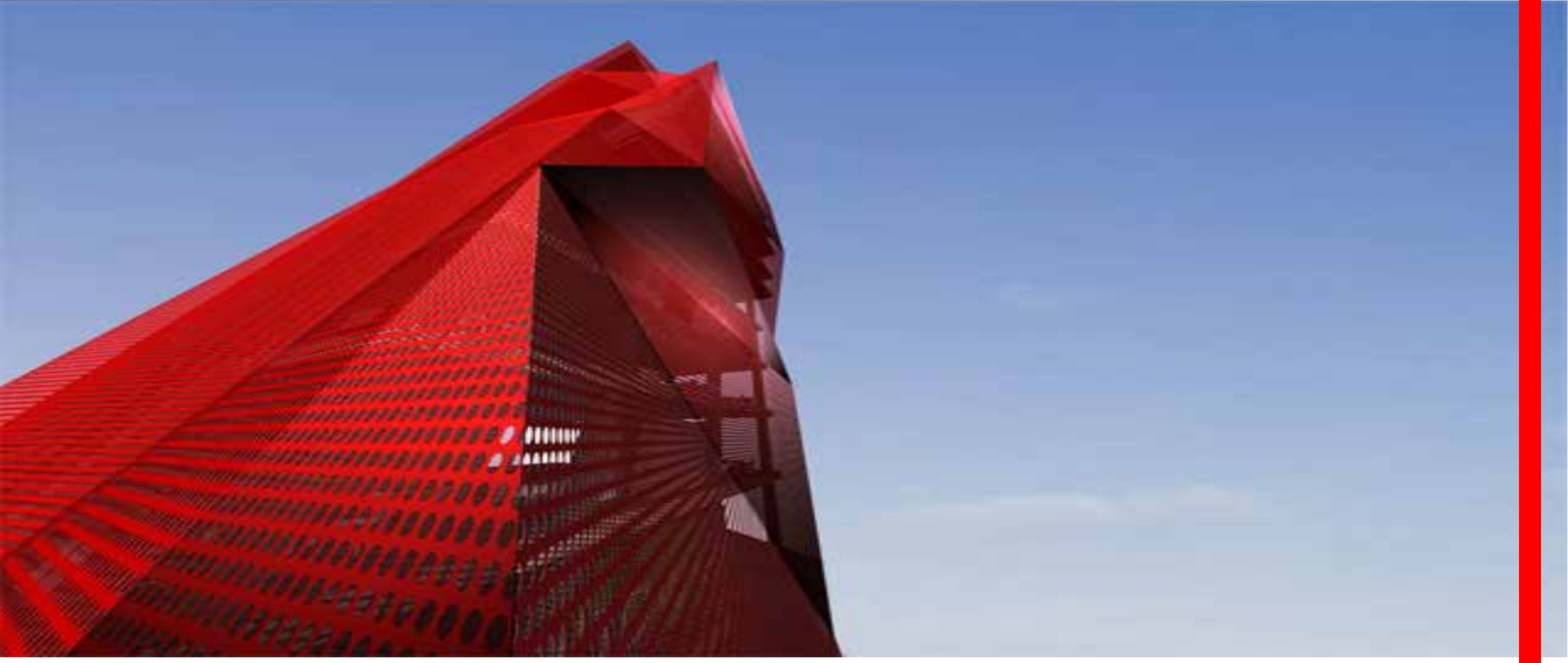


Training – Technical development

Spring IOC 3.0

Duration – 4 hours

HSBC

Agenda

- Annotation Based Spring Configuration
- @Bean
- @Autowired
- @Resource
- @Required
- @Value
- @Import
- @PostConstruct
- @PreDestroy
- @Scope
- Java-based container configuration

What's New in Spring 3.0

- Spring has undergone two major revisions: Spring 2.0, released in October 2006, and Spring 2.5, released in November 2007.
- The Spring Framework is now based on Java 5, and Java 6 is fully supported.
- Furthermore, Spring is compatible with J2EE 1.4 and Java EE 5, while at the same time introducing some early support for Java EE 6.
- The entire framework code has been revised to take advantage of Java 5 features like generics, varargs and other language improvements.

