Inhaltsverzeichnis

1 Servlet

`javax.servlet.Servlet` ist das Interface, das letztendlich entscheidet, ob eine Java-Klasse überhaupt ein Servlet ist. Jedes Servlet muss dieses Interface direkt oder indirekt implementieren. De facto wird man aber wohl in den meisten Fällen nicht das Interface selber implementieren, sondern auf `javax.servlet.GenericServlet` (falls man kein Servlet für HTTP-Anfragen schreibt), bzw. `javax.servlet.http.HttpServlet` für http-Anfragen zurückgreifen. HttpServlet implementiert alle wesentlichen Methoden bis auf die http-doXXX-Methoden wie doGet oder doPost. Man muss nur noch die Methoden doGet, doPost oder 'doXYZ überschreiben, je nachdem, welche HTTP-Methoden man unterstützen will.

```
public class BasicServlet extends HttpServlet {

    enum Directories {questionnaires, funny}

    protected void doGet(HttpServletRequest request, HttpServletResponse response) throws IOException {

        response.setContentType("text/html; charset=utf-8");

        String[] pathElements = request.getPathInfo().split("/");
        Arrays.stream(pathElements).forEach(log::debug);

        if (pathElements.length > 0) {

            if (Arrays.stream(Directories.values()).anyMatch(d -> d.name().equals(pathElements[1]))) {

                switch (Directories.valueOf(pathElements[1])) {
                    case questionnaires:
                        Long id = pathElements.length == 3 ? Long.valueOf(pathElements[2]) : null;
                        // Path contains ID
                        if (id != null) {
                            handleSingleQuestionaire(id, request, response);
                        } else {
                            handleQuestionaires(request, response);
                        }
                        return;
                    default:
                        handleRoot(request, response);

                }
            } else {
                handelDefault(request, response);
            }
        } else {
            handleRoot(request, response);
        }
    }

    private void handleSingleQuestionaire(Long id, HttpServletRequest request, HttpServletResponse
    ↪  response) throws IOException {
       ...
    }

    private void handleQuestionaires(HttpServletRequest request, HttpServletResponse response) throws
    ↪  IOException {
        QuestionnaireRepository questionnaireRepository = (QuestionnaireRepository)
        ↪  request.getServletContext().getAttribute(QuestionnaireRepository.class.getName());
        PrintWriter writer = response.getWriter();
        writer.append(HTML_HEAD);
        writer.append("<h3>{questionnaires}</h3>");
        List<Questionnaire> questionnaires = questionnaireRepository.findAll();
        questionnaires.forEach(questionnaire -> {
            final String url = request.getContextPath() + request.getServletPath() + "/questionnaires/" +
            ↪  questionnaire.getId().toString();
            writer.append(String.format("<p><a href='%s'>%s</a></p>", response.encodeURL(url),
            ↪  questionnaire.getTitle()));
        });
        writer.append(HTML_BODY_END);
    }

    private void handleRoot(HttpServletRequest request, HttpServletResponse response) throws IOException {
       ...
    }
}
```

1.1 Servlet Container

Der Servlet Container stellt eine Umgebung zur Verfügung indem die Servlets laufen gelassen werden können. Der Container sorgt zunächst einmal für den korrekten Lebenszyklus der Servlets. Zumeist arbeitet der Container im Zusammenspiel mit einem Webserver.

Laden der Servlet Klasse

Zunächst muss der Classloader des Containers die Servlet-Klasse laden. Wann der Container dies macht, bleibt ihm überlassen, es sei denn, man definiert den entsprechenden Servlet-Eintrag im Deployment-Deskriptor. Mit dem optionalen Eintrag «load-on-startup» wird definiert, dass der Servlet-Container die Klasse bereits beim Start des Containers lädt. Die Zahlen geben dabei die Reihenfolge vor (Servlets mit niedrigeren «load-on-startup»-Werten, werden früher geladen).

1.2 Servlet Request und Response

In den Verarbeitungsmethoden von Servlets (`service()`, `doGet()`, `doPost()`...) hat man stets Zugriff auf ein Objekt vom Typ ServletRequest. Arbeitet man mit HttpServlet handelt es sich um das Interface `javax.servlet.http.HttpServletRequest`, ansonsten um das Interface `javax.servlet.ServletRequest`. Diese Interfaces stellen wesentliche Informationen über den Client-Request zur Verfügung.

1.3 ServletContext

Jedes Servlet wird im Kontext der Webapplikation ausgeführt. Dieser Kontext gilt für alle Servlets der entsprechenden Webapplikation. Deshalb werden Informationen, die im Scope der Webapplikation gelten hier abgelegt, so dass jedes Servlet auf diese zugreifen kann.

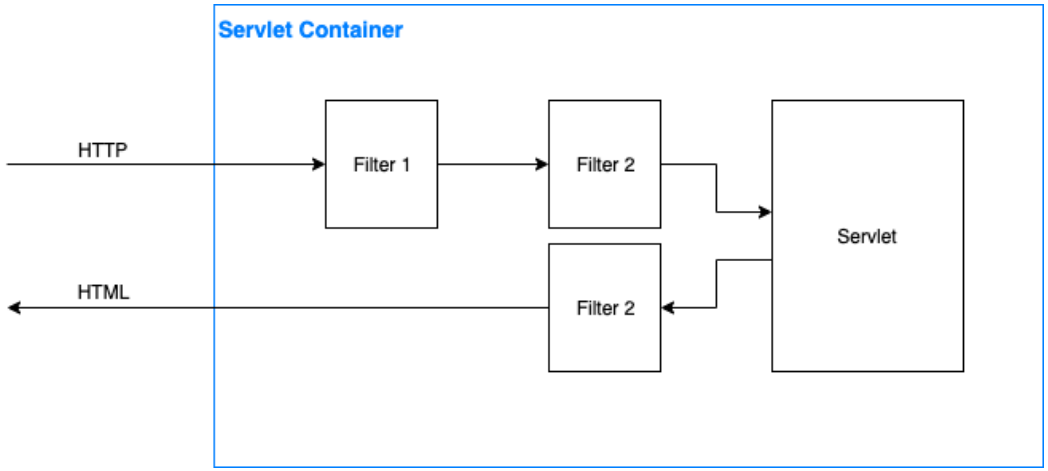
```
|-- src
|-- main
|   |-- java
|   |   |-- ch
|   |       |-- fhnw
|   |           |-- webfr
|   |               |-- flashcard
|   |                   |-- domain
|   |                   |   |-- Questionnaire.java
|   |                   |-- filter
|   |                   |   |-- BasicFilter.java
|   |                   |   |-- TranslationFilter.java
|   |                   |-- listener
|   |                   |   |-- BasicListener.java
|   |                   |-- persistence
|   |                   |   |-- QuestionnaireRepository.java
|   |                   |-- util
|   |                   |   |-- QuestionnaireInitializer.java
|   |                   |-- web
|   |                       |-- BasicServlet.java
|   |-- resources
|   |   |-- log4j.properties
|   |   |-- messages.properties
|   |-- webapp
|       |-- WEB-INF
|       |   |-- web.xml
|       |-- css
|       |   |-- styles.css
|       |-- smiley.png
|-- test
```

1.4 web.xml

Konfigurationsfile für die Webapp. Das File wird beim Hochfahren der Webapp vom Web-Container gelesen, um die Webapp entsprechend zu konfigurieren.

```xml
<?xml version="1.0" encoding="ISO-8859-1" standalone="no"?>
<web-app xmlns="http://java.sun.com/xml/ns/javaee" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
↪  version="3.0"
        xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
        ↪  http://java.sun.com/xml/ns/javaee/web-app_3_0.xsd">

    <display-name>Basic Servlet Web Application</display-name>
    <servlet>
        <servlet-name>Basic Servlet</servlet-name>
        <servlet-class>ch.fhnw.webfr.flashcard.web.BasicServlet</servlet-class>
    </servlet>
    <servlet-mapping>
        <servlet-name>Basic Servlet</servlet-name>
        <url-pattern>/*</url-pattern>
    </servlet-mapping>
    <filter>
        <filter-name>Basic Filter</filter-name>
        <filter-class>ch.fhnw.webfr.flashcard.filter.BasicFilter</filter-class>
    </filter>
    <filter-mapping>
        <filter-name>Basic Filter</filter-name>
        <url-pattern>/*</url-pattern>
    </filter-mapping>
    <listener>
        <listener-class>ch.fhnw.webfr.flashcard.listener.BasicListener</listener-class>
    </listener>
    <context-param>
        <param-name>mode</param-name>
        <param-value>test</param-value>
    </context-param>
</web-app>
```

1.5 Filter



Servlet-Filter bieten eine Möglichkeit auf Request und Response zwischen Client und Servlet zuzugreifen. Dabei können mehrere Filter eine Filterkette bilden. Dabei wird mittels Mapping-Regeln bestimmt, welche Filter für welche Requests wann zuständig sind. Es gibt zahlreiche Möglichkeiten, bei denen der Einsatz eines Filters sinnvoll sein kann. Der einfachste Anwendungsfall ist das Tracing, um zu sehen welche Ressource angesprochen wurde und wie lange die Bereitstellung der Ressource gedauert hat. Weitere typische Anwendungsfälle umfassen die Security bspw. eine Entschlüsselung des Requests und die Verschlüsselung der Response. Filter sind im Deployment-Descriptor deklariert (oder über die Annotation `@WebFilter`). Sie werden vom Container verwaltet und müssen das Interface `javax.servlet.Filter` implementieren. Es gibt drei Methoden, die den Lifecycle bestimmen:

- `init(FilterConfig)` - wird aufgerufen, nachdem der Container die Instanz der Filterklasse erzeugt hat
- `doFilter(ServletRequest, ServletResponse, FilterChain)` - wird bei der Abarbeitung der FilterChain gerufen
- `destroy()` - die Filterinstanz wird gleich beseitig

Interceptor Pattern

Interceptors sollen technische Dienste bereitstellen, die nicht direkt mit der Anwendungslogik zu tun haben.

Chain of Responsibility

Pattern Der Auslöser und der Verarbeiter einer Nachricht werden entkoppelt. Es wird dabei eine Kette von Objekten durchlaufen, welche die Nachricht verarbeiten können. Die Nachricht wird solange weitergereicht, bis ein Objekt diese verarbeitet hat oder das Ende der Kette erreicht ist.

