**@Autowired** is used for automatic dependency injection. It tells Spring to automatically resolve and inject the collaborating bean (i.e., a dependency) into a class.

**In Spring Boot , @Qualifier is used with @Autowired to resolve the ambiguity when there are multiple beans of the same type in the application context.**

**@Autowired @Qualifier("car")**

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

**@ComponentScan:**  The @ComponentScan annotation tells Spring **where to look for components**, like: @Component, @Service, @Repository, @Controller

By default, Spring Boot automatically scans the package where your main application class resides and all its sub-packages. But you can customize it using @ComponentScan.

**@SpringBootApplication:** It has following annotations:AD

* **@EnableAutoConfiguration:**  It auto-configures the application based on the dependencies on the classpath.ex: Spring Web,JPA etc
* **@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.

**@Bean:** It is a method-level annotation. It tells the method to produce a bean to be managed by Spring Container.

**@Component:** is a core **annotation** in Spring Framework used to define a class as a **Spring-managed bean**.

**@Controller:** 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 tells the Spring that class contains the **business logic**.

**@Repository:** The repository is a **DAOs** that accesses 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:** @RequestBody is used to **bind the body of an HTTP request** to a **Java object**. It is commonly used in POST, PUT, and PATCH APIs where data is sent in JSON format.

@PostMapping

public String createUser(@RequestBody User user) {}

* **@ResponseBody:** It binds the method return value to the http response body. It tells the Spring Boot Framework to serialize a return an object into JSON and XML format.

@GetMapping("/user")

@ResponseBody

public User getUser() {

User user = new User("Alice", "alice@example.com");

return user;

}

* **@PathVariable:** It is used to extract the values from the URI and bind them to method parameter.

@GetMapping("/user/{id}")

public String getUserById(@PathVariable("id") int userId) {}

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 binds values from the **query string** into method parameters. 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.

@GetMapping("/greet")

public String greetUser(@RequestParam String name) {}

* **@RequestHeader:** @RequestHeader is used to **extract values from HTTP headers** and bind them to method parameters in a controller.

By default, @RequestHeader is **required**.If the header is missing, you’ll get a 400 Bad Request.

public String getToken(@RequestHeader(name = "X-Auth-Token", required = false) String token) {}

* **@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.

**First set the vaule:** request.setAttribute("userId", 42); // setting value

public String getUser(@RequestAttribute("userId") int userId) {}

|  |  |  |  |
| --- | --- | --- | --- |
| Method | Task |  | Ex |
| POST | Create | Create a **new resource**. | Create a new Entry |
| PUT | Fully Update | **Replace** an existing resource **entirely**. | Replaces the entire user resource. If a field is missing  , it may be **overwritten or nullified**. |
| PATCH | Partially Update | **Modify part** of an existing resource. | Updates just the email, keeping other fields unchanged. |

**consumes —** What content type the method can accept as input

**produces —** What content type it will return as output

@RestController

@RequestMapping("/api")

public class UserController {

@PostMapping(value = "/user", consumes = "application/json", produces = "application/json")

public User createUser(@RequestBody User user) {

// process user and return response

return user;

}

}

**Difference Between @RequestParam vs @PathVariable**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **Feature** | | | **@RequestParam** | | | **@PathVariable** |
| **Used for** | | | Extracting values from query parameters | | | Extracting values from the URI path |
| **Typical Use** | | | Optional or filter/search parameters | | | Identifiers like user ID, product ID, etc. |
| **URL Pattern Example** | | | /users?name=Alice | | | /users/Alice |
| **Annotation syntax** | | | @RequestParam("name") String name | | | @PathVariable("name") String name |
|  | | | part of the **URL query string** | | | part of the **URL path** |
| **Default Values** | | | Can use defaultValue and required=false | | | No default; always required unless handled manually |
|  | | |  | | |  |
|  | | |  | | |  |
| **Feature** | **@RequestParam** | | **@PathVariable** | **@RequestAttribute** | |
| **Source** | URL query string | | URL path | Request attributes (middleware/interceptor) | |
| **Syntax** | ?name=John | | /users/101 | request.setAttribute("key", value) | |
| **Spring Annotation** | @RequestParam("name") | | @PathVariable("id") | @RequestAttribute("userId") | |
| **Typical Use** | Filters, searches, optional input | | Resource identification (ID, name) | Pass data from filters/interceptors | |
| **Optional Support** | Yes (required=false, defaultValue) | | No (required by default) | Yes (required=false) | |
| **Use in URL** | /search?query=java | | /user/123 | Not visible in URL | |
| **Set By** | Client/user in request | | Client/user in request | Server-side filter/interceptor | |

