ch4mpy / spring-addons

Code | Issues 4 | Pull requests | Discussions | Actions | Projects | Security

spring-addons / samples / tutorials / reactive-resource-server /

Add file

ch4mpy [maven-release-plugin] prepare for next development iteration ✓ f0b2ae9 · 2 days ago

Name	Name	Last commit date
..		
src	Option to force PKCE usage, eve...	4 months ago
README.md	fix some typos in several READ...	8 months ago
pom.xml	[maven-release-plugin] prepare...	2 days ago

README.md

Configure a Reactive OAuth2 Resource Server (REST API)

In this tutorial, we'll configure a reactive (WebFlux) Spring Boot 3 application as an OAuth2 resource server with authorities mapping to enable RBAC using roles defined on OIDC Providers, *without `spring-addons-starter-oidc` **, which makes quite more verbose compared to the "webflux" projects in samples.

We'll also see how to accept access tokens issued by several, potentially heterogeneous, OIDC Providers (or Keycloak realms).

0. Disclaimer

There are quite a few samples, and all are part of CI to ensure that sources compile and all tests pass. Unfortunately, this README is not automatically updated when source changes. Please use it as a guidance to understand the source. **If you copy some code, be sure to do it from the source, not from this README.**

1. Project Initialization

We start after prerequisites, and consider that we have a minimum of 1 OIDC Provider configured (2 would be better) and users with and without `NICE` role declared on each OP. As usual, we'll start with http://start.spring.io/ adding the following dependencies:

- Spring Reactive Web

- OAuth2 Resource Server
- lombok

2. REST Controller

We'll use a very simple controller, just accessing basic `Authentication` properties:

```
@RestController
public class GreetingController {

    @GetMapping("/greet")
    public Mono<MessageDto> getGreeting(Authentication auth) {
        return Mono.just(new MessageDto("Hi %s! You are granted with: %s.".formatted(a
    }

    static record MessageDto(String body) {
    }
}
```

This is enough to demo that the username and roles are mapped from different claims depending on the authorization-server which issued the access token (Keycloak, Auth0 or Cognito).

3 Security Configuration

This is how we want our REST API to be configured:

- use OAuth2 for requests authorization
- accept identities issued by 3 different OIDC authorization-servers (Keycloak, Auth0 and Cognito)
- enabled CORS (with configurable allowed origins)
- state-less session management (no session, user state in access token only)
- disabled CSRF (safe because there is no session)
- public access to a limited list of resources
- non "public" routes require users to be authenticated, fine-grained access-control being achieved with method-security (`@PreAuthrorize` and alike)
- 401 (unauthorized) instead 302 (redirecting to login) when a request to a protected resource is made with missing or invalid authorization

3.1. Security Properties

Let's replace our `application.properties` file with an `application.yml` having the following content:

```
scheme: http
origins: http://localhost:4200,https://localhost:4200
permit-all: /public/**
keycloak-port: 8442
```

```yaml
keycloak-issuer: ${scheme}://localhost:${keycloak-port}/realms/master
cognito-issuer: https://cognito-idp.us-west-2.amazonaws.com/us-west-2_RzhmgLwjl
auth0-issuer: https://dev-ch4mpy.eu.auth0.com/

server:
  error:
    include-message: always
  ssl:
    enabled: false

spring:
  security:
    oauth2:
      resourceserver:
        jwt:
          issuer-uri: ${keycloak-issuer}

---
scheme: https
keycloak-port: 8443

server:
  ssl:
    enabled: true

spring:
  config:
    activate:
      on-profile: ssl

---
spring:
  config:
    activate:
      on-profile: auth0
  security:
    oauth2:
      resourceserver:
        jwt:
          issuer-uri: ${auth0-issuer}

---
spring:
  config:
    activate:
      on-profile: cognito
  security:
    oauth2:
      resourceserver:
        jwt:
          issuer-uri: ${cognito-issuer}
```

There are a few things worth noting here:

- `spring.security.oauth2.resourceserver.jwt.issuer-uri` is single valued. This means that Spring Boot accepts OAuth2 identities issued only by a single OIDC Provider at a time. We use profiles to switch our resource server from one OP to another. This requires a restart, but we'll see how to configure spring-security to accept identities from several OIDC Providers in a single resource server instance.

