**SPRING BOOT**

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

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

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

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

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

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

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

*SECURITY ON A WEB APPLICATION*

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

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

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

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

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

**SPRING BOOT ACTIVE PROFILE**

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

*SPRING ACTIVE PROFILE IN APPLICATION.PROPERTIES*

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

***application.properties***

server.port = 8080

spring.application.name = demoservice

***application-dev.properties***

server.port = 9090

spring.application.name = demoservice

***application-prod.properties***

server.port = 4431

spring.application.name = demoservice

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



*SPRING ACTIVE PROFILE FOR APPLICATION.YML*

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

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

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

spring:

application:

name: demoservice

server:

port: 8080

---

spring:

profiles: dev

application:

name: demoservice

server:

port: 9090

---

spring:

profiles: prod

application:

name: demoservice

server:

port: 4431

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



*MAIN DIFFERENCES BETWEEN YAML AND .PROPERTIES*

| YAML(.yml) | .properties |
| --- | --- |
| Spec can be found [here](https://yaml.org/spec/) | It doesn’t actually have a spec. The closest thing it has to a spec is the javadoc. |
| Human Readable (both do quite well in human readability) | Human Readable |
| Supports key/val, basically map, List and scalar types (int, string etc.) | Supports key/val, but doesn’t support values beyond the string |
| Its usage is quite prevalent in many languages like Python, Ruby, and Java | It is primarily used in java |
| Hierarchical Structure | Non-Hierarchical Structure |
| Spring Framework doesn’t support @PropertySources with .yml files | supports @PropertySources with .properties file |
| If you are using spring profiles, you can have multiple profiles in one single .yml file | Each profile need one separate .properties file |
| While retrieving the values from .yml file we get the value as whatever the respective type (int, string etc.) is in the configuration | While in case of the .properties files we get strings regardless of what the actual value type is in the configuration |

**We need to use. properties or .yml file. when:**

Strictly speaking, .yml file is advantageous over .properties file as it has type safety, hierarchy and supports list but if you are using spring, spring has a number of conventions as well as type conversions that allow you to get effectively all of these same features that YAML provides for you.

One advantage that you may see out of using the YAML(.yml) file is if you are using more than one application that read the same configuration file. you may see better support in other languages for YAML(.yml) as opposed to .properties.

**SPRING BOOT LOGGING**

While building applications, we often face errors which must be debugged. So, with the help of logs, we can easily get information about what is happening in the application with a record of errors and unusual circumstances.

Therefore, logs will be not be stored permanently, and are displayed one by one, as it is a single-threaded environment.

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

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

The Java logging components help the developer to create logs, pass the logs to the respective destination and maintain an proper format. The following are the three components:

* Loggers – Responsible for capturing log records and passing them to the corresponding Appender.
* Appenders or Handlers – They are responsible for recording log events to a destination. Appenders format events with the help of Layouts, before sending outputs.
* Layouts or Formatters – Responsible to determine how data looks when it appears in the log entry.

Loggers in Java are objects which trigger log events, they are created and are called in the code of the application, where they generate Log Events before passing them to the next component which is an Appender. You can use multiple loggers in a single class to respond to various events or use Loggers in a hierarchy. They are normally named using the hierarchical dot-separated namespace. Also, all the Logger names must be based on the class or the package name of the logged component.

Apart from this, each Logger keeps a track of the nearest existing ancestor in the [Logger](https://docs.oracle.com/javase/7/docs/api/java/util/logging/Logger.html) namespace and also has a “Level” associated with it.

The way to declare a Logger in spring is as the example below where a special class is defined to be a Controller:

@RestController

**public** **class** **LoggingController** { **Logger** logger = LoggerFactory.getLogger(LoggingController.class);

@RequestMapping("/") **public** String **index**() { logger.trace("A TRACE Message");

logger.debug("A DEBUG Message"); logger.info("An INFO Message");

logger.warn("A WARN Message");

logger.error("An ERROR Message");

**return** "Howdy! Check out the Logs to see the output...";

}

}

Spring Boot is a very helpful framework. It allows us to forget about most of the configuration settings, many of which it opinionatedly auto-tunes.

In the case of logging, the only mandatory dependency is Apache Commons Logging.

