Core Spring 5.x Essentials

TT3325: Student Guide

Trivera Technologies LLC

Collaborative IT Training, Mentoring & Courseware Solutions

Trivera Technologies LLC is a premier provider of **IT education and training, structured mentoring programs, custom courseware design & licensing** and **specialized education** services and solutions. Our extensive team of subject matter experts, engaging instructors and courseware designers brings years of current, practical, real-world experience into every classroom.

Our goal is to make *you* a success. Whether you're a manager arranging training for your company, a trainer or training firm using our courseware materials at your valued client site, or a student attending a course presented by one our experts, **our core mission is to develop and deliver exactly the program you need to achieve your goals**, held to the highest industry quality standards, with complete support and assistance before, during and after your event.

We are a skills-oriented services firm that offers introductory through advanced level training in hundreds of IT topics. Our courses are fully customizable to suit your unique requirements, and can presented live in your classroom, virtually for private delivery for your team, or in public training courses available around the globe. We **wholly-own** the majority of the courses we teach, providing us with the ultimate flexibility to create the most effective, targeted programs. Our expert delivery and management team services your event from end to end, producing and delivering targeted events with limited burden on your own firm, providing you with the most value for your training dollar.

Trivera Technologies Services Include

- **World Wide Instructor Led Training offered Onsite, Online, Self-Paced & Blended Solutions**
- **Extensive Open Enrollment Public Schedule**
- **Collaborative Mentoring Programs, Skills Immersion Camps & Code Academy Programs**
- **Skills Assessment Services; Gap Training Solutions and Corporate Training Plans**
- **Corporate Training & Vendor Management Solutions**
- **Custom Course Development Services**
- **Courseware Licensing – Corporate or OnDemand**
- **Flexible Pricing Programs and Options**
- **Satisfaction Guaranteed Services**

Trivera Technologies

Educate. Collaborate. Accelerate!

Training • Mentoring • Courseware • Programs
In-Person • Online Live • Public Schedule • Custom

Training@triveratech.com | +609.647.7572 phone
www.triveratech.com

Trivera tech.ed

100% Woman-Owned Small Business

A Sample of our Course Offerings includes....

> Application Development, Programming & Coding: Intro to Programming • Java / JEE • C # / .Net • ASP.Net • VB.Net • C++ • C • COBOL • R • Scala • SQL • Cloud • Spring • Spring Batch / Boot / Security / Cloud • Hibernate • JSF • Struts • Secure Coding • Microservices • Web Services • MVC • RESTful Services • Design Patterns • Cloud

> Agile & TDD: Agile Development • TDD / BDD • Unit Testing • Frameworks • Scrum • SAFE • Kanban

> Web Development & Design: HTML5 / CSS3 • JavaScript • JQuery • Angular • React / Redux • MEANStack / MERNStack • Node.JS • UX / UI • BootStrap • XML / XSLT • Responsive Design • Python • Perl • PHP • XML

> Python: Python for Web • Python Networking / Sys Admin • Python Data Science • Python for Security Pros

> Security: CyberSecurity • Secure Software Design • Secure Coding • Secure Web Development • Database Security • OWASP • STIG • CCSK

> Mobile: Android • IOS • PhoneGap • Kotlin • Swift • Mobile Application Testing • Secure Mobile Development

> Big Data / Data Science, AI , Machine Learning, & Deep Learning: Machine Learning • Hadoop Admin • Hadoop Development • Pig / Hive / MapReduce • Scala • Spark • R for Data Science • Data Science • Python for Data Science • AI – Artifical Intelligence

> SOA: SOA • Architecture • Analysis • Design • Governance • Implementation • SOA Security

> Databases: DB Design • DB Security • DB2 • SQL • PLSQL • Oracle • MySQL • SQLServer • MongoDB • NoSQL • MariaDB • Cassandra • Oracle • JDBC

> MainFrame: C • COBOL • DB2 • Assembler • Transition to Web • Java for COBOL

> O/S & SysAdmin: Windows • Linux • UNIX • Mainframe • Z/OS • Administration • IOS

> Oracle: DBA • SQL • PL/SQL • New Features • OBIEE

> Tools • Eclipse • Ant / Maven • Oracle • Oracle BI • IntelliJ • NetBeans • Tableau • Selenium • Cucumber

> DevOps: DevOps • GIT • GITHub • Jenkins • JIRA • Docker

> IT Skills: Agile • Scrum • ITIL • Project Management • Leadership • Soft Skills • End User

> Software Architecture, Design & Engineering: Architecture • Analysis • Requirements • Estimation • Use Cases • UML • OO • BDD • Data Modeling & Design • Software Design

> Software Testing: QA • Test Automation • Unit Testing • TDD • Selenium • Cucumber • Gherkin

---

Please visit www.triveratech.com for the complete course catalog.

Table of Contents

Core Spring 5.x Essentials

Core Spring 5.x Essentials

TT3325

TT3325 Core Spring 5.x Essentials

Version 20191017

Trivera Technologies | Collaborative IT Education Solutions

Training | Mentoring | Courseware
In-Person | Online Live | Self-Paced | Private Onsite | Public Schedule | Custom Programs

About This Course

- **Mastering Spring is a four-day hands-on Spring training course geared for experienced Java developers who need to understand what the Spring Framework is in terms of today's systems and architectures, and how to use Spring in conjunction with other technologies and frameworks.**

Welcome to Core Spring 5.x Essentials

Thank you for choosing us to provide you with the very highest standard of technical training!

We take great pride in our attention to detail in designing the very best quality courseware in the industry.

This course is the result of many developer-months of planning, design, development, and review. Every course is very carefully designed using proper guidelines for instructional technology, including the use of specific performance objectives, relevant laboratory exercises, and professional quality content layout to ensure the most effective transfer of knowledge.

Because we realize that no work is ever perfect, we are constantly striving to improve our materials. Throughout this course you will have ample opportunity to provide us with your feedback on all aspects of this course. Please do take the time to give us your input. It is extremely valuable to us.

Again, many thanks for entrusting us with your educational needs. Enjoy the course!

Workshop Agenda

- Session: Introduction to Spring
  - Lesson: The Spring Framework
    - Tutorial: Setup IntelliJ for Using Maven
  - Lesson: Configuring Spring Managed Beans
    - Exercise: Spring Java Config
  - Lesson: Defining Bean dependencies
    - Exercise: Configuring Bean Dependencies using Annotations
    - Exercise: Creating the Candidate Component Index
  - Lesson: Introduction to Spring Boot
    - Exercise: Introduction to Spring using Spring Boot
  - Lesson: Working with Spring Boot
    - Exercise: Create REST Repository using Spring Boot

Workshop Agenda (cont'd)

- Session: Spring AOP
  - Lesson: Introduction to Aspect Oriented Programming
  - Lesson: Spring AOP
    - Exercise: Spring AOP: Adding Interceptors
- Session: Persistence in Spring
  - Lesson: Transaction Management in Spring
  - Lesson: Spring JDBC
    - Exercise: Using Spring JDBC
- Session: Spring Data (Introduction)
  - Lesson: Spring Data Overview
    - Exercise: Spring Data JPA Using Spring Boot
    - Exercise: Spring Data JPA Using Spring Boot (Part 2)

Workshop Agenda (cont'd)

- Session: Spring Data (Introduction) (cont'd)
  - Lesson: Spring Data Overview (cont'd)
    - Exercise: Spring Data JPA (Without Spring Boot) (Optional)
  - Lesson: Spring Data Query Methods
    - Exercise: Spring Data Query Methods
- Session: Implementing REST with Spring
  - Lesson: REST principles
  - Lesson: Introduction to RESTful Services in Spring
    - Exercise: Working with Spring REST
  - Lesson: Introduction to REST Clients in Spring
    - Exercise: Implementing the Spring REST Client
  - Lesson: Bootstrapping the REST application

Workshop Agenda (cont'd)

- Session: Implementing REST with Spring (cont'd)
  - Lesson: Bootstrapping the REST application (cont'd)
    - Exercise: Spring Hotel Reservation
    - Exercise: Spring Hotel Reservation Client
  - Lesson: Content Representation
    - Exercise: Spring REST Content Negotiation
  - Lesson: Implementing the REST Service
    - Exercise: Spring REST Services
- Session: Spring Security Framework
  - Lesson: Enterprise Spring Security
  - Lesson: Spring Web Security (JavaConfig)
    - Exercise: Using Spring Web Security

Additional Topics: Time Permitting

- Lesson: Introduction to Spring MVC
  - Exercise: Using Spring MVC

Formalities

- Introductions
- Logistics (breaks, facilities, lunch, etc.)
- Rules of Classroom Engagement
- Let's Get Started!

Student Introductions

- Name
- Company
- Role
- Previous projects of interest
- Current project
- Languages/Areas of expertise
- Why are you taking this workshop?

Getting to Know You

The best way for your instructor to help you in learning is for him/her to understand a little bit about your background. When it's your turn, please don't be shy! Speak up so everyone in the class can hear you. You will probably find that there is at least one other person in the class with a background and set of goals similar to your own. Much of the value in taking a class is the networking that can occur between developers.

When each of the class members have had the opportunity to talk about their background and goals, the instructor will spend a few minutes sharing his/her own background, including qualifications for teaching this course.

Questions before we begin?

Ready?

Before moving on into the course contents take a moment to digest everything you've seen so far. Feel free to ask your instructor any questions that will make you feel more comfortable with the class.

Session: Introduction to Spring

The Spring Framework
Configuring Spring Managed Beans
Defining Bean dependencies
Introduction to Spring Boot
Working with Spring Boot

The Spring Framework

The Spring Framework
Configuring Spring Managed Beans
Defining Bean dependencies
Introduction to Spring Boot
Working with Spring Boot

Lesson Agenda

- Understand the value of Spring
- Explore Dependency Injection (DI) and Inversion of Control (IoC)
- Introduce different ways of configuring collaborators
- Spring as an Object Factory
- Initializing the Spring IoC Container

Introduction to Spring

- Spring is an open source application framework
  - Eases the development of enterprise applications
- Provides services for use in Java SE and Java EE applications
- Light-weight non-invasive framework
  - Applies to all architectural tiers of a enterprise application
- Focuses on the use of Plain Old Java Objects

Spring is an open source application framework, primarily focused on easing the development of Java EE development. Although the framework provides functionality that can also be used within Java SE applications (e.g. JDBC Helper

classes), one of the goals of the Spring framework is to provide the developer with a consistent framework that can be applied to all architectural tiers of a Java EE application.

Spring provides a consistent, simple programming model in many areas, making it an ideal architectural 'glue'. You can see this consistency in the Spring approach to JDBC, JMS, JavaMail, JNDI and many other important APIs.

The Spring framework can be used on top of APIs to ease the development and testing of large Java applications.

The changes needed to implement applications within the Spring framework should be minimal. In most cases, classes that rely on the Spring framework do not have to implement any additional interfaces nor subclass any Spring specific super classes to make them function.

Within a Spring based application, services are provided to components through the use of dependency injection. Replacing the service with a mocked implementation takes place in a configuration file, without the need to change the code that has to be tested. This makes Spring a perfect candidate for test driven development.

Spring allows developers to use basic Java objects to implement their business logic. Services like security and transactions, which are normally supplied by an EJB container, can now be provided to the component by the Spring framework (e.g. Using Aspects and Proxies).

The Benefits of Spring

- Makes code more maintainable and flexible
- Uses POJOs for business logic
  - Most code doesn't require Spring dependencies
- Get many Enterprise features without need for a server
  - Sometimes called a ”lightweight container"
- Spring eliminates "boilerplate" code
- Dependency injection leads to a uniform coding style
  - Facilitates maintenance and testing with JUnit

The benefits of Spring are in maintainability and flexibility of code. Code that leverages Spring tends to be shorter, more flexible and more reusable.

An example of getting rid of "boilerplate" code is when using JDBC. You typically write code for creating Statement objects, and closing Connections, Statements and ResultSets. This code is repetitive and must deal with exception handling for exceptions that rarely ever occur.

Goals of the Spring Framework

- Non-invasive
  - Application code is independent of the framework
  - Selectively apply Spring to existing application
  - Dependencies on Spring should be minimal
- Usable in any environment
  - With or without a Java EE container
  - With or without JNDI
  - With JDBC, JTA, Hibernate or other transaction models
- Facilitate good coding practices
  - Encourage coding with interfaces instead of classes

Spring is capable of using regular Java constructs for dependency injection (e.g. constructors and setter methods). The application code does not have to implement any Spring specific interfaces and/or extend any specific classes. Therefore the application code can still be used outside of the Spring framework without making any changes to the code. Until you start using any of Spring's template classes, the dependencies on Spring are minimal (or don't exist at all). Your code will still function correctly even when you do not want to use the Spring framework.

Spring allows for the development of applications that run within a variety of environments. Resources, needed by the application, are configured in Spring's configuration file. Whether these resources are obtained from JNDI or otherwise provided by an application server, they are configured using one of the Spring classes and/or frameworks. Therefore the code does not have to be altered when the environment changes.

Spring facilitates good coding practices, especially since it is based upon the principle of coding against interfaces instead of classes. When referencing components through their interface, the Spring framework can be used to inject any class that implements this interface. Again this is one way of facilitating a test-driven environment. Objects can be mocked during development ans easily replaced by the 'real-thing', by making a slight change to the configuration file.

Goals of the Spring Framework (cont'd)

- Promote pluggability
  - Injected services can be replaced easily
  - Annotations, Java Code, XML, Groovy or property files configure applications
  - Result - maintainable and reusable components within an "integration" framework
- Applications are easy to test
  - Most objects are POJOs
  - Easy use of mock objects through dependency injection
- Provide a consistent framework
  - Consistent approach is used across different parts of framework

Services are configured within the Spring configuration file and injected into the client component. A service itself might have dependencies on other services or resources which, in turn, can also be configured within the Spring configuration file. This way of wiring the services together and assembling the final application allows for a really pluggable architecture.

Because services and their relationships are configured within the configuration file, it becomes very easy to replace part of the application by a mocked implementation.

When it comes to Spring, you will find that a consistent approach has been taken throughout the different parts of the Spring framework. Once you have become familiar with the Spring configuration file and the use of dependency injection, configuring other parts of the Spring framework will be pretty straightforward..

(Some) Key Features of Spring

- Inversion of Control container (IoC)
  - Configuration management of POJOs
  - Allows for Dependency Injection (DI)
- Aspect-Oriented Programming (AOP) Framework
  - Services are modularized (out-of-the-box or custom)
  - Services are applied to application in a declarative manner
- Data access abstraction
  - Consistent architectural approach to data access
  - Independent of particular persistence products
  - Simplifies use of JDBC and other data access APIs
- Transaction abstraction
  - Consistent model for a variety of transaction services: JDBC, JTA, Hibernate, etc...
- ...and much, much more

IoC allows you to create an application context where you can construct objects, and then pass to those objects their collaborating objects. As the word inversion implies, IoC is like JNDI turned inside out. Instead of using a tangle of abstract factories, service locators, singletons, and straight construction, each object is constructed with its collaborating objects. Thus, the container manages the collaborators.

AOP allows developers to create non-behavioral concerns, called crosscutting concerns, and insert them in their application code. With AOP, common services like logging, persistence, transactions, and the like can be factored into aspects and applied to domain objects without complicating the object model of the domain objects.

Spring provides a variety of classes, interfaces and configuration objects for all of the tiers in both Java SE and Java EE application. Ranging from helper classes for developing Enterprise JavaBeans to JdbcTemplate classes to provide easy access to JDBC data sources. A number of Template classes are available to ease the development of data access classes. Although the Spring framework is mainly based on an IoC container, most of these classes can also be used to develop data access classes without this container.

Whether you use JDBC, Hibernate or JDO, Spring provides a consistent approach for accessing your data, independent of the actual persistence product sitting underneath.

Spring provides a number of way to manage transactions within the application, independent of the persistence layer used. By using Spring's transaction configuration framework, applications can be easily configured to make use of JTA of 'simple' JDBC transaction mechanisms, without making changes to the application code.

Plain Old Java Objects (POJOs)

- Plain Old Java Objects do not rely on specific framework
  - Define properties through getters and setters

```
public class FlightServiceImpl implements FlightService {
   private FlightRepository repository;
   public void setFlightRepository(FlightRepository repository) {
      this.repository = repository;
   }
   …
}
```

- Minimizes dependencies on technical-infrastructure APIs
  - Implementations often based on business interfaces

Plain Old Java Objects do not require the implementation of any framework specific interface, they are just following some standard Java naming conventions for the definition of accessor (setter and getter) methods.

POJOs and Interfaces

- Client code can code to interfaces, EXCEPT when instantiating the object
  - Client must then know which object to create
- Objects might require initialization
  - Where should initialization data be defined?
    - Constructor, JNDI, property files, factory classes, …

```
public class FlightService{
   private FlightRepository repository;
   public void setRepository(FlightRepository repository) {
      this.repository = repository;
   }
   public List<Flight> getFlights(String destinationCode, LocalDate date){
      return repository.getFlights(destinationCode, date);
   }
}
```

```
public interface FlightRepository {
   List<Flight> getFlights(String destination, LocalDate date);
}
```

  - Who chooses which dependencies to use?
  - How are dependencies initialized?

Even when the client application communicates with an interface, the configuration of the actual object must still take place. For example, when using a DataSource, the client might not really be aware of the configuration that needed to be done, but information like database URL, username and password must still be set before a DataSource instance is returned.

If database info is in the constructor, then it can not be changed at runtime. If some other code is used, then you have the same problem plus the question of "which other code"?

Using JNDI requires a JNDI environment (usually an application server). Properties files are great – except that they often lead to properties file profusion.

From the perspective of a client, every POJO consists of an interface (the operations that can be invoked) and the properties (accessed through setter and getter methods).

In the example show above, the FlightService has a dependency on a FlightRespository. Before the business method (getFlights) can be invoked, the dependency to the repository needs to be resolved.

Also, for all the methods, you have to consider what if you want to use a mock connection instead of a real one?

Inversion of Control (IoC)

- Also called "Dependency Injection"
  - Objects are provided with the components they rely on
    - Instead of 'looking' for these components
- Spring Framework resolves dependencies
  - Between multiple objects
  - Between objects and configuration parameters
- Dependencies are resolved by framework before business methods are called
  - Framework is responsible for instantiating IoC objects

By using Inversion of Control, configuration of the application is taking place outside of the code. These 'external' settings are then 'pushed' into the component before invoking the business logic.

Without IoC, the object will have to lookup these references themselves. With IoC, the relationships between the components are described in a configuration file. The IoC framework will make sure these references are resolved and made available to the object, before the business methods are called on the object.

Spring provides an IoC container, which will set configuration parameters and resolve dependencies between components, according to the information in the Spring configuration file.

It is the responsibility of the IoC framework to instantiate the objects and wire them together before handing them off to the caller. Objects obtained from the IoC container are completely configured and business methods can be invoked

immediately.

Inversion of Control (cont'd)

- Based on Java language constructs
  - Does not use framework-specific APIs
- Spring supports multiple types of Dependency Injection
  - Constructor, Setter and Method injection
- Method injection allows the framework to implement method at runtime
  - Returns a configured object
- Application classes are self-documenting
  - Relationships are exposed through constructors and methods

One of the goals of Spring is to be non-invasive. In other words, the object should not have to rely on Spring specific classes or interfaces before they can be used by an IoC container. Although Spring does provide a set of interfaces to allow for more advanced use of IoC container features, for 'simple' use, normal Java constructs are solved to push the configuration into the object.

Configuration of the object can be done by the IoC container while creating the object instance. (When invoking the constructor). Parameters can be defined in Spring's configuration and used as parameter values of a constructor. In addition to Constructor parameters, configuration of dependencies can also be injected into the object (by the framework) by invoking one or more setter methods. Last but not least, Spring IoC can 'implement' abstract methods of classes, by using information provided within the Spring configuration

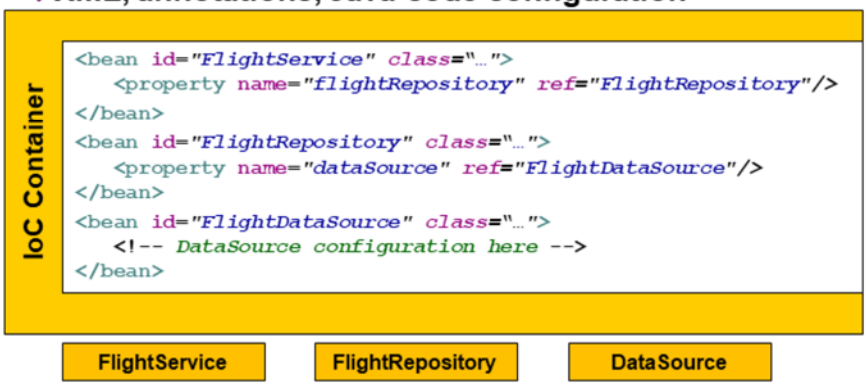
When using Method Injection, the framework will implement an abstract method of a class at runtime. The return value of this method will often be an object that was also configured by the Spring container.

Relationships between components are no longer embedded within lots of Java code. For example, when a JButton object is required within a Swing application, a JButton instance must be created and the properties must be set separately before the button can be added to a Panel or Frame. Using IoC, this same button can be configured completely (a JButton is a proper JavaBean!) and injected through the constructor or a setter method on the object (e.g. setOKButton(JButton okButton))

Without IoC, all dependencies between objects have to be resolved from within the application code.

Advantages of IoC

- Application classes are independent of particular frameworks
  - Framework is 'selected' during configuration
- Framework is responsible for reading configuration and resolving dependencies
  - Configuration can be done in several ways
    - XML, annotations, Java code configuration



Application classes can be developed as POJOs. The configuration of these objects as well as the relationships between these object can be configured later on, when the IoC framework is selected.

The actual configuration of the beans and their dependencies can be done using a variety of technologies. Originally Spring only supported configurations using XML documents. In later releases of the framework support for annotations, Groovy and Java-based configuration were introduced.

Spring is an Object Factory

- Client code asks Spring to provide object

```
ApplicationContext  applicationContext = …
FlightService service = applicationContext.getBean(FlightService.class);
```

- Spring looks up object in a configuration file
  - Defined using Java, XML or Groovy

```
@Configuration
public class JavaConfig {
   @Bean
   public FlightService flightService (){
      return new FlightServiceImpl();
   }
}
```

```
import flight.service.FlightServiceImpl
beans {
   flightService(FlightServiceImpl) {
   }
 }
```

```
<beans>
  <bean id="flightService" class="flight.service.FlightServiceImpl" />
</beans>
```

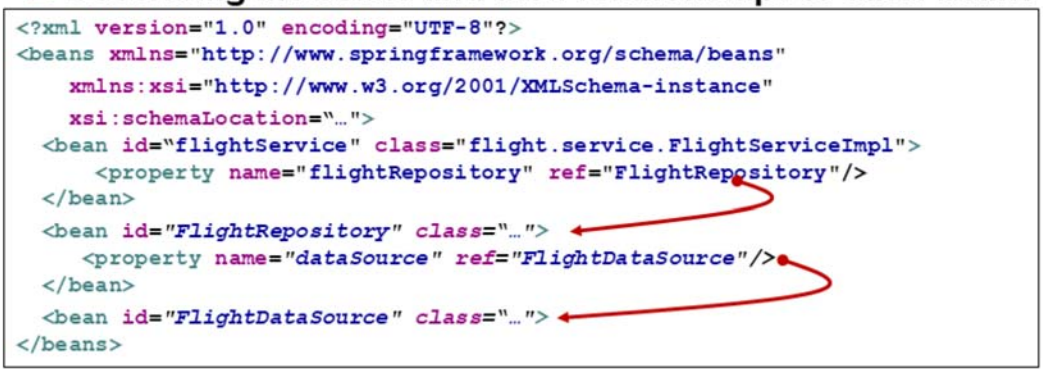
- Spring takes care of initialization of object
  - …and resolves any dependencies

Spring acts as an object factory from which pre-configured instances of beans can be obtained.

The config file used for the configuration of the beans can be changed at development time or at deployment time.

Configuring Spring (XML)

- Traditionally Spring was configured in XML files
  - Describing the beans and their relationships to other beans



- XML-based configuration often considered cumbersome
  - No compile-time type checking
  - Configuration external to beans
  - Developers had to learn different 'language' (XML)

The traditional way to configure Spring is using the XML files.

