AngularJS: API: $http

1. - service in module ng

The `$http` service is a core Angular service that facilitates communication with the remote HTTP servers via the browser's XMLHttpRequest object or via JSONP.

For unit testing applications that use `$http` service, see $httpBackend mock.

For a higher level of abstraction, please check out the $resource service.

The $http API is based on the deferred/promise APIs exposed by the $q service. While for simple usage patterns this doesn't matter much, for advanced usage it is important to familiarize yourself with these APIs and the guarantees they provide.

General usage

The `$http` service is a function which takes a single argument — a configuration object — that is used to generate an HTTP request and returns a promise.

```
// 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.
  });
```

The response object has these properties:

- **data** – `{string|Object}` – The response body transformed with the transform functions.
- **status** – `{number}` – HTTP status code of the response.
- **headers** – `{function([headerName])}` – Header getter function.
- **config** – `{Object}` – The configuration object that was used to generate the request.
- **statusText** – `{string}` – HTTP status text of the response.

A response status code between 200 and 299 is considered a success status and will result in the success callback being called. Any response status code outside of that range is considered an error status and will result in the error callback being called. Also, status codes less than -1 are normalized to zero. -1 usually means the request was aborted, e.g. using a `config.timeout`. Note that if the response is a redirect, XMLHttpRequest will transparently follow it, meaning that the outcome (success or error) will be determined by the final response status code.

Shortcut methods

Shortcut methods are also available. All shortcut methods require passing in the URL, and request data must be passed in for POST/PUT requests. An optional config can be passed as the last argument.

```
$http.get('/someUrl', config).then(successCallback, errorCallback);
$http.post('/someUrl', data, config).then(successCallback, errorCallback);
```

Complete list of shortcut methods:

Writing Unit Tests that use $http

When unit testing (using ngMock), it is necessary to call $httpBackend.flush() to flush each pending request using trained responses.

```
$httpBackend.expectGET(...);
$http.get(...);
$httpBackend.flush();
```

Deprecation Notice

The `$http` legacy promise methods `success` and `error` have been deprecated. Use the standard `then` method instead. If `$httpProvider.useLegacyPromiseExtensions` is set to `false` then these methods will throw `$http/legacy` error.

Setting HTTP Headers

The $http service will automatically add certain HTTP headers to all requests. These defaults can be fully configured by accessing the `$httpProvider.defaults.headers` configuration object, which currently contains this default configuration:

- `$httpProvider.defaults.headers.common` (headers that are common for all requests):
  - `Accept: application/json, text/plain, * / *`
- `$httpProvider.defaults.headers.post`: (header defaults for POST requests)
  - `Content-Type: application/json`
- `$httpProvider.defaults.headers.put` (header defaults for PUT requests)
  - `Content-Type: application/json`

To add or overwrite these defaults, simply add or remove a property from these configuration objects. To add headers for an HTTP method other than POST or PUT, simply add a new object with the lowercased HTTP method name as the key, e.g. `$httpProvider.defaults.headers.get = { 'My-Header' : 'value' }`.

The defaults can also be set at runtime via the `$http.defaults` object in the same fashion. For example:

```
module.run(function($http) {
  $http.defaults.headers.common.Authorization = 'Basic YmVlcDpib29w';
});
```

In addition, you can supply a `headers` property in the config object passed when calling `$http(config)`, which overrides the defaults without changing them globally.

To explicitly remove a header automatically added via $httpProvider.defaults.headers on a per request basis, Use the `headers` property, setting the desired header to `undefined`. For example:

```
var req = {
 method: 'POST',
 url: 'http://example.com',
 headers: {
   'Content-Type': undefined
 },
 data: { test: 'test' }
}

$http(req).then(function(){...}, function(){...});
```

Transforming Requests and Responses

Both requests and responses can be transformed using transformation functions: `transformRequest` and `transformResponse`. These properties can be a single function that returns the transformed value (`function(data, headersGetter, status)`) or an array of such transformation functions, which allows you to `push` or `unshift` a new transformation function into the transformation chain.

Note: Angular does not make a copy of the `data` parameter before it is passed into the `transformRequest` pipeline. That means changes to the properties of `data` are not local to the transform function (since Javascript passes objects by reference). For example, when calling `$http.get(url, $scope.myObject)`, modifications to the object's properties in a transformRequest function will be reflected on the scope and in any templates where the object is data-bound. To prevent this, transform functions should have no side-effects. If you need to modify properties, it is recommended to make a copy of the data, or create new object to return.

Default Transformations

The `$httpProvider` provider and `$http` service expose `defaults.transformRequest` and `defaults.transformResponse` properties. If a request does not provide its own transformations then these will be applied.

You can augment or replace the default transformations by modifying these properties by adding to or replacing the array.

Angular provides the following default transformations:

Request transformations (`$httpProvider.defaults.transformRequest` and `$http.defaults.transformRequest`):

- If the `data` property of the request configuration object contains an object, serialize it into JSON format.

Response transformations (`$httpProvider.defaults.transformResponse` and `$http.defaults.transformResponse`):

- If XSRF prefix is detected, strip it (see Security Considerations section below).
- If JSON response is detected, deserialize it using a JSON parser.

Overriding the Default Transformations Per Request

If you wish to override the request/response transformations only for a single request then provide `transformRequest` and/or `transformResponse` properties on the configuration object passed into `$http`.

Note that if you provide these properties on the config object the default transformations will be overwritten. If you wish to augment the default transformations then you must include them in your local transformation array.

