ingenieur
wissenschaften
htw saar

Hochschule für
Technik und Wirtschaft
des Saarlandes
University of
Applied Sciences

Projektbericht REST-API

vorgelegt von
Arne Kreuz,
Joshua Nestler,
Moritz Schönenberger

betreut und begutachtet von
Prof. Dr. Reinhard Brocks

Saarbrücken, 18. September 2024

Inhaltsverzeichnis

1 Projekt

1.1 Zielsetzung

Write

1.2 Projektbeiträge

- **Spezifikation**: Alle
- **Implementation**: Joshua Nestler
- **Testing - Postman**: Arne Kreuz
- **Testing - JMeter**: Moritz Schönenberger
- **CI/CD**: Joshua Nestler

1.3 Verwendete Technologien

Technologie	Version	Link
Java	21	`https://www.java.com/`
Maven	3.8.5	`https://maven.apache.org/`
Jakarta EE	11.0.0	`https://jakarta.ee/`
Wildfly	32.0.0	`https://www.wildfly.org/`
.NET	8.0	`https://dotnet.microsoft.com/`
ASP.NET Core	8.0	`https://dotnet.microsoft.com/apps/aspnet`
H2	2.2.224	`https://www.h2database.com/`
JMeter	5.4.1	`https://jmeter.apache.org/`
Postman	10.23.4	`https://www.postman.com/`
Docker	27	`https://www.docker.com/`
IntelliJ IDEA	2024	`https://www.jetbrains.com/idea/`
Rider	2024	`https://www.jetbrains.com/rider/`
Git	2.45.0	`https://git-scm.com/`
GitHub	-	`https://github.com/`
GitHub Actions	-	`https://github.com/features/actions`
OpenAPI	3.0.3	`https://www.openapis.org/`
LaTeX	2023	`https://www.latex-project.org/`

Tabelle 1.1: Verwendete Technologien

2 Spezifikation

2.1 Resourcen

Als Thema der API haben wir uns für einen einfachen Internet-Blog entschieden. In diesem Blog sollen Benutzer (`User`) Blog-Artikel (`Posts`) schreiben können und unter Artikeln Kommentare (`Comments`) verfassen können.

Daraus bilden wir ein Datenbank Entity Relationship (ER)-Diagram (siehe 2.1)

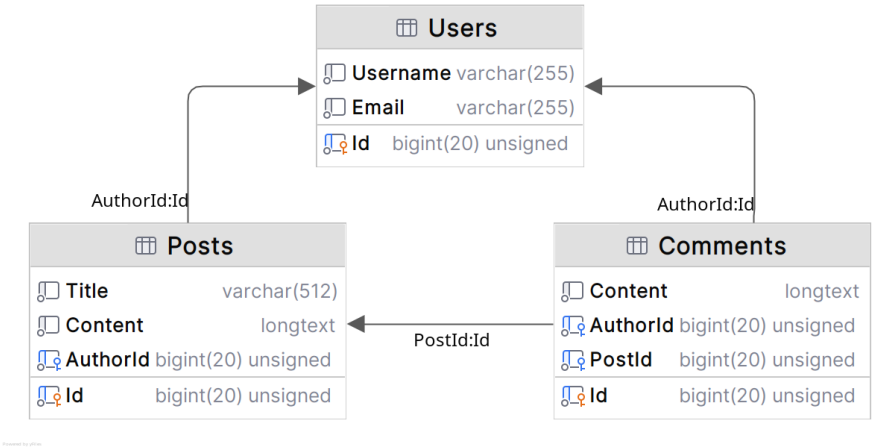

Abbildung 2.1: ER-Diagram

Die einzelnen Entities verwenden wir um daraus Resourcen der REST-API (und damit die Models und Data Transfer Object (DTO)) zu bauen.

2.1.1 Modelle

Die Modelle sind vollständig ersichtlich aus dem ER-Diagramm (siehe 2.1). Hier werden nur zusätzlich Einschränkungen zu den Eigenschaften der Modelle festgelegt. Beispielsweise muss die E-Mail eines Nutzers dem E-Mail-Format entsprechen. Daraus ergibt sich für jede Entity ein validierbares Schema, welches in der API verwendet werden kann.

2.2 OpenAPI

Die OpenAPI-Spezifikation dieses Projektes ist unter dem Pfad `docs/specification.yaml` *zu finden.*

Zur Spezifikation unserer REST-API benutzen wir die **OpenAPI**[1] Spezifikation. Mit diesem Standard ist es möglich die Struktur (Endpunkte) und das Verhalten (Antwortmöglichkeiten- und Formate, Authentifizierung, etc) zu spezifizieren. Dies geschieht wahlweise im Format `JSON` oder `YAML`, damit ist es maschinell lesbar.

