Controller/server communication

Mendel Rosenblum

Controller's role in Model, View, Controller

- Controller's job to fetch model for the view
  - May have other server communication needs as well (e.g. authentication services)
- Browser is already talking to a web server, ask it for the model
- Early approach: have the browser do a HTTP request for the model
  - First people at Microsoft liked XML so the DOM extension got called: `XMLHttpRequest`
- Allowed JavaScript to do a HTTP request without switching page
- Widely used and called **AJAX** - **A**synchronous **J**avaScript **a**nd **X**ML
- Since it is using an HTTP request it can carry XML or anything else
  - More often used with JSON

XMLHttpRequest

Sending a Request

```
xhr = new XMLHttpRequest();
xhr.onreadystatechange = xhrHandler;
xhr.open("GET", url);
xhr.send();
```

Any HTTP method (GET, POST, etc.) possible.

Responses/errors come in as events

Event handling

```
function xhrHandler(event) {
  // this === xhr
  if (this.readyState != 4) { // DONE
      return;
  }
  if (this.status != 200) { // OK
      return; // Handle error ...
  }
  ...
  let text = this.responseText;
  ...
```

XMLHttpRequest event processing

- Event handler gets called at various stages in the processing of the request

0	UNSENT	open() has not been called yet.
1	OPENED	send() has been called.
2	HEADERS_RECEIVED	send() has been called, and headers and status are available.
3	LOADING	Downloading; responseText holds partial data.
4	**DONE**	**The operation is complete**.

- Response available as:

  raw text - `responseText`

  XML document - `reponseXML`

- Can set request headers and read response headers

Traditional AJAX uses patterns

- Response is HTML

```
elem.innerHTML = xhr.responseText;
```

- Response is JavaScript

```
eval(xhr.responseText);
```

Neither of the above are the modern JavaScript framework way:

- Response is model data (JSON frequently uses here)

```
JSON.parse(xhr.responseText);
```

Fetching models with XMLHttpRequest

- Controller needs to communicate in the request what model is needed
- Can encode model selection information in request in:

URL path: `xhr.open("GET","userModel/78237489/fullname");`

Query params: `xhr.open("GET","userModel?id=78237489&type=fullname");`

Request body:

```
xhr.open("POST", url);
xhr.setRequestHeader("Content-type",
       "application/x-www-form-urlencoded");
xhr.send("id=78237489&type=fullname");
```

REST APIs

- REST - **re**presentational **s**tate **t**ransfer
- Guidelines for web app to server communications
- 2000 PhD dissertation that was highly impactful
  - Trend at the time was complex Remote Procedure Calls (RPCs) system
  - Became a must have thing: Do you have a REST API?
- Some good ideas, some not so good
  - Doesn't work for everything

Some RESTful API attributes

- Server should export **resources** to clients using unique names (**URIs**)
  - Example: `http://www.example.com/photo/` is a collection
  - Example: `http://www.example.com/photo/78237489` is a resource
- Keep servers "stateless"
  - Support easy load balancing across web servers
  - Allow caching of resources
- Server supports a set of HTTP methods mapping to **C**reate, **R**ead, **U**pdate, **D**elete (CRUD) on resource specified in the URL
  - GET method - Read resource (list on collection)
  - PUT method - Update resource
  - POST method - Create resource
  - DELETE method - Delete resource

REST API design

- Define the **resources** of the service and give them unique names (URIs)
  - Example: Photos, Users, Comments, ...
- Have clients use a CRUD operations using HTTP methods
- Extend when needed (e.g. transaction across multiple resources)

React accessing RESTful APIs

- React has no opinion. Prefer something higher level than `XMLHttpRequest`
  - Example: DoHTTPrequest(HTTP_METHOD, body, **doneCallback**)
- Popular: Axios - Promise based HTTP client for the browser and node.js
  - Wrapper around `XMLHttpRequest`
- REST Read (GET of URL): `result = axios.get(URL);`
- REST Create (POST to URL): `result = axios.post(URL, object);`
  - JSON encoding of `object` into body of POST request
- Similar patterns for REST Update (PUT) and REST Delete (DELETE)

Axios handling of HTTP responses

```
result = axios.get(URL); // Note: no callback specified!  It's a Promise

result.then((response) => {
                // response.status - HTTP response status (e.g. 200)
                // response.statusText - HTTP response status text (e.g. OK)
                // response.data - Response body object (JSON parsed)
            })
            .catch((err) => {
                    // err.response.{status, data, headers) - Non-2xxx status
                    // if !err.response - No reply, can look at err.request
            });
```

Minor Digression - Promises

Callbacks have haters - out of order execution

```
fs.ReadFile(fileName, function (error, fileData) {
    console.log("Got error", error, "Data", fileData);
});
console.log("Finished reading file");
```

What order to the console.log statements appear?

