University of St.Gallen

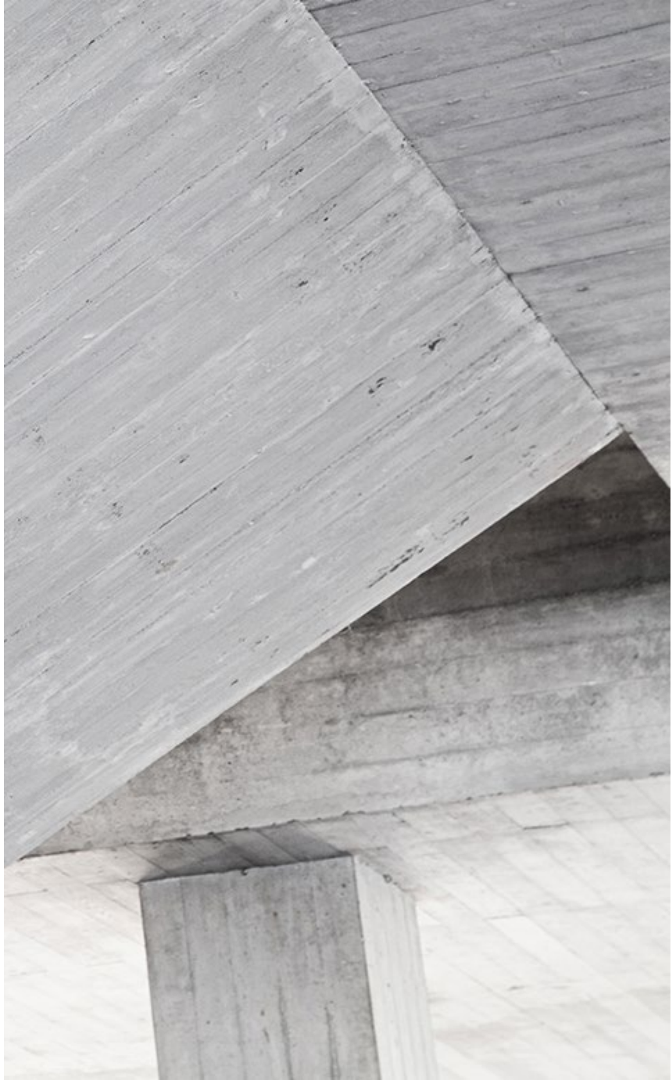
Institute of Computer Science

Entwurf von Softwaresystemen

Chapter 7: Frameworks

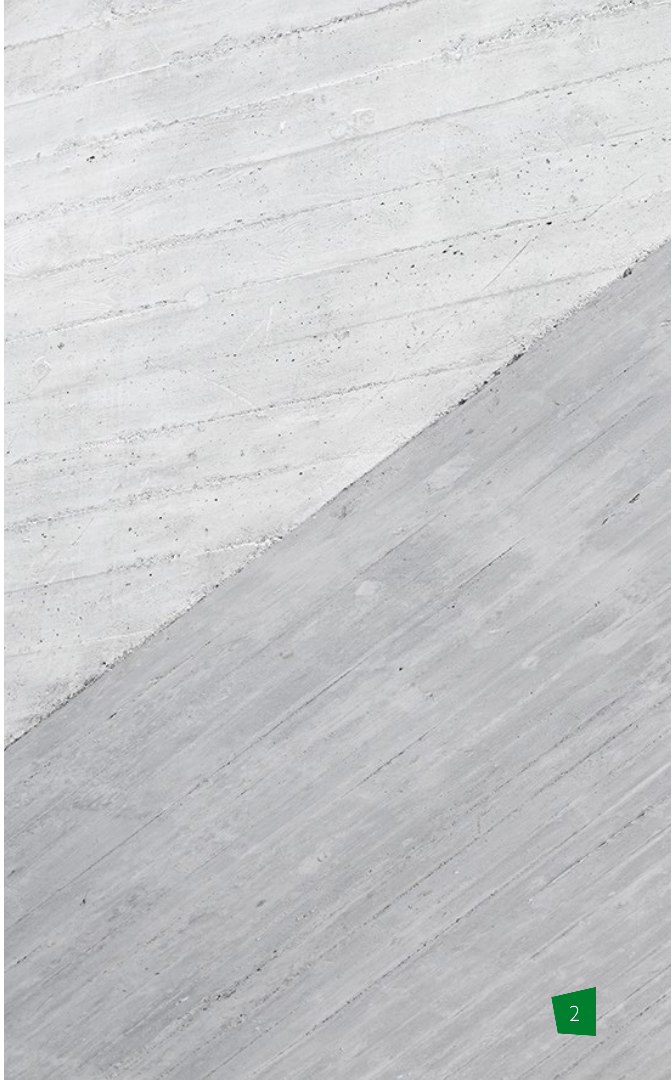
St.Gallen, 03. December 2025
Prof Dr. Ronny Seiger

From insight to impact.



Agenda

- Frameworks
- Principles and Patterns in Frameworks
- Spring & Spring Boot



Software Frameworks

- Allow you to reuse **design and implementation**
- Are a particular implementation technique for building families of software applications

- Represent a common design and partial implementation for the family:
  - A generic solution for a set of similar problems
  - Is incomplete by nature → application-specific functionality to be filled in by the *framework customizer* (developer of a concrete application)
  - Variations are specified by means of *hot spots* as extension points
  - Can be domain-specific (e.g., frameworks in financial or automotive applications) or general purpose (e.g., web application frameworks)

A **framework** is a set of cooperating classes that make up a reusable design for a specific class of software.

A **framework** provides a reusable, common structure to share among applications. Developers incorporate the framework into their own application and extend it to meet their specific needs. A framework defines a skeleton, and the application defines its own features to fill out the skeleton. The developer code is *called appropriately by the framework*.

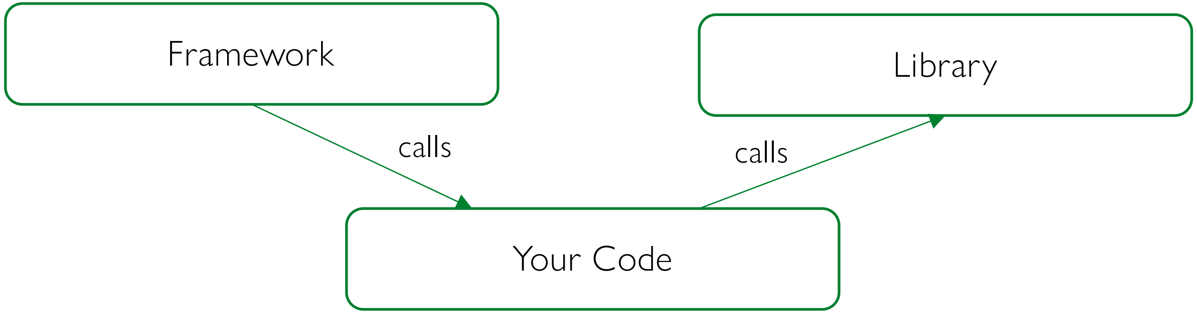
Library vs Framework

Library:

- A set of helper functions/objects/modules which your code calls for specific functionality
- Small scope, focus on reuse of functionality developed by others
- You call the library from your code. The library is not aware of your application.

Framework:

- The framework calls your code!



