Securing A

REST API

With OAuth 2.0

Module 1: Secure Defaults

Secure Defaults

Spring Security Defaults:

When spring security is on the classpath, Spring Boot waits to configure your application with the following defaults for a REST API:

- requires authentication for all requests
- responds with secure headers for all requests
- requires CSRF mitigation for all requests with side effects
- Allows for HTTP basic authentication with a default user
- Responds RESTfully to security failures
- Protects against malicious requests with an application firewall

Servlet, Filters and Dispatcher

A given HTTP request can pass through multiple dispatches. Each dispatch can be intercepted by multiple filters on its way to a single servlet.

- A servlet handles HTTP request and yields an HTTP response
- A filter intercepts HTTP request to handle cross-cutting concerns
- A dispatch represents a single pass an HTTP request walks through a set of filters and its target servlet

Responds with secure headers for all requests

- Spring Security always responds with certain headers by default

Caching Headers : Spring Security applies secure settings for Cache-Control

Strict Transport Security Header : forces a browser to upgrade requests to HTTPS for a specified period of time

Content Type Options : tells browsers not to try to guess content type of a response

Requires CSRF Mitigation for All requests with side-effects

- prevents third-party web sites from making requests without user consent.

Allows HTTP Basic Authentication with a default user

- creates password for default user

Module 2. Authentication

Adding Authentication

Content Negotiation

- Spring Security's default settings contain the principal's identity using the Form Login and HTTP Basic authentication scheme

- HTTPie http command will add the Authentication: Basic header for you if you pass it the -a parameter.

Authentication Process

Three parts:

1. Parses the request material into a credential
2. Tests that credential
3. If credential passes it translates that credential into a principal and authorities

Those three steps in action:

1. Spring Security decodes the Base64 - encoded username and password. Password is the credential
2. Tests username and password against a user store
3. If password matches, loads corresponding user and permissions and stores them in its security context

Authentication Result

- result is an Authentication instance

Authentication:

- principal (who)
- credentials (proof)
- authorities (permissions)

Subsequent Request

- Some authentication schemes are stateful / stateless

Stateful : your application remembers information about previous requests

Stateless : application remembers nothing

Limits of HTTP Basic

Limitations :

- long-term credentials
- authorization bypass
- sensitive data exposure

Long-Term Credentials

- anyone who steals your username and password can impersonate you

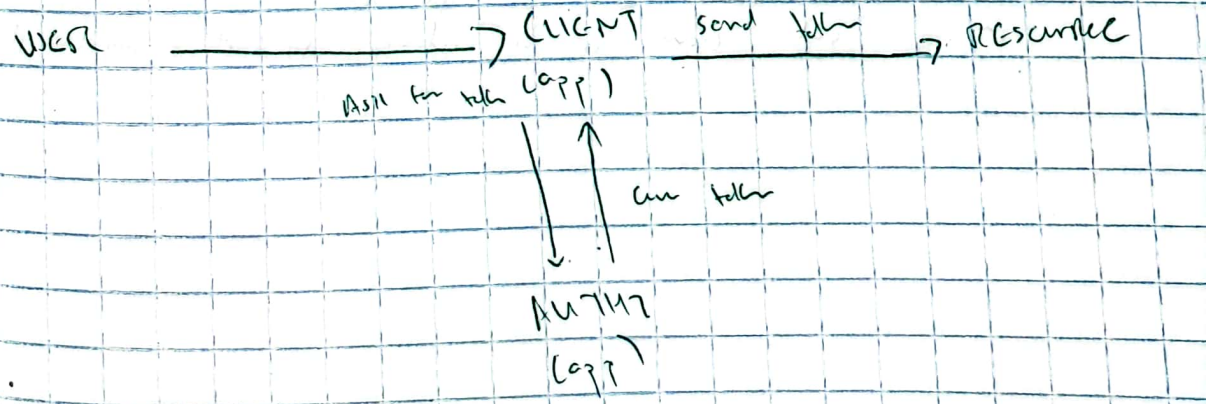
OAuth 2.0 and JWT

OAuth 2.0

An industry-standard protocol for authorization that has been adopted by thousands of companies and used in millions of applications

Involves three actors :

