**QUESTIONS**

**Spring Boot**

1. What is Spring Boot? Name advantages and disadvantages of Spring Boot.

## ****Spring Boot**** for application development

Despite the advantages of Spring Framework, authors decided to provide developers with some utilities that automate the configuration procedure and speed up the process of creating and deploying Spring applications. These utilities were combined under the general name of Spring Boot.

While the Spring Framework focuses on providing flexibility, Spring Boot seeks to reduce code length and simplify web application development. By leveraging annotation and boilerplate configuration, Spring Boot reduces the time it takes to develop applications. This capability helps you create standalone applications with less or almost no configuration overhead.

### **Ease of Dependency Management**

To speed up the dependency management process, Spring Boot implicitly packages the required third-party dependencies for each type of Spring-based application and provides them through so-called starter packages (spring-boot-starter-web, spring-boot-starter-data-jpa, etc.).

Starter packages are collections of handy dependency descriptors that you can include in your application. They allow you to get a universal solution for all Spring-related technologies, removing the necessity to search for code examples and load the required dependency descriptors from them.

For instance, if you want to start using Spring Data JPA to access your database, just include the spring-boot-starter-data-jpa dependency in your project and you’ll be done (no need to look for compatible Hibernate database drivers and libraries).

If you want to create a Spring web application, just add the spring-boot-starter-web dependency, which will pull all the libraries you need to develop Spring MVC applications into your project, such as spring-webmvc, jackson-json, validation-api, and Tomcat.

To say in a few words what is Spring Boot used for, it collects all common dependencies and defines them in one place, which allows developers to get to work right away instead of reinventing the wheel every time they create a new application.

Therefore, the pom.xml file contains much fewer lines when used in Spring Boot than in regular Spring.

### **Automatic Сonfiguration**

One of the advantages of Spring Boot, that is worth mentioning is automatic configuration of the application.

After choosing a suitable starter package, Spring Boot will try to automatically configure your Spring application based on the jar dependencies you added. For example, if you add Spring-boot-starter-web, Spring Boot will automatically configure registered beans such as DispatcherServlet, ResourceHandlers, and MessageSource.

If you are using spring-boot-starter-jdbc, Spring Boot automatically registers the DataSource, EntityManagerFactory, and TransactionManager beans and reads the database connection information from the application.properties file. If you are not going to use a database and do not provide any details about connecting manually, Spring Boot will automatically configure the database in the memory without any additional configuration on your part (if you have H2 or HSQL libraries). Automatic configuration can be completely overridden at any time by using user preferences.

### **Native Support for Application Server – Servlet Container**

Every Spring Boot application includes an embedded web server. Developers no longer have to worry about setting up a servlet container and deploying an application to it. The application can now run itself as an executable jar file using the built-in server. If you need to use a separate HTTP server, simply exclude the default dependencies. Spring Boot provides separate starter packages for different HTTP servers.

Building stand-alone web applications with embedded servers is not only convenient for development but also a valid solution for enterprise-grade applications; what’s more, it’s becoming increasingly useful in the world of microservices. The ability to quickly package an entire service (such as user authentication) into a standalone and fully deployable artefact that also provides an API makes installing and deploying an application much easier.

Spring Boot is part of the next generation of tools that simplify the configuration process for Spring applications. It is not a tool for automatic code generation, but a plugin for project build automation systems (supporting Maven and Gradle).

The plugin provides capabilities for testing and deploying Spring applications. The mvn spring-boot:run command runs your application on port 8080. In addition, Spring Boot allows you to package your application into a separate jar file with a full Tomcat container embedded. This approach was borrowed from the Play framework’s application deployment model (however, Spring Boot can also create traditional war files).

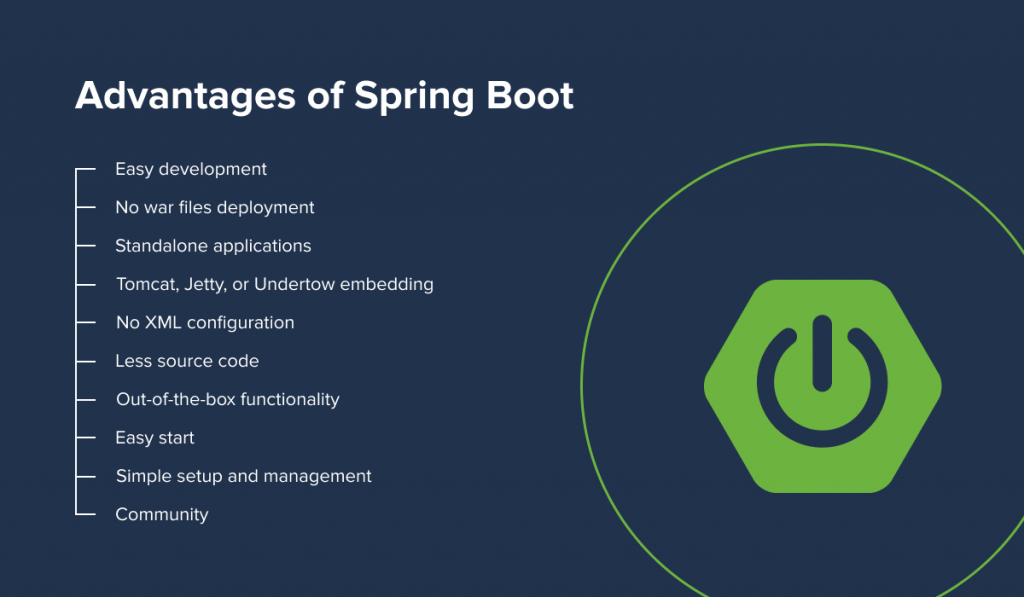
One of the key advantages of Spring Boot is the configuration of resources based on the content of the classpath. Spring Boot is pretty intuitive when it comes to the default configuration. You may not always agree with its choice of settings, but at least it will provide you with a working module. This is a very useful approach, especially for novice developers who can start with the default settings and make changes to them as they explore the alternatives later down the line. This is much better than answering a bunch of difficult questions every time you begin a new project. In addition, there are a number of full-fledged tutorials on Spring Boot’s official webpage. These will help you quickly understand and practically implement all the main types of projects at the initial level.

Spring Boot is still in its infancy, and it will naturally have to go through many metamorphoses before becoming fully stable. It may be too early to use it for building serious systems, but it is quite suitable for performing all sorts of personal, training, and test projects, in case you want to eliminate large amounts of unproductive, routine work that is in no way related to the creation of useful functionality.

As far as Spring Boot’s potential for growing into a serious development tool is concerned, the presence of acceptable technical documentation looks very encouraging.

Now let’s take a closer look at the pros and cons of using Spring Boot.

**Advantages of a Spring Boot application**



Spring Boot is designed to make the Spring Framework easier to use. Spring provides a loosely coupled application, which is a great feature by itself. However, when there are several loosely coupled blocks involved, keeping track of them all becomes a tedious and thankless task. This is where Spring Boot comes in, helping you keep things simple with the following features:

* Fast and easy development of Spring-based applications;
* No need for the deployment of war files;
* The ability to create standalone applications;
* Helping to directly embed Tomcat, Jetty, or Undertow into an application;
* No need for XML configuration;
* Reduced amounts of source code;
* Additional out-of-the-box functionality;
* Easy start;
* Simple setup and management;
* Large community and many training programs to facilitate the familiarization period.

With features like auto-configuration, Spring Boot saves you the hassle of coding and unnecessary configuration. The Spring Framework doesn’t just offer you features like dependency injection or transaction processing, it also serves as the foundation for other Spring frameworks. The best example of this is Spring Boot. Spring Boot uses the Spring Framework as its foundation. It simplifies Spring dependencies and runs applications directly from the command line. It also doesn’t require an external application container. Spring Boot helps control and customize application components externally.

**Disadvantages of Spring Boot**



In spite of many advantages of Spring Boot, it still has a couple of drawbacks that you should keep in mind:

* Lack of control. Spring Boot creates a lot of unused dependencies, resulting in a large deployment file;
* The complex and time-consuming process of converting a legacy or an existing Spring project to a Spring Boot application;
* Not suitable for large-scale projects. Although it’s great for working with microservices, many developers claim that Spring Boot is not suitable for building monolithic applications.

***Conclusion***

Spring Boot has become an integral part of the Java ecosystem, offering an efficient and scalable toolbox for building Spring applications with a microservices architecture. It speeds up the development and deployment processes by using intuitive default settings for unit and integration tests. What’s more, Spring Boot helps developers build robust applications with clear and secure configurations without spending a lot of time and effort on figuring out the intricacies of Spring. If you’re not sure about whether or not you should use this solution for your [Java project](https://bambooagile.eu/technologies/java), carefully review the pros and cons of using Spring Boot, its core features, and see how they align with your business goals. Alternatively, you can entrust a reliable software vendor with the development process. In that case, Bamboo Agile can become your trusted partner.

1. What Spring Boot features do you know?

***dzone.com/articles/top-5-spring-boot-features-java-developers-should***

**What Is Spring?**

**Simply put, the Spring framework provides comprehensive infrastructure support for developing Java applications**.

It's packed with some nice features like Dependency Injection, and out of the box modules like:

* Spring JDBC
* Spring MVC
* Spring Security
* Spring AOP
* Spring ORM
* Spring Test

These modules can drastically reduce the development time of an application.

For example, in the early days of Java web development, we needed to write a lot of boilerplate code to insert a record into a data source. By using the *JDBCTemplate* of the Spring JDBC module, we can reduce it to a few lines of code with only a few configurations.

**What Is Spring Boot?**

Spring Boot is basically an extension of the Spring framework, which eliminates the boilerplate configurations required for setting up a Spring application.

**It takes an opinionated view of the Spring platform, which paves the way for a faster and more efficient development ecosystem**.

Here are just a few of the features in Spring Boot:

* Opinionated ‘starter' dependencies to simplify the build and application configuration
* Embedded server to avoid complexity in application deployment
* Metrics, Health check, and externalized configuration
* Automatic config for Spring functionality – whenever possible

1. What are the differences between Spring Boot and Spring?

| 1. **Basis** | **Spring** | **Spring Boot** |
| --- | --- | --- |
| Where it’s used? | Spring framework is a java EE framework that is used to build applications. | Spring Boot framework is mainly used to develop REST API’s |
| Key feature | The primary or most important feature of the Spring framework is dependency injection(Dependency Injection (DI) is a design technique that removes dependencies from computer code, making the application easier to maintain and test). | The main or primary feature of the Spring Boot is Autoconfiguration( Simply described, Spring Boot autoconfiguration is a method of automatically configuring a Spring application based on the dependencies found on the classpath.) Autoconfiguration can speed up and simplify development by removing the need to define some beans that are part of the auto-configuration classes. |
| Why it’s used | Its goal is to make Java EE (Enterprise Edition) development easier, allowing developers to be more productive. | Spring Boot provides the RAD(Rapid Application Development) feature to the Spring framework for faster application development. |
| Type of Application Development | Spring framework helps to create a loosely coupled application. | Spring Boot helps to create a stand-alone application. |
| Servers dependency | In the Spring framework to test the Spring Project, we need to set up the servers explicitly. | Spring Boot offers built-in or embedded servers such as Tomcat and jetty. |
| Deployment descriptor | To run a Spring application a deployment descriptor is required. | In Spring Boot there is no need for the Deployment descriptor. |
| In-memory database support | Spring framework does not provide support for the in-memory database. | Spring Boot provides support for the in-memory database such as H2. |
| Boilerplate code | Spring framework requires too many lines of code (boilerplate code) even for minimal tasks. | You avoid boilerplate code which reduces time and increases productivity. |
| Configurations | In the Spring framework, you have to build configurations manually. | In Spring Boot there are default configurations that allow faster bootstrapping. |
| Dependencies | Spring Framework requires a number of dependencies to create a web app. | Spring Boot, on the other hand, can get an application working with just one dependency. There are several more dependencies required during build time that is added to the final archive by default. |
| HTTP Authentication | HTTP Basic Authentication is for enabling security confirmations, it indicates that several dependencies and configurations need to be enabled to enable security. Spring requires both the standard spring-security-web and spring-security-config dependencies to set up security in an application. Next, we need to add a class that extends the WebSecurityConfigurerAdapter and makes use of the @EnableWebSecurity annotation. | Spring Boot also requires these dependencies to make it work, but we only need to define the dependency of spring-boot-starter-security as this will automatically add all the relevant dependencies to the classpath. |
| Testing | Testing in Spring Boot is difficult in comparison to Spring Boot due to a large amount of source code. | Testing in Spring Boot is easier due to the reduced amount of source code. |
| XML Configuration | In the Spring framework, XML Configuration is required. | No need for XML configuration in Spring Boot. |
| CLI Tools | Spring framework does not provide any CLI tool for developing and testing applications. | Spring Boot provides a CLI tool for developing and testing Spring Boot applications. |
| Plugins | Spring framework does not provide any plugin for maven, Gradle, etc. like Spring Boot. | Spring Boot provides build tool plugins for Maven and Gradle. The Plugins offer a variety of features, including the packaging of executable jars. |

## Conclusion

By now, we have a good understanding of what Spring and SpringBoot are, even though they go hand in hand. To sum up Spring Boot contains all of the functionality of the standard Spring framework while also making application development much easier. When compared to Spring, you can get an application up and running in considerably less time because all Spring Boot attributes are auto-configured.

**Difference between Spring and Spring Boot :**

|  |  |  |
| --- | --- | --- |
| S.No. | Spring | Spring Boot |
| 1. | Spring is an open-source lightweight framework widely used to develop enterprise applications. | Spring Boot is built on top of the conventional spring framework, widely used to develop REST APIs. |
| 2. | The most important feature of the Spring Framework is dependency injection. | The most important feature of the Spring Boot is Autoconfiguration. |
| 3. | It helps to create a loosely coupled application. | It helps to create a stand-alone application. |
| 4. | To run the Spring application, we need to set the server explicitly. | Spring Boot provides embedded servers such as Tomcat and Jetty etc. |
| 5. | To run the Spring application, a deployment descriptor is required. | There is no requirement for a deployment descriptor. |
| 6. | To create a Spring application, the developers write lots of code. | It reduces the lines of code. |
| 7. | It doesn’t provide support for the in-memory database. | It provides support for the in-memory database such as H2. |

1. What are the ways to create Spring Boot application do you know?

[https://start.spring.io](https://start.spring.io/)

using IntelliJIDEA

1. What Spring Boot annotations do you know?

**Spring Boot Annotations** is a form of metadata that provides data about a program that is not a part of the program itself. They do not have any direct effect on the operation of the code they annotate. **Spring Boot Annotations** do not use XML and instead use the convention over configuration principle.

Given below are some important **Spring Boot Annotations.**

**Table of Contents**

* [Spring Boot Annotations Everyone Should Know](https://www.upgrad.com/blog/spring-boot-annotations/#Spring_Boot_Annotations_Everyone_Should_Know)
  + [1. @Bean](https://www.upgrad.com/blog/spring-boot-annotations/#1_Bean)
  + [2. @Service](https://www.upgrad.com/blog/spring-boot-annotations/#2_Service)
  + [3. @Repository](https://www.upgrad.com/blog/spring-boot-annotations/#3_Repository)
  + [4. @Configuration](https://www.upgrad.com/blog/spring-boot-annotations/#4_Configuration)
  + [5. @Controller](https://www.upgrad.com/blog/spring-boot-annotations/#5_Controller)
  + [6. @RequestMapping](https://www.upgrad.com/blog/spring-boot-annotations/#6_RequestMapping)
  + [7. @Autowired](https://www.upgrad.com/blog/spring-boot-annotations/#7_Autowired)
  + [8. @Component](https://www.upgrad.com/blog/spring-boot-annotations/#8_Component)
  + [9. @SpringBootApplication](https://www.upgrad.com/blog/spring-boot-annotations/#9_SpringBootApplication)
  + [10. @EnableAutoConfiguration](https://www.upgrad.com/blog/spring-boot-annotations/#10_EnableAutoConfiguration)
  + [11. @ComponetScan](https://www.upgrad.com/blog/spring-boot-annotations/#11_ComponetScan)
  + [12. @Required](https://www.upgrad.com/blog/spring-boot-annotations/#12_Required)
  + [13. @Qualifier](https://www.upgrad.com/blog/spring-boot-annotations/#13_Qualifier)
  + [14. @CookieValue](https://www.upgrad.com/blog/spring-boot-annotations/#14_CookieValue)
  + [15. @Lazy](https://www.upgrad.com/blog/spring-boot-annotations/#15_Lazy)

[Spring Boot](https://www.geeksforgeeks.org/introduction-to-spring-boot/) is built on the top of the spring and contains all the features of spring. And is becoming a favorite of developers these days because of its rapid production-ready environment which enables the developers to directly focus on the logic instead of struggling with the configuration and setup. Spring Boot is a microservice-based framework and making a production-ready application in it takes very little time. Following are some of the features of Spring Boot:

* It allows avoiding heavy configuration of XML which is present in spring
* It provides easy maintenance and creation of REST endpoints
* It includes embedded Tomcat-server
* Deployment is very easy, war and jar files can be easily deployed in the tomcat server

**Spring Annotations** are a form of metadata that provides data about a program. Annotations are used to provide supplemental information about a program. It does not have a direct effect on the operation of the code they annotate. It does not change the action of the compiled program. So in this article, we are going to discuss annotations that are available in Spring Boot with examples.

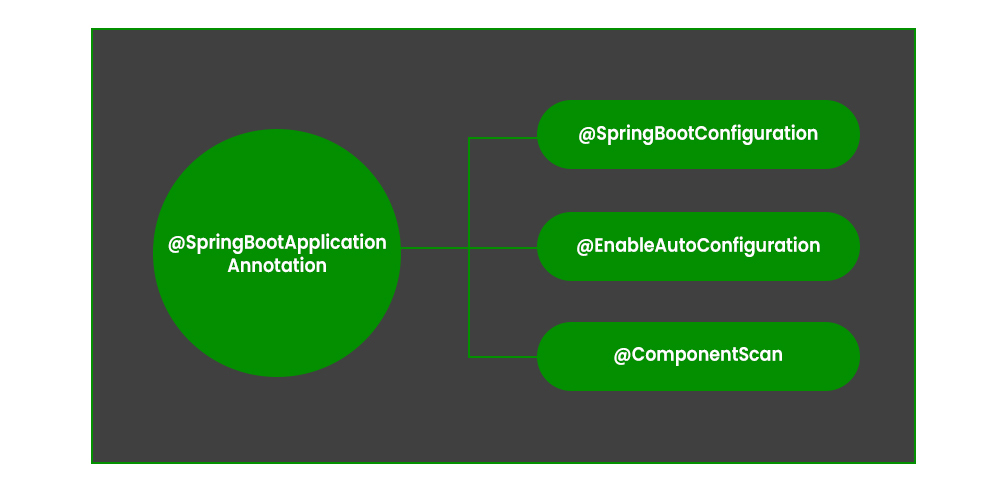
### [**Spring Boot Annotations**](https://www.geeksforgeeks.org/spring-framework-annotations/)

Spring annotations present in the ***org.springframework.boot.autoconfigure*** and ***org.springframework.boot.autoconfigure.condition***packages are commonly known as Spring Boot annotations. Some of the annotations that are available in this category are:

* @SpringBootApplication
* @SpringBootConfiguration
* @EnableAutoConfiguration
* @ComponentScan
* Auto-Configuration Conditions
  + @ConditionalOnClass, and @ConditionalOnMissingClass
  + @ConditionalOnBean, and @ConditionalOnMissingBean
  + @ConditionalOnProperty
  + @ConditionalOnResource
  + @ConditionalOnWebApplication and @ConditionalOnNotWebApplication
  + @ConditionalExpression
  + @Conditional

**1. @SpringBootApplication Annotation**

This annotation is used to mark the main class of a Spring Boot application. It encapsulates **@SpringBootConfiguration**, **@EnableAutoConfiguration**, and **@ComponentScan** annotations with their default attributes.



**Example:**

* Java

|  |
| --- |
| @SpringBootApplication    // Class  public class DemoApplication {        // Main driver method      public static void main(String[] args)      {            SpringApplication.run(DemoApplication.class, args);      }  } |

**2. @SpringBootConfiguration Annotation**

It is a class-level annotation that is part of the Spring Boot framework. It implies that a class provides Spring Boot application configuration. It can be used as an alternative to Spring’s standard **@Configuration** annotation so that configuration can be found automatically. Most Spring Boot Applications use @SpringBootConfiguration via @SpringBootApplication. If an application uses @SpringBootApplication, it is already using @SpringBootConfiguration.

**Example:**

* Java

|  |
| --- |
| @SpringBootConfiguration  public class Application {        public static void main(String[] args) {          SpringApplication.run(Application.class, args);      }        @Bean      public StudentService studentService() {          return new StudentServiceImpl();      }  } |

**3. @EnableAutoConfiguration Annotation**

This annotation auto-configures the beans that are present in the classpath. It simplifies the developer’s work by assuming the required beans from the classpath and configure it to run the application. This annotation is part of the spring boot framework. For example, when we illustrate the **spring-boot-starter-web** dependency in the classpath, Spring boot auto-configures [Tomcat](https://www.geeksforgeeks.org/embedding-tomcat-server-in-maven-project/), and [Spring MVC](https://www.geeksforgeeks.org/difference-between-spring-mvc-and-spring-boot/). The package of the class declaring the @EnableAutoConfiguration annotation is considered as the default. Therefore, we need to apply the @EnableAutoConfiguration annotation in the root package so that every sub-packages and class can be examined.

**Example:**

* Java

|  |
| --- |
| @Configuration  @EnableAutoConfiguration  public class Application {        public static void main(String[] args) {          SpringApplication.run(Application.class, args);      }    } |

**4. @ComponentScan Annotation**

@ComponentScan tells Spring in which packages you have annotated classes that should be managed by Spring. So, for example, if you have a class annotated with [@Controller](https://www.geeksforgeeks.org/spring-controller-annotation-with-example/) which is in a package that is not scanned by Spring, you will not be able to use it as a Spring controller. So we can say @ComponentScan enables Spring to scan for things like configurations, controllers, services, and other components that are defined. Generally, @ComponentScan annotation is used with @Configuration annotation to specify the package for Spring to scan for components.

**Example:**

* Java

|  |
| --- |
| @Configuration  @ComponentScan    // Main class  public class Application {        // Main driver method      public static void main(String[] args)      {            SpringApplication.run(Application.class, args);      }  } |

**5. @ConditionalOnClass Annotation and @ConditionalOnMissingClass Annotation**

@ConditionalOnClass Annotation used to mark auto-configuration bean if the class in the annotation’s argument is present/absent.

**Example:**

* Java

|  |
| --- |
| @Configuration  @ConditionalOnClass(MongoDBService.class)    class MongoDBConfiguration {      // Insert code here  } |

**6. @ConditionalOnBean Annotation and @ConditionalOnMissingBean Annotation**

These annotations are used to let a bean be included based on the presence or absence of specific beans.

