CWEB280 -wk2

# Recommended Quiz: 20 minutes

Review : <https://developer.mozilla.org/en-US/docs/Learn/Server-side/First_steps/Introduction>

In your own words try an explain the difference between Static Sites and Dynamic Sites

# Recommended Tools: ESLint

Webstorm and ESLint can enforce a coding style for all our JS files and can fix/ point out issues like extra spaces or missing semi colons

Install es lint – type the following command in the terminal (cweb2021\lo1serveronly folder)

npm i eslint

node node\_modules\eslint\bin\eslint –init

Follow the Prompts and select the bolded option

? How would you like to use ESLint? ...

To check syntax only

To check syntax and find problems

**> To check syntax, find problems, and enforce code style**

? What type of modules does your project use? ...

JavaScript modules (import/export)

**> CommonJS (require/exports)**

None of these

? Which framework does your project use? ...

React

Vue.js

**> None of these**

? Does your project use TypeScript? » **No** / Yes

? Where does your code run? ... (Press <space> to select, <a> to toggle all, <i> to invert selection)

Browser

**√ Node**

? How would you like to define a style for your project? ...

**> Use a popular style guide**

Answer questions about your style

Inspect your JavaScript file(s)

? Which style guide do you want to follow? ...

Airbnb: https://github.com/airbnb/javascript

Standard: https://github.com/standard/standard

**> Google: https://github.com/google/eslint-config-google**

XO: https://github.com/xojs/eslint-config-xo

? What format do you want your config file to be in? ...

JavaScript

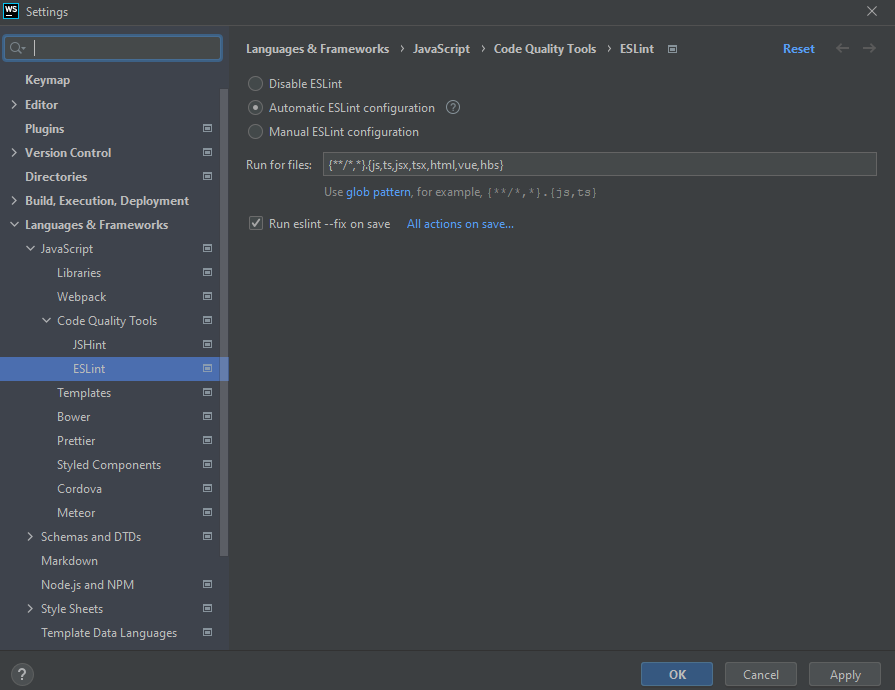
YAML

**> JSON**

eslint-config-google@latest eslint@>=5.16.0

? Would you like to install them now with npm? No / **Yes**

In Webstorm – Open the settings menu Ctrl+Alt+S and find the ESLint settings and set them to the picture below – select Auto ESLint configuration and check ‘Run eslint –fix on save’



No when you save a file Webstorm will automatically run ESLint on you code and fix any issues it can find.

# Template Engines in Express

Express can support several types of template engine. A template engine works in a rather simple manner:

* create a template (with the appropriate syntax) example {[title}}
* create an appropriate route to render the template,
* send “options” (or parameters) to the template that correspond to the placeholders in the template

## Handlebars

In this course we use the **Handlebars** template engine:

Learn more about Handlebars Syntax: <https://handlebarsjs.com/guide/>

Handlebars is an enhanced version of Mustache template engine. Handlebars allow for some built in helpers where as Moustache does not all any logic (no Conditional, Loops or variable declaration)

## PUG

Probably the most popular template engine is called PUG (formerly Jade)

PUG allows for logic block in the template (variable declaration, conditionals, loops etc)

PUG syntax does not use standard html tags - *example: p= 'Hello World' instead of <p>Hello World</p>*

Learn more about PUG syntax: <https://pugjs.org/language/code.html>

## Partials

Template engines also allow for reusable blocks called partials. Partials can place content on multiple templates, so we do not have to include the same mark up in all templates.

