How to Define a Spring Boot Filter?

## 1. Overview

Spring Security is based on a chain of servlet filters. Each filter has a specific responsibility and depending on the configuration, filters are added or removed.

## **Security Debugging**

First, we'll enable security debugging which will log detailed security information on each request.

We can enable security debugging using the debug property:

@EnableWebSecurity(debug = true)

This way, when we send a request to the server, all the request information will be logged.

We'll also be able to see the entire security filter chain:

Security filter chain: [

WebAsyncManagerIntegrationFilter

SecurityContextPersistenceFilter

HeaderWriterFilter

LogoutFilter

UsernamePasswordAuthenticationFilter

// ...

]

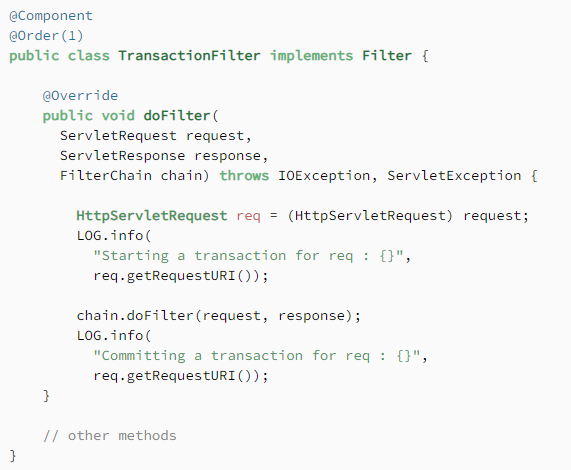
## 2. Defining Filters and the Invocation Order

Let's start by creating two filters:

1. TransactionFilter – to start and commit transactions
2. RequestResponseLoggingFilter – to log requests and responses

In order to create a filter, we simply need to implement the Filter interface:

Example 1 :



Example 2 :



In order for Spring to recognize a filter, we need to define it as a bean with the *@Component* annotation.

**Moreover, to have the filters fire in the right order, we need to use the *@Order* annotation.**

### **2.1. Filter With URL Pattern**

In the example above, our filters are registered by default for all of the URLs in our application. However, we may sometimes want a filter to only apply to certain URL patterns.

In this case, we have to remove the @Component annotation from the filter class definition and **register the filter using a *FilterRegistrationBean***:



Note, that in this case we need to explicitly set the order using a *setOrder()* method.

Now the filter will only apply for paths that match the */users/\** pattern.

**To set URL patterns for the filter, we can use the *addUrlPatterns()* or *setUrlPatterns()* methods.**

## 3. A Quick Example

Now let's create a simple endpoint and send an HTTP request to it:



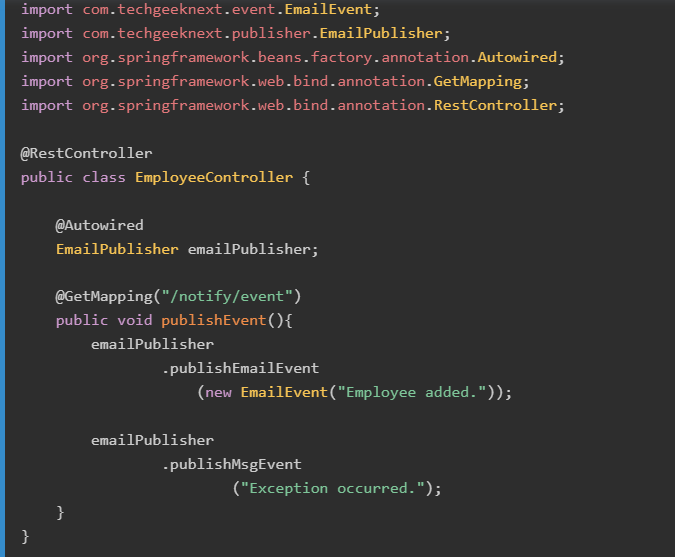
## **Spring Boot Event Listener**

1. A listener can be defined in two ways. Either the @EventListener annotation or the ApplicationListener interface can be used.
2. Spring enables to create and publish custom events that are **synchronous** by default. This means that the publisher thread is blocked until all listeners have completed processing the event.
3. To have an event listener execute in **async mode**, just add the @Async annotation to that listener.
4. We have created two listeners, one for listening Email Event and other for String message event.



## **Controller to test Spring Boot Event Listener**

Spring Boot Event Listener can be used for cross-cutting issues such as sending email in response to an event or performing a task if a business event/exception occurs in the application.



Ref:

<https://www.techgeeknext.com/spring-boot/spring-boot-event-listener-example#:~:text=Boot%20Event%20Listener-,A%20listener%20can%20be%20defined%20in%20two%20ways.,have%20completed%20processing%20the%20event>.