New module organization and build system

The framework modules have been revised and are now managed separately with one source-tree per module jar:

- org.springframework.aop
- org.springframework.beans
- org.springframework.context
- org.springframework.context.support
- org.springframework.expression
- org.springframework.instrument
- org.springframework.jdbc
- org.springframework.jms
- org.springframework.orm
- org.springframework.oxm
- org.springframework.test
- org.springframework.transaction
- org.springframework.web
- org.springframework.web.portlet
- org.springframework.web.servlet
- org.springframework.web.struts

Overview of new features

This is a list of new features for Spring Framework 3.0.

- Spring Expression Language
- IoC enhancements/Java based bean metadata
- General-purpose type conversion system and field formatting system
- Object to XML mapping functionality (OXM) moved from Spring Web Services project
- Comprehensive REST support
- @MVC additions
- Declarative model validation
- Early support for Java EE 6
- Embedded database support

Core APIs updated for Java 5

BeanFactory interface returns typed bean instances as far as possible:

- T getBean(Class<T> requiredType)
- T getBean(String name, Class<T> requiredType)
- Map<String, T> getBeansOfType(Class<T> type)

Spring Expression Language

- Spring introduces an expression language which is similar to Unified EL in its syntax but offers significantly more features. The expression language can be used when defining XML and Annotation based bean definitions and also serves as the foundation for expression language support across the Spring portfolio.
- The Spring Expression Language was created to provide the Spring community a single, well supported expression language that can be used across all the products in the Spring portfolio. Its language features are driven by the requirements of the projects in the Spring portfolio, including tooling requirements for code completion support within the Eclipse based SpringSource Tool Suite.

- The following is an example of how the Expression Language can be used to configure some properties of a database setup

```
<bean class="mycompany.RewardsTestDatabase">
    <property name="databaseName"
        value="#{systemProperties.databaseName}"/>
    <property name="keyGenerator"
        value="#{strategyBean.databaseKeyGenerator}"/>
</bean>
```

- This functionality is also available if you prefer to configure your components using annotations

The Inversion of Control (IoC) container

- **Java based bean metadata**
- Some core features from the JavaConfig project have been added to the Spring Framework now. This means that the following annotations are now directly supported:
- @Configuration
- @Bean
- @DependsOn
- @Primary
- @Lazy
- @Import
- @ImportResource
- @Value

Spring Annotation-based container configuration

- An alternative to XML setups is provided by annotation-based configuration which rely on the bytecode metadata for wiring up components instead of angle-bracket declarations.
- Instead of using XML to describe a bean wiring, the developer moves the configuration into the component class itself by using annotations on the relevant class, method, or field declaration.
- Spring 2.5 adds support for JSR-250 annotations such as @Resource, @PostConstruct, and @PreDestroy.
- Spring 3.0 adds support for JSR-330 (Dependency Injection for Java) annotations contained in the javax.inject package such as @Inject, @Qualifier, @Named, and @Provider if the JSR330 jar is present on the classpath.
- **Note :** Annotation injection is performed *before* XML injection, thus the latter configuration will override the former for properties wired through both approaches

9

Are annotations better than XML for configuring Spring?

- The introduction of annotation-based configurations raised the question of whether this approach is 'better' than XML. The short answer is ***it depends*.**
- The long answer is that **each approach has its pros and cons**, and usually it is up to the developer to decide which strategy suits her better.
- Due to the way they are defined, annotations provide a lot of context in their declaration, leading to shorter and more concise configuration. However, XML excels at wiring up components **without touching their source code** or recompiling them.
- Some developers prefer having the **wiring close to the source** while others argue that annotated classes are no longer POJOs and, furthermore, that the **configuration becomes decentralized and harder to control.**

- Note: Annotation injection is performed *before* XML injection, thus the latter configuration will override the former for properties wired through both approaches

10

Spring Framework 2.5 onwards

- Comprehensive support for **annotation-based configuration**
- – @Autowired
  - • with optional @Qualifier or custom qualifier
- – @Transactional
- – @Service, @Repository, @Controller
- Common Java EE 5 annotations supported too
- – @PostConstruct, @PreDestroy
- – @PersistenceContext, @PersistenceUnit
- – @Resource, @EJB, @WebServiceRef
- – @TransactionAttribute

