

Systemtechnik Labor

4AHIT 2017/18

Document Oriented Middleware

Laborprotokoll

Marc Rousavy

10. Juni 2018

Bewertung:

Betreuer: Thomas Micheler

Version: 1.1

Begonnen: 6. Juni 2018

Beendet: 7. Juni 2018

Inhaltsverzeichnis

1	Einführung	3
1.1	Ziele	3
1.2	Voraussetzungen	3
1.3	Aufgabenstellung	3
1.4	Bewertung	4
1.5	Fragestellung für Protokoll	4
2	Lösung	5
2.1	Projekt migrieren	5
2.1.1	Neues Projekt erstellen	6
2.2	Windpark	8
2.2.1	Allgemein	8
2.2.2	ReST Controller	8
2.2.3	JSON	9
2.2.4	Port	11
2.2.5	Start	11
2.3	Zentrale	13
2.3.1	Allgemein	13
2.3.2	Model	13
2.3.3	Data Access Layer	13
3	Fragen	16

1 Einführung

Diese Übung soll helfen die Funktionsweise und Einsatzmöglichkeiten eines dokumentenorientierten dezentralen Systems mit Hilfe des Frameworks Spring Data MongoDB zu demonstrieren. Die Daten werden in dieser Übung in einem NoSQL Repository gespeichert und verarbeitet.

Die Umsetzung erfolgt, wie auch bei GK8.1 "Message Oriented Middleware", anhand des Beispiels von Windkraftanlagen. Es wird angenommen, dass die Werte der Windkraftanlagen am Parkrechner im XML Format vorliegen. Mit Hilfe einer REST Schnittstelle sollen die Daten an die Zentrale weitergegeben und hier mit Hilfe eines dokumentenorientiertem dezentralen Systems gespeichert werden. Von diesem System können die Daten für verschiedene Anwendungsfälle weiterverarbeitet werden.

1.1 Ziele

Das Ziel dieser Übung ist die Implementierung einer dokumentenorientiertem Middleware, die die Daten der Windparks zentral in einer entsprechenden Form ablegt. In dieser Übung soll auch ein Anwendungsfall umgesetzt werden, bei dem bestimmte Daten der Windparks in einem Browser angezeigt werden.

1.2 Voraussetzungen

- Grundlagen zu XML & JSON & REST
- Grundlagen Architektur von verteilten Systemen
- Grundlagen Architektur von verteilten Systemen
- Grundlagen NoSQL
- Installation MongoDB
- Installation MongoDB Verwendung der XML-Datenstruktur eines Parkrechner "parknode-data.xml"

1.3 Aufgabenstellung

Implementieren Sie eine dokumentenorientierte Middleware mit Hilfe von Spring Data MongoDB und simulieren Sie die ständige Aktualisierung der Daten an der REST Schnittstelle der Parkrechner. Es sollen dabei keine Daten verloren gehen, sondern stets mit einem Zeitstempel und einem entsprechenden Format in der Zentrale abgespeichert werden. Bedenken Sie, dass die Daten aller Windparks und somit aller Windkraftanlagen zusammentreffen. Entwerfen Sie eine geeignet Datenstruktur, um eine kontinuierliche Speicherung der Daten zu gewährleisten.

Die Daten liegen im XML-Format am Parkrechner vor und sollen als JSON-Struktur in MongoDB gespeichert werden. In welcher Form und in welchen Zeitabständen die Daten eintreffen wird von Ihnen (System Architekt) spezifiziert und umgesetzt.

Die Daten werden in der Zentrale in einem MongoDB Repository gespeichert und können hier zu Kontrollzwecken abgerufen werden (mongo Shell).

Ebenso soll ein einfaches Webinterface für die Zentrale implementiert werden, die die Daten anhand einer von Ihnen gewählten Fragestellung auswertet und diese im Browser darstellt. Dabei soll die einfache Verarbeitung der Daten, die im JSON Format vorliegen, aufgezeigt werden.