Filter Mapping

Die Reihenfolge der Filter wird im `web.xml` definiert. Die Reihenfolge kann nicht über Annotationen gemacht werden.

```java
public class BasicListener implements ServletContextListener {

    enum Mode {test,prod}

    @Override
    public void contextInitialized(ServletContextEvent sce) {
        ServletContext ctx = sce.getServletContext();
        Object attribute = ctx.getInitParameter("mode");
        try {
            Mode mode = Mode.valueOf(String.valueOf(attribute));
            log.info("Mode: {}", mode.name());
            switch (mode) {
                case test:
                    QuestionnaireRepository questionnaireRepository = new
                    ↪  QuestionnaireInitializer().initRepoWithTestData();
                    ctx.setAttribute(QuestionnaireRepository.class.getName(), questionnaireRepository);
                    log.info("Test Repository initialized and set.");
                    break;
                case prod:
                    // init prod repo
                    break;
                default:
                    log.info("No mode set.");
            }
        } catch (IllegalArgumentException e) {
            log.error("Invalid mode {}", attribute);
        }
    }
}
```

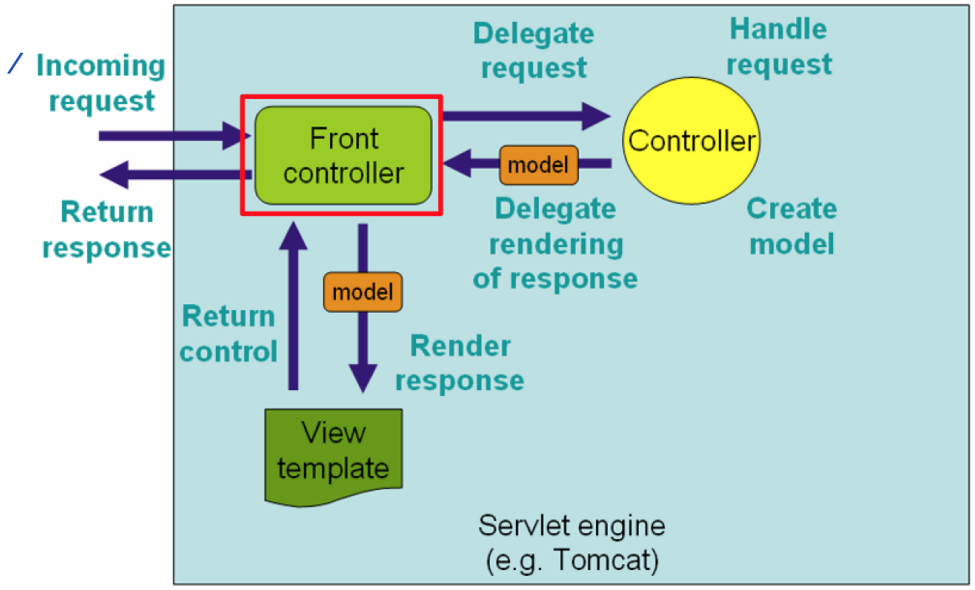
1.6 Response Modification

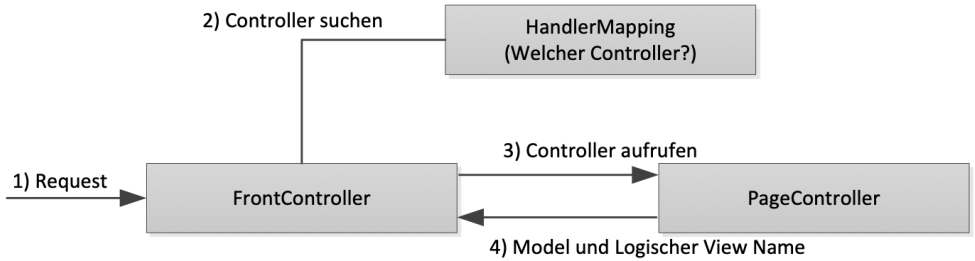
Es ist auf dem Outbound nicht möglich den Original Response direkt zu manipulieren, da mit dem Abschluss der service()-Methode im Servlet der Servlet-Container implizit ein flush() und close() auf den OutputStream aufruft. Somit ist eine Modifikation ausgeschlossen - und deshalb kommen die folgenden Wrapper Klassen in Spiel:

- HttpServletResponseWrapper: Provides a convenient implementation of the HttpServletResponse interface that can be subclassed by developers wishing to adapt the response from a Servlet.
- HttpServletRequestWrapper: Provides a convenient implementation of the HttpServletRequest interface that can be subclassed by developers wishing to adapt the request to a Servlet.

```java
@Slf4j
public class TranslationFilter implements Filter {

    public void doFilter(ServletRequest request, ServletResponse response, FilterChain chain) throws
    ↪  IOException, ServletException {

        log.debug("TranslationFilter doFilter!");

        ServletResponse newResponse = response;

        if (request instanceof HttpServletRequest) {
            newResponse = new CharResponseWrapper((HttpServletResponse) response);
        }

        chain.doFilter(request, newResponse);

        if (newResponse instanceof CharResponseWrapper) {
            String text = newResponse.toString();
            log.debug(text);
            if (text != null) {
                Locale currentLocale =
                ↪  Locale.forLanguageTag(request.getServletContext().getInitParameter("lang"));
                log.info("Language: {}", currentLocale);
                ResourceBundle resourceBundle = ResourceBundle.getBundle("messages", currentLocale);

                Map<String, String> texts = resourceBundle.keySet().stream().collect(Collectors.toMap(k
                ↪  -> k, resourceBundle::getString));
                String translatedText = StrSubstitutor.replace(text, texts, "{", "}");
                log.debug(translatedText);
                response.getWriter().write(translatedText);
                response.setContentLength(translatedText.length());
            }
        }

    }

    @Override
    public void init(FilterConfig fConfig) {
        log.debug("TranslationFilter init!");
    }

    @Override
    public void destroy() {
        log.debug("TranslationFilter destroy!");
    }

    class CharResponseWrapper extends HttpServletResponseWrapper {
        protected CharArrayWriter charWriter;

        protected PrintWriter writer;

        protected boolean getOutputStreamCalled;

        protected boolean getWriterCalled;

        public CharResponseWrapper(HttpServletResponse response) {
            super(response);

            charWriter = new CharArrayWriter();
        }

        public ServletOutputStream getOutputStream() throws IOException {
            if (getWriterCalled) {
                throw new IllegalStateException("getWriter already called");
            }

            getOutputStreamCalled = true;
            return super.getOutputStream();
        }

        public PrintWriter getWriter() throws IOException {
            if (writer != null) {
                return writer;
            }
            if (getOutputStreamCalled) {
                throw new IllegalStateException("getOutputStream already called");
            }
            getWriterCalled = true;
            writer = new PrintWriter(charWriter);
            return writer;
        }

        public String toString() {
            String s = null;

            if (writer != null) {
                s = charWriter.toString();
            }
            return s;
        }
    }

}
```

1.7 MVC





1.8 Front Controller

Das «DispatcherServlet» wird bei jeder SpringMVC Applikation aufgrund der Annotationen `@SpringBootApplication` -> `@EnableAutoConfiguration` automatisch initialisiert. Das ´DispatcherServlet˝ wird per Default auf ” gemappt.

```
@SpringBootApplication
@Slf4j
public class FlashcardApplication implements ApplicationRunner {

    @Autowired
    private QuestionnaireRepository questionnaireRepository;

    public static void main(String[] args) {
        SpringApplication.run(FlashcardApplication.class, args);
    }
}
```

```
@Controller
@RequestMapping("/questionnaires")
@Slf4j
public class QuestionnaireController {

    @Autowired
    private QuestionnaireRepository questionnaireRepository;

    @DeleteMapping("{id}")
    public ModelAndView delete(@PathVariable String id) {
        Optional<Questionnaire> questionnaire = questionnaireRepository.findById(id);
        if (questionnaire.isPresent()) {
            questionnaireRepository.deleteById(id);
            return new ModelAndView("redirect:/questionnaires");
        } else {
            return new ModelAndView("404");
        }
    }

    @PostMapping()
    public ModelAndView create(@Valid Questionnaire questionnaire, BindingResult result) {
        if (result.hasErrors()) {
            log.error("Validation Errors: {}", result.getAllErrors());
            ModelAndView modelAndView = new ModelAndView("create");
            modelAndView.addObject("questionnaire", questionnaire);
            return modelAndView;
        }
        log.debug("Create Questionnaire: {}", questionnaire);
        questionnaireRepository.save(questionnaire);
        log.debug("Questionnaire persisted: {}", questionnaire);
        return new ModelAndView("redirect:/questionnaires");
    }

    @GetMapping(params = "form")
    public ModelAndView getCreateForm() {
        ModelAndView modelAndView = new ModelAndView("create");
        modelAndView.addObject("questionnaire", new Questionnaire());
        log.debug("Call view 'create'");
        return modelAndView;
    }

    @GetMapping("update/{id}")
    public ModelAndView getUpdateForm(@PathVariable String id) {
        Optional<Questionnaire> questionnaire = questionnaireRepository.findById(id);
        if (!questionnaire.isPresent()) {
            return new ModelAndView("404");
        }
        ModelAndView modelAndView = new ModelAndView("update");
        modelAndView.addObject("questionnaire", questionnaire.get());
        log.debug("Call view 'update'");
        return modelAndView;
    }

    @PutMapping()
    public ModelAndView update(@Valid Questionnaire questionnaire, BindingResult result) {
        if (result.hasErrors()) {
            log.error("Validation Errors: {}", result.getAllErrors());
            ModelAndView modelAndView = new ModelAndView("update");
            modelAndView.addObject("questionnaire", questionnaire);
            return modelAndView;
        }
        log.debug("Update Questionnaire: {}", questionnaire);
        questionnaireRepository.save(questionnaire);
        log.debug("Questionnaire persisted: {}", questionnaire);
        return new ModelAndView("redirect:/questionnaires");
    }
```

```
    @GetMapping
    public String findAll(Model model) {
        List<Questionnaire> questionnaires = questionnaireRepository.findAll();
        log.debug("{} questionnaires found", questionnaires.size());
        model.addAttribute("questionnaires", questionnaires);
        return "list";
    }

    @GetMapping(value = "/{id}")
    public String findById(@PathVariable String id, Model model) {
        log.debug("find questionnaire by id {}", id);
        Optional<Questionnaire> questionnaire = questionnaireRepository.findById(id);
        if (questionnaire.isPresent()) {
            model.addAttribute("questionnaire", questionnaire.get());
            log.debug("found questionnaire {}", id);
            return "show";
        } else {
            log.debug("no questionnaire found");
            return "404";
        }
    }
}
```