You also can enforce default content type globally via **Spring config** in **application.properties**

spring.mvc.contentnegotiation.favor-path-extension=false

spring.mvc.contentnegotiation.media-types.json=application/json

spring.mvc.default-content-type=application/json

If you want to support XML, include the Jackson XML dependency: “jackson-dataformat-xml”

**@EnableScheduling** is a Spring annotation used to **enable scheduled task execution** in your application.

When you annotate a configuration class(Main Class) with @EnableScheduling, it tells Spring:

Scan for methods annotated with @**Scheduled** and run them based on the defined schedule.”

**What is @ModelAttribute?**

In **Spring MVC**, @ModelAttribute is used for:

1. **Binding HTTP request parameters to a Java object** (e.g., form submission)
2. **Exposing objects to the model** (used in views like JSP, Thymeleaf, etc.)

It works with both GET and POST methods.

@PostMapping("/form")

public String submitForm(@ModelAttribute User user) {}

**@ExceptionHandler** is an annotation used in **Spring MVC** to handle exceptions **locally or globally** and provide a custom response when an exception occurs during request handling.

When a controller method throws an exception, Spring looks for a method annotated with @ExceptionHandler that matches the **type of exception**, and calls that method to handle it.

**Local Exception Handling in a Controller:**

@ExceptionHandler(IllegalArgumentException.class)

public ResponseEntity<String> handleIllegalArg(IllegalArgumentException ex) {}

**Global Exception Handling:** apply @ControllerAdvice on a class, then define a method for the exception.

**@ControllerAdvice:** It applies to all the controllers.

**ResponseEntity**: is a class in Spring used to represent the entire HTTP response, including: Status code, Headers, body, etc.

It allows full control over the HTTP response that is returned from a controller method.

**return ResponseEntity.status(HttpStatus.OK).body(data);**

**@ResponseStatus** is used to mark a class or method with a specific HTTP status code. It's commonly used to:

* Set a custom status code on exceptions
* Define default HTTP responses in REST APIs

@ResponseStatus(HttpStatus.NOT\_FOUND)

public class UserNotFoundException extends RuntimeException {

public UserNotFoundException(String message) {}

**@EnableAsync**

@EnableAsync is used to **enable asynchronous method execution**. When a method is annotated with @Async, it runs **in a separate thread**, allowing the caller to continue without waiting for the method to finish. **ex: email sending**

**@EnableCaching:** @EnableCaching is an annotation used to enable Spring’s annotation-driven caching mechanism in your application**.**

When you add @EnableCaching to a configuration class, Spring:

* Looks for methods annotated with caching annotations like:
  + @**Cacheable**: Stores method return value in cache
  + @**CachePut**: Updates cache after method call
  + @**CacheEvict**: Removes entry from cache
  + @**Caching**: Combines multiple caching operations

@Cacheable("products") // Cache result using 'products' as cache name

public Product getProductById(Long id) {}

@CacheEvict(value = "products", key = "#id")

public void deleteProduct(Long id) {}

**@Caching:** It allow you to apply multiple cache annotations (@**Cacheable**, @**CachePut**, @**CacheEvict**) on the same method.

This is useful when:

* You need to **cache using multiple keys**
* You want to **evict and put** in the cache at once
* You want to combine cache operations in a clean, structured way

@Caching(

cacheable = {

@Cacheable(value = "cache1", key = "#id"),

@Cacheable(value = "cache2", key = "#name")

},

put = {

@CachePut(value = "cache3", key = "#result.id")

},

evict = {

@CacheEvict(value = "cache4", key = "#id")

}

)

## **Maven Dependency Management System**

The **Dependency Management System** is one of its core features, which simplifies how external libraries (JARs) are included and managed in your project.