Configuring Spring (Annotations)

- Spring 3 introduced configuration using annotations
  - Supporting JSR-330 (Dependency Injection for Java)
- Allowing configuration within beans
  - Defining injection at field, method and constructor level
  - Supports life-cycle annotations

```
@Service
public class FlightServiceImpl implements FlightService {
    @Inject @Qualifier("boston")
    private FlightRepository repository;
    @PostConstruct
    public void init() {…}
    …
}
```

```
<beans xmlns="…" xmlns:xsi="…"
    xmlns:context="http://www.springframework.org/schema/context"
    xsi:schemaLocation="…">
 <context:component-scan base-package="com.springclass" />
</beans>
```

- Container must scan packages to find and register beans within the application context

With the introduction of annotations in Java 5, the Spring framework was updated so annotations could be used to configure beans and their dependencies. Using annotations, classes could now be defined as components which are ‘automatically’ loaded into the Spring context. While the core configuration was still done using an XML document, the context could now be configured to scan certain Java packages for classes that are annotated. Once these classes are found, their dependencies (also defined using annotations) would be resolved and the bean would be registered into the context.

The annotations can provide a much simpler configuration mechanism, the biggest issue is that annotations are still in the Java code and not in an external ‘file’

Configuring Spring (Java Config)

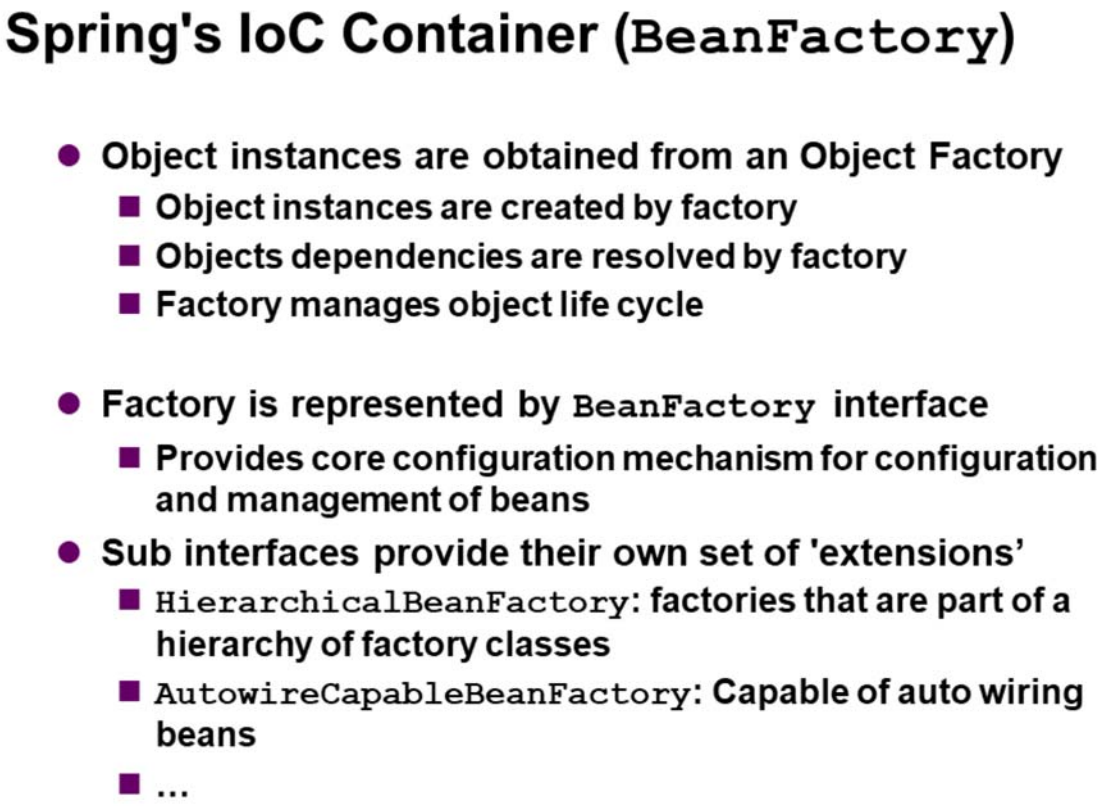
- Spring 3 also introduced Java-based configuration
  - Which has several advantages
    - Type-safety check during compilation
    - Code completion
    - Refactoring support
    - Easy to find bean references within workspace

```
@Configuration
public class JavaConfig {
   @Bean
   public FlightService flightService() {
      FlightServiceImpl service =  new FlightServiceImpl();
      service.setRepository(flightRepository());
      return service;
   }
   @Bean
   public FlightRepository flightRepository() {
      return new BostonFlightRepository();
   }
}
```

Resolving dependency

Spring 3 also introduced Java-Based configuration, allowing for all beans and their dependencies to be defined in Java. Being able to configure the context using Java Code instead of XML meant that not only the type-safety problem of XML was resolved, but the tooling provided by the IDE (like code completion) could now also be used.

Developers who are using Spring (and are programming in Java) no longer need to learn a different 'language' (XML).



Spring's BeanFactory is the IoC factory responsible for configuring the IoC objects and their dependencies. It is responsible for the entire life cycle of the IoC objects. It will create the object instances, resolves their references and remove the objects (closing resources along the way) at the end of their life cycle.

The BeanFactory implementation provides the core mechanism needed to configure and manage the beans within the IoC container. Several sub-interfaces exist, each providing their own set of 'extensions'

org.springframework.beans.factory.HierarchicalBeanFactory: This interface is implemented by bean factories that are part of a hierarchy of factory classes, thereby allowing beans to be loaded by different factories, while still able to reference each other.

org.springframework.beans.factory.ListableBeanFactory: This interface is implemented by bean factories that allow enumeration of bean instances instead of looking up beans by their name. This interface has not been designed to handle large amount of invocations, but rather for maintenance and monitoring purposes.

org.springframework.beans.factory.AutowireCapableBeanFactory: Bean factories implementing this interface are capable of auto wiring bean. In other words, these types of bean factories are capable of resolving dependencies between beans by attempting to set as many properties as possible, by using introspection and reflection.

Spring's IoC Container (ApplicationContext)

- ApplicationContext is a sub-interface of BeanFactory
  - Provides additional functionality
    - Automatic recognition and usage of pre- and post-processors
    - MessageSource provides support for localization of messages
    - Support of framework or application events and listeners
    - ResourceLoader used for handling low-level resources
- ApplicationContext is most often used
  - BeanFactory can still be used for lightweight applications

The ApplicationContext interface is a sub interface of the BeanFactory. It provides a large set of additional functionality:

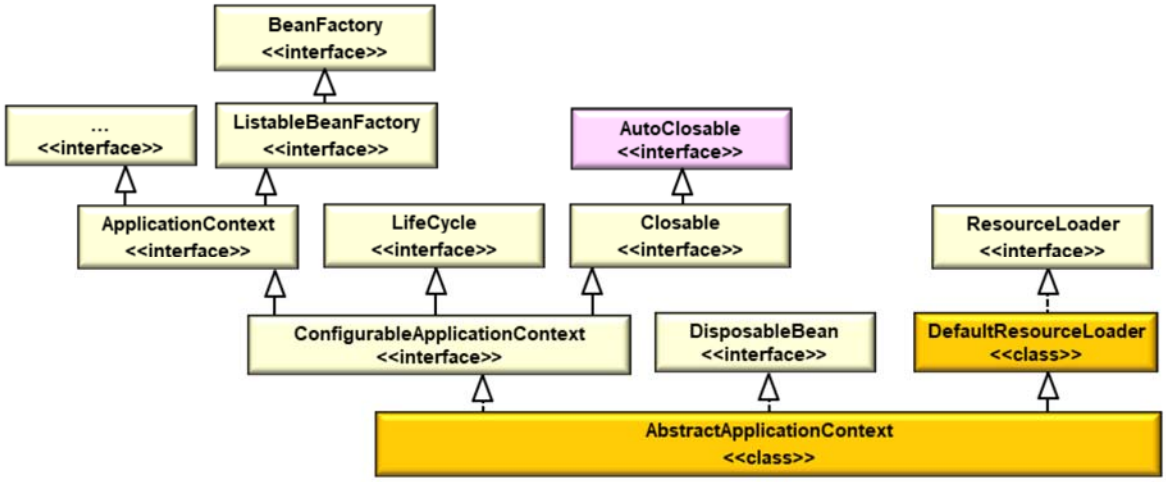
- Pre- and post-processors can be registered with the application context, which are then called upon during the life cycle of IoC beans.
- Capable of resolving messages from resource files, supporting internationalization
- Allows the registration of listeners that will be notified about application events that occur within the ApplicationContext.

Since ApplicationContext provides a great number of additional functions on top of the core functionality provided by the

BeanFactory, most applications will make use of the ApplicationContext implementations.

AbstractApplicationContext

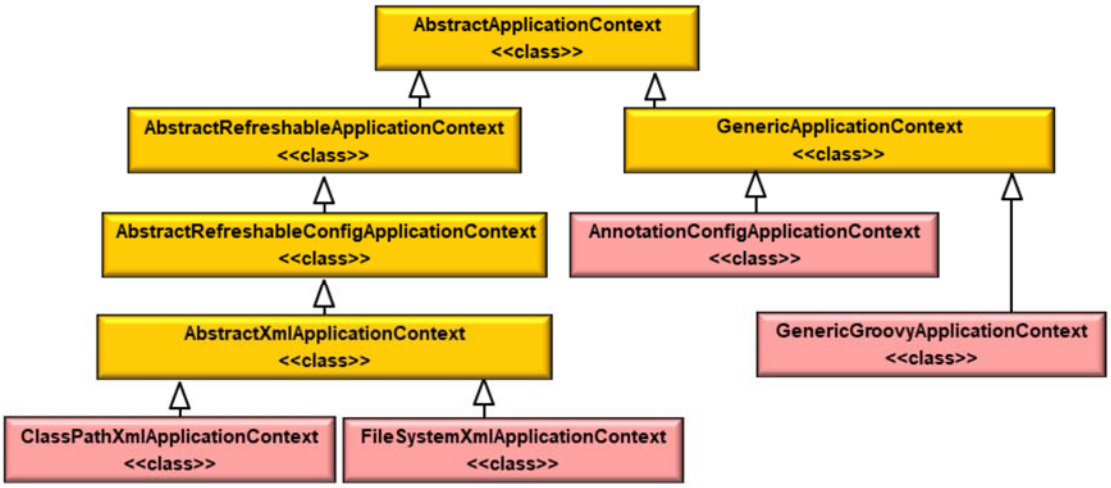
- Provides abstract implementation of `ApplicationContext`
  - Does not define type used for configuration
  - Implements common context functionality



The AbstractApplicationContext is an implementation of the ApplicationContext interface. It does not define how the beans are being defined (XML, Groovy, Java), but only implements the most common context functionality

Spring Container Hierarchy

- Different context implementations are available
  - Each supporting a different type of configuration
    - Java-Config, XML, Groovy



Where the AbstractApplicationContext defines the common functionality, several subclasses are available to allow for the different configuration types.

The implementation you are using depends (naturally) on the type of configuration that you will be using the bootstrap the Spring context.

Initializing the Container

- `ApplicationContext` can be initialized in various ways
  - Pointing to a Java Code configuration(s)

```
ApplicationContext context =
   new AnnotationConfigApplicationContext(Config.class, MoreConfig.class);
```

  - Pointing to XML configuration resource(s) on the classpath

```
ApplicationContext context =
   new ClassPathXmlApplicationContext("beans.xml","repositories.xml");
```

  - Loading Groovy based configuration resource(s)

```
ApplicationContext context =
    new GenericGroovyApplicationContext("context.groovy");
```

In order to initialize the ApplicationContext, you need to need to provide implementations of this interface with references to the configuration.

When using a Java-based approach, you would (for example) use an instance of AnnotationConfigApplicationContext, while the ClassPathXmlApplicationContext would be used when the core configuration of the context was done in XML

Shutdown the Context

- In non-web applications, context should be shutdown
  - Properly releasing resources
  - Allows for lifecycle methods to be called
- Shutdown methods are defined by `ConfigurableApplicationContext` interface
- Context can be shutdown
  - By invoking close method

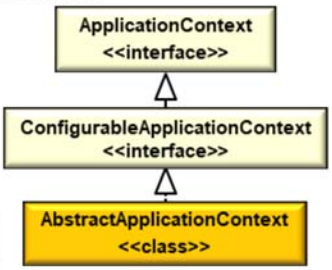
```
abstractApplicationContext.close();
```

  - By registering shutdown hook

```
abstractApplicationContext.registerShutdownHook();
```

  - By creating context within try-with-resources
    - `AbstractApplicationContext` implements `AutoClosable`

```
try (AbstractApplicationContext applicationContext =
                new ClassPathXmlApplicationContext("beans.xml");) { … }
```



With a web application, the context will be automatically shutdown when the application is taken out of service. In a standalone application, the context should be properly shutdown. Not only does this properly release any references the context might have on resources, it also allows the context to call any life-cycle (destroy) methods that might have been developed on the beans that are being managed by this context.

The context can be shutdown explicitly by invoking the close method, or by registering a shutdown hook. With the introduction of try-with-resources block in Java 7, the context can also be created within one on these try-with-resources blocks, as long is the type that is being created is a (sub-)type of AbstractApplicationContext. It is the AbstractApplicationContext class which implements the AutoClosable interface that is required for an object to be instantiated within a try-with-resources block.

Accessing Beans in Container

- `BeanFactory` defines several methods for obtaining beans

org.springframework.beans.factory. BeanFactory
`Object getBean(String name)`
`T getBean(Class<T> requiredType)`
`T getBean(String name,Class<T> requiredType)`
`Object getBean(String name,Object... args)`
`T getBean(Class<T> requiredType,Object... args)`

  - Objects should be referenced by type
    - Generics avoid need for down-casts

```
AbstractApplicationContext context = …
// Only single instance of FlightService has been configured
FlightService service = context.getBean(FlightService.class);
// Multiple instances of TravelService configured,
// specify (unique) bean name
TravelService travelService =
     context.getBean("localService", TravelService.class);
```

It is the BeanFactory interface (supertype of ApplicationContext) which defines the methods that allow us to obtain beans from the context.

Notice that several overloaded methods have been defined. Beans can not only be obtained by their name (a String value) but also by their type. When beans are being obtained by their type, it might result in multiple instances becoming eligible to be returned (e.g. several implementations of the same interface). To avoid such problems, methods have been defined which accept both name and type.

The versions taking variable arguments allow us to pass in constructor or factory method arguments.

Tutorial: Setup IntelliJ for Using Maven

Please refer to the written lab exercise and follow the directions as provided by your instructor

Configuring Spring Managed Beans

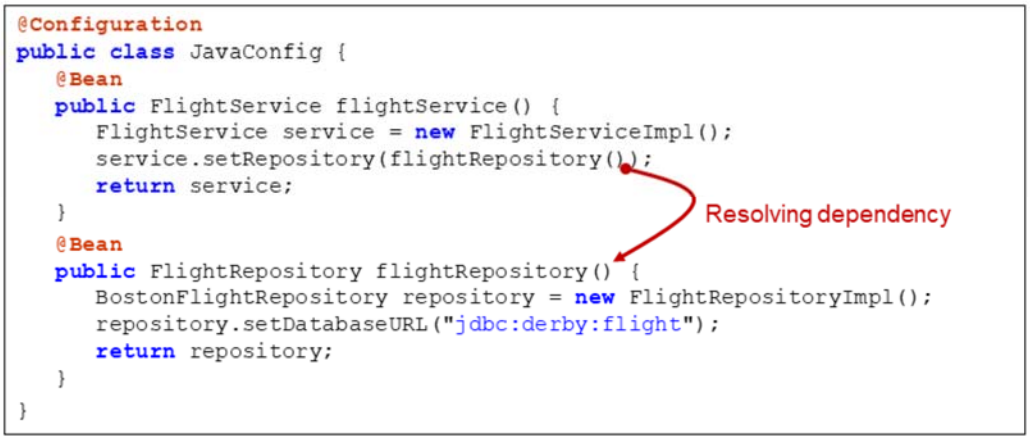
The Spring Framework
Configuring Spring Managed Beans
Defining Bean dependencies
Introduction to Spring Boot
Working with Spring Boot

Lesson Agenda

- Introduce Java-based configuration
- The `@Configuration` and `@Bean` annotations
- Define bean dependencies
- Bootstrapping Java Config context
- Injection in configuration classes
- Using context profiles
- Conditionally loading beans and configurations
- Bean life-cycle methods

JavaConfig Bean Configuration

- Spring supports Java-based configuration
  - Instead of (or in combination with) XML-based configuration
- Java-Code config makes configuration type-safe
  - Configuration is checked at compile-time



Spring 3 provided developers with a new way of defining the Spring context: Java-based configuration. Not only does Java-Code configuration add type-safety to the configuration, developers now no longer need to learn two different languages (XML and Java) to configure Spring.

The @Configuration Annotation

- Indicates class contains one or more bean definitions
  - Processed by context as source of bean definitions

```
@Configuration
public class JavaConfig { … }
```

  - Explicit name for configuration can be defined
    - Otherwise name will be generated

```
@Configuration("FlightServiceContext")
public class JavaConfig { … }
```

```
@Target(ElementType.TYPE)
@Retention(RetentionPolicy.RUNTIME)
@Documented
@Component
public @interface Configuration {
   String value() default "";
}
```

The @Configuration annotation is used to annotate classes that contain one or more bean definitions that should be registered within a context. When defining a context using this annotation, a name for this context can be explicitly defined using the value attribute of the annotation.

The custom name defined within the annotation applies only when the Configuration class is loaded using component scanning or supplied to a AnnotationConfigApplicationContext.

Bean Configuration (@Bean)

- Annotates bean definition method
  - Implementation creates (and configures) bean instance
  - Object returned by method must be registered in context

```
@Configuration
public class JavaConfig {
   @Bean
   public FlightService flightService() { … }
}
```

- By default bean name will be same as method name
  - Name(s) can be defined explicitly

```
@Bean("JFKFlightRepository")
public FlightRepository flightRepository() {
   return new JFKFlightRepository();
}
```

```
@Bean({"bostonFlightRepository","LoganFlightRepository"})
public FlightRepository flightRepository() {
   return new BostonFlightRepository();
}
```

Instances of beans that need to be registered within a context are created within methods of the configuration class. Each method that is annotated using the Bean annotation returns a preconfigured instance of a managed bean.

By default, the bean that is being returned by the method will be registered in the context under the method-name of the method from which it originated. The Bean annotation does allow us to define one or more bean names for this bean using the value attribute of the annotation.

@Bean Annotation Attributes

- Annotation contains several (optional) attributes
  - value : String array of names
    - (when no other attributes are used)

```
@Bean( {"bostonFlightRepository", "bostonFlightDataSource"} )
```

  - name : String array of names
  - autowireCandidate : Can this bean be autowired into other bean (default: true)

```
@Bean(name= {"bostonFlightRepository"}, autowireCandidate=false)
```

  - initMethod : Name of initialization method on bean
  - destroyMethod : Name of destroy method of bean

```
@Bean(initMethod = "init", destroyMethod="cleanup")
```

The annotation can be used to define the name(s) of the bean, but also the autowiring options and lifecycle methods.

Resolving Dependencies

- Bean dependencies can be resolved by invoking other bean methods
  - Dependent bean must be defined in same class

```
public class FlightServiceImpl implements FlightService {
   private FlightRepository repository;
   public void setRepository(FlightRepository repository) {
      this.repository = repository;
   }
   …
}
```

```
@Bean
public FlightService flightService() {
   FlightServiceImpl service = new FlightServiceImpl();
   service.setRepository(flightRepository());
   return service;
}
@Bean
public FlightRepository flightRepository() {
   BostonFlightRepository repository = new BostonFlightRepository();
   repository.setDatabaseURL("jdbc:derby:flight");
   return repository;
}
```

To resolve the dependencies between two beans, you can simply call another @Bean annotated method within the same class

JavaConfig Dependency Injection

- `@Bean` method may define parameters
  - Defining required dependencies

```
@Configuration
public class JavaConfig {
   @Bean
   public FlightService flightService(FlightRepository flightRepository) {
      FlightServiceImpl service = new FlightServiceImpl();
      service.setRepository(flightRepository);
      return service;
   }
}
```

- Dependent bean can be defined in different class

```
@Configuration
public class RepositoryConfig {
   @Bean @Description("Repository containing Flight information")
   public FlightRepository flightRepository() {
      …
   }
}
```

A Bean might depend on a type that is declared in a different Configuration class (or even in an XML document). When this is the case, you cannot simply invoke the @Bean annotated method to obtain the reference to the other bean. To overcome this problem, your @Bean annotated method may also define parameters. Each parameter is considered a required dependency and will be resolved by the context.

Bean Scopes

- By default beans are created as singletons
  - References are held and re-used by the container
- Scope can be defined using `@Scope` annotation

```
@Bean
@Scope(ConfigurableBeanFactory.SCOPE_PROTOTYPE)
public Booking getNewBooking() {
   …
}
```

- The `request`, `session` and `global` scopes are only available in web-aware contexts
  - Otherwise `IllegalStateException` will be thrown

```
@Scope(WebApplicationContext.SCOPE_SESSION)
```

By default, all beans are created as singletons within a context. When a new instance should be created each time the method is invoked (prototype), the @Bean method can be annotated using the @Scope annotation.

The request, session and global scopes can only be used within a web-aware application context. An IllegalStateException will be thrown when the scope is not supported by the context in which it is being registered.

Java-Based Configuration Internals

- Multiple bean dependencies can reference same bean

```
@Bean public FlightService flightService() {
   FlightServiceImpl service = new FlightServiceImpl();
   service.setRepository(flightRepository());
   return service;
}
@Bean public BookingService bookingService() {
   BookingServiceImpl service = new BookingServiceImpl();
   service.setRepository(flightRepository());
   return service;
}
@Bean public FlightRepository flightRepository() { …}
```

- Method invocation can be used to resolve dependency
  - Wouldn't this result into two instances of the repository?
- Configuration classes are sub-classed at startup
  - Subclass will 'intercept' inner-class method calls
    - Checking container before invoking the method in superclass
- Configuration class may not be declared as `final`

So what happens if two beans have a dependency on the same bean? In the example shown above, two service beans have a dependency on the same repository. Since both bean declarations resolve this dependency by invoking the flightRepository method, wouldn't this result into two instances of the repository instead of the singleton that is 'promised' by the Spring context.

The answer is NO. All configuration classes are subclassed at startup. These subclasses will intercept each call to the @Bean annotated methods. The subclass will check the container to determine whether the bean instance was created before. If so, this instance will be returned, otherwise the bean method of the superclass will be invoked.

Since all configuration classes are being subclasses by Spring, the class you are using to define your beans should not be defined as final!

Bootstrapping JavaConfig Context

- `AnnotationConfigApplicationContext` accepts annotated classes
  - Supports configuration using `@Configuration` classes
- Bean definitions often separated over multiple resources
  - Multiple `@Configuration` classes
  - A combination of XML and Java-based configurations
- Configuration classes can be supplied to constructor
  - Defining one or more `@Configuration` classes

```
context = new AnnotationConfigApplicationContext(JavaConfig.class);
```

- Packages can be scanned for `@Configuration` classes
  - Defining one or more base packages to be scanned

```
context = new AnnotationConfigApplicationContext("com.flight.config")
```

To bootstrap a context that was configured using this Java-Code approach, an AnnotationConfigApplicationContext instance can be used. Constructors of this class accept either one or more configuration classes or the package name of a Java package that should be scanned for configuration classes.

Bootstrapping JavaConfig Context (cont'd)

- Context can also be configured programmatically
  - Allows for resources to be registered after construction of context

```
AnnotationConfigApplicationContext context =
                    new AnnotationConfigApplicationContext();
context.register(JavaConfig.class);
context.register(RepositoryConfig.class);
context.refresh();
```

  - The `refresh` method must be called after registering configurations

Instead of defining the configuration classes during the construction of the AnnotationConfigApplicationContext instance, it is also possible to programmatically configure the context. In order to do so, the AnnotationConfigApplicationContext can be instantiated using its default (no-argument) constructor. The configuration can now be registered with the instance using one of its register methods.

Once the entire configuration of the context has taken place, the refresh method must be called to trigger the initialization of the context.

The `@Import` Annotation

- Allows for bean definitions to be loaded from other configuration classes

```
@Configuration
@Import(RepositoryConfig.class)
public class JavaConfig {
   …
}
```

  - Imported configuration does not need to be explicitly supplied to context

```
Context = new AnnotationConfigApplicationContext(JavaConfig.class);
```

Instead of providing the `AnnotationConfigApplicationContext` with a list of all the Configuration classes that make up a context, you could also decide to provide only one Configuration class to the `AnnotationConfigApplicationContext`.

A Configuration class can use the `@Import` annotation to allow for bean definitions to be loaded from other configuration classes.

