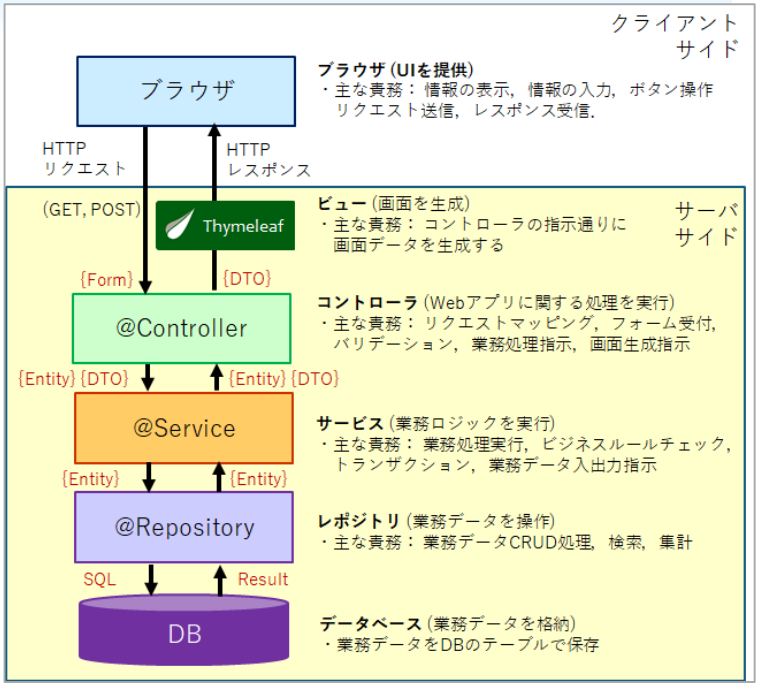
ITスペシャリスト養成コース

第３回

Webアプリケーション開発実践

前回身に付けた知識

- Webアプリケーションとは
  - 原理，URI, HTTP-メソッド, ROA, Web-MVC, RESTful, Spring Boot
  - POST, GET, PUT, DELETE
  - RESTful
- Spring Boot基礎
  - レイヤの責務
    - ビュー (Thymeleaf)
    - コントローラ (@Controller)
    - サービス (@Service)
    - レポジトリ(@Repository)
    - データ (Entity, DTO, Form)
  - 猫アプリ
  - ToDo管理アプリ



今日の目的

- Spring Bootで**実用に耐える**Webアプリを作成
  - 例外処理
  - バリデーション
  - 画面 (CSS, Thymeleaf)
  - RESTController
  - 認証　【余裕があれば】

例外処理

例外処理

- 良くないことが起こった時にどうするか？
  - メンバーID被ってたらどうする？
  - メンバーが存在しなかったらどうする？
  - 人のToDoを完了しようとしたらどうする？
- ２つの処理方法
  - 拾うか（catch），投げるか(throw)

検査例外・非検査処理

- **検査例外：Exception**
  - try/catch, throws必須．無いとコンパイル不可
    - IOException, InterruptedException, SQLException
  - 呼出側で回避できない例外
- **非検査例外： RuntimeException**
  - try/catch, throws不要．拾わないと落ちる
    - NullPointerException, IllegalArgumentException ArrayIndexOutOfBoundsException
  - 呼出側で頑張れば回避できる例外

やってはいけないこと

■例外のにぎりつぶし

◆エラーを拾って何も処理しないこと

```
void doSomething() {
    try {
        foo();
    } catch (IOException e) {
        e.printStackTrace();
    }
}
```

ダメ！：スタックトレースは出力されるが何事もなかったように進む

対策1：拾って処理する

```
void doSomething() {
    try {
        foo();
    } catch (IOException e) {
        e.printStackTrace();
        handleException(); //処理
    }
}
```

対策2：拾わず投げる

```
void doSomething() throws IOException {
        foo();
}
```

実践的な例外処理

- **自前の非検査例外に包んで投げなおす**
  - エラーコードを含めてもよい
  - 元の例外(cause)も渡せば原因究明しやすい
  - その場その場で処理しない
    - 処理がコードのあちこちに散らばるため

```
void doSomething() {
    try {
        foo();
    } catch (IOException ex) {
        //ここでは処理を書かずに，非検査例外に包んで投げる
        throw new MyAppException(10001, “〇〇エラー”, ex);
    }
}
```