* Avoids manual JAR downloads
* Ensures version consistency
* Centralizes library version control

**spring-boot-starter-parent:**

It provides **default configurations**, **dependency management**, and **plugin management** so you don’t have to configure everything manually in pom.xml.

|  |  |
| --- | --- |
| **Feature** | **Description** |
| **Dependency management** | Centralized versions for common libraries like Jackson, Tomcat, Logback, etc. |
| **Plugin management** | Configures plugins like maven-compiler-plugin, spring-boot-maven-plugin |
| **Default Java version** | Typically sets default compiler version (e.g., Java 17) |
| **Test setup** | Configures test dependencies (JUnit, AssertJ) |
| **Version locking** | You don’t need to specify versions for most common Spring Boot dependencies |

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 within <dependencyManagement>..

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

**<scope>** tag: controls **how and when** a dependency is available (compile, test, runtime, etc.).

**application.properties vs application.yml**

|  |  |  |
| --- | --- | --- |
| **Feature** | **application.properties** | **application.yml (YAML)** |
| **Format Type** | Key-Value pairs | Hierarchical (indentation-based) |
| **Readability** | Simple, but less readable for nested config | Cleaner and more readable for complex config |
| **Support for Complex Structures** | Not very elegant | Very elegant (especially for lists/maps) |
| **Multiline Values** | Requires \ or inline | Native support |
| **Profile Management** | application-dev.properties | application-dev.yml |
| **Error Sensitivity** | Less error-prone | Sensitive to indentation and spacing |
| **Learning Curve** | Easier for beginners | Slightly harder if new to YAML |
| Ex | |  | | --- | |  | | | |  | | --- | |  | | |
|  |
|  |

**Spring Boot Property Categories**

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

**Core, Cache, Mail, JSON, Data, Transaction, Data Migration, Integration, Web, Templating, Server, Security, RSocket, Actuator, DevTools,Testing Properties**

|  |  |  |
| --- | --- | --- |
| **Property** | **Default Values** | **Description** |
| Debug | FALSE | It enables debug logs. |
| spring.application.name |  | 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 locatio |
| server.port | 8080 | Configures the HTTP server port |
| server.servlet.context-path |  | It configures the context path of the application. |
| logging.file.path |  | configures the location of the log file. |
| spring.banner.charset | UTF-8 | Banner file encoding. |
| spring.banner.location | classpath:banner.txt | used to set banner file location. |
| logging.file |  | It is used to set log file name. For example, data.log. |
| spring.application.index |  | It is used to set application index. |
| 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 is used to set default MimeMessage encoding. |
| spring.mail.host |  | It is used to set SMTP server host. For example, smtp.example.com. |
| spring.mail.password |  | It is used to set login password of the 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 |  | 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 |  | used for Server response header (no header is sent if empty) |
| server.servlet-path | / | It is used to set path of the main dispatcher servlet |
| server.ssl.enabled |  | It is used to enable SSL support. |
| spring.http.multipart.enabled | TRUE | It is 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. For example, 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 is 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 Starters are a set of convenient dependency descriptors (pre-defined pom.xml templates) you can include in your project.

**spring-boot-starter-actuator:** module **adds production-ready features** to help you monitor and manage your Spring Boot application..

|  |  |
| --- | --- |
| **Feature** | **Description** |
| **/actuator/health** | Shows application health info (disk space, DB, etc.) |
| **/actuator/metrics** | Exposes metrics like JVM memory, CPU, HTTP requests, etc. |
| **/actuator/info** | Shows custom app info from application.yml |
| **/actuator/env** | Displays environment properties (careful with sensitive data) |
| **/actuator/loggers** | Dynamically view/change logging levels |
| **/actuator/beans** | Displays all Spring beans in the context |
| **/actuator/mappings** | Shows all HTTP request mappings |
| **/actuator/heapdump** | Dumps the memory heap (useful for memory analysis) |
| **/actuator/threaddump** | Thread dump of running threads |

**spring-boot-starter-web**

When you add this starter to your project, it automatically pulls in:

|  |  |
| --- | --- |
| **Dependency** | **Purpose** |
| **Spring MVC** | Web framework for building REST APIs and web apps |
| **Jackson** | JSON serialization/deserialization |
| **Tomcat (embedded)** | Default embedded web server (can be replaced with Jetty) |
| **Validation API** | For request data validation via @Valid, @NotNull, etc |

**spring-boot-starter-web vs. spring-boot-starter-tomcat**

The spring-boot-starter-web contains the spring web dependencies that includes spring-boot-starter-tomcat.