Component Scanning

- This allows to scan the base package at the startup
  - Sub-packages of the base package are also scanned

- <context:component-scan base-package=*"com.hsbc.glt"*/>

Annotation based configuration

- @AutoWired

- Indicates that Spring should inject a dependency
  - Based on its type (default)
  - Based on its name if used with @Qualifier annotation

- Setter Wiring

```
@Autowired
public void setAdd(final Address add) {
this.add = add;
}
```

- Constructor based wiring

```
@Autowired
public Person(final Address add) {
this.add = add;
}
```

- Field wiring

```
@Autowired
Address add;
```

@Required

- The @Required annotation applies to bean property setter methods
- This annotation simply indicates that the affected bean property must be populated at configuration time, through an explicit property value in a bean definition or through autowiring.
- The container throws an exception if the affected bean property has not been populated; this allows for eager and explicit failure, avoiding NullPointerExceptions or the like later on.
- It is still recommended that you put assertions into the bean class itself, for example, into an init method. Doing so enforces those required references and values even when you use the class outside of a container.

```
@Autowired
@Required
public void setAdd(final Address add) {
this.add = add;
}
```

- Exp: Third

@Autowired and Required

- By default, the autowiring fails whenever *zero* candidate beans are available; the default behavior is to treat annotated methods, constructors, and fields as indicating *required* dependencies. This behavior can be changed as demonstrated below.

- Only *one annotated constructor per-class* can be marked as *required*, but multiple non-required constructors can be annotated. In that case, each is considered among the candidates and Spring uses the *greediest* constructor whose dependencies can be satisfied, that is the constructor that has the largest number of arguments.

- @Required, on the other hand, is stronger in that it enforces the property that was set by any means supported by the container. If no value is injected, a corresponding exception is raised.

Fine-tuning annotation-based autowiring with qualifiers

- Because autowiring by type may lead to multiple candidates, it is often necessary to have more control over the selection process.
- One way to accomplish this is with Spring's @Qualifier annotation. You can associate qualifier values with specific arguments, narrowing the set of type matches so that a specific bean is chosen for each argument.

```
@Autowired
@Required
public void setAdd(@Qualifier("another")Address add) {
this.add = add;
}
```

- The bean with qualifier value “another" is wired with the setter argument that is qualified with the same value.
- For a fallback match, the bean name is considered a default qualifier value. Thus you can define the bean with an id “another" instead of the nested qualifier element, leading to the same matching result.
- However, although you can use this convention to refer to specific beans by name, @Autowired is fundamentally about type-driven injection with optional semantic qualifiers.

@Qualifier

- The @Qualifier annotation can also be specified on individual constructor arguments or method parameters:

```
@Autowired
public Person(@Qualifier("another") Address add) {
this.add = add;
}
```

@Resource

- Spring also supports injection using the JSR-250 @Resource annotation on fields or bean property setter methods.
- This is a common pattern in Java EE 5 and 6, for example in JSF 1.2 managed beans or JAX-WS 2.0 endpoints. Spring supports this pattern for Spring-managed objects as well.
- @Resource takes a name attribute, and by default Spring interprets that value as the bean name to be injected.
- In other words, it follows *by-name* semantics, as demonstrated in this example:

```
@Resource(name = "another")
public void setAdd(Address add) {
this.add = add;
}
```

@Resource…

- If no name is specified explicitly, the default name is derived from the field name or setter method.
- In case of a field, it takes the field name; in case of a setter method, it takes the bean property name.
- So the following example is going to have the bean with name “add" injected into its setter method:

```
@Resource
public void setAdd(Address add) {
this.add = add;
}
```

@PostConstruct and @PreDestroy

- The CommonAnnotationBeanPostProcessor not only recognizes the @Resource annotation but also the JSR-250 lifecycle annotations.
- Introduced in Spring 2.5, the support for these annotations offers yet another alternative to those described in initialization callbacks and destruction callbacks.
- We have used afterPropertyset() and destroy() methods in XML based configuration with the help of InitializingBean and Destroy Bean Interface.
- We can achieve the same feature by using @PostConstruct and @PreDestroy annotations

```
@PostConstruct
public void beforeConstruct()
{
System.out.println("Before Construction");
}
@PreDestroy
public void afterDest()
{
System.out.println("After Destruction");
}
```

Exp: Eighth

Classpath scanning and managed components

- The previous section demonstrated how to provide a lot of the configuration metadata through source-level annotations. However, the "base" bean definitions are explicitly defined in the XML file, while the annotations only drive the dependency injection.

