



## **Bluemix Hands-On Workshop**

### **Section 5 - Maximising the value of Bluemix**

#### Section 5 - Maximising the value of Bluemix

**Version:** 3  
**Last modification date:** 8-Jul-15  
**Owner:** IBM Ecosystem Development

## **Table of Contents**

---

Bluemix Hands-On Workshop.....	1
Section 5 - Maximising the value of Bluemix.....	1
Introduction .....	3
Exercise 5.a - Creating the project source code repository .....	4
Exercise 5.b - Setting up development tooling .....	14
Exercise 5.c - Test Driven Development .....	20
Implement divisibleBy .....	20
Implement convertToFizzBuzz .....	24
Implement convertRangeToFizzBuzz (introduces Sinon) .....	26
Exercise 5.d - Adding the REST API and deploying to Bluemix.....	28
Setup DevOps pipeline to automatically test and deploy code .....	30

## Introduction

This workshop lab will demonstrate an approach for developing applications using principles from agile development using DevOps processes and tooling.

The lab takes you through the steps to develop a REST API that will calculate the FizzBuzz result for a given range.

FizzBuzz is a children's numeracy game where any number divisible by 3 is replaced by the word 'Fizz', any number divisible by 5 is replaced with the word 'Buzz' and any number divisible by both 3 and 5 is replaced with the work 'FizzBuzz'

e.g. for the range 1 .. 20 the response is:

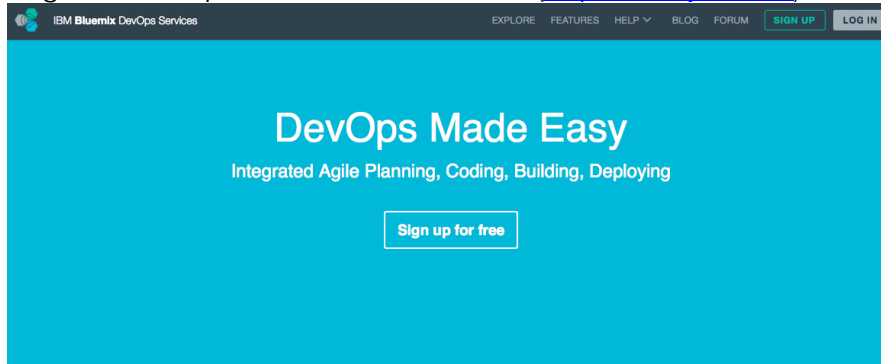
"1, 2, Fizz, 4, Buzz, Fizz, 7, 8, Fizz, Buzz, 11, Fizz, 13, 14, FizzBuzz, 16, 17, Fizz, 19, Buzz"

You will be using NodeJS as the runtime and will use a test driven development practice to create the solution.

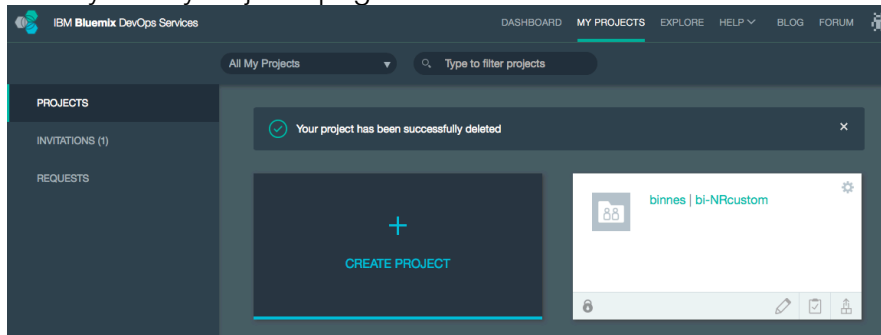
## Exercise 5.a - Creating the project source code repository

Before writing any code a developer needs to have the correct tooling and a source code repository. For this lab we will be using the Git service from IBM DevOps Services for Bluemix.

