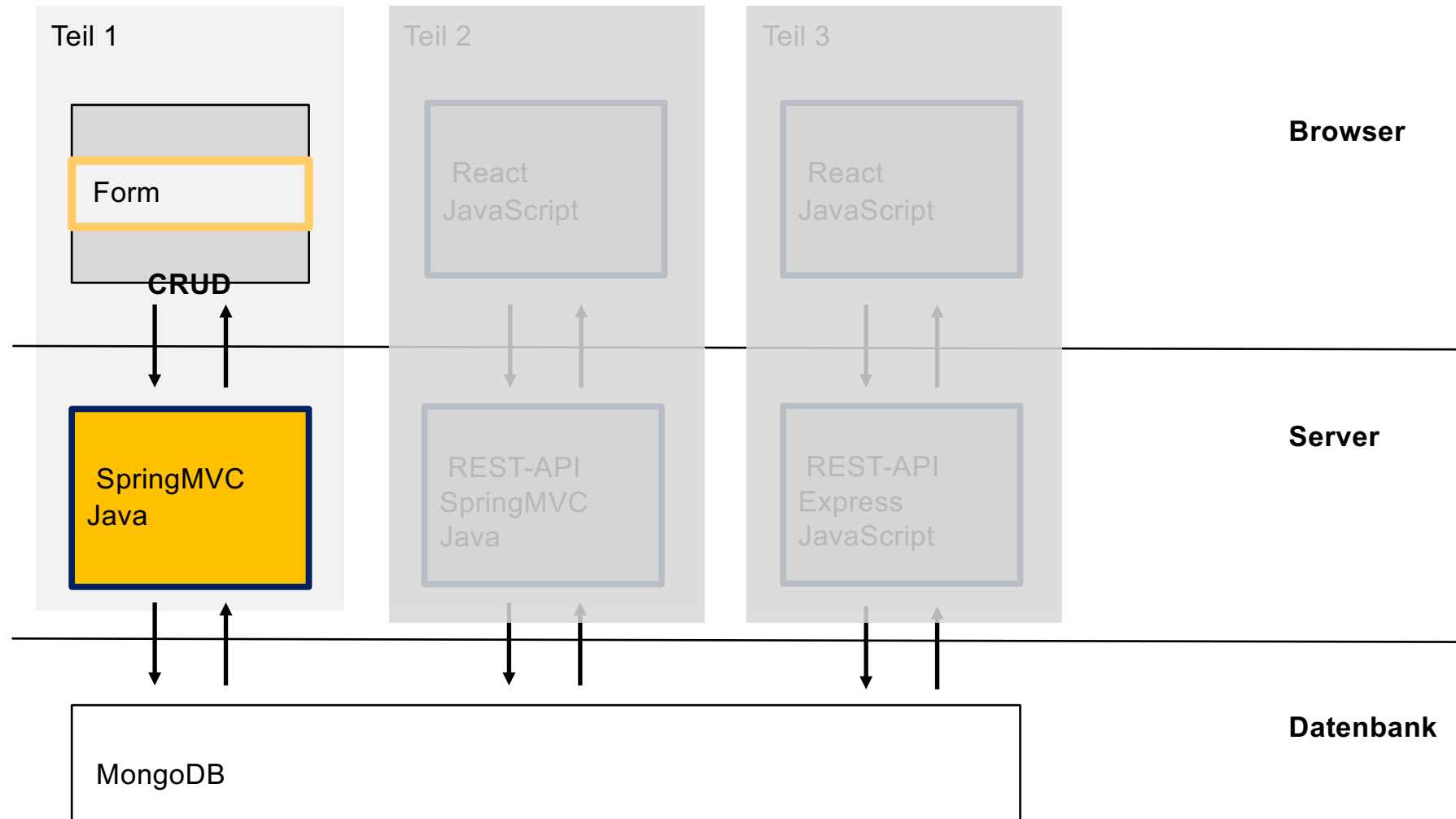


Validierung in Formulare
CRUD Operationen
Error Handling
Unit Tests der Controller

Themen heute

- Besprechung Übung 4
- Web Formulare: Wichtige Aspekte
 - Double-Submit Problem
 - Input Validierung
- Delete & Update
- Error Handling
- Testing

Lab "flashcard": Setup



Besprechung Übung 4 (1/6)

