# Spring Framework

## Spring Inversion of Control (IoC)-XML Configuration

Inversion of Control is an approach of outsourcing the construction and management of objects

Spring Inversion of Control: Spring provides Object Factory based on configuration file or annotation Spring will give appropriate implementation supporting application.

# Spring Container

Primary functions:

* Create and manage objects (IoC)
* Inject object’s dependencies (Dependency Injection)

# How to Configuring Spring Container

1. XML configuration file (legacy)
2. Java Annotations (modern)
3. Java Source Code (modern)

# Spring Development Process

### Configure Spring Beans

*<beans…>*

*<beans id=”myCoach”*

*class=”com.JavaSpring.BaseballCoach”>*

*</beans>*

*</beans>*

### Create a Spring Container

Spring container is generally known as ApplicationContext

Specialized implementations

* ClassPathXmlApplicationContext

*ClassPathXmlApplicationContext context = new ClassPathXmlapplicationContext(“applicationContext.xml”);*

* AnnotationConfigApplicationcontext
* GenericWebApplicationContext

### Retrieve Beans from Spring Container

*Coach theCoach = context.getBean(“myCoach”, Coach.class);*

“myCoach “ defined in the configuration

“Coach” is interface that BaseballCoach implemented

## Why do we specify the Coach interface in getBean()?

For example:

Coach theCoach = context.getBean("myCoach", Coach.class);

---

**Answer**

When we pass the interface to the method, behind the scenes Spring will cast the object for you.

context.getBean("myCoach", Coach.class)

However, there are some slight differences than normal casting.

From the Spring docs:

Behaves the same as getBean(String), but provides a measure of type safety by throwing a BeanNotOfRequiredTypeException if the bean is not of the required type. This means that ClassCastException can't be thrown on casting the result correctly, as can happen with getBean(String).

# Dependency Injection

The dependency inversion principle. The client delegates to call to another object the responsibility of providing its dependencies. Basically outsourcing contructions and injection of the objects to external entities.

Injection Types

There are many types of injection with Spring. Two most common: Constructor Injection and Setter Injection

## Constructor Injection Development Process

1. Define the dependency interface and class
2. Create a Constructor in your class for injections
3. Configure the dependency injection in Spring config file

**FAQ: What is the purpose for the no arg constructor?**

**Question:**I was wondering why you created a no arg constructor? I thought that they are implied by Java and only required when you also have an overloaded constructor. Or is this a Spring specific thing?

---

**Answered by: Oleksandr Palamarchuk**

When you don’t define any constructor in your class, compiler defines default one for you, however when you declare any constructor (in your example you have already defined a parameterized constructor), compiler doesn’t do it for you.

Since you have defined a constructor in class code, compiler didn’t create default one. While creating object you are invoking default one, which doesn’t exist in class code. Then the code gives an compilation error.

# Specialized Annotation for DAO: @Repository

Applied to DAO implementations

Spring will automatically register the DAO implementation thanks to component-scanning

Spring will provides translation of any JDBC related exceptions

# Web Patient History Spring Hibernate Project

## 1. Setup and Connect to Database

Create database named patient\_history. Then grant access to the database using user=”web\_user” and pass=”web\_user”

Create TestDBServlet.java to connect to the database.

## 2. Setup Development Environment

* Copy starter config files: web.xml and spring config
* Copy over JSTL libs
* Copy latest Spring JAR files <http://www.luv2code.com/downloadspring>
* Copy latest Hibernate JAR files : <http://hibernate.org/orm/releases/5.3/>

