Client-side web APIs (Part 1)

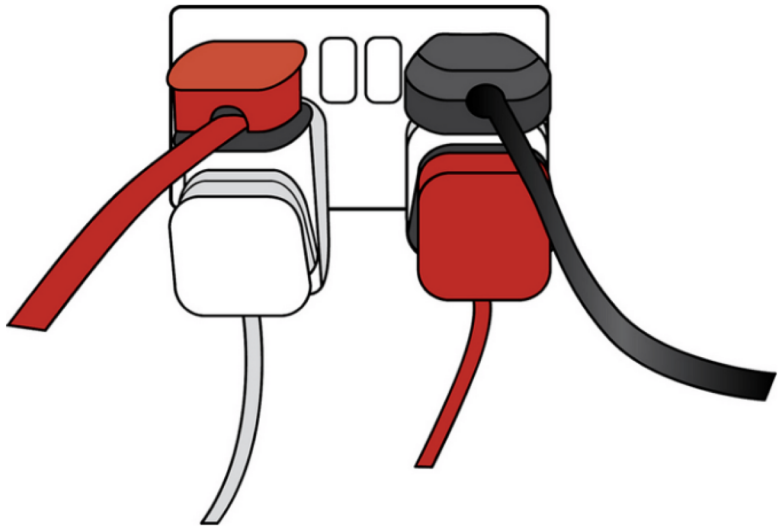
Assoc. Prof. Dr. Kanda Runapongsa Saikaew
(krunapon@kku.ac.th)
Dept. of Computer Engineering
Khon Kaen University

Agenda

- Introduction to web APIs
- Manipulating documents
- Fetching data from the server

What are APIs?

- Application Programming Interfaces (APIs) are constructs made available in programming languages to allow developers to create complex functionality more easily
- They abstract more complex code away from you, providing some easier syntax to use in its place.



APIs in client-side JavaScript

- Client-side JavaScript has APIs that fall into two categories
  - **Browser APIs** are built into a web browser and are able to expose data from the browser
    - Example: **Geolocation API** which retrieves location data to plot your location a Google map
  - **Third party APIs** are not built into the browser by default
    - Example: **Twitter API** which allows you to do things like displaying your latest tweets on your website

Relationship between JavaScript, APIs, and other JavaScript tools

- JavaScript — A high-level scripting language built into browsers that allows you to implement functionality on web pages/apps
  - Note that JavaScript is also available in other programming environments, such as Node.
- Browser APIs — constructs built into the browser that sit on top of the JavaScript language and allow you to implement functionality more easily
- Third party APIs — constructs built into third-party platforms (e.g. Twitter, Facebook) that allow you to use some of those platform's functionality in your own web pages

JavaScript libraries vs JavaScript frameworks

- JavaScript libraries — Usually one or more JavaScript files containing custom functions that you can attach to your web page to speed up or enable writing common functionality
  - Examples include jQuery, Mootools and React.
- JavaScript frameworks — The next step up from libraries, JavaScript frameworks (e.g. Angular and Ember) tend to be packages of HTML, CSS, JavaScript, and other technologies that you install and then use to write an entire web application from scratch
- The key difference between a library and a framework is “Inversion of Control”
  - When calling a method from a library, the developer is in control.
  - With a framework, the control is inverted: the framework calls the developer's code.

Common browser APIs (1/3)

- APIs for manipulating documents
  - DOM (Document Object Model) API which allows you to manipulate HTML and CSS
- APIs that fetch data from the server
  - APIs that make this possible include XMLHttpRequest and the Fetch API
- APIs for drawing and manipulating graphics
  - The most popular ones are Canvas and WebGL which allow you to update the pixel data in an HTML <canvas>

Common browser APIs (2/3)

- Audio and Video APIs
  - HTMLMediaElement, the Web Audio API, and WebRTC allow you to do really interesting things with multimedia
- Device APIs
  - APIs for manipulating and retrieving data from modern device hardware
    - Geolocation API, Notifications API, and Vibration API

Common browser APIs (3/3)

- Client-side storage APIs
  - The ability to store data on the client-side is very useful if you want to create an app that will save its state between page loads, and perhaps even work when the device is offline
  - Examples
    - Simple name/value storage with the Web Storage API
    - Complex tabular data storage with the IndexedDB API

Common third-party APIs (1/2)

- Twitter API – allows you to do things like displaying your latest tweets on your website
- Google Maps API – allows you to do all sorts of things with maps on your web pages
  - Google Maps API Picker
- Facebook suite of APIs - enable you to use various parts of the Facebook ecosystem to benefit your app
  - Example: Providing app login using Facebook login

Common third-party APIs (2/2)

- YouTube API - allows you to embed YouTube videos on your site, search YouTube, build playlists, and more.
- Twilio API - provides a framework for building voice and video call functionality into your app, sending SMS/MMS from your apps
- You can find information on a lot more 3rd party APIs at the Programmable Web API directory.

How do APIs work? (1/4)

- They are based on objects
  - Your code interacts with APIs using one or more JavaScript objects
  - Example: Geolocation API which consists of a few simple objects
    - **Geolocation** which contains three methods for controlling the retrieval of geodata
    - **Position** which represents the position of a device at a given time
    - **Coordinate** which contains the actual position information, plus a timestamp representing the given time

How do APIs work? (2/4)

- They have recognizable entry points
  - The Document Object Model (DOM) API has a simple entry point
    - It is found hanging off the Document object

```
1  var em = document.createElement('em'); // create a new em element
2  var para = document.querySelector('p'); // reference an existing p element
3  em.textContent = 'Hello there!'; // give em some text content
4  para.appendChild(em); // embed em inside para
```

How do APIs work? (3/4)

- They use events to handle changes in state
- Instances of the XMLHttpRequest object represents an HTTP request to the server to retrieve a new resource of some kind

```
1  var requestURL = 'https://mdn.github.io/learning-area/javascript/oojs/json/superheroes.json';
2  var request = new XMLHttpRequest();
3  request.open('GET', requestURL);
4  request.responseType = 'json';
5  request.send();
6
7  request.onload = function() {
8    var superHeroes = request.response;
9    populateHeader(superHeroes);
10   showHeroes(superHeroes);
11 }
```

How do APIs work? (4/4)

- They have additional security mechanisms where appropriate
  - WebAPI features are subject to the same security considerations as JavaScript and other web technologies (for example same-origin policy)
  - Some of the more modern WebAPIs will only work on pages served over HTTPS due to them transmitting potentially sensitive data (examples include Service Workers and Push).

Same-origin policy

- The **same-origin policy** is a critical security mechanism that restricts how a document or script loaded from one origin can interact with a resource from another origin
- It helps to isolate potentially malicious documents, reducing possible attack vectors.

