* [Spring Security](https://docs.spring.io/spring-security/reference/5.8/index.html)
* [Preparing for 6.0](https://docs.spring.io/spring-security/reference/5.8/migration/index.html)
* [Servlet Migrations](https://docs.spring.io/spring-security/reference/5.8/migration/servlet/index.html)
* [Configuration](https://docs.spring.io/spring-security/reference/5.8/migration/servlet/config.html)

**Configuration Migrations**

The following steps relate to changes around how to configure HttpSecurity, WebSecurity, and AuthenticationManager.

**Add @Configuration annotation**

In 6.0, @Configuration is removed from @EnableWebSecurity, @EnableMethodSecurity, @EnableGlobalMethodSecurity, and @EnableGlobalAuthentication.

To prepare for this, wherever you are using one of these annotations, you may need to add @Configuration. For example, @EnableMethodSecurity changes from:

* **Java**

@EnableMethodSecurity

public class MyConfiguration {

*// ...*

}

Copied!

* **Kotlin**

@EnableMethodSecurity

open class MyConfiguration {

*// ...*

}

Copied!

to:

* **Java**

@Configuration

@EnableMethodSecurity

public class MyConfiguration {

*// ...*

}

Copied!

* **Kotlin**

@Configuration

@EnableMethodSecurity

open class MyConfiguration {

*// ...*

}

Copied!

**Use the new requestMatchers methods**

In Spring Security 5.8, the [antMatchers](https://docs.spring.io/spring-security/site/docs/5.8.12/api/org/springframework/security/config/annotation/web/AbstractRequestMatcherRegistry.html#antMatchers(java.lang.String%E2%80%A6%E2%80%8B)), [mvcMatchers](https://docs.spring.io/spring-security/site/docs/5.8.12/api/org/springframework/security/config/annotation/web/AbstractRequestMatcherRegistry.html#mvcMatchers(java.lang.String%E2%80%A6%E2%80%8B)), and [regexMatchers](https://docs.spring.io/spring-security/site/docs/5.8.12/api/org/springframework/security/config/annotation/web/AbstractRequestMatcherRegistry.html#regexMatchers(java.lang.String%E2%80%A6%E2%80%8B)) methods were deprecated in favor of new [requestMatchers methods](https://docs.spring.io/spring-security/reference/5.8/servlet/authorization/authorize-http-requests.html#_request_matchers).

The new requestMatchers methods were added [to authorizeHttpRequests](https://docs.spring.io/spring-security/reference/5.8/servlet/authorization/authorize-http-requests.html), authorizeRequests, CSRF configuration, WebSecurityCustomizer and any other places that had the specialized RequestMatcher methods. The deprecated methods are removed in Spring Security 6.

These new methods have more secure defaults since they choose the most appropriate RequestMatcher implementation for your application. In summary, the new methods choose the MvcRequestMatcher implementation if your application has Spring MVC in the classpath, falling back to the AntPathRequestMatcher implementation if Spring MVC is not present (aligning the behavior with the Kotlin equivalent methods).

To start using the new methods, you can replace the deprecated methods with the new ones. For example, the following application configuration:

* **Java**

@Configuration

@EnableWebSecurity

public class SecurityConfig {

@Bean

public SecurityFilterChain securityFilterChain(HttpSecurity http) throws Exception {

http

.authorizeHttpRequests((authz) -> authz

.antMatchers("/api/admin/\*\*").hasRole("ADMIN")

.antMatchers("/api/user/\*\*").hasRole("USER")

.anyRequest().authenticated()

);

return http.build();

}

}

Copied!

can be changed to:

* **Java**

@Configuration

@EnableWebSecurity

public class SecurityConfig {

@Bean

public SecurityFilterChain securityFilterChain(HttpSecurity http) throws Exception {

http

.authorizeHttpRequests((authz) -> authz

.requestMatchers("/api/admin/\*\*").hasRole("ADMIN")

.requestMatchers("/api/user/\*\*").hasRole("USER")

.anyRequest().authenticated()

);

return http.build();

}

}

Copied!

