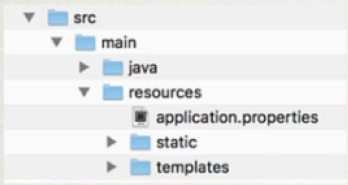
Dependency Coordinates

- To add given dependency project, we need
  - **Group ID**, **Artifact ID**
  - **Version** is optional …
    - Best practice is to include the version (repeatable builds)

DevOps

- May see this referred to as: **GAV**
  - **G**roup ID, **A**rtifact ID and **V**ersion

Static Content



WARNING:

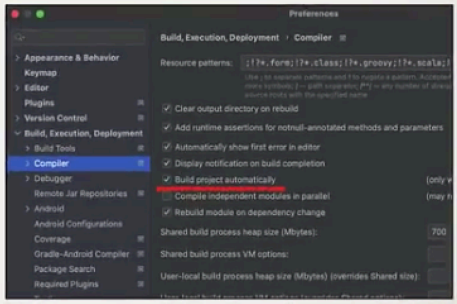
Do not use the `src/main/webapp` directory if your application is packaged as a JAR.

Although this is a standard Maven directory, it works only with WAR packaging.

It is silently ignored by most build tools if you generate a JAR.

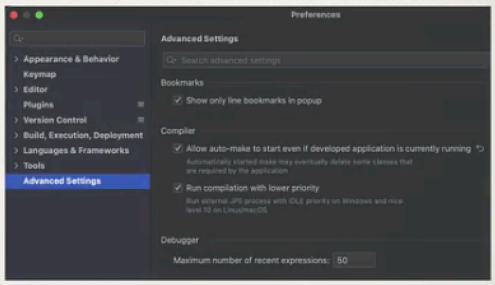
IntelliJ Community Edition - DevTools

- IntelliJ Community Edition does not support DevTools by default
- Select: **Preferences > Build, Execution, Deployment > Compiler**
  - Check box: **Build project automatically**



IntelliJ Community Edition

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



Spring Boot Properties

- Don't let the 1,000+ properties overwhelm you
- The properties are roughly grouped into the following categories

Core | Web | Security | Data

Actuator | Integration | DevTools | Testing

Core Properties

Core

File: src/main/resources/application.properties

```
# Log levels severity mapping
logging.level.org.springframework=DEBUG
logging.level.org.hibernate=TRACE
logging.level.com.luv2code=INFO

# Log file name
logging.file.name=my-crazy-stuff.log
logging.file.path=c:/myapps/demo
...
```

Logging Levels

TRACE
DEBUG
INFO
WARN
ERROR
FATAL
OFF

Spring Boot Logging
www.luv2code.com/spring-boot-logging

Inversion of Control (IoC)

The approach of outsourcing the construction and management of objects.

Spring Container

- Primary functions
  - Create and manage objects *(Inversion of Control)*
  - Inject object's dependencies *(Dependency Injection)*

Step 1: Define the dependency interface and class

File: Coach.java

```
package com.luv2code.springcoredemo;

public interface Coach {

    String getDailyWorkout();
}
```

@Component annotation marks the class as a Spring Bean

File: CricketCoach.java

```
package com.luv2code.springcoredemo;

import org.springframework.stereotype.Component;

@Component
public class CricketCoach implements Coach {

    @Override
    public String getDailyWorkout() {
        return "Practice fast bowling for 15 minutes";
    }
}
```

@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

Injection Types - Which one to use?

- 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

2657

A JavaBean is just a standard. It is a regular Java `class`, except it follows certain conventions:

1. All properties are private (use getters/setters)
2. A public no-argument constructor
3. Implements `Serializable`.

That's it. It's just a convention. Lots of libraries depend on it though.

With respect to `Serializable`, from the API documentation:

> Serializability of a class is enabled by the class implementing the java.io.Serializable interface. Classes that do not implement this interface will not have any of their state serialized or deserialized. All subtypes of a serializable class are themselves serializable. The serialization interface has no methods or fields and serves only to identify the semantics of being serializable.

In other words, serializable objects can be written to streams, and hence files, object databases, anything really.

Also, there is no syntactic difference between a JavaBean and another class -- a class is a JavaBean if it follows the standards.

There is a term for it, because the standard allows libraries to programmatically do things with class instances you define in a predefined way. For example, if a library wants to stream any object you pass into it, it knows it can because your object is serializable (assuming the library requires your objects be proper JavaBeans).

Spring Injection Types

- Recommended by the spring.io development team
  - Constructor Injection: required dependencies
  - Setter Injection: optional dependencies
- Not recommended by the spring.io development team
  - Field Injection

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

Solution: Be specific! - @Qualifier

```
package com.luv2code.springcoredemo.rest;

import com.luv2code.springcoredemo.common.Coach;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.beans.factory.annotation.Qualifier;
…

@RestController
public class DemoController {

    private Coach myCoach;

    @Autowired
    public DemoController(@Qualifier("cricketCoach") Coach theCoach) {
        myCoach = theCoach;
    }

    @GetMapping("/dailyworkout")
    public String getDailyWorkout() {
        return myCoach.getDailyWorkout();
    }
}
```

Specify the bean id: cricketCoach

Same name as class, first character lower-case

Other bean ids we could use:
baseballCoach, trackCoach, tennisCoach

Lazy Initialization

- Instead of creating all beans up front, we can specify lazy initialization
- A bean will only be initialized in the following cases:
  - It is needed for dependency injection
  - Or it is explicitly requested
- Add the @Lazy annotation to a given class

Lazy Initialization

Lazy initialization feature
is disabled by default.

You should profile your application
before configuring lazy initialization.

Avoid the common pitfall of premature optimization.

- 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

Explicitly Specify Bean Scope

```
import org.springframework.beans.factory.config.ConfigurableBeanFactory;
import org.springframework.context.annotation.Scope;
import org.springframework.stereotype.Component;


@Component
@Scope(ConfigurableBeanFactory.SCOPE_SINGLETON)
public class CricketCoach implements Coach {

    ...

}
```

Additional Spring 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 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!