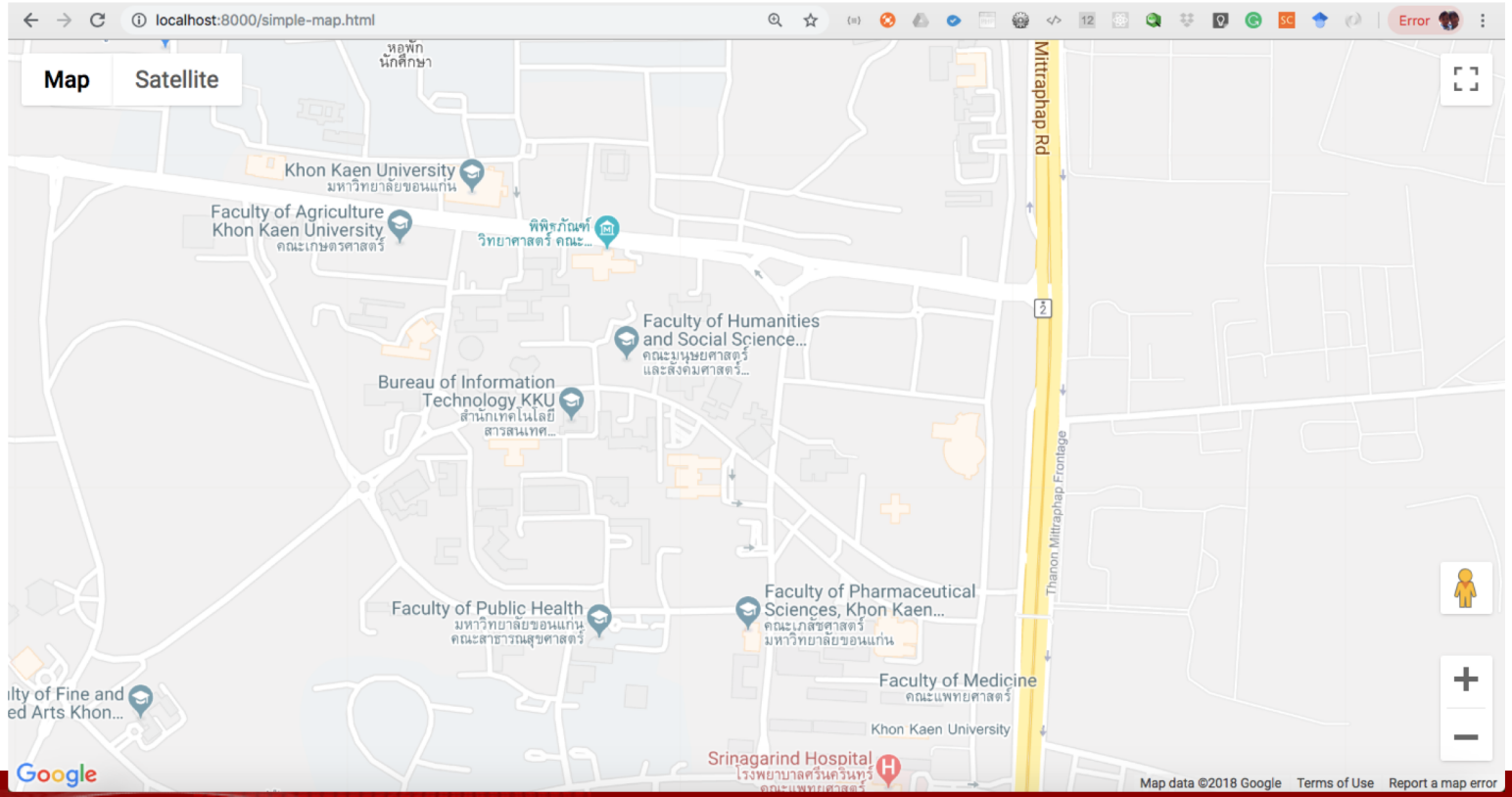
Same-origin policy example

- The following table gives examples of origin comparisons to the URL http://store.company.com/dir/page.html

URL	Outcome	Reason
`http://store.company.com/dir2/other.html`	Success	
`http://store.company.com/dir/inner/another.html`	Success	
`https://store.company.com/secure.html`	Failure	Different protocol
`http://store.company.com:81/dir/etc.html`	Failure	Different port
`http://news.company.com/dir/other.html`	Failure	Different host

Exercise 1: Calling Simple Map API in a Web Page

- https://developers.google.com/maps/documentation/javascript/tutorial
- Display a map of Khon Kaen University with zoom level 16



Agenda

- Introduction to web APIs
- **Manipulating documents**
- Fetching data from the server

Manipulating documents

- When writing web pages and apps, one of the most common things you'll want to do is manipulate the document structure in some way
- This is usually done by using the Document Object Model (DOM), a set of APIs for controlling HTML and styling information that makes heavy use of the Document object

The important parts of a web browser

The window

- The window is the browser tab that a web page is loaded into
- This is represented in JavaScript by the Window object
- Using method available on this object you can get the windows size (Window.innerWidth and window.innerHeight)

The navigator

- The navigator represents the state and identity of the the browser
- This is represented by the Navigator object
- You can use this object to retrieve things like
  - Geolocation information
  - The user's preferred language
  - A media stream from the user's webcam

The document

- The document (represented by the DOM in browsers) is the actual page loaded into the window
- It is represented in JavaScript by the Document object
  - Get a reference to an element in the DOM
  - Change its text content
  - Apply new styles to it
  - Create new elements and add them to the current element as children

The document object model

- The document currently loaded in each one of your browser tabs is represented by a document object model
- This is a "tree structure" representation created by the browser that enables the HTML structure to be easily accessed by programming languages

