Spring

> *"Spring helps development teams everywhere to build simple, portable, fast and flexible JVM-based systems and applications."*

Why Spring?

- Writing nicer object oriented software
- Enterprise ready
  - Transactions
  - Security
- Used by many organizations
- Easy to use
- Alternative for Java EE

Agenda

- Spring introduction
- Dependency injection
- Spring Boot
- Bean Validation
- Database configuration
- Data access
- Transactions
- JPA

Agenda continued:

- MVC
- REST
- AOP
- External values
- Caching
- Asynchronous
- Scheduling
- JMS
- Security
- Conclusion

Rules of engagement

- Training hours
- Lunches
- Phones
- Training material
- Evaluation
- It's your course!

Have fun!

InfoSupport
Solid Innovator

Spring introduction

InfoSupport
Solid Innovator

History

- 2003: Rod Johnson released a Java EE book. The book addresses some problems with Java EE and a framework to solve those issues: Spring. It's mainly aimed as a means for inversion of control (dependency injection)
- 2004: Spring release 1.0
- 2009: Spring becomes part of VMWare
- 2013: Spring was transferred to Pivotal Software a joint venture between VMWare and EMC Corporation
- 2014: first release of Spring Boot
- 2017: Spring 5.0 focuses on functional and reactive

The Spring Platform

- Is a collection of some 20 up modules
- Modules are grouped along technologies:
  - Core Container
  - Data Access / integration
  - Dependency injection
  - AOP (Aspect Oriented Programming)
  - Transaction management
  - Security
  - Messaging
  - Test

Spring Tool Suite (STS)

- Eclipse based IDE
- Possible to install as a plugin in Eclipse or standalone
- You can pick whatever IDE you want

Spring vs Java EE

- Spring can be seen as an alternative to Java EE
- Although Spring is proprietary, the development of the platform seems uninterrupted
- Spring bundles it's dependencies
- Spring is evolving faster

Dependency injection

Why dependency injection?

- Let someone else manage object instances and lifecycle
- To follow object oriented principles
  - Single responsibility principle
  - Decoupling
- Makes it easier to manage dependencies between objects
- Software with dependency injection is easier to test
- Less boilerplate code like factories to create instances

Plain Java example

```
public class UserController {
    private UserRepository userRepository = new UserRepository();
}
```

- The UserController needs to 'know' (instantiate) the UserRepository
- It's not the UserController's responsibility from an object oriented perspective

Plain Java example

```
public class UserController {
    private UserRepository userRepository = new UserRepository();
}
```

Spring dependency injection example

```
public class UserController {
    @Autowired
    private UserRepository userRepository;
}
```

Each Spring application has an ApplicationContext

- The ApplicationContext is also called the Spring container
- It is an Inversion of Control (IoC) container taking care of Dependency Injection (DI)
- Configuration metadata can be supplied in 3 ways
  - Defined in a XML (less popular nowadays)
  - Defined with Java code
  - Defined with annotations

Configuration metadata

- Our application consists of:
  - Business objects (POJO's)
  - Configuration metadata
- The Spring container is created based on those
- The Spring container produces a running application

POJO

> *"a plain old Java object (POJO) is an ordinary Java object, not bound by any special restriction and not requiring any class path"*

- Spring bean is a POJO, optionally implements an interface
- Must have a no arg-constructor

Example POJO

```
public class POJO {
    private String property;

    public String getProperty() {
        return property;
    }

    public void setProperty(String property) {
        this.property = property;
    }
}
```

XML configuration

The ApplicationContext

- Defined in a XML file for instance application-config.xml
- Should be on the classpath (in src/main/resources) of the application

```
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:context="http://www.springframework.org/schema/context"
    xsi:schemaLocation="http://www.springframework.org/schema/bea
    http://www.springframework.org/schema/beans/spring-beans.xsd
    http://www.springframework.org/schema/context
    http://www.springframework.org/schema/context/spring-context.

    ...
</beans>
```

Spring beans

- Any Java class configured in applicationContext.xml
- A Spring bean is singleton by default

```
<bean id="printer" class="lab1.ConsolePrinter" />
```

- *printer* is the unique identifier
- *lab1.ConsolePrinter* is the fully qualified classname

Spring bean example

```
public class ConsolePrinter implements PrinterService {
    public void print(String message) {
        System.out.println(message);
    }
}
```

Starting a Spring container

- Just for stand-alone applications

```
ApplicationContext applicationContext =
    new ClassPathXmlApplicationContext("application-config.xml");

// If ConsolePrinter has a unique implementation.
ConsolePrinter printer =
    applicationContext.getBean(ConsolePrinter.class);

// Retrieves the "printer" bean of the type ConsolePrinter.
ConsolePrinter printer2 =
    applicationContext.getBean("printer", ConsolePrinter.class);

// Retrieves the "printer" bean, needs a type cast.
ConsolePrinter printer3 =
    (ConsolePrinter) applicationContext.getBean("printer");
```

InfoSupport Solid Innovator

Dependency Injection

- Only define a dependency, don't instantiate or lookup dependencies
- Spring wires dependencies
  - Constructor injection
  - Setter injection
- Dependencies are injected at bean creation time

InfoSupport Solid Innovator

Constructor Injection

```
public class ConstructorDI {
    private PrinterService printerService;

    public ConstructorDI(PrinterService printerService) {
        this.printerService = printerService;
    }
}
```

```
<bean id="printerService" class="lab1.ConsolePrinter" />

<bean id="constructorDI" class="lab1.ConstructorDI">
    <constructor-arg ref="printerService" />
</bean>
```

Constructor Injection

- Arguments are resolved by type
  - argument order cannot be determined

```
public ConstructorDI(PrinterService printerService,
        OtherDependency otherDependency) {
    this.printerService = printerService;
    this.otherDependency = otherDependency;
}
```

```
<bean id="printerService" class="lab1.ConsolePrinter" />

<bean id="constructorDI" class="lab1.ConstructorDI">
    <constructor-arg ref="printerService"/>
    <constructor-arg ref="otherDependency"/>
</bean>
```

Constructor type ambiguity

- Remove ambiguity using explicit types.

```
public ConstructorDI(String myString, int myInt) {
    this.myString = myString;
    this.myInt = myInt;
}
```

```
<bean id="constructorDI" class="lab1.ConstructorDI">
    <constructor-arg type="java.lang.String" value="Hello" />
    <constructor-arg type="int" value="1" />
</bean>
```

Constructor type ambiguity

- Remove ambiguity using explicit types.

```
public ConstructorDI(String myString, int myInt) {
    this.myString = myString;
    this.myInt = myInt;
}
```

```
<bean id="constructorDI" class="lab1.ConstructorDI">
    <constructor-arg index="0" value="Hello" />
    <constructor-arg index="1" value="1" />
</bean>
```

Setter injection

```
public class SetterDI {

  private PrinterService printerService;

  public void setPrinterService(PrinterService printerService) {
    this.printerService = printerService;
  }
}
```

```
<bean id="printerService" class="lab1.ConsolePrinter" />

<bean id="setterDI" class="lab1.SetterDI" />
    <property name="printerService" ref="printerService"/>
</bean>
```

AutoWiring

- Wire dependencies automatically
- Don't specify dependencies using or
- Less XML configuration
- Easier to add or remove dependencies
- Can be turned on globally or per bean

AutoWiring Types

- no
  - No autowiring (default)
- byName
  - Property name must match bean id
- byType
  - Property type must match **exactly one** bean with a default constructor.
- constructor
  - Injection by type using a constructor. All arguments must be resolvable.

Autowiring byType

```
<bean id="printerService" class="lab1.ConsolePrinter"/>

<bean id="autoWireDI" class="lab1.AutoWireDI" autowire="byType"/>
```

```
public class AutoWireDI {

    private PrinterService printerService;

    public void setPrinter(PrinterService printerService) {
        this.printerService = printerService;
    }
}
```

Autowiring constructor

```
<bean id="printerService" class="lab1.ConsolePrinter"/>

<bean id="autoWireDI" class="lab1.AutoWireDI" autowire="construct
```

```
public class AutoWireDI {

    private PrinterService printerService;

    public AutoWireDI(PrinterService printerService) {
        this.printerService = printerService;
    }
}
```

Autowiring byName

```
<bean id="printerService" class="lab1.ConsolePrinter"/>

<bean id="autowiringDI" class="lab1.AutoWireDI" autowire="byName"
```

```
public class AutoWireDI {

    private PrinterService printerService;

    public void setPrinterService(PrinterService printerService)
        this.printerService = printerService;
    }
}
```

- Bean id "printerService" should match "setPrinterService"

Global AutoWiring

```
<beans ... default-autowire="byType" />
```

Autowire-candidate

- When multiple implementations of an interface exist Spring can't autowire
- You have to choose which implementation to inject
- Don't use for injection

```
<bean id="otherDependency" class="lab1.OtherDependency"
  autowire-candidate="false"/>
```

- Always use for injection

```
<bean id="otherDependency" class="lab1.OtherDependency"
  primary="true"/>
```

Life-cycle callbacks

```
public class LifecycleCallbacks {
    public void init() {
        System.out.println("Creating bean");
    }

    public void destroy() {
        System.out.println("Destroying bean");
    }
}
```

```
<bean id="lifecycleCallbacks" class="lab1.LifecycleCallbacks"
    init-method="init" destroy-method="destroy"/>
```

Java configuration

Spring beans

```
@Configuration
public class TestConfig {

    @Bean
    public PrinterService printerService(){
        return new ConsolePrinter();
    }
}
```

- This creates a Spring bean implementing the PrinterService interface.
- A Spring bean is a Singleton by default.

Getting a reference to a Spring bean

```
@Configuration
public class TestConfig {
    @Bean
    public PrinterService printerService(){
        return new ConsolePrinter();
    }
}
```

```
try (AnnotationConfigApplicationContext context =
      new AnnotationConfigApplicationContext(TestConfig.class)) {
    PrinterService printerService1 =
        context.getBean(PrinterService.class);
    printerService1.print("Hello context");

    PrinterService printerService2 =
        context.getBean("printerService",PrinterService.class);
    printerService2.print("Hello context");

    PrinterService printerService3 =
        (PrinterService)context.getBean("printerService");
    printerService3.print("Hello context");
}
```

Constructor Injection

```
public class ConstructorDI {
    private PrinterService printerService;

    public ConstructorDI(PrinterService printerService){
        this.printerService=printerService;
    }
}
```

```
@Configuration
public class TestConfig {
    @Bean
    public PrinterService printerService(){
        return new ConsolePrinter();
    }

    @Bean
    public ConstructorDI constructorDI(
            PrinterService printerService){
        return new ConstructorDI(printerService);
    }
}
```

InfoSupport Solid Innovator

Constructor Injection

- Arguments are resolved by type

```
@Configuration
public class TestConfig {
    @Bean
    public PrinterService printerService(){
        return new ConsolePrinter();
    }
    @Bean
    public OtherDependency otherDependency(){
        return new OtherDependency();
    }

    @Bean
    public ConstructorDI constructorDI(PrinterService ps,
            OtherDependency otherDependency){
        return new ConstructorDI(ps, otherDependency);
    }
```

InfoSupport Solid Innovator

Setter Injection

```
public class SetterDI {
  private PrinterService printerService;

  public void setPrinterService(PrinterService printerService) {
    this.printerService = printerService;
  }
}
```

```
@Configuration
public class TestConfig {
    @Bean
    public PrinterService printerService(){
        return new ConsolePrinter();
    }

    @Bean
    public SetterDI setterDI(PrinterService printerService){
        SetterDI setterDI = new SetterDI();
        setterDI.setPrinterService(printerService);
        return setterDI;
    }
}
```


AutoWiring

- Wire dependencies automatically
- Use the container to collect the dependencies

```
public class AutoWireDI {
    @Autowired
    private PrinterService printerService;

    public void sayHello(){
        printerService.print("Hello from autowired bean");
    }
}
```

Factory method

- A Factory allows an object to create new objects without having to know the details. The object doesn't know how the other objects are created or what the dependencies of those objects are.

Factory Method

```
public class FactoryMethod{
    private PrinterService printer;
    private String value;

    public void printValue() {
        printer.print(value);
    }
    public static FactoryMethod create(PrinterService printer,
            String value) {
        FactoryMethod instance = new FactoryMethod();
        instance.printer = printer;
        instance.value = value;
        return instance;
    }
}
```

```java
@Bean
public PrinterService printer() {
    return new ConsolePrinter();
}

@Bean
public FactoryMethod factoryMethod(PrinterService printer) {
    return FactoryMethod.create(printer, "MyTest");
}
```

Factory bean

- Construct and initialize beans that you don't control

```java
public class NameFactory {
  public List<String> createNameList() {
    return Arrays.asList("Jan", "Alie", "Bert");
  }
}
```

```java
@Configuration
public class TestConfig {
    @Bean
    public NameFactory nameFactory() {
         return new NameFactory();
    }

    @Bean
    public List<String> names() {
        return nameFactory().createNameList();
    }
}
```

Scopes

- singleton
  - A single instance of a bean. Dependencies to the bean are shared.
- prototype
  - New instance is created once for each injection point.
- prototype + method injection
  - New instance is created for each call to the injected method.

Web only scopes

- request
  - Instance per HTTP request.
- session
  - Instance per HTTP session.