An alternative to partials is the \views\layouts.hbs file in the lo1serveronly project. This file is used as a wrapper to all the templates in the view folder

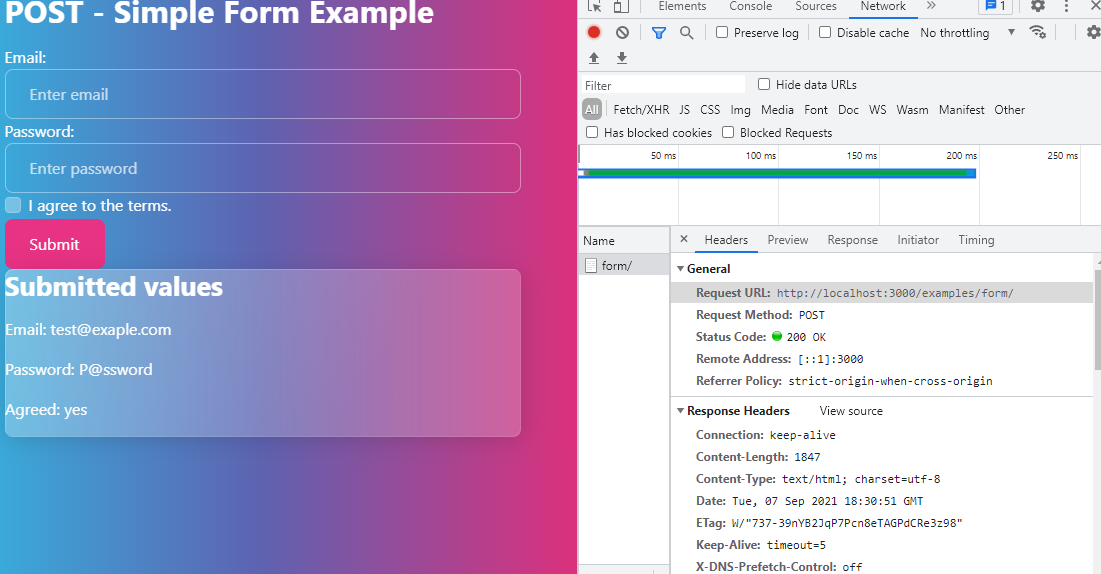
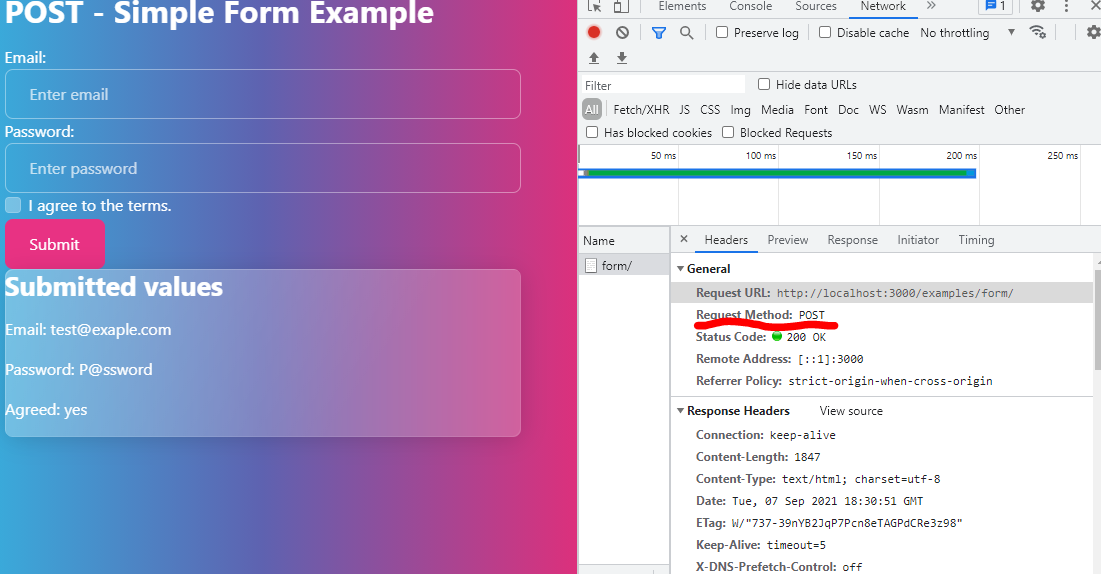
Learn more about popular template engines:   
<https://blog.logrocket.com/top-express-js-template-engines-for-dynamic-html-pages/>

