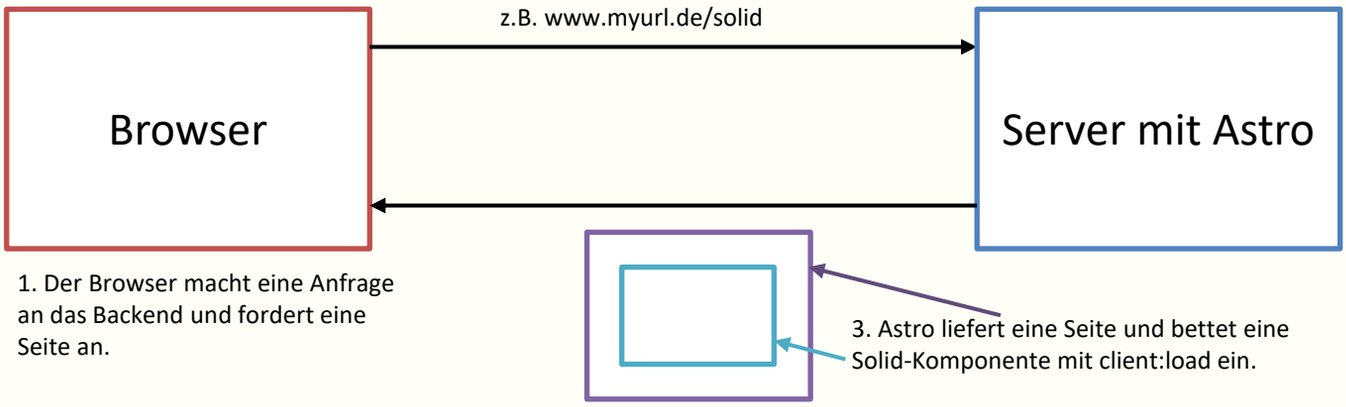
HOCHSCHULE
RAVENSBURG-WEINGARTEN
UNIVERSITY
OF APPLIED SCIENCES

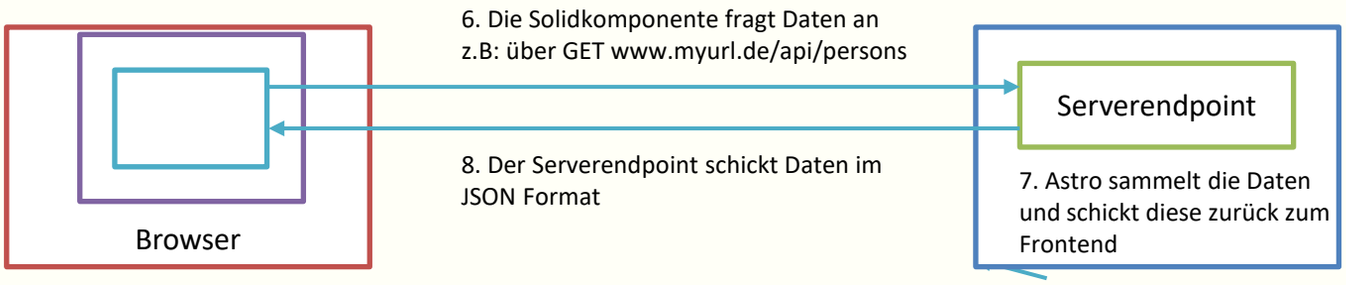
REST, Interaktion Frontend-Backends

Prof. Dr.-Ing. Thorsten Weiss

Version 1.4

Grundlegendes Konzept



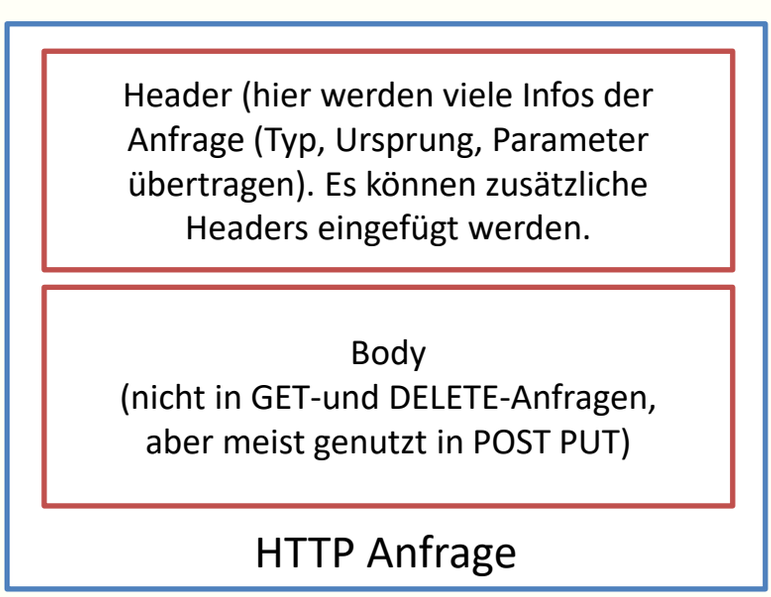


Es gibt 3 Möglichkeiten, Daten in die Anfrage zu packen.

1. In die URL (params)
   - www.myurl.de/api?a=1;b=2;c=3 ← Variante mit Variablennamen
   - www.myurl.de/api/1/2/3 ← Variante mit Routen

2. Headers

3. Body



Kommentiertes Beispiel:
Frontend: 2_solidPersonSimple
Backend: /api/personsimple.ts

REST / JSON

- REST (**Representational State Transfer)** ist ein Programmiervorgehen / **Regelwerk** zum Austausch von Daten zwischen einem Client und einem Server.
- Der Datenaustausch ist **zustandslos**. Jede Anfrage muss alle Informationen enthalten, die zum Abarbeiten vom Server nötig sind.
- REST arbeitet **asynchron** und wird dabei häufig mit AJAX (asynchroner Datenaustausch zwischen Browser und Server) in Verbindung gebracht. Eine Anwendung kann also parallel mehrere Anfragen an den Server schicken.