Injection in Configuration Classes

- `@Configuration` classes are managed beans within the context
- `@Autowired` annotation can be used to define configuration dependencies
  - Injecting other configuration class

```
@Configuration public class JavaConfig {
   @Autowired private RepositoryConfig repositoryConfig;
   @Bean
   public FlightService flightService() {
      FlightServiceImpl service = new FlightServiceImpl();
      service.setRepository(repositoryConfig.flightRepository());
      return service;
   }
```

  - Injecting single bean (defined in other configuration class)

```
@Configuration public class JavaConfig {
   @Autowired
   private FlightRepository repository;
   …
}
```

```
@Configuration public class RepositoryConfig {
   @Bean public FlightRepository flightRepository() { … }
}
```

A class that has been annotated using the Configuration annotation becomes a managed bean within a context, just like the beans that are defined within. The Configuration annotation actually is a Component type (the annotation contains the @Component meta-data annotation). In other words, when classpath component scanning has been enabled for the package in which the configuration class resides, the configuration class (and the beans defined within) will be discovered and registered automatically.

In the example shown above, the JavaConfig class defines a dependency on the RepositoryConfig class. By having the context resolve this dependency, bean dependencies that need to be resolved within the @Bean annotated methods of the JavaConfig class can be resolved by simply invoking methods on the RepositoryConfig class.

```
@Target(ElementType.TYPE)
@Retention(RetentionPolicy.RUNTIME)
@Documented
@Component
public @interface Configuration {
   String value() default "";
}
```

Conditionally Enabling Beans

- Configurations and beans can be loaded conditionally
  - Depending on system state
- `@Conditional` annotation can be added to definition
  - References one or more `Conditional` implementations
- Resource must match condition(s) before being added to context

```
public class LoggingCondition implements Condition {
   @Override
   public boolean matches(ConditionContext context,
                          AnnotatedTypeMetadata metadata) {
      int cores = Runtime.getRuntime().availableProcessors();
      return cores > 1;
   }
}
```

```
@Bean
@Conditional(LoggingCondition.class)
public static LoggingContext logging() {   … }
```

Sometimes the decision to add a bean or configuration to the context can not be determined by activating one or more profiles. The @Conditional annotation can be used to annotate bean definitions. The annotation must reference an implementation of the Condition interface. The implementation of the matches method defined by this interface determines whether or not the bean should be added to the context.

The @Profile annotation is actually implemented using the @Conditional annotation.

@Conditional as Meta-Annotation

- @Conditional can be used as a meta-annotation
  - Custom 'condition' annotations can be defined

```
@Target({ ElementType.TYPE, ElementType.METHOD })
@Retention(RetentionPolicy.RUNTIME)
@Conditional(LoggingCondition.class)
public @interface Logging {
}
```

```
@Bean
@Logging
public static LoggingContext logging() {
    …
}
```

Just like the @Profile annotation, the @Conditional annotation can also be used to define your own condition annotation. By adding the @Conditional annotation to your custom annotation, the Condition implementation that should be used can now be defined within your custom annotation and does no longer need to be defined when adding the annotation to the bean definition.

Using Profiles

- Different beans can be loaded for different 'environments'
  - Using different `DataSource` during development or testing
  - Conditionally registering monitoring beans
  - ...
- `@Profile` beans allow dynamic inclusion/exclusion of bean declarations
  - Bean will not be loaded unless profile is active

```
@Profile("development")
@Bean public CacheManager cacheManager() {
   return new ConcurrentMapCacheManager("flightCache");
}
@Profile("production")
@Bean public CacheManager cacheManager() {
   return new EhCacheCacheManager("flightCache");
}
```

Some beans (or configurations) should only be loaded when running in a specific environment. A different DataSource (for example) might be used during development and test, or beans should only be registered when the target runtime provides proper support for its functionality. (Monitoring might only be enabled when enough system resources are available)

By using Spring profiles, parts of the configuration can be made available only in certain environment. By annotating beans (or configurations) using the Profile annotation, the beans will only be loaded into the context when the defined profile is active.

Using Profiles (cont'd)

- `@Component` (or `@Configuration`) classes can be annotated using `@Profile`
  - Beans defined in these classes will not be loaded unless profile is active

```
@Configuration
@Profile("production")
public class MonitoringConfig {
    @Bean
    public static MonitoringContext monitoringContext() {
        …
    }
}
```

- `@Profile` is implemented using `@Conditional`

When the Profile annotation is being added to a Configuration class, all beans defined within that class will only be loaded when the defined Profile is active.

Defining Profiles

- Multiple profiles can be defined

```
@Profile({ "production", "monitoring" })
```

  - Resource will be registered when one of them is activated
- Profile name can also be prefixed with NOT (!) operator

```
@Profile("!production")
```

  - Resource will be registered when profile not activated
- `@Profile` can be used as a meta-annotation
  - Custom Profile annotations can be defined

```
@Target(ElementType.TYPE)
@Retention(RetentionPolicy.RUNTIME)
@Profile("monitoring")
public @interface Monitoring {
}
```

```
@Configuration
@Monitoring
public class MonitoringConfig {
    ...
}
```

The Profile annotation allows the definition of multiple profile names. The resource will only be added when at least one of them is activated. The exclamation mark can be used as the NOT operator, which results in the bean being registered only when the defined profile is NOT active.

The Profile annotation can also be used to define custom Profile annotations.

Activating Profiles

- Profiles are deactivated by default
- Profile can be activated
  - Programmatically defining active profiles in `Environment`

```
try (AnnotationConfigApplicationContext context =
                    new AnnotationConfigApplicationContext();) {
   context.register(JavaConfig.class);
   ConfigurableEnvironment environment = context.getEnvironment();
   environment.setActiveProfiles("monitoring");
   context.refresh();
}
```

  - Defining JVM properties

```
java -jar -Dspring.profiles.active=monitoring application.jar
```

  - Defining System properties
    - Before initializing Context!

```
System.setProperty(
    AbstractEnvironment.ACTIVE_PROFILES_PROPERTY_NAME, "monitoring");
```

By default, profiles are deactivated. A profile can be active programmatically during the configuration of the context, but profiles can also be active by using JVM or system properties (both set before the context is initialized).

Activating Profiles (cont'd)

- By defining an `ApplicationListener`
  - Processed while context is initialized

```
public class ApplicationContextListener
      implements ApplicationListener<ContextRefreshedEvent> {
   @Override
   public void onApplicationEvent(ContextRefreshedEvent event) {
      ApplicationContext context = event.getApplicationContext();
      ConfigurableEnvironment env =
              (ConfigurableEnvironment) context.getEnvironment();
      env.setActiveProfiles("monitoring");
   }
}
```

An ApplicationContextListener can also be registered within the context. Beans of this type are being processed during the initialization process of the context, allowing for profiles to be enabled before beans are being loaded into context.

Bean Life-Cycle Methods

- Beans support standard lifecycle methods
  - Defined using `@PostConstruct` and `@PreDestroy`

```
@PreDestroy
public void cleanup() {
    …
}
```

  - Defined using lifecycle callback interfaces
    - `Lifecycle, InitializingBean, DisposableBean`

```
public class RepositoryImpl implements FlightRepository, InitializingBean {
   @Override
   public void afterPropertiesSet() throws Exception {   …   }
   …
}
```

  - Defined using annotation attributes

```
@Bean(initMethod = "init", destroyMethod="cleanup")
```

As we have seen before, lifecycle methods can be defined for each bean that needs to be invoked on the bean instance after the dependencies have been resolved or just before the bean is taken out of context.

In addition to annotating the life-cycle methods within the bean or implementing Spring-specific lifecycle callback interfaces, these methods can also be defined using the initMethod and destroyMethod attributes of the @Bean annotation.

Bean Life-Cycle Methods (cont'd)

- When bean is registered using JavaConfig, a public `close` or `shutdown` method is automatically registered as a destruction callback method

```
@Repository
public class HoustonFlightRepository implements FlightRepository {
   public void close() {
      …
   }
   …
}
```

- When callback method should not be invoked, `destroyMethod` should be set to empty String
  - Should always be done for `DataSource` types

```
@Bean(destroyMethod="")
public DataSource dataSource() {
   …
}
```

When a beans contains a 'close' or 'shutdown' method, these methods are considered destruction callback methods as soon as the bean is registered using the JavaConfig approach.

When using DataSources in a JEE environment, their lifecycle is most often managed outside of the application (e.g. by the application server). As a result the close method of the DataSource should not be called by Spring.

Lifecycle Method Order

- Lifecycle methods will be invoked in predefined order
  - Annotation method will be invoked first
  - Callback interface method will be called second
    - InitializingBean : afterPropertiesSet()
    - DisposableBean : destroy()

```
public class TravelServiceImpl implements TravelService, InitializingBean {
   @Override
   public void afterPropertiesSet() throws Exception {
      logger.info("Callback configured Init");
   }
   @PostConstruct
   public void initAnnotation() {
      logger.info("Annotation configured Init");
   }
```

By adding support for these lifecycle annotations to Spring, it is now possible to define lifecycle methods in several places. Any methods that is being annotated with a lifecycle method will be invoked before any calls are being made to methods defined by any callback interfaces that are implemented by the class.

The lifecycle methods that are being defined by the init-method and destroy-method attributes within the XML are called upon last.

Exercise: Spring Java Config

~/StudentWork/code/spring.javaconfig/lab-code

Please refer to the written lab exercise and follow the directions as provided by your instructor

Defining Bean dependencies

Lesson Agenda

- Introduce Spring annotations for defining dependencies
- Explore the `@Autowired` annotation
- Stereotype annotations
- Qualifying injection points
- Lifecycle annotations
- Using properties in Java based configuration
- The `@Value` annotation
- Using the Candidate Components Index

Component Scanning

- Not all beans need to be explicitly defined
  - Annotations can be used to define components and their dependencies
- Spring provides annotations to stereotype classes
  - Defining application tier to which they belong
- Stereotyped classes can be automatically registered

Stereotype annotations	
@Component	Generic stereotype for Spring-managed components
@Controller	Stereotypes class as Spring MVC Controller
@Repository	Stereotypes class as repository
@Service	Stereotypes class as service

- `@Component` annotation can be used to stereotype any bean-managed component
  - `@Repository`, `@Service` and `@Controller` are specializations

As we have seen before, not all beans need to be explicitly defined in XML or within a Java configuration class. Classpath scanning can still be used to have the container scan packages for classes that were annotated using one of the stereotype annotations.

Since the @Configuration annotation also is a stereotype annotation, component scanning can also be used to have the container discover additional configuration classes.

There is confusion about the four annotations to define components. They all do the same thing. The difference is that if you have a class for a Service, a Repository, or a Controller, the specific class may be able to identify the class as more than just a generic bean. This extra identification may be more important through filtered component-scanners or other customizations. By default there is no difference.

Component Scanning (@ComponentScan)

- Classpath scanning must be enabled to detect stereotyped classes
  - Defining packages to be scanned
  - Registering BeanPostProcessers with context
- @ComponentScan defines scanning directive for @Configuration classes

```
@Configuration
@ComponentScan(basePackages = "com.triveratravel")
public class JavaConfig  {
    ...
}
```

- Also detects other configuration classes
  - @Configuration is a stereotype annotation
- Component scanning automatically enables annotation based configuration

When using a Java-base approach for configuring the context, you still need to enable classpath scanning in order for the context to scan for beans that have been annotated using one of the @Component annotations.

To enable classpath scanning, the ComponentScan annotation should be added to the configuration class to define the base packages that should be scanned. Keep in mind that the @Configuration annotation also is a stereotype annotation. As a result, any Configuration classes defined in the scanned packages will also be registered within the context.

The @ComponentScan Annotation

- Component scanning can be configured
  - Defining the base package for scanning components
    - Defining the package name(s) as String

```
@ComponentScan(basePackages = "com.triveratravel")
```

    - Or using a type-safe alternative (entire package of classes will be scanned)

```
@ComponentScan(basePackageClasses={FlightService.class, FlightRepo.class})
```

  - Defining default initialization strategy

```
@ComponentScan(basePackages = "com", lazyInit = true)
```

  - Defining bean naming generator

```
public class CustomBeanNameGenerator implements BeanNameGenerator {
   public String generateBeanName(BeanDefinition definition,
                                  BeanDefinitionRegistry registry) {
      …
   }
}
```

```
@ComponentScan(basePackages = "com",
               nameGenerator = CustomBeanNameGenerator.class)
```

The ComponentScan annotation must be configured to define the packages that need to be scanned. Not only does the annotation provide an annotation to define the package names as a String, it also contains an attribute that provides a more type-safe approach. By defining one or more classes, the container will scan all classes located in the same package as the one that is being defined.

The annotation also allows the definition of a BeanNameGenerator, which makes it possible (for example) to add a prefix to all bean names of classes which are loaded as a result of this scan

Annotation: `@Component`

- `@Component` annotated beans will be registered with Spring context
  - When classpath component scanning is enabled
- Annotation precedes the class declaration
  - Bean name defaults to class name of registered bean
    - (i.e. `flightServiceImpl`)

```
@Component
public class FlightServiceImpl{
   …
}
```

  - Bean name can be explicitly set:

```
@Component("flightService")
public class FlightServiceImpl{
   …
}
```

The Component annotation can be used to stereotype a bean. By doing so, an instance of this bean will be registered within the Spring context (assuming that classpath component scanning is enabled). By default, the name of the bean will be the unqualified name of the class (class without package name). The bean name can be explicitly defined using the value attribute of the Component annotation.

JSR330 introduced a replacement for the @Component annotation. Instead of using the @Component annotation defined by Spring, you can use the @Named annotations defined by JSR330.

Annotation: @Repository

- Automatically adds exception translation to repository (DAO) implementation
- DAO throws subclasses of RuntimeException
  - DataAccessException, PersistenceException, ...

```
@Repository
public class FlightRepositoryImpl implements FlightRepository {
   @Autowired private JdbcTemplate jdbcTemplate;
   public List<Flight> getFlights(String destinationCode, LocalDate date) {
      …
      return jdbcTemplate.query(statement, arguments, rowMapper);
   }
}
```

- Client can catch sub-types
  - Instead of having to interpret error message

```
try {
   return repository.getFlights(destinationCode, date);
} catch (CannotGetJdbcConnectionException | QueryTimeoutException e) {
   …
}
```

```
try {
   repository.addReservation(destinationCode, date, noOfPassengers);
} catch (DuplicateKeyException e) { … }
```

One specialized component type is defined by the @Repository annotation. Not only does it provide more information about the architectural layer to which the bean belongs, but by adding this annotation, a post processor will be registered within the context that will proxy this bean to add exception translation to the methods. So the implementations of these methods can 'simply' throw subtypes of RuntimeException.

Because of the exception translation that is being added by the @Repository annotation, the client can now catch individual subtypes of (in this example) DataAccessException. Keep in mind that the standard JDBC API would only throw a single type of exception (SQLException). Clients would have to inspect the error message to determine what caused the exception. By adding exception translation, different exceptions are being thrown for different exceptional conditions.

The @Autowired Annotation

- Used to define fine grained auto-wiring information
- May be applied to setter methods of bean

```
@Autowired
public void setRepository(FlightRepository repository) {
   this.repository = repository;
}
```

  - ...or directly to fields of a bean

```
public class FlightServiceImpl implements FlightService {
   @Autowired
   private FlightRepository repository;
   …
}
```

- Can also be added to constructor of bean
  - Only one constructor can be annotated

When a property is annotated with this annotation, the container will attempt to resolve the dependency automatically. When using this annotation on a constructor of a bean, do keep in mind that only a single constructor can be annotated.

The @Autowired Annotation (cont'd)

- Can be added to arrays and (strongly typed) Collections
  - Will be populated with all beans of appropriate type

```
@Autowired
private FlightRepository[] repositories;
```

```
@Autowired
private List<FlightRepository> repositories;
```

- Can be added to `java.util.Map`
  - Key must be of type `java.lang.String`
  - Key will be set to bean name

```
@Autowired private Map<String, FlightRepository> repositories;
```

- Auto-wired properties are treated as required properties
  - Configuration will fail when zero beans are found
- Auto-wiring can be defined as optional

```
@Autowired(required=false)
private Map<String, FlightRepository> repositories;
```

When the annotation is used on a (strongly typed) collection property, the collection will be populated with all available beans of this type.

The @Autowire annotation can even be used on properties of type Map. However, the key of the Map must be of type java.lang.String. When the property is set by the container, the Map will be populated with all available beans of the appropriate type, the key under which they are added to the Map is the name of the bean, as defined in the configuration file.

When annotating a property using the @Autowire annotation, the property is considered required. When zero beans are found (keep in mind a property of type Collection might hold several bean references) an exception will be thrown.

It is possible to define an auto-wire property as optional by setting the required property of the annotation to false. However, this does mean that when no appropriate bean can be found by the container this property will be left to its

default value (which might result in NullPointerExceptions in the application code).

The Well-Known Dependencies

- `@Autowired` can be used to inject "well-known" resolvable dependencies:
  - `BeanFactory`
  - `ApplicationContext`
  - `Environment`
  - `ResourceLoader`
  - `ApplicationEventPublisher`
  - `MessageSource`
- These types (and their sub-types) are automatically resolved
  - No additional configuration is required.

```
@Autowired
private ApplicationContext applicationContext;
```

Several objects that are representing the Spring context can also be injected into a managed bean using the @Autowired annotation. These instances do not have to be explicitly defined within the context since they are already part of this context.

Dependency Injection for Java

- Spring 3 added support for JSR-330
  - Dependency Injection for Java
- Dependencies can now be defined using 'standard' annotations
  - `@Inject` instead of `@Autowired`
  - `@Named` instead of `@Component`
  - `@Singleton` instead of `@Scope("singleton")`
- Requires `javax.inject` library to be added to classpath

```
<dependency>
   <groupId>javax.inject</groupId>
   <artifactId>javax.inject</artifactId>
   <version>1</version>
</dependency>
```

As mentioned before, later versions of Spring started to provide support for more annotations defined by the Java platform. Several annotations that were defined by JSR-330 can be used instead of the Spring annotations to accomplish the same task. Keep in mind that in order to use these annotations the javax.inject library must be added to the classpath in addition to the Spring libraries.

Annotations: `@Autowired` and `@Primary`

- Auto-wire capabilities often result in multiple candidates for injection
  - Multiple beans of the same type will be found
- A bean can be annotated as `@Primary`
  - Makes it the preferred bean when multiple injection candidates are available

```
@Repository
@Primary
public class FlightRepositoryImpl implements FlightRepository {
    ...
}
```

When using auto-wiring, the container might end up with multiple candidates for injection. When left to the container, you cannot be sure which one of the available candidates is going to be injected.

When multiple beans of the same type are present within a context, one of these beans can be annotated using the @Primary annotation. The bean will then become the preferred bean for injection. When the container finds multiple instances of a given type and only one of them has been annotated using @Primary, this will be the instance that will be injected.

Annotation: `@Qualifier`

- `@Qualifier` provides more control of injection candidates
  - Can be applied to fields, methods and parameters

```
@Autowired
public void init(@Qualifier("boston") FlightRepository repository) {
   …
}
```

```
@Autowired
@Qualifier("boston")
private FlightRepository repository;
```

- Candidate classes should be qualified

```
@Repository
@Qualifier("boston")
public class BostonFlightRepository implements FlightRepository {   …   }
```

- Annotation can also be applied to Collections
  - All beans that match type and Qualifier will be injected

```
@Autowired
@Qualifier("domestic")
private Set<FlightService> flightServices;
```

To provide developers with more control over the injection candidates, Spring has defined the Qualifier annotation. This annotation can be applied to fields, methods and parameters and informs the context that only a bean that also contains this qualifier may be injected.

The Qualifier annotation that should be used is org.springframework.beans.factory.annotation.Qualifier (not javax.inject.Qualifier!)

When properties of type Collections are being annotated using the Qualifier annotation, all beans that match the generic type of the collection and match the qualifier will be injected.

The Environment API

- The Environment API abstracts context environment
  - Defines Profiles and Properties

```
public interface Environment extends PropertyResolver {
   String[] getActiveProfiles();
   String[] getDefaultProfiles();
   boolean acceptsProfiles(String... profiles);
}
```

- Profiles define logical groups of bean definitions
  - Conditionally added to the Spring context
- Properties can be used for application configuration

The Environment API is an integrated part of the context environment. If defines both properties and profiles that can be used to configure the context during startup.

Property Sources

- `PropertySource` abstracts sources containing key-value pairs
  - Can be defined in a variety of resources
    - Property files
    - JVM properties
    - Environment variables
    - JNDI
    - …
- `Environment` is preconfigured with two property sources
  - JVM system properties

```
Environment environment = …
String javaVersion = environment.getProperty("java.version");
```

  - System environment variables

```
String javaHome = environment.getProperty("JAVA_HOME");
```

A PropertySource is an abstraction of a source that contains key-value pairs. The actual source can be defined in a variety of resources. By default only two property sources are registered within the context: The JVM and System properties

@PropertySource and @Value

- @PropertySource adds additional property sources to environment

```
@Configuration
@PropertySource("classpath:application.properties")
public class JavaConfig {

}
```

```
location.code=LAX
```

- @Value injects property value into fields of managed beans
  - Can be applied at field or parameter level

```
@Component
public class TravelServiceImpl implements TravelService {
   @Value("${location.code}")
   private String aiportLocationCode;
   …
}
```

When property sources (other than JVM and system properties) need to be registered with the context, the @PropertySource can be used. The @Value annotation was introduced to be able to inject 'simple' values into properties of a bean. The annotation can be added to a field or a parameter and even though the value of this annotation could hold a simple value, it is most commonly used to inject a value that is defined within a properties file or a system property.

Candidate Components Index

- Alternative to runtime classpath scanning
- Uses metadata file : `META-INF/spring.components`
  - Generated at compile time
- Each file entry represents a candidate component
  - Fully qualified name and stereotypes

```
com.travel.service.FlightServiceImpl=org.springframework.stereotype.Component
com.travel.service.TravelServiceImpl=org.springframework.stereotype.Component
com.travel.repository.ReservationRepository=org.springframework.stereotype.Component
com.travel.Application=org.springframework.stereotype.Component
```

- Component index is created at build-time
- Intended to decrease startup time of large applications
  - (e.g. containing 200 classes or more)
  - Slow or expensive IO operations
  - Beans are not created by factory methods

Even though classpath scanning allows for the definition of components and their dependencies within the class itself, it does introduce some overhead during startup of the application. This is especially true in applications with several hundred managed beans where the startup time can exceed 60 seconds. Even though this might not sound like much, during development developers might deploy the application 30 times a day, this results in developers sitting idle for more than half an hour each day.

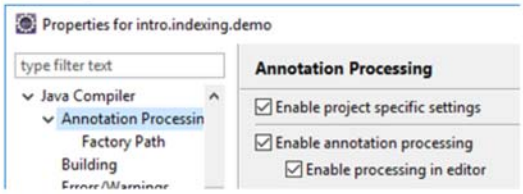
Candidate Components Index is a file containing a list of all candidate components that should be managed by the Spring ApplicationContext.

Generating the Index File

- Index file is created at build time
  - Using annotation processor tool: Spring Indexer
- Indexer identifies all `@Indexed` annotated components
- Context indexer must be added to build path

```
<dependency>
   <groupId>org.springframework</groupId>
   <artifactId>spring-context-indexer</artifactId>
   <optional>true</optional>
</dependency>
```

- Annotation Processor must be enabled during build
  - Part of Maven or Gradle build process
  - Enabled within IDE



The index file is created during the build of the project by registering an annotation processor with the build process. The Spring Indexer identifies all @Indexed annotated components and adds them to the spring.components file located in the META-INF folder.

To enable annotation processing, the indexer library must be added to the Classpath. Within an IDE (like Eclipse) annotation processing must be explicitly enabled to make sure the spring.components file is updated each time beans are being adding or modified.

Indexed Classes

- Candidate components include
  - `@Component` annotated classes
    - `@Component` contains `@Index` meta annotation

```
@Target(TYPE) @Retention(RUNTIME)
@Documented @Indexed
public @interface Component {
  String value() default "";
}
```

  - All Stereotype annotated classes
    - `@Repository`, `@Controller`, `@Service`, `@Configuration`
  - Classes and interfaces annotated using `javax` packages
    - CDI annotations : `@Named`, `@ManagedBean`
    - JPA annotations: `@Entity`, `@EntityListeners`
    - Servlet annotations : `@WebListener`
  - Classes and interfaces annotated with `@Indexed`

Once the indexer has been added (and registered with the build process) all @Indexed annotated classes will be added to the file. Since the @Component annotation contains the @Index meta annotation, all stereotype annotated classes will be added to the file. In addition, all classes annotated with annotations for the Java enterprise packages will also be added.

Adding @Indexed

- Adding @Indexed does not make bean managed
  - 'Only' defines stereotype to be considered in index
- All subclasses of @Indexed type will be indexed

```
@Indexed
public interface TravelService {
    …
}
```

```
@Service
public class BostonTravelService implements TravelService {
    …
}
```

```
com.travel.service.TravelService=com.trivera.travel.service.TravelService
com.travel.service.BostonTravelService=org.springframework.stereotype.Component,
                                       com.travel.service.TravelService
```

- Stereotype annotation has to added to managed types
  - Making bean eligible for injection

Just adding @Indexed to a bean definition does not make the component a managed bean (and does not make it eligible for injection). When the annotation is added to an interface of a superclass, all sub-types will automatically be added to the Candidate Components Index. To make sure the beans are also managed by Spring and become eligible for injection, the bean must also be annotated using one of the Stereotype annotations.

