APPUNTI JAVA EE

**Table of Contents**

[TO DO 2](#id.gjdgxs)

[WHAT IS JAVA EE 3](#id.30j0zll)

[Enterprise JavaBeans (EJB) 4](#id.1fob9te)

[The Lifecycle of a Stateless Session Bean 5](#id.3znysh7)

[Annotations 7](#id.2et92p0)

[Stateful Session Bean (SFSB) 7](#id.tyjcwt)

[The lifecycle of a Stateful Session Bean 9](#id.3dy6vkm)

[The Singleton Session Bean 9](#id.1t3h5sf)

[Message-driven Bean (MDB) 10](#id.4d34og8)

[Java Persistence API (JPA) 11](#id.2s8eyo1)

[Entities 11](#id.17dp8vu)

[WEB APPLICATION 13](#id.3rdcrjn)

[Servlet container 14](#id.26in1rg)

[The web application and its constituent components are managed and executed inside the web 14](#id.lnxbz9)

[container, also called a servlet container, which provides additional features to the web application 14](#id.35nkun2)

[such as security. When the web server gets a request for specific functionality that a particular web 14](#id.1ksv4uv)

[component (such as a servlet or a JSP page) can provide, the web server forwards the request to 14](#id.44sinio)

[the servlet container in which the web component resides. All requests for the dynamic content 14](#id.2jxsxqh)

[(that is, all requests to the web component that is responsible for generating the dynamic content) 14](#id.z337ya)

[are mediated by the servlet container, as shown in Figure: 14](#id.3j2qqm3)

[Servlets 15](#id.1y810tw)

[JSP 18](#id.4i7ojhp)

[Struts 2 19](#id.2xcytpi)

[JavaServer Faces 20](#id.1ci93xb)

[Spring 21](#id.3whwml4)

[Wiring 22](#id.2bn6wsx)

[Beans, Beans, the Magic Fruit 22](#id.qsh70q)

[The Spring Life Cycle 22](#id.3as4poj)

[Understanding Bean Scopes 23](#id.1pxezwc)

[Eliminating boilerplate code 25](#id.49x2ik5)

[Spring container 25](#id.2p2csry)

[Application context 26](#id.147n2zr)

[Spring configuration 26](#id.3o7alnk)

[Spring Data 27](#id.23ckvvd)

[DAO 27](#id.ihv636)

[Inversion of Control 27](#id.32hioqz)

[Hibernate 29](#id.1hmsyys)

[Mappings 30](#id.41mghml)

[The Steps Needed to Integrate and Configure Hibernate 30](#id.2grqrue)

[Maven 31](#id.vx1227)

[Maven is a dependency management tool. Maven is a command-line tool that is used to build 31](#id.3fwokq0)

[and package projects and to manage dependencies. It makes life easier for developers working 31](#id.1v1yuxt)

[across multiple projects by providing the same directory structure across multiple projects. 31](#id.4f1mdlm)

[OTHER THINGS TO SEARCH 32](#id.2u6wntf)

# TO DO

* Page 24 of “Spring persistence with Hibernate” about “Dependency injection”

# WHAT IS JAVA EE

Java EE (Enterprise Edition) is a widely used platform containing a set of coordinated technologies for developing, deploying, and managing multi-tier, server-centric applications.

Some of the fundamental components of Java EE include:

* Enterprise JavaBeans (**EJB**): a managed, server-side component architecture to encapsulate the business logic of an application. EJB technology enables rapid and simplified development of distributed, transactional, secure and portable applications based on Java technology.
* Java Persistence API (**JPA**): a framework that allows developers to manage data using object-relational mapping (ORM) in applications built on the Java Platform.

# Enterprise JavaBeans (EJB)

In the late 1990s, as Java was bolsterd by the emergence of separate technologies (such as RMI, JTA, and CORBA) that addressed the enterprise needs of large-scale applications, a need arose for a business component framework that could unify these technologies and incorporate them under a standard component development model. EJB was born to fill this need.

There are two types of enterprise beans:

* Session beans
* Message-driven beans (MDBs)

Completing this, the Java Persistence API (JPA) principally defines the following persistent object type:

* Entities are objects that have unique identities and represent persistent business data.

Session and message-driven beans are EJBs, and they are often referred to collectively as enterprise *beans*.

Most enterprise applications have a number of common requirements such as transactions, security, scalability, and so forth. **Enterprise JavaBeans (EJBs)** allow application developers to focus on implementing business logic, while not having to worry about implementing these requirements. Threre are two types of EJBs, Session Beans and Message-Driven Beans. Session Beans simplify server side business logic implementation.

*NB: Previous versions of J2EE included Entity Beans as well, as of Java EE 5, Entity Beans have been deprecated in favor of the Java Persistence API.*

Session Beans encapsulate business logic for enterprise applicatins. It is a good idea to use session beans when developing enterprise applications, since we as application developers can focus on developing business logic, and noe worry about other enterprise application requirements such as scalability, security, transactions, so on.

There are two types of session beans:

* stateless session beans: general-purpose beans that may be of use to a number of different client applications.
* stateful session beans: client-specific beans.

