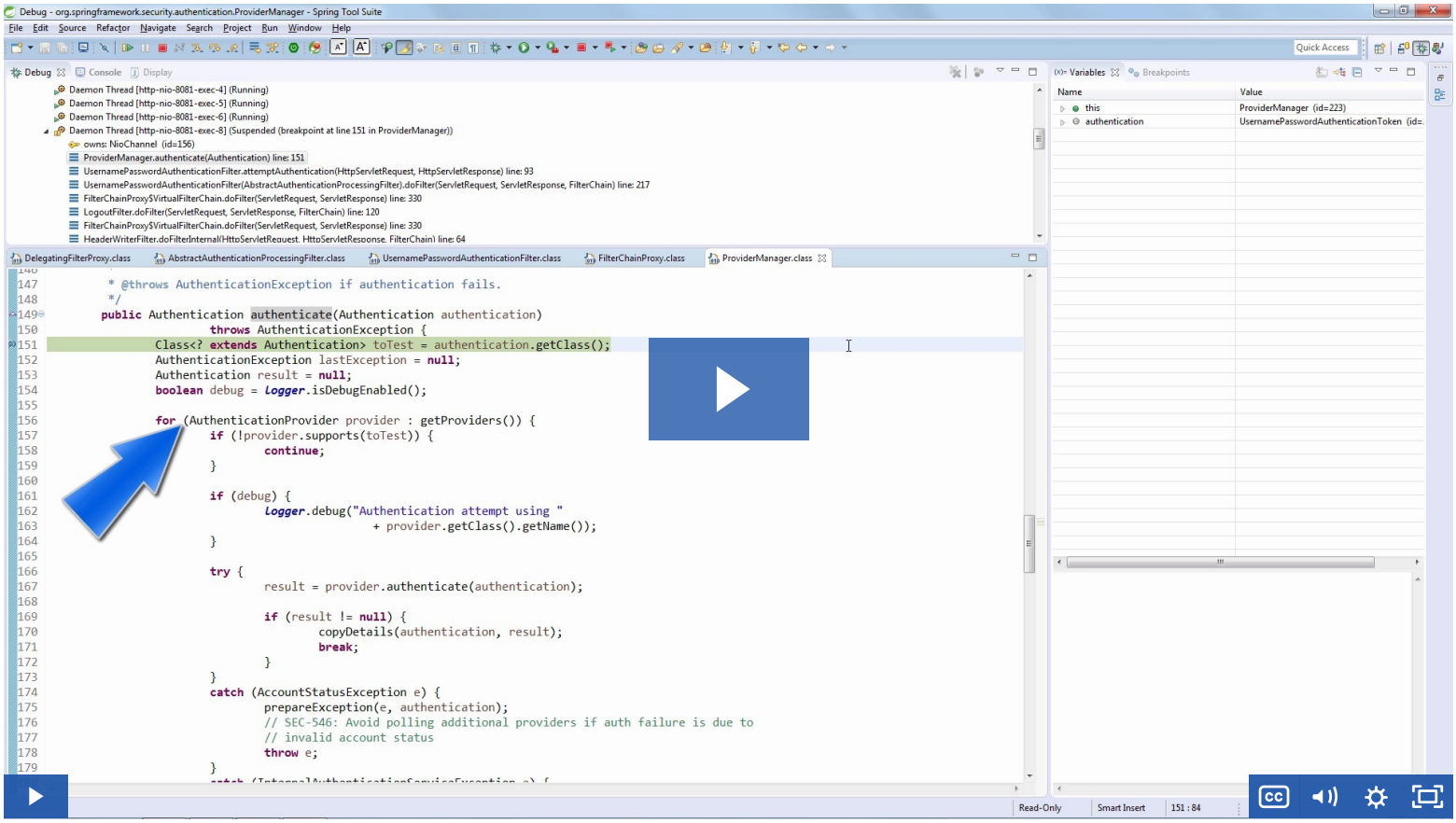
Lesson 1: Breaking Down the Authentication Flow



1. Goals

The goal of this lesson is to break down exactly what happens during an authentication flow in Spring Security.

2. Lesson Notes

2.1. From Tomcat to the UsernamePasswordAuthenticationFilter

We're finally going to take a step back and go through the full details of the auth process.