**Example:**

* Java

|  |
| --- |
| @Bean  @ConditionalOnMissingBean(type = "JpaTransactionManager")    JpaTransactionManager jpaTransactionManager(      EntityManagerFactory entityManagerFactory)  {      // Insert code here  } |

**7. @ConditionalOnProperty Annotation**

These annotations are used to let configuration be included based on the presence and value of a Spring Environment property.

**Example:**

* Java

|  |
| --- |
| @Bean  @ConditionalOnProperty(name = "usemongodb",                         havingValue = "local")    DataSource  dataSource()  {      // Insert code here  }    @Bean  @ConditionalOnProperty(name = "usemongodb",                         havingValue = "prod")    DataSource  dataSource()  {      // Insert code here  } |

**8. @ConditionalOnResource Annotation**

These annotations are used to let configuration be included only when a specific resource is present in the classpath.

**Example:**

* Java

|  |
| --- |
| @ConditionalOnResource(resources                         = "classpath:mongodb.properties")    Properties  additionalProperties()  {      // Insert code here  } |

**9. @ConditionalOnExpression Annotation**

These annotations are used to let configuration be included based on the result of a**SpEL (Spring Expression Language) expression.**

***SpEL (Spring Expression Language):****It is a powerful expression language that supports querying and manipulating an object graph at runtime.*

**Example:**

* Java

|  |
| --- |
| @Bean  @ConditionalOnExpression("${env} && ${havingValue == 'local'}")    DataSource dataSource()  {      // Insert code here  } |

**10. @ConditionalOnCloudPlatform Annotation**

These annotations are used to let configuration be included when the specified cloud platform is active.

**Example:**

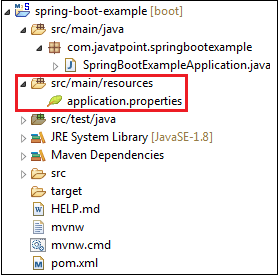
* Java

|  |
| --- |
| @Configuration  @ConditionalOnCloudPlatform(CloudPlatform.CLOUD\_FOUNDRY)    public class CloudConfigurationExample  {    // Insert code here  } |

1. What are the Spring Boot properties?

# Spring Boot Application Properties

Spring Boot Framework comes with a built-in mechanism for application configuration using a file called **application.properties**. It is located inside the **src/main/resources** folder, as shown in the following figure.



Spring Boot provides various properties that can be configured in the **application.properties**file. The properties have default values. We can set a property(s) for the Spring Boot application. Spring Boot also allows us to define our own property if required.

The application.properties file allows us to run an application in a **different environment.**In short, 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

In the above example, we have configured the **application name** and **port**. The port 8081 denotes that the application runs on port **8081**.

10.6M

76

Twitter To Pay $150 Million for Using Phone Numbers and Emails To Target Ads

**Next**

**Stay**

#### **Note: The lines started with # are comments.**

**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.

**Example of application.yml**

1. spring:
2. application:
3. name: demoApplication
4. server:
5. port: 8081

In the above example, we have configured the **application name** and **port**. The port 8081 denotes that the application runs on port **8081**.

## 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. For example, 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 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 |  | 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 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. |

1. What are the Spring Boot starters? Name some of them and their purpose.

Before Spring Boot was introduced, Spring Developers used to spend a lot of time on Dependency management. Spring Boot Starters were introduced to solve this problem so that the developers can spend more time on actual code than Dependencies. Spring Boot Starters are dependency descriptors that can be added under the **<dependencies>** section in pom.xml. There are around 50+ Spring Boot Starters for different Spring and related technologies. These starters give all the dependencies under a single name. For example, if you want to use Spring Data JPA for database access, you can include **spring-boot-starter-data-jpa**dependency.

The advantages of using Starters are as follows:

* Increase productivity by decreasing the Configuration time for developers.
* Managing the POM is easier since the number of dependencies to be added is decreased.
* Tested, Production-ready**,** and supported dependency configurations.
* No need to remember the name and version of the dependencies.

Spring Boot Starter Data JPA is illustrated below:

<dependencies>

<dependency>

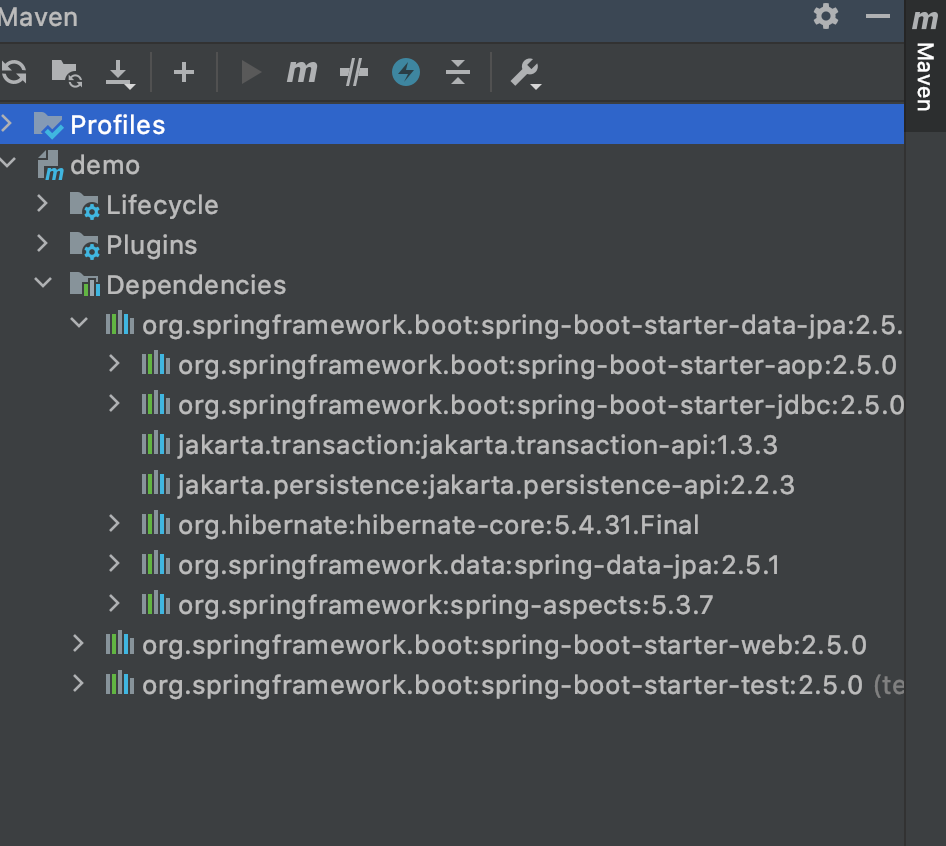
<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-data-jpa</artifactId>

</dependency>

</dependencies>

This gives all the required dependencies and can be seen under the Maven tab in IntelliJ IDEA.



In earlier days, developers used to include all those dependencies. Now Spring Boot Starters provides all those with just a single dependency. The official starters follow a naming convention **spring-boot-starter-\***, where **\***denotes application type. For example, if we want to build web including RESTful applications using Spring MVC we have to use **spring-boot-starter-web**dependency.

Third-Party Starters

If you want to make your own starter or other third-party starters name should not start with **spring-boot** as it reserved for official Spring Boot Starters. It can start with the name of the project. For example, if the name of the project is **gfg-code-template**, then the name of the starter can be **gfg-code-template-spring-boot-starter**.

Here out we will be discussing all 3 starters and the following starters are provided by the Spring Boot under **org.springframework.boot** group. They are namely and covered as follows:

1. Application Starters
2. Production Starters
3. Technical Starters

Let us elaborate these starters in the same sequential order which is as follows:

**(A)** **Spring Boot Application Starters**

| Name | Description |
| --- | --- |
| **spring-boot-starter** | Core starter, including auto-configuration support, logging, and YAML |
| **spring-boot-starter-activemq** | Starter for JMS messaging using Apache ActiveMQ |
| **spring-boot-starter-amqp** | Starter for using Spring AMQP and Rabbit MQ |
| **spring-boot-starter-aop** | Starter for aspect-oriented programming with Spring AOP and AspectJ |
| **spring-boot-starter-artemis** | Starter for JMS messaging using Apache Artemis |
| **spring-boot-starter-batch** | Starter for using Spring Batch |
| **spring-boot-starter-cache** | Starter for using Spring Framework’s caching support |
| **spring-boot-starter-data-cassandra** | Starter for using Cassandra distributed database and Spring Data Cassandra |
| **spring-boot-starter-data-cassandra-reactive** | Starter for using Cassandra distributed database and Spring Data Cassandra Reactive |
| **spring-boot-starter-data-couchbase** | Starter for using Couchbase document-oriented database and Spring Data Couchbase |
| **spring-boot-starter-data-couchbase-reactive** | Starter for using Couchbase document-oriented database and Spring Data Couchbase Reactive |
| **spring-boot-starter-data-elasticsearch** | Starter for using Elasticsearch search and analytics engine and Spring Data Elasticsearch |
| **spring-boot-starter-data-jdbc** | Starter for using Spring Data JDBC |
| **spring-boot-starter-data-jpa** | Starter for using Spring Data JPA with Hibernate |
| **spring-boot-starter-data-ldap** | Starter for using Spring Data LDAP |
| **spring-boot-starter-data-mongodb** | Starter for using MongoDB document-oriented database and Spring Data MongoDB |
| **spring-boot-starter-data-mongodb-reactive** | Starter for using MongoDB document-oriented database and Spring Data MongoDB Reactive |
| **spring-boot-starter-data-neo4j** | Starter for using Neo4j graph database and Spring Data Neo4j |
| **spring-boot-starter-data-r2dbc** | Starter for using Spring Data R2DBC |
| **spring-boot-starter-data-redis** | Starter for using Redis key-value data store with Spring Data Redis and the Lettuce client |
| **spring-boot-starter-data-redis-reactive** | Starter for using Redis key-value data store with Spring Data Redis reactive and the Lettuce client |
| **spring-boot-starter-data-rest** | Starter for exposing Spring Data repositories over REST using Spring Data REST |
| **spring-boot-starter-freemarker** | Starter for building MVC web applications using FreeMarker views |
| **spring-boot-starter-groovy-templates** | Starter for building MVC web applications using Groovy Templates views |
| **spring-boot-starter-hateoas** | Starter for building hypermedia-based RESTful web application with Spring MVC and Spring HATEOAS |
| **spring-boot-starter-integration** | Starter for using Spring Integration |
| **spring-boot-starter-jdbc** | Starter for using JDBC with the HikariCP connection pool |
| **spring-boot-starter-jersey** | Starter for building RESTful web applications using JAX-RS and Jersey. An alternative to spring-boot-starter-web |
| **spring-boot-starter-jooq** | Starter for using jOOQ to access SQL databases. An alternative to spring-boot-starter-data-jpa or spring-boot-starter-jdbc |
| **spring-boot-starter-json** | Starter for reading and writing json |
| **spring-boot-starter-jta-atomikos** | Starter for JTA transactions using Atomikos |
| **spring-boot-starter-mail** | Starter for using Java Mail and Spring Framework’s email sending support |
| **spring-boot-starter-mustache** | Starter for building web applications using Mustache views |
| **spring-boot-starter-oauth2-client** | Starter for using Spring Security’s OAuth2/OpenID Connect client features |
| **spring-boot-starter-oauth2-resource-server** | Starter for using Spring Security’s OAuth2 resource server features |
| **spring-boot-starter-quartz** | Starter for using the Quartz scheduler |
| **spring-boot-starter-rsocket** | Starter for building RSocket clients and servers |
| **spring-boot-starter-security** | Starter for using Spring Security |
| **spring-boot-starter-test** | Starter for testing Spring Boot applications with libraries including JUnit Jupiter, Hamcrest and Mockito |
| **spring-boot-starter-thymeleaf** | Starter for building MVC web applications using Thymeleaf views |
| **spring-boot-starter-validation** | Starter for using Java Bean Validation with Hibernate Validator |
| **spring-boot-starter-web** | Starter for building web, including RESTful, applications using Spring MVC. Uses Tomcat as the default embedded container. |
| **spring-boot-starter-web-services** | Starter for using Spring Web Services |
| **spring-boot-starter-webflux** | Starter for building WebFlux applications using Spring Framework’s Reactive Web support |
| **spring-boot-starter-websocket** | Starter for building WebSocket applications using Spring Framework’s WebSocket support |

**(B)** **Spring Boot Production Starters**

| Name | Description |
| --- | --- |
| **spring-boot-starter-actuator** | Starter for using Spring Boot’s Actuator which provides production-ready features to help you monitor and manage your application |

**(C)** **Spring Boot Technical Starters**

| Name | Description |
| --- | --- |
| **spring-boot-starter-jetty** | Starter for using Jetty as the embedded servlet container. An alternative to spring-boot-starter-tomcat |
| **spring-boot-starter-log4j2** | Starter for using Log4j2 for logging. An alternative to spring-boot-starter-logging |
| **spring-boot-starter-logging** | Starter for logging using Logback. Default logging starter |
| **spring-boot-starter-reactor-netty** | Starter for using Reactor Netty as the embedded reactive HTTP server. |
| **spring-boot-starter-tomcat** | Starter for using Tomcat as the embedded servlet container. Default servlet container starter used by spring-boot-starter-web |
| **spring-boot-starter-undertow** | Starter for using Undertow as the embedded servlet container. An alternative to spring-boot-starter-tomcat |

1. What is Spring Boot auto-configuration?

Spring Boot is heavily attracting developers toward it because of three main features as follows:

1. Auto-configuration – such as checking for the dependencies, the presence of certain classes in the classpath, the existence of a bean, or the activation of some property.
2. An opinionated approach to configuration.
3. The ability to create stand-alone applications.

### Auto-Configuration in Spring Boot

* @Conditional annotation acts as a base for the Spring Boot auto-configuration annotation extensions.
* It automatically registers the beans with @Component, @Configuration, @Bean, and meta-annotations for building custom stereotype annotations, etc.
* The annotation @EnableAutoConfiguration is used to enable the auto-configuration feature.
* The @EnableAutoConfiguration annotation enables the auto-configuration of Spring ApplicationContext by scanning the classpath components and registering the beans.
* This annotation is wrapped inside the @SpringBootApplication annotation along with @ComponentScan and @SpringBootConfiguration annotations.
* When running main() method, this annotation initiates auto-configuration.

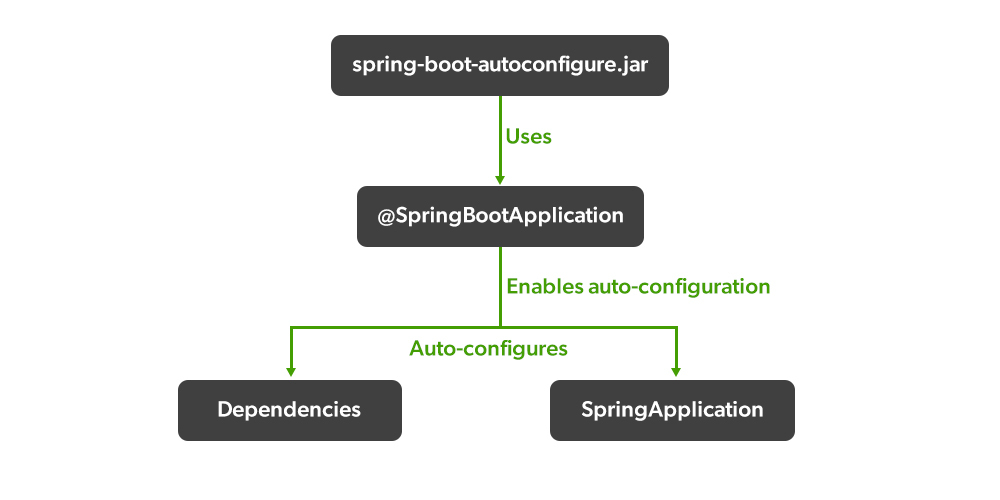
**Implementation:**Bootstrapping of Application

* Java

|  |
| --- |
| // Java Program to Illustrate Bootstrapping of Application    package gfg;    // Importing required classes  import org.springframework.boot.SpringApplication;  import org.springframework.boot.autoconfigure.SpringBootApplication;    // Annotation  @SpringBootApplication    // Class  public class GfgApplication {        // Main driver method      public static void main(String[] args)      {          SpringApplication.run(GfgApplication.class, args);      }  } |

***Note****: You should use the ‘@EnableAutoConfiguration’ annotation only one time in your application.*

* *‘spring-boot-autoconfigure.jar’ is the file that looks after all the auto-configuration.*
* *All auto-configuration logic for MVC, data, JMS, and other frameworks is present in a single jar*



### **Working of Auto-Configuration in Spring Boot**

**A: Dependencies**

* Auto-Configuration is the main focus of the Spring Boot development.
* Our Spring application needs a respective set of dependencies to work.
* Spring Boot auto-configures a pre-set of the required dependencies without a need to configure them manually.
* This greatly helps and can be seen when we want to create a stand-alone application.
* When we build our application, Spring Boot looks after our dependencies and configures both the underlying Spring Framework and required jar dependencies (third-party libraries ) on the classpath according to our project built.
* It helps us to avoid errors like mismatches or incompatible versions of different libraries.
* If you want to override these defaults, you can override them after initialization.

### **Understanding Auto-Configuration of Dependencies**

* When you build a Spring Boot project, the ‘Starter Parent’ dependency gets automatically added in the ‘pom.xml’ file.
* It notifies that the essential ‘sensible’ defaults for the application have been auto-configured and you therefore can take advantage of it.

<parent>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-parent</artifactId>

<version>...</version>

</parent>

* To add the dependency ( library of tech stacks ), you don’t need to mention the version of it because the Spring Boot automatically configures it for you.
* Also, when you update/change the Spring Boot version, all the versions of added dependencies will also get updated/changed.

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-thymeleaf</artifactId>

</dependency>

* It is Spring Boot’s auto-configuration that makes managing dependencies supremely easy for us.
* With the help of enabling ‘debug logging’ in the ‘application.properties’ file, we can know more about auto-configuration.

logging.level.org.springframework: DEBUG

### Tool B: Gradle

**Example 2: build.gradle**

buildscript {

repositories {

jcenter()

}

dependencies {

classpath("org.springframework.boot:spring-boot-gradle-plugin:1.3.8.RELEASE")

}

}

apply plugin: 'java'

apply plugin: 'spring-boot'

repositories {

jcenter()

}

dependencies {

compile("org.springframework.boot:spring-boot-starter-web")

testCompile("org.springframework.boot:spring-boot-starter-test")

}

**B:** Spring Application

**Illustration:** Class

1. @Bean is a method-level annotation.
2. @Bean annotation specifies that a method produces a return value registered as a bean ( data ) with BeanFactory – managed by Spring Container.
3. This particular java program uses @Configuration annotation specifying that the class contains one or more @Bean annotations which help to automatically register (initialize) in the Spring Container (Spring Application Context).
4. @Configuration is a class-level annotation.

**Example**

* Java

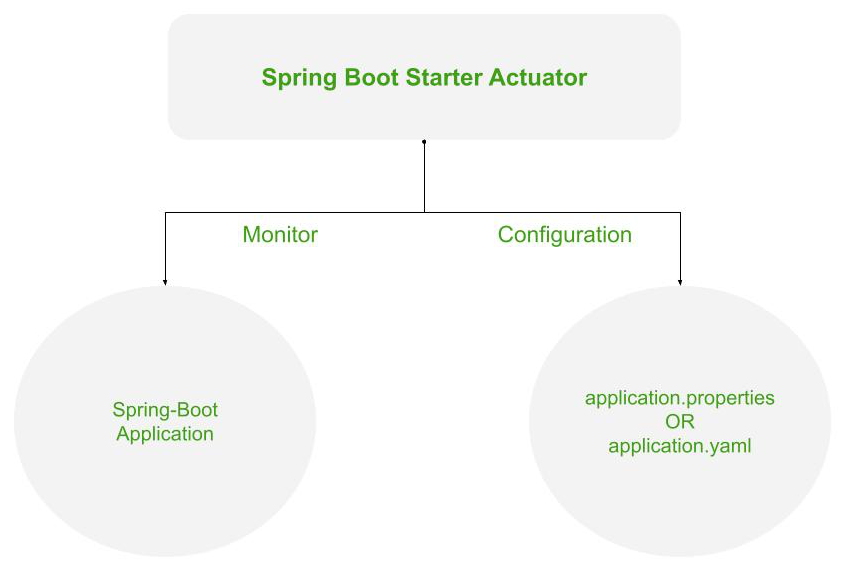
|  |
| --- |
| // Java Program Illustrating Configuration of  // DataSourceConfiguration of DataSource    package gfg;    // Importing required classes  import javax.sql.DataSource;  import org.springframework.boot.jdbc.DataSourceBuilder;  import org.springframework.context.annotation.Bean;  import org.springframework.context.annotation.Configuration;    // Annotation  @Configuration    // Class  public class ConfigDataSource {        // Annotation      @Bean public static DataSource source()      {            DataSourceBuilder<?> dSB              = DataSourceBuilder.create();          dSB.driverClassName("com.mysql.jdbc.Driver");            // MySQL specific url with database name          dSB.url("jdbc:<mysql://localhost:3306/userdetails>");            // MySQL username credential          dSB.username("user");            // MySQL password credential          dSB.password("password");            // builds and returns a new          // configured datasource object          return dSB.build();      }  } |

***Note:****Java Spring Boot framework’s auto configuration feature enables you to start developing your Spring-based applications fast and reduces the possibility of human errors.*

1. What is Spring Boot Actuator used for?

### **Spring Boot – Actuator**

* With the help of Spring Boot, we can achieve the above objectives.
* Spring Boot’s ‘Actuator’ dependency is used to monitor and manage the Spring web application.
* We can use it to monitor and manage the application with the help of HTTP endpoints or with the JMX.



*Working of the Spring’s Actuator*

To use the ‘Actuator’ add the following dependency in your application’s project build.

***Maven* -> pom.xml**

<dependencies>

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-actuator</artifactId>

</dependency>

</dependencies>

***Gradle* -> build.gradle**

dependencies {

implementation 'org.springframework.boot:spring-boot-starter-actuator'

}

## Spring Boot Actuator

**Spring Boot Actuator** is a sub-project of the Spring Boot Framework. It includes a number of additional features that help us to monitor and manage the Spring Boot application. It contains the actuator endpoints (the place where the resources live). We can use **HTTP** and **JMX** endpoints to manage and monitor the Spring Boot application. If we want to get production-ready features in an application, we should use the S**pring Boot actuator.**

### **Spring Boot Actuator Features**

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. Spring Boot provides a number of built-in endpoints. We can also create our own endpoint. We can enable and disable each endpoint individually. Most of the application choose **HTTP**, where the Id of the endpoint, along with the prefix of **/actuator,**is mapped to a URL.

For example, the **/health** endpoint provides the basic health information of an application. The actuator, by default, mapped it to **/actuator/health**.

