305CDE Week 11

Using `map()` and `reduce()` in JavaScript and CouchDB

Colin Stephen

December 2014

Overview

- Types of programming
  - Imperative `for` loops over arrays
  - A bit about functional programming
- Important JS array methods
  - `Array.prototype.map()`
  - `Array.prototype.reduce()`
  - `Array.prototype.filter()`
- CouchDB
  - Selecting (map)
  - Grouping (reduce)
  - Searching (filter)

Imperative vs Declarative vs Hybrid Languages

Imperative Languages

- Focus on *what steps the computer should take* rather than what the computer will *do*.
  - C++, C, Java

Declarative

- Focus on what the computer should *do* rather than on how it should do it.
  - Logic (Prolog)
  - Functional (Haskell)

Hybrid

- Mix imperative and declarative approaches.
- Python, *JavaScript*

Imperative Approach

Some Typical Code

Lots of `for` and `while` loops: i.e. *how* to do the computation.

Example to retrieve a list of incomplete tasks for a user and sort by days remaining. First get the tasks.

```
var getIncompleteTasksFor = function(who) {
  return fetchMonthlyTasks()  // returns a promise
    .then(function(data) {
      return data.tasks;
    })
```

Some Typical Code (continued 1)

Then find the ones for this user.

```
.then(function(tasks) {
  var results = [];
  for (var i = 0, len = tasks.length; i < len; i++) {
      if (tasks[i].member == who) {
          results.push(tasks[i]);
      }
  }
  return results;
})
```

Some Typical Code (continued 2)

Then find the ones that are not completed.

```
.then(function(tasks) {
  var results = [];
  for (var i = 0, len = tasks.length; i < len; i++) {
      if (!tasks[i].complete) {
          results.push(tasks[i]);
      }
  }
  return results;
})
```

Some Typical Code (continued 3)

Then summarise with their title and a (calculated) number of remaining days.

```
.then(function(tasks) {
  var results = [], task;
  for (var i = 0, len = tasks.length; i < len; i++) {
      task = tasks[i];
      var today = new Date().getDate();
      results.push({
          title: task.title,
          remain: task.due - today
      })
  }
  return results;
})
```

Some Typical Code (continued 4)

Finally sort based on days remaining to complete.

```
  .then(function(tasks) {
    tasks.sort(function(first, second) {
        return first.remaining - second.remaining;
    });
    return tasks;
  });
```

The Functional Approach

Previous Code Refactored

```
var getIncompleteTasksFor = function(who) {
  return fetchMonthlyTasks()  // returns a promise
    .then(function(data) {
      return data.tasks
      .filter(function(task){return (task.member==who)})
      .filter(function(task){return !(task.complete)})
      .map(function(task){
        var remaining = task.due-(new Date().getDate());
        return {title: task.title, remain: remaining}
      })
      .sort(function(first,second){
        return first.remain - second.remain;
      });
    })
```

Benefits of Functional Approach

- Code describes *what to do* not how to do it
- Much shorter
  - Less to go wrong (e.g. changing a variable value)
  - Easier to read
  - Quicker to debug
  - Easier to unit test
- Can be parallelised easily
  - e.g. if task list contains 10 Billion tasks!

How to Achieve These Benefits in JS

Available in "everyday" JS:

- first class functions
- lambdas / anonymous functions with closures
- compact (terse) functions
- function composition
- **functional array methods** – rest of this lecture

Available with some care in JS:

- mostly stateless processing
- currying: $f(x, y) \rightarrow f(x)(y)$
- side-effect-free function calls

Array Methods

A few key array methods in JS offer a lot of "functional programming benefits". Look them up on MDN: map, reduce, and filter. These are present in many hybrid languages.

`Array.prototype.map()`

- Creates a new array with the results of calling a provided function on every element in this array.

`Array.prototype.reduce()`

- Apply a function against an accumulator and each value of the array (from left-to-right) so as to reduce it to a single value.

`Array.prototype.filter()`

- Creates a new array with all of the elements of this array for which the provided predicate function returns `true`.

