Express

REVIEW: Lesson 3 Review — 00:10

Author: Shaun McKinnon

Promises — 00:20

https://medium.com/dev-bits/writing-neat-asynchronous-node-js-code-with-promises-32ed3a4fd098

Let's change our original insult code so that it uses promises:

```
// Promises
const request = require ( 'request' )

function getInsult ( url ) {
  let options = {
    url: 'https://insult.mattbas.org/api/insult',
    headers: { 'User-Agent': 'request' }
  }

  // 1
  return new Promise( ( resolve, reject ) => {
    // 2
    request.get( options, ( err, resp, body ) => {
      if ( err ) {
        // 3
        reject( err )
      } else {
        // 4
        resolve( body )
      }
    })
  })
}


getInsult( url ).then( ( result ) => {
  console.log( result )
})
```

So now you're probably saying "Great!" so we still have 3 nested levels of callbacks. What exactly did this solve? Well, one, we converted all our functions to arrow functions. Whoop. We now have responses based on if our promise is rejected or resolved. Ok...Promising (see what I did there?). We also now have the ability to chain events.

```
getInsult( url )
.then( ( result ) => {
  console.log( result )
})
.then(() => console.log('This is next'))
.then(() => console.log('And then this one'))
```

And we can sequence promises:

```
let insultOne = getInsult()
let insultTwo = getInsult()
let insultThree = getInsult()

Promise.all( [insultOne, insultTwo, insultThree] ).then( ( result ) => console.log( result ) )
Promise.all( [insultTwo, insultOne, insultThree] ).then( ( result ) => console.log( result ) )
Promise.all( [insultThree, insultTwo, insultOne] ).then( ( result ) => console.log( result ) )
```

The **Promise.all()** function takes a list of promises, in any given order, and returns a new promise once they are completed.

It is important to understand that resolved or rejected promise is immutable. The value will never change.

Author: Shaun McKinnon

Node: A little deeper — 00:30

The File Server

Node allows us to read files:

```
const fs = require( 'fs' )

fs.readFile( 'index.html', ( err, data ) => {
  if ( !err ) console.log( data )
})
```

The above will read the file, and log the data if present. It is important to note, that if no encoding is defined (the **utf-8** arg), it will return a raw buffer.

Using that information, we can read a file and send its contents to our client in our response instead of just raw text. We'll have to make some changes though (and we might as well use Promises):

```
const connect = require( 'connect' );
const app = connect();
const fs = require( 'fs' )

function getFile ( path ) {
  return new Promise( ( resolve, reject ) => {
    fs.readFile( path, ( err, data ) => {
      resolve( data )
    })
  })
}

let home = ( req, res ) => {
  getFile( 'index.html' )
  .then( ( data ) => {
    res.writeHead( 200, {'Content-Type': 'text/html'})
    res.write( data )
    res.end();
  })
};

app.use( '/', home );

app.listen( 3001 );

console.log( 'Server running at http://localhost:3001' );
```

There you have it: a way to send a file to the browser. We have some gotchyas though; if we link to a CSS file, it won't be handled correctly. That is because we are only writing a header for text/html and not really caring about any other requests that may come through. We should fix this:

```
const connect = require( 'connect' );
const app = connect();
const fs = require( 'fs' )
const url = require( 'url' )

function getFile ( path ) {
  return new Promise( ( resolve, reject ) => {
    fs.readFile( path, 'utf-8', ( err, data ) => {
      resolve( data )
    })
  })
}

const mime = {
  html: 'text/html',
  css: 'text/css',
  js: 'application/javascript',
  jpg: 'image/jpeg',
  png: 'image/png'
}

let home = ( req, res ) => {
  let path = (url.parse(req.url)).pathname
  path = '.' + ( ( path != '/' ) ? path : '/index.html' )
  let ext = path.split('.').pop()
  let type = mime[ext] || 'text/plain'

  getFile( path )
  .then( ( data ) => {
    res.writeHead( 200, {'Content-Type': type})
    res.write( data )
    res.end();
  })
};

app.use( '/', home );

app.listen( 3001 );

console.log( 'Server running at http://localhost:3001' );
```

Here we're adding a lookup object so we can find the mime type. The line **let ext = path.split('.').pop()** is splitting **path** at the **.** period into an array and then popping off the last element. The last element (always hopefully) should be the file extension. Then we give the extension as a key and grab the mime type, or just return 'text/plain'.