- Bei WebApps ist die Idee, funktionierende Anwendungen vollständig im Browser zu laden.
- **Zustandsänderungen, Eingaben etc. die auf dem Server gespeichert oder geladen** werden sollen, werden mit **kleinen Datenpaketen** ausgetauscht. Dadurch muss die Seite nicht ständig neu geladen werden, sondern nur kleine Datenpakete. Einzelne Informationen auf der Seite werden aktualisiert.
- Falls die WebApp keine Kommunikation mit dem Server benötigt, wird sie einmalig geladen und es erfolgt keine weitere Kommunikation. Beispiel: App, die °C in Fahrenheit umrechnet oder Meter in Inch → dies kann ohne Serverkommunikation umgesetzt werden.

Übermittlung der Daten via JSON (ggf. XML oder binär (z.B. für Bilder etc.)

Serverseitiges Skript/Service,
z.B. PHP, Java, C#,...

Unidirectional:
iPhone lädt Daten hoch.
iPhone fragt an und lädt Daten herunter (REST)

Bidirectional:
Upload/Download (Sockets)

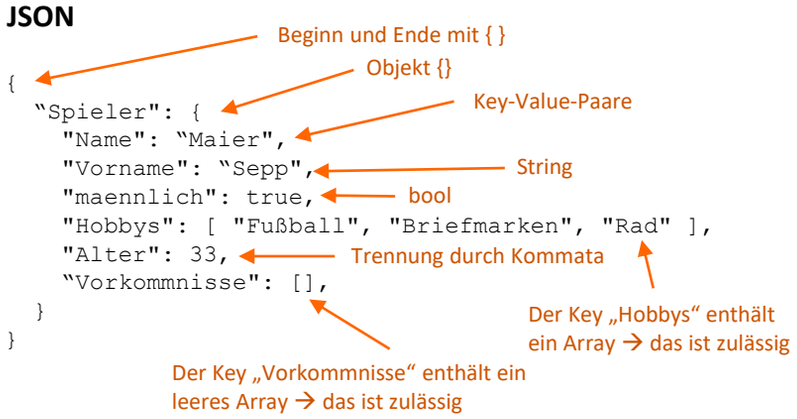
- Das übliche Austauschformat ist HTML in Verbindung mit **JSON**. Vor allem in der Microsoft-Welt wird oftmals mit XML gearbeitet.
- **REST unterscheidet 4 Anfragetypen:**

GET	POST	PUT	DELETE
Datenabrufen	Neuer Datensatz ablegen	Datensatz aktualisieren.	Datensatz löschen

Get kann sowohl ganze Seiten liefern als auch JSON Pakete

- Im Javascript (JS)-Code im Client werden Anfragen gestartet.
- Nach **asynchronem Empfang** werden die Ergebnisse im JS verarbeitet

JSON



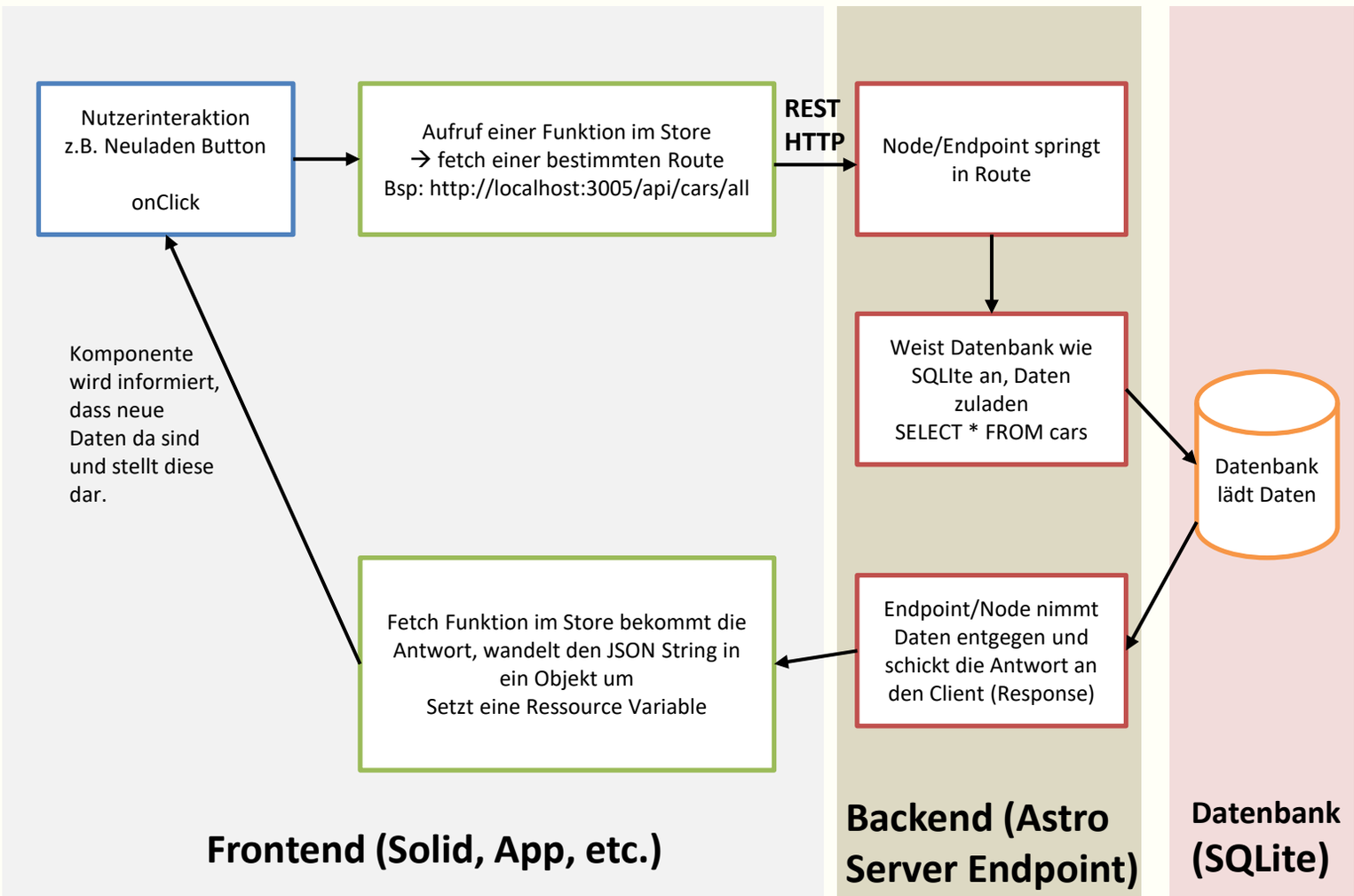
JSON - Folgende Datentypen sind zulässig:

- **Nullwert:** null
- **Boolscher Wert:** true/false
- **Zahl:** double / int
- **String:** Beginnt und endet mit ".
- **Array:** beginnt mit [und endet mit].
  - Elemente werden durch Kommata getrennt.
  - Arrays können Werte oder Objekte gleichen oder verschiedenen Typs im selben Array enthalten.
  - Auch leere Arrays sind erlaubt.
- **Objekt** beginnt mit { und endet mit }.
  - Enthält eine ungeordnete Liste von Eigenschaften, die durch Kommata getrennt werden.
  - Eigenschaften bestehen aus einem Schlüssel (ein String) und einem Wert (Key-Value)
  - Schlüssel müssen eindeutig sein → nur einmal enthalten.
  - Objekte ohne Eigenschaften („leere Objekte“) sind zulässig.

Cloud Computing mit React / Nodev

Eine **Datenaktion im Cloudcomputing** durchläuft folgende Phasen

- Nutzerinteraktion in der Komponente triggert einen http-Call (fetch oder axios).
- Der Server bearbeitet die Anfrage (get, post, put, delete) und führt eine Aktion auf der Datenbank aus (select, insert, update, delete)
- Danach werden die Daten an den Client zurückgesendet.
- Die Komponenten werden über die neu empfangenen Komponenten informiert und können neu rendern (bei Solid erledigt das createRessource).