Stateless Session Beans (SLSB) are composed of a variety of pieces:

* *The XML Deployment Descriptor*: EJB has an optional XML deployment descriptor defined in the META-INF/ejb-jar.xml file of the EJB's JAR file. You can use this descriptor as an alternative to annotations, to augment metadata that is not declared as an annotation, or to **override** an annotation. What's interesting about an XML-only deployment is that Java code may contain no references to any EJB-specific APIs. If you looked at the Java code, you wouldn't even know that it was an EJB.
* *SessionContext*: The javax.ejb.SessionContext interface provides a view into the EJB container's enviroment. The SessionContext object can be used as the bean instance's interface to the EJB container to obtain information about the context of the method invocation call and to provide quick access to various EJB services. A session bean can obtain a reference to its SessionContext by using the @Resource annotation:

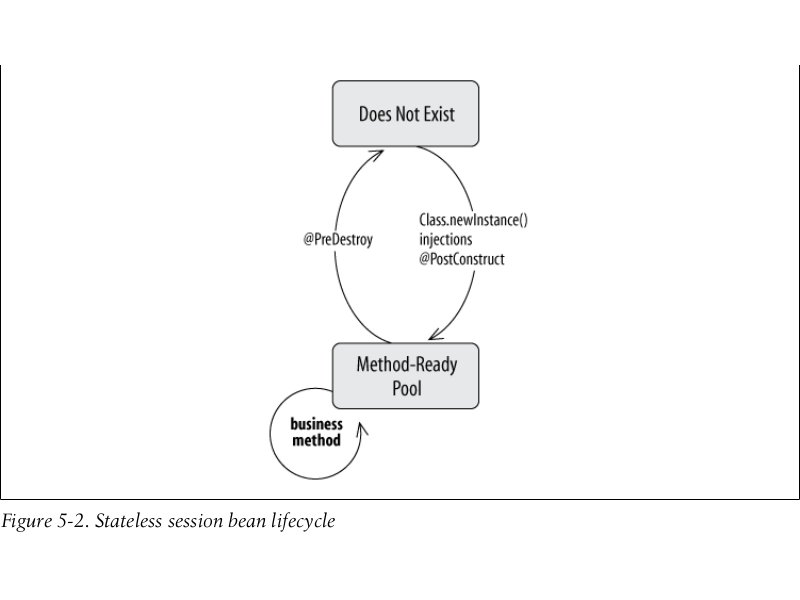
@Resource

private SessionContext context;

* *EJBContext*:

## The Lifecycle of a Stateless Session Bean

The lifecycle of a stateless session bean is very simple. It has only two states: *Does Not Exist* and *Method-Ready Pool*. The Method-Ready Pool is an instance pool of stateless session bean objects that are not in use.

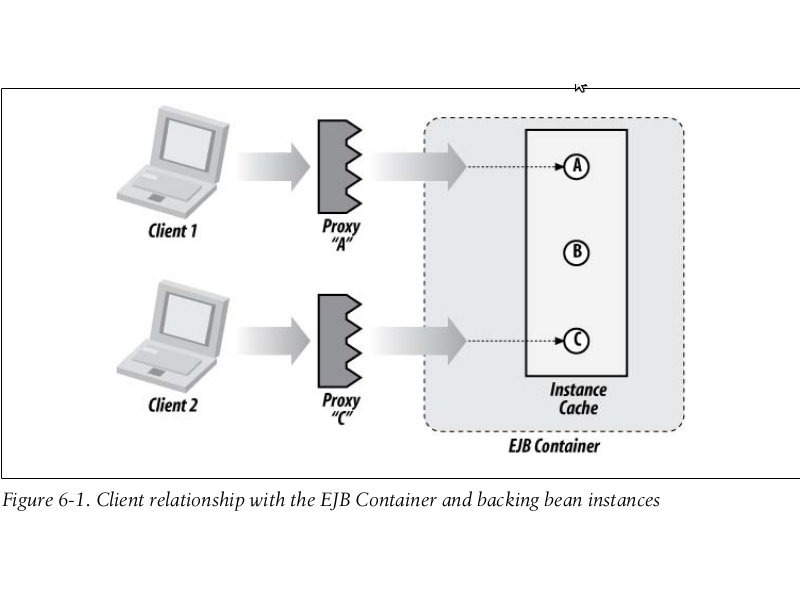


## Annotations

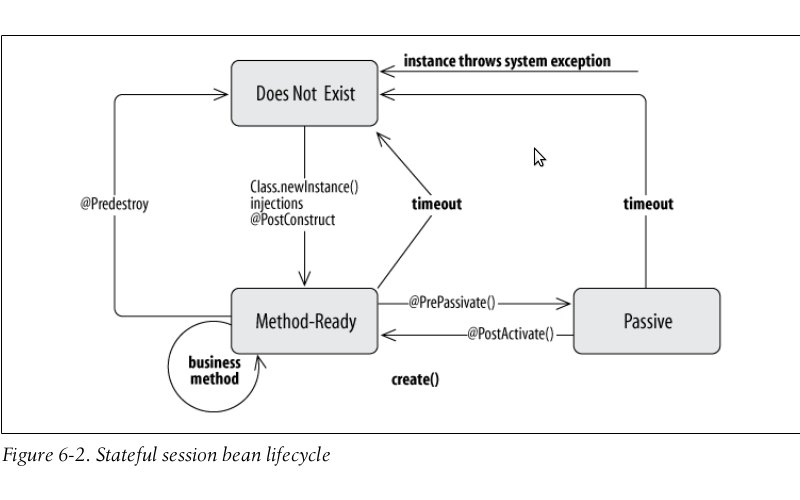
|  |  |
| --- | --- |
| Name | Description |
| @Stateless | Annotation to identify that it is a stateless session bean. |
| @Local |  |
| @Remote |  |
| @PostConstruct | Annotation for method that initialize the bean. |
| @WebServlet | Annotation used to declare a servlet. |
|  |  |

