Getting Started with IBM Bluemix

Hands-On Workshop

Exercise 6a: Adding a Service to an Application

Adding a service to an application

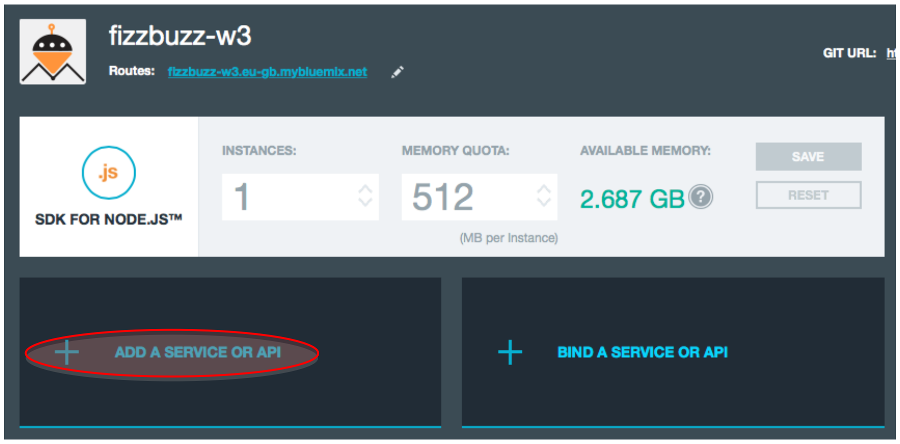
In this exercise, you'll extend the FizzBuzz implementation to cache results to a NoSQL database.

When a range is requested, the database will be searched to see whether this request was previously saved. If a request was saved, the FizzBuzz result will be returned from the database. If the range was not previously saved, it will be calculated and saved to the database for future requests. Then, the results will be sent to the requesting user.

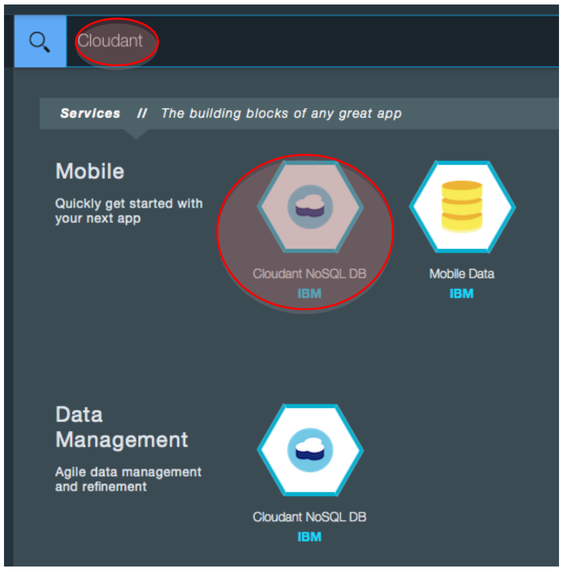
For this exercise, you'll use the Cloudant database service from Bluemix. You must add an instance to the application that is running on Bluemix.

1. Create the Cloudant service for the application:
   a. In the Bluemix web UI, select your application from the dashboard.

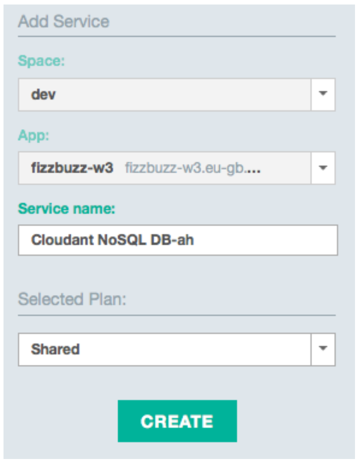
   b. From the Overview page of your Bluemix application, click **ADD A SERVICE OR API**.


fizzbuzz-w3
Routes: fizzbuzz-w3.eu-gb.mybluemix.net
GIT URL:
.js
SDK FOR NODE.JS™
INSTANCES: 1
MEMORY QUOTA: 512 (MB per Instance)
AVAILABLE MEMORY: 2.687 GB
SAVE
RESET
ADD A SERVICE OR API
BIND A SERVICE OR API

   c. Enter `Cloudant` into the search bar and click the **Cloudant NoSQL DB** under Data Management.