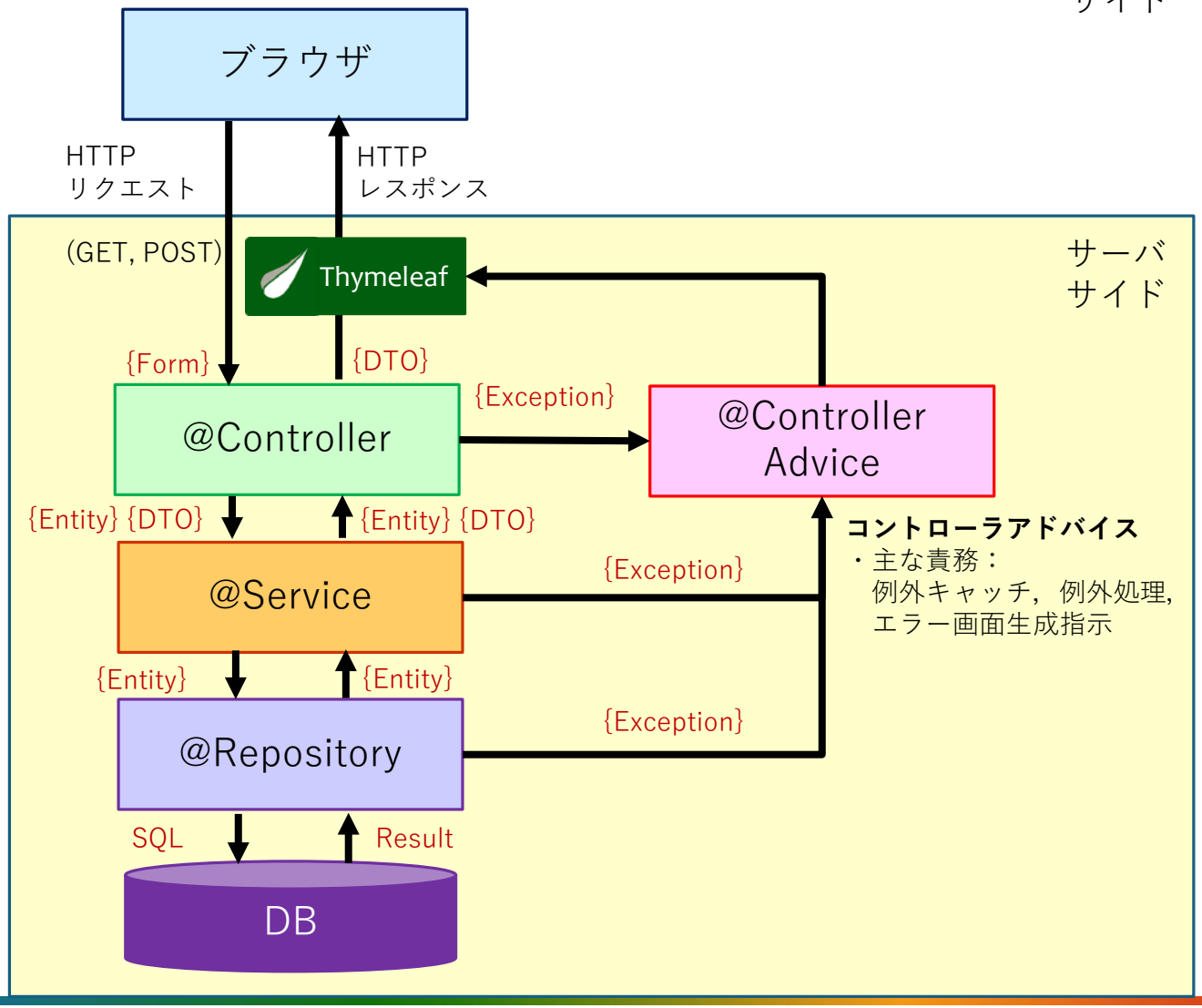
実践的な例外処理

■例外ハンドラを定義する

- ◆ 指定した例外をまとめて処理するメソッド
- ◆ Webアプリの場合，エラー画面を返す
  - ● **コントローラ(@Controller)**のメソッドで定義
    - ✓ または別クラス**@ControllerAdvice**にまとめる
  - ● 例外に応じたメッセージを画面に埋め込む

```
@ExceptionHandler(MyAppException.class)
public void handleException(ToDoAppException ex, Model model)()
{
    String errMsg = ex.getMessage();
     //この間, errMsg を加工してもよい.
    model.addAttribute("exception", errMsg);
    return "error"; //error.html をレンダリング
}
```

例外処理を追加したレイヤ構造



ToDoアプリに例外処理を追加

- wikiを参考にして，ToDoアプリに例外処理を追加してみよう

バリデーション

バリデーション（Validation）

- 入力の妥当性チェック．Webアプリの場合，主にフォームの入力チェックを指す
  - **抜け**がないか
  - **書式に誤り**がないか
    - メールアドレス書式
    - 電話番号書式
    - パスワード書式
  - **妥当な数値**か
    - 範囲チェック

