

# 付録

## Spring Security

- A-1 Spring Securityを振り返ろう
- A-2 カスタマイズ設定を行おう
- A-3 許可処理をプログラムに取り込もう
- A-4 カスタムエラーページを作ろう
- A-5 カスタム属性を追加しよう

Appendix

A-1

# Spring Security 振り返ろう

Spring超入門を最後までお読みいただきありがとうございました。ここからは、書籍内に収まりきらなかった「ToDoアプリ」に対してのSpring Securityを使用した「カスタムログイン設定」、「認可機能」の導入について解説します。

## A-1-1 Spring Security 振り返り

Spring Securityは、認証や認可、およびその他多くのセキュリティ対策を簡易に実装できる、Springプロジェクトが提供するフレームワークです(図A.1)。多機能であるため、ビギナーの方にはやや敷居が高いと言われています。

図A.1 Spring Security



### □ 認証と認可

Spring Securityが提供する「認証」と「認可」について、表A.1に示します。

表A.1 認証と認可

言葉	説明
認証 (Authentication)	ユーザーが自分の身元を証明する方法。簡単に言うと「ログイン」のことです
認可 (Authorization)	認証されたユーザーが特定のリソースにアクセスできるかどうかを決定する方法。簡単に言うと「権限」のことです

## A-1-2 カスタマイズ設定の概要

Spring Securityのデフォルト設定を「書籍：Spring超入門」で説明してきましたが、実際にSpring Securityを使用する場合、デフォルト設定ではアプリケーションの要件を完全に満たせない場合があります。そんな時は、実際の使用状況に合わせてセキュリティ設定をカスタマイズします。

Spring Securityのカスタマイズは、アプリケーションのセキュリティ要件に応じて、様々なセキュリティ機能を調整するために重要です。このカスタマイズには、URLベースのアクセス制御、カスタムログインフォームの作成、さまざまな認証メカニズムの設定などが含まれます。

ビギナーの方にとっては、Spring Securityのカスタマイズ設定が難しく感じられるかもしれませんが、理解の第一歩として覚えておくべき重要なキーワードが「UserDetails」と「UserService」です。多くの設定項目がありますが、まずはこれらのキーワードを意識してください(表A.2)。

表A.2 「UserDetails」と「UserService」

言葉	説明
UserDetails	認証されるユーザーの情報を表すインターフェース
UserService	UserDetailsを認証するためのサービス

Appendix

# A-2 カスタマイズ設定を行おう

Spring Securityのデフォルト設定は、基本的なセキュリティ要件について学ぶのに適していますが、多くのアプリケーションでは、より具体的なカスタマイズが必要になります。デフォルト設定からカスタマイズへの変更を行うことで、アプリケーションのセキュリティを強化することができます。

## A-2-1 カスタマイズ設定：ログイン画面表示

自分だけのログイン画面（カスタムログイン画面）を作成し、データベースに保存されたデータを用いて認証を行うために、Spring Securityの設定をデフォルトからカスタマイズへ変更しましょう。

「view」→「form」→「controller」→「設定」の順で作成します。

### 01 ログイン画面

カスタムのログイン画面を作成します。

設定内容

親フォルダを入力または選択	webapp/src/main/resources/templates
ファイル名	login.html

※ 他はデフォルト設定

「login.html」の内容はリスト A.1 のようになります。

リスト A.1 login.html

```
001: <!DOCTYPE html>
002: <html xmlns:th="http://www.thymeleaf.org">
003: <head>
004:   <title>ログイン</title>
005: </head>
006: <body>
007:   <h2>ログイン画面</h2>
008:   <!-- エラー表示 -->
```

```
009:   <div th:if="${param.error}">
010:     <p style="color: red;">usernameまたはpasswordが違います</p>
011:   </div>
012:   <!-- ログアウト表示 -->
013:   <div th:if="${param.logout}">
014:     <p style="color: blue;">ログアウトしました</p>
015:   </div>
016:   <form th:action="@{/authentication}" method="post" th:object="${loginForm}">
017:     <div><!-- ユーザー名 -->
018:       <label for="usernameInput">username</label>
019:       <input type="text" th:field="*{usernameInput}">
020:     </div>
021:     <div><!-- パスワード -->
022:       <label for="passwordInput">password</label>
023:       <input type="password" th:field="*{passwordInput}">
024:     </div>
025:     <div>
026:       <input type="submit" value="login" />
027:     </div>
028:   </form>
029: </body>
030: </html>
```

9行目の「<div th:if="\${param.error}">」は、Spring Securityのデフォルト動作に基づくものです。認証に失敗すると、リダイレクトされたログインページのURLに「?error」というパラメータが追加されるため、それを使ってエラーメッセージを画面に表示しています。Thymeleafはこのリクエストパラメータを検出し、エラーメッセージを表示するために「param.error」を使用します。

10行目では、「ユーザー名またはパスワードが間違っています」という一般的なエラーメッセージを表示します。具体的な理由を示すのではなく、このような一般的なメッセージを使うことで、攻撃者<sup>(注1)</sup>に余計な情報を与えずに済みます。

13行目の「<div th:if="\${param.logout}">」では、Spring Securityの標準動作により、ログアウト成功後にリダイレクトされたログインページのURLに「?logout」が追加されます。Thymeleafはこのリクエストパラメータを検出し、適切なメッセージを表示するために「param.logout」を使用します。

16行目の「th:action="@{/authentication}"」は、ユーザー認証処理を行うURLを指定します。「th:object="\${loginForm}"」では、バックエンドの「loginForm」オブジェクトのプロパティ（この例では「usernameInput」と「passwordInput」）を、19行目のユーザー名フィールドと23行目のパスワードフィールドにThymeleafの「th:field」を使ってバインドします。

(注1) ログイン画面を攻撃する攻撃者は、不正にアクセス権を得ることを目的としています。

02 Formの作成

カスタムのログイン画面で使用するFormを作成します。

設定内容

パッケージ	com.example.webapp.form
名前	LoginForm

「LoginForm」クラスの内容はリスト A.2 のようになります。

リスト A.2 LoginForm

```
001: package com.example.webapp.form;
002:
003: import lombok.Data;
004:
005: @Data
006: public class LoginForm {
007:     /** ユーザー名 */
008:     private String usernameInput;
009:     /** パスワード */
010:     private String passwordInput;
011: }
```

03 Controllerの作成

カスタムのログイン画面を表示するコントローラを作成します。

設定内容

パッケージ	com.example.webapp.controller
名前	LoginController

「LoginController」クラスの内容はリスト A.3 のようになります。

リスト A.3 LoginController

```
001: package com.example.webapp.controller;
002:
003: import org.springframework.stereotype.Controller;
004: import org.springframework.web.bind.annotation.GetMapping;
005: import org.springframework.web.bind.annotation.ModelAttribute;
006: import org.springframework.web.bind.annotation.RequestMapping;
```

```
007:
008: import com.example.webapp.form.LoginForm;
009:
010: @Controller
011: @RequestMapping("/login")
012: public class LoginController {
013:
014:     @GetMapping
015:     public String showLogin(@ModelAttribute LoginForm form) {
016:         // templatesフォルダ配下のlogin.htmlに遷移
017:         return "login";
018:     }
019: }
```

15行目「@ModelAttribute LoginForm form」で、カスタムログイン画面の入力データを保持する LoginForm オブジェクトをメソッドの引数として定義しています。この定義によりカスタムログイン画面のフォームの入力フィールドと LoginForm クラスのプロパティが自動的に結びつけられます。

04 セキュリティ設定

カスタマイズ設定にて使用するセキュリティ設定を作成します。

設定内容

パッケージ	com.example.webapp.config
名前	SecurityConfig

「SecurityConfig」クラスの内容はリスト A.4 のようになります。

リスト A.4 SecurityConfig

```
001: package com.example.webapp.config;
002:
003: import org.springframework.context.annotation.Bean;
004: import org.springframework.context.annotation.Configuration;
005: import org.springframework.security.config.annotation.web.builders.HttpSecurity;
006: import org.springframework.security.config.annotation.web.configuration.
007:     EnableWebSecurity;
008: import org.springframework.security.web.SecurityFilterChain;
009:
010: @Configuration
011: @EnableWebSecurity
012: public class SecurityConfig {
```

```
012:
013: // SecurityFilterChainのBean定義
014: @Bean
015: public SecurityFilterChain filterChain(HttpSecurity http) throws Exception {
016:     http
017:         // ★HTTPリクエストに対するセキュリティ設定
018:         .authorizeHttpRequests(authz -> authz
019:         // 「/login」へのアクセスは認証を必要としない
020:         .requestMatchers("/login").permitAll()
021:         // その他のリクエストは認証が必要
022:         .anyRequest().authenticated())
023:         // ★フォームベースのログイン設定
024:         .formLogin(form -> form
025:         // カスタムログインページのURLを指定
026:         .loginPage("/login")
027:         );
028:     return http.build();
029: }
030: }
```

このソースコードは、Spring Securityを使用してWebアプリケーションのセキュリティ設定をカスタマイズするための設定クラスです。このクラスで、どのURLにアクセスするために認証が必要か、ログイン処理の扱い方など、セキュリティ関連のカスタマイズを定義しています。

9行目の「@Configuration」は、このクラスがSpringの設定クラスであることを示します。10行目の「@EnableWebSecurity」は、Spring Securityを有効化し、Webセキュリティの設定をカスタマイズすることを意味します。つまり、この2つのアノテーションをクラスに付けることで、このクラスがセキュリティのカスタマイズ設定を担当することを表します。

14行目～29行目では、SecurityFilterChainのBeanを定義することで、HTTPリクエストに対するセキュリティ設定を行います。SecurityFilterChainは、Webリクエストがサーバーに到達する際に一連のセキュリティチェックを実施します。

17行目～22行目は、HTTPリクエストに対するセキュリティ設定です。

18行目「.authorizeHttpRequests(authz -> authz」は、どのHTTPリクエストに認証が必要かを定義しています。authz -> authzの部分は、ラムダ式です。ラムダ式 (Lambda Expression) は、Java 8で導入された機能で、簡潔にコードを記述するための方法です。

