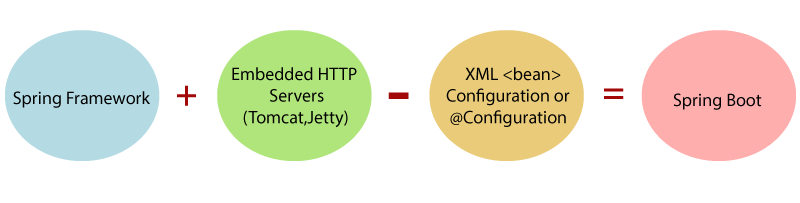
## What is Spring Boot

Spring Boot is a project that is built on the top of the Spring Framework. It provides an easier and faster way to set up, configure, and run both simple and web-based applications.

It is a Spring module that provides the **RAD (*Rapid Application Development*)** feature to the Spring Framework. It is used to create a stand-alone Spring-based application that you can just run because it needs minimal Spring configuration.



In short, Spring Boot is the combination of **Spring Framework** and **Embedded Servers**.

We can use Spring **STS IDE** or **Spring Initializr** to develop Spring Boot Java applications.

**Why should we use Spring Boot Framework?**

We should use Spring Boot Framework because:

* The dependency injection approach is used in Spring Boot.
* It contains powerful database transaction management capabilities.
* It simplifies integration with other Java frameworks like JPA/Hibernate ORM, Struts, etc.
* It reduces the cost and development time of the application.

Along with the Spring Boot Framework, many other Spring sister projects help to build applications addressing modern business needs. There are the following Spring sister projects are as follows:

* **Spring Data:** It simplifies data access from the relational and **NoSQL** databases.
* **Spring Batch:** It provides powerful **batch** processing.
* **Spring Security:** It is a security framework that provides robust **security** to applications.
* **Spring Social:** It supports integration with **social networking** like LinkedIn.
* **Spring Integration:** It is an implementation of Enterprise Integration Patterns. It facilitates integration with other **enterprise applications** using lightweight messaging and declarative adapters.

ADAdvantages of Spring Boot

* It creates **stand-alone** Spring applications that can be started using Java **-jar**.
* It tests web applications easily with the help of different **Embedded** HTTP servers such as **Tomcat, Jetty,** etc. We don't need to deploy WAR files.
* It provides opinionated '**starter**' POMs to simplify our Maven configuration.
* It provides **production-ready** features such as **metrics, health checks,** and **externalized configuration**.
* There is no requirement for **XML** configuration.
* It offers a **CLI** tool for developing and testing the Spring Boot application.
* It offers the number of **plug-ins**.
* It also minimizes writing multiple **boilerplate codes** (the code that has to be included in many places with little or no alteration), XML configuration, and annotations.
* It **increases productivity** and reduces development time.

## Limitations of Spring Boot

Spring Boot can use dependencies that are not going to be used in the application. These dependencies increase the size of the application.

ADGoals of Spring Boot

The main goal of Spring Boot is to reduce **development, unit test,** and **integration test** time.

* Provides Opinionated Development approach
* Avoids defining more Annotation Configuration
* Avoids writing lots of import statements
* Avoids XML Configuration.

By providing or avoiding the above points, Spring Boot Framework reduces **Development time, Developer Effort,** and **increases productivity**.

**Web Development**

It is a well-suited Spring module for web application development. We can easily create a self-contained HTTP application that uses embedded servers like **Tomcat, Jetty,** or Undertow. We can use the **spring-boot-starter-web** module to start and run the application quickly.AD

**SpringApplication**

The SpringApplication is a class that provides a convenient way to bootstrap a Spring application. It can be started from the main method. We can call the application just by calling a static run() method.

1. **public** **static** **void** main(String[] args)
2. {
3. SpringApplication.run(ClassName.**class**, args);
4. }

**Admin Support**

Spring Boot provides the facility to enable admin-related features for the application. It is used to access and manage applications remotely. We can enable it in the Spring Boot application by using **spring.application.admin.enabled** property.

**Externalized Configuration**

Spring Boot allows us to externalize our configuration so that we can work with the same application in different environments. The application uses YAML files to externalize configuration.

**Properties Files**

Spring Boot provides a rich set of **Application Properties**. So, we can use that in the properties file of our project. The properties file is used to set properties like **server-port =8082** and many others. It helps to organize application properties.

**YAML Support**

It provides a convenient way of specifying the hierarchical configuration. It is a superset of JSON. The SpringApplication class automatically supports YAML. It is an alternative of properties file.

**Type-safe Configuration**

The strong type-safe configuration is provided to govern and validate the configuration of the application. Application configuration is always a crucial task which should be type-safe. We can also use annotation provided by this library.

**Logging**

Spring Boot uses Common logging for all internal logging. Logging dependencies are managed by default. We should not change logging dependencies if no customization is needed.

**Security**

Spring Boot applications are spring bases web applications. So, it is secure by default with basic authentication on all HTTP endpoints. A rich set of Endpoints is available to develop a secure Spring Boot application.

## Performance Improvements

In Spring Boot 2.2.1 the following performance has been improved:

**Lazy Initialization**

In Spring Boot 2.2.1, we can enable global lazy initialization by using the property **spring.main.lazy-initialization** property. It reduces the application startup time.

**Java 13 Support**

Spring Boot 2.2.1 now supports the latest version of Java that is Java 13.

**Immutable Binding**

In the newer version of Spring Boot, Configuration properties support constructor-based binding. The class annotates with **@ConfigurationProperties** annotation is to be immutable. It can be enabled by adding an annotation **@ConfugurationProperties** to a class or one of its constructors with **@ConstructorBinding.**

**RSocket Support**

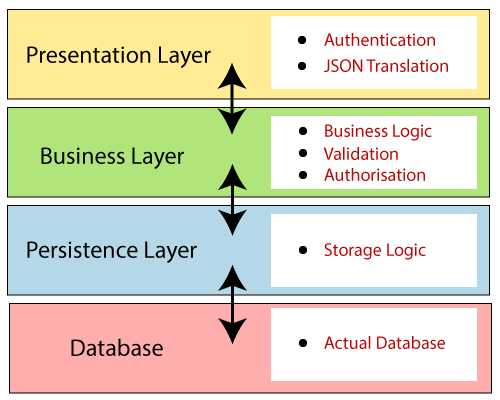
It is a part of **Spring Security**. RSocket integration is auto-configured when an application finds **spring-security-rsocket** is present on the classpath.

