Chapter 1: AJAX

What is AJAX

AJAX stands for **Asynchronous JavaScript and XML**, although XML has been replaced by JSON.

- Client Side
- Allows Javascript to make HTTP requests and use the results without redirecting the browser.
- Enables heavy-clients and lightweight webservices
- Can be used to avoid presentation responsibility on the webservice.
- JSON is a common replacement for XML
- Twitter.com is heavy on Ajax

AJAX is great an all but has its issues.

- You have to manage History, Back button, Bookmarks in JS
- Security: browsers heavily restrict AJAX to prevent abuse, has Same-Origin Policy
- Even more HTTP requests, consuming more CPU and RAM resources.

Making Requests

Use JS to make a HTTP request and get the content The old school way of doing that was to use:

```
new XMLHttpRequest()
```

The new way is to use the `Fetch` API

```
promise = new Request("some-url");
```

When you use the `Fetch` API it returns a `Promise` object which represents the eventual completion (or failure) of an asynchronous operation and its resulting value.

Promise

When you make a `new Request()` it returns a Request object, that Request object can be asynchronously executed by calling `fetch(request)` on the request. This will create a "pending" Promise object. Once created that request will be put into a queue waiting to be asynchronous processed. Once processed or rejected it will then run a callback function given as a argument of the `then()` method of the promise object.

`then()` can take in two callback functions, the first function being called on a successful operation, and the second function being called on a failed operation. Both callback functions can be optional, and are replaced with an identity function if not specified. The returning object from a `then()` method is another promise. This allows users to chain promises together.

Example

```
function makeAPromise(data) {
  return new Promise((resolve, reject) => {
    resolve(data);
  });
}
function promiseExample() {
  // .then returns a NEW promise
  var promise1 = makeAPromise("X");
  var promise2 = promise1.then((data2) => {
    // inner function gets called after the
    ↪ promise is resolved
    alert("data2: " + data2);
    return "Y";
  });
  // .then calls its argument with the return
  ↪ value of the
  //     previous promises's callback
  ↪ function
  var promise3 = promise2.then((data3) => {
    alert("data3: " + data3);
  });
}
```

In the `promiseExample()` the first promise is to resolve the value "X", then makes another promise to alert the user that data2 is X, where it will return the value "Y". After which is another promise to promise3 where it will alert the user with data3 is Y.

Promise dot-chaining

You can also dot-chain or cascade promises into one another.

```
  var request = new
  ↪ Request("images/fetch.gif");
  var promise1 = fetch(request);
  var promise2 =
  ↪ promise1.then(function(response) {
    console.log("Got headers!");
    return response.blob(); // return a
    ↪ promise for raw binary data
  });
  promise2.then(function(blob) {
    console.log("Got data blob!");
    var objectURL =
    ↪ URL.createObjectURL(blob);
    img.src = objectURL;
  });
}
```

is the same as

```
var request = new
↪ Request("images/fetch.gif");
fetch(request).then(function(response) {
  console.log("Got headers!");
  return response.blob(); // return a promise
  ↪ for raw binary data
}).then(function(blob) {
  console.log("Got data blob!");
  var objectURL = URL.createObjectURL(blob);
  img.src = objectURL;
});
```

Fetching JSON

This is a generic JSON GET function

```
function fetchJSON(url) {
  var request = new Request(url);
  return fetch(request).then((response) => {
    if (response.status === 200) { // OK
      return response.json(); // return a
      ↪ Promise
    } else {
      alert("Something went wrong: " +
      ↪ response.status);
    }
  });
}

var getterID; // global
// Get some JSON every second
function startGetting() {
  getterID = window.setInterval(() => { //
  ↪ callback
    var now = new Date();
    var s = 1 + (now.getSeconds() % 4); //
    ↪ remainder
    var url = s + ".json" // 1.json 2.json
    ↪ 3.json...
    fetchJSON(url).then((json) => { //
    ↪ another callback
      console.log(json); // browser turned
      ↪ the JSON into an object
      text = json.message; // it has
      ↪ properties
      document.querySelector("#ajaxy").
      ↪ innerText = text;
    });
  }, 1000); // 1 second or 1000ms
}
function stopGetting() {
  window.clearInterval(getterID);
}
```

