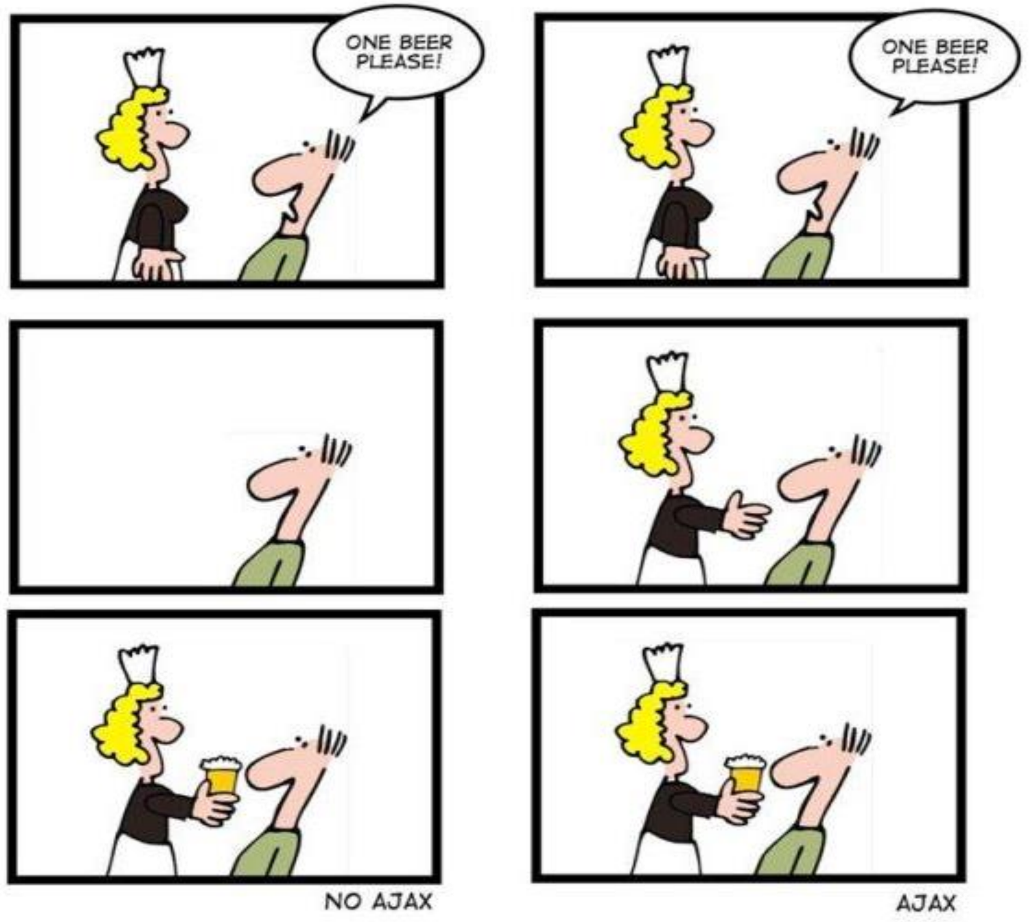