- there are values (issuer URIs) which need an edit with what was obtained when accomplishing prerequisites.

3.2. Security Filter-Chain

As development and production environments will likely allow different origins, we need a CORS configuration function taking allowed-origins as parameter.

Also, when a request to a protected resource is made with a missing or invalid authorization the HTTP status should be 401 (`Unauthorized`) but, by default, Spring returns 302 (redirect to login, which makes no sense on a resource server). To change that, we'll have to write an access denied handler.

Let's put our security configuration together and provide a security filter-chain bean complying with all our specifications:

```
@EnableWebFluxSecurity
@EnableReactiveMethodSecurity
@Configuration
public class WebSecurityConfig {

    @Bean
    SecurityWebFilterChain
            filterChain(ServerHttpSecurity http, ServerProperties serverProperties, @V
                    throws Exception {

        http.oauth2ResourceServer(resourceServer -> resourceServer.jwt());

        http.cors(cors -> cors.configurationSource(corsConfigurationSource(origins)));

        // State-less session (state in access token only)
        http.securityContextRepository(NoOpServerSecurityContextRepository.getInstance

        // Disable CSRF because of state-less session-management
        http.csrf().disable();

        // Return 401 (unauthorized) instead of 302 (redirect to login) when
        // authorization is missing or invalid
        http.exceptionHandling(exceptionHandling -> {
            exceptionHandling.accessDeniedHandler(accessDeniedHandler());
        });

        // If SSL enabled, disable http (https only)
        if (serverProperties.getSsl() != null && serverProperties.getSsl().isEnabled()
            http.redirectToHttps();
        }

        http.authorizeExchange(exchange -> exchange.pathMatchers(permitAll).permitAll(

        return http.build();
    }

    private UrlBasedCorsConfigurationSource corsConfigurationSource(String[] origins)
```

```
        final var configuration = new CorsConfiguration();
        configuration.setAllowedOrigins(Arrays.asList(origins));
        configuration.setAllowedMethods(List.of("*"));
        configuration.setAllowedHeaders(List.of("*"));
        configuration.setExposedHeaders(List.of("*"));

        final var source = new UrlBasedCorsConfigurationSource();
        source.registerCorsConfiguration("/**", configuration);
        return source;
    }

    private ServerAccessDeniedHandler accessDeniedHandler() {
        return (var exchange, var ex) -> exchange.getPrincipal().flatMap(principal ->
            var response = exchange.getResponse();
            response.setStatusCode(principal instanceof AnonymousAuthenticationToken ?
            response.getHeaders().setContentType(MediaType.TEXT_PLAIN);
            var dataBufferFactory = response.bufferFactory();
            var buffer = dataBufferFactory.wrap(ex.getMessage().getBytes(Charset.defau
            return response.writeWith(Mono.just(buffer)).doOnError(error -> DataBuffer
        });
    }
}
```

3.3. Authorities Mapping

As a reminder, the `scope` of a token defines what a resource-owner allowed an OAuth2 client to do on his behalf, where "roles" are a way to represent what a resource-owner himself is allowed to do on resource servers.

RBAC is a very common pattern for access-control, but neither OAuth2 nor OpenID define a standard representation for "roles". Each vendor implements it with its own private-claim(s).

Spring Security default authorities mapper, which maps from the `scope` claim, adding the `SCOPE_` prefix, won't satisfy to our needs (unless we twisted the usage of the `scope` claim on the authorization-server to contain user roles, of course).

We'll implement a mapping of Spring Security authorities from OpenID private claims with the following specifications:

- possibly use several claims as source, all of those claims being a string array or a single string of comma separated values (Keycloak for instance can provide with roles in `realm_access.roles` and `resource_access.{client-id}.roles`)
- configure case processing and prefix independently for each claim (for instance use `SCOPE_` prefix for scopes in `scp` claim and `ROLE_` prefix for roles in `realm_access.roles` one)
- provide with a different configuration for each provider (Keycloak, Auth0 and Cognito all use different private claims for user roles)

3.3.1. Dependencies

To ease roles claims parsing, we'll use json-path. Let's add it to our dependencies:

```xml
<dependency>
    <groupId>com.jayway.jsonpath</groupId>
    <artifactId>json-path</artifactId>
</dependency>
```