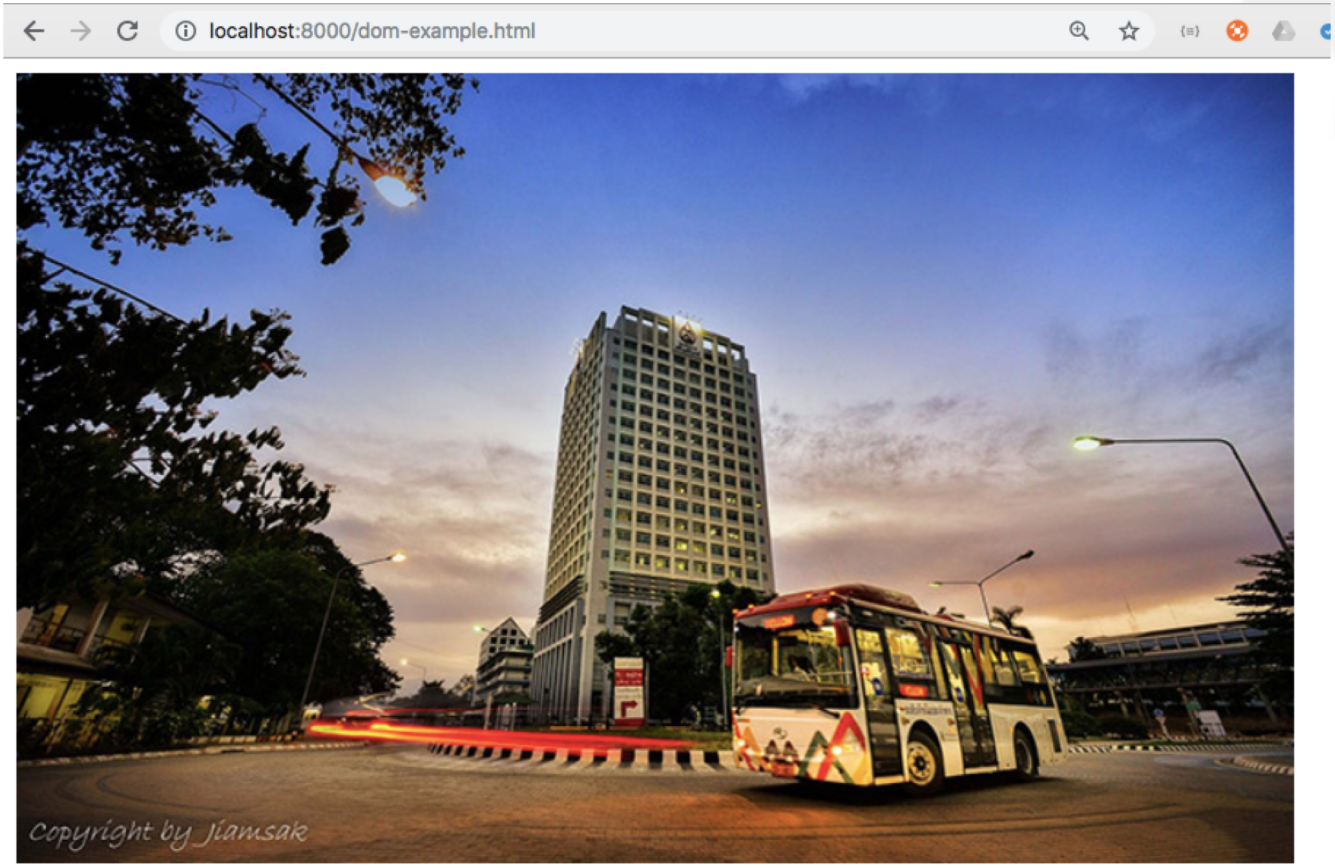
DOM: File dom-example.html

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="utf-8">
    <title>Simple DOM example</ti
  </head>
  <body>
     <section>
       <img src="images/kku.jpg"
         alt="Khon Kaen University">
       <p>Here we will add
a link to the
<a href="https://www.mozilla.org/
     </section>
  </body>
</html>
```

DOM view (hide, refresh):

```
DOCTYPE: html
HTML
  HEAD
    #text:
    META charset="utf-8"
    #text:
    TITLE
      #text: Simple DOM example
    #text:
  #text:
  BODY
    #text:
    SECTION
      #text:
      IMG src="images/kku.jpg" alt="Khon Kaen University"
      #text:
      P
        #text: Here we will add a link to the
        A href="https://www.mozilla.org/"
          #text: Mozilla homepage
      #text:
    #text:
```

The output of dom-example.html



Here we will add a link to the KKU

DOM Node (1/3)

- **Element node**: An element, as it exists in the DOM.
- **Root node**: The top node in the tree, which in the case of HTML is always the HTML node (other markup vocabularies like SVG and custom XML will have different root elements).
- **Child node**: A node *directly* inside another node
  - For example, IMG is a child of SECTION in the above example

DOM Node (2/3)

- **Descendant node**: A node *anywhere* inside another node
  - For example, IMG is a child of SECTION in the above example, and it is also a descendant
  - IMG is not a child of BODY, as it is two levels below it in the tree, but it is a descendant of BODY.
- **Parent node**: A node which has another node inside it
  - For example, BODY is the parent node of SECTION in the above example.

DOM Node (3/3)

- **Sibling nodes**: Nodes that sit on the same level in the DOM tree
  - For example, IMG and P are siblings in the above example.
- **Text node**: A node containing a text string
  - For example, “Here we will add a link to the” is the text node

Querying nodes

- Save file dom-example.html as dom-example2.html

```
var link = document.querySelector('a’);
link.textContent = "Faculty of Engineering, KKU";
link.href = "http://www.en.kku.ac.th";
```

Methods to query nodes

- Document.querySelector() is the recommended modern approach
  - It allows you to select elements using CSS selectors
- Document.querySelectorAll() matches every element in the document that matches the selector, and stores references to them in an array-like object called a NodeList

Older methods for getting elements

- Document.getElementById() selects an element with a given id attribute value
  - var elementRef = document.getElementById('myId’)
- Document.getElementsByTagName() returns an array containing all the elements on the page of a given type
  - var elementRefArray = document.getElementsByTagName('p')

Creating and placing new nodes

- Grabbing a reference to the our <section> element

```
var sect = document.querySelector('section');
```

- Create a new paragraph using Document.createElement()

```
var para = document.createElement('p');
        para.textContent = 'We hope you enjoy learning’;
```

- Append the new paragraph at the end of the section using Node.appendChild()

```
sect.appendChild(para);
```

Appending and removing nodes

- Creating text nodes and appending nodes

```
var text = document.createTextNode(' — the place where we are now');
var linkPara = document.querySelector('p');
linkPara.appendChild(text);
sect.appendChild(linkPara);
```

- Removing node

