
StayAhead Training
the training specialist
bringing people and technology together
Oracle Database
Oracle Development
Oracle Fusion Middleware
Oracle Business Intelligence
Oracle e-Business
MySQL
MariaDB
MongoDB
Unit Testing Frameworks
Maven
Hibernate
Acceptance Testing Frameworks
Java Technology
Web Development HTML & CSS
JavaScript & jQuery
PHP, Python Perl & Ruby
OOAD
Spring
UNIX
Solaris, AIX & HP-UX
Red Hat Linux
SUSE Linux
LPI Linux
Business Analysis ITIL® & PRINCE2®
London | Birmingham | Leeds | Manchester | Sunderland | Bristol | Edinburgh
scheduled | closed | virtual training
www.stayahead.com
+44 (0) 20 7600 6116
sales@stayahead.com

Spring 4.x with REST

JP Morgan

Although StayAhead Training Limited makes every effort to ensure that the information in this manual is correct, it does not accept any liability for inaccuracy or omission.

StayAhead Training does not accept any liability for any damages, or consequential damages, resulting from any information provided within this manual.

Spring 4.x with REST

Duration: 4 days

Spring 4.x with REST Course Overview

The Spring Framework is the leading Java EE solution for enterprise software.

The course comprises sessions dealing with Dependency Injection (DI); Aspect Oriented Programming, (AOP); Spring Web including MVC, REST, and WebSocket; Spring Data including JDBC, ORM, and Transaction Management; Spring Security; and Spring Boot. It also includes an introduction to Spring Cloud.

Exercises and examples are used throughout the course to give practical hands-on experience with the techniques covered.

Skills Gained

The delegate will practice:

- Obtaining Spring Beans using DI
- Implementing cross-cutting concerns using AOP
- Building a Spring MVC web app
- Building a REST API with Spring MVC
- Building a WebSocket service
- Using a JDBC Template
- Integrating a Spring app with Hibernate and JPA
- Transaction Management
- User authentication
- URL and method-level authorization
- Building a simple Spring Boot app

Who will the Course Benefit?

Spring 4.x with REST course will benefit Java developers who are new to Spring and want to develop large and complex enterprise-level applications.

Course Objectives

This course aims to provide the delegate with the knowledge to be able to develop (or contribute to the development of) a Spring web app which exposes a RESTful API and that both reads and writes data to/from persistent storage. The web app should be secure insofar as it authenticates users and restricts access. The delegate will also be aware of Spring Boot and Spring Cloud.

Requirements

Delegates should be competent Java developers able to produce Java applications that exploit the core elements of the language including variables, expressions, selection, iteration, arrays, methods, classes, objects, and exception handling. This knowledge can be gained by attendance on the Core Java course.

Pre-Requisite Courses

- Core Java

Notes

- Course technical content is subject to change without notice.
- Course content is structured as sessions, this does not strictly map to course timings. Concepts, content and practicals often span sessions.

Table of Contents

Chapter 1: Introduction to Spring

Chapter 2: Dependency Injection

Chapter 2: Dependency Injection (continued)

Chapter 3: Spring AOP

Chapter 4: Spring WebMVC

Chapter 5: REST

Chapter 6: WebSocket

Chapter 7: JDBC

Chapter 8: Spring & Hibernate ORM

Chapter 8: Spring & Hibernate ORM (continued)

Chapter 9: Spring Transaction Management

Chapter 10: Spring Security

Chapter 10: Spring Security (continued)

Chapter 11: Spring Boot

Chapter 12: Introduction to Spring Cloud

Chapter 12: Introduction to Spring Cloud (continued)

CHAPTER 1

Introduction to Spring

Introduction

So, what is the Spring Framework?

Light-weight yet comprehensive framework for building Java applications:

- Web applications
- Enterprise applications
- Standalone applications
- Batch application
- Integration application

Spring is an open-source framework, created by Rod Johnson, created to address the complexity of enterprise application development. Spring makes it possible to use plain-vanilla JavaBeans to achieve things that were previously only possible with old EJBs. It is also useful beyond the server side. It provides application benefit in terms of simplicity, testability, and loose coupling.

"Spring is a lightweight inversion of control and aspect-oriented container framework"

Lightweight

Spring is lightweight in terms of both size and overhead. Spring is nonintrusive: objects in a Spring-enabled application typically have no dependencies on Spring specific classes.

It also doesn't take over or dictate, and can be used alongside existing code where required.

Inversion of control

Loose coupling is promoted through inversion of control (IoC), where objects are given their dependencies instead of creating or looking themselves.

Aspect-oriented

Spring supports aspect-oriented programming which separates application business logic from system services (such as auditing and transaction management).

Container

Spring is a container in the sense that it contains and manages the life cycle and configuration of application objects. Bean are configurable based on a configurable prototype. This is light, unlike old EJB containers.

Framework

Application objects are composed declaratively, typically in an XML file or by annotations. Spring also provides much infrastructure functionality like transactions, persistence etc, leaving the development of application logic to the developer.

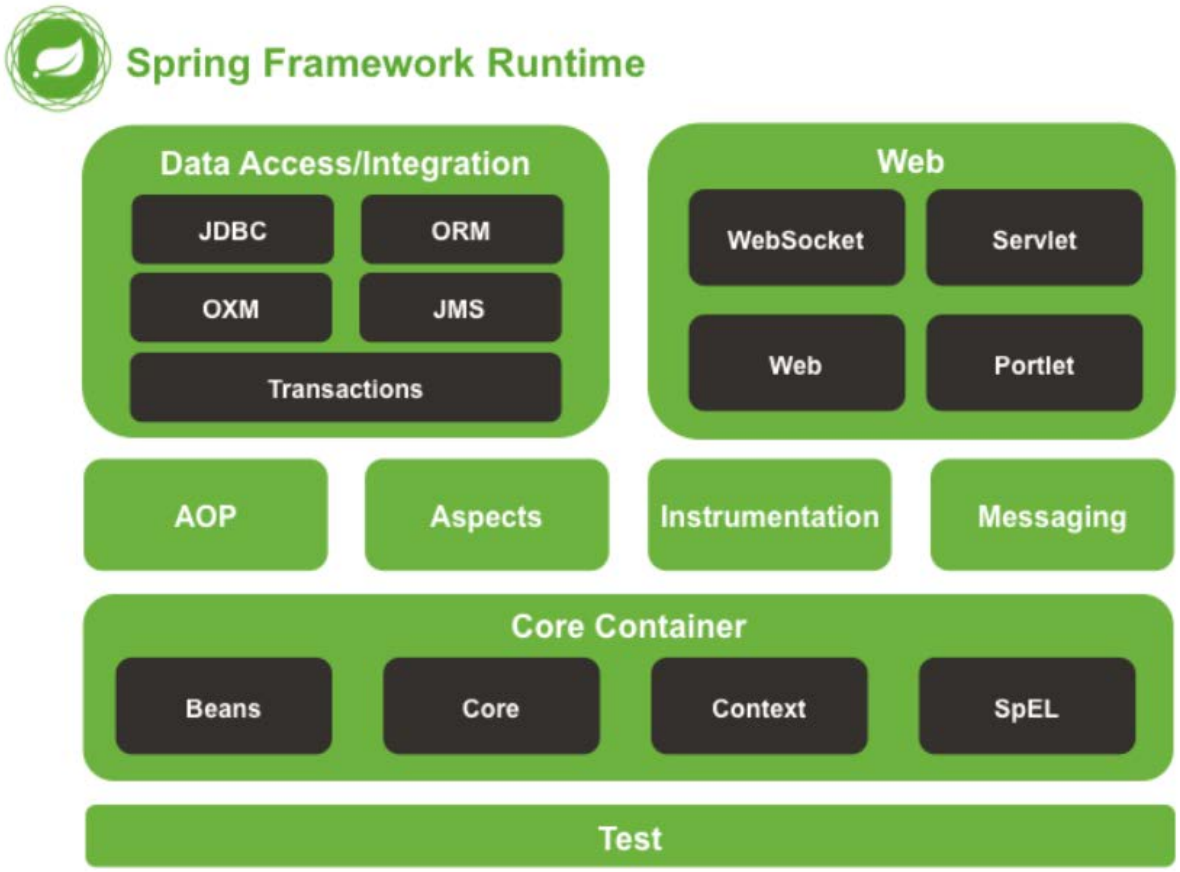
This section will look at each of the major Spring modules, providing an introduction to each.

The Spring Framework

As the name implies, this is the core of the Spring Framework, providing support for dependency injection, transaction management, web applications, data access, messaging, testing, and much more.

- Dependency Injection
- Aspect-Oriented Programming
- Spring MVC web application
- RESTful web services
- Foundational support for JDBC, JPA, JMS

The Spring Framework consists of about 20 modules, grouped into Core Container, Data Access/Integration, Web, AOP (Aspect Oriented Programming), Instrumentation, Messaging, and Test:



Note Spring 5 replaces Portlet with WebFlux.

The Spring Framework is a Java platform that provides comprehensive infrastructure support for developing Java applications.

Spring enables applications to be built from POJOs and adds enterprise services non-invasively to those POJOs.

Spring's Core Principles

Much of the ability of Spring comes about as a result of the use of Dependency Injection (DI) and Inversion of Control (IoC).

A Java application will typically consist of collaborating objects; objects that dependencies on each other. The question of how to organise these objects into a coherent whole is a difficult one, often the domain of architects and developers.

Patterns can help in this organisation; Factory, Abstract Factory, Builder, Decorator, and Service Locator help to compose classes and instances, these patterns essentially being best practices that should be followed.

Spring Framework IoC addresses this concern by providing a formalized means of composing disparate components. Design patterns are formalised as first-class objects to be readily integrated into application(s).

The name "Inversion of Control" was renamed for simplicity by Martin Fowler as "Dependency Injection":

http://martinfowler.com/articles/injection.html

Service assembly using DI has since become the norm; an alternative to the Service Locator pattern. The simplicity of the SL pattern does not render it useless, but where using classes in multiple applications, DI is favoured.

Martin advocated the use of constructor over setter injection; also advocating the use of a container that provides support for both. The Spring container has just that capability.

Core Container

The Core Container consists of the following modules:

- spring-core
- spring-beans

The fundamentals of the framework, including the IoC and Dependency Injection features. The BeanFactory is a sophisticated implementation of the factory pattern. It removes the need for programmatic singletons and allows the decoupling of configuration and specification of dependencies from program logic.

- spring-context
- spring-context-support

The core container extends the core and beans modules, providing a means to access objects in a manner similar to JNDI. It also adds internationalization, event propagation, resource loading, and the transparent creation of contexts by, for example, a Servlet container. Also supports EJB, JMX, and basic remoting.

Support is provided by common third-party libraries for caching (EhCache, Guava, JCache), mailing (JavaMail), scheduling (CommonJ, Quartz) and template engines (FreeMarker, JasperReports, Velocity).

- spring-context-indexer (from Spring 5)

Improves startup performance of large applications by creating a static list of candidates at compilation time. Just needs to be present/configured via either Maven or Gradle.

- spring-expression

This provides an Expression Language for querying and manipulating an object graph at runtime. An extension of the unified expression language (unified EL) as specified in the JSP 2.1 specification.

AOP & Instrumentation

- spring-aop

The Spring AOP module provides an AOP Alliance-compliant aspect-oriented programming implementation allowing you to define, for example, method interceptors and pointcuts to cleanly decouple code that implements functionality that should be separated. Using source-level metadata functionality, you can also incorporate behavioural information into your code, in a manner similar to that of .NET attributes.

- spring-aspects

This provides integration with AspectJ.

- spring-instrument

Provides class instrumentation support and classloader implementations to be used in certain application servers.

- spring-instrument-tomcat (not available in Spring 5)

Contains Spring's instrumentation agent for Tomcat 6 or 7. Not intended to work with Tomcat 8+.

Messaging

- spring-messaging

This allows key abstractions from the Spring Integration project such as Message, MessageChannel, MessageHandler, and others to serve as a foundation for messaging-based applications. The module also includes a set of annotations for mapping messages to methods, similar to the Spring MVC annotation based programming model.

Data Access/Integration

This area of functionality consists of the JDBC, ORM, OXM, JMS, and Transaction modules.

- spring-jdbc

This provides a JDBC-abstraction layer that removes the need to do tedious JDBC coding and parsing of database-vendor specific error codes.

- spring-tx

Supports programmatic and declarative transaction management for classes that implement special interfaces and for all your POJOs (Plain Old Java Objects).

- spring-orm

Provides integration layers for popular object-relational mapping APIs, including JPA, JDO, and Hibernate. Using the spring-orm module you can use all of these O/R-mapping frameworks in combination with all of the other features Spring offers, such as the simple declarative transaction management feature mentioned previously.

- spring-oxm

Provides an abstraction layer that supports Object/XML mapping implementations such as JAXB, Castor, XMLBeans, JiBX and XStream.

- spring-jms

Contains features for producing and consuming messages. Since Spring Framework 4.1, it provides integration with the spring-messaging module.

Web

- spring-web

Provides basic web-oriented integration features such as multipart file upload functionality and the initialization of the IoC container using Servlet listeners and a web-oriented application context. It also contains an HTTP client and the web-related parts of Spring's remoting support.

- spring-webmvc

Also known as the Web-Servlet module, contains Spring's model-view-controller (MVC) and REST Web Services implementation for web applications. Spring's MVC framework provides a clean separation between domain model code and web forms and integrates with all of the other features of the Spring Framework.

- spring-webmvc-portlet (no longer available in Spring 5)

Also known as the Web-Portlet module) provides the MVC implementation to be used in a Portlet environment and mirrors the functionality of the spring-webmvc module.

- spring-webflux (from Spring 5)

A fully non-blocking web framework supporting Reactive Streams back pressure, and runing on servers such as Netty, Undertow, and Servlet 3.1+ containers.

- spring-websocket (from Spring 5)

WebSocket and SockJS infrastructure, including STOMP messaging support.

Test

- spring-test

Supports the unit testing and integration testing of Spring components with JUnit or TestNG. It provides consistent loading of Spring ApplicationContexts and caching of those contexts. It also provides mock objects that you can use to test your code in isolation.

Obtaining Spring

Maven Central contains many of the common libraries that Spring depends on and a large section of the Spring community uses Maven for dependency management. The names of the jars here are in the form spring-*-<version>.jar and the Maven groupId is org.springframework.

ArtifactId	Description
spring-aop	Proxy-based AOP support
spring-aspects	AspectJ based aspects
spring-beans	Beans support, including Groovy
spring-context	Application context runtime
spring-context-indexer	Static compilation helper (5+)
spring-context-support	Support classes
spring-core	Core utilities
spring-expression	Spring EL support
spring-instrument	Agent for JVM bootstrapping
spring-instrument-tomcat	Instrumentation agent for Tomcat
spring-jdbc	JDBC support package
spring-jms	JMS support package
spring-messaging	Messaging architectures/protocols
spring-orm	O/R Mapping: JPA and Hibernate
spring-oxm	Object/XML Mapping
spring-test	Unit testing and integration testing
spring-tx	Transaction infrastructure
spring-web	Web support packages
spring-webflux	Reactive support (5+)
spring-webmvc	REST Web Services and MVC
spring-webmvc-portlet	MVC Implementation for Portlets (not in 5)
spring-websocket	WebSocket and SockJS

Maven Dependencies

Spring intentionally keeps its mandatory dependencies to a minimum: simple use cases should not require too much pain. Simple dependency injection requires just one mandatory external dependency for logging.

To create an application context and use dependency injection to configure an application, the dependencies entry is simply (against Maven Central):

```
<dependencies>
    <dependency>
        <groupId>org.springframework</groupId>
        <artifactId>spring-context</artifactId>
        <version>5.X.X.RELEASE</version>
        <scope>runtime</scope>
    </dependency>
</dependencies>
```

The scope can be declared as runtime if you don't need to compile against Spring APIs; usually the case for basic dependency injection use cases.

The alternative is to use the Spring Maven repository for milestones and snapshots:

For full releases:

```
<repositories>
    <repository>
        <id>io.spring.repo.maven.release</id>
        <url>http://repo.spring.io/release/</url>
        <snapshots><enabled>false</enabled></snapshots>
    </repository>
</repositories>
```

For milestones:

```
<repositories>
    <repository>
        <id>io.spring.repo.maven.milestone</id>
        <url>http://repo.spring.io/milestone/</url>
        <snapshots><enabled>false</enabled></snapshots>
    </repository>
</repositories>
```

And for snapshots:

```
<repositories>
    <repository>
        <id>io.spring.repo.maven.snapshot</id>
        <url>http://repo.spring.io/snapshot/</url>
        <snapshots><enabled>true</enabled></snapshots>
    </repository>
</repositories>
```

Mixing versions of Spring-provided JARs can lead to some rather difficult to debug situations. It may be that another library, or Spring project, uses a transitive dependency to an older release.

Spring BOM

Maven supports the concept of a "bill of materials" (BOM) dependency. Importing the spring-framework-bom ensures that all spring dependencies (both direct and transitive) are at the same version. This can be extremely useful.

```
<dependencyManagement>
    <dependencies>
        <dependency>
            <groupId>org.springframework</groupId>
            <artifactId>spring-framework-bom</artifactId>
            <version>5.X.X.RELEASE</version>
            <type>pom</type>
            <scope>import</scope>
        </dependency>
    </dependencies>
</dependencyManagement>
```

Using the BOM means it is not required to specify the <version> attribute when depending on Spring Framework artifacts:

```
<dependencies>
    <dependency>
        <groupId>org.springframework</groupId>
        <artifactId>spring-context</artifactId>
    </dependency>
    <dependency>
        <groupId>org.springframework</groupId>
        <artifactId>spring-web</artifactId>
  </dependency>
</dependencies>
```


Spring Zipped

It is still possible to download a distribution zip file, which are published to the Spring Maven Repository.

To download a distribution zip:

http://repo.spring.io/release/org/springframework/spring

, select the subfolder for the chosen version.

Distribution files end -dist.zip.

Distributions are also published for milestones and snapshots.

Spring Logging

Having unified logging configured centrally for the whole application is the aim, which means choosing a logging framework.

Mandatory logging dependency in Spring is the Jakarta Commons Logging API (JCL). This uses a runtime discovery algorithm that looks for other logging frameworks in well known places on the classpath and uses one that it thinks is appropriate, or can be told. If nothing else is available you get pretty nice looking logs just from the JDK (java.util.logging or JUL for short). Spring should log out of the box in most situations.

Other Logging Choices

Switch off commons-logging by either excluding the dependency from the spring-core module, or depend on a commons-logging dependency that replaces the library with an empty jar.

To exclude commons-logging:

```
<dependencies>
    <dependency>
        <groupId>org.springframework</groupId>
        <artifactId>spring-core</artifactId>
        <version>4.1.6.RELEASE</version>
        <exclusions>
            <exclusion>
                <groupId>commons-logging</groupId>
                <artifactId>commons-logging</artifactId>
            </exclusion>
        </exclusions>
    </dependency>
</dependencies>
```

, and provide a new logging alternative.

SLF4J

SLF4J provides bindings to many common logging frameworks, and can bridge to and from those frameworks. A common choice is bridge Spring to SLF4J, and then provide explicit binding from SLF4J to Log4J. A more common choice is to bind directly to Logback.

Log4J

An extremely well-used library, Spring provides utilities for configuring and initializing Log4j, so it has an optional compile-time dependency on Log4j in some modules.

To make Log4j work with the default JCL dependency (commons-logging) all you need to do is put Log4j on the classpath, and provide it with a configuration file (log4j.properties or log4j.xml in the root of the classpath). So for Maven users this is your dependency declaration:

<dependencies>

```
        <dependency>
            <groupId>org.springframework</groupId>
            <artifactId>spring-core</artifactId>
            <version>4.1.6.RELEASE</version>
        </dependency>
        <dependency>
            <groupId>log4j</groupId>
            <artifactId>log4j</artifactId>
            <version>1.2.14</version>
        </dependency>
    </dependencies>
```

And a sample log4j.properties:

```
log4j.rootCategory=INFO, stdout

log4j.appender.stdout=org.apache.log4j.ConsoleAppender
log4j.appender.stdout.layout=org.apache.log4j.PatternLayout
log4j.appender.stdout.layout.ConversionPattern=%d{ABSOLUTE}
%5p %t %c{2}:%L - %m%n
log4j.category.org.springframework.beans.factory=DEBUG
```

Native JCL

When running Spring applications in a container that provides an implementation of JCL, such as IBM Websphere Application Server (WAS), causes problems. In this case, invert the class loader hierarchy (the "parent last") allowing the application to control the JCL dependency rather than the container.

New in Spring Framework

Spring 5.X is the latest major release with full support for Java 8 and 9. Spring may still be used against older versions of Java, the minimum now being Java SE 6. Many deprecated classes and methods have been removed. A migration guide for those moving from Spring 3 is available at:

https://github.com/spring-projects/spring-framework/wiki

Spring 5.X

The manual and course are concerned with version 5.X, though it remains pertinent to see where improvements have been made, and how Spring has evolved, and so some elements of Spring 4.X remain.

- JDK 8+9 and Java EE 7 baseline
- Removed support for technologies such as Portlet, Velocity, JasperReports, XMLBeans, JDO, Guava
- JDK 9 preparations in core container
- General web improvements, including Servlet 3.1 support and support for Protobuf 3.0
- Reactive programming model support in various modules, and spring-webflux
- Testing improvements, including support for JUnit 5
- Spring Boot 2.x inline with Spring 5
- Spring Cloud additions

Learning Spring

Learning Spring has previously been regarded as laborious, but this has been alleviated with the updated website:

https://spring.io/

There are a number of guides and starter projects that are invaluable as introductions to the various modules. Full details of new features in the subsequent minor releases can be found in the documentation.

Beyond the Spring Framework

The Spring landscape is vast, dwarfing JEE. Beyond the core Spring Framework are libraries and projects for all manner of operations. The main ones are listed here:

Spring Batch

Provides reusable functions that are essential in processing large volumes of records, including logging/tracing, transaction management, job processing statistics, job restart, skip, and resource management.

Spring Integration

Enables lightweight messaging within Spring-based applications and supports integration with external systems via declarative adapters.

Spring Security

A highly customizable authentication and access-control framework. The standard for securing Spring-based applications.

Spring Web Services

Facilitates contract-first SOAP service development, allowing for the creation of flexible web services using the many ways to manipulate XML payloads.

Spring Boot

Enables the creation of stand-alone, production-grade Spring based Applications that can be run as applications, getting things started with minimum fuss.

Spring HATEOAS

Provides APIs for REST representations that follow the HATEOAS principle. Addresses link creation and representation assembly.

There are many other projects, of which may are unobtrusive, meaning they can be used when required as part of application development.

Spring Cloud

Tools for developers to quickly build some of the common patterns in distributed systems (e.g. configuration management, service discovery, circuit breakers, intelligent routing, micro-proxy, control bus, one-time tokens, global locks, leadership election, distributed sessions, cluster state).

CHAPTER 2

Dependency Injection

Introduction

It is important to fully appreciate the core of the Spring Framework before examining individual projects.

Spring is based on an Inversion of Control (IoC) container alongside Aspect-Oriented Programming (AOP) technologies, with additional integration with AspectJ.

This section will address with IoC and Dependency Injection to provide the necessary grounding for learning Spring technologies.

Inversion of Control

Inversion of control is difficult to explain without solid code examples, but with the help of Wikipedia we can get some way there ...

With inversion of control, the flow of business logic in an application is determined by the object graph built up during execution, made possible by object interactions being defined through abstractions. Run-time binding is achieved by mechanisms such as dependency injection or a service locator.

IoC can still statically link during compilation, but through reading description from external configuration instead of direct referencing the code.

In dependency injection, a dependent object or module is coupled to the object it needs at run time. Which particular object will satisfy the dependency during program execution typically cannot be known at compile time using static analysis.

In order for runtime binding to work, objects must possess compatible interfaces. Classes may delegate behaviour to interfaces which are implemented by other classes; the program instantiates both classes, and then injects one into the other.

Core Dependency Injection

The core principle of dependency injection is to separate behaviour from dependency resolution.

Dependency injection involves four elements:

1. The implementation of a service object
2. The client object depending on the service
3. The interface the client uses to communicate with the service
4. The injector object, which injects the service into the client.

The injector object may also be referred to as an assembler, provider, container, factory, or spring.

Implementation of dependency injection is very similar to the strategy pattern, but while the strategy pattern is intended for dependencies to be interchangeable throughout an object's lifetime, in dependency injection only a single instance of a dependency is used.

Dependency injection is the most common example of inversion of control. The effect is to take code like this:

```
public class FileReader { }

public class Application {
    // Inflexible concrete dependency.
    private FileReader fileReader = new FileReader();
}
```

, and make it far more dislocated:

```
public interface Reader { }

public class FileReader implements Reader { }

public class Application {

   // Inject flexible abstract dependency.
      private Reader reader;

   // Injection via constructor.
   public Application(Reader reader) {
            this.reader = reader;
      }

   // Injection via setter.
   public void setReader(Reader reader) {
            this.reader = reader;
      }
}
```

Dislocating further, and application can be created with Consumers:

```
public interface Consumer {
   void work(String foo);
}
```

, where the application can delegate calls to the injected implementation:

```
public class Application implements Consumer {

   private Service service;

   public Application() { }

   // Constructor injection.
   public Application(Service service){
      this.service = service;
   }

   // Setter injection.
   public void setService(Service service) {
      this.service = service;
   }

   // Delegate.
   @Override
   public void work(String foo) {
      // Do some validation, logic here.
      service.work(foo);
   }
}
```

The injected service is itself an implementation of a Service interface:

```
public interface Service {

   void work(foo);
}
```

, such as:

```
public class FooServiceImpl implements Service {

   @Override
   public void work(String foo) {
      System.out.println("Foo: " + foo);
   }
}
```


The service is injected by an implementation of an injector at runtime:

```
public interface ServiceInjector {

   public Consumer getConsumer();
}
```

, for instance:

```
public class FooServiceInjector implements ServiceInjector {

   @Override
   public Consumer getConsumer() {
      return new Application(new FooServiceImpl());
   }
}
```

, so that, at runtime, services can be injected and consumed as required:

```
public static void main(String[] args) {

   String message = "Balalalalal";
   ServiceInjector injector = null;
   Consumer app = null;

   // Use foo service.
   injector = new FooServiceInjector();
   app = injector.getConsumer();
   app.work(message);

   // Use bar service.
   injector = new BarServiceInjector();
   app = injector.getConsumer();
   app.work(message);
}
```

DI with Google Guice

Google Guice is an open source software framework for the Java platform released by Google under the Apache License, which provides support for dependency injection using annotations for configuration.

Guice allows implementation classes to be bound programmatically to an interface, then injected into constructors, methods or fields using an @Inject annotation. When more than one implementation of the same interface is needed, the user can create custom annotations that identify an implementation, then use that annotation when injecting it.

Guice was the first generic framework for dependency injection using Java annotations, in 2008, and is a popular tool when just DI is required in an application.

To provide a concise Guice overview:

With dependency injection, objects accept dependencies in their constructors. Constructing objects means first build its dependencies. But to build each dependency, you need its dependencies, and so on. So building an object, you really need to build an object graph.

Building object graphs by hand is labour intensive, laborious, and difficult to test. Guice can help, but needs to be configured to build the graph exactly as required.

For instance, a BillingService class accepts its dependent interfaces CardProcessor and TransactionLog in its constructor. To make it explicit that the BillingService constructor is invoked by Guice, add the @Inject annotation:

```
class BillingService {
   private final CardProcessor processor;
   private final TransactionLog transactionLog;

   @Inject
   BillingService(CardProcessor processor,
            TransactionLog transactionLog) {
      this.processor = processor;
         this.transactionLog = transactionLog;
   }

   public Receipt chargeOrder(DinnerOrder order, Card card) {
         ...
   }
}
```

Build a BillingService using VisaCardProcessor and XMLFileTransactionLog. Guice uses bindings to map types to their implementations. A module is a collection of bindings specified using fluent, English-like method calls:

```
public class BillingModule extends AbstractModule {
   @Override
   protected void configure() {

      /*
      * Tells Guice that whenever it sees a dependency
      * on a TransactionLog, it should satisfy the dependency
      * using a DatabaseTransactionLog.
      */
      bind(TransactionLog.class).to(XMLFileTransactionLog.class);

         /*
            * Binding tells Guice that when CardProcessor is used in
      * a dependency, it should be satisfied
* with a VisaCreditCardProcessor.
            */
         bind(CardProcessor.class).to(VisaCreditCardProcessor.class);
   }
}
```

The modules are the building blocks of an injector, which is Guice's object-graph builder. First, create the injector, and then use that to build the BillingService:

```
public static void main(String[] args) {
   /*
   * Guice.createInjector() takes Modules, returns a new Injector
   * instance.  Most applications call this method once, in main().
   */
   Injector injector = Guice.createInjector(new BillingModule());

      // Got the injector, now can build objects.
   BillingService billingService =
      injector.getInstance(BillingService.class);
      ...
}
```

In building the billingService, an object graph exists through Guice. The graph contains the billing service and its dependent card processor and transaction log.

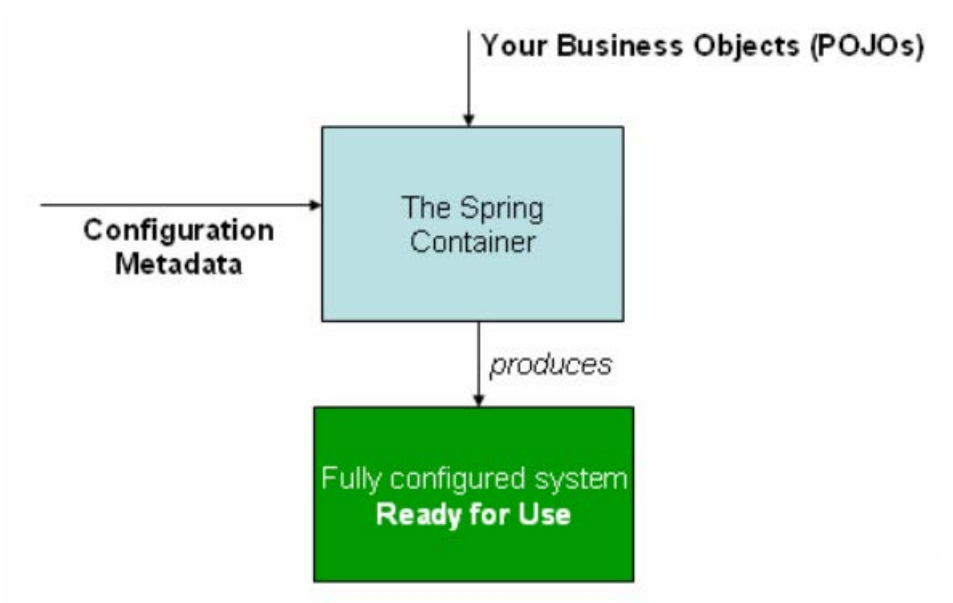
Spring IoC Container

Dependency injection is also known as Inversion of Control (IoC), a process where objects define their dependencies, or other objects they work with, through either constructor arguments, arguments to a factory method, or properties that are set on the object instance after it is constructed or returned from a factory method. The container then injects those dependencies when it creates the bean.

Spring uses the org.springframework.beans and org.springframework.context packages for its IoC container.

The BeanFactory provides the configuration framework and basic functionality, and the ApplicationContext adds enterprise-specific functionality.

In Spring, application objects are constructed and managed by the Spring IoC container, and are called beans. Beans, and their dependencies, are reflected in the container configuration metadata.



Configuration Metadata

The Spring IoC container consumes a form of configuration metadata; this configuration metadata represents how the developer tells the Spring container to instantiate, configure, and assemble application objects.

Configuration metadata is traditionally supplied in a simple and intuitive XML format,

XML-based metadata is not the only allowed form of configuration metadata. The Spring IoC container itself is totally decoupled from the format in which this configuration metadata is actually written. Many developers choose the alternate Java-based configuration.

Spring configuration consists of at least one and typically more than one bean definition that the container must manage. XML-based configuration metadata shows these beans configured as <bean/> elements inside a top-level <beans/> element. Java configuration typically uses @Bean annotated methods within a @Configuration class.

These bean definitions correspond to the actual objects that make up the application, defining service layer objects, data access objects (DAOs), presentation objects, infrastructure objects such as Hibernate SessionFactories, JMS Queues etc. Typically fine-grained domain objects are not configured in the container, because it is usually the responsibility of DAOs and business logic to create and load domain objects. However, Spring's integration with AspectJ can be used to configure objects that have been created outside the control of an IoC container.

The structure of XML-based configuration metadata:

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
    http://www.springframework.org/schema/beans/spring-beans.xsd">
    <bean id="..." class="...">
        <!-- bean config -->
    </bean>
    <bean id="..." class="...">
        <!-- bean config -->
    </bean>
    ...
</beans>
```

The id attribute is used to identify individual bean definitions. The class attribute defines the type of the bean and uses the fully qualified classname. The value of the id attribute refers to collaborating objects.

Container Instantiation

The location path or paths supplied to an ApplicationContext constructor are resource strings that allow the container to load configuration metadata from a variety of external resources; the file system, the classpath, and so on.

