# TOC

Introduction

What is Spring

Data access with Spring (JPA)

* tangent: caching with Redis
* tangent: user profile management with Spring Data MongoDB’s GridFsTemplate

# Introduction

The hardest part about putting together a tutorial is figuring out its structure - there’s no consensus on which technologies to touch on, which paradigms to introduce. This tutorial's mission statement is “to explain how to build applications with Spring.”

It's easy to look at that mission statement and leap to the proper noun, “Spring.” Spring describes a technology platform, as well as a series of libraries, as well as an ecosystem and a sort of approach, or style, to building applications. Clearly, there's much to discuss there, to clarify, to articulate. Good stuff, too.

For a start, it would be easy to suggest that somebody read the documentation. Spring's documentation is second to none. Good, illustrative documentation is in Spring's DNA – it owes its very existence to two books that Spring’s original creators Rod Johnson and Jürgen Hoeller, among others, wrote! But, the documentation speaks more about how implement the Spring framework, and less so to the process, and to the design choices - the why. Let’s instead tackle learning Spring from the other end; let's talk about building applications.

# Introducing the Application

We’re going to look at a simple application that – while trivial – has enough of the same moving pieces common to most applications as to be relatable. Namely, we’re going to look a web application with Spring MVC that handles basic customer data using services built using Spring’s robust JPA and data access support. Then, we’ll look at securing that application with Spring Security. Along the way, we’ll look at introducing and understanding more advanced concepts like aspect oriented programming, and NoSQL or so-called big-data, and how to work with a cloud like VMware’s Cloud Foundry, an open source platform-as-a-service.

# What is Spring?

We’ve got the requirements, and an idea of the technologies we’re going to use. Let’s talk about what Spring brings to the table. At first blush, Spring’s a container for your objects. Objects that live in the Spring container benefit from the lifecycle hooks and services that Spring provides (if they want them). Spring can also help you wire your objects up, together, so that if one object needs a reference to another, then Spring can ensure that the reference is there. It can do this with regular objects. You don’t need to modify the code to look up the objects in some external service like JNDI. Spring provides a component model.

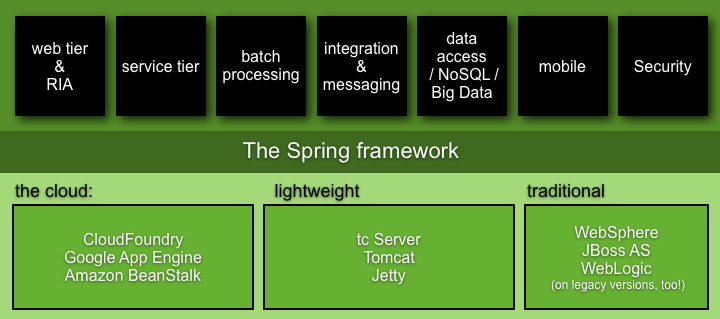
It’s easy to see how this arrangement can be useful. Suppose you want to reference a javax.sql.DataSource. In traditional code, you might construct the DataSource object at the site of the place where you need it. Or, alternatively, you might write code to look up the resource from some external context, like a directory tree. This doesn’t work in larger applications, however. The reference to the DataSource is often needed in many places, and having the creation of the DataSource buried all over the place leads to code that is hard to change. More pragmatically, DataSources are expensive objects to create: it’s much cheaper to create it once, and share it.

Code mired in construction logic is not portable: while I might use a DataSource that points to a database server running on localhost in my local development, my production environment’s database will likely specify very different connection configuration. Some environments require a different DataSource type all together; perhaps you use an in-memory, embedded database for faster iteration in development and then specify something more production-worthy like PostgreSQL or Oracle in production. Similarly, you might use mock versions of objects when running unit tests, and real objects elsewhere. Spring makes it easy to decouple where an object is used from how it is created. This is called dependency injection, which is a form of information hiding, and it promotes cleaner code.

Once you’ve told Spring about your objects, it can transparently apply services to your objects on your behalf, often without requiring any changes to your code. This slight of hand is possible because Java types can always be substituted for the more specialized subclasses of the same type that Spring provides at runtime.

