FALCOR

A JavaScript library for efficient data fetching

Table of Content

Introduction

What Is Falcor?

Falcor is a Javascript library for data fetching. When using Falcor, you can model all backend data into a single JSON object on Node server, which can be accessed from the client side using Javascript operations. The backend data reflects the API used by the client.

Falcor consists of two parts, client side and server side. Server side will route requests coming from client side to the corresponding action and return the requested data to client side. Client side not only makes the request, but it'll also try optimize the request to server side by either caching or batching the request sent.

Falcor is not a replacement for your backend stack (application server, MVC framework, nor database) nor a replacement for your frontend technology (AngularJs, ReactJs), but it can be used along with these existing technology instead.

Falcor has three main characteristics, they are:

1. One model everywhere
   All backend data can be modelled into one big JSON resource on the application server. Subsets of this resource can be requested by client on demand.
2. The data is the API
   Client request data from the server the same way as how it would be from an in-memory JSON object.
3. Bind to the cloud
   Required data will be requested to the server if needed, otherwise, Falcor will just get the cached value on the client.

Getting Started

We'll create a really simple application that will retrieve data from our Falcor server using the Falcor model from the browser. Remember the big single JSON object I've mentioned at the start? In this example, the object is not that big. But any kind of request, will be requested through this single JSON.

To get started, you must have NodeJs ready first. If you've installed NodeJs, then run the commands below:

```
mkdir falcor
cd falcor
npm init

npm install express falcor-express falcor-router --save
```

The commands above will create our initial project and download the required dependencies. After the command completed successfully, we'll create our index.js file. Our index.js file is used to route all request sent to the server, including Falcor data request, and static file request. The content of index.js is as follows:

```
// index.js
var falcorExpress = require('falcor-express');
var Router = require('falcor-router');
var express = require('express');

var app = express();

app.use('/model.json', falcorExpress.dataSourceRoute(function (req, res) {
  return new Router([
    {
      route: "greeting",
      get: function() {
        return {path:["greeting"], value: "Hello World"};
      }
    }
  ]);
}));

// serve static files from current directory
app.use(express.static(__dirname + '/'));

var server = app.listen(3000);
```

The code above will create an Express app which listens to port 3000, and will serve any static file and a route called model.json

After finishing the index.js, we'll create another new file, which is index.html. This file will request the greeting data using Falcor Model.

```
<!-- index.html -->
<html>
  <head>
    <script src="https://netflix.github.io/falcor/build/falcor.browser.js"></script>
    <script>
      var dataSource = new falcor.HttpDataSource('/model.json');
      var model = new falcor.Model({ source: dataSource });

      model
        .get("greeting")
        .then(response => document.write(response.json.greeting));
    </script>
  </head>
  <body>
  </body>
</html>
```

The code above will create a request to the server and then display it. The request is sent to the model.json route with the path of greeting, where the value of this path is “Hello World”. This request will be sent to the server, and when server returns the data, Falcor will cache the result on the browser. This means that any subsequent call to this path, will not make any additional data request to the server, but instead it will use the cached data on the browser.

Path and PathSets

Path

Path is used to refer a location in a JSON object. There are two ways to access data using Falcor, array of keys and path syntax string. In the getting started example, we used the path syntax string, which is “greeting”.

```
var log = console.log.bind(console);

var model = new falcor.Model({
  cache: {
    foods: [
      {
        name: "Sushi",
        price: 10
      }
    ]
  }
});

model
  .get('foods[0].name', 'foods[0].price')
  .then(log);
```

In the example above, we create a model which have data about foods. The path in the example above is `foods[0].name`. This path use the string syntax. If we want to use the array path, then it will be `['foods', 0, 'name']`. Keep in mind that if you use path syntax string, Falcor will parse the string into array of keys. Which means using path array is more efficient than using path string.

Values that are not string will be converted to string, using Javascript’s toString algorithm.

