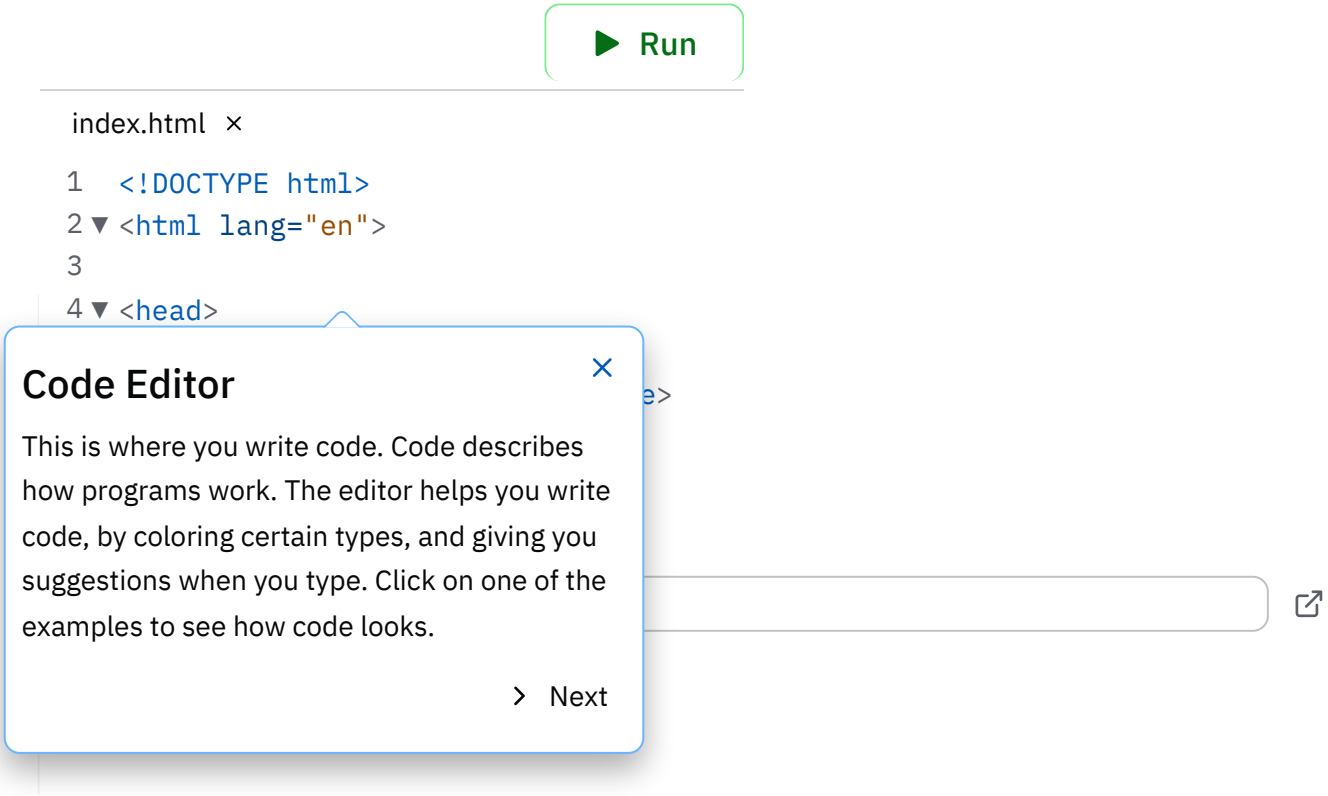
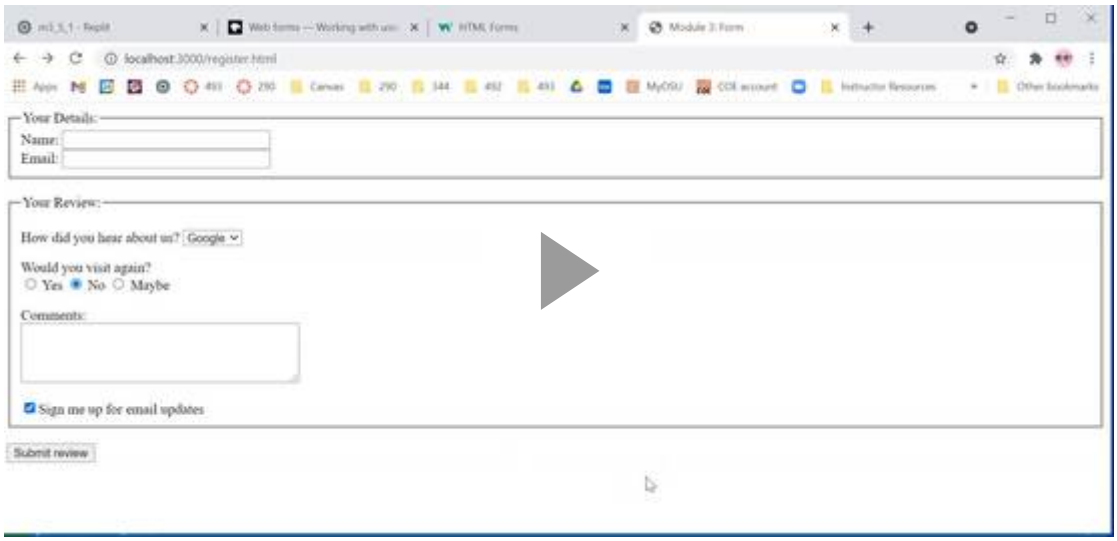
Exploration — HTML Forms

