API Beschreibung: FileService

Ausbildungsbereich: Technik

Studiengang: Informatik

Vorlesung: Web-Engineering I

Dozent: Martin Spörl

Allgemeine Hinweise:

Der hier gezeigte Webservice ist **nicht für den produktiven Einsatz** entwickelt wurden! Durch Fehlen diverser Sicherheitsmechanismen, Continuity-Mechanismen und Fehlererkennungen kann ein Einsatz im produktiven Umfeld hohe Sicherheits- und Businessrisiken mit sich führen! Der Webservice dient lediglich zu lehrenden Zwecken!

Die gezeigten Beispiele gehen davon aus, dass der Service unter **„http://localhost:8080“** erreichbar ist.

Contents

1. Service Beschreibung 2
1 Installation 2
1.1 Docker 2
1.2 Native 3
2 Authentifizierung 5
3 API-Endpunkte 7
3.1 GET - Ordner 7
3.2 GET - Dateien 9
3.3 POST - Ordner 10
3.4 POST – Dateien 11
3.5 DELETE - Ordner 13
3.6 DELETE – Dateien 14

1. Service Beschreibung

Der FileService ist eine Webservice, welcher eine Ordnerstruktur per REST-API präsentiert. Dabei unterstützt dieser Webservice das Schreiben bzw. Erstellen von Ordnern und Dateien sowie das Löschen derer.

Der Webservice ist für die Ausführung mit dem eingebauten PHP-Webserver gedacht[1].

1 Installation

Der Code des Webservices ist für die Ausführung in Docker vorbereitet. Als Basis sollte das offizielle PHP-CLI Docker-Image dienen[2] . Während der Erstellung der Dokumentation ist php:7.4-cli das empfohlene Image.

Alternativ kann der Code auch nativ mit PHP ausgeführt werden, falls die Installation von Docker nicht möglich ist oder zu Problemen führt. Die Ausführungsschritte werden im Folgendem genauer erklärt.

Vor dem Starten des Webservices, laden Sie das Code-Package herunter und entpacken Sie es an einen beliebigen Ort. Als Beispiel soll für Windows „C:\fileservice“ und unter Linux „/fileservice“ dienen.

1.1 Docker

Der Container kann mit folgendem Befehl gestartet werden:

Linux:

```
docker run --rm -v /fileservice:/app -v /fileservice/upload.ini:/usr/local/etc/php/conf.d/upload.ini -w /app -p 8080:80 -d php:7.4-cli php -S 0.0.0.0:80 router.php
```

Windows:

```
docker run --rm -v C:/fileservice:/app -v C:/fileservice/upload.ini:/usr/local/etc/php/conf.d/upload.ini -w /app -p 8080:80 -d php:7.4-cli php -S 0.0.0.0:80 router.php
```

Anschließend wird der Webservice unter „http://localhost:8080“ verfügbar sein.

[1] https://www.php.net/manual/de/features.commandline.webserver.php
[2] https://hub.docker.com/_/php

1.2 Native

Um den Webservice auf dem Host-System direkt auszuführen muss unter Linux PHP aus den Paketquellen oder direkt aus den PHP-Quellen gebaut und installiert werden. Für Windows kann „PHP for Windows“ installiert werden.

Um die Funktionsweise der Datenbank sicherzustellen, muss der Webservice Schreibrechte auf dem Ordner haben. (Tipp: Es sollte beachtet werden, unter welchem Benutzer der PHP-Prozess aktiv ist)

Anschließend kann der Webservice mit folgenden befehlen gestartet werden:

Linux:

```
cd /fileservice
php -S localhost:8080 router.php
```

Windows[3]:

```
cd C:/fileservice
php -S localhost:8080 router.php
```

1.3 Upload-Limit

Innerhalb von PHP wird die Größe von Dateiuploads sowie die Größe eines POST-Bodys reguliert. Die mitgelieferte upload.ini setzt die entsprechenden Eigenschaften auf ein – für diesen Webservice – empfohlenes Maß. Somit sind Uploads mit bis zu 64 MB möglich. In einer nativen Installation muss diese Anpassung in der Konfigurationsdatei „php.ini“[4] vorgenommen werden

```
file_uploads = On
memory_limit = 64M
upload_max_filesize = 64M
post_max_size = 64M
```

1.4 Module

Der Webservice greift auf diverse Module zurück, die innerhalb von PHP aktiviert werden müssen. Im Falle einer Docker-Installation sind diese Module bereits aktiv. In einer nativen Installation müssen diese in der Konfigurationsdatei „php.ini“[5] zunächst aktiviert werden:

- pdo_sqlite
- fileinfo

[3] Nicht immer ist der Pfad zur php.exe in der PATH-Variable hinterlegt. Daher kann alternativ auch der vollständige Pfad angegeben werden (z.B. <XAMPP-Ordner>/php/php.exe)
[4] Für XAMPP liegt php.ini meist in „<XAMPP-Ordner>/php
[5] Für XAMPP liegt php.ini meist in „<XAMPP-Ordner>/php

2 Authentifizierung

Jede Anfrage an den Webservice muss authentifiziert werden. Diese Authentifizierung wird anhand eines API-Token durchgeführt, der zu nächst angefordert werden muss. Sobald ein Token erhalten wurde, ist dieser 600 Sekunden gültig. Wird dieser Token innerhalb der 600 Sekunden verwendet, wird die Gültigkeit des Tokens wieder auf 600 Sekunden gesetzt. Andernfalls verfällt der Token und eine neue Authentifizierung muss stattfinden.

Token anfordern:

Ein API-Token kann mittels eines HTTP-Post-Requests an „http://localhost:8080/login“ erfragt werden. Folgende Parameter müssen dabei gesendet werden:

Parameter	Beschreibung	Anmerkung
Username	Benutzername des Users	Standard Username ist „admin“
Password	Password des Users	Standard Password ist „admin“

Eine Beispiel anfrage sehe so aus:

```
curl -X POST -d username=admin -d password=admin http://localhost:8080/login
```

Im Fehlerfall wird folgendes JSON mit HTTP-Status 401 zurückgegeben:

```
{
	"error": "authentication failed"
}
```

Hinweis: Neben falscher Nutzerkennung, kann ein Grund für fehlgeschlagene Authentifizierung auch ein Fehler mit der Datenbank sein (z.B. keine Schreibrechte).

Im Erfolgsfall wird folgendes JSON mit HTTP-Status 200 zurückgegeben:

```
{
	"token": "<token>"
}
```

Requests authentifizieren:

Keine Anfrage darf unautorisiert erfolgen. Daher muss jede Anfrage mit dem vorab ermittelten Token authentifiziert werden.

Dazu wird jeder Anfrage mit dem HTTP-Header „Authorization“ versendet:

Authorization: Basic <AUTH CODE>

Der „AUTH CODE" ist da bei der Base64[6] Wert aus Benutzername:Token. Im Sinne der HTTP Basic Autentifizierung stellt der Token somit das Passwort für den Benutzer dar[7].

Eine Beispielhafte Anfrage kann wie folgt aussehen:

```
curl -X GET -H "Authorization:Basic <CODE>" http://localhost:8080/
```

Falls eine Authentifizierung nicht möglich war, weil z.B. der Token ungültig oder abgelaufen ist, wird der Request mit HTTP-Status 401 und folgendem JSON beantwortet:

```
{
        "error": " authorization failed"
}
```

Gültigen Token ungültig machen („Logout")

Wenn ein Token innerhalb der 600 Sekunden nicht mehr benötigt wird, sollte diese für ungültig markiert werden. Dazu muss eine HTTP-GET Anfrage an „http://localhost:8080/logout" gesendet werden

```
curl -X GET -H "Authorization:Basic <CODE> " http://localhost:8080/logout
```

Im Erfolgsfall meldet die API

```
{
        "message": "logout successful"
}
```

Andernfalls wird eine Fehlermeldung ausgegeben:

```
{
        "error": " logout failed"
}
```

[6] https://de.wikipedia.org/wiki/Base64
[7] https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/Authorization

3 API-Endpunkte

Die Endpunkte der API entsprechen den Pfaden innerhalb des „data“ Ordners des Webservices. So wird mit der Anfrage auf „/Folder1/File1.txt“ auf die Datei „data/Folder1/File1.txt“ verwiesen. Entsprechend würden die Anfrage auf „/Folder1“ auf den Ordner „data/Folder1“ verweisen.

Für Dateien und Ordner werden die 3 Methoden „GET“, „POST“ und „DELETE“ unterstützt.

