Spring Security

v1.0.0

Spring Security

Spring Security

Spring Security jest to projekt, który umożliwia definiowanie mechanizmów bezpieczeństwa w projektach opartych na **Springu**.

Dzięki niemu możemy w prosty i szybki sposób wzbogacić naszą aplikacje o mechanizmy zabezpieczenia dostępu do naszej aplikacji.

Strona projektu:

http://projects.spring.io/spring-security/

Podstawowe pojęcia

Uwierzytelnianie - to sprawdzanie tożsamości danego użytkownika. Najczęściej utożsamiane z procesem logowania za pomocą formularza.

Autoryzacja - jest to proces nadawania uprawnień. W ten sposób ograniczamy dostęp do określonych elementów naszej aplikacji.

Spring Security

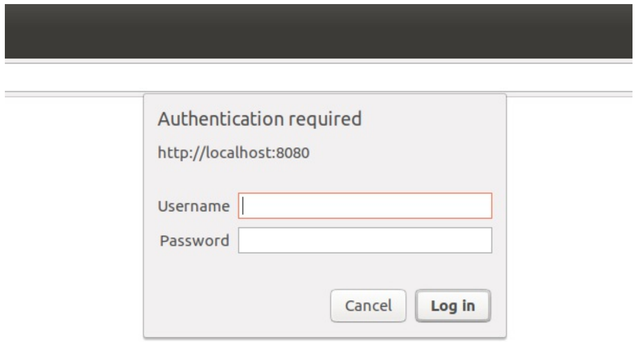
Dodanie wsparcia dla **Spring Security** w naszym projekcie sprowadza się do uzupełnienia pliku **pom.xml** o następujący starter:

```
<dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-security</artifactId>
</dependency>
```

Możemy go również wskazać podczas tworzenia projektu **Spring Boot** przy użyciu **Initializer** lub możliwości naszego **IDE**.

Spring Security

Po restarcie aplikacji oraz odświeżeniu w przeglądarce naszej aplikacji otrzymamy prośbę o podanie nazwy użytkownika oraz hasła.



Spring Security

Nazwa użytkownika do logowania to **user** natomiast hasło wymagane do autoryzacji otrzymamy wypisane w konsoli podczas uruchamiania aplikacji:

```
2018-01-14 22:02:57.766  INFO 14541 --- [  restartedMain]
2018-01-14 22:02:58.315  INFO 14541 --- [  restartedMain]

Using default security password: 9d78a01c-309f-461d-8551-e3ccd159f569

2018-01-14 22:02:58.387  INFO 14541 --- [  restartedMain]
2018-01-14 22:02:58.387  INFO 14541 --- [  restartedMain]
```

Spring Security

Nazwa użytkownika do logowania to **user** natomiast hasło wymagane do autoryzacji otrzymamy wypisane w konsoli podczas uruchamiania aplikacji:

```
2018-01-14 22:02:57.766  INFO 14541 --- [  restartedMain]
2018-01-14 22:02:58.315  INFO 14541 --- [  restartedMain]

Using default security password: 9d78a01c-309f-461d-8551-e3ccd159f569

2018-01-14 22:02:58.387  INFO 14541 --- [  restartedMain]
2018-01-14 22:02:58.387  INFO 14541 --- [  restartedMain]
```

Wygenerowane hasło.

Spring Security

Dane do autoryzacji możemy określić w pliku **application.properties** za pomocą wpisów:

```
security.user.name=admin
security.user.password=admin
```

Tego typu zabezpieczenie może być użyteczne tylko w przypadku bardzo prostych aplikacji, która nie definiuje kontroli dostępu - czyli jakie zasoby/możliwości są dostępne dla zalogowanego użytkownika.

Za pomocą tej konfiguracji nie mamy możliwości definicji większej ilości użytkowników, ani operacji zmiany hasła.

Brak również definicji roli dla różnych użytkowników.

Spring Security

Do naszej aplikacji dodamy klasę konfiguracji, która umożliwi nam większą kontrolę oraz możliwości konfiguracyjne dla modułu **Spring Security**:

```
package pl.coderslab;

import org.springframework.context.annotation.Configuration;
import org.springframework.security.config.annotation
.web.configuration.EnableWebSecurity;
import org.springframework.security.config.annotation
.web.configuration.WebSecurityConfigurerAdapter;

@Configuration
@EnableWebSecurity
public class SecurityConfig extends WebSecurityConfigurerAdapter {
}
```

Spring Security

Do naszej aplikacji dodamy klasę konfiguracji, która umożliwi nam większą kontrolę oraz możliwości konfiguracyjne dla modułu **Spring Security**:

```
package pl.coderslab;

import org.springframework.context.annotation.Configuration;
import org.springframework.security.config.annotation
.web.configuration.EnableWebSecurity;
import org.springframework.security.config.annotation
.web.configuration.WebSecurityConfigurerAdapter;

@Configuration
@EnableWebSecurity
public class SecurityConfig extends WebSecurityConfigurerAdapter {
}
```

Włączamy ustawienia bezpieczeństwa w aplikacji **Spring** .

Spring Security

Za pomocą metody **configure(AuthenticationManagerBuilder auth)** klasy **SecurityConfig** możemy zdefiniować użytkowników ich hasła oraz role.

W naszym przypadku definiujemy użytkowników zapisując ich w pamięci. W dalszej części będziemy rozbudowywać nasz system przechowywania danych o użytkownikach.

```
@Override
public void configure(AuthenticationManagerBuilder auth) throws Exception {
    auth.inMemoryAuthentication()
        .withUser("user1").password("user123").roles("USER")
        .and()
        .withUser("admin1").password("admin123").roles("ADMIN");
}
```

Spring Security

Za pomocą metody **configure(AuthenticationManagerBuilder auth)** klasy **SecurityConfig** możemy zdefiniować użytkowników ich hasła oraz role.

