Lesson 11

JSON, APIs and Ajax

Half Way!

- Thank you for the kind remarks that were in the reviews.
- And thank you for the constructive criticisms in the reviews as well.

Learning Objectives

- Know what JSON is and what it’s used for.
- Review HTTP and understand what HTTP verbs are, especially GET.
- Describe how to make calls to an API and consume data from it.
- Make AJAX requests using the native DOM functions.
- Make AJAX requests using jQuery.
- Access public APIs and get data back.

Intro to JSON

- JSON is a lightweight text-based data interchange format based on JavaScript.
- JSON stands for “JavaScript Object Notation”
- It’s text and it looks like JS, so both humans and computers find it easy to read and generate

Intro to JSON

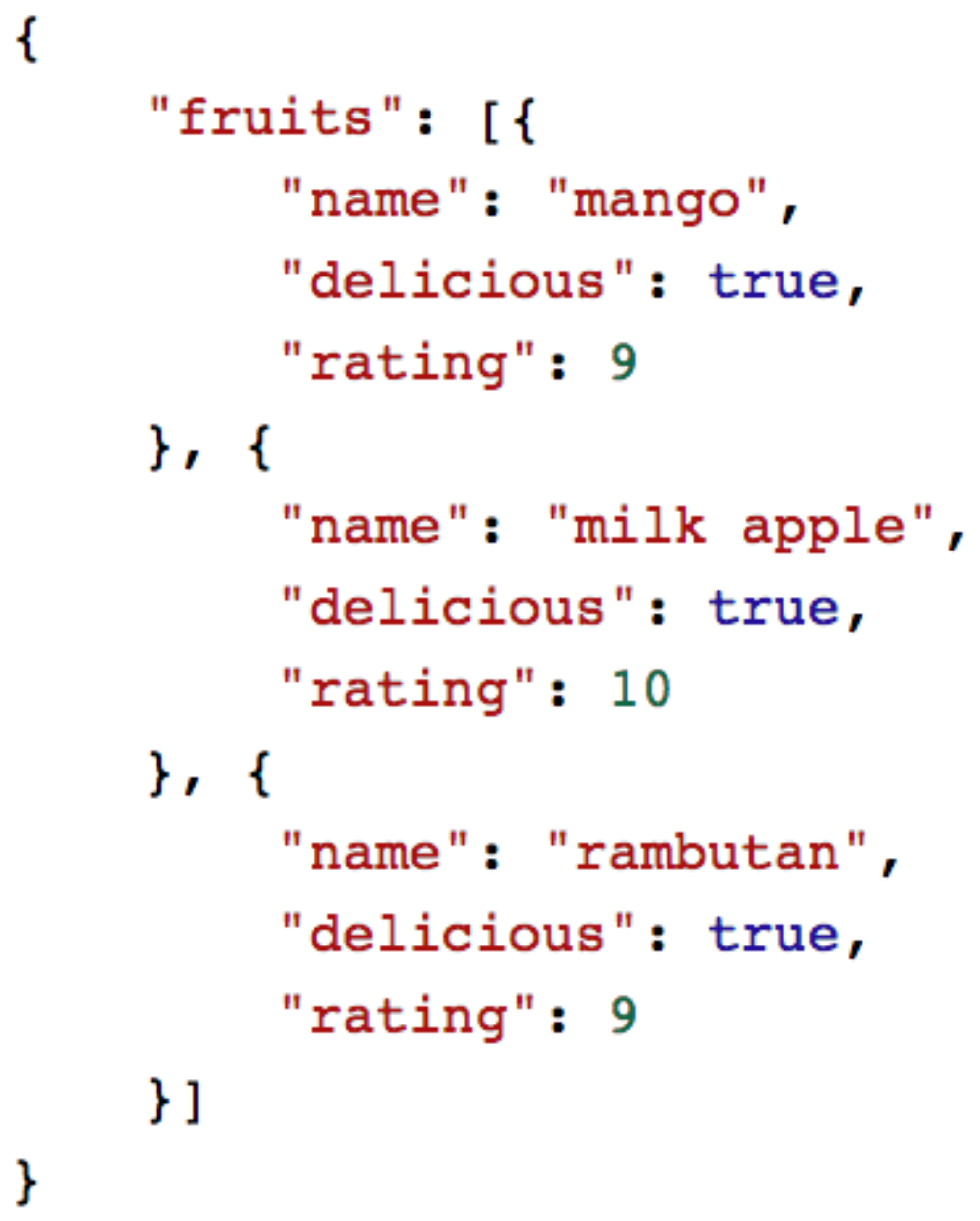
- We use JSON to transfer data between applications (like between a JavaScript program running on our browser, and a server on the Internet)
- For consistency, the rules of how JSON can be formatted are STRICTER than they are in JavaScript programs

JSON Syntax rules

- **All strings must be double-quoted.**
- **Property names in objects must also be double-quoted strings.**
- **No comments!**
- Trailing commas are forbidden.
- Leading zeroes are prohibited.
- In numbers, a decimal point must be followed by at least one digit.
- Most characters are allowed in strings; however, certain characters (such as ', ", \, and newline/tab) must be 'escaped' with a preceding backslash (\) in order to be read as characters.