- the client application, which need your data and protected your server
- the resource server, holds and secure your data
- Authorization server, authorizes a client to access your data



JSON Web Token

industry standard format for encoding claims tokens

A decoded JWT at its most basic is a set of headers and claims:

- headers contain metadata about the token, like how a resource server should process it
- claims are facts that the token is asserting, like what principal the token represents

Eg:

```
{
  "typ": "JWT",
  "alg": "RS256"
}
```

+ headers

```
{
  "aud": "https://content.example.org",
  "exp": 1689364988,
  "iat": 1689361738,
  "iss": "https://issuer.example.org",
  "scp": ["content.read", "content.write"],
  "sub": "spruce"
}
```

- iss claim identifies the authorization server that minted the token
- exp claim indicates when token expires
- scp claim indicates the set of permissions authorization server granted
- sub claim is a reference to the principal the token represents

Spring Security Support

For resource server, steps on how Spring security process authentication:

1) Parse request method into a credential. Spring Security looks for Authorization: Bearer {jwt}

2) Tests that credential. Uses a JwtDecoder instance to query the authorization server for keys, uses those keys to verify the JWTs signature and validate its from a trusted issuer and is still within expiry window

3) If credential passes it branches that credential into a principal and authorities

Accessing Authentication in Spring MVC

Method Injection

- can get current Authentication instance in any spring MVC handler method by including it as a method parameter like so:

```
@GetMapping
public ResponseEntity<Iterable<Cocktail>> findAll(Authentication authentication){...}
```

- Authentication #getPrincipal return object

Principal Type Conversion

- the @CurrentSecurityContext annotation allows you to remove some of the boilerplate around getting specific values like the principal from current authentication

- For example with Bearer JWT Authentication, Authentication #getPrincipal holds a Jwt instance. Can get underlying Jwt instance:

```
@GetMapping
public ResponseEntity<Iterable<Cat>> findAll(@CurrentSecurityContext
        (expression = "authentication.principal")
        Jwt jwt) { }
```

- not all @CurrentSecurityContext annotations only produced by Controller methods

- you can obtain any attribute from the Authentication instance that you need. For example, the above snippet can be simplified to call Authentication#getName like so:

```
@GetMapping
public ResponseEntity<Iterable<Cat>> findAll(@CurrentSecurityContext
        (expression = "authentication.name")
        String owner) { }
```

Meta-Annotations

- A meta-annotation is an annotation that aliases another annotation

Eg: create a custom annotation called @CurrentOwner that extracts owner from the name:

```
@Target(ElementType.PARAMETER)
@Retention(RetentionPolicy.RUNTIME)
@CurrentSecurityContext(expression = "authentication.name")
public @interface CurrentOwner { }
```

Can use it like so:

```
@GetMapping
public ResponseEntity<Iterable<Contact>> findAll(@CurrentSecurityContext ...)
```

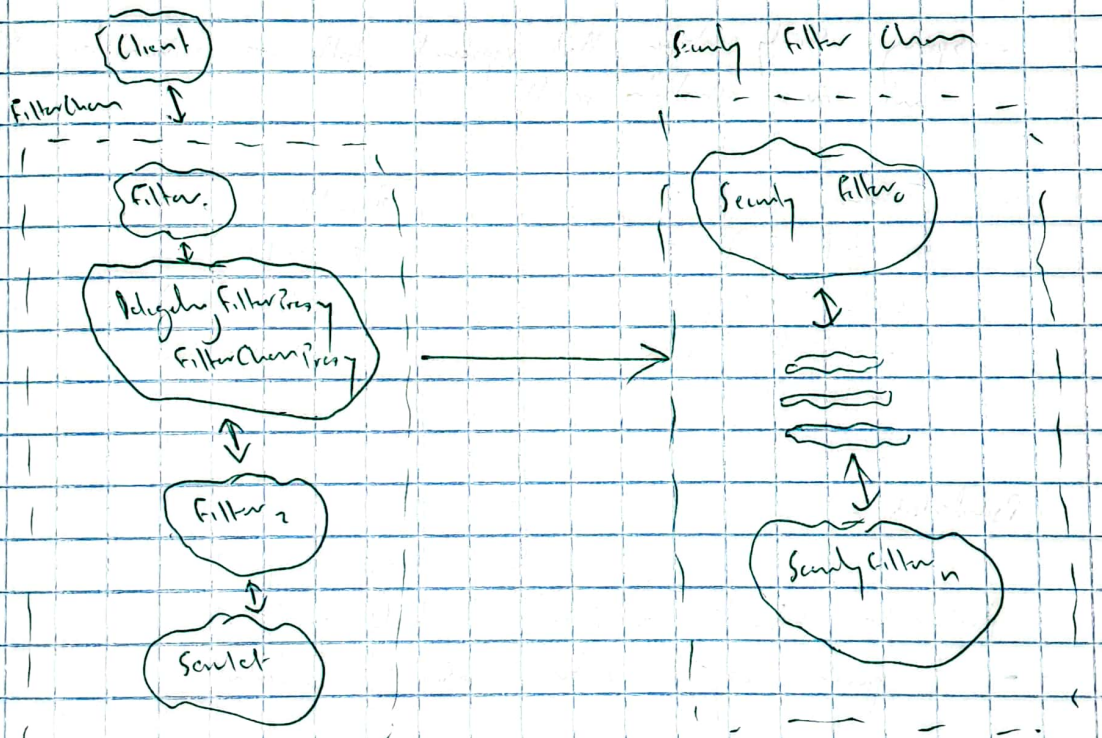
@AuthenticationPrincipal

```
@AuthenticationPrincipal
```

