LECTURE 16

Introduction to Server-Side Programming

Taking the JavaScript you already know and using it to build the other half of a web app.

CSC336 Web Technologies

OVERVIEW

In This Lecture

1. Client-side vs. server-side responsibilities
2. Web server, application server, and runtime environment -- what each term actually means
3. Node.js as a JavaScript runtime, and how it stays fast with the event loop and non-blocking I/O
4. npm (Node Package Manager) and package.json
5. Setting up a new Express.js project from scratch
6. Building your first web server and a basic route

TWO SIDES

Client-Side Responsibilities

1. Rendering the page the user sees
2. Responding to clicks, typing, and other user interactions
3. Validating a form before sending it (nicer UX -- never trust this alone)
4. Making requests to a server for data, using fetch (Lecture 15)

TWO SIDES

Server-Side Responsibilities

1. Deciding what data to send back for a given request
2. Talking to a database to read or save information
3. Checking whether a user is allowed to see or change something (authentication and authorization)
4. Enforcing business rules the client should never be trusted to enforce alone

TWO SIDES

Client-Side Code Is Public

NOTE

Anyone can open their browser's developer tools and read, and even change, your client-side JavaScript. Any check that truly matters -- "is this password correct?", "does this user own this post?" -- must be done AGAIN on the server, which the user cannot tamper with.

TWO SIDES

Client-Side vs. Server-Side

	Client-side	Server-side
Runs on	The user's device (browser)	A computer you control
Seen/edited by user?	Yes	No
Typical languages	HTML, CSS, JavaScript	JavaScript (Node.js), Python, Java, PHP...
Typical job	Display, interactivity, UX	Data, business logic, security, storage

The Two Sides Talk Over HTTP

The same request-response pattern from Lecture 1 -- JavaScript on the client then uses the data to update the page.

THREE TERMS

Web Server vs. App Server vs. Runtime

WEB SERVER

- Listens for HTTP requests, sends back HTTP responses
- Just speaks HTTP -- receive a request, send a response
- Examples: Apache, Nginx, or a Node.js server you build

APPLICATION SERVER

- Runs your app's actual logic -- decides WHAT to send back
- In small Express projects, this is the SAME program as the web server
- Can be a separate piece of software in larger systems

RUNTIME ENVIRONMENT

- The software that actually executes your code
- A browser's runtime gives JS the DOM, fetch, localStorage
- Node.js's runtime gives JS the file system and network instead

Why Can't JavaScript Just Run Anywhere?

NOTE

JavaScript, as a language, only defines variables, functions, loops, objects. It does NOT define how to read a file or open a network connection -- those abilities are added by whatever environment runs the code. Node.js deliberately leaves out browser-only things like the DOM.

A JavaScript Runtime Outside the Browser

1. Node.js ("Node") lets you run JavaScript directly on a computer, outside any browser -- including on a server
2. Built in 2009 on top of Chrome's V8 engine -- the same engine that runs JS inside Chrome
3. Adds capabilities a browser withholds: reading/writing files, listening for network connections
4. The SAME language you already know can now write your server -- no new language required

I/O and Blocking

1. I/O (Input/Output): any operation where your program talks to something outside itself and waits -- reading a file, querying a database, a network request

2. These are typically thousands of times slower than running JavaScript in memory

3. A blocking operation freezes the ENTIRE program until it finishes -- like one cashier who stops serving everyone else to find one item

Node.js Uses Non-Blocking I/O Instead

1. When Node starts a slow operation, it does NOT wait around -- it hands off the task and moves on
2. When the slow task finishes, Node is notified and runs the code attached to handle the result
3. This lets one Node.js process serve many clients at once, even though it only does one thing at any single instant

Non-Blocking I/O in Action

The event loop constantly checks: is there a finished task waiting for its callback? If so, run it. If not, keep checking.

This Is Why Lecture 15 Mattered

TIP

Server-side Node.js code is full of non-blocking operations -- reading files, querying databases, calling other APIs -- and you will use async/await constantly to write that code cleanly.

Non-Blocking I/O Doesn't Help Everything

WARNING

It only helps with waiting on EXTERNAL things (disk, network, database). A huge, slow calculation directly in JavaScript WILL block the single thread and freeze your entire server for every user. Node.js is great at "wait for many slow things at once," not at "crunch huge amounts of pure computation."

The Node Package Manager

1. Almost no real project is written entirely from scratch -- developers rely on packages (libraries/modules)

2. npm is installed automatically alongside Node.js, and lets you download, install, and manage packages

3. npm also maintains the world's largest registry of JavaScript packages, at npmjs.com

Every Project Has a package.json

```
{
  "name": "my-first-server",
  "version": "1.0.0",
  "main": "index.js",
  "scripts": { "start": "node index.js" },
  "dependencies": { "express": "^4.19.2" }
}
```

Lists basic project info, your dependencies, and custom shortcut commands (scripts) like starting your server.

node_modules

1. Installing a package downloads its code into a folder called node_modules, and records it in package.json
2. node_modules is never committed to version control (excluded via .gitignore)
3. Anyone can recreate it exactly by running npm install with no arguments -- npm reads package.json and fetches every listed dependency

The Most Widely Used Node.js Web Framework

1. A framework handles common, repeated tasks for you: parsing requests, matching URLs to your code, sending responses

2. Lets you focus on your application's actual logic instead of rebuilding these basics every time

Express Is Not Your Only Option

NOTE

This course focuses on Express because it keeps you using JavaScript. The COMSATS course plan also permits Python frameworks like Django and FastAPI, which solve the exact same problems. Routing, request handling, middleware, and status codes apply equally whether you use Express, Django, or FastAPI.

