WEEK-09 DAY-3
AJAX

AJAX

The objective of this lesson is for you to know how AJAX works. This lesson is relevant because AJAX is a foundational component of how modern web applications work.

When you are done, you should be able to do the following.

- Explain what an AJAX request is
- Identifying the advantages of using an AJAX request.
- Identify what the acronym AJAX means and how it relates to modern Web programming
- Describe the different steps in an AJAX request/response cycle
- Fully use the `fetch` API to make dynamic Web pages without refreshing the page lesson
  is relevant because AJAX is a foundational component of how modern web applications work.

When you are done, you should be able to do the following.

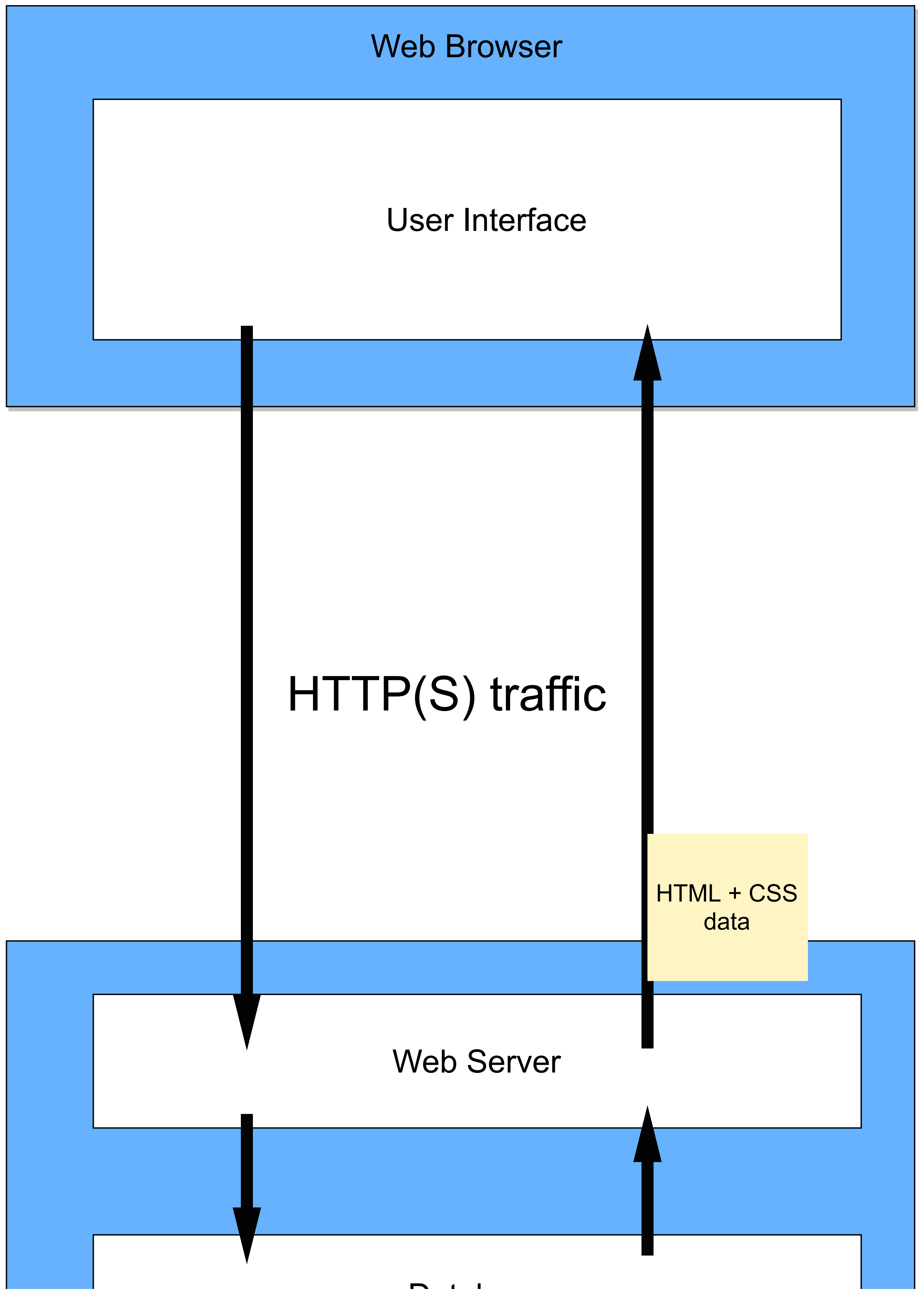
- Explain what an AJAX request is
- Identifying the advantages of using an AJAX request.
- Identify what the acronym AJAX means and how it relates to modern Web programming
- Describe the different steps in an AJAX request/response cycle
- Fully use the `fetch` API to make dynamic Web pages without refreshing the page

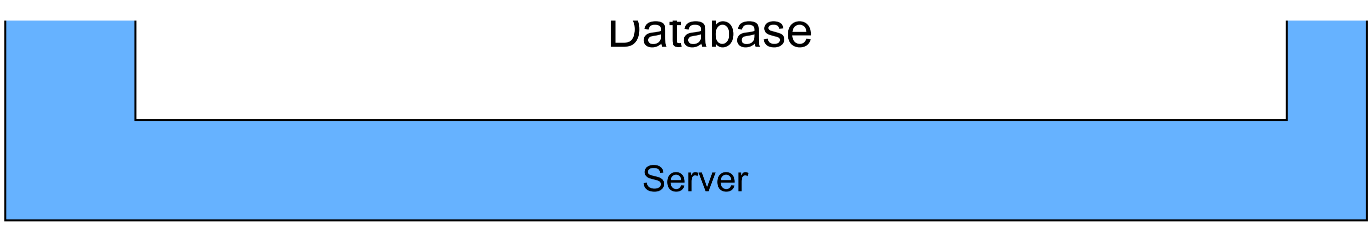
AJAX: Paving the Road to the Modern Web

In this lesson, you'll learn about AJAX (Asynchronous JavaScript and XML), which is a set of technologies that allow for efficient client (web browser) and server interaction. You'll first learn about the pre-AJAX approach that required full page reloads whenever the client and server interacted. Then, you'll get an overview of how AJAX works to make the client-server interaction much more efficient.