PathSets

PathSets is used as a shorthand for multiple paths. Similar to Path, PathSets also can be written in two ways, string, and array. The example above can be rewritten using PathSets as below.

```
var log = console.log.bind(console);

var model = new falcor.Model({
  cache: {
    foods: [
      {
        name: "Sushi",
        price: 10
      }
    ]
  }
});

model
  // .get('foods[0]["name", "price"]')
  .get(['foods', 0, ['name', 'price']])
  .then(log);
```

Other than using multiple paths, you can also use multiple PathSets.

```
var log = console.log.bind(console);

var model = new falcor.Model({
  cache: {
    foods: [
      {
        name: {
          first: "Sushi",
          last: "Meal"
        },
        price: {
          total: 10,
          base: 8
        }
      }
    ]
  }
});

model
  .get(
    ['foods', 0, 'name', ['first', 'last']],
    ['foods', 0, 'price', ['total', 'base']])
  .then(log);
```

Multiple PathSets is used just like how you would with multiple Path. The result of multiple PathSets will still be one JSON, as it is combined by Falcor.

JSON Graph

JSON graph is a convention for modelling graph information as JSON object. JSON naturally can't have any references, all data will be duplicated in place of reference. This data duplication means bigger memory needed to store the data.

Other than the memory size, there's also another problem that it can cause, stale data. Stale data occur when data stored in a cache is not invalidated after an update, which may display an inconsistent data to the user.

To avoid this problem, Falcor remove the duplicated data, so that all the referrer will refer to the same object. If there's an update, we only need to update that one object that is referred by others.

New Type

There are three additional types added to JSON graph:

1. Reference
2. Atom
3. Error

These new types will be treated like any other primitive type. To denote a value is of these type, there will be two special keys that will be used, that is `$type` and `value`. $type is used to tell an object is of the additional type, and value is the actual value.

Reference

Reference is an additional type which has a $type value of `ref`. Reference is used to point to another location by using Path. Reference makes it possible to model a graph in JSON. Below is an example of reference typed value.

```
var model = new falcor.Model({
  cache: {
    foods: [
    {
      name: {
        first: "Sushi",
        last: "Meal"
      },
      price: {
        total: 10,
        base: 8
      },
      category: {
        $type: 'ref',
        value: ['categoryById', 1]
      }
    }
    ],
```

```
    categoryById: {
      1: {
        name: 'japanese'
      }
    }
  }
});
```

The food category is a referenced typed value. This references the Japanese category. To retrieve the category name of the first food, we can navigate it as we usually would with other type by using Path or PathSets.

```
model.get(
    ['foods', 0, 'category', 'name']
  ).then(log);
```

Reference works just like a link to another location according to the path sets in the value key. When the current evaluated path meet a reference, it will restart again at the root of the graph and prepend the referenced path to the front of current path.

Atom

Atom is an additional type which has a $type value of `atom`. Atom is treated like value type, it is retrieved and set in its entirety. Atom cannot be mutated using abstract JSON graph operation. It must be replaced entirely using the abstract set operation.

Atom also enables metadata attached to a primitive value. The example below won't work without the Atom type.

```
var totalVisitor = 42;
totalVisitor['$expires'] = 100;
```

This won't work because when converting totalVisitor to JSON, the $expires is not serialized. But we can achieve this instead by making the total visitor an object like below.

```
var totalVisitor = {
  $expires: 100,
  value: 42
}
```

This will work, as the $expires can be serialized. But it's a little awkward to access the value of totalVisitor now. We must navigate to the value index first. By using atom type, we can ensure that $expires get serialized, and we can access the value directly.

```
var model = new falcor.Model({
  cache: {
    totalVisitor: {
      $type: 'atom',
      $expires: 100,
      value: 42
    }
  }
});

model.get('totalVisitor'); // We'll get 42!
```