Introduction

In this section we will discuss HTML form elements, syntax, options, and best practices. HTML forms enable web applications to collect information from the user. Any time you are submitting data via a website you are using a form. If you are creating a new account at Amazon, logging into OSU's website or uploading a file, that is all happening with forms. So it is important to get a good grasp of what is going on on the frontend HTML side of things to submit data before we talk about actually processing that data on the server in the backend. Here is an example form that we will use to discuss the various parts of a form.



The form tag



0:00 / 17:11

We add a form to an HTML document using a `<form>` tag that must have an opening and a closing tag. In general, we can put any content inside a `<form>` tag except other forms. In addition to general content like paragraphs, headers or images forms almost always include `<input>` elements. These elements are where the user will actually type data or select options to submit. For multi-line text input we can specify the `<extarea>` control, which can be stretched vertically and horizontally by default.

If you look at the HTML of the above form you will see at the outermost level there is a `<form>` tag, then within that, there `<fieldset>`s for grouping form controls.

<form> attributes

The `<form>` tag itself has two very important attributes.

- The first is `action` which specifies where the request from the form should be sent.
  - When the form is submitted the browser will package up all the contents of the form and send it to the URL specified in the action.
  - If the `action` attribute is missing, then the form is submitted to the same URL from which it was downloaded.
  - The value of the `action` attribute can be a relative URL or an absolute URL. A relative URL is interpreted relative to the URL of the page containing the form.
    - When we want to submit the form to the same website from which the webpage containing the form was download, it is typical to use a relative URL.
- The second important attribute is `method` which specifies the HTTP method to be used in the HTTP request sent when the form is submitted.
  - The value of the attribute is case insensitive.

- If this attribute is omitted, it defaults to `GET`.
- The typical values are `GET` or `POST`.

<label>

Form controls include the <label> tag to help the user (especially those using a screen reader) to understand the purpose of their data entry. The for="" attribute is added so that it matches up with the form control's id="".

<input>

The `input` tag is the most commonly used tag to get user data within a form. It stands out a little from other elements because it has an attribute `type` that vastly changes how it is displayed, how it is used, and even what other attribute can be used. In addition to having the `type` attribute, it is also critical that it provide a `name` attribute. When the data submitted by the form is received by the server-side, the `name` attribute is how the server knows which part of the form each data item is associated with.

There are a lot of different kinds of inputs and attributes. We highly recommend **MDN's Input Reference (https://developer.mozilla.org/en-US/docs/Web/HTML/Element/input)** if you ever need an answer to "Is there an input type for this?" In addition, the MDN reference enumerates all the other possible attributes which can be used for things like limiting the length or do some types of validation.