0

**Metrics:** Spring Boot Actuator provides dimensional metrics by integrating with the**micrometer**. The micrometer is integrated into Spring Boot. It is the instrumentation library powering the delivery of application metrics from Spring. It provides vendor-neutral interfaces for **timers, gauges, counters, distribution summaries,** and **long task timers** with a dimensional data model.

**Audit:** Spring Boot provides a flexible audit framework that publishes events to an **AuditEventRepository.** It automatically publishes the authentication events if spring-security is in execution.

## Enabling Spring Boot Actuator

We can enable actuator by injecting the dependency **spring-boot-starter-actuator** in the pom.xml file.

1. **<dependency>**
2. **<groupId>**org.springframework.boot**</groupId>**
3. **<artifactId>**spring-boot-starter-actuator**</artifactId>**
4. **<version>**2.2.2.RELEASE**</version>**
5. **</dependency>**

## Spring Boot Actuator Endpoints

The actuator endpoints allow us to monitor and interact with our Spring Boot application. Spring Boot includes number of built-in endpoints and we can also add custom endpoints in Spring Boot application.

The following table describes the widely used endpoints.

|  |  |  |
| --- | --- | --- |
| **Id** | **Usage** | **Default** |
| 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 |

For Spring MVC, the following additional endpoints are used.

|  |  |  |
| --- | --- | --- |
| **Id** | **Description** | **Default** |
| docs | It is used to display documentation, including example requests and responses for the Actuator's endpoints. | False |
| heapdump | It is used to return a GZip compressed hprof heap dump file. | True |
| jolokia | It is used to expose JMX beans over HTTP (when Jolokia is on the classpath). | True |
| logfile | It is used to return the contents of the logfile. | True |
| prometheus | It is used to expose metrics in a format that can be scraped by a prometheus server. It requires a dependency on micrometer-registry- prometheus. | True |

## Spring Boot actuator properties

Spring Boot enables security for all actuator endpoints. It uses **form-based** authentication that provides **user Id** as the user and a randomly generated **password**. We can also access actuator-restricted endpoints by customizing basicauth security to the endpoints. We need to override this configuration by **management.security.roles** property. For example:

1. management.security.enabled=true
2. management.security.roles=ADMIN
3. security.basic.enabled=true
4. security.user.name=admin
5. security.user.passowrd=admin

### **Spring Boot Actuator Example**

Let's understand the concept of the actuator through an example.

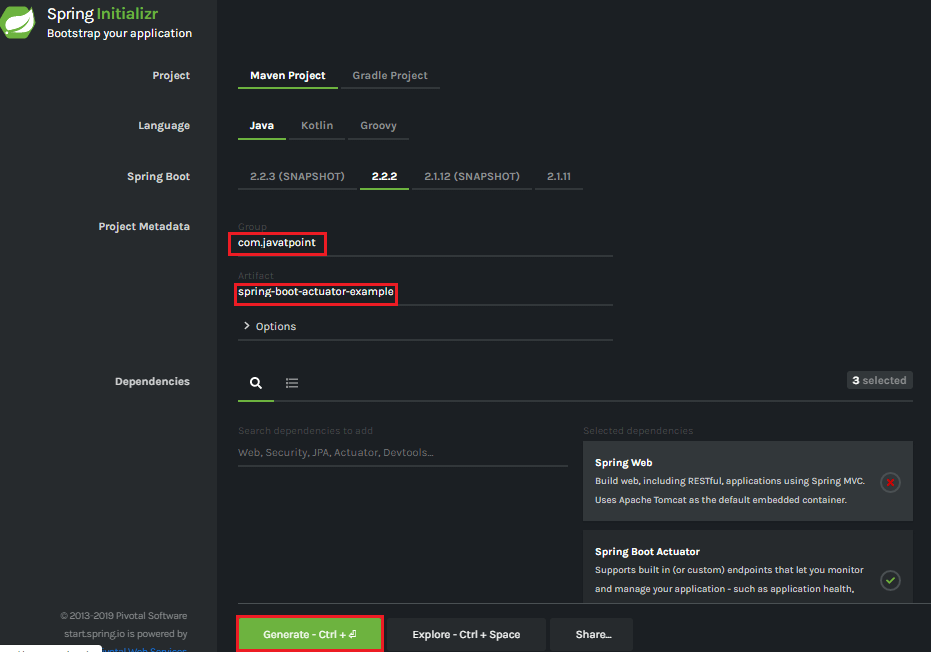
**Step 1:** Open Spring Initializr <https://start.spring.io/> and create a **Maven** project.

**Step 2:** Provide the **Group** name. We have provided **com.javatpoint.**

**Step 3:** Provide the **Artifact** Id. We have provided the **spring-boot-actuator-example.**

**Step 4:** Add the following dependencies: **Spring Web, Spring Boot Starter Actuator,** and **Spring Data Rest HAL Browser**.

**Step 5:** Click on the **Generate** button. When we click on the Generate button, it wraps all the specifications related to the project into a **Jar** file and downloads it to our local system.

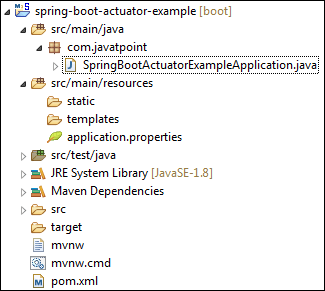


**Step 6:** Extract the Jar file and paste it into the STS workspace.

**Step 7:** Import the project folder.

File -> Import -> Existing Maven Projects -> Browse -> Select the folder spring-boot-actuator-example -> Finish

It takes some time to import. After importing the project, we can see the project directory in the package explorer section.



**Step 8:** Create a Controller class. We have created the controller class with the name DemoRestController.

**DemoRestController.java**

1. package com.javatpoint;
2. import org.springframework.web.bind.annotation.GetMapping;
3. import org.springframework.web.bind.annotation.RestController;
4. @RestController
5. public class DemoRestController
6. {
7. @GetMapping("/hello")
8. public String hello()
9. {
10. return "Hello User!";
11. }
12. }

**Step 9:** Open the **application.properties** file and disable the security feature of the actuator by adding the following statement.

**application.properties**

1. management.security.enabled=false

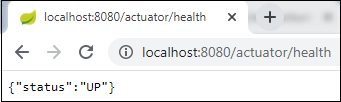
**Step 10:** Run the **SpringBootActuatorExampleApplication.java** file.

**Step 11:** Open the browser and invoke the URL http://localhost:8080/actuator. It returns the following page:

1. {"\_links":{"self":{"href":"http://localhost:8080/actuator","templated":false},"health":{"href":"http://localhost:8080/actuator/health","templated":false},"health-path":{"href":"http://localhost:8080/actuator/health/{\*path}","templated":true},"info":{"href":"http://localhost:8080/actuator/info","templated":false}}}

The application runs on port 8080 by default. Once the actuator has started, we can see the list of all the endpoints exposed over HTTP.

Let's invoke the **health** endpoint by invoking the URL http://localhost:8080/actuator/health. It denotes the status **UP**. It means the application is healthy and running without any interruption.



Similarly, we can invoke other endpoints that helps us to monitor and manage the Spring Boot application.

1. Describe how to build and run spring boot application: a. What's included in spring boot starter? b. How to run spring boot application from command line? c. Where do the properties place when building WAR? d. Where is the servlet container in Spring Boot and in normal Spring application

# Spring Properties File Outside jar

Last modified: May 28, 2022

by [baeldung](https://www.baeldung.com/author/baeldung/)

* [**Spring**](https://www.baeldung.com/category/spring/)**+**

### Get started with Spring 5 and Spring Boot 2, through the reference Learn Spring course:

[**>> LEARN SPRING**](https://www.baeldung.com/ls-course-start)

## 1. Overview

Property files are a common method that we can use to store project-specific information. Ideally, we should keep it external to the packaging to be able to make changes to the configuration as needed.

In this quick tutorial, we'll look into various **ways to load the properties file from a location outside the jar in a**[**Spring Boot application**](https://www.baeldung.com/properties-with-spring)**.**

## 2. Using the Default Location

By convention, Spring Boot looks for an externalized configuration file — *application.properties*or application.yml — in four predetermined locations in the following order of precedence:

* A /config subdirectory of the current directory
* The current directory
* A classpath /config package
* The classpath root

Therefore, **a property defined in application.properties and placed in the /config subdirectory of the current directory will be loaded.** This will also override properties in other locations in case of a collision.

## 3. Using the Command Line

If the above convention doesn't work for us, we can **configure the location directly in the command line**:

java -jar app.jar --spring.config.location=file:///Users/home/config/jdbc.properties

We can also pass a folder location where the application will search for the file:

java -jar app.jar --spring.config.name=application,jdbc --spring.config.location=file:///Users/home/config

Finally, an alternative approach is running the Spring Boot application through the [Maven plugin](https://www.baeldung.com/spring-boot-command-line-arguments).

There, we can use a -D parameter:

mvn spring-boot:run -Dspring.config.location="file:///Users/home/jdbc.properties"

## 4. Using Environment Variables

Now let's say that we can't alter the startup command.

What's great is that **Spring Boot will also read the environment variables SPRING\_CONFIG\_NAME and SPRING\_CONFIG\_LOCATION**:

export SPRING\_CONFIG\_NAME=application,jdbc

export SPRING\_CONFIG\_LOCATION=file:///Users/home/config

java -jar app.jar

Note that the default file will still be loaded. But **the environment-specific property file takes precedence**in case of a property collision.

## 5. Using Application Properties

As we can see, we have to define spring.config.name and spring.config.location properties before the application starts, so using them in the application.properties file (or the YAML counterpart) will have no effect.

Spring Boot modified how properties are handled in version 2.4.0.

Together with this change, the team introduced a new property that allows importing additional configuration files directly from the application properties:

spring.config.import=file:./additional.properties,optional:file:/Users/home/config/jdbc.properties

## 6. Programmatically

If we want programmatic access, we can register a PropertySourcesPlaceholderConfigurer bean:

**public** PropertySourcesPlaceholderConfigurer **propertySourcesPlaceholderConfigurer**() {

**PropertySourcesPlaceholderConfigurer** properties =

**new** **PropertySourcesPlaceholderConfigurer**();

properties.setLocation(**new** **FileSystemResource**("/Users/home/conf.properties"));

properties.setIgnoreResourceNotFound(false);

**return** properties;

}

Here we've used PropertySourcesPlaceholderConfigurer to load the properties from a custom location.

## ****7. Excluding a File From the Fat Jar****

The Maven Boot plugin will automatically include all files in the src/main/resources directory to the jar package.

If we don't want a file to be part of the jar, we can use a simple configuration to exclude it:

<**build**>

<**resources**>

<**resource**>

<**directory**>src/main/resources</**directory**>

<**filtering**>true</**filtering**>

<**excludes**>

<**exclude**>\*\*/conf.properties</**exclude**>

</**excludes**>

</**resource**>

</**resources**>

</**build**>

In this example, we've filtered out the conf.properties file from being included in the resulting jar.

## 8. Conclusion

This article showed how the Spring Boot framework itself takes care of [externalized configuration](https://www.baeldung.com/configuration-properties-in-spring-boot) for us.

Often, we just have to place the property values in the correct files and locations. But we can also use Spring's Java API for more control.

As always, the full source code of the examples is available [over on GitHub](https://github.com/eugenp/tutorials/tree/master/spring-boot-modules/spring-boot-environment).

*DispatcherServlet* receives all of the HTTP requests and delegates them to controller classes.

**Before the Servlet 3.x specification, *DispatcherServlet* would be registered in the *web.xml* file for a Spring MVC application.** Since the Servlet 3.x specification, we can register servlets programmatically using *ServletContainerInitializer*.

Let's see a *DispatcherServlet*example configuration in the*web.xml*file:

<**servlet**>

<**servlet-name**>dispatcher</**servlet-name**>

<**servlet-class**>

org.springframework.web.servlet.DispatcherServlet

</**servlet-class**>

</**servlet**>

<**servlet-mapping**>

<**servlet-name**>dispatcher</**servlet-name**>

<**url-pattern**>/</**url-pattern**>

</**servlet-mapping**>

Spring Boot provides the *spring-boot-starter-web* library for developing web applications using Spring MVC. One of the main features of Spring Boot is autoconfiguration. **The Spring Boot autoconfiguration registers and configures the *DispatcherServlet* automatically**. Therefore, we don’t need to register the *DispatcherServlet* manually.

By default, the *spring-boot-starter-web* starter configures *DispatcherServlet* to the URL pattern “/”. So, we don't need to complete any additional configuration for the above *DispatcherServlet*example in the*web.xml*file. However, we can customize the URL pattern using *server.servlet.*\* in the *application.properties* file:

server.servlet.context-path=/demo

spring.mvc.servlet.path=/baeldung

With these customizations, *DispatcherServlet* is configured to handle the URL pattern */baeldung* and the root *contextPath* will be */demo*. Thus, *DispatcherServlet* listens at *http://localhost:8080/demo/baeldung/.*

**REST**

1. Why we should support filtering, sorting, and pagination in RESTful APIs?
2. As consumer expectations ramp up, API performance has never been more important than it is today. It’s a well-known statistic that [53% of web users](https://www.thinkwithgoogle.com/data/mobile-site-load-time-statistics/) will abandon a web page if it takes more than 3 seconds to load.
3. These expectations don’t necessarily line up with the technical requirements of an API. In the era of big data and analytics, APIs are dealing with larger amounts of data in their backend than ever before. To truly stand their ground in today’s digital economy, APIs must be [optimized for peak efficiency](https://nordicapis.com/optimizing-the-api-response-package/). API pagination is a key strategy for making sure your APIs run smoothly and effectively.
4. But what is API pagination? How can API pagination help your APIs function at peak performance? We’re going to tell you, in our complete guide to API pagination.

# Everything You Need To Know About API Pagination

1. To make sure we’re on the same page, let’s start by looking at what pagination is. Then we’ll delve deeper into API pagination with example code implementations.

## What Is Pagination?

1. Have you ever clicked through an image gallery? Or read through an extensive web tutorial broken up into multiple segments? Do you know the numbers on the bottom of the gallery or webpage?
2. That’s pagination.
3. Sitechecker.pro, a technical SEO website, [defines pagination](https://sitechecker.pro/pagination/) as “an ordinal numbering of pages, which is usually located at the top or bottom of the site pages.” API pagination just applies that principle to the realm of API design.
4. API queries to dense databases could potentially return millions, if not billions, of results. There’s no telling what kind of drain that could put on your API. Pagination thus helps to limit the number of results to help keep network traffic in check.
5. Let’s look at some of the most common API pagination methods before we look at coding examples.
6. What are filtering, sorting, and pagination best practices can you name?

Web applications often have tables of data, whether it's the list of items for sale on Amazon, or notes in Evernote, and so on. Usually, users of the application are going to want to filter the results or sort through that data in some way.

If the dataset is pretty small, maybe a few hundred results, the API can return all the data at once and the front end will handle all the filtering, and no more API calls are required. Most of the time, however, the data could consist of tens of thousands to millions of rows, and it's better to just get the data you need from smaller API calls as opposed to trying to request a million results every time the page loads.

Recently, I made a backend API for some list endpoints, and implemented **filtering, sorting, and pagination**. There's not really a set standard for creating these types of endpoints, and almost every one I've come across is different in some way. I made a few notes on what made sense to me, so this resource could be helpful for someone who is working on designing an API.

#### GOALS

In this article I'll make an example **API endpoint** and **SQL query** for various sort, paginate, and filter APIs, with users as the table for all examples.

#### CONTENTS

* [**Response**](https://www.taniarascia.com/rest-api-sorting-filtering-pagination/#response)
* [**Pagination**](https://www.taniarascia.com/rest-api-sorting-filtering-pagination/#pagination)
* [**Sorting**](https://www.taniarascia.com/rest-api-sorting-filtering-pagination/#sorting)
  + [**Ascending vs. Descending**](https://www.taniarascia.com/rest-api-sorting-filtering-pagination/#ascending-vs-descending)
  + [**Single column**](https://www.taniarascia.com/rest-api-sorting-filtering-pagination/#single-column)
  + [**Multiple columns**](https://www.taniarascia.com/rest-api-sorting-filtering-pagination/#multiple-columns)
* [**Filtering**](https://www.taniarascia.com/rest-api-sorting-filtering-pagination/#filtering)
  + [**String (exact)**](https://www.taniarascia.com/rest-api-sorting-filtering-pagination/#string-exact)
  + [**String (exact, multiple)**](https://www.taniarascia.com/rest-api-sorting-filtering-pagination/#string-exact-multiple)
  + [**String (partial)**](https://www.taniarascia.com/rest-api-sorting-filtering-pagination/#string-partial)
  + [**Number (exact)**](https://www.taniarascia.com/rest-api-sorting-filtering-pagination/#number-exact)
  + [**Number (greater than)**](https://www.taniarascia.com/rest-api-sorting-filtering-pagination/#number-greater-than)
  + [**Number (less than)**](https://www.taniarascia.com/rest-api-sorting-filtering-pagination/#number-less-than)
  + [**Number (range)**](https://www.taniarascia.com/rest-api-sorting-filtering-pagination/#number-range)
  + [**Date (range)**](https://www.taniarascia.com/rest-api-sorting-filtering-pagination/#date-range)

## Response

When using any pagination or filtering in an API, you're going to want to know how many results you have, how many results there are total, and what page you're on.

API Response

{

content: [], // all the response items will go in this array

page: 1, // current page

results\_per\_page: 5, // how many items available in "content"

total\_results: 100 // total number of items

}

From there, you can discern that there are 20 pages with total\_results / results\_per\_page and anything else you might need for the front end.

## Pagination

Pagination is how you move between the pages when you don't want to retrieve all the results at once.

* **Page** and **results per page** are required inputs
* For the SQL query, offset is equal to (page - 1) \* results\_per\_page

GET /users?page=3&results\_per\_page=20

SELECT \* FROM users

LIMIT 20

OFFSET 40

## Sorting

Sorting allows you to order the results by any field, in ascending or descending order.

### **Ascending vs. Descending**

I always forget what ascending and descending mean for alphabetical, numerical, and date-based responses, so I wrote this up for reference.

| **Type** | **Order** | **Example** | **Description** |
| --- | --- | --- | --- |
| Alphabetical | Ascending | A - Z | First to last |
| Alphabetical | Descending | Z - A | Last to first |
| Numerical | Ascending | 1 - 9 | Lowest to highest |
| Numerical | Descending | 9 - 1 | Highest to lowest |
| Date | Ascending | 01-01-1970 - Today | Oldest to newest |
| Date | Descending | Today - 01-01-1970 | Newest to oldest |

### **Single column**

If you only need to sort one column at a time, you could put the column name in sort\_by and the sort direction in order.

GET /users?sort\_by=first\_name&order=asc

SELECT \* FROM users

ORDER BY first\_name ASC

### **Multiple columns**

If the ability to sort multiple columns is required, you could comma-separate each column:order pair and put it in one sort parameter. This could also be used for a single column if you prefer the syntax.

GET /users?sort=first\_name:asc,age:desc

SELECT \* FROM users

ORDER BY first\_name ASC, age DESC

## Filtering

Filtering is by far the most complex of the three. There are several ways to handle it. Some APIs will use a POST and pass all the data in the body of the request for searching. This might be necessary for advanced searching in some situations, but a GET is preferable.

Some API will attempt to put everything on a single filter parameter, like this:

GET users?filter={"first\_name":["Tania","Joe"],"age":[30,31,32]}

However, this will have to be URI encoded.

I've opted for treating each parameter as a column in the database.

### **String (exact)**

Exact search by a single column.

GET /users?first\_name=Tania

SELECT \* FROM users

WHERE first\_name = 'Tania'

### **String (exact, multiple)**

Depending on how you want to handle the API, multiple options for a single column can be handled [**in different ways**](https://stackoverflow.com/questions/24059773/correct-way-to-pass-multiple-values-for-same-parameter-name-in-get-request). If splitting by comma isn't an issue, it might be the easiest. You might also just want to repeat the parameter name or use a custom delimiter.

GET /users?first\_name=Tania,Joe

GET /users?first\_name=Tania&first\_name=Joe

GET /users?first\_name[]=Tania&first\_name[]=Joe

SELECT \* FROM users

WHERE first\_name IN ('Tania', 'Joe')

Some systems might require using [] for multiple parameters of the same name, and some might now allow [], so I provided both options.

### **String (partial)**

Often, searches are expected to be partial, so that when I look for "Tan" it will show me "Tania" and "Tanner". The solution I liked was using like:Tan as value as opposed to modifying the parameter (such as first\_name[like]=Tan).

GET /users?first\_name=like:Tan

SELECT \* FROM users

WHERE first\_name LIKE '%Tan%'

### **Number (exact)**

Exact number search on a column.

GET /users?age=30

SELECT \* FROM users

WHERE age = 31

### **Number (greater than)**

Similar to like:, you can use gt: to handle greater than. Adding the option for gte: (greater than or equal) is also an option.

GET /users?age=gt:21

SELECT \* FROM users

WHERE age > 21

### **Number (less than)**

Same with lt: for less than lte: for less than or equal.

GET /users?age=lt:21

SELECT \* FROM users

WHERE age < 21

### **Number (range)**

If you need a range between two number values, using [and] in between them could be one option. This one could get complicated, depending on if you want to allow both greater than and greater than or equal, or other options.

GET /users?age=gt:12[and]lt:20

SELECT \* FROM users

WHERE age > 12 AND age < 20

### **Date (range)**

If you need a range between two dates, you can use start and end, or since and to.

GET /users?start=01-01-1970&end=09-09-2020

SELECT \* FROM users

WHERE created\_at BETWEEN '01-01-1970' AND '09-09-2020'

## Conclusion

These examples are pretty simple and cover basic use cases. If your API is very complicated, you might need to change it up to add more options, particularly with ranges, and various combinations of "and" and "or". Hopefully this will be a helpful starting point!

