# Links for reference

<https://github.com/spring-projects/spring-framework/wiki/What%27s-New-in-Spring-Framework-6.x>

<https://github.com/spring-projects/spring-framework/wiki/Upgrading-to-Spring-Framework-5.x>

<https://docs.spring.io/spring-framework/reference/core/beans/java/bean-annotation.html>

start here for full reference

<https://docs.spring.io/spring-framework/reference/overview.html>

## Bean creation topics

The @Import annotation in Spring Boot is used to **import one or more configuration classes**

Suppose if bean definitions are present in another configuration classes and if that is in another package u can import using

@Import({DatabaseConfig.class, SecurityConfig.class}) // Import multiple classes

public class MyMainConfig {

// ... other configuration details

}

2) Dynamically create & add/register a bean to spring context at runtime without using @ConditionalOnClass / @ConditionalOnProperty annotations & not at startup

<https://medium.com/@venkivenki4b6/spring-dynamically-register-beans-in-4-ways-at-run-time-c1dc45dcbeb9>

like if u want to add an existing object to the container then we should use this approach

• This can be done by below

• GenericBeanDefinition

• BeanDefinitionBuilder

• BeanFactoryPostProcessor

• BeanDefinitionRegistryPostProcessor

//here we are just giving the class name, means even the object also will be created by spring

|  |  |
| --- | --- |
| //Register a bean using BeanDefinitionBuilder  BeanDefinitionBuilder beanDefinition =BeanDefinitionBuilder.*rootBeanDefinition*(Hero.class)  .addPropertyValue("heroName","manideep")  .addPropertyValue("heroAddress","HYD");  GenericApplicationContext gac= (GenericApplicationContext) ac; gac.registerBeanDefinition("hero1",beanDefinition.getBeanDefinition());  Hero hero1 = ac.getBean(Hero.class); System.***out***.println("fetched bean definition from context"+hero1); |  |
| --This BeanFactoryPostProcessor. postProcessBeanFactory() methods will be called after all beans have loaded but no beans have been instantiated yet  public class MyConfigBean implements BeanFactoryPostProcessor {  @Override  public void postProcessBeanFactory (  ConfigurableListableBeanFactory beanFactory)  throws BeansException { GenericBeanDefinition bd = new GenericBeanDefinition();  bd.setBeanClass(MyBean.class);  bd.getPropertyValues().add("strProp", "my string property"); ((DefaultListableBeanFactory) beanFactory)  .registerBeanDefinition("myBeanName", bd);  } } | This BeanDefinitionRegistryPostProcessor. postProcessBeanDefinitionRegistry() also does the same thing but this focuses more on bean definition registration rather than general purpose beanFactory post processor  public class MyConfigBean implements BeanDefinitionRegistryPostProcessor { @Override  public void postProcessBeanDefinitionRegistry (BeanDefinitionRegistry registry)  throws BeansException {  GenericBeanDefinition bd = new GenericBeanDefinition();  bd.setBeanClass(MyBean.class);  bd.getPropertyValues().add("strProp", "my string property");  registry.registerBeanDefinition("myBeanName", bd);  } @Override  public void postProcessBeanFactory (ConfigurableListableBeanFactory beanFactory)  throws BeansException {  //no op  } } |

Bean Naming Conventions

The convention is to use the standard Java convention for instance field names when naming beans. That is, bean names start with a lowercase letter and are camel-cased from there. Examples of such names include accountManager, accountService, userDao, loginController, and so forth.

