Introduction to Computer Science I
COMP 2406A – Winter 2020

Templating Engines

Dave McKenney
david.mckenney@carleton.ca

Learning Outcomes

by the End of this Lecture, Students that have Completed the Reading Assignment and Review Questions should be Able to:

Use a template engine in your Node.js app

Use the EJS template engine

Use the Pug template engine

Current State of Things

At this point, we can make basic HTTP servers

We can serve static HTML

How can we serve HTML in a dynamic way?
i.e., how can we customize a page's contents

Current State of Things

We could use template literals (` ... ` strings in Javascript)

Pass variables to these strings and substitute their values into the correct positions

Current State of Things

We could add functions to read files and replace specific sections with provided variable values

But we will see we can avoid having to write all the code to do this ourselves…

Writing Javascript functions for generating HTML has a number of downsides:

Prone to mistakes

Hard to read

Clutters server code

Verbose

Not modular

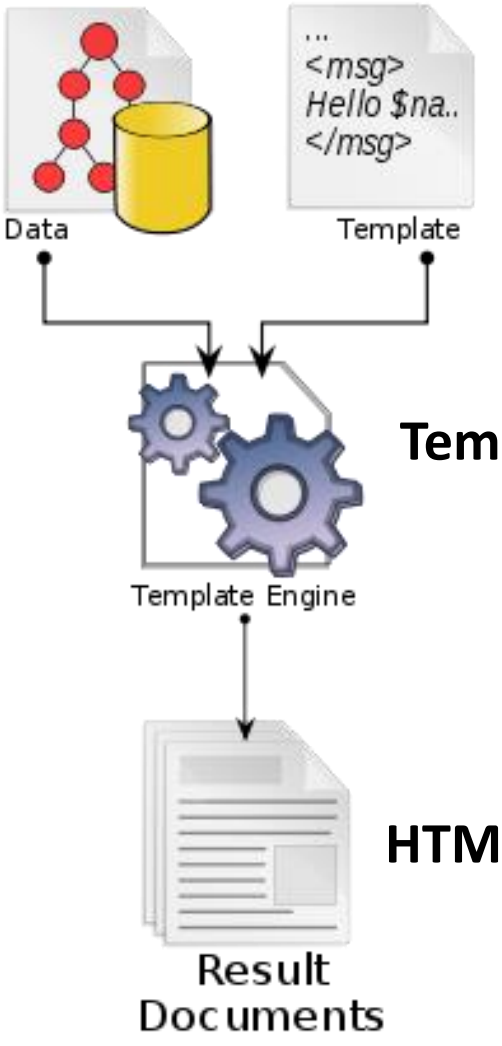
Intro to Template Engines

Server-side template engines allow us to separate our visual specification from data and server logic

We can then combine data with page structure to produce HTML to serve to clients

Intro to Template Engines

Data (e.g., products, product, user, etc.)



Template (document structure and generalized contents)

HTML Page

Intro to Template Engines

We will look at two different template engines: EJS and Pug

Both can be used in tutorials/assignments

Embedded Javascript Templates

EJS (Embedded Javascript) is the first template engine we will look at

Combines HTML with Javascript to dynamically produce some parts of the page

Must be installed using npm:

```
npm install ejs
```

Embedded Javascript Templates

EJS supports the idea of 'partials' – a piece of an entire 'page' that can be re-used (e.g., headers, footers, menus, etc.)

This is a common idea within templating engines

Leads to more modular code

Embedded Javascript Templates

For example, see header.ejs and footer.ejs in the ‘partials’ directory

These define the HTML for the header/footer of a website

Partials can be 'included' into other pages:

```
<%- include("./relative/filepath", data); %>
```

So we can use the previous partials in other pages, such as the index.ejs file in the 'pages' directory

Including EJS in Your Server

Adding EJS to your server involves:

1. **Installing EJS through NPM**
2. **Requiring EJS in your server**
3. **'Rendering' EJS templates within your server**

Including EJS in Your Server

1. **Installing EJS through NPM**

```
npm install ejs
```

2. Requiring EJS in your server

```
const ejs = require("ejs");
```

3. 'Rendering' EJS templates within your server

There are several ways to perform rendering - we will focus on rendering a template stored in a file:

```
ejs.renderFile(filename, data, options, cb);
```

Including EJS in Your Server

Setup a server using EJS to render the home page

Add support for the "About" page too

EJS and Application Data

One of the big advantages of templating is allowing data to be passed into a template

We can handle the data in the template by specifying where we want different pieces of information to be

