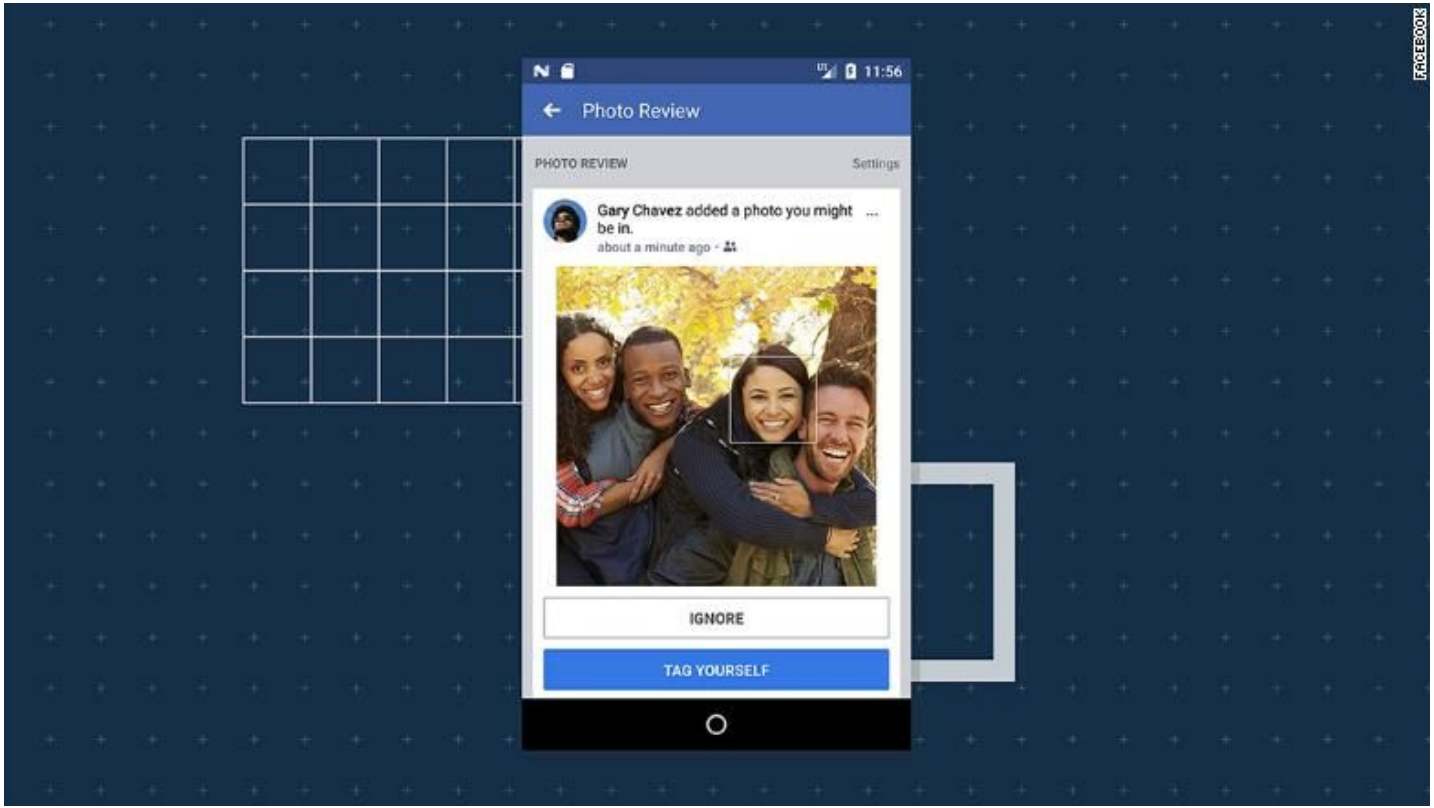
Day 3

Asynchronous JavaScript

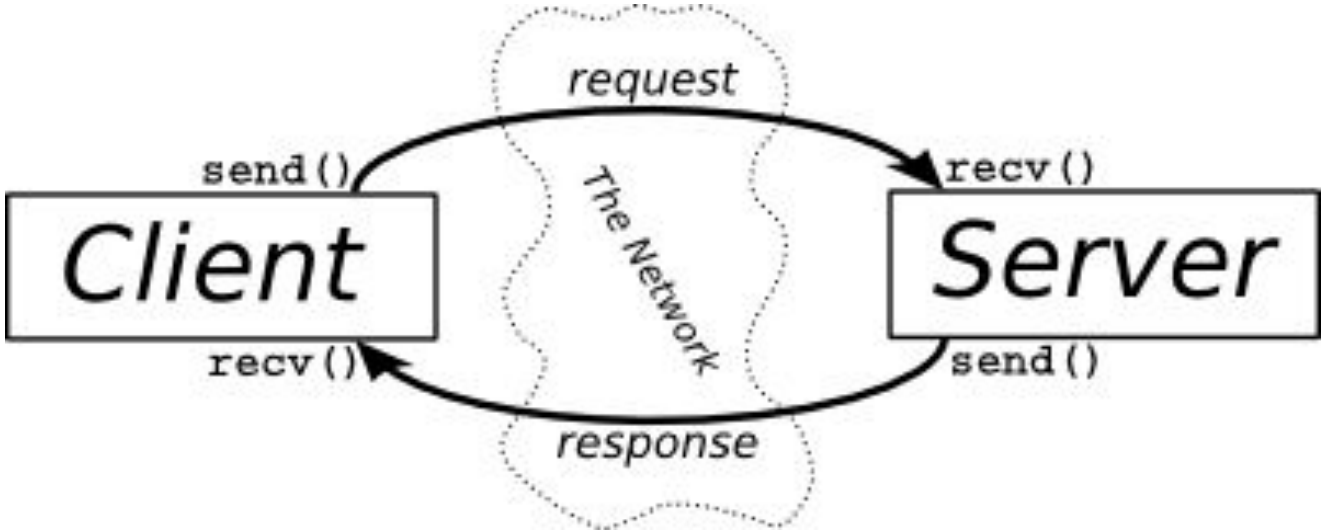
Resources

- **JavaScript Run-time:** https://developer.mozilla.org/en-US/docs/Web/JavaScript/EventLoop
- **Promises:** https://developers.google.com/web/fundamentals/primers/promises
- **AngularJS promises:** https://thinkster.io/a-better-way-to-learn-angularjs/promises
- **Async Javascript - callbacks vs. promises:** https://www.pluralsight.com/guides/introduction-to-asynchronous-javascript