```
ApplicationContext context =
    new ClassPathXmlApplicationContext(
   new String[] {"myServices.xml", "myDaos.xml"});
```

The following example shows the service layer objects (myServices.xml) configuration file:

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-beans.xsd">

    <bean id="store" class="org.services.StoreServiceImpl">
        <property name="accountDao" ref="accountDao"/>
        <property name="itemDao" ref="itemDao"/>
        ...
    </bean>
    ...
</beans>
```

The following example shows the data access objects daos.xml file:

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
    http://www.springframework.org/schema/beans/spring-beans.xsd">

    <bean id="accountDao"
        class="org.dao.jpa.JpaAccountDao">
        ...
    </bean>

    <bean id="itemDao" class="org..dao.jpa.JpaItemDao">
        ...
    </bean>
    ...
</beans>
```

The service layer StoreServiceImpl, and two data access objects JpaAccountDao and JpaItemDao. The name property element refers to the name of the JavaBean property, and the ref element refers to the name of another bean definition; indicating the dependency between collaborating objects.

Using the Container

The ApplicationContext is the interface for an advanced factory capable of maintaining a registry of different beans and their dependencies.

Use the method to retrieve instances of beans.

```
T getBean(String name, Class<T> requiredType)
```

The ApplicationContext enables the reading of bean definitions and access:

```
// Create and configure beans.
ApplicationContext context =
new ClassPathXmlApplicationContext(
   new String[] {"myServices.xml", "myDaos.xml"});

// Retrieve configured instance.
StoreService service = context.getBean("store", StoreService.class);

// Use configured instance.
List<String> userList = service.getUsernameList();
```

The getBean() method is used to retrieve bean instances. The ApplicationContext interface has a few other methods for retrieving beans, but ideally these are never used.

Application code should have no calls to the getBean() method at all, and thus no dependency on Spring APIs: Spring's integration with web frameworks allows dependency injection for various web framework classes such as controllers and managed beans.

Bean Overview

A Spring IoC container manages one or more beans. These beans are created with the configuration metadata supplied to the container in the form of XML <bean/> definitions, or equivalent.

Within the container itself, these bean definitions are represented as BeanDefinition objects, which contain metadata:

- A package-qualified class name: typically the actual implementation class of the bean being defined.
- Bean behavioural configuration elements, stating bean behaviour (scope, lifecycle callbacks etc).
- References to other beans required by the bean; also called collaborators or dependencies.
- Other configuration settings, such as the number of connections to use in a bean that manages a connection pool, or the size limit of the pool etc

In addition to bean definitions that contain information on how to create a specific bean, the ApplicationContext also permit the registration of existing objects created outside the container.

The ApplicationContext's BeanFactory can be accessed via the method getBeanFactory() and supports this registration through:

- registerSingleton(..)
- registerBeanDefinition(..)

Typical applications work solely with beans defined through metadata bean definitions.

INSTANTIATING BEANS

If using XML-based configuration metadata, specify the type (or class) of object that is to be instantiated in the class attribute of the <bean/> element. This class attribute, which internally is a Class property on a BeanDefinition instance, is usually mandatory. Use the Class property in one of two ways:

- Typically, to specify the bean class to be constructed in the case where the container itself directly creates the bean by calling its constructor reflectively, somewhat equivalent to Java code using the new operator.
- To specify the actual class containing the static factory method that will be invoked to create the object, in the less common case where the container invokes a static factory method on a class to create the bean. The object type may be the same class or another class entirely.

Constructor Instantiation

The Spring IoC container can manage virtually any class you want it to manage. Most Spring users prefer actual JavaBeans with only a default (no-argument) constructor and appropriate setters and getters modelled after the properties in the container. Spring can also manage less coherent classes.

To specify the bean class:

```
<bean id="exampleBean" class="examples.ExampleBean"/>
<bean name="anotherExample" class="examples.ExampleBeanTwo"/>
```

Static Factory Instantiation

The definition does not specify the type (class) of the returned object, only the class containing the factory method. In the example, the createInstance()must be a static method.

```
<bean id="clientService" class="com.ClientService"
factory-method="createInstance"/>

public class ClientService {
    private static ClientService clientService = new ClientService();
    private ClientService() {}

    public static ClientService createInstance() {
        return clientService;
    }
}
```

Factory Method Instantiation

Here, leave the class attribute empty, and in the factory-bean attribute, specify the name of a bean in the current (or parent/ancestor) container that contains the instance method that is to be invoked to create the object. Set the name of the factory method itself with the factory-method attribute.

One factory class can hold more than one factory method:

```
<!-- Factory bean, containing the createInstance() method -->
<bean id="serviceLocator" class="examples.DefaultServiceLocator">
    <!-- inject dependencies required by locator bean -->
</bean>

<!-- Beans to be created via the factory -->
<bean id="clientService" factory-bean="serviceLocator"
factory-method="createClientServiceInstance"/>

<bean id="accountService" factory-bean="serviceLocator"
factory-method="createAccountServiceInstance"/>

public class DefaultServiceLocator {

    private static ClientService clientService = new
ClientServiceImpl();
    private static AccountService accountService =
new AccountServiceImpl();

    private DefaultServiceLocator() {}

    public ClientService createClientServiceInstance() {
        return clientService;
    }

    public AccountService createAccountServiceInstance() {
        return accountService;
    }
}
```

This approach shows that the factory bean itself can be managed and configured through dependency injection (DI).

SPRING DEPENDENCY INJECTION

Constructor DI

Constructor argument resolution matching occurs using the argument's type. If no potential ambiguity exists in the constructor arguments of a bean definition, then the order in which the constructor arguments are defined in a bean definition is the order in which those arguments are supplied to the appropriate constructor when the bean is being instantiated. Consider:

```
public class Foo {
    public Foo(Bar bar, Baz baz) { ... }
}
```

No potential ambiguity exists, assuming Bar and Baz are not related by inheritance. Thus the following configuration works, and constructor argument indexes and/or types explicitly are not required in the <constructor-arg/> element.

```
<beans>
    <bean id="foo" class="com.Foo">
        <constructor-arg ref="bar"/>
        <constructor-arg ref="baz"/>
    </bean>

    <bean id="bar" class="com.Bar"/>
    <bean id="baz" class="com.Baz"/>
</beans>
```

When another bean is referenced, the type is known, and matching can occur (as was the case with the preceding example). When a simple type is used, such as <value>true</value>, Spring cannot determine the type of the value, and so cannot match by type without help:

```
public class ExampleBean {

    private int years;
    private String answer;

    public ExampleBean(int years, String answer) {
        this.years = years;
        this.answer = answer;
    }
}
```

The container can use type matching with simple types if you explicitly specify the type of the constructor argument using the type attribute:

```
<bean id="exampleBean" class="com.ExampleBean">
    <constructor-arg type="int" value="7500000"/>
    <constructor-arg type="java.lang.String" value="42"/>
</bean>
```

The index attribute to specify explicitly the index of constructor arguments:

```
<bean id="exampleBean" class="com.ExampleBean">
    <constructor-arg index="0" value="7500000"/>
    <constructor-arg index="1" value="42"/>
</bean>
```

In addition to resolving the ambiguity of multiple simple values, specifying an index resolves ambiguity where a constructor has two arguments of the same type. Note that the index is 0 based. The constructor parameter name can also be used:

```
<bean id="exampleBean" class="com.ExampleBean">
    <constructor-arg name="years" value="7500000"/>
    <constructor-arg name="answer" value="42"/>
</bean>
```


Setter DI

Setter-based DI is accomplished by the container calling setter methods on beans after invoking a no-argument constructor or no-argument static factory method to instantiate the bean.

The following example shows a class that can only be dependency-injected using pure setter injection. This class is conventional Java. It is a POJO that has no dependencies on container specific interfaces, base classes or annotations.

```
public class MovieLister {

    private MovieFinder movieFinder;

    public void setMovieFinder(MovieFinder movieFinder) {
        this.movieFinder = movieFinder;
    }
    ...
}
```

The ApplicationContext supports constructor-based and setter-based DI for the beans it manages. It also supports setter-based DI after some dependencies have already been injected through the constructor approach. Configure the dependencies in the form of a BeanDefinition, which you use in conjunction with PropertyEditor instances to convert properties from one format to another.

Most Spring developers do not work with these classes directly but with XML bean definitions, annotated components (using @Component, @Controller, etc.), or @Bean methods in Java-based @Configuration classes. These sources are then converted internally into instances of BeanDefinition and used to load an entire Spring IoC container instance.

Which DI?

A good rule of thumb to use constructors for mandatory dependencies and setter methods or configuration methods for optional dependencies. The @Required annotation on a setter method can be used to make the property a required dependency.

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

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

Sometimes, when dealing with third-party classes for which you do not have the source, the choice is made for you. For example, if a third-party class does not expose any setter methods, then constructor injection may be the only available form of DI.

The container performs bean dependency resolution as follows:

- The ApplicationContext is created and initialized with configuration metadata that describes all the beans.
- For each bean, its dependencies are expressed in the form of properties, constructor arguments, or arguments to the static-factory method if you are using that instead of a normal constructor. These dependencies are provided to the bean, when the bean is actually created.
- Each property or constructor argument is an actual definition of the value to set, or a reference to another bean in the container.
- Each property or constructor argument which is a value is converted from its specified format to the actual type of that property or constructor argument. By default Spring can convert a value supplied in string format to all built-in types, such as int, long, String, boolean, etc.

The Spring container validates the configuration of each bean as the container is created. However, the bean properties themselves are not set until the bean is actually created. Beans that are singleton-scoped and set to be pre-instantiated (the default) are created when the container is created, otherwise, the bean is created only when it is requested. Creation of a bean potentially causes a graph of beans to be created, as the bean's dependencies and its dependencies' dependencies (and so on) are created and assigned.

DI Example

1. In Eclipse, create a new Maven Project using the archetype-maven-quickstart as shown:

Use the following settings:

```
Group Id com.training
Artifact Id example
Version  0.0.1-SNAPSHOT
```

Package com.training.example

2. Add Spring Framework support to the POM:

```
<properties>
   <spring.version>4.1.4.RELEASE</spring.version>
</properties>
<dependencies>
   <dependency>
      <groupId>org.springframework</groupId>
      <artifactId>spring-core</artifactId>
      <version>${spring.version}</version>
   </dependency>
   <dependency>
      <groupId>org.springframework</groupId>
      <artifactId>spring-context</artifactId>
      <version>${spring.version}</version>
   </dependency>
</dependencies>
```

3. Immediately after the </dependencies> tag, add support for the maven-compiler-plugin, in order to bring Java up to a modern version:

```
<build>
  <plugins>
   <plugin>
      <artifactId>maven-compiler-plugin</artifactId>
      <version>3.5.1</version>
      <configuration>
         <source>1.8</source>
         <target>1.8</target>
      </configuration>
   </plugin>
  </plugins>
</build>
```

In order for the changes to take place, run Maven | Update Project (Alt + F5).

4. In src/main/resources, create an applicationContext.xml. Use one of either constructor or setter injection.

You may need to create src/main/resources as a new source folder.

In the new applicationContext.xml, add XML-based configuration metadata for setter-based DI, for instance:

```
<bean id="helloBean" class="com.training.example.Hello">
   <constructor-arg value="Greg" />
</bean>

<!-- Or, can use property call. -->
<bean id="helloBean" class="com.training.example.Hello">
   <property name="name" value="Greg" />
</bean>
```

5. Proceed to define the simple POJO referenced in the XML:

```
public class Hello {

   private String name;

   public Hello() { }

   public Hello(String name) {
      this.name = name;
   }

   public String getName() {
      return name;
   }

   public void setName(String name) {
      this.name = name;
   }
}
```

6. In Main.main(), create the container from the context, and lookup the bean.

The ApplicationContext is the interface for an advanced factory capable of maintaining a registry of different beans and their dependencies. Using the method T getBean(String name, Class<T> requiredType), retrieve instances of beans.

The ApplicationContext enables the reading of bean definitions and accessing them.

The standalone XML application context, ClassPathXmlApplicationContext takes the context definition files from the class path, interpreting plain paths as class path resource names that include the package path (e.g. "mypackage/myresource.txt"). Useful for test harnesses as well as for application contexts embedded within JARs:

```
// Create and configure beans.
try (ClassPathXmlApplicationContext context =
      new ClassPathXmlApplicationContext(
"/applicationContext.xml")) {
   // Retrieve configured instance.
   Hello output = (Hello) context..getBean("helloBean");

   // Use configured instance.
   System.out.println(output.getName());
}
```

7. Run the application to show the name.

Autowiring

As discussed in the previous section, bean dependencies can be set in the configuration file, and this is normally the best practice. There is an alternative process that can be followed instead. This is called autowiring, and it involves allowing the Spring container to automatically build relationships between collaborating beans by inspecting the contents of BeanFactory.

Autowiring is a feature provided by the Spring framework that helps to reduce some of the configuration required to enable Spring beans to reference one another. For example, every member variable in the Spring bean has to be configured and if a bean references another bean, we have to specify the reference explicitly.

With autowiring, Spring provides features where it is not necessary to provide bean injection details explicitly. Instead the framework intelligently guesses what the reference is. The Spring container can autowire relationships between collaborating beans without using the <constructor-arg> and <property> elements.

The main advantage of autowiring is that it reduces the size and complexity of the XML in configuration files. Spring is capable of automatically resolving dependencies at runtime. This automatic resolution of bean dependencies is also referred to as autowiring.

By default autowiring is disabled. To enable it, developers must specify which beans they want to autowire and the autowiring mode to be used by the Spring container.

The autowiring modes available are byname, byType, constructor, and autodetect.

Autowiring can be configured using XML in the applicationContext.xml file, or using annotations since Spring 2.5.

Configuring Autowiring

An example XML configuration syntax is shown below:

```
<bean id="myBean" class ="MyBean" autowire="byName" />
```

The root beans element has a default-autowire attribute. This can be used to override the default autowiring mode for all beans in the configuration file. This can be used if you want all beans to participate in autowiring. Each bean also has an autowire-candidate attribute that defaults to true. To have it excluded from the container's autowiring default it can be set to false.

Spring recommends not using autowiring for large configurations since what is being set on a bean can become difficult to identify. Bean inheritance is a good alternative to autowiring since it reduces repetitive injections, but it's still very clear in the configuration what's happening. Autowiring is very powerful, but should be used with caution since it's possible for a property to be set that wasn't intended to be set as development continues.

Autowiring Modes

The autowire element of the <bean> tag specifies the autowiring mode to be used. The following table describes the different modes available.

Mode	Description
no	No autowiring is performed. All references to other beans must be explicitly injected. This is the default mode.
byName	Based on the name of a property, a matching bean name in the IoC container will be injected into this property if it exists.
byType	Based on the type of class on a setter, if only one instance of the class exists in the IoC container it will be injected into this property. If there is more than one instance of the class a fatal exception is thrown.
constructor	Based on a constructor argument's class types, if only one instance of the class exists in the IoC container it will be used in the constructor.
autodetect	If there is a valid constructor, the constructor autowiring mode will be chosen. Otherwise if there is a default no-argument constructor the byType autowiring mode will be chosen.

byName

The autowiring 'byName' mode matches a property name to a bean name in the IoC container. For example a property of setHibernateTemplate(HibernateTemplate template) has a property name of 'hibernateTemplate'. So if a hibernateTemplate bean is found it will automatically be injected during the autowiring process.

byType

When using the byType autowiring mode the class of a setter will be used to find a bean. If there is more than one class found, a fatal exception is thrown. If there isn't a matching class in the IoC container, nothing will be set and there won't be an exception during processing. If this is undesirable behaviour, the bean element's dependency-check attribute can be set to 'object'. Then an error will occur if there no matching class exists for the property during the autowire process.

constructor

The constructor autowiring mode is similar to the byType autowiring mode, but it's for constructors. The constructor with the most parameters that can successfully be autowired will be chosen. There can only be one bean definition for a type to be autowired successfully. If one constructor takes a Department class and the other takes an Employee class, then if there are two Employee bean definitions and only one bean definition for a Department the constructor that takes the Department class will be used to instantiate the bean. If there is a no parameter constructor, this will be used rather than throw an exception. If there isn't an eligible constructor, a fatal exception will be thrown.

Using XML

The following example wires the bean explicitly using a ref attribute:

```
<bean id="department" class="com.training.Department" >
   <property name="employee" ref="employee" />
</bean>

<bean id="employee" class="com.training.Employee" >
   <property name="empname" value="Fred" />
</bean>
```

If autowire by name is configured, there is no longer any need for the property tag. As long as the "employee" bean has same name as the property of the "department" bean, which is "employee", Spring will wire it automatically.

```
<bean id="department" class=" com.training.Department"
   autowire="byName" />

<bean id=" employee " class="com.training.Employee" >
   <property name=" empname " value="Fred" />
</bean>
```


Using Annotations

The following examples use a Department bean declared in bean configuration file, as shown below. This well be autowired to the Employee bean with the @Autowired annotation.

```
public class Department {

   // This is the field to autowire.
   private Employee employee;

   private int type;
   private String action;
   …
}
```

This is declared as shown below:

```
<beans xmlns=…">
   <bean id="DepartmentBean" class="com.training.Department">
      <property name="location" value="London" />
   </bean>

   <bean id="EmployeeBean" class="com.training.Employee">
      <property name="name" value="Fred" />
      <property name="salary" value="30000" />
   </bean>
</beans>
```

Next the AutowiredAnnotationBeanPostProcessor must be registered to enable the @Autowired annotation. There are two ways to achieve this:

Enable Spring context by adding <context:annotation-config /> to the bean configuration file.

```
<beans ...http://www.springframework.org/schema/
context/spring-context.xsd">
   …
   <context:annotation-config />
   ...
</beans>
```

Alternatively, the developer can include AutowiredAnnotationBeanPostProcessor directly in the configuration file.

```
<beans xmlns=…>
<bean class="org.springframework.beans.factory.annotation
.AutowiredAnnotationBeanPostProcessor"/>

   <bean id="DepartmentBean" class=" com.training.Department">
      <property name="location" value="London" />
   </bean>

   <bean id="EmployeeBean" class=" com.training.Employee">
      <property name="name" value="Fred" />
      <property name="salary" value="30000" />
   </bean>
</beans>
```

The class can have the @Autowired annotation applied to a field, a setter method or a constructor.

Using a field:

```
import org.springframework.beans.factory.annotation.Autowired;

public class Department {

   @Autowired
   private Employee employee;
   private String location;
   …
}
```

Using a setter method:

```
import org.springframework.beans.factory.annotation.Autowired;

public class Department {

   private Employee employee;
   private String location;
   …

   @Autowired
   public void setEmployee(Employee Employee) {
      this.Employee = Employee;
   }
}
```


Using a constructor:

```
import org.springframework.beans.factory.annotation.Autowired;

public class Department {

   private Employee employee;
   private String location;
   …

   @Autowired
   public Department(Employee employee) {
      this.employee = employee;
   }
}
```

Finally a class with a main method can invoke the beans:

```
import org.springframework.context.ApplicationContext;
import org.springframework.context.support.
ClassPathXmlApplicationContext;

public class StaffMain {

    public static void main( String[] args ) {
      ApplicationContext context =
         New ClassPathXmlApplicationContext(
            new String[] {"SpringBeans.xml"});

      Department dept =
         (Department) context.getBean("DepartmentBean");
      System.out.println(dept);

    }
}
```

Autowiring Drawbacks

Autowiring works well when it is used consistently. It could be confusing to see wiring used for only a few bean definitions. However, it can be confusing (if not using IntelliJ) as searching for the annotation can turn into a game of cat and mouse.

There are also a few drawbacks to autowiring:

Explicit dependencies in property and constructor-arg settings always override autowiring. Simple properties such as primitives, Strings, and Classes cannot be autowired. This limitation is by-design.

While Spring is careful to avoid ambiguous wiring, autowiring is always going to be less exact than explicit wiring.

Tools and IDEs may not be able to generate documentation from the container, or trace the wiring annotations with any clarity.

If no unique bean definition is available, an exception is thrown. There are then several options:

- Abandon autowiring for explicit wiring.
- Avoid autowiring for a bean definition by setting its autowire-candidate attributes to false.
- Designate a single bean definition as the primary candidate by setting the primary attribute of its <bean/> element to true.
- Implement the more fine-grained control available with annotation-based configuration.

Excluding a Bean from Autowiring

Useful for beans that are never to be injected into other beans by autowiring. It does not mean that an excluded bean cannot itself be configured using autowiring. Rather, the bean itself is not a candidate for autowiring other beans.

- Set the autowire-candidate attribute of the <bean/> element to false; the container makes that specific bean definition unavailable to the autowiring infrastructure.
- Alternatively limit autowire candidates based on pattern-matching against bean names. The top-level <beans/> element accepts one or more patterns within its default-autowire-candidates attribute.

CHAPTER 3

Spring AOP

Introduction

Aspect-Oriented Programming (AOP) is a complement to Object-Oriented Programming (OOP); providing another way of thinking about program structure. The key unit of modularity in OOP is the class, whereas in AOP the unit of modularity is the aspect.

Aspects enable the modularization of concerns such as transaction management that cut across multiple types and objects. These are often termed crosscutting concerns in AOP literature.

One of the key components of Spring is the AOP framework. While the Spring IoC container does not depend on AOP, AOP complements Spring IoC to provide a very capable middleware solution.

Spring 2.0 introduced a simpler and more powerful way of writing custom aspects using either a schema-based approach or the @AspectJ annotation style. Both of these styles offer fully typed advice and use of the AspectJ pointcut language, while still using Spring AOP for weaving.

AOP is used with Spring to:

- Provide declarative enterprise services, especially as a replacement for EJB declarative services. The most important being declarative transaction management.
- Allow developers to provide custom aspects, complementing their use of OOP with AOP.

AOP Concepts

AOP terminology is not particularly intuitive; however, it would be even more confusing if Spring used its own terminology.

Aspect

A modularization of a concern that cuts across multiple classes. Transaction management is a good example of a crosscutting concern in enterprise Java applications.

Join point

A point during execution such as the execution of a method or the handling of an exception.

Advice

Action taken by an aspect at a particular join point. Include "around," "before" and "after" advice.

Pointcut

A predicate that matches join points. Advice is associated with a pointcut expression and runs at any join point matched by the pointcut.

Introduction

Declaring additional methods or fields on behalf of a type. Spring AOP allows for the introduction of new interfaces (and implementations) to any advised object.

Target object

Object being advised by one or more aspects; also referred to as the advised object. This will always be a proxied object.

AOP proxy

An object created by the AOP framework in order to implement the aspect contracts (advise method executions and so on).

Weaving

Linking aspects with other application types or objects to create an advised object. This can be done at compile time, load time, or runtime. Spring AOP performs weaving at runtime.

Advice Types

Before advice

Executes before a join point, but which does not have the ability to prevent execution flow proceeding to the join point (unless it throws an exception).

After returning advice

Executed after a join point completes normally.

After throwing advice

Advice to be executed if a method exits by throwing an exception.

After (finally) advice

Executed regardless of the means by which a join point exits (normal or exceptional return).

Around advice

Advice that surrounds a join point such as a method invocation. This is the most powerful kind of advice. Around advice can perform custom behaviour before and after the method invocation. It is also responsible for choosing whether to proceed to the join point or to shortcut the advised method execution by returning its own return value or throwing an exception.

It is recommended to use the least powerful advice type that can implement the required behaviour.

The concept of join points, matched by pointcuts, is the key to AOP which distinguishes it from older technologies offering only interception. Pointcuts enable advice to be targeted independently of the Object-Oriented hierarchy. For example, an around advice providing declarative transaction management can be applied to a set of methods spanning multiple objects (such as all business operations in the service layer).

AspectJ Support

AspectJ refers to a style of declaring aspects as regular Java classes annotated with annotations. Spring interprets the same annotations as AspectJ 5, using a library supplied by AspectJ for pointcut parsing and matching. The AOP runtime is still pure Spring AOP though, and there is no dependency on the AspectJ compiler or weaver.

Enabling @AspectJ

To use @AspectJ aspects in a Spring configuration, enable Spring support for configuring Spring AOP based on @AspectJ aspects, and autoproxying beans based on whether or not they are advised by those aspects. By autoproxying we mean that if Spring determines that a bean is advised by one or more aspects, it will automatically generate a proxy for that bean to intercept method invocations and ensure that advice is executed as needed.

The @AspectJ support can be enabled with XML or Java style configuration. In either case AspectJ's aspectjweaver.jar library is required on the classpath.

AspectJ Java Configuration

Add the @EnableAspectJAutoProxy annotation:

```
@Configuration
@EnableAspectJAutoProxy
public class AppConfig {
...
}
```

AspectJ XML Configuration

To enable @AspectJ support with XML based configuration use the aop:aspectj-autoproxy element:

```
<aop:aspectj-autoproxy/>
```

This assumes that schema support is being used.

Declaring an Aspect

Any bean defined in the application context with a class that is an @AspectJ aspect will be automatically detected by Spring and used to configure Spring AOP. The minimal definition required for a regular bean definition in the application context, pointing to a bean class that has the @Aspect annotation:

```
<bean id="myAspect" class="com.training.AnotherAspect">
    <!-- configure properties of aspect here as normal -->
</bean>
```

And the AnotherAspect class definition, annotated with the

org.aspectj.lang.annotation.Aspect annotation;

```
@Aspect
public class AnotherAspect {
...
}
```

Aspects (classes annotated with @Aspect) may have methods and fields just like any other class. They may also contain pointcut, advice, and introduction (inter-type) declarations.

Aspect classes may be added as regular beans in the Spring XML configuration, or autodetected through classpath scanning. However, the @Aspect annotation is not sufficient for autodetection in the classpath, add a separate @Component annotation

In Spring AOP, it is not possible to have aspects themselves be the target of advice from other aspects. The @Aspect annotation on a class marks it as an aspect, and hence excludes it from auto-proxying.

Declaring a Pointcut

Pointcuts determine join points of interest, and thus enable control of when advice executes. Spring AOP only supports method execution join points for Spring beans, so think of a pointcut as matching the execution of methods on Spring beans.

A pointcut declaration has two parts: a signature comprising a name and any parameters, and a pointcut expression that determines exactly which method executions are used. In the @AspectJ annotation-style of AOP, a pointcut signature is provided by a regular method definition, and the pointcut expression is indicated using the @Pointcut annotation (the method serving as the pointcut signature must have a void return type).

The following defines a pointcut named 'anyOldTransfer' that will match the execution of any method named 'transfer':

```
@Pointcut("execution(* transfer(..))")   // Pointcut expression
private void anyOldTransfer() {}   // Pointcut signature
```

The pointcut expression that forms the value of the @Pointcut annotation is a regular AspectJ 5 pointcut expression.

Supported Pointcut Designators

Spring AOP supports the following AspectJ pointcut designators (PCD) for use in pointcut expressions. The set of pointcut designators supported by Spring AOP may be extended in future releases to support more of the AspectJ pointcut designators.

execution

For matching method execution join points, this is the primary pointcut designator used when working with Spring AOP

within

Limits matching to join points within certain types (the execution of a method declared within a matching type with Spring AOP)

this

Limits matching to join points (the execution of methods with Spring AOP) where the bean reference (Spring AOP proxy) is an instance of the given type

target

Limits matching to join points (the execution of methods with Spring AOP) where the target object (application object being proxied) is an instance of the given type

args

Limits matching to join points (the execution of methods with Spring AOP) where the arguments are instances of the given types

@target

Limits matching to join points (the execution of methods with Spring AOP) where the class of the executing object has an annotation of the given type

@args

Limits matching to join points (the execution of methods with Spring AOP) where the runtime type of the actual arguments passed have annotations of the given type(s)

@within

Limits matching to join points within types that have the given annotation (the execution of methods declared in types with the given annotation with Spring AOP)

@annotation

Limits matching to join points where the subject of the join point (method being executed in Spring AOP) has the given annotation

Because Spring AOP limits matching to only method execution join points, the discussion of the pointcut designators above gives a narrower definition than found in the AspectJ programming guide. In addition, AspectJ itself has type-based semantics and at an execution join point both this and target refer to the same object - the object executing the method. Spring AOP is a proxy-based system and differentiates between the proxy object itself (bound to this) and the target object behind the proxy (bound to target).

Spring AOP also supports an additional PCD named bean. This PCD allows the developer to limit the matching of join points to a particular named Spring bean, or to a set of named Spring beans (when using wildcards). The bean PCD has the following form:

```
bean(idOrNameOfBean)
```

The idOrNameOfBean token can be the name of any Spring bean: limited wildcard support using the * character is provided, so when using a naming convention in Spring beans it is easy to write a bean PCD expression to pick them out. As is the case with other pointcut designators, the bean PCD can be &&'ed, ||'ed, and ! (negated) too.

Combining Pointcut Expressions

Pointcut expressions can be combined using &&, || and !. It is also possible to refer to pointcut expressions by name. The following example shows three pointcut expressions: anyPublicOperation (which matches if a method execution join point represents the execution of any public method); inTrading (which matches if a method execution is in the trading module), and tradingOperation (which matches if a method execution represents any public method in the trading module).

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

@Pointcut("within(com.xyz.someapp.trading..*)")
private void inTrading() {}

@Pointcut("anyPublicOperation() && inTrading()")
private void tradingOperation() {}
```

It is a best practice to build more complex pointcut expressions out of smaller named components as shown above. When referring to pointcuts by name, normal Java visibility rules apply (see private pointcuts in the same type, protected pointcuts in the hierarchy, public pointcuts anywhere and so on). Visibility does not affect pointcut matching.

Declaring Advice

Advice is associated with a pointcut expression, and runs before, after, or around method executions matched by the pointcut. The pointcut expression may be either a simple reference to a named pointcut, or a pointcut expression declared in place.

Before Advice

Before advice is declared in an aspect using the @Before annotation:

```
@Aspect
public class BeforeExample {

    @Before("com.SystemArchitecture.dataAccessOperation()")
    public void doAccessCheck() {
        ...
    }
}
```

If using an in-place pointcut expression we could rewrite the above example as:

```
@Aspect
public class BeforeExample {

    @Before("execution(* com.dao.*.*(..))")
    public void doAccessCheck() {
        ...
    }
}
```

After (Finally) Advice

After (finally) advice runs however a matched method execution exits. It is declared using the @After annotation. After advice must be prepared to handle both normal and exception return conditions. It is typically used for releasing resources:

```
@Aspect
public class AfterFinallyExample {

    @After("com.SystemArchitecture.dataAccessOperation()")
    public void doReleaseLock() {
       ...
    }
}
```

After Returning Advice

After returning advice runs when a matched method execution returns normally. It is declared using the @AfterReturning annotation:

```
@Aspect
public class AfterReturningExample {

    @AfterReturning("com.SystemArchitecture.dataAccessOperation()")
    public void doAccessCheck() {
        ...
    }
}
```

Note

It is possible to have multiple advice declarations inside the same aspect.

In order to provide access in the advice body to the actual value that was returned, use the form of @AfterReturning that binds the return value:

```
@Aspect
public class AfterReturningExample {

    @AfterReturning(
        pointcut="com.SystemArchitecture.dataAccessOperation()",
        returning="retVal")
    public void doAccessCheck(Object retVal) {
        ...
    }
}
```

The name used in the returning attribute must correspond to the name of a parameter in the advice method. When a method execution returns, the return value will be passed to the advice method as the corresponding argument value. A returning clause also restricts matching to only those method executions that return a value of the specified type (Object in this case, which will match any return value).

Note

It is not possible to return a different reference when using after-returning advice.

After Throwing Advice

After throwing advice runs when a matched method execution exits by throwing an exception. It is declared using the @AfterThrowing annotation:

```
@Aspect
public class AfterThrowingExample {

    @AfterThrowing("com.SystemArchitecture.dataAccessOperation()")
    public void doRecoveryActions() {
        ...
    }
}
```

If the advice needs to run only when exceptions are thrown, and access to exception is needed in the advice body, use the throwing attribute to both restrict matching (if desired, use Throwable as the exception type otherwise) and bind the thrown exception to an advice parameter

```
@Aspect
public class AfterThrowingExample {

    @AfterThrowing(
        pointcut="com.SystemArchitecture.dataAccessOperation()",
         throwing="ex")
    public void doRecoveryActions(DataAccessException ex) {
        ...
    }
}
```

The name used in the throwing attribute must correspond to the name of a parameter in the advice method. When a method execution exits by throwing an exception, the exception will be passed to the advice method as the corresponding argument value. A throwing clause also restricts matching to only those method executions that throw an exception of the specified type.

Around Advice

Around advice can do work both before and after the method executes, and to determine when, how, and even if, the method actually gets to execute at all. Around advice is often used to share state before and after a method execution in a thread-safe manner (starting and stopping a timer for example). Don't use around advice if simple before advice would fit the requirement.

Around advice is declared using the @Around annotation. The first parameter of the advice method must be of type ProceedingJoinPoint. Within the body of the advice, calling proceed() on the ProceedingJoinPoint causes the underlying method to execute. The proceed method may also be called passing in an Object[] - the values in the array will be used as the arguments to the method execution when it proceeds.

```
@Aspect
public class AroundExample {

    @Around("com.SystemArchitecture.businessService()")
    public Object doBasicProfiling(ProceedingJoinPoint pjp)
      throws Throwable {
      // Start stopwatch.
         Object retVal = pjp.proceed();
         // Stop stopwatch.
         return retVal;
    }
}
```

The value returned by the around advice will be the return value seen by the caller of the method. A simple caching aspect for example could return a value from a cache if it has one, and invoke proceed() if it does not.

Note

Proceed may be invoked once, many times, or not at all within the body of the around advice.