# Spring Session

## **1. Overview**

Spring Session has the simple goal of free up session management from the limitations of the HTTP session stored in the server.

The solution makes it easy to share session data between services in the cloud without being tied to a single container (i.e. Tomcat). Additionally, it supports multiple sessions in the same browser and sending sessions in a header.

In this article, we'll use Spring Session to manage authentication information in a web app. While Spring Session can persist data using JDBC, Gemfire, or MongoDB, we will use Redis.

## **2. A Simple Project**

What is Session Management?

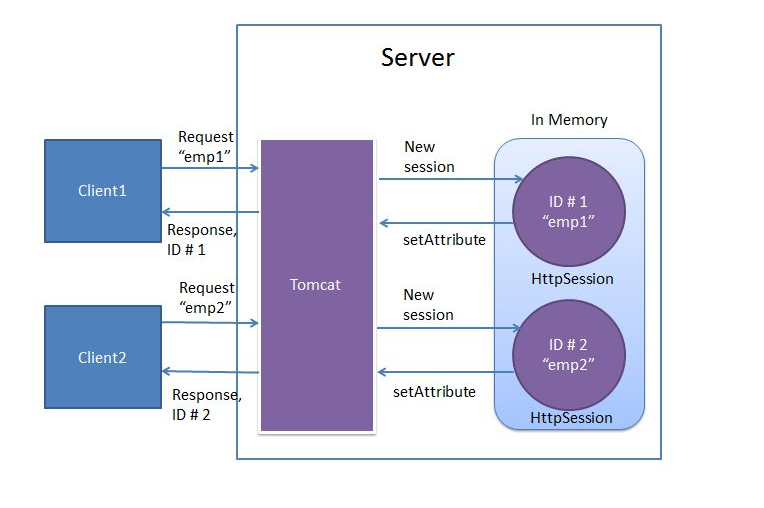
We all know that **HTTP is a stateless protocol**. All requests and responses are independent. The server cannot distinguish between new visitors and returning visitors. But sometimes we may need to keep track of client's activity across multiple requests. This is achieved using Session Management. It is a mechanism used by the Web container to store session information for a particular user.  
Session management can be achieved in one of the following ways-

* Cookies
* Hidden form field
* URL Rewriting
* HttpSession

In this example we will be making use of HttpSession to achieve Session management. Also we will be using the **Spring Session module**  
Spring Session consists of the following modules:

* **Spring Session Core -** provides core Spring Session functionalities and APIs
* **Spring Session Data Redis -** provides SessionRepository and ReactiveSessionRepository implementation backed by Redis and configuration support
* **Spring Session JDBC -** provides SessionRepository implementation backed by a relational database and configuration support
* **Spring Session Hazelcast -** provides SessionRepository implementation backed by Hazelcast and configuration support

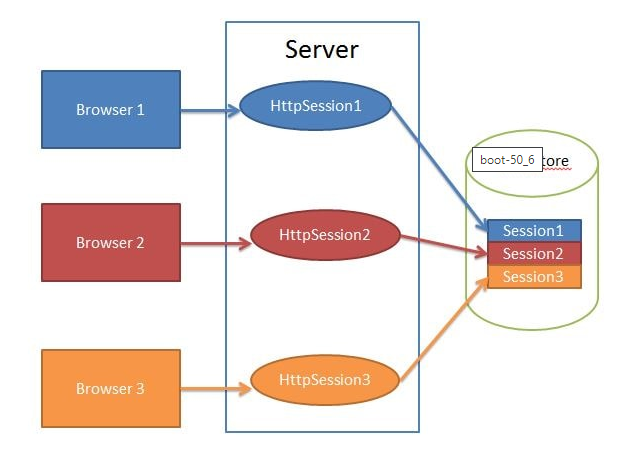
In this post we will be using Spring Session JDBC to store spring session information. By default Apache Tomcat stores HTTP session objects in memory.



In order to achieve writing the session objects to MySQL database, we dont have to write any code. Spring Boot provides us this functionality out of the box by specifying the following configuration property

**spring.session.store-type=jdbc**

Spring session replaces the HttpSession implementation by a custom implementation. To perform this task spring session creates a SessionRepositoryFilter bean named as springSessionRepositoryFilter.



Ref:

<https://www.javainuse.com/spring/springboot_session>

# How to Use Cookies in Spring Boot

An HTTP Cookie (also known as a **web cookie**or **browser cookie**) is a small piece of information stored by the server in the user's browser. The server sets the cookies while returning the response for a request made by the browser. The browser stores the cookies and sends them back with the next request to the same server. Cookies are generally used for session management, user-tracking, and to store user preferences.

Cookies help server remember the client across multiple requests. Without cookies, the server would treat every request as a new client.

In this tutorial, we will learn how to read, set, and remove HTTP cookies in a Spring Boot application.