First - we hit the *DelegatingFilterProxy* - this is an actual Servlet filter.

This delegates to the *FilterChainProxy* - *doFilterInternal*.

Here we see the actual list of filters - we have 11 in our configuration.

Now - the execution of the filter chain starts (with the help of an internal chain implementation - the *VirtualFilterChain*).

The main method we're going to keep looking at is of course *doFilter* - and we'll follow the chain until we get to the *UsernamePasswordAuthenticationFilter*.

This is **the main filter we care about** here, as this is the one that's going to handle the authentication process (in our case - form login).

2.2. From the UsernamePasswordAuthenticationFilter to the AuthenticationManager

Here, we're going to go forward by doing an *attemptAuthentication*.

The username and password are obtained from the request and wrapped into a token.

This token is the beginning of the actual authentication process.

With the token, we now delegate to the Authentication Manager.

2.3. Inside the AuthenticationManager

The manager iterates over the Authentication Providers it has and tries to authenticate through them.

Notice that the auth manager is actually a hierarchy, so it will keep delegating to the parent if it can't authentication with its own providers.

In our case, it's the parent that has the main auth provider and can actually do the auth process.

2.4. Back to the Filter

On success - the auth manager returns a fully populated *Authentication* instance - back to the *UsernamePasswordAuthenticationFilter*

The filter continues with running the session strategy - that's going to depend on how you have sessions configured in the system.

Next, *successfulAuthentication* runs - this is where the security context is established:

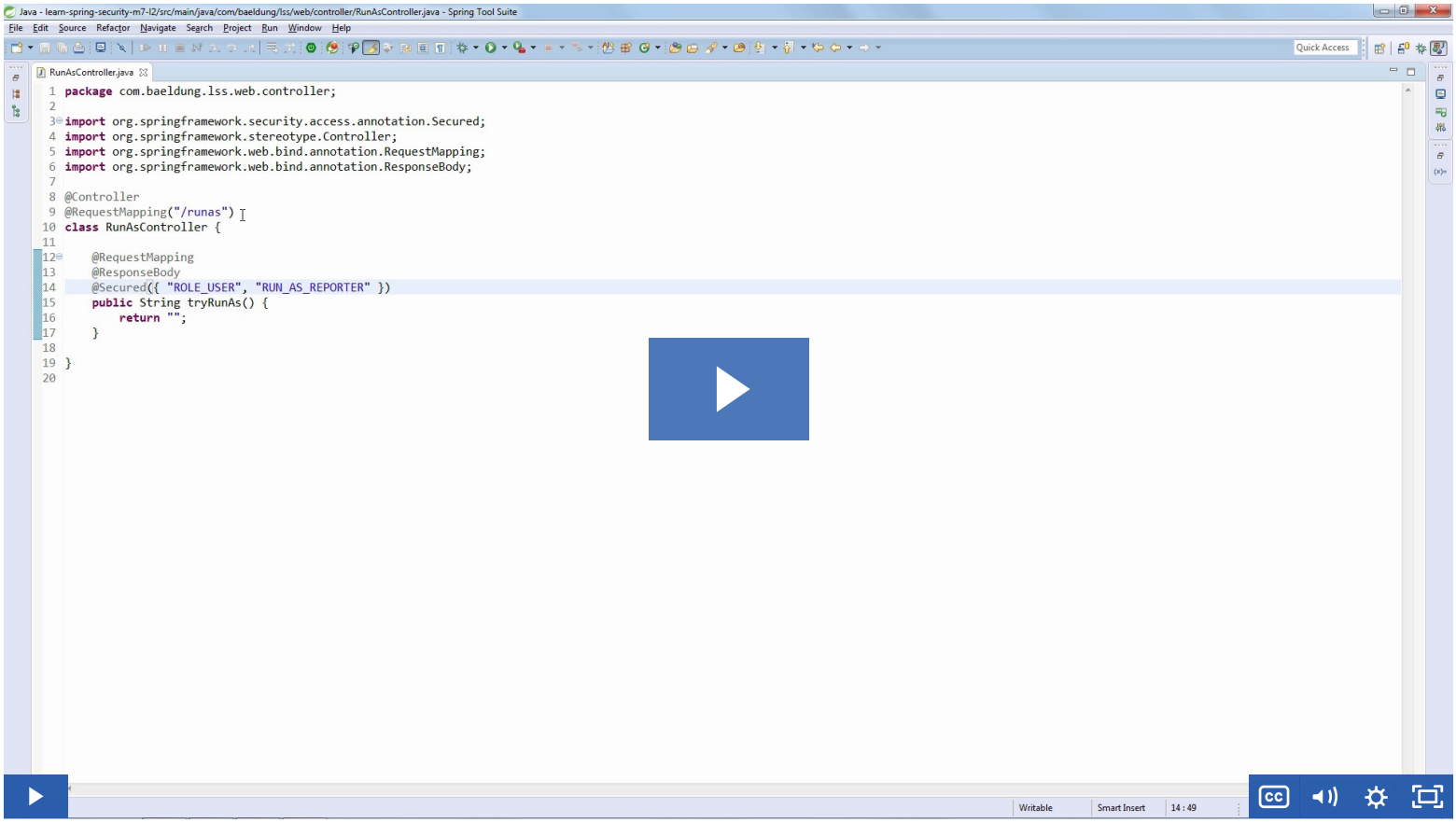
```
SecurityContextHolder.getContext().setAuthentication(…)
```