1.4 Bewertung

- Gruppengröße: 1 Person
- Anforderungen "überwiegend erfüllt"
 - Installation und Konfiguration einer dokumentenorientierten Middleware mit Hilfe von Spring Data MongoDB
 - Entwurf und Umsetzung einer entsprechenden JSON Datenstruktur
 - Transformierung der XML-Daten (parknodedata.xml) in ein entsprechendes JSON-Format
 - Formulierung einer sinnvollen Fragestellung für einen Anwendungsfall in der Zentrale und deren Abfrage in einer Mongo Shell
 - Umsetzung von einem Parkrechner
- Anforderungen "zur Gänze erfüllt"
 - Konzeption und Implementierung der kontinuierlichen Speicherung der Daten (Cron-job, Scheduler, Trigger, etc.)
 - Implementieren eines Webinterfaces zur Darstellung der Fragestellung von oben.
 - Logging der neuen Daten und ggf. der auftretenden Probleme
 - Umsetzung von n Parkrechnern

1.5 Fragestellung für Protokoll

- Nennen Sie 5 Vorteile eines NoSQL Repository im Gegensatz zu einem relationalen DBMS
- Nennen Sie 4 Nachteile eines NoSQL Repository im Gegensatz zu einem relationalen DBMS
- Welche Schwierigkeiten ergeben sich bei der Zusammenführung der Daten?
- Können die Daten der MongoDB von Mitarbeitern geändert werden? Ja/Nein, Begründen Sie Ihre Antwort.
- Beschreiben Sie die wichtigsten Eigenschaften des Spring Frameworks?
- Was versteht man unter dem Spring Boot Projekt?
- Nennen Sie jeweils 3 Argumente für und gegen den Einsatz von Spring bei der Entwicklung solcher Projekte

2 Lösung

2.1 Projekt migrieren

Da die Aufgabe die bestehende Aufgabe '**Message Oriented Middleware**' erweitert, wird die gesamte Codebase als Grundlage weiterverwendet.

Das Projekt bestand aus einem Windpark welcher eine Liste aus Windrädern besitzt.

Außerdem gibt es eine Zentrale welche über die Middleware die Informationen des Windparks erhält.

