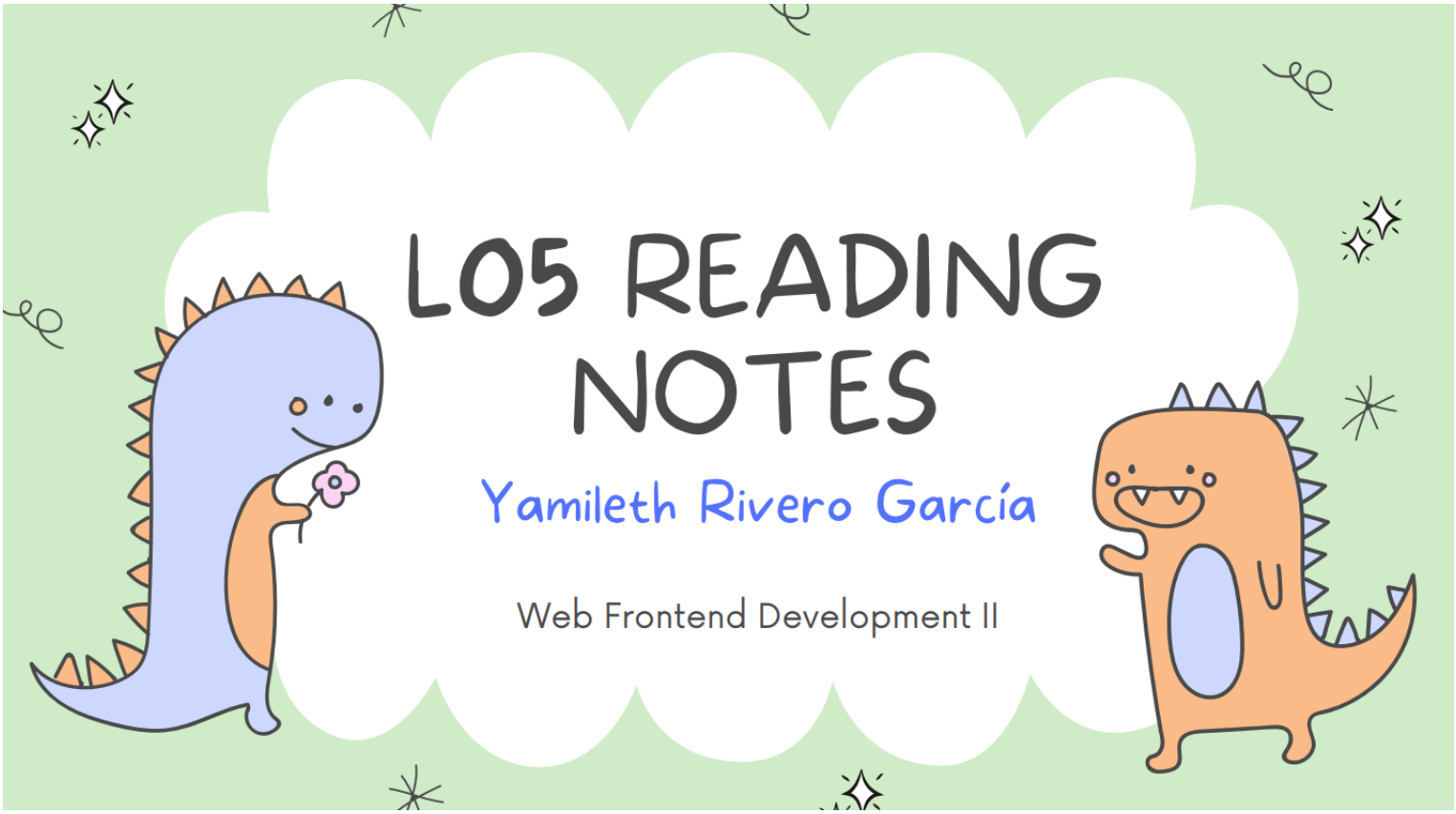

1.- CH13: AJAX

Ajax is a technique that allows web pages to communicate asynchronously with a server.