20行目「.requestMatchers("/login").permitAll()」は、「/login」へのアクセスは認証を必要としないことを指定しています。つまり、ログインページは誰でもアクセス可能です。

22行目「.anyRequest().authenticated()」は、その他の全てのリクエストには認証が必要であることを指定しています。

23行目～27行目では、フォームベースのログイン設定をしており、「.formLogin(form -> form.loginPage("/login"))」でカスタムログインページのURLを設定しています。28行目の「return http.build();」は、セキュリティ設定を構築し、最終的なSecurityFilterChainオブジェクトを返すために使用されます。

05 ここまでの動作確認

「Bootダッシュボード」で「webapp」を選択し、プロジェクトを起動します。ブラウザを立ち上げURL「http://localhost:8080/」を指定すると「カスタムログイン画面」が表示されます(図A.2)。

図A.2 カスタムログイン画面



A-2-2 カスタマイズ設定：認証処理①

自分で作成したカスタムログイン画面の表示ができたので、次は「カスタムログイン画面」でデフォルトの認証処理を実行可能に修正しましょう。

01 認証処理の修正

独自のログイン処理を追加するために、セキュリティ設定を修正します。

修正内容

パッケージ	com.example.webapp.config
名前	SecurityConfig

「SecurityConfig」クラスの内容をリストA.5のように修正します。

リストA.5 SecurityConfig

```
001: 既存コード
002: ...
003: // ★フォームベースのログイン設定
004: .formLogin(form -> form
005: // カスタムログインページのURLを指定
006: .loginPage("/login")
007: // ログイン処理のURLを指定
008: .loginProcessingUrl("/authentication")
009: // ユーザー名のname属性を指定
010: .usernameParameter("usernameInput")
011: // パスワードのname属性を指定
```



```
012:     .passwordParameter("passwordInput")
013:     // ログイン成功時のリダイレクト先を指定
014:     .defaultSuccessUrl("/")
015:     // ログイン失敗時のリダイレクト先を指定
016:     .failureUrl("/login?error")
017: );
018: return http.build();
019: ...
020: 既存コード
```

7行目から16行目が修正部分です。以下に修正部分について説明します。

● ログイン処理のURLを指定

8行目「`loginProcessingUrl("/authentication")`」は、ユーザーがログインフォームを送信する際に使用するURLを指定しています。つまり認証処理を実行するURLを指定しています。何も設定しない場合、Spring Securityはデフォルトで「`/login`」をログイン処理のURLとして使用しますが、ここではカスタム設定としてURL「`/authentication`」に変更しています。

● ユーザー名とパスワードのname属性を指定

10行目「`usernameParameter("usernameInput")`」は、ログインフォーム内のユーザー名入力フィールドのname属性を指定します。何も設定しない場合、Spring Securityはデフォルトで「`username`」をname属性に使用しますが、ここでは「`usernameInput`」に変更しています。

12行目「`passwordParameter("passwordInput")`」は、ログインフォーム内のパスワード入力フィールドのname属性を指定します。何も設定しない場合、Spring Securityはデフォルトで「`password`」をname属性に使用しますが、ここでは「`passwordInput`」に変更しています。

● ログイン成功時と失敗時のリダイレクト先を指定

14行目「`defaultSuccessUrl("/")`」は、ログイン成功時のリダイレクト先を指定します。デフォルトでは、Spring Securityはログイン前にユーザーがアクセスしようとしていたページにリダイレクトしますが、ここではメニュー画面を表示するURL「`/`」を設定しています。

16行目「`failureUrl("/login?error")`」は、ログイン失敗時のリダイレクト先を指定します。デフォルトでは、ログイン失敗時にログインページにリダイレクトされますが、ここではエラーパラメータ「`error`」を追加することで、ログインに失敗したことをユーザーに知らせるメッセージをビュー側で表示できます。

02 ここまでの動作確認

「Bootダッシュボード」で「webapp」を選択し、プロジェクトを起動します。ブラウザを立ち上げURL「`http://localhost:8080/`」を指定すると「カスタムログイン画面」が表示されます。

「カスタムログイン画面」にて「`username`」に「`user`」、「`password`」にターミナルに表示されたランダムな文字列を入力して、「`login`」ボタンをクリックすると、自分が作成したセキュリティ設定通りに「メニュー画面」が表示されます。図A.3に認証失敗時に表示される画面を示します。

図A.3 認証(エラー画面)

### ログイン画面

usernameまたはpasswordが違います

username

password

login

03 ログアウト処理の修正

独自のログアウト処理を追加するために、セキュリティ設定を修正します。

● 修正内容

パッケージ	com.example.webapp.config
名前	SecurityConfig

「SecurityConfig」クラスの内容をリストA.6のように修正します。

リストA.6 SecurityConfig

```
001: 既存コード
002: ...
003:     // ログイン成功時のリダイレクト先を指定
004:     .defaultSuccessUrl("/")
005:     // ログイン失敗時のリダイレクト先を指定
006:     .failureUrl("/login?error"))
007:     // ★ログアウト設定
008:     .logout(logout -> logout
009:     // ログアウトを処理するURLを指定
010:     .logoutUrl("/logout")
011:     // ログアウト成功時のリダイレクト先を指定
012:     .logoutSuccessUrl("/login?logout")
013:     // ログアウト時にセッションを無効にする
014:     .invalidateHttpSession(true)
015:     // ログアウト時にCookieを削除する
016:     .deleteCookies("JSESSIONID")
017: );
```

```
018: return http.build();
019: ...
020: 既存コード
```

6行目～16行目が修正部分です。以下に修正部分について説明します。

● ログアウト URL の指定

10行目「`logoutUrl("/logout")`」は、ログアウト処理を行うためのURLをカスタマイズしています。デフォルトでは、Spring SecurityはURL「`/logout`」をログアウト処理のURLとして使用しますが、ここでは、わかりやすいように設定を明示的に指定しています。

● ログアウト成功時のリダイレクト先

12行目「`logoutSuccessUrl("/login?logout")`」は、ユーザーがログアウトに成功した後にリダイレクトされるURLを指定しています。デフォルトでは、Spring Securityはログインページにリダイレクトしますが、ここではクエリパラメータ`logout`を追加することで、ログアウトしたことをユーザーに知らせるメッセージをビュー側で表示できます。

● セッションの無効化

14行目「`invalidateHttpSession(true)`」は、ログアウト時にHTTPセッションを無効にする設定です。これはセキュリティのベストプラクティスであり、セッションハイジャック<sup>(注2)</sup>を防ぐのに役立ちます。デフォルト設定では、この挙動は有効になっていますが、わかりやすいように記述しています。

● クッキーの削除

16行目「`deleteCookies("JSESSIONID")`」は、ログアウト時に特定のクッキー（ここでは`JSESSIONID`）を削除する設定です。`JSESSIONID`<sup>(注3)</sup>はセッション識別用のクッキーで、ログアウト時にこれを削除することで、古いセッションが再利用されるのを防ぎます。

04 ここまでの動作確認

アプリケーションを起動し、「カスタムログイン画面」にて「`username`」に「`user`」、「`password`」にターミナルに表示されたランダムな文字列を入力して、「`login`」ボタンをクリックして表示された「メニュー画面」で、「ログアウト」ボタンをクリックすることで、セキュリティ設定に記述した内容でログアウト処理が実行されることを確認できます。

(注2) セッションハイジャックは、攻撃者がユーザーのセッションIDを盗み、そのユーザーとしてWebアプリケーションにアクセスする攻撃のことを指します。  
(注3) `JSESSIONID`は、JavaベースのWebアプリケーションでよく使用されるクッキーです。

図 A.4 ログアウト



A-2-3 カスタマイズ設定：認証処理②

カスタマイズされた認証設定で重要なのは、「`UserDetails`」と「`UserDetailsService`」という2つのキーワードです。これらの概念を再度確認してから、実装に進みましょう。

□ UserDetails と UserDetailsService

● UserDetails

`UserDetails`は、Spring Securityで使用されるインターフェースで、ユーザーの認証情報を表すものです。このインターフェースは、認証プロセスにおいてユーザーを識別し、その権限を管理するためにSpring Securityによって使用されます。開発者は、`UserDetails`インターフェースを実装することで、アプリケーション固有のユーザー認証情報を定義できます。

● UserDetailsService

`UserDetailsService`は、指定された「ユーザー名」に基づいて「ユーザー情報（`UserDetails`の実装クラス）」をデータベースや他のデータソースから取得し、認証プロセスに利用するためのサービスを提供します。

01 UserDetails 実装クラスの作成

ユーザーの認証情報を表す`UserDetails`実装クラスを作成しましょう。  
「webapp」の「`src/main/java`」フォルダを選択し、マウスを右クリックし、「新規」→「クラス」を選択します。クラス設定画面にて以下「設定内容」を記述後、「完了」ボタンを押します。

● 設定内容

パッケージ	com.example.webapp.entity
名前	LoginUser

※ 他はデフォルト設定

「`LoginUser`」クラスの内容はリスト A.7 のようになります。

リスト A.7 LoginUser

```
001: package com.example.webapp.entity;
002:
003: import java.util.Collection;
004:
005: import org.springframework.security.core.GrantedAuthority;
006: import org.springframework.security.core.userdetails.User;
007:
008: /**
009:  * ユーザーの認証情報を表すUserDetails実装クラス
010:  */
011: public class LoginUser extends User {
012:     /** 最低限の情報を保持したUserDetails
013:      * 実装クラスUserを作成する */
014:     public LoginUser(String username,
015:         String password,
016:         Collection<? extends GrantedAuthority> authorities) {
017:         super(username, password, authorities);
018:     }
019: }
```

6行目「import org.springframework.security.core.userdetails.User;」は、Spring Securityが提供する UserDetails インターフェースを簡易実装したクラスです。UserDetails インターフェースを実装して認証用クラスを作成する場合、様々なメソッドをオーバーライドしなければいけません。今回は必要最低限の機能を利用するため、Spring Securityが提供する「User」クラスという簡易版を利用しています。そのため11行目で「extends User」としています。

17行目「super(username, password, authorities);」は User クラスのコンストラクタです。コンストラクタの各引数について表A.2に記述します。