Schema AOP Example

1. Add Spring AOP AspectJ dependencies to the POM.

```
<dependency>
   <groupId>org.aspectj</groupId>
   <artifactId>aspectjrt</artifactId>
   <version>${aspectj.version}</version>
   <scope>runtime</scope>
</dependency>
<dependency>
   <groupId>org.aspectj</groupId>
   <artifactId>aspectjtools</artifactId>
   <version>${aspectj.version}</version>
</dependency>
```

2. Create a simple POJO.

```
package com.training.model;
import com.training.aspect.Loggable;

public class Employee {
private String name;

   public String getName() {
      return name;
   }

   @Loggable
   public void setName(String name) {
      this.name = name;
   }

   public void throwException() {
      throw new RuntimeException("Test Exception");
   }
}
```

Loggable being a custom annotation, provided by the following interface:

```
   package com.training.aspect;

   public @interface Loggable { }
```

3. Create a service class; an access point for the Employee.

```
package com.training.service;

import com.training.model.Employee;

public class EmployeeService {

   private Employee employee;

   public Employee getEmployee(){
      return employee;
   }

   public void setEmployee(Employee employee){
      this.employee = employee;
   }
}
```

4. Add AOP configuration to the applicationContext.xml.

Add the following to <beans>:

```
xsi:schemaLocation="http://www.springframework.org/schema/beans
   http://www.springframework.org/schema/beans/spring-beans.xsd
   http://www.springframework.org/schema/aop
   http://www.springframework.org/schema/aop/spring-aop.xsd"
```

Enable the AspectJ style of AOP at runtime

```
<aop:aspectj-autoproxy />
```

Add the POJO and service beans:

```
   <bean name="employee" class="com.training.model.Employee">
      <property name="name" value="Greg"></property>
   </bean>

   <bean name="employeeService"
      class="com.training.service.EmployeeService">
      <property name="employee" ref="employee"></property>
   </bean>
```


Add Spring AOP XML configuration:

```
<aop:config>
   <aop:aspect ref="employeeXMLConfigAspect"
      id="employeeXMLConfigAspectId"
      order="1">
         <aop:pointcut expression="execution(*
         com.training.model.Employee.getName())"
         id="getNamePointcut" />
      <aop:around method="employeeAroundAdvice"
         pointcut-ref="getNamePointcut"
         arg-names="proceedingJoinPoint" />
   </aop:aspect>
</aop:config>
```

Add configuration for an Aspect bean, to be built in the next step:

```
<bean name="employeeXMLConfigAspect"
   class="com.training.aspect.EmployeeXMLConfigAspect" />
```

5. Define the aspect class:

```
package com.training.aspect;

import org.aspectj.lang.ProceedingJoinPoint;

public class EmployeeXMLConfigAspect {

   public Object employeeAroundAdvice(
      ProceedingJoinPoint proceedingJoinPoint){

      System.out.println("EmployeeXMLConfigAspect:
            Before invoking getName()");
      Object value = null;
      try {
         value = proceedingJoinPoint.proceed();
      } catch (Throwable e) {
         e.printStackTrace();
      }
      System.out.println("EmployeeXMLConfigAspect:
         After invoking getName(). Return value: " + value);
      return value;
   }
}
```

6. Define a standard Main.main() to invoke the Employee class via the service:

```
public static void main(String[] args) {
   ClassPathXmlApplicationContext context =
      new ClassPathXmlApplicationContext(
         "applicationContext.xml");
   EmployeeService employeeService =
      context.getBean("employeeService",
         EmployeeService.class);
   System.out.println(employeeService.getEmployee().getName());
   employeeService.getEmployee().setName("Curtis");
   employeeService.getEmployee().throwException();
   context.close();
}
```

The invocation should produce the following:

```
EmployeeXMLConfigAspect: Before invoking getName()
EmployeeXMLConfigAspect: After invoking getName(). Return value: Greg
   Greg
Exception in thread "main" java.lang.RuntimeException: Test Exception
   at com.training.model.Employee.throwException(Employee.java:19)
   ...
```

This shows that the advice has been executed according to the pointcut definition.

Add Around Aspect Using Annotations

1. Add another bean to the config to represent another aspect:

```
<bean name="employeeAroundAspect"
   class="com.training.aspect.EmployeeAroundAspect" />
```

This bean is to be configured using annotations within the bean itself.

2. Create the Aspect:

```
@Aspect
public class EmployeeAroundAspect {

   @Around("execution(*
      com.training.model.Employee.getName())")
   public Object employeeAroundAdvice(
         ProceedingJoinPoint proceedingJoinPoint){
      System.out.println("EmployeeAroundAspect :
         Before invoking getName()");
      Object value = null;
      try {
         value = proceedingJoinPoint.proceed();
      } catch (Throwable e) {
         e.printStackTrace();
      }
      System.out.println("EmployeeAroundAspect :
         After invoking getName(). Returns: " + value);
      return value;
   }
}
```

3. Comment the previous aspect, leaving just the around aspect effective.

Running the application should output the following, showing that execution is in accordance with the @Around annotation:

```
EmployeeAroundAspect: Before invoking getName()
EmployeeAroundAspect: After invoking getName(). Return value: Greg
Greg
Exception in thread "main" java.lang.RuntimeException: Test Exception
   at com.training.model.Employee.throwException(Employee.java:19)
```

Your instructor can add further scenarios to this example.

Add Advice With Annotation Pointcut

1. Add the bean definition to the config:

```
<bean name="employeeAnnotationAspect"
   class="com.training.aspect.EmployeeAnnotationAspect" />
```

2. Create the Aspect:

```
package com.training.aspect;

import org.aspectj.lang.annotation.Aspect;
import org.aspectj.lang.annotation.Before;

@Aspect
public class EmployeeAnnotationAspect {

   @Before("@annotation(com.training.aspect.Loggable)")
   public void myAdvice(){
      System.out.println("Executing myAdvice()");
   }
}
```

myAdvice() will only apply adive to the setName() method due to the presence of the annotation. This provides a clean, safe means of adding advice.

CHAPTER 4

Spring WebMVC

Introduction

The Model View Controller (MVC) design pattern was first described in 1979 and initially aimed at desktop computing. Since then, the pattern has evolved as architecture for web applications. The purpose of the MVC design pattern is to separate the components of a web application so they have particular roles and responsibilities.

Spring MVC is the web component of Spring Framework for building dynamic web applications using the MVC framework. Spring MVC offers many built-in classes that simplify development of web applications and require fewer configurations for development. It's easy to integrate Spring MVC with other web frameworks like Struts and JSF.

Here, we introduce MVC and Spring MVC, alongside the annotations and support classes required to build web applications.

Introducing MVC

MVC separates the concerns in the presentation layer into various components, with each component having its own functions. The loose coupling between the three main components enables parallel development.

MVC Components

MVC separates application components into three main types and defines the relationship between them:

Model

Implements the logic and is responsible for managing and maintaining data, usually in the form of POJOs.

View

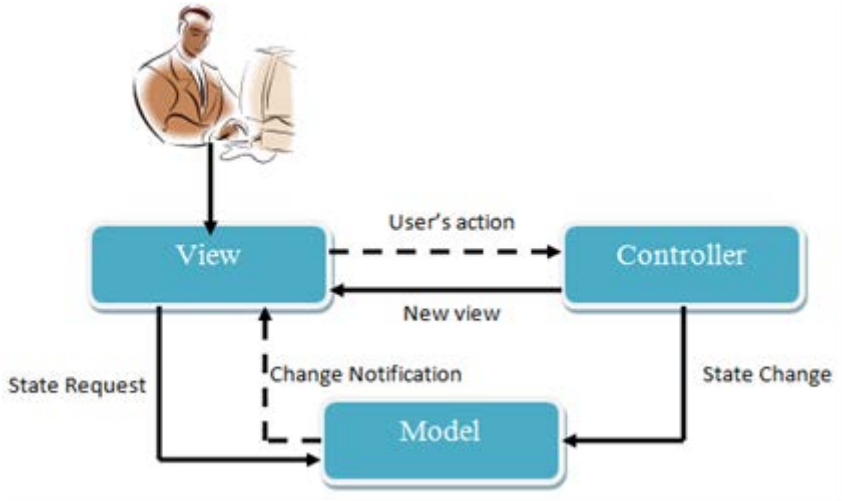
Represents the user interface. It generates output on the basis of the information passed on by the controller and changes gathered from the model.

Controller

Controls the interaction between the model and view, that is, the interactions between the application data and how the data delivered to the user. The controller handles user interaction, works with the model, and selects a view that displays the user interface. The controller passes on commands:

- To the model to update the its state
- To the view to update the presentation based on the updated model

Interactions among the three components of the MVC design pattern:



MVC Benefits

Separation of Concerns

The separation of the model, view, and controller within your web application allows the reuse of business logic across the application. For example, when a view is separated from the controller, it enables the controller to call on multiple views rendered to reflect every change in the data model. Similarly, a separate controller can handle requests from different presentation layer views. As a result, a single application can present multiple views to multiple users in a format that can be unique to each user.

Loose Coupling between Layers

Loose coupling of the different layers of the application reduces the dependency between them, and as a result, any change in one layer will not affect the other layers (such as the service, persistence, data access, and UI layers). For example, if you need to change the technical stack used in the database layer from SQL to NOSQL, then you can change the code only in the database layer. Loosely coupled components in an application also simplify testing of the code, especially unit testing.

Easy Code Maintenance

The separation of concerns allowed by the MVC design pattern enables you to separately develop multiple user interfaces without accessing the business logic code bases. By separating concerns, you can reuse methods and objects in multiple components, which reduces the work needed to maintain a project. In addition, when you separate the web-based code from the main business logic, you can protect the integrity of the application.

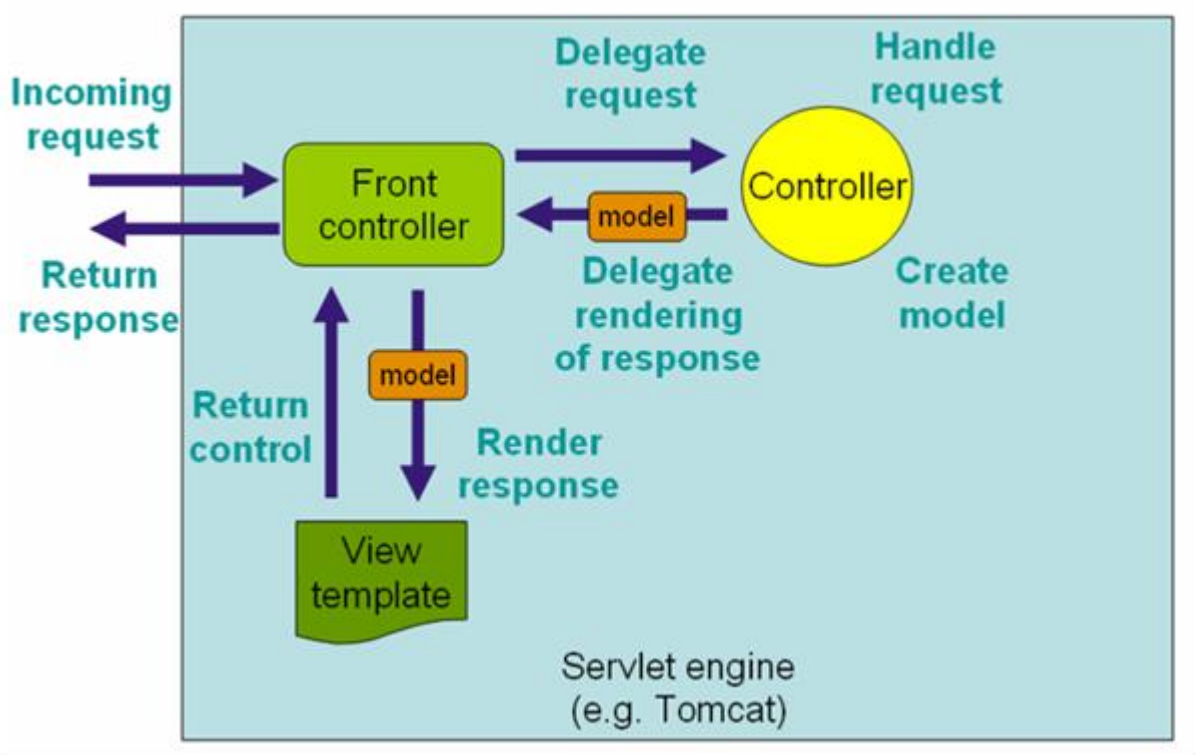
High Level Spring MVC Flow

An event can be triggered, causing the controller to manipulate the data in the model. The model is updated and the changes pushed to the view, which then updates itself with the latest model data.

In a web application, an event can be triggered by raising a request. The application updates and renders the view which is returned to the user. Therefore, rather than pushing the changes to the view, the changes are pulled from the server. However, in a web environment, the flow of tasks as seen in MVC design pattern does not work due to the nature of HTTP being stateless, so pulling changes using a model is not easy, as the model works on the states of data it controls.

Spring's web MVC framework is request-driven, designed around a central Servlet that dispatches requests to controllers and offers other functionality that facilitates the development of web applications. Spring's DispatcherServlet is also completely integrated with the Spring IoC container and as such allows all of Springs features.

The request processing workflow of the Spring Web MVC DispatcherServlet is illustrated in the following diagram. The DispatcherServlet is an expression of the "Front Controller" design pattern:



The Servlet engine (front controller) handles incoming requests. The request is delegated to an appropriate controller, which processes and updates the model.

The front controller then determines which view to render based on the response from the controller. Usually, this front controller is implemented as a Servlet.

The DispatcherServlet is an actual Servlet (based on HttpServlet), and declared in web.xml. Map requests to be handled by DispatcherServlet using a URL mapping in the same file:

```
<web-app>
      <servlet>
            <servlet-name>example</servlet-name>
            <servlet-class>
               org.springframework.web.servlet.DispatcherServlet
            </servlet-class>
            <load-on-startup>1</load-on-startup>
      </servlet>
      <servlet-mapping>
            <servlet-name>example</servlet-name>
            <url-pattern>/example/*</url-pattern>
      </servlet-mapping>
</web-app>
```

Here, all requests starting with /example will be handled by the DispatcherServlet instance named example. In a Servlet 3.0+ environment, this can optional be configured programmatically:

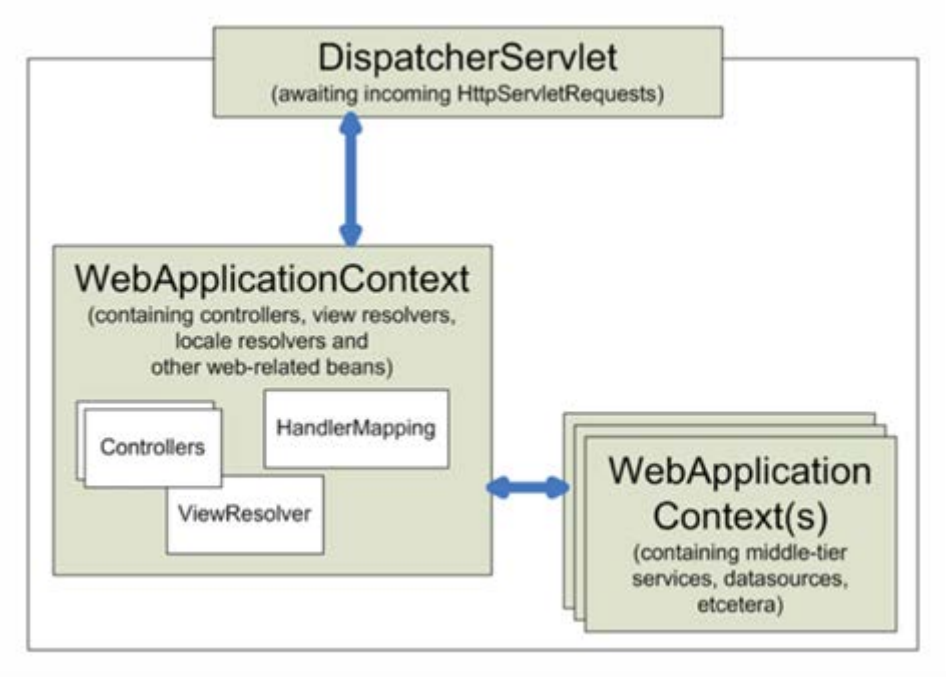
```
public class MyWebApplicationInitializer
implements WebApplicationInitializer {

    @Override
    public void onStartup(ServletContext container) {
        ServletRegistration.Dynamic registration =
   container.addServlet("dispatcher", new DispatcherServlet());
        registration.setLoadOnStartup(1);
        registration.addMapping("/example/*");
    }
}
```

WebApplicationInitializer is an interface provided that ensures code-based configuration is detected and automatically used. An abstract base class implementation of this interace named AbstractDispatcherServletInitializer makes it even easier to register the DispatcherServlet by simply specifying its servlet mapping.

After this, it is necessary to configure the various beans used by the Spring Web MVC framework.

ApplicationContext instances in Spring can be scoped. Each DispatcherServlet has its own WebApplicationContext, inheriting the beans already defined in the root context. These inherited beans can be overridden in the servlet-specific scope, and the developer can define new scope-specific beans local to a given Servlet instance:



Upon initialization of a DispatcherServlet, Spring MVC looks for a file named [servlet-name]-servlet.xml in the WEB-INF directory and creates the beans defined there, overriding the definitions of any beans defined with the same name in the global scope.

Consider the following web.xml configuration:

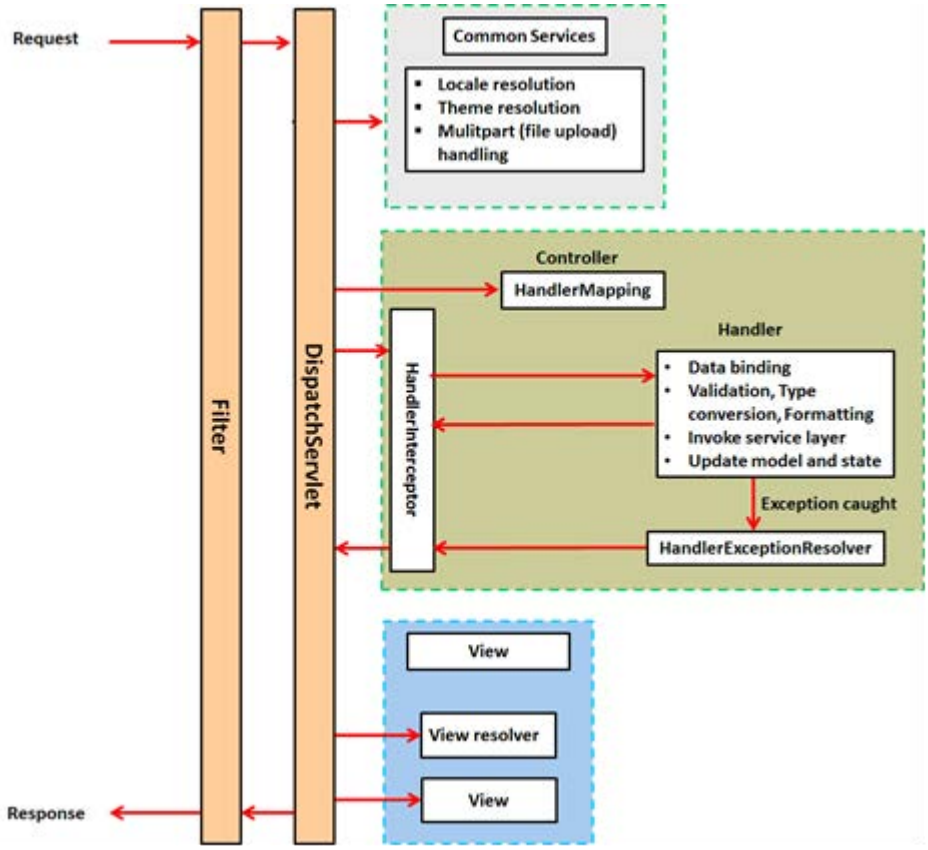
```
<web-app>
    <servlet>
        <servlet-name>sports</servlet-name>
   <servlet-class>
      org.springframework.web.servlet.DispatcherServlet
   </servlet-class>
   <load-on-startup>1</load-on-startup>
   </servlet>
   <servlet-mapping>
      <servlet-name>sports</servlet-name>
      <url-pattern>/sports/*</url-pattern>
   </servlet-mapping>
</web-app>
```

With the above Servlet configuration in place, a file called /WEB-INF/sports-servlet.xml should be in the application, containing all beans components.

The WebApplicationContext is an extension of the plain ApplicationContext that has some extra features necessary for web applications.

Spring MVC Request Lifecycle

The process of handling a request by Spring MVC involves many components, each with a specific purpose:



The processes and components involved in a Spring MVC request life cycle are:

Filter

A request filter applied to every request to direct it to the correct DispatcherServlet.

Dispatcher Servlet

A servlet evaluating incoming requests and redirecting to suitable controllers.

Common Services

Services applied to every request to support i18n, themes, views, and uploaded files. Configured in the DispatcherServlet's WebApplicationContext.

Handler Mapping

The request is mapped to the handler. From version 2.5, not required due to Spring MVC automatically registering the org.springframework.

web.servlet.mvc.annotation.DefaultAnnotationHandler mapping class to map handlers based on HTTP paths.

Handler Interceptor

Handler interceptors enable handlers to implement common checking or logic. For example, a handler interceptor can check and ensure that the handlers are invoked only during office hours.

Handler Exception Resolver

The HandlerExceptionResolverinterface is designed to deal with unexpected exceptions thrown during request processing by handlers.

View Resolver

This interface provides support for view resolution based on a logical name that is provided by the controller. Views can be dynamically resolved based on the media types supported by the client's system and this feature is supported by the ContentNegotiatingViewResolver class. The media types on the client's system could be XML, PDF, and JSON.

Building Spring MVC Applications

In order to develop web applications using Spring MVC, the following components must be used:

- Controllers - class or an interface acting as a gateway between user and application.
- View resolvers - components that define an application's view and identify them by name.
- Handler mappings - objects that handle predetermined tasks implemented before, after, or during a method implementation in the application's business logic
- Property editors - custom editors that can convert the properties of an object, such as the date and time, into human readable forms.

Controllers

Controllers provide access to the behaviour of an application, defined by a service interface. Their main purpose is to translate user requests to application code. They also convert the application's response to a form that is understandable by users and display this in the appropriate view. Spring Framework supports different types of controllers, for example, if the UI holds a simple form on the web page, then a SimpleFormController can handle incoming requests.

Controllers in Spring Framework are held in the org.springframework.web.servlet.mvc.controller interface. To access controllers:

```
public interface Controller {

   // Process request & return ModelAndView to be rendered.
   ModelAndView handleRequest(
      HttpServletRequest request,
      HttpServletReponse response)
         throws Exception;
}
```

This single method handles a request and returns the correct model and view, implementing the basic concept of MVC. Custom controllers can define any functionality in the application, such as methods initiated only by certain types of requests, data object methods that return error messages after a comparison etc.

When a request reaches the DispatcherServlet, it redirects to a controller by declaring a controller object. The controller object returns org.springframework.web.servlet.ModelAndView back to the DispatcherServlet. The model object represents the data that can be used by the view to display responses to the user. The model object is maintained as a map for storing application data. By using the org.springframework.web.servlet.View class, any kind of view technology (Excel, Jasper, PDF, HTML, Velocity) can be used:

```
// View object.
View pdfView = …

// Model object.
Map modelData = newHashMap();
```

, or:

```
ModelAndView mv1 = new ModelAndView (pdfView, modelData);
```

When specifying the view using the ModelAndView('myView', someData) method, the view is called a logical view, meaning myView can point to a JSP, PDF or XML response in the UI, as configured.

Controller Types

The following are the types of abstract controllers available in Spring Framework that can be designed to perform a specified task in the application.

Abstract Controller

Provides basic infrastructure, all of Spring Framework's Controller classes inherit from AbstractController, a class which supports caching of information needed for your application. This class allows you to implement your logic by overriding the handleRequestInterval() method and then returning a ModelAndView object:

```
protected ModelAndView handleRequestInternal(
      HttpServletRequest request,
      HttpServletReponse response) throws Exception {

   ModelAndView modelAndView = new ModelAndView("index.jsp");
   modelAndView.addObjects("User", "Greg");
   return modelAndView;
}
```

MultiActionController

Multiple actions can be logically grouped into one controller using the MultiActionController class. This class can map requests to method names and then invoke the correct method in the application:

```
public ModelAndView doPostRequest(
      HttpServletRequest request,
      HttpServletReponse response)


public ModelAndView doLogins(
      HttpServletRequest request,
      HttpServletReponse response,
      String username, String password)


public ModelAndView exceptionHandler(
      HttpServletRequest request,
      HttpServletReponse response,
      Exception exception)
```

CommandController

Provides a way to interact with the application's data objects and allows parameters to be bound to a specified data object.

Handler Mapping

These are used to map incoming requests to the correct handlers. SimpleUrlHandlerMapping or BeanNameUrlHandlerMapping can be used as defaults. The basic function of any handler mapping is to deliver a HandlerExecutionChain which contains information such as:

- Which handler to contact for a particular request
- Which handler interceptor to apply to the request

When an application receives a web request, it is redirected by the DispatcherServlet to the handler mapping. The handler mapping reviews the request and comes up with a suitable HandlerExecutionChain. The DispatcherServlet then executes the handlers and interceptors defined in the chain.

Most commonly used handler mappings are:

BeanNameUrlHandlerMapping

Maps the names of incoming HTTP requests to names of beans defined in the webapplicationcontext. Here, a form that allows users to create an account is to be submitted through a URL:

```
<bean id="/submitEnroll.html" class="com.controllers.SubmitEnroll">
   <property name="formView"
value="submitEnrollRequest>
</property>
   <property name="commandName"
value="enrollRequest">
</property>
   <property name="commandClass"
      value="com.beans.EnrollRequest">
</property>
</bean>

<bean id="handlerMapping" class=
"org.springframework.web.servlet.handler.BeanNameUrlHandlerMapping" />
```

SimpleUrlHandlerMapping

Configurable in the context, used to match handlers based on properties and request the controller to invoke. Both handlers share properties such as handler interceptors and the order of handler mapping available. Handler interceptors are interfaces that intercept a request when specified:

- Just before the controller, using the preHandle() method initiates the handler execution if set to true and stops it if set to false.
- Just after a controller, using the postHandler() method allows manipulation of the ModelAndView object before rendering the view.
- Just before the response is sent to view, using the afterCompletion() method after the view is rendered ensures cleanup of the resources used.

These handler interceptions apply specific functionalities to certain requests and can be configured at run time:

```
<bean id="urlMapping"
class="org.springframework.web.servlet.handler.
SimpleUrlHandlerMapping">
   <property name="urlmap">
      <map>
         <entry key="enrollRequest.html">
            <ref bean="submitEnrollRequest" />
         </entry>
      </map>
   </property>

   <property name="interceptors"
      <list>
         <ref bean="executeTimeInterceptor" />
      </list>
   </property>
</bean>

<bean id="executeTimeInterceptor" class="com.Interceptor">
</bean>
```

After defining handler interceptors, define and configure a URL handler for the application.

Special Bean Types in WebApplicationContext

The DispatcherServlet uses Spring MVC beans to process requests and render views. Choose which beans to use by configuring one or more of them in the WebApplicationContext, though Spring maintains a list of default beans to use if none are configured. The bean types in the WebApplicationContext:

HandlerMapping

Maps incoming requests to handlers and a list of pre- and post-processors (interceptors) based on some criteria the details of which vary by HandlerMapping implementation.

HandlerAdapter

Helps the DispatcherServlet to invoke a handler mapped to a request regardless of the handler is actually invoked.

HandlerExceptionResolver

Maps exceptions to views also allowing for more complex exception handling code.

ViewResolver

Resolves logical String-based view names to actual View types.

LocaleResolver & LocaleContextResolver

Resolves the locale a client is using and possibly time zone, enabling internationalized views

ThemeResolver

Resolves available themes, enabling personalised layouts

MultipartResolver

Parses multi-part requests enabling support for processing file uploads from HTML forms.

FlashMapManager

Stores and retrieves the "input" and the "output" FlashMap used to pass attributes between requests, usually across a redirect.

Resolving Views

Spring provides view resolvers, able to render models in a browser without tying in specific view technology. Spring enables JSPs, Velocity templates and XSLT views out of the box. The two interfaces that are important here are ViewResolver and View.

- ViewResolver provides a mapping between view names and actual views.
- View interface addresses the preparation of the request and hands the request over to one of the view technologies.

ViewResolver Interface

All handler methods in the Spring Web MVC controllers must resolve to a logical view name, either explicitly or implicitly (based on conventions). Views in Spring are addressed by a logical view name and are resolved by a view resolver. Spring comes with several view resolvers:

AbstractCachingViewResolver

Abstract view resolver that caches views. Often views need preparation before they can be used; extending this view resolver provides caching.

XmlViewResolver

Accepts a configuration file written in XML with the same DTD as Spring's XML bean factories. The default configuration file is /WEB-INF/views.xml.

ResourceBundleViewResolver

Uses bean definitions in a ResourceBundle, specified by the bundle base name. Typically you define the bundle in a properties file, located in the classpath. The default file name is views.properties.

UrlBasedViewResolver

Effects the direct resolution of logical view names to URLs, without an explicit mapping definition. This is appropriate if your logical names match the names of your view resources in a straightforward manner, without the need for arbitrary mappings.

InternalResourceViewResolver

Subclass of UrlBasedViewResolver that supports InternalResourceView and subclasses such as JstlView and TilesView.

VelocityViewResolver and **FreeMarkerViewResolver**

Subclass of UrlBasedViewResolver that supports VelocityView or FreeMarkerView respectively, and custom subclasses.

ContentNegotiatingViewResolver

Resolves a view based on the request file name or Accept header.

For example, with JSP as a view, the UrlBasedViewResolver can translate a view name to a URL and hands the request over to the RequestDispatcher to render:

```
<bean id="viewResolver"
        class=
"org.springframework.web.servlet.view.UrlBasedViewResolver">
    <property name="viewClass"
   value="org.springframework.web.servlet.view.JstlView"/>
    <property name="prefix" value="/WEB-INF/jsp/"/>
    <property name="suffix" value=".jsp"/>
</bean>
```

When returning test as a logical view name, this view resolver forwards the request to the RequestDispatcher that will send the request to /WEB-INF/jsp/test.jsp.

When combining view technologies, use the ResourceBundleViewResolver:

```
<bean id="viewResolver"
        class="org.springframework.web.servlet.view.
ResourceBundleViewResolver">
    <property name="basename" value="views"/>
    <property name="defaultParentView" value="parentView"/>
</bean>
```


A Simple Application

In this example, the application allows users to fill out a form and submit it to a server. The server returns a message to the user regarding whether the form submission was successful. The steps to build are:

1. **Configure Spring**. Configure the DispatcherServlet in the web.xml file:

```
<servlet>
   <servlet-name>Loan</servlet-name>
   <servlet-class>
      org.springframework.web.servlet.DispatcherServlet
   </servlet-class>
   <load-on-startup>1</load-on-startup>
</servlet>

<servlet-mapping>
   <servlet-name>Loan</servlet-name>
   <url-pattern>/</url-pattern>
</servlet-mapping>
```

2. **Create Loan-servlet.xml**. This holds bean definitions needed for the application:

```
<?xml version="1.0" encoding="UTF-8" ?>
<beans xmlns="http://www.springframework.org/schema/beans"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:context="http://www.springframework.org/schema/context"
  xmlns:security="http://www.springframework.org/schema/security"
  xsi:schemaLocation="http://www.springframework.org/schema/beans
   http://www.springframework.org/schema/beans/spring-beans.xsd
   http://www.springframework.org/schema/context
   http://www.springframework.org/schema/context/spring-context.xsd
   http://www.springframework.org/schema/security
   http://www.springframework.org/schema/security/spring-security.xsd">
```

3. **Add Controllers.**

First, provide a class that performs a method on the application data:

```
public class LoanRequest {

   private String forename;
   private String surname;
   ...
```

, then create a controller:

```
public ModelAndView submitEnrollRequest(
   @ModelAttribute("submitRequest") LoanRequest submitRequest,
   BindingResult result, SessionStatus status) {

   LoanRequest loanRequest = (LoanRequest) submitRequest;
   dataService.save(loanRequest);

   System.out.println("Request  ::: " + loanRequest.getFirstname()
         + "\t\t" + loanRequest.getLastname() + "\t\t"
         + loanRequest.getIncome() + "\t Address::"
         + loanRequest.getAddress() + "\t tenure:: "
         + loanRequest.getTenure());

   ModelAndView modelAndView = new ModelAndView("success");

   return modelAndView;
}
```

4. **Generate the View**. The main aspects of which are below:

```
<body>
  <form:form commandName="submitRequest"
method="POST" name="login">
    Intial Registration Form
    <form:input path="firstname"/>
      LastName :
      <form:input path="lastname" />
      Income :
      <form:input path="income" id="textbox" /></td>
      Address :
      <form:textarea path="address"
...
```


5. **Configure the Application**. Define the controller SubmitEnrollRequest as a bean.

6. **Provide Appropriate Pages**. Such as a page for successful form submission:

```
<html>
<head>
<meta http-equiv="Content-Type"
content="text/html; charset=ISO-8859-1">
<title>Success Page</title>
</head>
<body>
<h1>Successful Submitted</h1>
</body>
</html>
```

7. **Define Handler Mapping**. Match handlers based on properties and request the controller to be invoked:

```
<bean id="submitEnrollRequest"
   class="com.controllers.SubmitRequest">
   <property name="formView" value="submitEnrollRequest"></property>
   <property name="commandName" value="submitRequest"></property>
   <property name="commandClass"
   value="com.beans. LoanRequest"></property>
</bean>

<bean id="urlMapping"

class="org.springframework.web.servlet.handler.SimpleUrlHandlerMapping"
>
   <property name="urlMap">
      <map>
         <entry key="/enrollRequest.html">
            <ref bean="submitEnrollRequest" />
         </entry>
      </map>
   </property>
</bean>
```

8. **Configure Handler Interceptors**.

