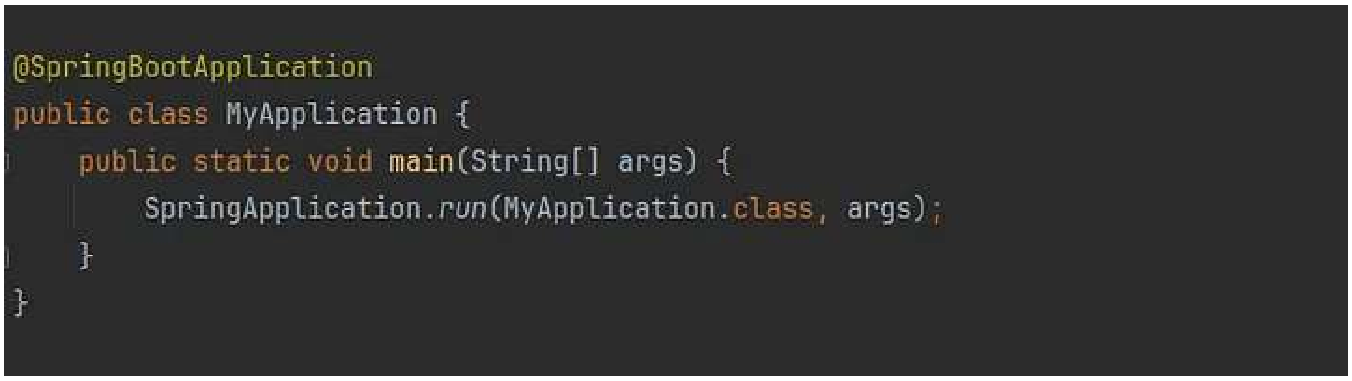
Spring Boot Security: Managing Multiple JWT Issuers

Sargis Vardanyan · Follow
Published in Octa Labs Insights
4 min read · May 2, 2024

Spring Boot is renowned for its simplicity and power in creating robust web applications. A standout component of Spring Boot is Spring Security, which simplifies security management by abstracting away complexity. This allows developers to focus on building applications. Spring Security offers easy ways to configure complex authentication and authorization mechanisms. In this article, we'll explore how Spring Security's flexibility allows developers to effortlessly configure multiple OAuth JWT issuers.

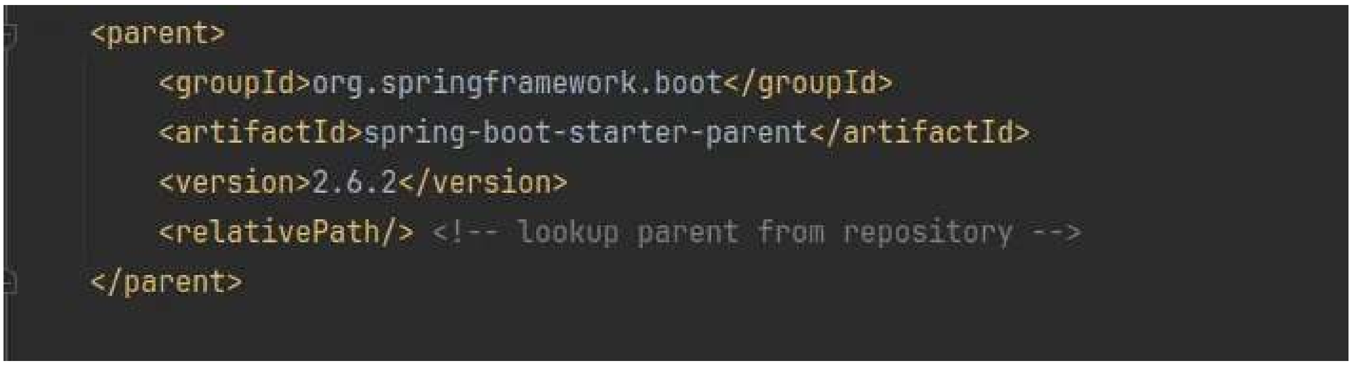
Basic configuration

To begin, let’s look at the basic configuration of a Spring Boot application. In a typical setup, you would have an application class annotated with `@org.springframework.boot.autoconfigure.SpringBootApplication` and a `main` method to bootstrap the application.