- equivalent of: `@CurrentSecurityContext(expression="authentication.principal")`

The big picture

A The filter Chain



Filter Chain can be separated into four categories:

- Defense Filters
- Authentication Filters
- Authorization Filters
- Instrumentation Filters

Default Filters

- Csrf Filter
- Header Filter

Authentication Filters

- Basic Authentication Filter : handle HTTP basic authentication
- Bearer Token Authentication Filter : handle Bearer token authentication (JWTs)
- Username Password Authentication filter : handle form login authentication
- Anonymous Authentication filter : populate context with a Null object authentication instance

Authentication

- Spring Security interface that represent both an authentication token and an authentication result

Token	Result
principal (who)	principal (who)
credentials (proof)	credentials (pr.)
authenticated = false	authenticated = true

Pseudo code :

```
Authentication token = getAuthenticationRequest(request)
```
- parse request method into a credential

```
Authentication result = authenticationManager.authenticate(token)
```
- tests that credential and return a principal and authorities
- contains the principal and authorities

Authentication Manager

- interface that tells an authentication filter

SecurityContext

- an object that holds the current Authentication

Reviewing Bearer JWT Authentication

① BearerTokenAuthenticationFilter extracts the JWT into a JwtAuthenticationToken instance

② Passes that authentication token to an AuthenticationManager. The manager holds a JwtAuthenticationProvider instance

③ JwtAuthenticationProvider authenticates the JWT and returns an authenticated instance of JwtAuthenticationToken that includes the parsed JWT and granted authorities

④ Finally stores authentication result in a SecurityContext instance

Authentication Filters

- any request that contains a bearer token in Authentication: Bearer, Spring Security will attempt to authenticate.

The SecurityFilterChain Bean

- when you wish for a reference to HttpSecurity, you typically only need to specify your authentication and authorization rules like so:

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

The Default OAuth 2.0 Resource Server Bean

- since our Cash Card app is an OAuth 2.0 server it should instead declare an OAuth2 Resource Server for authentication instead of http basic like this:

@Bean
SecurityFilterChain securityFilterChain(HttpSecurity http) throws exception {
    http.
        oauth2ResourceServer((oauth2) -> oauth2
            .jwt(withDefaults())
)
        .authorizeHttpRequests((authorize) -> authorize
            .anyRequest().authenticated()
        );
    return http.build();
}

Accessing Authentication Anywhere

// The SecurityContextHolder

- eg: when you're creating a servlet filter, which executes before Spring MVC

Using it static class

SecurityContext context = SecurityContextHolder.getContext();
Authentication authentication = context.getAuthentication();

- this snippet uses SecurityContextHolder, a static class that provides access to the current security context, including the authentication details. You can access it anywhere

A Companion to Spring MVC Method Argument

To implement findAll:

```
@GetMapping
public ResponseEntity<Iterable<CashCard>> findAll(Authentication authentication) {
        var result = this.cashCards.findByOwner(authentication.getName());
        return ResponseEntity.ok(result);
}
```

To use SecurityContextHolder instead:

```
@GetMapping
public ResponseEntity<Iterable<CashCard>> findAll() {
    Authentication authentication = SecurityContextHolder.getContext().getAuthentication();
    return ResponseEntity.ok(this.cashCards.findByOwner(authentication.getName()));
}
```

What about POJOs?