9. **Invoke the Controller**. A request made to enrollRequest.html will invoke the controller, as defined above.

10. **Define the View Resolver**. These help in rendering a view for the UI depending on the request parameters obtained from the user:

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

11. **Final Configuration**. Redirect the user's form submission request to the URL bean enrollRequest.html:

```
<html>
<head>
<meta http-equiv="Content-Type"
content="text/html; charset=ISO-8859-1">
<title>Insert title here</title>
</head>
<body>
   <c:redirect url="./enrollRequest.html" />
</body>
</html>
```

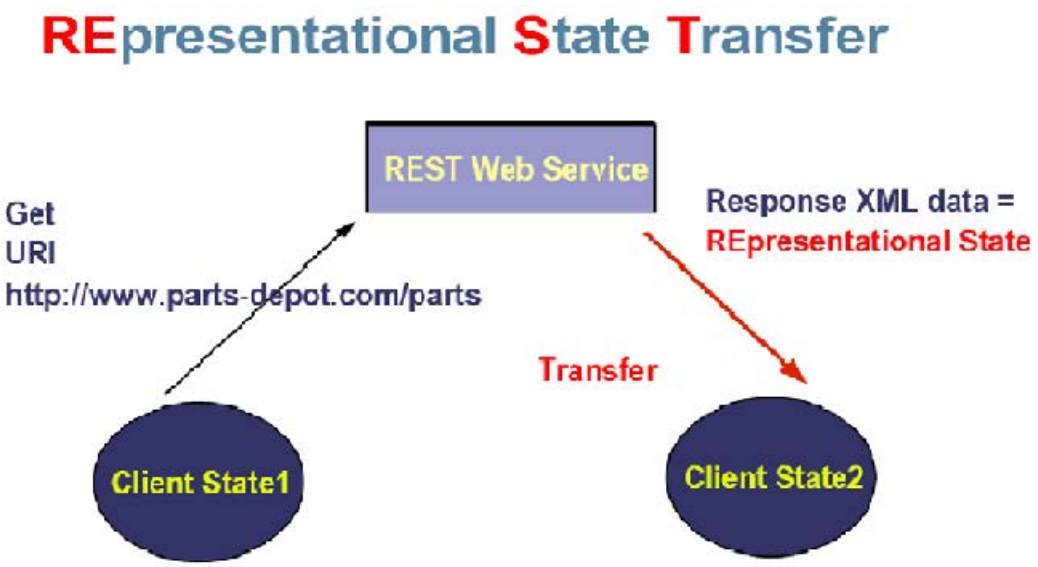
Now to deploy the application and access the URL.

CHAPTER 5

REST

Introduction

Roy Fielding coined the acronym REST in his PhD dissertation. Fielding is a principal author of the HTTP 1.1 specification and a cofounder of the Apache Software Foundation.



A few points about REST:

- RESTful services are stateless. Each request must contain all the information necessary to understand the request
- RESTful services have a uniform interface, which maps to the HTTP methods GET, POST, PUT, and DELETE
- REST architectures are built from resources identified by URIs
- Resources are manipulated through the exchange of "representations" of the resources e.g. a purchase order resource is represented by an XML document, each purchase order is made through a combination HTTP POST method with XML document, which represents the order, sent to a unique URI

REST and SOAP are very different.

- SOAP is a messaging protocol in which the messages are XML documents
- REST is a style of software architecture in which linked media is stored across a network. The World Wide Web for example.

For the web, HTTP is both a transport protocol and a messaging system. The payloads of HTTP messages can be typed using the MIME type system, with types such as text/html, application/octet-stream, and audio/mpeg3, as well as having response status codes to convey service information:

Status Code	Message	Meaning
200	OK	Request OK
303	See Other	Redirect
400	Bad Request	Request malformed
401	Unauthorized	Authentication error
403	Forbidden	Request refused
404	Not Found	Resource not found
405	Method Not Allowed	Method unsupported
415	Unsupported Media Type	Content type not allowed
500	Internal Server Error	Request processing failed

The name "REST" in interesting because the resource aspect does not occur in the acronym. A REST resource is something accessible through HTTP via a unique name, the URI (Uniform Resource Identifier). A URI has two subtypes:

- URL, specifying location
- URN, a symbolic name

URIs are uniform and structured, with a known syntax. A URI is a standardized name for a resource and acts as noun.

A resource is therefore an accessible, informational item that may be hyperlinked. They are MIME typed, whereas the web was mainly "text/html", there are now far more representations; audio, visual etc.

A RESTful request targets a resource, but the resource itself typically resides on a server, and may be persisted and maintained separately. A representation of the resource is received upon request.

A client does two things in an HTTP request:

- Names the targeted resource through a URI
- Specifies a verb, or HTTP method, to indicate an action e.g. read, create a new resource, edit, or delete a resource.

So, RESTful services involve resources to represent and client-invoked operations on resources. HTTP acts as both transport and API in this scenario, with its own verbs, or methods:

HTTP Verb	CRUD Operation
GET	Read
POST	Create
PUT	Update
DELETE	Delete

HTTP verbs are traditionally written in uppercase, but this is not required.

HTTP provides request verbs and MIME types for client requests and status codes (and MIME types) for service responses.

Browsers typically generate only GET and POST requests:

- Visiting a page generates a GET request
- Submitting a form usually generates a POST request

Although they can be used interchangeably, it's not generally the done thing. Java servlets have separate doGet and doPost methods to GET and POST requests, and although occasionally the two callbacks may execute the same code, this is also not the done thing.

Ultimate, the RESTful style should respect the original meanings of the HTTP verbs, so GET request should be idempotent because a GET is a read rather than a create, update, or delete operation. This is known as a safe GET.

Although the REST approach is simple, it does not imply that resources resource processing are such.

Services may be as complicated as SOAP, but with an attempt to simplify the service implementation by using HTTP and MIME more thoroughly.

REST as a design philosophy tries to isolate application complexity at the client and at the service. Both sides may be logic-heavy, but REST keeps the complexity out of the transport level, as a resource representation is transferred to the client as the body of an HTTP response message. The same is possible with REST as with SOAP and DOA approaches; there is no sacrifice may for simplicity.

Verbs and Nouns

In HTTP a URI is meant to be opaque, which means that the URI:

```
http://southampton/councilllors/jones
```

, has no inherent connection to the URI:

```
http://southampton/councillors
```

, although Jones happens to be a Southampton councillor. These are simply two different, independent identifiers. Good URI design will mean URIs that suggest what they are meant to identify, but that URIs have no intrinsic hierarchical structure.

URIs can and should be interpreted, but these interpretations are imposed on URIs, not inherent in them, which means getting out of the normal mindset of URIs as an abstraction of the local or remote file system.

- A URI is an opaque identifier, a logically proper name that should denote exactly one resource.

REST with JAX-RS/Jersey

REST as a technique is entirely plausible with core Servlet and JSP APIs, though this approach is convoluted and has been superceded by the JAX-RS library.

JAX-RS (Java API for XML-RESTful Services relies upon Java annotations to advertise the RESTful role that a class and its encapsulated methods play.

Jersey is the Oracle-provided reference implementation (RI) of JAX-RS used here.

Jersey Person Service

PersonApplication Class

There are several means of hosting the Application, and the javax.ws.rs.core.Application class is an easy means of packaging and providing servlet context path information, as well as application components e.g.

```
@ApplicationPath("/app")
public class RestfulPerson extends Application {
    @Override
    public Set<Class<?>> getClasses() {
      Set<Class<?>> set = new HashSet<Class<?>>();
      set.add(PersonServices.class);
      return set;
...
```

Here, the application is available at the root url "app", and consists of one service layer: PersonServices.class.

Person Class

A POJO representing a Person, drawn from an in-memory list maintained in an PersonDAO implementation.

Nothing marks this class apart, though a JAX-B annotation is provided:

```
@XmlRootElement(name="person")
public class Person implements Comparable<Person> {
...
```

The JAX-B annotation **XmlRootElement** links a Java data type such as String to an XML type, in this case xsd:string. The annotation signals that a Person object can be JAX-B transformed into an XML document with a root node of person e.g.

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<person>
   <id>1</id>
   <name>Keith Kenniff</name>
   <url>http://en.wikipedia.org/wiki/Keith_Kenniff</url>
</person>
```

{"id":1,"name":"Keith Kenniff","url":"http://en.wikipedia.org/wiki/Keith_Kenniff"}

PersonServices

A services class that exposes the application via endpoints, each provided by JAX-RS annotations. Here the service class makes use of an in-memory DAO, and is available on the path "service" e.g.

```
@Path("/service")
public class PersonServices {
   static PersonDao dao = new PersonInMemoryDao();
...
```

This class is stateless, and provides endpoints that map to HTTP verbs e.g.

```
@GET
@Produces({ MediaType.APPLICATION_JSON, MediaType.APPLICATION_XML })
public List<Person> findAll() {
   return dao.findAll();
}

@GET
@Path("{id}")
@Produces({ MediaType.APPLICATION_JSON, MediaType.APPLICATION_XML })
public Response findById(@PathParam("id") String id) {
   return Response.ok() //200
         .entity(dao.findById(Integer.parseInt(id)))
         .header("Access-Control-Allow-Origin", "*")
         .header("Access-Control-Allow-Methods",
               "GET, POST, DELETE, PUT")
         .build();
}

@POST
@Consumes({ MediaType.APPLICATION_JSON, MediaType.APPLICATION_XML})
@Produces({ MediaType.APPLICATION_XML })
public Person create(Person person) {
   return dao.create(person);
}

@DELETE @Path("{id}")
@Produces({ MediaType.APPLICATION_JSON, MediaType.APPLICATION_XML })
public void remove(@PathParam("id") int id) {
   dao.remove(id);
}
```

Here, the findById method extracts detail from the URL parameter, and makes use of the builder pattern to dynamically add request headers.

Depending on the request headers for the GET , POST and DELETE methods, either XML or JSON will be returned.

To process JSON, a JSON processor must be present. Popular processors include **GJSON**, **Jackson** and **GENSON**.

JAXB JAX-RS Example

The conversion between Java and XML is able to be automated. A Java client against a RESTful service can request a response from a service that produces an XML document, and use JAXB to unmarshal the XML document into a Java object.

The example will be to create a service from a simple class, schemagen.Product.java. This is close to the Java EE 5/6/7 example provided in the documentation, but with a twist.

```
@XmlRootElement(name="product")
@XmlAccessorType(XmlAccessType.FIELD)
public class Product {

   @XmlElement (required=true)
   protected int id;
   @XmlElement (required=true)
   protected String name;
   @XmlElement (required=true)
   protected String description;
   @XmlElement (required=true)
   protected int price;

   public Product() { }
   ...
```

Run the JAXB schema generator on the command line to generate the corresponding XML schema definition:

```
schemagen schemagen.Product
```

This command produces the XML schema as an .xsd file:

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<xs:schema version="1.0"
   xmlns:xs="http://www.w3.org/2001/XMLSchema">
   <xs:element name="product" type="product"/>
   <xs:complexType name="product">
      <xs:sequence>
         <xs:element name="id" type="xs:int"/>
         <xs:element name="name" type="xs:string"/>
         <xs:element name="description" type="xs:string"/>
         <xs:element name="price" type="xs:int"/>
      </xs:sequence>
   <xs:complexType>
</xs:schema>
```

Once you have this mapping, you can create Product objects in your application,return them, and use them as parameters in JAX-RS resource methods. The JAX-RS runtime uses JAXB to convert the XML data from the request into a Product object and to convert a Product object into XML data for the response. Continue to create a standard services class and publish a service:

```
@Path("/product")
public class ProductService {

   @GET
   @Path("/get")
   @Produces("application/xml")
   public Product getProduct() {
      Product product = new Product();
      product.setId(1);
      product.setName("Table");
      product.setDescription("Has 4 legs");
      product.setPrice(10000);
      return product;
   }
}
```

Some IDEs run the schema generator tool automatically during the build process if you add Java classes with JAXB annotations to your project.

The example makes use of the Genson library, a small, extremely simple library that can be dropped into the project, which Jersey will then autoscan and use for conversions. This library takes care of the XML-to-JSON conversion invoked when calling the createProduct method:

```
@POST
@Path("/create")
@Consumes("application/xml")
@Produces({ MediaType.APPLICATION_JSON })
public Response createProduct(Product prod) {

   // XML auto-converted to object.
   Product p = prod;
   return Response.ok(p, "application/json").build();
}
```

JAXB auto-converts the incoming XML to a Java object, and Grenson deals with the subsequent JSON conversion for response. The Grenson library is detailed here:

```
http://owlike.github.io/genson/
```


Jersey and Spring

Core Spring DI can be used to augment and simplify JAX-RS applications, where developers wish to rely on Jersey as the implementation.

Viable augmentations are:

Mock Database as Component

Core Spring DI can usefully place classes into a chosen scope as beans e.g.

```
@Component
@Scope(value=ConfigurableBeanFactory.SCOPE_SINGLETON)
public class MockDatabase {

   private int personId;
   private List<Person> list;

   public MockDatabase(){
      list = new ArrayList<Person>();
      list.add(new Person(1, "Francois Kevorkian", "http://fkurl "));
   ...
```

The mock database can then be injected into the @Service annotated DAO implementation e.g.

```
@Service("PersonDao")
@Qualifier("memdao")
public class PersonInMemoryDao implements PersonDao {

   @Autowired
   MockDatabase mockDatabase;

   public PersonInMemoryDao() {}

   public List<Person> findAll() {
        return mockDatabase.getPersonList();
   }
...
```

, and subsequently injected into the service class e.g.

```
@Path("/service")
public class PersonServices {

   @Autowired
   @Qualifier("memdao")
   private PersonDao dao;

   @GET
   @Produces({ MediaType.APPLICATION_JSON })
   public List<Person> findAll() {
      System.out.println("findAll");
      return dao.findAll();
   }
...
```

Spring Boot & Jersey

Jersey remain entirely viable in Boot applications. As long as ResourceConfig is present and Jersey is on the classpath e.g.

@Component

```
public class JerseyConfig extends ResourceConfig {

    public JerseyConfig() {
        register(RequestContextFilter.class);
        packages("com.training.*");
        register(LoggingFilter.class);
        register(CORSFilter.class);
        register(RestResource.class);

    }
}
```

Spring REST

The Spring framework supports two ways of creating RESTful services:

- The antiquated way, using MVC with ModelAndView
- Using HTTP message converters

The latter approach is lighter and easier to implement, with minimal configuration.

For modern REST development with Spring, it is wise to ignore the MVC approach.

With Spring, requests are handled by a Controller e.g.

```
@RestController
@RequestMapping("/service")
public class RestResource {

   @Autowired
   UserService userService;

   // ResponseEntity is a wrapper for the entire HTTP response,
   // which can control status codes, headers and body.
   //@CrossOrigin(origins = "http://localhost")
   @GetMapping("/service/user")
   @RequestMapping(value = "/user", method = RequestMethod.GET)
   public ResponseEntity<List<User>> listAllUsers() {
      List<User> users = userService.findAllUsers();
      if (users.isEmpty()) {
         return new ResponseEntity<List<User>>(HttpStatus.NO_CONTENT);
         // You many decide to return HttpStatus.NOT_FOUND
      }
      return new ResponseEntity<List<User>>(users, HttpStatus.OK);
   }
...
```

To explain the annotations:

@RestController

Marks the class as a controller where methods return a domain object instead of a view. The annotation is shorthand for @Controller and @ResponseBody rolled together.

@RequestMapping

At class level, ensures that HTTP requests to /service are mapped to the service class.

@ReponseEntity

This represents an HTTP response, including headers, body, and status. While @ResponseBody puts the return value into the body of the response, ResponseEntity allows the additions of headers and status codes. Effectively, a useful wrapper class.

For example:

```
@RequestMapping(value = "/user", method = RequestMethod.POST)
public ResponseEntity<Void> createUser(@RequestBody User user,
   UriComponentsBuilder ucBuilder) {
   System.out.println("Creating User " + user.getUsername());
   if (userService.isUserExist(user)) {
      System.out.println("User: " + user.getUsername()
               + " already exist");
      return new ResponseEntity<Void>(HttpStatus.CONFLICT);
   }
   userService.saveUser(user);

   HttpHeaders headers = new HttpHeaders();
   headers.setLocation(ucBuilder.path("/user/{id}").buildAndExpand(
         user.getId()).toUri());
   return new ResponseEntity<Void>(headers, HttpStatus.CREATED);
}
```

Converting to JSON is simple, as Jackson 2 is on the classpath, Spring's MappingJackson2HttpMessageConverter defaults to convert the instance to JSON.

Spring RestTemplate

A useful way to consume a REST web service is programmatically. Spring provides a template class called RestTemplate to make interacting with most RESTful services very easy.

Having created a domain class, such as the previous Person POJO, a client application can interact quite easily e.g.

```
public static void main(String args[]) {
   RestTemplate restTemplate = new RestTemplate();
   Person person = restTemplate.getForObject(
         "http://localhost:8080/app/service/1", Person.class);
   System.out.println(person.toString());
}
```

As the Jackson library is in the classpath, RestTemplate uses it to convert incoming JSON into a Person.

RestTemplate supports all HTTP verbs.

RestTemplate defines many methods, though there are eleven main methods e.g.

Method	Description
delete()	Performs an HTTP DELETE request on a resource at a specified URL
exchange()	Executes a specified HTTP method against a URL, returning a ResponseEntity containing an object mapped from the response body
execute()	Executes a specified HTTP method against a URL, returning an object mapped from the response body
getForEntity()	Sends an HTTP GET request, returning a ResponseEntity containing an object mapped from the response body
getForObject()	Sends an HTTP GET request, returning an object mapped from a response body
headForHeaders()	Sends an HTTP HEAD request, returning the HTTP headers for the specified resource URL
optionsForAllow()	Sends an HTTP OPTIONS request, returning the Allow header for the specified URL
postForEntity()	POSTs data to a URL, returning a ResponseEntity containing an object mapped from the response body
postForLocation()	POSTs data to a URL, returning the URL of the newly created resource
postForObject()	POSTs data to a URL, returning an object mapped from the response body
put()	PUTs resource data to the specified URL

REST Tooling

Using curl

The curl utility can be used to make requests to services:

```
curl -v http://localhost:8080/service/user
```

The response includes a trace of the HTTP request and response messages. The HTTP request is:

```
* Adding handle: conn: 0x50e5050
* Adding handle: send: 0
* Adding handle: recv: 0
* Curl_addHandleToPipeline: length: 1
* - Conn 0 (0x50e5050) send_pipe: 1, recv_pipe: 0
* About to connect() to localhost port 8080 (#0)
*   Trying 127.0.0.1...
* Connected to localhost (127.0.0.1) port 8080 (#0)
> GET /service/user HTTP/1.1
> User-Agent: curl/7.33.0
> Host: localhost:8080
> Accept: */*
>
< HTTP/1.1 200
< Content-Type: application/json;charset=UTF-8
< Transfer-Encoding: chunked
< Date: Wed, 28 Mar 2018 15:10:10 GMT
<
[{"id":1,"username":"Greg","address":"London","email":"greg@abc.com"},{
"id":2,"username":"John","address":"Southampton","email":"john@abc.com"
},{"id":3,"username":"Julia","address":"Weymouth","email":"julia@abc.co
m"}]* Connection #0 to host localhost left intact
```

As HTTP GET has no body, the entire message is the start line and the headers. The response shows the generated JSESSSIONID session identifier (a 128-bit unique number, in hex) in the header.

In the event of a POST request:

```
c:\curl -v -X POST -H "Content-Type: application/json" -d
"{\"id\":\"45\",\"username\":\"Bath\",\"email\":\"me@me.com\",\"address
\":\"Unknown\"}" http://localhost:8080/service/user
```

The request header:

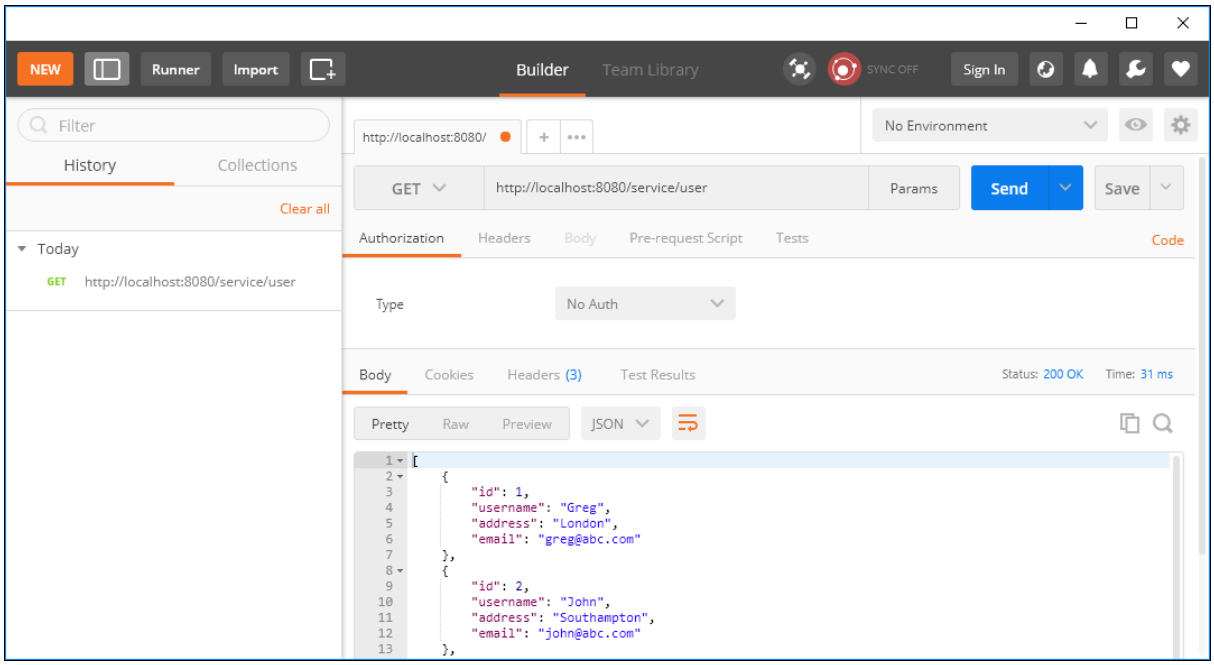
```
* Adding handle: conn: 0x5214240
* Adding handle: send: 0
* Adding handle: recv: 0
* Curl_addHandleToPipeline: length: 1
* - Conn 0 (0x5214240) send_pipe: 1, recv_pipe: 0
* About to connect() to localhost port 8080 (#0)
*   Trying 127.0.0.1...
* Connected to localhost (127.0.0.1) port 8080 (#0)
> POST /service/user HTTP/1.1
> User-Agent: curl/7.33.0
> Host: localhost:8080
> Accept: */*
> Content-Type: application/json
> Content-Length: 69
```

The response header:

```
* upload completely sent off: 69 out of 69 bytes
< HTTP/1.1 201
< Location: http://localhost:8080/user/4
< Content-Length: 0
< Date: Wed, 28 Mar 2018 15:23:59 GMT
<
* Connection #0 to host localhost left intact
```

Using Postman

Postman is probably the most thorough and configurable of the browser plugin tools, which has now moved over to a desktop application, though remains available as an extension for the time being.



URL

When entering the request URL, previously-used URLs will show an autocomplete dropdown. The Params button is used to add key-value pairs.

Headers

The Headers tab shows the headers key-value editor. Set any string as the header name, and the autocomplete dropdown provides suggestions of common headers.

Cookies

Manage Cookies; click the Cookies link under the Send button.

Request Body

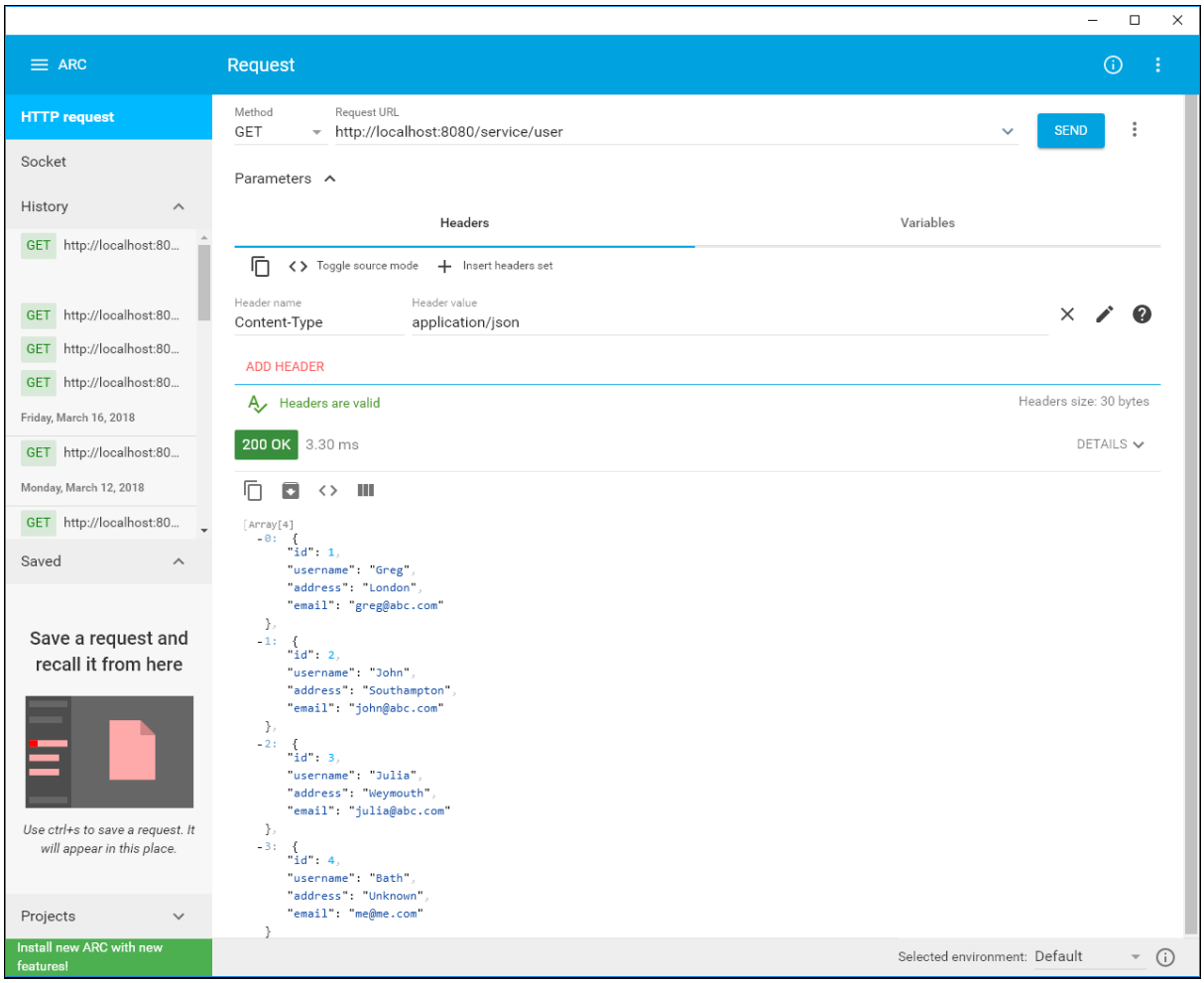
Postman allows almost any kind of HTTP request. The body editor is divided into 4 areas and has different controls, depending on the body type.

Full Postman details are available here: https://www.getpostman.com/

Using ARC

The Advanced REST Client, or ARC, still remains a useful Chrome Extension.

The application is available from the Chrome Store in the usual way.



The extension operates similarly to Postman, and is fairly intuitive.

CHAPTER 6

WebSocket

Introduction

The web has moved from request-response, through AJAX, towards full-duplex communications with WebSockets.

With the death of Flash, a solution that offered low-latency without the overhead of HTTP's cookies and headers was needed, and WebSocket, standardised by the W3C and supported by most major browsers, allows ongoing conversations initiated by client or server.

JavaScript, being browser-embedded, is a likely candidate for the client, and there are multiple implementations. Among them are:

µWS

WebSocket and HTTP implementation for clients and servers. Simple, efficient and lightweight.

Socket.IO

A long polling/WebSocket based third party transfer protocol for Node.js.

Faye

WebSocket and EventSource implementation for Node.js Server and Client.

A WebSocket server is a TCP application listening on any port of a server that follows a specific protocol. It is fairly straightforward to implement a WebSocket server on multiple platforms.

Any programming language capable of Berkeley sockets is viable for the server. The focus here, and with Spring, is on Java.

Prior to investigating Spring WebSocket support, it is wise to cover the basics of both client, server, and the protocol itself.

WebSocket Handshake

Clients establish a WebSocket connection through an initial handshake, where the client initiates a normal HTTP request to the server, with an Upgrade header included to let the server know that the client wishes to establish a WebSocket connection.

The client handshake is similar to:

```
GET ws://websocket.training.com/ HTTP/1.1
Origin: http://training.com
Connection: Upgrade
Host: websocket.training.com
Upgrade: websocket
```

Note

- The method must be GET
- Common headers such as User-Agent, Cookie etc may also be present
- There may be a Sec-WebSocket-Version header with version detail
- The Origin header may be used for security
- Post-handshake HTTP status codes are no longer viable

Notice the ws scheme, which also has an equivalent wss for secure connections, similar to HTTPS.

A server that supports the protocol responds with a corresponding HTTP response, with an Upgrade header:

```
HTTP/1.1 101 WebSocket Protocol Handshake
Date: Mon, 12 Mar 2018 12:12:34 GMT
Connection: Upgrade
Upgrade: WebSocket
```

Handshake complete, the initial HTTP connection is replaced by a WebSocket connection using the same underlying TCP/IP connection. Either partt may now initiate communcation.

Data is transferred as messages consisting of one or more frames, with far less overhead than HTTP.

Message reconstruction is guaranteed via frame prefixes, where metadata describes the payload.

For example, the WebSocket API is present in VanillaJS, where a socket may be opened (or the handshake initiaited) like so:

```
// Create a new WebSocket.
var socket = new WebSocket('wss://echo.websocket.org/');
```

Exchanging Data

Either side may initiate communication post-handshake. The data, or frames of data, is masked using 32-bit keyed XOR encryption.

The details of messaging, frames, decoding, reading the data etc can be found at W3C:

https://developer.mozilla.org/en-US/docs/Web/API/WebSockets_API/Writing_WebSocket_servers#Exchanging_Data_Frames

, and in the specification:

https://tools.ietf.org/html/rfc6455

The maintainance of the connection is enabled with constant heartbeats. Either client or server may ping, and the recipient must pong as soon as possible.

The ping (and pong) is a control frame: pings with the opcode 0x9, and pongs opcode 0xA.

A VanillaJS client would send data quite simply:

```
form.onsubmit = (e) => {
  e.preventDefault();
  // Retrieve message from textarea.
  var message = messageField.value;

  // Send message through WebSocket.
  socket.send(message);

  messagesList.innerHTML += '<li class="sent"><span>Sent:</span>'
                         + message + '</li>';
  messageField.value = '';
  return false;
};
```

Closing Connections

Client or server sends a control frame with data containing a control sequence to begin the closing handshake. Upon receipt, a Close frame is sent in response. The first peer then closes the connection. Any further data is discarded/ignored.

Either side may close; in a VanillaJS client:

```
closeBtn.onclick = (e) => {
  e.preventDefault();

  // Close WebSocket.
  socket.close();

  return false;
};
```

Simple Java WebSocket Server

JSR 356 or the Java API for WebSocket, specifies an API that covers both server side and Java client side.

Java has built in WebSocket support via javax.websocket package.

A simple application initially includes the dependency:

```
<dependency>
   <groupId>javax.websocket</groupId>
   <artifactId>javax.websocket-api</artifactId>
   <version>1.1</version>
   <scope>provided</scope>
</dependency>
```

Endpoints are often annotation-based, though may be extension-based via the javax.websocket.Endpoint class.

Annotation are cleaner and more commonly used.

Endpoint lifecycle events are handled by the following annotations:

@ServerEndpoint

Container ensures availability of the class as a WebSocket server listening to a specific URI.

@ClientEndpoint

Class treated as a WebSocket client.

@OnOpen

Method invoked by container when a new WebSocket connection is initiated.

@OnMessage

Method receives information from the WebSocket container when a message is sent to the endpoint.

@OnError

Method invoked when there are communication errors.

@OnClose

Method called by the container when the WebSocket connection closes.

The following code shows a simple echo server, using classes imported from the core libarary javax.websocket:


```
/**
 * @ServerEndpoint is the relative name for the endpoint
 * accessed via ws://localhost:8080/example/echo
 * "localhost" : host address,
 * " example" : web context root,
 * "echo" : server endpoint.
 */
@ServerEndpoint("/echo")
public class EchoServer {

    /**
     * @OnOpen intercepts the creation of a new session.
     * The session class allows sending of data to the user.
     * onOpen : used to let user know that handshake was successful.
     */
    @OnOpen
    public void onOpen(Session session){
        System.out.println(session.getId() +
                    " has opened a connection");
        try {
            session.getBasicRemote().sendText(
                    "Connection Established");
        } catch (IOException ex) {
            ex.printStackTrace();
        }
    }

   /**
    * OnMessage intercepts the message from the server,
    * Here, the message is read as a String.
    */
   @OnMessage
   public void onMessage(String message, Session session){
       System.out.println("Message from " + session.getId()
                   + ": " + message);
      System.out.println("Message Received: " + message);
      for (Session s : session.getOpenSessions()) {
         if (s.isOpen())
            s.getAsyncRemote().sendText(message);
      }

   }

   /**
    * The user closes the connection.
    * Messages cannot be sent to the client with this method.
    */
   @OnClose
   public void onClose(Session session){
       System.out.println("Session " + session.getId()
                + " has ended");
   }
}
```

Message Types

WebSocket supports both text and binary formats. The Java API supports these and adds capabilities to work with Java objects and health check messages (ping-pong) as defined in the specification:

Text

Textual data (java.lang.String, primitives or wrapper classes).

