WBE-Praktikum 7

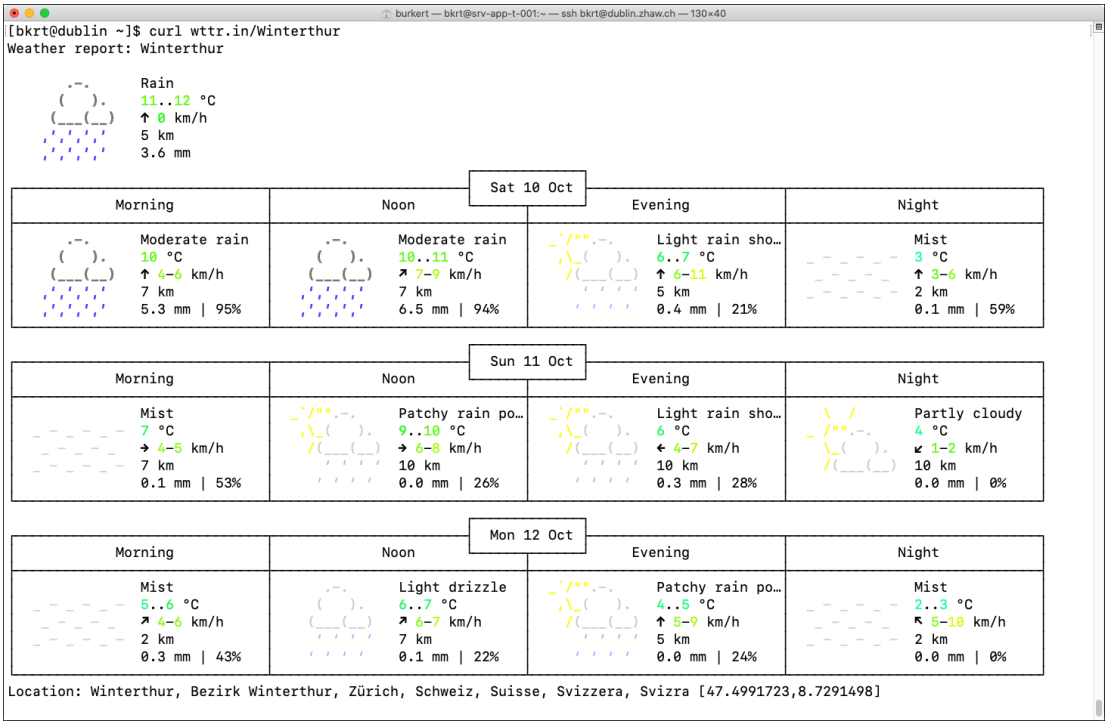
REST APIs

Aufgabe 1: Zugriff auf REST APIs

Zahlreiche Services im Internet stellen REST APIs zur Verfügung, etwa zur Abfrage von Fahrplänen, Wechsel- oder Aktienkursen, Musiktiteln oder Songtexten, Lexikoneinträgen oder Wetterdaten. Auf die meisten dieser Services sind natürlich nur GET-Requests zur Abfrage von Informationen möglich. Zudem muss man sich für viele Services registrieren: man erhält dann normalerweise einen API-Key, den man an Anfragen anhängt. Auf diese Weise können Anfragen registrierten Nutzern zugeordnet werden. Einige Services sind jedoch auch ohne API-Key zugänglich. Vielfach ist der Gratiszugang auf bestimmte Weise limitiert, zum Beispiel derart, dass nur eine bestimmte Zahl Zugriffe pro Zeiteinheit möglich sind.

Probieren Sie auf der Kommandozeile einmal folgenden Aufruf (falls *curl* nicht installiert ist, stellen Sie zuvor eine SSH-Verbindung zum *dublin.zhaw.ch* her):

```
$ curl wttr.in/Winterthur
```