Email

The HTML code for the email field in the above form is shown below:

```
<input type="email" name="email" size="30" maxlength="100" placeholder="name@host.com" />
```

The email field in our example is an `input` element with `email` as the value of the `type` attribute. The `name` attribute in this example is also `email`. If the user entered the value `nauman@example.com` in this field and submitted the form, the request will contain the value `email=nauman@example.com`. The server can thus know what value the user entered in the field named `email` in the form. Note that in this example, the values of both the `name` attribute and the `type` attribute are `email`. But this is just a coincidence and there is no requirement that the values of the `name` and `type` attributes of an input element must match, as we can see in our next examples.

Checkboxes

Checkboxes (https://developer.mozilla.org/en-US/docs/Web/HTML/Element/input/checkbox) need both a `name` and `value` just like other inputs. However, the server will only ever know about the existence of the checkbox if it is checked.

Checkbox Example 1

The HTML code for the checkbox in the example we introduced earlier in the exploration is shown below.

```
<input type="checkbox" name="subscribe" checked />
```

If this checkbox is checked when the form is submitted, the request will contain the value `subscribe=checked`. If it is unchecked, the server will not known that the form contains this checkbox.

Checkbox Example 2

If a form contains the checkbox `<input type="checkbox" name="fruit" value="apple">` and the checkbox is checked when the form is submitted, then the request will contain the value `fruit=apple`. If it is unchecked the server will not get any indication that this checkbox even exists.

Radio Buttons

Radio buttons add on a little twist. If no radio button is selected, nothing will get sent, just like with a check box. But every radio button that has the same name will be in a radio button group. Within that radio button group, only one radio button can be selected. Radio buttons with different values for the name attributes belong to different groups and one radio button can be selected from each group.

Radio Button Example

In our example form, radio buttons are using in the “Would you visit again?” portion of the form, as is shown below.

```
<input type="radio" name="rating" id="yes" value="yes">
<label for="yes">Yes</label>
<input type="radio" name="rating" id="no" value="no">
<label for="no">No</label>
<input type="radio" name="rating" id="maybe" value="maybe">
<label for="maybe">Maybe</label>
```

All three radio buttons have the same name, i.e., `rating`, and thus these three radio buttons are all in the same group. So we can select only one of the 3 choices at a given time. The value selected at the time of submitting the form will be sent in the request with the name `rating`. For example, if the user had selected `maybe`, the request will contain the value `rating=maybe`.

<select> with <option>s

The HTML `select` element is used to provide the user a drop-down list of choices. Similar to the `input` element, the `name` attribute is used for the `select` element when sending the user choice in

the request. Within the `select` tag, `option` tags are used to define the choices in the drop-down list. The selected attribute will select the Other option by default, which the user can override.

