Spring Fundamentals and walkthrough to Advance

By Krishantha Dinesh
(HDIT, PGDIT, MscIT, MBCS, MIEEE)

Prerequisites

- Basic java understanding
- Mobilephone.soundmode=soundmode.completesilent && soundmode.completesilent != soundmode.vibrate

What is Spring

- Originally build to reduce the complexity of Enterprise Java Development
- It is POJO based and interface driven
- Very lightweight compared to old J2EE methodologies
- Build around patterns and best practices (singleton , factory and etc.)

What It resolve ?

- Testability
- Maintainability
- Scalability
- Complexity
- Business focus [complex code in faster]
- Developers can more focus on business need
- Code can focus on testing
- Can remove configuration from codebase

How it is

- Everything in spring is POJO
- Can be consider as glorified hashmap
- Spring can be use as registry
  - It can wired up in a meaningful way in order to deliver results.

Prerequisites

- Java 7 (5 or above)
- Eclipse Juno or later (**Spring sts-3.X.X.RELEASE IDE is Preferred.)**
- spring-framework-3.2.4.RELEASE (or 3.x)

Lets start

- Open eclipse and create new java project
- Give the project name [SpringTraining]
- Create new class under that project
  - Package - `com.krishantha.training.salesmanager.model`
  - Class – `Employee`
- create two properties
  - `String employeeName;`
  - `String employeeLocation;`
- `Create getters and setters for those`

Employee

```java
package com.krishantha.training.salesmanager.model;

public class Employee {

    String employeeName;
    String employeeLocation;

    public String getEmployeeName() {
        return employeeName;
    }

    public void setEmployeeName(String employeeName) {
        this.employeeName = employeeName;
    }

    public String getEmployeeLocation() {
        return employeeLocation;
    }

    public void setEmployeeLocation(String employeeLocation) {
        this.employeeLocation = employeeLocation;
    }

}
```

Repository [DTO / DAO]

- Create new package
  - `com.krishantha.training.salesmanager.repository`
- Create new class
  - `HibernateEmployeeRepositoryImpl`
- With this get a idea that we are going to hide the real implementation from the model / class
- Implement a method to return all employees as a ArrayList

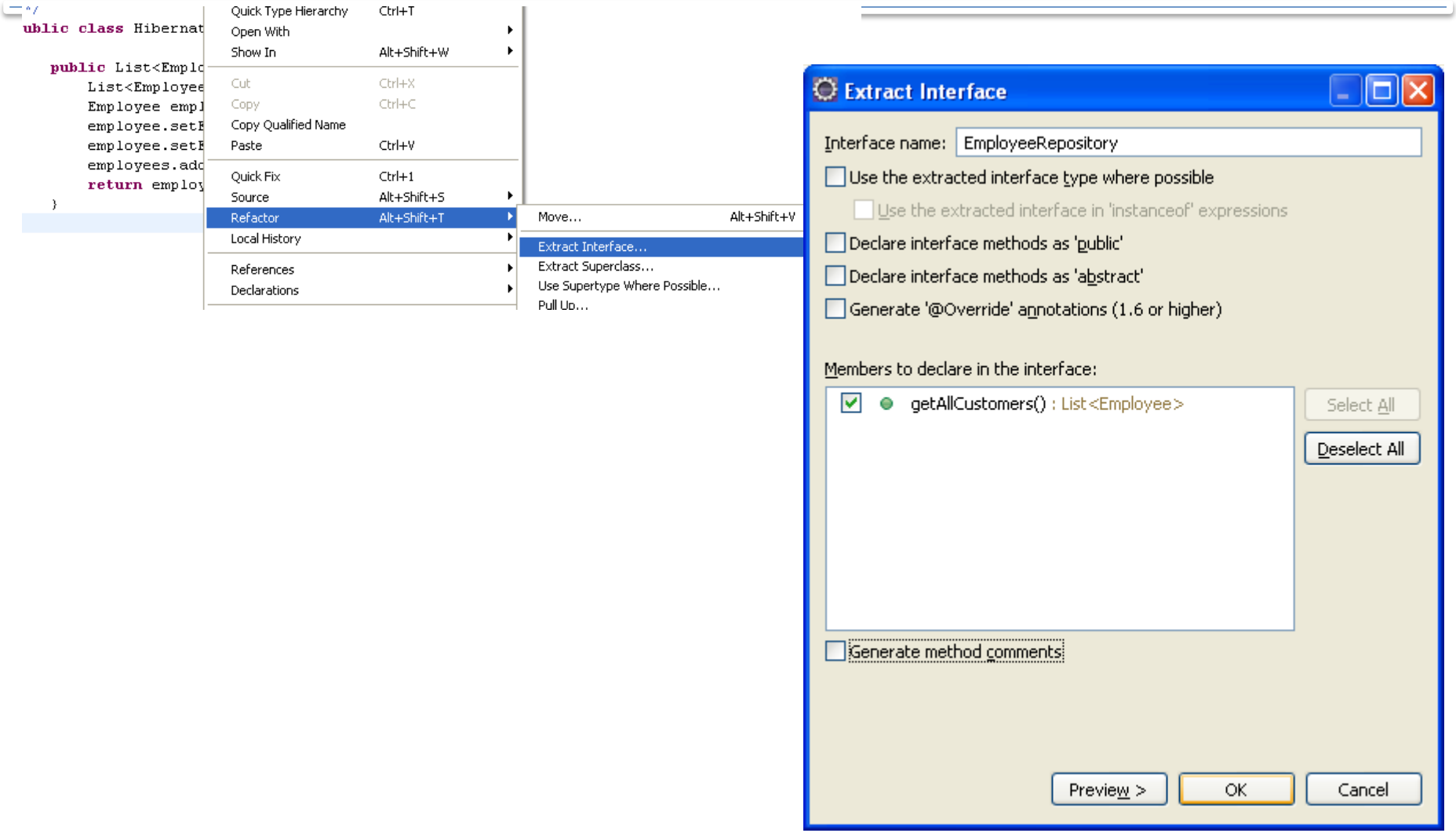
Repository implementation

```
package com.krishantha.training.salesmanager.repository;

import java.util.ArrayList;
import java.util.List;

import com.krishantha.training.salesmanager.model.Employee;

/**
 *
 * @author Krishantha Dinesh
 *
 */

public class HibernateEmployeeRepositoryImpl {

    public List<Employee> getAllEmployees() {
        List<Employee> employees = new ArrayList<>();

        Employee employee = new Employee();
        employee.setEmployeeName("Krishantha");
        employee.setEmployeeLocation("Kadawatha");
        employees.add(employee);
        return employees;

    }

}
```

Generate / Create Interface for repository

- Program to interface not for object
- Using interfaces is a key factor in making your code easily testable in addition to removing unnecessary couplings between your classes
- Support for safety of changes
  - List myList = new ArrayList(); // programming to the List interface
  - This will ensures that you only call methods on myList that are defined by the List interface (so no ArrayList specific methods)
- later on you can decide that you really need
  - List myList = new TreeList();

Refactor the interface



Service

- Create new package
  - `com.krishantha.training.salesmanager.service`
  - Create new class
  - `EmployeeServiceImpl`
- Implement class as per next slide
- Extract the interface from the service class

