Multi Database Connections with Spring Boot

Ritesh Shergill · Follow

7 min read · Jun 16, 2023

Listen Share More

Any Java developer worth their salt would have used **Spring Boot** for rapid development at some point or the other during the course of their careers.

If you haven't, you are missing out. Spring Boot is one the most **capable, robust and flexible** Java frameworks out there and it makes enterprise application development

as easy as mixing ice in water.

Spring Boot provides a nifty package called **Data JPA** that allows you to connect to RDMS using ORM like interfaces. It's easy enough to use and implement and all it takes is an entry in the pom.xml (If you are using Maven)

```
<dependencies>
        <!-- Spring boot dependencies -->
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter</artifactId>
        </dependency>
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-data-jpa</artifactId>
        </dependency>
</dependencies>
```

And in the Main Spring Application class, 2 annotations that allow spring Boot to configure database connections

```
@SpringBootApplication
@EnableJpaRepositories
@EnableAutoConfiguration
public class SpringMainApplication {

    public static void main(String[] args) {
        SpringApplication.run(SpringMainApplication.class, args);
    }
}
```

Add a repository package with repository connection files, some settings in the properties file, entities mapped to your database and that's it — you are good to go. Your application is ready to communicate with the database.

But, there is one more feature in Spring Boot that I consider way more awesome than connecting to a database.

That is..

Connecting to multiple databases

You heard that right. You can connect to multiple databases from a single Spring Boot app.

> *Caveat: The multiple databases should have the same driver though. This means that you cannot connect to a MySql and Postgres SQL database. It has to be 2 MySql databases. Also, the database Schemas have to be the same. You cannot connect to 2 databases with different schemas.*

But why would you even **connect to multiple similar databases from a single Spring Boot app**?

1️⃣ To support **multitenancy** within the same app with the same schemas.

2️⃣ To **dynamically simulate behavior on multiple environment databases without requiring an application restart** — For eg, you can dynamically connect to your dev database or your QA database without requiring an application restart

3️⃣ To **support multiple play ground databases to simulate various automation testing scenarios.** One database could have very different configuration and static information from another and this could mean that you can cover several test cases from one automation test script.

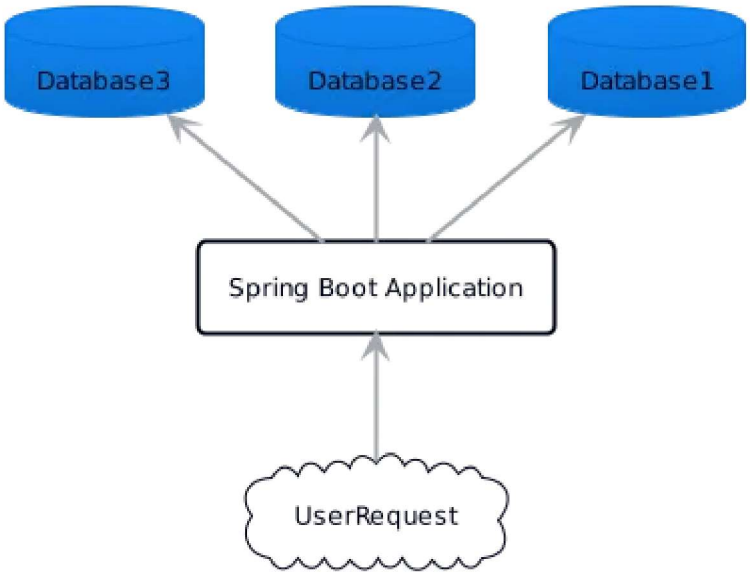
4️⃣ To **support multiple organizations within the same app.** Based on the user login, you can dynamically decide which organization's database their data should go into and which database would store the more common information

5️⃣ **Seeding multiple databases with data in one go** — For eg, you have a batch job that creates data from scripts. You can connect to multiple databases in one go and run the scripts against all those databases without the hassle of pointing to different applications or restarting the server to do so.

So, as you can see, there could be several use cases to support multi database connections.

> *Of course, these are very specific use cases, and if you truly want to support multiple databases, it is advisable to create an application layer over each DB and expose CRUD endpoints through application Gateways.*

Conceptually, this is what it looks like



A User Request containing the payload as well as the Database identification String is submitted to the Rest Endpoint. Based on the DB identification String, Spring Boot decided which database to point the request towards.

