Seneca
SCHOOL OF INFORMATION AND COMMUNICATIONS TECHNOLOGY

WEB322

Web Programming Tools and Frameworks

WEB322 Week 6 Notes

Review: Communicating to the Client

Express.js is a powerful library for helping us create web servers in Node.js. In very few lines of code we can send / receive data in a way that is very straightforward and easy to understand. Recall our first example, where we were able to create two routes: “/” and “/about”, each corresponding to a specific response from our server:

```
var express = require("express");
var app = express();
var path = require("path");

var HTTP_PORT = process.env.PORT || 8080;

// call this function after the http server starts listening for requests
function onHttpStart() {
    console.log("Express http server listening on: " + HTTP_PORT);
}

// setup a 'route' to listen on the default url path (http://localhost)
app.get("/", function(req,res){
    res.send("Hello World<br /><a href='/about'>Go to the about page</a>");
});

// setup another route to listen on /about
app.get("/about", function(req,res){
    res.sendFile(path.join(__dirname, "/week2-assets/about.html"));
});

// setup http server to listen on HTTP_PORT
app.listen(HTTP_PORT, onHttpStart);
```

In the above example, we make use of the **get** method of the app object to define a route and a callback function that’s executed when the route is encountered. We can leverage the 2nd parameter “res” to send either an HTML formatted string (for route “/”), or a static html page (for route “/about”).

If we wanted to send (JSON formatted) data only, we can use the following route (/getData):

```
app.get("/getData", function(req,res){

    var someData = {
    name: "John",
    age: 23,
    occupation: "developer",
```

```
    company: "Scotiabank"
    };

    res.json(someData);
});
```

This will return the JSON-formatted string:

```
{"name":"John","age":23,"occupation":"developer","company":"Scotiabank"}
```

We will find this useful later on when we discuss AJAX & RESTful API's.

The important thing to notice here is that our server can return HTML formatted strings, static HTML (.html) files, and JSON data. However, what if we want to return a valid HTML5 page to the client that actually references some data stored on the server? One solution would be to build out a string that contains both **HTML code** and **values**, ie:

```
app.get("/viewData", function(req,res){

    var someData = {
        name: "John",
        age: 23,
        occupation: "developer",
        company: "Scotiabank"
    };

    var htmlString = "<!doctype html>" +
                     "<html>" +
                         "<head>" +
                             "<title>" + "View Data" + "</title>" +
                         "</head>" +
                         "<body>" +
                             "<table border='1'>" +
                                 "<tr>" +
                                     "<th>" + "Name" + "</th>" +
                                     "<th>" + "Age" + "</th>" +
                                     "<th>" + "Occupation" + "</th>" +
                                     "<th>" + "Company" + "</th>" +
                                 "</tr>" +
                                 "<tr>" +
                                     "<td>" + someData.name + "</td>" +
                                     "<td>" + someData.age + "</td>" +
                                     "<td>" + someData.occupation + "</td>" +
                                     "<td>" + someData.company + "</td>" +
                                 "</tr>" +
                             "</table>" +
                         "</body>" +
                     "</html>";

    res.send(htmlString);
});
```

While this will work to send a valid HTML5 page containing our data back to the client, it's clearly not the best way to approach this problem. What if we had a complex page that contains data in different places throughout the layout? We would be building out an enormous string containing normal, static html and in a few places, inserting a reference to our data (someData object). Wouldn't it be great if we could just write a normal HTML document that **references** the data, instead of having to build one huge string for the whole page?

Template Engine - Handlebars.js

Fortunately, we can leverage "template engines" with express to solve this exact problem. From the express.js documentation:

> *A template engine enables you to use static template files in your application. At runtime, the template engine replaces variables in a template file with actual values, and transforms the template into an HTML file sent to the client. This approach makes it easier to design an HTML page.*

This sounds like exactly what we need, however there are many different template engines that we can choose from. Given that we are all strong HTML developers from our experience in WEB222, why not use a template engine that leverages this experience and works with regular HTML, rather than describing html.

Introducing Handlebars

Handlebars does just that, plus many other features that will help us generate HTML that renders complex data. For example, consider the problem with our "/viewData" route from above; we can leverage the Handlebars template engine to write a simple (separate) HTML5 document that references the data using "handlebars" - ie curly braces {{ }} .