```
sect.removeChild(linkPara);
```

Setting styles for element para

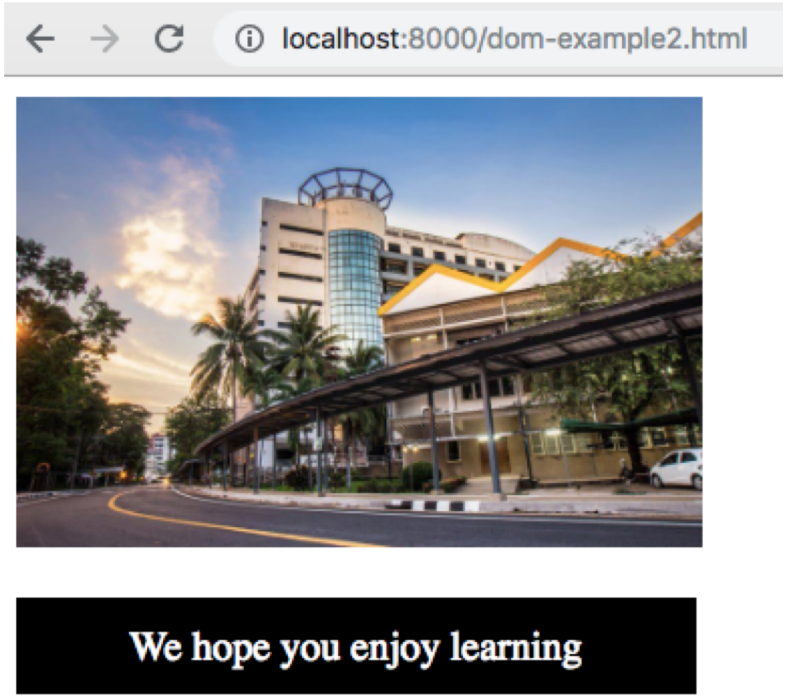
- Setting styles for element para

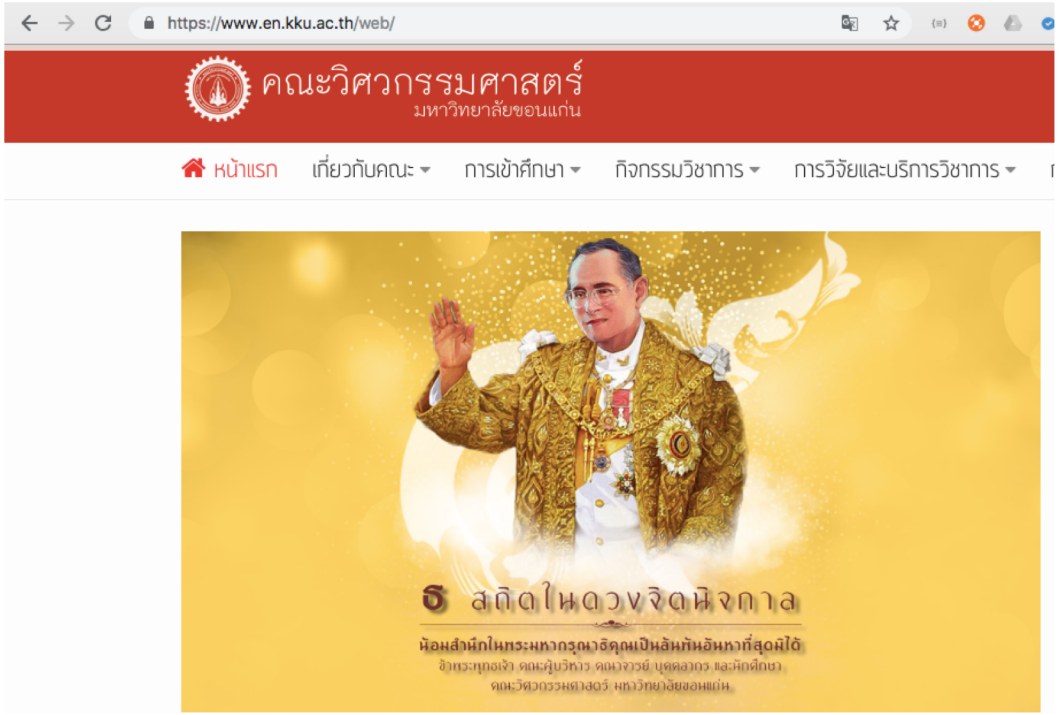
```
para.style.color = 'white';
para.style.backgroundColor = 'black';
para.style.padding = '10px';
para.style.width = '250px';
para.style.textAlign = 'center’;
```