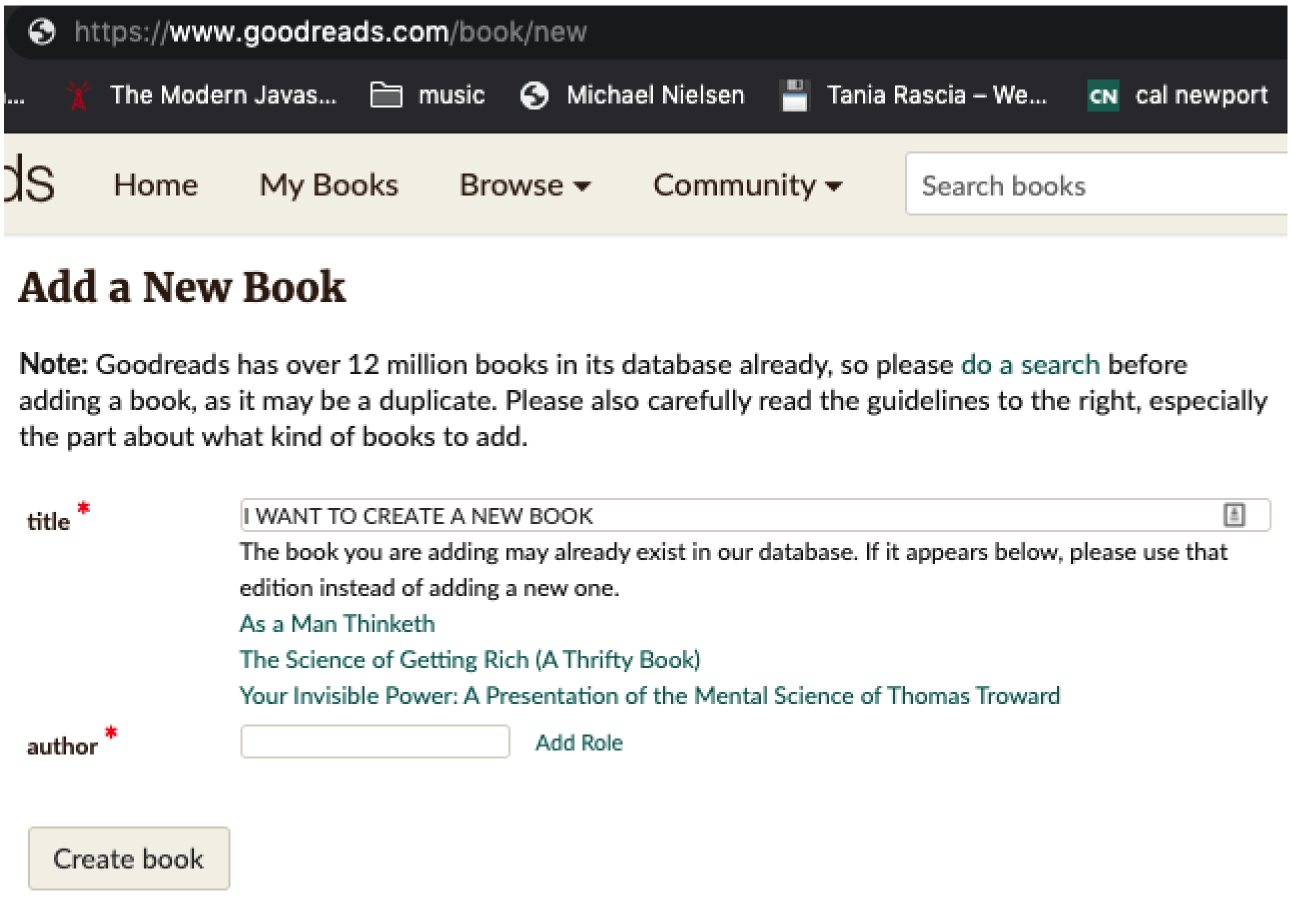
Classic Full Page Reloads

In the early days of the Web, most websites were fairly simple. When you land on a website, the web browser would make requests to a server, and the server would send back an entire HTML page for the browser to load:


Web Browser
User Interface
HTTP(S) traffic
HTML + CSS data
Web Server



This made sense for simple features. For example, let's say you were on a website like Goodreads and wanted to create a new book:



Add a New Book

Note: Goodreads has over 12 million books in its database already, so please do a search before adding a book, as it may be a duplicate. Please also carefully read the guidelines to the right, especially the part about what kind of books to add.

title * I WANT TO CREATE A NEW BOOK

The book you are adding may already exist in our database. If it appears below, please use that edition instead of adding a new one.

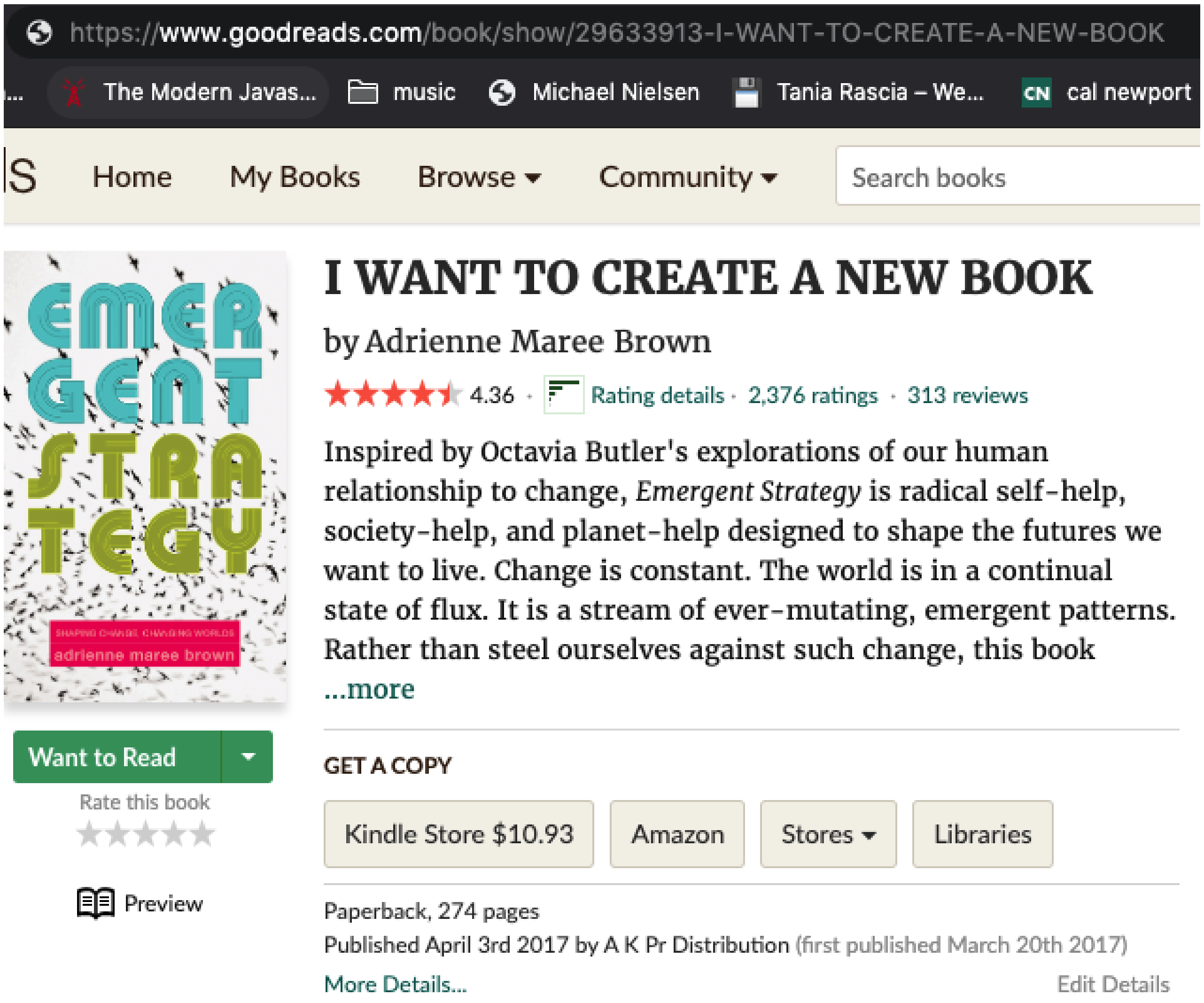
As a Man Thinketh

The Science of Getting Rich (A Thrifty Book)

Your Invisible Power: A Presentation of the Mental Science of Thomas Troward

author * Add Role

Once you submit this form, the server will add a new record to the database, and then redirect you to the page for that newly-created book:


https://www.goodreads.com/book/show/29633913-I-WANT-TO-CREATE-A-NEW-BOOK

I WANT TO CREATE A NEW BOOK

by Adrienne Maree Brown

4.36 · Rating details · 2,376 ratings · 313 reviews

Inspired by Octavia Butler's explorations of our human relationship to change, *Emergent Strategy* is radical self-help, society-help, and planet-help designed to shape the futures we want to live. Change is constant. The world is in a continual state of flux. It is a stream of ever-mutating, emergent patterns. Rather than steel ourselves against such change, this book ...more