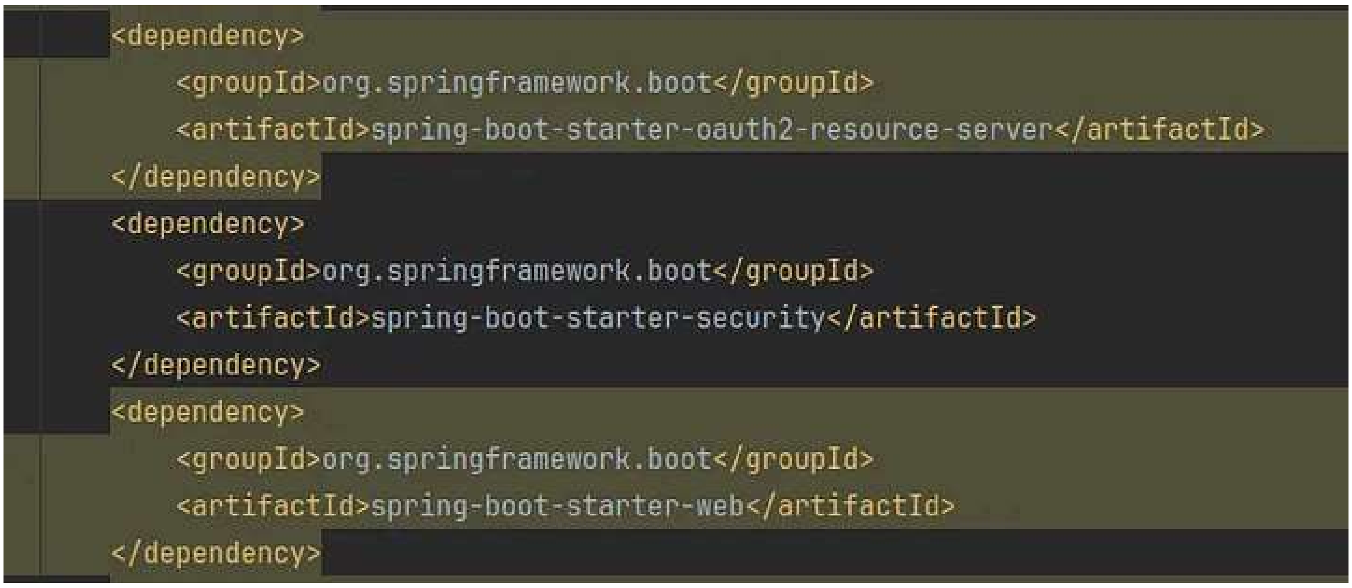
```java
@SpringBootApplication
public class MyApplication {
    public static void main(String[] args) {
        SpringApplication.run(MyApplication.class, args);
    }
}
```

For this tutorial, we are using Spring Boot version `2.6.2`. Our build tool of choice is Maven, and we are using the `spring-boot-maven-plugin` to build our application. If you're working with a multi-module application, ensure that your root `pom.xml` includes the following parent configuration:



```xml
    <parent>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-parent</artifactId>
        <version>2.6.2</version>
        <relativePath/> <!-- lookup parent from repository -->
    </parent>
```

