Wagby User Group

超高速開発コミュニティ Wagby分科会

Spring Security

OUTLINE

- Spring Security
- Spring Securityを使った認証の仕組み
- Spring Securityを使った独自認証
- 認証エラーメッセージの変更

Spring Security

Spring Securityとは

- アプリケーションのセキュリティを高めるためのフレームワーク
  - 認証、認可機能
  - その他、多数のセキュリティ関連の機能を持つ
  - 対応する認証機能
    - JDBC認証
    - LDAP認証
    - CAS認証
    - X509認証
    - Basic認証
    - etc

なぜSpring Security?

- メリット
  - Spring Framework標準の認証用プロダクト
  - 多彩な基本機能
    - JDBC認証、LDAP認証, OAuth2認証
    - 基本機能なので設定のみで対応可能。カスタマイズは不要。
  - 拡張性の向上
    - 多くのカスタマイズポイントが用意されている。

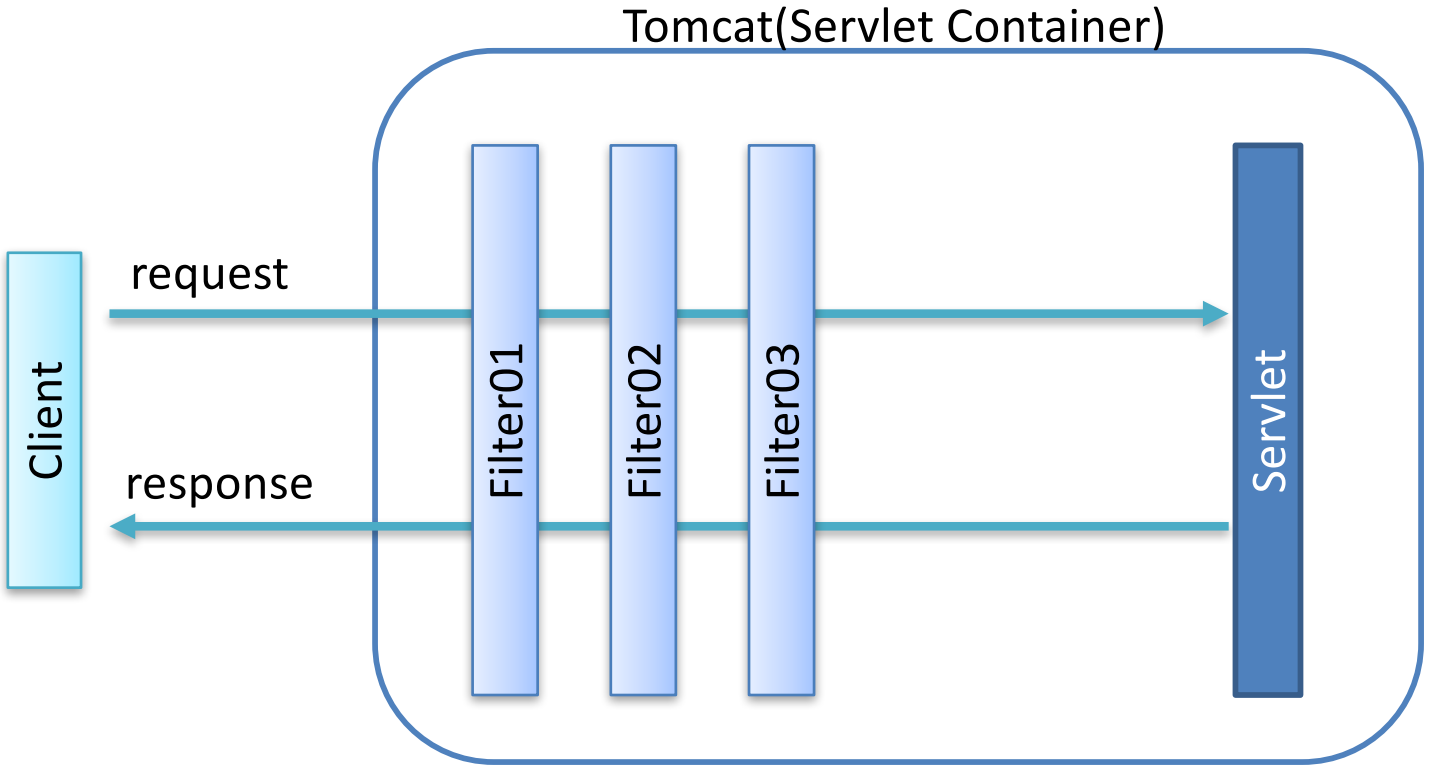
Spring Securityを使った認証の仕組み

セキュリティレイヤ

- セキュリティの向上 = セキュリティレイヤの導入
  - 各レイヤと独立してセキュリティ機能を付加する
    - ネットワーク : ファイアウォール、DMZ、侵入検知システム
    - OS : ファイアウォール
  - Spring Security = セキュリティレイヤ
    - Webアプリケーションにセキュリティレイヤを提供
    - Webアプリケーションの機能とは疎結合

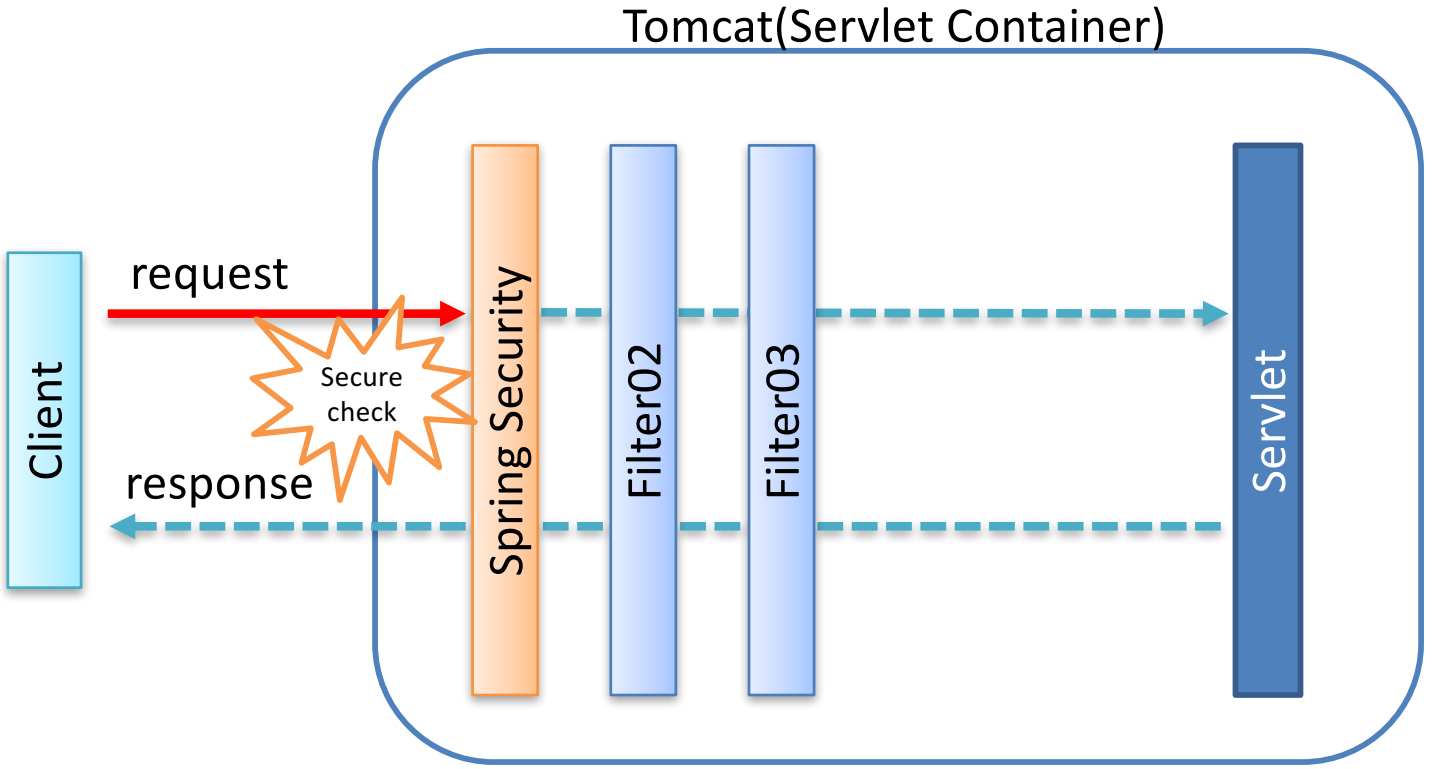
Servlet Filter

- Servlet Filterとは
  - クライアントからリクエストの前処理やサーバーからのレスポンスの後処理を追加できる機能



Spring Securityが提供するセキュリティレイヤ

- Spring Security = Servlet Filter
  - すべての処理に先立ってセキュリティチェックを行う
  - セキュリティ要件を満たさないリクエストはエラーとする