To begin, create the following file in your "views" directory and name it "viewData.hbs":

```
<!doctype html>
<html>
    <head>
        <title>View Data</title>
    </head>

    <body>
        <table border='1'>
            <tr>
                <th>Name</th>
                <th>Age</th>
                <th>Occupation</th>
                <th>Company</th>
            </tr>
            <tr>
                <td>{{data.name}}</td>
                <td>{{data.age}}</td>
                <td>{{data.occupation}}</td>
                <td>{{data.company}}</td>
            </tr>
        </table>
    </body>
</html>
```

This is a much cleaner approach. We no longer have to generate the full page as a string within our "/viewData" route and most importantly, all of the **view** logic (HTML) is separate from our **control** logic (routing).

In order to set this up correctly and get express to understand the file above, we need to modify our server code slightly:

1. The first thing that we need to do is download / install the Handlebars.js package using NPM. Open a terminal in Visual Studio Code (ctrl + `or View -> Integrated Terminal) and make sure that your working directory is somewhere within your project and run the command

   ```
   npm install express-handlebars
   ```

   This will install the "handlebars" package in the same way that we installed the "express" package and update the dependencies in our package.json file:

   ```
   "dependencies": {
       "express": "^4.14.0",
       "express-handlebars": "^5.2.0"
   }
   ```

2. Next, we need to update our server.js file to use our newly installed Handlebars.js module, by adding the line:

```
const exphbs = require('express-handlebars');
```

at the top of our server file.

3. Our server needs to know how to handle HTML files that are formatted using handlebars, so near the top of our code (after we define our “app”), add the lines:

```
app.engine('.hbs', exphbs({ extname: '.hbs' }));
app.set('view engine', '.hbs');
```

This will tell our server that any file with the “.hbs” extension (instead of “.html”) will use the handlebars “engine” (template engine).

4. The next step involves updating our “/viewData” route to “render” our handlebars file with the data:

```
app.get("/viewData", function(req,res){

    var someData = {
        name: "John",
        age: 23,
        occupation: "developer",
        company: "Scotiabank"
    };

    res.render('viewData', {
        data: someData,
        layout: false // do not use the default Layout (main.hbs)
    });

});
```

Now, the route no longer returns a string consisting of our HTML + data using res.send(), but instead invokes the render method on the response object (res). We pass the name of our new file without the extension (ie: “viewData” instead of “viewData.hbs”), and “data” object to hold all of our data (someData).

NOTE Newer versions of Express Handlebars require the layout to be explicitly set to “false” if not using a default layout (see “Layouts / Default Layout” below)

If you have followed all of the steps above, your server.js should look something like this:

```
// setup our requires
const HTTP_PORT = 8080;
const express = require("express");
const exphbs = require("express-handlebars");

const app = express();

// call this function after the http server starts listening for requests
function onHttpStart() {
    console.log("Express http server listening on: " + HTTP_PORT);
}

// Register handlebars as the rendering engine for views
app.engine(".hbs", exphbs({ extname: ".hbs" }));
app.set("view engine", ".hbs");

app.get("/viewData", function(req,res){

    var someData = {
        name: "John",
        age: 23,
        occupation: "developer",
```

```
        company: "Scotiabank"
    };

    res.render('viewData', {
        data: someData,
        layout: false // do not use the default Layout (main.hbs)
    });

});

// start the server to listen on HTTP_PORT
app.listen(HTTP_PORT, onHttpStart);
```

You should also have a “viewData.hbs” file in your “views” directory.

Now, when you navigate to the **/viewData** route, you should see an HTML table rendered with all of your data!

Adding Logic to Templates

Handlebars not only provides us with a mechanism to serve static HTML files with data from the server; it also has a number of Built-In Helpers that we can use within our templates to help render the data. Helpers in Handlebars use the following pattern:

```
{{#helperName context}} block {{/helperName}}
```

where **helperName** represents the name of the helper (built-in or custom), **context** represents a (optional) variable passed in to the helper and **block** is the block of HTML / Handlebars logic affected by the helper.