Select Example

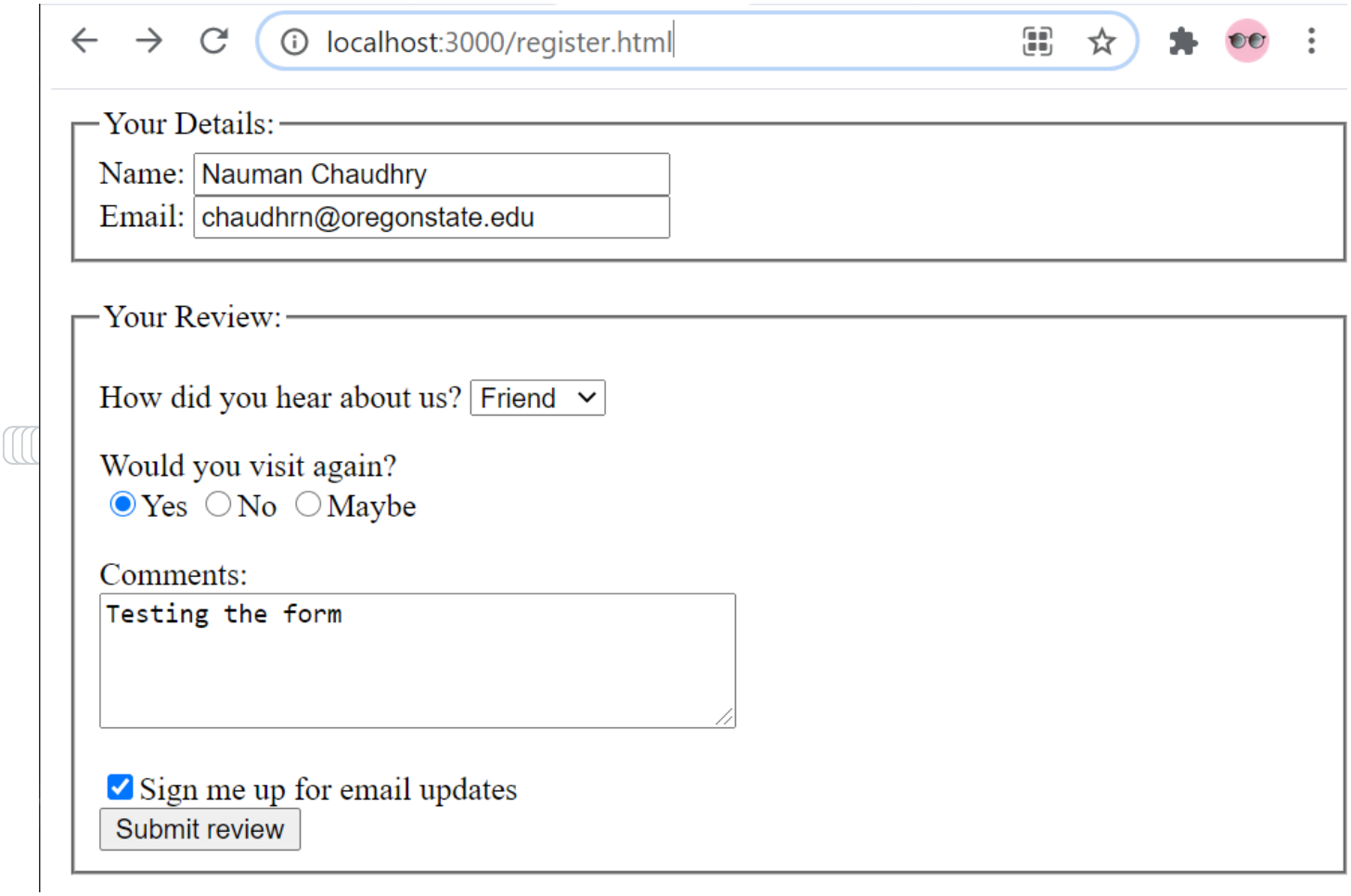
In our form example, the `select` tag is used in the section "How did you hear about us?" to present the user with 4 choices.

```
<select name="referrer" id="how">
   <option value="Google">Google</option>
   <option value="Friend">Friend</option>
   <option value="Advertisement">Advertisement</option>
   <option value="Other" selected >Other</option>
</select>
```

In this example, the option selected by the user will be sent in the request with the name `referrer`. For example, if the user had selected `advert`, the request will contain the value `referrer=advert`.

GET and POST

Sending HTTP requests using either the HTTP method `GET` or the HTTP method `POST` is the primary way for the browser to send data to the server, other than the data usually sent in the HTTP headers. When submitting standard HTML forms `GET` and `POST` are the only options for sending the form data to the server. It is important to understand the differences and to know when one would be preferred over another. To make the discussion concrete, consider that we deploy a web application at `http://localhost:3000` and fill out the example form with the values shown in the following figure:



GET

A GET request sends the data as key value pairs as parts of the URL. In our example form, if we set the value of the `method` attribute to `GET`, fill out the form as shown in the above figure and submit it, the URL will be as follows:

```
http://localhost:3000/review?name=Nauman+Chaudhry&email=chaudhrn%40oregonstate.edu&referrer=friend&rating=yes&comments=Testing+the+form&subscribe=on
```

The ? indicates the start of the query string. The query string is composed of key-value pairs, with an `=` between a key and its corresponding value, and an `&` separating the different key-value pairs. The keys are `name`, `email`, `referrer`, `rating`, `comments`, and `subscribe`. These keys correspond to the `name` attributes of the the `input` elements in the form. The value for each key is the value of the `input` element with that `name` attribute.