Paperback, 274 pages

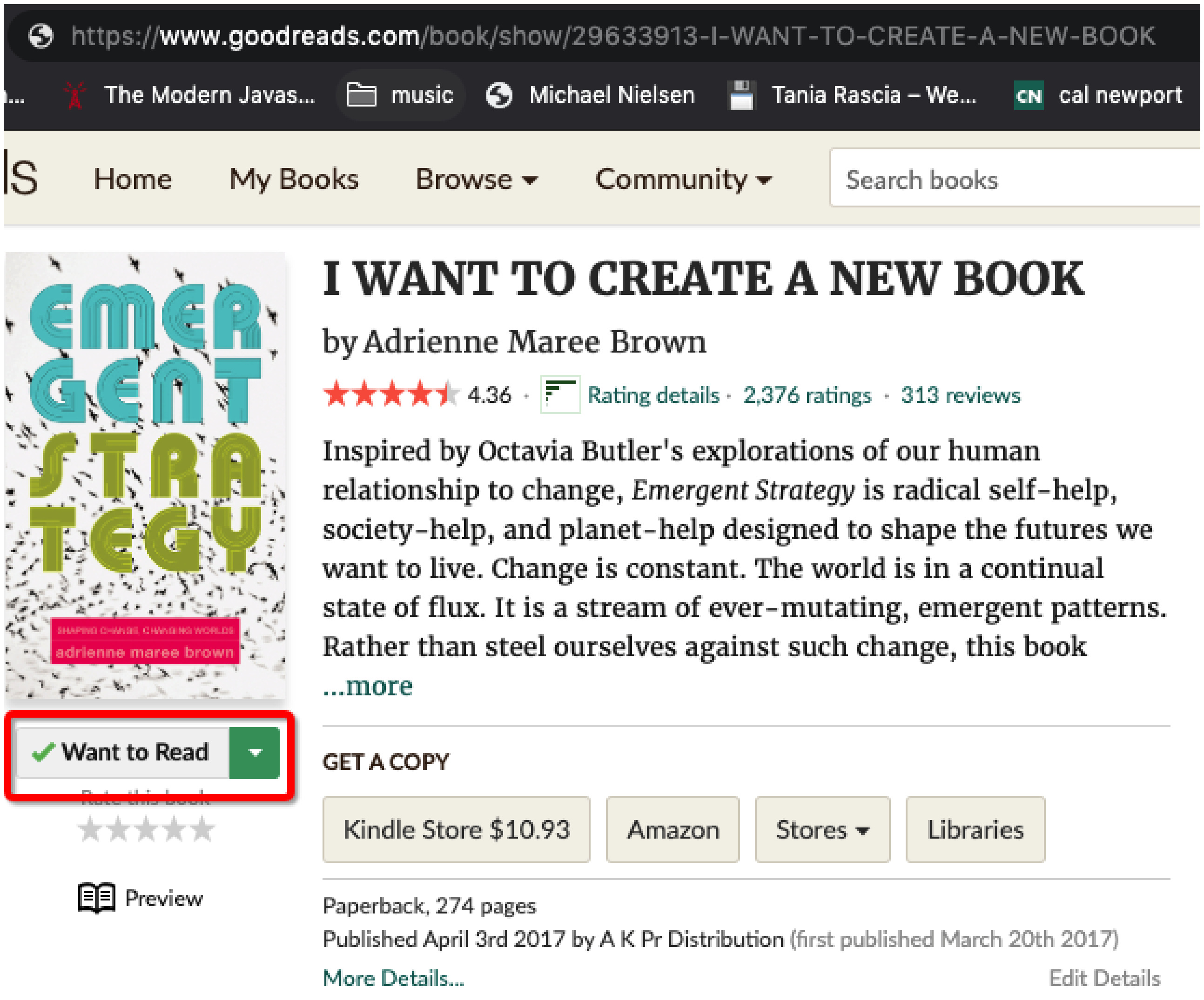
Published April 3rd 2017 by A K Pr Distribution (first published March 20th 2017)

More Details... Edit Details

When you arrive on that page, you get an entirely new HTML page from the server, and your browser loads it up.

Back in the early days of the Web, this kind of flow sufficed. However, as websites progressively got more interactive, this approach became inefficient and insufficient. For example, let's imagine that you're on that book's show page above, and all you wanted to do was hit the `Want to Read` button. Using the old approach, here's what would happen:

1. You submit a request to the server to mark that specific book as "Want To Read".
2. The server would make the necessary changes to the database.
3. The server prepares the entire HTML page for this book, except this time it would render 'Want To Read' under the book image in that HTML document so that the user can see the updated status of the book.
4. Finally, your browser would load up the entire newly-received HTML document just to show the change in status:


https://www.goodreads.com/book/show/29633913-I-WANT-TO-CREATE-A-NEW-BOOK

I WANT TO CREATE A NEW BOOK

by Adrienne Maree Brown

4.36 · Rating details · 2,376 ratings · 313 reviews

Inspired by Octavia Butler's explorations of our human relationship to change, *Emergent Strategy* is radical self-help, society-help, and planet-help designed to shape the futures we want to live. Change is constant. The world is in a continual state of flux. It is a stream of ever-mutating, emergent patterns. Rather than steel ourselves against such change, this book ...more

Paperback, 274 pages
Published April 3rd 2017 by A K Pr Distribution (first published March 20th 2017)

AJAX at a high level

You'll get a more in-depth explanation of how AJAX works in the next reading, but at a high level, AJAX is a group of different technologies that work together to allow a website to communicate with a server in the background without requiring the website to reload in order to display new changes.

Specifically, the key difference with AJAX is that when a change happens, the server is no longer responsible for updating the HTML and then sending the entire HTML document back. Instead, the server would send back data about the change, either in an XML or JSON format, and the website could then process that data and update the DOM accordingly.

Asynchronous JavaScript and XML

Let's use our Goodreads "Want to Read" example to break down the AJAX acronym.

Asynchronous

Using AJAX, when the user hits the "Want to Read" button, the updates would now happen asynchronously. In other words, the browser would interact with the server in the background without blocking any other events from happening on the web page.

JavaScript

JavaScript is the engine behind AJAX. When the user hits the "Want to Read" button, JavaScript is used to make the request to the server. When the data comes back from the server, JavaScript can also then be used to make the necessary updates to the DOM. In the next reading, we'll go more in-depth into the specifics of how JavaScript is used.

XML

The XML portion of this acronym is outdated. In the early days of AJAX, after successfully updating the book's status to "Want To Read", the server would send back data in XML format, like so:

```xml
<?xml version="1.0" encoding="UTF-8"?>
<book>
  <id>29633913</id>
  <status>Want To Read</status>
</book>
```

Nowadays, XML has been largely replaced by JSON, and in this course, you will almost always be dealing with a JSON response from a server that might look like this:

```json
{
  "book": {
    "id": 29633913,
    "status": "Want to Read"
  }
}
```

Notes on AJAX

As you can see, using AJAX requires a bit more complexity than the old client-server approach. The payoff is an improved user experience: updating just part of the page is almost always quicker than reloading the entire page.

Additionally, AJAX allows you to keep the user in their current context. For example, the user doesn't lose their current position on the page since there's no longer a full page reload.

Over time, JavaScript libraries have emerged that made using AJAX easier (i.e. jQuery, client side templating libraries, etc.) Eventually, AJAX led to the development of Single Page Applications, websites that have one and only one HTML page. You'll learn much more about Single Page Applications once you get to the React portion of this curriculum!

What you've learned

So far, you've mainly looked at AJAX from a high level, but at this point, you should have a clear understanding of:

- what AJAX is and the advantages of using AJAX
- what each letter of AJAX stands for and why