## Stateful Session Bean (SFSB)

While the strenghts of the stateless session bean lie in speed and efficiency, stateful session beans are built as a server-side extention of the client. Each SFSB is dedicated to one client for the life of the bean instance; it acts on behalf of that client as its agent.

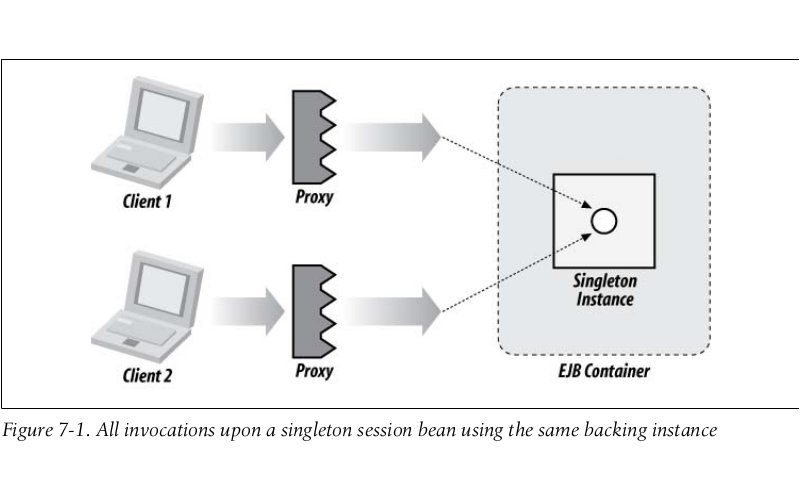
Although stateful session beans maintain conversational state, they are not themselves persistent; the state of a SFSB is lost when the session is removed, the session times out, or the server restarts.

### The lifecycle of a Stateful Session Bean

The lifecycle of a stateful session bean has three states: Does Not Exist, Method-Ready, and Passivated. This sounds a lot like stateless session bean, but the Method-Ready state is significantly different from the Method-Ready Pool of stateless beans.

## The Singleton Session Bean

Singleton session bean is a scheme in which a single shared instance is used for all clients.

Problems with Singleton session bean:

* Due to concurrent invocations, the EJB must be designed as thread-safe.
* Locking or synchronization done to ensure thread safety may result in blocking (waiting) for new requests.

## Message-driven Bean (MDB)

# Java Persistence API (JPA)

Database vocabulary is copletely unknown in an object-oriented languagae such as Java. In java, we manipulate object that are instances of classes. Object inherit from others, have references to collections of other objects, and sometimes point themselves in a recursive manner. We have concrete classes, abstract classes, interfaces, enumerations, annotations, methods, attributes, and so on. Objects encapsulate state and behavior in a nice way, but this state is only accessible when the Java Virtual Machine is running: if the JVM stops or the garbage collector cleans its memory content, objects disappear, as well as their state. Some object need to be persistent. By **persistent data**, I mean data that are deliberately stored in a permanent form on magnetic media, flash memory, and so forth. An object that can store its state to get reused later is said to be persistent.

The principle of **object-relational mapping** (ORM) is to bring the world of databse and objects together. It involves delegating access to relational databases to external tools or frameworks, which in turn give an object-oriented view of relational data, and vice versa. Mapping tools have a bidirectional correspondence between the database and objects. Several frameworks achive this, such as Hibernate, TopLink, anda Java Data Objects (JDO), but Java Persistence API (JPA) is the preferred technology and is part of Java EE 7.

## Entities

When talking about mapping objects to a relational database, persisting objects, or querying objects, the term “entity” should be used rather than “object”. Objects are instances that just live in memory. Entities are objects that live shortly in memory and persistently in a database. They have the ability to be mapped to a database; they can be concrete or abstract; and they support inheritance, relationships, and so on. These entities, once mapped, can be managed by JPA. You can persist an entity in the database, remove it, and query it using a queyr language Java Persistence Query Language, or JPQL.

In the JPA persistence model, an entity is a **Plain Old Java Object** (POJO). This menas an entity is declared, instantiated, and used just like any other Java class.

Simple Example of a Book Entity

**@Entity**

public class Book {

**@Id** @GeneratedValue

private Long id;

private String title;

private Float price;

private String description;

private String isbn;

private Integer nbOfPage;

private Boolean illustrations;

public Book() {

}

// Getters, setters

}

The example represents a Book entity. As you can see, except for some annotations, this entity looks exaclty like any Java class: it has several attributes (id, title, price, etc.) of different types (Long, String, Float, Integer, and Boolean), a default contructor, and getters and setters for each attribute. So how does this map to a table? The answer is thank to *annotations*.

