1 Build a Back-End with Node/Express.js

1.1 Introduction

1.2 Node REPL

- Is an abbrebivation for Read-eval-print loop

- Node comes with built-in javascript REPL

- `.editor` goes into editor mode

  - Use `CTRL + D` when ready to evaluate the input

- A REPL can be extremely useful for performing calculations

- The Node environment contains a number of Node-specific `global` elements in addition to those built into the JavaScript language

  - can be examined using command `console.log(global)`

1.3 Running a Program with Node

- Done using command `node myProgram.js`

- Javascript code is written to file `.js` extension

1.4 Accessing the Process Object

- Node has a global `process` object with useful methods and information about the current process.

  - `process.env` property is an object which stores and controls information about the environment in which the process is currently running

    * `PWD` - holds a string with the directory where the current process is located
    * `NODE_ENV` - holds a value of either production or development

      Example

      ```
      1    if (process.env.NODE_ENV === 'development'){
      2        console.log('Testing! Testing! Does everything work?');
      3    }
      4
      ```

    * `process.memoryUsage()` returns information on the CPU demands of the current process.
    * `process.memoryUsage().heapUsed` return a number representing how many bytes of memory the current process is using.

– `process.argv` property holds an array of command line values provided when the current process was initiated

  ∗ first element in the array is the absolute path to Node
  ∗ second element in the array is the path to the file that's running
  ∗ following elements will be any command line arguments provided when the process was initiated (like C)!!!

```
1    node myProgram.js testing several features
2
```

```
1    console.log(process.argv[3]); // Prints 'several'
2
```

1.5   Core Modules and Local Modules

- **Modularity** is a software design technique where one program has distinct parts each providing a single piece of the overall functionality.

  – Is essential when creating scalable programs
    ∗ incorporate libraries and frameworks and separate the program's concerns into manageable chunks

- **Modules** come together to build a cohesive whole

  – is imported using `require()`
```
1    // Require in the 'events' core module:
2    let events = require('events');
3
```

  – is exported using `module.exports`
```
1    module.exports = class Dog {
2
3        constructor(name) {
4            this.name = name;
5        }
6
7        praise() {
8            return `Good dog, ${this.name}!`;
9        }
10   };
11
```

1.6   Node Package Manager

- NPM, which stands for Node Package Manager

1.7   Event-Driven Architecture

- Node is often described as having event-driven architecture

- Node provides an `EventEmitter` class which we can access by requiring in the events core module

```
1     // Require in the 'events' core module
2     let events = require('events');
3
4     // Create an instance of the EventEmitter class
5     let myEmitter = new events.EventEmitter();
6
```

1.8   Event-Driven Architecture

- Node is often described as having event-driven architecture.

  - This feels so much like threaded programming

- event emitter instance has an `.on()` method which assigns a listener callback function to a named event.

  - first argument - the name of the event as a string
  - second argument - the listener callback function

```
1
2     let newUserListener = (data) => {
3         console.log('We have a new user: ${data}.');
4     };
5
6     // Assign the newUserListener function as the listener callback for '
      new user' events
7     myEmitter.on('new user', newUserListener)
8
9     // Emit a 'new user' event
10    myEmitter.emit('new user', 'Lily Pad') //newUserListener will be
      invoked with 'Lily Pad'
11
```

1.9   Asynchronous JavaScript with Node.js

- Node was designed to use an event loop like the one used in browser-based JavaScript execution

- The event-loop enables asynchronous actions to be handled in a non-blocking way.

  - APIs trigger the subscription to and emitting of events to signal the completion of the operation

Example

```
1    let keepGoing = true;
2
3    let callback = () => {
4        keepGoing = false;
5    };
6
7    setTimeout(callback, 1000); // Run callback after 1000ms
8
9    while(keepGoing === true) {
10        console.log('This is the song that never ends. Yes, it just goes
    on and on my friends. Some people started singing it, not knowing what
    it was, and they'll continue singing it forever just because...')
11    };
12
```
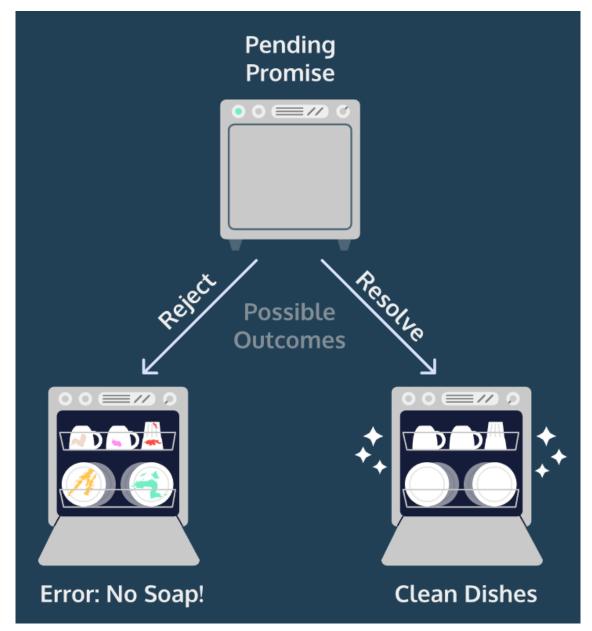
– The while loop will continue forever

– Why? because no signal has been sent

– To resolve this issue, replace `setTimeout` with `setTimeInterval`

- `Promise, async ... await`

    – modern way of handling asynchronous tasks

1.10 Asynchronous Javascript - Introduction

- **asynchronous operation** is one that allows the computer to "move on" to other tasks while waiting for the asynchronous operation to complete.

1.11   Asynchronous Javascript - What is a Promise?

- **Pending:** The initial state— the operation has not completed yet.

- **Fulfilled:** The initial state— the operation has not completed yet.

- **Rejected:** The initial state— the operation has not completed yet.

1.12   Constructing a Promise Object

- use the `new` keyword and the `Promise` constructor method

```
const executorFunction = (resolve, reject) => { };
const myFirstPromise = new Promise(executorFunction);

```

- `Promise` constructor method takes a function parameter called the `executor` function which runs automatically when the constructor is called

- The `executor` function has has two function parameters

  - `resolve()`
    * is a function with one arguement
    * invoke causes the promise's status to change from `pending` to `fulfilled`
    * sets promises' resolved value to be the arguement passed to `resolve`
  - `reject()`
    * takes a reason or error as an argument
    * invoke causes `reject()` to change the promise's status from `pending` to `rejected`

Example

```
1    const executorFunction = (resolve, reject) => {
2        if (someCondition) {
3            resolve('I resolved!');
4        } else {
5            reject('I rejected!');
6        }
7    }
8    const myFirstPromise = new Promise(executorFunction);
9
```

1.13   The Node setTimeout() Function

-