“if” Helper

The “if” helper is used when we wish to conditionally render a block of HTML. It works by looking at the variable passed into its “context” argument and if it holds **true**, render the block. It also has the notion of an {{else}} clause which will cause a separate block of HTML to render if the argument holds false. It is important to note that comparison operators cannot be used here and code like {{#if data.name == “Joe”}} will not work. This is designed to take a variable and check for true / false. For example, say we wish to conditionally show our developer “John”:

```
var someData = {
    name: "John",
    age: 23,
    occupation: "developer",
    company: "Scotiabank",
    visible: true
};
```

Notice, we have added a “visible” property that we can reference before we render “someData” in our view. Using the {{#if variable}} … {{else}} … {{/if}} construct, we can easily hide or show rows in the table:

```
<table border='1'>
    <tr>
        <th>Name</th>
        <th>Age</th>
        <th>Occupation</th>
        <th>Company</th>
    </tr>
    {{#if data.visible}}
        <tr>
            <td>{{data.name}}</td>
            <td>{{data.age}}</td>
            <td>{{data.occupation}}</td>
            <td>{{data.company}}</td>
```

```
        </tr>
    {{else}}
        <tr>
            <td colspan="4">User: '{{data.name}}' has hidden their information</td>
        </tr>
    {{/if}}
</table>
```

“unless” Helper

The “unless” helper functions in the same way as the “if” helper, only it renders the block if the “context” variable holds **false** instead of true. For example, say we add another property to our “someData” object; in this case, we wish to store whether or not “John” is a contract employee:

```
var someData = {
    name: "John",
    age: 23,
    occupation: "developer",
    company: "Scotiabank",
    visible: true,
    contract: false
};
```

We can add the {{#unless}} helper to only show the “Full-Time employee” table row if the “contract” property is explicitly set to **false**. So, instead of needing true to show the block (like the “if” helper), we must have a **false** value to show the block.

```
{{#unless data.contract}}
    <tr>
        <td colspan="4">Full-Time employee</td>
    </tr>
{{/unless}}
```

“each” Helper

The “each” helper is one of the most useful built-in helpers available to us in Handlebars. It is often the case that the data we wish to render belongs to a collection / array, rather than a single object. The “each” helper automatically repeats a block for every element in the collection (specified by the “context” variable) and gives us a mechanism to render the values in the current iteration. For example, any variables (properties) referenced within the #each block will automatically use the current iteration as the working object.

Additionally, we can specify an {{else}} clause that will render it’s block if the specified “context” variable does not contain any elements (ie, the list is empty).

To see how this works in practice, let’s make someData an array of objects:

```
var someData = [{
    name: "John",
    age: 23,
    occupation: "developer",
    company: "Scotiabank"
},
{
    name: "Sarah",
    age: 32,
    occupation: "manager",
    company: "TD"
}];
```

In order for us to reference each object in the array, we need to use the #each helper to iterate over each element. We can create a <tr> element that will repeat for every element in the "data" collection and reference each property we want to show within a <td>:

```
{{#each data}}
    <tr>
        <td>{{name}}</td>
        <td>{{age}}</td>
        <td>{{occupation}}</td>
        <td>{{company}}</td>
    </tr>
{{else}}
        <tr>
        <td colspan="4">No Data Available</td>
    </tr>
{{/each}}
```

In our case, this will cause the rendered table to contain two <tr> elements, each containing 4 columns with our corresponding data! If for some reason the "data" collection is empty, the {{else}} clause will be used and a single row containing the message "No Data Available" will be rendered. This is preferable to simply showing an empty table, as users will immediately think that something went wrong and/or the page is broken.

Note: if you wish to access a specific element, you can use the syntax data.[1].name - which in this case, will output "Sarah"

The "each" helper also exposes an "@index" variable that allows us to reference the current iteration index (ie, 0, 1, 2, 3, etc.), for example:

```
{{#each data}}
    <tr>
        <td>{{@index}}</td>
        <td>{{name}}</td>
        <td>{{age}}</td>
        <td>{{occupation}}</td>
        <td>{{company}}</td>
    </tr>
{{else}}
        <tr>
        <td colspan="5">No Data Available</td>
    </tr>
{{/each}}
```