So, Spring describes a component model and it provides dependency injection. Who cares? That, and two dollars will buy me a cup of coffee! Spring’s real power lays in the rich frameworks and libraries that build (and layer) on top of the component model and use dependency injection. SpringSource - the company that develops the Spring framework - offers several other open source frameworks that provide real value to the enterprise developer. You will learn about many of these projects in this tutorial, as they’re all designed to work together. However, you might also consult the open source repository where the code is hosted at github.com/SpringSource for the last word on all the available projects. Let’s look at a short list of some of the technologies and what they provide.

* The Spring framework - provides the core component model and dependency injection libraries. Additionally, it provides integration technologies for interfacing with persistence technologies like JDBC, Hibernate, and JPA. It provides object-to-XML serialization, JMS integration, RPC-style services, and numerous other very handy utilities. Spring core also supports utilities and basic infrastructure code for working with Spring from Servlet environments.
* Spring MVC - Spring MVC is a web application framework, designed to build-based HTTP servlet applications. It includes support for building RESTful web services.
* Spring Web Services - Spring WS is a framework for building contract-first web services. These web services are typically exposed as SOAP-based endpoints. This builds on the core servlet support in Spring core and on the Spring MVC component model.
* Spring Integration - Spring Integration provides an easy way to build event driven architectures and integrations, for many of the same use cases you might have used an Enterprise Service Bus (ESB) or a traditional Enterprise Application Integration (EAI) broker for.
* Spring Batch - Spring Batch helps design long-running batch jobs. This is a common use case that begs for reusable infrastructure code, which Spring Batch provides. The use cases are myriad: perhaps you want to read in records from a file, or iterate through a large database dataset and update them. Most people typically have a few dozen CRON or Autosys jobs in their organization that would be better served by Spring Batch than the homegrown infrastructure code that’s likely already being used.
* Spring Flex - Spring Flex provides an integration of Adobe’s BlazeDS middleware which can be used to expose beans in the Spring container to Flex, Flash and Adobe AIR clients on the web.
* Spring Data - Spring Data is an integration technology that makes working with the new breed of alternative databases - often termed NOSQL. There are several different types of databases including MongoDB, Redis, Riak.
* Spring Social - today’s users live in their social networks and their third party service providers, like Facebook and Netflix and TripIt. Often, your application can extend its reach by providing integration with these third party APIs. Spring Social makes it easy, providing a foundation on which to build clients that support OAuth (1, 1.0.a and 2), the standard that describes the mechanism that most of these APIs use to secure access to the APIs on behalf of the user. Additionally, Spring Social provides numerous APIs that are built on this foundation to work with well-known service providers like Twitter, Facebook, and LinkedIn.
* Spring Security - Spring Security is the most comprehensive security framework available today, providing the ability to restrict access to parts of your application - be it mobile, web, service, or integration flow - by authentication and authorization. The specific strategies used to authenticate and authorize users are pluggable, and the framework ships numerous choices out of the box. You might for example secure RESTful web services using Spring Security OAuth module.



Spring manages your objects for you. To tell it which objects to manage, you need to wire them up; you need to tell Spring about your objects. There are many styles to convey this information to Spring including an XML file format, Java configuration, annotations, the Groovy Bean Builder, a Scala API (or, more specifically, a DSL), etc.

The center of the Spring API is the ApplicationContext hierarchy. ApplicationContext is responsible for managing all the objects under it. There are different subclasses of the ApplicationContext that need to be told how it shall learn about your objects. If you plan on giving it a reference to an XML file on the class path, then you should use the ClassPathXmlApplicationContext. If you want to use the Java and annotation-centric configuration style, then use the AnnotationConfigApplicationContext. If you’d like to use Spring from a servlets environment, and you’d like to use the Java and annotation-centric configuration style, then use the AnnotationConfigWebApplicationContext.