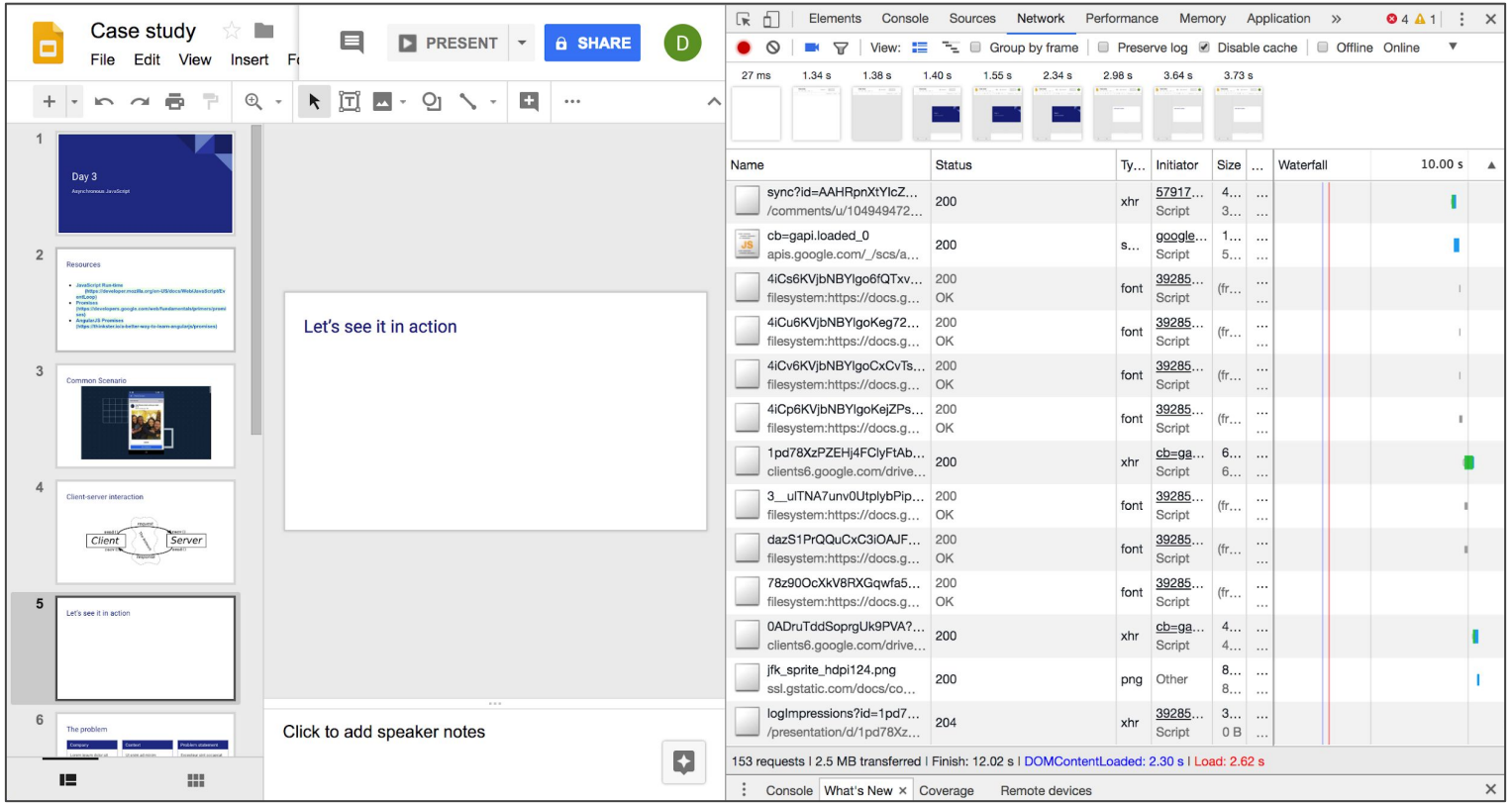
Common Scenario



Client-server interaction



Let’s see it in action



Let’s see it in action

▼ General

Request URL: https://docs.google.com/comments/u/104949472398518478618/d/AAHRpnXtYIcZWODAZYO9EtZkzoRW_N06Xcr_kk2pZ1YH9aKe7taVLcdO5ZEN1ATPCLgh3Y3ZgZEpXDRhWvYG2X8V9hyFrngWjOWurNARNCxYxRgxqflJ-OMU/docos/p/sync?id=AAHRpnXtYIcZWODAZYO9EtZkzoRW_N06Xcr_kk2pZ1YH9aKe7taVLcdO5ZEN1ATPCLgh3Y3ZgZEpXDRhWvYG2X8V9hyFrngWjOWurNARNCxYxRgxqflJ-OMU&sid=1a2ca1736e9e0df2&c=0&w=0&smv=4&token=AGNctVbEzr6FOu10603tPQLSHU2nJx1TWg%3A1525874857883

Request Method: POST

Status Code: 200

Remote Address: 172.217.13.110:443

Referrer Policy: no-referrer-when-downgrade

▼ Response Headers

alt-svc: hq=":443"; ma=2592000; quic=51303433; quic=51303432; quic=51303431; quic=51303339; quic=51303335,quic=":443"; ma=2592000; v="43,42,41,39,35"

cache-control: no-cache, no-store, max-age=0, must-revalidate

content-disposition: attachment; filename="response.bin"; filename*=UTF-8''response.bin

