1 Ein einfacher HTTP Server

Alle SmolHTTPd-Klassen befinden sich im Paket *de.troebs.smol*. Als Antwortformat kommt grundsätzlich JSON zum Einsatz. Zur Verarbeitung wird eine weitere Bibliothek als Abhängigkeit benötigt - „Jackson“. [1] Diese unterteilt sich in drei verschiedene JAR Archive. [2][3][4]

Zum Öffnen eines einfachen HTTP Servers genügt das Erzeugen eines *SmolHTTPd* Objekts und das Registrieren eines Handlers für einen bestimmten Pfad. Der Handler muss eine Methode enthalten, die mit einer zur Anfragemethode passenden Annotation versehen ist. Die unterstützten Methoden umfassen „GET“, „POST“, „PUT“, „PATCH“ und „DELETE“.

```
 1 class Handler() {
 2     @Get
 3     public int get() { return 1; }
 4
 5     @Post
 6     public int post() { return 2; }
 7
 8     @Put
 9     public int put() { return 3; }
10
11     @Patch
12     public String patch() { return "test"; }
13
14     @Delete
15     public boolean delete() { return true; }
16 }
17
18 SmolHTTPd server = new SmolHTTPd(8080);
19 server.register("/", new Handler());
```

Der Server wandelt die Rückgabe der Methode in ein JSON Objekt um und liefert es als Antwort zurück. Sollte es sich bei der Rückgabe um einen primitiven Datentyp oder einen String handeln, wird diese in einer Wrapper-Klasse verpackt. Wenn kein Handler unter dem angefragten Pfad gefunden wird, gibt der Server automatisch den Status 404 *Not Found* zurück.

Die Klasse SmolHTTPd implementiert zudem das „AutoCloseable“ Interface.

2 Parameter in der URL

Die Bibliothek bietet eine einfache Möglichkeit, Parameter aus der aufgerufenen Ressource zu extrahieren. Im Beispiel sollen die Nachrichten eines bestimmten Nutzers unter der Ressource */users/1/messages* abgerufen und dazu die URL verarbeitet werden. In der registrierten URL können dafür benannte Platzhalter in geschweiften Klammern angelegt und über die Annotation „Param“ in der Handlermethode referenziert werden.

Unterstützt und automatisch umgewandelt werden primitive Datentypen, Strings und Arrays dieser. Optionale Parameter oder Überdeckungen können durch die Verwendung mehrerer Methoden erreicht werden. Groß- und Kleinschreibung werden unterschieden.

```
1 class Handler {
2     @Get
3     public String get(@Param("id") int id) {
4         return String.format("keine von %d gefunden", id);
5     }
6 }
7
8 SmolHTTPd server = new SmolHTTPd(8080);
9 server.register("/users/{id}/messages", new Handler());
```

3 Parameter in Headern

Zusätzlich lassen sich auch die Werte bestimmter Header auslesen. Dazu steht die Annotation „Header“ zur Verfügung, die als Wert den Schlüssel des Headers erwartet. Sie funktioniert analog zu *Param* in Abschnitt 2, wobei auch hier Groß- und Kleinschreibung unterschieden wird.

Unterstützt und automatisch umgewandelt werden primitive Datentypen, Strings und Arrays dieser.

```
1 class Handler {
2     @Get
3     public String get(@Header("User-Agent") String agent) {
4         return agent;
5     }
6 }
7
8 SmolHTTPd server = new SmolHTTPd(8080);
9 server.register("/", new Handler());
```

4 HTTP Basisauthentifizierung

Zur einfacheren Verarbeitung des Authorization Headers unterstützt die Bibliothek die *HTTP Basic Authentication*. Übergebene Benutzername-Passwort-Kombinationen werden entsprechend des Standards automatisch dekodiert und verarbeitet. [5]