Cloudant
Services // The building blocks of any great app
Mobile
Quickly get started with your next app
Cloudant NoSQL DB
IBM
Mobile Data
IBM
Data Management
Agile data management and refinement
Cloudant NoSQL DB
IBM

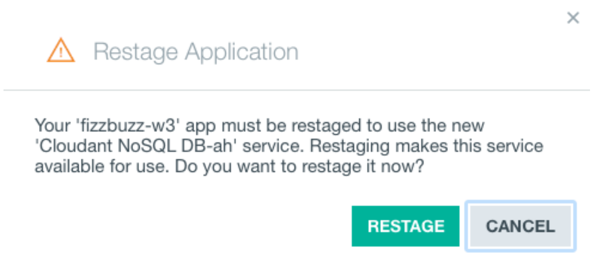
d. Use the default values. However, you can provide your own service name so that you can associate the service name with your application.



Space: dev
App: fizzbuzz-w3 fizzbuzz-w3.eu-gb....
Service name: Cloudant NoSQL DB-ah
Selected Plan: Shared

e. Click **CREATE** to create and bind the database to your application.

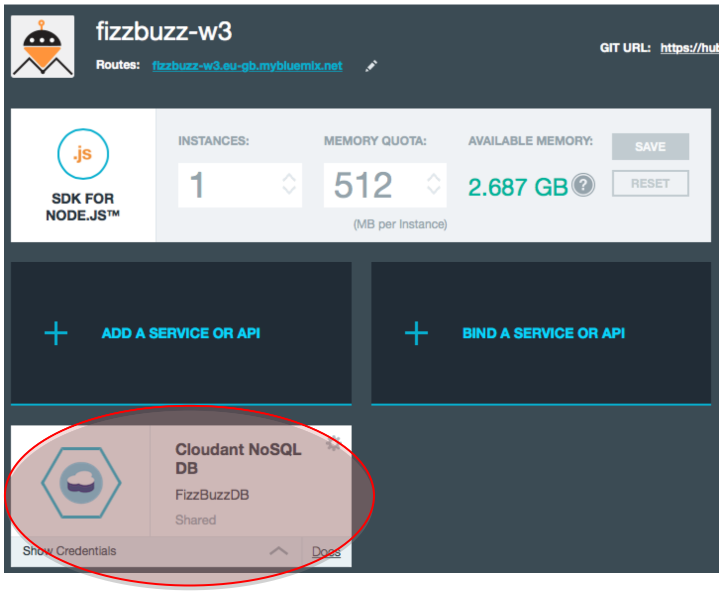
f. Restage the application to make it available to the application by clicking **RESTAGE**.



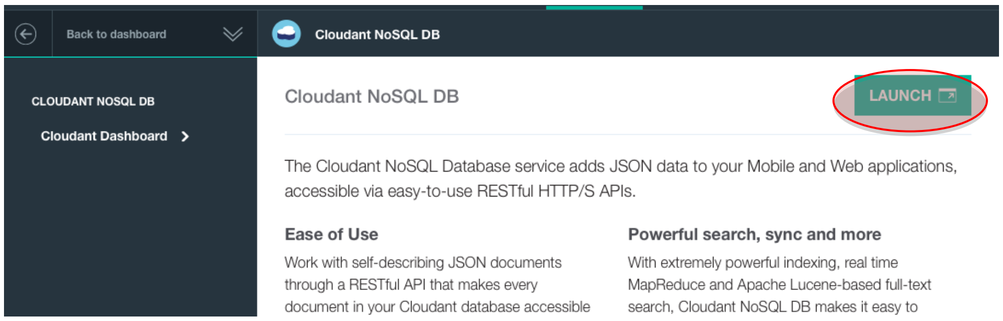
Your 'fizzbuzz-w3' app must be restaged to use the new 'Cloudant NoSQL DB-ah' service. Restaging makes this service available for use. Do you want to restage it now?

Wait until the application finishes restaging and is running again.

2. Configure the database for the FizzBuzz application:
   a. Launch the Cloudant console by clicking the Cloudant service from the application Overview page.



b. Click **LAUNCH** from top right of Cloudant Dashboard.



Cloudant NoSQL DB

The Cloudant NoSQL Database service adds JSON data to your Mobile and Web applications, accessible via easy-to-use RESTful HTTP/S APIs.

