Spring Boot Security

☑ Favorite	☐
↗ Tag	Securing a REST API with OAuth 2.0

▼ Authentication Instance

```
JwtAuthenticationToken
[Principal=org.springframework.security.oauth2.jwt.Jwt@da5265
[PROTECTED], Authenticated=true, Details=WebAuthenticationDet
[RemoteIpAddress=127.0.0.1, SessionId=null], Granted Authorit
[SCOPE_cashcard:read, SCOPE_cashcard:write]]
```

A principal → set of claims
A credential → original, signed JWT, and
A set of authorities → Each scope prefixed by SCOPE_

▼ Accessing Authentication in Spring MVC

▼ Principal Type Conversion

1. **Authentication Principal**: In Spring Security, the authenticated user is represented by the "principal." This could be a user object, a JWT token, or just a username.
2. **Reducing Boilerplate**: Instead of manually extracting information from the authentication object (like username or JWT details), Spring provides annotations like `@CurrentSecurityContext`.
3. `@CurrentSecurityContext`: This annotation simplifies getting user-related info from the authentication object. For example, if you need the username or JWT details, you can use `@CurrentSecurityContext` to get them directly.
4. **Example with JWT**: If you're using JWT authentication, `@CurrentSecurityContext` can directly give you the JWT object without needing to manually extract it from the authentication object.

5. **SpEL Expressions**: These are used in `@CurrentSecurityContext` to specify what information you want from the authentication object. It's like telling Spring Security, "Hey, I need this specific piece of information from the authenticated user."
6. **Meta-annotations**: To avoid repetition and potential errors in SpEL expressions, Spring allows you to create custom annotations that encapsulate commonly used expressions. This helps keep your code cleaner and more secure.

In essence, `@CurrentSecurityContext` simplifies accessing user-related information in Spring MVC controllers, making your code cleaner and safer by reducing boilerplate and potential errors.

▼ What are cross cutting concerns

Imagine you're building a really cool treehouse with your friends. Each friend has a specific job to do – one paints the walls, another installs the windows, and so on. But there are some things that affect everyone, no matter what job they're doing. These are called "cross-cutting concerns."

Here's an example: Safety. Making sure the treehouse is safe is important for everyone, not just the person building the floors or the person painting. Safety is a concern that "cuts across" all the different jobs – it's something everyone has to think about.

In software, it's kind of like that too. When you're building a program, there are certain things that affect the whole program, not just one part of it. These are cross-cutting concerns.

For instance, security is a cross-cutting concern. It doesn't matter if you're working on the login page or the shopping cart feature – you need to make sure the whole program is secure from hackers or bad actors.

Another example is logging. You might want to keep track of what's happening in your program – like who's logging in, what actions they're taking, and any errors that occur. Logging is something that affects the entire program, not just one part of it.

So, cross-cutting concerns are basically those important things that apply to the whole program, not just specific parts of it. They're like the safety and

organization aspects of building a treehouse that everyone has to pay attention to, regardless of their specific job.

▼ Caching Headers

When you visit a website, your web browser downloads a bunch of stuff to show you the page – like images, text, and other data. Sometimes, this data gets saved in a temporary storage area called a cache, so if you visit the same website again, your browser can load it faster.

Now, imagine you're using a banking app, and it shows your account balance. After you log out, you'd expect that information to disappear, right? But sometimes, even after you log out, the banking app's data can stay in your browser's cache. This is a problem because someone else who uses the same computer could peek into your banking info.

In the example you mentioned, the REST API returns some sensitive data – like account balances – which could get saved in the browser's cache. Even if you log out or close the app, that data might still be hanging around in the cache, waiting to be seen by someone else.

To prevent this from happening, Spring Security sets certain rules for the cache, like saying "Don't store this data" or "Make sure this data expires quickly." These rules help keep sensitive information from getting stuck in the browser's cache where it shouldn't be.

So, Spring Security's secure settings for cache control help protect your sensitive data by making sure it doesn't get saved where it shouldn't be, reducing the risk of unauthorized access or exposure.

▼ Requires CSRF Mitigation for All Requests With Side-Effects

Imagine you have a secret code that you use to access your bank account. Normally, you keep this code safe and only use it when you want to do something with your account, like checking your balance or transferring money.

Now, imagine you're browsing a website, and suddenly, a hidden button appears that says "Get Free Stuff!" You click it out of curiosity, but behind the scenes, it's actually sending a request to your bank with your secret code, asking to transfer money to someone else's account!

This is a big problem because your browser automatically sends your secret code (known as cookies and authentication details) to any website you visit, even if it's just to load an image. So, if a sneaky website includes an image tag like the one in the example, your browser will happily send your secret code along with it.

To prevent this, Spring Security adds a special token to your browser when you first log in. This token is like a secret handshake that only you and the bank know about. When you try to do something important with your account, like transferring money, Spring Security checks if you have this token. If you do, it knows that you're really you and not some sneaky website trying to trick your browser.

So, by using this special token, Spring Security protects your bank account and other important actions from being accessed by unauthorized websites, keeping your information safe and secure.

▼ Allows HTTP Basic Authentication With a Default User

Imagine you're building a secret club, and you want to make sure only the right people can get in. So, you decide to have a special password that members need to know to enter.

In the world of software, Spring Security acts like the gatekeeper of your secret club. It helps control who can access your application and what they can do inside.

Now, imagine you set up your club, but you forget to change the default password from "password123" to something more secure. Anyone who knows this default password could potentially get into your club and cause trouble.

To prevent this, Spring Security generates a default user with a random password each time your application starts up. This means that even if you forget to change the password, it's different every time, making it much harder for someone to guess.

So, by using this approach, Spring Security ensures that your application is secure by default. It's like having a lock on your door that changes its combination every time you use it, so even if someone knows the default combination, it won't work next time.

And if you ever need to find out what the password is, you can look in the startup logs of your application, where Spring Security will tell you what the current password is. This way, you can still access your club if you need to, but it remains secure from unauthorized access by others.