Why JavaScript?

On the client
Why JavaScript?

On the client

On the server
Why JavaScript?

- On the client
- On the server
- Even in hardware!
Why JavaScript?

JavaScript has become the modern lingua franca!

On the client

On the server

Even in hardware!
```javascript
console.log('Hello');
$.get('data.json', callback);
$('#button').click(eventHandler);
console.log('World');
```

1. `log('Hello')`
2. Do HTTP get call
3. Register event Handler
4. `log('World')`
5. `callback()`
6. `eventHandler()`
console.log('Hello');
$.get('data.json', callback);
$('button').click(eventHandler);
console.log('World');

1. log('Hello')
2. Do HTTP get call
3. Register event Handler
4. log('World')
5. callback()
6. eventHandler()
JavaScript Event-driven Model

```javascript
console.log('Hello');
$.get('data.json', callback);
$('button').click(eventHandler);
console.log('World');
```

1. log('Hello')
2. Do HTTP get call
3. Register event handler
4. log('World')
5. callback()
6. eventHandler()
```javascript
console.log('Hello');
$.get('data.json', callback);
$('button').click(eventHandler);
console.log('World');
```

1. `log('Hello')`
2. Do HTTP get call
3. `Register event Handler`
4. `log('World')`
5. `callback()`
6. `eventHandler()`
JavaScript Event-driven Model

```javascript
console.log('Hello');
$.get('data.json', callback);
$('button').click(eventHandler);
console.log('World');
```
console.log('Hello');
$.get('data.json', callback);
$("button").click(eventHandler);
console.log('World');

1. log('Hello')
2. Do HTTP get call
3. Register event Handler
4. log('World')
5. callback()
6. eventHandler()
JavaScript Event-driven Model

1. `console.log('Hello');`
2. $.get('data.json', callback);
3. `$('button').click(eventHandler);`
4. `console.log('World');`
5. `callback();`
6. `eventHandler();`
JavaScript Event-driven Model

```javascript
console.log('Hello');
$.get('data.json', callback);
$('button').click(eventHandler);
console.log('World');
```

1. `log('Hello')`
2. Do HTTP get call
3. Register event Handler
4. `log('World')`
5. `callback()`

eventHandler()
Why Callbacks?

Used For:
- HTTP Request/Response
- File I/O in Node.js
- Mouse click/drag events in the browser

```javascript
$("button").click(buttonHandler);

function buttonHandler(event){
    alert("Button Clicked.");
}
```
Why Callbacks?

Used For:
- HTTP Request/Response
- File I/O in Node.js
- Mouse click/drag events in the browser

You cannot write an interesting program in JavaScript without using **callbacks**!
Motivation

- Callbacks are important in all non-trivial JS programs.
- But, how do developers use callbacks?
- No prior research on this topic.

**Goal of this work**
Outline

Motivation ✔
Methodology & Overview
Characterizing Problems
    Anonymous Callbacks
    Asynchronous Callbacks
    Nested Callbacks

Characterizing Solutions
    Error-first Protocol
    Async.js
    Promises

Conclusion
Outline

Motivation ✔

Methodology & Overview

Characterizing Problems
  Anonymous Callbacks
  Asynchronous Callbacks
  Nested Callbacks

Characterizing Solutions
  Error-first Protocol
  Async.js
  Promises

Conclusion
Subject Systems

138 popular open source JavaScript subject systems, from 6 distinct categories, with both Client-side & Server-side code.

