Western Norway
University of
Applied Sciences

DAT152 – Advanced Web Applications

Web Frameworks

Today’s agenda

- Web frameworks
  - Types of web frameworks
  - Patterns we explored in the last lecture
  - How they are comparable to the Spring Web MVC framework
- Extend our library service using Spring Web MVC framework
  - Spring MVC + Spring boot
  - Spring Data, JPA

Web Frameworks

- Classification of web framework is based on the “programming model” offered to the developers.
- For webapps, the core model is based on a request-response style over HTTP stateless protocol
- Frameworks that expose this model by linking “actions” to URLs (requests) are usually called **action-based**
- Frameworks that hide the request-response model behind the UI components are often called **component-based**.

Web Frameworks - Examples

- Action-based
  - Spring Web MVC (spring.io)
  - Apache Struts (struts.apache.org)
- Component-based
  - Vaadin (vaading.com)

We'll focus more on using Spring Web MVC

- Being a framework, it offers some built-in services:
  - Integration with template languages
  - Others:
    - Form validation
    - Error handling
    - Request parameter type conversion
    - Internationalization
    - IDE integration
    - Security features

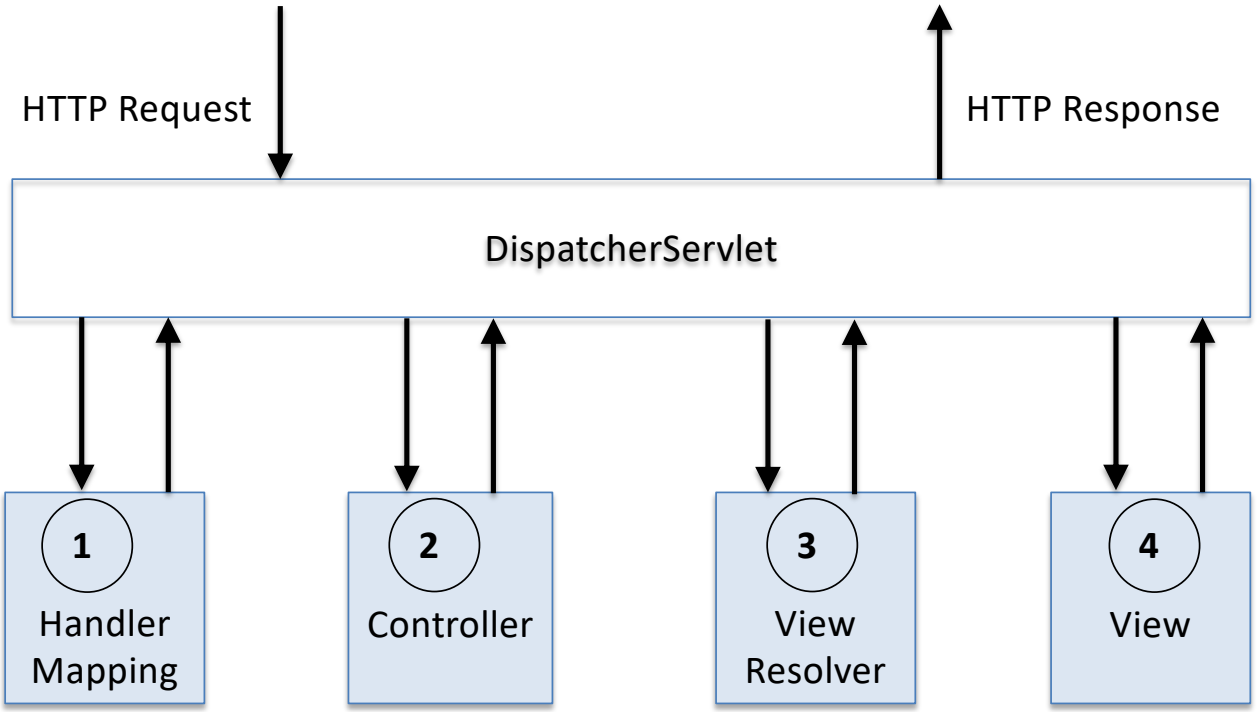
Spring Web MVC

- Patterns
  - MVC
  - Inversion of Control (IoC)/Dependency Injection
  - Configuration with Annotations/XML
  - ++
- Spring boot
  - Automatic configuration for spring applications

Why strong coupling is bad and why you need dependency injection: You may want to read this paper
Oyetoyan, T. D., Cruzes, D. S., & Thurmann-Nielsen, C. (2015). A decision support system to refactor class cycles. In *2015 IEEE International Conference on Software Maintenance and Evolution (ICSME)* (pp. 231-240).

Spring Web MVC

- From the last lecture, we explored some MVC strategies
- Spring Web MVC uses FrontController Pattern (DispatcherServlet)
  - org.springframework.web.servlet.DispatcherServlet



Spring Web MVC

1. When HTTP request arrives, DispatcherServlet contacts the HandlerMapping to call the appropriate Controller.
2. The Controller takes the request and calls the appropriate service methods based on the GET or POST method.
   - The service method will set model data based on defined business logic and returns view name to the DispatcherServlet.
3. The DispatcherServlet will also use the ViewResolver to select the defined view for the request.
4. Once view is finalized, The DispatcherServlet passes the model data to the view which is finally rendered on the browser.