| **Spring** | **Spring Boot** |
| --- | --- |
| **Spring Framework** is a widely used Java EE framework for building applications. | **Spring Boot Framework** is widely used to develop **REST APIs**. |
| It aims to simplify Java EE development that makes developers more productive. | It aims to shorten the code length and provide the easiest way to develop **Web Applications**. |
| The primary feature of the Spring Framework is **dependency injection**. | The primary feature of Spring Boot is **Autoconfiguration**. It automatically configures the classes based on the requirement. |
| It helps to make things simpler by allowing us to develop **loosely coupled** applications. | It helps to create a **stand-alone** application with less configuration. |
| The developer writes a lot of code (**boilerplate code**) to do the minimal task. | It **reduces** boilerplate code. |
| we need to set up the sever explicitly. | Spring Boot offers **embedded server** such as **Jetty** and **Tomcat**. |
| It does not provide support for an in-memory database. | It offers several plugins for working with an embedded and **in-memory** database such as **H2**. |
| Developers manually define dependencies for the Spring project in **pom.xml**. | Spring Boot comes with the concept of **starter** in pom.xml file that internally takes care of downloading the dependencies **JARs** based on Spring Boot Requirement. |

| **Spring Boot** | **Spring MVC** |
| --- | --- |
| **Spring Boot** is a module of Spring for packaging the Spring-based application with sensible defaults. | **Spring MVC** is a model view controller-based web framework under the Spring framework. |
| It provides default configurations to build **Spring-powered** framework. | It provides **ready to use** features for building a web application. |
| There is no need to build configuration manually. | It requires build configuration manually. |
| There is **no requirement** for a deployment descriptor. | A Deployment descriptor is **required**. |
| It avoids boilerplate code and wraps dependencies together in a single unit. | It specifies each dependency separately. |
| It **reduces** development time and increases productivity. | It takes **more** time to achieve the same. |

# Spring Boot Architecture

Spring Boot follows a layered architecture in which each layer communicates with the layer directly below or above (hierarchical structure) it.



**Presentation Layer:** The presentation layer handles the HTTP requests, translates the JSON parameter to object, and authenticates the request and transfer it to the business layer. In short, it consists of **views** i.e., frontend part.

**Business Layer:** The business layer handles all the **business logic**. It consists of service classes and uses services provided by data access layers. It also performs **authorization** and **validation**.

**Persistence Layer:** The persistence layer contains all the **storage logic** and translates business objects from and to database rows.

**Database Layer:** In the database layer, **CRUD** (create, retrieve, update, delete) operations are performed.

# Spring Initializr

**Spring Initializr** is a **web-based tool** provided by the Pivotal Web Service. With the help of **Spring Initializr**, we can easily generate the structure of the **Spring Boot Project**. It offers extensible API for creating JVM-based projects.

## Generating a Project

Before creating a project, we must be friendly with UI. Spring Initializr UI has the following labels:

* **Project:** It defines the **kind** of project. We can create either **Maven Project** or **Gradle Project**. We will create a **Maven Project** throughout the tutorial.
* **Language:** Spring Initializr provides the choice among three languages **Java, Kotlin,** and **Groovy**. Java is by default selected.
* **Spring Boot:** We can select the Spring Boot **version**. The latest version is **2.2.2**.
* **Project Metadata:** It contains information related to the project, such as **Group**, Artifact, etc. Group denotes the **package** name; **Artifact** denotes the **Application** name. The default Group name is **com.example**, and the default Artifact name is **demo**.
* **Dependencies:** Dependencies are the collection of artifacts that we can add to our project.

There is another **Options** section that contains the following fields:

* **Name:** It is the same as **Artifact**.
* **Description:** In the description field, we can write a **description** of the project.
* **Package Name:** It is also similar to the **Group** name.
* **Packaging:** We can select the **packing** of the project. We can choose either **Jar** or **War**.
* **Java:** We can select the **JVM** version which we want to use. We will use **Java 8** version throughout the tutorial.

**@SpringBootApplication**

A single @SpringBootApplication annotation is used to enable the following annotations:AD

* **@EnableAutoConfiguration:** It enables the Spring Boot auto-configuration mechanism.
* **@ComponentScan:** It scans the package where the application is located.
* **@Configuration:** It allows us to register extra beans in the context or import additional configuration classes.

It is a class-level annotation. The class annotated with @Configuration used by Spring Containers as a source of bean definitions.

**@ComponentScan:** It is used when we want to scan a package for beans. It is used with the annotation @Configuration. We can also specify the base packages to scan for Spring Components.

**@Bean:** It is a method-level annotation. It is an alternative of XML <bean> tag. It tells the method to produce a bean to be managed by Spring Container.

**@Component:** It is a class-level annotation. It is used to mark a Java class as a bean. A Java class annotated with **@Component** is found during the classpath. The Spring Framework pick it up and configure it in the application context as a **Spring Bean**.

**@Controller:** The @Controller is a class-level annotation. It is a specialization of **@Component**. It marks a class as a web request handler. It is often used to serve web pages. By default, it returns a string that indicates which route to redirect. It is mostly used with **@RequestMapping** annotation.

**@Service:** It is also used at class level. It tells the Spring that class contains the **business logic**.

**@Repository:** It is a class-level annotation. The repository is a **DAOs** (Data Access Object) that access the database directly. The repository does all the operations related to the database.

* **@RequestMapping:** It is used to map the **web requests**. It has many optional elements like **consumes, header, method, name, params, path, produces**, and **value**. We use it with the class as well as the method.
* **@GetMapping:** It maps the **HTTP GET** requests on the specific handler method. It is used to create a web service endpoint that **fetches** It is used instead of using: **@RequestMapping(method = RequestMethod.GET)**
* **@PostMapping:** It maps the **HTTP POST** requests on the specific handler method. It is used to create a web service endpoint that **creates** It is used instead of using: **@RequestMapping(method = RequestMethod.POST)**
* **@PutMapping:** It maps the **HTTP PUT** requests on the specific handler method. It is used to create a web service endpoint that **creates** or **updates** It is used instead of using: **@RequestMapping(method = RequestMethod.PUT)**
* **@DeleteMapping:** It maps the **HTTP DELETE** requests on the specific handler method. It is used to create a web service endpoint that **deletes** a resource. It is used instead of using: **@RequestMapping(method = RequestMethod.DELETE)**
* **@PatchMapping:** It maps the **HTTP PATCH** requests on the specific handler method. It is used instead of using: **@RequestMapping(method = RequestMethod.PATCH)**
* **@RequestBody:** It is used to **bind** HTTP request with an object in a method parameter. Internally it uses **HTTP MessageConverters** to convert the body of the request. When we annotate a method parameter with **@RequestBody,** the Spring framework binds the incoming HTTP request body to that parameter.
* **@ResponseBody:** It binds the method return value to the response body. It tells the Spring Boot Framework to serialize a return an object into JSON and XML format.
* **@PathVariable:** It is used to extract the values from the URI. It is most suitable for the RESTful web service, where the URL contains a path variable. We can define multiple @PathVariable in a method.
* **@RequestParam:** It is used to extract the query parameters form the URL. It is also known as a **query parameter**. It is most suitable for web applications. It can specify default values if the query parameter is not present in the URL.
* **@RequestHeader:** It is used to get the details about the HTTP request headers. We use this annotation as a **method parameter**. The optional elements of the annotation are **name, required, value, defaultValue.** For each detail in the header, we should specify separate annotations. We can use it multiple time in a method
* **@RestController:** It can be considered as a combination of **@Controller** and **@ResponseBody** annotations**.** The @RestController annotation is itself annotated with the @ResponseBody annotation. It eliminates the need for annotating each method with @ResponseBody.
* **@RequestAttribute:** It binds a method parameter to request attribute. It provides convenient access to the request attributes from a controller method. With the help of @RequestAttribute annotation, we can access objects that are populated on the server-side.

