Asynchronous JS

SWE 432, Fall 2019
Web Application Development
Today

- What is asynchronous programming?
- What are threads?
- Writing asynchronous code

For further reading:

Review: Closures

- Closures are expressions that work with variables in a specific context
- Closures contain a function, and its needed state
  - Closure is a stack frame that is allocated when a function starts executing and not freed after the function returns
- That state just refers to that state by name (sees updates)

```javascript
var x = 1;
function f() {
  var y = 2;
  return function() {
    console.log(x + y);
    y++;
  };
}
var g = f();
g(); // 1+2 is 3
```

This function attaches itself to x and y so that it can continue to access them.

It “closes up” those references
Review: Closures

```javascript
var x = 1;
function f() {
    var y = 2;
    return function() {
        console.log(x + y);
        y++;
    }
}

var g = f();
g(); // 1+2 is 3

// After g is called, x is still 1
// because it was the global variable
// 1+3 is 4

// If x is changed in the closure
// it will affect the new value
var x = 2;
g();
```

Diagram:
- **Global**: `var x`
- **Closure**: `var y` (inside function)
Review: Closures

```javascript
var x = 1;
function f() {
    var y = 2;
    return function() {
        console.log(x + y);
        y++;
    };
}
var g = f();
g(); // 1+2 is 3
// 1+3 is 4
g();
```

Diagram:
- `f()` function
- `g()` function
- Closure
- Global scope

Variables:
- `x`: Value 1
- `y`: Value 2
Review: Closures

```javascript
var x = 1;
function f() {
  var y = 2;
  return function() {
    console.log(x + y);
    y++;
  };
}
var g = f();
g(); // 1+2 is 3
// 1+3 is 4
```
Why Asynchronous?

- Maintain an interactive application while still doing stuff
  - Processing data
  - Communicating with remote hosts
  - Timers that countdown while our app is running
- Anytime that an app is doing more than one thing at a time, it is asynchronous
What is a thread?

Program execution: a series of sequential method calls (★'s)

App Starts

App Ends
What is a thread?

Program execution: a series of sequential method calls (★'s)

App Starts

App Ends

Multiple threads can run at once -> allows for asynchronous code
Multi-Threading in Java

- Multi-Threading allows us to do more than one thing at a time
- Physically, through multiple cores and/or OS scheduler
- Example: Process data while interacting with user

![Diagram of Multi-Threading]

- Main thread 0
  - Interacts with user
  - Draws Swing interface on screen, updates screen
- Worker thread 1
  - Processes data, generates results
  - Share data
  - Signal each other
Woes of Multi-Threading

```java
public static int v;
public static void thread1()
{
    v = 4;
    System.out.println(v);
}

public static void thread2()
{
    v = 2;
}
```

This is a data race: the println in thread1 might see either 2 OR 4
Multi-Threading in JS

```javascript
var request = require('request');
request('http://www.google.com', function (error, response, body) {
  console.log("Heard back from Google!");
});
console.log("Made request");
```

Output:
Made request
Heard back from Google!

Request is an **asynchronous call**
Multi-Threading in JS

- Everything you write will run in a single thread* (event loop)
- Since you are not sharing data between threads, races don’t happen as easily
- Inside of JS engine: many threads
- Event loop processes events, and calls your callbacks
The Event Loop

Event Queue

- response from google.com
- response from facebook.com
- response from gmu.edu

JS Engine

Event Being Processed:
The Event Loop

Event Queue

response from facebook.com
response from gmu.edu

JS Engine

Event Being Processed:
response from google.com

Are there any listeners registered for this event?
If so, call listener with event
After the listener is finished, repeat
Event Being Processed:

Are there any listeners registered for this event?
If so, call listener with event
After the listener is finished, repeat
The Event Loop

Event Queue

JS Engine

event loop

thread 1
thread 2
thread 3
thread n

Event Being Processed:

response from gmu.edu

Are there any listeners registered for this event? If so, call listener with event After the listener is finished, repeat
The Event Loop

• Remember that JS is **event-driven**

```javascript
var request = require('request');
request('http://www.google.com', function (error, response, body) {
    console.log("Heard back from Google!");
});
console.log("Made request");
```

• Event loop is responsible for dispatching events when they occur

• Main thread for event loop:

```javascript
while(queue.waitForMessage()){
    queue.processNextMessage();
}
```
How do you write a “good” event handler?

• Run-to-completion
  • The JS engine will not handle the next event until your event handler finishes

• Good news: no other code will run until you finish (no worries about other threads overwriting your data)

• Bad/OK news: Event handlers must not block
  • Blocking -> Stall/wait for input (e.g. alert(), non-async network requests)
  • If you *must* do something that takes a long time (e.g. computation), split it up into multiple events
More Properties of Good Handlers

• Remember that event events are processed in the order they are received

• Events might arrive in unexpected order

• Handlers should check the current state of the app to see if they are still relevant
Prioritizing events in node.js

- Some events are more important than others
- Keep separate queues for each event "phase"
- Process all events in each phase before moving to next

Benefits vs. Explicit Threading (Java)

• Writing your own threads is reason about and get right:
  
  • When threads share data, need to ensure they correctly synchronize on it to avoid race conditions
  
• Main downside to events:
  
  • Can not have slow event handlers
  
  • Can still have races, although easier to reason about
Run-to-completion semantics

• Run-to-completion
  • The function handling an event and the functions that it (transitively) synchronously calls will keep executing until the function finishes.
  • The JS engine will not handle the next event until the event handler finishes.