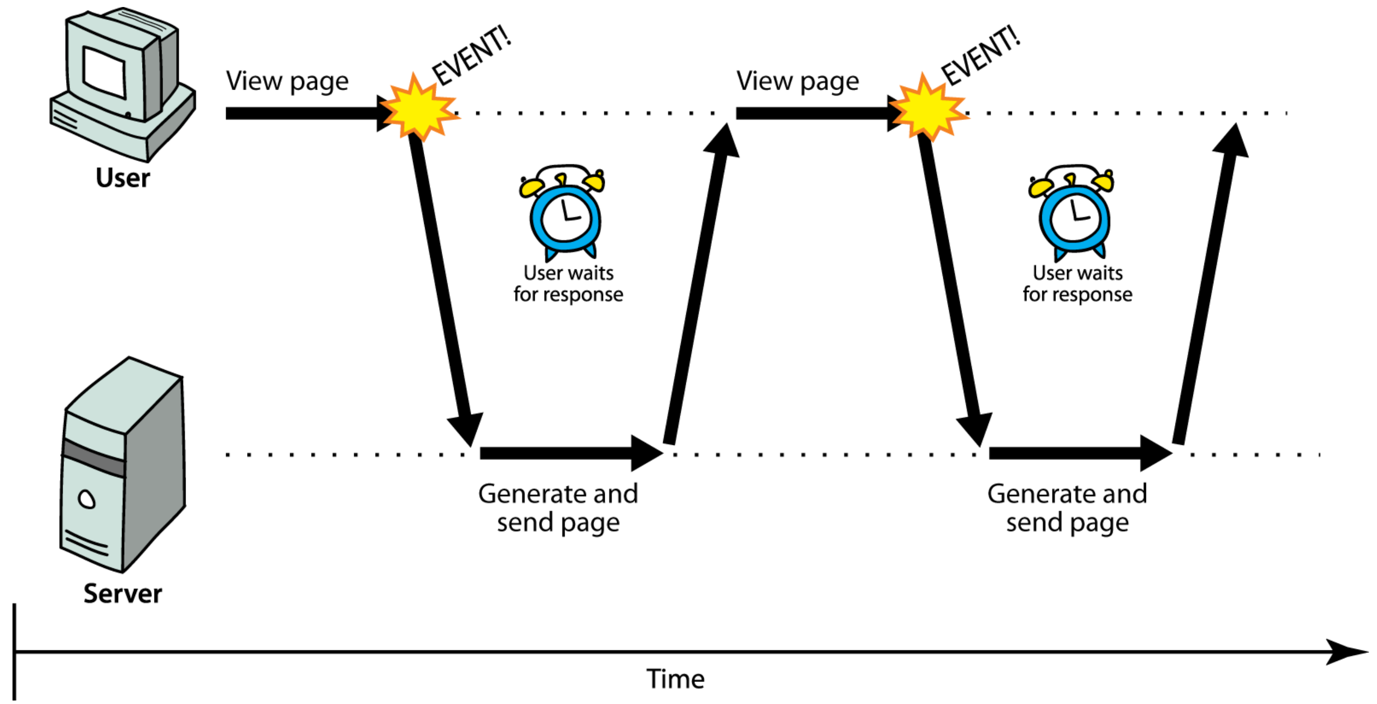
CSc 337