We need to import it only when using Spring 4.x ([Spring Boot 1.x](https://github.com/spring-projects/spring-boot/blob/1.5.x/spring-boot-dependencies/pom.xml#L154)) since it's provided by Spring Framework’s ***spring-jcl*** module in Spring 5 ([Spring Boot 2.x](https://github.com/spring-projects/spring-boot/blob/2.0.x/spring-boot-project/spring-boot-dependencies/pom.xml#L154)).

**We shouldn't worry about importing spring-jcl at all if we're using a Spring Boot Starter** (which we almost always are). That's because every starter, like our spring-boot-starter-web, depends on spring-boot-starter-logging, which already pulls in spring-jcl for us.

Spring Boot also **gives us access to a more fine-grained log level setting via environment variables.** There are several ways we can accomplish this.

If we're using Maven, we can **define our log settings via the command line**:

mvn spring-boot:run

-Dspring-boot.run.arguments=--logging.level.org.springframework=TRACE,--logging.level.com.baeldung=TRACE

If we want to change the verbosity permanently, we can do so in the application.properties file as described:

logging.level.root=WARN

logging.level.com.baeldung=TRACE

we can **change the logging level permanently by using our logging framework configuration file.**

Spring Boot Starter uses Logback by default. Let's see how to define a fragment of a Logback configuration file in which we set the level for two separate packages:

<**logger** name="org.springframework" level="INFO" />

<**logger** name="com.baeldung" level="INFO" />

**If the log level for a package is defined multiple times** using the different options mentioned above, but **with different log levels, the lowest level will be used.**

If we set the logging levels using Logback, Spring Boot, and environment variables at the same time, the log level will be *TRACE*, as it is the lowest among the requested levels.

*LOGBACK CONFIGURATION LOGGING*

Even though the default configuration is useful (for example, to get started in zero time during POCs or quick experiments), it's most likely not enough for our daily needs.

Let's see **how to include a Logback configuration** with a different color and logging pattern, with separate specifications for console and file output, and with a decent rolling policy to avoid generating huge log files.

First, we should find a solution that allows for handling our logging settings alone instead of polluting application.properties, which is commonly used for many other application settings.

**When a file in the classpath has one of the following names, Spring Boot will automatically load it** over the default configuration:

* logback-spring.xml
* logback.xml
* logback-spring.groovy
* logback.groovy

**Spring recommends using the -spring variant** over the plain ones whenever possible.

You can check some examples of this at this site:

[Logging in Spring Boot | Baeldung](https://www.baeldung.com/spring-boot-logging)

*LOG4J2 LOGGING*

While Apache Commons Logging is at the core, and Logback is the reference implementation provided, all the routings to the other logging libraries are already included to make it easy to switch to them.

**In order to use any logging library other than Logback, though, we need to exclude it from our dependencies.**

**We need to have something like this:**

<dependency>

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

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

<exclusions> <exclusion>

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

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

</exclusion>

</exclusions>

</dependency>

<dependency>

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

<artifactId>spring-boot-starter-log4j2</artifactId>

</dependency>

At this point, we need to place in the classpath a file named one of the following:

* log4j2-spring.xml
* log4j2.xml

We'll print through Log4j2 (over SLF4J) without further modifications.

To check the example and another ways for logging visit:

[Logging in Spring Boot | Baeldung](https://www.baeldung.com/spring-boot-logging)

In addition to the XML configuration, Log4j2 allows us to use also a YAML or JSON configuration, described [here](https://docs.spring.io/spring-boot/docs/current/reference/html/howto-logging.html#howto-configure-log4j-for-logging-yaml-or-json-config).

Some recommendations:

* Spring Boot also supports JDK logging, through the logging.properties configuration file.

There are cases when it's not a good idea to use it, though. From [the documentation](https://docs.spring.io/spring-boot/docs/current/reference/html/boot-features-logging.html#boot-features-custom-log-configuration):

There are known classloading issues with Java Util Logging that cause problems when running from an ‘executable jar'. We recommend that you avoid it when running from an ‘executable jar' if at all possible.

* It's also a good practice when using Spring 4 to manually exclude commons-logging in pom.xml, to avoid potential clashes between the logging libraries. Spring 5 instead handles it automatically, so we don't need to do anything when using Spring Boot 2.

**SPRING BOOT EXCEPTION HANDLING**

Spring boot can achieve this with some tools such as:

*Controller advice.*

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

*Exception Handler.*

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

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

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

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

*DEBUGGING IN SPRING BOOT*

Debugging is one of the most important tools for writing software.