Map

- Takes a unary (1-argument) callback.
- Callback can return any JS object.

```
var nums = [2,3,4,5,6,7];

var square = function(num) {return num*num};
var Counter = function(start) {this.value=start}

nums.map(square);  // returns [4,9,16,25,36,49]
nums.map(toString);  // returns ["2","3",...,"7"]
nums.map(function(num){ return new Counter(num); });
  // returns an array of Counter objects
  // with different start values
```

Filter

- Takes a unary (1-argument) callback.
- Callback must return `true` or `false`.
- Such a callback is called a *predicate function*.

```
var nums = [2,3,4,5,6,7];

var even = function(num) {return (num % 2 == 0)};
var morethan = function(min) {
  return function(num) { return (num > min); };
};

nums.filter(even);  // returns [2,4,6]
nums.filter(morethan(4));  // returns [5,6,7]
```

Reduce

- Takes a binary (2-argument) callback and an *optional initial value*.
  - First callback argument represents the *intermediate result* of processing so far
  - Second callback argument represents the next array item to be processed

```
var nums = [2,3,4,5,6,7];
var sum = function(a,b) {return a+b};
nums.reduce(sum,0);  // returns 27
nums.reduce(sum,10);  // returns 37

var arrs = [[1,3], [5,7], [2,4]];
var concat = function(a,b) {return a.concat(b)};
arrs.reduce(concat,[]); // returns [1,3,5,7,2,4]
```

Chaining

The array operations can be chained, for convenience.

```
var nums = [2,3,4,5,6,7];

nums
   .filter(even)
   .map(square)
   .reduce(sum)

   // returns sum of squares of evens
   // i.e. 2*2 + 4*4 + 6*6 = 56
   // without using for loops
```

NB: all of this works on arrays of *any JS object*, for example **promises**!

Application To CouchDB Views

Key Observation

- You can just think of a CouchDB database like an *array of documents*!
  - (Actually it is a key/value store, but the same ideas apply).
- Which means you can use map, reduce, and (indirectly) filter across the DB documents.
- These functions are called *views* by CouchDB.
  - Map views `emit` key/value pairs rather than `return` arbitrary objects.
    - Otherwise they are the same thing as described above.
  - Reduce views have a `rereduce` flag to determine when to stop the reduction.
    - Otherwise they are the same thing as described above.

CouchDB Views

Primary tool used for querying and reporting on CouchDB documents.

Permanent

- stored inside special documents called design documents
- can be accessed via an HTTP GET request to /{dbname}/{docid}/{viewname}
  - {docid} has the prefix _design/
  - {viewname} has the prefix _view/

Temporary

- executed on demand
- HTTP POST request to /{dbname}/_temp_view
  - body of the request contains the code of the view function
  - Content-Type header is set to application/json.

Remember: views are just special JS functions corresponding to maps and reductions.

Map View Example

It is just a `map()` callback as we saw above!

```
function(doc) {
  if (doc.Type == "customer") {
    emit(doc._id,
      {LastName: doc.LastName,
       FirstName: doc.FirstName});
  }
}
```

For each document in the database that has a `Type` field with the value `customer`, a row is created in the view. The value column of the view contains the LastName, and FirstName. The key for each documents is just the _id.

Using a Different Key

If you wish to sort or filter on a field other than `_id`, just define your map function to emit the appropriate key:

```
function(doc) {
  if (doc.Type == "customer") {
    emit(doc.LastName,
      {FirstName: doc.FirstName,
       Address: doc.Address});
  }
}
```

Reduce View Example

- If a view has a reduce function, it is used to produce **aggregate results** for that view.
- Reduce functions are associated with maps.
- It is essentially a `reduce()` callback as we saw above, with a few additional rules applied.
  - A reduce function is passed a set of intermediate values and combines them to a single value.

```
function (key, values, rereduce) {
    return sum(values);
}
```

- the `rereduce` parameter is a boolean which can be used to stop the reduction at an "intermediate" stage
- **Constraint**: reduce functions must accept, as input, results emitted by its corresponding `map` function (in the same view) as well as results returned by the reduce function itself.

Grouping

- Calling a reduce view over HTTP defaults to reducing to a single value
- Passing `group=true`, you get a separate reduce value for each unique key emitted by the map.
- For example given a DB of customer purchases:
  - use the map to emit a `(customer_ID, purchase_price)` key/value pair *for each purchase*
  - you may have multiple records in the view with the *same key*
  - do a reduce that returns the sum of the values (purchase prices) with `group=true`
  - this will return `(customer_ID, total_purchases)` key/value pairs, where the keys are now unique

Reference

- See `http://wiki.apache.org/couchdb/Introduction_to_CouchDB_views`
- CouchDB map and reduce are only slightly different from regular `map()` and `reduce()` in JS
- If you understand the latter, then you can understand the former.