```
▼<section>
    <img src="images/kku.jpg" alt="KKU">
    <p style="color: white; background-color: black; padding: 10px; width:
    250px; text-align: center;">We hope you enjoy learning</p>
  </section>
```

Exercise 2: Manipulating DOM

- Take a local copy of the [dom-example.html page]
- Modify it to develop the web app that has the interface and behavior as shown below

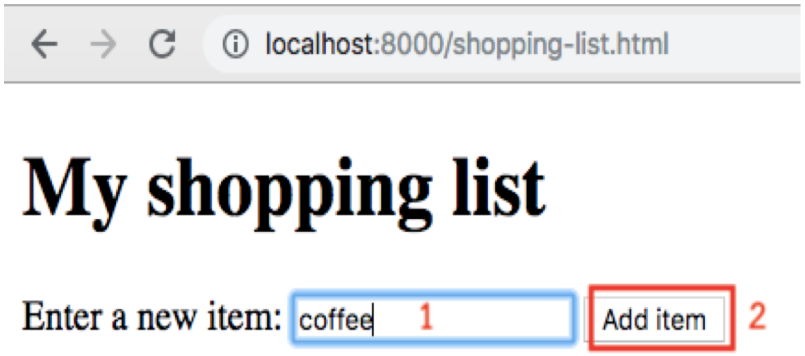


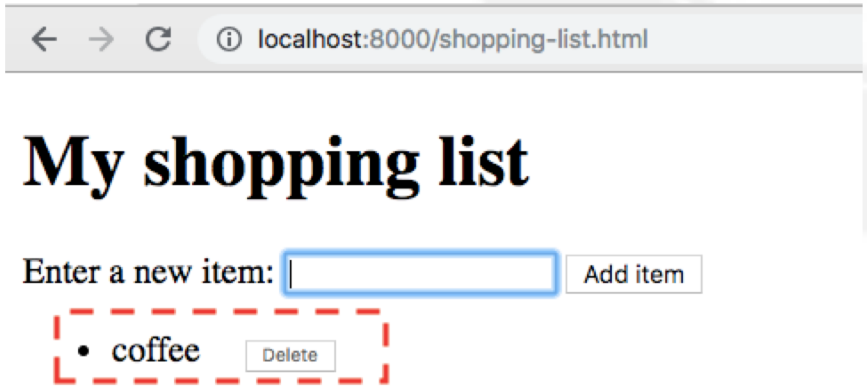


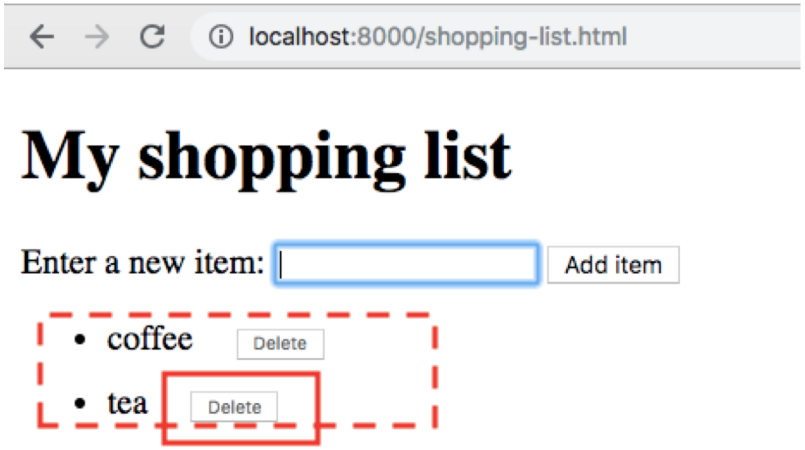
Exercise 3: A dynamic shopping list

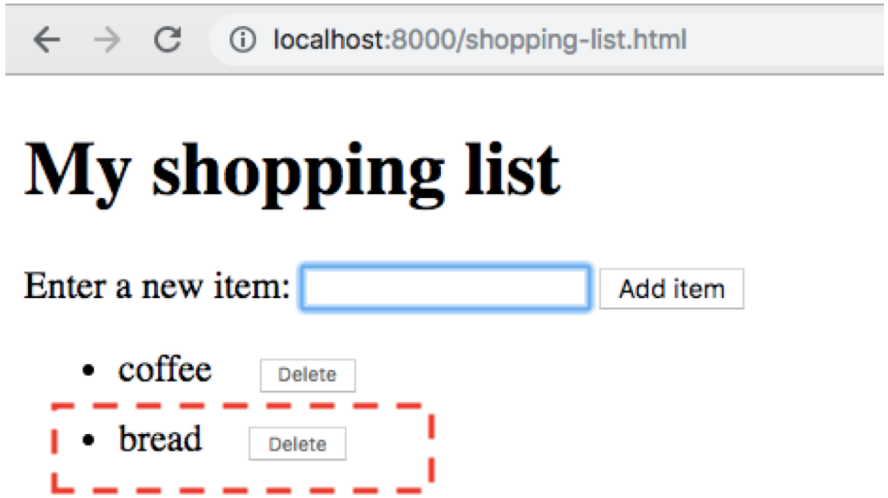
- We want to make a simple shopping list example that allows you to dynamically add items to the list using a form input and button
- When you add an item to the input and press the button
  - The item should appear in the list
  - Each item should be given a button that can be pressed to delete that item off the list
  - The input should be emptied and focus ready for you to enter another item

Exercise 3 Demo









A dynamic shopping list exercise (1/4)

- To start with, download a copy of our shopping-list.html starting file and make a copy of it somewhere
- You'll see that it has some minimal CSS, a list with a label, input, and button, and an empty list and <script> element
- You'll be making all your additions inside the script.

A dynamic shopping list exercise (2/4)

- Create three variables that hold references to the list (<ul>), <input>, and <button> elements.
- Create a function that will run in response to the button being clicked.
- Inside the function body, start off by storing the current value of the input element in a variable.

A dynamic shopping list exercise (3/4)

- Next, empty the input element by setting its value to an empty string — ''.
- Create three new elements — a list item (<li>), , and <button>, and store them in variables.
- Append the span and the button as children of the list item.

A dynamic shopping list exercise (4/4)

- Set the text content of the span to the input element value you saved earlier, and the text content of the button to 'Delete'.
- Append the list item as a child of the list.
- Attach an event handler to the delete button, so that when clicked it will delete the entire list item it is inside.
- Finally, use the focus() method to focus the input element ready for entering the next shopping list item.

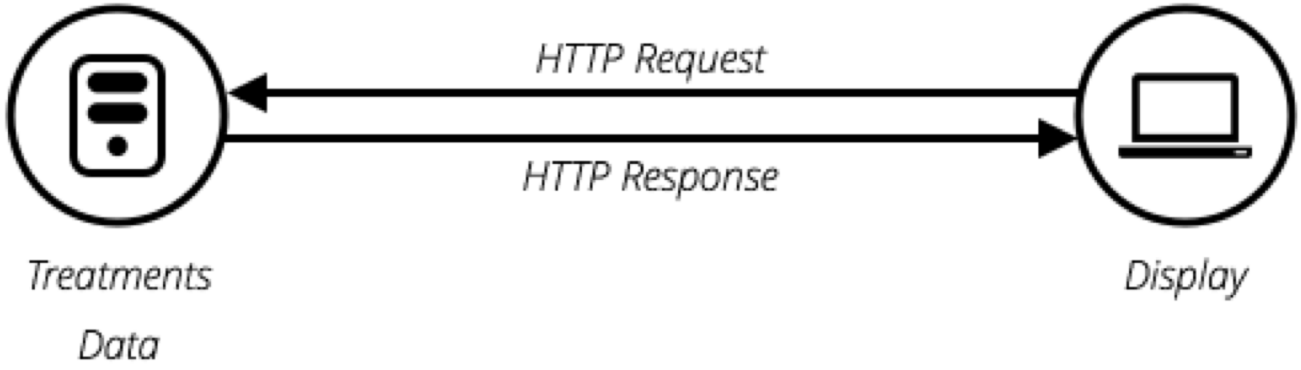
Agenda

- Introduction to web APIs
- Manipulating documents
- **Fetching data from the server**

Fetching data from the server

- Another very common task in modern websites and applications is retrieving individual data items from the server to update sections of a webpage without having to load an entire new page
- This seemingly small detail has had a huge impact on the performance and behavior of sites
