Security First docs (0.0.1)

Maksim Kostromin

Version 0.0.1, 2018-05-28 05:05:18 EEST

Table of Contents

1. Introduction . . . 2
2. JDBC . . . 3
    2.1. authorization server . . . 3
    2.2. implementation . . . 3
        2.2.1. provide database . . . 3
        2.2.2. jdbc user details service . . . 6
        2.2.3. authentication manager config . . . 6
        2.2.4. jdbc oauth2 auth server config . . . 7
        2.2.5. password encoder config . . . 8
    2.3. resource server . . . 9
    2.4. implementation . . . 9
        2.4.1. remote token services . . . 9

Travis CI status:

Chapter 1. Introduction

Yeah, same as mobile-first. Main goal of that project is learn spring-security topic from basis to advanced and share with people. Because no one enterprise application can't go live without security, I believe it should be done first. It's also can help you avoid situation when application architecture needs to be refactored to have possibility to apply security to it… I saw that many times… For some reason, big part of software developers community do not care about security from beginning or even to the end. I think main reason is because security hard topic. And it's really sad - many developers want doing it right, but for some reasons, people teaching them develop software missing security.

Whole purpose of that project is learn and share such important topic in software development world. Any enterprise project cannot go live without security. So my personal lifestyle is always think about security first, and only after build any other application functionality. Any good framework must provide developers some good security solution …and spring does. **Spring Soot** is amazing framework. It's really one of the best java framework I know. As part of it, **Spring Security** provide us a lot of great features - it's easiest security solutions I have ever seen and used in java.

Chapter 2. JDBC

2.1. authorization server

Let's start with building `jdbc-oauth2-auth-server`

build, run and test

```
./gradlew clean :apps:jd-o-a-s:bootRun
http -a clientPassword:secret --form post :8001/oauth/token grant_type=password
username=usr password=pwd
```

output

```
{
    "access_token": "53af6890-21bc-444e-800b-e93bef1e1e7d",
    "expires_in": 43199,
    "refresh_token": "e1bcb2b4-1a7e-4727-b807-8c9790de7cca",
    "scope": "read",
    "token_type": "bearer"
}
```

2.2. implementation

1. provide database
    a. `DataSourceConfig`
    b. `src/main/resources/application-h2.yaml`
    c. `src/main/resources/schema-h2.sql`
2. provide UserDetailsService (mocked in our case for simplicity: `JdbcUserDetailsService`)
3. after you have UserDetailsService, you can create `AuthenticationManagerConfig`
4. finally you can configure `JdbcOauth2AuthServerConfig`
5. and of course, do not forget about password encoder: `PasswordEncoderConfig`

2.2.1. provide database

application-h2.yaml

```
spring:
  datasource:
    url: jdbc:h2:file:./security-
first;AUTO_SERVER=TRUE;MULTI_THREADED=TRUE;MODE=MYSQL;DB_CLOSE_ON_EXIT=FALSE;AUTO_RECO
NNECT=TRUE
    username: security-first
    password: security-first
    driver-class-name: org.h2.Driver
    hikari.connection-test-query: 'SELECT 1;'
  h2.console.enabled: true
logging.level.org.springframework.jdbc: 'DEBUG'
```

application-h2.yaml

```
  @Bean
  public DataSourceInitializer dataSourceInitializer(final DataSource dataSource) {
    final DataSourceInitializer initializer = new DataSourceInitializer();
    initializer.setDataSource(dataSource);
    initializer.setDatabasePopulator(databasePopulator());
    return initializer;
  }

  private DatabasePopulator databasePopulator() {
    final ClassPathResource schema = new ClassPathResource("/schema-h2.sql",
DataSourceConfig.class.getClassLoader());
    return new ResourceDatabasePopulator(false, true, UTF_8.displayName(), schema);
  }
```

schema-h2.sql

```
drop table if exists oauth_client_details;
create table oauth_client_details (
  client_id VARCHAR(255) PRIMARY KEY,
  resource_ids VARCHAR(255),
  client_secret VARCHAR(255),
  scope VARCHAR(255),
  authorized_grant_types VARCHAR(255),
  web_server_redirect_uri VARCHAR(255),
  authorities VARCHAR(255),
  access_token_validity INTEGER,
  refresh_token_validity INTEGER,
  additional_information VARCHAR(4096),
  autoapprove VARCHAR(255)
);

drop table if exists oauth_client_token;
create table oauth_client_token (
  token_id VARCHAR(255),
```

```
  token LONGVARBINARY,
  authentication_id VARCHAR(255) PRIMARY KEY,
  user_name VARCHAR(255),
  client_id VARCHAR(255)
);

drop table if exists oauth_access_token;
create table oauth_access_token (
  token_id VARCHAR(255),
  token LONGVARBINARY,
  authentication_id VARCHAR(255) PRIMARY KEY,
  user_name VARCHAR(255),
  client_id VARCHAR(255),
  authentication LONGVARBINARY,
  refresh_token VARCHAR(255)
);

drop table if exists oauth_refresh_token;
create table oauth_refresh_token (
  token_id VARCHAR(255),
  token LONGVARBINARY,
  authentication LONGVARBINARY
);

drop table if exists oauth_code;
create table oauth_code (
  code VARCHAR(255), authentication LONGVARBINARY
);

drop table if exists oauth_approvals;
create table oauth_approvals (
    userId VARCHAR(255),
    clientId VARCHAR(255),
    scope VARCHAR(255),
    status VARCHAR(10),
    expiresAt TIMESTAMP,
    lastModifiedAt TIMESTAMP
);

drop table if exists ClientDetails;
create table ClientDetails (
  appId VARCHAR(255) PRIMARY KEY,
  resourceIds VARCHAR(255),
  appSecret VARCHAR(255),
  scope VARCHAR(255),
  grantTypes VARCHAR(255),
  redirectUrl VARCHAR(255),
  authorities VARCHAR(255),
  access_token_validity INTEGER,
  refresh_token_validity INTEGER,
  additionalInformation VARCHAR(4096),
```

```
  autoApproveScopes VARCHAR(255)
);
```

