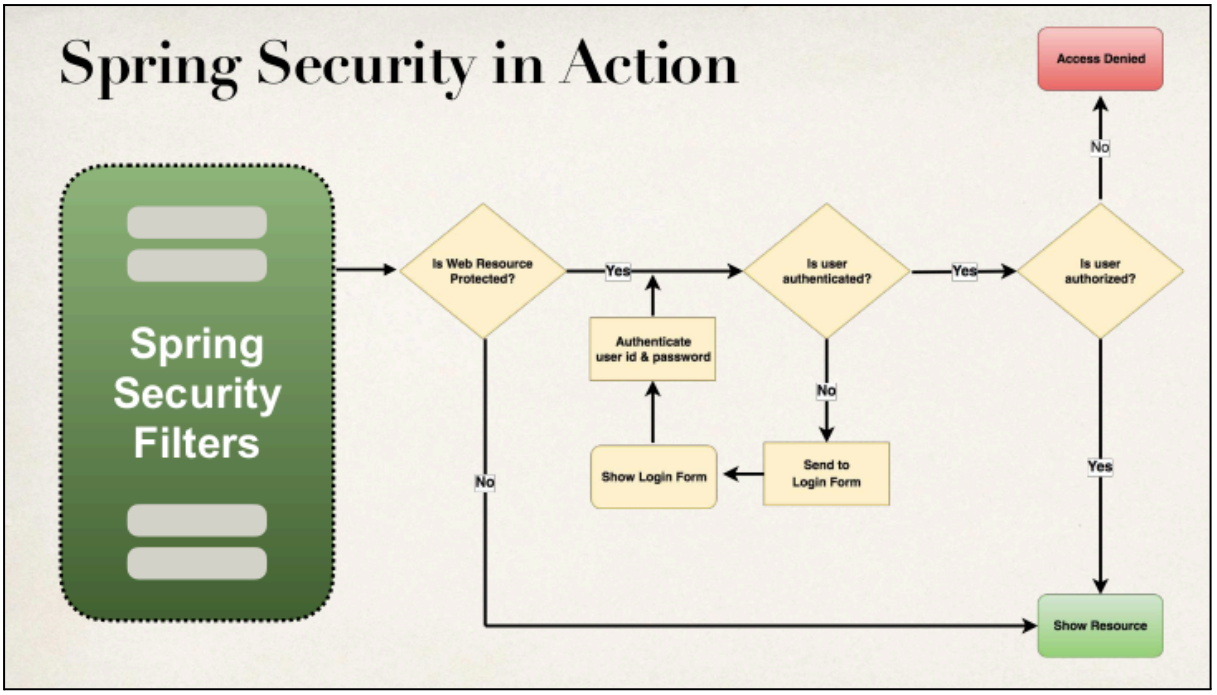
Spring Security Model

- Spring Security defines a framework for security
- Implemented using Servlet filters in the background
- Two methods of securing an app: declarative and programmatic

Spring Security with Servlet Filters

- Servlet Filters are used to pre-process / post-process web requests
- Servlet Filters can route web requests based on security logic
- Spring provides a bulk of security functionality with servlet filters





Security Concepts,

Authentication
Check user id and password with credentials stored in app / db

Authorization
Check to see if user has an authorized role

Declarative Security

- Define application's security constraints in configuration
    - All Java config: @Configuration
- Provides separation of concerns between application code and security

Programmatic Security

- Spring Security provides an API for custom application coding
- Provides greater customization for specific app requirements

Declarative Security vs Programmatic Security

Declarative security and programmatic security are two approaches to implementing security in software applications, each with its own advantages and use cases:

1. Declarative Security :

- Definition : Declarative security involves specifying security requirements and access control rules using metadata or configuration rather than writing code.

- Usage : Declarative security is commonly used in frameworks and platforms that support it, such as Java EE, Spring Security, and ASP.NET Core Identity. It allows developers to define security constraints in configuration files or annotations.

- Advantages :
    - Separation of Concerns : Security concerns are separated from application logic, promoting cleaner and more maintainable code.
    - Ease of Configuration : Security policies can be defined and updated without modifying the application code, making it easier to manage.
    - Visibility : Security constraints are often clearly visible in configuration files or annotations, making it easier to understand the application's security requirements.

- Example : In Spring Security, you can use annotations like `@Secured`, `@PreAuthorize`, or XML-based configuration to specify access control rules for methods or URLs.

2. Programmatic Security :

- Definition : Programmatic security involves writing code to enforce security policies and access control rules within the application.

- Usage : Programmatic security is often used when fine-grained control over security behavior is required, or when declarative security mechanisms are insufficient for the application's needs.

- Advantages :
    - Flexibility : Programmatic security allows developers to implement complex security logic and dynamic access control based on runtime conditions.
    - Customization : Developers have full control over the implementation details and can tailor security behavior to specific requirements.
    - Dynamic Behavior : Security policies can be adapted dynamically based on runtime conditions or user interactions.

- Example : In Java EE applications, programmatic security can be implemented using the `javax.security.enterprise` API or by directly invoking security-related methods such as `HttpServletRequest#isUserInRole()`.

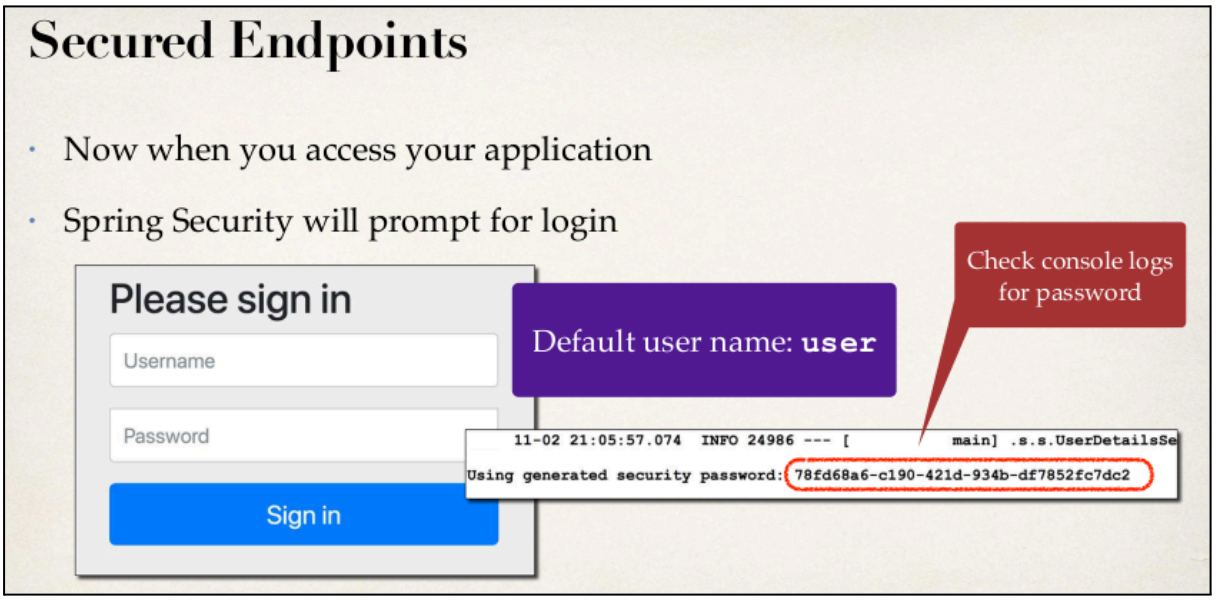
In practice, both declarative and programmatic security approaches may be used together in an application, depending on the specific requirements and constraints. Declarative security is often preferred for common and static security requirements, while programmatic security is used for more complex scenarios that cannot be easily expressed declaratively.

Enabling Spring Security

1. Edit pom.xml and add spring-boot-starter-security

```
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-security</artifactId>
</dependency>
```

2.This will automagically secure all endpoints for application



Spring Security configuration

- You can override default user name and generated password

File: src/main/resources/application.properties

```
spring.security.user.name=scott
spring.security.user.password=test123
```

Authentication and authorization are crucial aspects of application security, ensuring that users are who they claim to be and that they have appropriate access rights to resources within the application. Various mechanisms and strategies can be employed to implement authentication and authorization, including:

1. In-Memory Authentication and Authorization :

- Authentication : User credentials (e.g., username and password) are stored in memory, typically configured statically within the application.

- Authorization : Role-based access control (RBAC) is commonly used, where users are assigned roles, and access rights are granted to these roles.

2. JDBC (Database) Authentication and Authorization :

- Authentication : User credentials are stored in a relational database, and authentication is performed by querying the database.

- Authorization : Access control lists (ACLs) or RBAC can be implemented using database tables to define user-role mappings and resource permissions.

3. LDAP (Lightweight Directory Access Protocol) Authentication and Authorization :

- Authentication : User credentials are stored in an LDAP directory server (e.g., Active Directory, OpenLDAP), and authentication is performed by querying the LDAP server.

- Authorization : LDAP directories often support hierarchical access control mechanisms, allowing fine-grained control over resource permissions based on user attributes and group memberships.

4. Custom / Pluggable Authentication and Authorization :

- Authentication : Custom authentication mechanisms can be implemented to authenticate users using various methods such as OAuth, OpenID Connect, JWT (JSON Web Tokens), or third-party identity providers.

- Authorization : Custom authorization logic can be implemented based on application-specific requirements, such as business rules or dynamic access control policies.

5. Others :

- OAuth and OpenID Connect : These are widely used protocols for authentication and authorization in distributed systems, particularly in the context of single sign-on (SSO) and identity federation.

- SAML (Security Assertion Markup Language) : Another protocol used for exchanging authentication and authorization data between identity providers and service providers.

- Token-based Authentication : This approach involves issuing and validating tokens (e.g., JWT) for authentication, often combined with OAuth or custom authentication mechanisms.

The choice of authentication and authorization mechanism depends on factors such as security requirements, scalability, integration capabilities, and compliance with standards and regulations. In many cases, a combination of different mechanisms may be used to address various authentication and authorization scenarios within an application or across a distributed system.

Configuring Basic Security

User ID	Password	Roles
john	test123	EMPLOYEE
mary	test123	EMPLOYEE, MANAGER
susan	test123	EMPLOYEE, MANAGER, ADMIN

We can give ANY names for user roles

Development Process

1. Create Spring Security Configuration (@Configuration)
2. Add users, passwords and roles

Step 1: Create Spring Security Configuration

File: DemoSecurityConfig.java

```
import org.springframework.context.annotation.Configuration;

@Configuration
public class DemoSecurityConfig {

  // add our security configurations here …

}
```

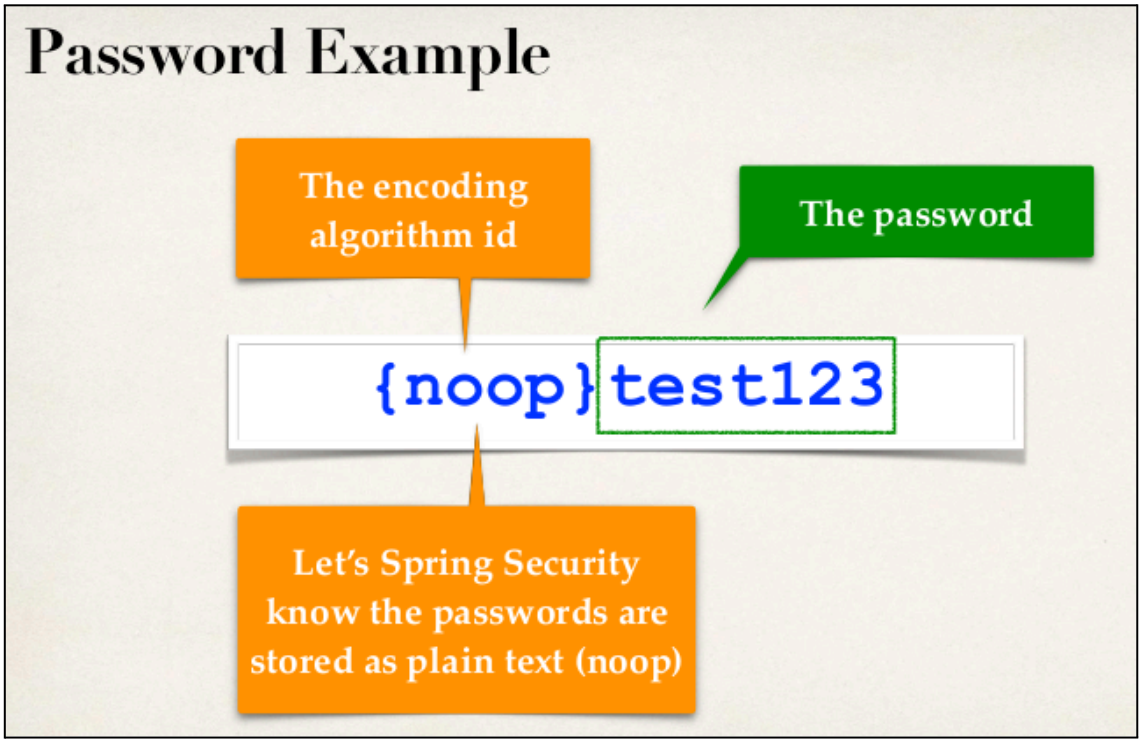
Spring Security Password Storage

- In Spring Security, passwords are stored using a specific format

```
{id}encodedPassword
```

ID	Description
noop	Plain text passwords
bcrypt	BCrypt password hashing
...	...

Password Example



Step 2: Add users, passwords and roles

File: DemoSecurityConfig.java

```java
@Configuration
public class DemoSecurityConfig {

    @Bean
    public InMemoryUserDetailsManager userDetailsManager() {

        UserDetails john = User.builder()
                .username("john")
                .password("{noop}test123")
                .roles("EMPLOYEE")
                .build();

        UserDetails mary = User.builder()
                .username("mary")
                .password("{noop}test123")
                .roles("EMPLOYEE", "MANAGER")
                .build();

        UserDetails susan = User.builder()
                .username("susan")
                .password("{noop}test123")
                .roles("EMPLOYEE", "MANAGER", "ADMIN")
                .build();

        return new InMemoryUserDetailsManager(john, mary, susan);
    }
}
```

User ID	Password	Roles
john	test123	EMPLOYEE
mary	test123	EMPLOYEE, MANAGER
susan	test123	EMPLOYEE, MANAGER, ADMIN

We will add DB support in later videos

(plaintext and encrypted)

==========

Basic Authentication ->

SecurityConfig.java

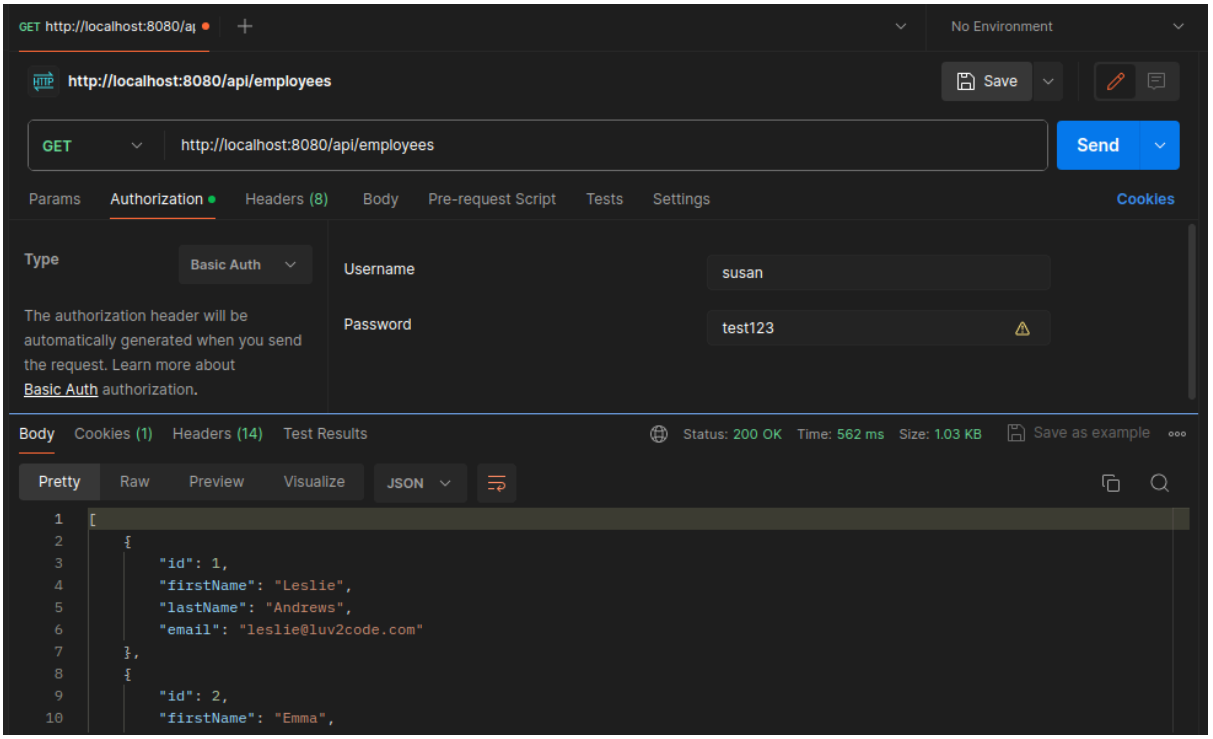
```java
package com.example.secuiritybasicdemo.security;

import ...

@EnableWebSecurity
@Configuration
public class SecurityConfig {

//    @Bean
//    public PasswordEncoder passwordEncoder() {
//        return new BCryptPasswordEncoder();
//    }

}
```

SecurityConfig.java DemoSecurityConfig.java

```java
@Configuration
public class DemoSecurityConfig {

    @Bean
    public PasswordEncoder passwordEncoder() {
        return new BCryptPasswordEncoder();
    }

    @Bean
    public InMemoryUserDetailsManager userDetailsManager() {
        PasswordEncoder passwordEncoder = passwordEncoder();

        UserDetails john = User.builder()
                .username("john")
                .password(passwordEncoder.encode( rawPassword: "test123"))
                .roles("EMPLOYEE")
                .build();

        UserDetails susan = User.builder()
                .username("susan")
                .password(passwordEncoder.encode( rawPassword: "test123"))
                .roles("EMPLOYEE", "MANAGER", "ADMIN")
                .build();

        return new InMemoryUserDetailsManager(john,  susan);
    }
}
```

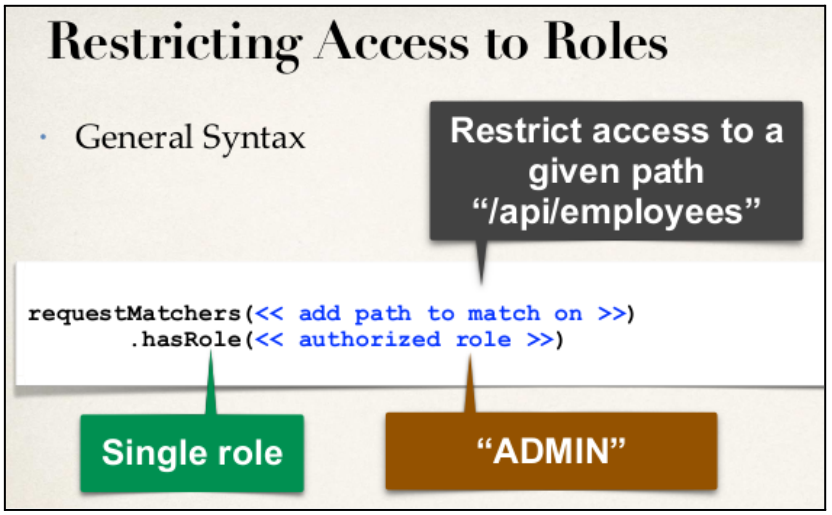
application.properties

```
spring.datasource.url=jdbc:mysql://localhost:3306/employee_directory
spring.datasource.username=springstudent
spring.datasource.password=springstudent


spring.security.user.name=ruvini
spring.security.user.password=1234


spring.main.allow-bean-definition-overriding=true

```



Restrict Access Based on Roles

HTTP Method	Endpoint	CRUD Action	Role
GET	/api/employees	Read all	EMPLOYEE
GET	/api/employees/{employeeId}	Read single	EMPLOYEE
POST	/api/employees	Create	MANAGER
PUT	/api/employees	Update	MANAGER
DELETE	/api/employees/{employeeId}	Delete employee	ADMIN



Restricting Access to Roles

Specify HTTP method: GET, POST, PUT, DELETE ...

Restrict access to a given path "/api/employees"

```
requestMatchers(<< add HTTP METHOD to match on >>, << add path to match on >>)
        .hasRole(<< authorized roles >>)
```

Single role

Restricting Access to Roles

```
requestMatchers(<< add HTTP METHOD to match on >>, << add path to match on >>)
    .hasAnyRole(<< list of authorized roles >>)
```

Any role

Comma-delimited list

Authorize Requests for EMPLOYEE role

```
requestMatchers(HttpMethod.GET, "/api/employees").hasRole("EMPLOYEE")
requestMatchers(HttpMethod.GET, "/api/employees/**").hasRole("EMPLOYEE")
```

HTTP Method	Endpoint	CRUD Action	Role
GET	/api/employees	Read all	EMPLOYEE
GET	/api/employees/{employeeId}	Read single	EMPLOYEE
POST	/api/employees	Create	MANAGER
PUT	/api/employees	Update	MANAGER
DELETE	/api/employees/{employeeId}	Delete employee	ADMIN

The ** syntax: match on all sub-paths

Authorize Requests for MANAGER role

```
requestMatchers(HttpMethod.POST, "/api/employees").hasRole("MANAGER")
requestMatchers(HttpMethod.PUT, "/api/employees").hasRole("MANAGER")
```

HTTP Method	Endpoint	CRUD Action	Role
GET	/api/employees	Read all	EMPLOYEE
GET	/api/employees/{employeeId}	Read single	EMPLOYEE
POST	/api/employees	Create	MANAGER
PUT	/api/employees	Update	MANAGER
DELETE	/api/employees/{employeeId}	Delete employee	ADMIN

Authorize Requests for ADMIN role

```
requestMatchers(HttpMethod.DELETE, "/api/employees/**").hasRole("ADMIN")
```

HTTP Method	Endpoint	CRUD Action	Role
GET	/api/employees	Read all	EMPLOYEE
GET	/api/employees/{employeeId}	Read single	EMPLOYEE
POST	/api/employees	Create	MANAGER
PUT	/api/employees	Update	MANAGER
DELETE	/api/employees/{employeeId}	Delete employee	ADMIN

Pull It Together

User ID	Password	Roles
john	test123	EMPLOYEE
mary	test123	EMPLOYEE, MANAGER
susan	test123	EMPLOYEE, MANAGER, ADMIN

```
@Bean
public SecurityFilterChain filterChain(HttpSecurity http) throws Exception {

    http.authorizeHttpRequests(configurer ->
                configurer
                        .requestMatchers(HttpMethod.GET, "/api/employees").hasRole("EMPLOYEE")
                        .requestMatchers(HttpMethod.GET, "/api/employees/**").hasRole("EMPLOYEE")
                        .requestMatchers(HttpMethod.POST, "/api/employees").hasRole("MANAGER")
                        .requestMatchers(HttpMethod.PUT, "/api/employees").hasRole("MANAGER")
                        .requestMatchers(HttpMethod.DELETE, "/api/employees/**").hasRole("ADMIN"));

    // use HTTP Basic authentication
    http.httpBasic(Customizer.withDefaults());

    return http.build();
}
```

Use HTTP Basic Authentication

HTTP Method	Endpoint	CRUD Action	Role
GET	/api/employees	Read all	EMPLOYEE
GET	/api/employees/{employeeId}	Read single	EMPLOYEE
POST	/api/employees	Create	MANAGER
PUT	/api/employees	Update	MANAGER
DELETE	/api/employees/{employeeId}	Delete employee	ADMIN

Stateless REST API
A stateless REST API, in the context of web services, refers to an architectural style where each request from a client to the server must contain all the information needed by the server to fulfill the request. In other words, the server does not store any information about the client session between requests. Each request is treated independently and is self-contained.

Cross-Site Request Forgery (CSRF)

- Spring Security can protect against CSRF attacks
- Embed additional authentication data/token into all HTML forms
- On subsequent requests, web app will verify token before processing
- Primary use case is traditional web applications (HTML forms etc …)

Cross-Site Request Forgery (CSRF) is a type of security vulnerability that occurs when an attacker tricks a user into performing unwanted actions on a web application where the user is authenticated. CSRF attacks exploit the trust that a web application has in a user's browser by sending unauthorized requests from the user's browser without the user's knowledge.

Here's how a CSRF attack typically works:

1. Authentication : The victim user logs into a web application and receives an authentication token or session cookie.

2. Malicious Request : While the victim user is still authenticated, they visit a malicious website or click on a malicious link. This triggers a request to the vulnerable web application, sending a forged request that performs an action on behalf of the victim.

3. Unauthorized Action : The web application receives the forged request and processes it as if it were a legitimate request from the authenticated user. As a result, the attacker can perform actions such as changing the user's password, making unauthorized purchases, or modifying account settings.

CSRF attacks can target any action that a user is able to perform while authenticated, including changing account settings, making purchases, or submitting forms. They are particularly dangerous because they exploit the trust between a user and a web application, and the user may not be aware that the attack is taking place.

To mitigate CSRF attacks, web developers can implement various security measures, including:

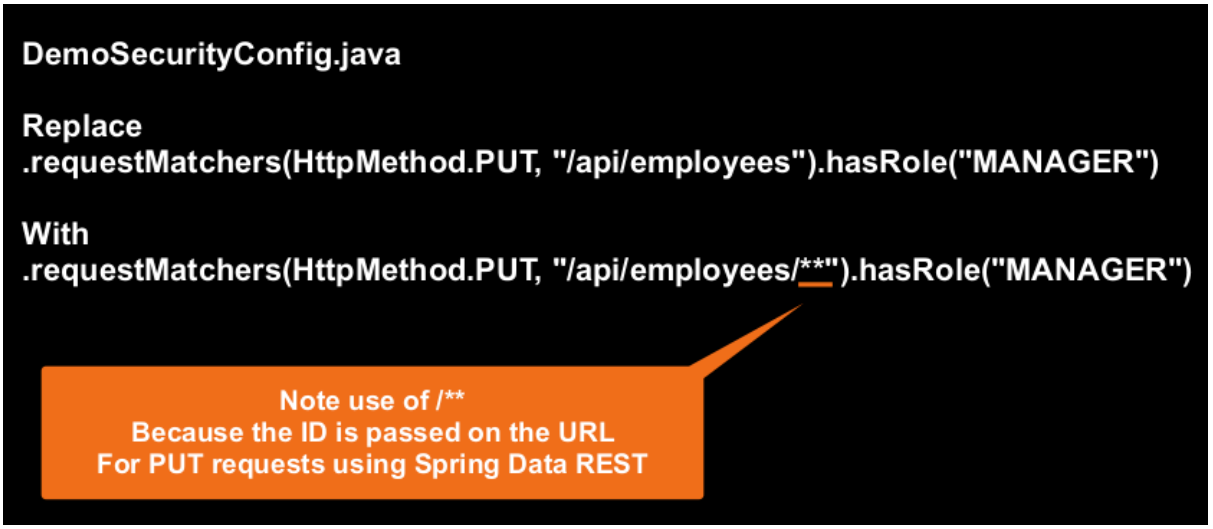
1. CSRF Tokens : Include a unique token in each form or request that is tied to the user's session. This token must be submitted along with the request, and the server verifies its authenticity before processing the request.

2. SameSite Cookies : Set the SameSite attribute on cookies to prevent them from being sent in cross-origin requests, reducing the risk of CSRF attacks.

3. Referrer Policy : Configure the web application to use a strict Referrer Policy, which controls how much information is included in the Referer header of outgoing requests, thereby limiting the impact of CSRF attacks.

4. Authentication Patterns : Implement strong authentication mechanisms, such as multi-factor authentication (MFA), to reduce the likelihood of unauthorized access even if CSRF attacks are successful.

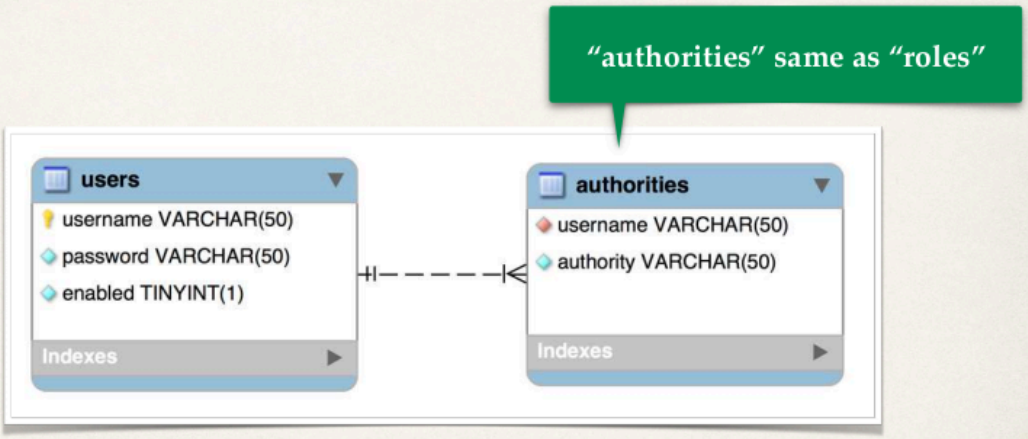
By implementing these best practices and staying informed about emerging security threats, web developers can effectively mitigate the risk of CSRF attacks and protect the integrity of their web applications.



Spring Security User Accounts Stored in Database

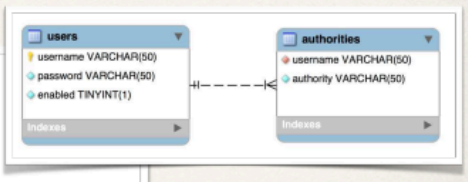
Development Process
1. Develop SQL Script to set up database tables
2. Add database support to Maven POM file
3.Create JDBC properties file
4. Update Spring Security Configuration to use JDBC

Default Spring Security Database Schema



Step 1: Develop SQL Script to setup database tables

```
CREATE TABLE `users` (
  `username` varchar(50) NOT NULL,
  `password` varchar(50) NOT NULL,
  `enabled` tinyint NOT NULL,

  PRIMARY KEY (`username`)

) ENGINE=InnoDB DEFAULT CHARSET=latin1;
```



Step 1: Develop SQL Script to setup database tables

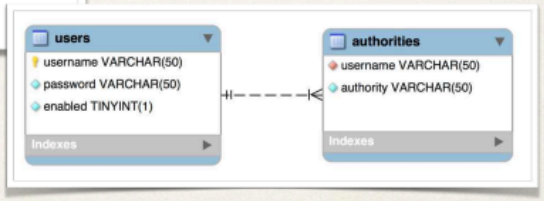
The encoding algorithm id

The password

```
INSERT INTO `users`
VALUES
('john','{noop}test123',1),
('mary','{noop}test123',1),
('susan','{noop}test123',1);
```

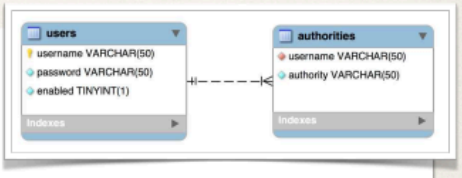
User ID	Password	Roles
john	test123	EMPLOYEE
mary	test123	EMPLOYEE, MANAGER
susan	test123	EMPLOYEE, MANAGER, ADMIN

Let's Spring Security know the passwords are stored as plain text (noop)



Step 1: Develop SQL Script to setup database tables

```
CREATE TABLE `authorities` (
  `username` varchar(50) NOT NULL,
  `authority` varchar(50) NOT NULL,

  UNIQUE KEY `authorities_idx_1` (`username`,`authority`),

  CONSTRAINT `authorities_ibfk_1`
  FOREIGN KEY (`username`)
  REFERENCES `users` (`username`)
) ENGINE=InnoDB DEFAULT CHARSET=latin1;
```



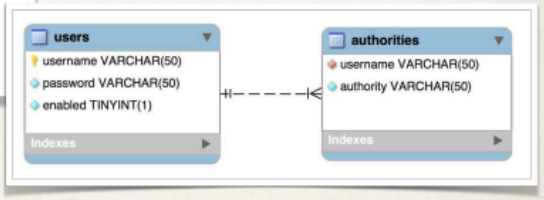
Step 1: Develop SQL Script to setup database tables

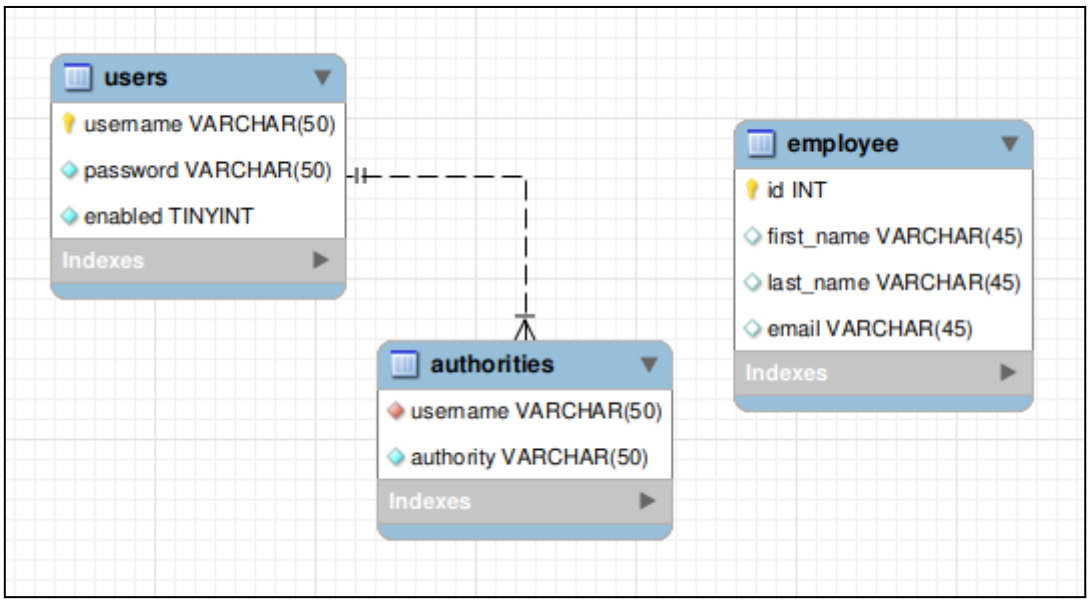
"authorities" same as "roles"

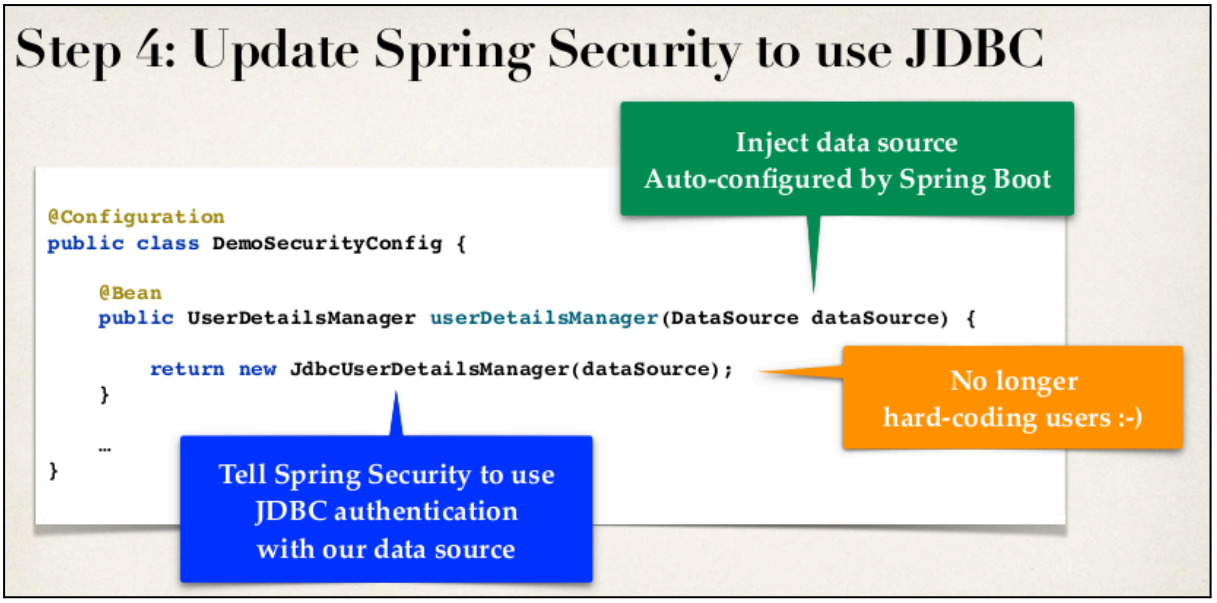
```
INSERT INTO `authorities`
VALUES
('john','ROLE_EMPLOYEE'),
('mary','ROLE_EMPLOYEE'),
('mary','ROLE_MANAGER'),
('susan','ROLE_EMPLOYEE'),
('susan','ROLE_MANAGER'),
('susan','ROLE_ADMIN');
```

Internally Spring Security uses "ROLE_" prefix

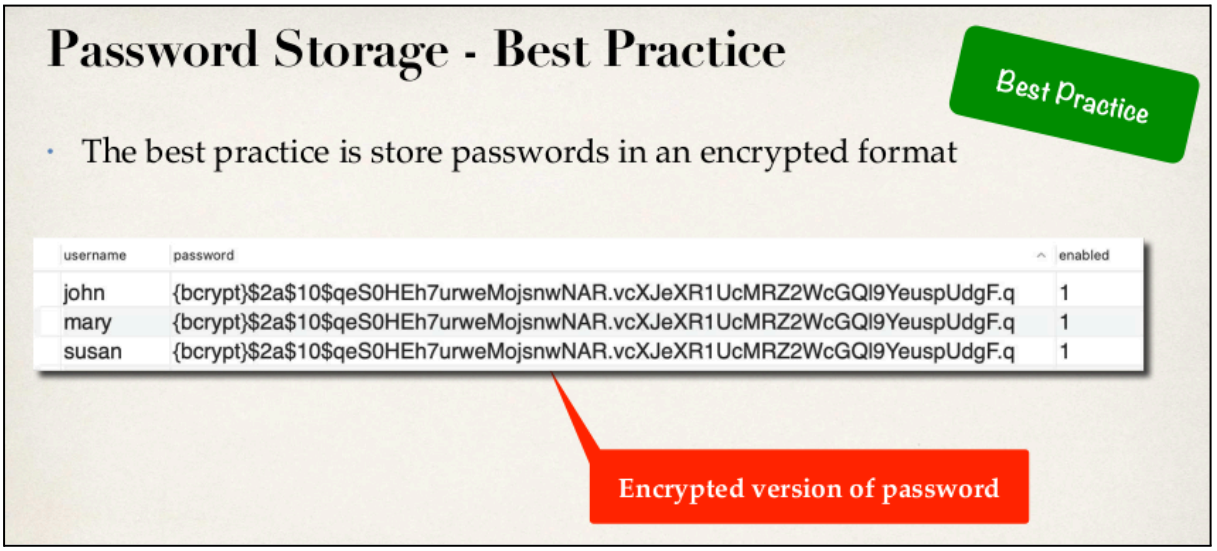
User ID	Password	Roles
john	test123	EMPLOYEE
mary	test123	EMPLOYEE, MANAGER
susan	test123	EMPLOYEE, MANAGER, ADMIN







Spring Security Password Encryption



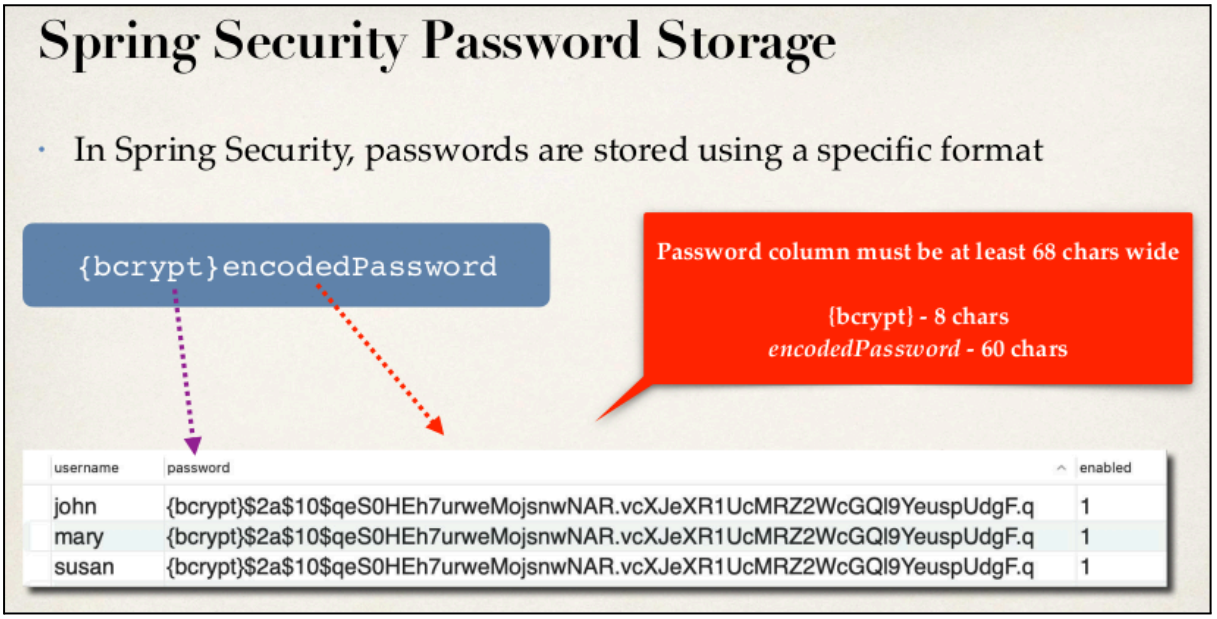
Spring Security Team Recommendation

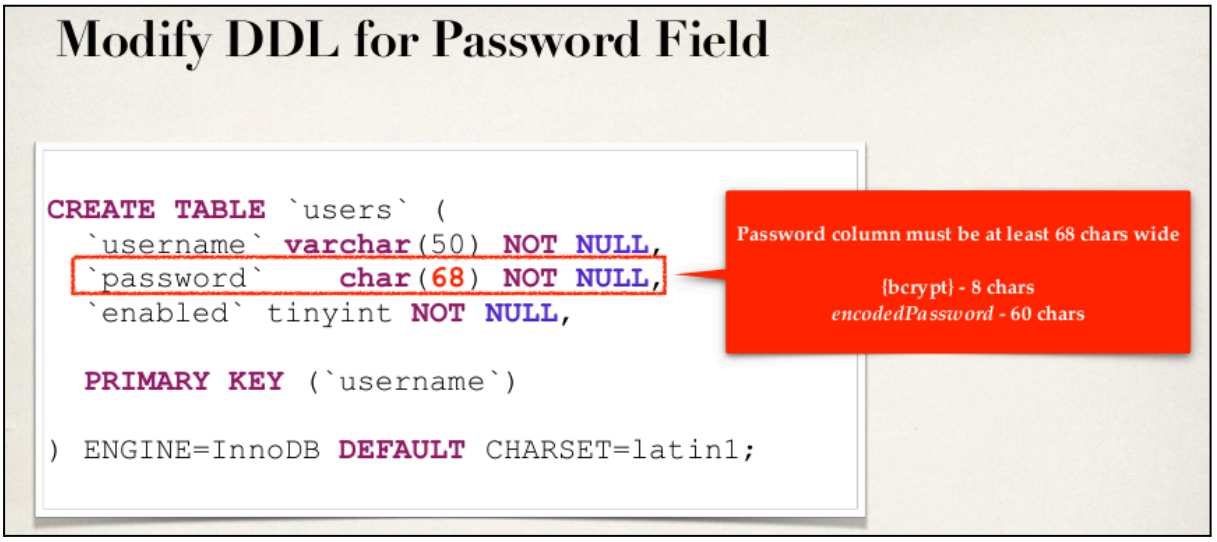
- Spring Security recommends using the popular bcrypt algorithm
- Bcrypt
  - performs one-way encrypted hashing
  - Adds a random salt to the password for additional protection
  - Includes support to defeat brute force attacks