バリデーションの2つの方法

- **クライアントサイド・バリデーション**
  - フォームを入力直後にブラウザ上でチェック
  - 長所： 即時性がありUXに優れる
  - 短所： 入力画面以外からの送信は対応不可
- **サーバサイド・バリデーション**
  - 送られてきたデータをサーバ上でチェック
  - 長所：どこからデータが来ても対応可能
  - 短所：コーディング量多し．UX的に△
- **どちらもやっておくと完璧**

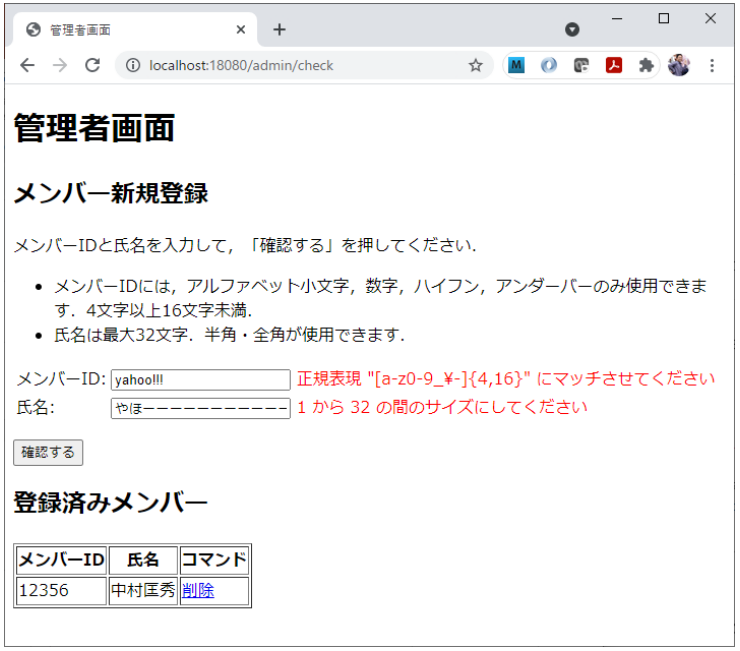
クライアントサイド・バリデーション

- ブラウザ上で入力チェックするやり方
- <input>にオプションをつける
  - required: 必須項目
  - pattern=“パターン”: 正規表現
  - type=“email”: メールアドレス
  - type=“number”：数値
    - min=“最小値” max=“最大値”
  - type=“date”: 日付
  - type=“datetime-local”: 日付時刻

- 画面以外から直接サーバにPOSTされたら無防備
  - curl/postman

サーバサイド・バリデーション

- サーバ側でデータチェックするやり方
  - **Controller層の責務**（水際対策）
    - V1: Formクラスの各属性に**アノテーション**を付与
    - V2: コントローラのメソッドのForm引数の前に**@Validated**をつける
    - V3: フォーム入力画面のHTMLテンプレートに**エラー表示フィールド**を追加