表 A.2 コンストラクタの引数

型	引数	説明
String	username	ユーザー名
String	password	パスワード
Collection <? extends GrantedAuthority>	authorities	ユーザーに割り当てられた権限のリスト。権限は、ユーザーが実行可能な操作やアクセス可能なリソースを定義するために使用されます。つまり認可で使用されます

02 UserDetailsService 実装クラスの作成

認証サービスを提供する UserDetailsService 実装クラスを作成しましょう。

「webapp」の「src/main/java」フォルダを選択し、マウスを右クリックし、「新規」→「クラス」を選択します。クラス設定画面にて以下「設定内容」を記述後、「完了」ボタンを押します。

○ 設定内容

パッケージ	com.example.webapp.service.impl
名前	LoginUserDailsServiceImpl

※ 他はデフォルト設定

「LoginUserDailsServiceImpl」クラスの内容はリスト A.8 のようになります。

リスト A.8 LoginUserDailsServiceImpl

```
001: package com.example.webapp.service.impl;
002:
003: import java.util.Collections;
004:
005: import org.springframework.security.core.userdetails.UserDetails;
006: import org.springframework.security.core.userdetails.UserDetailsService;
007: import org.springframework.security.core.userdetails.UsernameNotFoundException;
008: import org.springframework.stereotype.Service;
009:
010: import com.example.webapp.entity.LoginUser;
011:
012: /**
013:  * カスタム認証サービス
014:  */
015: @Service
016: public class LoginUserDailsServiceImpl implements UserDetailsService {
017:     @Override
018:     public UserDetails loadUserByUsername(String username)
019:         throws UsernameNotFoundException {
020:         // 「ユーザー名：tarou」が入力されると、UserDetailsの実装クラスを返す
021:         if (username.equals("tarou")) {
022:             // 対象データが存在する
023:             // UserDetailsの実装クラスを返す
024:             return new LoginUser("tarou",
025:                 "pass",
026:                 Collections.emptyList());
027:         } else {
028:             // 対象データが存在しない
029:             throw new UsernameNotFoundException(
030:                 username + " => 指定しているユーザー名は存在しません");
031:         }
032:     }
033: }
```



16行目「implements UserDetailsService」をすることで、UserDetailsService インターフェースの認証メソッドである「UserDetails loadUserByUsername(String username) throws UsernameNotFoundException」をオーバーライドする必要があります。

処理内容は、指定されたユーザー名 (username) を受け取り、現時点では固定値でユーザー名 (tarou) とパスワード (pass)、そして権限を持たない (Collections.emptyList())、「ユーザーの認証情報」を担う UserDetails 実装クラスを返却しています。返却された UserDetails 実装クラスを利用して Spring Security が認証を実施してくれます。

UsernameNotFoundException は、ユーザーが見つからなかったことを示す例外です。

03 PasswordEncoder の作成

認証サービスで利用するパスワードエンコーダーを作成します。パスワードエンコーダーとは、ユーザーのパスワードを安全に保管するためにパスワードをエンコード (ハッシュ化) するための仕組みです。

「webapp」の「src/main/java」フォルダを選択し、マウスを右クリックし、「新規」→「クラス」を選択します。クラス設定画面にて以下「設定内容」を記述後、「完了」ボタンを押します。

設定内容

パッケージ	com.example.webapp.config
名前	PasswordConfig

※ 他はデフォルト設定

「PasswordConfig」クラスの内容はリスト A.9 のようになります。

リスト A.9 PasswordConfig

```
001: package com.example.webapp.config;
002:
003: import org.springframework.context.annotation.Bean;
004: import org.springframework.context.annotation.Configuration;
005: import org.springframework.security.crypto.password.NoOpPasswordEncoder;
006: import org.springframework.security.crypto.password.PasswordEncoder;
007:
008: @Configuration
009: public class PasswordConfig {
010:     @Bean
011:     public PasswordEncoder passwordEncoder() {
012:         // エンコードの設定
013:         return NoOpPasswordEncoder.getInstance();
014:     }
015: }
```

8行目で「@Configuration」アノテーションを使用することにより、このクラスが Spring の設定クラスであることを示しています。このアノテーションによって、Spring はこのクラス内のメソッドから Bean (Spring によって管理されるオブジェクト) を生成し、それをアプリケーションのコンテナに登録します。

10行目の「@Bean」アノテーションは、メソッドが Bean を生成することを示します。Spring はこのメソッドが返すオブジェクトを依存性注入 (DI) の対象として扱います。

13行目の「return NoOpPasswordEncoder.getInstance();」では、NoOperation (何もしない) のパスワードエンコーダーを使用しており、これによりパスワードは変換されずにそのままの形で保存されます。NoOpPasswordEncoder は、実際のアプリケーションでの使用は推奨されていません。この方法ではパスワードが平文で保存されるため、セキュリティリスクが非常に高くなりますが、現在は理解を容易にするために使用しています。

04 セキュリティ設定の修正

セキュリティ設定へ、先ほど作成した「LoginUserDetailsServiceImpl」と「PasswordEncoder」を使用するようにソースコードを修正します。

修正内容

パッケージ	com.example.webapp.config
名前	SecurityConfig

「SecurityConfig」クラスへの修正内容はリスト A.10 のようになります。

リスト A.10 SecurityConfig

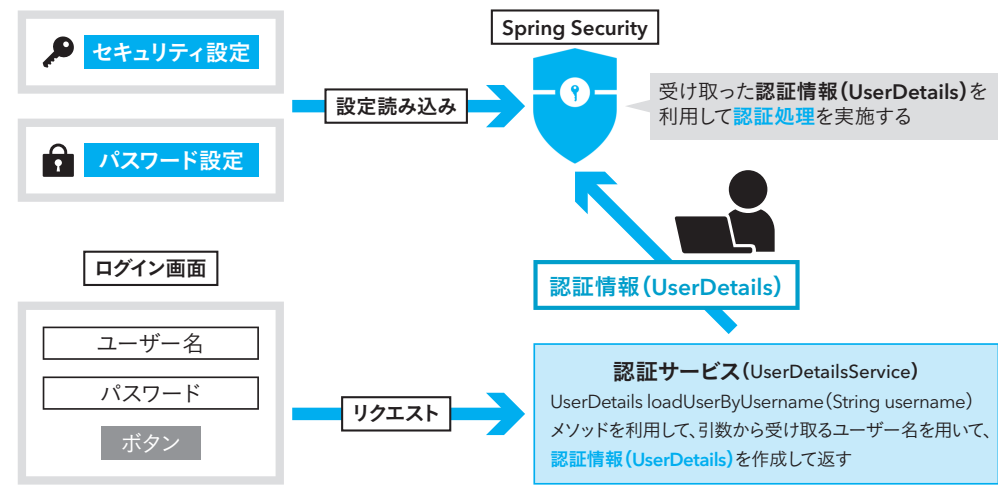
```
001: 既存コード
002: ...
003:
004: @Configuration
005: @EnableWebSecurity
006: @RequiredArgsConstructor
007: public class SecurityConfig {
008:
009:     /** DI対象が存在すれば、DIして使用する */
010:     private final UserDetailsServiceImpl userDetailsServiceImpl;
011:     private final PasswordEncoder passwordEncoder;
012:
013:     ...
014: 既存コード
```

6行目「@RequiredArgsConstructor」と10行目、11行目の final を使用しているフィールドが

ら UserDetailsService と PasswordEncoder がコンストラクションされ、カスタマイズされた認証方法が適用されます。

ここまで作成した内容のイメージを以下に示します (図 A.5)。

図 A.5 カスタム認証イメージ



05 ここまでの動作確認

アプリケーションを起動し、「カスタムログイン画面」にて「username」に「tarou」、「password」に「pass」を入力して (図 A.6)、「login」ボタンをクリックして表示された「メニュー画面」で、「ログアウト」ボタンをクリックすることでログアウト処理が実行されることを確認します (図 A.7)。

図 A.6 カスタム認証

ログイン画面

username

password

図 A.7 ログアウト

ログイン画面

ログアウトしました

username

password

A-2-4 カスタマイズ設定：データベースからの取得

次は、データベースに認証専用のテーブルを作成し、そのデータを使用して認証を実行するように「カスタマイズ設定」していきましょう。

01 認証用テーブルの作成

○ schema.sql の作成

「src/main/resources」フォルダ配下のファイル名「schema.sql」を修正します。修正内容をリスト A.11 に記述します。

リスト A.11 schema.sql

```
001: -- テーブルが存在したら削除する
002: DROP TABLE IF EXISTS todos;
003: DROP TABLE IF EXISTS authentications;
004:
005: ...
006: 既存コード
007: ...
008:
009: -- 認証情報を格納するテーブル
010: CREATE TABLE authentications (
011:   -- ユーザー名：主キー
012:   username VARCHAR(50) PRIMARY KEY,
013:   -- パスワード
014:   password VARCHAR(255) NOT NULL
015: );
```

3 行目「DROP TABLE IF EXISTS」を利用して、テーブルが存在したら削除する処理に「認証テーブル」を追加しています。

10 行目～15 行目で、認証処理に使用する認証テーブルを作成しています。

ER 図を図 A.8 に示します。

図 A.8 ER 図

todos		authentications	
id	serial	username	VARCHAR(50)
todo	varchar(255) NN	password	VARCHAR(255) NN
detail	text		
created_at	timestamp		
updated_at	timestamp		

○ data.sql の作成

「src/main/resources」フォルダ配下のファイル名「data.sql」の末尾に認証テーブルへのダミーデータ登録処理を追加します。修正内容をリスト A.12 に記述します。

リスト A.12 data.sql

```
001: ...
002:     既存コード
003: ...
004:
005: -- 認証テーブルへのダミーデータの追加
006: INSERT INTO authentications (username, password) VALUES ('admin', 'adminpass');
```

schema.sql で作成した認証テーブルに対して、ダミーデータを登録する INSERT 文を記述しています。

## 02 エンティティの作成

認証情報用のエンティティクラスを作成しましょう。

「webapp」の「src/main/java」フォルダを選択し、マウスを右クリックし、「新規」→「クラス」を選択します。クラス設定画面にて以下「設定内容」を記述後、「完了」ボタンを押します。