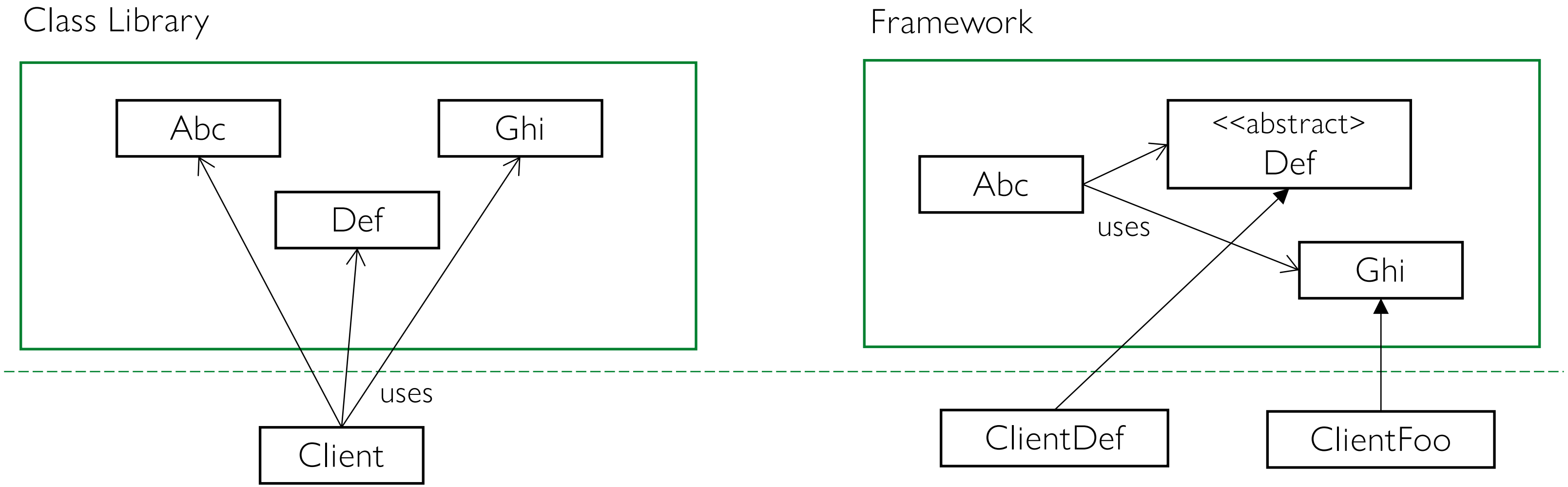
Inversion of Control (IoC)

Aka Hollywood principle: “Don’t call us, we’ll call you”.

When using a framework, the application-specific code written by the programmer gets called by the framework.

The framework takes control over coordination. It defines interaction patterns.

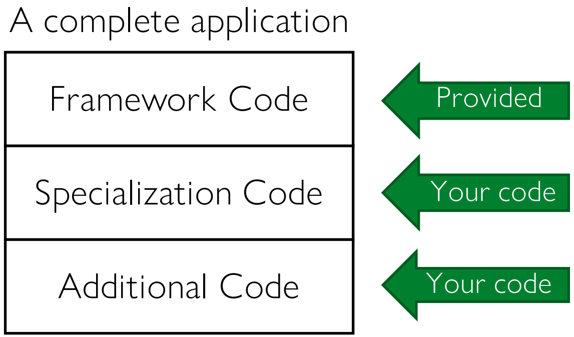
Using libraries, your code is the coordinator.



Object-oriented Frameworks

- An object-oriented class hierarchy + a built-in model of interaction
- Defines how objects derived from the class hierarchy interact with each other
- Building a custom application is typically done through specialization (extension) and composition of classes
- Core functionality is implemented as set of abstract classes that cooperate in a well-defined manner (closed for modification but open for extension)

- Building your own application using a framework:
  - Specialize the abstract classes by concrete subclasses
  - Use other concrete classes provided by the framework developer
  - Add new application-specific classes (+ drivers, utilities, libraries, etc.)



A complete application

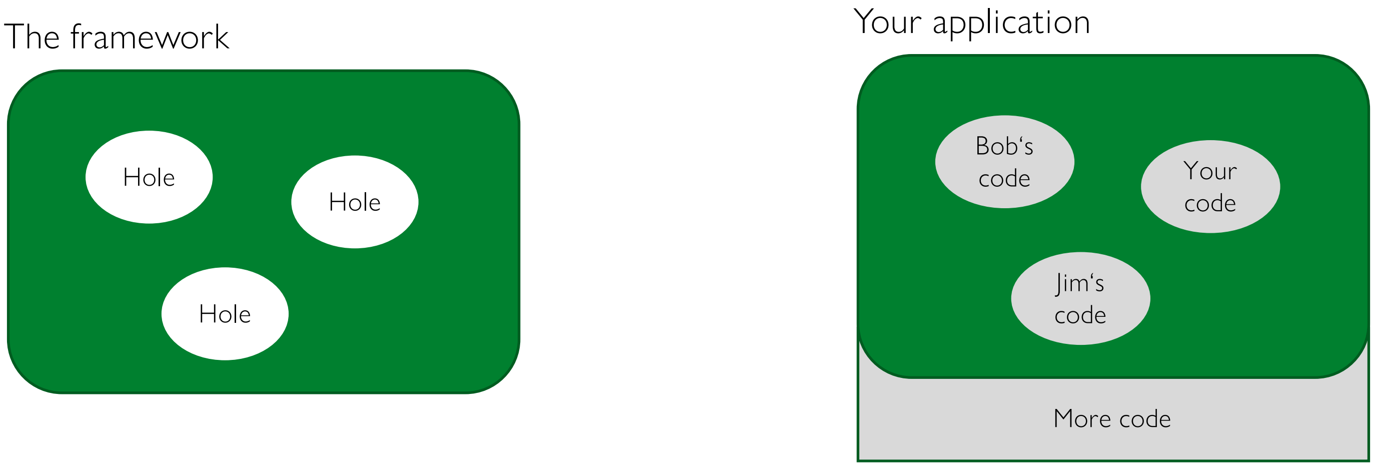
Framework Code

Specialization Code

Additional Code

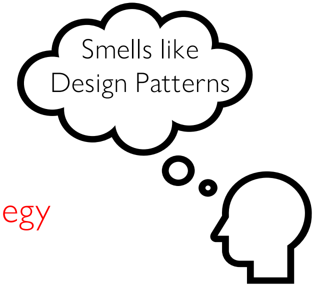
Development with Frameworks

A framework is a partial application → "Programming with Holes"



- Fixed points / features of a framework are called **Frozen Spots** (invariant methods; cannot easily be changed)
  - Define the overall architecture of the software system (basic components and their relationships)
- The Holes are called **Hot Spots** (variant methods, variation points, hooks or hook methods; allow for variability)

Hot Spots

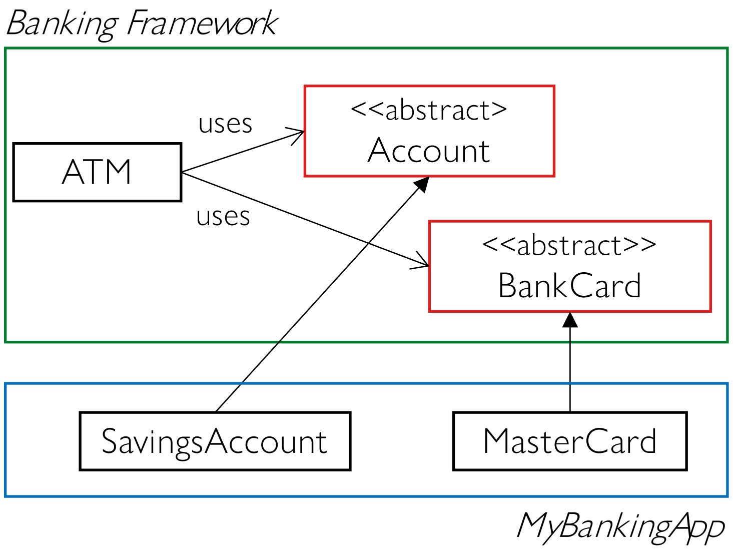


Template Method

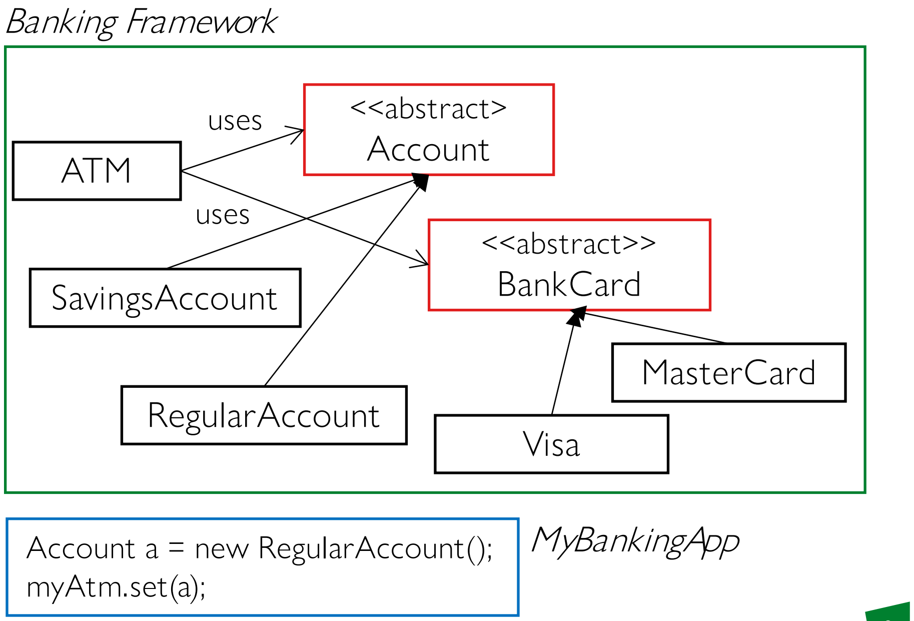
Basic design principle: “Separate code that changes from the code that doesn’t.”