content-encoding: gzip

content-type: application/json; charset=utf-8

date: Wed, 09 May 2018 14:07:46 GMT

expires: Mon, 01 Jan 1990 00:00:00 GMT

pragma: no-cache

server: GSE

set-cookie: S=comments=gyIKjyb3rOUWPn0sH0he0y-byjDQ0nEA; Domain=.docs.google.com; Expires=Wed, 09-May-2018 14:22:46 GMT; Path=/comments/u/104949472398518478618/d/AAHRpnXtYIcZWODAZYO9EtZkzoRW_N06Xcr_kk2pZ1YH9aKe7taVLcdO5ZEN1ATPCLgh3Y3ZgZEpXDRhWvYG2X8V9hyFrngWjOWurNARNCxYxRgxqflJ-OMU; Secure; HttpOnly; Priority=LOW

set-cookie: SIDCC=AEfoLeYe4XNG5YcHwU1uFhsp-EClPNrHJLI-9o_YFgawAceEv6hWslYc5B2WYSRAroJoJImEoFGX; expires=Tue, 07-Aug-2018 14:07:46 GMT; path=/; domain=.google.com; priority=high

status: 200

Let’s see it in action

Requirements

- A browser is constantly loading data dynamically.
- We would like to have a “non-blocking” UI which always offers the user interaction, even as data is getting prepared in the background.

Requirements

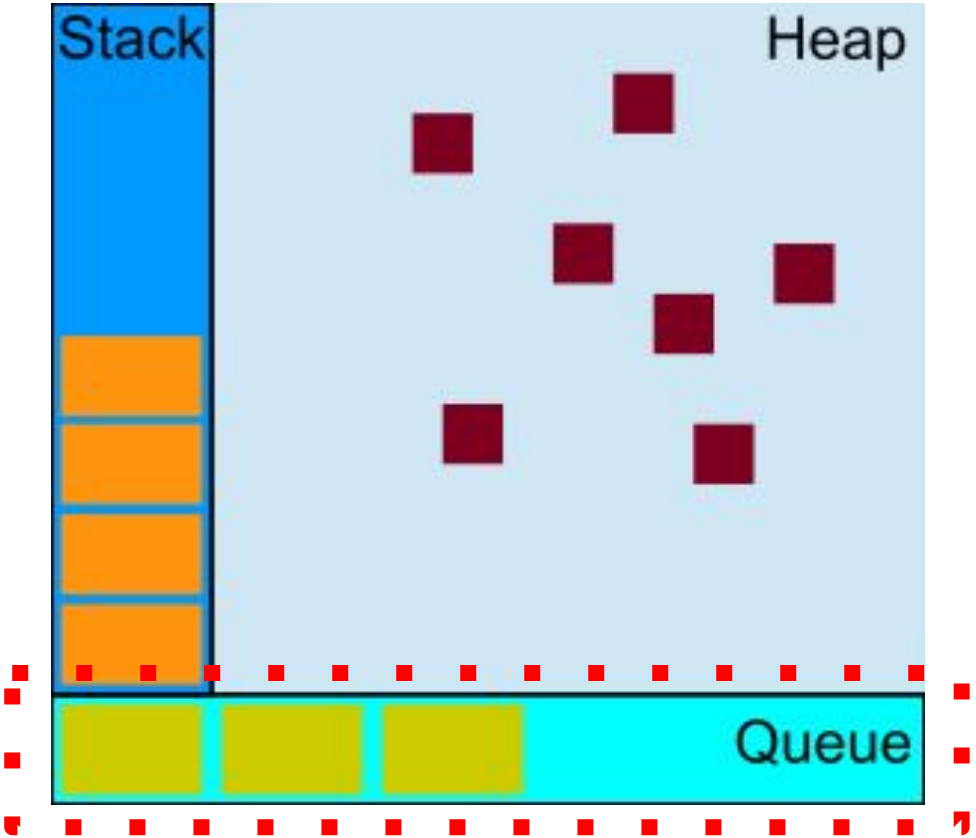
- A browser is constantly loading data dynamically.
- We would like to have a “non-blocking” UI which always offers the user interaction, even as data is getting prepared in the background.