Writing Custom Helpers

While the built-in helpers mentioned above simplify working with data in our HTML (.hbs) files, there are many situations where it is preferable to code our *own* helpers. For example, maybe we wish to define a container that always uses the same html pattern, such as the "Warning" alert used in the Bootstrap framework:

```
<div class="alert alert-warning alert-dismissible" role="alert">
    <button type="button" class="close" data-dismiss="alert" aria-label="Close">
        <span aria-hidden="true">&times;</span>
    </button>
    ... Some Warning Text ...
</div>
```

Rather than writing out this same block of code every time we wish to create a dismissible warning, it would be much easier to create a helper to do it for us:

```
{{#warning}}... Some Warning Text ...{{/warning}}
```

To define this and other custom helpers, we need to add a "helpers" property to the Express Handlebars configuration object:

```
app.engine('.hbs', exphbs({
    extname: '.hbs',
    helpers: {
        helper1: function(options){
            // helper without "context", ie {{#helper}} ... {{/helper}}
        },
        helper2: function(context, options){
            // helper with "context", ie {{#helper context}} ... {{/helper}}
        }
    }
}));
```

The name of each property belonging to the “helpers” object corresponds to the name of the helper (ie, “helper1” and “helper2”). The helpers themselves are defined as a function with either 1 or 2 parameters:

- **helper1: function(options)**

  Helpers with only the “options” parameter are used to define helpers in the pattern {{#helperName}} … {{/helperName}}. These types of helpers are generally used to solve the problem outlined above (ie, wrapping content in HTML). The “options” parameter contains a property named fn, which is a method used to reference the content within the helper. By invoking the method with the “this” keyword, we ensure that the context is correctly set for any other helpers that are nested inside this one. This ensures that “options.fn() will return the correct content.

  In short, referencing “options.fn(this)” allows us to build a string containing new data/html in addition to the existing content within the helper. If we return this new string from the function, Handlebars will use this string in place of the helper in our view!

  For example, consider the following helper that wraps text in a “” element:

  ```
      strong: function(options){
          return '<strong>' + options.fn(this) + '</strong>';
      }
  ```

  To use this helper within our .hbs files, we use the syntax:

  ```
  name: {{#strong}}{{data.name}}{{/strong}}
  ```

- **helper2: function(context, options)**

  Helpers with both the “context” and “options” parameters are used to define helpers in the pattern {{#helperName context}} … {{/helperName}}. These types of helpers are generally used to create “iterative” helpers that work with collections of data, much like the built-in “each” helper. The “context” parameter refers to the object passed into the helper as the “context” variable (ie: {{#helperName **context**}}). When building our new string to return from the function, we can use the context variable with the options.fn() function to set the correct context for the current iteration within the helper (in the same way that we used “this” in the previous example).

  For example, consider the following helper that works to create an unordered list:

  ```
  list: function(context, options) {
      var ret = "<ul>";

      for(var i = 0; i < context.length; i++) {
          ret = ret + "<li>" + options.fn(context[i]) + "</li>";
      }

      return ret + "</ul>";
  }
  ```

  To use this helper within our .hbs files, we use the syntax:

```
{{#list data}}
    Name: {{name}} Age: {{age}} Occupation: {{occupation}} Company: {{company}}
{{/list}}
```

This will result in two list items nested within an unordered list.

“Partial” Templates

As we have seen, built-in & custom helpers are great for helping us create reusable chunks (blocks) of code that work with data. This can simplify the development of our “views” (.hbs files) and make them easier to maintain. However, what if we have a larger block of HTML code that we know will be repeated more than once in your app or website? Maybe you want to divide your view into multiple files so that it’s easier to maintain (ie, the “marketing” section should be separate from the “sales” section, etc). Fortunately, Handlebars comes to the rescue again by providing us with the notion of **“Partial” Templates**.

Partial Templates are separate .hbs files that are included dynamically within your working view. By default, these files are located in a directory called “partials” within your “views” directory. If we place our partial templates within this directory, or server will have no problem locating them and we don’t have to explicitly state where they are located. If we do decide to store them in a different directory however, we must set the “partialsDir” property in the Express Handlebars configuration object (the same place where we specified our helpers) using the syntax: partialsDir: “some/path/to/partials”.

