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, 29. September 2024

Inhaltsverzeichnis

1 Projekt

1.1 Zielsetzung

Gegenstand des Projekts war die Implementierung und das Testen einer REST-API sowie die Bewertung der eingesetzten Werkzeuge.

Die primäre Implementierung der API wurde mit Jakarta umgesetzt. Um eine fundierte Bewertung des Frameworks zu ermöglichen, wurde eine alternative Implementierung mit ASP.NET umgesetzt.

Neben der Implementierung der API bestand ein wesentlicher Bestandteil des Projekts in der Automatisierung von Tests zur Sicherstellung der Korrektheit und Zuverlässigkeit der API. Dies umfasste sowohl funktionale Tests, als auch Load Tests. Besonders sollte hier auf die Technologien zur Automatisierung solcher Tests konzentriert werden.

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)

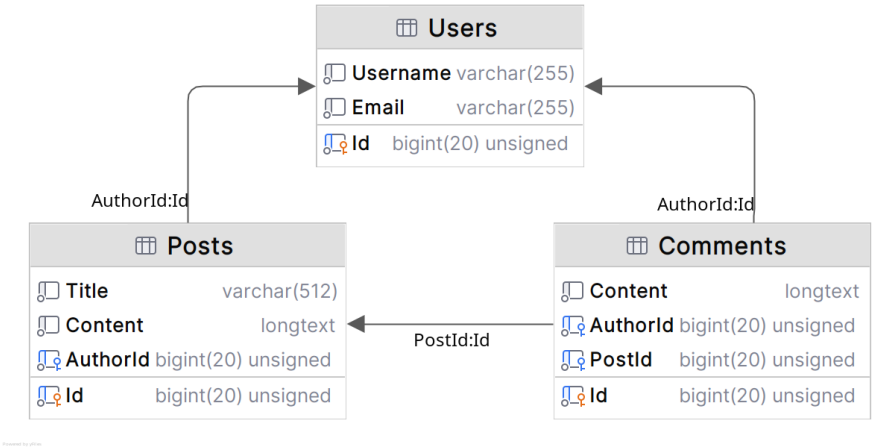

Abbildung 2.1: ER-Diagram

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 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