While the **spring-boot-starter-tomcat** contains everything related to Tomcat server.

Ex. Core, el, logging etc

**<exclusion>** tag is used to exclude transitive dependencies—dependencies that are brought in indirectly by other dependencies(from starter). Ex: replace tomcat server.

**Spring Boot Data JPA**

Spring Data JPA adds its own features such as the no-code implementation of the repository pattern and **the creation of database queries from the method name.**

|  |  |
| --- | --- |
| **Feature** | **Description** |
| JpaRepository | Built-in CRUD methods |
| Query methods | Auto-generate queries from method names |
| JPQL / Native Query | Custom queries with @Query |
| Transaction management | Automatic via @Transactional |
| Pagination & Sorting | Out-of-the-box support |

Hibernate is the implementation of JPA.

**@Transactional**

The @Transactional annotation in Spring is used to **manage database transactions** declaratively. It ensures that a **method executes within a transaction context**, and it can **commit or roll back** automatically depending on the success or failure of the method.

**Spring Boot DevTools**

**Spring Boot DevTools** is a development-time tool that enhances the **developer experience** by providing features like:

|  |  |
| --- | --- |
| **Feature** | **Description** |
| **Automatic Restart** | Restarts the application whenever classpath files change.ex Service or controller etc |
| **LiveReload** | Triggers browser refresh when static resources (HTML/CSS/JS) change |
| **Property Defaults** | Changes default behavior to speed up development |
| **Remote Debug Support** | (Optional) Supports remote app reload |
| **Disables Caching** | For templates (Thymeleaf, FreeMarker), etc. |

**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.

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.

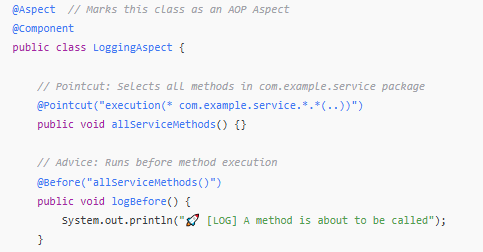
**Spring Boot AOP (Aspect-Oriented Programming)**

Spring Boot AOP helps to separate **cross-cutting concerns** (like logging, security, transactions, etc.) from the main business logic.

**Use Cases of AOP in Spring Boot**

* Logging method calls
* Security checks
* Transaction management
* Caching
* Performance monitoring

|  |  |  |
| --- | --- | --- |
| **Term** | **Description** | **Example** |
| **Aspect** | Class containing cross-cutting logic | LoggingAspect |
| **Advice** | Advice is the actual code that is executed at a Join Point. (ex. before or after a method). | @Before, @After |
| **Join Point** | Where an advice can run (e.g., method call) | saveUser() method call |
| **Pointcut** | Expression that selects join points | execution(\* com.example..\*) |
| **Weaving** | Connecting aspects to the app’s actual flow (in runtime) | Spring proxy intercepting method |



|  |  |  |
| --- | --- | --- |
| **Advice Type** | **When It Runs** | **Use Case Example** |
| @Before | Before method | Validation, Logging |
| @After | After method (always) | Cleanup logic |
| @AfterReturning | After successful execution | Auditing |
| @AfterThrowing | If exception occurs | Error handling, alerting |
| @Around | Before and after (full control) | Transactions, Security, Metrics |

**@EnableAspectJAutoProxy:** It enables support for handling components marked with @**Aspect**. It tells Spring to look for classes annotated with @Aspect and create proxies for them using AspectJ (annotation-style AOP).