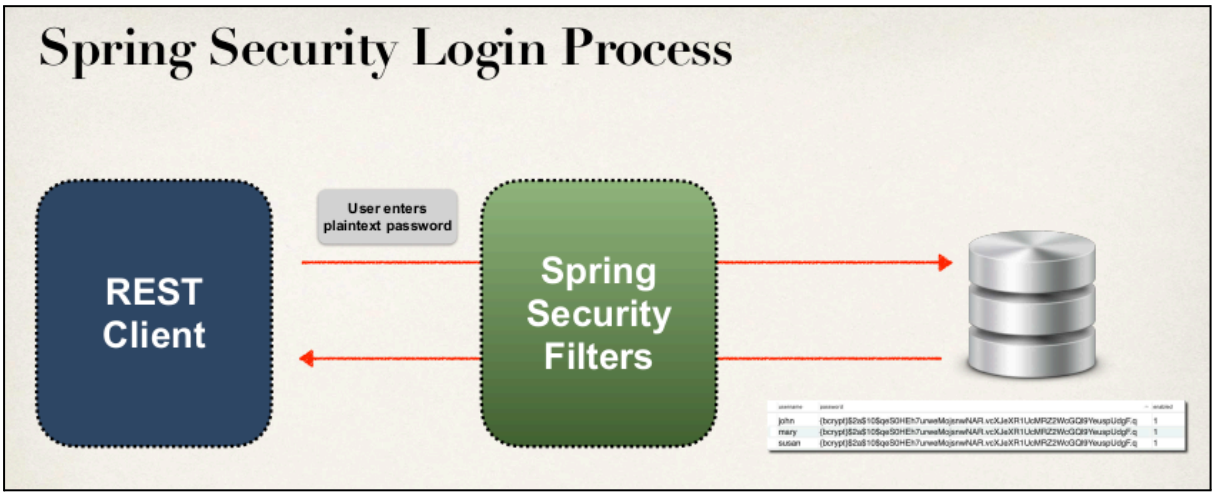
Development Process

1. Run SQL Script that contains encrypted passwords

   Modify DDL for password field, length should be 68

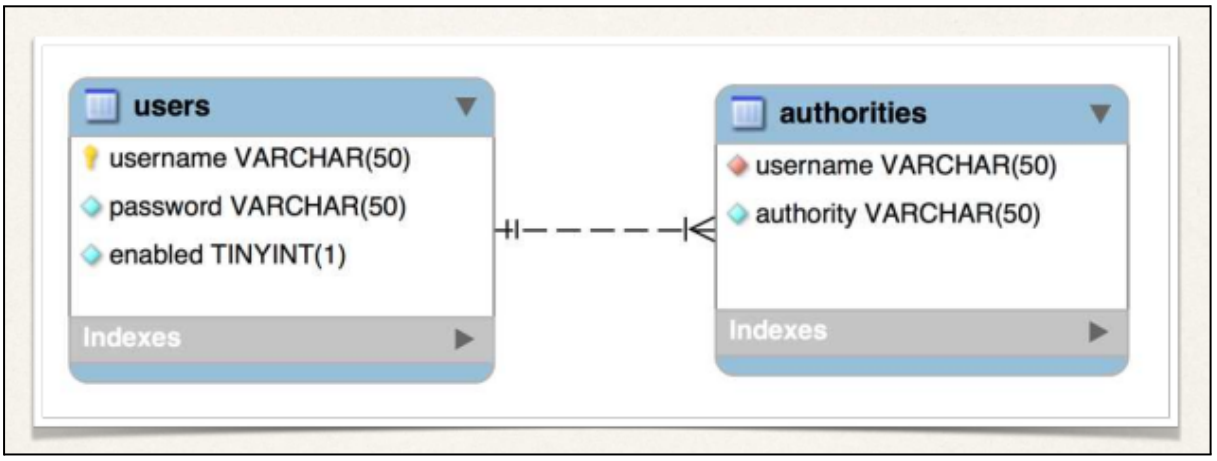


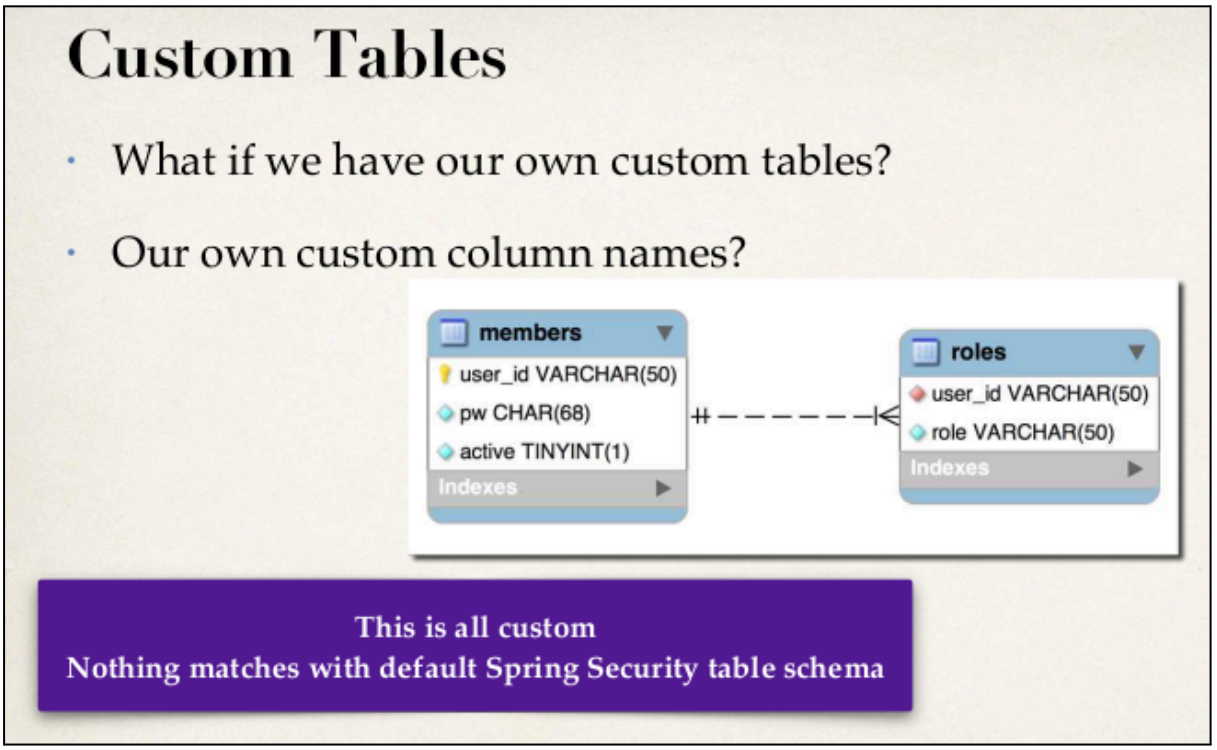




Spring Security Custom Tables

Default Spring security database schema



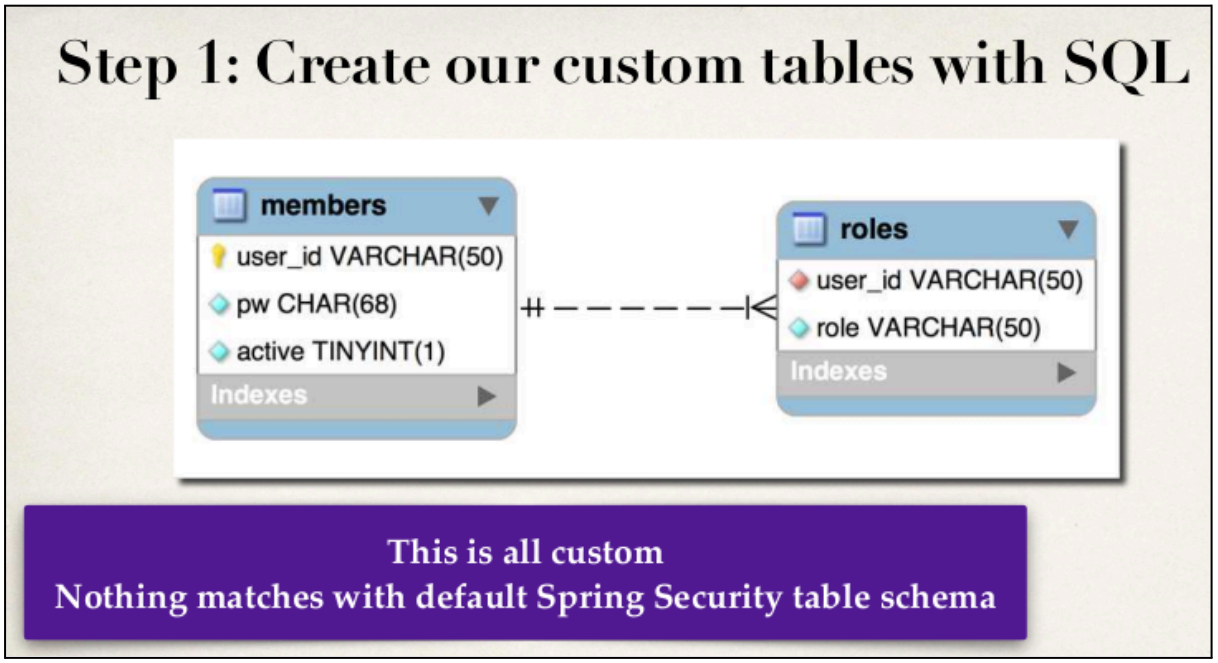


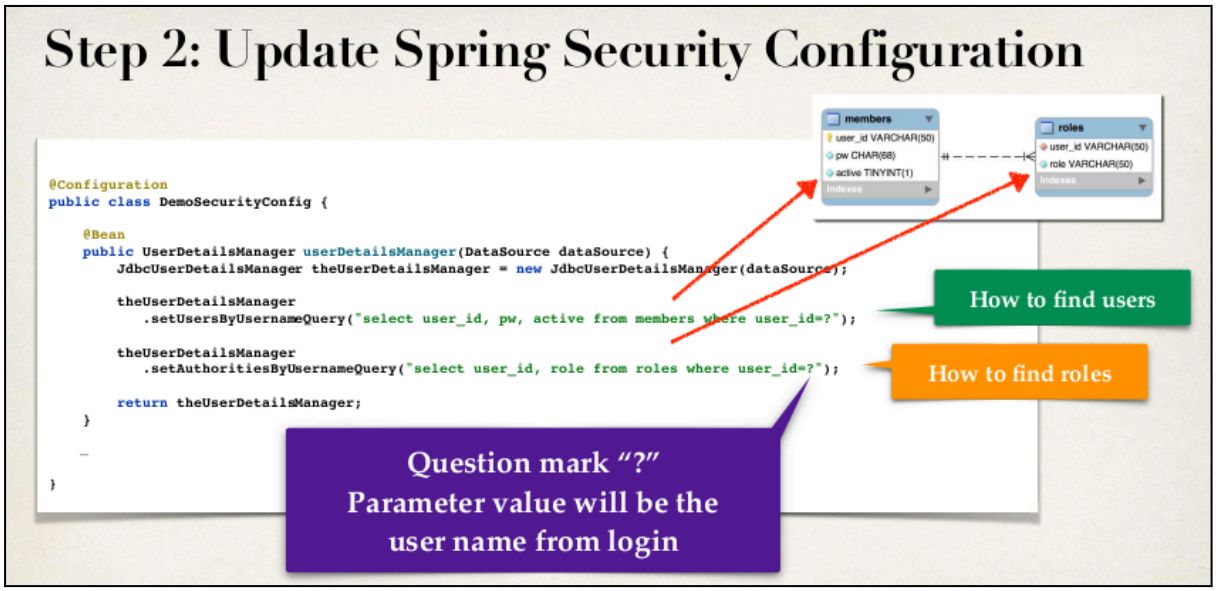
For Security Schema Customization

- Tell Spring how to query your custom tables
- Provide query to find user by user name
- Provide query to find authorities / roles by user name

Development Process

1. Create our custom tables with SQL
2. Update Spring Security Configuration
    - Provide query to find user by user name
    - Provide query to find authorities / roles by user name





More : [Spring Boot REST Security with JPA/Hibernate Tutorial]

Source Code : [spring-boot-rest-security-jpa-hibernate-bcrypt-code]

Thymeleaf with Spring Boot

What is Thymeleaf?

- Thymeleaf is a Java templating engine
- Commonly used to generate the HTML views for web apps
- However, it is a general purpose templating engine
- Can use Thymeleaf outside of web apps (more on this later)

Thymeleaf is a modern server-side Java template engine for web and standalone environments. It's a powerful and flexible tool that allows you to create dynamic web pages by embedding special attributes into your HTML templates.

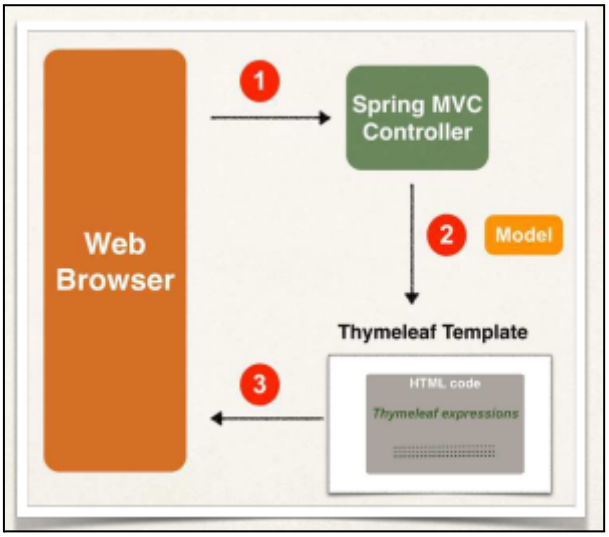
Overall, Thymeleaf is a versatile and developer-friendly template engine that simplifies the process of creating dynamic web pages in Java web applications. Its integration with the Spring Framework and support for modern web development practices make it a popular choice among Java developers.

What is a Thymeleaf template?

- Can be an HTML page with some Thymeleaf expressions
- Include dynamic content from Thymeleaf expressions

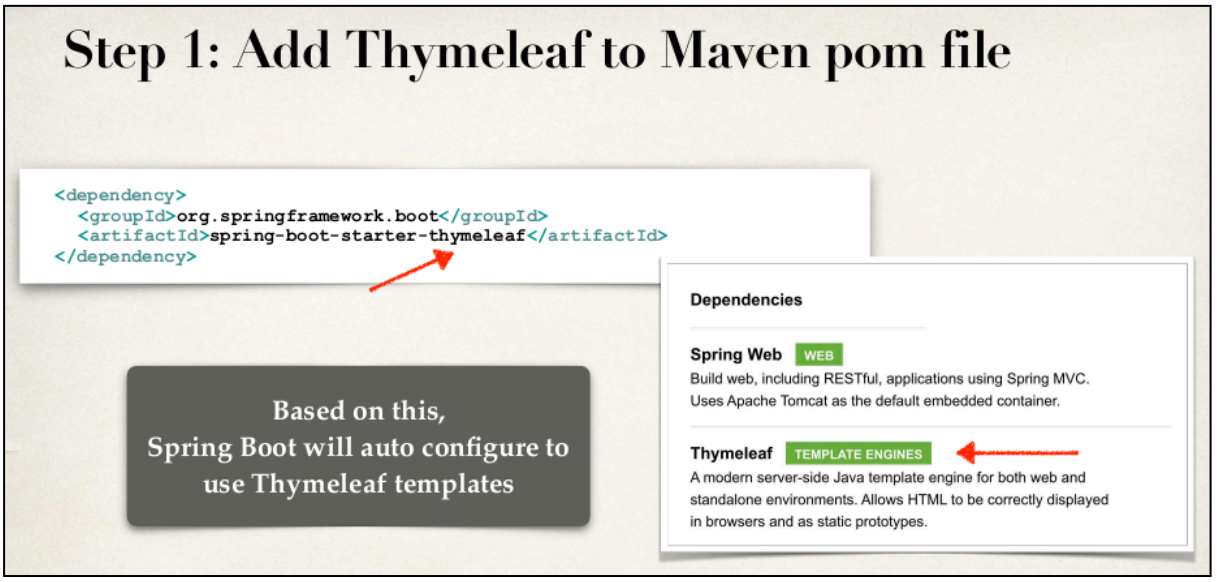
Where is the Thymeleaf template processed?

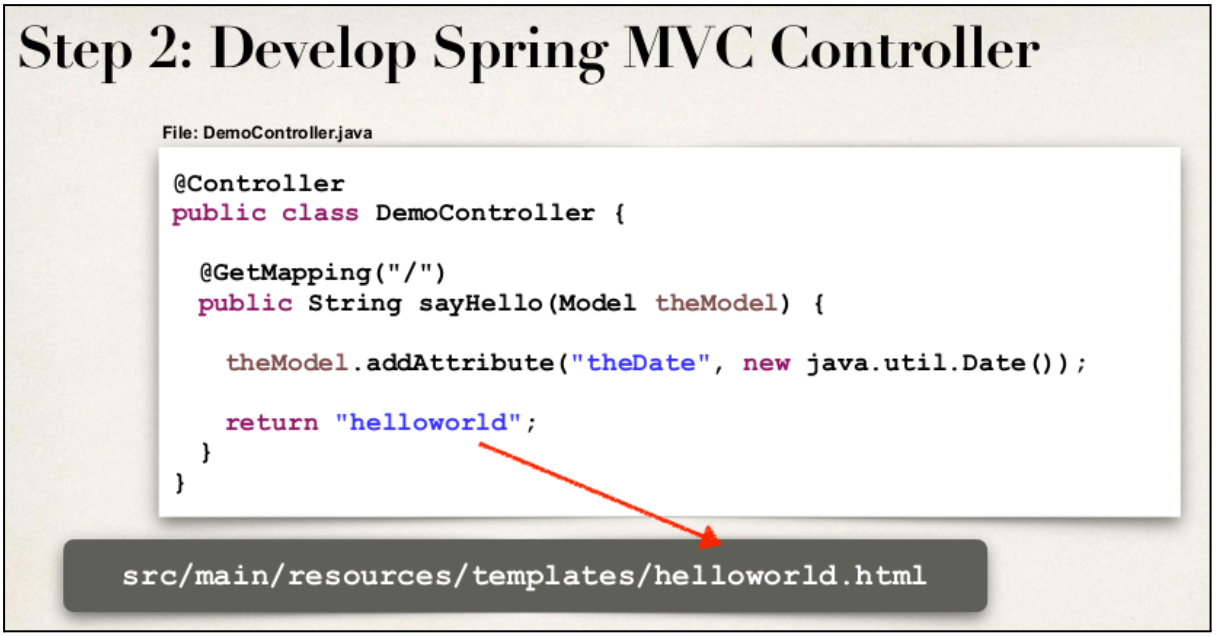
- In a web app, Thymeleaf is processed on the server
- Results included in HTML returned to browser



Development Process

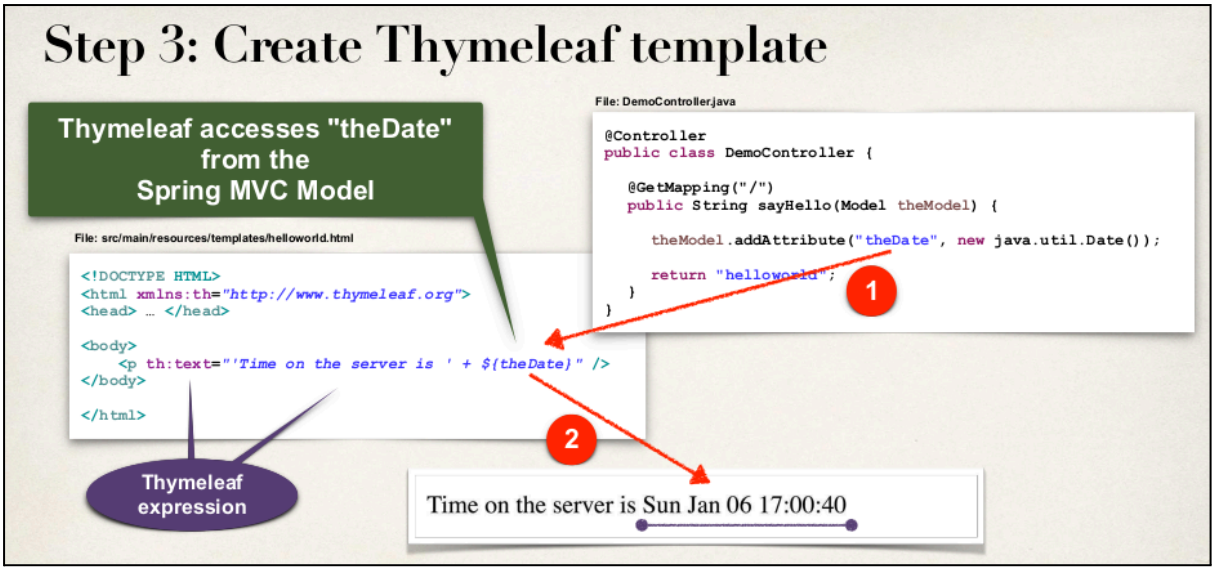
1. Add Thymeleaf to Maven POM file
2. Develop Spring MVC Controller
3. Create Thymeleaf template





Where to place the Thymeleaf template?

- In Spring Boot, your Thymeleaf template files go in
- src/main/resources/templates
- For web apps, Thymeleaf templates have a .html extension



Additional Features

- Looping and conditionals
- CSS and JavaScript integration
- Template layouts and fragments

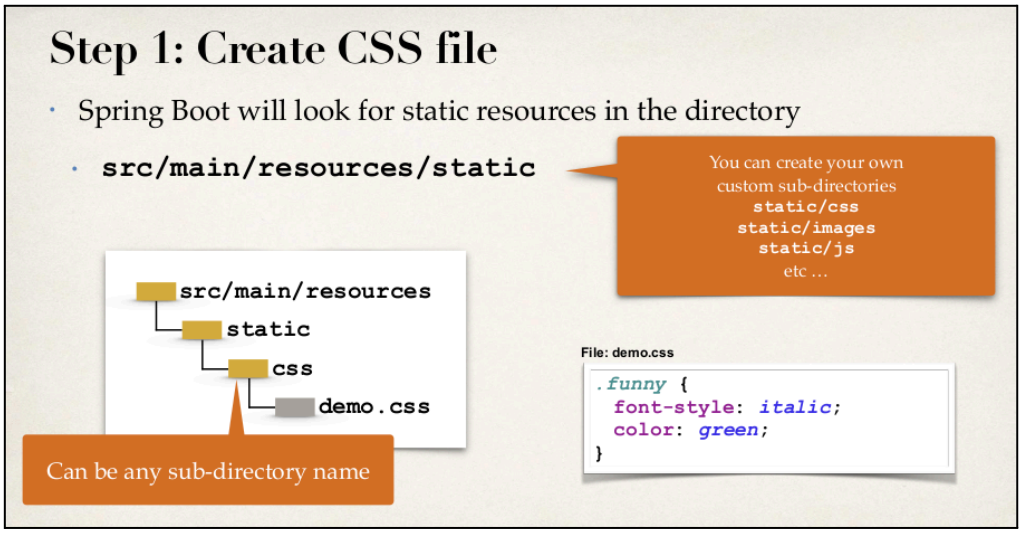
CSS and Thymeleaf

Using CSS with Thymeleaf Templates

- You have the option of using
- Local CSS files as part of your project
- Referencing remote CSS files
- We'll cover both options in this video

Development Process

1. Create CSS file
2. Reference CSS in Thymeleaf template
3. Apply CSS style

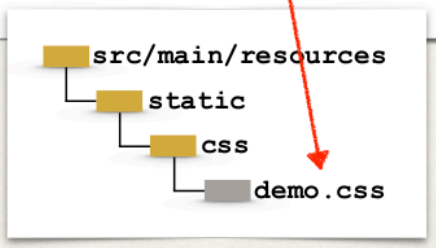


Step 2: Reference CSS in Thymeleaf template

File: helloworld.html

```
<head>
  <title>Thymeleaf Demo</title>

  <!-- reference CSS file -->
  <link rel="stylesheet" th:href="@{/css/demo.css}" />

</head>
```

@ symbol
Reference context path of your application
(app root)



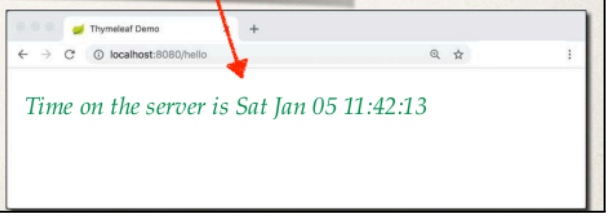
Step 3: Apply CSS

File: helloworld.html

```
<head>
  <title>Thymeleaf Demo</title>

  <!-- reference CSS file -->
  <link rel="stylesheet" th:href="@{/css/demo.css}" />
</head>

<body>
  <p th:text="'Time on the server is ' + ${theDate}" class="funny" />
</body>
```

File: demo.css

```
.funny {
  font-style: italic;
  color: green;
}
```



Other search directories

Spring Boot will search following directories for static resources:

/src/main/resources

1. /META-INF/resources
2. /resources
3. /static
4. /public

Search order: top-down

3rd Party CSS Libraries - Bootstrap

- Local Installation
- Download Bootstrap file(s) and add to **/static/css** directory


```
<head>
  … …

  <!-- reference CSS file -->
  <link rel="stylesheet" th:href="@{/css/bootstrap.min.css}" />

</head>
```

3rd Party CSS Libraries - Bootstrap

- Remote Files

```
<head>
  … …

  <!-- reference CSS file -->
  <link rel="stylesheet"
        href="https://cdn.jsdelivr.net/npm/bootstrap@5.2.3/dist/css/bootstrap.min.css" />

  … …
</head>
```

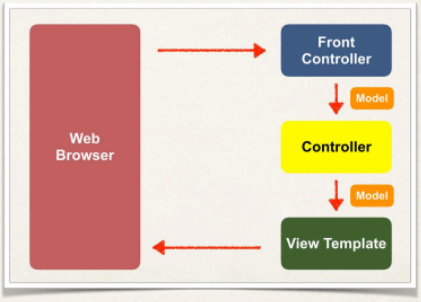
Spring MVC Behind the Scenes

Components of a Spring MVC Application

- A set of web pages to layout UI components
- A collection of Spring beans (controllers, services, etc…)
- Spring configuration (XML, Annotations or Java)

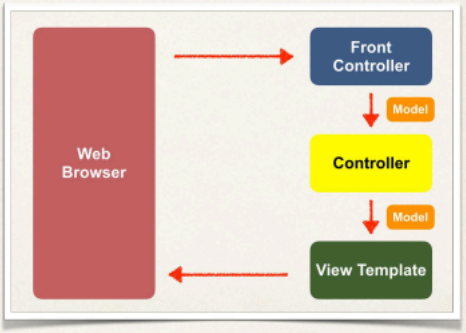
Spring MVC Front Controller

- Front controller known as **DispatcherServlet**
  - Part of the Spring Framework
  - Already developed by Spring Dev Team
- You will create
  - Model objects (orange)
  - View templates (dark green)
  - Controller classes (yellow)



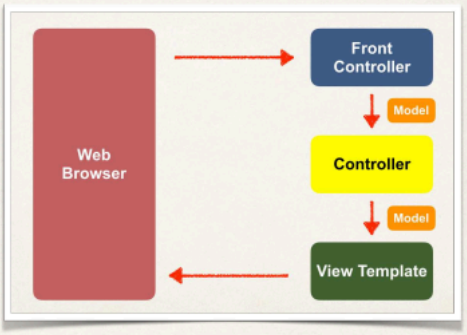
Controller

- Code created by developer
- Contains your business logic
  - Handle the request
  - Store/retrieve data (db, web service…)
  - Place data in model
- Send to appropriate view template



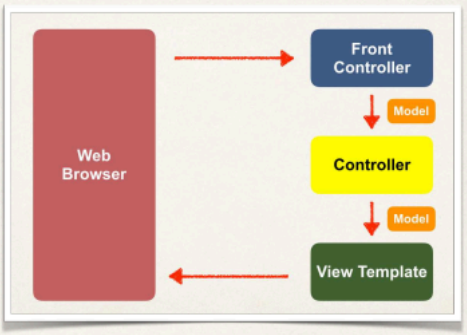
Model

- Model: contains your data
- Store/retrieve data via backend systems
  - database, web service, etc…
  - Use a Spring bean if you like
- Place your data in the model
  - Data can be any Java object/collection



View Template

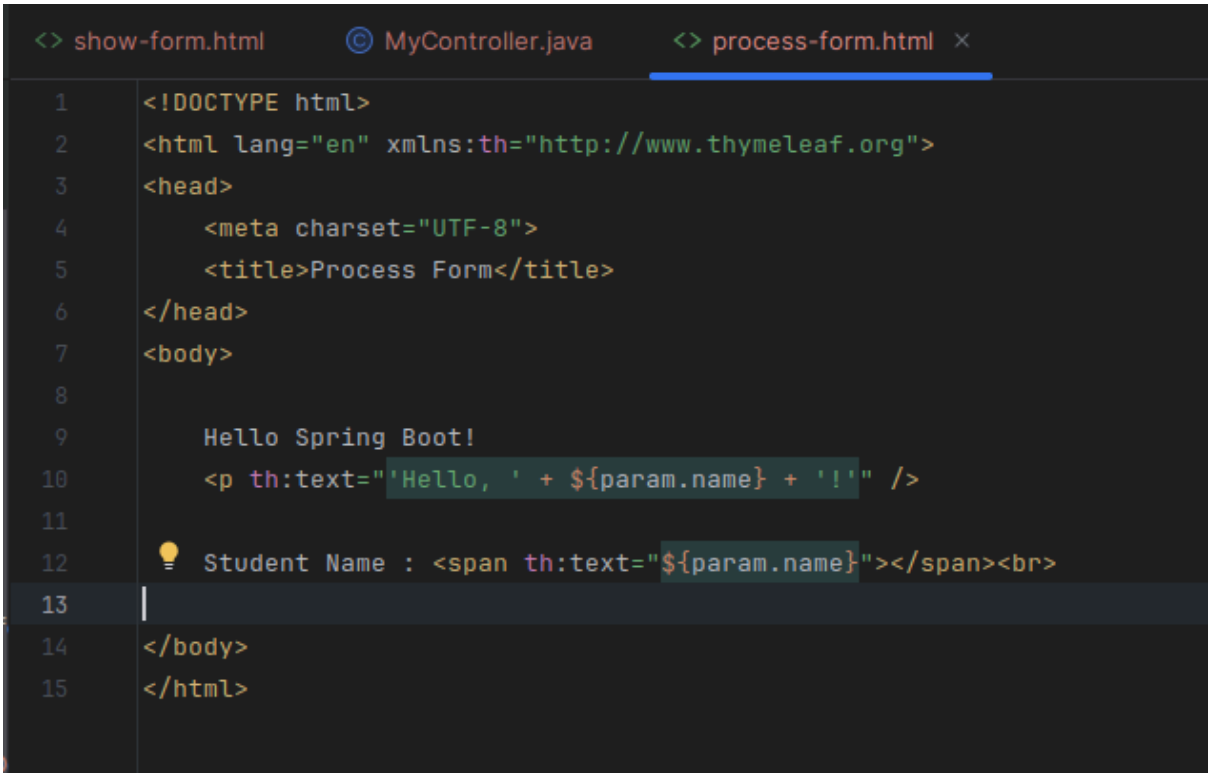
- Spring MVC is flexible
  - Supports many view templates
- Recommended: Thymeleaf
- Developer creates a page
  - Displays data



```java
@Controller
@RequestMapping("/api")
public class MyController {

    @GetMapping("/show")
    public String showForm() {
        return "show-form";
    }

    @GetMapping("/process")
    public String processForm() { return "process-form"; }

}
```

show-form.html × MyController.java process-form.html

```html
<!DOCTYPE html>
<html lang="en" xmlns:th="http://www.thymeleaf.org" >
<head>
    <meta charset="UTF-8">
    <title>Show Form</title>
</head>
<body>

    <form th:action="@{/api/process}">
        <label for="name"></label>
        <input type="text" id="name" name="name"
               placeholder="What is your name?" th:value="${name}" />
        <input type="submit">
    </form>

</body>
</html>

```



Reading Form Data with Spring MVC

Development Process

1. Create Controller class
2. Show HTML form
    a. Create controller method to show HTML Form
    b. Create View Page for HTML form
3. Process HTML Form
    a. Create controller method to process HTML Form
    b. Develop View Page for Confirmation

Adding Data to Spring Model

Spring Model

- The Model is a container for your application data
- In your Controller
- You can put anything in the model
- strings, objects, info from database, etc…
- Your View page can access data from the model

Passing Model to your Controller

```
@RequestMapping("/processFormVersionTwo")
public String letsShoutDude(HttpServletRequest request, Model model) {

   // read the request parameter from the HTML form
   String theName = request.getParameter("studentName");

   // convert the data to all caps
   theName = theName.toUpperCase();

   // create the message
   String result = "Yo! " + theName;

   // add message to the model
   model.addAttribute("message", result);

   return "helloworld";
}
```

```
<html><body>

Hello World of Spring!
...

The message: <span th:text="${message}" />

</body></html>
```

Reading HTML Form Data with @RequestParam Annotation

```
@RequestMapping("/processFormVersionTwo")
public String letsShoutDude(
        @RequestParam("studentName") String theName,
        Model model) {

   // now we can use the variable: theName

}
```

Behind the scenes:

Spring will read param from request: studentName

Bind it to the variable: theName

Constrain the Request Mapping - GET

```
@RequestMapping(path="/processForm", method=RequestMethod.GET)
public String processForm(...) {
  ...
}
```

- This mapping **ONLY** handles **GET** method
- Any other HTTP REQUEST method will get rejected

Show Form - Add Model Attribute

Code snippet from Controller

```
@GetMapping("/showStudentForm")
public String showForm(Model theModel) {

  theModel.addAttribute("student", new Student());

  return "student-form";
}
```

Setting up HTML Form - Data Binding

Name of model attribute

```
<form th:action="@{/processStudentForm}" th:object="${student}" method="POST">

  First name: <input type="text"  th:field="*{firstName}" />

  <br><br>

  Last name: <input type="text"  th:field="*{lastI

  <br><br>

  <input type="submit" value="Submit" />

</form>
```

```
@GetMapping("/showStudentForm")
public String showForm(Model theModel) {

  theModel.addAttribute("student", new Student());

  return "student-form";
}
```

Setting up HTML Form - Data Binding

```
<form th:action="@{/processStudentForm}" th:object="${student}" method="POST">

   First name: <input type="text"  th:field="*{firstName}" />

   <br><br>

   Last name: <input type="text"  th:field="*{lastName}" />

   <br><br>

   <input type="submit" value="Submit" />

</form>
```

*{ … } is shortcut syntax for: ${student.firstName}

*{ … } is shortcut syntax for: ${student.lastName}

When Form is Loaded … fields are populated

```
<form th:action="@{/processStudentForm}" th:object="${student}" method="POST">

   First name: <input type="text"  th:field="*{firstName}" />

   <br><br>

   Last name: <input type="text"  th:field="*{lastName}" />

   <br><br>

   <input type="submit" value="Submit" />

</form>
```

When form is **loaded**, Spring MVC will read student from the model, then call:

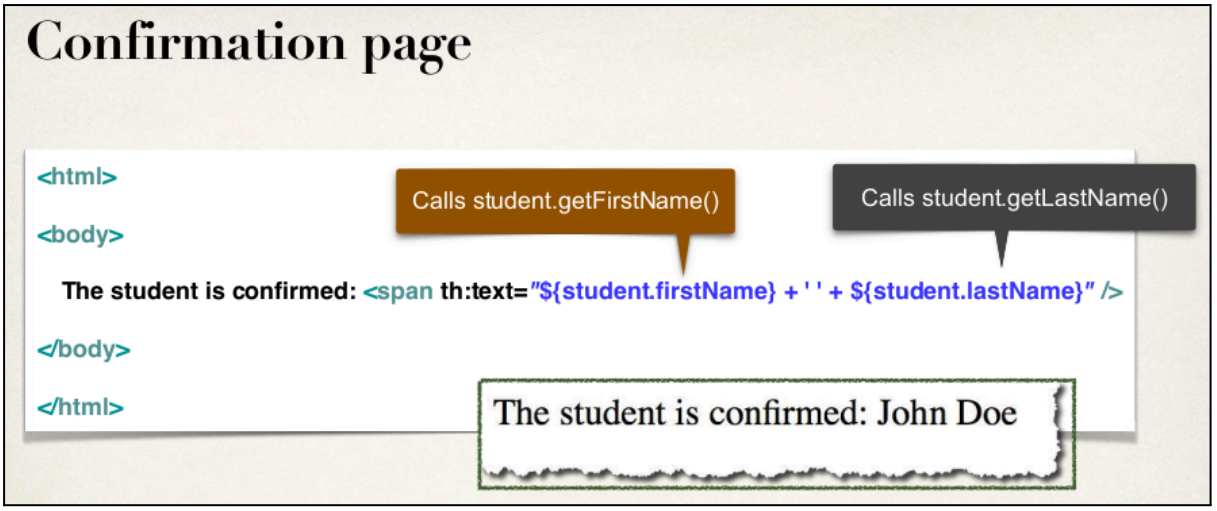
student.getFirstName()
…
student.getLastName()

When Form is submitted … calls setter methods

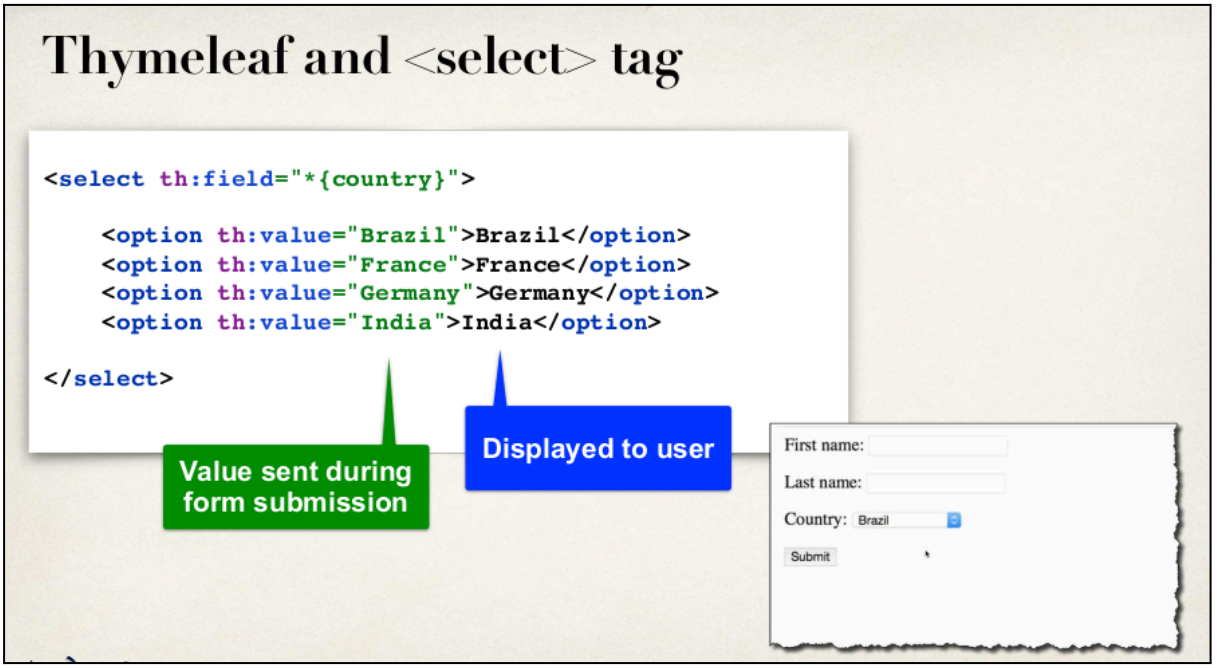
```
<form th:action="@{/processStudentForm}" th:object="${student}" method="POST">

   First name: <input type="text"  th:field="*{firstName}" />

   <br><br>

   Last name: <input type="text"  th:field="*{lastName}" />

   <br><br>

   <input type="submit" value="Submit" />

</form>
```

When form is **submitted**, Spring MVC will create a **new** Student instance and add to the model, then call:

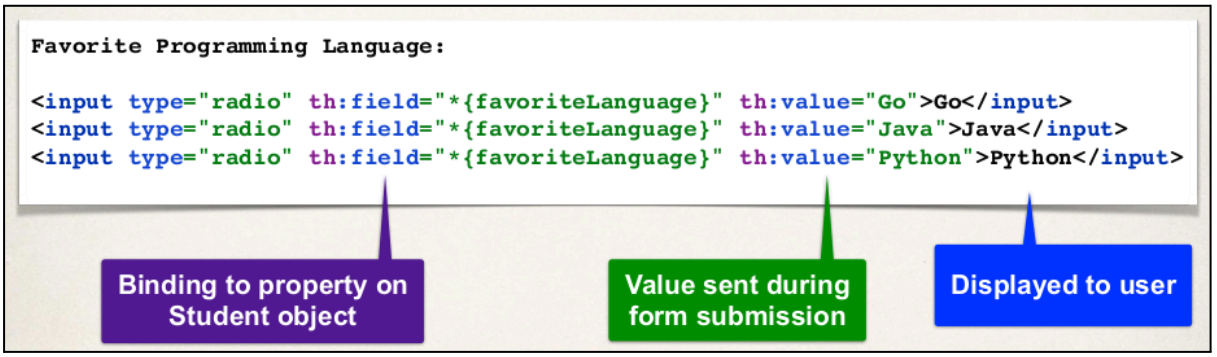
student.setFirstName(…)
…
student.setLastName(…)



Spring MVC Form - Drop Down List



Spring MVC Form - Radio Buttons



Spring MVC Forms - Checkbox

```
<input type="checkbox" th:field="*{favoriteSystems}" th:value="Linux">Linux</input>
<input type="checkbox" th:field="*{favoriteSystems}" th:value="macOS">macOS</input>
<input type="checkbox" th:field="*{favoriteSystems}"
                       th:value="'Microsoft Windows'">Microsoft Windows</input>
```

Spring MVC Form Validation

Java's Standard Bean Validation API

- Java has a standard Bean Validation API
- Defines a metadata model and API for entity validation
- Spring Boot and Thymeleaf also support the Bean Validation API

Bean Validation Features

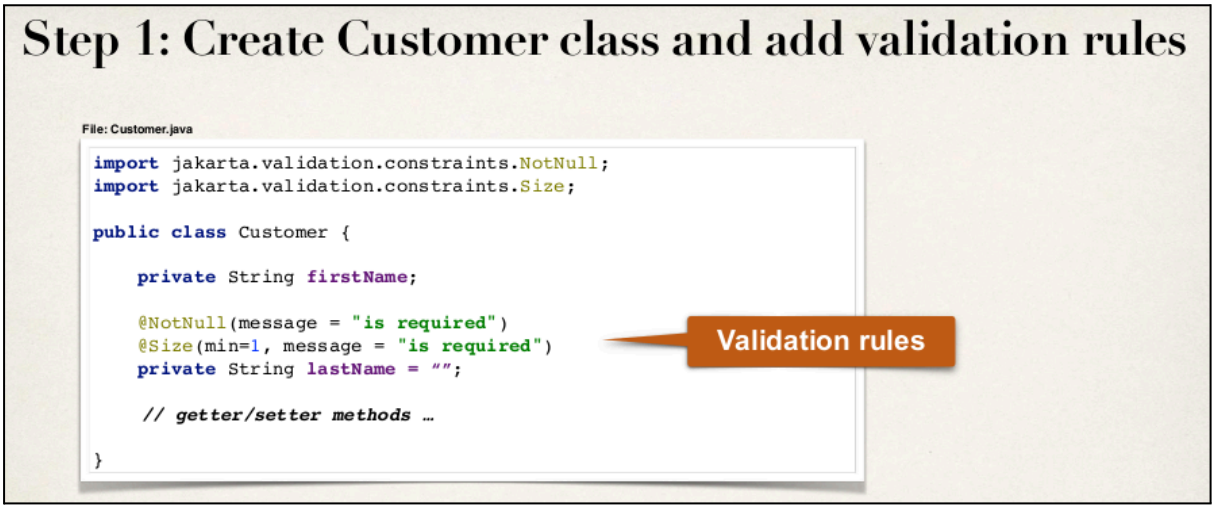
Validation Feature
required
validate length
validate numbers
validate with regular expressions
custom validation

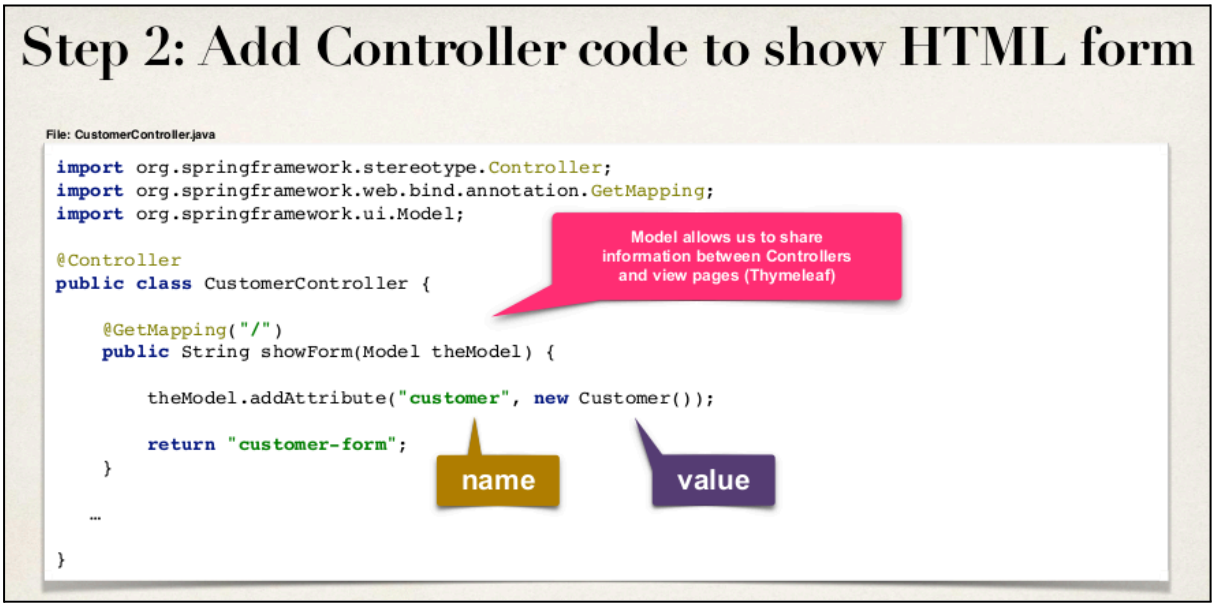
Validation Annotations

Annotation	Description
@NotNull	Checks that the annotated value is not null
@Min	Must be a number >= value
@Max	Must be a number <= value
@Size	Size must match the given size
@Pattern	Must match a regular expression pattern
@Future / @Past	Date must be in future or past of given date
others ...	

