Strip Trailing Slashes

```
app.config(['$resourceProvider', function($resourceProvider) {
  // Don't strip trailing slashes from calculated URLs
  $resourceProvider.defaults.stripTrailingSlashes = false;
}]);
```

Default Custom Actions

```
app.config(['$resourceProvider', function($resourceProvider) {
  // use this to add more default actions so that all resources have these actions
  // I often use this for PATCH and PUT
  $resourceProvider.defaults.actions.patch = {
    method: 'PATCH'
  };
}]);
```

Defining

The Basics

JSON File

```
mod.factory('MyResource', function($resource) {
  return $resource('path/to/file.json');
});
```

RESTful Endpoint

```
mod.factory('MyResource', function($resource) {
  return $resource('path/to/api');
})
```

Parameterized

Each key value in the parameter object is first bound to url template if present and then any excess keys are appended to the url search query after the ?.

```
mod.factory('MyResource', function($resource) {
  return $resource('path/to/api/:id');
})
```

In this case the `id` property of the object would be bound to `:id` in the path.

Parameter Defaults

If the parameter value is prefixed with @ then the value for that parameter will be extracted from the corresponding property on the data object (provided when calling an action method). For example, if the defaultParam object is {someParam: '@someProp'} then the value of someParam will be data.someProp.

```
mod.factory('MyResource', function($resource) {
  return $resource('path/to/api/:id', {id: '@id'});
})
```

Custom Actions

Parameters, and Parameter Defaults, work the same for custom actions as they do for the base resource. Each `url` can have parameters with the `:` prefix and each action can have a `params` property that defines any Parameter Defaults.

For all custom actions, the `path/to/api` must be the *full* relative path to the API, just like listed in the initial `$resource` definition.

Returns Array

```
mod.factory('MyResource', function($resource) {
  return $resource('path/to/api', {}, {
      action1: {
        method: 'get|put|post|delete|jsonp',
        url: 'path/to/api',
        isArray: true
      }
    });
})
```

Returns a Single Object

```
mod.factory('MyResource', function($resource) {
  return $resource('path/to/api', {}, {
      action1: {
        method: 'get|put|post|delete|jsonp',
        url: 'path/to/api/:id'
      }
    });
})
```

Transform the Request

By default, `transformRequest` will contain one function that checks if the request data is an object and serializes to using `angular.toJson`. To prevent this behavior, set `transformRequest` to an empty array: `transformRequest: []`.

You can send in an array of functions that will be executed as different transforms. This is something you might do if you had a large list of transforms that were shared across your application.

One transform that you might share across resources is adding an `Authorization` token.

```
mod.factory('MyResource', function($resource) {
  return $resource('path/to/api', {}, {
      action1: {
        method: 'get|put|post|delete|jsonp',
        url: 'path/to/api/:id',
        transformRequest: function(data, getHeaders) {
          var headers = getHeaders();
          headers.push('Content-Type', 'application/json');
          return angular.toJson(data);
        }
      }
    });
})
```

Transform the Response

By default, `transformResponse` will contain one function that checks if the response looks like a JSON string and deserializes it using `angular.fromJson`. To prevent this behavior, set `transformResponse` to an empty array: `transformResponse: []`.

You can send in an array of functions that will be executed as different transforms. This is something you might do if you had a large list of transforms that were shared across your application.

```
mod.factory('MyResource', function($resource) {
  return $resource('path/to/api', {}, {
      action1: {
        method: 'get|put|post|delete|jsonp',
        url: 'path/to/api/:id',
        transformResponse: function(data, getHeaders) {
          var obj = angular.fromjson(data);
          // do some kind of transform on the object
          return obj;
        }
      }
    });
})
```

Caching

By setting `cache` to `true` a default `$http` cache will be used to cache this response. However, you can also send in an instance of `$cacheFactory` instead and it will use that cache factory.

```
mod.factory('MyResource', function($resource) {
  return $resource('path/to/api', {}, {
      action1: {
        method: 'get',
        url: 'path/to/api',
        cache: true
      }
    });
})
```

Timeout

```
mod.factory('MyResource', function($resource) {
  return $resource('path/to/api', {}, {
      action1: {
        method: 'get',
        url: 'path/to/api',
        timeout: 1000
      }
    });
})
```

Cancellable

Setting this to `true` does also require some **specific usage**. You can only cancel *non-instance* requests. It is safe to call `$cancelRequest` at any time because if it's already completed or canceled nothing will happen.

```
mod.factory('MyResource', function($resource) {
  return $resource('path/to/api', {}, {
      action1: {
        method: 'get',
        url: 'path/to/api',
        cancellable: true
      }
    });
})
```

Interceptor

```
mod.factory('MyResource', function($resource) {
  return $resource('path/to/api', {}, {
      action1: {
        method: 'get',
        url: 'path/to/api',
        interceptor: {
          response: function(obj) {
            // do something with obj (it's already deserialized)
            return obj;
          },
          responseError: function(obj) {
            // do something on error
            if (canRecover(rejection)) {
              return responseOrNewPromise
            }
            return $q.reject(rejection);
          }
        }
      }
    });
})
```

Usage

Creating a New Instance

This is commonly used on add forms.

```
var obj = new MyResource();
```

Non-instance Usage

GET Requests

Here you can leverage any action that is a `GET` request. This includes the built-in `query` and `get` along with custom actions.

```
var params = {someId: 1};
function success(value, responseHeaders) {}
function error(err) {}

var response = MyResource.query();

var response = MyResource.query(params);

var response = MyResource.query(success, error);

var response = MyResource.query(params, success, error);
```

Non-GET Requests

While you *could* call these without a `success` and `error` callback, **it would not be recommended.** Generally speaking with non-GET operations you need to fully understand the outcome and perform operations based on that.

Furthermore, these methods can set a `response` object just like the `GET` methods can. If that's useful for you, then just keep that in mind.

```
var params = {someId: 1};
var postData = {key1: "val1", key2: "val2"};
function success(value, responseHeaders) {}
function error(err) {}

MyResource.save(postData, success, error);

MyResource.save(params, postData, success, error)
```

Instance Usage

These imply that you have an object instance that came from a Resource query. This is common on add or edit forms. In these cases you *do not* need to send the `postData` because the object itself is implied as being the `postData`.

```
function success(value, responseHeaders) {}
function error(err) {}

obj.$save(success, error)
```