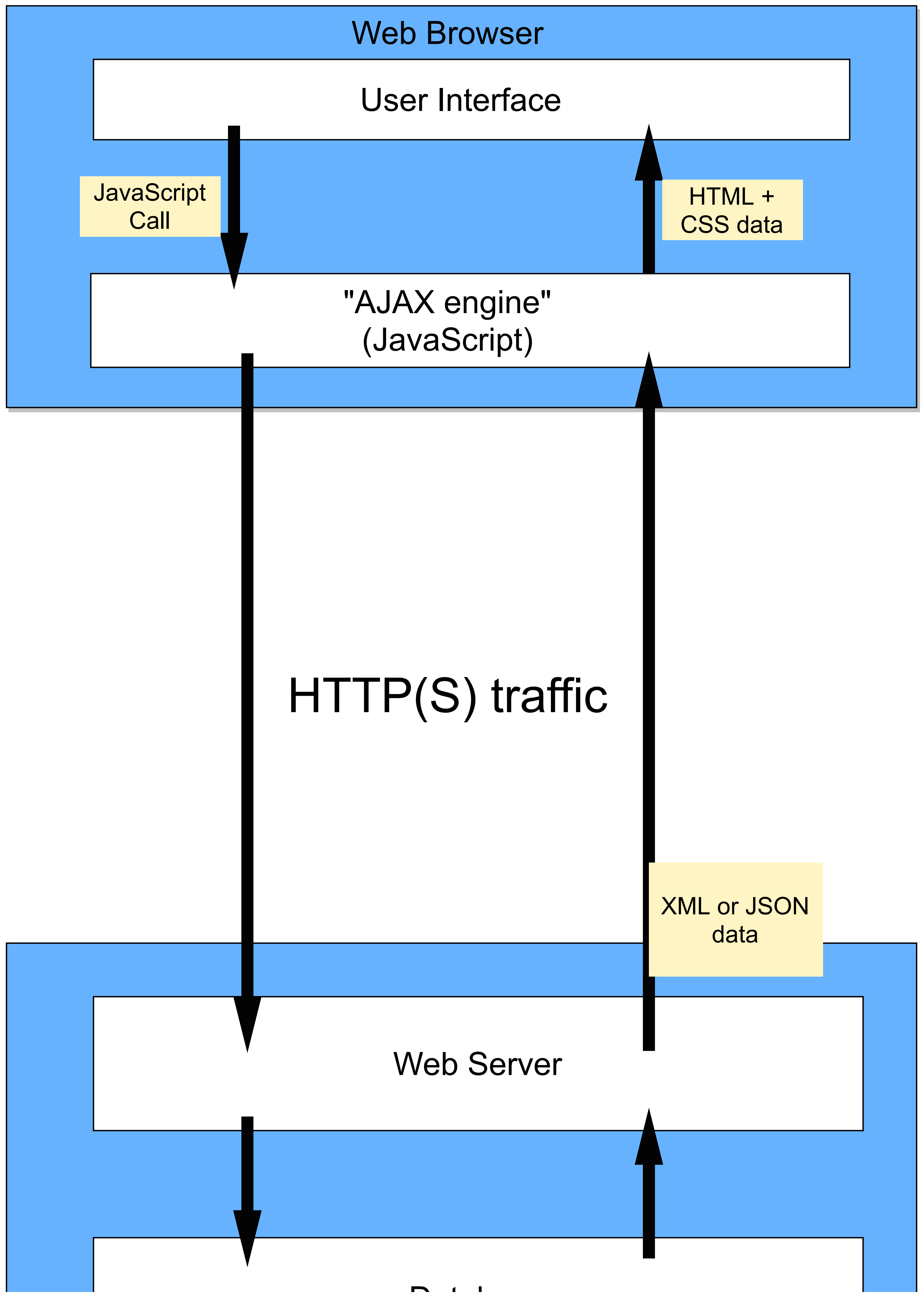
In the next lesson, we'll do a deeper dive into each step of AJAX.

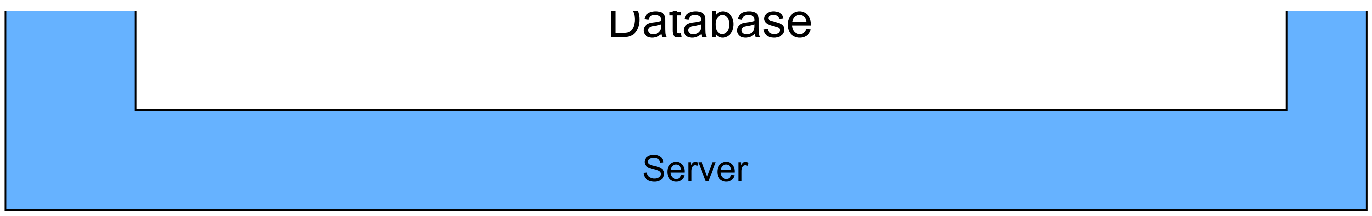
AJAX Steps

In the previous reading, you learned about what the purpose behind AJAX and what AJAX meant.

In this reading, you'll walk through the specific steps of AJAX using the example from the previous reading.

Specifically, we'll walk through each step of the Goodreads example using this diagram:


Web Browser
User Interface
JavaScript Call
HTML + CSS data
"AJAX engine" (JavaScript)
HTTP(S) traffic
XML or JSON data
Web Server



Quick overview of the Fetch API

As you go through the AJAX example, you'll use an API that you've used in the past: the Fetch API.

At a high level, Fetch is used to make HTTP requests. It uses Promises to handle the asynchronous nature of HTTP requests and responses.

To learn more about Fetch, let's walk through a series of examples using `https://jservice.xyz/`, which is a publicly available API that allows users to create, update, or delete Jeopardy-related resources (ie. games, categories, or clues).

Since the Fetch API is provided by almost all major browsers, feel free to open up a console in Chrome or Firefox and follow along.

GET request

Let's start with a GET request. As a refresher, GET requests are used to retrieve information from the server. Here's what a GET request might look like using the Fetch API:

```
fetch("https://jservice.xyz/api/games")
  .then(function(res) {
    console.log("response: ", res);
    return res.json();
  })
  .then(function(data) {
    console.log("data:", data);
  });
```

In the code example above, the following happens:

1. `fetch`'s first argument is the URL that you want to make a request to. The second argument is an optional `options` object that is not necessary for now, but will be used in a later example.

2. Invoking `fetch` returns a Promise that resolves with a Fetch Response object. This Response object represents the entire HTTP response, and it holds crucial information like `status` and `url`.
3. In the first callback, the body of the response is a ReadableStream. We won't get into the details of data streams here, but for now just know that the `.json()` method takes that stream and, according to the MDN docs on the .json() method, *"It returns a promise that resolves with the result of parsing the body text as JSON."* The `.json()` method is the equivalent of using `JSON.parse()` except that it works on a ReadableStream instead of just a string.
4. In the second callback, you can now access the data found in the body of the response. In this case, a GET request to `https://jservice.xyz/api/games` returns a list of Jeopardy games stored in an object that looks like this:

```
{
  games: [
    {
      id: 1,
      episodeId: 4596,
      aired: "2004-09-06",
      canon: true
    },
    // More games here
  ]
}
```

POST request

Next, let's walk through a POST request. As a reminder, POST requests are used to create data on the server.

```
fetch("https://jservice.xyz/api/categories", {
  method: "POST",
  headers: {
    "Content-Type": "application/json"
  },
  body: JSON.stringify({
    title: "ACTORS & THEIR FILMS"
  })
})
  .then(function(res) {
    return res.json();
  })
  .then(function(data) {
    console.log(data);
  });
```

If you're following along, you probably just got an error response here with a message notifying you that the `category already exists`. Go ahead and update that `title` to be something different until you are able to create a unique category.

Now that it's a POST request, a second `options` argument is passed in to configure this HTTP request. In this example, you specify the request as a POST request, notify the server that you will be sending data in a JSON format, and then also pass along the data in the body.

When the POST request succeeds, the server responds with data about your newly created category.

Error handling

At this point, something might seem a little off to you. Specifically, in the previous example, the server responded with an error, yet the Promise resolved instead of being rejected.

Since the example did not have a `catch` block to handle errors, you may have expected some sort of `Uncaught (in promise)` error.

It turns out that the Fetch API will not reject on HTTP error status codes (between 400 and 600). It only rejects on errors like network failures. Try turning off your WiFi and then executing that code snippet if you want to confirm this.

Instead, Fetch requires you to check the `ok` key inside of the Response object, and if that key is set to `false` , then you can then properly handle the error. The `ok` property is set to `true` if the `status` code is in the 200s range.

```
fetch("https://jservice.xyz/api/categories", {
  method: "POST",
  headers: {
    "Content-Type": "application/json"
  },
  body: JSON.stringify({
    title: "ACTORS & THEIR FILMS"
  })
})
  .then(function(res) {
    console.log(res);
    if (!res.ok) {
      throw Error(res.statusText);
    }
    return res.json();
  })
  .then(function(data) {
    console.log(data);
  })
  .catch(function(error) {
    console.log(error);
  });
```

Now, if the server responds with an error, the code throws an error that will be handled by the `catch` block. This `catch` block will also handle any errors that result from network failures.

AJAX broken down

Now that you've gone over how the Fetch API works, let's begin walking through the AJAX example!

Recall that in the previous example, all you wanted to do was allow the user to click the "Want To Read" button without requiring an entire page reload. Before AJAX, this was not possible because when the user clicked the "Want To Read" button, an HTTP request would be sent to the server to mark that book as "Want To Read", and then the server would send back an entire HTML document (with the book marked as "Want To Read") for the browser to load up.