If you have Spring MVC in the classpath and are using the mvcMatchers methods, you can replace it with the new methods and Spring Security will choose the MvcRequestMatcher implementation for you. The following configuration:

* **Java**

@Configuration

@EnableWebSecurity

@EnableWebMvc

public class SecurityConfig {

@Bean

SecurityFilterChain securityFilterChain(HttpSecurity http) throws Exception {

http

.authorizeHttpRequests((authz) -> authz

.mvcMatchers("/admin/\*\*").hasRole("ADMIN")

.anyRequest().authenticated()

);

return http.build();

}

}

Copied!

is equivalent to:

* **Java**

@Configuration

@EnableWebSecurity

@EnableWebMvc

public class SecurityConfig {

@Bean

SecurityFilterChain securityFilterChain(HttpSecurity http) throws Exception {

http

.authorizeHttpRequests((authz) -> authz

.requestMatchers("/admin/\*\*").hasRole("ADMIN")

.anyRequest().authenticated()

);

return http.build();

}

}

Copied!

If you are customizing the servletPath property of the MvcRequestMatcher, you can now use the MvcRequestMatcher.Builder to create MvcRequestMatcher instances that share the same servlet path:

* **Java**

@Configuration

@EnableWebSecurity

@EnableWebMvc

public class SecurityConfig {

@Bean

SecurityFilterChain securityFilterChain(HttpSecurity http) throws Exception {

http

.authorizeHttpRequests((authz) -> authz

.mvcMatchers("/admin").servletPath("/path").hasRole("ADMIN")

.mvcMatchers("/user").servletPath("/path").hasRole("USER")

.anyRequest().authenticated()

);

return http.build();

}

}

Copied!

The code above can be rewritten using the MvcRequestMatcher.Builder and the requestMatchers method:

* **Java**

@Configuration

@EnableWebSecurity

@EnableWebMvc

public class SecurityConfig {

@Bean

SecurityFilterChain securityFilterChain(HttpSecurity http, HandlerMappingIntrospector introspector) throws Exception {

MvcRequestMatcher.Builder mvcMatcherBuilder = new MvcRequestMatcher.Builder(introspector).servletPath("/path");

http

.authorizeHttpRequests((authz) -> authz

.requestMatchers(mvcMatcherBuilder.pattern("/admin")).hasRole("ADMIN")

.requestMatchers(mvcMatcherBuilder.pattern("/user")).hasRole("USER")

.anyRequest().authenticated()

);

return http.build();

}

}

Copied!

If you are having problem with the new requestMatchers methods, you can always switch back to the RequestMatcher implementation that you were using. For example, if you still want to use AntPathRequestMatcher and RegexRequestMatcher implementations, you can use the requestMatchers method that accepts a RequestMatcher instance:

* **Java**

import static org.springframework.security.web.util.matcher.AntPathRequestMatcher.antMatcher;

import static org.springframework.security.web.util.matcher.RegexRequestMatcher.regexMatcher;

@Configuration

@EnableWebSecurity

public class SecurityConfig {

@Bean

SecurityFilterChain securityFilterChain(HttpSecurity http) throws Exception {

http

.authorizeHttpRequests((authz) -> authz

.requestMatchers(antMatcher("/user/\*\*")).hasRole("USER")

.requestMatchers(antMatcher(HttpMethod.POST, "/user/\*\*")).hasRole("ADMIN")

.requestMatchers(regexMatcher(".\*\\?x=y")).hasRole("SPECIAL") *// matches /any/path?x=y*

.anyRequest().authenticated()

);

return http.build();

}

}

Copied!

Note that the above sample uses static factory methods from [AntPathRequestMatcher](https://docs.spring.io/spring-security/site/docs/5.8.12/api/org/springframework/security/web/util/matcher/AntPathRequestMatcher.html) and [RegexRequestMatcher](https://docs.spring.io/spring-security/site/docs/5.8.12/api/org/springframework/security/web/util/matcher/RegexRequestMatcher.html) to improve readability.

If you are using the WebSecurityCustomizer interface, you can replace the deprecated antMatchers methods:

* **Java**

@Bean

public WebSecurityCustomizer webSecurityCustomizer() {

return (web) -> web.ignoring().antMatchers("/ignore1", "/ignore2");

}

Copied!

with their requestMatchers counterparts:

* **Java**

@Bean

public WebSecurityCustomizer webSecurityCustomizer() {

return (web) -> web.ignoring().requestMatchers("/ignore1", "/ignore2");

}

Copied!