Spring MVC Form Validation Required Fields

Development Process
1. Create Customer class and add validation rules
2. Add Controller code to show HTML form
3. Develop HTML form and add validation support
4. Perform validation in the Controller class
5. Create confirmation page





Step 3: Develop HTML form and add validation support

File: customer-form.html

Where to submit form data

Model attribute name

```
<form th:action="@{/processForm}" th:object="${customer}" method="POST">

    First name: <input type="text" th:field="*{firstName}" />

    <br><br>

    Last name (*): <input type="text" th:field="*{lastName}" />

    <!-- Show error message (if present) -->
    <span th:if="${#fields.hasErrors('lastName')}"
          th:errors="*{lastName}"
          class="error"></span>

    <br><br>

    <input type="submit" value="Submit" />

</form>
```

Property name from Customer class

Property name from Customer class

Step 4: Perform validation in Controller class

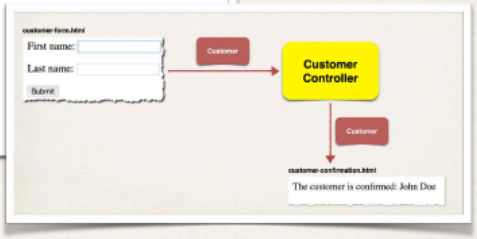
File: CustomerController.java

Model attribute name

Tell Spring MVC to perform validation

The results of validation

```
...

@PostMapping("/processForm")
public String processForm(
        @Valid @ModelAttribute("customer") Customer theCustomer,
        BindingResult theBindingResult) {

    if (theBindingResult.hasErrors()) {
        return "customer-form";
    }
    else {
        return "customer-confirmation";
    }
}
```



Step 5: Create confirmation page

File: customer-confirmation.html

```
<!DOCTYPE html>
<html xmlns:th="http://www.thymeleaf.org">

<body>

The customer is confirmed: <span th:text="${customer.firstName + ' ' + customer.lastName}" />

</body>

</html>
```

Spring MVC Validation @InitBinder

White Space

- Our previous example had a problem with white space
- Last name field with all whitespace passed … YIKES!
- Should have failed!
- We need to trim whitespace from input fields

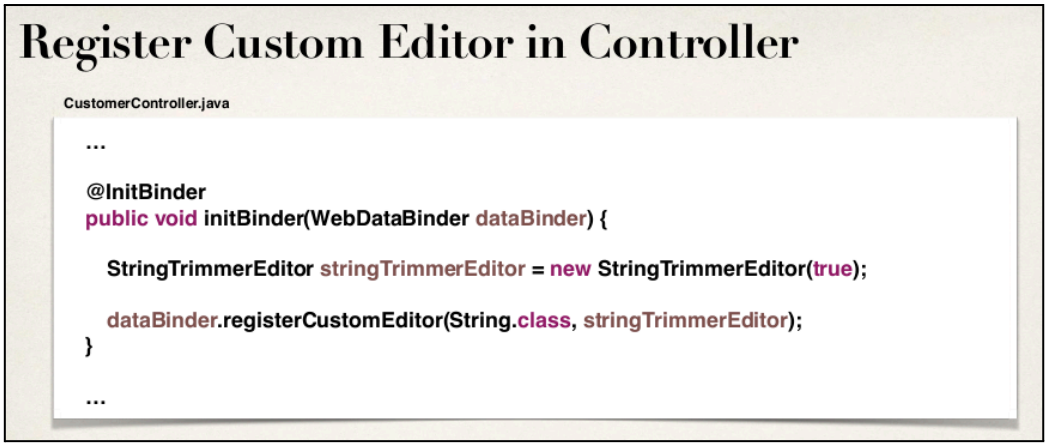
@InitBinder

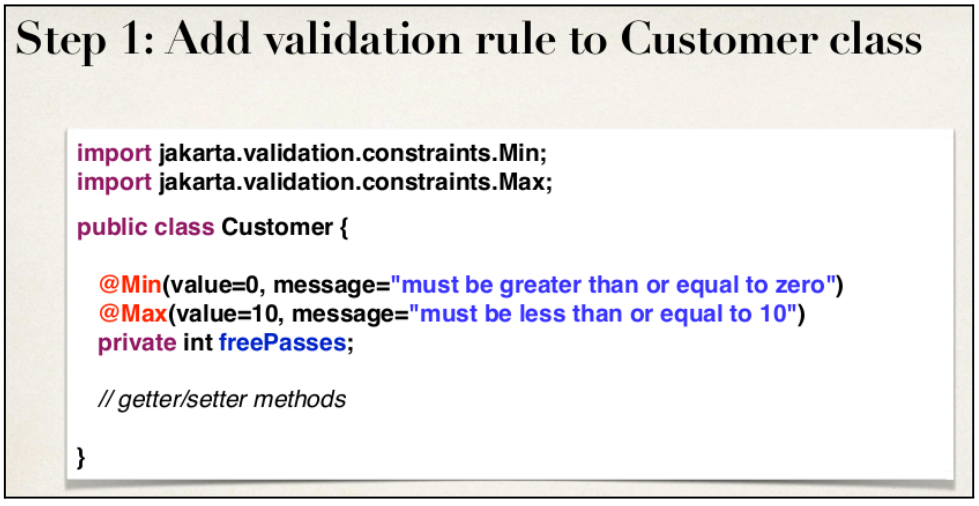
- @InitBinder annotation works as a pre-processor
- It will pre-process each web request to our controller
- Method annotated with @InitBinder is executed

@InitBinder

- We will use this to trim Strings
- Remove leading and trailing white space
- If String only has white spaces … trim it to null
- Will resolve our validation problem … whew :-)

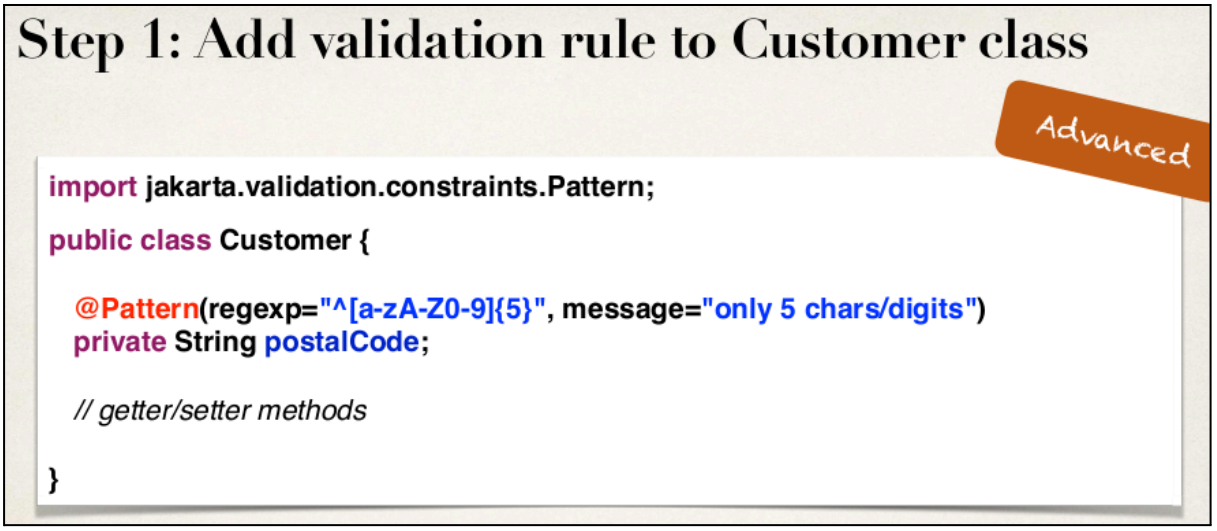
In Spring Boot, @InitBinder is an annotation used to customize the data binding process for web request parameters. This annotation is typically used within a controller to initialize a WebDataBinder instance. WebDataBinder is a data binder which can bind request parameters to JavaBean objects, and apply validation rules.



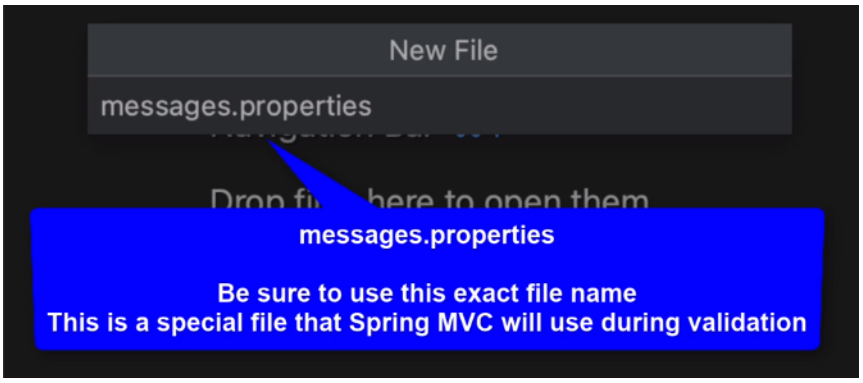


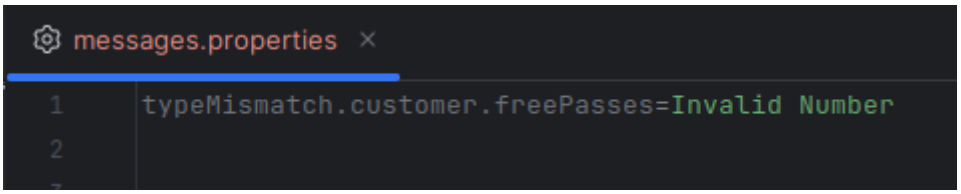
Spring MVC Validation Regular Expressions

- A sequence of characters that define a search pattern
    - This pattern is used to find or match strings
- Regular Expressions is like its own language (advanced topic)
    - I will assume you already know about regular expressions
- If not, then plenty of free tutorials available



Spring MVC Validation Handle String Input for Integer Fields





Spring MVC Validation Custom Validation

Custom Validation

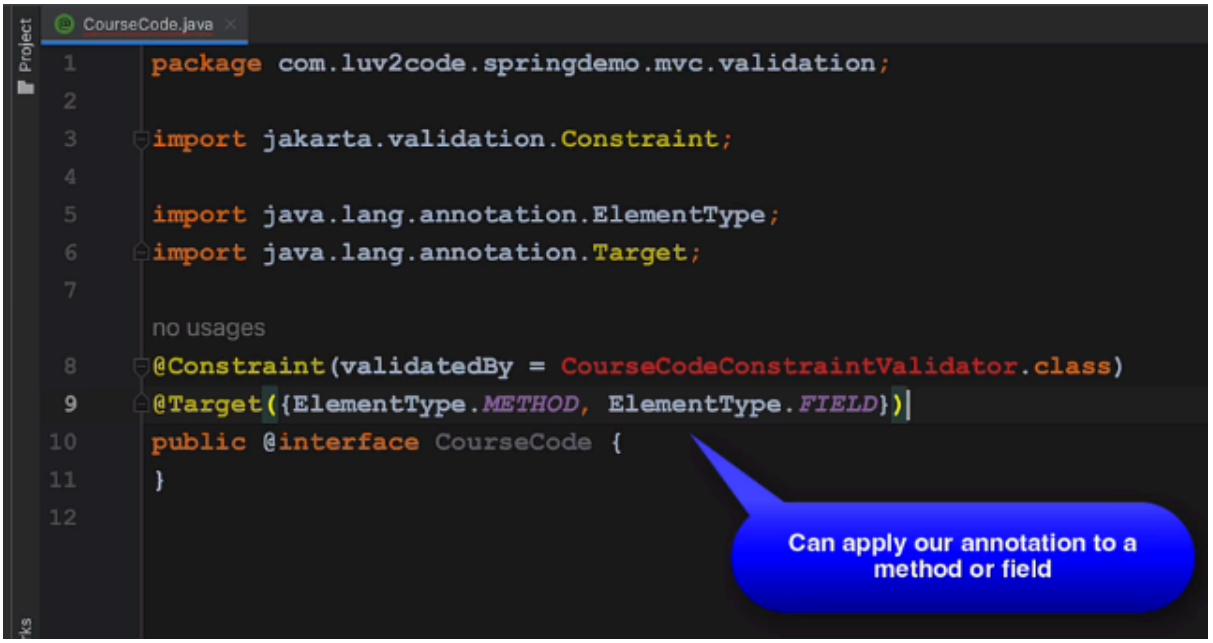
✤ Perform custom validation based on your business rules
✤ Our example: Course Code must start with “LUV”
✤ Spring MVC calls our custom validation
✤ Custom validation returns boolean value for pass/fail (true/false)

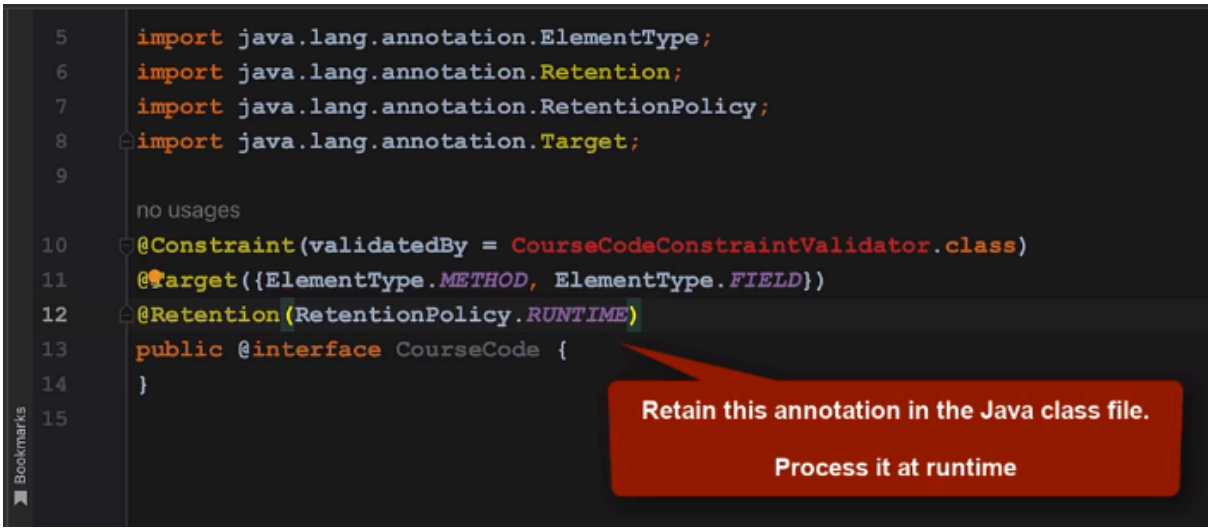
Development Process

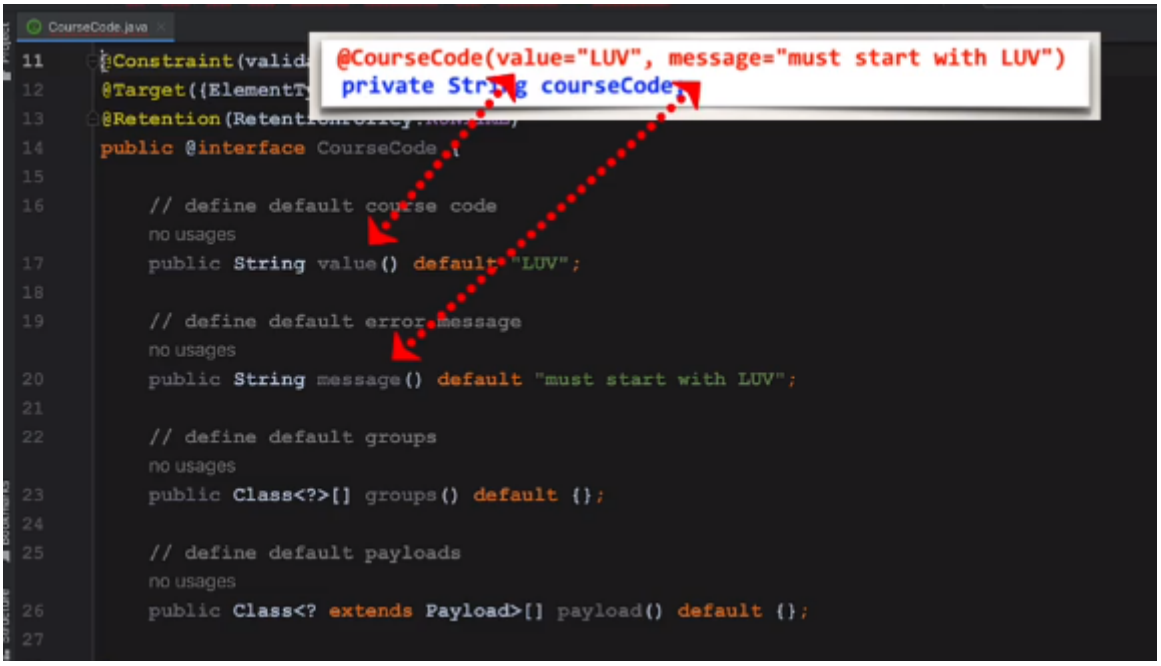
1. Create custom validation rule
2. Add validation rule to Customer class
3. Display error messages on HTML form
4. Update confirmation page

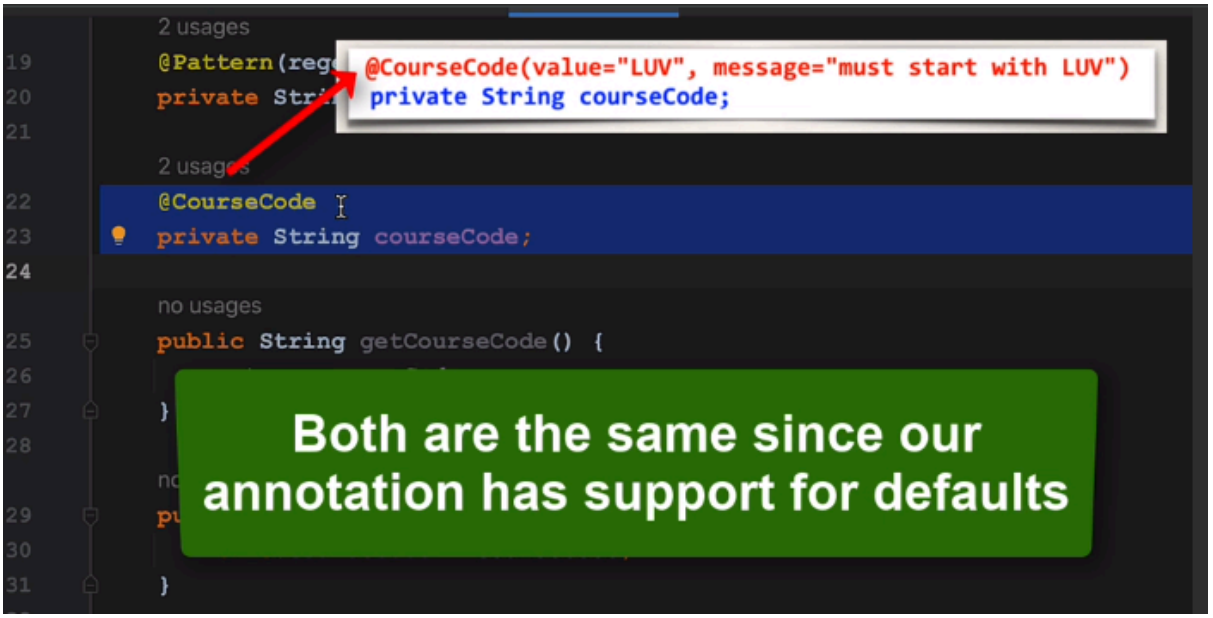
Development Process - Drill Down

1. Create custom validation rule
    a. Create @CourseCode annotation
    b. Create CourseCodeConstraintValidator




import java.lang.annotation.ElementType;
import java.lang.annotation.Retention;
import java.lang.annotation.RetentionPolicy;
import java.lang.annotation.Target;
no usages
@Constraint(validatedBy = CourseCodeConstraintValidator.class)
@Target({ElementType.METHOD, ElementType.FIELD})
@Retention(RetentionPolicy.RUNTIME)
public @interface CourseCode {
}
Retain this annotation in the Java class file.
Process it at runtime


CourseCode.java
@CourseCode(value="LUV", message="must start with LUV")
private String courseCode;
// define default course code
no usages
public String value() default "LUV";
// define default error message
no usages
public String message() default "must start with LUV";
// define default groups
no usages
public Class<?>[] groups() default {};
// define default payloads
no usages
public Class<? extends Payload>[] payload() default {};


2 usages
@CourseCode(value="LUV", message="must start with LUV")
private String courseCode;
2 usages
@CourseCode
private String courseCode;
no usages
public String getCourseCode() {
Both are the same since our annotation has support for defaults

Step 1a: Create @CourseCode annotation

Advanced

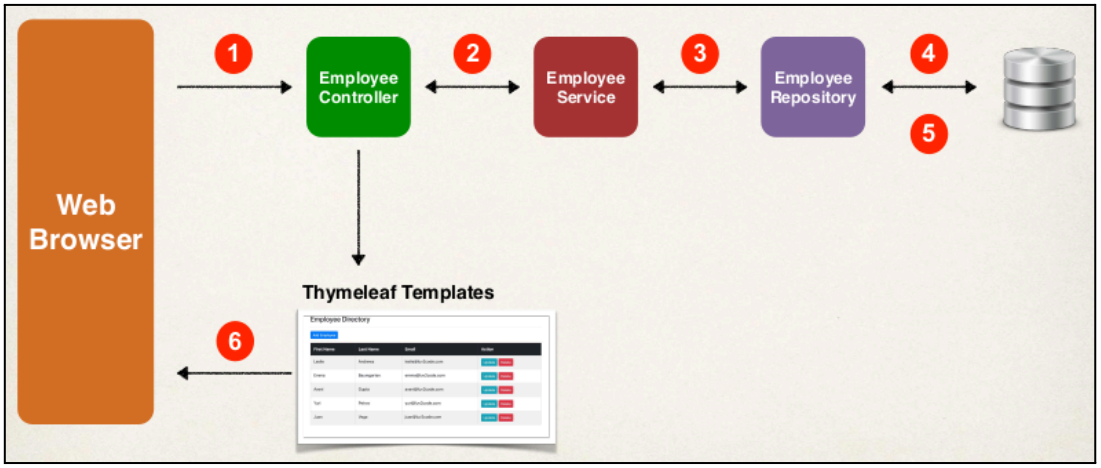
```
@Constraint(validatedBy = CourseCodeConstraintValidator.class)
@Target( { ElementType.METHOD, ElementType.FIELD } )
@Retention(RetentionPolicy.RUNTIME)
public @interface CourseCode {

   // define default course code
   public String value() default "LUV";

   // define default error message
   public String message() default "must start with LUV";

   ...
}
```

```
@CourseCode(value="LUV", message="must start with LUV")
private String courseCode;
```

Step 1b: Create CourseCodeConstraintValidator

```
import jakarta.validation.ConstraintValidator;
import jakarta.validation.ConstraintValidatorContext;

public class CourseCodeConstraintValidator
    implements ConstraintValidator<CourseCode, String> {

    private String coursePrefix;

    @Override
    public void initialize(CourseCode theCourseCode) {
        coursePrefix = theCourseCode.value();
    }

    @Override
    public boolean isValid(String theCode,
                           ConstraintValidatorContext theConstraintValidatorContext) {

        boolean result;

        if (theCode != null) {
            result = theCode.startsWith(coursePrefix);
        }
        else {
            result = true;
        }

        return result;
    }
}
```

Thymeleaf CRUD - Real Time Project



Project SetUp

- We will extend our existing Employee project and add DB integration
- Add EmployeeService, EmployeeRepository and Employee entity
- Available in one of our previous projects
- We created all of this code already from scratch … so we'll just copy/paste it
- Allows us to focus on creating EmployeeController and Thymeleaf
- templates

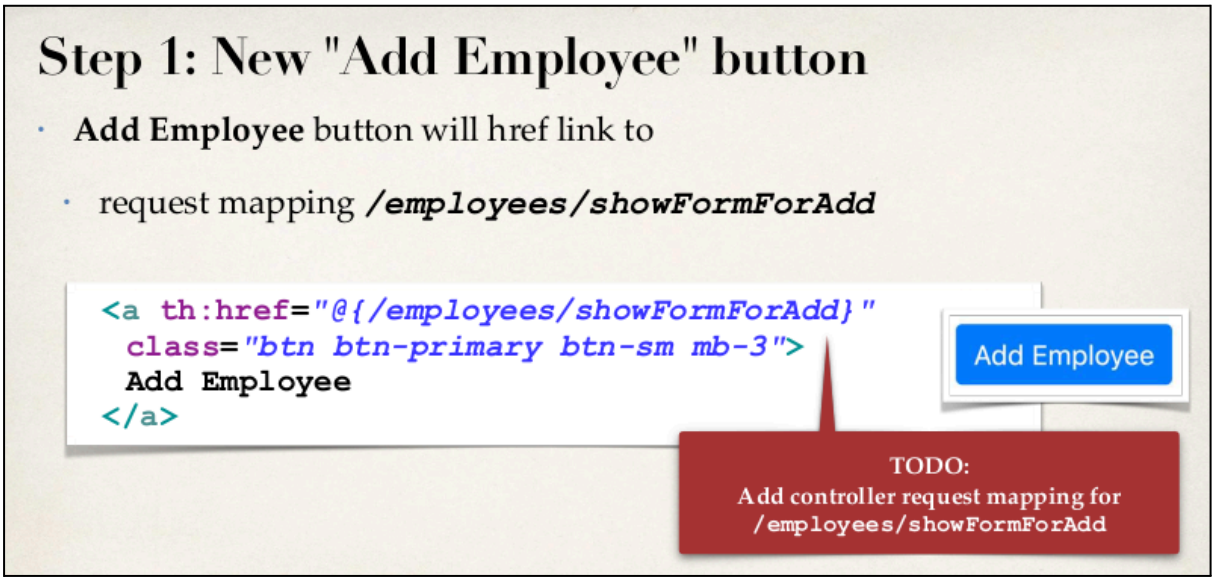
Thymeleaf - List Employee

list-employees.html

```
<!DOCTYPE html>
<html lang="en" xmlns:th="http://www.thymeleaf.org">
<head>
    <meta charset="UTF-8">
    <title>Employee Directory</title>
    <meta name="viewport" content="width=device-width, initial-scale=1">
    <title>Bootstrap demo</title>
    <link href="https://cdn.jsdelivr.net/npm/bootstrap@5.3.3/dist/css/bootstrap.min.css" rel="
</head>
<body>
<div class="container">
    <h3>Employee Directory</h3><hr>
    <table class="table table-bordered table-striped">
        <thead class="table-dark">
        <tr>
            <th>First Name</th>
            <th>Last Name</th>
            <th>Email</th>
        </tr>
        </thead>
        <tbody>
        <tr th:each="tempEmployee : ${employees}">
            <td th:text="${tempEmployee.firstName}" />
            <td th:text="${tempEmployee.lastName}" />
            <td th:text="${tempEmployee.email}" />
        </tr>
        </tbody>
```

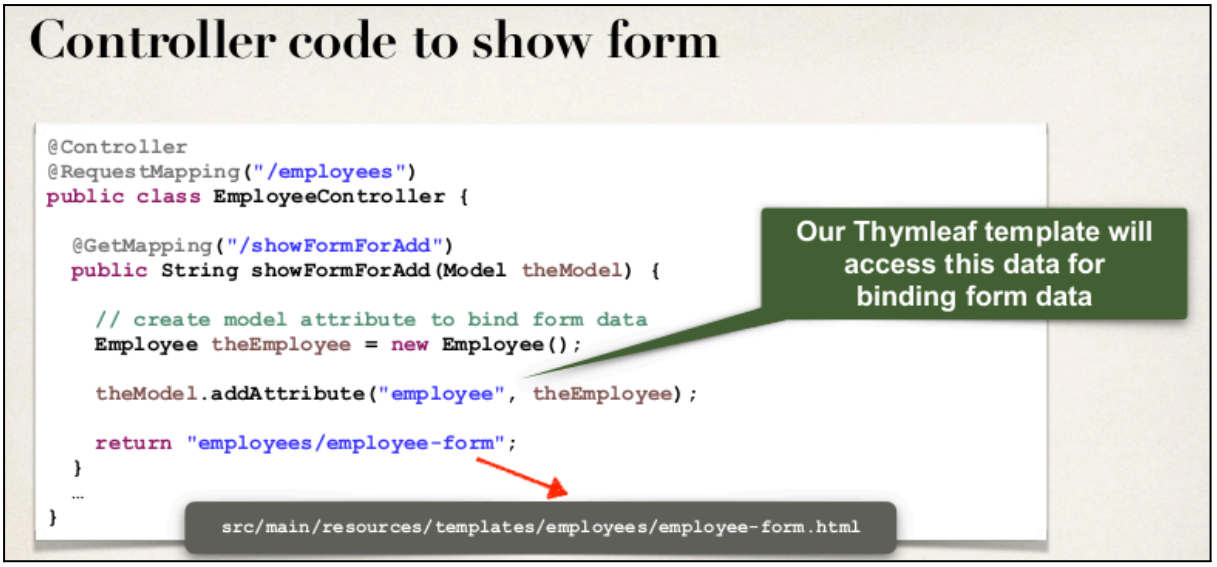
```java
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.RequestMapping;

import java.util.List;

Ruvini-Rangathara *
@Controller
@RequestMapping("/employees")
public class EmployeeController {
    2 usages
    private final EmployeeService employeeService;
    Ruvini-Rangathara
    public EmployeeController(EmployeeService theEmployeeService) { employeeService = theEmploye
    // add mapping for GET /employees
    Ruvini-Rangathara
    @GetMapping("/list")
    public String listEmployees(Model theModel) {
        List<Employee> theEmployees = employeeService.findAll();
        theModel.addAttribute( attributeName: "employees", theEmployees);
        return "employee/list-employees";
    }
}
```

Thymeleaf - Add Employee



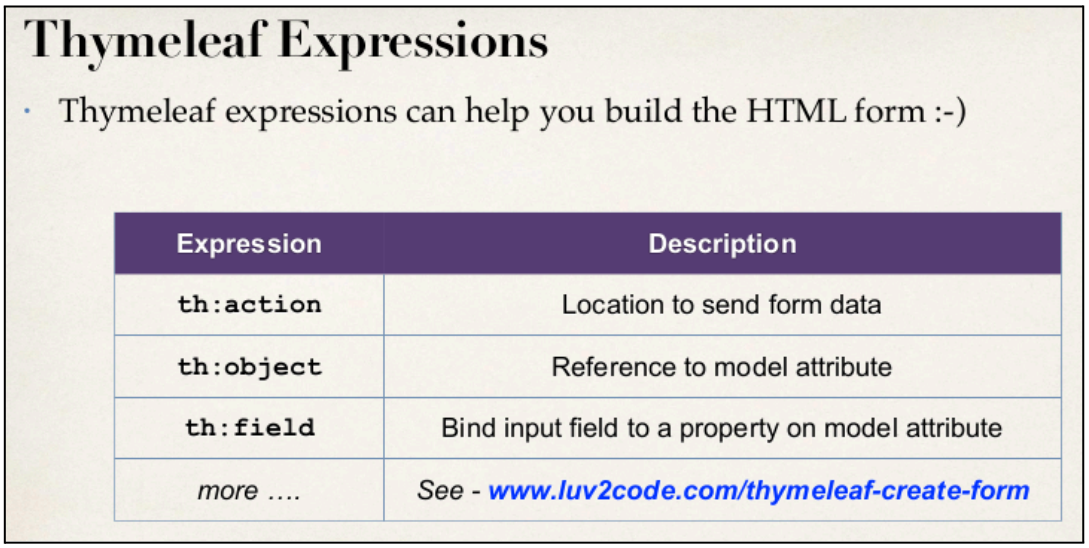
Showing Form
In your Spring Controller

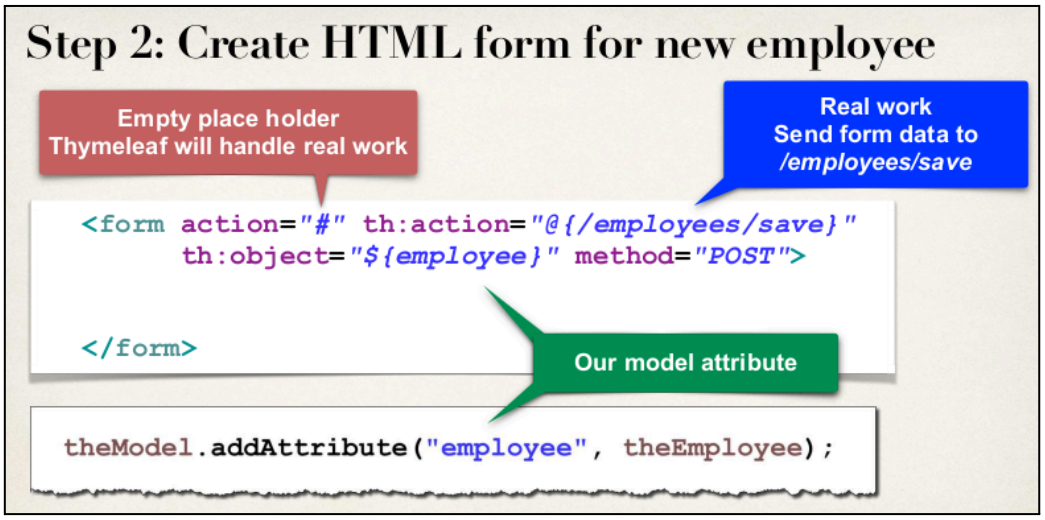
- Before you show the form, you must add a model attribute
- This is an object that will hold form data for the data binding



Thymeleaf and Spring MVC Data Binding

- Thymeleaf has special expressions for binding Spring MVC form data
- Automatically setting / retrieving data from a Java object
-





Step 2: Create HTML form for new employee

```
<form action="#" th:action="@{/employees/save}"
          th:object="${employee}" method="POST">

  <input type="text" th:field="*{firstName}" placeholder="First name">

  <input type="text" th:field="*{lastName}" placeholder="Last name">

  <input type="text" th:field="*{email}" placeholder="Email">

  <button type="submit">Save</button>

</form>
```

1 When form is **loaded**, will call:

employee.getFirstName()
...
employee.getLastName

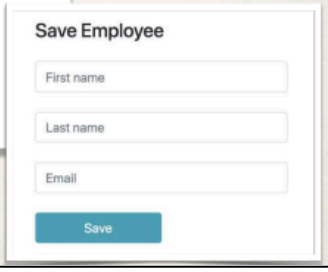
2 When form is **submitted**, will call:

employee.setFirstName(...)
...
employee.setLastName(...)