W naszym przypadku definiujemy użytkowników zapisując ich w pamięci. W dalszej części będziemy rozbudowywać nasz system przechowywania danych o użytkownikach.

```
@Override
public void configure(AuthenticationManagerBuilder auth) throws Exception {
    auth.inMemoryAuthentication()
        .withUser("user1").password("user123").roles("USER")
        .and()
        .withUser("admin1").password("admin123").roles("ADMIN");
}
```

Definiujemy użytkownika o nazwie **user1**, haśle **user123** oraz roli **USER**.

Spring Security

Za pomocą metody **configure(AuthenticationManagerBuilder auth)** klasy **SecurityConfig** możemy zdefiniować użytkowników ich hasła oraz role.

W naszym przypadku definiujemy użytkowników zapisując ich w pamięci. W dalszej części będziemy rozbudowywać nasz system przechowywania danych o użytkownikach.

```
@Override
public void configure(AuthenticationManagerBuilder auth) throws Exception {
    auth.inMemoryAuthentication()
        .withUser("user1").password("user123").roles("USER")
        .and()
        .withUser("admin1").password("admin123").roles("ADMIN");
}
```

Definiujemy użytkownika o nazwie **user1**, haśle **user123** oraz roli **USER**.

Definiujemy użytkownika o nazwie **admin1**, haśle **admin123** oraz roli **ADMIN**.

Spring Security

Kolejnym krokiem jest zdefiniowanie jakie adresy mają być zabezpieczone, a jakie dostępne dla wszystkich.

Dla przykładu zdefiniujemy kontroler zawierający dwie akcje:

```
@Controller
public class HomeController {

    @GetMapping("/")
    @ResponseBody
    public String home() { return "home"; }

    @GetMapping("/admin")
    @ResponseBody
    public String admin() { return "admin"; }
}
```

Spring Security

Kolejnym krokiem jest zdefiniowanie jakie adresy mają być zabezpieczone, a jakie dostępne dla wszystkich.

Dla przykładu zdefiniujemy kontroler zawierający dwie akcje:

```
@Controller
public class HomeController {

    @GetMapping("/")
    @ResponseBody
    public String home() { return "home"; }

    @GetMapping("/admin")
    @ResponseBody
    public String admin() { return "admin"; }
}
```

Akcja będzie dostępna dla wszystkich.

Spring Security

Kolejnym krokiem jest zdefiniowanie jakie adresy mają być zabezpieczone, a jakie dostępne dla wszystkich.

Dla przykładu zdefiniujemy kontroler zawierający dwie akcje:

```
@Controller
public class HomeController {

    @GetMapping("/")
    @ResponseBody
    public String home() { return "home"; }

    @GetMapping("/admin")
    @ResponseBody
    public String admin() { return "admin"; }
}
```

Dostęp do tej akcji będzie ograniczony.

Spring Security

W celu określenia zasad dostępu uzupełniamy klasę **SecurityConfig** o poniższą metodę

```
@Override
protected void configure(HttpSecurity http) throws Exception {
    http.authorizeRequests()
        .antMatchers("/").permitAll()
        .antMatchers("/admin").authenticated()
        .and().formLogin();
}
```

Spring Security

W celu określenia zasad dostępu uzupełniamy klasę **SecurityConfig** o poniższą metodę

```
@Override
protected void configure(HttpSecurity http) throws Exception {
    http.authorizeRequests()
        .antMatchers("/").permitAll()
        .antMatchers("/admin").authenticated()
        .and().formLogin();
}
```

Za pomocą metody **antMatchers** wskazujemy adres, a następnie przy pomocy metody kolejnych metod określamy zasady dostępu, **permitAll()** - wskazuje że dostęp nie wymaga uwierzytelnienia.

Spring Security

W celu określenia zasad dostępu uzupełniamy klasę **SecurityConfig** o poniższą metodę

```
@Override
protected void configure(HttpSecurity http) throws Exception {
    http.authorizeRequests()
        .antMatchers("/").permitAll()
        .antMatchers("/admin").authenticated()
        .and().formLogin();
}
```

Dla kolejnego adresu ustawiamy dostęp, **authenticated()** - wskazuje że dostęp wymaga uwierzytelnienia.

Spring Security

W celu określenia zasad dostępu uzupełniamy klasę **SecurityConfig** o poniższą metodę

```
@Override
protected void configure(HttpSecurity http) throws Exception {
    http.authorizeRequests()
        .antMatchers("/").permitAll()
        .antMatchers("/admin").authenticated()
        .and().formLogin();
}
```

Włączamy domyślną stronę logowania.

Spring Security

Metoda **antMatchers** może przyjmować poniższe konstrukcje:

Wskazujemy wiele adresów:

```
.antMatchers("/admin","/admin/users")
```

Możemy wskazać że wszystkie adresy z prefiksem **admin** mają wymagać uwierzytelnienia:

```
.antMatchers("/admin/**")
```

Możemy wskazać że dany adres ale tylko dla określonej metody HTTP mają wymagać uwierzytelnienia:

```
.antMatchers(HttpMethod.POST, "/admin/user/add")
```

hasRole

Możemy jednoznacznie wskazać dla jakiej roli dostęp ma być dozwolony.

Służy do tego metoda: **hasRole**, np:

```
@Override
protected void configure(HttpSecurity http) throws Exception {
    http.authorizeRequests()
        .antMatchers("/admin/**").hasRole("ADMIN")
        .and().formLogin();
}
```

hasRole

Możemy jednoznacznie wskazać dla jakiej roli dostęp ma być dozwolony.

Służy do tego metoda: **hasRole**, np:

```
@Override
protected void configure(HttpSecurity http) throws Exception {
    http.authorizeRequests()
        .antMatchers("/admin/**").hasRole("ADMIN")
        .and().formLogin();
}
```

Określamy że wszystkie adresy z prefiksem **/admin/** mają być dostępne tylko dla roli **ADMIN**.

hasAnyRole

Możemy wskazać zestaw ról, dla których dostęp ma być dozwolony.

Służy do tego metoda: **hasAnyRole**, np:

```
@Override
protected void configure(HttpSecurity http) throws Exception {
    http.authorizeRequests()
        .antMatchers("/admin/**").hasAnyRole("USER","ADMIN")
        .and().formLogin();
}
```

hasAnyRole

Możemy wskazać zestaw ról, dla których dostęp ma być dozwolony.