Singleton Scope (default)

- There is **only one** HitCounter instance for all hitter's

```
@Bean
@Scope(scopeName="singleton")
public HitCounter hitCounter(){
    return new HitCounter();
}
```

```
@Bean
public Hitter hitter1(HitCounter hitCounter){
    return new Hitter( hitCounter);
}

@Bean
public Hitter hitter2(HitCounter hitCounter){
    return new Hitter( hitCounter);
}
```

Prototype Scope

- There is **one** HitCounter instance for every hitter

```
@Bean
@Scope(scopeName="prototype")
public HitCounter hitCounter(){
    return new HitCounter();
}
```

```
@Bean
public Hitter hitter1(HitCounter hitCounter){
    return new Hitter( hitCounter);
}

@Bean
public Hitter hitter2(HitCounter hitCounter){
    return new Hitter( hitCounter);
}
```

The Bean annotation

```
public @interface Bean {

    String[] name() default {};

    Autowire autowire() default Autowire.NO;

    String initMethod() default "";

    String destroyMethod() default AbstractBeanDefinition.INFER_M

}
```

- String[] name() => aliases for this bean
- String initMethod() => method to call during initialization
- String destroyMethod() => method to call upon closing the application context

Annotation configuration

Defining Spring beans

```
@Component
public class ConsolePrinter implements PrinterService {

    @Override
    public void print(String message) {
        System.out.println("Message: " + message);
    }
}
```

Is equivalent to:

```
@Bean
public PrinterService consolePrinter() {
    return new ConsolePrinter();
}
```

Instruct the Spring container to look for @Component

```
try (AnnotationConfigApplicationContext context =
        new AnnotationConfigApplicationContext()) {
    context.scan("com.example");
    context.refresh();
    PrinterService bean1 = context.getBean(PrinterService.class);
    bean1.print("Hello context");

    PrinterService bean2 =
        context.getBean("consolePrinter", PrinterService.class);
    bean2.print("Hello context");

    PrinterService bean3 =
        (PrinterService) context.getBean("consolePrinter");
    bean3.print("Hello context");
}
```

Instruct the Spring container to look for @Component alternative

```
@Configuration
@ComponentScan(basePackages="com.example")
public class ApplicationConfiguration {

}
```

@Component alternatives

- @Repository
- @Service
- @Controller

@Autowired

```
// Field injection
@Autowired
private PrinterService printerService;

// Constructor injection
@Autowired
public SetterDI(PrinterService printerService) {
    this.printerService = printerService;
}

// Setter injection
@Autowired
public void setPrinterService(PrinterService printerService) {
    this.printerService = printerService;
}
```

Logging by the container part 1

Returning cached instance of singleton bean 'consolePrinter' Autowiring by type bean 'setterDI' via **constructor** to 'consolePrinter'

Logging by the container part 2

Processing injected element of bean 'setterDI': AutowiredFieldElement for **private PrinterService printerService** Returning cached instance of singleton bean 'consolePrinter'

Logging by the container part 3

Autowiring by type from 'setterDI' to 'consolePrinter' Processing injected element of bean 'setterDI': AutowiredMethodElement for **setPrinterService(PrinterService)** Returning cached instance of singleton bean 'consolePrinter' Autowiring by type from 'setterDI' to 'consolePrinter'

Finished creating instance of bean 'setterDI'

Method injection

- Normally injection is done at bean creation time
- Injecting a prototype scope bean in singleton scope bean effectively makes the prototype scope bean a singleton scope bean
- Method injection allows injection at each call to a bean
- Spring injects an abstract method implementation

Method injection

```
public class MySingleton {
    @Autowired
    private javax.inject.Provider<MyPrototype> myPrototype;

    public void something(){
        MyPrototypeBean instance = myPrototype.get();
    }
}
```

Method injection alternative

```
public class MySingleton {

    public void something(){
        MyProtoTypeBean instance = getPrototypeBean();
    }

    @Lookup
    public MyProtoTypeBean getPrototypeBean() {
        // Spring will override this method
        return null;
    }
}
```

Lifecycle callbacks (not Spring specific)

```
public class LifecycleApplication implements CommandLineRunner {
    @Override
    public void run(String... arg0) throws Exception {
        System.out.println("Run");
    }

    @PostConstruct
    public void init() {
        System.out.println("PostConstruct");
    }
    @PreDestroy
    public void clean() {
        System.out.println("PreDestroy");
    }
```

```
PostConstruct
Run
PreDestroy
```

Conforming to standards
javax.inject.Inject

- Same semantics as @Autowired
- Standardized in JSR-330
- Needs JSR-330 on the classpath

```
@Inject
private PrinterService printerService;
```

```
<dependency>
    <groupId>javax.inject</groupId>
    <artifactId>javax.inject</artifactId>
    <version>[version]</version>
</dependency>
```

About @Scope

"The JSR-330 default scope is like Spring's prototype. However, to keep it consistent with Spring's general defaults, JSR-330 bean declared in the container is a singleton by default. To use a scope other than singleton use Spring's @Scope. javax.inject also provides a @Scope annotation. This one is intended to be used for creating your own annotations."

Qualifiers

- Qualifiers bind an injection point to a specific implementation of an interface
  - runtime binding, not compile time
  - unit testing still possible
- Necessary when multiple implementations exist

String based Qualifiers

Use Qualifier when declaring spring bean

```
@Component
@Qualifier("file")
public class FilePrinter implements PrinterService {

    @Override
    public void print(String message) {...}
}
```

String based Qualifiers

Use Qualifier at injection point

```
@Autowired
@Qualifier("file")
private PrinterService printerService;
```

Custom Qualifier annotation

```
@Retention(RUNTIME)
@Target(value={FIELD,TYPE,CONSTRUCTOR,METHOD})
@Qualifier
public @interface File {
}
```

```
@Component
@File
public class FilePrinter implements PrinterService{
```

```
@Autowired
@File
private PrinterService printerService;
```

Injecting multiple implementations

- All MovieCatalog implementations

```
@Autowired
private List<PrinterService> services;
```

- All MovieCatalog implementations with bean id

```
@Autowired
private Map<String,PrinterService> services;
```

Importing configuration

```
@Configuration
@Import({ DatasourceConfig.class, ControllerConfig.class })
public class ApplicationConfig {

}
```

Dependency injection summary

- Only define a dependency, don't instantiate or lookup dependencies
- Spring wires dependencies
  - Constructor injection
  - Setter injection
  - Field injection
- Dependencies are injected at bean creation time

Best form of dependency injection

Constructor injection

- For mandatory dependencies
- To create immutable objects by assigning dependencies to final fields

Setter injection

- For optional dependencies.

Field injection

- Annotations are clearly visible when opening the class file
- Uses reflection
- Classes are tightly coupled to dependency injection and cannot be used without it
  - Harder to unit test
- Dependencies are hidden from the outside
- Unable to create immutable objects
- Considered a bad practice in most cases

According to the Spring documentation

"The Spring team generally advocates constructor injection as it enables one to implement application components as immutable objects and to ensure that required dependencies are not null. Furthermore constructor-injected components are always returned to client (calling) code in a fully initialized state."

According to the Spring documentation

"Setter injection should primarily only be used for optional dependencies that can be assigned reasonable default values within the class. Otherwise, not-null checks must be performed everywhere the code uses the dependency. One benefit of setter injection is that setter methods make objects of that class amenable to reconfiguration or re-injection later."

XML, Java config or annotations?

- For most situations all three work
- XML is mainly used in 'older' applications
- Annotations have the advantage that they are in the same file as the code

XML, Java config or annotations?

- Java config is advised when adding dependency injection capabilities to classes from third party dependencies
- Annotations are advised for your own code

Exercise:

- Create an application with annotations for dependency injection
- Try the different forms of dependency injection:
  - Constructor, Setter, Field
- Implement a lifecycle callback
- Compare two objects with singleton scope
- Compare two objects with prototype scope Bonus: create another application that uses Java code for dependency injection

Spring Boot

" Takes an opinionated view of building production-ready Spring applications. Spring Boot favors convention over configuration and is designed to get you up and running as quickly as possible."

Spring Boot

- Opinionated
- Convention over configuration
- Spring is automatically configured
- Embedded Tomcat webserver
- XML no longer necessary
- Metrics and health checks
- Maven or Gradle

Maven configuration

```
<parent>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-parent</artifactId>
    <version>[version]</version>
</parent>
<dependencies>
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter</artifactId>
    </dependency>
</dependencies>
```

What's a Spring Boot starter?

```
<artifactId>spring-boot-starter</artifactId>
    ...
    <dependency>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-starter-logging</artifactId>
      <version>[version]</version>
      <scope>compile</scope>
    </dependency>
    <dependency>
      <groupId>javax.annotation</groupId>
      <artifactId>javax.annotation-api</artifactId>
      <version>[version]</version>
      <scope>compile</scope>
    </dependency>
    <dependency>
```

Gradle configuration

```
dependencies {
  compile("org.springframework.boot:spring-boot-starter:[version]")
}
```

SpringBoot Application

```
@SpringBootApplication
public class SpringBootBasicApplication {
    public static void main(String[] args) {
        SpringApplication.run(SpringBootBasicApplication.class,
            args);
    }
}
```

@SpringBootApplication

The main parts of the SpringBootApplication interface:

```
@SpringBootConfiguration
@EnableAutoConfiguration
@ComponentScan
public @interface SpringBootApplication
```

Spring Boot example

```
@SpringBootApplication
public class SpringBootBasicApplication
        implements CommandLineRunner {
    public static void main(String[] args) {
        SpringApplication.run(
            SpringBootBasicApplication.class, args);
    }

    @Override
    public void run(String... args) throws Exception {
        System.out.println("Hello Spring Boot.");
    }
}
```

Automatically stop application

In the previous example the application will run forever. If you want to stop the application after the execution you can change the code as indicated below.

Replace

```
SpringApplication.run(HelloCommandLine.class, args);
```

with

```
new SpringApplicationBuilder(HelloCommandLine.class)
    .web(WebApplicationType.NONE).run(args);
```

Traditional logging

```
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.stereotype.Component;

@Component
public class TraditionalLogging {

    private static final Logger logger =
            LoggerFactory.getLogger(TraditionalLogging.class);

    public void log() {
        logger.info("Logging something.");
    }
}
```

Alternative logging

- The traditional solution requires quite some code and you need to make sure you supply the correct class to the logger
- Another option is to create a logger bean
- The logger bean can be Autowired/injected in any class
- The logger bean definition uses the InjectionPoint to figure out in which class the logger is used

Alternative logging configuration

```
@SpringBootApplication
public class App implements CommandLineRunner {
    public static void main(String[] args) {
        new SpringApplicationBuilder(App.class).web(WebApplicatio
            .run(args);
    }

    @Bean
    @Scope("prototype")
    Logger log(InjectionPoint ip) {
        return Logger.getLogger(
            ip.getMember().getDeclaringClass().getName());
    }
```

Alternative logging

```
import java.util.logging.Logger;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Component;

@Component
public class AlternativeLogging {

    @Autowired
    private Logger logger;

    public void log() {
        logger.info("Logging something.");
    }
}
```

Executable 'fat' JAR

```
<build>
    <plugins>
        <plugin>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-maven-plugin</artifactId>
        </plugin>
    </plugins>
</build>
```

`mvn package` creates the JAR including all dependencies

Exercise:

- Create a simple helloworld Spring Boot application

Bean Validation

Bean validation

- Bean Validation API
- Define field constraints on Java classes
- Integrates with other frameworks such as JSF, Spring, JPA

Bean validation example

```
@Entity
public class Employee {
    @NotNull @Size(min = 2, max = 20)
    private String name;

    @Past
    private Date birthDate;

    @Min(value = 1000)
    private double salary;
```

Example constraints

- @Null
- @NotNull
- @AssertTrue
- @AssertFalse
- @Min
- @Max

Example constraints 2

- @DecimalMin
- @DecimalMax
- @Size
- @Digits
- @Past
- @Future
- @Pattern

Exercise:

- Implement Bean Validation

Database configuration

Options

- This chapter discusses two options
  - In-memory database
  - MySQL database
- Both work for JDBC and JPA
- You can choose which one to use for the training
- Configuration to enable SQL logging is provided as well

In-memory database

POM dependencies

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-web</artifactId>
</dependency>
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-devtools</artifactId>
</dependency>
<dependency>
    <groupId>com.h2database</groupId>
    <artifactId>h2</artifactId>
</dependency>
```

application.properties (optional)

```
spring.jpa.properties.hibernate.show_sql=true
spring.jpa.properties.hibernate.format_sql=true

spring.datasource.url=jdbc:h2:mem:testdb
spring.datasource.username=sa
spring.datasource.password=

spring.jpa.hibernate.ddl-auto=update

#spring.datasource.url=jdbc:mysql://localhost/test
#spring.datasource.driverClassName=com.mysql.jdbc.Driver
```

Example application

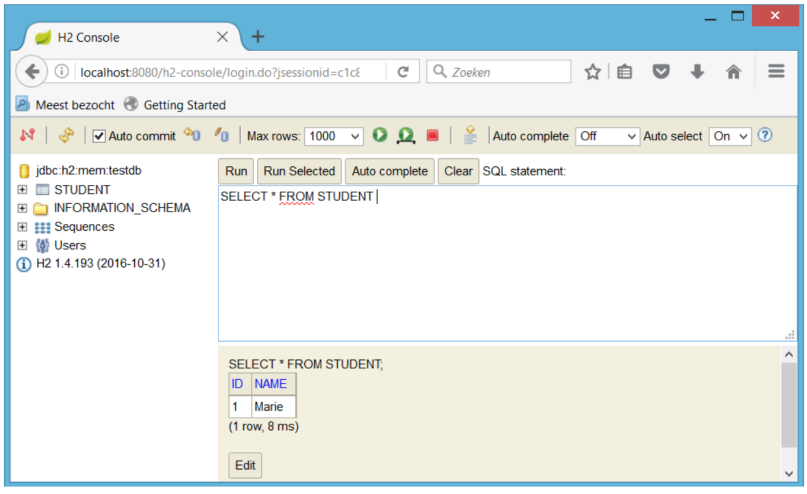
```
@SpringBootApplication
public class JDBCApplication implements CommandLineRunner {
    @Autowired
    private JdbcTemplate jdbcTemplate;

    public void run(String... strings) throws InterruptedExceptio
        jdbcTemplate.execute("DROP TABLE student IF EXISTS");
        jdbcTemplate.execute(
            "CREATE TABLE student(id SERIAL, name VARCHAR(255))")
        String insertSql="insert into student(name) values(?)";
        jdbcTemplate.update(insertSql,new Object[]{"Marie"});
    }
    public static void main(String[] args) {
        SpringApplication.run(JDBCApplication.class, args);
    }
}
```

Accessing the database

- Browse to: http://localhost:8080/h2-console/
- Use the following JDBC url: jdbc:h2:mem:testdb
- Alternatively connect to MySQL or other database

H2 Console



MySQL database

InfoSupport Solid Innovator

POM dependencies