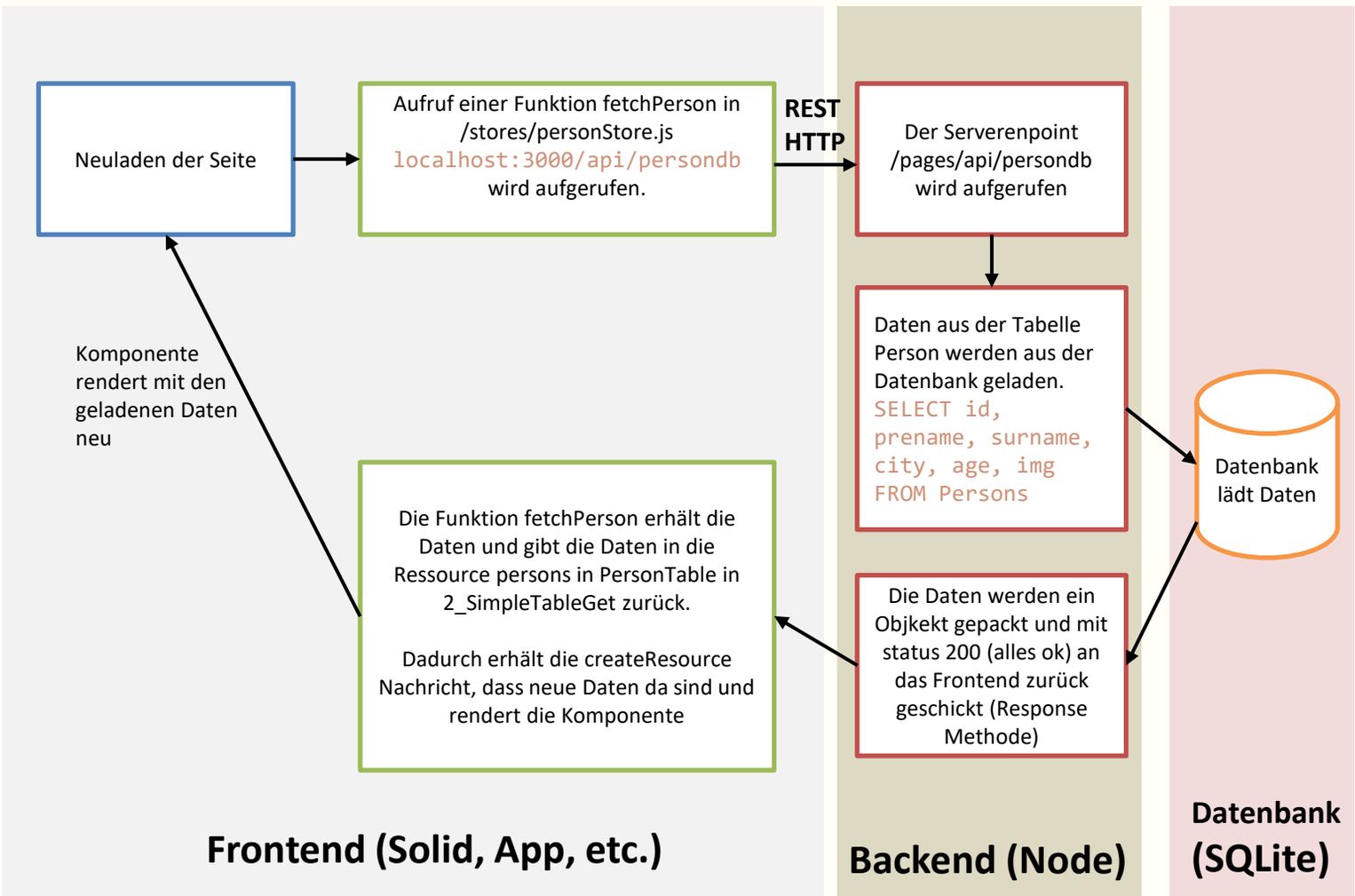


Siehe Inventory App

Zyklus am Beispiel des Web2 Projekts

Beispiel des Zykluses am Beispiel 2_SimpleTableGet.jsx

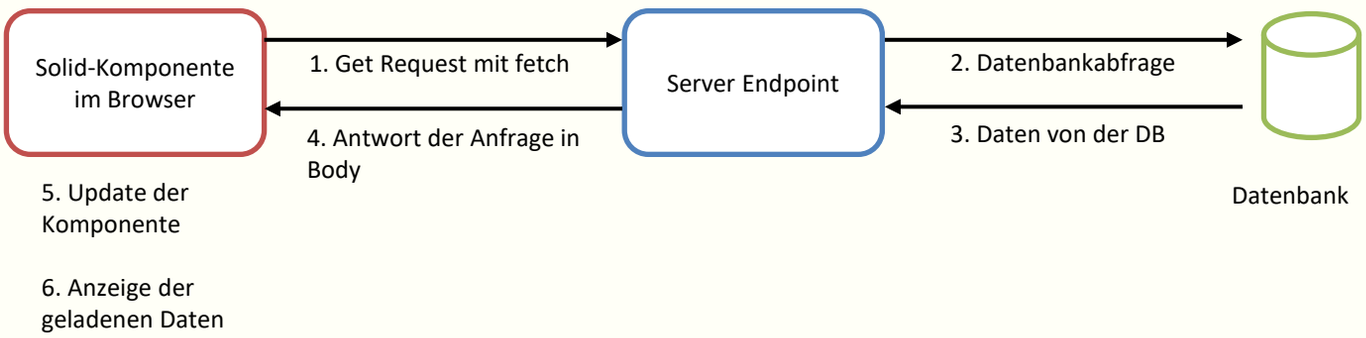
- Beim erstmaligen Rendern von **PersonTable** in 2_SimpleTableGet.jsx wird die Methode fetchPersons aufgerufen. Dies wird von createResource automatisch gemacht.



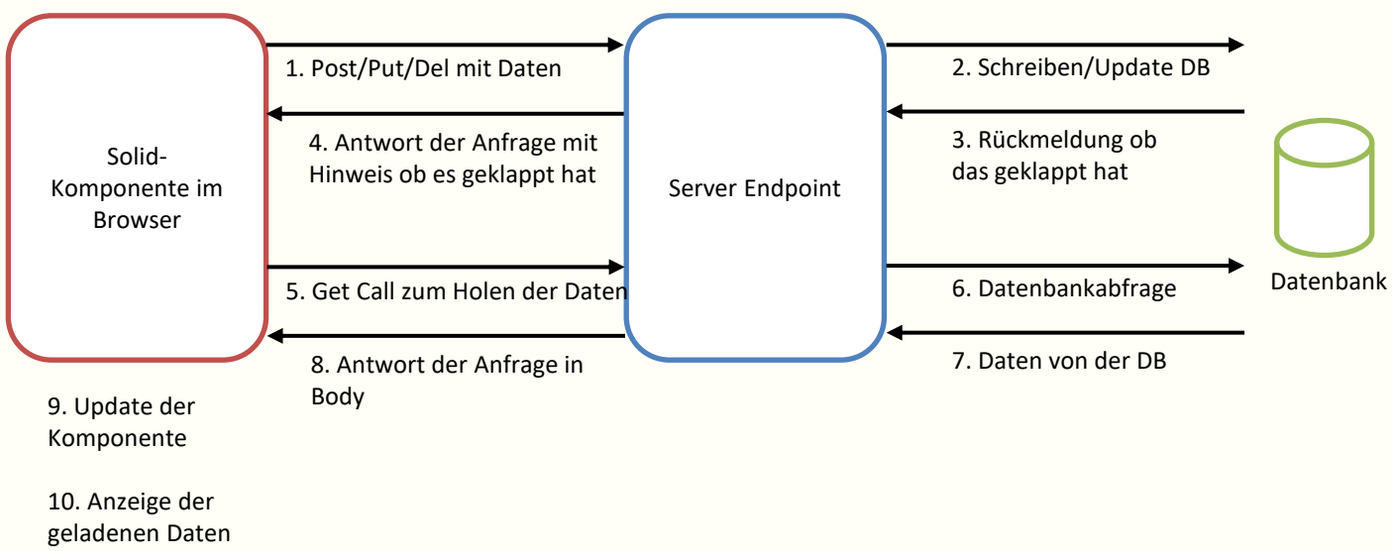
Siehe Inventory App

CRUD Calls

- **Ablauf eines Get-Calls:**



- **Ablauf eines Post/Put/Delete-Calls**



Astro Endpoints

- Ein beispielhafter, kommentierter RESTful-Service (also alle 4 Operationen Get, Put, Post, Delete bereitgestellt) ist in der Datei persondb.ts bereitgestellt.
- Bitte beachten: Damit das Projekt funktioniert, muss die Datei initdb.js im Verzeichnis ./src/database ausgeführt werden, um die Datenbank zu erzeugen.
- Öffnen sie das Terminal: mit cd [Verzeichnisname] können sie in ein Unterverzeichnis wechseln. Die Tab-Taste vervollständigt wenn möglich die Namen.

  cd src
  cd database
  npx node initdb.js

- **In den Datei initdb.js und persondb.ts finden sie den kommentierten Code. Bitte anschauen.**

Rückgabe und Fehlercodes

Damit der Status und Fehler bei den REST Calls übermittelt werden können, wurden Rückgabe und Fehlercodes definiert

Der Zahlenraum 2xx der Rückgabecodes zeigen fehlerfreie Übertragung und Bearbeitung an

200 : OK

Die Anfrage war erfolgreich und die Antwort enthält die angeforderten Daten.