<table>
<thead>
<tr>
<th>Category</th>
<th>Subject systems</th>
<th>Client side</th>
<th>Server side</th>
<th>Total files</th>
<th>Total LOC</th>
</tr>
</thead>
<tbody>
<tr>
<td>NPM Modules</td>
<td>86</td>
<td>✓</td>
<td>✓</td>
<td>4,983</td>
<td>1,228,271</td>
</tr>
<tr>
<td>Web Apps.</td>
<td>16</td>
<td>✓</td>
<td></td>
<td>1,779</td>
<td>494,621</td>
</tr>
<tr>
<td>Game Engines</td>
<td>16</td>
<td>✓</td>
<td>✓</td>
<td>1,740</td>
<td>1,726,122</td>
</tr>
<tr>
<td>Frameworks</td>
<td>8</td>
<td>✓</td>
<td></td>
<td>2,374</td>
<td>711,172</td>
</tr>
<tr>
<td>DataViz Libs.</td>
<td>6</td>
<td>✓</td>
<td></td>
<td>3,731</td>
<td>958,983</td>
</tr>
<tr>
<td>Games</td>
<td>6</td>
<td>✓</td>
<td></td>
<td>347</td>
<td>119,279</td>
</tr>
<tr>
<td>Total</td>
<td>138</td>
<td>✓</td>
<td>✓</td>
<td>14,954</td>
<td>5,238,448</td>
</tr>
</tbody>
</table>
What is a callback?

A **callback** is a function that is passed as an argument to another function, which is expected to invoke it either *immediately* or at some point *in the future*.

```javascript
function getInput (options, callback) {
    allUserData.push (options);
    callback (options);
}

var logStuff = function () { ... }
getInput ({name:"Rich", speciality:"JavaScript"}, logStuff);
```
What is a callback?

A **callback** is a function that is passed as an argument to another function, which is expected to invoke it either *immediately* or at some point *in the future*.

```javascript
function getInput (options, callback) {
  allUserData.push (options);
  callback (options);
}

var logStuff = function () {
  ... 
}

getInput ({name: "Rich", speciality: "JavaScript"}, logStuff);
```
Detecting callbacks: An example

```javascript
function getRecord(id, callback) {
  http.get('http://foo/' + id, function (err, doc) {
    if (err) {
      return callback(err);
    }
    return callback(null, doc);
  });
}

var logStuff = function () { ... }
getRecord('007', logStuff);
```

getRecord() accepts logStuff() as a callback because there exist a path...
Detecting callbacks: An example

```
function getRecord(id, callback) {
  http.get('http://foo/' + id, function (err, doc) {
    if (err) {
      return callback(err);
    }
    return callback(null, doc);
  });
}

var logStuff = function () { ... }
getRecord('007', logStuff);
```

getRecord() accepts logStuff() as a callback because there exist a path...

getRecord(cb) ➔ http.get() ➔ Anonymous() ➔ logStuff()
Detecting callbacks: An example

```javascript
function getRecord(id, callback) {
  http.get('http://foo/' + id, function (err, doc) {
    if (err) {
      return callback(err);
    }
    return callback(null, doc);
  });
}

var logStuff = function () { ... }
getRecord('007', logStuff);
```

getRecord() accepts logStuff() as a callback because there exist a path...
getRecord(cb) ➔ http.get() ➔ Anonymous() ➔ logStuff()
Detecting callbacks: An example

getRecord() accepts logStuff() as a callback because there exist a path...

getRecord(cb) → http.get() → Anonymous() → logStuff()
Detecting callbacks: An example

getRecord() accepts logStuff() as a callback because there exist a path...

getRecord(cb) ➔ http.get() ➔ Anonymous() ➔ logStuff()
Detecting callbacks: An example

```javascript
function getRecord(id, callback) {
    http.get(`http://foo/${id}`, function (err, doc) {
        if (err) {
            return callback(err);
        } else {
            return callback(null, doc);
        }
    });
}
```

getRecord() accepts `logStuff()` as a callback because there exist a path...

getRecord(cb) ⇒ http.get() ⇒ Anonymous() ⇒ logStuff()
Detecting callbacks: An example

```javascript
function getRecord(id, callback) {
  http.get('http://foo/' + id, function (err, doc) {
    if (err) {
      return callback(err);
    }
    return callback(null, doc);
  });
}

var logStuff = function() { ... }

getRecord(1234, logStuff);
```

getRecord() accepts logStuff() as a callback because there exist a path...

getRecord(cb) ➔ http.get() ➔ Anonymous() ➔ logStuff()
Detecting callbacks: An example

```javascript
function getRecord(id, callback) {
  http.get('http://foo/' + id, function (err, doc) {
    if (err) {
      return callback(err);
    }
    return callback(null, doc);
  });
}
```

getRecord() accepts logStuff() as a callback because there exist a path...

getRecord(cb) ★ http.get() ★ Anonymous() ★ logStuff()
Detecting callbacks: An example

```
function getRecord(id, callback) {
  http.get('http://foo/' + id, function (err, doc) {
    if (err) {
      return callback(err);
    }
    return callback(null, doc);
  });
}

var logStuff = function () { ... }
getRecord('007', logStuff);
```

getRecord() accepts logStuff() as a callback because there exist a path...

getRecord(cb) ➔ http.get() ➔ Anonymous() ➔ logStuff()
How prevalent are callbacks?