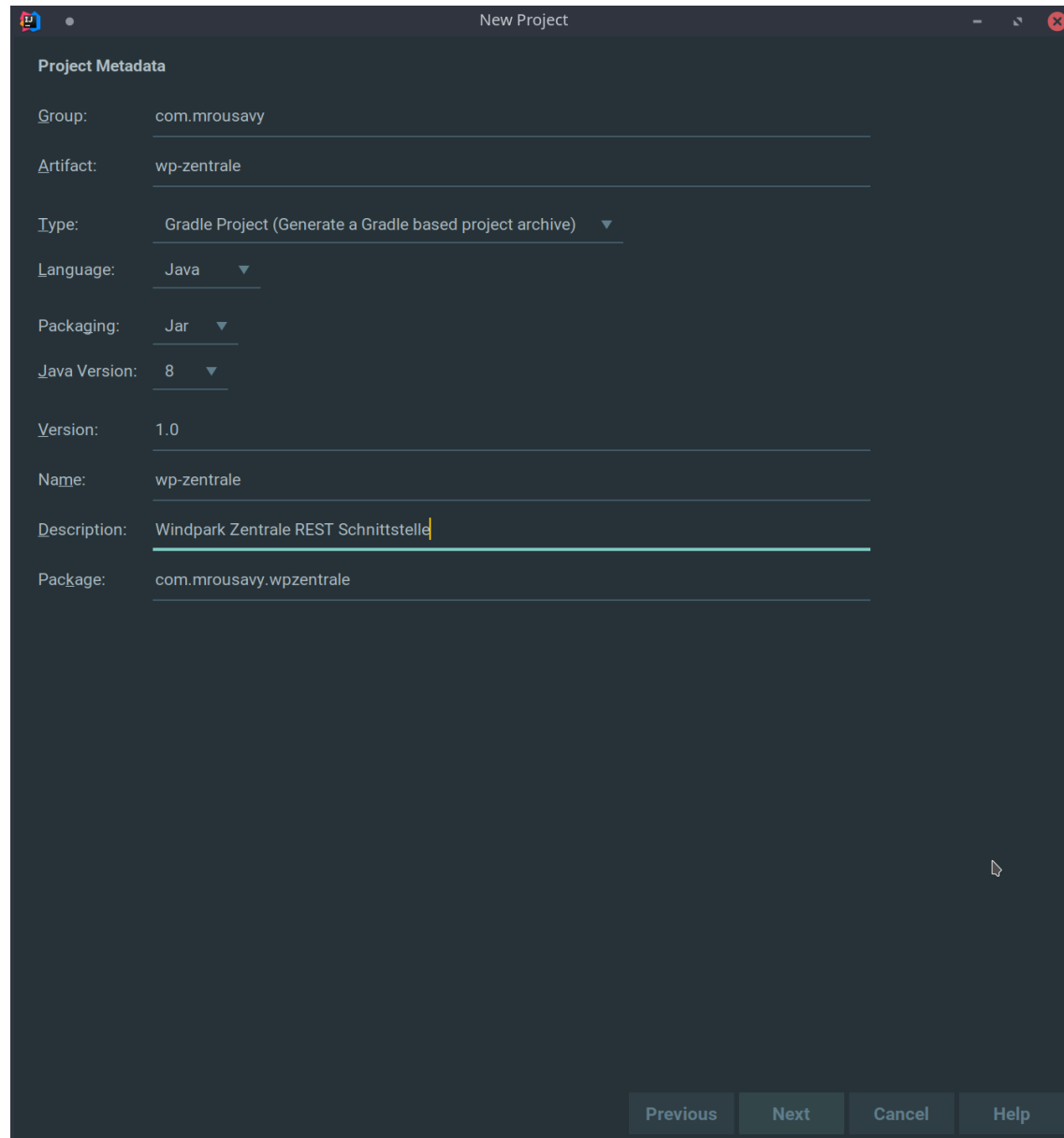
Dies soll erweitert werden, sodass statt der **Message Oriented** Middleware eine **Document Oriented** Middleware verwendet wird.

2.1.1 Neues Projekt erstellen

Das erstellen des neuen Projektes erfolgt mit dem **Spring Initializr**.

Folgende allgemeine Projekt Einstellungen habe ich konfiguriert:

Abbildung 1: Spring Initializr Projektkonfiguration



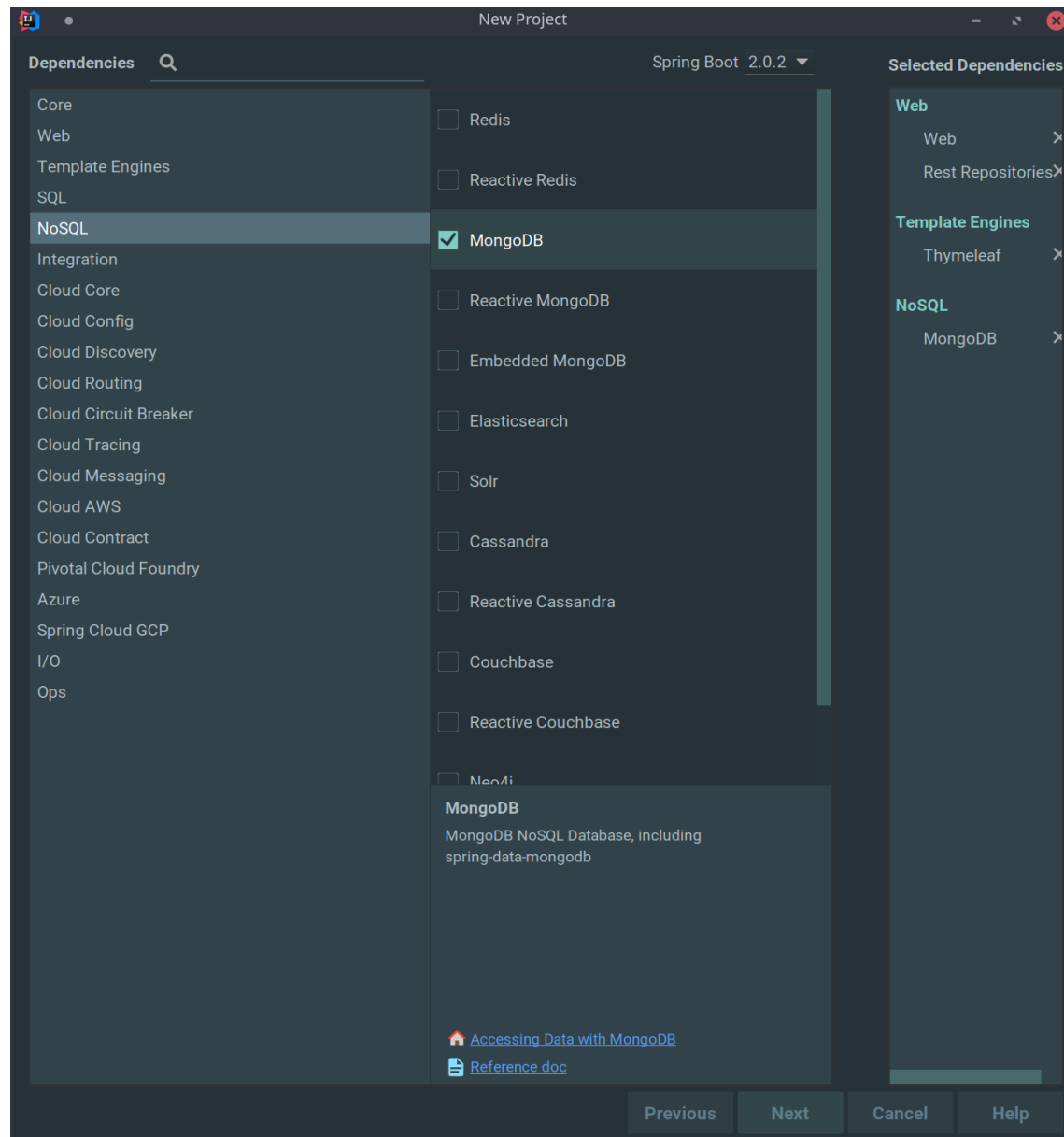
The screenshot shows the 'New Project' window of the Spring Initializr. The window has a dark theme and contains the following configuration fields under the 'Project Metadata' section:

- Group:** com.mrousavy
- Artifact:** wp-zentrale
- Type:** Gradle Project (Generate a Gradle based project archive) ▼
- Language:** Java ▼
- Packaging:** Jar ▼
- Java Version:** 8 ▼
- Version:** 1.0
- Name:** wp-zentrale
- Description:** Windpark Zentrale REST Schnittstelle
- Package:** com.mrousavy.wpzentrale

At the bottom right of the window, there are four buttons: 'Previous', 'Next', 'Cancel', and 'Help'.

Sowie folgende Frameworks für das Windpark Projekt:

Abbildung 2: Spring Initializr Framework Konfiguration



2.2 Windpark

2.2.1 Allgemein

Eine Spring Web Application wird nun einen einzelnen Windpark darstellen.

Es wird also zusätzlich zu folgender Application Klasse:

```
1 package com.mrousavy.wpwindpark;  
2  
3 import org.springframework.boot.SpringApplication;  
4 import org.springframework.boot.autoconfigure.SpringBootApplication;  
5  
6 @SpringBootApplication  
7 public class WpWindparkApplication {  
8     public static void main(String[] args) {  
9         SpringApplication.run(WpWindparkApplication.class, args);  
10    }  
11 }
```

..., welche mittels des *Spring Initializr* automatisch erstellt wurde, noch ein **ReST Controller** erstellt.

2.2.2 ReST Controller

Dieser **ReST Controller** kümmert sich um alle Anfragen über *HTTP/HTTPS*, und liefert einen String zurück.