Right, enough Theory, lets see code!

First, we **setup a Spring Boot app** with standard packages. For this particular example, I am using

📌 *Intellij Idea Community edition as IDE*

📌 *Maven as Build tool*

📌 *PostgresSQL for my databases*

📌 *Spring Data JPA for establishing database connections*

This is what the **pom file** would look like:

```
<dependencies>
        <!-- Spring boot dependencies -->
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter</artifactId>
        </dependency>
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-actuator</artifactId>
```

```
        </dependency>
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-devtools</artifactId>
        </dependency>
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-data-jpa</artifactId>
        </dependency>
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-web</artifactId>
        </dependency>

        <!-- Swagger dependencies -->
        <dependency>
            <groupId>io.springfox</groupId>
            <artifactId>springfox-swagger2</artifactId>
            <version>2.9.2</version>
        </dependency>
        <dependency>
            <groupId>io.springfox</groupId>
            <artifactId>springfox-swagger-ui</artifactId>
            <version>2.9.2</version>
        </dependency>

        <!-- lombok dependency -->
        <dependency>
            <groupId>org.projectlombok</groupId>
            <artifactId>lombok</artifactId>
            <version>1.18.12</version>
            <scope>provided</scope>
        </dependency>

        <!-- Database dependency -->
        <dependency>
            <groupId>org.postgresql</groupId>
            <artifactId>postgresql</artifactId>
        </dependency>


        <!-- test dependencies -->
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-test</artifactId>
            <scope>test</scope>
            <exclusions>
                <exclusion>
                    <groupId>org.junit.vintage</groupId>
                    <artifactId>junit-vintage-engine</artifactId>
                </exclusion>
            </exclusions>
        </dependency>
```

```
        </dependencies>

        <dependencyManagement>
            <dependencies>
                <dependency>
                    <groupId>org.springframework.cloud</groupId>
                    <artifactId>spring-cloud-starter-parent</artifactId>
                    <version>${spring-cloud-dependencies.version}</version>
                    <type>pom</type>
                    <scope>import</scope>
                </dependency>
                <dependency>
                    <groupId>org.springframework.cloud</groupId>
                    <artifactId>spring-cloud-gcp-dependencies</artifactId>
                    <version>${project.version}</version>
                    <type>pom</type>
                    <scope>import</scope>
                </dependency>
            </dependencies>
        </dependencyManagement>
        <build>
            <plugins>
                <plugin>
                    <groupId>org.springframework.boot</groupId>
                    <artifactId>spring-boot-maven-plugin</artifactId>
                </plugin>
                <plugin>
                    <groupId>org.apache.maven.plugins</groupId>
                    <artifactId>maven-surefire-plugin</artifactId>
                </plugin>
            </plugins>
        </build>
```

Apart from the Data JPA dependencies, we have

➡️ **Actuator** to let you monitor and interact with your application, as well check the health status of the application

➡️ **Devtools** to have an embedded LiveReload server that can be used to trigger a browser refresh when a resource is changed

➡️ **Starter Web** to expose Rest Endpoints for the application

➡️ **Swagger** to visualize the API contracts from a web page and invoke the APIs as well

➡️ **Lombok** to replace boiler plate code like Getters Setters in classes with Nifty annotations

➡️ And finally, the **Postgres SQL dependency** jar

Next, we visit our **application.properties** file to add the multiple database connection configurations

```
app.datasource.db1.jdbc-url=jdbc:postgresql://db1.com:5432/dbname1
app.datasource.db1.username=postgres
app.datasource.db1.password=password

app.datasource.db2.jdbc-url=jdbc:postgresql://db2.com:5432/dbname2
app.datasource.db2.username=postgres
app.datasource.db2.password=password

app.datasource.db3.jdbc-url=jdbc:postgresql://db3.com:5432/dbname3
app.datasource.db3.username=postgres
app.datasource.db3.password=password
```

These are 3 seperate PostgresSQL instances with the same schemas but different data.

The next step is to add the **Multidatabase configuration**. First, we add appropriate Annotations in the Spring Application Main file

```
@SpringBootApplication
@EnableJpaRepositories
@EnableAutoConfiguration
public class MultidatabaseApplication {

    public static void main(String[] args) {
        SpringApplication.run(MultidatabaseApplication.class, args);
    }
}
```

This will tell Spring boot to use Spring Data JPA *@Repositories* to perform CRUD operations.