For example, we could add some data to our home page route:

```
let featured = [
        {name: "Rusty Old Bike", price=15.99},
        {name: "Broken Wooden Chair", price=56.99},
        {name: "Rare Dinosaur Egg", price=9.99}
];
let storeMotto = "We break it, you buy it.";
```

Note: this data can come from any source

When we call the render methods, we can pass an object with data to the template

The properties of the data object can be accessed within the template to retrieve the values

This allows for dynamic page creation

Variables in EJS

To include the value of a variable inside an EJS template, you can use the following syntax:

<%= variableName %>

Note the = after <%, this is important…

Variables in EJS

In EJS:

<% ... %> produces no output and is generally used for control flow

<%= ... %> outputs values and escapes HTML (e.g., <body> → <body>)

<%- ... %> outputs values and does not escape HTML

Escaping HTML is VERY important when working with user-specified data

Branches in EJS

You can add conditionals to EJS:

```
<% if(someVal){ %>
  <h1>Do something for true</h1>
 <% } else{ %>
  <h1>Do something for false</h1>
<% } %>
```

Allows you to create pages that display different information based on the given data (e.g., logged in or not, exists or not, etc.)

Looping in EJS

To loop over an array in EJS:

```
<% features.forEach(function(value){ %>
        //handling code for value
<% }); %>
```

Within the function body, you can include HTML as usual, or additional EJS statements inside <% %>

Variables in EJS

Update the store home page so that the motto and featured items are shown on the page.

EJS vs. Pug

The good about EJS: familiar Javascript syntax

The bad about EJS: it is messy

Now we will look at Pug (formerly known as Jade)

It is less verbose and less messy overall, while still allowing us the same kind of template abilities

Pug relies on whitespace/indentation (like Python)

The start of each line represents an HTML element

Indenting indicates nested tags and there are no closing tags (they are implicit)

Attributes of the HTML elements can be specified in brackets beside the element, like:

a(href="/products") → <a href="/products>

**div(id="menu") → **

script(src="/js/somescript.js") → <script src="/js/somescript.js">

You can specify the CSS class name of an element:

div.main

p.someClass

And can specify an ID with a shorthand using #:

button#submit

input#name(type="textbox")

There are several ways to include plain text in your Pug template. You can also include the text after the HTML tag:

```
p some plain text here
```

Gives you:

```
<p>some plain text here</p>
```

There are several ways to include plain text in your Pug template. You can also use the pipe operator

```
p
  | some plain text here
```

Gives you:

```
<p> some plain text here</p>
```

Pug Hello World

So this Pug code corresponds to…

```
html(lang='en’)
  head
    title Hello World!
  body
    h1 Hello World!
    div.someClass
      p What an impressive Pug example!
```

Pug Hello World

The HTML below:

```
<html lang="en">
 <head>
  <title>Hello World!</title>
 </head>
 <body>
  <h1>Hello World!</h1>
  <div class="someClass">
   <p>What an impressive Pug example!</p>
  </div>
 </body>
</html>
```

Intro to Pug

Like EJS, you can include other files into your Pug template, allowing for partial page definitions

See the example header/footer/index using Pug

Including Pug in Your Server

Adding Pug to your server involves:

1. **Installing Pug through NPM**
2. **Requiring Pug in your server**
3. **'Rendering' Pug templates within your server**

Including Pug in Your Server

1. Installing Pug through NPM

```
npm install pug
```

2. Requiring Pug in your server

```
const pug = require("pug");
```

3. 'Rendering' Pug templates within your server

To render Pug in your server, can render a file similar to what was done with EJS:

```
pug.renderFile('someTemplate.pug', {data})
```

Note: this can have some performance implications

3. 'Rendering' Pug templates within your server

You can also compile a template into a rendering function, and then call that function providing data:

```
const compiledFunction =
pug.compileFile('template.pug');

compiledFunction({data});
```

Including Pug in Your Server

Modify another base server to provide Pug template rendering for the home and about pages.