The other use case for atom type is when we want to take an object or array as a value that we can retrieve all at a time. For example, we may want to retrieve all foods at once.

```
var log = console.log.bind(console);

var model = new falcor.Model({
  cache: {
    foods: {
      $type: 'atom',
      value: [
        {
          name: {
            first: "Sushi",
            last: "Meal"
          },
          price: {
            total: 10,
            base: 8
          }
        }
      ]
    }
  }
});

model.get('foods')
  .then(log);
```

Error

Error is an additional type which has a $type value of `error`. Error may be used when executing an abstract graph operation if the operation failed. An error object will be returned instead of the supposed value in case of error. Below is an example of an error type object.

```
{
  $type: 'error',
  value: 'Request timeout'
}
```

Error prevents failing to retrieve other values when there's a failure when retrieving a value. Failure retrieving a value can be caused by several things, like timeout from the server when under a heavy load.

The Abstract JSON Graph Operations

There are three abstract JSON graph operation:

1. Get
2. Set
3. Call

The three abstract JSON graph operations need to be executed by `Router` object or `Model` object. Each operation returns a JSON object with a `json` key which contains a JSON graph subset. Each operation will update the cache on the client side. The call operation may include an `invalidated` key to delete the client cache, making sure there's no stale data left on the client.

The Abstract get Operation

This operation retrieves all primitive values encountered on the given paths. The retrieved value is presented in a JSON graph object. Get operation should be idempotent, which means it should not change any value in the JSON graph.

Below is an example of using get operation on model.

```
var log = console.log.bind(console);

var model = new falcor.Model({
  cache: {
    foods: [
      {
        name: {
          first: "Sushi",
          last: "Meal"
        },
        price: {
          total: 10,
```

```
          base: 8
        }
      }
    ]
  },
});

model.get('foods[0].name.first').then(log);
```

The Abstract set Operation

Set operation is used to set data to a path on the graph. You may specify multiple paths at once. And the returned result of this operation is subset of the JSON graph according to the given paths. Set operation does not only receive paths, but it should also be accompanied by value.

```
var log = console.log.bind(console);

var model = new falcor.Model({
  cache: {
    greeting: 'Hello world'
  }
});

model.get('greeting')
  .then(log);

model.setValue('greeting', 'Hola').then(log);
```

Code above will change the greeting value to Hola. There are several ways to invoke the set operation. The first on is using the setValue to set the value on JSON graph. The example above uses this method. Other methods are using pathValue or JSONEnvelope. PathValue is a Javascript object that have two keys, path and value, and JSONEnvelope is also a Javascript object that have a key named json and the value of this key is a JSON graph.

Below is the example of using PathValue with set operation.

```
var log = console.log.bind(console);

var model = new falcor.Model({
  cache: {
    greeting: 'Hello world'
  }
});

/*
{
  path: 'greeting',
```

```
  value: 'Hola'
}
*/
var pathValue = falcor.pathValue('greeting', 'Hola');
model.set(pathValue).then(log);
```

Notice that we use falcor.pathValue to create a PathValue object, but it is not a mandatory. You can just instantiate a new object with path and value as key, and the result will still be the same.

And below this is another example of set operation, but this time using JSONEnvelope.

```
var model = new falcor.Model({
  cache: {
    greeting: 'Hello world'
  }
});

model.set({
  json: {
    greeting: 'Hola'
  }
}).then(log);
```

The Abstract call Operation

JSON Graph object may contain function that does certain activities. To invoke those function, you may use the call operation to the path containing the function. The call method on model has the prototype of:

```
call(callPath:Path, arguments:any[], refPaths?: PathSet[], thisPaths?:PathSet[]):
ModelResponse
```

Below are the description of the call arguments:

1. callPath argument is the path where the function is located. Unlike other operation, you cannot call multiple function at once.
2. arguments is an array of parameter which will be passed to the function . This argument is optional.
3. refPaths arguments is used to get certain reference fields of the function's return value. This prevent another roundtrip to retrieve the data. Keep in mind that it only traverse the reference type.

4. thisPaths arguments work just like refPaths, but instead of traversing the JSON graph object, thisPaths traverses the “this” object of the function. The result is then added to the response of the call operation. This argument is optional.