### 設定内容

パッケージ	com.example.webapp.entity
名前	Authentication

※ 他はデフォルト設定

「Authentication」クラスの内容はリスト A.13 のようになります。

リスト A.13 Authentication

```
001: package com.example.webapp.entity;
002:
003: import lombok.AllArgsConstructor;
004: import lombok.Data;
005: import lombok.NoArgsConstructor;
006:
007: @Data
008: @NoArgsConstructor
009: @AllArgsConstructor
010: public class Authentication {
011:     /** ユーザー名 */
012:     private String username;
013:     /** パスワード */
014:     private String password;
015: }
```

「ユーザー名」と「パスワード」をフィールドに持っています。

## 03 Repository の作成

「Repository」は「インターフェース」で作成します。「認証テーブルのデータ操作」メソッドを記述します (実装内容は記述しません)。認証テーブル用の Repository を作成します。

「webapp」の「src/main/java」フォルダを選択し、マウスを右クリックし、「新規」→「インターフェース」を選択します。インターフェース設定画面にて以下「設定内容」を記述後、「完了」ボタンを押します。

### 設定内容

パッケージ	com.example.webapp.repository
名前	AuthenticationMapper

※ 他はデフォルト設定

「AuthenticationMapper」クラスの内容はリスト A.14 のようになります。

リスト A.14 AuthenticationMapper

```
001: package com.example.webapp.repository;
002:
003: import org.apache.ibatis.annotations.Mapper;
004:
005: import com.example.webapp.entity.Authentication;
006:
007: @Mapper
008: public interface AuthenticationMapper {
009:
010:     /**
011:      * ユーザー名でログイン情報を取得します。
012:      */
013:     Authentication selectByUsername(String username);
014: }
```

ユーザー名でログイン情報を取得するメソッドを定義します。

## 04 マッパーファイルの作成

「webapp」の「src/main/resources」フォルダを選択し、マウスを右クリックし、「新規」→「その他」を選択します。ウィザード選択画面にて、ウィザードに「mybatis」と入力し表示される「MyBatis XML Mapper」を選択後、「次へ」ボタンをクリックします。

MyBatis XML マッパー画面にて以下「設定内容」を記述後、「完了」ボタンを押します。

○ 設定内容

親フォルダを入力または選択	webapp/src/main/resources/com/example/webapp/repository
ファイル名	AuthenticationMapper

※ 他はデフォルト設定

「AuthenticationMapper」XML ファイルの内容はリスト A.15 のようになります。

リスト A.15 AuthenticationMapper

```
001: <?xml version="1.0" encoding="UTF-8"?>
002: <!DOCTYPE mapper PUBLIC "-//mybatis.org//DTD Mapper 3.0//EN" "http://mybatis.org/dtd/mybatis-3-mapper.dtd">
003: <mapper namespace="com.example.webapp.repository.AuthenticationMapper">
004:     <!-- ユーザー名で認証情報を検索 -->
005:     <select id="selectByUsername" resultType="com.example.webapp.entity.
006:         Authentication">
007:         SELECT username, password FROM authentications WHERE username =
008:             #{username}
009:     </select>
010: </mapper>
```

5行目の「resultType」には認証テーブルにあたるエンティティ「Authentication」クラスをFQCNで設定します。

6行目の「username = #{username}」で使用している「#{フィールド名}」はMyBatisの「プレースホルダ」です。

05 UserDetailsService 実装クラスの修正

認証サービスを提供するUserDetailsService実装クラスを、作成した「認証テーブル」からデータを取得するように修正しましょう。

○ 修正内容

パッケージ	com.example.webapp.service.impl
名前	LoginUserDetailsServiceImpl

「LoginUserDetailsServiceImpl」クラスの修正内容はリスト A.16 のようになります。

リスト A.16 LoginUserDetailsServiceImpl

```
001: package com.example.webapp.service.impl;
002:
003: import java.util.Collections;
004:
005: import org.springframework.security.core.userdetails.UserDetails;
006: import org.springframework.security.core.userdetails.UserDetailsService;
007: import org.springframework.security.core.userdetails.UsernameNotFoundException;
008: import org.springframework.stereotype.Service;
009:
010: import com.example.webapp.entity.Authentication;
011: import com.example.webapp.entity.LoginUser;
012: import com.example.webapp.repository.AuthenticationMapper;
013:
014: import lombok.RequiredArgsConstructor;
015:
016: /**
017:  * カスタム認証サービス
018:  */
019: @Service
020: @RequiredArgsConstructor
021: public class LoginUserDetailsServiceImpl implements UserDetailsService {
022:     /** DI */
023:     private final AuthenticationMapper authenticationMapper;
024:
025:     @Override
026:     public UserDetails loadUserByUsername(String username)
027:         throws UsernameNotFoundException {
028:         // 「認証テーブル」からデータを取得
029:         Authentication authentication = authenticationMapper.selectByUsername(username);
030:
031:         // 対象データがあれば、UserDetailsの実装クラスを返す
032:         if (authentication != null) {
033:             // 対象データが存在する
034:             // UserDetailsの実装クラスを返す
035:             return new LoginUser(authentication.getUsername(),
036:                 authentication.getPassword(),
037:                 Collections.emptyList()
038:             );
039:         } else {
040:             // 対象データが存在しない
041:             throw new UsernameNotFoundException(
042:                 username + " => 指定しているユーザー名は存在しません");
043:         }
044:     }
045: }
```

20行目「@RequiredArgsConstructor」と23行目のfinalを使用しているフィールドからAuthenticationMapperがコンストラクティンジェクションされ、認証情報をDBから取得する処理が追加されています。

29行目「authenticationMapper.selectByUsername(username);」で、usernameを使用して、DBから認証情報を取得します。

35～38行目で「認証テーブル」から取得したデータを利用して、ユーザーの認証情報を表すUserDetails実装クラスを作成し、戻り値として返します。

□ ここまでの動作確認

アプリケーションを起動し、「カスタムログイン画面」にて「username」に「admin」、「password」に「adminpass」を入力後(図A.9)、「login」ボタンをクリックし「メニュー画面」を表示することで、DBから認証情報を取得してログインできることを確認します(図A.10)。

図A.9 DBから認証情報を取得



図A.10 認証成功



06 PasswordEncoderの修正

現状の平文でのパスワード保存は大きなセキュリティリスクをもたらします。ハッシュ化されたパスワードを使用するように修正し、セキュリティの強化を高めましょう。

○ 修正内容

パッケージ	com.example.webapp.config
名前	PasswordConfig

「PasswordConfig」クラスの修正内容はリストA.17のようになります。

リストA.17 PasswordConfig

```
001: package com.example.webapp.config;
002:
003: import org.springframework.context.annotation.Bean;
004: import org.springframework.context.annotation.Configuration;
005: import org.springframework.security.crypto.bcrypt.BCryptPasswordEncoder;
006: import org.springframework.security.crypto.password.PasswordEncoder;
007:
008: @Configuration
009: public class PasswordConfig {
010:     @Bean
011:     public PasswordEncoder passwordEncoder() {
012:         // エンコードの設定
013:         return new BCryptPasswordEncoder();
014:     }
015: }
```

13行目「return new BCryptPasswordEncoder();」は、Spring Securityにおいてパスワードを安全にハッシュ化するために使用されるクラスの一つです。BCryptという特定のハッシュ関数を用いて、パスワードをハッシュ化します。この設定により、ハッシュ化されたパスワードを認証できるようになります。

07 パスワードの作成

ハッシュ化したパスワードを生成するクラスを作成します。今回作成するアプリケーションの機能とは関係ないプログラムになりますが、このクラスで作成されたハッシュ化した文字列を認証テーブルのパスワードの値として使用します。

「webapp」の「src/main/java」フォルダを選択し、マウスを右クリックし、「新規」→「クラス」を選択します。クラス設定画面にて以下「設定内容」を記述後、「完了」ボタンを押します。

○ 設定内容

パッケージ	com.example.webapp.utility
名前	PasswordGenerator

※ 他はデフォルト設定

「PasswordGenerator」クラスの内容はリストA.18のようになります。



リスト A.18 PasswordGenerator

```
001: package com.example.webapp.utility;
002:
003: import org.springframework.security.crypto.bcrypt.BCryptPasswordEncoder;
004:
005: /**
006:  * ハッシュ化した文字列を返すクラス
007:  */
008: public class PasswordGenerator {
009:     public static void main(String[] args) {
010:         // 「BCrypt」のインスタンス化
011:         BCryptPasswordEncoder encoder = new BCryptPasswordEncoder();
012:         // 入力値
013:         String rawPassword = "adminpass";
014:         // パスワードをハッシュ化
015:         String encodedPassword = encoder.encode(rawPassword);
016:         // 表示
017:         System.out.println("ハッシュ化されたパスワード: " + encodedPassword);
018:     }
019: }
```

13行目で平文対象の文字列を設定し、15行目の「BCryptPasswordEncoder」の「encode」メソッドを使用してハッシュ化した文字列を生成し、17行目でハッシュ化した文字列を表示します。

PasswordGenerator ファイルを選択し、右クリック→実行→Java アプリケーションを行い(図 A.11)、平文の「adminpass」をハッシュ化します(図 A.12)。

図 A.11 実行

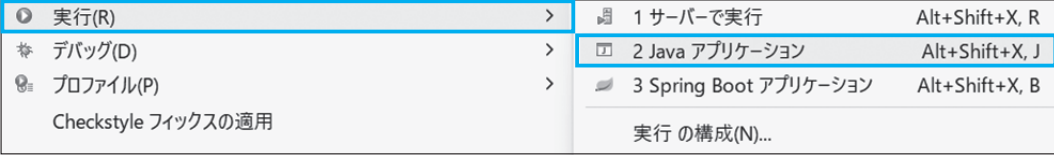


図 A.12 ハッシュ化

```
|ハッシュ化されたパスワード: $2a$10$1NH4dLsCH4/g7aZZq14QG.PvnC7rkeN395ZWanW/hT0i5k6y009mm
```