フィルタベースの実装

- フィルタベースの実装
  - Spring Securityを有効にすると自動的にフィルタが追加
  - フィルタで様々な機能を実現
  - 実際は次の順で処理が移譲されている
    1. DelegatingFilterProxy
    2. FilterChainProxy
    3. Spring Security用Filter(複数)

様々なフィルタ

- Spring Securityが提供しているフィルタ(一部)
  - SecurityContextPersistenceFilter
    - 認証情報を管理する SecurityContext の保持を行う
  - LogoutFilter
    - ログアウト処理を行う
  - **UsernamePasswordAuthenticationFilter**
    - 認証処理を行う
  - FilterSecurityInterceptor
    - 認証結果をもとにしたアクセス権のチェックを行う
- フィルタは設定により追加･除去が可能

UsernamePasswordAuthenticationFilter

- UsernamePasswordAuthenticationFilterでの認証
  - ユーザー名/パスワードでの認証処理を行う
  - 特定のURLにPOSTリクエストがくると動作する
    - wagby の場合は logon.do
  - 認証情報を表すAuthenticationインスタンスを作成

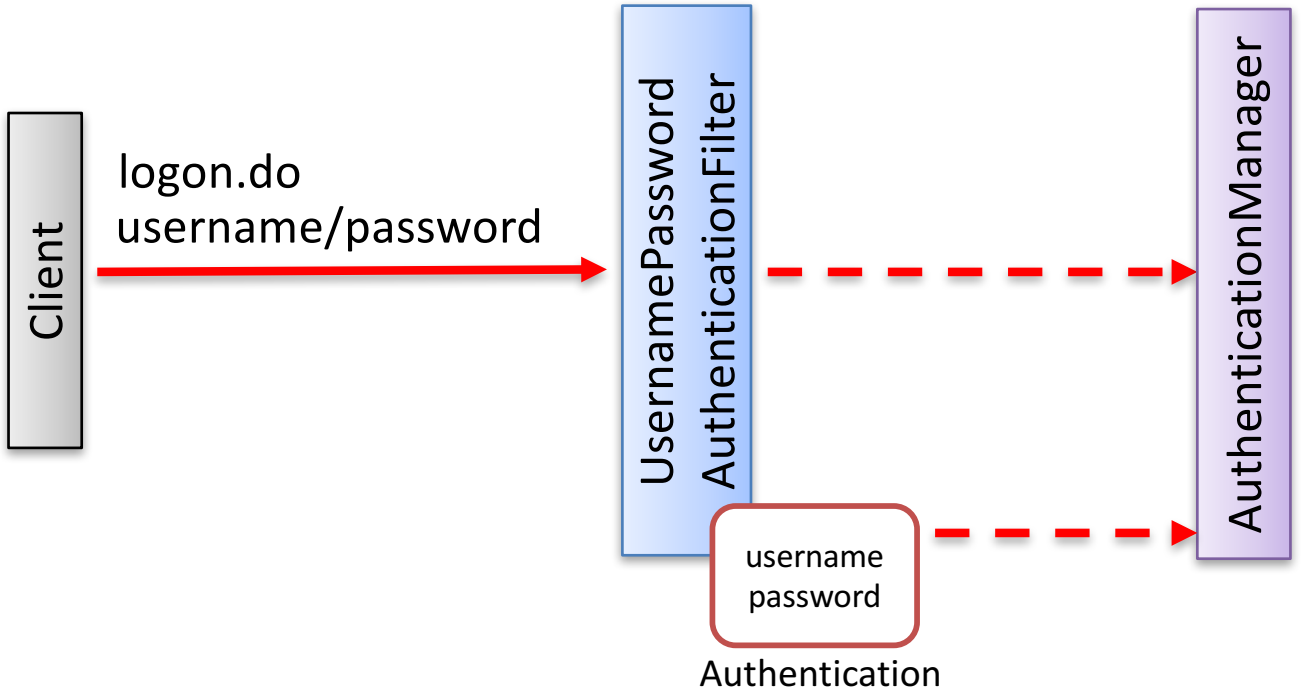
```
// 画面で入力されたusername,passwordを保持するAuthenticationの作成
Authentication authentication
    = new UsernamePasswordAuthenticationToken(username, password);
authentication.isAuthenticated(); // この時点では false
```

