TESTYANTRA

SOFTWARE SOLUTIONS (INDIA) PVT. LTD.

Spring Core

EXPERIENTIAL
learning factory

Spring Framework Introduction

TESTYANTRA SOFTWARE SOLUTIONS (INDIA) PVT. LTD.

EXPERIENTIAL learning factory

- It is a Dependency Injection based Java Application Framework.
- It is Open Source, Light Weight, Loosely Coupled and Aspect Oriented Framework to develop Modern Java-based Enterprise Applications.
- It gives a platform to developer for developing applications.
- It reduces the programmer's efforts to develop the application by providing in-built functionalities which are very common and required to develop the application.

Dependency

- A typical enterprise application does not consist of a single object (or bean in the Spring context).
- Even the simplest application has a few objects that work together. In other words, one object working with other object is nothing but it is dependent on the other object.

Dependency Injection / IoC

- IoC is also known as “Dependency Injection (DI)”
- It is a process whereby objects define their dependencies (i.e., the other object they work with) only through constructor arguments / arguments to a factory method or properties that are set on the object instance after it is constructed or returned from a factory method. The container then injects those dependencies when it creates the bean.
- In simple words, in Dependency Injection, the underlying container / software / framework creates the main object and dependent object and also assigns dependent object to main object. Everything is taken care by the underlying container / software / framework.

Dependency Injection / IoC

- Why it is called IoC (i.e. Inversion of Control)?
- In general, if main object is dependent on some other object then main object has to create / search the dependent object and get the things done.
- But in Spring, what happens, someone else is doing all this thing and giving object to us.
- It is Reverse of regular control flow. Hence, Inversion of Control.

Dependency Injection

- Various ways of Dependency Injection in Spring –

  - Setter Injection
  - Constructor Injection
  - Arguments to a factory method

Dependency Injection

Setter injection

- Dependent class object is assigned to main object by calling setter method.

Setter injection

```
class Abc {
    ----
    ----
}
```

Setter injection

```
class Xyz {
    Abc ref;
    ----
    ----
    public void setRef(Abc ref) {
            this.ref = ref;
    }
}
```

Dependency Injection

Constructor injection

- Dependent class object is assigned to main object through parametrized constructor.

Constructor injection

```
class Abc {
    ----
    ----
}
```

Constructor injection

```
class Xyz {
    Abc ref;
    ----
    ----
    Xyx (Abc ref, ---- ---, ---- ---) {
            this.ref = ref;
    }
}
```

Arguments to a factory method

- To understand this, first we should know what is a factory method.

Arguments to a factory method

```
class Abc {
    public void m1() {
        …….
    }
}
```

Arguments to a factory method

```
class AbcFactory {
   public static Abc getInstance() {
                return new Abc();
   }
}
```

Arguments to a factory method

```
class Xyz {
    private Abc ref = AbcFactory.getInstance();
}
```

IoC Container

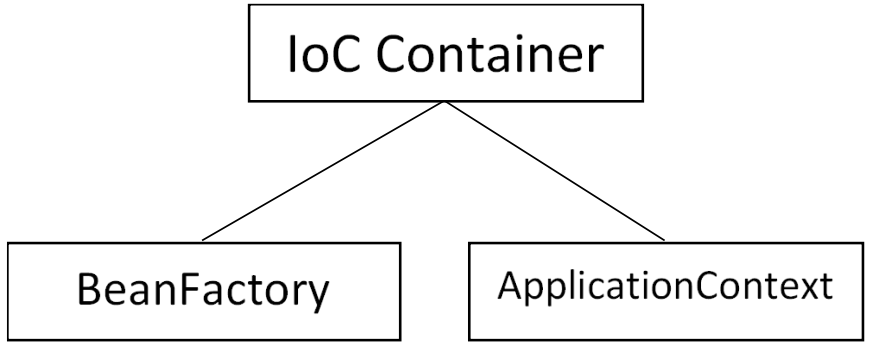
- IoC container is the one who manages the objects in our application.