Step 2: Create HTML form for new employee

TODO:
Add controller request mapping for
/employees/save

```
<form action="#" th:action="@{/employees/save}"
          th:object="${employee}" method="POST">

  <input type="text" th:field="*{firstName}" placeholder="First name"
         class="form-control mb-4 w-25">

  <input type="text" th:field="*{lastName}" placeholder="Last name"
         class="form-control mb-4 w-25">

  <input type="text" th:field="*{email}" placeholder="Email"
         class="form-control mb-4 w-25">

  <button type="submit" class="btn btn-info col-2">Save</button>

 </form>
```



Step 3: Process form data to save employee

```
@Controller
@RequestMapping("/employees")
public class EmployeeController {

   private EmployeeService employeeService;

   public EmployeeController(EmployeeService theEmployeeService) {
      employeeService = theEmployeeService;
   }

   @PostMapping("/save")
   public String saveEmployee(@ModelAttribute("employee") Employee theEmployee) {

      // save the employee
      employeeService.save(theEmployee);

      // use a redirect to prevent duplicate submissions
      return "redirect:/employees/list";
   }
   ...
}
```

"Post/Redirect/Get" pattern

For more info see
www.luv2code.com/post-redirect-get

Redirect to request mapping
/employees/list

Thymeleaf - Update Employee

Step 1: "Update" button

- **Update** button includes employee id


```
<tr th:each="tempEmployee : ${employees}">
  ...
  <td>

    <a th:href="@{/employees/showFormForUpdate(employeeId=${tempEmployee.id})}"
       class="btn btn-info btn-sm">
      Update
    </a>

  </td>

</tr>
```

Appends to URL

?employeeId=xxx

Step 2: Pre-populate Form

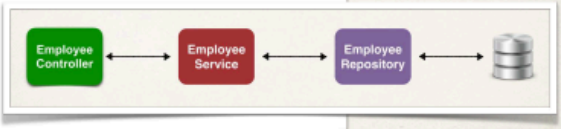
```
<form action="#" th:action="@{/employees/save}"
          th:object="${employee}" method="POST">

  <!-- Add hidden form field to handle update -->
  <input type="hidden" th:field="*{id}" />

  <input type="text" th:field="*{firstName}"
      class="form-control mb-4 w-25" placeholder="First name">

  <input type="text" th:field="*{lastName}"
      class="form-control mb-4 w-25" placeholder="Last name">

  <input type="text" th:field="*{email}"
      class="form-control mb-4 w-25" placeholder="Email">

  <button type="submit" class="btn btn-info col-2">Save</button>

</form>
```

Hidden form field required for updates

Step 3: Process form data to save employee

- No need for new code … we can reuse our existing code
- Works the same for add or update :-)

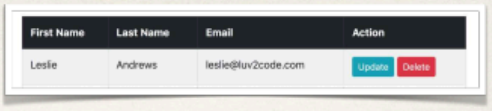
```
@Controller
@RequestMapping("/employees")
public class EmployeeController {
  ...

  @PostMapping("/save")
  public String saveEmployee(@ModelAttribute("employee") Employee theEmployee) {

    // save the employee
    employeeService.save(theEmployee);

    // use a redirect to prevent duplicate submissions
    return "redirect:/employees/list";
  }
  ...
}
```



Thymeleaf - Delete Employee

Step 1: "Delete" button

- **Delete** button includes employee id



Appends to URL

?employeeId=xxx

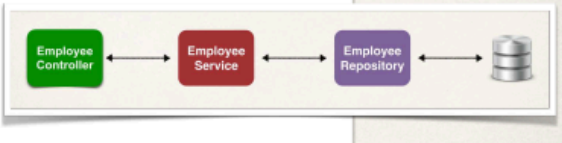
```
<tr th:each="tempEmployee : ${employees}">
  ...
  <td>

    <a th:href="@{/employees/delete(employeeId=${tempEmployee.id})}"
       class="btn btn-danger btn-sm"
       onclick="if (!(confirm('Are you sure you want to delete this employee?'))) return false">
      Delete
    </a>

  </td>

</tr>
```

JavaScript to prompt user before deleting

Step 2: Add controller code for delete

```
@Controller
@RequestMapping("/employees")
public class EmployeeController {
  ...

  @GetMapping("/delete")
  public String delete(@RequestParam("employeeId") int theId) {

    // delete the employee
    employeeService.deleteById(theId);

    // redirect to /employees/list
    return "redirect:/employees/list";
  }
  ...
}
```

```
<a th:href="@{/employees/delete(employeeId=${tempEmployee.id})}"
```



Spring MVC Security Overview

You will learn how to …
Secure Spring MVC Web App

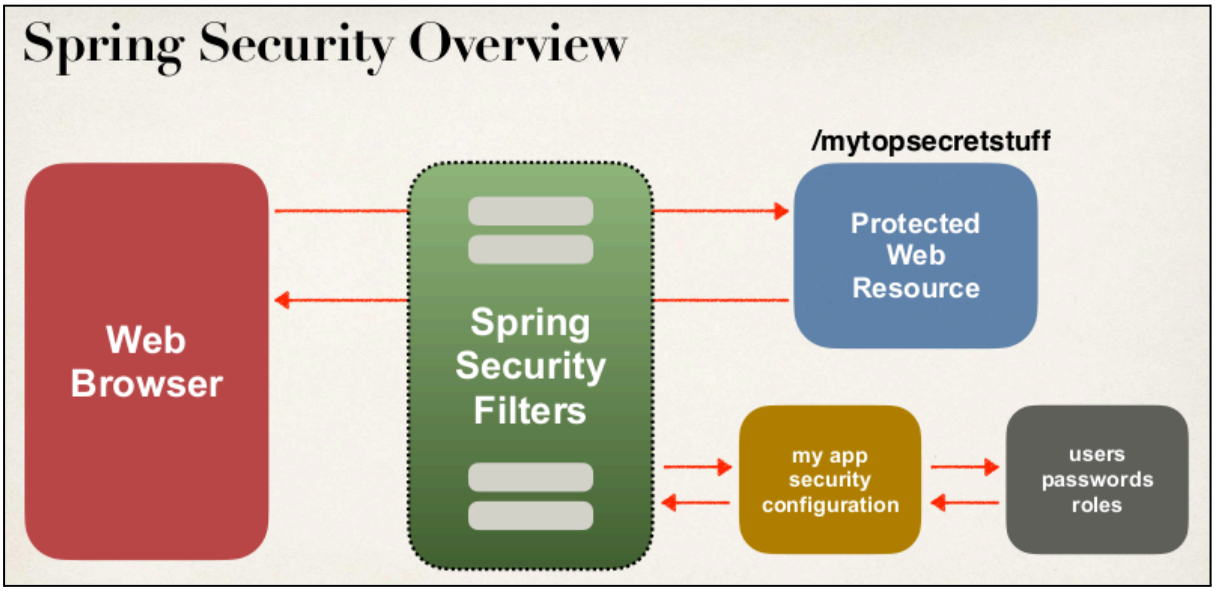
- Develop login pages (default and custom
- De ne users and roles with simple authentication
- Protect URLs based on rolE
- Hide/show content based on role
- Store users, passwords and roles in DB (plain-text -> encrypted)

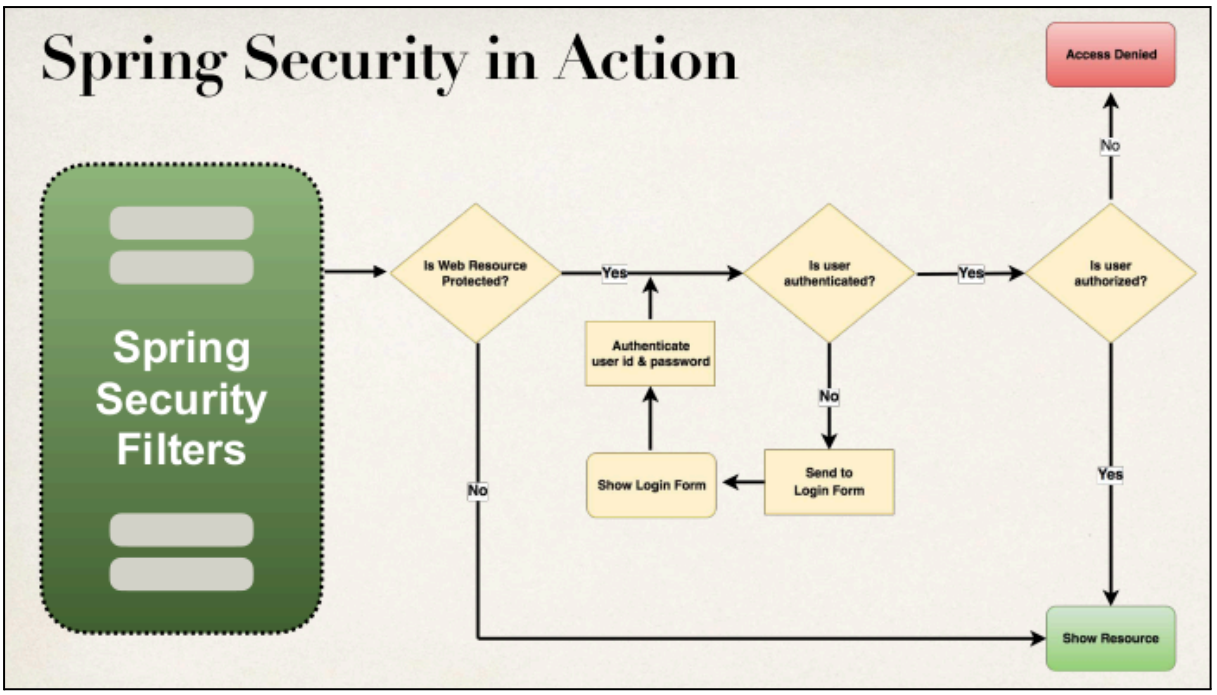
Spring Security Model
Spring Security de nes a framework for security

- Implemented using Servlet filters in the background
- Two methods of securing an app: declarative and programmatic

Spring Security with Servlet Filters
Servlet Filters are used to pre-process / post-process web requests

- Servlet Filters can route web requests based on security logic
- Spring provides a bulk of security functionality with servlet filters





Security Concepts

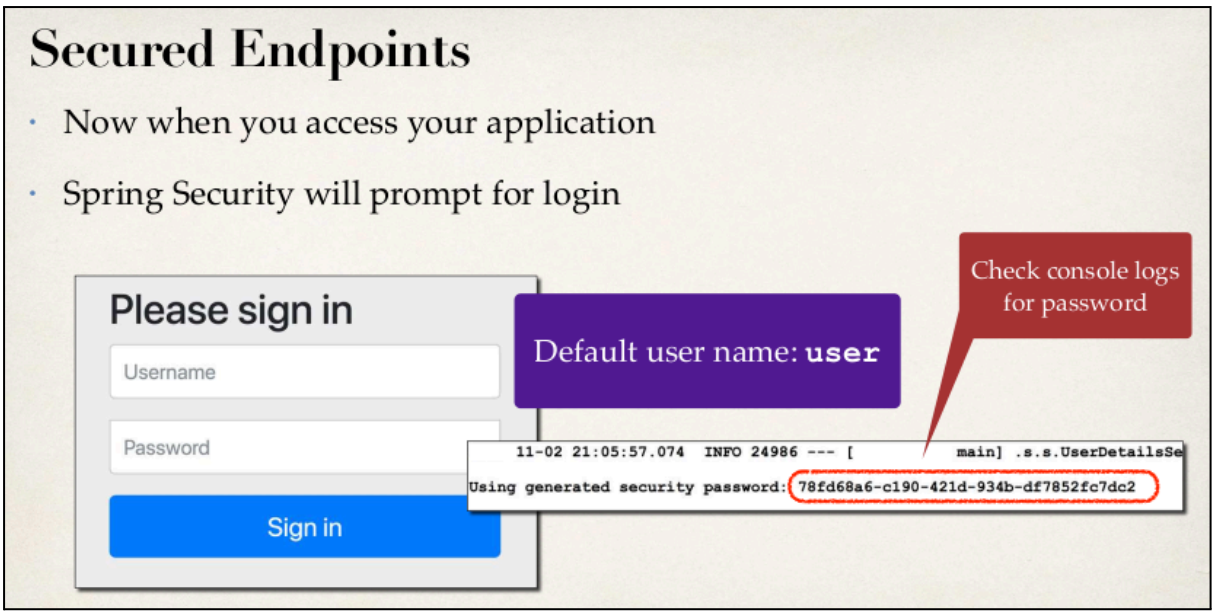
Authentication
Check user id and password with credentials stored in app / db
Authorization
Check to see if user has an authorized role

Declarative Security

- All Java config: @Configuration
- De ne application's security constraints in configuration
- Provides separation of concerns between application code and security

Programmatic Security

- Spring Security provides an API for custom application coding
- Provides greater customization for specific app requirements



Different Login Methods

- HTTP Basic Authentication
- Default login form
    - Spring Security provides a default login form
- Custom login form
    - your own look-and-feel, HTML + CSS

Spring MVC Security Project Setup

Development Process

1. Create project at Spring Initializr website
2. Add Maven dependencies for Spring MVC Web App, Security, Thymeleaf
3. Develop our Spring controller
4. Develop our Thymeleaf view page

Configuring Basic Security

User ID	Password	Roles
john	test123	EMPLOYEE
mary	test123	EMPLOYEE, MANAGER
susan	test123	EMPLOYEE, MANAGER, ADMIN

Development Process

1. Create Spring Security Configuration (@Configuration)
2. Add users, passwords and roles

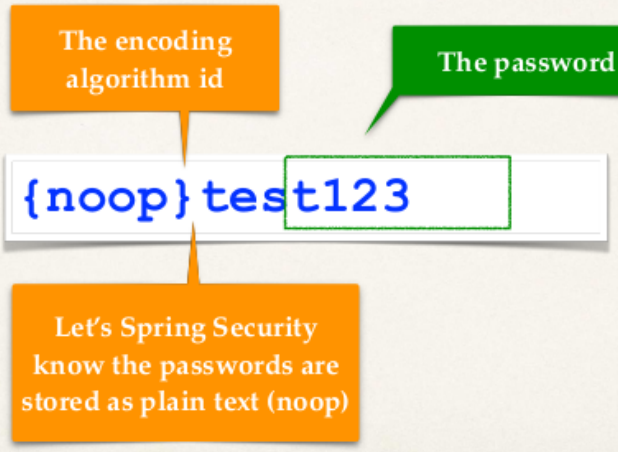
Spring Security Password Storage

- In Spring Security, passwords are stored using a specific format

```
{id}encodedPassword
```

ID	Description
noop	Plain text passwords
bcrypt	BCrypt password hashing
...	...

Password Example



Step 2: Add users, passwords and roles

File: DemoSecurityConfig.java

```
@Configuration
public class DemoSecurityConfig {

    @Bean
    public InMemoryUserDetailsManager userDetailsManager() {

        UserDetails john = User.builder()
                .username("john")
                .password("{noop}test123")
                .roles("EMPLOYEE")
                .build();

        UserDetails mary = User.builder()
                .username("mary")
                .password("{noop}test123")
                .roles("EMPLOYEE", "MANAGER")
                .build();

        UserDetails susan = User.builder()
                .username("susan")
                .password("{noop}test123")
                .roles("EMPLOYEE", "MANAGER", "ADMIN")
                .build();

        return new InMemoryUserDetailsManager(john, mary, susan);
    }
}
```

User ID	Password	Roles
john	test123	EMPLOYEE
mary	test123	EMPLOYEE, MANAGER
susan	test123	EMPLOYEE, MANAGER, ADMIN

We will add DB support in later videos

(plaintext and encrypted)

© luv2code LLC

Step 1: Modify Spring Security Configuration

File: DemoSecurityConfig.java

```
@Bean
public SecurityFilterChain filterChain(HttpSecurity http) throws Exception {

    http.authorizeHttpRequests(configurer ->
                    configurer
                            .anyRequest().authenticated()
            )
            .formLogin(form ->
                    form
                            .loginPage("/showMyLoginPage")
                            .loginProcessingUrl("/authenticateTheUser")
                            .permitAll()
            );

    return http.build();
}
```

Login form should POST data to this URL for processing

(check user id and password)

Step 2: Develop a Controller to show the custom login form

File: LoginController.java

```
@Controller
public class LoginController {

  @GetMapping("/showMyLoginPage")
  public String showMyLoginPage() {

    return "plain-login";

  }
}
```

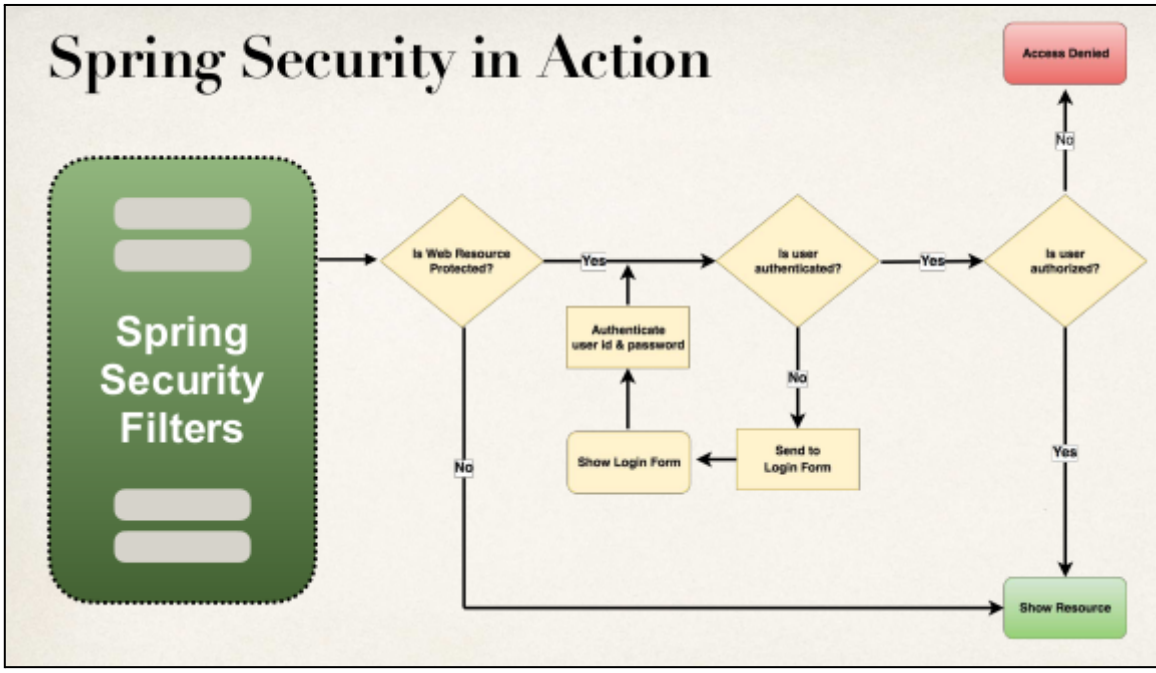
```
.loginPage("/showMyLoginPage")
.loginProcessingUrl("/authenticateTheUser")
.permitAll()
```

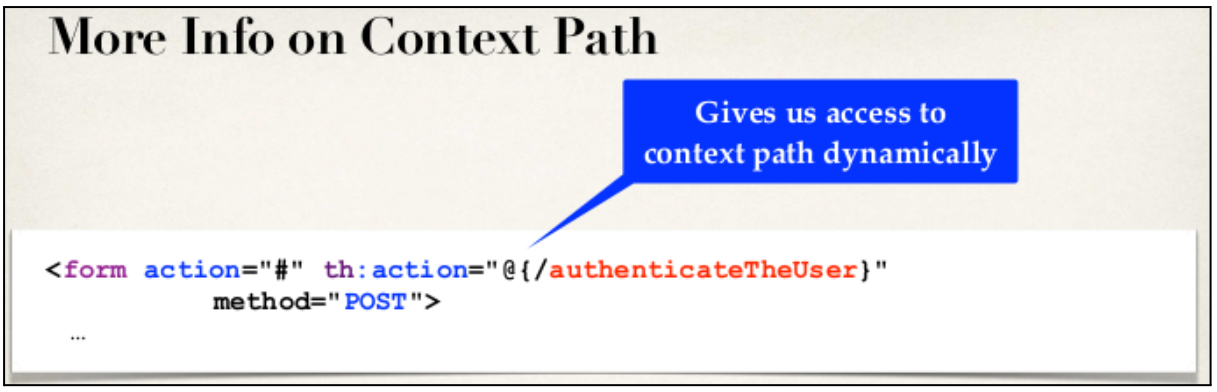
View name

TO DO
We need to create this file

src/main/resources/templates/plain-login.html

Spring Security in Action

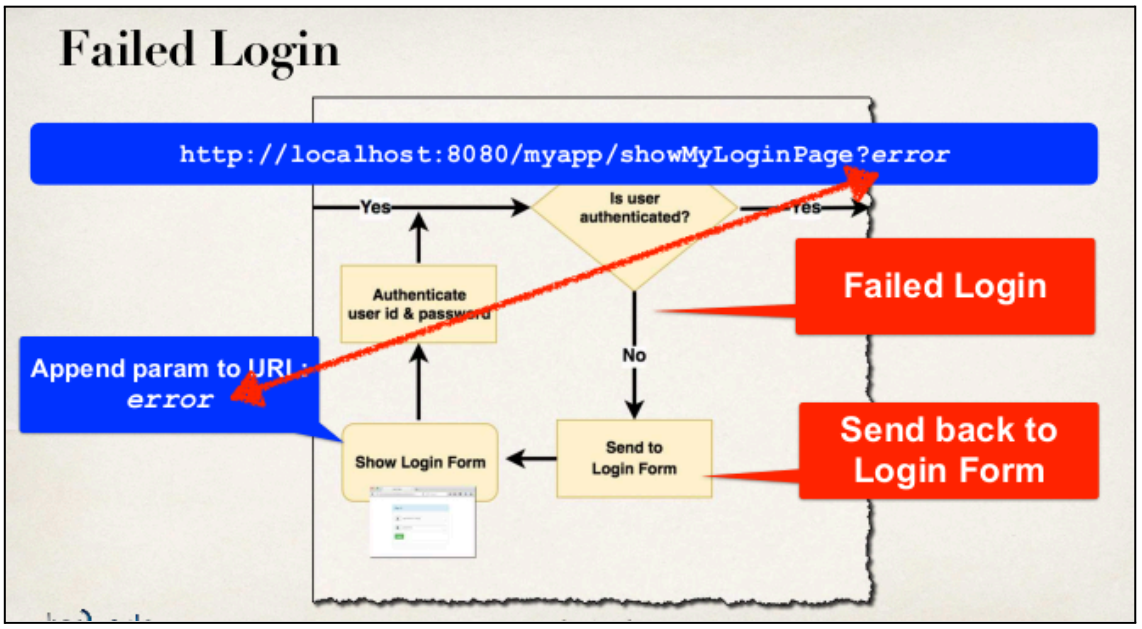




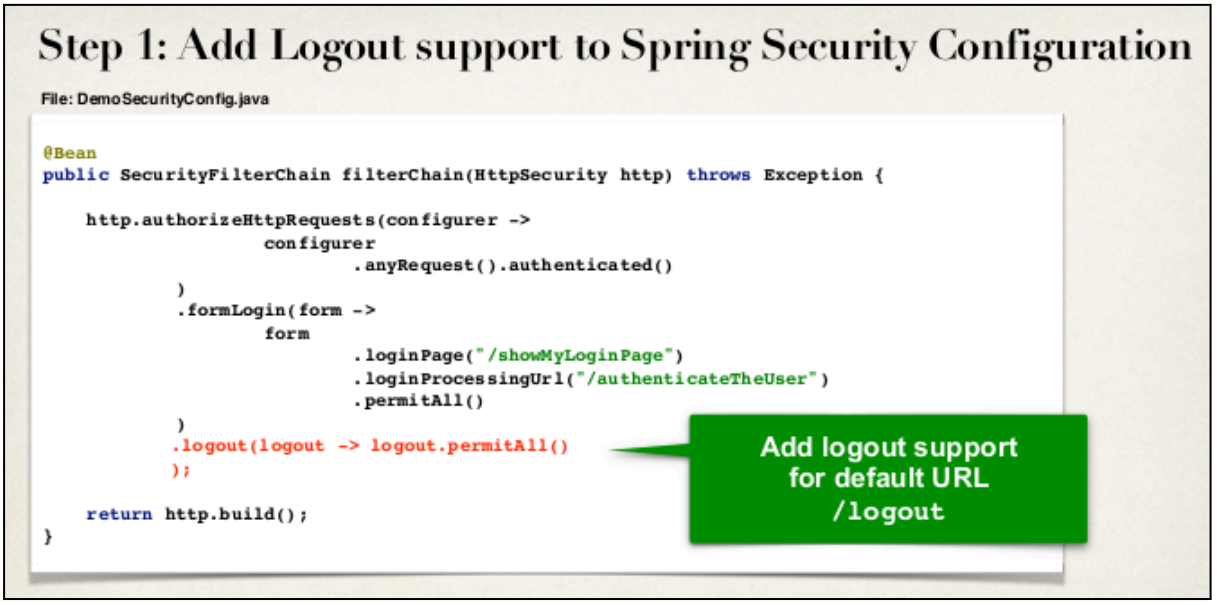
Why use Context Path?

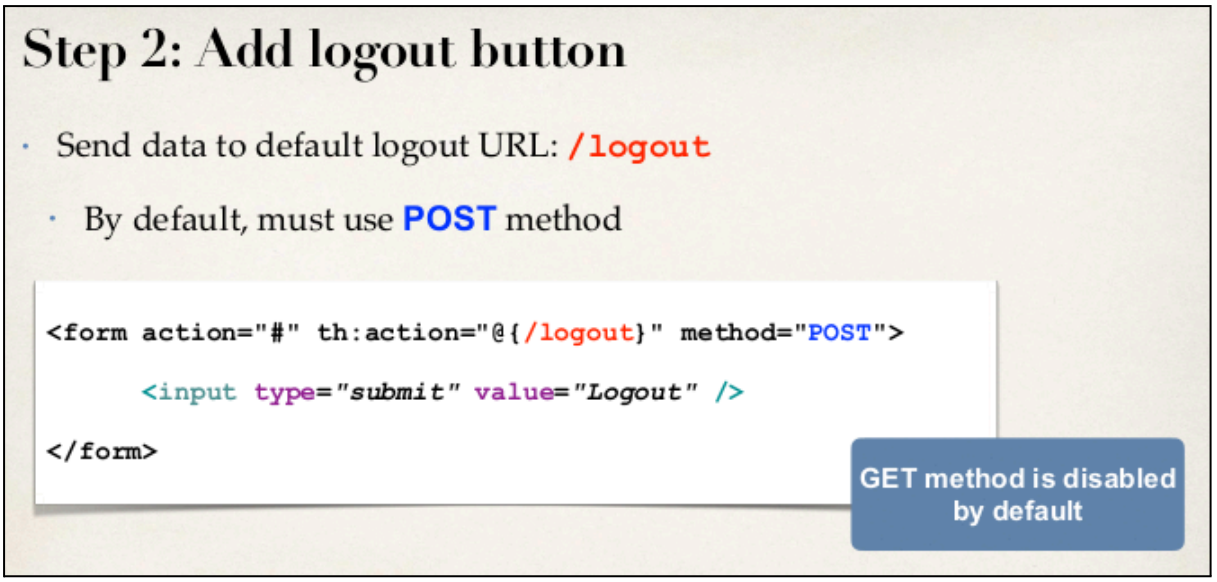
- Allows us to dynamically reference context path of application
- Helps to keep links relative to application context path
- If you change context path of app, then links will still work
- Much better than a hard-coding context path …

Spring Security - Show Login Error



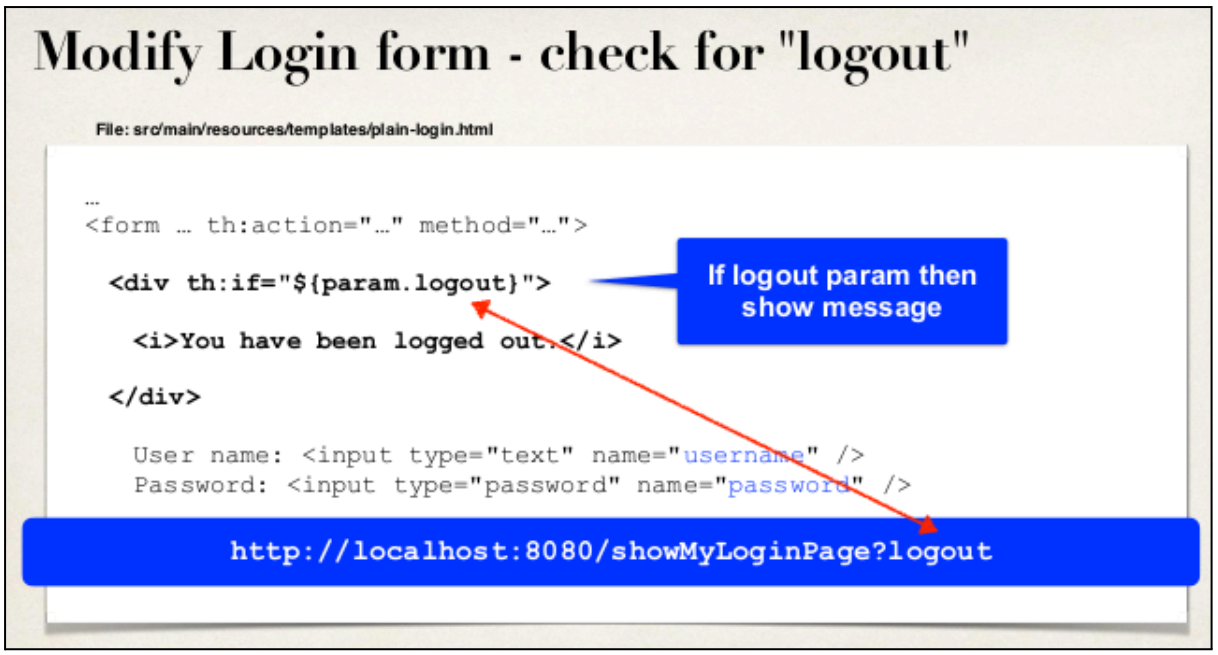
Spring Security - Logout





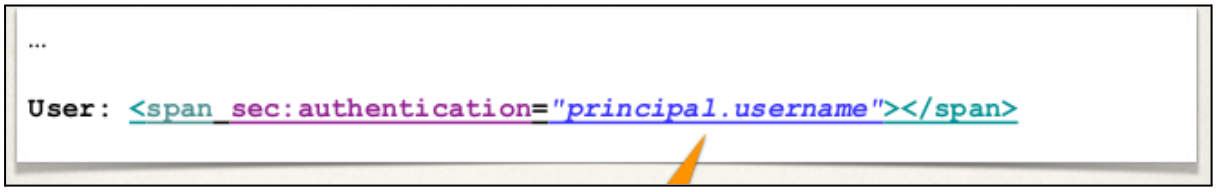
Logout process

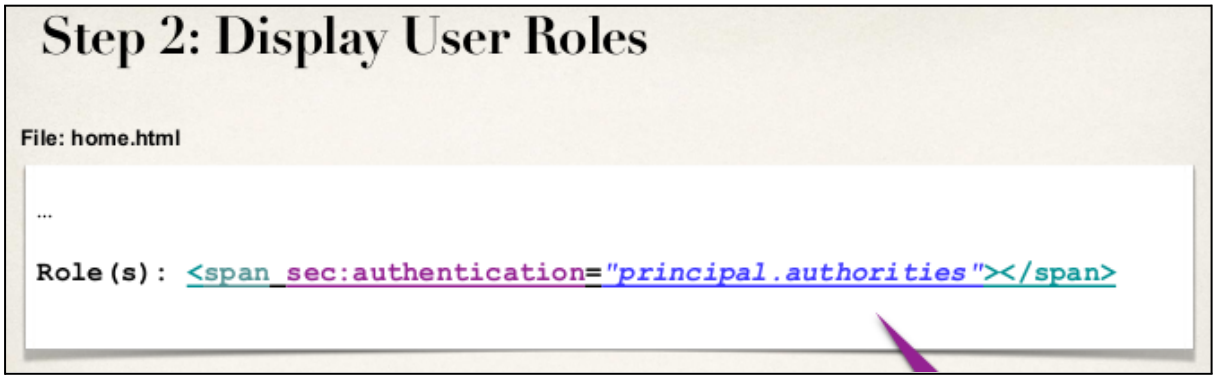
- When a logout is processed, by default Spring Security will …
- Invalidate user's HTTP session and remove session cookies, etc
- Send user back to your login page
- Append a logout parameter: ?logout



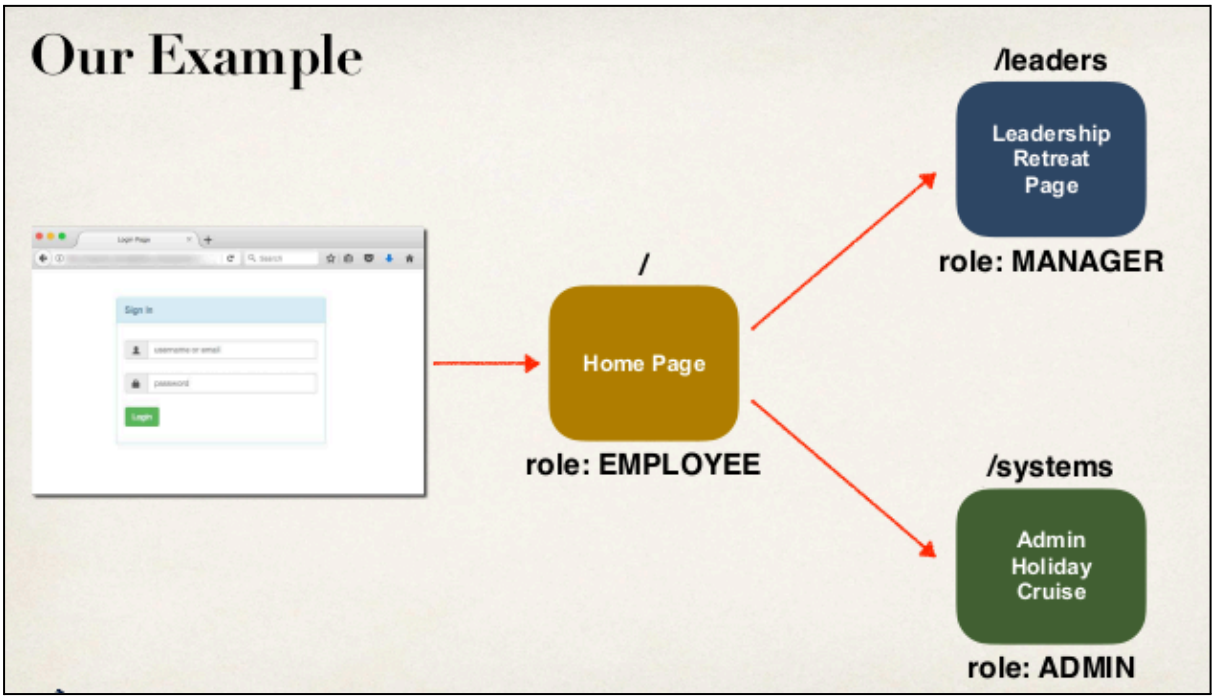
Display User ID and Roles

- Spring Security provides support for accessing user id and roles



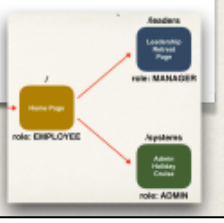


Restrict Access Based on Roles




```
@Bean
public SecurityFilterChain filterChain(HttpSecurity http) throws Exception {

    http.authorizeHttpRequests(configurer ->
                    configurer
                            .requestMatchers("/").hasRole("EMPLOYEE")
                            .requestMatchers("/leaders/**").hasRole("MANAGER")
                            .requestMatchers("/systems/**").hasRole("ADMIN")
                            .anyRequest().authenticated()
            )
    ...
}
```

User ID	Password	Roles
john	test123	EMPLOYEE
mary	test123	EMPLOYEE, MANAGER
susan	test123	EMPLOYEE, MANAGER, ADMIN



Custom Access Denied Page

Step 1: Configure custom page access denied

```
@Bean
public SecurityFilterChain filterChain(HttpSecurity http) throws Exception {

    http.authorizeHttpRequests(configurer ->
                    configurer
                            .requestMatchers("/").hasRole("EMPLOYEE")
                            …
            )
            .exceptionHandling(configurer ->
                    configurer
                            .accessDeniedPage("/access-denied")
            );
    …
}
```

Our request mapping path

Display Content Based on Roles

User: john

Role(s): [ROLE_EMPLOYEE]

Leadership Meeting (Only for Manager peeps)

IT Systems Meeting (Only for Admin peeps)

Logout

Since John is an employee,

he shouldn't be able to see this content / links

Spring Security

Only show this section for users with ADMIN role

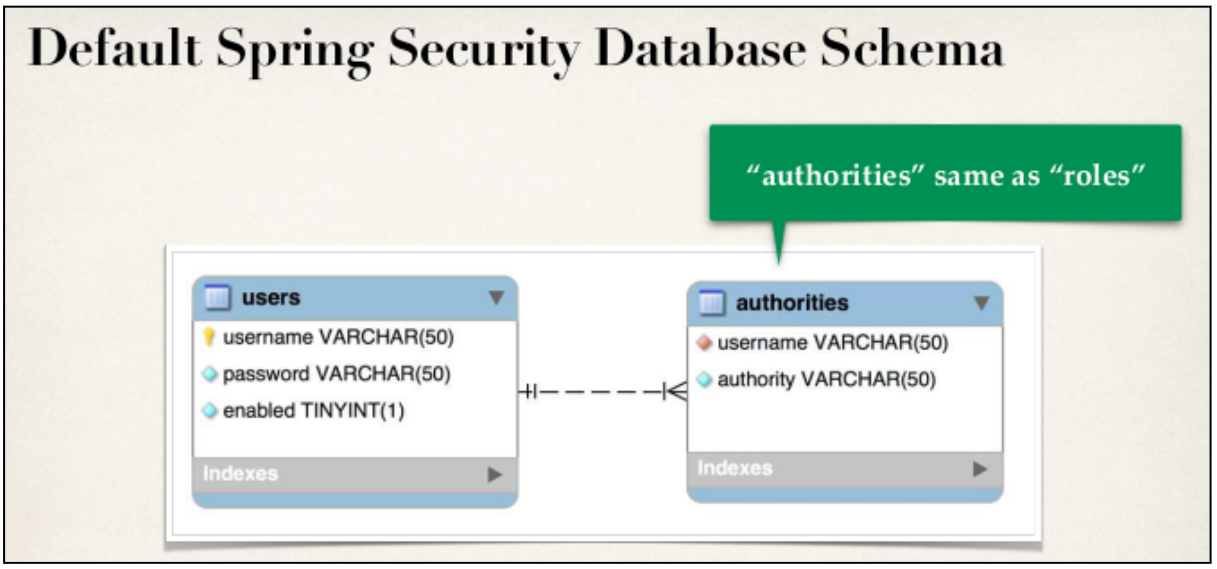
```
…
<div sec:authorize="hasRole('ADMIN')">
  <p>
     <a th:href="@{/systems}">
       IT Systems Meeting
     </a>
    (Only for Admin peeps)
  </p>
</div>
```

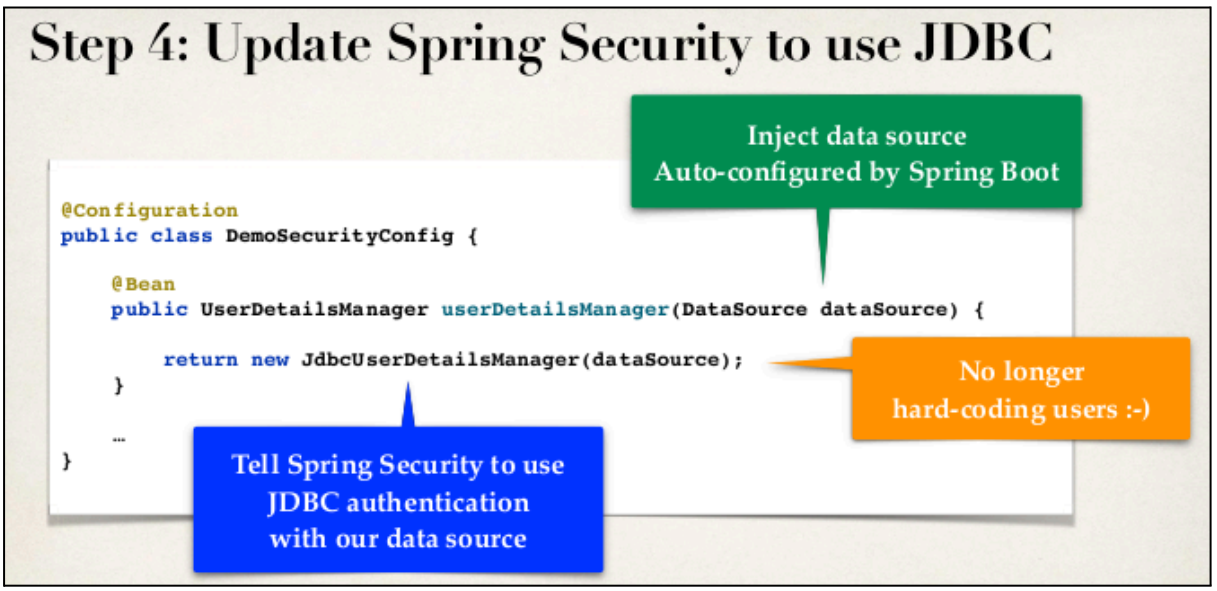
User: susan

Role(s): [ROLE_ADMIN, ROLE_EMPLOYEE]

IT Systems Meeting (Only for Admin peeps)

Spring Security User Accounts Stored in Database

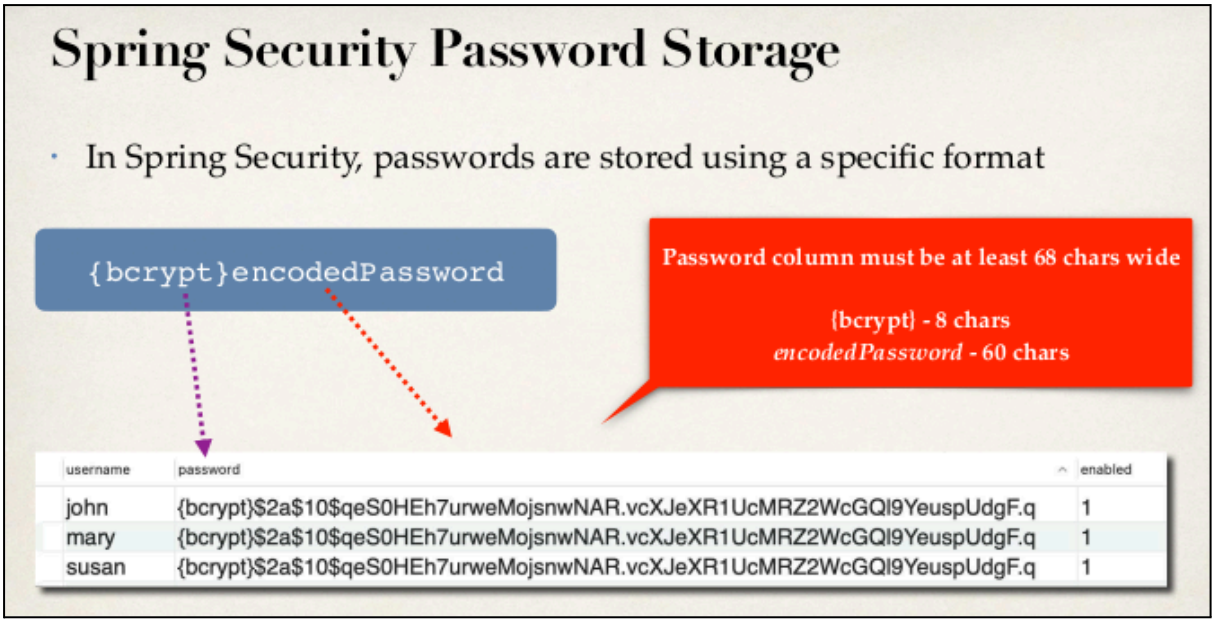


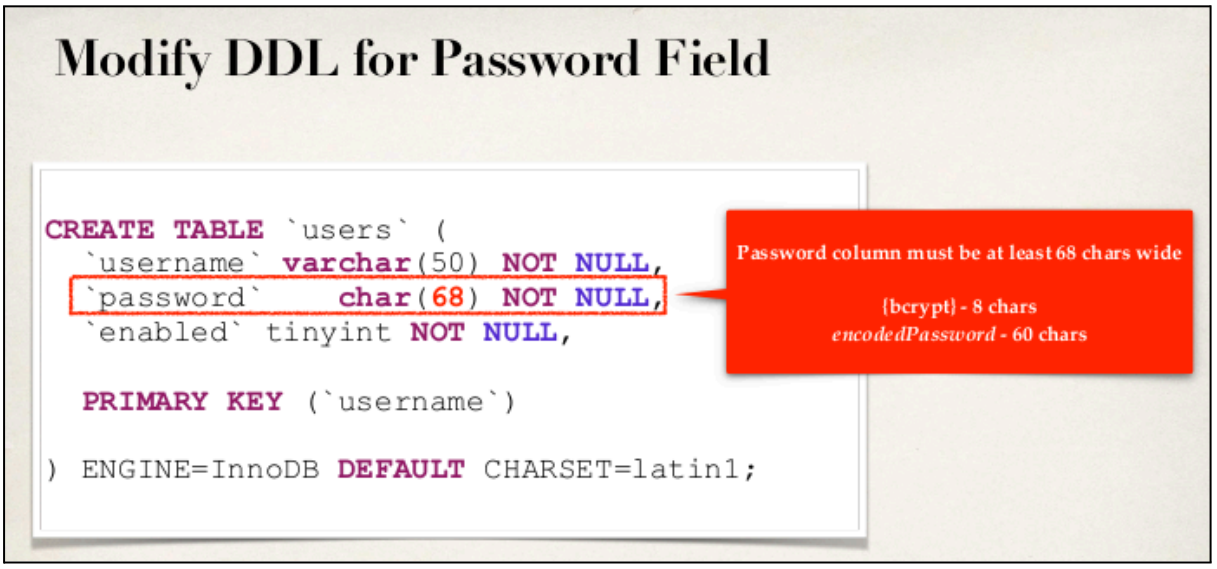


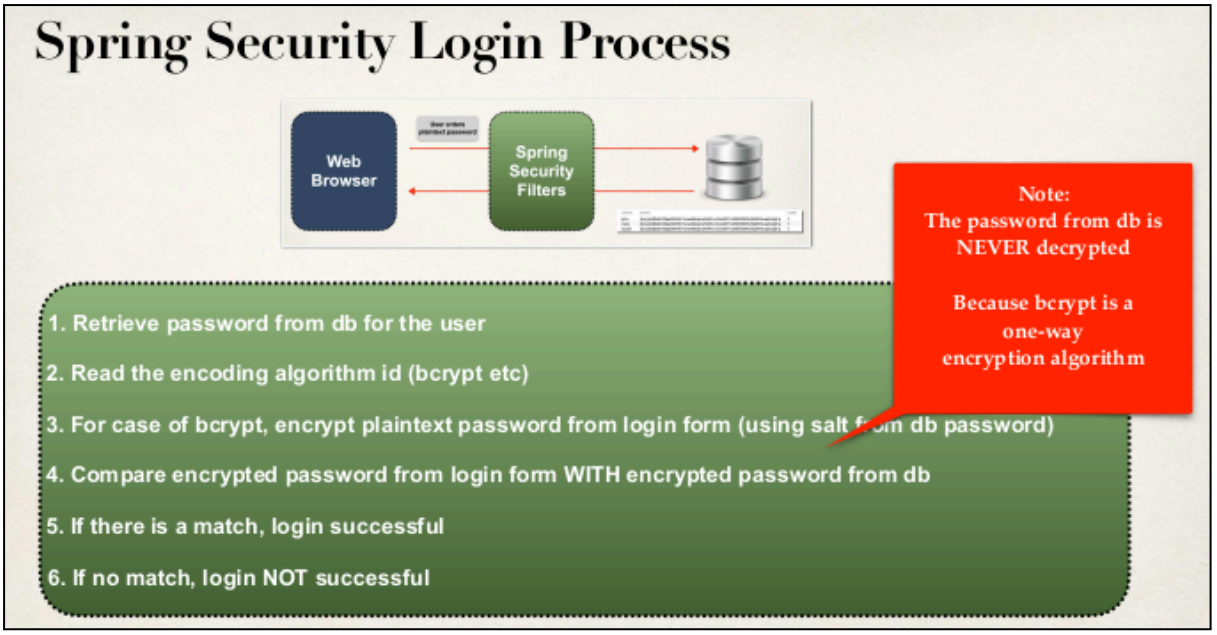
Spring Security Password Encryption

Spring Security Team Recommendation

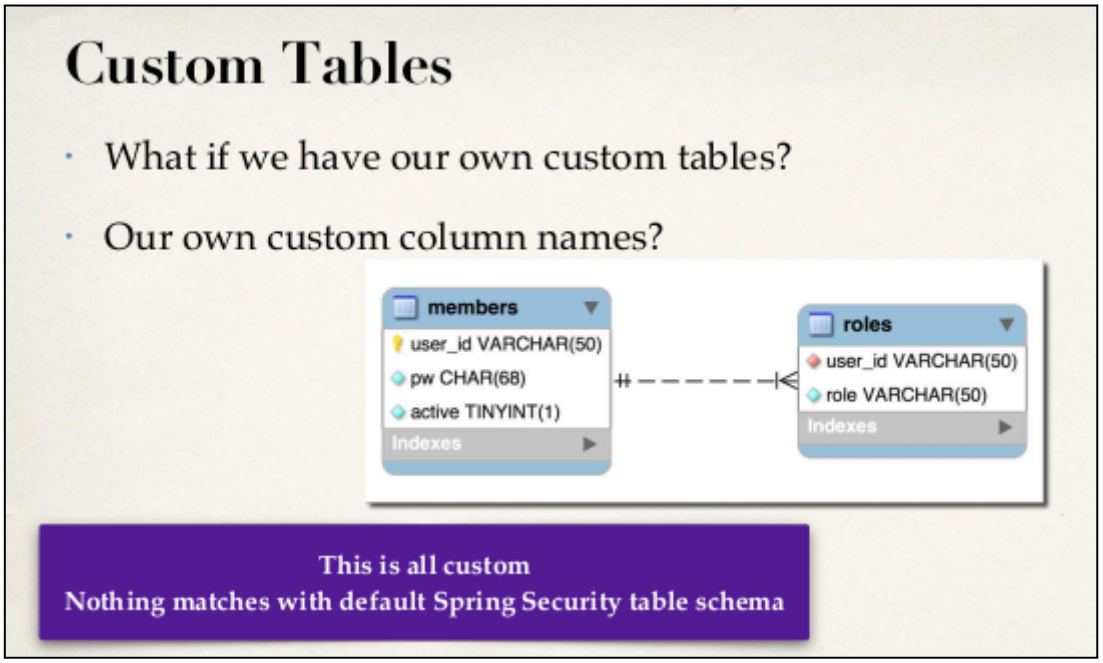
- Spring Security recommends using the popular bcrypt algorithm
- bcrypt
- Performs one-way encrypted hashing
- Adds a random salt to the password for additional protection
- Includes support to defeat brute force attacks







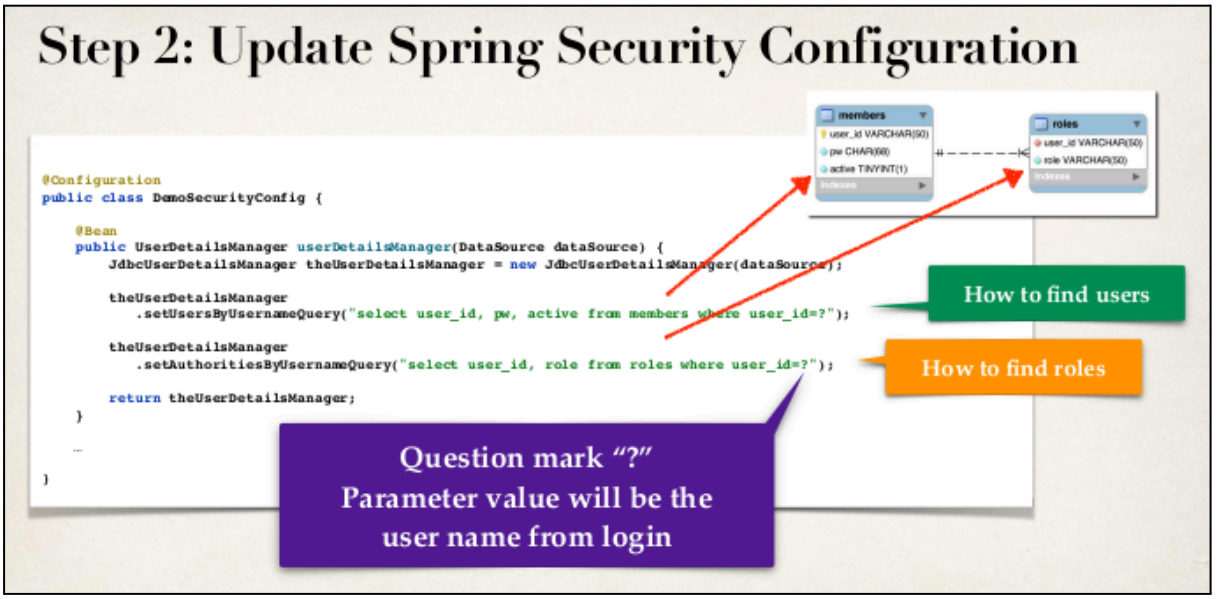
Spring Security Custom Tables



For Security Schema Customization

- Tell Spring how to query your custom tables
- Provide query to nd user by user name
- Provide query to nd authorities / roles by user name

Development Process
1. Create our custom tables with SQL
2. Update Spring Security Configuration
    Provide query to find user by user name
    Provide query to nd authorities / roles by user name



Custom Table with JPA/Hibernate

PDF
https://www.luv2code.com/bonus-lecture-spring-mvc-security-jpa-hibernate-bcrypt-pdf

Source Code
https://www.luv2code.com/bonus-lecture-spring-mvc-security-jpa-hibernate-bcrypt-code

We'll also cover the steps of encrypting the user's password using Java code.