Filtering the Component Scan

- By default all `@Component` classes are registered
  - Filters can be specified to limit candidate components
    - Explicitly including or excluding classes

```
@ComponentScan(basePackages = "com.flight",
   includeFilters = @Filter(type = FilterType.ASSIGNABLE_TYPE,
                            classes = FlightRepository.class),
   excludeFilters = @Filter(type = FilterType.ANNOTATION,
                            classes = Repository.class))
```

- `@Filter` can be defined using
  - Annotations defined on bean
  - The type of the bean
  - Using a custom `TypeFilter`
  - Using and AspectJ or Regex expression

By default, all Component classes that are being discovered during the classpath scan are registered within the context. The annotation also allows for the definition of filters that can be used to exclude (or include) certain beans.

Filtering the Component Scan (cont'd)

- Component scan filter can be used to explicitly include `@Indexed` types
  - Explicitly referencing type

```
@Indexed
public class NYCTravelService implements TravelService { … }
```

```
@SpringBootApplication
@ComponentScan(basePackages = {"com.trivera.travel"}, includeFilters =
   @ComponentScan.Filter(type = ASSIGNABLE_TYPE, value = NYCTravelService.class))
public class Application {
```

  - Explicitly referencing annotation

```
@Target(TYPE) @Retention(RUNTIME)
@Indexed
public @interface IndexedService { … }
```

```
@ComponentScan(basePackages = {"com.trivera.travel" }, includeFilters =
    @ComponentScan.Filter(type = ANNOTATION, value = IndexedService.class))
```

To make sure custom @Indexed types are being added to context the @ComponentScan annotation can be used to explicitly load beans by their type, or through the use of a custom annotation which in turn has been annotated using the @Indexed meta annotation

Disabling Candidate Component Index

- Candidate Component Index is used when `spring.components` file is found on classpath
  - Disables classpath component scanning
- Use of Candidate Component Index can be disabled
  - By defining `spring.index.ignore` system property
  - Setting `spring.index.ignore` in `spring.properties` file
    - Defined at root of application classpath

One drawback of using the Candidate Component Index is that the index is automatically enabled when a META-INF/spring.components file is found on the classpath during start-up of the application. As soon as this file is found, the regular Classpath component scanning is disabled; the two of them cannot be used at the same time. When beans are needed that are not defined within the spring.components file, the candidate component scanning has to be disabled.

Spring provides feature flag spring.index.ignore which can be set using a system property or can be defined in the spring.properties file (at the root of the Classpath) to disable the use of the candidate component index.

Exercise 2: Configuring Bean Dependencies using Annotations

~/StudentWork/code/spring.bean.dependencies/lab-code

Please refer to the written lab exercise and follow the directions as provided by your instructor

Exercise 3: Creating the Candidate Component Index

~/StudentWork/code/spring.bean.dependencies/lab-code

Please refer to the written lab exercise and follow the directions as provided by your instructor

Introduction to Spring Boot

The Spring Framework
Configuring Spring Managed Beans
Defining Bean dependencies
Introduction to Spring Boot
Working with Spring Boot

Lesson Agenda

- Introduce the basics of Spring Boot
- Explain auto-configuration
- Introduce the Spring Initializr application
- Bootstrapping a Spring Boot application

Spring Libraries

- Spring provides large set of libraries
  - Each Spring project requires unique set of Spring libraries
- Projects often need a set of dependent jars
  - Coming from a variety of sources
    - Apache, database drivers, logging implementations, etc.
- Resolving dependencies within a Spring project can be difficult
  - Build tools like Maven can help resolve the dependencies

```
<dependencyManagement>
  <dependencies>
    <dependency>
      <groupId>org.springframework</groupId>
      <artifactId>spring-framework-bom</artifactId>
      <version>5.2.0.RELEASE</version>
      <type>pom</type>
      <scope>import</scope>
    </dependency>
  </dependencies>
</dependencyManagement>
```

While early versions of the Spring framework were provided as a single jar, over the years the Spring framework has grown into a large development 'platform' for the development of large enterprise applications. (Spring 2.x provided single jar which contained the majority of the Spring framework.)

To support this growth, Spring now consists of several libraries, each library focusing on a particular 'topic' (data access, web development, AOP, etc.). In addition to these Spring classes, additional libraries also need to be downloaded from a variety of locations.

To overcome a lot problems when setting up a project, build environments like Maven are often used. Maven supports the concept of a "bill of materials" (BOM) dependency. When the spring-framework-bom is imported into the dependencyManagement section of the POM file, it ensures that all spring dependencies (both direct and transitive) are of the same version. In addition, when using this BOM the <version> attribute no longer needs to be defined when adding Spring Framework artifacts to the project

Life Without Spring Boot

- Spring did simplify creation of enterprise applications
  - Classes can be tested outside of JEE container
  - Classes are framework agnostic
    - They do not need to implement framework specific interfaces
- A lot of boilerplate code is required!
  - Focus should be on implementation of business logic
- Library dependencies need to be managed manually
  - Developers need to define dependencies
  - Version conflicts need to be resolved

Even though Spring did simplify the creation of enterprise applications, the previous slides give you an idea of the amount of boilerplate code that is required to setup the project. All the steps shown had to do with the configuration of the environment and none had to do with the implementation of the actual business logic.

Spring Boot Overview

- Provide easier mechanisms for Spring development
  - Managing dependencies
  - Minimizing Spring configuration
  - Work with Maven and Gradle
- Provide non-functional capabilities for monitoring and measuring projects
  - Security
  - Metrics
  - Health checks
  - Embedded servers and externalized configuration
- Spring Boot impacts dependencies and configuration
  - Has no impact on programming model

One of the goals of Spring Boot is to allow a developer to quickly setup a new Spring project without having to worry about selecting the correct libraries, setting up data-sources and defining templates. In other words, Spring Boot allows developers to once again focus on the implementation of the business logic and not waste time writing all this boilerplate code and configuration.

In addition to providing a quick and easy way to setup a new project, Spring Boot also provides a number of ‘tools’ that allow us to monitor and measure applications at runtime.

Most importantly, Spring Boot impacts the way we define project dependencies and project configuration. It does not require our application classes to implement specific interfaces. Spring Boot has no impact on the programming model.

Auto-Configuration

- Spring 4 introduced conditional configurations
  - Using the `Condition` interface and `@Conditional` annotation
- Spring Boot attempts auto-configuration of Spring beans
  - Based on libraries found on classpath
    - e.g. When `HibernateTemplate` is found it will be registered
  - Based on absence of bean(s)
    - When bean is already present, Spring Boot will not configure another
- Auto-configuration is non-invasive
  - All configuration can be made explicit
    - Replacing automatically configured beans

Spring 4 introduced the Condition interface and the @Conditional annotation, allowing beans (and entire configurations) to be loaded only when a certain condition is met. The condition might be a Profile that has been enabled or a System property that has been defined, but more advanced conditions can also be specified. Beans can be registered when a specific class is present on the classpath, a bean of a certain type hasn't already been registered in context or when a file exists on a given location.

Spring Boot attempts to register beans it finds on the classpath, using smart defaults. For example, when the Spring-JPA library is found on the classpath, Spring Boot attempts to register the HibernateTemplate (and all beans required to support this template).

Spring Boot relies of smart defaults. It is not dictating a setup, it configures some default settings but it still allows developers to override all of these settings.

Starting with Spring Boot

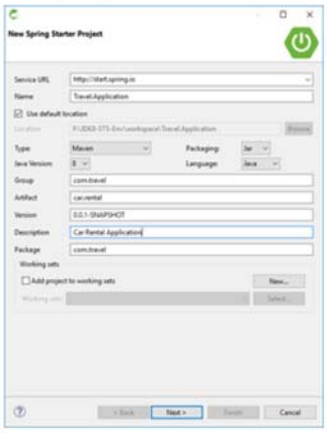
- Spring Boot eliminates the pain of startup dependencies
  - Automatically registering Spring beans to setup framework
- Tooling creates a working project
  - Using smart defaults for beans found on classpath
  - Can be completely tailored for 'final' application
    - All defaults can be overridden
- Spring Boot lets us focus on application functionality
  - Embraces *"convention over configuration"* approach

Spring Boot provides us with tooling to set up a new project. When developing an application (Spring based component) from scratch, the tooling can be used to create a new project which not only has all the required dependencies in place, but also has been configured to work out of the box. When we want to develop a new Spring-based web application, the tooling will generate a Maven (or Gradle) project containing all the library dependencies, some template classes and, most importantly, has been completely configured. The generated application is ready to be deployed as a web application. In the case of a web application, the generated application even contains an embedded web server (Tomcat), allowing us to run the web application from the command-line, without the need of deploying the application onto a web server before testing it.

Spring Boot completely focuses on 'Convention over Configuration'. Everything that can be configured by default will be configured. Only when those defaults do not apply to the application do we need to provide an alternative setup.

Spring Boot Initializr

- Spring provides tooling for creating a starter project
  - Spring Boot Initializr Web UI
  - Spring Boot Command-line-interface (CLI)
  - IDE integrated tooling
    - IntelliJ has built in Initializr support
    - Spring Tool Suite™ provides Initializr plugin
      - Requires Internet connection to connect to Initializr Web UI

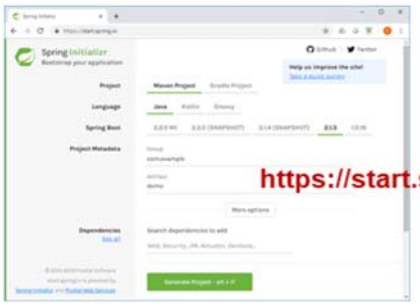


Spring Boot provides tooling that can be used to configure and generate a new project. Besides a web-based application (covered next), it provides a command-line interface. IDE's like IntelliJ or the (Eclipse based) Spring Tool Suite, provide wizards to setup a new Spring Boot project.

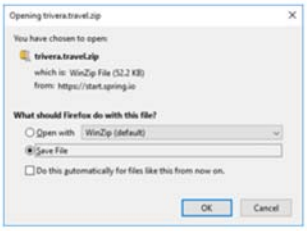
It should be noted that at the time of writing, STS requires an internet connection in order to be able to setup a new Spring Boot project. The wizard within STS uses the REST API of the Initializr application to create the new project.

Spring Initializr Web UI

- Spring Initializr Web UI configures starter project
  - Language version
  - Preferred build system (Maven or Gradle)
  - Packaging
    - Jar / War
  - Project dependencies



- Select functionality needed by application
  - Not the required dependencies
- A (fully working) project will be created
  - Downloaded as a single zip



The Spring Initializr application is a ‘simple’ web application which allows us to define the basic configuration of the new application. Not only does it allow us to specify the Spring Boot version that should be used, the build environment (Maven or Gradle) can also be selected as well as the packaging type of the new application.

Once the basic configuration of the new project has completed, we can select the ‘dependencies’ that we would like to be part of our new application. The dependency is not a dependency on a single library as we select the dependency on a specific ‘technology’. You can select the JPA dependency, or the Derby dependency. By doing so Spring Boot, will make sure that all the libraries that are needed to write a JPA based application which stores its state in a Derby database.

Once all the configuration has been done, a complete project can be downloaded.

The Starter Project

- Complete project setup has been created
  - Bootstrap class
  - Test class
  - Properties file
- Other projects might contain more files and folders
  - Web application initializers
  - Page templates

- MVNW commands are Maven wrappers
  - Project can be executed without having Maven installed
    - Correct Maven version will be downloaded when not found

The zip file must be extracted on the local filesystem. Once extracted, you will find a complete Maven (or Gradle) project. In the example shown above, a Maven project has been created. It contains a complete pom.xml file describing the project and its dependencies. The project also contains a Bootstrap class (Application.java), a JUnit test class (ApplicationTest.java), and a properties file (application.properties). This (optional) properties files can be used later on to overwrite some of the defaults of the application.

Keep in mind that in this lesson we are developing a ‘simple’ JPA application without a UI. If we would have selected the Web dependency in the Spring Initializr, the template project would have also contained classes to setup the web application.

The zip file even contains two ‘mvmw’ scripts. These scripts can be used to execute the maven command even when Maven has not been installed on the system. When executing these scripts the correct Maven version will be downloaded automatically if needed.

The POM File

- Each selected dependency results in a starter dependency
  - Downloads all necessary dependencies

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

- Library versions are determined by the parent pom
  - Downloaded from Spring repository
  - All dependency versions are tied to Spring Boot version
    - Selected in Initializr

```
<parent>
   <groupId>org.springframework.boot</groupId>
   <artifactId>spring-boot-starter-parent</artifactId>
   <version>2.1.9.RELEASE</version>
</parent>
```

The generated POM file contains a spring-boot-starter-… dependency for each of the dependencies that was selected in the Spring Initializr. The library versions that are added to the classpath are tied to the Spring Boot version that was selected.

The POM File (cont'd)

- Database libraries are also added to classpath
  - Results in automatic `DataSource` configuration

```
<dependency>
   <groupId>org.apache.derby</groupId>
   <artifactId>derby</artifactId>
   <scope>runtime</scope>
</dependency>
```

- Each Spring Boot application is pre-configured for testing

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

```
@RunWith(SpringRunner.class)
@SpringBootTest
public class ApplicationTests {
   @Test
   public void contextLoads() {}
}
```

By default, each Spring Boot project comes pre-configured with the spring-boot-starter-test and a 'skeleton' test class. This test class does not only serve as an example of how to write a unit test within a Spring Boot application, it already contains a single test method. Even though the method is empty, as the name already suggests it tests that the ApplicationContext can be loaded successfully.

The Bootstrap Class

- Generated `Application` class serves two purposes
  - Primary configuration class
  - Bootstraps the application

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

- `@SpringBootApplication` enables component scanning and auto-configuration
  - Combination of three annotations
    - `@Configuration`, `@ComponentScan` and `@EnableAutoConfig`
- Should be placed in a root package of the application
  - Allowing all sub-packages to be scanned

The application that was generated by the Initializr application also contains an Application class, which serves two purposes. It acts as the primary configuration class of the Spring Boot application and is the bootstrap class, allowing the application to be started from the command-line.

The Application class has been annotated using the @SpringBootApplication, which serves the same purpose as adding @Configuration, @EnableAutoConfiguration and @ComponentScan

@Component and @Autowired

- @Component annotated beans will be added to Spring context

```
@Component
public class FlightService{
   …
}
```

- @Autowired defines dependency between beans
  - To be resolved by Spring context

```
@Component
public class FlightService {
   @Autowired
   private FlightRepository repository;
   …
}
```

When adding the @Component annotation to a class and adding this class to the Classpath of the Spring Boot application, the bean will become a managed bean within the Spring context.

By annotating properties of the bean with @Autowired, the dependencies between beans can be defined. At runtime the dependencies will be resolved by Spring before business methods are invoked on the instance.

CommandLineRunner

- Implementations run just before application startup completes
  - Provides access to applications command-line arguments

```
@FunctionalInterface
public interface CommandLineRunner {
   void run(String... args) throws Exception;
}
```

- When multiple implementations are available, order of execution can be defined
  - Using Spring's `@Order` annotation
  - By implementing Spring's `Ordered` interface

```
@Order(Ordered.HIGHEST_PRECEDENCE)
public class InitializeApplicationData implements CommandLineRunner {
   public void run(String... args) throws Exception { … }
}
```

Implementations of CommandLineRunner can be added to Spring context. Each implementation will be executed by the runtime just before the application startup completes.

When multiple instances of this interface have been registered, the order in which they are executed can be defined by either annotating the implementation with the @Order annotation or by having the implementation implement the Ordered interface

Exercise 4: Introduction to Spring using Spring Boot

~/StudentWork/code/spring.introduction/lab-code

Please refer to the written lab exercise and follow the directions as provided by your instructor

Working with Spring Boot

The Spring Framework
Configuring Spring Managed Beans
Defining Bean dependencies
Introduction to Spring Boot
Working with Spring Boot

Lesson Agenda

- Provide an overview of Spring Boot
- Introduce starter dependencies
- Introduce auto-configuration
- `@Enable...` annotations
- Conditional configuration
- Spring Boot Externalized Configuration
- Bootstrapping Spring Boot

Spring Boot Overview

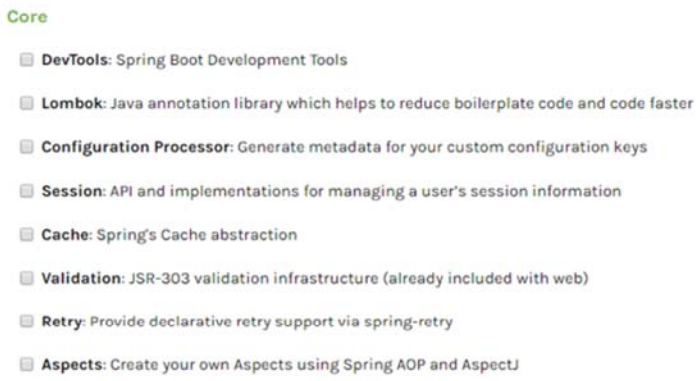
- Provides easy and approachable mechanism for Spring development
  - Managing library dependencies
  - Minimizing Spring configuration
- Works closely with Maven and Gradle
  - Plugins are available for both build environments
- Provides many non-functional capabilities for monitoring and measuring projects
  - Security,
  - Metrics
  - Health checks
  - Embedded servers and externalized configuration

Overview of Starter Dependency Categories

- Core
- Web
- Template Engines
- Security
- SQL
- NoSQL
- Integration
- Cloud Core
- Cloud Support
- Cloud Config
- Cloud Discovery
- Cloud Routing
- Cloud Circuit Breaker
- Cloud Tracing
- Cloud Messaging
- Cloud AWS
- Cloud Contract
- Pivotal Cloud Foundry
- Azure
- Spring Cloud GCP
- Google Cloud Platform
- I/O
- Ops

Starter Dependencies

- Each group contains several starter dependencies
  - Defining the kind of functionality to be added



- Spring Boot will add a starter dependency to build file
  - Referencing libraries required for selected functionality
    - ...and their transitive dependencies

Starter Dependencies (cont'd)

- Aggregates sets of dependencies into a single dependency
  - <spring-boot-starter-jdbc> includes:
    - spring-jdbc, tomcat-jdbc, spring-tx, …
  - <spring-boot-starter-data-rest> includes:
    - spring-bootstarter-web, Jackson annotations, spring-core, spring-tx, …
- Build handles transitive dependency resolution
  - Dependencies may have dependencies of their own
- Starters reduce the size of dependency list
  - Add consistency by using these prepackaged starters

Spring Boot Starters

- Adding a starter to Gradle:

```
dependencies {
  compile('org.springframework.boot:spring-boot-starter-data-jpa')
}
```

- Adding a starter to Maven:

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

- All Maven starters inherit from a master boot parent:

```
<parent>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-parent</artifactId>
    <version>2.1.3.RELEASE</version>
    <relativePath/> <!-- lookup parent from repository -->
</parent>
```

Spring Boot, Maven and Gradle

- Spring Boot defines plugins for both Maven and Gradle
  - Used for building (and running) the project
    - Gradle:

```
apply plugin: 'org.springframework.boot'
```

    - Maven:

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

- Maven plugin defines several goals
  - repackage: create a jar or war file that is auto-executable
  - run: run the Spring Boot application
  - start / stop: Integrate Spring Boot application into the integration-test phase
  - build-info: generate build information (for use by Actuator)

Spring Boot Maven Usage

The plugin provides several goals to work with a Spring Boot application:

repackage: create a jar or war file that is auto-executable. It can replace the regular artifact or can be attached to the build lifecycle with a separate classifier.

run: run your Spring Boot application with several options to pass parameters to it.

start and **stop:** integrate your Spring Boot application to the integration-test phase so that the application starts before it.

build-info: generate a build information that can be used by the Actuator.

Auto-Configuration Overview

- Spring Boot embraces
  "convention over configuration"
- Enable auto-configuration of Spring Application Context
  - Attempts to 'guess' and configure beans that are most likely needed
- Auto-configuration is usually applied based on classpath and beans that were explicitly defined
- Modules are only auto-configured when included as a boot-starter in project

The `@Enable...` Annotations

- `@Enable...` annotations were introduced in Spring 3
  - To replace XML files
- This is a common approach to enable features in Spring applications
  - `@EnableTransactionManagement`: enables declarative transaction management
  - `@EnableWebMvc`: enables Spring MVC
  - `@EnableScheduling`: Initialize a scheduler
  - ...

Auto-Configuration

- Auto-Configuration must be explicitly enabled
  - By annotating one of the `@Configuration` classes
    - `@EnableAutoConfiguration`
    - `@SpringBootApplication`

```
@EnableAutoConfiguration
@Configuration
public class JavaConfig {
   …
}
```

- `@SpringBootApplication` enables component scanning and auto-configuration
  - Combination of three annotations
    - `@Configuration`
    - `@Componentscan`
    - `@EnableAutoConfig`
- Should be placed in a root package of the application
  - Allowing all sub-packages to be scanned

The auto-configuration of the components added to the classpath must be enabled explicitly by annotating one of the @Configuration classes with the @EnableAutoConfiguration annotation. It is recommended that the annotation is only added once within the configuration and preferably located in the root package of the application to allow for all sub-packages defined by the application to be scanned.

Auto-Configuration (cont'd)

- `@EnableAutoConfiguration` imports `AutoConfigurationImportSelector`
  - Loads available `*AutoConfiguration` based factories from `SpringFactoriesLoader#loadFactoryNames`
- Factory implementations are loaded from spring-boot-autoconfigure jar
  - Defined in `META-INF/spring.factories`
- Many factories can be loaded, including:
  - `EnableAutoConfiguration`
  - `ConfigurationPropertiesAutoConfiguration`
  - `MessageSourceAutoConfiguration`
  - `PropertyPlaceholderAutoConfiguration`
  - `JpaRepositoriesAutoConfiguration`
  - ...

The **EnableAutoConfigurationImportSelector** uses **SpringFactoriesLoader#loadFactoryNames** of Spring core. **SpringFactoriesLoader** will look for jars containing a file with the path **META-INF/spring.factories**.

When it finds such a file, the **SpringFactoriesLoader** will look for the property named after our configuration file. In our case, **org.springframework.boot.autoconfigure.EnableAutoConfiguration**.

Overriding Auto-Configuration

- Default configuration can easily be overridden
  - Can be done by explicit configuration of beans
- Spring 4 introduced conditional configurations
  - Through `Condition` interface and `@Conditional` annotation
- Auto-configuration makes use of conditional configuration

```
@Configuration
@ConditionalOnBean(DataSource.class)
@ConditionalOnClass(JpaRepository.class)
@ConditionalOnMissingBean({JpaRepositoryFactoryBean.class,
                    JpaRepositoryConfigExtension.class })
ConditionalOnProperty(prefix = "spring.data.jpa.repositories",
     name ="enabled", havingValue ="true", matchIfMissing =true)
@Import(JpaRepositoriesAutoConfigureRegistrar.class)
@AutoConfigureAfter(HibernateJpaAutoConfiguration.class)
public class JpaRepositoriesAutoConfiguration {
}
```

Conditional Annotations

Commonly used Conditional annotations	
`@ConditionalOnBean`	Configure when specified bean is defined
`@ConditionalOnMissingBean`	Configure when specified bean is NOT defined
`@ConditionalOnClass`	Configure when classes are available on classpath
`@ConditionalOnMissingClass`	Configure when classes are NOT available on classpath
`@ConditionalOnProperty`	Configure when property is present
`@ConditionalOnResource`	Configure when resource is present
`@ConditionalOnWebApplication`	Configure when application is web application
`@ConditionalOnNotWebApplication`	Configure when application is not a web application
`@ConditionalOnExpression`	Configure based on outcome of SpEL expression
...	

An application is considered a web application when a WebApplicationContext is configured, a session scope is defined, or a StandardServletEnvironment is used

Disable Auto-Configuration

- Auto-Configuration can be disabled
  - By excluding *AutoConfiguration class(es)
    - Using Annotation

```
@SpringBootApplication(exclude = { DataSourceAutoConfiguration.class,
                                   JpaRepositoriesAutoConfiguration.class })
```

    - Using YAML

```
spring:
  autoconfigure:
    exclude:
      - o.s.b.aa.data.jpa.JpaRepositoriesAutoConfiguration
      - o.s.b.aa.jdbc.DataSourceAutoConfiguration
```

    - Using Properties file

```
spring.autoconfigure.exclude[0]=o.s.b.a.data.jpa.JpaRepositoriesAutoConfiguration
spring.autoconfigure.exclude[1]=o.s.b.a.jdbc.DataSourceAutoConfiguration
```

Spring Boot Externalized Configuration

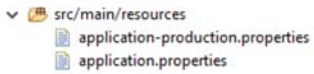
- Spring Boot defined over 300 properties for fine-tuning auto-configuration
  - Allow for modification of automatically configured beans
    - Database URL
    - Server port
    - ...
- Properties can be defined in different ways
  - Properties files
  - YAML (YAML Ain't Markup Language) files
  - Environment variables
  - Command-line arguments
  - ...