Strategy

Achieved via **Inheritance**:
Filling in hotspots by specialising abstract classes, methods and interfaces



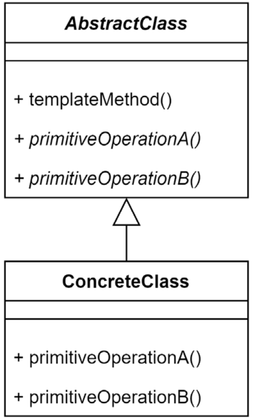
Achieved via **Composition**:
Filling in parameters or objects by prefabricated components



```
Account a = new RegularAccount();
myAtm.set(a);
```

The Template Method Pattern Revisited

Intent:
Defines the skeleton of an algorithm in an operation, deferring some steps to subclasses. Template Method lets subclasses redefine certain steps of the algorithm without changing the algorithm's structure.



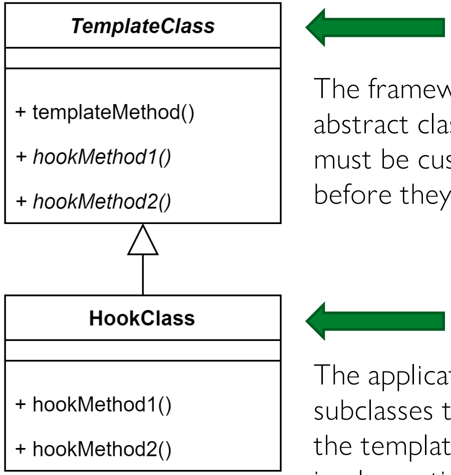
Solution:

- Break out primitive steps into separate methods in abstract class
- Construct method for the basic algorithm in abstract class that calls the primitive methods
- Override the primitive methods in child classes to implement specific tasks
- A primitive method in the abstract class may provide a default behavior that child classes may optionally override (called *Hook Methods*)

→ Inversion of Control ("Hollywood Principle")

Template Method in Frameworks

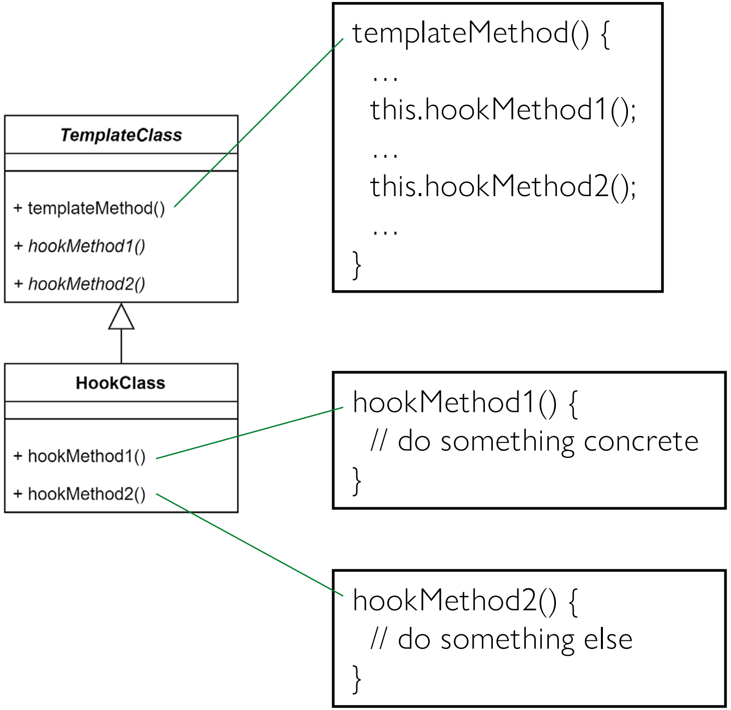
- Hotspots are often implemented via a Template Method:
  - The template method defines the skeleton of the hotspot
  - The variable parts are deferred to *hook methods* (can be abstract in the template class, or not)
  - The template method is defined in a *template class* which is part of the framework
  - The hook methods are defined in *hook classes*
    - Concrete subclasses of the template class that are provided by the framework user (to customize the framework)
  - → Inheritance as parametrisation mechanism
  - → Dynamic binding of hook method implementation via concrete subclass at runtime



The framework provides abstract classes that must be customized before they can be used

The application provides subclasses that customize the template methods by implementing the (abstract) hook methods

Template Method in Frameworks: Example



```
templateMethod() {
  …
  this.hookMethod1();
  …
  this.hookMethod2();
  …
}
```

```
hookMethod1() {
  // do something concrete
}
```

```
hookMethod2() {
  // do something else
}
```

Example from
Lecture 5

```
public abstract class HouseTemplate {

    public final void buildHouse(){
        buildFoundation();
        buildPillars();
        buildWalls();
        buildWindows();
        System.out.println("House is built.");
    }

    private void buildWindows() {
        System.out.println("Building Glass Windows");
    }

    public abstract void buildWalls();
    public abstract void buildPillars();

    private void buildFoundation() {
        System.out.println("Building foundation with
cement, iron rods and sand");
    }
```

Overwrites

```
public class WoodenHouse extends HouseTemplate {
@Override
    public void buildPillars() {
        System.out.println("Building Pillars with Wood coating");
    }
}
```

Framework Customization Approaches

White-box Frameworks

- Customization through inheritance
- Require insight in and access to implementation
- Relatively easy to design, but more difficult to learn
- More programming required for framework user
- More flexibility

Black-box Frameworks

- Customization through composition
- Require insight into provided components
- Harder to design, easier to learn
- Less programming required for framework user
- Limited flexibility (no unanticipated variations)

Grey-box Frameworks

- Most frameworks live in between white-box and black-box
- Try to realise benefits of both and to avoid limitations of both
- Frameworks often start as white-box and mature towards grey or even black-box

Using and Developing Frameworks

The framework (developer) must:

- Provide extensive domain knowledge and design
- Provide concrete, reliable, executable software
- Be sufficiently flexible to specialize for the required context
- Be usable and easy to learn

How to refactor towards a framework:

- Create template methods
- Optimize class hierachies
  - Refactor to specialize
  - Refactor to generalize
- Incorporate composition relationships

The application (developer) must:

- Keep the **contracts** of hotspots
- Understand and follow the interaction rules

→ Frameworks are rather difficult to learn, interaction patterns are often hidden, abstract classes might be difficult to understand

→ Steep learning curve; black-box is easier to learn than white-box

Design Patterns vs. Frameworks

- Both are ways of describing and documenting solutions to common problems.
- Design patterns are not frameworks, they are more abstract.
- Many patterns may be involved in the solution of one problem.

