COMMUNICATING WITH A BACK END SERVER

Ori calvo @2015

http://oricalvo.com
ori.calvo@gmail.com

Objectives

- Managing asynchronous functions using promises
- Getting to know more Angular built-in services
  - $http
  - $resource
  - $q

Communicating with a Back-end server

- $.ajax is not supported out of the box
- However, you can include jquery.js script and start using $.ajax

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

```
<div class="home-view">
    <h1>Contact</h1>
    <div>{{status}}</div>
    <ul>
        <li ng-repeat="contact in contacts">
            <span>{{contact.Name}}</span>
        </li>
    </ul>
</div>
```

- Why is HTML not updated?

Asynchronous JavaScript

- Angular does not know $.ajax
- This means that when $.ajax completes, Angular has no idea that it has to refresh the HTML
  - This is true for any asynchronous operation initiated outside of Angular (for example, setTimeout)
- Two solutions
  - Use $scope.$apply
  - Use $http service

```
$.ajax({
     type: "GET",
     url: "/api/contact",
     complete: function () {
         $scope.$apply();
     }
});
```

http://oricalvo.com
ori.calvo@gmail.com

$http

- Angular offers a general purpose $http service
- $http Encapsulates XMLHttpRequest work
- Behaves similar to $.ajax but not the same
- Integrates better with other Angular's facilities
  - Automatic DOM update
  - $q service
  - Interceptors
- In addition, Angular offers a $resource service to easily target RESTFul services

http://oricalvo.com
ori.calvo@gmail.com

$http

- Has one method for each HTTP verb
- **GET**: $http.get(url, config)
- **POST**: $http.post(url, data, config)
- **PUT**: $http.put(url, data, config)
- **DELETE**: $http.delete(url, config)
- **HEAD**: $http.head
- **JSONP**: $http.jsonp(url, config)

$http


```
$http.get("/api/contact")
    .success(function (data, status, headers, config) {
        $scope.contacts = data;
        $scope.status = "Done";
     })
     .error(function (data, status, headers, config) {
        $scope.status = "Error";
     });
```

```
$http.post("/api/contact", { name: "Ori" })
     .success(function (data, status, headers, config) {
         $scope.status = "Transfer completed";
     })
     .error(function (data, status, headers, config) {
         $scope.status = "Transfer failed";
     });
```

- No need to tell Angular to update the DOM

http://oricalvo.com
ori.calvo@gmail.com

$http Configuration

- Each $http method receives additional parameter named config
- This is a simple JavaScript options object with the following properties
  - **method:** HTTP verb
  - **url**
  - **params:** Parameters to be added to the URL query string
  - **headers:** Additional HTTP headers
  - **timeout**
  - **cache:** disable XHR GET request caching
  - **transformRequest/Response:** Pre-process and post-process data transformation

Request Transformation

- $http.post and $http.put methods accept
  - Any JavaScript object
  - string
- If data is object it will by converted to JSON
- The conversion mechanism ignores all properties beginning with $
  - Those are considered private
  - Might be problematic for some back-ends (MongoDB) which have special treat for $ parameters
- You may consider using JSON.stringify and send only strings to Angular

Custom Transformation

- Use $httpProvider.defaults.transformRequest
- Can use it as an array. You may push/unshift new transformations
- Can use it as a single transformation function

```
$httpProvider.defaults.transformResponse.push(function (data) {
    if (angular.isArray(data)) {
        for (var i = 0; i < data.length; i++) {
            var obj = data[i];
            for (var key in obj) {
                if (key.indexOf("Date") != -1) {
                    obj[key] = new Date(obj[key]);
                }
            }
        }
    }

    return data;
});
```

HTTP response

- Both success and error callbacks support the following parameters
  - **data:** The actual response data
    - May be not a string → Depends on the response content-type
  - **status:** The HTTP status code
    - 200 to 299 are treated as success
    - 3xx (redirect) are automatically followed by XMLHttpRequest
  - **headers:** A function giving access to the HTTP response headers
  - **config:** The same configuration object that was supplied when sending the request

```
.success(function (data, status, headers, config) {
    $scope.status = "Transfer completed";
})
```

Request Headers

- The $http service automatically add certain headers to all requests
  - Accept: application/json, text/plain, */*
  - Content-Type: application/json (POST and PUT only)
- You can change defaults using $httpProvider

```
angular.module("myApp", [])
     .config(function ($routeProvider, $locationProvider, $httpProvider) {
         $httpProvider.defaults.headers.common.MyHeader = "Blabla";
     });
```

Caching

- Set $httpProvider.defaults.cache to true
- Angular caches the http response per URL and reuses it

```
angular.module("myApp", [])
     .config(function ($httpProvider) {
         $httpProvider.defaults.cache = true;
     });
```

- No expiration time
- You can set a custom cache object

$cacheFactory

- A service which knows how to create cache objects
- Receives cache id and returns a new cache object
- A cache object supports the following API
  - put(string, value)
  - get(string)
  - remove(string), removeAll()

```
angular.module("myApp", [])
     .config(function () {
     })
     .run(function ($http, $cacheFactory) {
         $http.defaults.cache = $cacheFactory({
             capacity: 5,
         });
     });
```

