MVC-based JavaScript Web App

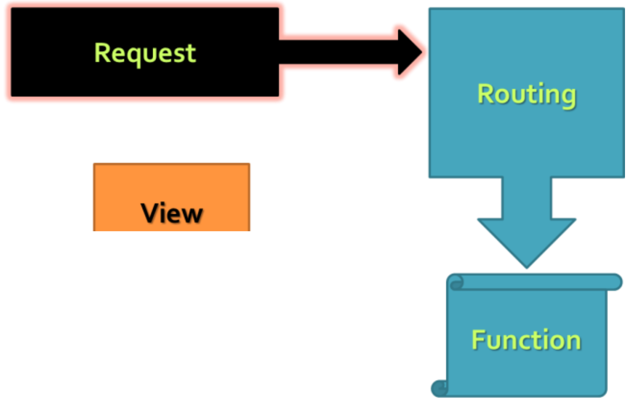
Node.js Express

UI Template using Handlebars

Fetch API

Express

Web Application Framework for Node.js



Back

First Express App

```
let express = require('express');

let app = express();

app.get('/', (request, response) => {
    response.send('Welcome to Express!');
});

app.get('/customer/:id', (req, res) => {
    res.send('Customer requested is ' + req.params['id']);
});

app.listen(3000);
```

HTTP Methods

- app.get(), app.post(), app.put() & app.delete()
- By default Express does not know what to do with the request body, so we should add the ***bodyParser*** middleware

```
app.use( express.bodyParser() );
```

- ***bodyParser*** will parse the request body and place the parameters in the ***req.body***

Post Sample

```html
<form method="post" action="/">
  <input type="text" name="username" />
  <input type="text" name="email" />
  <input type="submit" value="Submit" />
</form>
```

```javascript
app.use(express.bodyParser());

app.post('/', (req, res) => {
    console.log(req.body.user);
    res.send('Welcome ' + req.body.username);
});
```

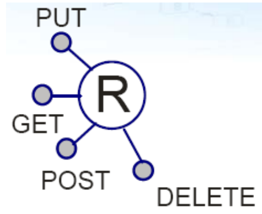
REST Services

Back

What is a REST Service?

- ❐ A server-side component programmatically accessible at a particular URL
- ❐ You can think of it as a Web page returning json instead of HTML
- ❐ A service provides data in standard format mostly JSON format
- ❐ Major goal = **interoperability between heterogeneous systems**

REST Principles



- **Addressable Resources** (nouns): Identified by a URI

(e.g., http://example.com/customers/123)

- **Uniform Interface** (verbs): GET, POST, PUT, and DELETE

  -Use verbs to exchange application state and representation

  -Embracing HTTP as an Application Protocol

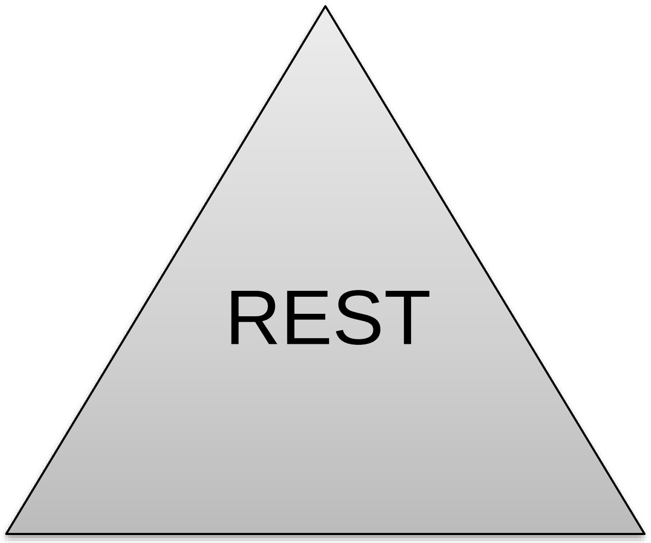
- **Representation-oriented**

  -**Representation of the resource state** transferred between client and server in a variety of data formats: XML, JSON, (X)HTML, RSS..