Solution

- A language whose semantic model is built to accommodate this nature. This is where **JavaScript** comes in.

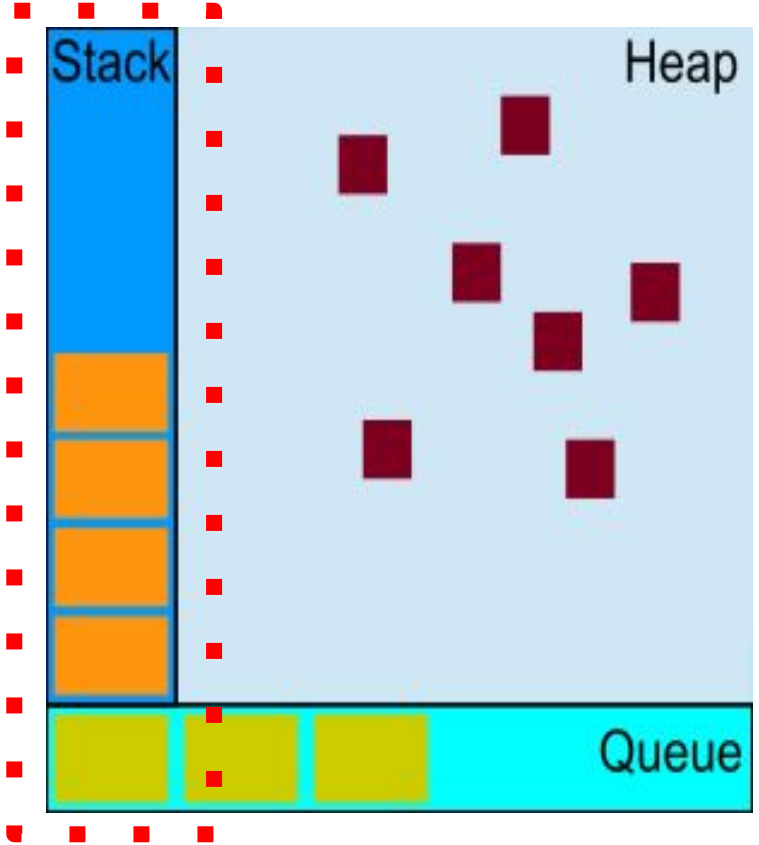
JavaScript run-time

The Run-time



The stack

- Controls actual execution in JavaScript
- Functions are pushed onto it as execution progresses.
- Only one function is executed at a time
- If a function is long-lasting it **will** block the UI
- Code executes as you would expect, **in order.**



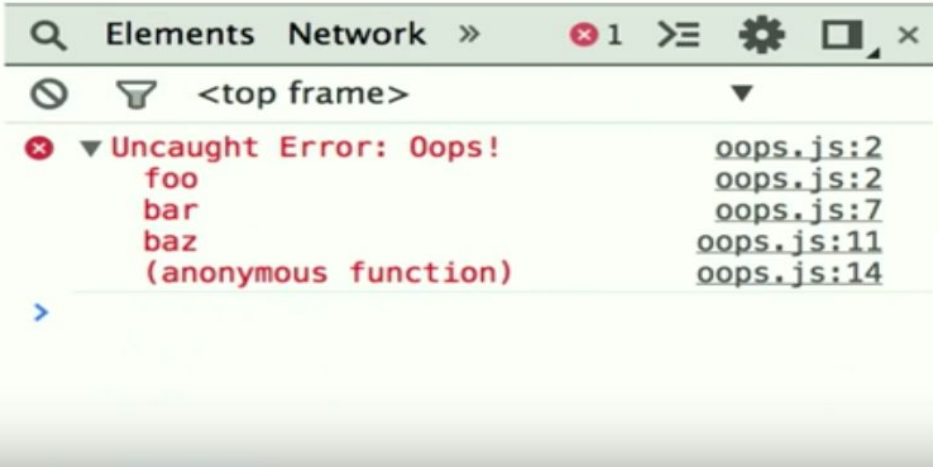
The stack

```
function foo() {
  throw new Error('Oops!');
}


function bar() {
  foo();
}

function baz() {
  bar();
}

baz();
```