Authenticationクラス

- Authenticationクラスの役割
  - 送信されたユーザー名とパスワードを保持する
  - 認証状況(認証済/未認証)の情報を保持する
  - 認証後は認証ソース(LDAP や AD, JDBC テーブル)から取得したユーザー名/パスワード等も保持する
    - ただしパスワードは認証処理が終わると削除され、長期保持はされない
  - Authenticationのサブクラス
    - AnonymousAuthenticationToken, **UsernamePasswordAuthenticationToken**, RunAsUserToken
  - 認証処理はAuthenticationManagerへ移譲する

処理の流れ(AuthenticationFilter)

- UsernamePasswordAuthenticationFilter
  - 認証Token(Authenticationインスタンス)を作成

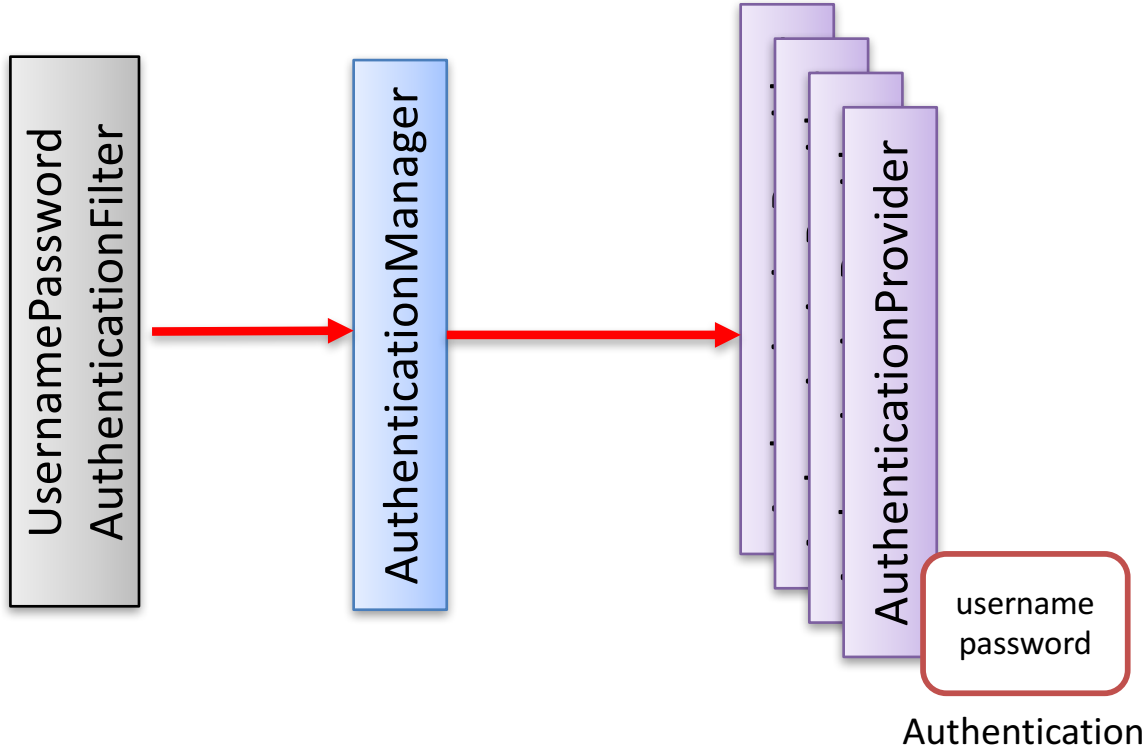


AuthenticationManager

- AuthenticationManager/AuthenticationProvider
  - AuthenticationManagerは複数のAuthenticationProviderを保持
  - 実際の認証処理はAuthenticationProviderへ更に移譲
  - いずれか一つのAuthenticationProviderで認証が成功すれば認証済みとなる
  - AuthenticationProviderの主なサブクラス
    - DaoAuthenticationProvider
    - LdapAuthenticationProvider
    - ActiveDirectoryLdapAuthenticationProvider

処理の流れ(AuthenticationManager)