To debug correctly it will depend on the environment and tools we use to develop our application, so I’ll give you some sources to visit them and watch the process that the one you need:

[Debugging Spring Applications | Baeldung](https://www.baeldung.com/spring-debugging)

[Debugging in Spring Boot | ShipTwo](https://shiptwo.github.io/spring-boot-debug/#:~:text=Debugging%20in%20Spring%20Boot%20Spring%20Boot%20is%20an,to%20run%20the%20apps%20like%20simple%20Java%20Programs.)

**SPRING DATA**

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

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

**SPRING DATA JPA & JDBC**

JPA is a Java standard that allows us to bind Java objects to records in a relational database.[It's one possible approach](https://en.wikipedia.org/wiki/List_of_object%E2%80%93relational_mapping_software#Java)**to Object Relationship Mapping(ORM)**, allowing the developer to retrieve, store, update, and delete data in a relational database using Java objects. Several implementations are available for the JPA specification.

Spring Data JPA, part of the larger Spring Data family, makes it easy to easily implement JPA based repositories. This module deals with enhanced support for JPA based data access layers. It makes it easier to build Spring-powered applications that use data access technologies.

Implementing a data access layer of an application has been cumbersome for quite a while. Too much boilerplate code has to be written to execute simple queries as well as perform pagination, and auditing. Spring Data JPA aims to significantly improve the implementation of data access layers by reducing the effort to the amount that’s actually needed. As a developer you write your repository interfaces, including custom finder methods, and Spring will provide the implementation automatically.

## Features

* Sophisticated support to build repositories based on Spring and JPA
* Support for [Querydsl](http://www.querydsl.com/) predicates and thus type-safe JPA queries
* Transparent auditing of domain class
* Pagination support, dynamic query execution, ability to integrate custom data access code
* Validation of @Query annotated queries at bootstrap time
* Support for XML based entity mapping
* JavaConfig based repository configuration by introducing @EnableJpaRepositories.

In terms of databases**, Spring Data JDBC** requires a [dialect](https://docs.spring.io/spring-data/jdbc/docs/current/reference/html/#jdbc.dialects) to abstract common SQL functionality over vendor-specific flavours. Spring Data JDBC includes direct support for the following databases:

* DB2
* H2
* HSQLDB
* MariaDB
* Microsoft SQL Server
* MySQL
* Oracle
* Postgres

Spring Data JDBC was created to fill a void that sits between Spring JDBC and Spring Data JPA. If you look at Spring JDBC, you could argue that it is too low level to work because it only helps with the connection to the database. Spring Data JPA could seem too complex because it gives you a lot of options and it can be difficult to master all these options. Spring Data JDBC is a framework that tries to give you the same power you get from using Spring Data JPA but makes it more understandable by using DDD principles. It also gives you more control by working on a lower level and by letting you decide when database interactions need to be done like Spring JDBC, but in an easier way.

Spring JDBC, Spring Data JPA and Spring Data JDBC are all three based on a different mindset. Spring JDBC only helps with the connection to the database and with executing queries on this database. Spring Data JPA wants to help you manage your JPA based repositories. It wants to remove boilerplate code, make implementation speed higher and provide help for the performance of your application. Spring Data JDBC also wants to provide help with access to relational databases but wants to keep the implementations less complex.

The help that Spring JDBC provides is by providing a framework to execute SQL. Spring JDBC handles the connection with the database and lets you execute queries using JdbcTemplates. This solution is very flexible because you have complete control over the executed queries. You are also free to define your class structure because you are in complete control of the mapping.

Spring Data JPA uses entities, so the class structure needs to be comparable with the database structure because some mapping is done automatically. In the simplest form, the database tables will each represent an entity and can be mapped almost directly on an entity class. This mapping can be done by using Java configuration. By using annotations, you can define on which table the class is mapped but also how the tables are linked together.

I recommend you visit my example folder of the subject to get a better understanding of this.

When you use Spring Data JDBC, you will also need to create entity classes which will be mapped to the database. The big difference is that there are more rules that you need to follow when you create the class structure. The class structure needs to follow the rules of aggregate design of DDD. Spring Data enforces this because this will lead to the creation of more simple and understandable projects.