2 Tiny Url

2.1 TinyUrlController

```
@Controller
public class TinyUrlController {
    private static final Log logger = LogFactory.getLog("BUMI");

    @Autowired
    private TinyUrlRepository tinyUrlRepository;

    @GetMapping(value = "/urls", params = "form")
    public String createForm(Model model) {
        model.addAttribute("urlentry", new UrlEntry());
        return "create";
    }

    @RequestMapping(value = "/{tinyUrl}", method = RequestMethod.GET)
    public String redirectToTinyUrl(@PathVariable String tinyUrl, Model model) {
        Optional<UrlEntry> urlEntry = tinyUrlRepository.findByTinyUrl(tinyUrl);
        if(urlEntry.isPresent()){
            return "redirect:" + urlEntry.get().getOriginalUrl();
        }
        return "404";
    }

    @RequestMapping(value="/urls", method = RequestMethod.POST)
    public String create(@Valid UrlEntry urlEntry, BindingResult result, Model model) {
        if (result.hasErrors()) {
            result.getAllErrors().forEach(err -> logger.error(err.getDefaultMessage()));
            model.addAttribute("text", "Result has errors!");
            return "error";
        }
        if(tinyUrlRepository.findByTinyUrl(urlEntry.getTinyUrl()).isPresent()){
            return "redirect:409"; //Need redirect to get out of POST
        };


        tinyUrlRepository.save(urlEntry);
        return "redirect:/urls?form";
    }
}
```

2.2 AdminController

```
@Controller
@RequestMapping("/admin")
public class AdminController {
    @Autowired
    private TinyUrlRepository tinyUrlRepository;

    @GetMapping
    public String adminUi(Model model){
        List<UrlEntry> list = tinyUrlRepository.findAll();
        model.addAttribute("filterString", "");
        model.addAttribute("tinyUrls", list);
        return "/tinyUrl/list";

    }

    @GetMapping(params = "filterString")
    public String filter(@RequestParam String filterString, Model model){
        List<UrlEntry> list = tinyUrlRepository.findByTinyUrlLike(filterString);
        model.addAttribute("filterString", filterString);
        model.addAttribute("tinyUrls", list);
        return "/tinyUrl/list";
    }

    @DeleteMapping(path = "/{tinyUrl}")
    public String delete(@PathVariable String tinyUrl){
        Optional<UrlEntry> url  = tinyUrlRepository.findByTinyUrl(tinyUrl);
        if (url.isPresent()) {
            tinyUrlRepository.delete(url.get());
        } else {
            return "redirect:404";
        }
    return "redirect:/admin";
    }
}
```

3 Single Page Application

3.1 Klassische Webapplikation

Vorteile:

- Full Control
- Sicherheit

Nachteile:

- User Experience
- Server-Roundtrip

3.2 Klassische Webapplikation

Vorteile:

- User Feedback

- Client- vs. Server-Logik

Nachteile:

- Browser-Unterstützung
- Verschiedene Technologien

3.3 Definition Single Page Application (SPA)

Eine Single Page Application ist eine Webanwendung, die keinen Seitenwechsel (Roundtrip) durchführt, sondern die Anzeige nur durch Austausch von Seitenelementen via Javascript/DOM verändert. Es gibt dabei also keine serverseitige Seitennavigation. Die URL ändert sich grundsätzlich nicht (kann aber simuliert werden!). Initial wird eine komplette Seite oder zumindest das Grundgerüst einer Webseite vom Server geladen. Die Seite lädt anschliessend Daten über Webservices (meist REST-basierte Dienste) nach und erzeugt die Darstellung clientseitig (clientseitiges Rendern). Eine Single Page Application wirkt damit wie eine Desktopanwendung.

4 React

- für die Implementation von grossen Applikationen
- ist eine JavaScript Bibliothek für den View einer SPA
- reduziert die Applikationsentwicklung auf reines JavaScript
- basiert auf JavaScript-Komponenten
- führt einen Virtuellen DOM
- minimiert dadurch die Manipulationen des DOMs im Browser
- nutzt keine HTML-Templates
- führt JSX als Erweiterung von JavaScript ein
- nutzt Properties um read-only Eigenschaften zu definieren
- nutzt State um den Zustand einer Komponenten zu beschreiben
- nutzt die Methode render() der Komponente, um die Komponente zu zeichnen
- ruft bei jeder Änderung des State die Methode render() automatisch auf

4.1 Komponenten

Komponenten erlauben eine Benutzeroberfläche in unabhängige, wiederverwendbare Teile aufzubrechen und jeden als isoliert zu betrachten.

Nutzen: Wird an einer Komponente etwas geändert, müssen die übrigen Komponenten nicht getestet werden, da diese nicht von den Änderungen betroffen sein können.

Reacts zentraler und einziger Baustein sind Komponenten.

Eine React Komponente muss immer eine render() Methode haben.

4.2 Properties einer Komponente

Properties werden von aussen an die Komponente bergeben (analog Input Parameter einer Methode). Innerhalb der Komponente sind Properties nicht veränderbar.

4.3 State einer Komponente

Eine React-Komponente kann einen Zustand (State) besitzen, der sich zur Laufzeit ändert. Der State beschreibt die veränderlichen Daten der Komponente. Wichtig: Nur die Komponente selber kann seinen State mit setState(...) verändern. Eine Änderung am State löst automatisch einen Call auf die Methode render() aus. Die Komponente und ihre Kinder werden neu gezeichnet.

4.4 Lifecylce einer Komponente

Die Methode "componentDidMount()"wird con React aufgerufen, sobald die Komponente an den virtuellen DOM-Tree eingebunden ist.

Die Methode "componentWillUnmount()"wird aufgerufen kurz bevor die Komponente aus dem DOM-Tree entfern und gelöscht wird.

4.5 Functional Component

React Component
- Komponente ist eine Klasse
- Stateful

React Functional Component
- Komponente ist eine Funktion
- Stateless
- Kein this Keyword
- Weniger Code
- Destructuring

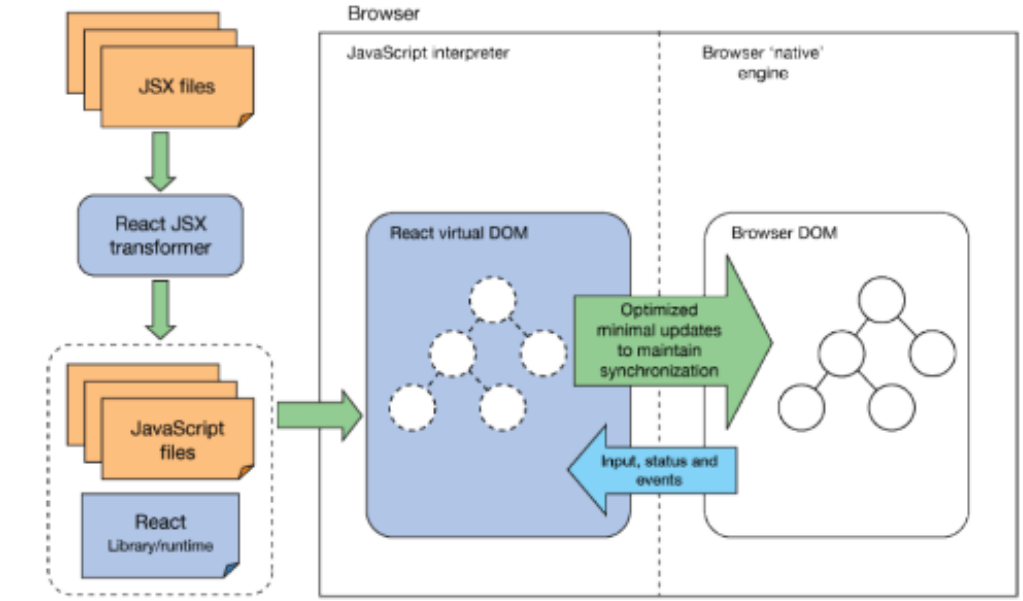
```
class App extends Component {

    constructor(props) {
        super(props);

        this.state = {
            counter: 0
        }
    }

    componentDidMount() {
        this.timer = setInterval(this.increment, 1000);
    }

    componentWillUnmount() {
        clearInterval(this.timer);
    }

    increment = () => {
        this.setState({counter: this.state.counter + 1});
    }
    reset = () => {
        this.setState({counter: 0});
    }

    render() {
        return (
            <div className="container">
                <div>
                    <Jumbotron>
                        <h1>{this.state.counter}</h1>
                    </Jumbotron>
                </div>
                <div>
                    <Button onClick={this.reset}>RE-FRÄSCH</Button>
                </div>
            </div>

        )
    }

}
```

5 Virtueller DOM

Die Manipulation des Document Object Models (DOM) im Browser ist teuer und deshalb langsam. Jede Änderung an dieser Baumstruktur quittiert der Browser mit teurer Neuberechnung seiner Geometrie. Je mehr geändert wird, desto länger dauert es. Je weniger man den DOM des Browsers verändert, desto schneller ist die Applikation.

JavaScript selber ist performant: Viele JavaScript-Frameworks suchen einen Kompromiss mittels Verwendung von altbewährten Entwurfsmustern, um die Komplexität zwischen Einfachheit und Performance zu bändigen. (Two-way binding, dirty-checking, etc.) React versucht es mit einem extremen Ansatz, der in erster Linie die Einfachheit und nicht die Performanz in den Fokus stellt. React rendert bei jedem Update einfach alles neu!

Virtueller DOM: Mit React Komponenten und JSX arbeitet man nicht direkt mit dem DOM des Browsers, sondern mit normalen JavaScript Objekten (–> Virtueller DOM), die schnell gelesen und bearbeitet werden können, ohne dass damit tatsächliche Änderungen am DOM ausgelöst werden. Bei jeder Änderung der Daten erstellt React einen neuen virtuellen DOM. Ein stark optimierter und heuristischer Algorithmus vergleicht diesen neuen Baum mit den vorherigen und errechnet eine Liste von minimalen Änderungen am richtigen DOM aus. Diese werden gesammelt und nicht direkt, sondern im Batch (=Stapel) an den Browser weitergeleitet.