DispatcherServlet in web.xml

DispatcherServlet (FrontController) receives all the HTTP requests (based on url pattern) and delegates them to controller classes

```
<web-app>
…
<servlet>
   <servlet-name>Front-controller-dispatcher</servlet-name>
   <servlet-class>
      org.springframework.web.servlet.DispatcherServlet
   </servlet-class>
</servlet>

<servlet-mapping>
   <servlet-name> Front-controller-dispatcher</servlet-name>
   <url-pattern>/</url-pattern>
</servlet-mapping>
…
</web-app>
```

Spring default DispatcherServlet

URL to be captured by DispatcherServlet

Compare to Apache Strut Framework

- Apache Strut framework uses similar configuration – frontcontroller dispatcher (StrutsPrepareAndExecuteFilter)

Filter Example (web.xml)

```
<web-app xmlns="http://xmlns.jcp.org/xml/ns/javaee"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/javaee
         http://xmlns.jcp.org/xml/ns/javaee/web-app_3_1.xsd"
         version="3.1">

    <filter>
        <filter-name>struts2</filter-name>
        <filter-class>org.apache.struts2.dispatcher.filter.StrutsPrepareAndExecuteFilter</filter-class>
    </filter>

    <filter-mapping>
        <filter-name>struts2</filter-name>
        <url-pattern>/*</url-pattern>
    </filter-mapping>

    <!-- ... -->

</web-app>
```

Configuring ViewResolver

- ViewResolver provides a mapping between view names and actual views.
- Different types of ViewResolver
  - InternalResourceViewResolver (jsp, servlets)
  - BeanNameViewResolver
  - ContentNegotingViewResolver (Content-Type)
  - etc.

Configuring ViewResolver

- e.g., InternalResourceViewResolver

Can also use special prefix for view names
- Redirecting (`return "redirect:viewbooks";`)
- Forwarding (`"forward:viewbooks";`)

```
<bean class = "org.springframework.web.servlet.view.InternalResourceViewResolver">
     <property name = "prefix" value = "/WEB-INF/jsp/" />
     <property name = "suffix" value = ".jsp" />
</bean>
```



```
@RequestMapping(value="/", method=RequestMethod.GET)
public String defaultView() {
    //Spring uses InternalResourceViewResolver and return
    index.jsp
    return "index";
}
```

If Controller returns "**index**", InternalResourceViewResolver looks for it as view **/WEB-INF/jsp/index.jsp**

Compare to Apache Strut Framework

- Apache Strut framework uses similar configuration for resolving an action to a view

struts.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE struts PUBLIC
    "-//Apache Software Foundation//DTD Struts Configuration 2.5//EN"
    "http://struts.apache.org/dtds/struts-2.5.dtd">

<struts>

    <constant name="struts.devMode" value="true" />

    <package name="basicstruts2" extends="struts-default">
        <action name="index">
            <result>/index.jsp</result>
        </action>
    </package>

</struts>
```

if the URL ends in index.action to redirect the browser to index.jsp

Similarity to our custom FrontController

	Our Custom FrontController	Spring Web MVC
HTTP Request Entry Point	FrontController	DispatcherServlet (=FrontController)
Actions	Action classes	Controller classes (@Controller, @Bean)
GET and POST commands	Command pattern – using Map	HandlerMapping (e.g, SimpleUrlHandlerMapping)
Page flow	FlowManager – Map	ViewResolver (e.g., InternalResourceViewResolver)
++		

Essence of spring boot

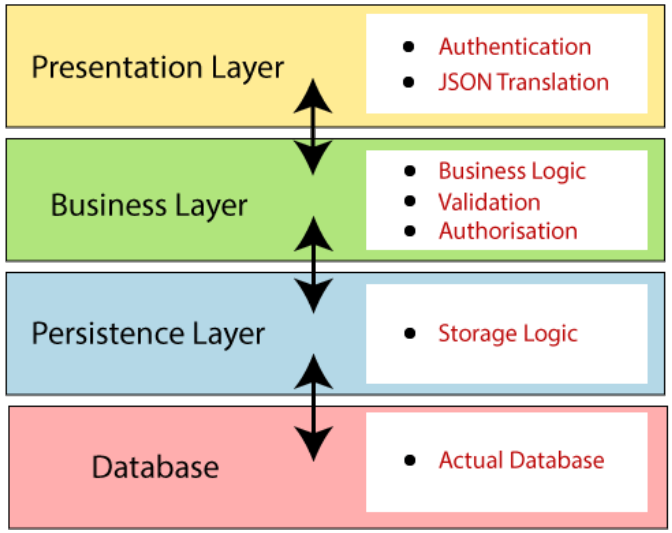
- Configuration can become a pain point when developing large applications
- Provides auto configuration and eliminates the need for xml-based manual configuration (web.xml)
  - @SpringBootApplication is a convenience annotation that enables the following:
  - @Configuration: Allows to register extra beans in the application context or import additional configuration classes
  - @EnableAutoConfiguration: Tells Spring Boot to start adding beans based on classpath settings, other beans, and various property settings.
  - @EnableWebMvc: Flags the application as a web application and activates key behaviors, such as setting up a DispatcherServlet. Spring Boot adds it automatically when it sees spring-webmvc on the classpath.
  - @ComponentScan: Tells Spring to look for other components, configurations, and services.

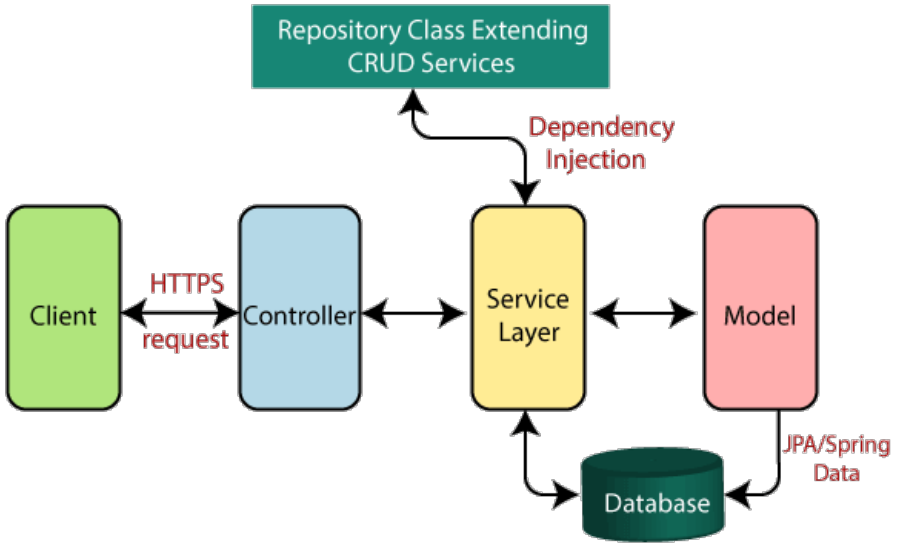
https://docs.spring.io/spring-boot/docs/2.0.x/reference/html/using-boot-using-springbootapplication-annotation.html
https://www.tutorialspoint.com/spring/spring_web_mvc_framework.htm

Spring boot

Defines 4-layers

- Presentation Layer
- Business Layer
- Persistence Layer
- Database Layer





https://www.javatpoint.com/spring-boot-architecture

application.properties

- Configurations are to be done here!