Basically, we group different entities together which have a strong coupling and we call them aggregates. The top entity of the aggregate is called the aggregate root. There are some other rules that need to be followed:

* An entity can only be part of 1 aggregate.
* All relations inside an aggregate need to be unidirectional.
* The aggregate root needs to manage the top relation.

This means that by following links starting from the aggregate root, every entity inside the aggregate can be found. Because of this, we do not need a repository for each entity like in Spring Data JPA, but only for the aggregate roots.

A big difference in creating the classes used by Spring Data JDBC versus Spring Data JPA is that no @Entity and no relation annotations like @OneToMany need to be used. Spring Data JDBC knows a class is an aggregate root when it contains a repository for that class. And because of the rules that the aggregate entities are connected through object references, Spring Data JDBC also knows what the aggregates are and can transfer data to the database as aggregates.

*INSERTING DATA*

With Spring JDBC you write your insert statements yourself and execute them with a JdbcTemplate. The advantage of writing all the queries yourself is that you have complete control over them. This can be something like:

**JdbcTemplate** template = new **JdbcTemplate**(ds);

template.execute("create table car (id int, model varchar)");

template.execute("insert into car (id, model) values (1, 'Volkswagen Beetle')");

template.execute("insert into car (id, model) values (1, 'Volkswagen Beetle')");

If you use Spring Data JPA for inserting data, you will need to use the repositories and the entities. This makes it possible to think on a higher level and let Spring Data JPA handle the creation of queries. When you want to create data for an entity, the only thing you need to do is create an object with the correct values and call the save method on your Spring Data repository. Spring Data JPA will then look at your entities with all their annotations to map them to the necessary insert or update statements. You can see this through our example folder of this topic.

Spring Data JDBC uses a syntax that is comparable to Spring Data JPA. The biggest differences are under the hood. The management of the persistence is handled by the repository like in Spring Data JPA, but only the aggregate root has a repository. This means that if you want to insert or update data, the entire aggregate needs to be saved. You will need to call the save method of the repository of the aggregate root and this will first save the aggregate root and then all of the referenced entities get saved. If you want to insert only a part of an aggregate, the whole aggregate will be updated, and the referenced entities will be deleted and inserted again.

To retrieve data from our database, we write queries. Spring JDBC will let you use the JdbcTemplate and let you map the result with a RowMapper. Spring Data JDBC and Spring Data JPA will also let you create queries, using JPQL or SQL queries, but you will write them in the repositories and the frameworks will help you with the mapping.

The main tool that Spring JDBC uses for querying is the JdbcTemplate. The downside of using this is that it only provides the connection, everything else you need to do yourself. If you search for objects, you will need to map the results to Java objects by implementing a RowMapper. You will also need to do the exception handling by creating a ExceptionTranslator. An example for this is something like:

public class **CarRowMapper** implements **RowMapper**<**Car**> {

@Override

public **Car** mapRow(**ResultSet** resultSet, int rowNumber) throws **SQLException** {

**Car** car = new **Car**();

car.setId(resultSet.getInt("ID"));

car.setColor(resultSet.getString("COLOR"));

car.setBrand(resultSet.getString("BRAND"));

car.setModel(resultSet.getString("MODEL"));

return car;

}}

//////////////////////////////////////////////////////////////////////////////

**List**<**Car**> cars = jdbcTemplate.queryForObject(

"SELECT \* FROM CAR WHERE ID = ?", new **Object**[] {id}, new **CarRowMapper**());

When you use the Spring Data framework, it will help you with building your queries and fetching the right data. The Spring Data JPA framework uses implementations of the JPA specifications like Hibernate. They make it possible to query the database using user friendly interfaces. When you want to query the database, instead of writing the entire query yourself, Hibernate will help you. There are multiple ways to query the database using Spring Data JPA, but they all need you to extend the repository of the entity you want to query.

Some basic queries can be written using derived queries. An example of this is findById. For these methods Spring Data will generate the SQL entirely on its own.

If you need to write more advanced queries that can’t easily be defined as a derived query, you can define the query yourself using the @Query annotation. Inside the @Query annotation, you write JPQL or SQL statements. JPQL is an SQL-like syntax that provides an abstraction layer on top of regular SQL. When JPQL is used, it is possible for Spring Data to help you with handling the data. For example, paging and sorting can be done by simply adding a parameter.