3.3.2. Application Properties

Then, we need some additional configuration properties to provide with the flexibility to change, for each issuer the claims containing the username and roles

```java
@Data
@Configuration
@ConfigurationProperties(prefix = "spring-addons")
public class SpringAddonsProperties {
    private IssuerProperties[] issuers = {};

    @Data
    static class IssuerProperties {
        private URL uri;
        private ClaimMappingProperties[] claims;
        private String usernameJsonPath = JwtClaimNames.SUB;

        @Data
        static class ClaimMappingProperties {
            private String jsonPath;
            private CaseProcessing caseProcessing = CaseProcessing.UNCHANGED;
            private String prefix = "";

            static enum CaseProcessing {
                UNCHANGED, TO_LOWER, TO_UPPER
            }
        }
    }

    public IssuerProperties get(URL issuerUri) throws MisconfigurationException {
        final var issuerProperties = Stream.of(issuers).filter(iss -> issuerUri.equals
        if (issuerProperties.size() == 0) {
            throw new MisconfigurationException("Missing authorities mapping propertie
        }
        if (issuerProperties.size() > 1) {
            throw new MisconfigurationException("Too many authorities mapping properti
        }
        return issuerProperties.get(0);
    }

    static class MisconfigurationException extends RuntimeException {
        private static final long serialVersionUID = 5887967904749547431L;

        public MisconfigurationException(String msg) {
            super(msg);
        }
    }
}
```

We'll also need the yaml properties matching this configuration:

```yaml
spring-addons:
  issuers:
  - uri: ${keycloak-issuer}
    username-json-path: $.preferred_username
    claims:
    - jsonPath: $.realm_access.roles
    - jsonPath: $.resource_access.*.roles
  - uri: ${cognito-issuer}
    claims:
    - jsonPath: $.cognito:groups
  - uri: ${auth0-issuer}
    claims:
    - jsonPath: $.roles
    - jsonPath: $.groups
    - jsonPath: $.permissions
```

3.3.3. Authorities and Authentication Converters

We can now write a converter bean building Spring authorities from JWT claims, as specified in the configuration above:

```java
@RequiredArgsConstructor
static class JwtGrantedAuthoritiesConverter implements Converter<Jwt, Collection<? ext
    private final SpringAddonsProperties.IssuerProperties properties;

    @Override
    @SuppressWarnings({ "rawtypes", "unchecked" })
    public Collection<? extends GrantedAuthority> convert(Jwt jwt) {
        return Stream.of(properties.claims).flatMap(claimProperties -> {
            Object claim;
            try {
                claim = JsonPath.read(jwt.getClaims(), claimProperties.jsonPath);
            } catch (PathNotFoundException e) {
                claim = null;
            }
            if (claim == null) {
                return Stream.empty();
            }
            if (claim instanceof String claimStr) {
                return Stream.of(claimStr.split(","));
            }
            if (claim instanceof String[] claimArr) {
                return Stream.of(claimArr);
            }
            if (Collection.class.isAssignableFrom(claim.getClass())) {
                final var iter = ((Collection) claim).iterator();
                if (!iter.hasNext()) {
                    return Stream.empty();
                }
                final var firstItem = iter.next();
                if (firstItem instanceof String) {
                    return (Stream<String>) ((Collection) claim).stream();
                }
                if (Collection.class.isAssignableFrom(firstItem.getClass())) {
                    return (Stream<String>) ((Collection) claim).stream().flatMap(colI
                }
            }
            return Stream.empty();
        }).map(SimpleGrantedAuthority::new).map(GrantedAuthority.class::cast).toList()
    }
}
```

This authorities converter will be instantiated and used by such an authentication converter:

```
@Component
@RequiredArgsConstructor
static class SpringAddonsJwtAuthenticationConverter implements Converter<Jwt, Mono<? e
    private final SpringAddonsProperties springAddonsProperties;

    @Override
    public Mono<? extends AbstractAuthenticationToken> convert(Jwt jwt) {
        final var issuerProperties = springAddonsProperties.get(jwt.getIssuer());
        final var authorities = new JwtGrantedAuthoritiesConverter(issuerProperties).c
        final String username = JsonPath.read(jwt.getClaims(), issuerProperties.getUse
        return Mono.just(new JwtAuthenticationToken(jwt, authorities, username));
    }
}
```