Callbacks have haters - Pyramid of Doom

```
fs.ReadFile(fileName, function (error, fileData) {
        doSomethingOnData(fileData, function (tempData1) {
            doSomethingMoreOnData(tempData1, function (tempData2) {
                finalizeData(tempData2, function (result) {
                    // Called Pyramid of Doom
                    doneCallback(result);
    }); }); }); });
```

- An alternative to pyramid: Have each callback be an individual function
  - Sequential execution flow jumps from function to function - not ideal

Same code without pyramid: Control jumps around

```
fs.ReadFile(fileName, readDone);

function readDone(error, fileData) {
    doSomethingOnData(fileData, doSomeDone);
}

function doSomeDone (someData) {
    doSomethingMoreOnData(someData, doSomeMoreDone);
}

function doSomeMoreDone (someMoreData) {
    finalizeData(someMoreData, doneCallback);
}
```

Idea behind promises

- Rather than specifying a done callback

  ```
  doSomething(args, doneCallback);
  ```

- Return a promise that will be filled in when done

  ```
  var donePromise = doSomething(args);
  ```

  `donePromise` will be filled in when operation completes

- Doesn't need to wait until you need the promise to be filled in
- Still using callbacks under the covers

`then()` - Waiting on a promise

- Get the value of a promise (waiting if need be) with **`then`**

```
donePromise.then(function (value) {
    // value is the promised result when successful
}, function (error) {
    // Error case
});
```

Example of Promise usage

- `axios.get()` returns a promise

```
axios.get(url).then(function(response) {
            var ok = (response.status === 200);
            doneCallback(ok ? response.data : undefined);
          }, function(response) {
            doneCallback(undefined);
          });
```

Promises

```
var myFile = myReadFile(fileName);
var tempData1 = myFile.then(function (fileData) {
    return doSomethingOnData(fileData);
});
var finalData = tempData1.then(function (tempData2) {
    return finalizeData(tempData2);
});
return finalData;
```

- Note no **Pyramid of Doom**
- Every variable is a promise
  - A standard usage: Every variable - If **thenable** call `then()` on it otherwise just use the variable as is.

Chaining promises

```
return myReadFile(fileName)
    .then(function (fileData) { return doSomethingOnData(fileData); })
    .then(function (data) { return finalizeData(data); })
    .catch(errorHandlingFunc);
```

- Add in ES6 JavaScript arrow functions:

```
return myReadFile(fileName)
    .then((fileData) => doSomethingOnData(fileData))
    .then((data) => finalizeData(data))
    .catch(errorHandlingFunc);
```

Going all in on promises

```
function doIt(fileName) {
    let file = ReadFile(fileName);
    let data = doSomethingOnData(file);
    let moreData = doSomethingMoreOnData(data);
    return finalizeData(moreData);
}
```

- All reads of variables become "then" calls:

`myVar` becomes `myVar.then( fn => ...`

Promises vs Callbacks

- Easy to go from Promise to Callback: Just call `.then(callbackFunc)`
  - `axios.get(url).then(callback)`
- Going from Callback to Promise requires creating a Promise

```
var newPromise = new Promise(function (fulfill, reject) {
    // calls fulfill(value) to have promise return value
    // calls reject(err) to have promise signal error
});
```

Converting callbacks to Promises

```
function myReadFile(filename) {
  return new Promise(function (fulfill, reject) {
    fs.readFile(filename, function (err, res) {
      if (err)
         reject(err);
      else
         fulfill(res);
    });
  });
}
```

Language support: `async` and `await` keywords

- async `function` - Declare a function to return a Promise

```
async function returnOne() { // returns a Promise
   return 1;
}
```

- `await` - Resolve the promise and returns its value

```
let one = await returnOne();
console.log(one);                    // Prints 1
```

- `await` only valid inside of `async function` functions

`async` and `await` makes it easier to use promises

```
async function doIt(fileName) {
    let file = await ReadFile(fileName);
    let data = await doSomethingOnData(file);
    let moreData = await doSomethingMoreOnData(data);
    return finalizeData(moreData);
}
```

- file, data, moreData can be regular variables, not forced to be promises
- doIt() does return a promise

End Digression

Other Transports: HTML5 WebSockets

- Rather than running over HTTP, HTML5 brings sockets to the browser
  - TCP connection from JavaScript to backend Web Server
- Event-based interface like XMLHttpRequest:

```
var socket = new WebSocket("ws://www.example.com/socketserver");
socket.onopen = function (event) {
  socket.send(JSON.stringify(request));
};

socket.onmessage = function (event) {
  JSON.parse(event.data);
};
```

Remote Procedure Call (RPC)

- Traditional distributed computation technology supporting calling of a function on a remote machine.
  - Browser packages function's arguments into a message to the web server.
  - Function is invoked with the arguments on the server.
  - Function's return value is sent back to the browser.
- Allows arbitrary code to be run on server - handles complex, multiple resource operations
  - Reduces number of round trip messages and makes failure handling easier.
- Can result in more complex to use interface compared to REST
  - Need to document the API (i.e. functions and calling sequence)
- RPC can be done over HTTP (e.g. POST) or WebSockets

Trending approach: GraphQL

- Standard protocol for backends from Facebook
  - Like REST, server exports resources that can be fetched by the web app
  - Unlike REST
    - Server exports a "schema" describing the resources and supported queries.
    - Client specifies what properties of the resource it is interested in retrieving.
    - Can fetch from many different resources in the same request (i.e. entire model in one query).
- Update operations specified in the exported schema
  - Allows an RPC-like interface
- Gaining in popularity particularly compared to REST
  - Gives a program accessible backend - **Application Programming Interface (API)**