Das Jakarta Projekt befindet sich im Ordner: **rest-jakarta**

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 Resourcen

Die Resourcen der API sind im **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)
{...}
```

3.1.2 Datenbankkommunikation

Datenbank-Entities befinden sich im Package **model/entities**;
Klassen zur Kommunikation mit der Datenbank befinden sich im **repositories**-*Package*

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

3.1.3 Authentifizierung

Die Klassen zur Authentifizierung sind im Package **middlewares**

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

Das ASP.NET Projekt befindet sich im Ordner: **rest-aspnet/Rest.AspNet**

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 Resourcen

Die Resourcen der API werden im **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.

3.2.2 Datenbankkommunikation

Datenbank-Entities befinden sich im Namespace **Model**;
Klassen zur Kommunikation mit der Datenbank befinden sich im **Data**-*Namespace*

In ASP.Net übernimmt die Datenbankkommunikation `EntityFramwork` (EF).
Wie auch in Hibernate werden die Models mit Attributen versehen, welche die Datenbankstruktur wiederspiegeln sollen, allerdings wird das Meiste automatisch erkannt (so zum Beispiel die Referenzen zwischen den Entities) und wir benötigen nur ein Attribut für die automatisierte ID-Generation: `[DatabaseGenerated(DatabaseGeneratedOption.Identity)]`

Im Gegensatz zu Hibernate müssen keine Klassen für jede Interaktion mit den Entities geschrieben werden, sondern EF erkennt die Zusammenhänge aus den Entities und stellt `LINQ`-Methoden[1] zu Verfügung.

[1] `https://learn.microsoft.com/en-us/dotnet/csharp/linq/`

3.2.3 Authentifizierung

Die Authentifizierung ist in der Startklasse **Program.cs** *implementiert.*

Authentifizierung ist deutlich komplexer in ASP.NET als in Jakarta. Dies liegt größtenteils daran, dass ASP.NET bereits ein komlexes System für Authentifizierung und Authorisierung enthält und einbindet. Dementsprechend schwierig ist es, auf diesem System eine API-Key-Authentifizierung zu schreiben. Daher haben wir uns dazu entschieden eine externe Implementierung für die API-Key Authentifizierung zu benutzen.

3.3 Probleme

Dependency Management - Maven `Maven` als Build-Managment-Tool ist leider nicht in Java integriert und muss separat installiert werden. Des Weiteren ist die Konfiguration von Maven deutlich komplexer als die von `.NET Core`. In Dotnet Core ist das Dependency-Management in der `.csproj`-Datei integriert und wird größtenteils (durch die IDE) automatisch verwaltet. Zusätzlich ist die Konfiguration in der `.csproj`-Datei deutlich einfacher und übersichtlicher als die in der `pom.xml`-Datei.

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`[2] verwendet.

Die meisten dieser Webserver sind nicht dazu ausgelegt, eine einzelne Anwendung zu hosten. Dies macht es schwieriger, die Anwendung isoliert zu betreiben, wie etwa in einer Micro-Servicearchitektur mit Docker (wie wir es umsetzen wollten). Wir haben dazu das offizielle `Wildfly` Docker Image benutzt, allerdings hat auch dieses seine Schwächen. Alle Konfiguration bezüglich Datenbanken lässt sich nur in Wildfly vornehmen. Selbst wenn die Applikation die Datenbanken selbst für sich konfiguriert, wird diese Konfiguration von Wildfly überschrieben. Dies bedeutet, dass diese Service-Referenzen nur in Wildfly vorgenommen werden können. Wir haben im Rahmen dieses Projektes dies nicht automatisiert (für das Docker-Deployment) umsetzen können und daher sind wir von der ursprünglich geplanten MariaDB-Datenbank auf die In-Memory-Datenbank H2 umgestiegen.

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

4 Tests

4.1 Funktionale Tests