# WEB APPLICATION

(OR WEBAPP): Application that runs on a web server.

A web application is a collection of **web components** that work together to provide a specific

functionality on the Web. In the Java EE specification, a web component is defined to be either a

Servlet or a Java Server Page (JSP) page.

**Note** Other than servlets and JSP pages, a web application may also comprise static resources such

as HTML documents, images, and the metadata or configuration files that define properties of the web

application; however, these are not considered to be web components.

# Servlet container

The web application and its constituent components are managed and executed inside the web

container, also called a servlet container, which provides additional features to the web application

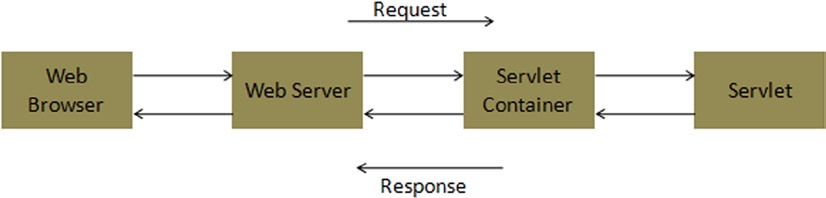
such as security. When the web server gets a request for specific functionality that a particular web

component (such as a servlet or a JSP page) can provide, the web server forwards the request to

the servlet container in which the web component resides. All requests for the dynamic content

(that is, all requests to the web component that is responsible for generating the dynamic content)

are mediated by the servlet container, as shown in Figure:



# Servlets

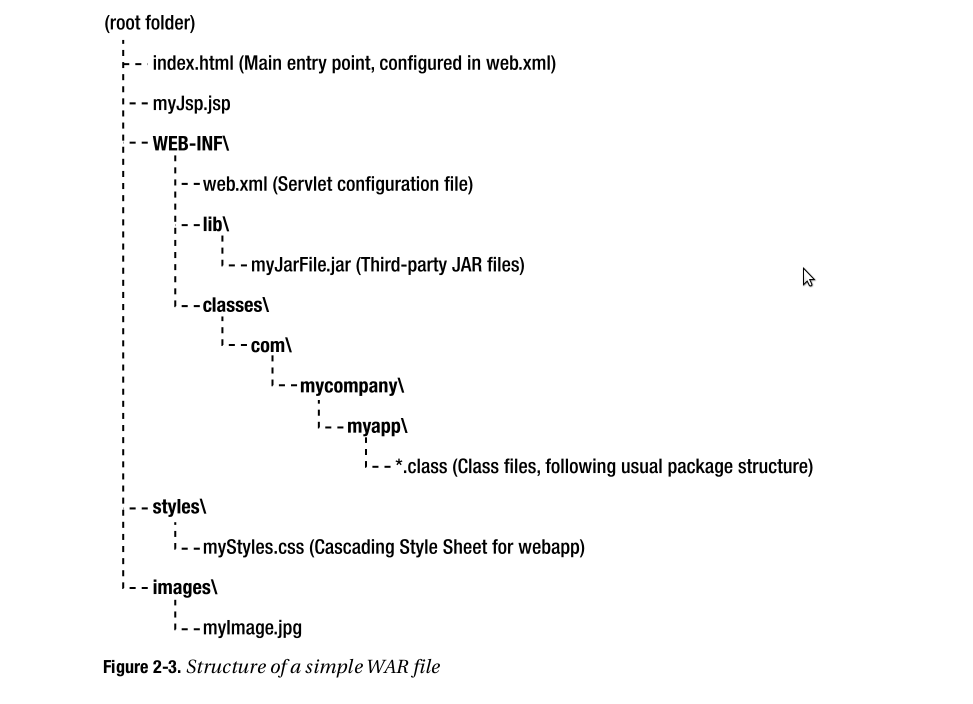
Servlets are Java classes you write in order to handle HTTP requests and responses. These classes are usually subclasses of HttpServlet, which has many convenience functions to help you servlet subclasses do their job.

Servlets run under a web container like Tomcat, GlassFish etc.

A web application is bundled as a WAR file (for Web Application aRchive), which is just an ordinary JAR file with the .war extension. To deploy a web application, you simply drop its WAR file in the *webapps* directory(of the container like Tomcat) and presto! Tomcat automatically installs it for you. You know this because Tomcat will create a subdirectory of webapps with the same name as your web application.

The WAR archive file must contain the following (specifically for Struts but reference to it in general):

* **WEB-INF** directory, into which you put a file called web.xml, used to configure your webapp.
* **WEB-INF\lib** directory, into which you put all Struts JAR files, including third-party JAR files your application uses.
* **WEB-INF\classes** directory, into which go the .class files for your webapp.
* Your JSP, HTML, image, CSS, and other files. JSP and HTML files are usually on the *“root”* of the WAR archive, CSS usually in a *style* subdirectory, and images in a *images* subdirectory.