3.1 GET - Ordner

Beispiel:

```
curl -X GET -H "Authorization:Basic <CODE>" http://localhost:8080/Folder1
```

Parameter:

Keine

Funktion:

Sendet eine Liste der Elemente in dem Angefragten Ordner. Dabei wird für jedes Element der Name und der Type zurückgegeben. Sollte das Element ein Ordner sein, so ist der Type „dir“, andernfalls der Mime-Type der Datei[8]

Erfolgsmeldung:

HTTP 200

```
[
	{
		"Name":"Folder1",
		"Type":"dir"
		},
	{
		"Name":"File1.txt",
		"Type":"text/plain"
	}
]
```

[8] https://wiki.selfhtml.org/wiki/MIME-Type/%C3%9Cbersicht

Fehlermeldung:

HTTP 500

```
{
        "error": "failed to load directory entries"
}
```

Hinweis: sollte der angefragte Ordner nicht existieren, geht die API zunächst davon aus, dass es sich um eine Datei handelt. Sollte diese ebenfalls nicht existieren wird die Fehlermeldung „file does not exist“ ausgegeben!

3.2 GET - Dateien

Beispiel:

```
curl -X GET -H "Authorization:Basic <CODE>" http://localhost:8080/File1?format=base64
```

Parameter:

Parameter	Beschreibung	Anmerkung
format	Optional kann mit format=base64 statt dem original Inhalt, der base64 kodierte Inhalt erhalten werden	Dieser Parameter ist optional. Standardmäßig wird der Inhalt der Datei ohne Anpassung gesendet

Funktion:

Liefert den Inhalt einer Datei zurück.

Erfolgsmeldung:

HTTP 200 – mit Inhalt der Datei

Fehlermeldung:

HTTP 500

```
{
        "error": "file does not exists"
}
```

3.3 POST - Ordner

Beispiel:

```
curl -X POST -H "Authorization:Basic <CODE>" -d "type=dir" http://localhost:8080/Folder1
```

Parameter:

Parameter	Beschreibung	Anmerkung
type	Um einen Ordner zu erstellen muss „type“ den Wert „dir“ haben.	

Funktion:

Erstellt einen neuen Ordner mit angegebenem Pfad.

Erfolgsmeldung:

HTTP 200

```
{
	"message": "directory created successfully"
}
```

Fehlermeldung:

HTTP 500

```
{
	"error": "failed to create directory"
}
```

3.4 POST – Dateien

Anmerkung: Mittels POST können Dateien angelegt und hochgeladen werden. Dabei unterstütz die API 2 Arten:

1. Upload via POST-Parameter im ContentType “application/x-www-form-urlencoded”
2. Upload via Multipart-POST

Sollte die hochgeladene Datei leer oder der übergeben Inhalt leer sein, so wird ein Fehler geworfen.

Beispiel:

```
curl -X POST -H "Authorization:Basic <CODE>" -d "content=<base64>" http://localhost:8080/File1
curl -X POST -H "Authorization:Basic <CODE>" -F "newFile=@local/File" http://localhost:8080/File1
```

Parameter:

Parameter	Beschreibung	Anmerkung
content	Base64 Kodierter String, der den Inhalt der Datei repräsentiert	Wird nur bei senden des Inhaltes via Content-Type „application/x-www-form-urlencoded“ benötigt. Ist „content“ leer oder nicht vorhanden, hat die Datei nach erstellen / bearbeiten keinen Inhalt.
newFile	Multipart-Content mit Inhalt der Datei	Wird nur bei upload mittels Multipart benötigt.

Funktion:

Erstellt eine neue Datei oder überschreibt eine bestehende mit entsprechendem Inhalt.

Erfolgsmeldung:

HTTP 200

```
{
        "message": "file written successfully"
}
```

Fehlermeldung:

HTTP 500

```
{
        "error": "failed to write file"
}
```

3.5 DELETE - Ordner

Beispiel:

```
curl -X DELETE -H "Authorization:Basic <CODE>" http://localhost:8080/Folder1
```

Parameter:

Keine

Funktion:

Löscht einen neuen Ordner mit angegebenem Pfad.

Erfolgsmeldung:

HTTP 200

```
{
        "message": "directory deleted successfully"
}
```

Fehlermeldung:

HTTP 500

```
{
        "error": "failed to delete directory"
}
```

3.6 DELETE – Dateien

Beispiel:

```
curl -X DELETE -H "Authorization:Basic <CODE>" http://localhost:8080/File1
```

Parameter:

Keine

Funktion:

Löscht eine Datei unter dem angegebenen Pfad.

Erfolgsmeldung:

HTTP 200

```
{
        "message": "file deleted successfully"
}
```

Fehlermeldung:

HTTP 500

```
{
        "error": "failed to delete file"
}
```