- ‘Manages the objects’ means IoC Container is responsible for basically three things-
  - Instantiate the object
  - Configure the object
  - Inject the dependencies

  - What objects it needs to instantiate, how to configure them all this information will be present in an xml file and we call it as **Configuration Metadata**.

  - The IoC Container reads this Configuration Metadata and performs the operations.

IoC Container

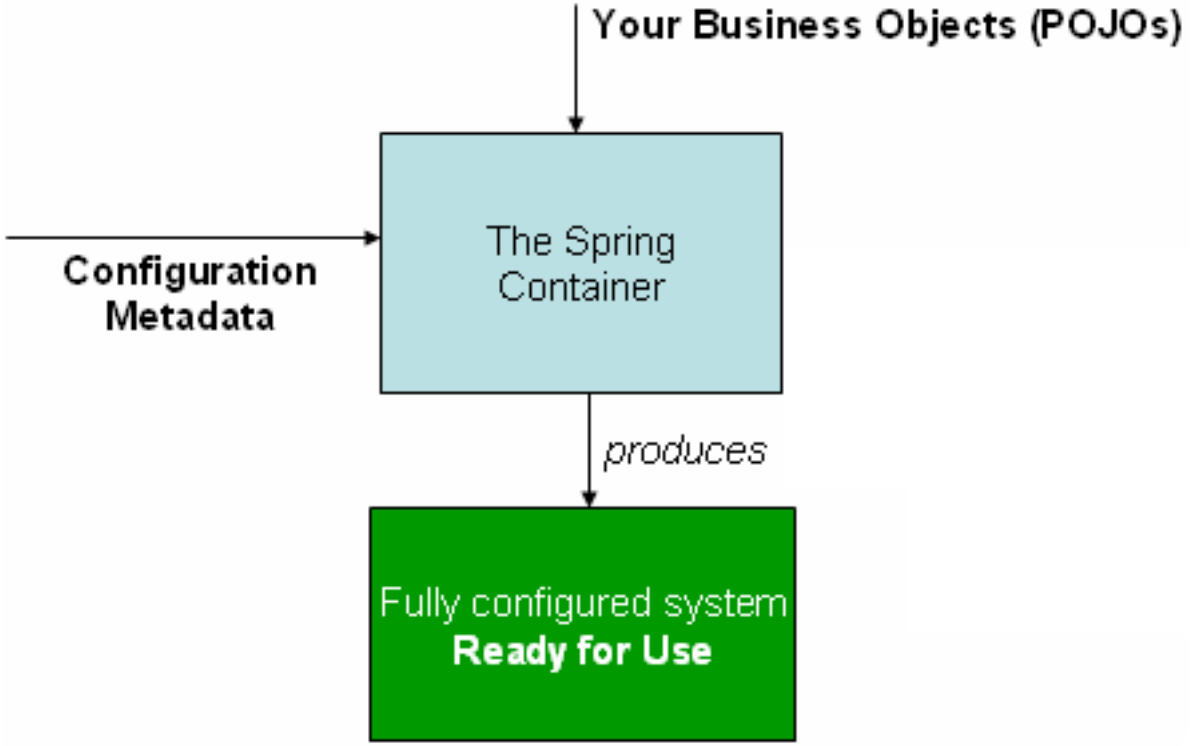


- **org.springframework.beans** and **org.springframework.context** packages are the basis for Spring framework's IoC Container.
- The **BeanFactory** interface provides configuration mechanism which is capable of managing any type of object and also provides some basic functionality.
- **ApplicationContext** is a *Sub-Interface of BeanFactory* and it adds more enterprise-specific functionality.

IoC Container

- In other words, BeanFactory provides configurations and basic functionalities and ApplicationContext adds more enterprise-specific functionalities.
- We can say that “ApplicationContext is a complete superset of BeanFactory”.

Configuration Metadata



- We know that, IoC Container needs some configuration data to know what objects it has to instantiate, configure and how to inject the dependencies.
- This data is called as **Configuration Metadata**.