JSON example

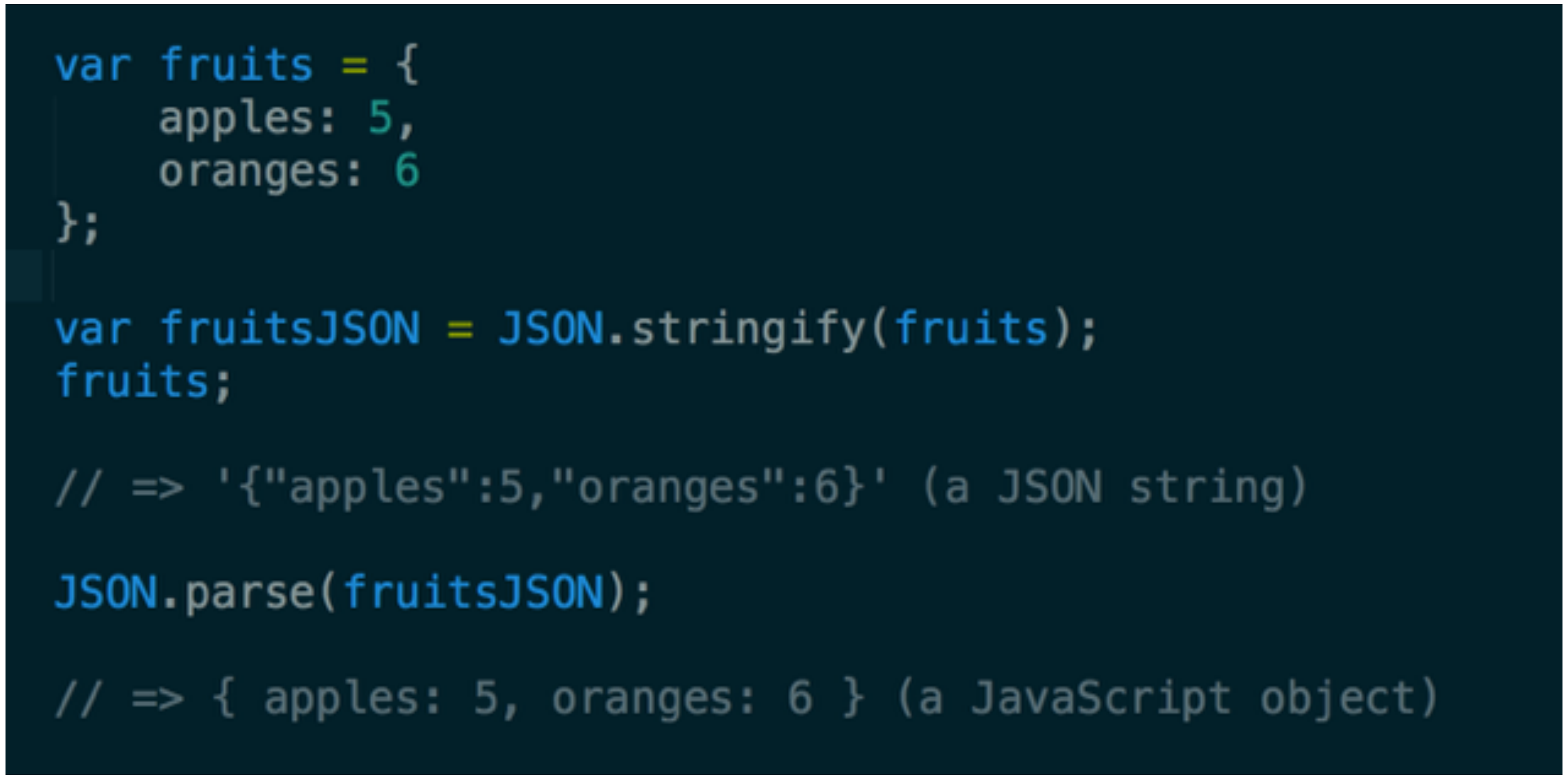
- Instead of “JavaScript **Object** Notation”, should be called “JavaScript **Data** Notation”, because you can represent all kinds of datatypes with it — objects, arrays, strings, booleans and numbers



```
{
    "fruits": [{
        "name": "mango",
        "delicious": true,
        "rating": 9
    }, {
        "name": "milk apple",
        "delicious": true,
        "rating": 10
    }, {
        "name": "rambutan",
        "delicious": true,
        "rating": 9
    }]
}
```

Converting from JS to JSON and back again

- Use the global object **JSON**, and its methods
  - **JSON.stringify** (JS => JSON)
  - **JSON.parse** (JSON => JS)



```
var fruits = {
    apples: 5,
    oranges: 6
};

var fruitsJSON = JSON.stringify(fruits);
fruits;

// => '{"apples":5,"oranges":6}' (a JSON string)

JSON.parse(fruitsJSON);

// => { apples: 5, oranges: 6 } (a JavaScript object)
```

JSON codealong

- http://bit.ly/jsdev2-lesson11-json-codealong
- We'll be copying and pasting stuff from this file into the Chrome console.

Practice on your own (5 minutes)

- In the Chrome console, create a regular JavaScript object and assign it to a variable, **myObject**.
- Run **JSON.stringify** on **myObject**, and assign the result to the variable **myJSONString**.
- Print **myJSONString** by typing “myJSONString” into the console.
- Run **JSON.parse** on **myJSONString**. Is the result the same thing you started with, **myObect**?

Intro to APIs

- “Application Programming Interface”
- When a server has a structured way for you to request data from it, and use it in your program, that’s called an API.

Intro to APIs

- Real life scenarios of consuming an API:
    - The Twitter application on your phone
    - The“Login with Facebook” button on some websites.
    - Feed readers!