Callback-accepting function definitions

- On average, **10%** of all function definitions take callback arguments.
- They are more prevalent in server-side code (10%) than in client-side code (4.5%).

Callback-accepting function call-sites

- **19%** of all function call-sites take callback arguments.
- Callback-accepting function call-sites are more prevalent in server-side code (24%) than in client-side code (9%).
How prevalent are callbacks?

Callback-accepting function definitions

- On average, 10% of all function definitions take callback arguments.
- They are more prevalent in server-side code (10%) than in client-side code (4.5%).

Callback-accepting function call-sites

- Every 10th function definition takes a callback.
- Every 5th function call-site takes a callback.
- Callbacks are extensively used in the server-side.

- Callback-accepting function call-sites are more prevalent in server-side code (24%) than in client-side code (9%).
Outline

Motivation ✔️
Methodology & Overview ✔️
Characterizing Problems
  Anonymous Callbacks
  Asynchronous Callbacks
  Nested Callbacks

Characterizing Solutions
  Error-first Protocol
  Async.js
  Promises

Conclusion
Outline

Motivation ✓
Methodology & Overview ✓

Characterizing Problems
- Anonymous Callbacks
- Asynchronous Callbacks
- Nested Callbacks

Characterizing Solutions
- Error-first Protocol
- Async.js
- Promises

Conclusion
define('admin/general/dashboard', 'semver', function(semver)
{
    var Admin = {};

    $(
    
    
    
    
    
    
    $(
    .restart').on('click', function() {
        bootbox.confirm('Are you sure you wish to restart NodeBB?', function(confirm) {
            if (confirm) {
                $(window).one('action:reconnected', function() {
                    app.alert({ alert_id: 'instance_restart', });
                    socket.emit('admin.restart');
                });
            }
        });
    return Admin;
});

});

JavaScript in the wild...
Anonymous callbacks

Notorious for:
Difficulty to debug, maintain, test, or reuse
Anonymous Vs. Named Callbacks

If a function callsite is ..

- callback-accepting and
- has an anonymous function expression as an argument

..it is an instance of an anonymous callback.

```javascript
function getRecord(id, callback) {
    http.get('http://foo/' + id, function (err, doc) {
        if (err) {
            return callback(err);
        }
        return callback(null, doc);
    });

    var logStuff = function () {
        ... 
    }

    getRecord('007', logStuff);
```
Anonymous Vs. Named Callbacks

If a function callsite is..

- callback-accepting and
- has an anonymous function expression as an argument

..it is an instance of an anonymous callback.
Anonymous Vs. Named Callbacks - Results

- 43% of all callback-accepting function callsites are invoked with at least one anonymous callback.
- There is little difference between client-side and server-side code in how they use anonymous callbacks.
Anonymous Vs. Named Callbacks - Results

- 43% of all callback-accepting functions are invoked with at least one anonymous callback.
- There is little difference between client-side and server-side code in how they use anonymous callbacks.