Binary

Binary data as a java.nio.ByteBuffer or byte[].

Java objects

Native Java objects are possible via custom transformers (encoders/decoders) to convert them into compatible formats (text, binary) as specified by the protocol.

Ping-Pong

A javax.websocket.PongMessage is an acknowledgment sent by a WebSocket peer in response to a health check (ping) request.

Encoder

Takes a Java object and provides a representation suitable for the WebSocket specification message (JSON, XML or binary), by implementing Encoder.Text<T> or Encoder.Binary<T> interfaces.

Here, a simple Message class is encoded using the Google Gson library:

```
public class MessageEncoder implements Encoder.Text<Message> {

   private static Gson gson = new Gson();

   @Override
   public String encode(Message message) throws EncodeException {
       return gson.toJson(message);
   }
...
```

Decoder

Here, data is transformed back to a Java object, using Decoder.Text<T> or Decoder.Binary<T> interfaces:

```
public class MessageDecoder implements Decoder.Text<Message> {
    private static Gson gson = new Gson();

    @Override
    public Message decode(String s) throws DecodeException {
        return gson.fromJson(s, Message.class);
    }

    @Override
    public boolean willDecode(String s) {
        return (s != null);
    }
...
```

Encoder and Decoder Usage

Encoding and decoding classes are added to the class annotation @ServerEndpoint:

```
@ServerEndpoint(
  value="/chat/{username}",
  decoders = MessageDecoder.class,
  encoders = MessageEncoder.class )
```

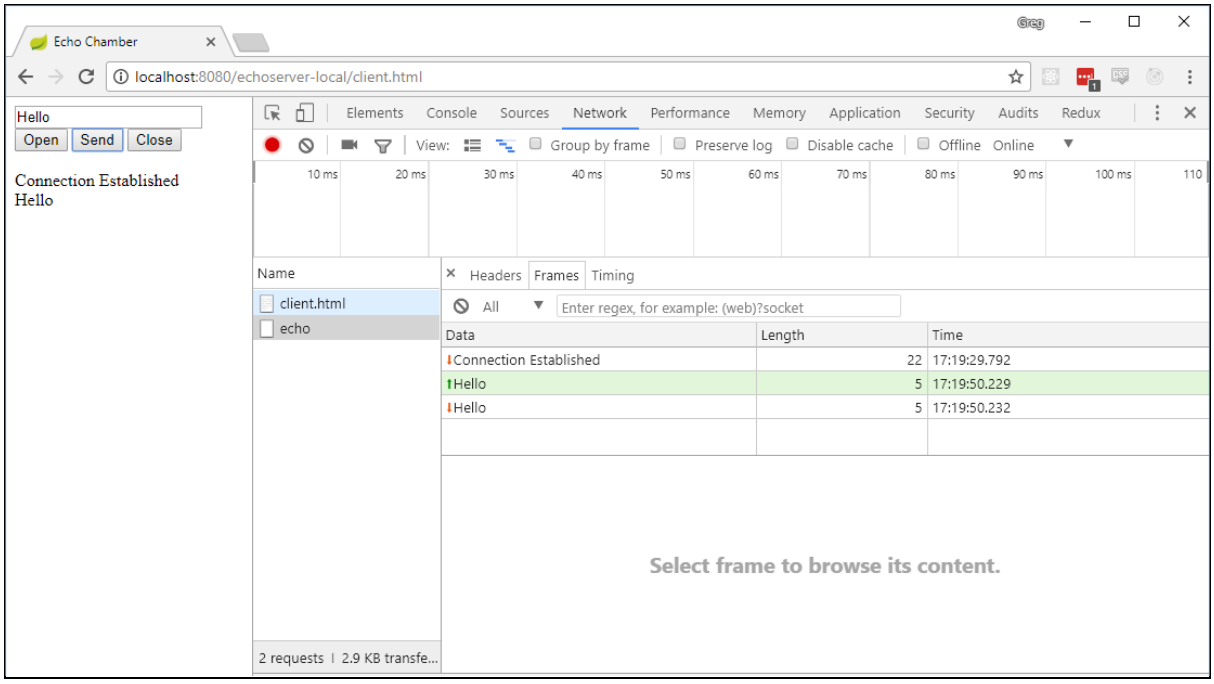
Each message is then converted to JSON or Java.

WebSocket Monitoring

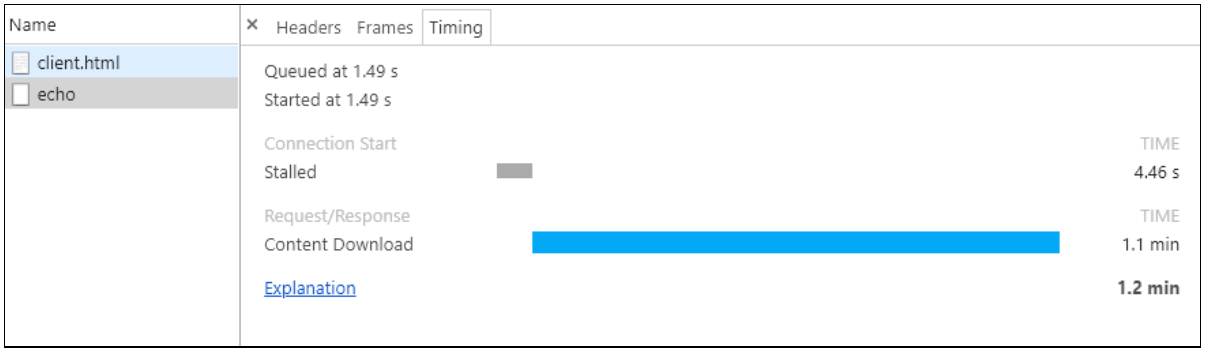
Chrome Developer Tools is able to monitor WebSocket traffic.

- Open Developer Tools (F12)
- Go to the Network tab
- Click on the WebSocket connection (echo below)
- Open the Frames tab

A summary of data sent through the connection is available.



The Timing sub-tab displays the open connection, and when closed, displays a summary of communications:



Spring WebSocket

Spring has support for WebSockets both with the Servlet amd Reactive stacks.

WebSocket API

Spring provideas a WebSocket API for messagings, similar to the core Java provision.

Create a WebSocket server by implementing WebSocketHandler or extending either TextWebSocketHandler or BinaryWebSocketHandler:

```
import org.springframework.web.socket.WebSocketHandler;
import org.springframework.web.socket.WebSocketSession;
import org.springframework.web.socket.TextMessage;

public class MyHandler extends TextWebSocketHandler {

    @Override
    public void handleTextMessage(
            WebSocketSession session, TextMessage message) {
...
```

WebSocket handlers are mapped using configuration via Java-config e.g

```
import org.springframework.web.socket.config.
annotation.EnableWebSocket;
import org.springframework.web.socket.config.annotation.
WebSocketConfigurer;
import org.springframework.web.socket.config.annotation.
WebSocketHandlerRegistry;

@Configuration
@EnableWebSocket
public class WebSocketConfig implements WebSocketConfigurer {

    @Override
    public void registerWebSocketHandlers(
            WebSocketHandlerRegistry registry) {
        registry.addHandler(myHandler(), "/myHandler");
    }

    @Bean
    public WebSocketHandler myHandler() {
        return new MyHandler();
    }
}
```

, or XML configuration:

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

    <websocket:handlers>
        <websocket:mapping path="/myHandler" handler="myHandler"/>
    </websocket:handlers>

    <bean id="myHandler"
         class="org.springframework.samples.MyHandler"/>

</beans>
```

WebSocket Handshake

The handshake can be customised using a HandshakeInterceptor, with before and after implementations:

```
@Override
public void registerWebSocketHandlers(WebSocketHandlerRegistry
registry) {
    registry.addHandler(new MyHandler(), "/myHandler")
        .addInterceptors(new HttpSessionHandshakeInterceptor());
}
```

Server and Deployment

The API is able to integrate into a Spring MVC application, and can be also be used outside of Spring MVC using WebSocketHttpRequestHandler.

WebSocket clients and servers can negotiate the use of a higher-level, messaging protocol, such as STOMP, via the "Sec-WebSocket-Protocol" header on the HTTP handshake request, or via a custom route. Many major containers support this, and expose configuration often via a bean, in order to tailor the WebSocket runtime e.g.

```
@Bean
public ServletServerContainerFactoryBean createWebSocketContainer() {
   ServletServerContainerFactoryBean container =
         new ServletServerContainerFactoryBean();
   container.setMaxTextMessageBufferSize(32768);
   container.setMaxBinaryMessageBufferSize(32768);
   return container;
}
```

SockJS

Over the Internet, restrictive proxies may render WebSocket interactions unusable. The Upgrade header may not be allowed, or persistent connection may be closed.

Here, WebSocket emulation; that is a WebSocket attempt, before falling back to HTTP-based WebSocket emulation, is the only option.

Spring provides support for the SockJS protocol.

SockJS consists of:

- The SockJS protocol
- The SockJS JavaScript browser-based client
- SockJS server implementation (spring-websocket)
- A SockJS Java client

To configure:

```
@Configuration
@EnableWebSocket
public class WebSocketConfig implements WebSocketConfigurer {

    @Override
    public void registerWebSocketHandlers(
            WebSocketHandlerRegistry registry) {
        registry.addHandler(myHandler(), "/myHandler").withSockJS();
    }
...
```

Full details of Spring support can be found here:
https://docs.spring.io/spring/docs/current/spring-framework-reference/web.html#websocket-fallback

SockJS lives on GitHub: https://github.com/sockjs

Stomp

WebSocket defines text and binary messages, while content of these messages has no definition. Clients and server can negotiate a sub-protocol to add definition here (content, format etc).

STOMP is a simple, text-oriented messaging protocol originally for scripting languages to connect to message brokers, that can handle both text or binary payloads.

Using STOMP means a more capable model than the low-level WebSockets, much like HTTP layering over TCP.

- STOMP clients are available in Java and Spring.
- Message brokers can be easily used
- Spring Security can be used with STOMP

Support is in spring-messaging and spring-websocket modules. With dependencies added , expose a STOMP endpoints with SockJS fallback e.g

```
@Override
public void registerStompEndpoints(StompEndpointRegistry registry) {
    registry.addEndpoint("/portfolio").withSockJS();
}

@Override
public void configureMessageBroker(MessageBrokerRegistry config) {
    config.setApplicationDestinationPrefixes("/app");
    config.enableSimpleBroker("/topic", "/queue");
}
```

In the browser, a sockjs-client such as webstomp-client can be used:

```
var socket = new SockJS("/spring-websocket-example/test");
var stompClient = webstomp.over(socket);

stompClient.connect({}, function(frame) {
}
```

Full STOMP details for Spring are available here:
https://docs.spring.io/spring/docs/current/spring-framework-reference/web.html#websocket-stomp

Details can be found here: http://jmesnil.net/stomp-websocket/doc/ , and although content is thorough, support has not been maintained.

Spring Boot Support

Spring Boot adds WebSocket and STOMP configuration using annotations:

```
@Configuration
@EnableWebSocketMessageBroker
public class WebSocketConfig implements
WebSocketMessageBrokerConfigurer {

   @Override
   public void registerStompEndpoints(StompEndpointRegistry registry) {
      registry.addEndpoint("/socket")
              .setAllowedOrigins("*")
              .withSockJS();
  }

  @Override
  public void configureMessageBroker(MessageBrokerRegistry registry) {
      registry.setApplicationDestinationPrefixes("/app")
              .enableSimpleBroker("/chat");
  }
}
```

- @Configuration indicates a Spring configuration class.
- @EnableWebSocketMessageBroker enables WebSocket message handling, backed by a message broker

Endpoints and configuration is as per regular Spring.

Controller support also comes via annotations.

```
@MessageMapping("/hello")
@SendTo("/topic/greetings")
public Greeting greeting(HelloMessage message) throws Exception {
    return new Greeting("Hello, " + message.getName() + "!");
}
```

- @MessageMapping maps the method to the destination, with message payload bound to a HelloMessage object.

Browser support comes via the standard sockjs and stomp libraries, which are available via WebJars:

```
<script src="/webjars/sockjs-client/sockjs.min.js"></script>
<script src="/webjars/stomp-websocket/stomp.min.js"></script>
```

CHAPTER 7

JDBC

Introduction

At some point an application must store its data externally. During the process of saving and retrieving data to the external store the application must not damage the data in that store.

This is made more straightforward by using a data access framework which abstracts the tasks of communicating with the data store, allowing the application to focus on the business logic. Spring provides a framework which supports other data access frameworks such as JDBC, Hibernate, JDO and JPA.

This section focuses on using Spring to retrieve and manipulate data using JDBC and examines the following topics.

- Principles of Spring JDBC
- Using Spring JDBC
- The Spring JdbcTemplate
- Utilising stored procedures
- Handling exceptions
- Using annotations

The majority of Java applications use relational databases to store data. Spring enables applications to perform queries and transactions using JDBC data sources. Spring helps developers to do this by supplying a data access component which abstracts data management away from the application.

Spring supplies the following JDBC support features:

- Connection handling
- Exception handling
- Transaction management
- Resource unwrapping

Principles of Spring JDBC

The API used by Java developers to work with data sources is known as JDBC. The JDBC technology doesn't specify that the data source will be a relational database but this is the most common type. As you would expect, JDBC enables clients, such as JEE web components and EJBs, to connect to the database, send SQL queries through to the database and extract data from the database.

The JDBC API is contained in the JSE packages **java.sql** and **javax.sql**, which consists mainly of interfaces. This is because JDBC abstracts common database tasks into a set of abstract methods for use polymorphically by developers to work with any DBMS that has implemented the JDBC abstraction. This database-specific implementation (often provided as a zip or JAR file), is known as a driver.

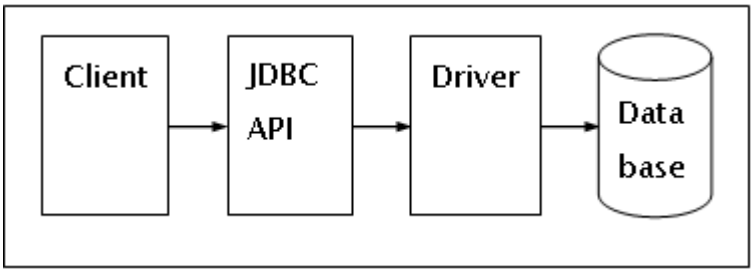
The first step when working with a database, therefore, is to locate the driver and use it to make a connection to the database.

Connecting to Databases

To obtain a connection, the client does not need to open any sockets because the connection details are handled by the driver. Once loaded into memory, the driver is managed by a class within the JVM called **DriverManager**, which works with the driver to create a connection for the client to use. Note that if connections are pooled in an application server's namespace, a naming service is used instead of the **DriverManager** to lookup the connection from the pool (for details on naming services, see the JNDI module in this manual).

The steps involved in the connection process are as follows:

- The driver is loaded into memory.
- The client tells the **DriverManager** to use the driver software to return a connection to the database. Alternatively, the client uses JNDI to lookup a connection from the pool.
- Once connected, the client uses the JDBC API (via the driver's implementation of same) to send SQL queries and extract data.



Loading the Driver

The **DriverManager** class is responsible for managing the drivers and handing out connections to client code. To carry out these tasks it needs to know which drivers are to be loaded and which database the client wants to connect to.

There are two ways to load the driver class. The first technique involves the use of the class **Class** to retrieve the **Class** object:

```
Class.forName("drivername");
```

Here, the driver registers itself with the DriverManager through the JVM, which creates the driver instance.

The second technique involves setting a system property:

```
System.setProperty("jdbc.drivers", "drivername");
```

Here, the **DriverManager** searches the **jdbc.drivers** system property and loads the drivers listed. Using this technique, multiple drivers can be loaded at the same time, each one separated by a colon.

Although both samples use a string literal for the driver name, in the real world this value would be stored in a variable to avoid having to recompile the code when the client wanted to change drivers. The variable's value would then be obtained from an external source, such as the command line or a configuration file.

Note that the driver software does not need to be present at compile time - it merely has to be available on the classpath at run time.

Making the Connection

To connect to the database, the client uses a static method in the DriverManager class called 'getConnection' to obtain from the driver an object that implements the Connection interface. This method has a String parameter (known as a JDBC url), which the DriverManager uses to identify the database that the client wishes to connect to.

The JDBC url has the following structure:

```
<protocol>:<subprotocol>:<subname>
```

As an example of a JDBC url, the following shows a Connection object retrieved from a url that can be used to connect to a Cloudscape database:

```
Connection con = DriverManager.getConnection(
   "jdbc:cloudscape:rmi://localhost:1099/cloudscapeDB");
```

- The protocol is always set to jdbc.
- The subprotocol identifies the vendor of the driver.
- The subname identifies the data source and may include additional information needed to make the connection, in our case, the IP address and port number used by the database server. The rmi protocol allows us to communicate with another Java object (the database server) within a separate JVM and port 1099 is the standard RMI port used by Cloudscape.

Consult the documentation that comes with your driver to find what url identifiers to use with that driver.

Statements

Once connected to a database, the next stage is to do something with the table(s) contained within it. JDBC abstracts the task of sending SQL queries to retrieve, update, insert and delete data into an interface called 'Statement', which can be found in 'java.sql'.

Executing a Statement

A Statement object enables the client to perform basic SQL statements where the SQL is prepared, checked by the DBMS and, assuming the SQL is syntactically correct, executed in one step. The Statement object is created using the 'createStatement' method in the Connection interface:

```
Statement stmt = con.createStatement();
```

Once obtained, the Statement object can be used to send SQL queries through to the database using one of four methods:

Method	Description
ResultSet executeQuery(String sql)	For 'SELECT' sql queries. Returns a ResultSet object.
int executeUpdate(String sql)	For 'CREATE', 'UPDATE', 'INSERT' and 'DELETE' sql commands. Returns the number of rows affected.
int[ ] executeBatch( )	Sending multiple sql commands in one go. Returns the number of rows affected as an int array.
boolean execute(String sql)	For unknown sql commands. Returns true for a ResultSet.

For example, the following code queries all the records in a table:

```
Resultset result = stmt.executeQuery
      ("Select * From Employees");
```

Executing a Batch of Statements

Performance of JDBC clients can be improved by executing multiple SQL commands to the Statement object in a single batch operation. The commands are stored in memory until the client tells the Statement to send them to the DBMS for execution:

```
stmt.addBatch(sqlCreateString);
stmt.addBatch(sqlInsertString);
stmt.addBatch(sqlUpdateString);
stmt.addBatch(sqlDeleteString);
// Send the batch to be executed.
int[] results = stmt.executeBatch();
```

ResultSets

The results of a 'SELECT' query are returned in an object that implements the 'ResultSet' interface. This object consists of rows and columns of the data requested and its interface provides methods to allow the client to use the data returned.

The basic ResultSet object returned by the empty 'executeQuery' method seen earlier only allows the client to move from the first row to the last. It doesn't allow random or backward movement. As a result, the only method available for moving around in the basic ResultSet is:

```
boolean next( )   // Returns false at the end of the ResultSet
```

When the ResultSet is first returned, its cursor is positioned immediately prior to the first row of data, so the 'next' method places you on the first row the first time the method is called.

Scrolling Through ResultSets

To navigate fully through a ResultSet, you need to use an overload of the 'executeQuery' method. This version takes two integer arguments that set the type and updateability of the ResultSet. The first refers to whether backward scrolling is permissible and it accepts one of three arguments defined as static fields in the ResultSet interface:

- ResultSet.TYPE_SCROLL_SENSITIVE
- ResultSet.TYPE_SCROLL_INSENSITIVE
- ResultSet.TYPE_FORWARD_ONLY

The first two create a fully scrollable ResultSet. If changes occur to the database from another client while you are scrolling, the first type indicates that you will see those changes, whilst the second means that you won't.

The third type is the default type already discussed. A scrollable ResultSet makes the following navigational methods available:

Method	Description
boolean next( )	Returns false at the end of the ResultSet
boolean previous( )	Returns false at the start of the ResultSet
boolean last( )	Moves to the last row. Returns false if there are no rows
boolean first( )	Moves to the first row. Returns false if there are no rows
boolean isFirst( )	Returns true if cursor is on the first row
boolean isLast( )	Returns true if cursor is on the last row
boolean absolute(int row)	Moves cursor to the given row
boolean relative(int row)	Moves cursor a relative number of rows from the current position

Many of these methods can often be seen in while loops and conditional statements as they return false to stop the loop at the end or beginning of the ResultSet and they enable decisions to be made if the cursor is at a particular row.

Retrieving Data from ResultSets

The ResultSet interface supplies a large number of getX methods designed to allow you to read the data in a ResultSet by referencing the relevant column by number or name. There is a pair of overloaded getX methods for each Java primitive and for several objects.

For example, if you know that a column value is of the integer type, you can use the 'getInt' method, passing in either a string for the column label or a number (must be greater than 0 as database columns start at 1) for the column index.

The following code sample shows how to navigate through and read from a ResultSet using column labels:

```
Statement stmt = con.createStatement(
   ResultSet.TYPE_SCROLL_INSENSITIVE,
   ResultSet.CONCUR_READ_ONLY);

ResultSet result = stmt.executeQuery(
      "SELECT * FROM Employees");

while(result.next()) {
   StringBuffer rowText = new StringBuffer("");
   rowText.append(result.getString("NAME") + "\t");
   rowText.append(result.getInt("ID") + "\t");
   rowText.append(result.getDouble("SALARY") + "\t");
   System.out.println(rowText.toString());
}
```

Releasing Database Resources

All the interfaces discussed contain a close method to release their objects. If the client fails to call close, then connections, statements and result sets (and the database resources they hold onto) may be tied up indefinitely because the garbage collector is not guaranteed to release them. So the first rule of thumb is to call 'close' on a JDBC object when it is no longer required.

In addition, we need to ensure that the call to close will be executed in the event of an exception being thrown. The second rule, therefore, is to place the call to close in a finally block to ensure execution. This could be done like so:

```
finally {
   try {
      result.close();
      stmt.close();
      con.close();
   }
   catch(SQLException e) { }
}
```

However, if an exception was thrown when the result set or statement was closing the connection would remain open. Therefore, close each object in a separate method, which in turn should contain its own try ... catch block, so any thrown exception would not prevent subsequent objects being released.

Using Spring JDBC

The Spring framework provides mechanisms to allow developers to overcome the limitations of using JDBC directly in an application. The most important of these are detailed below.

- Improved connection handling

This can prevent open connections to databases. The database connection should be closed once the results of a query have been returned. However the close() methods associated with JDBC connection objects may throw exceptions, and this may connections to stay open and waste resources.

Spring can handle the low-level details related to handling connections, including opening and closing connections, handling exceptions and executing SQL statements managing transactions.

- More detailed exception classes

JDBC uses SQLException almost exclusively. This encapsulates database vendor-specific errors and makes it difficult to develop database-agnostic error handling.

- Dependency Injection of connection details

Developers often have to develop separate code to fulfill the lookup requirements of different DataSource objects, each of which deals with access to a different database. These data sources must be identified using a naming service such as JNDI, which results in more code.

Spring allows database connection details to be defined as Spring beans, allowing them to be injected into the application where required.

- Usage of data access objects

Spring uses Data Access Objects (DAOs) to read data from and write data to a database. These are stored in an interface so that the rest of the application has access to them. This decouples data access from the service layer implementation, allowing easier testing and a choice of persistence framework.

Spring JDBC Packages

Spring provides a thin, robust, and highly extensible JDBC abstraction framework which takes care of all the low-level details, such as establishing connections, preparing the statement object, executing the query, and releasing the database resources. While using it for data access, the developer needs to specify the SQL statement for executing and retrieving the result.

Spring JDBC provides the following packages:

Package Name	Description
org.springframework.jdbc.core	This package contains the core JDBC classes and the JdbcTemplate class. It simplifies the database operation using JDBC.
org.springframework.jdbc.datasource	This package contains DataSource implementations and helper classes, which can be used to run the JDBC code outside the JEE container.
org.springframework.jdbc.object	This package contains the classes that help in converting the data returned from the database into plain Java objects.
org.springframework.jdbc.support	This package contains several classes including SQLExceptionTranslator, which recognizes the error code used by the database. It achieves this by mapping the error code to a higher level of exception.
org.springframework.jdbc.config	This package contains the classes that support JDBC configuration within ApplicationContext of the Spring Framework.

The DriverManagerDataSource Class

The DriverManagerDataSource class can configure the DataSource for application. It has the following characteristics:

- It is defined in the spring.xml configuration file
- It is supplied as part of the spring-jdbc-5.x.x.RELEASE.jar file

The configuration of DriverManagerDataSource is shown here. We need to provide the driver class name and the connection URL. We can also add the username and the password in the property if the database requires it.

The code below shows the configuration of the DriverManagerDataSource class in the spring.xml file. To connect, we need to supply the driver class name and JDBC connection URL.

```
<context:annotation-config />
<context:component-scan base-package="org.examples.JDBC.dao" />
<bean id="dataSource"
class="org.springframework.jdbc.datasource. DriverManagerDataSource">
<property name="driverClassName" value="${jdbc.driverClassName}" />
<property name="url" value="${jdbc.url}" />
</bean>
<context:property-placeholder location="jdbc.properties" />
```

The example above uses placeholders to supply connection information from a properties file. This is loaded by Spring's property placeholder from the jdbc.properties file shown below:

```
jdbc.driverClassName=oracle.jdbc.OracleDriver
jdbc.url=jdbc:oracle:thin:@localhost:1521:xe
```

Accessing the DataSource

Once the configuration files are ready, we can use Spring to perform the following typical operations:

- Set up connection to a database
- Create a prepared statement

This usually involves creating a data access object (DAO) class. The example code below uses the annotation @Repository, so that Spring automatically scans this class and registers it as the Spring bean employeeDao.

We have also defined DataSource as a field and annotated it with the @Autowired annotation. The getConnection() method of this DataSource can be called to establish the connection based on the configuration definition in spring.xml.

This example also contains a sample getEmployeeById() method which uses a prepared statement.

```
@Repository
public class EmployeeDao {

   @Autowired
   private DataSource dataSource;
   public Employee getEmployeeById(int id) {
            Employee employee = null;
         Connection conn = null;
         try {
            conn = dataSource.getConnection();
            PreparedStatement ps = conn .prepareStatement
            ("select * from employee where id = ?");
            ps.setInt(1, id);
            ResultSet rs = ps.executeQuery();
            if (rs.next()) {
               employee = new Employee(id, rs.getString("name"));
            }
            rs.close();
            ps.close();
         } catch (SQLException e) {
            throw new RuntimeException(e);
         }
         return employee;
      }
      …
```

This can be accessed using a class with a main method, as show below.

```
public class EmployeeMain {

   public static void main(String[] args) {

      @SuppressWarnings("resource")
      ApplicationContext context =
         new ClassPathXmlApplicationContext("Spring.xml");
      EmployeeDao employeeDao =
      context.getBean("employeeDao", EmployeeDao.class);
      Employee employee = employeeDao.getEmployeeById(1000);
      System.out.println("Employee name: " + employee.getName());
   }
}
```

Spring JDBC Exceptions

The following table lists the ordinary JDBC exception hierarchy versus the Spring framework data-access exceptions.JDBC's exceptions Spring's data-access exceptions.

All of the Spring exceptions above extends DataAccessException. This is an unchecked exception, so Spring avoids excessive try-catch blocks.

To take advantage of Spring's data-access exceptions, it is necessary to use one of Spring's supported data-access templates.

JDBC Exceptions	Spring Data Access Exceptions
	BadSqlGrammarException
BatchUpdateException	CannotAcquireLockException
DataTruncation	CannotSerializeTransactionException
SQLException	CannotGetJdbcConnectionException
SQLWarning	CleanupFailureDataAccessException
	ConcurrencyFailureException
	DataAccessException
	DataAccessResourceFailureException
	DataIntegrityViolationException
	DataRetrievalFailureException
	DataSourceLookupApiUsageException
	DeadlockLoserDataAccessException
	DuplicateKeyException
	EmptyResultDataAccessException
	IncorrectResultSizeDataAccessException
	IncorrectUpdateSemanticsDataAccessException
	InvalidDataAccessApiUsageException
	InvalidDataAccessResourceUsageException
	InvalidResultSetAccessException
	JdbcUpdateAffectedIncorrectNumberOfRowsException
	LobRetrievalFailureException
	NonTransientDataAccessResourceException
	OptimisticLockingFailureException
	PermissionDeniedDataAccessException
	PessimisticLockingFailureException
	QueryTimeoutException
	RecoverableDataAccessException
	SQLWarningException
	SqlXmlFeatureNotImplementedException
	TransientDataAccessException
	TransientDataAccessResourceException
	TypeMismatchDataAccessException
	UncategorizedDataAccessException
	UncategorizedSQLException

Using JdbcTemplate

Spring JDBC provides the JdbcTemplate class which includes the most common mechanisms invoked when using the JDBC API to access data, create connections, create and execute statements and release resources. The JdbcTemplate class can be found in the org.springframework.jdbc.core package.

The JdbcTemplate class instances are thread-safe once configured. A single JdbcTemplate can be configured and injected into multiple DAOs.

In SQL Data Manipulation Language (DML) is used for inserting, retrieving, updating, and deleting the data in the database and Data Definition Language (DDL) is used to create and manipulate database storage structures such as tables. JdbcTemplate can be used to execute both types of operation.

The JdbcTemplate class is found in the org.springframework.jdbc.core package. It has an overloaded constructor and can be initiated using in of the following ways:

- JdbcTemplate(): The setDataSource() method must be used to set the DataSource before using this object for executing the statement.
- JdbcTemplate(DataSource): This initialises the object with specified DataSource.
- JdbcTemplate(DataSource, boolean): This initialises the object with specified DataSource and identifies how the SQL exception translator is initialised. If true then it will wait until execution, if false it will initialise immediately.

The SQL exception translator catches the JDBC exception and translates it into another exception hierarchy which is defined in the org.springframework.dao package. This class avoids common error and executes the SQL queries, updates the statements, stores the procedure calls, or extracts the results.

While using the JdbcTemplate, the application developer has to provide the code for preparing the SQL statement and dealing with the data retrieved. The JdbcTemplate handles much of the boilerplate code required for creating and releasing the database connection, leaving the developer free to concentrate on the business logic.

The following extract from the spring.xml file shows how to initialise the JdbcTemplate object using the DataSource bean as ref.

```
   …
   <context:annotation-config />
   <context:component-scan base-package="examples.JDBC.dao" />
   <bean id="dataSource" class=
   "org.springframework.jdbc.datasource. DriverManagerDataSource">
      <property name="driverClassName">
         <value={jdbc.driverClassName}></value>
      </property>
      <property name="url">
         <value={jdbc.url}></value>
      </property>
   </bean>
   <bean id="jdbcTemplate" class="org.springframework.jdbc.core.
   JdbcTemplate">
      <property name="dataSource" ref="dataSource" />
   </bean>
   <context:property-placeholder location="jdbc.properties"/>
</beans>
```

JdbcTemplate Operations

The JdbcTemplate class provides useful methods for simplifying the database operations.

This example uses a select command to query the database using the JdbcTemplate class and passing a bind variable. It counts the number of employees with job title "Clerk",

```
int count = this.jdbcTemplate.queryForObject(
"select count(*) from employee where job = ?", Integer.class, "Clerk");
```

The following queries the name of an employee with a specific id:

```
String empName = this.jdbcTemplate.queryForObject(
"select name from employee where employee_nr = ?",
new Object[]{1000}, String.class);
```

The Update() method is used to perform operations such as insert, update, or delete. The parameter values are usually provided as an object array or var args.

The following is an insert operation:

```
this.jdbcTemplate.update("insert into department (department_nr,  name,
location) values (?, ?, ?)", 50, "IT", "Stockholm");
```

An update operation:

```
this.jdbcTemplate.update("update employee set salary = ? where
employee_nr = ?", "5000", 1036);
```

A delete operation:

```
this.jdbcTemplate.update("delete from employee
where employee_nr = ?" ,Long.valueOf(empId));
```

Additionally, the execute() method can be used to run arbitrary SQL; often useful for DDL. e.g.

```
this.jdbcTemplate.execute("
   create table details (id integer, name    varchar(100))");
```

It can also be used to call stored procedures:

```
this.jdbcTemplate.update(
         "call UTIL.UPDATE_EMP_DETAILS(?)",
```

Long.valueOf(empId));

JdbcTemplate Best Practice

Instances of the JdbcTemplate class are threadsafe once configured; meaning that a single instance can be injected into multiple DAOs or repositories.

It is also stateful, maintaining a reference to a DataSource, but this state is not conversational state.

Often, using the template or NamedParameterJdbcTemplate, a DataSource is configured, and injected into the DAO classes; the JdbcTemplate is created in the setter for the DataSource e.g.

