EECS 368
Programming Language Paradigms

David O. Johnson

Fall 2022

Reminders

- Assignment 3 due: 11:59 PM, Monday, October 3
- Assignment 4 due: 11:59 PM, Monday, October 17

Any Questions?

In-Class Problem Solution

- 15-(9-28) In-Class Problem Solution.pptx

Any Questions?

Chapter 18 - HTTP and Forms

- ~~The protocol~~
- ~~Browsers and HTTP~~
- Fetch
  - Promises
- Security and HTTPS
- Form fields
- Focus
- Disabled fields
- The form as a whole
- Text fields
- Checkboxes and radio buttons
- Select fields
- File fields
- Storing data client-side

HTTP Client Requests

- Last time we learned how a client can use HTTP to access resources on a server.
- Specifically:
  - GET to get a resource from the server
  - DELETE to delete a resource on the server
  - PUT to create or replace a resource on the server
  - POST to send information to a resource on the server
- How do we make HTTP client requests in JavaScript?

JavaScript Client-Server Communication

- When building a system that requires communication between a JavaScript program running in the browser (client-side) and a program on a server (server-side), the most flexible approach is to build your communication around the concept of resources and HTTP methods.
- For example, to add a user to a system:
  - Use a PUT request to /users/larry …
  - encoding that user's properties (e.g., user ID, password, email) in a JSON document format (or use an existing format) that represents a user …
  - which is delivered in the body of the PUT request.
  - A resource is fetched by making a GET request to the resource's URL (for example, /user/larry), …
  - which again returns the document representing the resource.
- This approach makes it easier to use some of the features that HTTP provides, such as support for caching resources (keeping a copy on the client for fast access).

Fetch

- The interface through which browser JavaScript can make HTTP requests is called fetch.
- Since it is relatively new, it uses promises which is rare for browser interfaces.

- But what is a promise?

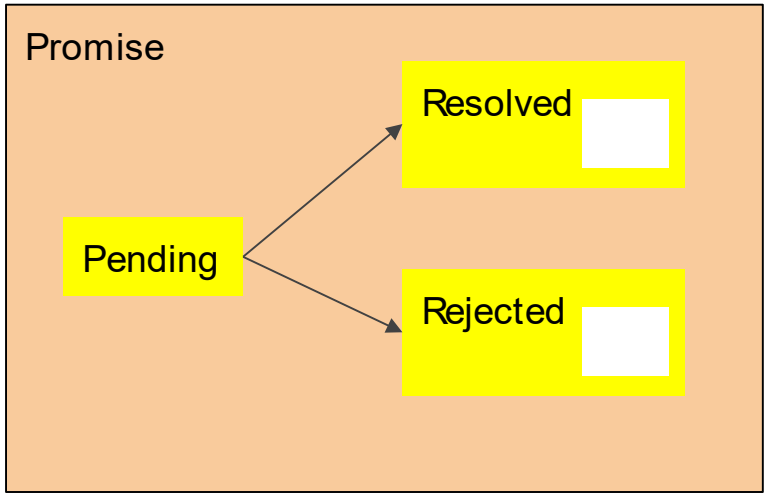
Promises

- A promise is an asynchronous action that may complete at some point and produce a value.
- It is able to notify anyone who is interested when its value is available.
- The standard class Promise is what this is for.
- The easiest way to create a promise is by calling Promise.resolve.
- This function ensures that the value you give it is wrapped in a promise.
- To get the result of a promise, you can use its then method.
- This registers a callback function to be called when the promise resolves and produces a value.

```
let fifteen = Promise.resolve(15);
fifteen.then(value => console.log(`Got ${value}`));
// → Got 15
```

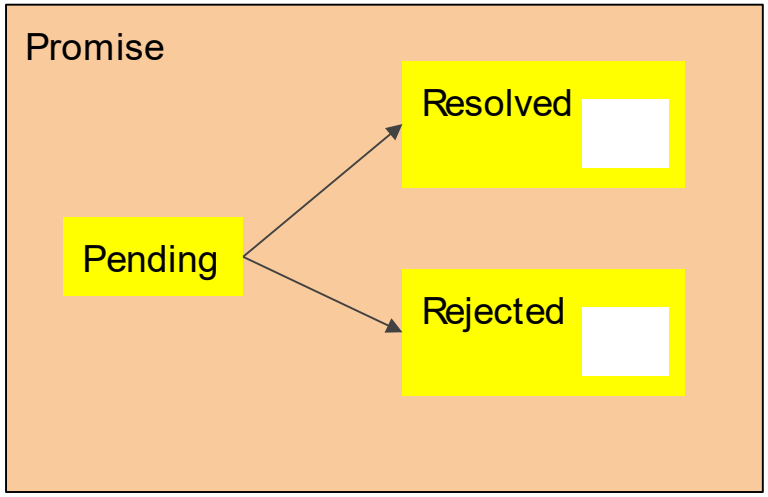
Rejected Promises

- Regular JavaScript computations can fail by throwing an exception.
- Asynchronous computations often need something like that.
- A network request may fail, or some code that is part of the asynchronous computation may throw an exception.
- Promises make this easy.
- They can be either:
  - resolved (the action finished successfully)
  - rejected (it failed)



Rejected Promises

- Resolve handlers (as registered with then) are called only when the action is successful.
- Rejections are automatically propagated to the new promise that is returned by then.
- And when a handler throws an exception, this automatically causes the promise produced by its then call to be rejected.
- So if any element in a chain of asynchronous actions fails, …
- …the outcome of the whole chain is marked as rejected,
- …and no success handlers are called beyond the point where it failed.



