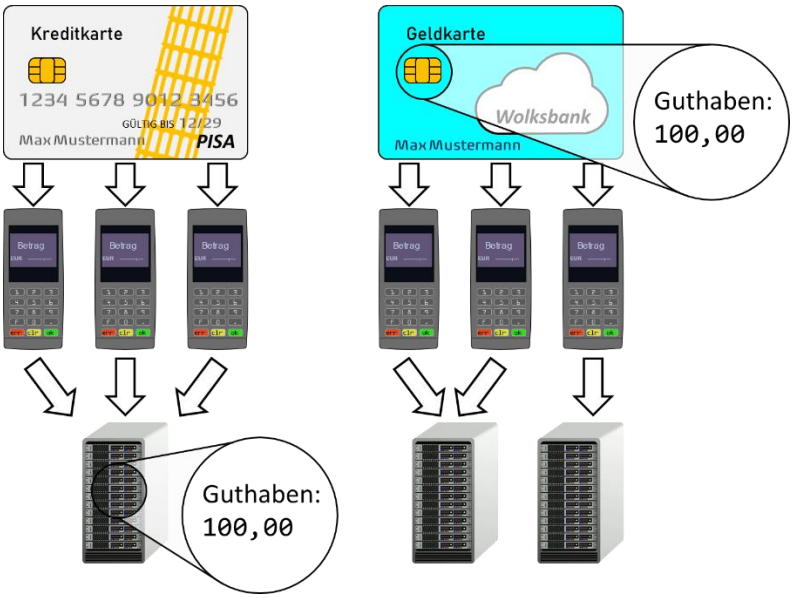
1 Stateless und stateful

In diesem Dokument behandeln wir den Ansatz für eine REST API. Im Begriff „REST API“ sind gleich zwei Akronyme enthalten. REST und API. Beginnen wir mit API, was für Application Programming Interface steht. Hiermit möchte man ausdrücken, dass es eine Applikation gibt, welche eine von außen per Programm ansteuerbare Schnittstelle aufweist. Eine API ist also eine klar definierte Möglichkeit, mit der zugehörigen Applikation zu kommunizieren.

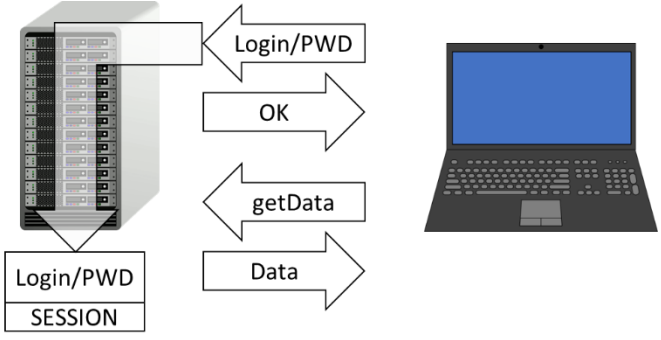
REST wiederum steht für „Representational State Transfer“ und wird oft auch als „stateless“ (zustandslos) beschrieben; es gibt somit stateless und stateful Kommunikationsformen. Um dies zu verdeutlichen, sehen wir uns folgende Grafik an:



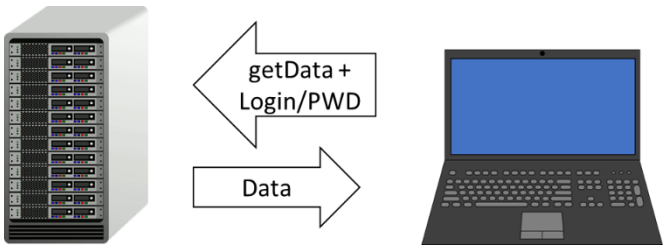
Bei der Kreditkarte wird der aktuelle Status – sprich der zur Verfügung stehende Betrag – auf dem Server gespeichert, was ihn stateful macht. Jedes Endgerät muss mit dem zentralen Server kommunizieren, um den Geldbetrag zu erhalten. Die Geldkarte hingegen wird aufgeladen – der zur Verfügung stehende Geldbetrag ist somit nicht auf irgendwelchen Servern. Die Kommunikation mit den Endgeräten erfolgt daher stateless. Dadurch ist es egal, zu welchem Server das einzelne Kartenlesegerät Kontakt aufnimmt.

Bei der Webtechnologie finden wir auch beide Formen von Kommunikation. Wenn wir bspw. auf dem Server die aktuelle Session mit dem Browser speichern und für die Kommunikation nutzen, so hat der Server den Status gespeichert, was ihn stateful macht. Bringt jedoch der Client mit jeder Kommunikation alle wichtigen Informationen mit, so muss dies nicht auf dem Server gespeichert werden – sie ist dann stateless.