Important Servlet Classes

You didn't have to know much about serlvets in order to use Struts. This is true, except for two servlet helper classes that are very important to Struts. You need to pay special attention to these. They will certainly be used in your Struts applications:

* **HttpServletRequest**: Used to read parameter values on an incoming HTTP request and read or set “attributes” that you can access from your JSPs.
* **HttpSession**: Represents the current user session. Like HttpServletRequest, HttpSession also has get and set functions for attributes but they are associated with the user session instead of the request.

# JSP

Servlets enable the web server to generate dynamic content. However, servlets have one major

disadvantage in that the HTML code is required to be hardwired in the Java code. To eliminate

this cross-cutting of concerns, the Java Server Pages (JSP) technology was created. JSP uses a

combination of static HTML content and dynamic content to generate web pages, thus separating

the concern of embedding HTML content in Java code.

What really happens behind the scenes is that the JSP container translates the JSP page into a Java Servlet and then compiles the Servlet source code into runnable byte code. The resulting Servlet is then responsible for generating the Web page to send back to the user.

JSP defines six types of tag elements:

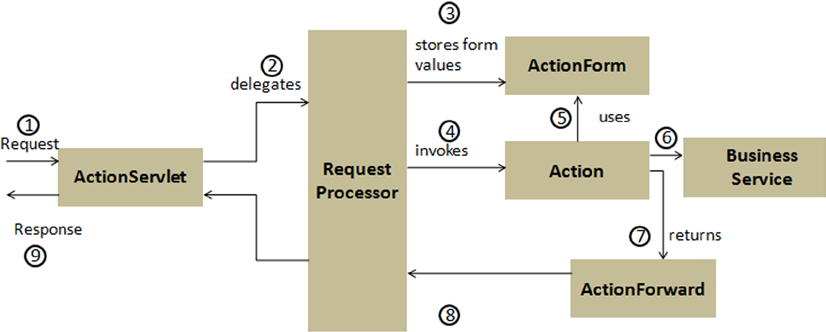
* Action
* Directive
* Declaration
* Expression
* Scriptlet
* Comment

JSP Action tags:

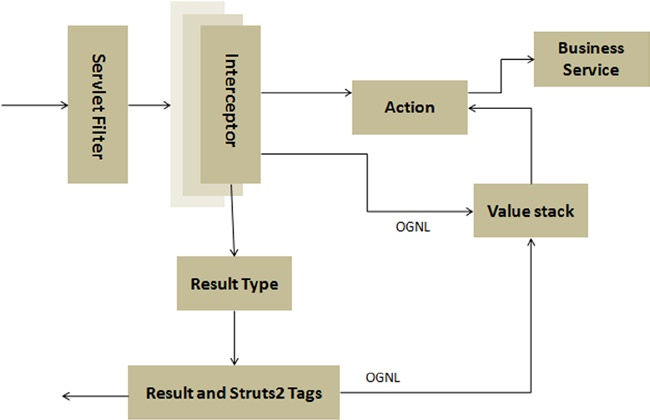
* <jsp:useBean>
* <jsp:setProperty>
* <jsp:getProperty>
* <jsp:param>
* <jsp:include>
* <jsp:foward>

# Struts 2

**Struts schema:**



**Struts 2 schema:**



Struts 2 is also an MVC-based framework that implements the Front

Controller pattern. The sequence of events in Struts 2 framework is as follows:

1. The request is mapped to the configuration metadata.

2. The request passes through a stack of interceptors that provide

preprocessing and postprocessing for the request and cross-cutting features.

3. The Action and the method in the Action that provides the logic to process

this request is invoked.

4. The Result is invoked to render the response.

5. The response is returned to the user.

Struts 2 key elements:

1. Action

Actions are the core of the action-oriented Struts 2 framework because they provide the necessary

logic for request processing. Actions are not required to implement any interface or extend any

class, and actions can be POJOs.

Even though you can use POJO actions, Struts 2 provides two action helpers that can be used: the

**Action interface** and the **ActionSupport class**.

1.1 Action interface

public interface Action {

public static final String ERROR = "error";

public static final String INPUT = "input";

public static final String LOGIN = "login";

public static final String NONE = "none";

public static final String SUCCESS = "success";

public String execute();

}

This interface provides the common string-based return values as constants and the default

execute() method that should be implemented by the implementing classes.

1.2 ActionSupport class

The ActionSupport class implements the Action interface and provides an implementation of the

execute() method that returns the SUCCESS value. The ActionSupport class also implements some

interfaces that provide support for validation, localization, and internationalization.

2. Interceptors

Interceptors promote the separation of concerns by separating the implementation of the cross-cutting

concerns from the action. Struts 2 comes with a set of prebuilt interceptors and interceptor stacks

that you can use out of the box.

3. ValueStack and OGNL

The Object-Graph Navigation Language (OGNL 1 ) is a powerful expression language that is used to

set and get properties from JavaBeans and to invoke methods from Java classes. It also helps in

data transfer and type conversion. OGNL is similar to EL and JSTL, which evaluate expressions and

navigate object graphs using dot notation. As you saw in Figure 4-2, OGNL and the ValueStack,

though not part of MVC, are at the core of the Struts 2 framework. All the MVC components interact

with ValueStack to provide contextual data. These components access ValueStack using OGNL

syntax, and OGNL and ValueStack work together in Struts 2 to handle a request. *Specifically, when*

*a request is sent to the Struts 2 application, a ValueStack object is created for that request, and*

*the references to all the objects that are created to serve that request as well as scope attributes*

*are maintained in the ValueStack.* All these objects are available to the view through OGNL. You will

not find OGNL difficult to use because it is similar to EL and JSTL (which are covered in Chapter 3).