```
<dependency>
    <groupId>mysql</groupId>
    <artifactId>mysql-connector-java</artifactId>
</dependency>
```

InfoSupport Solid Innovator

Configure MySQL database

- application.properties file in src/main/resources

```
# First create a 'test' database in MySQL manually
# Change username and password if necessary
spring.datasource.url: jdbc:mysql://localhost/test
spring.datasource.username=root
spring.datasource.password=
spring.datasource.driverClassName=com.mysql.jdbc.Driver
```

Exercise:

- Create an application which uses JDBC and the in memory database

Bonus: create another application that uses a MYSQL database

Data access

Spring data access overview

- Declarative transaction management
- Exception handling
- Simplified JDBC support
- ORM integration

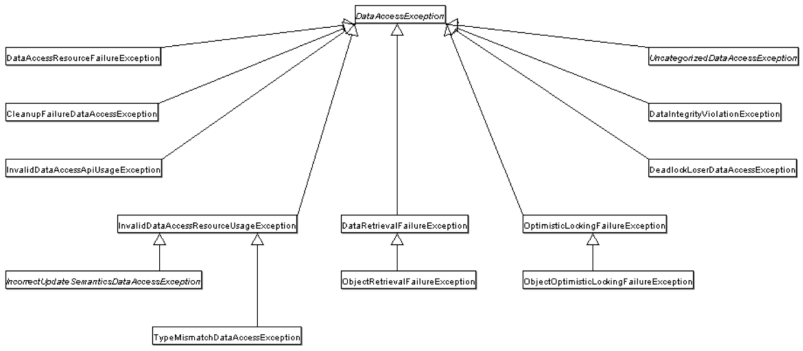
Things wrong with JDBC

- A lot of mandatory checked exceptions
  - Most are non recoverable
- SQLException is very generic
  - have to parse the sql error-code yourself
- Clumsy API
  - simple things require a lot of code

Spring JDBC support

- Exception wrapping
  - All exceptions are translated to unchecked exceptions
- Consistent exceptions
  - SQL error codes are translated to understandable exceptions
- Simplified APIs

Consistent exception hierarchy



InfoSupport
Solid Innovator

Dependencies

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-jdbc</artifactId>
</dependency>
<dependency>
    <groupId>com.h2database</groupId>
    <artifactId>h2</artifactId>
</dependency>
```

InfoSupport
Solid Innovator

JDBC Template

```
@Autowired
private JdbcTemplate jdbcTemplate;
```

Movie

```
public class Movie {

    private long id;
    private String title;
    private String genre;
    private String releaseDate;

    // Constructor(s)
    // Getters and setters
}
```

Queries and mapping

```
public List<Movie> listMovies() {
    return jdbcTemplate.query("SELECT * FROM movies",
        new RowMapper<Movie>() {

        @Override
        public Movie mapRow(ResultSet rs, int rowNum)
                throws SQLException {
            Movie movie = new Movie();
            movie.setId(rs.getLong("id"));
            movie.setTitle(rs.getString("title"));
            movie.setGenre(rs.getString("genre"));
            movie.setReleaseDate(rs.getString("releaseDate"));
            return movie;
        }
    });
}
```

Queries and mapping Lamdba style

```
public List<Movie> listMoviesLambda() {
    return jdbcTemplate.query("SELECT * FROM movies",
            (resultSet, rowNumber) ->
                new Movie(
                    resultSet.getLong("id"),
                    resultSet.getString("title"),
                    resultSet.getString("genre"),
                    resultSet.getString("releaseDate")));
}
```

QueryForObject

- Only one result is returned from the query

```
public Movie findMovie(int id) {
    return jdbcTemplate.queryForObject(
        "SELECT * FROM movies where id = ?",
            new RowMapper<Movie>() {
        @Override
        public Movie mapRow(ResultSet rs, int i)
                throws SQLException {
            Movie movie = new Movie();
            movie.setId(rs.getLong("id"));
            movie.setTitle(rs.getString("title"));
            movie.setGenre(rs.getString("genre"));
            movie.setReleaseDate(rs.getString("releaseDate"));
            return movie;
        }
    }, id);
}
```

InfoSupport Solid Innovator

Count the number of movies

```
public int countMovies() {
    int numberOfRows = jdbcTemplate.queryForObject(
        "SELECT COUNT(*) FROM movies", Integer.class);
    return numberOfRows;
}
```

InfoSupport Solid Innovator

Insert movie

```
public void jdbcTemplateInsertMovie(Movie movie) {
    List<Object[]> parameters = new ArrayList<Object[]>();
    parameters.add(new Object[] { movie.getTitle(),
        movie.getGenre(),
        movie.getReleaseDate() });
    jdbcTemplate.batchUpdate(
        "INSERT INTO movies(title, genre, releaseDate)
            VALUES (?,?,?)", parameters);
}
```

Delete movie

```
public void deleteMovie(long id) {
    jdbcTemplate.update("DELETE FROM movies WHERE id=?", id);
}
```

Exercise:

- Create an application with an in memory database
- Use the JdbcTemplate to insert some items into the database
- Use mapRow from the JdbcTemplate query to query the database
- Use Lambdas with the JdbcTemplate query to query the database
- Use the JdbcTemplate queryForObject to retrieve one result
- Create a query that counts the number of results
- Use the JdbcTemplate to delete some items

Transactions

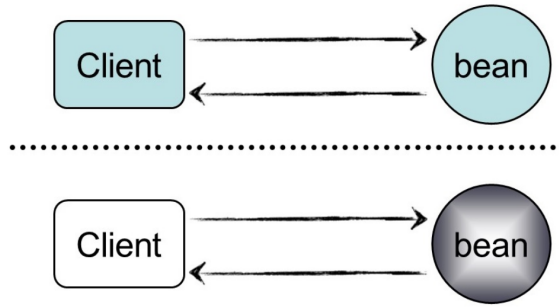
Transaction management

- Consistent programming model across different technologies
  - JDBC, JPA, Hibernate etc.
  - Run JDBC and JPA code in the same transaction
- Declarative tx-management
- Simplified programmatic tx-management API

Creating a transaction manager

- Different transaction managers for different persistence solutions
  - DataSourceTransactionManager
  - HibernateTransactionManager
  - JpaTransactionManager
  - WebLogicJtaTransactionManager
  - WebSphereUowTransactionManager

Required

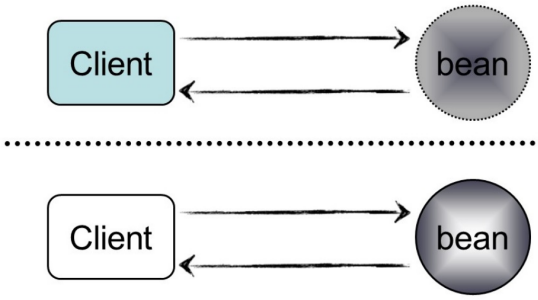


client's transactional context

no transactional context

bean's transactional context

Nested



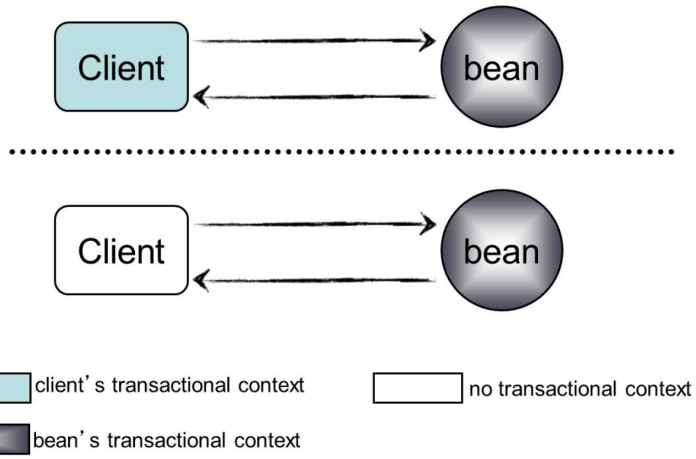
client's transactional context

no transactional context

bean's transactional context

nested transactional context

Requires new



Dependencies

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-jdbc</artifactId>
</dependency>
<dependency>
    <groupId>com.h2database</groupId>
    <artifactId>h2</artifactId>
</dependency>
```

Use an existing transaction

```
@Component
public class ExistingTransaction {
    @Autowired
    private JdbcTemplate jdbcTemplate;

    @Transactional(propagation = Propagation.REQUIRED)
    public void registerTrainer(String trainer) {
        jdbcTemplate.update(
            "INSERT INTO Teacher(name) values (?)",
            trainer);
    }
}
```

Create a new transaction

```
@Component
public class NewTransaction {
    @Autowired
    private JdbcTemplate jdbcTemplate;

    @Transactional(propagation = Propagation.REQUIRES_NEW)
    public void registerTrainer(String trainer) {
        jdbcTemplate.update(
            "INSERT INTO Teacher(name) values (?)",
            trainer);
    }
}
```

TrainingService

```
@Component
public class TrainingService {
    @Autowired
    private JdbcTemplate jdbcTemplate;
    @Autowired
    private ExistingTransaction existingTransaction;
    @Autowired
    private NewTransaction newTransaction;
```

TrainingService continued

```
    @Transactional(propagation = Propagation.REQUIRED)
    public void registerTrainers(boolean newTransactionBoolean,
            String... trainers) {
        for (String trainer : trainers) {
            if (newTransactionBoolean) {
                newTransaction.registerTrainer(trainer);
            } else {
                existingTransaction.registerTrainer(trainer);
```

This does not work

- It's not possible to call a transactional method direct
- Always call the method on a bean.

```
@Transactional(propagation = Propagation.REQUIRED)
public void registerTrainers(boolean newTransactionBoolean,
        String... trainers) {
    registerTrainerNewTransaction(trainer);
}

@Transactional(propagation = Propagation.REQUIRES_NEW)
public void registerTrainerNewTransaction(String trainer) {
    jdbcTemplate.update("INSERT INTO Teacher(name) values (?)",
        trainer);
}
```

RollbackApplication

```
@SpringBootApplication
public class RollbackApplication implements CommandLineRunner {
    @Autowired
    private JdbcTemplate jdbcTemplate;
    @Autowired
    private TrainingService trainingService;

    ....
```

RollbackApplication

```
@SpringBootApplication
public class RollbackApplication implements CommandLineRunner {
    ...

    private void recreateTable() {
        jdbcTemplate.execute("DROP TABLE Teacher IF EXISTS");
        jdbcTemplate.execute(
            "CREATE TABLE Teacher(id SERIAL, name VARCHAR(5))");
    }
    public static void main(String[] args) {
        new SpringApplicationBuilder(RollbackApplication.class).
            web(WebApplicationType.NONE).run(args);
    }
```

Run method

```
@Override
public void run(String... arg0) throws Exception {
    recreateTable();

    // Use existing transaction

    recreateTable();

    // Use new transaction
}
```

Use existing transaction

```
boolean newTransaction = false;
trainingService.registerTrainers(
    newTransaction, "Pim", "Jan");

trainingService.registerTrainers(
    newTransaction, "Henk", "Pieter");
trainingService.showTrainers();
```

Use new transaction

```
newTransaction = true;
trainingService.registerTrainers(
    newTransaction, "Pim", "Jan");

trainingService.registerTrainers(
    newTransaction, "Henk", "Pieter");
trainingService.showTrainers();
```

@Transactional attributes

- value
  - Optional qualifier specifying the tx-manager to use
- propagation
  - Transaction propagation
- isolation
  - Transaction isolation

@Transactional attributes

- readOnly
  - Read/write or read-only transaction
- rollbackFor
  - Array of Class objects that extends Throwable and should result in a rollback
- noRollbackFor
  - Array of Class objects that extends Throwable and should not result in a rollback

Rollback behavior

- Exceptions bubbled up the stack are handled by Spring
- Transaction is rolled back for unchecked exceptions
  - probably an unrecoverable error
- Transaction is not rolled back for checked exceptions
  - probably recoverable situation

Info Support Solid Innovator

Exercise:

- Create an application with an in memory database
- Make sure one column has a constraint such as "name VARCHAR(5)"
- Create a class that saves one item to the database using Propagation.REQUIRES_NEW
- Save two items to the database using the previous class
- Create a class that saves one item to the database using Propagation.Propagation.REQUIRED
- Save two items to the database using the previous class
- Observe the different results

Info Support Solid Innovator

JPA

InfoSupport Solid Innovator

Dependencies

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

InfoSupport Solid Innovator

Topics

- The Object/Relational paradigm mismatch
- Lifecycle of an entity
- Developing simple entities
  - Persistent identity (primary key)
  - Persistence Context
  - EntityManager API
  - Entity life cycle callbacks
  - Basic relational mapping

Info Support
Solid Innovator

The Object/Relational paradigm mismatch

- Granularity
- Subtypes
- Identity
- Associations
- Object Graph Navigation

Info Support
Solid Innovator

The evolution of database access

- Java application
- Data Access Object (DAO)
- JDBC with JDBC Driver
- Database

Uses database specific code

Typical layers for database access

- Java application
- Data Access Object (DAO)
- Hibernate
- JDBC with JDBC Driver
- Database

Hibernate converts code into database specific queries

The evolution of database access

- So we can forget about SQL and the database?
  - No JPA is not suitable for every situation
  - It takes effort to effectively use JPA
  - JPA is often not used correct resulting in performance issues

Java Persistence API (JPA)

- Map Java classes to tables using annotations
  - Including relations and inheritance
- Use a OO query language
  - abstraction on database specific sql
- An API to persist, update, delete and get Entities

Java Persistence API (JPA)

- Standardize ORM into single Java Persistence API
  - Usable in both Java SE and Java EE
  - Based on best practices from EJB 2.x, Hibernate, JDO, TopLink, etc.
- Support for pluggable, third-party persistence providers

Entities, reborn!

- Persistent objects
  - Entities, not Entity Beans
  - Java objects, not 'components'
  - Concrete classes
    - Support use of new keyword
  - Indicated by @Entity annotation

JPA implementations

- Hibernate
- EclipseLink
- OpenJPA
- Etcetera

Info Support
Solid Innovator

JPA implementations

- Use standardized API for most tasks
- Use provider specific API for advanced features

Info Support
Solid Innovator

Basic mapping

- @Entity to make a class an Entity
- @Id to configure Primary Key
  - required for each Entity
- @GeneratedValue to let the database generate keys
- By default each field is persisted

Basic mapping