Service Cont.

```java
package com.krishantha.training.salesmanager.service;

import java.util.List;

import com.krishantha.training.salesmanager.model.Employee;
import com.krishantha.training.salesmanager.repository.EmployeeRepository;
import com.krishantha.training.salesmanager.repository.HibernateEmployeeRepositoryImpl;

public class EmployeeServiceImpl implements EmployeeService {

    EmployeeRepository employeeRepository = new HibernateEmployeeRepositoryImpl();


    public List<Employee> getAllEmployees() {
        return employeeRepository.getAllEmployees();
    }

}
```

```java
package com.krishantha.training.salesmanager.service;

import java.util.List;

public interface EmployeeService {

    List<Employee> getAllEmployees();

}
```

Execute and see results

- Now we have developed a layered application (even we not in very cleared tier separation)
- Create a class with main method and execute it

```
import java.util.List;

import com.krishantha.training.salesmanager.model.Employee;
import com.krishantha.training.salesmanager.service.EmployeeService;
import com.krishantha.training.salesmanager.service.EmployeeServiceImpl;

public class Application {

    public static void main(String[] args) {

        EmployeeService employeeService = new EmployeeServiceImpl();

        List<Employee> employees = employeeService.getAllEmployees();

        for (Employee employee : employees) {
            System.out.println(employee.getEmployeeName() + " at "
                    + employee.getEmployeeLocation());
        }

    }
}
```

Now Dress with Spring ☺

Why configuration code should move out ?

- Make things easy
- Easy to move to different environment
- Easy to unit test and other testing
- Testing hard is not because code complex. It because way code
- Configuration nothing to do with the business flow or logic

Where we went wrong ?

- Service and implementation hardly bind – service should not know that it use hibernate specifically

```
EmployeeRepository employeeRepository = new HibernateEmployeeRepositoryImpl();
```

- Hard coded data on repository

```
public List<Employee> getAllCustomers() {
    List<Employee> employees = new ArrayList<>();
    Employee employee = new Employee();
    employee.setEmpoyeeName("Krishantha");
    employee.setEmploeeLocation("Colombo-ATC");
    employees.add(employee);
    return employees;
}
```

- Interface bind to its concrete class

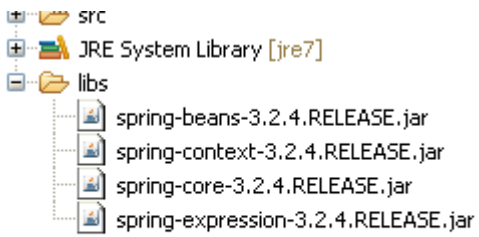
```
EmployeeService employeeService = new EmployeeServiceImpl();
```

Download spring

- Download spring framework. This presentation use the version 3.2.4
- http://repo.springsource.org/libs-release-local/org/springframework/spring/3.2.4.RELEASE/spring-framework-3.2.4.RELEASE-dist.zip

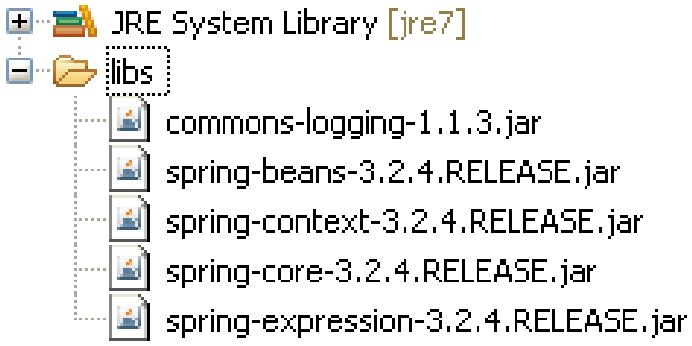
Add spring to projects

- Mainly we need only 4 jar files
  - spring-core-3.2.4.RELEASE
  - spring-context-3.2.4.RELEASE
  - spring-beans-3.2.4.RELEASE
  - spring-expression-3.2.4.RELEASE
- Add those files to your eclipse project from the download location
  - Create new folder call **libs** under project [right click on project -> new -> folder]
  - Drag jar files to that folder and select "COPY" option when ask



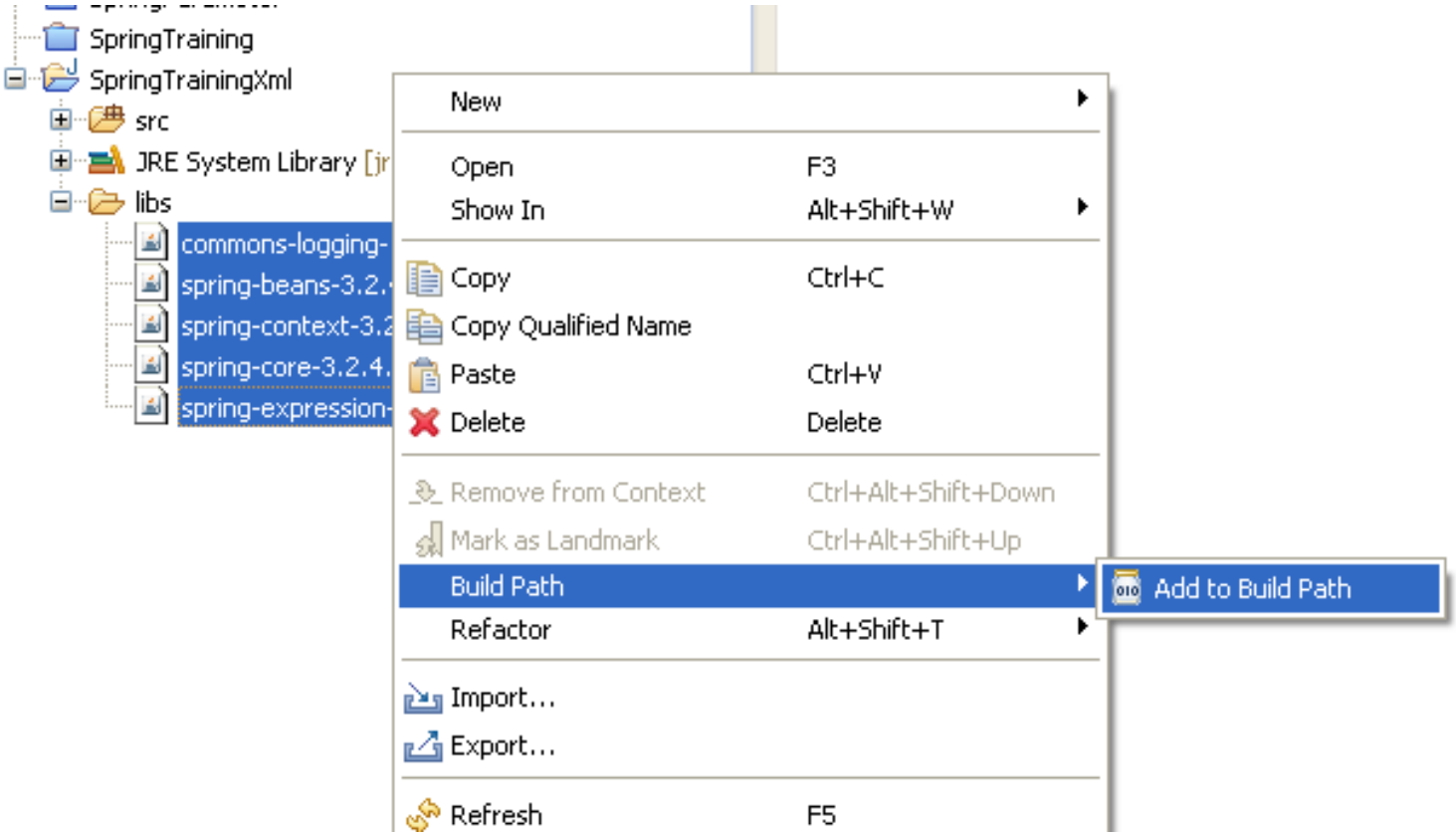
Add commons-logging

- Apache commons logging is platform which used by many open source application to logging.
- It has few jar files however we need only one file
- Download and add it
- http://apache.mirrors.hoobly.com//commons/logging/binaries/commons-logging-1.1.3-bin.zip