RangeError: Maximum call stack size exceeded

Asynchronous code

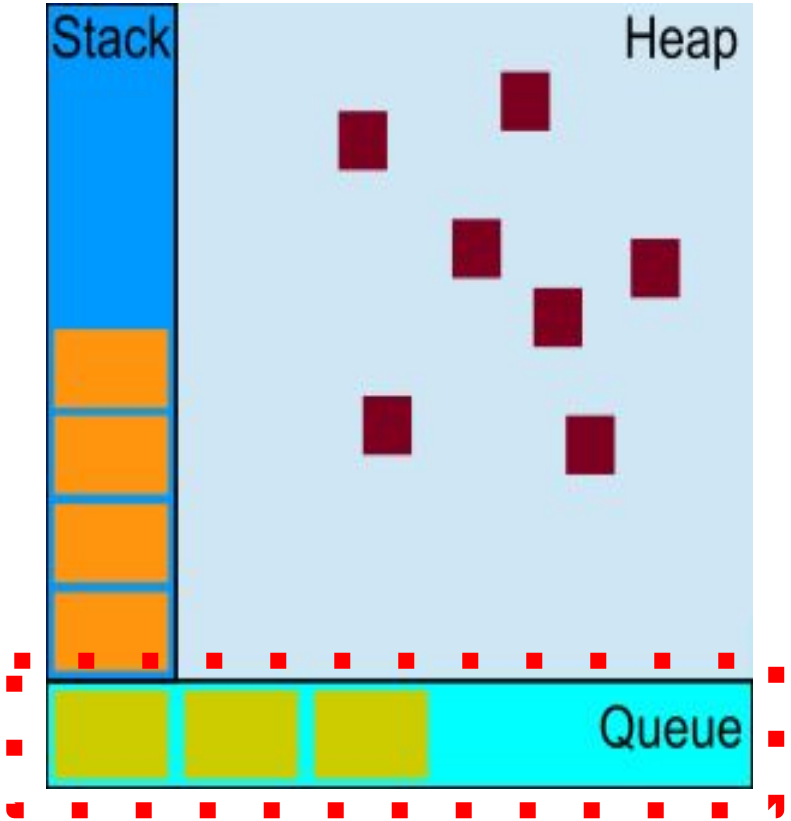
```
function foo()
{
    console.log('foo')
}
function bar()
{
    console.log('bar')
}
function getImage()
{
    fetch("image.png")
        .then(function(image_response){
            console.log("image");
        });
}
// Execution
foo();
getImage();
bar();
// Output
/*
* foo
* bar
* image
*/
```

Callback

```
.then(function(image_response){
    console.log("image");
});
```

The queue

- The **queue** keeps track of all the **callbacks** in asynchronous requests.
- Waits for the stack to be empty before requesting to place the **callback** back onto the stack to be executed



Handling Async code

The problem

- We have many asynchronous call made continuously or at the same time. How do we handle this in JavaScript?

Solution

- Many ways to do this in JavaScript
  - Events and callbacks
  - **Promises**
  - await/sync
- **AngularJS** uses promises, we will focus on this one.
- You can transform all this representations into one another. Is about **expressivity** and **clarity**!!!
- Reference: https://www.pluralsight.com/guides/introduction-to-asynchronous-javascript

What is a promise?

- **Definition:** A promise represents the eventual result of an asynchronous operation. It is a placeholder into which the successful result value or reason for failure will materialize
- A promise contains three states:
  - Pending
  - Resolved
  - Failed

```
// Simple GET request example:
$http({
    method: 'GET',
    url: '/someUrl'
}).then(function successCallback(response) {
    // this callback will be called asynchronously
    // when the response is available
}, function errorCallback(response) {
    // called asynchronously if an error occurs
    // or server returns response with an error status.
});
```

How do we ***promisify***?

Promisifying: Converting async code into a promise.