Next, we setup the **Application Config** for Multi-Database connections

```
@Configuration
@EnableTransactionManagement
@EnableJpaRepositories(
        entityManagerFactoryRef = "multiEntityManager",
        transactionManagerRef = "multiTransactionManager")
@EntityScan("com.sample.client.repositories.dto.entity")
public class DatabaseConfiguration {
    //add JPA entities path here
    private final String PACKAGE_SCAN = "com.sample.client.repositories.dto.ent

    //set db1 as the primary and default database connection
    @Primary
    @Bean(name = "db1DataSource")
    @ConfigurationProperties("app.datasource.db1")
    public DataSource db1DataSource() {
        return DataSourceBuilder.create().type(HikariDataSource.class).build();
    }

    //connection objects for the remaining 2 databases
    @Bean(name = "db2DataSource")
    @ConfigurationProperties("app.datasource.db2")
    public DataSource db2DataSource() {
        return DataSourceBuilder.create().type(HikariDataSource.class).build();
    }
    @Bean(name = "db3DataSource")
    @ConfigurationProperties("app.datasource.db3")
    public DataSource db3DataSource() {
        return DataSourceBuilder.create().type(HikariDataSource.class).build();
    }

    //The multidatasource configuration
    @Bean(name = "multiRoutingDataSource")
    public DataSource multiRoutingDataSource() {
        Map<Object, Object> targetDataSources = new HashMap<>();
        targetDataSources.put(ClientNames.DB1, db1DataSource());
        targetDataSources.put(ClientNames.DB2, db2DataSource());
        targetDataSources.put(ClientNames.DB3, db3DataSource());
        MultiRoutingDataSource multiRoutingDataSource
            = new MultiRoutingDataSource();
        multiRoutingDataSource.setDefaultTargetDataSource(db1DataSource());
        multiRoutingDataSource.setTargetDataSources(targetDataSources);
        return multiRoutingDataSource;
    }

    //add multi entity configuration code
    @Bean(name = "multiEntityManager")
    public LocalContainerEntityManagerFactoryBean multiEntityManager() {
        LocalContainerEntityManagerFactoryBean em
            = new LocalContainerEntityManagerFactoryBean();
        em.setDataSource(multiRoutingDataSource());
        em.setPackagesToScan(PACKAGE_SCAN);
```

```
            HibernateJpaVendorAdapter vendorAdapter
                = new HibernateJpaVendorAdapter();
            em.setJpaVendorAdapter(vendorAdapter);
            em.setJpaProperties(hibernateProperties());
            return em;
        }

        @Bean(name = "multiTransactionManager")
        public PlatformTransactionManager multiTransactionManager() {
            JpaTransactionManager transactionManager
                    = new JpaTransactionManager();
            transactionManager.setEntityManagerFactory(
                    multiEntityManager().getObject());
            return transactionManager;
        }

        @Primary
        @Bean(name="entityManagerFactory")
        public LocalSessionFactoryBean dbSessionFactory() {
            LocalSessionFactoryBean sessionFactoryBean = new LocalSessionFactoryBea
            sessionFactoryBean.setDataSource(multiRoutingDataSource());
            sessionFactoryBean.setPackagesToScan(PACKAGE_SCAN);
            sessionFactoryBean.setHibernateProperties(hibernateProperties());
            return sessionFactoryBean;
        }

        //add hibernate properties
        private Properties hibernateProperties() {
            Properties properties = new Properties();
            properties.put("hibernate.show_sql", true);
            properties.put("hibernate.format_sql", true);
            properties.put("hibernate.dialect", "org.hibernate.dialect.PostgreSQLDi
            properties.put("hibernate.id.new_generator_mappings", false);
            properties.put("hibernate.jdbc.lob.non_contextual_creation", true);
            return properties;
        }
    }
```

This completes our multi database configuration.

com.sample.client.repositories.dto.entity — This directory contains the JPA entities common to all 3 databases

> *The **MultiRoutingDataSource class** is our actual implementation that allows Spring Boot to work its magic and allow connections to multiple databases*

