AJAX

What is AJAX?

- Asynchronous JavaScript and XML
  - XML? Why?
- AJAX is a web development technique
- Allow us to fetch data/HTML from the server without blocking
  - Thus, making the application more responsive
- Usually the data is injected into the DOM without refreshing the whole page
- Originally was implemented using IFrame
- These days we use XMLHttpRequest

XMLHttpRequest - History

- Originally created by Outlook Web Access developers
- Was shipped as COM interface named *IXMLHTTPRequest* inside the MSXML library
- Mozila offered the same API but named it XMLHttpRequest
  - Became a da facto standard
- Finalized by the W3C only at 2009
- XMLHttpRequest 2 was finalized at 2011

XMLHttpRequest

- send is non blocking
- Need to monitor the readyState flag
- Value 4 (Loaded) means “HTTP response was received completely by the browser”

```
var url = "http://www.yahoo.com";

var status = $("#status");
status.text("Sending request to " + url + " ...");

var request = new XMLHttpRequest();
request.open("GET", url);
request.send();

request.onreadystatechange = function () {
     if (request.readyState == 4 && request.status == 200) {
         status.text("Done " + request.responseText.length);
     }
 }
```

XMLHttpRequest – More Details

- Can send the request synchronously
  - No good reason to do that
- Can read/write response/request headers

```
request.setRequestHeader("key", "value");
```

- Use abort method to cancel the operation
- send's first parameter is the request body (for HTTP post request)

```
request.send(myDataAsString);
```

XMLHttpRequest – Threading Model

- JavaScript is single threaded (Almost … HTML5)
- The HTTP request/response is handled internally by the browser using dedicated threads
- On completion a message is posted to the single thread
- Only on the next event loop the single thread may noticed the AJAX request completion and react accordingly
- This means that from developer perspective AJAX request does not create parallelism
  - No need to lock or guard against race condition

Concurrent XMLHttpRequest(s)

- The developer may create as much AJAX request as he would like
- However, most browsers limit the number of concurrent AJAX request
- Beyond the limit, the AJAX request will not be sent to the server until an older request has completed
- These are latest limits reported by the community
  - Chrome – 6
  - Firefox – 6
  - IE 9,10,11 – 6,8,13

AJAX Wrappers

- The native XMLHttpRequest API is simple but not convenient
- As developer we would like to set the HTTP verb, url and specify success and error handlers
  - All other details should be handled internally
- Most JavaScript frameworks/libraries offer their own wrappers around XMLHttpRequest
- jQuery has one too

$.ajax - GET

- A global function of the global jQuery object
    - Not available on jQuery wrapped set
- Type GET is the default value and can be omitted
    - Keep it

```
$.ajax({
     type: "GET",
     url: "http://localhost/demo/contact",
     success: function (responseText) {
         status.text("Done:  " + responseText);
     },
     error: function () {
         status.text("Error");
     }
});
```

$.ajax – Sending data with GET

- The data property may hold a
  - string – Is appended to the URL as is
  - object – Is serialized into query string format

```
$.ajax({
     type: "GET",
     url: "http://localhost/demo/contact",
     data:  {id: 1},
     success: function (responseText) {
         status.text("Done: " + responseText);
     },
     error: function () {
         status.text("Error");
     }
});
```

```
GET http://localhost/demo/contact?id=1 HTTP/1.1
Host: localhost
Connection: keep-alive
Accept: text/html, */*; q=0.01
```

$.ajax - POST

- Use the data option when posting to the server
  - Same behavior as GET but the data is sent as the request body and not as part of the URL

```
$.ajax({
     type: "POST",
     url: url,
     data: { name: "Udi", email: "udi@gmail.com" },
     success: function (responseText) {
         status.text("Done: " + responseText);
     },
     error: function () {
         status.text("Error");
     }
});
```

```
POST http://localhost/Demo/contact HTTP/1.1
Host: localhost
Connection: keep-alive
Content-Length: 30
Accept: */*
Safari/537.36
Content-Type: application/x-www-form-urlencoded; charset=UTF-8

name=Udi&email=udi%40gmail.com
```

