Intro to Servers & Node.js

CS 390 – Web Application Development

J. Setpal

October 3, 2023

Outline

Outline

WIWYT – Servers

- Servers allow us to render dynamic content to webpages.

WIWYT – Servers

- Servers allow us to render dynamic content to webpages.
- Even static webpages need a server to function!

WIWYT – Node.js

- Node is a JavaScript runtime environment that does not use a browser to run.

WIWYT – Node.js

- Node is a JavaScript runtime environment that does not use a browser to run.
- This backend allows us to develop a server that scales effectively, has built-in concurrency, and does not require us to learn a new language.

Outline

What is a Server?

The internet effectively is a lot of **computers communicating over a network**.

What is a Server?

The internet effectively is a lot of **computers communicating over a network**. Certain computers host ‘services’ – black-box softwares that perform abstracted functionality – these computers are **servers**.

What is a Server?

The internet effectively is a lot of **computers communicating over a network**. Certain computers host 'services' – black-box softwares that perform abstracted functionality – these computers are **servers**.

The computers that access this abstracted functionality are **clients**.

What is a Server?

The internet effectively is a lot of **computers communicating over a network**. Certain computers host ‘services’ – black-box softwares that perform abstracted functionality – these computers are **servers**.

The computers that access this abstracted functionality are **clients**.

Servers can refer to both hardware (HPC clusters) and software (services). We will focus our discussion on the software aspect of servers – that forms the backend of your web application.

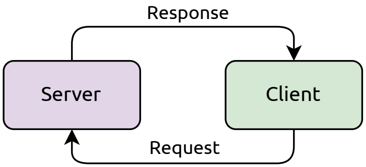
What is a Server?

The internet effectively is a lot of **computers communicating over a network**. Certain computers host 'services' – black-box softwares that perform abstracted functionality – these computers are **servers**.

The computers that access this abstracted functionality are **clients**.

Servers can refer to both hardware (HPC clusters) and software (services). We will focus our discussion on the software aspect of servers – that forms the backend of your web application.

Functionally, servers follow a straightforward lifecycle:



Server Types

There are three server protocols that are useful to know:

1. **HTTP**: Implements the standard client-server protocol for webpage rendering.
   a. Requests: GET, POST, OPTIONS, DELETE, TRACE methods with headers that transmit metadata.
   b. Repsonse: Codes (200, 404) for status and MIME for type.

Server Types

There are three server protocols that are useful to know:

1. **HTTP**: Implements the standard client-server protocol for webpage rendering.
   a. Requests: GET, POST, OPTIONS, DELETE, TRACE methods with headers that transmit metadata.
   b. Repsonse: Codes (200, 404) for status and MIME for type.
2. **WebSockets**: Protocol for real-time communication, similar to Java Sockets.

Server Types

There are three server protocols that are useful to know:

1. **HTTP**: Implements the standard client-server protocol for webpage rendering.
   a. Requests: GET, POST, OPTIONS, DELETE, TRACE methods with headers that transmit metadata.
   b. Repsonse: Codes (200, 404) for status and MIME for type.
2. **WebSockets**: Protocol for real-time communication, similar to Java Sockets.
3. **Proxy**: Routes an HTTP request to a specific applications. Can be used to host multiple services on a single sever endpoint.

The Serverless Paradigm

Serverless services – like AWS's Lambda – have replaced a lot of bepsoke servers.

The Serverless Paradigm

Serverless services – like AWS's Lambda – have replaced a lot of bepsoke servers. These work by specifying stand-alone *functions* that adopt a pre-specified I/O, and are cold-started when a request is created.

Q: Does this mean that they are inherently different from the original client-server paradigm?

The Serverless Paradigm

Serverless services – like AWS's Lambda – have replaced a lot of bepsoke servers. These work by specifying stand-alone *functions* that adopt a pre-specified I/O, and are cold-started when a request is created.

Q: Does this mean that they are inherently different from the original client-server paradigm?
A: No! Their only difference is that these are stateless in nature. A server still processes user input and returns a response.

The Serverless Paradigm

Serverless services – like AWS's Lambda – have replaced a lot of bepsoke servers. These work by specifying stand-alone *functions* that adopt a pre-specified I/O, and are cold-started when a request is created.

Q: Does this mean that they are inherently different from the original client-server paradigm?
A: No! Their only difference is that these are stateless in nature. A server still processes user input and returns a response.