[1] https://www.openapis.org

2.2.1 Vorgehen

Da die Resourcen feststehen (siehe 2.1) sind die Endpunkte der REST-API entsprechend einfach zu spezifizieren.

Pro Resource (beispielsweise unter `/resource`) gibt es 5 Endpunkte:

- **GET** `/resource`: Rufe alle Elemente dieser Resource ab.
- **POST** `/resource`: Erstelle ein neues Element dieser Resource.
- **GET** `/resource/{id}`: Rufe das Element mit der ID `{id}` ab.
- **PUT** `/resource/{id}`: Aktualisiere das Element mit der ID `{id}`.
- **DELETE** `/resource/{id}`: Lösche das Element mit der ID `{id}`.

Zu jedem dieser Endpunkte spezifizieren wir das Anfrageschema und die möglichen Antwortschemata. Hier sind dies beispielsweise folgende mögliche Antworten:

- **200** `OK`: Anfrage ist erfolgreich verarbeitet.
- **400** `Bad Request`: Anfrage ist fehlerhaft.
- **403** `Unauthorized`: Anfrage ist nicht authentifiziert.

Wir können weitere Meta-Daten zu diesen Endpunkten angeben, wie zum Beispiel eine kurze Zusammenfassung, die Authentifizierungsanforderungen oder Gruppierungen.

2.2.2 Möglichkeiten der OpenAPI-Spezifikation

Eine vollständige OpenAPI-Spezifikation kann in meheren Bereichen eingesetzt werden. Im folgenden eine kurze, unvollständige Auflistung der Möglichkeiten.

Für eine größere Liste an Möglichkeiten, siehe `https://openapi.tools/`

Code-Generation

Mit OpenAPI-Generators[2] kann man sich Clients, Server Stubs, Tests oder Dokumentation generieren lassen.

Beispiele für generierte Ausgaben der OpenAPI-Generators für dieses Projekt sind zu finden unter:

- `docs/openapi-generated-html1`: HTML-Dokumentation 1
- `docs/openapi-generated-html2`: HTML-Dokumentation 2
- `docs/openapi-generated-markdown`: Markdown-Dokumentation
- `openapi-generated-jaxrs-jersey`: Jersey Server Stubs
- `openapi-generated-java-client`: Java API Client
- `openapi-generated-aspnetcore`: ASP.NET Core Server Stubs

Mock-Server

Es gibt verschiedene Services, welche eine Spezifikation dazu verwenden Standardantworten zurückzusenden. Dies funktioniert auch ohne Serverimplementation.

[2] https://github.com/OpenAPITools/openapi-generator

3 Implementierung Webserver

3.1 Jakarta EE

Als Referenzimplementierung für dieses Projekt wurde `Jakarta EE` verwendet.

Diese Implementierung dient als Basis gegenüber der ASP.NET Implementierung (siehe 3.2)

3.1.1 Projektstruktur

Das Jakarta-Projekt gliedert sich in folgende Packages:

- **middlewares**: Beinhaltet die Anfrage-Präprozessoren (hier die Authentifizierung)
- **model**: Beinhaltet die DTOs und die Database-Entities.
- **repositories**: Beinhaltet die Klassen zur Interaktion mit der Datenbank
- **resources**: Beinhaltet die Implementation der Resourcen der API

3.1.2 Resourcen

Die Resourcen der API werden in dem `resources`-Package implementiert.

Jede Resource ist eine eigene Klasse und wird mit der `@Path`-Annotation versehen, welche den Basis-Pfad der Endpunkte in dieser Resource vorgibt.

Alle Endpunkte dieser Resource werden als Methooden in der Klasse implementiert. Es gibt dafür verschiedene Annotationen, wie:

- `@GET / @POST / ...`: Legt die HTTP-Methode der Anfrage fest.
- `@Path`: Gibt den Pfad nach dem für die Resource deklarierten Basis-Pfad an.
- `@Consumes`: Den `MediaType` den der Body der Anfrage haben muss.
- `@Produces`: Den `MediaType` den die Antwort haben wird.

Die Methoden-Argumente werden aus den für die Argumente definierten Orten der Anfrage übernommen. Sollte kein Ort angegeben sein, nimmt das Framework dies aus dem Körper der Anfrage.

Alle nötigen Konvertierungen zwischen den Java Objekten und der Serializierung für die Anfragen/Antworten sind implizit und werden von dem Framework übernommen.

Listing 3.1: PUT /posts/{id} - Deklaration

```
@PUT
@Path("{postId}")
@Consumes(MediaType.APPLICATION_JSON)
@Produces(MediaType.APPLICATION_JSON)
@Secured
public PostDto updatePost(@PathParam("postId") long postId,
                          @Valid final PostDto post)
{...}
```

3.1.3 Datenbankkommunikation

Zur Kommunikation mit der Datenbank wird das Hibernate-Framework verwendet.
Die Entities sind in `model/entities` implementiert und haben Annotationen für die Relationen untereinander.
Jede Datenbanktabelle hat eine Implementierung in `repositories`, welche die nötigen Methoden zur Kommunikation implementiert.