#### we have specified only the Spring Boot version. If we want to add additional starters, simply remove the <version> tag. Similarly, we can also override the individual dependency by overriding a property in our project.

For example, if we want to add another dependency with the same artifact that we have injected already, inject that dependency again inside the **<properties>** tag to override the previous one.

<properties> <java.version>1.8</java.version> </properties>

### 

### Spring Boot without Parent POM

If we don't want to use **spring-boot starter-parent** dependency, but still want to take the advantage of the dependency management, we can use **<scope>** tag, as follows:

#### Note: It does not maintain the plugin management.

# Spring Boot Application Properties

we can use the application.properties file to:

* Configure the Spring Boot framework
* define our application custom configuration properties

### Example of application.properties

1. #configuring application name
2. spring.application.name = demoApplication
3. #configuring port
4. server.port = 8081

**YAML Properties File**

Spring Boot provides another file to configure the properties is called **yml** file. The Yaml file works because the **Snake YAML** jar is present in the classpath. Instead of using the application.properties file, we can also use the application.yml file, but the **Yml** file should be present in the classpath.

## Spring Boot Property Categories

There are **sixteen** categories of Spring Boot Property are as follows:

1. Core Properties
2. Cache Properties
3. Mail Properties
4. JSON Properties
5. Data Properties
6. Transaction Properties
7. Data Migration Properties
8. Integration Properties
9. Web Properties
10. Templating Properties
11. Server Properties
12. Security Properties
13. RSocket Properties
14. Actuator Properties
15. DevTools Properties
16. Testing Properties

## Application Properties Table

The following tables provide a list of common Spring Boot properties:

| **Property** | **Default Values** | **Description** |
| --- | --- | --- |
| Debug | false | It enables debug logs. |
| spring.application.name |  | It is used to set the application name. |
| spring.application.admin.enabled | false | It is used to enable admin features of the application. |
| spring.config.name | application | It is used to set config file name. |
| spring.config.location |  | It is used to config the file name. |
| server.port | 8080 | Configures the HTTP server port |
| server.servlet.context-path |  | It configures the context path of the application. |
| logging.file.path |  | It configures the location of the log file. |
| spring.banner.charset | UTF-8 | Banner file encoding. |
| spring.banner.location | classpath:banner.txt | It is used to set banner file location. |
| logging.file |  | It is used to set log file name. ex, data.log. |
| spring.application.index |  | It is used to set application index. |
| spring.application.name |  | It is used to set the application name. |
| spring.application.admin.enabled | false | It is used to enable admin features for the application. |
| spring.config.location |  | It is used to config the file locations. |
| spring.config.name | application | It is used to set config the file name. |
| spring.mail.default-encoding | UTF-8 | It used to set default MimeMessage encoding. |
| spring.mail.host |  | It is used to set SMTP server host. smtp.example.com. |
| spring.mail.password |  | It’s used to set login password of SMTP server. |
| spring.mail.port |  | It is used to set SMTP server port. |
| spring.mail.test-connection | false | It is used to test that the mail server is available on startup. |
| spring.mail.username |  | It is used to set login user of the SMTP server. |
| spring.main.sources |  | It is used to set sources for the application. |
| server.address |  | It is used to set network address to which the server should bind to. |
| server.connection-timeout |  | It is used to set time in milliseconds that connectors will wait for another HTTP request before closing the connection. |
| server.context-path |  | It is used to set context path of the application. |
| server.port | 8080 | It is used to set HTTP port. |
| server.server-header |  | It is used for the Server response header (no header is sent if empty) |
| server.servlet-path | / | It’s used to set path of main dispatcher servlet |
| server.ssl.enabled |  | It is used to enable SSL support. |
| spring.http.multipart.enabled | True | It used to enable support of multi-part uploads. |
| spring.servlet.multipart.max-file-size | 1MB | It is used to set max file size. |
| spring.mvc.async.request-timeout |  | It is used to set time in milliseconds. |
| spring.mvc.date-format |  | It is used to set date format. ex. dd/MM/yyyy. |
| spring.mvc.locale |  | It is used to set locale for the application. |
| spring.social.facebook.app-id |  | It is used to set application's Facebook App ID. |
| spring.social.linkedin.app-id |  | It is used to set application's LinkedIn App ID. |
| spring.social.twitter.app-id |  | It is used to set application's Twitter App ID. |
| security.basic.authorize-mode | role | It used to set security authorize mode to apply. |
| security.basic.enabled | true | It is used to enable basic authentication. |
| Spring.test.database.replace | any | Type of existing DataSource to replace. |
| Spring.test.mockmvc.print | default | MVC Print option |
| spring.freemaker.content-type | text/html | Content Type value |
| server.server-header |  | Value to use for the server response header. |
| spring.security.filter.dispatcher-type | async,error,request | Security filter chain dispatcher types. |
| spring.security.filter.order | -100 | Security filter chain order. |
| spring.security.oauth2.client.registration.\* |  | OAuth client registrations. |
| spring.security.oauth2.client.provider.\* |  | OAuth provider details. |

# Spring Boot Starters

**Spring Boot** provides a number of **starters** that allow us to add jars in the classpath. Spring Boot built-in **starters** make development easier and rapid. **Spring Boot Starters** are the **dependency descriptors**.

n the Spring Boot Framework, all the starters follow a similar naming pattern: **spring-boot-starter-\***, where **\*** denotes a particular type of application. For example, if we want to use Spring and JPA for database access, we need to include the **spring-boot-starter-data-jpa** dependency in our **pom.xml** file of the project.