PDF: bonus-lecture-spring-boot-spring-mvc-security-user-registration-pdf

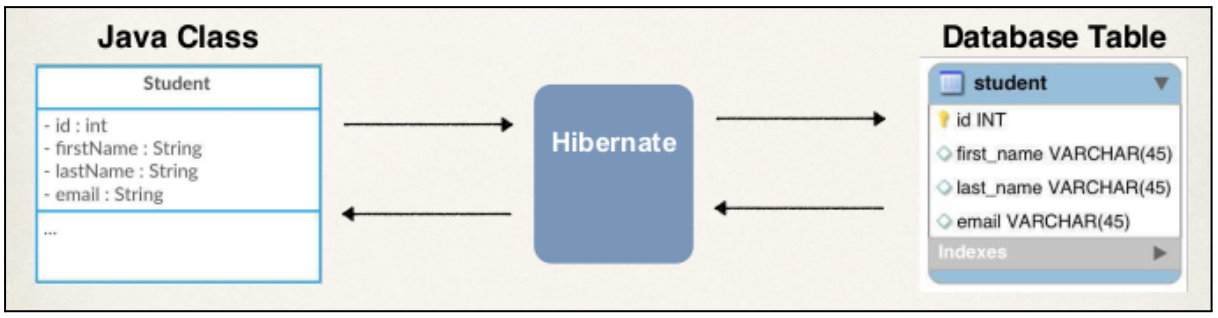
Source Code: bonus-lecture-spring-boot-spring-mvc-security-user-registration-code

It shows you how to set up a public landing page.

PDF: bonus-lecture-spring-boot-spring-mvc-security-landing-page.pdf

Source code: bonus-code-spring-boot-spring-mvc-security-landing-page.zip

JPA / Hibernate Advanced Mappings



Primary Key and Foreign Key

Primary key :

- identify a unique row in a table

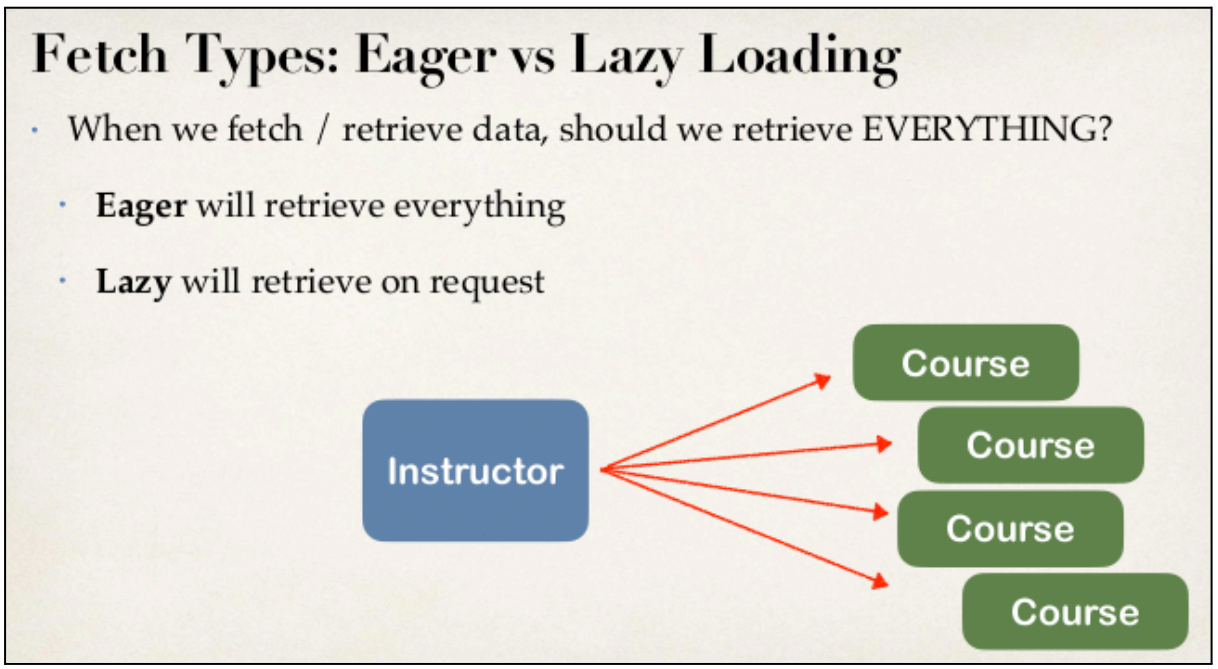
Foreign key :

- Link tables together
- a field in one table that refers to primary key in another table

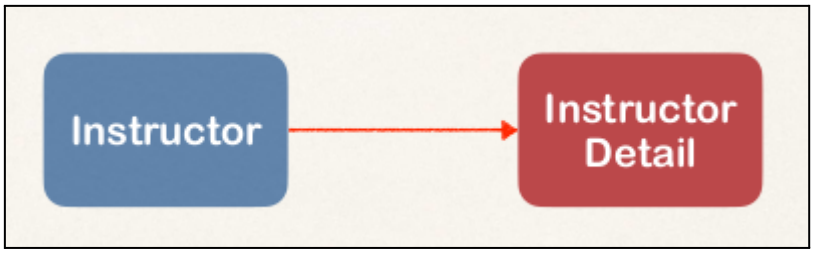
Cascade

- You can cascade operations
- Apply the same operation to related entities

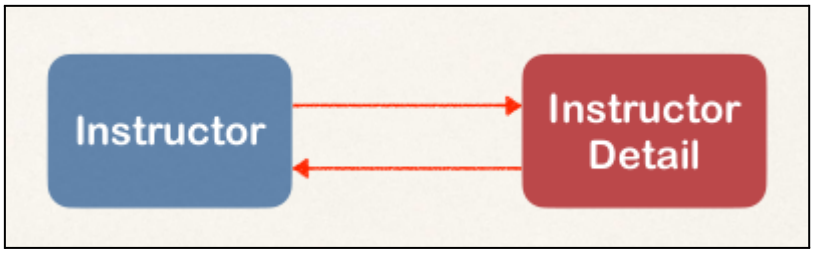
- If we delete an instructor, we should also delete their instructor_detail
- This is known as “CASCADE DELETE”



Uni Directional



Bi Directional



JPA / Hibernate One-to-One

- An instructor can have an “instructor detail” entity
- Similar to an “instructor profile”

Foreign Key

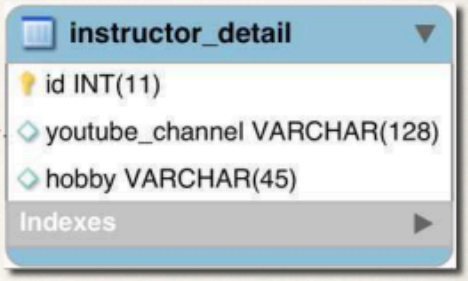
- Link tables together
- A field in one table that refers to primary key in another table

More on Foreign Key

- Main purpose is to preserve relationship between tables
    - Referential Integrity
- Prevents operations that would destroy relationship
- Ensures only valid data is inserted into the foreign key column
    - Can only contain valid reference to primary key in other table

Step 2: Create InstructorDetail class

```
@Entity
@Table(name="instructor_detail")
public class InstructorDetail {

   @Id
   @GeneratedValue(strategy=GenerationType.IDENTITY)
   @Column(name="id")
   private int id;

   @Column(name="youtube_channel")
   private String youtubeChannel;

   @Column(name="hobby")
   private String hobby;

   // constructors

   // getters / setters
}
```

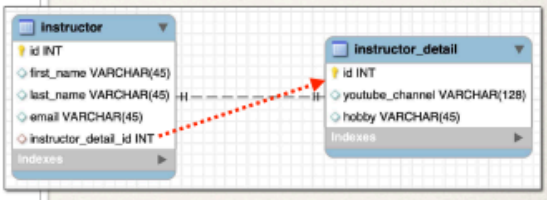


Step 3: Create Instructor class

```
@Entity
@Table(name="instructor")
public class Instructor {

   @Id
   @GeneratedValue(strategy=GenerationType.IDENTITY)
   @Column(name="id")
   private int id;

   @Column(name="first_name")
   private String firstName;

   @Column(name="last_name")
   private String lastName;

   @Column(name="email")
   private String email;

   ...
   // constructors, getters / setters
}
```

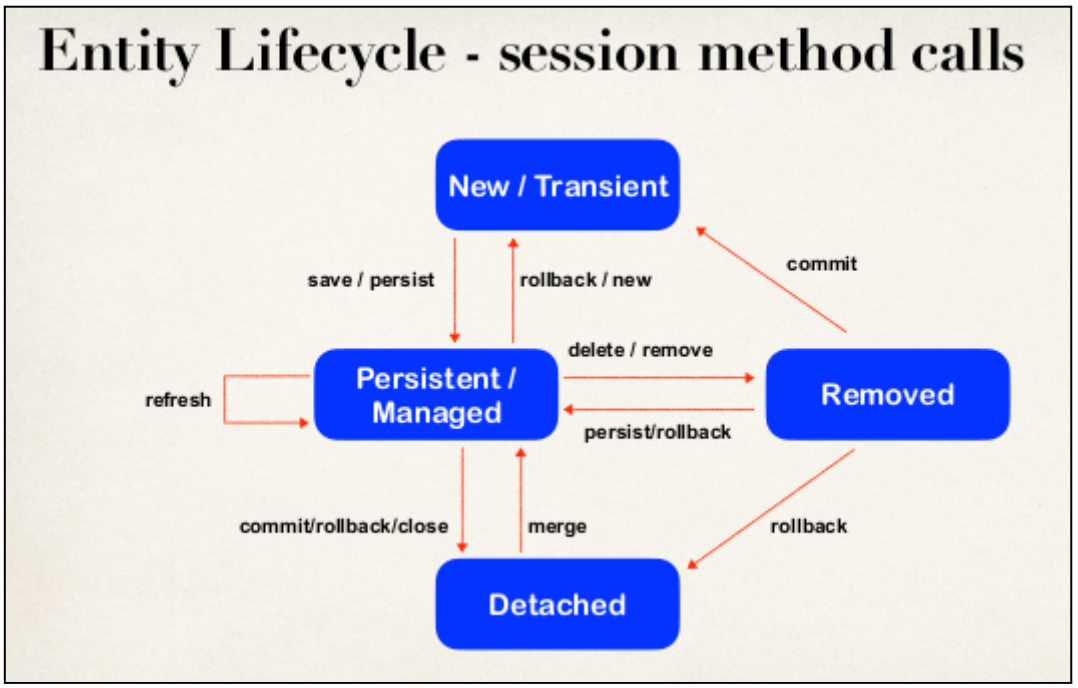
Step 3: Create Instructor class - @OneToOne

```
@Entity
@Table(name="instructor")
public class Instructor {
   ...

   @OneToOne
   @JoinColumn(name="instructor_detail_id")
   private InstructorDetail instructorDetail;

   ...
   // constructors, getters / setters
}
```



Entity Life Cycle

Operations	Description
Detach	If entity is detached, it is not associated with a Hibernate session
Merge	If instance is detached from session, then merge will reattach to session
Persist	Transitions new instances to managed state. Next flush / commit will save in db.
Remove	Transitions managed entity to be removed. Next flush / commit will delete from db.
Refresh	Reload / synch object with data from db. Prevents stale data



@OneToOne - Cascade Types

Cascade Type	Description
PERSIST	If entity is persisted / saved, related entity will also be persisted
REMOVE	If entity is removed / deleted, related entity will also be deleted
REFRESH	If entity is refreshed, related entity will also be refreshed
DETACH	If entity is detached (not associated w/ session), then related entity will also be detached
MERGE	If entity is merged, then related entity will also be merged
ALL	All of above cascade types

Configure Cascade Type

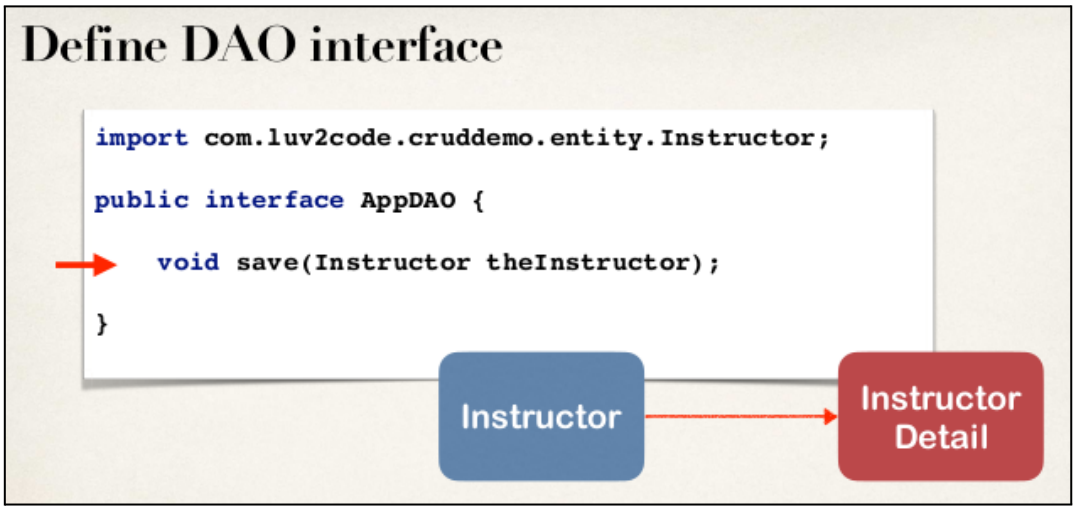

```
@Entity
@Table(name="instructor")
public class Instructor {
   ...

   @OneToOne(cascade=CascadeType.ALL)
   @JoinColumn(name="instructor_detail_id")
   private InstructorDetail instructorDetail;

   ...
   // constructors, getters / setters
}
```

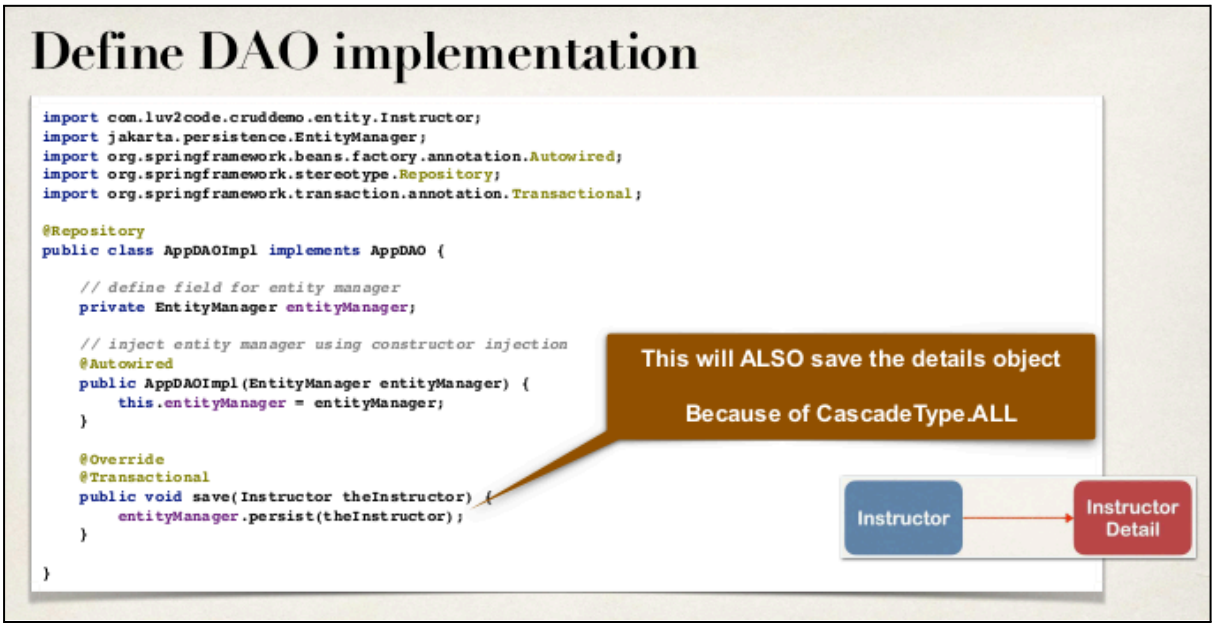
By default, no operations are cascaded.

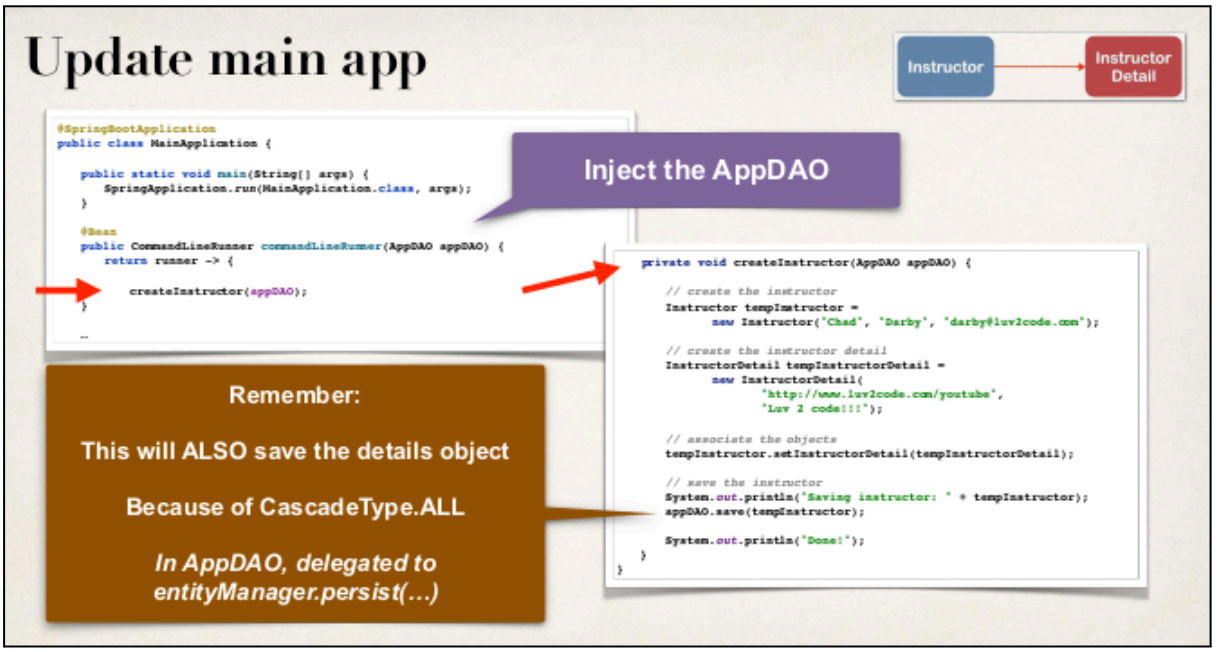
Configure Multiple Cascade Types

```
@OneToOne(cascade={CascadeType.DETACH,
                   CascadeType.MERGE,
                   CascadeType.PERSIST,
                   CascadeType.REFRESH,
                   CascadeType.REMOVE})
```

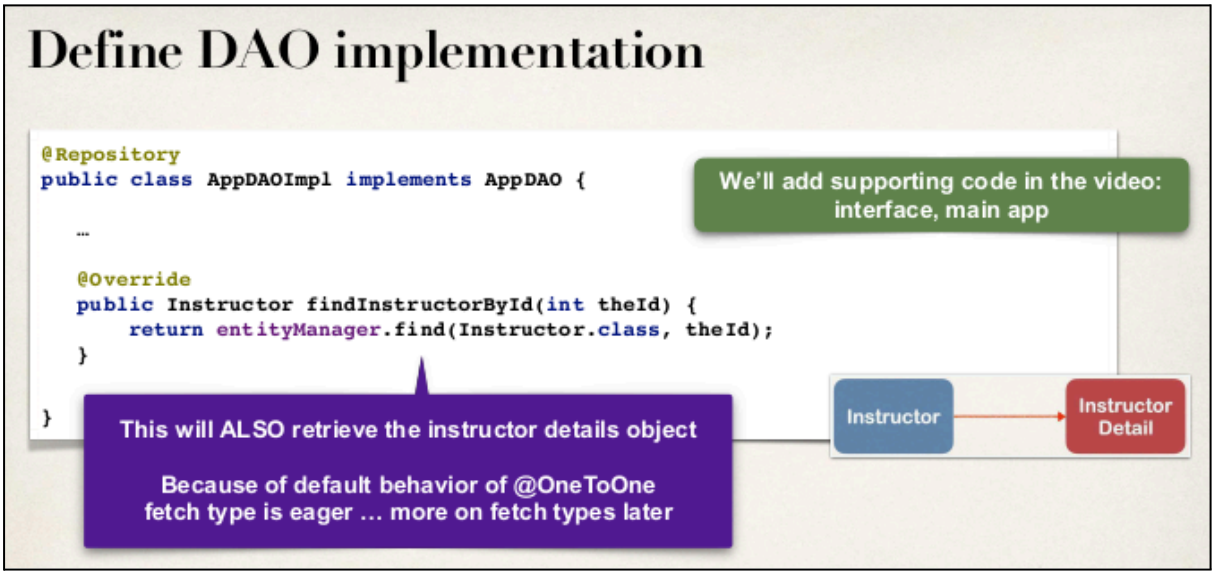
Create Spring Boot Command Line App



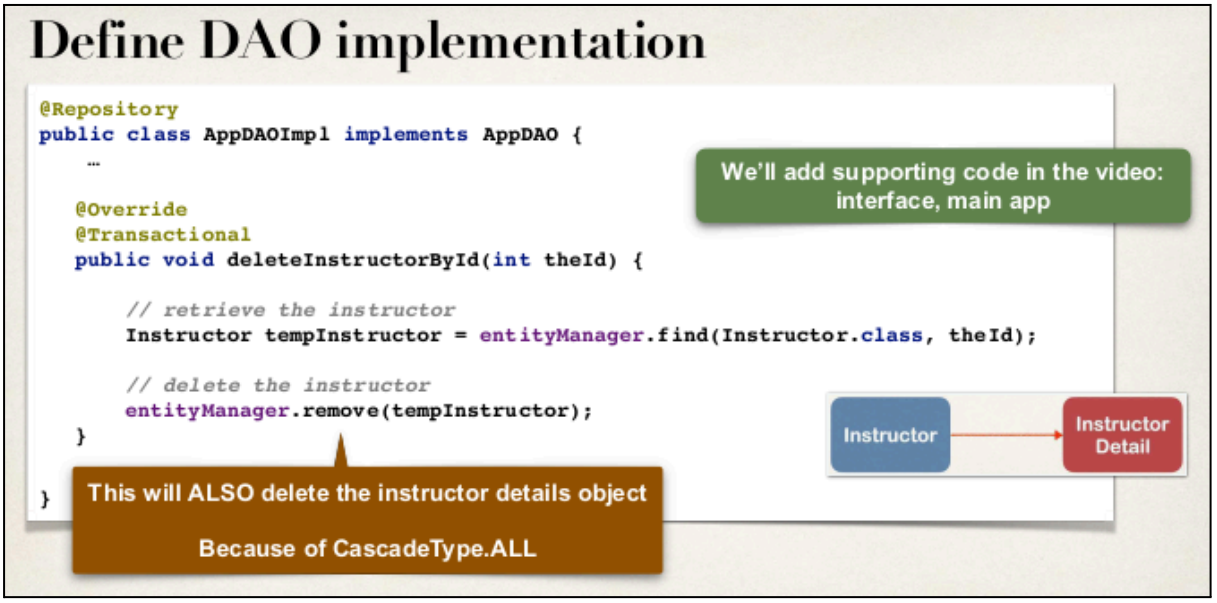




JPA / Hibernate One-to-One: Find an entity



JPA / Hibernate One-to-One: Delete an entity



JPA / Hibernate One-to-One: Bi-Directional

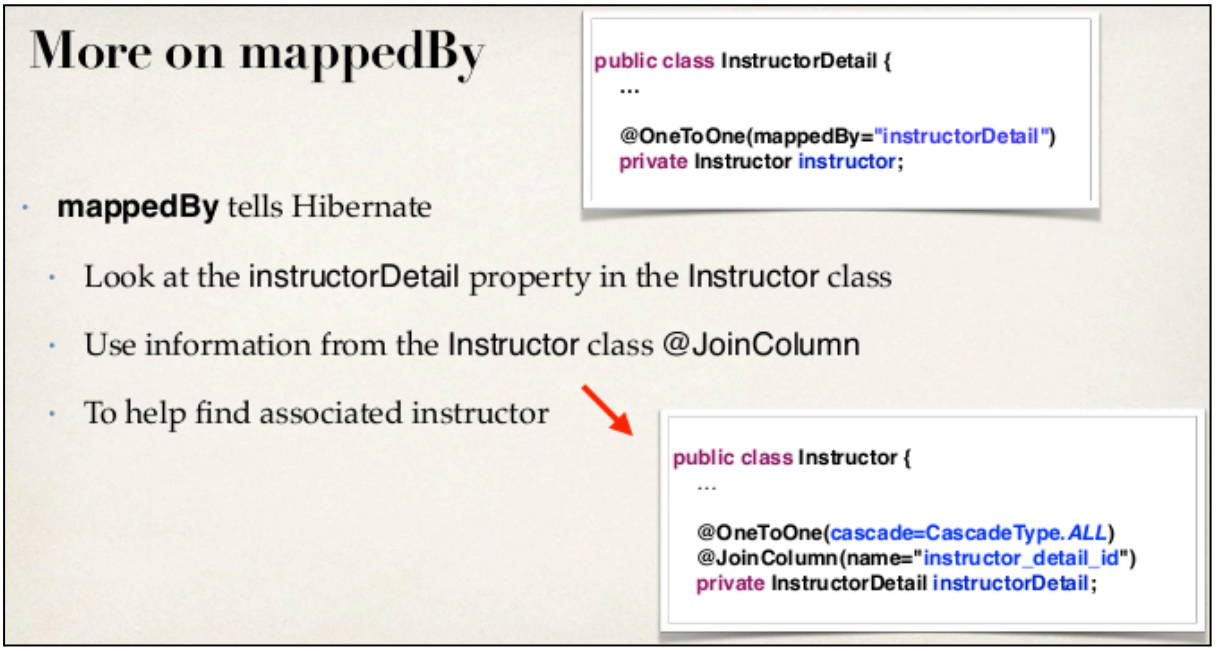
New Use Case

- If we load an InstructorDetail
- Then we'd like to get the associated Instructor
- Can't do this with the current unidirectional relationship :-(

- Bi-Directional relationship is the solution
- We can start with InstructorDetail and make it back to the Instructor

To use Bi-Directional,

- we can keep the existing database schema
- No changes required to database



JPA / Hibernate One-to-Many

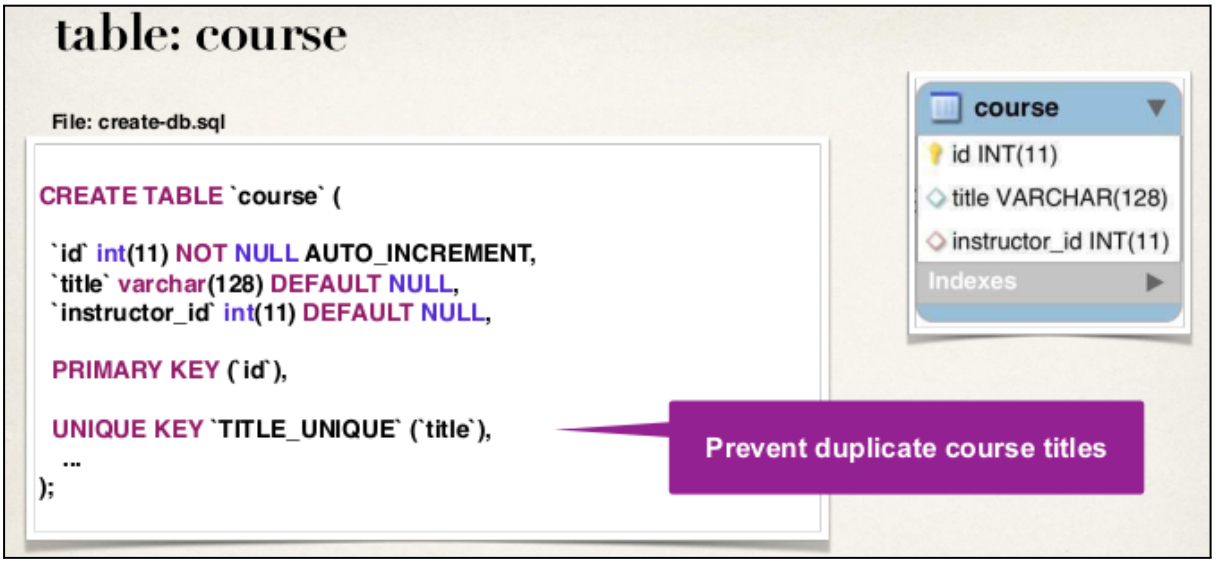


table: course - foreign key

File: create-db.sql

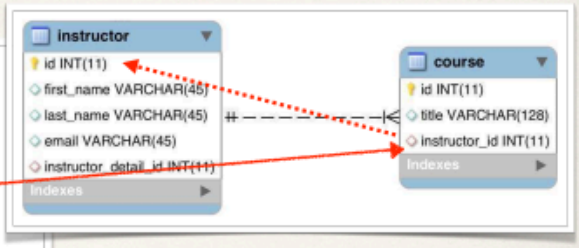
```
CREATE TABLE `course` (
  ...
  KEY `FK_INSTRUCTOR_idx` (`instructor_id`),
  CONSTRAINT `FK_INSTRUCTOR`
  FOREIGN KEY (`instructor_id`)
  REFERENCES `instructor` (`id`)
  ...
);
```

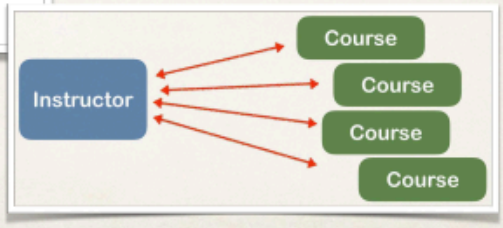
Step 2: Create Course class

```
@Entity
@Table(name="course")
public class Course {

   @Id
   @GeneratedValue(strategy=GenerationType.IDENTITY)
   @Column(name="id")
   private int id;

   @Column(name="title")
   private String title;

   ...
   // constructors, getters / setters
}
```

Step 2: Create Course class - @ManyToOne

```
@Entity
@Table(name="course")
public class Course {
   ...

   @ManyToOne
   @JoinColumn(name="instructor_id")
   private Instructor instructor;

   ...
   // constructors, getters / setters
}
```





More: mappedBy

```
public class Instructor {
   ...

   @OneToMany(mappedBy="instructor")
   private List<Course> courses;
```

- **mappedBy** tells Hibernate
  - Look at the instructor property in the Course class
  - Use information from the Course class @JoinColumn
  - To help find associated courses for instructor

```
public class Course {
   ...

   @ManyToOne
   @JoinColumn(name="instructor_id")
   private Instructor instructor;
```

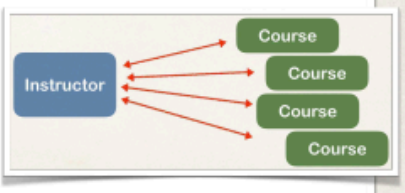
Add support for Cascading

```
@Entity
@Table(name="course")
public class Course {
   ...

   @ManyToOne(cascade={CascadeType.PERSIST, CascadeType.MERGE
                       CascadeType.DETACH, CascadeType.REFRESH})
   @JoinColumn(name="instructor_id")
   private Instructor instructor;

   ...
   // constructors, getters / setters
}
```

Do not apply cascading deletes!

Add convenience methods for bi-directional

```
@Entity
@Table(name="instructor")
public class Instructor {
   ...
   // add convenience methods for bi-directional relationship

   public void add(Course tempCourse) {

      if (courses == null) {
         courses = new ArrayList<>();
      }

      courses.add(tempCourse);

      tempCourse.setInstructor(this);
   }
   ...
}
```



Fetch Types: Eager vs Lazy

Fetch Types: Eager vs Lazy Loading

- When we fetch / retrieve data, should we retrieve EVERYTHING?
- Eager will retrieve everything
- Lazy will retrieve on request

Eager Loading

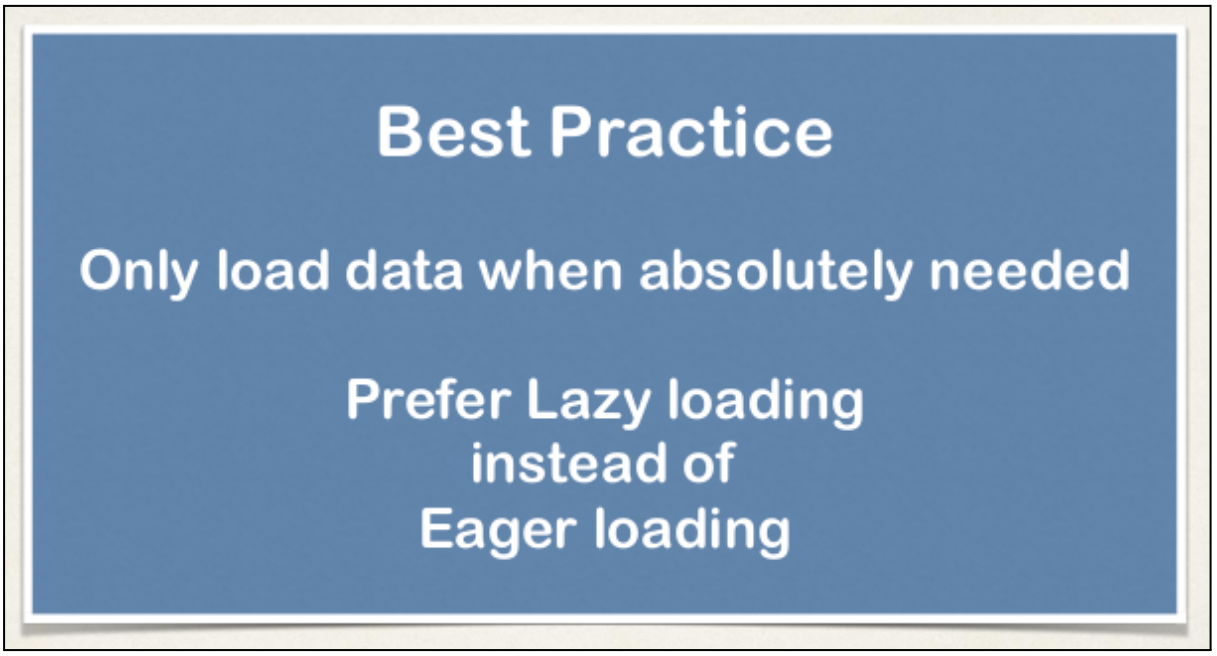
- Eager loading will load all dependent entities
- Load instructor and all of their courses at once

Eager Loading

- What about courses and students?
- Could easily turn into a performance nightmare …

Eager Loading

- In our app, if we are searching for a course by keyword
- Only want a list of matching courses
- Eager loading would still load all students for each course …. not good!