Listing 4-10 illustrates how OGNL looks. Notice that OGNL uses #, unlike JSP EL, which uses $.

Listing 4-10. Using OGNL

<s:property value="#book.bookTitle" />

4. ResultType and Result

# JavaServer Faces

JSF is a framework includes in the JavaEE platform.

From an application developer's point of view, a JSF application consists of a series of XHTML pages containing custom JSF tags, one or more JSF managed beans, and an optional configuration file named *faces-config.xml*.

**Managed Beans**

JSF managed beans are standard JavaBeans that are used to hold user-entered data in JSF applications.

**Facelets**

Facelets is the view declaration language (aka view handler) for JSF. It is the replacement for JSP, which is now retained only for backward compatibility. New features introduced in version 2 of JSF specification are only exposed to page authors using facelets.

Facelets pages are authored usting **XHTML 1.0** and CSS.

Facelets provides *Expression Language* integration. To refere to EL use: **#{espression language}**.

Facelets provides a powerful **templating** system that allows you to provide a consistent look and feel across multiple pages in a web application.

**The web.xml configuration file**

<context-param>

example of parameters:

<context-param>

<param-name>javax.faces.PROJECT\_STAGE</param-name>

<param-value>Development</param-value>

</context-param>

Where param-value could be: Development, Production, SystemTest, UnitTest.

# Spring

Spring is composed of a series of modules. The beauty of this design is that you can pick and choose the components that you would like to use. There's no monolithic JAR file. Instead, you explicity add the components that you want to your project dependencies.

Conceptual keys:

* Inversion of control (realized through Dependency injection)
* Wiring

Spring is used to simplify Java programming in primis by encouraging POJO-oriented development.

Spring does a lot of things. But when you break it down to its core parts, Spring's primary features are dependency injection (DI) and aspect-oriented programming (AOP).

Concepts to know in order to understand Spring:

* POJO → ok
* JavaBean → ok
* EJB → ok
* **Dependency Injection (DI)**: with DI, on the other hand, objects are given their dependencies at creation time by some third party that coordinates each object in the system. Objects aren't expected to create or obtain their dependencies – dependencies are injected into the objects that need them.
* Aspect-oriented Programming AOP)

To back up its attack on Java complexity, Spring employs four key strategies:

* Lightweight and minimally invasive development with plain old Java objects (POJOs);
* Loose coupling through dependency injection and interface orientation;
* Declarative programming through aspects and common conventions;
* Boilerplate reduction through aspects and templates.

Unlike other frameworks Spring avoids (as much as possible) littering your application code with its API. Spring almost never forces you to implement a Spring-specific ingerface or extend a Spring-specific class. Instead, the classes in a Spring-based application often have no indication that they're being used by Spring. At worst, a class may be annotated with one of Spring's annotations, but is otherwise a POJO. (see example pag. 5-6 of Spring in action book).

Unlike many other frameworks such as Struts, which is confined to developing web applications, the Spring Framework can be used to build stand-alone, web, and JEE applications.

## Wiring

Now that your BraveKnight class is written in such a way that you can give him any quest you want, how can you specify which Quest to give him? The act of creating associations between application components is commonly referred to as **wiring**. In Spring, there are many ways to wire components together, but a common approach has always been via XML.

## Beans, Beans, the Magic Fruit

A big part of the secret sauce for the Spring Framework is the use of Plain Old Java Objects, or POJOs. Martin Fowler, Rebecca Persons, and Josh MacKenzie originally coined the term POJO in 2000. POJOs are objects that have no contracts imposed on them; that is, they don't implements interfaces or extend specified classes.

There is often quite a bit of confusion about the differences between JavaBeans and POJOs. The terms tend to be used interchangeably, but that's not always accurate. JavaBeans are best characterized as special kind of POJO. Put simply, a JavaBean is a POJO that follows three simple conventions:

* It is serializable.
* It has a publc, default, and nullary constructor.
* It contains public getters and setters for each property that is to be read or written, respectively (write permissions can be obscured simply by defining a getter, without defining a setter).

The concept of JavaBeans was originally devised for Swing to facilitate the development of stand-alone GUI components, but the pattern has been repurposed for the land of Spring beans and back-end persistence with Hibernate.

Application POJO objects: In Spring, the application objects that are managed

by the Spring IoC container are called beans. A Spring bean is an object that is

instantiated, assembled, and managed by a Spring IoC container.

**Key Objectives of the Spring Framework**

Dependency injection is not the only key benefit of using the Spring Framework. The goal of

the Spring Framework is to simplify the complexity of developing an enterprise application. This

complexity manifests itself in an enterprise application in several ways, and most enterprise

applications prior to the Spring Framework were inadvertently afflicted with few or even all of the

following tribulations:

 Tight coupling

 Cross-cutting concerns

 Boilerplate code

Fundamentally, Spring enables you to build applications from POJOs and apply enterprise services

nonintrusively to POJOs so that the domain model has no dependencies on the framework itself.