•**Hyperlinks** define relationships between resources and valid state transitions of the service interaction

REST Services Main Concepts

Nouns (Resources)
e.g., http://example.com/employees/12345



Verbs
e.g., GET, POST

Representations
e.g., XML, JSON

Resources

- The key abstraction in REST is a **resource**
- A resource is a conceptual mapping to a set of entities
  - Any **information that can be named can be a resource**: a document or image, a temporal service (e.g. "today's weather in Doha"), a collection of books and their authors, and so on
- Represented with a global identifier (URI in HTTP)
  - http://www.boeing.com/aircraft/747

Naming Resources

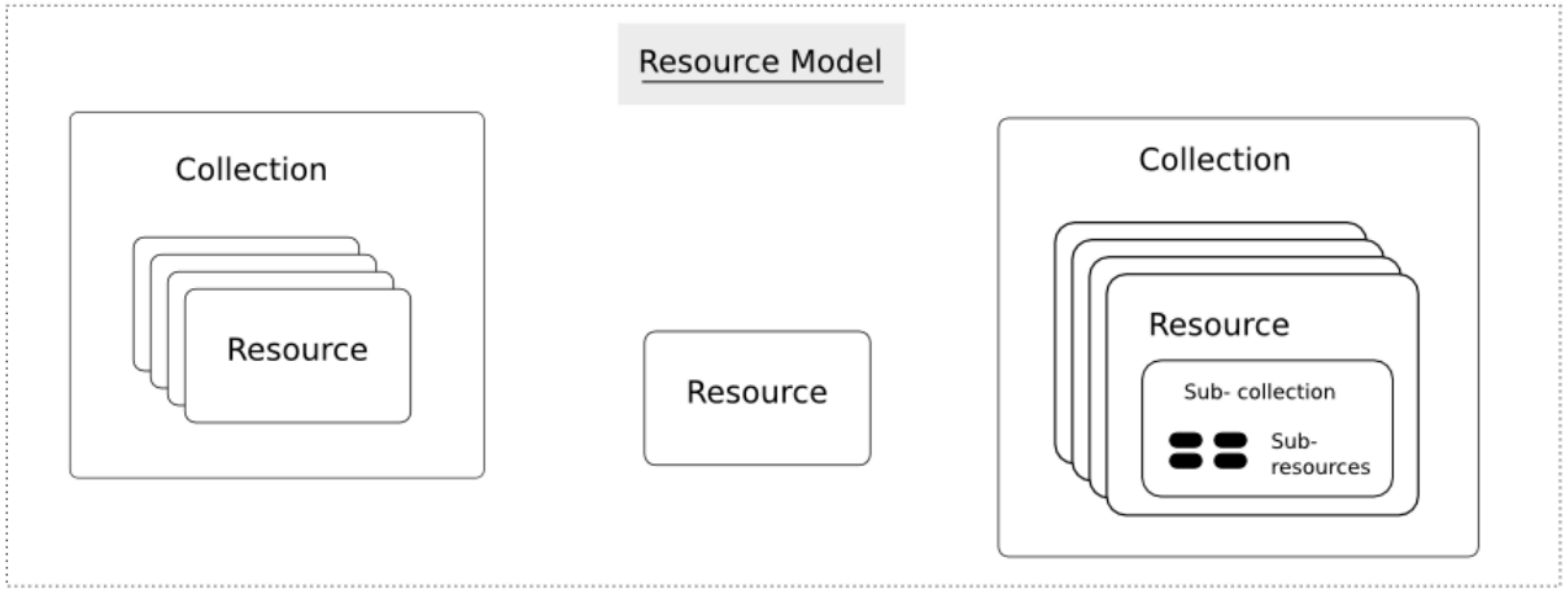
- REST uses URI to identify resources

  - http://localhost/books/
  - http://localhost/books/ISBN-0011
  - http://localhost/books/ISBN-0011/authors

  - http://localhost/classes
  - http://localhost/classes/cmps356
  - http://localhost/classes/cs356/students

- As you traverse the path from more generic to more specific, you are navigating the data

A Collection with Resources



A Collection with Resources

A Singleton Resource

Sub-collections and Sub-resources

Representations

- Specify the data format used when returning a resource representation to the client

- Two main formats:
  - JavaScript Object Notation (JSON)
  - XML

- It is common to have multiple representations of the same data

Representations

- XML

```
<course>
    <code>cmps356</code>
    <name>Enterprise Application
    Development</name>
</course>
```

- JSON

```
{
   code: ‘cmps356’,
    name: ‘Enterprise Application Development’
}
```

HTTP Verbs

- Represent the actions to be performed on resources
- Retrieve a representation of a resource: **GET**
- Create a new resource:
  - Use **POST** when the server decides the new resource URI
  - Use **PUT** when the client decides the new resource URI. Also **PUT** is also typically used for update
- Delete an existing resource: **DELETE**
- Get metadata about an existing resource: **HEAD**
- See which of the verbs the resource understands: **OPTIONS**