Prototype Beans and Lazy Initialization

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

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

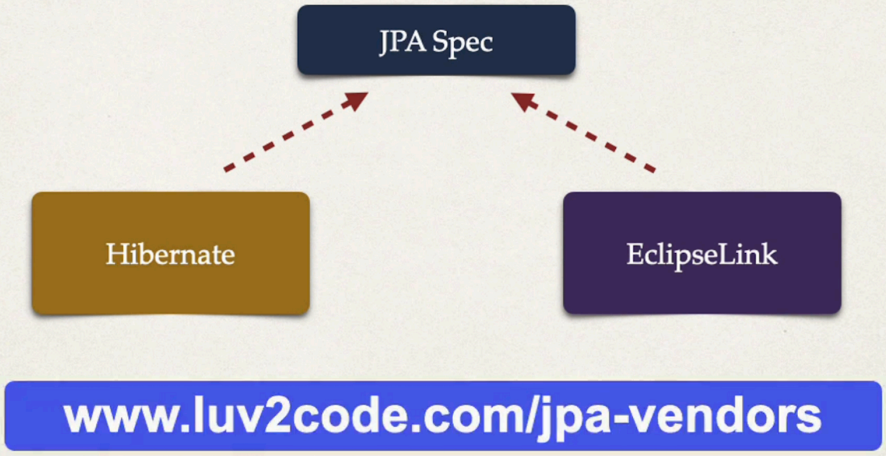
Configure AWS S3 Client using @Bean

```
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;
    }
}
```

Inject the S3Client as a bean

```
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;
    }

  …
}
```

JPA - Vendor Implementations



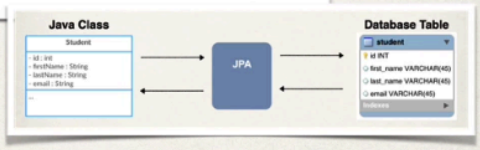
www.luv2code.com/jpa-vendors

Hibernate / JPA and JDBC

- Hibernate / JPA uses JDBC for all database communications

Step 1: Define DAO interface

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

public interface StudentDAO {

    void save(Student theStudent);

}
```



Step 2: Define DAO implementation


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

public class StudentDAOImpl implements StudentDAO {

    private EntityManager entityManager;

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

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

}
```

Inject the Entity Manager

Spring @Transactional

- Spring provides an **@Transactional** annotation
- **Automagically** begin and end a transaction for your JPA code
  - No need for you to explicitly do this in your code
- This Spring **magic** happens behind the scenes

Step 2: Define DAO implementation

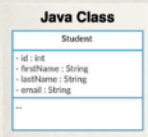
Specialized annotation for repositories

Supports component scanning

Translates JDBC exceptions

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

@Repository
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);
    }

}
```

Retrieving all Students



Name of JPA Entity ... the class name

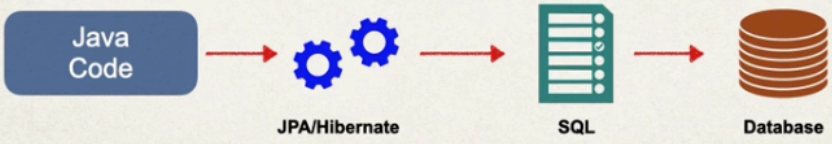
```
TypedQuery<Student> theQuery = entityManager.createQuery("FROM Student", Student.class);

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

Create database tables: `student`

- JPA/Hibernate provides an option to automagically create database tables
- Creates tables based on Java code with JPA/Hibernate annotations
- Useful for development and testing



Configuration - application.properties

```
spring.jpa.hibernate.ddl-auto=PROPERTY-VALUE
```

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

Basic Projects

- If you want to create tables once … and then keep data, use: `update`

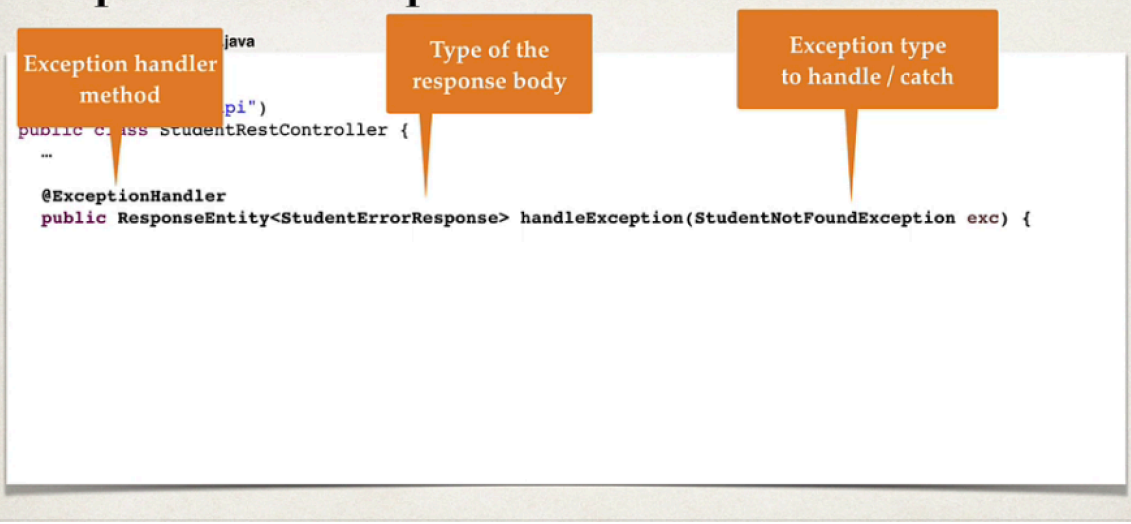
```
spring.jpa.hibernate.ddl-auto=update
```

- However, will ALTER database schema based on latest code updates
- Be VERY careful here … only use for basic projects

Step 4: Add exception handler method

- Define exception handler method(s) with `@ExceptionHandler` annotation
- Exception handler will return a `ResponseEntity`
- `ResponseEntity` is a wrapper for the HTTP response object
- `ResponseEntity` provides fine-grained control to specify:
  - HTTP status code, HTTP headers and Response body

Step 4: Add exception handler method



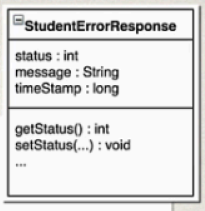
Step 4: Add exception handler method

File: StudentRestController.java

```
@RestController
@RequestMapping("/api")
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




