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)!!!

```
node myProgram.js testing several features

```

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

```

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()`

```
// Require in the 'events' core module:
let events = require('events');

```

  - is exported using `module.exports`

```
module.exports = class Dog {

    constructor(name) {
        this.name = name;
    }

    praise() {
        return `Good dog, ${this.name}!`;
    }
};

```

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

```
// Require in the 'events' core module
let events = require('events');

// Create an instance of the EventEmitter class
let myEmitter = new events.EventEmitter();

```

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

```

let newUserListener = (data) => {
    console.log(`We have a new user: ${data}.`);
};

// Assign the newUserListener function as the listener callback for 'new user' events
myEmitter.on('new user', newUserListener)

// Emit a 'new user' event
myEmitter.emit('new user', 'Lily Pad') //newUserListener will be invoked with 'Lily Pad'

```

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

```
    let keepGoing = true;

    let callback = () => {
        keepGoing = false;
    };

    setTimeout(callback, 1000); // Run callback after 1000ms

    while(keepGoing === true) {
        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...')
    };

```

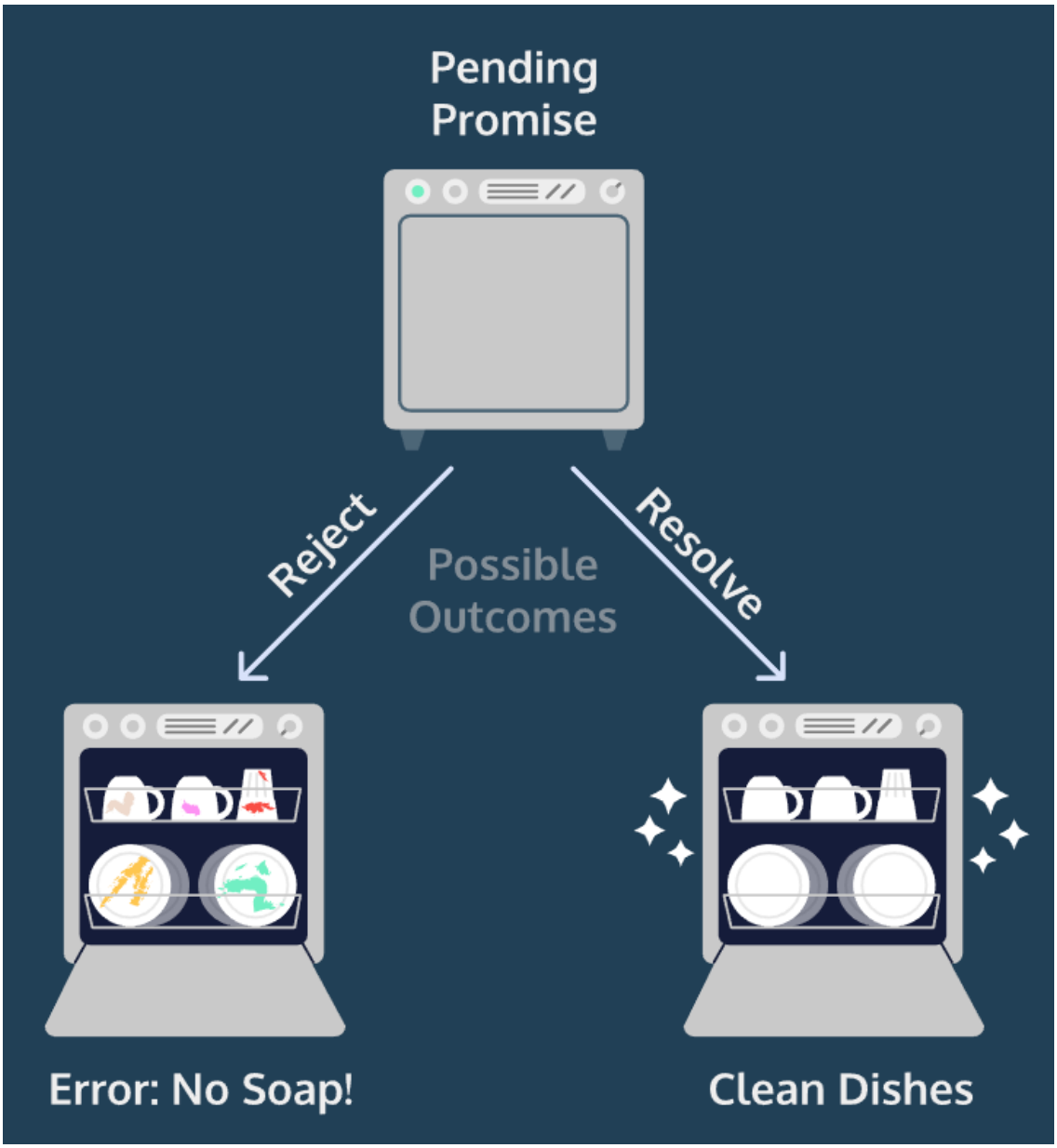
  - The while loop will continue forever
  - Why? because no signal has been sent
  - To resolve this issue, replace `setTimeout` with `setTimeInterval`

- `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