- Technologies that make it possible are such as XMLHttpRequest and the Fetch API.

What is the problem here?



- To display a new set of products or load a new page, you've got to load the entire page again
- This is extremely wasteful and results in a poor user experience, especially as pages get larger and more complex.

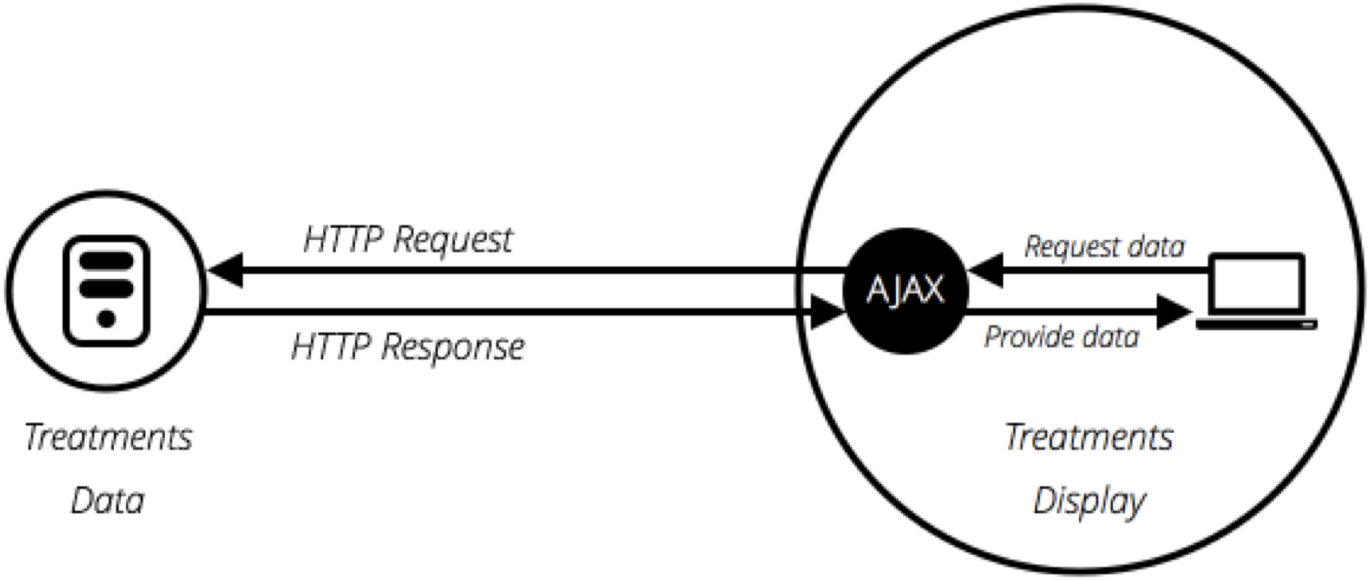
Enter Ajax

- This led to the creation of technologies that allow web pages to request small chunks of data (such as HTML, XML, JSON, or plain text) and display them only when needed, helping to solve the problem described above
- This is achieved by using APIs like XMLHttpRequest or — more recently — the Fetch API. These technologies allow web pages to directly handle making HTTP requests for specific resources available on a server, and formatting the resulting data as needed, before it is displayed.

Ajax term

- In the early days, this general technique was known as Asynchronous JavaScript and XML (Ajax), because it tended to use XMLHttpRequest to request XML data
- This is not normally the case these days (you'd be more likely to use XMLHttpRequest or Fetch to request JSON), but the result is still the same, and the term "Ajax" is still often used to describe the technique.

Ajax model


HTTP Request
HTTP Response
AJAX
Request data
Provide data
Treatments
Data
Treatments
Display

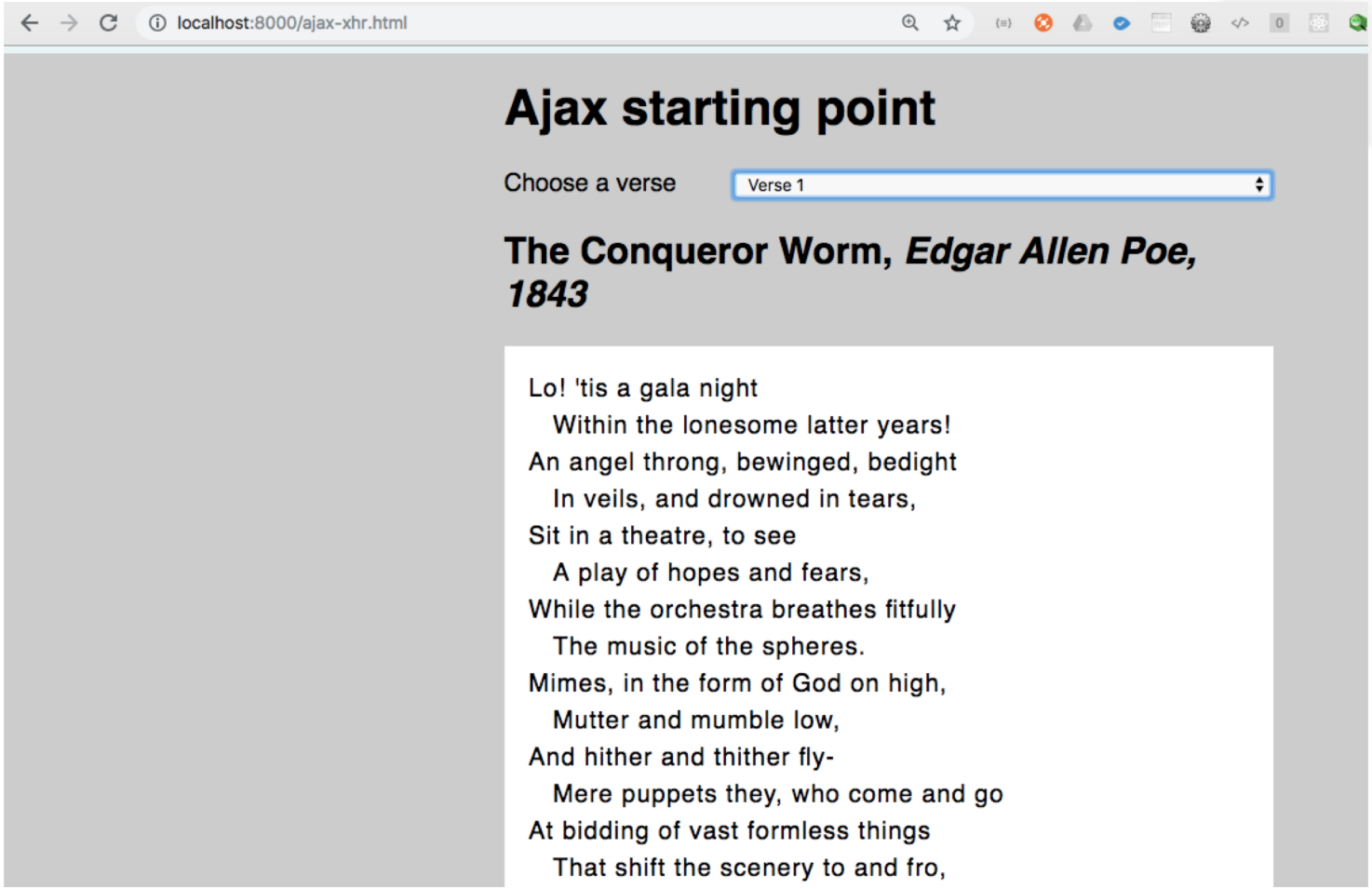
Benefits of Ajax model

- Page updates are a lot quicker and you don't have to wait for the page to refresh, meaning that the site feels faster and more responsive.
- Less data is downloaded on each update, meaning less wasted bandwidth
- This may not be such a big issue on a desktop on a broadband connection, but it's a major issue on mobile devices and in developing countries that don't have ubiquitous fast Internet service.

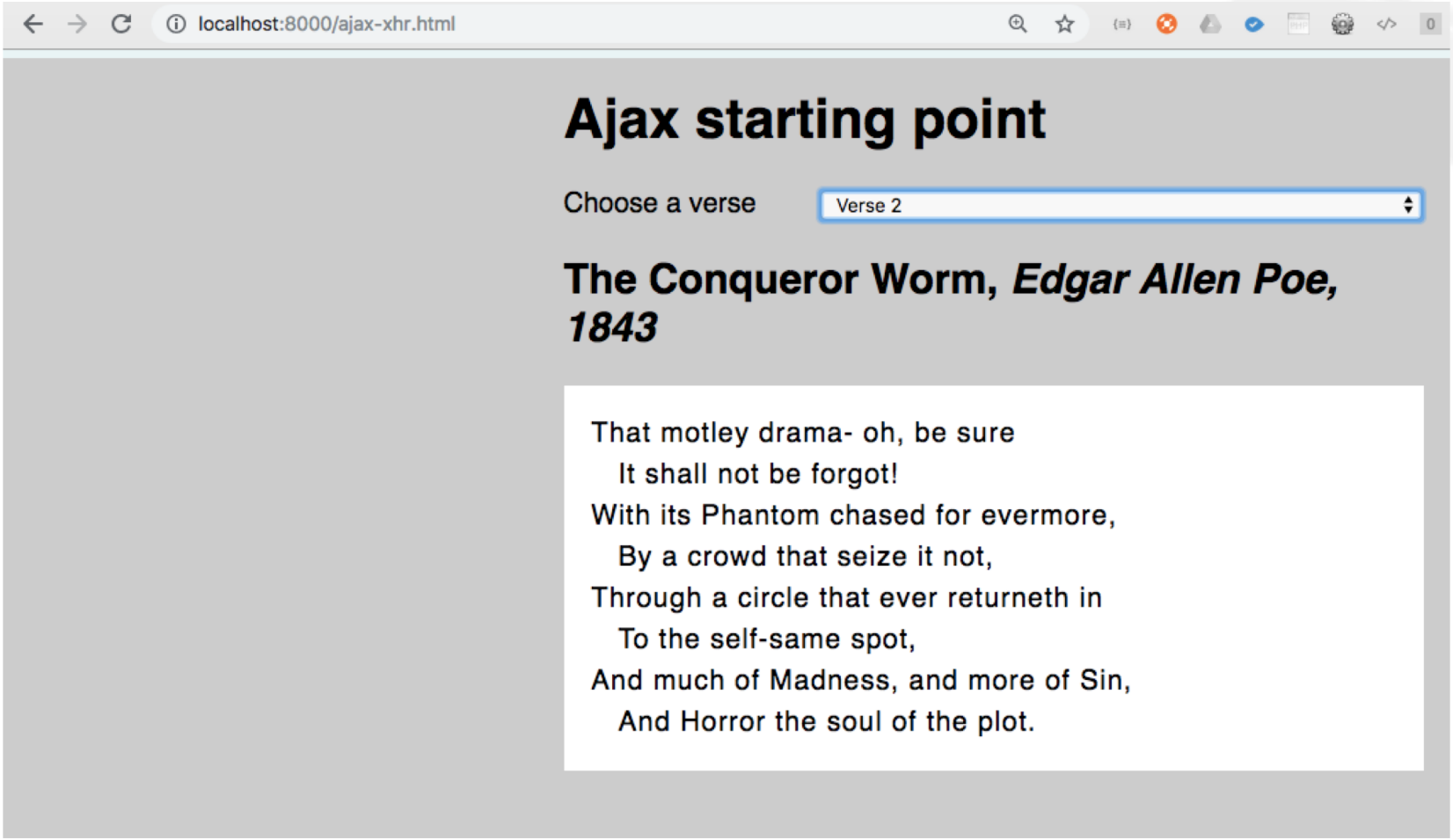
XMLHttpRequest

- XMLHttpRequest (which is frequently abbreviated to XHR) is a fairly old technology now
- It was invented by Microsoft in the late 1990's and has been standardized across browsers for quite a long time

Using XMLHttpRequest (1/2)



Using XMLhttpRequest (2/2)



XMLHttpRequest Tutorial (1/7)

- To begin this example, make a local copy of ajax-start.html and the four text files — verse1.txt, verse2.txt, verse3.txt, and verse4.txt — in a new directory on your computer
- In this example we will load a different verse of the poem (which you may well recognize) via XHR when it's selected in the drop down menu.

XMLHttpRequest Tutorial (2/7)

- Just inside the <script> element, add the following code
- This stores a reference to the <select> and <pre> elements in variables, and defines an onchange event handler function so that when the select's value is changed, its value is passed to an invoked function updateDisplay() as a parameter