```
{
    "status": 404,
    "message": "Student id not found - 9999",
    "timeStamp": 15261496
}
```

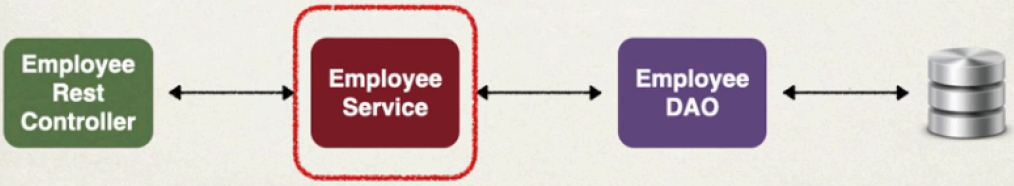
Spring @ControllerAdvice

- `@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

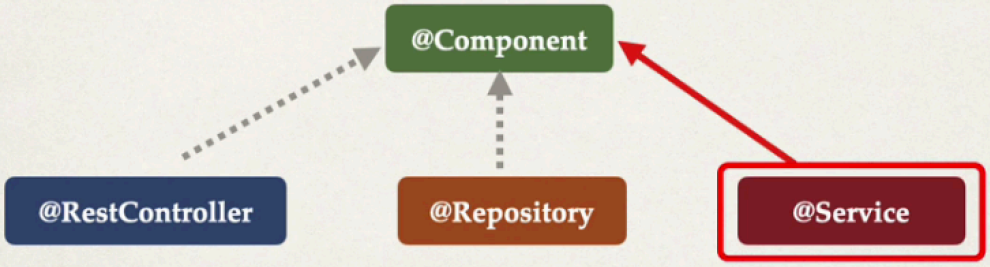
Purpose of Service Layer

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



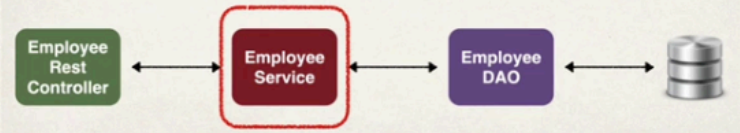
Specialized Annotation for Services

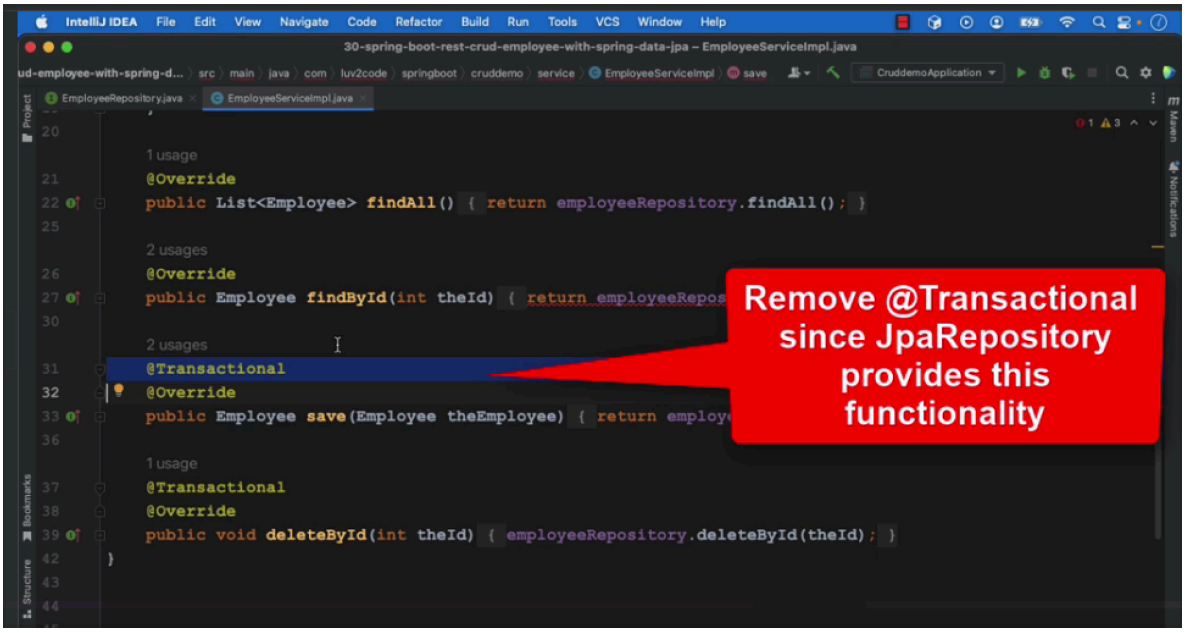
- Spring provides the **@Service** annotation



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





Step 1: Add Spring Data REST to POM file

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

That's it!!!

Absolutely NO CODING required

Spring Data REST will scan for JpaRepository

HTTP Method		CRUD Action
POST	/employees	Create a new employee
GET	/employees	Read a list of employees
GET	/employees/{employeeId}	Read a single employee
PUT	/employees/{employeeId}	Update an existing employee
DELETE	/employees/{employeeId}	Delete an existing employee

Get these REST endpoints for free

In A Nutshell

For Spring Data REST, you only need 3 items

We already have these two

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

Only item that is new

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

Solution

- Specify plural name / path with an annotation

```
@RepositoryRestResource(path="members")
public interface EmployeeRepository extends JpaRepository<Employee, Integer> {

}
```

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
```

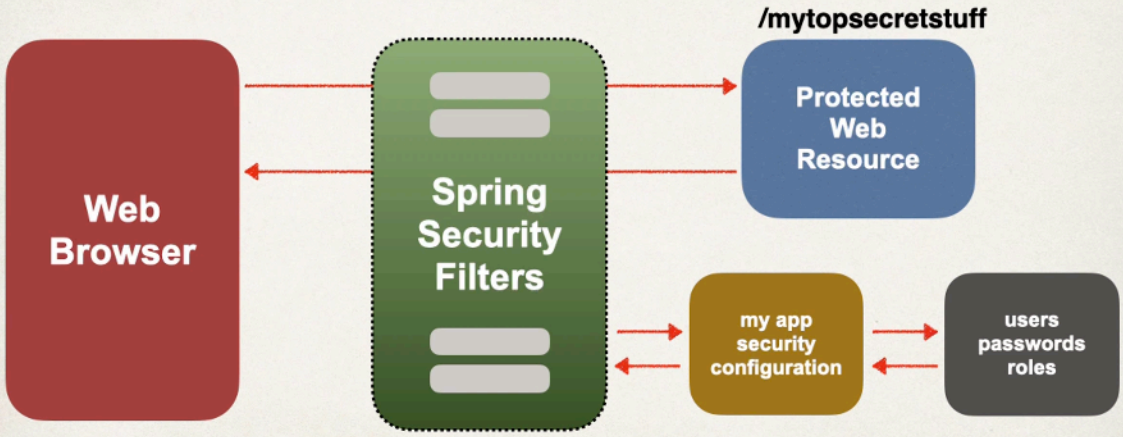
- Sort by first name, descending

```
http://localhost:8080/employees?sort=firstName,desc
```

- Sort by last name, then first name, ascending

```
http://localhost:8080/employees?sort=lastName,firstName,asc
```

Spring Security Overview



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

Enabling Spring Security

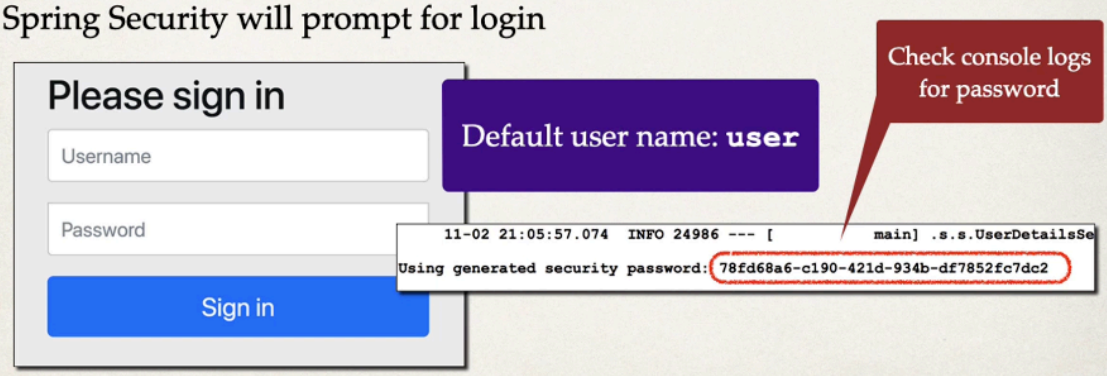
1. Edit **`pom.xml`** and add **`spring-boot-starter-security`**