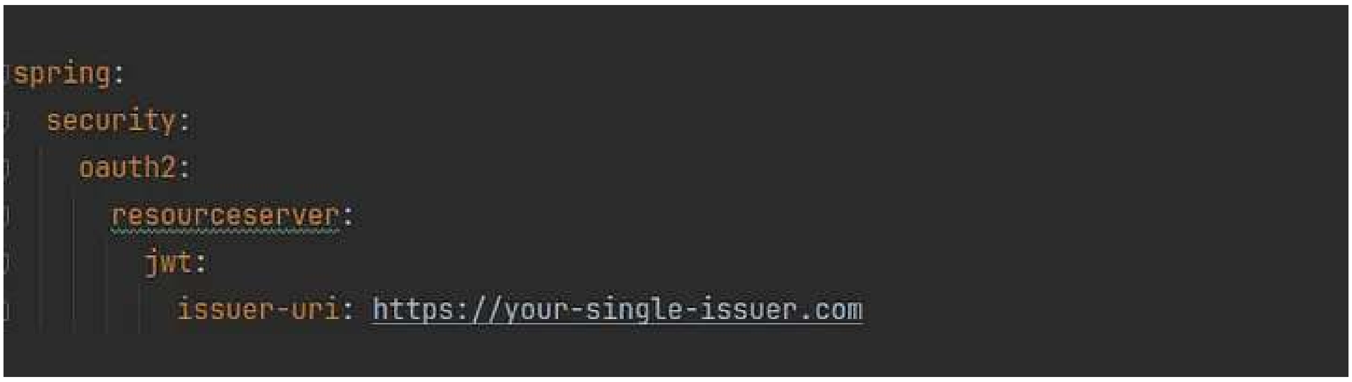
In the module where you are handling the security part of your application, make sure to include the following dependencies in the `pom.xml` file:



```xml
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-oauth2-resource-server</artifactId>
</dependency>
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-security</artifactId>
</dependency>
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-web</artifactId>
</dependency>
```