V1: フォームにアノテーションを付与

- フォームクラスの各属性の上にアノテーションをつける
- よく使うもの
  - @NotBlank: 空文字，空白のみでないこと
  - @Pattern(regexp=“正規表現”): パターンマッチング
  - @Email: Emailの書式にあっているか
  - @Size(min= 整数, max= 整数): 桁数範囲
  - @Min(数値), @Max(数値)： 値範囲

- 興味深いもの
  - @CreditCardNumber: 妥当なクレカ番号か
  - @EAN: 妥当なバーコード識別番号か

V2: @Validatedをつける

- フォームを受け付けるメソッド(PostMapping)の引数に@Validatedをつけ，その次にBindingResultという変数を追加する
- メソッドの内部で，エラーが出たらフォームを表示するメソッド(GetMapping)に戻るようにする

```
@PostMapping("/check")
  String checkUserForm(@Validated @ModelAttribute(name = "MemberForm") MemberForm form,
        BindingResult bindingResult, Model model) {
    // 入力チェックに引っかかった場合、ユーザー登録画面に戻る
    if (bindingResult.hasErrors()) {
      // GETリクエスト用のメソッドを呼び出して、ユーザー登録画面に戻る
      return showUserForm(form, model);
    }
    model.addAttribute("MemberForm", form);
    return "check";
  }
```

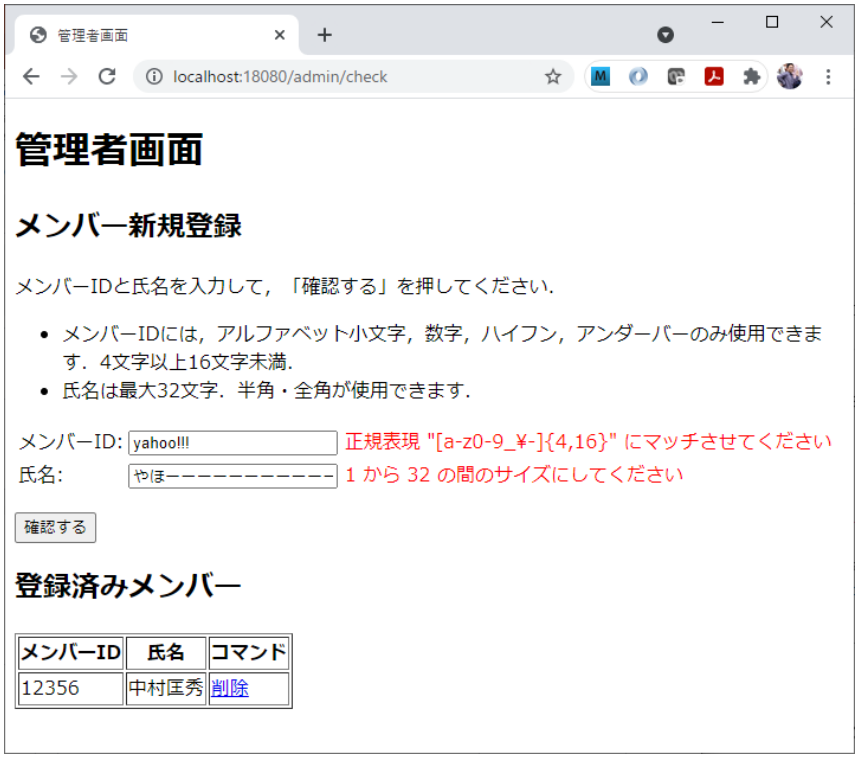
V3: HTMLにエラー表示フィールドをつける

- Thymeleafのif を使ってエラー時のみ表示されるようにしておく

```
<form role="form" th:action="@{/admin/check}" th:object="${MemberForm}" method="post">
    <table>
      <tr>
        <td><label>メンバーID: </label></td>
        <td><input type="text" required th:field="*{mid}" />
          <span th:if="${#fields.hasErrors('mid')}" th:errors="*{mid}" style="color: red"></span>
        </td>
      </tr>
      <tr>
        <td><label>氏名: </label></td>
        <td><input type="text" required th:field="*{name}" />
          <span th:if="${#fields.hasErrors('name')}" th:errors="*{name}" style="color: red"></span>
        </td>
      </tr>
    </table>
    <p><input type="submit" value="確認する" /></p>
</form>
```