Configuration Metadata

- In other words, we can say that, the Configuration Metadata represents the way an application developer tells the Spring container, nothing but IoC container, that how to instantiate, configure and assemble the objects in his application.
- But, XML is not the only way to supply configuration metadata to IoC Container.
- We can provide this configuration metadata using annotations.

Configuration Metadata

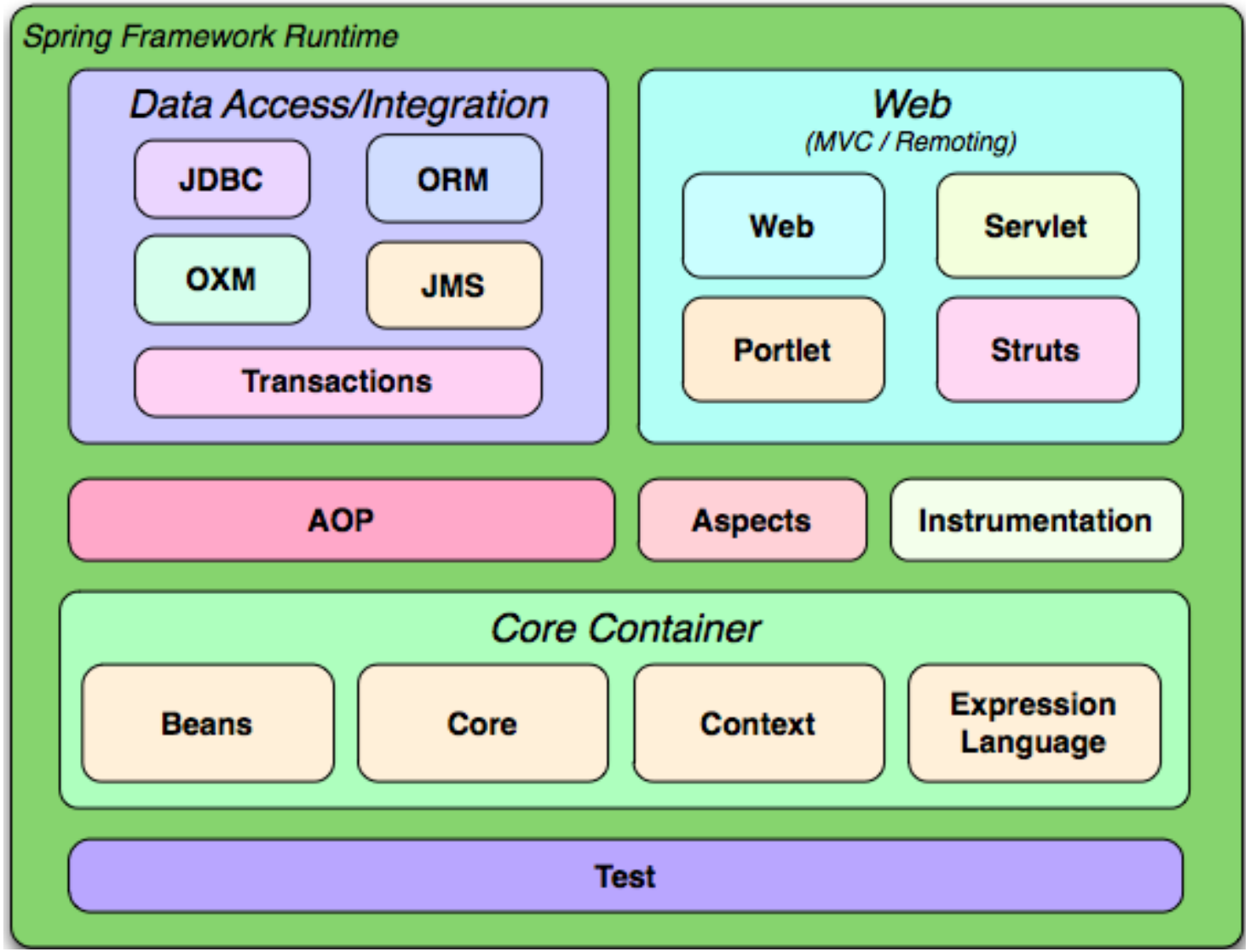
- Sample of the XML-Based Configuration Metadata –

```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">

  <bean id="..." class="...">
    <!-- collaborators and configuration for this bean go here -->
  </bean>

  <bean id="..." class="...">
    <!-- collaborators and configuration for this bean go here -->
  </bean>

  <!-- more bean definitions go here -->

</beans>
```