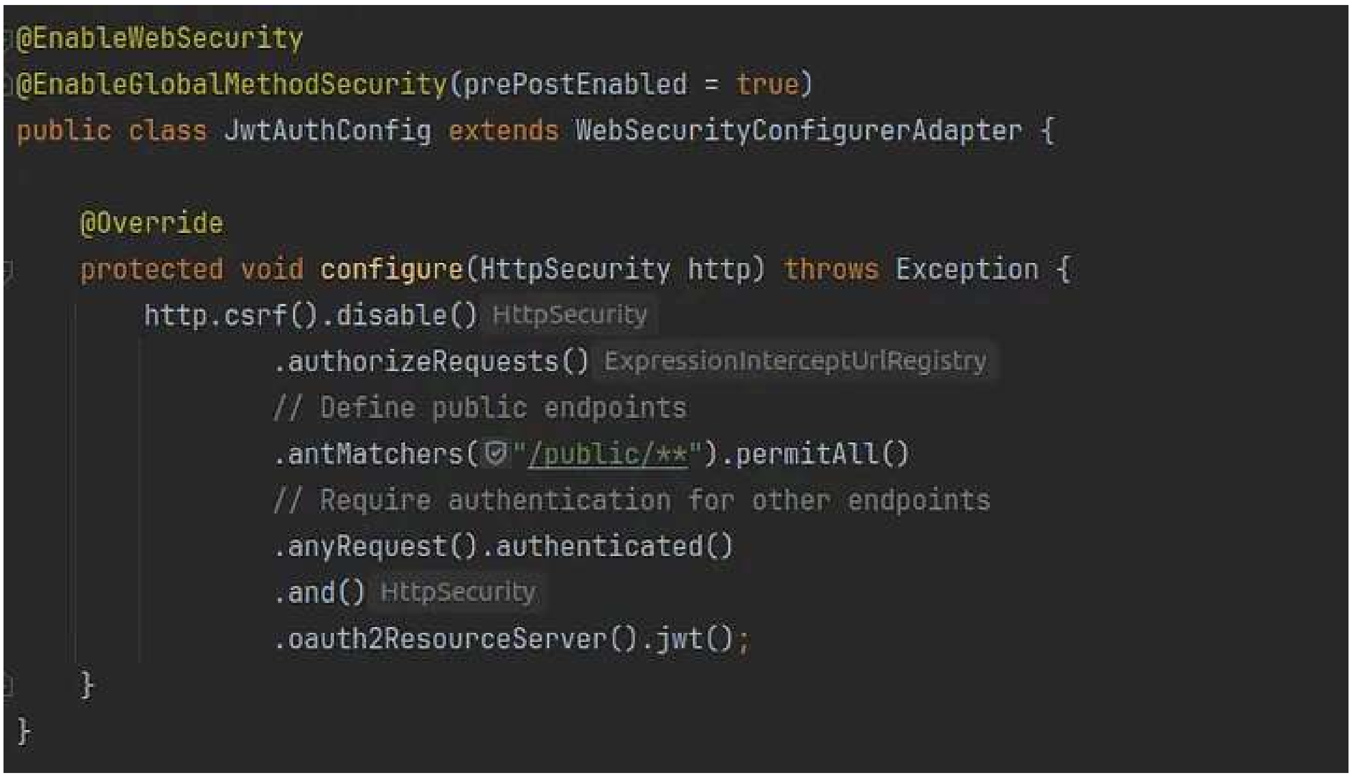
Handling a Single JWT issuer

To handle authentication and authorization with a single JWT issuer, we need to configure our Spring Security settings. Let’s start by defining our JWT issuer URI in the `application.yml` file:



```yaml
spring:
  security:
    oauth2:
      resourceserver:
        jwt:
          issuer-uri: https://your-single-issuer.com
```

Next, we’ll create a class that extends `org.springframework.security.config.annotation.web.configuration.WebSecurityConfigurerAdapter` and annotate it with `@org.springframework.security.config.annotation.web.configuration.EnableWebSecurity` and `@org.springframework.security.config.annotation.method.configuration.EnableGlobalMethodSecurity` to enable Spring Security and method-level security configuration:



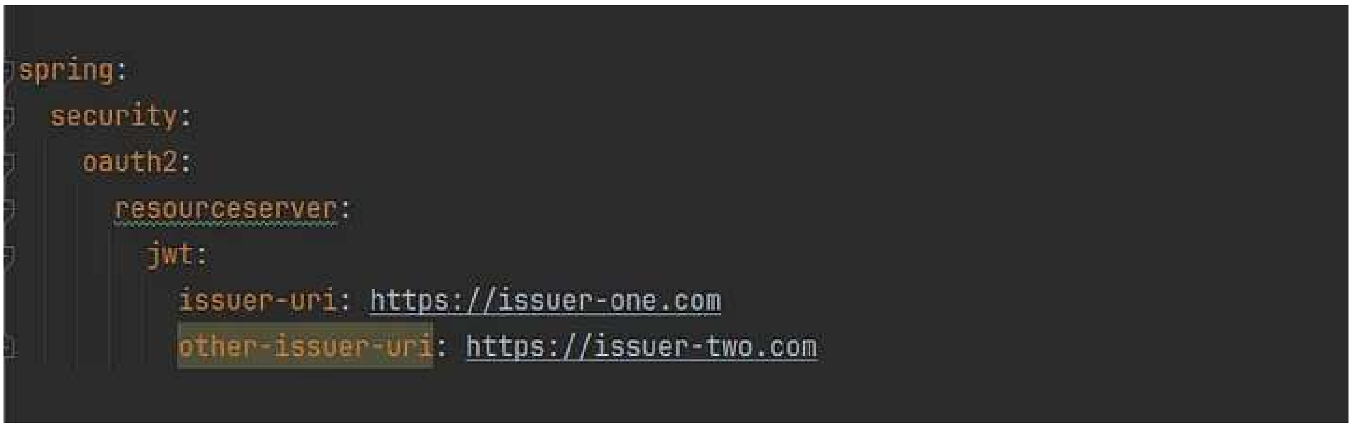
```
@EnableWebSecurity
@EnableGlobalMethodSecurity(prePostEnabled = true)
public class JwtAuthConfig extends WebSecurityConfigurerAdapter {

    @Override
    protected void configure(HttpSecurity http) throws Exception {
        http.csrf().disable() HttpSecurity
                .authorizeRequests() ExpressionInterceptUrlRegistry
                // Define public endpoints
                .antMatchers("/public/**").permitAll()
                // Require authentication for other endpoints
                .anyRequest().authenticated()
                .and() HttpSecurity
                .oauth2ResourceServer().jwt();
    }
}
```

In this configuration, we disable CSRF protection (as it's not relevant for stateless JWT authentication), define public endpoints that don't require authentication, and specify that all other requests need to be authenticated using JWT. With these configurations in place, our application is now ready to handle authentication and authorization using a single JWT issuer.