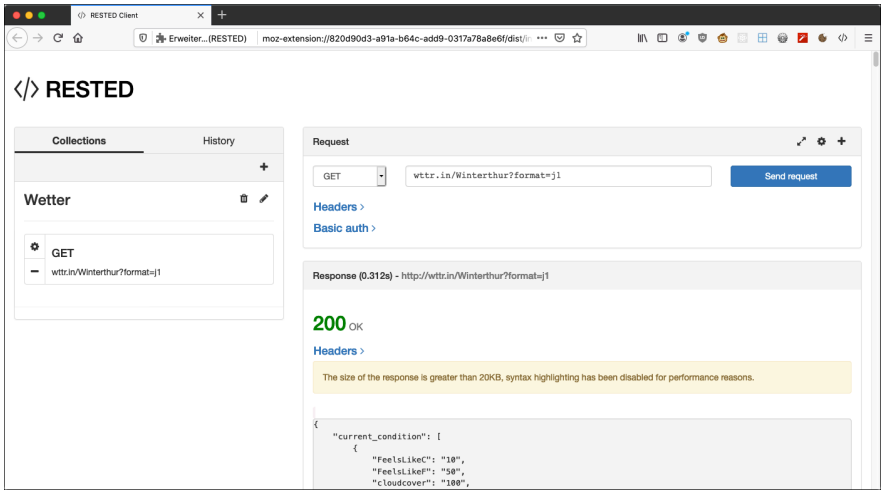
Das ist zwar eine schöne Ansicht (vorausgesetzt das Terminal-Fenster ist gross genug), für die Weiterverarbeitung in einem Programm ist die Ausgabe aber noch nicht ideal. Wir können aber auch eine JSON-Ausgabe erhalten:

```
$ curl "wttr.in/Winterthur?format=j1"
{
    "current_condition": [
        {
            "FeelsLikeC": "11",
            "FeelsLikeF": "53",
            "cloudcover": "75",
    ...
}
```

Weitere Möglichkeiten sind hier beschrieben:
https://github.com/chubin/wttr.in

Solche GET-Requests sind auch mit dem Browser möglich. Für die Arbeit mit REST-APIs ist es aber sinnvoll, geeignete Werkzeuge einzusetzen, welche auch andere Typen von HTTP-Requests zulassen und es am besten auch erlauben, Anfragen zu speichern und wieder abzurufen, sowie die Ergebnisse in geeigneter Form auszugeben oder zu speichern. Hier eine kleine Auswahl von Werkzeugen:

- Auf der Kommandozeile ist *curl* das Tool der Wahl, um Serverzugriffe mit verschiedenen Protokollen zu machen. Für Zugriffe auf externe Services können Sie wie oben auf *dublin.zhaw.ch* zugreifen. Zum Test eigener Services ist eine lokale Installation sinnvoll.
  https://curl.haxx.se/
- Ein geeignetes Programm mit grafischer Oberfläche ist *Insomnia*. Es basiert auf Electron (baut also auf Browser-Technologie auf) und ist somit plattformübergreifend verfügbar. Es ist ein kommerzielles Tool, die Gratisvariante bietet aber alle für unsere Zwecke nötigen Funktionen.
  https://insomnia.rest/
- Die Browser-Erweiterung RESTED bietet ebenfalls die wichtigsten Funktionen eines REST-Clients.
  https://addons.mozilla.org/en-US/firefox/addon/rested/



Es gibt natürlich noch zahlreiche weitere REST-Clients. Bei der Arbeit mit REST-APIs ist es sicher sinnvoll, neben *curl* noch wenigstens einen REST-Client mit grafischer Oberfläche installiert zu haben.

Aufgabe 2: REST-Zugriff aus einem Script (Abgabe)

Schreiben Sie ein Node.js-Script *currentTemp.js*, welches *wttr.in* verwendet, um die aktuelle Temperatur an einem bestimmten Ort auszugeben:[1]

```
$ node currentTemp.js Winterthur
11°

$ node currentTemp.js Madrid
15°
```

Verwenden Sie *https.get* des *https*-Moduls von Node.js. Die Daten des eingehenden Streams können Sie einsammeln, indem Sie *data*-Events bearbeiten. Das *end*-Event zeigt den Abschluss des Streams an, so dass die Daten mit *JSON.parse* ausgewertet werden können.

Abgabe

Geben Sie die Datei *currentTemp.js* ab.

Abgabeserver:	https://radar.zhaw.ch/python/UploadAndCheck.html
Name Praktikum:	WBE8
Dateiname:	currentTemp.js
Funktionsname:	beliebig (kein Jasmine-Test)
Export im Script:	nicht nötig

Aufgabe 3: Eigener REST-Service

Mit Hilfe von *Express* soll nun ein einfacher REST-Service aufgebaut werden. Im Unterricht wurde bereits eine kurze Einführung in *Express* gegeben. Weitere Informationen finden Sie auf der Website: https://expressjs.com

Ziel dieser Aufgabe ist es, einen Service zu entwickeln, welcher es erlaubt, JavaScript-Datenstrukturen zu speichern und mit Hilfe einer serverseitig generierten *id* wieder abfragen zu können. Die Datenstrukturen werden dazu im JSON-Format übertragen.

[1] Es hätte sich angeboten, das Script *temp.js* zu nennen, aber das klingt etwas zu temporär ☺.