These are cost-efficient for hosting applications that perform a highly specific operation on containerized input, are not called on continually, and don't have complicated initialization sequences.

The Serverless Paradigm

Serverless services – like AWS's Lambda – have replaced a lot of bepsoke servers. These work by specifying stand-alone *functions* that adopt a pre-specified I/O, and are cold-started when a request is created.

Q: Does this mean that they are inherently different from the original client-server paradigm?
A: No! Their only difference is that these are stateless in nature. A server still processes user input and returns a response.

These are cost-efficient for hosting applications that perform a highly specific operation on containerized input, are not called on continually, and don't have complicated initialization sequences.

Notably, it does not **idle** when unused.

Server v/s Client Side Rendering

A naive approach to render dynamic page content is as follows:

1. Return a template HTML file.
2. The client sends a request to a server for data.
3. The client then uses the data returned as a JSON to render the dynamic sections.
4. Render the HTML file to the browser.

Server v/s Client Side Rendering

A naive approach to render dynamic page content is as follows:

1. Return a template HTML file.
2. The client sends a request to a server for data.
3. The client then uses the data returned as a JSON to render the dynamic sections.
4. Render the HTML file to the browser.

This is **client-side rendering**.

Server v/s Client Side Rendering

A naive approach to render dynamic page content is as follows:

1. Return a template HTML file.
2. The client sends a request to a server for data.
3. The client then uses the data returned as a JSON to render the dynamic sections.
4. Render the HTML file to the browser.

This is **client-side rendering**. Alternatively:

1. Generate HTML infused with dynamic content in the server.
2. Return the complete HTML file.
3. Render the HTML file to the browser.

Server v/s Client Side Rendering

A naive approach to render dynamic page content is as follows:

1. Return a template HTML file.
2. The client sends a request to a server for data.
3. The client then uses the data returned as a JSON to render the dynamic sections.
4. Render the HTML file to the browser.

This is **client-side rendering**. Alternatively:

1. Generate HTML infused with dynamic content in the server.
2. Return the complete HTML file.
3. Render the HTML file to the browser.

This is **server-side rendering**.

Server v/s Client Side Rendering

A naive approach to render dynamic page content is as follows:

1. Return a template HTML file.
2. The client sends a request to a server for data.
3. The client then uses the data returned as a JSON to render the dynamic sections.
4. Render the HTML file to the browser.

This is **client-side rendering**. Alternatively:

1. Generate HTML infused with dynamic content in the server.
2. Return the complete HTML file.
3. Render the HTML file to the browser.

This is **server-side rendering**. It's much quicker, since it does not rely on multiple processes running synchronously.

Outline

What is Node?

From `https://nodejs.org/en/about`: [Node.js is] an **asynchronous event-driven JavaScript runtime**.

What is Node?

From `https://nodejs.org/en/about`: [Node.js is] an **asynchronous event-driven JavaScript runtime**. Let's break it down, in *reverse order*:

1. **JavaScript Runtime**: An environment within which JavaScript can be executed.

What is Node?

From `https://nodejs.org/en/about`: [Node.js is] an **asynchronous event-driven JavaScript runtime**. Let's break it down, in *reverse order*:

1. **JavaScript Runtime**: An environment within which JavaScript can be executed.
2. **Event-driven**: Node initializes an event-loop to evaluate user requests. Actions taken by the driver thread are triggered by events.

What is Node?

From `https://nodejs.org/en/about`: [Node.js is] an **asynchronous event-driven JavaScript runtime**. Let's break it down, in *reverse order*:

1. **JavaScript Runtime**: An environment within which JavaScript can be executed.
2. **Event-driven**: Node initializes an event-loop to evaluate user requests. Actions taken by the driver thread are triggered by events.
3. **Asynchronous**: Despite being single-threaded by default, Node uses asynchrony to handle multiple connections concurrently by offloading operations to the system kernel.

What is Node?

From `https://nodejs.org/en/about`: [Node.js is] an **asynchronous event-driven JavaScript runtime**. Let's break it down, in *reverse order*:

1. **JavaScript Runtime**: An environment within which JavaScript can be executed.
2. **Event-driven**: Node initializes an event-loop to evaluate user requests. Actions taken by the driver thread are triggered by events.
3. **Asynchronous**: Despite being single-threaded by default, Node uses asynchrony to handle multiple connections concurrently by offloading operations to the system kernel.

This makes Node scalable without requiring significant manual configuration when building low-latency applications.

What is Node?