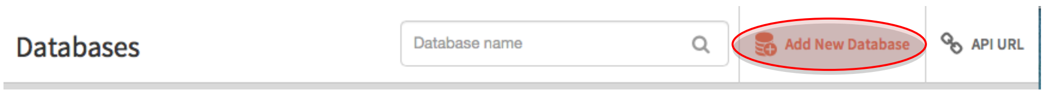
Ease of Use

Work with self-describing JSON documents through a RESTful API that makes every document in your Cloudant database accessible

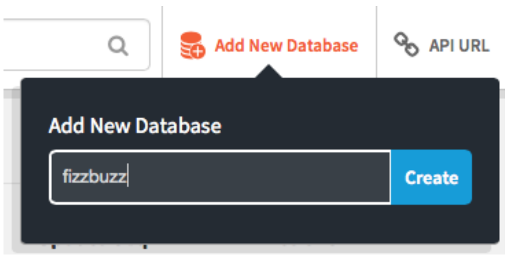
Powerful search, sync and more

With extremely powerful indexing, real time MapReduce and Apache Lucene-based full-text search, Cloudant NoSQL DB makes it easy to

c. After the Cloudant console loads, click **Add New Database** from the top menu.

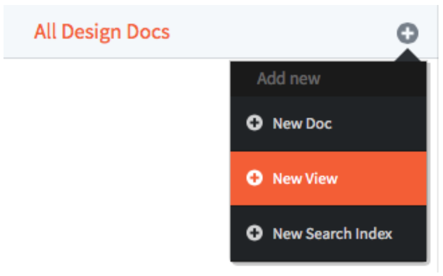


d. Enter `fizzbuzz` as the database name.

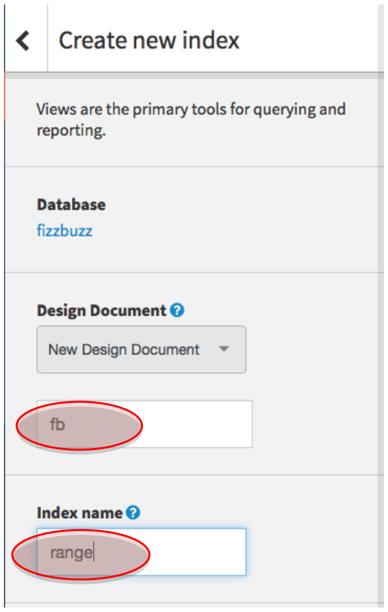


e. Click **Create**.

3. Create a view to be able to search for documents with a given range [from, to]:
   a. Click the plus sign (+) next to **All Design Docs**.



b. In the Create new index dialog, enter the following values:
- `fb` as the Design Document name
- `range` as the Index name



Create new index

Views are the primary tools for querying and reporting.

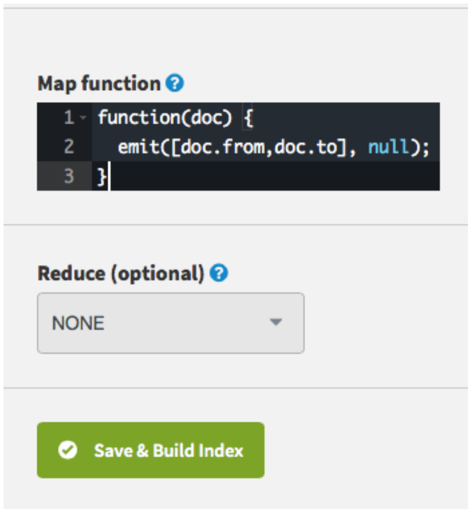
Database
fizzbuzz

Design Document
New Design Document
fb

Index name
range

c. Change the Map function to the following code:

```
function(doc) {
  emit([doc.from,doc.to], null);
}
```



Map function

```
function(doc) {
  emit([doc.from,doc.to], null);
}
```

Reduce (optional)
NONE

d. Click **Save & Build Index**.

The database is now ready to receive documents.

To use Cloudant in Node.js, there are a number of options from making direct REST API calls against the Cloudant API to using one of the libraries. In this exercise, you'll use the Cloudant library.

4. Add the Cloudant library to the Eclipse project:
   a. In Eclipse in the Terminals view, stop Mocha if it's still running: Ctrl+C.