The server can return several types of value compared to get operation. The return value can consists the usual pathValue, but it also may include invalidate keys. The invalidate value marks the path which is affected and should be removed from the client cache, so there won’t be any stale data.

```
// Server side
var greeting = 'Hello World';

app.use('/model.json', falcorExpress.dataSourceRoute(function (req, res) {

  return new Router([
    {
      route: "greeting",
      get: function() {
        return { path:["greeting"], value: greeting };
      },
      call: function() {
        greeting += '!';

        return { path: ["greeting"], invalidated: true};
      }
    }
  ]);
}));
```

```
// Client side
var log = console.log.bind(console);
var source = new falcor.HttpDataSource('/model.json');

var model = new falcor.Model({
  source: source
});

model.get('greeting').then(log);
model.call('greeting').then(() => model.get('greeting').then(log));
```

Every time there’s a call to greeting, then the greeting value will be appended with an exclamation mark. Because of the greeting path is invalidated, it will remove that path from client cache, resulting in another request to the server if another get operation is invoked. Notice that if you did not specify any return value (returning empty object), all client cache will be invalidated.

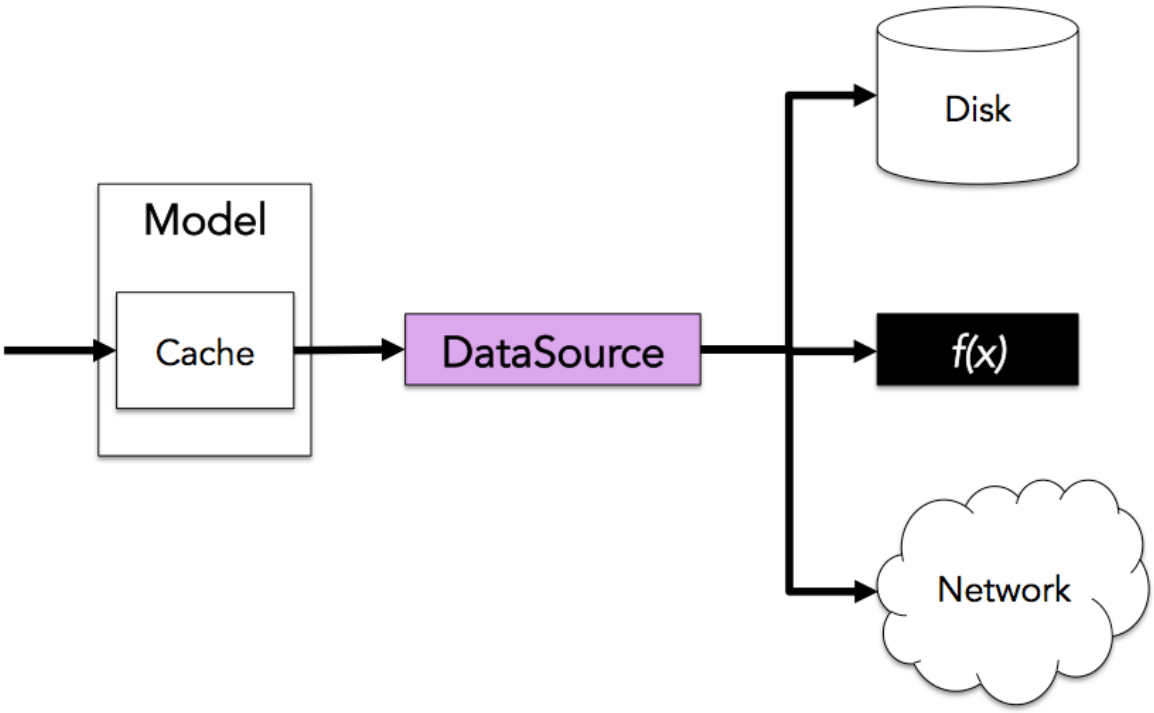
To prevent another server round trip to get the latest data, we can return the affected value to the client. This is the same as the get operation.

```
var greeting = 'Hello World';

app.use('/model.json', falcorExpress.dataSourceRoute(function (req, res) {

  return new Router([
    {
      route: "greeting",
      get: function() {
        return { path:["greeting"], value: greeting };
      },
      call: function() {
        greeting += '!';

        return { path: ["greeting"], value: greeting };
      }
    }
  ]);
}));
```

By returning the latest value, we can remove the server round trip to get the latest data, as the affected data is returned directly.

