# Best practices

Never mix business logic with NFR’s / cross cutting concerns, write them seperately

For cross cutting concerns/ (NFR)Non functional requirements like logging, validation,or any logic other than business functionality keep that logic inside AOP

Ex:- if u want to find out how much time it took to execute controller method below are the ways

1. Either write System.currentTimeinMillis(); in controller first and last time and substract the ms and find the diff and log that or
2. Write that logic in separate AOP class – this is best , because we are not mixing the primary and sec business logics
3. Authentication should be implemented in separate class such as – filters/ interceptors
4. Caching – don’t mix caching logic with main business logic, keep both of them seperate

# 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>

|  |  |
| --- | --- |
| For property files | <https://docs.spring.io/spring-boot/reference/features/external-config.html#features.external-config.files> |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |

## 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). |

Running a spring boot application

|  |  |
| --- | --- |
| Running the application with **debug log statements**  Note:- Jar file will be created only when u do  Mvn clean install  Running jar with different properties | 1. java -jar myproject-0.0.1-SNAPSHOT.jar -–debug (in production I think this command will be executed via some script file to start the application ) 2. debug=true in your application.properties so that application will print all those debug logger statements as well   Note if u can open d code then u can keep this prop, but in prod or in any other env if u just have jar then use this flag  ex:- java -jar SpringBootDemo-0.0.1-SNAPSHOT.jar  it will just run the jar file – this will not again do compilation and all   1. u can give properties as well to this command so that those properties will be added and overridden in spring context with these values   **java -jar SpringBootDemo-0.0.1-SNAPSHOT.jar --app.name=mani**  **if u gave a key here this will be given with highest precedence, values kept in prop file will be overridden with these**   1. **if u have multiple properties , u can give as json**   java -jar myapp.jar --spring.application.json='{"my":{"name":"test"}}'  in urban code tool, we provide multiple key value pairs I think all those properties will be applied like above like  they might be added to java –jar command as a json , because individually if u want to add command will become too big  When your application starts, any spring.application.json or SPRING\_APPLICATION\_JSON properties will be parsed and added to the Environment.  For example, the SPRING\_APPLICATION\_JSON property can be supplied on the command line in a UN\*X shell as an environment variable:   1. f |
|  |  |
| Using maven | mvn spring-boot:run (here spring-boot is the plugin name)  Here this command will again compile & package the application again   1. we should keep that plugin in pom.xml the beauty is it will run the appln without creating jar file 2. it even supports hot code replace (but I didn’t tested it) |
| Run your Spring Boot application using Gradle | gradle bootRun  (here bootrun is the task name) |
| Running application on a dynamic spring port | server.port=0 if u give 0 every time u start restart, then app will start on diff port number |

Property files

1. Custom property file name

By default spring will recognize the property file name “application.properties”, If u changed that default name then you have to tell the property file name to spring

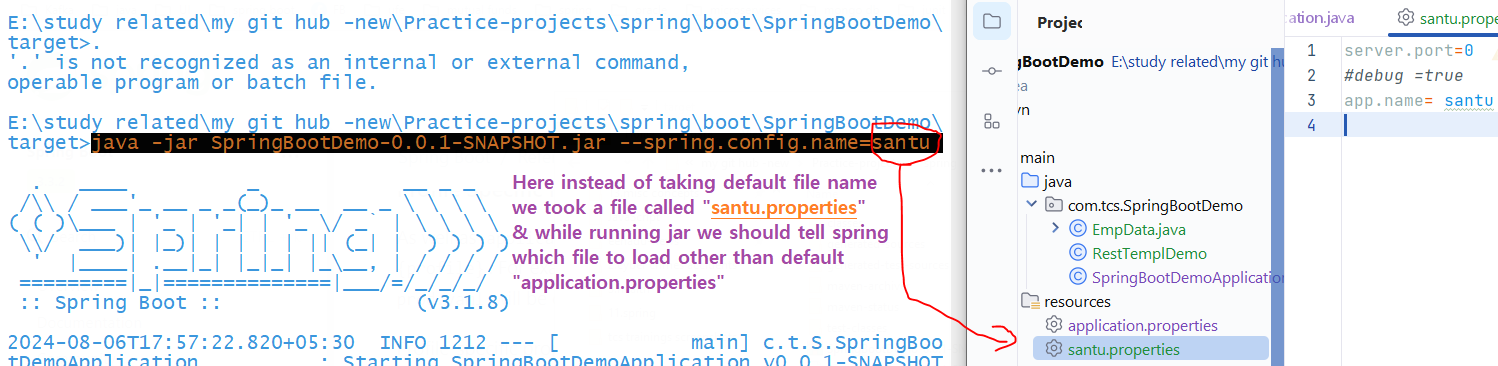
Using spring.config.name

Ex:- java -jar myproject.jar --spring.config.name=myproject

now it will load a file called “myproject.properties”

java -jar SpringBootDemo-0.0.1-SNAPSHOT.jar --spring.config.name=santu

This property u can give either as an argument while running jar or u can set to env variable



1. Custom location

The property file can be in any location if it is in classpath spring can easily recognize and it will load

But if it is in any other location, we should tell the path with “spring.config.location”

|  |  |
| --- | --- |
| Adding a location of property file | Adding additional locations |
| java -jar myproject.jar --spring.config.location=\  optional:classpath:/default.properties,\  optional:classpath:/override.properties   1. optional:classpath:custom-config/ 2. optional:file:./custom-config/   This I tried but it didn’t worked out | If you prefer to add additional locations, rather than replacing them, you can use spring.config.additional-location. Properties loaded from additional locations can override those in the default locations. For example, if spring.config.additional-location is configured with the value optional:classpath:/custom-config/,optional:file:./custom-config/, the complete set of locations considered is:   1. optional:classpath:/;optional:classpath:/config/ 2. optional:file:./;optional:file:./config/;optional:file:./config/\*/ 3. optional:classpath:custom-config/ 4. optional:file:./custom-config/ |

Use the prefix optional: if the [locations are optional](https://docs.spring.io/spring-boot/reference/features/external-config.html#features.external-config.files.optional-prefix) and you do not mind if they do not exist.