```
server.port=8090
server.error.path=/error


server.servlet.context-path=/demo
spring.mvc.servlet.path=/dat152


spring.thymeleaf.cache=false
spring.thymeleaf.enabled=true
spring.thymeleaf.prefix=classpath:/templates/
spring.thymeleaf.suffix=.html

spring.application.name=eLibrary Spring Boot Application

#spring.datasource.driver-class-name=org.h2.Driver
#;DB_CLOSE_DELAY=-1
#spring.datasource.url=jdbc:h2:file:~/h2/librarydb
spring.datasource.url=jdbc:h2:mem:librarydb
spring.datasource.driverClassName=org.h2.Driver
spring.datasource.username=sa
spring.datasource.password=
```

https://docs.spring.io/spring-boot/appendix/application-properties/index.html

Annotations

- Controller
  - @Controller
- Dependency injection
  - @Autowired
- Handling HTTP Request
  - @RequestMapping
  - @RequestBody
  - @PathVariable
  - @MatrixVariable
  - @RequestParam
- Handling HTTP Response
  - @ResponseBody
  - @ResponseStatus
  - @ExceptionHandler
- ++

@Bean
@Component
@WebListener
@Service
@Entity
@Table
@...

Annotations

- **@Controller**: Indicates that an annotated class is a "Controller"

```
@Controller
public class BookController {
    ...
}
```

- **@Autowired**: indicates a field, method, or constructor is to be autowired (injected) by Spring's *dependency injection* facilities

```
@Autowired
private BookRepository bookRepository;
```

Injection bug:
https://rules.sonarsource.com/java/tag/spring/RSPEC-3306/

Annotations

- **@RequestMapping**: Annotation for mapping web requests onto methods (used with controllers)
  - Accepts the following parameters
    - method (GET/POST/PUT/DELETE…)
    - produces (ContentType)
    - consumes (ContentType)
    - Params (e.g., @GetMapping(path = "/pets/{petId}", params = "myParam=myValue"))
    - Headers (e.g., @GetMapping(path = "/pets/{petId}", headers = "myHeader=myValue"))

```
@RequestMapping(value="/", method=RequestMethod.GET, produces="text/html")
public String defaultView() {
    return "index";
}
```

```
@PostMapping(path = "/pets", consumes = "application/json")
public void addPet(@RequestBody Pet pet) {
    // ...
}
```

Annotations

- **@PathVariable**: a method parameter should be bound to a URI template variable

```
@RequestMapping(value="viewbooks/{id}", method=RequestMethod.GET)
public String findOne(@PathVariable Long id, Model model) {
    ...
}
```

GET http://localhost:8090/viewbooks/1

- **@MatrixVariable**: Annotation for mapping name-value pairs in path segments

```
// GET /cars/2011;color=red,color=green,color=blue
@RequestMapping(value="/cars/{year}", method=RequestMethod.GET)
public String findCar((@PathVariable Long year, @MatrixVariable String color)
{
    ...;
}
```

Annotations

- **@RequestParam**: indicates that a method parameter should be bound to a web request parameter.