Calling an API

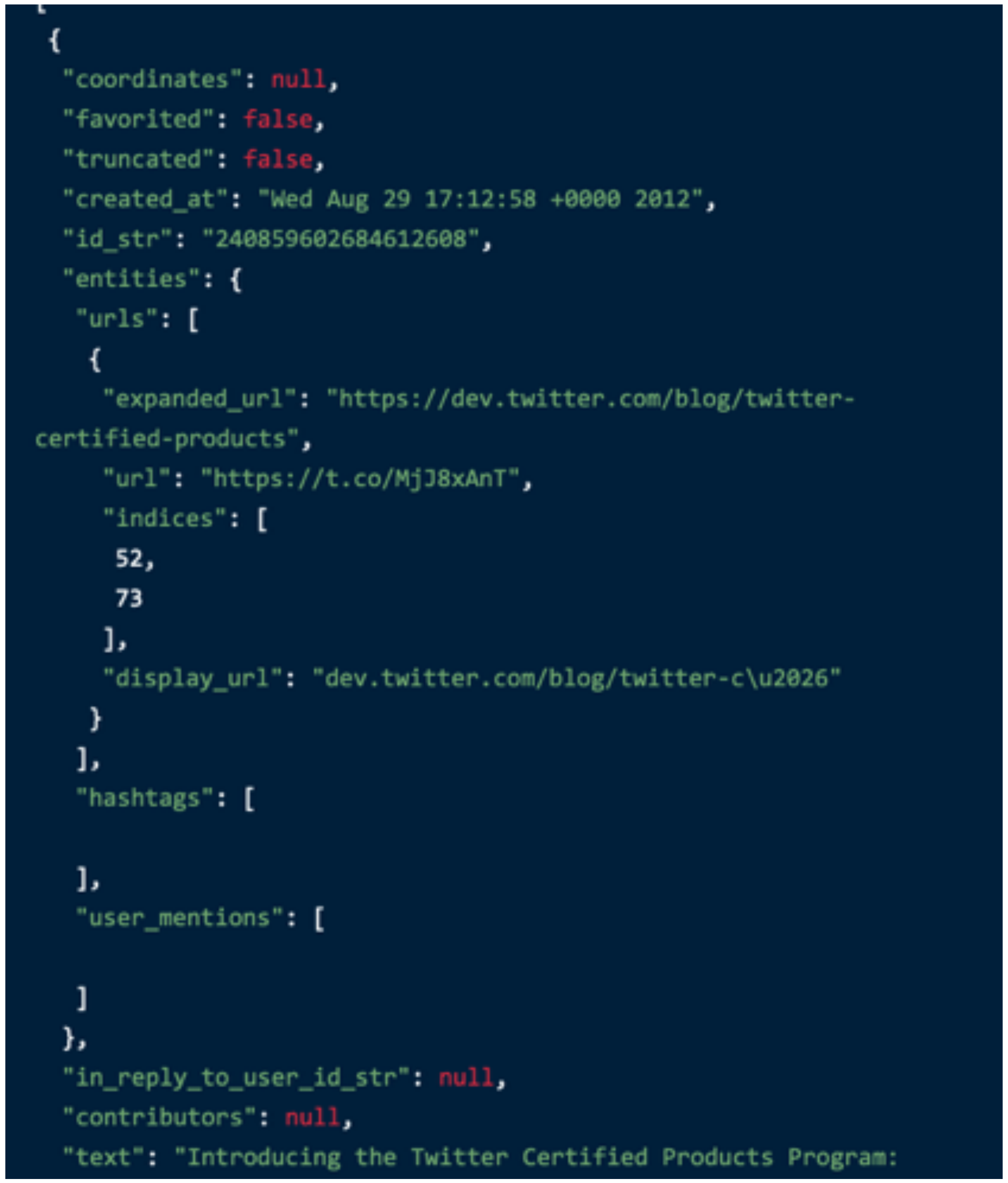
- When a client interacts with an API, it’s referred to as “**calling**”.

- **Calling** an API most of the time is an HTTP request (the same kind of request that browsers use to retrieve HTML pages).

- But instead of **HTML**, the API will usually return data in the **JSON** format.

Twitter for example

- This is part of the response you get from Twitter when retrieving your timeline.
- It gives you very nicely formatted JSON.
- JSON is convenient for both humans and computers to digest.