Definiert werden **ReST Controller** in **Spring** mittels der `@RestController` Annotation:

```
1 @RestController  
2 public class WindparkRestController {
```

Sobald also die Spring Application über die URL beispielsweise in einem Browser aufgerufen wird, sucht **Spring** einen passenden **ReST Controller** um die Anfrage zu verarbeiten. Dies gelangt zur Laufzeit mittels der vorhin erwähnten `@RestController` Annotation.

Wir können einerseits eine *Index* (also die Startseite, beispielsweise erreichbar unter `172.0.1.2`), oder eine Subseite (beispielsweise erreichbar unter `172.0.1.2/hello`) definieren. Hierbei verwenden wir die Spring Web Annotation `@GetMapping(String)`.

Um einen Wert (String) zurückzugeben, wird die Spring Web Annotation `@ResponseBody` verwendet.

Also können wir beispielsweise folgende Funktion definieren:

```
1 @GetMapping("/hello")  
2 @ResponseBody  
3 public String sayHello() {  
4     return "<p>Hello world!</p>";  
5 }
```


Der Windpark Controller ist so aufgebaut, dass damit gerechnet wird, dass ein vorgefertigtes XML File auf dem Server abgespeichert ist, welches über eine *ReST GET-Function* abgerufen wird.

Um eine Text-Datei in Java 7 zu lesen, wird der `BufferedReader` verwendet. Es wird mit der `BufferedReader.readLine()` Methode gearbeitet.

```

1  package com.mrousavy.wpwindpark;
2
3  import org.springframework.web.bind.annotation.GetMapping;
4  import org.springframework.web.bind.annotation.ResponseBody;
5  import org.springframework.web.bind.annotation.RestController;
6
7  @RestController
8  public class WindparkRestController {
9
10     private final String FILE_NAME = "parknodedata.xml";
11
12     public String readXml() throws IOException {
13         try (BufferedReader br = new BufferedReader(new FileReader(FILE_NAME))) {
14             StringBuilder builder = new StringBuilder();
15
16             String sCurrentLine;
17             while ((sCurrentLine = br.readLine()) != null) {
18                 builder.append(sCurrentLine);
19             }
20
21             return builder.toString();
22         }
23     }
24 }

```

2.2.3 JSON

Außerdem wird die Dependency `org.json` verwendet, um das XML zu einem JSON Format zu konvertieren:

```

1  dependencies {
2      compile('org.springframework.boot:spring-boot-starter-data-rest')
3      compile('org.springframework.boot:spring-boot-starter-web')
4      compile('org.jdom:jdom2:2.0.6')
5      compile('xerces:xercesImpl:2.11.0')
6      compile group: 'org.json', name: 'json', version: '20180130'
7      testCompile('org.springframework.boot:spring-boot-starter-test')
8  }

```

Der vollständige Code des **ReST Controllers** sieht nun folgendermaßen aus:

```

1  @RestController
2  public class WindparkRestController {
3

```

```
4     private final String FILE_NAME = "parknodedata.xml";
5
6     public String readXml() throws IOException {
7         try (BufferedReader br = new BufferedReader(new FileReader(FILE_NAME))) {
8             StringBuilder builder = new StringBuilder();
9
10            String sCurrentLine;
11            while ((sCurrentLine = br.readLine()) != null) {
12                builder.append(sCurrentLine);
13            }
14
15            return builder.toString();
16        }
17    }
18
19    @GetMapping("/xml")
20    @ResponseBody
21    public String getXml() {
22        System.out.println("Working Directory = " +
23            ↪ System.getProperty("user.dir"));
24
25        try {
26            return readXml();
27        } catch (Exception e) {
28            e.printStackTrace();
29            return "<error>" + e.getMessage() + "</error>";
30        }
31    }
32
33    @GetMapping("/json")
34    @ResponseBody
35    public String getJson() {
36        System.out.println("Working Directory = " +
37            ↪ System.getProperty("user.dir"));
38
39        try {
40            String xml = readXml();
41
42            JSONObject json = XML.toJSONObject(xml);
43            String prettyJson = json.toString(4);
44            System.out.println("Read JSON: " + prettyJson);
45            return prettyJson;
46        } catch (Exception e) {
47            e.printStackTrace();
48            return "<error>" + e.getMessage() + "</error>";
49        }
50    }
```

2.2.4 Port

Da wir mehrere Windparks auf einem Host starten möchten, müssen wir Spring mitteilen, verschiedene Ports für den Tomcat Server zu verwenden.

