[Java開発者向け] シングルサインオンへの対応
- Javaカスタマイズコードの書き方

OUTLINE

- Spring Security
- Spring Securityを使った認証の仕組み
- Spring Securityを使ったシングル・サインオン

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。

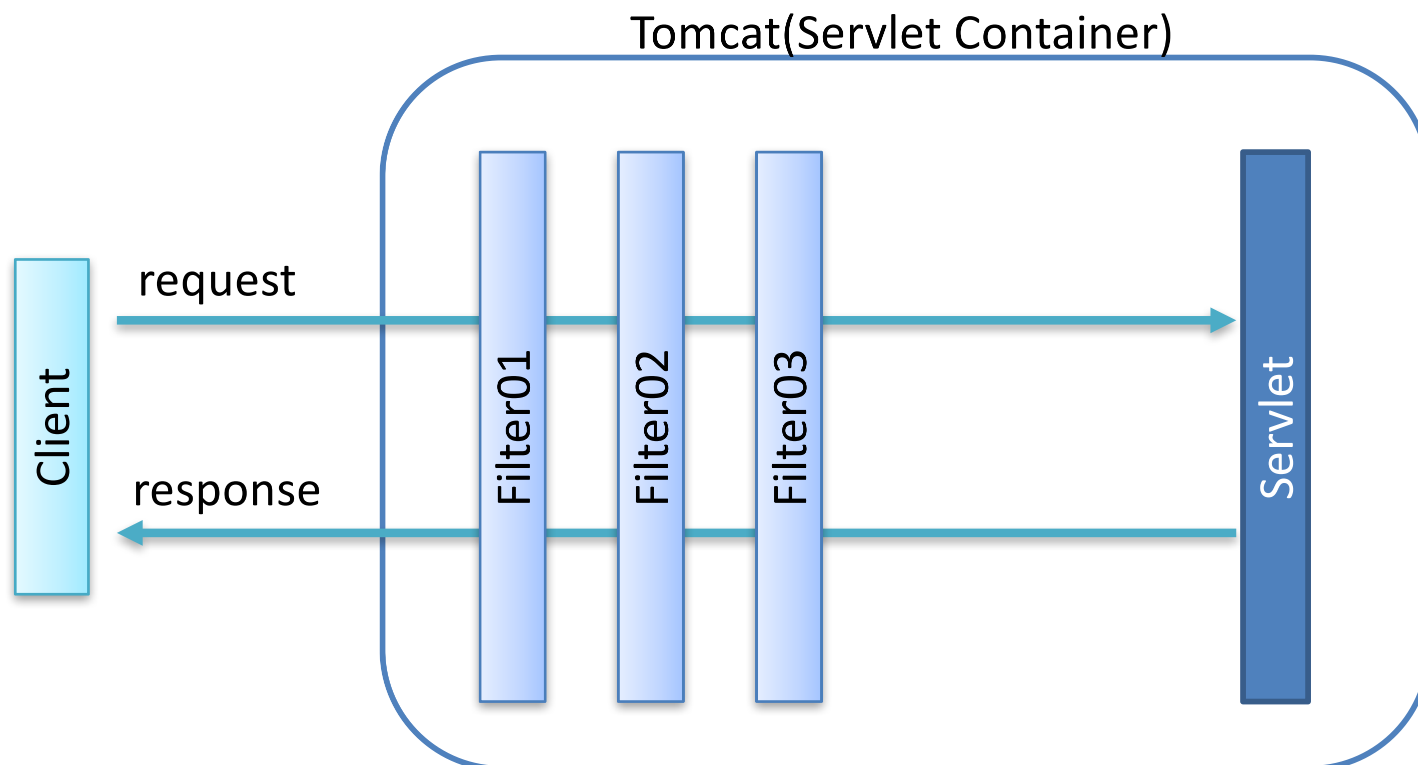
Spring Securityを使った認証の仕組み

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

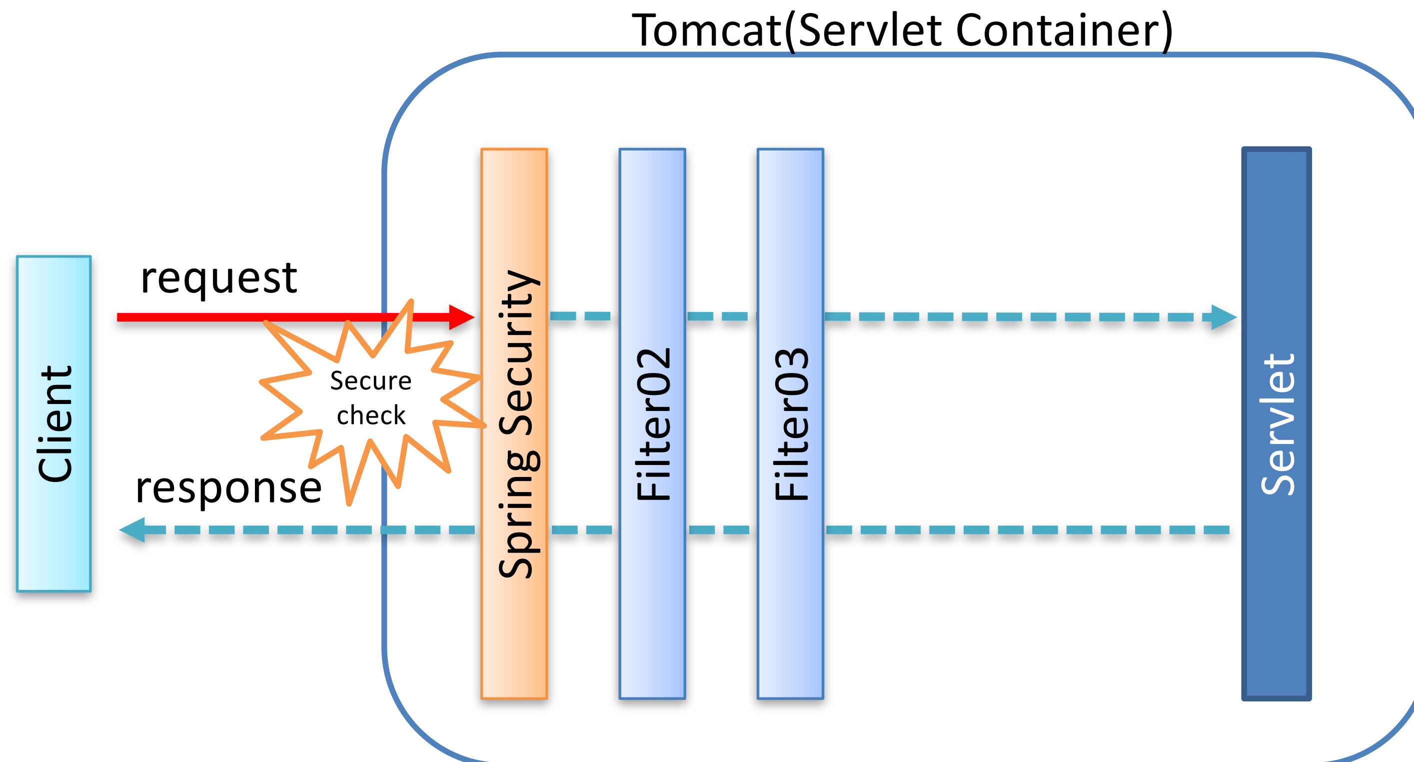