Rejected Promises

- Rejected promises are usually handled by printing out an error message (e.g., Bogus).
- The rejected promise handler is created with Promise.reject.
- The rejected promise handler is activated by the catch keyword.

```
let pf = Promise.resolve("Hello");
pf.then((o) => { console.log(o)});
// → "Hello"

let pr = Promise.reject("Bogus");
pr.then((o) => { console.log(o)});
pr.catch((o) => { console.log(o)});
// → "Bogus"

let pf2 = Promise.resolve("Hello");
pf2.then((o) => {
      console.log("resolved",o)
}).catch((o) => {
     console.log("rejected",o)
}
// → resolved "Hello"
```

Promise Chains

- As a shorthand, then also accepts a rejection handler as a second argument, so you can install both types of handlers in a single method call.
- A function passed to the Promise constructor receives a second argument, alongside the resolve function, which it can use to reject the new promise.

```
new Promise((_, reject) => reject(new Error("Fail")))
  .then(value => console.log("Handler 1"))
  .catch(reason => {
    console.log("Caught failure " + reason);
    return "nothing";
  })
  .then(value => console.log("Handler 2", value));
// → Caught failure Error: Fail
// → Handler 2 nothing
```

Promise Chains

- The chains of promise values created by calls to then and catch can be seen as a pipeline through which asynchronous values or failures move.
- Since such chains are created by registering handlers, each link has a success handler or a rejection handler (or both) associated with it.
- Handlers that don't match the type of outcome (success or failure) are ignored.
- But those that do match are called, and their outcome determines what kind of value comes next:
  - Success when it returns a non-promise value
  - Rejection when it throws an exception
  - Outcome of a promise when it returns one of those.

- Much like an uncaught exception is handled by the environment, JavaScript environments can detect when a promise rejection isn't handled and will report this as an error.

Promise Chains

```
function wait(n,v) {
    return new Promise((resolve,reject) => {
        setTimeout(() => {
            resolve(v)},n);
    })
}

wait(1000,22).then(o => {
  console.log("got to first then with", o)
  return wait(1000,o + o)
}).then(o => {
  console.log("got to second then with",o)
}).catch(o => {
  console.log("rejected with",o)
})
```

```
… waiting …
got to first then with 22
… waiting …
got to second then with 44
```

Promise Chains

```
function wait(n,v) {
    return new Promise((resolve,reject) => {
        setTimeout(() => {
            resolve(v)},n);
    })
}

wait(1000,22).then(o => {
  console.log("got to first then with", o)
  return Promise.reject("Bogus")
}).then(o => {
  console.log("got to second then with",o)
}).catch(o => {
  console.log("rejected with",o)
})
```

… waiting …
got to first then with 22
rejected with Bogus

Any Questions?

Fetch Example

- As we said earlier, the interface through which browser JavaScript can make HTTP requests is called fetch.
- fetch uses promises.

- Suppose we want to retrieve a file from ku.edu called /example/data.txt, which contains some example data we want to display in a text box in a web browser.

- The HTTP request from the client and the response from the server might look something like this:

Request from client

```
GET /example/data.txt 1.1
Host: ku.edu
User-Agent: Your browser's name
```

Response from server

```
HTTP/1.1 200 OK
Content-Length: 31
Content-Type: text/plain
Last-Modified: Mon, 08 Jan 2018 10:29:45 GMT

This is the content of data.txt
```

Fetch

Request from client

```
GET /example/data.txt 1.1
Host: ku.edu
User-Agent: Your browser's name
```

Response from server

```
HTTP/1.1 200 OK
Content-Length: 31
Content-Type: text/plain
Last-Modified: Mon, 08 Jan 2018 10:29:45 GMT

This is the content of data.txt
```

```
fetch('http://ku.edu/example/data.txt').then(response => {
  console.log(response.status);
  // → 200
  console.log(response.headers.get("Content-Type"));
  // → text/plain
});
```

- Calling fetch returns a promise that resolves to a Response object holding information about the server's response, such as its status code and its headers.
- The headers are wrapped in a Map-like object that treats its keys (the header names) as case insensitive because header names are not supposed to be case sensitive.
- This means headers.get("Content-Type") and headers.get("content-TYPE") will return the same value.