Służy do tego metoda: **hasAnyRole**, np:

```
@Override
protected void configure(HttpSecurity http) throws Exception {
    http.authorizeRequests()
        .antMatchers("/admin/**").hasAnyRole("USER","ADMIN")
        .and().formLogin();
}
```

Określamy że wszystkie adresy z prefiksem **/admin/** mają być dostępne dla każdej z roli **ADMIN** oraz **USER**.

Nazwy roli

Warto zwrócić uwagę że pełna nazwa roli **USER** to **ROLE_USER** podobnie jak roli **ADMIN** to **ROLE_ADMIN**.

Metody **hasRole**, **hasAnyRole** automatycznie dodają przedrostek **ROLE_**.

Jeżeli omyłkowo będziemy określać np.:

```
.hasRole("ROLE_ADMIN")
```

otrzymamy wyjątek:

```
Caused by: java.lang.IllegalArgumentException:
    role should not start with 'ROLE_' since it is automatically inserted.
    Got 'ROLE_ADMIN'
```

Jest to szczególnie ważne ponieważ istnieją również metody **hasAuthority** oraz **hasAnyAuthority**, którym podajemy pełne nazwy roli.

Zabezpieczenie kontrolera

Oprócz definicji zabezpieczeń za pomocą omówionej metody **configure** możemy również nadawać uprawnienia za pomocą adnotacji na poziomie całego kontrolera, za pomocą adnotacji **@Secured** :

```
@Controller
@Secured("ROLE_ADMIN")
public class SecuredController {

    @GetMapping("/secured1")
    @ResponseBody
    public String secured1(){
        return "secured1";
    }
}
```

Dla zabezpieczenia na poziomie metod wymagane jest dodanie w konfiguracji adnotacji:
@EnableGlobalMethodSecurity(securedEnabled = true)

Zabezpieczenie akcji

Adnotacji **@Secured** możemy używać dla akcji kontrolera, np:

```
@Secured("ROLE_USER")
@GetMapping("/user")
    public String user() {
        return "admin/panel";
}
```

Zabezpieczenie dowolnych metod

Adnotacji **@Secured** możemy również używać do zabezpieczenia dowolnych metod naszej aplikacji, np:

```
@Service
public class MyService {

  @Secured("ROLE_USER")
  public String secure() {
    return "Hello Security";
  }

}
```

Pozostałe adresy

Po określeniu adresów, które wymagają uwierzytelnienia możemy za pomocą wpisu:

```
.anyRequest().permitAll();
```

określić że dla wszystkich nie ujętych za pomocą definicji **antMatchers** adresów dostęp nie wymaga uwierzytelniania, np:

```
@Override
protected void configure(HttpSecurity http) throws Exception {
    http.authorizeRequests()
        .antMatchers("/admin/**").hasAnyRole("USER","ADMIN")
        .anyRequest().permitAll()
        .and().formLogin();
}
```

Strona logowania

Jeżeli włączymy za pomocą metody **formLogin()** logowanie za pomocą formularza, **Spring** wygeneruje dla nas domyślny formularz.

Aby nadpisać domyślny widok formularza w konfiguracji określamy jej adres w następujący sposób:

```
@Override
protected void configure(HttpSecurity http) throws Exception {
    http.authorizeRequests()
    .antMatchers("/admin/**").hasAnyRole("USER","ADMIN")
        .antMatchers("/admin/**").hasAnyAuthority("ROLE_ADMIN")
        .anyRequest().permitAll()
        .and().formLogin()
    .loginPage("/login");
}
```

Strona logowania

Jeżeli włączymy za pomocą metody **formLogin()** logowanie za pomocą formularza, **Spring** wygeneruje dla nas domyślny formularz.

Aby nadpisać domyślny widok formularza w konfiguracji określamy jej adres w następujący sposób:

```
@Override
protected void configure(HttpSecurity http) throws Exception {
    http.authorizeRequests()
    .antMatchers("/admin/**").hasAnyRole("USER","ADMIN")
        .antMatchers("/admin/**").hasAnyAuthority("ROLE_ADMIN")
        .anyRequest().permitAll()
        .and().formLogin()
    .loginPage("/login");
}
```

Metoda określająca adres url dla strony logowania.

Strona logowania

Dodajemy akcję wyświetlającą nowy widok formularza logowania. Posłużymy się w tym celu nowym kontrolerem o nazwie **LoginController**.

Kontroler musi zawierać poniższą akcję:

```
@RequestMapping(value = {"/login"}, method = RequestMethod.GET)
public String login() {
    return "admin/login";
}
```

Strona logowania

Dodajemy akcję wyświetlającą nowy widok formularza logowania. Posłużymy się w tym celu nowym kontrolerem o nazwie **LoginController**.

Kontroler musi zawierać poniższą akcję:

```
@RequestMapping(value = {"/login"}, method = RequestMethod.GET)
public String login() {
    return "admin/login";
}
```

Określamy lokalizację pliku widoku, w naszym przypadku widok znajduje się w katalogu **admin** .

Strona logowania

W przypadkach gdy zadaniem akcji naszej aplikacji jest jedynie wyświetlenie widoku, możemy wykorzystać, klasę rozszerzającą **WebMvcConfigurerAdapter** a następnie w metodzie **addViewControllers** zdefiniować adresy oraz widoki, które dla tych adresów mają zostać wyświetlone.

```
package pl.coderslab;
import org.springframework.context.annotation.Configuration;
import org.springframework.web.servlet.config.annotation.ViewControllerRegistry;
import org.springframework.web.servlet.config.annotation.WebMvcConfigurerAdapter;

@Configuration
public class WebAppConfig extends WebMvcConfigurerAdapter {
    @Override
    public void addViewControllers(ViewControllerRegistry registry) {
        registry.addViewController("/login").setViewName("admin/login");
    }}
```

Strona logowania

W przypadkach gdy zadaniem akcji naszej aplikacji jest jedynie wyświetlenie widoku, możemy wykorzystać, klasę rozszerzającą **WebMvcConfigurerAdapter** a następnie w metodzie **addViewControllers** zdefiniować adresy oraz widoki, które dla tych adresów mają zostać wyświetlone.

```
package pl.coderslab;
import org.springframework.context.annotation.Configuration;
import org.springframework.web.servlet.config.annotation.ViewControllerRegistry;
import org.springframework.web.servlet.config.annotation.WebMvcConfigurerAdapter;

@Configuration
public class WebAppConfig extends WebMvcConfigurerAdapter {
    @Override
    public void addViewControllers(ViewControllerRegistry registry) {
        registry.addViewController("/login").setViewName("admin/login");
    }}
```

Dla adresu **/login** zostanie wyświetlony widok o nazwie **login.html** z folderu **admin**.