Servlet Filter

- 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

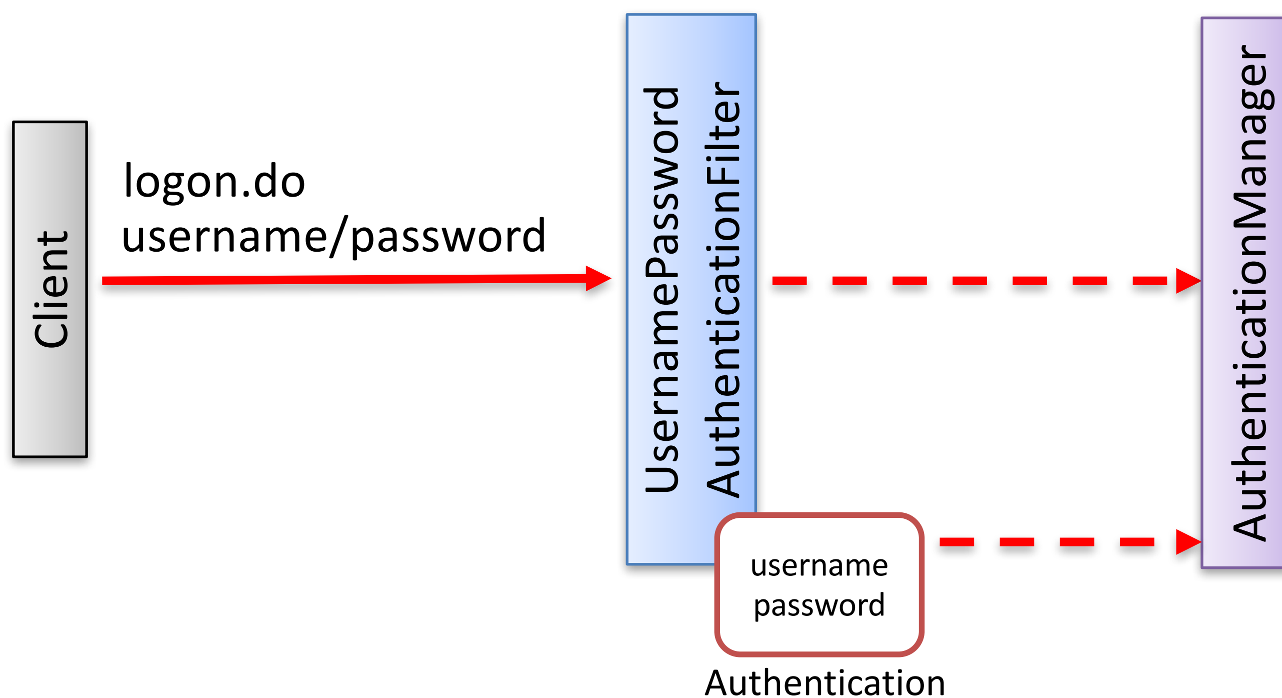
```
// 画面で入力されたusername,passwordを保持するAuthenticationの作成
Authentication authentication
    = new UsernamePasswordAuthenticationToken(username, password);
authentication.isAuthenticated(); // この時点では false
```