- Die Subview ist durch das File "create.html" im Ordner "src/main/resources/templates/questionnaires" implementiert.
- Die zentralen Elemente in der Subview sind:
 - Das Model-Objekt hinter dem Formular
 - Eine Instanz von Questionnaire
 - Der Zugriff auf das Model-Objekt
 - Über den Key "questionnaire" in einer "Model"-Instanz
 - Thymeleaf Expression `th:object="${questionnaire}"`
 - Der Zugriff auf Properties dieses Model-Objektes, wie "title"
 - Über die Thymeleaf Funktion `th:field="*{title}"`
 - HTTP Request, der aus dem Formular ausgelöst werden soll
 - HTTP-POST auf `th:action="@{/questionnaires}"`

Besprechung Übung 4 (2/6)

```
<form action="#" th:action="@{/questionnaires}"
        th:object="${questionnaire}" method="post" >
    <input type="text" th:field="*{title}"/>
    <input type="submit" value="New"/>
</form>
```

`th:action="@{/questionnaires}"`

Legt für das Formular die Aktion als Request URL fest. Notation `@{...}` ist wichtig.

`th:object="${questionnaire}"`

Referenziert das Model-Objekt über den Key "questionnaire"

`th:field="*{title}"`

Referenziert die Property "title" des Model-Objekts. Notation `*{...}` ist wichtig.

Das Model-Objekt muss eine entsprechende Setter-Methode zur Verfügung stellen.

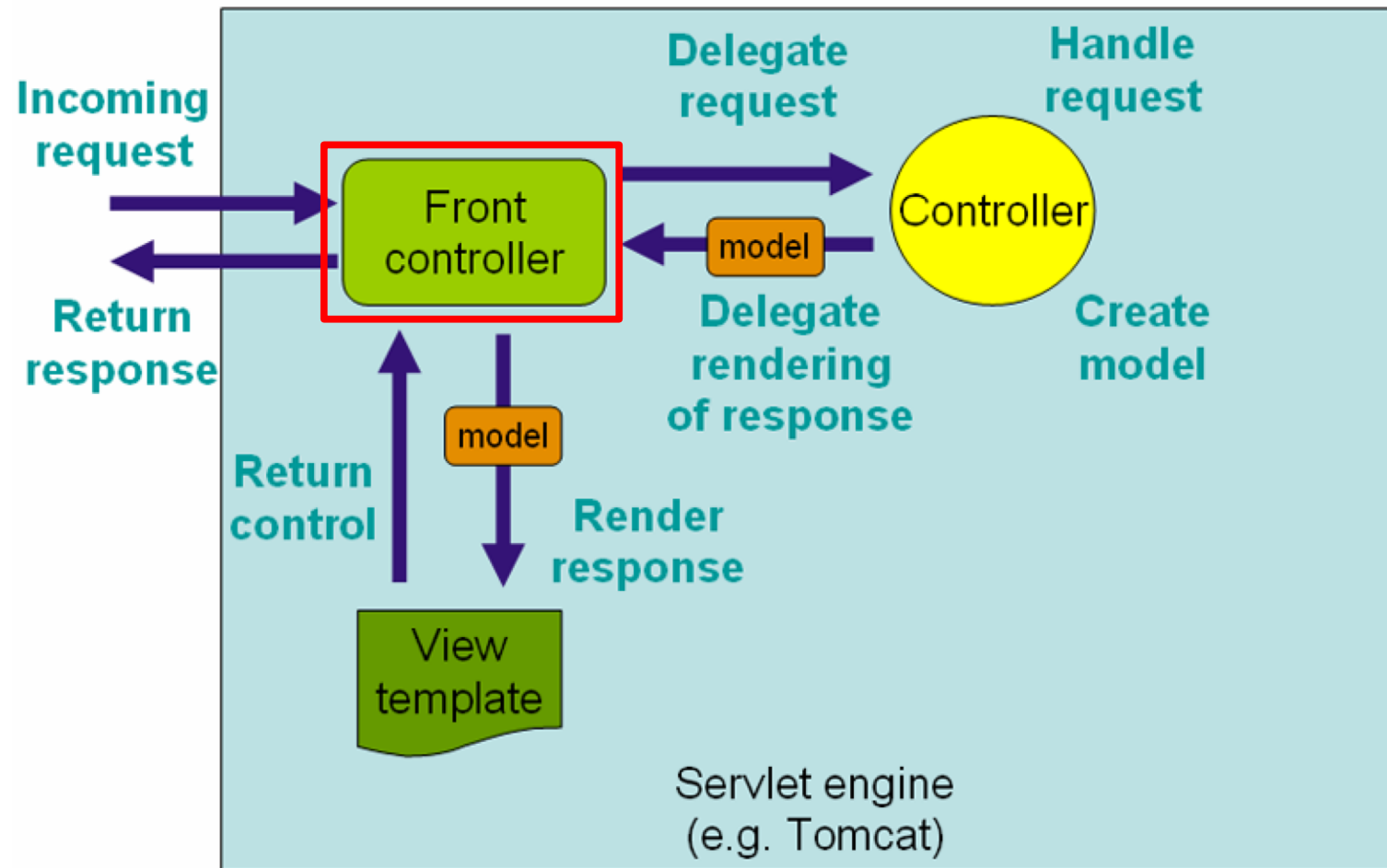
Besprechung Übung 4 (3/6)