Hierzu verwenden wir die `application.properties` Datei:

```
1 server.port = 8081
```

2.2.5 Start

Sobald wir die Main-Methode starten, wird der Spring Server initialisiert:

Abbildung 3: Spring Console Output - Initialization

```
c.m.wpark.WpWindparkApplication : Starting WpWindparkApplication on xps-15 with PID 6113 (/home/mrousavy/School/SYT/GK8.2/wp-windpark/out/production/classes started by mrousavy in /home/mro
c.m.wpark.WpWindparkApplication : No active profile set, falling back to default profiles: default
ConfigServletWebServerApplicationContext : Refreshing org.springframework.boot.web.servlet.context.AnnotationConfigServletWebServerApplicationContext@30b7c004: startup date [Sun Jun 10 13:37:44 CEST 2018]
o.s.b.f.s.DefaultListableBeanFactory : Overriding bean definition for bean 'httpRequestHandlerAdapter' with a different definition: replacing [Root bean: class [null]; scope=; abstract=false; lazyIni
o.s.b.w.embedded.tomcat.TomcatWebServer : Tomcat initialized with port(s): 8081 (http)
o.apache.catalina.core.StandardService : Starting service [Tomcat]
org.apache.catalina.core.StandardEngine : Starting Servlet Engine: Apache Tomcat/8.5.31
o.a.catalina.core.AprLifecycleListener : The APR based Apache Tomcat Native library which allows optimal performance in production environments was not found on the java.library.path: [/usr/java/packages
o.a.c.c.C.[Tomcat].[localhost].[/] : Initializing Spring embedded WebApplicationContext
o.s.web.context.ContextLoader : Root WebApplicationContext: initialization completed in 866 ms
o.s.b.w.servlet.ServletRegistrationBean : Servlet dispatcherServlet mapped to [/]
```

Außerdem wird hier die `/xml` **ReST Schnittstelle** "gemapped":

Abbildung 4: Spring Console Output - XML ReST Interface Mapping

```
o.s.w.s.m.a.RequestMappingHandlerMapping : Mapped "{[/xml],methods=[GET]}" onto public java.lang.String com.mrousavy.wpark.WindparkRestController.getXml()
```

Zuletzt wird der Tomcat Server gestartet:

Abbildung 5: Spring Console Output - Tomcat Server Start finish

```
o.s.j.e.a.AnnotationMBeanExporter : Registering beans for JMX exposure on startup
o.s.b.w.embedded.tomcat.TomcatWebServer : Tomcat started on port(s): 8081 (http) with context path ''
c.m.wpark.WpWindparkApplication : Started WpWindparkApplication in 2.416 seconds (JVM running for 2.896)
o.a.c.c.C.[Tomcat].[localhost].[/] : Initializing Spring FrameworkServlet 'dispatcherServlet'
o.s.web.servlet.DispatcherServlet : FrameworkServlet 'dispatcherServlet': initialization started
o.s.web.servlet.DispatcherServlet : FrameworkServlet 'dispatcherServlet': initialization completed in 9 ms
```

Die Spring Application ist nun erreichbar unter `localhost:8081/xml`.