Next, the remember me logic runs as well; in our case, the "do nothing" implementation will run because we're not using remember me.

Finally, some post auth cleanup runs via the *successHandler*.

And we are done - this has been the entire authentication process driven by Spring Security.

Lesson 2: Run As a Different User



1. Goal

The goal of this lesson is to introduce you to the Run-As functionality in Spring Security and basically to show you how to run some operations under a different principal, with different authorities.

2. Lesson Notes

The relevant module you need to import when you're starting with this lesson is: m7-lesson2

If you want to skip and see the complete implementation, feel free to jump ahead and import: m7-lesson3

The credentials used in the code of this lesson are *test@email.com/pass* (PostConstruct)

2.1. High Level Run-As

First, let's discuss this high level concept of Run-As.

The simple goal is to replace the current (and successful) *Authentication* with another one that has a different set of authorities.

This is useful in a variety of scenarios - for example, systems that deal with calling remote services, but not only (as we'll see next).

2.2. Our Scenario

Another scenario where Run-As is useful is if you simply need to temporarily elevate the privileges of the current user for an one-off operation.

For example, let's say **we're generating a new report that needs to access more data** than the user may regularly need to see.

One way to do this would be - we could log out and log back in with a reporting user that has more authorities.

The Run-As functionality is another good way to go so that you don't have to go through this log out - log in process.

2.3. The Impl

The first thing we'll need to do is to configure method security:

```
@Configuration
@EnableGlobalMethodSecurity(securedEnabled = true)
public class MethodSecurityConfig extends GlobalMethodSecurityConfiguration {
    //
}
```

We're not going to define our Run-As auth manager bean:

```
@Override
protected RunAsManager runAsManager() {
    RunAsManagerImpl runAsManager = new RunAsManagerImpl();
    runAsManager.setKey("MyRunAsKey");
    return runAsManager;
}
```

Then we need to set up a new auth provider - the Run-As provider:

```
@Bean
public AuthenticationProvider runAsAuthenticationProvider() {
    RunAsImplAuthenticationProvider authProvider = new RunAsImplAuthenticationProvider();
    authProvider.setKey("MyRunAsKey");
    return authProvider;
}
```

And wire that in:

```
auth.authenticationProvider(runAsAuthenticationProvider());
```

Now we're going to create a new controller for this Run-As API

```
@Controller
@RequestMapping("/runas")
class RunAsController {
    //
}
```

And the operation:

```
@RequestMapping
@ResponseBody
public String tryRunAs() {
   //
}
```

We're going to secure the controller method with an extra role:

```
@Secured({ "ROLE_USER", "RUN_AS_REPORTER" })
```

We're going to now create a new service:

```
@Service
public class RunAsService {
    //
}
```

And secure that:

```
@Secured({ "ROLE_RUN_AS_REPORTER" })
public Authentication getCurrentUser() {
    Authentication authentication = SecurityContextHolder.getContext().getAuthentication();
    return authentication;
}
```

Notice the authority we're using to secure the service method here.

It's important to understand that the *RUN_AS_REPORTER* at the Controller level is just a marker role - not an actual role assigned to the user.

