Introduction to Express.js

Express.js is a minimal and flexible Node.js framework that significantly simplifies server-side development. It provides powerful features for building web and mobile applications quickly and efficiently.

With robust support for routing, middleware, and REST APIs, Express has become the go-to choice for developers looking to create scalable backend services with JavaScript.

Setting up Express.js

Install Prerequisites

Install Node.js and npm on your system, then create a new project folder for your application.

Initialize Project

Run the following commands in your terminal:

```
npm init -y
npm install express
```

Create Basic Server

```
const express = require("express");
const app = express();

app.listen(3000, () =>
  console.log("Server running"));
```

This creates a minimal Express application that listens on port 3000. Your server is now ready to handle requests!

Express.js Routes

Routes define how your application responds to client requests at specific endpoints (URLs) with specific HTTP methods. Express provides a simple way to handle these requests:

```
app.get("/hello", (req, res) => {
  res.send("Hello World");
});
```

This example creates a route that responds to GET requests at the "/hello" endpoint with the text "Hello World".

GET

Retrieve data from the server

POST

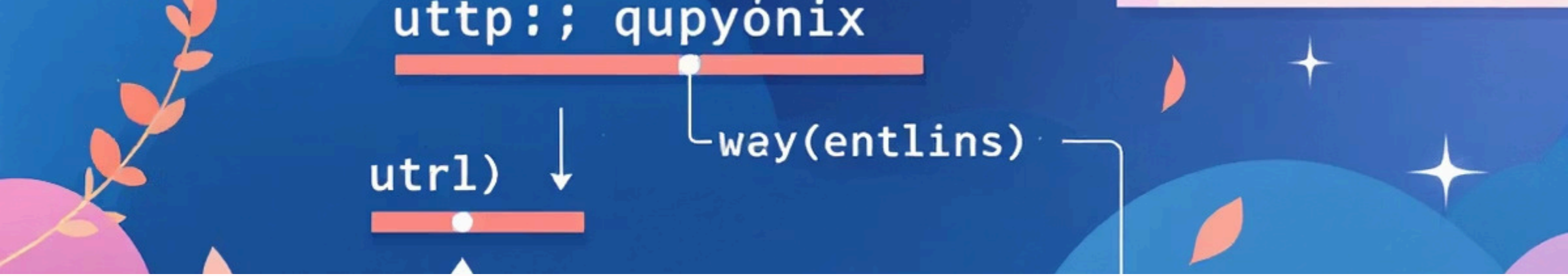
Create new data on the server

PUT

Update existing data

DELETE

Remove data from the server



Parameters & Body Handling

Query Parameters

`/search?name=John`

Access with: `req.query.name`

Used for optional parameters and filtering

Route Parameters

`/user/:id`

Access with: `req.params.id`

Used for required values that are part of the URL path

Request Body

Enable with: `express.json()`

Access with: `req.body`

Used for sending data in POST/PUT requests

Mini Task 1: Creating Basic Routes

Let's practice by creating two simple routes:

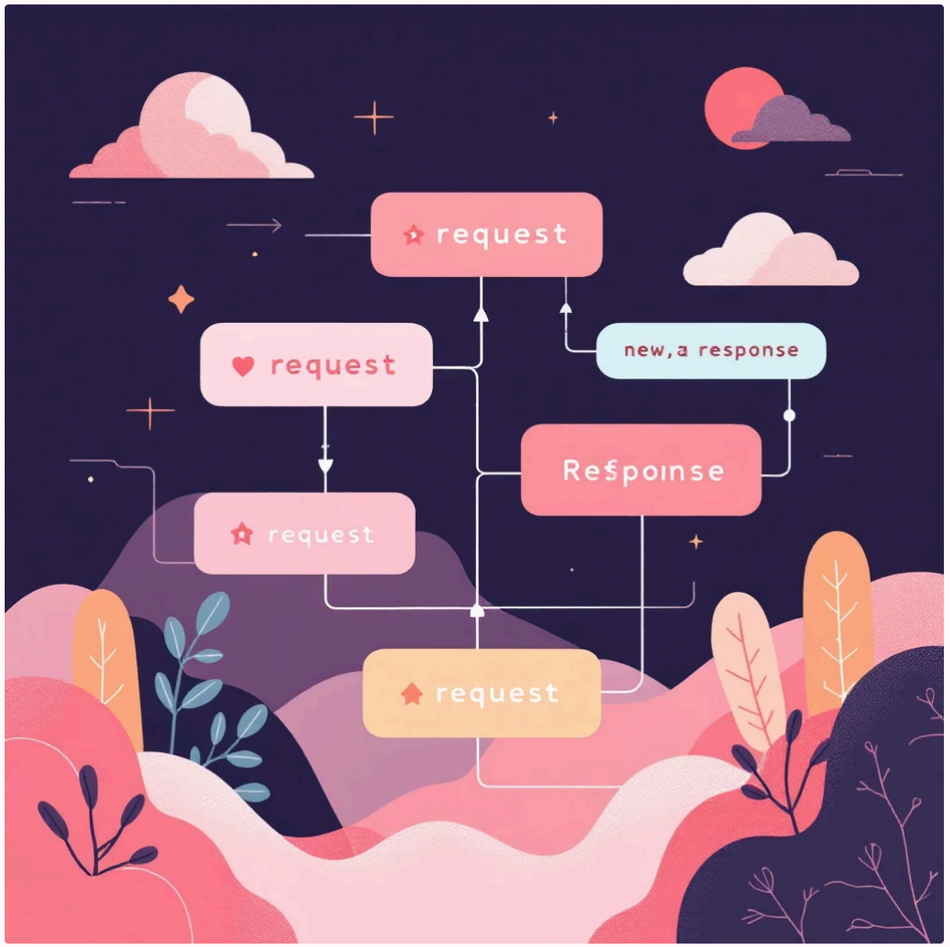
```
// Simple greeting route
app.get("/hello", (req, res) => {
  res.send("Hello, World");
});

// Dynamic route with parameters
app.get("/user/:id", (req, res) => {
  res.send(`Hello, ${req.params.id}`);
});
```

The first route responds with a static message. The second route extracts the `id` parameter from the URL and uses it in the response, creating a personalized greeting.

Try accessing `http://localhost:3000/hello` and `http://localhost:3000/user/Alice` to see different responses.

Middleware in Express.js



What is Middleware?

Middleware functions run **between** receiving a request and sending a response. They can:

- Execute code
- Modify request/response objects
- End the request-response cycle
- Call the next middleware function

```
app.use((req, res, next) => {
  console.log("Request received at:", new Date());
  next(); // Pass control to the next middleware
});
```

The `next()` function is crucial - it passes control to the next middleware function. Without it, your request will hang!

JSON Body Parsing

Enable Body Parsing

Include this middleware to parse JSON bodies from incoming requests:

```
app.use(express.json());
```