You may have noticed that in the URL the value for certain keys, e.g., `name` and `email`, is a little different from the value that we entered in the form. Specifically, the space in `Nauman Chaudhry` has been replaced by the `+` character, and the `@` character in the email has been replaced by `%40`. This is called **percent-encoding** or **URL encoding**, whereby certain special characters that have a specific meaning within URLs are **encoded with other characters (https://developer.mozilla.org/en-US/docs/Glossary/percent-encoding)**.

POST

A POST request will send the same data as GET. But instead of sending it as part of the URL, it is sent in the body of the request. That means that you cannot tell what data (if any) was sent to a server via a POST request by just looking at the URL.

GET vs POST

If we use GET, then the users will see the data as part of the URL, but the data is not displayed in the URL when using POST. This by itself does not mean that POST is secure. Unless we use the HTTPS protocol, which encrypts the HTTP communication, an intruder can see the body of a POST request. However, even when using HTTPS protocol, we must not use GET for submitting sensitive data, such as passwords. Many web servers log the URL of HTTP requests, and putting sensitive data in the URL with GET requests can result in that data being saved in log files. Another limitation of GET is that browsers often place a limitation on the length of the query string, whereas no such limits are placed on the body length.

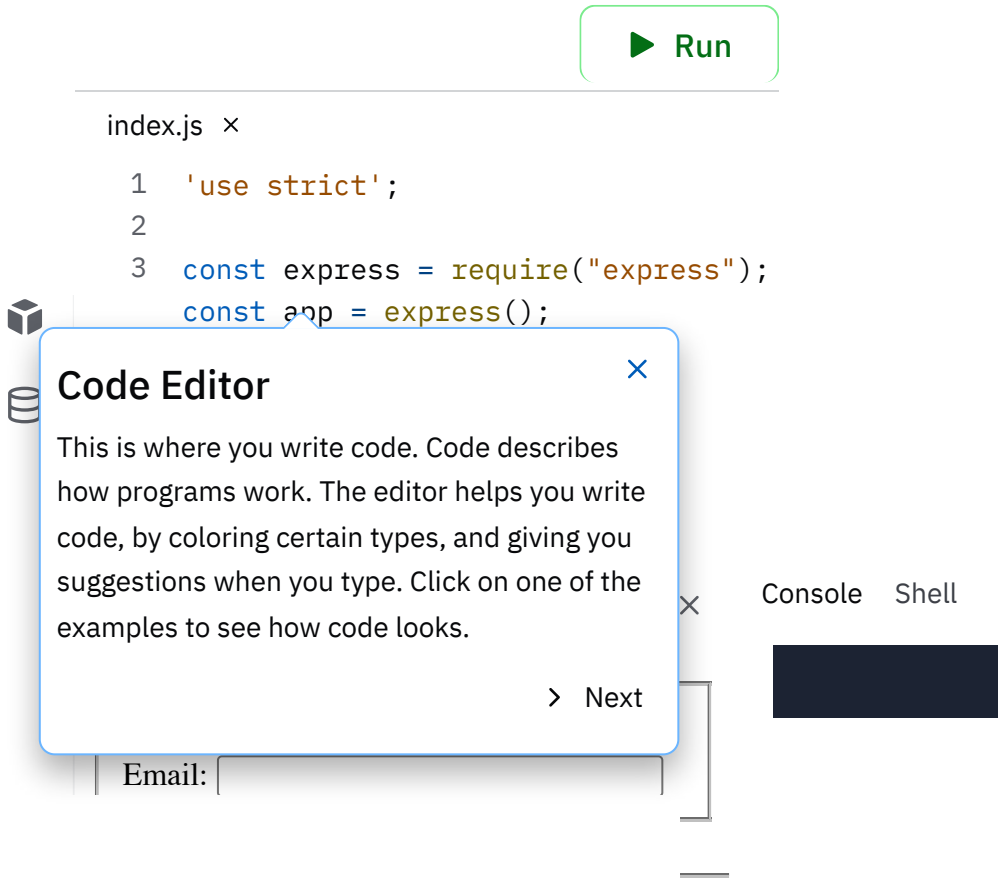
In general, GET is used for retrieving data from the server. So if the form is being used to conduct a search or simply retrieve data, and the restriction on the length of the query string will not be an issue, then GET is suitable. If the form is making any changes on the server or the restriction on the length of the query string can be an issue, then POST is suitable.