Für die Durchführung der funktionalen Tests wurde *Postman* verwendet.

Während funktionale Tests darauf abzielen, die Korrektheit der Implementierung zu verifizieren, liegt der Fokus darauf, sicherzustellen, dass die Application Programming Interface (API)-Endpunkte die spezifizierten Anforderungen in verschiedenen Anwendungsfällen korrekt erfüllen. Funktionale Tests umfassen unter anderem folgende Ziele:

- Validierung der HTTP-Statuscodes
- Überprüfung der Struktur und Inhalte der Rückgaben
- Behandlung von Randfällen und fehlerhaften Anfragen
- Test von Autorisierungsmechanismen

4.1.1 Postman

Postman ist ein API-Client, der es ermöglicht, HTTP-Anfragen an eine API zu senden und die Antworten zu überprüfen. Es bietet eine grafische Benutzeroberfläche, um Anfragen zu erstellen und zu senden, sowie die Möglichkeit, Testfälle zu definieren und zu automatisieren. Besonders nützlich ist die Möglichkeit, sogenannte *Collections* zu erstellen, die mehrere Anfragen und Testfälle enthalten können, welche dann in einer Sequenz ausgeführt werden. Diese *Collections* können als JavaScript Object Notation (JSON)-Dateien exportiert und geteilt werden, um Tests zu reproduzieren oder zu teilen.

Test Aufbau in Postman

Jede Anfrage, Ordner, oder Collection in Postman kann mit sogennannten *Pre-Request* und *Post-Request* Skripten versehen werden, welche in JavaScript geschrieben sind. Diese werden vor bzw. nach der eigentlichen Anfrage/Liste von Anfragen ausgeführt und können dazu verwendet werden, um Variablen zu setzen, oder die Antwort zu verarbeiten. In den *Post-Request* Skripten werden die Tests mit dem eingebauten Test-Runner von Postman geschrieben, welcher es ermöglicht, die Antwort auf verschiedene Kriterien zu prüfen.
Generel unterscheidet man zwischen zwei Arten von funktionalen Tests für APIs: *Endpoint Monitoring* und *Scenario Testing*.

4.1.2 Endpoint Monitoring

Beim *Endpoint Monitoring* werden die Endpunkte der API einzeln nach dem Schema der Spezifikation betrachtet. Es wird also für jeden Endpunkt ein Testfall erstellt, der die möglichen Statuscodes und Rückgabewerte überprüft. Zusätzlich wird auch die verwendete Authentifizierungsmethode überprüft. Hierbei werden die *Pre-Request* Skripte verwendet, um eventuelle Abhängigkeiten zwischen den Anfragen zu lösen, z.B. wird ein Post erstellt, damit für diesen ein Kommentar in der zu testenden Anfrage erstellt werden kann.

Als Beispiel wird der Endpoint *PUT /posts/postId/comments/commentId* betrachtet, welcher einen Kommentar eines Posts bearbeitet. Die möglichen Ergebnisse der Anfrage sind:

- **200 OK** - Der Kommentar wurde erfolgreich bearbeitet
- **401 Unauthorized** - Der Benutzer ist nicht autorisiert
- **404 Not Found** - Der Kommentar oder der Post existiert nicht

200 OK

Damit die Anfrage ausgeführt werden kann, muss jeweils ein Nutzer, ein Post und ein Kommentar erstellt werden. Dies wird in den *Pre-Request* Skripten erledigt. Wie in Listing A.1 zu sehen, wird zuerst ein Nutzer erstellt, dann ein Post und schließlich ein Kommentar. Dazu werden die benötigten Strings, wie Nutzername, Email, Titel und Inhalt des Posts, sowie der Inhalt des Kommentars, generiert. Diese werden dann in den Anfragen verwendet, welche den entsprechenden Nutzer, Post und Kommentar erstellen. Die IDs der erstellten Objekte werden in Variablen gespeichert, um sie in der eigentlichen Anfrage zu verwenden. Der Body der Anfrage enthält den neuen Inhalt des Kommentars, welcher im *Pre-Request* Skript als Variable erstellt wurde (Listing 4.1). Die Authorisierung erfolgt über einen *API-Key*, welcher in einer Variable gespeichert ist (Abbildung 4.1).

Listing 4.1: Request Body für die Bearbeitung eines Kommentars