Handling multiple JWT issuers

When dealing with multiple JWT issuers, we need to configure our Spring Security settings to recognize and validate tokens from each issuer. Let's start by defining our JWT issuer URIs in the `application.yml` file:

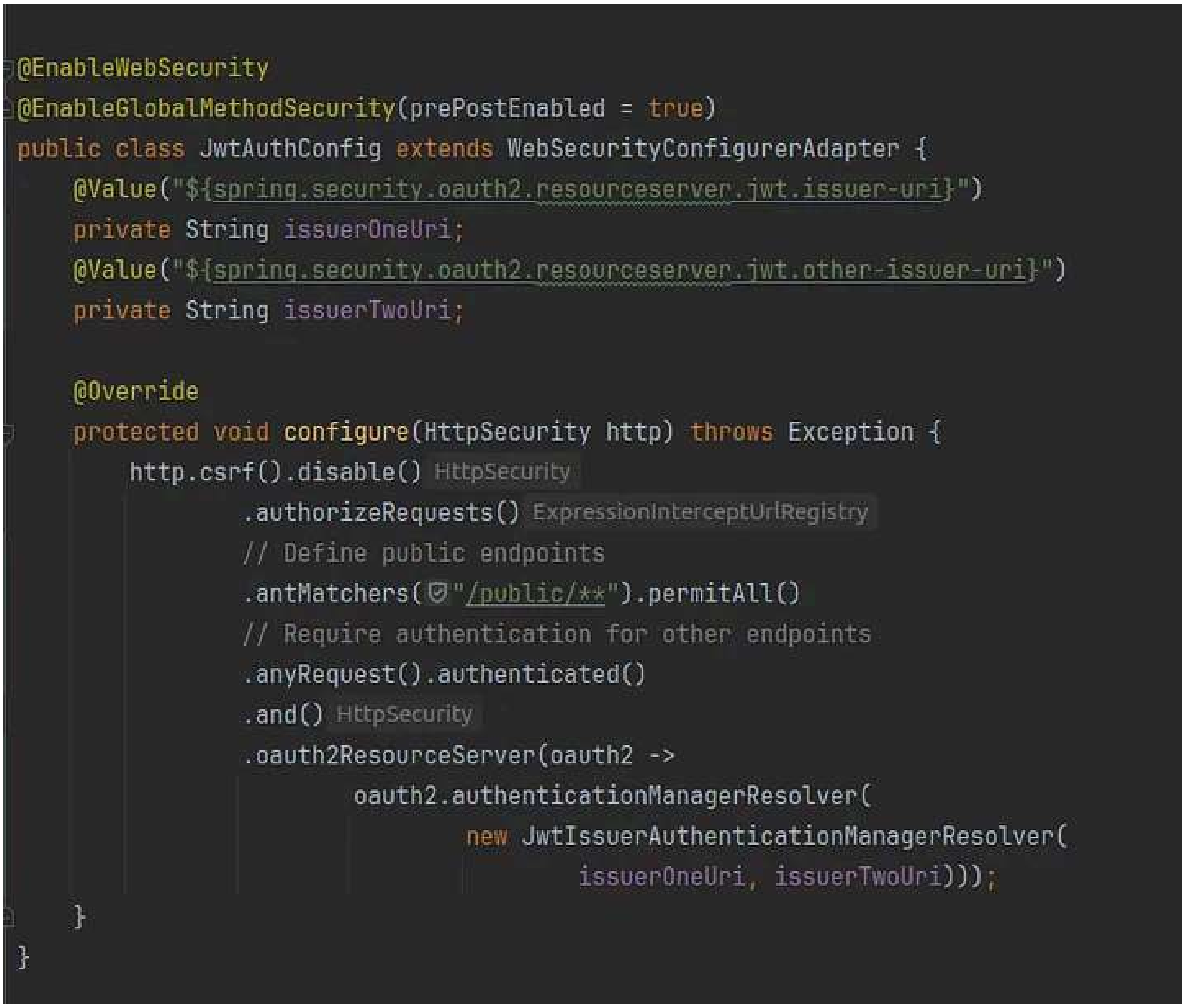


```
spring:
  security:
    oauth2:
      resourceserver:
        jwt:
          issuer-uri: https://issuer-one.com
          other-issuer-uri: https://issuer-two.com
```