If you want to have a bit more control, you can use SQL by setting the native property of the @Query annotation to true. Then you don’t use this extra layer, but it then also can’t help you anymore. Be aware that even though you use SQL directly, you will still return entities that are managed by Hibernate which is a java framework and ORM (Object Relation Mapping) tool that is used to provide the implementation of the JPA methods.

You are using entities when you query using Spring Data JPA. These entities have connections to other entities. Spring Data JPA will help you with defining whether you want to return these connected entities directly or not. It can help you with searching for these entities when you do need them. This is called eager and lazy loading, and this can all be managed by Spring Data JPA.

It will also try to improve performance by giving you the option to turn on the query cache. When this is turned on, Spring Data JPA will try to reuse the generated SQL queries, and if possible, the results of these queries.

Spring Data JDBC has less abstractions than Spring Data JPA but uses Spring Data concepts to make it easier to do CRUD operations than Spring JDBC.

It sits closer to the database because it does not contain the most part of the Spring Data magic when querying the database. Every query is executed directly on the JDBC and there is no lazy loading or caching, so when you query the database, it will return the entire aggregate.

An important difference is that we cannot use derived queries, so we need to use the @Query annotation. To do this we need to implement something like this:

public interface **RentalCompanyRepository** extends **CrudRepository**<**RentalCompany**, **Long**> {

@Query(value = "SELECT \* " +

"FROM Rental rental " +

"JOIN Car car ON car.id = rental.car\_id " +

"WHERE rental.rental\_company = :companyId " +

"AND car.type = :carType")

**List**<**Rental**> findRentalsByIdAndCarType(@Param("companyId") **Long** companyId, @Param("carType")**String** carType);

}

Spring JDBC again only provides a framework when updating data from the database. The JdbcTemplate exposes an update method. This method can accept a query and optional parameters like in this code:

**String** query = "update Car set color = ? where id = ?";

jdbcTemplateObject.update(query, color, id);

Spring Data JPA provides more tools to update the data, like the proxy around the entities. The state of these entities is stored in a persistence context. By using this, Spring Data JPA can keep track of the changes to these entities. It uses the information of these changes to keep the database up to date. Spring Data JPA makes managed entities from these entities. Instead of always needing to create queries to update data in the database, we can edit these entities. These changes will then always be persisted automatically. This tracking is called dirty tracking because when you change the entities, these updates are making the entity “dirty” since the state is different than in the database. When the Hibernate session is flushed, these changes will be persisted and the entity will be “clean” again. This will only be done for changes within a transaction. If the changes are not done within a transactional context, you will have to call the save method of the repository to persist those changes.

If you want to make bigger changes, it is also possible to create update methods in the repositories. Like querying the database and creating entities, you can also create methods in the repository. Using JPQL you can create a query which can update multiple entities at once. Please make sure to add the @Modifying annotation. This is a security measure so you cannot modify something by mistake.

*SUMMARY*

One of the biggest advantages that you can get from using Spring Data JDBC is that it will force you to follow the rules of DDD design like using aggregates.

Because only one-to-many relationships are used, it makes it easier to see what the exact relationships are between the classes. Classes can only be part of 1 aggregate. Together with the one-to-many rule this makes it impossible to create circular dependencies. If you need to create relationships between aggregates, you need to use id’s. This makes the coupling between the aggregates as small as possible.

It is also clear where the logic of the interactions with the data can be found because only the aggregate roots have repositories because they are responsible for these interactions.

Only the aggregate roots are responsible for handling the persistence. This makes it clear what needs to be persisted and who is responsible for doing this. Because the persistence always needs to be initiated by calling the save method, it makes it easier to understand when changes will be persisted than with Spring Data JPA.

It is easy to see what classes are part of an aggregate since aggregates are connected using object references. When id’s are used, those classes are part of different aggregates. When you query the database, instead of the lazy loading which is standard in Spring Data JPA, every call using Spring Data JDBC is done using eager loading. Every time you need data, a call to the database will be done because no caches are used. Together these rules make sure that it is easy to know when a call to the database will be done (always), what parts of the data are returned from the database calls (entire aggregate) and it is easy to know what these aggregates are composed of.

Because you are responsible for saving when something needs to be saved, and when you do a call through a repository, the entire aggregate is returned. The result of this is that you need to do a little bit more yourself, but it also gives you complete control of the entire data flow.

With Spring Data JDBC you have a little more control which query will be executed on the database since it is executed directly on the JDBC instead of going through a middle layer. All the queries are eager, this is also an advantage because less queries need to be sent to the database.