1. What is Richardson Maturity Model? Name the Levels of the Model and explain them.

The **Richardson Maturity Model** (**RMM**) is a [maturity model](https://en.wikipedia.org/wiki/Maturity_model) suggested in 2008 by Leonard Richardson which classifies [Web APIs](https://en.wikipedia.org/wiki/Web_API) based on their adherence and conformity to each of the model's four levels. The aim of the research of the model as stated by the author was to find out the relationship between the constraints of REST and other forms of web services.[[1]](https://en.wikipedia.org/wiki/Richardson_Maturity_Model#cite_note-1)

It divides the principle parts of [RESTful design](https://en.wikipedia.org/wiki/Representational_state_transfer) into three steps: resource identification ([URI](https://en.wikipedia.org/wiki/URI)), [HTTP verbs](https://en.wikipedia.org/wiki/HTTP_Verbs), and [hypermedia controls](https://en.wikipedia.org/wiki/HATEOAS) (e.g. [hyperlinks](https://en.wikipedia.org/wiki/HTML_link)).[[2]](https://en.wikipedia.org/wiki/Richardson_Maturity_Model#cite_note-2)

The RMM has been cited useful in evaluating the quality of particularly RESTful Web API design (even though it is not restricted to REST alone) and criticized for not addressing how a system could achieve the highest maturity levels of the model as well as for considering a limited number of quality attributes.[[3]](https://en.wikipedia.org/wiki/Richardson_Maturity_Model#cite_note-:0-3)



## Contents

* [1Overview](https://en.wikipedia.org/wiki/Richardson_Maturity_Model#Overview)
* [2Structure](https://en.wikipedia.org/wiki/Richardson_Maturity_Model#Structure)
  + [2.1Levels](https://en.wikipedia.org/wiki/Richardson_Maturity_Model#Levels)
    - [2.1.1Level 0 : The Swamp of POX](https://en.wikipedia.org/wiki/Richardson_Maturity_Model#Level_0_:_The_Swamp_of_POX)
    - [2.1.2Level 1 : Resources](https://en.wikipedia.org/wiki/Richardson_Maturity_Model#Level_1_:_Resources)
    - [2.1.3Level 2 : HTTP verbs](https://en.wikipedia.org/wiki/Richardson_Maturity_Model#Level_2_:_HTTP_verbs)
    - [2.1.4Level 3 : Hypermedia controls](https://en.wikipedia.org/wiki/Richardson_Maturity_Model#Level_3_:_Hypermedia_controls)
* [3References](https://en.wikipedia.org/wiki/Richardson_Maturity_Model#References)

## Overview[[edit](https://en.wikipedia.org/w/index.php?title=Richardson_Maturity_Model&action=edit&section=1)]

The RMM can be employed to determine how well a Web service architecture adheres to REST principles. It categorizes a Web API into four levels (from 0 to 3) with each higher level corresponding to a more complete adherence to REST design. The next level also contains all the characteristics of the previous one.[[4]](https://en.wikipedia.org/wiki/Richardson_Maturity_Model#cite_note-4)[[5]](https://en.wikipedia.org/wiki/Richardson_Maturity_Model#cite_note-5)

Other classification systems for Web API services design also exist, such as CoHA and WS3.[[3]](https://en.wikipedia.org/wiki/Richardson_Maturity_Model#cite_note-:0-3)

## Structure[[edit](https://en.wikipedia.org/w/index.php?title=Richardson_Maturity_Model&action=edit&section=2" \o "Edit section: Structure)]

### Levels**[[edit](https://en.wikipedia.org/w/index.php?title=Richardson_Maturity_Model&action=edit&section=3" \o "Edit section: Levels)]**

#### Level 0 : The Swamp of POX**[**[**edit**](https://en.wikipedia.org/w/index.php?title=Richardson_Maturity_Model&action=edit&section=4)**]**

The lowest level of the model describes a Web API with a single URI (typically POST over HTTP) accepting all the range of operations supported by the service. Resources in this form cannot be well-defined. Messaging is done in XML, JSON, or other text formats. These are typical [RPC POX](https://en.wikipedia.org/wiki/Plain_Old_XML) and many [SOAP](https://en.wikipedia.org/wiki/SOAP) services.

Level Zero systems don't classify as [RESTful](https://en.wikipedia.org/wiki/Restful).

|  |  |  |
| --- | --- | --- |
| **Level 0 API Examples** | | |
| **URI** | **HTTP Verb** | **Operations** |
| /bookingService | POST | retrieve destinations/hotels/rooms; add/cancel a reservation; etc. |
| /newsFeedService | POST | get all news; get all news in category specified; get all news of an outlet specified; etc. |

#### Level 1 : Resources**[**[**edit**](https://en.wikipedia.org/w/index.php?title=Richardson_Maturity_Model&action=edit&section=5)**]**

Introduces resources and allows to make requests to individual URIs (still all typically POST) for separate actions instead of exposing one universal endpoint (API). The API resources are still generalized but it is possible to identify the scope of each one.

Level One design is not RESTful, yet it is organizing the API in the direction of becoming one.

|  |  |  |
| --- | --- | --- |
| **Level 1 API Examples** | | |
| **URI** | **HTTP Verb** | **Operation** |
| /bookingDestinations | POST | retrieve destinations |
| /bookingReservations | POST | add/cancel reservations |
| /bookingRooms | POST | add/cancel special requests to a reservation |
| /bookingFeedback | POST | leave feedback |

#### Level 2 : HTTP verbs**[**[**edit**](https://en.wikipedia.org/w/index.php?title=Richardson_Maturity_Model&action=edit&section=6)**]**

The system starts making use of [HTTP Verbs](https://en.wikipedia.org/wiki/HTTP_Verbs). This allows to further specialize the resource and thus narrow down the functionality of each individual operation with the service. The principle separation at this level consists in splitting a given resource into two - one request for obtaining data only (GET), the other for modifying the data (POST). Further granularization is also possible. GET requests only fetch data, POST/PUT calls introduce new and update existing data, DELETE requests remove or otherwise invalidate previous actions. One drawback of providing a distributed service with more than GET and POST per resource might be growing complication of using such a system.

|  |  |  |
| --- | --- | --- |
| **Level 2 API Examples** | | |
| **URI** | **HTTP Verb** | **Operation** |
| /destinations | GET | retrieve destinations |
| /reservations | GET | get reservations according to certain criteria |
| /reservations | POST | add/cancel reservations |
| /rooms | POST | request room extras |
| /rooms | DELETE | remove room extras |

#### Level 3 : Hypermedia controls**[**[**edit**](https://en.wikipedia.org/w/index.php?title=Richardson_Maturity_Model&action=edit&section=7)**]**

The last level introduces the hypermedia representation. Also called HATEOAS (Hypermedia As The Engine of Application State), these are elements embedded in the response messages of resources which allow to establish a relation between individual data entities returned from and pass to the APIs. For instance, a GET request to a hotel reservation system might return a number of available rooms along with hypermedia links (these would be html hyperlink controls in the early days of the model) allowing to book specific rooms.

This is the last level of the Richardson Maturity Model.

Request:

GET /room/?customerId=1&date=10-11-2020&hotelCode=ASTORIA HTTP/1.1

Response:

{

customerId: "1",

reservations: [{room: "102", checkin: "10-11-2020", checkout: "11-14-2020", price: "100", href: "https://localhost:8080/room/102"}]

}

Leonard Richardson developed the Richardson Maturity Model to grade APIs based on their adherence to the[REST constraints](https://www.geeksforgeeks.org/rest-api-architectural-constraints/). APIs with a high REST compliance score is considered to perform better.

In determining the maturity of a service, Richardson emphasized three main factors. They include:

* URI
* HTTP Methods
* HATEOAS (Hypermedia)

**URI:**A Uniform Resource Identifier (URI) is a unique sequence of characters used by web technologies to identify resources on the web.

[**HTTP METHODS**](https://www.geeksforgeeks.org/html-form-method-attribute/)**:**Hypertext Transfer Protocol (HTTP) is a protocol used to transfer hypermedia documents. HTTP requests are sent to servers by HTTP clients in the form of request messages. HTTP defines a set of request methods to specify the action to be taken on a given resource.

* **GET:** The GET method retrieves a representation of the specified resource.
* **POST:** A POST request transmits data to the server.
* **PUT:** The PUT method replaces all existing representations of the resource.
* **PATCH:** A PATCH request makes partial changes to a resource.
* **DELETE:** The DELETE method removes the specified resource.

HATEOAS (Hypermedia as the Engine of Application State ) refers to discoverability. The client can interact with a REST API solely through the server’s responses. It is a self-documentary Hypermedia. Clients need not refer to any documentation to interact with a new API.

**REST services are divided into maturity levels according to the Richardson Maturity Model.**

* Level 0
* Level 1
* Level 2
* Level 3

**LEVEL 0: POX swamp**

Level 0 is also often referred to as POX (Plain Old XML). At level 0, HTTP is used only as a transport protocol. For zero maturity level services, we use a single URL and a single HTTP method. We send a request to the same URI for obtaining and posting the data. Only the POST method can be used. for example, A particular company can have a lot of customers or users. We have only one endpoint for all the customers. All operations are performed via the POST method.

* To get the data: POST http://localhost:8080/users
* To post the data: POST http://localhost:8080/users

**LEVEL 1: Multiple URI based resource and single verb**

In level 1 , each resource is mapped to a specific URI. However, only one HTTP method (POST) is used for retrieving and creating data. for example, we need to access the employees working in a company.

* To add an employee to a particular department:  
  POST/department/<department-id>/employee
* To access a specific employee :  
  POST/department/<department-id>/employee/<employee-id>

**LEVEL 2: Multiple URI based resource and HTTP verbs**

At Level 2 requests are sent with the correct HTTP verb. A correct HTTP response code is returned for each request.

For example: To get the users of the company, we send a request with the URI  
http://localhost:8080/users and the server sends proper response 200 OK.

**LEVEL 3: HATEOS**

Level 3 is the highest. It combines level 2 and HATEOS. It is helpful in self-documentation. HATEOS guides where new resources can be found. Imagine a Chinese restaurant as an analogy. You ordered noodles, the waiter brings you the desired dish, explains what you just ordered and where you can find the other available dishes. Thus, we can consider the desired dish to be JSON data, while the rest dishes are hypermedia.

We consider an API to be RESTful when it reaches level 4. The other levels are only stepping stones to becoming one. Let’s make a RESTFUL API following Richardson’s Maturity Model

**Approach:**We will create a RESTFUL API called gfg-wiki. We will insert articles and send HTTP requests. In this process, we will fetch, modify and delete articles. Robo3T will be used for the database. Postman will be used to send requests.

In order to create a RESTFUL API in **node.js**, install :

**node:** A javaScript runtime environment

* Download link: https://nodejs.org/en/download/

**Robo3t:** A MongoDB GUI. we will create a database using robo3t.

* Download link:  https://robomongo.org/

**Postman:** An API development and testing platform.

* Download link: https://www.postman.com/

**Visual Studio Code or (any code editor)**

* Download link: https://code.visualstudio.com/download

**JSON** viewer pro Chrome extension

* Download link:  https://chrome.google.com/webstore/detail/json-viewer-pro/eifflpmocdbdmepbjaopkkhbfmdg

1. What is HATEOAS? Why do we need it?

HATEOAS stands for **Hypermedia as the Engine of Application State**and it is a component of RESTful API architecture and design. With the use of HATEOAS, the client-side needs minimal knowledge about how to interact with a server. This is made possible by the network application responding to the client’s requests with dynamically generated information through the use of hypermedia.



When accessing a webpage through a browser, users have the ability to interact with the webpage by using buttons, inputs, clicking on links, etc. However, traditional API responses have no such functionality present to allow an application to interact with the server through the response. HATEOAS acts as a way to address this. A HATEOAS request allows you to not only send the data but also specify the related actions.

### Application State Change Through HATEOAS

When using HATEOAS architecture, a client will be able to access the API for a network application through a simple, static, RESTful URL call. Now, any further actions, that the client may wish to take, will be enabled by the data, returned by the server, in the original call. This will enable the client to move from one application state to the next just by interacting with the details contained in the responses by the server.

The “data”, within the response, that enables this change of state is simple hypermedia links. This is how HATEOAS manages the change in application states through hypermedia.

### Demonstration of the Use of HATEOAS

As an example, consider that a client wants to interact with a network application to fetch details of employees’ payroll within an organization. The RESTful call to enable this would be as follows:

GET /payroll/employee\_123 HTTP/1.1

The server will respond with a JSON containing the required details. Additionally, the response will contain hypermedia links that allow the client to take further actions. As an example, consider the response by the server is as follows.

HTTP/1.1 200 OK

Content-Type: application/+json

Content-Length: ...

{

"payroll": {

"employee\_number": "employee\_123",

"salary" : 1000,

"links": {

"increment": "/payroll/employee\_123/increment",

"decrement": "/payroll/employee\_123/decrement",

"close": "/payroll/employee\_123/close"

}

}

}

We can observe that, in addition to the expected information being received in the response, additional information is presented in the form of RESTful hypermedia calls under the “links” title. These links allow further interaction with the server by incrementing or decrementing the salary or closing the account. It may be noted that these links correspond to the respective API endpoints to increment, decrement, and close the payroll account. Also, these links are pre-populated with the employee identifier. This means that such content is dynamically generated.

An additional example of how these hypermedia links are dynamically generated can be demonstrated is as follows:

Assume that for a given employee, the account has been closed. Thus, the increment and decrement methods are irrelevant to such an account. Thus, hitting the payroll endpoint for such an employee would result in response as follows:

HTTP/1.1 200 OK

Content-Type: application/+json

Content-Length: ...

{

"payroll": {

"employee\_number": "employee\_123"

"links": {

"start": "/payroll/employee\_123/start"

}

}

}

In this case, the links have changed to include functions that are relevant to the current state only. Thus, the only action made available is to “start” the payroll for such an employee.

### Need for HATEOAS

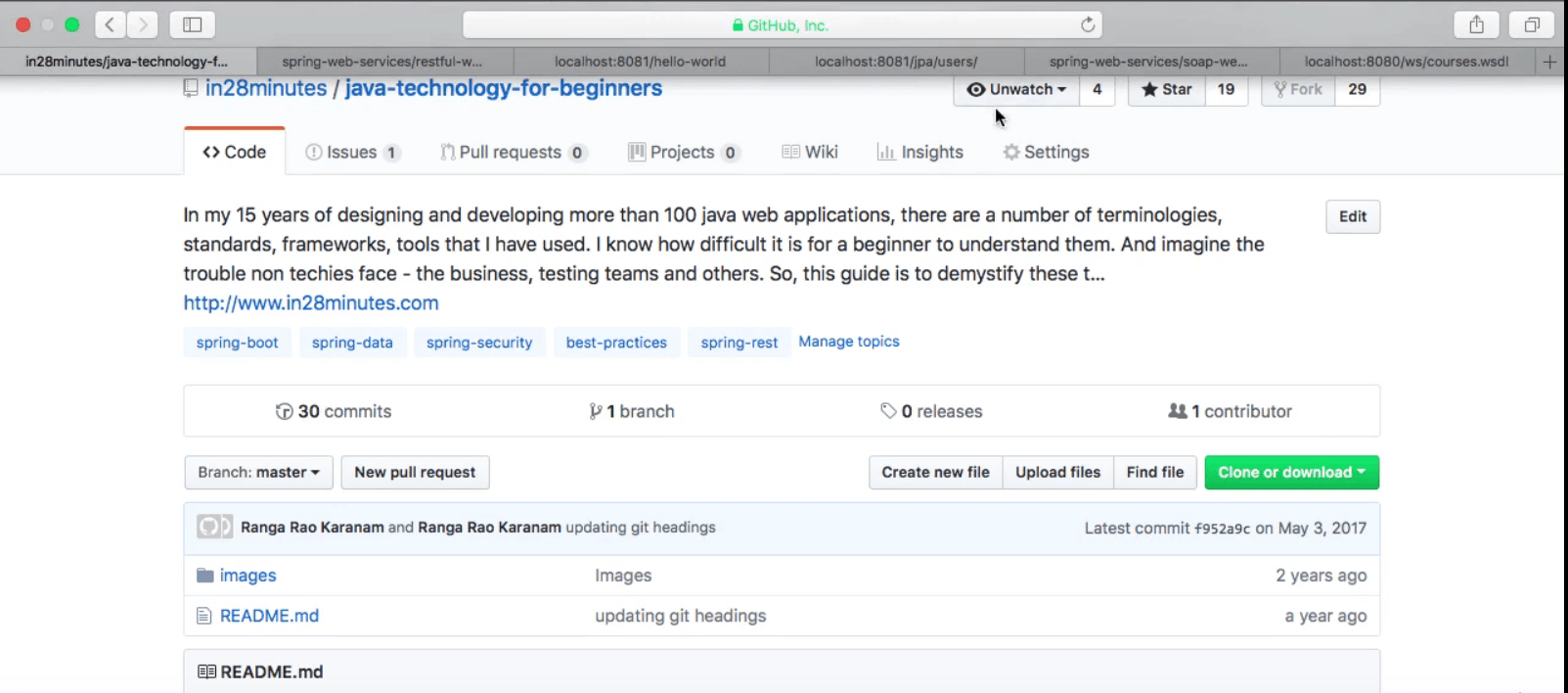
With traditional, non-HATEOAS based API systems, the API endpoints need to be hard-coded within the client-side application. Any changes to this endpoint would result in the client application systems breaking down. Thus, such changes would need to be updated in each of the client’s applications. This system is, therefore, tightly coupled.

With HATEOAS, the system becomes loosely coupled as the URLs do not require hard-coding. Instead, the URLs are generated on the fly on the server-side and supplied to the client through the JSON responses. Clients can now use these URLs from the response and be sure that these URLs are the latest versions.

## What Does HATEOAS Stand For?

The term **HATEOAS** stands for the phrase **H**ypermedia **A**s **T**he **E**ngine **O**f **A**pplication **S**tate. To understand this further, we first need to understand the meaning of **Hypermedia**.

Have a look at the following web page:

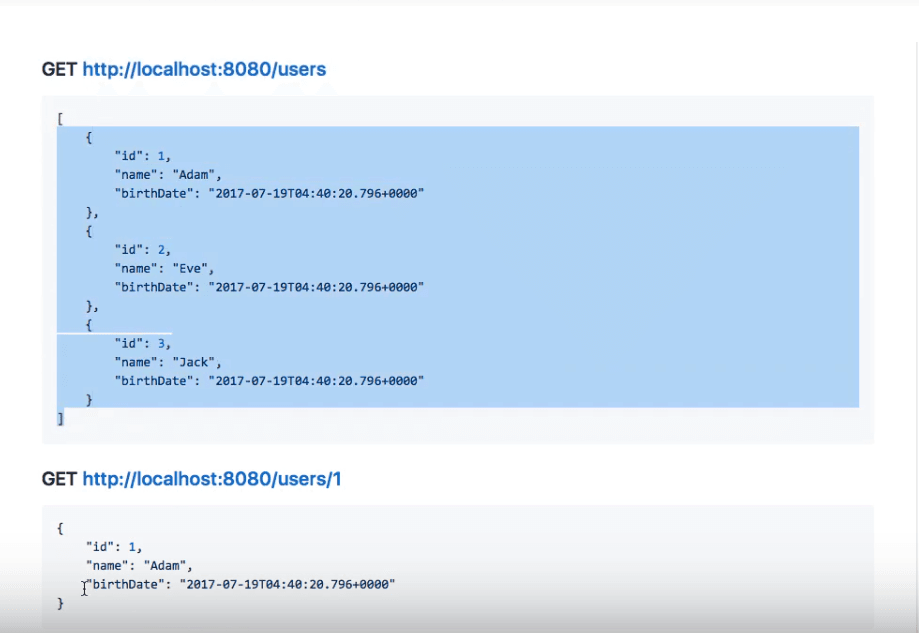


When the browser loads the page, you definitely can see all the content that the page has to offer. More interestingly, the page also allows you to perform a lot of actions around that data, such as:

* Clicking on buttons (the green "Clone or Download")
* Clicking on tabs (to view the "Issues", for instance)
* and several more

Now let's look at how our REST API's behave:

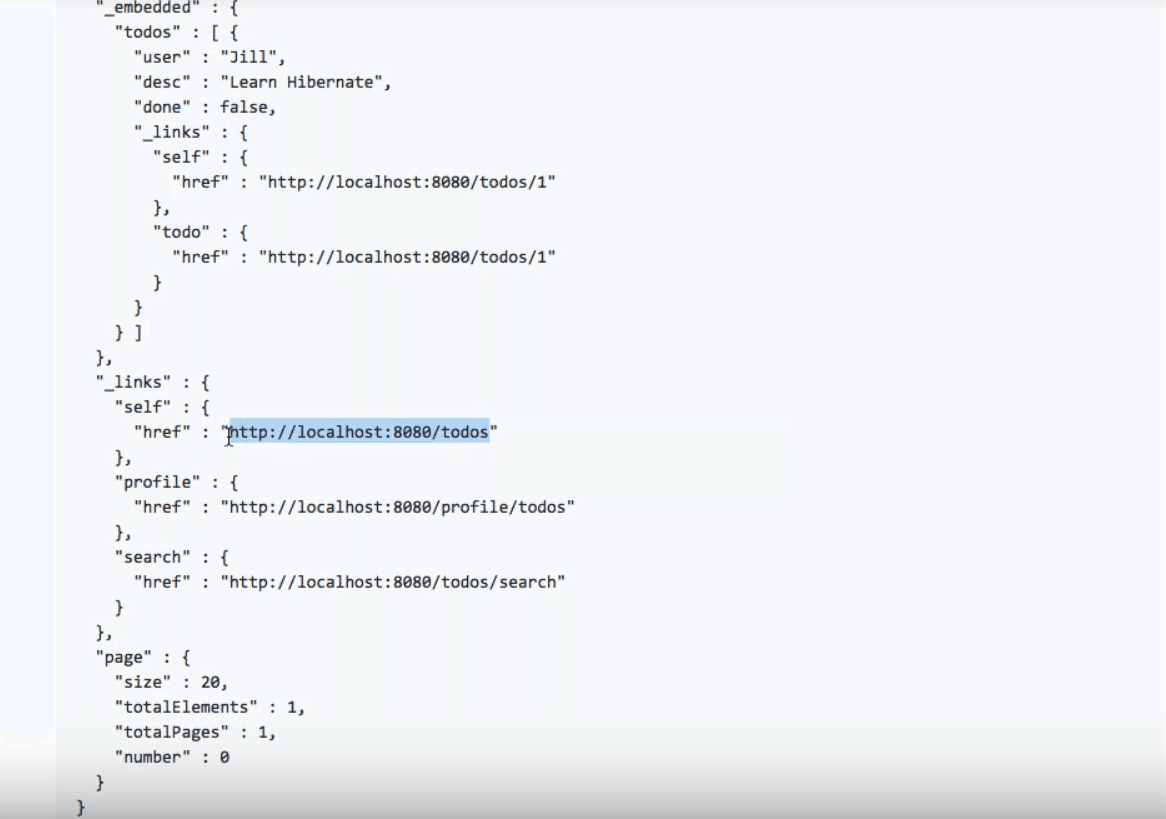
If you look at a typical GET request to a RESTful server, such as this one:



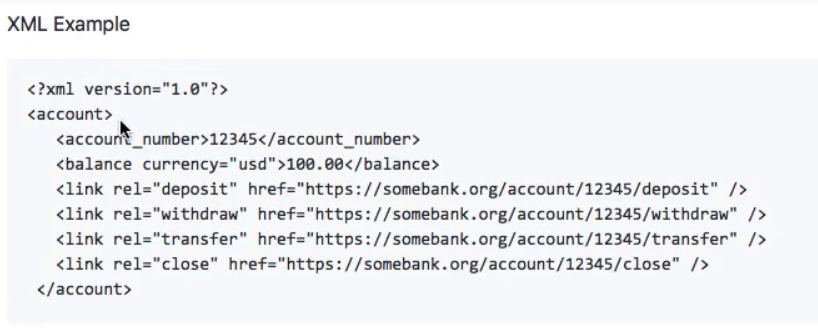
The request to GET http://localhost:8080/users retrieves a set of three users' details in this case. Sending a request with GET http://localhost:8080/users/1 will retrieve details of just the first user.