```
public class EmployeeDAOJDBCTemplateImpl
      implements EmployeeDAO {

   private JdbcTemplate jdbcTemplate;

   public void setDataSource(DataSource dataSource) {
      this.jdbcTemplate = new JdbcTemplate(dataSource);
   }
   ...
```

or created a little later:

```
public class EmployeeDAOJDBCTemplateImpl
      implements EmployeeDAO {

   private DataSource dataSource;

   public void setDataSource(DataSource dataSource) {
      this.dataSource = dataSource;
   }

   @Override
   public void save(Employee employee) {
      JdbcTemplate jdbcTemplate =
         new JdbcTemplate(dataSource);
   ...
```

Configuration may resemble:

```
<bean id="employeeDao" class="com.training.dao.EmployeeDao">
   <property name="dataSource" ref="dataSource"/>
</bean>

<bean id="dataSource"
    class="org.apache.commons.dbcp.BasicDataSource" >
   <property name="driverClassName" value="${jdbc.driverClassName}"/>
   <property name="url" value="${jdbc.url}"/>
   <property name="username" value="${jdbc.username}"/>
   <property name="password" value="${jdbc.password}"/>
</bean>

<context:property-placeholder location="jdbc.properties"/>
```

Alternatively, use component scanning and annotation support for dependency injection. Annotate the class with @Repository to enable it for component scanning, and annotate the DataSource setter method with @Autowired e.g.

```
@Repository
public class EmployeeDAOJDBCTemplateImpl
      implements EmployeeDAO {

   private JdbcTemplate jdbcTemplate;

   @Autowired
   public void setDataSource(DataSource dataSource) {
      this.jdbcTemplate = new JdbcTemplate(dataSource);
   }
   ...
```

The corresponding configuration being:

```
   <context:component-scan base-package="com.training.*" />

   <bean id="dataSource"
         class="org.apache.commons.dbcp.BasicDataSource">
            <property name="driverClassName"
          value="${jdbc.driverClassName}"/>
            <property name="url" value="${jdbc.url}"/>
            <property name="username" value="${jdbc.username}"/>
            <property name="password" value="${jdbc.password}"/>
   </bean>

   <context:property-placeholder location="jdbc.properties"/>
   ...
```

Regardless of chosen style, it should not be necessary to create a new instance of JdbcTemplate each time SQL is to be executed. A configured template instance is thread safe.

Multiple template instances may be required if the application accesses multiple databases, uses multiple DataSources, and hence multiple templates.

NamedParameterJdbcTemplate

NamedParameterJdbcTemplate class adds support for JDBC statements using named parameters. The class wraps a JdbcTemplate, and delegates to the wrapped JdbcTemplate to do the work. e.g.

```
private NamedParameterJdbcTemplate namedParameterJdbcTemplate;

public void setDataSource(DataSource dataSource) {
      this.namedParameterJdbcTemplate =
      new NamedParameterJdbcTemplate(dataSource);
}

public void insertEmployee(Employee emp) {
   String query = "insert into employee
      (emp_id, surname, desc) values
         (:empId, :surname, :desc)";
   Map namedParameters = new HashMap();
   namedParameters.put("empId",
      Integer.valueOf(emp.getEmpId()));
   namedParameters.put("surname", emp.getSurname());
   namedParameters.put("desc", emp.getDesc());
   namedParameterJdbcTemplate.update(query, namedParameters);
}
...
```

A rather nice SqlParameterSource implementation is the BeanPropertySqlParameterSource class. This wraps an arbitrary JavaBean and uses the properties of the wrapped JavaBean as the source of named parameter values e.g.

```
public int countOfEmployees(Employee emp) {

   String sql = "select count(*) from employee
      where forename = :forename and surname = :surname";

      SqlParameterSource namedParameters =
      new BeanPropertySqlParameterSource(emp);

      return this.namedParameterJdbcTemplate.
      queryForObject(sql, namedParameters, Integer.class);
}
```

Calling Stored Procedures

A stored procedure is a program unit which is stored in a DBMS. Procedures are usually written in a language that extends the database-specific SQL (PL/SQL in Oracle, T-SQL in SQL Server, etc.). They can be called from the database to perform operations on it in a manner analogous the method calling.

The class SimpleJdbcCall can be used to represent a call to a stored procedure as a multithreaded and reusable object. It handles the lookups required to simplify the code used to access the basic stored procedure. All that is required to execute is the name of the stored procedure.

It can match any supplied parameters with the IN and OUT parameters, specified during the declaration of the stored procedure.

The following example creates an instance of the SqlParameterSource interface which contains the parameters that must match the name of the parameter declared in the stored procedure. The execute() method accepts the IN parameter as an argument ("empNr" in this example) and returns a map containing the OUT parameters specified in the stored procedure. Here the OUT parameter is "name". The retrieved value is set to the employee instance of employee.

```
@Repository
public class EmployeeDao {

   @Autowired
   private DataSource dataSource;

   @Autowired
   private JdbcTemplate jdbcTemplate;

   private SimpleJdbcCall jdbcCall;

   public void setJdbcTemplateObject(JdbcTemplate jdbcTemplate) {
      this.jdbcTemplate = jdbcTemplate;
   }

   @Autowired
   public void setDataSource(DataSource dataSource) {
      this.dataSource = dataSource;
      this.jdbcCall = new SimpleJdbcCall(this.dataSource)
         .withProcedureName("getEmployee");
   }


   @Override
   public Employee getEmployee(Integer empNr) {
      SqlParameterSource in = new MapSqlParameterSource()
         .addValue("id", id);
      Map<String, Object> simpleJdbcCallResult = jdbcCall.execute(in);
      Employee employee = new Employee(empNr, (String)
      simpleJdbcCallResult.get("name"));
      return employee;
   }
}
```

CHAPTER 8

Spring & Hibernate ORM

Introduction

Data persistence is the ability to preserve the state of an object so that it can regain the same state in the future. This section focuses on saving in-memory objects into the database using ORM tools that have wide support in Spring

The JDBC data access mechanisms described in the previous section are extremely powerful when persisting data to permanent storage using an SQL database, but they do have their limitations. As applications become more complex, so do our persistence requirements. It is more useful to be able to map object properties to database columns and have our statements and queries created by a third party. This frees developers from writing large amounts of boilerplate code and complex prepared statements.

Using object relationship mapping (ORM) software with Spring provides these features and also allows others that are more sophisticated:

- Lazy loading

As object graphs become more complex, you sometimes don't want to fetch entire relationships immediately. Lazy loading allows you to grab data only as it's needed.

- Eager fetching

This is the opposite of lazy loading. Eager fetching allows you to grab an entire object graph in one query. In the cases where you know you need a Department object and its associated Employee objects, eager fetching lets you get this from the database in one operation, saving you from costly round-trips.

- Cascading

Sometimes changes to a database table should result in changes to other tables as well. Going back to the department example, when a Department object is deleted, you also want to delete the associated Employee data from the database.

Spring provides support for several persistence frameworks, including Hibernate,

iBATIS, Java Data Objects (JDO), and the Java Persistence API (JPA). As with Spring's JDBC support, Spring's support for ORM frameworks provides integration points to the frameworks as well as some additional services:

- Integrated support for Spring declarative transactions
- Transparent exception handling
- Thread-safe, lightweight template classes
- DAO support classes
- Resource management

Although Spring supports a large number of ORM solutions, the manner in which it does so is consistent between them. Becoming familiar with how Spring deals with one ORM framework makes it easy to switch to another.

Spring uses POJO-based development and also uses declarative configuration management to overcome the complex architecture associated with the EJB. ORM software became popular for Java development because it emphasises using a simple, lightweight POJO-based framework.

Hibernate is one of the most successful ORM libraries available in the open source community. It gained popularity with features such as its POJO-based approach, support of relationship definitions, and ease of development.

Spring provides support for several persistence frameworks, including Hibernate,

iBATIS, Java Data Objects (JDO), and the Java Persistence API (JPA). As with Spring's JDBC support, Spring's support for ORM frameworks provides integration points to the frameworks as well as some additional services:

- Integrated support for Spring declarative transactions
- Transparent exception handling
- Thread-safe, lightweight template classes
- DAO support classes
- Resource management

Although Spring supports a large number of ORM solutions, the manner in which it does so is consistent between them. Becoming familiar with how Spring deals with one ORM framework makes it easy to switch to another.

Spring uses POJO-based development and also uses declarative configuration management to overcome the complex architecture associated with the EJB. ORM software became popular for Java development because it emphasises using a simple, lightweight POJO-based framework.

Hibernate is one of the most successful ORM libraries available in the open source community. It gained popularity with features such as its POJO-based approach, support of relationship definitions, and ease of development.

Features of Object Relational Mapping

A relational database is the persistence mechanism used in most enterprise applications. Object-oriented languages such as Java represent data as an interconnected Graph of Objects, whereas relational database systems represent data in a table-like format.

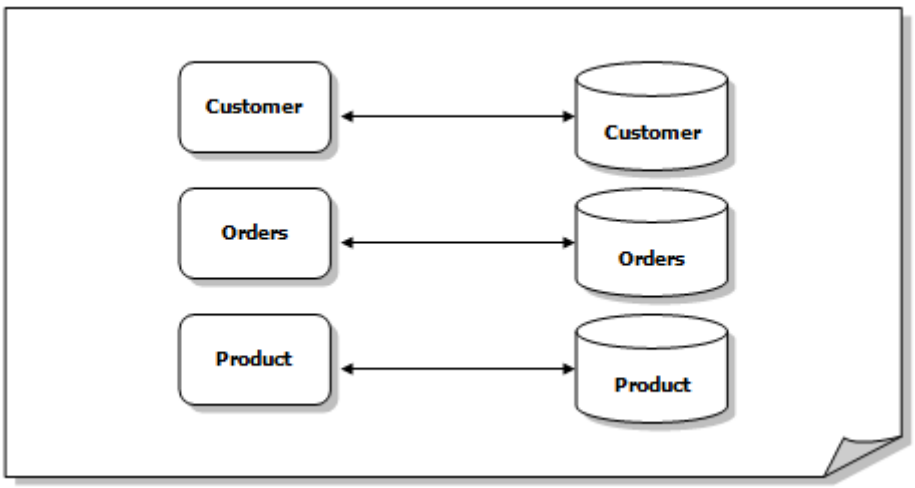
Because both the models are quite different in the way they represent data, when we load or store graphs of objects using relational databases, it can cause mismatch problems. A Java object can encompass partial data from a single database table or include data from multiple tables depending on the structure of the relational database.

ORM software is designed to address the following issues:

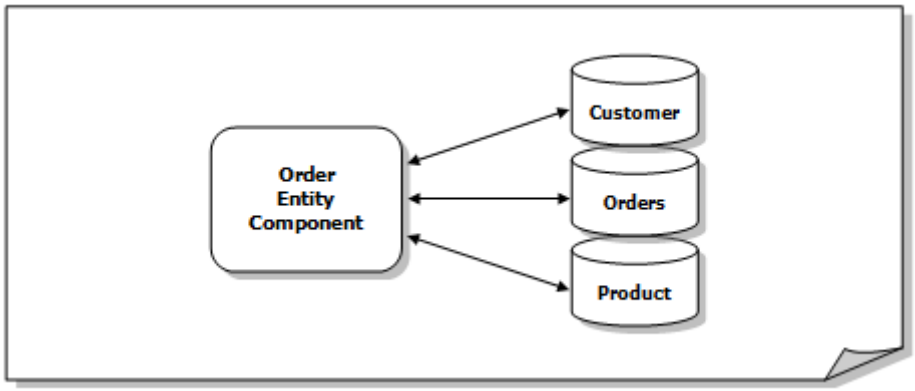
Mapping Schemas

It can be a very time consuming task to write code to translate from an object-oriented domain scheme to a relational database scheme. Object Relational Mapping (ORM) software can be used to provide mapping to OO software developers without the need for complex coding. Often the configuration information in the form of code annotation or xml configuration files is all that needs to be supplied to ORM software.

The simplest ORM software supports a simple mapping of Java objects to database tables. This is demonstrated in the diagram below.



ORM software has the ability to map Java objects to database table structures that do not have a simple one to one mapping. This is demonstrated in the figure below.



Data Type Conversion

Each relational database has its own data type implementations. Most of these are very similar, but there are notable differences in how they handle dates, timestamps, floating point numbers and Booleans, to name a few.

When using JDBC, it is the responsibility of the developer to ensure that these potential mismatches are correctly handled, and that conversions from the database to Java and back again do not produce errors.

Updating the Database

Managing changes to object state is another issue that needs to be addressed. If there are some changes to object state, then the developer must manually execute the procedure to make these changes and possibly to reframe the SQL queries and update the database by as well.

The ORM Solution

Developers can configure ORM to store its mapping information as metadata using XML files or as annotations on mapped objects. This can be used to define how to map a persistent class and its fields into database tables and theirs columns.

The database-specific SQL used ("database dialect" in ORM terms) is also specified by the developer and generated by the ORM software when required.

Introducing Hibernate

A very useful high level overview is provided courtesy of the Hibernate ORM project page:

Hibernate ORM enables developers to more easily write database-enabled applications. As an **Object/Relational Mapping** (ORM) framework, Hibernate is concerned with data persistence as it applies to relational databases (via JDBC).

As well as its own API, Hibernate is also an implementation of the Java Persistence API (JPA) specification, making it a **JPA Provider**, and can be easily used in any environment supporting JPA including Java SE applications, Java EE application servers, Enterprise OSGi containers.

Hibernate caters for **idiomatic persistence**, meaning that persistent classes following natural Object-oriented idioms including inheritance, polymorphism, association, composition, and the Java collections framework. Hibernate requires no interfaces or base classes for persistent classes and enables any class or data structure to be persistent.

Hibernate supports **lazy initialization**, numerous fetching strategies and optimistic locking with automatic versioning and time stamping. Hibernate requires no special database tables or fields and generates much of the SQL at system initialization time instead of at runtime.

Hibernate consistently offers **superior performance** over straight JDBC code, both in terms of developer productivity and runtime performance.

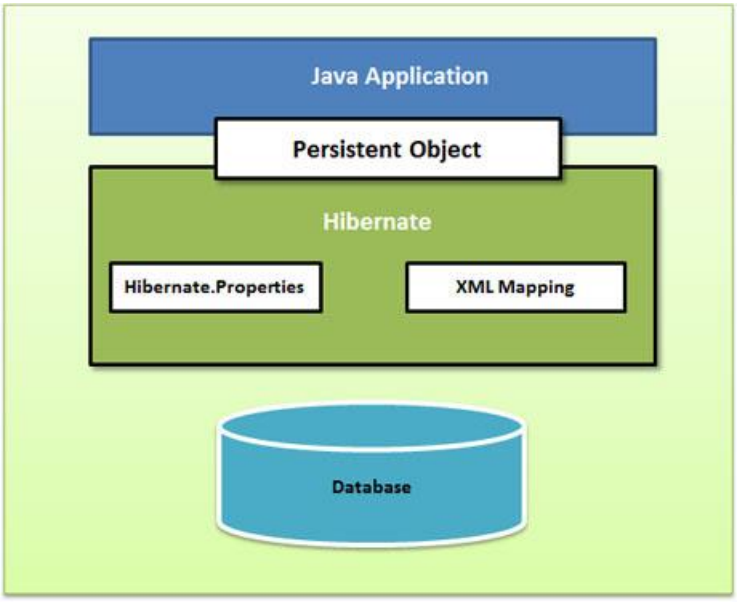
Hibernate was designed to work in an application server cluster and deliver a highly **scalable** architecture. Hibernate scales well in any environment: Use it to drive your in-house Intranet that serves hundreds of users or for mission-critical applications that serve hundreds of thousands.

Hibernate is well known for its excellent **stability** and **quality**, proven by the acceptance and use by tens of thousands of Java developers.

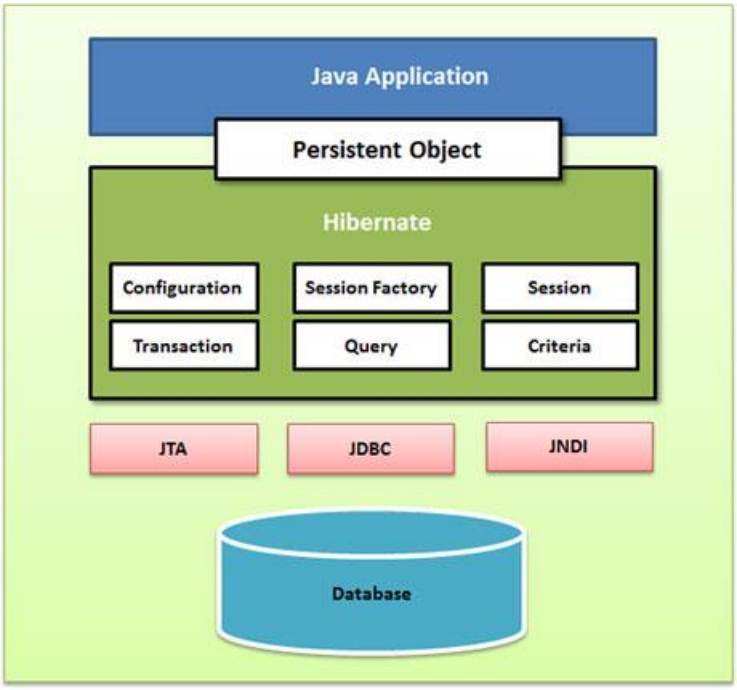
Hibernate is highly **configurable** and **extensible**.

Hibernate Architecture

The Hibernate architecture is layered to isolate the developer from having to know the underlying APIs. Hibernate makes use of the database and configuration data to provide persistence services (and persistent objects) to the application. For a high level view of the architecture:



A more detailed view of the architecture, including some of the core classes:



Hibernate uses various existing Java APIs, like JDBC, Java Transaction API(JTA), and Java Naming and Directory Interface (JNDI). JDBC provides a standard level of abstraction of functionality common to relational databases, allowing almost any database with a JDBC driver to be supported. JNDI and JTA allow Hibernate to be integrated with Java application servers.

In order to appreciate the architecture, it is useful to have a high-level introduction to the main components and classes present in a Hibernate application:

Configuration Object

The Configuration object is the first Hibernate object created in a Hibernate application and usually created just once during application initialization. It represents a configuration or properties file required by Hibernate. The object provides two keys components:

- Database Connection. This is handled through one or more configuration files; hibernate.properties and hibernate.cfg.xml.
- Class Mapping Setup. This component creates the connection between Java classes and database tables..

SessionFactory Object

The Configuration object is used to create a SessionFactory object, which in turn continued configuration and allows for the instantiation of a Session object. This is a thread safe object used by all application threads.

The SessionFactory is a heavyweight object, so is usually created during application start up and retained. Only one SessionFactory object is required per database, so if using multiple databases, there are multiple SessionFactory objects to be managed.

Session Object

A Session is used to obtain a physical connection with a database. This object is lightweight and designed to be instantiated each time an interaction is required with the database. Persistent objects are saved and retrieved through the Session object.

Session objects should not be kept open for a long time, as they are not usually thread safe. They should instead be created and destroyed them as required.

Transaction Object

A Transaction represents a unit of work with the database and most relational databases support transactions. Transactions in Hibernate are handled by an underlying transaction manager and transaction from JDBC or the JTA.

This is an optional interface; Hibernate application developers may choose not to use this, and instead to manage transactions in separate application code.

Query Object

Query objects use SQL or Hibernate Query Language (HQL) strings to retrieve more custom data from the database and create objects. A Query instance is used to bind query parameters, limit results returned by the query, and to execute the query.

Criteria Object

Criteria object are used to create and execute object oriented criteria queries to retrieve objects.

Integrating Hibenate and Spring

The Hibernate framework allows the developer to configure the details for accessing a particular data source in the XML files and/or in the Java annotations. There is no need to write the code to manage the connection or to deal with statements and result sets.

When using Hibernate with Spring, the business objects are configured with the help of the IoC container and can be externalized from the application code. Hibernate objects can be used as Spring beans in your application and you can avail all the benefits of the Spring Framework.

The simplest way to integrate Hibernate with Spring is to have a bean for SessionFactory and make it a singleton. The data access objects can then gain access to that bean and inject its dependency, giving them the session from the SessionFactory. The first step in creating a Spring Hibernate project is to integrate Hibernate and connect with the database.

Downloading Hibernate

It is assumed that Java SE 7/8 is installed locally.

Choose whether you want to install Hibernate on Windows, or Unix and then proceed to the next step to download .zip file for windows and .tz file for Unix.

Download and unzip the latest version of the Hibernate ORM installation from:

http://hibernate.org/orm/downloads/

Hibernate 5 is the current release at time of writing.

Installing Hibernate

Copy all the library files from /lib/required to the local CLASSPATH, and update the classpath to include these JARs.

Extended JARs are available in the neighbouring lib directories, for instance /jpa has the Entity Manager content.

The Hibernate SessionFactory

Spring allows developers to define the Hibernate SessionFactory as a Spring bean in an application context. This prevents the need for hardcoded resource lookups for application objects.

The Session interface in the Hibernate API provides methods to find, save, and delete database objects. Developers can create a Hibernate session by first creating the SessionFactory. The Spring Framework provides a number of classes to configure Hibernate SessionFactory as a Spring bean containing the desired properties.

Spring provides an implementation of the AbstractSessionFactoryBean subclass to enable session creation. The following two classes are available:

- LocalSessionFactoryBean – supports the Hibernate XML configuration file
- AnnotationSessionFactoryBean - supports annotation metadata for mappings

The configuration file requires the following settings:

- sessionFactory
- dataSource

  To set the name of the data source to be accessed by the underlying application
- packagesToScan or annotatedClasses

  To scan the domain object with the ORM annotation under the specified package
- hibernateProperties

  To set the configuration details for Hibernate

The following table describes these properties:

Property	Purpose
hibernate.dialect	Hibernate uses this property to generate the appropriate SQL optimized for the chosen relational database.
hibernate.max_fetch_depth	This property is used to set the maximum depth for the outer join when the mapping object is associated with other mapped objects. This property is used to determine the number of associations Hibernate will traverse by join when fetching data. The recommended value lies between 0 and 3.
hibernate.jdbc.fetch_size	This property is used to set the total number of rows that can be retrieved by each JDBC fetch.
hibernate.show_sql	This property file is used to output all SQL to the log file or console, which is an alternative to set log to debug and troubleshooting process. It can be set to either True or False.

XML Configuration

Spring beans, the data source, a SessionFactory, and a transaction manager bean are configured in the app-context.xml file, an implementation of which is shown below:

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:context="http://www.springframework.org/schema/context"
xmlns:tx="http://www.springframework.org/schema/tx"
xmlns:jdbc="http://www.springframework.org/schema/jdbc"
xsi:schemaLocation="
http://www.springframework.org/schema/jdbc
http://www.springframework.org/schema/jdbc/spring-jdbc.xsd
http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans.xsd
http://www.springframework.org/schema/tx
http://www.springframework.org/schema/tx/spring-tx.xsd
http://www.springframework.org/schema/context
http://www.springframework.org/schema/context/spring-context.xsd">
```

The component scan attribute instructs Spring where to look for the relevant classes.

```
<context:annotation-config />
<context:component-scan base-package="org.spring.hibernate" />
```

The property-placeholder attribute points to a file called hibernate.properties file, as shown below:

```
<context:property-placeholder
location="classpath:/META-INF/spring/hibernate.properties" />
```

This declares the dataSource bean:

```
<bean id="dataSource"
class="org.springframework.jdbc.datasource. DriverManagerDataSource">
<property name="driverClassName" value="${jdbc.driverClassName}" />
<property name="url" value="${jdbc.url}" />
<property name="username" value="${jdbc.username}" />
<property name="password" value="${jdbc.password}" />
</bean>
```

The following example declares the sessionFactory bean using the built-in class AnnotationSessionFactoryBean. This also instructs Hibernate to scan for the ORM annotated object and specifies hibernateProperties for configuration details:

```
<bean id="sessionFactory"
class="org.springframework.orm.hibernate3.
   annotation.AnnotationSessionFactoryBean">
<property name="dataSource" ref="dataSource" />
<property name="annotatedClasses"
value="org.spring.hibernate.model.Employee" />
<property name="hibernateProperties">
<props>
<prop key="hibernate.dialect">${hibernate.dialect}</prop>
<prop key="hibernate.show_sql">${hibernate.show_sql}</prop>
</props>
</property>
</bean>
```

A transaction manager is required to access transactional data. This example uses org. springframework.orm.hibernate3 which is provided by Spring.

HibernateTransactionManager:

```
<bean id="transactionManager"
class="org.springframework.orm.hibernate3.
  HibernateTransactionManager">
<property name="sessionFactory" ref="sessionFactory" />
</bean>
```

The <tx:annotation-driven> tag declared support for transaction specification using annotations:

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

This example stores the database-specific properties in a hibernate.properties file, as show below:

```
# JDBC Properties
jdbc.driverClassName=oracle.jdbc.OracleDriver
jdbc.url=jdbc:oracle:thin:@localhost:1521:user1
jdbc.username=postgres
jdbc.password=sa
# Hibernate Properties
hibernate.dialect=org.hibernate.dialect.OracleDialect
hibernate.show_sql=true
```

Hibernate Class Annotations

The Java Persistence API (JPA) defines the entity class in the object tier as a representation of a table in the database tier. An entity instance is defined as the object tier equivalent of a row in a database table.

The following table displays the default mapping of object tier elements to database tier elements:

OBJECT TIER ELEMENT	DATABASE TIER ELEMENT
Entity class	Database table
Field of entity class	Database table column
Entity instance	Database table record

Hibernate annotation provides the metadata for object and relational table mapping. Hibernate provides a JPA implementation, which allows the user to use JPA annotation in Hibernate beans. The following table explains the most common JPA annotations:

JPA annotation	Description
@Entity	Declares a class as an entity bean that can be persisted by Hibernate
@Table	Can be used to define table mapping. It provides four attributes that allows us to override the table name, its catalogue, and its schema.
@Id	Defines the primary key, and it will automatically determine the appropriate primary key generation strategy to be used
@GeneratedValue	Identifies that the field will be automatically generated. The GenerationType.IDENTITY strategy allows the generated id value to be mapped to the bean so it can be retrieved by the Java program
@Column	Maps the field with the table column.

Entity Classes

Entity classes have their state synchronized with a database. The state of an entity class is obtained from either its variables (fields) or its accessor methods (properties). Field or property-based access is determined by the placement of annotations. Hibernate supports both, but persistent fields is the more common option, and is illustrated below.

```
@Entity
@Table(name = "EMPLOYEE")
public class Employee {

   @Id
   @Column(name = "EMPLOYEE_NR")
   @GeneratedValue(strategy = GenerationType.IDENTITY)
   private Integer empNr;
   @Column(name = " NAME")
   private String name;
   @Column(name = "JOB")
   private String job;
   @Column(name = "DEPARTMENT_NR")
   private String departmentNr;
   @Column(name = "SALARY")
   private int salary;

   ...constructors, get and set methods, equals() and hashCode() …
}
```

Hibernate Session Interface

The Hibernate Session interface offers create, delete, and read operations for instances of mapped entity classes and is obtained from a SessionFactory. It is initiated each time an interaction with the database needs to occur. It is not usually thread safe, so should not be kept open for a long time.

An instance may exist in one of the following three states at a given point in time:

Instance State	Meaning
Transient	This instance of the persistence class is new and has no corresponding record in a database and is not associated with Session.
Persistent	The instance of a persistence class has a corresponding record in the database.
Detached	The persistent object will be detached from the database once the Hibernate Session will be closed.

Example Data Access Object

The following example is a data access object (DAO) class. It does the following:

- Gains access to a SessionFactory by annotating a variable of type SessionFactory with the @Autowired annotation that automatically initialises it.
- Gets a session object from the sessionFactory
- Uses the session object's createQuery method to create and run queries.
- Allows Spring to automatically close the session.

```
@Repository
@Transactional(readOnly = true)
public class EmployeeDao {

   @Autowired
   private SessionFactory sessionFactory;

   public List<Employee> getEmployees() {
      Session session = sessionFactory.openSession();
      String hql = "FROM Employee";
      Query query = session.createQuery(hql);
      List<Employee> emList = query.list();
      return emList;
   }

   @Transactional(readOnly = false)
   public void insertEmployee(Employee employee) {
      Session session = sessionFactory.openSession();
      session.save(employee);
   }
}
```

Example Service to Access DAO

The class below declares a variable named of type EmployeeDao and annotates it with @Autowired. This class is annotated with the @Service annotation, so it can be accessed as a service class:

```
@Service
public class EmployeeServiceImpl implements EmployeeService {

   @Autowired
   private EmployeeDao employeeDao;

   public List<Employee> getEmployees() {
      List<Employee> emps = employeeDao.getEmployees();
      return emps;
   }

   public void insertEmployee(Employee employee) {
      employeeDao.insertEmployee(employee);
   }
}
```

Running the Application

This sample main() method uses the ApplicationContext to initialize the container.

```
public static void main(String[] args) {

   ApplicationContext context =
      new ClassPathXmlApplicationContext(
         "/META-INF/spring/applicationContext.xml");
   EmployeeService employeeService = context.getBean(
      "employeeServiceImpl", EmployeeService.class);

   for (Employee employee : employeeService.getEmployees())
      System.out.println(employee);
}
```

CHAPTER 9

Spring Transaction Management

Introduction

Spring Framework has comprehensive transaction support with the following benefits:

- Consistent programming model across different transaction APIs such as Java Transaction API (JTA), JDBC, Hibernate, Java Persistence API (JPA), and Java Data Objects (JDO).
- Support for declarative transaction management.
- Simpler API for programmatic transaction management than complex transaction APIs such as JTA.
- Excellent integration with Spring's data access abstractions.

Prior to Spring, Java EE developers were limited to global or local transactions; here, we look at the landscape of transactions before seeing how Spring can offer help.

Transaction Managers

These application components handle the coordination of transactions between multiple resources. The transaction manager is responsible for the creation and maintenance of transaction objects and the tracking of all the transaction resources.

Traditionally, committing a transaction made the changes implemented within it permanent. When using a transaction manager, the internals undergo a two-phase commit protocol:

Phase 1	All resources that are part of the transaction are prepared prior to a commit.
Phase 2	Resource managers are informed of the transaction result status by the manager. Abort or commit.

Resource Managers also undertake other work:

Log Maintenance

Records events, maintain a log of each transaction. Logging start times, commit times, times of related decisions, when resources are used, when a transaction commits or is aborted.

Transaction Demarcation

The manager uses the begin, commit and rollback methods.

Control Transaction Context

All information needed to monitor transactions is held in the context.

Multiple Resource Management

Usually a manager may handle multiple application instances simultaneously.

Failure Recovery

Ensure that resources are consistent in case of a failure condition; this ensuring database integrity.

Transaction Manager Types

There are different types of transaction managers:

Local

Handles transactions over a single resource and is embedded within the resource. An application may have multiple local transaction managers working independently.

Global

Handles transactions that span multiple resources. Usually, a Transaction Processing (TP) monitor is required to coordinate the transaction in this case.

Distributed

If resources are distributed, connections becomes difficult. A distributed transaction manager is used to indirectly connect multiple resources through a web service.

Spring Transaction Management

The Spring Framework's transaction management module is capable of supporting:

- Global and local transactions
- Nested transactions
- Save points
- All Java based environments
- Programmatic transaction management using TransactionTemplate
- Declarative transaction management using XML metadata and annotations

PlatformTransactionManager that can support:

- Transaction managed on JDBC connection
- ORM related transaction management
- JTA managed transactions

Spring Framework allows consistent abstraction for transaction management which has the following benefits for transaction management:

- Provides a programming model compatible with various transaction APIs, such as JTA, JDBC, Hibernate, JPA, and JDO.

Supports declarative transaction management to separate transaction code from the business logic. Annotations or XML-based configurations can be used.

Transaction Manager Interfaces

Spring Framework's transaction abstraction is governed by a transaction strategy that is defined by the org.springframework.transaction.PlatformTransactionManager interface. Below is an example of implementing this interface.

```
public interface PlatformTransactionManager {

   TransactionStatus getTransaction(
         TransactionDefinition definition)
      throws TransactionException;
   void commit(TransactionStatus status)
      throws TransactionException;
   void rollback(TransactionStatus status)
      throws TransactionException;
}
```

The PlatformTransactionManager is an interface, so can be mocked or stubbed. It is not bound to any look up strategy. The PlatformTransactionManager implementations can be defined like any other bean.

Method	Description
void commit (TransactionStatus status)	Commits a transaction based on its status: New, which means to commit the new transaction and move on to the next Not new, which means to reject to commit to complete the transaction Roll back, which means to perform roll back and not commit
getTransactionStatus(TransactionDefinition definition)	Used to return an active transaction or a new transaction based on the specified behaviour
void rollback(TransactionStatus status)	Used to perform a roll back on the specified transaction

The TransactionDefinition interface is also important. The properties of a transaction that can be defined are:

Property	Description
ISOLATION	ISOLATION_DEFAULT Use the default isolation level of the underlying datastore. ISOLATION_READ_COMMITTED Indicates that dirty reads are prevented; non-repeatable reads and phantom reads can occur. ISOLATION_READ_UNCOMMITTED Indicates that dirty reads, non-repeatable reads and phantom reads can occur. ISOLATION_REPEATABLE_READ Indicates that dirty reads and non-repeatable reads are prevented; phantom reads can occur. ISOLATION_SERIALIZABLE Indicates that dirty reads, non-repeatable reads and phantom reads are prevented.