The following code demonstrates adding a new response transformation to be run after the default response transformations have been run.

```
function appendTransform(defaults, transform) {

  // We can't guarantee that the default transformation is an array
  defaults = angular.isArray(defaults) ? defaults : [defaults];

  // Append the new transformation to the defaults
  return defaults.concat(transform);
}

$http({
  url: '...',
  method: 'GET',
  transformResponse: appendTransform($http.defaults.transformResponse, function(value)
{
    return doTransform(value);
  })
});
```

Caching

`$http` responses are not cached by default. To enable caching, you must set the config.cache value or the default cache value to TRUE or to a cache object (created with `$cacheFactory`). If defined, the value of config.cache takes precedence over the default cache value.

In order to:

- cache all responses - set the default cache value to TRUE or to a cache object
- cache a specific response - set config.cache value to TRUE or to a cache object

If caching is enabled, but neither the default cache nor config.cache are set to a cache object, then the default `$cacheFactory("$http")` object is used.

The default cache value can be set by updating the `$http.defaults.cache` property or the `$httpProvider.defaults.cache` property.

When caching is enabled, `$http` stores the response from the server using the relevant cache object. The next time the same request is made, the response is returned from the cache without sending a request to the server.

Take note that:

- Only GET and JSONP requests are cached.
- The cache key is the request URL including search parameters; headers are not considered.
- Cached responses are returned asynchronously, in the same way as responses from the server.
- If multiple identical requests are made using the same cache, which is not yet populated, one request will be made to the server and remaining requests will return the same response.
- A cache-control header on the response does not affect if or how responses are cached.

Interceptors

Before you start creating interceptors, be sure to understand the $q and deferred/promise APIs.

For purposes of global error handling, authentication, or any kind of synchronous or asynchronous pre-processing of request or postprocessing of responses, it is desirable to be able to intercept requests before they are handed to the server and responses before they are handed over to the application code that initiated these requests. The interceptors leverage the promise APIs to fulfill this need for both synchronous and asynchronous pre-processing.

The interceptors are service factories that are registered with the `$httpProvider` by adding them to the `$httpProvider.interceptors` array. The factory is called and injected with dependencies (if specified) and returns the interceptor.

There are two kinds of interceptors (and two kinds of rejection interceptors):

- `request`: interceptors get called with a http config object. The function is free to modify the `config` object or create a new one. The function needs to return the `config` object directly, or a promise containing the `config` or a new `config` object.
- `requestError`: interceptor gets called when a previous interceptor threw an error or resolved with a rejection.
- `response`: interceptors get called with http `response` object. The function is free to modify the `response` object or create a new one. The function needs to return the `response` object directly, or as a promise containing the `response` or a new `response` object.
- `responseError`: interceptor gets called when a previous interceptor threw an error or resolved with a rejection.

```
// register the interceptor as a service
$provide.factory('myHttpInterceptor', function($q, dependency1, dependency2) {
  return {
    // optional method
    'request': function(config) {
      // do something on success
      return config;
    },

    // optional method
   'requestError': function(rejection) {
      // do something on error
      if (canRecover(rejection)) {
        return responseOrNewPromise
      }
      return $q.reject(rejection);
    },



    // optional method
    'response': function(response) {
      // do something on success
      return response;
    },

    // optional method
   'responseError': function(rejection) {
      // do something on error
      if (canRecover(rejection)) {
        return responseOrNewPromise
      }
      return $q.reject(rejection);
    }
  };
});

$httpProvider.interceptors.push('myHttpInterceptor');


// alternatively, register the interceptor via an anonymous factory
$httpProvider.interceptors.push(function($q, dependency1, dependency2) {
  return {
   'request': function(config) {
       // same as above
    },
```

```
    'response': function(response) {
       // same as above
    }
  };
});
```

Security Considerations

When designing web applications, consider security threats from:

Both server and the client must cooperate in order to eliminate these threats. Angular comes pre-configured with strategies that address these issues, but for this to work backend server cooperation is required.

JSON Vulnerability Protection

A JSON vulnerability allows third party website to turn your JSON resource URL into JSONP request under some conditions. To counter this your server can prefix all JSON requests with following string `")]}',\n"`. Angular will automatically strip the prefix before processing it as JSON.

For example if your server needs to return:

```
['one','two']
```

which is vulnerable to attack, your server can return:

```
)]}',
['one','two']
```

Angular will strip the prefix, before processing the JSON.

Cross Site Request Forgery (XSRF) Protection

XSRF is an attack technique by which the attacker can trick an authenticated user into unknowingly executing actions on your website. Angular provides a mechanism to counter XSRF. When performing XHR requests, the $http service reads a token from a cookie (by default, `XSRF-TOKEN`) and sets it as an HTTP header (`X-XSRF-TOKEN`). Since only JavaScript that runs on your domain could read the cookie, your server can be assured that the XHR came from JavaScript running on your domain. The header will not be set for cross-domain requests.

To take advantage of this, your server needs to set a token in a JavaScript readable session cookie called `XSRF-TOKEN` on the first HTTP GET request. On subsequent XHR requests the server can verify that the cookie matches `X-XSRF-TOKEN` HTTP header, and therefore be sure that only JavaScript running on your domain could have sent the request. The token must be unique for each user and must be verifiable by the server (to prevent the JavaScript from making up its own tokens). We recommend that the token is a digest of your site's authentication cookie with a salt for added security.

The name of the headers can be specified using the xsrfHeaderName and xsrfCookieName properties of either $httpProvider.defaults at config-time, $http.defaults at run-time, or the per-request config object.

In order to prevent collisions in environments where multiple Angular apps share the same domain or subdomain, we recommend that each application uses unique cookie name.