Procedure

- Wrap the async code around a promise.
- In AngularJS we use the **$q** dependency.
- Reference:
  - https://docs.angularjs.org/api/ng/service/$q

```
// Suppose function okToGreet exists
function asyncGreet(name) {
    var deferred = $q.defer();

    setTimeout(function() {
        if (okToGreet(name)) {
            deferred.resolve('Hello, ' + name + '!');
        } else {
            deferred.reject('Greeting ' + name +
                ' is not allowed.');
        }
    }, 1000);
    return deferred.promise;
}
```

Opal promise creation example

```
function requestToServer(request, params)
{
    var deferred = $q.defer();
    var db = firebase.database();
    var key = db.set("request",{"name":request, parameters:params});
    db.ref("response"+"/"+key).once("value", function(snapshot){
        deferred.resolve(snapshot.value());
    }).catch(function(err){
        deferred.reject(err);
    });
    return deferred.promise;
}
```

How do we call a promise?

- Once we have *'promisified'* a function, how do we call it?
  - Use the **then/catch** promise semantics

```
// Suppose function okToGreet exists
function asyncGreet(name) {
    var deferred = $q.defer();

    setTimeout(function() {
        if (okToGreet(name)) {
            deferred.resolve('Hello, ' + name + '!');
        } else {
            deferred.reject('Greeting ' + name +
                ' is not allowed.');
        }
    }, 1000);
    return deferred.promise;
}
```

```
asyncGreet('Robin Hood')
    .then(function(greeting){
    alert('Success: ' + greeting);
    }).catch(function(error){
    alert('Failed: ' + reason);
});
```

Common Opal promises

- $http.get
- Firebase
- The whole back-end!

```
// Simple GET request example:
$http({
    method: 'GET',
    url: '/someUrl'
}).then(function successCallback(response) {
    // this callback will be called asynchronously
    // when the response is available
}, function errorCallback(response) {
    // called asynchronously if an error occurs
    // or server returns response with an error status.
});
```

```
requestToServer("GetConversations",{userId:1})
    .then(function(response){
        // Handle conversations
    })
    .catch(function(error){
        // Handle error
    });
```

Common async scenarios

Cases

- **Scenario 1:** One simple async request (Done)
- **Scenario 2:** Two or more simple requests that **depend** on one another
- **Scenario 3:** Two or more simple requests that **do not depend** on one another.
- Every other scenario is a combination of this tree.

Scenario 1

- **Description:** A simple async request

- **Procedure:**
  1. Promisify request (if not promisified)
  2. Use **then/catch**

Scenario 1 - Example

```
fetchUrlContent(imageUrl)
    .then(function(content){

    }).catch(function(error){

    });
```

Scenario 2

- **Description**: Two or more simple async requests that depend on one another

- **Procedure:**
    1. Promisify the requests (if not promisified)
    2. Chain them one after the other.

Scenario 2 - example

```
// Assume getImages function exists, which fetches
// the images from conversations
requestToServer("GetConversations",{userId:1})
    .then(function(response){

        return getImages(response.data.conversations);
    }).then(function(conversationsWithImages){
        // Handle conversations
    })
    .catch(function(error){ alert(error); });
```

Scenario 3

- **Description:** Two or more simple async requests that *do not depend* on one another. (order of arrival does not matter)

- **Procedure**
  1. Promisify requests (if not promisified)
  2. Create array of unresolved promises with each image
  3. Use **$q.all()**
- **Common cases:**
  - Fetching images for each of the conversations. These do not depend on one another but must all return before you can use conversations.

Scenario 3 - Example

```
function getImages(conversations){
    var promiseArray = [];
    for(var i = 0; i< conversations.length; i++)
    {
        promiseArray.push(fetchUrlContent(conversations[i].imageUrl));
    }
    return $q.all(promiseArray).then(function(images){
        images.forEach(function(image,index){
            conversations[index].image = image;
        });
        return images;
    });
}
```

Last comments

- You will encounter this repeatedly as a web-developer!
- Read references at the beginning of presentation
- Exercises will come soon!
- I will also put together a README.md giving more context and detail.
- If you master this, you are well on your way with JavaScript :)

The End - Thank you!