Lecture 8

Node.js (part 3)

Today’s Content

- EJS template engine
- Handlebars template engine
- Working with template engines
- MVC model
- Node.js Crypto Module

Template Engine - EJS

What is EJS?

- EJS is a simple templating language that lets you generate HTML markup with plain JavaScript.
- No re-invention of iteration and control-flow. It's just plain JavaScript.
- A template engine allows you to store HTML templates on the server-side and render dynamic contents into these templates before sending the final HTML code to the client.
- Besides EJS, other popular template engines such as Handlebars and Pug are also commonly used in Node.js applications.

EJS

- Install `ejs` module

```
npm install -s ejs
```

- Set view engine to `ejs`

```
const express = require('express');
const app = express();
app.set('view engine', 'ejs');
```

Rendering EJS

- Render a template

```
app.get('/', (req, res) => {
    res.render('pages/index');
});
```

- `res.render()` will look in the `views` folder for the view.
  - The full path to the template above should be `views/pages/index.ejs`

Render an EJS template with data

- Render a template with a `data` object

```
res.render('pages/index', data);
```

- This data object is accessible in the EJS template.
  - Put all data that you want to display on the web page into this object.
  - The *keys* from this object become *named variables* in the template.
    (see examples in later slides)

Example data object

```
let data = {
    title: 'About page',
    user: {
        id: 1,
        email: "quandd@hanu.edu.vn",
        name: "Dang Dinh Quan"
    }
};
res.render('pages/index', data);
```

VSCode EJS Language Support

<% EJS %>

EJS language support v1.3.1

DigitalBrainstem | 694,495 | ★★★★☆ (27)

2019 - EJS language support for Visual Studio Code.

Install

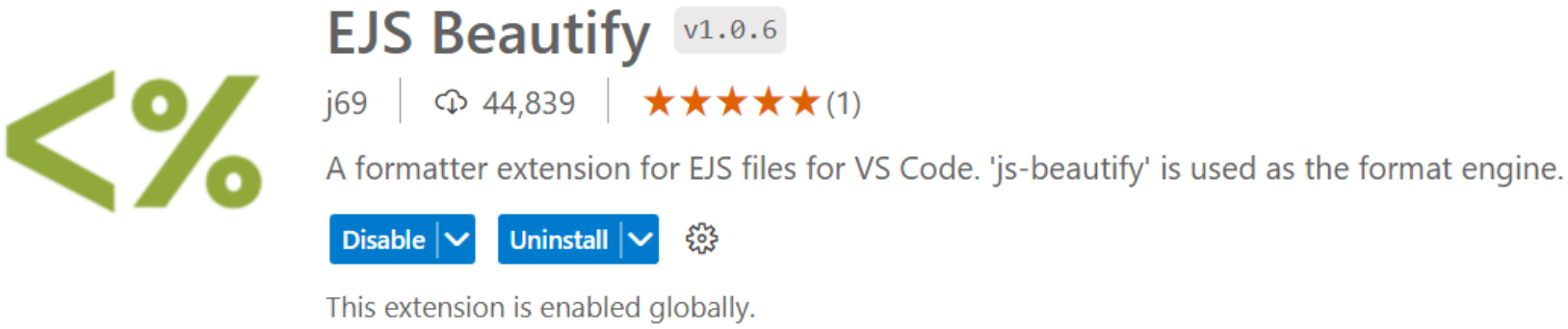
Details Feature Contributions Changelog

Visual Studio Marketplace v1.3.1 license MIT

EJS Language Support

- This extension adds EJS code highlighting

VSCode EJS Beautify



- You should configure VSCode to use this extension as default formatter for `.ejs` files.

EJS Syntax

- Displaying a value
  (HTML escaped)

```
<h2><%= user.name %></h2>
```

- Adding control-flow
  (or any other JavaScript code)

```
<%
let x=5;
let arr=["abc", "def", "ghj" ];
%>
```

- Display a raw value
  (without HTML escape)

```
<p><%- user.name %></p>
```

EJS Syntax

- Embedding JavaScript code in an EJS template:

```
<% for (let i = 0; i < links.length; i++) { %>
    <li class="nav-icon">
        <a href="<%= links[i].href %>" class="nav-link">
            <%= links[i].icon %>
            <span class="link-text">
                <%= links[i].text %>
            </span>
        </a>
    </li>
<% } %>
```