## Reading HTTP Cookie

The Spring Framework provides the @CookieValue annotation to get the value of any HTTP cookie without iterating over all the cookies fetched from the request. This annotation can be used to map the value of a cookie to the controller method parameter.

1

@GetMapping("/")

2

public String readCookie(@CookieValue(value = "username", defaultValue = "Atta") String username) {

3

return "Hey! My username is " + username;

4

}

In the above code snippet, notice the defaultValue = "Atta". If the default value is not set, Spring will throw a java.lang.IllegalStateException exception on failure to find the cookie with name username in HTTP request.

## Setting HTTP Cookie

To set a cookie in Spring Boot, we can use HttpServletResponse class's method addCookie(). All you need to do is to create a new instance of Cookie class and add it to the response.



## Reading All Cookies

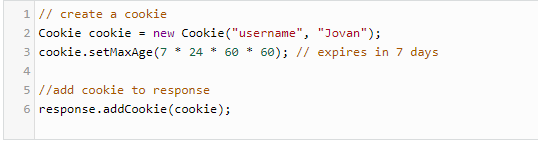
Instead of using @CookieValue annotation, we can also use HttpServletRequest class as a controller method parameter to read all cookies. This class provides getCookies() method, which returns all cookies sent by the browser as an array of Cookie.



## Cookie Expiration

If no expiration time is specified for a cookie, it lasts as long as the session is not expired. Such cookies are called **session cookies**. Session cookies remain active until the user closes their browser or clears their cookies. The username cookie created above is in fact a session cookie.

But you can override this default behavior and set the cookie expiration time using setMaxAge() method of Cookie class.

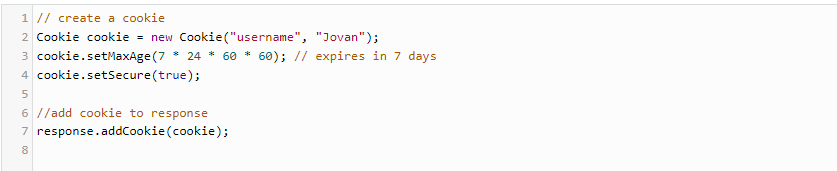


Now, instead of expiring when the browser is closed, the username cookie will remain active for the next 7 days. Such cookies, which expire at a specified date and time, are called **permanent cookies**.

**The expiry time passed to setMaxAge() method is in seconds. The expiry date and time is relative to the client where the cookie is being set, not the server.**

## Secure Cookie

A secure cookie is the one which is only sent to the server over an encrypted HTTPS connection. Secure cookies cannot be transmitted to the server over unencrypted HTTP connections.



## HttpOnly Cookie

HttpOnly cookies are used to prevent cross-site scripting (XSS) attacks and are not accessible via JavaScript's Document.cookie API. When the HttpOnly flag is set for a cookie, it tells the browser that this particular cookie should only be accessed by the server.



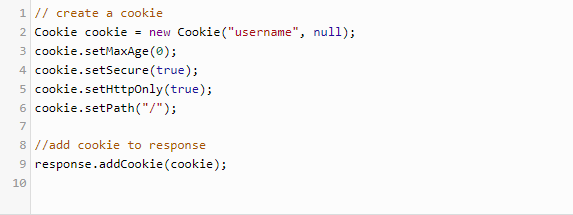
## Cookie Scope

If the scope is not specified, a cookie is only sent to the server for a path that was used to set it in the browser. We can change this behavior using setPath() method of the Cookie class. This sets the Path directive for the cookie.



## Deleting Cookie

To delete a cookie, set the Max-Age directive to 0 and unset its value. You **must** also pass the same other cookie properties you used to set it. Don't set the Max-Age directive value to -1. Otherwise, it will be treated as a session cookie by the browser.



## Summary

Cookies provide a way to exchange the information between the server and the browser to manage sessions (logins, shopping carts, game scores), remember user preferences (themes, privacy policy acceptance), and to track the user behavior across the site.

Spring Boot provides an easy way to read, write, and remove HTTP cookies.

* @CookieValue annotation maps the value of the cookie to the method parameter. You should set the default value to avoid runtime exception when the cookie is not available.
* HttpServletResponse class can be used to set a new cookie in the browser. You just need to create an instance of Cookie class and add it to the response.
* To read all cookies, you can use HttpServletRequest's getCookies() method which returns an array of Cookie.
* Max-Age directive specifies the date and time when the cookie should expire.
* If you are storing sensitive information in a cookie, make sure to set Secure and HttpOnly flags to avoid XSS attacks.
* Set the Path=/ to make a cookie accessible everywhere for the current domain.
* To delete a cookie, set the Max-Age to 0 and pass all the properties you used to set it.

Ref:

<https://dzone.com/articles/how-to-use-cookies-in-spring-boot>