```
@RequestMapping(value="/addbook", method=RequestMethod.POST)
public String create(@RequestParam String title, @RequestParam String author) {
    Book book = new Book(title, author);
    bookRepository.save(book);
    return "redirect:viewbooks";
}
```

POST http://localhost:8090/addbook

```
<form action="addbook" method="POST">
...
<tr>
<td>Title:</td><td><input name="title" /></td>
</tr>
<tr>
<td>Author:</td><td><input name="author" /></td>
</tr>
...
<form>
```

Annotations

- **@RequestBody**: indicates that a method parameter should be bound to the body of the web request. The body of the request is passed through an HttpMessageConverter to resolve the method argument depending on the content type of the request. (e.g., JSON, XML)

```
@PostMapping(”books")
@ResponseStatus(HttpStatus.OK)
@ResponseBody
public Book updateBook(@RequestBody Book book)
{
     Book book = bookService.saveBook(book);
    ...
    return book;
}
```

POST http://localhost:9091/library/api/v1/books

```
{
    ”isbn”:”abcd-1234”,
    “title”:”Software Engineering
}
```

RequestBody: Book

```
public class Book {

    private String isbn;
    private String title;
    ...
}
```

Annotations

@ResponseBody: indicates a method return value should be bound to the web response body. No view!

```
@PostMapping(”books")
@ResponseBody
public Book updateBook(@RequestBody Book book)
{
    Book book = bookService.saveBook(book);
   ...
   return book;
}
```

Annotations

@ResponseStatus: Marks a method or exception class with the status code and reason that should be returned.

```
@PostMapping(”books")
@ResponseStatus(HttpStatus.OK)
@ResponseBody
public Book updateBook(@RequestBody Book book)
{
    Book book = bookService.saveBook(book);
   ...
   return book;
}
```

@ExceptionHandler - Handling Exceptions

- Many things can go wrong when developing applications
  - Validation errors
  - Database errors
  - Server errors
  - Client errors
  - etc.
- Communicate errors (source, type, etc) to end-users
- You can build a totally custom error response
- Redirect the user to a dedicated error view

Annotations

@ExceptionHandler: for handling exceptions in specific handler classes and/or handler methods (requires **@ControllerAdvice**)

```java
@ExceptionHandler(value = BookNotFoundException.class)
public String bookNotFound(Model model, BookNotFoundException ex) {
    model.addAttribute("error", "BookNotFoundException");
    model.addAttribute("status", Response.SC_NOT_FOUND);
    model.addAttribute("message", ex.getMessage());
    return "error";
}
```

error.jsp

```
<body>
<h1>Error Occured!</h1>
<b>[<span th:text="${status}">status</span>]
<span th:text="${error}">error</span>
</b>
<p th:text="${message}">message</p>
</body>
```

BookNotFoundException.java

```java
public class BookNotFoundException extends Exception {

    public BookNotFoundException(String customMessage) {
        super(customMessage);
    }
}
```

Model

- Controllers and Views share a Java object known as Model, ‘M’ in MVC.
- Can be of type Model or Map
- View uses the model to display dynamic data given by the controller based on expression language (EL)

```
//Controller
@RequestMapping(value="viewbook/{id}", method=RequestMethod.GET)
public String findOne(@PathVariable Long id, Model model) {
    Book book = bookRepository.findById(id).get();
    model.addAttribute("book", book);
    return "viewbook";
}
```