EJS Partials

- Front-end materials (HTML/CSS/JS) that are re-used.
  - Example: footer, header, menu, widgets...
- Adding EJS Partials to Views

```
<body>
    <%- include('partials/navbar') %>
    <div id="main">
        <!-- page body goes here -->
    </div>
    <%- include('partials/footer') %>
</body>
```

EJS Partials

- The EJS partial has access to all the same data as the parent view. But be careful. If you are referencing a variable in a partial, it needs to be defined in every view that uses the partial or it will throw an error.
- You can also define and pass variables to an EJS partial in the include syntax like this:

```
<header>
    <%- include('../partials/header', {variant: 'compact'}); %>
</header>
```

Checking for undefined variables in EJS

- In EJS templates & partials, you can use the following syntax to check if a variable exists or not, before using:

```
<div>
  Username:<br />
  <%- typeof userErr !== 'undefined' ? `<div class="err">${userErr}</div>` : '' %>
  <input
    type="text"
    name="username"
    value="<%= typeof username !== 'undefined' ? username : '' %>" />
</div>
```

Template Engine - Handlebars

Handlebars vs EJS: Why Learn Both?

You’ve already learned about EJS, but now we're introducing Handlebars.

So, why learn both?

Review EJS

Definition: EJS (Embedded JavaScript) is a templating engine that lets you generate HTML using JavaScript logic.

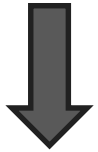
Key Features:

- Simple syntax similar to JavaScript.
- Allows embedding JS code directly into HTML.
- Uses `<% %>` for control flow and `<%= %>` for output.

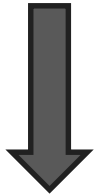
Use Case: Dynamic web pages with JavaScript-heavy logic.

Why Handlebars?

EJS **allows us** to mix **JavaScript logic with HTML**,



It make the code harder to read, especially for larger projects.



Handlebars

- Clear and concise templates.
- Easy to learn and use.
- Encourages writing logic-free templates, promoting code readability.

What is Handlebars?

Definition: Handlebars is a logic-less template engine that helps to generate HTML with dynamic content from the server.

Key Features:

- Logic-less: No embedded JavaScript inside the templates.
- Simpler, more focused on data rendering.
- Uses `{{}}` for output and helpers.

Use Case: Rendering views with minimal logic and emphasis on separation of concerns.

Key Differences Between Handlebars and EJS

Feature	EJS	Handlebars
Syntax	JavaScript-like `<% %>`	Simpler, logic-less `{{}}`
Logic in Views	Can embed full JavaScript code	Focuses on keeping logic outside templates
Complexity	Allows more complex logic within templates	Simpler, logic moved to helpers
Use of Helpers	Limited, usually relies on JS code	Extensive use of helpers for logic handling
Maintainability	May get cluttered with logic	Cleaner, logic is separated from presentation

Handlebars Syntax Basics

- Embedding variables with `{{variableName}}`.
- Handlebars supports simple expressions, no complex logic.
- By default, Handlebars escapes HTML to prevent injection attacks.

```
<h1>{{title}}</h1>
<p>{{description}}</p>
```

Handlebars Helpers

- Handlebars provides helpers for more functionality within templates.

```
{{#if isAdmin}}
  <p>Welcome, Admin!</p>
{{/if}}
```

- Built-in Helpers: `{{#if}}`, `{{#each}}`, etc.

Partials

- Handlebars partials allow for code reuse by creating shared templates. You can put partials inside the `views/partials` sub-directory.

- You can embed partials in views or layouts using:

```
{{>partial}}
```

- The following template and input:

```
{{#each persons}}
  {{>person person=.}}
{{/each}}
```

```
{
  persons: [
    { name: "Nils", age: 20 },
    { name: "Teddy", age: 10 },
    { name: "Nelson", age: 40
},],}
```

Setting Up `Handlebars` with `Express`

Install Handlebars

```
npm install express-handlebars
```

This view engine uses sensible defaults that leverage the "Express-way" of structuring an app's views. This makes it trivial to use in basic apps:

Directory Structure:

```
.
├── app.js
└── views
    ├── home.handlebars
    └── layouts
        └── main.handlebars

2 directories, 3 files
```

app.js file

This is a simple **`Express`** app with **`Handlebars`** as the **`view engine`**. When the user visits the home page **`(/)`**, the app renders the home view from the **`home.handlebars`** file in the views folder.