- AuthenticationManager
  - AuthenticationManagerはAuthenticationProviderへ処理を委譲



AuthenticationProvider

- AuthenticationProvider
  - 認証処理を実行するクラス
  - 定義されているメソッドは2つ
    - authenticate()メソッド : 認証処理を実装するメソッド
    - supports()メソッド : この認証プロバイダがサポートするAuthenticationクラスの指定。
      - 通常はUsernamePasswordAuthenticationToken

```
@Override
public boolean supports(Class<?> authentication) {
    // POST で送信されたユーザー名とパスワードで認証を行う。
    return UsernamePasswordAuthenticationToken.class
            .isAssignableFrom(authentication);
}
```

authenticate() メソッド

- 認証処理を行うメソッド
  - 認証エラーの場合はAuthenticationExceptionをthrow
  - 認証成功時の処理
    - 認証ソース(LDAP や AD, JDBC テーブル)から取得したユーザー名とパスワードからUserDetailsインスタンスを作成
    - 認証情報を表すAuthenticationインスタンスにUserDetailsをセットする
    - AuthenticationにUserDetailsがセットされていれば認証が成功したものと判断する

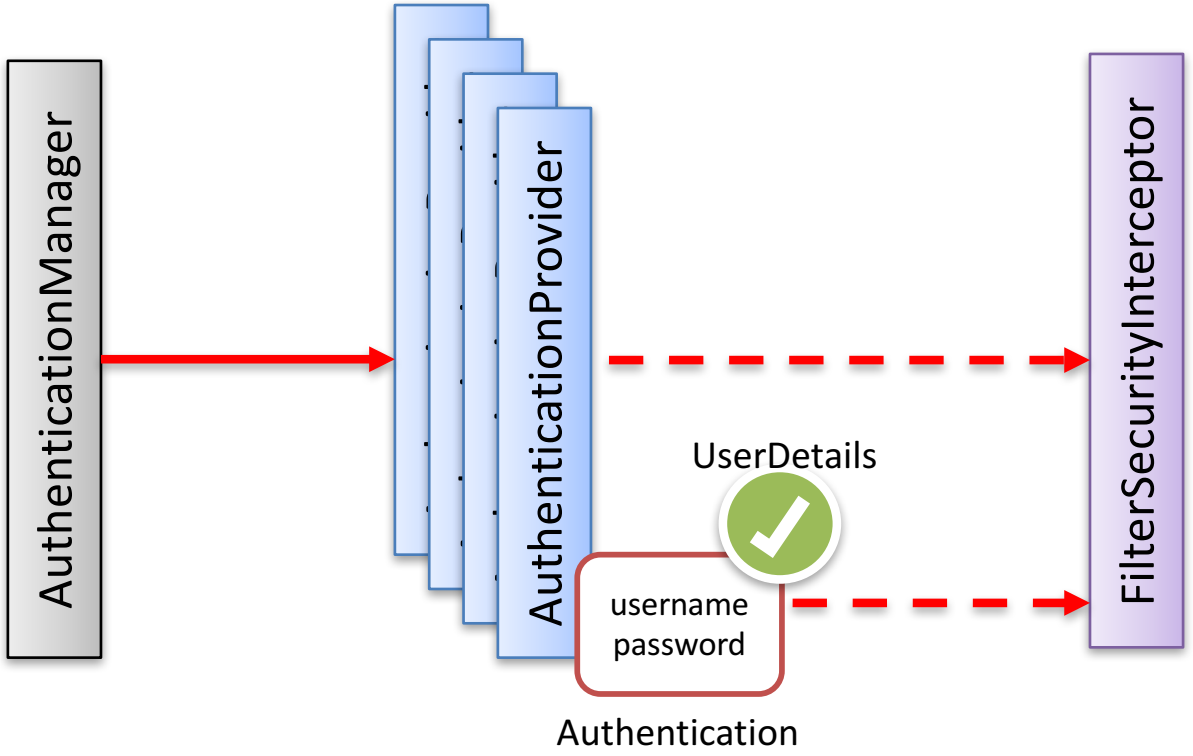
認証成功時の実装

```
// username, password は認証ソースから取得したもの。
// 権限は ROLE_USER 固定(Wagbyでは利用されない)。
UserDetails user = new User(username, password,
      AuthorityUtils.createAuthorityList("ROLE_USER"));

// 認証情報に UserDetails オブジェクトを格納。
UsernamePasswordAuthenticationToken authenticationResult
     = new UsernamePasswordAuthenticationToken(user,
          authentication.getCredentials(), user.getAuthorities());
authenticationResult.setDetails(authentication.getDetails());
authentication.isAuthenticated(); // この時点ではtrue
```