今回は認証情報の CRUD 処理を作成することができませんが、認証テーブルへのパスワード登録時には、「平文」を「ハッシュ化」した「値」を認証用テーブルに「INSERT」してください。

● ハッシュ化されたパスワードの設定

ハッシュ化した文字列を認証テーブルのパスワードとして使用したいと思います。「src/main/resources」フォルダ配下のファイル名「data.sql」の認証テーブルへのダミーデータ登録処理を修

正します。修正内容をリスト A.19 に記述します。

リスト A.19 data.sql

```
001: 既存コード
002: ...
003:
004: -- 認証テーブルへのダミーデータの追加
005: INSERT INTO authentications (username, password) VALUES
006: ('admin', '$2a$10$1NH4dLsCH4/g7aZZq14QG.PvnC7rkeN395ZWanW/hT0i5k6y009mm');
```

PasswordGenerator ファイルで作成した、ハッシュ化された文字列を password に登録します。

アプリケーションを起動し、「カスタムログイン画面」にて「username」に「admin」、「password」に「adminpass」を入力後、「login」ボタンをクリックし「メニュー画面」を表示することで、DB に登録している「ハッシュ化されたパスワード」を利用してログインできることを確認します。

Column | Spring Security を使用するうえでの注意点

Spring Security を使用する際に注意すべきポイントは、主に以下のようなものがあります。

- 互換性  
使用している Spring Framework や Spring Boot のバージョンに対して、適切な Spring Security のバージョンを選択し使用してください。互換性のないバージョンを使用することで、予期しないエラーや問題が発生する可能性があります。
- 非推奨の機能と新機能や記述方法  
新バージョンでは、旧バージョンの機能が非推奨となり、新しい機能が追加されることがあります。また、記述方法もバージョンによって変わります。これらの変更を理解し、適切にコードを更新する必要があります。
- ドキュメンテーションとリリースノートの確認  
バージョンアップする場合、変更点や新機能、改善されたセキュリティ対策については、公式のドキュメンテーションやリリースノートを確認してください。

□ バージョン確認

本書執筆時点の Spring Framework のバージョンは「6.1.4」、Spring Boot のバージョンは「3.2.3」、Spring Security のバージョンは「6.2.2」です。バージョンを確認する方法は色々ありますが、ここではプログラムで確認してみましょう。

各ライブラリのバージョン確認を実行するクラスを作成します。

「webapp」の「src/main/java」フォルダを選択し、マウスを右クリックし、「新規」→「クラス」を選択します。クラス設定画面にて以下「設定内容」を記述後、「完了」ボタンを押します。

設定内容

パッケージ	com.example.webapp.utility
名前	SpringVersionCheck

※ 他はデフォルト設定

「SpringVersionCheck」クラスの内容はリスト A.20 のようになります。

リスト A.20 SpringVersionCheck

```
001: package com.example.webapp.utility;
002:
003: import org.springframework.boot.SpringBootVersion;
004: import org.springframework.core.SpringVersion;
005: import org.springframework.security.core.SpringSecurityCoreVersion;
006:
007: public class SpringVersionCheck {
008:     public static void main(String[] args) {
009:         // Spring Frameworkのバージョン
010:         String springVersion = SpringVersion.getVersion();
011:         System.out.println("Spring Frameworkのバージョン: " + springVersion);
012:         // Spring Bootのバージョン
013:         String bootVersion = SpringBootVersion.getVersion();
014:         System.out.println("Spring Bootのバージョン: " + bootVersion);
015:         // Spring Securityのバージョン
016:         String securityVersion = SpringSecurityCoreVersion.getVersion();
017:         System.out.println("Spring Securityのバージョン: " + securityVersion);
018:     }
019: }
```

Spring Framework、Spring Boot、Spring Security のバージョンを確認しています。

SpringVersionCheck ファイルを選択し、右クリック→実行→Java アプリケーションを行うことで、バージョンの確認ができます (図 A.13)。

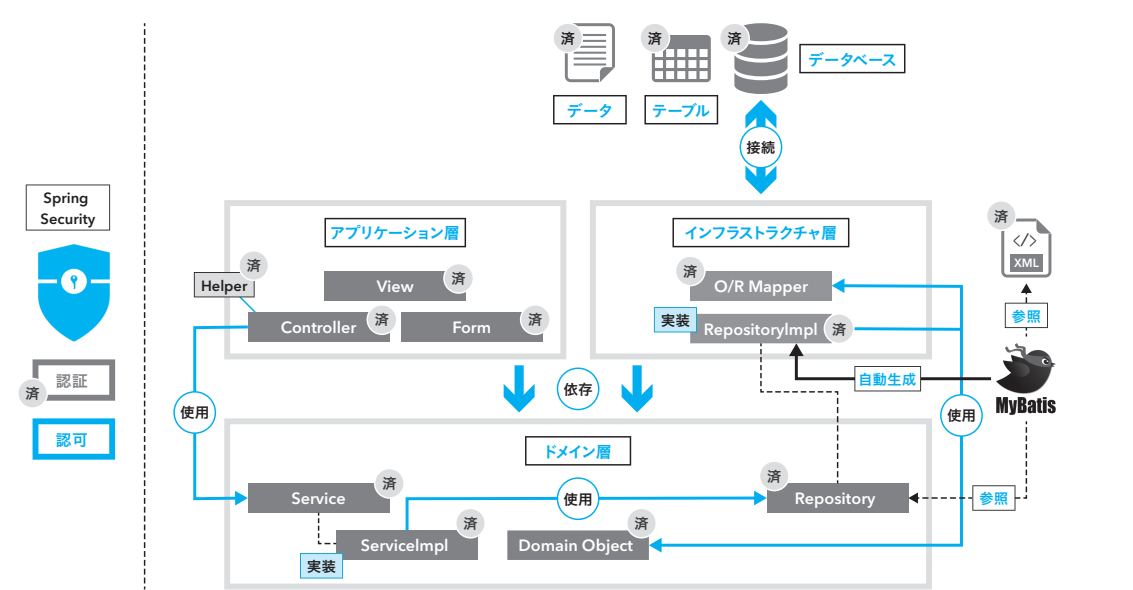
図 A.13 バージョン確認

```
Spring Frameworkのバージョン: 6.1.4
Spring Bootのバージョン: 3.2.3
05:15:57.570 [main] INFO org.springframework.security.core.SpringSecurityCoreVersion
Spring Securityのバージョン: 6.2.2
```

Spring Security のカスタマイズ設定を使用して、認証処理を実装できました。次は、Spring Security の認可を実装しましょう。

ここまでの進捗を以下に示します (図 A.14)。

図 A.14 ここまでの進捗



Appendix

A-3

認可処理をプログラムに取り込む

認可 (Authorization) は、ユーザーやシステムが特定のリソースへのアクセスや操作を許可するプロセスのことを指します。つまり「権限」です。「認可」処理をアプリケーションに取り入れましょう。

A-3-1 権限の追加

01 schema.sql の修正

「src/main/resources」フォルダ配下のファイル名「schema.sql」を修正します。修正内容をリスト A.21 に記述します。

リスト A.21 schema.sql

```
001: -- テーブルが存在したら削除する
002: DROP TABLE IF EXISTS todos;
003: DROP TABLE IF EXISTS authentications;
004: DROP TYPE IF EXISTS role;
005:
006: ...
007: 既存コード
008: ...
009:
010: -- 権限用のENUM型
011: CREATE TYPE role AS ENUM ('ADMIN', 'USER');
012:
013: -- 認証情報を格納するテーブル
014: CREATE TABLE authentications (
015:   -- ユーザー名：主キー
016:   username VARCHAR(50) PRIMARY KEY,
017:   -- パスワード
018:   password VARCHAR(255) NOT NULL,
019:   -- 権限
020:   authority role NOT NULL
021: );
```

今回追加する ENUM 型を削除する処理を 4 行目に追加します。

11 行目の「CREATE TYPE role AS ENUM ('ADMIN', 'USER');」では、「role」という新しい ENUM 型を作成しています。ENUM 型は、限定された一連の許可された値を持つデータ型です。この場合、「role」は「ADMIN」と「USER」の 2 つの値を持つことができます。

20 行目の「authority role NOT NULL」は、権限を格納するカラムを表します。ここでは、11 行目で作成された「role」という ENUM 型を使用し、NOT NULL 制約を設定しています。これは、カラム「authority」に「role」で定義された「ADMIN」と「USER」のみが設定可能であることを意味します。

02 data.sql の修正

「src/main/resources」フォルダ配下のファイル名「data.sql」を修正します。修正内容をリスト A.22 に記述します。

リスト A.22 data.sql

```
001: ...
002: 既存コード
003: ...
004:
005: -- 認証テーブルへのダミーデータの追加
006: -- password : adminpass
007: INSERT INTO authentications (username, password, authority) VALUES
008:   ('admin', '$2a$10$1NH4dLsCH4/g7aZZq14QG.PvnC7rkeN395ZWanW/hTOi5k6y009mm', 'ADMIN');
009: -- password : userpass
010: 1INSERT INTO authentications (username, password, authority) VALUES
011:   ('user', '$2a$10$/jar9xXQ6lrnVjLvLGv5BepFkLnGI049RrGx42p2i.1hQt1BZ/7E2', 'USER');
```

authority 列には、ENUM 型 role で設定した「ADMIN」と「USER」を設定しています。

03 エンティティの修正

認証情報用のエンティティクラスに権限フィールドを追加します。その権限フィールドに設定するための ENUM クラスを作成します。

「webapp」の「src/main/java」フォルダを選択し、マウスを右クリックし、「新規」→「列挙型」を選択します。列挙型設定画面にて以下「設定内容」を記述後、「完了」ボタンを押します。

設定内容

パッケージ	com.example.webapp.entity
名前	Role

※ 他はデフォルト設定

「Role」列挙型の内容はリスト A.23 のようになります。

リスト A.23 Role

```
001: package com.example.webapp.entity;
002:
003: public enum Role {
004:     ADMIN, USER
005: }
```