```
@Entity
public class Contact {
    @Id
    @GeneratedValue
    private long id;

    private String firstname;
    private Date birthDate;

    // Getters and setters
}
```

Persistent Identity

- Define generator strategy type
  - AUTO, IDENTITY, TABLE, SEQUENCE
  - Depends on the underlying database

```
@Entity
public class Contact {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private long id;
}
```

Synchronizing entities with the database

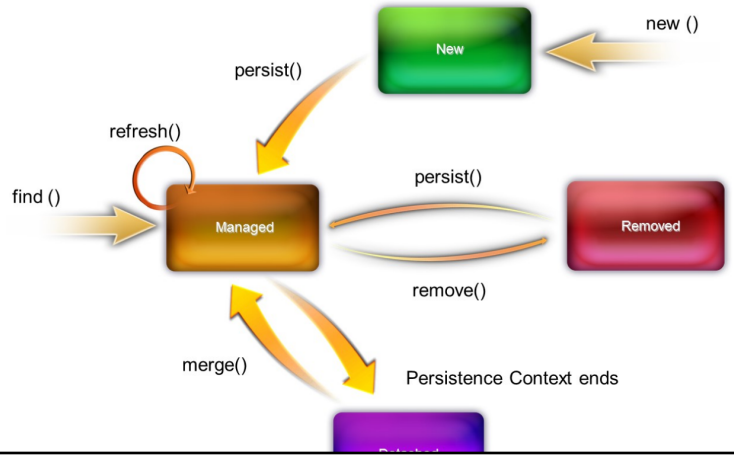
- EntityManager
  - API for object/relational mapping (ORM)
  - In Java EE:
    - Inject with @PersistenceContext
  - In Java SE:
    - Created by EntityManagerFactory (thread safe)
    - not thread safe

Synchronizing entities with the database

- Persistence Context
  - Set of “managed” entities (at runtime)

InfoSupport Solid Innovator

Life cycle



InfoSupport Solid Innovator

Example: Persist

- Insert a new instance of the entity into the database
- The entity instance becomes managed in the Persistence Context

```
@PersistenceContext
EntityManager entityManager;

public void saveContact(Contact contact) {
    entityManager.persist(contact);
}
```

Example: Merge

- State of detached entity gets merged into a managed copy of the entity
  - merge() returns managed entity with different Java identity than detached entity

```
@PersistenceContext
EntityManager entityManager;

public Contact saveUpdatedContact(Contact contact) {
    return entityManager.merge(contact);
}
```

Example: Find

- Find on primary key
- EntityManager returns a managed entity
- Returns null when key does not exists

```
@PersistenceContext
EntityManager entityManager;

public Contact changeContactName(long id, String newName) {
    Contact contact = entityManager.find(Contact.class, id);
    contact.setName(newName);
    return contact;
```

Removing entities

- Removal via EntityManager API
- Entities must be managed to be removed
  - E.g. do a find() first
- Use JPQL for batch deletes

```
@PersistenceContext
EntityManager entityManager;

public void removeContact(long id) {
    Contact contact = entityManager.find(Contact.class, id);
    em.remove(contact);
}
```

Basic relational mapping

- JPA provides enough flexibility to start from either direction:
  - Database to Entities
  - Entities to Database
- Elementary schema mappings:
  - Table and column mappings:
    - @Table
    - @Column

Mapping example

```
@Entity
@Table(name="CONTACTS")
public class Contact {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private long id;

    @Column(name = "C_NAME", length = 50, nullable = false)
    private String name;

    @Column(unique = true)
    private String email;

    @Temporal(value = TemporalType.TIMESTAMP)
    private Date birthDate;
}
```

Type conversion

- JPA 2.1 facilitates type conversion
  - @Converter
    - Annotate a class that implements the attributeConverter interface.
    - Use autoApply=true to convert any supported type
  - @Convert
    - Explicitly apply a converter on a property

Converter entity

```
@Entity
public class Movie {
    @Id
    private String title;

    @Convert(converter = BooleanTFConverter.class)
    private Boolean released;

    private String director;
}
```

BooleanTFConverter

```
public class BooleanTFConverter implements
    AttributeConverter<Boolean, String> {

    @Override
    public String convertToDatabaseColumn(Boolean value) {
        if (value) {
            return "T";
        } else {
            return "F";
        }
    }
}
```

Relationships

- Common relationships supported:
  - @ManyToOne, @OneToOne, @OneToMany, @ManyToMany
  - Unidirectional or bidirectional
- Owning side of relationship can specify physical mapping
  - @JoinColumn
  - @JoinTable

One-to-one

```java
@Entity
public class Contact {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private long id;

    @OneToOne
    private Address address;
}
```

```java
@Entity
public class Address {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private long id;

    @OneToOne(mappedBy = "address")
    private Contact contact;
}
```

Info Support
Solid Innovator

One-to-one

Contact

id	name	addr_fk

Address

id	street	city

- addr_fk in Contact table is linked to id in the address Table)

Info Support
Solid Innovator

One-to-Many unidirectional

```java
@Entity
@Table(name = "orders")
public class Order {
    @Id
    @GeneratedValue
    private long id;

    @OneToMany
    private List<LineItem> items;
    // Getters and setters
}
```

```java
@Entity
public class LineItem {
    @Id
    @GeneratedValue
    private long id;

    private String item;
    // Getters and setters
}
```

One-to-Many unidirectional

Orders

id	name

Orders_LineItem

orders_id	items_id

LineItem

id	item

One-to-Many bidirectional

```java
@Entity
@Table(name = "orders")
public class Order {
    @Id @GeneratedValue
    private long id;

    // Passive side
    @OneToMany(mappedBy = "order")
    private List<LineItem> items;
}
```

```java
@Entity
public class LineItem {
    @Id @GeneratedValue
    private long id;

    @ManyToOne
    private Order order;
}
```

Info Support
Solid Innovator

One-to-Many bidirectional

Orders

id	name

LineItem

id	item	order_id

Info Support
Solid Innovator

Lazy loading

- Multi-value associations are by default loaded lazily
  - prevents loading the whole database...
- Collections are proxied and will be loaded when used
- An entity must be managed to load associations!
  - LazyInitializationException

Bidirectional mappings

- One side is the “owning” side
- The other side is “passive” or “inverse”
  - the passive side does not synchronize changes

Works

```
Order order = new Order();
entityManager.persist(order);

LineItem item = new LineItem();
item.setOrder(order);
entityManager.persist(item);
```

```
@Entity
@Table(name = "orders")
public class Order {
    @Id @GeneratedValue
    private long id;

    private Date orderDate;

    @OneToMany(mappedBy = "order")
    private List<LineItem> items;
}
```

Doesn't work

- The relation is not set!

```
LineItem item = new LineItem();
entityManager.persist(item);

Order order = new Order();
order.getItems().add(item);
entityManager.persist(order);
```

```
@Entity
public class LineItem {
    @Id @GeneratedValue
    private long id;

    private String item;

    @ManyToOne
    private Order order;
}
```

Fixing the passive side

- Common trick to work with passive relations

```
@OneToMany(mappedBy = "order")
private List<LineItem> items;

public void addItem(LineItem item) {
    items.add(item);
    // Set the owning side:
    item.setOrder(this);
}
```

```
LineItem item = new LineItem();
entityManager.persist(item);

Order order = new Order();
order.addItem(item);
entityManager.persist(order);
```

InfoSupport Solid Innovator

Default fetch type

Relation	FetchType
OneToMany	LAZY
ManyToMany	LAZY
ManyToOne	EAGER
OneToOne	EAGER

Disclaimer: might differ for another version of JPA, Hibernate...

InfoSupport Solid Innovator

Cascading

- TransientObjectException
  - The referenced order is not persisted yet

```
Order order = new Order();
LineItem item = new LineItem();
item.setOrder(order);
entityManager.persist(item);
```

Cascading operations

- Cascading can be set on all relationship annotations
  - Default: no cascading
  - Values: PERSIST, MERGE, REMOVE, REFRESH, ALL
- Decide on entity-by-entity the most appropriate cascading setting

```
@ManyToOne(cascade = CascadeType.PERSIST)
private Order order;
```

```
@ManyToOne(cascade = {CascadeType.PERSIST, CascadeType.MERGE})
private Order order
```

Inheritance

- Entities can extend:
  - Other entities
  - Other plain Java classes
- Mapping inheritance hierarchy to:
  - Single table: everything in one table
    - Requires discriminator value
  - Joined: each class in a separate table
  - Table per concrete class

Inheritance mapping strategies

Single table

Pet				
id	name	breed	can_fly	DTYPE

Inheritance mapping strategies

Joined tables

Pet

id	name

Bird

id	can_fly

Dog

id	breed

Inheritance mapping strategies

Table per concrete class

Bird

id	name	can_fly

Dog

id	name	breed

Single Table

Employee table

id	name	platform	commission	DTYPE
1	Paul	Java	null	Programmer
2	Jaap	null	50	Sales

Single Table

```
@Entity
@Inheritance(strategy = InheritanceType.SINGLE_TABLE)
public class Employee {
    @Id @GeneratedValue
    private long id;

    private String name;
```

```
@Entity
public class Sales extends Employee {
    private int commission;
```

```
@Entity
public class Programmer extends Employee {
    private String platform;
```

Single Table

- Fast for each type of query
  - Always hit a single table
- Subclass fields must be nullable!

Joined table

Employee table

id	name
1	Paul
2	Jaap

Joined table

Programmer table

id	platform
1	Java

Sales table

id	comission
2	50

Joined table

- Normalized in the database
- Multiple tables for each type of query

Joined table

```
@Entity
@Inheritance(strategy = InheritanceType.JOINED)
public class Employee {
```

Table per concrete class

Employee

id	name

Programmer

id	name	platform
1	Paul	Java

Sales

id	name	comission
2	Jaap	50

Table per concrete class

- De-normalized in the database
- Single table queries for specific types
- Slow union query for querying all employees

Table per concrete class

```
@Entity
@Inheritance(strategy = InheritanceType.TABLE_PER_CLASS)
public class Employee {
    @Id @GeneratedValue(strategy = GenerationType.TABLE)
    private long id;
```

Embeddables

- Map multiple classes to a single table

```
@Entity
public class Purchase {
  @Id @GeneratedValue
  private long id;

  private String orderDescription;

  @Embedded
  private Contact contact;
}
```

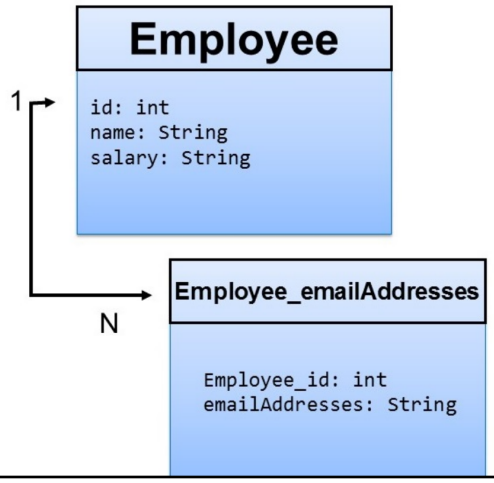
```
@Embeddable
public class Contact {
  @Column(name = "first_name")
  private String firstname;
  private String lastname;
```

Embeddables

- Purchase table

```
id: int
orderDescription: String
first_name: String
lastname: String
```

Value collections



Value collections

- Collections of simple types or Embeddable

```
@Entity
public class Employee {
    @Id
    private Long id;

    private String name;

    private double salary;

    @ElementCollection
    private Set<String> emailAddresses;
```

Queries

- EntityManager is a factory for Query objects
  - Using createQuery() methods
- Uses new and improved JPQL
- Queries can return entities, non-entities, or projections of entity data
- Native queries
  - Not portable across databases!

JPQL

- Query language for entities
- Vendor and implementation independent
- Similar to SQL, with slightly different syntax
- Translated to SQL using a dialect at runtime

Query examples

- Select all Employees (polymorphic)

```
SELECT emp FROM Employee emp
```

- Select only Programmers

```
SELECT p FROM Programmer p
```

- Where clause

```
SELECT emp FROM Employee emp WHERE emp.salary > 3000
```

Info Support
Solid Innovator

Query examples

- Between keyword

```
SELECT emp FROM Employee emp where emp.salary between 2000 and 30
```

- Subquery

```
SELECT emp FROM Employee emp WHERE emp.salary >
    (SELECT AVG(emp.salary) FROM Employee emp)
```

Info Support
Solid Innovator

Query examples

- Where clause on relation

```
SELECT emp FROM Employee emp where emp.department.id = :id
```

- Same as above

```
SELECT emp FROM Department department,
    IN(department.employees) emp
    WHERE department.id=:id
```

Executing a query

```
TypedQuery<Employee> query = em.createQuery(
    "SELECT emp FROM Employee emp where emp.department.id = :id",
    Employee.class;

query.setParameter("id", id);
List<Employee> employees = query.getResultList();
```

Query results

- Return managed entity

```
SELECT emp FROM Employee emp
```

- Return List

```
SELECT emp.name, emp.department.name FROM Employee emp
```

Query results

- Return List

```
SELECT new demo.Name(emp.name, emp.department.name) FROM Employee
```

```
public class Name {
    private String firstname;
    private String departmentName;

    public Name(String name, String departmentName) {
        this.firstname = name;
        this.departmentName = departmentName;
    }
}
```

Join queries

- A join is generated automatically when:
  - a path expression is used in the select
- Employees without a department are excluded

```
SELECT emp.department.name, emp.name FROM Employee emp
```

```
select
  department1_.name as col_0_0_,
  employee0_.name as col_1_0_
from
  Employee employee0_,
  Department department1_
where
  employee0_.department_id=department1_.id
```

Join queries

- Use the join keywords
- Employees without a department are now included

```
SELECT department.name, emp.name
FROM Employee emp
LEFT OUTER JOIN emp.department department
```

```
select
  department1_.name as col_0_0_,
  employee0_.name as col_1_0_
from        Employee employee0_
  left outer join
    Department department1_ on \
    employee0_.department_id=department1_.id
```

Join queries

- Use the join keywords to return collections

```
SELECT department.name, emp.name
FROM Department department
JOIN department.employees emp
```

```
select
department0_.name as col_0_0_,
employees1_.name as col_1_0_
from
Department department0_
  inner join
   Employee employees1_ on department0_.id=employees1_.department
```

Join queries

- Use the join keywords to prefetch collections
- Would normally be lazy loaded

```
SELECT department
FROM Department department
JOIN FETCH department.employees
```

Case expressions

- Conditional expressions

```
SELECT emp.name,
    CASE WHEN TYPE(emp) = Programmer
        THEN 'Cool dev'
    ELSE 'Just someone'
    END
FROM Employee emp
```

```
select employee0_.name as col_0_0_,
case when employee0_.DTYPE='Programmer'
    then 'Cool dev guy'
    else 'Just some guy'
    end as col_1_0_
from Employee employee0_
```

Bulk updates

- Updating or deleting many entities can be done in a bulk update
  - much cheaper than iterating and updating in Java!