We also need a **DBContextHolder** class to hold the Database reference and dynamically change it at runtime.

```
public class DBContextHolder {
    private static final ThreadLocal<ClientNames> contextHolder = new ThreadLoc
    public static void setCurrentDb(ClientNames dbType) {
        contextHolder.set(dbType);
    }
    public static ClientNames getCurrentDb() {
        return contextHolder.get();
    }
    public static void clear() {
        contextHolder.remove();
    }
}
```

As well as a **ClientNames Enum** for the database names

```
public enum ClientNames {
    DB1, DB2, DB3
}
```

Next, we need the **MultiRoutingDataSource class**

```
public class MultiRoutingDataSource extends AbstractRoutingDataSource {
    @Override
    protected Object determineCurrentLookupKey() {
        return DBContextHolder.getCurrentDb();
    }
}
```

Which **decides** which database the application **should connect to dynamically.**

Right, that completes our configuration. Next, we will expose an endpoint to check whether our Multi-Database configuration works or not

```
@RestController
@RequestMapping("/client")
public class ClientDataController {

    @Autowired
    private ClientMasterService clientMasterService;

    @GetMapping("/{clientdb}")
    public String findFromDatabase(@PathVariable String clientdbName) {
        return clientMasterService.getClientNames(clientdbName);
    }
}
```

This Controller exposes an endpoint that accepts the database name and allows the ClientMasterService to decide which database to connect to and serve the information from.

This is the **ClientMasterService** code

```
@Service
public class ClientMasterService {

    @Autowired
    private ClientMasterRepository clientMasterRepository;

    public String getClientNames(String client) {
        switch (client) {
            case "db1":
                DBContextHolder.setCurrentDb(ClientNames.DB1);
                break;
            case "db2":
                DBContextHolder.setCurrentDb(ClientNames.DB2);
                break;
            case "db3":
                DBContextHolder.setCurrentDb(ClientNames.DB3);
                break;
        }
        Entity1 e1 = clientMasterRepository.findByEntity1Name("John Doe");
        if(e1 != null) {
            return "found in database: " + client + " with id " + e1.getId();
        }
        return "found in " + client + " nada!";
```

```
        }
    }
```

This service uses the **DBContextHolder** class to set the correct Database to point to based on what database name was passed in from the Rest Endpoint — db1, db2 or db3

And finally, the Repository to fetch data from the database

```
@Repository
public interface ClientMasterRepository extends JpaRepository<Entity1, String>
    Entity1 findByEntity1Name(String name);
}
```

And the Entity1 class

```
@Entity
@Table(name = "entity1")
@Getter
@Setter
public class Entity1 implements Serializable {
  @Id
  @Column(name = "id", nullable = false)
  private Integer id;

  @Column(name = "entity1Name")
  private String entity1Name;
}
```

And this completes our Multi-Database setup.

. . .

Such a configuration is very useful to **setup test beds and sandbox environments.** There are certain limitations though apart from the ones we mentioned above and

those would be

1. The **Context Switching** must be done in a more **controlled manner** to make sure there are no **race conditions** and data isn't read from the wrong database
2. **Too many database connections** might **slow down** the application so memory does become a constraint
3. It becomes harder to track **which Service is written for which database** as different services **might do different things with the different instances.** So coding for a Multi-Database must be done **conscientously to avoid Spaghetti code**

But overall the pros outweigh the cons as long as the use case is specific and clear

> And there we have it — One Application Many databases — One-for-all and All-for-one

. . .

Follow me Ritesh Shergill for more content on system design, coding practices career guidance and Agile.

Spring Boot | Database | Jpa

Follow

Written by Ritesh Shergill

228 Followers

Cybersec and Software Architecture Consultations | Career Guidance | Ex Vice President at JP Morgan Chase | Startup Mentor | Angel Investor | Author

More from Ritesh Shergill

Ritesh Shergill

Dynamic Task scheduling with Spring Boot

How many times have we had the requirement to schedule tasks ahead of time in our projects? Sometimes it seems like a major headache to...