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

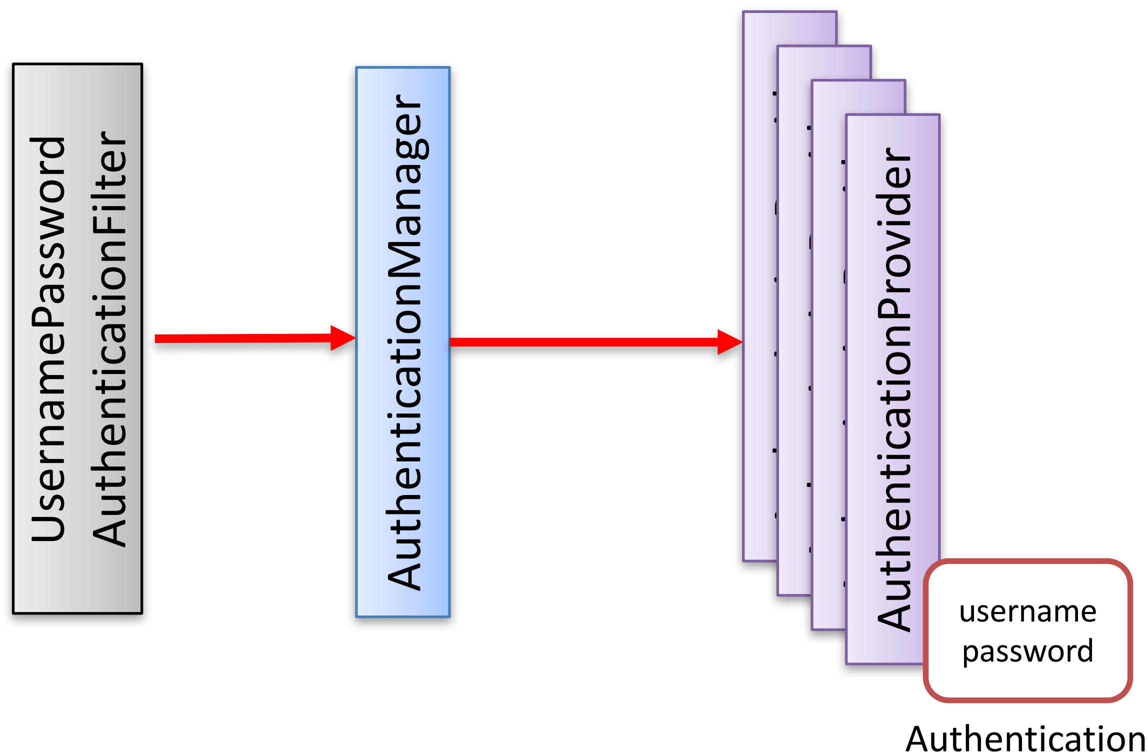


AuthenticationManager

- 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

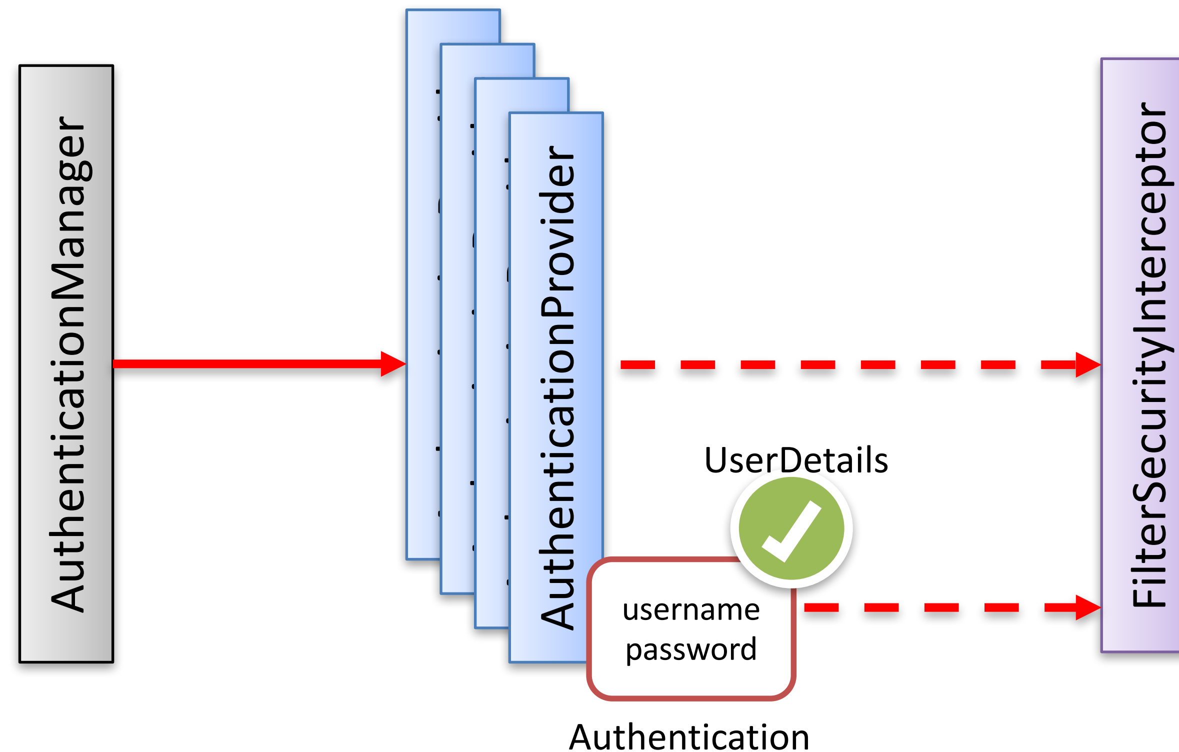
認証成功時の実装

```
// 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());
authenticationResult.isAuthenticated(); // この時点ではtrue
```