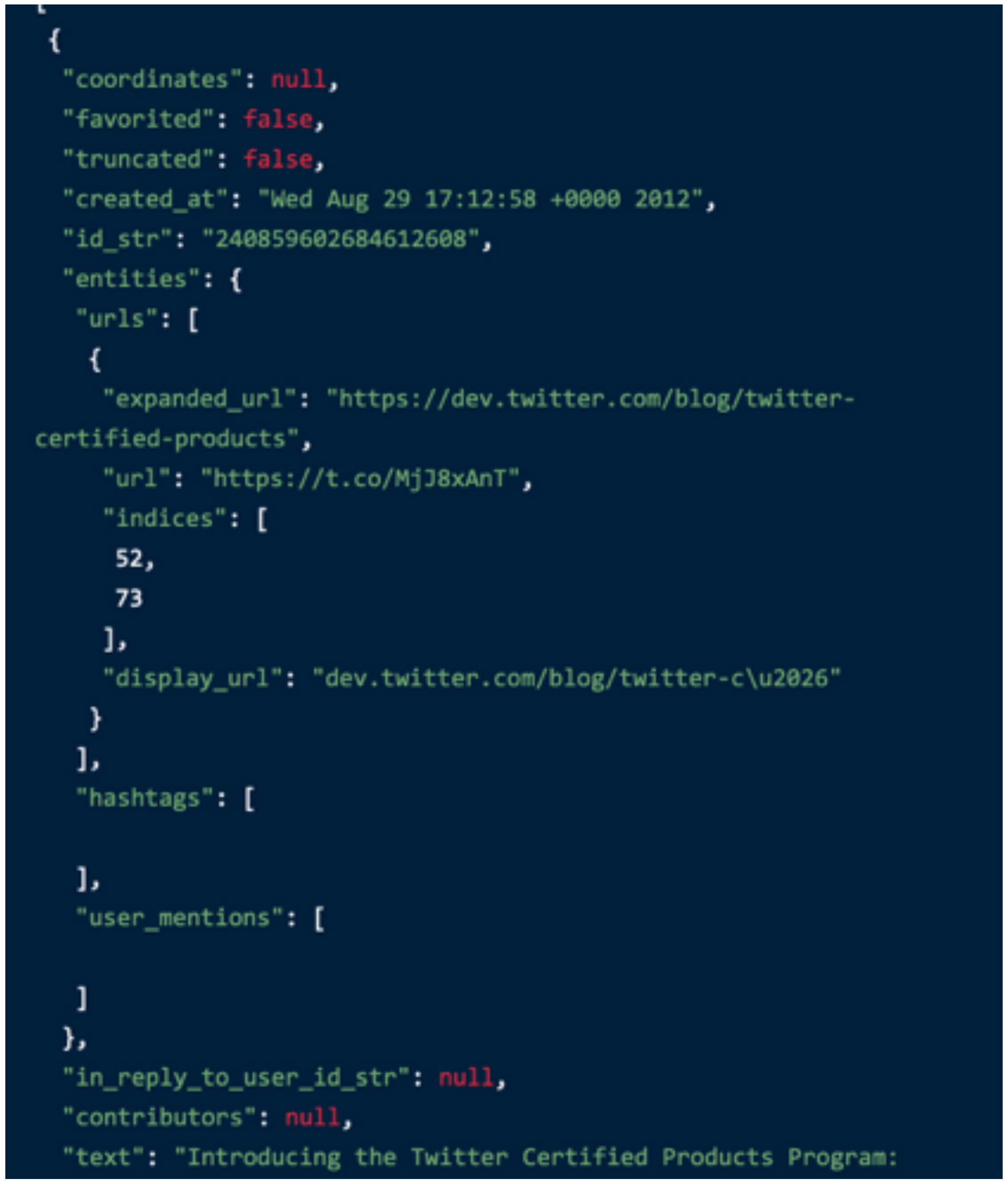
```
  {
   "coordinates": null,
   "favorited": false,
   "truncated": false,
   "created_at": "Wed Aug 29 17:12:58 +0000 2012",
   "id_str": "240859602684612608",
   "entities": {
    "urls": [
     {
      "expanded_url": "https://dev.twitter.com/blog/twitter-
certified-products",
      "url": "https://t.co/MjJ8xAnT",
      "indices": [
       52,
       73
      ],
      "display_url": "dev.twitter.com/blog/twitter-c\u2026"
     }
    ],
    "hashtags": [

    ],
    "user_mentions": [

    ]
   },
   "in_reply_to_user_id_str": null,
   "contributors": null,
   "text": "Introducing the Twitter Certified Products Program:
```

Twitter for example

- This API call requires something called "**Authentication**".
- Typically, you would use a **key,** a long string comprised of letters and numbers.
- You would make then perform an **API call** with the key to retrieve your timeline.



```
 {
  "coordinates": null,
  "favorited": false,
  "truncated": false,
  "created_at": "Wed Aug 29 17:12:58 +0000 2012",
  "id_str": "240859602684612608",
  "entities": {
   "urls": [
    {
     "expanded_url": "https://dev.twitter.com/blog/twitter-
certified-products",
     "url": "https://t.co/MjJ8xAnT",
     "indices": [
      52,
      73
     ],
     "display_url": "dev.twitter.com/blog/twitter-c\u2026"
    }
   ],
   "hashtags": [

   ],
   "user_mentions": [

   ]
  },
  "in_reply_to_user_id_str": null,
  "contributors": null,
  "text": "Introducing the Twitter Certified Products Program:
```

HTTP review

- HTTP is a protocol — an agreed-upon method for transferring information over the Internet
- HTTP requests and responses consist of text: a bunch of header lines, and a body.
- You will *never* write a raw HTTP request. Your browser writes them for you.

HTTP Review

- The **client** (your browser) is the one that sends a **request** to the **server** — the client “calls” the server
- The **server** sends a **response** to the **client**
- But actually… What we call the “web server” is usually a middleman between the client and something called the “web application”

Web Applications

- Web applications typically do the heavy lifting of a request.
- For example, logging in a user by communicating with a database of some sort.
- Posting a comment of some sort.
- Basically any *dynamic* feature of a website, is probably hitting a web application, such as Ruby on Rails, Django, Express, etc.

An analogy

- Think of a restaurant…
  - You are a **web browser** (the client) sitting at a table.
  - You request from the **server**, “I would like the Foie Gras please”.
  - The server says “Absolutely”, and then gives the request to the kitchen **(web application)** for them to fulfill the request and respond with food.