Wylogowanie

Po wylogowaniu zostaniemy przekierowani na domyślny adres: **/login?logout**, aby zmienić to ustawienie, dodajemy w metodzie **configure** następującą konfigurację:

```
@Override
protected void configure(HttpSecurity http) throws Exception {
    http.authorizeRequests()
    .antMatchers("/admin/**").hasAnyRole("USER","ADMIN")
        .anyRequest().permitAll()
        .and().formLogin().loginPage("/login")
        .and().logout().logoutSuccessUrl("/")
        .permitAll();
}
```

Wylogowanie

Po wylogowaniu zostaniemy przekierowani na domyślny adres: **/login?logout**, aby zmienić to ustawienie, dodajemy w metodzie **configure** następującą konfigurację:

```
@Override
protected void configure(HttpSecurity http) throws Exception {
    http.authorizeRequests()
    .antMatchers("/admin/**").hasAnyRole("USER","ADMIN")
        .anyRequest().permitAll()
        .and().formLogin().loginPage("/login")
        .and().logout().logoutSuccessUrl("/")
        .permitAll();
}
```

Dla akcji wylogowania ustawiamy za pomocą metody **logoutSuccessUrl** adres na jaki zostaniemy przekierowani.

Strona - brak autoryzacji

Uzupełnimy nasz projekt o stronę na której wyświetlimy w przejrzysty sposób informację o braku dostępu do określonego zasobu.

Dodajemy w metodzie **addViewControllers** klasy, **WebAppConfig** dodatkowy wpis dla url: **/403**.

```
@Configuration
public class WebAppConfig extends WebMvcConfigurerAdapter {

    @Override
    public void addViewControllers(ViewControllerRegistry registry) {
        registry.addViewController("/login").setViewName("admin/login");
        registry.addViewController("/403").setViewName("403");
    }
}
```

Strona - brak autoryzacji

Uzupełnimy nasz projekt o stronę na której wyświetlimy w przejrzysty sposób informację o braku dostępu do określonego zasobu.

Dodajemy w metodzie **addViewControllers** klasy, **WebAppConfig** dodatkowy wpis dla url: **/403**.

```
@Configuration
public class WebAppConfig extends WebMvcConfigurerAdapter {

    @Override
    public void addViewControllers(ViewControllerRegistry registry) {
        registry.addViewController("/login").setViewName("admin/login");
        registry.addViewController("/403").setViewName("403");
    }
}
```

Wskazujemy widok o nazwie **403** .

Strona - brak autoryzacji

Uzupełniamy konfigurację w pliku **WebSecurityConfigurerAdapter** o następującą definicję określającą adres strony błędu:

```
 .and().exceptionHandling().accessDeniedPage("/403")
```

Cała metoda wygląda następująco:

```
protected void configure(HttpSecurity http) throws Exception {
    http.authorizeRequests()
    .antMatchers("/admin/**").hasAnyRole("USER","ADMIN")
        .anyRequest().permitAll()
        .and().formLogin().loginPage("/login")
        .and().logout().logoutSuccessUrl("/")
        .permitAll()
        .and().exceptionHandling().accessDeniedPage("/403");
}
```

JDBC

Posługiwanie się kontami i rolami zapisanymi w kodzie nie jest rozwiązaniem wygodnym, dlatego do przechowywania informacji o użytkownikach wykorzystamy bazę danych.

W tym celu zmodyfikujemy klasę **SecurityConfig**:

```
@Autowired
DataSource dataSource;

@Override
public void configure(AuthenticationManagerBuilder auth) throws Exception {
    auth.jdbcAuthentication().dataSource(dataSource);
}
```

JDBC

Posługiwanie się kontami i rolami zapisanymi w kodzie nie jest rozwiązaniem wygodnym, dlatego do przechowywania informacji o użytkownikach wykorzystamy bazę danych.

W tym celu zmodyfikujemy klasę **SecurityConfig**:

```
@Autowired
DataSource dataSource;

@Override
public void configure(AuthenticationManagerBuilder auth) throws Exception {
    auth.jdbcAuthentication().dataSource(dataSource);
}
```

Wstrzykujemy obiekt typu **DataSource**, jest on automatycznie tworzony przez **Springa**.

JDBC

Posługiwanie się kontami i rolami zapisanymi w kodzie nie jest rozwiązaniem wygodnym, dlatego do przechowywania informacji o użytkownikach wykorzystamy bazę danych.

W tym celu zmodyfikujemy klasę **SecurityConfig**:

```
@Autowired
DataSource dataSource;

@Override
public void configure(AuthenticationManagerBuilder auth) throws Exception {
    auth.jdbcAuthentication().dataSource(dataSource);
}
```

Określamy, że korzystamy z autentyfikacji opartej na **JDBC**, a następnie ustawiamy źródło danych.

JDBC

Kolejnym krokiem jest dodanie tabel bazy danych, poniżej kod **sql**, który tworzy proste tabele bazy danych do przechowywania informacji o użytkownikach oraz ich rolach.