Data Source

Data source is where the model can retrieve data from. It can be a function, disk, or network. We can execute three operations to the data source, get, set, and call operation. These operation works exactly the same as we've explained earlier. Different from model, data source does not support path string syntax, it only supports path array syntax.



Falcor comes with three implemented data source:

1. HttpDataSource
2. Router (server only)
3. ModelDataSource

The way these three communicate is usually done by client request data through ModelDataSource, and ModelDataSource forward the request to HttpDataSource, which will send the request through HTTP protocol to the Router.

Router

Router is an implementation of data source, which communicate with other data source like how we explained before. Router works by matching the requested path against virtual JSON graph. This JSON graph is initialized when there's a request to the specified path, after the request is finished, this JSON graph will be released from the memory, hence the name virtual.

Keep in mind that Falcor Router may split one request to several different routes. This means that route handler cannot generate one optimized query to get all the required data.

There are three primary differences between falcor router and a REST router.

1. Falcor Router match JSON path, not URL path
   Each route in Falcor Router is associated using JSON path and an operation, compared to REST which match router by using URL path.

```
app.use('/model.json', falcorExpress.dataSourceRoute(function (req, res) {

  return new Router([
    {
      route: "greeting.en",
      get: function() {
        return { path:['greeting', 'en'], value: "Hello world" };
      }
    },
    {
      route: "greeting.id",
      get: function() {
        return { path: ['greeting','id'], value: "Halo dunia" };
      }
    }
  ]);
}));
```

2. A single Falcor route can match multiple path

```
// Server side
var greetings = {
  en: 'Hello World',
  id: 'Halo dunia'
};

app.use('/model.json', falcorExpress.dataSourceRoute(function (req, res) {

  return new Router([
    {
      route: "greeting.['en', 'id']",
      get: function(pathSet) {
        return pathSet[1].map(function(key) {
          return {
            path:['greeting', key],
            value: greetings[key]
          }
        });
      }
    }
  ]);
}));
```

```
// Client side
var log = console.log.bind(console);
var source = new falcor.HttpDataSource('/model.json');

var model = new falcor.Model({
  source: source
```

```
});

model.get('greeting.en').then(log);
model.get('greeting.id').then(log);
model.get('greeting.["en", "id"]').then(log);
```

3. Falcor Router can retrieve related resource in one trip
   We can traverse the related (referenced) data in one go.

```
// Server side
var categories = [
  {
    id: 11,
    name: 'foods'
  },
  {
    id: 12,
    name: 'drinks'
  }
];

app.use('/model.json', falcorExpress.dataSourceRoute(function (req, res) {

  return new Router([
    {
      route: "categories[{integers:indexes}]",
      get: function(pathSet) {
        return pathSet.indexes.map(function(index) {
          var category = categories[ index ];

          if (category == null) {
            return {
              path: [ 'categories', index ],
              value: category
            };
          } else {
            return {
              path: [ 'categories', index ],
              value: {
                $type: 'ref',
                value: [ 'categoryById', category.id ]
              }
            };
          }
        });
      }
    },
    {
      route: "categoryById[{integers:idList}].['id', 'name']",
      get: function (pathSet) {
        function findCategoryById(id) {
```

```
          return categories.filter(function (c) {
            return c.id == id;
          })[ 0 ];
        };

        var pathValues = [];

        pathSet.idList.forEach(function(id) {
          var category = findCategoryById(id);

          if (category == null) {
            pathValues.push({
              path: [ 'categoryById', id ],
              value: category
            });
          } else {
            pathSet[2].forEach(function (key) {
              pathValues.push({
                path: [ 'categoryById', id, key ],
                value: category[ key ]
              });
            });
          }
        });

        return pathValues;
      }
    }
  ]);
}));
```

```
//Client side
var log = console.log.bind(console);
var source = new falcor.HttpDataSource('/model.json');

var model = new falcor.Model({
  source: source
});

model.get('categories[0..1].["id", "name"]').then(log);
```