Our function hasn't changed, but we've abstracted the arguments we're providing. This will allow us to attach images, html, css, and javascript to our HTML page if we choose to.

Author: Shaun McKinnon

BREAK — 00:15

Introducing Express — 00:20

What is Middleware?

- Middleware glues together complex modular systems
- This could be
  - A messaging system
  - An event system
  - Task/Job scheduler

        - A router
        - An ORM or ODM
    - Middleware can also be considered as a communication or interpreter system that allows output from one system to be work as acceptable input on another system

Why is Express Considered Middleware?

- Our user makes a request to our server which is listening
- Express intercepts the request and evaluates the request packet the user is sending
    - For example, Express evaluates the request path
    - From that, Express determines which registered middleware to execute
    - The interesting thing here is that the middleware is calling more middleware where we can perform operations based on the request packet
- The first intercept is known as routing

A Quick 2 Minutes on Express

- Express is a Node.js web server project
    - As stated earlier, the community is driving the web application side of Node.js
    - Express is one of those projects
- Express is a bit of sugar on top of Node.js' createServer
- Declarative routing
    - Routing is the process of analyzing, modifying, and delivering to the appropriate end-point

Building an Express Web Application

- The **Express framework** is a small set of common web application features, kept to a minimum in order to maintain the **Node.js** style.
- It is built on top of **Connect** and makes use of its middleware architecture.
- Its features extend **Connect** to allow a variety of common web applications' use cases, such as the inclusion of **modular HTML template engines**, extending the response object to support various data format outputs, a routing system, and much more.
- In order for us to use Express, we should initialize our package dependency file and install the Express module
    - Navigate to the **/COMP_2068/Activities/Activity_04/** folder
    - To do so, use your command-line tool, and navigate to your working folder. Then execute the following command:

      $ npm init
      $ npm install --save express

- After creating your **package.json** file and installing express as one of your dependencies, you can now create your first **Express** application by adding your already familiar **server.js** file with the following lines of code:

```
// include the express module
const express = require( 'express' )

// create a new app
const app = express()

// define a route and register a middleware to happen
app.get( '/', ( req, res ) => res.send( 'Hello World' ) )

// start the server, listening on port 3001, and throw a callback executing console.log
app.listen( 3001, () => console.log( 'Listening on 3001' ) )
```

The Application, Request, and Response Objects

- **Express** presents three major objects that you'll frequently use.
    - The **application object** is the instance of an Express application you created in the first example and is usually used to configure your application.
    - The **request object** is a wrapper of Node's HTTP request object and is used to extract information about the currently handled HTTP request.
    - The **response object** is a wrapper of Node's HTTP response object and is used to set the response data and headers.
- The **application object** contains the following methods to help you configure your application:
    - **app.set(name, value)**: This is used to set environment variables that Express will use in its configuration.
    - **app.get(name):** This is used to get environment variables that Express is using in its configuration.

- **app.engine(ext, callback):** This is used to define a given template engine to render certain file types, for example, you can tell the **EJS template engine** to use HTML files as templates like this: **app.engine('html', require('ejs').renderFile).**
- **app.locals:** This is used to send application-level variables to all rendered templates.
- **app.use([path], callback):** This is used to create an Express middleware to handle HTTP requests sent to the server. Optionally, you'll be able to mount middleware to respond to certain paths.
- **app.VERB(path, [callback...], callback):** This is used to define one or more middleware functions to respond to HTTP requests made to a certain path in conjunction with the HTTP verb declared. For instance, when you want to respond to requests that are using the GET verb, then you can just assign the middleware using the app.get() method. For POST requests you'll use **app.post()**, and so on.
- **app.route(path).VERB([callback...], callback):** This is used to define one or more middleware functions to respond to HTTP requests made to a certain unified path in conjunction with multiple HTTP verbs. For instance, when you want to respond to requests that are using the GET and POST verbs, you can just assign the appropriate middleware functions using **app.route(path).get(callback).post(callback)**.
- **app.param([name], callback):** This is used to attach a certain functionality to any request made to a path that includes a certain routing parameter. For instance, you can map logic to any request that includes the userId parameter using **app.param('userId', callback)**.