Codealong

Codealong: HTTP

- Open your Chrome developer tools and follow along…

HTTP wrapup

- Request methods:
    - **GET** => requests data from the server (which the server will often get from a database)
    - **POST** => sends new data to the server (to be stored in a database, usually)
    - **PUT** => tells the server to update existing data
    - **DELETE** => tells the server to delete some data
- Today we'll mostly be using **GET**

HTTP wrapup

- Common response status codes that might come down from a server:

Code	Description	What the server is saying
200	**OK**	Here you go; here's what you requested!
304	**Not modified**	We're not going to send you another copy of the thing you just downloaded five minutes ago.
400	**Bad Request**	The way you formatted your request was not what we were expecting.
403	**Forbidden**	You are not allowed to receive the thing that you requested.
404	**Not Found**	Your request made sense and you're allowed to make it, but we can't find what you're looking for.
500	**Internal Server Error**	Wait, did somebody just unplug the machine our web application was running on?

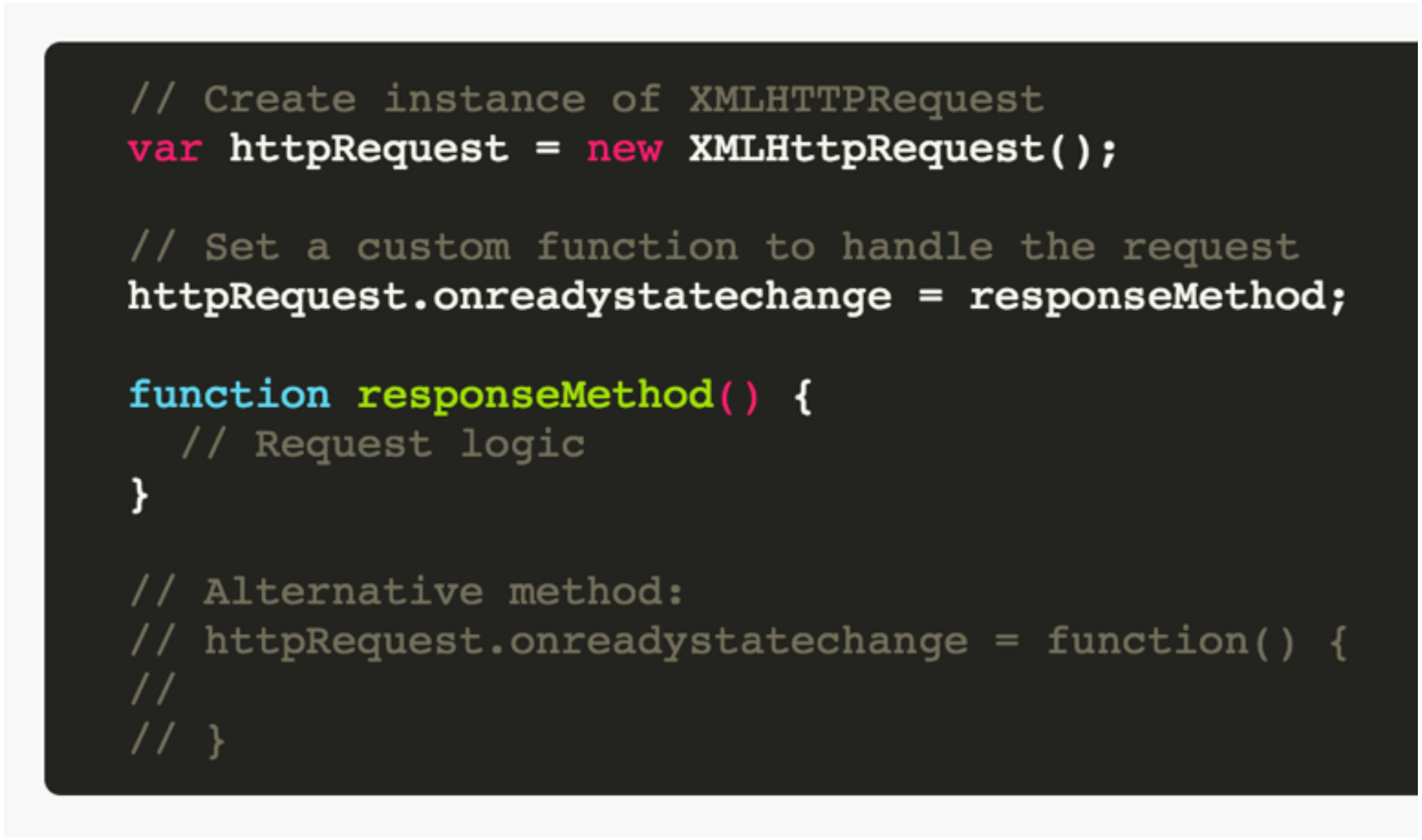
AJAX

- **AJAX** stands for “**A**synchronous **J**avaScript **a**nd **X**ML”
    - (Although it’s more **AJAJ** these days — **A**synchronous **J**avaScript **a**nd **J**SON)
- **AJAX** is the backbone of dynamic web pages updating from server calls.
- Facebook, Twitter, Google, all rely on **AJAX** to give you rich, dynamic websites.

AJAX is Asynchronous

- AJAX allows us to communicate with servers **asynchronously**.
- This means we make a request and then we go on about our business, running other code, while we wait for the response in the background.
- This all happens without any page refresh.