```
<%--View--%>

<html>
    <body>
        <h3>${book.title}</h3>
    </body>
</html>
```

@ModelAttribute

@ModelAttribute: binds a method parameter or method return value to a named model attribute, exposed to a web view.

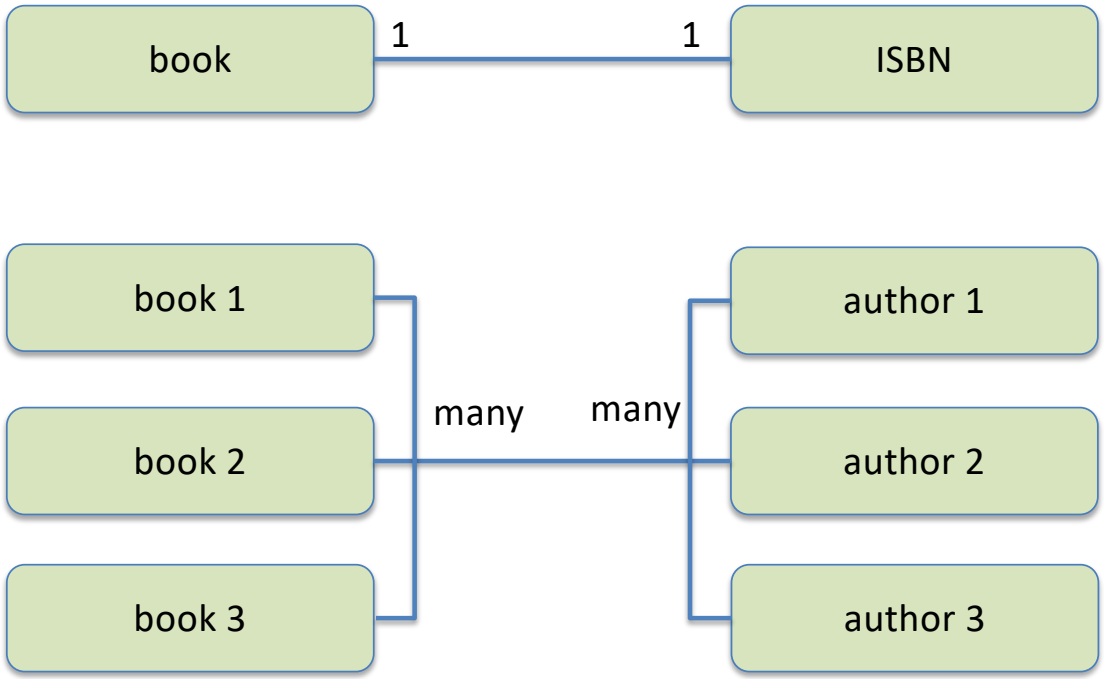
```
@PostMapping("/owners/{ownerId}/pets/{petId}/edit")
public String processSubmit(@ModelAttribute Pet pet) {
    // method logic...
}
```

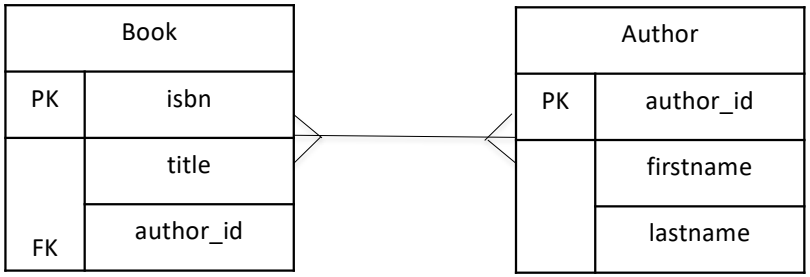
Library service (Revisited)

- Services (for now)
  - Add Book
  - Update Book
  - View Book
  - Delete Book
  - Add Author

CRUD operations

Model


book
1
1
ISBN
book 1
book 2
book 3
many
many
author 1
author 2
author 3


Book
PK
isbn
title
FK
author_id
Author
PK
author_id
firstname
lastname

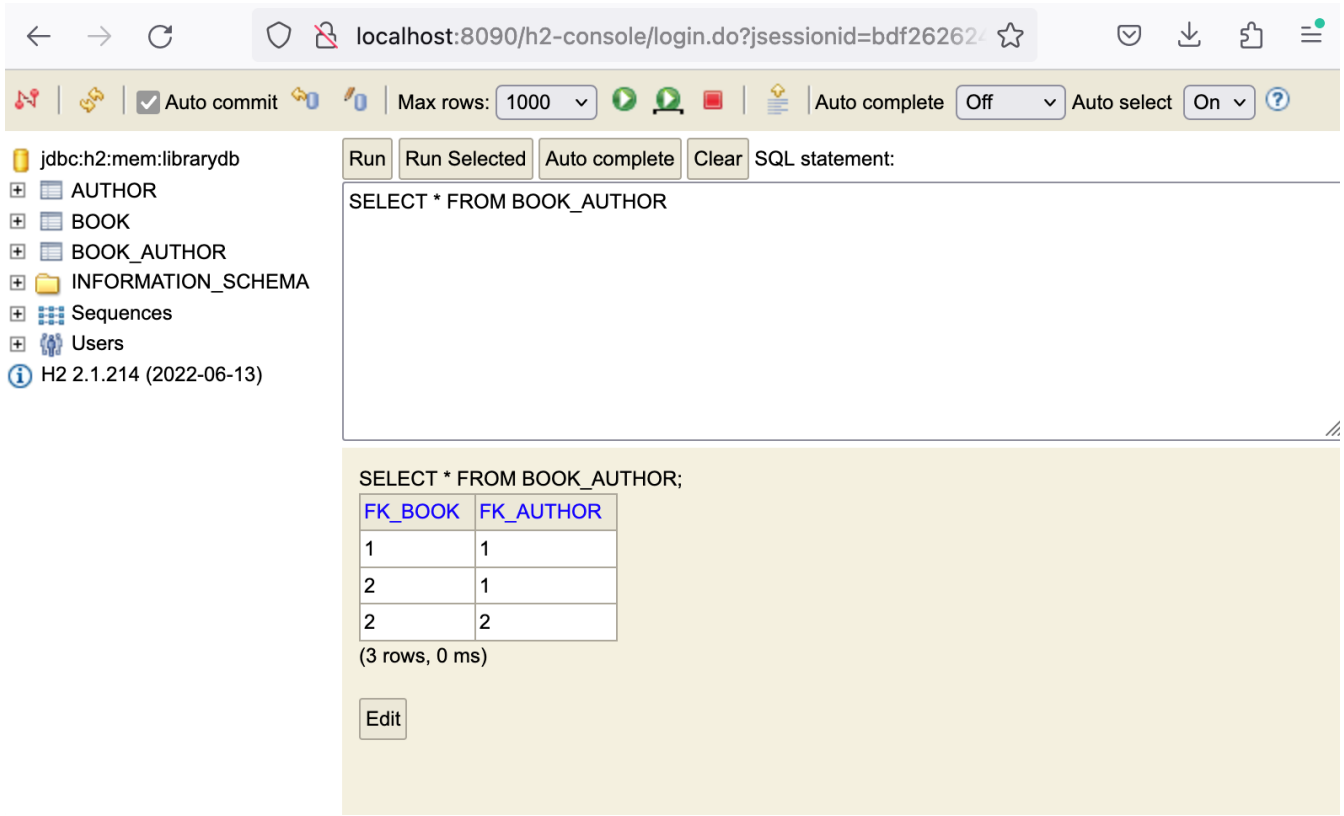
Database Layer

- H2 (in-memory) database
  - Can also point the db to a persistent disk file to avoid data loss after application restart
    - e.g., jdbc:h2:file:/data/librarydb