処理の流れ(AuthenticationProvider)

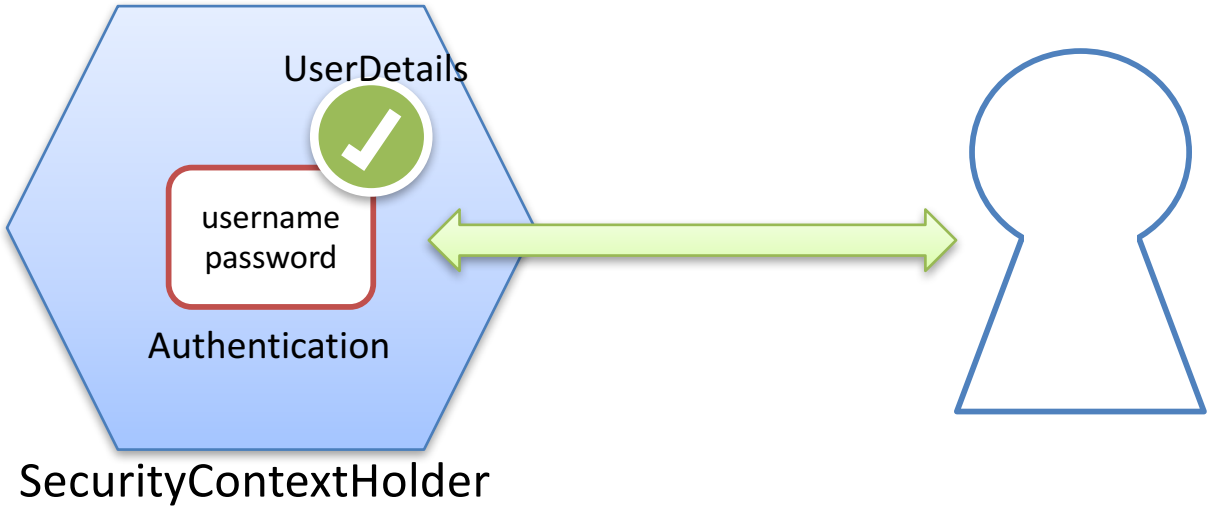
- AuthenticationProvider
  - 認証処理に成功すると認証TokenにUserDetailsオブジェクトがセットされる



認証後の認証情報の取得

- 認証情報は SecurityContextHolderが保持
  - Spring Security処理後は認証情報は SecurityContextHolderを介して取得する

```
Authentication authentication
        = SecurityContextHolder.getContext().getAuthentication();
authentication.isAuthenticated(); // 認証状況を確認できる
```



各クラスの役割

クラス	役割
UsernamePasswordAuthenticationFilter	認証処理の入口となるクラス。 Authenticationを作成する
Authentication (UsernamePasswordAuthenticationToken)	認証情報を保持するクラス (認証済/未認証)
AuthenticationManager	AuthenticationProviderに実際の認証処理を委譲するクラス
AuthenticationProvider	認証処理を実行するクラス
UserDetails	認証成功を意味するクラス。認証ソースから取得したユーザ情報を保持する。

処理の流れ(全体)

1. ログオン画面でユーザー名とパスワードを入力し、ログオン。
2. ブラウザからlogon.doにPOSTリクエストを送信
3. UsernamePasswordAuthenticationFilterでユーザー名とパスワードを保持したUsernamePasswordAuthenticationTokenを作成(この時点では未認証)
4. 認証処理はAuthenticationManagerへ移譲される
5. AuthenticationManagerは更に複数のAuthenticationProviderへ処理を委譲
6. 複数のAuthenticationProviderのうちUsernamePasswordAuthenticationTokenの認証をサポートするクラスのみが認証処理を行う

処理の流れ(2)

7. JDBC認証用AuthenticationProviderであればデータベースからユーザー名とパスワードを取得し、ログオン画面で入力されていたものと一致していれば認証成功とする
8. 認証成功の場合はUserDetailsオブジェクトを作成し、Authentication(認証情報)に格納する
9. 認証失敗の場合はAuthenticationExceptionをthrowする