- Erstellen Sie ein neues Projekt und installieren Sie Express:

```
$ mkdir expresso
$ cd expresso
$ npm init
$ npm install express –save
```

- Bei den Praktikumsunterlagen finden Sie ein Script *index.js*, in welchem bereits einige grundlegende Funktionen des Service umgesetzt sind. Analysieren Sie den Aufbau des Scripts. Am besten konsultieren Sie dabei die Dokumentation auf der Express-Website.

- Kopieren Sie das Script ins Projekt und starten Sie den Server. Hier noch einmal der Hinweis zu lokalen Servern: achten Sie darauf, dass Ihr Notebook auf dem verwendeten Port nicht von aussen erreichbar ist. Grundsätzlich muss damit gerechnet werden, dass der Service nicht gegen alle möglichen Angriffe abgesichert ist.

Die folgenden Schritte sollten Sie einerseits mit *curl* und andererseits zum Vergleich mit einem anderen REST-Client machen. In den Beispielen ist jeweils der *curl*-Aufruf angegeben.

- Machen Sie einen GET-Request zum Service auf die id=1234567890:
  `$ curl "http://localhost:3000/api/data/1234567890"`
  Es sollte ein Fehler mit dem Hinweis auf den fehlenden API-Key zurückgegeben werden.

- Zur Demonstration verlangen wir immer den gleichen API-Key:
  `$ curl "http://localhost:3000/api/data/1234567890?api-key=wbeweb"`

- Ein POST-Request enthält die zu speichernden Daten im JSON-Format. Diese werden unter einer zufällig generierten *id* gespeichert, welche der Server als Antwort liefert.

```
$ curl -d "{\"name\": \"Eva\"}" "http://localhost:3000/api/data?api-key=wbeweb"
-H "Content-Type: application/json"
```

- Verwenden Sie die generierte *id*, um die soeben gespeicherten Daten mit einem GET-Request wieder abzufragen.

Was unserem Service noch fehlt ist eine Möglichkeit, gespeicherte Daten zu überschreiben oder wieder zu löschen. Diese beiden Funktionen sollen nun noch implementiert werden.

- Ergänzen Sie den Service um die Methode DELETE. Diese bezieht sich auf eine einzelne Ressource, enthält also wie der GET-Request die *id*. Die Ressource wird gelöscht und der Statuscode 204 (*no data*) zurückgegeben. Wenn gar keine Ressource mit dieser *id* existiert, wird ebenfalls 204 zurückgegeben, denn mehrere gleiche Aufrufe sollen dasselbe Verhalten zeigen.

- Ergänzen Sie den Service um die Methode PUT. Diese bezieht sich auf eine einzelne Ressource, enthält also wie der GET-Request die *id*. Die Ressource wird mit den im Body als JSON gesendeten

Daten überschrieben. Diese werden ausserdem zurückgegeben. Wenn keine Ressource mit dieser *id* existiert erfolgt wie bei GET ein *404 (not found).*

- Testen Sie die neuen Methoden gründlich, am besten sowohl mit *curl* als auch mit einem grafischen Client. Bei *curl* kann die HTTP-Methode mit der Option -X übergeben werden:
  `curl "http://localhost:3000/api/data/...?api-key=wbeweb" -X DELETE`

Aufgabe 4: Fileserver (fakultativ)

Im Unterricht wurde als Beispiel Code eines Fileservers gezeigt, der es erlaubt, mit entsprechenden HTTP-Requests Dateien anzulegen, auszugeben, zu ändern und zu löschen. Das Beispiel stammt aus *Eloquent JavaScript, 3rd Edition*.

Der JavaScript-Code ist im Script *fileserver.js* enthalten. Benötigt wird noch das Modul *mime*, um beim GET-Request den korrekten Content-Type mitzuliefern. Sie können es mit *npm* installieren.

- Testen Sie die verschiedenen Funktionen des Fileservers. Es ist vermutlich eine gute Idee, dass Script von einem leeren Unterverzeichnis des Projekts aus zu starten. Warum?

- Arbeiten Sie sich durch den Code, um die Funktionsweise zu verstehen. Dieses Beispiel ist sicher dem Bereich «JavaScript für Fortgeschrittene» zuzuordnen. Machen Sie sich also keine Sorgen, wenn Sie nicht alles verstehen. Es ist hilfreich, den Sourcecode um Kommentare zu ergänzen. Erklärungen finden Sie auch im entsprechenden Kapitel von *Eloquent JavaScript*: https://eloquentjavascript.net/20_node.html#h_yAdw1Y7bgN