**Lesson 3: HTTP Requests & Dev Tools**

1. URI Hierarchies

Imagine your API is like a **vending machine**, and each button is labeled with what you want.

Now instead of pressing buttons, you type a special **URL** (called a URI) to tell your website **what you want to do** and **where** to do it.

A **URI** is like an address for a piece of data. You use it to find, add, change, or delete things — like contacts.

We use **paths** that follow a pattern to keep everything neat.

Here’s an example with "contacts" (people in your address book):

| **What you want to do** | **What you type (URI path)** |
| --- | --- |
| See **all** contacts | /contacts |
| See **one** contact (by ID) | /contacts/123 |
| Add a **new** contact | /contacts |
| Update a contact (by ID) | /contacts/123 |
| Delete a contact (by ID) | /contacts/123 |

***\* The number (like 123)*** *is the ID of the   
contact you want to change.*

In Code Example:

app.get('/contacts', (req, res) => {

res.send('All contacts');

});

app.get('/contacts/:id', (req, res) => {

res.send('Contact with ID ' + req.params.id);

});

Tips:

* Use **plural nouns** (/contacts, not /contact)
* Think of **folders**: /contacts is the folder, /contacts/5 is a file inside it.

Additional Resources:

* [URI - MDN Web Docs Glossary: Definitions of Web-related terms | MDN](https://developer.mozilla.org/en-US/docs/Glossary/URI)
* [Difference Between URL and URI | GeeksforGeeks](https://www.geeksforgeeks.org/difference-between-url-and-uri/)

2. HTTP POST (Create New Contact)

**POST** is like **sending a form** to a website to add new stuff. When you use a **POST request**, you're saying: "Hey server, here's some new information I want you to save." Use POST when you are **adding** something new to the database (like a new user, new product, or new contact), **sending form data** from a web page, or doing something that **changes the data.**

Let’s say you run a donut shop. You want to add a new donut to your menu.

You send a POST request to: /donuts

With this **new donut info** in the request:

{

"flavor": "Chocolate",

"frosted": true,

"sprinkles": true

}

The server saves it, and replies with something like:

{

"message": "New donut created!",

"id": "abc123"

}

Tips:

In Code Example:

* You send data to /contacts
* The server accepts it and sends back a **201 status** (means: “created successfully”)

const express = require('express');

const app = express();

app.use(express.json()); // lets us read JSON body

app.post('/contacts', (req, res) => {

const newContact = req.body;

console.log(newContact); // Show the data

res.status(201).send({ message: 'Contact created!', id: 'abc123' });

});

app.listen(3000, () => console.log('Server running on port 3000'));

* Use POST for creating new resources
* Always validate the data (make sure required fields are there)
* Return the **new item’s ID** or a success message
* Use **status 201** if created successfully

Additional Resources:

* [Difference between PUT and POST in REST API](https://restfulapi.net/rest-put-vs-post/)
* [POST - HTTP | MDN](https://developer.mozilla.org/en-US/docs/Web/HTTP/Reference/Methods/POST)

3. HTTP POST (Create New Contact)

**PUT** is used to **update an existing item** in your database — like editing a contact’s phone number. Think of it like saying: “Hey server, replace this old thing with this updated version.” You tell the server *which item* to update by including its **ID** in the URI.

Example: PUT /contacts/123  
This means: “Find the contact with ID 123 and replace it with this new version I’m sending.”

*Important: PUT usually* ***replaces the whole item****, not just one part.   
(Though in practice, some APIs allow partial updates.)*

Tips:

In Code Example:

app.put('/contacts/:id', (req, res) => {

const contactId = req.params.id;

const updatedContact = req.body;

// Example placeholder for updating in database

console.log(`Updating contact with ID ${contactId}`, updatedContact);

// Respond with 204: No Content (successful, no response body)

res.status(204).send();

});

You’d call this with a tool like Postman or REST Client, sending a **JSON body** like this:

{

"name": "Jane Doe",

"email": "jane@example.com",

"phone": "123-456-7890"

}

* Use PUT when **replacing or updating** an existing item
* Always include the item's **ID in the URL** (e.g. /contacts/123)
* Return a **204 No Content** status if no message is needed back
* Validate the data before saving it (check for required fields)
* Don’t use PUT to create something new — that’s what POST is for

Additional Resources:

* [PUT - HTTP | MDN](https://developer.mozilla.org/en-US/docs/Web/HTTP/Reference/Methods/PUT)
* [Difference between PUT and POST in REST API](https://restfulapi.net/rest-put-vs-post/)

4. HTTP DELETE (Remove a Contact)

**DELETE** is used to **remove** something from your database — like deleting a contact from your list. Think of it like saying: “Hey server, I don’t need this anymore. Please remove it.” You tell the server *what* to delete by including the **ID** in the URL.