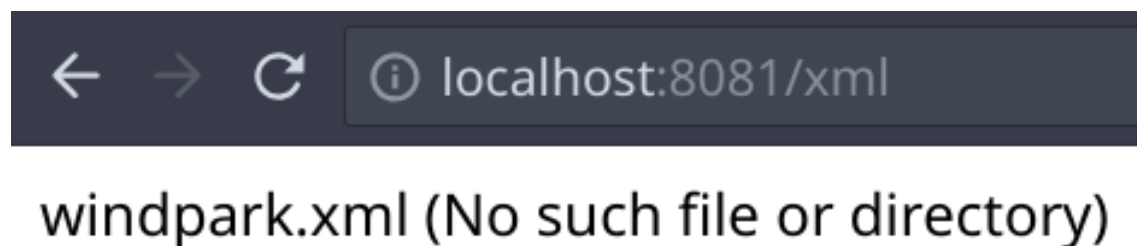
Abbildung 6: Spring Application Web-Aufruf (JSON)

```
< - - localhost:8081/json
{"windpark": { "windrad": [ { "transfertime": "850 ms", "bladeposition": "25.8 deg", "blindpower": "382.54 kWh", "temperature": "25.8 C",
"windspeed": "40.54 km/h", "rotationspeed": "0.42 m/s", "id": "0001", "power": "752.51 kWh" }, { "transfertime": "857 ms", "bladeposition": "23.1
deg", "blindpower": "281.54 kWh", "temperature": "23.1 C", "windspeed": "32.54 km/h", "rotationspeed": "0.25 m/s", "id": "0002", "power":
"421.35 kWh" }, { "transfertime": "777 ms", "bladeposition": "19.0 deg", "blindpower": "222.25 kWh", "temperature": "24.9 C", "windspeed": "29.10
km/h", "rotationspeed": "0.21 m/s", "id": "0003", "power": "251.2 kWh" } ], "id": "NOE0001" }}
```

```
1  {"windpark": {
2    "windrad": [
3      {
4        "transfertime": "850 ms",
5        "bladeposition": "25.8 deg",
6        "blindpower": "382.54 kWh",
7        "temperature": "25.8 C",
8        "windspeed": "40.54 km/h",
9        "rotationspeed": "0.42 m/s",
10       "id": "0001",
11       "power": "752.51 kWh"
12     },
13     {
14       "transfertime": "857 ms",
15       "bladeposition": "23.1 deg",
16       "blindpower": "281.54 kWh",
17       "temperature": "23.1 C",
18       "windspeed": "32.54 km/h",
19       "rotationspeed": "0.25 m/s",
20       "id": "0002",
21       "power": "421.35 kWh"
22     },
23     {
24       "transfertime": "777 ms",
25       "bladeposition": "19.0 deg",
26       "blindpower": "222.25 kWh",
27       "temperature": "24.9 C",
28       "windspeed": "29.10 km/h",
29       "rotationspeed": "0.21 m/s",
30       "id": "0003",
31       "power": "251.2 kWh"
32     }
33   ],
34   "id": "NOE0001"
35 }}
```

Und falls es irgendeinen Fehler in der Applikation gibt:

Abbildung 7: Spring Application Web-Aufruf - Error caught



2.3 Zentrale

2.3.1 Allgemein

Nun erstellen wir eine **Spring Application** welche eine **Windpark Zentrale** repräsentiert.

Der Aufbau dieser **Spring Application** ist folgendermaßen gedacht:

- Es werden mehrere Windparks eingetragen welche überwacht werden sollen. Diese Windparks werden mittels ihrer **ReST** Adresse eingetragen (also **IP**, **Port** und **Subseite** - beispielsweise: 172.0.1.2:8081/xml)
- Diese einzelnen Windparks werden über Ihre **ReST** Schnittstelle *durch-iteriert*, und einzeln *geparsed*.
- Nun werden die Windparks mittels Aufzeichnungsdatum fortlaufend auf einer **NoSQL MongoDB Datenbank** im **JSON** Format abgespeichert.
- Verwaltet werden diese Daten mittels den **MongoDB** Tools (Mongo Shell)
- Zusätzlich soll ein Webinterface erstellt werden, wo die Daten von der Mongo Datenbank angezeigt werden.

2.3.2 Model

Es wird ein **Model** definiert, welches die einzelnen Windräder, Windparks und Windpark Versionen als POJOs (**Plain Old Java Objects**) repräsentiert.