201 : Created

Die Anfrage war erfolgreich und eine neue Ressource wurde erstellt (z. B. nach POST).

202 : Accepted (in dieser VL nicht behandelt)

Die Anfrage wurde akzeptiert, aber noch nicht vollständig verarbeitet (z. B. für asynchrone Operationen).

204 : No Content (in dieser VL nicht nötig)

Die Anfrage war erfolgreich, aber es gibt keine Rückgabedaten (z. B. bei erfolgreichem DELETE).

Der Zahlenraum 3xx zeigt Weiterleitungen an (in dieser VL nicht genutzt)

301 : Moved Permanently

Die Ressource wurde dauerhaft an eine neue URL verschoben. Der Client sollte zukünftig diese neue Adresse verwenden.

302 : Found (Moved Temporarily)

Die Ressource wurde vorübergehend an eine andere URL verschoben.

303 : See Other

Die Antwort zu dieser Anfrage ist unter einer anderen URL zu finden. Häufig nach POSTs verwendet.

304 : Not Modified

Die Ressource wurde seit der letzten Anfrage nicht verändert. Der Client kann die zwischengespeicherte Version verwenden.

Der Zahlenraum 4xx und 5xx zeigt Fehler an.

400 : Bad Request

Die Anfrage ist fehlerhaft oder unvollständig (z. B. fehlende Parameter, ungültiges Format).

401 : Unauthorized

Authentifizierung fehlt oder ist ungültig. Der Client muss sich anmelden.

403 : Forbidden

Zugriff verweigert, obwohl Authentifizierung erfolgt ist (z. B. unzureichende Berechtigungen).

404 : Not Found (berühmte 404-Seiten, wenn man eine falsche URL eingegeben hat)

Der angeforderte Endpunkt oder die Ressource wurde nicht gefunden.

405 : Method Not Allowed

Die verwendete HTTP-Methode (z. B. POST statt GET) ist für diesen Endpunkt nicht erlaubt.

409 : Conflict

Ein Konflikt mit dem aktuellen Status der Ressource (z. B. doppelter Eintrag).

422 : Unprocessable Entity

Die Anfrage war syntaktisch korrekt, konnte aber aufgrund semantischer Fehler nicht verarbeitet werden (z. B. Validierungsfehler).

500 : Internal Server Error

Ein unerwarteter Fehler ist auf Serverseite aufgetreten.

Datenmodell : 1:1, 1:n (1)

- Für den Entwurf eines Datenmodells sollte genug Zeit und Sorgfalt investiert werden.
- Es hat sich gezeigt, dass eine **einheitliche Namensgebung** von Spalten und Tabellen sehr viel Ärger, Zeit und Missverständnisse spart. Daher sollten die Namenskonventionen zu Beginn festgelegt werden, bevor die erste Zeile programmiert wird.

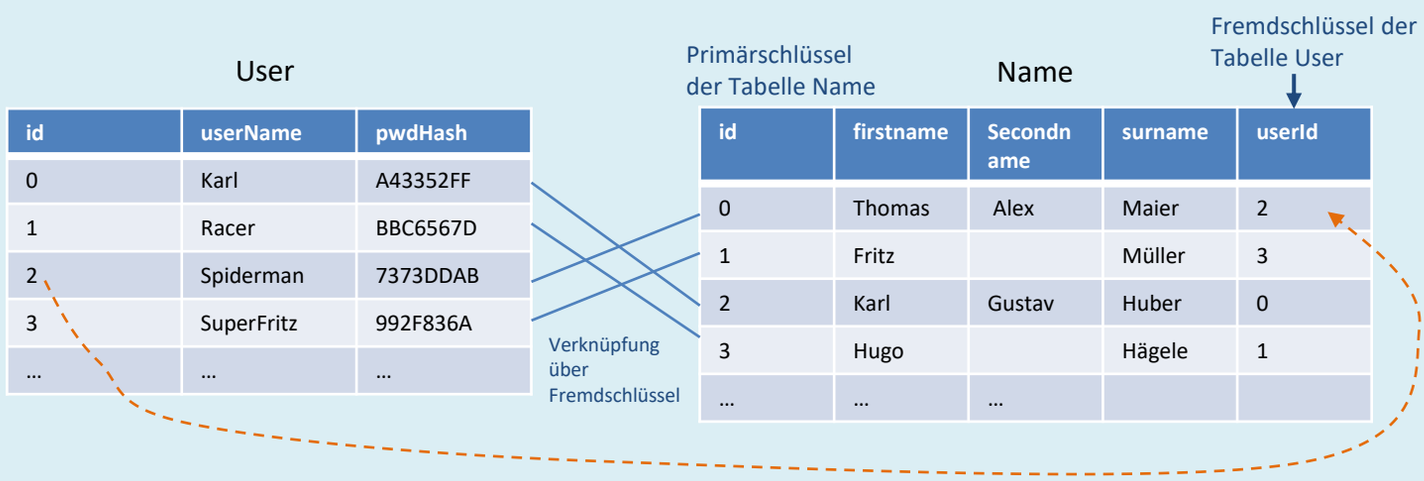
  Beispiele für Regeln:
  - Keine Underscores (MyTable statt my_table)
  - Alle Bezeichnungen auf Englisch
  - Alle Tabellennamen UpperCamelCase (MyTable)
  - Abkürzungen einheitlich (no = number of → noCalls, ms = measurement → msSensorTempC)
  - Alle Spaltennamen lowerCamelCase (userName, adress, noCalls)
  - Alle Spalten, die eine physikalische Größe beinhalten, die Einheit anhängen. (velWheelFrontMs: Geschwindigkeit vorne in m/s, sensorTempOuterC: Außentemperatur in Celsius).
  - Alle Primärschlüssel heißen „id“.
  - Alle Fremdschlüssel heißen tableNameId (partId, carId)
  - usw…

- Das Speichern und Abrufen von einfachen Tabellen ist auf den letzten Seiten gezeigt. Sequelize wandelt Tabellen in Javaskript Dateien direkt um.