Next, we'll update our `JwtAuthConfig` class to handle multiple issuers. In this configuration, we will use the

`org.springframework.security.oauth2.server.resource.authentication.JwtIssuerAuthen` `ticationManagerResolver` to resolve the correct authentication manager based on the issuer of the incoming JWT. Each authentication manager is responsible for validating tokens from a specific issuer. Additionally, we provide an `org.springframework.security.config.annotation.web.configurers.oauth2.server.resource.OAuth2ResourceServerConfigurer` customizer to the `oauth2ResourceServer` method. This customizer specifies how the resource server should be configured, including the authentication manager resolver that handles multiple issuers.

With these configurations in place, our application is now capable of handling authentication and authorization using multiple JWT issuers.



```java
@EnableWebSecurity
@EnableGlobalMethodSecurity(prePostEnabled = true)
public class JwtAuthConfig extends WebSecurityConfigurerAdapter {
    @Value("${spring.security.oauth2.resourceserver.jwt.issuer-uri}")
    private String issuerOneUri;
    @Value("${spring.security.oauth2.resourceserver.jwt.other-issuer-uri}")
    private String issuerTwoUri;

    @Override
    protected void configure(HttpSecurity http) throws Exception {
        http.csrf().disable() HttpSecurity
                .authorizeRequests() ExpressionInterceptUrlRegistry
                // Define public endpoints
                .antMatchers("/public/**").permitAll()
                // Require authentication for other endpoints
                .anyRequest().authenticated()
                .and() HttpSecurity
                .oauth2ResourceServer(oauth2 ->
                        oauth2.authenticationManagerResolver(
                                new JwtIssuerAuthenticationManagerResolver(
                                        issuerOneUri, issuerTwoUri)));
    }
}
```

Conclusion

In this article, we've seen how Spring Security simplifies authentication and authorization in Spring Boot applications, especially with JWT. We discussed basic Spring Boot configuration and then explored how to configure JWT issuers, both single and multiple, using Spring Security's flexibility.

By providing configurations in application.yml and extending WebSecurityConfigurerAdapter, developers can secure their applications easily. Spring Security's JwtIssuerAuthenticationManagerResolver makes handling multiple JWT issuers straightforward, ensuring robust authentication and authorization mechanisms.

In summary, Spring Security's powerful features abstract away security complexities, allowing developers to focus on building secure Spring Boot applications efficiently.

Spring Security · Jwt · Spring Boot · Oauth2 · Jwt Authentication

Follow

Written by Sargis Vardanyan

15 Followers · Writer for Octa Labs Insights

More from Sargis Vardanyan and Octa Labs Insights

Sargis Vardanyan in Octa Labs Insights

Deploying Firebase Services with GitHub Actions: A Step-by-Step Guide

Introduction As we know, Google provides numerous ways to manage Firebase resources with various authentication mechanisms and tools. In...

Jul 12

Artur Tarverdyan in Octa Labs Insights

A Beginner's Guide to Telegram Mini Apps

Messaging apps have become an integral part of our daily lives, offering convenience and connectivity at our fingertips. Now, Telegram has...

May 6 210 1

Artur Tarverdyan in Octa Labs Insights

Clicker App: Telegram Mini Apps Part 2

Create your clicker app under 5 minutes

Jun 21 29

Sargis Vardanyan in Octa Labs Insights

Do we really need passwords any more?