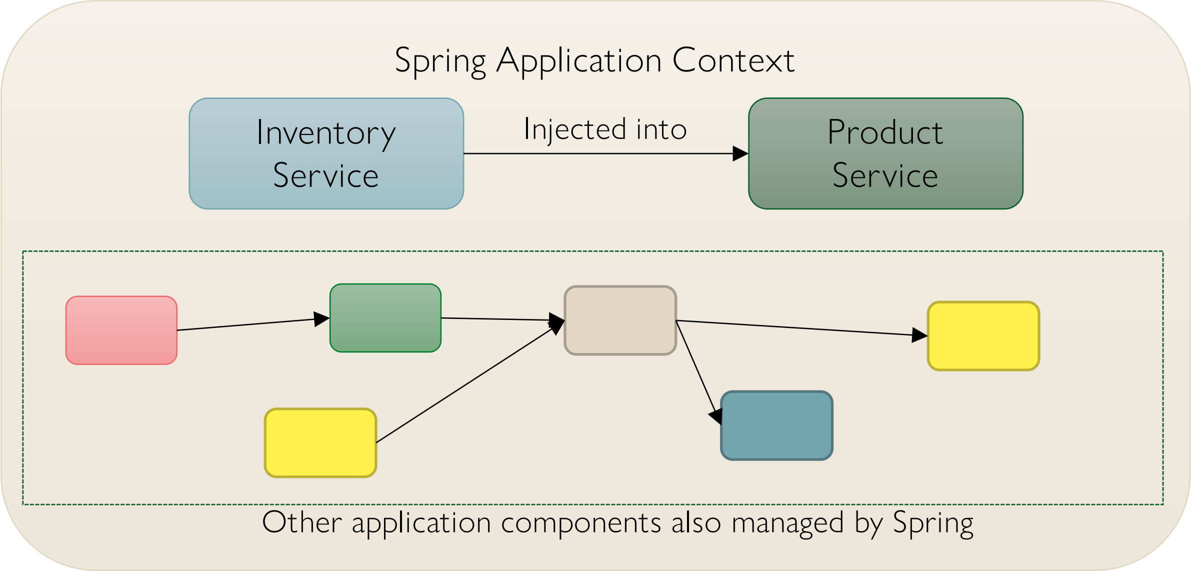
- Frameworks:
  - Design and code for solving a family of problems within a specific domain
  - Are instantiated by inheritance and composition of classes
  - Can contain several instances of multiple design patterns
  - Are more oriented towards immediate use

- Design Patterns are more abstract, smaller architectural elements, less specialized than frameworks.

Spring Framework

- Very popular framework for full-stack development of Java (+ web) applications
- Widely used in software industry
- Goal: make development of business application faster and more lightweight (compared to J2EE)

- Important principles:
  - Inversion of Control
  - Dependency Injection
  - Containers and Beans
  - Application Context



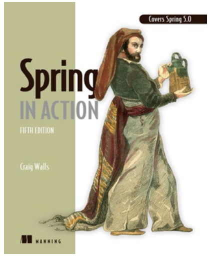
Spring Application Context

Inventory Service

Injected into

Product Service

Other application components also managed by Spring



Dependency Injection (DI)

- A technique (pattern) for inversion of control
- Being able to pass (inject) the dependencies of a class when required instead of initializing the dependencies inside of the recipient class
- Decouple the construction of your classes from the construction of your classes' dependencies
- Setter injection, Constructor injection, Method injection

Example of Constructor Injection:

```
class Main {
  public static void main(String... args) {
    /* Notice that we are creating ClassB first */
    ClassB classB = new ImprovedClassB();

    /* Constructor Injection */
    ClassA classA = new ClassA(classB);

    System.out.println("Ten Percent: " + classA.tenPercent());
  }
}
```

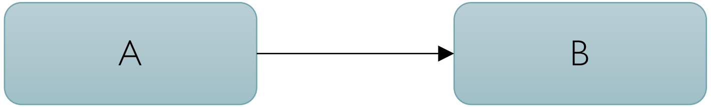
```
class ClassA {

  ClassB classB;

  /* Constructor Injection */
  ClassA(ClassB injected) {
    classB = injected;
  }

  int tenPercent() {
    return classB.calculate() * 0.1d;
  }
}
```

IoC Container

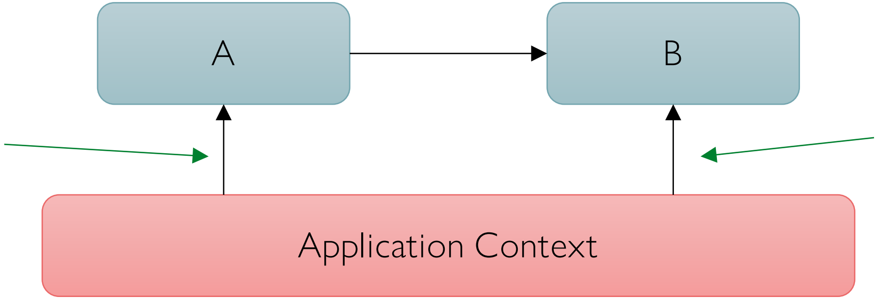
- One or more instances of a container are used to configure and wire together application and framework objects, and manage their lifecycles using IoC, dependency injection and reflection → application code does not have to be aware of the container (no dependencies)
- The basic IoC container in Spring is called **Bean Factory**.
- **Bean** in Spring*:* an object that is instantiated, assembled and otherwise managed by a Spring IoC container; beans are the backbone of your application.
- **Application Context** is a special, more advanced form of a Bean Factory.
  - Will be used automatically when you use annotations to declare beans and is the best choice.

Dependency Inversion with IoC Container

Traditional approach: dependencies inside the code



DI (IoC): objects know nothing about each other

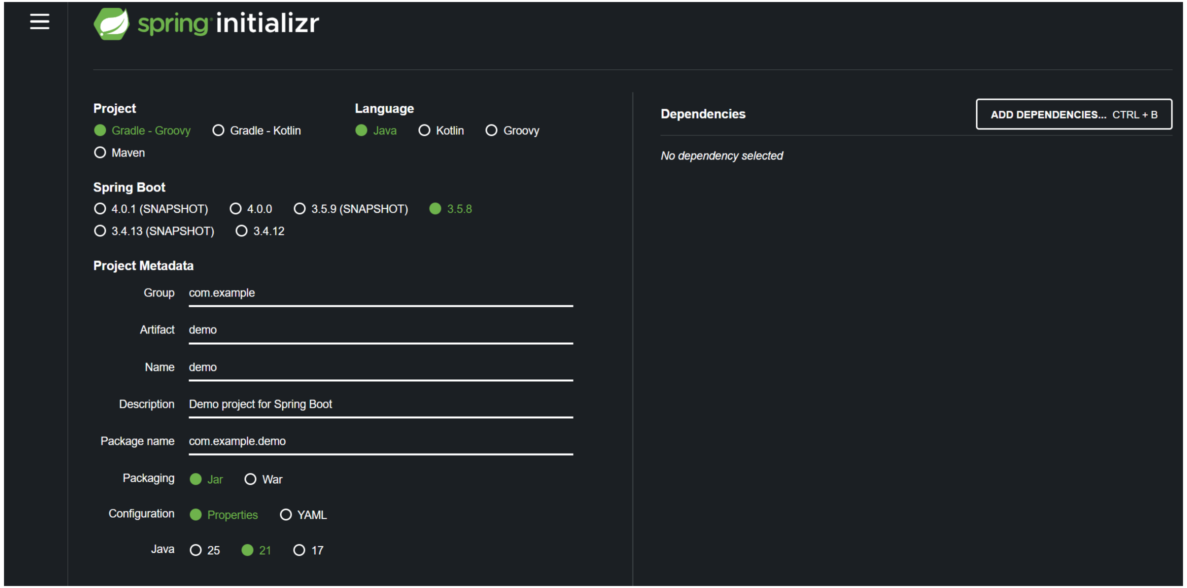


- Creates object A
- Initializes it and injects object B dependency

- Creates object B
- Initializes it

Hands-on Spring Initialzr (1)

Either via Web (+ open in IntelliJ) or integrated in IntelliJ (New Project, Spring Boot)



Project
● Gradle - Groovy ○ Gradle - Kotlin
○ Maven

Language
● Java ○ Kotlin ○ Groovy

Spring Boot
○ 4.0.1 (SNAPSHOT) ○ 4.0.0 ○ 3.5.9 (SNAPSHOT) ● 3.5.8
○ 3.4.13 (SNAPSHOT) ○ 3.4.12

Project Metadata

Group: com.example
Artifact: demo
Name: demo
Description: Demo project for Spring Boot
Package name: com.example.demo
Packaging: ● Jar ○ War
Configuration: ● Properties ○ YAML
Java: ○ 25 ● 21 ○ 17