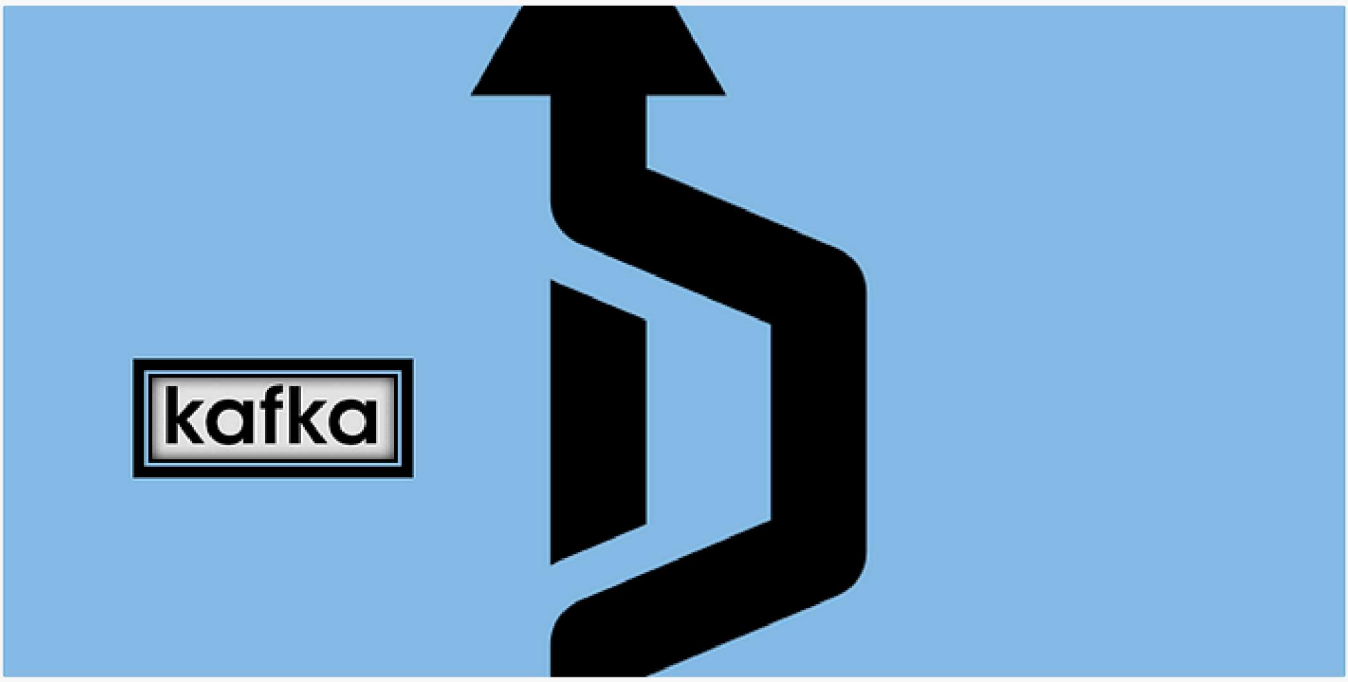
Mar 17, 2021 199 16

Ritesh Shergill

Kolmogorov-Arnold Networks with Gaussian Functions

I replaced BSpline activation functions with Gaussian Functions in the PyKan implementation of Kolmogorov-Arnold Networks.

Jun 6 2 1



Ritesh Shergill

Kafka: Decoupling Circuit breaker and Exponential Back off

When it comes to building resilient systems, you must never assume that the systems your system will rely on will be available 100 percent...