Tabela do przechowywania danych o użytkownikach:

```
create table users (
  username varchar(256),
  password varchar(256),
  enabled boolean
);
```

Tabela do przechowywania danych o rolach:

```
create table authorities (
  username varchar(256),
  authority varchar(256)
);
```

JDBC

Uzupełniamy informacje w bazie danych, wczytując poniższe zapytania:

Dodajemy użytkowników:

```
insert into users (username, password, enabled) values ('user', 'pass', true);
```

Dodajemy role:

```
insert into authorities (username, authority) values ('user', 'ROLE_ADMIN');
```

Po uzupełnieniu danych w bazie już możemy się poprawnie zalogować na utworzone konto przy pomocy loginu **user** oraz hasła **pass**.

Przechowywanie haseł

Jak zapewne zwróciliśmy uwagę, hasło przechowujemy w bazie w postaci tekstowej, co, jak doskonale wiemy, nie jest dobrym pomysłem ze względów bezpieczeństwa.

Definiujemy bean typu **BCryptPasswordEncoder** jest to gotowa implementacja poznanego przez nas algorytmu **BCrypt** dostarczona razem z **Spring Security**.

Dodatkowy elementy w klasie **SecurityConfig**:

```
@Bean
public BCryptPasswordEncoder passwordEncoder() {
    return new BCryptPasswordEncoder();
}
```

Przechowywanie haseł

Przy pomocy metody **passwordEncoder** ustawiamy sposób kodowania w następujący sposób:

```
@Override
public void configure(AuthenticationManagerBuilder auth) throws Exception {
    auth.jdbcAuthentication().dataSource(dataSource)
    .passwordEncoder(passwordEncoder());
}
```

Tworzenie użytkownika

Operacja z poprzedniego slajdu spowoduje, że nie będziemy w stanie zalogować się na wcześniej utworzone konta.

Prostym sposobem na utworzenie konta do celów developerskich jest wykorzystanie metody **withUser** w następujący sposób

```
@Override
public void configure(AuthenticationManagerBuilder auth) throws Exception {
    auth.jdbcAuthentication().dataSource(dataSource)
    .passwordEncoder(passwordEncoder())
     .withUser("admin").password(passwordEncoder().encode("admin"))
                     .roles("ADMIN", "USER");
}
```

Tworzenie użytkownika

Operacja z poprzedniego slajdu spowoduje, że nie będziemy w stanie zalogować się na wcześniej utworzone konta.

Prostym sposobem na utworzenie konta do celów developerskich jest wykorzystanie metody **withUser** w następujący sposób

```
@Override
public void configure(AuthenticationManagerBuilder auth) throws Exception {
    auth.jdbcAuthentication().dataSource(dataSource)
    .passwordEncoder(passwordEncoder())
     .withUser("admin").password(passwordEncoder().encode("admin"))
                    .roles("ADMIN", "USER");
}
```

Pamiętajmy, że każdorazowa uruchomienie aplikacji spowoduje ponowne dodanie użytkownika do bazy danych .

Encje

Mamy w bazie danych tabele odpowiedzialne za przechowywanie danych użytkowników, tabele w bazie nie mają jednak swojej reprezentacji w postaci klas w naszej aplikacji.

Utworzymy odpowiednie encje, a następnie dokonamy modyfikacji ustawień.

Encja Role

Utworzymy encję reprezentującą role w naszej aplikacji:

```
@Entity
@Table(name = "role")
public class Role {
    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    @Column(name = "role_id")
    private int id;
    @Column(name = "role")
    private String name;
    //gettery, settery
}
```

Encja User

Następna encja będzie reprezentować użytkownika oraz będzie połączona z rolą:

```
@Entity(name = "users")
public class User {
    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    private Long id;
    @Column(nullable = false, unique = true)
    private String username;
    private String password;
    private int enabled;
    @ManyToMany(cascade = CascadeType.ALL, fetch = FetchType.EAGER)
    @JoinTable(name = "user_role", joinColumns = @JoinColumn(name = "user_id"),
                    inverseJoinColumns = @JoinColumn(name = "role_id"))
    private Set<Role> roles;
    //gettery, settery
}
```

Repozytoria

Dla naszych encji tworzymy odpowiadające im repozytoria:

Dla klasy **Role**:

```
@Repository
public interface RoleRepository extends JpaRepository<Role, Integer> {
    Role findByName(String name);
}
```

Dla klasy **User**:

```
@Repository
public interface UserRepository extends JpaRepository<User, Long> {
    User findByUsername(String username);
}
```

Serwis UserService

Tworzymy interfejs dla serwisu odpowiadającego za operacje na użytkownikach:

```
public interface UserService {
    User findByUserName(String name);

    void saveUser(User user);
}
```

Umieszczamy w nim metody, którymi posłużymy się w kolejnych etapach.

Implementacja serwisu

```
@Service
public class UserServiceImpl implements UserService {

    private final UserRepository userRepository;
    private final RoleRepository roleRepository;
    private final BCryptPasswordEncoder passwordEncoder;

    public UserServiceImpl(UserRepository userRepository,
        RoleRepository roleRepository, BCryptPasswordEncoder passwordEncoder) {
        this.passwordEncoder = passwordEncoder;
        this.userRepository = userRepository;
        this.roleRepository = roleRepository;
    }
}
```

Implementacja serwisu

```
@Service
public class UserServiceImpl implements UserService {

    private final UserRepository userRepository;
    private final RoleRepository roleRepository;
    private final BCryptPasswordEncoder passwordEncoder;

    public UserServiceImpl(UserRepository userRepository,
        RoleRepository roleRepository, BCryptPasswordEncoder passwordEncoder) {
        this.passwordEncoder = passwordEncoder;
        this.userRepository = userRepository;
        this.roleRepository = roleRepository;
    }
}
```

Za pomocą konstruktora wstrzykujemy potrzebne nam serwisy.

