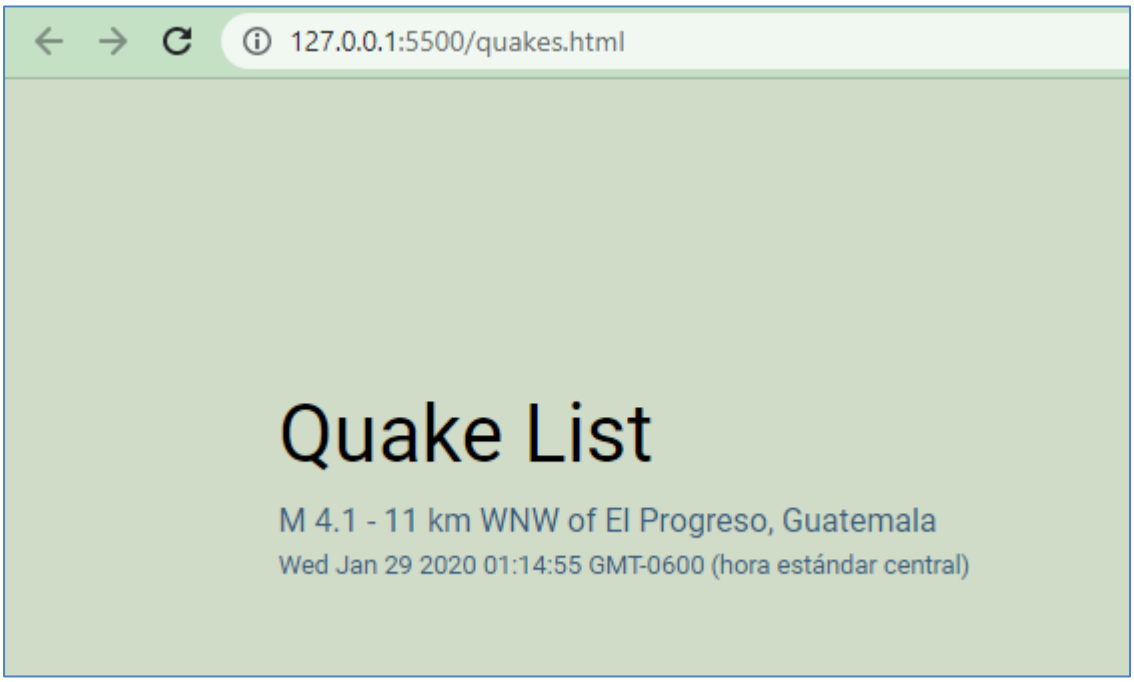
WDD 330 Web Frontend Development II
Samuel Palacios
Week 10
June 24th, 2022

Readings Assignment Notes

1. Quakes Near Me



2. MDN: Validating forms

- ✓ **The validation** of the data entered by the user is essential for any application because it promotes consistency and prevents the user from making mistakes. There are 2 types of validations, on the **client side** (**form validation**) with everything that can be done on a form, and on the **server side** (**input sanitization**). By default, it is necessary to do both, since the first one can be deactivated through an attribute in the form control called validate, which can be set to false and would disable all validations.

- ✓ **Form-level validations** can be grouped into 4 types: if required, string length or element size, value range, and input pattern. If it is required or not, it indicates whether the data is mandatory or not (for example, in a form the name is mandatory but the nickname is optional), the string length or size will depend on the input (an input cannot be more than 2 characters maximum) , as well as the range of values (example, only positive values are accepted), and the input pattern can be defined with a specific pattern or by input type (an input type email will not allow strings other than user@server .domain).

```
48
49  <fieldset>
50      <legend>Personal Information</legend>
51      <label class="top">Full Name*<input type="text" name="fullName" pattern="[A-Za-z ]{5,}" required></labe
52      <label class="top">Phone Number*<input type="tel" name="phoneNumber" placeholder="(001) (1234567)" requ
53      <label class="top">Zip Code* <input type="number" name="zipCode" required></label>
54      <label class="top">Email<input type="email" name="email" placeholder="username@servername.dom"></label>
55  </fieldset>
56
```

- ✓ At the **CSS** level, the user can also be shown when an input is not **valid** or something is missing, with this our applications will be more intuitive.

```
.top input:required {
    border-left: 5px solid red;
}

.top input:required:valid {
    border-left: 5px solid green;
}
```

- ✓ The first type is implemented with the **"required"** attribute, the next with the **"minlength, maxlength"** attributes, the other with the **"min-max"** attributes, and the last with the **"pattern"** or **"type"** attribute.