The Spring Boot Framework provides the following application starters under the **org.springframework.boot** group.

| **Name** | **Description** |
| --- | --- |
| spring-boot-starter-thymeleaf | It is used to build MVC web applications using Thymeleaf views. |
| spring-boot-starter-data-couchbase | It is used for the Couchbase document-oriented database and Spring Data Couchbase. |
| spring-boot-starter-artemis | It is used for JMS messaging using Apache Artemis. |
| spring-boot-starter-web-services | It is used for Spring Web Services. |
| spring-boot-starter-mail | It is used to support Java Mail and Spring Framework's email sending. |
| spring-boot-starter-data-redis | It is used for Redis key-value data store with Spring Data Redis and the Jedis client. |
| spring-boot-starter-web | It is used for building the web application, including RESTful applications using Spring MVC. It uses Tomcat as the default embedded container. |
| spring-boot-starter-data-gemfire | It is used to GemFire distributed data store and Spring Data GemFire. |
| spring-boot-starter-activemq | It is used in JMS messaging using Apache ActiveMQ. |
| spring-boot-starter-data-elasticsearch | It is used in Elasticsearch search and analytics engine and Spring Data Elasticsearch. |
| spring-boot-starter-integration | It is used for Spring Integration. |
| spring-boot-starter-test | It is used to test Spring Boot applications with libraries, including JUnit, Hamcrest, and Mockito. |
| spring-boot-starter-jdbc | It is used for JDBC with the Tomcat JDBC connection pool. |
| spring-boot-starter-mobile | It is used for building web applications using Spring Mobile. |
| spring-boot-starter-validation | It is used for Java Bean Validation with Hibernate Validator. |
| spring-boot-starter-hateoas | It is used to build a hypermedia-based RESTful web application with Spring MVC and Spring HATEOAS. |
| spring-boot-starter-jersey | It is used to build RESTful web applications using JAX-RS and Jersey. An alternative to spring-boot-starter-web. |
| spring-boot-starter-data-neo4j | It is used for the Neo4j graph database and Spring Data Neo4j. |
| spring-boot-starter-data-ldap | It is used for Spring Data LDAP. |
| spring-boot-starter-websocket | It is used for building the WebSocket applications. It uses Spring Framework's WebSocket support. |
| spring-boot-starter-aop | It is used for aspect-oriented programming with Spring AOP and AspectJ. |
| spring-boot-starter-amqp | It is used for Spring AMQP and Rabbit MQ. |
| spring-boot-starter-data-cassandra | It is used for Cassandra distributed database and Spring Data Cassandra. |
| spring-boot-starter-social-facebook | It is used for Spring Social Facebook. |
| spring-boot-starter-jta-atomikos | It is used for JTA transactions using Atomikos. |
| spring-boot-starter-security | It is used for Spring Security. |
| spring-boot-starter-mustache | It is used for building MVC web applications using Mustache views. |
| spring-boot-starter-data-jpa | It is used for Spring Data JPA with Hibernate. |
| spring-boot-starter | It is used for core starter, including auto-configuration support, logging, and YAML. |
| spring-boot-starter-groovy-templates | It is used for building MVC web applications using Groovy Template views. |
| spring-boot-starter-freemarker | It is used for building MVC web applications using FreeMarker views. |
| spring-boot-starter-batch | It is used for Spring Batch. |
| spring-boot-starter-social-linkedin | It is used for Spring Social LinkedIn. |
| spring-boot-starter-cache | It is used for Spring Framework's caching support. |
| spring-boot-starter-data-solr | It is used for the Apache Solr search platform with Spring Data Solr. |
| spring-boot-starter-data-mongodb | It is used for MongoDB document-oriented database and Spring Data MongoDB. |
| spring-boot-starter-jooq | It is used for jOOQ to access SQL databases. An alternative to spring-boot-starter-data-jpa or spring-boot-starter-jdbc. |
| spring-boot-starter-jta-narayana | It is used for Spring Boot Narayana JTA Starter. |
| spring-boot-starter-cloud-connectors | It is used for Spring Cloud Connectors that simplifies connecting to services in cloud platforms like Cloud Foundry and Heroku. |
| spring-boot-starter-jta-bitronix | It is used for JTA transactions using Bitronix. |
| spring-boot-starter-social-twitter | It is used for Spring Social Twitter. |
| spring-boot-starter-data-rest | It is used for exposing Spring Data repositories over REST using Spring Data REST. |

## Spring Boot Production Starters

| **Name** | **Description** |
| --- | --- |
| spring-boot-starter-actuator | It is used for Spring Boot's Actuator that provides production-ready features to help you monitor and manage your application. |
| spring-boot-starter-remote-shell | It is used for the CRaSH remote shell to monitor and manage your application over SSH. Deprecated since 1.5. |

## Spring Boot Technical Starters

| **Name** | **Description** |
| --- | --- |
| spring-boot-starter-undertow | It is used for Undertow as the embedded servlet container. An alternative to spring-boot-starter-tomcat. |
| spring-boot-starter-jetty | It is used for Jetty as the embedded servlet container. An alternative to spring-boot-starter-tomcat. |
| spring-boot-starter-logging | It is used for logging using Logback. Default logging starter. |
| spring-boot-starter-tomcat | It is used for Tomcat as the embedded servlet container. Default servlet container starter used by spring-boot-starter-web. |
| spring-boot-starter-log4j2 | It is used for Log4j2 for logging. An alternative to spring-boot-starter-logging. |

## Spring Boot Starter Parent :- **The spring-boot-starter-parent is a project starter. It provides default configurations for our applications. It is used internally by all dependencies. All Spring Boot projects use spring-boot-starter-parent as a parent in pom.xml file.**

**Spring Boot Dependencies**

The spring-boot-starter-parent dependency inherit from the spring-boot-dependencies, it shares all these characteristics as well. Hence the Spring Boot manages the list of the dependencies as the part of the dependency management.

## Spring Boot Starter without Parent

In some cases, we need not to inherit spring-boot-starter-parent in the pom.xml file. To handle such use cases, Spring Boot provides the flexibility to still use the dependency management without inheriting the spring-boot-starter-parent.

In the above code, we can see that we have used **<scope>** tag for this. It is useful when we want to use different version for a certain dependency.

Parent Poms allow us to manage the following things for multiple child projects and modules:

* **Configuration:** It allows us to maintain consistency of Java Version and other related properties.
* **Dependency Management:** It controls the versions of dependencies to avoid conflict.
* Source encoding
* Default Java Version
* Resource filtering
* It also controls the default plugin configuration.

# Spring Boot Starter Web

There are two important features of spring-boot-starter-web:

* It is compatible for web development
* Auto configuration

If we want to develop a web application, we need to add the following dependency in pom.xml file:

Starter of Spring web uses Spring MVC, REST and Tomcat as a default embedded server.

The spring-boot-starter-web transitively depends on the following:

* org.springframework.boot:spring-boot-starter
* org.springframework.boot:spring-boot-starter-tomcat
* org.springframework.boot:spring-boot-starter-validation
* com.fasterxml.jackson.core:jackson-databind
* org.springframework:spring-web
* org.springframework:spring-webmvcAD

By default, the spring-boot-starter-web contains the tomcat server dependency:

We can also use **spring-mvc** without using the embedded Tomcat server. If we want to do so, we need to exclude the Tomcat server by using the **<exclusion>** tag

### 

### Spring Data JPA Features

There are **three** main features of Spring Data JPA are as follows:

* **No-code repository:** It is the most popular persistence-related pattern. It enables us to implement our business code on a higher abstraction level.
* **Reduced boilerplate code:** It provides the default implementation for each method by its repository interfaces. It means that there is no longer need to implement read and write operations.
* **Generated Queries:** Another feature of Spring Data JPA is the **generation of database queries** based on the method name. If the query is not too complex, we need to define a method on our repository interface with the name that starts with **findBy**. After defining the method, Spring parses the method name and creates a query for it.