```
1  var verseChoose = document.querySelector('select');
2  var poemDisplay = document.querySelector('pre');
3
4  verseChoose.onchange = function() {
5    var verse = verseChoose.value;
6    updateDisplay(verse);
7  };
```

XMLHttpRequest Tutorial (3/7)

- We'll start our function by constructing a relative URL pointing to the text file we want to load
  - so for example "Verse 1". The corresponding verse text file is "verse1.txt”
- Add the following lines inside your updateDisplay() function

```
1 verse = verse.replace(" ", "");
2 verse = verse.toLowerCase();
3 var url = verse + '.txt';
```

XMLHttpRequest Tutorial (4/7)

- To begin creating an XHR request, you need to create a new request object using the XMLHttpRequest() constructor

```
1 | var request = new XMLHttpRequest();
```

- Next, you need to use the open() method to specify what HTTP request method to use to request the resource from the network, and what its URL is.

```
1 | request.open('GET', url);
```

XMLHttpRequest Tutorial (5/7)

- Next, we'll set the type of response we are expecting — which is defined by the request's responseType property — as text

```
1 | request.responseType = 'text';
```

- Fetching a resource from the network is an asynchronous operation
  - You've got to wait for that operation to complete (e.g., the resource is returned from the network) before you can then do anything with that response, otherwise an error will be thrown

XMLHttpRequest Tutorial (6/7)

- XHR allows you to handle this using its onload event hander — this is run when the load event fires (when the response has returned)
- When this has occurred, the response data will be available in the response property of the XHR request object

```
request.onload = function() {
  poemDisplay.textContent = request.response;
};
```

- The above is all setup for the XHR request — it won't actually run until we tell it to, which is done using the send() method

```
request.send();
```

XMLHttpRequest Tutorial (7/7)

- One problem with the example as it stands is that it won't show any of the poem when it first loads
- To fix this, add the following two lines at the bottom of your code (just above the closing </script> tag) to load verse 1 by default, and make sure the <select> element always shows the correct value