LECTURE 12: THE FETCH API AND AJAX

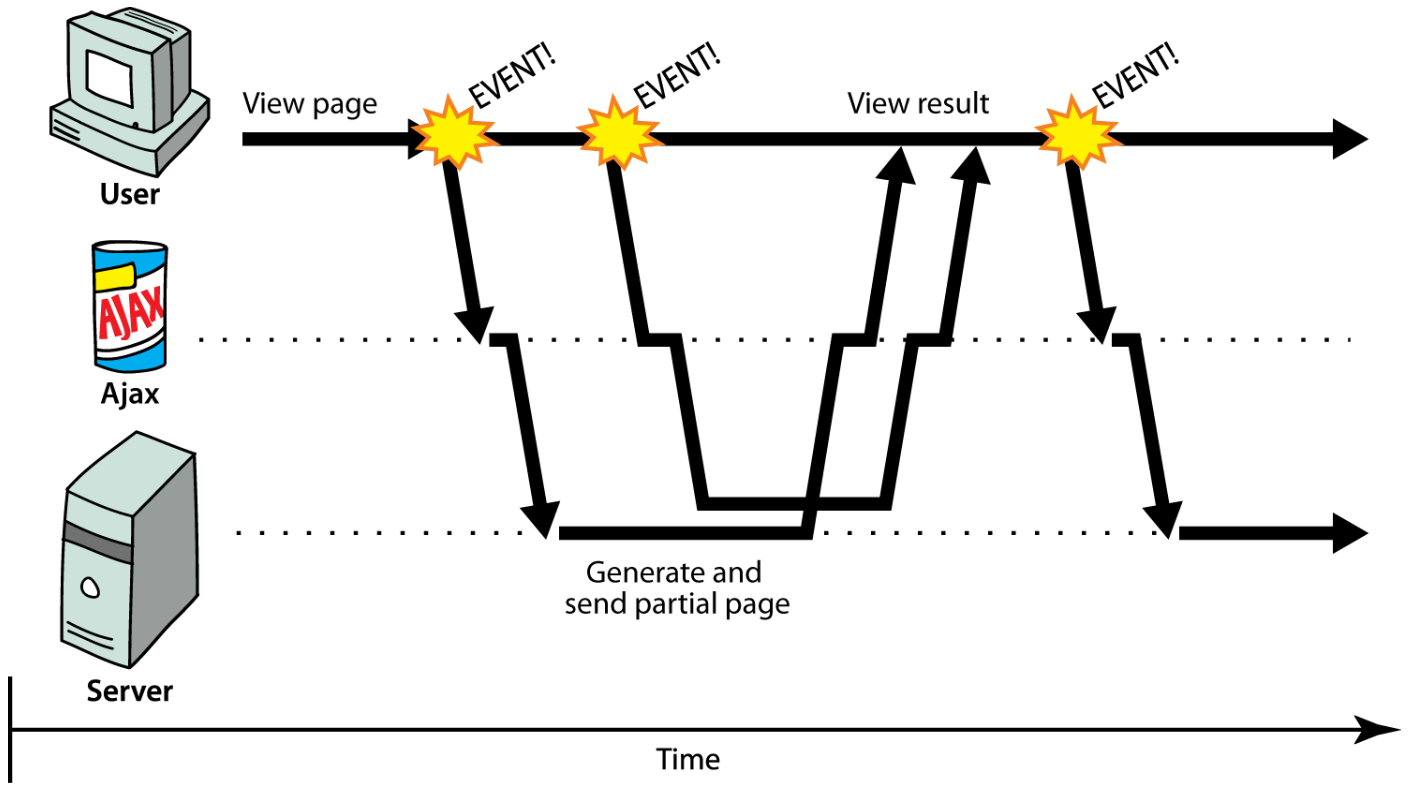


Synchronous web communication



- **synchronous**: user must wait while new pages load
    - the typical communication pattern used in web pages (click, wait, refresh)

Asynchronous web communication



- **asynchronous**: user can keep interacting with page while data loads

Why synchronized requests suck

- your code waits for the request to completely finish before proceeding
- easier for you to program, but ...
  - the user's *entire browser LOCKS UP* until the download is completed
  - a terrible user experience (especially if the page is very large or slow to transfer)

- better solution: use an *asynchronous request* that notifies you when it is complete
  - this is accomplished by learning about the event properties of `XMLHttpRequest`

The Old Way: Ajax

- **Ajax**: Asynchronous JavaScript and XML
  - not a programming language; a particular way of using JavaScript
  - downloads data from a server in the background
  - allows dynamically updating a page without making the user wait
  - avoids the "click-wait-refresh" pattern
  - examples: Diff Tool, CodeStepByStep; Google Suggest

The New Way: Promises

Promise: A JS object that executes some code that has an uncertain outcome

Promises have three states:

- Pending
- Fulfilled
- Rejected

Example: “I promise to post homework 4”
Pending: Not yet posted
Fulfilled: Homework 4 posted
Rejected: Wrong homework posted, or not posted in time

Why are they better

- Help deal with uncertainty
- You determine whether it is fulfilled or rejected
- You define what happens when it fulfills or is rejected