When you create or update entities in an aggregate through Spring Data JDBC, it will do this by deleting and again saving these entities. Spring Data JDBC needs to do this since it does not have a persistence context and wants to make sure that everything is up to date. The downside of this is that it is possible that sometimes unnecessary operations will be executed.

With Spring Data JPA you have more possibilities for fine tuning performance. For example, with the possibility of using the lazy loading and the usage of a cache. Because of these possibilities it is also more difficult to create a configuration for a good performance.

**H2 with Spring Boot**

H2 database is an open source, embedded and in memory relational database management system. It is written in Java and provides a client/server application. It stores data in system memory instead of disk. Once program is closed, data is also lost. An in memory database is used when we don't want to persist the data and unit test the overall functionality. Some of the other popular in memory databases are HSQLDB or HyperSQL Database and Apache Derby. H2 is the most popular one among other embedded databases.

Some of the advantages are:

No configuration − Spring Boot intrinsically supports H2 and no extra configuration required to configure H2 database.

* Easy to Use − H2 Database is very easy to use.
* Lightweight and Fast − H2 database is very lightweight and being in memory, it is very fast.
* Switch configurations − Using profiles, you can easily switch between production level database and in-memory database.
* Supports Standard SQL and JDBC − H2 database supports almost all the features of Standard SQL and operations of JDBC.
* Web Based Console − H2 Database can be managed by its web based console application.

To implement this we need to add the dependencies to our project and configure the application properties as below (accordingly with the project we develop)

spring.datasource.url=jdbc:h2:mem:testdb

spring.datasource.driverClassName=org.h2.Driver

spring.datasource.username=sa

spring.datasource.password=

spring.jpa.database-platform=org.hibernate.dialect.H2Dialect

spring.h2.console.enabled=true

If persistent storage is needed than add the following configuration in application.properties.

spring.datasource.url=jdbc:h2:file:/data/database

spring.datasource.url=jdbc:h2:C:/data/database

For more details visit:   
[Spring Data JPA - Reference Documentation](https://docs.spring.io/spring-data/jpa/docs/1.11.1.RELEASE/reference/html/#jpa.query-methods.at-query)

[An Introduction to Spring Data JDBC - Wout Meskens — Ordina JWorks Tech Blog (ordina-jworks.github.io)](https://ordina-jworks.github.io/java/2020/01/02/Spring-Data-Jdbc.html#:~:text=A%20big%20difference%20in%20creating%20the%20classes%20used,when%20it%20contains%20a%20repository%20for%20that%20class.)

[Spring Boot with Hibernate | Baeldung](https://www.baeldung.com/spring-boot-hibernate)

[Spring Boot & H2 - Overview (tutorialspoint.com)](https://www.tutorialspoint.com/spring_boot_h2/spring_boot_h2_overview.htm)

**SRPING CLOUD**

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

A picture containing graphical user interface

Description automatically generated

**SPRING CLOUD CONFIG**

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

Spring Cloud Config Server features:

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

Config Client features (for Spring applications):

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

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

**SPRING CLOUD CONFIG SERVER**

Spring Cloud Config Server provides an HTTP resource-based API for external configuration (name-value pairs or equivalent YAML content). The server is embeddable in a Spring Boot application, by using the @EnableConfigServer annotation. Consequently, the following application is a config server:

Like all Spring Boot applications, it runs on port 8080 by default, but you can switch it to the more conventional port 8888 in various ways. The easiest, which also sets a default configuration repository, is by launching it with spring.config.name=configserver (use your own application.properties).

To get a better understanding of this visit in this repository: **Examples/SpringCloudConfigServerExample**

[*ENVIRONMENT REPOSITORY*](https://docs.spring.io/spring-cloud-config/docs/current/reference/html/#_environment_repository)

Where should you store the configuration data for the Config Server? The strategy that governs this behaviour is the EnvironmentRepository, serving Environment objects. This Environment is a shallow copy of the domain from the Spring Environment (including propertySources as the main feature). The Environment resources are parametrized by three variables:

* {application}, which maps to spring.application.name on the client side.
* {profile}, which maps to spring.profiles.active on the client (comma-separated list).
* {label}, which is a server side feature labelling a "versioned" set of config files.