```
{
    "content": "{{updatedCommentContent}}",
    "authorId": {{userId}}
}
```

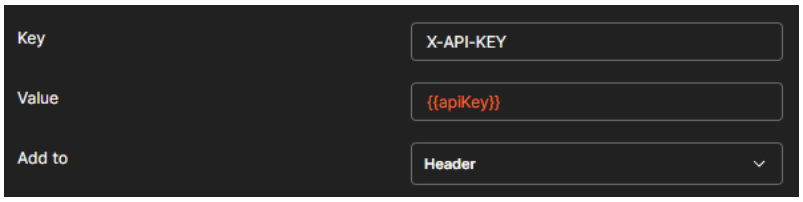

Abbildung 4.1: Authentifizierung in Postman

Zum Testen der Anfrage wird im *Post-Request* Skript ein Testfall erstellt, welcher die Antwort auf den Statuscode 200 OK überprüft. Außerdem wird bestätigt, dass der Inhalt des Kommentars aktualisiert wurde (Listing 4.2).

Listing 4.2: *Post-Request* Skript für die Bearbeitung eines Kommentars

```
pm.test("Status code is 200", function () {
    pm.response.to.have.status(200);
});

// Test to check if the response contains updated comment data
pm.test("Response contains updated comment data", function () {
    var jsonData = pm.response.json();
    pm.expect(jsonData).to.have.property("id", parseInt(pm.
        collectionVariables.get("commentId")));
    pm.expect(jsonData).to.have.property("content", pm.
        collectionVariables.get("updatedCommentContent"));
```

```
    pm.expect(jsonData).to.have.property("authorId", parseInt(pm.
        collectionVariables.get("userId")));
    pm.expect(jsonData).to.have.property("postId", parseInt(pm.
        collectionVariables.get("postId")));
});
```

401 Unauthorized

Um den Statuscode 401 Unauthorized zu testen, wird die Anfrage mit dem falschen *API-Key* gesendet. Zum Schluss muss dann nur der Statuscode getestet werden (Listing 4.3).

Listing 4.3: *Post-Request* Skript für 401 Unauthorized

```
pm.test("Status code is 401", function () {
    pm.response.to.have.status(401);
});
```

404 Not Found

Um den Statuscode 404 Not Found zu testen, wird die Anfrage einmal mit ungültiger Post-ID und einmal mit ungültiger Kommentar-ID gesendet. Die Tests sind identisch zu denen für 401 Unauthorized, nur dass der Statuscode 404 getestet wird (Listing 4.4).

Listing 4.4: *Post-Request* Skript für 404 Not Found