Aug 22, 2023 27

Ritesh Shergill

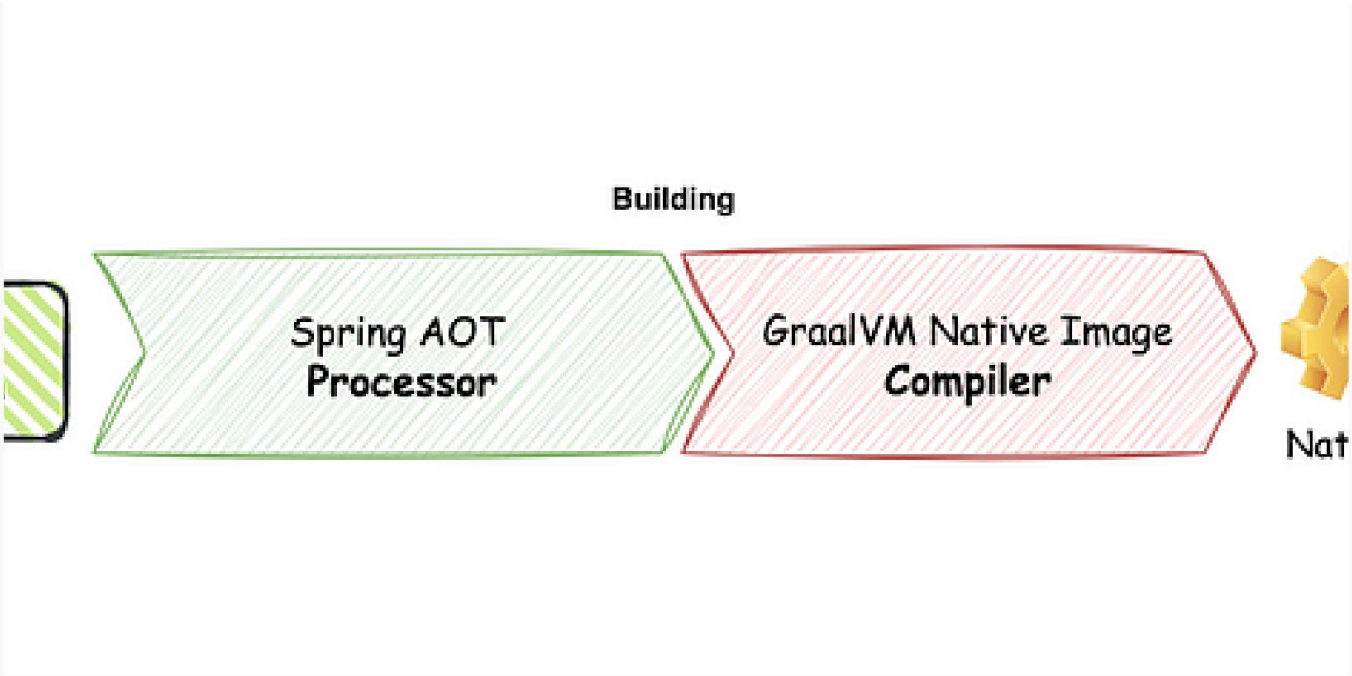
🔍 Build a Search Ecosystem with Vector Databases and LLMs 🔎

Searching for information is probably 80 Percent of the job in IT.

Jun 18 8

See all from Ritesh Shergill

Recommended from Medium



Saeed Zarinfam in ITNEXT

10 Spring Boot Performance Best Practices

Making Spring Boot applications performant and resource-efficient

Jun 24 202 3

Oliver Foster

Spring Boot: How Many Requests Can Spring Boot Handle Simultaneously?

My article is open to everyone; non-member readers can click this link to read the full text.