Author: Shaun McKinnon

The Express Generator: Our First App — 00:30

Scaffolding

- Scaffolding is the process of stubbing out predefined template blocks
- The Express Generator will stub out our directories as well as some basic configurations and a basic app for us to test with
- The default template engine that the Express Generator uses is **pug** but we're going to use **EJS**
  - A template engine allows us to quickly write HTML mixed with Javascript code
  - The engine will parse this file and create the necessary constructs required to separate the HTML and the Javascript from each other
  - Template engines SHOULD
    - Make development quicker
    - Be fairly simple to learn
    - Follow common structure patterns

Generating Our First Application

1. Create a new folder called **Activity_05** under **/COMP_2068/Activities/**
2. Navigate to that folder in your command line
3. Run
   $ npm install express-generator -g
   - This will globally install the Express Generator, an app scaffolder
4. We have several options that we can use when we generate an app
5. Let's run the following for now
   $ express --css=sass --view=ejs our_first_app
   $ cd our_first_app
   $ npm install
6. The above will
   - Create our application
   - Move into our apps directory
   - Install our required packages
7. Let's open the new app in our IDE

Understanding the Directory Structure

- our_first_app
  - app.js
    - Our entire application configuration is handled in this file
    - It registers all the middleware we're going to require for our app to work properly
      - Our preprocessors for the SASS and EJS files

        - A Cookie parser
        - A default logger
        - A globally defined path and views access
        - Some resource routes
        - And some error handling
    - As our app evolves and we identify new resources, we will need to define some resource routes in here. You can see that the application has defined some default ones for us (lines 29 and 30):

```
app.use('/', indexRouter);
app.use('/users', usersRouter);
```

    - If you follow what is happening with these lines, you will notice they are pointing at some predefined variables
        - These are set on lines 8 and 9 and are importing 2 files under the routes directory
        - If you open one of these files (routes/index.js), you will notice they have created a custom module and are exporting it
        - The module is simply a middleware that they've created that execute when that route is accessed
- bin
    - www
        - You will notice that line 7 of this file requires our app.js file from the root directory of our application folder
            - This means that all of our registered middleware is now executing from this point
        - This file is our server entry point. If you look at the package.json file, you can see this stated in the start script
        - This file also conveniently sets up some common pieces such as
            - Our port
            - The server
            - The listener
            - and Error handling
- node_modules
    - This folder will contain all of our installed node modules that we defined in the package.json file
- public
    - This directory is our publicly accessible point for our users
    - images
    - javascripts
    - stylesheets
        - style.sass
        - Our stylesheets can have the extension of either *.sass or *.scss, whatever your preference may be. I will be using *.scss
            - Sass is a preprocessor language that enables writing CSS quickly and efficiently. It enables some programming paradigms in CSS that make writing CSS less painful
- routes
    - Routes allow us to determine how we want to handle a particular request as it enters our application
    - We have two routes setup currently
        - index.js
            - This responds with a default page
        - user.js
            - This will send to a response system (likely a controller) that will handle this request and return with an appropriate response
- views
    - error.ejs
        - Default error view
    - index.ejs
        - The 'home' page delivered by the index route
- package.json
    - Manages our dependencies, scripts, and application info
- package-lock.json
    - Stores our version information and any other decencies a dependency may require

Running Our Default Application

1. To run our application, navigate to the **our_first_app** in your command line tool
2. Run
   $ npm start

3. If everything is good, you should see

Author: Shaun McKinnon

BREAK — 00:15

Changing the Text and SASS — 00:30

Quick Look at EJS

- As said before, **EJS** is a templating language that allows us to write HTML pages with embedded Javascript quickly
- Let's look at what is happening in the **index.js** file
  - **block content**
    - This identifies our content block. Everything defined under this will be considered when rendering the **layout**

/views/index.ejs

```
<!DOCTYPE html>
<html>
  <head>
    <title><%= title %></title>

    <link rel='stylesheet' href='/stylesheets/style.css' />
  </head>

  <body>
    <h1><%= title %></h1>
    <p>Welcome to <%= title %></p>
  </body>
</html>
```

Let's Add Bootstrap

1. Copy the following:
   - **<link href="https://stackpath.bootstrapcdn.com/bootstrap/4.1.1/css/bootstrap.min.css" rel="stylesheet" integrity="sha384-WskhaSGFgHYWDcbwN70/dfYBj47jz9qbsMId/iRN3ewGhXQFZCSftd1LZCfmhktB" crossorigin="anonymous">**
2. Paste that in the head of our **index.ejs** file

```
<!DOCTYPE html>
<html>
  <head>
    <title><%= title %></title>

    <!-- Stylesheets -->
    <link href="https://stackpath.bootstrapcdn.com/bootstrap/4.1.1/css/bootstrap.min.css"
rel="stylesheet" integrity="sha384-WskhaSGFgHYWDcbwN70/dfYBj47jz9qbsMId/iRN3ewGhXQFZCSftd1LZCfmhktB"
crossorigin="anonymous">

    <link rel='stylesheet' href='/stylesheets/style.css' />
  </head>
  <body>
    <h1><%= title %></h1>
    <p>Welcome to <%= title %></p>
  </body>
</html>
```