```
Query query =
    entityManager.createQuery("update Employee e
        set e.salary = e.salary + 100");
int changed = query.executeUpdate();
```

Named queries

- Specify re-usable queries on Entity class

```
@Entity
@NamedQuery(
    name = "findByName",
    query = "select emp from Employee emp
        where emp.name LIKE :name")
public class Employee {
```

```
TypedQuery<Employee> query =
    entityManager.createNamedQuery("findByName", Employee.class);
query.setParameter("name", name + "%");
```

Native queries

- JPQL only supports a subview of SQL
  - unsupported features examples:
    - Inline views, hierarchical queries, stored procedures and vender specific extensions
- API more friendly than JDBC
- Supports mapping to Entity classes

Native queries

- Map result to Employee objects
- Each column must be a property on the Entity

```
Query query = entityManager.createNativeQuery(
    "select * from employee where name like ?",
    Employee.class);
query.setParameter(1, name + "%");
return query.getResultList();
```

SqlResultMapping

- Specify how query results are mapped to an Entity

```
@SqlResultSetMapping(name = "employeeResult", entities = {
    @EntityResult(entityClass = Employee.class, fields = {
        @FieldResult(name = "name", column = "EMP_NAME"),
        @FieldResult(name = "id", column = "id")})})
public class Employee {
```

```
Query query = em.createNativeQuery(
    "select id, name as EMP_NAME from employee where name like ?"
    "employeeResult");
```

Criteria API

- The criteria API is used to build queries from code
- Useful for queries that are dynamically created at runtime
  - e.g. a search screen with optional fields
- Does not replace JPQL

InfoSupport Solid Innovator

Criteria API

- CriteriaBuilder
  - contains methods to construct a query (equals, gt, max etc.)
- CriteriaQuery
  - uses a fluent API to build the query
  - the type parameter should be the type that is returned by the SELECT
- Root
  - the first Entity in the FROM

InfoSupport Solid Innovator

Criteria example

```
CriteriaBuilder criteriaBuilder = em.getCriteriaBuilder();
CriteriaQuery<Employee> criteriaQuery =
    criteriaBuilder.createQuery(Employee.class);
Root<Employee> employee = criteriaQuery.from(Employee.class);
criteriaQuery.select(emp);

List<Employee> employees =
    em.createQuery(criteriaQuery).getResultList();
```

where clause

```
c.select(emp).where(cb.equal(emp.get("name"), name));
```

- equals: operator
- emp.get("name"): on the 'name' property
- name: the argument

predicates and paths

```
c.select(emp).
    where(
        cb.and(
            cb.equal(emp.get("name"), name),
            cb.equal(emp.get("department").get("id"), depId)));
```

- Defines the 'and' predicate
- emp.get("department").get("id"): path expression

CrudRepository

- Creates standard methods like save / delete / find etcetera
- Some methods are present in the interface
- Other methods can be added easily

JPAApplication

```
@SpringBootApplication
public class JPAApplication {
}
```

Book Entity

```
@Entity
public class Book {
    @Id
    @GeneratedValue
    private Long id;

    private String name;

    private String author;

    public Book(String name, String author) {
        super();
        this.name = name;
        this.author = author;
    }
```

Book Repository

```
public interface BookRepository
        extends CrudRepository<Book, Long> {
     List<Book> findByName(String name);
     List<Book> findByAuthor(String author);
}
```

@Query

```
@Query("SELECT b FROM Book b WHERE b.name=:name
    OR b.author=:author")
List<Book> findBooksByNameOrAuthor(
    @Param("name") String name, @Param("author") String author);
```

Streaming

```
@Repository
public interface StreamingBookRepository
        extends CrudRepository<Book, Long> {
    @Query("select b from Book b")
    Stream<Book> findBooks();
}
```

JPARepository

- JPARepository extends PagingAndSortingRepository
- PagingAndSortingRepository extends CrudRepository

JPARepository

- PagingAndSortingRepository and CrudRepository are generic data interfaces
- JPARepository is a store specific interface for JPA
- Try to avoid store specific interfaces. It leaks the persistence technology in the client.

Testing DAO's

- Unit testing doesn’t make sense
  - Queries can only be tested with a real database
- Run automated tests within a Spring container
  - Use a real database or an embedded one

Dependencies

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

Test setup

```
@RunWith(SpringRunner.class) @DataJpaTest
public class BookRepositoryTest {
    @Autowired
    private BookRepository bookRepository;

    private Book springBootResultingBook;

    @Before
    public void init() {
        Book springBook = new Book("Pro Spring", "Bob Harrop");
        Book springBootBook =
            new Book("Spring Boot in Action", "Craig Walls");
        bookRepository.save(springBook);
        springBootResultingBook =
            bookRepository.save(springBootBook);
    }
```

Tests

```
@Test
public void testFindOne() {
    Optional<Book> optionalBook =
        bookRepository.findById(springBootResultingBook.getId());
    Book book = optionalBook.get();
    assertEquals("Spring Boot in Action", book.getName());
    assertEquals("Craig Walls", book.getAuthor());
}
```

Tests

```
@Test
public void testFindByAuthor() {
    List<Book> findByAuthorBookList =
        bookRepository.findByAuthor("Craig Walls");
    Book book = findByAuthorBookList.get(0);
    assertEquals("Spring Boot in Action", book.getName());
    assertEquals("Craig Walls", book.getAuthor());
}
```

Testing DAO's

- Transactions are rolled back after each test by default
  - no need to re-insert test data for each test
- It's possible to test JPA code using JDBC queries

Testing DAO's

- Do not rollback one unit test or a whole unit test class

```
@Transactional(propagation = Propagation.NOT_SUPPORTED)
```

SQL Queries

- Number of SQL queries might be different from the number of JPA queries
- Big source of performance issues
- Always verify the (number of) generated SQL queries!

Book

```
@Entity
public class Book {

    @Id
    @GeneratedValue
    private Long id;

    private String name;

    // Constructors
    // Getters and setters
}
```

Course

```
@Entity
public class Course {
    @Id
    @GeneratedValue
    private Long id;

    private String name;

    @OneToMany(fetch = FetchType.EAGER)
    private List<Book> books;

    // Constructors
    // Getters and setters
}
```

Database structure

- http://localhost:8080/h2-console
- Three tables
  - BOOK
  - COURSE
  - COURSE_BOOKS

Enable SQL query logging

Add the following lines to application.properties

```
spring.jpa.properties.hibernate.show_sql=true
spring.jpa.properties.hibernate.format_sql=true
spring.jpa.properties.hibernate.type=trace
```

Creating entities

```
List<Book> springBooks = new ArrayList<Book>();
Book book1 = new Book("Spring in Action");
springBooks.add(book1);
Book book2 = new Book("Spring Boot in Action");
springBooks.add(book2);

List<Book> javaBooks = new ArrayList<Book>();
Book book3 = new Book("Core Java");
javaBooks.add(book3);
Book book4 = new Book("Head First Java");
javaBooks.add(book4);
```

Save entities

```
bookRepository.save(book1);
bookRepository.save(book2);
bookRepository.save(book3);
bookRepository.save(book4);

courseRepository.save(new Course("Spring", springBooks));
courseRepository.save(new Course("Java", javaBooks));
```

CrudRepository findAll()

```
Iterable<Course> courseIterable = courseRepository.findAll();
```

- Number of SQL queries: 3

```
Hibernate:
    select
        course0_.id as id1_1_,
        course0_.name as name2_1_
    from
        course course0_
Hibernate:
    select
        books0_.course_id as course_i1_2_0_,
        books0_.books_id as books_id2_2_0_,
        book1_.id as id1_0_1_,
        book1_.name as name2_0_1_
    from
        course_books books0_
    inner join
```

JPQL

```
List<Course> courses = (List<Course>)
    entityManager.createQuery("SELECT c FROM Course c").
        getResultList();
```

- Resulting courses:

```
Number of courses: 2
Spring : Spring in Action | Spring Boot in Action
Java : Core Java | Head First Java
```

- Number of SQL queries: 3

```
Hibernate:
    select
        course0_.id as id1_1_,
        course0_.name as name2_1_
    from
        course course0_
Hibernate:
    select
        books0_.course_id as course_i1_2_0_,
        books0_.books_id as books_id2_2_0_,
        book1_.id as id1_0_1_,
        book1_.name as name2_0_1_
    from
        course_books books0_
    inner join
```

JPQL join fetch

```
List<Course> courses = (List<Course>)
    entityManager.createQuery(
        "SELECT c FROM Course c join fetch c.books").
            getResultList();
```

- Resulting courses:

```
Spring : Spring in Action | Spring Boot in Action
Spring : Spring in Action | Spring Boot in Action
Java : Core Java | Head First Java
Java : Core Java | Head First Java
```

- Number of SQL queries: 1

```
Hibernate:
    select
        course0_.id as id1_1_0_,
        book2_.id as id1_0_1_,
        course0_.name as name2_1_0_,
        book2_.name as name2_0_1_,
        books1_.course_id as course_i1_2_0__,
        books1_.books_id as books_id2_2_0__
    from
        course course0_
    inner join
        course_books books1_
            on course0_.id=books1_.course_id
    inner join
        book book2_
```

JDBC versus JPA

- JDBC
  - Better performance
  - SQL queries
  - Accessing data in a relational way

JDBC versus JPA

- JPA
  - Easier migration to another database vendor
  - Possible to use database vendor specific features
  - Accessing data in a object oriented way
  - Abstraction layer doesn't always work intuitive
  - It's a mistake to think that you no longer need SQL knowledge

CrudRepository vs JPQL vs Criteria API vs @(Named)Query

- In general
- CrudRepository for all easy queries
- JPQL for the 'custom' queries
- Criteria for dynamic queries with variable parameters and type safety
- @NamedQuery for queries that are used in multiple places

Exercise:

- Create an application with an in memory database
- Enable SQL query logging
- Create an object that maps to the database
- Use the EntityManager to store, find and remove
- Create a OneToOne unidirectional and bidirectional
- Create OneToMany and ManyToMany relations
- Use cascading to save underlying objects
- Create a JPQL query with at least one parameter
- Create a JPQL join query
- Use the CrudRepository with standard and custom queries

MVC

MVC

- MVC design pattern implemented in Spring
- Web frontend
- Can be used for instance with JSP's
- Uses Expression Language (EL) to render model values

POM dependencies

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

MVC setup

```
@SpringBootApplication
@EnableAutoConfiguration
public class MVCApplication {
    public static void main(String[] args) {
        SpringApplication.run(MVCApplication.class, args);
    }
}
```

MVC Controller

```
@Controller
public class WelcomeController {
    @RequestMapping("/welcome")
    public String welcome(@RequestParam(value="name",
            required=false,
            defaultValue="John Doe") String name, Model model) {
        model.addAttribute("name", name);
        return "welcomeView";
    }
}
```

HTML page

```
<html xmlns:th="http://www.thymeleaf.org">
<head>
    <title>Getting Started: Serving Web Content</title>
    <meta http-equiv="Content-Type" content="text/html;
        charset=UTF-8" />
</head>
<body>
    <p th:text="'Hello, ' + ${name} + '!'" />
</body>
</html>
```

- src/main/resources/templates/welcomeView.html
- http://localhost:8080/welcome
- http://localhost:8080/welcome?name=Wim

URL path variable

- /products/{productId}
- /products/{productId}/something

```
@RequestMapping("products/{productId}")
public Product productDetails(@PathVariable int productId) {
    productService.retrieveProduct(productId);
    ...
}
```

Request parameter

- /search?query=Spring

```
@RequestMapping("search")
public Product search(
    @RequestParam(required = true) String query) {

    productService.retrieveProduct(query);
    ...
}
```

Request headers

- Accept: text/html

```
@RequestMapping("/example")
public Product example(@RequestHeader String accept) {
    ...
}
```

Request Cookie

- Accept: text/html

```
@RequestMapping("/example")
public Product example(@CookieValue String userId) {
    ...
}
```

Handling specific requests

```
@RequestMapping(value = "example", method = RequestMethod.GET)
```

Exercise:

- Create a MVC application

REST

Why REST?

- Communication between services or with the (JavaScript) frontend
- Alternative to SOAP with XML

REST fact sheet

- REpresentational State Transfer
- Introduced by Roy Fielding
  - Dissertation in 2000
  - An architectural style for distributed systems
- HTTP is an example of REST

RESTful web services

- Services implemented conform the REST principles
- Mostly based at HTTP

The REST hype

- More public web APIs
  - Google, Amazon, Flickr etc.
- Popularity of lightweight web frameworks
  - Rails / Grails
- People are tired of WSDL
- XML is not always the best format

Everything is a resource

- A list of books
- A product
- A list of search results
- An order

Representation of a resource

- XML
- JSON
- HTML

Representation of a resource

```
<product>
    <productId>1004</productId>
    <name>Product A</name>
</product>
```