Jun 17 840 12

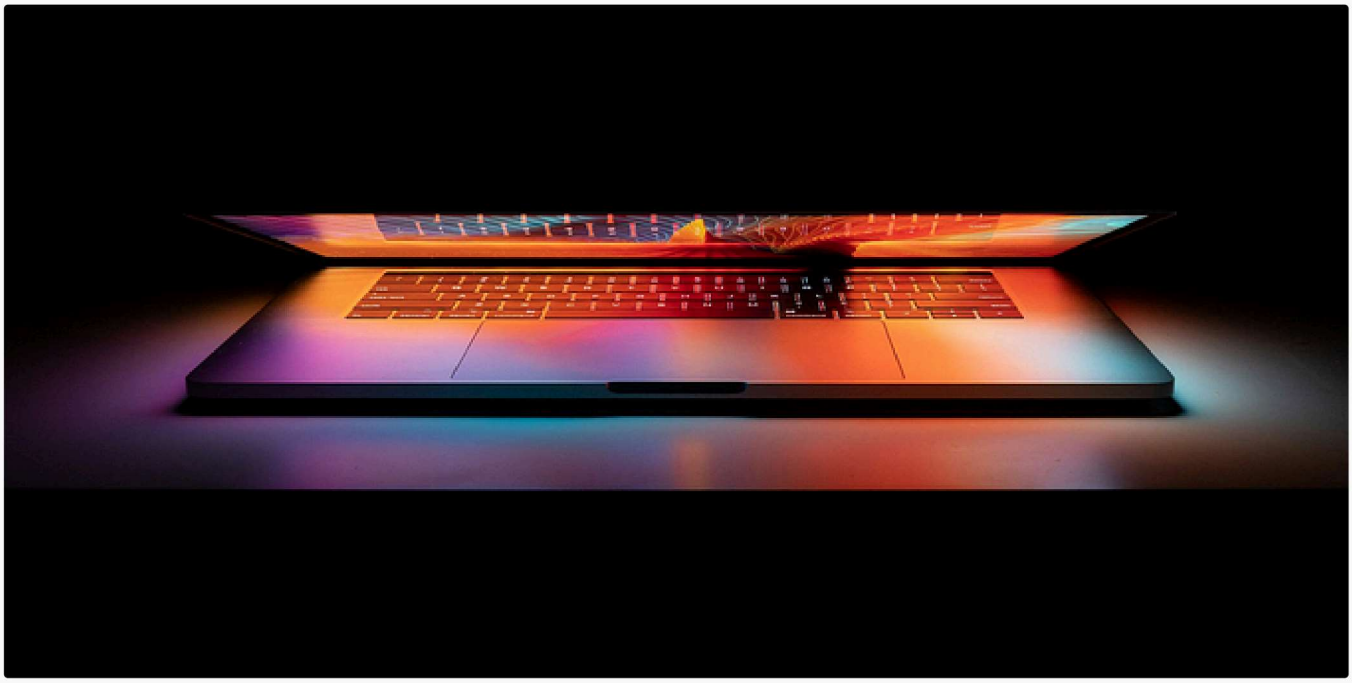
Lists

data science and AI

40 stories · 198 saves

Natural Language Processing

1567 stories · 1115 saves

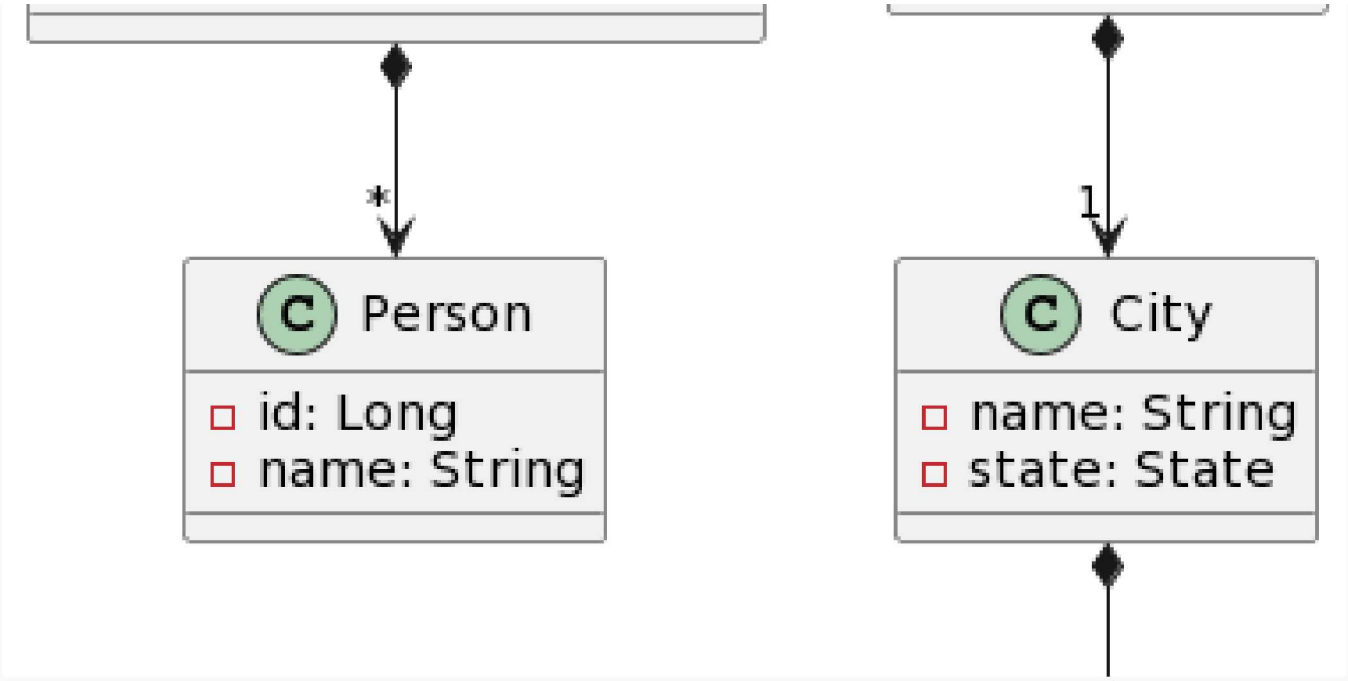


Mikael Svens

Why You Should Stop Using @Value Annotations In Spring (And Use This Instead)

If you have been working with Java and Spring Boot, I'm sure you have come across the @Value annotation. I'm here to show you an...