In **Spring Boot**, **don’t need to add it manually** because it's **enabled by default** via @SpringBootApplication.

|  |  |  |
| --- | --- | --- |
| **Feature / Aspect** | **JSP** | **Thymeleaf** |
| **Definition** | Java-based server-side template engine | Modern XML/XHTML/HTML5 template engine |
| **Spring Boot Compatibility** | Not officially supported in Spring Boot (complex setup) | Fully supported with Spring Boot |
| **Template Type** | .jsp | .html |
| **Rendering Mode** | Server-side only | Server-side + Static rendering in browsers |
| **Readable in Browser** | Not directly (raw tags not visible) | Yes – renders as normal HTML |
| **Expression Language** | JSTL, EL (${}) | Thymeleaf Standard Expressions (${}, \*{}) |
| **HTML Validity** | Often breaks HTML validity | 100% HTML5-compliant |
| **Learning Curve** | Moderate (older syntax and JSTL tags) | Easy and intuitive (especially for front-end devs) |
| **UI Designer Friendly** | No – needs servlet container | Yes – viewable as static HTML |
| **Performance** | Decent, but outdated | Optimized for Spring + modern features |

**Why we use Thymeleaf?**

JSP is more or less similar to HTML. But it is not completely compatible with HTML like Thymeleaf. We can open and display a Thymeleaf template file normally in the browser while the JSP file does not.

**Caching**

Caching is a part of temporary memory ([RAM](https://www.tpointtech.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.

**Why should we use the cache?**

The primary reason for using cache is to make data access faster and less expensive. When the highly requested resource is requested multiple times,

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**

It is the area that is frequently used. It stores key-value between application and database.

**Ex Tool:** Java HashMap, Spring @Cacheable (default), Ehcache (in-process),Redis

**Use Case:** Fast access to session data, configurations, or user preferences.

**Database Caching**

Database caching is a technique used to **store frequently accessed data** closer to the application, thereby **reducing the load on the database** and improving performance. Instead of querying the database every time,

Ex: User Profile

**Web Server Caching**

Web server caching is a mechanism that stores data for **reuse**. Ex: when a user visits the page for the first time. a copy of a web page created by a web server. If the user requests the same next time, the cache serves a copy of the page. Web server caching enhances the page delivery speed and reduces the work to be done by the backend server.

**CDN Caching**

It improves the delivery of the content by **replicating** commonly requested files (such as [HTML](https://www.tpointtech.com/html-tutorial) Pages, stylesheet, [JavaScript](https://www.tpointtech.com/javascript-tutorial), images, videos, etc.) across a globally distributed set of **caching servers.**

It delivers a local copy of the content from a nearby **cache edge** (a cache server that is closer to the end-user), or a **Point of Presence (PoP)**.

|  |  |  |  |
| --- | --- | --- | --- |
| **Type** | **Stored At** | **Best For** | **Examples** |
| In-memory Caching | App RAM | Fastest access, small frequent data | Ehcache, Guava |
| Database Caching | Near DB/app layer | Reducing DB load | Hibernate Cache, Redis |
| Web Server Caching | Web server | Full page or partial response cache | Nginx, Apache |
| CDN Caching | Edge servers | Global content delivery | Cloudflare, CloudFront |

**Cache vs. Buffer**

|  |  |
| --- | --- |
| **Cache** | **Buffer** |
| The cache is based on **Least Recently Used**. | Buffer is based on **First-In-First-Out.** |
| 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. |

**Hibernate First-Level Cache**

All entities loaded within a **session** are stored in memory (cache) so that subsequent access to the same entity doesn’t hit the database again during that **session**.

|  |  |
| --- | --- |
| **Feature** | **Description** |
| Scope | **Session-scoped** (per Hibernate session) |
| Enabled | **By default**, no configuration needed |
| Storage | Stored in memory inside the Session object |
| Invalidated | When the session is **closed**, the cache is **cleared** |
| Usage | Improves performance by **avoiding redundant DB calls** |

**Internal working:** When you load an entity using session.get() or session.load(), Hibernate **checks its first-level cache** (i.e., the Session object). If the entity is found in the cache (by primary key), it's **returned directly**. Otherwise, Hibernate **fetches it from the database**, and then **stores it in the cache** for future use within the same session.

**Limitations:**

* **Session-scoped only**: Cache does not survive beyond the session.
* Not shared between multiple sessions.
* If the entity is updated outside the session (like by another process), it can **become stale**.

