Build a Data Structure for your Application

- Cheatsheet -

1: Get Informed - How to make information persistent with a database

Assuming you have an understanding of your customers demand, e.g. via a workflow or user stories, take a moment to understand how database logic works. In a nutshell, **databases are information storages that keep the input of your users persistent, even when the application is closed**

1. SQL Databases, e.g. MySQL, MSSQL, SQLite
Historically, the most used databases are so-called SQL databases. Within each of these, several tables are being stored. **Each table has columns**, which contain predefined data types, say **characters** (alphanumeric), **numeric** (numbers) or **booleans** (true / false).
Values in this database can be accessed with keys. Each table has one or several of these, which allow for data to be uniquely recognizeable.

Column	Type	Key
Customer ID	Numeric	X
First Name	Character	
Second Name	Character	
Adress	Character	
Street	Character	

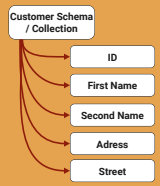
Based on this structure, an actual table is created. This can then be filled out with values. These values, again, can be used to query another table as specific key fields. This approach to store and access information is often referred to as relational databases.

Supply

Filled out table with applied table structure

Customer ID	First Name	Second Name	City	Street
1	John	Doe	Wisconsin	Washington Str. 14
2	Jane	Doe	Chicargo	Hollywood Boulveard 25
3	John	Smith	Washington	Chicargo Av. 38

2. NoSQL Databases, e.g. MongoDB, CouchDB
Over the last years, NoSQL databases received a growth in popularity. In comparison to SQL databases, **NoSQL follows a document approach**, meaning data is not (only) stored in several tables, but in a single entity. A single document does not have a predefined structure. This brings several advantages, e.g. quick & dirty input-output handling, but also includes the risk of several documents having distinct structures, making data queries impossible.



Therefor, it is good practice to give documents a structure while being created. Instead of predefining a table, it is common practice to work with so called schema. A schema also permits several documents to be grouped in a collection.

2. Decide what's the best approach to store your application's data

Deciding on a way to save and process your data requires a certain degree of technical knowledge, while depending on the type of data being stored themselves. As a rule of thumb, the tabular approach (1) is better for complex data relations, while the collection approach (2) is to be favored when dealing with a big amount of non-distinct data. In any way, you will want to plan the needed architecture.

Take the following business case
A customer would like to have a database in which he can store the locations of his stores and access related customerdata via their ID. **> Relational approach with two database tables:**

City Structure

Column	Type	Key
City Postal Code	Numeric	X
City Name	Character	
City Customer IDs	Numeric	

Customer Structure

Column	Type	Key
Customer ID	Numeric	X
First Name	Character	
Second Name	Character	
Adress	Character	
Street	Character	

City Structure

Column	Type	Key
City Postal Code	Numeric	X
City Name	Character	
City Customer IDs	Numeric	

Example of a table's value being used to query another table (City=>Customer)

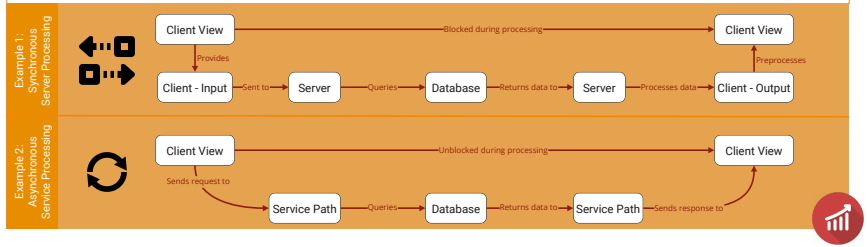
Query

Customer Structure

Column	Type	Key
Customer ID	Numeric	X
First Name	Character	
Second Name	Character	
Adress	Character	
Street	Character	

3. Decide how to process queries

While building the structure for your database, you should have a strategy on how to make these accessible to a requesting client. Also, consider whether you want your queries to happen synchronously or asynchronously.



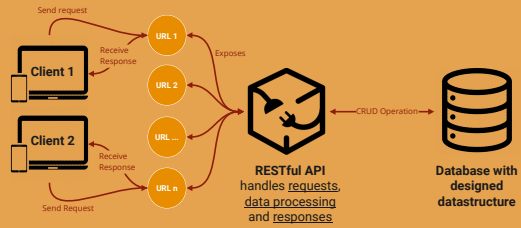
4. Make your data consumable

After designing and establishing a data structure, it has to be made available to your application. The majority of web applications uses the http - protocol and related methods to **Create, Read, Update and Delete** entries in DB's. A very popular way of implementing such interface is the **REST architecture**

Representational State Transfer
REST interfaces allows to decouple a client that operates with data from the server that holds it. In a nutshell, for an API to be RESTful, it has to consider:

- Client - Server architecture
- Statelessness
- Cacheability
- Layered system
- Code on demand
- Uniform interface

These constraints allow for maximum flexibility when building backend services



API

5. (Optional) Implement models or state management in the frontend

Now that your data can be consumed, it makes sense to establish a structure on the client side that makes communication between frontend and backend easier. It's a tradeoff between short term and long time productivity, however, therefor implementation might not make sense in every case.

```
// My Api Model to handle
const myApiModel = {
  // Define the necessary endpoints for the business case
  domain: "https://myapi.com",
  customerUrl: "/customer",
  storeUrl: "/store",

  // Define the methods that take use of these endpoints
  getCustomerByCustomerId = async function (customerId) {

    // Get a single customer by his id and return it to the calling function
    const customer = await fetch(this.domain + this.customerUrl + customerId)
    return await customer.json();
  },
  getCustomerByStoreId = async function (storeId) {

    // Get a single store by its ID
    const store = await fetch(this.domain + this.storeUrl + storeId)

    // Then, using this store, get a set of customers related to it
    const customers = await fetch(this.domain + this.customerUrl + store.json())
    return await customers.json()
  }
}
```

Consider the example on the left with REST principles.
A class / object on the frontend that deals with these data might look like this (Javascript Code)

It is meant to fulfill the following business requirements:

- Return a single customer by its ID
- Return a set of customers by a store ID

There are several ways on how to do proper state management. If you use one of these, you should commit to keep it till the project is done.

- The ELM architecture
- FLUX by Facebook
- The MVC Model
- The MV-VM Model