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

2.This will ***automagically*** secure all endpoints for application

Secured Endpoints

- Now when you access your application
- Spring Security will prompt for login



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=tiger
```

Authentication and Authorization

- In-memory
- JDBC
- LDAP
- Custom / Pluggable
- *others …*

users
passwords
roles

We will cover password storage in DB as plain-text AND encrypted

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
…	…

Restricting Access to Roles

- General Syntax

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

```
requestMatchers(<< add path to match on >>)
        .hasRole(<< authorized role >>)
```

Single role

```
@Configuration
public class SpringSecurityConfig {

    Payam Mousavi *
    @Bean
    public SecurityFilterChain filterChain(HttpSecurity httpSecurity) throws Exception {
        // Authorization roles
        httpSecurity.authorizeHttpRequests(auth -> auth
                .requestMatchers(new AntPathRequestMatcher( pattern: "/api/employees", HttpMethod.GET.name())).hasRole("EMPLOYEE")
                .requestMatchers(new AntPathRequestMatcher( pattern: "/api/employees/**", HttpMethod.GET.name())).hasRole("EMPLOYEE")
                .requestMatchers(new AntPathRequestMatcher( pattern: "/api/employees", HttpMethod.POST.name())).hasRole("MANAGER")
                .requestMatchers(new AntPathRequestMatcher( pattern: "/api/employees", HttpMethod.PUT.name())).hasRole("MANAGER")
                .requestMatchers(new AntPathRequestMatcher( pattern: "/api/employees", HttpMethod.DELETE.name())).hasRole("ADMIN")
                .anyRequest().authenticated()
        );
```

When to use CSRF Protection?

- The Spring Security team recommends
  - Use CSRF protection for any normal browser web requests
  - Traditional web apps with HTML forms to add/modify data
- If you are building a REST API for non-browser clients
  - you *may* want to disable CSRF protection
- In general, not required for stateless REST APIs
  - That use POST, PUT, DELETE and/or PATCH

Pull It Together

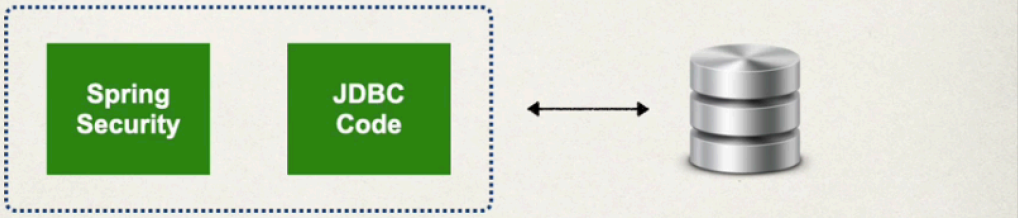
```
@Bean
public SecurityFilterChain filterChain(HttpSecurity http) throws Exception {

   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());

   // disable Cross Site Request Forgery (CSRF)
   http.csrf(csrf -> csrf.disable());


   return http.build();
}
```

In general, CSRF is not required for stateless REST APIs that use POST, PUT, DELETE and/or PATCH

Database Support in Spring Security

Out-of-the-box

- Follow Spring Security's predefined table schemas



Development Process

Step-By-Step

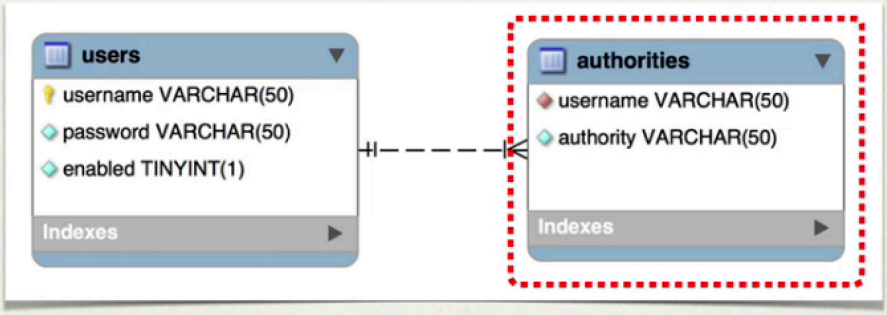
1. Develop SQL Script to set up database tables

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

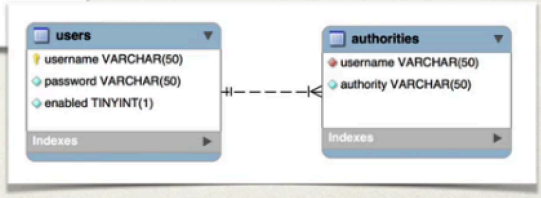
The encoding algorithm id

The password

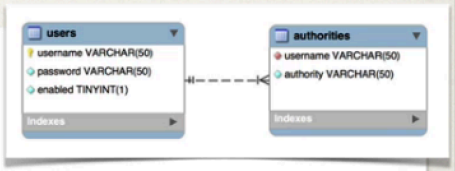
```
INSERT INTO `users`
VALUES
('john','{noop}test123',1),
('mary','{noop}test123',1),
('susan','{noop}test123',1);
```

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

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



Step 1: Develop SQL Script to setup database tables

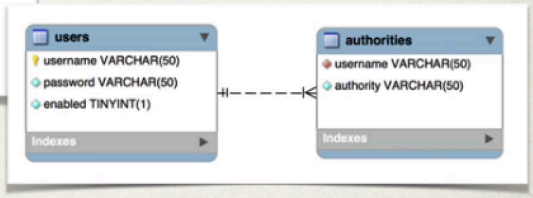



```
CREATE TABLE `authorities` (
  `username` varchar(50) NOT NULL,
  `authority` varchar(50) NOT NULL,

  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

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

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



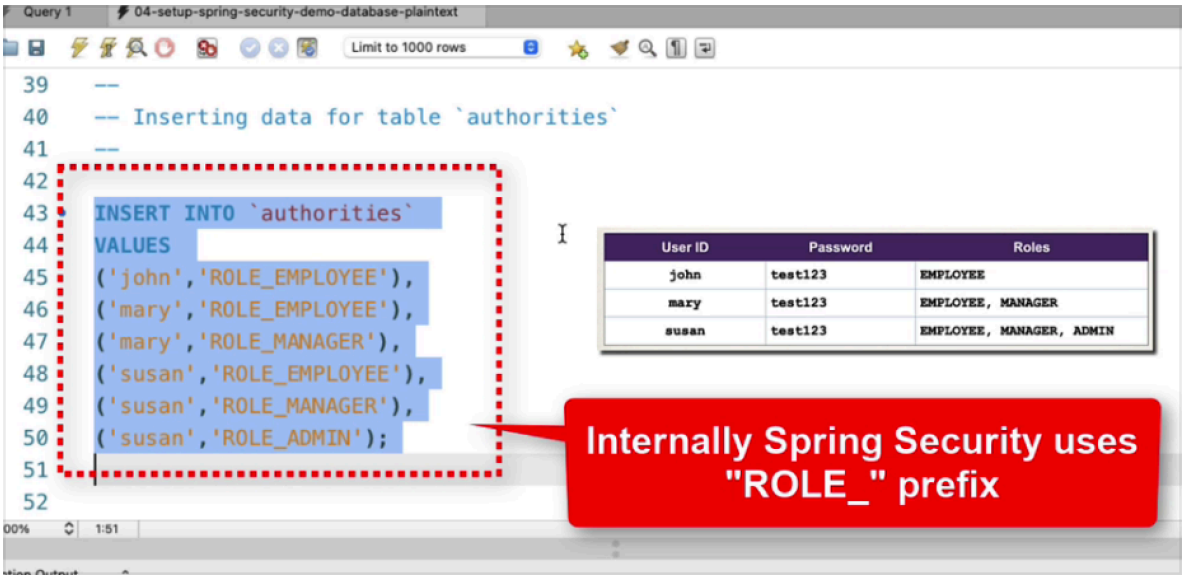
Step 4: Update Spring Security to use JDBC