QuestionnaireController mit CREATE Funktionalität:

```
@RequestMapping(params = "form", method = RequestMethod.GET)
public String createForm(Model model) {
    model.addAttribute("questionnaire", new Questionnaire());
    return "questionnaires/create";
}
```

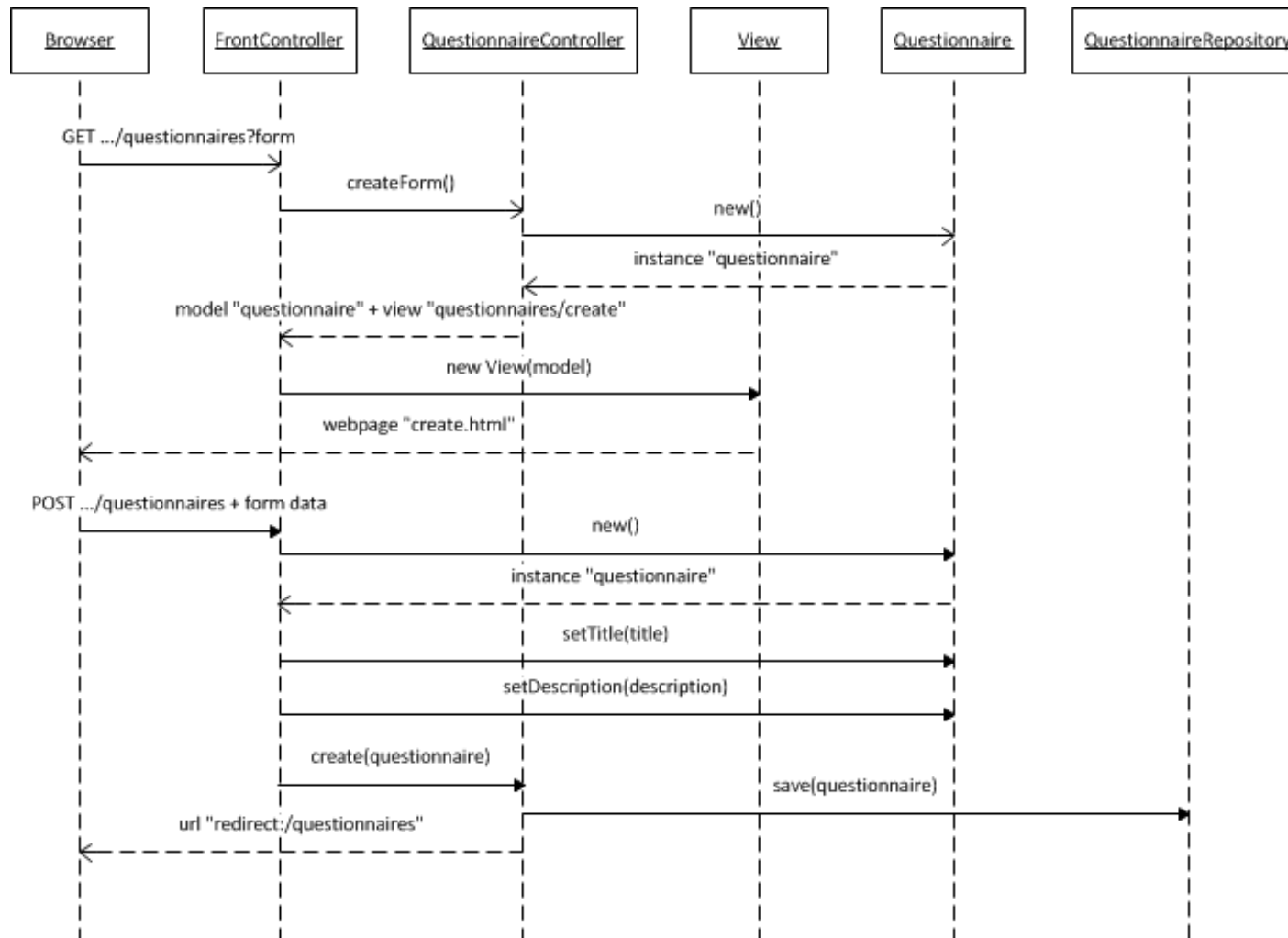
```
@RequestMapping(method = RequestMethod.POST)
public String create(Questionnaire questionnaire) {
    questionnaireRepository.save(questionnaire);
    return "redirect:/questionnaires";
}
```