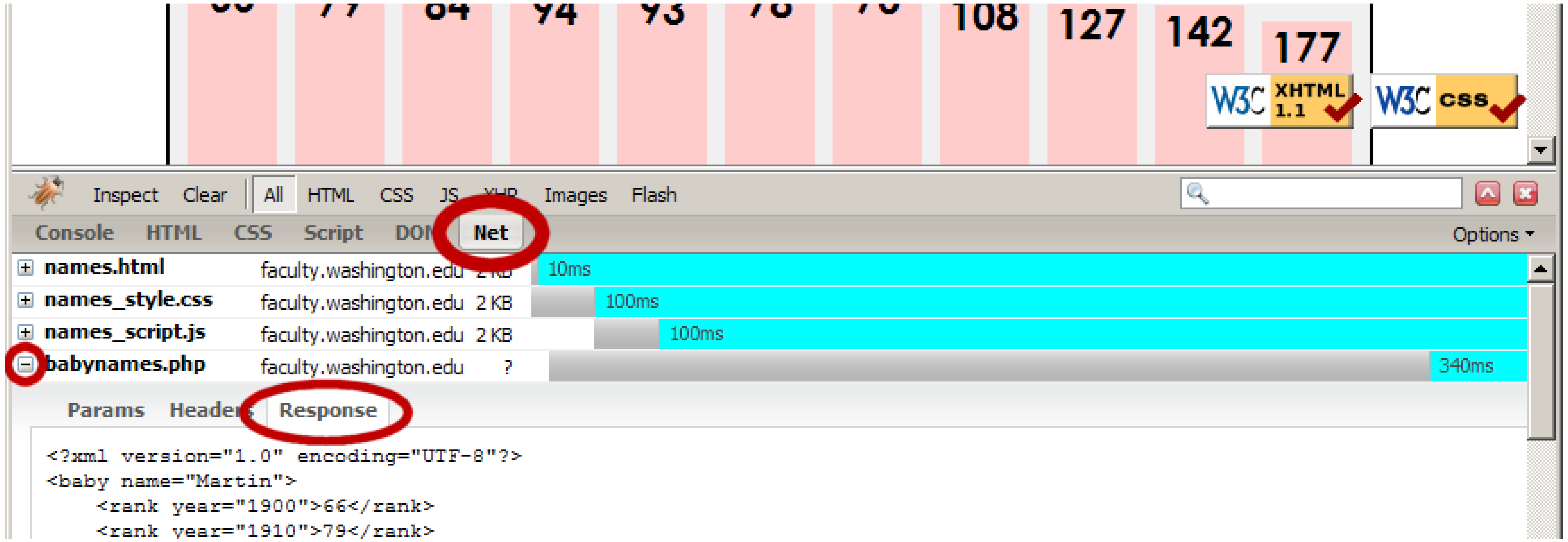
Promise syntax

```
function callAjax(){
  var url = ..... // put url string here
  fetch(url)
    .then(checkStatus)
    .then(function(responseText) {
         //success: do something with the responseText
     })
    .catch(function(error) {
        //error: do something with error
  });
}
```

Promise syntax

```
function checkStatus(response) {
    if (response.status >= 200 && response.status < 300) {
        return response.text();
    } else {
        return Promise.reject(new Error(response.status+":
                                           "+response.statusText));
    }
}
```

Debugging Ajax code



```
names.html          faculty.washington.edu  2 KB  10ms
names_style.css     faculty.washington.edu  2 KB  100ms
names_script.js     faculty.washington.edu  2 KB  100ms
babynames.php       faculty.washington.edu  ?     340ms
```

```
<?xml version="1.0" encoding="UTF-8"?>
<baby name="Martin">
    <rank year="1900">66</rank>
    <rank year="1910">79</rank>
```

- Firebug **Net** tab (or Chrome's Network tab) shows each request, parameters, response, errors
- expand a request with **+** and look at **Response** tab to see Ajax result
- check **Console** tab for any errors that are thrown by requests

Exercise: Using Node.js

Node.js

We will be using node.js to run our server code (The code that generates what your fetch requests get)

Download node.js: https://nodejs.org/en/

Double click on the downloaded file and follow the instructions to install it.

Running node.js

You will need a command line to run node.js

Window: you can use the built in command prompt, PowerShell (recommended), or download a command line like Cygwin

Mac: there is a nice built in command line. Just search for "console"

Linux: there is a nice built in command line

Testing that node is installed

Open up your command line and type the following:

```
node -v
```

The version number of node on your machine should be output. If nothing is output or if there is an error node didn't install correctly.

Installing Express

Run the following to install Express:

```
npm install express
```

Node server

Copy the following code into a file and name it `server1.js`

```
// CSC 337 hello world server
const express = require("express");
const app = express();
app.use(express.static('public'));
app.get('/', function (req, res) {
     res.header("Access-Control-Allow-Origin", "*");
    res.send('Hello World!');
})
app.listen(3000);
```

Running the node.js server

You will need to navigate to the directory containing the file you created from the code on the last slide. To move between directories you can use:

`cd` <directory>

`cd` will move you into the directory specified. If you want to move into the parent directory, use `..` as the directory name.

If you want to see what directories exists in the directory you are in use `ls` to list them

Running the node.js server

Once you have gotten to the directory that contains your server code, you can run it with the following command:

```
node service1.js
```

You will not see anything appear on the command line. To see it running open your browser and type the following in the address bar:

```
http://localhost:3000/
```

Writing code to fetch from the server

Write HTML and JavaScript to, when a button on the page is clicked, fetch data from the web server and inject that data into the page.