Caching

- Browser might cache AJAX GET request
- In this case second request will not be sent
  - The $.ajax succeeds as if the request was sent
- To eliminate caching
  - Use POST request
  - Or, append a random value into the URL
    - jQuery supports this technique using cache property

```
$.ajax({
     type: "GET",
     url: url,
     success: function (responseText) {...},
     error: function () {...},
     cache: false,
```

```
GET http://localhost/Demo/html?_=1396678859948 HTTP/1.1
Host: localhost
Connection: keep-alive
```

JSON – Java Script Object Notation

- □ XML is difficult to parse inside the browser
- □ JSON is based on JavaScript object literal notation
  - Therefore, a text format
  - Easy to read and write
- □ Is lighter than XML
- □ Built-in browser support
- □ Keys must be surrounded with double quotes

```
var json = '{"id": 1, "name": "Ori", "email": "ori@gmail.com"}';
```

Built-In Browser Support

- Modern browsers offer a global object named JSON
  - For older browser consider using JSON2 https://github.com/douglascrockford/JSON-js
- Serialization

```
var obj = {
     id: 123,
     name: "Ori",
     email: "ori@gmail.com",
 };

var json = JSON.stringify(obj);
```

- Deserialization

```
var json = '{"id":1,"name":"Ori","email":"ori@gmail.com"}';
var obj = JSON.parse(json);
```

JSON Limitations

- Cannot serialize cyclic references

```
var ori = { id: 1, name: "Ori" };
var roni = { id: 2, name: "Roni" };

ori.sibling = roni;
roni.sibling = ori;

JSON.stringify(ori); // exception is thrown
```

- Object type is lost when serialization/deserializing

```
function Contact(name) {
     this.name = name;
}

var ori = new Contact("Ori");
var clone = JSON.parse(JSON.stringify(ori));

console.log(clone instanceof Contact); // prints false
```

toJSON

- □ JSON.stringify checks for toJSON method on the method being serialized
- □ If found, the method is executed and the returned value is serialized into JSON
- □ The method should return an object not a string

```
var obj = {
     id: 123,
     name: "Ori",
     toJSON: function () {
         return {
             name: this.name,
         };
     }
};
```

Receiving JSON

- □ jQuery analyzes the response Content-Type
- □ If equals to application/json it automatically parses the returned data and pass it to the success handler
- □ Old servers might not specify a Content-Type
- □ Use dataType option

```
$.ajax({
    type: "GET",
    url: "/Home/Get",
    dataType: "json",
    success: function (data) {
        console.log(data);
    }
});
```

Sending JSON

- □ By default when sending AJAX request the content type is application/x-www-form-urlencoded
- □ jQuery does not parse the data string, thus, it has no idea that we are sending JSON
- □ The misleading Content-Type might confuse the back end server
- □ Fix it

```
$.ajax({
     type: "POST",
     url: "/api/contact",
     contentType: "application/json",
     data: JSON.stringify({id:1, name: "Ori"}),
});
```

Global Ajax Settings

- Repetitive $.ajax options can be factored out into $.ajaxSetup
- Any time you invoke $.ajax the options parameter is merged with the global options object

```
$.ajaxSetup({
    beforeSend: function (jqXHR, settings) {
        if (settings.data.charAt(0) == '[' || settings.data.charAt(0) == '{') {
            jqXHR.setRequestHeader("Content-Type", "application/json");
        }
        return true;
    }
});
```

Global Ajax Events

- Monitor all AJAX requests
- Must be attached to the document element

```
$(document).ajaxStart(function () {
    console.log("Start");
});
```

- ajaxSend/ajaxSuccess/ajaxError/ajaxComplete
- ajaxStart/ajaxStop
  - Fire only once per all pending AJAX request
  - Fire only once when all pending AJAX requests complete

$.ajax Wrappers

- jQuery offers some simpler wrappers around $.ajax
  - **get**(url, data, success)
  - **getJSON**(url, data, success) – dataType is json
  - **.load**(url, data, complete)
    - Is invoked on a jQuery object
    - The returned content is replaced with that element

```
$("table").load("/Home/GetHTML/");
```