Handle JSON Data

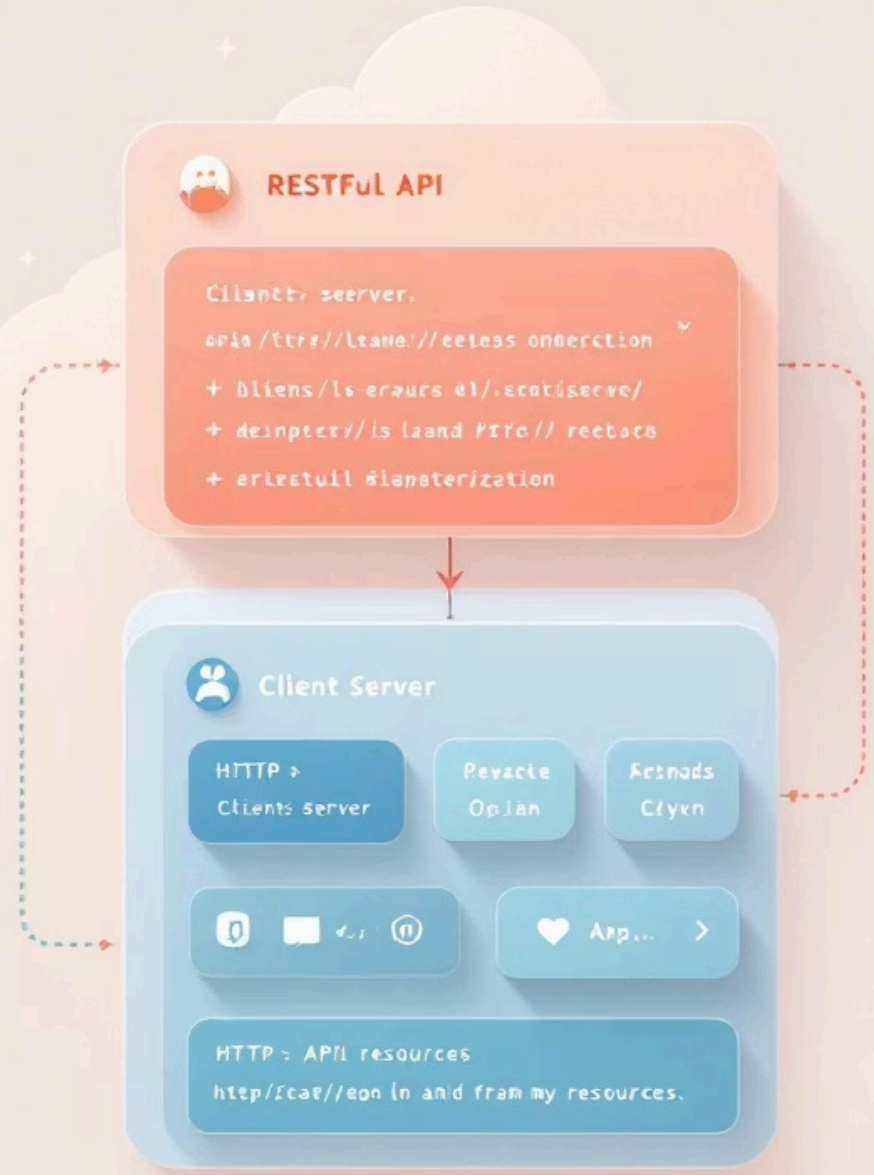
Access request body data in your route handlers:

```
app.post("/data", (req, res) => {
  console.log(req.body);
  res.json(req.body);
});
```

Test with Tools

Use Postman, cURL, or Fetch API to send JSON data to your endpoint and verify it works correctly.

Body parsing middleware transforms the raw request body into a JavaScript object, making it easy to work with client-submitted data in your application.



REST API Principles

Stateless

Server doesn't store client data between requests. Each request contains all information needed.

Resource-based URLs

Use nouns for resources: /books, /users, /products

HTTP Methods

Use GET, POST, PUT, DELETE to represent CRUD operations

Status Codes

Use appropriate HTTP status codes to indicate response status

Following these principles creates consistent, maintainable APIs that are easy for developers to understand and use.

Status Codes & Error Handling

Common HTTP Status Codes

200	Success
201	Resource Created
400	Bad Request
404	Not Found
500	Server Error

Global Error Handler

```
app.use((err, req, res, next) => {
  console.error(err.stack);
  res.status(500).json({
    message: err.message,
    error: process.env.NODE_ENV ===
      'production' ? {} : err
  });
});
```

Always include this at the end of your middleware chain!

Proper error handling improves debugging and creates a better experience for API consumers by providing meaningful feedback.

Mini Task 2: Building a CRUD Books API

Let's create a complete RESTful API for managing books:

```
let books = [];

// GET all books
app.get("/books", (req, res) => {
  res.json(books);
});

// POST a new book
app.post("/books", (req, res) => {
  books.push(req.body);
  res.status(201).json(req.body);
});

// PUT (update) a book
app.put("/books/:id", (req, res) => {
  const id = parseInt(req.params.id);
  books[id] = req.body;
  res.json(req.body);
});

// DELETE a book
app.delete("/books/:id", (req, res) => {
  const id = parseInt(req.params.id);
  books.splice(id, 1);
  res.status(200).send("Book deleted");
});
```

In a production application, you would use a database instead of an array, add validation, and implement proper error handling for each route.