Step 1 -- Confirm Node.js Is Installed

```
node --version
npm --version
```

If these print version numbers (e.g. v20.11.0 and 10.2.4), Node.js and npm are ready. Otherwise install Node.js from nodejs.org.

Step 2 -- Initialize a New Project

```
mkdir my-first-server
cd my-first-server
npm init -y
```

npm init creates a new package.json; -y accepts all default answers (you can edit it by hand afterward).

Step 3 -- Install Express

```
npm install express
```

Downloads Express into node_modules and adds it as a dependency in package.json.

index.js

```
const express = require('express');
const app = express();
const PORT = 3000;

app.get('/', (req, res) => {
  res.send('Hello, world! This is my first Express server.');
});

app.listen(PORT, () => {
  console.log(`Server is running at http://localhost:${PORT}`);
});
```

Unpacking Each Piece

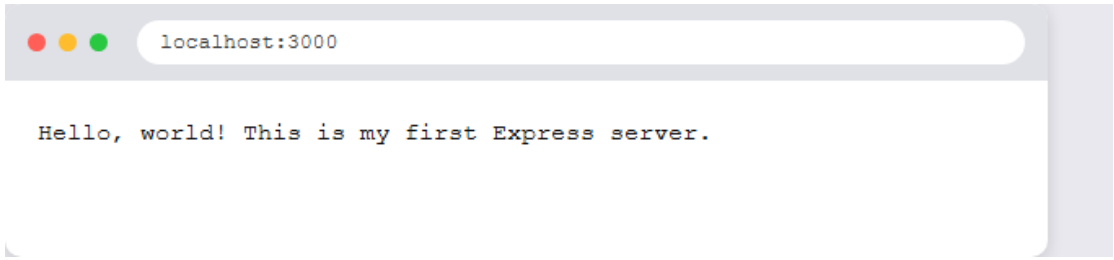
1. require('express') loads the package (Node's original module system; import express from 'express' is the modern equivalent)
2. express() creates an application object, usually named app -- represents your entire server
3. app.get('/', callback) defines a route: on GET /, run this function, with req (the request) and res (the response)
4. res.send(...) sends data back and ends the response -- plain text here, but also HTML, JSON, and more
5. app.listen(PORT, callback) actually starts listening for network connections -- nothing happens until you call it

Visiting localhost:3000

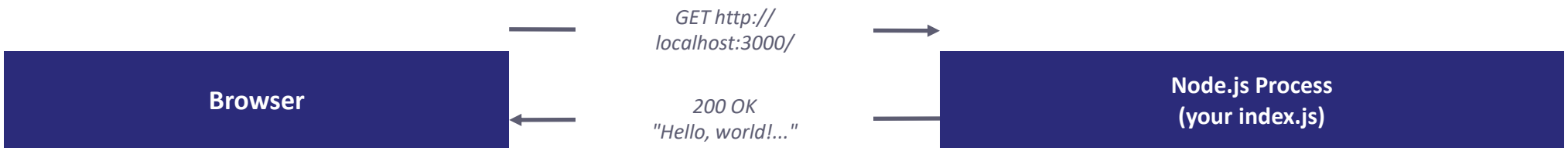
```
node index.js

// terminal prints:
// Server is running at
// http://localhost:3000

// then open a browser to
// http://localhost:3000
```



One Request, Start to Finish



app.listen(3000) means the server is already waiting; the route matches app.get('/', ...) and responds every time you visit.

What Is "localhost"?

TIP

localhost is a special hostname that always means "this same computer." While developing, your server and browser both run on your own machine. When you later deploy your app, visitors reach it through a real domain name instead.

EADDRINUSE: Port Already in Use

WARNING

Two programs cannot listen on the exact same port at the same time. If you see EADDRINUSE, something (perhaps an earlier copy of your own server) is already using that port -- stop it, or choose a different port number.

KEY TAKEAWAYS

Lecture 16 in Six Points

1. Client-side code runs in the browser and can be seen/altered by the user; server-side code runs on a machine you control and handles data, logic, and security.
2. A web server speaks HTTP, an application server runs your app's logic, and a runtime environment (browser or Node.js) executes your code -- in small Express apps, the first two are the same program.
3. Node.js is a JavaScript runtime (built on Chrome's V8 engine) that runs JavaScript outside the browser, including on servers.
4. Node.js handles many requests at once with non-blocking I/O and the event loop, instead of blocking the whole program while waiting.
5. npm installs and manages reusable packages, tracked in package.json, with code stored in node_modules.
6. Express.js simplifies building servers: express() creates an app, app.get() defines routes, and app.listen() starts listening.