In an increasingly digital world, the use of passwords has been a cornerstone of online security for decades. However, the landscape of...

Apr 1 12

See all from Sargis Vardanyan

See all from Octa Labs Insights

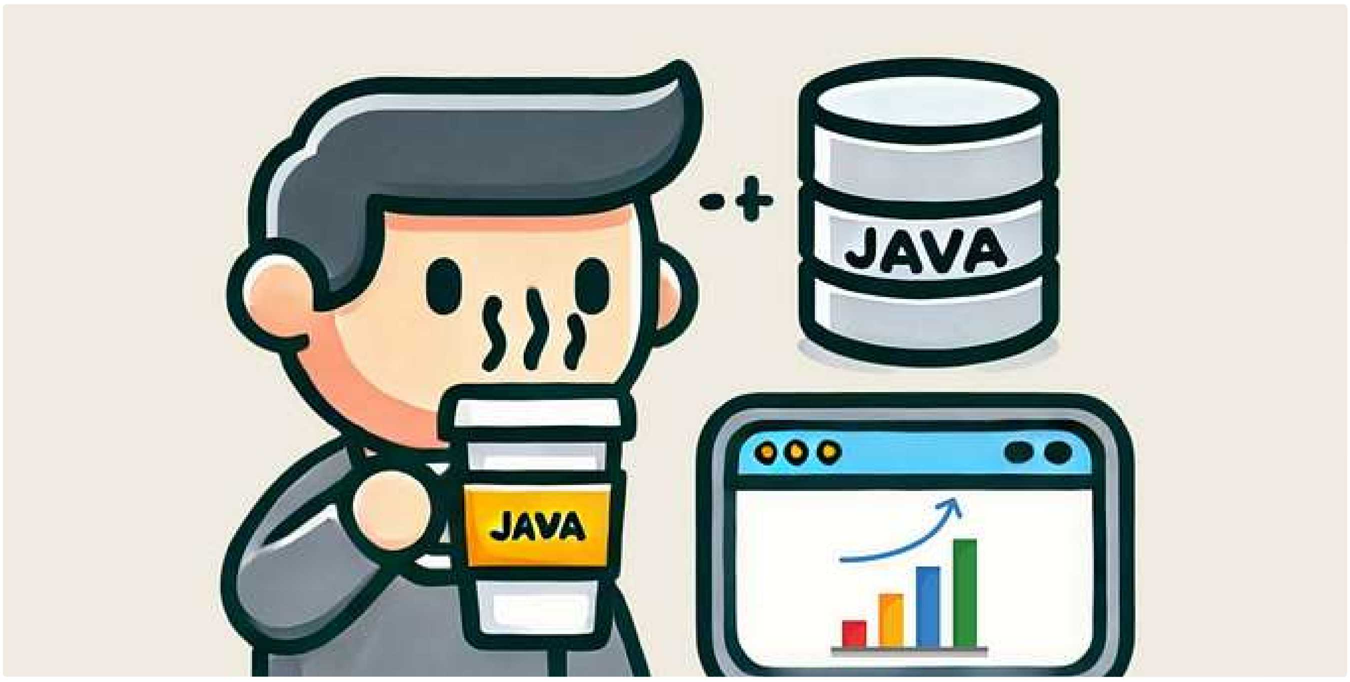
Recommended from Medium

Ankita Kolhe

PUT vs PATCH

In my recent interview, Interviewer asked me lot of question on this topic.

May 30 108 1



Alan Matheus

Improving Database Configuration Performance in a Spring Java Application

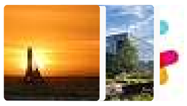
My article is open to everyone; non-member readers can click this link to read the full text.

