Introduction To Backend Development

Software Development Bootcamp

Understanding Servers, And Express

Topic

Introduction To Servers

What Is A Server?

A **server** is a computer program or device that provides functionality, data, or services to other programs or devices, called **clients**

Key Points About Servers

- Backend code runs on the server
- Servers can be physical machines or virtual instances
- They operate 24/7 to respond to client requests
- Servers can handle multiple client requests simultaneously

Server Roles In Web Development

- Host websites and web applications
- Process user requests and generate responses
- Manage database operations
- Handle authentication and authorization
- Perform complex computations
- Integrate with other services and APIs

APIs And Servers

While APIs and servers are closely related in web development, they serve different purposes

Relationship

- An API is often implemented on a server.
- A single server can host multiple APIs.
- When a client makes an API call, it's sent to a server, which processes the request and sends back a response according to the API specifications.

Example

- **Server**: A machine running Node.js and Express
- **API**: The routes and endpoints defined in your Express application that clients can interact with

Topic

Introduction To Express

What Is Express?

Express is a minimal and flexible Node.js web application framework that provides a robust set of features for web and mobile applications.

Key Features Of Express

- Simplifies server-side development with Node.js
- Provides a thin layer of fundamental web application features
- Allows for rapid development of robust APIs

Why Use Express?

- **Minimalist and Unopinionated:** Developers have the freedom to structure their applications as they see fit
- **Performance:** Lightweight and fast, optimized for building web applications
- **Extensive Middleware Ecosystem:** Easy to add functionality through middleware packages
- **Easy Database Integration:** Works well with most databases through appropriate drivers

Installing Express

Before we can use Express, we need to install it. Here's how to get started with ES modules

Prerequisites:

- Node.js version 14.6.0 or higher installed on your system
- **npm** (Node Package Manager) which comes with Node.js

What Is npm?

- npm stands for Node Package Manager
- It's the world's largest software registry
- Comes pre-installed with Node.js
- Allows developers to share and borrow packages

Key npm Features

- Manages dependencies for your project
- Can specify and lock down versions
- Runs scripts defined in package.json
- Provides command-line interface for package installation

`package.json`

- A manifest file for your project
- Lists project dependencies and their versions
- Defines scripts for running your application
- Can be created with `npm init`

Installing A Package Locally

```
npm install package-name
```

- Adds the package to your projects `node_modules` folder
- Updates `package.json` with the new dependency
- `npm uninstall package-name` Removes the package from `node_modules` and `package.json`

Installing Express

1. Create a new directory for your project
   a. `mkdir my-express-app`
   b. `cd my-express-app`
2. Initialize a new Node.js project
   a. `npm init -y`
3. Open `package.json` and add the "`type`":"`module`" field.

```
{
 "name": "my-express-app",
 "version": "1.0.0",
 "type": "module",
...
}
```

Installing Express Cont.

4. Install Express
    a. `npm install express` This adds Express as a dependency in your `package.json` file
5. Create a new file for your Express application (e.g. `app.js`)
6. Go back to your `package.json` file and find the key "`main`", and make sure the value matches the name of the file you just created.
    a. The "`main`" field in the `package. json` file defines the primary entry point for your Express application (e.g. `app.js`)

Setting Up Your Express Application

- `import express from 'express'`: Using ES6 module syntax to import the Express framework
- `const app = express()`: Creating an instance of an Express application. Assign this new application to the constant variable `app`
- `const port = 3000`: Declares a constant variable `port` and assigns it the value 3000. The port number is where our server will listen for incoming requests (**localhost:3000**)

app.js

```
import express from 'express'
const app = express()
const port = 3000
```

Topic

CRUD And HTTP Methods

HTTP Methods

HTTP methods, also known as HTTP verbs, indicate the desired action to be performed on the identified resource

- **GET:** Retrieve a resource (The Rick & Morty `fetch` was a **GET** request by default)
- **POST:** Submit data to be processed, creates a new resource
- **PUT:** Update and existing resource, replaces the entire resource
- **PATCH:** Partially modify an existing resource
- **DELETE:** Remove a resource

CRUD

CRUD stands for Create, Read, Update, Delete. These CRUD operations map directly to HTTP methods.

- Create → POST
- Read → GET
- Update → PUT / PATCH
- Delete → DELETE

How Express Handles HTTP Requests

- **Parsing**: Express automatically parses incoming requests. It extracts method, URL, headers, and body (with appropriate middleware).
- **Routing**: Express matches the request's method and URL to defined routes. Example: `app.get('/users', ...)` handles GET requests to **'/users'**
- **Middleware:** Functions that have access to the request and response objects. Can process data, add headers, etc. Example: `app.use(express.json())` for parsing JSON request bodies
- **Request Object**: Express provides a `req` object with useful properties and methods. Example: `req.body` contains the parsed body of the request.
- **Response Object**: Express provides a **res** object to send the response back to the client. Example: `res.json()` sends a JSON response. `res.status()` sets the HTTP status code.

Adding A Simple Request

- `app.get()`: Specifies that this route will handle GET requests.
- '/': This is the path for which this route is defined. The forward slash / represents the root path of your application
- **(req, res) => {...}:** This is the route handler function written as an arrow function. It takes two parameters
  - **req:** The request object containing information about the HTTP request
  - **res:** The response object, used to send back the HTTP response
- **res.send(**"Hello World"**):** This is the acton performed when this route is accessed.

app.js

```
import express from 'express'
const app = express()
const port = 3000

app.get('/', (req, res) => {
   res.send(`Hello World!`)
})
// Listens for connections to our port
app.listen(port, () => {
   console.log(`listening on port ${port}`)
})
```

Testing Our Endpoint

- In order to test our endpoint we need to add a script to run our code
- In the `package.json` inside the "`scripts`" field add the following "`dev": "node app.js`"
- Now in the terminal we can run the command `npm run dev`
- If we go to **localhost:3000/** in the browser we should see the text we sent in our response.

Additional Points To Consider

- This is a very basic example. In a real application, you might send HTML, or return JSON data instead of a simple string.
- You can define similar routes for other HTTP methods (POST, PUT, DELETE, etc.) and for different paths.
- The order in which you define routes in Express matters. More specific routes should generally come before more general ones

Route Parameters

Route parameters are named URL segments used to capture values specified at their position in the URL. These captured values are stored in the `req.params` object, with the name of the route parameter specified in the path as their respective keys.

Route Parameter Example

- Route parameters are defined by prefixing a colon (`:`) to the parameter name in the route path.
- Route parameters are available in the **`req.params`** object.
- The parameter names are the keys in this object.

```
app.get('/users/:userId', (req, res) => {
    res.send(req.params)
    // if the URL is /users/123 req.params
would be {userId: "123"}
})
```

Why Use Route Parameters?

- **Dynamic URLs:** Allow creation of flexible, resource-specific routes.
  - Example: **/users/:userId** can handle requests for any userID
- **Facilitate creation of clean, intuitive APIs**
  - Example: **/articles/:articleId** clearly identifies the resource being accessed
- **Enhanced Readability:** Make routes more descriptive and self-explanatory

Exercise

Hello Express