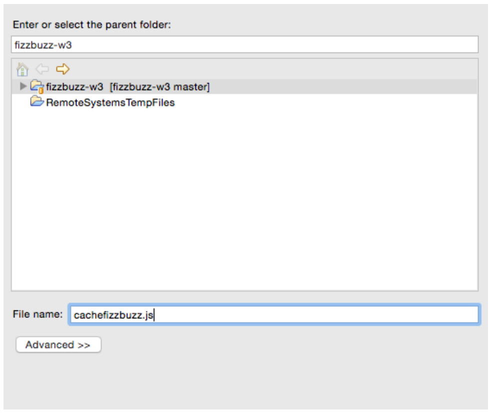
   b. Enter the following command:

```
npm install cloudant@1.0.0 --save --save-exact
```

   This command ensures that version 1.0.0 is used and is entered in the `package.json` file to use exactly 1.0.0, not 1.0.0 or later.

For this exercise, you'll create a new route in the server so that you have access to the caching and noncaching versions of the API.

5. Create the files that are needed to start implementing the caching version of the API:
   a. Right-click the project name in the Project Explorer view and click **New > File**. Name the file `cachefizzbuzz.js`.



Enter or select the parent folder:
fizzbuzz-w3

File name: cachefizzbuzz.js

   b. Enter the following code in the file and then save the file:

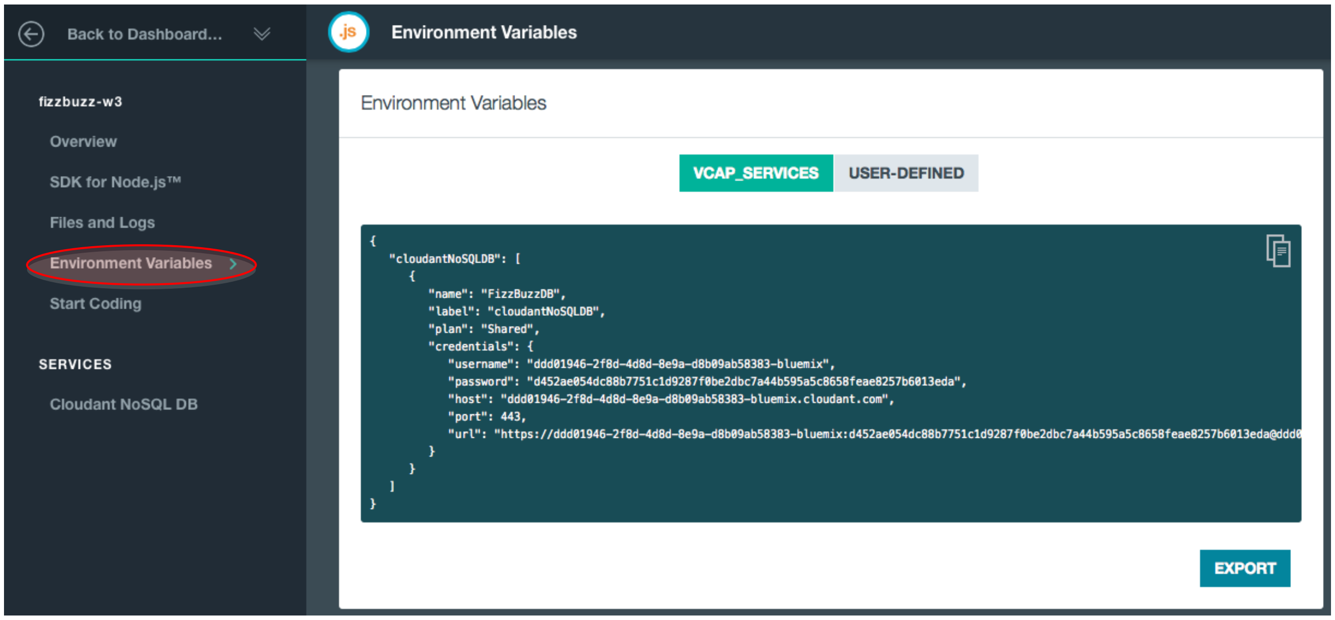
```
var FizzBuzz = require("./fizzbuzz");

var CacheFizzBuzz = function(dbURL) {
  this._dbURL = dbURL;
  this._Cloudant = require("cloudant")(dbURL);
  this._fizzbuzz = new FizzBuzz();
};

module.exports = CacheFizzBuzz;
```

   c. Right-click the test directory in the Project Explorer view and click **File > New**. Name the file `cachefizzbuzz.test.js`.