```
{"product": {"productId":"1004","name":"ProductA"}}
```

```
<p class="product">
    <span class="productId">1004</span>
    <span class="name">Product A</span>
</p>
```

HTTP content negotiation

- A client can ask for specific formats
- The accept header
  - Accept: "application/xml"

Dynamic resources

- A resource can be 'static'
  - A record in your database
  - A file
- A resource can be 'dynamic'
  - Calculated results
  - Generated data

RESTful properties

- Uniform Interface
- Addressability
- Connectedness
- Statelessness

Uniform Interface

- GET
  - Retrieve a resource representation
- HEAD
  - GET without body: “Does this resource exists?”
- POST
  - Overloaded: implementation may vary. Might generate a new URI
- DELETE
  - Delete a resource with a specified URI

Uniform Interface

- PUT (replace)
  - Modify a complete resource
- PATCH (update)
  - Modify partial resources
  - Might fail is as the value can be changed in the meantime

Uniform Interface

PUT /user/jthijssen HTTP/1.1

Joshua

PATCH /user/jthijssen HTTP/1.1

Josh Joshua

Possibilities

- These are best practices
- It's possible to implement a GET to work as a POST etc.
- GET should be safe (idempotent)

Addressability

- /products
- /product/{id} => /product/10
- /products?color=red
- /search?q=jax-rs

Each resource has a Unique Resource Identifier (URI)

Connectedness

- Navigate from one resource to another
- Clients do not generate URIs
- One of the most important WEB concepts
  - Hyperlinks

Not connected

```
<searchresult>
   <product name="Product 1"/>
   <product name="Product 2"/>
   <product name="Product 3"/>
   <product name="Product 4"/>
</searchresult>
```

How do I get product information?

Connected

Linked to more information

```
<searchresult>
   <product name="Product 1" url="http://myservice/product/1"/ >
   <product name="Product 2" url="http://myservice/product/2"/ >
   <product name="Product 3" url="http://myservice/product/3"/ >
   <product name="Product 4" url="http://myservice/product/4"/ >
</searchresult>
```

Connected

- Also known as Hypermedia
- Spring HATEOAS offers a library to build what they call a 'Hypermedia-Driven RESTful Web Service'

RESTful web services in Spring

- Web Services are implemented using controllers
- Familiar Spring MVC programming model

@ResponseBody

- The object returned is converted using a HttpMessageConverter
  - Jaxb2RootElementHttpMessageConverter
  - MappingJacksonHttpMessageConverter
  - StringHttpMessageConverter
  - ...

InfoSupport Solid Innovator

@ResponseBody

```
@XmlRootElement
public class Book {
```

```
@RequestMapping(method = RequestMethod.GET, value = "books",
    headers = "accept=application/xml")
public @ResponseBody BookList listBooksXml() {
    List<Book> books = bookCatalog.listBooks();
    return new BookList(books);
}
```

InfoSupport Solid Innovator

List of elements

- Don't return a list of elements
- Wrap the list of elements in a wrapper object

Choosing handlers

- How to offer data both as XML and HTML?
  - use the HTTP accept header
  - use a different extension
  - use a request parameter
  - use content negotiation

Choosing handlers

```
@RequestMapping(method = RequestMethod.GET
    value = "books.xml")
```

```
@RequestMapping(method = RequestMethod.GET
    value = "books",
    headers = "accept=application/xml")
```

```
@RequestMapping(method = RequestMethod.GET
    value = "books",
    params = "contentType=application/xml"
```

POM dependencies

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

HelloWorld

```
@RestController
public class HelloWorldController {
    @RequestMapping("/hello")
    public String helloWorld() {
        return "Hello world";
    }
}
```

Available on http://localhost:8080/hello

Book and Course

```
public class Book {
    private String name;
```

```
public class Course {
    private String location;

    private Book book;
```

Course controller

```
@RestController
public class CourseController {

    @RequestMapping("/course")
    public Course helloWorld() {
        Book book = new Book("Core Spring");
        Course course = new Course("Veenendaal", book);
        return course;
    }
}
```

Available on http://localhost:8080/course

RESTful clients

- Use RESTful web services using a template

POM dependencies

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-web</artifactId>
</dependency>
<dependency>
    <groupId>com.fasterxml.jackson.core</groupId>
    <artifactId>jackson-databind</artifactId>
</dependency>
```

Consume REST endpoint

```
@SpringBootApplication
public class Application {
    public static void main(String[] args) {
        RestTemplate restTemplate = new RestTemplate();
        Course course = restTemplate.getForObject(
            "http://localhost:8080/course", Course.class);
        System.out.println(course.toString());
    }
}
```

Sending data to a REST endpoint

```
@SpringBootApplication
public class Application {
    public static void main(String[] args) {
        RestTemplate restTemplate = new RestTemplate();

        ...

        restTemplate.put(
            "http://localhost:8080/course", changedCourse);
        restTemplate.postForLocation(
            "http://localhost:8080/course", newCourse);
    }
}
```

Documentation

- Could document your API with XML
- Or use alternatives such as Swagger

Swagger

- View the API as a website
- Minimal configuration
- Works automatically for the REST endpoints
- Possibility to add documentation on the endpoint with annotations

POM dependencies

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

<dependency>
    <groupId>io.springfox</groupId>
    <artifactId>springfox-swagger2</artifactId>
    <version>2.6.1</version>
</dependency>
<dependency>
    <groupId>io.springfox</groupId>
    <artifactId>springfox-swagger-ui</artifactId>
    <version>2.6.1</version>
</dependency>
```

Config

```
@Configuration
@EnableSwagger2
public class SwaggerConfig {
    @Bean
    public Docket api() {
        return new Docket(DocumentationType.SWAGGER_2)
            .select()
            .apis(RequestHandlerSelectors.any())
            .paths(PathSelectors.any())
            .build();
    }
}
```

Swagger website

http://localhost:8080/swagger-ui.html

Optional documentation

```
@RestController
public class SwaggerCourseController {
    @ApiOperation(notes = "This method allows you to retrieve
            a fixed course", value = "Retrieve course")
        @ApiResponses({
            @ApiResponse(code = 200, message = "Everything ok.",
                response = Course.class),
            @ApiResponse(code = 404, message = "No course.")
    })
    @RequestMapping("/swaggercourse")
    public Course helloWorld() {
        Book book = new Book("Core Spring");
        Course course = new Course("Veenendaal", book);
        return course;
    }
}
```

Exercise:

- Create a REST endpoint application
- Create a REST consumer application
- Use the REST consumer to retrieve data from the REST endpoint
- Use the REST consumer to send data to the REST endpoint

Aspect Oriented Programming (AOP)

Why AOP?

- Some concerns in an application are cross cutting
  - the same code would be repeated at many places
  - e.g. security, logging, retry-on-error etc.
- AOP complements the Object Oriented paradigm

AOP in Java

- AspectJ is the most used AOP implementation
- Needs an additional weaving compiler after the Java compiler
- Makes the build process more complicated
- Spring AOP is easier but less powerful

AOP definitions

- Join point
  - a point during execution of code (a method execution)
- Advice
  - Action taken at a join point
  - e.g. “around”, “before” or “after”
  - this is where you implement your code
- Pointcut
  - predicate that matches join points
  - e.g. “all methods in a certain package”

POM dependencies

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

Example class

```
@Component("student")
public class AOPStudent {
    private String name = "Wim";

    public String getName() {
        return name;
    }
}
```

Declaring an Aspect

```
@Aspect
@Component
public class AOPStudentAspect {

    //Pointcut and advice

    @Before("execution(* com.example.*.get*())")
    public void getGetterAdvice(){
        System.out.println("Getter is called");
    }
}
```

Triggering the aspect

```
@SpringBootApplication
public class AOPApplication implements CommandLineRunner {
    @Autowired
    private AOPStudent student;

    @Override
    public void run(String... arg0) throws Exception {
        System.out.println(student.getName());
    }

    public static void main(String[] args) {
        // Used to make sure Tomcat is not started
        new SpringApplicationBuilder(AOPApplication.class)
            .web(WebApplicationType.NONE).run(args);
    }
}
```

Pointcut definitions

```
execution(public String lab2.MovieCatalog.toString(..))
```

- `public`: access modifier
- `String`: return type
- `MovieCatalog`: type
- `toString`: method
- `..`: arguments

Pointcut examples

- execution(public * *(..))
  - All public methods
- execution(String *(..))
  - All methods returning a String
- execution(public * save(..))
  - All public save methods
- execution(public * a.b.dao.* (..))
  - All public methods in the dao package

Pointcut examples

- execution(public String *(String, Integer))
- All public methods returning a String and arguments of type String and Integer
- @annotation(a.b.MyAnnotation)
- All methods annotated a.b.MyAnnotation

Before advice

```
@Before("execution(public String hello())")
public void before(JoinPoint joinpoint){
}
```

- Make sure you import the correct JoinPoint

```
import org.aspectj.lang.JoinPoint;
```

JoinPoint

```
@Before("execution(public String hello())")
public void before(JoinPoint joinPoint){
    System.out.println(joinPoint.getStaticPart().toString());
    System.out.println("Begin");
}
```

Passing arguments

```
@Before("execution(* *(..)) && args(message)") {
public void before(JoinPoint joinPoint,
        String message ) {
    System.out.println("Calling method with String: " + message);
}
```

- Declare the arguments: `args(message)`
- The `message` in the pointcut must match the `message` argument of the `before` method

After advices

```
@AfterReturning(value = "execution(* * (..))",
    returning = "retval")
public void afterSucces(Object retval){
    System.out.println("I returned succesfully: " + retval);
}

@AfterThrowing(value = "execution(* * (..))",
    throwing = "exception")
public void afterException(Exception exception){
    System.out.println("An exception was thrown " +
        exception.getMessage());
}
```

InfoSupport Solid Innovator

After advices

```
@After(value = "execution(* * (..))")
public void after(Object retval){
    System.out.println("I'm done");
}
```

InfoSupport Solid Innovator

Around advice

- Call `proceed` to continue the original method call

```
@Around(value = "execution(public * *(..))")
public Object trace(ProceedingJoinPoint proceedingJoinPoint)
        throws Throwable {
    long startTime = System.currentTimeMillis();

    Object result = proceedingJoinPoint.proceed();

    long elapsedTime = System.currentTimeMillis() - startTime;
    System.out.println(proceedingJoinPoint.getStaticPart().
        getSignature().getName() + " " + elapsedTime);
    return result;
}
```

InfoSupport Solid Innovator

Retry example

```
@Around("execution(public * *(..))")
public Object retry(ProceedingJoinPoint proceedingJoinPoint)
        throws Throwable {
    Object result = null;

    try {
        result = proceedingJoinPoint.proceed();
    } catch (Throwable e) {
        System.out.println("Sleep 2 seconds before retry");
        TimeUnit.SECONDS.sleep(2);
        result = proceedingJoinPoint.proceed();
    }
    return result;
}
```

InfoSupport Solid Innovator

Reusable pointcut definitions

```
@Aspect
@Component
public class MyPointcuts {

    @Pointcut("execution(* a.b.dao(..))")
    public void inDAOLayer() {}
}
```

```
@After("lab2.aop.MyPointcuts.inDAOLayer")
public void after() {
    System.out.println("I'm done!");
}
```

Introductions

- Introduce new methods and interfaces to a class
- How to make a car fly and shoot?
- Plain car implementation

```
@Component
public class CarImpl implements Car {
    @Override
    public void drive() {
        System.out.println("Driving...");
    }
}
```

Introduction example

```
@Component
@Aspect
public class AbilityIntroduction {
    @DeclareParents(value="com.example.CarImpl",
        defaultImpl = com.example.FlyerImpl.class)
    public Flyer flyer;

    @DeclareParents(value="com.example.CarImpl",
        defaultImpl = com.example.ShooterImpl.class)
    public Shooter shooter;
}
```

InfoSupport Solid Innovator

Introduction example

```
public class FlyerImpl implements Flyer {
    @Override
    public void fly() {
        System.out.println("Flying!");
    }
}
```

```
public class ShooterImpl implements Shooter {
    @Override
    public void shoot() {
        System.out.println("Shooting!");
    }
}
```

InfoSupport Solid Innovator

Using the introduction

```
@Component
public class IntroductionExample {
    @Autowired
    private Car car;

    public void testCarAbilities() {
        car.drive();

        Flyer flyer = (Flyer) car;
        flyer.fly();

        Shooter shooter = (Shooter) car;
        shooter.shoot();
    }
}
```

Info Support Solid Innovator

Exercise:

- Create an application
- Implement a before advice that shows the start of a method
- Change the before advice to show the arguments of the method
- Implement an around advice that shows the duration of the method

Info Support Solid Innovator

External values

POM dependencies

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

Application

```
@SpringBootApplication
@PropertySource("classpath:/com/example/config.properties")
public class App implements CommandLineRunner {

    public static void main(String[] args) {
        new SpringApplicationBuilder(App.class)
            .web(WebApplicationType.NONE).run(args);
    }

    ...
}
```

config.properties

```
environment=development
version=42

title="Spring course"
values.split=1-2-3
```

Using the property file

```
@Autowired
private Environment env;

@Value("#{'${title}'}")
String title;

@Value("#{'${values.split}'.split('-')[1]}")
String second;

@Override
public void run(String... arg0) throws Exception {
    System.out.println(env.getProperty("environment"));
    System.out.println(env.getProperty("UNKNOWN", "Default"));
    System.out.println("Title retrieved by SpEL " + title);
    System.out.println("Second value split by SpEL " + second);
```

Output

```
development
Default
Title retrieved by SpEL "Spring course"
Second value after split by SpEL 2
```

Exercise:

- Create an application
- Create a config.properties with some values
- Use the values in the application

Caching

POM dependencies

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

Info Support
Solid Innovator

NameService

```
public class NameService {

    public List<String> getNames() {
        List<String> names = new ArrayList<>();
        names.add("Wim");
        names.add("Anna");

        try {
            Thread.sleep(3000L);
        } catch (InterruptedException e) {
            // TODO Auto-generated catch block
            e.printStackTrace();
        }
        return names;
    }
}
```

Info Support
Solid Innovator

UncachedNameService

```
@Component
public class UncachedNameService extends NameService {

    @Override
    public List<String> getNames() {
        return super.getNames();
    }
}
```

CachedNameService