Spring Securityを使った独自認証

独自認証処理を行う

1. ログオン画面でユーザー名とパスワードを入力し…

…

~~7. JDBC認証用AuthenticationProviderであればデータベースからユーザー名とパスワードを取得し、ログオン画面で入力されていたものと一致していれば認証成功とする~~

7. カスタマイズしたAuthenticatonProviderで独自認証を行う

8. 認証成功の場合はUserDetailsオブジェクトを作成し、Authentication(認証情報)に格納する

9. 認証失敗の場合はAuthenticationExceptionをthrowする

AuthenticationProvierをひとつ作成するだけで実現可能

独自認証用AuthenticationProvider

```
public class SampleAuthenticationProvider
        implements AuthenticationProvider {

    /** {@inheritDoc} **/
    @Override
    public Authentication authenticate(Authentication authentication)
            throws AuthenticationException {
        // ここに独自認証ロジックを記述する。
    }

    /** {@inheritDoc} **/
    @Override
    public boolean supports(Class<?> authentication) {
        // POST で送信されたユーザー名とパスワードで認証を行う。
        return UsernamePasswordAuthenticationToken.class
                .isAssignableFrom(authentication);
    }
}
```

独自認証処理(例)

- 画面で入力されたユーザー名パスワードを取得
  - authenticationから取得できる

```
public Authentication authenticate(Authentication authentication)
        throws AuthenticationException {

    authentication.isAuthenticated(); // この時点では false;

    // 入力されたユーザー名とパスワードを取得。
    String username = authentication.getName();
    String password = authentication.getCredentials().toString();
```

独自認証処理(例)

- 独自認証ロジック
  - ユーザー名、パスワードのいずれかが未入力の場合は認証エラー(AuthenticationCredentialsNotFoundException)
  - パスワードが「secret」であれば認証成功。その他の文字は認証エラー(UsernameNotFoundException)

```
if (StringUtils.isBlank(username)
        || StringUtils.isBlank(password)) {
    // ユーザー名/パスワードのいずれかが未入力の場合
    throw new AuthenticationCredentialsNotFoundException(
            "User is not Authenticated");
}
if (!"secret".equals(password.trim())) {
    // パスワードが"secret"ではない場合。
    throw new UsernameNotFoundException(
            "Username " + username + " not found");
}
```

独自認証処理(例)

- 認証成功時の処理
  - 認証済であることを示すUserDetails オブジェクトを作成
    - ユーザー名、パスワードは認証ソースから取得したものをセット
    - 権限はROLE_USER固定(Wagbyでは利用されない)
  - 認証情報(Authentication)にUserDetailsをセットする
    - 厳密には元のAuthenticationをベースに再作成

```
UserDetails user = new User(username.trim(), password.trim(),
        AuthorityUtils.createAuthorityList("ROLE_USER"));
 // 認証情報に UserDetails オブジェクトを格納。
UsernamePasswordAuthenticationToken authenticationResult
        = new UsernamePasswordAuthenticationToken(user,
                authentication.getCredentials(),
                user.getAuthorities());
authenticationResult.setDetails(authentication.getDetails());
authenticationResult.isAuthenticated(); // この時点では true
 return authenticationResult;
```

独自認証用AuthenticationProviderを設定する

- SecurityConfiguration
  - Wagby標準のSpring Securityの設定用クラス
  - Spring Securityが提供しているWebSecurityConfigurerAdapterを継承
  - JDBC認証(juserテーブル),LDAP認証ActiveDirectory認証に対応
  - application.properties に設定を行うだけで利用可能

独自認証用AuthenticationProviderを設定する(2)

- SecurityConfigurationの拡張クラスを作成
  - パッケージ名:jp.jasminesoft.wagby.autoconfiguration
    - リポジトリで定義したパッケージ名 + .autoconfiguration
  - jp.jasminesoft.jfc.autoconfiguration.SecurityConfigurationを継承する
  - @Configurationアノテーションを付与する
  - クラス名:任意
  - 上記の条件を満たしたSecurityConfigurationの拡張クラスを用意すると自動的にWagby標準クラスは無効化される
  - configureGlobal (AuthenticationManagerBuilder)メソッドをオーバーライドし独自認証用のAuthenticationProviderを追加