d. Enter the following content in the `cachefizzbuzz.test.js` file and then save the file:

```
var sinon = require("sinon");
var CacheFizzBuzz = require("../cachefizzbuzz.js");
```

To use the Cloudant database in the application, use the VCAP_SERVICES environment variable to get access to the database location and credentials. See the value of the environment variable in the Bluemix web UI.



```
{
   "cloudantNoSQLDB": [
      {
         "name": "FizzBuzzDB",
         "label": "cloudantNoSQLDB",
         "plan": "Shared",
         "credentials": {
            "username": "ddd01946-2f8d-4d8d-8e9a-d8b09ab58383-bluemix",
            "password": "d452ae054dc88b7751c1d9287f0be2dbc7a44b595a5c8658feae8257b6013eda",
            "host": "ddd01946-2f8d-4d8d-8e9a-d8b09ab58383-bluemix.cloudant.com",
            "port": 443,
            "url": "https://ddd01946-2f8d-4d8d-8e9a-d8b09ab58383-bluemix:d452ae054dc88b7751c1d9287f0be2dbc7a44b595a5c8658feae8257b6013eda@ddd0
         }
      }
   ]
}
```

For the Cloudant library, only the URL is needed from the credentials section.

6. Parse the VCAP_SERVICES variable and extract the URL. If the application is not running on Bluemix, create a local URL:
   a. Add the following code to the `server.js` file:

      - Add code to parse the VCAP_SERVICES and extract the database URL under the existing code to obtain the server host and port values.
      - Add a new variable named `CacheFizzBuzz` and initialize it with `require("./cachfizzbuzz");`

The top of the `server.js` file should now contain this code:

```
var express = require("express");
var app = express();
var FizzBuzz = require("./fizzbuzz");
var CacheFizzBuzz = require("./cachefizzbuzz");

var server_port = process.env.VCAP_APP_PORT || 3000;
var server_host = process.env.VCAP_APP_HOST || "localhost";
var dbURL = "";
if (process.env.VCAP_SERVICES) {
      var env = JSON.parse(process.env.VCAP_SERVICES);
      dbURL = env.cloudantNoSQLDB[0].credentials.url + "/fizzbuzz";
} else dbURL = "http://localhost:5984/fizzbuzz";
```

This code uses the database name `fizzbuzz`.

b. In the `server.js` file, add the new route to the server. After the code that defines the /fizzbuzz_range route, add the following code:

```
app.get("/cache_fizzbuzz_range/:from/:to", function (req, res) {
  var cachefizzbuzz = new CacheFizzBuzz(dbURL);
  var from = req.params.from;
  var to = req.params.to;

  cachefizzbuzz.fizzBuzzRange(from, to, function(data) {
     res.send(data);
  });
});
```

Because a database call is used in the implementation of this API, the code must use a callback function to make the res.send(data) call because the database call is asynchronous.

c. Save the `server.js` file.

To implement the new API, this exercise will not follow a strict, test-driven approach due to time constraints. Instead, you'll focus on the requirements to test code by invoking a remote service.

The application needs to perform two actions against the database:

- Search for results for a given range
- Save the result for a calculated range

Calls to external services need to use asynchronous function calls in JavaScript to avoid blocking the user thread.

There are a number of ways to handle asynchronous calls in JavaScript. Passing an anonymous function as a parameter is a common approach, however this can be problematic to test. Having deeply nested stack of callbacks can also make code difficult to understand and maintain. To overcome this problem one approach is to create functions to handle callbacks outside the asynchronous calls1. This way they can be tested individually.

```
asyncCall(p1, p2, function(err, data) {
   <implementation of callback function>
});
```

becomes:

```
function cb(err, data) {
  <implementation of callback function>
}
….
asyncCall(p1, p2, cb);
```

With this implementation the callback function 'cb' can be unit tested without the 'aysnCall' being invoked. If 'asynCall is a call to a remote server, then we can now test how our code handles the response without any calls to a remote server.

