**SPRING BOOT**

Transforms how you approach Java programming tasks, radically streamlining your experience. Spring boot combines necessities such as an application context and an auto-configured, embedded web server to make microservice development a cinch. To go even faster, you can combine Spring Boot with Spring Cloud’s rich set of supporting libraries, servers, patterns, and templates, to safely deploy entire microservices-based architectures into the cloud, in record time.

Spring Boot offers a fast way to build applications. It looks at your classpath and at the beans you have configured, makes reasonable assumptions about what you are missing, and adds those items. With Spring Boot, you can focus more on business features and less on infrastructure.

The following examples show what Spring Boot can do for you:

* Is Spring MVC on the classpath? There are several specific beans you almost always need, and Spring Boot adds them automatically. A Spring MVC application also needs a servlet container, so Spring Boot automatically configures embedded Tomcat.
* Is Jetty on the classpath? If so, you probably do NOT want Tomcat but instead want embedded Jetty. Spring Boot handles that for you.
* Is Thymeleaf on the classpath? If so, there are a few beans that must always be added to your application context. Spring Boot adds them for you.

These are just a few examples of the automatic configuration Spring Boot provides. At the same time, Spring Boot does not get in your way. For example, if Thymeleaf is on your path, Spring Boot automatically adds a SpringTemplateEngine to your application context. But if you define your own SpringTemplateEngine with your own settings, Spring Boot does not add one. This leaves you in control with little effort on your part.

Spring Boot does not generate code or make edits to your files. Instead, when you start your application, Spring Boot dynamically wires up beans and settings and applies them to your application context.

You can see an easy project at our repository named “Demo”.

*SECURITY ON A WEB APPLICATION*

On this example we will build a Spring MVC application that secures the page with a login form that is backed by a fixed list of users.

We have an application without security at **SpringbootUnsecured** folder on our repository.

For the setting up security, if we want to prevent unauthorized users from viewing the greeting page at /hello. As it is now, if visitors click the link on the home page, they see the greeting with no barriers to stop them. You need to add a barrier that forces the visitor to sign in before they can see that page.

You do that by configuring Spring security in the application. If spring security is on the classpath, Spring boot automatically secures all HTTP endpoints with “basic” authentication. However, you can further customize the security settings. The first thing you need to do is add Spring Security to the classpath.

You can check the complete application in the **SpringbootSecured** with all the necessarily dependencies at the pom.xml file for maven and the correct way that this page should appear for every user visit our page.

**SPRING BOOT ACTIVE PROFILE**

Spring Boot supports different properties based on the Spring active profile. For example, we can keep two separate files for development and production to run the Spring Boot application.

*SPRING ACTIVE PROFILE IN APPLICATION.PROPERTIES*

By default, application,properties will be used to run the Spring boot application. If you want to use profile based properties, we can keep separate properties file for each profile as shown below

***application.properties***

server.port = 8080

spring.application.name = demoservice

***application-dev.properties***

server.port = 9090

spring.application.name = demoservice

***application-prod.properties***

server.port = 4431

spring.application.name = demoservice

While running the JAR file, we need to specify the spring active profile based on each properties file. By default, Spring Boot application uses the application.properties file. The command to set the spring active profile is shown below and the syntax applies for each properties file previously created.



*SPRING ACTIVE PROFILE FOR APPLICATION.YML*

You can keep the Spring active profile properties in the single application.yml file. No need to use the separate file like application.properties.

The following is an example code to keep the Spring active profiles in application.yml file.

Note that the delimiter (---) is used to separate each profile in application.yml file.

spring:

application:

name: demoservice

server:

port: 8080

---

spring:

profiles: dev

application:

name: demoservice

server:

port: 9090

---

spring:

profiles: prod

application:

name: demoservice

server:

port: 4431

As we saw before is the same command to set development active profile:



**SPRING BOOT LOGGING**

Spring Boot uses Apache Commons logging for all internal logging. Spring Boot’s default configurations provides a support for the use of Java Util Logging, Log4j2, and Logback. Using these, we can configure the console logging as well as file logging.