Configuration Metadata

- **<beans> ……..... </beans>** is the root element of the configuration metadata.
- Inside this <beans> ……. </beans> element, there is a sub-element, **<bean> …….. </bean>**.
- Spring container / IoC container needs to know that what objects it has to manage. That information we supply in Configuration Metadata using **<bean> …… </bean>** element.
- There may be 1 or more **<bean> …… </bean>** element(s) in the configuration metadata.

Spring Architecture



Spring Architecture

- ***Core Container*** – It provides some fundamental parts of the framework, like IoC and Dependency Injection features.
  – That creates a solid base of the framework.

- ***Data Access / Integration*** – As the name defines, it is for data access.
  – It consists of JDBC, ORM and some other modules.
  – Where JDBC module provides the abstraction layer that removes the need of writing the traditional JDBC code.
  – ORM module provides the integration layer for popular object-relation mapping APIs, like- JPA, Hibernate.

Spring Architecture

- ***Web*** – This layer provides web-oriented integration features (nothing but features that are required to build a web application).
  – It also includes Spring's Model-View-Controller (MVC) implementation for web applications.

- ***Test*** – The Test module supports the testing of Spring components with JUnit or TestNG.

Beans (in Java)

1. It should be a concrete class.
2. Should have a public default constructor.
3. All the data members should be private.
4. Should have public getters and setters.
5. No other methods.
6. Should implements Serializable.

Spring Beans

- In Spring, the objects that form the backbone of our application and are managed by the Spring IoC container are called beans.
- A bean is an object that is instantiated and assembled, and otherwise managed by a Spring IoC container. Otherwise, a bean is simply one of many objects in your application.
- These beans are created with the configuration metadata that you supply to the container, for example, in the form of XML <bean/> definitions.
- Within the container itself, these bean definitions are represented as BeanDefinition objects, which contain (among other information) the following metadata-

Spring Beans

- Within the container itself, these bean definitions are represented as BeanDefinition objects, which contain (among other information) the following metadata –

1. *A package-qualified class name*: typically, the actual implementation class of the bean being defined.

2. Bean behavioral configuration elements, which state how the bean should behave in the container (scope, lifecycle etc.).

3. References to other beans that are needed for the bean to do its work; these references are also called ***collaborators*** or ***dependencies***.

4. Other configuration settings to set in the newly created object, for example, the number of connections to use in a bean that manages a connection pool, or the size limit of the pool.

Spring Beans - Scope

- **Singleton** – This scopes the bean definition to a single instance per Spring IoC container (default)

```
<bean   id="----"   class="----" > …… </bean>        OR
<bean   id="----"   class="----"   scope="singleton" > …… <bean>      OR
@Configuration
public class MessageConfig {
        @Bean
        @Scope("singleton")        // Optional (default)
        public MessageBean getMessageBean() {
                // …
        }
}
```

Spring Beans - Scope

- **Prototype** – This scopes a single bean definition to have any number of object instances.

```
<bean   id="----"   class="----"   scope="prototype" > …… <bean>
```

OR

```
@Configuration
public class MessageConfig {
        @Bean
        @Scope("prototype")
        public MessageBean getMessageBean() {
                // …
        }
}
```

Spring Beans - Scope

- **Request** – This scopes a bean definition to an HTTP request.
  Only valid in the context of a web-aware Spring ApplicationContext.

```
<bean  id="----"   class="----"   scope="request" > …… <bean>
```

OR

```
@RequestScope
@Component
public class MyClass {
        // …
}
```

Spring Beans - Scope

- **Session** – This scopes a bean definition to an HTTP session.
  Only valid in the context of a web-aware Spring ApplicationContext.

```
<bean   id="----"   class="----"   scope="session" > …… <bean>
```

OR

```
@SessionScope
@Component
public class MyClass {
        // …
}
```

Spring Beans - Scope

- **Application** – This scopes a bean definition to an application.
  Only valid in the context of a web-aware Spring ApplicationContext.