Reconciliation beschreibt den Mechanismus, der die Zustandsänderungen in einer React-Komponente beobachtet. einen aktualisierten Zustand auf den Bildschirm schreibt.

Fragen zum virtuellen DOM

- Wann wird eine Reconciliation-Phase ausgelöst?
- Welche Änderungen im Component Tree sind "teueründ welche "billig"?
- Was ist die Funktion eines "keyïn einer Liste?
- Welche Schritte in der Reconciliation-Phase werden bei der React-App aus Übung 6 konkret ausgeführt?

6 Hooks

```jsx
import React, { Component } from 'react'
import { Button, Jumbotron } from 'reactstrap'

class App extends Component {
    constructor(props) {
        super(props)
        this.state = { counter: 0 }
    }

    componentDidMount() {
        this.timerId = setInterval(this.incr, 1000)
    }

    componentWillUnmount() {
        clearInterval(this.timerId)
    }

    incr = () => {
        this.setState({ counter: this.state.counter + 1 })
    }

    refresh = () => {
        this.setState({ counter: 0 })
    }

    render() {
        return <div>
            <Jumbotron><h1>{ this.state.counter }</h1></Jumbotron>
            <Button onClick={ this.refresh }>REFRESH</Button>
        </div>
    }
}
export default App
```

```jsx
import React, { useState, useEffect } from 'react'
import { Button, Jumbotron } from 'reactstrap'

const App = () => {
    const [counter, setCounter] = useState(0)

    useEffect(() => {
        const timerId = setInterval(incr, 1000)
        return () => { clearInterval(timerId) }
    }, [])

    const incr = () => {
        // setCounter(counter + 1)
        setCounter(prevCounter => prevCounter + 1)
    }

    const refresh = () => {
        setCounter(0)
    }

    return <div>
            <Jumbotron><h1>{ counter }</h1></Jumbotron>
            <Button onClick={ refresh }>REFRESH</Button>
        </div>
}

export default App
```

`setInterval` wie in `componentDidMount`

`clearInterval` wie in `componentWillUnmount`

Statt `setCounter(counter + 1)` damit keine Abhängigkeit von counter

6.1 Hooks Regeln

- Hooks dürfen nur vom Top-Level der Funktionalen Komponente aufgerufen werden.

- Die Reihenfolge von Hook-Aufrufen muss immer gleich bleiben.

7 REST

7.1 RESTFullness

- Stufe 0 - RESTIess -> kein REST Support
- Stufe 1 - Ressourcen -> Jede identifizierbare Ressource hat eine eigene URI.
- Stufe 2 - HTTP Verben -> GET, POST, HEAD, PUT, DELETE und OPTIONS werden gemäss ihrer Spezifikation unterstützt.
- Stufe 3 – Hypermedia -> Verknüpfung von Ressourcen und Repräsentationen durch Hyperlinks.

7.2 Cross-Origin Resource Sharing (CORS)

Cross-Origin Resource Sharing (CORS) CORS ist ein Mechanismus, um Webbrowsern oder auch anderen Webclients Cross-Origin-Requests zu ermöglichen. Zugriffe dieser Art sind normalerweise durch die Same-Origin-Policy untersagt. CORS ist ein Kompromiss zugunsten grösserer Flexibilität im Internet unter Berücksichtigung möglichst hoher Sicherheitsmassnahmen. CORS definiert eine Reihe von neuen Headern im HTTP- Protokoll, die es dem Server erlauben gezielt REST-Aufrufe für bestimmte Domains zuzulassen.

Preflight Request

Durch den Preflight Request werden die CORS (Cross-Origin Resource Sharing)-Regeln für den Serverdienst abgefragt, bevor die tatsächliche Anforderung gesendet wird.

- Ein Webbrowser oder ein anderer Benutzer-Agent sendet eine Preflight- Anforderung, die die Ursprungsdomäne, die Methode und die Header der tatsächlichen Anforderung umfasst.
- Wenn CORS für den Serverdienst aktiviert wird, dann wertet der Serverdienst die Preflight-Anforderung anhand der CORS-Regeln aus und akzeptiert die Anforderung bzw. weist sie zurück.

7.3 Async Kommunikation

Synchrone Kommunikation

Der Client wartet nach Absenden der Anfrage an den Server so lange, bis er eine Rückantwort erhält, und kann dann erst andere Aktivitäten ausführen.

Asynchrone Kommunikation

Der Client sendet die Anfrage an den Server und arbeitet sofort weiter. Beim Eintreffen der Rückantwort werden dann bestimmte Routinen beim Client aufgerufen. Die durch den Server initiierten Rückrufe (Callbacks) erfordern zusätzliche Kontrolle beim Client, wenn ungewünschte Unterbrechungen der Arbeit vermieden werden sollen.

Vorteile

- Sendender Prozess kann weiterarbeiten, noch während die Nachricht übertragen wird.
- Es kann ein höherer Grad an Parallelität erreicht werden.
- Die Entkoppelung von Sender und Empfänger ist stärker, d.h. grösser Unabhängigkeit beider Komponenten.

Nachteile

- Der Sender weiss nicht, ob / wann die Nachricht angekommen ist. Das Debugging der Anwendung wird anspruchsvoller.
- Betriebssystem muss Puffer verwalten.

7.4 Event Loop

JavaScript basiert auf einem Event Loop

- ”queue.waitForMessage()” wartet synchron auf eine Message
- Sobald Message vorhanden, wird die entsprechende Callback-Funktion auf den Stack gelegt und aufgerufen.
- Die Callback-Funktion wird komplett abgearbeitet.
- Erst wenn Callback-Funktion beendet ist, kann der der Event-Loop die Queue auf die nächste Message testen.

EventLoop ist Single-Threaded
Vorteile

- Einfach, vor allem für die GUI-Programmierung
- Event und Rendering werden sequentiell ausgeführt
- Keine synchronisierung der Threads notwendig
- Typisches Programmiermodell für GUI-Entwicklung (z.B. auch in Android)

Nachteile:

- Aufwändige, zeitintensive Aufgaben können GUI blockieren, dadurch schlechtes User Erlebnis

Darum sollten zeitintensive Aufgaben asynchron programmiert werden (Ajax: Asynchronous JavaScript and XML)

7.5 Rest Controller

```
@CrossOrigin
@RestController
@RequestMapping("/questionnaires")
@RequiredArgsConstructor
@Slf4j
public class QuestionnaireController {

    private final QuestionnaireRepository questionnaireRepository;

    @GetMapping
    public ResponseEntity<List<Questionnaire>> findAll() {
        Sort sort = Sort.by(Sort.Direction.ASC, "title");
        return new ResponseEntity<>(questionnaireRepository.findAll(sort), HttpStatus.OK);
    }

    @GetMapping("/{id}")
    public ResponseEntity findById(@PathVariable String id) {
        log.info("Find by id {}", id);
        Optional<Questionnaire> questionnaire = questionnaireRepository.findById(id);
        if (questionnaire.isPresent()) {
            log.info("Find a Questionnaire for id {}", id);
            return new ResponseEntity(questionnaire.get(), HttpStatus.OK);
        }
        log.error("No Questionnaire found for id {}", id);
        return new ResponseEntity<>(HttpStatus.NOT_FOUND);
    }

    @PostMapping
    public ResponseEntity<Questionnaire> create(@Valid @RequestBody Questionnaire questionnaire,
    ↪  BindingResult result) {
        if (result.hasErrors()) {
            return new ResponseEntity<>(HttpStatus.PRECONDITION_FAILED);
        }
        questionnaire = questionnaireRepository.save(questionnaire);
        log.debug("Created questionnaire with id=" + questionnaire.getId());
        return new ResponseEntity<>(questionnaire, HttpStatus.CREATED);
```

```
    }

    @PutMapping(value = "/{id}")
    public ResponseEntity<Questionnaire> update(@PathVariable String id,
                                                @Valid @RequestBody Questionnaire questionnaire,
                                                ↪  BindingResult result) {
        if (result.hasErrors()) {
            return new ResponseEntity<>(HttpStatus.PRECONDITION_FAILED);
        }
        Optional<Questionnaire> questionnaireOptional = questionnaireRepository.findById(id);
        if (questionnaireOptional.isPresent()) {
            Questionnaire updateQuestionnaire = questionnaireOptional.get();
            if (questionnaire.getTitle() != null) {
                updateQuestionnaire.setTitle(questionnaire.getTitle());
            }
            if (questionnaire.getDescription() != null) {
                updateQuestionnaire.setDescription(questionnaire.getDescription());
            }
            questionnaire = questionnaireRepository.save(updateQuestionnaire);
            log.debug("Updated questionnaire with id=" + updateQuestionnaire.getId());
            return new ResponseEntity<Questionnaire>(updateQuestionnaire, HttpStatus.OK);
        }
        log.debug("No questionnaire with id=" + id + " found");
        return new ResponseEntity<>(HttpStatus.NOT_FOUND);
    }

    @DeleteMapping(value = "/{id}")
    public ResponseEntity<String> delete(@PathVariable String id) {
        Optional<Questionnaire> questionnaireOptional = questionnaireRepository.findById(id);
        if (questionnaireOptional.isPresent()) {
            questionnaireRepository.deleteById(id);
            log.debug("Deleted questionnaire with id=" + id);
            return new ResponseEntity<>(HttpStatus.OK);
        }
        log.debug("No questionnaire with id=" + id + " found");
        return new ResponseEntity<>(HttpStatus.NOT_FOUND);
    }
}
```