There are numerous others, but they all work basically the same way. Once you have a reference to one, you’ll usually only need to treat it as an ApplicationContext. Spring is in of itself not a runtime. It doesn’t have a startup script. You can use it in a simple main(String[] args) method, or in a unit test, or in a web application, or in a full blown Java EE server, or in any cloud platform available today. The specialized implementations simply let Spring better adapt to, and exploit, the environments in which it runs, but is not something to be concerned with when working with Spring beyond the initial set up.

Here’s how to set up Spring to run in standalone Java SE:

package com.springsource.examples;

import com.springsource.examples.config.AConfigurationClass;

import org.springframework.context.\*;

import org.springframework.context.annotation.\*;

public class EntryPoint {

public static void main(String [] args) throws Throwable {

ApplicationContext ac = new AnnotationConfigApplicationContext(MySimpleConfigurationClass.class);

}

}

Until we start working with web applications, we’ll use this entry point to test and run our code. Later on, we’ll introduce unit testing. Spring is a container, and it manipulates and enhances objects on your behalf. Once you’ve instantiated a Spring ApplicationContext instance, you can obtain a bean instance (of type MyCustomObject, for example) from the ApplicationContext ac, like this:

MyCustomObject mco = ac.getBean(MyCustomObject.class);

Beans may also have an id, which you can use to retrieve a particular instance if there are multiple classes of the same type.

MyCustomObject mco = ac.getBean(“**myCustomBeanId**”, MyCustomObject.class);

If you want to get all instances of a given type, you may ask for them thusly:

MyCustomObject[] mcos = ac.getBeans(MyCustomObject.class);

# Persistence with Java Persistence API (JPA)

## The JPA entities

This application is not expected to see Google-like traffic spikes. So, a standard SQL database should work just fine. We'll work with the open-source, embedded H2 database. H2 is a popular, and open source database that is easy to setup in any environment, and works well for unit testing. Later, we’ll see how we can leverage PostgreSQL on Cloud Foundry. Spring provides rich support for all sorts of databases, including JDBC compliant databases, and the new breed of so-called NOSQL (Not-Only SQL) databases like MongoDB and Redis, both of which we’ll look at later.

There are a lot of ways to interact with data stores. Spring supports numerous options, including JDBC and Hibernate. For our application, we’ll take advantage of the Java Persistence API (JPA), version 2. JPA lets you map business domain objects to Relational Database Modeling System (RDBMS) tables in a fairly transparent way. Let’s look at one entity to see how JPA’s annotations can be layered in to existing domain classes to map those classes to database tables.

Our application lets users (modeled by type User) manage their customers (modeled by type Customer). JPA lets you annotate entities in such a way that JPA can map the object to a relational database table. Here is the User entity. Note that I’ve not reprinted all the getter and setter methods in the name of brevity.

package org.springsource.examples.spring31.services;

import javax.persistence.\*;

import java.util.Date;

import java.util.HashSet;

import java.util.Set;

@Entity

public class User {

**@Id**

**@GeneratedValue(strategy = GenerationType.AUTO)**

private Integer id;

**@OneToMany(cascade = CascadeType.ALL, fetch = FetchType.LAZY, mappedBy = "customer")**

private Set<Customer> customers = new HashSet<Customer>();

private String email;

private String password;

private boolean enabled;

private Date signupDate;

}

The user has an ID (which the database will provide on the id field), and various fields that hold the email, password, account status (enabled) and the date that the user signed up for an account (signupDate). The User object in turn contains a collection of all the customers (of type Customer) that it manages. The code for the entity Customer is below, again with getters and setters omitted:

package org.springsource.examples.spring31.services;

import javax.persistence.Entity;

import javax.persistence.GeneratedValue;

import javax.persistence.GenerationType;

import javax.persistence.Id;

import java.io.Serializable;

import java.util.Date;

@Entity

public class Customer {

@Id

@GeneratedValue(strategy = GenerationType.AUTO)

private Integer id;

private Date signupDate;

private String firstName;

private String lastName;

}

JPA uses convention to infer the name of the table and columns from the annotated Java class. The names of the table to which the User and the Customer object are mapped to are USERS and CUSTOMERS.