Add libs to CLASSPATH

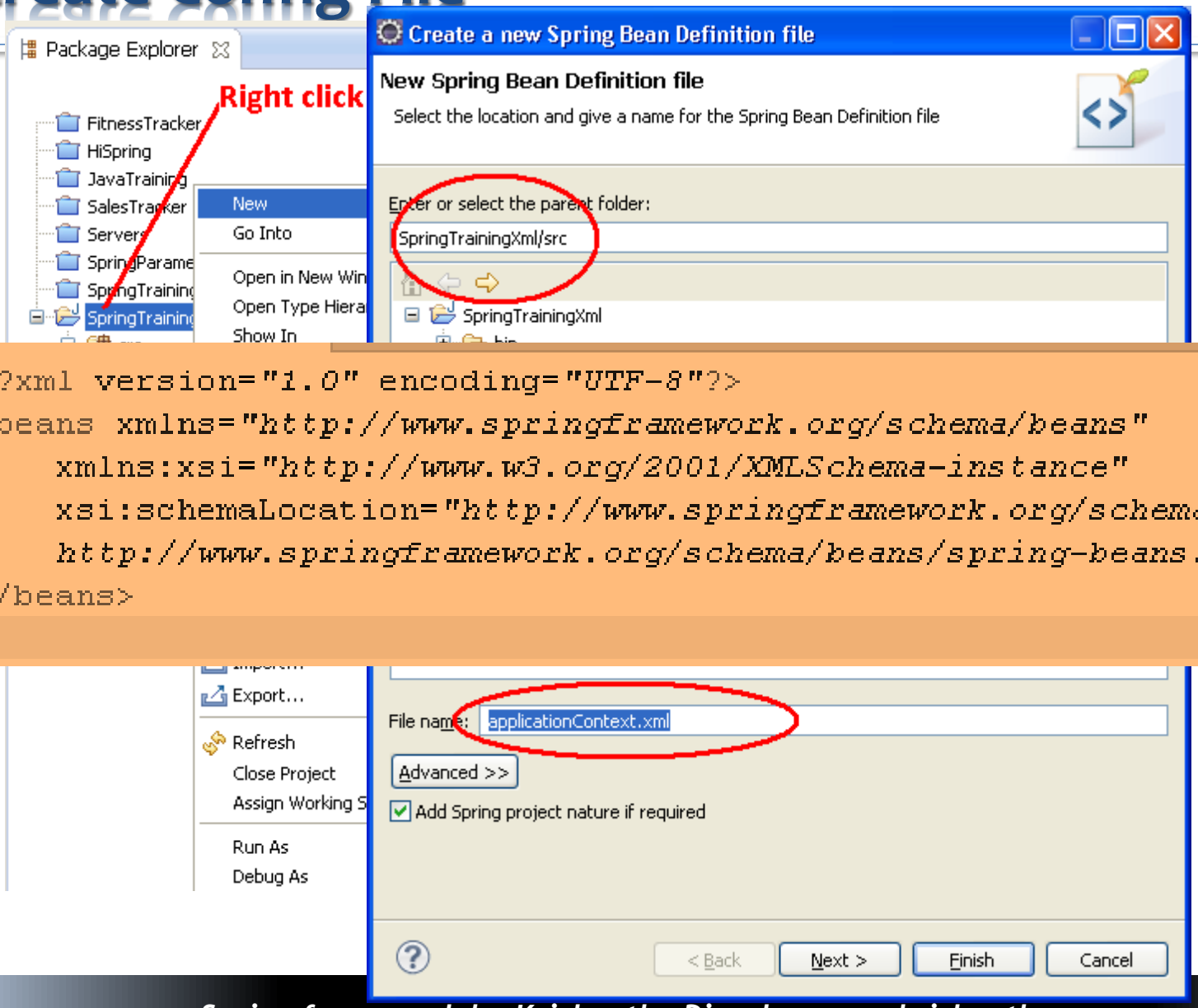
- Add out custom /external jar file to classpath



applicationContext.xml

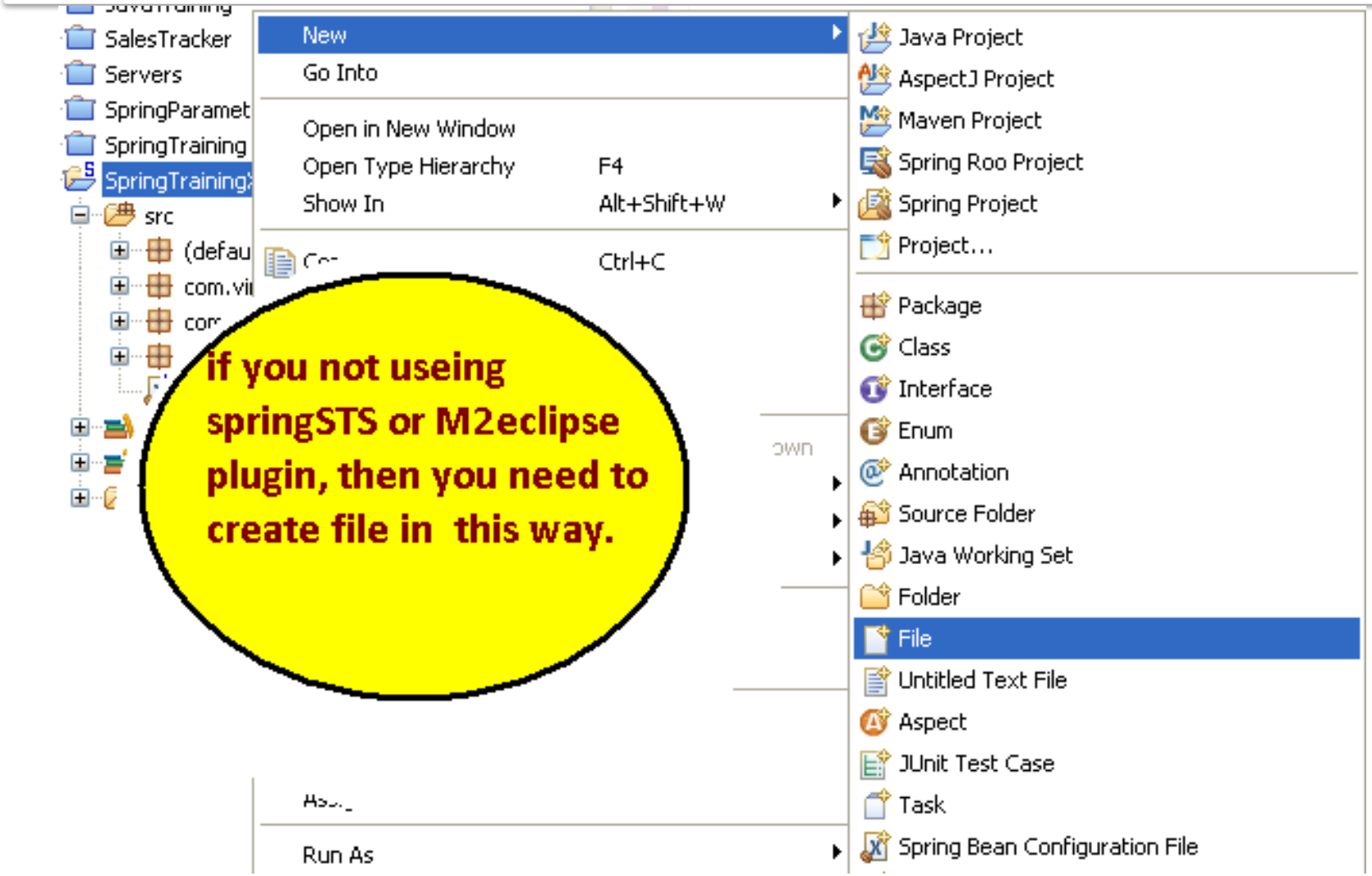
- This is the root of the configuration for the application with xml and spring
- Name can be anything – doesn't have to be applicationContext.xml
- But applicationContext.xml is the standard one
- Spring is kind of hashmap of object. We defined it here
- There are namespaces which help us to do the configuration and validation

