

**REST APIs:**

- An **API** is a set of definitions and protocols for building and integrating application software. It's sometimes referred to as a contract between an information provider and an information user.  
I.e. If you want to interact with a computer or system to retrieve information or perform a function, an API helps you communicate what you want to that system so it can understand and fulfill the request.  
I.e. An API is a set of rules that allows programs to talk to each other.
- An **endpoint** is the location from which APIs can access the resources they need to carry out their function. For web APIs, endpoints are usually URLs.
- **REST** stands for **representational state transfer** and was created by computer scientist Roy Fielding.
- **REST APIs** are also known as **RESTful APIs**.
- A REST API is an API that conforms to the constraints of REST architectural style and allows for interaction with RESTful web services.  
**Note:** REST is a set of architectural constraints, not a protocol or a standard. API developers can implement REST in a variety of ways.
- When a client request is made via a RESTful API, it transfers a representation of the state of the resource to the requester or endpoint. This representation is delivered in one of several formats via HTTP, of which JSON is the most popular to use because it's language-agnostic and readable by both humans and machines.
- In order for an API to be considered RESTful, it has to conform to these criteria:
  1. A client-server architecture made up of clients, servers, and resources, with requests managed through HTTP.
  2. Stateless client-server communication, meaning no client information is stored between get requests and each request is separate and unconnected.
  3. Cacheable data that streamlines client-server interactions.
  4. A uniform interface between components so that information is transferred in a standard form. This requires that:
    - a. resources requested are identifiable and separate from the representations sent to the client.
    - b. resources can be manipulated by the client via the representation they receive because the representation contains enough information to do so.
    - c. self-descriptive messages returned to the client have enough information to describe how the client should process it.
    - d. hypertext/hypermedia is available, meaning that after accessing a resource the client should be able to use hyperlinks to find all other currently available actions they can take.
  5. A layered system that organizes each type of server involved in the retrieval of requested information into hierarchies, invisible to the client.
  6. Code-on-demand: The ability to send executable code from the server to the client when requested, extending client functionality.
- The function names of a REST API consist of the HTTP method and the URL.
- The function arguments of a REST API consist of a URL and the request body. The return value of a function is a status code and the response body.

- E.g.

	HTTP request	HTTP response
Create a new message	POST /messages/ "Hello World"	200 "78"
Get all messages	GET /messages/	200 "['Hello world', ...]"
Get a specific messages	GET /messages/78/	200 "Hello World"
Delete a specific messages	DELETE /messages/78/	200 "success"

- The server is more or less a storage system that stores:
  1. Collections/Resources
  2. Elements that belong to one or several collections.
 For example, in the example above, in the third row, messages is a collection while 78 is an element.
- Usually, the pattern is collection/element/collection/element/etc.
- E.g.


Type	Example
one-to-one	/users/sansthie/profile/firstname/
one-to-many	/users/sansthie/messages/89/
many-to-many	/users/sansthie/teams/8/ /teams/8/users/sansthie/

Here, in the first row, users is a collection while sansthie is an element, and profile is a collection while firstname is an element.

- REST APIs have 3 relationships:
  1. One-to-one
  2. One-to-many
  3. Many-to-many

**CRUD Operations:**

- CRUD stands for:
  - Create
  - Read
  - Update
  - Delete
- They are 4 basic functions of persistent storage.
- How HTTP methods map to CRUD operations:

CRUD	HTTP	Collection	Element
Create	POST		Create a new element
	PUT	Replace the entire collection	Create (or replace if exists) a specific element
Read	GET	List all elements	Retrieve a specific element
Update	PATCH	Update some attributes of some elements	Update some attributes of a specific element
Delete	DELETE	Delete the entire collection	Delete a specific element

- **PUT vs POST:**

PUT	POST
PUT is idempotent. So if you send a request multiple times, that should be equivalent to a single request modification.	POST is NOT idempotent. So if you send a request N times, you will end up having N resources.
Use PUT when you want to modify a singular resource which is already a part of resources collection. PUT replaces the resource in its entirety. Use PATCH if the request updates part of the resource.	Use POST when you want to add a child resource under resources collection.
Generally, in practice, use PUT for UPDATE operations that replaces the resource in its entirety.	Use POST for CREATE operations.