How to make these calls



```
// Create instance of XMLHTTPRequest
var httpRequest = new XMLHttpRequest();

// Set a custom function to handle the request
httpRequest.onreadystatechange = responseMethod;

function responseMethod() {
  // Request logic
}

// Alternative method:
// httpRequest.onreadystatechange = function() {
//
// }
```

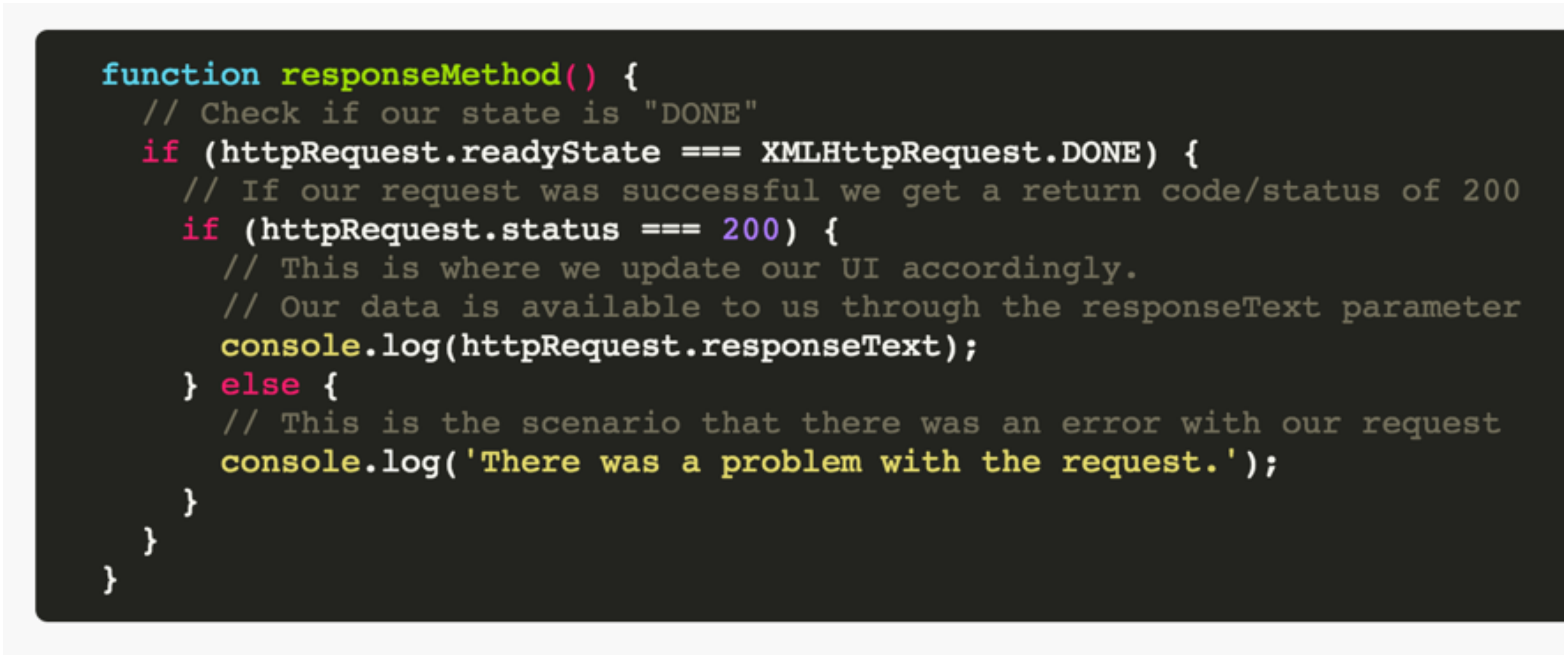
Keeping an eye on the State

- The previous slide tells how to perform the request and check when the “state” of it has changed.

- This is only the beginning, we still have to check if the the “status” of the new state tells us that the AJAX call has been **successful**!

- If there’s been an error, we should also notify the user of that fact.