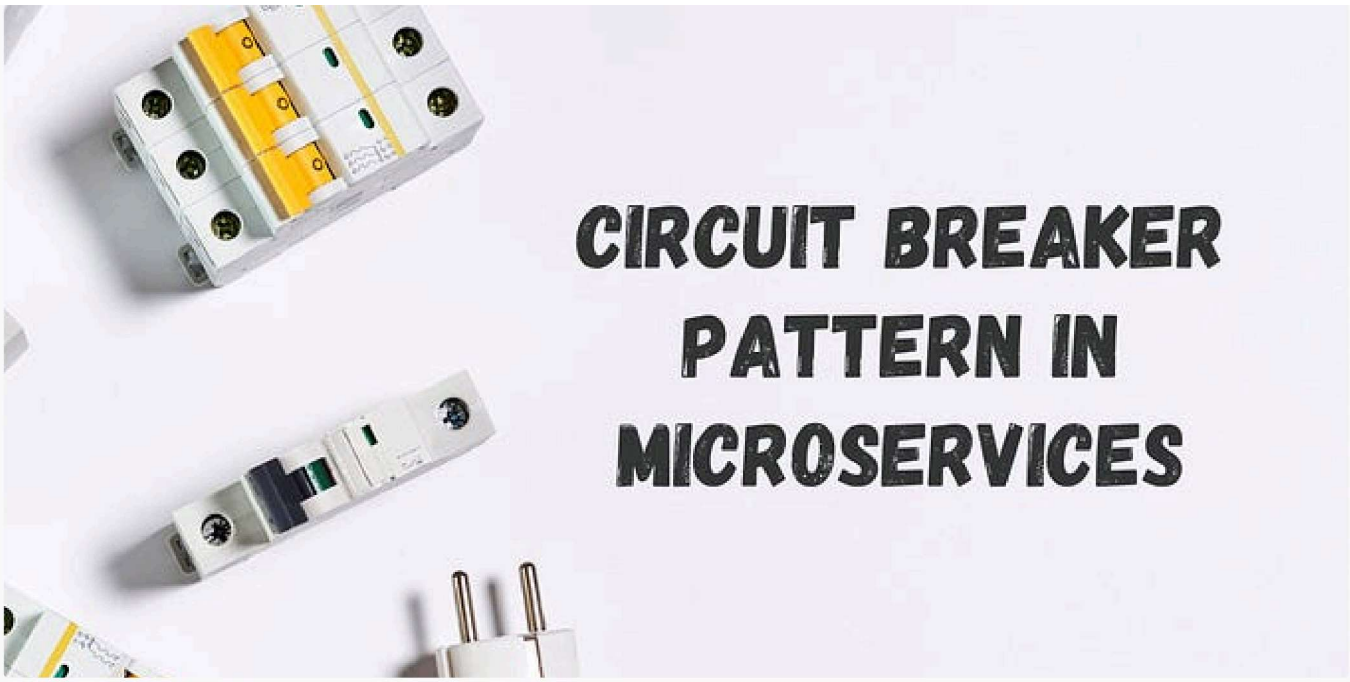
Mar 20 1.2K 27



Renan Schmitt in JavaJams

7 Tricks of Java Streams

Jan 16 1.1K 14



Chameera Dulanga in Bits and Pieces

Circuit Breaker Pattern in Microservices

How to Use the Circuit Breaker Software Design Pattern to Build Microservices

Jan 11, 2023 722 4

Vasko Jovanoski

Say Goodbye to Repetition: Building a Common Library in Spring Boot

Learn to create a shared library in Spring Boot, enhancing code reusability, simplifying maintenance, and reducing bugs in your...