# HTML Forms to interact with the Server

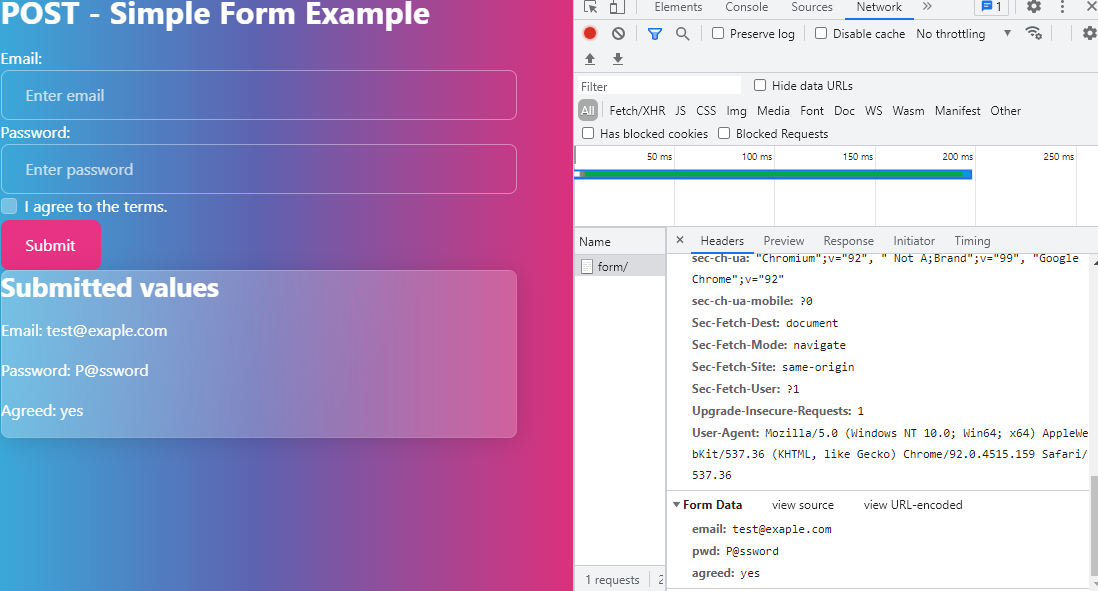
One of the ways users can interact with websites is to submit html forms. The user submits a form and the browser encodes the data in a way that the server can interpret.

Developers can then use the user submitted data to adjust the response to the request.

Form data usually send use the POST request method. To see the browser submitting the form to the server open the developer tools. In the Network tab -> select the file and examine the “Headers” section



If you scroll down further in the window you will notice the **Form Data**



# Create an Interactive Form

Create a new file in the view folder call it **form-example.hbs**

**IMPORTANT: the input tag MUST have a name attribute for the form to work**

**\view\form-example.js – add the following code to**

<form action="/examples/form/" method="post" >

<h1>{{ title }}</h1>

<div class="form-group">

<label for="email">Email:</label>

<!--IMPORTANT the input tag MUST have a name attribute to post to the server ie. name="email"-->

<input type="email" class="form-control" placeholder="Enter email" name="email" id="email" required/>

</div>

<div class="form-group">

<label for="pwd">Password:</label>

<input type="password" class="form-control" placeholder="Enter password" name="pwd" id="pwd" required/>

</div>

<div class="form-group form-check">

<label class="form-check-label">

<input class="form-check-input" type="checkbox" name="agreed" value="yes" required/> I agree to the terms.

</label>

</div>

<button type="submit" class="btn btn-primary">Submit</button>

</form>

{{#if isSubmitted}}

<div class="card p-2">

<h2>Submitted values</h2>

<p>Email: {{submittedEmail}}</p>

<p>Password: {{submittedPassword}}</p>

<p>Agreed: {{submittedAgreed}}</p>

</div>

{{/if}}

Notice that the form method="post" – otherwise the browser will submit the form as part of the URL query string.

**\routes\examples.js – add the following code above the last line that reads -- module.exports = router;**

/\* GET content for path: http://localhost:3000/examples/form \*/

router.get('/form', function(req, res, next) {

res.render('form-example', {

title: 'GET - Simple Form Example',

});

});

/\* POST submit form data to path : http://localhost:3000/examples/form \*/

router.post('/form', function(req, res, next) {

res.render('form-example', {

title: 'POST - Simple Form Example',

isSubmitted: req.body.agreed === 'yes', // check to see if user checked the checkbox <input name="agreed" value="yes" />

// example <input name="someName" /> then we use req.body.someName access the user input value

submittedEmail: req.body.email, // <input name="email" /> then we use: req.body.email

submittedPassword: req.body.pwd, // <input name="pwd" /> then we use: req.body.pwd

submittedAgreed: req.body.agreed, // <input name="agreed" value="yes" /> then we use: req.body.agreed

});

});

Restart the webserver for the changes to take effect - Navigate to <http://localhost:3000/examples/form/>

# Query Strings to interact with the Server

Web browsers can also send data to the server using query strings in the address bar.

Example URL: <http://localhost:3000/examples/form/?email=t%40t.ca&pwd=SAGFSD&agreed=yes>

* Query strings are appended to the end of the website path and start with a ? question mark character
* Query Strings are split up into the name value pairs with the use of & - amperes and
* The name is on the left of the = equal sign and the value is in the right.

# Code support for Query Strings

Web browsers can also send data to the server using query strings in the address bar.

Example URL: <http://localhost:3000/examples/form/?email=t%40t.ca&pwd=SAGFSD&agreed=yes>

* Query strings are appended to the end of the website path and start with a ? question mark character
* Query Strings are split up into the name value pairs with the use of & - amperes and
* The name is on the left of the = equal sign and the value is in the right.