In spite of maintenance challenges, Anonymous callbacks are here to stay!
Asynchronous callbacks

```
define('admin/general/dashboard', 'semver', function(semver)

    var Admin = {};

    $('#logout-link').on('click', function() {
        $.post(RELATIVE_PATH + '/logout', function() {
            window.location.href = RELATIVE_PATH + '/';
        });
    });

    ...

    $('.restart').on('click', function() {
        bootbox.confirm('Are you sure you wish to restart
            NodeBB?', function(confirm) {
            if (confirm) {
                $(window).one('action:reconnected', function() {
                    app.alert({ alert_id: 'instance_restart', });
                });

                socket.emit('admin.restart');
            }
        });
    });

    return Admin;
});
```

Notorious for:
Making it hard to reason about the execution.
Asynchronous Callbacks

<table>
<thead>
<tr>
<th>Category</th>
<th>Examples</th>
<th>Availability</th>
</tr>
</thead>
<tbody>
<tr>
<td>DOM events</td>
<td>addEventListener, onclick</td>
<td>Browser</td>
</tr>
<tr>
<td>Network calls</td>
<td>XMLHttpRequest.open</td>
<td>Browser</td>
</tr>
<tr>
<td>Timers (macro-Task)</td>
<td>setImmediate(), setTimeout(), setInterval()</td>
<td>Browser, Node.js</td>
</tr>
<tr>
<td>Timers (micro-task)</td>
<td>process.nextTick()</td>
<td>Node.js</td>
</tr>
<tr>
<td>I/O</td>
<td>APIs of fs, net</td>
<td>Node.js</td>
</tr>
</tbody>
</table>

Some Asynchronous APIs in JavaScript
More than half (56%) of all callbacks are Asynchronous.

Asynchronous callbacks, on average, appear more frequently in client-side code (72%) than in server-side code (55%).
Asynchronous Callbacks – Results

- More than half (56%) of all callbacks are Asynchronous.
- Asynchronous callbacks, on average, appear more frequently in client-side code (72%) than in server-side code (55%).

Program analyses techniques must account for Asynchrony.
Nested Callbacks

Notorious for:
Callback hell aka Pyramid of Doom

```javascript
define('admin/general/dashboard', 'semver', function(semver)
{
  var Admin = {};

  $('#logout-link').on('click', function()
  {
    $.post(RELATIVE_PATH + '/logout', function()
    {
      window.location.href = RELATIVE_PATH + '/';
    });
  });

  ...

  $('.restart').on('click', function()
  {
    bootbox.confirm('Are you sure you wish to restart NodeBB?', function(confirm)
    {
      if (confirm) {
        $(window).one('action:reconnected', function()
        {
          app.alert({ alert_id: 'instance_restart', });
        });

        socket.emit('admin.restart');
      }
    });
  });

  return Admin;
});
```
Nested Callbacks - Results

- Callbacks are nested up to a depth of 8.
- There is a peak at nesting level of 2.
Nested Callbacks - Results

Widely-used Nesting - An opportunity for tool builders!
Outline

Motivation ✔
Methodology & Overview ✔
Characterizing Problems ✔
  Anonymous Callbacks ✔
  Asynchronous Callbacks ✔
  Nested Callbacks ✔

Characterizing Solutions
  Error-first Protocol
  Async.js
  Promises

Conclusion
Outline

Motivation ✔
Methodology & Overview ✔
Characterizing Problems ✔
  Anonymous Callbacks ✔
  Asynchronous Callbacks ✔
  Nested Callbacks ✔

► Characterizing Solutions
  Error-first Protocol
  Async.js
  Promises

Conclusion
Error-first Protocol

- JS has no explicit language support for asynchronous error-signaling
- Developer community has the convention:

  Dedicate the 1st argument in the callback to be a permanent place-holder for error-signalling

```
var fs = require('fs');

// read a file
function read_the_file(filename, callback) {
  fs.readFile(filename, function (err, contents) {
    if (err) return callback(err);
    // if no error, continue
    read_data_from_db(null, contents, callback);
  });
}

function read_data_from_db(err, contents, callback) {
  // some long running task
}

read_the_file('/some/file', function (err, result) {
  if (err) {
    // handle the error
    console.log(err);
    return;
  }
  // do something with the result
});
```
Error-first Protocol

- JS has no explicit language support for asynchronous error-signaling
- Developer community has the convention:
  
  Dedicate the 1st argument in the callback to be a permanent place-holder for error-signalling

```
var fs = require('fs');
// read a file
function read_the_file(filename, callback) {
  fs.readFile(filename, function (err, contents) {
    if (err) return callback(err);
    // if no error, continue
    read_data_from_db(null, contents, callback);
  });
}

function read_data_from_db(err, contents, callback) {
  //some long running task
}
read_the_file('/some/file', function (err, result) {
  if (err) {
    //handle the error
    console.log(err);
    return;
  }
  // do something with the result
});
```
Error-first Protocol - Results

To detect error-first protocol we checked
- if the first parameter $p$ of a function definition $f$ has the name ‘error’ or ‘err’
- if $f$’s callsites also contain ‘error’ or ‘err’ as their first argument.