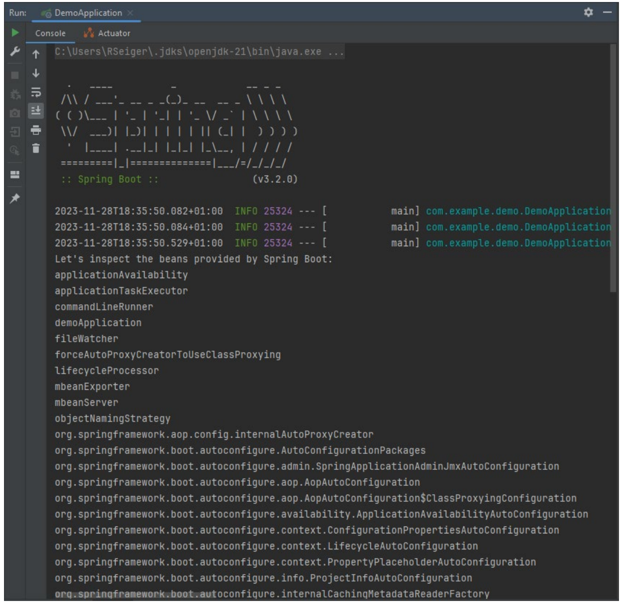
Dependencies

No dependency selected

Hands-on Spring Initialzr (2)

Copy & paste the following code below the *main()* method in the DemoApplication class:

```
@Bean
public CommandLineRunner commandLineRunner(ApplicationContext ctx) {
    return args -> {

        System.out.println("Let's inspect the beans provided by Spring Boot:");

        String[] beanNames = ctx.getBeanDefinitionNames();
        Arrays.sort(beanNames);
        for (String beanName : beanNames) {
            System.out.println(beanName);
        }

    };
}
```



```
C:\Users\RSeiger\.jdks\openjdk-21\bin\java.exe ...

  .   ____          _            __ _ _
 /\\ / ___'_ __ _ _(_)_ __  __ _ \ \ \ \
( ( )\___ | '_ | '_| | '_ \/ _` | \ \ \ \
 \\/  ___)| |_)| | | | | || (_| |  ) ) ) )
  '  |____| .__|_| |_|_| |_\__, | / / / /
 =========|_|==============|___/=/_/_/_/
 :: Spring Boot ::                (v3.2.0)

2023-11-28T18:35:50.082+01:00  INFO 25324 --- [           main] com.example.demo.DemoApplication
2023-11-28T18:35:50.084+01:00  INFO 25324 --- [           main] com.example.demo.DemoApplication
2023-11-28T18:35:50.529+01:00  INFO 25324 --- [           main] com.example.demo.DemoApplication
Let's inspect the beans provided by Spring Boot:
applicationAvailability
applicationTaskExecutor
commandLineRunner
demoApplication
fileWatcher
forceAutoProxyCreatorToUseClassProxying
lifecycleProcessor
mbeanExporter
mbeanServer
objectNamingStrategy
org.springframework.aop.config.internalAutoProxyCreator
org.springframework.boot.autoconfigure.AutoConfigurationPackages
org.springframework.boot.autoconfigure.admin.SpringApplicationAdminJmxAutoConfiguration
org.springframework.boot.autoconfigure.aop.AopAutoConfiguration
org.springframework.boot.autoconfigure.aop.AopAutoConfiguration$ClassProxyingConfiguration
org.springframework.boot.autoconfigure.availability.ApplicationAvailabilityAutoConfiguration
org.springframework.boot.autoconfigure.context.ConfigurationPropertiesAutoConfiguration
org.springframework.boot.autoconfigure.context.LifecycleAutoConfiguration
org.springframework.boot.autoconfigure.context.PropertyPlaceholderAutoConfiguration
org.springframework.boot.autoconfigure.info.ProjectInfoAutoConfiguration
org.springframework.boot.autoconfigure.internalCachingMetadataReaderFactory
```

Resolve the missing imports and click “Run”.

Some Annotations in Spring

@Bean: used to declare a method as bean

@Component: used to declare custom classes as beans; Spring will scan them automatically and inject any specific dependency into them/inject them wherever needed

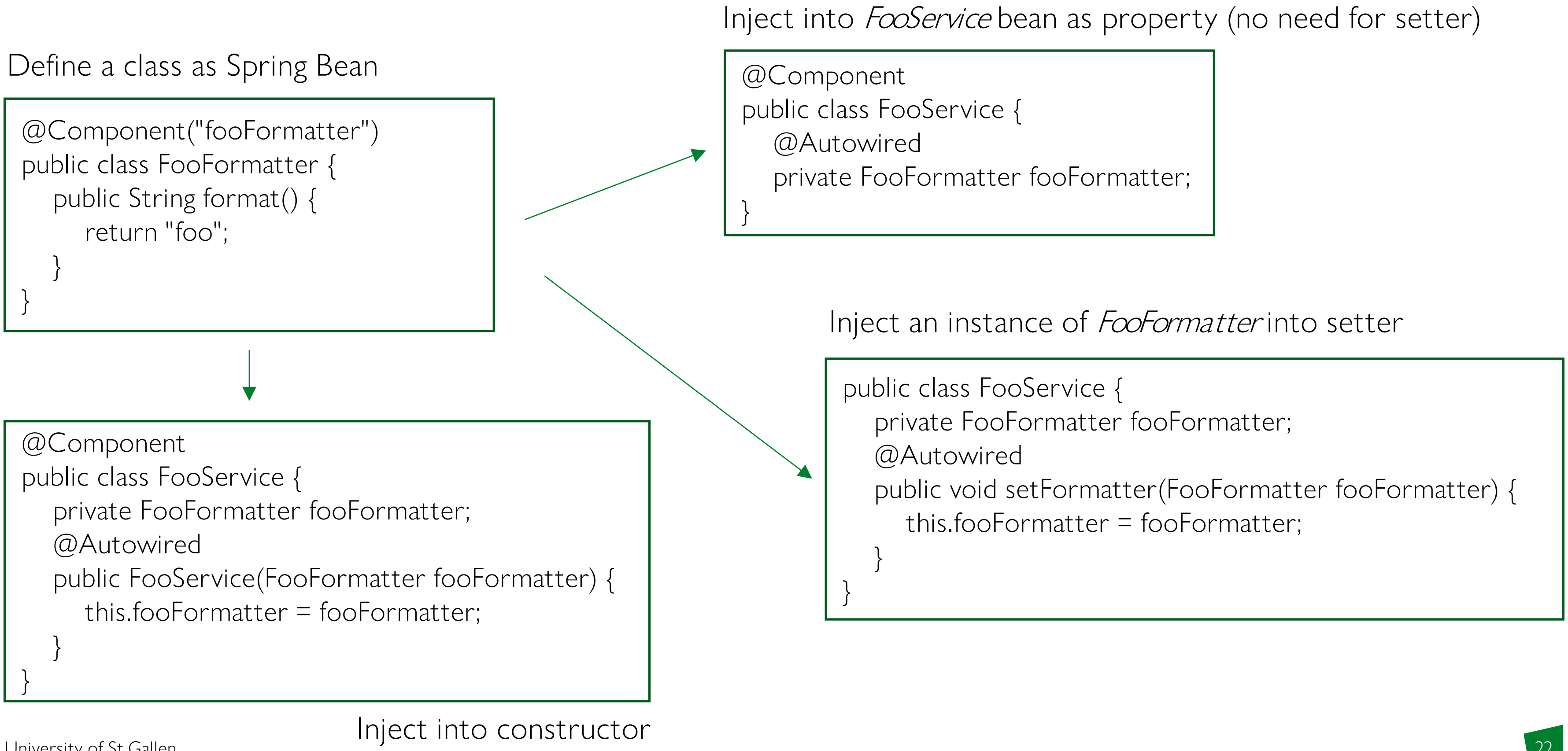
Specialized forms of @Component:

- @Controller (@RestController) for controller layer
- @Service for services/business logic
- @Repository for persistence

@SpringBootApplication = @Configuration (tag class as source of bean definitions for application context) + @EnableAutoconfiguration (tell Spring to add beans automatically, based on class path) + @ComponentScan (tell Spring to look for other components, configurations, services in the project package(s))

@Autowired: explicitly inject beans on properties, constructors or setters

@AutoWired Examples



Define a class as Spring Bean

```
@Component("fooFormatter")
public class FooFormatter {
    public String format() {
        return "foo";
    }
}
```

Inject into *FooService* bean as property (no need for setter)

```
@Component
public class FooService {
    @Autowired
    private FooFormatter fooFormatter;
}
```

Inject an instance of *FooFormatter* into setter

```
public class FooService {
    private FooFormatter fooFormatter;
    @Autowired
    public void setFormatter(FooFormatter fooFormatter) {
        this.fooFormatter = fooFormatter;
    }
}
```

```
@Component
public class FooService {
    private FooFormatter fooFormatter;
    @Autowired
    public FooService(FooFormatter fooFormatter) {
        this.fooFormatter = fooFormatter;
    }
}
```