REST Services using Node.js

- See posted Node.js REST Services example
- Test them using Postman Chrome plugin

https://www.getpostman.com/

UI Template using Handlebars

http://handlebarsjs.com/

< Back

UI Template

- **View engine** (template engine) is a framework/library that generates views
  - Provide cleaner way to dynamically create DOM elements
- The engine generates a valid HTML based on a template and a given JavaScript object
- There are lots of JavaScript view engines such as Handlebars.js, KendoUI, jQuery, AngularJS, etc.
- **Handlebars.js** is recommended. It is a library for creating client-side or server-side UI templates

Usage

- Add Handlebars script

```
<script src="path/to/handlebars.js"></script>
```

- Create a template

```
<script id="post-template" type="text/x-handlebars-template">
  <div class='post'>
    <h1 class="post-title">{{title}}</h1>
    <p class="post-content">{{{content}}}</p>
  </div>
</script>
```

- Render the template

```
let post = {title: '…', content: '…'},
    htmlTemplate = postTemplateNode.innerHTML,
    postTemplate = Handlebars.compile(htmlTemplate);
postNode.innerHTML = postTemplate(post);
```

Creating HTML Templates

- HTML templates act much like JavaScript String template
  - Put placeholders within a template string, and replace these placeholders with values
- Handlebars.js marks placeholders with double curly brackets `{{value}}`
  - When rendered, the placeholders between the curly brackets are replaced with the corresponding value

HTML Escaping

- Handlebars.js escapes all values before rendering them (i.e., html tags in the value are ignored)
- If the value contains HTML tags that should not be escaped then use triple curly brackets "triple-stash" in the template string

```
{{{value}}}
```

Iterating over a collection of elements

- **{{#each collection}} {{/each}}** block expression is used to iterate over a collection of objects
  - Everything between will be evaluated for each object in the collection

```
<ul class="people_list">
  {{#each people}}
    <li>{{this}}</li>
  {{/each}}
</ul>
```

```
{
  people: [
    "Ali Faleh",
    "Fatma Jasime",
    "Abbas Ibn Firnas"
  ]
}
```

Conditional Expressions

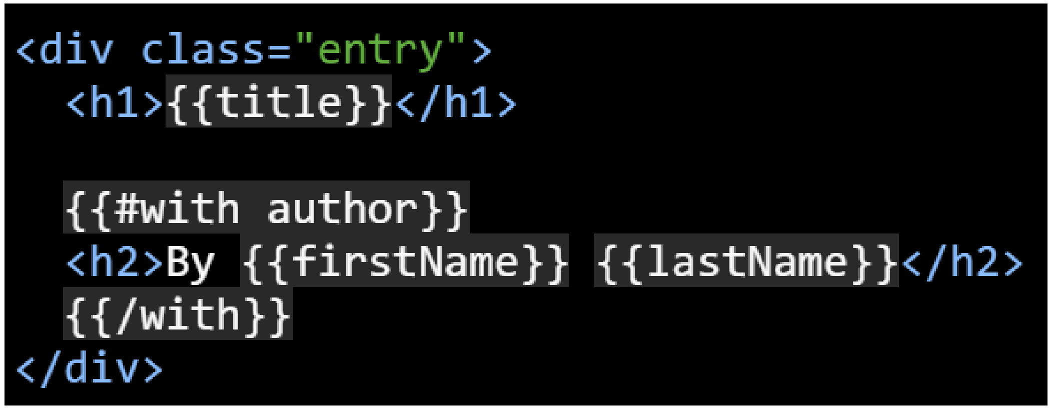
- Render fragment only if a condition is fulfilled
  - Using `{{#if condition}} {{/if}}` or `{{unless condition}} {{/unless}}`

```
<div class="entry">
  {{#if author}}
    <h1>{{firstName}} {{lastName}}</h1>
  {{else}}
    <h1>Unknown Author</h1>
  {{/if}}
</div>
```

```
<div class="entry">
  {{#unless license}}
  <h3 class="warning">WARNING: This entry does not have a license!</h3>
  {{/unless}}
</div>
```

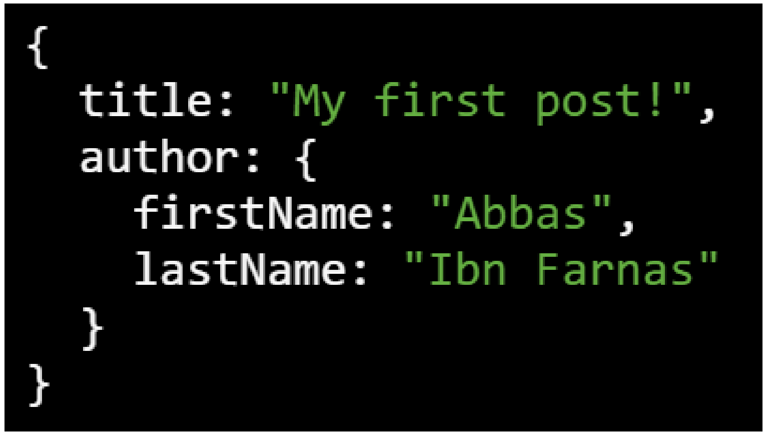
The with Block Helper

- `{{#with obj}} {{/with}}`
  - Used to minify the path
  - Write `{{prop}}` Instead of `{{obj.prop}}`