Custom Cache


```
angular.module("myApp").factory("CustomCacheFactory", function () {
     return function () {
         return new CustomCache();
     }
});
```

```
angular.module("myApp", [])
     .config(function () {
     })
     .run(function ($http, CustomCacheFactory) {
         $http.defaults.cache = CustomCacheFactory();
     });
```

http://oricalvo.com
ori.calvo@gmail.com

XSRF

- A technique by which an unauthorized site can gain your user's private data
- Websites typically don't verify that a request came from an unauthorized user
- Instead they verify only that a request came from the browser of an authorized user
- Attacker need to
  - Convince your user to click on an HTML link/image
  - The link sends an HTTP request with a known side effects like: password reset, sending email

XSRF Protection

- $http service reads a token from a cookie
  - XSRF-TOKEN
- Set it is an HTTP header
  - X-XSRF-TOKEN
- Your server should validate each request for the appropriate X-XSRF-TOKEN value
- The token must be unique for each user
- Must be a value that is hard to guess

Communicating with RESTful Service

- $http can be used easily to communicate with RESTful service
- However, Angular goes one step further and provides a dedicated $resource service
- $resource eliminates repetitive code
- Provides a higher abstraction level of data manipulation
  - Is focused around objects instead of $http calls
- $resource is useless when communicating with RPC services

$resource – Getting Started

- $resource is distributed in a separate file named **angular-resource.js**
- Resides in a dedicated module named ngResource
  - You need to declare the dependency

```
angular.module("myApp", ["ngRoute", "ngResource"])
     .config(function ($routeProvider, $locationProvider) {
     });
```

```
angular.module("myApp").controller("HomeCtrl", function ($scope, $resource) {
    var contacts = $resource("/api/contact");

    $scope.contacts = contacts.query();
});
```

$resource – Class Methods

- $resource returns a constructor function
- Can use class methods on the constructor itself
  - **query**(params, success, error)
  - **get**(params, success, error)
  - **save**(params, data, success, error)
  - **delete**(params, success, error)

```
var queryString = { active: true };
var requestBody = { name: "Ori" };

Contact.save(queryString, requestBody, function () {
     $scope.status = "Done";
});
```

$resource – Instance Methods

- The constructor can be used to create new $resource object
- Offers the same API
  - **$get**(params, success, error)
  - **$save**(params, success, error)
  - **$delete**(params, success, error)

```
var Contact = $resource("/api/contact");

var contact = new Contact({
     Name: "Carmit" });

contact.$save({}, function () {
     $scope.status = "Done";
});
```

Instance vs. Class

- Which API should we use ?
- In most cases this is just a matter of style
- However, instance methods are smarter
  - $save sends HTTP POST request to the server
  - It merges back the response into the object itself
  - For example, auto generated ID field

```
var contact = new Contact({
     Name: "Carmit",
});
console.log(!!contact.ID); //false
contact.$save({}, function () {
     console.log(!!contact.ID); //true
});
```

Parameterized URL

- URL may contain placeholders
- $resource replaces placeholder with value specified at the class/object level
- Unresolved placeholders are omitted

```
var Contact = $resource("/api/:entity/:id", {
     entity: "contact",
     id: "@ID",
});

var contact = new Contact({
     ID: 1,
});

contact.$get({}, function () {
     $scope.status = "Done";
});
```

Customize $resource object

- The constructor returned by $resource can be customized like any other JavaScript constructor
- In addition, simple customization can be done at the $resource invocation

```
var Contact = $resource("/api/:entity/:id", {
                    entity: "contact",
                    id: "@ID",
                }, {
                    update: { method: "PUT" }
                });

Contact.prototype.getDisplayName = function () {
     return this.Name + ", " + this.EMail;
}
```

```
var contact = new Contact({
     ID: 1,
     Name: "XXX"
});

contact.$update();
```

http://oricalvo.com
ori.calvo@gmail.com

$q

- The docs say "A service that helps you run functions asynchronously"
  - Implies that $q knows how to make a function asynchronous
- A better explanation would be: "A service that helps you better manage asynchronous function"

http://oricalvo.com
ori.calvo@gmail.com

Promise

- No single formal definition for promise object
  - Promises/A/B/KISS/C/D
- Many libraries
  - Q, RSVP,when.js, FutureJS
  - Angular mimics Q library (by Kris Kowal)
- All implementations agree that a promise object is an interface for interacting with asynchronous operation

http://oricalvo.com
ori.calvo@gmail.com

Moving to Promise

- Stop using success and error callbacks
- Instead create a deferred object
  - Has a state
  - Can be resolved/rejected
- Return to client the promise projection
  - Cannot be changed
  - Only allows for registering handlers for success and error

From Callbacks to Promise

Callback based async function

```
function asyncFunc(success, error) {
    setTimeout(function () {
        if (Math.ceil(Math.random() * 2) % 2 == 0) {
            success();
        }
        else {
            error(new Error("Ooops"));
        }
    }, 1000);
}
```