**Hibernate Second-Level Cache**

Second-Level Cache (2LC) in Hibernate is a **shared cache** for entities across multiple **Session objects**.  
Unlike First-Level Cache, which is session-scoped, **Second-Level Cache** is **session factory–scoped** and persists beyond individual sessions.

It stores data in a cache provider (like Ehcache, Redis, Infinispan) **so that repeated queries or entity loads don’t hit the database** even in different sessions.

|  |  |  |
| --- | --- | --- |
| **Feature** | **First-Level Cache** | **Second-Level Cache** |
| Scope | Per Session | Per SessionFactory |
| Default | Always enabled | Optional (needs setup) |
| Shared Across Sessions | No | Yes |
| Eviction | When session ends | Manual / configured |
| Configuration | Not needed | Required via annotations and config |

**How It Works:**

1. App opens a Session. Hibernate first checks **First-Level Cache**.
2. If not found → Hibernate checks **Second-Level Cache** (shared region).
3. If still not found → Hibernate hits the **database**.
4. Once fetched from DB → data is stored in both **First-Level** and **Second-Level** caches.

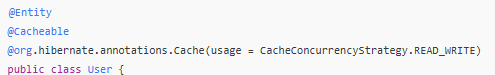
How to Enable Second-Level Cache

1. Add Cache Provider Dependency (e.g., Ehcache 3)

2. Hibernate Configuration (application.properties / yml)

* spring.jpa.properties.hibernate.cache.use\_second\_level\_cache=true
* spring.jpa.properties.hibernate.cache.region.factory\_class=org.hibernate.cache.jcache.JCacheRegionFactory
* spring.cache.jcache.config=classpath:ehcache.xml

3. Annotate Entity



A screenshot of a computer code

AI-generated content may be incorrect.

|  |  |
| --- | --- |
| **Element** | **Purpose** |
| <config> | Root element for the cache configuration |
| <cache alias="com.example.model.User"> | Declares a cache named com.example.model.User |
| <heap unit="entries">100</heap> | Limits to 100 objects in JVM memory |
| <ttl unit="minutes">10</ttl> | Items expire 10 minutes after creation |

**Why apply @cachable annotation at class level**

The @Cacheable annotation at the class level (usually in Hibernate) is used to indicate that instances of the entity should be stored in the second-level cache.

**Cache Provider**

A cache provider is a library or tool that actually performs the caching logic—storing and retrieving data from storage to improve application performance.

**Redis configuration**

**1**. Add Redis Dependency

**2.** Configure Redis in application.properties

* spring.cache.type=redis
* spring.redis.host=localhost
* spring.redis.port=6379
* # Optional: Timeout and password
* # spring.redis.timeout=60000
* # spring.redis.password=your\_password

**3**. Enable Caching in Spring Boot

**4.** Use @Cacheable, @CacheEvict, etc. in Your Code

6. Run Redis Server

* Linux: sudo apt install redis && redis-server

A screenshot of a computer program

AI-generated content may be incorrect.

|  |  |  |
| --- | --- | --- |
| **Use Case** | **Use Ehcache** | **Use Redis** |
| Runs in | Embeded, Same JVM as application | Separate server (local or remote) |
| Cacche Type | In Memory,Second level | Second level |
| Single-node Spring Boot app | ✅ Best fit | ❌ Overkill |
| Hibernate second-level cache | ✅ Preferred | ⚠️ Possible, not native |
| Distributed microservices | ❌ Not ideal | ✅ Best fit |
| Need persistent cache (restart-safe) | ✅ Yes | ✅ Yes |
| Token storage, session store | ❌ No | ✅ Yes |