This code will fetch a JSON object as a promise and return the content of the JSON as another promise to be process/read. The third promise is not consumed or used. The content of the JSON file is put into blockquotes using DOM manipulation. A new content is put into the blockquotes after a second or so.

```
<button type="button"
↪ onclick="startGetting()">Start
↪ Getting</button>
<button type="button"
↪ onclick="stopGetting()">Stop
↪ Getting</button>
<blockquote id="ajaxy"></blockquote>
```

Timers

- `window.setInterval` lets you run a function every so many milliseconds.
- `window.setTimeout` lets you run a function once after so many milliseconds.

Note
These timers are not accurate and are not designed to be accurate.

JSON

JSON
- JavaScript Object Notation
- Strict subset of JavaScript
- `JSON.parse` parses JSON text into an Object
- `JSON.stringify` turns an Object into JSON text

Example

```
function stringifyExample() {
  var obj = {
    "food": "hotdog",
    "condiments":
    ↪ ["ketchup","mustard","cheese"],
    "sausage": "weiner"
  };
  document.querySelector("#hotdog").value =
    JSON.stringify(obj, null, " "); // pretty
    ↪ print
}
function parseExample() {
  text =
  ↪ document.querySelector("#hotdog").value;
  var newObj = JSON.parse(text);

  ↪ document.querySelector("#sausage").innerText
  ↪ = newObj.sausage;
}
```

Design with AJAX

The design goal of AJAX is to minimize AJAX requests and traffic by bundling together request, etc. Don't hook into every event, use timeouts. This was use to ease page state transition.

What are your events?

- Per user input?
- Per user commit?
- Time based?
- Per Server action?
- Polling?
- Data?
- Content oriented?
- Messages?
- Multimedia?
- Read-based (reddit)

AJAX Observer Pattern

Observer Pattern

Observer pattern is where an observable keeps a collection of observers (listeners) and notifies those observers if anything changed by sending an update message.

This works great with AJAX if the observable is held client side in a browser and the observer is client side in the browser! Go ahead!

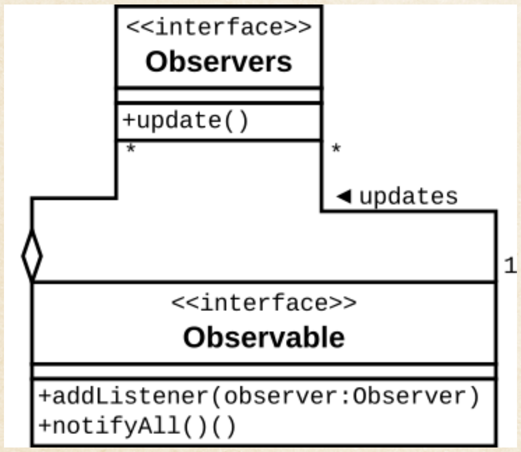

Figure 1.1: Observer Pattern UML

- Still works well with observable in browser and the observers server-side, the client simply notifies via the server's observers whenever an update occurs (but it should also communicate some lightweight state).
- Due to the lack of a client-side HTTP server it is problematic to do the observer pattern with client side observers.

Observing the Server

HTTP is stateless, so a client needs to communicate somehow all of the objects it is observing. Perhaps a serverside Observer proxy that stores observables for a client. Clients need to poll the server to check if there are updates. For the observer pattern to work this polling should allow the server to send update commands. Due to bandwidth concerns and latency concerns, an update from the server should probably include relevant data.

Fighting against:

- Latency
- Bandwidth
- Lack of communication channels
- Lack of ability to push messages to a client
- Polling
- Timer smashing

Solution?