Create Config 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">
</beans>
```

No springSTS or m2eclipse plugin



<Xml Based Configuration/>

XML configuration

- Bean
  - Bean (POJO classes)
  - This bean definition will remove NEW keyword
- Constructor arguments
  - used to reference the properties of the constructor
- Properties
  - Getters and setters which defined in POJO
- References
  - Reference to other bean
- Values
- Basic primitive values which use for bean (POJO)

What is bean

- Has to have ID or name
  - Id has to be a valid xml identifier , cannot have special characters
  - Name can contain special characters (but not recommend when goes to spring MVC)
- No-arg constructor (default)
  - Setter injection
  - Constructor injection
- class

Lets create a bean for HibernateEmployeeRepositoryImpl

- Open applicationContext.xml
- Add new bean as we discussed. (name and or id , class etc)
  - `<!-- name can be anything like abc or xyz -->`
- `<bean name="employeeRepository"`
- `class="com.krishantha.training.salesmanager.repository.HibernateEmployeeRepositoryImpl"/>`
  - Since we do not have setter or constructor method at the class we do not need to use injection

Lets create a bean for EmployeeServiceImpl

- It is different than previous
  - `private EmployeeRepository employeeRepository = new HibernateEmployeeRepositoryImpl();`
  - Above red color part make a bond to concrete reference
- Since we do not need to have hardcode concrete references remove that part and code should look like follows.
  - `private EmployeeRepository employeeRepository ;`
- Now generate or write setter method for above property
  - ```
    public void setEmployeeRepository(EmployeeRepository employeeRepository) {
    this.employeeRepository = employeeRepository;
    }
    ```
    ```

Wired up

```xml
<?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">

    <!-- name can be anything like abc or xyz -->
    <bean name="employeeRepository"
        class="com.virtusa.training.spring.repository.HibernateEmployeeRepositoryImpl" />

    <bean name="employeeService"
        class="com.virtusa.training.spring.service.EmployeeServiceImpl">
        <property name="employeeRepository" ref="employeeRepository"></property>

    </bean>
</beans>
```

@ applicationContext.xml

Bean naming convention

```java
public class EmployeeServiceImpl implements EmployeeService {
    //code to interface :)
    private EmployeeRepository employeeRepository ;

    public void setEmployeeRepository(EmployeeRepository employeeRepository) {
        this.employeeRepository = employeeRepository;
    }

    public List<Employee> getAllCustomers(){
        return employeeRepository.getAllCustomers();
    }
```

@ EmployeeServiceImpl

Wiring Done. Now power it up

- Now we have wired the layers using xml
- Now modify the calling app in order to use spring over classic execution.

```
import java.util.List;

public class Application {

    public static void main(String[] args) {

        ApplicationContext applicationContext = new ClassPathXmlApplicationContext(
                "applicationContext.xml");

        EmployeeService employeeService = applicationContext.getBean(
                "employeeService", EmployeeService.class);
        List<Employee> employees = employeeService.getAllEmployees();

        for (Employee employee : employees) {
            System.out.println(employee.getEmployeeName() + " at "
                    + employee.getEmployeeLocation());
        }

    }
}
```

```
Dec 06, 2014 4:39:56 PM org.springframework.context.support.ClassPathXr
INFO: Refreshing org.springframework.context.support.ClassPathXmlApplic
Dec 06, 2014 4:39:56 PM org.springframework.beans.factory.xml.XmlBeanDe
INFO: Loading XML bean definitions from class path resource [applicatic
Dec 06, 2014 4:39:56 PM org.springframework.beans.factory.support.Defau
INFO: Pre-instantiating singletons in org.springframework.beans.factory
Krishantha at Kadawatha
```

Anatomy of springnize

```java
public static void main(String[] args) {
    // EmployeeService employeeService = new EmployeeServiceImpl();
    ApplicationContext applicationContext = new ClassPathXmlApplicationContext("applicationContext.xml");
    EmployeeService employeeService = applicationContext.getBean("employeeService", EmployeeService.class);
    List<Employee> employees = employeeService.getAllCustomers();
    for (Employee employee : employees) {
        System.out.println(employee.getEmpoyeeName() + " - "
                + employee.getEmploeeLocation());

    }
}
```

@ Appication.java

@ applicationContext.xml

interface

```xml
<bean name="employeeService"
      class="com.virtusa.training.spring.service.EmployeeServiceImpl">
      <property name="employeeRepository" ref="employeeRepository"></property>
```

Dependency injection

- Member variable injection
- Constructor injection
- Setter injection

Constructor injection

- Less common than setter injection
- We can guaranteed that no one can call class without setting constructor argument
- Lets change the project from setter injection to constructor injection
  - Create a constructor method on service class
  - Change the applicationContext.xml

Convert to constructor injection

- Change the employeeServiceImpl as follows

```
package com.krishantha.training.salesmanager.service;

import java.util.List;

public class EmployeeServiceImpl implements EmployeeService {

    EmployeeRepository employeeRepository;

    public EmployeeServiceImpl(EmployeeRepository employeeRepository) {
        this.employeeRepository = employeeRepository;
    }

    public EmployeeRepository getEmployeeRepository() {
        return employeeRepository;
    }

    public void setEmployeeRepository(EmployeeRepository employeeRepository) {
        this.employeeRepository = employeeRepository;
    }

    public List<Employee> getAllEmployees() {
        return employeeRepository.getAllEmployees();
    }

}
```

Convert to constructor injection Cont

- Change the applicationContext.xml as follows

```
    <bean name="employeeService"
        class="com.krishantha.training.salesmanager.service.EmployeeServiceImpl">
        <!-- <property name="employeeRepository" ref="employeeRepository" /> -->
        <constructor-arg index="0" ref="employeeRepository"></constructor-arg>
    </bean>

/beans>
```