独自認証用AuthenticationProviderを設定する(3)

```
package jp.jasminesoft.wagby.autoconfiguration;

import ...

@Configuration
public class MySecurityConfiguration extends SecurityConfiguration {

    /** {@inheritDoc} **/
    @Override
    protected void configureGlobal(AuthenticationManagerBuilder auth)
            throws Exception {
        auth.authenticationProvider(new SampleAuthenticationProvider());
        super.configureGlobal(auth); // Wagby のJDBC認証を併用しない場合は不要。
    }
}
```

認証エラーメッセージの変更

認証エラーメッセージを変更する

- 認証時のエラーの種類はAuthenticationExceptionのサブクラスで表現
  - AuthenticationExceptionのサブクラス(一部)
    - AccountExpiredException
    - LockedException
    - UsernameNotFoundException
    - BadCredentialsException
  - Spring Securityでは認証エラー時の処理はAuthenticationFailureHandlerで実装する
  - AuthenticationFailureHandlerでAuthenticationExceptionを取得し、種類に応じたエラーメッセージをセットする

AuthenticationFailureHandlerの拡張クラス

- MultipleSessionAuthenticationFailureHandler
  - マルチセッション用にAuthenticationFailureHandlerを拡張したクラス(Wagby側で用意したクラス)
  - 独自のエラーメッセージを表示するにはMultipleSessionAuthenticationFailureHandlerのsaveErrorMessage()メソッドを拡張する
  - 作成した拡張クラスをSpringのBeanとして登録する

拡張AuthenticationFailureHandler

```
package jp.jasminesoft.wagby;

import ...

/**
 * カスタマイズ用認証失敗時のハンドラです。
 */
public class SampleAuthenticationFailureHandler
        extends MultipleSessionAuthenticationFailureHandler {
    /**
     * コンストラクタ。
     * @param defaultFailureUrl 認証失敗時の遷移先
     */
    public SampleAuthenticationFailureHandler(String defaultFailureUrl) {
        super(defaultFailureUrl);
    }
    /** {@inheritDoc} */
    @Override
    public void saveErrorMessage(HttpServletRequest request,
            HttpServletResponse response, AuthenticationException exception) {
        // ここでエラーメッセージをセットする。
    }
}
```

saveErrorMessage()メソッド実装例

```
/** {@inheritDoc} */
@Override
public void saveErrorMessage(HttpServletRequest request,
        HttpServletResponse response, AuthenticationException exception) {
    super.saveErrorMessage(request, response, exception);

    Jfcerror jfcerror = new Jfcerror();
    if (exception instanceof LockedException) {
        jfcerror.setContent("アカウントがロックされています。");
    } else if (exception instanceof AccountExpiredException) {
        jfcerror.setContent("アカウントの有効期限が切れています。");
    } else {
        // 何もせずデフォルトのエラーメッセージを表示させる。
        return;
    }
     // エラーメッセージを格納している JfcErrors を取得。
    Jfcerrors errors = (Jfcerrors) request.getSession(true).getAttribute(
            BaseController.JfcerrorsRequestName);
    // すでにセットされているエラーメッセージを表示させない場合はerrors をクリアする。
    errors.clearJfcerror();
    // 作成したエラーメッセージを追加する。
    errors.addJfcerror(jfcerror);
}
```

AuthenticationFailureHandler拡張クラスの登録

- 次のコードをSecurityConfigurationの拡張クラスに実装することで登録できる
  - 以下のメソッドをMySecurityConfigurationに追加
  - これにより従来Wagbyが利用していたMultipleSessionAuthenticationFailureHandlerが無効化されSampleAuthenticationFailureHandlerが利用される

```
/**
 * AuthenticationFailureHandler の bean 定義。
 * @return AuthenticationFailureHandler
 */
@Bean
protected AuthenticationFailureHandler authenticationFailureHandler() {
    return new SampleAuthenticationFailureHandler(
            securityProperties.getLogonPage() + "?error");
}
```

まとめ

- 独自認証処理を行う
  1. 独自認証用AuthenticationProviderを作成
     - authenticate() メソッドに認証処理を実装
  2. SecurityConfigurationの拡張クラスを作成
     - 独自認証用AuthenticationProviderを設定
- エラーメッセージの変更
  1. MultipleSessionAuthenticationFailureHandlerのサブクラスを作成
     - saveErrorMessage()メソッドにてエラーメッセージを登録
  2. SecurityConfigurationの拡張クラスを作成(すでに存在する場合は不要)
     - MultipleSessionAuthenticationFailureHandlerのサブクラスをBean登録