Now, however, we can allow for that HTTP request to happen asynchronously and then update the DOM when the server returns a response about whether or not that HTTP request was successful.

Let's go step by step into how this works!

Event listener and Fetch

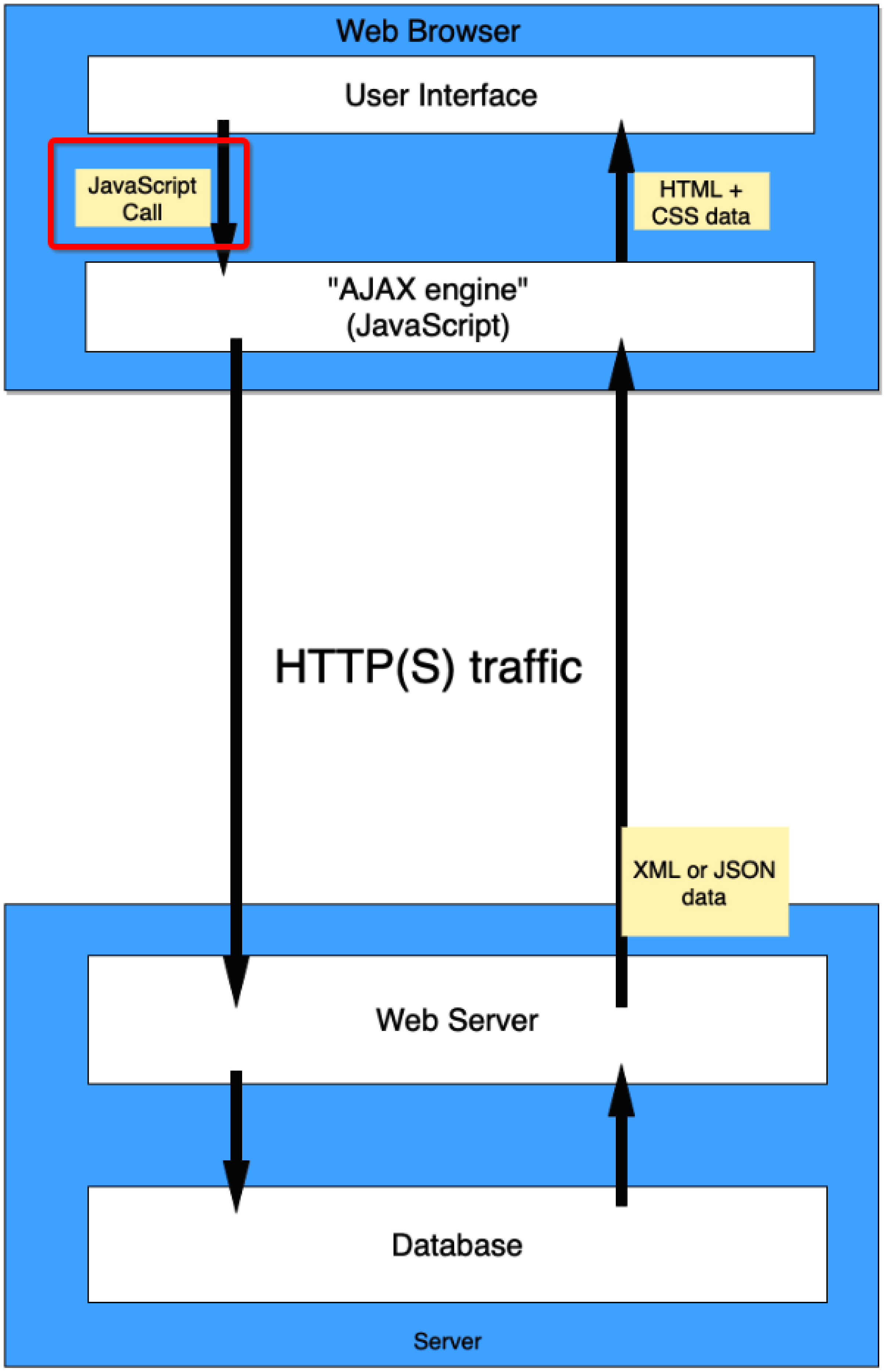
The "Want to Read" button has a `click` event handler. This handler would then interact with a browser API for making asynchronous call to the server.

The code for this might look like:

```
<button class="want-to-read">Want to Read</button>

<script>
  document.querySelector(".want-to-read").addEventListener("click", function() {
    fetch(`https://api.goodreads.com/books/${BOOK_ID}/update-status`, {
      method: "PATCH", // PATCH since we'll just be modifying the book's status
      body: JSON.stringify({
        status: "Want to Read"
      })
    });
    // HANDLING THE SERVER RESPONSE WILL GO HERE AND WILL BE COVERED
    // IN A LATER STEP.
  });
</script>
```

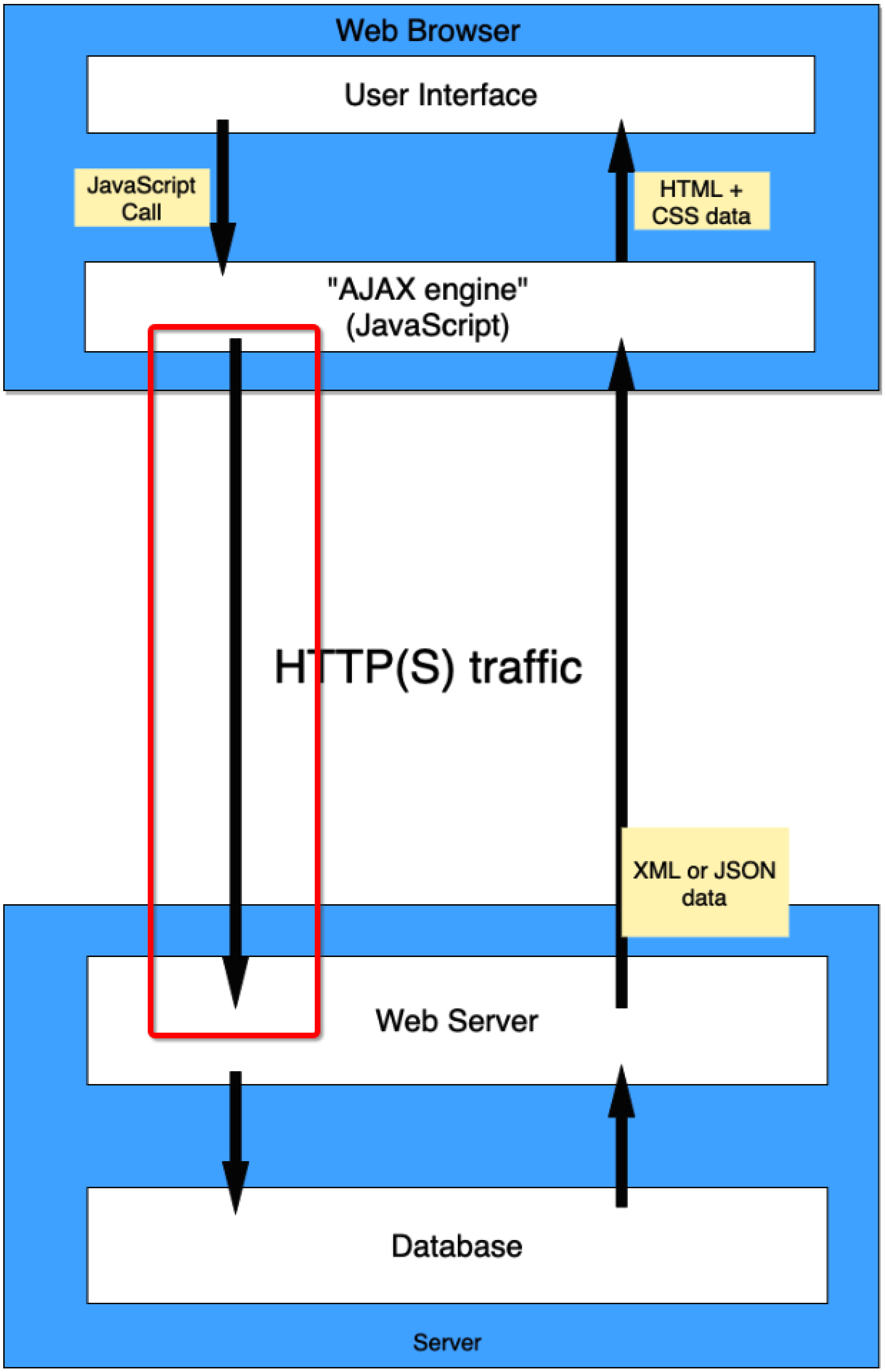
In the diagram, this is the step where the user interface makes a "javascript call" to the "AJAX engine":


Web Browser
User Interface
JavaScript Call
HTML + CSS data
"AJAX engine" (JavaScript)
HTTP(S) traffic
XML or JSON data
Web Server
Database
Server

To be clear, there is no actual formal "AJAX engine" that lives in the browser. Rather, in this example, the "AJAX engine" is really just the JavaScript code (ie. the click event handler, invocations of the Fetch API, and any sort of DOM manipulation).