Spring Boot Property Sources

- Spring Boot can read properties from a variety of resources
  1. Command-line arguments
  2. JNDI attributes (java:comp/env)
  3. JVM system properties
  4. Environment variables (from operating system)
  5. Randomly generated values for properties
  6. External Properties or YAML files
  7. Properties or YAML files packaged with application
  8. Properties defined by `@PropertySource` annotation
  9. Default properties
- Properties are read in order defined above
  - e.g. command-line argument will override JNDI attribute

Profile-Specific Properties

- Explicit configuration might be different per profile
  - Would require multiple properties to be defined per profile
- Spring Boot supports profile-specific property files
  - Named `application-{profile}.properties`



- Profile-specific files override non-specific property files

Building a REST Repository

- Use the Initializr to define the project
  - Selecting required project facets
    - Spring Data JPA
    - Rest Repositories
    - A Database (e.g. H2)
- Generate, download and inspect the project

```
<dependencies>
   <dependency>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-starter-data-jpa</artifactId>
   </dependency>
   <dependency>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-starter-data-rest</artifactId>
   </dependency>
   <dependency>
      <groupId>com.h2database</groupId>
      <artifactId>h2</artifactId>
      <scope>runtime</scope>
   </dependency>
</dependencies>
```

```
<parent>
   <groupId>org.springframework.boot</groupId>
   <artifactId>spring-boot-starter-parent</artifactId>
   <version>2.1.9.RELEASE</version>
   <relativePath/>
</parent>
```

Introduction to Spring Data

- Provides 'simple' programming model for data access
  - Independent of data access technology used
- Data access layer often contains a lot of boilerplate code
  - Obtaining connections
  - Defining and executing queries
  - Iterating over query results and populating entities
- Spring Data makes it possible to completely remove DAO implementations
  - Only DAO interface needs to be defined
- DAO interface for JPA extends `JpaRepository`

```
public interface ReservationRepository
                 extends JpaRepository<Reservation, Integer> {
}
```

```
reservationRepository.save(entity);
List<Reservation> reservations = reservationRepository.findAll();
Reservation reservation = reservationRepository.findOne(id);
```

Entity type

Primary Key Type

Another time-consuming task during the development of an application is the implementation of the data access layer. Even though properly storing data in a persistent data store is a very important part of any application, you could argue that it is not really part of the business process that needs to be implemented.

Writing a data access layer consists of writing a lot of boilerplate code, which is pretty much the same every time an entity state needs to be read or stored. A connection to the datasource needs to obtained and maintained (properly closed), queries need to be defined and populated with parameter values and result sets need to be iterated over to create new entity instances for each row in the result.

Spring Data is another step in the evolution of the Spring API's. Earlier versions of Spring assisted in the implementation of DAO layers by providing template and utility classes. Spring Data makes it possibly to implement you entire DAO layer by just defining the interface of the DAO object that is needed.

To define a DAO interface for a JPA implementation using Spring Data, the interface should extend the JpaRepository interface. The generic JpaRepository interface defines some of the basic CRUD methods, while its generic types define the Entity type and the primary key type of the entity.

Spring Data Automatic Custom Queries

- Automatic queries will be generated from method names defined in interface

```
public interface ReservationRepository
                 extends JpaRepository<Reservation, Integer> {
   List<Reservation> findReservationsByRentalLocationAndRentalDate(
                             String rentalLocation, LocalDate rentalDate);
}
```

- Method name is stripped of predefined prefixes
  - findBy, find, readBy, read, getBy and get
- Remainder of the method name is parsed
  - Query conditions can be defined using properties and keywords

```
findByNameOnReservationAndArrivalDate(…)
findByArrivalDateBetween(…)
findByNameOnReservationLike(…)
findByArrivalDateOrderByNameOnReservationDesc(…)
```

Besides the methods that are 'inherited' by the JpaRepository, the DAO interface may also define other query methods. When using Spring Data, you will not have to write an implementation of this interface. An implementation of the interface will be generated.

The properties and conditions that need to be part of the selection criteria (e.g. the WHERE clause of a SQL statement) should make up the method name. Of all methods defined on the interface, the prefix 'findBy', 'find', 'readBy', 'read', 'getBy' or 'get' will be stripped. The remainder of the method name is parsed.

The Domain Object and Repository

- Domain objects are annotated using JPA annotations

```
import javax.persistence.*;
@Entity
public class User {
   @Id
   @GeneratedValue(strategy = GenerationType.AUTO)
   private long id;
   private String firstName;
   private String lastName;
   //getters and setters
}
```

By default, the URL path is derived from the uncapitalized, pluralized, simple class name of the domain class being managed.

Executing the Application

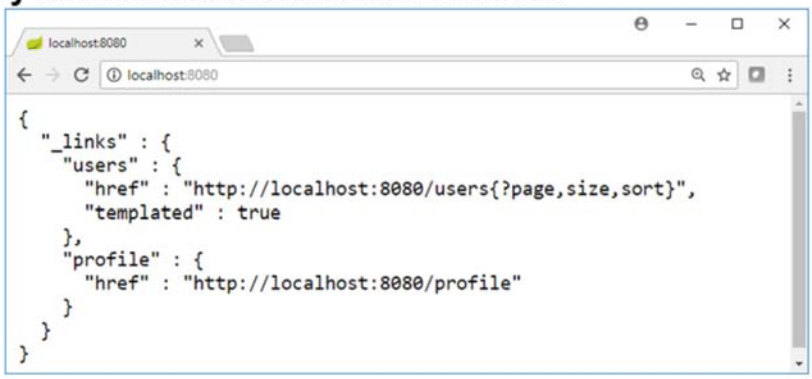
- An embedded server is started by executing application's main method
  - Runs `SpringApplication.run()` method

```
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;

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

```
 .   ____          _            __ _ _
 /\\ / ___'_ __ _ _(_)_ __  __ _ \ \ \ \
( ( )\___ | '_ | '_| | '_ \/ _` | \ \ \ \
 \\/  ___)| |_)| | | | | || (_| |  ) ) ) )
  '  |____| .__|_| |_|_| |_\__, | / / / /
 =========|_|==============|___/=/_/_/_/
 :: Spring Boot ::        (v2.1.3.RELEASE)

2013-04-23 13:38:54.776  INFO 18224 --- [           main]
```

Accessing the REST Repository

- Service endpoints can now be accessed
  - By default hosted at localhost:8080



- Repository endpoint already provides pagination and sorting options
- 'Profile' endpoint provides meta-data information

Testing the 'Service'

- Simple JUnit test can be developed
  - Utilizing `RestTemplate`

```
@RunWith(SpringJUnit4ClassRunner.class)   <-- JUnit 4 Test runner
@SpringBootTest                           <-- Run Spring Boot based test
public class ApplicationTests {

  @Test
  public void getUsers() {
      RestTemplate restTemplate = new RestTemplate();
      String url = "http://localhost:8080/users";
      ResponseEntity<String> response =
               restTemplate.getForEntity(url, String.class);
      Assert.assertEquals(HttpStatus.OK, response.getStatusCode());
   }
}
```

Overriding Default Properties

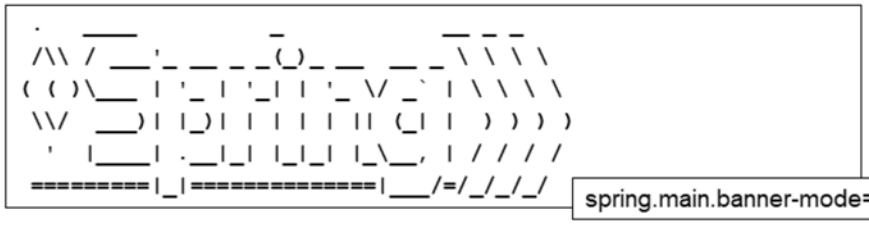
- Default port 8080 might not be available
- Alternative port can be configured
  - Within the annotation

```
@SpringBootTest(webEnvironment = WebEnvironment.DEFINED_PORT,
                properties = "server.port=8090")
```

  - In a properties file

```
server.port=8888
```

  - Using command line arguments, environment variables, …
- Even the Spring Boot ASCII logo can be disabled



Random Server Port

- Server port can be randomly selected
  - By setting the server port to 0 (zero)

```
server.port=0
```

  - Using the annotation

```
@SpringBootTest(webEnvironment = WebEnvironment.RANDOM_PORT)
```

- Port number is made available as property 'local.server.port'
  - Can be injected using SpEL expression

```
@Value("${local.server.port}")
private int serverPort;
```

  - Or using `@LocalServerPort` convenience annotation

```
@LocalServerPort
private int serverPort;
@Test
public void getUsers() {
   String url = String.format("http://localhost:%d/users", serverPort);
   …
}
```

Bootstrapping Spring Boot

- Spring Boot has two interfaces that allow for bootstrapping application startup:
  - `CommandLineRunner.run(String[])`
  - `ApplicationRunner.run(ApplicationArguments)`
- Execute code before completion of Spring Boot app start

```
@Component
public class AppStartupRunner implements CommandLineRunner {
   @Override
   public void run(String... args) throws Exception {
      //args references command-line arguments
   }
}
```

- Multiple application/command line runners can be registered
  - Load order can be defined using `@Order` or `Ordered` interface

org.springframework.boot.CommandLineRunner.run(String[])

ApplicationRunner

- ApplicationRunner wraps application arguments
  - Exposing ApplicationArguments type
    - Providing several convenience methods

```
public class ApplicationBootstrapper
             implements ApplicationRunner {
  ...
   @Override
   public void run(ApplicationArguments args) throws Exception {
    LOG.info("Boot started with option names: {}",
                                       args.getOptionNames());
   }
}
```

org.springframework.boot.ApplicationRunner.run(org.springframework.boot.ApplicationArguments)

Exercise 5: Create REST Repository using Spring Boot

~/StudentWork/code/spring.boot.flightservice/lab-code

Please refer to the written lab exercise and follow the directions as provided by your instructor

Session: Spring AOP

Introduction to Aspect Oriented Programming
Spring AOP

Introduction to Aspect Oriented Programming

Introduction to Aspect Oriented Programming

Spring AOP

Lesson Agenda

- Understand AOP basics – benefits and concepts
- Explain Crosscutting Concerns

Introduction to AOP

- Object-Oriented programming approaches an application as set of collaborating objects
  - Classes hide implementations behind interface
- Polymorphism allows for specialized implementations behind a generic interface
  - Allows changes to be made to the implementation without affecting the calling application
- OOP does not adequately address behavior that spans over multiple modules
  - Modules might be completely unrelated, but logging logic occurs in all of them
- Making changes to these crosscutting functionalities often affects the entire OO application

Within Object-Oriented programming, objects expose just their interface and the implementation of the object is completely hidden from the outside world.

By using the OO approach, in combination with polymorphism techniques, developers are capable of writing specialized implementations while still exposing a common (well-known) interface. A good example of this is the use of the JDBC API, where developers use the interfaces as specified by the JDBC specification. Database vendors provide a specialized implementation behind this interface. By 'hiding' the implementation behind a predefined interface, the implementation can be changed without affecting the client application.

Although OOP does a nice job in providing a clean separation between the different components within an application, it does not handle an implementation that is needed throughout the entire application. For example, logging functionality is often required within multiple components. When the logging requirements change (for example, change the logging implementation from log4j to the Logger API), this change will have an impact on a lot (if not all) components of an application.

Aspect Oriented Programming - AOP

- Aspect-Oriented Programming (AOP) complements OO programming
  - Allows for dynamic modification of the static OO model
  - Applications become easier to design, understand and maintain
- Aspect Oriented Programming is a different "paradigm" of programming
  - It has its own set of buzzwords
  - Implemented by multiple tools including:
    - Spring
    - AspectJ
    - ...

Programming Concerns

- When designing an application, two major concerns have to be taken into account
  - Core functional requirements (Business Logic)
  - Secondary or system-level requirements
    - Security, logging, etc.
- Secondary requirements tend to take focus away from core functionality
  - Architects not always aware of all system-level concerns that need to be addressed
  - Future developments are almost impossible to predict, making it hard to design an extensible system
  - Developers need to implement similar logic in multiple modules
    - Even when these modules are unrelated

When designing an application, the main concern should always be the implementation of the functional requirements that were defined by a particular use case. After all, you are writing an application to solve a particular business problem, not because you simply want to write log-files onto the file system. Besides the core-functional requirements, applications have a number of secondary requirements, like logging, security, persistence, etc.

Although these secondary requirements are a very important part of the application, a design that addresses all these requirements is often almost impossible to create. Even worse, the actual functionality that needs to be available in the application might become buried underneath all the 'details'. Architects do not always know the exact details of the system-level requirements on the platform. They might be able to specify that security is required, but not the actual security implementation that is going to be used. Unless you can predict the future, it is an extremely difficult task to design a system that is capable of being extended when future developments require additional functionality of the application. All the architectural problems aside, when implementing these secondary requirements directly into your component, you will find yourself writing similar logic in more than one component.

Crosscutting Concerns

- System-level concerns are often crosscutting concerns
  - Tend to affect multiple modules of the application
- Aspect-oriented programming focuses on separating crosscutting concerns
  - Allows for modularization of crosscutting concerns
- Enterprise applications often contain crosscutting concerns that decrease program modularity
- Crosscutting concerns include:
  - Logging
  - Transactions
  - Security
- Application requirements outside of core business logic

System-level requirements that span across multiple components are known as crosscutting concerns. Making a change to the system-level requirement often has an affect on multiple components within the application. Aspect-oriented programming focuses on the separation of these crosscutting concerns from the core functional requirements of the application.

When looking at the different components that make up an application, you will be able to see certain pieces of functionality re-appear in each of these components (logging, transactions, security, etc.). These 'concerns' span across multiple components of the application but, although they are very important to the final application, are most often an addition to the actual business case implemented by the component. Having these concerns implemented over and over again in all components not only makes maintenance a lot harder, it tremendously decreases program modularity.

Crosscutting concerns include

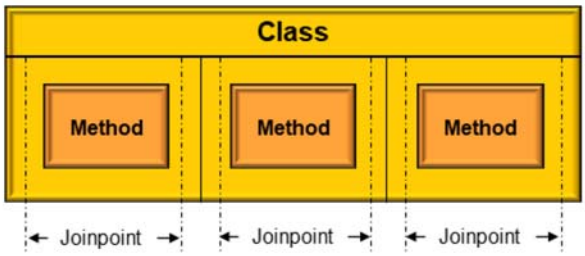
Logging: Whether or not you are using a logging framework, adding trace information to your application would require adding lines of code at the start and end of each method (to be more precise, you would have to add logging information at every exit point of the method, including when exceptions are thrown)

Transactions : Enterprise JavaBeans are a good example of how transaction information is declared outside of the code. The bean itself focuses on the business logic and transactional requirements are defined in the deployment descriptor of the bean.

Security : Adding security to your business components can be an complex and high maintenance task. Imagine what would happen if access rights are granted independently by each component. Changing security implementations or even updating access rights would involve making changes to all components.

AOP Definitions

- **Crosscutting concerns**
  - Secondary requirements, common in multiple modules within OO model
- **Joinpoints**
  - Points in program execution where an aspect can be called



- **Pointcut**
  - Definition of one or more joinpoints
  - 'glue' between joinpoint and logic that needs to be executed when joinpoint is encountered

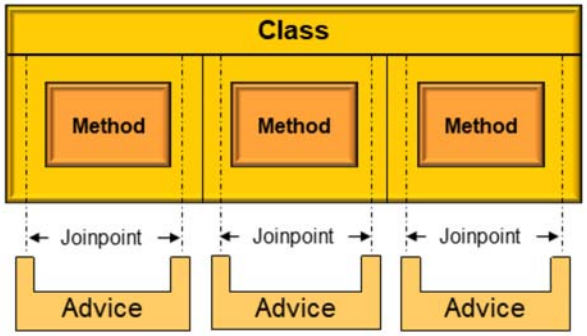
Crosscutting concerns are pieces of functionality that are needed within multiple components of the application.

Joinpoints are predefined points within the application at which calls to other modules can be made. For example: every method call can be defined as a joinpoint, to allow a logging mechanism to intercept these calls and report the entry and/or exit of this method

A pointcut is a collection of joinpoints. So when logging is required, all method joinpoints are combined into one pointcut. A pointcut is the glue between the joinpoint and the logic that needs to be executed when the joinpoint is encountered.

AOP Definitions (Advice / Aspect)

- Advice / Aspect
  - Code that is executed when predefined joinpoint is encountered

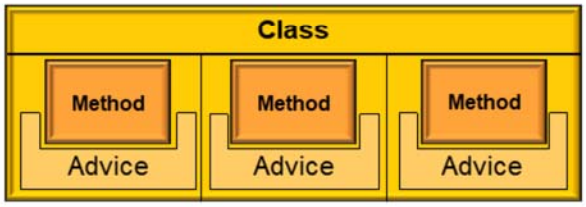


- Aspect is the "cross-cutting" functionality to be added
- Advice is the actual implementation of the aspect

An advice contains the actual logic that is to be executed when a joinpoint is encountered. As the name 'interceptor' already implies, it intercepts the operation, allowing additional functionality to take place, before the remaining part of the main module is executed. An Aspect is a class that implements the crosscutting concern. An Advice is the actual action that will take place (e.g. the method that gets executed).

AOP Definitions (Advisor / Interceptor)

- Advisor / Interceptor
  - Combines Advices and Pointcuts



- When using Spring, object is proxied

```
public class Catalog {
   public void addToCatalog(Item item){
      //do catalog stuff
   }
}
```

+

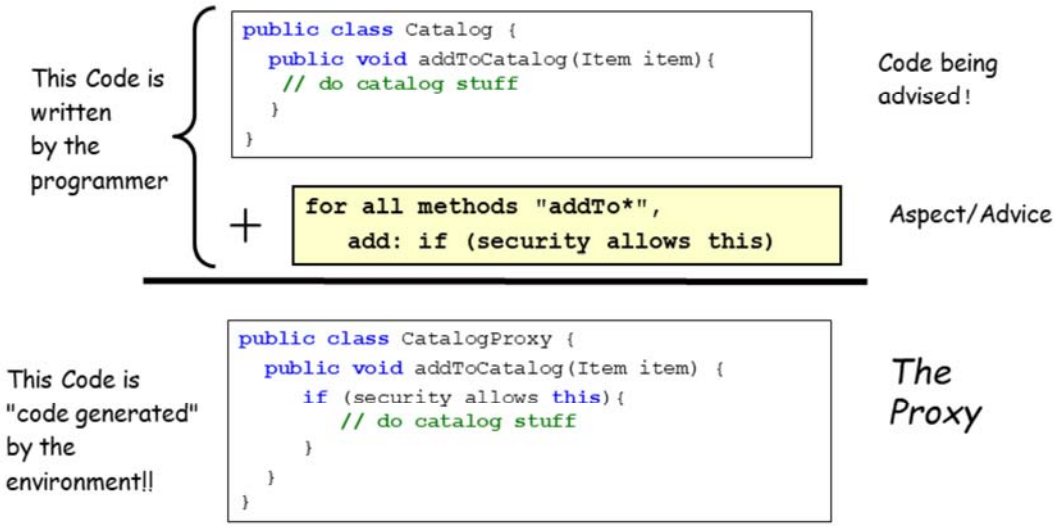
```
for all methods "addTo*",
   add: if (security allows this)
```