Property	Description
PROPAGATION	PROPAGATION_MANDATORY Support a current transaction; throw an exception if no current transaction exists. Analogous to the EJB transaction attribute of the same name. PROPAGATION_NESTED Execute within a nested transaction if a current transaction exists, behave like PROPAGATION_REQUIRED else. There is no analogous feature in EJB. PROPAGATION_NEVER Do not support a current transaction; throw an exception if a current transaction exists. Analogous to the EJB transaction attribute of the same name. PROPAGATION_NOT_SUPPORTED Do not support a current transaction; rather always execute non-transactionally. Analogous to the EJB transaction attribute of the same name. PROPAGATION_REQUIRED Support a current transaction; create a new one if none exists. Analogous to the EJB transaction attribute of the same name. PROPAGATION_REQUIRES_NEW Create a new transaction, suspending the current transaction if one exists. Analogous to the EJB transaction attribute of the same name. PROPAGATION_SUPPORTS Support a current transaction; execute non-transactionally if none exists. Analogous to the EJB transaction attribute of the same name.
TIMEOUT_DEFAULT	Use the default timeout of the underlying transaction system, or none if timeouts are not supported.

READONLY	Does not change any data but used for optimization

The TransactionStatus interface provides a simple way for transactional code to control transaction execution and query transaction status. The concepts should be familiar, as they are common to all transaction APIs:

```
public interface TransactionStatus extends SavepointManager {

   boolean isNewTransaction();
   boolean hasSavepoint();
   void setRollbackOnly();
   boolean isRollbackOnly();
   void flush();
   boolean isCompleted();
}
```

Whether using declarative or programmatic transactions, using the correct implementation is important. They are usually delivered via dependency injection.

For example, using JDBC, firstly define a DataSource:

```
<bean id="dataSource"
    class="org.apache.commons.dbcp.BasicDataSource" >
   <property name="driverClassName"
      value="${jdbc.driverClassName}" />
   <property name="url" value="${jdbc.url}" />
   <property name="username" value="${jdbc.username}" />
   <property name="password" value="${jdbc.password}" />
</bean>
```

The PlatformTransactionManager bean will reference the DataSource:

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

Refer to the Spring Framework documentation for implementations for JPA and Hibernate.

Transaction Management Types

Spring Framework's transaction management module supports both declarative and programming approaches:

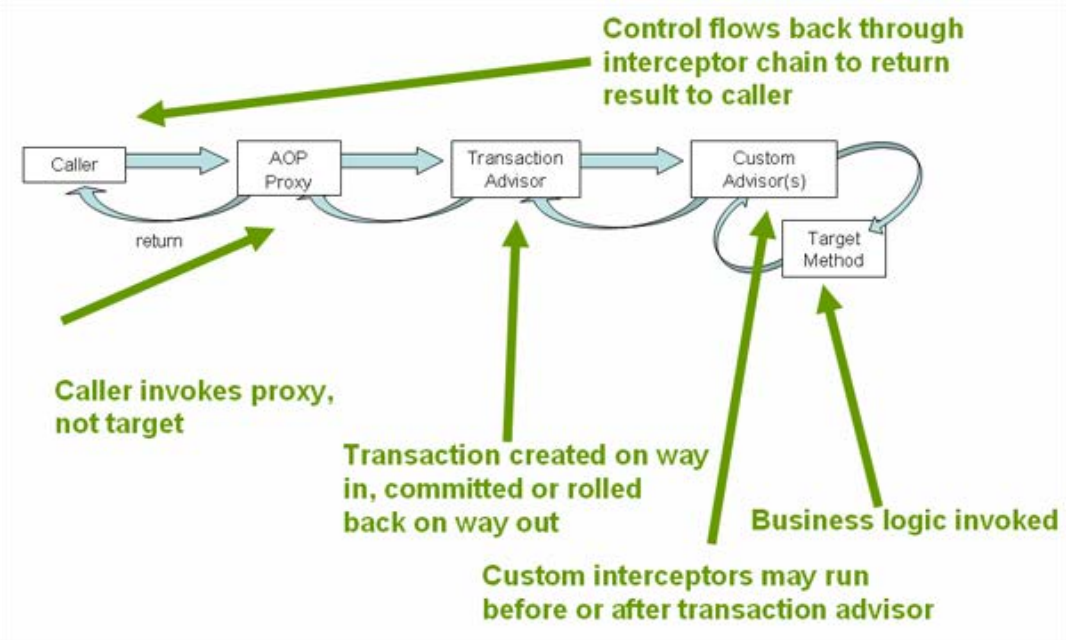
1. Declarative, by declaring AOP proxy and using this proxy to define transactions.
2. Programmatic, by defining transactions directly in the application code.

Declarative Transaction Management

Declarative transaction management is made possible through the use of the Spring AOP Framework. The transaction Advice is derived from transactional metadata, which is used to create an AOP proxy using the TransactionInterceptor and PlatformTransactionManager interfaces to execute transactions around method invocation. The benefits of using the declarative approach are:

- Works in any environment; JDBC, JDO, or Hibernate.
- Can be applied to any class, not just EJB classes.
- Allows roll back to be controlled declaratively using roll back rules. These allow chosen exceptions to be triggered on rollback via XML-based configuration.

The diagram shows how a method can be called on a transactional AOP proxy:



Programmatic Transaction Management

Spring Framework provides two means of programmatic transaction management:

1. TransactionTemplate
2. PlatformTransactionManager

TransactionTemplate is recommended for programmatic transaction management, while the PlatformTransactionManager is similar to the JTA UserTransaction API.

Using the TransactionTemplate

Similar to JdbcTemplate, this uses a callback approach. To use this approach:

1. Write a TransactionCallback implementation as an anonymous inner class.
2. The inner class has the code to execute in the transaction context.
3. Pass an instance of TransactionCallback to the execute() method

If there is no return value, use the TransactionCallbackWithoutResult class:

```
transactionTemplate.execute(
      new TransactionCallbackWithoutResult() {
      protected void doInTransactionWithoutResult(
         TransactionStatus status) {
            updateOperation1();
      }
});
```

Callback code can roll the transaction back using the setRollbackOnly() method on the supplied status:

```
transactionTemplate.execute(
      new TransactionCallbackWithoutResult() {

      protected void doInTransactionWithoutResult(
         TransactionStatus status) {
            try {
               updateOperation1();
            } catch (Exception ex) {
                status.setRollbackOnly();
            }
      }

});
```

Transaction Settings

Settings such as propagation mode, isolation level, and timeout may be set either programmatically or via configuration. e.g.

```
private final TransactionTemplate transactionTemplate;

public SimpleService(
      PlatformTransactionManager transactionManager) {

   Assert.notNull(transactionManager,
      "The 'transactionManager' argument must not be null.");
   this.transactionTemplate =
      new TransactionTemplate(transactionManager);

   this.transactionTemplate.setIsolationLevel(
      TransactionDefinition.ISOLATION_READ_UNCOMMITTED);
   this.transactionTemplate.setTimeout(10);
...
```

Which Approach?

Programmatic transaction management is helpful only when you have a small number of transaction operations to be performed. Perhaps a web application, where transactions are needed only sparingly. Use the TransactionTemplate method to explicitly set the transaction name and execute a simple transaction.

Declarative transaction management is useful when there are large numbers of transactions to manage. Perhaps a banking application with numerous transactions take place between accounts both local and disparate. Declarative transaction management is simple to configure and significantly reduces the cost of transaction management in larger applications.

Transactions with Annotations

Annotations can be used to configure transactions. The @Transactional annotation is used for this. For instance, in an Employee service, the creation of Employees may be transactional:

```
public class EmployeeManagerImpl implements EmployeeManager {

   private EmployeeDAO employeeDAO;

   public void setEmployeeDAO(EmployeeDAO employeeDAO) {
      this. employeeDAO = employeeDAO;
   }

   @Transactional
   public void createEmployee (Employee employee) {
      employeeDAO.create(employee);
   }
}
```

The EmployeeManagerDAO is defined as a bean in the configuration. To enable the bean as transactional, add to the XML configuration:

```
<bean id="employeeManager"
   class="com.training.service.EmployeeManagerImpl">
   <property name="employeeDAO"
       ref="employeeDAO"></property>
</bean>

<tx:annotation-driven proxy-target-class="true"
   transaction-manager="transactionManager" />

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

</bean>

The @Transactional annotation can then be used to annotate:

- Interface definitions
- Methods on an interface
- Class definitions or a public methods on a class

It is easier to annotate classes rather than interfaces because annotations do not inherit properties. If annotating an interface, the classes contained within the interface may not inherit the transactional definitions contained within the interface.

Like any other annotation, you can define the optional attributes to the class you are annotating with the @Transactional annotation. The optional attributes are listed below:

Attribute	Description
transaction-manager	Specify the name of transaction manager to be used when name is user defined.
proxy-target-class	Controls the type of transactional proxies created for implementing transactions: When set to "true," Spring creates class-based proxies When set to "false", the standard JDK interface-based proxies are created
order	Used to define the order of the transaction advice that will be applied to beans

@Transaction to Set Properties

The @Transactional annotation can set transactional properties:

- Propagation method
- Isolation level
- Read only transaction or read and write transaction
- Transaction time out
- Roll back

The various properties of the @Transactional annotation are:

Property	Description	Default Values
propagation	Sets propagation setting	PROPAGATION_REQUIRED
isolation	Sets isolation level	ISOLATION_DEFAULT
readOnly	if transaction is Read/write vs. read-only transaction	Read/write
timeout	Specifies transaction timeout	Default time out of the underlying database
rollbackFor	Specifies the array of exception classes which must cause roll back	Roll backs are triggered by only runtime exceptions

For instance:

```
@Transactional(
   readOnly=false,propagation=Propagation.REQUIRES_NEW)
public void saveEmployeeDetails(Employee employee) {
   employeeDetailsDao.saveEmployeeDetails(employee);
}
```

CHAPTER 10

Spring Security

Introduction

This section introduces Spring security, first by explaining its authentication and authorisation features, then by introducing its dependencies. Spring security was developed to work with Java EE Enterprise applications and focuses on Spring web based applications.

The web components of Spring Security are dependent on servlet filters for their operation. This section investigates servlet filters and the Spring interceptor, and their relationship to security.

Spring makes use of two key components for security, namely the authentication manager and authentication provider. There are different ways of logging into web applications, such as HTTP basic authentication, form-based login services, anonymous login, and also "Remember Me", all of which are supported by Spring Security.

Security credentials can also be stored in a variety of sources such as flat files and databases. Spring also supports the JEE-related concept of method-level security.

The list of topics covered in this chapter is as follows:

- Introduction to Spring Security
- Review on Servlet filters
- Security use case
- Spring Security configuration
- Securing web application's URL access
- Logging into web application
- Users authentication
- Method-level security
- Developing an application using Spring MVC, Hibernate, and Security

Spring security framework began as the Acegi Security Framework, and was later adopted by Spring, and acts as a de facto standard to secure Spring-based applications. It handles authentication and authorisation for enterprise Java applications at both the web request level and the method invocation level. Spring security provides a highly customisable and powerful access control framework.

Authentication and Authorisation

The two primary functions provided by Spring security are authentication and authorization. Note that authentication does not imply authorisation.

Authentication

This is the process of assuring that the user is the one that the user claims to be. Authentication is a combination of identification and verification. Identification can be performed in a number of ways. For example, through a username and password that can be stored in a database, an LDAP server, via or single sign-on protocol. Spring provides a password encoder interface to make sure that the user's password is hashed.

Authorisation

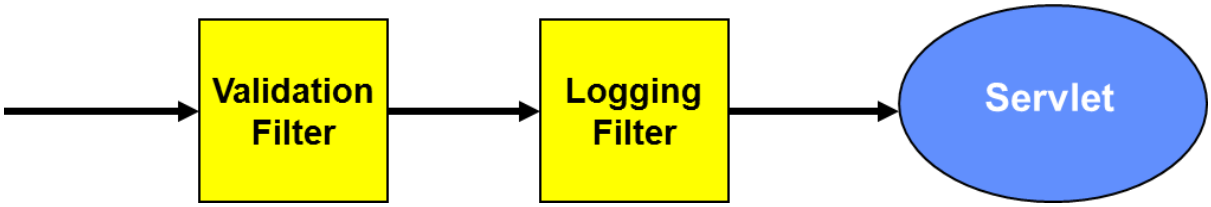
This provides access control to an authenticated user. Authorisation is the process of assuring that the authenticated user is allowed access only to those resources that they are authorized to use. The access rights given to the user of the system will determine the access rules.

In web-based applications, this is often done through URL-based security and is implemented using filters that play a primary role in securing the web application.

In addition to URL-based security, Spring provides another access layer via method-level security. Using this mechanism, individual methods can only be invoked by an authorised user.

Servlet Filters

The Spring security framework uses servlet filters concept as part of its functionality. Spring security registers a filter of the type DelegatingFilterProxy.



A filter is an object that can transform the header and content (or both) of a request or response. Filters differ from web components in that filters usually do not themselves create a response. Instead, a filter provides functionality that can be "attached" to any kind of web resource. Consequently, a filter should not have any dependencies on a web resource for which it is acting as a filter; this way it can be composed with more than one type of web resource.

The main tasks that a filter can perform are as follows:

- Query the request and act accordingly.
- Block the request-and-response pair from passing any further.
- Modify the request headers and data. You do this by providing a customized version of the request.
- Modify the response headers and data. You do this by providing a customized version of the response.
- Interact with external resources.

Applications of filters include authentication, logging, image conversion, data compression, encryption, tokenizing streams, XML transformations, and so on.

You can configure a web resource to be filtered by a chain of zero, one, or more filters in a specific order. This chain is specified when the web application containing the component is deployed and is instantiated when a web container loads the component.

Programming Filters

The filtering API is defined by the Filter, FilterChain, and FilterConfig interfaces in the javax.servlet package. You define a filter by implementing the Filter interface.

The most important method in this interface is doFilter(), which is passed request, response, and filter chain objects. This method can perform the following actions:

- Examine the request headers.
- Customize the request object if the filter wishes to modify request headers or data.
- Customize the response object if the filter wishes to modify response headers or data.
- Invoke the next entity in the filter chain. If the current filter is the last filter in the chain that ends with the target web component or static resource, the next entity is the resource at the end of the chain; otherwise, it is the next filter that was configured in the WAR. The filter invokes the next entity by calling the doFilter method on the chain object (passing in the request and response it was called with, or the wrapped versions it may have created). Alternatively, it can choose to block the request by not making the call to invoke the next entity. In the latter case, the filter is responsible for filling out the response.
- Examine response headers after it has invoked the next filter in the chain.
- Throw an exception to indicate an error in processing.

In addition to doFilter(), you must implement the init and destroy methods.

- The init() method is called by the container when the filter is instantiated. If you wish to pass initialization parameters to the filter, you retrieve them from the FilterConfig object passed to init.

The destroy() method is provided so that the filter writer can gain control before the Web container unloads the filter.

Security Logon Process

1. The user attempts to access a protected resource.
2. The Spring security framework retrieves the login page.
3. The login page checks the user's credentials against an authentication provider such as a plain text file, a database, an LDAP server, etc.
4. If the credentials given by the user are correct they will be allowed access to the protected resource, if not the login fails.
5. When the user logs out, they will be directed to the homepage.

Configuring Spring Security

Adding Spring security to a Spring web application involves the following steps.

1. Adding Spring JAR files.
2. Updating web.xml with springSecurityFilterChain.
3. Creating a Spring security configuration file.

Adding Spring Dependencies

Spring security requires three jar files, as shown below. These can be downloaded from the Spring website.

1. spring-security-core-5.X.X.release.jar

Core security library functionality

2. spring-security-config-5.X.X.release.jar

Required for Spring Security's XML namespace support

3. spring-security-web-5.X.X.release.jar

Enables filter based web security support

Maven based applications require the following updates to pom.xml:

```
<properties>
   <spring.security.version>5.0.3.RELEASE</spring.security.version>
</properties>
<!-- Spring Security -->
<dependency>
      <groupId>org.springframework.security</groupId>
      <artifactId>spring-security-core</artifactId>
      <version>${spring.security.version}</version>
</dependency>
<dependency>
   <groupId>org.springframework.security</groupId>
   <artifactId>spring-security-web</artifactId>
   <version>${spring.security.version}</version>
</dependency>
<dependency>
   <groupId>org.springframework.security</groupId>
   <artifactId>spring-security-config</artifactId>
   <version>${spring.security.version}</version>
</dependency>
```


The Spring configuration file requires an entry to specify the security framework's schema details. The SpringDispatcher-servlet.xml needs the following extra entries:

```
<beans xmlns="http://www.springframework.org/schema/beans"
   xmlns:security="http://www.springframework.org/schema/security"
   ...existing entries…
   http://www.springframework.org/schema/security
   http://www.springframework.org/schema/security/spring-security-
   4.0.xsd">
</beans>
```

Updating web.xml

Configuring web security involves setting up a filter associated with the HttpServletRequest object that provides various security features. To enable Spring Security, the filter must be defines as a mapping in the web.xml file.

The first step is to configure DelegatingFilterProxy instance in web.xml while securing the web application's URL access with Spring Security.

```
<!-- Spring Security -->
<filter>
   <filter-name>springSecurityFilterChain</filter-name>
      <filter-class>
org.springframework.web.filter.DelegatingFilterProxy
      </filter -class>
</filter>
   <filter-mapping>
<filter-name>springSecurityFilterChain</filter-name>
<url-pattern>/*</url-pattern>
</filter-mapping>
```

The DelegatingFilterProxy filter class, which is a special servlet filter, delegates the control to an implementation of javax.servlet.Filter, which is registered as a special bean with ID is springSecurityFilterChain in Spring application context.

The /* defined in the URL pattern maps to all the HTTP requests and associates springSecurityFilterChain with them. This means that all resources in this site are protected.

Creating a Security Configuration File

It is possible to separate the entire security specific configuration into a separate configuration file named spring-security.xml. If that is required the security namespace must be changed to be the primary namespace for that file. An example of spring-security.xml is show below:

```
<beans:beans
   xmlns="http://www.springframework.org/schema/security"
      xmlns:beans="http://www.springframework.org/schema/beans"
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xsi:schemaLocation="http://www.springframework.org/schema/beans
      http://www.springframework.org/schema/beans/spring-beans.xsd
      http://www.springframework.org/schema/security
      http://www.springframework.org/schema/security/spring-security
   .xsd">
      <http auto-config="true">
         <intercept-url pattern='/employeeList'
            access='USER_ROLE,ADMIN_ROLE ' />
         <intercept-url pattern='/employeeAdd' access='USER_ROLE' />
         <intercept-url pattern='/employeeDelete' access='ADMIN_ROLE'/>
      </http>
      <authentication-manager>
      <authentication-provider>
         <user-service>
            <user name="admin" password="letmein1"
               authorities="ADMIN_ROLE" />
            <user name="user1" password="letmein1"
            authorities="USER_ROLE" />
         </user-service>
      </authentication-provider>
      </authentication-manager>
```

</beans:beans>

The tags used in this configuration fall into two main categories:

- <http> and <intercept-url> tags

Web security is enabled using the <http> tag. It acts as the container element for the HTTP security configuration. Once defined it automatically sets up FilterChainProxy. The auto-config=true attribute automatically configures the basic Spring Security Services that a web application needs. This can be extended using sub-elements in the tag.

- <authentication-manager>, <authentication-provider> and <user-service> tags

The <intercept-url> element is defined inside the <http> configuration element. It restricts access to specific URLs. The <intercept-url> tag defines the URL pattern and set of access attributes that are required to access URLs. It is mandatory to include a wildcard at the end of the URL pattern since failing to do so could bypass the security check if an arbitrary request parameter is appended to the URL.

The access attributes decide if the user can access the URLs. In most cases, access attributes are defined in terms of roles. In the previous code snippet, users with the USER_ROLE role are able to access the /employeeList and /employeeAdd URLs. However, to delete an employee via the /employeeDelete URL, a user must have the role defined as ADMIN_ROLE.

The <authentication-manager> tag used to process authentication information. The <authentication-provider> tag is nested inside the <authentication-manager> tag, and used to define credential information and the roles that will be granted to this user. In the preceding code snippet, inside the <authentication-manager> tag, we have provided the <authentication-provider> tag, which specifies a text-based user ID and password.

Web Application Login

Spring security supports multiple methods for logging in to a web application.

Authentication method	Description
HTTP Basic	Client enters a username and password to a built-in browser login window. The browser sends the username and password in HTTP header in plain text, Base64 encoded. The username and password are then decoded on the server.
Form	Client authenticates using HTML form by entering a username and password. The sending mechanism, password encoding, and the style of the HTML form are customizable by the user. The form attributes (input field names and form action attributes) are defined as part of Java Servlet specification.
Logout Service	Allows users to log out of this application.
Anonymous Login	Grants authority to an anonymous user like normal user.
Remember Me Support	Remembers a user's identity across multiple browser sessions.

To demonstrate these, disable the HTTP auto-configuration option by removing the <http auto-config="true"> attribute from the <http> tag.

HTTP Basic Authentication

The HTTP basic authentication in Spring security can be configured by using the <http-basic/> element. Here, the browser will display a login dialog for user authentication:

```
<beans:beans
    …
      <http>
         <intercept-url pattern='/employeeList'
            access='USER_ROLE,ADMIN_ROLE ' />
         <intercept-url pattern='/employeeAdd' access='USER_ROLE' />
         <intercept-url pattern='/employeeDelete' access='ADMIN_ROLE'/>
     <!-- Adds Support for basic authentication -->
        <http-basic/>
      </http>
   …
</beans:beans>
```

When using HTTP authentication the browser presents a login form to the user automatically. As each request contains user authentication information that is the same as the HTTP stateless mechanism, there's no need to maintain session.

Form-based Login Service

Spring also supports the html form-based login service by providing the default login form page for users to input their login details. The <form-login> element defines the support for the login form, as shown in the following code snippet. By default, a login form, which will map to the /spring_security_login URL, will automatically be created by Spring Security, as shown here:

```
<http>
   . . .
      <!-- Adds Support for basic authentication -->
      <form-login />
</http>
```

It is also possible to create a custom login page. This is normally creates in the root directory of the web application and not in the WEB-INF directory as this will prevent users from accessing it directly.

Logout Service

The logout service handles logout requests and is configured via the <logout> element. Spring maps the logout service to the /j_spring_security_URL pattern. When the logout is successful, it redirects the user to the context path's root:

```
<http>
   . . .
      <logout />
</http>
```

This can be extended to provide the logout link in an HTML page by referencing the logout URL pattern:

```
<a href="/j_spring_security_logout"> Logout </a>
```

We can also configure log out so that the user is redirected to another URL after the logout is successful, as shown in the following code snippet:

```
<http>
   . . .
     <logout logout-success-url="/login" />
</http>
```

Anonymous Login

The <anonymous> element can be used to configure anonymous login service. In this circumstance the username and authority of the anonymous user can be configured:

```
<http>
      <intercept-url pattern='/employeeList'
         access='ROLE_ USER,ADMIN_ROLE,ROLE_GUEST ' />
      <intercept-url pattern='/employeeAdd' access='USER_ROLE' />
      <intercept-url pattern='/employeeDelete' access='ADMIN_ROLE' />
      . . .
      <anonymous username='guest' granted-authority='ROLE_GUEST' />
</http>
```

Remember Me Support

The <remember-me /> element can be used to configure the Remember Me support. This it encodes authentication information and the Remember Me expiration time along with private key as a token. It stores this to the user's browser cookie. The next time a user accesses the same application, they can be log in automatically using the token:

```
<http>
   . . .
      <remember-me />
```

</http>

Authentication Providers

The purpose of an authentication provider is to authenticate and authorise the user's principle when they attempt to log into applications and access secure resources. If a user is successfully authenticated by the authentication provider, then they will gain access to the protected resource and will be authorised to carry out approved actions.

Spring security supports a number of ways to authenticate users, such as a built-in provider with a built-in XML element or authentication against a user detail store such as a relational database or LDAP repository. Password encryption is also supported via the use of the MD5 and SHA algorithms.

In-Memory Authentication

Small numbers of relatively static application users can be defined in Spring security's configuration file. This means that their details are loaded into memory, as soon as the application starts.

```
<authentication-manager>
     <authentication-provider>
       <user-service>
         <user name="admin" password="admin1"
           authorities="ADMIN_ROLE" />
         <user name="user1" password="letmein1"
           authorities="USER_ROLE" />
        <user name="user" password="letmein1" disabled="true"
           authorities="USER_ROLE" />
       </user-service>
     </authentication-provider>
</authentication-manager>
```

The user's details can be defined in <user-service> tag.Multiple <user> elements can be defined inside this, each of which can have a username, password, disabled status, and a granted authority specified.

The disabled=true attribute indicates that the user cannot currently access the application.

Using a Properties Files

The user details can also be stored externally in a properties file. The following example indicates the location of a properties file for this purpose:

```
<authentication-manager>
     <authentication-provider>
       <user-service properties="/WEB-INF/usersinfo.properties" />
     </authentication-provider>
</authentication-manager>
```

The example properties file shown below, uses the username as the key of the property record while the property value is a comma-separated list of attributes.

The first attribute is the password and the second the user's granted authority.

```
admin=admin1,ADMIN_ROLE
user1=letmein1,USER_ROLE
user=letmein1,disabled,USER_ROLE
```

Persistent Authentication

A community of users which is larger and / or frequently modified would be too cumbersome to manage using the above methods. In this case a database or LDAP store may be used to allow for easy maintenance. Spring security provides built-in support to query user details from the database.

Method Level Security

From version 2.0, Spring Security has improved for adding security to service layer methods. JSR-250 annotation security and the @Secured annotation are available, with expression-based annotations available from 3.0.

<GLOBAL-METHOD-SECURITY>

This element is used to enable annotation-based security through setting the appropriate attributes on the element, and also to group together security pointcut declarations which will be applied across the application context. The elements should only be declared once. The following enables support for @Secured:

```
<global-method-security secured-annotations="enabled" />
```

Adding an annotation to a method (on an class or interface) would then limit the access to that method accordingly. Spring Security's native annotation support defines a set of attributes for the method. These will be passed to the AccessDecisionManager for it to make the actual decision:

```
public interface BankService {

   @Secured("IS_AUTHENTICATED_ANONYMOUSLY")
   public Account readAccount(Long id);

   @Secured("ROLE_TELLER")
   public Account post(Account account, double amount);
}
```

As well as using the tried and tested XML route, annotations can complete the same task. The @EnableGlobalMethodSecurity can take several arguments:

- prePostEnabled: Determines if pre/post annotations should be enabled.
- secureEnabled: Determines if secured annotation should be enabled.
- jsr250Enabled: Determines if JSR-250 annotations should be enabled.

@Secured may be used to define a list of security configuration attributes for business methods. Specify the security requirements (roles/permissions) on a method to limit availability, and then only the user with those roles/permissions can invoke that method. The AccessDenied exception is thrown should the appropriate roles/permissions not be present. This annotation does not support Spring EL expressions.

For a method to be invoked by users in multiple roles, the @PreAuthorize/@PostAuthorize annotations must be used.

```
<global-method-security jsr250-annotations="enabled" />
```

, allows support for JSR-250 annotations. These are standards-based and allow simple role-based constraints to be applied.

```
<global-method-security pre-post-annotations="enabled" />
```

Allows the new expression-based syntax, with the Java being:

```
public interface BankService {

   @PreAuthorize("isAnonymous()")
   public Account readAccount(Long id);

   @PreAuthorize("hasAuthority('ROLE_TELLER')")
   public Account post(Account account, double amount);
}
```

Expression-based annotations are a good choice if you need to define simple rules that go beyond checking the role names against the user's list of authorities.

@PreAuthorize/@PostAuthorize annotations are preferred way for applying method-level security, supports Spring EL, and provide expression-based access control.

@PreAuthorize is suitable for verifying authorization before entering the method. It can take into account the roles/permissions of the users and method arguments.

@PostAuthorize, though seldom used, checks for authorization after method have been executed, so is suitable for verifying authorization on returned values, via the Spring EL provided returnObject object.

Adding Security Pointcuts Using Protect-Pointcut

The use of protect-pointcut is particularly powerful, as it allows you to apply security to many beans with only a simple declaration. Consider the following example:

```
<global-method-security>
<protect-pointcut expression="execution(* com.training.*Service.*(..))"
   access="ROLE_USER"/>
</global-method-security>
```

This will protect all methods on beans declared in the application context whose classes are in the com.mycompany package and whose class names end in "Service". Only users with the ROLE_USER role will be able to invoke these methods. As with URL matching, the most specific matches must come first in the list of pointcuts, as the first matching expression will be used. Security annotations take precedence over pointcuts.

CHAPTER 11

Spring Boot

Introduction

Spring Boot is an interesting addition to the Spring stable. Over time it has become increasingly difficult to just "get going" with a Spring application. Resolving dependencies, choosing runtime containers, and more, all while trying to get to grips with a new framework/technology.

Usually, working examples are needed, and particularly with Spring, where getting up and running is not always easy, even more necessary. Boot is a solution to this conundrum, enabling the creation of stand-alone, production-grade Spring applications.

Spring Boot takes an opinionated view of both the platform and third-party libraries in order to get up and running quickly, with little configuration.

Applications built with Spring Boot may be started with java -jar, through war deployment, or through a command line tool.

From the documentation, the primary goals of Spring Boot are:

- Provide a radically faster and widely accessible getting started experience for all Spring development.
- Be opinionated out of the box, but get out of the way quickly as requirements start to diverge from the defaults.
- Provide a range of non-functional features that are common to large classes of projects (e.g. embedded servers, security, metrics, health checks, externalized configuration).
- Absolutely no code generation and no requirement for XML configuration.

Spring Boot 2.0.0 is the first major revision of Spring Boot since first release, and now provides support for Spring Framework 5.0.

Spring Boot Project

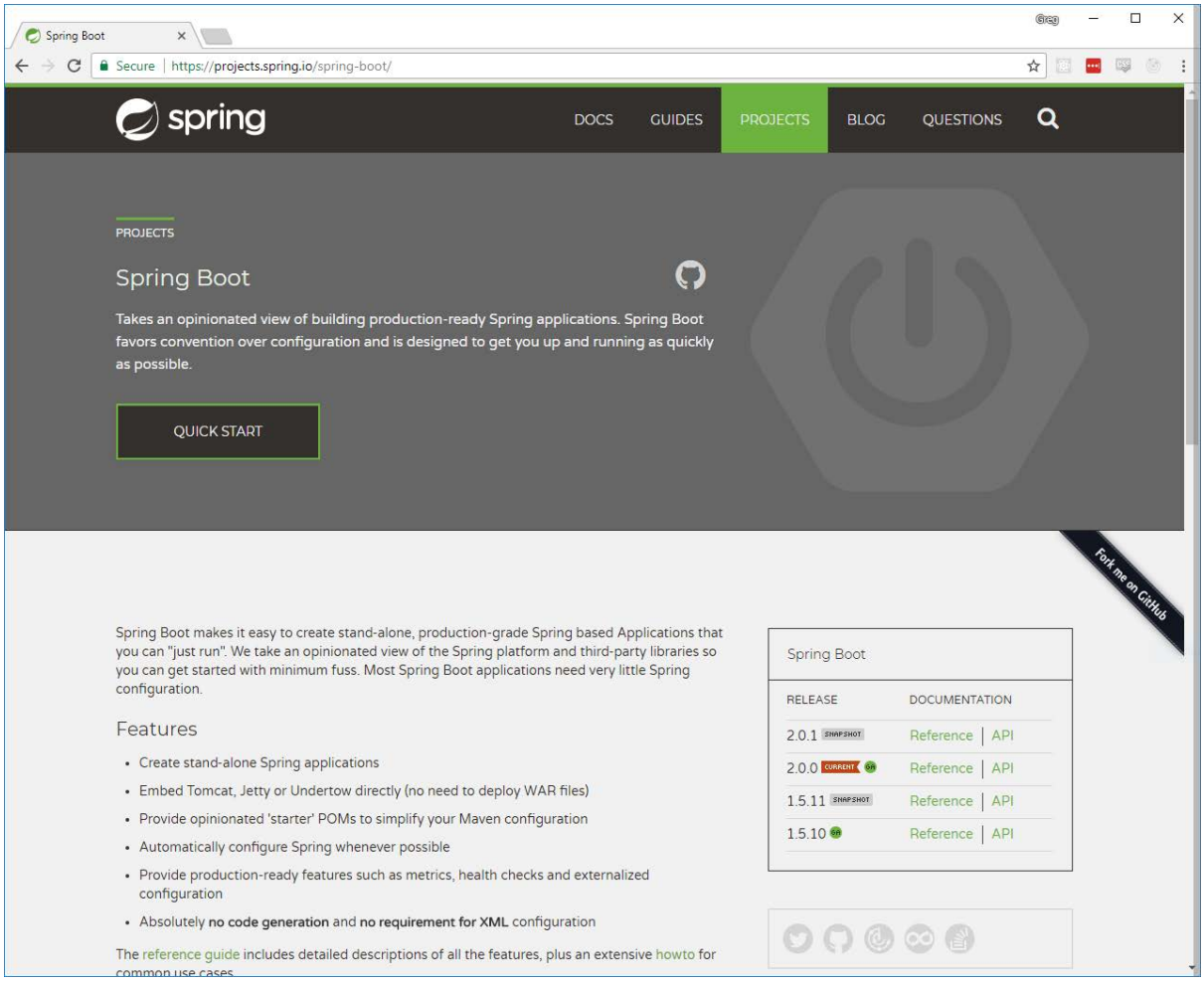
The Spring Boot project is available at:

http://projects.spring.io/spring-boot

Documentation is available at:

http://docs.spring.io/spring-boot/docs

The Spring Boot homepage:



Simplifying Spring Applications

Creating Spring applications can be complex. For instance; to create a web application using Spring, you need to abide by certain rules, particularly a folder structure that:

- Contains WEB-INF (with lib and classes directories)
- Folders/files for JSP, HTML, CSS and JS
- A possible web.xml with the DispatcherServlet mentioned
- A Spring beans XML
- And more

This means Maven or Gradle mucking in for packaging and creating the WAR, while an application server/container is needed to run the result. STS, IntelliJ, Yeoman, and even Eclipse try to help but this is where Spring Boot steps in to relieve the developer of the boilerplate.

Things are even more complex when additions such as persistence, messaging and other components are required.

Why Spring Boot?

Spring Boot has many features that make it suitable for creating applications that adhere to the twelve-factor methodology (http://12factor.net), which refers to software commonly delivered as a service. The methodology describes applications that:

- Use declarative formats for setup automation
- Have a clean contract with the underlying operating system
- Are suitable for deployment on modern cloud platforms
- Minimize divergence between development and production
- Scale without significant changes to tooling, architecture, or development practices

Boot also increases productivity by reducing time of development and deployment, and is suitable for creating production-ready Spring applications.

The addition of non--functional requirements such as metrics, health checks, and management, as well as embedded containers makes things easier.

With the advent of microservices; Spring Boot certainly aids in the creation of scalable, available, robust applications, allowing developers to focus on business logic while the Spring Framework takes care of the rest.

Windows Installation

Download the ZIP binary distribution and uncompress it. The latest release at time of writing is 2.0.0:

http://repo.spring.io/release/org/springframework/boot/spring-boot-cli

To allow access to spring.bat or spring scripts:

- Set JAVA_HOME to the location of the Java SDK
- Set SPRING_HOME to the location of the binary distribution
- Ensure that the PATH includes the %SPRING_HOME%\bin path

The binary distribution contains a Groovy version, so you are set if you want to run Groovy scripts. Verify the installation using:

```
>spring --version
Spring CLI v2.0.0.RELEASE.
```

You have the Spring Boot CLI, so what's next? In the previous chapter, you saw a simple web application written in Groovy and Java, and the way that you run it is by executing either of the following:

```
>spring run *.groovy
>spring run *.java
```

While Spring Boot CLI can run the application, it can also create application structure using either of the following:

```
>spring init --build myapp
>spring init --build gradle myapp
```

This uses the web service at https://start.spring.io to create a folder named myapp. Omitting the gradle option defaults to using Maven. Further features are also possible from the command line:

```
>spring init -dweb,data-jpa,h2 --build maven myapp --force
```


Spring Boot with Maven

The following base POM is needed for every Spring Boot application:

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/xsd/maven-4.0.0.xsd">
   <modelVersion>4.0.0</modelVersion>

   <groupId>com.example</groupId>
   <artifactId>newapp</artifactId>
   <version>0.0.1-SNAPSHOT</version>

   <parent>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-starter-parent</artifactId>
      <version>2.0.0.RELEASE</version>
      <relativePath/>
   </parent>

   <properties>
      <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
      <project.reporting.outputEncoding>
      UTF-8</project.reporting.outputEncoding>
      <java.version>1.8</java.version>
   </properties>

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

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

```
            </plugin>
         </plugins>
      </build>

   </project>
```

The <parent/> section includes the spring-boot-starter-parent artifact, which contains all the application needs to run, such as spring-core.

The starter poms are for the declaration of the dependencies of those features to be used in the application. Starter POMs accrue the application dependencies, so for a web application all that is required is the spring-boot-starter-web artifact:

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

This will include all the spring-core, spring-web, spring-webmvc, embedded Tomcat server, and other related libraries.

The final section is the Spring Boot Maven plugin, which aids packaging the application as a JAR or WAR via:

```
>mvn package
```

There are several available goals/tasks, such as:

```
>mvn spring-boot:run
```

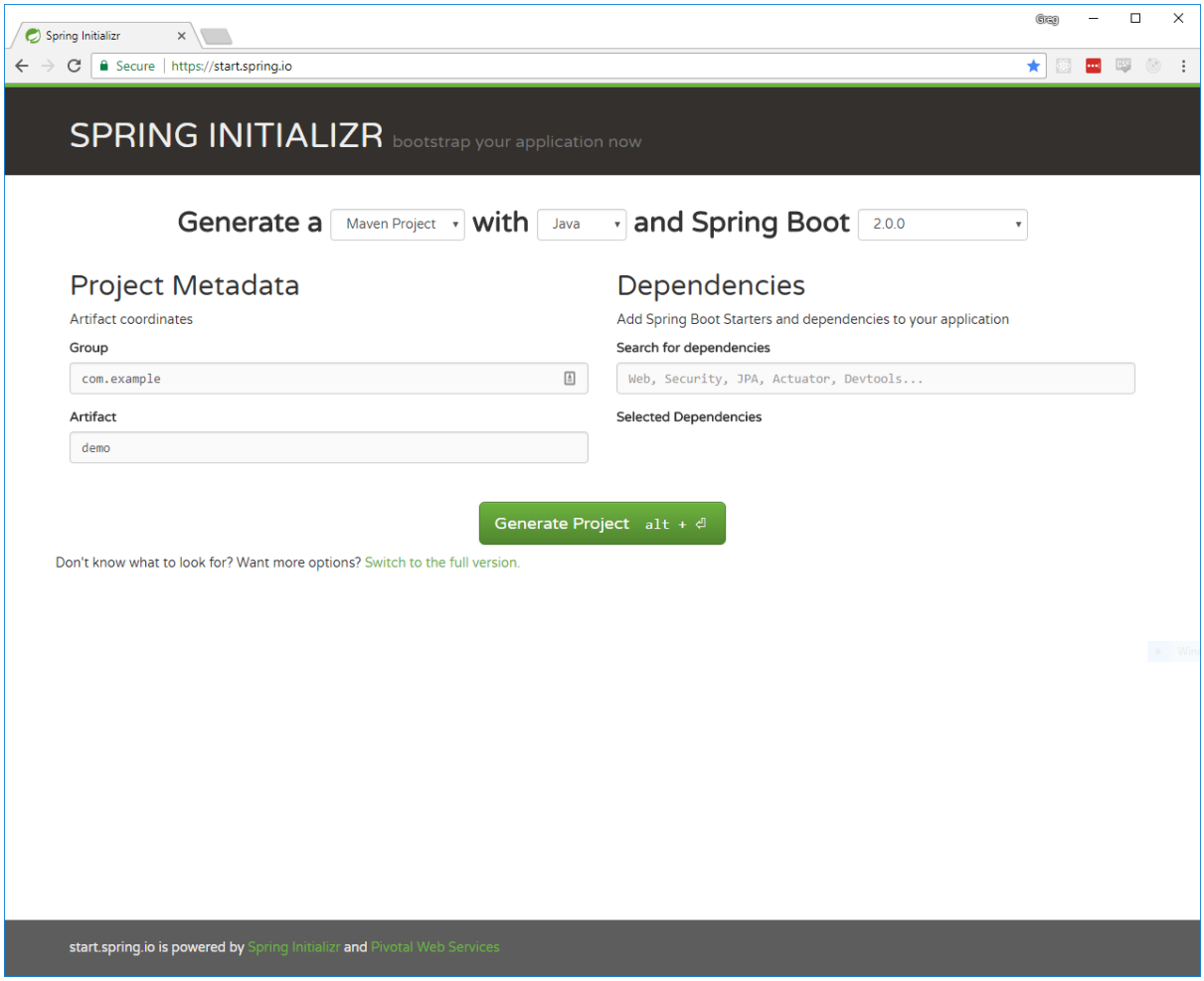
More plugin information is available via:

http://docs.spring.io/spring-boot/docs/2.0.0.RELEASE/maven-plugin/

Using Spring Initializr

The Spring team created a reference architecture tool/service called Spring Initializr as an online service. The service is available at:

http://start.spring.io



The interface allows dependencies to be added via the search box, as well as adding project necessities and choosing either Maven or Gradle as the chosen build processor, before the resulting generated project subsequently being downloaded as a ZIP.

The tool is also available from curl, and can be used in a variety of ways:

```
$ curl -s https://start.spring.io/starter.zip -o newapp.zip
$ curl -s https://start.spring.io/starter.zip -o newapp.zip -d
type=maven- project -d dependencies=web,data-jpa
```

Other options are available from:

```
$ curl start.spring.io
```


Spring Boot Example Application

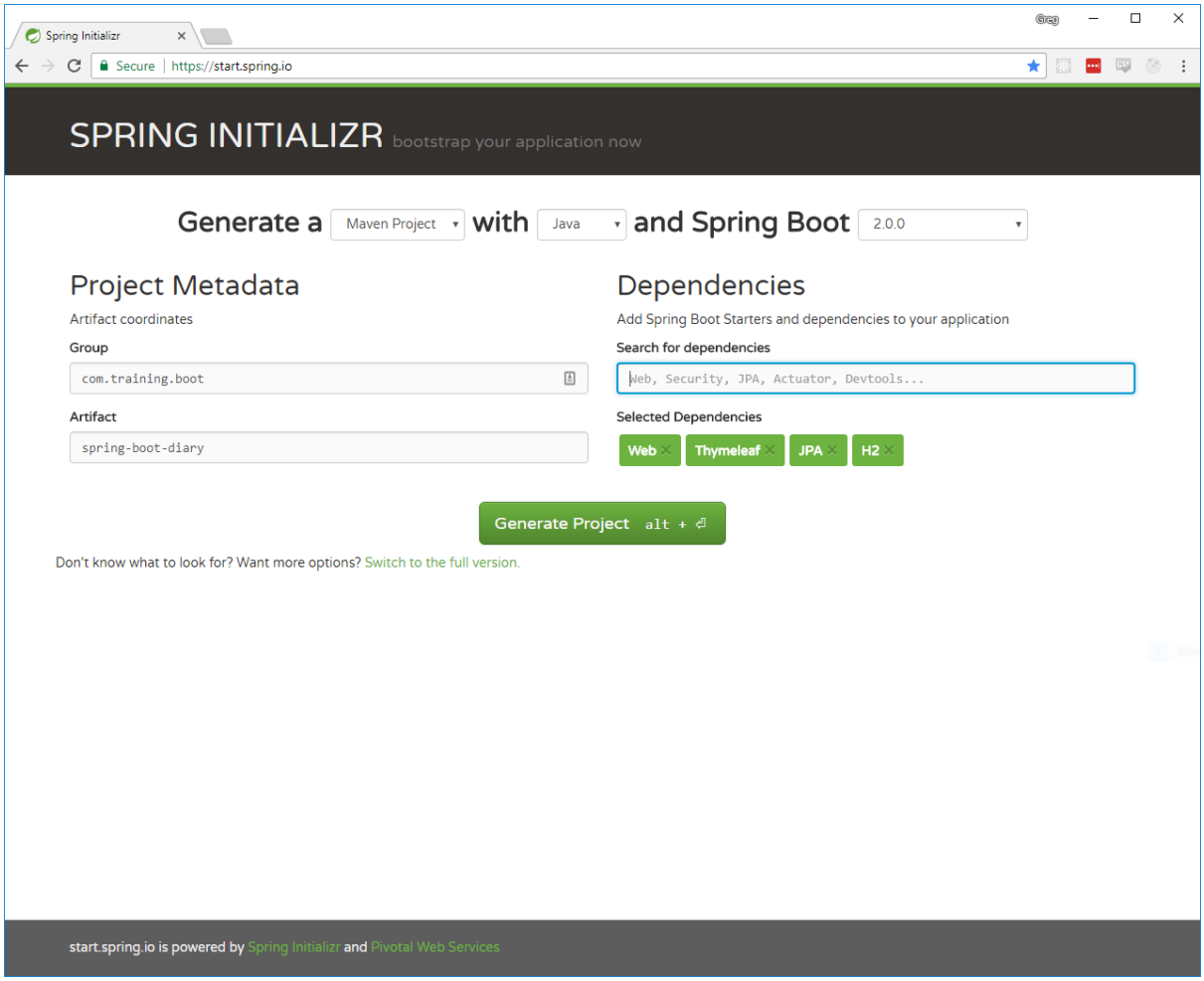
Create an application with the following detail:

- Name spring-boot-diary
- Type Maven
- Packaging JAR
- Group com.training.boot
- Artifact spring-boot-diary
- Package com.training.boot

Choose the following dependencies:

- Web (Web)
- Template Engines (Thymeleaf)
- Data (JPA)
- Database (H2)

If using Spring Initializr, these are the settings:



Import the downloaded project into Eclipse and inspect the POM; the following dependencies should be listed:

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

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


Add a Diary domain class using the JPA annotations:

@Entity, @Id, @GeneratedValue, @Transient

Add two constructors, a no-arg constructor is required by the JPA engine and the other with arguments used to populate the database:

```
@Entity
public class Diary {

   @Id
   @GeneratedValue(strategy = GenerationType.AUTO)
   private Long id;
   private String title;
   private Date created;
   private String summary;
   @Transient
   private SimpleDateFormat format =
      new   SimpleDateFormat("MM/dd/yyyy");

   public Diary() {
      // TODO Auto-generated constructor stub
   }

   public Diary(String title, String summary, String date)
         throws ParseException {
      this.title = title;
      this.summary = summary;
      this.created = format.parse(date);
   }
   ...
```

Spring Data JPA technology is used to create a persistence mechanism. Creating an interface and extending the JpaRepository interface achieves this:

```
public interface DiaryRepository
      extends JpaRepository<Diary, Long> { }
```

This is a marker interface that allows the Spring Data Repository

engine to apply the necessary proxy classes to implement not only the base CRUD (Create, Read, Update, Delete) actions, but also some custom methods.

Actions will then be set as transactional by default.

As the application is a web application, create a web controller:

```
@Controller
public class DiaryController {

   @Autowired
   DiaryRepository repo;

   @RequestMapping("/")
   public String index(Model model) {
      model.addAttribute("diary", repo.findAll());
      return "index";
   }
}
```

This will return all the diary entries.

@Controller is a marker for the Spring MVC engine so this class is treated as web controller.

@Autowired will instantiate the DiaryRepository variable repo, so it can be used in the index method.

The index method is marked with the @RequestMapping annotation, making this method the handler for every request in the default route /. A Model class parameter will be created, adding an attribute named diary with a value that is the result of calling the DiaryRepository interface, repo.findAll() method.

By extending a JpaRepository, there are different default methods, one of them is the findAll method, which will return all the entries from the database. The return will be the name of the page index, then the Spring MVC engine will look for the index.html in the templates folder.

Then in the src/main/resources/templates folder, create the index.html file.

```
<!DOCTYPE html>
<html xmlns:th="http://www.thymeleaf.org">
<head>
<meta charset="utf-8"></meta>
<meta http-equiv="Content-Type" content="text/html"></meta>
<title>Spring Boot Diary</title>
<link rel="stylesheet" type="text/css" media="all"
   href="css/bootstrap.min.css"></link>
<link rel="stylesheet" type="text/css" media="all"
   href="css/bootstrap-glyphicons.css">
</link>
<link rel="stylesheet"
   type="text/css" media="all" href="css/styles.css"></link>
</head>
<body>
<div class="container">
 <h1>Spring Boot Diary</h1>
 <ul class="timeline">
  <div th:each="entry,status : ${diary}">
   <li th:attr="class=${status.odd}?'timeline-inverted':''">
   <div class="tl-circ"></div>
   <div class="timeline-panel">
   <div class="tl-heading">
    <h4><span th:text="${entry.title}">TITLE</span></h4>
    <p><small class="text-muted">
     <i class="glyphicon glyphicon-time"></i>
     <span th:text="${entry.createdAsShort}">CREATED</span>
     </small></p>
   </div>
   <div class="tl-body">
   <p>
   <span th:text="${entry.summary}">SUMMARY</span>
   </p>
   ...
</html>
```

The index is rendered using the Thymeleaf engine, hence the namespace in the html tag. Each th:each instruction fetches diary entries as a collection and iterates to create different tags based on the number of entries, the th:text accesses the property for each entry.

Spring Boot will look for the static/ path to collect all the public files to be exposed to the web: expose JavaScript, images, and CSS in this manner.

Now, the main application in com.training.boot:

```
@SpringBootApplication
public class SpringBootDiaryApplication {

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

Nothing extra is required here; this was generated by Spring Boot.

We do need some data, so add the following to the SpringBootDiaryApplication:

```
@Bean
InitializingBean saveData(DiaryRepository repo) {
   return () -> {
      repo.save(new Diary(
         "Preliminary Reading",
         "Started reading/research", "01/04/2016"));
      repo.save(new Diary(
         "Build Simple Project",
         "First Spring Boot Project",
          "05/04/2016"));
      repo.save(new Diary(
         "Further Boot Research",
         "More Boot", "10/04/2016"));
      repo.save(new Diary(
         "Spring Boot & JAX-RS",
         "Spring Boot using Jersey", "15/04/2016"));
   };
}
```

Exposing the Diary as a service is as simple as adding the following code to the controller:

```
   @RequestMapping(value="/diary",
      produces = {MediaType.APPLICATION_JSON_VALUE})
   public @ResponseBody List<Diary> getDiary(){
      return repo.findAll();
   }
```

Visit the following URL to invoke the service:

http://localhost:8080/diary

Understanding Spring Boot

It's important to note that the simple Boot applications do not use configurations: web.xml, @Configuration classes, or Spring definitions.

The Application class is run, annotated with @SpringBootApplication annotation;

org.springframework.boot.autoconfigure.SpringBootApplication is a composed annotation which contains:

- @Configuration
- @EnableAutoConfiguration
- @ComponentScan

Spring Boot tries to simplify things without the need for a configuration file. The important key for Spring Boot to work is the @EnableAutoConfiguration annotation, because it contains the Auto-Configuration feature.

Spring Boot will use auto-configuration based on classpath, annotations, and configuration to add the right technologies: the annotations facilitate how Spring Boot will configure the application.

First it will inspect the classpath, and because the dependency is a spring-boot-starter-web, it will try to configure the application as a web application. It will also identify that the DiaryController class is a web controller because it is marked with the @Controller and because it contains the @RequestMapping annotations. And

because the spring-boot-starter-web has the Tomcat server as a dependency, Spring Boot will use it when you run your application.

To create a standalone application, at the command line execute:

```
>mvn package
```

This creates a JAR file in the target folder. Execute the JAR using:

```
>java -jar target/spring-boot-diary-0.0.1-SNAPSHOT.jar
```

The application will run on:

http://localhost:8080

CHAPTER 12

Introduction to Spring Cloud

Introduction

Spring Cloud is a gathering of tools that help build common patterns in distributed systems e.g.

- Configuration management
- Service registration and discovery
- Load management
- Service-to-service calls
- Service discovery
- Circuit breakers
- Intelligent routing
- Micro-proxy
- Control bus
- One-time tokens
- Global locks
- Distributed sessions

The idea is that many issues with this systems can be solved with boiler plate patterns that work well in any distributed environment; from a laptop, to bare metal servers, to platforms such as Cloud Foundry.

Spring Cloud takes a declarative approach, and often features are available with a classpath change and/or an annotation. For example, this is a discovery client:

```
@SpringBootApplication
@EnableDiscoveryClient
public class Application {

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

There are multiple projects under the Cloud umbrella, some from Spring themselves, and others that have emerged from Twitter and Netflix.

This section will introduce the core of Spring Cloud, as well as the more prominent projects.

Cloud Native Applications

Spring Cloud is built around the Cloud Native approach, which is itself related to the 12-factor Application approach.

Cloud Native is an approach to building and deploying applications that exploit the advantages of the cloud computing delivery model: exploiting the offer of computing power on-demand, alongside modern data and application services.

Organizations require a platform for building and operating cloud-native applications and services that automates and integrates the concepts of:

DevOps

The collaborative developer-operations approach to enable more consistent build-test-release cycles.

Continuous delivery

Agile approach to automated high-frequency, low-risk software updates.

Microservices

An application as a collection of small, independant services, communicating over HTTP or messaging.

Containers

Containers, such as Docker, rkt and CoreOS offer faster application delivery cycles, better runtime execution, lower infrastructure costs, and simplified deployment. Build once, run anywhere.

Twelve-factor Methodology

The Twelve-Factor App is a website originally created by a co-founder of Heroku, as a manifesto that describes SaaS applications designed to take advantage of the common practices of modern cloud platforms.

The core ideas of a compliant twelve-factor application:

- Use declarative formats for setup automation, to minimize time and cost for new developers joining the project.
- Have a clean contract with the underlying operating system, offering maximum portability between execution environments.
- Are suitable for deployment on modern cloud platforms, obviating the need for servers and systems administration.
- Minimize divergence between development and production, enabling continuous deployment for maximum agility.
- Can scale up without significant changes to tooling, architecture, or development practices.

The constraints that help applications enable to the core ideas above are listed as:

Factor	Description
Codebase	One codebase tracked in revision control, many deployments
Dependencies	Explicitly declare and isolate dependencies
Config	Store config in the environment
Backing services	Treat backing services as attached resources
Build, release, tun	Strictly separate build and run stages
Processes	Execure the app as one or more stateless processes
Port binding	Export services via port binding
Concurrency	Scale out via the process model
Disposability	Maximise robustness with fast startup and graceful shutdown
Dev/prod parity	Keep development, staging, and production as similar as possible
Logs	Treat logs as event streams
Admin processes	Run admin/management tasks as one-off processes

There is depth here, and many books written that expand on each factor/constraint above. More detail can be found at the website: https://12factor.net/

Spring Cloud Context & Commons

The starting point of Spring Cloud is a set of features to which all components in a distributed system need easy access.

Many features are delivered by Spring Boot, which is the basis for Spring Cloud.

Other features are delivered by Spring Cloud as two libraries:

Spring Cloud Context

Utilities and services for the ApplicationContext. Bootstrap context, encryption, refresh scope, and environment endpoints.

A bootstrap context is created, a parent context for the main application responsible for loading configuration properties from external sources and decrypting properties in the local external configuration files.

The bootstrap context uses bootstrap.yml, keeping the external configuration for bootstrap and main context separate. For example:

```
spring:
  application:
    name: foo
  cloud:
    config:
      uri: ${SPRING_CONFIG_URI:http://localhost:8888}
```

Remote properties may be overriden, customised, marked as refreshable, enc/decrypted, and may also have Actuator endpoints.

Spring Cloud Commons

A set of abstractions and common classes used in implementations such as Spring Cloud Netflix and Spring Cloud Consul.

Spring Cloud Config Server

Provides server and client-side support for externalized configuration in a distributed system: a central place to manage external properties for applications across all environments, backed by a git repository.

Full details here: https://cloud.spring.io/spring-cloud-config/

Spring Cloud Config Server has:

- HTTP, resource-based API for external configuration (name-value pairs, or equivalent YAML content)
- Encrypt and decrypt property values (symmetric or asymmetric)
- Embeddable easily in a Spring Boot application using @EnableConfigServer

Config Client features (for Spring applications):

- Bind to the Config Server and initialize Spring Environment with remote property sources
- Encrypt and decrypt property values (symmetric or asymmetric)

Adding the config server to the POM results in a config server running on http://localhost:8888, the default value of spring.cloud.config.uri, which can be altered via system property or env variable.

Once in place, using annotations to access properties results in a call to either the local or remote server:

```
@Configuration
@EnableAutoConfiguration
@RestController
public class Application {

  @Value("${config.details}")
  String details = "Remote property ...";

  @RequestMapping("/")
  public String home() {
    return "Details " + details;
  }
...
```

Spring Cloud Netflix

Netflix OSS integrations for Spring Boot applications. An annotations-driven approach to apply common patterns for distributed systems using Netflix components.

The patterns include:

- **Eureka**, for Service Discovery
- **Zuul**, for Intelligent Routing
- **Hystrix**, as a Circuit Breaker
- **Ribbon**, for Client Side Load Balancing

Netflix Eureka

The Netflix Service Discovery Server and Client. The server can be configured and deployed to be highly available, with each server replicating state about the registered services to the others.

The server is created by including the dependency:

```
<dependency>
   <groupId>org.springframework.cloud</groupId>
   <artifactId>spring-cloud-starter-netflix-eureka-server</artifactId>
</dependency>
```

, and enabling the application at runtime:

```
...
import
org.springframework.cloud.netflix.eureka.server.EnableEurekaServer;

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

A client can then register with the Eureka server and provide metadata such as host, port, health indicator URL, home page etc. The server will then receive heartbeat messages from service instances.

A simple client configuration registers with the server with configuration:

```
spring:
  application:
    name: cloud-2-eureka-client

server:
```



```
    port: 8081

eureka:
  client:
    serviceUrl:
      defaultZone: ${EUREKA_URI:http://localhost:8761/eureka}
  instance:
    preferIpAddress: true
```

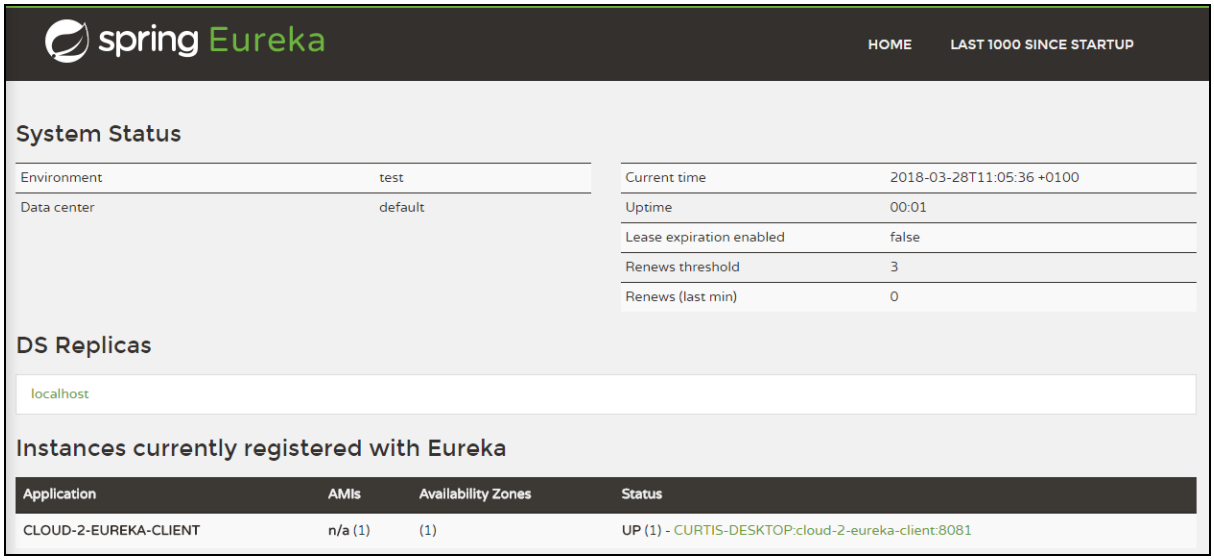
, and enables the client with annotations:

```
...
import com.netflix.discovery.EurekaClient;

@SpringBootApplication
@EnableEurekaClient
@RestController
public class EurekaClientApplication implements GreetingController {
    @Autowired
    @Lazy
    private EurekaClient eurekaClient;

    @Value("${spring.application.name}")
    private String appName;
...
```

Once registered with the server, the server displays instance details:



Netflix Zuul

Zuul is a JVM-based router (primarily Java/Groovy) and server-side load balancer from Netflix, used for:

- Authentication
- Insights
- Stress Testing
- Canary Testing
- Dynamic Routing
- Service Migration
- Load Shedding
- Security
- Static Response handling
- Active/Active traffic managemen

Zuul is a series of filters that action during the routing of HTTP requests and responses; the standard filter types are:

PRE execute before routing to the origin (request authentication, choosing origin servers, and logging debug info).

ROUTING handle routing the request to an origin. This is where the origin HTTP request is built and sent using Apache HttpClient or Netflix Ribbon.

POST execute after the request has been routed to the origin. Can add standard HTTP headers to the response, gathering statistics and metrics, and streaming the response from the origin to the client.

ERROR execute when an error occurs during one of the other phases.

For example, to proxy/filter access to a service, initially add the dependency:

```
<dependency>
   <groupId>org.springframework.cloud</groupId>
   <artifactId>spring-cloud-starter-zuul</artifactId>
</dependency>
```

, and configure Zuul:

```
zuul.routes.people.url=http://localhost:8090
ribbon.eureka.enabled=false
server.port=8080
```

The routes endpoint could be visited directly, as has no direct dependency upon Zuul itself, but here Zuul will act as an intercepting proxy.

Configuring the Zuul server using annotations is straightforward, and the filter can add output simple detail to the console in this case:

```
import org.springframework.cloud.netflix.zuul.EnableZuulProxy;
import org.springframework.context.annotation.Bean;
import hello.filters.pre.SimpleFilter;
```



```
@EnableZuulProxy
@SpringBootApplication
public class GatewayApplication {

  public static void main(String[] args) {
    SpringApplication.run(GatewayApplication.class, args);
  }

  @Bean
  public SimpleFilter simpleFilter() {
    return new SimpleFilter();
  }
}
```

The core details of the filter:

```
...
@Override
public String filterType() {
  return "pre";
}

@Override
public int filterOrder() {
  return 1;
}

@Override
public Object run() {
  RequestContext ctx = RequestContext.getCurrentContext();
  HttpServletRequest request = ctx.getRequest();
  log.info(String.format("%s request to %s",
      request.getMethod(), request.getRequestURL().toString()));
  return null;
}
...
```

With a service available directly on port 8090, visiting indirectly via the Gateway service at localhost:8080/service/endpoint will log to the console before handing over to the application:

```
2018-01-11 11:49:35.084  INFO 10768 --- [nio-8080-exec-8]
hello.filters.pre.SimpleFilter : GET request to
http://localhost:8080/people/available
```

Netflix Hystrix

Hystrix is a library that implements the circuit breaker pattern, where failures in lower-level core services can cascade failure up to the user.

With Hystric in the default setup, service failure reaches a threshold of 20 failures in 5 seconds before the circuit opens and the call fails. This open circuit and failure can be dealt with programmatically with a fallback.

To start, include the dependency:

```
<dependency>
   <groupId>org.springframework.cloud</groupId>
   <artifactId>spring-cloud-starter-netflix-hystrix</artifactId>
</dependency>
```

In a setup where a client service makes requests to an server service, the Hystrix library looks for any method annotated with the @HystrixCommand annotation, and wraps that method in a proxy connected to a circuit breaker so that it can be monitored:

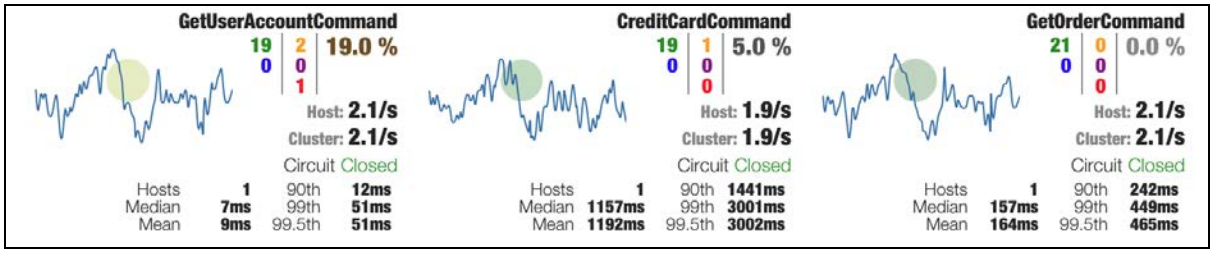
```
@HystrixCommand(fallbackMethod = "localMethod")
public String readingList() {
   URI uri = URI.create("http://localhost:8090/service");
   return this.restTemplate.getForObject(uri, String.class);
}

public String localMethod() {
   return "This will happen as a fallback.";
}
```

Hystrix Dashboard

An application to help visualise event stream monitoring. On the home page is a form where the URL for an event stream to monitor is entered, e.g. http://localhost:9000/hystrix.stream

Applications that use @EnableHystrix will expose the stream.



When a circuit is failing it changes colour, which can help reduce the time needed to discover and recover from operational events.

NetFlix Ribbon

Ribbon is a client-side load balancer, which offers a set of configuration options such as connection timeouts, retries, retry algorithm and more.

Ribbon comes built in with a pluggable and customizable Load Balancing component.

The offered strategies are:

- Simple Round Robin
- Weighted Response Time
- Zone Aware Round Robin
- Random
- Client-Side Load Balancing

To enable client-side load balancing, firstly add the dependency:

```
<dependency>
   <groupId>org.springframework.cloud</groupId>
   <artifactId>spring-cloud-starter-netflix-ribbon</artifactId>
</dependency>
```

Adding a list of servers to application.yml/properties enables the Ribbon client to create an ApplicationContext for each Ribbon client name in the application.

```
say-hello:
  ribbon:
    eureka:
      enabled: false
    listOfServers: localhost:8090,localhost:8091
    ServerListRefreshInterval: 15000
```

This is used to give the client a set of beans for instances of Ribbon components, including:

- **IClientConfig**: stores client configuration for a client or load balancer
- **ILoadBalancer**: represents a software load balancer
- **ServerList**: defines how to get a list of servers to choose from
- **IRule**: describes a load balancing strategy
- **IPing**: how periodic pings of a server are performed

The Ribbon application add some Ribbon-concerned annotations to enable LB:

@RibbonClient

Supply the client name and a class with extra configuration for that client.

@LoadBalanced

Inform Spring Cloud to use load balancing support. Ribbon is on the classpath, so it is used.

StayAhead Training Limited
6 Long Lane
London
EC1A 9HF

020 7600 6116

www.stayahead.com

Terms & Conditions