In der Konsequenz heißt das, wir müssen uns bei der stateful Kommunikation nur auf die Informationen konzentrieren, welche für den aktuellen Request wichtig sind – bspw. „zeige mir alle meine Bestellungen“. Wir können in der Session für jeden Datenaustausch die UserID und die Tatsache, dass sich der User authentifiziert hat, speichern. Wir müssen es nicht immer neu schicken.



Bei der stateless Kommunikation müssen wir immer alles, sprich auch den Kontext mitschicken. Dies macht zwar die Kommunikation komplizierter, ermöglicht aber eine extrem flexible Lastverteilung auf verschiedene Server – schließlich muss kein Server auf eine gespeicherte Historie zugreifen. Dies macht eine stateless Kommunikation so interessant für die Industrie.



2 CRUD

Wenn man die Kommunikation mit einem Server als reinen Datenaustausch interpretiert, so kann man jede Kommunikation einer von vier Grundtypen zuordnen. Diese finden sich auch in anderen Bereichen, wie im http-Protokoll oder auch bei Datenbankinteraktionen wieder:

Grundtyp:	**Bedeutung:**	**http Methode:**	**DB-Befehl:**
Create	Erzeugen von neuen Daten auf dem Server	POST	INSERT
Read	Lesen von Daten auf dem Server	GET	SELECT
Update	Ändern von Daten auf dem Server	PATCH/PUT	UPDATE/REPLACE
Delete	Löschen von Daten auf dem Server	DELETE	DELETE

Bei Update finden wir im Regelfall zwei Möglichkeiten – einmal die selektive Änderung von einzelnen Eigenschaften eines Datensatzes (was bei Datenbanken das „UPDATE“ Kommando wäre) und das komplette Austauschen eines Datensatzes (was bei Datenbanken das „REPLACE“ Kommando ist).

In Express haben wir bereits die Verarbeitung von GET-Nachrichten gesehen. Diese wurden entweder direkt durch das `app` Objekt entgegengenommen:

```
app.get("/api/eins", (req, res, next) => {
    res.send("eins");
})
```

oder durch eine Middleware im `router` Objekt:

```
router.get("/eins", (req, res, next) => {
    res.write("I am in my middleware\n");
    next();
})
```

Die Methoden `put`, `post`, `patch` und `delete` werden entsprechend unterstützt. Wichtig ist jedoch Folgendes:

3 Vorgaben für eine REST Schnittstelle

Roy Fielding hat in einer Doktorarbeit die Grundvoraussetzungen für eine einfache Schnittstelle vorgeschlagen und damit den Grundstein für REST gelegt. Die Vorgaben waren primär eine Sammlung von „best Practices" – weshalb man REST auch nicht als Protokoll verstehen soll, sondern eine Art Pattern – man spricht auch von RESTful Design. Bei der Formulierung dieses „Patterns" wurden folgende Forderungen bzw. „Einschränkungen" (engl. Constraints) gestellt:

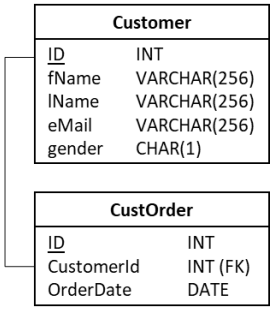
- **Uniform Interface, oder „einheitliche Schnittstelle"**
  Alle Clients nutzen die gleiche Schnittstelle, egal ob es sich um eine App, ein Desktopprogramm oder einen JavaScript Client handelt. Hierbei gilt:
  - Sämtliche Ressourcen sind eindeutig identifizierbar – also keine Doppeldeutigkeiten. Die Identifikation wird über URI durchgeführt.
  - Vom Client angefragte Ressourcen werden vom Server als eine Ressourcenrepräsentation geschickt – meist JSON, es sind aber auch andere Strukturen möglich. Dies kann so weit gehen, dass im http-Header über den Content-Type die erwartete Struktur eingefordert wird und der Server sich danach richtet (bspw. schickt bei GET/Content-Type: application/xml der Server eine XML-Repräsentation und bei GET/Content-Type: application/json eben eine JSON Repräsentation.
  - Die Nachrichten müssen „selbstbeschreibend" sein, so dass der Empfänger in der Lage ist, sie zu verstehen.
  - Die Schnittstelle muss dem HATEOAS Prinzip (Hypermedia as the Engine of Applications State) folgen. Die Idee ist, dass man lediglich eine Adresse benötigt und die Rückantwort Rückschlüsse auf weitere Interaktionsmöglichkeiten gibt.