Besprechung Übung 4 (4/6)



see Spring Reference Documentation "22.2 The DispatcherServlet"

Besprechung Übung 4 (5/6)



Besprechung Übung 4 (6/6)

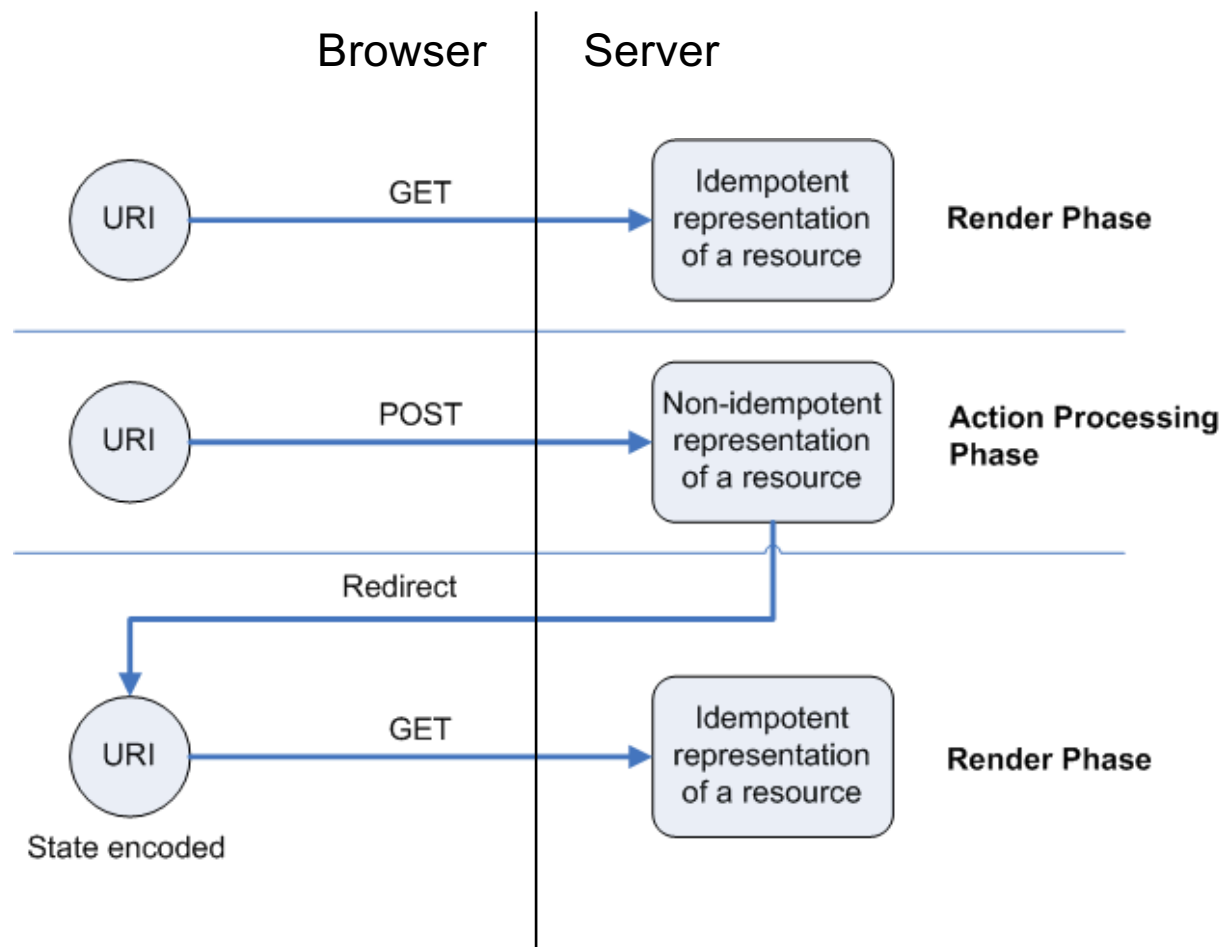
■ Zusammenfassung **Request Mapping**

- Für das Mapping der Handler Methoden sind folgende Annotation wichtig:
 - **@RequestMapping** (siehe AB5)
 - **value** Mapping auf ein Path-Element
 - **method** Mapping auf eine HTTP Methode (oder **@GetMapping**, **@PostMapping**, ...)
 - **params** Mapping auf einen Request Parameter
 - headers, produces, consumes, ...
- Zusätzlich kann mit folgenden Annotationen bei den Input Parameter Elemente aus der Request-URL in der Methode zugänglich gemacht werden:
 - **@PathVariable** (siehe AB5)
 - **@RequestParam** (siehe UB3)