Typically, when we perform a REST request, we only get the data and not any actions around it. This is where HATEOAS comes in the fill in the gap.

A HATEOAS request allows you to not only send the data but also specify the related actions:



This example was in the JSON format. An XML format for another example would look something like this:



When you send out this request to retrieve account details, you get both:

* Account number and balance details
* Links that provide actions to do a deposit/withdrawal/transfer/closure

With HATEOAS, a request for a REST resource gives me both data, and actions related to the data.

## Why Do We Need HATEOAS?

The single most important reason for HATEOAS is **loose coupling**. If a consumer of a REST service needs to hard-code all the resource URLs, then it is tightly coupled with your service implementation. Instead, if you return the URLs, it could use for the actions, then it is loosely coupled. There is no tight dependency on the URI structure, as it is specified and used from the response.

A few significant topics related to HATEOAS are:

### HAL — Hypertext Application Language

When you design a RESTful service, there is a need to specify how to return data and links corresponding to a request. HAL is a simple format that gives an easy, consistent way to hyperlink between resources in your REST API. Here is an example:



With HAL, you have a few categories of representations:

* **Links:** Specified as a combination of
  + Target — Given as a URI
  + Relation — A name
* **Embedded Resources:** Other resources contained within a given REST resource
* **State:** The actual resource data

If you happen to use the Spring Framework to develop your REST service, then Spring HATEOAS is a good engine to use for your service.

1. What are the basic principles and alternatives for Rest Assured?

REST-assured was designed to simplify the testing and validation of REST APIs and is highly influenced by testing techniques used in dynamic languages such as Ruby and Groovy.

The library has solid support for HTTP, starting of course with the verbs and standard HTTP operations, but also going well beyond these basics.

In this guide, we are going to **explore REST-assured** and we're going to use Hamcrest to do assertion. If you are not already familiar with Hamcrest, you should first brush up with the tutorial: [Testing with Hamcrest](https://www.baeldung.com/java-junit-hamcrest-guide).

Rest Assured is used to verify the REST APIs with the help of the Java library. Java library acts like a headless client to act upon the Rest web services. The libraries based on the Rest Assured library are also capable of validating the HTTP responses from the server.

Response status code, body, message, headers, and so on can be tested with the Rest Assured library. It can be integrated with build tools like Maven, unit test frameworks like JUnit and TestNG. It has an efficient matching mechanism with which we can verify the expected results.

Application Programming Interface or API acts as an interface between a couple of applications and establishes a connection. It is an assembly of agreements, functions, and tools that an application presents to the users to interact with another application. We would need an API while we navigate and work on any application.

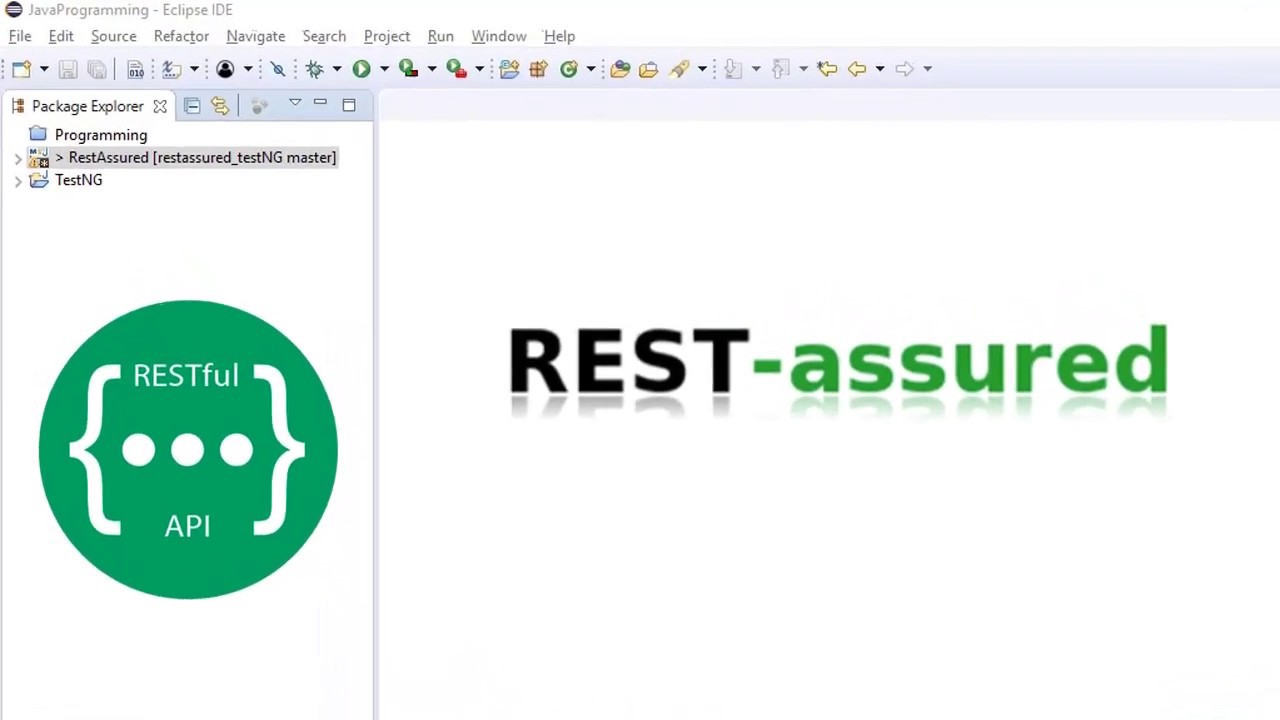
Let us know discuss why Rest Assured extensively for testing purposes.

While surfing an application, for example, Facebook, we are provided with numerous features like adding or searching for friends, creating a post, and so on. If we want to test all these features before the development of the web interface, we would need to take the help of the API for these features.

We can verify these APIs with multiple combinations of data with the help of automation. Rest Assured can be a good choice as a tool for API testing. It comes without licensing cost and has to be proved to be a successful tool for backend automation.

1. What type of testing is done with Rest Assured?

With samplerequests and tests for ***GET, POST , PUT, PATCH, DELETE***



Rest assured is java library for testing Restful Web services. It can be used to test XML & JSON based web services. It supports GET, POST, PUT, PATCH, DELETE, OPTIONS and HEAD requests and can be used to validate and verify the response of these requests. Also it can be integrated with testing frameworks like JUnit, TestNG etc.

Prerequisites : Java, IDE (Eclipse, IntelliJ, etc), Maven & TestNG

Steps to create project for API Testing :First create a maven project & add dependencies in pom.xml. Then create your test script, verify & run it.

**JPA**

1. What is the Java Persistence API?

JPA is basically an abstraction, using ORM techniques. If you map various model classes to the database, then JPA can a) generate an appropriate SQL query/update, b) convert the resultsets to the model classes. JPA also includes caching, and abstracts transaction handling.

In the end it doesn't really do any thing magical - everything ends up going through your JDBC driver, becoming raw SQL and returning JDBC resultsets and such. It merely allows you to hide a lot of that code away and just work with your model classes as Plain Old Java Objects (POJOs) where setting a property triggers a UPDATE and getting a property triggers a SELECT (the caching of everything and organization into transactions allows far better performance than you would get through a simple one-to-one implementation.

1. What is the object-relational mapping?

While they differ in execution, every JPA implementation provides some kind of ORM layer. In order to understand JPA and JPA-compatible tools, you need to have a good grasp on ORM.

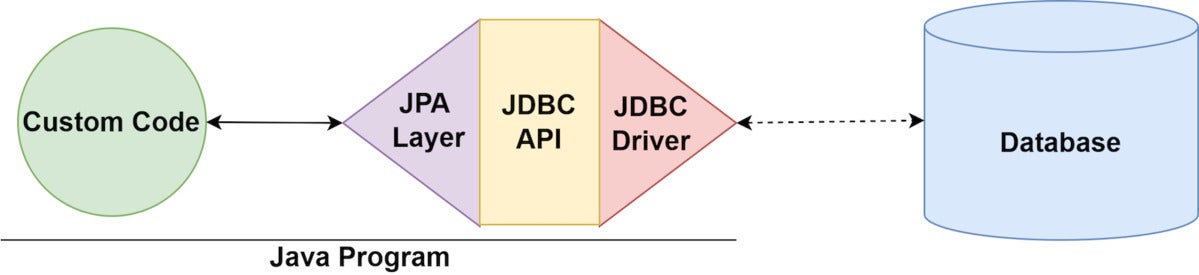
Object-relational mapping is a task–one that developers have good reason to avoid doing manually. A framework like Hibernate ORM or EclipseLink codifies that task into a library or framework, an ORM layer. As part of the application architecture, the ORM layer is responsible for managing the conversion of software objects to interact with the tables and columns in a relational database. In Java, the ORM layer converts Java classes and objects so that they can be stored and managed in a relational database.

By default, the name of the object being persisted becomes the name of the table, and fields become columns. Once the table is set up, each table row corresponds to an object in the application. Object mapping is configurable, but defaults tend to work, and by sticking with defaults, you avoid having to maintain configuration metadata.

### JPA with NoSQL

Until fairly recently, non-relational databases were uncommon curiosities. The NoSQL movement changed all that, and now a variety of NoSQL databases are available to Java developers. Some JPA implementations have evolved to embrace NoSQL, including [Hibernate OGM](http://hibernate.org/ogm/) and [EclipseLink](https://www.eclipse.org/eclipselink).

Figure 1 illustrates the role of JPA and the ORM layer in application development.

[](https://images.idgesg.net/images/article/2022/05/what-is-jpa.drawio-1-100928128-orig.jpg?auto=webp&quality=85,70)

Object Relational Mapping (ORM) is a functionality which is used to develop and maintain a relationship between an object and relational database by mapping an object state to database column. It is capable to handle various database operations easily such as inserting, updating, deleting etc.



ORM Frameworks

Following are the various frameworks that function on ORM mechanism: -

* Hibernate
* TopLink
* ORMLite
* iBATIS
* JPOX

Mapping Directions

Mapping Directions are divided into two parts: -

* **Unidirectional relationship -** In this relationship, only one entity can refer the properties to another. It contains only one owing side that specifies how an update can be made in the database.
* **Bidirectional relationship -** This relationship contains an owning side as well as an inverse side. So here every entity has a relationship field or refer the property to other entity.

Types of Mapping

Following are the various ORM mappings: -

Competitive questions on Structures in HindiKeep Watching

* **One-to-one -** This association is represented by @OneToOne annotation. Here, instance of each entity is related to a single instance of another entity.
* **One-to-many -** This association is represented by @OneToMany annotation. In this relationship, an instance of one entity can be related to more than one instance of another entity.
* **Many-to-one -** This mapping is defined by @ManyToOne annotation. In this relationship, multiple instances of an entity can be related to single instance of another entity.
* **Many-to-many -** This association is represented by @ManyToMany annotation. Here, multiple instances of an entity can be related to multiple instances of another entity. In this mapping, any side can be the owing side.

1. What are the advantages and disadvantages of JPA?

PRO:

* Separates SQL from your business logic. This lets you work with a higher level of abstraction as you can navigate through related classes.
* This higher level of abstraction is useful for people who need to maintain your code after the fact. Changes to business logic would be easier for someone else to take on.
* No need to worry about counting ? signs in in-clauses.
* Integrates with Java Bean Validation which allows you to have richer validation logic than what could be provided by SQL databases.
* Access is portable across databases by default. Useful when you have enterprise clients which can either be SQL Server, MySQL, DB2 or Oracle. So you can support more clients.
* Manages all the SQL in a **single** transaction inside a single request effectively. That being said, avoid using JPA when spanning past a single transaction/request.

CONS:

* Not thread-safe.
* Not good for batch transactions. Takes up way too much memory. The work around is to detach from the entity manager once processing is done for one.
* Understanding of what occurs is needed to improve performance by specifying extra fetches.
* Mapping is hard.

Some extra things to look for are:

* database migrations, for this, use Liquibase, this makes things more predictable than JPA.
* JOOQ is a useful stop-gap if you don’t want to go JPA all the way as it gives an OO interface to SQL. I’ve used that in a project as a stop-gap to get rid of SQLs assembled via strings.
* Also, DO NOT propagate your JPA objects outside of your service facade. Even though that’s a selling point, it’s going to cause more problems than it will solve, it’s better to use a different technology to expose interfaces to your services (e.g. GRPC)

1. What id generation strategies can you name? Can you describe their pros and cons?

## ****Overview****

Identifiers in Hibernate represent the primary key of an entity. This implies the values are unique so that they can identify a specific entity, that they aren't null and that they won't be modified.

Hibernate provides a few different ways to define identifiers. In this article, we'll review each method of mapping entity ids using the library.

## ****2. Simple Identifiers****

The most straightforward way to define an identifier is by using the @Id annotation.

Simple ids are mapped using @Id to a single property of one of these types: Java primitive and primitive wrapper types, String, Date, BigDecimal and BigInteger.

Let's see a quick example of defining an entity with a primary key of type long:

@Entity

**public** **class** **Student** {

@Id

**private** **long** studentId;

// standard constructor, getters, setters

}

## ****3. Generated Identifiers****

If we want to automatically generate the primary key value, **we can add the @GeneratedValue annotation.**

This can use four generation types: AUTO, IDENTITY, SEQUENCE and TABLE.

If we don't explicitly specify a value, the generation type defaults to AUTO.

### **3.1.**AUTO**Generation**

If we're using the default generation type, the persistence provider will determine values based on the type of the primary key attribute. This type can be numerical or UUID.

For numeric values, the generation is based on a sequence or table generator, while UUID values will use the UUIDGenerator.

Let's first map an entity primary key using AUTO generation strategy:

@Entity

**public** **class** **Student** {

@Id

@GeneratedValue

**private** **long** studentId;

// ...

}

In this case, the primary key values will be unique at the database level.

**Now we'll look at the UUIDGenerator, which was introduced in Hibernate 5.**

In order to use this feature, we just need to declare an id of type UUID with @GeneratedValue annotation:

@Entity

**public** **class** **Course** {

@Id

@GeneratedValue

**private** UUID courseId;

// ...

}

Hibernate will generate an id of the form “8dd5f315-9788-4d00-87bb-10eed9eff566”.

### **3.2.**IDENTITY**Generation**

This type of generation relies on the IdentityGenerator, which expects values generated by an identity column in the database. This means they are auto-incremented.

To use this generation type, we only need to set the strategy parameter:

@Entity

**public** **class** **Student** {

@Id

@GeneratedValue (strategy = GenerationType.IDENTITY)

**private** **long** studentId;

// ...

}

One thing to note is that IDENTITY generation disables batch updates.

### **3.3.**SEQUENCE**Generation**

To use a sequence-based id, Hibernate provides the SequenceStyleGenerator class.

This generator uses sequences if our database supports them. It switches to table generation if they aren't supported.

In order to customize the sequence name, we can use the @GenericGenerator annotation with SequenceStyleGenerator strategy:

@Entity

**public** **class** **User** {

@Id

@GeneratedValue(generator = "sequence-generator")

@GenericGenerator(

name = "sequence-generator",

strategy = "org.hibernate.id.enhanced.SequenceStyleGenerator",

parameters = {

@Parameter(name = "sequence\_name", value = "user\_sequence"),

@Parameter(name = "initial\_value", value = "4"),

@Parameter(name = "increment\_size", value = "1")

}

)

**private** **long** userId;

// ...

}

In this example, we've also set an initial value for the sequence, which means the primary key generation will start at 4.

SEQUENCE is the generation type recommended by the Hibernate documentation.

**The generated values are unique per sequence.** If we don't specify a sequence name, Hibernate will reuse the same hibernate\_sequence for different types.

### **3.4. TABLE Generation**

The TableGenerator uses an underlying database table that holds segments of identifier generation values.

Let's customize the table name using the @TableGenerator annotation:

@Entity

**public** **class** **Department** {

@Id

@GeneratedValue(strategy = GenerationType.TABLE,

generator = "table-generator")

@TableGenerator(name = "table-generator",

table = "dep\_ids",

pkColumnName = "seq\_id",

valueColumnName = "seq\_value")

**private** **long** depId;

// ...

}

In this example, we can see that we can also customize other attributes such as the pkColumnName and valueColumnName.

However, the disadvantage of this method is that it doesn't scale well and can negatively affect performance.

**To sum up, these four generation types will result in similar values being generated but use different database mechanisms.**

1. What JPA inheritance types can you name? Can you describe them?

Inheritence is a key feature of object-oriented programming language in which a child class can acquire the properties of its parent class. This feature enhances reusability of the code.

The relational database doesn't support the mechanism of inheritance. So, Java Persistence API (JPA) is used to map the key features of inheritance in relational database model.

JPA Inheritence Annotations

Following are the most frequently used JPA inheritance annotations: -

* **@Inheritence** - This annotation is applied on the root entity class to define the inheritance strategy. If no strategy type is defined with this annotation then it follows single table strategy.
* **@MappedSuperclass** - This annotation is applied to the classes that are inherited by their subclasses. The mapped superclass doesn't contain any separate table.
* **@DiscriminatorColumn** - The discriminator attribute differentiates one entity from another. Thus, this annotation is used to provide the name of discriminator column. It is required to specify this annotation on the root entity class only.
* **@DiscriminatorValue** - This annotation is used to specify the type of value that represents the particular entity. It is required to specify this annotation on the sub-entity classes.

## ****Overview****

Relational databases don't have a straightforward way to map class hierarchies onto database tables.

To address this, the JPA specification provides several strategies:

* MappedSuperclass – the parent classes, can't be entities
* Single Table – The entities from different classes with a common ancestor are placed in a single table.
* Joined Table – Each class has its table, and querying a subclass entity requires joining the tables.
* Table per Class – All the properties of a class are in its table, so no join is required.

Each strategy results in a different database structure.

**Entity inheritance means that we can use polymorphic queries for retrieving all the subclass entities when querying for a superclass.**

Since Hibernate is a JPA implementation, it contains all of the above as well as a few Hibernate-specific features related to inheritance.

In the next sections, we'll go over available strategies in more detail.

## ****2.****MappedSuperclass

Using the MappedSuperclass strategy, inheritance is only evident in the class but not the entity model.

Let's start by creating a Person class that will represent a parent class:

@MappedSuperclass

**public** **class** **Person** {

@Id

**private** **long** personId;

**private** String name;

// constructor, getters, setters

}

**Notice that this class no longer has an @Entity annotation**, as it won't be persisted in the database by itself.

Next, let's add an Employee subclass:

@Entity

**public** **class** **MyEmployee** **extends** **Person** {

**private** String company;

// constructor, getters, setters

}

In the database, this will correspond to one MyEmployee table with three columns for the declared and inherited fields of the subclass.

If we're using this strategy, ancestors cannot contain associations with other entities.

## ****3. Single Table****

**The Single Table strategy creates one table for each class hierarchy.** JPA also chooses this strategy by default if we don't specify one explicitly.

We can define the strategy we want to use by adding the @Inheritance annotation to the superclass:

@Entity

@Inheritance(strategy = InheritanceType.SINGLE\_TABLE)

**public** **class** **MyProduct** {

@Id

**private** **long** productId;

**private** String name;

// constructor, getters, setters

}

The identifier of the entities is also defined in the superclass.

Then we can add the subclass entities:

@Entity

**public** **class** **Book** **extends** **MyProduct** {

**private** String author;

}

@Entity

**public** **class** **Pen** **extends** **MyProduct** {

**private** String color;

}

### **3.1. Discriminator Values**

Since the records for all entities will be in the same table, **Hibernate needs a way to differentiate between them.**

**By default, this is done through a discriminator column called DTYPE** that has the name of the entity as a value.

To customize the discriminator column, we can use the @DiscriminatorColumn annotation:

@Entity(name="products")

@Inheritance(strategy = InheritanceType.SINGLE\_TABLE)

@DiscriminatorColumn(name="product\_type",

discriminatorType = DiscriminatorType.INTEGER)

**public** **class** **MyProduct** {

// ...

}

Here we've chosen to differentiate MyProduct subclass entities by an integer column called product\_type.

Next, we need to tell Hibernate what value each subclass record will have for the product\_type column:

@Entity

@DiscriminatorValue("1")

**public** **class** **Book** **extends** **MyProduct** {

// ...

}

@Entity

@DiscriminatorValue("2")

**public** **class** **Pen** **extends** **MyProduct** {

// ...

}

Hibernate adds two other predefined values that the annotation can take — null and not null:

* @DiscriminatorValue(“null”) means that any row without a discriminator value will be mapped to the entity class with this annotation; this can be applied to the root class of the hierarchy.
* @DiscriminatorValue(“not null”) – Any row with a discriminator value not matching any of the ones associated with entity definitions will be mapped to the class with this annotation.

Instead of a column, we can also use the Hibernate-specific @DiscriminatorFormula annotation to determine the differentiating values:

@Entity

@Inheritance(strategy = InheritanceType.SINGLE\_TABLE)

@DiscriminatorFormula("case when author is not null then 1 else 2 end")

**public** **class** **MyProduct** { ... }

**This strategy has the advantage of polymorphic query performance since only one table needs to be accessed when querying parent entities.**

On the other hand, this also means that **we can no longer use NOT NULL constraints on subclass** entity properties.

## ****4. Joined Table****

**Using this strategy, each class in the hierarchy is mapped to its table.** The only column that repeatedly appears in all the tables is the identifier, which will be used for joining them when needed.

Let's create a superclass that uses this strategy:

@Entity

@Inheritance(strategy = InheritanceType.JOINED)

**public** **class** **Animal** {

@Id

**private** **long** animalId;

**private** String species;

// constructor, getters, setters

}

Then we can simply define a subclass:

@Entity

**public** **class** **Pet** **extends** **Animal** {

**private** String name;

// constructor, getters, setters

}

Both tables will have an animalId identifier column.

The primary key of the Pet entity also has a foreign key constraint to the primary key of its parent entity.

To customize this column, we can add the @PrimaryKeyJoinColumn annotation:

@Entity

@PrimaryKeyJoinColumn(name = "petId")

**public** **class** **Pet** **extends** **Animal** {

// ...

}

**The disadvantage of this inheritance mapping method is that retrieving entities requires joins between tables**, which can result in lower performance for large numbers of records.

The number of joins is higher when querying the parent class because it will join with every single related child — so performance is more likely to be affected the higher up the hierarchy we want to retrieve records.

## ****5. Table per Class****

**The Table per Class strategy maps each entity to its table, which contains all the properties of the entity, including the ones inherited.**

The resulting schema is similar to the one using @MappedSuperclass. But Table per Class will indeed define entities for parent classes, allowing associations and polymorphic queries as a result.