From `https://nodejs.org/en/about`: [Node.js is] an **asynchronous event-driven JavaScript runtime**. Let's break it down, in *reverse order*:

1. **JavaScript Runtime**: An environment within which JavaScript can be executed.
2. **Event-driven**: Node initializes an event-loop to evaluate user requests. Actions taken by the driver thread are triggered by events.
3. **Asynchronous**: Despite being single-threaded by default, Node uses asynchrony to handle multiple connections concurrently by offloading operations to the system kernel.

This makes Node scalable without requiring significant manual configuration when building low-latency applications.

Why Node?

What is Node?

From `https://nodejs.org/en/about`: [Node.js is] an **asynchronous event-driven JavaScript runtime**. Let's break it down, in *reverse order*:

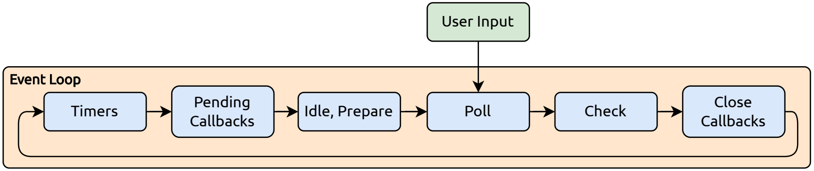
1. **JavaScript Runtime**: An environment within which JavaScript can be executed.
2. **Event-driven**: Node initializes an event-loop to evaluate user requests. Actions taken by the driver thread are triggered by events.
3. **Asynchronous**: Despite being single-threaded by default, Node uses asynchrony to handle multiple connections concurrently by offloading operations to the system kernel.

This makes Node scalable without requiring significant manual configuration when building low-latency applications.

Why Node? Node runs JavaScript, and using the same language for both the front and backend reduces developer overhead.

The Node Event Loop

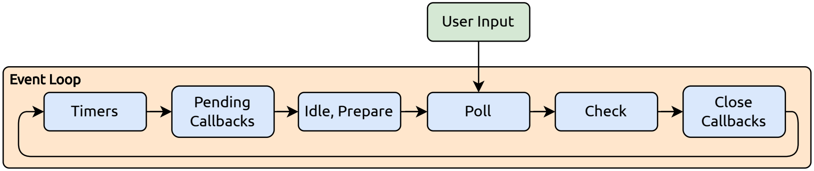
The node event loop allows Node to efficiently perform **non-blocking I/O operations**. It first processes the provided input script / REPL, and begins running the following event loop:[1]



[1]from node.js documentation

The Node Event Loop

The node event loop allows Node to efficiently perform **non-blocking I/O operations**. It first processes the provided input script / REPL, and begins running the following event loop:[1]

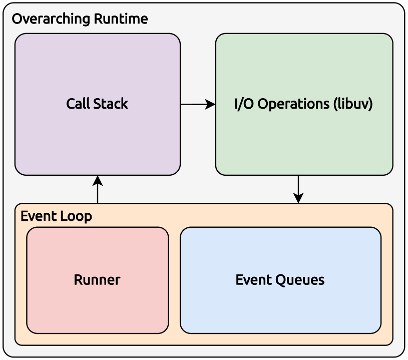


Each of the phases contains it's own callback queue that is either triggered or executed.

[1] from node.js documentation

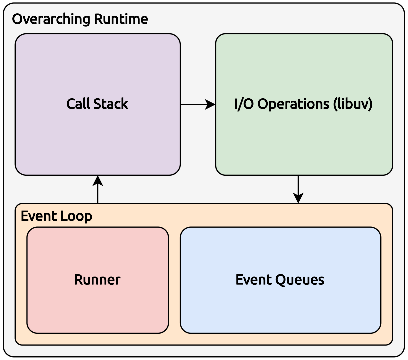
Blocking v/s Non-Blocking Processes

The event loop itself is a part of the overarching construct that sets up the Node runtime:



Blocking v/s Non-Blocking Processes

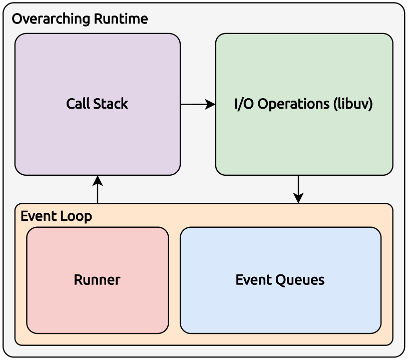
The event loop itself is a part of the overarching construct that sets up the Node runtime:



Within this context, the primary distinction made between tasks is if they are executed on the call stack. Such a process is called a **blocking process**.