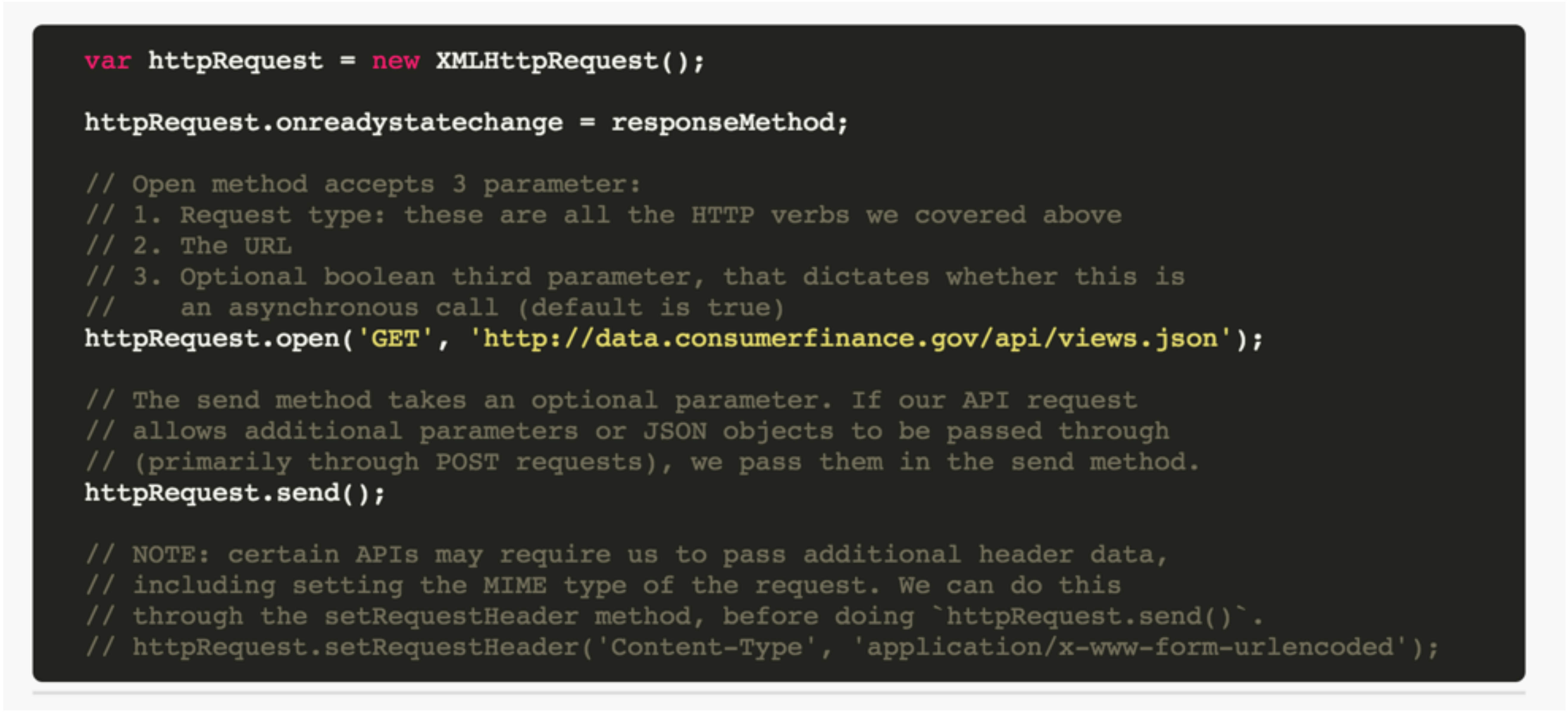
What to do with the response



```
function responseMethod() {
  // Check if our state is "DONE"
  if (httpRequest.readyState === XMLHttpRequest.DONE) {
    // If our request was successful we get a return code/status of 200
    if (httpRequest.status === 200) {
      // This is where we update our UI accordingly.
      // Our data is available to us through the responseText parameter
      console.log(httpRequest.responseText);
    } else {
      // This is the scenario that there was an error with our request
      console.log('There was a problem with the request.');
    }
  }
}
```

- First check if the state changed to “done”
- If so and it’s ok (status code 200), log the responseText
- If it was not ok (not 200), sound the alarm.

Finally…



```
var httpRequest = new XMLHttpRequest();

httpRequest.onreadystatechange = responseMethod;

// Open method accepts 3 parameter:
// 1. Request type: these are all the HTTP verbs we covered above
// 2. The URL
// 3. Optional boolean third parameter, that dictates whether this is
//    an asynchronous call (default is true)
httpRequest.open('GET', 'http://data.consumerfinance.gov/api/views.json');

// The send method takes an optional parameter. If our API request
// allows additional parameters or JSON objects to be passed through
// (primarily through POST requests), we pass them in the send method.
httpRequest.send();

// NOTE: certain APIs may require us to pass additional header data,
// including setting the MIME type of the request. We can do this
// through the setRequestHeader method, before doing `httpRequest.send()`.
// httpRequest.setRequestHeader('Content-Type', 'application/x-www-form-urlencoded');
```

1. Tell the XHR object what to do when something changes.

2. Tell it which URL to call, using which HTTP method.

3. **send()** it on its way and move on to other things while waiting for the response.

Codealong

- **cd** to **~/GA-JS**
- Create a **lesson11** folder
- **cd** into it
- create **ajax.js** and **ajax.html** files
- **subl .**
- **(Note: jquery link is https://code.jquery.com/jquery-2.2.4.js)**

AJAX Lab 1

- Instructions and code at http://bit.ly/jsdev2-lesson11-ajax-exercise-1

AJAX with jQuery

- jQuery provides the ability to very easily make AJAX requests with simple methods.

- This is usually what you'll see in the wild because it removes a lot of the boiler plate required for using XMLHTTPRequest.

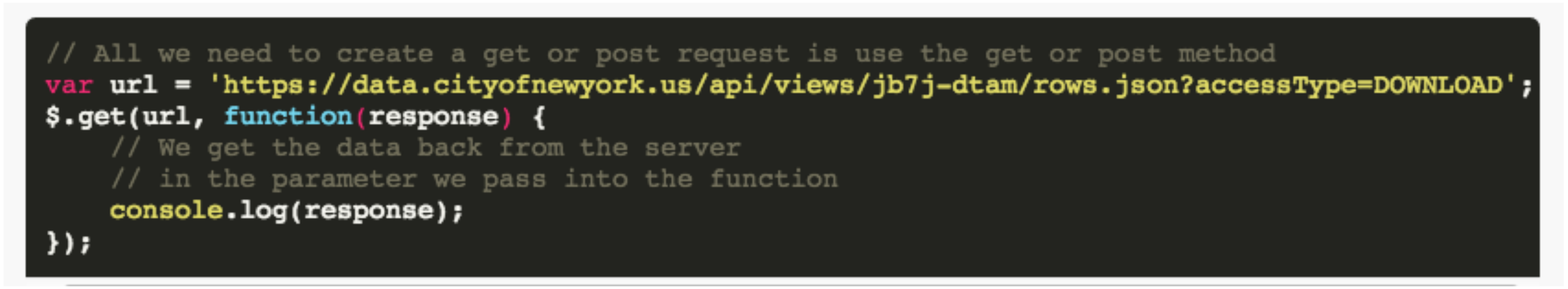
- The jQuery way of doing it is extremely similar to the next-generation DOM API way of doing it, called 'fetch', which we won't be going into but is the wave of the future, AJAX-wise

AJAX with jQuery

- Simple! Handles JSON conversion for you.