## Spring Data Repository

Spring Data JPA provides **three** repositories are as follows:

* **CrudRepository:** It offers standard **create, read, update,** and **delete** It contains method like **findOne(), findAll(), save(), delete(),** etc.
* **PagingAndSortingRepository:** It extends the **CrudRepository** and adds the findAll methods. It allows us to **sort** and **retrieve** the data in a paginated way.
* **JpaRepository:** It is a **JPA specific repository** It is defined in **Spring Data Jpa**. It extends the both repository CrudRepository and PagingAndSortingRepository. It adds the JPA-specific methods, like **flush()** to trigger a flush on the persistence context.

### Hibernate vs. JPA

Hibernate is the implementation of JPA. It is the most popular ORM framework, while JPA is an API that defines the specification. Hibernate understands the mapping that we add between objects and tables. It ensures that data is retrieved/ stored from the

There are **three** main features of Spring Boot Actuator:

* **Endpoints**
* **Metrics**
* **Audit**

**Endpoint:** The actuator endpoints allows us to monitor and interact with the application.

| **Id** | **Usage** | **Dfult** |
| --- | --- | --- |
| actuator | It provides a hypermedia-based **discovery page** for the other endpoints. It requires Spring HATEOAS to be on the classpath. | True |
| auditevents | It exposes audit events information for the current application. | True |
| autoconfig | It is used to display an auto-configuration report showing all auto-configuration candidates and the reason why they 'were' or 'were not' applied. | True |
| beans | It is used to display a complete list of all the Spring beans in your application. | True |
| configprops | It is used to display a collated list of all @ConfigurationProperties. | True |
| dump | It is used to perform a thread dump. | True |
| env | It is used to expose properties from Spring's ConfigurableEnvironment. | True |
| flyway | It is used to show any Flyway database migrations that have been applied. | True |
| health | It is used to show application health information. | False |
| info | It is used to display arbitrary application info. | False |
| loggers | It is used to show and modify the configuration of loggers in the application. | True |
| liquibase | It is used to show any Liquibase database migrations that have been applied. | True |
| metrics | It is used to show metrics information for the current application. | True |
| mappings | It is used to display a collated list of all @RequestMapping paths. | True |
| shutdown | It is used to allow the application to be gracefully shutdown. | True |
| trace | It is used to display trace information. | True |

We can disable the security feature of the actuator by adding the following statement.

**application.properties**

1. management.security.enabled=false

# Spring Boot DevTools

The aim of the module is to try and improve the development time while working with the Spring Boot application. Spring Boot DevTools pick up the changes and restart the application.

Spring Boot DevTools provides the following features:

* **Property Defaults**
* **Automatic Restart**
* **LiveReload**
* **Remote Debug Tunneling**
* **Remote Update and Restart**

**Property Defaults:** Spring Boot provides templating technology **Thymeleaf** that contains the property **spring.thymeleaf.cache.** It disables the caching and allows us to update pages without the need of restarting the application. But setting up these properties during the development always creates some problems.

**Automatic Restart:** Auto-restart means reloading of Java classes and configure it at the server-side. After the server-side changes, it deployed dynamically, server restarts happen, and load the modified code. It is mostly used in microservice-based applications. Spring Boot uses **two** types of ClassLoaders:

* The classes that do not change (third-Jars) are loaded in the **base ClassLoader.**
* The classes that we are actively developing are loaded in the **restart ClassLoader.**

When the application restarts, the restart ClassLoader is thrown away, and a new one is populated. Therefore, the base ClassLoader is always available and populated.

We can disable the auto-restart of a server by using the property **spring.devtools.restart.enabled** to **false.**

### Remember:

* The DevTools always monitors the classpath resources.
* There is only a way to trigger a restart is to update the classpath.
* DevTools required a separate application classloader to work properly. By default, Maven fork the application process.
* Auto-restart works well with **LiveReload.**
* DevTools depends on the application context's shutdown hook to close it during the restart.

**LiveReload:** The Spring Boot DevTools module includes an embedded server called **LiveReload.** It allows the application to automictically trigger a browser refresh whenever we make changes in the resources. It is also known as **auto-refresh.**

**Note:** We can disable the LiveReload by setting the property **spring.devtools.livereload.enabled** to **false.**

### Remember

* We can run one LiveReload server at a time.
* Before starting the application, ensure that no other LiveReload server is running.
* If we start multiple applications from IDE, it supports only the first LiveReload.

**Remote Update and Restart:** There is another trick that DevTools offers is: it supports remote application **updates** and **restarts.** It monitors local classpath for file changes and pushes them to a remote server, which is then restarted. We can also use this feature in combination with LiveReload.

## 

## Using a Trigger File

Automatic restart sometimes can slow down development time due to frequent restarts. To remove this problem, we can use a **trigger file.** Spring Boot monitors trigger file and detects modifications in that file. It restarts the server and reloads all previous changes.

# Spring Boot Multi-Module Project

a **multi-module project** is built from a parent pom that manages a group of submodules. Or A **multi-module project** is defined by a parent POM referencing one or more submodules.

The parent maven project must contain the packaging type **pom** that makes the project as an aggregator. The **pom.xml** file of the parent project consists the list of all **modules, common dependencies,** and **properties** that are inherited by the child projects. The parent pom is located in the project's root directory. The child modules are actual Spring Boot projects that inherit the maven properties from the parent project.

When we run the multi-module project, all the modules are deployed together in an embedded Tomcat Server. We can deploy an individual module, also.

## Parent POM

The parent POM defines the **Group ID, Artifact ID, version**, and **packaging.** In the previous Maven projects, we have seen that the parent POM defines the packaging **jar.** But in the multi-module project, the parent **POM** defines the packaging pom. The packaging pom refers to other Maven projects.

We only need to specify all the dependencies in the parent pom. All the other modules share the same pom, so we need not to specify the same dependency in each module separately. It makes the code easier to keep in order with a big project.

## Child module-ear, war, and jar

we are creating an **EAR** (Enterprise ARchive), **WAR** (Web ARchive), and **JAR** (Java ARchive) file. A JAR file is bundled into a war file that is bundled into an EAR file. The EAR file is the final package that is ready to deploy on the application server.

The EAR file contains one or many WAR files. Each WAR file contains the service project that has common code to all WAR files and packaging type in the JAR.

When we create Maven Modules in the project, Spring Boot automatically configures the modules in the parent pom inside the **module** tag, as shown below.