Lazy Loading

- Lazy loading will load the main entity first
- Load dependent entities on demand (lazy)

Fetch Type

- When you define the mapping relationship
  - You can specify the fetch type: EAGER or LAZY


```
@Entity
@Table(name="instructor")
public class Instructor {
  ...

  @OneToMany(fetch=FetchType.LAZY, mappedBy="instructor")
  private List<Course> courses;

  ...

}
```

Default Fetch Types

Mapping	Default Fetch Type
@OneToOne	FetchType.EAGER
@OneToMany	FetchType.LAZY
@ManyToOne	FetchType.EAGER
@ManyToMany	FetchType.LAZY

Overriding Default Fetch Type

- Specifying the fetch type, overrides the defaults

Mapping	Default Fetch Type
@ManyToOne	FetchType.EAGER

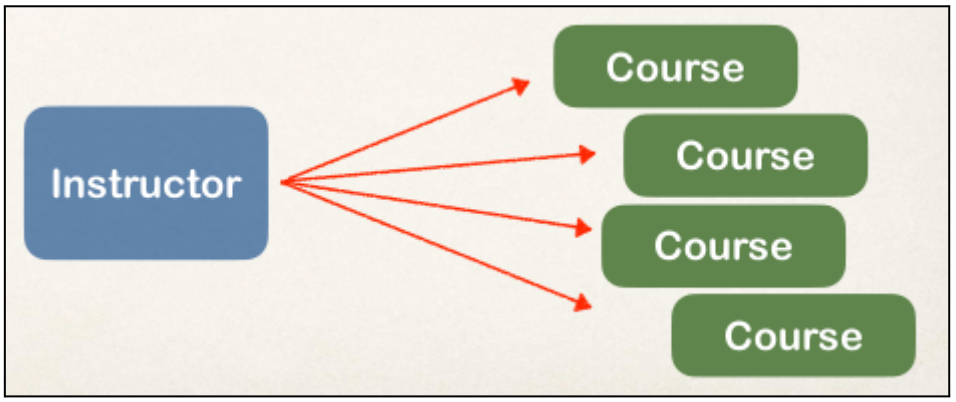
```
@ManyToOne(fetch=FetchType.LAZY)
@JoinColumn(name="instructor_id")
private Instructor instructor;
```

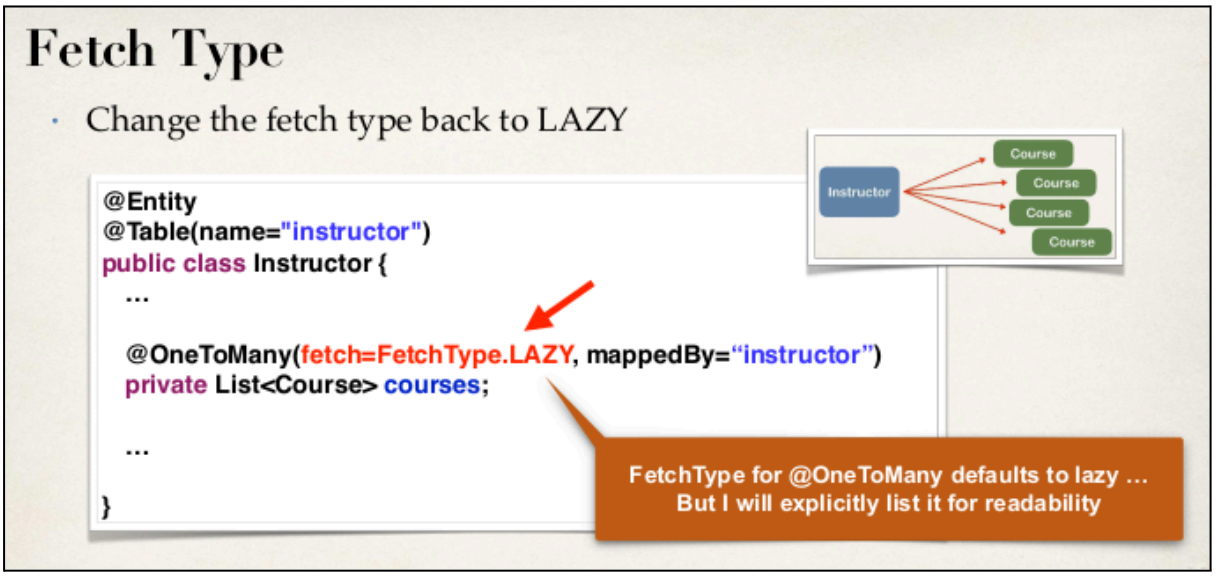
More about Lazy Loading

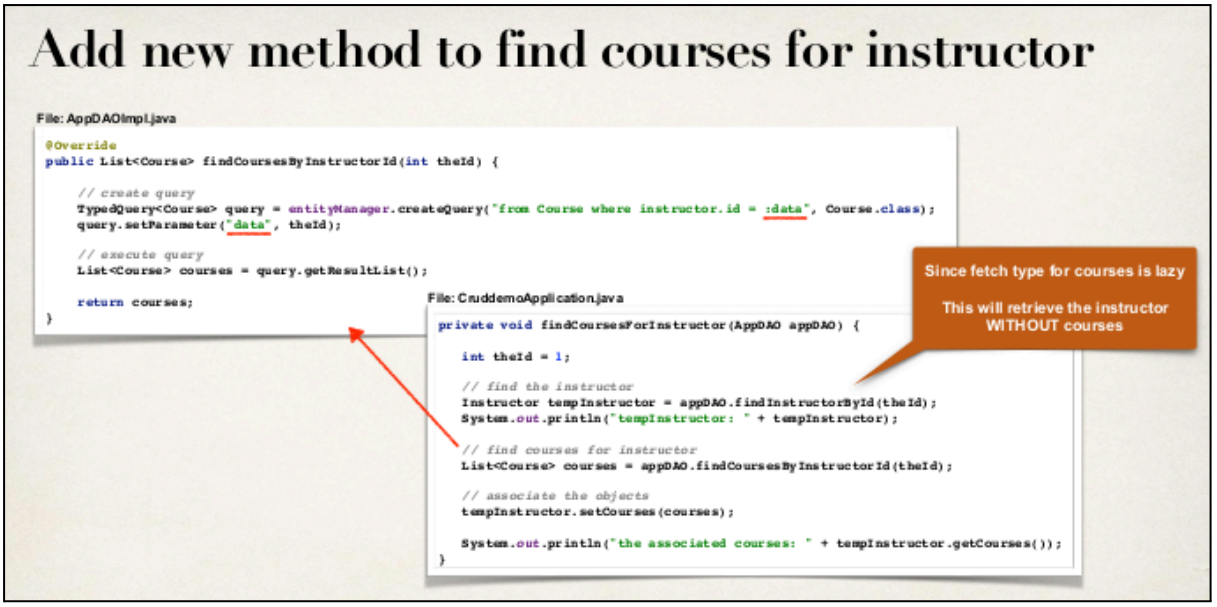
- When you lazy load, the data is only retrieved on demand
- However, this requires an open Hibernate session
- need an connection to database to retrieve data
- If the Hibernate session is closed
- And you attempt to retrieve lazy data
- Hibernate will throw an exception

Previous Solution: Eager

- Eager will retrieve everything … all of the courses for an instructor
- But we may not want this ALL of the time
- We'd like the option to load courses as needed …

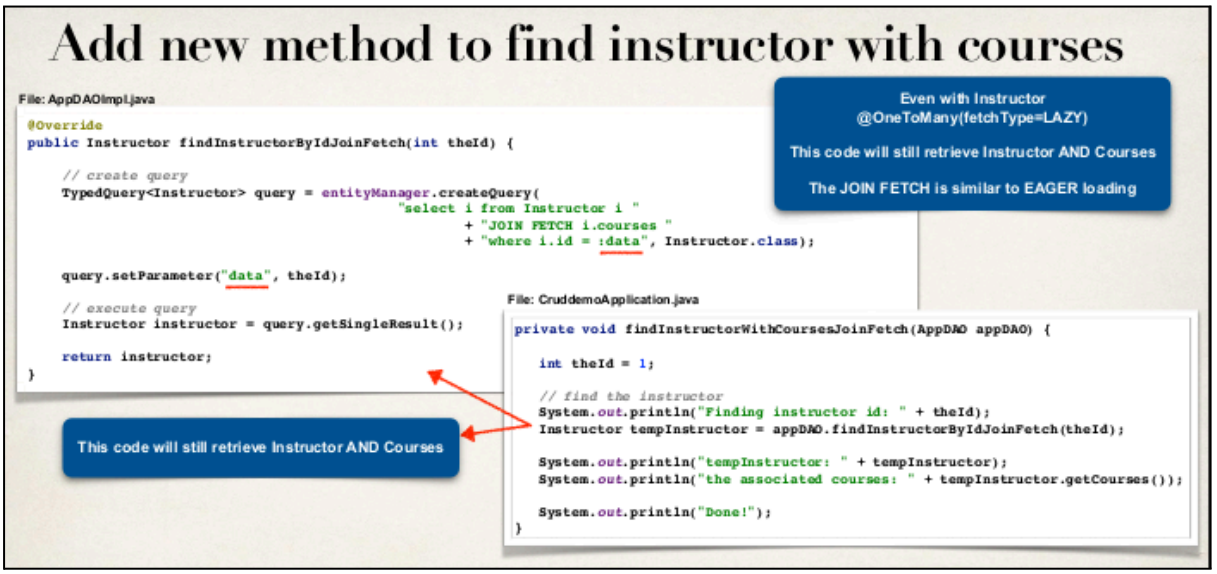






Previous Solution: Find Courses for Instructor

- Previous solution was OK … but …
- Required an extra query
- I wish we could have a new method that would
- Get instructor AND courses … in a single query
- Also keep the LAZY option available … don't change fetch type



We have options now

- If you only need Instructor … and no courses, then call
    - If you need Instructor AND courses, then call
- appDAO.findInstructorById(…)
    - appDAO.findInstructorByIdJoinFetch(…)
- This gives us flexibility instead of having EAGER fetch hard-coded

@OneToMany: Update Instructor

Add new DAO method to update instructor

File: AppDAOImpl.java

```
@Override
@Transactional
public void update(Instructor tempInstructor) {
    entityManager.merge(tempInstructor);
}
```

merge(...) will update an existing entity

Main app

File: CruddemoApplication.java

```
private void updateInstructor(AppDAO appDAO) {

  int theId = 1;

  System.out.println("Finding instructor id: " + theId);
  Instructor tempInstructor = appDAO.findInstructorById(theId);

  System.out.println("Updating instructor id: " + theId);
  tempInstructor.setLastName("TESTER");

  appDAO.update(tempInstructor);

  System.out.println("Done");
}
```

Change instructor's data

Call DAO method to update database

@OneToMany: Update Course

Add new DAO method to update course

File: AppDAOImpl.java

```
@Override
@Transactional
public void update(Course tempCourse) {
    entityManager.merge(tempCourse);
}
```

merge(...) will update an existing entity

Main app

File: CruddemoApplication.java

```
private void updateCourse(AppDAO appDAO) {

  int theId = 10;

  System.out.println("Finding course id: " + theId);
  Course tempCourse = appDAO.findCourseById(theId);

  System.out.println("Updating course id: " + theId);
  tempCourse.setTitle("Enjoy the Simple Things");

  appDAO.update(tempCourse);

  System.out.println("Done");
}
```

Change course's data

Call DAO method to update database

@OneToMany: Delete Instructor

Add new DAO method to delete instructor

File: AppDAOImpl.java

```
@Override
@Transactional
public void deleteInstructorById(int theId) {

    // retrieve the instructor
    Instructor tempInstructor = entityManager.find(Instructor.class, theId);

    List<Course> courses = tempInstructor.getCourses();

    // break associations of all courses for instructor
    for (Course tempCourse : courses) {
        tempCourse.setInstructor(null);
    }

    // delete the instructor
    entityManager.remove(tempInstructor);
}
```

Remove the instructor from the courses

We only delete the instructor ... not the associated course based on our cascade types

Error message

- If you don't remove instructor from courses … **constraint violation**

```
Caused by: java.sql.SQLIntegrityConstraintViolationException:

Cannot delete or update a parent row: a foreign key constraint fails

(`hb-03-one-to-many`.`course`,
CONSTRAINT `FK_INSTRUCTOR` FOREIGN KEY (`instructor_id`) REFERENCES `instructor` (`id`))
```

- An instructor can not be deleted if it is referenced by a course
- You must remove the instructor from the course first

Main app

File: CruddemoApplication.java

```
private void deleteInstructor(AppDAO appDAO) {

  int theId = 1;
  System.out.println("Deleting instructor id: " + theId);

  appDAO.deleteInstructorById(theId);

  System.out.println("Done!");
}
```

@OneToMany: Delete Course

Add new DAO method to delete course

File: AppDAOImpl.java

```
@Override
@Transactional
public void deleteCourseById(int theId) {

    // retrieve the course
    Course tempCourse = entityManager.find(Course.class, theId);

    // delete the course
    entityManager.remove(tempCourse);
}
```

Main app

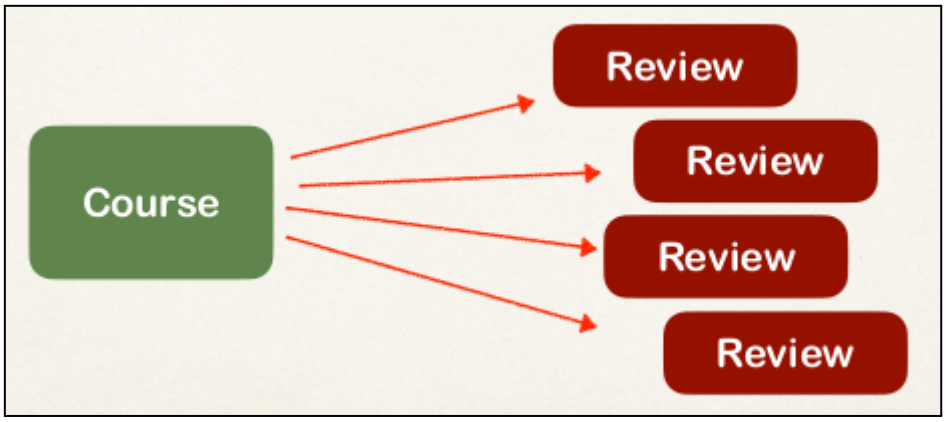
File: CruddemoApplication.java

```
private void deleteCourseById(AppDAO appDAO) {

  int theId = 10;
  System.out.println("Deleting course id: " + theId);

  appDAO.deleteCourseById(theId);

  System.out.println("Done!");

}
```

@OneToMany: Uni-Directional

One-to-Many Mapping

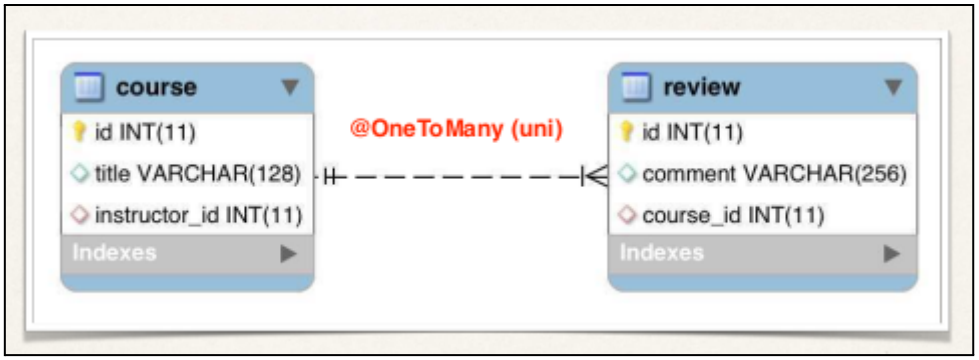
- A course can have many reviews
    - Uni-directional

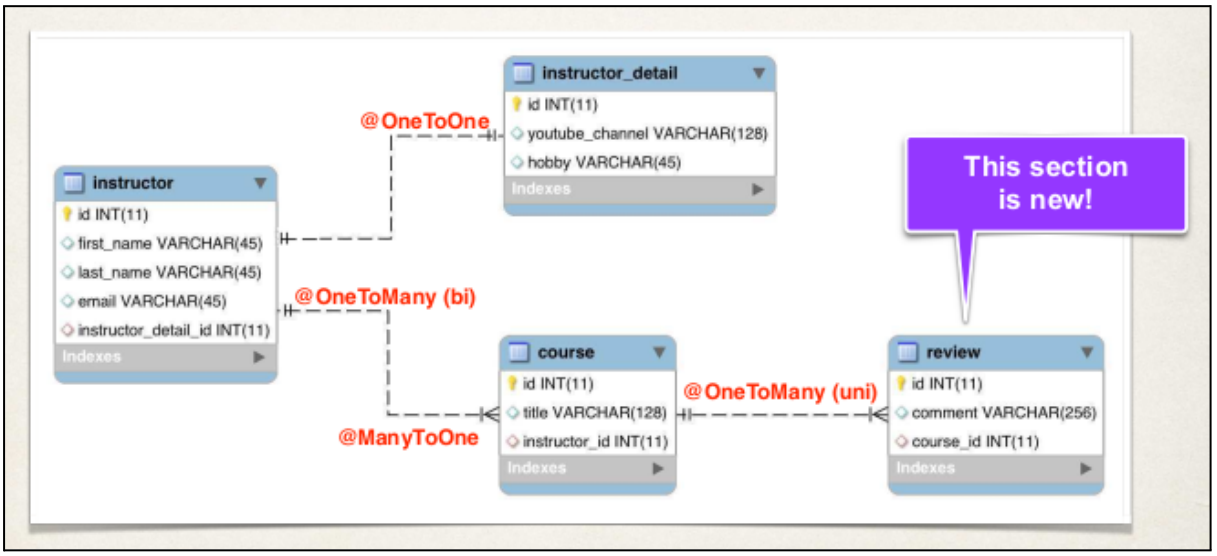


Real-World Project Requirement

- If you delete a course, also delete the reviews
- Reviews without a course … have no meaning

@OneToMany


course
id INT(11)
title VARCHAR(128)
instructor_id INT(11)
Indexes
@OneToMany (uni)
review
id INT(11)
comment VARCHAR(256)
course_id INT(11)
Indexes


instructor_detail
id INT(11)
youtube_channel VARCHAR(128)
hobby VARCHAR(45)
Indexes
@OneToOne
instructor
id INT(11)
first_name VARCHAR(45)
last_name VARCHAR(45)
email VARCHAR(45)
instructor_detail_id INT(11)
Indexes
This section is new!
@OneToMany (bi)
course
id INT(11)
title VARCHAR(128)
instructor_id INT(11)
Indexes
@ManyToOne
@OneToMany (uni)
review
id INT(11)
comment VARCHAR(256)
course_id INT(11)
Indexes

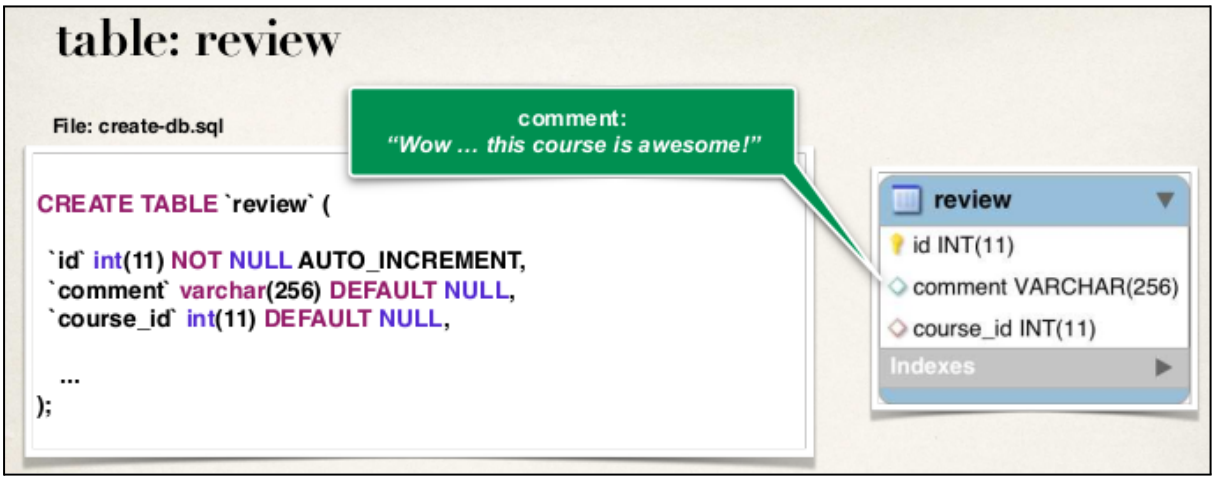


table: review
File: create-db.sql
CREATE TABLE `review` (
`id` int(11) NOT NULL AUTO_INCREMENT,
`comment` varchar(256) DEFAULT NULL,
`course_id` int(11) DEFAULT NULL,
...
);
comment:
"Wow ... this course is awesome!"
review
id INT(11)
comment VARCHAR(256)
course_id INT(11)
Indexes

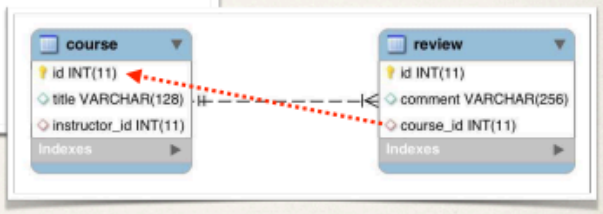
table: review - foreign key

File: create-db.sql

```
CREATE TABLE `review` (
  ...
  KEY `FK_COURSE_ID_idx` (`course_id`),
  CONSTRAINT `FK_COURSE`
  FOREIGN KEY (`course_id`)
  REFERENCES `course` (`id`)
  ...
);
```

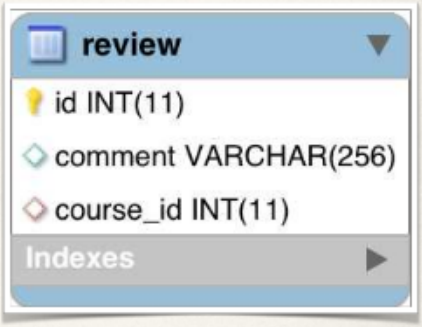
Table

Column



Step 2: Create Review class

```
@Entity
@Table(name="review")
public class Review {

   @Id
   @GeneratedValue(strategy=GenerationType.IDENTITY)
   @Column(name="id")
   private int id;

   @Column(name="comment")
   private String comment;

   ...
   // constructors, getters / setters
}
```



Add support for Lazy loading

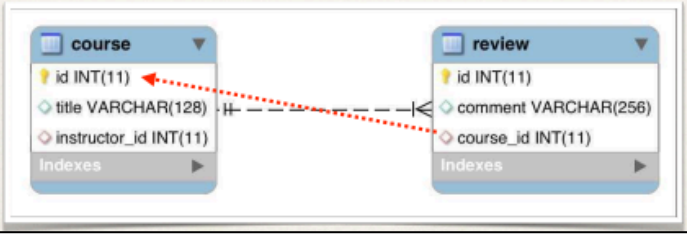
Lazy load the reviews

```
@Entity
@Table(name="course")
public class Course {
   ...

   @OneToMany(fetch=FetchType.LAZY, cascade=CascadeType.ALL)
   @JoinColumn(name="course_id")
   private List<Review> reviews;

   ...

}
```

More: @JoinColumn

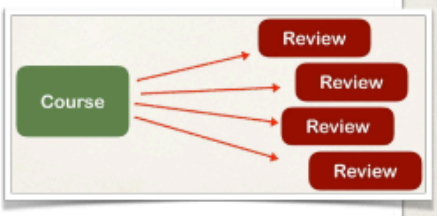
- In this scenario, **@JoinColumn** tells Hibernate
  - Look at the **course_id** column in the **review** table
  - Use this information to help find associated reviews for a course

```
public class Course {
   ...

   @OneToMany
   @JoinColumn(name="course_id")
   private List<Review> reviews;
```



Add convenience method for adding review

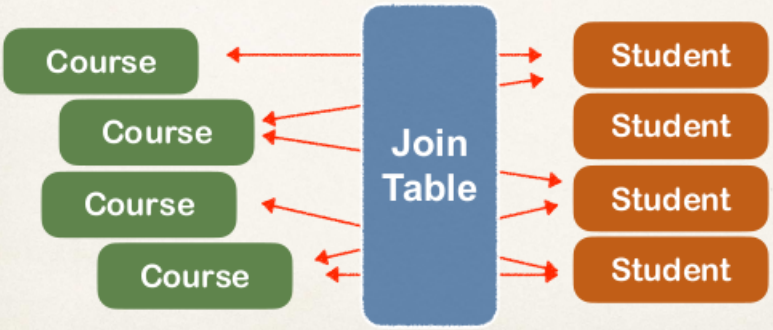
```
@Entity
@Table(name="course")
public class Course {
   ...
   // add convenience methods for adding reviews

   public void add(Review tempReview) {

      if (reviews == null) {
         reviews = new ArrayList<>();
      }

      reviews.add(tempReview);

   }
   ...
}
```



@ManyToMany

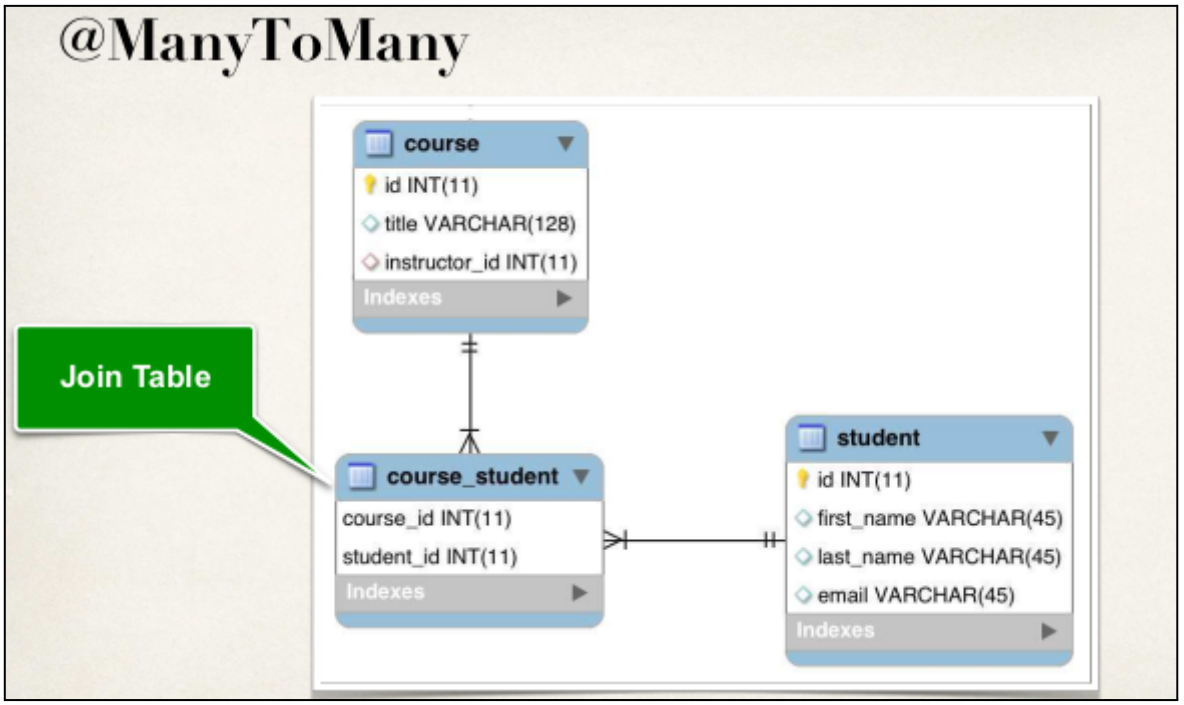
Keep track of relationships

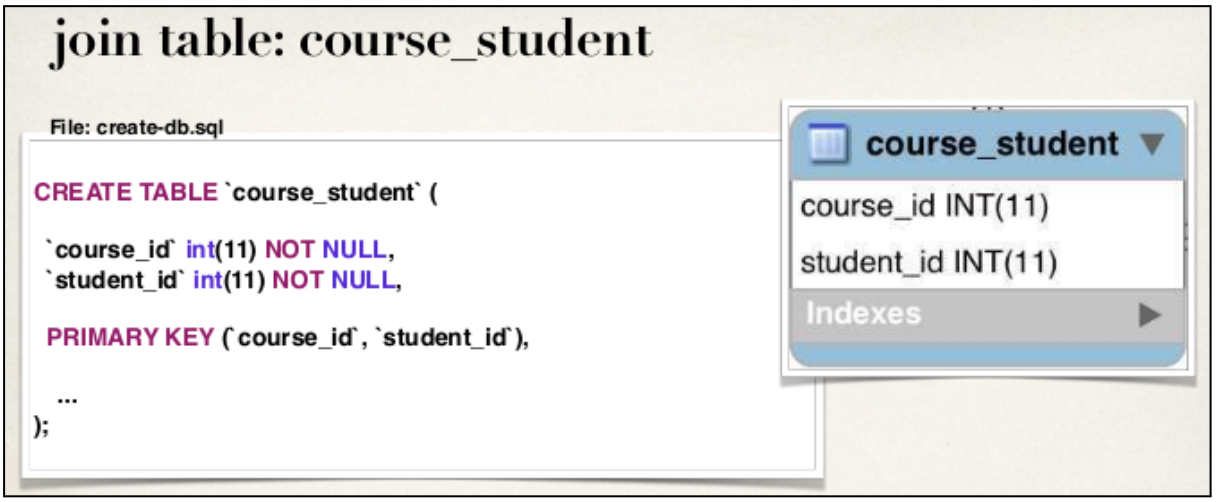
- Need to track which student is in which course and vice-versa



Join Table

A table that provides a mapping between two tables.
It has foreign keys for each table to define the mapping relationship.



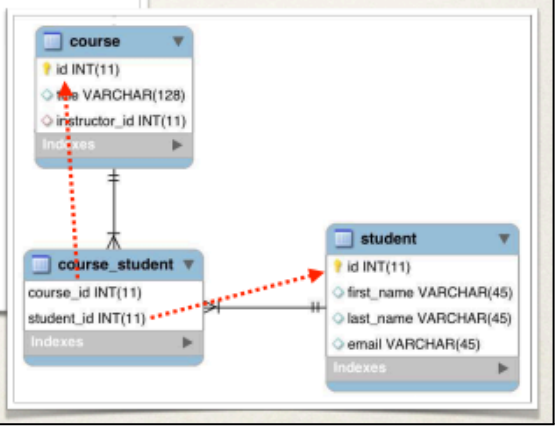


join table: course_student - foreign keys

```
CREATE TABLE `course_student` (
  ...
  CONSTRAINT `FK_COURSE_05`
  FOREIGN KEY (`course_id`)
  REFERENCES `course` (`id`),

  CONSTRAINT `FK_STUDENT`
  FOREIGN KEY (`student_id`)
  REFERENCES `student` (`id`)
  ...
);
```

Table

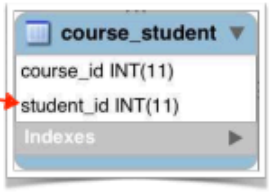
Column



Add @ManyToMany annotation

```
@Entity
@Table(name="course")
public class Course {
   ...

   @ManyToMany
   @JoinTable(
        name="course_student",
        joinColumns=@JoinColumn(name="course_id"),
        inverseJoinColumns=@JoinColumn(name="student_id")
        )
   private List<Student> students;

   // getter / setters
   ...

}
```

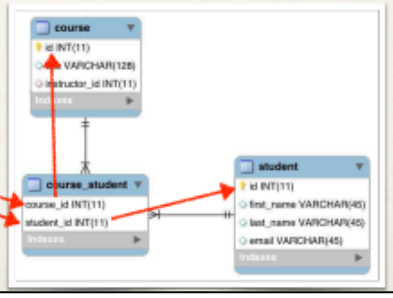
Refers to "student_id" column in "course_student" join table



More: @JoinTable

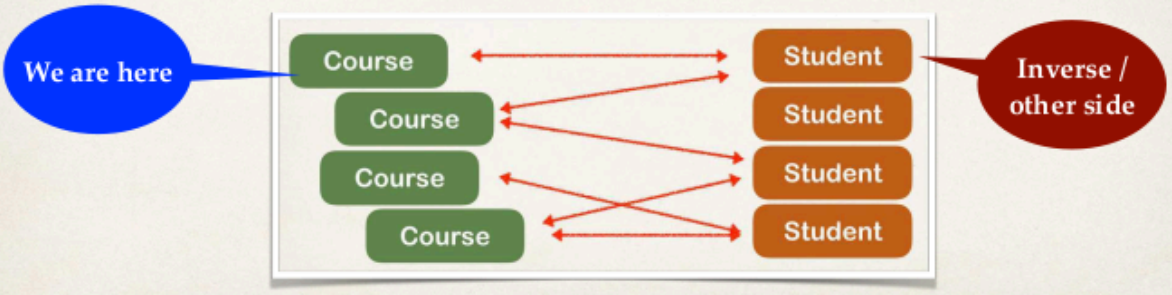
- **@JoinTable** tells Hibernate
  - Look at the **course_id** column in the **course_student** table
  - For other side (inverse), look at the **student_id** column in the **course_student** table
  - Use this information to find relationship between **course** and **students**

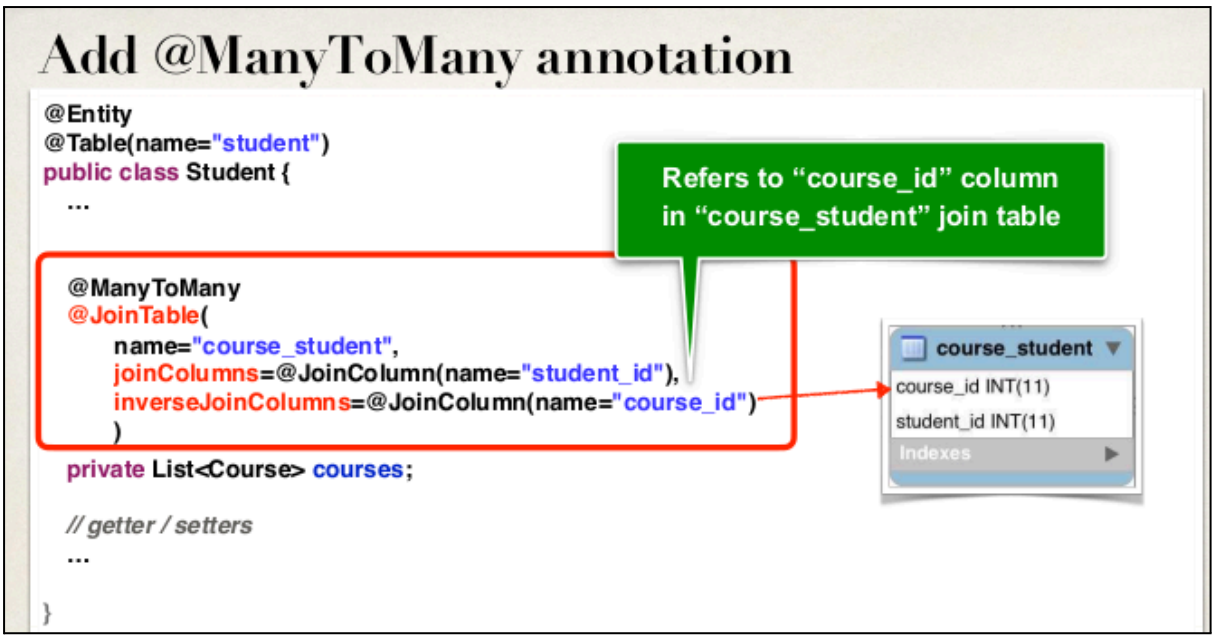
```
public class Course {

    @ManyToMany
    @JoinTable(
            name="course_student",
            joinColumns=@JoinColumn(name="course_id"),
            inverseJoinColumns=@JoinColumn(name="student_id")
            )
    private List<Student> students;

}
```

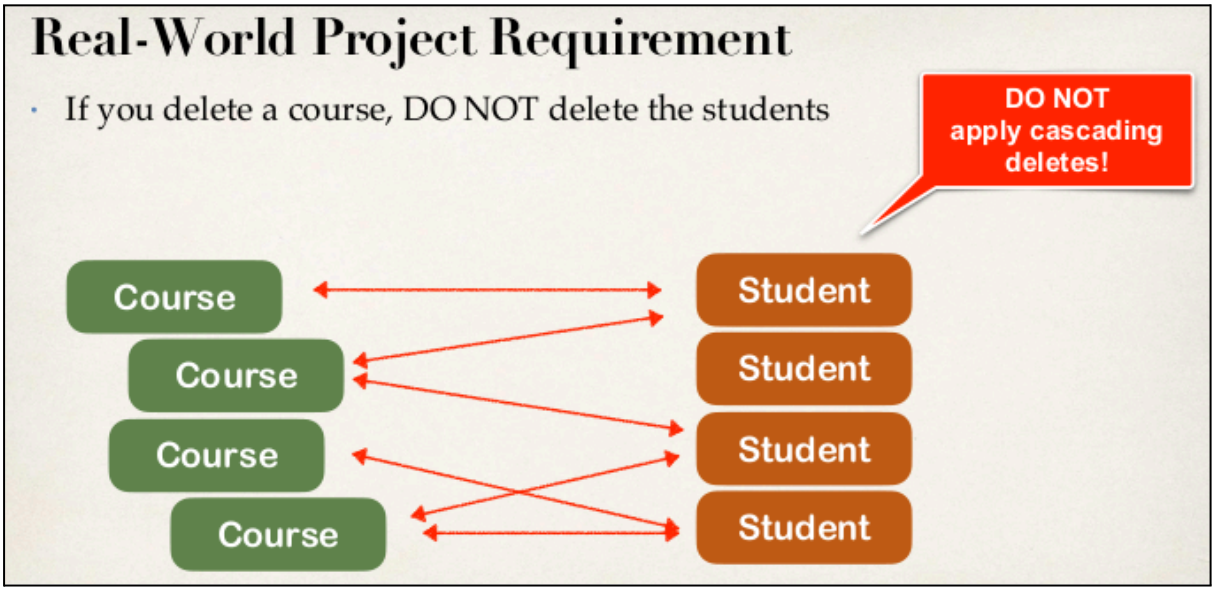


More on "inverse"

- In this context, we are defining the relationship in the **Course** class
- The **Student** class is on the "other side" … so it is considered the "inverse"
- "Inverse" refers to the "other side" of the relationship







Other features

- In the next set of videos, we'll add support for other features
- Lazy Loading of students and courses
- Cascading to handle cascading saves … but NOT deletes
- If we delete a course, DO NOT delete students
- If we delete a student, DO NOT delete courses

Aspect-Oriented Programming (AOP) Overview

"AOP" is a programming paradigm that aims to increase modularity by allowing separation of cross-cutting concerns. In AOP, cross-cutting concerns are aspects of a program that affect multiple modules, and AOP provides mechanisms to modularize these concerns separately from the main business logic.

1. Logging : Implement logging functionality across the application to record important events or errors. AOP can be used to inject logging code without cluttering the main business logic.

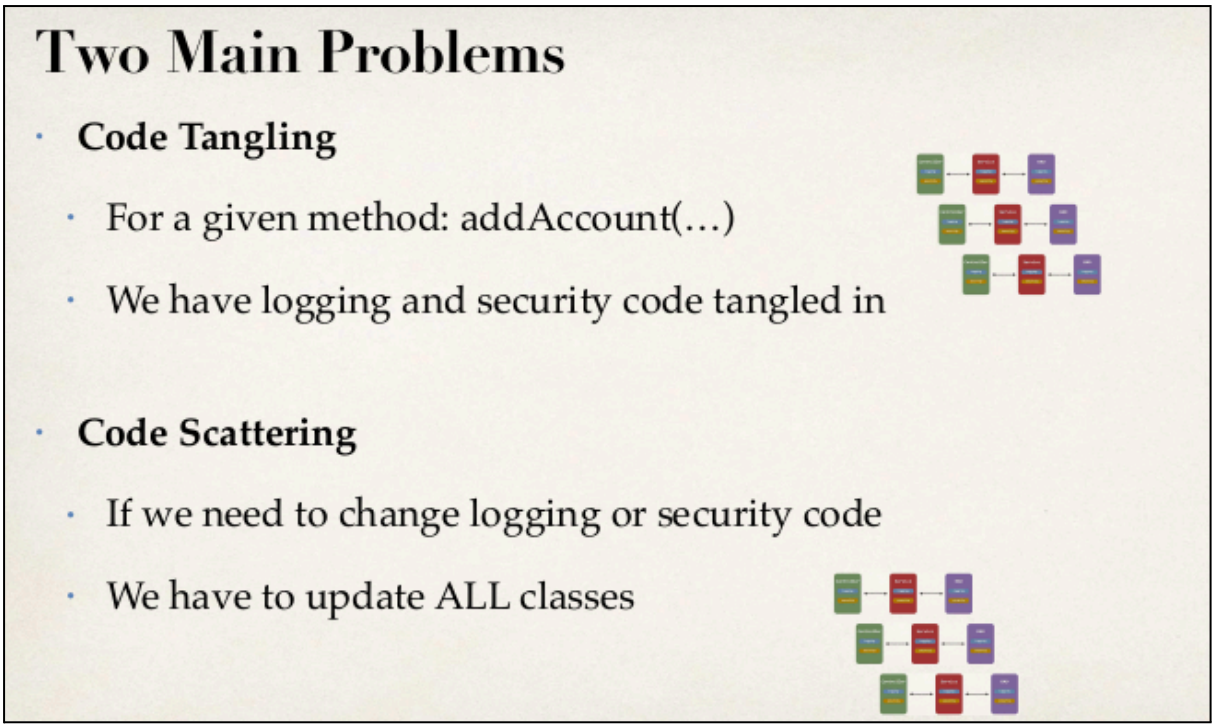
2. Security : Enforce security measures such as authentication and authorization. AOP can help intercept method calls to enforce security checks consistently across the application.

3. Transaction Management : Ensure consistency and reliability in database operations. AOP can automate transaction management by intercepting method calls and handling transactions transparently.

4. Caching : Improve performance by caching frequently accessed data. AOP can intercept method calls to cache data and retrieve it from the cache when needed, reducing database load.

5. Error Handling : Centralized error handling logic to handle exceptions gracefully. AOP can intercept method calls to catch exceptions and handle them uniformly across the application.

By leveraging Aspect-Oriented Programming in the UniCloud360 project, you can enhance modularity, improve code maintainability, and address cross-cutting concerns more effectively.



Other possible solutions?

Inheritance?

- Every class would need to inherit from a base class
- Can all classes extend from your base class? … plus no multiple inheritance

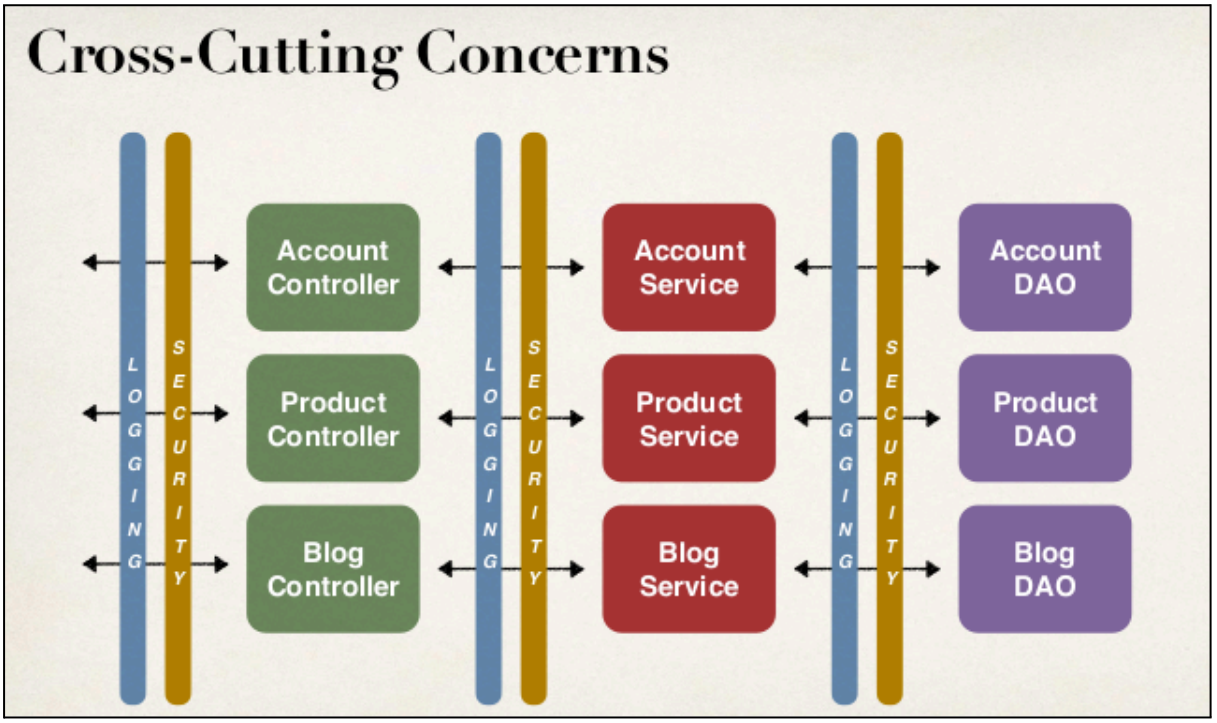
Delegation?

- Classes would delegate logging, security calls
- Still would need to update classes if we wanted to
- add/remove logging or security
- add new feature like auditing, API management, instrumentation

Aspect-Oriented Programming

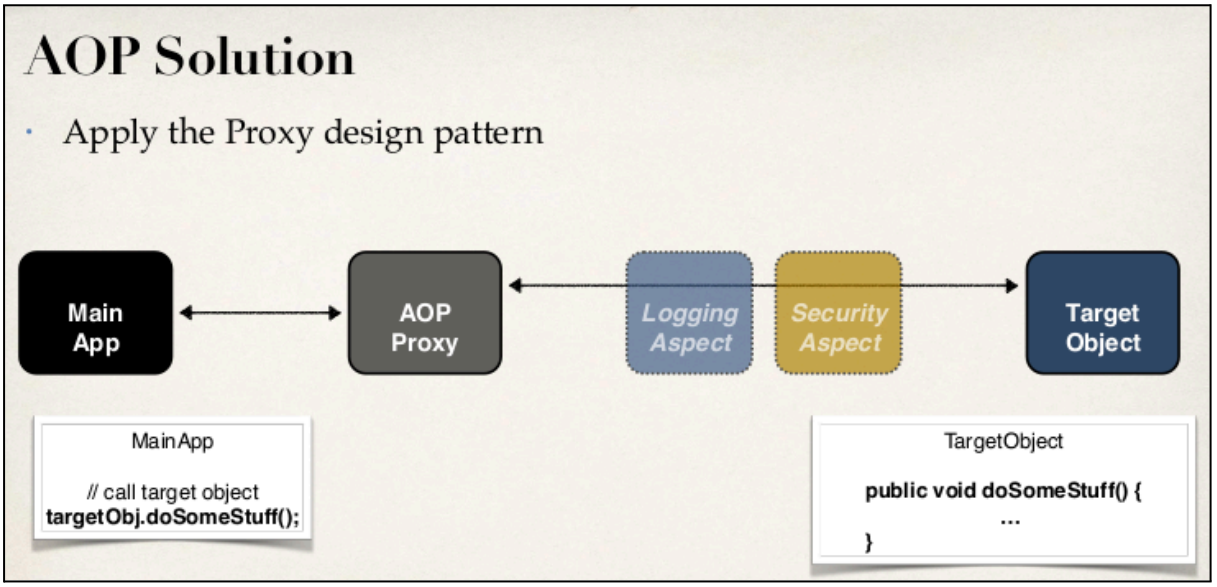
- Programming technique based on concept of an Aspect
- Aspect encapsulates cross-cutting logic

- "Concern" means logic / functionality



Aspects

- Aspect can be reused at multiple locations
- Same aspect/class … applied based on configuration

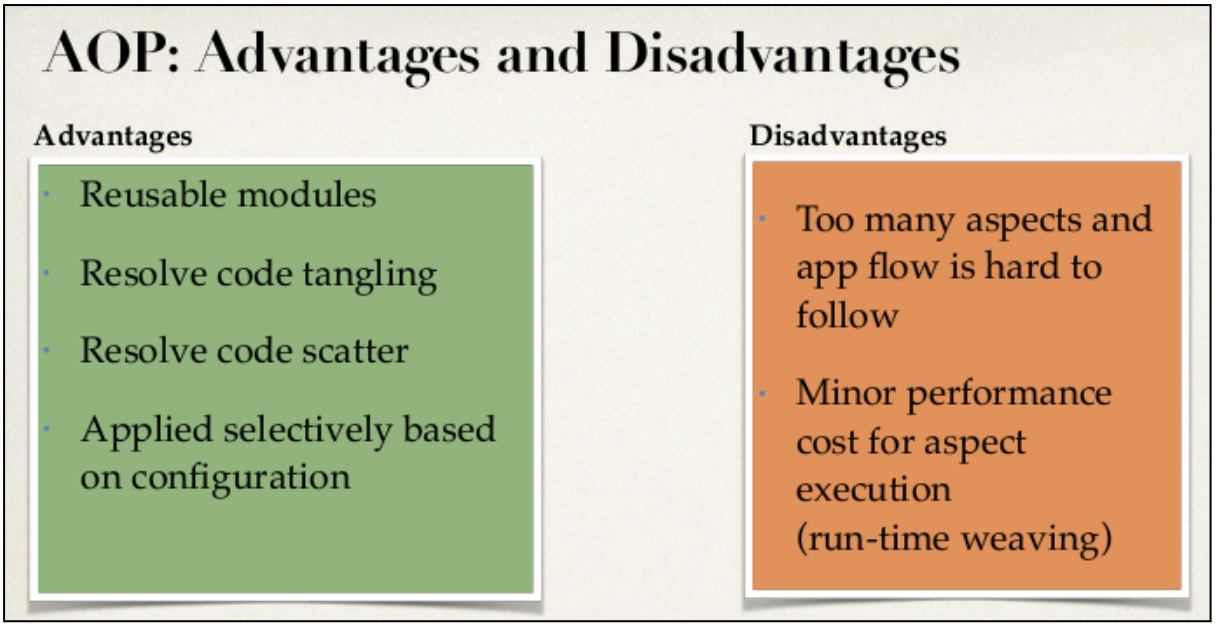


Benefits of AOP

- Code for Aspect is defined in a single class
    - Much better than being scattered everywhere
    - Promotes code reuse and easier to change
- Business code in your application is cleaner
    - Only applies to business functionality: addAccount
    - Reduces code complexity
- Con gurable
    - Based on configuration, apply Aspects selectively to different parts of app
    - No need to make changes to the main application code … very important!