SecurityContextHolder uses a ThreadLocal to store the security context which means it's specific to current thread. Therefore can access authentication information with the same thread throughout your application

Eg: Consider the theoretical CashCardDiscount-calculation class that uses the currently authenticated user to help determine if they qualify for a special discount:

```
public class CashCardDiscountCalculator {
    public boolean CurrentUserQualifiesForDiscount() {
        Authentication authentication = SecurityContextHolder.getContext().getAuthentication();
        return qualifiesForDiscount(authentication.getName());
    }

    private boolean qualifiesForDiscount(String name) {}
}
```

Validating Claims

By default Spring Security authenticates JWT by:

1) Validating the signature
2) Checking that current time is between timestamps in "nbf" (issued At) and "exp" (Expired At) claims

Best practice properties to simply adding these two validation steps:

1) issuer-uri:

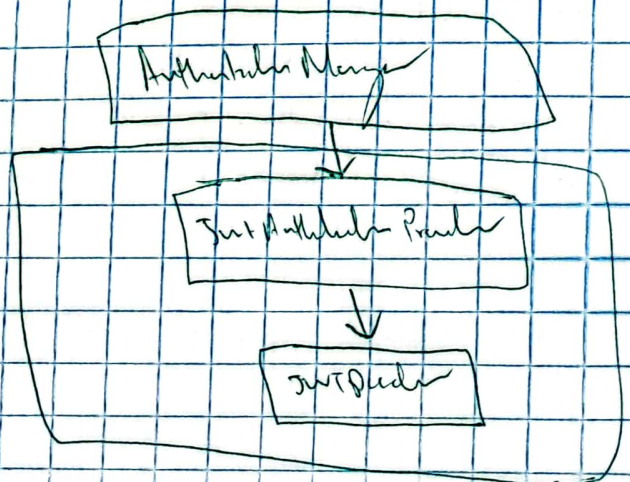
```
spring:
  security:
    oauth2:
      resourceserver:
        jwt:
          issuer-uri: https://issuer.example.org
```

2) audiences:

```
spring:
  security:
    oauth2:
      resourceserver:
        jwt:
          audiences: https://client.example.org
```

Custom Validation

For authenticating JWTs, there is a more specific component called JwtDecoder:



Default implementation of JwtDecoder is NimbusJwtDecoder

Eg: Can specify custom validation steps by creating your own NimbusJwtDecoder like so:

```
@Bean
JwtDecoder jwtDecoder(String issuer, String audience) {
    OAuth2TokenValidator<Jwt> defaults = JwtValidators.createDefaultWithIssuer(issuer);
    OAuth2TokenValidator<Jwt> audience = new JwtClaimValidator<List<String>>
            (AUD, (aud) -> aud != null && aud.contains(audience));
    OAuth2TokenValidator<Jwt> all = new DelegatingOAuth2TokenValidator<>
            (defaults, audience);
    NimbusJwtDecoder jwtDecoder = NimbusJwtDecoder.withIssuerLocation(issuer).build();
    jwtDecoder.setOAuth2TokenValidator(all);
    return jwtDecoder;
}
```

Processing features:

AuthenticationEntryPoint

- If client is a browser, Spring Security will use LoginUrlAuthenticationEntryPoint
- else, if client is an HTTP Basic client then spring security will use the BasicAuthenticationEntryPoint
- else if the client is a Bearer token client, spring security will use BearerTokenAuthenticationEntryPoint

Authentication Filters

- filter will invoke the correct entry point.