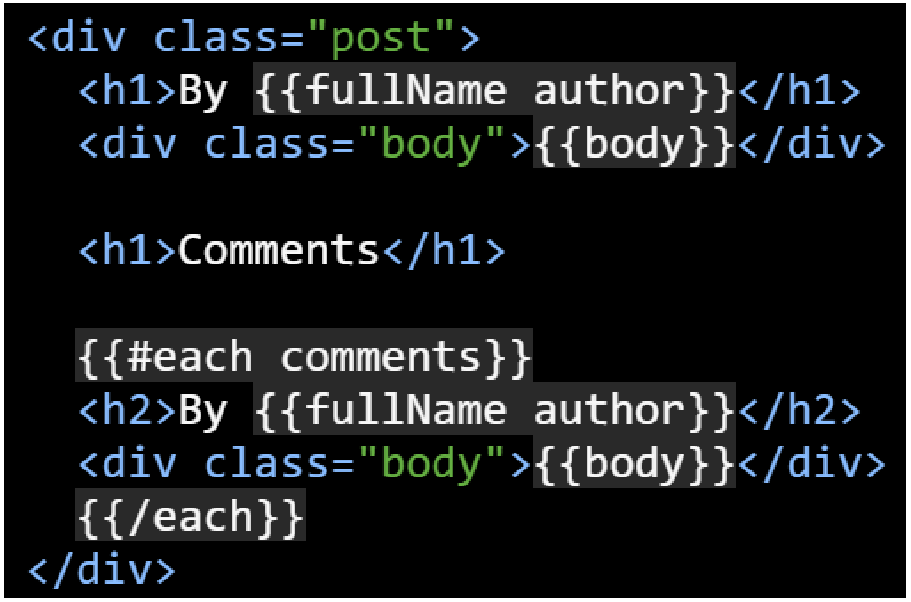
```
<div class="entry">
  <h1>{{title}}</h1>

  {{#with author}}
  <h2>By {{firstName}} {{lastName}}</h2>
  {{/with}}
</div>
```



```
{
  title: "My first post!",
  author: {
    firstName: "Abbas",
    lastName: "Ibn Farnas"
  }
}
```

Custom Helper

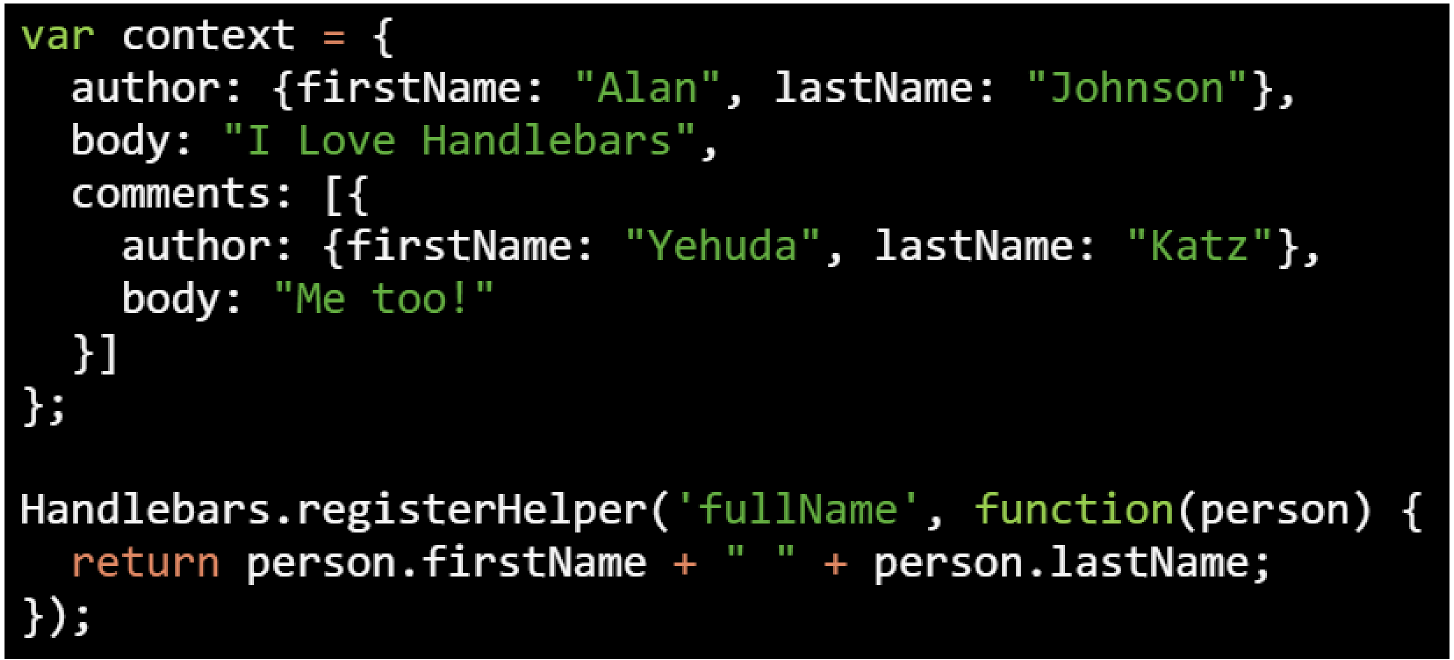


```html
<div class="post">
  <h1>By {{fullName author}}</h1>
  <div class="body">{{body}}</div>

  <h1>Comments</h1>

  {{#each comments}}
  <h2>By {{fullName author}}</h2>
  <div class="body">{{body}}</div>
  {{/each}}
</div>
```