1. Log into DevOps Services for Bluemix (<http://hub.jazz.net>)




2. From your MyProjects page select CREATE PROJECT



3. Enter the following details:
  1. A project name
  2. Select to create a new repository
  3. Select to create a Git repo on Bluemix

## Create a project

binnes |  Create a new repository or link to an existing one 

Create a new repository



Link to an existing repository

Choose your new repository location 

Create a Git repo on GitHub



Create a Git repo on Bluemix



Create Jazz SCM on Bluemix

Private Git URL: <https://hub.jazz.net/git/binnes/fizzbuzz-w3>

4. If you want to share your project and make it searchable to public untick the Private Project box, otherwise leave it ticked to leave it as a private project
5. Add features for Scrum development if you want to use the track and plan tooling to support a Scrum agile methodology
6. Tick the box to make the project a Bluemix project, you will then be asked to provide the details of the Bluemix space the application will be deployed to

Select project contents

☒ Private ProjectPrivate projects are only accessible by invited team members. [Learn more](#)☒ Add features for Scrum development (This option can only be added at project creation time.)Select this if you're familiar with Scrum and plan to deliver software on regular sprints. ☒ Make this a Bluemix ProjectSelect this if you want to deploy your application to the IBM Bluemix cloud platform. [Find out how](#) Bluemix projects are charged for Track & Plan and Delivery Pipeline (Build & Deploy) usage in accordance with the [Bluemix pricing plan](#).

Select a Bluemix space to bill your services to:

Region

Organization

Space

These selections can be changed later in the options for your Project Settings.

Your project must be connected to a [Bluemix space](#) for billing purposes. Normally you should select the "dev" (Development) space you want to use with this application.

If you prefer to do this later just uncheck "Make this a Bluemix Project" to continue.

Under the Standard plan, Track and Plan is free for 3 users. Delivery Pipeline offers 60 minutes of build time and 2 deployers per month for free. To continue using these services beyond the free tier, you must add one or both of the Track and Plan and Delivery Pipeline (Build & Deploy) services to the space you have configured for billing. [Learn more](#).

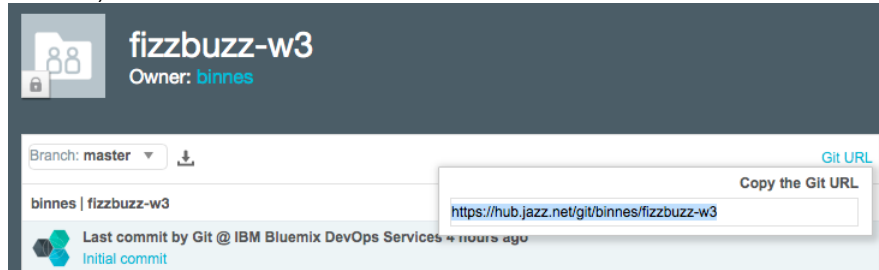
After you've added either of these services to your billing space, you can monitor usage in the Bluemix Account usage page.

**CREATE**

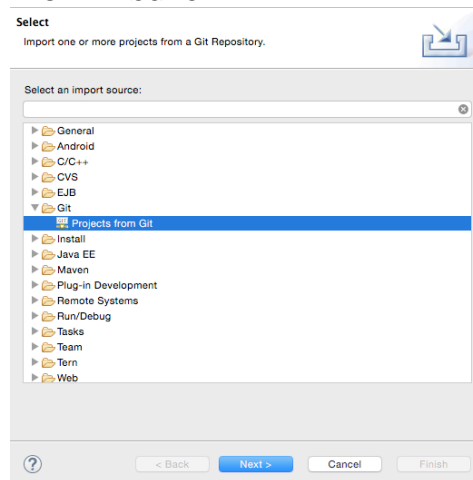
7. Press CREATE to create the project

Now the code repository has been created we can create a new Eclipse project based on the repository.

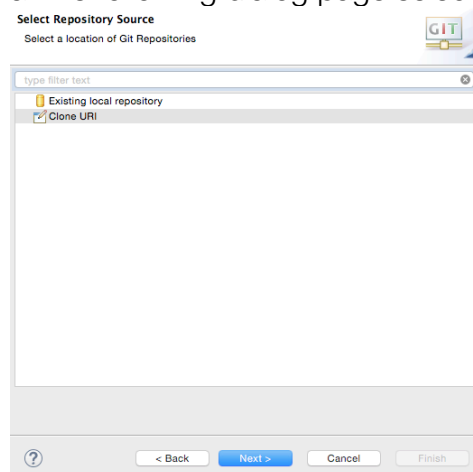
1. In DevOps Services console select the Git URL link then copy the URL (Ctrl C or Cmd C)



2. Launch Eclipse and then from the main menu select:
  - File -> Import
  - From the dialog that appears, choose Git -> Project from Git then select 'Next >' button