The same way, if you are customizing the CSRF configuration to ignore some paths, you can replace the deprecated methods with the requestMatchers methods:

* **Java**

@Bean

public SecurityFilterChain filterChain(HttpSecurity http) throws Exception {

http

.csrf((csrf) -> csrf

.ignoringAntMatchers("/no-csrf")

);

return http.build();

}

Copied!

can be changed to:

* **Java**

@Bean

public SecurityFilterChain filterChain(HttpSecurity http) throws Exception {

http

.csrf((csrf) -> csrf

.ignoringRequestMatchers("/no-csrf")

);

return http.build();

}

Copied!

**Use the new securityMatchers methods**

In Spring Security 5.8, the antMatchers, mvcMatchers and requestMatchers methods from HttpSecurity were deprecated in favor of new securityMatchers methods.

Note that these methods are not the same from authorizeHttpRequests methods [which were deprecated](https://docs.spring.io/spring-security/reference/5.8/migration/servlet/config.html#use-new-requestmatchers) in favor of the requestMatchers methods. However, the securityMatchers methods are similar to the requestMatchers methods in the sense that they will choose the most appropriate RequestMatcher implementation for your application. In summary, the new methods choose the MvcRequestMatcher implementation if your application has Spring MVC in the classpath, falling back to the AntPathRequestMatcher implementation if Spring MVC is not present (aligning the behavior with the Kotlin equivalent methods). Another reason for adding the securityMatchers methods is to avoid confusion with the requestMatchers methods from authorizeHttpRequests.

The following configuration:

* **Java**

@Bean

public SecurityFilterChain filterChain(HttpSecurity http) throws Exception {

http

.antMatcher("/api/\*\*", "/app/\*\*")

.authorizeHttpRequests((authz) -> authz

.requestMatchers("/api/admin/\*\*").hasRole("ADMIN")

.anyRequest().authenticated()

);

return http.build();

}

Copied!

can be rewritten using the securityMatchers methods:

* **Java**

@Bean

public SecurityFilterChain filterChain(HttpSecurity http) throws Exception {

http

.securityMatcher("/api/\*\*", "/app/\*\*")

.authorizeHttpRequests((authz) -> authz

.requestMatchers("/api/admin/\*\*").hasRole("ADMIN")

.anyRequest().authenticated()

);

return http.build();

}

Copied!

If you are using a custom RequestMatcher in your HttpSecurity configuration:

* **Java**

@Bean

public SecurityFilterChain filterChain(HttpSecurity http) throws Exception {

http

.requestMatcher(new MyCustomRequestMatcher())

.authorizeHttpRequests((authz) -> authz

.requestMatchers("/api/admin/\*\*").hasRole("ADMIN")

.anyRequest().authenticated()

);

return http.build();

}

public class MyCustomRequestMatcher implements RequestMatcher {

*// ...*

}

Copied!

you can do the same using securityMatcher:

* **Java**

@Bean

public SecurityFilterChain filterChain(HttpSecurity http) throws Exception {

http

.securityMatcher(new MyCustomRequestMatcher())

.authorizeHttpRequests((authz) -> authz

.requestMatchers("/api/admin/\*\*").hasRole("ADMIN")

.anyRequest().authenticated()

);

return http.build();

}

public class MyCustomRequestMatcher implements RequestMatcher {

*// ...*

}

Copied!

If you are combining multiple RequestMatcher implementations in your HttpSecurity configuration:

* **Java**

@Bean

public SecurityFilterChain filterChain(HttpSecurity http) throws Exception {

http

.requestMatchers((matchers) -> matchers

.antMatchers("/api/\*\*", "/app/\*\*")

.mvcMatchers("/admin/\*\*")

.requestMatchers(new MyCustomRequestMatcher())

)

.authorizeHttpRequests((authz) -> authz

.requestMatchers("/admin/\*\*").hasRole("ADMIN")

.anyRequest().authenticated()

);

return http.build();

}

Copied!

you can change it by using securityMatchers:

* **Java**

@Bean

public SecurityFilterChain filterChain(HttpSecurity http) throws Exception {

http

.securityMatchers((matchers) -> matchers

.requestMatchers("/api/\*\*", "/app/\*\*", "/admin/\*\*")

.requestMatchers(new MyCustomRequestMatcher())

)

.authorizeHttpRequests((authz) -> authz

.requestMatchers("/admin/\*\*").hasRole("ADMIN")

.anyRequest().authenticated()

);

return http.build();

}

Copied!