```
@Component
public class CachedNameService extends NameService {

    @Override
    @Cacheable("names")
    public List<String> getNames() {
        return super.getNames();
    }
}
```

Application

```
@SpringBootApplication
@EnableCaching
public class App implements CommandLineRunner {

    public static void main(String[] args) {
        new SpringApplicationBuilder(App.class)
            .web(WebApplicationType.NONE).run(args);
    }

    @Autowired
    private CachedNameService cachedNameService;

    @Autowired
    private UncachedNameService uncachedNameService;
```

InfoSupport Solid Innovator

Application

```
@Override
public void run(String... args) throws Exception {
    calculate(uncachedNameService, "Uncached");
    calculate(uncachedNameService, "Uncached");
    calculate(cachedNameService, "Cached");
    calculate(cachedNameService, "Cached");
}

public void calculate(NameService teacher, String value) {
    long time = System.currentTimeMillis();
    nameService.getNames();
    long elapsedTime = System.currentTimeMillis() - time;
    System.out.println(value + ": " + elapsedTime + "ms");
}
```

```
Uncached: 3001ms
Uncached: 3000ms
Cached: 3031ms
Cached: 3ms
```

InfoSupport Solid Innovator

Exercise:

- Create an application
- Create a class with a slow method
- Create a class that calls the slow method multiple times
- Implement caching

Asynchronous

Why?

- We have two separate tasks. One takes 2 seconds, the other one 3 seconds.
- Executing them synchronous will take at least **5 seconds**.
- Executing them asynchronous will take at least **3 seconds**.

POM dependencies

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

SleepService

```
@Component
public class SleepService {

    @Async
    public void sleep() throws InterruptedException {
        Thread.sleep(3000L);
    }
}
```

Asynchronous example

```
@SpringBootApplication
@EnableAsync
public class App implements CommandLineRunner {
    @Autowired
    private SleepService sleepService;

    public void run(String... strings)
            throws InterruptedException {
        long startTime = System.currentTimeMillis();
        sleepService.sleep();
        sleepService.sleep();
        long endTime = System.currentTimeMillis();
        long time = endTime - startTime;
        System.out.println("Elapsed time: " + time);
    }
}
```

Time to run the application

Synchronous

Elapsed time: 6001

Asynchronous

Elapsed time: 21

A future for your result

SleepService

```
@Component
public class SleepService {

    @Async
    public Future<String> sleep() throws InterruptedException {
        Thread.sleep(3000L);
        return new AsyncResult<String>("OK");
    }
}
```

Future example

```
public void run(String... strings) throws InterruptedException,
        ExecutionException, TimeoutException {
    long startTime = System.currentTimeMillis();
    Future<String> result1 = sleepService.sleep();
    Future<String> result2 = sleepService.sleep();

    System.out.println(
        "Result 1: " + result1.get(5000L, TimeUnit.MILLISECONDS))
    System.out.println(
        "Result 2: " + result2.get(5000L, TimeUnit.MILLISECONDS))

    long endTime = System.currentTimeMillis();
    long time = endTime - startTime;
    System.out.println("Elapsed time: " + time);
}
```

```
Result 1: OK
Result 2: OK
Elapsed time: 3040
```

Exercise:

- Create an application
- Create two slow methods
- Display the time it takes to call the two slow methods
- Make the two methods asynchronous
- Display the time it takes to call the two asynchronous methods
- Implement the methods with Futures to retrieve the return values

Info Support
Solid Innovator

Scheduling

Info Support
Solid Innovator

POM dependencies

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

```
@SpringBootApplication
@EnableScheduling
public class App {
    private static final SimpleDateFormat df =
            new SimpleDateFormat("HH:mm:ss");

    @Scheduled(fixedRate = 5000)
    public void fixedRateSchedule() {
        System.out.println("5s: " + df.format(new Date()));
    }
    // 10 min past the hour between 9 and 17 o'clock from MON-FRI
    @Scheduled(cron="0 10 9-17 \* \* MON-FRI")
    public void cronSchedules() {
        System.out.println("Cron: " + df.format(new Date()));
    }
```

Exercise:

- Create an application
- Create a method that shows something based on a schedule

JMS

JMS

- Interested parties can subscribe to message topic
- Sender sends messages to a specific topic
- Sender does not know the receivers
- Comparable to e-mail newsletters
- Implemented with a message broker such as ActiveMQ
- Messages can be persisted by the broker

Why JMS?

- Decoupled sending and receiving of messages
- Possible to add/remove receivers without modifying the sender
- Easily send messages between various systems

POM dependencies

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-activemq</artifactId>
</dependency>
<dependency>
    <groupId>org.apache.activemq</groupId>
    <artifactId>activemq-broker</artifactId>
</dependency>
<dependency>
    <groupId>com.fasterxml.jackson.core</groupId>
    <artifactId>jackson-databind</artifactId>
</dependency>
```

Message

```
public class Message {
    private String content;
```

JMS setup

```
@SpringBootApplication
@EnableJms
public class Application {
    @Bean
    public JmsListenerContainerFactory<?> myFactory(
            ConnectionFactory connectionFactory,
            DefaultJmsListenerContainerFactoryConfigurer
            configurer) {
        DefaultJmsListenerContainerFactory factory =
            new DefaultJmsListenerContainerFactory();
        configurer.configure(factory, connectionFactory);
        return factory;
    }
```

Message conversion

```
@Bean
public MessageConverter jacksonJmsMessageConverter() {
    MappingJackson2MessageConverter converter =
        new MappingJackson2MessageConverter();
    converter.setTargetType(MessageType.TEXT);
    converter.setTypeIdPropertyName("_type");
    return converter;
}
```

Sending a JMS message

```
public static void main(String[] args) {
    ConfigurableApplicationContext context =
        SpringApplication.run(Application.class, args);
    JmsTemplate jmsTemplate = context.getBean(JmsTemplate.class);

    jmsTemplate.convertAndSend("messagebox",
        new Message("Hello"));
}
```

Receiving a JMS message

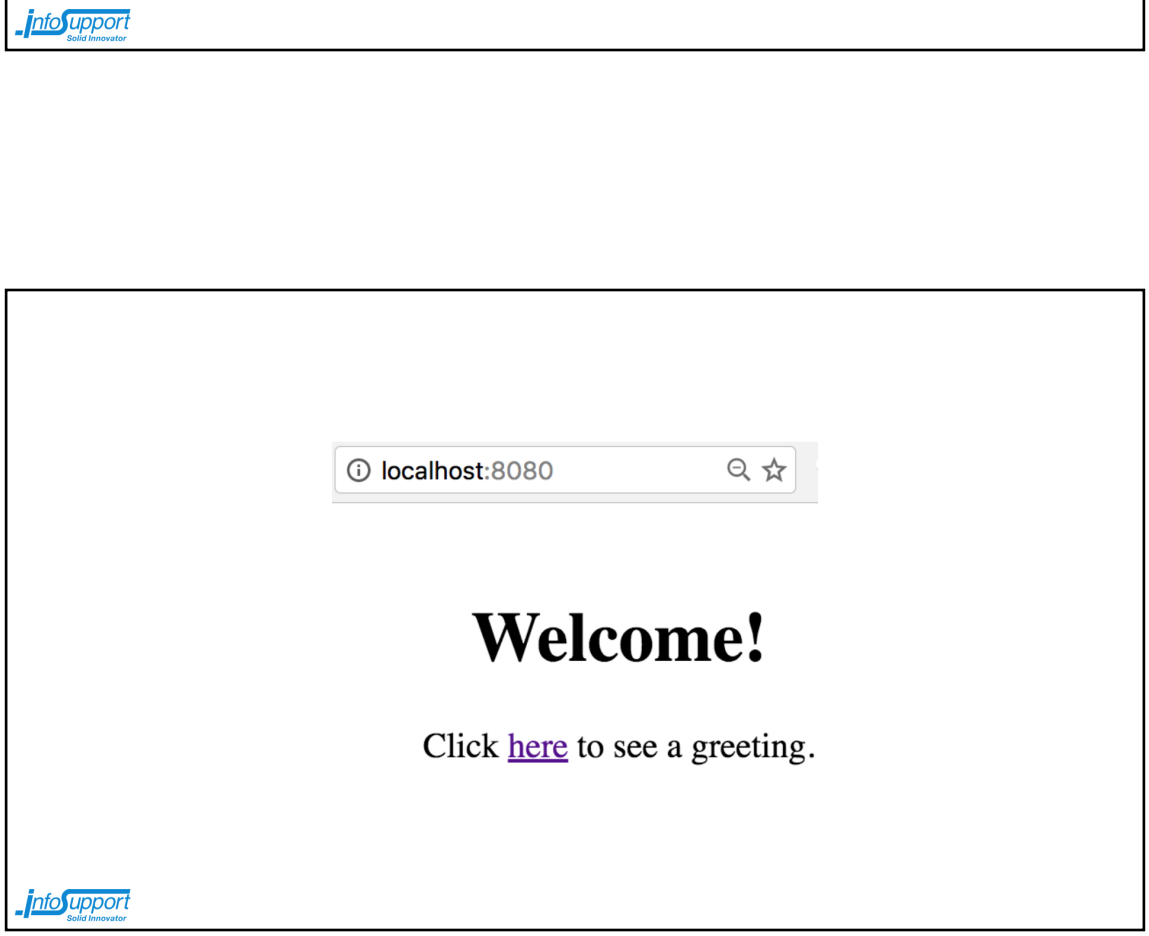
```
@Component
public class MessageReceiver {
    @JmsListener(destination = "messagebox")
    public void receiveMessage(Message message) {
        System.out.println("Received message: " + message);
    }
}
```

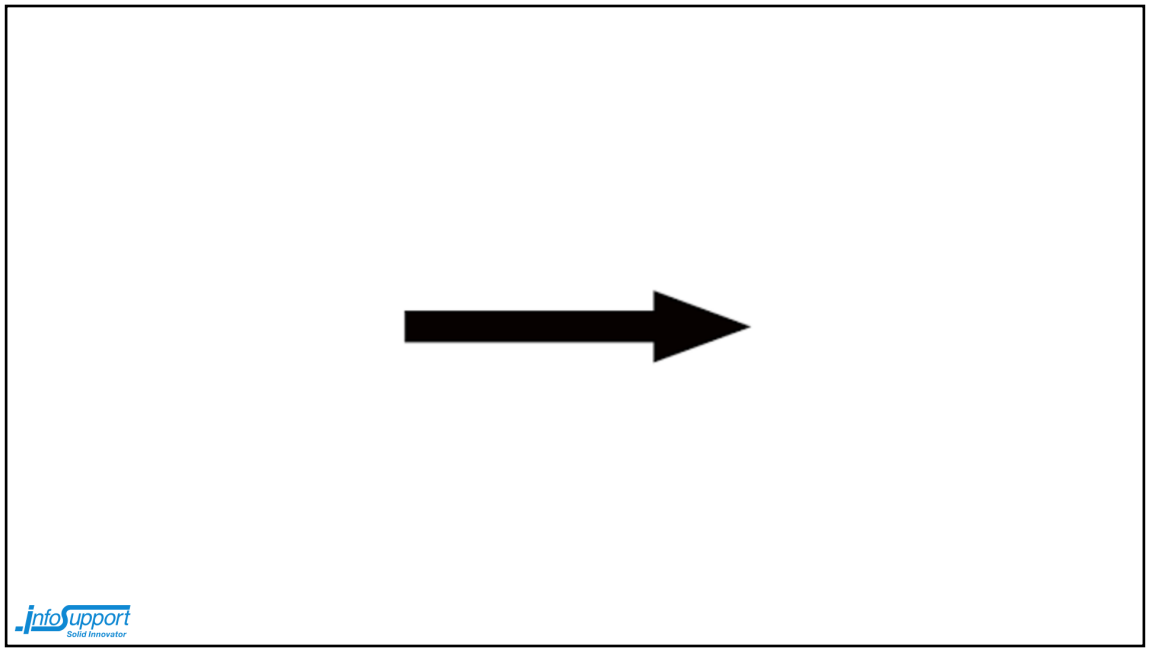
Exercise:

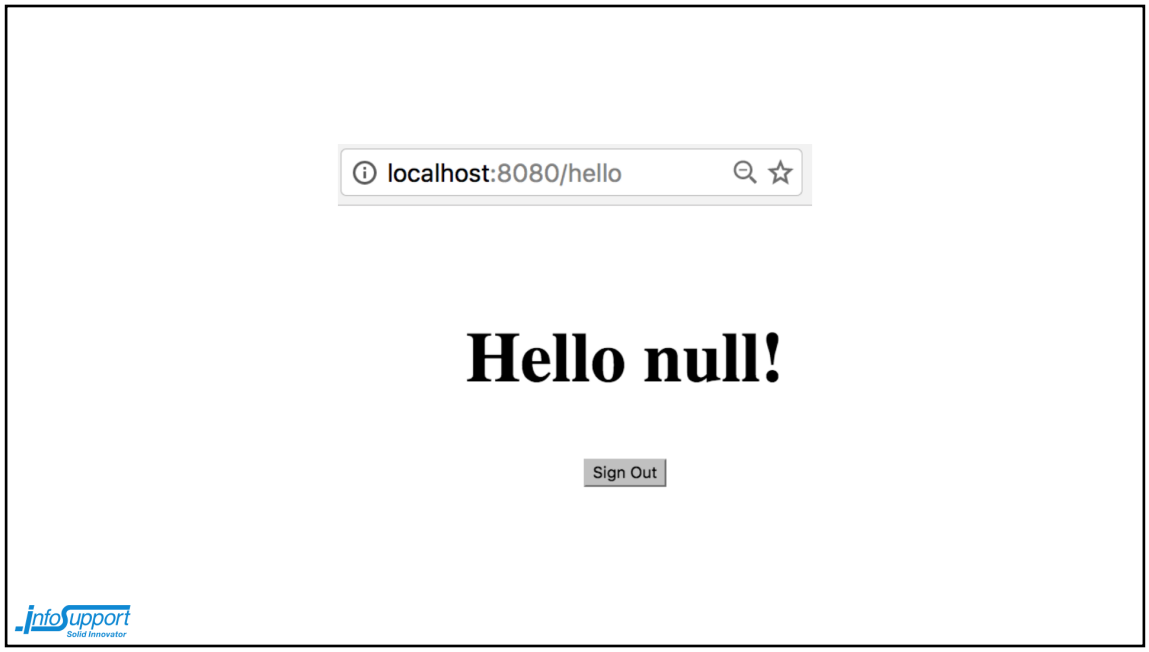
- Create an application
- Create a JMS listener
- Send a message to the listener