```
spring.datasource.url=jdbc:h2:mem:librarydb
spring.datasource.driverClassName=org.h2.Driver
spring.datasource.username=sa
spring.datasource.password=
```

Database Layer

- H2 Console: administer database




```
spring.h2.console.enabled=true
# default path: h2-console
```

Persistence Layer

- Spring Data, JPA (Java Persistence API): to access and manipulate data – CRUD
- JavaBean: Model, Data
- Java class can be turned into an entity Object
- Mapping between the Java instance variables and database tables and fields

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

```
spring.jpa.show-sql=true
spring.jpa.properties.hibernate.dialect=org.hibernate.dialect.H2Dialect
# Hibernate ddl auto (create, create-drop, validate, update)
spring.jpa.hibernate.ddl-auto=update
```

Persistence Layer

- JavaBean – Model, Data

```
public class Book {

   private long isbn;
   private String title;
   private int authorId;

   public String getIsbn() {
      return isbn;
   }

   ...
}
```

Persistence Layer

- We turn the Java class into an entity Object by marking it with @Entity annotation

```
@Entity
public class Book {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private long isbn;
    @Column(nullable = false, unique = true)
    private String title;
    @Column(nullable = false)
    private int authorId;

    ...
}
```

Persistence Layer

Mapping between the Java instance variables and database tables and fields

```
@Entity
public class Book {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private long isbn;
    @Column(nullable = false, unique = true)
    private String title;
    @Column(nullable = false)
    private int authorId;

    ...
}
```

– @Entity annotation indicates that the class is a persistent Java class.
– @Table annotation provides the table that maps this entity.
– @Id annotation is for the primary key.
– @GeneratedValue annotation is used to define generation strategy for the primary key.
– @Column annotation is used to define the column in database that maps annotated field.

Persistence Layer

Spring Data JPA creates the tables and constraints based on our ER specifications

```
Hibernate: create table author (author_id integer not null, firstname varchar(255)...)
Hibernate: create table book (id bigint not null, isbn varchar(255) not null,...)
Hibernate: create table book_author (fk_book bigint not null, fk_author...)
...
```

Spring Data - CRUD API

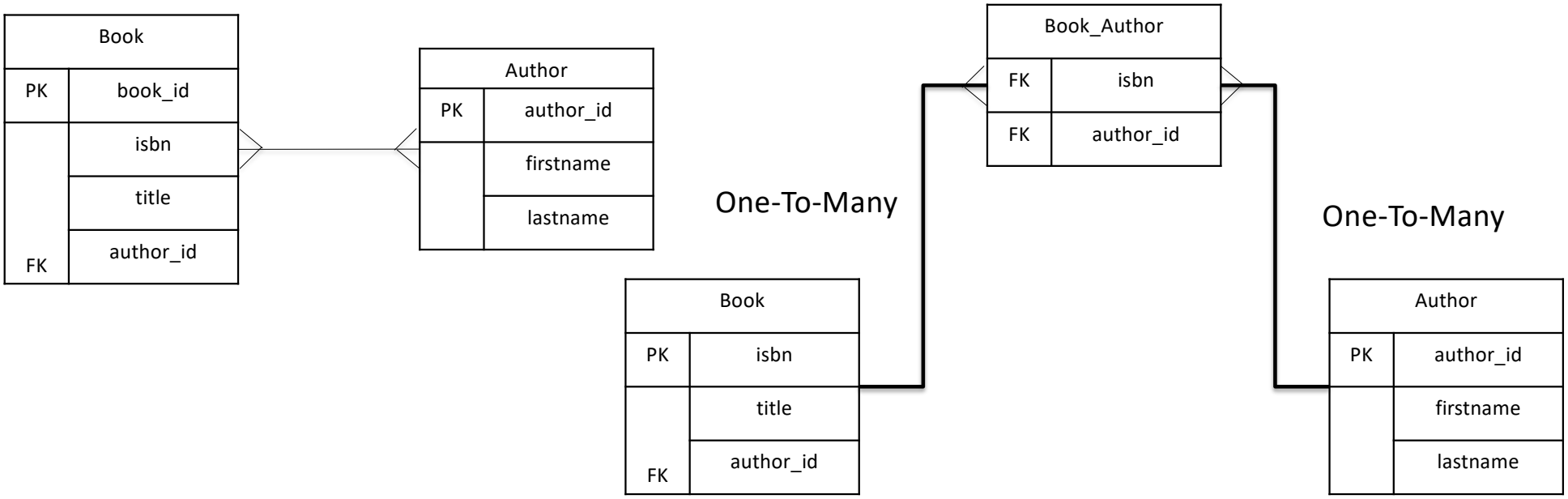
```
public interface CrudRepository<T, ID> extends Repository<T, ID> {
  <S extends T> S save(S entity);
  Optional<T> findById(ID primaryKey);
  Iterable<T> findAll();
  long count();
  void delete(T entity);
  boolean existsById(ID primaryKey);

  // ... more functionality omitted.
}
```

By extending the CrudRepository class, our object can then have access to the CRUD API operations

```
public interface BookRepository extends CrudRepository<Book, Long> {}
```

Persistence Layer

- Many-To-Many relationship
- Using JPA and Hibernate



Persistence Layer

- Many-To-Many
- Entities
  - Owning entity -> Book
    - @JoinTable
  - Inverse entity -> Author
    - mappedBy