- Polling: The most common
- Push API: Not supported on all browsers
- Comet "long polling": Difficult server-side support
- Websockets: need to make a websocket server.

Polling the Server

- Don't send too many requests
- Batch (bundle together) requests to the server
- Minimize the number of timers and the frequency of timers   E.g. if drawing, a user doesn't need more than 30FPS!
- Don't make requests until the previous request finished...
- Don't make requests you don't have to

Async and Await

The `async` and `await` are JavaScript keywords that makes asynchronous code easier to write and read afterwards.

- `async` functions returns a promise
- `await` in the async function blocks code execution (stop and wait for the promise to resolve).

Example

```
var getterID; // global
async function get2() {
  var now = new Date();
  var s = 1 + (now.getSeconds() % 4); //
  ↪ remainder
  var url = s + ".json" // 1.json 2.json
  ↪ 3.json...
  var json = await fetchJSON(url);
  console.log(json); // browser turned the
  ↪ JSON into an object
  var text = json.message; // it has
  ↪ properties
  document.querySelector("#ajaxy2").innerText
  ↪ = text;
}
function startGetting2() {
  getterID2 = window.setInterval(get2, 1000);
  ↪ // 1 second or 1000ms
}
function stopGetting2() {
  window.clearInterval(getterID2);
}

<button type="button"
↪ onclick="startGetting2()">Start
↪ Getting</button>
<button type="button"
↪ onclick="stopGetting2()">Stop
↪ Getting</button>
<blockquote id="ajaxy2"></blockquote>
```

Disadvantages

- Execution stops at await, instead of continuing in parallel, although if the browser supports threads

then the waiting execution will be threaded.
- with `.then(...)` in a loop, the loop completes instantly and each callback function can run in parallel as soon as its ready
- With `await` in a loop, the loop will keep stopping each time in gets to the await.

Review

AJAX.

- Asynchronous JavaScript and XML now JSON
- Allows JS to make HTTP request and use the result without redirecting
- A lot of websites uses AJAX
- The goal of using AJAX is to minimize AJAX requests and traffic by either bundling requests together, etc

XMLHttpRequest.

- The the old school of making async requests.
- You have to setup an `onreadystatechange()` function as an event handler and
- have to process a lot of things by yourself

Request.

- Is the new, modern way of doing things.
- It will handle all the state changes and event handlers and all you need to do is implement the promise's handler or callback function.
- `new Request("url")` will create a request object that is the interface for the Fetch API.
- Calling `fetch(request)` on the request object will create a pending promise object that will be executed asynchronously.
- If the promise successfully executes, it will run the callback function given by the promise's `.then(func)` method which will then return another promise.
- You can chain or cascade promises together by successively calling `then()` after each promise.
- You can also create a normal promise by calling `new Promise(func)` where `func` is a function with two callback function parameters.
- You can put promises on timers to execute them periodically or one after a certain period of time. The timers themselves are not accurate.

JSON.

- JavaScript Object Notation
- Strict subset of JS
- `JSON.parse(text)` will turn JSON `text` into a JS object
- `JSON.stringify(object)` will turn the JS `object` into JSON text.

AJAX observer pattern.

- Where an observable keeps a collection of observers (listeners) and notifies those observers if anything changed by sending an update message.
- Works great in AJAX if both observable and observers are client side.
- Works well if the observable is client side and the observers is the server.
- Works **not so well** if the observer is the client and the observable is the server due to the lack of a client-side HTTP server.
- For observing the server you can poll, push if the browser supports it, comet or use a websocket. But you have to deal with latency, bandwidth, etc.

Async and Await.

- `async` and `await` are JavaScript keywords that makes async code easier to write and read afterwards
- `async` on a function returns a promise
- `await` within an `async` function will block until the promises resolve.
- Issues is that it stops code execution with the await unless your browser supports threading. Big issue if the await is in a loop.