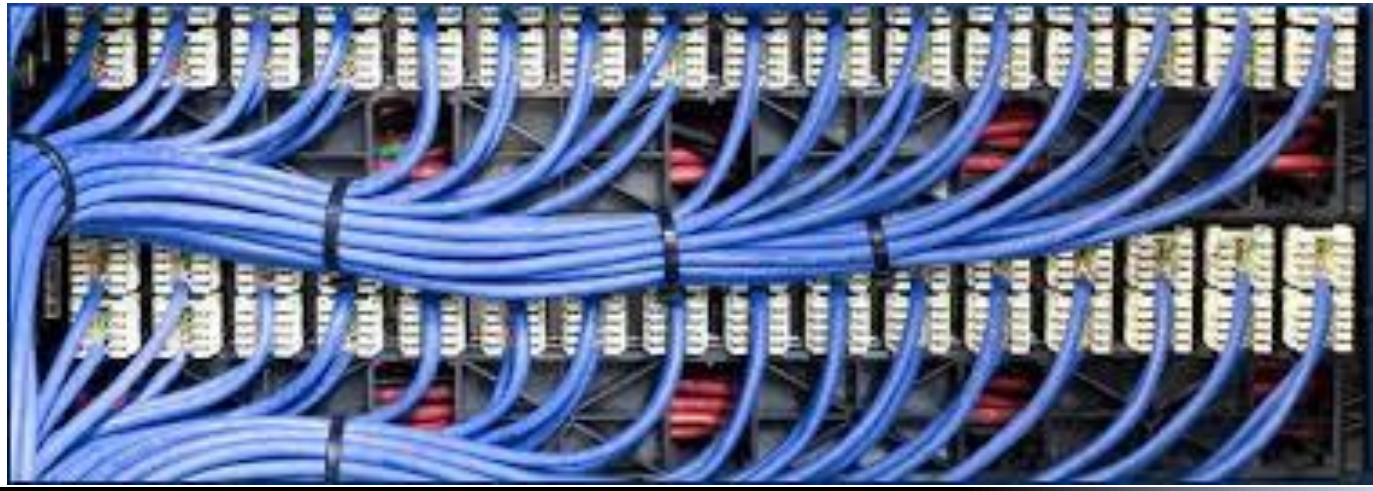
Auto wiring

- Spring can do those wiring automatically instead of do things manually
  1. no: This option is default for spring framework and it means that auto wiring is OFF. You have to explicitly set the dependencies using <property> tags in bean definitions.
  2. byName: This option enables the dependency injection based on bean names. When auto wiring a property in bean, property name is used for searching a matching bean definition in configuration file. If such bean is found, it is injected in property. If no such bean is found, a error is raised.
  3. byType: When autowiring a property in bean, property's class type is used for searching a matching bean definition in configuration file. If such bean is found, it is injected in property. If no such bean is found, a error is raised. Work only if one bean of property type exist on container
  4. constructor: Autowiring by constructor is similar to byType, but applies to constructor arguments. In autowire enabled bean, it will look for class type of constructor arguments, and then do a autowire by type on all constructor arguments. Please note that if there isn't exactly one bean of the constructor argument type in the container, a fatal error is raised.
  5. autodetect: Autowiring by autodetect uses either of two modes i.e. constructor or byType modes. First it will try to look for valid constructor with arguments, If found the constructor mode is chosen. If there is no constructor defined in bean, or explicit default no-args constructor is present, the autowire byType mode is chosen.

Autowire - constructor

- Change applicationContext.xml as follows. If you missed autowire part you will get null pointer exception

```
<bean name="employeeService"
    class="com.krishantha.training.salesmanager.service.EmployeeServiceImpl" autowire="constructor">
    <!-- <property name="employeeRepository" ref="employeeRepository" /> -->
    <!-- <constructor-arg index="0" ref="employeeRepository"></constructor-arg> -->
</bean>
```



Autowire – setter by type

- You must have constructor - **default no argument constructor**
- You must have setter method

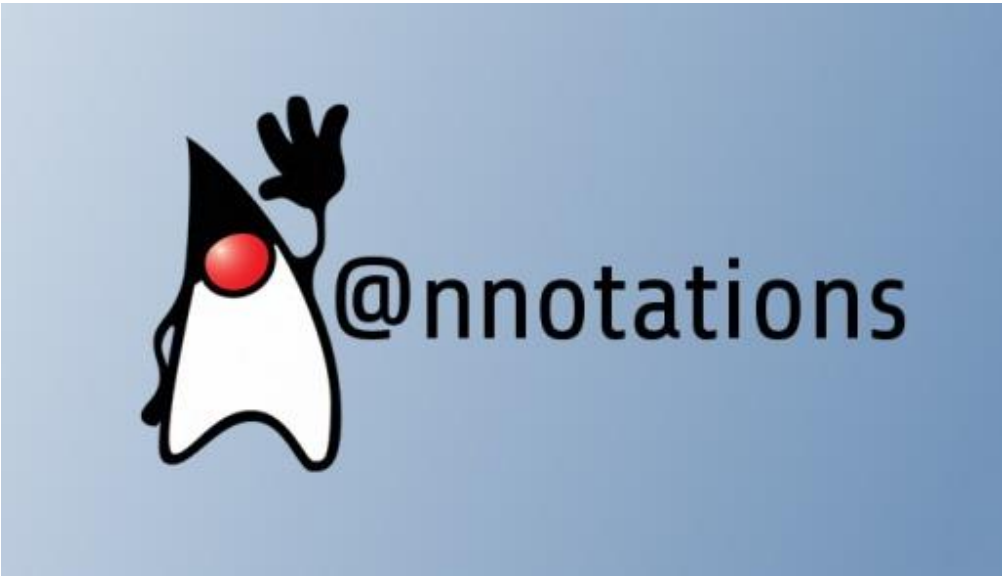
```
<bean name="employeeService"
    class="com.krishantha.training.salesmanager.service.EmployeeServiceImpl" autowire="byType">
    <!-- <property name="employeeRepository" ref="employeeRepository" /> -->
    <!-- <constructor-arg index="0" ref="employeeRepository"></constructor-arg> -->
</bean>
```

```
<bean name="employeeService"
    class="com.krishantha.training.salesmanager.service.EmployeeServiceImpl" autowire="byName">
    <!-- <property name="employeeRepository" ref="employeeRepository" /> -->
    <!-- <constructor-arg index="0" ref="employeeRepository"></constructor-arg> -->
</bean>
```

Annotation based Configuration

Annotation based configuration

- Get a copy of first project
- Paste as SpringTrainingAnotation
- Add spring libraries to build path (please follows the previous example)
- Try and run to make sure it is working



applicationContext.xml

- We going to use applicationContext.xml to bootstrap the annotation scanner
- Under this we need few specific configurations
- We need to add context namespace
- Need to configure to run annotation based
- Configure the component scanner
- Add `context` to the xml file and it will look like

```
<?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"
    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-3.2.xsd">


</beans>
```

applicationContext.xml configuration

- Use following specific configurations
  - `<context:annotation-config/>`
- It will tell that this application is based on annotation based configuration
  - `<context:component-scan base-package="com.krishantha"/>`
  - This will tell that where it should look for annotation mapping

Annotations

- There are 3 major type of annotations
  1. @Component
  2. @Repository
  3. @Service
- @Component can use for any pojo which we going to convert to bean
- @Sevice and @Repository both extend by @Component
- @Service is the layer which put the business logic
- @Repository is DAO layer which use to deal with database with ORM