  <bean id="----" class="----" scope="application" > …… <bean>

  OR

```
@ApplicationScope
@Component
public class MyClass {
        // …
}
```

Spring Beans - Lifecycle

- Just like servlet lifecycle, here the beans also have instantiation phase, initialization phase, and destroy phase.
- In between initialization phase and destroy phase we operate on beans object.
- For instantiation spring container uses public default constructer. For initialization and destroy phase there are 3 ways to implement.

  1) Implementing ***InitializingBean*** and ***DisposableBean*** for initialization and destroy phase respectively for the bean class.

  2) Providing attributes ***init-method*** and ***destroy-method*** for *<bean>* element in the xml file.

  3) Provide @***PostConstruct*** and @***PreDestroy*** for the init method and destroy method respectively.

Spring Beans - Lifecycle

- Implementing ***InitializingBean*** and ***DisposableBean***

```
public class MyBean implements InitializingBean, DisposableBean {
        // …

        @Override
        public void afterPropertiesSet() throws Exception {
                // initialization phase
        }

        @Override
        public void destroy() throws Exception {
                // Destroy Phase
        }

}// End of class
```

Spring Beans - Lifecycle

- Providing attributes ***init-method*** and ***destroy-method***

```
public class MyBean {
        // …

        public void init() throws Exception {
                // initialization phase
        }

        public void destroy() throws Exception {
                // Destroy Phase
        }
} // End of class
```

beans.xml

```
<bean  id="----"   class="----"   init-method="init"   destroy-method="destroy"  />
```

Spring Beans - Lifecycle

- Using @***PostConstruct*** and @***PreDestroy***

```
public class MyBean {
        // …

        @PostConstruct
        public void init() throws Exception {
                // initialization phase
        }

        @PreDestroy
        public void destroy() throws Exception {
                // Destroy Phase
        }

} // End of class
```

Spring Beans – Auto-wiring

- To inject bean normally to another bean, elements <constructor-arg> and <property> are used.
- But spring container can auto-wire the injection without <constructor-arg> and <property> elements.
- Auto-wiring is configured in <bean> element in xml file.
- There are 3 modes in auto-wiring –
  1) byName
  2) byType
  3) Constructor

Spring Beans – Auto-wiring

1. No auto-wiring (default) using "**ref**" –

```
<bean id="----" class="----" >
        <property name="----" value="----" />
        <property name="----" ref="----" />
</bean>
```

1) byName –

```
<bean id="----" class="----" autowire="byName" >
        <property name="----" value="----" />
</bean>
```

2) byType –

```
<bean id="----" class="----" autowire="byType" >
        <property name="----" value="----" />
</bean>
```

3) Constructor –

```
<bean id="----" class="----" autowire="constructor" >
        <constructor-arg name="----" value="----" />        // for other properties
</bean>
```

- **Auto-Wiring Using Annotation –**

```
public class MyClass {

        @Autowired
        private DependentClass dependentClassRef;

}
```

Spring Beans – Auto-wiring

- If more than one dependent beans are qualified for auto-wiring then one can be auto-wired using “primary” attribute in xml or “@Qualifier” annotation along with @Autowired.

```
<bean id=“----” class=“----” autowire="byType" >
        <property name=“----” value=“----” />
</bean>

<!– more than one dependent beans -->
<bean id=“----” class=“----” primary="true">
        <property name=“----” value=“----” />
</bean>

<bean id=“----” class=“----” >
        <property name=“----” value=“----” />
</bean>
```