Also werden die Klassen `Windpark.java`, `Windrad.java` und `WindparkVersion.java` erstellt, welche für den **MongoDB** Zugriff auch noch Mongo Repositories benötigen.

Ein Mongo Repository verwaltet die Schnittstelle zwischen der Java Runtime und dem MongoDB Treiber. Für den Java Code ist es also quasi die Schnittstelle zur Datenbank.

Eine Repository könnte beispielsweise folgendermaßen aussehen:

```
1 public interface WindparkRepository extends MongoRepository<Windpark, String> {  
2     public Windpark findById(String id);  
3 }
```

2.3.3 Data Access Layer

Um die einzelnen Daten nun von den Windparks zu bekommen, erstellen wir einen **Data Access Layer**, welcher folgende Aufgaben hat:

- Windpark Zentrale Config lesen (zu überwachende Windparks auslesen)
- Windparks JSON auslesen über deren ReST Schnittstelle
- Einzelne Windparks Zusammenfassen
- Auf MongoDB Datenbank fortlaufend speichern

Es wird die Java Klasse `DataAccessLayer.java` erstellt, welche die einzelnen Mongo Repositories als Klassenvariablen verwendet:

```

1  public class DataAccessLayer {
2      // MongoDB Repositories
3      private WindparkRepository windparks;
4      private WindparkVersionRepository windparkVersions;
5      private WindradRepository windrads;
6
7
8      public DataAccessLayer(WindparkRepository windparks,
9          ↪ WindparkVersionRepository windparkVersions, WindradRepository windrads)
10         ↪ {
11         this.windparks = windparks;
12         this.windparkVersions = windparkVersions;
13         this.windrads = windrads;
14     }
15
16     ...

```

Außerdem verwenden wir eine Konfigurationsdatei welche alle Windpark URLs enthält, diese wird folgendermaßen ausgelesen:

```

1  private static final String FILE_NAME = "urls.txt";
2
3  public ArrayList<String> getUrls() {
4      try (BufferedReader br = new BufferedReader(new FileReader(FILE_NAME))) {
5          ArrayList<String> urls = new ArrayList<>();
6
7          String currentLine;
8          while ((currentLine = br.readLine()) != null) {
9              urls.add(currentLine.trim());
10         }
11
12         return urls;
13     } catch (Exception e) {
14         e.printStackTrace();
15         return new ArrayList<>();
16     }
17 }

```

Die JSON Informationen der einzelnen Windparks können wir mittels folgender Funktion ermitteln:

```

1  public static String getWindparkData(String url) throws URISyntaxException {
2      RestTemplate rest = new RestTemplate();
3      return rest.exchange(new URI(url), HttpMethod.GET, null,
4          ↪ String.class).getBody();
5  }

```

Hierbei ist aufzupassen, dass die URLs im richtigen format sind.
TODO: rest

3 Fragen

- Nennen Sie 5 Vorteile eines NoSQL Repository im Gegensatz zu einem relationalen DBMS
- Nennen Sie 4 Nachteile eines NoSQL Repository im Gegensatz zu einem relationalen DBMS
- Welche Schwierigkeiten ergeben sich bei der Zusammenführung der Daten?
- Können die Daten der MongoDB von Mitarbeitern geändert werden? Ja/Nein, Begründen Sie Ihre Antwort.
- Beschreiben Sie die wichtigsten Eigenschaften des Spring Frameworks?
- Was versteht man unter dem Spring Boot Projekt?
- Nennen Sie jeweils 3 Argumente für und gegen den Einsatz von Spring bei der Entwicklung solcher Projekte

Abbildungsverzeichnis

1	Spring Initializr Projektkonfiguration	6
2	Spring Initializr Framework Konfiguration	7
3	Spring Console Output - Initialization	11
4	Spring Console Output - XML ReST Interface Mapping	11
5	Spring Console Output - Tomcat Server Start finish	11
6	Spring Application Web-Aufruf (JSON)	11
7	Spring Application Web-Aufruf - Error caught	12