```
1    const executorFunction = (resolve, reject) => { };
2    const myFirstPromise = new Promise(executorFunction);
3
```

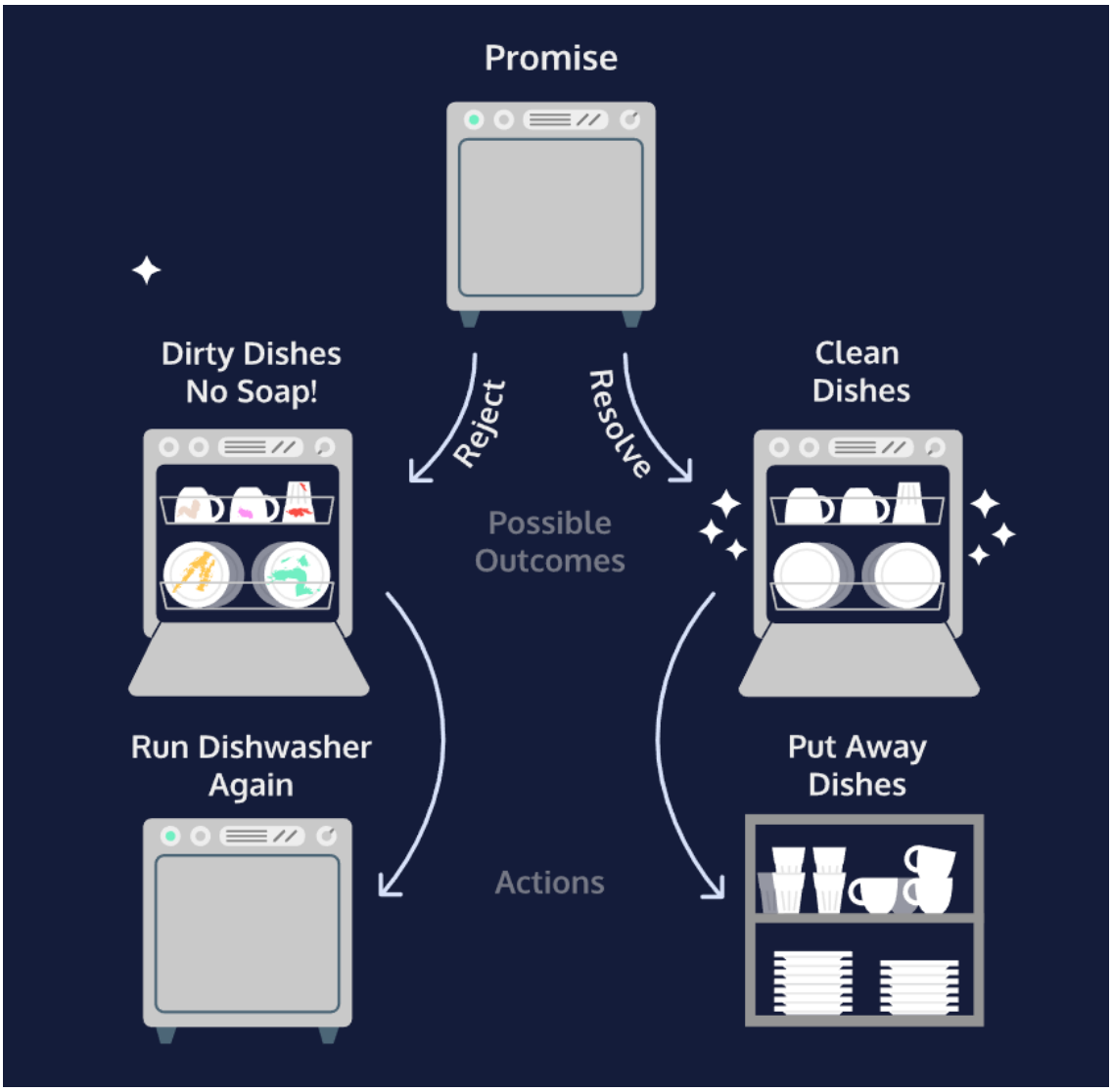
- `Promise` constructor method takes a function parameter called the `executor` function which runs automatically when the constructor is called
- The `executor` function has has two function parameters
  - `resolve()`
    - is a function with one arguement
    - invoke causes the promise's status to 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

```
const executorFunction = (resolve, reject) => {
    if (someCondition) {
        resolve('I resolved!');
    } else {
        reject('I rejected!');
    }
}
const myFirstPromise = new Promise(executorFunction);

```

1.13 The Node setTimeout() Function



- Rather than constructing promises, you'll be handling `Promise` objects returned to you as the result of an asynchronous operation
  - It will start off pending but settle eventually.
- `setTimeout()`
  - Is a node API
  - Has two parameters
    - a callback function
    - a delay in milliseconds.