Spring Beans – Auto-wiring


```
public class MyClass {
        @Autowired
        @Qualifier("d1")
        private DependentClass ref;
}

@Configuration
public class MyConfig {
        @Bean(name = "d1")
        public DependentClass getDependentBean1() {
                // …
        }
        @Bean(name = "d2")
        public DependentClass getDependentBean2() {
                // …
        }
}
```

BeanPostProcessor

- Factory hook that allows for custom modification of new bean instances.
- A class needs to implement “BeanPostProcessor” to create a custom bean post processors.
- This class need to be configured in configuration metadata.
- It will be populated along with the initialization phase of a bean.
- If created multiple bean post processors, then the order of execution can be prioritize by two ways-

  1) by the order of configuring these post processors in xml file

  OR

  2) by implementing “Ordered” interface and return the int value from getOrder() method.

BeanPostProcessor


```
public class MyBeanPostProcessor implements BeanPostProcessor, Ordered {
        @Override
        public Object postProcessBeforeInitialization(Object bean, String beanName)
        throws BeansException {
                // will be executed just before initialization phase of every bean.
                return bean;
        }
        @Override
        public Object postProcessAfterInitialization(Object bean, String beanName)
        throws BeansException {
                // will be executed just after initialization phase of every bean.
                return bean;
        }
        @Override
        public int getOrder() {
                return 0;            // lower the value, higher the priority
        }
}
```

beans.xml

```
        <bean  class="package.subpackage.MyBeanPostProcessor"  />
```

BeanFactoryPostProcessor

- Allows for custom modification of an application context's bean definitions, adapting the bean property values of the context's underlying bean factory.
- Application contexts can auto-detect BeanFactoryPostProcessor beans in their bean definitions and apply them before any other beans get created.
- ApplicationContexts can auto-detect BeanPostProcessor beans in their bean definitions and apply them to any beans subsequently created.
- It will be populated before the creation of a bean.
- The provided configuration metadata for a bean can be validated and changed using bean factory post processor.

BeanFactoryPostProcessor

- A class needs to implement “BeanFactoryPostProcessor” to create a custom bean factory post processors.
- If created multiple bean factory post processors, then the order of execution can be prioritize by two ways-

  1) by the order of configuring these post processors in xml file.

  OR

  2) by implementing “Ordered” interface and return the int value from getOrder() method.

BeanFactoryPostProcessor


```
public class MyBeanFactoryPostProcessor implements BeanFactoryPostProcessor {

        @Override
        public void postProcessBeanFactory(ConfigurableListableBeanFactory beanFactory)
        throws BeansException {
                // will be executed before the creation of a bean.

                BeanDefinition beanDefinition = beanFactory.getBeanDefinition("messageBean");
                MutablePropertyValues propertyValues = beanDefinition.getPropertyValues();

                propertyValues.addPropertyValue("message", "Message from my BFPP.");
        }

}
```

beans.xml

```
        <bean  class="package.subpackage.MyBeanFactoryPostProcessor"  />
```

Aware Interfaces

- A marker super interface indicating that a bean is eligible to be notified by the Spring container of a particular framework object through a callback-style method.
- It typically consists of just one void-returning method that accepts a single argument.
- Note that merely implementing Aware provides no default functionality. Rather, processing must be done explicitly.
- There are 16 aware interfaces in Spring 5.
  E.g. – ApplicationContextAware, BeanClassLoaderAware, BeanFactoryAware, BeanNameAware, ServletConfigAware, ServletContextAware, etc.
- A bean can implements one or more Aware interfaces at a time.

Aware Interfaces

- **E.g. –**

```
public class MyBean implements BeanNameAware {
        @Override
        public void setBeanName(String name) {
                // TODO Auto-generated method stub
        }
}
```

TESTYANTRA

SOFTWARE SOLUTIONS (INDIA) PVT. LTD.

Contact Us

No.01, 3rd cross Basappa Layout, Gavipuram Extension, Kempegowda Nagar, Bengaluru, Karnataka 560019

sagar.g@testyantra.com
gurupreetham.c@testyantra.com
praveen.d@testyantra.com

www.testyantra.com

EXPERIENTIAL learning factory