ToDoアプリにバリデーションを追加

- wikiを参考にして，ToDoアプリにバリデーションを追加してみよう



RESTコントローラ

根源的な要求

- ToDoアプリの画面以外からでもToDo管理したい
  - LINEから直接登録したい
  - バーチャルエージェントにToDo管理してもらいたい
- 画面は不要．ToDo管理のコア機能（Service）のみ使いたい
  - HTTPでリクエストしたら，結果がオブジェクト(JSON)で返ってきてほしい
  - 結果をどう処理するかは，各アプリで考えたい

ToDo管理のAPIを設計

■ToDoのCRUDをRESTFulに実行するAPI

操作	API呼び出し	戻り値
ToDo作成	POST /api/{uid}/todos {"title": "タイトル"}	作成されたToDoオブジェクト
ToDo取得	GET /api/{uid}/todos/{seq}	指定したToDoオブジェクト
ToDo取得	GET /api/{uid}/todos	そのユーザのToDoリスト
ToDo取得	GET /api/{uid}/dones	そのユーザのDoneリスト
ToDo更新	PUT /api/{uid}/todos/{seq}/done	完了したToDoオブジェクト
ToDo更新	PUT /api/{uid}/todos/{seq} {"title":"変更するタイトル"}	更新されたToDoオブジェクト
ToDo削除	DELETE /api/{uid}/todos/{seq}	true

例外・ステータスマッピング

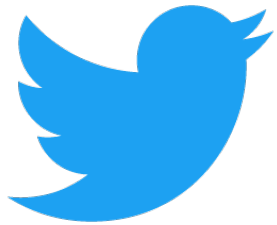
- 例外発生時，Controllerではエラー画面を返せばよかったが，RestControllerではリソースに対する**操作の結果をHTTPステータス返す**ことが望ましい
  - 存在しない系例外　→　404 Not Found
  - 引数おかしい系例外　→　400 Bad Request
  - アクセス違反系例外　→　403 Forbidden
  - 想定外の例外　→　500 Internal Server Error
- ResponseEntityに例外を包んで返すと簡単

```
@ExceptionHandler(Exception.class)
  public ResponseEntity<Object> handleException(Exception ex) {
    return new ResponseEntity<>(ex, HttpStatus.INTERNAL_SERVER_ERROR);
  }
```

RESTコントローラ

- アノテーション@RestController を付与したコントローラ
- 画面を返すのではなく，オブジェクトを返す
- 戻り値は自動的にJSONオブジェクトに変換されて，HTTPレスポンスで返される
- 画面や画面遷移を伴わないのでシンプル
  - ほぼサービスをラップしているだけ
  - 画面や遷移はブラウザ側で制御する
    - JavaScript等でやる