```
@Entity
public class Book {
    ...
    @ManyToMany
    @JoinTable(name = "book_author",
    joinColumns = { @JoinColumn(name = "fk_book")},
    inverseJoinColumns = { @JoinColumn(name = "fk_author")})
    private Set<Author> authors = new HashSet<Author>();
    ...
}
```

```
@Entity
public class Author {
    ...
    @ManyToMany(mappedBy = "authors")
    private Set<Book> books = new HashSet<Book>();
    ...
}
```

Persistence Layer

- With JPA:
- Impose server-side constraints
- Can perform server-side validations on database fields

```
@Column(nullable=false, length=512, unique=true)

@Column(name="ORDER_COST", updatable=false, precision=12, scale=2)
```

Business Layer

- Here, we put our services
- Can be annotated with @Service
- Business logic
- Sits between the persistence layer and the controller
- Inject our JpaRepository
  - BookRepository
    - CRUD API

```
@Service
public class BookService {

    @Autowired
    private BookRepository bookRepository;

    public Book saveBook(Book book) {
        return bookRepository.save(book);
    }
    ...
}
```

Controller

- Handles the incoming requests, performs mapping, and delegate to the appropriate service method.

```
@Controller
public class BookController {

    @Autowired
    private BookService bookService;
    @Autowired
    private AuthorService authorService;

    @GetMapping("/")
    public String defaultView() {
        return "index";
    }

    @GetMapping("/viewbooks")
    public String findAll(Model model) {
        List<Book> books = (List<Book>) bookService.findAll();
        model.addAttribute("books", books);

        return "viewbooks";
    }
}
```

Presentation Layer

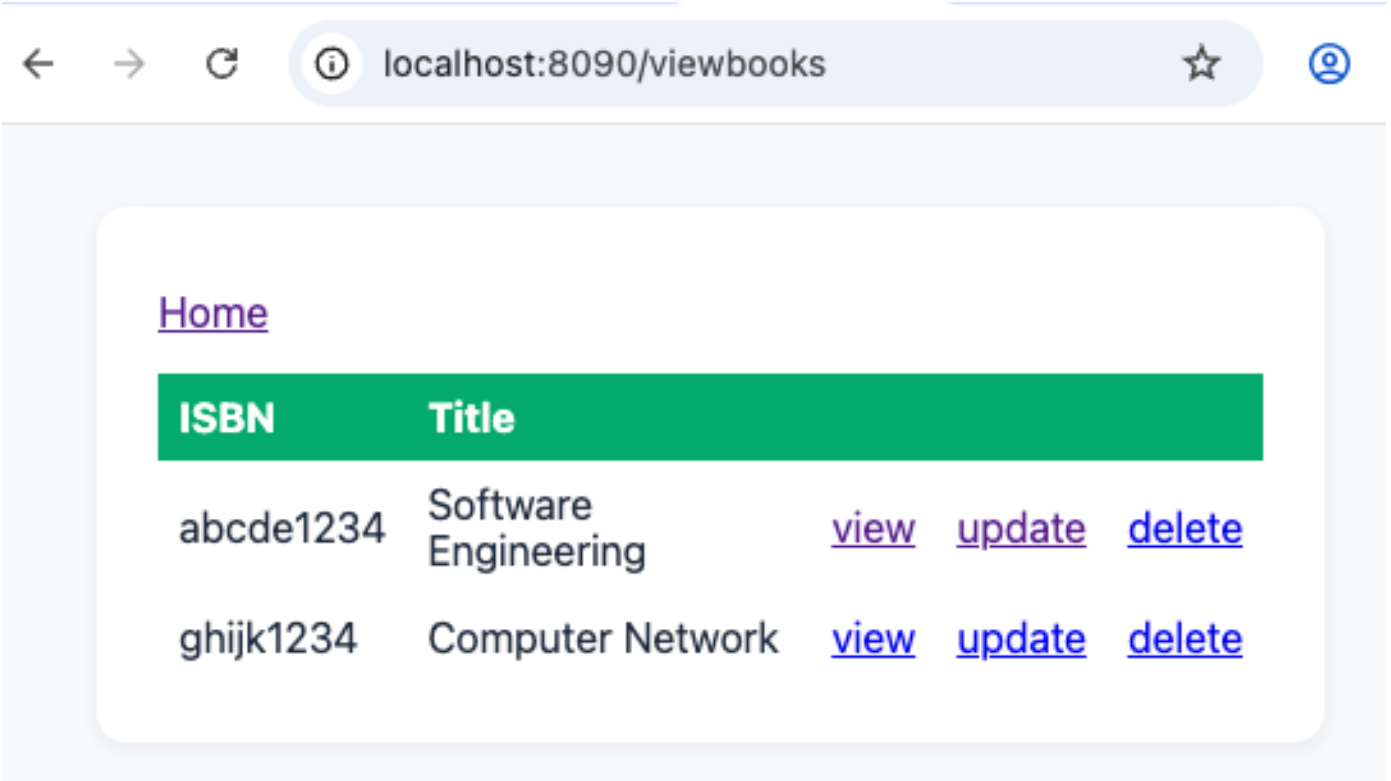
- Spring integrates with many view technologies
  - Thymeleaf template
  - JSP and JSTL
  - XML Marshalling
  - FreeMarker
  - …

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