8 Konfiguration

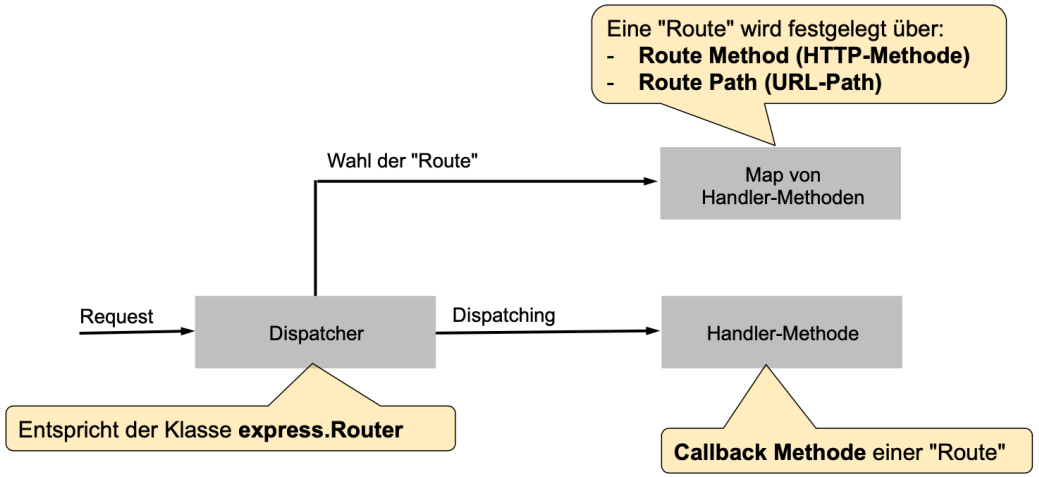
Konfiguration über die Environment ".env": Die Environment wird in spezifischen Files festgelegt, wie ".env", ".env.development", ".env.production". Diese Files werden in unterschiedlichen Entwicklungsphasen von Tools wie webpack (automatisch vom Tool "Create React Appïntegriert) gelesen – aber immer bei einem Build der Applikation. Das bedeutet, dass zur Laufzeit keine Aktualisierung der Konfigurationseinstellungen möglich ist. Beachten muss man auch die Vorgaben von React für die Namensgebung der Variablen (siehe https://create-react-app.dev/docs/adding-custom-environment-variables)

Konfiguration über ein JavaScript-File: Um die Konfiguration zur Laufzeit zu ändern, muss die App beim Laden entsprechende Informationen und Anweisungen ausführen. Bei einer SPA-Applikation wird das File ïndex.html"geladen und verarbeitet. Ein Eintrag <script src="config.js/> führt z.B. zum Ausführen des Skripts "config.jsünd man kann z.B. eine entsprechende globalen Variable setzen. Alle globalen Objekte werden in eine Browserapplikation automatisch an die "windowInstanz angehängt. Diese Variante hat jedoch in Bezug auf die Sicherheit seine berechtigten Nachteile, da einfach ausführbarer Code in die Applikationen integriert werden kann!

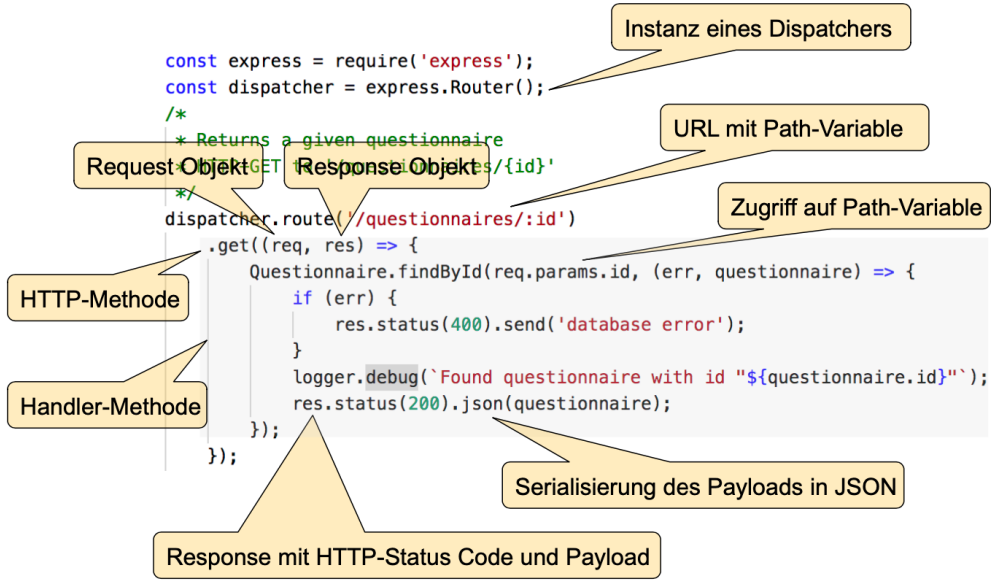
Konfiguration über ein JSON-File: Die Konfigurationseinstellungen werden am einfachsten in JSON formuliert. Dieses Format kann mit JavaScript einfach verarbeitet werden. Das Konfigurationsfile, z.B. äpplication.json"wird auf dem Server liegen, wo die Einstellungen vorgenommen werden können. Beim Starten der Client Applikation im Browser muss dieses Konfigurationsfile von der App geladen werden, um die entsprechenden Einstellungen zu setzen. Das bedeutet, dass bei der Initialisierung der App entsprechende Logik ausprogrammiert werden muss.

9 Node Express

9.1 Request Handling



9.2 Beispiel



app.js

```
"use strict"
```

```
const log4js = require('log4js')
const dotenv = require('dotenv-extended')
const express = require('express')
const cors = require('cors')
const bodyParser = require('body-parser')
const mongoose = require('mongoose')
const routes = require('./web/dispatcher')


// Read the properties from file '.env' and '.env.defaults'
dotenv.load({ silent: true })

// Configure log4js based on the definitions in file 'log4js.json'
log4js.configure('log4js.json')
const logger = log4js.getLogger('app')

// Use native promises, needed with Mongoose since v4.1.0
mongoose.Promise = global.Promise

// Connect to the database using the connection parameters found in the property-files
const url = 'mongodb://' + process.env.MONGO_HOST + '/' + process.env.MONGO_DATABASE
logger.debug(`Database URL used '%s'`, url)
mongoose.connect(url, { useNewUrlParser: true, useUnifiedTopology: true })

// Create the Express Server App
const app = express()

// Configure body-parser. The parser handles the JSON payload.
app.use(bodyParser.json())

// Enable CORS (for all requests)
app.use(cors())

// Example how to modify the http response (see HttServletResponseWrapper in Java)
const modifyResponseBody = (req, res, next) => {
    var origSend = res.send
    res.send = body => {
        // arguments[0] (or `body`) contains the response body
        body = "modified: " + body
        origSend.apply(res, [body])
    }
    next()
}
// app.use(modifyResponseBody)

// Configure the dispatcher with all its controllers
app.use('/flashcard-express', routes)


// Read PORT from the configuration, default to 9090
const PORT = process.env.PORT || 9090

// Start the App as HTTP server
app.listen(PORT)

// Use backquotes for the es6 feature
logger.info(`Server started on port ${PORT}`)

module.exports = app
```

9.3 dispatcher.js

```
"use strict";

const dispatcher = require('express').Router();

const index_controller = require('./index-controller')
const questionnaire_controller = require('./questionnaire-controller')

dispatcher.route('/').get(index_controller.index)
dispatcher.route('/questionnaires').get(questionnaire_controller.findAll)
dispatcher.route('/questionnaires/:id').get(questionnaire_controller.findById)
dispatcher.route('/questionnaires').post(questionnaire_controller.create)
dispatcher.route('/questionnaires/:id').put(questionnaire_controller.update)
dispatcher.route('/questionnaires/:id').delete(questionnaire_controller.delete)

module.exports = dispatcher;
```

9.4 questionnaire-controller.js

```
"use strict"

const log4js = require('log4js')
const Questionnaire = require('../domain/questionnaire')

// Create a logger
const logger = log4js.getLogger('controller')

/*
 * Returns all questionnaires
 * HTTP-GET to '/questionnaires'
 */
exports.findAll = (req, res) => {
    Questionnaire.find((err, questionnaires) => {
        if (err) {
            return res.status(400).send('database error')
        }
        logger.debug(`Found ${questionnaires.length} questionnaires`)
        res.status(200).json(questionnaires)
    })
}

/*
 * Returns a given questionnaire
 * HTTP-GET to '/questionnaires/{id}'
 */
exports.findById = (req, res) => {
    Questionnaire.findById(req.params.id, (err, questionnaire) => {
        if (err) {
            return res.status(400).send('database error')
        }
        logger.debug(`Found questionnaire with id "${questionnaire.id}"`)
        res.status(200).json(questionnaire)
    })
}

/*
 * Creates a new questionnaire
 * HTTP-POST to '/questionnaires'
 */
exports.create = (req, res) => {
    // Create a new instance of the Questionnaire model
    let questionnaire = new Questionnaire()
    questionnaire.title = req.body.title
    questionnaire.description = req.body.description

    // Save the questionnaire and check for errors
    questionnaire.save((err, questionnaireCreated) => {
        if (err) {
            logger.error(`Could not create new questionnaire`)
            res.status(412).send('database error')
        } else {
```

```
            logger.debug(`Successfully created questionnaire with id "${questionnaire.id}"`)
            res.status(201).json(questionnaireCreated)
        }
    })
}

/*
 * Updates a given questionnaire
 * HTTP-PUT to to '/questionnaires/{id}'
 */
exports.update = (req, res) => {
    Questionnaire.findById(req.params.id, (err, questionnaire) => {
        if (err) {
            logger.error(`Could not update questionnaire with id "${req.params.id}"`)
            return res.status(404).send('database error')
        }
        questionnaire.title = req.body.title
        questionnaire.description = req.body.description

        // Update the questionnaire and check for errors
        questionnaire.save(err => {
            if (err) {
                return res.status(404).send('database error')
            }
            logger.debug(`Successfully updated questionnaire with id "${questionnaire.id}"`)
            res.status(200).json(questionnaire)
        })
    })
}

/*
 * Deletes a given questionnaire
 * HTTP-DELETE to '/questionnaires/{id}'
 */
exports.delete = (req, res) => {
    Questionnaire.deleteOne({ _id: req.params.id }, err => {
        if (err) {
            logger.error(`Could not delete questionnaire with id "${req.params.id}"`)
            return res.status(404).send('database error')
        }
        logger.debug(`Successfully deleted questionnaire with id "${req.params.id}"`)
        res.status(200).send()
    })
}
```

10 Middleware

Middlewarefunktionen sind Funktionen, die Zugriff auf das Request, das Response-Objekt und die nächste Middlewarefunktion im Request/Response-Cycle haben. Die nächste Middlewarefunktion wird im Allgemeinen durch die Variable **next** bezeichnet. Über Middlewarefunktionen lassen sich folgende Tasks ausführen:

- Ausführen von Code
- Vornehmen von Änderungen Request- und Response-Objekt
- Beenden des Request/Response-Cycle
- Aufrufen der nächsten Middleware im Stack