RELOAD und das Double-Submit Problems

- Views auf Geschäftsdaten werden ausschliesslich über GET Anfragen geholt.
 - Solche Anfragen sind 'safe' in dem Sinne, dass sie auf dem Server keine Änderungen verursachen und damit idempotent sind.
- Änderungen der Geschäftsdaten erfolgen über POST Anfragen.
 - Solche Anfragen verändern den Datenbestand.
 - Eine Mehrfachverarbeitung der gleichen POST Anfrage ist normalerweise ungewollt und deshalb ein Fehler.
 - Eine Mehrfachverarbeitung kann z.B. über den RELOAD Button ausgelöst werden.
- **Das POST-Redirect-GET Pattern regelt die semantisch korrekte Verwendung von POST und GET Anfragen.**

Redirect-after-Post Pattern

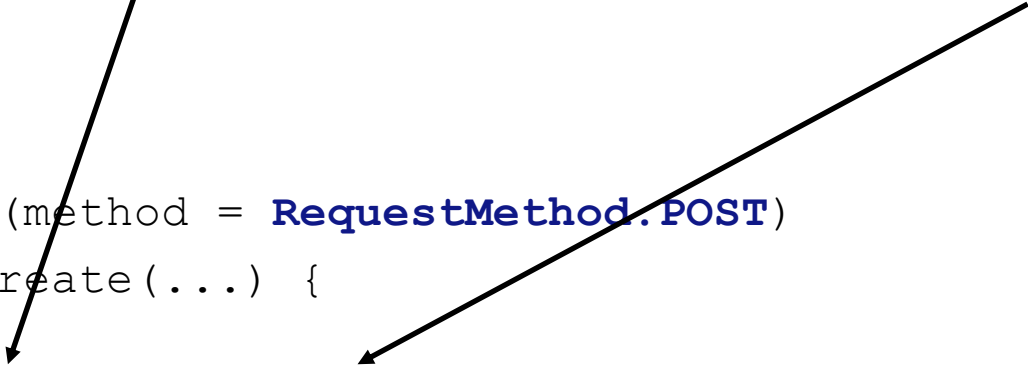


Redirect-after-Post in SpringMVC

- Die Unterstützung in SpringMVC für das Pattern ist sehr gut.

"The special **redirect:** prefix allows you to accomplish this. If a view name is returned that has the **prefix redirect:**, the `UrlBasedViewResolver`(and all subclasses) will recognize this as a special indication that a redirect is needed. The rest of the view name will be treated as the **redirect URL**."

```
@RequestMapping(method = RequestMethod.POST)
public String create(...) {
    ...
    return ("redirect:/questionnaires");
}
```



Validierung einer Formular-Eingabe

- Die **Validierung der Eingabe ist sehr wichtig**, um die Konsistenz in der Datenbank erhalten zu können.
- Die Validierung kann auf verschiedenen Schichten erfolgen
 - ☐ Datenbank
 - ☐ Controller
 - ☐ View
- **JSR-303 Bean Validation API**
 - ☐ für eine Validierung auf der Serverseite in den Schichten:
 - Datenbank
 - Controller

Die wichtigsten JSR-303 Annotationen

@Null @NotNull	Die Referenz muss null beziehungsweise nicht null sein.	Referenzvariablen
@AssertTrue @AssertFalse	Das Element muss true beziehungsweise false sein.	boolean/Boolean
@Min(value=) @Max(value=)	Muss eine Zahl und grösser/kleiner oder gleich dem Wert sein.	byte/Byte, short/Short, int/Integer, long/Long, BigInteger, BigDecimal
@DecimalMin(value=) @DecimalMax(value=)	Muss eine Zahl und grösser/kleiner oder gleich dem Wert sein.	Double/Double und float/Float sowie String, byte/Byte, short/Short, int/Integer, long/Long, BigInteger, BigDecimal
@Size([min=],[max=])	Die Grösse muss sich in einem Intervall bewegen.	String, Collection, Map, Feld
@Digits(integer=,fraction=)	Das Element muss eine gegebene Anzahl an Stellen besitzen.	String, byte/Byte, short/Short, int/Integer, long/Long, BigInteger, BigDecimal
@Past @Future	Das Element ist ein Datum in der Vergangenheit/Zukunft bezogen auf jetzt.	Date, Calendar
@Pattern(regex=,[flags=])	Der String muss einem Pattern gehorchen.	String

JSR-303 Beispiel der Entität

```
public class Player {  
    @NotNull  
    private String name;  
  
    @Size(min=5, max=20)  
    private String nickname;  
  
    @Min(10)  
    @Max(110)  
    private int age;  
  
    ...  
}
```

- Die Property "name" darf nicht NULL sein.
- Die Länge der Property "nickname" muss zwischen 5 und 20 sein (jeweils inklusiv).
- Der Wert der Property "age" muss zwischen 10 und 110 liegen (jeweils inklusiv).