Dazu muss der Typ des Authorization Headers als „BasicAuthCredentials“ definiert werden. Über die finalen Eigenschaften „username“ und „password“ des Objekts können die entsprechenden Daten abgefragt werden. Wurden vom Client keine oder nur solche im falschen Format bereitgestellt, so sind beide Werte *null*.

```
1  class Handler {
2      @Get
3      public String get(
4          @Header("Authorization") BasicAuthCredentials auth
5      ) {
6          return auth.username + " | " + auth.password;
7      }
8  }
9
10 SmolHTTPd server = new SmolHTTPd(8080);
11 server.register("/", new Handler());
```

5 Parameter im Body

Die Bibliothek bietet mehrere Möglichkeiten den Body der Anfrage auszulesen und zu verarbeiten. Die entsprechende Variante wird über separate Annotations ausgewählt:

1. Parameter, die im Format *key1=value1&key2=value2* übergeben werden, lassen sich mithilfe der Annotation „BodyParam“ auslesen. Unterstützt und automatisch umgewandelt werden primitive Datentypen, Strings und Arrays dieser.

   ```
   1  class Handler {
   2      @Post
   3      public String post(@BodyParam("key1") String val) {
   4          return val;
   5      }
   6  }
   7
   8  SmolHTTPd server = new SmolHTTPd(8080);
   9  server.register("/", new Handler());
   ```

2. Alternativ kann der Body der Anfrage als Stream angefordert werden.

```
1 class Handler {
2     @Post
3     public int post(@BodyStream InputStream stream) {
4         return stream.available();
5     }
6 }
7
8 SmolHTTPd server = new SmolHTTPd(8080);
9 server.register("/", new Handler());
```

3. Ebenfalls ist es möglich, den gesamten Body als String zu verarbeiten.

```
1 class Handler {
2     @Post
3     public String post(@BodyString String body) {
4         return body;
5     }
6 }
7
8 SmolHTTPd server = new SmolHTTPd(8080);
9 server.register("/", new Handler());
```

4. Analog zum Auslesen als String kann der Inhalt des Body auch als Byte Array empfangen werden.

```
1 class Handler {
2     @Post
3     public String post(@BodyBytes byte[] body) {
4         return new String(body);
5     }
6 }
7
8 SmolHTTPd server = new SmolHTTPd(8080);
9 server.register("/", new Handler());
```

5. Zuletzt besteht ebenfalls die Möglichkeit die empfangenen Daten direkt als JSON Objekt zu verarbeiten. Dazu wird die ebenfalls benötigte Bibliothek Jackson und ihre eigenen Annotations verwendet.

```
1 class Request {
2     public int id;
3     public String name;
4 }
```

```
5
6  class Handler {
7      @Post
8      public int post(@BodyJSON Request request) {
9          return request.id;
10     }
11 }
12
13 SmolHTTPd server = new SmolHTTPd(8080);
14 server.register("/", new Handler());
```

6 Fehler und Verwendung von Statuscodes

Fehler innerhalb der Handler-Methode werden durch Exceptions ausgedrückt. Eine Handler-Methode darf jede Art von Exception werfen, auch selbst definierte oder solche vom Typ RuntimeException. Anfragenden Clients werden aufgetretene Fehler durch den HTTP Statuscode 500 *Internal Server Error* und einen leeren Body der Antwort symbolisiert.

```
1  class Handler {
2      @Get
3      public String get() throws IOException {
4          throw new IOException();
5      }
6  }
7
8  SmolHTTPd server = new SmolHTTPd(8080);
9  server.register("/", new Handler());
```

Bei einer erfolgreichen Bearbeitung der Anfrage wird mit dem Statuscode 200 *OK* geantwortet. Soll ein anderer Code zur Symbolisierung eines Fehlers verschickt werden, geschieht dies über spezielle Klassen, die entweder als Exception geworfen oder als Ergebnis der Handler-Funktion zurückgegeben werden können. Zu den meisten offiziellen Statuscodes existiert eine entsprechende gleichnamige Klasse. [6]