We will return to a discussion of GET vs. POST later in the course. For now, when in doubt submit forms using POST.

Processing Form Submission using Express

Processing a submitted form in Express requires adding a route for the URL and HTTP method specified in the form. Express then provides the data submitted in the form as a JSON object to our route handler function. Let us look at an example program that has routes for handling both GET and POST requests for our example form submitted to the URL `/review`. Note that there is no directory named `review` on our server. Instead, we have defined routes in our server file to handle requests for `/review`. The response for requests to this URL are dynamically generated by executing the route handlers provided for these routes.

> **Note:** Route handlers are typically written using anonymous functions. If you need to refresh your knowledge, review the section on Function Expressions in the **Exploration - Functions and Functional Programming (https://canvas.oregonstate.edu/courses/1879154/pages/exploration-functions-and-functional-programming)** in Module 2.



GET requests

In our example, we define a route for a GET request to the URL `/review` as follows:

```
app.get("/review", (req, res) => {
    console.log(req.query);
    res.send(req.query);
});
```

For GET requests, Express creates an object containing the data submitted in the form and sets it as the `query` property of the request argument of the route handler function. In the route handler shown above, we are using **the send method on the response object (https://expressjs.com/en/api.html#res.send)** to simply send back the `query` object in the response.

POST requests

In our example, we define a route for a POST request to the URL `/review` as follows:

```
app.post("/review", (req, res) => {
    console.log(req.body);
    res.send(req.body);
});
```

For POST requests, Express creates an object containing the data submitted in the form and sets it as the `body` property of the request argument of the route handler function. In the route handler shown above, we are sending back this `body` object in the response.