```
pm.test("Status code is 404", function () {
    pm.response.to.have.status(404);
});
```

4.1.3 Scenario Testing

Beim *Scenario Testing* werden mehrere Endpunkte in einer Sequenz getestet, um die Funktionalität der API in verschiedenen Anwendungsfällen zu überprüfen. Dies ist oft bei komplexeren Anwendungsfällen nützlich, bei denen mehrere Endpunkte zusammenarbeiten müssen, um das gewünschte Ergebnis zu erzielen. So können besonders kritische Pfade in der API getestet werden, um sicherzustellen, dass sie korrekt funktionieren. Dies ist vor allem notwendig bei Anwendungen, die auf einer umfangricheren API basieren, wie z.B. Webanwendungen oder mobile Apps. Ein Beispiel für ein simples Szenario wäre das “Erstellen und Abrufen eines Posts”:

1. **Ein Nutzer erstellt einen Post - POST /posts**
   - existiert der Nutzer?
   - sind alle Parameter vorhanden?
   - ist der Inhalt des Posts valide?
   - wird der Post erstellt?
2. **Der erstellte Post wird abgerufen - GET /posts/postId**
   - existiert der Post?
   - ist der Inhalt des Posts korrekt?

3. **Der erstellte Post wird gelöscht - DELETE /posts/postId**
    - wird der Post gelöscht?

Da die API in diesem Projekt relativ einfach ist, wurden keine komplexen Szenarien getestet, sondern nur die einzelnen Endpunkte.

4.1.4 Automatisierung und Variablen

Um eine effiziente Testausführung zu gewährleisten, wurden in Postman Variablen verwendet, um Werte wie *postId* und *authorId* zwischen den Testläufen dynamisch zu speichern. Dies ermöglichte eine Automatisierung der Testsequenzen, bei denen Ergebnisse aus einem Testlauf in den nächsten übernommen wurden. Zudem konnten Fehler bei ungültigen Eingaben erkannt werden, z.B. bei fehlenden Parametern oder unzulässigen Datentypen. Die Verwendung von Postman ermöglichte die Durchführung dieser Tests sowohl manuell als auch automatisiert, wobei alle Testfälle erfolgreich durchlaufen wurden und die Funktionalität der API bestätigten. Für die automatisierte Ausführung der Tests wurde eine *Continuous Integration (CI)* Pipeline in Github Actions erstellt, welche bei jedem Push ausgeführt wird (siehe Continous Integration 4.3)./

4.1.5 Bewertung

Postman erweist sich als ein benutzerfreundliches und leistungsstarkes Werkzeug für API-Tests. Es bietet eine Vielzahl von Funktionen, die die Erstellung, Ausführung und Automatisierung von Tests erleichtern. Besonders hervorzuheben ist die Möglichkeit, Variablen zu verwenden, um Werte wie *postId* und *authorId* zwischen den Testläufen dynamisch zu speichern. Außerdem ist die Vielseitigkeit von *Collections* zur Strukturierung und Organisation von Testfällen sehr nützlich. Dies ermöglicht eine effiziente und automatisierte Testausführung. Zudem können Fehler bei ungültigen Eingaben, wie fehlenden Parametern oder unzulässigen Datentypen, leicht erkannt werden.

Allerdings bringt das Always-Online-Modell von Postman auch einige Nachteile mit sich. Die ständige Abhängigkeit von der Postman API kann störend sein, insbesondere bei der Integrierung in CI-Pipelines. Zudem schränkt das Bezahlmodell die Anzahl der täglichen Testausführungen in der kostenlosen Version ein, was bei umfangreicheren Projekten zu erheblichen Einschränkungen führen kann.

Angesichts dieser Einschränkungen wäre es sinnvoll, für zukünftige Entwicklungen auch alternative Tools in Betracht zu ziehen. Tools wie *Newman*, die CLI-Version von Postman, welche in der CI Pipeline des Projekts verwendet wurde, war hier eine nötige Ergänzung.

Andere API-Testwerkzeuge wie *Hoppscotch*, *Bruno*, *Insomnia*, oder *Katalon Studio* könnten eine gute Alternative darstellen, um die Abhängigkeit von der Postman Struktur zu reduzieren und die Flexibilität bei der Testausführung zu erhöhen.

Insgesamt bietet Postman eine solide Grundlage für API-Tests, jedoch sollten die genannten Einschränkungen bei der Planung und Durchführung von Tests berücksichtigt werden.