3. Copy the following:
   - **<script src="https://code.jquery.com/jquery-3.3.1.min.js" integrity="sha256-FgpCb/KJQlLNfOu91ta32o/NMZxltwRo8QtmkMRdAu8=" crossorigin="anonymous"></script>**
   - **<script src="https://stackpath.bootstrapcdn.com/bootstrap/4.1.1/js/bootstrap.min.js" integrity="sha384-smHYKdLADwkXOn1EmN1qk/HfnUcbVRZyYmZ4qpPea6sjB/pTJ0euyQp0Mk8ck+5T" crossorigin="anonymous"></script>**
4. Paste after the **<p>** tag in the **index.ejs** file

```
<!DOCTYPE html>
<html>
  <head>
    <title><%= title %></title>

    <!-- Stylesheets -->
    <link href="https://stackpath.bootstrapcdn.com/bootstrap/4.1.1/css/bootstrap.min.css"
rel="stylesheet" integrity="sha384-WskhaSGFgHYWDcbwN70/dfYBj47jz9qbsMId/iRN3ewGhXQFZCSftd1LZCfmhktB"
crossorigin="anonymous">

    <link rel='stylesheet' href='/stylesheets/style.css' />
  </head>
  <body>
    <h1><%= title %></h1>
    <p>Welcome to <%= title %></p>

    <!-- Javascript -->
    <script src="https://code.jquery.com/jquery-3.3.1.min.js" integrity="sha256-
FgpCb/KJQlLNfOu91ta32o/NMZxltwRo8QtmkMRdAu8=" crossorigin="anonymous"></script>
    <script src="https://stackpath.bootstrapcdn.com/bootstrap/4.1.1/js/bootstrap.min.js"
integrity="sha384-smHYKdLADwkXOn1EmN1qk/HfnUcbVRZyYmZ4qpPea6sjB/pTJ0euyQp0Mk8ck+5T"
crossorigin="anonymous"></script>
  </body>
</html>
```

Let's Change Some Text

Open **index.js**

```
<!DOCTYPE html>
<html>
  <head>
    <title><%= title %></title>

    <!-- Stylesheets -->
    <link href="https://stackpath.bootstrapcdn.com/bootstrap/4.1.1/css/bootstrap.min.css" rel="stylesheet"
integrity="sha384-WskhaSGFgHYWDcbwN70/dfYBj47jz9qbsMId/iRN3ewGhXQFZCSftd1LZCfmhktB"
crossorigin="anonymous">

    <link rel='stylesheet' href='/stylesheets/style.css' />
  </head>
  <body>
    <div class="container">
      <header>
        <div class="jumbotron">
          <h1>Our First Application</h1>
        </div>
      </header>

      <article>
        <div class="row">
          <div class="col-sm-6">
            <div class="card">
              <div class="card-body">
                <h5 class="card-title">Card 1</h5>
                <p>Hello World</p>
                <a href="#" class="btn btn-primary">Card 1 Button</a>
              </div>
            </div>
          </div>

          <div class="col-sm-6">
            <div class="card">
              <div class="card-body">
                <h5 class="card-title">Card 2</h5>
                <p>Goodbye World</p>
                <a href="#" class="btn btn-primary">Card 2 Button</a>
              </div>
            </div>
          </div>
        </div>
      </article>
    </div>

    <!-- Javascript -->
    <script src="https://code.jquery.com/jquery-3.3.1.min.js" integrity="sha256-
FgpCb/KJQlLNfOu91ta32o/NMZxltwRo8QtmkMRdAu8=" crossorigin="anonymous"></script>
    <script src="https://stackpath.bootstrapcdn.com/bootstrap/4.1.1/js/bootstrap.min.js"
integrity="sha384-smHYKdLADwkXOn1EmN1qk/HfnUcbVRZyYmZ4qpPea6sjB/pTJ0euyQp0Mk8ck+5T"
crossorigin="anonymous"></script>
  </body>
</html>
```