To use this strategy, we only need to add the @Inheritance annotation to the base class:

@Entity

@Inheritance(strategy = InheritanceType.TABLE\_PER\_CLASS)

**public** **class** **Vehicle** {

@Id

**private** **long** vehicleId;

**private** String manufacturer;

// standard constructor, getters, setters

}

Then we can create the subclasses in the standard way.

This is not that different from merely mapping each entity without inheritance. The distinction is clear when querying the base class, which will return all the subclass records as well by using a UNION statement in the background.

**The use of UNION can also lead to inferior performance when choosing this strategy.** Another issue is that we can no longer use identity key generation.

## ****6. Polymorphic Queries****

As mentioned, querying a base class will retrieve all the subclass entities as well.

Let's see this behavior in action with a JUnit test:

@Test

**public** **void** **givenSubclasses\_whenQuerySuperclass\_thenOk**() {

**Book** book = **new** **Book**(1, "1984", "George Orwell");

session.save(book);

**Pen** pen = **new** **Pen**(2, "my pen", "blue");

session.save(pen);

assertThat(session.createQuery("from MyProduct")

.getResultList()).hasSize(2);

}

In this example, we've created two Book and Pen objects and then queried their superclass MyProduct to verify that we'll retrieve two objects.

Hibernate can also query interfaces or base classes that are not entities but are extended or implemented by entity classes.

Let's see a JUnit test using our @MappedSuperclass example:

@Test

**public** **void** **givenSubclasses\_whenQueryMappedSuperclass\_thenOk**() {

**MyEmployee** emp = **new** **MyEmployee**(1, "john", "baeldung");

session.save(emp);

assertThat(session.createQuery(

"from com.baeldung.hibernate.pojo.inheritance.Person")

.getResultList())

.hasSize(1);

}

Note that this also works for any superclass or interface, whether it's a @MappedSuperclass or not. The difference from a usual HQL query is that we have to use the fully qualified name since they are not Hibernate-managed entities.

If we don't want a subclass to be returned by this type of query, we only need to add the Hibernate @Polymorphism annotation to its definition, with type EXPLICIT:

@Entity

@Polymorphism(type = PolymorphismType.EXPLICIT)

**public** **class** **Bag** **implements** **Item** { ...}

In this case, when querying for Items, the Bag records won't be returned.

## ****7. Conclusion****

In this article, we've shown the different strategies for mapping inheritance in Hibernate.

The full source code of the examples can be found [over on GitHub](https://github.com/eugenp/tutorials/tree/master/persistence-modules/hibernate-mapping).

1. What fetch strategies do you know? What is the default? How to reproduce LazyInitializationException?

The *FetchType* defines when Hibernate gets the related entities from the database, and it is one of the crucial elements for a fast persistence tier. In general, you want to fetch the entities you use in your business tier as efficiently as possible. But that’s not that easy. You either get all relationships with one query or you fetch only the root entity and initialize the relationships as soon as you need them.

I’ll explain both approaches in more detail during this post and also provide you some links to more advanced solutions that combine flexibility and efficiency.

**Contents**[[hide](https://thorben-janssen.com/entity-mappings-introduction-jpa-fetchtypes/)]

* [1 Default FetchTypes and how to change it](https://thorben-janssen.com/entity-mappings-introduction-jpa-fetchtypes/#Default_FetchTypes_and_how_to_change_it)
* [2 FetchType.EAGER – Fetch it so you’ll have it when you need it](https://thorben-janssen.com/entity-mappings-introduction-jpa-fetchtypes/#FetchTypeEAGER_8211_Fetch_it_so_you8217ll_have_it_when_you_need_it)
* [3 FetchType.LAZY – Fetch it when you need it](https://thorben-janssen.com/entity-mappings-introduction-jpa-fetchtypes/#FetchTypeLAZY_8211nbspFetch_it_when_you_need_it)
* [4 Summary](https://thorben-janssen.com/entity-mappings-introduction-jpa-fetchtypes/#Summary)

Default FetchTypes and how to change it

When you started with Hibernate, you most, likely either didn’t know about *FetchTypes* or you were told to always use *FetchType.LAZY*. In general, that’s a good recommendation. But what does it exactly mean? And what is the default if you don’t define the *FetchType*?

The default depends on the cardinality of the relationship. All to-one relationships use *FetchType.EAGER* and all to-many relationships *FetchType.LAZY*.

Even the best default doesn’t fit for all use cases, and you sometimes want to change it. You can do this by providing your preferred *FetchType* to the relationship annotation as you can see in the following code snippet.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10 | @Entity  @Table(name = "purchaseOrder")  public class Order implements Serializable {      @OneToMany(mappedBy = "order", fetch = FetchType.EAGER)    private Set<OrderItem> items = new HashSet<OrderItem>();      ...    } |

OK, now let’s have a more detailed look at the different *FetchTypes*.

*FetchType.EAGER* – Fetch it so you’ll have it when you need it

The *FetchType.EAGER* tells Hibernate to get all elements of a relationship when selecting the root entity. As I explained earlier, this is the default for to-one relationships, and you can see it in the following code snippets.

I use the default *FetchType* (*EAGER*) for the many-to-one relationship between the *OrderItem* and *Product* entity.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | @Entity  public class OrderItem implements Serializable  {       @ManyToOne     private Product product;       ...  } |

When I now fetch an *OrderItem* entity from the database, Hibernate will also get the related *Product* entity.

|  |  |
| --- | --- |
| 1  2  3 | OrderItem orderItem = em.find(OrderItem.class, 1L);  log.info("Fetched OrderItem: "+orderItem);  Assert.assertNotNull(orderItem.getProduct()); |
| 1  2 | 05:01:24,504 DEBUG SQL:92 - select orderitem0\_.id as id1\_0\_0\_, orderitem0\_.order\_id as order\_id4\_0\_0\_, orderitem0\_.product\_id as product\_5\_0\_0\_, orderitem0\_.quantity as quantity2\_0\_0\_, orderitem0\_.version as version3\_0\_0\_, order1\_.id as id1\_2\_1\_, order1\_.orderNumber as orderNum2\_2\_1\_, order1\_.version as version3\_2\_1\_, product2\_.id as id1\_1\_2\_, product2\_.name as name2\_1\_2\_, product2\_.price as price3\_1\_2\_, product2\_.version as version4\_1\_2\_ from OrderItem orderitem0\_ left outer join purchaseOrder order1\_ on orderitem0\_.order\_id=order1\_.id left outer join Product product2\_ on orderitem0\_.product\_id=product2\_.id where orderitem0\_.id=?  05:01:24,557  INFO FetchTypes:77 - Fetched OrderItem: OrderItem , quantity: 100 | |

This seems to be very useful in the beginning. Joining the required entities and getting all of them in one query is very efficient.

But keep in mind, that Hibernate will ALWAYS fetch the *Product* entity for your *OrderItem*, even if you don’t use it in your business code. If the related entity isn’t too big, this is not an issue for to-one relationships. But it will most likely slow down your application if you use it for a to-many relationship that you don’t need for your use case. Hibernate then has to fetch tens or even hundreds of additional entities which creates a significant overhead.

*FetchType.LAZY* – Fetch it when you need it

The *FetchType.LAZY* tells Hibernate to only fetch the related entities from the database when you use the relationship. This is a good idea in general because there’s no reason to select entities you don’t need for your uses case. You can see an example of a lazily fetched relationship in the following code snippets.

The *one-to-many* relationship between the *Order* and the *OrderItem* entities uses the default *FetchType* for *to-many* relationships which is lazy.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10 | @Entity  @Table(name = "purchaseOrder")  public class Order implements Serializable {      @OneToMany(mappedBy = "order")    private Set<OrderItem> items = new HashSet<OrderItem>();      ...    } |

The used *FetchType* has no influence on the business code. You can call the *getOrderItems()* method just as any other getter method.

|  |  |
| --- | --- |
| 1  2  3 | Order newOrder = em.find(Order.class, 1L);  log.info("Fetched Order: "+newOrder);  Assert.assertEquals(2, newOrder.getItems().size()); |

Hibernate handles the lazy initialization transparently and fetches the *OrderItem* entities as soon as the getter method gets called.

|  |  |
| --- | --- |
| 1  2  3 | 05:03:01,504 DEBUG SQL:92 - select order0\_.id as id1\_2\_0\_, order0\_.orderNumber as orderNum2\_2\_0\_, order0\_.version as version3\_2\_0\_ from purchaseOrder order0\_ where order0\_.id=?  05:03:01,545  INFO FetchTypes:45 - Fetched Order: Order orderNumber: order1  05:03:01,549 DEBUG SQL:92 - select items0\_.order\_id as order\_id4\_0\_0\_, items0\_.id as id1\_0\_0\_, items0\_.id as id1\_0\_1\_, items0\_.order\_id as order\_id4\_0\_1\_, items0\_.product\_id as product\_5\_0\_1\_, items0\_.quantity as quantity2\_0\_1\_, items0\_.version as version3\_0\_1\_, product1\_.id as id1\_1\_2\_, product1\_.name as name2\_1\_2\_, product1\_.price as price3\_1\_2\_, product1\_.version as version4\_1\_2\_ from OrderItem items0\_ left outer join Product product1\_ on items0\_.product\_id=product1\_.id where items0\_.order\_id=? |

Handling lazy relationships in this way is perfectly fine if you work on a single Order entity or a small list of entities. But it becomes a performance problem when you do it on a large list of entities. As you can see in the following log messages, Hibernate has to perform an additional SQL statement for each *Order* entity to fetch its *OrderItems*.

|  |  |
| --- | --- |
| 1  2  3  4  5 | 05:03:40,936 DEBUG ConcurrentStatisticsImpl:411 - HHH000117: HQL: SELECT o FROM Order o, time: 41ms, rows: 3  05:03:40,939  INFO FetchTypes:60 - Fetched all Orders  05:03:40,942 DEBUG SQL:92 - select items0\_.order\_id as order\_id4\_0\_0\_, items0\_.id as id1\_0\_0\_, items0\_.id as id1\_0\_1\_, items0\_.order\_id as order\_id4\_0\_1\_, items0\_.product\_id as product\_5\_0\_1\_, items0\_.quantity as quantity2\_0\_1\_, items0\_.version as version3\_0\_1\_, product1\_.id as id1\_1\_2\_, product1\_.name as name2\_1\_2\_, product1\_.price as price3\_1\_2\_, product1\_.version as version4\_1\_2\_ from OrderItem items0\_ left outer join Product product1\_ on items0\_.product\_id=product1\_.id where items0\_.order\_id=?  05:03:40,957 DEBUG SQL:92 - select items0\_.order\_id as order\_id4\_0\_0\_, items0\_.id as id1\_0\_0\_, items0\_.id as id1\_0\_1\_, items0\_.order\_id as order\_id4\_0\_1\_, items0\_.product\_id as product\_5\_0\_1\_, items0\_.quantity as quantity2\_0\_1\_, items0\_.version as version3\_0\_1\_, product1\_.id as id1\_1\_2\_, product1\_.name as name2\_1\_2\_, product1\_.price as price3\_1\_2\_, product1\_.version as version4\_1\_2\_ from OrderItem items0\_ left outer join Product product1\_ on items0\_.product\_id=product1\_.id where items0\_.order\_id=?  05:03:40,959 DEBUG SQL:92 - select items0\_.order\_id as order\_id4\_0\_0\_, items0\_.id as id1\_0\_0\_, items0\_.id as id1\_0\_1\_, items0\_.order\_id as order\_id4\_0\_1\_, items0\_.product\_id as product\_5\_0\_1\_, items0\_.quantity as quantity2\_0\_1\_, items0\_.version as version3\_0\_1\_, product1\_.id as id1\_1\_2\_, product1\_.name as name2\_1\_2\_, product1\_.price as price3\_1\_2\_, product1\_.version as version4\_1\_2\_ from OrderItem items0\_ left outer join Product product1\_ on items0\_.product\_id=product1\_.id where items0\_.order\_id=? |

This behavior is called n+1 select issue, and it’s the most common performance problem. It is so common that you most likely have it if you didn’t explicitly search for it. If you’re not sure how to do that, signup for my free, 3-part video course about [finding and fixing n+1 select issues](https://thorben-janssen.com/free-n1_select_course/).

There are two ways to avoid these issues:

1. You can use *FetchType.EAGER* if you know that all of your use cases that fetch an *Order* entity also need to process the related *OrderItem* entities. That will almost never be the case.
2. If there are some use cases which only work on *Order* entities (which is most likely the case), you should use *FetchType.LAZY* in your entity mapping and use one of these options to [initialize the relationship when you need them](https://thorben-janssen.com/5-ways-to-initialize-lazy-relations-and-when-to-use-them/).

Summary

As I said in the beginning, you need to make sure to use the right FetchType for your use case to avoid common Hibernate performance issues. For most use cases, the *FetchType.LAZY* is a good choice. But make sure that you don’t create any n+1 select issues.

Let’s quickly summarize the different *FetchTypes*.

EAGER fetching tells Hibernate to get the related entities with the initial query. This can be very efficient because all entities are fetched with only one query. But in most cases it just creates a huge overhead because you select entities you don’t need in your use case.

You can prevent this with FetchType.LAZY. This tells Hibernate to delay the initialization of the relationship until you access it in your business code. The drawback of this approach is that Hibernate needs to execute an additional query to initialize each relationship.

## When does Hibernate throw a LazyInitializationException

Hibernate throws the LazyInitializationException when it needs to initialize a lazily fetched association to another entity without an active session context. That’s usually the case if you try to use an uninitialized association in your client application or web layer.

Here you can see a test case with a simplified example.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15 | EntityManager em = emf.createEntityManager();  em.getTransaction().begin();    TypedQuery<Author> q = em.createQuery(          "SELECT a FROM Author a",          Author.class);  List<Author> authors = q.getResultList();  em.getTransaction().commit();  em.close();    for (Author author : authors) {      List<Book> books = author.getBooks();      log.info("... the next line will throw LazyInitializationException ...");      books.size();  } |

The database query returns an Author entity with a lazily fetched association to the books this author has written. Hibernate initializes the books attributes with its own List implementation, which handles the lazy loading. When you try to access an element in that List or call a method that operates on its elements, Hibernate’s List implementation recognizes that no active session is available and throws a LazyInitializationException.

## How to NOT fix the LazyInitializationException

As I wrote at the beginning, you can find lots of bad advice on how to fix the LazyInitializationException. Let me quickly explain which suggestions you should ignore.

### **Don’t use FetchType.EAGER**

Some developers suggest changing the [*FetchType*](https://thorben-janssen.com/entity-mappings-introduction-jpa-fetchtypes/) of the association to EAGER. This, of course, fixes the LazyInitializationException, but it introduces performance problems that will show up in production.

When you set the FetchType to EAGER, Hibernate will always fetch the association, even if you don’t use it in your use case. That obviously causes an overhead that slows down your application. But it gets even worse if you don’t use the EntityManager.find method and don’t reference the association in a JOIN FETCH clause. Hibernate then executes an additional query to fetch the association. This often results in the [n+1 select issue](https://thorben-janssen.com/free-n1_select_course/), which is the most common cause of performance issues.

So please, don’t use FetchType.EAGER. As explained in [various](https://thorben-janssen.com/6-hibernate-mappings-you-should-avoid-for-high-performance-applications/) [articles](https://thorben-janssen.com/best-practices-for-many-to-many-associations-with-hibernate-and-jpa/) on this blog, you should always prefer FetchType.LAZY.

1. What cascade types do you know? Is it possible to insert an entity using cascade?

We learned about [**mapping associated entities**](https://howtodoinjava.com/hibernate/how-to-define-association-mappings-between-hibernate-entities/) in hibernate already in previous tutorials such as [**one-to-one mapping**](https://howtodoinjava.com/hibernate/hibernate-one-to-one-mapping/) and [**one-to-many mappings**](https://howtodoinjava.com/hibernate/hibernate-one-to-many-mapping/). There we wanted to save the mapped entities whenever the relationship owner entity gets saved. To enable this behavior, we had used “CascadeType” attribute.

In this JPA Cascade Types tutorial, we will learn about various available options for configuring the cascading behavior via CascadeType.

**JPA Cascade Types**

The cascade types supported by the Java Persistence Architecture are as below:

1. **CascadeType.PERSIST** : cascade type presist means that save() or persist() operations cascade to related entities.
2. **CascadeType.MERGE** : cascade type merge means that related entities are merged when the owning entity is merged.
3. **CascadeType.REFRESH** : cascade type refresh does the same thing for the refresh() operation.
4. **CascadeType.REMOVE** : cascade type remove removes all related entities association with this setting when the owning entity is deleted.
5. **CascadeType.DETACH** : cascade type detach detaches all related entities if a “manual detach” occurs.
6. **CascadeType.ALL** : cascade type all is shorthand for all of the above cascade operations.

There is **no default cascade type in JPA**. By default, no operation is cascaded.

The cascade configuration option accepts an **array of CascadeTypes**; thus, to include only refreshes and merges in the cascade operation for a One-to-Many relationship as in our example, we might use the following:

@OneToMany(cascade={CascadeType.REFRESH, CascadeType.MERGE}, fetch = FetchType.LAZY)

@JoinColumn(name="EMPLOYEE\_ID")

private Set<AccountEntity> accounts;

Above cascading will cause accounts collection to be only merged and refreshed.

In this tutorial, we'll discuss what cascading is in JPA/Hibernate. Then we'll cover the various cascade types that are available, along with their semantics.

## 2. What Is Cascading?

Entity relationships often depend on the existence of another entity, for example the Person–Address relationship. Without the Person, the Address entity doesn't have any meaning of its own. When we delete the Person entity, our Address entity should also get deleted.

Cascading is the way to achieve this. **When we perform some action on the target entity, the same action will be applied to the associated entity.**

### 2.1. JPA Cascade Type

All JPA-specific cascade operations are represented by the javax.persistence.CascadeType enum containing entries:

* ALL
* PERSIST
* MERGE
* REMOVE
* REFRESH
* DETACH

### 2.2. Hibernate Cascade Type

Hibernate supports three additional Cascade Types along with those specified by JPA. These Hibernate-specific Cascade Types are available in org.hibernate.annotations.CascadeType:

* REPLICATE
* SAVE\_UPDATE
* LOCK

## 3. Difference Between the Cascade Types

### 3.1. CascadeType.ALL

CascadeType.ALL **propagates all operations — including Hibernate-specific ones — from a parent to a child entity.**

Let's see it in an example:

@Entity

**public** **class** **Person** {

@Id

@GeneratedValue(strategy = GenerationType.AUTO)

**private** **int** id;

**private** String name;

@OneToMany(mappedBy = "person", cascade = CascadeType.ALL)

**private** List<Address> addresses;

}

**Note that in OneToMany associations, we've mentioned cascade type in the annotation.**

Now let's see the associated entity Address:

@Entity

**public** **class** **Address** {

@Id

@GeneratedValue(strategy = GenerationType.AUTO)

**private** **int** id;

**private** String street;

**private** **int** houseNumber;

**private** String city;

**private** **int** zipCode;

@ManyToOne(fetch = FetchType.LAZY)

**private** Person person;

}

### 3.2. CascadeType.PERSIST

The persist operation makes a transient instance persistent. **Cascade Type PERSIST propagates the persist operation from a parent to a child entity.** When we save the person entity, the address entity will also get saved.

Let's see the test case for a persist operation:

@Test

**public** **void** **whenParentSavedThenChildSaved**() {

**Person** person = **new** **Person**();

**Address** address = **new** **Address**();

address.setPerson(person);

person.setAddresses(Arrays.asList(address));

session.persist(person);

session.flush();

session.clear();

}

When we run the above test case, we'll see the following SQL:

Hibernate: insert into **Person** (name, id) values (?, ?)

Hibernate: insert into **Address** (

city, houseNumber, person\_id, street, zipCode, id) values (?, ?, ?, ?, ?, ?)

### 3.3. CascadeType.MERGE

The merge operation copies the state of the given object onto the persistent object with the same identifier. **CascadeType.MERGE propagates the merge operation from a parent to a child entity.**

Let's test the merge operation:

@Test

**public** **void** **whenParentSavedThenMerged**() {

**int** addressId;

**Person** person = buildPerson("devender");

**Address** address = buildAddress(person);

person.setAddresses(Arrays.asList(address));

session.persist(person);

session.flush();

addressId = address.getId();

session.clear();

**Address** savedAddressEntity = session.find(Address.class, addressId);

**Person** savedPersonEntity = savedAddressEntity.getPerson();

savedPersonEntity.setName("devender kumar");

savedAddressEntity.setHouseNumber(24);

session.merge(savedPersonEntity);

session.flush();

}

When we run the test case, the merge operation generates the following SQL:

Hibernate: **select** address0\_.id **as** id1\_0\_0\_, address0\_.city **as** city2\_0\_0\_, address0\_.houseNumber **as** houseNum3\_0\_0\_, address0\_.person\_id **as** person\_i6\_0\_0\_, address0\_.street **as** street4\_0\_0\_, address0\_.zipCode **as** zipCode5\_0\_0\_ **from** Address address0\_ **where** address0\_.id=?

Hibernate: **select** person0\_.id **as** id1\_1\_0\_, person0\_.name **as** name2\_1\_0\_ **from** Person person0\_ **where** person0\_.id=?

Hibernate: **update** Address **set** city=?, houseNumber=?, person\_id=?, street=?, zipCode=? **where** id=?

Hibernate: **update** Person **set** name=? **where** id=?

Here, we can see that the merge operation first loads both address and person entities and then updates both as a result of CascadeType.MERGE.

### 3.4. CascadeType.REMOVE

As the name suggests, the remove operation removes the row corresponding to the entity from the database and also from the persistent context.

**CascadeType.REMOVE propagates the remove operation from parent to child entity.** **Similar to JPA's CascadeType.REMOVE, we have CascadeType.DELETE, which is specific to Hibernate.** There is no difference between the two.

Now it's time to test CascadeType.Remove:

@Test

**public** **void** **whenParentRemovedThenChildRemoved**() {

**int** personId;

**Person** person = buildPerson("devender");

**Address** address = buildAddress(person);

person.setAddresses(Arrays.asList(address));

session.persist(person);

session.flush();

personId = person.getId();

session.clear();

**Person** savedPersonEntity = session.find(Person.class, personId);

session.remove(savedPersonEntity);

session.flush();

}

When we run the test case, we'll see the following SQL:

Hibernate: **delete** **from** Address **where** id=?

Hibernate: **delete** **from** Person **where** id=?

The address associated with the person also got removed as a result of CascadeType.REMOVE.

### 3.5. CascadeType.DETACH

The detach operation removes the entity from the persistent context. **When we use CascadeType.DETACH, the child entity will also get removed from the persistent context.**

Let's see it in action:

@Test

**public** **void** **whenParentDetachedThenChildDetached**() {

**Person** person = buildPerson("devender");

**Address** address = buildAddress(person);

person.setAddresses(Arrays.asList(address));

session.persist(person);

session.flush();

assertThat(session.contains(person)).isTrue();

assertThat(session.contains(address)).isTrue();

session.detach(person);

assertThat(session.contains(person)).isFalse();

assertThat(session.contains(address)).isFalse();

}

Here, we can see that after detaching person, neither person nor address exists in the persistent context.