## 3. Configuration for Spring + Hibernate (in spring-mvc-crud-demo-servlet.xml)

* Define database dataSource/ Connection pool

***<bean id=”myDataSource” class=”com.mchange.v2.c3p0.ComboPooledDataSource”***

***destroy-method=”close”>***

***<property name=”driverClass” value=”com.mysql.jdbc.Driver” />***

***<property name=”jdbcUrl”***

***value=”jdbc:mysql://localhost:3306/patient\_history?useSSL=false” />***

***<property name=”user” value=”web\_user” />***

***<property name=”password” value=”web\_user” />***

***<!-- these are connection pool properties for C3P0 -->***

***<property name=”minPoolSize” value=”5” />***

***<property name=”maxPoolSize” value=”20” />***

***<property name=”maxIdleTime” value=”30000” />***

***</bean>***

* Setup Hibernate session factory

***<bean id=”sessionFactory”***

***class=”org.springframework.orm.hibernate5.LocalsessionFactoryBean”>***

***<property name=”dataSource” ref=”myDataSource” />***

***<property name=”packagesToScan” value=”com.java\_spring\_hibernate.entiy” />***

***<property name=”hibernateProperties”>***

***<props>***

***<prop key=”hibernate.dialect”>org.hibernate.dialect.MySQLDialect</prop>***

***<prop key=”hibernate.show\_sql”>true</prop>***

***</props>***

***</property>***

***</bean>***

* Setup Hibernate transaction manager

***<bean id=”myTransactionManager”***

***class=”org.springframework.orm.hibernate5.HibernateTransactionManager”>***

***<property name=”sessionFactory” ref=”sessionFactory”/>***

***</bean>***

* Enable configuration of transactional annotations

***<tx:annotation-driven transaction-manager=”myTransactionManager”>***

# 4. Controller

***Controller will receive request from web browser. The controller uses Data Access Object which access to database for us using the Hibernate API to get data. The controller will place data in Spring MVC model and send it to JSP Pages. The JSP Page will render those data to its Spring.***

***Data Access Object (DAO) is a Helper class or Utility class to talk to the database. DAO uses Hibernate API to access the database to get data and send it back to whoever called the method in the controller (Spring MVC PatientController)***

Create a package to hold all controllers and create a PatientController.java class inside the package.

Next create a view folder in WEB-INF which will contains jsp page for instance list-patients.jsp

# 5. List Patients

#### Create entity class called Patient.java. This class is mapped to a database table via Hibernate

#### Create PatientDAO.java and PatientDAOImpl.java

#### Create PatientController.java

#### Create JSP page: list-patients.jsp

# 6. **Create Package to hold all entity**

The entity package name must be matched with the name defined in xml file. Here we use:

***com.java\_spring\_hibernate.entity***

# 7. Define DAO Interface and DAO Implementation

## Hibernate Session Factory

DAO needs Hibernate Session Factory. HSF then needs Data Source, which defines database connection info to connect to database. All of these are dependencies and we will wire them together with Dependency Injection (DI)

# 8. Create Package for DAO

In the package, create interface called PatientDAO

Create an implementation class

# 9. Modifying Controller

Inject dependencies to the controller using @Autowired

get data from DAO

Add data to Model

# 10 Create JSP View Page

# 11 Create Resources Folder in Web Content

Create a resources folder containing sub-folder such as css, js, images, etc..

Add support for reading web resources in spring-mvc-crud-demo-servlet.xml

***<mvc:resources location=”/resources” mapping=”/resources/\*\*”></mvc:resources>***

# Mapping Relationships between Tables

## Many To Many: pharmacies and patients

* Assume that each patient has only one pharmacy associated with him or her.
* However, a pharmacy can hold many patients
* If you delete a patient, DONOT delete pharmacy
* If you delete a pharmacy, DONOT delete the patient
* pharmacies has a foreign key “patient\_id” that point to id in patients table. Thus setup mapping in pharmacies table

## One To Many (bi-directional): patients and medications

* It is obviously a patient has many medications.
* From a patient we can pull out a list of medications. On the other hand, we can add or update medication to a patient. Thus, bi-directional relationship is appropridate.
* And of course, medications will be deleted when patient is no longer present in table.

## Many To Many: patients and physicians

* It is obviously a patient has many doctors including primary doctor or specialist doctors.
* Also, doctor has many patients
* Lazy setup

# View Patient Information

1. Update list-patients.jsp:

add “View Patient Info” link

<!-- Construct an view patient link with patient id -->

<c:url var=*"viewLink"* value=*"/patient/viewPatientInfo"*>

<c:param name=*"patientId"* value=*"*${tempPatient.id }*"*/>

</c:url>

The viewPatientInfo is created in partient controller with specific ID send through param

1. Create view-patient-form.jsp: populate patient info

----------------------------------------------------Debug-------------------------------------------------------------

org.springframework.beans.NotReadablePropertyException: Invalid property 'patient' of bean class [com.java\_spring\_hibernate.entity.Physician]: Bean property 'patient' is not readable or has an invalid getter method: Does the return type of the getter match the parameter type of the setter?