

# Session 5

## 5. HTTP, REST API Concepts and Fetch

### 5.1 How do two computers talk to each other?

Imagine your computer (at home) wants to talk to another computer thousands of miles away — say, Google's. Here's how it happens, in the simplest terms:


- First, your computer needs to *know where to send the message*. This is its **IP address**, like a home address written on an envelope.
- Then, it needs a reliable *delivery method*, that's **TCP**, the system that makes sure your message arrives safely and in the right order, like a trusted mail service.
- Finally, the computers need a *common language* to exchange messages on the web, that's **HTTP**.

So when you type in Google, your computer:

1. Looks up Google's **address (IP)**.
2. Uses **TCP** to send a request: "Hey, Google, I want the homepage."
3. Google replies in the same language — **HTTP** — "Here's your homepage."

### What about HTTPS?

- **HTTP** is like speaking aloud in public, anyone can overhear.
- **HTTPS** is like using a secret code language, hidden, encrypted, so only the two computers understand.

That's why you see the  **lock icon** — it means you're using that secret-code language.

Steps to try out a new API :

- Search google for a free api endpoint
  - read the documentation for the said api
  - test it out in postman
  - code it out
-

## 5.2 What challenge do computers face when sharing data?

Different computers and apps often store data in **different formats**.

One might keep info in a database, another in a file, another in memory.

👉 The challenge:

How can one computer **ask for data** from another without knowing *how* that data is stored internally?

💡 **That's where APIs come in.**

APIs define a **standard way to request and exchange data**, so systems can communicate clearly without needing to know each other's inner workings.

---

## 5.3 What is an API?

API stands for **Application Programming Interface**. It's like a **waiter at a restaurant**:

- In the restaurant analogy:
  - You (the app) tell the waiter (API) what you'd like from the kitchen (server).
  - The waiter brings the cooked meal back to you.

**Key point:** You don't need to know how things work under the hood, you just need a clear set of instructions to get what you need.

---

## 5.3 What is a REST API?

REST means **Representational State Transfer**. Think of it like ordering from a **menu at the restaurant**:

- **Endpoints** = Menu items (what you can ask for)
- **HTTP methods** = Actions you can take:
  - **GET** → ask for a food item
  - **POST** → order a new dish
  - **PUT** → modify your order (e.g., extra cheese)
  - **DELETE** → cancel your dish

Each request is independent, and you follow a standard set of rules—like ordering from a familiar menu every time.

---

## 5.4 Common HTTP Methods

Method	What it does	Real-world analogy
GET	Retrieve data	Ask waiter for your order
POST	Send new data	Place a new order on menu
PUT	Update existing data	Change your burger toppings
DELETE	Remove data	Cancel your order

---

## 5.5 Common Status Codes

Code	Meaning	Analogy
200	OK	Waiter brings correct dish
201	Created	Chef made a new dish just for you
400	Bad Request	You misspelled the dish name
401	Unauthorized	You didn't pay (authentication fail)
403	Forbidden	You're not allowed to order it
404	Not Found	That dish isn't on the menu
500	Server Error	Restaurant kitchen is on fire

These codes help you understand what happened with your request—success, failure, or something else.

---

## 5.6 What is Postman?

Think of Postman as a **virtual API café**:

- You choose a method ( `GET` , `POST` , etc.)
  - Type in the URL
  - Click **Send**
  - See the response—all without writing a single line of code.
-

## 5.7 Testing an API in Postman

Follow these steps:

1. Open Postman
2. Select method → **GET**
3. URL → `https://api.chucknorris.io/jokes/random`
4. Click **Send**
5. See your response:

```
{  
  "value": "Chuck Norris doesn't wear a watch. He decides what time it is."  
}
```

Just like ordering a joke from a funny waiter.

## Making API calls from code

Before we dive into `fetch()`, let's answer one thing:

👉 How does your app connect with the outside world?

Your app lives on your computer, but the data you need (weather, jokes, news, etc.) lives on **other computers (servers)**. To connect them, you make **API calls** — and in JavaScript, the easiest way to do this is with `fetch()`.

## 5.8 What is fetch()?

`fetch()` is a **function** that helps you make **API requests** — basically, it lets your app **talk to other apps or servers**.

Imagine asking a website:

| "Hey, give me a random joke."

`fetch()` sends that request, waits for the response, and gives you the data back.

## 5.9 Why do we use fetch()?

You use `fetch()` when you want your app to get some information for you. Some examples are listed below:

- Get weather info from a weather API
- Get news from a news API
- Get jokes from a joke API (like we'll do!)

Instead of building everything yourself, **you ask existing services for data.**

---

## 5.10 What does `fetch()` return?

It returns a **Promise**.

That means:

- It doesn't give you the result immediately
  - Instead, it says: "I'll give you the result later"
  - You need to **wait** for it using either:
    - `.then()` chaining
    - `async/await` syntax (cleaner)
- 

## 5.11 How Do You Handle Errors?

You use a `try...catch` block. Inside `try`, you write the code that *might fail*. Inside `catch`, you write what to do *if* it fails.

For example:

```
try {  
  const response = await fetch(...);  
  const data = await response.json();  
} catch (error) {  
  console.error("Something went wrong:", error.message);  
}
```

This way, your app **doesn't crash** if:

- There's no internet
- The API is broken

- You typed the URL wrong

## 5.12 Two Ways to Use fetch()

### Method 1: Using `.then()`

```
fetch('https://api.chucknorris.io/jokes/random')
  .then(response => response.json())    // convert raw data to JSON
  .then(data => console.log(data.value)) // access and log the joke
  .catch(err => console.error("Error:", err));
```

### Code Breakdown

- `fetch()` → makes request (returns Promise).
- `res.json()` → converts raw response to JS object.
- `data.value` → contains the actual parsed data from fetch(actual joke).
- `.catch()` → handles errors if anything goes wrong.
- Once the JSON is ready, we now have the actual **data** from the API.
- `data` is an object that looks like this:

```
{
  "categories": [],
  "created_at": "...",
  "value": "Chuck Norris doesn't read books. He stares them down until
  he gets the information he wants."
  ...
}
```

### Method 2: Using `async/await`

```
async function getJoke() {
  try {
    const res = await fetch('https://api.chucknorris.io/jokes/random');
    const data = await res.json();
    console.log(data.value);
  }
}
```

```
} catch (err) {  
  console.error("Failed to fetch joke", err);  
}  
}  
  
getJoke();
```

## Code Breakdown

- `await fetch()` → waits for response.
- `await res.json()` → parse to JS object.
- `data.value` → contains the actual parsed data from fetch.
- `catch` → handle errors if anything goes wrong.
- Now `data` is a usable object like:

```
{  
  "categories": [],  
  "created_at": "...",  
  "value": "Chuck Norris can hear sign language."  
  ...  
}
```

- We print just the joke using `data.value`.

This version:

- Looks **cleaner** ✓
- Reads like **normal top-to-bottom** code ✓
- Is easier to **debug and understand** ✓

Overall, this method is **cleaner and more readable**, especially when working with multiple async steps.

---

## 5.13 Using an API in Node.js (Quick Steps)

1. **Identify** the API endpoint (search online if necessary).
2. **Review** the official documentation thoroughly.

3. **Test** the endpoint in Postman to verify it works.
  4. **Implement** the API call in your Node.js application.
- 

SSSSO Vidyakshina 2025