1. **<modules>**
2. **<module>**module1**</module>**
3. **<module>**module2**</module>**
4. **</modules>**

## WAR

**WAR** stands for **Web Archive.** WAR file represents the web application. Web module contains servlet classes, JSP files, HTML files, JavaScripts, etc. are packaged as a JAR file with .**war** extension. It contains a special directory called **WEB-INF**.

WAR is a module that loads into a web container of the Java Application Server. The Java Application Server has **two** containers: **Web Container** and **EJB Container**.

The **Web Container** hosts the web applications based on Servlet API and JSP. The web container requires the web module to be packaged as a WAR file. It is a WAR file special JAR file that contains a **web.xmlv** file in the **WEB-INF** folder.

## JAR

In other words, A file that encapsulates one or more Java classes, a manifest, and descriptor(XML file) is called JAR file. It is the lowest level of the archive.

## EAR

**EAR** stands for **Enterprise Archive.** EAR file represents the enterprise application. It can contain multiple EJB modules (JAR) and Web modules (WAR). It is a special JAR that contains an **application.xml** file in the **META-INF** folder.

HOW TO BUILD JAR

1. In pom.xml add tag <packaging>jar</packaging>.
2. you can build the maven project using command **mvn package** or **maven install.** It will create the .jar file inside target folder.
3. run command from location where .jar is present - java -jar jarName.jar

# Spring Boot Auto-configuration

Spring Boot auto-configuration automatically configures the Spring application based on the jar dependencies that we have added.

For example, if the H2 database Jar is present in the classpath and we have not configured any beans related to the database manually, the Spring Boot's auto-configuration feature automatically configures it in the project.

## Disable Auto-configuration Classes

We can also disable the specific auto-configuration classes, if we do not want to be applied. We use the **exclude** attribute of the annotation @EnableAutoConfiguration to disable the auto-configuration classes. For example:

1. @Configuration(proxyBeanMethods = false)
2. @EnableAutoConfiguration(exclude={DataSourceAutoConfiguration.class})
3. public class MyConfiguration { }

We can exclude any number of auto-configuration classes by using the property **spring.autoconfigure.exclude**.

## AOP

### AOP **(Aspect-Oriented Programming)** is a programming pattern that increases modularity by allowing the separation of the **cross-cutting concern**. These cross-cutting concerns are different from the main business logic. **We can add additional behavior to existing code without modification of the code itself.**

| **Spring AOP** | **AspectJ** |
| --- | --- |
| There is a need for a separate compilation process. | It requires the AspectJ compiler. |
| It supports only method execution pointcuts. | It supports all pointcuts. |
| It can be implemented on beans managed by Spring Container. | It can be implemented on all domain objects. |
| It supports only method level weaving. | It can wave fields, methods, constructors, static initializers, final class, etc. |

@EnableAspectJAutoProxy(proxyTargetClass=true)

### Object-Relation Mapping (ORM)

### In ORM, the mapping of Java objects to database tables, and vice-versa is called **Object-Relational Mapping.** The ORM mapping works as a bridge between a **relational database** (tables and records) and **Java application** (classes and objects).

## JDBC Connection Pooling

**JDBC connection pooling** is a mechanism that manages **multiple** database connection requests. In other words, it facilitates connection reuse, a memory cache of database connections, called a **connection pool.** A connection pooling module maintains it as a layer on top of any standard JDBC driver product.

It also improves the performance of an application. Connection pool performs the following tasks:

* Manage available connection
* Allocate new connection
* Close connection

### HikariCP

* If the HikariCP is present on the classpath, the Spring Boot automatically configures it.
* If the HikariCP is not found on the classpath, Spring Boot looks for the **Tomcat JDBC Connection Pool.** If it is on the classpath Spring Boot, pick it up.
* If both the above options are not available, Spring Boot chooses **Apache Commons DBCP2** as the JDBC connection pool.

| **Spring Boot JDBC** | **Spring JDBC** |
| --- | --- |
| There is only a **spring-boot-starter-jdbc** dependency is required. | In Spring JDBC, multiple dependencies need to be configured like **spring-jdbc** and **spring-context.** |
| It automatically configures Datasource bean, if not maintain explicitly. If we do not want to use the bean, we can set a property **spring.datasource.initialize** to **false**. | In Spring JDBC, it is necessary to create a database bean either using **XML** or **javaconfig**. |
| We do not need to register Template beans because Spring Boot automatically registers beans. | The Template beans such as **PlatformTransactionManager, JDBCTemplate, NamedParameterJdbcTemplate** must be registered. |
| Any db initialization scripts stored in .sql file gets executed automatically. | If any db initialization scripts like dropping or creation of tables are created in SQL file, this info needs to be given explicitly in the configuration. |

## JDBC vs. Hibernate

| **JDBC** | **Hibernate** |
| --- | --- |
| JDBC is a **technology**. | Hibernate is an **ORM** framework. |
| In JDBC, the user is responsible for creating and closing the connections. | In Hibernate, the run time system takes care of creating and closing the connections. |
| It does not support lazy loading. | It supports lazy loading that offers better performance. |
| It does not support associations (the connection between two separate classes). | It supports associations. |

## Persistence vs. In-memory Database

### The persistent database persists the data in physical memory. The data will be available even if the database server is bounced. Some popular persistence databases are [**Oracle**](https://www.javatpoint.com/oracle-tutorial)**,** [**MySQL**](https://www.javatpoint.com/mysql-tutorial)**,** [**Postgres**](https://www.javatpoint.com/postgresql-tutorial)**,** etc.

### In the case of the **in-memory database,** data store in the **system memory**. It lost the data when the program is closed. It is helpful for **POC**s (Proof of Concepts), not for a production application. The widely used in-memory database is **H2.**

## What is the H2 Database

### **H2** is an **embedded, open-source,** and **in-memory** database. It is a relational database management system written in [Java](https://www.javatpoint.com/java-tutorial). It is a **client/server** application. It is generally used in **unit testing**. It stores data in memory, not persist the data on disk.

### **Advantages**

### Zero configuration

### It is easy to use.

### It is lightweight and fast.

### It provides simple Configuration to switch between a real database and in-memory database.

### It supports standard SQL and JDBC API.

### It provides a web console to maintain in the database.

## CrudRepository vs. JpaRepository

| **CrudRepository** | **JpaRepository** |
| --- | --- |
| CrudRepository does not provide any method for pagination and sorting. | It extends PagingAndSortingRepository. It provides all the methods for implementing the pagination. |
| It works as a **marker** interface. | JpaRepository extends both **CrudRepository** and **PagingAndSortingRepository**. |
| It provides CRUD function only. For example **findById(), findAll(),** etc. | It provides some extra methods along with the method of PagingAndSortingRepository and CrudRepository. For example, **flush(), deleteInBatch().** |
| It is used when we do not need the functions provided by JpaRepository and PagingAndSortingRepository. | It is used when we want to implement pagination and sorting functionality in an application. |

### 

## What is Thymeleaf?

It is a **HTML5/XHTML/XML** template engine. It is a **server-side Java template** engine for both web (servlet-based) and non-web (offline) environments.