Implementacja serwisu

```
@Service
public class UserServiceImpl implements UserService {

    private final UserRepository userRepository;
    private final RoleRepository roleRepository;
    private final BCryptPasswordEncoder passwordEncoder;

    public UserServiceImpl(UserRepository userRepository,
        RoleRepository roleRepository, BCryptPasswordEncoder passwordEncoder) {
        this.passwordEncoder = passwordEncoder;
        this.userRepository = userRepository;
        this.roleRepository = roleRepository;
    }
}
```

Naszą uwagę może zwrócić brak adnotacji **@Autowired** - jeżeli bean ma jeden konstruktor, możemy ją pominąć: https://docs.spring.io/spring-boot/docs/current/reference/html/using-boot-spring-beans-and-dependency-

Serwis UserService

Uzupełniamy serwis o implementację wymaganych metod:

```
@Override
public User findByUserName(String username) {
    return userRepository.findByUsername(username);
}
@Override
public void saveUser(User user) {
    user.setPassword(passwordEncoder.encode(user.getPassword()));
    user.setEnabled(1);
    Role userRole = roleRepository.findByName("ROLE_USER");
    user.setRoles(new HashSet<Role>(Arrays.asList(userRole)));
    userRepository.save(user);
}
```

Serwis UserService

Uzupełniamy serwis o implementację wymaganych metod:

```
@Override
public User findByUserName(String username) {
    return userRepository.findByUsername(username);
}
@Override
public void saveUser(User user) {
    user.setPassword(passwordEncoder.encode(user.getPassword()));
    user.setEnabled(1);
    Role userRole = roleRepository.findByName("ROLE_USER");
    user.setRoles(new HashSet<Role>(Arrays.asList(userRole)));
    userRepository.save(user);
}
```

Przy pomocy wstrzykniętego repozytorium zwracamy użytkownika pobranego na podstawie nazwy użytkownika.

Serwis UserService

Uzupełniamy serwis o implementację wymaganych metod:

```
@Override
public User findByUserName(String username) {
    return userRepository.findByUsername(username);
}
@Override
public void saveUser(User user) {
    user.setPassword(passwordEncoder.encode(user.getPassword()));
    user.setEnabled(1);
    Role userRole = roleRepository.findByName("ROLE_USER");
    user.setRoles(new HashSet<Role>(Arrays.asList(userRole)));
    userRepository.save(user);
}
```

Metoda pozwoli nam utworzyć testowego użytkownika w dalszym etapie.

Serwis UserService

Uzupełniamy serwis o implementację wymaganych metod:

```
@Override
public User findByUserName(String username) {
    return userRepository.findByUsername(username);
}
@Override
public void saveUser(User user) {
    user.setPassword(passwordEncoder.encode(user.getPassword()));
    user.setEnabled(1);
    Role userRole = roleRepository.findByName("ROLE_USER");
    user.setRoles(new HashSet<Role>(Arrays.asList(userRole)));
    userRepository.save(user);
}
```

Ustawiamy hasło, ale wcześniej je enkodujemy.

Serwis UserService

Uzupełniamy serwis o implementację wymaganych metod:

```
@Override
public User findByUserName(String username) {
    return userRepository.findByUsername(username);
}
@Override
public void saveUser(User user) {
    user.setPassword(passwordEncoder.encode(user.getPassword()));
    user.setEnabled(1);
    Role userRole = roleRepository.findByName("ROLE_USER");
    user.setRoles(new HashSet<Role>(Arrays.asList(userRole)));
    userRepository.save(user);
}
```

Ustawiamy aktywność użytkownika.

Serwis UserService

Uzupełniamy serwis o implementację wymaganych metod:

```
@Override
public User findByUserName(String username) {
    return userRepository.findByUsername(username);
}
@Override
public void saveUser(User user) {
    user.setPassword(passwordEncoder.encode(user.getPassword()));
    user.setEnabled(1);
    Role userRole = roleRepository.findByName("ROLE_USER");
    user.setRoles(new HashSet<Role>(Arrays.asList(userRole)));
    userRepository.save(user);
}
```

Przy pomocy repozytorium pobieramy rolę, którą domyślnie nadamy użytkownikowi.

Serwis UserService

Uzupełniamy serwis o implementację wymaganych metod:

```
@Override
public User findByUserName(String username) {
    return userRepository.findByUsername(username);
}
@Override
public void saveUser(User user) {
    user.setPassword(passwordEncoder.encode(user.getPassword()));
    user.setEnabled(1);
    Role userRole = roleRepository.findByName("ROLE_USER");
    user.setRoles(new HashSet<Role>(Arrays.asList(userRole)));
    userRepository.save(user);
}
```

Ustawiamy role.

Serwis UserService

Uzupełniamy serwis o implementację wymaganych metod:

```
@Override
public User findByUserName(String username) {
    return userRepository.findByUsername(username);
}
@Override
public void saveUser(User user) {
    user.setPassword(passwordEncoder.encode(user.getPassword()));
    user.setEnabled(1);
    Role userRole = roleRepository.findByName("ROLE_USER");
    user.setRoles(new HashSet<Role>(Arrays.asList(userRole)));
    userRepository.save(user);
}
```

Zapisujemy użytkownika.

Serwis UserDetailsService

Tworzymy serwis implementujący interfejs:

org.springframework.security.core.userdetails.UserDetailsService;

Interfejs ten zawiera jedną metodę:

```
UserDetails loadUserByUsername(String username) throws UsernameNotFoundException;
```

Metoda ta ma zwrócić obiekt implementujący **UserDetail** - jest to obiekt, który zawiera podstawowe informacje o użytkowniku wymagane do jego autoryzacji.

W celu utworzenia tego obiektu posłużymy się konstruktorem klasy:

org.springframework.security.core.userdetails.User

Serwis UserDetailsService

```
public class SpringDataUserDetailsService implements UserDetailsService {

    private UserService userService;

    @Autowired
    public void setUserRepository(UserService userService) {
        this.userService = userService;
    }
}
```

Serwis UserDetailsService

```
public class SpringDataUserDetailsService implements UserDetailsService {

    private UserService userService;

    @Autowired
    public void setUserRepository(UserService userService) {
        this.userService = userService;
    }
}
```

Wstrzykujemy naszą implementację serwisu do zarządzania użytkownikami.