If you are using Spring Boot Starters, Logback will provide a good support for logging. Besides, Logback also provides a use of good support for Common Logging, Util Logging, Log4J, and SLF4J.

**SPRING BOOT EXCEPTION HANDLING**

Spring boot can achieve this with some tools such as:

*Controller advice.*

The @ControllerAdvice is an annotation used to handle the specific exceptions and sending the custom responses to the client.

*Exception Handler.*

The @ExceptionHandler is an annotation used to handle the specific exceptions and sending the custom responses to the client.

I recommend you check the repository at “Examples/SpringBootExceptionHandling” to get the code for the example at the below link and try it.

(Note: You’ll need to add the database dependency according to the one you decide to implement)

Complete example at:  
[Spring Boot - Exception Handling - GeeksforGeeks](https://www.geeksforgeeks.org/spring-boot-exception-handling/)

**SPRING DATA**

Spring Data’s provides a familiar and consistent, Spring-based programming model for data access while still retaining the special traits of the underlying data store.

It makes it easy to use data access technologies, relational and non-relational databases, map-reduce frameworks, and cloud-based data services. This is an umbrella project which contains many subprojects that are specific to a given database. The projects are developed by working together with many of the companies and developers that are behind these exciting technologies.

**SRPING CLOUD**

Spring Cloud provides tools for developers to quickly build some of the common patterns in distributed systems (e.g. configuration management, service discovery, circuit breakers, intelligent routing, micro-proxy, control bus, one-time tokens, global locks, leadership election, distributed sessions, cluster state). Coordination of distributed systems leads to boiler plate patterns and using Spring Cloud developers can quickly stand-up services and applications that implement those patterns. They will work well in any distributed environment, including the developer’s own laptop, bare metal data centers, and managed platforms such as Cloud Foundry.

A picture containing graphical user interface

Description automatically generated

*CONFIGURATION.*

Spring Cloud Config provides server and client-side support for externalized configuration in a distributed system. With the Config Server you have a central place to manage external properties for applications across all environments. The concepts on both client and server map identically to the **Spring Environment** and **PropertySource** abstractions, so they fit very well with Spring applications, but can be used with any application running in any language. As an application moves through the deployment pipeline from dev to test and into production you can manage the configuration between those environments and be certain that applications have everything they need to run when they migrate. The default implementation of the server storage backend uses git so it easily supports labelled versions of configuration environments, as well as being accessible to a wide range of tooling for managing the content. It is easy to add alternative implementations and plug them in with Spring configuration.

Spring Cloud Config Server features:

* HTTP, resource-based API for external configuration (name-value pairs, or equivalent YAML content)
* Encrypt and decrypt property values (symmetric or asymmetric)
* Embeddable easily in a Spring Boot application using @EnableConfigServer

Config Client features (for Spring applications):

* Bind to the Config Server and initialize Spring Environment with remote property sources
* Encrypt and decrypt property values (symmetric or asymmetric)

Check our Examples folder at our repository to watch how it works.

**NETFLIX EUREKA**

Eureka is a RESTful (Representational State Transfer) service that is primarily used in the AWS cloud for the purpose of discovery, load balancing and failover of middle-tier servers. It plays a critical role in Netflix mid-tier infra.

Client-side service discovery allows services to find and communicate with each other without hard-coding the hostname and port. The only ‘fixed point' in such an architecture is the service registry, with which each service has to register.

One drawback is that all clients must implement a certain logic to interact with this fixed point. This assumes an additional network round trip before the actual request.

With Netflix Eureka, each client can simultaneously act as a server to replicate its status to a connected peer. In other words, a client retrieves a list of all connected peers in a service registry and makes all further requests to other services through a load-balancing algorithm.

Spring Cloud common DiscoveryClient interrogate the services. The results contain information like the hostname and the port for each service.

Spring Cloud Feign integration is a handy project from Netflix that lets you describe a REST API client declaratively with annotations on an interface.

To get a better understanding of Eureka go to Examples/EurekaExample/EurekaImplementationExample.