処理の流れ(AuthenticationProvider)

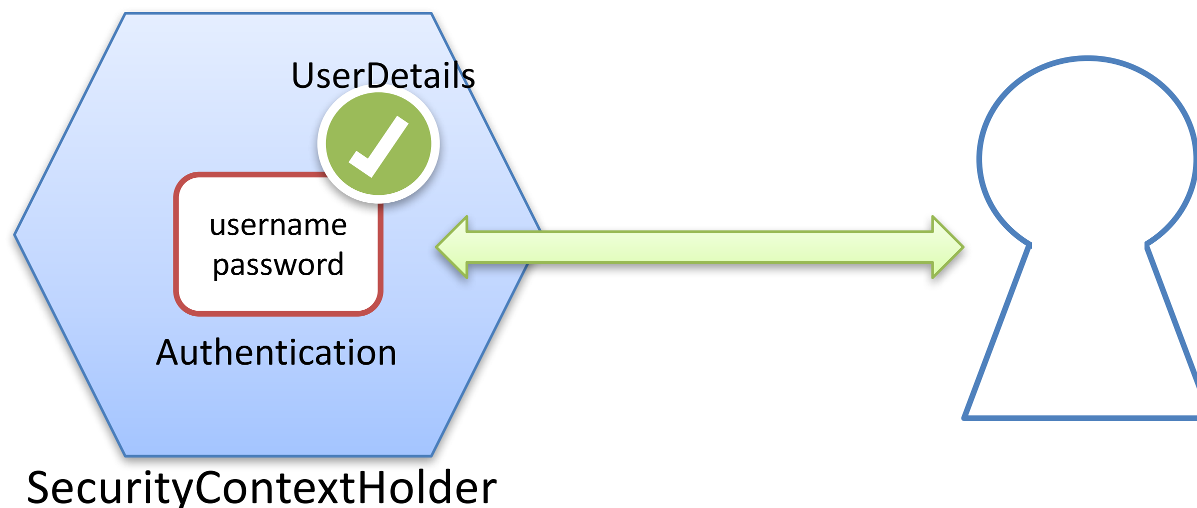
- AuthenticationProvider
  - 認証処理に成功すると認証TokenにUserDetailsオブジェクトがセットされる



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)

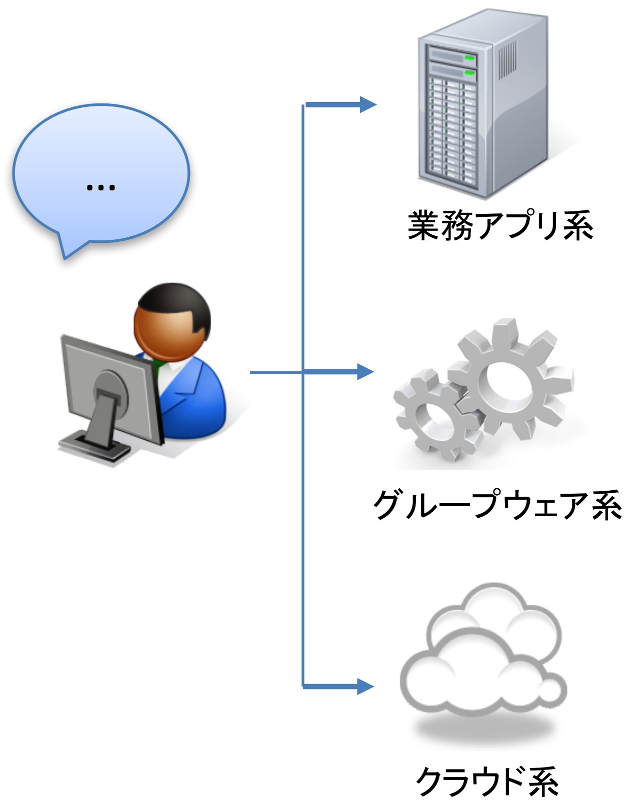
7. JDBC認証用AuthenticationProviderであればデータベースからユーザー名とパスワードを取得し、ログオン画面で入力されていたものと一致していれば認証成功とする
8. 認証成功の場合はUserDetailsオブジェクトを作成し、Authentication(認証情報)に格納する
9. 認証失敗の場合はAuthenticationExceptionをthrowする