Jun 30 49 1

Lists

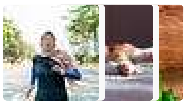
Staff Picks

700 stories · 1180 saves



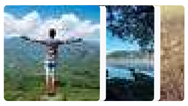
Stories to Help You Level-Up at Work

19 stories · 712 saves



Self-Improvement 101

20 stories · 2427 saves



Productivity 101

20 stories · 2125 saves

Eric Anicet

Lab8 (Spring Boot/K8S): Deploy a Spring Boot application on Kubernetes using Helm Chart

In this story, we'll explore the basic concepts of using Helm to deploy a Spring Boot application on Kubernetes cluster.

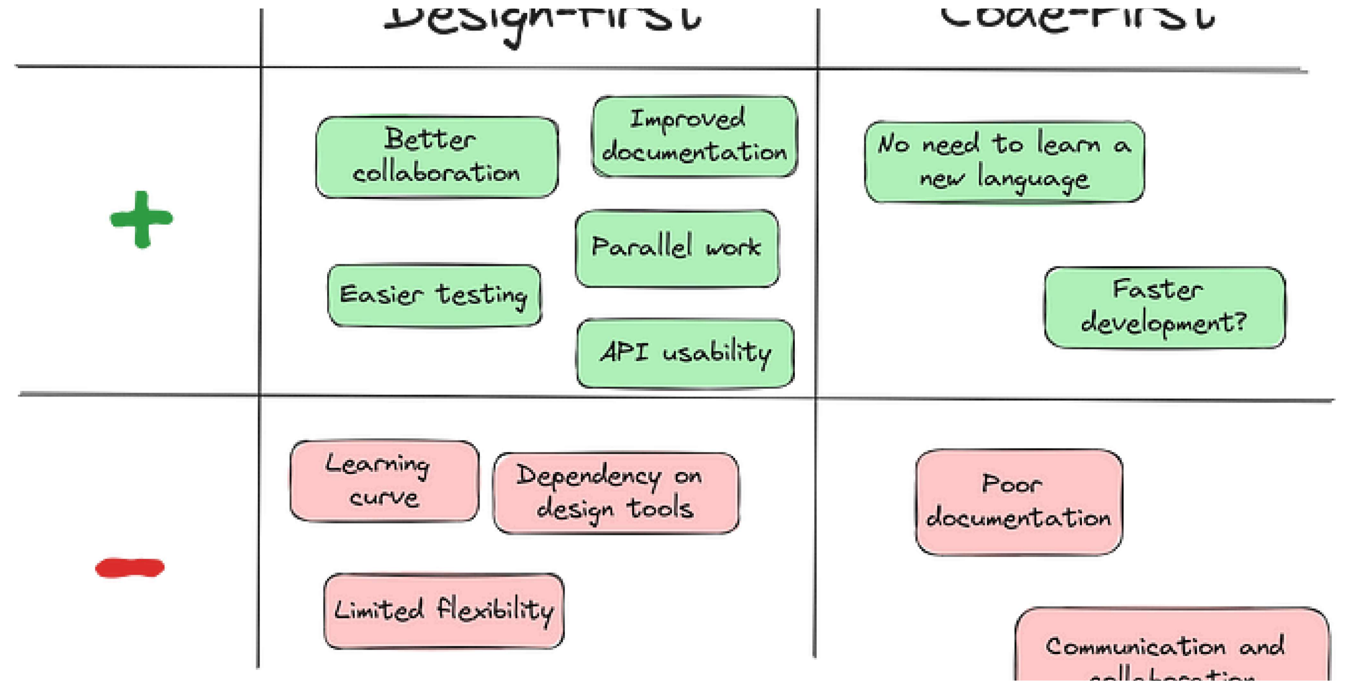
Jul 22 5

Nishada Liyanage in hexaDefence

Keycloak Integration with a Spring Boot API

Keycloak Access token validation

Jul 19 5



Ansgar Schulte in T3CH

API-Design: Design-First vs. Code-First

Is there the „best" approach?

Feb 10 103

Satish Dixit

How to improve WebClient response time in Spring Boot

Improving the response time of WebClient in a Spring Boot application can significantly enhance the performance of your web services.

Jul 18 24 1

See more recommendations