Promise based async function

```
function asyncFuncEx() {
    var deferred = $q.defer();
    asyncFunc(
        function success() {
            deferred.resolve();
        },
        function error(err) {
            deferred.reject(err);
        }
    );
    return deferred.promise;
}
```

Client usage

```
asyncFuncEx()
    .then(function () {
        console.log("Success");
    })
    .catch(function (err) {
        console.log("Error: " + err.message);
    });
```

http://oricalvo.com
ori.calvo@gmail.com

Deferred vs. Promise

- Deferred
  - Is writable
  - Caller can change state by rejecting/resolving the promise
  - Once state is determined cannot change it again
- Promise is associated to exactly one Deferred object
  - Is read only
  - “Sees” the state
  - However, cannot modify it

Chaining Promises

- The then function returns a new promise
  - Always
- The new promise is effected by
  - The end result of the original promise
  - The return value/exception of the then handler
- Which means that both promises may have different status/result

Changing Returned Value

- What is the output ?

```
var promise = asyncFunc();

var newPromise = promise.then(function (result) {
     console.log(result);
     return result % 2;
});

newPromise.then(function (result) {
     console.log(result);
});

function asyncFunc() {
     var deferred = $q.defer();

     setTimeout(function () {
         var num = Math.ceil(Math.random() * 1000);
         deferred.resolve(num);
     }, 50);

     return deferred.promise;
}
```

Throwing Exception

- Original promise is resolved
- Then handler throws an exception → New promise is rejected

```
var promise = asyncFunc();

var newPromise = promise.then(function (result) {
     throw new Error("Ooops");
});

newPromise
     .then(function (result) {
         console.log("SUCCESS: " + result);
     })
     .catch(function (err) {
         console.log("ERROR: " + err.message);
     });
```

```
function asyncFunc() {
     var deferred = $q.defer();
     setTimeout(function () {
         var num = Math.ceil(Math.random() * 1000);
         deferred.resolve(num);
     }, 50);
     return deferred.promise;
}
```

$q.reject

- Rejecting a then handler can be done using $q.reject
  - Instead of throwing an exception

```
var promise = asyncFunc();

var newPromise = promise.then(function (result) {
     return $q.reject(new Error("Ooops"));
});

newPromise
     .then(function (result) {
         console.log("SUCCESS: " + result);
     })
     .catch(function (err) {
         console.log("ERROR: " + err.message);
     });
```

$q.reject vs. Throwing Exception

- Both techniques have the same effect
- The main difference is logging
- Angular assumes that an exception being thrown from a then handler is “by mistake” and therefore delegates it to the $exceptionHandler service

```
<top frame>                    Preserve log
HomeCtrl created
  ▶ Error: blabla
      at HomeCtrl.js:18
      at processQueue (angular.js:13189)
      at angular.js:13205
      at Scope.$get.Scope.$eval (angular.js:14401)
      at Scope.$get.Scope.$digest (angular.js:14217)
      at angular.js:14440
      at completeOutstandingRequest (angular.js:4905)
      at angular.js:5285
Error: blabla
>
```

$q.when

- Returns a resolved promise
- The specified parameter is considered to be the promise result

```
ContactStore.prototype.getAll = function () {
     var me = this;

     if (me.contacts) {
         return me.$q.when(me.contacts);
     }

     return me.$http.get("/api/contact")
         .then(function (response) {
             return me.contacts = response.data;
         });
}
```

```
function when(value, callback, errback, progressBack) {
     var result = new Deferred();
     result.resolve(value);
     return result.promise.then(callback, errback, progressBack);
 };
```

$q.all

- Aggregate multiple promises into one
- The "combined" promise is resolved only if all sub promises are resolved

```
ContactStore.prototype.getContactsAndProducts = function () {
    var promise1 = this.$http.get("/api/contact")
        .then(function (response) {
            return response.data;
        });

    var promise2 = this.$http.get("/api/product")
        .then(function (response) {
            return response.data;
        });

    return this.$q.all([promise1, promise2]).then(function (result) {
        return {
            contacts: result[0],
            products: result[1],
        };
    });
}
```

Promise is always asynchronous

- Even when resolving a promise immediately, Angular injects the listeners on the next digest cycle
- Uses $evalAsync

```
$scope.reload = function () {
    console.log("BEFORE getContacts");

    ContactStore.getContacts().then(
        function (contacts) {
            console.log("THEN getContacts");
            $scope.contacts = contacts;
        },
        function (err) {
            console.log("ERROR: " + err.message);
        });

    console.log("AFTER getContacts");
}
```

```
ContactStore.prototype.getContacts = function () {
    console.log("getContacts");
    return this.$q.when([{ ID: 1, Name: "Tommy" }]);
}
```

http://oricalvo.com
ori.calvo@gmail.com

Additional Services

- $compile
- $exceptionHandler
- $interpolate
- $log
- $parse
- $timeout/$interval

Summary

- Angular offers many built-in services
- You can register your own
  - And you should …
- Popular services are
  - $http
  - $resource
  - $q
  - $location