```
public class CatalogProxy {
  public void addToCatalog(Item item){
     if (security allows this){
        //do catalog stuff
     }
  }
}
```

Spring AOP applies these aspects to the target object at runtime (whereas other AOP frameworks are also capable of doing this at compilation time). Because of this 'restriction' Spring makes use of proxies to apply the aspect to the target component. The object returned to the client is therefore always a proxy class, which utilizes the target object to perform the business operations.

The code syntax for the Aspect is not accurate. It is a conceptual representation of how an Aspect works. The bottom left rectangle is an "aspect" it says to add "if (security allows this)" to all the methods named "addTo*".

What is an "Aspect"?



Advice Types

- Several types of advices exist for the implementation of crosscutting concerns
  - Spring AOP only supports joinpoints on method execution of Spring beans
- Before advice
  - Only called before the joinpoint
- After returning advice
  - Only called after a joinpoint
    - Not when method throws exception
- After (Finally) advice
  - Only called after a joinpoint (regardless of outcome)
- Around advice
  - Called before and after a joinpoint (regardless of outcome)

Whereas other AOP frameworks are capable of weaving the aspects at a more fine-grained level within the business component, Spring only provides support for method level weaving. Keep in mind that Spring utilizes proxies to add aspects to the component during runtime.

A 'Before advice' will only be called upon the Joinpoint is reached. When using Spring AOP, this means that the aspect is only invoked when the method is invoked.

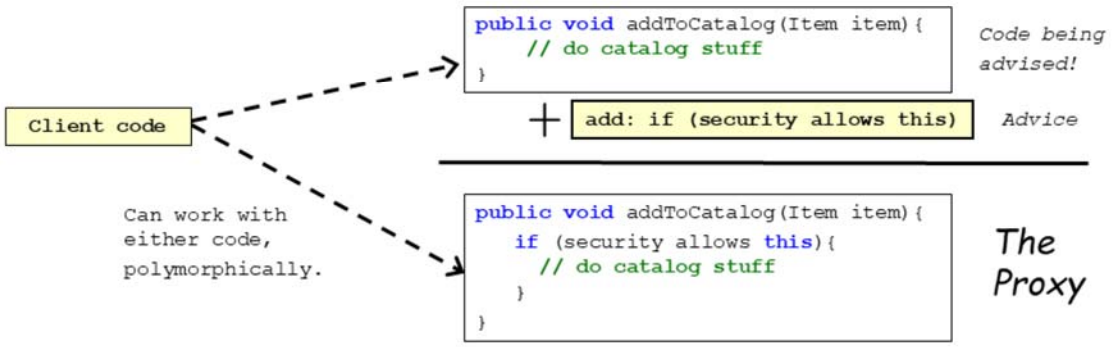
An ‘After returning advice’ will only be called after the execution of the business logic defined by the joinpoint but it will not be called upon when the business logic throws an exception.

A 'Finally advice' (or 'After advice') is always called after the execution of the business logic defined by the joinpoint, regardless of the outcome of the method.

An ‘Around advice’ will be called before and after a joinpoint. When using Spring AOP, this means that the advice will be called before the invocation of the method and also when the method returns or throws an exception.

Aspects and Decoupling

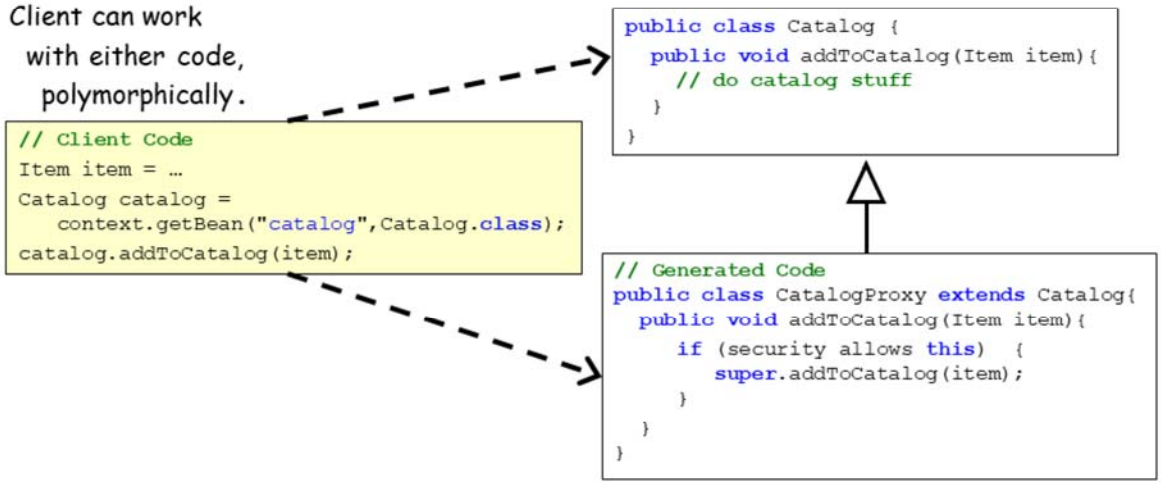
- **Code being advised** does not know it is being advised
  - It can work standalone without the advice
- **Advice code** will typically not know much about the code being advised
- **Client code** can work with either the original core code or advised version
  - Client sees the same interface either way



The client can use either the original class or the "advised" proxy version of the class. The client does not know which one it is accessing. This choice will be made in the Spring config file.

The Structure of the Proxy – Version 1

- Proxy code is generated
  - Proxy must be "type compatible" with code being advised

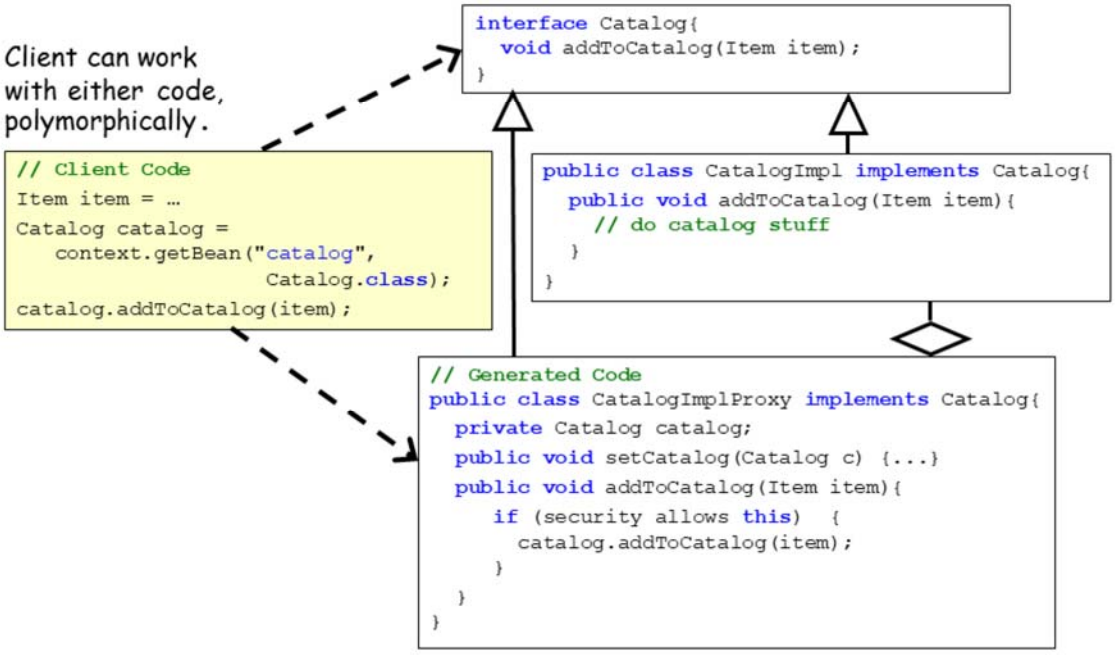


The proxy is code generated. The generated proxy must be "type compatible" with code being advised. There are two ways for "type compatibility" to occur: the proxy is a subclass of advised code or proxy and advised code implement one interface.

The Spring factory can return either the original Catalog class, or the CatalogProxy to the client. This technique uses the CGLib libraries to generate the proxy.

The choice of which is returned to the client depends on the configuration provided in the config file. Spring will "code generate" the proxy if it needs to.

The Structure of the Proxy – Version 2



An alternative is to have a shared interface - Use Interface inheritance + aggregation

This is not the precise code that is generated. All we care about is how things look from the perspective of the client. In this code generation style, Spring will, when needed, create the proxy and then inject an instance of the original class into the proxy. There will now be two objects – the proxy and the original target object.

This style uses the built-in to the JVM Dynamic Proxy capability. See the Javadocs for java.lang.reflect.Proxy. It is beyond the scope of this course to describe how this class works. Suffice it to say that it is the most powerful class in the entire JDK library.

The Spring factory can return either the original Catalog class, or the CatalogProxy to the client.

Tradeoffs Between Code Generation Styles

- CGLib and using subclassing
  - Original class must not be final
- Dynamic Proxies - Using Interfaces
  - There must be an "interface"
  - Spring encourages the use of interfaces
  - Code might have to be refactored to add an interface
- Performance
  - You have to try it both ways
  - Varies between JVMs

JVM generated Dynamic Proxies used to be slower than CGLib generated proxies. Recently (JVM 1.5+) JVM Dynamic Proxies seem to be faster. The only way to find out for sure is to try it on your JVM.

There are many JVMs available, not only the Oracle provided ones. Alternatives include IBM, Eclipse and BEA JVMs.

Cross Cutting Concerns – A 2D View

	Security	Transactions	Remoting	Billing	Auditing	Reporting	Monitoring
Shopping Cart	✓	✓				✓	
Credit Cards	✓	✓	✓		✓		
Shipping Service	✓		✓	✓			
Manufacturing		✓			✓		✓
Banking Services	✓		✓		✓		✓
Photo Retrieval			✓				

"Pick and Choose" the Features to add

Why is AOP Important?

- More declarative features are possible
  - Not just transactions and security
- No server is needed
  - AOP can be used in lightweight programs
  - AOP can be used in standalone programs
- No remoting is needed
- Runtime performance is very good

Summary Crosscutting Concerns

- Enterprise applications often contain crosscutting concerns:
  - Aspects of a program that span the design of the application
  - Decreasing program modularity
- These crosscutting concerns include:
  - Logging
  - Transactions
  - Security and others
- In short: application requirements outside of the core business logic

Spring AOP

Introduction to Aspect Oriented Programming

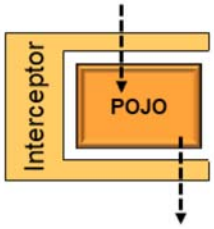
Spring AOP

Lesson Agenda

- Spring AOP in a nutshell
- `@AspectJ` support
- Spring AOP advice types
- AspectJ pointcut designators

Spring's AOP in a Nutshell

- Proxy-based framework
  - Utilizes the concept of interceptors
  - Based on the AOP Alliance API
  - Request to target object is intercepted before and after



- Interceptor remains separate from target object
  - Not compiled into target object (unlike AspectJ)

Spring utilizes the concept of proxies and interceptors to apply advices to the joinpoints of the components (beans).

The implementation of Spring AOP is based on the programming interface as defined by the AOP Alliance:

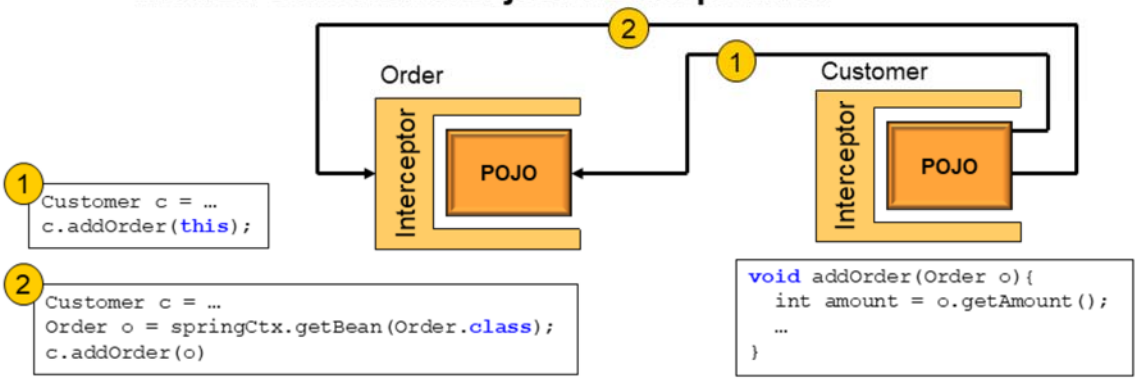
"We believe that Aspect-Oriented Programming (AOP) offers a better solution to many problems than do existing technologies such as EJB. AOP Alliance intends to facilitate and standardize the use of AOP to enhance existing middleware environments (such as Java EE), or development environments (e.g. IntelliJ and Eclipse). The AOP Alliance also aims to ensure interoperability between Java/JEE AOP implementations to build a larger AOP community.“

The image shows the interception of a method call to the target object. In this example, the call is intercepted before and after the actual method invocation.

Referring to the image shown above, you should notice that the actual target is not aware of the fact that it is being proxied (that the call was intercepted before and after its method invocation). As stated earlier, Spring utilized proxies and interceptors to support AOP. The actual implementation of the crosscutting concerns is placed in a separate object and is NOT added to the code of the target object. (As might be the case when using compile time weaving as supported by AspectJ)

Limitations of 'Dynamic' Interception

- Spring AOP is limited to method interception
  - Can not intercept field/variables
  - Can not intercept "inside" a method
- Target object remains available to the application
- Target object is unaware of proxy
  - Be wary of using `this` reference
  - Method calls within object are not proxied



The drawback of runtime weaving (as provided by Spring) is that the interception point can only be placed on methods. Only calls to methods on beans can be intercepted.

Since weaving only takes place at runtime, the target component must be made available to the Spring container. This means that this component also becomes available to the application without the proxy being applied to it (unless the bean is defined as an inner bean definition, in which case it does not have a public identity).

When using AOP, it is important to remember that the target object is not aware of the fact that it is being proxied. A proxied component should never return a 'this' reference as a result of a method invocation since invoking methods on the 'this' reference would result in a direct invocation of the method on the target, thereby bypassing the proxy.

AOP Concepts and Terminology

- Aspect : Implementation of crosscutting concern
- Joinpoint : A point of execution within a program
  - In Spring, AOP always represents method execution
- Advice : Action to be taken by aspect at joinpoint
  - around, before and after method invocation
- Pointcut : Predicate matching joinpoints
- Target object : Object to be advised by aspect(s)
  - In Spring, AOP will become proxied object
- AOP proxy : Object created by framework to add aspects
  - In Spring, AOP JDK dynamic proxy or CGLIB proxy
- Weaving : Linking aspects to objects
  - Spring AOP performs weaving at runtime
  - Can also be done at compile time (using AspectJ compiler)

When applying AOP to your application, it is important to understand the core concepts and terminology of AOP.

One of the concepts that provides the foundation of AOP is ‘weaving’. Weaving defines to process of applying functionality to the target object without making modifications to the target code. During the weaving process, one or more aspects are added to joinpoints on the target object.

In Spring, weaving happens at runtime through the use of AOP proxies. When using a AspectJ compiler, the weaving can also take place at compile time. Compile time weaving affects the build process but allows for interception at a more fine grained level than runtime weaving. Runtime weaving only allows for interception at method execution.

Annotation: @Aspect

- @Aspect annotated beans will be used to configure Spring AOP
- Class must explicitly be registered in context
  - Defining the bean in XML
  - Annotating the class as @Component and enable classpath-scanning

```
@Component
@Aspect
public class LoggingAspect {
    …
}
```

- Aspects can not be advised!
  - @Aspect annotated classes are excluded from auto-proxying

To implement a cross-cutting concern in Spring AOP, a regular Java class can be used. This class must be annotated using the @Aspect annotation. By annotating the class with this annotation, the Spring runtime will use this class to configure Spring AOP.

However, annotating the class using @Aspect does not mean that the class is automatically discovered when classpath-scanning is enabled within the context. The bean must explicitly be registered within the context, either by configuring it within the XML configuration file, or by adding one of the Stereotype annotations (and enabling classpath-scanning for the package in which the aspect has been placed)

It is not possible to define aspects to be applied to other aspects. In other words, aspect implementations cannot be advised themselves. By annotating a class using the @Aspect annotation, the class is excluded from auto-proxying

Annotation: `@AspectJ` Support

- Support must be enabled
  - By adding `EnableAspectJAutoProxy` to configuration

```
@Configuration
@EnableAspectJAutoProxy
public class JavaConfig {

}
```

  - Automatically proxy beans when advised by aspects

By enabling AspectJ support within the context, Spring allows us to define our aspects as regular Java classes. In order to support this, support for AspectJ must be enabled. This can be accomplished by adding the EnableAspectJAutoProxy annotation to the Spring configuration. This will automatically generate a proxy for every bean that has been advised by one or more aspects.

To enable @AspectJ support in an XML based configuration the <aop:aspectj-autoproxy/> element can be used.

Defining Pointcuts: @AspectJ

- Pointcuts define joinpoint(s) to which advice is applied
  - Defines when an advice is executed
- Pointcut is defined by a method and an expression
  - Method defines pointcut name and parameters
  - Expression defines method executions to be intercepted
- @Pointcut annotation holds pointcut designator

```
@Pointcut("execution(public * *(..))")
```

pointcut designator

- Pointcut method name defines pointcut name
  - Method must return void

```
@Pointcut("execution(public * *(..))")
public void allPublicMethods() {}
```

pointcut designator

To apply an advice to your application logic, you must define where it should be applied. A joinpoint is a single point of interception, when using Spring AOP the 'only' available joinpoint type is a method execution.

A pointcut defines to which method(s) an advice is to be applied. In the case of Spring AOP, a pointcut is a collection of one or more method references.

A pointcut is defined by defining an (often empty) method. The name of the method becomes the name of the pointcut, while the pointcut expression defines the methods to which an advice should be applied.

Pointcut designators will be explained shortly.

Advice Types

- Spring AOP advice runs before, after or around method executions
  - Before advice (`@Before`)
    - Runs before method is executed

```
@Before("…")
public void doLog() { … }
```

  - After returning advice (`@AfterReturning`)
    - Runs when method returns normally
  - After throwing advice (`@AfterThrowing`)
    - Runs when methods throws an exception
  - After (finally) advice (`@After`)
    - Runs independent of method result
  - Around advice (`@Around`)
    - Runs both before and after method execution

In Spring AOP, advices can 'only' be added to a method execution. But we do have a choice of when this advice is applied. An advice can be applied before the method is executed and/or when the method returns. When applying an advice on method return, we can define if this advice should only be applied when the method returns normally, or also when the method throws an exception.

Defining the Advice

- Method execution to be advised is defined by pointcut (expression)
  - Defined by a named pointcut
    - Defined in same class

```
@Pointcut("execution(public * *(..))")
public void allPublicMethods() {}

@Before("allPublicMethods()")
public void doLog() { … }
```

advice

    - Defined in other class

```
@Before("com.car.aop.PointcutConfig.allPublicMethods()")
```

  - Defined as a pointcut designator

```
@Before("execution(public * *(..))")
```

To apply an advice, the method that defines the advice must be annotated with an advice annotation, which contains the pointcut information that defines to which method the advice is to be applied.

In the example shown above the doLog method contains the implementation of the advice. The @Before annotation defines that the advice should be applied before the target method is invoked. The actual method(s) to which the advice is applied is defined by a pointcut. The @Before annotation can reference a pointcut method defined within the same class, or even a pointcut definition defined in a different class.

The advantage of pre-defined pointcut definitions is that they can be reused by multiple advice definitions (by simply referencing the pointcut method defining the pointcut). When a pointcut definition is not pre-defined, the pointcut expression can also be provided to the @Before method directly.

AspectJ Pointcut Designators

- Spring AOP supports subset AspectJ pointcut designators (PCD)
- within : Methods defined by types in specific packages
  - All types defined in package

```
within(com.car.rental.reservation.*)
```

  - All types defined in (sub) package

```
within(com.car.rental.reservation..*)
```

- this : Any method defined on proxy of type

```
this(com.car.rental.CarRentalService)
```

- target : Any method executed by target object of type

```
target(java.io.Serializble)
```

- args : any method with single parameter of type

```
args(java.time.LocalDate)
```

The pointcut expression that defines where the advice is to be applied is specified using a pointcut designator.

All pointcut designators defined above allow us to define pointcuts based on types of bean.

AspectJ Pointcut Designators (cont'd)

- Designators by annotation type
  - @target : Target *object* is annotated with annotation of type

```
@target(com.car.rental.log.LogMethods)
```

  - @args : Method with single parameter. Runtime argument is annotated with annotation of type

```
@args(javax.persistence.Entity)
```

  - @within : Target *type* is annotated with annotation of type

```
@within(com.car.rental.log.LogMethods)
```

  - @annotation : Executing method is annotated with annotation of type

```
@annotation(com.car.rental.log.Log)
```

Pointcuts can also be defined based on annotations that are present on the target object or method.

Notice that @within() is matched statically, requiring the corresponding annotation type to have only the CLASS retention. Diametrically, @target() is matched at runtime, requiring the same to have the RUNTIME retention

AspectJ Execution Pointcut Designator

- execution : matching method execution joinpoints

execution(modifiers return type type method-name (parameters) throws)

- Patterns can be used to define 'parts' of the expression
  - modifiers (optional) – Method visibility
    - public, protected, private, *
  - return type – Return type of the method
  - type (optional) – Declaring package or class
  - method-name – Method name
  - parameters – Method parameters
  - throws (optional) – Methods throwing this exception

The most flexible pointcut designator is the execution designator. It allows us to provide detailed information about the method to which an advice is applied. When defining an execution pointcut designator, the return type, method name and method parameter-types always need to be defined. All other parts of the expression are optional.

Execution Pointcut Designator Examples

- **Execution of any method**

```
execution(* *(..))
```

- **Execution of any method returning `java.util.List`**

```
execution(java.util.List *(..))
```

- **Execution of any public method**

```
execution(public * *(..))
```

- **Execution of any method starting with 'add' that might throw exception**

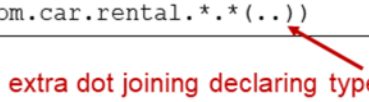
```
execution(* add*(..)) ReservationException
```

The execution pointcut designator allows the use of wildcards within the expression. To apply an advice to all methods, the expression * *(..) must be used. The first wildcard indicates any return-type, the second wildcard indicated any method name, while the two dots define zero or more parameters of any type.

Execution Pointcut Designator Types

- When specifying declaring type, trailing dot joins it to method-name
  - Any method on types defined in package

```
execution(* com.car.rental.*.*(..))
```



  - Any method on types defined in package with classname starting with

```
execution(* com.car.rental.Reservation*.*(..))
```

  - Any method on specific type

```
execution(* com.car.rental.ReservationService.*(..))
```

  - Any method on specific type with method-name starting with

```
execution(* com.car.rental.ReservationService.add*(..))
```

By adding more ‘parts’ of the execution designator, it is possible to further limit the set of method to which an advice is applied.

Execution Pointcut Designator Parameters

- Parameters pattern allow for definition of zero or more parameters
  - () matches methods that takes no parameters

```
execution(* *())
```

  - (..) matches methods with zero or more parameters

```
execution(* *(..))
```

  - (*) matches methods with one parameter (of any type)

```
execution(* *(*))
```

  - (<type>) matches method with one parameter (of type)

```
execution(* *(java.time.LocalDate))
```

  - (*,<type>,*) matches methods with three parameters
    - First and last can be of any type, the second must be of type

```
execution(* *(*,java.time.LocalDate,*))
```

The execution pointcut designator even allows us to define a pointcut based upon the parameter types of methods, as shown by the examples above.

The bean Designator

- Spring AOP defines one additional Pointcut Designator
  - The bean designator
- Name can be id or name of any managed bean
  - The * character can be used as wildcard

```
@Before("bean(reservation*)")
```

- Only supported in Spring AOP
  - Not supported by native AspectJ weaving

Spring defines a pointcut designator called ‘bean’ which allows us to apply advices to one or more beans defined within the context by referencing their bean name.

Keep in mind that this designator is only supported within Spring AOP. It can not be used when taking the native AspectJ weaving approach.

Combining Pointcut Expressions

- Pointcut expressions can be combined
  - using && or ||

```
//Within (sub) package AND
//single argument annotated with Entity annotation
@Before("within(com.car.rental..*) &&
         @args(javax.persistence.Entity)")
```

- Or negated (using !)

```
//All method execution exception on CarRentalService
@Before("execution(* *(..)) &&
   !this(com.car.rental.CarRentalService)")
```

Pointcut expressions can be combined to define a more complex pointcut definition.

JoinPoint

- Each advice method may declare parameter of type `org.aspectj.lang.JoinPoint`
  - Must be first parameter in method signature
- `JoinPoint` provides access to
  - Method parameters
  - The proxy class
  - The target object
  - The method signature of the method being executed

```
@Before("execution(* *(..))")
public void logMethodExecution(JoinPoint jp) {
   Object target = jp.getTarget();
   Signature signature = jp.getSignature();
   String methodname = signature.getName();
   …
}
```

Sometimes the advice needs information about the joinpoint on which it is applied. To access the join point information, the advice method may declare a parameter of type JoinPoint. This parameter must be the first parameter in the method signature.

The Before and After (finally) Advice

- Before advice is declared using `@Before`
  - Executed before method is invoked

```
@Before("execution(* *(..))")
public void logMethodExecution(JoinPoint jp) {
    ...
}
```

- After advice is declared using `@After`
  - Executed when method returns or throws exception

```
@After("execution(* *(..))")
public void logMethodReturn(JoinPoint jp) {
    ...
}
```

By annotation a method using @Before or @After, we can specify that the logic within should be executed before the method is executed or after it returns.

Keep in mind that the @Before method can not 'block' the call to the target object and the @After method should be able to deal with both a normal method return and a method throwing an exception

After Returning Advice

- Declared using `@AfterReturning`
- Executed when method returns normally
  - Not when exception is thrown
- Method result can be made available to advice
  - The `returning` attribute must correspond parameter name

```
@AfterReturning(pointcut = "execution(* get*(..))", returning = "result")
public void logResult(JoinPoint jp, Object result) {
   …
}
```

- A returning clause restricts method matching
  - To methods returning parameter type

```
@AfterReturning(pointcut = "execution(* get*(..))", returning = "result")
public void logResult(JoinPoint jp, List<String> result) { … }
```

The @AfterReturning advice will only be invoked when the method returns normally (not throwing an exception). The method result can be made available by defining a method parameter to the advice method. The parameter name of the parameter representing the result object must be defined using the returning attribute of the @AfterReturning annotation.

While the parameter type holding the return value can be defined as Object, it is possible to define a more restrictive type. By doing so the number of methods to which the advice is applied is lower. While the pointcut expression in the example shown above still defines that is should be applied to any method, independent of their return types, most often the pointcut expression will already define the return type to match the parameter type of the advice method.

After Throwing Advice

- Declared using `@AfterThrowing`
- Executed when method exits by throwing exception
- Exception thrown can be made available to advice
  - The `throwing` attribute must correspond parameter name

```
@AfterThrowing(pointcut = "execution(* *(..))",
               throwing = "exception")
public void logException(JoinPoint jp, RentalException exception) {
   …
}
```

- A `throwing` clause restricts method matching
  - To methods throwing given type

The @AfterThrowing annotation is applied to advice methods that need to be invoked when the method throws an exception. The exception object that was thrown by the method can be defined as a parameter of the advice method, as long as the parameter name of the parameter is defined within the throwing attribute of the @AfterThrowing advice.

Around Advice

- Declared using `@Around`
  - Allows interception both before and after method call
  - Can prevent method execution on target
  - Allows before and after state to be shared
- First parameter of advice method must be of type `ProceedingJoinPoint`
  - Method implementation must call `proceed` method
    - Can be invoked zero or more times in method

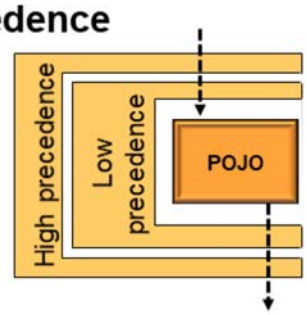
```
@Around("execution(* *(..))")
public Object around(ProceedingJoinPoint jp) throws Throwable{
   // before target-method invocation
   Object returnValue = jp.proceed();
   //after target-method invocation
   return returnValue;
}
```

- Return value of advice is value returned to client

The Around advice is the most flexible advice of all. It will be applied before and after the method execution. When defining an Around advice, the first parameter of the method must be of type ProceedingJoinPoint. Within the method implementation the proceed method of the ProceedingJoinPoint object must be called to call the target method (or other advice in the callstack). When the proceed method returns the target method was invoked and a result object might be available.

Advice Ordering

- Multiple advices may be applied to same joinpoint
- Advice execution takes place on precedence
  - Highest precedence wins (on way in)



- When two advices for the same joinpoint are defined in same aspect, execution order is undefined

```
@Component @Aspect
public class SampleAspects {
   @Around("execution(* *(..))")
   public Object around(ProceedingJoinPoint jp) throws Throwable { … }
   @Before("execution(* *(..))")
   public void logMethodExecution(JoinPoint jp) {…}}
```

When using AOP multiple advices can be configured for the same JoinPoint (e.g. Logging advice, Security advice, and Transaction advice). The runtime will first execute the advice with the highest precedence before invoking the method, which also means it will be invoked last once the method returns.

Until now we have not defined how you would be able to define the precedence of an advice. As a matter of fact, when two advices are defined within the same aspect, the order in which they are executed is undefined.

Advice Ordering (cont'd)

- Aspect can be annotated using `@Order`
  - Or implement `Ordered` interface
- The lower value has higher precedence

```
@Component @Aspect
@Order(500)
public class AroundAspects {
   @Around("execution(* *(..))")
   public Object around(ProceedingJoinPoint jp) throws Throwable {
      …
   }
}
```