3.1.4 Authentifizierung

Eine API-Key Authentifizierung ist nicht vom Framework implementiert, daher muss dies von uns implementiert werden. Dazu implementieren wir das `ContainerRequestFilter`-Interface um Anfragen ohne gültigen API-Key als Unauthentifiziert abzuweisen. Wir deklarieren, dass durch diese Authentifizierung geschützte Endpunkte mit der `@Secured` Annotation versehen werden.

3.2 ASP.NET Core

Als Vergleichsimplementierung wird das ASP.NET Core Framework verwendet.
Über den Verlauf dieser Sektion werden Vergleiche zur Referenzimplementierung (siehe 3.1) gezogen.

3.2.1 Projektstruktur

Das ASP.NET-Projekt gliedert sich in folgende Namespaces:

- **Controllers**: Beinhaltet die Implementation der Resourcen der API
- **Data**: Beinhaltet die Klassen zur Interaktion mit der Datenbank
- **Models**: Beinhaltet die DTOs und die Database-Entities.

3.2.2 Resourcen

Die Resourcen der API werden in dem `Controllers`-Namespace implementiert.
Jede Resource ist auch hier eine eigene Klasse. Diese erweitert hierbei die `ControllerBase`-Klasse und wird mit dem `Route`-Attribut versehen, welches analog zur `@Path`-Anotation in Java den Basispfad der Resource angibt.

Auch in ASP.NET sind die einzelnen Endpunkte der Resourcen als Methoden implementiert, und wie auch in Java mit Attributen versehen.

3.2.3 Datenbankkommunikation

3.2.4 Authentifizierung

Write

3.3 Fazit

Jakarta - Abhängigkeiten von Webservern Im Gegensatz zu ASP.NET oder Spring Boot benötigt Jakarta EE einen seperaten Webserver, um die Anwendung auszuführen.

Die Konfiguration und das Setup sind stark abhängig von dem verwendeten Webserver. In diesem Fall wurde `Wildfly`[1] verwendet.

Die meisten dieser Webserver sind nicht dazu ausgelegt, eine einzelne Anwendung zu hosten. Dies macht es schwieriger, die Anwendung in einer Microservice-Architektur zu betreiben.

Um dieses Projekt mit diesem Webserver zu betreiben, ist es notwendig, ein Konfigurations-Skript zu schreiben, welches in den Bauprozess des Docker-Abbilds integriert wird.

- Konfiguration und Setup ist stark abhängig von dem verwendeten Webserver (in diesem Fall `Wildfly`)

Write

[1] `https://www.wildfly.org/`

4 Tests

4.1 Funktionale Tests

Write

4.2 Load Tests

Während funktionale Tests die Korrektheit der Implementierung prüfen, evaluieren Load Tests die Performance des Systems unter hoher Last, beispielsweise bei vielen nebenläufigen Zugriffen.

Es gibt zahlreiche Gründe eine Anwendung Load Tests zu unterziehen. Diese sind unter anderem:

- Performance des Systems messen
- Bottlenecks identifizieren
- Skalierbarkeit des Systems bewerten
- Race Conditions entdecken

Zur Umsetzung dieser Tests wird JMeter & 5.4.1 & `https://jmeter.apache.org/` eingesetzt.

4.2.1 Starten von JMeter

JMeter kann im GUI- oder CLI-Modus gestartet werden. Der GUI-Modus wird zum Erstellen und Debuggen von Tests eingesetzt, welche anschließend im CLI-Modus ausgeführt werden können.

JMeter startet standardmäßig im GUI-Modus. Um Tests im CLI-Modus auszuführen, kann folgender Befehle eingesetzt werden.

```
jmeter -n -t testplan/TestPlan.jmx -l "results/result.jtl" -j "
    logs/logs.log"
```

Die eingesetzten Optionen haben folgende Bedeutung:

- `-n` - CLI-Modus
- `-t` - Pfad zum Testplan
- `-l` - Pfad zur Testlogdatei (Testergebnisse)
- `-j` - Pfad zur Jmeterlogdatei (Informationen über Testausführung und aufgetretene Fehler)

4.2.2 Erstellung von Testplänen

Tests werden in Form eines Testplans geschrieben. Ein Testplan ist ein Baum, der Elemente aus verschiedenen Kategorien enthalten kann (siehe `https://jmeter.apache.org/usermanual/test_plan.html`).