Spring Securityを使ったシングル・サインオン

シングル・サインオン

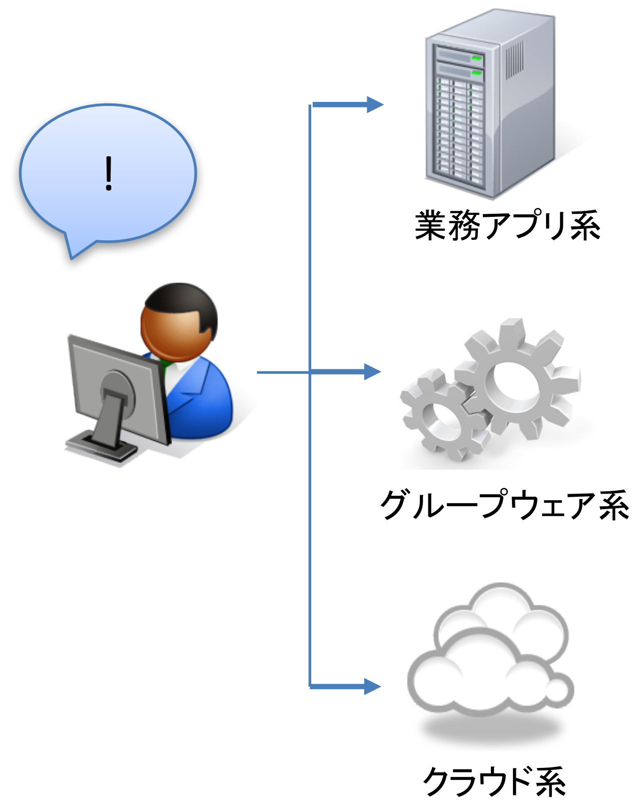
シングルサインオンなし

各アプリケーションに毎回ログオンする。



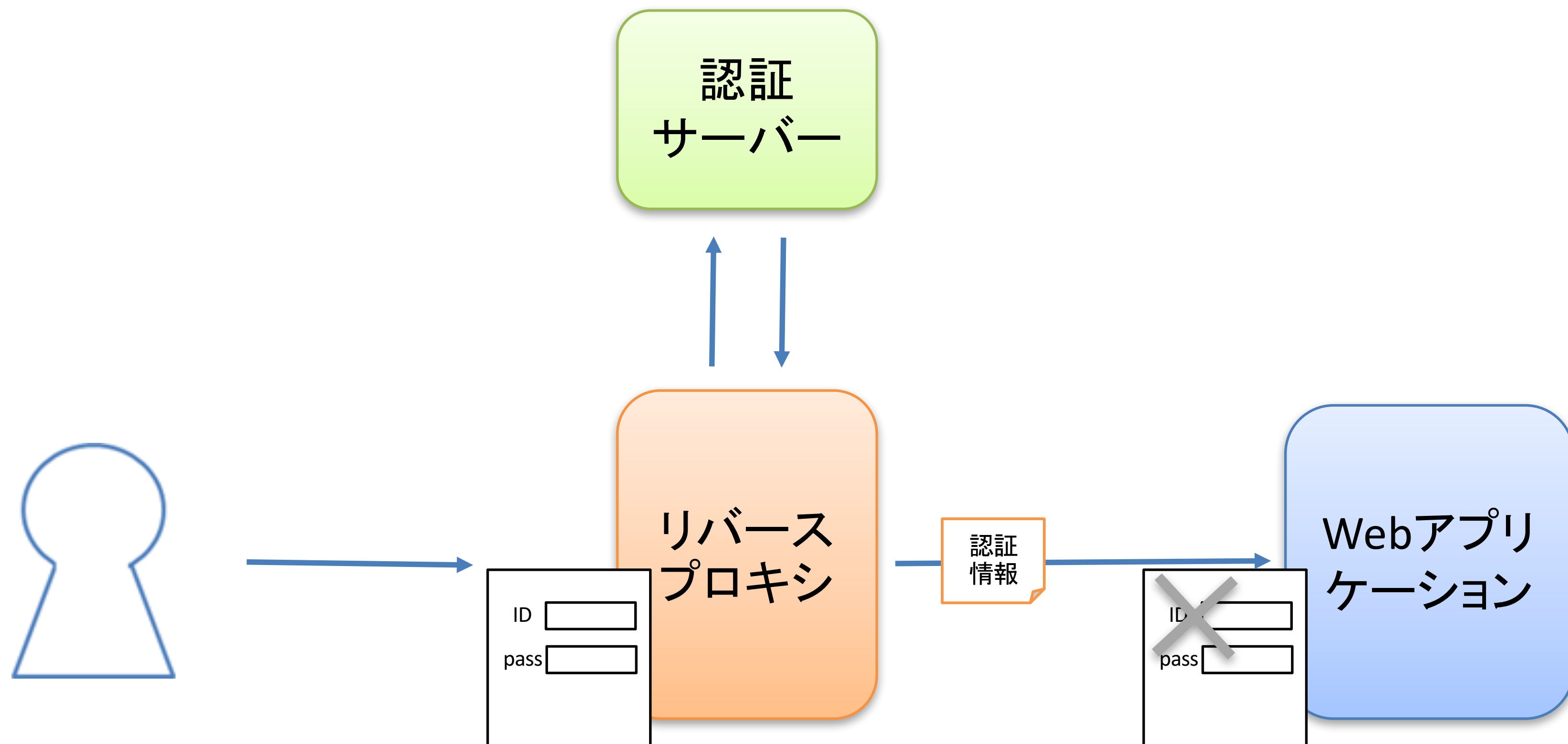
シングルサインオンあり

一回のログオンですべてのアプリを利用



シングル・サインオン[2]

- 一般的なシングル・サインオンの流れ
  - リバースプロキシ型の例



シングル・サインオン[3]

- 一般的なシングル・サインオンの流れ
  - リバースプロキシ型の例
    1. Webアプリケーションにアクセスする
    2. アクセス経路にあるリバースプロキシでログオン画面が表示される
    3. ユーザーID/パスワードを入力しログオン
    4. 入力されたユーザーID/パスワードを認証サーバーへ問い合わせ、正しければWebアプリケーションへアクセスが可能となる
    5. Webアプリケーションにアクセスする際はリバースプロキシがリクエストに認証情報を埋め込む
    6. Webアプリケーションはリクエストに埋め込まれた認証情報を受け取り、認証処理をスキップさせ、メニュー画面を表示する
    7. 以降のアクセスはリバースプロキシ、Webアプリケーションともに認証済み状態としてアクセスされる

シングル・サインオン連携

- シングル・サインオン連携とは
  - リクエストに埋め込まれた認証情報を受け取りログオン処理をスキップさせる機能
    - 埋め込まれた認証情報があれば認証済みと判断
  - 連携機能がないアプリケーションではログオン画面が出てしまう

アカウント情報

- アカウントの作成は必須
  - Wagbyが連携するのはユーザーIDのみ
  - 所属グループ、権限などの情報が不足した状態ではログオンできない
  - パスワードは使わない