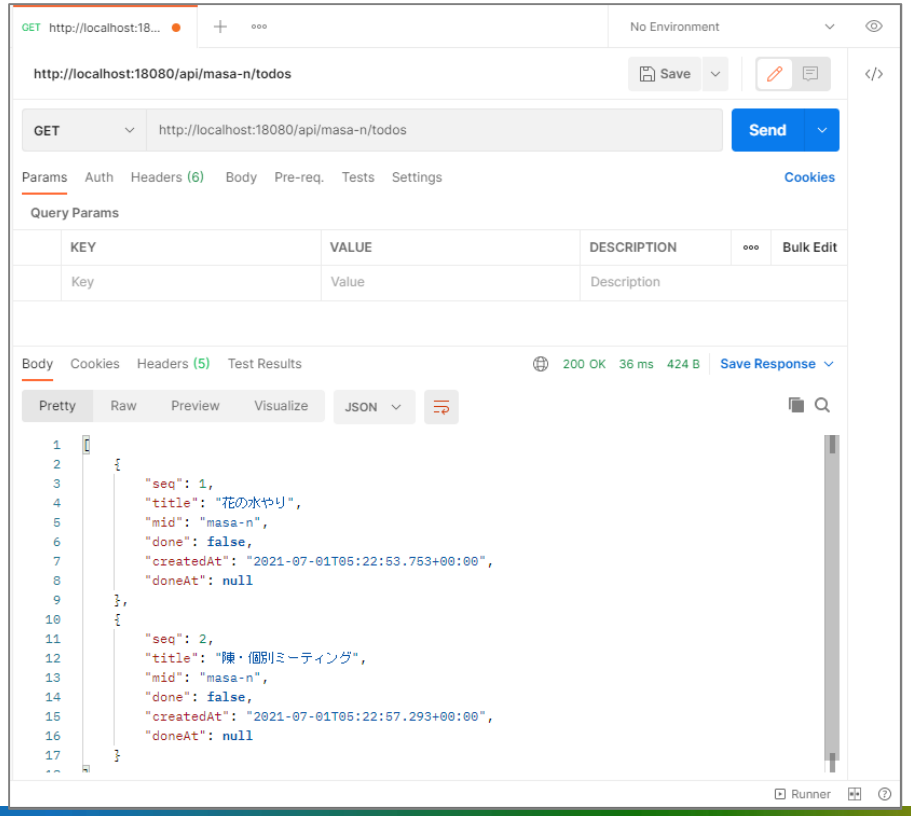
APIで外部アプリと連携

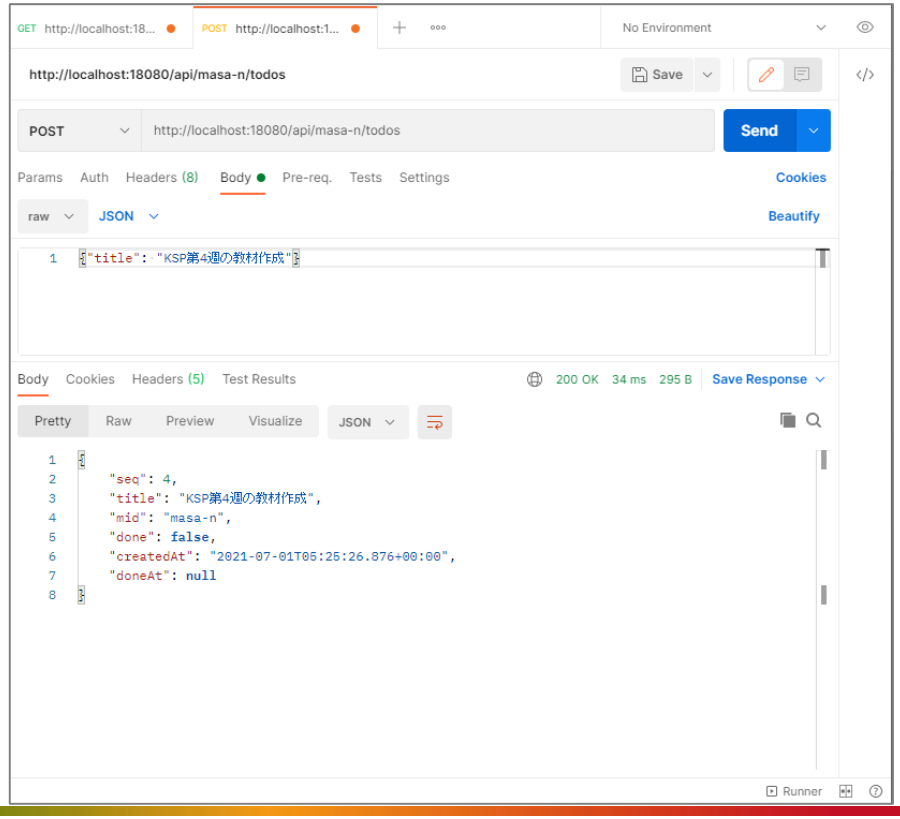


REST
API

ToDoアプリのRESTコントローラを実装

- wikiを参考にして，ToDoRestController.java を実装してみる
- Postmanでテストしてみよう





認証・認可

認証とは

- 認証（Authentication）
  - 通信の相手が誰であるかを確認すること
  - セキュリティが必要なアプリでは必須

認証に必要なもの

- ユーザ識別子
  - 自分は誰かを表す情報
    - ユーザID，社員番号，Eメール，etc

- 認証手段
  - 本人を証明するための情報
    - パスワード，指紋，顔，網膜，etc.