The goal of Thymeleaf is to provide a **stylish** and **well-formed** way of creating templates. It is based on XML tags and attributes.

The architecture of Thymeleaf allows the **fast** **processing** of templates that depends on the caching of parsed files. It uses the least possible amount of I/O operations during execution.

To process files in this specific mode, Thymeleaf performs a transformation that converts files into a **well-formed XML** file (valid HTML5 file).

#### Note: In Thymeleaf, validation is available only for XHTML and XML template.

# Spring Boot Caching

In Spring, the **cache abstraction** is a mechanism that allows consistent use of various caching methods with minimal impact on the code.

## Cache Abstraction

The cache abstraction mechanism applies to [**Java**](https://www.javatpoint.com/java-tutorial) **methods**. The main objective of using cache abstraction is to **reduce** the number of executions based on the information present in the cache. It applies to expensive methods such as **CPU** or **IO bound.**

Every time, when a method invokes, the abstraction applies a cache behavior to the method. It checks whether the method has already been executed for the given argument or not.

* If yes, the cached result is returned without executing the actual method.
* If no, first, the method executes, and the result is cached and returned to the user.

#### Note: This approach works only for the methods that are guaranteed to return the same result for a given input. It does not matter how many times the method executes.

The developers take care of two things while working with cache abstractions.

* **Cache Declaration:** It identifies the methods that need to be cached.
* **Cache Configuration:** The backing cache where the data is stored and read from.

### Caching

Caching is a part of temporary memory ([RAM](https://www.javatpoint.com/ram-full-form)). It lies between the application and persistence database. It stores the recently used data that reduces the number of database hits as much as possible. In other words, caching is to store data for future reference.

## What data should be cached?

* The data that do not change frequently.
* The frequently used read query in which results does not change in each call, at least for a period.

## Types of Caching

There are **four** types of caching are as follows:

* In-memory Caching
* Database Caching
* Web server Caching
* CDN Caching

### In-memory Caching

In-memory caching increases the performance of the application. It is the area that is frequently used. [**Memcached**](https://www.javatpoint.com/memcached-tutorial) and **Redis** are examples of in-memory caching. It stores key-value between application and database. Redis is an **in-memory, distributed,** and advanced caching tool that allows backup and restore facility. We can manage cache in distributed clusters, also.

### Database Caching

Database caching is a mechanism that generates web pages on-demand (dynamically) by fetching the data from the database. It is used in a **multi-tier** environment that involved clients, web-application server, and database. It improves **scalability** and **performance** by distributing a query workload. The most popular database caching is the first level cache of [Hibernate](https://www.javatpoint.com/hibernate-tutorial).

### Web Server Caching

Web server caching is a mechanism that stores data for **reuse**. For example, a copy of a web page served by a web server. It is cached for the first time when a user visits the page. If the user requests the same next time, the cache serves a copy of the page.

CDN Caching

The **CDN** stands for **Content Delivery Network**. It is a component used in modern web applications. It improves the delivery of the content by **replicating** commonly requested files (such as [HTML](https://www.javatpoint.com/html-tutorial) Pages, stylesheet, [JavaScript](https://www.javatpoint.com/javascript-tutorial), images, videos, etc.) across a globally distributed set of **caching servers.**

## Cache vs. Buffer

| **Cache** | **Buffer** |
| --- | --- |
| The cache is based on **Least Recently Used**. | Buffer is based on **First-In-First-Out.** |
| It is the size of the page cache. | It is an in-memory raw block I/O buffer. |
| It lived for a **long** period. | It lived for a **short** period. |
| We **read** from the cache. | We **write** into the buffer. |
| It stores the **actual** file data. | It stores the file **metadata**. |
| It improves **read** performance. | It improves **write** performance. |

## Spring Boot Cache Annotations

### @EnableCaching

It is a class-level annotation. We can enable caching in the Spring Boot application by using the annotation **@EnableCaching.** It is defined in **org.springframework.cache.annotation** package. It is used together with **@Configuration** class.

The auto-configuration enables caching and setup a **CacheManager,** if there is no already defined instance of CacheManager. It scans for a specific provider, and when it does not find, it creates an in-memory cache using concurrent **HashMap.**

### @CacheConfig

It is a class-level annotation that provides a common cache-related setting. It tells the Spring where to store cache for the class.

**@CacheConfig(cacheNames={"employee"})**

1. **public class UserService**
2. **{ }**

### @Caching

It is used when we need both annotations **@CachePut** or **@CacheEvict** at the same time on the same method. In other words, it is used when we want to use multiple annotations of the same type.

But **Java does not allow multiple annotations** of the same type to be declared for a given method. To avoid this problem, we use **@Caching** annotation.

In the following example, we have used the annotation **@Caching** and grouped all the **@CacheEvict** annotations.

1. @Caching(evict = {@CacheEvict("phone\_number"), @CacheEvict(value="directory", key="#student.id") })
2. **public** String getAddress(Student student) { }

### @Cacheable

It is a method level annotation. It defines a cache for a method's return value. The Spring Framework manages the requests and responses of the method to the cache that is specified in the annotation attribute.

We can also apply a condition in the annotation by using the condition attribute. When we apply the condition in the annotation, it is called **conditional caching**.

1. @Cacheable(value="student", condition="#name.length<20")
2. **public** Student findStudent(String name) { }

### @CacheEvict

It is a method level annotation. It is used when we want to remove stale or unused data from the cache. It requires one or multiple caches that are affected by the action. We can also specify a key or condition into it. If we want wide cache eviction, the @CacheEvict annotation provides a parameter called **allEntries**. It evicts all entries rather than one entry based on the key.

it can be used with void methods because the method acts as a trigger. It avoids return values. On the other hand, the annotation @Cacheable requires a return value that adds/updates data in the cache.

### @CachePut

It is a method level annotation. It is used when we want to **update** the cache without interfering the method execution. It means the method will always execute, and its result will be placed into the cache. It supports the attributes of @Cacheable annotation.

**There is a slight difference between @Cacheable and @CachePut annotation is that the** @**Cacheable** annotation **skips the method execution** while the **@CachePut** annotation **runs the method** and put the result into the cache.

# Spring Boot Cache Provider

The Spring Boot framework allows the integration of various **cache providers,** such as **EhCache, Redis, Hazelcast, Infinispan, Caffeine,** etc.

The auto-configuration of caching includes the following steps:

* Add the annotation **@EnableCaching** in the configuration file.
* Add the required **caching libraries** in the classpath.
* In the root of the classpath, add the **configuration file** for the cache provider.