**Note:** These are just guidelines. The implementation details are left up to the developers.

- We can use attributes to query/filter a subset of a collection.  
E.g. GET /messages/?from=67&to=99

- **HTTP Methods for RESTful Services:**

HTTP Verb	CRUD	Entire Collection (e.g. /customers)	Specific Item (e.g. /customers/{id})
POST	Create	201 (Created), 'Location' header with link to /customers/{id} containing new ID.	404 (Not Found), 409 (Conflict) if resource already exists.
GET	Read	200 (OK), list of customers. Use pagination, sorting and filtering to navigate big lists.	200 (OK), single customer. 404 (Not Found), if ID not found or invalid.
PUT	Update/Replace	405 (Method Not Allowed), unless you want to update/replace every resource in the entire collection.	200 (OK) or 204 (No Content). 404 (Not Found), if ID not found or invalid.
PATCH	Update/Modify	405 (Method Not Allowed), unless you want to modify the collection itself.	200 (OK) or 204 (No Content). 404 (Not Found), if ID not found or invalid.
DELETE	Delete	405 (Method Not Allowed), unless you want to delete the whole collection—not often desirable.	200 (OK). 404 (Not Found), if ID not found or invalid.

**Note:** Once again, these are just the guidelines.

**Handling Data:**

- To handle data, we'll use a database.
- We use a database because:
  - Persistency
  - Concurrency (avoid race conditions)
  - Query
  - Scalability

- **SQL vs NOSQL databases:**

Parameter	SQL	NOSQL
Type	Are table based databases	Can be document based, key-value pairs, or graph databases.
Schema	Have a predefined schema	Use dynamic schema for unstructured data.
Ability to scale	Are vertically scalable	Are horizontally scalable

- The concept of NoSQL databases became popular with internet giants like Google, Facebook, Amazon, etc who deal with huge volumes of data. The system response time becomes slow when you use RDBMS for massive volumes of data.

To resolve this problem, we could **scale up** our systems by upgrading our existing hardware. This process is expensive. (SQL uses scaling up.)

The alternative for this issue is to distribute the database load on multiple hosts whenever the load increases. This method is known as **scaling out**. (NOSQL uses scaling out.)

Relational database (SQL database)	
Data structure	tables and tuples
Query language	SQL
Inconvenient	not-optimized for big data analysis
Advantage	complex queries
Technology	<i>PostgreSQL, MySQL, MariaDB, SQLite, MSSQL</i>

NoSQL database	
Data structure	key/value pairs
Query language	API style
Inconvenient	not adequate for complex queries
Advantage	optimized for big data analysis
Technology	<i>MongoDB, Redis, CouchDB, NeDB</i>

- **ORM (Object Relational Mapping):**
- **ORM** is a technique that lets you query and manipulate data from a database using an object-oriented paradigm. It provides a mapping between objects and the database structure.
- An ORM library is a library written in your language of choice that encapsulates the code needed to manipulate the data, so you don't use SQL anymore. You interact directly with an object in the same language you're using.
- A few advantages of using ORMs are:
  - You write your data model in only one place, and it's easier to update, maintain, and reuse the code. (DRY principle)
  - It sanitizes the query statement, preventing SQL injections.
- Examples:
  - Sequelize for PostgreSQL, MySQL, MariaDB, SQLite
  - Mongoose for MongoDB

**Connecting the REST API with a database:**

- Do retrieve selected elements only rather than retrieving an entire collection and filtering afterwards. This is because the database is going to be bigger than the memory capacity of the backend, so if you retrieve an entire collection, it could be dangerous.
- Do define primary keys rather than relying on auto-generated ones.
- Do split data into different collections rather than storing list attributes.
- Do create join collections whenever appropriate (only for NoSQL databases without performant join feature).
- Only retrieve what you need from a potentially large collection. (Use pagination)

**Handling Files:**

- Recall that JavaScript in the browser cannot open and read files on the user's computer/desktop.
- The only way to upload files is through file input forms.  
I.e.  

```
<form . . . >
  <input type="file" name="img" multiple>
  (The multiple lets users upload multiple files at once.)
  <input type="submit">
</form>
```
- There are 2 ways to send a file from the browser:
  1. Old Way:
    - Form action (with page refresh)  
E.g.  

```
<form action="/url"
      method="POST"
      enctype="multipart/form-data">
```
    - **Note:** The enctype="multipart/form-data" tells the server that we're going to send a form that contains both text and binary data.
  2. New Way:
    - Ajax request (without page refresh)  
E.g.  

```
var file = document.get ...
var formdata = new FormData();
formdata.append("picture", file);
xhr.send(formdata);
```
    - **Note:** We use FormData because we're sending a mix of binary and text.
- The best approach is to store files on discs, but you can store files in databases.
- The server receives the following information:
  - File metadata, which includes the filename, mimetype (file type), size and others.
  - File content which is a compressed binary or string.
- **MIME Type:**
- **MIME (Multipurpose Internet Mail Extensions)** is also known as the content type.
- It defines the format of a document exchanged on the internet.
- **Do/Don't with files:**
- Do not send a base64 encoded file content with JSON, use multipart/form-data instead (compression).
- Do not store uploaded files with the static content.
- Do not serve uploaded files statically. If you upload files statically, everyone can access it, causing security issues.
- Do store the mimetype and set the HTTP response header mimetype when files are sent back.