Example: DELETE /contacts/123  
This means: “Find the contact with ID 123 and remove it.”

Tips:

In Code Example:

app.delete('/contacts/:id', (req, res) => {

const contactId = req.params.id;

// Placeholder for delete logic

console.log(`Deleting contact with ID ${contactId}`);

// Return 200 to show the delete was successful

res.status(200).send({ message: 'Contact deleted successfully' });

});

You’d call this with a DELETE request to:

http://localhost:3000/contacts/123

*No body is needed — just the ID in the URL.*

* Use DELETE when you want to **remove an item** by its ID
* Return **200 OK** or **204 No Content** to show it worked
* Double-check that the item exists before deleting it
* Be careful — deletes are **permanent** (unless you soft-delete)
* Don’t send a body with DELETE — just use the ID in the URL

Additional Resources:

* [DELETE - HTTP | MDN](https://developer.mozilla.org/en-US/docs/Web/HTTP/Reference/Methods/DELETE)
* [REST API Design: Endpoint to Delete a Large Number of Items – eloquent code](https://eloquentcode.com/rest-api-design-endpoint-to-delete-a-large-number-of-items)

5. MongoDB CRUD

In Code Example, let’s assume you have a contacts collection in MongoDB:

const { MongoClient } = require('mongodb');

const uri = 'your-mongo-uri-here';

const client = new MongoClient(uri);

async function run() {

try {

await client.connect();

const db = client.db('myDatabase');

const contacts = db.collection('contacts');

// Create

await contacts.insertOne({

name: 'Alice',

email: 'alice@example.com',

phone: '555-1234'

});

// Read

const contactList = await contacts.find().toArray();

console.log(contactList);

// Update

await contacts.updateOne(

{ name: 'Alice' },

{ $set: { phone: '555-0000' } }

);

// Delete

await contacts.deleteOne({ name: 'Alice' });

} finally {

await client.close();

}

}

run().catch(console.dir);

CRUD stands for **Create, Read, Update, Delete** — these are the four basic things you can do with data in a database. In MongoDB (a NoSQL database), you use JavaScript-style objects to work with your data. Mongo stores data in documents that look like JSON.

Here’s what each part of CRUD means in MongoDB:

| **Action** | **HTTP Verb** | **MongoDB Method** | **Description** |
| --- | --- | --- | --- |
| Create | POST | insertOne() | Add a new document |
| Read | GET | find() / findOne() | Get one or more documents |
| Update | PUT | updateOne() | Change a document's data |
| Delete | DELETE | deleteOne() | Remove a document |

Tips:

* Always connect to your DB **before** making requests
* Use insertOne for adding one item, insertMany for several
* Use $set to update specific fields
* Use **unique IDs** to read/update/delete documents
* Make sure to handle errors and close the DB connection

Additional Resources:

* [MongoDB CRUD Operations - Database Manual v8.0 - MongoDB Docs](https://www.mongodb.com/docs/manual/crud/)
* [Node.js Driver v6.16 - MongoDB Docs](https://www.mongodb.com/docs/drivers/node/current/quick-start/)

6. Linters & Code Formatters

When you're writing code, it's easy to make **mistakes** or use **inconsistent styles** (extra spaces, missing semicolons, unused variables, etc.). That’s where **linters** and **formatters** help:

* **Linters** check your code for **bugs, errors, and bad practices.**
* **Formatters** fix how your code **looks** (indentation, spacing, etc.).

Together, they help you write **clean**, **consistent**, and **error-free** code — which is super important when working on teams or big projects.

Two common tools:

* **ESLint** – A JavaScript linter that finds and reports problems in your code.
* **Prettier** – A code formatter that makes everything look clean and consistent.

In Code Example (Basic ESLint + Prettier Setup):

**1. Install ESLint and Prettier:**

npm install --save-dev eslint prettier

**2. Set up ESLint:**

npx eslint --init

*Answer the setup questions. Then ESLint will create a config file: .eslintrc.json.*

**3. Add Prettier Config (optional but useful):**

Create a .prettierrc file:

{

"semi": true,

"singleQuote": true,

"tabWidth": 2

}

**4. Example: Before and After**

**Messy code:**

function greet(name){console.log("Hi "+name)}

**After linting & formatting:**

function greet(name) {

console.log('Hi ' + name);

}

You can now run:

npx eslint yourFile.js

npx prettier --write yourFile.js

}

Tips:

* Run linters **before committing** your code (or use pre-commit hooks)
* Linters **help catch bugs early**
* Formatters help keep your code style **consistent across your team**
* You can install **VS Code extensions** for ESLint and Prettier to auto-run as you type

Additional Resources:

* [What is Prettier? · Prettier](https://prettier.io/docs/)
* [Getting Started with ESLint - ESLint - Pluggable JavaScript Linter](https://eslint.org/docs/latest/use/getting-started)