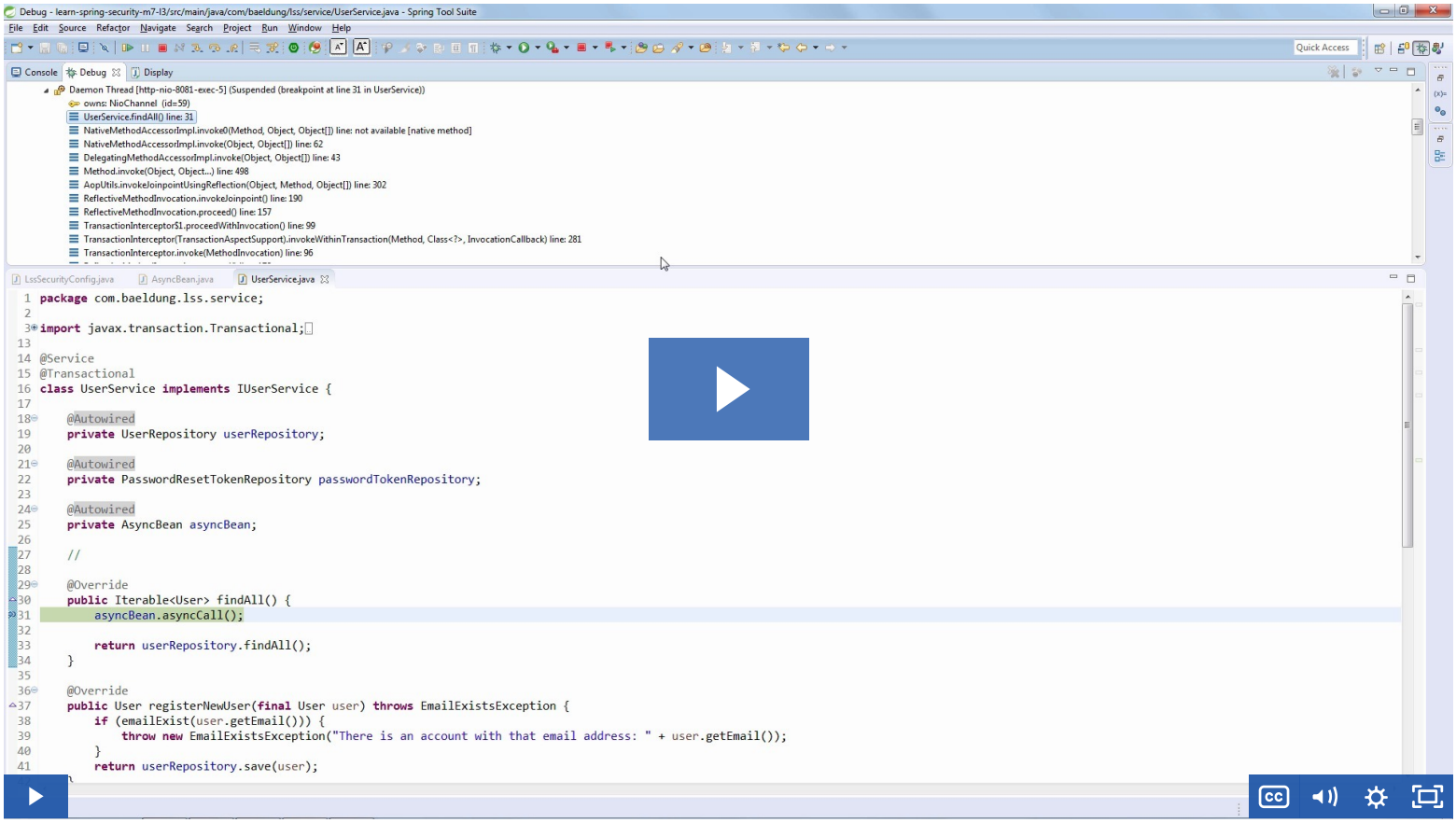
This is the core aspect of the Run-As logic.

This previous *RUN_AS* marker is converted to the new authority, receives the extra *ROLE_* prefix in the process, and **is now available on the current *Authentication* object.**

3. Resources

- Run-As Authentication Replacement in the Official Reference

Lesson 3: The Security Context



1. Goal

The goal of this lesson is to explain how the security context is established, stored and used across requests within the same session.