@Repository

```java
package com.krishantha.training.salesmanager.repository;

import java.util.ArrayList;

/**
 *
 * @author Krishantha Dinesh
 *
 */
@Repository("employeeRepository")
public class HibernateEmployeeRepositoryImpl implements EmployeeRepository {


    public List<Employee> getAllEmployees() {
        List<Employee> employees = new ArrayList<>();

        Employee employee = new Employee();
        employee.setEmployeeName("Krishantha");
        employee.setEmployeeLocation("Kadawatha");
        employees.add(employee);
        return employees;

    }

}
```

same name with first letter simple (standard)

@Service

```java
package com.krishantha.training.salesmanager.service;

import java.util.List;

@Service("employeeService")
public class EmployeeServiceImpl implements EmployeeService {

    EmployeeRepository employeeRepository;

    public EmployeeServiceImpl() {
        // TODO Auto-generated constructor stub
    }

    public EmployeeServiceImpl(EmployeeRepository employeeRepository) {
        this.employeeRepository = employeeRepository;
    }

    public EmployeeRepository getEmployeeRepository() {
        return employeeRepository;
    }

    public void setEmployeeRepository(EmployeeRepository employeeRepository) {
        this.employeeRepository = employeeRepository;
    }

    public List<Employee> getAllEmployees() {
        return employeeRepository.getAllEmployees();
    }

}
```

Auto wired again

- We can use 3 place to inject the annotation based autowire
- Member variable
- Constructor
- Setter
- If we set to Member variable it use reflection to get things done and if we set on constructor it call actual constructor

Member variable injection

```
import java.util.List;

@Service("employeeService")
public class EmployeeServiceImpl implements EmployeeService {

    @Autowired
    EmployeeRepository employeeRepository;

    public EmployeeServiceImpl() {
        // TODO Auto-generated constructor stub
    }

    public EmployeeServiceImpl(EmployeeRepository employeeRepository) {
        this.employeeRepository = employeeRepository;
    }
```

@repository name should be here

Execute application without any change to Application.java

Setter Injection (make sure no-arg constructor)

```
@Service("employeeService")
public class EmployeeServiceImpl implements EmployeeService {


    EmployeeRepository employeeRepository;

    public EmployeeServiceImpl() {
        // TODO Auto-generated constructor stub
    }

    public EmployeeServiceImpl(EmployeeRepository employeeRepository) {
        this.employeeRepository = employeeRepository;
    }

    public EmployeeRepository getEmployeeRepository() {
        return employeeRepository;
    }

    @Autowired
    public void setEmployeeRepository(EmployeeRepository employeeRepository) {
        System.out.println("Setter inection fired");
        this.employeeRepository = employeeRepository;
    }

    public List<Employee> getAllEmployees() {
        return employeeRepository.getAllEmployees();
    }

}
```

Constructor Injection

```
@Service("employeeService")
public class EmployeeServiceImpl implements EmployeeService {


    EmployeeRepository employeeRepository;

    public EmployeeServiceImpl() {
        // TODO Auto-generated constructor stub
        System.out.println("Default constructore executted");
    }
    @Autowired
    public EmployeeServiceImpl(EmployeeRepository employeeRepository) {
        System.out.println("overloaded  constructore executted");
        this.employeeRepository = employeeRepository;
    }

    public EmployeeRepository getEmployeeRepository() {
        return employeeRepository;
    }
```

Try with setter injection plus trace on constructor

```java
@Service("employeeService")
public class EmployeeServiceImpl implements EmployeeService {


    EmployeeRepository employeeRepository;

    public EmployeeServiceImpl() {
        // TODO Auto-generated constructor stub
        System.out.println("Default constructore executted");
    }

    public EmployeeServiceImpl(EmployeeRepository employeeRepository) {
        System.out.println("overloaded  constructore executted");
        this.employeeRepository = employeeRepository;
    }

    public EmployeeRepository getEmployeeRepository() {
        return employeeRepository;
    }

    @Autowired
    public void setEmployeeRepository(EmployeeRepository employeeRepository) {
        System.out.println("Setter inection fired");
        this.employeeRepository = employeeRepository;
    }

    public List<Employee> getAllEmployees() {
        return employeeRepository.getAllEmployees();
    }

}
```

Spring Configuration with Java

Java based configuration

- Copy first project
- Paste as SpringTraingJava
- Add all libs to build path
- Make sure application is running

Why java based configurations

- Do not expect a xml knowledge as a java developer
- Peoples wanted to have no xml or less xml configurations
- From 3.0 spring is support for pure java configurations
- NO applicationContext.xml anymore ☺

Welcome applicationConfiguration.java

- We do not need to have applicationContext.xml
- Create a new java class as ApplicationConfiguration.java
- Use @Configuration annotation on top class
- Methods use @Bean annotation to instances spring beans
- @Bean is method level annotation
- Classes and method name can be anything

Setter Injection

- Simple like normal method call
- No need to worry as xml based setter injection
- This is almost like call setter method

- Before proceed change the service class as follows. Remove concrete implementation

Setter Injection Demo

```java
package com.krishantha.training.salesmanager.service;

import java.util.List;

import com.krishantha.training.salesmanager.model.Employee;
import com.krishantha.training.salesmanager.repository.EmployeeRepository;

public class EmployeeServiceImpl implements EmployeeService {

    EmployeeRepository employeeRepository;

    public EmployeeRepository getEmployeeRepository() {
        return employeeRepository;
    }

    public void setEmployeeRepository(EmployeeRepository employeeRepository) {
        this.employeeRepository = employeeRepository;
    }

    public List<Employee> getAllEmployees() {
        return employeeRepository.getAllEmployees();
    }

}
```

Bean just born

```java
package com.krishantha.training.salesmanager.config;

import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;

import com.krishantha.training.salesmanager.repository.EmployeeRepository;
import com.krishantha.training.salesmanager.repository.HibernateEmployeeRepositoryImpl;
import com.krishantha.training.salesmanager.service.EmployeeService;
import com.krishantha.training.salesmanager.service.EmployeeServiceImpl;

@Configuration
public class ApplicationConfiguration {

    @Bean(name = "employeeService")
    public EmployeeService getEmployeeService() {
        return new EmployeeServiceImpl();
    }

    @Bean(name = "employeeRepositiry")
    public EmployeeRepository getEmployeeRepository() {
        return new HibernateEmployeeRepositoryImpl();
    }

}
```

Change the wiring with setters

```
package com.krishantha.training.salesmanager.config;

import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;

import com.krishantha.training.salesmanager.repository.EmployeeRepository;
import com.krishantha.training.salesmanager.repository.HibernateEmployeeRepositoryImpl;
import com.krishantha.training.salesmanager.service.EmployeeService;
import com.krishantha.training.salesmanager.service.EmployeeServiceImpl;

@Configuration
public class ApplicationConfiguration {

    @Bean(name = "employeeService")
    public EmployeeService getEmployeeService() {

        EmployeeServiceImpl employeeService = new EmployeeServiceImpl();
        employeeService.setEmployeeRepository(getEmployeeRepository());

        return employeeService;
    }

    @Bean(name = "employeeRepositiry")
    public EmployeeRepository getEmployeeRepository() {
        return new HibernateEmployeeRepositoryImpl();
    }

}
```

Calling

```
import java.util.List;

public class Application {

    public static void main(String[] args) {

        ApplicationContext applicationContext = new AnnotationConfigApplicationContext(ApplicationConfiguration.class);

        EmployeeService employeeService = applicationContext.getBean(
                "employeeService", EmployeeService.class);
        List<Employee> employees = employeeService.getAllEmployees();

        for (Employee employee : employees) {
            System.out.println(employee.getEmployeeName() + " at "
                    + employee.getEmployeeLocation());
        }

    }
}
```