PATCH request is made to web server

During this step, the PATCH request is made asynchronously to the server:


Web Browser
User Interface
JavaScript Call
HTML + CSS data
"AJAX engine" (JavaScript)
HTTP(S) traffic
XML or JSON data
Web Server
Database
Server

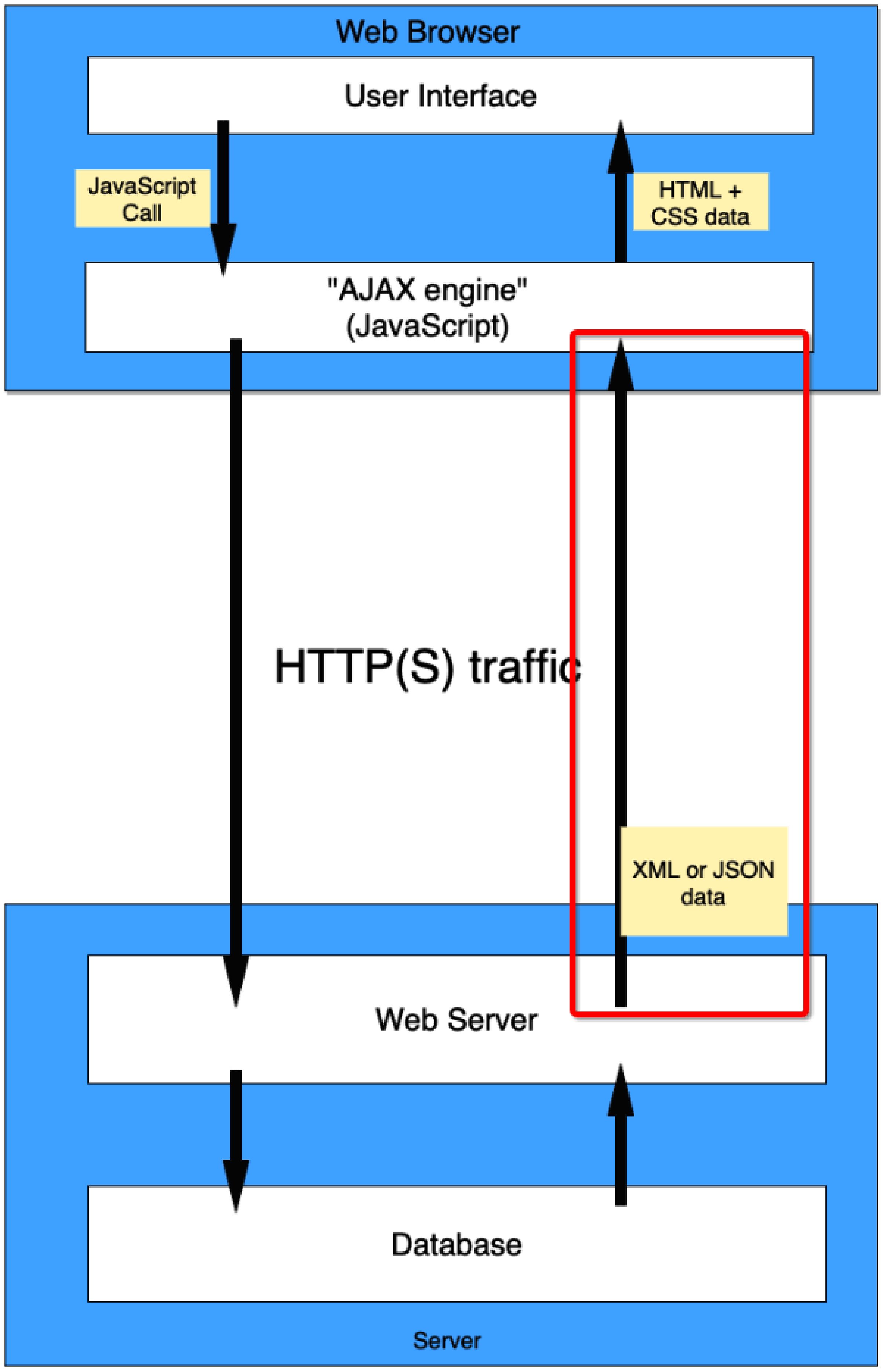
At this step, the JavaScript code (in this case the `click` event handler) calls the Fetch API and delegates all of the logic of making the PATCH request to the Fetch API. Meanwhile, the rest of the JavaScript can continue working without being blocked. For example, if there was another click handler for another element on the page, it **does not** have to wait for this "Want To Read" status update PATCH request to be finished.

If you need a refresher on how JavaScript works asynchronously, please refer back to the previous lesson on asynchronous JavaScript from earlier in the course.

The PATCH request is then made to the server, and the server updates the status of that specific book.

Server sends back a response

Once the server is done updating the status of the book, it sends a response back to the client:


Web Browser
User Interface
JavaScript Call
HTML + CSS data
"AJAX engine" (JavaScript)
HTTP(S) traffic
XML or JSON data
Web Server
Database
Server

As the previous reading mentioned, when AJAX was first introduced, the data was often sent back in the XML format (hence the XML in ajaX). Nowadays, JSON is the most common format for server responses due to its smaller payload size, readability, and ease in interacting with JavaScript.

Here's an example of what that JSON response might look like:

```
{
  "book": {
    "id": 29633913,
    "status": "Want to Read"
  }
}
```