### 3.6. CascadeType.LOCK

**Unintuitively, CascadeType.LOCK reattaches the entity and its associated child entity with the persistent context again.**

Let's see the test case to understand CascadeType.LOCK:

@Test

**public** **void** **whenDetachedAndLockedThenBothReattached**() {

**Person** person = buildPerson("devender");

**Address** address = buildAddress(person);

person.setAddresses(Arrays.asList(address));

session.persist(person);

session.flush();

assertThat(session.contains(person)).isTrue();

assertThat(session.contains(address)).isTrue();

session.detach(person);

assertThat(session.contains(person)).isFalse();

assertThat(session.contains(address)).isFalse();

session.unwrap(Session.class)

.buildLockRequest(**new** **LockOptions**(LockMode.NONE))

.lock(person);

assertThat(session.contains(person)).isTrue();

assertThat(session.contains(address)).isTrue();

}

As we can see, when using CascadeType.LOCK, we attached the entity person and its associated address back to the persistent context.

### 3.7. CascadeType.REFRESH

Refresh operations **reread the value of a given instance from the database.** In some cases, we may change an instance after persisting in the database, but later we need to undo those changes.

In that kind of scenario, this may be useful. **When we use this operation with Cascade Type REFRESH, the child entity also gets reloaded from the database whenever the parent entity is refreshed.**

For better understanding, let's see a test case for CascadeType.REFRESH:

@Test

**public** **void** **whenParentRefreshedThenChildRefreshed**() {

**Person** person = buildPerson("devender");

**Address** address = buildAddress(person);

person.setAddresses(Arrays.asList(address));

session.persist(person);

session.flush();

person.setName("Devender Kumar");

address.setHouseNumber(24);

session.refresh(person);

assertThat(person.getName()).isEqualTo("devender");

assertThat(address.getHouseNumber()).isEqualTo(23);

}

Here, we made some changes in the saved entities person and address. When we refresh the person entity, the address also gets refreshed.

### 3.8. CascadeType.REPLICATE

**The replicate operation is used when we have more than one data source and we want the data in sync.** With CascadeType.REPLICATE, a sync operation also propagates to child entities whenever performed on the parent entity.

Now let's test CascadeType.REPLICATE:

@Test

**public** **void** **whenParentReplicatedThenChildReplicated**() {

**Person** person = buildPerson("devender");

person.setId(2);

**Address** address = buildAddress(person);

address.setId(2);

person.setAddresses(Arrays.asList(address));

session.unwrap(Session.class).replicate(person, ReplicationMode.OVERWRITE);

session.flush();

assertThat(person.getId()).isEqualTo(2);

assertThat(address.getId()).isEqualTo(2);

}

Because of CascadeType.REPLICATE, when we replicate the person entity, its associated address also gets replicated with the identifier we set.

### 3.9. CascadeType.SAVE\_UPDATE

CascadeType.SAVE\_UPDATE propagates the same operation to the associated child entity. It's useful when we use **Hibernate-specific operations like save, update and saveOrUpdate.**

Let's see CascadeType.SAVE\_UPDATE in action:

@Test

**public** **void** **whenParentSavedThenChildSaved**() {

**Person** person = buildPerson("devender");

**Address** address = buildAddress(person);

person.setAddresses(Arrays.asList(address));

session.saveOrUpdate(person);

session.flush();

}

Because of CascadeType.SAVE\_UPDATE, when we run the above test case, we can see that the person and address both got saved.

Here's the resulting SQL:

Hibernate: **insert** **into** Person (name, id) **values** (?, ?)

Hibernate: **insert** **into** Address (

city, houseNumber, person\_id, street, zipCode, id) **values** (?, ?, ?, ?, ?, ?)

## 4. Conclusion

In this article, we discussed cascading and the different cascade type options available in JPA and Hibernate.

1. When do we need property orphalRemoval?

**Overview**

In this tutorial, we'll be discussing the difference between two of the options we have for removing entities from our databases when working with [JPA](https://www.baeldung.com/the-persistence-layer-with-spring-and-jpa).

First, we'll start with *CascadeType.REMOVE* which is a way to **delete a child entity or entities when the deletion of its parent happens**. Then we'll take a look at the *orphanRemoval* attribute, which was introduced in JPA 2.0. This provides us with a way to **delete orphaned entities from the database**.

Throughout the tutorial, we'll be using a simple online store domain to demonstrate our examples.

**2. Domain Model**

As mentioned earlier, this article makes use of a simple online store domain. Wherein the *OrderRequest* has a *ShipmentInfo* and a list of *LineItem*.

Given that, let's consider:

* For the removal of *ShipmentInfo,* when the deletion of an *OrderRequest* happens, we'll use *CascadeType.REMOVE*
* For the removal of a *LineItem* from an *OrderRequest*, we'll use *orphanRemoval*

First, let's create a *ShipmentInfo*entity:

@Entity

**public** **class** **ShipmentInfo** {

@Id

@GeneratedValue(strategy = GenerationType.AUTO)

**private** Long id;

**private** String name;

// constructors

}

Next, let's create a *LineItem* entity:

@Entity

**public** **class** **LineItem** {

@Id

@GeneratedValue(strategy = GenerationType.AUTO)

**private** Long id;

**private** String name;

@ManyToOne

**private** OrderRequest orderRequest;

// constructors, equals, hashCode

}

Lastly, let's put it all together by creating an *OrderRequest* entity:

@Entity

**public** **class** **OrderRequest** {

@Id

@GeneratedValue(strategy = GenerationType.AUTO)

**private** Long id;

@OneToOne(cascade = { CascadeType.REMOVE, CascadeType.PERSIST })

**private** ShipmentInfo shipmentInfo;

@OneToMany(orphanRemoval = true, cascade = CascadeType.PERSIST, mappedBy = "orderRequest")

**private** List<LineItem> lineItems;

// constructors

**public** **void** **removeLineItem**(LineItem lineItem) {

lineItems.remove(lineItem);

}

}

It's worth highlighting the *removeLineItem* method, which detaches a *LineItem* from an *OrderRequest*.

**3. *CascadeType.REMOVE***

As stated earlier, marking a reference field with *CascadeType.REMOVE* is a way to**delete a child entity or entities whenever the deletion of its parent happens**.

In our case, an *OrderRequest* has a *ShipmentInfo*, which has a *CascadeType.REMOVE*.

To verify the deletion of *ShipmentInfo* from the database when the deletion of an *OrderRequest* happens, let's create a simple integration test:

@Test

**public** **void** **whenOrderRequestIsDeleted\_thenDeleteShipmentInfo**() {

createOrderRequestWithShipmentInfo();

**OrderRequest** orderRequest = entityManager.find(OrderRequest.class, 1L);

entityManager.getTransaction().begin();

entityManager.remove(orderRequest);

entityManager.getTransaction().commit();

Assert.assertEquals(0, findAllOrderRequest().size());

Assert.assertEquals(0, findAllShipmentInfo().size());

}

**private** **void** **createOrderRequestWithShipmentInfo**() {

**ShipmentInfo** shipmentInfo = **new** **ShipmentInfo**("name");

**OrderRequest** orderRequest = **new** **OrderRequest**(shipmentInfo);

entityManager.getTransaction().begin();

entityManager.persist(orderRequest);

entityManager.getTransaction().commit();

Assert.assertEquals(1, findAllOrderRequest().size());

Assert.assertEquals(1, findAllShipmentInfo().size());

}

From the assertions, we can see that the deletion of *OrderRequest* resulted in the successful deletion of the related *ShipmentInfo* as well.

**4. *orphanRemoval***

As stated earlier, its usage is to **delete orphaned entities from the database**. An entity that is no longer attached to its parent is the definition of being an orphan.

In our case, an *OrderRequest* has a collection of *LineItem*objects wherewe use the *@OneToMany*annotation to identify the relationship*.*This is where we also set the *orphanRemoval*attribute to *true*. To detach a *LineItem* from an *OrderRequest*, we can use the *removeLineItem* method that we previously created.

With everything in place, once we use the *removeLineItem* method and save the *OrderRequest*, the deletion of the orphaned *LineItem*from the database should happen.

To verify the deletion of the orphaned *LineItem* from the database, let's create another integration test:

@Test

**public** **void** **whenLineItemIsRemovedFromOrderRequest\_thenDeleteOrphanedLineItem**() {

createOrderRequestWithLineItems();

**OrderRequest** orderRequest = entityManager.find(OrderRequest.class, 1L);

**LineItem** lineItem = entityManager.find(LineItem.class, 2L);

orderRequest.removeLineItem(lineItem);

entityManager.getTransaction().begin();

entityManager.merge(orderRequest);

entityManager.getTransaction().commit();

Assert.assertEquals(1, findAllOrderRequest().size());

Assert.assertEquals(2, findAllLineItem().size());

}

**private** **void** **createOrderRequestWithLineItems**() {

List<LineItem> lineItems = **new** **ArrayList**<>();

lineItems.add(**new** **LineItem**("line item 1"));

lineItems.add(**new** **LineItem**("line item 2"));

lineItems.add(**new** **LineItem**("line item 3"));

**OrderRequest** orderRequest = **new** **OrderRequest**(lineItems);

entityManager.getTransaction().begin();

entityManager.persist(orderRequest);

entityManager.getTransaction().commit();

Assert.assertEquals(1, findAllOrderRequest().size());

Assert.assertEquals(3, findAllLineItem().size());

}

Again, from the assertions, it shows that we have successfully deleted the orphaned *LineItem* from the database.

Additionally, it's worth mentioning that the *removeLineItem*method modifies the list of *LineItem* instead of reassigning a value to it. Doing the latter will lead to a *PersistenceException*.

To verify the stated behavior, let's create a final integration test:

@Test(expected = PersistenceException.class)

**public** **void** **whenLineItemsIsReassigned\_thenThrowAnException**() {

createOrderRequestWithLineItems();

**OrderRequest** orderRequest = entityManager.find(OrderRequest.class, 1L);

orderRequest.setLineItems(**new** **ArrayList**<>());

entityManager.getTransaction().begin();

entityManager.merge(orderRequest);

entityManager.getTransaction().commit();

}

**5. Conclusion**

In this article, we've explored the difference between *CascadeType.REMOVE* and *orphanRemoval* using a simple online store domain. Also, in order to verify the entities were deleted correctly from our database, we created several integration tests.

1. What entity states do you know?
2. What is the difference between methods find vs getReference?

JPA has the concept of an EntityManager, as you know. During your work in the entity manager some objects are loaded from the database, can be modified and afterwards flushed to the database.

find() has to return an initialized instance of your object. If it is not already loaded in the EntityManager, it is retrieved from the database.

getReference() is allowed to return a proxy instead of an initialized instance, if the entity has not been loaded in the EntityManager before. In this proxy, only the primary key attribute is initialized. **Proxies can be created without hitting the database,** because the only initialized attribute is already given to the getReference() function.

The latter is useful when you have an entity A referencing an entity B, and you want to set the b-attribute of A to B, without having to load B from the database.

Only if you reference other attributes of B, the proxy will be initialized.

The getReference() Method

Let’s take a closer look at the *getReference* method before discussing the differences to the *find* method. JPA’s EntityManager interface defines the *T getReference​(Class<T> entityClass, Object primaryKey)*, and its JavaDoc describes it as follows:

*Get an instance, whose****state may be lazily fetched****. If the requested instance does not exist in the database, the****EntityNotFoundException****is thrown when the instance****state is first accessed****. (The persistence provider runtime is permitted to throw the EntityNotFoundException when getReference is called.) The application should not expect that the instance state will be available upon detachment, unless it was accessed by the application while the entity manager was open.*

[*JavaDoc JPA specficiation*](https://jakarta.ee/specifications/persistence/3.0/apidocs/jakarta.persistence/jakarta/persistence/entitymanager#getReference(java.lang.Class,java.lang.Object))*(emphasis added)*

I highlighted the method’s 2 most important characteristics in the quoted JavaDoc:

1. The entity’s state may be fetched lazily.
2. You can get a reference to a non-existing entity, which will throw an *EntityNotFoundException*on first access.

Both characteristics indicate what’s happening internally when you call the *getReference* method. Instead of generating and executing a database query, Hibernate only instantiates and returns a proxy object using the provided primary key value. As explained in my [guide to Hibernate’s proxies](https://thorben-janssen.com/hibernate-proxies/), Hibernate generates the required proxy at runtime, intercepts all method calls and triggers a database query when necessary.

You can see that in the [log output](https://thorben-janssen.com/hibernate-logging-guide/) when you execute the following test case.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | ChessPlayer chessPlayer = em.getReference(ChessPlayer.class, 1L);  log.info("ChessPlayer class name: "+chessPlayer.getClass().getName());    assertThat(chessPlayer.getId()).isNotNull();  log.info("==== Test Assertions: no select statement till here ====");    log.info("==== Test Assertions: notice the select statement when accessing non-primary key attribute ====");  String firstName = chessPlayer.getFirstName(); |

As you can see in the first line of below’s log output, the call of the *getReference* method returns an object of a generated proxy class. Hibernate sets the primary key attributes when creating the proxy object. Due to this, it doesn’t need to execute a query when you call the *getId()* method.

|  |  |
| --- | --- |
| 1  2  3  4 | 13:19:09,603  INFO TestSample:46 - ChessPlayer class name: com.thorben.janssen.sample.model.ChessPlayer$HibernateProxy$QOLR0LtZ  13:19:09,603  INFO TestSample:48 - ==== Test Assertions: no select statement till here ====  13:19:09,664  INFO TestSample:50 - ==== Test Assertions: notice the select statement when accessing non-primary key attribute ====  13:19:09,671 DEBUG SQL:144 - select chessplaye0\_.id as id1\_1\_0\_, chessplaye0\_.birthDate as birthdat2\_1\_0\_, chessplaye0\_.firstName as firstnam3\_1\_0\_, chessplaye0\_.lastName as lastname4\_1\_0\_, chessplaye0\_.version as version5\_1\_0\_ from ChessPlayer chessplaye0\_ where chessplaye0\_.id=? |

But accessing any of the non-primary key attributes requires a query. Hibernate generates and executes a database query that fetches all columns mapped by the entity class. My *ChessPlayer*entity follows my general recommendation to not use [eager fetching](https://thorben-janssen.com/entity-mappings-introduction-jpa-fetchtypes/). But if it would model any eagerly fetched associations, Hibernate would execute an additional query to fetch each of them.

In the previous example, I called the getReference method with the primary key value of an existing database record. If you do the same with a non-existing primary key value, Hibernate will not recognize this until you access a non-primary key value. It then executes a query, which returns an empty result and throws an *EntityNotFoundException*.

|  |  |
| --- | --- |
| 1  2  3  4  5 | 14:47:58,600  INFO TestSample:62 - ChessPlayer class name: com.thorben.janssen.sample.model.ChessPlayer$HibernateProxy$wONtr20Y  14:47:58,600  INFO TestSample:64 - ==== Test Assertions: no select statement till here ====  14:47:58,643  INFO TestSample:66 - ==== Test Assertions: notice the select statement when accessing non-primary key attribute ====  14:47:58,647 DEBUG SQL:144 - select chessplaye0\_.id as id1\_1\_0\_, chessplaye0\_.birthDate as birthdat2\_1\_0\_, chessplaye0\_.firstName as firstnam3\_1\_0\_, chessplaye0\_.lastName as lastname4\_1\_0\_, chessplaye0\_.version as version5\_1\_0\_ from ChessPlayer chessplaye0\_ where chessplaye0\_.id=?  14:47:58,654 ERROR TestSample:72 - javax.persistence.EntityNotFoundException: Unable to find com.thorben.janssen.sample.model.ChessPlayer with id 9999 |

Differences to the find() Method

If you have been using JPA or Hibernate for a while, the main difference between the *getReference* and the *find* method might have already become obvious. It’s the time when your persistence provider executes the database query. The *find* method returns an instance of the entity object and not just a proxy object. If the persistence context doesn’t already contain that object, this requires a database query.

You can see that in the log output when we execute the following test case.

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | ChessPlayer chessPlayer = em.find(ChessPlayer.class, 1L);  log.info("ChessPlayer class name: "+chessPlayer.getClass().getName());    log.info("==== Test Assertions: select query is already done ====");  assertThat(chessPlayer).isNotNull();  assertThat(chessPlayer.getLastName()).isEqualTo("Smyslow"); |

This time, the call of the *find*method triggered a database query and returned an instance of my *ChessPlayer* class. Hibernate initialized all basic attributes of that object, and the call of the *getLastName()* method doesn’t require an additional database query.

|  |  |
| --- | --- |
| 1  2  3 | 14:42:47,925 DEBUG SQL:144 - select chessplaye0\_.id as id1\_1\_0\_, chessplaye0\_.birthDate as birthdat2\_1\_0\_, chessplaye0\_.firstName as firstnam3\_1\_0\_, chessplaye0\_.lastName as lastname4\_1\_0\_, chessplaye0\_.version as version5\_1\_0\_ from ChessPlayer chessplaye0\_ where chessplaye0\_.id=?  14:42:47,952  INFO TestSample:61 - ChessPlayer class name: com.thorben.janssen.sample.model.ChessPlayer  14:42:47,952  INFO TestSample:63 - ==== Test Assertions: select query is already done ==== |

And if no database record with the provided primary key value exists, Hibernate returns null.

|  |  |
| --- | --- |
| 1  2  3  4 | ChessPlayer chessPlayer = em.find(ChessPlayer.class, 1L);    log.info("==== Test Assertions: select query is already done ====");  assertThat(chessPlayer).isNull(); |

When to Use the getReference() Instead of the find() Method

After we discussed the differences between the *find* and the *getReference* methods, you might ask yourself when you should use which of them. You could, of course, use the *getReference* as a general replacement of the *find* method. But I don’t recommend it. In most situations, it doesn’t provide any benefits. But the delayed execution of the SELECT statement and a potential *EntityNotFoundException* make your application harder to understand and debug. That’s why I recommend:

*Always use the find method unless you only need a reference to an entity but not the entity object itself.*

1. What query types do you know? Can you name their pros and cons?

## ****Overview****

In this tutorial, we'll discuss the different types of [JPA](https://www.baeldung.com/jpa-hibernate-difference) queries. Moreover, we'll focus on comparing the differences between them and expanding on each one's pros and cons.

## 2. Setup

Firstly, let's define the UserEntity class we'll use for all examples in this article:

@Table(name = "users")

@Entity

**public** **class** **UserEntity** {

@Id

**private** Long id;

**private** String name;

//Standard constructor, getters and setters.

}

**There are three basic types of JPA Queries:**

* Query, written in Java Persistence Query Language (JPQL) syntax
* NativeQuery, written in plain SQL syntax
* Criteria API Query, constructed programmatically via different methods

Let's explore them.

## 3. Query

**A Query is similar in syntax to SQL, and it's generally used to perform CRUD operations:**

**public** UserEntity **getUserByIdWithPlainQuery**(Long id) {

**Query** jpqlQuery = getEntityManager().createQuery("SELECT u FROM UserEntity u WHERE u.id=:id");

jpqlQuery.setParameter("id", id);

**return** (UserEntity) jpqlQuery.getSingleResult();

}

This Query retrieves the matching record from the users table and also maps it to the UserEntity object.

There are two additional Query sub-types:

* TypedQuery
* NamedQuery

Let's see them in action.

### **3.1.**TypedQuery

We need to pay attention to the return statement in our previous example. JPA can't deduce what the Query result type will be, and, as a result, we have to cast.

But, **JPA provides a special Query sub-type known as a TypedQuery.**This is always preferred if we know our Query result type beforehand. Additionally, it makes our code much more reliable and easier to test.

Let's see a TypedQuery alternative, compared to our first example:

**public** UserEntity **getUserByIdWithTypedQuery**(Long id) {

TypedQuery<UserEntity> typedQuery

= getEntityManager().createQuery("SELECT u FROM UserEntity u WHERE u.id=:id", UserEntity.class);

typedQuery.setParameter("id", id);

**return** typedQuery.getSingleResult();

}

This way, **we get stronger typing for free,** avoiding possible casting exceptions down the road.

### **3.2.**NamedQuery

While we can dynamically define a Query on specific methods, they can eventually grow into a hard-to-maintain codebase. What if we could keep general usage queries in one centralized, easy-to-read place?

JPA's also got us covered on this with another Query sub-type known as a [*NamedQuery*](https://www.baeldung.com/hibernate-named-query).

We can define NamedQueries in orm.xml or a properties file.

**Also, we can define NamedQuery on the Entity class itself, providing a centralized, quick and easy way to read and find an Entity‘s related queries.**

All NamedQueries must have a unique name.

Let's see how we can add a NamedQuery to our UserEntity class:

@Table(name = "users")

@Entity

@NamedQuery(name = "UserEntity.findByUserId", query = "SELECT u FROM UserEntity u WHERE u.id=:userId")

**public** **class** **UserEntity** {

@Id

**private** Long id;

**private** String name;

//Standard constructor, getters and setters.

}

**The @NamedQuery annotation has to be grouped inside a @NamedQueries annotation if we're using Java before version 8. From Java 8 forward, we can simply repeat the @NamedQuery annotation at our Entity class.**

Using a NamedQuery is very simple:

**public** UserEntity **getUserByIdWithNamedQuery**(Long id) {

**Query** namedQuery = getEntityManager().createNamedQuery("UserEntity.findByUserId");

namedQuery.setParameter("userId", id);

**return** (UserEntity) namedQuery.getSingleResult();

}

## 4. NativeQuery

**A NativeQuery is simply an SQL query. These allow us to unleash the full power of our database, as we can use proprietary features not available in JPQL-restricted syntax.**

This comes at a cost. We lose the database portability of our application with NativeQuery because our JPA provider can't abstract specific details from the database implementation or vendor anymore.

Let's see how to use a NativeQuery that yields the same results as our previous examples:

**public** UserEntity **getUserByIdWithNativeQuery**(Long id) {

**Query** nativeQuery

= getEntityManager().createNativeQuery("SELECT \* FROM users WHERE id=:userId", UserEntity.class);

nativeQuery.setParameter("userId", id);

**return** (UserEntity) nativeQuery.getSingleResult();

}

We must always consider if a NativeQuery is the only option. **Most of the time, a good JPQL Query can fulfill our needs and most importantly, maintain a level of abstraction from the actual database implementation.**

Using NativeQuery doesn't necessarily mean locking ourselves to one specific database vendor. After all, if our queries don't use proprietary SQL commands and are using only a standard SQL syntax, switching providers should not be an issue.

## 5. Query, NamedQuery, and NativeQuery

So far, we've learned about Query, NamedQuery, and NativeQuery.

Now, let's revisit them quickly and summarize their pros and cons.

### 5.1. Query

We can create a query using entityManager.createQuery(queryString).