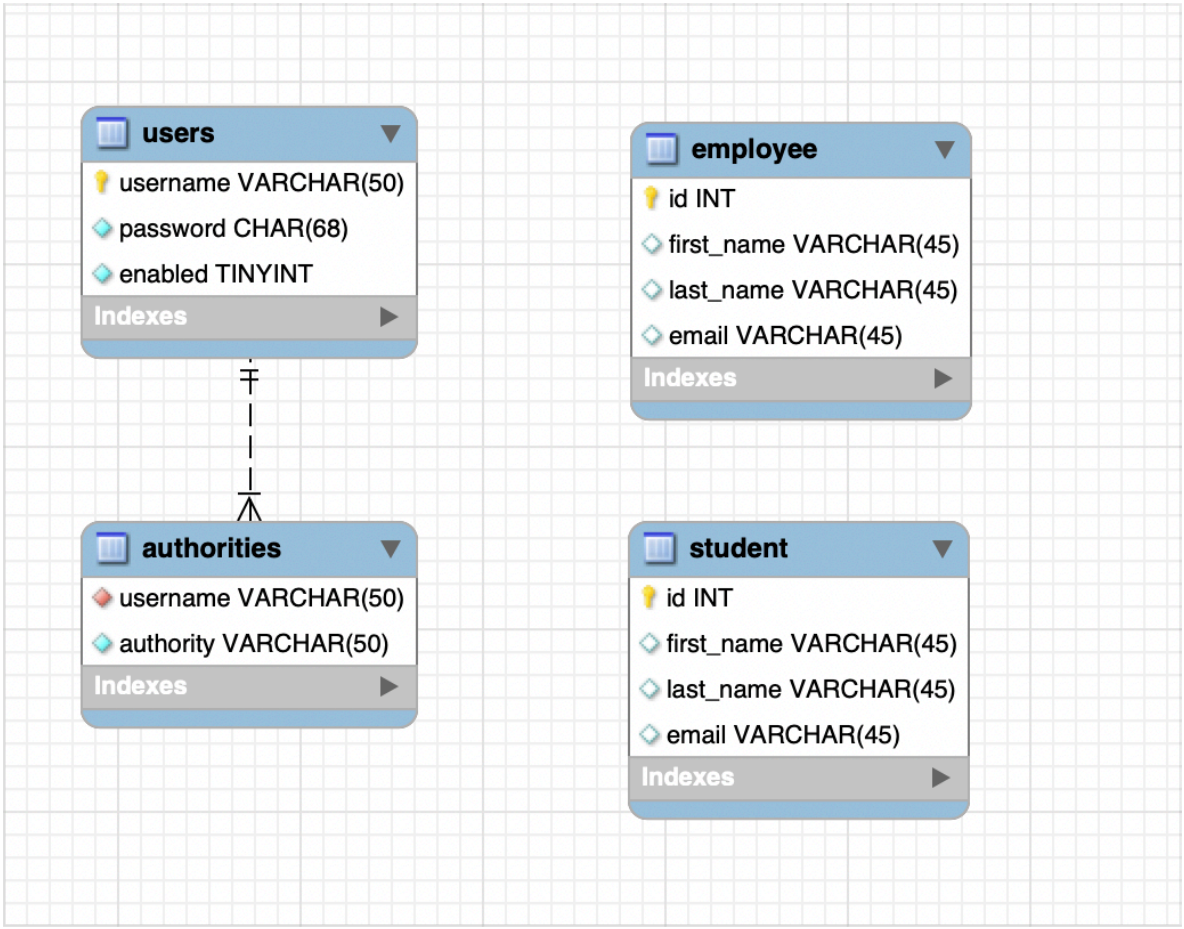
Inject data source
Auto-configured by Spring Boot

```
@Configuration
public class DemoSecurityConfig {

    @Bean
    public UserDetailsManager userDetailsManager(DataSource dataSource) {

        return new JdbcUserDetailsManager(dataSource);
    }

    ...
}
```

No longer
hard-coding users :-)

Tell Spring Security to use
JDBC authentication
with our data source


Query 1
04-setup-spring-security-demo-database-plaintext
Limit to 1000 rows
39 --
40 -- Inserting data for table `authorities`
41 --
42
43 INSERT INTO `authorities`
44 VALUES
45 ('john','ROLE_EMPLOYEE'),
46 ('mary','ROLE_EMPLOYEE'),
47 ('mary','ROLE_MANAGER'),
48 ('susan','ROLE_EMPLOYEE'),
49 ('susan','ROLE_MANAGER'),
50 ('susan','ROLE_ADMIN');
51
52
User ID
Password
Roles
john
test123
EMPLOYEE
mary
test123
EMPLOYEE, MANAGER
susan
test123
EMPLOYEE, MANAGER, ADMIN
Internally Spring Security uses "ROLE_" prefix


users
username VARCHAR(50)
password CHAR(68)
enabled TINYINT
Indexes
employee
id INT
first_name VARCHAR(45)
last_name VARCHAR(45)
email VARCHAR(45)
Indexes
authorities
username VARCHAR(50)
authority VARCHAR(50)
Indexes
student
id INT
first_name VARCHAR(45)
last_name VARCHAR(45)
email VARCHAR(45)
Indexes

Password Storage - Best Practice

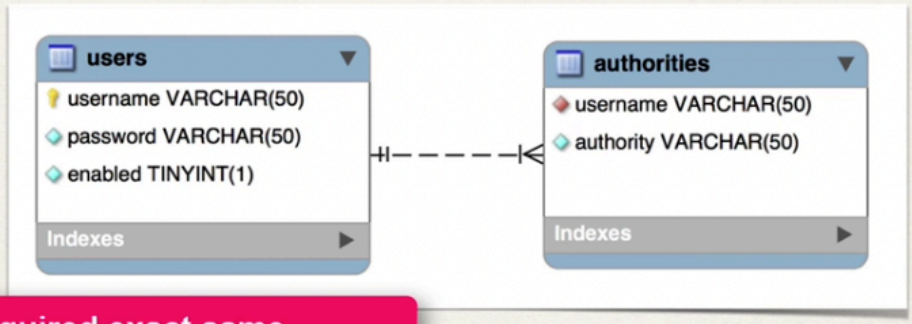
Best Practice

- The best practice is store passwords in an encrypted format

username	password	enabled
john	{bcrypt}$2a$10$qeS0HEh7urweMojsnwNAR.vcXJeXR1UcMRZ2WcGQl9YeuspUdgF.q	1
mary	{bcrypt}$2a$10$qeS0HEh7urweMojsnwNAR.vcXJeXR1UcMRZ2WcGQl9YeuspUdgF.q	1
susan	{bcrypt}$2a$10$qeS0HEh7urweMojsnwNAR.vcXJeXR1UcMRZ2WcGQl9YeuspUdgF.q	1