Inject into constructor

@AutoWired Hands-on

1. Create the two new classes in the “demo“ package:

```
@Component("fooFormatter")
public class FooFormatter {
    public String format() {
        return "foo";
    }
}
```

```
@Component
public class FooService {
    private FooFormatter fooFormatter;
    @Autowired
    public FooService(FooFormatter fooFormatter) {
        this.fooFormatter = fooFormatter;
        System.out.println(fooFormatter.format());
    }
}
```

2. Add the new property to the class DemoApplication:

```
private FooService fooService;
```

3. Add the new method to the class DemoApplication:

```
@Autowired
public void callFooService(FooService fooService) {
}
```

4. Call the new method from the commandLineRunner method:

```
callFooService(fooService);
```

Spring Boot

- Spring Boot was already used in the demo examples; based on Spring under the hood
- An extension of Spring, eliminating much of the boilerplate and configuration that characterizes Spring
- Provides a pre-configured platform for building Spring-based application with minimal annotation and configuration effort
- Used to create standalone web applications (with REST APIs), embedded web servers are available → easy to deploy, easy to launch
- Automatic configurations of the various components (makes integration, e.g., of databases very easy)

- Drawback: generates a large number of unused dependencies (becomes resource intensive)

REST-based Interfaces in Spring

- In lecture 8, Vaadin took care of the communication between the web-based UI and the Java application.
- If you want to use a different UI framework (e.g., JavaScript-based) you will add some REST-based interfaces to make your Spring Boot application act as a web service. You communicate via its web API.

- In Spring Boot, the @RestController annotation lets you define web APIs very easily.

1. Add as new class to the demo application:

```
@RestController
public class HelloController {

        @GetMapping("/")
        public String index() {
                return "Greetings from Spring Boot!";
        }

}
```

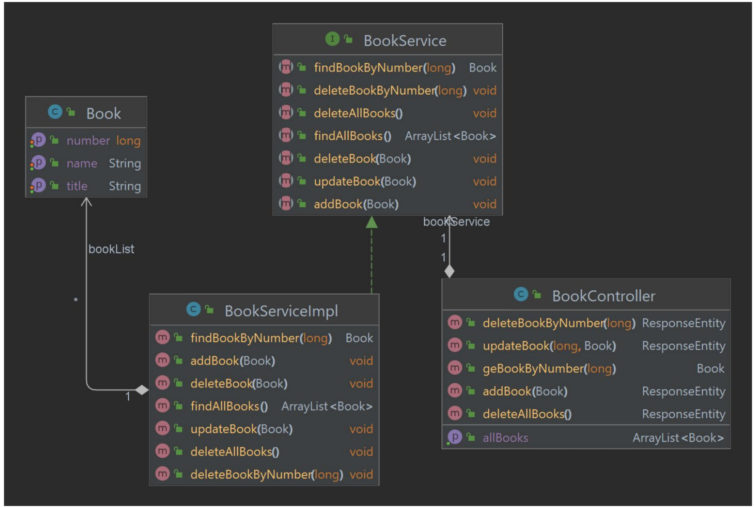
2. Add as new dependency to pom.xml:

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

3. Start and open web browser or use PostMan to open: http://localhost:8080/

Simple Web Application in Spring

- Check the “book-example” project in Canvas.



Book

number	long
name	String
title	String

BookService

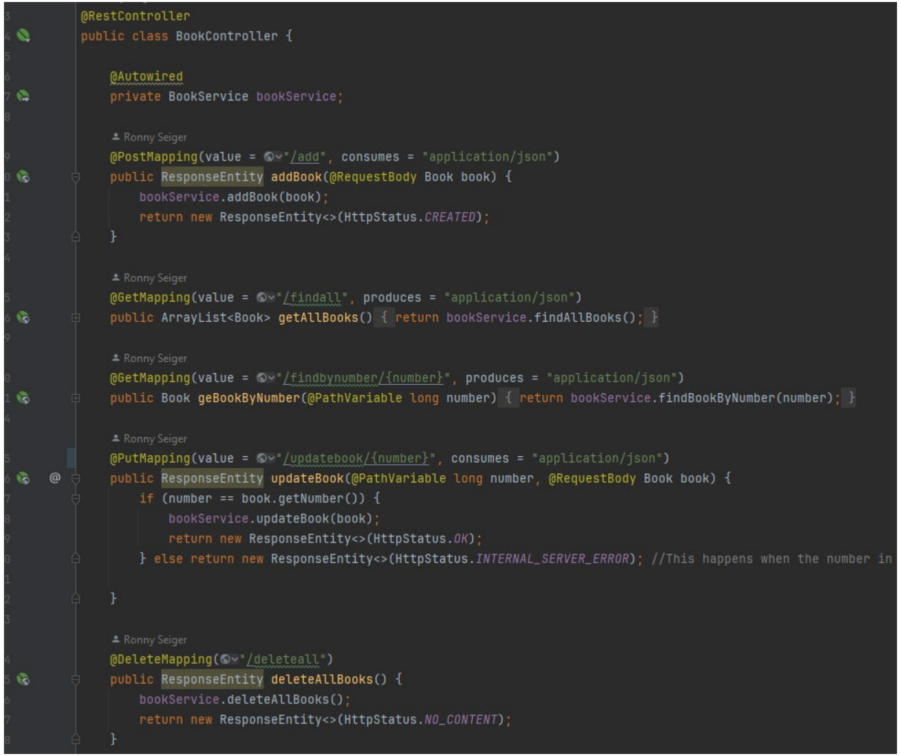
findBookByNumber(long)	Book
deleteBookByNumber(long)	void
deleteAllBooks()	void
findAllBooks()	ArrayList<Book>
deleteBook(Book)	void
updateBook(Book)	void
addBook(Book)	void

BookServiceImpl

findBookByNumber(long)	Book
addBook(Book)	void
deleteBook(Book)	void
findAllBooks()	ArrayList<Book>
updateBook(Book)	void
deleteAllBooks()	void
deleteBookByNumber(long)	void

BookController

deleteBookByNumber(long)	ResponseEntity
updateBook(long, Book)	ResponseEntity
geBookByNumber(long)	Book
addBook(Book)	ResponseEntity
deleteAllBooks()	ResponseEntity
allBooks	ArrayList<Book>



```
@RestController
public class BookController {

    @Autowired
    private BookService bookService;

    Ronny Seiger
    @PostMapping(value = "/add", consumes = "application/json")
    public ResponseEntity addBook(@RequestBody Book book) {
        bookService.addBook(book);
        return new ResponseEntity<>(HttpStatus.CREATED);
    }

    Ronny Seiger
    @GetMapping(value = "/findall", produces = "application/json")
    public ArrayList<Book> getAllBooks() { return bookService.findAllBooks(); }

    Ronny Seiger
    @GetMapping(value = "/findbynumber/{number}", produces = "application/json")
    public Book geBookByNumber(@PathVariable long number) { return bookService.findBookByNumber(number); }

    Ronny Seiger
    @PutMapping(value = "/updatebook/{number}", consumes = "application/json")
    public ResponseEntity updateBook(@PathVariable long number, @RequestBody Book book) {
        if (number == book.getNumber()) {
            bookService.updateBook(book);
            return new ResponseEntity<>(HttpStatus.OK);
        } else return new ResponseEntity<>(HttpStatus.INTERNAL_SERVER_ERROR); //This happens when the number in

    }

    Ronny Seiger
    @DeleteMapping("/deleteall")
    public ResponseEntity deleteAllBooks() {
        bookService.deleteAllBooks();
        return new ResponseEntity<>(HttpStatus.NO_CONTENT);
    }
```

Brief Excursion: Project Lombok

Tired of Writing/Generating Getters/Setters and Constructors?