- Webアプリの場合，ユーザIDとパスワードが利用される

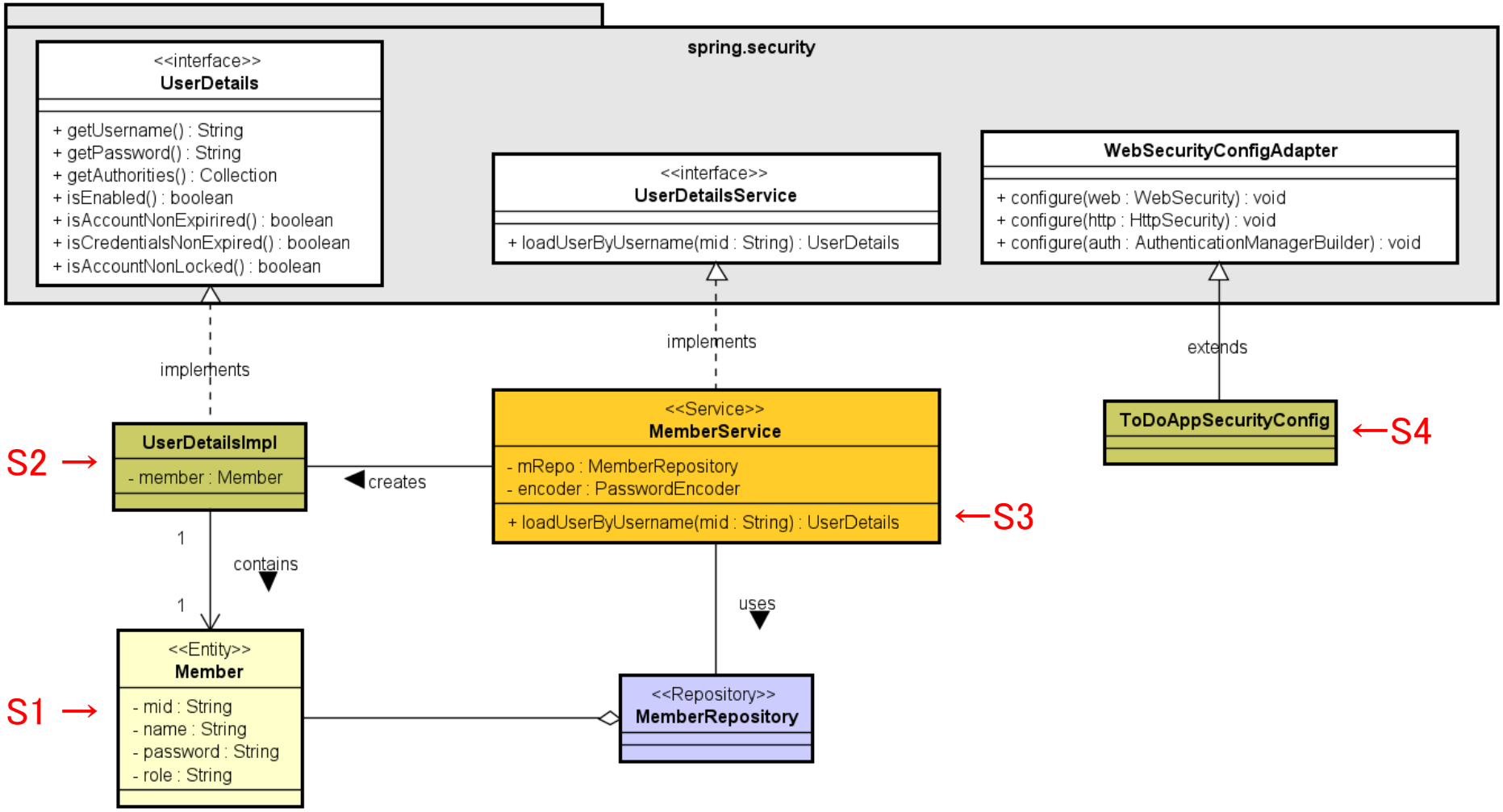
認可とは

- 認可（Authorization）
  - 認証されたユーザが処理やリソースに**どこまでアクセスしてよいか**を管理する仕組み
  - **ロール**を定義して権限分掌する