4.2 Load Tests

Zur Umsetzung des Load Testings wurde Apache JMeter eingesetzt.

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:

- Messung der Systemperformance
- Identifikation von Bottlenecks
- Untersuchung der Skalierbarkeit des Systems
- Entdeckung von Race Conditions

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 Befehl eingesetzt werden.

Listing 4.5: Beispielkonfiguration des CLI-Modus

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

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)
- `-e` - Report generieren
- `-o` - Pfad für Reportdateien
- `-f` - Alte Ergebnisse und Reports löschen

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 Groups kontrollieren wie oft Sampler ausgeführt werden, insbesondere wie viele Anfragen nebenläufig stattfinden.

- Logic Controller beeinflussen die Ausführungsreihenfolge der Sampler.
- Sampler enthalten die auszuführenden Tests, z.B. HTTP-Anfragen.
- Listener speichern die Ergebnisse eines Samplers und können 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.

Die Elemente können auf unterschiedliche Weise verschachtelt werden, gängig ist dabei folgende Reihenfolge:

- Der Testplan enthält `Thread Groups`, wobei jeder Thread einer Gruppe einen User darstellt.
- `Thread Groups` enthalten `Sampler`, welche von jedem Thread der Gruppe 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`, `Samplers` oder dem Wurzelelement des Testplans definiert.

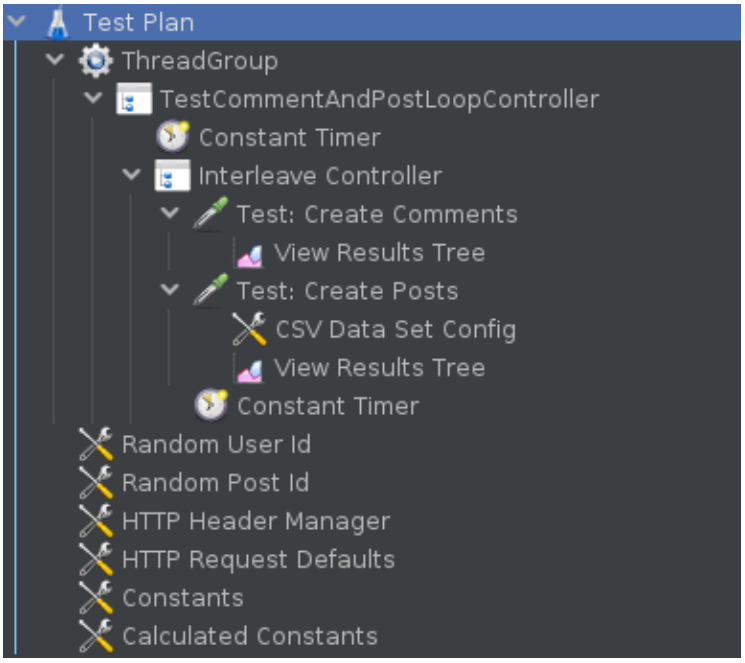

Abbildung 4.2: Beispiel eines Testplans

4.2.3 Report Dashboard

JMeter kann die erfassten Leistungsparameter in einem HTML-Dashboard zusammenfassen. Dieses besteht aus verschiedenen Graphen, welche etwa Antwortzeiten und Datendurchsatz über den Verlauf des Tests darstellen. Um einen Report zu generieren, muss

die Option `-e` benutzt werden. Das Dashboard ist direkt ohne weitere Konfiguration benutzbar und sollte für die meisten Projekte ausreichend sein. Nur in fortgeschritteneren Anwendungsfällen, etwa zur automatischen Auswertung, ist die Verwendung von Listenern mit eigenen Result-Dateien notwendig.

JMeter unterstützt Anpassungen an der Graphgenerierung, etwa die Anpassung der zeitlichen Auflösung. Weiterhin können eigene Graphen hinzugefügt werden. Siehe hierzu `https://jmeter.apache.org/usermanual/generating-dashboard.html`.

Ein Beispielreport befindet sich im Ordner `jmeter/ReportBeispiel`.

4.2.4 Verwendung von Variablen und Scripting

JMeter ermöglicht die Definition von Variablen und Skripten, um Tests besser wart- und anpassbar zu machen.

Zur Definition von Variablen kann etwa das Configuration Element `User Defined Variables` eingesetzt werden. In diesem Element definierte Konstanten können grundsätzlich mit der Syntax `${Variablenname}` referenziert werden, was allerdings nicht von allen Eingabefeldern unterstützt wird. So funktioniert dies etwa im Körper einer HTTP Request oder zur Angabe der Wiederholungen einer Threadgroup, nicht aber zur Angabe von Dateipfaden für Sampler.

JMeter unterstützt mehrere Skriptinterpreter (Beanshell, Groovy, Javascript, Jexl). Die Dokumentation empfiehlt die Verwendung von Groovy oder Jexl3. Skripte können statt Literalen zur Definition verschiedener Parameter des Testplans eingesetzt werden. Die Syntax zur Ausführung von Skripten folgt diesem Schema: `${__interpreter(script)}`.

Wird die Verwendung einer Variable mittels `${Variablenname}` nicht direkt unterstützt, kann der Wert der Variable oftmals durch ein Skript ausgelesen werden. Hierzu kann darin die Funktion `vars.get("Variablenname")` benutzt werden. Leider funktioniert auch dieser Ansatz nicht in allen Fällen.

Scripting 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 Auflösung der abhängigen Variable und sie wird als leerer String behandelt.

Listing 4.6: Beispiel eines Groovy-Skripts