10.1 JSON

Die Handler-Methoden erwarten JSON, deshalb muss aus dem Request-Payload zuerst ein JSON-Objekt generiert werden.

```
app.use(bodyParser.json());
```

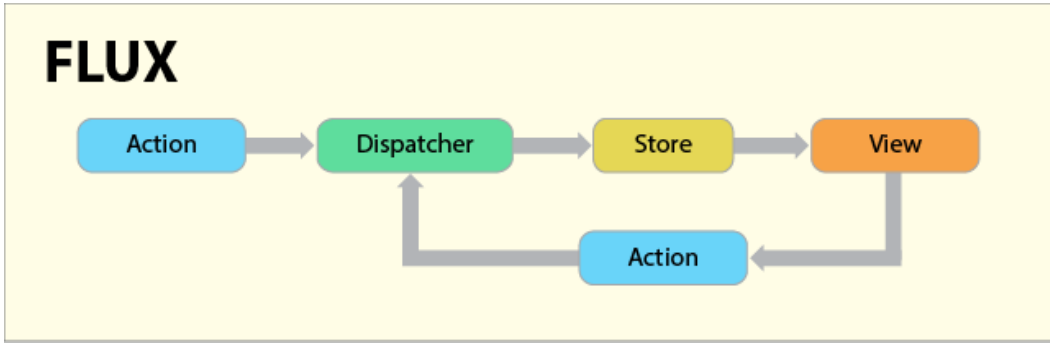
10.2 CORS

```
const cors = require('cors');
app.use(cors());
```

11 Redux

11.1 Flux

In klassischen Architekturen wie MVC und MVVM ist der State in den entsprechenden Models verteilt. Der State ist innerhalb der Applikation stark fragmentiert. Als Antwort zur Fragmentierung entstand ein neuer Trend in SPA hin zu den One-Way Data Flow Patterns. Daten fliessen in Form von Nachrichten in einer Richtung durch die Applikation. Der State wird zentral im Store verwaltet. Ein Vorreiter der One-Way Data Flow Pattern ist Flux von Facebook. Innerhalb von Flux kann es beliebig viele Stores, Actions und Views geben, jedoch nur einen Dispatcher.



11.2 Redux

Redux setzt das One-Way Data Flow Patterns um. Eine Redux-Applikation kennt genau einen einzigen Store. Im Store wird der komplette State der Anwendung zentral gespeichert wird. Aktionen stellen (wie bei Flux) die einzige Möglichkeit dar, den State im Store zu verändern. Um zu definieren, wie genau diese Änderungen funktionieren, gibt es das Konzept der Reducer. Reducers sind als sogenannte "Pure Functions"realisiert, JavaScript-Funktionen ohne Seiteneffekte => funktionale Programmierung.

Die 3 Kernprinzipien von Redux

Single source of truth Der Zustand der Anwendung wird zentral über einen einzigen Store gemanagt.

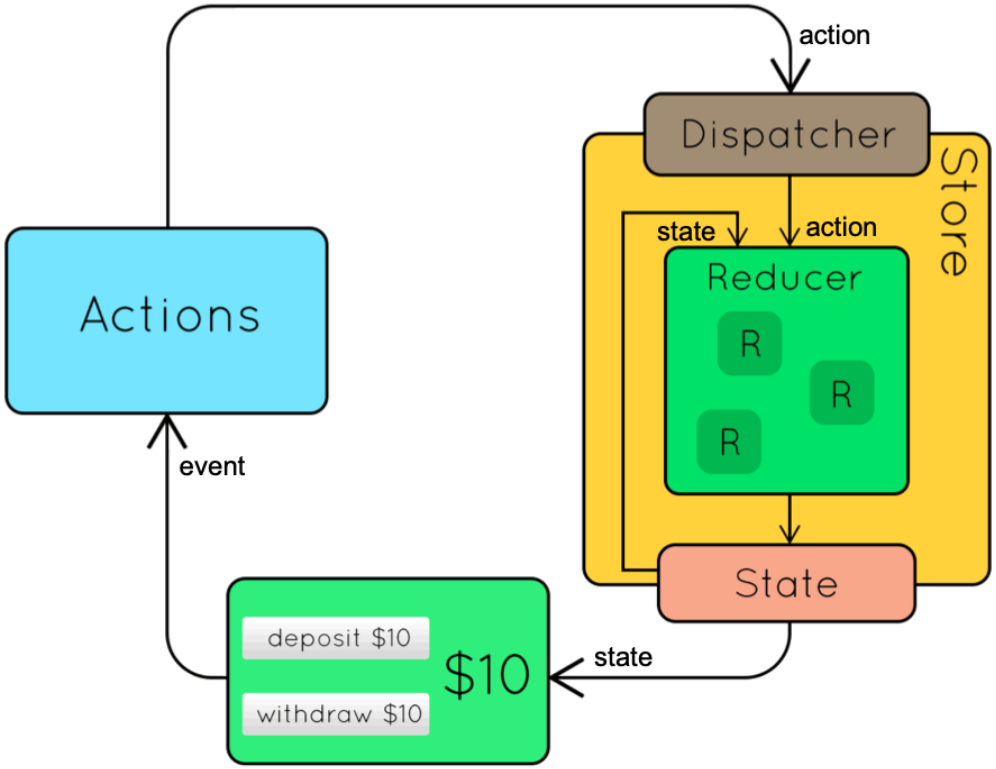
State is read-only Die Anwendung reagiert auf Benutzerinteraktionen, indem sie mit einem ActionCreator eine Action erzeugt, die anzeigt, was gerade passiert ist. Mit einem Reducer wird dann abhängig von der Action aus dem alten Zustand ein neuer Zustand erzeugt.

Changes are made with pure functions Reducer sind seiteneffektfreie Funktionen. Sie erhalten zwei Eingabeparameter (den derzeitigen Anwendungszustand sowie die Action, die gerade ausgelöst wurde) und erzeugen daraus eine Ausgabe (den neuen Anwendungszustand).

Schlüsselkonzepte

- Alle Applikationsdaten werden über eine einzige Datenstruktur, genannt State, im Store verwaltet.
- Die Applikation liest den State immer aus dem Store.
- Der State wird nie ausserhalb des Stores verändert.
- Ein View kann eine Action auslösen, die beschreibt was passieren soll.
- Der neue State wird in sogenannten Reducer aus dem alte State und der Action neu bestimmt.

Datenfluss



Reducers

In einer grossen Applikation müssen sehr viele unterschiedliche Aspekte als State im Store verwaltet werden. Deshalb ist es notwendig, dass das Konzept "Reducerßkaliert, so dass ein Reducer überblickbar bleibt. Eine Aufteilung in mehrere Reducers wird von Redux unterstützt, wobei jeder Reducer für die Verwaltung eines konkreten State- Aspekt verantwortlich ist. In Redux müssen diese Reducers schlussendlich zusammengefasst werden mit: redux.combineReducers(reducer1, reducer2, ...)

11.3 Code

```
const redux = require('redux')
const logFactory = require('redux-logger')
const thunk = require('redux-thunk').default

const logger = logFactory.createLogger({
    colors: false
});

function counterReducer(state = 0, action) {
    switch (action.type) {
        case 'INCREMENT':
            return state.counter + 1
        case 'DECREMENT':
            return state.counter - 1
        default:
            return state
    }
}


function waitingReducer(state = false, action) {
    switch (action.type) {
        case 'WAITING':
            return true
        case 'RUNNING':
            return false
        default:
            return state
    }
}

function incrementAsync() {
    return (dispatch) => {
        dispatch({type: 'WAITING'})
        setTimeout(() => {
            dispatch({type: 'RUNNING'})
            dispatch({type: 'INCREMENT'})
        }, 2000);
    }
}

const reducers = redux.combineReducers({ counter: counterReducer,
    waiting: waitingReducer
})

let store = redux.createStore(reducers, redux.applyMiddleware(thunk, logger));

store.subscribe(() => console.log("The actual state counter is " + store.getState()))

store.dispatch({type: 'INCREMENT'})
store.dispatch(incrementAsync())
store.dispatch({type: 'INCREMENT'})
store.dispatch({type: 'DECREMENT'})
```

11.4 Beispiel Filme

index.js

```
const ACTIONS = {
    'UPDATE_FILTER_TERM': (state, action) => ({ ...state, filterTerm: action.filterTerm })
}

const reducer = (state, action) => _.get(ACTIONS, action.type, _.identity)(state, action)

const initialState = {
    movies: [
        {rank: 1, title: 'The Shawshank Redemption', director: 'Frank Darabont', year: 1994},
        {rank: 2, title: 'The Godfather', director: 'Francis Ford Coppola', year: 1972},
        {rank: 3, title: 'The Dark Knight', director: 'Christopher Nolan', year: 2008},
        {rank: 4, title: 'The Godfather: Part II', director: 'Francis Ford Coppola', year: 1974},
        {rank: 5, title: 'The Lord of the Rings: The Return of the King', director: 'Peter Jackson', year: 200
        {rank: 6, title: 'Pulp Fiction', director: 'Quentin Tarantino', year: 1994},
        {rank: 7, title: 'Schindlers List', director: 'Steven Spielberg', year: 1993},
        {rank: 8, title: '12 Angry Men', director: 'Sidney Lumet', year: 1957},
        {rank: 9, title: 'Fight Club', director: 'David Fincher', year: 1999}
    ],
    filterTerm: ''
}
```

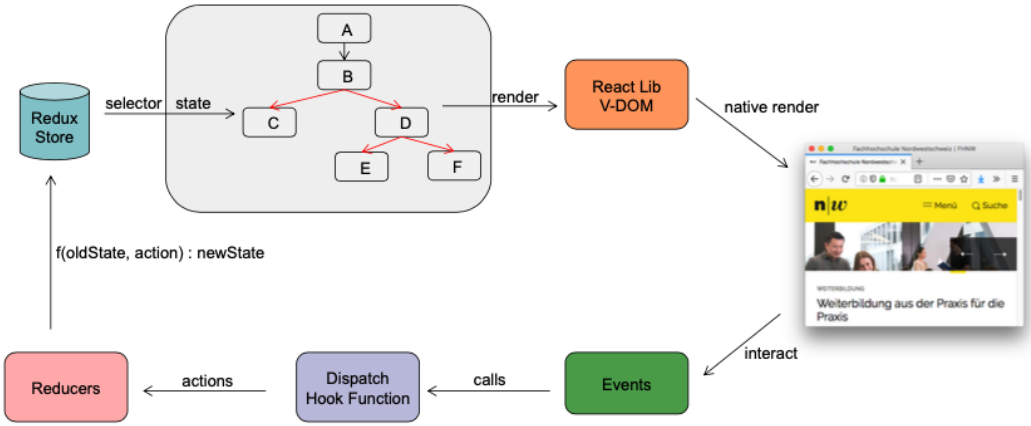
```
const store = createStore(reducer, initialState)

ReactDOM.render(
    <Provider store={store}>
        <App/>
    </Provider>,
    document.getElementById('app')
)
```