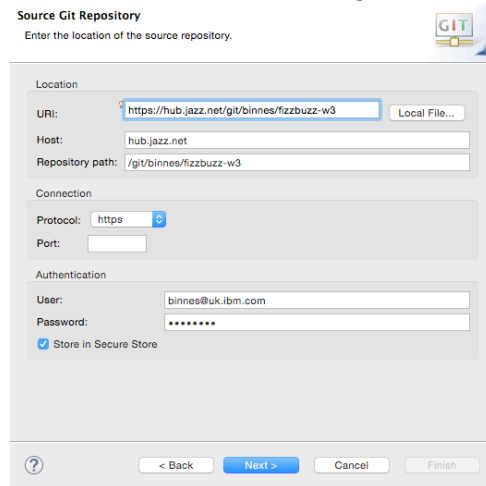


- at the following dialog page select Clone URI then the 'Next >' button



- at the following dialog page paste the Git repository URL into the URI Location field. This will automatically populate the Host and Repository path. You will need to enter your DevOps Services user and password (optionally select to store the Git credentials in a secure store within Eclipse -

you will have to provide additional information to initialize the secure store if this is the first time storing credentials). Select 'Next >' button



**Source Git Repository**  
Enter the location of the source repository.

Location

URI:  Local File...

Host:

Repository path:

Connection

Protocol:

Port:

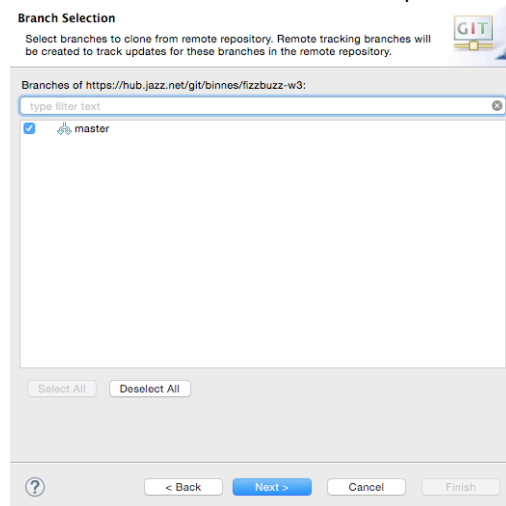
Authentication

User:

Password:

☒ Store in Secure Store

- Eclipse will query the Git repository and list the branches available - as we have just created our repository, only the master branch exists. Leave the master branch selected and press 'Next >'



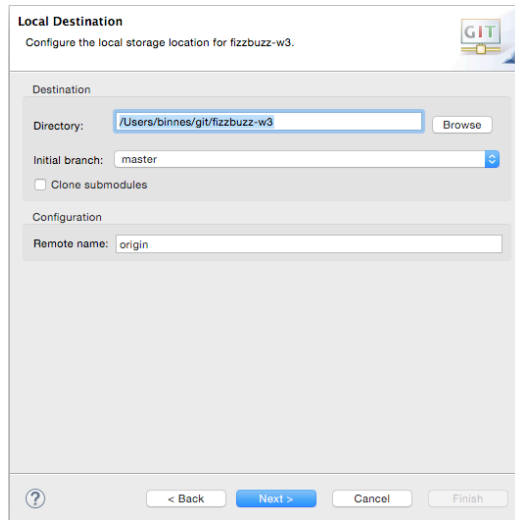
**Branch Selection**  
Select branches to clone from remote repository. Remote tracking branches will be created to track updates for these branches in the remote repository.

Branches of https://hub.jazz.net/git/binnes/fizzbuzz-w3:

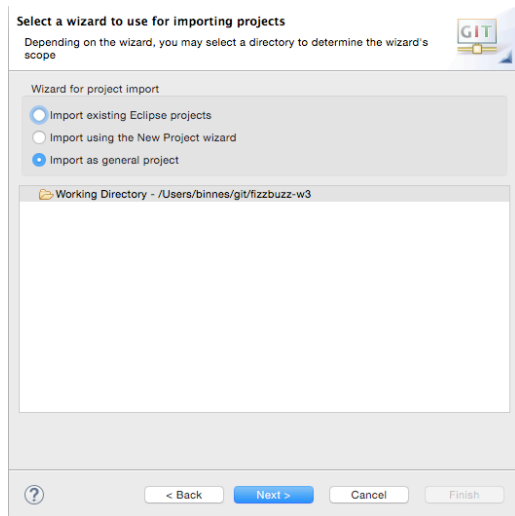
type filter text

☒ master

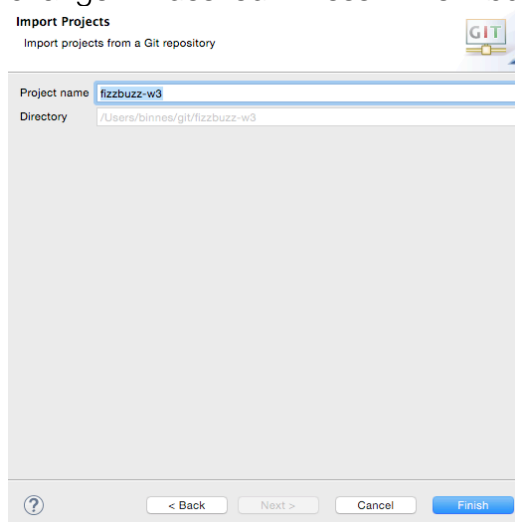
- The local destination is then chosen. The default location is <user home directory>/git/<project name>. I suggest you leave this location at the suggested default. If you want to change it you need to remember the location as you will need it later in the lab. Select 'Next >' button.



- Select the wizard to import as a general project then select the 'Next >' button.



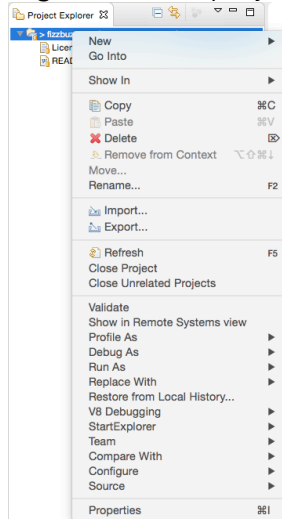
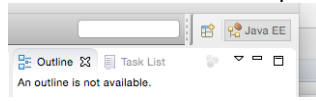
- At the next dialog screen you can set the Eclipse project name. I suggest you leave it the same as the DevOps Service project name, but feel free to change it if desired. Press 'Finish' button



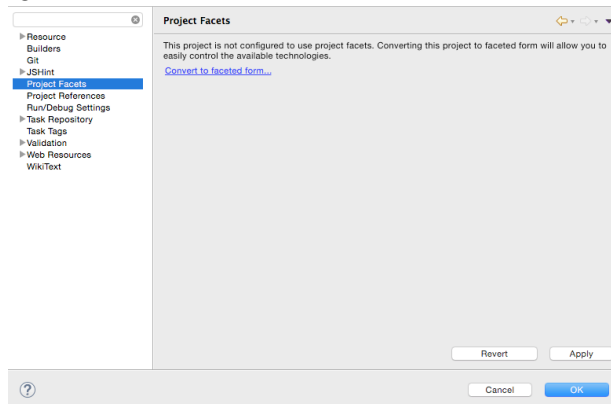


You now have an Eclipse project linked to your DevOps Services Git repository. To complete the project setup we need to configure Eclipse to support Javascript development:

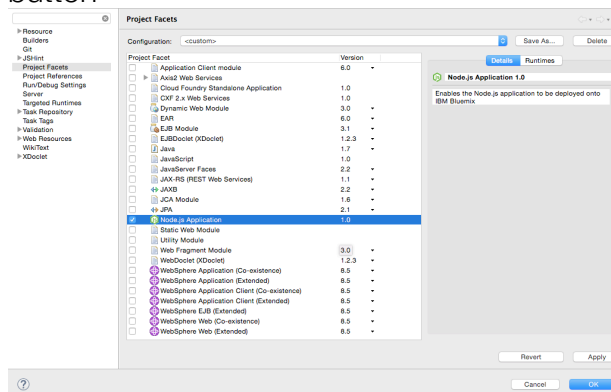
1. Switch Eclipse to the Javascript perspective.
  - Select the add Perspective button at the top right of the Eclipse window
  - select Javascript from the list that appears.
2. Enable the NodeJS facet on the project
  - Right click the project name and select properties from the list that appears



- In the dialog that appears select 'Project Facets' then 'Convert to faceted form...'