Let's Change Some Styles

1. Remove the **/stylesheets/style.css**, **/stylesheets/style.css.map**, and the **/stylesheets/style.sass**
2. Add the file **/stylesheets/style.scss**
3. In **app.js**, change the line (24) **indentedSyntax: true** to **indentedSyntax: false**
4. SCSS is very similar to CSS, but has nesting, mixins, and variables which still give it power. The one big thing over SASS is you can natively run vanilla CSS which means those powers are completely optional

/stylesheets/style.scss

```
/* helpful and more predictable mode */
html {
  -webkit-box-sizing: border-box;
  -moz-box-sizing: border-box;
  box-sizing: border-box;
}
*, *:before, *:after {
  -webkit-box-sizing: inherit;
  -moz-box-sizing: inherit;
  box-sizing: inherit;
}

$background: #ecf0f1;
$button-color: #e74c3c;

/* our styles */
body {
  background-color: $background;

  .container {
    background-color: #fff;
    padding: 1em;
    margin-top: 1em;
    border-radius: 1em;

    .card {
      .btn {
        background-color: $button-color;
        border: $button-color;
      }
    }
  }
}
```

The styles above just modify our default Bootstrap a little. Feel free to modify the styles as you want. Over the next coming weeks we will investigate working with SCSS more, especially nesting, variables, and mixins.

Author: Shaun McKinnon

Working with data in EJS — 00:15

Our JSON Data Structure

In our **/views/index.ejs** file, you will notice this line:

```
res.render('index', {title: 'Express'});
```

The second parameter (after **index** is how we can pass locally scoped data (the data relevant to the view). You will notice that the data is in JSON format. We can modify this to our advantage and stub out some dynamic data:

```
{
  title: 'Express',
  cards: [
    {
      cardTitle: 'Card One',
      cardBody: 'Boorakacha!',
      cardLink: {
        linkLabel: 'To Example One',
        linkHREF: 'http://example.com'
      }
    },
    {
      cardTitle: 'Card Two',
      cardBody: 'Boorakacha!',
      cardLink: {
        linkLabel: 'To Example One',
        linkHREF: 'http://example.com'
      }
    }
  ]
}
```

This can be read as **title = Express**, **cards = an array of card objects**, and each card object has **cardTitle, cardBody,** and **cardLink**. The **cardLink** then has an **object** as well containing **two 'variables', both containing a string value**.

Replace the **{title: 'Express'}** with our new JSON structure:

```
var express = require('express');
var router = express.Router();

/* GET home page. */
router.get('/', function(req, res, next) {
  res.render('index', {
    title: 'Express',
    cards: [
      {
        cardTitle: 'Card One',
        cardBody: 'Boorakacha!',
        cardLink: {
          linkLabel: 'To Example One',
          linkHREF: 'http://example.com'
        }
      },
      {
        cardTitle: 'Card Two',
        cardBody: 'Boorakacha!',
        cardLink: {
          linkLabel: 'To Example One',
          linkHREF: 'http://example.com'
        }
      }
    ]
  });
});

module.exports = router;
```