Same Origin Policy

- AJAX request sent to different domain is prohibited
  - Security reason
- The following is considered different domain
  - **Schema** - http vs. https
  - **Host –** g.com vs. goole.com
  - **Port –** http://localhost:123 vs. http:/localhost:124
- This means that one web site cannot share its data with other web sites
  - For example, Twitter would like any site to get a list of the 10 latest most popular twitts

JSONP

- A technique to overcome the same origin policy limitation
- It involves both server and client side modification
- Server
  - Instead of JSON string
  - Returns a JavaScript code which call an arbitrary method and passes the relevant JSON string

```
clientMethod({"id": 1, "name": "Ori"});
```

  - The name of the method can be specified by the client

```
http://twitter.com/latest?callback=clientMethod
```

JSONP - Client

- Dynamically appends script tag into the HTML
  - src attribute should point to the server side URL which returns the JSONP content
- The browser downloads the script and executes it
- Need to remove the script tag
- Error handling it tricky

```
$("head").append('<script src="/Home/Get" />');

function clientMethod(str) {
    console.log(JSON.parse(str));
}
```

JSONP - jQuery

- □ jQuery supports JSONP invocation through $.ajax
- □ This is a totally different mechanism since there is no use of XMLHttpRequest object
- □ Allow us to consume JSONP content as if it was plain HTTP service which returns JSON

```
$.ajax({
    url: "/Home/Get",
    dataType: "jsonp",
    success: function (data) {
        console.log(data);
    },
    error: function () {
        console.log("ERROR");
    }
});
```

CORS

- HTML5 introduces the concept of CORS
  - Cross Origin Resource Sharing
  - A challenge-response mini protocol
- Old servers
  - Do not support the new protocol
  - Therefore the returned response is detected as invalid
  - The browser rejects the response
  - Application does not get a chance to process the response

CORS

- Assuming new browser and server
- The browser allows the request to be sent
  - Appends an HTTP header named Origin
  - Contains the URL of the requesting domain
- The response includes an HTTP header named Access-Control-Allow-Origin
  - Contains a list of allowed domains
- If the requesting domain is included inside the allowed list the browser let the application process the response as if it was a plain AJAX response

Access-Control-Allow-Credentials

- By default the browser does not sent any cookie alongside the request
- To change this behavior
  - Specify xhrFields.withCredentials = true
  - Server must return Access-Control-Allow-Credentials HTTP header with value equals true

```
$.ajax({
     url: "http://localhost:10659/api/contact",
     type: "GET",
     success: function (contacts) {     },
     error: function (jqXHR, text, error) {     },
     xhrFields: {
         withCredentials: true,
     }
});
```

Summary

- AJAX manipulation is very common
- JSON is the web preferred data format
- Sending/Receiving JSON with $.ajax is easy
- Same origin policy limits access to different domains
- Use JSONP to bypass same origin policy

PROMISE API

Motivation

- We want asynchronous code
  - Else, UI is locked
- Usually asynchronous function uses callbacks
  - You lose separation of input/output parameters
  - Difficult to compose multiple serial operations
  - Bubbling up exceptions is a challenge
  - Cannot use built-in control flow constructs

Compose Multiple Operations

- □ Asynchronous version

```
step1(function (value1) {
    step2(value1, function (value2) {
        step3(value2, function (value3) {
            step4(value3, function (value4) {
                // Do something with value4
            });
        });
    });
});
```

- □ Synchronous version

```
Step4(step3(step2(step1())));
```

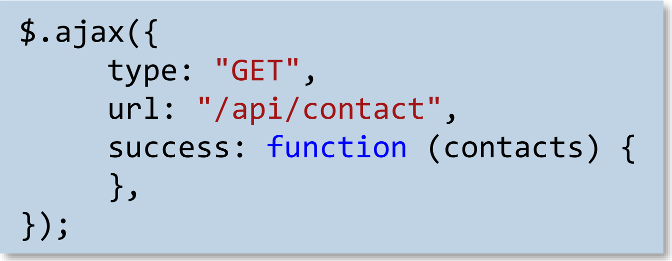
- □ Code might be even more complex when integrating error handling

