[Spring Convention](https://wiki.eisgroup.com/display/GRC/Spring+Convention)

V20 uses auto-configuration to define beans. By adding jar as a dependency, all required beans are initialized and created and no additional spring configuration is required.

Extension points are marked so that they can be easily overridden, either by defining a bean of an appropriate type or name.

# Configuration

We use [Spring Boot](https://docs.spring.io/spring-boot/docs/2.1.1.RELEASE/reference/html/boot-features-developing-auto-configuration.html) to provide auto-configuration of Spring context. Genesis defines all of Springs beans using the Java based configuration.

The configuration classes should be created under <packageBase>.config package.

To make the configuration available to Spring Boot, META-INF/spring.factories resource should be added to the module's classpath:

**spring.factories**

|  |
| --- |
| org.springframework.boot.autoconfigure.EnableAutoConfiguration=\  com.eisgroup.sample.config.FooConfiguration |

The Spring bean configuration should always be defined in the **impl**modules.

## **Defining Extendable Configuration**

The classes which are intended to be overridden must always have [@ConditionalOnMissingBean](https://docs.spring.io/spring-boot/docs/current/api/org/springframework/boot/autoconfigure/condition/ConditionalOnMissingBean.html) annotation applied on the bean definition:

**FooConfig.java**

|  |
| --- |
| @Configuration  **public** **class** FooConfig {      @Bean      @ConditionalOnMissingBean  **public** FooBean fooBean() {  **return** **new** FooBean();      }  } |

If the classes are expected to be available only when specific profile is enabled,  either the @Configuration class or the @Bean itself need to be annotated with @Profile:

**FooConfig.java**

|  |
| --- |
| @Configuration  @Profile("bar")  **public** **class** FooConfig {      @Bean      @ConditionalOnMissingBean  **public** FooBean fooBean() {  **return** **new** FooBean();      }  } |

## **Overriding Beans**

If core has an extension point interface Foo and provides default implementation that suits in most cases, that implementation will be initialized once appropriate dependency is added. However if default implementation does not suit, then it can be overridden just by defining bean that implements given interface.

|  |
| --- |
| **public** **class** Bar **implements** Foo {      // new implementation here  } |

When Bar is defined in a spring context then it will be used instead of default provided implementation.:

|  |
| --- |
| @Configuration  **public** **class** FooOverrideConfig {      @Bean      @Primary  **public** Bar bar() {  **return** **new** Bar();      }  } |

### Overrides by Name

In most cases, just defining a new bean for an SPI would be enough. But the overrides by name are also allowed. Consider the following structure:And the following configuration:

**DefaultSerializerConfig.java**

|  |
| --- |
| @Configuration  **public** **class** DefaultSerializerConfig {  **public** **static** **final** String RESPONSE\_SERIALIZER = "responseSerializer";  **public** **static** **final** String MESSAGE\_SERIALIZER = "blobSerializer";        @Bean(name = RESPONSE\_SERIALIZER)      @ConditionalOnMissingBean  **public** Serializer responseSerializer() {  **return** **new** JsonSerializer();      }          @Bean(name = MESSAGE\_SERIALIZER)      @ConditionalOnMissingBean  **public** Serializer messageSerializer() {  **return** **new** BlobSerializer();      }  } |

In such cases, as the @Bean methods return the same Serializer interface, it's impossible to override them without refering to the name.

To override them, refer to the name of the bean:

|  |
| --- |
| @Configuration  **public** **class** OverridenSerializerConfig {      @Bean(name = DefaultSerializerConfig.RESPONSE\_SERIALIZER)      @Primary  **public** Serializer responseSerializer() {  **return** **new** BlobSerializer();      }  } |

This approach requires the name of the bean to be defined as an API level constant. **DO NOT**use the names of the methods directly (default bean names).

## **Extending by Delegating**

Above examples lead to a situations where default implementations are never created and as a result cannot be referenced by other beans even through xml references or qualifiers. However there are cases when we want to have original implementation and just add post or pre processing to fit specific needs. Then the new implementation should implement [DelegatingBean](https://gendocs.eisgroup.com/LATEST/platform/javadoc/com/eisgroup/genesis/spring/delegate/DelegatingBean.html) interface which will not prevent from default implementation being created. When registering such bean it should be marked as primary so that other services would prefer this implementation over others.

|  |
| --- |
| **public** **class** Bar **implements** Foo, DelegatingBean {      // new implementation here  } |

For the DelegatingBean to work, the default implementations have to have @ConditionalOnMissingBean(ignored = DelegatingBean.class) set.

# Dependency Injection

## **Best Practices**

* Constructor injection is the preferred way to inject the dependencies into the class.
* The @Value placeholders must be defined in the Configuration classes and injected using the constructor.
* Avoid @Autowired usage.

## **Bean Registries**

If there is a need to define a container class for multiple interface implementations, the [BeanRegistry](https://gendocs.eisgroup.com/LATEST/platform/javadoc/com/eisgroup/genesis/spring/registry/BeanRegistry.html) should be used.

**Warning:** Do not use @Autowired Collections of Spring beans.

# Multiple Spring Contexts

As documented at [Modular Design](https://wiki.eisgroup.com/display/GRC/Modular+design), there can be multiple spring contexts. To make sure that beans are not initialized on parent context. Make sure that service facade and command processing modules are not marked as @Configuration . Beans defined there will still be initialized even without marking it.

# Property Placeholders

Spring property placeholders are configured in following order (first overrides last):

1. JVM System properties
2. Environment variables
3. Properties resolved from **classpath://local.properties** file
4. Properties resolved from **file://${GENESIS\_CONFIG\_PATH}/\*.properties** files
5. Properties resolved from**classpath://META-INF/config-${env}/\*.properties** files (since 21.14)
6. Properties resolved from **classpath://META-INF/config/\*-override.properties** files
7. Properties resolved from **classpath://META-INF/config/\*-default.properties** files

Default placeholder resolver has [Ordered.html#LOWEST\_PRECEDENCE](https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/core/Ordered.html" \l "LOWEST_PRECEDENCE) order that allows to define additional property placeholder location overrides by defining [PropertyPlaceholderConfigurer](https://docs.spring.io/spring/docs/current/javadoc-api/org/springframework/beans/factory/config/PropertyPlaceholderConfigurer.html) spring beans with higher precedence.

In order to define **external classpath** you can add additional location by setting an environment variable called ***loader.path.***More info about this environment variable can be found [here](https://docs.spring.io/spring-boot/docs/current/reference/html/appendix-executable-jar-format.html).

**Docker example:**

* Create folder with external properties files.
* Mount created folder to docker container by adding following command into **volumes** section:
* - "/path/source/on/host:/path/destination/on/container"
* Add loader.path environment variable with "/path/destination/on/container"  to **JAVA\_OPTS**configuration:
* "-Dloader.path=/path/destination/on/container"