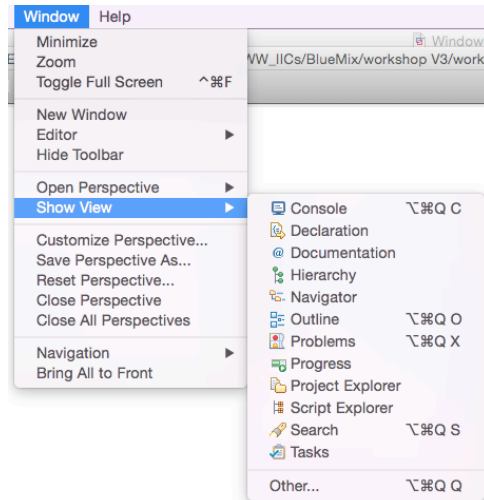
- When the list of facets appears select 'Node.js Application' then the 'OK' button



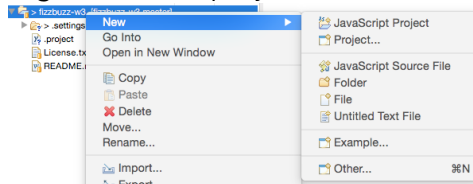
adding the facet will automatically create a sample application in the project. Remove the app.js and package.json file (right click each file and select delete, then confirm the deletion)

You will notice that the project is showing there are changes to the local copy of the project. These are the Eclipse project setting files. I don't want to save these as part of the source code:

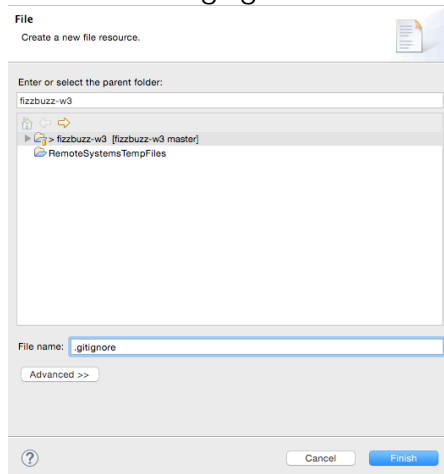
3. Add a navigator view to see all the files in the project:
  - Select Window -> Show View then select Navigator



- Right click the project name in the Navigator view and select New -> File



- name the file .gitignore then select the 'Finish' button.

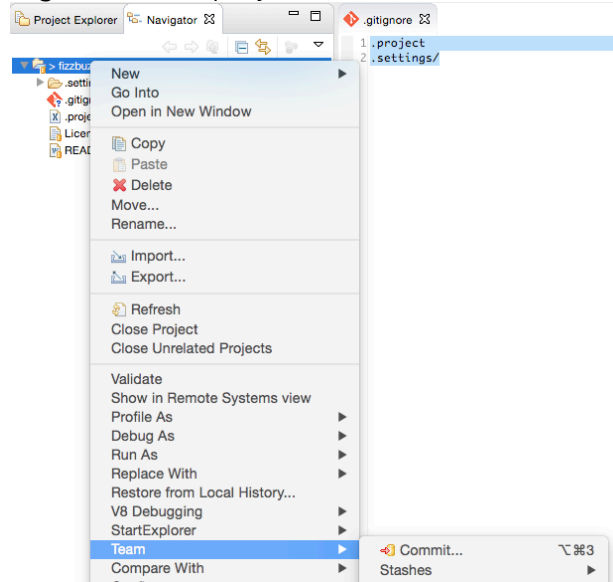


- add the following 2 lines to the .gitignore file (note: Windows users, leave the slash as is specified, don't change it to '.settings\'):
- ```
.project
.settings/
```

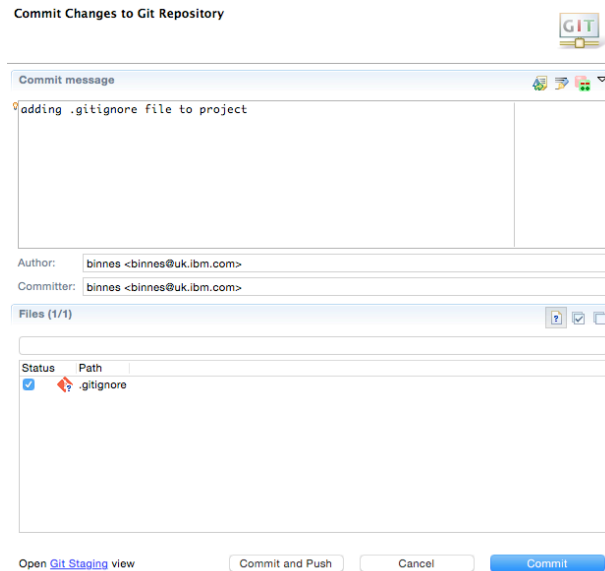
The project still shows that there are changes. This is because the .gitignore file is a new file not committed and pushed to the master branch on the git server.