Dynamifying index.ejs

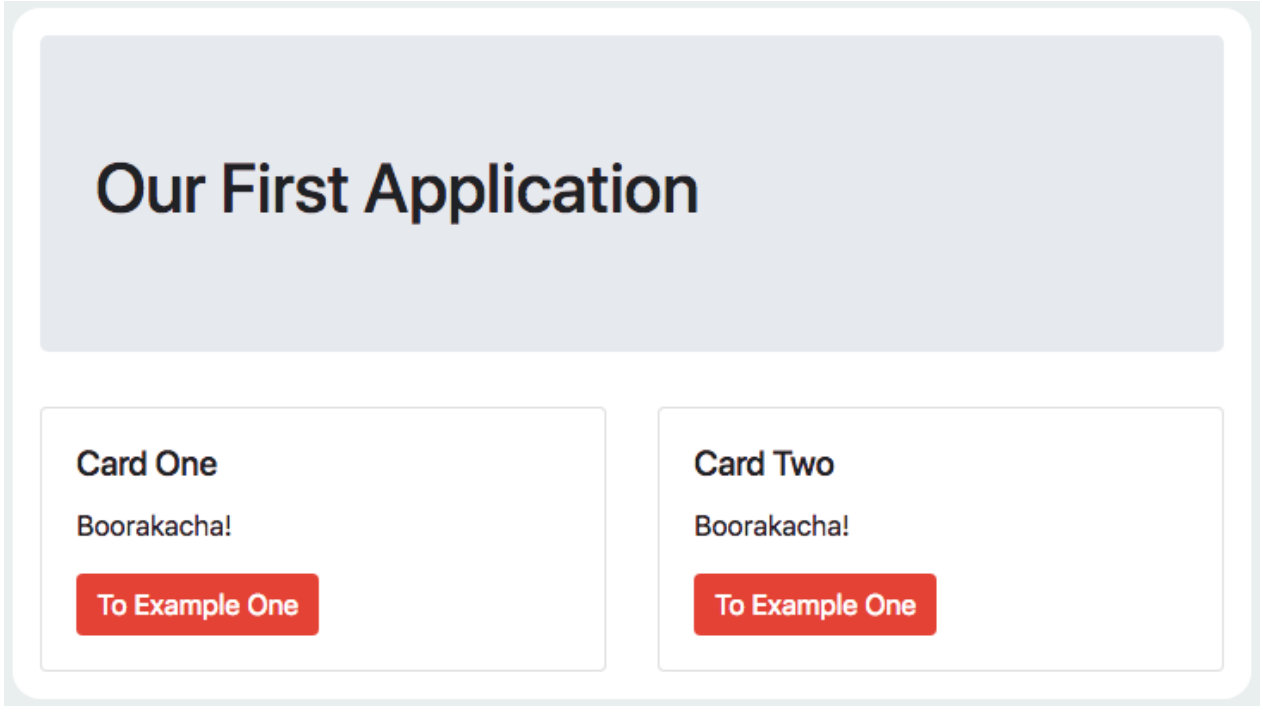
Now that we have some data to work with, we can auto generate the cards needed. One thing to note: Once we change our locals we MUST restart our server so they will take effect.

```
<!DOCTYPE html>
<html>
  <head>
    <title><%= title %></title>

    <!-- Stylesheets -->
    <link href="https://stackpath.bootstrapcdn.com/bootstrap/4.1.1/css/bootstrap.min.css" rel="stylesheet"
integrity="sha384-WskhaSGFgHYWDcbwN70/dfYBj47jz9qbsMId/iRN3ewGhXQFZCSftd1LZCfmhktB"
crossorigin="anonymous">

    <link rel='stylesheet' href='/stylesheets/style.css' />
  </head>
  <body>
    <div class="container">
      <header>
        <div class="jumbotron">
          <h1>Our First Application</h1>
        </div>
      </header>

      <article>
        <div class="row">
          <% for ( let card of cards ) { %>
            <div class="col-sm-6">
              <div class="card">
                <div class="card-body">
                  <h5 class="card-title"><%= card.cardTitle %></h5>
                  <p>
                    <%= card.cardBody %>
                  </p>
                  <a href="<%= card.cardLink.linkHREF %>" class="btn btn-primary">
                    <%= card.cardLink.linkLabel %>
                  </a>
                </div>
              </div>
            </div>
          <% } %>
        </div>
      </article>
    </div>

    <!-- Javascript -->
    <script src="https://code.jquery.com/jquery-3.3.1.min.js" integrity="sha256-
FgpCb/KJQlLNfOu91ta32o/NMZxltwRo8QtmkMRdAu8=" crossorigin="anonymous"></script>
    <script src="https://stackpath.bootstrapcdn.com/bootstrap/4.1.1/js/bootstrap.min.js"
integrity="sha384-smHYKdLADwkXOn1EmN1qk/HfnUcbVRZyYmZ4qpPea6sjB/pTJ0euyQp0Mk8ck+5T"
crossorigin="anonymous"></script>
  </body>
</html>
```

Let's explain what's happening here:

1. We iterate through the **cards**:
   <% for (let card of cards) { %>
   <% } %>
2. Each card has now been assigned to the variable **card** (which is block scoped ;)
3. We then output the value of the **cardTitle** in the **<h5>** element:
   <h5 class="card-title"><%= card.cardTitle %></h5>
4. We do the same for the **cardBody**:
   ** <%= card.cardBody %> **
5. Last, we do it for the **<a>** link. Because the link is two parts (the **HREF** and the **label**), we created an **object**. To **access the object properties** we set, we just use **.chaining**:
   <a href="<%= card.cardLink.linkHREF %>" class="btn btn-primary"><%= card.cardLink.linkLabel %>

Once we're finished, we should still get 2 cards that are unique, but we only had to build one.



Author: Shaun McKinnon