Javaの列挙型 (Enum) は、固定された定数のセットを表すために使用される特別なクラスタイプです。列挙型は、特定の値グループ (例えば、月火水木金土日などの曜日や、東西南北などの方向) を限定したグループとして定義するのに適しています。今回は、権限で使用する「ADMIN」、「USER」を列挙型として定義しています。

次に、認証用エンティティ「Authentication」に対して、列挙型Roleのフィールドを追加します。

修正内容

パッケージ	com.example.webapp.entity
名前	Authentication

「Authentication」クラスの修正内容はリスト A.24 のようになります。

リスト A.24 Authentication

```
001: package com.example.webapp.entity;
002:
003: import lombok.AllArgsConstructor;
004: import lombok.Data;
005: import lombok.NoArgsConstructor;
006:
007: @Data
008: @NoArgsConstructor
009: @AllArgsConstructor
010: public class Authentication {
011:     /** ユーザー名 */
012:     private String username;
013:     /** パスワード */
014:     private String password;
015:     /** 権限 */
016:     private Role authority;
017: }
```

列挙型Roleを指定した権限フィールドを追加します。

04 マッパーファイルの作成

認証用のマッパーファイルを修正します。

修正内容

親フォルダを入力または選択	webapp/src/main/resources/com/example/webapp/repository
ファイル名	AuthenticationMapper

「AuthenticationMapper」XML ファイルの修正内容はリスト A.25 のようになります。

リスト A.25 AuthenticationMapper

```
001: <?xml version="1.0" encoding="UTF-8"?>
002: <!DOCTYPE mapper PUBLIC "-//mybatis.org//DTD Mapper 3.0//EN" "http://mybatis.org/dtd/mybatis-3-mapper.dtd">
003: <mapper namespace="com.example.webapp.repository.AuthenticationMapper">
004:     <!-- ユーザー名で認証情報を検索 -->
005:     <select id="selectByUsername" resultType="com.example.webapp.entity.Authentication">
006:         SELECT username, password, authority FROM
007:             authentications WHERE username = #{username}
008:     </select>
009: </mapper>
```

権限にあたる authority フィールドを SELECT 文で取得できるように修正します。

05 LoginUserDatailsServiceImpl の修正

認証サービスを提供する「LoginUserDatailsServiceImpl」を「権限」を設定するように修正します。

修正内容

パッケージ	com.example.webapp.service.impl
ファイル名	LoginUserDatailsServiceImpl

「LoginUserDatailsServiceImpl」クラスの修正内容はリスト A.26 のようになります。

リスト A.26 LoginUserDetailsServiceImpl

```
001: 既存コード
002: ...
003:
004:     if (authentication != null) {
005:         // 対象データが存在する
006:         // UserDetailsの実装クラスを返す
007:         return new LoginUser(authentication.getUsername(),
008:             authentication.getPassword(),
009:             getAuthorityList(authentication.getAuthority()));
010:     };
011: } else {
012:     // 対象データが存在しない
013:     throw new UsernameNotFoundException(
014:         username + " => 指定しているユーザー名は存在しません");
015: }
016: }
017:
018: /**
019:  * 権限情報をリストで取得する
020:  */
021: private List<GrantedAuthority> getAuthorityList(Role role) {
022:     // 権限リスト
023:     List<GrantedAuthority> authorities = new ArrayList<>();
024:     // 列挙型からロールを取得
025:     authorities.add(new SimpleGrantedAuthority(role.name()));
026:     // ADMIN ロールの場合、USERの権限も付与
027:     if (role == Role.ADMIN) {
028:         authorities.add(
029:             new SimpleGrantedAuthority(Role.USER.toString()));
030:     }
031:     return authorities;
032: }
033: }
```

9行目で呼んでいる「getAuthorityListメソッド」は、特定のユーザーの権限情報に基づいて、そのユーザーが持つ権限をリスト形式で返します。

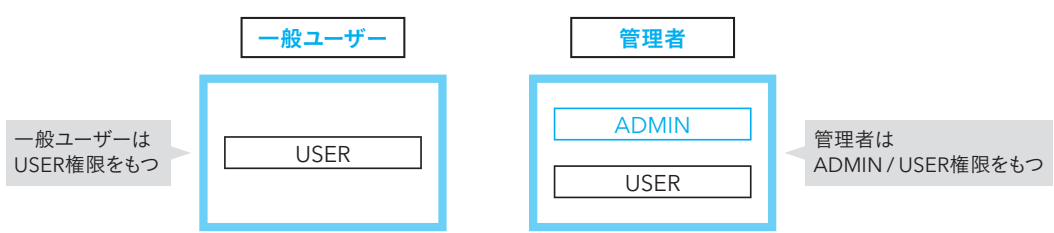
21行目で使用する「Role」はリスト A.23 で作成した列挙型のRoleになります。

23行目「GrantedAuthority」は、Spring Securityが提供する「権限」を表すために使用されるインターフェースです。

25行目「SimpleGrantedAuthority」は、Spring Securityが提供するGrantedAuthorityインターフェースを簡易実装したクラスです。

27行目～30行目で、ユーザーがADMIN権限（管理者）を持つ場合、USER権限もリストに追加しています。これにより、管理者ユーザーはADMIN権限、USER権限の両方を持ちます（図A.15）。

図 A.15 権限



06 メニュー画面へのリンク追加

ログアウトボタンがメニュー画面にしかないので、ToDo一覧画面にメニュー画面へのリンクを追加します。

修正内容

親フォルダを入力または選択	webapp/src/main/resources/templates/todo
ファイル名	list.html

ToDo一覧画面のlist.htmlの修正内容はリスト A.27 のようになります。

リスト A.27 list.html

```
001: ...
002: 既存コード
003: ...
004: </table>
005: <a th:href="@{/todos/form}">新規登録</a>
006: <br>
007: <!-- メニュー画面へ -->
008: <a th:href="@{/}">メニュー画面へ</a>
009: </body>
010: </html>
```

A-3-2 アクセスコントロール

DBの認証テーブルから権限を取得することができました。取得した権限を利用して、アクセスコントロールを設定します。アクセスコントロールとは、システムにおいて、リソースへのアクセスを制御・管理するプロセスを指します。

01 Thymeleafへの設定

権限によるアクセスコントロールを、「メニュー画面」に追加します。



修正内容

親フォルダを入力または選択	webapp/src/main/resources/templates
ファイル名	menu.html

「menu.html」の修正内容はリスト A.28 のようになります。

リスト A.28 menu.html

```
001: <!DOCTYPE html>
002: <html xmlns:th="http://www.thymeleaf.org"
003:       xmlns:sec="http://www.thymeleaf.org/thymeleaf-extras-springsecurity6">
004: <head>
005:   <title>メニュー</title>
006: </head>
007: <body>
008:   <h2>メニュー画面</h2>
009:   <hr>
010:   <a sec:authorize="hasAuthority('ADMIN')" th:href="@{/todos}">
011:     ToDo一覧へ
012:   </a>
013:   <br>
014:   <div sec:authorize="hasAuthority('USER')">
015:     【一般権限で表示される部分】
016:   </div>
017:   <div sec:authentication="name">
018:     ログイン情報のname値に書き換わる
019:   </div>
020:   <div sec:authorize="isAuthenticated()">
021:     認証されたユーザーにのみ表示
022:   </div>
023:   <br>
024:   <!-- ログアウト -->
025:   <form th:action="@{/logout}" method="post">
026:     <input type="submit" value="ログアウト">
027:   </form>
028: </body>
029: </html>
```

10行目の「sec:authorize="hasAuthority('ADMIN')」は、ログイン中のユーザーが「ADMIN」権限を持っている場合のみ、囲まれたコンテンツ（この例では「ToDo一覧へ」のリンク）を表示するために使用します。ユーザーが「ADMIN」権限を持っているかどうかのチェックは、Spring Securityによって行われます。

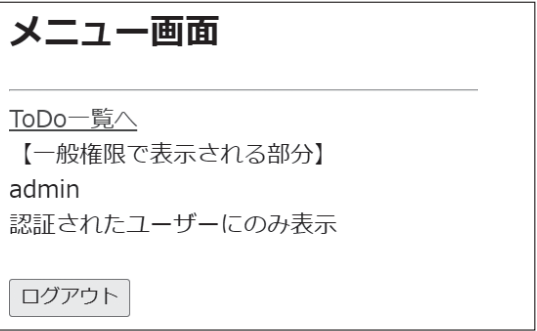
14行目の「sec:authorize="hasAuthority('USER')」も同様に、ログイン中のユーザーが「USER」権限を持っている場合のみ、囲まれたコンテンツ（この例では「文字列」）を表示します。この「文字列」に特別な意味はありませんが、認可処理の確認のために使用しています。

02 ここまでの動作確認

アプリケーションを起動し、「カスタムログイン画面」にて「username」に「admin」、「password」に「adminpass」を入力後、「login」ボタンをクリックし「メニュー画面」を表示します。

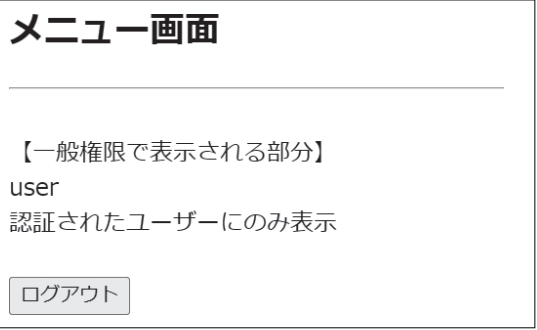
username:adminは管理者権限を持っています。管理者権限は、一般ユーザー権限も含むため、メニュー画面では「ToDo一覧」へのリンクと一般ユーザー権限で表示する「文字列」が表示されま

図 A.16 メニュー画面（管理者権限）



ログアウト実行後、ログイン画面にて「username」に「user」、「password」に「userpass」を入力後、「login」ボタンをクリックし「メニュー画面」を表示します。Username:userは一般ユーザー権限を持っています。そのためメニュー画面では「文字列」が表示されます（図 A.17）。

図 A.17 メニュー画面（一般ユーザー権限）



Thymeleafを使って権限に基づいて表示内容を制御できることが確認できました。しかし、ブラウザのアドレスバーに「http://localhost:8080/todos」と入力すると、一般ユーザー権限を持つユーザーでも、本来は管理者権限のみがアクセスできるはずのToDo一覧画面が表示されてしま