- ✓ It can also be validated using JavaScript using the "constraint validation API". You can dynamically and programmatically add or remove any validation at the form level, and even perform more custom actions because it's with JS.

```
const email = document.getElementById("mail");

email.addEventListener("input", function (event) {
  if (email.validity.typeMismatch) {
    email.setCustomValidity("I am expecting an e-mail address!");
    email.reportValidity();
  } else {
    email.setCustomValidity("");
  }
});
```

3. MDN: Using Fetch

- ✓ This API is the most used from JS to make **synchronous and asynchronous requests** to server-side resources. This API has a wide variety of options and parameters, to be

able to consume any service or resource on the server side and in the conditions that are required.

```
// Example POST method implementation:
async function postData(url = '', data = {}) {
    // Default options are marked with *
    const response = await fetch(url, {
      method: 'POST', // *GET, POST, PUT, DELETE, etc.
      mode: 'cors', // no-cors, *cors, same-origin
      cache: 'no-cache', // *default, no-cache, reload, force-cache, only-if-cached
      credentials: 'same-origin', // include, *same-origin, omit
      headers: {
        'Content-Type': 'application/json'
        // 'Content-Type': 'application/x-www-form-urlencoded',
      },
      redirect: 'follow', // manual, *follow, error
      referrerPolicy: 'no-referrer', // no-referrer, *no-referrer-when-downgrade, origin, or
      body: JSON.stringify(data) // body data type must match "Content-Type" header
    });
    return response.json(); // parses JSON response into native JavaScript objects
  }

  postData('https://example.com/answer', { answer: 42 })
    .then(data => {
      console.log(data); // JSON data parsed by `data.json()` call
    });
```

- ✓ Usually when this API is used and some resource is consumed on the server side, it is done through a GET or POST and sending parameters, but complete, binary or text files can also be sent, such as **JSON files**, and not just 1 to the same time if not multiple, etc.

```
fetch('https://example.com/profile', {
  method: 'POST', // or 'PUT'
  headers: {
    'Content-Type': 'application/json',
  },
  body: JSON.stringify(data),
})
.then(response => response.json())
.then(data => {
  console.log('Success:', data);
})
.catch((error) => {
  console.error('Error:', error);
});
```

```
const fileField = document.querySelector('input[type="file"]');

formData.append('username', 'abc123');
formData.append('avatar', fileField.files[0]);

fetch('https://example.com/profile/avatar', {
  method: 'PUT',
  body: formData
})
.then(response => response.json())
.then(result => {
  console.log('Success:', result);
})
.catch(error => {
  console.error('Error:', error);
});
```

```
const photos = document.querySelector('input[type="file"][multiple]');

formData.append('title', 'My Vegas Vacation');
for (let i = 0; i < photos.files.length; i++) {
  formData.append(`photos_${i}`, photos.files[i]);
}

fetch('https://example.com/posts', {
  method: 'POST',
  body: formData,
})
.then(response => response.json())
.then(result => {
  console.log('Success:', result);
})
.catch(error => {
  console.error('Error:', error);
});
```

- ✓ One cool thing about this API is that you can define an action when the resource consumption **succeeds or fails**. This is important, when an application is deployed in a client or browser, the connectivity or anything in between may fail and the server-side resource consumption may not be successful, so our application must inform the user of this, and define an action to overcome that limitation or try later.

```
fetch('flowers.jpg')
  .then(response => {
    if (!response.ok) {
      throw new Error('Network response was not OK');
    }
    return response.blob();
  })
  .then(myBlob => {
    myImage.src = URL.createObjectURL(myBlob);
  })
  .catch(error => {
    console.error('There has been a problem with your fetch operation:', error);
  });
```

- ✓ Just as a kind of **REQUEST object** is built to be able to consume a server-side resource with the FETCH API or function, it is also possible to obtain the **RESPONSE object** that brings everything necessary to obtain the response sent from the server.

```
const myBody = new Blob();

addEventListener('fetch', function(event) {
  // ServiceWorker intercepting a fetch
  event.respondWith(
    new Response(myBody, {
      headers: { 'Content-Type': 'text/plain' }
    })
  );
});
```