```
@Component @Aspect
@Order(100)
public class BeforeAspects {
   @Before("execution(* *(..))")
   public void logMethodExecution(JoinPoint jp) {
      …
   }
}
```

The class that has been annotated with the @Aspect annotation may also be annotated using the @Order annotation (or implement the Ordered interface). The integer value that is defined within the @Order annotation (or returned by the getOrder method of the Ordered interface).

The lower the value, the higher the precedence of the advices defined within the Aspect.

Note: The value is defined by an Integer, meaning that the value can also be a negative value! Advices defined within and Aspect with the value set to Integer.MIN_VALUE have the highest precedence.

Spring AOP and Spring Boot

- Spring Boot provides starter dependency for AOP

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

Aspects [Core]
Create your own Aspects using Spring AOP and AspectJ

  - Enables autoconfiguration

```
@SpringBootApplication
public class Application { … }
```

```
@Configuration
@ConditionalOnClass({ EnableAspectJAutoProxy.class,
Aspect.class, Advice.class, AnnotatedElement.class })
@ConditionalOnProperty(prefix = "spring.aop", name = "auto",
havingValue = "true", matchIfMissing = true)
public class AopAutoConfiguration {
```

- `EnableAspectJAutoProxy` no longer needs to be added to configuration

When bootstrapping your application using Spring Boot, you can make sure of the spring-boot-starter-aop starter dependency.

The @SpringBootApplication annotation that is added to the 'main' configuration class contains the @EnableAutoConfiguration annotation. This auto configuration is accomplished using Configuration classes which use @Conditional type annotations (like @ConditionalOnClass and @ConditionalOnProperty) to scan the classpath and look for key classes (e.g. Aspects). Once found the support for AOP will automatically be enabled.

Pointcuts in Spring Boot

- To apply AOP, classes need to be proxied
  - Target class should be implementation of interfaces or not be defined as final
- When defining 'generic' pointcut, all classes on classpath will be proxied

```
@Around("execution(* *(..))")
```

- Spring Boot defines number of final classes
  - Resulting in exception at startup

```
Error creating bean with name
'org.springframework.boot.autoconfigure.AutoConfigurationPackages'
```

- Pointcut should be limited to application classes

```
@Around("execution(* com.application..*.*(..))")
```

In order to be able to apply Aspects to the Spring Bean, Spring needs to be able to proxy the target classes. To accomplish this, the target class should either have been defined using interfaces or at least should not be defined as final.

When defining a very generic pointcut (in the example above 'all method' invocations available public methods of every class on the classpath will have to be proxied. When using Spring AOP in a Spring Boot application, this cannot be accomplished because some of the classes that are being used by Spring Boot are defined as final and can therefore not be proxied.

Exercise 6: Spring AOP: Adding Interceptors

~/StudentWork/code/spring-aop/lab-code

Please refer to the written lab exercise and follow the directions as provided by your instructor

Session: Persistence in Spring

Transaction Management in Spring
Spring JDBC

Transaction Management in Spring

Transaction Management in Spring

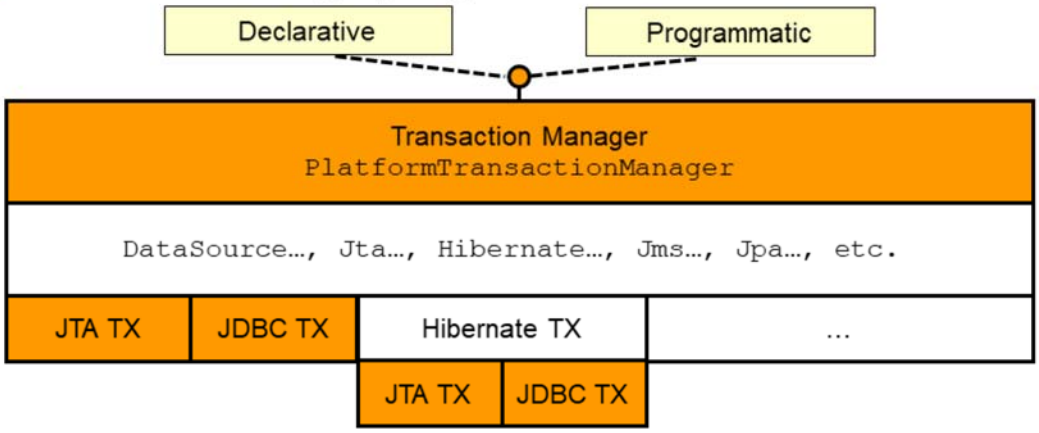
Spring JDBC

Lesson Agenda

- Understand Transaction Demarcation within Spring
- Configure the `PlatformTransactionmanager`
- The `@Transactional` annotation

Spring Transactions

- Consistent API across different transaction APIs
  - e.g. local JDBC transactions or JTA
- Programmatic demarcation using transaction abstraction within Spring
- Declarative using Spring AOP or annotations



Spring provides a consistent API for managing transactions across a variety of transaction APIs

Transaction Manager

- The transaction manager is encapsulated by `PlatformTransactionManager`
- Transaction manager has several implementations for specific technologies
  - `DataSourceTransactionManager`
  - `JtaTransactionManager`
  - `HibernateTransactionManager`
  - `JmsTransactionManager`
  - ...
- Transaction manager can be used directly
  - (one of the programmatic demarcation implementation models)
- Transaction is associated with single thread of execution

The PlatformTransactionManager interface defines an abstraction to transaction management and is therefore not tied into any particular type of transaction management. Because of this approach, transaction management can easily be mocked during development, replaced by local transaction management during testing and finally replaced by a distributed transaction manager when the component is taken into production.

Depending of the persistence technologies used and, more importantly, the environment in which the component will be used, a particular implementation of the PlatformTransactionManager is used. Spring comes with a variety of implementations that can be used out of the box.

PlatformTransactionManager Interface

```
public interface PlatformTransactionManager {
  TransactionStatus getTransaction(TransactionDefinition td);
  void commit(TransactionStatus status);
  void rollback(TransactionStatus status);
}
```

- All methods throw `TransactionException`
  - Subtype of `RuntimeException`
- `getTransaction` returns an active or new transaction
  - Based on information in the `TransactionDefinition`
- `TransactionStatus` is used to obtain or influence the outcome of the transaction

```
public interface TransactionStatus {
  boolean isNewTransaction();
  boolean hasSavepoint(); // based on nested tx
  void setRollbackOnly();
  boolean isRollbackOnly();
  boolean isCompleted();
}
```

If you have every developed components which run in a transactional environment, the methods of this interface should look familiar to you. These methods provide the functionality to monitor and 'manage' the transaction. When all operations within an atomic unit of work complete without errors, commit will be invoked; this confirms that all changes made (and/or messages sent) can be made permanent. When the operation did not complete successfully, rollback should be used, which indicates that all changes made to the data (and/or messages sent) should be ignored.

When errors occur while managing the transaction using these methods, a TransactionException will be thrown. The getTransaction method returns a transaction. Whether this transaction is a new transaction or an existing transaction depends on the parameters specified for this transaction.

The TransactionStatus object that is being returned by the getTransaction method contains several methods that can be used to query the state of the current transaction.

TransactionDefinition

- Defines Spring-compliant transaction properties
  - Name : Transaction description
  - PropagationBehavior : Behavior when method is invoked while transaction already exists
  - IsolationLevel : Degree of isolation from other transactions
  - Timeout : Time out period before transaction is rolled back automatically
  - ReadOnly : Transaction not used to modify data

```
public interface TransactionDefinition {
   int getPropagationBehavior();
   int getIsolationLevel();
   int getTimeout();
   boolean isReadOnly();
   String getName();
}
```

Whether a new or an new transaction is returned by the getTransaction method, depends on the definition (properties) of the transaction.

Transaction Propagation Attributes

- NOT_SUPPORTED
  - Will not start a transaction
  - Existing transaction association is suspended
- SUPPORTS
  - Will not start a transaction
  - Existing transaction association is not suspended
- REQUIRED
  - Will use existing transaction
  - If not in TX, will start new transaction, terminate when done
- REQUIRES_NEW
  - Always starts new transaction, terminates when done
  - Suspends existing transaction association

NOT_SUPPORTED : A transaction is not started for the thread, and if the thread already has a transaction, it is disassociated from it while running in this method.

SUPPORTS : If the thread has a transaction, it will run the method in the context of that transaction. If it doesn't have a transaction the container will not provide one.

REQUIRED : If the thread does not have a transaction, the container will provide one. If the container provides the transaction, then the container will also terminate the transaction when the thread returns.

REQUIRES_NEW : Regardless of whether the thread has a transaction, the container will provide a new one and terminate it when the thread returns.

Transaction Propagation Attributes (cont'd)

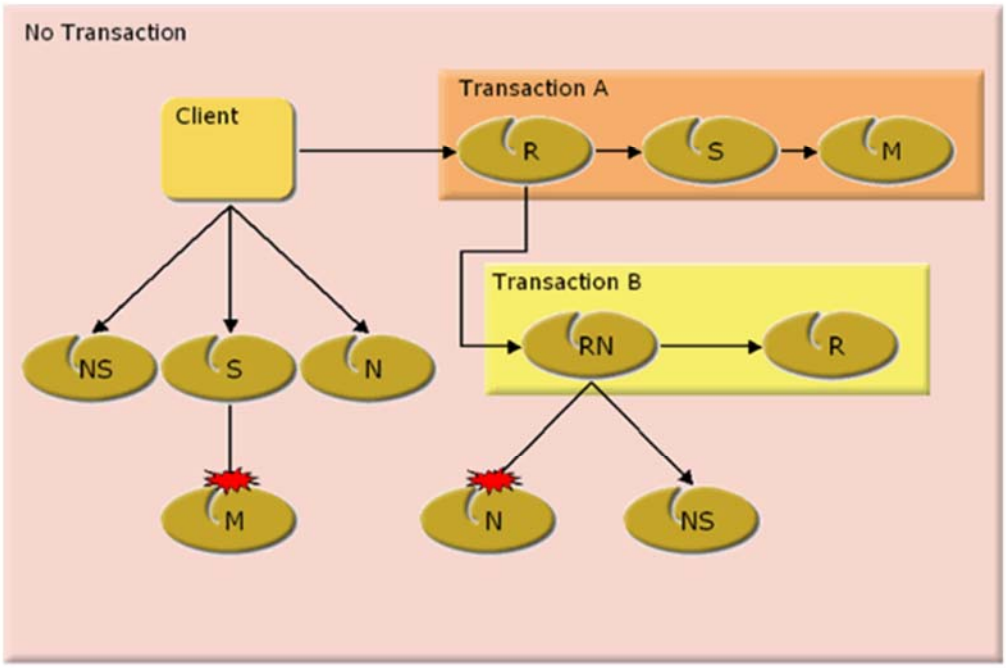
- MANDATORY
  - Must be called within a transaction
- NEVER
  - May never be called while in a transaction
- NESTED
  - Utilizes single transaction with multiple save points

MANDATORY : If the thread does not already have a transaction, an exception is thrown.

NEVER : If the thread has a transaction, an exception is thrown.

NESTED : uses a single transaction with extra save points, allowing the inner transaction to rollback without affecting outcome of outer transaction.

Transaction Propagation Scenarios



The diagram above illustrates some transaction scenarios The non-transactional client and all the other beans not enclosed in a box are running without a transaction. The two boxes represent transactions.

When the client invokes methods on beans with attributes Not supported (NS), Supports (S) or Never (N), the called beans will run without an transaction. If the client would invoke a method on a bean with an attributed of Required (R) a transaction is started; Transaction A in the example above (The same would happed if the attribute is Required New, which is not illustrated)

When the Bean in Transaction A calls upon beans with Supports (S) or Mandatory (M), the called beans will run in the same transaction (Transaction A in the example)

When a bean with attribute Requires New (RN) is invoked a new transaction is started (Transaction B in the example above)

Any time a bean is invoked with an attribute of Not Supported (NS) the transaction of the caller is suspended.

When a method with TX attribute Mandatory (M) is invoked from a non-transactional context, the invocation will fail, the same for Never (N) when invoked from a transactional context.

Isolation Level Concepts

- Dirty Read – Read a value that is later rolled back:
  - Two transactions
    - First sets a value; second reads it
    - First then rolls back
- Non Repeatable Read – Read a value, then later in the same transaction get a different value:
  - Two transactions
    - First reads a value; second changes the value
    - First re-reads value
- Phantom Read – Row does not exist, then later, row does exist:
  - Two transactions
    - First selects rows
    - Second adds a row
    - First selects rows again

Isolation Level - Constants

- ISOLATION_DEFAULT
  - Use default isolation level of the underlying data store
- ISOLATION_READ_UNCOMMITTED
  - Dirty reads, non-repeatable reads, phantom reads can occur
- ISOLATION_READ_COMMITTED
  - Dirty reads are prevented; non-repeatable reads and phantom reads can occur
- ISOLATION_REPEATABLE_READ
  - Dirty and non-repeatable reads are prevented; phantom reads can occur
- ISOLATION_SERIALIZABLE
  - Dirty, non-repeatable, and phantom reads are prevented

The constants shown above are defined by the org.springframework.transaction.annotation.Isolation enumeration.

Configuring the Transaction Manager

- `PlatformTransactionManager` needs to be configured for context
- Implementation depends on 'environment' used
  - Using a JDBC DataSource

```
@Bean
public AbstractPlatformTransactionManager txManager() {
   return new DataSourceTransactionManager(dataSource());
}
@Bean
public DataSource dataSource() {   …   }
```

  - Using JTA
    - Does not need to reference a resource

```
@Bean
public AbstractPlatformTransactionManager txManager()
{
   return new JtaTransactionManager();
}
```

An instance of the PlatformTransactionManager needs to be configured within the Spring context. The actual implementation class that needs to be configured depends on the 'storage environment' (e.g. JDBC, Hibernate, etc.) that is being used.

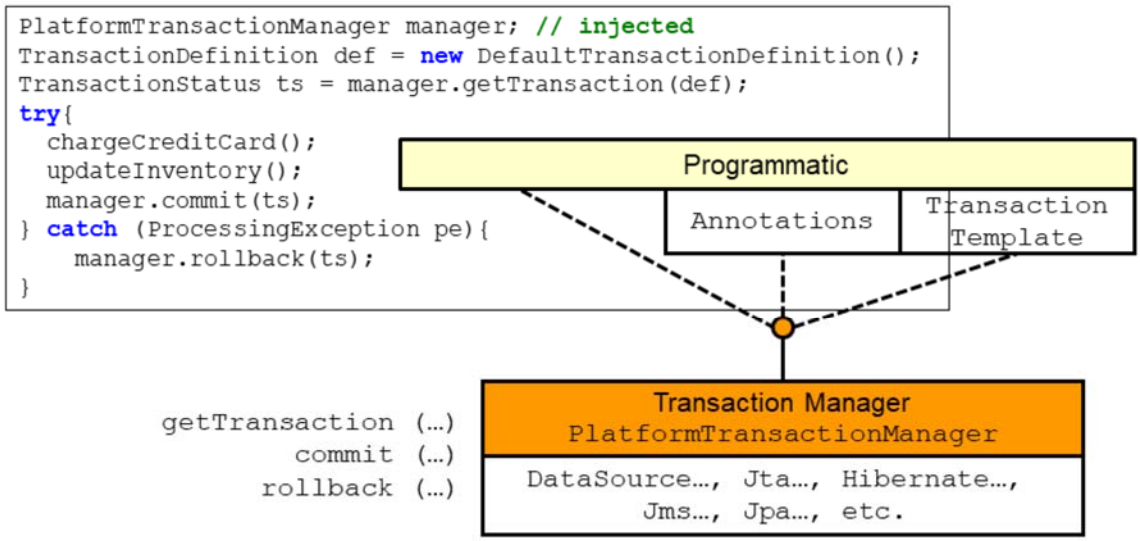
The transaction manager can be configured in XML, instead of configuring the transaction manager using JavaConfig:

```
<bean name="txManager"
class="org.springframework.jdbc.datasource.DataSourceTransactionManager">
  <property name="dataSource" ref="datasource"/>
</bean>

<bean name="datasource"
class="org.springframework.jdbc.datasource.DriverManagerDataSource">
    …
</bean>
```

A JtaTransactionManager does require a reference to a DataSource (or any other resources), since it relies on the container's transaction management infrastructure.

Programmatic Transaction

- **Used when transaction scope needs to be managed from within the code**



Programmatic transaction demarcation can be used when the scope of a transaction needs to be managed directly from within the code. Although this might sound like the most logical place, keep in mind that when a component manages the start and end of a transaction from within, this component can never participate as part of a larger transaction.

Notice that the PlatformTransactionManager is injected into the bean. You might control the scope you the transaction yourself, but since the transaction manager is injected into the bean, the bean can be used in a non distributed environment (using local transaction), but can also be used in a distributed environment, using distributed transaction (e.g. JTA)

The TransactionTemplate

- Utilizes callbacks for managing resources and handling exceptions

```
TransactionTemplate txTemplate = new TransactionTemplate(txManager);

Double totalPrice = txTemplate.execute(txStatus -> {
   RentalReservation reservation = reservationService.getReservation(id);
   double rentalPrice = calculateRentalPrice(reservation);
   creditCardService.chargeCreditCard(creditCardNumber,
                                      expirationDate, rentalPrice);
   reservationService.confirmReservation(reservationID);
   return rentalPrice;
});
```

```
public interface TransactionCallback<T> {
   T doInTransaction(TransactionStatus status);
}
```

Instead of managing the transactions manually, a TransactionTemplate can be used (which requires a reference to a transaction manager). The template utilizes callback for managing resources within a transactional context, while handling all transaction management even when exceptions occur while executing the service logic.

Declarative Transactions

- Specify transaction requirements in "declarative" fashion
  - Within the Spring configuration (XML)
  - Using `@Transactional` annotation within bean implementation
- Utilizes combination of `TransactionInterceptor` and generic AOP `ProxyFactoryBean`

A more flexible approach to transaction management is using declarative transactions. Instead of adding the transaction management code to the implementation of the bean, the transactional requirements of the bean methods will now be defined externally. This means that transactional requirements can be defined in an external configuration file, or by supplying the transaction meta-data information by adding the Transactional annotation to the bean (method)

Declarative transaction management relies on the use of proxies and interceptors to start and stop the transactions.

Benefits of Declarative Transactions

- Transactional requirements not comingled with code
  - Can be defined in separate file
- Transactional requirements are easy to understand
  - Simpler syntax than programmatic transactions
- Transactional requirements are easy to modify
  - Can be modified at deploy time
- Service bean is reusable in multiple contexts

By separating transaction logic from the business logic, it becomes easier to maintain the application since we can now focus on only the business logic. At the same time, configuration of the application can take place without the need of modifying the source code. A component can now be used in a different transactional context by just defining different transactional requirements.

The `@Transactional` Annotation

- Used to define transactional settings within the code
  - Can be used on interfaces, classes and public methods

`@Transactional` attributes			
Property	**Description**	**Type**	**Default**
`transactionManager`	Qualifier or name of specific PlatformTransactionManager bean	String	<empty>
`propagation`	Transaction propagation	Propagation	***REQUIRED***
`isolation`	Transaction isolation level	Isolation	***DEFAULT***
`timeout`	Transaction timeout (seconds)	**int**	-1
`readOnly`	Is transaction readOnly?	Boolean	false
`rollbackFor`	Comma separated list of exception classes resulting in rollback	Class<? extends Throwable>[]	<empty>
`rollbackForClassName`	Comma separated list of exception names resulting in rollback	String[]	<empty>
`noRollbackFor`	Comma separated list of exception classes NOT resulting in rollback	Class<? extends Throwable>[]	<empty>
`noRollbackForClassName`	Comma separated list of exception names NOT resulting in rollback	String[]	<empty>

Instead of specifying the transactional requirements of service methods within the configuration file, this type of information can also be added to the class using annotations.

The @Transactional annotation can be applied to interfaces, classes and public method within a class. This way the transactional requirements of each method can be defined directly within the class

Declarative Transaction Annotations

- Transaction rules can be defined in service bean code
  - Useful when service layer object should always be deployed with same transaction rules

```
@Service
public class CreditCardServiceImpl implements CreditCardService {
   @Transactional(propagation=Propagation.REQUIRED)
   public void chargeCreditCard(String ccNumber,
                           YearMonth expirationDate, double amount){
      …
}
```

- `@EnableTransactionManagement` annotation needs to be added to configuration

```
@Configuration
@EnableTransactionManagement
public class RepositoryConfig {
   …
}
```

- Code itself does not have to deal with the transactions

Instead of defining the transactional requirements in XML, the @Transactional annotation can be added to interfaces, classes and/or methods. As you can see, this is a much simpler approach than using the XML configuration and is extra useful when the service layer should always be used within a transactional context.

Transaction management using the @Transactional annotation must be enabled within the context. This can be accomplished by adding the @EnableTransactionManagement annotation to the configuration class.

When using XML based configurations all you have to modify the Spring config file to turn on annotation driven @Transactions using the tx:annotation-driven element.

```
<tx:annotation-driven  transaction-manager="txManager"/>
```

Applying Transactions

- Both client and service are unaware of transaction(s)
  - Transactions will be automatically applied as configured
- The Client :

```
CarRentalService service = context.getBean(CarRentalService.class);
LocalDate today = LocalDate.now();
LocalDate returnDate = today.plusDays(14);
int reservationID = service.addReservation(CONVERTIBLE, today, returnDate);
service.payRental(reservationID, "378282246310", YearMonth.of(2021,APRIL));
```

- The Service (method) :

```
public void payRental(Integer id, String ccNumber, YearMonth expires) {
   RentalReservation reservation = reservationService.getReservation(id);
   double price = calculateRentalPrice(reservation);
   creditCardService.chargeCreditCard(ccNumber, expires, price);
   reservationService.confirmReservation(reservationID);
}
```

Now that the transactional requirements have been defined, the code will be executed within a transaction that is managed by Spring. Both the client and the service implementation code are unaware of the fact that transactions are being used.

Exceptions and Transactions

- Unchecked exceptions thrown by transactional method result in rollback
  - Framework marks transaction for rollback
- Checked exceptions do not result in rollback
  - Unless `rollbackFor` attribute has been defined

```
@Transactional(propagation = Propagation.REQUIRED,
               rollbackFor = PaymentRejectedException.class)
public void chargeCreditCard(String ccNumber, YearMonth expires,
               double amount) throws PaymentRejectedException {
   …
}
```

When the container is managing the transactions of the application it should also know when the transaction is committed and when it is rolled-back.

When the transactional method returns without exceptions, the Spring context will initiate a commit. When the method throws an unchecked exception (any subtype of RuntimeException) it will initiate a rollback, but when the method throws a checked exception, the transaction will commit.

This behavior can be altered by using the rollbackFor and noRollbackFor attributes of the @Transactional annotation

Spring JDBC

Transaction Management in Spring
Spring JDBC

Lesson Agenda

- Understand an overview of Spring JDBC support
- Define `DataSources`
- Understand Spring's JDBC exceptions and the `SQLExceptionTranslater`
- Create JDBC DAOs
- Use the `JdbcTemplate` class
- Map rows to `Objects` using the `JdbcTemplate`
- Map data to `Objects` using SQL helper objects (operation classes)

Spring's Support of JDBC Functionality

- Obtaining DataSources
- DAO support classes for JDBC
  - `JdbcDaoSupport`
  - `JdbcTemplate` template class
- Tighter exception model
  - Translation of native error codes to Spring exception hierarchy
- JDBC *Operation* classes
  - Making database access more object-oriented

Spring provides a number of ways to obtain a datasource. This lesson will introduce helper classes and configuration parameters which will allow you to obtain a reference to a data source in a flexible way.

Spring provides support and template classes to ease the development of data access implementations. Since this lesson focuses on implements in JDBC, the JdbcDaoSupport and the jdbcTemplate classes will be explained here.

Most methods within the JDBC API declare to throw only a single type of exception (SQLException). Whether the error indicates a primary key violation or a corrupt SQL statement, you are always presented with the same type of exception. Determining the actual cause of the exception not only proved cumbersome, but most often also required the interpretation of error codes, specific for the actual data store residing underneath. Spring provides us with an exception translation mechanism, which interprets the SQLException (which, to make matters even worse is a checked-exception) and throw a more specific exception, depending on the cause of the error. The exceptions being thrown by Spring are also implemented as runtime (unchecked) exceptions.

Finally, operation classes can be developed, by utilizing Spring template classes, which really encapsulate data access.

Obtaining a DataSource (JNDI)

- Using JNDI to lookup a DataSource
  - Uses the generic JndiObjectFactoryBean

```
@Bean(name="datasource")
public DataSource prodDataSource() {
    JndiObjectFactoryBean jndiFactory = new JndiObjectFactoryBean();
    jndiFactory.setJndiName("jdbc/ReservationDataSource");
    jndiFactory.setResourceRef(true);
    jndiFactory.setProxyInterface(javax.sql.DataSource.class);
    return (DataSource)jndiFactory.getObject();
}
```

  - Setting the property setResourceRef to true prefixes the jndiName with java:comp/env/

A DataSource reference can be obtained in a variety of ways. Often, the DataSource is obtained using a JNDI lookup.