2. Lesson Notes

The relevant module you need to import when you're starting with this lesson is: m7-lesson3

If you want to skip and see the complete implementation, feel free to jump ahead and import: m7-lesson4

The credentials used in the code of this lesson are *test@email.com/pass* (PostConstruct)

2.1. The Security Context Basics

Let's start by defining our terms:

- the *SecurityContext* - the fundamental security information associated to the running thread
- the *SecurityContextHolder* - the storage mechanism for this security information.

As we saw in Lesson 1 - the *SecurityContext* is established on a successful authentication. This happens in: *AbstractAuthenticationProcessingFilter - successfulAuthentication*:

```
SecurityContextHolder.getContext().setAuthentication(authResult);
```

By default, the *SecurityContentHolder* uses a *ThreadLocal* to store these details, which means that it's transparently (and correctly) holding on to a context per thread.

The problem with that approach is that - if we're working with *@Async* - the new thread will no longer be able to access to the same principal as the main thread.

In order to sort this out. we'll need to change the default strategy.

We can do that either via a env property:

```
spring.security.strategy = MODE_INHERITABLETHREADLOCAL
```

Or programmatically on startup:

```
SecurityContextHolder.setStrategyName("MODE_INHERITABLETHREADLOCAL")
```

Finally, in order to test everything out, here's the simple way to get the info about the current user:

```
Object principal = SecurityContextHolder.getContext().getAuthentication().getPrincipal();
```

2.2. The Implementation

First, we're going to of course add the *@EnableAsync* on our configuration.

Then we're going to create a new bean, called *AsyncBean* - and we're going to define an *@Async* method in this bean.

And then we're going to wire the bean into the *UserService* and use it there.

With the help of a couple of breakpoints, we can easily follow exactly what happens both outside and inside the new async method.

If we run the system and have a look at the state of things inside the async thread:

```
org.springframework.security.core.context.SecurityContextHolder.getContext().getAuthentication()
```

We're going to see that there's no *Authentication* - which is of course to be expected.

After we configure the system and change the default strategy, we're going to run the system again.

And we're going to be able to see that now the *Authentication* is there and so it has been propagated correctly from the parent thread into this async thread.

2.3.

Now, let's discuss how the Security Context is maintained between user operations / requests.

In a typical MVC app, after login, the user is identified by its session id. Remember that the management of the context is done by the *SecurityContextPersistenceFilter*.