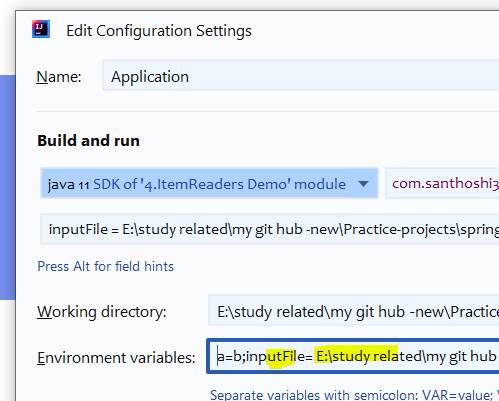
They must be defined as an environment property (typically an OS environment variable, a system property, or a command-line argument).

If spring.config.location contains directories (as opposed to files), they should end in /. At runtime they will be appended with the names generated from spring.config.name before being loaded. Files specified in spring.config.location are imported directly.

|  |  |
| --- | --- |
|  | Both directory and file location values are also expanded to check for [profile-specific files](https://docs.spring.io/spring-boot/reference/features/external-config.html#features.external-config.files.profile-specific). For example, if you have a spring.config.location of classpath:myconfig.properties, you will also find appropriate classpath:myconfig-<profile>.properties files are loaded. |

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 $

Using annotations

|  |  |
| --- | --- |
| @Configuration  @PropertySource("classpath:app.properties")  public class AppConfig { // ... } | By default if the property is in src/main/resources spring will load that property file  If that property is not in classpath then we have to use this annotation to explicitly load that property file |
| Properties  spring.config.additional-location=file:/path/to/your/config/directory | Using spring.config.additional-location Property:  * Define the spring.config.additional-location property in an existing property file (e.g., application.properties) to specify additional locations. |
| How to inject those value  Just use – “double quotes lo dollar” | @Component  public class MyBean {  @Value("${name}")  private String name; } |
| Command line properties are set to spring environment | By default spring appln adds all the command line arguments/ key-value pairs to spring environment  that is, arguments starting with --, such as --server.port=9000)  command line properties always take precedence over file-based property sources  ex:-  target>**java -jar SpringBootDemo-0.0.1-SNAPSHOT.jar --app.name=mani**  **I already gave a property called “**app.name= charan**” hardcoded in my property file, because of this command line property**  **Highest precedence will be given to this instead of hardcoded value In property file**  If you do not want command line properties to be added to the Environment, you can disable them by using SpringApplication.setAddCommandLineProperties(false) |

Spring Boot uses a very particular PropertySource order that is designed to allow sensible overriding of values. Later property sources can override the values defined in earlier ones. Sources are considered in the following order:

1. Default properties (specified by setting SpringApplication.setDefaultProperties).
2. [@PropertySource](https://docs.spring.io/spring-framework/docs/6.1.11/javadoc-api/org/springframework/context/annotation/PropertySource.html) annotations on your @Configuration classes. Please note that such property sources are not added to the Environment until the application context is being refreshed. This is too late to configure certain properties such as logging.\* and spring.main.\* which are read before refresh begins.
3. Config data (such as application.properties files).
4. A RandomValuePropertySource that has properties only in random.\*.
5. OS environment variables.
6. Java System properties (System.getProperties()).
7. JNDI attributes from java:comp/env.
8. ServletContext init parameters.
9. ServletConfig init parameters.
10. Properties from SPRING\_APPLICATION\_JSON (inline JSON embedded in an environment variable or system property).
11. Command line arguments.
12. properties attribute on your tests. Available on [@SpringBootTest](https://docs.spring.io/spring-boot/api/java/org/springframework/boot/test/context/SpringBootTest.html) and the [test annotations for testing a particular slice of your application](https://docs.spring.io/spring-boot/reference/testing/spring-boot-applications.html#testing.spring-boot-applications.autoconfigured-tests).
13. [@DynamicPropertySource](https://docs.spring.io/spring-framework/docs/6.1.11/javadoc-api/org/springframework/test/context/DynamicPropertySource.html) annotations in your tests.
14. [@TestPropertySource](https://docs.spring.io/spring-framework/docs/6.1.11/javadoc-api/org/springframework/test/context/TestPropertySource.html) annotations on your tests.
15. [Devtools global settings properties](https://docs.spring.io/spring-boot/reference/using/devtools.html#using.devtools.globalsettings) in the $HOME/.config/spring-boot directory when devtools is active.

Config data files are considered in the following order:

1. [Application properties](https://docs.spring.io/spring-boot/reference/features/external-config.html#features.external-config.files) packaged inside your jar (application.properties and YAML variants).
2. [Profile-specific application properties](https://docs.spring.io/spring-boot/reference/features/external-config.html#features.external-config.files.profile-specific) packaged inside your jar (application-{profile}.properties and YAML variants).
3. [Application properties](https://docs.spring.io/spring-boot/reference/features/external-config.html#features.external-config.files) outside of your packaged jar (application.properties and YAML variants).
4. [Profile-specific application properties](https://docs.spring.io/spring-boot/reference/features/external-config.html#features.external-config.files.profile-specific) outside of your packaged jar (application-{profile}.properties and YAML variants).