Promise

- Asynchronous function should return a promise instead of the input success/error callbacks
- A promise represents a value that is the result of an asynchronous operation
- The result may be an exception
- This is a well known pattern - late seventies !!!
  - But only recently integrated into JavaScript
- The future is promising ☺
  - ECMA Script 6 introduces the concept of Generator

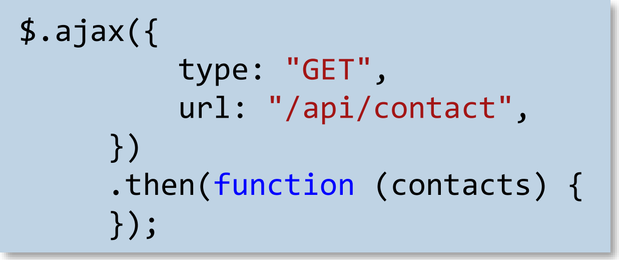
Basics

- □ $.ajax



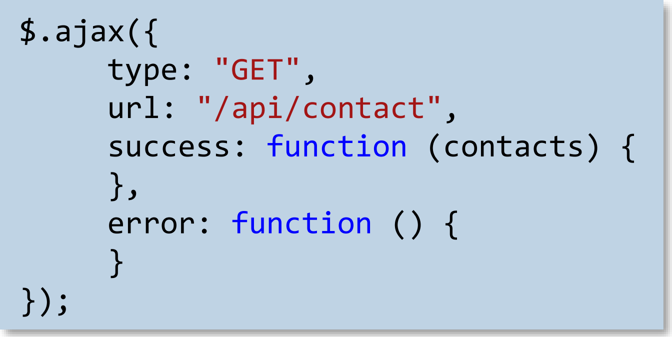
```
$.ajax({
     type: "GET",
     url: "/api/contact",
     success: function (contacts) {
     },
});
```

- □ Becomes

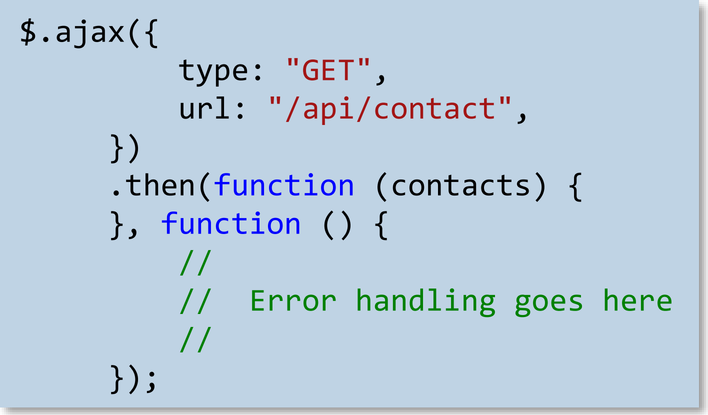


```
$.ajax({
         type: "GET",
         url: "/api/contact",
     })
     .then(function (contacts) {
     });
```

Basics – Handling Errors



```
$.ajax({
     type: "GET",
     url: "/api/contact",
     success: function (contacts) {
     },
     error: function () {
     }
});
```



```
$.ajax({
         type: "GET",
         url: "/api/contact",
     })
     .then(function (contacts) {
     }, function () {
         //
         //  Error handling goes here
         //
     });
```

- Doesn’t feel like a big improvement …

Layered Application

```
BL.getContacts = function (success, error) {
    DAL.getContacts(function (contacts) {
        if (success) {
            success(transform(contacts));
        }
    },
    function (err) {
        if (error) {
            error(err);
        }
    });
}
```

```
BL.getContacts = function () {
    return DAL.getContacts().then(transform);
}
```

- Now it shines ☺

Exception Handling

```
BL.getContacts = function (success, error) {
    DAL.getContacts(function (contacts) {
        if (success) {
            var transformed;
            try {
                transformed = transform(contacts);
            }
            catch (err) {
                if (error) {
                    error(err);
                }
                return;
            }
            success(transformed);
        }
    }, function (err) {
        if (error) {
            error(err);
        }
    });
}
```

```
BL.getContacts = function () {
    return DAL.getContacts().then(transform);
}
```

- Promise based code does not require any change !!!