We found
- 20% of function definitions follow the error-first protocol
- The error-first protocol is used twice as often in server-side code than in client-side code (30% Vs 16%)
- 73% (63 out of 86) NPM modules and 93% (15 out of 16) web applications had instances of it.
Error-first Protocol - Results

We found

- 20% of function definitions follow the error-first protocol.
- The error-first protocol is used twice as often in server-side code than in client-side code (30% vs 16%).
- 73% (63 out of 86) NPM modules and 93% (15 out of 16) web applications had instances of it.

To detect error-first protocol we checked

- if the first parameter of a function definition has the name 'error' or 'err'
- if the callsites also contain 'error' or 'err' as their first argument.

Cannot depend on APIs/libraries to enforce error-first protocol.
Usage of Async.js

Async.js: popular library to manage asynchronous control flow, and to help with functional programming. For example:

Without Async.js

```javascript
var users = [];
fetchUsers(function() {
    renderUsersOnPage(function() {
        fadeInUsers(function() {
            loadUserPhotos(function() {
                // do something
            });
        });
    });
});
```

With Async.js

```javascript
var users = [];
async.series([
    function(callback) {
        fetchUsers(callback);
    },
    function(callback) {
        renderUsersOnPage(callback);
    },
    function(callback) {
        fadeInUsers(callback);
    },
    function(callback) {
        loadUserPhotos(callback);
    } ],
);
Usage of Async.js - Results

- More than half of the web applications (56%) use Async.js.
- Usage is much lower (11%) in the NPM modules.
- Async.js library is used differently in these 2 categories of subject systems.

Top 10 Async.js invoked methods in JavaScript Web Applications (Left) and NPM modules (Right).

<table>
<thead>
<tr>
<th>Rank</th>
<th>Method</th>
<th>Count</th>
<th>Rank</th>
<th>Method</th>
<th>Count</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>nextTick</td>
<td>18</td>
<td>1</td>
<td>parallel</td>
<td>189</td>
</tr>
<tr>
<td>2</td>
<td>queue*</td>
<td>16</td>
<td>2</td>
<td>apply</td>
<td>81</td>
</tr>
<tr>
<td>3</td>
<td>each</td>
<td>14</td>
<td>3</td>
<td>waterfall</td>
<td>72</td>
</tr>
<tr>
<td>3</td>
<td>setImmediate*</td>
<td>14</td>
<td>4</td>
<td>series</td>
<td>61</td>
</tr>
<tr>
<td>3</td>
<td>series</td>
<td>14</td>
<td>5</td>
<td>each</td>
<td>48</td>
</tr>
<tr>
<td>6</td>
<td>auto*</td>
<td>11</td>
<td>6</td>
<td>map</td>
<td>37</td>
</tr>
<tr>
<td>6</td>
<td>waterfall</td>
<td>11</td>
<td>7</td>
<td>eachSeries*</td>
<td>20</td>
</tr>
<tr>
<td>6</td>
<td>parallel</td>
<td>11</td>
<td>8</td>
<td>eachLimit*</td>
<td>12</td>
</tr>
<tr>
<td>9</td>
<td>map</td>
<td>10</td>
<td>9</td>
<td>whilst*</td>
<td>10</td>
</tr>
<tr>
<td>9</td>
<td>apply</td>
<td>10</td>
<td>9</td>
<td>nextTick</td>
<td>10</td>
</tr>
</tbody>
</table>
Usage of Async.js - Results

- More than half of the web applications (56%) use Async.js.
- Usage is much lower (11%) in the NPM modules.
- Async.js library is used differently in these 2 categories of subject systems.

Difference in API usage across categories indicate different underlying concerns in callback management!

<table>
<thead>
<tr>
<th>Rank</th>
<th>Method</th>
<th>Count</th>
<th>Rank</th>
<th>Method</th>
<th>Count</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td>auto*</td>