- This section describes an option for implicitly detecting the candidate components by scanning the classpath.

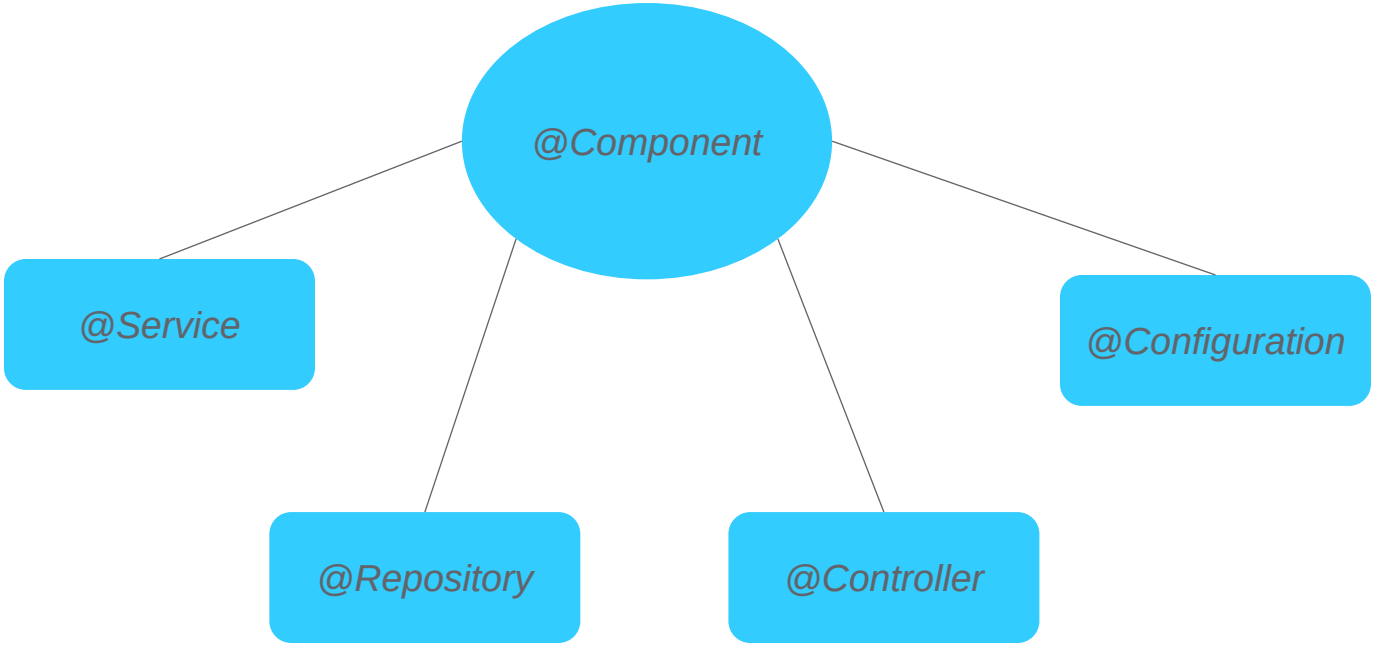
- Candidate components are classes that match against a filter criteria and have a corresponding bean definition registered with the container. **This removes the need to use XML** to perform bean registration, instead you can use annotations (for example @Component), AspectJ type expressions, or your own custom filter criteria to select which classes will have bean definitions registered with the container

- Starting with Spring 3.0, many features provided by the Spring JavaConfig project are part of the core Spring Framework. This allows you to define beans using Java rather than using the traditional XML files.