If you are having problems with the securityMatchers methods choosing the RequestMatcher implementation for you, you can always choose the RequestMatcher implementation yourself:

* **Java**

import static org.springframework.security.web.util.matcher.AntPathRequestMatcher.antMatcher;

@Bean

public SecurityFilterChain filterChain(HttpSecurity http) throws Exception {

http

.securityMatchers((matchers) -> matchers

.requestMatchers(antMatcher("/api/\*\*"), antMatcher("/app/\*\*"))

)

.authorizeHttpRequests((authz) -> authz

.requestMatchers(antMatcher("/api/admin/\*\*")).hasRole("ADMIN")

.anyRequest().authenticated()

);

return http.build();

}

Copied!

**Stop Using WebSecurityConfigurerAdapter**

**Publish a SecurityFilterChain Bean**

Spring Security 5.4 introduced the capability to publish a SecurityFilterChain bean instead of extending WebSecurityConfigurerAdapter. In 6.0, WebSecurityConfigurerAdapter is removed. To prepare for this change, you can replace constructs like:

* **Java**
* **Kotlin**

@Configuration

public class SecurityConfiguration extends WebSecurityConfigurerAdapter {

@Override

protected void configure(HttpSecurity http) throws Exception {

http

.authorizeHttpRequests((authorize) -> authorize

.anyRequest().authenticated()

)

.httpBasic(withDefaults());

}

}

Copied!

with:

* **Java**
* **Kotlin**

@Configuration

public class SecurityConfiguration {

@Bean

public SecurityFilterChain filterChain(HttpSecurity http) throws Exception {

http

.authorizeHttpRequests((authorize) -> authorize

.anyRequest().authenticated()

)

.httpBasic(withDefaults());

return http.build();

}

}

Copied!

**Publish a WebSecurityCustomizer Bean**

Spring Security 5.4 [introduced WebSecurityCustomizer](https://github.com/spring-projects/spring-security/issues/8978) to replace configure(WebSecurity web) in WebSecurityConfigurerAdapter. To prepare for its removal, you can replace code like the following:

* **Java**
* **Kotlin**

@Configuration

public class SecurityConfiguration extends WebSecurityConfigurerAdapter {

@Override

public void configure(WebSecurity web) {

web.ignoring().antMatchers("/ignore1", "/ignore2");

}

}

Copied!

with:

* **Java**
* **Kotlin**

@Configuration

public class SecurityConfiguration {

@Bean

public WebSecurityCustomizer webSecurityCustomizer() {

return (web) -> web.ignoring().antMatchers("/ignore1", "/ignore2");

}

}

Copied!

**Publish an AuthenticationManager Bean**

As part of WebSecurityConfigurerAdapter removal, configure(AuthenticationManagerBuilder) is also removed. Preparing for its removal will differ based on your reason for using it.

**LDAP Authentication**

If you are using auth.ldapAuthentication() for [LDAP authentication support](https://docs.spring.io/spring-security/reference/5.8/servlet/authentication/passwords/ldap.html), you can replace:

* **Java**
* **Kotlin**

@Configuration

public class SecurityConfiguration extends WebSecurityConfigurerAdapter {

@Override

protected void configure(AuthenticationManagerBuilder auth) throws Exception {

auth

.ldapAuthentication()

.userDetailsContextMapper(new PersonContextMapper())

.userDnPatterns("uid={0},ou=people")

.contextSource()

.port(0);

}

}

Copied!

with:

* **Java**
* **Kotlin**

@Configuration

public class SecurityConfiguration {

@Bean

public EmbeddedLdapServerContextSourceFactoryBean contextSourceFactoryBean() {

EmbeddedLdapServerContextSourceFactoryBean contextSourceFactoryBean =

EmbeddedLdapServerContextSourceFactoryBean.fromEmbeddedLdapServer();

contextSourceFactoryBean.setPort(0);

return contextSourceFactoryBean;

}

@Bean

AuthenticationManager ldapAuthenticationManager(BaseLdapPathContextSource contextSource) {

LdapBindAuthenticationManagerFactory factory =

new LdapBindAuthenticationManagerFactory(contextSource);

factory.setUserDnPatterns("uid={0},ou=people");

factory.setUserDetailsContextMapper(new PersonContextMapper());

return factory.createAuthenticationManager();

}

}

Copied!