Next, let's explore the pros and cons of Query:

Pros:

* When we create a query using EntityManager, we can build dynamic query strings
* Queries are written in JPQL, so they're portable

Cons:

* For a dynamic query, it may be compiled into a native SQL statement multiple times depending on the [**query plan cache**](https://www.baeldung.com/hibernate-query-plan-cache)
* The queries may scatter into various Java classes and they're mixed in with Java code. Therefore, it could be difficult to maintain if a project contains many queries

### 5.2. NamedQuery

Once a NamedQuery has been defined, we can reference it using the EntityManager:

entityManager.createNamedQuery(queryName);

Now, let's look at the advantages and disadvantages of NamedQueries:

Pros:

* NamedQueries are compiled and validated when the persistence unit is loaded. That is to say, they're compiled only once
* We can centralize NamedQueries to make them easier to maintain – for example, in orm.xml, in properties files, or on @Entity classes

Cons:

* NamedQueries are always static
* NamedQueries can be referenced in Spring Data JPA repositories. However, [**dynamic sorting**](https://www.baeldung.com/spring-data-sorting#sorting-with-spring-data) is not supported

### 5.3. NativeQuery

We can create a NativeQuery using EntityManager:

entityManager.createNativeQuery(sqlStmt);

Depending on the result mapping, we can also pass the second parameter to the method, such as an Entity class, as we've seen in a previous example.

NativeQueries have pros and cons, too. Let's look at them quickly:

Pros:

* As our queries get complex, sometimes the JPA-generated SQL statements aren't the most optimized. In this case, we can use NativeQueries to make the queries more efficient
* NativeQueries allow us to use database vendor-specific features. Sometimes, those features can give our queries better performance

Cons:

* Vendor-specific features can bring convenience and better performance, but we pay for that benefit by losing the portability from one database to another

## 6. Criteria****API Query****

**[Criteria API queries](https://www.baeldung.com/hibernate-criteria-queries) are programmatically-built, type-safe queries – somewhat similar to JPQL queries in syntax:**

**public** UserEntity **getUserByIdWithCriteriaQuery**(Long id) {

**CriteriaBuilder** criteriaBuilder = getEntityManager().getCriteriaBuilder();

CriteriaQuery<UserEntity> criteriaQuery = criteriaBuilder.createQuery(UserEntity.class);

Root<UserEntity> userRoot = criteriaQuery.from(UserEntity.class);

**UserEntity** queryResult = getEntityManager().createQuery(criteriaQuery.select(userRoot)

.where(criteriaBuilder.equal(userRoot.get("id"), id)))

.getSingleResult();

**return** queryResult;

}

It can be daunting to use Criteria API queries first-hand, but they can be a great choice when we need to add dynamic query elements or when coupled with the [JPA Metamodel.](https://www.baeldung.com/hibernate-criteria-queries-metamodel)

## ****7. Conclusion****

In this quick article, we learned what JPA Queries are, along with their usage.

JPA Queries are a great way to abstract our business logic from our data access layer as we can rely on JPQL syntax and let our JPA provider of choice handle the Query translation.

1. Can we update an entity without calling merge method?

JPA and Hibernate provide different methods to persist new and to update existing entities. You can choose between JPA’s persist and merge and Hibernate’s save and update methods.

It seems like there are 2 pairs of 2 methods that do the same.  You can use the methods persist and save to store a new entity and the methods merge and update to store the changes of a detached entity in the database. That’s why a lot of developers are wondering which of these methods they should use. Let’s take a closer look at the details and small differences of these methods.

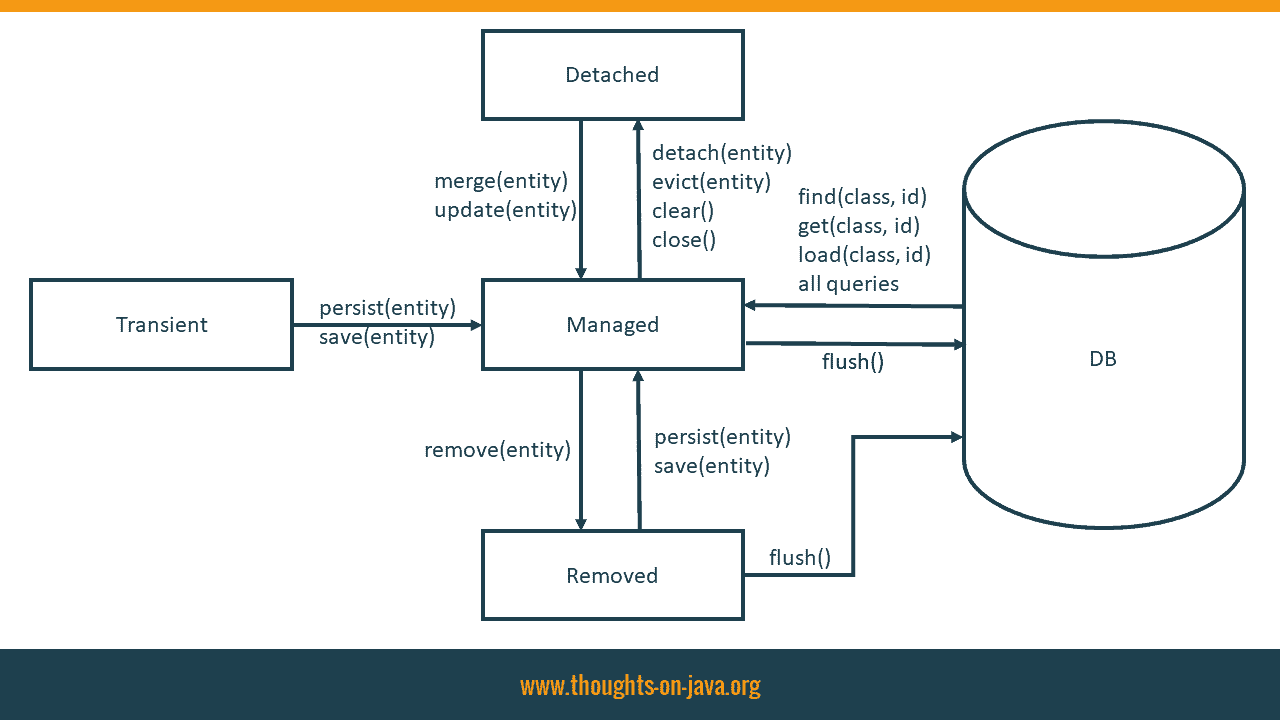
Special thanks to [Steve Ebersole (Lead Developer – Hibernate ORM)](https://twitter.com/steveebersole) who provided his feedback and great insights on some of Hibernate’s hidden implementation details!

**Contents**[[hide](https://thorben-janssen.com/persist-save-merge-saveorupdate-whats-difference-one-use/)]

* [1 Entity State Transitions](https://thorben-janssen.com/persist-save-merge-saveorupdate-whats-difference-one-use/#Entity_State_Transitions)
* [2 Persisting A New Entity Using persist Or save](https://thorben-janssen.com/persist-save-merge-saveorupdate-whats-difference-one-use/#Persisting_A_New_Entity_Using_persist_Or_save)
  + [2.1 Specification vs. Proprietary API](https://thorben-janssen.com/persist-save-merge-saveorupdate-whats-difference-one-use/#Specification_vs_Proprietary_API)
  + [2.2 Return Types And Execution Of SQL Statements](https://thorben-janssen.com/persist-save-merge-saveorupdate-whats-difference-one-use/#Return_Types_And_Execution_Of_SQL_Statements)
    - [2.2.1 Not generated](https://thorben-janssen.com/persist-save-merge-saveorupdate-whats-difference-one-use/#Not_generated)
    - [2.2.2 Generated with IDENTITY strategy](https://thorben-janssen.com/persist-save-merge-saveorupdate-whats-difference-one-use/#Generated_with_IDENTITY_strategy)
    - [2.2.3 Generated with SEQUENCE strategy](https://thorben-janssen.com/persist-save-merge-saveorupdate-whats-difference-one-use/#Generated_with_SEQUENCE_strategy)
    - [2.2.4 Generated with TABLE strategy](https://thorben-janssen.com/persist-save-merge-saveorupdate-whats-difference-one-use/#Generated_with_TABLE_strategy)
  + [2.3 Which one to choose?](https://thorben-janssen.com/persist-save-merge-saveorupdate-whats-difference-one-use/#Which_one_to_choose)
* [3 Updating a detached entity](https://thorben-janssen.com/persist-save-merge-saveorupdate-whats-difference-one-use/#Updating_a_detached_entity)
  + [3.1 JPA’s merge method](https://thorben-janssen.com/persist-save-merge-saveorupdate-whats-difference-one-use/#JPA8217s_merge_method)
  + [3.2 Hibernate’s update method](https://thorben-janssen.com/persist-save-merge-saveorupdate-whats-difference-one-use/#Hibernate8217s_update_method)
  + [3.3 Which one to choose?](https://thorben-janssen.com/persist-save-merge-saveorupdate-whats-difference-one-use/#Which_one_to_choose-2)
* [4 Updating a managed entity](https://thorben-janssen.com/persist-save-merge-saveorupdate-whats-difference-one-use/#Updating_a_managed_entity)

## Entity State Transitions

Before we get into the details of these 4 methods, I need to give you a quick introduction to JPA’s entity lifecycle states.

[](https://thorben-janssen.com/wp-content/uploads/2017/08/Entity-LifeCycle-State-tiny.png)

If an entity is attached to the current persistence context, it has the lifecycle state managed. That means that it is mapped to a database record. Your persistence provider generates the required SQL INSERT and UPDATE statements to propagate all changes. A managed entity is also stored in the 1st level cache.

When you create a new entity, it’s in the transient state. It remains in this state until you attach it to the current persistence context. I will show you how you can do that with JPA’s persist and Hibernate’s save method, in the following section. As long as an entity is in the transient state, it is not mapped to a database record and not managed by any persistence context.

Entities in the detached lifecycle state are no longer managed by the persistence context. That can be the case because you closed the persistence context or you explicitly detached the entity from the current context. I will get into more details about how you can reattach these entities with JPA’s merge and Hibernate’s update methods in a later part of this post.

And the last lifecycle state is removed. These entities were previously in the state managed, before you scheduled them for removal. Removing entities is outside of the scope of this post, so I will not get into too many details about it. You can schedule an entity for removal by calling the remove method on the EntityManager interface.

## Persisting A New Entity Using persist Or save

When you create a new entity object, it’s in the transient lifecycle state. It does not map any database record.

|  |  |
| --- | --- |
| 1  2  3 | Author a = new Author();  a.setFirstName("Thorben");  a.setLastName("Janssen"); |

You need to attach the entity to a persistence context so that it becomes managed and gets persisted in the database. You can either use JPA’s persist or Hibernate’s save method to do that. Both methods seem to do the same, but there are a few differences.

### **Specification vs. Proprietary API**

The most obvious difference is that the JPA specification defines the persist method. You can use it with all JPA implementations. The save method, on the other hand, is Hibernate-specific. It is, therefore, not available in other JPA implementations.

But that’s only relevant if you want to be able to replace Hibernate with another JPA implementation, like Eclipse Link or OpenJPA.

### **Return Types And Execution Of SQL Statements**

Another obvious difference between these 2 methods is their return type. JPA’s persist method returns void and Hibernate’s save method returns the primary key of the entity.

That might seem like a huge difference, especially when you take a closer look at Hibernate’s Javadoc and the JPA specification:

* The Javadoc of Hibernate’s save method states that it generates the primary key value first:

*Persist the given transient instance, first assigning a generated identifier.*[*Javadoc Session.save(entity)*](https://docs.jboss.org/hibernate/orm/5.2/javadocs/org/hibernate/Session.html#save-java.lang.Object-)

* You don’t find any information about this in the JPA specification. It doesn’t define when the primary key value has to be assigned. So, the persistence provider can do that at any time between the call of the persist method and the flush of the persistence context.

In most cases, it doesn’t make any difference if you call the save or persist method. Hibernate uses the name of the entity class and the primary key value to store the entity in the first level cache. It, therefore, needs a primary key value when it executes the persist method.

In almost all situations, Hibernate [generates the primary key value](https://thorben-janssen.com/jpa-generate-primary-keys/) immediately and triggers an SQL statement if necessary, when you call the persist or save method.

But that is not the case if you use the IDENTITY strategy and try to persist an entity without an active transaction or with FlushMode.MANUAL. If you call the persist method in one of these situations, Hibernate delays the execution of the SQL INSERT statement and creates a temporary primary key value. But if you call the save method, Hibernate performs the SQL INSERT statement immediately and retrieves the primary key value from the database.

You can then retrieve it as the return value of the save method.

|  |  |
| --- | --- |
| 1  2  3  4  5 | Author a = new Author();  a.setFirstName("Thorben");  a.setLastName("Janssen");    Long id = (Long) em.unwrap(Session.class).save(a); |

Or you can call the getter method of the primary key attribute of your managed entity if you use JPA’s persist method.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | Author a = new Author();  a.setFirstName("Torben");  a.setLastName("Janssen");    em.persist(a);    Long id = a.getId(); |

Hibernate executes the same SQL statements when you call the persist or the save method. Which and when it does that depends on your [primary key generation strategy](https://thorben-janssen.com/jpa-generate-primary-keys/):

#### **Not generated**

If you set the primary key value programmatically, e.g. to a [natural identifier](https://thorben-janssen.com/naturalid-good-way-persist-natural-ids-hibernate/), Hibernate only performs an SQL INSERT statement when it flushes the persistence context.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | 14:08:34,979  INFO TestPersistSaveMerge:237 - Save entity  14:08:35,052  INFO TestPersistSaveMerge:240 - Commit transaction  14:08:35,123 DEBUG SQL:92 -      insert      into          Author          (firstName, lastName, version, id)      values          (?, ?, ?, ?) |

[Follow me on YouTube](https://bit.ly/2cUsid8) to not miss any new videos.

#### **Generated with IDENTITY strategy**

If you use the IDENTITY strategy to generate the primary key value, Hibernate needs to execute the INSERT statement when you call the save or persist method to retrieve the primary key value from the database.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | 14:09:28,264  INFO TestPersistSaveMerge:237 - Save entity  14:09:28,336 DEBUG SQL:92 -      insert      into          Author          (firstName, lastName, version)      values          (?, ?, ?)  14:09:28,354  INFO TestPersistSaveMerge:240 - Commit transaction |

[Follow me on YouTube](https://bit.ly/2cUsid8) to not miss any new videos.

#### **Generated with SEQUENCE strategy**

And if you use the SEQUENCE, Hibernate performs an SQL SELECT statement to retrieve the next value from the database sequence. Hibernate then delays the INSERT statement until it flushes the persistence context. In this example, the flush happens when the transaction gets committed.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12 | 14:10:27,994  INFO TestPersistSaveMerge:237 - Save entity  14:10:28,002 DEBUG SQL:92 -      select          nextval ('hibernate\_sequence')  14:10:28,042  INFO TestPersistSaveMerge:240 - Commit transaction  14:10:28,096 DEBUG SQL:92 -      insert      into          Author          (firstName, lastName, version, id)      values          (?, ?, ?, ?) |

#### **Generated with TABLE strategy**

You shouldn’t use the TABLE strategy because it requires row level locks on the primary key table and doesn’t scale well. If you use this strategy anyways, Hibernate performs an SQL SELECT statement to retrieve the next primary key value from the database and writes the new value to the database table. It delays the execution of the SQL INSERT statement for the new entity until it flushes the persistence context.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32 | 14:11:17,368  INFO TestPersistSaveMerge:237 - Save entity  14:11:17,482 DEBUG SQL:92 -      select          tbl.next\_val      from          hibernate\_sequences tbl      where          tbl.sequence\_name=? for update              of tbl  14:11:17,531 DEBUG SQL:92 -      insert      into          hibernate\_sequences          (sequence\_name, next\_val)      values          (?,?)  14:11:17,534 DEBUG SQL:92 -      update          hibernate\_sequences      set          next\_val=?      where          next\_val=?          and sequence\_name=?  14:11:17,584  INFO TestPersistSaveMerge:240 - Commit transaction  14:11:17,655 DEBUG SQL:92 -      insert      into          Author          (firstName, lastName, version, id)      values          (?, ?, ?, ?) |

### **Which one to choose?**

You could expect that the save and persist method behave differently because there are a few differences between the JPA specification and the Javadoc of Hibernate’s proprietary methods.

But almost all of these differences disappear when you take a look at the internal implementation. The only ones that remain are 2 corner cases in which Hibernate might delay the retrieval of the primary key, the return type of the method and the support by other JPA implementations.

For most applications, it doesn’t make any difference if you get the generated primary key value as the return type of Hibernate’s save method or from the getter method of your primary key attribute. As long as you don’t use an extended persistence context and perform all database operations with an active transaction, I recommend using JPA’s persist method.

## Updating a detached entity

When you close the current persistence context or explicitly remove an entity from it by calling the clear or detach methods on the EntityManager interface, the entity becomes detached. That means that it is no longer stored in the [1st level cache](https://www.youtube.com/watch?v=0lZnBTTbmRQ) and that Hibernate will not replicate any of the applied changes to the database.

You can use Hibernate’s update or JPA’s merge method to associate a detached entity with a persistence context. After you’ve done that, Hibernate will update the database based on the entity attribute values.

The effect of the update and merge method seem to be the same, but as you will see in the following sections, there is an important difference.

### **JPA’s merge method**

JPA’s merge method copies the state of a detached entity to a managed instance of the same entity. Hibernate, therefore, executes an SQL SELECT statement to retrieve a managed entity from the database. If the persistence context already contained a managed instance of the entity, Hibernate uses the existing one instead. It then copies all attribute values to the managed entity and returns it to the caller.

|  |  |
| --- | --- |
| 1 | Author managedAuthor = em.merge(a); |

After [activating the logging of SQL statements](https://thorben-janssen.com/hibernate-logging-guide/), you can see the executed SELECT and UPDATE statements in the log output.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35 | 11:37:21,172 DEBUG SQL:92 -      select          books0\_.bookId as bookId1\_2\_0\_,          books0\_.authorId as authorId2\_2\_0\_,          book1\_.id as id1\_1\_1\_,          book1\_.fk\_author as fk\_autho6\_1\_1\_,          book1\_.format as format2\_1\_1\_,          book1\_.publishingDate as publishi3\_1\_1\_,          book1\_.title as title4\_1\_1\_,          book1\_.version as version5\_1\_1\_,          author2\_.id as id1\_0\_2\_,          author2\_.firstName as firstNam2\_0\_2\_,          author2\_.lastName as lastName3\_0\_2\_,          author2\_.version as version4\_0\_2\_      from          BookAuthor books0\_      inner join          Book book1\_              on books0\_.authorId=book1\_.id      left outer join          Author author2\_              on book1\_.fk\_author=author2\_.id      where          books0\_.bookId=?  11:37:21,180  INFO TestPersistSaveMerge:82 - Before commit  11:37:21,182 DEBUG SQL:92 -      update          Author      set          firstName=?,          lastName=?,          version=?      where          id=?          and version=? |

When Hibernate flushes the persistence context for the next time, its dirty checking mechanism checks all managed entities. If it detects that the merge operation changed any entity attribute value, it triggers the required SQL UPDATE statement.

There is one important detail you need to know when you use JPA’s merge method. Hibernate copies the attribute values of the detached entity to the managed entity. This overwrites any changes that you performed on this entity within the current Session.

### **Hibernate’s update method**

Hibernate’s update method doesn’t trigger an SQL SELECT statement. It just attaches the entity to the current persistence context. In contrast to JPA’s merge method, you can’t lose any changes by calling the update method. If the persistence context already contains a managed instance of the entity you want to update, it throws an exception.

|  |  |
| --- | --- |
| 1 | em.unwrap(Session.class).update(a); |

When Hibernate performs the next flush, it doesn’t perform any dirty checks. That’s not possible because Hibernate didn’t read the latest version of the entity from the database. It just executes an SQL UPDATE statement for the reattached entity.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11 | 11:38:28,151  INFO TestPersistSaveMerge:121 - Before commit  11:38:28,153 DEBUG SQL:92 -      update          Author      set          firstName=?,          lastName=?,          version=?      where          id=?          and version=? |

The missing dirty check causes an unnecessary SQL UPDATE statement when the entity and the corresponding database record contain the same values. This might be a problem if your DBA registered an update trigger for the database table. In these situations, you can annotate your entity with @SelectBeforeUpdate.

|  |  |
| --- | --- |
| 1  2  3 | @Entity  @SelectBeforeUpdate  public class Author { ... } |

That tells Hibernate to select the entity and perform a dirty check before it generates the SQL UPDATE statement. As you can see in the log output, the behavior of the update method is now similar to JPA’s merge method.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21 | 19:08:16,530  INFO TestPersistSaveMerge:121 - Before commit  19:08:16,531 DEBUG SQL:92 -      select          author\_.id,          author\_.firstName as firstNam2\_0\_,          author\_.lastName as lastName3\_0\_,          author\_.version as version4\_0\_      from          Author author\_      where          author\_.id=?  19:08:16,592 DEBUG SQL:92 -      update          Author      set          firstName=?,          lastName=?,          version=?      where          id=?          and version=? |

But there is a significant difference between the 2 methods. When you call the update method, Hibernate will only select the entity which you provided as a method parameter. But when you call JPA’s merge method, Hibernate will also select all associations with CascadeType.MERGE. You should, therefore, prefer JPA’s merge method if you reattach a huge graph of entities.

### **Which one to choose?**

There is no general answer to this questions. As you have seen, both methods have their advantages and disadvantages. You have to decide for your specific use case if Hibernate needs to select the entity before it triggers the SQL UPDATE statement. And if that’s the case, you also need to consider the depth of your entity graph and the performance implications of the provided fetching behavior.

## Updating a managed entity

JPA and Hibernate make it very easy to update a managed entity. If your entity is in the lifecycle state managed, e.g. because you fetched it with a [JPQL query](https://thorben-janssen.com/jpql/) or the find method of the EntityManager, you just need to change the values of your entity attributes.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | em = emf.createEntityManager();  em.getTransaction().begin();    a = em.find(Author.class, a.getId());  a.setFirstName("Thorben");    log.info("Before commit");  em.getTransaction().commit();  em.close(); |

When Hibernate decides to flush the persistence context, the dirty checking mechanism will detect the change and perform the required SQL UPDATE statement.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21 | 11:41:49,178 DEBUG SQL:92 -      select          author0\_.id as id1\_0\_0\_,          author0\_.firstName as firstNam2\_0\_0\_,          author0\_.lastName as lastName3\_0\_0\_,          author0\_.version as version4\_0\_0\_      from          Author author0\_      where          author0\_.id=?  11:41:49,191  INFO TestPersistSaveMerge:335 - Before commit  11:41:49,193 DEBUG SQL:92 -      update          Author      set          firstName=?,          lastName=?,          version=?      where          id=?          and version=? |

You don’t need to, and you should not call Hibernate’s save method after you updated an entity. That triggers an additional SaveOrUpdate event without providing any benefits. When Hibernate decides to flush the persistence context, it will perform the dirty check anyway to detect all changes before it executes the required SQL UPDATE statements.