For example, if we want to implement **EhCache** in an application, first we enable the cache in the configuration file.

1. @SpringBootApplication
2. @EnableCaching
3. **public** **class** Employee {
4. @Bean
5. **public** CacheManager cacheManager() { } }

At the end, configure the file for cache provider. Here, we are using the EhCache so need to configure **ehcache.xml** file at the root of the classpath.

If the Spring Boot finds the more than one cache provider in the classpath, in such cases, we must specify the cache provider explicitly in the **application.properties** file.

1. spring.cache.ehcache.provider=net.sf.ehcache.CacheManager
2. spring.cache.ehcache.config=classpath:config/another-config.xml

We can set up a particular cache provider by using the property **spring.cache.type**. It is used in a certain environment if we want to disable caching.

1. spring.cache.type=none

The Spring Boot Framework automatically configures the CacheManager that can be further customized by implementing the **CacheManagerCustomizer** interface.

1. @Bean
2. **public** CacheManagerCustomizer<ConcurrentMapCacheManager> cacheManagerCustomizer() {
3. **return** **new** CacheManagerCustomizer<ConcurrentMapCacheManager>() {
4. @Override
5. **public** **void** customize(ConcurrentMapCacheManager cacheManager) {
6. cacheManager.setAllowNullValues(**false**); } }; }

The above bean expects a auto-configured **ConcurrentMapCacheManager**. If the ConcurrentMapCacheManager is not auto-configures, the customizer will not invoke in any way. We can have any number of customizer and arrange them in order by using the annotation **@Order** or **@Ordered.**

EhCache used a file called **ehcache.xml.**We can configure the [XML](https://www.javatpoint.com/what-is-xml) file by using the following property:

1. **spring.cache.ehcache.config=classpath:config/demo-config.xml**

# Spring Boot EhCaching

## EhCache

EhCache is an open-source, Java-based cache used to boost performance.

## Features of EhCache

* It is **fast**, **lightweight, Scalable,** and **Flexible**.
* It allows us to perform **Serializable** and **Object**
* It offers cache eviction policies such as **LRU, LFU, FIFO,**
* It stores the cache in **memory** and **disk** (SSD).
* It depends on **SLF4J** for logging.
* It has a full implementation of **JSR-107** and **Jcache**
* It supports distributed caching via **JGroups** or **JMS** and **RMI**.
* It uses **fluent query language** for distributed search

## EhCache Usage Patterns

The cache uses several access patterns. There are following patterns used by EhCache:

* **Cache-aside**
* **Cache-as-SoR (system-of-record)**
* **Read-through**
* **Write-through**
* **Write-behind**

### Cache-aside

In the **cache-aside** pattern, first, the application consults with the cache. If the data is found, it returns the data directly. In the opposite scenario, it fetches the data from the SoR, stores it into the cache, and then return.

### Cache-as-SoR

The **cache-as-SoR** pattern represents SoR reading and writing operations to the cache. It reduces the responsibility of the application. It uses the combination of read and write pattern that includes **read-through, write-through,** and **write-behind.** It reduces the difficulty of the application. It allows the cache to solve the thundering-herd problem

### Read-through

The **read-through** pattern also copies the cache-aside pattern while reading data from the cache. The difference between the read-through and cache-aside is that read-through pattern implements the **CacheEntryFactory** interface. It guides the cache how to read an object from the cache. It is better to wrap the EhCache instance with the instance of **SelfPopulatingCache** while using the read-through pattern.

### Write-through

The **write-through** pattern also copies the cache-aside pattern while writing data in the cache. The difference between write-through and cache-aside pattern is that write-through pattern implements the **CacheWriter** interface. It configures the cache for both write-through and write-behind pattern. It writes data to the SoR in the same thread of execution.

### Write-behind

The **write-behind** pattern is different form the other three patterns. It modifies the cache entries after a **configurable delay**. The delay may in **seconds, minutes, a day, a week,** or for a **long time**. Simultaneously, it also queues the data to write at a later time in the same thread of execution.

The data write using write-behind pattern happens outside of the scope of the transaction. It means that it creates a new transaction to commit the data in the SoR that is distinct from the main transaction.

## EhCaching Storage Tiers

EhCache allows us to use various data storage areas, such as heap, disk and clustered. We can configure a multi-storage cache (uses more than one storage area). It can be arranged and managed as **tiers.**

The tiers are organized in order. The bottom-most tier is known as **authority tier,** and the other tier is known as the **caching tier**. It is also known as **nearer** or **near cache.** The caching tier can have more than one storage area. The hottest data kept in the caching tier because it is faster than the authority tier. Other data is kept in the authority tier that is slower but richer in comparison to the caching tier.

There are **four** types of data storage supported by EhCache:

* **On-Heap Store**
* **Off-Heap Store**
* **Disk Store**
* **Clustered Store**

### On-Heap Store

It stores cache entries in Java heap memory. It shares the storage with [Java](https://www.javatpoint.com/java-tutorial) application. It is fast because it uses heap but has limited storage space. The garbage collector also scans the on-heap store.

### Off-Heap Store

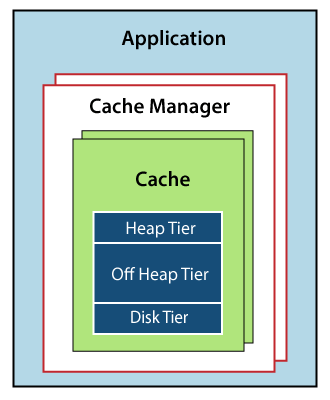
It uses the primary memory (RAM) to store cache entries. The garbage collector does not scan it. It is slower than the on-heap store because the cache entries move to the on-heap store before use. It is limited in size.

### Disk Store

It uses a disk to store cache entries. It is much slower than [RAM](https://www.javatpoint.com/ram-full-form)-based stores (on and off-heap store). It is better to use a dedicated disk if you are using a disk store pattern. It enhances throughput.

### Clustered Store

It stores cache entries on the remote server. It is slower than off-heap storage. It may have a failover server that provides high availability.



The above diagram shows that:

* An application may have more than one Cache Manager.
* Many caches can be handled by a Cache Manager.
* The caches can use more than one tier for storing cache entries.
* EhCache puts the recently used or frequently used data in the faster tier (caching tier).

# Spring Boot Change Port

* Using **application.properties** file
* Using **application.yml** file
* Using **EmbeddedServletContainerCustomizer** Interface
* Using **WebServerFactoryCustomizer** Interface
* Using **Command-Line Parameter**

**Application.properties–** server.port=8081

**We can also set the port property to 0. It scans the random port for the application. It uses a new port whenever we restart our application.**

## Using application.yml file

**server:**

**Port:8082**

## Using Command Line Parameter

right-click on the application file -< Run As -< Run Configurations.

Write **-Dserver.port=9001** in the **VM arguments**