- Check the class “AdvancedBook”.
- Project Lombok generates getters, setters and different forms of constructors for you.
- @Getters and @Setters annotation
- @NoArgsConstructor: empty constructor (without arguments)
- @AllArgsConstructor: constructor with all arguments
- @RequiredArgsConstructor: constructor for all required fields (i.e., final and @NonNull)

```
<dependency>
        <groupId>org.projectlombok</groupId>
        <artifactId>lombok</artifactId>
        <version>1.18.42</version>
</dependency>
```

Spring Data JPA

- JPA Data Access Abstraction
  - Can be used with Hibernate and other JPA providers (e.g., Eclipse Link)
  - Can be used to implement custom DAO patterns/solutions
  - Reduces amount of boilerplate code, generates queries from method names

Add persistence to book example project:

1.) add dependencies

```
<dependency>
   <groupId>org.springframework.boot</groupId>
   <artifactId>spring-boot-starter-data-jpa</artifactId>
</dependency>
<dependency>
    <groupId>com.h2database</groupId>
    <artifactId>h2</artifactId>
    <version>2.4.240</version>
</dependency>
```

2.) add to resources/application.properties

```
spring.datasource.url=jdbc:h2:mem:db;DB_CLOSE_DELAY=-1
spring.datasource.username=sa
spring.datasource.password=sa
spring.h2.console.enabled=true
spring.jpa.show-sql=true
```

Starts a simple web frontend for the DB at:
http://localhost:8080/h2-console
(Adjust the db url to: jdbc:h2:mem:db)

Spring Data JPA: Example (cont'd.)

3.) Annotate your model class(es) as entities

```
@Entity
public class Book {

    @Id
    @GeneratedValue
    private long id;
…
```

4.) Define the repository interface

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

  // CRUD methods are inherited from CrudRepository
  Book findByNumber(long number);
  Book findByName(String name);
}
```

No implementation required;
Spring tries to match the attributes

5.2) Use the repository in the BookServiceImpl class

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

```
@Override
    public void addBook(Book b) {
        bookList.add(b);
        bookRepository.save(b);
    }
```

```
@Override
public Book findBookByNumber(long number) {
    return bookRepository.findByNumber(number);
}
```

5.1) Declare the BookServiceImpl class as @Transactional

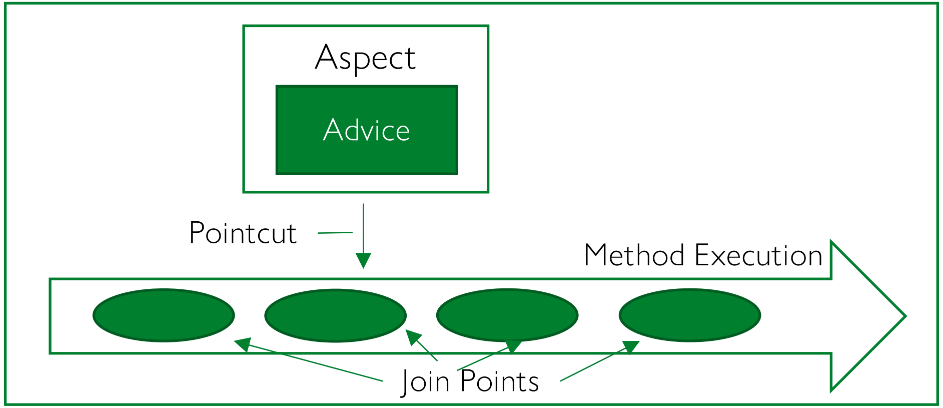
Aspect-oriented Programming

- @Transactional: used for transactions; a rollback happens for runtime unchecked exceptions (default config)
- Annotation of bean, either at a class or method level

- Example of Aspect-oriented Programming (AoP): separate cross-cutting concerns from business code
- Concerns that may affect code in many objects (may lead to a lot of code duplication)

- Typical AoP examples: logging, security, caching

- Join Points: define points where you can insert additional logic using AoP
- Advice: code that is executed at a Join Point
- Pointcut: predicate or expression that matches Join Points
- Aspect: combination of advices and pointcuts encapsulated in a class
- Weaving: process of inserting aspects into an application code at the appropriate point
- Target: the *advised* object

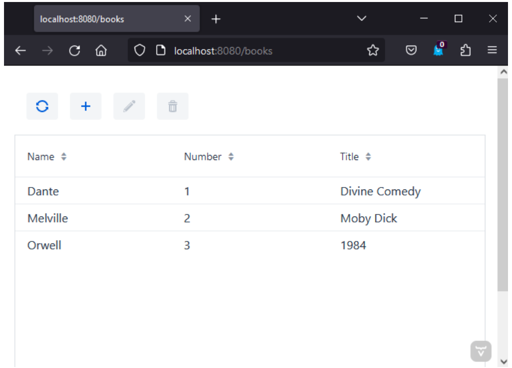


Spring Boot with Vaadin

- Vaadin is well integrated with Spring Boot. You need to add the Vaadin repositories and dependencies to the Maven build file and then can get started with Vaadin in Spring Boot.

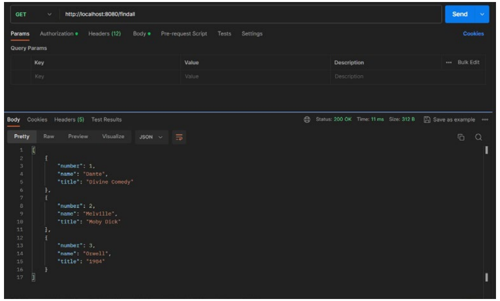
```
@Route("books")
public class MainView extends VerticalLayout {

    @Autowired
    public MainView(BookService bookService) {

        GridCrud<Book> bookGridCrud = new GridCrud<>(Book.class);
        bookGridCrud.setCrudListener(new BookCrudListener(bookService));

        add(bookGridCrud);

    }
}
```

Vaadin and the REST Controller interact with the same book service



localhost:8080/books

Name	Number	Title
Dante	1	Divine Comedy
Melville	2	Moby Dick
Orwell	3	1984



http://localhost:8080/findall

Status: 200 OK Time: 11 ms Size: 312 B