Aggregation

```
function doTwoThingsAsync(success, error) {
     var count = 0;
     var res = [];

     do1(function (data) {
         res[0] = data;
         if (++count == 2) {
             success(res);
         }
     }, function (err) {
         error(err);
     });

     do2(function (data) {
         res[1] = data;
         if (++count == 2) {
             success(res);
         }
     }, function (err) {
         error(err);
     });
}
```

```
function doTwoThingsAsync() {
     return Q.all([do1(), do2()]);
}
```

- Since an operation is represented as an object we can build common methods like all

Caching

- Suppose our client requests some data from us
- Data is not available and therefore an async operation is initiated
- Data is cached
- Next time our client is asking the data we can return the cached data synchronously
- This might be confusing from client perspective
  - Async vs. Sync behavior

Caching

- Can wrap simple JavaScript object as a promise
  - Promise is considered resolved
- But still, then handler is invoked on browser’s next event loop

```
function getData() {
    if (data) {
        return Q.when(data);
    }

    return initiateAsyncOperation()
        .then(function (result) {
            return data = result;
        });
}
```

then fail fin

- The promise based equivalent to try … catch … finally

```
doSomethingAsync()
     .then(function (data) {
         throw new Error("Ooops");
     })
     .fail(function (err) {
     })
     .fin(function () {
         //
         //  Is always being executed
         //
     });
```

- On modern browsers we can use
  - catch instead of fail
  - finally instead of fin

spread

- □ Q.all returns an array of values
- □ This might be inconvenient

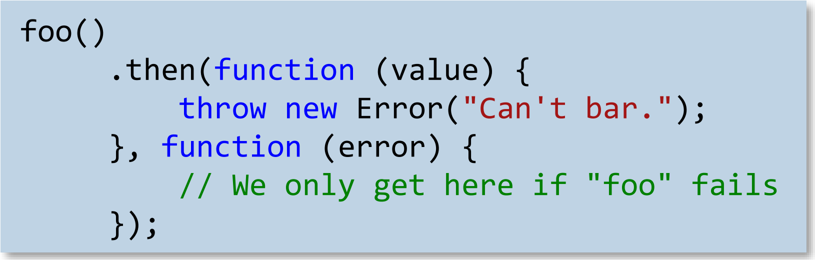
```
function doTwoThingsAsync() {
     return Q.all([do1(), do2()])
         .then(function (arr) {
             var res1 = arr[0];
             var res2 = arr[1];
         }); }
```

- □ You can use spread instead

```
function doTwoThingsAsync() {
     return Q.spread([do1(), do2()])
         .then(function(res1, res2) {
         });
}
```

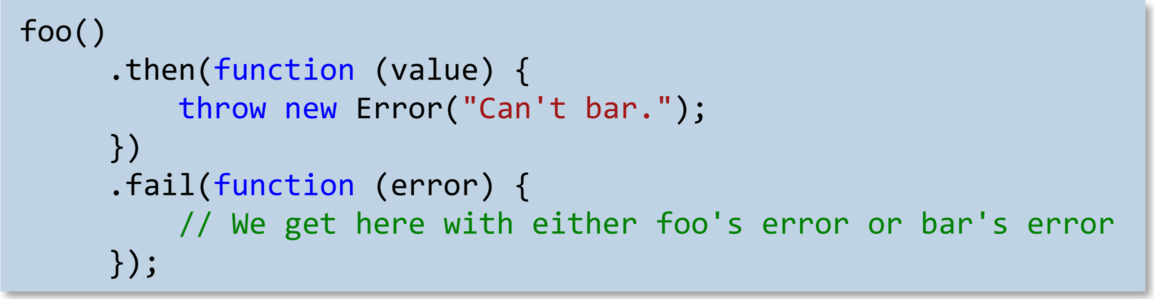
Handle Error – Be aware

- □ Error handler below does catch the error



```
foo()
     .then(function (value) {
         throw new Error("Can't bar.");
     }, function (error) {
         // We only get here if "foo" fails
     });
```