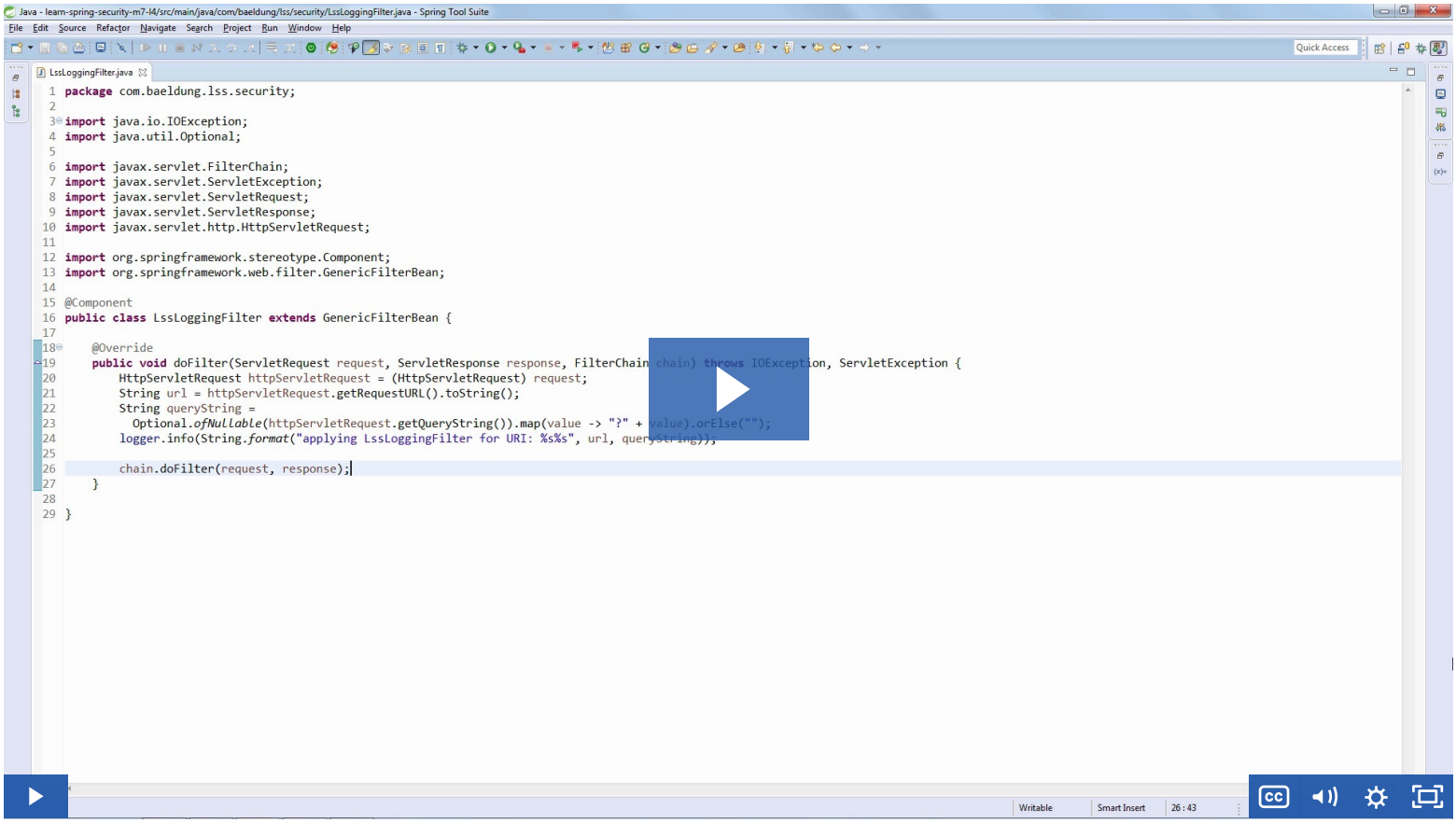
By default, it stores the context as an attribute of the HTTP session, and it then restores it for each request and clears it when the request ends.

Finally, a quick side-note. Even if we have a REST API and that's a stateless system with no session, **we still need this filter for this clear logic**.

3. Resources

- Storing the SecurityContext between requests in the Official Reference

Lesson 4: Configure the Filter Chain



1. Goal

The goal of this lesson is to show you how to go beyond the higher level configuration and do some low level work by configuring the filter chain of the security framework.

2. Lesson Notes

The relevant module you need to import when you're starting with this lesson is: m7-lesson4

If you want to skip and see the complete implementation, feel free to jump ahead and import: m7-lesson5

The credentials used in the code of this lesson are test@email.com/pass (PostConstruct)

First off, we're going to review the default filter chain by debugging through an authentication request.

We're going to stop in the last filter in the chain: *FilterSecurityInterceptor - doFilter*

And we're going to look at the chain (*additionalFilters*) - these are all the filters and they run in this exact order.

Next, we're going to define a new filter:

```
@Component
public class LssLoggingFilter extends GenericFilterBean {
    //
}
```

The implementation will be very simple:

```
log.info("Running through the new custom filter");
filterChain.doFilter(servletRequest, servletResponse);
```

And let's actually do something marginally useful - log a request:

```
HttpServletRequest httpServletRequest = (HttpServletRequest) servletRequest;
String url = httpServletRequest.getRequestURL().toString();
String queryString =
  Optional.ofNullable(httpServletRequest.getQueryString()).map(value -> "?" + value).orElse("");
log.info(String.format("applying LssLoggingFilter for URI: %s%s", url, queryString));
```

Now, let's use the filter in our security config.

First, we need to wire it in:

```
@Autowired private LssLoggingFilter lssLoggingFilter;
```

Then, we'll place the filter in the chain:

```
.addFilterBefore(lssLoggingFilter, AnonymousAuthenticationFilter.class)
```

Notice the API here - *addFilterBefore* - it's position based. We also have other APIs available here - *addFilterAfter* and *addFilter* (which will add the filter to its default order if it's a known filter).

Finally - we're going to run the system and see the new filter in the filter in the chain, right before the anonymous filter.

3. Resources

- Adding in Your Own Filters in the Official Reference