Verknüpfungen von Tabellen

- Tabellen können verknüpft werden, um eine Beziehung (Relationship) darzustellen. Hierbei unterscheiden wir drei Verknüpfungsarten:

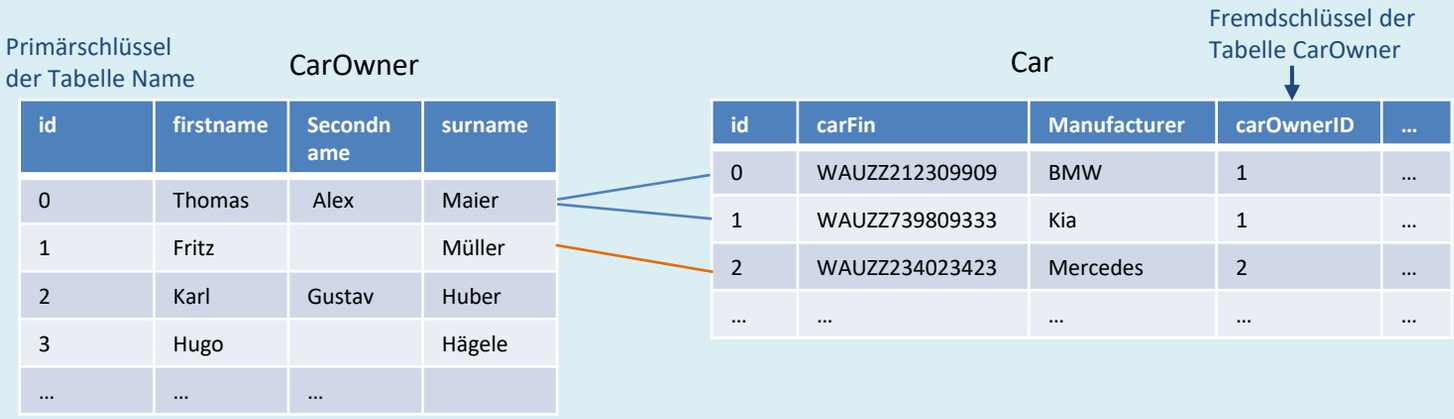
- **1:1 Verknüpfung:** Eine Zeile in einer Tabelle gehört zu einer Zeile in einer anderen Tabelle.
  Beispiel: Ein Tabelle User und eine Tabelle Name ist 1:1 verknüpft



id	userName	pwdHash
0	Karl	A43352FF
1	Racer	BBC6567D
2	Spiderman	7373DDAB
3	SuperFritz	992F836A
…	…	…

id	firstname	Secondname	surname	userId
0	Thomas	Alex	Maier	2
1	Fritz		Müller	3
2	Karl	Gustav	Huber	0
3	Hugo		Hägele	1
…	…	…		

- **1:n Verknüpfung:** Eine Zeile in einer Tabelle kann zu vielen Zeilen in einer anderen Tabelle gehören.
  Beispiel: Ein User kann Halter mehrerer Autos sein. Aber jedes Auto hat nur einen Halter:



id	firstname	Secondname	surname
0	Thomas	Alex	Maier
1	Fritz		Müller
2	Karl	Gustav	Huber
3	Hugo		Hägele
…	…	…	

id	carFin	Manufacturer	carOwnerID	…
0	WAUZZ212309909	BMW	1	…
1	WAUZZ739809333	Kia	1	…
2	WAUZZ234023423	Mercedes	2	…
…	…	…	…	…

Datenmodell : n:m, Bilder (2)

- **n:m Verknüpfung:** Eine Zeile in einer Tabelle gehört zu mehreren Zeilen in einer anderen Tabelle.
  Beispiel: Warenkorb in einem Shopsystem: Ein User kann mehrere Waren in seinem Einkaufswagen haben. Dieselbe Ware kann bei vielen Nutzern in einem Warenkorb liegen.
- In diesem Fall muss eine Zwischentabelle eingefügt werden, die diese Verknüpfung abbildet.

User

id	userName	pwdHash
0	Karl	A43352FF
1	Fritz	BBC6567D
2	Walter	7373DDAB
3	Francis	992F836A
...	...	...

UserArticle

id	idUser	idArticle
0	0	2
1	0	3
2	3	3
3	3	1
...	...	...

Primärschlüssel | Fremdschlüssel der Tabelle User | Fremdschlüssel der Tabelle Article

Article

id	name	price	orderNumber	...
0	shampoo	3.49	29308714	...
1	soap	1.99	12312312	...
2	showergel	2.99	33453252	...
3	aftershave	9.99	53345763	...
...	...	...		

Karl hat showergel und aftershave in seinem Warenkorb.
Francis hat aftershave und soap in seinem Warenkorb.
Fritz und Walter haben nichts in ihrem Warenkorb.