います（図 A.18）。この問題を解決するために、セキュリティ設定を修正しましょう。

図A.18 ToDo一覧

ToDo一覧

ID	ToDo	詳細
1	買い物	<a href="#">詳細</a>
2	図書館に行く	<a href="#">詳細</a>
3	ジムに行く	<a href="#">詳細</a>

[新規登録](#)  
[メニュー画面へ](#)

02 セキュリティ設定の修正

セキュリティ設定にアクセスコントロールを追加する修正をします。

修正内容

パッケージ	com.example.webapp.config
名前	SecurityConfig

「SecurityConfig」クラスの内容はリストA.29のようになります。

リストA.29 SecurityConfig

```
001: 既存コード
002: ...
003: // SecurityFilterChainのBean定義
004: @Bean
005: public SecurityFilterChain filterChain(HttpSecurity http) throws Exception {
006:     http
007:         // ★HTTPリクエストに対するセキュリティ設定
008:         .authorizeHttpRequests(authz -> authz
009:             // 「/login」へのアクセスは認証を必要としない
010:             .requestMatchers("/login").permitAll()
011:             // 【管理者権限設定】url:/todos/**は管理者しかアクセスできない
012:             .requestMatchers("/todos/**").hasAuthority("ADMIN")
013:             // その他のリクエストは認証が必要
014:             .anyRequest().authenticated()
015:         ...
016: 既存コード
```

12行目「requestMatchers("/todos/\*\*").hasAuthority("ADMIN")」を追加します。

「requestMatchers("/todos/\*\*")」は、URLパスが「/todos/」で始まる全てのリクエストを指定しています。「\*\*」は、このディレクトリ下のすべてのパスにマッチするワイルドカードです。

「hasAuthority("ADMIN")」は、現在認証されているユーザーが「ADMIN」という権限を持っているかどうかをチェックします。この設定により「/todos/」以下のURLには、管理者（ADMIN権限を持つユーザー）のみがアクセスできます。

03 ここまでの動作確認

アプリケーションを起動して、「カスタムログイン画面」にアクセスします。ここで「username」に「user」、そして「password」に「userpass」と入力し、「login」ボタンをクリックして「メニュー画面」を表示します。その後、ブラウザのアドレスバーに「http://localhost:8080/todos」と入力します。一般ユーザー権限を持つユーザーがこのURLにアクセスすると、ステータスコード：403がページに表示されます（図A.19）。

ステータスコード：403はHTTP応答ステータスコードの一つで、「Forbidden（禁止されている）」を意味します。このステータスコードがウェブサーバーからクライアントに送信されると、クライアントがリクエストしたリソースへのアクセスが許可されていないことを示します。

図A.19 ステータスコード：403ページ

Whitelabel Error Page

This application has no explicit mapping for /error, so you are seeing this as a fallback.

Thu Dec 28 15:27:17 JST 2023

There was an unexpected error (type=Forbidden, status=403).

Forbidden

Appendix

# A-4 カスタムエラーページを作ろう

先ほどの動作確認でステータスコード：403が発生した際、「Whitelabel Error Page」という画面が表示されました。この「Whitelabel Error Page」は、Spring Bootアプリケーションでエラーが発生したときにデフォルトで表示されるエラーページです。しかし、よりユーザーにわかりやすくするために、この「Whitelabel Error Page」をカスタムエラーページに置き換えたいと思います。

## A-4-1 カスタムエラーページの作成

Spring Bootでは、アプリケーションでエラーが発生した際に表示されるエラーページを簡単に作成することが可能です。

作成方法は、「src/main/resources/templates」ディレクトリ配下に「error」フォルダを作成し、そのフォルダ内に「エラーコード.html」ファイルを配置します。例えば、404.htmlはHTTP 404エラー（ページが見つからない）に対応し、500.htmlはHTTP 500エラー（サーバー内部エラー）に対応します。

### 01 カスタムエラーページの作成

ステータスコード：403に対応するカスタムエラーページを作成します。

○ 設定内容

親フォルダを入力または選択	webapp/src/main/resources/templates/error
ファイル名	403.html

「403.html」の内容はリスト A.30 のようになります。

リスト A.30 403.html

```
001: <!DOCTYPE html>
002: <html xmlns:th="http://www.thymeleaf.org">
003: <head>
004:   <title>403</title>
005: </head>
006: <body>
007:   <h2>403 - アクセス拒否</h2>
008:   <div>
009:     このページにアクセスする権限がありません。
010:   </div>
011:   <a th:href="@{/}">メニューへ</a>
012: </body>
013: </html>
```

ステータスコード：403に対応する、アクセス拒否がわかるようなレイアウトにしています。

### 02 ここまでの動作確認

アプリケーションを起動して、「カスタムログイン画面」にアクセスします。ここで「username」に「user」、そして「password」に「userpass」と入力し、「login」ボタンをクリックして「メニュー画面」を表示します。その後、ブラウザのアドレスバーに「http://localhost:8080/todos」と入力します。一般ユーザー権限しか持っていないので、先ほど作成したカスタムエラーページ「403.html」が表示されます（図 A.20）。

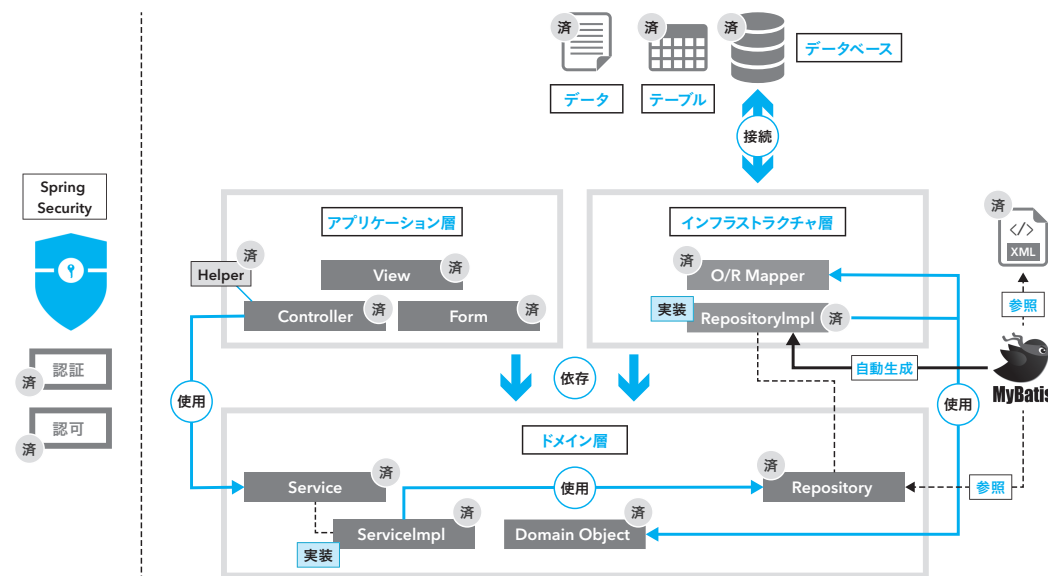
図 A.20 403.html

#### 403 - アクセス拒否

このページにアクセスする権限がありません。  
メニューへ

Springを使用したWebアプリケーションに「認可」処理を取り込みました。  
ここまでの進捗を以下に示します（図 A.21）。

図 A.21 ここまでの進捗



## Section

# A-5 カスタム属性を追加しよう

Spring Securityは、アプリケーションのセキュリティを管理するための強力なフレームワークです。ユーザーがアプリケーションにログインする際、Spring Securityはそのユーザーの詳細情報を保持するためにUserDetailsインターフェースを使用します。デフォルトでは、UserDetailsにはユーザー名、パスワード、アカウントの状態（有効/無効）、権限などの基本的な情報が含まれます。

## A-5-1 カスタム属性とは？

多くの場合、アプリケーションにはユーザーの名前やメールアドレスといった、標準のUserDetailsに含まれない追加情報を扱う必要があります。これらの追加情報を本書では「カスタム属性」と呼びます。Spring Securityでカスタム属性を扱うには、UserDetailsインターフェースを実装した独自のクラスを作成し、そこに必要な属性を追加します。

## A-5-2 カスタム属性の追加

### 01 schema.sqlの修正

「src/main/resources」フォルダ配下のファイル名「schema.sql」を修正します。修正内容をリストA.31に記述します。

#### リスト A.31 schema.sql

```
001: ...
002: 既存コード
003: ...
004:
005: -- 認証情報を格納するテーブル
006: CREATE TABLE authentications (
007:     -- ユーザー名：主キー
008:     username VARCHAR(50) PRIMARY KEY,
009:     -- パスワード
010:     password VARCHAR(255) NOT NULL,
011:     -- 権限
012:     authority role NOT NULL,
```

```
013:      -- 表示名
014:      displayname VARCHAR(50) NOT NULL
015:  );
```

画面に表示するための「displayname」属性を 14 行目に追加します。

02 data.sql の修正

「src/main/resources」フォルダ配下のファイル名「data.sql」を修正します。修正内容をリスト A.32 に記述します。

リスト A.32 data.sql

```
001:  ...
002:  既存コード
003:  ...
004:  -- 認証テーブルへのダミーデータの追加
005:  -- password : adminpass
006:  INSERT INTO authentications (username, password, authority, displayname) VALUES
007:    ('admin', '$2a$10$1NH4dLsCH4/g7aZZq14QG.PvnC7rkeN395ZWanW/hTOi5k6y009mm', 'ADMIN',
008:    '管理太郎');
009:  -- password : userpass
010:  INSERT INTO authentications (username, password, authority, displayname) VALUES
011:    ('user', '$2a$10$/jar9xXQ6l1rnVjLVLGv5BepFkLnGI049RrGx42p2i.1hQt1BZ/7E2', 'USER', '
012:    一般花子');
```