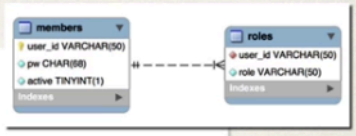
Encrypted version of password

Default Spring Security Database Schema



Required exact same
table names and column names

Step 2: Update Spring Security Configuration




```
@Configuration
public class DemoSecurityConfig {

    @Bean
    public UserDetailsManager userDetailsManager(DataSource dataSource) {
        JdbcUserDetailsManager theUserDetailsManager = new JdbcUserDetailsManager(dataSource);

        theUserDetailsManager
           .setUsersByUsernameQuery("select user_id, pw, active from members where user_id=?");

        theUserDetailsManager
           .setAuthoritiesByUsernameQuery("select user_id, role from roles where user_id=?");

        return theUserDetailsManager;
    }

    ...

}
```

How to find users

What is Thymeleaf?

- Thymeleaf is a Java templating engine
- Commonly used to generate the HTML views for
- However, it is a general purpose templating engine

Thymeleaf

www.thymeleaf.org

Separate project
Unrelated to spring.io

You can create Java apps with Thymeleaf

No need for Spring

web ap

But there is a lot of synergy between the two projects

Step 1: Add Thymeleaf to Maven pom file

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

start.spring.io

Dependencies

Spring Web WEB
Build web, including RESTful, applications using Spring MVC.
Uses Apache Tomcat as the default embedded container.

Thymeleaf TEMPLATE ENGINES
A modern server-side Java template engine for both web and standalone environments. Allows HTML to be correctly displayed in browsers and as static prototypes.

Step 2: Develop Spring MVC Controller

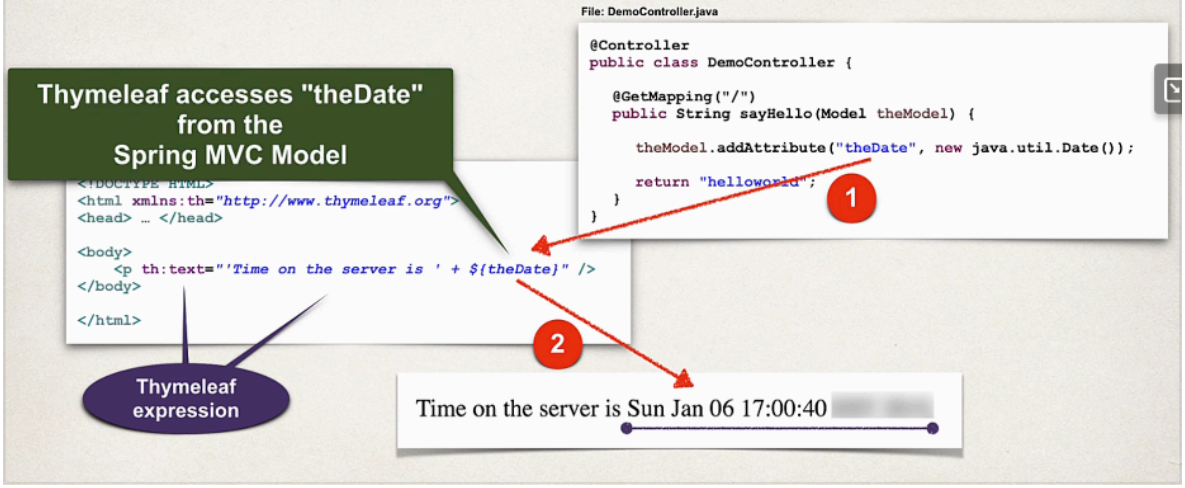
File: DemoController.java

```
@Controller
public class DemoController {

  @GetMapping("/")
  public String sayHello(Model theModel) {

    theModel.addAttribute("theDate", new java.util.Date());

    return "helloworld";
  }
}
```

Where to place Thymeleaf template?

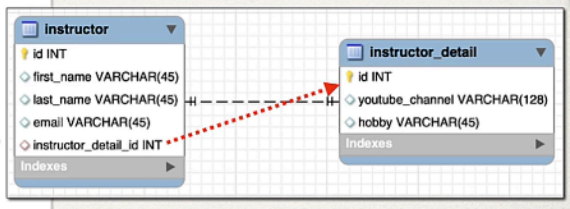
- In Spring Boot, your Thymeleaf template files go in
  - **src/main/resources/templates**
- For web apps, Thymeleaf templates have a .**html** extension

Step 3: Create Thymeleaf template



Step 3: Create Instructor class - @OneToOne

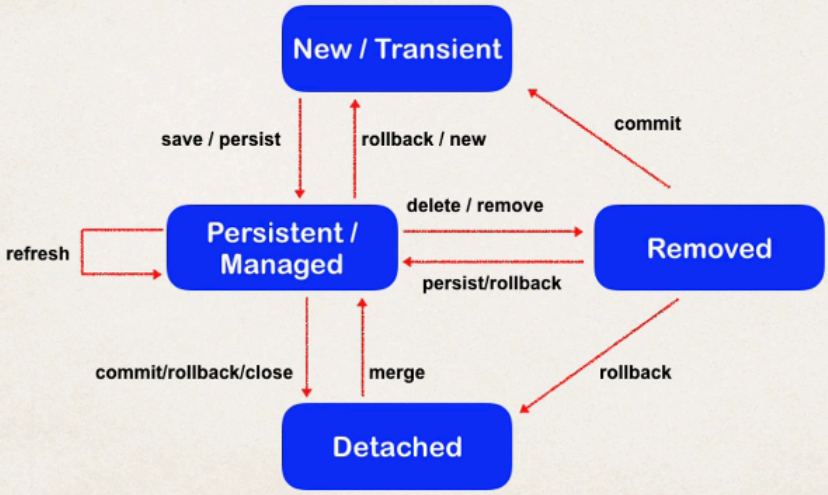
```
@Entity
@Table(name="instructor")
public class Instructor {
    ...
    @OneToOne
    @JoinColumn(name="instructor_detail_id")
    private InstructorDetail instructorDetail;
    ...
    // constructors, getters / setters
}
```