Spring Security

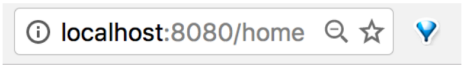
Security Disabled





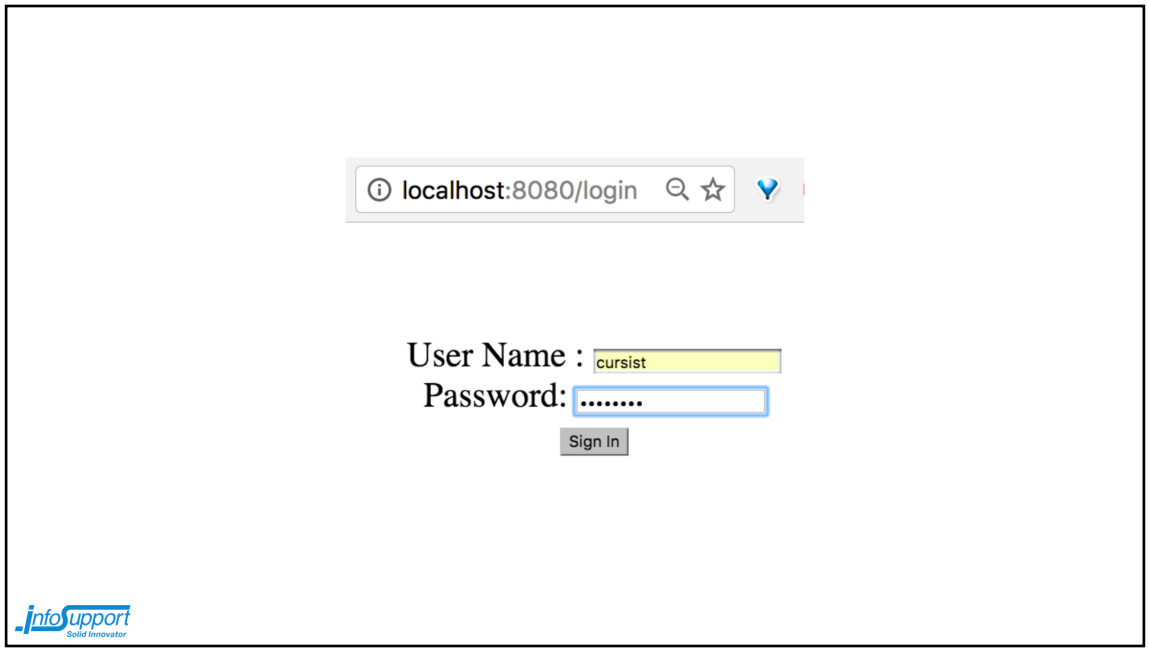


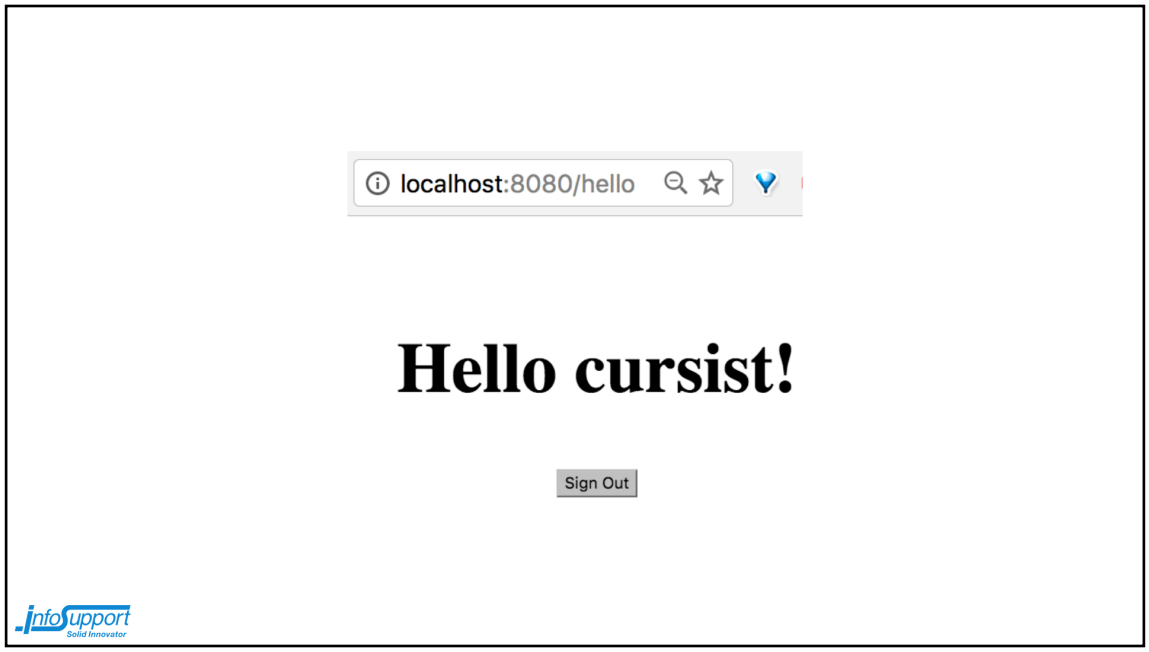
Security Enabled

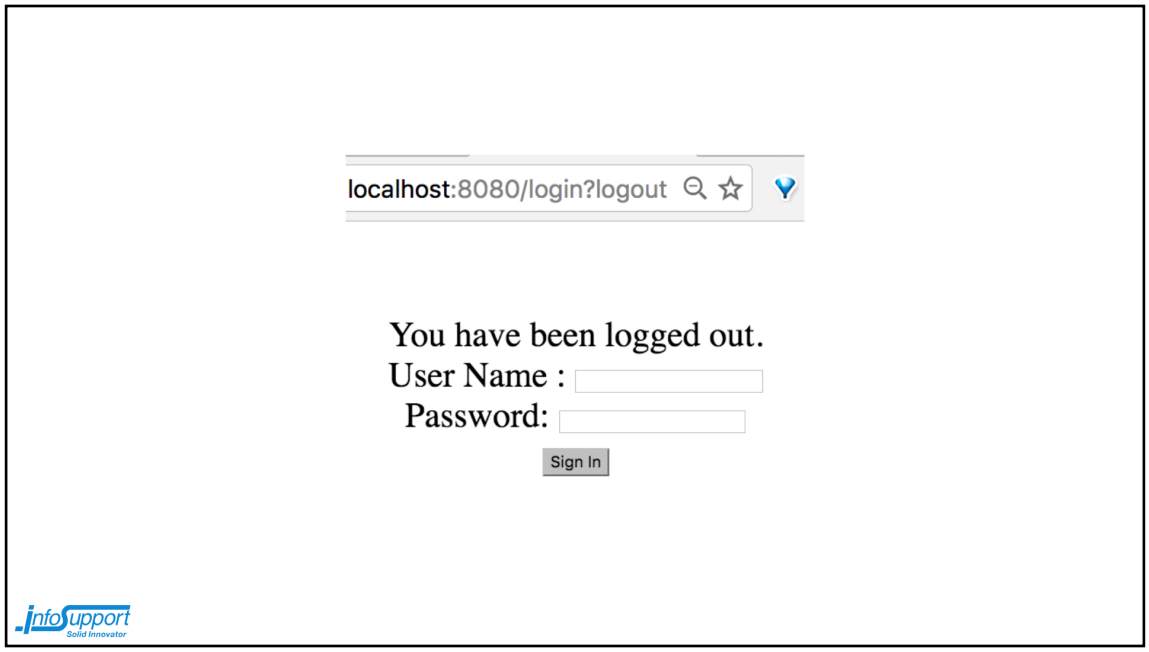


Welcome!

Click here to see a greeting.







What is Security?

Two problems to solve:

- Authentication -> who are you?
- Authorisation-> what are you allowed to do?

Authentication

Authentication is done by the AuthenticationManager

```
public interface AuthenticationManager {

    Authentication authenticate(Authentication authentication)
                                throws AuthenticationException;
}
```

- returns an Authentication instance -> successful authentication
- throws an exception if an invalid authentication attempt is perceived

The Authentication object

```
public interface Authentication extends Principal, Serializable {
    Collection<? extends GrantedAuthority> getAuthorities();
    Object getCredentials();
    Object getDetails();
    Object getPrincipal();
    boolean isAuthenticated();
    void setAuthenticated(boolean isAuthenticated)
        throws IllegalArgumentException;
}
```

Consider the Authentication Object as a Principal where additional information about authentication can be retrieved

Authentication against a JDBC realm

```
@Autowired
public void configAuthentication(
    AuthenticationManagerBuilder auth) throws Exception {
  auth.jdbcAuthentication().dataSource(dataSource)
  .usersByUsernameQuery(
  "select username,password, enabled from users where username=?"
  .authoritiesByUsernameQuery(
  "select username, role from user_roles where username=?");
}
```

InfoSupport Solid Innovator

Authorization

Authorization is delegated to the AccessDecisionManager

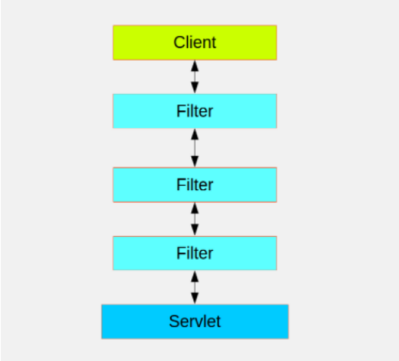
```
public interface AccessDecisionManager {
    void decide(Authentication authentication, Object object,
        Collection<ConfigAttribute> configAttributes)
            throws AccessDeniedException,
                InsufficientAuthenticationException;
}
```

- authentication -> principal
- object -> secure object decorated with config objects
- configAttributes -> metadata related to permissions

InfoSupport Solid Innovator

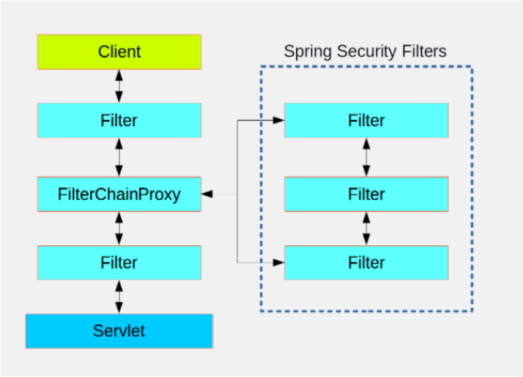
Web Security

Web security is based on Servlet Filters



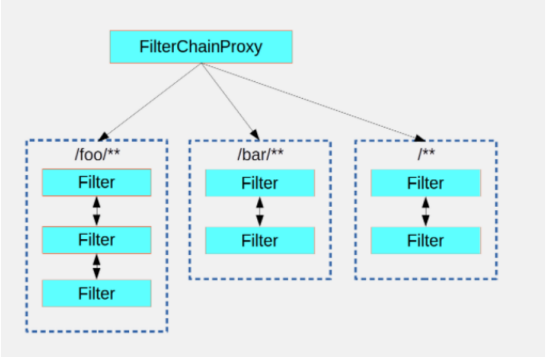
Note the order of a filter can be very important!

Spring security is implemented as one filter in the chain



One physical filter is a chain of internal filters

Multiple filter chains can exist



A default Spring Boot application with Security enabled

- Usually 6 filter chains are active
- 5 chains are there to ignore static resource patterns (/css/** ,/images/** and /error)

A default Spring Boot application with Security enabled

- The last chain matches the catch all path /** ,contains logic for
  - authentication
  - authorization
  - exception handling
  - session handling
  - header writing

Securing with Spring Boot

- Boot automatically secures all HTTP endpoints with "basic" authentication if Spring Security is on the classpath
- This behaviour can be adjusted further

POM dependencies

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

Spring Boot application

```
@SpringBootApplication
public class RESTSecureApplication {
    public static void main(String[] args) {
        SpringApplication.run(RESTSecureApplication.class, args);
    }
}
```

Configuring security part 1

```
@Bean
@Override
public UserDetailsService userDetailsService() {
    UserDetails user = User.withDefaultPasswordEncoder()
        .username("user").password("user1")
        .roles("USER").build();
    UserDetails admin = User.withDefaultPasswordEncoder()
        .username("admin").password("admin1")
        .roles("ADMIN").build();

    return new InMemoryUserDetailsManager(user, admin);
}
```

Configuring security part 2

```
@Override
protected void configure(HttpSecurity http) throws Exception {
    http.authorizeRequests()
        .antMatchers("/admin/**").hasRole("ADMIN")
        .antMatchers("/user/**").hasRole("USER")
        .and()
        .formLogin()
        .and()
        .logout().logoutRequestMatcher(
            new AntPathRequestMatcher("/logout"))
        .logoutSuccessUrl("/login");
}
```

Use http://localhost:8080/logout to logout

RestController

```
@RestController
public class RestSecureController {
    @RequestMapping("/user/hello")
    public String helloUser() {
        return "Hello user";
    }

    @RequestMapping("/admin/hello")
    public String helloAdmin() {
        return "Hello admin";
    }

    @RequestMapping("/hello")
    public String hello() {
        return "Hello nobody";
    }
```

Exercise:

- Create an application
- Create a REST endpoint
- Secure the REST endpoint

Conclusion

Interesting books

- Spring Boot in Action
- Spring in Action
- Pro Spring Boot
- Pro Spring

Disclaimer: check if the content of the book matches your experience and if it's still up to date.

Interesting websites

- https://spring.io/guides
- https://spring.io/docs

Conclusion

- Spring (Boot) is quite easy to use
- Everything works with and without Spring Boot
- Once you know the basics it's relatively easy to learn knew parts of Spring
- Spring (Boot) enables you to focus on implementing business functionality

Questions?

- You can always contact me / Info Support