Incorporating Data in Pug

Also like EJS, you can insert Javascript code into a Pug template by using:

- **Does not add output (e.g., variable declarations)**

= adds to output and escapes HTML

!= adds to output without escaping HTML

Variables are referenced using interpolation: #{someVariableName}

Incorporating Data in Pug

So you can have a Pug template like:

```
html
   body
      - let name = "Dave“
      p= "Your name is " + name
```

Using Javascript code, so string is in “ “

Which gives you:

```
<html><body>
<p>Your name is Dave</p>
</body></html>
```

Incorporating Data in Pug

Or it could look like:

```
html
   body
      - let name = "Dave“
      p Your name is #{name}
```

Using interpolation, so no “ “

Which gives you:

```
<html><body>
<p>Your name is Dave</p>
</body></html>
```

Conditionals in Pug

Pug provides conditional operations:

```
if someBoolean
    p The value was true
else if someOtherBoolean
    p First value was false, second was true
else
    p Both values were false
```

Iterating in Pug

Pug provides the ‘each’ keyword for iteration:

```
- const employees = ['Angela', 'Jim’, Toby',]
ul
   each employee in employees
      li= employee
```

Gives you:

```
<ul><li>Angela</li>
<li>Jim</li>
<li>Toby</li></ul>
```

Completing Pug Example

Modify the server using Pug template rendering to support the features products and store motto.

Summary

Template engines allow us to specify page templates that we can insert data into within our server

This allows us to further separate server logic and display details

With languages such as Pug, we can also write nicer looking HTML code that is not as verbose

Summary

There are many other template engines that you can consider as well, such as: Mustache, Handlebars, dot.js, Nunjucks, etc.

Summary

For further reading on EJS and Pug:

https://ejs.co/

https://pugjs.org/api/getting-started.html

https://codepen.io/mimoduo/post/learn-pug-js-with-pugs

Putting It All Together

We can now make GET/POST requests from a client, handle GET/POST requests on a server, and respond with dynamically generated HTML

Now we will work through a small development example

Putting It All Together

We want to provide a way to manage the store's catalog

We need ways to:

1. **List all products**
2. **View specific products**
3. **Add a new product**
4. **Set a product as featured or not**

Putting It All Together

First off, how we will store the products?

Putting It All Together

We will take a basic approach – storing products in an array on the server

There will be no persistence

Each product will have: an ID (number), name (string), price (number), and featured (Boolean)

Putting It All Together

See the base server code in store-server.js

This contains an array with a few products already created

Putting It All Together

Next – what routes will our server need?

We need ways to:

1. **List all products (GET /products)**
2. **View specific products (GET /products/*someID*)**
3. **Add a new product (POST /products with JSON)**
4. **Set a product as featured or not (POST /products/*someID* with JSON)**

Now that we have a decent idea of how are data will be organized and the routes are server will support, we can start to code.

Putting It All Together

We want to provide a way to manage the store's catalog

We need ways to:

1. **List all products**
2. **View specific products**
3. **Add a new product**
4. **Set a product as featured or not**

Putting It All Together

We will say the route /products will represent all products on the server

So a GET request for /products should receive a page containing the list of products - add handling to the server code to support this

Note – you could add query string parameters to this to perform more advanced filtering of products (e.g., /products?name=rustic)

Putting It All Together

We want to provide a way to manage the store's catalog

We need ways to:

1. **List all products**
2. **View specific products**
3. **Add a new product**
4. **Set a product as featured or not**

Putting It All Together

The route /products represents all products

The route /products/*someNumber* will represent the product with ID = *someNumber*

Putting It All Together

So a GET request for /products/5 should receive a page with the information for product with ID 5

How can you implement this route in code? Add support to the server for this general route

Putting It All Together

We want to provide a way to manage the store's catalog

We need ways to:

1. **List all products**
2. **View specific products**
3. **Add a new product**
4. **Set a product as featured or not**

Putting It All Together

The route /products represents all products

We will treat a POST request to /products as a request to add a product to the catalog

Putting It All Together

We will expect the body of such a request to contain a JSON representation of a new product

Add a page to the server for adding products (how?)

Update the server code to support this new route.

Putting It All Together

We want to provide a way to manage the store's catalog

We need ways to:

1. **List all products**
2. **View specific products**
3. **Add a new product**
4. **Set a product as featured or not**

Putting It All Together

We need a way to update/modify the state of products on the server

How can we do this?

Putting It All Together

POST requests to /products were used to add to (i.e., modify) the list of products

POST requests to /products/*someID* will be used to modify one specific product

We will assume the new information will be contained in the request body

Putting It All Together

We can modify the page returned for GET requests to /products/*someID* to add this functionality

Instead of displaying the value of 'featured' for the product, we can add radio buttons

We can add a 'Save' button to send the changes

Modify the code to provide the ability to modify the featured status of products

Summary

Next, we will begin to clean up our server code and introduce additional modules to give us more power/flexibility and reduce the potential for errors

We will be able to organize a large amount of data with a relatively small amount of code

Summary

Questions?