Thus, the driving force behind the Spring Framework was to promote best practices in Java EE

development by enabling a POJO-based programming model.

## The Spring Life Cycle

Spring not only instantiates objects and wires up dependencies, but it also handles each managed object's *life cycle*.

## Understanding Bean Scopes

By default, beans defined in Spring are all scoped as singleton. A *singleton* is a class that is guaranteed to have only a single instance in the JVM. Singletons are great for storing application state, or for any case where you want to be assured that there is only ever one reference in your application. Normally, you would need to write code to achieve this assurance.

The typical singleton meets the following criteria:

* Has a static method to return the single instance of the class (stored as a static reference within the class)
* Has a private constructor, ensuring that only the singleton itself can ever create a new instance (which is your assurance that you won't accidentally create more than once instance simply by invoking *new Singleton()*)

A singleton in your application might look like this:

Public class Singleton {

private static final Singleton INSTANCE = new Singleton();

private Singleton() {

}

public static Singleton getInstance() {

return INSTANCE;

}

}

Although the preceding sample illustrates a useful design pattern, Spring obviates the need to write this boilerplate code, once again allowing you to move these details into configuration. By default, all Spring beans are singletons. If this is not your intention, you need to specify a different scope for your bean.

|  |  |
| --- | --- |
| **Scope** | **Description** |
| Singleton | Scopes a single bean definition to a single object instance per Spring IoC container. This is the default scope. |
| Prototype |  |
| Request |  |
| Session | Scopes a single bean definition to the life cycle of an HTTP session. This scope is valid in the context of a web-aware Spring ApplicationContext |
| Global session |  |
| Simple thread |  |

## Eliminating boilerplate code

With Spring we can eliminate boilerplate code(see example on jdbc template).

## Spring container

Spring's job is to parse your configuration files and then instantiate your managed classes, resolving their interdependencies. Spring is often called a *container,* since it is designed to create and manage all the dependencies within your application, serving as a fondation and context through which beans may also be looked up. This core engine is represented by a base interface called **BeanFactory**.

The BeanFactory inteface defines the core Spring engine that conglomerates your beans and wires the collaborationg dependencies together. But the Spring container is capable of much more than just dependecy injection. It can also be used to publish events, provide AOP functionality, support a resource-loading abstraction, facilitate internationalization, and so on. For many of these advanced capabilities, you will need to use an **ApplicationContext** instance.

The ApplicationContext extends the BeanFactory inteface, providing a set of more robust features.

There's no single Spring container. Spring comes with several container implementations that can be categorized into two distinct types. *Bean factories* (defined by the org.springframework.beans.factory.BeanFactory interface) are the simplest of containers, providing basic support for DI. *Application contexts* (defined by the org.springframework.context.ApplicationContext interface) build on the notion of a bean factory by providing application framework services, such as the ability to resolve textual messages from a properties file and the ability to publish application events to interested event listeners.

Altough it's possible to work with Spring using either bean factories or application contexts, bean factories are often too low-level for most applications. Therefore, application contexts are preferred over bean factories.

## Application context

Spring comes with several flavors of application context. The three that you'll most likely encounter are:

* *ClassPathXmlApplicationContext* – Loads a context definition from an XML file located in the classpath, treating context definition files as classpath resources.
* *FileSystemXmlApplicationContext* – Loads a context definition from an XML file in the file system.
* *XmlWebApplicationContext* – Loads context definitions from an XML file contained within a web application.

## Spring configuration

As has been said already, Spring is a container-based framework. But if you don't configure Spring, then it's an empty container and doesn't serve much purpose. We need to configure Spring to tell it what beans it should contain and how to wire those beans so that they can work together.

As of Spring 3.0, there are two ways to configure beans in the Spring container. Traditionally, Spring configuration is defined in one or more XML files. But Spring 3.0 also offers a Java-based configuration option.

The core Spring Framework comes with ten configuration namespaces:

* aop
* beans
* context
* jee
* jms
* lang
* mvc
* oxm
* tx
* util

In addition to the namespaces that come with the Spring Framework, many of the members of the Spring porfolio, such as Spring Security, Spring Web Flow, and Spring Dynamic Modules, also provide their own Spring configuration namespace.

## Spring Data

We could use JDBC, Hibernate, the Java Persistence API (JPA), or any of a number of persistence frameworks. Fortunately, Spring supports all of those persistence mechanisms.

### DAO

DAO stands for data access object, which perfectly describes a DAO's role in an application. DAOs exist to provide a means to read and write data to the database.

### Inversion of Control

Da wikipedia:

In informatica Inversion of Control (IoC – inversione di controllo) è una tecnica di programmazione della programmazione orientata agli oggetti in cui l'accoppiamento tra gli oggetti è realizzato a runtime attraverso un IoC container.

Le dipendenze tra i singoli componenti sono dichiarate in modo semplice. Per esempio una classe Automobile che necessita di un oggetto di tipo Motore, dichiarerà una variabile di istanza di tipo Motore e un metodo per impostare a run-time il riferimento all'oggetto.