- Thread Group
  Kontrolliert wie oft Sampler ausgeführt werden, insbesondere wie viele Anfragen nebenläufig stattfinden.
- Logic Controllers
  Beeinflusst die Ausführungsreihenfolge der Sampler.
- Samplers
  Die auszuführenden Tests, z.B. HTTP-Anfragen.
- Listener
  Speichert die Ergebnisse eines Samplers und kann diese im GUI-Modus grafisch aufbereiten.
- Configuration Elements
  Stellen Daten für Tests zur Verfügung, unter anderem durch Definition von Konstanten, Erzeugung zufälliger Werte oder Einlesen externen Dateien.

Grundsätzlich beinhaltet ein Testplan `Thread Groups`. Jeder Thread steht für einen User. `Thread Groups` enthalten `Sampler`, welche von jeden Thread von oben nach unten ausgeführt werden. `Samplers` enthalten (einen) Listener, welcher die Ergebnisse aufzeichnet und im GUI-Modus visualisieren kann. `Configuration Elements` werden nach Bedarf als Kinder von `Thread Groups` oder `Samplers` definiert.

4.2.3 Verwendung von Variablen

JMeter ermöglicht die Definition von Variablen, um Tests besser wart- und anpassbar zu machen. Hierzu kann etwas das Configuration Element `User Defined Variables` eingesetzt werden. In diesem Element definierte Konstanten können grundsätzlich mit der Syntax `${Variablenname}` referenziert werden.

Leider wird diese Art Variablen zu referenzieren nicht überall unterstützt. Beispielsweise funktioniert dies im Körper einer HTTP Request und zur Angabe der Widerholungen eines Tests, aber nicht bei der Angabe von Dateipfaden.

In solchen Fällen kann der Wert einer Variable oftmals mit Groovy[1] ausgelesen werden. Dies ist mit folgender Syntax möglich: `${__groovy(vars.get("Variablenname"))}`. Leider funktioniert auch dieser Ansatz nicht in allen Fällen.

Der Einsatz von Groovy ermöglicht die Berechnung von Konstanten in Abhängigkeit von anderen Konstanten. Die abhängige Konstante muss in einem eigenen `User Defined Variables`-Element definiert werden. Dieses muss sich im Baum unter dem ursprünglichen Konfigurationselement befinden. Wird die Konstante im gleichen Element oder einem im Baum höherstehenden Element definiert, scheitert die Definition der abhängigen Variable und sie wird als leerer String behandelt.

Um Fehler beim Einsatz von Variablen zu untersuchen, kann der Sampler `Debug Sampler` eingesetzt werden. Dieser wertet bei Ausführung alle verfügbaren Variablen aus, sodass deren Werte überprüft werden können. Die Werte sind im GUI-Modus als `Response Data` des zum Sampler hinzugefügten Listener einsehbar.

[1] `https://www.groovy-lang.org/`

4.2.4 Relative Pfade

JMeter bietet keine vollständige Unterstützung für relative Pfade.

Die Angabe relativer Pfade ist im GUI-Modus nicht möglich. Ersetzt man im durch den GUI-Modus erzeugten Testplan, einer XML-Datei, absolute durch relative Pfade, so werden diese im CLI-Modus korrekt aufgelöst. Öffnet man den manipulierten Testplan erneut im GUI-Modus, kann JMeter die relative Pfade jedoch nicht auflösen.

Um trotzdem absolute Pfade zu vermeiden, kann ein absoluter Basispfad als Konstante definiert werden. Pfade werden dann als Konkatenation dieser Variable und des gewünschten relativen Pfades definiert. Somit muss nur der Basispfad angepasst werden, um die Ausführung auf einer anderen Maschine zu ermöglichen.

Listing 4.1: Workaround zur Angabe relativer Pfade

```
${__groovy(vars.get("PathBase"))}/listeners
```

4.3 Continuous Integration

Die Continuous Integration (CI)-Definitionen liegen im Verzeichnis `.github/workflows`.

Da wir nun alle Tests geschrieben haben, wollen wir dass diese automatisiert ausgeführt werden und Reports geben zum aktuellen Stand der Implementierung im Repository. Dazu benutzen wir **Github Actions**. Diese konfigurieren wir so, dass alle Tests (siehe 4.1 und 4.2) automatisch bei einem Git-Push auf das Repository ausgeführt werden.

Sollten Tests fehlschlagen wird dies automatisch durch diese Pipeline festgestellt und benachrichtigt den Autor des Commits. Des Weiteren generiert der Workflow für die Load-Tests einen HTML-Report mit Performance-Indikatoren und Graphen (siehe). Add reference

5 Fazit

Write

5.1 Zusammenfassung

Write

5.2 Lessons Learned

Write

5.3 Ausblick

Write

Abkürzungsverzeichnis

ER Entity Relationship

DTO Data Transfer Object

CI Continuous Integration