Construction injection in Java

- Almost like setter injection
- Some if MAGIC was with spring has been removed (behind seen)
- To demonstrate this add new constructor to service class
- Do not forget to add default constructor – for support for setter injection

```java
package com.krishantha.training.salesmanager.service;

import java.util.List;

import com.krishantha.training.salesmanager.model.Employee;
import com.krishantha.training.salesmanager.repository.EmployeeRepository;

public class EmployeeServiceImpl implements EmployeeService {

    EmployeeRepository employeeRepository;

    public EmployeeServiceImpl(EmployeeRepository employeeRepository) {
        System.out.println("overload constructore executed");
        this.employeeRepository = employeeRepository;
    }

    public EmployeeServiceImpl() {
    System.out.println("default constructore executed");
    }

    public EmployeeRepository getEmployeeRepository() {
        return employeeRepository;
    }
```

Constructor Injected

```
@Configuration
public class ApplicationConfiguration {

    @Bean(name = "employeeService")
    public EmployeeService getEmployeeService() {

        EmployeeServiceImpl employeeService = new EmployeeServiceImpl(
                getEmployeeRepository());

        return employeeService;
    }

    @Bean(name = "employeeRepositiry")
    public EmployeeRepository getEmployeeRepository() {
        return new HibernateEmployeeRepositoryImpl();
    }

}
```

AutoWired on java based

- Add @ComponentScan annotation
- It is almost like to @ComponentScanner at annotation based
- It can handle byName and byType both
- byName refer @Bean name
- byType refer instance type

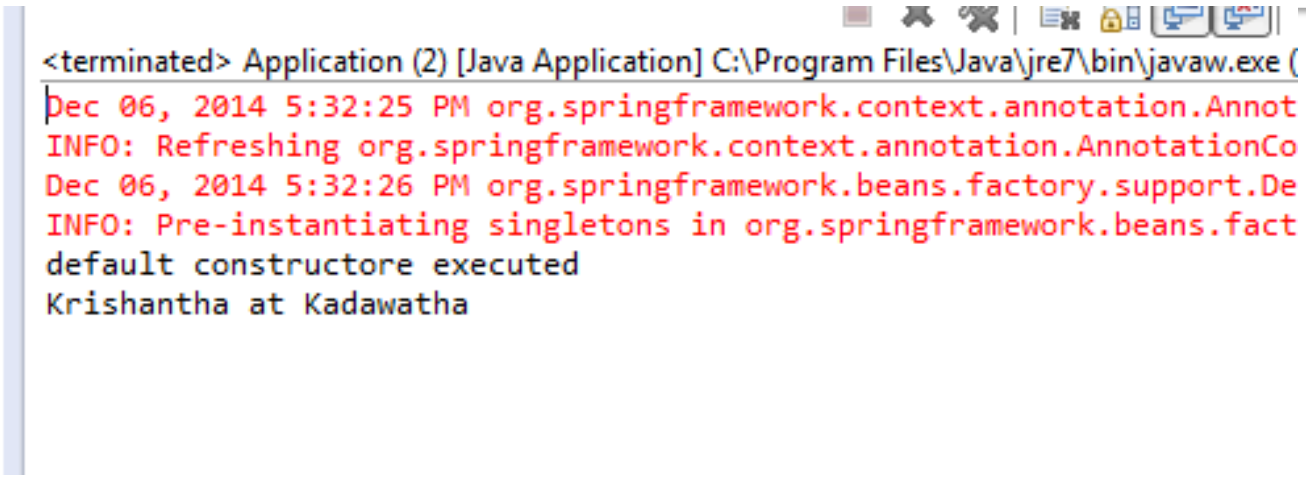
Add @ComponentScanner to Config file

```
@Configuration
@ComponentScan("com.krishantha")
public class ApplicationConfiguration {

    @Bean(name = "employeeService")
    public EmployeeService getEmployeeService() {

        EmployeeServiceImpl employeeService = new EmployeeServiceImpl();

        return employeeService;
    }

    @Bean(name = "employeeRepositiry")
    public EmployeeRepository getEmployeeRepository() {
        return new HibernateEmployeeRepositoryImpl();
    }

}
```

Change the service to auto wired

```
public class EmployeeServiceImpl implements EmployeeService {

    @Autowired
    EmployeeRepository employeeRepository;

    public EmployeeServiceImpl(EmployeeRepository employeeRepository) {
        System.out.println("overload constructore executed");
        this.employeeRepository = employeeRepository;
    }

    public EmployeeServiceImpl() {
        System.out.println("default constructore executed");
    }

    public EmployeeRepository getEmployeeRepository() {
        return employeeRepository;
    }

    public void setEmployeeRepository(EmployeeRepository employeeRepository) {
        System.out.println("setter executed");
        this.employeeRepository = employeeRepository;
    }

    public List<Employee> getAllEmployees() {
        return employeeRepository.getAllEmployees();
    }

}
```

Oppppps……!!!!!



- Its worked but no setter or constructor injection messages….
- Default constructor executed ????????
- **it used Member variable Injection with reflection**

Move @Autowired to setter

```
    }

    public EmployeeRepository getEmployeeRepository() {
        return employeeRepository;
    }
    @Autowired
    public void setEmployeeRepository(EmployeeRepository employeeRepository) {
        System.out.println("setter executed");
        this.employeeRepository = employeeRepository;
    }
```

Console

```
<terminated> Application (2) [Java Application] C:\Program Files\Java\jre7\bin\javaw.exe (Dec 6, 2014, 5:33:59
Dec 06, 2014 5:33:59 PM org.springframework.context.annotation.AnnotationConfigAppl
INFO: Refreshing org.springframework.context.annotation.AnnotationConfigApplication
Dec 06, 2014 5:34:00 PM org.springframework.beans.factory.support.DefaultListableBe
INFO: Pre-instantiating singletons in org.springframework.beans.factory.support.Def
default constructore executed
setter executed
Krishantha at Kadawatha
```

Setter injection in different way

- We can configure setter injection to use @Repository annotation.
- Remove the “employeeRepository” bean
- @Repository to the RepositoryImpl class

```
package com.krishantha.training.salesmanager.repository;

import java.util.ArrayList;

/**
 *
 * @author Krishantha Dinesh
 *
 */
@Repository("you-can-use-anyname")
public class HibernateEmployeeRepositoryImpl implements EmployeeRepository {


    public List<Employee> getAllEmployees() {
        List<Employee> employees = new ArrayList<>();

        Employee employee = new Employee();
        employee.setEmployeeName("Krishantha");
        employee.setEmployeeLocation("Kadawatha");
        employees.add(employee);
        return employees;

    }
```

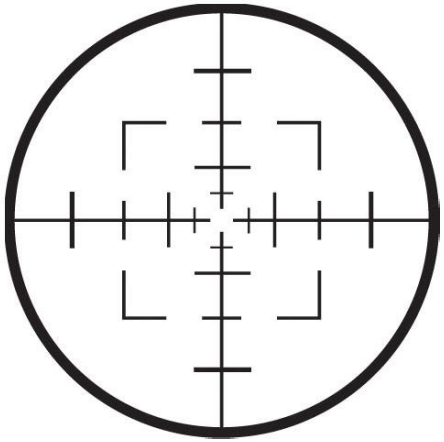
Setter Injection Done

```
package com.krishantha.training.salesmanager.config;

import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.ComponentScan;
import org.springframework.context.annotation.Configuration;

import com.krishantha.training.salesmanager.service.EmployeeService;
import com.krishantha.training.salesmanager.service.EmployeeServiceImpl;

@Configuration
@ComponentScan("com.krishantha")
public class ApplicationConfiguration {

    @Bean(name = "employeeService")
    public EmployeeService getEmployeeService() {

        EmployeeServiceImpl employeeService = new EmployeeServiceImpl();

        return employeeService;
    }
/*
    @Bean(name = "employeeRepositiry")
    public EmployeeRepository getEmployeeRepository() {
        return new HibernateEmployeeRepositoryImpl();
    }
*/
}
```