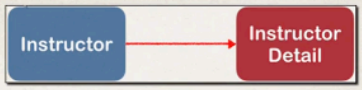
Entity Lifecycle

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

Entity Lifecycle - session method calls



@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

Add @OneToMany annotation

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

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

  public List<Course> getCourses() {
    return courses;
  }

  public void setCourses(List<Course> courses) {
    this.courses = courses;
  }

  …
```

Refers to “instructor” property in “Course” class

More on 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

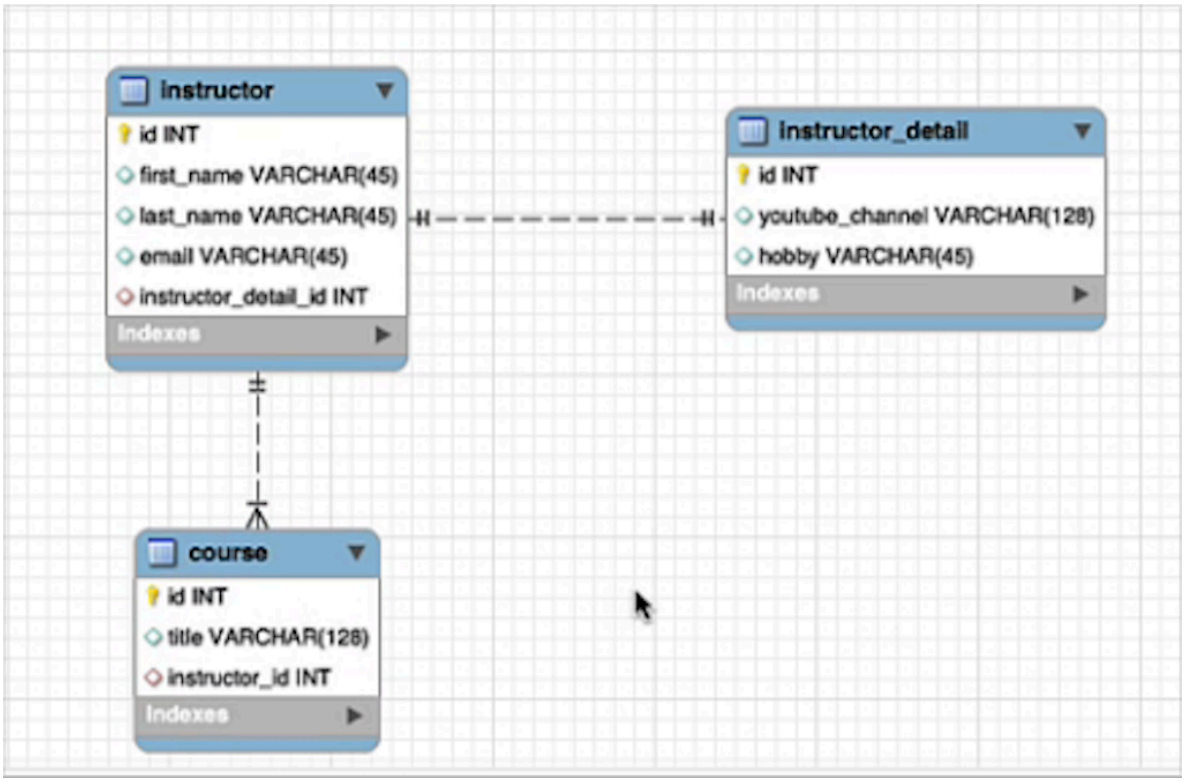
```
@Entity
@Table(name="instructor")
public class Instructor {
  …

  @OneToMany(mappedBy="instructor",
             cascade={CascadeType.PERSIST, CascadeType.MERGE
                      CascadeType.DETACH, CascadeType.REFRESH})
  private List<Course> courses;


  …

}
```

Do not apply cascading deletes!

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!



Default Fetch Types

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

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

More about Lazy Loading

- If the Hibernate session is closed
  - And you attempt to retrieve lazy data
  - Hibernate will throw an exception

Watch out for this!

Add new method to find instructor with courses

File: AppDAOImpl.java

```
@Override
public Instructor findInstructorByIdJoinFetch(int theId) {

    // create query
    TypedQuery<Instructor> query = entityManager.createQuery(
                                        "select i from Instructor i "
                                                + "JOIN FETCH i.courses "
                                                + "where i.id = :data", Instructor.class);

    query.setParameter("data", theId);

    // execute query
    Instructor instructor = query.getSingleResult();

    return instructor;
}
```

Even with Instructor
@OneToMany(fetchType=LAZY)

This code will still retrieve Instructor AND Courses

The JOIN FETCH is similar to EAGER loading

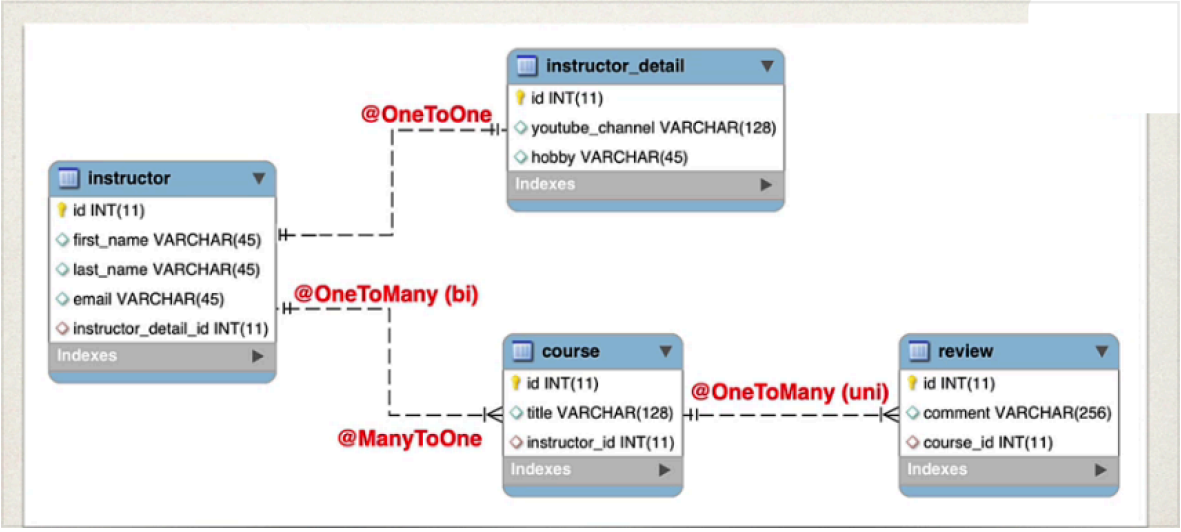
303

In this two queries, you are using JOIN to query all employees that have at least one department associated.

But, the difference is: in the first query you are returning only the Employes for the Hibernate. In the second query, you are returning the Employes **and** all Departments associated.

So, if you use the second query, you will not need to do a new query to hit the database again to see the Departments of each Employee.

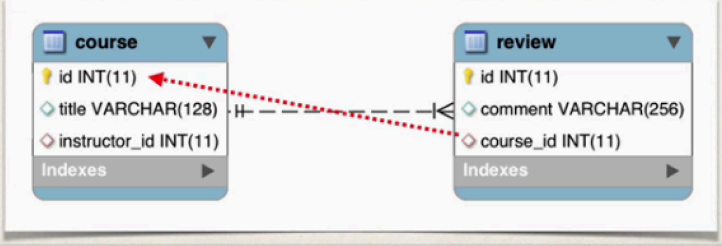
You can use the second query when you are sure that you will need the Department of each Employee. If you not need the Department, use the first query.



More: @JoinColumn

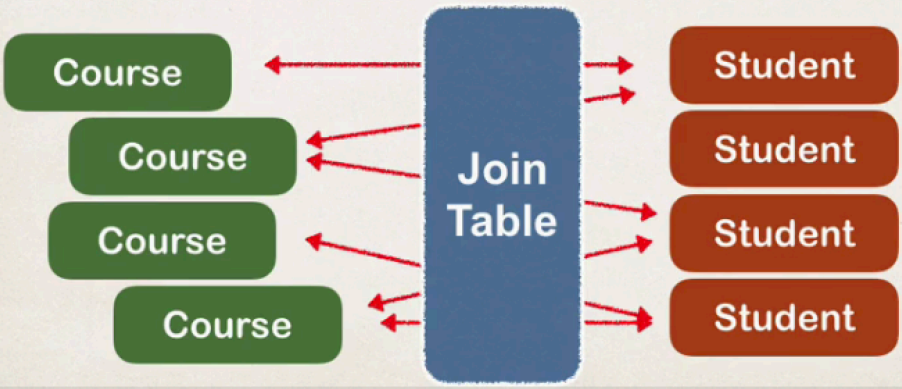
```
public class Course {
  …