Note: JavaScript Promises provide a better way to handle asynchronous behavior, but to keep the code easier to follow for non-JavaScript developers they are not used in this exercise.

d. Add the following code above the line of code starting with **`modules.export`** in the file `cachefizzbuzz.js`. This code implements the cache functionality.

```
CacheFizzBuzz.prototype.fizzBuzzRange = function(start, end, callback) {
  var self = this;
  var parms = {
      key : [ start, end ],
      include_docs : true
  };

  self._Cloudant.view('fb', 'range', parms, function(err, body) {
    self._processDBResult(err, body, start, end, callback);
  });
};

CacheFizzBuzz.prototype._dbStoreCalculatedResult = function(data) {
  this._Cloudant.insert(data);
};

CacheFizzBuzz.prototype._processDBResult = function(err, body, start,
end,
    callback) {
  var fromDB = false;
  var data = {};
  if ((!err) && (body.rows.length > 0)) {
    delete body.rows[0].doc._id;
```

```
            delete body.rows[0].doc._rev;
            data = body.rows[0].doc;
          } else {
            data = {
                "from" : start,
                "to" : end,
          "result" : this._fizzbuzz.convertRangeToFizzBuzz(start, end)
      };
      if (!err) {
        this._dbStoreCalculatedResult(data);
      }
    }
    callback(data);
  };
```

Implement the tests for the cachefizzbuzz code (optional)

Testing the previous code requires that you isolate the calls to the database and ensure that they are not executed. To do this, use mocks and stubs from Sinon.JS.

To test the dbStoreCalculatedResult() function, check that the Cloudant function is called with the correct parameters. Unit testing should not check that a document was created in the database.

When you test the function, the call to coudant.insert() should not be made. Sinon.JS provides stub and mock functionality to replace the call, but it allows the test to verify that the call will be correctly made when it's running in production mode.

1. Add the following code to the `cachefizzbuzz.test.js` file in the `test` folder:

```
var testResult1 = {
    "from" : "1",
    "to" : "20",
    "result" : [ "1", "2", "Fizz", "4", "Buzz", "Fizz", "7", "8", "Fizz",
"Buzz",
                 "11", "Fizz", "13", "14", "FizzBuzz", "16", "17", "Fizz",
"19", "Buzz" ]
};

describe("CacheFizzbuzz", function() {
      var f = new CacheFizzBuzz("http://user:password@localhost/fizzbuzz");

      describe("dbStoreCalculatedResult()", function() {
        it("calls Cloudant to store the doc", function() {

          var mock = sinon.mock(f._Cloudant);
          mock.expects("insert").withArgs(testResult1).once();

          f._dbStoreCalculatedResult(testResult1);

          mock.verify();
          mock.restore();
        });
      });

      // remaining tests go above here
 });
```

This test uses the mock function of Sinon.JS. The _Cloudant object is mocked, so all calls will be captured by the mock. You then set an expectation that the insert function will be called one time with a given set of FizzBuzz results.

After the mock and expectation are configured, the function being tested is called. Then, you ask the mock to verify that all expectations are met. At the end of the test, you restore the _Cloudant object to remove the mock.

2. Add the following code to the `cachefizzbuzz.test.js` file above the line “remaining tests go above here”:

```
describe("fizzBuzzRange()", function() {
            var cbFunction = function(data) {
            };

            it("calls Cloudant to get record from the fb/range view in DB",
                function() {
                  var stub = sinon.stub(f._Cloudant, "view");

                  f.fizzBuzzRange("1", "20", cbFunction);

                  expect(stub.withArgs('fb', 'range', {
                    include_docs : true, key : [ "1", "20" ]
                  }, sinon.match.any).calledOnce).to.be
                  .eql(true,
                  "Expected cloudant.view to be called only once with correct
parameters");

                  f._Cloudant.view.restore();
                });
          });
```

The code above tests the fizzBuzzRange() function. This function asks the database for any stored results for the range that is specified by the start and end input parameters.

The returned results are then passed to the _processDBResult() function. To test the function, you provide a fake callback function cbFunction and use the stub feature of Sinon.JS to ensure that no call to Cloudant is made. Unlike the mock, the expectations are tested after the function being tested is called. The anonymous function that is passed as the callback function to the cloudant view function call can be ignored in the test by using `sinon.match.any`.