Blocking v/s Non-Blocking Processes

The event loop itself is a part of the overarching construct that sets up the Node runtime:

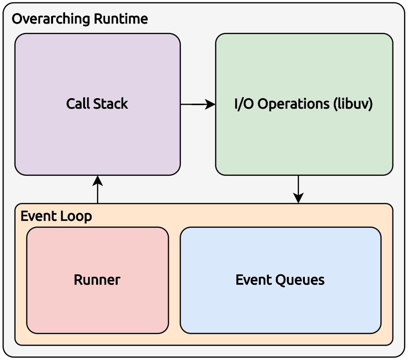


Within this context, the primary distinction made between tasks is if they are executed on the call stack. Such a process is called a **blocking process**.

Conversely, tasks that can be offloaded to an asynchronous handler (like libuv) is a **non-blocking process**.

Blocking v/s Non-Blocking Processes

The event loop itself is a part of the overarching construct that sets up the Node runtime:



Within this context, the primary distinction made between tasks is if they are executed on the call stack. Such a process is called a **blocking process**.

Conversely, tasks that can be offloaded to an asynchronous handler (like libuv) is a **non-blocking process**.

The objective is to maximize non-blocking asynchronous processes to minimize latency.

Variable Contexts

There a certain set of in-built functions within Node added on to the V8 runtime environment.

Variable Contexts

There a certain set of in-built functions within Node added on to the V8 runtime environment.

These can be accessed via `global`. It is a stand-in replacement for `window` in a browser runtime environment.

Variable Contexts

There a certain set of in-built functions within Node added on to the V8 runtime environment.

These can be accessed via `global`. It is a stand-in replacement for `window` in a browser runtime environment.

There is also a file-specific `module` object that can be accessed, however is not a part of the global context.

Node Module Structure

Every file in a node application is called a **module**.

Node Module Structure

Every file in a node application is called a **module**. A series of modules forms the package or the application that is being developed.

Node Module Structure

Every file in a node application is called a **module**. A series of modules forms the package or the application that is being developed.

Node packages built-in modules which enable us to break the V8 sandbox, to interface with the filesystem as well as parse and return HTTP queries.

Node Module Structure

Every file in a node application is called a **module**. A series of modules forms the package or the application that is being developed.

Node packages built-in modules which enable us to break the V8 sandbox, to interface with the filesystem as well as parse and return HTTP queries.

You can import modules using the `require` keyword. For example, to load the HTTP module, you can run: `const http = require('http')`.

Node Module Structure

Every file in a node application is called a **module**. A series of modules forms the package or the application that is being developed.

Node packages built-in modules which enable us to break the V8 sandbox, to interface with the filesystem as well as parse and return HTTP queries.

You can import modules using the `require` keyword. For example, to load the HTTP module, you can run: `const http = require('http')`.

You can export local functions in different modules by modifying the `exports` object.

Node Module Structure

Every file in a node application is called a **module**. A series of modules forms the package or the application that is being developed.

Node packages built-in modules which enable us to break the V8 sandbox, to interface with the filesystem as well as parse and return HTTP queries.

You can import modules using the `require` keyword. For example, to load the HTTP module, you can run: `const http = require('http')`.

You can export local functions in different modules by modifying the `exports` object.

Node has the **world's largest module repository**, that can be accessed using the Node Package Manager (`npm`).

Node Module Structure

Every file in a node application is called a **module**. A series of modules forms the package or the application that is being developed.

Node packages built-in modules which enable us to break the V8 sandbox, to interface with the filesystem as well as parse and return HTTP queries.

You can import modules using the `require` keyword. For example, to load the HTTP module, you can run: `const http = require('http')`.

You can export local functions in different modules by modifying the `exports` object.

Node has the **world's largest module repository**, that can be accessed using the Node Package Manager (`npm`). Notably, we will use the **Express.js** module to abstract the process of building server-side APIs.

Building a Simple HTTP Server with Node

If you can view this screen, I am making a mistake.

Outline

Thank you!

Have an awesome rest of your day!

Slides: **`https://cs.purdue.edu/homes/jsetpal/slides/intro-servers-node.pdf`**

If anything's incorrect or unclear, please ping `jsetpal@purdue.edu`
I'll patch it ASAP.