Additional AOP Use Cases

- Most common
    - Audit logging
- who, what, when, where
    - Exception handling
- logging, security, transactions
    - log exception and notify DevOps team via SMS/email
- API Management
    - how many times has a method been called user
    - analytics: what are peak times? What is the average load? Who is the top user?



Aspect-Oriented Programming (AOP) Spring AOP Support

AOP Terminology

- Aspect: module of code for a cross-cutting concern (logging, security, …)
- Advice: What action is taken and when it should be applied
- Join Point: When to apply code during program execution
- Pointcut: A predicate expression for where advice should be applied

Advice Types

- Before advice: run before the method
- After finally advice: run after the method (finally)
- After returning advice: run after the method (success execution)
- After throwing advice: run after method (if exception thrown)
- Around advice: run before and after method

Weaving

- Connecting aspects to target objects to create an advised object
- Different types of weaving
    - Compile-time, load-time or run-time
- Regarding performance: run-time weaving is the slowest

AOP Frameworks

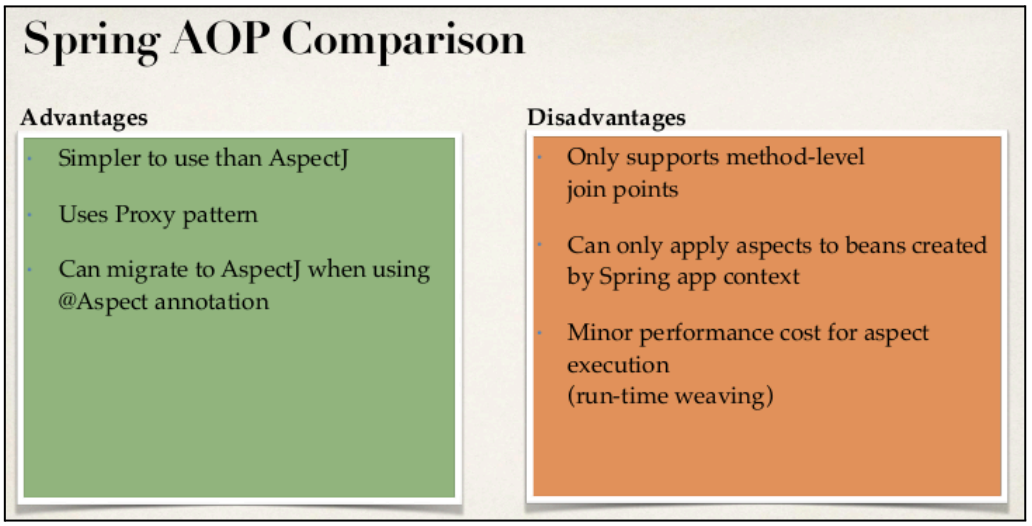
- Two leading AOP Frameworks for Java

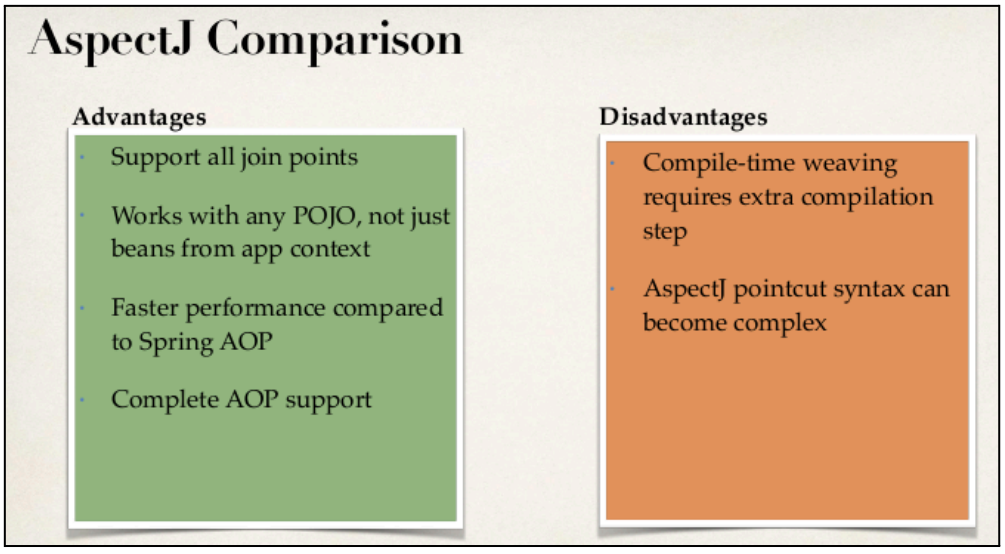
Spring AOP Support

- Spring provides AOP support
- Key component of Spring
- Security, transactions, caching etc
- Uses run-time weaving of aspects

AspectJ

- Original AOP framework, released in 2001
    - www.eclipse.org/aspectj
- Provides complete support for AOP
- Rich support for
    - join points: method-level, constructor, eld
    - code weaving: compile-time, post compile-time and load-time





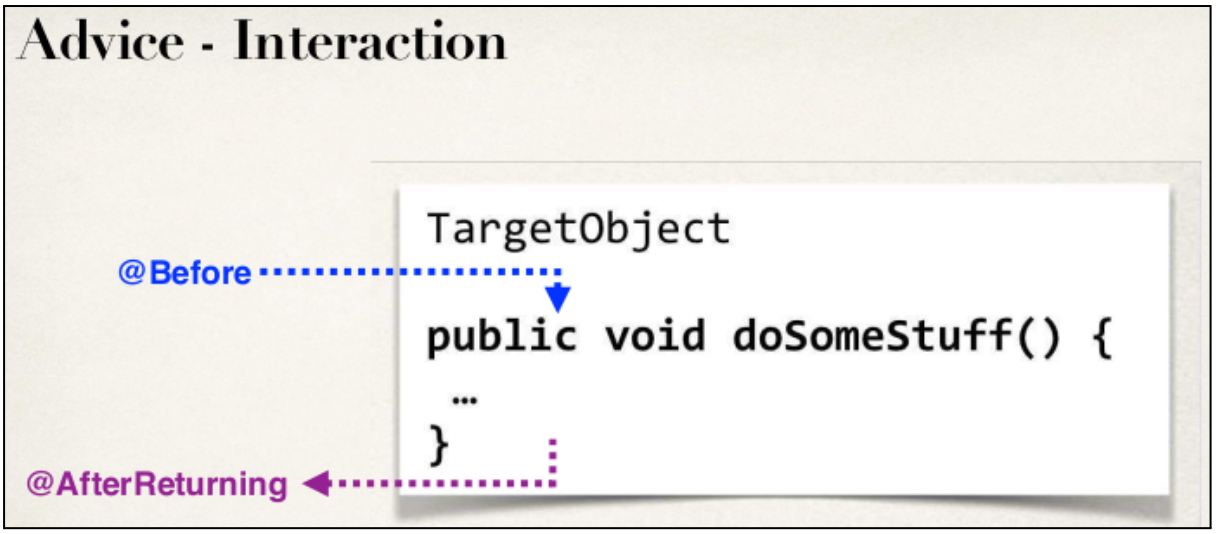
Comparing Spring AOP and AspectJ

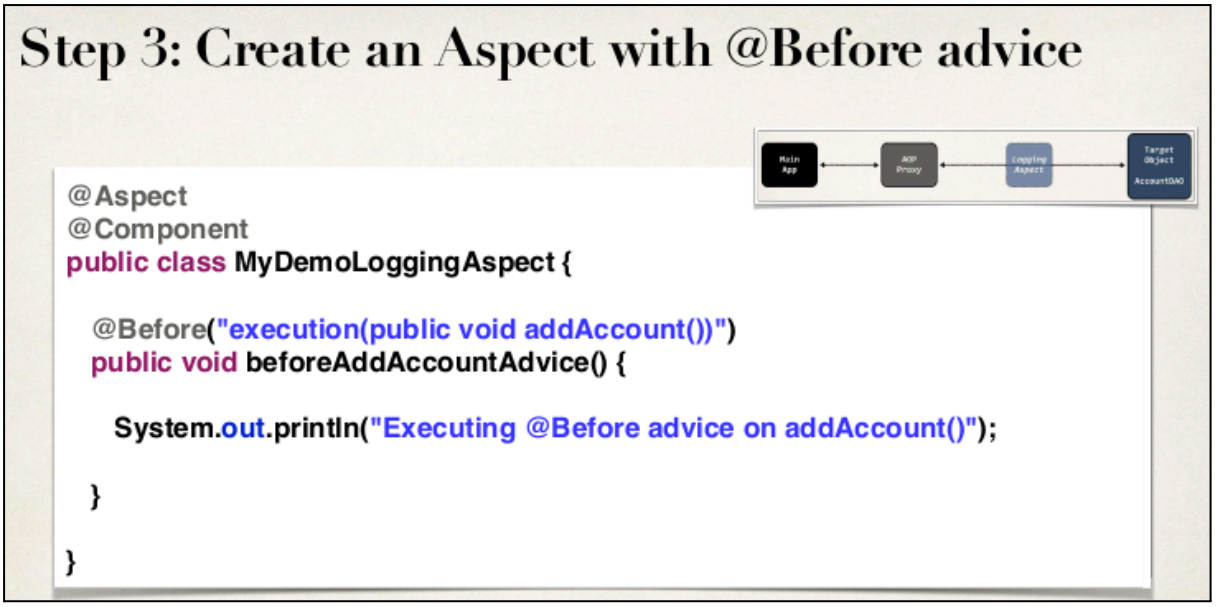
- Spring AOP only supports
  - Method-level join points
  - Run-time code weaving (slower than AspectJ)
- AspectJ supports
  - join points: method-level, constructor, eld
  - weaving: compile-time, post compile-time and load-time

Comparing Spring AOP and AspectJ

- Spring AOP is a light implementation of AOP
- Solves common problems in enterprise applications
- My recommendation
  - Start with Spring AOP … easy to get started with
  - If you have complex requirements then move to AspectJ

Aspect-Oriented Programming (AOP) @Before Advice





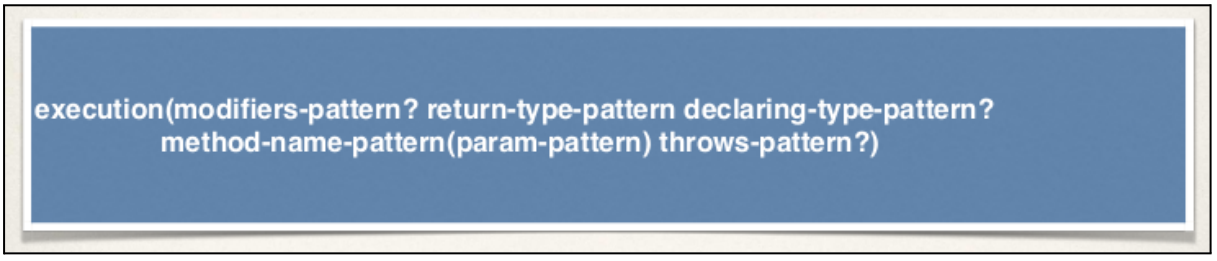
AOP - Pointcut Expressions

AOP Terminology
Pointcut: A predicate expression for where advice should be applied

Pointcut Expression Language

- Spring AOP using AspectJ's pointcut expression language
- We will start with execution pointcuts
- Applies to execution of methods

Pointcut Expression Language

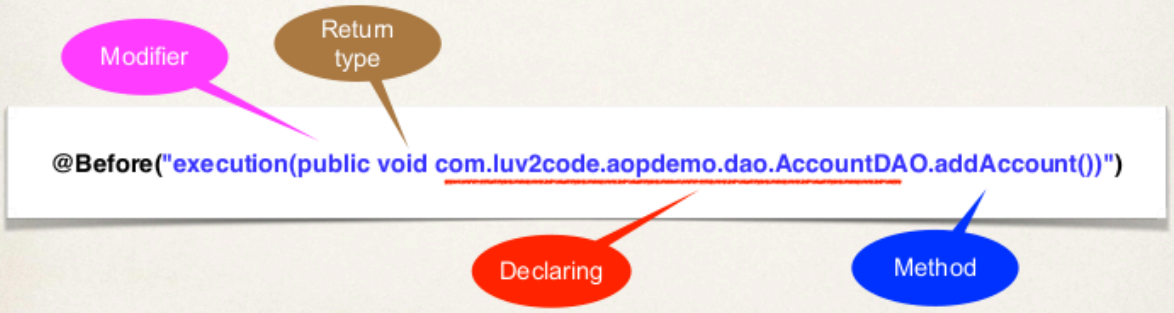


The pattern is optional if it has "?"

Pointcut Expression Examples

Match on method names

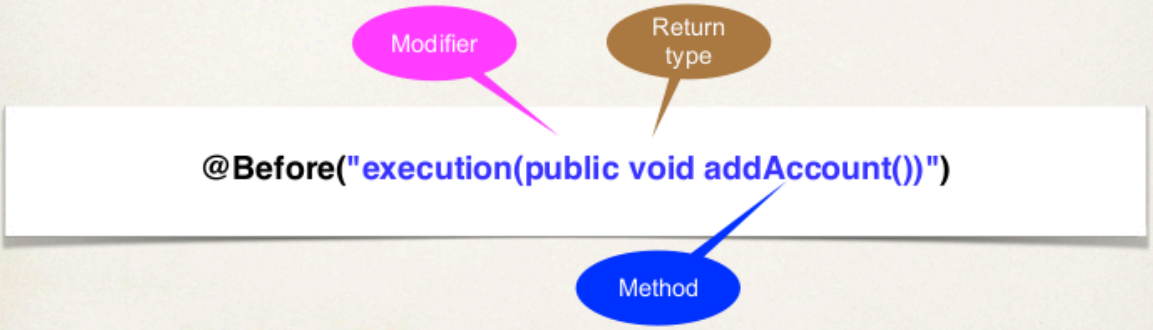
- Match only **addAccount()** method in **AccountDAO** class



Pointcut Expression Examples

Match on method names

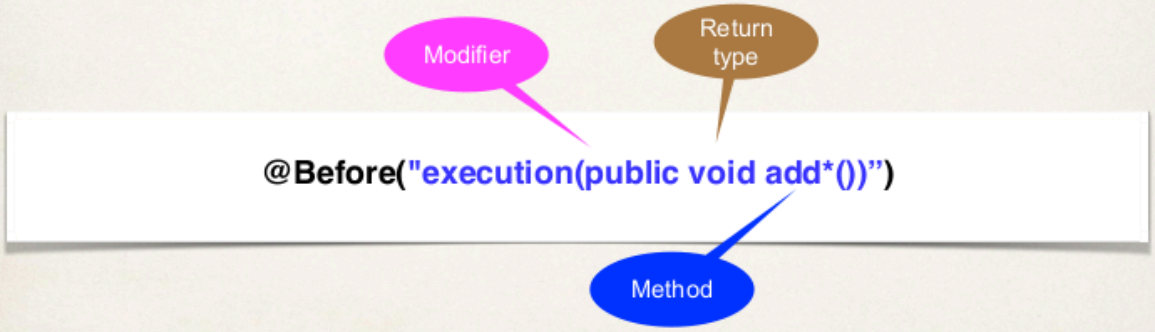
- Match any **addAccount()** method in **any** class



Pointcut Expression Examples

Match on method names (using wildcards)

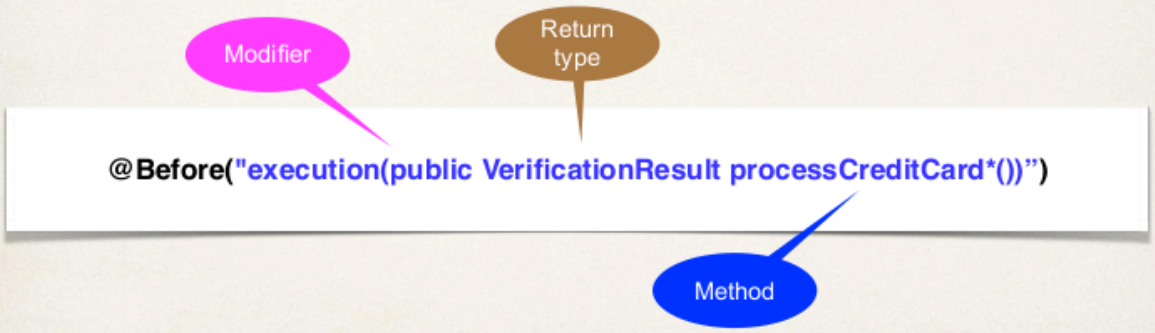
- Match methods **starting** with **add** in any class



Pointcut Expression Examples

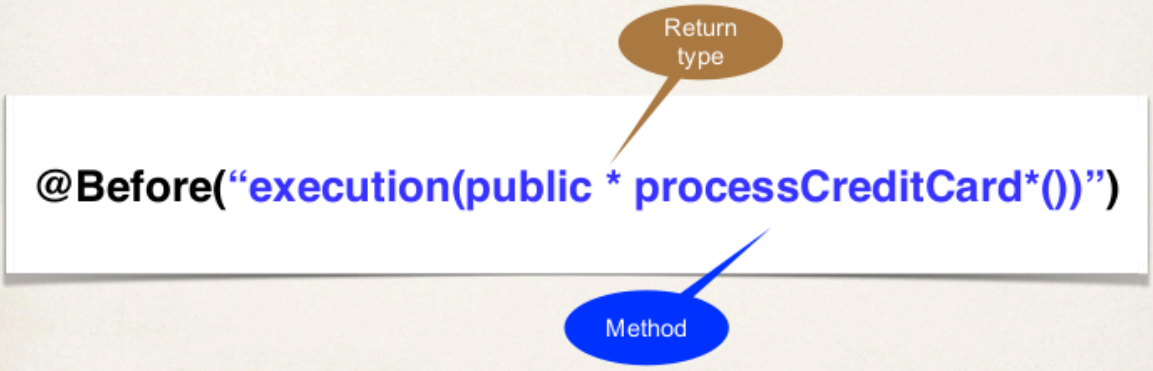
Match on method names (using wildcards)

- Match methods starting with **processCreditCard** in any class



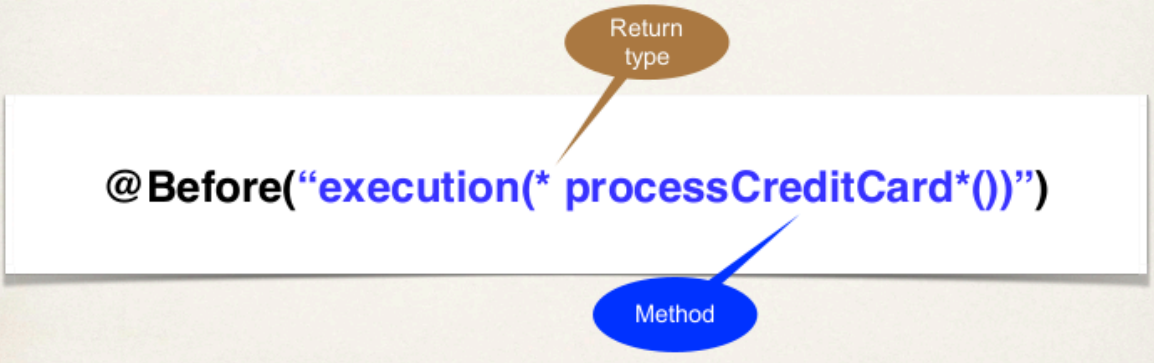
Pointcut Expression Examples

- Use wildcards on return type



Pointcut Expression Examples

- Modifier is optional … so you don't have to list it

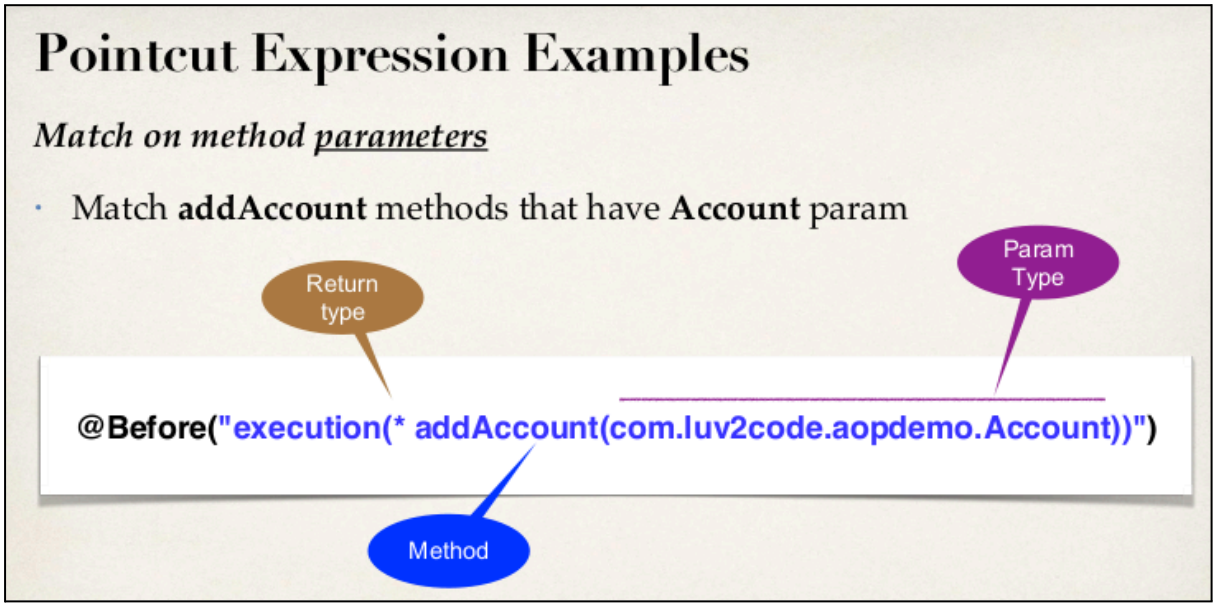


AOP - Pointcut Expressions (cont)

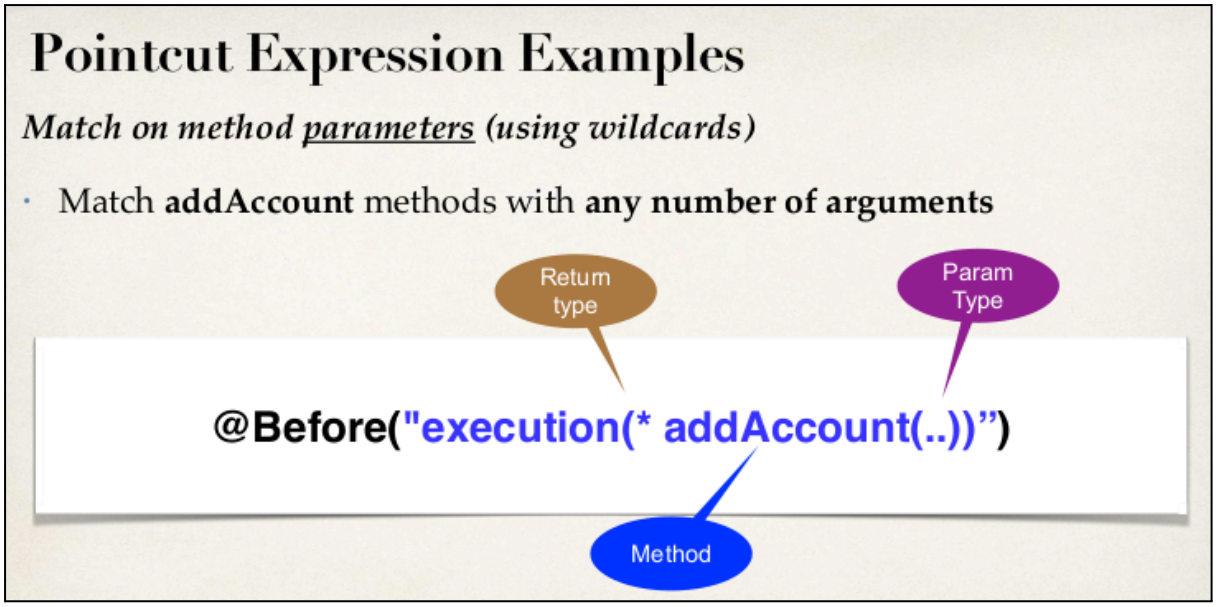
Parameter Pattern Wildcards

For param-pattern

- () - matches a method with no arguments
- (*) - matches a method with one argument of any type
- (..) - matches a method with 0 or more arguments of any type



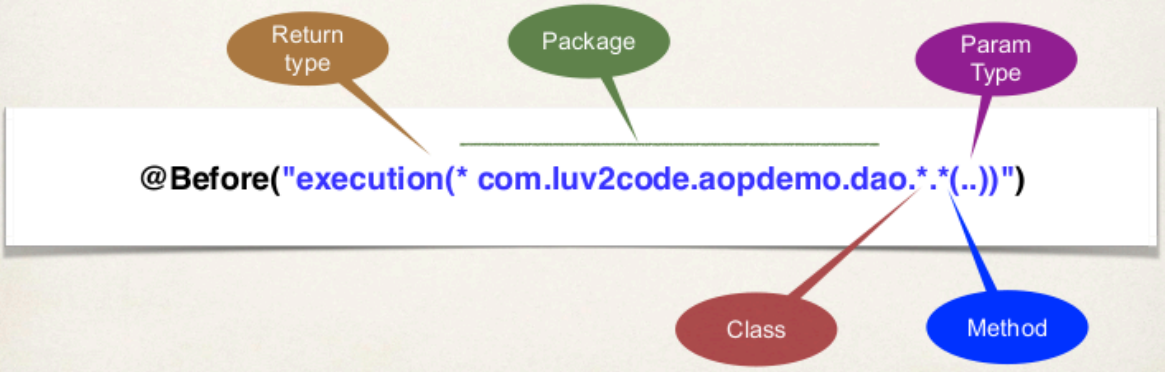
Param - should be fully qualified class name



Package - Pointcut Expression Examples

Match on methods in a package

- Match any method in our DAO package: **com.luv2code.aopdemo.dao**



When using wildcards with AOP, caution should be taken.

If new frameworks are added to your project, then you may encounter conflicts.

Recommendation is to:
- narrow your pointcut expressions
- limit them to your project package

In this case, our pointcut expression is too broad.

We can resolve this by:
- narrowing the pointcut expression
- only match within our project package

```
@Before("execution(* com.luv2code..add*(..))")
```

Narrow pointcut expression to our package

Aspect-Oriented Programming (AOP) Pointcut Declarations

Problem

- How can we reuse a pointcut expression?
    - Want to apply to multiple advices

```
@Before("execution(* com.luv2code.aopdemo.dao.*.*(..))")
public void beforeAddAccountAdvice() {
    ...
}
```

Ideal Solution

- Create a pointcut declaration once
- Apply it to multiple advices

Step 2 - Apply to Multiple Advices

```
@Aspect
@Component
public class MyDemoLoggingAspect {

    @Pointcut("execution(* com.luv2code.aopdemo.dao.*.*(..))")
    private void forDaoPackage() {}

    @Before("forDaoPackage()")
    public void beforeAddAccountAdvice() {
        ...
    }

    @Before("forDaoPackage()")
    public void performApiAnalytics() {
        ...
    }

}
```

Benefits of Pointcut Declarations

- Easily reuse pointcut expressions
- Update pointcut in one location
- Can also share and combine pointcut expressions (coming up later)

Aspect-Oriented Programming (AOP) Combine Pointcuts

Problem

- How to apply multiple pointcut expressions to single advice?
- Execute an advice only if certain conditions are met
- For example
    - All methods in a package EXCEPT getter/setter methods

Combining Pointcut Expressions
Combine pointcut expressions using logic operators

- AND (&&)
- OR (||)
- NOT (!)

Combining Pointcut Expressions

- Works like an "if" statement
- Execution happens only if it evaluates to true

```
@Before("expressionOne() && expressionTwo()")

@Before("expressionOne() || expressionTwo()")

@Before("expressionOne() && !expressionTwo()")
```

Step 1- Create Pointcut Declaration

```
@Pointcut("execution(* com.luv2code.aopdemo.dao.*.*(..))")
private void forDaoPackage() {}

// create pointcut for getter methods
@Pointcut("execution(* com.luv2code.aopdemo.dao.*.get*(..))")
private void getter() {}

// create pointcut for setter methods
@Pointcut("execution(* com.luv2code.aopdemo.dao.*.set*(..))")
private void setter() {}
```

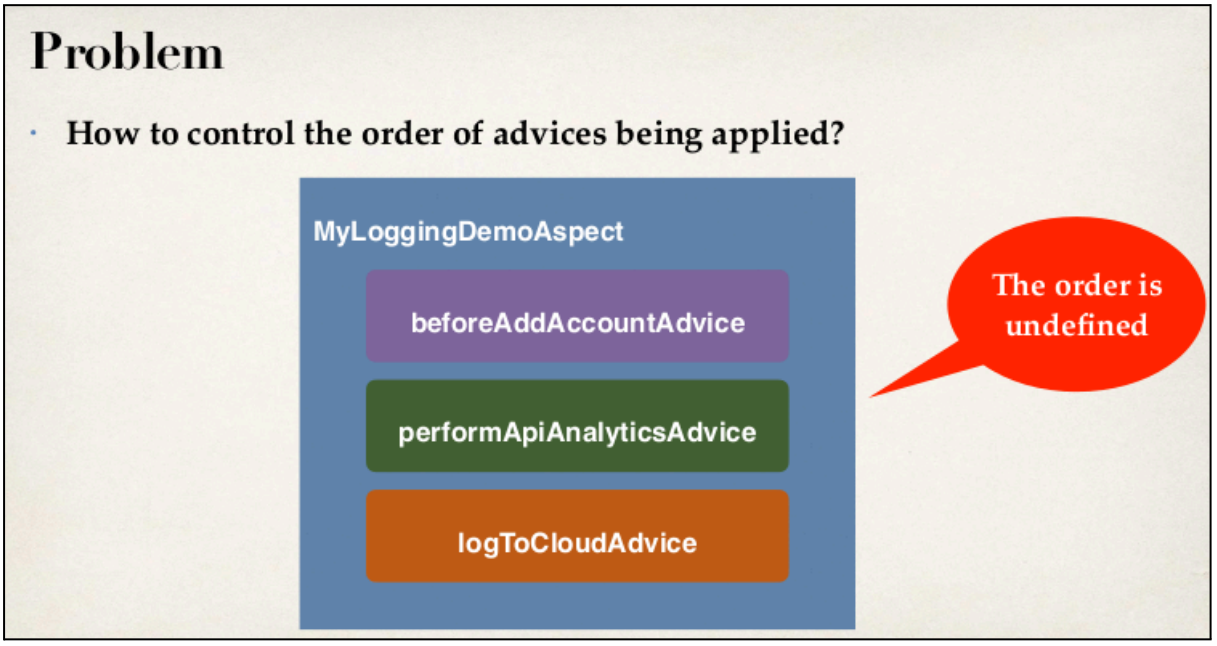
Step 2 - Combine Pointcut Declarations

```
@Pointcut("execution(* com.luv2code.aopdemo.dao.*.*(..))")
private void forDaoPackage() {}

// create pointcut for getter methods
@Pointcut("execution(* com.luv2code.aopdemo.dao.*.get*(..))")
private void getter() {}

// create pointcut for setter methods
@Pointcut("execution(* com.luv2code.aopdemo.dao.*.set*(..))")
private void setter() {}

// combine pointcut: include package ... exclude getter/setter
@Pointcut("forDaoPackage() && !(getter() || setter())")
private void forDaoPackageNoGetterSetter() {}
```

Step 3 - Apply Pointcut Declaration to Advice(s)

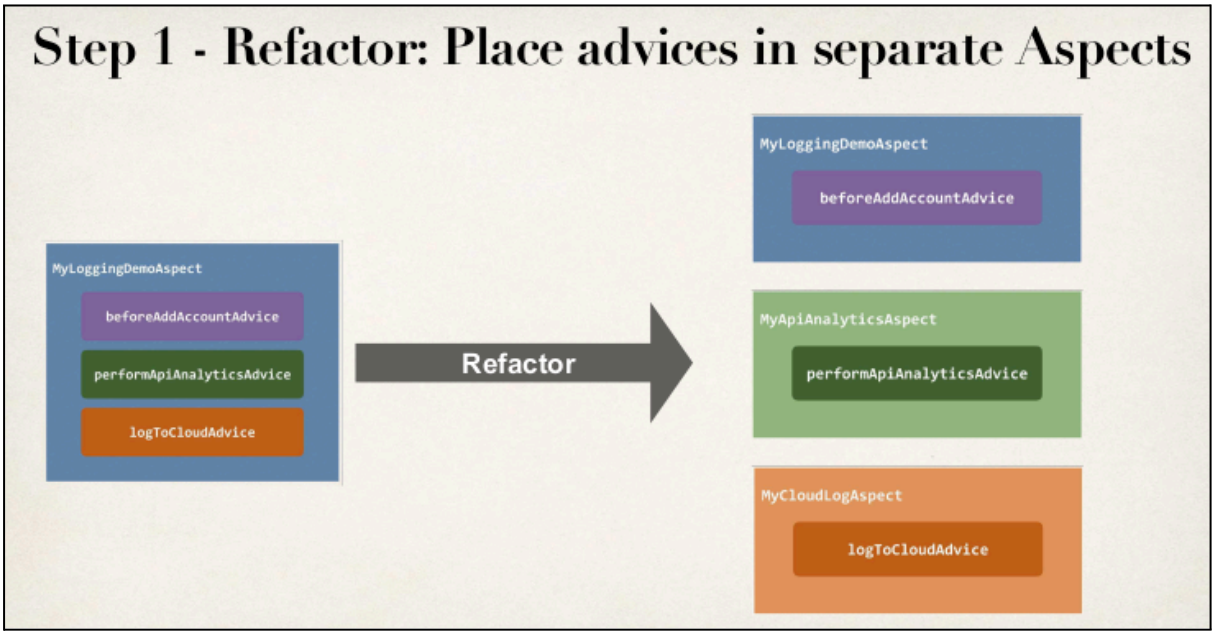
```
...

// combine pointcut: include package ... exclude getter/setter
@Pointcut("forDaoPackage() && !(getter() || setter())")
private void forDaoPackageNoGetterSetter() {}

@Before("forDaoPackageNoGetterSetter()")
public void beforeAddAccountAdvice() {
  ...
}
```

Aspect-Oriented Programming (AOP) Control Aspect Order



To Control Order

- Refactor: Place advices in separate Aspects
- Control order on Aspects using the @Order annotation
- Guarantees order of when Aspects are applied



Step 2 - Add @Order annotation

- Control order on Aspects using the @Order annotation

```
@Order(1)
public class MyCloudLogAspect {
  ...
}
```

- Guarantees order of when Aspects are applied
- Lower numbers have higher precedence

@Order annotation

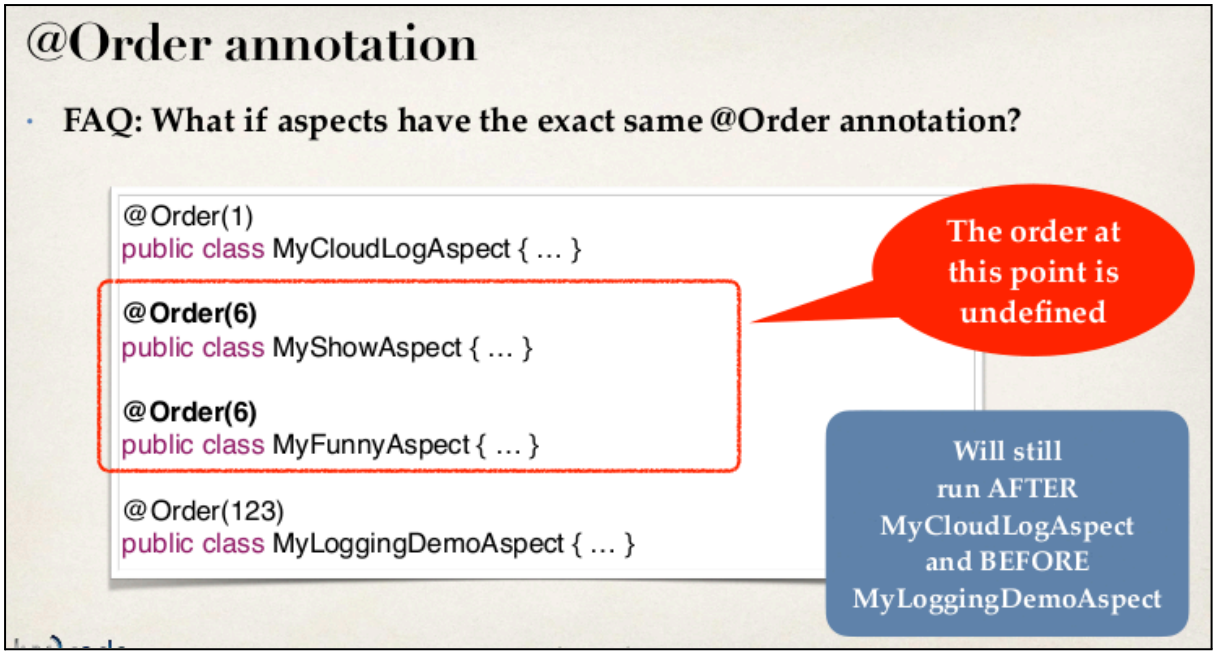

```
@Aspect
@Component
@Order(1)
public class MyCloudLogAspect {
 ...
}
```

```
@Aspect
@Component
@Order(2)
public class MyLoggingDemoAspect {
 ...
}
```

```
@Aspect
@Component
@Order(3)
public class MyApiAnalyticsAspect {
 ...
}
```

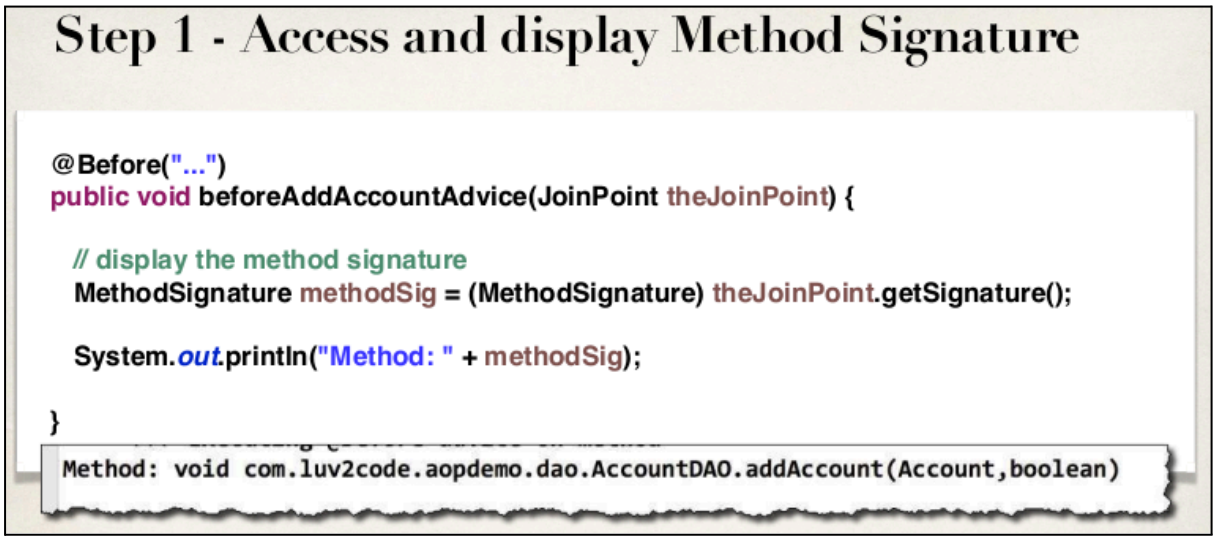
@Order annotation

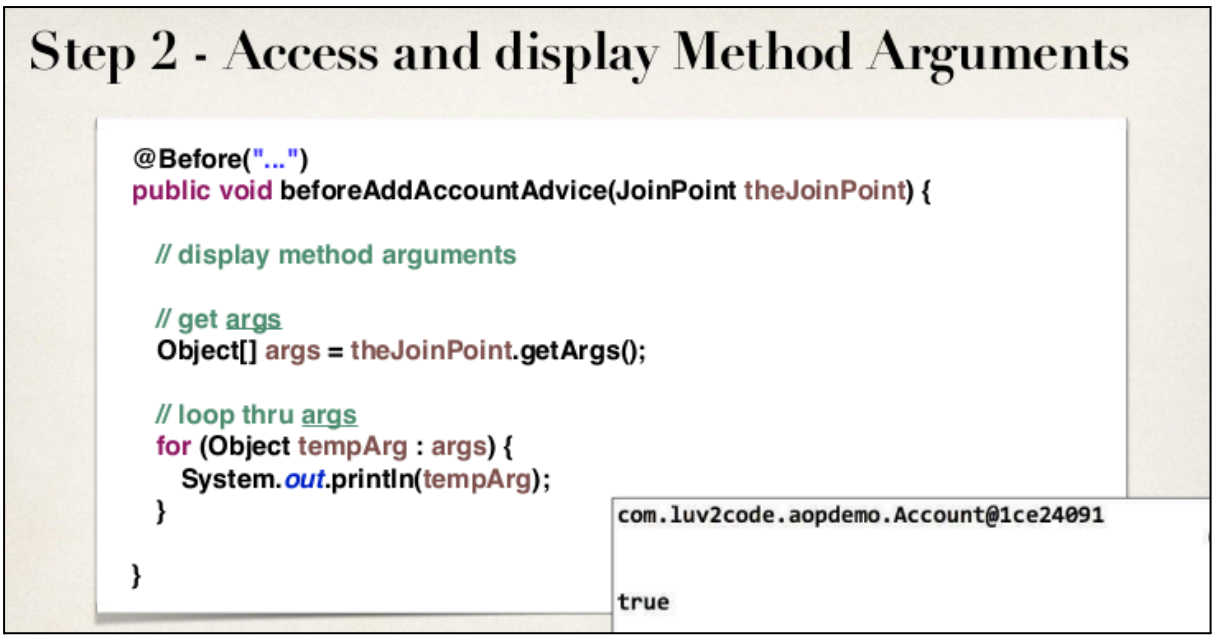
- Lower numbers have higher precedence
  - Range: Integer.MIN_VALUE to Integer.MAX_VALUE
  - Negative numbers are allowed
  - Does not have to be consecutive



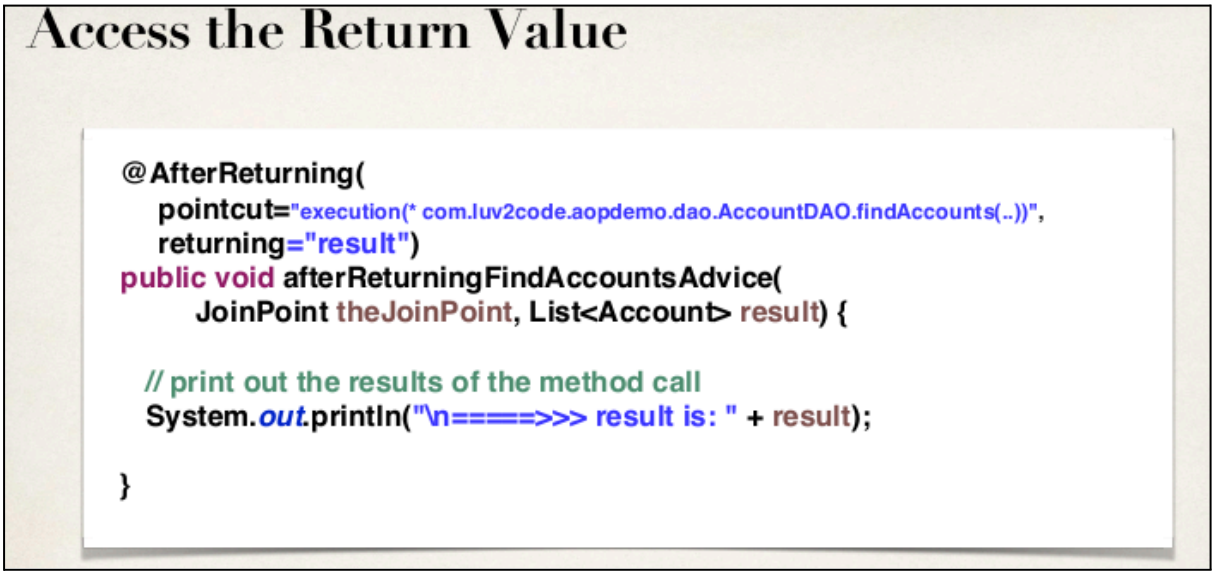
Aspect-Oriented Programming (AOP) Reading Method Arguments with JoinPoints

Problem
When we are in an aspect (ie for logging)
How can we access method parameters?