Bean Scopes Available in Spring

5 - Scopes

- ANY
  - Singleton
  - Prototypes
- WEB
  - Request
  - Session
  - Global



Mr. Singleton

- Default scope
- One instance for spring container [Not for JVM]
- For java based configuration we need aop jar file in classpath
- For xml based configuration we don't need it
- Open ApplicationConfiguration.java and add @scope annotation (add aop lib)

```
@Configuration
@ComponentScan("com.krishantha")
public class ApplicationConfiguration {

    @Bean(name = "employeeService")
    @Scope("singleton")
    public EmployeeService getEmployeeService() {

        EmployeeServiceImpl employeeService = new Emplo

        return employeeService;
    }
/*
```

Singleton With XML [no need aop jar]

```
<!-- name can be anything like abc or xyz -->
<bean name="employeeRepository"
    class="com. . . training.spring.repository.HibernateEmployeeRepositoryImpl" />

<bean name="employeeService"
    class="com..       training.spring.service.EmployeeServiceImpl" autowire="byName" scope="singleton">
    <!-- <property name="employeeRepository" ref="employeeRepository"/> -->
<!--    <constructor-arg index="0" value="employeeRepository"/> -->
```

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

    ApplicationContext applicationContext = new AnnotationConfigApplicationContext(ApplicationConfiguration.class);

    EmployeeService employeeService = applicationContext.getBean(
            "employeeService", EmployeeService.class);

    System.out.println(employeeService.toString());


    EmployeeService employeeService2 = applicationContext.getBean(
            "employeeService", EmployeeService.class);

    System.out.println(employeeService2.toString());

    List<Employee> employees = employeeService.getAllEmployees();

    for (Employee employee : employees) {
        System.out.println(employee.getEmployeeName() + " at "
                + employee.getEmployeeLocation());
    }

}
```

Same instance

```
default constructore executed
setter executed
com.krishantha.training.salesmanager.service.EmployeeServiceImpl@39b6e978
com.krishantha.training.salesmanager.service.EmployeeServiceImpl@39b6e978
Krishantha at Kadawatha
```

Prototype

- Opposite of singleton. Mean we get one instance per request
- Support for both java and xml configurations
- It is guaranteed that you get new request per request

Prototype scope

```
@Configuration
@ComponentScan("com.krishantha")
public class ApplicationConfiguration {

    @Bean(name = "employeeService")
    @Scope("prototype")
    public EmployeeService getEmployeeService

        EmployeeServiceImpl employeeService =

        return employeeService;
    }
```

```
ıted

ılesmanager.service.EmployeeServiceImpl@77fe6f88
ıted

ılesmanager.service.EmployeeServiceImpl@614c8743
```

Other scopes

- Web scope – spring mvc
- Request – return single bean for http request
- Session – single bean for a session
- Global – single bean for a application

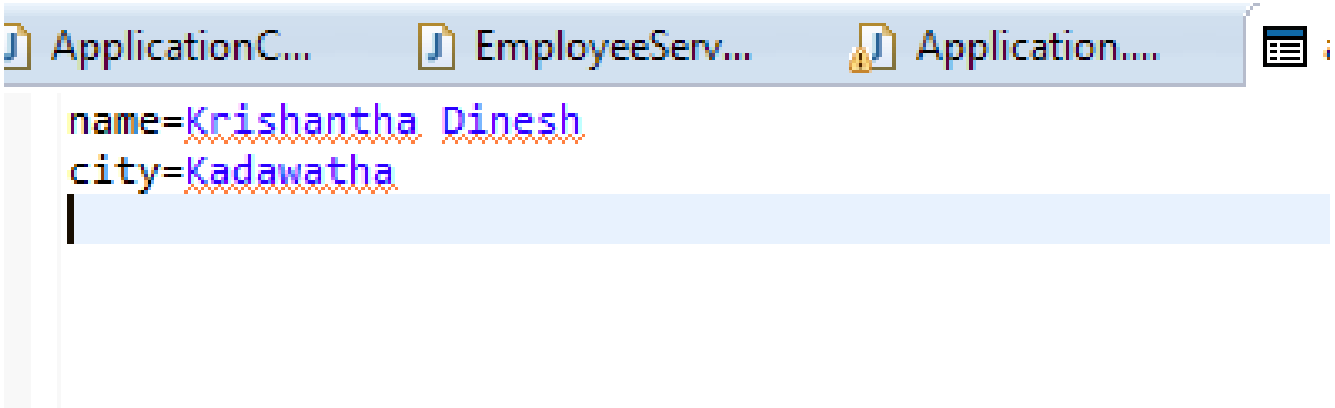
Property file with Spring

Change the Project to read property file [Anno :]

- Change applicationContext.xml to read property file

```
<context:property-placeholder location="application.properties"/>
```

- Add new property file to the same location which applicationContext.xml in as application.properties
- Add some properties



Read values

```
@Repository("you-can-use-anyname")
public class HibernateEmployeeRepositoryImpl implements EmployeeRepository {

    @Value("${name}")
    private String employeeName;
    @Value("${city}")
    private String employeeCity;

    public List<Employee> getAllEmployees() {
        List<Employee> employees = new ArrayList<>();

        Employee employee = new Employee();
        employee.setEmployeeName(employeeName);
        employee.setEmployeeLocation(employeeCity);
        employees.add(employee);
        return employees;

    }

}
```

Properties with java based configurations

- Use annotation to load the property file
- Need to configure a bean to make ready the value annotation in project
- Need aop jar file in class path
- Create property file on src
- Add some properties
- Add properties to applicationConfiguration.java file
- Add bean to read that file

Load property file with java

```
@Configuration
@ComponentScan("com.krishantha")
@PropertySource("application.properties")
public class ApplicationConfiguration {

    @Bean(name = "employeeService")
    @Scope("prototype")
    public EmployeeService getEmployeeService() {

        EmployeeServiceImpl employeeService = new EmployeeServiceImpl();

        return employeeService;
    }

    @Bean
    public static PropertySourcesPlaceholderConfigurer getPropertySourcesPlaceholderConfigurer() {
        return new PropertySourcesPlaceholderConfigurer();
    }
    /*
     * @Bean(name = "employeeRepositiry") public EmployeeRepository
```

Read property with java

```java
@Repository("you-can-use-anyname")
public class HibernateEmployeeRepositoryImpl implements EmployeeRepository {

    @Value("${name}")
    private String employeeName;
    @Value("${city}")
    private String employeeCity;

    public List<Employee> getAllEmployees() {
        List<Employee> employees = new ArrayList<>();

        Employee employee = new Employee();
        employee.setEmployeeName(employeeName);
        employee.setEmployeeLocation(employeeCity);
        employees.add(employee);
        return employees;

    }

}
```

Happy life with