- Using **$.get**, you give it a url and a function to be run once the request completes, and it does a request using the HTTP "GET" method.

- The **response** parameter of the callback function will contain the response body.

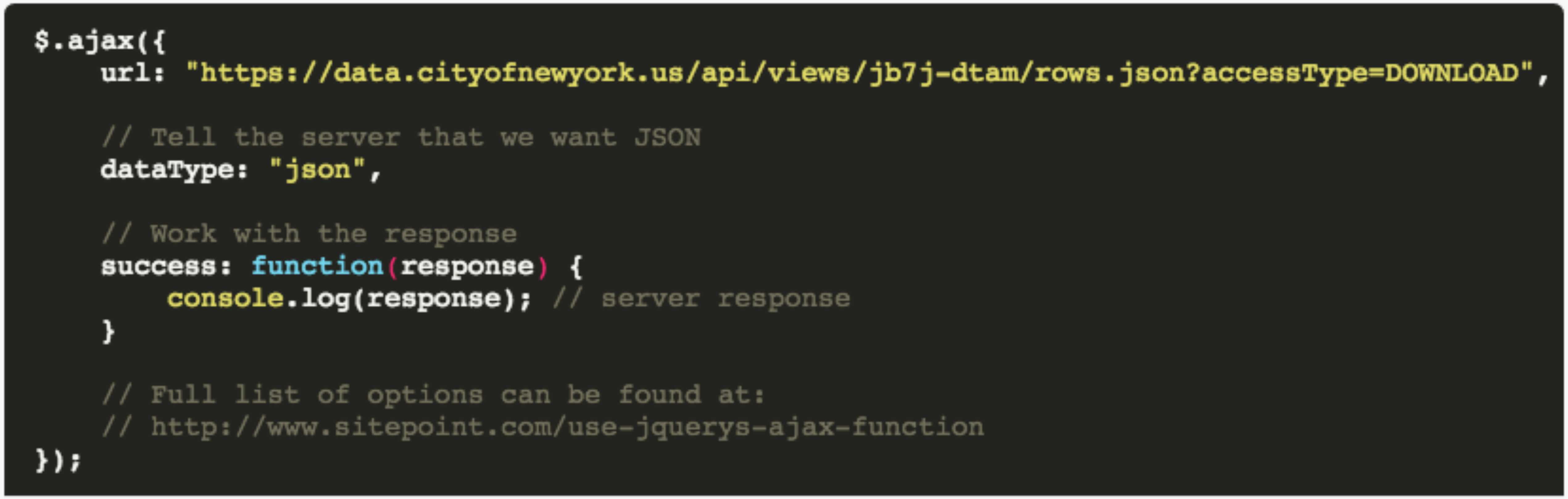


```
// All we need to create a get or post request is use the get or post method
var url = 'https://data.cityofnewyork.us/api/views/jb7j-dtam/rows.json?accessType=DOWNLOAD';
$.get(url, function(response) {
    // We get the data back from the server
    // in the parameter we pass into the function
    console.log(response);
});
```

The $.ajax function

- **$.get** (or **$.post**) are usually all we'll need, but they're actually shorthand for the **$.ajax** function, which provides more options if you need them.

- The following does exactly the same thing as **$.get()** :



```
$.ajax({
    url: "https://data.cityofnewyork.us/api/views/jb7j-dtam/rows.json?accessType=DOWNLOAD",

    // Tell the server that we want JSON
    dataType: "json",

    // Work with the response
    success: function(response) {
        console.log(response); // server response
    }

    // Full list of options can be found at:
    // http://www.sitepoint.com/use-jquerys-ajax-function
});
```

Codealong

- **cd** to **~/GA-JS**
- Create a **lesson11** folder (if you haven't already)
- **cd** into it
- create **ajax-jquery.js** and **ajax-jquery.html** files
- **subl .**
- **(Note: jquery link is https://code.jquery.com/jquery-2.2.4.js)**

AJAX Lab 2

- Code (and also the instructions below) at http://bit.ly/jsdev2-lesson11-ajax-exercise-2
- Let's bring it all together. Open the main.js file. We will talk with a weather API, and retrieve weather information. Thus far we have worked with just pulling static URLs. Follow the steps below.
  - Sign up for **openweathermap.org** and generate an API key.
  - User either **$.ajax** or **$.get** to pull weather current data for Washington DC
    - **hint:** http://api.openweathermap.org/data/2.5/weather?q=...
  - Print the temperature in console.
  - Bonus 1: add a form prompting user for the city and state.
  - Bonus 2: convert answer from kelvin to fahrenheit.

Suggested exercises

- Suggested additional exercises:
  - JSON exercise: http://bit.ly/jsdev2-lesson11-json-exercise
  - Open Weather API lab if we don't get to it in class: http://bit.ly/jsdev2-lesson11-ajax-exercise-2
  - (Feel free to use the jQuery AJAX functions **$.get** or **$.ajax** for the AJAX exercises)