Aspect-Oriented Programming (AOP) @AfterReturning Advice



Aspect-Oriented Programming (AOP) @AfterReturning Advice - Modify Return Value

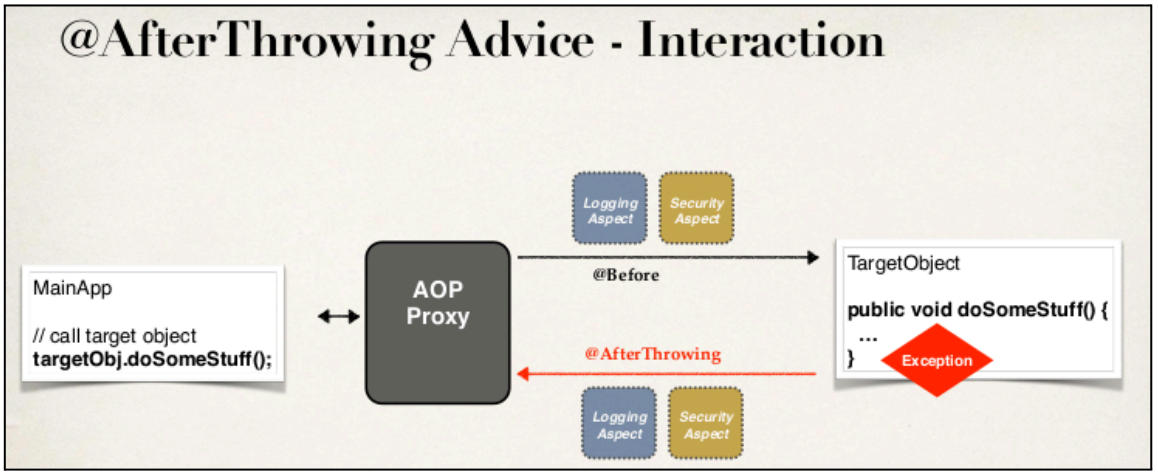
@AfterReturning Advice - Use Cases

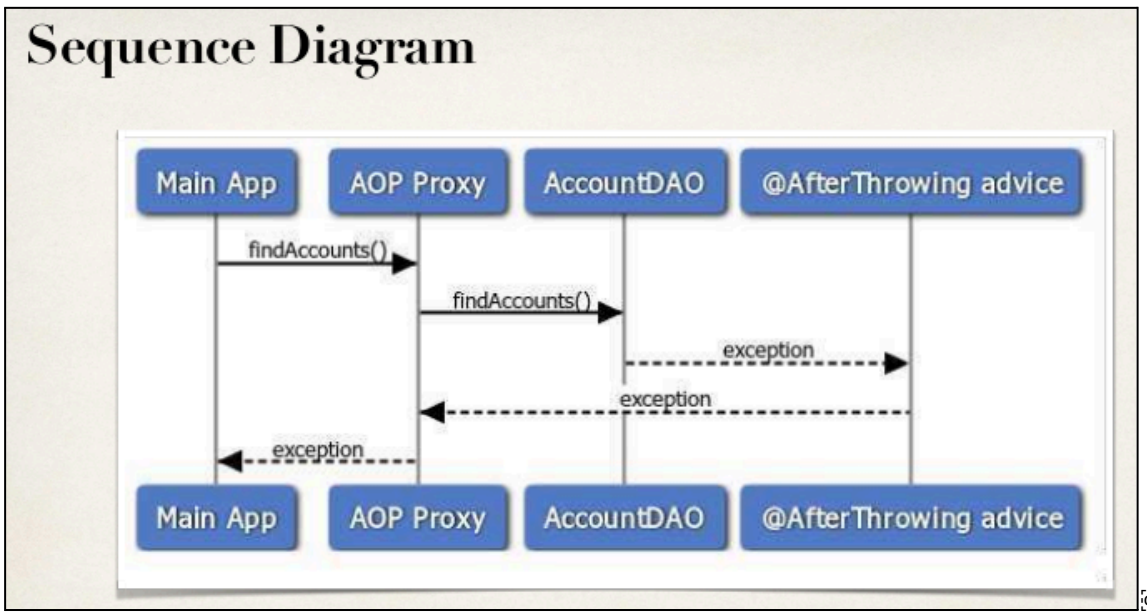
- Most common
  - Audit logging
- logging, security, transactions
  - who, what, when, where
- Post-processing Data
- Post process the data before returning to caller
- Format the data or enrich the data (really cool but be careful)

Modify the Return Value

```
@AfterReturning(
    pointcut="execution(* com.luv2code.aopdemo.dao.AccountDAO.findAccounts(..))",
    returning="result")
public void afterReturningFindAccountsAdvice(
        JoinPoint theJoinPoint, List<Account> result) {

    // modify "result" list: add, remove, update, etc ...
    if (!result.isEmpty()) {

        Account tempAccount = result.get(0);

        tempAccount.setName("Daffy Duck");
    }
}
```

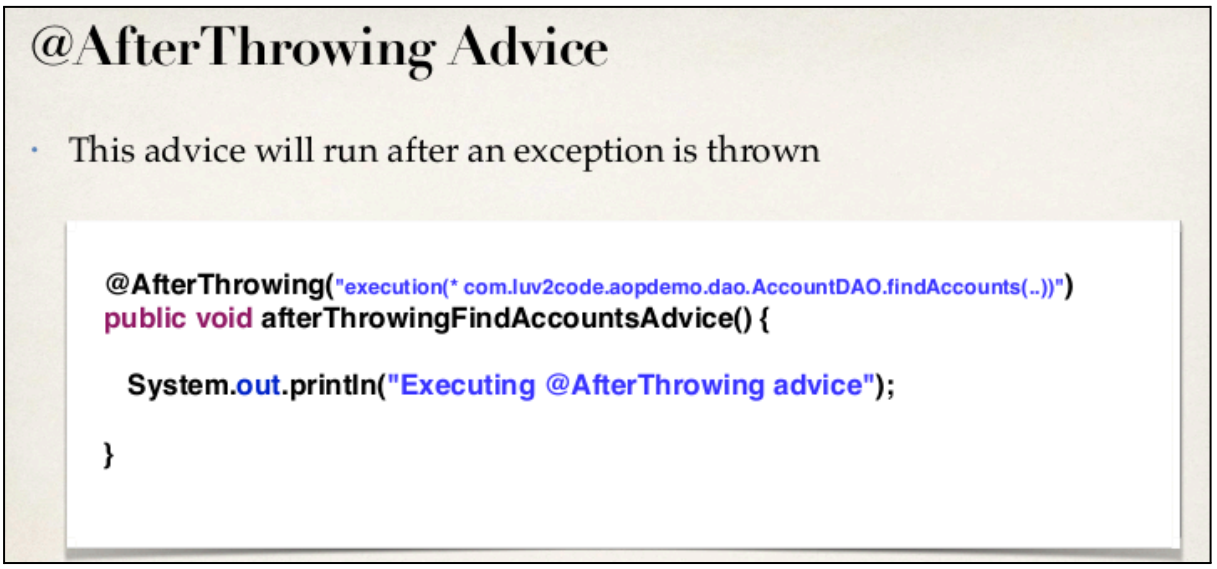
Aspect-Oriented Programming (AOP) @AfterThrowing Advice

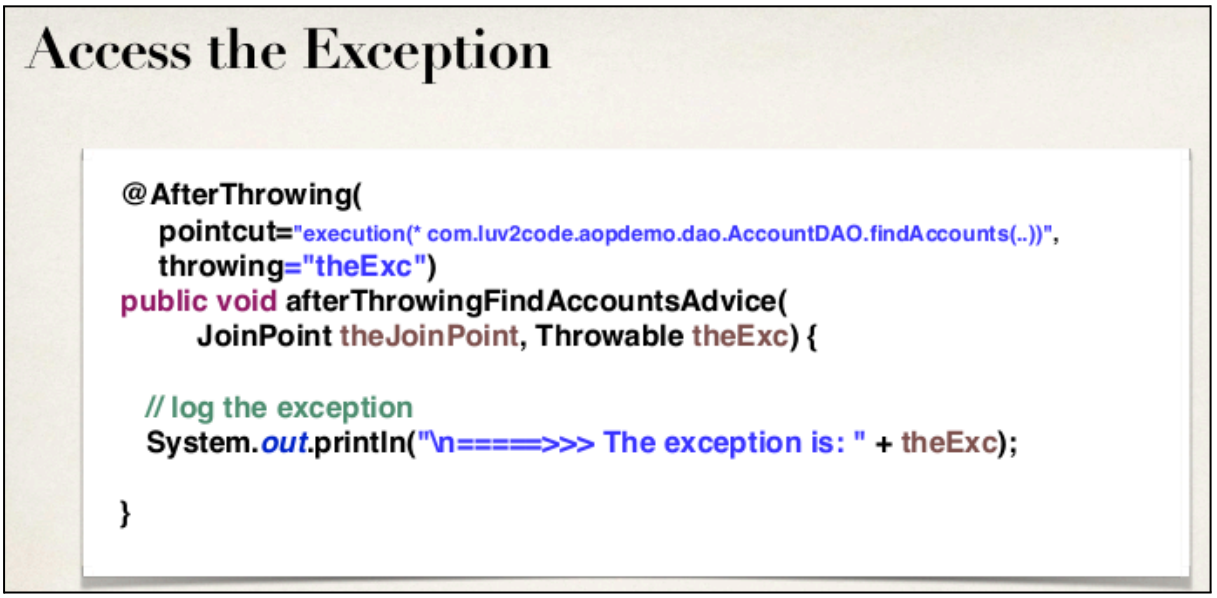




@AfterThrowing Advice - Use Cases

- Log the exception
- Perform auditing on the exception
- Notify DevOps team via email or SMS
- Encapsulate this functionality in AOP aspect for easy reuse



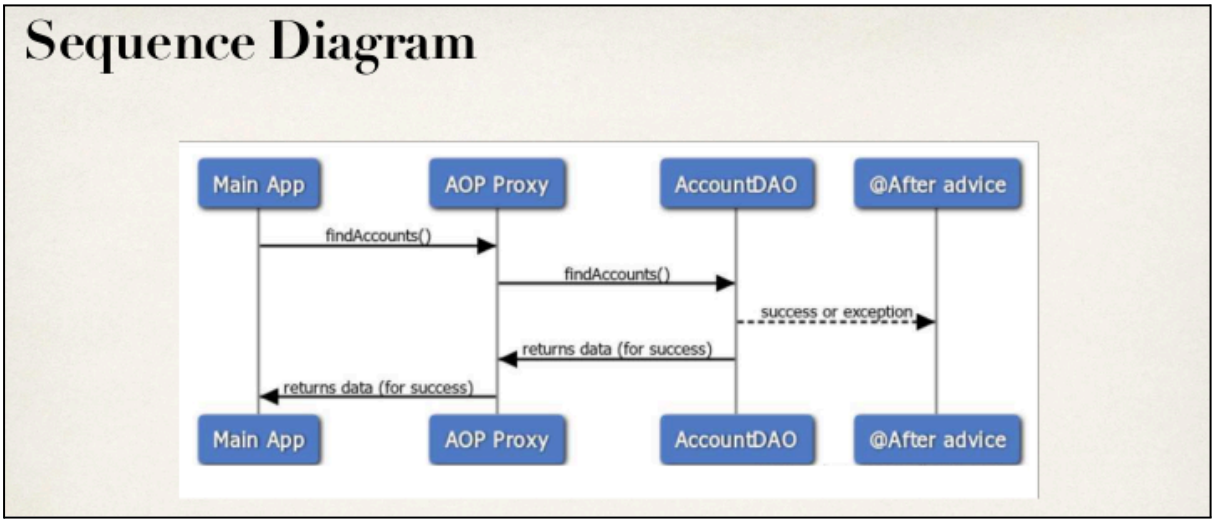


Exception Propagation
If you want to stop the exception propagation
then use the @Around advice

Development Process - @AfterThrowing

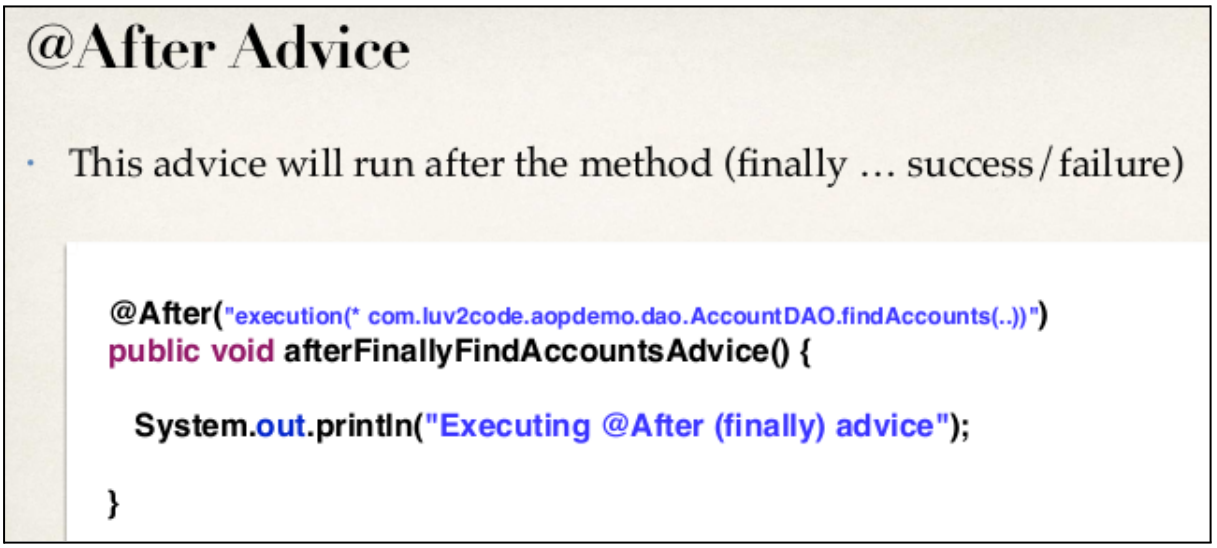
1. In Main App, add a try/catch block for exception handling
2. Modify AccountDAO to simulate throwing an exception
3. Add @AfterThrowing advice

Aspect-Oriented Programming (AOP) @After Advice



@After Advice - Use Cases

- Log the exception and/or perform auditing
- Code to run regardless of method outcome
- Encapsulate this functionality in AOP aspect for easy reuse



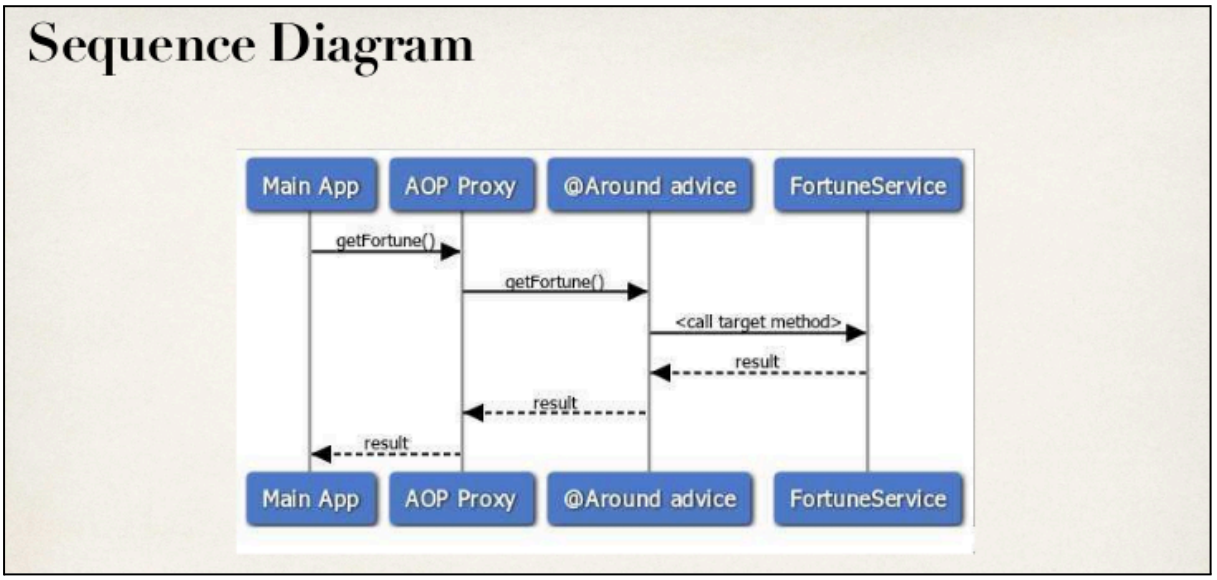
@After Advice - Tips

- The @After advice does not have access to the exception
    - If you need exception, then use @AfterThrowing advice
- The @After advice should be able to run in the case of success or error
    - Your code should not depend on happy path or an exception
    - Logging / auditing is the easiest case here

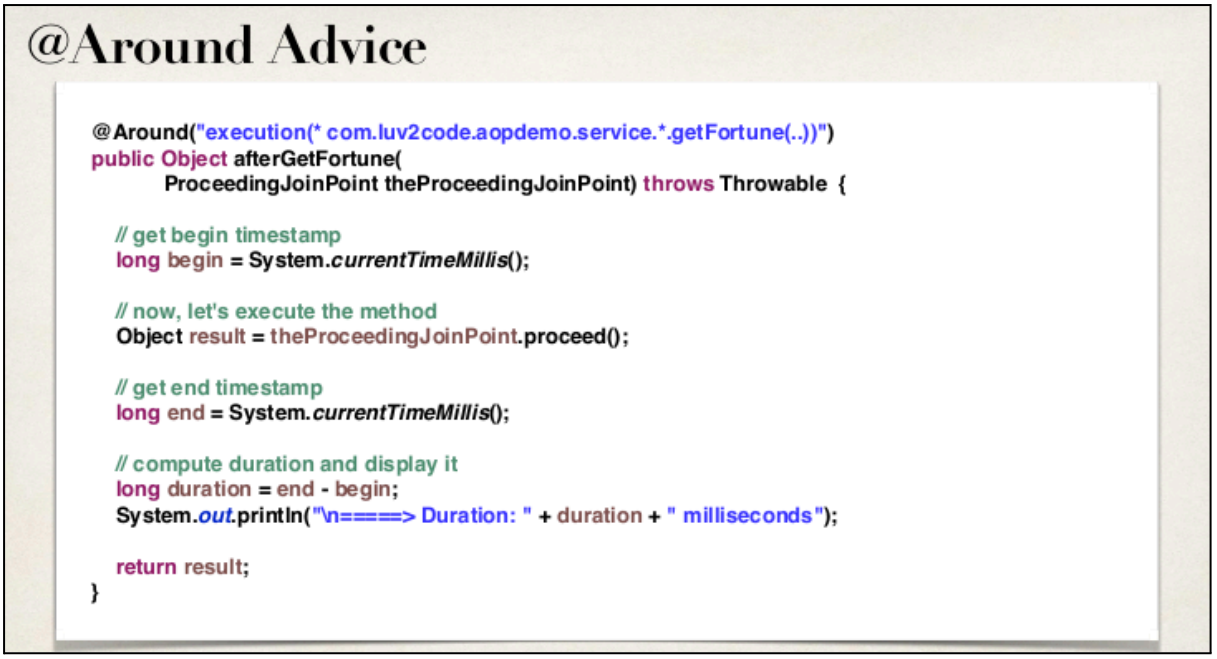
Development Process - @After
1. Prep work
2. Add @After advice
3. Test for failure/exception case
4. Test for success case

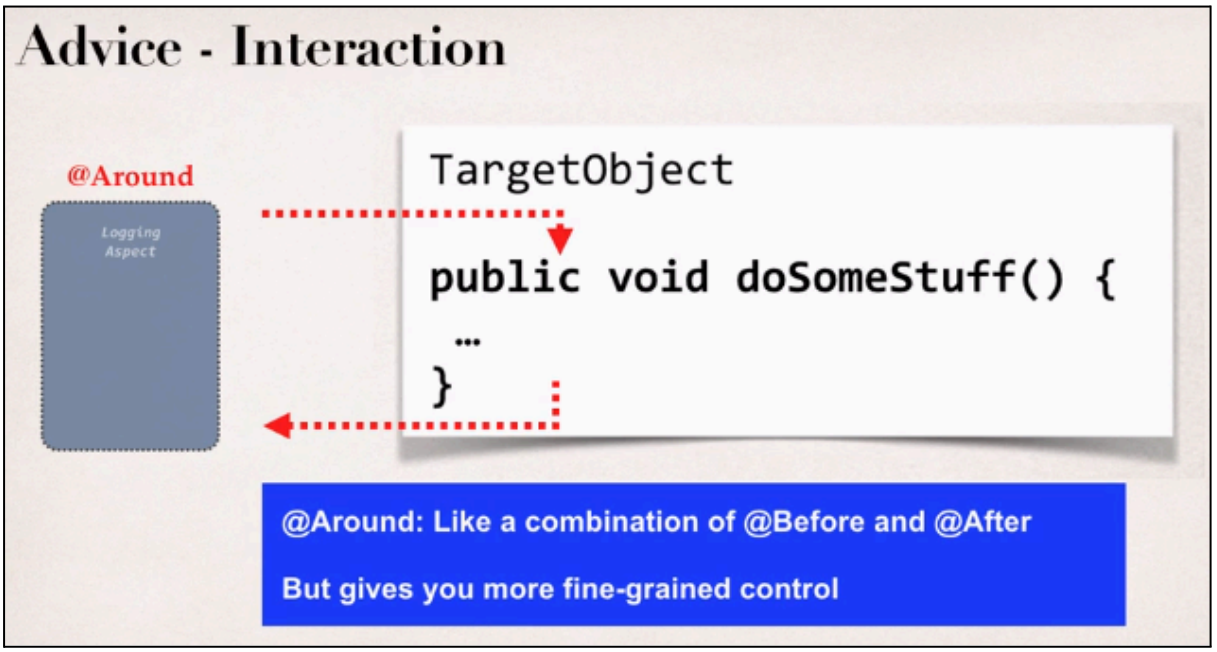
Aspect-Oriented Programming (AOP) @Around Advice

@Around Advice - Use Cases
- Most common: logging, auditing, security
- Pre-processing and post-processing data
- Instrumentation / pro ling code
    - How long does it take for a section of code to run?
- Managing exceptions
    - Swallow / handle / stop exceptions



ProceedingJoinPoint
- When using @Around advice
- You will get a reference to a “proceeding join point”
- This is a handle to the target method
- Your code can use the proceeding join point to execute target method



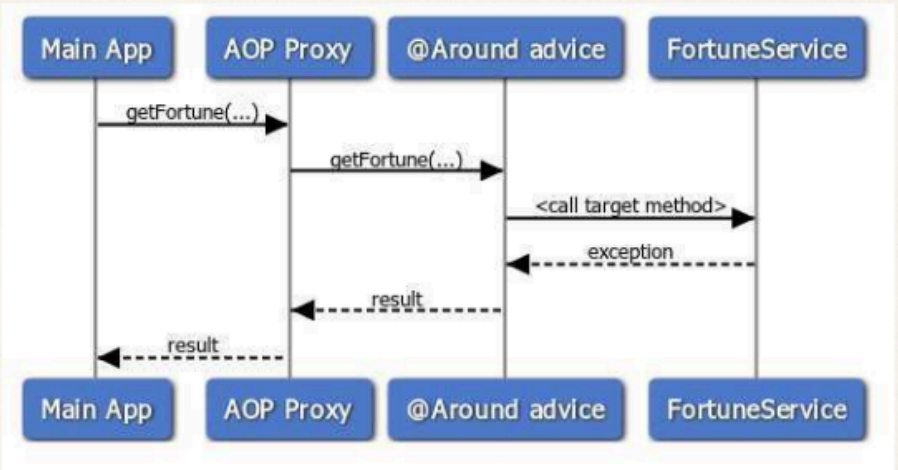


Development Process - @Around
1. Create Traf cFortuneService
2. Update main app to call Traf cFortuneService
3. Add @Around advice

Aspect-Oriented Programming (AOP) @Around Advice - Handle Exception

- ProceedingJoinPoint - Revisited
- For an exception thrown from proceeding join point
    - You can handle / swallow /stop the exception
    - Or you can simply rethrow the exception
- This gives you ne-grained control over how the target method is called

Sequence Diagram



Handle Exception

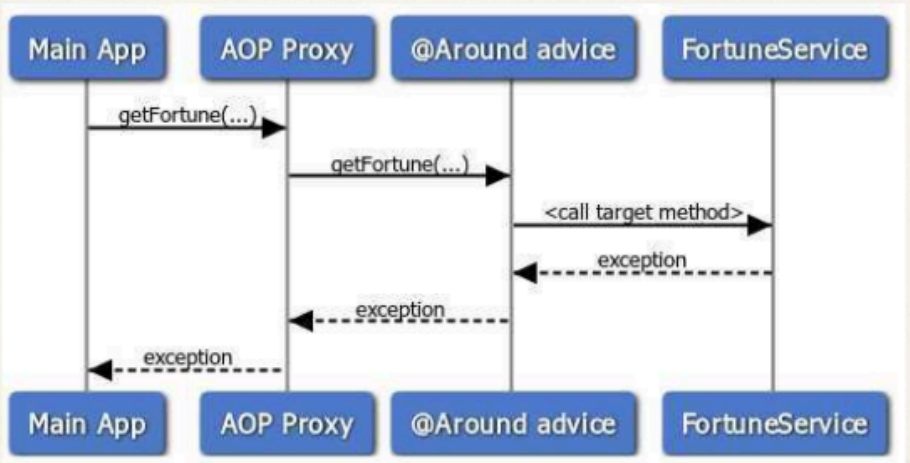
```
@Around("execution(* com.luv2code.aopdemo.service.*.getFortune(..))")
public Object afterGetFortune(
        ProceedingJoinPoint theProceedingJoinPoint) throws Throwable {

    Object result = null;

    try {
        // let's execute the method
        result = theProceedingJoinPoint.proceed();

    } catch (Exception exc) {
        // log exception
        System.out.println("@Around advice: We have a problem " + exc);

        // handle and give default fortune ... use this approach with caution
        result = "Nothing exciting here. Move along!";
    }

    return result;
}
```

Development Process - @Around
1. Add trip wire to simulate an exception
2. Modify @Around advice to handle exception

Aspect-Oriented Programming (AOP) @Around Advice - Rethrow Exception

- ProceedingJoinPoint - Revisited
- For an exception thrown from proceeding join point
    - You can handle / swallow /stop the exception
    - Or you can simply rethrow the exception

Sequence Diagram



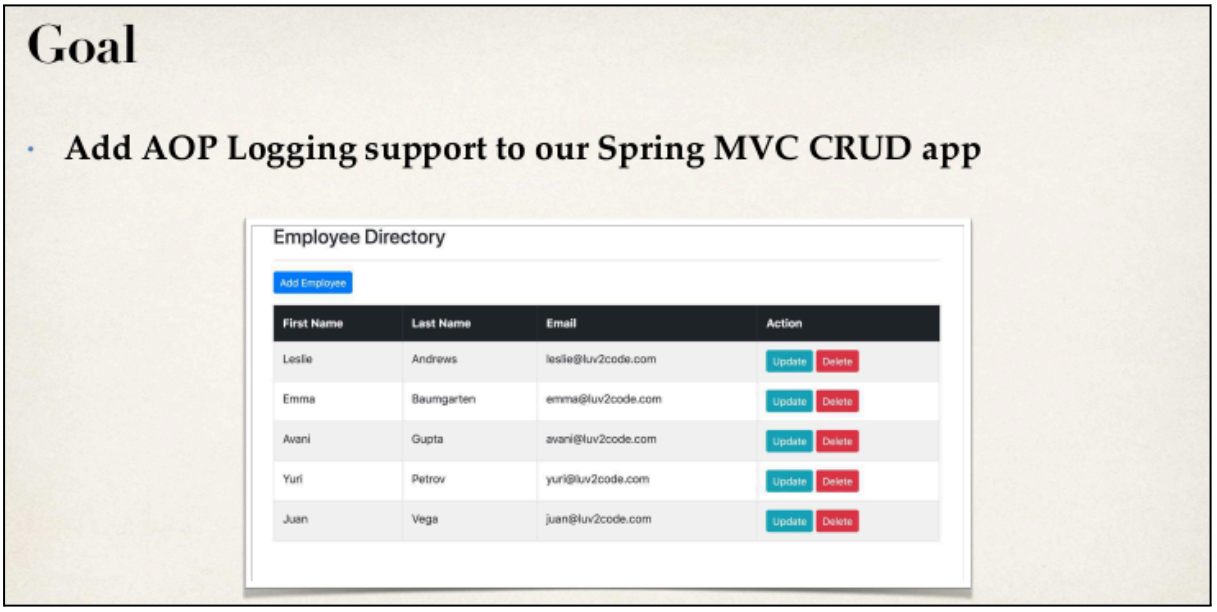
Rethrow Exception

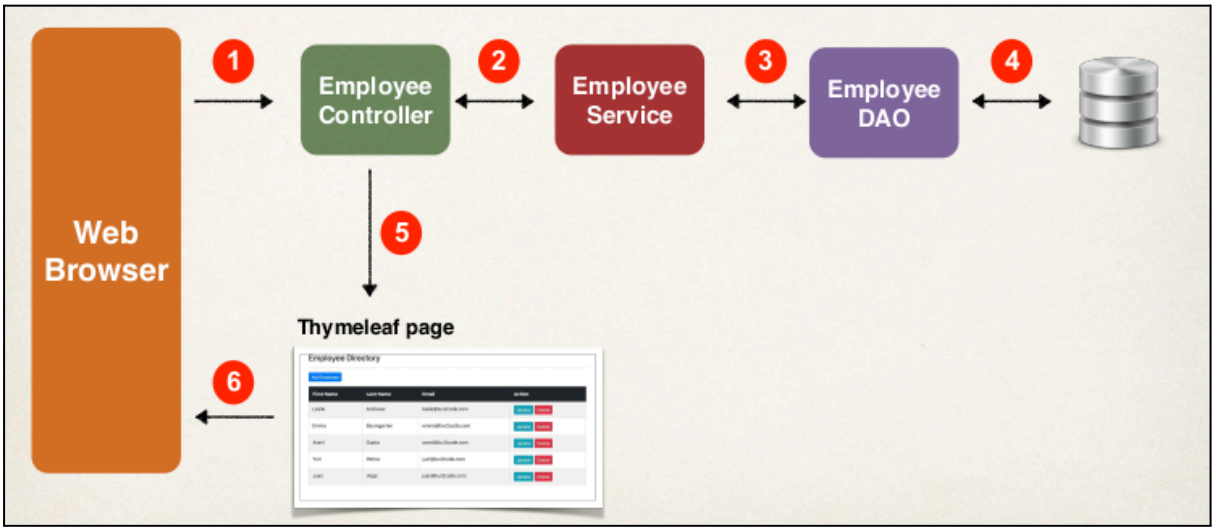
```
@Around("execution(* com.luv2code.aopdemo.service.*.getFortune(..))")
public Object afterGetFortune(
        ProceedingJoinPoint theProceedingJoinPoint) throws Throwable {

    try {
        // let's execute the method
        Object result = theProceedingJoinPoint.proceed();

        return result;
    }
    catch (Exception exc) {
        // log exception
        System.out.println("@Around advice: We have a problem " + exc);

        // rethrow it
        throw exc;
    }

}
```

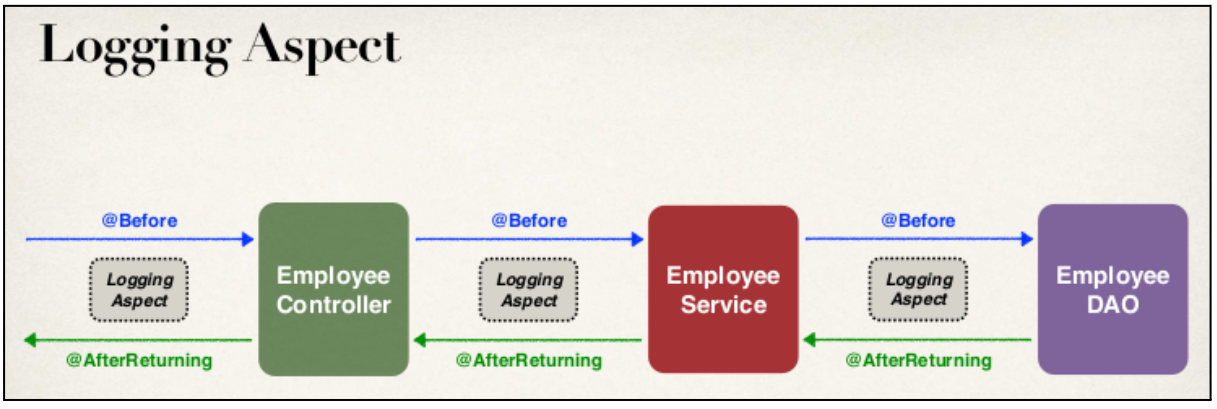
FAQ: JoinPoint vs ProceedingJoinPoint

- When to use JoinPoint vs ProceedingJoinPoint?
- Use JoinPoint with following advice type
    - @Before, @After, @AfterReturning, @AfterThrowing
- Use ProceedingJoinPoint with following advice type
    - @Around

Aspect-Oriented Programming (AOP) AOP and Spring MVC

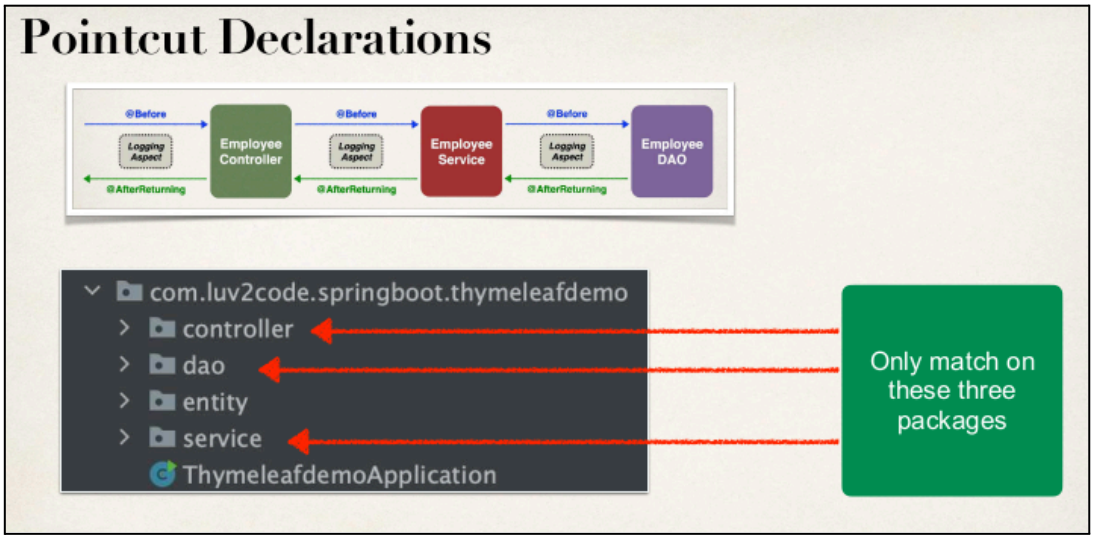






Development Process

1. Add Spring Boot AOP Starter to Maven pom le
2. Create Aspect
    1. Add logging support
    2. Setup pointcut declarations
    3. Add @Before advice
    4. Add @AfterReturning advice



EmployeeController.java | DemoLoggerAspect.java

```java
@Aspect
@Component
public class DemoLoggerAspect {
    2 usages
    private Logger logger = Logger.getLogger(getClass().getName());

    @Pointcut("execution(* com.example.aopcrudapp.controller.*.*(..))")
    private void forControllerPackage() {}

    @Pointcut("execution(* com.example.aopcrudapp.service.*.*(..))")
    private void forServicePackage() {}

    @Pointcut("execution(* com.example.aopcrudapp.dao.*.*(..))")
    private void forDaoPackage() {}

    @Pointcut("forControllerPackage() || forServicePackage() || forDaoPackage()")
    private void forAppFlow() {}

    @Before("forAppFlow()")
    public void before(JoinPoint joinPoint) {
        String method = joinPoint.getSignature().toShortString();
        logger.info( msg: "=======>>> Executing @Before advice on method : "+method);
        Object[] args = joinPoint.getArgs();
        for (Object arg : args) {
            logger.info( msg: "=======>>> Argument : "+arg);
        }
    }
}
```

DemoLoggerAspect.java

```
41      @AfterReturning(
42              pointcut = "forAppFlow()",
43              returning = "result"
44      )
45      public void afterReturning(JoinPoint joinPoint, Object result) {
46          String method = joinPoint.getSignature().toShortString();
47          logger.info( msg: "======>>> Executing @AfterReturning advice on method : "+method);
48          logger.info( msg: "======>>> Result : "+result);
49      }
50  }
51
```

Console Actuator

```
formInitiator       : HHH000489: No JTA platform available (set 'hibernate.transaction.jta.platform' to enable JTA platform integration)
yManagerFactoryBean : Initialized JPA EntityManagerFactory for persistence unit 'default'
JpaWebConfiguration : spring.jpa.open-in-view is enabled by default. Therefore, database queries may be performed during view rendering. Explicitly con
eReloadServer       : LiveReload server is running on port 35729
at.TomcatWebServer  : Tomcat started on port 8080 (http) with context path ''
dAppApplication     : Started AopCrudAppApplication in 7.681 seconds (process running for 8.519)
ocalhost].[/]       : Initializing Spring DispatcherServlet 'dispatcherServlet'
tcherServlet        : Initializing Servlet 'dispatcherServlet'
tcherServlet        : Completed initialization in 2 ms
.DemoLoggerAspect   : ======>>> Executing @Before advice on method : EmployeeController.listEmployees(..)
.DemoLoggerAspect   : ======>>> Argument : {}
.DemoLoggerAspect   : ======>>> Executing @Before advice on method : EmployeeServiceImpl.findAll()
.DemoLoggerAspect   : ======>>> Executing @Before advice on method : EmployeeRepository.findAllByOrderByLastNameAsc()
.DemoLoggerAspect   : ======>>> Executing @AfterReturning advice on method : EmployeeRepository.findAllByOrderByLastNameAsc()
.DemoLoggerAspect   : ======>>> Result : [Employee(id=1, firstName=Leslie, lastName=Andrews, email=leslie@luv2code.com), Employee(id=2, firstName=Emma,
.DemoLoggerAspect   : ======>>> Executing @AfterReturning advice on method : EmployeeServiceImpl.findAll()
.DemoLoggerAspect   : ======>>> Result : [Employee(id=1, firstName=Leslie, lastName=Andrews, email=leslie@luv2code.com), Employee(id=2, firstName=Emma,
.DemoLoggerAspect   : ======>>> Executing @AfterReturning advice on method : EmployeeController.listEmployees(..)
.DemoLoggerAspect   : ======>>> Result : employees/list-employees
```

Aspect-Oriented Programming (AOP)

Aspect-Oriented Programming (AOP) is useful in programming when you have certain concerns in your code that cut across multiple parts of your program. These concerns, called cross-cutting concerns, include things like logging, security, error handling, and performance monitoring.

1. Reusability : If you find yourself repeating the same code in multiple places to handle a cross-cutting concern, like logging or error handling, AOP can help. Instead of scattering this code throughout your application, you can define it once and apply it wherever needed.

2. Separation of Concerns : AOP allows you to separate core business logic from cross-cutting concerns. This makes your codebase cleaner and easier to maintain because each part of your program focuses on a single responsibility.

3. Modularity : AOP promotes modularity by encapsulating cross-cutting concerns into separate modules or aspects. This makes it easier to add, remove, or modify these concerns without affecting the rest of your codebase.

4. Consistency : AOP ensures consistent application of cross-cutting concerns across your codebase. By defining these concerns in one place, you can guarantee they are applied uniformly throughout your program.

Logging

Logging is the process of recording events, actions, or messages that occur during the execution of a program or system. These logs serve as a valuable tool for developers, system administrators, and support teams to understand the behavior of the software, diagnose issues, and monitor performance.

1. Recording Events : Logging involves capturing various events or messages that occur during the execution of a program. These events could include errors, warnings, informational messages, debug information, or any other significant actions.

2. Types of Logs :
- Error Logs : Capture details about errors or exceptions encountered during program execution. These logs help identify and troubleshoot issues.

- Warning Logs : Alert about potential problems or abnormal conditions that do not prevent the program from running but may require attention.

- Informational Logs : Provide general information about the program's execution, such as startup messages or progress updates.

- Debug Logs : Offer detailed information useful for debugging purposes, such as variable values or function execution paths.

3. Purpose :
- Debugging : Logs assist developers in diagnosing and fixing bugs by providing insights into the program's behavior and identifying the root cause of issues.

- Monitoring : Logs help monitor the health and performance of the system by tracking metrics like response times, resource usage, and user activities.

- Auditing : Logs serve as a record of actions performed within the system, facilitating compliance with regulatory requirements and investigating security incidents.