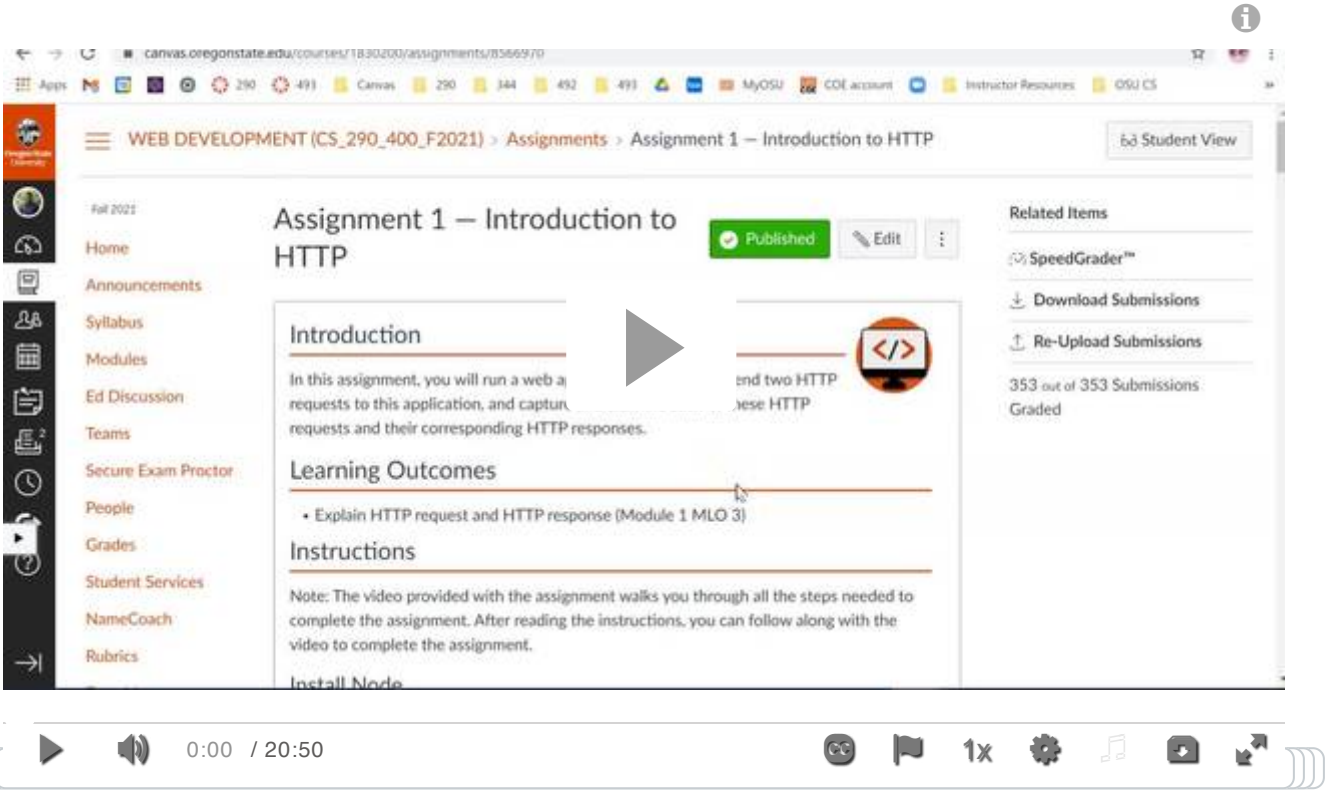
However, note that in order to parse the data in the body of a POST request and set the `body` property of the request object, Express requires us to do one more thing. We need to add the following statement to parse the data in the HTTP body.

```
app.use(express.urlencoded({
    extended: true
}));
```

In a later module, we will take a deeper look at Express and we will discuss this statement in more detail. For now, just remember to have this statement before any routes in your program.

Example: Forms in Assignment 1

When working on Assignment 1, we didn't look at the code of the web app and simply ran the web app to see the HTTP requests and responses using Dev Tools. But now that we have learned about HTML form, JavaScript, Node and Express, let's take a look at the files that are in the Assignment 1 web app. For reference, here is the file **assignment1_app.zip (https://canvas.oregonstate.edu/courses/1879154/files/93831996?wrap=1)** .



> **Tip:** As described in the above video, we can use the npm package **nodemon (https://www.npmjs.com/package/nodemon)** to automatically restart our Node app whenever we

> make a change to our JavaScript code.

Summary

In this exploration, we studied HTML forms and how to write Express apps that can process forms. We looked at the two most important attributes of a form, namely `action` and `method`. We studied some common input elements used in HTML forms. We then looked at how we can write route handlers in our Express applications to process requests generated by submitting HTML forms. The important thing to remember here is that

- When the form action is GET, the form data is sent as query parameters in the request and is available in our Express app in the object `request.query`.
- When the form action is POST, the form data is sent in the body of the request and is available to our Express app in the object `request.body`. Furthermore, to parse the form data in the body of a POST request, our Express app must include the statement `app.use(express.urlencoded({extended: true}));` before any route handlers.

Additional Resources

Here are some references to learn more about the topics we discussed in this exploration.

- MDN's **discussion of forms (https://developer.mozilla.org/en-US/docs/Web/HTML/Element/form)**.
- Discussion of **input element on MDN (https://developer.mozilla.org/en-US/docs/Web/HTML/Element/input)**.
- Percent-encoding, also called URL encoding, is described **here (https://developer.mozilla.org/en-US/docs/Glossary/percent-encoding)**.