Repository implementations generally behave like a Spring Boot application, loading configuration files from a spring.config.name equal to the {application} parameter, and spring.profiles.active equal to the {profiles} parameter.

Precedence rules for profiles are also the same as in a regular Spring Boot application: Active profiles take precedence over defaults, and, if there are multiple profiles, the last one wins (like adding entries to a Map).

(As usual with a Spring Boot application, these properties could also be set by environment variables or command line arguments).

You can set spring.cloud.config.server.accept-empty to false so that Server would return a HTTP 404 status, if the application is not found.By default, this flag is set to true.

For more details visit:

[Spring Cloud Config](https://docs.spring.io/spring-cloud-config/docs/current/reference/html/#_spring_cloud_config_server)

**NETFLIX EUREKA**

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

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

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

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

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

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

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

**SPRING CLOUD API GATEWAY**

This project provides an API Gateway built on top of the Spring Ecosystem, including: Spring 5, Spring Boot 2 and Project Reactor. Spring Cloud Gateway aims to provide a simple, yet effective way to route to APIs and provide cross cutting concerns to them such as: security, monitoring/metrics, and resiliency.

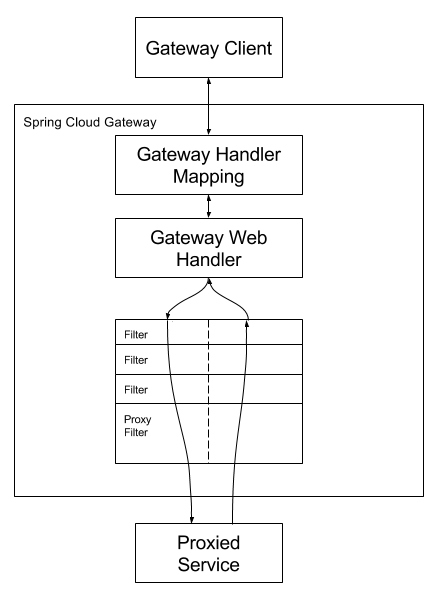
To include Spring Cloud Gateway in your project, use the starter with a group ID of org.springframework.cloud and an artifact ID of spring-cloud-starter-gateway. See the [Spring Cloud Project page](https://projects.spring.io/spring-cloud/) for details on setting up your build system with the current Spring Cloud Release Train.

If you include the starter, but you do not want the gateway to be enabled, set spring.cloud.gateway.enabled=false.

Important terms related:

* **Route**: The basic building block of the gateway. It is defined by an ID, a destination URI, a collection of predicates, and a collection of filters. A route is matched if the aggregate predicate is true.
* **Predicate**: This is a [Java 8 Function Predicate](https://docs.oracle.com/javase/8/docs/api/java/util/function/Predicate.html). The input type is a [Spring Framework ServerWebExchange](https://docs.spring.io/spring/docs/5.0.x/javadoc-api/org/springframework/web/server/ServerWebExchange.html). This lets you match on anything from the HTTP request, such as headers or parameters.
* **Filter**: These are instances of [GatewayFilter](https://github.com/spring-cloud/spring-cloud-gateway/tree/main/spring-cloud-gateway-server/src/main/java/org/springframework/cloud/gateway/filter/GatewayFilter.java) that have been constructed with a specific factory. Here, you can modify requests and responses before or after sending the downstream request.

How it works:



Clients make requests to Spring Cloud Gateway. If the Gateway Handler Mapping determines that a request matches a route, it is sent to the Gateway Web Handler. This handler runs the request through a filter chain that is specific to the request. The reason the filters are divided by the dotted line is that filters can run logic both before and after the proxy request is sent. All “pre” filter logic is executed. Then the proxy request is made. After the proxy request is made, the “post” filter logic is run.

*GATEWAYFILTER FACTORIES*

Route filters allow the modification of the incoming HTTP request or outgoing HTTP response in some manner. Route filters are scoped to a particular route. Spring Cloud Gateway includes many built-in GatewayFilter Factories.

*GLOBAL FILTERS*

The GlobalFilter interface has the same signature as GatewayFilter. These are special filters that are conditionally applied to all routes.

To get a better understanding of how API Gateway works check the Example of this repository at:

**Examples/API Gateway Example/api-gateway.zip**

For full details visit:

[Spring Cloud Gateway](https://docs.spring.io/spring-cloud-gateway/docs/current/reference/html/#gateway-starter)