```
[
    {
        "number": 1,
        "name": "Dante",
        "title": "Divine Comedy"
    },
    {
        "number": 2,
        "name": "Melville",
        "title": "Moby Dick"
    },
    {
        "number": 3,
        "name": "Orwell",
        "title": "1984"
    }
]
```

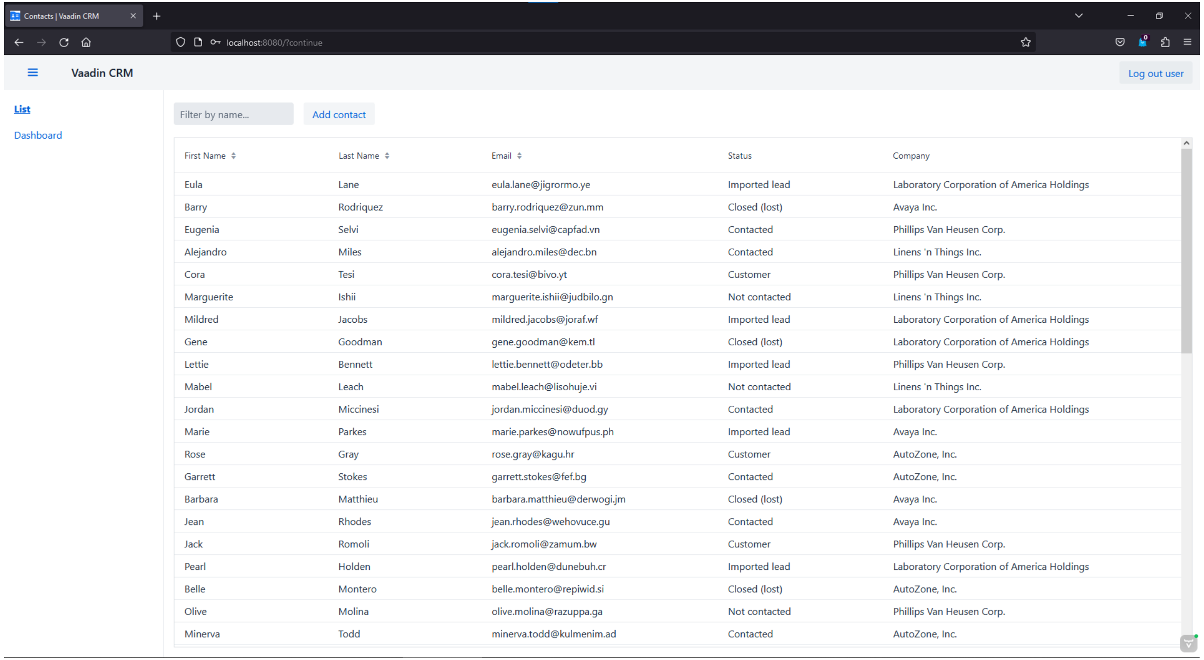
Spring Boot with Vaadin and JPA

Nice book and tutorial: https://github.com/vaadin/flow-crm-tutorial

vaadin}>

Building Modern Web Apps with Spring Boot and Vaadin

A practical introduction to web application development using Java.



First Name	Last Name	Email	Status	Company
Eula	Lane	eula.lane@jigrormo.ye	Imported lead	Laboratory Corporation of America Holdings
Barry	Rodriquez	barry.rodriquez@zun.mm	Closed (lost)	Avaya Inc.
Eugenia	Selvi	eugenia.selvi@capfad.vn	Contacted	Phillips Van Heusen Corp.
Alejandro	Miles	alejandro.miles@dec.bn	Contacted	Linens 'n Things Inc.
Cora	Tesi	cora.tesi@bivo.yt	Customer	Phillips Van Heusen Corp.
Marguerite	Ishii	marguerite.ishii@judbilo.gn	Not contacted	Linens 'n Things Inc.
Mildred	Jacobs	mildred.jacobs@joraf.wf	Imported lead	Laboratory Corporation of America Holdings
Gene	Goodman	gene.goodman@kem.tl	Closed (lost)	Laboratory Corporation of America Holdings
Lettie	Bennett	lettie.bennett@odeter.bb	Imported lead	Phillips Van Heusen Corp.
Mabel	Leach	mabel.leach@lisohuje.vi	Not contacted	Linens 'n Things Inc.
Jordan	Miccinesi	jordan.miccinesi@duod.gy	Contacted	Laboratory Corporation of America Holdings
Marie	Parkes	marie.parkes@nowufpus.ph	Imported lead	Avaya Inc.
Rose	Gray	rose.gray@kagu.hr	Customer	AutoZone, Inc.
Garrett	Stokes	garrett.stokes@fef.bg	Contacted	AutoZone, Inc.
Barbara	Matthieu	barbara.matthieu@derwogi.jm	Closed (lost)	Avaya Inc.
Jean	Rhodes	jean.rhodes@wehovuce.gu	Contacted	Avaya Inc.
Jack	Romoli	jack.romoli@zamum.bw	Customer	Phillips Van Heusen Corp.
Pearl	Holden	pearl.holden@dunebuh.cr	Imported lead	Laboratory Corporation of America Holdings
Belle	Montero	belle.montero@repiwid.si	Closed (lost)	AutoZone, Inc.
Olive	Molina	olive.molina@razuppa.ga	Not contacted	Phillips Van Heusen Corp.
Minerva	Todd	minerva.todd@kulmenim.ad	Contacted	AutoZone, Inc.

Software Project lite for Assignment 6

- Develop a complete Spring Boot service providing CRUD operations and some business logic for at least four different types of entities (classes) with one-to-one and one-to-many associations among themselves.
- You are free to choose the application domain yourselves.
- The service should have a Vaadin-based UI, a REST-based Web API, unit tests, and persistence for all entities (with Spring Data JPA and an H2 db).

- Document your project.
- Work in groups of 1 to 2 people. Sign up to Assignment 6 groups via Canvas.
- Align your project ideas / domains with the tutors in the exercise sessions.
- Last exercise session (Dec. 19) completely online via Zoom (link will be on Canvas)
  - 1st part: Project presentations (voluntary)
  - 2nd part: Exam Q&A
- Deadline for project hand-in is 09.01.2026

Suggested package structure (follows the book example):

- controller
- model
- persistence
- service
- ../test/java

Book Example Hands-on

10 mins

- Check the book-example-complete project from Canvas

→ Add new Books via PostMan and check the Web-UI

→ Add new Books via Web-UI and send requests via PostMan

→ Add/modify entries via Web-UI or PostMan and retrieve data from DB with SQL at:

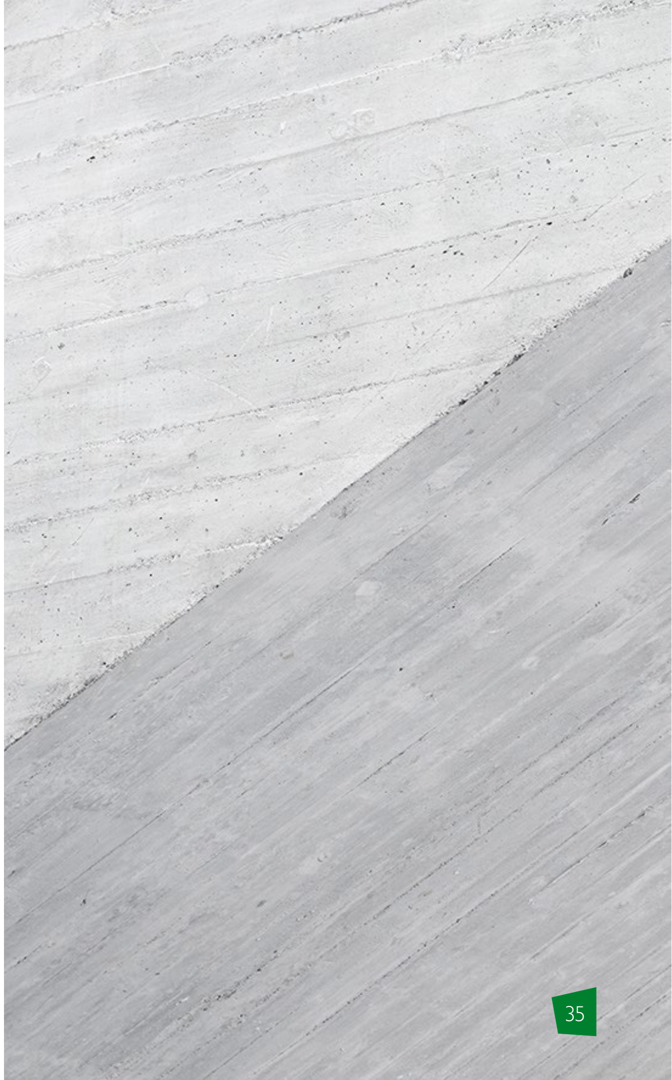
http://localhost:8080/h2-console

Recap & Outlook

- Frameworks
- Spring and Spring Boot

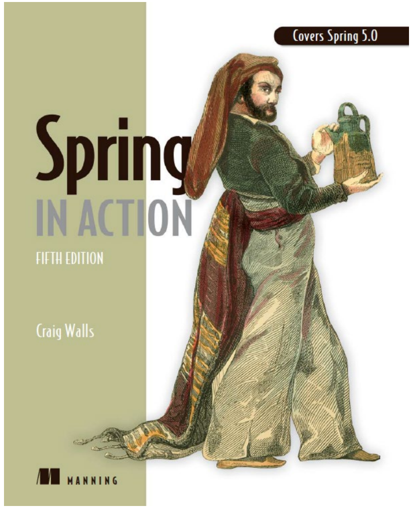
- Exercises on Friday & Assignment 6:
  - Full Stack Service with Spring Boot

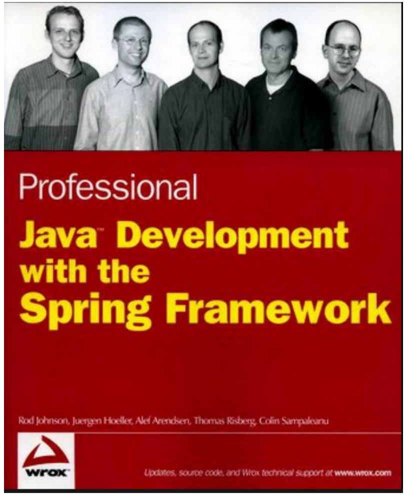
- Next week:
  - Architectural Patterns



Questions?
Comments?

References


Covers Spring 5.0
Spring
IN ACTION
FIFTH EDITION
Craig Walls
MANNING


Professional
Java™ Development with the Spring Framework
Rod Johnson, Juergen Hoeller, Alef Arendsen, Thomas Risberg, Colin Sampaleanu
wrox
Updates, source code, and Wrox technical support at www.wrox.com

Xpert.press
Manfred Broy
Marco Kuhrmann
Einführung in die Softwaretechnik
Springer Vieweg