Fetch API receives the response and resolves the promise

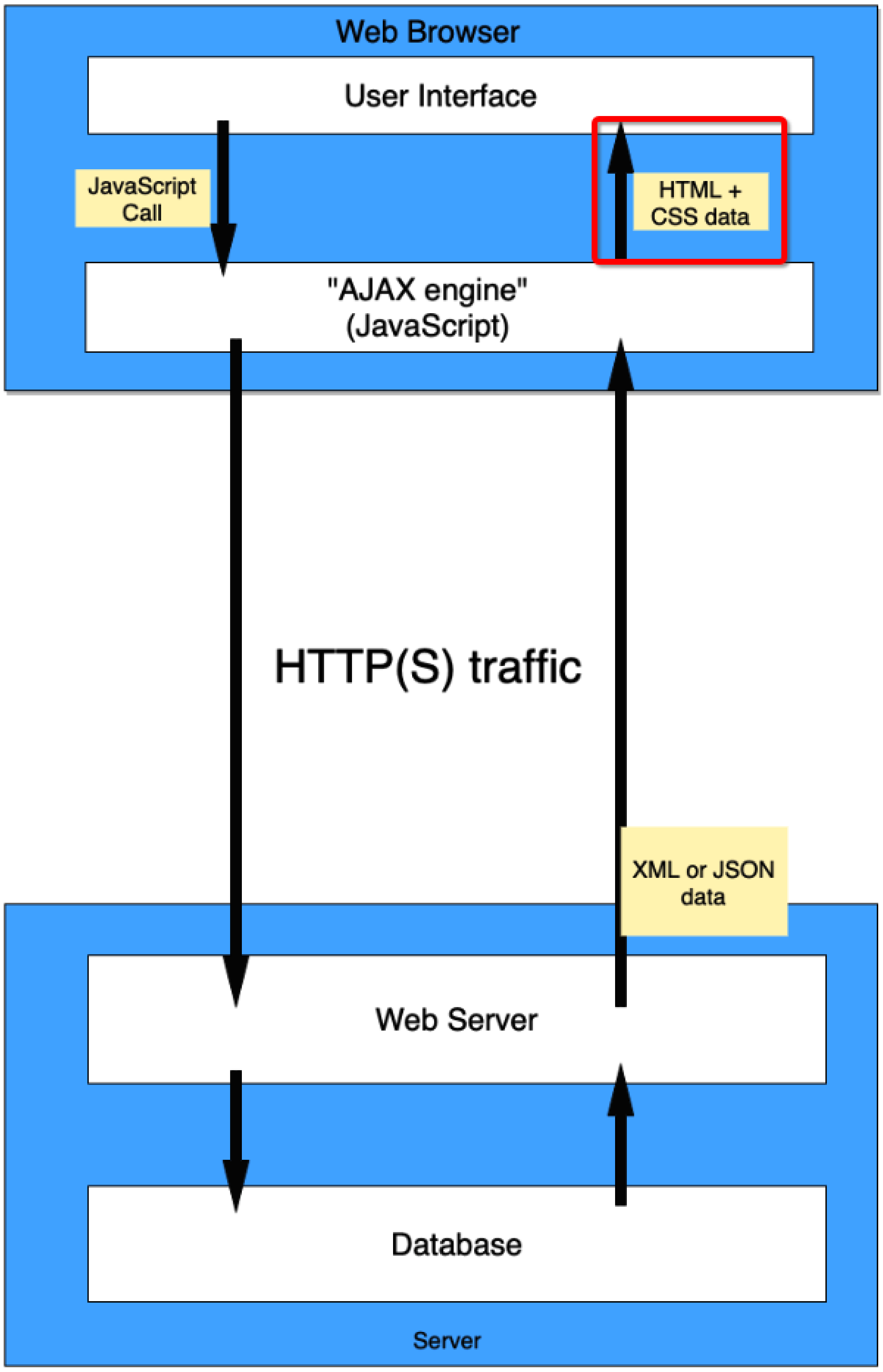
Once the response is received, your JavaScript code can properly handle the response:

```
<button class="want-to-read">Want to Read</button>

<script>
  document.querySelector(".want-to-read").addEventListener("click", function() {
    fetch(`https://api.goodreads.com/books/${BOOK_ID}/update-status`, {
      method: "PATCH", // using PATCH since we'll just be modifying the book's status
      headers: {
        "Content-Type": "application/json"
      },
      body: JSON.stringify({
        status: "Want to Read"
      })
    })
      .then(function(res) {
        if (!res.ok) {
          throw Error(res.statusText); // handle any potential server errors
        }
        return res.json(); // extract JSON from the response
      })
      .then(function(data) {
        // handle the response data here
      })
      .catch(function(error) {
        // handle errors here
      });
  });
</script>
```

Response is handled and DOM is updated

You're now in the last step of the AJAX flow:


Web Browser
User Interface
JavaScript Call
HTML + CSS data
"AJAX engine" (JavaScript)
HTTP(S) traffic
XML or JSON data
Web Server
Database
Server

In this step, you can update the DOM based on the data that was returned:

```html
<button class="want-to-read">Want to Read</button>

<script>
  document.querySelector(".want-to-read").addEventListener("click", function() {
    fetch(`https://api.goodreads.com/books/${BOOK_ID}/update-status`, {
      method: "PATCH", // using PATCH since we'll just be modifying the book's status
      headers: {
        "Content-Type": "application/json"
      },
      body: JSON.stringify({
        status: "Want to Read"
      })
    })
      .then(function(res) {
        if (!res.ok) {
          throw Error(res.statusText); // handle any potential server errors
        }
        return res.json(); // extract JSON from the response
      })
      .then(function(data) {
        document.querySelector(".want-to-read").innerHTML = "✓ Want To Read";
      })
      .catch(function(error) {
        const errorMessage = document.createElement("p");
        errorMessage.appendChild(
          document.createTextNode("Something went wrong. Please try again!")
        );

        // This example appends an error message to the body for simplicity's sake.
        // Please do not copy this kind of DOM manipulation in your own projects:
        document.querySelector("body").appendChild(errorMessage);
      });
  });
</script>
```

In the example above, since there was a successful response from the server, then the button is updated to include a check mark to show that the book has been successfully marked as "Want To Read".

Recall from the section on the Fetch API that even if the server responds with an error, the response is still resolved. Because of that, it's important to check whether or not the response was successful in the response's `ok` and/or `status` code and then handle errors accordingly.

In the `catch` block, we ensure that if an error were to have happened, then the website would show an error message to the user.

What you've learned

You've now learned each step of a typical AJAX flow. As a recap, it usually starts with an event on the client side that triggers an HTTP request to the server. In this case, we used the Fetch API to asynchronously interact with the server. When the server sent its response, the Fetch API resolved the promise, and the DOM was then updated to reflect the updated data.

More importantly, using AJAX, the web page can be updated without requiring a full page reload.

Now that you've learned about each step of AJAX, it's time to try it out yourself!

AJAX Project Preparation

Now that you've learned the fundamentals of AJAX, it's time to put that knowledge into practice by building out a project that uses AJAX.

Since AJAX is comprised of multiple technologies, you'll work with multiple components across the frontend and backend of a web application in this AJAX project. Specifically, in this project, you'll work with:

- an Express server
- an HTML document
- JavaScript event handling
- Fetch API

This reading will go over how ths project will work, and then also give a high level over view of the Express framework.

Overview

At a high level, here's the flow of how your project will work:

1. The user navigates to your web application's root URL. At this point, the browser makes a request to the Express server, and the server responds with the HTML document for the web application.
2. Then, when different events happen on the DOM, various JavaScript event handlers (that you will implement) will trigger requests to the server using the Fetch API.
3. The Express server processes those requests and responds with JSON data.
4. The response data is handled and used to manipulate the DOM.

Project organization

Your project will be organized like this:

```
AJAX project
│   index.js
|   package.json
└───public
│   │   events.js
│   │   index.html
│   │   index.css
```

The `index.js` file holds the Express server. You'll learn more about the Express server in the next section.

The `public` directory holds all of the static assets that the Express will serve up to the client, including the `index.html` , the `index.css` that styles the `index.html` , and the `events.js` that loads up all of the event listeners that listen to events being performed on the various elements in the `index.html` document.

Next, let's go over how the Express server works!

Express

Express is a Node.js framework for building web applications.

Let's go over some of the core parts of an Express web server by breaking down the following example:

```
const express = require("express");
const faker = require("faker");
const path = require("path");
const bodyParser = require("body-parser");

const app = express();
app.use(bodyParser.json());
app.use(express.static("public"));

const port = 3000;

app.get("/", (req, res) => {
  res.sendFile(path.join(__dirname + "/public/index.html"));
});

app.get("/names", (req, res) => {
  const randomName = faker.name.findName();
  res.json({ name: randomName });
});

app.listen(port, () => console.log(`Example app listening on port ${port}!`));
```

In the first line of the example, we require the `express` dependency so that we can then start an Express server.

Middleware

This particular Express server uses two middleware: `bodyParser` and `static`. You'll learn much more about Express and its middleware in a later lesson, but for now, all you need to know is that middleware sit between the client and the server and can process and/or transform the HTTP requests or responses that pass between them.

`app.use(bodyParser.json())` sets up the `bodyParser` middleware to process data in HTTP requests into a JSON object that can then be used in the routes.

`app.use(express.static('public'))` sets up the `static` middleware. In this example, this allows the Express server to serve any static assets that are located in the `public` directory. Some examples of static assets include images, html documents, or CSS files.

`app.use(morgan('dev'))` sets up the `morgan` middleware. This will show the method and the url for any request made to the server in the server terminal.

Routes

This example Express server has two routes: `/` and `/names` . In setting up those routes, you have to specify the HTTP verb that the client should use in order to hit those routes.

In this case a GET request to `/` would invoke the callback function with `res.sendFile(path.join(__dirname + "/public/index.html"));` in it.

When the user lands on the root path of the website, the client is served with that `index.html` document.

A GET request to `/names` would invoke the callback function that generates a random name using the `faker` library and then sends back a JSON response with that newly generated name.