<td>11</td>
<td>6</td>
<td>map</td>
<td>37</td>
</tr>
<tr>
<td>6</td>
<td>waterfall</td>
<td>11</td>
<td>7</td>
<td>eachSeries*</td>
<td>20</td>
</tr>
<tr>
<td>6</td>
<td>parallel</td>
<td>11</td>
<td>8</td>
<td>eachLimit*</td>
<td>12</td>
</tr>
<tr>
<td>9</td>
<td>map</td>
<td>10</td>
<td>9</td>
<td>whilst*</td>
<td>10</td>
</tr>
<tr>
<td>9</td>
<td>apply</td>
<td>10</td>
<td>9</td>
<td>nextTick</td>
<td>10</td>
</tr>
</tbody>
</table>

The * symbol denotes calls that do not appear in both tables.
Conclusions

• Callbacks are extensively used in the server-side

• Program analyses techniques must account for Asynchrony

• In spite of maintenance challenges, Anonymous callbacks are here to stay

• Widely-used Nesting - An opportunity for tool builders.

• Cannot depend on APIs/libraries to enforce error-first protocol.

http://salt.ece.ubc.ca/callback-study
Extra Slides
Usage of Promises

Promises: a native language feature for solving the Asynchronous composition problem. For example:

Without Promises

getUser('mjackson', function (error, user) {
  if (error) {
    handleError(error);
  } else {
    getUser('mjackson', function (error, user) {
      if (error) {
        handleError(error);
      } else {
        getNewTweets(user, function (error, tweets) {
          if (error) {
            handleError(error);
          } else {
            updateTimeline(tweets, function (error) {
              if (error) handleError(error);
            });
          }
        });
      }
    });
  }
});

With Promises

getUser('mjackson').
  .then(getNewTweets, null)
  .then(updateTimeline)
  .catch(handleError);
Usage of Promises - Results

27% of 138 subject systems use Promises.

<table>
<thead>
<tr>
<th>Category</th>
<th>Subjects creating Promises (%)</th>
<th>Subjects using Promises (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>DataViz libraries</td>
<td>6</td>
<td>31</td>
</tr>
<tr>
<td>Game Engines</td>
<td>0</td>
<td>25</td>
</tr>
<tr>
<td>Frameworks</td>
<td>50</td>
<td>75</td>
</tr>
<tr>
<td>Games</td>
<td>0</td>
<td>17</td>
</tr>
<tr>
<td>Web Applications</td>
<td>13</td>
<td>50</td>
</tr>
<tr>
<td>NPM Modules</td>
<td>3</td>
<td>12</td>
</tr>
<tr>
<td>Total</td>
<td>8</td>
<td>26</td>
</tr>
</tbody>
</table>
Usage of Promises - Results

Promises are prominently used on the Client-Side

Max: 513 usage instances
Usage of Promises - Results

Tools will help to migrate existing large projects to Promises.
Callback Accepting Functions

• **f** is a *callback-accepting function definition* if at least one argument to **f** is used as a callback.

<table>
<thead>
<tr>
<th>• A parameter <strong>p</strong> of a function <strong>f</strong> definition is a callback..</th>
</tr>
</thead>
<tbody>
<tr>
<td>○ if <strong>p</strong> is invoked as a function in the body of <strong>f</strong></td>
</tr>
<tr>
<td>○ if <strong>p</strong> is passed to a known callback-accepting function (e.g., <code>setTimeout()</code>)</td>
</tr>
<tr>
<td>○ if <strong>p</strong> is used as an argument to an unknown function <strong>f’</strong></td>
</tr>
<tr>
<td>■ and then we can recursively determine if <strong>p</strong> is a callback parameter in <strong>f’</strong></td>
</tr>
</tbody>
</table>

• a *callsite* is callback-accepting if it is of
  ○ a function that was detected to be callback-accepting, as above
  ○ a function known a-priori to be callback-accepting (e.g., `setTimeout()`)