Spring Security

- 認証・認可に必要な典型的な処理を扱う枠組み
- 以下の4つの手順で実施する
  - S1: ユーザのエンティティにパスワードとロールの属性を追加する
  - S2: UserDetailsの実装クラスを実装する
  - S3: ユーザのサービスにloadUserByUsername()を実装する
  - S4: Webアプリ全体のセキュリティ設定を行う

ToDoアプリの場合のクラス図



S1: パスワードとロールを追加

- ユーザ管理のためのエンティティクラスにパスワードとロールの属性を追加する
  - パスワードは暗号化済のものを入れる（後述）
  - ロールはStringあるいはenumで定義
    - 分掌したい権限を予め決める（一般，管理者）
  - 関連するフォームも更新

```
@Entity
public class Member {
   @Id
   String mid;   //メンバーID
   String name;  //名前
   String password; //パスワード（暗号化済）
   String role; //ロール
}
```

```
@Data
public class MemberForm {
   @Pattern(regexp ="[a-z0-9_¥¥-]{4,16}")
   @NotBlank
   @Size(min = 1, max = 32)
   String name; //名前．最大32文字
   @NotBlank
   @Size(min = 8)
   String password; //パスワード
   String role = "MEMBER"; //ロール．デフォルトは
"MEMBER"
   :
}
```

S2: UserDetailsの実装クラス

- 新しいDTOとしてUserDetailsImpl を作成
  - Springが認証に使うユーザクラス
  - アプリのユーザクラスをラップしてギャップを埋める
    - ユーザID取得
    - パスワード取得
    - 権限取得
    - 凍結フラグ等

```
public class UserDetailsImpl implements UserDetails {
   Member member;
   Collection<GrantedAuthority> authorities = new ArrayList<>(); //権限リスト
   public UserDetailsImpl(Member member) {
      this.member=member;
      //メンバーのロールから権限を生成して追加
      this.authorities.add(new SimpleGrantedAuthority("ROLE_" + member.getRole()));
   }
   @Override
   public Collection<? extends GrantedAuthority> getAuthorities() {
      return authorities;
   }
   @Override
   public String getPassword() {
      return member.getPassword();
   }
   @Override
   public String getUsername() {
      return member.getMid();
   :  //ほかのメソッドはすべてreturn true; としておけばOK
}
```

S3: loadUserByUsername()を実装

- ユーザクラスを管理するサービスに，UserDetailsServiceを継承させ，メソッドloadUserByUsername()を実装する
  - パスワード・エンコーダも入れておく
  - ユーザを生成するメソッドについて，リポジトリへセーブする前にパスワードをエンコーダで暗号化するよう変更する
  - 管理者を登録するメソッドも作っておく

- コードはwiki参照のこと

S4: Webアプリ全体のセキュリティ設定

- セキュリティ設定を行う設定クラスを作成し，3種類のconfigure()メソッドを実装する(詳細はwiki)

```
@Configuration
@EnableWebSecurity  //  (1) Spring Securityを使うための設定
public class WebBasicAuthSecurityConfiguration extends WebSecurityConfigurerAdapter {

    @Override
    public void configure(WebSecurity web) throws Exception {
        //  (2) 主に全体に対するセキュリティ設定を行う
        //  web.ignoring().antMatchers("/css/**","/js/**","/images/**");
    }

    @Override
    protected void configure(HttpSecurity http) throws Exception {
        //  (3) 主にURLごとに異なるセキュリティ設定を行う
    }

    @Override
    protected void configure(AuthenticationManagerBuilder auth) throws Exception {
        //  (4) 主に認証方法の実装の設定を行う
    }
}
```

認証・認可のメリットデメリット

- メリット
  - セキュリティをかけられる
- デメリット
  - ユーザへの負担（パスワード管理）
  - コードが複雑化する
  - 運用が大変（漏洩対策，忘却時の復旧, etc.）
- 委任認証
  - 別の認証サービスに任せる (OAuth2)

- 開発するアプリで本当に必要かを考えよう