```
${__groovy(
    vars.get("UserCount").toInteger()
    * vars.get("PostsPerUserCount").toInteger()
)}
```

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 unter `Response Data` des zum Sampler hinzugefügten Listener einsehbar.

4.2.5 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 Basispfad als Variable definiert werden. Pfade werden dann als Konkatenation dieser Variable und des gewünschten relativen Pfades definiert. Der Basispfad kann als Literal oder durch ein Skript definiert werden.

Listing 4.7: Ermitteln des Working Directory[1]

```
${__BeanShell(
    import org.apache.jmeter.services.FileServer;
    FileServer.getFileServer().getBaseDir();
)}
${__BeanShell(File.separator,)}
```

4.2.6 Bewertung

JMeter stellt trotz einiger Mängel eine sinnvolle Wahl zur Erfassung und Überwachung von Leistungsparametern dar.

Der Aufbau und damit die Erstellung von Testplänen ist nicht intuitiv, weshalb eine explizite Einarbeitung in das Werkzeug notwendig ist. Auch nach dieser Einarbeitung überraschen einige Fallstricke und Eigenheiten neue Anwender, etwa die erwähnten Probleme beim Einsatz von Variablen oder relativen Pfaden. Dank des Scripting-Supports lassen sich die meisten dieser Einschränkungen umgehen, dies stellt jedoch einen anfangs unerwarteten Mehraufwand dar.

Hat man den Aufbau der Testpläne jedoch verstanden und alle anfänglichen Hürden überwunden, wird man von JMeter mit einer anpassbaren und überschaubaren Zusammenfassung der Leistungsparameter in Form des HTML-Reports belohnt. Daten der einzelnen Sampler können hierbei direkt in dieser Ansicht ein- und ausgeblendet werden, sodass stets ein guter Überblick ermöglicht wird. Für komplexere Anwendungsfälle können auch eigene Graphen konfiguriert werden.

Dank der hohen Zahl unterstützter Protokolle ist JMeter für viele Projekte geeignet. Die verschiedenen `Logic Controller` und `Configuration Elements` ermöglichen die Abbildung komplexer Anwendungsfälle, wenngleich die Erstellung solcher Testpläne nicht trivial ist.

Zusammengefasst kann JMeter zur Durchführung von Load Tests empfohlen werden. Aufgrund der zwingend notwendigen Einarbeitung und verschiedener Eigenheiten des Werkzeugs sollte die Zuständigkeit zur Erstellung und Wartung der Testpläne jedoch ausgewiesenen Projektmitgliedern übertragen werden.

4.3 Continuous Integration

Die 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.

[1] Quelle: `https://stackoverflow.com/questions/11268015`

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 4.2.3).

5 Fazit

Im Rahmen des Projekts wurde erfolgreich eine REST-API implementiert und getestet.

Zunächst wurde die API unter Einsatz von OpenAPI konzeptioniert und definiert. Die hierbei entstandene Spezifikation führt alle Endpunkte sowie ihre Antworten auf. Basierend auf der OpenAPI-Spezifikation wurde die API anschließend in Jakarta und ASP.NET implementiert. Zur Implementierung unter Jakarta wurde Wildfly als Server eingesetzt. Die Korrektheit der Implementierung wurde unter Einsatz von Postman entsprechend der Spezifikation geprüft. Zusätzlich wurde ein JMeter-Testplan zur Erfassung von Leistungsdaten entwickelt. Die Postman und JMeter Tests wurden in einer CI-Pipeline automatisiert ausgeführt.

Die Implementierung mit Jakarta wies einige Tücken auf. Insbesondere durch das Fehlen eines integrierten webservers und Probleme bei der Konfiguration einer Datenbank unter Wildfly. Unter ASP.NET ging die Implementierung größtenteils problemlos von Statten. Dies kann in Teilen aber auch auf unsere größere praktische Erfahrung im Umgang mit ASP.NET zurückgeführt werden.

Postman erweist sich als ein benutzerfreundliches und leistungsstarkes Werkzeug für API-Tests, allerdings kann das Always-Online-Modell störend sein, da es eine ständige Abhängigkeit von der Postman API mit sich bringt. Zudem schränkt das Bezahlmodell die Anzahl der täglichen Testausführungen in der kostenlosen Version ein, was bei umfangreicheren Projekten zu erheblichen Einschränkungen führen kann. Angesichts dessen wäre es sinnvoll, für zukünftige Entwicklungen auch alternative Tools in Betracht zu ziehen.

JMeter kann grundsätzlich trotz einiger Fallstricke und Einschränkungen empfohlen werden. Das Werkzeug erfordert eine explizite Einarbeitung, kann nach dieser jedoch für viele Testszenarien eingesetzt werden. Die Generierung des HTML-Reports funktioniert out of the box und bereitet die erfassten Leistungsparameter sinnvoll auf. Daten der verschiedenen Sampler können direkt im HTML-Report ein- und ausgeblendet werden, sodass der Report einen guten Überblick bietet.

Abkürzungsverzeichnis

API	Application Programming Interface
ER	Entity Relationship
DTO	Data Transfer Object
CI	Continuous Integration
JSON	JavaScript Object Notation

Anhang

A Listings

Listing A.1: Pre-Request Skript für die Erstellung eines Benutzers, Posts und Kommentars