**How to Enable Web Cache**

|  |  |
| --- | --- |
| **What to Cache** | **How to Enable** |
| For Static files (JS, CSS) | spring.web.resources.cache.period=timeperiod |
| REST API responses | @Cacheable + Cache-Control header. Ex. ResponseEntity.ok()  .cacheControl(CacheControl.maxAge(60, timeunit)).body(body); |
| Global HTTP headers | Add filter (HttpServletResponse) |
| CDN / proxy compatibility | Set Cache-Control, ETag, Expires |

|  |  |  |
| --- | --- | --- |
| **Feature** | **Spring MVC** | **Spring Boot** |
| **Definition** | A module in the Spring Framework for building **web applications** using MVC architecture. | A **framework built on top of Spring** to simplify application setup and development with **auto-configuration** and **opinionated defaults**. |
| **Setup & Configuration** | Requires **manual configuration** (XML or Java-based), including servlet container setup. | Comes with **auto-configuration** and embedded servers like **Tomcat**, reducing boilerplate. |
| **Deployment** | Typically deployed as a WAR file to an **external server** (like Tomcat). | Deployed as a **standalone JAR** with an **embedded server**. |
| **Starter Dependencies** | Developers manually include dependencies for MVC, JPA, etc. | Uses **starter dependencies** like spring-boot-starter-web, spring-boot-starter-data-jpa, etc. |
| **Application Properties** | Configuration typically done via XML or Java Config. | Uses application.properties or application.yml for centralized and easy configuration. |
| **Focus** | Only web layer (controllers, view resolvers). | Full-stack support including data, web, messaging, security, etc., with **microservices** support. |
| **Ease of Development** | More complex and verbose to set up. | Faster and easier to develop applications with **production-ready defaults**. |
| **Actuator, DevTools** | Must be added and configured manually. | Built-in support via dependencies like spring-boot-starter-actuator. |

|  |  |  |
| --- | --- | --- |
| **Feature** | **Spring** | **Spring Boot** |
| **Primary Focus** | Comprehensive framework for enterprise Java | Simplified, rapid application development |
| **Configuration** | Extensive manual configuration (XML, Java-based) | Auto-configuration with sensible defaults |
| **Dependency Management** | Manual inclusion and management of dependencies | Starter dependencies for simplified dependency management |
| **Deployment** | Packaged as WAR, deployed to external servers | Standalone JAR with embedded servers |
| **Embedded Servers** | Not provided by default | Includes embedded servers like Tomcat, Jetty, Undertow |
| **Auto-Configuration** | Requires manual configuration | Automatic configuration based on included dependencies |
| **Development Tools** | Standard development and testing setup | Spring Boot DevTools for enhanced development experience |
| **Production-Ready Features** | Requires additional setup | Built-in features like health checks, metrics, externalized configuration |
| **Initial Setup Complexity** | Higher due to extensive configuration options | Lower due to auto-configuration and starters |
| **Microservices Support** | Can be used but requires more setup | Designed for microservices with lightweight, independent deployment |
| **Cloud-Native Applications** | Requires additional configuration | Easily deployable to cloud environments |
| **Application Bootstrap** | Manual setup for application context and main class | Simplified with @SpringBootApplication annotation |
| **Learning Curve** | Steeper due to extensive feature set | Gentler due to conventions over configuration |
| **Testing** | Requires manual context setup for integration tests | Simplified with @SpringBootTest annotation |
| **Customization** | High degree of customization | Customization possible but built on auto-configuration |
| **Documentation and Community** | Extensive documentation and large community | Extensive documentation with active community |
| **Integration with Legacy Systems** | Well-suited for integrating with legacy systems | Can integrate but more suited for new projects |
| **Use Cases** | Large enterprise applications, legacy systems | Microservices, rapid prototyping, cloud-based applications |
| **Example Dependency (Maven)** | xml <dependency> <groupId>org.springframework</groupId> <artifactId>spring-context</artifactId> </dependency> | xml <dependency> <groupId>org.springframework.boot</groupId> <artifactId>spring-boot-starter-web</artifactId> </dependency> |
| **Example Main Class** | java @Configuration @ComponentScan @EnableAutoConfiguration public class MyApp { public static void main(String[] args) { SpringApplication.run(MyApp.class, args); } } | java @SpringBootApplication public class MyApp { public static void main(String[] args) { SpringApplication.run(MyApp.class, args); } } |

**@ConfigurationProperties:** @ConfigurationProperties is an annotation in Spring Boot used to **bind external configuration (e.g., application.properties or YAML)** to a Java object.

Ex. In POJO class **@ConfigurationProperties(prefix = "app")**