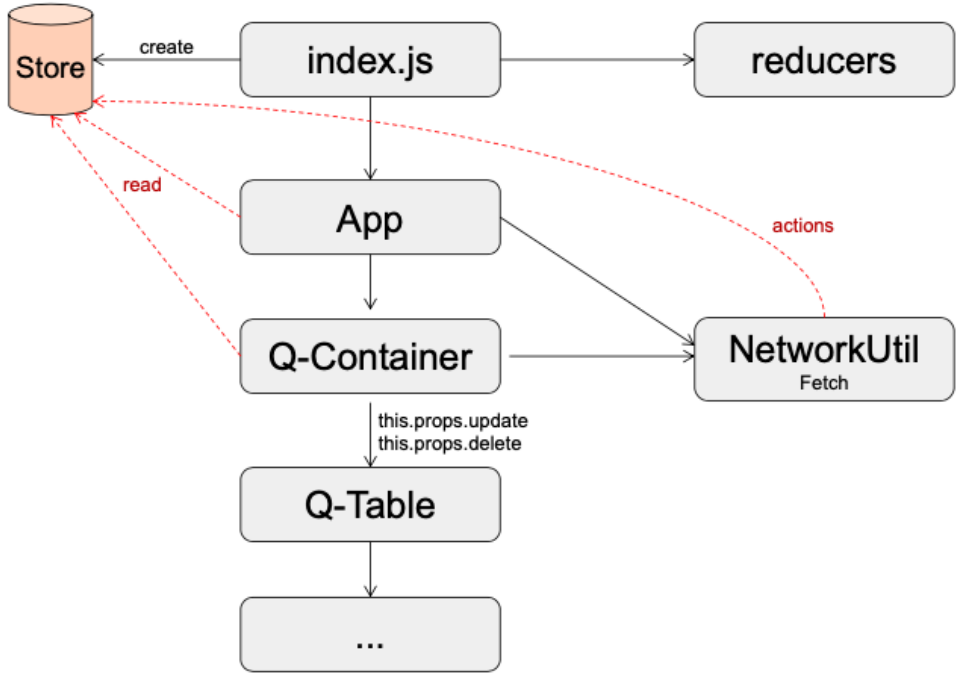
App.js

```
const filter = (movies, term) => {
    let filterTerm = '^(?=.*' + _.trim(term).split(/\s+/).join(')(?=.*') + ').*$'
    let pattern = RegExp(filterTerm, 'i')

    return _.filter(movies, movie =>
        pattern.test(_.join([movie.year, movie.director, movie.title], ' ')))
}

const App = () => {
    const movies = useSelector(state => state.movies, _.isEqual)
    const filterTerm = useSelector(state => state.filterTerm, _.isEqual)
    const dispatch = useDispatch()

    const updateFilterTerm = term =>
        dispatch({ type: 'UPDATE_FILTER_TERM', filterTerm: term })

    return <main>
        <Filter term={ filterTerm } updateFilterTerm={ updateFilterTerm } />
        { _.map(filter(movies, filterTerm), movie => <Movie key={ movie.rank } data={ movie } />) }
    </main>
}

export default App
```

Filter.js

```
const Filter = ({ filterTerm }) => {
    const dispatch = useDispatch()

    const onChange = event =>
        dispatch({ type: 'UPDATE_FILTER_TERM', filterTerm: event.target.value })

    return <form>
        <input type='text' placeholder='Liste Filtern mit...' value={ filterTerm } onChange={ onChange } />
    </form>
}

export default Filter
```





12 Final Flashcard

12.1 index.js

```
import React from 'react'
import ReactDOM from 'react-dom'
import { Provider } from 'react-redux'
import { createStore, applyMiddleware } from 'redux'
import ReduxThunk from 'redux-thunk'
import reducer from './reducers/reducers'
import 'bootstrap/dist/css/bootstrap.min.css'
import App from './app/App'

const initialState = {
    qs: [],
    message: '',
    loading: false,
    error: false,
    config: null // null, damit der Null-Test beim Rendern funktioniert.
}

const store = createStore(reducer, initialState, applyMiddleware(ReduxThunk) )

ReactDOM.render(<Provider store={ store }><App /></Provider>, document.getElementById('app'))
```

12.2 App.js

```
import React, { useEffect } from 'react'
import { useSelector, useDispatch } from 'react-redux'
import _ from 'lodash'
import { Container } from 'reactstrap'

import Header from './Header'
import QuestionnaireContainer from '../questionnaire/QuestionnaireContainer'
import Message from './Message'
import Footer from './Footer'
import doFetch from '../network/NetworkUtil'

/**
 * Die Wurzel der React-App.
 */
const App = () => {

    const config = useSelector(state => state.config, _.isEqual)
    const error = useSelector(state => state.error, _.isEqual)
    const message = useSelector(state => state.message, _.isEqual)
    const dispatch = useDispatch()

    const readConfig = () => {
        dispatch(
            doFetch({
                url: 'application.json',
                actionType: 'CONFIG'
            })
        )
    }

    useEffect(readConfig, [])

    const renderQuestionnaireContainer = config =>
        config ? <QuestionnaireContainer serverUrl={ `${ config.url }/questionnaires` } /> : null

    const renderMessage = () =>
        error ? <Message message={ message } /> : null

    return <Container>
        <Header title='Flashcard Client with React' subtitle='Version 3' />
        { renderQuestionnaireContainer(config) }
        { renderMessage() }
        <Footer message='@ The FHNW Team' />
    </Container>
}

export default App
```

12.3 Header.js

```
import React from 'react'
import { Jumbotron } from 'reactstrap'

const Header = ({ title, subtitle }) =>
    <Jumbotron >
        <h1>{ title }</h1>
        <h3>{ subtitle }</h3>
    </Jumbotron >

export default Header
```

12.4 Footer.js

```
import React from 'react'

// Ohne Destructuring.
const Footer = props =>
    <section>
        { props.message }
    </section>

export default Footer
```

12.5 Message.js

```
import React from 'react'
import { Alert } from 'reactstrap'

/**
 * Diese Komponente zeigt eine Meldung an den Benutzer an. Falls die Nachricht
 * null oder ein Leerstring ist, wird die Komponente nicht angezeigt.
 *
 * @param {string} message Die Nachricht, welche angezeigt werden soll
 * @param {bool} isError  Falls True, eine Fehlermeldung, sonst eine Infomeldung (Default: Fehlermeldung)
 */
const Message = ({ message, isError = true }) =>
    message ? <Alert color={ isError ? 'danger' : 'info' }>{ message }</Alert> : null

export default Message
```

12.6 Loader.js

```
import React from 'react'
import loader from './images/loader.gif'

const style = { display: 'block', margin: 'auto' }

const Loader = () =>
    <img style={ style } src={loader} height="128" alt="Loading..."/>

export default Loader
```

12.7 Dialog.js

```
import React, { useState, useEffect } from 'react'
import _ from 'lodash'
import { Button, Modal, ModalHeader, ModalBody, Form, FormGroup, Col } from 'reactstrap'
import InputGroup from './InputGroup'

/**
 * Modale Dialog Komponente.
 *
 * @param buttonLabel Label des Buttons in der Tabelle
 * @param title Der Titel des Dialogs
 * @param actionButtonLabel Label des 'Submit' Buttons
 * @param questionnaire Der Questionnaire
 * @param isReadOnly True, wenn die Input Felder im Dialog nicht editierbar sind, false sonst (Default: false)
 * @param actionFn Die Funktion, welche aufgerufen wird, wenn der Dialog geschlossen wird (Save, Close)
 * @param css Die CSS Klasse für den Button in der Tabelle
 */
const Dialog = ({ buttonLabel, title, actionButtonLabel, questionnaire: qx, isReadOnly = false, actionFn, css }) => {

    let [showModal, setShowModal] = useState(false)
    let [questionnaire, setQuestionnaire] = useState(qx)

    /**
     * Wir beobachten hier qx. qx ist der Questionnaire, der als props in die Komponente
     * hinein gegeben wird. Wenn sich qx ändert, wollen wir diese Änderung übernehmen.
     */
    useEffect(() => {
        setQuestionnaire(qx)
    }, [qx])

    const change = event =>
        setQuestionnaire({ ...questionnaire, [event.target.name]: event.target.value })

    const close = () =>
        setShowModal(false)

    const open = () =>
        setShowModal(true)

    /**
     * Diese Funktion wird aufgerufen, wenn Save, oder Close gedrückt wird.
     * Sie übernimmt die actionFn, die mittels props in diese Komponente
     * übergeben wird. Falls es keine actionFn gibt, wird identity verwendet.
     * Diese Funktion schliesst nach dem Aufruf der actionFn den Dialog.
     *
     * @param questionnaire Der Questionnaire
     * @param actionFn Die Action-Funktion (update, create)
     */
    const onAction = (questionnaire, actionFn) => {
        (actionFn || _.identity)(questionnaire)
        close()
    }

    return <div>
        <Button color={ css || 'secondary'} onClick={ open }
            className='float-right'>{ buttonLabel }</Button>
        <Modal isOpen={ showModal } toggle={ close } size='lg' autoFocus={false}>
            <ModalHeader toggle={ close } >
                { title }
            </ModalHeader>
            <ModalBody>
                <Form>
                    <InputGroup
                        label='Title'
                        id='formTitle'
                        changeFn={ change }
                        name='title'
                        value={ questionnaire.title }
                        isReadOnly={ isReadOnly }
                    />

                    <InputGroup
                        label='Description'
                        id='formDescription'
                        changeFn={ change }
                        name='description'
                        value={ questionnaire.description }
                        isReadOnly={ isReadOnly }
                    />

                    <FormGroup>
                        <Col className='clearfix' style={{ padding: '.2rem' }}>
                            <Button className='float-right' color='secondary'
                                onClick={ _.partial(onAction, questionnaire, actionFn) }>{ actionButtonLabel }
                            </Col>
                        </FormGroup>
                    </Form>
                </ModalBody>
            </Modal>
        </div>
    }

    export default Dialog
```