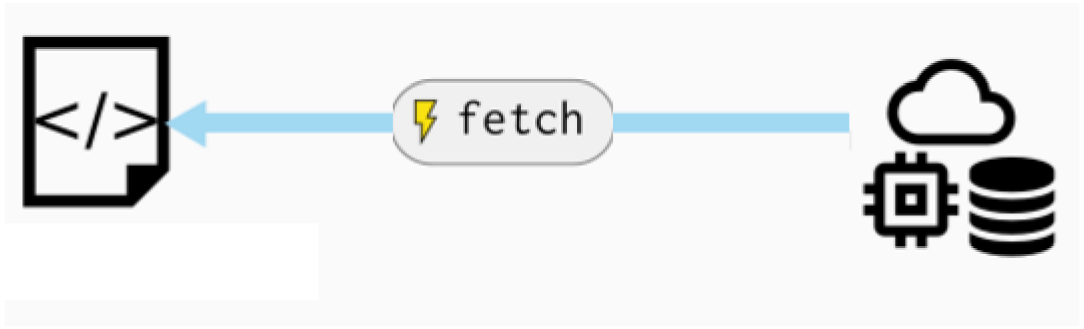
- You can register a helper with the **Handlebars.registerHelper** method



```javascript
var context = {
  author: {firstName: "Alan", lastName: "Johnson"},
  body: "I Love Handlebars",
  comments: [{
    author: {firstName: "Yehuda", lastName: "Katz"},
    body: "Me too!"
  }]
};

Handlebars.registerHelper('fullName', function(person) {
  return person.firstName + " " + person.lastName;
});
```

Communicating with the server using Fetch API



Back

- AJAX is acronym of Asynchronous JavaScript and XML
  - AJAX == technique for asynchronously loading (in the background) of dynamic Web content and data from the Web server into a HTML page
  - Allows dynamically changing the DOM (client-side) in Web applications

- Two styles of AJAX
  - Partial page rendering
    - Load an HTML fragment and display it in a **`<div>`**
  - Call REST service then client-side rendering of received JSON
    - Loading a JSON object and render it at the client-side with JavaScript / jQuery

Getting a resource from the server using Fetch API

- Fetch content from the server

```
let url = "data/student.json";
fetch(url).then(response => response.json())
          .then(students => {
             console.log(students);
          })
          .catch(err => console.log(err));
```

- Fetch returns a Promise. Promise-fulfilled event(.then) receives a Response object.
- **.json()** method is used to get the response body into a JSON object

Posting a request to the server using Fetch API

- Fetch could be used to post a request to the server

```
let email = document.querySelector( "#email" ).value,
  password = document.querySelector("#password").value;

fetch( "/login", {
    method: "post",
    headers: { "Accept": "application/json",
               "Content-Type": "application/json" }
    body: JSON.stringify({
        email,
        password
    })
});    //headers parameter is optional
```

Resources

- NodeSchool

https://nodeschool.io/

- Mozilla Developer Network:

https://developer.mozilla.org/en-US/docs/Learn/Server-side/Express_Nodejs