Automatically detecting classes and registering bean definitions

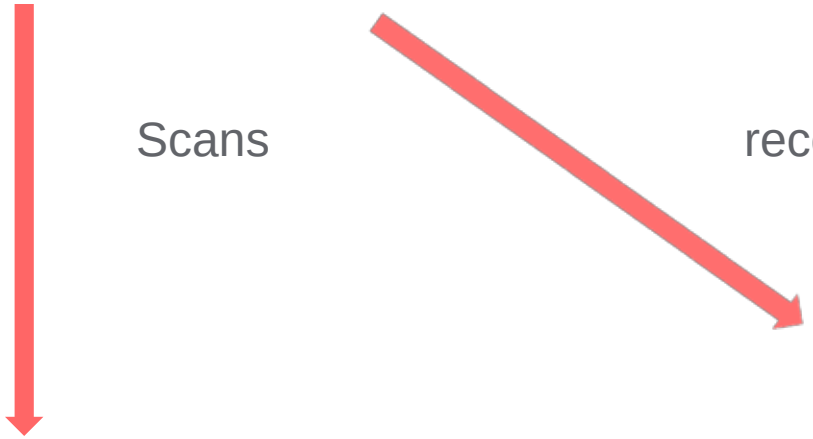
- Spring can automatically detect stereotyped classes and register corresponding Bean Definitions with the ApplicationContext

- Spring framework stereotype annotations:



How it works

- `<context:component-scan base-package="com.hsbc.bean"/>`




```
@Service
public class Person {…}
```

Custom Annotation

```
@Repository
public class Address implements AddIntf
{..}
```

@Value = On a field or method or on a parameter of autowired method or constructor

```
@Value(“Paris")
String city;
```

Component scanning : Best Practices

- Really Bad

```
<context:component-scan base-package="com.hsbc"/>
```

- Bad

```
<context:component-scan base-package="hsbc"/>
```

- OK

```
<context:component-scan base-package="com.hsbc.bean"/>
```

- Optimized

```
<context:component-scan base-package="com.hsbc.bean.respository,
        com.hsbc.bean.controller "/>
```

Java Based Approach

- Spring 3.0 uses java based configurations

- Used to be a separate project called Spring JavaConfig
- @Configuration annotated classes are key artifacts
- @Bean methods instead of <bean/> elements

- It allows Java Code to define and configure beans

- Please note configuration will still be external to POJO classes to be configured
- More control over bean instantiation and configuration
- More type-safety

Basic concepts: @Configuration and @Bean

- The central artifact in Spring's new Java-configuration support is the @Configuration-annotated class. These classes consist principally of @Bean-annotated methods that define instantiation, configuration, and initialization logic for objects to be managed by the Spring IoC container.

- Annotating a class with the @Configuration indicates that the class can be used by the Spring IoC container as a source of bean definitions. The simplest possible @Configuration class would read as follows:

```
@Configuration
public class AddressConfig {

public @Bean
Address add() {
return new Address();
}
}
```

- For those more familiar with Spring <beans/> XML, the AppConfig class above would be equivalent to:

```
<bean id = 'add' class="com.hsbc.bean.Address">
```

Instantiating the Spring container using AnnotationConfigApplicationContext

- Spring's AnnotationConfigApplicationContext, new in Spring 3.0. This versatile ApplicationContext implementation is capable of accepting not only @Configuration classes as input, but also plain @Component classes and classes annotated with JSR-330 metadata.

- When @Configuration classes are provided as input, the @Configuration class itself is registered as a bean definition, and all declared @Bean methods within the class are also registered as bean definitions.
- When @Component and JSR-330 classes are provided, they are registered as bean definitions, and it is assumed that DI metadata such as @Autowired is used within those classes where necessary

```
public static void main(String[] args) {
ApplicationContext ctx = new AnnotationConfigApplicationContext(
AddressConfig.class);
Address myAddress = ctx.getBean(Address.class);
System.out.println(myAddress.getCountry());
}
```

- WebApplicationContext variant of AnnotationConfigApplicationContext is available with AnnotationConfigWebApplicationContext. This implementation may be used when configuring the Spring ContextLoaderListener servlet listener, Spring MVC DispatcherServlet, etc.