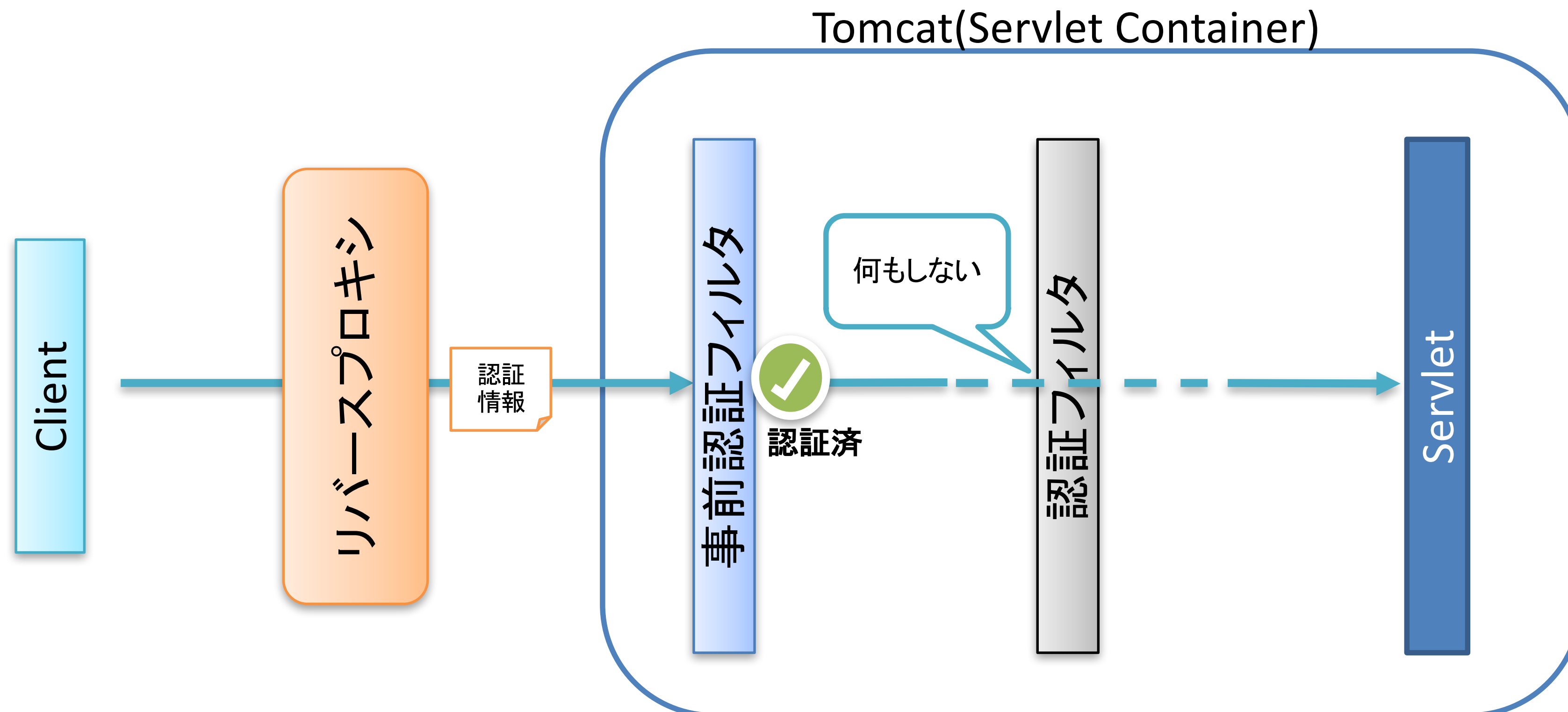
※ LDAP/Active Directory連携とルールは同じ

事前認証(PreAuthentication)

- Spring Securityの事前認証機能
  - 通常の認証処理の前に実施される
  - 事前認証で認証されていれば通常の認証処理では何もしない
    - すでに認証済みとして扱われる
  - 事前認証の仕組みを利用してシングル・サインオンを実現する

事前認証(PreAuthentication)[2]

- Spring Securityの事前認証機能
  – シングル・サインオンサーバーで認証済みであれば事前認証処理でこれを検知して認証処理を行う



Spring Securityが提供するクラス

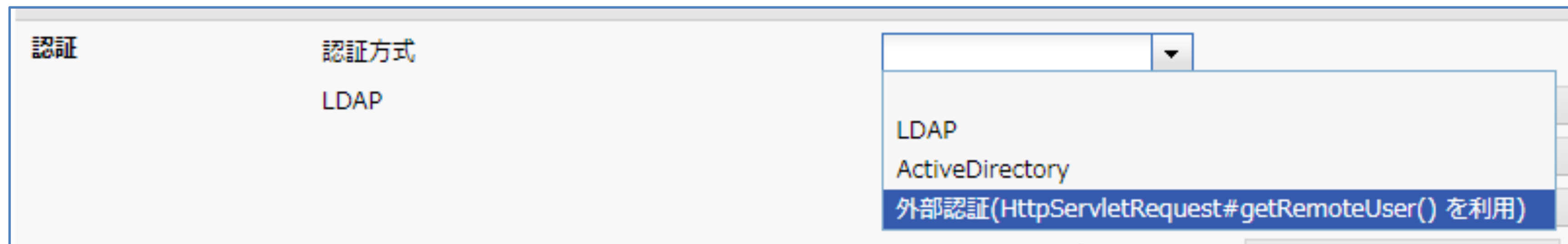
- 事前認証に利用するクラスが提供されている
  - 認証フィルタ
    - RequestHeaderAuthenticationFilter
      - リクエストヘッダに埋め込まれた認証情報を取り出す
    - RequestAttributeAuthenticationFilter
      - リクエスト属性に埋め込まれた認証情報を取り出す
  - 認証プロバイダ
    - PreAuthenticatedAuthenticationProvider
  - Authenticationクラス
    - PreAuthenticatedAuthenticationToken

認証情報の取得

- 認証情報の受け渡し方法はシングル･サインオン製品によって様々
  - request.getRemoteUser()でユーザーIDを取得
    - サーブレットの仕様
  - リクエスト属性"REMOTE_USER"でユーザーIDを取得
    - String userid = (String) request.getAttribute("REMOTE_USER");
  - リクエストヘッダ" SM_USER"でユーザーID取得
    - String userid = request.getHeader("SM_USER");
    - リクエストヘッダは簡単に偽装可能なため、リバースプロキシなどで偽装ができないよう配慮する必要がある

Wagbyのシングル・サインオン連携