When the client requests the path `categories[0..1].["id", "name"]`, Falcor will route this request, to two separate handler. The first one will get all categories within the range of zero to one, and then the last is to get the id and name of the category in that list. Because of this, the response of this request will look like there's two separate request too.

```
{
  "jsonGraph": {
```

```
    "categories": {
      "0": {
        "$type": "ref",
        "value": [
          "categoryById",
          1
        ]
      },
      // …
    },
    "categoryById": {
      "1": {
        "id": 1,
        "name": "foods"
      },
      // …
    }
  }
}
```

Although the root path that we request is categories, categoryById is added to our response, because categories have references to it.

Route Object

Route Object consists of path that is used to match a request and three optional operations handler associated to the path, get, set, and call operation.

Each route handler takes a pathSet as an argument. This pathSet is an array of path that is matched to the current route. The path is split by each separator. For example the path of categories[0..2].name is split into three parts, categories, [0, 1, 2], and name.

Route handler should only return pathValue or array of pathValue, with the exception of call operation which may return an object with a path and an invalidated key. After the request is fully handled, then all the result will be combined into a single JSON graph. This JSON graph will be return to the requester (client) which the client will combine it to its own JSON graph.

Route Pattern Matching

Route pattern just like the path string syntax, but in addition to be able to use some special keys, such as

1. Integers
   Integers will match any integers in paths, including ranges. The matched keys will be stored in array of integers.

```
app.use('/model.json', falcorExpress.dataSourceRoute(function (req, res) {

  return new Router([
    {
      route: "categories[{integers:indexes}]",
      get: function(pathSet) {
        console.log(pathSet.indexes);
        // ...
      }
    }
  ])
}
```

If the requested path for the example above is `categories[1,2,4..6]` then the `pathset.indexes` values is `[1,2,4,5,6]`. Notice that the range from 4 to 6 is expanded into 4, 5, and 6.

2. Ranges

   This is the opposite of integers, every integer will be converted into ranges.

```
app.use('/model.json', falcorExpress.dataSourceRoute(function (req, res) {

  return new Router([
    {
      route: "categories[{integers:indexes}]",
      get: function(pathSet) {
        console.log(pathSet.indexes);
        // ...
      }
    }
  ])
}
```

If the request path for the example above is `categories[1..3, 5, 7]`, then the value of `pathSet.indexes` is `[{ from: 1, to: 3 }, { from: 5, to: 5 }, { from: 7, to: 7 }]`.

3. Keys

   This pattern will match any valid key (string, number, boolean).

```
app.use('/model.json', falcorExpress.dataSourceRoute(function (req, res) {

  return new Router([
    {
      route: "categoryById[{integers:idList}].[{keys:propList}]",
      get: function (pathSet) {
        console.log (pathSet.idList, pathSet.propList);
        // ...
      }
    }
  ]);
}));
```

If the requested path for the example above is `categoryById[1]["id", "name"]`, then the value of `pathSet.propList` is `["id", "name"]`.

Keep in mind that even if there's only one matched pattern, the result will still be an array (array of one element).