@Valid Annotation

- Das Validation-Framework kann ganze Objektgraphen bearbeiten und so eine "tiefe" Validierung durchführen.
 - Automatisch wird diese tiefe Prüfung aber nicht durchgeführt.
 - Mit **@Valid** Annotation kann die Prüfung forciert werden.

```
@RequestMapping(method = RequestMethod.POST)
public String create(@Valid Questionnaire questionnaire,
                    BindingResult bindingResult, Model model) {
    if (bindingResult.hasErrors()) {
        ...
    }
    ...
}
```


Validator

- Notwendige jar-Bibliotheken
 - **validation-api-2.x.x.jar**
stellt das API gemäss JSR-303 zur Verfügung
 - **hibernate-validator-6.x.x.jar**
enthält die konkrete Implementation des API
- **Diese Bibliotheken werden von Spring Boot über das Dependency Management automatisch geladen**, sobald eine Abhängigkeit zu einem Web Context besteht.
- Auch die Aktivierung dieser Validatoren ist automatisiert und basiert auf der Konvention, dass die entsprechenden jar-Bibliotheken über den CLASSPATH vorhanden sind.

Validierungsfehler in der Thymeleaf View

■ Integration des Validation Frameworks von Spring in Thymeleaf über:

- Thymeleaf Class **Fields** mit Methoden

```
boolean hasErrors(String field)
```

```
List<String> errors()
```

- Referenz von Fields in der Subview über **#fields**

```
${#fields.hasErrors('title')}
```

- Zugriff auf Fehlermeldung eines Feldes

```
th:errors="*{title}"
```

■ Beispiel:

```
<span th:if="${#fields.hasErrors('title')}" th:errors="*{title}"  
      class="text-danger">  
    Incorrect title  
</span>
```

Arbeitsblatt 9: Validierung implementieren

- Aufgabe 1: Validierung auf Datenbankschicht einführen
- Aufgabe 2: Validierung im Controller einführen
- Aufgabe 3: Validierungsfehler anzeigen

CRUD Operationen vervollständigen

■ QuestionnaireController

Operation	HTTP Methode	Exists
CREATE .../questionnaires	POST	x
READ .../questionnaires .../questionnaires/{id}	GET GET	x x
UPDATE .../questionnaires/{id}	PUT	–
DELETE .../questionnaires/{id}	DELETE	–

Browser, HTML5 und HTTP Methoden

from the HTML5 Spec: 4.10.19.6 Form submission

...

The *method* and *formmethod* content attributes are enumerated attributes with the following keywords and states:

- The keyword *get*, mapping to the state **GET**, indicating the HTTP GET method.
- The keyword *post*, mapping to the state **POST**, indicating the HTTP POST method.
- The keyword *dialog*, mapping to the state **dialog**, indicating that submitting the form is intended to close the dialog box in which the form finds itself, if any, and otherwise not submit.

... und was ist mit HTTP PUT und HTTP DELETE?

CRUD-Support: HiddenHttpMethodFilter

```
package org.springframework.web.filter;

...

private String methodParam = "_method";

protected void doFilterInternal(HttpServletRequest request,
                                HttpServletResponse response, FilterChain filterChain)
    throws ServletException, IOException {
    String paramValue = request.getParameter(this.methodParam);
    if ("POST".equals(request.getMethod()) && StringUtils.hasLength(paramValue)) {
        String method = paramValue.toUpperCase(Locale.ENGLISH);
        HttpServletRequest wrapper = new HttpMethodRequestWrapper(method, request);
        filterChain.doFilter(wrapper, response);
    } else {
        filterChain.doFilter(request, response);
    }
}
```

HiddenHttpMethodFilter supports the **conversion of HTTP method** by finding **hidden input parameter** that defines the actual HTTP Method.