processing of event queue

callback1 → f → g
  ↘
  → h → ... → i

callback2 → ... → j
Implications of run-to-completion

• Good news: no other code will run until you finish (no worries about other threads overwriting your data)

\[ \text{processing of event queue} \]

\[ \text{callback1} \quad \rightarrow \quad \text{f} \quad \rightarrow \quad \text{g} \]
\[ \begin{align*} \text{h} \quad \rightarrow \quad \ldots \quad \rightarrow \quad \text{i} \\ \end{align*} \]

\[ \text{callback2} \quad \rightarrow \quad \ldots \quad \rightarrow \quad \text{j} \]

\textit{j will not execute until after i}
Implications of run-to-completion

• Bad/OK news: Nothing else will happen until event handler returns

• Event handlers should never block (e.g., wait for input) --> all callbacks waiting for network response or user input are **always** asynchronous

• Event handlers shouldn't take a long time either

\[ \text{processing of event queue} \]

\[ \text{callback1} \]

\[ \text{callback2} \]

\[ j \text{ will not execute until } i \text{ finishes} \]
Decomposing a long-running computation

• If you *must* do something that takes a long time (e.g. computation), split it into multiple events
  • doSomeWork();
  • ... [let event loop process other events]..
  • continueDoingMoreWork();
  • ...

Dangers of decomposition

• Application state may **change** before event occurs

  • Other event handlers may be interleaved and occur before event occurs and mutate the same application state

  • --> Need to check that update still makes sense

• Application state may be in **inconsistent** state until event occurs

  • Application

  • leaving data in inconsistent state...

• Loading some data from API, but not all of it...
Example: Writing Asynchronous Tasks

• From an array of 10 URL’s:
  
  • Request each URL
  
  • Then for each page, save it to disk
  
  • Then once all of the pages are downloaded and saved, print out the total size of all of the files that were saved
Sequencing events

• We'd like a better way to sequence events.

• Goals:

  • Clearly distinguish **synchronous** from **asynchronous** function calls.

  • Enable computation to occur only **after** some event has happened, without adding an additional nesting level each time (no pyramid of doom).

  • Make it possible to handle **errors**, including for multiple related async requests.

  • Make it possible to **wait** for multiple async calls to finish before proceeding.
Sequencing events with Promises

• Promises are a **wrapper** around async callbacks

• Promises represents **how** to get a value

• Then you tell the promise what to do **when** it gets it

• Promises organize many steps that need to happen in order, with each step happening asynchronously

• At any point a promise is either:
  • Is unresolved
  • Succeeds
  • Fails
Using a Promise

• Declare what you want to do when your promise is completed (then), or if there’s an error (catch)

```javascript
fetch('https://github.com/').
  .then(function(res) {
    return res.text();
  });

fetch('http://domain.invalid/').
  .catch(function(err) {
    console.log(err);
  });
```
Promise one thing then another

- Promise to get some data
  - then
  - Promise to get some data based on that data
    - then
    - Use that data to update application state
      - then
      - If there’s an error…
        - Report on the error
          - If there’s an error…
Chaining Promises

myPromise.then(function(resultOfPromise){
    //Do something, maybe asynchronously
    return theResultOfThisStep;
})
  .then(function(resultOfStep1){
    //Do something, maybe asynchronously
    return theResultOfStep2;
})
  .then(function(resultOfStep2){
    //Do something, maybe asynchronously
    return theResultOfStep3;
})
  .then(function(resultOfStep3){
    //Do something, maybe asynchronously
    return theResultOfStep4;
})
  .catch(function(error){
  });
Writing a Promise

• Most often, Promises will be generated by an API function (e.g., fetch) and returned to you.

• But you can also create your own Promise.

```javascript
var p = new Promise(function(resolve, reject) {
  if (/* condition */) {
    resolve(/* value */); // fulfilled successfully
  }
  else {
    reject(/* reason */); // error, rejected
  }
});
```
Example: Writing a Promise

- `loadImage` returns a promise to load a given image

```javascript
function loadImage(url){
    return new Promise((resolve, reject) => {
        var img = new Image();
        img.src = url;
        img.onload = function(){
            resolve(img);
        }
        img.onerror = function(e){
            reject(e);
        }
    });
}
```

Once the image is loaded, we’ll resolve the promise.

If the image has an error, the promise is rejected.
Writing a Promise

• Basic syntax:
  
  • do something (possibly asynchronous)
  
  • when you get the result, call resolve() and pass the final result
  
  • In case of error, call reject()

```javascript
var p = new Promise( function(resolve, reject){
  // do something, who knows how long it will take?
  if(everythingIsOK)
  {
    resolve(stateIWantToSave);
  }
  else
  {
    reject(Error("Some error happened"));
  }
});
```
Promises in Action

- Firebase example: get some value from the database, then push some new value to the database, then print out “OK”

```javascript
todosRef.child(keyToGet).once('value')
.then(function(foundTodo){
  return foundTodo.val().text;
})
.then(function(theText){
  todosRef.push({'
text': ''Seriously: '' + theText});
})
.then(function(){
  console.log("OK!");
})
.catch(function(error){
  //something went wrong
});
```

Do this

Then, do this

Then do this

And if you ever had an error, do this
Testing Promises

```javascript
function getUserName(userID) {
    return request-promise(`/users/` + userID).then(user => user.name);
}
```

```javascript
it('works with promises', () => {
    expect.assertions(1);
    return user.getUserName(4).then(data => expect(data).toEqual('Mark'));
});
```

```javascript
it('works with promises', () => {
    expect.assertions(1);
    return user.getUserName(4).then(data => expect(data).toEqual('Mark'));
});
```

```javascript
it('works with resolves', () => {
    expect.assertions(1);
    return expect(user.getUserName(5)).resolves.toEqual('Paul');
});
```

https://jestjs.io/docs/en/tutorial-async
Next Time

• More asynchronous examples

• async/wait keywords

• Threading in JS