Fetch

Request from client

```
GET /example/data.txt 1.1
Host: ku.edu
User-Agent: Your browser's name
```

Response from server

```
HTTP/1.1 200 OK
Content-Length: 31
Content-Type: text/plain
Last-Modified: Mon, 08 Jan 2018 10:29:45 GMT

This is the content of data.txt
```

```
fetch('http://ku.edu/example/data.txt').then(response => {
  console.log(response.status);
  // → 200
  console.log(response.headers.get("Content-Type"));
  // → text/plain
});
```

- Note that the promise returned by fetch resolves successfully even if the server responded with an error code.
- It might also be rejected if there is a network error or …
- if the server that the request is addressed to can't be found.

Fetch URL

```
fetch('http://ku.edu/example/data.txt').then(response => {
  console.log(response.status);
  // → 200
  console.log(response.headers.get("Content-Type"));
  // → text/plain
});
```

- The first argument to fetch is the URL that should be requested.
- When that URL doesn't start with a protocol name (such as http:), it is treated as relative, which means it is interpreted relative to the current document. For example:

  example/data.txt

- When it starts with a slash (/), it replaces the current path, which is the part after the server name. For example:

  /example/data.txt

Getting Actual Content of a Response

- To get at the actual content of a response, you can use its text method.
- Because the initial promise is resolved as soon as the response's headers have been received …
- and because reading the response body might take a while longer, …
- this again returns a promise.

Response from server

```
HTTP/1.1 200 OK
Content-Length: 31
Content-Type: text/plain
Last-Modified: Mon, 08 Jan 2018 10:29:45 GMT

This is the content of data.txt
```

```
fetch("example/data.txt")
  .then(resp => resp.text())
  .then(text => console.log(text));
// → This is the content of data.txt
```

- A similar method, called json, returns a promise that resolves to the value you get when parsing the body as JSON …
- or rejects if it's not valid JSON.

Using other HTTP Methods with Fetch

- By default, fetch uses the HTTP GET method to make its request and does not include a request body.
- You can configure it differently by passing an object with extra options as a second argument.
- For example, this request tries to delete example/data.txt:

```
fetch("example/data.txt", {method: "DELETE"}).then(resp => {
  console.log(resp.status);
  // → 405
});
```

Request from client

```
DELETE example/data.txt 1.1
Host: ku.edu
User-Agent: Your browser's name
```

Response from server

```
HTTP/1.1 405 Method Not Allowed
```

- The 405 status code means “method not allowed”, …
- an HTTP server’s way of saying “I can’t do that”.
- You can also use this technique to execute the other HTTP methods, PUT and POST.

Adding a Body or Headers

- To add a request body, you can include a body option.
- To set headers, there's the headers option.
- For example, this request includes a Range header, which instructs the server to return only part of a response.

```
fetch("example/data.txt", {headers: {Range: "bytes=8-18"}})
   .then(resp => resp.text())
   .then(console.log(resp.text));
// → the content
```

Request from client

```
GET example/data.txt 1.1
Host: ku.edu
User-Agent: Your browser's name
Range: bytes=8-18
```

Response from server

```
HTTP/1.1 200 OK
Content-Length: 11
Content-Type: text/plain
Last-Modified: Mon, 08 Jan 2018 10:29:45 GMT

the content
```

Adding Headers

- The browser will automatically add some request headers, such as “Host” and those needed for the server to figure out the size of the body.
- But adding your own headers is often useful to include things such as …
- authentication information …
- or to tell the server which file format you’d like to receive.

Adding Multiple Headers

- Multiple headers can be added as follows:

```
fetch("example/data.txt", {headers: {Range: "bytes=8-19"},
                                     {User-Agent: "Your browser's name"}
                           })
   .then(resp => resp.text())
   .then(console.log(resp.text));
```

Request from client

```
GET example/data.txt 1.1
Host: ku.edu
Range: bytes=8-19
User-Agent: Your browser's name
```

Any Questions?

Security and HTTPS

- Data traveling over the Internet tends to follow a long, dangerous road.
- To get to its destination, it must hop through anything from coffee shop Wi-Fi hotspots to networks controlled by various companies and states.
- At any point along its route it may be inspected or even modified.

- If it is important that something remain secret, such as the password to your email account, or that it arrive at its destination unmodified, such as the account number you transfer money to via your bank's website, plain HTTP is not good enough.
- The secure HTTP protocol, used for URLs starting with https://, wraps HTTP traffic in a way that makes it harder to read and tamper with.
- Before exchanging data, the client verifies that the server is who it claims to be by asking it to prove that it has a cryptographic certificate issued by a certificate authority that the browser recognizes.
- Next, all data going over the connection is encrypted in a way that should prevent eavesdropping and tampering.

Security and HTTPS

- Thus, when it works right, HTTPS prevents other people from impersonating the website you are trying to talk to and from snooping on your communication.
- It is not perfect, and there have been various incidents where HTTPS failed because of forged or stolen certificates and broken software, but it is a lot safer than plain HTTP.

Any Questions?

In-Class Problem

Write a JavaScript program to send the following HTTP command from a client to a server:

```
GET /18_http.html HTTP/1.1
Host: eloquentjavascript.net
User-Agent: Your browsers name
```

Use the appropriate fetch parameters to explicitly specify the method and headers.