**JDBC Authentication**

If you are using auth.jdbcAuthentication() for [JDBC Authentication support](https://docs.spring.io/spring-security/reference/5.8/servlet/authentication/passwords/jdbc.html), you can replace:

* **Java**
* **Kotlin**

@Configuration

public class SecurityConfiguration extends WebSecurityConfigurerAdapter {

@Bean

public DataSource dataSource() {

return new EmbeddedDatabaseBuilder()

.setType(EmbeddedDatabaseType.H2)

.build();

}

@Override

protected void configure(AuthenticationManagerBuilder auth) throws Exception {

UserDetails user = User.withDefaultPasswordEncoder()

.username("user")

.password("password")

.roles("USER")

.build();

auth.jdbcAuthentication()

.withDefaultSchema()

.dataSource(this.dataSource)

.withUser(user);

}

}

Copied!

with:

* **Java**
* **Kotlin**

@Configuration

public class SecurityConfiguration {

@Bean

public DataSource dataSource() {

return new EmbeddedDatabaseBuilder()

.setType(EmbeddedDatabaseType.H2)

.addScript(JdbcDaoImpl.DEFAULT\_USER\_SCHEMA\_DDL\_LOCATION)

.build();

}

@Bean

public UserDetailsManager users(DataSource dataSource) {

UserDetails user = User.withDefaultPasswordEncoder()

.username("user")

.password("password")

.roles("USER")

.build();

JdbcUserDetailsManager users = new JdbcUserDetailsManager(dataSource);

users.createUser(user);

return users;

}

}

Copied!

**In-Memory Authentication**

If you are using auth.inMemoryAuthentication() for [In-Memory Authentication support](https://docs.spring.io/spring-security/reference/5.8/servlet/authentication/passwords/in-memory.html), you can replace:

* **Java**
* **Kotlin**

@Configuration

public class SecurityConfiguration extends WebSecurityConfigurerAdapter {

@Override

protected void configure(AuthenticationManagerBuilder auth) throws Exception {

UserDetails user = User.withDefaultPasswordEncoder()

.username("user")

.password("password")

.roles("USER")

.build();

auth.inMemoryAuthentication()

.withUser(user);

}

}

Copied!

with:

* **Java**
* **Kotlin**

@Configuration

public class SecurityConfiguration {

@Bean

public InMemoryUserDetailsManager userDetailsService() {

UserDetails user = User.withDefaultPasswordEncoder()

.username("user")

.password("password")

.roles("USER")

.build();

return new InMemoryUserDetailsManager(user);

}

}

Copied!

**Add @Configuration to @Enable\* annotations**

In 6.0, all Spring Security’s @Enable\* annotations had their @Configuration removed. While convenient, it was not consistent with the rest of the Spring projects and most notably Spring Framework’s @Enable\* annotations. Additionally, the introduction of support for @Configuration(proxyBeanMethods=false) in Spring Framework provides another reason to remove @Configuration meta-annotation from Spring Security’s @Enable\* annotations and allow users to opt into their preferred configuration mode.

The following annotations had their @Configuration removed:

* @EnableGlobalAuthentication
* @EnableGlobalMethodSecurity
* @EnableMethodSecurity
* @EnableReactiveMethodSecurity
* @EnableWebSecurity
* @EnableWebFluxSecurity

For example, if you are using @EnableWebSecurity, you will need to change:

* **Java**

@EnableWebSecurity

public class SecurityConfig {

*// ...*

}

Copied!

to:

* **Java**

@Configuration

@EnableWebSecurity

public class SecurityConfig {

*// ...*

}

Copied!

And the same applies to every other annotation listed above.

**Other Scenarios**

If you are using AuthenticationManagerBuilder for something more sophisticated, you can [publish your own AuthenticationManager @Bean](https://docs.spring.io/spring-security/reference/5.8/servlet/authentication/architecture.html#servlet-authentication-authenticationmanager) or wire an AuthenticationManager instance into the HttpSecurity DSL with [HttpSecurity#authenticationManager](https://docs.spring.io/spring-security/site/docs/5.8.12/api/org/springframework/security/config/annotation/web/builders/HttpSecurity.html#authenticationManager(org.springframework.security.authentication.AuthenticationManager)).