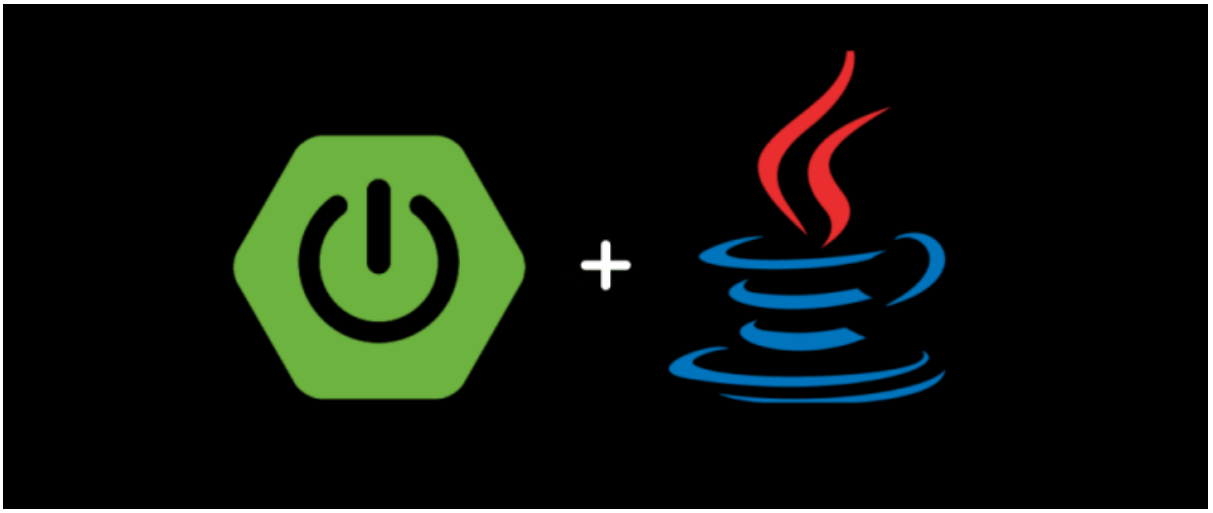
# Upgrading Spring Boot Version from 2.x.x to 3.x.x: A Step-by-Step Guide

Are you using Spring Boot 2.x.x and want to upgrade to the latest version, 3.x.x? In this blog post, we will walk you through the process of upgrading your Spring Boot application, taking into account the necessary changes and considerations. Let's get started!



**Prerequisites**
Before we begin the upgrade process, make sure you have the following prerequisites in place:

**Java Version**: Ensure that you have Java 17 installed, as Spring Boot 3.x.x requires this version.

**Spring Security Version**: Check that you are using Spring Security 6.0. If not, upgrade your Spring Security version to 6.0 before proceeding with the Spring Boot upgrade.

**Dependencies Update**: Spring Boot 3.x.x uses Jakarta EE 9 APIs (jakarta.*) instead of EE 8 (javax.*). To update the necessary dependencies, make the following changes:

For the **javax.persistence** package, replace the old import statement with the new **jakarta.persistence** package. Additionally, update the dependency in your project's pom.xml file as follows:

```
<dependency>
    <groupId>jakarta.persistence</groupId>
  <artifactId>jakarta.persistence-api</artifactId>
    <version>3.1.0</version> // latest version
</dependency>
```

For the **javax.servlet.*** package, replace the old import statement with the new **jakarta.servlet.*** package. Similarly, update the servlet dependency in your pom.xml file:

```
<dependency>
    <groupId>jakarta.servlet</groupId>
    <artifactId>jakarta.servlet-api</artifactId>
    <version>6.0.0</version>
</dependency>
```

**Swagger Dependency Update**: If you are using Swagger for API documentation, update **springfox-swagger** to **springdoc-openapi-starter-webmvc-ui** dependency in your pom.xml file.

```
<dependency>
    <groupId>org.springdoc</groupId>

<artifactId>springdoc-openapi-starter-webmvc-ui</a
rtifactId>
    <version>2.1.0</version>
</dependency>
```

Additionally, update the OpenAPI definition to reflect the changes in Swagger's configuration:

Replace the old **@OpenAPIDefinition** annotation with the new one, including the updated structure:
**Old Way**:

```
@OpenAPIDefinition(info = @Info(title = "Title",
version = "1.0", description = "description"))
@SecurityScheme(name = "bearerAuth", scheme =
"bearer", type = SecuritySchemeType.HTTP,
bearerFormat = "JWT")
```

**New Way**:

```
@OpenAPIDefinition(info = @Info(
    title = "Title",
    version = "1.0",
    description = "REST API description...",
    contact = @Contact(name = "Contact")),
    security = {@SecurityRequirement(name =
"bearerToken")}
)
```

Add the new **@SecuritySchemes** annotation to define the bearer token security scheme:

```
@SecuritySchemes({
    @SecurityScheme(name = "bearerToken", type =
SecuritySchemeType.HTTP,
            scheme = "bearer", bearerFormat =
"JWT")
})
```

**Configuration Changes**

Spring Boot 3.x.x introduces some changes to the configuration of Spring Security and HTTP security.

**WebSecurityConfigurerAdapter** class is deprecated in the new version of spring boot. Configuring AuthenticationManager without WebSecurityConfigurerAdapter:

**Old Way**:

```
@Override
@Bean
public AuthenticationManager
authenticationManagerBean() throws Exception {
        return super.authenticationManagerBean();
}
```

**New way:**

```
@Bean
public AuthenticationManager
authenticationManager(AuthenticationConfiguration
config)
            throws Exception {
    return config.getAuthenticationManager();
}
```

**Configuring HTTP Security with SecurityFilterChain**: In the new version, the WebSecurityConfigurerAdapter class is no longer necessary. Instead, you can use the **SecurityFilterChain** interface to configure the HTTP security. Here's an example of how to configure the HttpSecurity using SecurityFilterChain:

```
@Configuration
public class SecurityConfiguration {
    @Bean
    public SecurityFilterChain
filterChain(HttpSecurity http) throws Exception {
        http
            .authorizeHttpRequests((authz) ->
authz
                .anyRequest().authenticated()
            )
            .httpBasic(withDefaults());
        return http.build();
    }

}
```

**Configuring WebSecurity without WebSecurityConfigurerAdapter:**
Similarly, if you were previously using **WebSecurityConfigurerAdapter** to configure WebSecurity, you can now achieve the same result using the **WebSecurityCustomizer** interface. Here's an example:

```
@Configuration
public class SecurityConfiguration {

    @Bean
    public WebSecurityCustomizer
webSecurityCustomizer() {
        return (web) ->
web.ignoring().antMatchers("/ignore1",
"/ignore2");
    }

}
```

**Route Permissions with requestMatchers**:
In the new version, the use of antMatchers has been deprecated. Instead, you can permit or restrict API routes using requestMatchers. Here's an example of the old and new approaches:
**Old Way**:

```
@Override
protected void configure(HttpSecurity http) throws
Exception {
    http
```

```
        .cors().and().csrf().disable()
        .authorizeRequests()
        .antMatchers("/swagger-ui/**",
"/v3/api-docs/**", "/swagger-ui.html").permitAll()
        .anyRequest().authenticated();
}
```

**New Way**:

```
@Bean
public SecurityFilterChain
filterChain(HttpSecurity http) throws Exception {

http.csrf(AbstractHttpConfigurer::disable).cors(Cu
stomizer.withDefaults())
            .authorizeHttpRequests(auth -> auth

.requestMatchers("/swagger-ui/**",
"/v3/api-docs/**",
"/swagger-ui.html").permitAll());
    return http.build();
}
```

## Conclusion

In this blog post, we covered the essential steps and changes required for the upgrade, including updating dependencies, configuring security, and handling API documentation with Swagger.

Happy coding!

References:
https://docs.spring.io/spring-security/reference/migration/servlet/session-management.html
https://www.baeldung.com/spring-deprecated-websecurityconfigureradapter
https://www.baeldung.com/spring-rest-openapi-documentation
https://spring.io/blog/2022/02/21/spring-security-without-the-websecurityconfigureradapter