|  |  |
| --- | --- |
| By default, the bean name is the same as the method name in above  @Configuration  public class AppConfig {  @Bean  public TransferServiceImpl transferService() {  return new TransferServiceImpl();  }  } | * we can even write bean definitions in interface as default methods   & u can place @configuration annotation on an implementing class  So that if child classes wants to override they can override that default method  public interface BaseConfig {  @Bean  default TransferServiceImpl transferService() {  return new TransferServiceImpl();  }  }  @Configuration  public class AppConfig implements BaseConfig {  } |
| Bean Dependencies  @Bean method can have any number of parameters | @Configuration  public class AppConfig {  @Bean  public TransferService transferService(AccountRepository accountRepository) {  return new TransferServiceImpl(accountRepository);  }  } |
| Configuring the callbacks  In above  public class Hero {  String **heroName**;  String **heroAddress**;  int **heroAge** ;    public void initEnergies(){  System.***out***.println("init energies method called");  }  public void selfDestroy(){  System.***out***.println("self Destroy method called");  } } | 1. configuring initMethod() and destroy method() for @Bean methods   @Configuration public class AppConfiguration {  @Bean(name = "hero2", initMethod = "initEnergies",destroyMethod = "selfDestroy")  public Hero createHero(){  System.***out***.println("hero bean created");  return new Hero();  } }  Now spring while creating bean these initmethod(), destroy method will be called in hero class after creating the bean |
| Disabling the auto registering of destroy method  By default, beans defined with Java configuration that have a public close or shutdown method are automatically enlisted with a destruction callback. If you have a public close or shutdown method and you do not wish for it to be called when the container shuts down, you can add @Bean(destroyMethod = "") to your bean definition to disable the default (inferred) mode. | Some classes like datasource will have the close() method, This @Bean will register  That close method as destroy method automatically, so if u want to stop that behaviour then we have to explicitly declare destroy method as null  @Bean(destroyMethod = "")  public DataSource dataSource() throws NamingException {  return (DataSource) jndiTemplate.lookup("MyDS");  } |
| Configuring the scope to a bean | @Configuration  public class MyConfiguration {  @Bean  @Scope("prototype")  public Encryptor encryptor() {  *// ...*  }  } |
| Naming and aliasing a bean  Naming a bean    @Bean("myThing")  public Thing thing() {  return new Thing();  } | @Configuration  public class AppConfig {  //Aliasing  @Bean({"dataSource", "subsystemA-dataSource", "subsystemB-dataSource"})  public DataSource dataSource() {  *// instantiate, configure and return DataSource bean...*  }  } |

|  |  |
| --- | --- |
|  | With component scanning in the classpath, Spring generates bean names for unnamed components, following the rules described earlier: essentially, taking the simple class name and turning its initial character to lower-case. However, in the (unusual) special case when there is more than one character and both the first and second characters are upper case, the original casing gets preserved. These are the same rules as defined by java.beans.Introspector.decapitalize (which Spring uses here). |

1) Maven command to run the code or to start the appln

mvn spring-boot:run

2)Run your Spring Boot application using Gradle

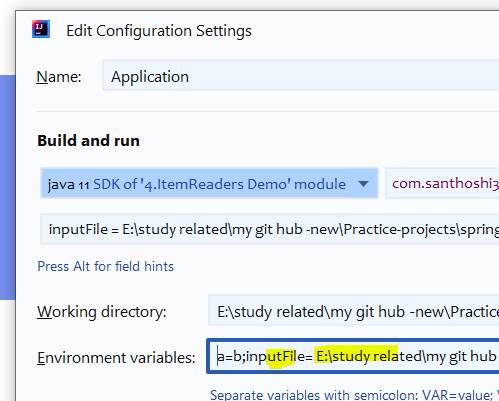
gradle bootRun

3) dynamic spring port

server.port=0 if u give 0 every time u start restart, then app will start on diff port number

Program or vm arguments passed to application can be fetch using ${} symbol

@Value("${inputFile}")  
String **fileLocation**;  
  
@Bean  
@StepScope  
public JsonItemReader<StudentJsonPOJO> itemReader(  
 @Value("#{jobParameters['inputFile']}") FileSystemResource *fileSystemResource*) {  
 System.***out***.println("parameters are -->" +*fileSystemResource*);  
 System.***out***.println("file location passed to vm args is --> "+**fileLocation**);



See the variables that are passed here we were able to fetch using $ symbol, means these data is injected to spring application context

By default All these variables and their values are injected to context, we can simply fetch using $