12.8 InputGroup.js

```
import React from 'react'
import { FormGroup, Label, Col, Input } from 'reactstrap'

/**
 * Zeigt ein Label und ein Input Field an.
 *
 * @param {string} label Das Label des Input Field
 * @param {string} id Die id, welche das Label und das Input Field verbindet (for - id)
 * @param {function} changeFn Die onChange Funktion des Input Fields
 * @param {string} name Der Name des Input Fields
 * @param {string} value Der Value des Input Fields
 * @param {bool} isReadOnly True, wenn das Input Field readonly ist, false sonst
 */
const InputGroup = ({ label, id, changeFn, name, value, isReadOnly }) =>
    <FormGroup row>
        <Label md={ 2 } for={ id }>
            { label }
        </Label>
        <Col md={ 10 }>
            <Input
                type='text'
                id={ id }
                onChange={ changeFn }
                name={ name }
                value={ value }
                plaintext={ isReadOnly }
                disabled={ isReadOnly } />
        </Col>
    </FormGroup>

export default InputGroup
```

12.9 QuestionnaireContainer.js

```
import React, { useEffect } from 'react'
import { useSelector, useDispatch } from 'react-redux'
import _ from 'lodash'
import QuestionnaireTable from './QuestionnaireTable'
import QuestionnaireCreateDialog from './QuestionnaireCreateDialog'
import doFetch from '../network/NetworkUtil'
import Message from '../app/Message'
import Loader from '../app/Loader'

const headers = { headers: { 'Content-Type': 'application/json; charset=utf-8' } }

/**
 * Die Questionnaire Funktionalität (Crerate, Tabelle der Questionnaires).
 *
 * @param {string} serverUrl Die URL des Backends
 */
const QuestionnaireContainer = ({ serverUrl }) => {

    const qs = useSelector(state => state.qs, _.isEqual)
    const error = useSelector(state => state.error, _.isEqual)
    const message = useSelector(state => state.message, _.isEqual)
    const loading = useSelector(state => state.loading, _.isEqual)
    const dispatch = useDispatch()

    const readAll = () => {
        dispatch(
            doFetch({
                url: serverUrl,
                actionType: 'READ_QUESTIONNAIRES',
                errorText: 'Not Found'
            })
        )
    }

    useEffect(readAll, [])

    const create = async questionnaire => {
        dispatch(
            doFetch({
                url: serverUrl,
                requestObject: { method: 'POST', body: JSON.stringify(questionnaire), ...headers },
                actionType: 'CREATE_QUESTIONNAIRES',
                errorText: 'Creation failed.'
            })
        )
    }

    const update = questionnaire => {
        dispatch(
            doFetch({
                url: `${ serverUrl }/${ questionnaire.id }`,
                requestObject: { method: 'PUT', body: JSON.stringify(questionnaire), ...headers },
                actionType: 'UPDATE_QUESTIONNAIRES',
                errorText: 'Not found, or update failed.'
            })
        )
    }

    const _delete = id => {
        dispatch(
            doFetch({
                url: `${ serverUrl }/${ id }`,
                requestObject: { method: 'DELETE' },
                actionType: 'DELETE_QUESTIONNAIRES',
                errorText: 'Not found, or delete failed.'
            })
        )
        // Ist nötig, wenn wir die REST Schnittstelle nicht verändern wollen.
        // Besser wäre es, wenn wir das gelöschte Questionnaire zurückgeben
        // und dann im NetworkUtil die ID mittels Action an den Reducer mitgeben.
        readAll()
    }

    const renderMessage = () =>
        error ? <Message message={ message } /> : null

    const renderQuestionnaireTable = (qs, update, _delete) =>
        loading ? <Loader /> : <QuestionnaireTable qs={ qs } update={ update } _delete={ _delete } />

    return <div>
            <QuestionnaireCreateDialog create={ create } />
            <h3>Questionnaires</h3>
            { renderMessage() }
            { renderQuestionnaireTable(qs, update, _delete) }
        </div>
}

export default QuestionnaireContainer
```

12.10 QuestionnaireCreateDialog.js

```
import React from 'react'
import Dialog from './Dialog'

/**
 * Erzeugt einen Questionnaire.
 *
 * @param {function} create Die create Function
 */
const QuestionnaireCreateDialog = ({ create }) => {
    return <Dialog
        buttonLabel='Add Questionnaire'
        title='Add Questionnaire'
        actionButtonLabel='Save'
        questionnaire={ { title: '', description: '' } }
        actionFn={ create }
        css='success' />
}

export default QuestionnaireCreateDialog
```

12.11 QuestionnaireShowDialog.js

```
import React from 'react'
import Dialog from './Dialog'

/**
 * Zeigt einen Questionnaire (readonly) an.
 *
 * @param {object} questionnaire Der Questionnaire, der angezeigt wird
 */
const QuestionnaireShowDialog = ({ questionnaire }) =>
    <Dialog
        buttonLabel='Show'
        title='Show Questionnaire'
        actionButtonLabel='Close'
        questionnaire={ questionnaire }
        isReadOnly={ true } />

export default QuestionnaireShowDialog
```

12.12 QuestionnaireTable.js

```
import React from 'react'
import _ from 'lodash'
import { Table } from 'reactstrap'
import QuestionnaireTableElement from './QuestionnaireTableElement'

/**
 * Die Tabelle der Questionnaires und die dazugehörigen
 * Controls (Show, Edit, Delete).
 *
 * @param {array} qs Die Liste der Questionnaires
 * @param {function} update Die Funktion zum Updaten
 * @param {function} _delete Die Funktion zum Löschen
 */
const QuestionnaireTable = ({ qs, update, _delete }) =>
    <Table hover>
        <tbody>
        {
            _.map(qs, questionnaire =>
                <QuestionnaireTableElement
                    key={ questionnaire.id }
                    questionnaire={ questionnaire }
                    update={ update }
                    _delete={ _delete } />
            )
        }
        </tbody>
    </Table>

export default QuestionnaireTable
```

12.13 QuestionnaireTableElement.js

```
import React from 'react'
import _ from 'lodash'
import { Button } from 'reactstrap'
import QuestionnaireShowDialog from './QuestionnaireShowDialog'
import QuestionnaireUpdateDialog from './QuestionnaireUpdateDialog'

/**
 * Eine Zeile in der Tabelle.
 *
 * @param {object} questionnaire Der Questionnaire, der angezeigt werden soll
 * @param {function} update Die Funktion zum Updaten
 * @param {function} _delete Die Funktion zum Löschen
 */
const QuestionnaireTableElement = ({ questionnaire, update, _delete }) => (
    <tr key={ questionnaire.id } >
        <td>{ questionnaire.id }</td>
        <td>{ questionnaire.title }</td>
        <td>{ questionnaire.description }</td>
        <td><QuestionnaireShowDialog questionnaire={ questionnaire } /></td>
        <td><QuestionnaireUpdateDialog update={ update } questionnaire={ questionnaire } /></td>
        <td><Button color='danger'
                    onClick={ _.partial(_delete, questionnaire.id) }
                    className='float-right' >Delete</Button>
        </td>
    </tr>
)

export default QuestionnaireTableElement
```

12.14 QuestionnaireUpdateDialog.js

```
import React from 'react'
import Dialog from './Dialog'

/**
 * Passt einen bestehenden Questionnaire an.
 *
 * @param {object} questionnaire Der Questionnaire, der angepasst werden soll
 * @param {function} update Die update Funktion
 */
const QuestionnaireUpdateDialog =  ({ questionnaire, update }) =>
    <Dialog
        buttonLabel='Edit'
        title='Edit Questionnaire'
        actionButtonLabel='Save'
        questionnaire={ questionnaire }
        actionFn={ update }
        css='primary' />

export default QuestionnaireUpdateDialog
```

12.15 reducers.js

```
import _ from 'lodash'

const REDUCERS = {
    'READ_QUESTIONNAIRES': (state, action) => ({ ...state, qs: action.data }),
    'CREATE_QUESTIONNAIRES': (state, action) => ({ ...state, qs: [...state.qs, action.data ] }),
    'UPDATE_QUESTIONNAIRES': (state, action) => ({ ...state, qs: _.map(state.qs, q => q.id === action.data.id
    'DELETE_QUESTIONNAIRES': (state, action) => ({ ...state, qs: _.reject(state.qs, { id: action.data }) }),
    'LOADING': (state, action) => ({ ...state, loading: action.data }),
    'MESSAGE': (state, action) => ({ ...state, message: action.data }),
    'ERROR': (state, action) => ({ ...state, error: action.data }),
    'CONFIG': (state, action) => ({ ...state, config: action.data })
}

const reducer = (state, action) => _.get(REDUCERS, action.type, _.identity)(state, action)

export default reducer
```

12.16 NetworkUtil.js

```
/**
 * Eine Funktion, welche den Fetch-Request absetzt und die entsprechenden
 * Actions an den Store dispatched.
 *
 * @param {string} url Der URL
 * @param {object} requestObject Das Objekt mit den Headern, Methoden (wenn nicht GET) (Optional)
 * @param {string} actionType Der Action-Type
 * @param {string} errorText Den Fehlertext
 * @return {function} Die Dispatch Funktion für Redux-Thunk
 */
const doFetch = ({ url, requestObject, actionType, errorText }) =>
    async dispatch => {
        try {
            dispatch({ type: 'LOADING', data: true })
            const response = await fetch(url, requestObject)
            if(response.ok) {
                // Mit leerem Response umgehen:
                const json = response.status !== 204 ? await response.json() : null
                dispatch({ type: actionType, data: json })
                dispatch({ type: 'ERROR', data: false })
                dispatch({ type: 'MESSAGE', data: '' })
            }
            else {
```

```
            throw new Error(`${ errorText }: ${ response.status }`)
        }
    }
    catch(error) {
        dispatch({ type: 'ERROR', data: true })
        dispatch({ type: 'MESSAGE', data: error.message })
    }
    dispatch({ type: 'LOADING', data: false })
}

export default doFetch
```