```
        const executorFunction = (resolve, reject) => {
            if (someCondition) {
                resolve('I resolved!');
```

```
            } else {
                reject('I rejected!');
            }
        }
        const myFirstPromise = new Promise(executorFunction);

```

- the embedded code will exectue after said time (not exactly at the time)
  - because the function within is added to a line of code waiting to be run

1.14 Consuming Promises

- `.then()` is that it always returns a promise. We'll return to this in more detail in a later exercise and explore why it's so important.
- `.then()` takes two callback functions as arguements
  - First argument - is the success handler
  - Second argument - is the failure handler

1.15 The onFulfilled and onRejected Functions

1.16 Using catch() with Promises

- `separation of concerns` - one way to write cleaner code
- Javascript doesn't mind whitespace
- `.catch()` function takes only one argument, onRejected

Example

```
prom
.then((resolvedValue) => {
    console.log(resolvedValue);
})
.catch((rejectionReason) => {
    console.log(rejectionReason);
});

```

1.17 Chaining Multiple Promises

- `composition` - the process of chaining promises together
- Promise is designed with composition in mind

Example

```
    firstPromiseFunction()
    .then((firstResolveVal) => {
        return secondPromiseFunction(firstResolveVal);
    })
    .then((secondResolveVal) => {
        console.log(secondResolveVal);
    });

```

- We invoke a function `firstPromiseFunction()` which returns a promise.
- `.then()` is invoked with an anonymous function as the success handler
- Inside the success handler we return a new promise— the result of invoking a second function, `secondPromiseFunction()` with the first promise's resolved value.
- second `.then()` is invoked to handle the logic for the second promise settling.
- Inside second `.then()`, we have a success handler which will log the second promise's resolved value to the console.

```
const {checkInventory, processPayment, shipOrder} = require('./library.js');

const order = {
    items: [['sunglasses', 1], ['bags', 2]],
    giftcardBalance: 79.82
};

checkInventory(order)
.then((resolvedValueArray) => {
    // Write the correct return statement here:
    return processPayment(resolvedValueArray);
})
.then((resolvedValueArray) => {
    // Write the correct return statement here:
    return shipOrder(resolvedValueArray);
})
.then((successMessage) => {
    console.log(successMessage);
})
.catch((errorMessage) => {
    console.log(errorMessage);
});


```

1.18 Avoiding Common Mistakes

- **Mistake 1:** Nesting promises instead of chaining them.
  - Works fine

  - Imagine if we are handling five or then promises

```
    returnsFirstPromise()
    .then((firstResolveVal) => {
        return returnsSecondValue(firstResolveVal)
        .then((secondResolveVal) => {
            console.log(secondResolveVal);
        })
    })

```

- **Mistake 2:** Forgetting to return a promise.

  -

```
    returnsFirstPromise()
    .then((firstResolveVal) => {
        returnsSecondValue(firstResolveVal)
    })
    .then((someVal) => {
        console.log(someVal);
    })

```

  - `returnsFirstPromise()` which returns a promise.
  - invoke a second `.then()`. Since we didn't return, this `.then()` is invoked on a promise with the same settled value as the original promise

```
 const {checkInventory, processPayment, shipOrder} = require('./library.js
');

 const order = {
     items: [['sunglasses', 1], ['bags', 2]],
     giftcardBalance: 79.82
 };

 // Refactor the code below:

 checkInventory(order)
     .then((resolvedValueArray) => {
         return processPayment(resolvedValueArray);
     })
     .then((resolvedValueArray) => {
         return shipOrder(resolvedValueArray);
     })
     .then((successMessage) => {
         console.log(successMessage);
     });

```

1.19 Using Promise.all()

- promise composition is a great way to handle situations where asynchronous operations depend on each other or execution order matters
- What if we don't care about order? - make simple using `Promise.all()`
- `Promise.all()` accepts an array of promises as its argument and returns a single promise. That single promise will settle in one of two ways:
  - If every promise in the argument array resolves, the single promise returned from Promise.all() will resolve with an array containing the resolve value from each promise in the argument array.
  - If any promise from the argument array rejects, the single promise returned from Promise.all() will immediately reject with the reason that promise rejected.
    - This behavior is sometimes referred to as **failing fast.**

Example

```
1    let myPromises = Promise.all([returnsPromOne(), returnsPromTwo(),
    returnsPromThree()]);
2
3    myPromises
4        .then((arrayOfValues) => {
5        console.log(arrayOfValues);
6        })
7        .catch((rejectionReason) => {
8        console.log(rejectionReason);
9        });
10
```

  - `myPromises` assigned to invoking `Promise.all()`.
  - `Promise.all()` has an array of three promises— the returned values from functions.
  - `.then()` with a success handler which will print the array of resolved values if each promise resolves successfully.
  - `.catch()` with a failure handler which will print the first rejection message if any promise rejects.