It dynamically updates web pages without reloading

IMPORTANT: This enables data to be sent and received in the background, as well as portions of a page to be updated in response to user events, while the rest of the program continues to run.

The use of Ajax revolutionized how websites worked

- Web pages were no longer static, but dynamic applications.

Clients and Servers

A client, such as a web browser, will request a resource (usually a web page) from a server, which processes the request and sends back a response to the client.

- Ajax allows JavaScript to request resources from a server on behalf of the client.
- A server is required when requesting resources using Ajax

A Brief History of AJAX

1990s

- Web pages contained static content
- Changes to the content on the page required a full page reload
- Screen going blank while new pages loaded

1999

- Microsoft implemented the XMLHTTP ActiveX control in Internet Explorer 5
- Allowed data to be sent asynchronously in the background using JavaScript

2004

- Asynchronous loading techniques started to be noticed
- Google used asynchronous loading techniques to enhance the user experience by changing the parts of the page without a full refresh

2005

- The term **'Ajax'** was coined by Jesse James Garrett
- **Ajax** was a neat acronym that referred to the different parts of the process being used: Asynchronous JavaScript and XML

Asynchronous: When a request for data is sent, the program doesn't have to stop and wait for the response

When the term **Ajax** was originally coined, XML documents were often used to return data. Many different types of data can be sent, but by far the most commonly used in Ajax nowadays is **JSON.**

After the publication of Garrett's article, **Ajax** use really started to take off.

Now users could see new content on web pages without having to refresh the page.

Today, it's unusual for Ajax not to be used when a partial web page update is required.

The Fetch API

Is currently a living standard for requesting and sending data asynchronously across a network.

- Uses promises to avoid callback hell

Different interfaces that it uses:

Basic Usage

The Fetch API provides a global **fetch()** method that only has one mandatory argument, which is the URL of the resource you wish to fetch.

Example:

```
fetch('https://example.com/data')
.then( // code that handles the response )
.catch( // code that runs if the server returns an error )
```

We can also use a **catch** statement at the end to deal with any errors that may occur.

Response Interface

Deals with the object that's returned when the promise is fulfilled.

Each response object has an **ok** property that checks to see if the response is successful.

Some other properties of the Response object are:

- `headers` – A Headers object (see later section) containing any headers associated with the response
- `url` – A string containing the URL of response
- `redirected` – A boolean value that specifies if the response is the result of a redirect
- `type` – A string value of 'basic', 'cors', 'error' or 'opaque'. A value of 'basic' is used for a response from the same domain. A value of 'cors' means the data was received from a valid cross-origin request from a different domain. A value of 'opaque' is used for a response received from 'no-cors' request from another domain, which means access to the data will be severely restricted. A value of 'error' is used when a network error occurs.

Redirects

The **redirect()** method can be used to redirect to another URL. It creates a new promise that resolves to the response from the redirected URL

EXAMPLE:

```
fetch(url)
.then( response => response.redirect(newURL)); // redirects to another URL
.then( // do something else )
.catch( error => console.log('There was an error: ', error))
```

Text Responses

The **text()** method takes a stream of text from the response, reads it to completion and then returns a promise that resolves to a USVSting object that can be treated as a string in JavaScript.

Example:

```
fetch(url)
.then( response => response.text() ); // transforms the text stream into a JavaScript
.then( text => console.log(text) )
.catch( error => console.log('There was an error: ', error))
```

File Responses

The **blob()** method is used to read a file of raw data, such as an image or a spreadsheet. Once it has read the whole file, it returns a promise that resolves with a **blob** object.

Example:

```
fetch(url)
.then( response => response.blob() ); // transforms the data into a blob object
.then( blob => console.log(blob.type) )
.catch( error => console.log('There was an error: ', error))
```

JSON Responses

The **json()** method is used to deal with these by transforming a stream of JSON data into a promise that resolves to a JavaScript object.

Example:

```
fetch(url)
.then( response => response.json() ); // transforms the JSON data into a JavaScript
.then( data => console.log(Object.entries(data)) )
.catch( error => console.log('There was an error: ', error))
```