- **Client-Server Architektur**
  Lose Kopplung der API (also der Sichtweise des Clients) und der dahinterliegenden Backend-Architektur
- **Zustandslose Kommunikation**
  Client liefert alle für die Kommunikation notwendigen Informationen
- **Cacheable**
  Die Rückmeldungen sollten Rückschlüsse darüber geben, ob sie gecached werden dürfen, um die empfangenen Informationen ggf. später wiederzuverwenden
- **Mehrschichtiges System**
  Hierunter versteht man die Aufteilung der einzelnen der serverseitigen Aufgaben in einzelne Komponenten.

Die Primäre Idee ist nun, dass eine Serverseitiger Datensatz (die Ressource) als Datenrepräsentation angesehen wird und einen aktuellen Status hat (bspw. der aktuelle Kontostand des Kontos). Dieser wird über REST vom Server zum Client transferiert, dort ggf. geändert und wieder zurücktransferiert. Daher auch der Name „Representational State Transfer".

4 Adressierung der Ressourcen

Gehen wir für unsere Überlegungen vom rechts stehenden Datenmodell aus. Wir sehen, dass jeder Kunde durch eine ID adressiert wird und ein Kunde mehrere Bestellungen haben kann. Hierfür können wir nun die URIs für die Adressierung der Ressourcen festlegen:



Eine Liste der Kunden:	http://localhost/api/customer
Ein Kunde mit der ID 1234:	http://localhost/api/customer/1234
Alle Bestellungen des Kunden 1234:	http://localhost/api/customer/1234/orders
Die Bestellung mit der ID 12345678	http://localhost/api/orders/12345678

Oft kennzeichnet man die URI für eine REST API mit einem „api" zu Beginn des Endpunktes. Danach folgt ein Substantiv in Mehrzahl formuliert – da man hier eine Liste erwartet. Wenn wir aus einer Liste eine spezielle Ressource anfragen, so folgt nach dem Substantiv die ID. Möchten wir hierzu wiederum die Liste der Bestellungen, so folgt wieder ein Substantiv in Plural – in unserem Fall die Bestellungen. Üblicherweise ist es jedoch nicht mehr sinnvoll, hier nochmal eine ID abzufragen, es sei denn, die IDs der Bestellungen haben nur für

diesen einen Kunden eindeutige IDs (was aber so eher nicht vorkommt). Für die einzelnen Bestellungen erzeugen wir somit wieder eine eigene Ressourcenanfrage. Würden wir nun nur http://localhost/api/orders anfragen, so würden wir alle Bestellungen (unabhängig des Kunden) erhalten.

Wenn wir nun das **HATEOAS** Prinzip verfolgen, so müssen wir nun pro Kundenrepräsentation noch die möglichen URIs zur weiteren Spezifikation mitsenden. Die JSON-Repräsentation eines Kunden könnte somit wie folgt aussehen:

```
{
  "ID":1234,
  "fname":"Peter",
  "lName":"Lustig",
  "eMail":"peter@lustig.de",
  "gender":"m",
  "link":[
    {"rel":"self","href":"api/customer/1234"},
    {"rel":"orders","href":"api/customer/1234/orders{?dateFrom,dateTo}"}
  ]
}
```

Mit "`self`" bestätigt man den Zugriff auf die eigene Ressource, was vor allem bei der Rückgabe von Listen sinnvoll ist. Mit „`orders`" indiziert man, wie man auf die Bestellungen des aktuellen Kunden zugreifen kann. Hier wurden noch die möglichen Querystring Parameter `dateFrom` und `dateTo` hinterlegt.

Ergänzen Sie nun das JSON um die Möglichkeit, dass zu dem Kunden auch Adressdaten angefragt werden können:

```
"link":[
    {"rel":"self","href":"api/customer/1234"},
    {"rel":"orders","href":"api/customer/1234/orders{?dateFrom,dateTo}"}
```

5 Empfangen von JSON

Die vom Client gesendeten Daten werden auch meist im JSON-Format übermittelt. Um diese nun abgreifen zu können, müssen wir einen JSON-Parser als Middleware einfügen, wodurch wir Zugriff auf die Daten erhalten. Gehen wir davon aus, dass folgender JSON-String per POST im http-Body an den Server gesendet wird:

```
{ "ID":1234,
  "fName":"Peter",
  "lName":"Lustig",
  "eMail":"peter@lustig.de",
  "gender":"m"}
```

Die Verarbeitung kann nun wie folgt durchgeführt werden:

```
…
const bodyParser = require("body-parser");
…
app.use(bodyParser.json());
app.post("/name", (req, res) => {
   console.log(req.body.fName + " " + req.body.lName);
   res.send({processStatus:"ok"});
})
```

Wir greifen also direkt auf das aus JSON erzeugte Objekt im Body zu und senden auch wieder ein Objekt als Replynachricht zurück – so dass Clientseitig das Ergebnis wieder in Objektform vorliegt.