Model

Model is one of the pre-implemented data source class from Falcor. Even though we can retrieve data directly from an HttpDataSource, using model is preferable because of the following reasons:

1. Model returns data hierarchically, compared to in JSON graph format.
   With the server side code using the code shown before at Falcor Router. If we use the HttpDataSource instead of Model, then the result we get will be of JSON graph.

```
// Client side
var log = console.log.bind(console);
var source = new falcor.HttpDataSource('/model.json');
source.get([
  ['categories', {from: 0, to: 1}, ['id', 'name']]
]).subscribe(jsonGraph => log(jsonGraph));
```

```
// Response
{
  "jsonGraph": {
    "categories": {
      "0": {
        "$type": "ref",
        "value": [
          "categoryById",
          1
        ]
      },
      "1": {
        "$type": "ref",
        "value": [
          "categoryById",
          2
        ]
      }
    },
    "categoryById": {
      "1": {
        "id": 1,
        "name": "foods"
      },
      "2": {
        "id": 2,
        "name": "drinks"
```

```
      }
    }
  }
}
```

Note that the first parameter for get operation when using data source should be array of pathSet.

2. Model cache the result
   Model can cache the result of the requested operation. Making the next request does not need to go through network.
3. Model can optimize the path by using previously cached JSON graph references.
4. Models can batch requests before sending to the server, reducing latency.

Keep in mind that models cannot request an object or array directly. You can only request primitive values, and then these values will be merged into one. Making it indirectly an array or object. The primitive values also includes the reference, atom, and error.

There're two additional operation other than the abstract operation that model has. The getValue and setValue operation. These operations can get a direct value of a path, in the contrary of getting JSON graph when using get operation. The same goes to setValue.

Configuring Model

Model can be instantiated with an object with certain keys. Following are the valid keys to configure a Model:

1. cache
   This key is used to initialize the cache in model. The value should be in the form of JSON graph.
2. maxSize
   This determine the max size available for the cache.
3. collectRatio
   In the case of clearing cache because the cache size is over maxSize, then the cache will be cleared so that it will have a maximum cache size collectRatio * maxSize.
4. source
   This is the data source for the model
5. onChange
   OnChange is a function called whenever the Model's cache is changed. The callback has no parameter.
6. comparator
   Comparator is a function called when there's an update to the initial value in the cache. This function receives two parameter, the old and new value, and should return a boolean indicating whether the new and old value are equal.
7. errorSelector
   Error selector is used to transform the value in the case of receiving object with the type $error. For example you may want to try to get the value from the server again after two minutes.

```
var model = new falcor.Model({
  source: new falcor.HttpDataSource('/model.json'),
  errorSelector: function(error){
    error.$expires = -1000 * 60 * 2;
  }
});
```

Sentinel Metadata

Values can have metadata to control how model handles them. Metadata is set as a key that starts with $ sign to a JSON object. There are four metadata that can be used:

1. $type
   Determines the type of the value, can be ref, atom, or error.
2. $expires
   Determines how long should the data be cached. There are four kinds of value:
   a. Expire immediately $expires = 0
   b. Never expire $expires = 1
   c. Expire at an absolute time $expires > 0
   d. Expire at an relative time $expires < 0
3. $timestamp
   This helps to remove the race condition caused by multiple concurrent requests. If there are several different value given for a same path, then the real value is the value with the highest timestamp.
4. $size
   This determine the size of the value in the specified path. Model do not approximate the size of an object, so to help model count the size of an object, the size metadata can be used.

Because the metadata can only be set to a JSON object, when you want to set metadata to Javascript's primitive type (strings, integer), then the value should be wrap into and object with type atom.

```
// Don't this
var number = 1;
number.$expires = 0;

// Do this
var number = {
  $type: 'atom',
  $expires: 0,
  value: 1
};
```

Batching Requests

Each request is sent to the server separately by default. To batch multiple request before sending them to the server, you can use the batch method.

```
var log = console.log.bind(console);
var source = new falcor.HttpDataSource('/model.json');

var model = new falcor.Model({
  source: source
});

var batch = model.batch();
batch.get('categories[0].name').then(log);
batch.get('categoryById[1].name').then(log);
```

References

http://netflix.github.io/falcor

https://auth0.com/blog/2015/08/28/getting-started-with-falcor/