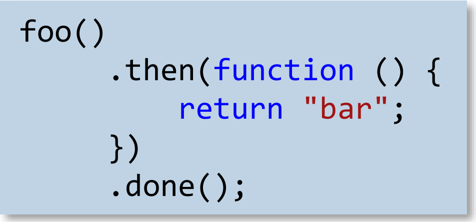
- □ Can use chaining



```
foo()
     .then(function (value) {
         throw new Error("Can't bar.");
     })
     .fail(function (error) {
         // We get here with either foo's error or bar's error
     });
```

Don’t loose your exceptions

- The top most application layer is usually responsible for handling errors
- Avoiding a fail handler means that an error might be unnoticed
- At the minimum use a done handler



```
foo()
     .then(function () {
         return "bar";
     })
     .done();
```

- The exception will be re-thrown and reported as unhandled exception

From function to promise

- Q.fcall gets a function and returns a promise
- The specified function is executed in the next event loop and may return
  - Simple value → Promise is resolved
  - Exception → Promise is rejected
  - Promise → As is

```
function doSomething() {
     if (notNow) {throw new Error("Not now");}

     if (data) {return data;}

     return getDataFromServer();
}
```

```
Q.fcall(doSomething)
     .then(function (data) {
         console.log("DONE");
     }, function (err) {
         console.log("ERR");
     }
);
```

Deferred

- An object that represent an asynchronous operation
- As opposed to promise a deferred object can be rejected/resolved
- A deferred object can be converted to a promise
  - But not vice versa
- A promise may be considered as the read-only API to a deferred object

Deferred

- □ Suppose we want to wrap an old fashion async API named FS.readFile

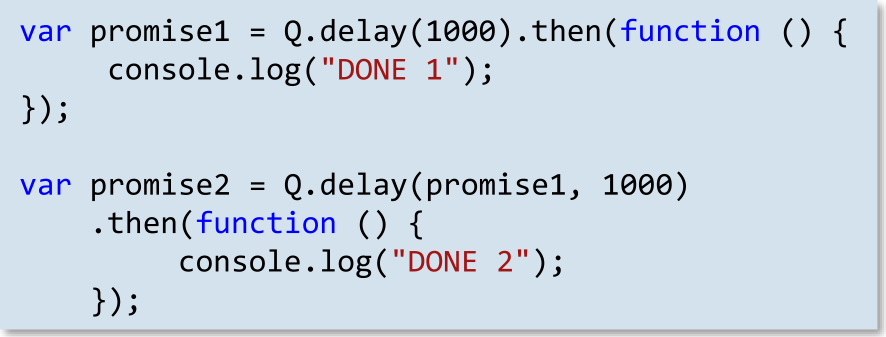
```
function readFile(fileName, encoding) {
    var deferred = Q.defer();

    FS.readFile(fileName, encoding, function (error, text) {
        if (error) {
            deferred.reject(error);
        } else {
            deferred.resolve(text);
        }
    });

    return deferred.promise;
}
```

Promise State

- isFullfilled – Returns true for resolved promise or simple value
- isRejected – Returns true for rejected promise
- isPending – Promise is still executing
- inspect – Returns an object which describe the promise state
  - state – “pending”, “fulfilled”, “rejected”
  - value – Only when resolved
  - Reason – Only when rejected

Q.delay

- Small wrapper around setTimeout
- Can wrap an existing promise
  - Thus delaying a successful operation



```
var promise1 = Q.delay(1000).then(function () {
     console.log("DONE 1");
});

var promise2 = Q.delay(promise1, 1000)
    .then(function () {
        console.log("DONE 2");
    });
```

getUnhandledReasons

- A failed promise does not cause unhandled exception
- It is the developer responsibility to catch the error using fail/done
- getUnhandledReasons allows you to get a list of all failures that were not handled by the developer
  - This is usually an indication of a bug

```
var reasons = Q.getUnhandledReasons();
console.log(reasons);
```

Summary

- Promise API makes your code cleaner
- The pattern can be used even with native code
- The idea is simple
  - Use an object to represent an action
- Q is not the only implementation of Promise API for JavaScript
  - Angular has its own implementation