Launching the server

In the code example above Express starts listening for network requests on port 3000.

To launch the server, you can run `node [NAME_OF_EXPRESS_SERVER_FILE]` . Let's assume that the code example above lives in an `index.js` file located a the root of the directory. To launch the server, you would run `node index.js` .

What you've learned

In this reading, you got a preview and brief breakdown of the project that you'll be building. You also got a high level overview of the Express framework.

It's okay if you don't feel like you fully understand everything about Express yet. You'll be learning much more about Express in the following weeks. For now, make sure you understand the key points of Express outlined in this reading so that you're able to confidently interact with the Express server while making AJAX requests in your project.

Catstagram

Today you'll be building a project called Catstagram! Here are the features of Catstagram:

- It shows a random kitten picture from the `https://thecatapi.com/` API.
- Users can vote on the picture.
- Users can comment on the picture.
- Users can delete comments.
- Users can request a new random kitten image to be shown.

You can download the starter project from
https://github.com/appacademy-starters/ajax-project-starter.

The most important feature of Catstagram is that all of the other features listed above can be performed without requiring a page reload. For example, when a new comment is created, the comment smoothly gets added to the page without requiring the website to reload.

In building Catstagram, you will master the fundamentals of AJAX. Specifically, you will implement multiple AJAX cycles in this project, and you will become familiar with the nuances of each step.

To start, download the skeleton zip. The skeleton contains an Express server, an `index.html` page, and an `event.js` JavaScript file. If you need a refresher on how these different components work together, please review the previous reading.

Take some time to browse through each file. Start with the `public/index.html` file and note how the document is structured. Then, review the `public/index.css` file to see how it's currently styling the HTML page.

You'll be primarily working in the `public/events.js` file today, and at the current moment, it's blank. In this file, you'll be setting up event listeners and implementing AJAX requests using the Fetch API.

Finally, check out the `index.js` file. It might seem like there's a lot going on in this file, but the only piece that you have to edit in this file today is the `ERROR_RATE` . Otherwise, your primary interaction with this file will involve looking at the endpoints that you should be making requests to and seeing how the server handles each client request.

When you're ready, launch the server by running `npm start` , and then go to your browser and navigate to `localhost:3000` .

API Endpoints

Here are the API endpoints on the server that you will be using. You should still read the `index.js` file for how the API endpoints are being created by the server.

GET /kitten/image

Fetches an image from an external API, `https://thecatapi.com/`, that returns information on a random cat image url.

```
{
  "score": 0,
  "comments": [],
  "src": string (image url)
}
```

If it doesn't succeed, it returns an error message.

```
{
  "message": string
}
```

PATCH /kitten/upvote

Increments the score of the current kitten by 1 and returns the current score.

```
{
  "score": number
}
```

PATCH /kitten/downvote

Decrements the score of the current kitten by 1 and returns the current score.

```
{
  "score": number
}
```

POST /kitten/comments

Creates a new comment for the current kitten. To create a comment, the server expects the request body to look like the JSON below.

```
{
  "comment": string
}
```

After sending a request to the server, the API endpoint returns all the comments of the kitten in the order that they were created. The JSON format below is the way the API endpoint returns information.

```json
{
  "comments": [ string ]
}
```

Phase 1: Load the initial cat image

Let's start with the `index.html` in the `public` directory. Right now this `html` document is being served by the Express server when the user navigates to `localhost:3000` and lands on the root route ('/').

`index.html` has an `<img>` element, but the element has an empty `src` attribute. In this first phase, set up an event listener that waits for the DOM content to be loaded. When the DOM content is loaded, the client should make a GET request, using the Fetch API, to the `/kitten/image` route.

When that route is hit, the server makes a request to The Cat API for a random kitten image. Once that kitten image is returned, it sends data about the kitten image back to the client.

When the server responds, update the DOM so that it's showing the kitten picture.

Phase 2: Implement `New Pic` button

Now that your website loads a kitten picture initially, we want to implement a feature where the user can request a new kitten picture.

At the moment, the user could simply refresh the website to do that, but let's instead use AJAX to give the user a smooth experience that does not require a full page reload.

To do this, in `public/events.js` , add another event listener that now listens for when anyone clicks the "New Pic" button. When that button is clicked, it should ask the server for another kitten image and then display that image.

Once you've finished that, go ahead and let the user know when the client is waiting for the response from the server. We can do this by displaying the text

"Loading..." in the div with the class of "loader" any time we fire off an HTTP request to the server and then clear that "Loading..." text once the response arrives.

Phase 3: Error Handling

Various issues can arise in the HTTP request/response cycle. As developers, it's your responsibility to handle errors that might come back from the server.

For example, if your server's request for a new image from the The Cat API failed, then your app should notify your users of the issue.

Let's simulate errors by adjusting the `ERROR_RATE` variable in `index.js`. As the comment above the `ERROR_RATE` variable in `index.js` file mentions, `ERROR_RATE` represents the probability that an error will be thrown. So, if the `ERROR_RATE` is updated to 100 percent, then the `generateRandomError` function will throw an error every single time it's invoked. If the rate is at 0, then an error will never be thrown.

Go ahead and bump the rate up to 100 as you're implementing error handling. For now, if the server responds with an error, go ahead and just `alert` the user with a "Something went wrong! Please try again!" message.

Phase 4: Voting

Next, let's add the ability to upvote and downvote the kitten picture. As you're implementing this feature, pay close attention to the HTTP method used to make this request.

Although the `upvote` and `downvote` endpoints don't have the `generateRandomError()` function inside, it's still possible for errors to occur, so be sure to implement error handling here as well.

One more thing, take this time to refactor and DRY up any code that might have gotten too repetitive. Be sure to continue refactoring throughout whenever there's an opportunity to make your code more concise and readable.

Phase 5: Create Comment

To allow users to create comments, set up an event listener for the comment form's `submit` event.

Check out the FormData interface for more info on how to extract the data from the form.

By default, when a `submit` event happens on a form, an HTTP request is automatically made based on the form's `action` and `method` attributes. During this default HTTP request, the page reloads. Because you'll be using AJAX instead to make an HTTP request without reloading the page, go ahead and call event.preventDefault() on the `submit` event handler's callback function.

If you are struggling with making this HTTP request, be sure to go back to the previous reading and review the section where it made a similar type of HTTP request.

When this feature is properly implemented, any time the user submits a comment, the newly created comment should be appended below all the existing comments.

Phase 6: Improve error handling

Let's improve our error handling to not use the `alert` function. Instead, let's display the specific error message that our server is responding with.

If we look at `index.js` , `generateRandomError` will randomly throw one of the following the following three error messages:

- "No cat for you!"
- "Sad day. No kitten here."
- "Please try again!"

It would be ideal if the user can see the specific error message that's coming from the server.

Please display the specific error message in the div with the class of "error". Feel free to adjust the `ERROR_RATE` again in order to make it easier to implement this feature.

Recap

At this point, you should have the following features implemented:

- An image should load from the server when a user arrives on Catstagram.
- If users want to fetch a new kitten image, then clicking the 'New Pic' button should load up a new image. (Also, when the image is being loaded, the text 'Loading...' should show up in the `.loader` div. Once the image is done loading, that `Loading...` text should disappear.)
- If the server responds with an error, the user should see that specific error message in the `.error` div.
- The user can upvote and downvote the kitten image.
- The user can create a comment. When a comment is created, it gets appended below all the other existing comments.

Great job on implementing all the core features of Catstagram! Before moving on to the bonus section, review your `events.js` file one more time to see if there's any other refactoring you can do to clean up your code.

Bonus: Delete Comments

Nice work getting to the bonus feature! For the bonus, implement a feature that would allow users to delete existing comments.

To implement this, you will have to refactor your comment creation feature to now include a `Delete` button next to the comment.

Then, each button will require event handling so that whenever it is clicked, it makes a request to the `delete /kitten/comments/:id` endpoint.

To properly implement this, you should try to use Event Delegation so that you don't have to set up an individual click handler on each `Delete` button. If you need a refresher on Event Delegation, please go back to the Event Delegation content from earlier in the course. Alternatively, you could also review JavaScript.Info's lesson on Event Delegation.