- Es können auch mehrere Verknüpfungen pro Tabelle vorhanden sein. So könnte die Tabelle User als 1:1 Verknüpfung mit einer Tabelle Payment 1:1 verknüpft sein, in der Zahlungsinfos enthalten sind. Weiterhin könnte der User 1:n mit Adress verknüpft sein, um mehrere Adressen pro Nutzer zu speichern.

- **Das Datenmodell mit Hilfe von Tabellen kann somit die gängigen Verknüpfungen abbilden. Datenmodelle werden daher so gestaltet, dass die Problemstellung mit diesen Tabellen abgebildet werden kann.**

Speichern von Massendaten wie Bildern

- Eine SQL Datenbank ist nicht dafür vorgesehen, Bilder, Dateien, sprich große Daten zu speichern.
- **Daher kann ein Link, ein Ort oder ein Hinweis auf einen Webspace angegeben wird.**
- Beispiel: Bilder einer Galerie sollen gespeichert werden:

Images

id	Name	Link	...
0	Rose	/img/rose.jpg	...
1	Tree	/img/tree.png	...
2	Flower	/img/flo.jpg	...
3	Leaf	/img/23814412.jpg	...
...	...	...	

Datentypen in SQlite

SQLite kennt folgende Datentypen.

INTEGER : Ganze Zahlen beliebiger Größe
REAL : Gleitkommazahlen (Fließkommazahlen)
TEXT : Zeichenketten (beliebige Länge)
BLOB : Binärdaten (kleine Datenmengen!)
NULL : Kein Wert vorhanden

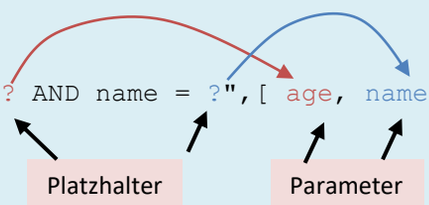
Anmerkung TW zu NULL*: Ich vermeide NULL Werte in meinen Projekten, dies kann beim Erstellen durch den Parameter NOT NULL erreicht werden. Der Parameter default setzt dann einen Wert, der nicht NULL ist. Ein leerer String "" enthält zwar keine Zeichen, ist aber ein vollwertiger String, aber eben ohne Zeichen. NULL macht oft Probleme beim Zugriff. Viele Programmierer nutzen den NULL Wert allerdings als Stilmittel um wirklich leere Datensätze anzuzeigen.*

Datenbanksysteme – hier als Beispiel mySQL (zusätzlicher eigenen Server für große Datenmengen) kennt noch mehr Datentypen.

INT / INTEGER : Ganze Zahlen (z. B. 0, 1, 100)
BIGINT : Sehr große ganze Zahlen
DECIMAL(p,s) : Feste Dezimalzahlen, z. B. für Geldbeträge
FLOAT / DOUBLE : Gleitkommazahlen
CHAR(n) : Feste Zeichenlänge (z. B. 'JA', 'NEIN')
VARCHAR(n) : Variable Zeichenketten (z. B. Namen, Texte)
TEXT : Lange Zeichenketten (z. B. Artikelbeschreibungen, Markdown)
DATE : Datum (JJJJ-MM-TT), Beispiel 2025-04-30
DATETIME / TIMESTAMP : Zeitpunkt (Datum und Zeit). Beispiel 2025-04-30T12:03:10+03:00 (das +03:00 bedeutet die Zeitverschiebung zu Greenwich Mean Time und ändert sich bei Sommer und Winterzeit)
BLOB : Binärdaten - hier sollten nie große Datenmengen gespeichert werden.
BOOLEAN : Wahr/Falsch (intern als TINYINT gespeichert)

SQL Queries immer mit Parameter ausführen anstatt mit String Operationen (verhindert viele SQL Injections)

```
let name = req.body.name || ``;
let age = req.body.age || 0;
db.query("SELECT * FROM users WHERE age > ? AND name = ?",[ age, name ] …
```



SQL Queries (optionale Info!!!)

- SQL ist eine eigene Skript-Sprache für Datenbankabfragen. In den Seeds sind viele Beispiele zu finden.
- Abfragen sollten mit Parametern ausgeführt sein.

Mit Joins können Tabellen miteinander kombiniert werden.

SQL CHEAT SHEET http://www.sqltutorial.org

QUERYING DATA FROM A TABLE

```
SELECT c1, c2 FROM t;
```
Query data in columns c1, c2 from a table

```
SELECT * FROM t;
```
Query all rows and columns from a table

```
SELECT c1, c2 FROM t
WHERE condition;
```
Query data and filter rows with a condition

```
SELECT DISTINCT c1 FROM t
WHERE condition;
```
Query distinct rows from a table

```
SELECT c1, c2 FROM t
ORDER BY c1 ASC [DESC];
```
Sort the result set in ascending or descending order

```
SELECT c1, c2 FROM t
ORDER BY c1
LIMIT n OFFSET offset;
```
Skip *offset* of rows and return the next n rows

```
SELECT c1, aggregate(c2)
FROM t
GROUP BY c1;
```
Group rows using an aggregate function

```
SELECT c1, aggregate(c2)
FROM t
GROUP BY c1
HAVING condition;
```
Filter groups using HAVING clause

QUERYING FROM MULTIPLE TABLES

```
SELECT c1, c2
FROM t1
INNER JOIN t2 ON condition;
```
Inner join t1 and t2

```
SELECT c1, c2
FROM t1
LEFT JOIN t2 ON condition;
```
Left join t1 and t1

```
SELECT c1, c2
FROM t1
RIGHT JOIN t2 ON condition;
```
Right join t1 and t2

```
SELECT c1, c2
FROM t1
FULL OUTER JOIN t2 ON condition;
```
Perform full outer join

```
SELECT c1, c2
FROM t1
CROSS JOIN t2;
```
Produce a Cartesian product of rows in tables

```
SELECT c1, c2
FROM t1, t2;
```
Another way to perform cross join

```
SELECT c1, c2
FROM t1 A
INNER JOIN t2 B ON condition;
```
Join t1 to itself using INNER JOIN clause