The final part to test is the function that processes the return from the Cloudant.view call. This function should look at the returned results. If a document is found to match the input key, the result should be returned. If there is no match, the result must be calculated by using the functionality that is implemented in the previous exercise. The result must be stored in the database before the result is returned.

3. Create the tests by adding more test data. Under the definition of test1, add the following code:

```
var testResult2 = {
    "from" : "2",
    "to" : "21",
    "result" : ["2", "Fizz", "4", "Buzz", "Fizz", "7", "8", "Fizz", "Buzz",
                "11", "Fizz", "13", "14", "FizzBuzz", "16", "17", "Fizz",
"19", "Buzz", "Fizz"]
};

var DBResult1 = {
    "total_rows" : 7,
    "offset" : 6,
    "rows" : [ {
      "id" : "35523244d141fb56c2e6b8dfa58d7fed",
      "key" : [ "1", "20" ],
      "value" : null,
      "doc" : {
        "_id" : "35523244d141fb56c2e6b8dfa58d7fed",
        "_rev" : "1-a0337a71e877726cb8fda7d6ceeb2bfc",
        "from" : "1",
        "to" : "20",
        "result" : [ "1", "2", "Fizz", "4", "Buzz", "Fizz", "7", "8", "Fizz",
                     "Buzz", "11", "Fizz", "13", "14", "FizzBuzz", "16", "17",
"Fizz",
                     "19", "Buzz" ]
      }
    } ]
};

 var DBResult2 = {
     "total_rows" : 7,
     "offset" : 7,
     "rows" : []
 };
```

There are 3 test scenarios to cover:

- Results are found in the database
- No results are found in the database
- Database returns an error

4. Add the following code below the previous test but above the line “remaining tests go above here”:

```
describe("processDBResult()", function() {
   var cbFunction = sinon.spy();
   var fizzBuzzSpy = null;

   beforeEach( function() {
     fizzBuzzSpy = sinon.spy(f._fizzbuzz, "convertRangeToFizzBuzz");
   });

   afterEach( function() {
     cbFunction.reset();
     f._fizzbuzz.convertRangeToFizzBuzz.restore();
   });

       it("processes the results from Cloudant with a valid result set",
           function() {
             var storeResultsSpy = sinon.spy(f, "_dbStoreCalculatedResult");

             f._processDBResult(false, DBResult1, "1", "20", cbFunction);

             expect(cbFunction.withArgs(testResult1).calledOnce).to.be
             .eql(true, "Expected processDBResult to parse DB return");
             expect(fizzBuzzSpy).callCount(0);
             expect(storeResultsSpy).callCount(0);

             f._dbStoreCalculatedResult.restore();
           });

       it("calculates the results when no results found then saves to DB",
                    function() {
                      storeResultsStub = sinon.stub(f, "_dbStoreCalculatedResult");

                      f._processDBResult(false, DBResult2, "2", "21", cbFunction);

                      expect(cbFunction.withArgs(testResult2).calledOnce).to.be
                      .eql(true,"Expected calculated result when no data from DB");
                      expect(fizzBuzzSpy.withArgs("2", "21").calledOnce).to.be
                      .eql(true, "convertRangeToFizzBuzz should be called to
    calculate results");
                      expect(storeResultsStub.withArgs(testResult2).calledOnce).to.be
                      .eql(true, "Expect calculated restuls to be stored in DB");

                      f._dbStoreCalculatedResult.restore();
                    });

                it("calculates results when DB error,but doesn't store results",
                    function() {
                      storeResultsSpy = sinon.spy(f, "_dbStoreCalculatedResult");

                      f._processDBResult(true, DBResult1, "1", "20", cbFunction);

                      expect(cbFunction.withArgs(testResult1).calledOnce).to.be
                      .eql(true, "Expected processDBResult create empty result with
    DB error");
                      expect(fizzBuzzSpy.withArgs("1", "20").calledOnce).to.be
                      .eql(true, "convertRangeToFizzBuzz should be called to
    calculate results");
                      expect(storeResultsSpy).callCount(0);

                      f._dbStoreCalculatedResult.restore();
                    });
              });
```

To avoid repetition in tests, the beforeEach and afterEach hook functions are used to set up the spies and stubs used in each test.

All the tests should now pass.

This exercise should give you a good idea of how a unit test can be used when services are used. Because unit testing is a complex topic, we’ve provided only a short example.