- request.getRemoteUser()を利用
  - WagbyDesigner > 環境 > サーバ > 認証 > 認証方式
    - 外部認証(HttpServletRequest#getRemoteUser()を使用)



  - 他の取得方式の場合はカスタマイズが必要

カスタマイズクラスの作成

- 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
  - preAuthenticationConfigure(HttpSecurity http)メソッドをオーバーライドしカスタマイズコードを記述する

リクエストヘッダを使った連携

- カスタマイズコード
  - 例) リクエストヘッダ"UID"からユーザーIDを取得

```
@Configuration
public class MySecurityConfiguration extends SecurityConfiguration {

    /** {@inheritDoc} **/
    @Override
    public void preAuthenticationConfigure(HttpSecurity http) throws Exception {
        if (!securityProperties.isValidPreAuthentication()) {
            return;
        }
        // 認証サーバで認証済みのユーザーIDをHTTPヘッダから取得する。
        RequestHeaderAuthenticationFilter filter
                = new RequestHeaderAuthenticationFilter();
        filter.setExceptionIfHeaderMissing(false);
        // ヘッダUIDにユーザーIDがセットされている。
        filter.setPrincipalRequestHeader("UID");

        http.apply(new PreAuthenticationConfigurer()
                .preAuthenticationFilter(filter));
    }
}
```

