REST

> REST = Representational State Transfer
>
> Rest means that we are transferring the representational states between the client and the server.

- There are 6 REST constraints an API must satisfy to be considered RESTful
- If an API is **RESTful** it means that it satisfies the REST cosntraints.

REST constraints

1. Client Server
2. Stateless
3. Cache
4. Uniform Interface
5. Layered System
6. Code-On-Demand (*optional*)

1. Client Server

- The system must be made up of **Clients** and **Servers**.
- Servers have resources that Clients want to use.
- **Separation of Concerns** (reduces complexity, improves scalability):
  - **Server**: Handles back-end stuff, like data storage, business rules, ...
  - **Client**: Handles front-end stuff, like user interface
- HTTP is the most common protocol for client-server communication, however, you could also use any other fitting protocol for the communication.

2. Stateless

- The communication between the Server and the Client must be **stateless**, which simplifies the communication.
- All information about the client's session is kept on the client, the server doesn't know anything about it.
  - → There should be no **browser cookies**, **session variables** or other stateful features.
  - → As a consequence, a request must contain all information needed to perform the request.
- The stateless constraint simplifies the server, since there is no need to keep track of the client sessions.
- The system is easier to understand. You only need to consider the input data for a request and what output data it resulted in. So there is no need to look up session variables or other stateful stuff.
- Stateless communication improves the reliance of the system, since no session context can got corrupted.

3. Cache

- Responses from Servers must be marked as **cacheable** or **non-chacheable**.

- An effective cache can reduce the number of client-server interactions and thus contribute to a better performance of the system.

4. Uniform Interface

- The **uniform interface** decouples the **interface** from the actual **implementation**, which makes interactions simpler.
- The **uniform interface** constraint is made up of 4 sub-constraints:
  - Identification of Resources
  - Manipulation of Resources through Representations
  - Self-Descriptive Messages
  - Hypermedia as the Engine of Application State

4.1 Identification of Resources

- The REST style is centered around resources.
- A resource is anything that can be named, e.g. static pictures, stock prices, ...
- Each resource in a RESTful design must be **uniquely identifiable** via an **URI**, and the identifier must be stable even when the underlying resource is updated.
  - "Cool URIs don't change"
  - Resources you want to expose through a REST API must have its own URI.
- Pattern to access a **collection resource** (e.g. customers):
  - `https://api.example.com/customers`
  - The resource name is usually in plural, because it is a collection of resources, rather than a single one.
- Query specific elements of the collection resource, by using a query parameter:
  - `https://api.example.com/customers?lastname=Skywalker`
- Access individual items of the collection by appending the unique identifier of the item:
  - `https://api.example.com/customers/932612`
- An example for a not resource-oriented API is Flickr, e.g.
  - `https://api.flickr.com/services/rest/?method=flickr.galleries.addPhoto`
  - The `addPhoto` method name means that it is not resource-oriented, but instead an RPC interface.

4.2 Manipulation of Resources through Representations

- In a Uniform Interface the resources are manipulated through **representations**.
- This means that the client doesn't interact directly with the server's resources → Instead, the server exposes a representation of the resource's state, which means that we show the resource's data in a neutral format.
- The most common format for REST APIs is **JSON**.
- When a client wants to update a resource, it gets the representation of it, changes it, and then asks the server to update it according to the representation sent back by the client.
- Manipulation through representations avoids strong coupling between client and server.
- Clients don't need to understand the underlying technology.
  - → This is a great opportunity for legacy systems to expose their data in a style that is much easier to understand for modern API clients.

4.3 Self-Descriptive Messages

- In a Uniform Interface a request or response message must include enough information for the receiver to understand it in isolation.
- Each message must have a media type that tells the receiver how the message should be parsed, e.g. `application/json`, `application/xml`, ...
- If HTTP and HTTP methods are used, then also their formal meaning should be followed, e.g. GET, POST, PUT, DELETE
- Allows everyone to understand the messages in isolation.

4.4 Hypermedia as the Engine of Application State (HATEOAS)

- A webpage is an instance of application state and hypermedia is text with hyperlinks.
- We should use links to navigate through the application (states).
- The benefit is, that the API user doesn't need to look in the API documentation to see how to find the customer's orders.

5. Layered System

- The client should only know the immediate layer, it is communicating with, and not know of any layers that may or may not be behind the server.
- If we place a proxy between client and server, it wouldn't affect their communication.
- A server can call multiple other servers to generate a response to the client.

6. Code-On-Demand (optional)

- **Code-On-Demand** means that a server can extend the functionality of a client on runtime, by sending code that it should execute.
- Very common in webpages: When your browser accesses a webpage, it almost always downloads a lot of JavaScript specific to the webpage, and hence extends the functionality of your browser on runtime trough code-on-demand.
- Can improve simplicity, since the client code becomes much easier to write.