```
<%@ page language="java" contentType="text/html; charset=UTF-8"
pageEncoding="UTF-8"%>
<%@taglib uri = "http://www.springframework.org/tags/form" prefix = "form"%>
```

Presentation Layer

View



Configurations

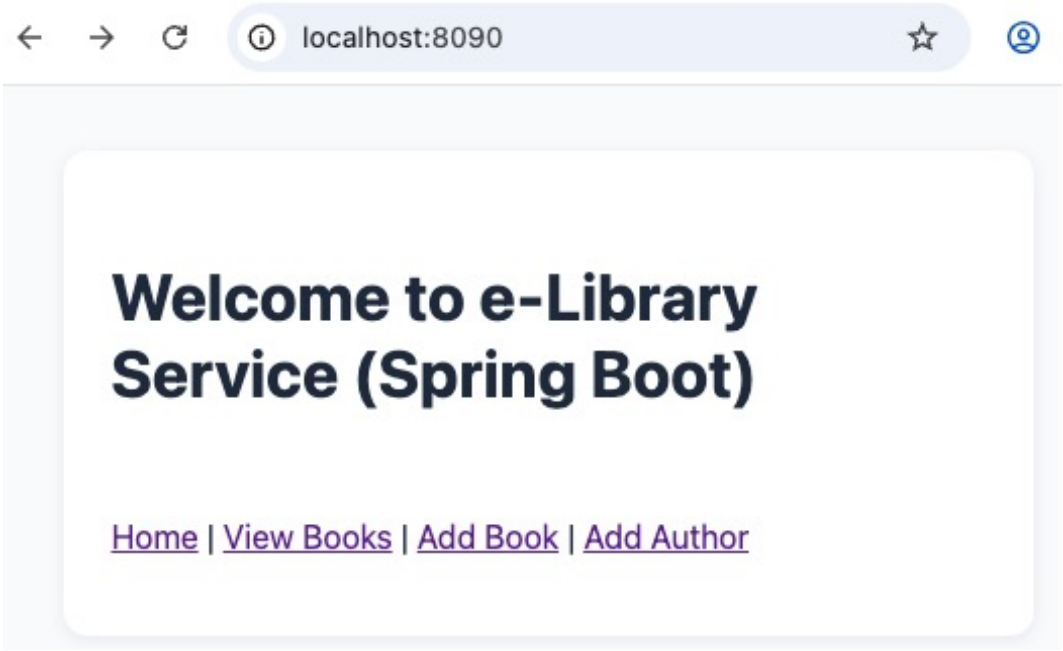
- Important Spring-boot configurations
  - Needs to scan project for beans, repos in different modules and classes
- @EnableJpaRepositories: package where your jpa classes
- @EntityScan: packages containing model
- @ComponentScan: include all other packages here (modules)

```
@SpringBootApplication
@EnableJpaRepositories("no.hvl.dat152.repository")
@EntityScan("no.hvl.dat152.model")
@ComponentScan(basePackages = {"no.hvl.dat152.service", "no.hvl.dat152.controller"})
@Configuration
```

Outlook

- Spring offers a comprehensive framework for web application development
  - Integrates with different technologies
  - Containerized with embedded server
  - Modular design
  - Loosely coupled – IoC/Dependency Injection
  - Ease of testability
- However
  - Complex with bundles of components
  - High learning curve
  - Parallel mechanisms to do the same thing (can be confusing)
  - Lots of configuration - XML (addressed with Spring boot)

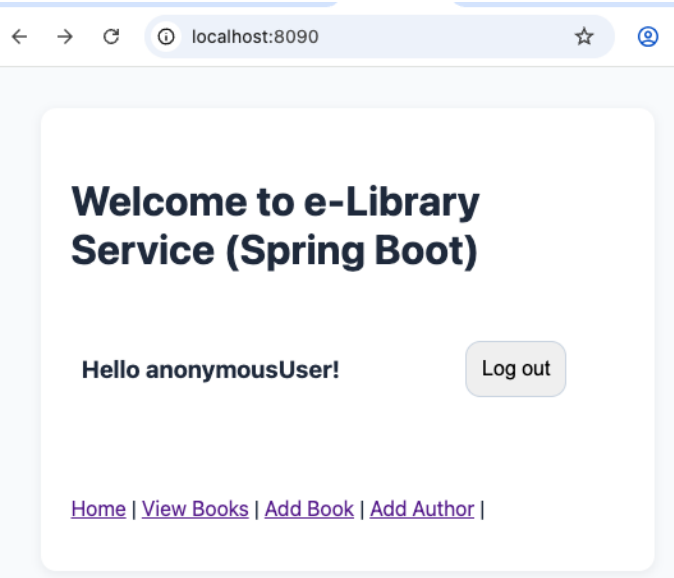
Lab - Spring Web MVC exercise - A



- New Features
  - Add Author
  - Delete Books

- Service Layer
  - BookService
  - AuthorService
- Controllers
  - BookController
  - AuthorController
- Presentation Layer (View)
  - addauthor.html/jsp
- ExceptionHandler
  - DeleteBookFailedException

Lab - Spring Web MVC exercise - B



- Presentation Layer (View)
  - login.html (custom login form)
- Controller
  - To handle GET request for login form display
- Security Configuration
  - Securityfilter:
    - formlogin
    - Logout
- Authorization
  - Display menu on view based on user roles
    - Index.html