2.2.2. jdbc user details service

JdbcUserDetailsService

```
@Service
@RequiredArgsConstructor
class JdbcUserDetailsService implements UserDetailsService {

  final PasswordEncoder encoder;
  final JdbcTemplate jdbcTemplate;

  @Override
  public UserDetails loadUserByUsername(String username) throws
UsernameNotFoundException {
    return User
        .withUsername("usr")
        //.password("pwd")
        .password(null == encoder ? "{noop}pwd" : encoder.encode("pwd"))
        .disabled(false)
        .accountExpired(false)
        .accountLocked(false)
        .credentialsExpired(false)
        .authorities("USER", "ADMIN", "ROLE_USER", "ROLE_ADMIN")
        .build();
  }
}
```

2.2.3. authentication manager config

AuthenticationManagerConfig

```
@Configuration
@RequiredArgsConstructor
class AuthenticationManagerConfig extends WebSecurityConfigurerAdapter {

  final JdbcUserDetailsService userDetailsService;

  @Override
  protected void configure(AuthenticationManagerBuilder auth) throws Exception {
    auth.userDetailsService(userDetailsService);
  }

  @Bean
  @Override
  public AuthenticationManager authenticationManagerBean() throws Exception {
    return super.authenticationManagerBean();
  }
}
```

2.2.4. jdbc oauth2 auth server config

JdbcOauth2AuthServerConfig

```
@Configuration
@RequiredArgsConstructor
@EnableAuthorizationServer
class JdbcOauth2AuthServerConfig extends AuthorizationServerConfigurerAdapter {

  final DataSource dataSource;
  final PasswordEncoder encoder;
  final AuthenticationManager authenticationManager;

  @Bean
  public TokenStore tokenStore() {
    return new JdbcTokenStore(dataSource);
  }

  @Override
  public void configure(final ClientDetailsServiceConfigurer clients) throws Exception
{
    clients
        .jdbc(dataSource)
        .withClient("clientId")
          .authorizedGrantTypes("implicit")
          //.secret("secret")
          .secret(null == encoder ? "{noop}secret" : encoder.encode("secret"))
          .scopes("read")
          .autoApprove(true)
          .and()
```

```
        .withClient("clientPassword")
          //.secret("secret")
          .secret(null == encoder ? "{noop}secret" : encoder.encode("secret"))
          .scopes("read")
          .authorizedGrantTypes(
              "authorization_code",
              "refresh_token",
              "password"
          )
    ;
  }

  @Override
  public void configure(AuthorizationServerEndpointsConfigurer endpoints) throws
Exception {
    endpoints
        .tokenStore(tokenStore())
        .authenticationManager(authenticationManager)
    ;
  }

  @Override
  public void configure(AuthorizationServerSecurityConfigurer oauthServer) {
    oauthServer
        .tokenKeyAccess("permitAll()")
        .checkTokenAccess("isAuthenticated()");
  }
}
```

2.2.5. password encoder config

PasswordEncoderConfig

```
  @Bean
  public PasswordEncoder encoder() {
    return PasswordEncoderFactories.createDelegatingPasswordEncoder();
  }
```

---

links:

- authorization server reference
- get some thought from here...
- auth/resource servers some examples
- Migration guide to spring-boot 2.x

2.3. resource server

Now we ready to go with building `jdbc-oauth2-resource-server`

build, run and test

```
./gradlew clean :apps:jd-o-r-s:bootRun
```

2.4. implementation

1. implement resource server `/check_token` remote token service endpoint: `RemoteTokenServicesConfig`

2.4.1. remote token services

RemoteTokenServicesConfig

```
@Bean
@Primary
public RemoteTokenServices tokenService() {
    final RemoteTokenServices tokenService = new RemoteTokenServices();
    tokenService.setCheckTokenEndpointUrl(
    "http://localhost:8080/spring-security-oauth-server/oauth/check_token");
    tokenService.setClientId("fooClientIdPassword");
    tokenService.setClientSecret("secret");
    return tokenService;
}
```