Creating container programmatically using register

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

AnnotationConfigApplicationContext ctx = new AnnotationConfigApplicationContext();
  ctx.register(AppConfig.class, OtherConfig.class);
  ctx.register(HelloConfig.class);
  ctx.refresh();
  Hello hello = ctx.getBean(Hello.class);
  hello.doStuff();
}
```

Creating container programmatically using scan

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

AnnotationConfigApplicationContext ctx = new AnnotationConfigApplicationContext();
  ctx.scan("com.hsbc");
  ctx.refresh();
  Hello hello = ctx.getBean(MyService.class);
}
```

Using the @Import annotation

- Much as the <import/> element is used within Spring XML files to aid in modularizing configurations, the @Import annotation allows for loading @Bean definitions from another configuration class:

```
@Configuration
@Import(AddressConfig.class)
public class PersonConfig {

private @Autowired
Address add;

@Bean
public Person myPerson() {

return new Person("HSBC", 10, add);
}
}
```

Using the @Import annotation….

- Now, rather than needing to specify both AddressConfig.class and PersonConfig.class when instantiating the context, only PersonConfig.class needs to be supplied explicitly:

```
public class ConfigClient {

public static void main(String[] args) {
ApplicationContext ctx = new AnnotationConfigApplicationContext(
PersonConfig.class);

Person myPerson = ctx.getBean(Person.class);
System.out.println(myPerson.getAdd().getCountry());
}
```

- This approach simplifies container instantiation, as only one class needs to be dealt with, rather than requiring the developer to remember a potentially large number of @Configuration classes during construction

Injecting dependencies via Java

```
@Configuration
public class AllConfig {

  @Bean
  public Person per() {
      return new Person(add());
  }

  @Bean
  public Address add() {
      return new Address("Moskow","Russia");
  }
```

Injecting Straight values (primitives, Strings, and so on)

```
@Configuration
public class AllConfig {

private @Value("Tokyo") String city;
private @Value("Japan") String country;


  @Bean
  public Person per() {
      return new Person(add());
  }

  @Bean
  public Address add() {
      return new Address(city,country);
  }

}
```

Receiving lifecycle callbacks

- Beans declared in a @Configuration-annotated class support the regular lifecycle callbacks. Any classes defined with the @Bean annotation can use the @PostConstruct and @PreDestroy annotations.

- The regular Spring lifecycle callbacks are fully supported as well. If a bean implements InitializingBean, DisposableBean, or Lifecycle, their respective methods are called by the container

- The @Bean annotation supports specifying arbitrary initialization and destruction callback methods, much like Spring XML's init-method and destroy-method attributes on the bean element:

```
@Bean(initMethod = "beforeConstruct")
public Person per() {
return new Person(add());
}
```

Receiving lifecycle callbacks…

- Of course, in the case of Person above, it would be equally as valid to call the `beforeConstruct`() method directly during construction:

```
@Bean
public Person per() {
        Person per = new Person(add());
        per.beforeConstruct();
        return per;
}
```

Specifying bean scope

- Using the @Scope annotation
- You can specify that your beans defined with the @Bean annotation should have a specific scope. You can use any of the standard scopes specified in the Bean Scopes section.

- The default scope is singleton, but you can override this with the @Scope annotation:

```
@Bean(initMethod = "beforeConstruct")
  @Scope("prototype")
  public Person per() {
       Person per = new Person(add());
       per.beforeConstruct();
       return per;
  }
```

Specifying lazy initialization of bean

- Using the @Lazy annotation
- You can specify that your singleton beans defined with the @Bean annotation should be laziliy initialized
- The default scope is singleton, and @Lazy can be set to true for singleton beans

```
@Bean(name = "Irfan",initMethod = "beforeConstruct")
  @Lazy(true)
  @Scope("singleton")
  public Person per() {
       Person per = new Person(add());
       per.beforeConstruct();
       return per;
  }
```

Customizing bean naming

- By default, configuration classes use a @Bean method's name as the name of the resulting bean. This functionality can be overridden, however, with the name attribute