CRUD-Support DELETE: Hidden Field

```
<form action="#" th:action="@{/questionnaires} + '/' + ${questionnaire.id}"
      method="post" >
  <input type="hidden" name="_method" value="DELETE"/>
  <button type="submit">Delete</button>
</form>
```

Request URL	http://localhost:8080/flashcard-mvc/questionnaires/1
Post Data	_method=DELETE

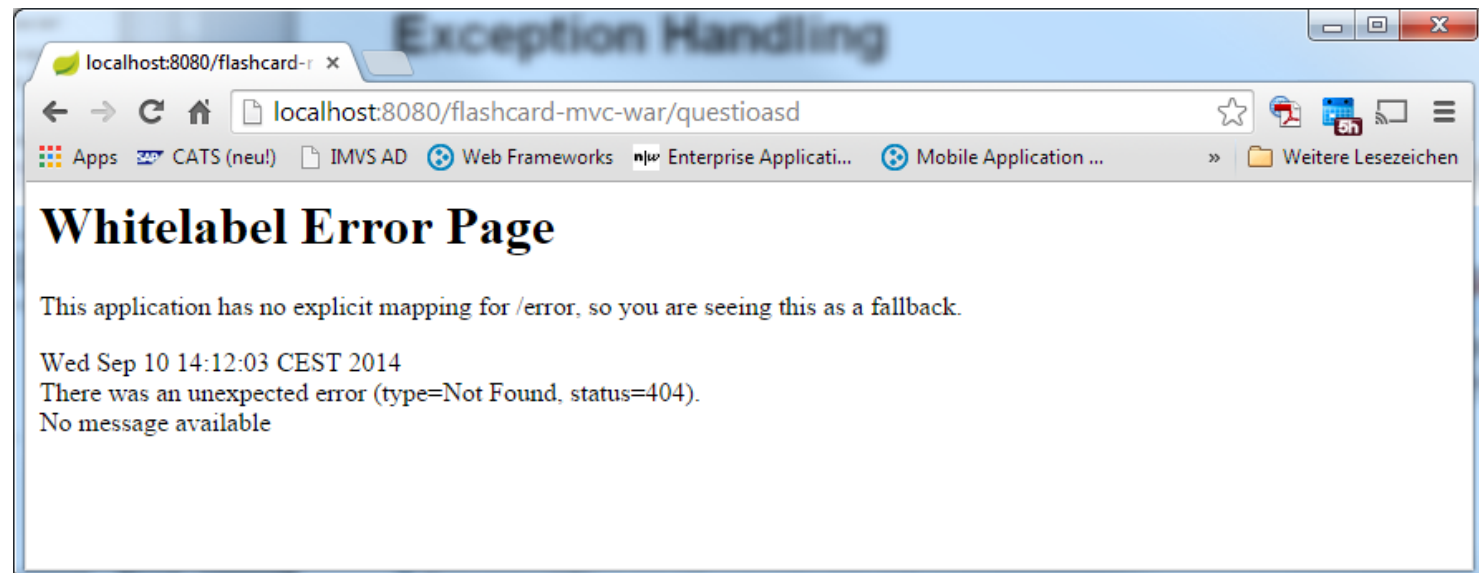
Arbeitsblatt 10: Delete Funktion

- QuestionnaireController ergänzen
 - ☐ **RequestMapping auf HTTP-Methode DELETE?**
 - ☐ Löschen der Entität
 - ☐ Response generieren

- Subview anpassen
 - ☐ "list.html" ergänzen
 - ☐ **Wie wird der "HiddenHttpMethodFilter" aus der HTML Page aktiviert?**