USING SQL OPERATORS

```
SELECT c1, c2 FROM t1
UNION [ALL]
SELECT c1, c2 FROM t2;
```
Combine rows from two queries

```
SELECT c1, c2 FROM t1
INTERSECT
SELECT c1, c2 FROM t2;
```
Return the intersection of two queries

```
SELECT c1, c2 FROM t1
MINUS
SELECT c1, c2 FROM t2;
```
Subtract a result set from another result set

```
SELECT c1, c2 FROM t1
WHERE c1 [NOT] LIKE pattern;
```
Query rows using pattern matching %, _

```
SELECT c1, c2 FROM t
WHERE c1 [NOT] IN value_list;
```
Query rows in a list

```
SELECT c1, c2 FROM t
WHERE  c1 BETWEEN low AND high;
```
Query rows between two values

```
SELECT c1, c2 FROM t
WHERE  c1 IS [NOT] NULL;
```
Check if values in a table is NULL or not

SQL CHEAT SHEET http://www.sqltutorial.org

MANAGING TABLES

```
CREATE TABLE t (
   id INT PRIMARY KEY,
   name VARCHAR NOT NULL,
   price INT DEFAULT 0
);
```
Create a new table with three columns

```
DROP TABLE t;
```
Delete the table from the database

```
ALTER TABLE t ADD column;
```
Add a new column to the table

```
ALTER TABLE t DROP COLUMN c;
```
Drop column c from the table

```
ALTER TABLE t ADD constraint;
```
Add a constraint

```
ALTER TABLE t DROP constraint;
```
Drop a constraint

```
ALTER TABLE t1 RENAME TO t2;
```
Rename a table from t1 to t2

```
ALTER TABLE t1 RENAME c1 TO c2;
```
Rename column c1 to c2

```
TRUNCATE TABLE t;
```
Remove all data in a table

USING SQL CONSTRAINTS

```
CREATE TABLE t(
   c1 INT, c2 INT, c3 VARCHAR,
   PRIMARY KEY (c1,c2)
);
```
Set c1 and c2 as a primary key

```
CREATE TABLE t1(
   c1 INT PRIMARY KEY,
   c2 INT,
   FOREIGN KEY (c2) REFERENCES t2(c2)
);
```
Set c2 column as a foreign key

```
CREATE TABLE t(
   c1 INT, c1 INT,
   UNIQUE(c2,c3)
);
```
Make the values in c1 and c2 unique

```
CREATE TABLE t(
 c1 INT, c2 INT,
 CHECK(c1> 0 AND c1 >= c2)
);
```
Ensure c1 > 0 and values in c1 >= c2

```
CREATE TABLE t(
   c1 INT PRIMARY KEY,
   c2 VARCHAR NOT NULL
);
```
Set values in c2 column not NULL

MODIFYING DATA

```
INSERT INTO t(column_list)
VALUES(value_list);
```
Insert one row into a table

```
INSERT INTO t(column_list)
VALUES (value_list),
       (value_list), ....;
```
Insert multiple rows into a table

```
INSERT INTO t1(column_list)
SELECT column_list
FROM t2;
```
Insert rows from t2 into t1

```
UPDATE t
SET c1 = new_value;
```
Update new value in the column c1 for all rows

```
UPDATE t
SET c1 = new_value,
    c2 = new_value
WHERE condition;
```
Update values in the column c1, c2 that match the condition

```
DELETE FROM t;
```
Delete all data in a table

```
DELETE FROM t
WHERE condition;
```
Delete subset of rows in a table

Views (optionale Info!!!))

SQL CHEAT SHEET http://www.sqltutorial.org

MANAGING VIEWS

```
CREATE VIEW v(c1,c2)
AS
SELECT c1, c2
FROM t;
```
Create a new view that consists of c1 and c2

```
CREATE VIEW v(c1,c2)
AS
SELECT c1, c2
FROM t;
WITH [CASCADED | LOCAL] CHECK OPTION;
```
Create a new view with check option

```
CREATE RECURSIVE VIEW v
AS
select-statement -- anchor part
UNION [ALL]
select-statement; -- recursive part
```
Create a recursive view

```
CREATE TEMPORARY VIEW v
AS
SELECT c1, c2
FROM t;
```
Create a temporary view

```
DROP VIEW view_name;
```
Delete a view

MANAGING INDEXES

```
CREATE INDEX idx_name
ON t(c1,c2);
```
Create an index on c1 and c2 of the table t

```
CREATE UNIQUE INDEX idx_name
ON t(c3,c4);
```
Create a unique index on c3, c4 of the table t

```
DROP INDEX idx_name;
```
Drop an index

SQL AGGREGATE FUNCTIONS

AVG returns the average of a list

COUNT returns the number of elements of a list

SUM returns the total of a list

MAX returns the maximum value in a list

MIN returns the minimum value in a list

MANAGING TRIGGERS

```
CREATE OR MODIFY TRIGGER trigger_name
WHEN EVENT
ON table_name TRIGGER_TYPE
EXECUTE stored_procedure;
```
Create or modify a trigger

WHEN
- **BEFORE** – invoke before the event occurs
- **AFTER** – invoke after the event occurs

EVENT
- **INSERT** – invoke for INSERT
- **UPDATE** – invoke for UPDATE
- **DELETE** – invoke for DELETE

TRIGGER_TYPE
- **FOR EACH ROW**
- **FOR EACH STATEMENT**

```
CREATE TRIGGER before_insert_person
BEFORE INSERT
ON person FOR EACH ROW
EXECUTE stored_procedure;
```
Create a trigger invoked before a new row is inserted into the person table

```
DROP TRIGGER trigger_name;
```
Delete a specific trigger