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

- 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



Keep track of relationships

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



Join Table

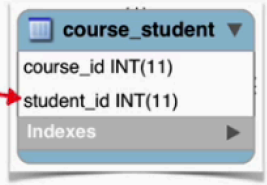
A table that provides a mapping between two tables.

It has foreign keys for each table
to define the mapping relationship.

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
  …

}
```

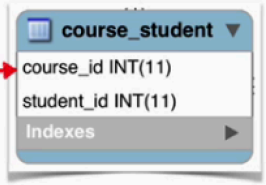
Refers to “student_id” column
in “course_student” join table

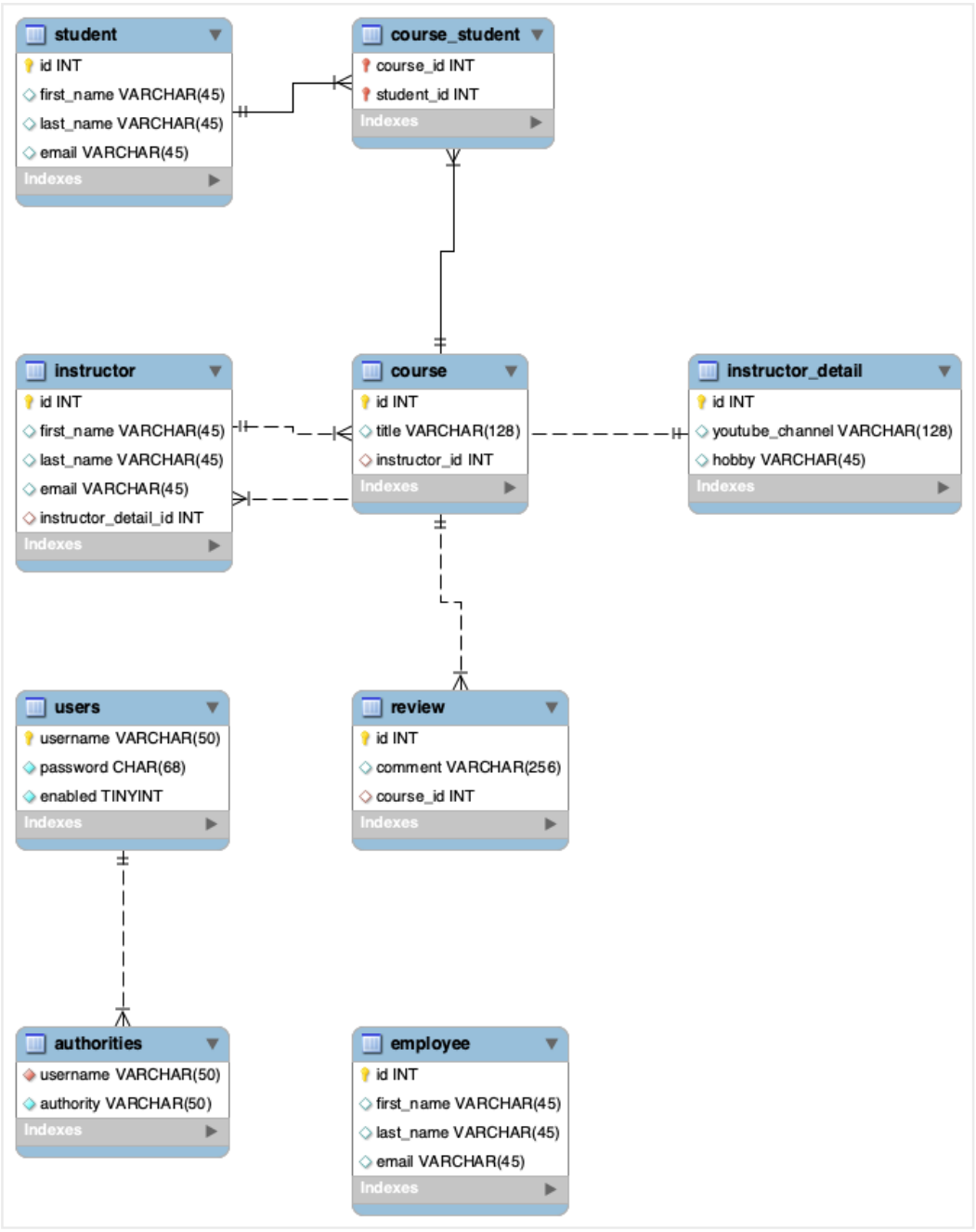


Add @ManyToMany annotation

```
@Entity
@Table(name="student")
public class Student {
  …

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

  // getter / setters
  …

}
```

Refers to “course_id” column in “course_student” join table




student
id INT
first_name VARCHAR(45)
last_name VARCHAR(45)
email VARCHAR(45)
Indexes
course_student
course_id INT
student_id INT
Indexes
instructor
id INT
first_name VARCHAR(45)
last_name VARCHAR(45)
email VARCHAR(45)
instructor_detail_id INT
Indexes
course
id INT
title VARCHAR(128)
instructor_id INT
Indexes
instructor_detail
id INT
youtube_channel VARCHAR(128)
hobby VARCHAR(45)
Indexes
users
username VARCHAR(50)
password CHAR(68)
enabled TINYINT
Indexes
review
id INT
comment VARCHAR(256)
course_id INT
Indexes
authorities
username VARCHAR(50)
authority VARCHAR(50)
Indexes
employee
id INT
first_name VARCHAR(45)
last_name VARCHAR(45)
email VARCHAR(45)
Indexes