Module 3: Authorization

Authorization Models Overview: Request vs Method

A Request-level Authorization
- two main ways to model your authorization: ① request level
② Method level

At request level:
- If request follows pattern X then require authority Y to honour request
- request: any part of HTTP request
- pattern: matching rule
- authority: authorization rule

At Method level
- activate it by adding @EnableMethodSecurity annotation to any Spring @Configuration class

Eg: can configure CashCardController#findAll to require cashcard:read permission by adding @PreAuthorize annotation

```
@GetMapping("/cashcard")
@PreAuthorize("hasAuthority('cashcard:read')")
public ResponseEntity<Iterable<CashCard>> findAll(@CurrentOwner String owner)
    { //... }
```

What request security can do that method security can't?

Consider:

```
@Bean
SecurityFilterChain appSecurity(HttpSecurity http) throws Exception {
  http
    //...
    .authorizeHttpRequests((authz) -> authz
      .requestMatchers("/css/**", "/js/**").permitAll()
      .requestMatchers("/admin/**").hasRole("ADMIN")
      .anyRequest().authenticated()
    //...
}
```

This configuration does three things that security method cannot do:

1. Protects endpoints that don't resolve to a method invocation
2. Protects entire sections of a matrix in a single declaration
3. Provides a catch-all for when new endpoints are introduced to the application

What Method Security can do that request security can't?

- method security can secure the method invocation as well as method return, request security can only secure request, not response

But really, you need both

- coarse-grained authorization: authorization rules that match a claim on only a single factor
- fine-grained authorization: authorization rules that match a claim based on multiple factors

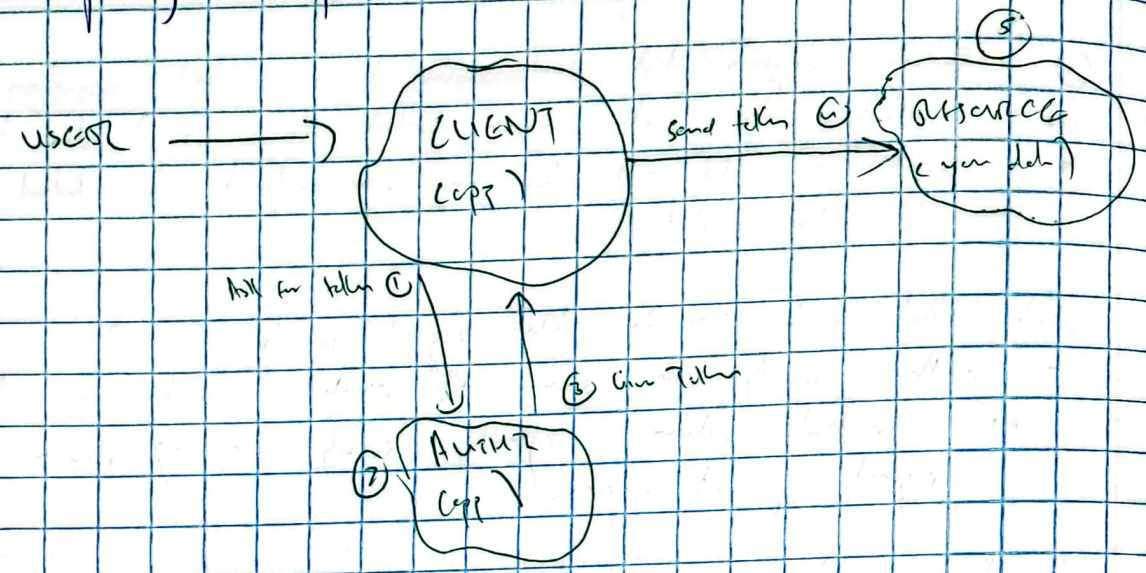
- by design, method security is great for fine-grained and request great for coarse-grained

OAuth 2.0 Scopes

```
Scp Claim
  {
    "aud": "https://content.example.com",
    "exp": 1618781111,
    "iat": 1618778211,
    "iss": "https://issuer.example.com",
    "scp": ["content:read", "content:write"],
    "sub": "serval"
  }
```

- Scp claim is something as seen as "scope" claim
- set of permissions the token has

Regarding Authorization



We're in step 2:

① Client knows what part of your data it needs to fetch, asks authorization server for permission to access or operate & that data

② Authorization server in turn calls you & ask user if you are okay with it

③ If you say yes authorization server creates a JWT which app client retrieves with permissions you decided to grant.

Granting Authorization

- scopes are granted authorizations