Serwis UserDetailsService

Uzupełniamy serwis o implementację wymaganej metody

```
@Override
public UserDetails loadUserByUsername(String username) {
    User user = userService.findByUserName(username);
    if (user == null) {throw new UsernameNotFoundException(username); }
    Set<GrantedAuthority> grantedAuthorities = new HashSet<>();
    for (Role role : user.getRoles()) {
        grantedAuthorities.add(new SimpleGrantedAuthority(role.getName()));
    }
    return new org.springframework.security.core.userdetails.User(
                    user.getUsername(), user.getPassword(), grantedAuthorities);
}
```

Serwis UserDetailsService

Uzupełniamy serwis o implementację wymaganej metody

```
@Override
public UserDetails loadUserByUsername(String username) {
    User user = userService.findByUserName(username);
    if (user == null) {throw new UsernameNotFoundException(username); }
    Set<GrantedAuthority> grantedAuthorities = new HashSet<>();
    for (Role role : user.getRoles()) {
        grantedAuthorities.add(new SimpleGrantedAuthority(role.getName()));
    }
    return new org.springframework.security.core.userdetails.User(
                    user.getUsername(), user.getPassword(), grantedAuthorities);
}
```

Za pomocą metody serwisu wyszukujemy użytkownika.

Serwis UserDetailsService

Uzupełniamy serwis o implementację wymaganej metody

```
@Override
public UserDetails loadUserByUsername(String username) {
    User user = userService.findByUserName(username);
    if (user == null) {throw new UsernameNotFoundException(username); }
    Set<GrantedAuthority> grantedAuthorities = new HashSet<>();
    for (Role role : user.getRoles()) {
        grantedAuthorities.add(new SimpleGrantedAuthority(role.getName()));
    }
    return new org.springframework.security.core.userdetails.User(
                    user.getUsername(), user.getPassword(), grantedAuthorities);
}
```

Zwracamy wyjątek jeżeli użytkownik nie istnieje.

Serwis UserDetailsService

Uzupełniamy serwis o implementację wymaganej metody

```
@Override
public UserDetails loadUserByUsername(String username) {
    User user = userService.findByUserName(username);
    if (user == null) {throw new UsernameNotFoundException(username); }
    Set<GrantedAuthority> grantedAuthorities = new HashSet<>();
    for (Role role : user.getRoles()) {
        grantedAuthorities.add(new SimpleGrantedAuthority(role.getName()));
    }
    return new org.springframework.security.core.userdetails.User(
                    user.getUsername(), user.getPassword(), grantedAuthorities);
}
```

Tworzymy listę ról wymaganych w konstruktorze klasy **org.springframework.security.core.userdetails.User** .

Serwis UserDetailsService

Uzupełniamy serwis o implementację wymaganej metody

```
@Override
public UserDetails loadUserByUsername(String username) {
    User user = userService.findByUserName(username);
    if (user == null) {throw new UsernameNotFoundException(username); }
    Set<GrantedAuthority> grantedAuthorities = new HashSet<>();
    for (Role role : user.getRoles()) {
        grantedAuthorities.add(new SimpleGrantedAuthority(role.getName()));
    }
    return new org.springframework.security.core.userdetails.User(
                    user.getUsername(), user.getPassword(), grantedAuthorities);
}
```

Uzupełniamy listę ról wykorzystując w tym celu podstawową implementację **GrantedAuthority** dostarczoną przez **Spring Security** - **SimpleGrantedAuthority** .

Serwis UserDetailsService

Uzupełniamy serwis o implementację wymaganej metody

```
@Override
public UserDetails loadUserByUsername(String username) {
    User user = userService.findByUserName(username);
    if (user == null) {throw new UsernameNotFoundException(username); }
    Set<GrantedAuthority> grantedAuthorities = new HashSet<>();
    for (Role role : user.getRoles()) {
        grantedAuthorities.add(new SimpleGrantedAuthority(role.getName()));
    }
    return new org.springframework.security.core.userdetails.User(
                    user.getUsername(), user.getPassword(), grantedAuthorities);
}
```

Zwracamy uzupełniony obiekt.

Modyfikacja konfiguracji

Modyfikujemy klasę **SecurityConfig**, tak by wykorzystała utworzone przez nas elementy:

```
@Bean
public SpringDataUserDetailsService customUserDetailsService() {
    return new SpringDataUserDetailsService();
}
```

Tworzymy bean - to wystarczy by **Spring Security** automatycznie skorzystał z naszej implementacji . https://docs.spring.io/spring-security/site/docs/current/reference/html/jc.html#jc-authentication-userdetailsservice

Modyfikacja konfiguracji

Pamiętajmy żeby usunąć wcześniejszy kod określający sposób autoryzacji, czyli metodę:

```
@Override
public void configure(AuthenticationManagerBuilder auth) throws Exception {
    auth.jdbcAuthentication().dataSource(dataSource)
    .passwordEncoder(passwordEncoder());
}
```

Pobranie informacji o użytkowniku

Bardzo powszechnym zagadnieniem jest pobranie informacji o aktualnie zalogowanym użytkowniku.

Jednym ze sposobów jest wstrzyknięcie odpowiedniego obiektu za pomocą adnotacji **@AuthenticationPrincipal**

```
@GetMapping("/admin")
@ResponseBody
public String admin2(@AuthenticationPrincipal UserDetails customUser) {
    log.info("customUSer class {} " , customUser.getClass());
    return "this is user " + customUser;
}
```

Wadą takiego podejścia jest brak informacji o powiązaniu zalogowanego użytkownika z użytkownikiem którego dane są przechowywane w bazie.

Pobranie informacji o użytkowniku

Utworzymy własną klasę, która będzie rozszerzać:
`org.springframework.security.core.userdetails.User;`

Następnie zmodyfikujemy serwis **SpringDataUserDetailsService**, tak by zamiast domyślnej klasy **Spring Security** korzystał z naszej implementacji.