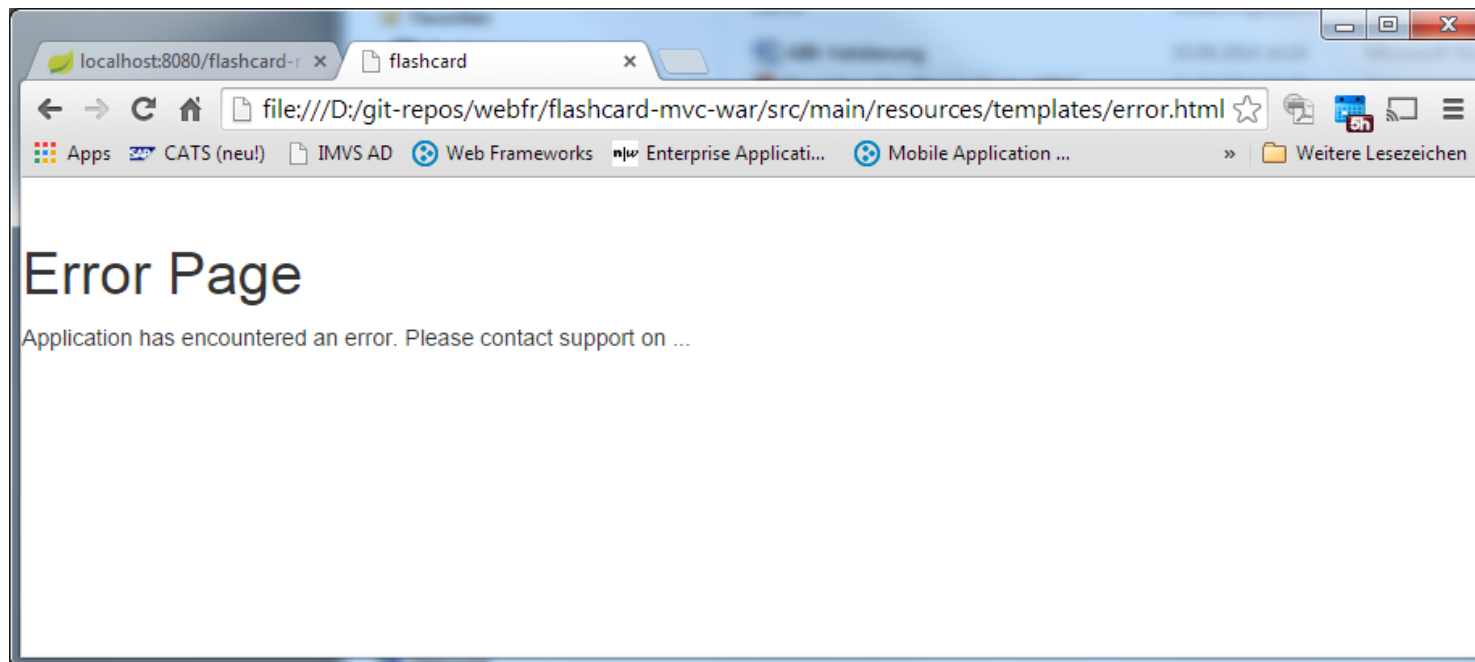
Error Handling

- Spring Boot antwortet bei einem Fehler (Webpage nicht vorhanden, Entität nicht vorhanden, ...) mit einer Standard Error Page.
- Negativ:
 - Layout passt nicht zur restlichen Applikation
 - Keine Meldung über den tatsächlichen Fehler



Default Error über Subview "error.html"

- File "error.html" muss das UI der Webapp übernehmen
 - Für Main Content im Composite View Pattern
 - Als Thymeleaf Fragment "content"
 - In Folder "templates" ablegen



Spezifische Error Pages für HTTP Status Codes

- Mit Spring und Thymeleaf lassen sich HTTP Status Codes einfach auf entsprechende HTML Subviews abbilden.
- Beispiel: **Status Code 404 "Not Found"**
 - Bei "findById(), update(), delete()" wird jeweils der Key der entsprechenden Entität übergeben. Ist diese Entität nicht vorhanden, sollte dem User eine entsprechende Fehlermeldung angezeigt werden.
 - Der HTTP Status Code 404 "Not Found" kann dafür verwendet werden.
 - Es braucht für diesen Fehlercode eine entsprechende Subview
 - Subview "404.html" erstellen
 - String "404" als View Name in der Methode zurückgeben

Arbeitsblatt 11: Error Handling

- Fehler-Handling im QuestionnaireController einführen
- Entsprechende Subviews implementieren

Testing Spring Controller (1/2)

```
@RunWith(SpringRunner.class)
@WebMvcTest(QuestionnaireController.class)
public class QuestionnaireControllerTest {
    @Autowired
    private MockMvc mockMvc;

    @MockBean
    private QuestionnaireRepository questionnaireRepositoryMock;

    @Before
    public void setUp() {
        Mockito.reset(questionnaireRepositoryMock);
    }
    ...
}
```

Testing Spring Controller (2/2)

...

```
@Test
public void create_NewQuestionnaire_ShouldReturnOK() throws Exception {
    Questionnaire q1 = new QuestionnaireBuilder("1")
        .description("MyDescription 1")
        .title("MyTitle 1")
        .build();

    when(questionnaireRepositoryMock.save(q1)).thenReturn(q1);

    mockMvc.perform(post("/questionnaires")
        .contentType(MediaType.APPLICATION_FORM_URLENCODED)
        .param("description", "MyDescription 1")
        .param("title", "MyTitle 1")
    )

    .andExpect(status().is3xxRedirection())
    .andExpect(view().name("redirect:/questionnaires"));
}
```

Übung 5: Update Formular implementieren

- Hausaufgabe
- UPDATE (CRUD) umsetzen: **Analog zu CREATE!**
 1. **GET Request**
Update Formular beim Server holen
 2. **PUT Request**
Existierender Questionnaire über PUT Request aktualisieren