Einige Codes benötigen zur korrekten Funktion weitere Header mit genaueren Details. Die Konstruktoren der entsprechenden Klassen besitzen dazu zusätzliche Parameter. 301 *Moved Permanently* erfordert beispielsweise einen „Location“-Header mit einer URL, an der die Ressource mittlerweile zu finden ist. 401 *Unauthorized* dagegen erfordert einen komplexeren Header, der sowohl die Authentifizierungsmethode als auch den Namen des geschützten Bereichs („realm“) beinhaltet. Die Bibliothek vereinfacht dies und fordert nur den Namen des Bereichs.

```
1  class Handler {
```

```
 2     @Get
 3     public Object get() {
 4         return new Unauthorized("geschützter Bereich");
 5     }
 6
 7     @Post
 8     public Object post() throws Unauthorized {
 9         throw new Unauthorized("geschützter Bereich");
10     }
11 }
12
13 SmolHTTPd server = new SmolHTTPd(8080);
14 server.register("/", new Handler());
```

7 Priorität von Handler-Methoden

In einigen Fällen kann es notwendig sein, mehrere überlappende Handler-Funktionen innerhalb einer Handler-Klasse zu definieren. Im Gegensatz zu registrierten Objekten kann die Reihenfolge von Funktionen innerhalb von Klassen aufgrund einiger Eigenheiten der Java Plattform nicht festgestellt werden. [7]

Die Annotations *GET*, *POST*, *PUT*, *PATCH* und *DELETE* besitzen deshalb einen zusätzlichen optionalen Parameter „priority“, der den zugehörigen Methoden eine Priorität zuweist. Ist dieser Parameter nicht angegeben wird eine Priorität von 0 angenommen. Je höher der Wert ist, desto eher wird die Methode zur Ausführung in Betracht gezogen.

```
 1 class Handler {
 2     @Get(priority = 5)
 3     public String get1() {
 4         return "priority 5";
 5     }
 6
 7     @Get // priority = 0
 8     public String get2() {
 9         return "priority 0";
10     }
11 }
12
13 SmolHTTPd server = new SmolHTTPd(8080);
14 server.register("/", new Handler());
```

8 Ein vollständiges Beispiel

Im Folgenden sollen Angestellte eines Unternehmens mithilfe eines REST-kompatiblen Servers verarbeitet werden. Um den eingefügten Quellcode kompakt zu halten wird dabei auf die Umsetzung von HATEOAS verzichtet, sodass nur Stufe 2 des Richardson Maturity Model erreicht wird. Für die Verwaltung müssen Ressourcen bereitstehen, die das Auslesen aller Angestellten erlauben, ein Anlegen und Verändern eines Datensatzes gestatten und auch das Entfernen von Mitarbeitern unterstützen. Dazu wird zunächst ein Methode angelegt, die zum Starten verwendet wird und verschiedene Handler registriert. Alle Angestellten werden in diesem Beispiel in einer flüchtigen Liste abgelegt.

```
 1 public class Launcher {
 2    public static void main(String[] args) throws IOException {
 3       try (SmolHTTPd http = new SmolHTTPd(8080, true)) {
 4          final List<String> employeeList = new ArrayList<>();
 5
 6          http.register("/employees", new GetEmployees(employeeList));
 7          http.register("/employees", new AddToEmployees(employeeList));
 8          http.register("/employees/{id}", new EditEmployee(employeeList));
 9          http.register("/employees/{id}", new RemoveEmployee(employeeList));
10
11          Scanner sc = new Scanner(System.in);
12          sc.nextLine();
13       }
14    }
15 }
```

Die Handler-Klasse „GetEmployees“ umfasst ausschließlich eine Methode, die auf GET Anfragen reagiert und die Liste aller Angestellten zurückgibt.

```
 1 public class GetEmployees {
 2    private final List<String> employees;
 3
 4    GetEmployee(List<String> employees) {
 5       this.employees = employees;
 6    }
 7
 8    @Get
 9    public Object get() {
10       return this.employees;
11    }
12 }
```