Since we will just be using the standard locations for our files in the server, we must first create the partials directory (“views/partials”) before adding files to it. Once this is done, we will add the file “welcome.hbs” to this directory, which contains some simple html:

```
<div>
    <strong><em>welcome!</em></strong>
</div>
```

Now, if we want to use this partial in our viewData.hbs view, we simply need to include it with the following syntax: **{{> partialfilename }}**. In our case, we wish to include our newly created “welcome.hbs” template, which can be done using the following line of code:

```
{{> welcome }}
```

This will dynamically pick up the content from our “welcome.hbs” template and insert it in place!

Passing Data to “Partial” Templates

Partial templates are great for reusing large chunks of html or logically dividing up our views into multiple sections. However, we are missing one key piece - adding data to our partial templates. The main reason that we wish to use a view engine like handlebars is to be able to render data seamlessly in our views and it only makes sense that we would want to do this in our “partial views” as well. Again, this turns out to be a simple task and leverages the same idea of “context” from our helpers. That is, to pass data to our views we simply use the “context” variable, ie: **{{> partialfilename context }}**.

Recall, we were working with the “someData” variable in our viewData.hbs view:

```
var someData = {
    name: "John",
    age: 23,
    occupation: "developer",
    company: "Scotiabank"
};

res.render('viewData', {
    data: someData,
    layout: false // do not use the default Layout (main.hbs)
});
```

So, if we update our “welcome.hbs” file to use the “name” property:

```
<div>
    <strong>Welcome <em>{{name}}!</em></strong>
</div>
```

We can pass the partial view the “context” (so that {{name}} makes sense) by passing “data” as the “context variable:

```
{{> welcome data }}
```

Essentially this tells our “welcome” partial view to access any properties on the “data” object, so {{name}} is really {{data.name}}.

Layouts / Default Layout

One of the most convenient features that express handlebars provides in terms of partials & view organization, is the notion of “layouts” and more specifically a “Master” or “Default” layout. Layouts are essentially the *opposite* of “partial” views, containing a full html page with a place to render your view, rather than a smaller block that’s rendered in your view.

Layouts are specified using the “.hbs” extension and are located by default in a directory called “layouts” within your “views” directory. If we place our layouts within this directory, or server will have no problem locating them. Once again, if we decide to store them in a different directory, we must set the “layoutsDir” property in the Express Handlebars configuration object (the same place where we specified our helpers) using the syntax: layoutsDir: “some/path/to/layouts”.

As before, we will be using the standard locations for our files in the server, so we must first create the layouts directory (“views/layouts”) before adding files to it. Once this is done, we will add the file “main.hbs” to this directory, which contains the bulk of the html for our page, plus a {{{body}}} placeholder:

```
<!doctype html>
<html>
    <head>
        <title>View Data</title>
    </head>

    <body>
        <!-- Top Menu -->
        <main>
            <h3>This is the Default Layout!</h3>
            {{{body}}}
        </main>
        <!-- footer -->
    </body>
</html>
```

It is within this {{{body}}} placeholder that our view (.hbs) files (from our “views” directory) will actually be rendered! No longer do we have to place the same common html elements, header, navigation menu, body container, footer etc. on every single page. Now, we can specify them once in our “default” layout and render all of our views in a common location within the layout. Every view will be rendered with the same header, footer, etc.

To set this up correctly and ensure that our new “main.hbs” file is the “default” layout, we need to make a simple addition to our familiar express handlebars configuration object.

```
app.engine('.hbs', exphbs({
    //...
    defaultLayout: 'main'
    //...
}));
```

By setting the “defaultLayout” property to our “main.hbs” layout, we can ensure that every time we “render” our .hbs files, the result will be placed within the {{{body}}} placeholder of our “main” layout.

If we don’t wish to use the default layout for a particular view, we must explicitly set “layout: false” when we invoke the “render” function:

```
res.render('viewData', {
    data: someData,
    layout: false // do not use the default Layout (main.hbs)
});
```

Similairly, we can use a different layout by specifying it in the “layout” property, ie:

```
res.render('viewData', {
    data: someData,
    layout: "otherLayout" // use "otherLayout.hbs" as a layout
});
```

Sources

- http://handlebarsjs.com
- http://expressjs.com/en/guide/using-template-engines.html
- https://www.npmjs.com/package/express-handlebars