Le dipendenze possono quindi essere “iniettate” dall'esterno: non si segue il normale flusso di controllo dei linguaggi imperativi, in cui, nel momento del bisogno, si richiamano funzioni di classi o librerie esterne. Gli oggetti non istanziano e richiamano gli oggetti dal quale il loro lavoro dipende, ma queste funzionalità vengono fornite da un ambiente tramite dei contratti definiti da entrame le entità.

Dependency injection

La Dependency injection è una delle tecniche con le quali si può attuare l'IoC. Essa prende il controllo su tutti gli aspetti di creazione degli oggetti e delle loro dipendenze. La libreria Java Spring usa molto diffusamente la Dependency Injection con il risultato, tra le altre cose, di eliminare dal codice applicativo ogni logica di inizializzazione. Normalmente, senza l'utilizzo di questa tecnica, se un oggetto necessita di accedere ad un particolare servizio, l'oggetto si prende la responsabilità di gestirlo, o avendo un diretto riferimento al servizio, o individuandolo con un “Service Locator” che gli restituisce un riferimento ad una specifica implementazione del servizio. Con l'utilizzo della dependency injection, l'oggetto ha in sè solamente una proprietà che può ospitare un riferimento a quel servizio e, quanto l'oggetto viene istanziato, un riferimento ad una implementazione di questo servizio gli viene iniettata dal framework esterno, senza che il programmatore che crea l'oggetto sappia nulla sul posizionamento del servizio o altri dettagli dello stesso.

Dependency Injection is an application of “Holliwood Principle”: Don't call me, I'll call you.

DEPENDENCY INJECTION

Tip **Constructor-based** and **setter-based DI** can be used simultaneously, but it is recommended to use

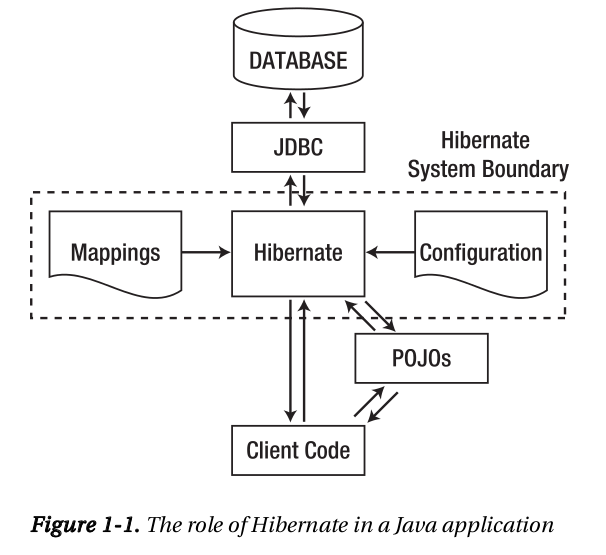
constructor arguments for mandatory dependencies and setters for optional dependencies.

## Spring MVC

To do list to construct a web application using Spring MVC:

1. Informing web.xml of DispatcherServlet: the Spring configuration file can be explicitly specified in the contextConfigLocation servlet parameter to ask Spring to load the configurations besides the default <servletname>-servlet.xml;

# Hibernate



Hibernate was born to manage persistence and to avoid boilerplate code.

Compared to other Java persistence solutions, integrating Hibernate into a Java application is easy. The designers of Hibernate avoided some of the more common pitfalls and problems with the existing Java persistence solutions, and created a clean but powerful ahitecture. In practice, this means that you do not have to run Hibernate inside any particular J2EE container or framework – Hibernate 3.5 only requires Java 2 Standard Edition (J2EE), version 5.0 (or later).

Hibernate uses standard Java Database Connectivity (JDBC) database drivers to access the relational database. Hibernate does not replace JDBC as a database connectivity layer – Hibernate sits on a level above JDBC.

In addition to the standard Java APIs, many Java web and application frameworks now integrate with Hibernate. Hibernate's simple, clean API makes it easy for these frameworks to support Hibernate in one way or another. The Spring framework provides excellent Hibnerate integration, including generic support for persistence objects, a generic set of persistence exceptions, and transaction management. For more detail see Appendix C of “Beginning Hibernate 2nd edition”.

Regardless of the enviroment that you are integrating Hibernate into, certain requirements remain constant. You will need to define the configuration details that apply – these are then represented by a Configuration object. From the Configuration object, a single SessionFactory object is created; and from this, Session objects are instantiated, through which your application accesses Hibernate's rapresentation of the database.

## Mappings

Hibernate needs something to tell it which tables realte to which objects. In Hibernate parlance, this is called a mapping. Mappings can either be provided through Java annotations, or through an XML mapping file. Using annotations gives us a clear picture of the code and what we are trying to accomplish.

## The Steps Needed to Integrate and Configure Hibernate

See page 9 of “Beginning Hibernate 2nd Edition”

**Spring Web MVC**

The center of the Spring Web MVC universe is the DispatcherServlet, a front controller that dispatches requests to registered request handlers.

# Maven

Maven is a dependency management tool. Maven is a command-line tool that is used to build

and package projects and to manage dependencies. It makes life easier for developers working

across multiple projects by providing the same directory structure across multiple projects.

# OTHER THINGS TO SEARCH

* Java Community Process (JCP)
* Java Specification Request (JSR)