「displayname」属性に値を登録するように SQL を修正します。

03 エンティティの修正

認証用エンティティ「Authentication」に対して、表示名の「displayname」フィールドを追加します。

修正内容

パッケージ	com.example.webapp.entity
名前	Authentication

「Authentication」クラスの修正内容はリスト A.33 のようになります。

リスト A.33 Authentication

```
001: package com.example.webapp.entity;
002:
003: import lombok.AllArgsConstructor;
004: import lombok.Data;
005: import lombok.NoArgsConstructor;
006:
007: @Data
008: @NoArgsConstructor
009: @AllArgsConstructor
010: public class Authentication {
011:     /** ユーザー名 */
012:     private String username;
013:     /** パスワード */
014:     private String password;
015:     /** 権限 */
016:     private Role authority;
017:     /** 表示名 */
018:     private String displayname;
019: }
```

04 マッパーファイルの作成

認証用のマッパーファイルを修正します。

修正内容

親フォルダを入力または選択	webapp/src/main/resources/com/example/webapp/repository
ファイル名	AuthenticationMapper

「AuthenticationMapper」XML ファイルの修正内容はリスト A.34 のようになります。

リスト A.34 Authentication

```
001: <?xml version="1.0" encoding="UTF-8"?>
002: <!DOCTYPE mapper PUBLIC "-//mybatis.org//DTD Mapper 3.0//EN" "http://mybatis.org/
003:   dtd/mybatis-3-mapper.dtd">
004: <mapper namespace="com.example.webapp.repository.AuthenticationMapper">
005:     <!-- ユーザー名で認証情報を検索 -->
006:     <select id="selectByUsername" resultType="com.example.webapp.entity.
007:       Authentication">
008:         SELECT username, password, authority, displayname FROM
009:           authentications WHERE username = #{username}
010:     </select>
011: </mapper>
```



05 LoginUserの修正

認証用クラス「LoginUser」にカスタム属性を追加するフィールドを追加します。

修正内容

パッケージ	com.example.webapp.entity
ファイル名	LoginUser

「LoginUser」クラスの修正内容はリスト A.35 のようになります。

リスト A.35 LoginUser

```
001: package com.example.webapp.entity;
002:
003: import java.util.Collection;
004:
005: import org.springframework.security.core.GrantedAuthority;
006: import org.springframework.security.core.userdetails.User;
007:
008: /**
009:  * ユーザーの認証情報を表すUserDetails実装クラス
010:  */
011: public class LoginUser extends User {
012:
013:     // 【追加部分】追加のフィールド
014:     private String displayname;
015:
016:     /** 最低限の情報を保持したUserDetails
017:      * 実装クラスUserを作成する */
018:     public LoginUser(String username,
019:         String password,
020:         Collection<? extends GrantedAuthority> authorities,
021:         String displayname) { // 【追加部分】displaynameを追加
022:         super(username, password, authorities);
023:         this.displayname = displayname;
024:     }
025:
026:     // 【追加部分】displaynameのgetter
027:     public String getDisplayname() {
028:         return displayname;
029:     }
030: }
```

14行目「private String displayname;」でログイン時に使用する「username」とは別の画面に表示する名前を格納するためのフィールドを用意しています。

18行目～24行目のLoginUserクラスのコンストラクタでは、Userクラスのコンストラクタに「ユーザー名」、「パスワード」、「権限情報」を渡すことで、これらの基本的なユーザー情報を設定しています。さらに、新しく追加したカスタム属性「displayname」も引数に追加することで、オブジェクトが生成される際に、「displayname」を設定しています。

27行目～29行目の「getDisplayname メソッド」は、displayname フィールドの値を取得するためのものです。これにより、LoginUser インスタンスから表示名を取得し、アプリケーションの他の部分で 사용할 ことができます。

06 LoginUserDetailsServiceImplの修正

認証サービスを提供する「LoginUserDetailsServiceImpl」で「UserDetailsの実装クラス」を返す部分を修正します。

修正内容

パッケージ	com.example.webapp.service.impl
ファイル名	LoginUserDetailsServiceImpl

「LoginUserDetailsServiceImpl」クラスの修正内容はリスト A.36 のようになります。

リスト A.36 LoginUserDetailsServiceImpl

```
001: ...
002: 既存コード
003: ...
004:     @Override
005:     public UserDetails loadUserByUsername(String username)
006:         throws UsernameNotFoundException {
007:         // 「認証テーブル」からデータを取得
008:         Authentication authentication = authenticationMapper.selectByUsername(username);
009:         // 対象データがあれば、UserDetailsの実装クラスを返す
010:         if (authentication != null) {
011:             // 対象データが存在する
012:             // UserDetailsの実装クラスを返す
013:             return new LoginUser(authentication.getUsername(),
014:                 authentication.getPassword(),
015:                 getAuthorityList(authentication.getAuthority()),
016:                 authentication.getDisplayname()
017:             );
018:         } else {
019:             // 対象データが存在しない
020:             throw new UsernameNotFoundException(
021:                 username + " => 指定しているユーザー名は存在しません");
022:         }
```

```
023:     }
024:     ...
025:     既存コード
026:     ...
```

13行目～17行目でUserDetailsの実装クラスである、LoginUserクラスのコンストラクタで、カスタム属性を引数に渡しています。

07 メニュー画面の修正

メニュー画面で「カスタム属性」を表示するように修正します。

修正内容

親フォルダを入力または選択	webapp/src/main/resources/templates
ファイル名	menu.html

ToDo一覧画面のlist.htmlの修正内容はリスト A.37 のようになります。

リスト A.37 menu.html

```
001: <!DOCTYPE html>
002: <html xmlns:th="http://www.thymeleaf.org"
003:       xmlns:sec="http://www.thymeleaf.org/thymeleaf-extras-springsecurity6">
004: <head>
005:   <title>メニュー</title>
006: </head>
007: <body>
008:   <h2>メニュー画面</h2>
009:   <hr>
010:   <a sec:authorize="hasAuthority('ADMIN')" th:href="@{/todos}">
011:     ToDo一覧へ
012:   </a>
013:   <br>
014:   <div sec:authorize="hasAuthority('USER')">
015:     【一般権限で表示される部分】
016:   </div>
017:   <div sec:authentication="name">
018:     ログイン情報のname値に書き換わる
019:   </div>
020:   <div th:text="${#authentication.principal.displayName}">
021:     新たに追加したdisplayName値に書き換わる
022:   </div>
```

```
023:   <div sec:authorize="isAuthenticated()">
024:     認証されたユーザーにのみ表示
025:   </div>
026:   <br>
027:   <!-- ログアウト -->
028:   <form th:action="@{/logout}" method="post">
029:     <input type="submit" value="ログアウト">
030:   </form>
031: </body>
032: </html>
```

20行目「<div th:text="\${#authentication.principal.displayName}">」でSpring Securityの認証情報からカスタム属性を取得しています。内容を分解して説明します。

\${}

Thymeleafで式を表すための記法です。この式を使用して、サーバーサイドの値をHTMLテンプレートに組み込むことができます。

#authentication

Spring Securityの認証情報にアクセスするためのオブジェクトです。このオブジェクトを通じて、現在認証されているユーザーの情報を取得することができます。

principal

認証プロセスで確認されたユーザーの主要な情報を保持するオブジェクトです。通常はUserDetailsの実装クラスのインスタンスを指します。

displayName

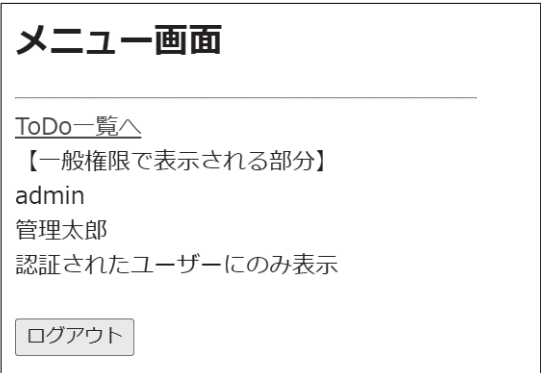
今回追加したカスタム属性です。標準のUserDetailsに含まれていない属性を保持したい場合は、カスタム属性を作成します。

08 ここまでの動作確認

アプリケーションを起動して、「カスタムログイン画面」にアクセスします。ここで「username」に「admin」、そして「password」に「adminpass」と入力し、「login」ボタンをクリックして「メニュー画面」を表示します。

カスタム属性で追加した「displayName」が表示されることを確認できます (図 A.22)。

図 A.22 カスタム属性の表示



を用いたセキュリティ強化された Web アプリケーション開発への入口にすぎませんが、強固なセキュリティ機能を持つアプリケーションを構築するための重要な第一歩になったと思います。

Spring Securityの学習は、複雑で難しく、挑戦的な側面もありますが、その努力は大きな価値をもたらします。深いセキュリティの理解は、アプリケーションを安全に保護し、ユーザーからの信頼を獲得するための鍵です。引き続き学習を進め、スキルセットを広げていくことをお勧めします。

A-5-3 最後に

開発が終わったので、アプリケーションを起動する度に実行される、DB へのテーブル削除・作成、データ登録処理を解除しましょう。

01 application.properties の修正

「src/main/resources」フォルダにある「application.properties」をリスト A.38 に修正します。

リスト A.38 application.properties

```
001: # DataSource
002: # Postgresのドライバーの設定
003: spring.datasource.driver-class-name=org.postgresql.Driver
004: # データベースへの接続URLを設定
005: spring.datasource.url=jdbc:postgresql://localhost:5432/springdb
006: # データベース接続のためのユーザー名を設定
007: spring.datasource.username=springuser
008: # データベース接続のためのパスワードを設定
009: spring.datasource.password=p@ss
010: # SQLスクリプトの初期化モードを設定
011: #spring.sql.init.mode=always ← コメントアウトする
012: # Log表示設定
013: logging.level.com.example.webapp.repository=DEBUG
```

11 行目「spring.sql.init.mode=always」は、Spring Boot アプリケーションが起動するたびに SQL スクリプトを実行するようにする設定です。コメントアウトすることで、設定は対象外になります。

「書籍：Spring Framework 超入門」と「Appendix Spring Security」を通じて、Spring Security の基本概念、認証、認可についての学習を進めてきました。これらの内容は、Spring Security