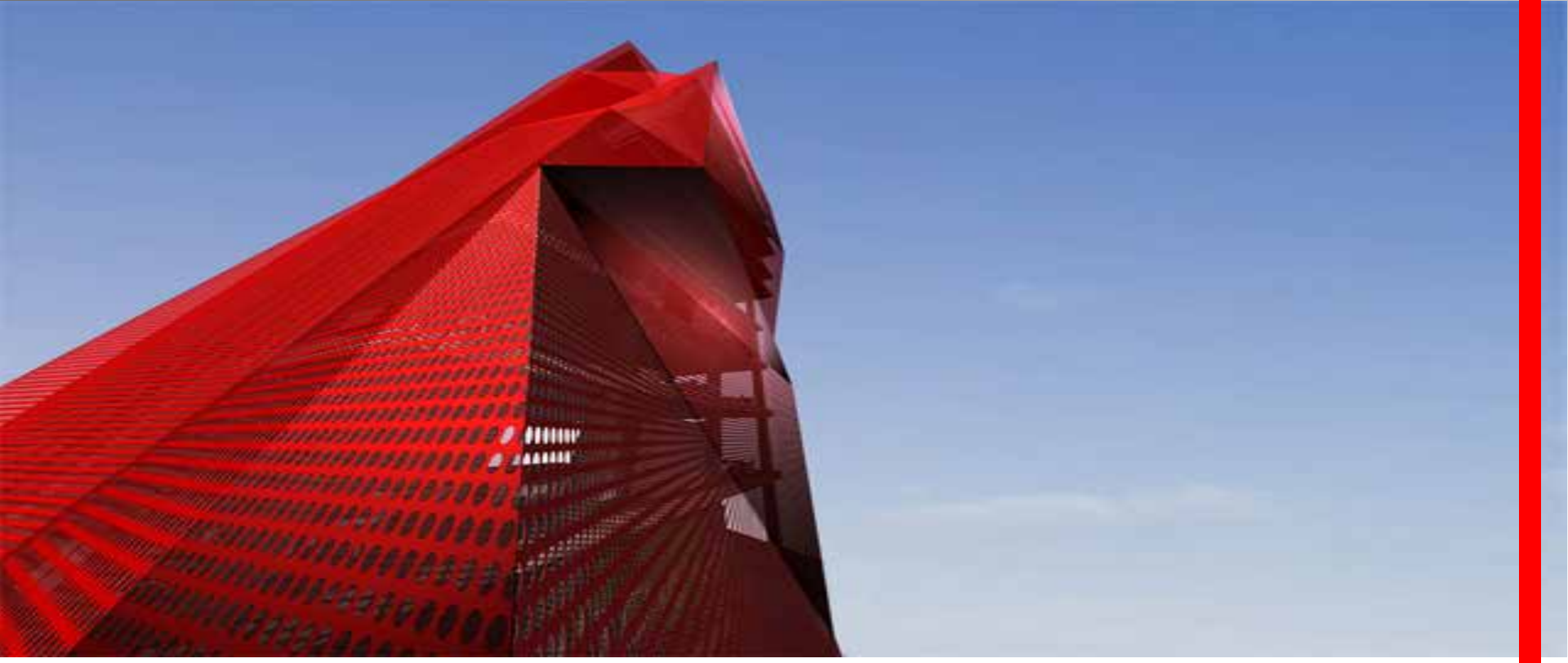
```
@Bean(name = "Irfan",initMethod = "beforeConstruct")
  @Scope("prototype")
  public Person per() {
       Person per = new Person(add());
       per.beforeConstruct();
       return per;
  }
```

Recommendations

- @Autowired's *required* attribute is recommended over the @Required annotation. The *required* attribute indicates that the property is not required for autowiring purposes, the property is ignored if it cannot be autowired.

- If you intend to express annotation-driven injection by name, do not primarily use @Autowired, even if is technically capable of referring to a bean name through @Qualifier values. Instead, use the JSR-250 @Resource annotation, which is semantically defined to identify a specific target component by its unique name, with the declared type being irrelevant for the matching process.

Thank You!!!!!!!!!!!!!!



Prepared by M Irfan
Engineering Services– Technical Development

+ 91 20 6705 2532 irfanismailmohammad@hsbc.co.in

Prepared July 2013

HSBC