Die Handler-Klasse „AddToEmployees“ enthält ebenfalls nur eine Methode, die auf POST Anfragen reagiert und den Namen des Mitarbeiters im Body der Anfrage erwartet, um anschließend einen neuen Nutzer zur Liste hinzuzufügen. Anschließend wird mit dem korrekten Statuscode 201 *Created* geantwortet.

```
 1 public class AddToEmployees {
 2    private final List<String> employees;
 3
 4    AddToEmployees(List<String> employees) {
 5       this.employees = employees;
 6    }
 7
 8    @Post
 9    public Object post(@BodyParam("name") String name) {
```

```
10        Employee newEmployee = new Employee(name);
11        this.employees.add(newEmployee);
12        return new Created(newEmployee, "/employees/" + newEmployee.id);
13    }
14 }
```

Die Handler-Klasse „EditEmployee“ enthält erneut nur eine Methode, die auf PATCH Anfragen reagiert und sowohl die Identifikationsnummer eines Angestellten sowie seinen neuen Namen im Body der Anfrage erwartet. Anschließend wird der entsprechende Nutzer, sofern er vorhanden ist, entsprechend angepasst.

```
 1 public class EditEmployee {
 2    private final List<String> employees;
 3
 4    EditEmployee(List<String> employees) {
 5       this.employees = employees;
 6    }
 7
 8    @Patch
 9    public Object patch(@Param("id") int id, @BodyParam("name") String name) {
10       if (id < this.employees.size()) {
11          this.employees.get(id).name = name;
12          return new NoContent();
13       }
14       else
15          return new NotFound();
16    }
17 }
```

Die letzte Handler-Klasse „RemoveEmployee“ erwartet die Identifikationsnummer eines Nutzers und löscht ihn, sofern er vorhanden ist, aus der Liste.

```
 1 public class RemoveEmployee {
 2    private final List<String> employees;
 3
 4    RemoveEmployee(List<String> employees) {
 5       this.employees = employees;
 6    }
 7
 8    @Delete
 9    public Object delete(@Param("id") int id) {
10       if (id < this.employees.size()) {
11          this.employees.remove(id);
12          return new NoContent();
13       }
14       else
15          return new NotFound();
16    }
17 }
```

Schlussendlich ist der Server nun einsatzbereit und lässt sich über die „Launcher“-Klasse starten. Über die nachfolgenden Aufrufe von *curl* kann die Funktion überprüft werden:

```
1 curl -X GET    "http://localhost:8080/employees"          # []
2 curl -X POST   "http://localhost:8080/employees"   -d "name=user1"
3 curl -X POST   "http://localhost:8080/employees"   -d "name=user2"
4 curl -X GET    "http://localhost:8080/employees"          # ["user1", "user2"]
5 curl -X PATCH  "http://localhost:8080/employees/1" -d "name=toast"
6 curl -X GET    "http://localhost:8080/employees"          # ["user1", "toast"]
7 curl -X DELETE "http://localhost:8080/employees/0"
```

```
8 curl -X GET    "http://localhost:8080/employees"          # ["toast"]
```

Quellenverweise

[1] (10. Jan. 2019). GitHub - FasterXML/jackson: Main Portal page for the Jackson project, Adresse: https://github.com/FasterXML/jackson.

[2] (10. Jan. 2019). Maven Repository: com.fasterxml.jackson.core » jackson-core, Adresse: https://mvnrepository.com/artifact/com.fasterxml.jackson.core/jackson-core.

[3] (10. Jan. 2019). Maven Repository: com.fasterxml.jackson.core » jackson-annotations, Adresse: https://mvnrepository.com/artifact/com.fasterxml.jackson.core/jackson-annotations.

[4] (10. Jan. 2019). Maven Repository: com.fasterxml.jackson.core » jackson-databind, Adresse: https://mvnrepository.com/artifact/com.fasterxml.jackson.core/jackson-databind.

[5] (25. Okt. 2018). RFC 2617 - HTTP Authentication: Basic and Digest Access Authentication, Adresse: https://tools.ietf.org/html/rfc2617.

[6] (27. Dez. 2018). Hypertext Transfer Protocol (HTTP) Status Code Registry, Adresse: https://www.iana.org/assignments/http-status-codes/http-status-codes.xhtml.

[7] (12. Jan. 2019). Java reflection: Is the order of class fields and methods standardized? - Stack Overflow, Adresse: https://stackoverflow.com/a/1097826.