Handlebars helps create dynamic interfaces with Express by separating layouts and templates.

```
import express from 'express';
import { engine } from 'express-handlebars';

const app = express();

app.engine('handlebars', engine());
app.set('view engine', 'handlebars');
app.set('views', './views');

app.get('/', (req, res) => {
    res.render('home');
});
app.listen(3000);
```

Rendering the home view from the **home.handlebars** file

`res.render()` function

Render a Handlebars view with Express:

```
app.get('/', (req, res) => {
  res.render('home', { title: 'Welcome', description: 'This
is Handlebars!' });
});
```

`res.render('home', {...}):`

This is the core of rendering a Handlebars view. The `res.render()` method does two things:

- It looks for a Handlebars file named `home.handlebars` (or `home.hbs`, depending on your setup) inside the `views` directory.
- It passes the data inside the object `{ title: 'Welcome', description: 'This is Handlebars!' }` to that template.

`home.handlebars:`

```
<h1>{{title}}</h1><p>{{description}}</p>
```

views/layouts/main.handlebars:

The main layout is the HTML page wrapper which can be reused for the different views of the app. **{{{body}}}** is used as a placeholder for where the main content should be rendered.

```
<!DOCTYPE html>
<html>
<head>
    <meta charset="utf-8">
    <title>Example App</title>
</head>
<body>

    {{{body}}}

</body>
</html>
```

views/home.handlebars:

The content for the app's home view which will be rendered into the layout's `{{{body}}}`.

```
<h1>Example App: Home</h1>
```

Understanding Handlebars layout & partial

Project structure:

```
project-directory
├── app.js
├── views
│   ├── index.hbs
│   ├── partial.hbs
│   └── layouts
│       └── default.hbs
└── package.json
```

Understanding Handlebars layout & partial

Content of `app.js`

```
const express = require('express');
const handlebars = require('express-handlebars');
const app = express();
app.engine('hbs', handlebars.engine({ // setup Handlebars view engine
    defaultLayout: 'default',
    extname: '.hbs',
}));
app.set('view engine', 'hbs'); // use Handlebars
app.get('/', (req, res) => {
    res.render('index', {
        title: 'Home Page',
        content: 'This is the main content.'
    });
});

app.listen(3000, () => console.log('Server listening on port 3000'));
```

Understanding Handlebars layout & partial

Content of `default.hbs`

```
<!DOCTYPE html>
<html>
<head>
    <title>{{title}}</title>
</head>
<body>
    <header>
    </header>
    <main>
        {{{body}}}
    </main>
    <footer>
    </footer>
</body>
</html>
```

Understanding Handlebars layout & partial

If `index.hbs` looks like this:

```
<h1>Home Page</h1>
{{> partial}}
```

And `partial.hbs` looks like this:

```
<p>This is the partial content.</p>
```

Understanding Handlebars layout & partial

Then, when Handlebars renders `index.hbs` and applies the `default.hbs` layout, the final HTML will look something like this:

```
<!DOCTYPE html>
<html>
<head>
    <title>Home Page</title>
</head>
<body>
    <header>
    </header>
    <main>
        <h1>Home Page</h1>
        <p>This is the partial content.</p>
    </main>
    <footer>
    </footer>
</body>
</html>
```

Select Handlebars layout when rendering

```
res.render('person', { layout: 'layout2', ...people[id] });
```

The `...` syntax is called the *Spread Operator*.
Read more about it here.

VSCode Handlebars Language Support

Handlebars v0.4.1

André Junges | 173,264 | ★★★☆☆ (7)

Handlebars Visual Studio Code

Install ☑ Auto Update ⚙

DETAILS FEATURES

Handlebars for Visual Studio Code

MVC model

MVC model

MVC stands for "Model-View-Controller." It is a **design pattern** used in software engineering. MVC is a software architecture pattern for creating user interfaces on computers.

Design patterns are general, reusable solutions to common problems that occur in software design. They provide proven methods and best practices to help developers solve issues more efficiently.

Design Pattern Key Concepts

1. **Purpose**: Help structure software systems in a clear, maintainable, and scalable way.
2. **Categories:**

- Creational patterns: Manage object creation.
  - Examples: Singleton, Factory Method, Abstract Factory.
- Structural patterns: Deal with object and class composition.
  - Examples: Adapter, Decorator, Composite.
- Behavioral patterns: Manage object interaction and communication.
  - Examples: Observer, Strategy, Command.

Design Pattern Key Concepts

3. **Benefits:**

- Reduce repetitive code.
- Make the codebase easier to understand, maintain, and extend.
- Increase flexibility when changing or expanding systems.