```
<bean name="dataSource"   class="org.springframework.jndi.JndiObjectFactoryBean">
  <property name="jndiName"
        value="java:comp/env/jdbc/userrealm" />
</bean>
```

Defining a DataSource

- Using one of the DataSource implementations
  - (e.g., Apache Commons DBCP's BasicDataSource or PoolingDataSource)
- Using one of the implementations provided by Spring
  - (e.g., DriverManagerDataSource, SingleConnectionDataSource)

```
@Bean
public DataSource dataSource() {
    DriverManagerDataSource ds = new DriverManagerDataSource();
    // org.apache.derby.jdbc.ClientDriver
    ds.setDriverClassName(ClientDriver.class.getCanonicalName());
    ds.setUrl("jdbc:derby://localhost:50505/CarRental");
    ds.setUsername("sa");
    ds.setPassword("password");
    return ds;
}
```

The DriverManagerDataSource is a convenience class that can be used to configure a DataSource, using a DriverManager, notice that all database properties which you would normally define programmatically are now available outside of your code.

Example of defining the DataSource in XML:

```
<bean name="datasource"
class="org.springframework.jdbc.datasource.DriverManagerDataSource">
  <property name="driverClassName"
        value="org.apache.derby.jdbc.ClientDriver" />
  <property name="url"
        value="jdbc:derby://localhost/springdemos"/>
  <property name="username" value="root" />
  <property name="password" value="masterkey" />
</bean>
```

Defining a DataSource (cont'd)

- Provide a custom DataSource implementation
  - (e.g., using Spring's base class AbstractDataSource)
- Using a DataSource pool
  - Use an Apache Commons DataSource Pool

```
@Bean
public DataSource dataSource() {
    BasicDataSource ds = new BasicDataSource();
    ds.setDriverClassName(ClientDriver.class.getCanonicalName());
    ds.setUrl("jdbc:derby://localhost:50505/CarRental");
    ds.setUsername("sa");
    ds.setPassword("password");
    ds.setInitialSize(2);
    ds.setMaxTotal(10);
    ds.setMaxIdle(5);
    return ds;
}
@PreDestroy
public void dataSourceCloseConnection() throws SQLException{
    dataSource().getConnection().close();
}
```

Or defined using XML

```
<bean name="dataSource"
   class="org.apache.commons.dbcp.BasicDataSource"
   destroy-method="close">
   <property name="driverClassName"  value="${db.driver}" />
   <property name="url" value="${db.url}" />
   <property name="username" value="${db.user}" />
   <property name="password" value="${db.pw}" />
   <property name="initialSize" value="2" />
   <property name="maxActive" value="5" />
   <property name="maxIdle" value="2" />
</bean>
```

JdbcDaoSupport – JDBC DAO Implementation

- Convenient base class for JDBC based DAOs
  - JDBC DAO implementation extends this class

```
@Repository
public class ReservationDAOImpl extends JdbcDaoSupport implements
ReservationDAO {
   @Autowired
   private DataSource dataSource;
   @PostConstruct
   private void initialize() {
      setDataSource(dataSource); //inherited method declared as final
   }
   public int countReservations() {
      String sql = "SELECT COUNT(*) FROM RESERVATIONS";
      return getJdbcTemplate().queryForObject(sql, Integer.class);
   }
}
```

Method/ Property	Description
jdbcTemplate	The JDBC Template Class
dataSource	DataSource (which is normally injected)
getConnection( )	returns a java.sql.Connection
getExceptionTranslator ( )	used to translate when accessing Connection directly
releaseConnection ( )	correctly closes an Obtained connection

When implementing DAO classes using JDBC, you can use the JdbcDaoSupport class as the base class for your implementation.

When extending JdbcDaoSupport you inherit the dataSource property. Therefore when the `ReservationDAO` implementation is registered it must be injected with a bean of type DataSource

As you can see, JdbcDaoSupport provides a set of properties and utility method that can be used to simplify the implementation of your DAO classes.

The jdbcTemplate

- Main helper class in Spring for using JDBC
  - Developers use only jdbcTemplate
    - (injecting the DataSource)

```
public class ReservationDAOImpl implements ReservationDAO {
   private JdbcTemplate template;
   @Autowired
   public void setDataSource(DataSource dataSource) {
      this.template = new JdbcTemplate(dataSource);
   }
   …
}
```

- Alternatively, JdbcTemplate can be injected
  - A thread-safe global singleton

```
@Bean
public JdbcTemplate jdbcTemplate() {
   return new JdbcTemplate(dataSource());
}
@Bean
public DataSource dataSource() {   …   }
```

Instead of extending JdbcDaoSupport, developers inject the DataSource into the DAO implementation class and create an instance of the JdbcTemplate themselves. Naturally it is also possible to configure the JdbcTemplate within the configuration file and inject a complete instance into you DAO implementation

Overview of `jdbcTemplate` Methods

Method	Description
`query(...)`	Send a query and handle the result in a callback.
`queryForList(...)`	Execute query for a result List
`queryForObject(...)`	Query for specified object type
`update(...)`	Send an SQL statement (allows parameters), returns update count.
`execute(...)`	Execute static SQL statement

- The `queryForObject` method expects only one record
  - Otherwise `EmptyResultDataAccessException` is thrown

```
public Integer getCustomerID(Integer reservationID) {
   String sql = "SELECT CUSTOMER_ID FROM RESERVATIONS"
              + " WHERE RESERVATION_ID=?";
   Object[] args = new Object[] { reservationID };
   return jdbcTemplate.queryForObject(sql, args, Integer.class);
}
```

- `JdbcTemplate` template relies heavily on callbacks
  - `RowMapper:` Map a row to an object
  - `RowCallbackHandler:` Process a single row

The template class defines a number of methods for executing SQL statements. The queryForObject method is intended for queries that return exactly one record. When no records or multiple records are found an exception is thrown.

The JdbcTemplate class (and other template classes provided by Spring) rely heavily on callback implementations. When the statement that was executed returns several columns from a table, the result data often needs to end up in entity instances, where each row in the result represents (at least) one entity and each column within a row represents a property value of the entity.

JdbcTemplate – RowMapper

- `RowMapper` returns object that corresponds to the row
  - Called "once per row" in `ResultSet`

```
public class ReservationMapper implements
RowMapper<RentalReservation> {
   public RentalReservation mapRow(ResultSet rs, int rowNum)
                                              throws SQLException {
      RentalReservation reservation = new RentalReservation();
      reservation.setReservationId(rs.getInt("RESERVATION_ID"));
      reservation.setCarId(rs.getString("CAR_ID"));
      …
      return reservation;
   }}
```

- The query method will return a `List` of those objects

```
final String SQL =
  "SELECT * FROM RESERVATIONS WHERE RENTAL_LOCATION = :location";
Map<String, Object> params = new HashMap<>();
params.put("location", location);
List<RentalReservation> result =
      template.query(SQL, params, new ReservationMapper());
```

- Should not call `next()` method on `ResultSet`

A RowMapper implementation can be defined to map each row in the ResultSet to a single object instance. The RowMapper implementation 'reads' the columns in the ResultSet and creates (and populates) a single instance. When the RowMapper implementation is provided to the query method of the JdbcTemplate, a List of these types will be returned.

The RowMapper should only use the ResultSet to obtain the values of the current row and should not invoke the next method !

In general, the RowMapper implementation is designed as stateless in order to make it reusable and reduce the memory requirements

JdbcTemplate – RowCallbackHandler

- `RowCallbackHandler` does not return an object
  - The `processRow` method returns void
- Result often stored in (instance) variable

```
final String SQL = "SELECT * FROM RESERVATIONS";
Map<String, List<RentalReservation>> reservations = new HashMap<>();
template.query(SQL, rs -> {
      String location = rs.getString("RENTAL_LOCATION");
      List<RentalReservation> reservations =
         reservations.computeIfAbsent(location, f -> new
ArrayList<>());
      RentalReservation reservation = new RentalReservation();
      reservation.setReservationId(rs.getInt("RESERVATION_ID"));
      …
      reservations.add(reservation);
});
```

```
public interface RowCallbackHandler {
   void processRow(ResultSet rs) throws SQLException;
}
```

A RowCallbackHandler interface is an alternative to the RowMapper. While the RowMapper returns a single instance for each row, the processRow method of the RowCallbackHandler returns void. The handler often relies on surrounding variables to ‘calculate’ a value based upon the records returned by the query.

JdbcTemplate – Defining Queries

- SQL statement must be supplied
  - As a `String`

```
String sql = "SELECT COUNT(*) FROM RESERVATIONS";
return template.queryForObject(sql, new HashMap<>(), Integer.class);
```

  - As a `PreparedStatementCreator`

```
PreparedStatementCreator creator = new PreparedStatementCreator() {
   public PreparedStatement createPreparedStatement(Connection con)
                                                    throws SQLException {
      PreparedStatement stmt = con.prepareStatement(
                "SELECT * FROM RESERVATIONS WHERE RENTAL_LOCATION =?");
      stmt.setString(1, location);
      return stmt;
   }
};
template.query(creator, rowMapper);
```

SQL statements most often contain placeholders that must be bound to specific values before being executed. The SQL statement that is about to be executed by the template may contain zero or more placeholders (question marks). The statement can be provided to the template as a ‘simple’ String or a PreparedStatementCreator. When using the PreparedStatementCreator, the values can be bound to the placeholders by implementing the `createPreparedStatement method`.

JdbcTemplate – Query Parameters

- Parameter values are supplied
  - as `Object[]`

```
String sql = "SELECT * FROM RESERVATIONS WHERE RENTAL_LOCATION=?";
Object[] params = { location };
return template.query(sql, params, rowMapper);
```

  - Using a `PreparedStatementSetter`

```
String sql = "SELECT * FROM RESERVATIONS WHERE RENTAL_LOCATION=?";
PreparedStatementSetter stmtSetter = new PreparedStatementSetter() {
   public void setValues(PreparedStatement ps) throws SQLException {
      ps.setString(1, location);
   }
};
return template.query(sql, stmtSetter, rowMapper);
```

When the statement has been provided as a simple String, the values of these placeholders must be provided separately. One way to do so is suppling an Object[] as second parameter of the query method. Instead an instance of `PreparedStatementSetter can be supplied to the query method.`

Batch Operations

- Grouping updates into batches limits number of round trips to database
  - Making multiple calls to same `PreparedStatement`
- `BatchPreparedStatementSetter` can be passed to `batchUpdate` method

```
String sql = "UPDATE RESERVATIONS SET STATUS = ? WHERE RESERVATION_ID=?";
BatchPreparedStatementSetter setter = …
template.batchUpdate(sql, setter);
```

  - `getBatchSize` method returns number of times `setValues` method is invoked
    - ...and query is executed

```
public interface BatchPreparedStatementSetter {
   void setValues(PreparedStatement ps, int i)
                                    throws SQLException;
   int getBatchSize();
}
```

Most current JDBC drivers provide improved performance when multiple calls to the same prepared statement are grouped into a single batch. Grouping multiple calls into a single batch limits the number of round trips that need to be made to the database.

The JdbcTemplate contains a batchUpdate method, which (besides the SQL statement to be executed) accepts a reference to a `BatchPreparedStatementSetter`. `Implementations of this interface need to provide an implementation of two methods. The getBatchSize method returns the number of 'updates' that need to take place. The number returned by this method is the number of times the setValues method is called and the update is performed.`

Batch Operations Example

- Query parameters can be provided
  - Using `BatchPreparedStatementSetter`

```
public void closeReservations(List<RentalReservation> reservations) {
  String sql = "UPDATE RESERVATIONS SET STATUS = ? WHERE RESERVATION_ID=?";
  BatchPreparedStatementSetter setter = new BatchPreparedStatementSetter()  {
    public void setValues(PreparedStatement ps, int i) throws SQLException {
      RentalReservation reservation = reservations.get(i);
      ps.setString(1, ReservationStatus.CLOSED.toString());
      ps.setInt(2, reservation.getReservationId());
    }
    public int getBatchSize() {
      return reservations.size();
    }
  };
  template.batchUpdate(sql, setter);
}
```

  - Using `List` of `Object[]` types

```
List<Object[]> parameters = …
template.batchUpdate(sql, parameters);
```

The code example shown above is an example of a batch-update method using `BatchPreparedStatementSetter interface.`

`An overloaded implementation of the batchUpdate method accepts (in addition to the SQL statement) a List of Object[] types. Each entry in the List holds the values that are to be bound to the placeholders in the query. The size of the list equals the amount of times the statements is about to be executed.`

NamedParameterJdbcTemplate

- Adds support for named parameters

```
String sql = "SELECT COUNT(*) FROM RESERVATIONS "
           + "WHERE RENTAL_LOCATION = :location";
```

  - Parameters are provided using Map-based implementation

```
Map<String, Object> params = new HashMap<>();
params.put("location", location);
Integer count =  template.queryForObject(sql, params, Integer.class);
```

- SqlParameterSource represents parameter/value pairs
  - MapSqlParameterSource provides named parameters using 'simple' Map implementation

```
MapSqlParameterSource params = new MapSqlParameterSource();
params.addValue("location", location);
Integer count = template.queryForObject(sql, params, Integer.class);
```

Normally you would use question marks within the SQL statement when using a PreparedStatement within JDBC. If you have ever done this, most likely, you ended up counting questions marks, to determine which parameter value had to be assigned to which parameter. NamedParameterJdbcTemplate allows you to replace these question marks by named parameters. Do notice that the named parameter value must start with a colon (:)

MapSqlParameterSource is based on a simple java.util.Map implementation, the key of the map must be the parameter name (without the colon).

BeanPropertySqlParameterSource

- Wraps JavaBean
  - Property names become parameter names

```
public void addReservation(RentalReservation reservation) {
   String sql = "INSERT INTO RESERVATIONS VALUES(:reservationId, "
     + " :customerId, :carId, :rentalLocation, :rentalDate,"
     + " :returnLocation, :returnDate)";
   BeanPropertySqlParameterSource params =
           new BeanPropertySqlParameterSource(reservation);
   template.update(sql, params);
}
```

```
public class RentalReservation {
   private int reservationId;
   private int customerId;
   private String carId;
   private String rentalLocation;
   private Date rentalDate;
   private String returnLocation;
   private Date returnDate;
   private ReservationStatus status
   //getter and setter methods
}
```

- Parameter names must match property names of JavaBean
- Enum type must be explicitly registered

```
params.registerSqlType("status", Types.VARCHAR);
```

BeanPropertySqlParameterSource can be used to wrap any JavaBean (class which follows JavaBean naming conventions). This implementation will use the property names of the JavaBean as parameter names.

Exception handling in JDBC applications

- `SQLException` is only exception for all database errors
  - `errorCode` is vendor specific
    - (hence cannot be used in database independent code)
  - `sqlState` is supposed to be portable as it is based on X/OPEN SQL spec or SQL99 convention
  - None of these provide easy abstraction or portability

```
String sql = "INSERT INTO RESERVATIONS VALUES(?,?,?,?,?,?,?)";
try (Connection connection = dataSource.getConnection();
     PreparedStatement stmt = connection.prepareStatement(sql);) {
      …
   stmt.executeUpdate();
} catch (SQLException sqle) {
   //Exception handling here
}
```

```
Caused by: ERROR 23505: The statement was aborted because it
would have caused a duplicate key value in a unique or
primary key constraint
```

Apache Derby

The JDBC API uses instances of SQLException for all database errors. To obtain information about the actual error (e.g. the error is a result of a primary key violation) you would have to obtain the vendor specific error code from the exception and translating this error code into a database independent error message

Exception Translation

- Spring uses vendor specific SQL code to exception translation
  - Standard translation described in `sql-error-codes.xml` XML resource

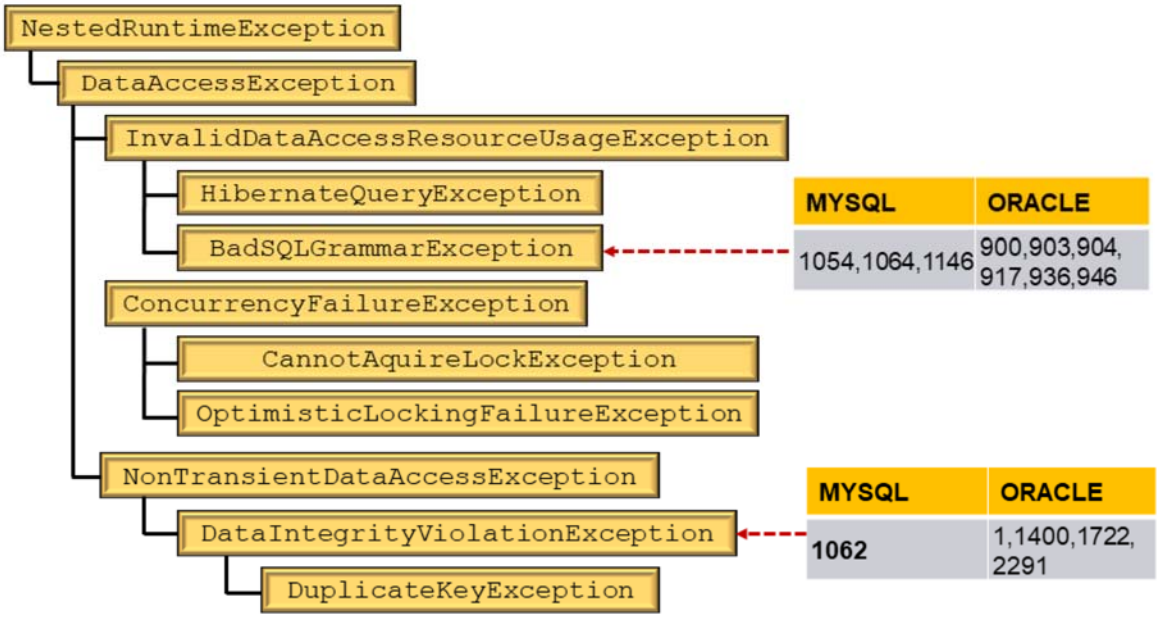
```
<bean id="Derby" class="org.springframework.jdbc.support.SQLErrorCodes">
  <property name="databaseProductName">
    <value>Apache Derby</value>
  </property>
  <property name="useSqlStateForTranslation">
    <value>true</value>
  </property>
  <property name="badSqlGrammarCodes">
    <value>42802,42821,42X01,42X02,42X03,42X04,42X05,42X06,42X07,42X08</value>
  </property>
  <property name="duplicateKeyCodes">
    <value>23505</value>
  </property>
  <property name="dataIntegrityViolationCodes">
    <value>22001,22005,23502,23503,23513,X0Y32</value>
  </property>
  ...
  </property>
</bean>
```

The Exception translator of Spring uses the SQL specific codes defined in the `sql-error-codes.xml` file. A snippet from this file is shown above.

The sql-error-codes.xml file is located in the org.springframework.jdbc.support package

Exception Handling

- Exception mapping is independent of JDBC



Spring Exception handling uses a translation table to interpret the error code (depending on the database used) and translate this error code into a vendor independent error message automatically

Operation Classes

- Spring RDBMS operation classes
  - Alternative approach to direct JDBC or `JdbcTemplate`
  - Provides a higher level of abstraction: SQL is encapsulated in *operation classes*
  - Provides a more object oriented approach to encapsulating JDBC access
- Encapsulation is placed in different classes extending:

Operation class	Description
`SqlQuery`	Base class for MappingSqlQuery
`MappingSqlQuery`	Used to encapsulate a SELECT
`SqlUpdate`	Used to encapsulate the other DML
`StoredProcedure`	Encapsulates a stored procedure call
`SqlFunction`	Wraps a single select statement

Operation classes provide an alternative approach to JDBC and the JdbcTemplate. Operation classes are regular JavaBeans that completely encapsulate the SQL operation and provide another layer of abstraction when implementing data access layer.

Operation Classes (cont'd)

- Operation class extends `MappingSqlQuery`
  - Generic type define return type

```
public class ReservationQuery extends MappingSqlQuery<RentalReservation> {
   private static final String SQL =
         "SELECT * FROM RESERVATIONS WHERE RENTAL_LOCATION=?";
   public ReservationQuery(DataSource dataSource) {
      super(dataSource, SQL);
      declareParameter(new SqlParameter("locationID", Types.VARCHAR));
      compile();
   }
   protected RentalReservation mapRow(ResultSet rs, int rowNum)
                                                   throws SQLException {
      RentalReservation reservation = new RentalReservation();
      reservation.setReservationId(rs.getInt("RESERVATION_ID"));
      return reservation;
   }
}
```

- SQL will be used to create `PreparedStatement`
  - Each parameter must be declared
  - The `compile()` method compiles and validates query

The SQL statement that is sent to the constructor of the super class will be used to create a PreparedStatement instance. This means that the SQL statement may contain placeholders for any parameters that are being passed in during execution. Each parameter defined in the query must be explicitly declared by invoking the declareParameter that is inherited from the super class. An instance of SqlParameter must be provided, each containing a name and the JDBC type as defined in the java.sql.Types class. Once all parameters have been defined, the compile() method must be called to have the statement prepared for execution.

Implementation of the class is thread-safe after it has been compiled.

Using the Operation Classes

- The `execute` method returns one or more 'rows'
  - Accepts zero or more parameter values

```
@Repository
public class ReservationDAOImpl implements ReservationDAO {
   @Autowired
   private DataSource dataSource;
   public List<RentalReservation> getReservationsByLocation(
                                              String location) {
      ReservationQuery query = new ReservationQuery(dataSource);
      return query.execute(location);
   }
   …
}
```

- When operation query only returns single instance `findObject` can be used

```
public RentalReservation getReservationByID(Integer reservationID)
{
   ReservationQuery query = new ReservationQuery(dataSource);
   return query.findObject(reservationID);
}
```

The execute method of the `MappingSqlQuery` implementation returns zero or more instances. Overloaded execute methods allow for the definition of multiple parameter values (when the statement to be executed contains multiple placeholders)

When the statement defined in the operation class only returns a single row the findObject method can be used instead of the execute method.

Update Operation Classes

- `SqlUpdate` encapsulates SQL update
  - Defines several `update` methods

```
public class UpdateReservationQuery extends SqlUpdate {
   private static final String SQL =
     "UPDATE RESERVATIONS SET STATUS=? WHERE RESERVATION_ID=?";

   public UpdateReservationQuery(DataSource dataSource) {
      super(dataSource, SQL);
      declareParameter(new SqlParameter("status", Types.VARCHAR));
      declareParameter(new SqlParameter("id", Types.NUMERIC));
      compile();
   }
   public int execute(int reservationID, ReservationStatus status) {
      return update(status, reservationID);
   }
}
```

SqlUpdate encapsulates an update statement. Instead of returning a set of data it updates zero or more rows in the data source. So instead of defining execute methods, it contains (overloaded) update methods.

Embedded Databases

- Spring provides build-in support for embedded databases
  - HSQL, H2, and Derby are supported out of the box
- Useful during development
  - Configuration is simple
  - Limited startup time of database
  - Easy to update DDL during development
- Embedded database is defined as a `javax.sql.DataSource`
- Best practice to provide unique name to each `DataSource`

Spring also provides support for embedded (in-memory) database. Out of the box, Spring comes with support for embedded HSQL, H2 and Derby database.

Due to their simple configuration and quick startup time, embedded databases can be extremely useful during development time. The Embedded database that is being configured is represented in context as an instance of javax.sql.DataSource and can easily be replaced by a production datasource at a later stage.

Even when using embedded databases it is common practice to provide a unique name to each DataSource

Programmatically defining database

- `EmbeddedDatabaseBuilder` is used to construct database
  - Provides a fluent API for defining properties
  - Allows execution of SQL scripts

```
@Bean
public DataSource dataSource() {
    return new EmbeddedDatabaseBuilder()
                .generateUniqueName(true)
                .setType(H2)
                .setScriptEncoding("UTF-8")
                .ignoreFailedDrops(true)
                .addScript("schema.sql")
                .addScripts("data.sql")
                .build();
}
```

The `EmbeddedDatabaseBuilder` class defines a fluent API for the definition of the embedded database, besides setting the type of the database, the builder class also accepts references to SQL scripts that will be executed during the construction of the database, allowing for tables to be created and data to be inserted into these tables at startup of the application.

Exercise 7: Using Spring JDBC

~/StudentWork/code/spring-jdbc/lab-code

Please refer to the written lab exercise and follow the directions as provided by your instructor

Session: Spring Data (Introduction)

Spring Data Overview
Spring Data Query Methods