

# NodeJS

- NodeJS
  - Asynchronous event driven JavaScript runtime
  - Designed for scalable network applications
  - Can be used to developed full web applications
  - Reference: <https://nodejs.org/en/about/>
- Relies on JavaScript V8 Engine
  - Written in C++
  - V8 incorporates just-in-time(JIT) compiler
    - Compiles JavaScript to machine code rather than interpreting it
- Why use it? Performance reasons

# Installation

- From [nodejs.org](https://nodejs.org)
- To start type on command line “node”
- To exit “.exit”
- For help
  - node --help
- Version
  - node -v
- To check syntax
  - node -c filename.js

# WebServer Example

- **Example:** webserver.js
  - Module (library)
  - **require** statement imports the module
  - http module is one of Node's core modules
- About modules
  - To add modules type **npm install <module name>** at the command line
  - **Example:** Installing Connect Middleware (CM) module
    - c:\tempExample>npm install cm

# Asynchronous Programming

- **fs** module – File System module
- Node supports both synchronous and asynchronous versions of most File System Functions
- **Synchronous Programming**
  - Code that performs one task after another waiting for one to complete before starting another
  - **Example:** readFileContentSync.js
- **Asynchronous Programming**
  - We don't wait for code to finish
  - Callback will take care of processing once an event has been triggered
  - **Example:** readFileContentAsync.js
- You cannot use try ... catch with asynchronous functions

# Global Objects

- **global** – similar to the browser global object
  - Provides access to all globally available Node objects and functions
  - **Example:** `console.log(global)`
- **Process**
  - Provides information about the Node environment and the runtime environment
  - Standard input/output occurs through **process**
  - **process.stdin** – stream for stdin
  - **process.stdout** – stream for stdout
  - **process.stderr** – stream for stderr
  - **Example:** `webServerControl.js`
  - **Example:** `imageServer.js`

# Event Queue

- How to enable asynchronous functionality?
  - **Alternative #1:** Assigning thread to each time-consuming task
    - Disadvantage – threads are expensive
  - **Alternative #2:** Event-driven model
    - Application does not wait for time-consuming task to finish; the task notifies the application when it has finished by generating an event
- Node and JavaScript (browser) rely on an event-driven model
- **JavaScript event loop** – facilitates user interface event handling
- **Node event loop** – facilitates server-based functionality, mainly I/O
  - Events associated with opening a file, reading contents into buffer, etc.
- Event Loop Video: <https://www.youtube.com/watch?v=8aGhZQkoFbQ> by Philip Roberts.
- Animation Tool: <https://goo.gl/iJRGvT>

# Processing in JavaScript

- Processing in JavaScript is supported by the following main components:
  - **Call Stack** – Runtime system (e.g., V8 engine) relies on it for code execution
  - **Callback Queue** – Processing of events (e.g., clicking on button, an Ajax request) are associated with functions (callbacks) that needs to be executed once the event takes place. This queue keeps track of those callbacks
  - **Apis** - Browsers and Node provide a set of Apis (e.g., an Ajax API) that will take care of adding a callback to the Callback Queue once an event has been identified
  - **Event loop** – Will periodically check the Call Stack and once it is empty it will move one callback from the Callback Queue to the Call Stack
- Notice that event though we say that JavaScript can only execute one thing at a time, there are other entities (Apis) that are executing as the runtime engine is executing code.

# About setTimeout

- When setTimeout is called with a value of 0, it does not mean code will be executed immediately. It means the Api will place the callback immediately in the Callback Queue, but not that will be placed in the Call Stack.



# Scheduling an Asynchronous Task

- **Example:** FibonacciAsync.js
- `process.nextTick()`
  - Callback function called once the current queue is empty, but before any new events are added

# Events

- For timers
  - `setTimeout()` – executes callback after delay time (milliseconds)
  - `setInterval()` – callback is executed at period intervals
  - `clearInterval()` – clears timer
  - **Example:** `timer.js`
- EventEmitter
  - Enables asynchronous event handling
  - `EventEmitter.emit()` generates event
  - The `EventEmitter.on()` event handler called when a specific event is generated

# References

- Learning Node, 2nd Edition
  - By: Shelley Powers
  - SBN-13: 978-1-4919-4312-0