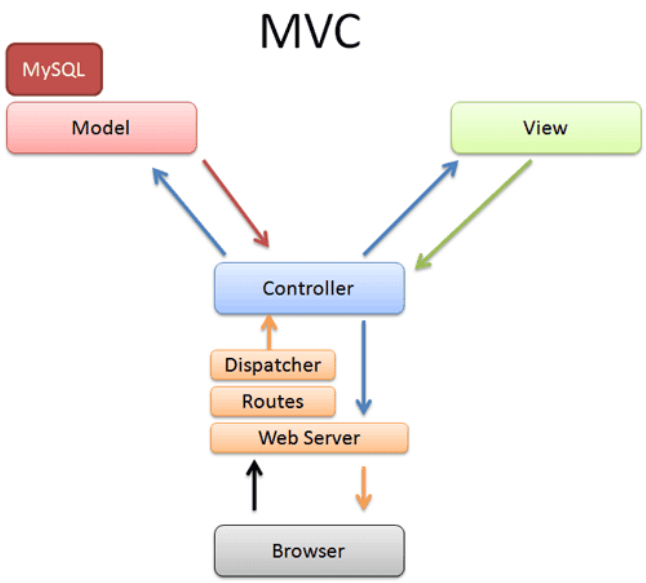
4. **Not specific code:**

- Design patterns are conceptual solutions or templates, not concrete code. Developers can adapt them to suit their project's context.

➢ **In summary,** design patterns are standardized solutions that help developers address common design challenges in a more efficient and structured way.

MVC pattern

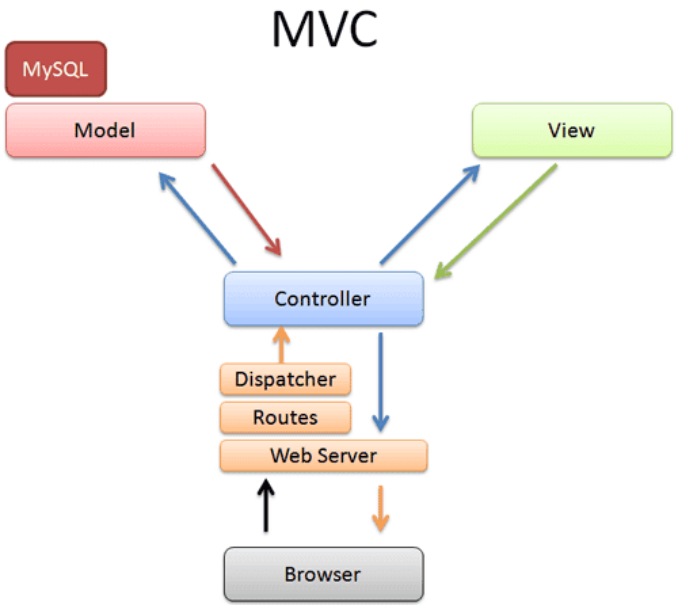
One of the most well-known design patterns, especially in the context of web development, is the **MVC pattern—Model-View-Controller.**



MVC model

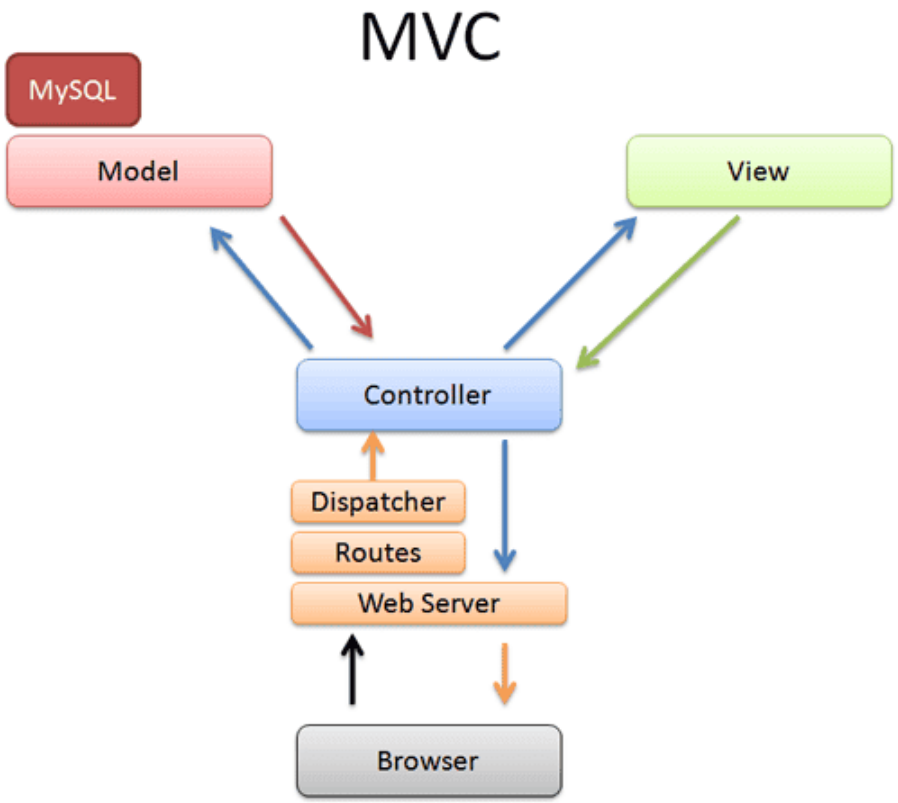
MVC is divided into three interconnected parts, and each part has its own specific responsibility, independent of the others. The names of the three components are:

1. **Model (data):** Manages and processes data.
2. **View (interface):** Displays data to the user.
3. **Controller (controller):** Controls the interaction between the Model and View.



MVC Data Flow

1. **Client** sends a request to the Controller.
2. The Controller processes the input and interacts with the Model.
3. The Model returns the **data** to the Controller.
4. The Controller sends the **data** to the View.
5. The View displays the data to the user.



Controller

Function:

- Handles data storage, manipulation, and business rules.
- Could be databases, API data, or even in-memory data structures.

```
const carModel = require('./carModel');
const carController = {
    getCars(req, res) {
        // Retrieve the list of cars from the Model
        const cars = carModel.getCars();

        // Render the View (carList.handlebars) with the data
        res.render('carList', { cars });
    }
};
module.exports = carController;
```

Model

Function:

- Handles data storage, manipulation, and business rules.
- Could be databases, API data, or even in-memory data structures.

```
const carModel = {
    cars: [
        { name: 'Coupe Maserati', clickCount: 0 },
        { name: 'Camaro SS 1LE', clickCount: 0 }
    ],
    getCars = () => this.cars,
    incrementClick(carName) {
        const car = this.cars.find(c => c.name === carName);
        if (car) { car.clickCount += 1; }
    }
};
module.exports = carModel;
```

View

Function:

- Responsible for rendering data to the user.
- Does not directly interact with the Model, it displays the data provided by the Controller.
  - Example Handlebars template: **`carList.handlebars`**

```
<ul>
    {{#each cars}}
    <li>
        <strong>{{this.name}}</strong> - Click Count: {{this.clickCount}}
    </li>
    {{/each}}
</ul>
```

Advantages of MVC

- **Separation of Concerns:**

Clear division between business logic (Model), UI (View), and input handling (Controller).

- **Parallel Development:**

Multiple developers can work on Model, View, and Controller simultaneously.

- **Easier to Maintain and Test:**

Isolating components makes debugging and testing simpler.

Why Use MVC?

- **Faster Development:**

Encourages parallel development.

- **Multiple Views:**

Same Model can be represented through different Views.

- **Supports Asynchronous Techniques:**

Works well with JavaScript frameworks and AJAX for faster loading.

MVC Conclusion

- MVC is a powerful architecture that separates logic, UI, and data management.
- It's widely adopted in modern web development frameworks.
- Ideal for larger projects with complex business logic and multiple views.

Node.js Crypto Module

Encryption example

- The `crypto` module is a core module (no need to install it).
- Import and use this module to encrypt some text:

```
const crypto = require('crypto');
// for aes-128-ecb, key length is 16 bytes
const key = 'mypassword123456';
const cipher = crypto.createCipheriv('aes-128-ecb', key, null);
let encrypted = cipher.update('my secret', 'utf8', 'base64');
encrypted += cipher.final('base64');
console.log(encrypted);
```

Decryption example

- To decrypt the text, you need the ciphertext and the original key (password):

```
const crypto = require('crypto');
// for aes-128-ecb, key length is 16 bytes
const key = 'mypassword123456';
const decipher = crypto.createDecipheriv('aes-128-ecb', key, null);
let decrypted = decipher.update('8Cx6EsM58Suj6jSIdlogLQ==', 'base64', 'utf8');
decrypted += decipher.final('utf8');
console.log(decrypted);
```

- There are many other encryption algorithms and modes. This `eas-128-ecb` algorithm is probably the easiest to use (since it doesn't require an IV).