Pobranie informacji o użytkowniku

Nasza implementacja wygląda następująco:

```
package pl.coderslab.service;
import java.util.Collection;
import org.springframework.security.core.GrantedAuthority;
import org.springframework.security.core.userdetails.User;
public class CurrentUser extends User {
        private final pl.coderslab.entity.User user;
        public CurrentUser(String username, String password, Collection<?
            extends GrantedAuthority> authorities,
                        pl.coderslab.entity.User user) {
                super(username, password, authorities); this.user = user;
        }
        public pl.coderslab.entity.User getUser() {return user;}
}
```

Pobranie informacji o użytkowniku

Nasza implementacja wygląda następująco:

```
package pl.coderslab.service;
import java.util.Collection;
import org.springframework.security.core.GrantedAuthority;
import org.springframework.security.core.userdetails.User;
public class CurrentUser extends User {
        private final pl.coderslab.entity.User user;
        public CurrentUser(String username, String password, Collection<?
            extends GrantedAuthority> authorities,
                        pl.coderslab.entity.User user) {
                super(username, password, authorities); this.user = user;
        }
        public pl.coderslab.entity.User getUser() {return user;}
}
```

Rozszerzamy klasę **org.springframework.security.core.userdetails.User**.

Pobranie informacji o użytkowniku

Nasza implementacja wygląda następująco:

```
package pl.coderslab.service;
import java.util.Collection;
import org.springframework.security.core.GrantedAuthority;
import org.springframework.security.core.userdetails.User;
public class CurrentUser extends User {
        private final pl.coderslab.entity.User user;
        public CurrentUser(String username, String password, Collection<?
            extends GrantedAuthority> authorities,
                        pl.coderslab.entity.User user) {
                super(username, password, authorities); this.user = user;
        }
        public pl.coderslab.entity.User getUser() {return user;}
}
```

Dodajemy właściwość, która będzie przechowywać obiekt naszej aplikacji powiązany z bazą danych .

Pobranie informacji o użytkowniku

Nasza implementacja wygląda następująco:

```
package pl.coderslab.service;
import java.util.Collection;
import org.springframework.security.core.GrantedAuthority;
import org.springframework.security.core.userdetails.User;
public class CurrentUser extends User {
        private final pl.coderslab.entity.User user;
        public CurrentUser(String username, String password, Collection<?
            extends GrantedAuthority> authorities,
                        pl.coderslab.entity.User user) {
                super(username, password, authorities); this.user = user;
        }
        public pl.coderslab.entity.User getUser() {return user;}
}
```

Wywołujemy konstruktor klasy nadrzędnej oraz przypisujemy wartość dla naszej nowej właściwości .

Pobranie informacji o użytkowniku

Nasza implementacja wygląda następująco:

```
package pl.coderslab.service;
import java.util.Collection;
import org.springframework.security.core.GrantedAuthority;
import org.springframework.security.core.userdetails.User;
public class CurrentUser extends User {
        private final pl.coderslab.entity.User user;
        public CurrentUser(String username, String password, Collection<?
            extends GrantedAuthority> authorities,
                        pl.coderslab.entity.User user) {
                super(username, password, authorities); this.user = user;
        }
        public pl.coderslab.entity.User getUser() {return user;}
}
```

Getter dla naszej nowej właściwości.

Pobranie informacji o użytkowniku

Modyfikujemy serwis **SpringDataUserDetailsService**

zmieniając linie:

```
return new org.springframework.security.core.userdetails.User(user.getUsername()
            , user.getPassword(), grantedAuthorities);
```

na:

```
return new CurrentUser(user.getUsername(),user.getPassword(),
            grantedAuthorities, user);
```

Pobranie informacji o użytkowniku

Modyfikujemy serwis **SpringDataUserDetailsService**

zmieniając linie:

```
return new org.springframework.security.core.userdetails.User(user.getUsername()
            , user.getPassword(), grantedAuthorities);
```

na:

```
return new CurrentUser(user.getUsername(),user.getPassword(),
            grantedAuthorities, user);
```

Tworzymy i wypełniamy obiekt naszej nowej klasy.

Pobranie informacji o użytkowniku

Następnie przy pomocy adnotacji **@AuthenticationPrincipal** wstrzykujemy odpowiedni obiekt:

```
@GetMapping("/admin")
@ResponseBody
public String admin(@AuthenticationPrincipal CurrentUser customUser) {
    User entityUser = customUser.getUser();
    return "this is user id " +entityUser.getId() ;
}
```

Dokumentacja dla adnotacji **@AuthenticationPrincipal**: https://docs.spring.io/spring-security/site/docs/current/reference/html/mvc.html#mvc-authentication-principal

Pobranie informacji o użytkowniku

Następnie przy pomocy adnotacji **@AuthenticationPrincipal** wstrzykujemy odpowiedni obiekt:

```
@GetMapping("/admin")
@ResponseBody
public String admin(@AuthenticationPrincipal CurrentUser customUser) {
    User entityUser = customUser.getUser();
    return "this is user id " +entityUser.getId() ;
}
```

Dokumentacja dla adnotacji **@AuthenticationPrincipal**: https://docs.spring.io/spring-security/site/docs/current/reference/html/mvc.html#mvc-authentication-principal

Wstrzykujemy do akcji kontrolera obiekt klasy **CurrentUser**.

Podsumowanie

Przedstawiony przykład jest tylko jednym z wielu sposobów na konfigurację **Spring Security** oraz **Spring Data JPA**.

Dość powszechnym sposobem jest utworzenie własnych encji a następnie modyfikacja odpowiednich zapytań, z których korzysta **Spring Security**.

Przykład takiej konfiguracji znajdziemy w tutorialu:

https://medium.com/@gustavo.ponce.ch/spring-boot-spring-mvc-spring-security-mysql-a5d8545d837d

Przykład ten został praktycznie sprawdzony.

Listę zapytań, jakie wykorzystuje używana w tym przypadku klasa **JdbcUserDetailsManager**, znajdziemy pod adresem:

https://github.com/spring-projects/spring-security/blob/master/core/src/main/java/org/springframework/security/provisioning/JdbcUserDetailsManager.java

Przydatne linki

Warto zapoznać się z implementacją udostępnioną na oficjalnym przez twórców **Springa** na ich repozytorium GitHub:

https://github.com/spring-projects/spring-boot/tree/master/spring-boot-samples/spring-boot-sample-web-secure-jdbc

Opis architektury **Spring Security**

https://spring.io/guides/topicals/spring-security-architecture/