リクエスト属性を使った連携

- カスタマイズコード
  - 例) リクエスト属性"UID"からユーザーIDを取得

```
@Configuration
public class MySecurityConfiguration extends SecurityConfiguration {

    /** {@inheritDoc} **/
    @Override
    public void preAuthenticationConfigure(HttpSecurity http) throws Exception {
        if (!securityProperties.isValidPreAuthentication()) {
            return;
        }
        // 認証サーバで認証済みのユーザーIDをリクエスト属性から取得する。
        RequestAttributeAuthenticationFilter filter
                = new RequestAttributeAuthenticationFilter();
        filter.setExceptionIfVariableMissing(false);
        // リクエスト属性UIDにユーザーIDがセットされている。
        filter.setPrincipalEnvironmentVariable("UID");

        http.apply(new PreAuthenticationConfigurer()
                .preAuthenticationFilter(filter));
    }
}
```

認証情報の信頼性チェック

- 認証情報のチェック
  - 認証情報を詐称できる場合がある
    - 詐称できないネットワーク構成とすることも可能
      - リバースプロキシを通さないとアプリケーションにアクセスできない
      - リバースプロキシを経由する際にリクエストヘッダXXXを上書きする
  - シングル・サインオン製品によっては認証情報と共に信頼性チェックのための付加情報を送付するものもある
    - チェックの実装方法は製品に依存

認証情報の信頼性チェック

```
@Configuration
public class MySecurityConfiguration extends SecurityConfiguration {

    /** {@inheritDoc} **/
    @Override
    public void preAuthenticationConfigure(HttpSecurity http) throws Exception {
        if (!securityProperties.isValidPreAuthentication()) {
            return;
        }
        AbstractPreAuthenticatedProcessingFilter filter
                = new AbstractPreAuthenticatedProcessingFilter() {
                    /** {@inheritDoc} **/
                    @Override
                    protected Object getPreAuthenticatedPrincipal(
                            HttpServletRequest request) {
                        if (!check(request)) {
                            // 信頼性チェックを満たさない場合は null を返す。
                            return null;
                        }
                        // リクエスト属性/ヘッダからユーザーIDを取得する。
                        return request.getAttribute("XXX");
                        //return request.getHeader("XXX");
                    }
```

認証情報の信頼性チェック

```java
                    /** 信頼性チェック */
                    private boolean check(HttpServletRequest request) {
                        // TODO 信頼性チェックを実装。問題無い場合は true を返す。
                        return false;
                    }

                    /** {@inheritDoc} **/
                    @Override
                    protected Object getPreAuthenticatedCredentials(
                            HttpServletRequest request) {
                        return "N/A";
                    }
                };

        http.apply(new PreAuthenticationConfigurer()
                .preAuthenticationFilter(filter));
    }
}
```

カスタマイズによるフィルタの追加

- エージェント用フィルタの追加
  - シングル・サインオン製品によってはWagbyアプリケーションにフィルタの追加が必要
  - フィルタの適用順には注意が必要
    - エージェント用であれば -299 から -200 の間の数値を指定

```
@Configuration
public class MySecurityConfiguration extends SecurityConfiguration {
    ...
    /** シングル・サインオン連携のためのフィルタを追加します。 */
    @Bean
    public FilterRegistrationBean httpServletRequestWrapperFilter() {
        FilterRegistrationBean filterBean = new FilterRegistrationBean(
                new XXXFilter()); // フィルタの指定
        filterBean.addUrlPatterns("/*"); // フィルタを適用する URL
        filterBean.setOrder(-299); // フィルタの適用順
        return filterBean;
    }
}
```

DB認証との併用

- 以下のようなケースで併用
  - シングル・サインオン側にadminアカウントを作成できない
  - システム管理者権限を付与する適切なユーザーがシングル・サインオン側に存在しない
- DB認証との併用
  - シングル・サインオン認証なしにアクセスされた場合はログオン画面を表示させ、juserテーブルを使った認証を行う
  - セキュリティ要件、動作要件が難しいので要相談。
    - リクエストヘッダや属性を詐称されることはないか
    - 通常のユーザーにログオン画面が見えてしまってもよいか

その他の認証連携

- OpenID Connect, OAuth2
  - Spring Security 5で正式対応(Wagbyは現在4.2を利用)
  - OpenID Connect対応時にAzure AD認証連携も対応予定

まとめ

- Spring Securityを使ったシングル･サインオン
  - Spring SecurityのPreAuthenticationの仕組み
  - PreAuthenticationを使ったシングル･サインオン連携
  - 認証情報の受け渡し
  - 様々な認証情報の取得方法(カスタマイズ)
  - 認証情報の信頼性チェック
  - 認証連携用フィルタの追加方法
- 今後の拡張
  - OpenID Connect, OAuth2への対応