3.3.4. Security Configuration Update

The last missing configuration piece is an update of the security filter-chain: inject our authentication converter in th resource server configuration:

```
@Bean
SecurityWebFilterChain filterChain(
        ServerHttpSecurity http,
        ServerProperties serverProperties,
        @Value("origins") String[] origins,
        @Value("permit-all") String[] permitAll,
        SpringAddonsProperties springAddonsProperties,
        SpringAddonsJwtAuthenticationConverter authenticationConverter)
        throws Exception {

    http.oauth2ResourceServer(resourceServer -> resourceServer.jwt(jwtConf -> jwtConf.

...

return http.build();
}
```

Cool, we can now map authorities from any JWT access token, issued by any OIDC Provider, by just editing a configuration property!

3.3.5. Restricted End-Point

To demo Role Based Access Control, we'll edit our `@RestController` to add a new end-point only accessible to users granted with the `NICE` role:

```
@GetMapping("/restricted")
@PreAuthorize("hasAuthority('NICE')")
public Mono<MessageDto> getRestricted() {
    return Mono.just(new MessageDto("You are so nice!"));
}
```

4. Multi-Tenancy

So far, our resource server is able to accept identities issued by any OIDC Provider, but only one per resource server instance. This is frequently enough, but not always.

For instance, when working with Keycloak, we can consider all realms as distinct OPs. We'll have to provide some additional configuration for a single resource server instance to accept requests from users identified on different realms.

Let's first remove the `spring.security.oauth2.resourceserver.jwt.issuer-uri` which is not adapted to our use-case. We'll iterate over the `spring-addons.issuers` instead. We can remove the `auth0` and `cognito` profiles too.

Next, we'll define a `ReactiveAuthenticationManagerResolver`:

```
@Bean
ReactiveAuthenticationManagerResolver<ServerWebExchange>
        authenticationManagerResolver(SpringAddonsProperties addonsProperties, SpringA
    final Map<String, Mono<ReactiveAuthenticationManager>> jwtManagers = Stream.of(add
            .map(URL::toString).collect(Collectors.toMap(issuer -> issuer, issuer -> M
    return new JwtIssuerReactiveAuthenticationManagerResolver(issuerLocation -> jwtMan
}

JwtReactiveAuthenticationManager authenticationManager(String issuer, SpringAddonsJwtA
    ReactiveJwtDecoder decoder = ReactiveJwtDecoders.fromIssuerLocation(issuer);
    var provider = new JwtReactiveAuthenticationManager(decoder);
    provider.setJwtAuthenticationConverter(authenticationConverter);
    return provider;
}
```

Last, when configuring the resource server within the security filter-chain, we'll replace the authentication converter configuration with our new authentication manager resolver:

```
@Bean
SecurityWebFilterChain filterChain(
        ServerHttpSecurity http,
        ServerProperties serverProperties,
        @Value("origins") String[] origins,
        @Value("permit-all") String[] permitAll,
        SpringAddonsProperties springAddonsProperties,
        ReactiveAuthenticationManagerResolver<ServerWebExchange> authenticationManager
        throws Exception {
```

```
        // Configure the app as resource server with an authentication manager resolver ca
        http.oauth2ResourceServer(resourceServer -> resourceServer.authenticationManagerRe
        ...
        return http.build();
    }
```

5. Testing

Explore the source code to have a look at how to mock identities in unit and integration tests and assert access-control is behaving as expected. All samples and tutorials include detailed access-control tests.

6. Conclusion

In this tutorial, we configured a reactive (WebFlux) Spring Boot 3 application as an OAuth2 resource server with authorities mapping to enable RBAC using roles defined in as many OIDC Providers (or Keycloak realms) as we need, no matter if they send user roles in the same claim(s).

But wait, what we did here is pretty verbose and we'll need it in almost any OAuth2 resource server we write. Do we really have to write all that again and again? Not really: this repo provides with a `spring-addons-webflux-jwt-resource-server` Spring Boot starter just for that, with a sample usage here. And if for whatever reason you don't want to use that one, you can still write your own starter to wrap the configuration we wrote here.