```
1 updateDisplay('Verse 1');
2 verseChoose.value = 'Verse 1';
```

The Fetch API

- The Fetch API is basically a modern replacement for XHR
  - It was introduced in browsers recently to make asynchronous HTTP requests easier to do in JavaScript
  - Both for developers and other APIs that build on top of Fetch.

Using Fetch API

- Make a copy of your previous finished example directory
- If you didn't work through the previous exercise, create a new directory, and inside it make copies of xhr-basic.html and the four text files — verse1.txt, verse2.txt, verse3.txt, and verse4.txt

Replace XHR code with using fetch

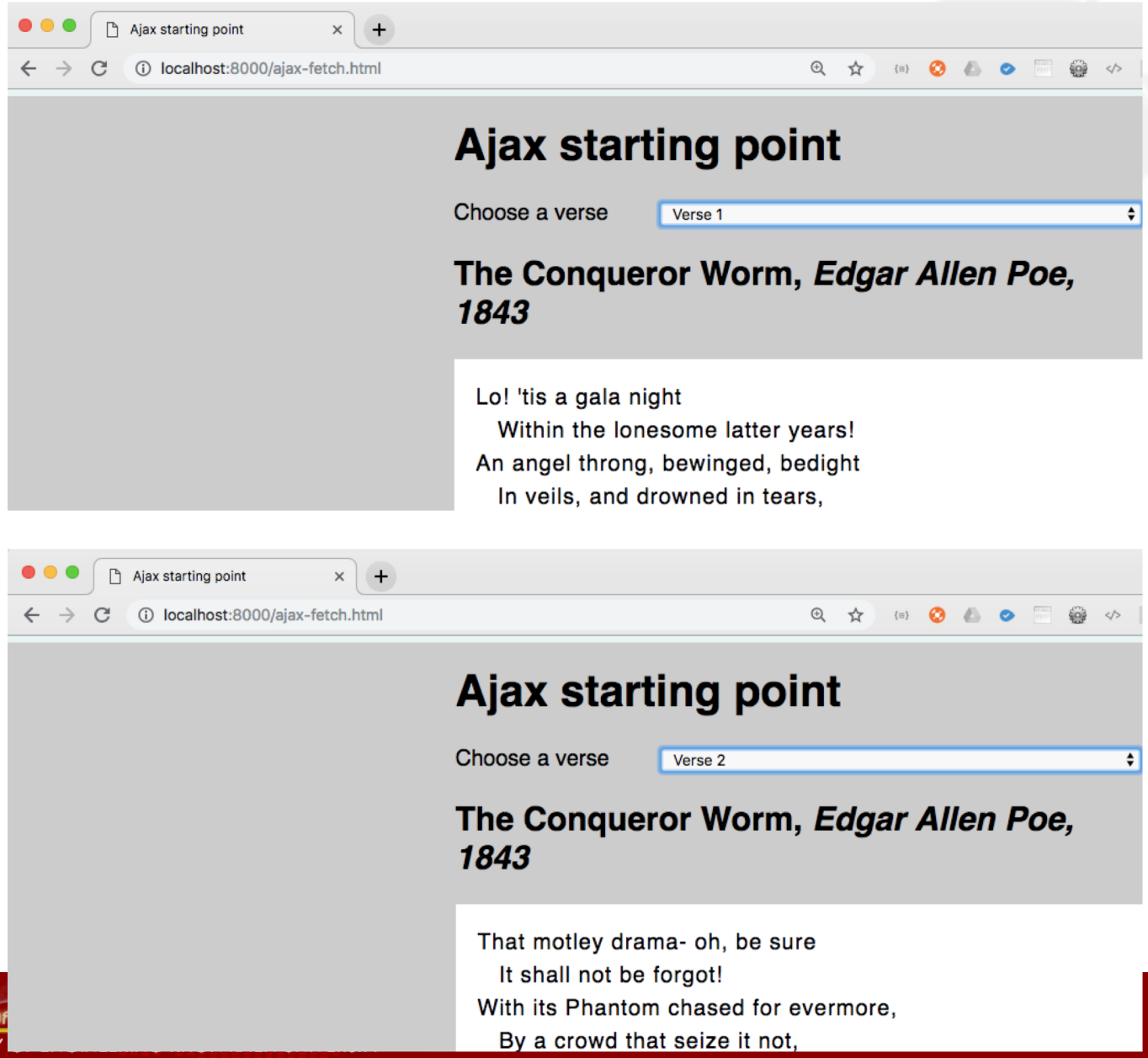
- Request using XHR code

```
var request = new XMLHttpRequest();
request.open('GET', url);
request.responseType = 'text';

request.onload = function() {
  poemDisplay.textContent = request.response;
};

request.send();
```

- Request using fetch

```
fetch(url).then(function(response) {
  response.text().then(function(text) {
    poemDisplay.textContent = text;
  });
});
```

The result of using fetch



What's going on in the fetch code? (1/4)

- First of all, we invoke the fetch() method, passing it the URL of the resource we want to fetch
- This is the modern equivalent of request.open() in XHR, plus you don't need any equivalent to .send()

```
71          fetch(url).then(function(response) {
72              response.text().then(function(text) {
73                  poemDisplay.textContent = text;
74              });
75          });
76          /*var request = new XMLHttpRequest();
77          request.open('GET', url);
78          request.responseType = 'text';
79          request.onload = function() {
80              poemDisplay.textContent = request.response;
81          };
82          request.send();*/
83      };
```

What's going on in the fetch code? (2/4)

- After that, you can see the .then() method chained onto the end of fetch()
  - This method is a part of Promises, a modern JavaScript feature for performing asynchronous operations
  - fetch() returns a promise, which resolves to the response sent back from the server
  - We use .then() to run some follow-up code after the promise resolves, which is the function we've defined inside it
  - This is the equivalent of the onload event handler in the XHR version

What's going on in the fetch code? (3/4)

- This function is automatically passed the response from the server as a parameter when the fetch() promise resolves
- Inside the function we grab the response and run its text() method, which basically returns the response as raw text
- This is the equivalent of request.responseType = 'text' in the XHR version.

What's going on in the fetch code? (4/4)

- You'll see that text() also returns a promise, so we chain another .then() onto it, inside of which we define a function to receive the raw text that the text()promise resolves to.
- Inside the inner promise's function, we do much the same as we did in the XHR version — set the <pre> element's text content to the text value.

Which mechanism should you use?

- This really depends on what project you are working on
- XHR has been around for a long time now and has very good cross-browser support
- Fetch and Promises, on the other hand, are a more recent addition to the web platform, although they're supported well across the browser landscape, with the exception of Internet Explorer
- If you need to support older browsers, then an XHR solution might be preferable. If however you are working on a more progressive project and aren't as worried about older browsers, then Fetch could be a good choice.

References

- https://developer.mozilla.org/en-US/docs/Learn/JavaScript/Client-side_web_APIs/Introduction
- Image source: Overloaded plug socket by The Clear Communication People, on Flickr.