```
function generateRandomString(length) {
    var chars = '
       abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ0123456789
       ';
    var result = '';
    for (var i = 0; i < length; i++) {
        result += chars.charAt(Math.floor(Math.random() * chars.
           length));
    }
    return result;
}

// Generate random username and email for user creation
var randomUsername = generateRandomString(8);
var randomEmail = randomUsername + '@example.com';

// Generate random title and content for post creation
var randomTitle = "Post " + generateRandomString(5);
var randomContent = "Content for post " + generateRandomString(5);

// Generate random content for comment creation
var randomComment = "Comment " + generateRandomString(5);

// Generate random content for comment update
var updatedCommentContent = "Updated comment " +
   generateRandomString(5);

// Store the updated values in collection variables for later verification
pm.collectionVariables.set("updatedCommentContent",
   updatedCommentContent);

// Step 1: Create a user
pm.sendRequest({
    url: pm.globals.get("baseUrl") + '/users',
    method: 'POST',
    header: {
        'Content-Type': 'application/json',
        'X-API-KEY': pm.collectionVariables.get("apiKey")
    },
    body: {
        mode: 'raw',
        raw: JSON.stringify({
            username: randomUsername,
            email: randomEmail
        })
    }
}, function (err, res) {
```

```
    if (err) {
        console.log("Error creating user:", err);
    } else {
        var userId = res.json().id;
        pm.collectionVariables.set("userId", userId);

        // Step 2: Create a post for the user
        pm.sendRequest({
            url: pm.globals.get("baseUrl") + '/posts',
            method: 'POST',
            header: {
                'Content-Type': 'application/json',
                'X-API-KEY': pm.collectionVariables.get("apiKey")
            },
            body: {
                mode: 'raw',
                raw: JSON.stringify({
                    title: randomTitle,
                    content: randomContent,
                    authorId: userId
                })
            }
        }, function (err, res) {
            if (err) {
                console.log("Error creating post:", err);
            } else {
                var postId = res.json().id;
                pm.collectionVariables.set("postId", postId);

                // Step 3: Create a comment on the post
                pm.sendRequest({
                    url: pm.globals.get("baseUrl") + '/posts/' +
                       postId + '/comments',
                    method: 'POST',
                    header: {
                        'Content-Type': 'application/json',
                        'X-API-KEY': pm.collectionVariables.get("
                           apiKey")
                    },
                    body: {
                        mode: 'raw',
                        raw: JSON.stringify({
                            content: randomComment,
                            authorId: userId
                        })
                    }
                }, function (err, res) {
                    if (err) {
                        console.log("Error creating comment:", err)
                           ;
                    } else {
                        var commentId = res.json().id;
                        pm.collectionVariables.set("commentId",
                           commentId);
                        console.log("Comment created successfully
                           with ID:", commentId);
                    }
```

```
                    });
                }
            });
    }
});
```