4. Commit and push the .gitignore file:

- Right click the project name and select Team -> Commit... from the menu



- in the dialog enter a commit message and select the .gitignore file then select 'commit and push' button



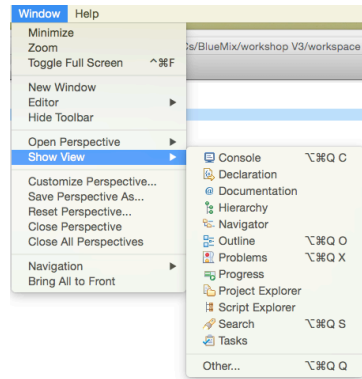
- close the confirmation dialog by selecting OK button

The project should no longer show there are outstanding changes.

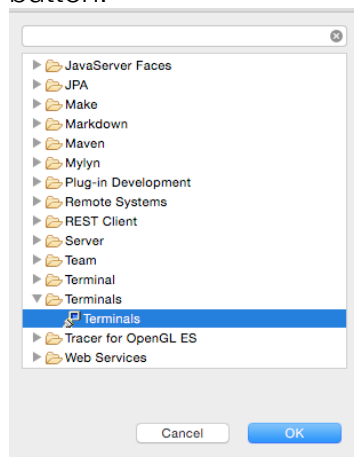
The final setup needed is to add a new window to access the command line - this is not absolutely necessary, you may choose to use a command window outside Eclipse, but I find having everything within Eclipse provides a better working environment.

## 5. Create a terminals view:

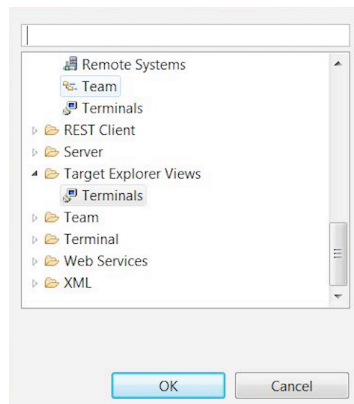
- From the main menu select Window -> Show View then select other



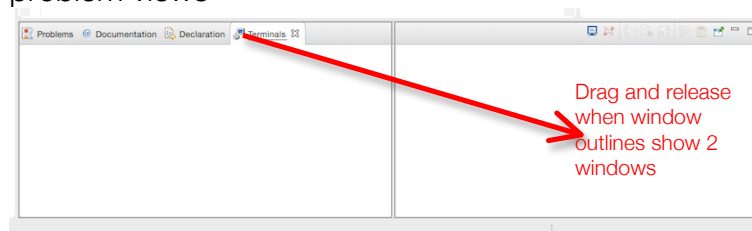
- In the dialog that appeared select Terminals then Terminals and then the OK button:



If Terminals is not in the list, you may have Target Explorer Views:

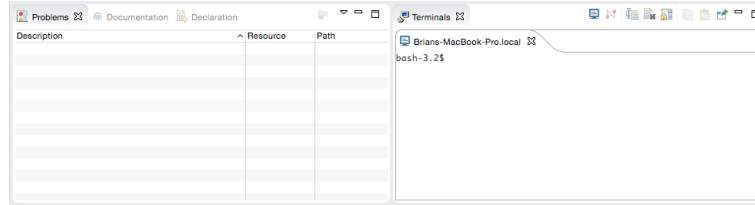


- When the terminal window appears you can drag the tab to split the bottom section of the screen to allow concurrent viewing of the terminal and the problem views



## Section 5 - Maximising the value of Bluemix

- If a command prompt doesn't appear you can press the open terminal icon to start a new terminal - use the default setting, on some platforms this will happen automatically.

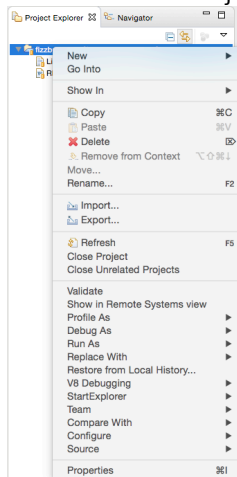


## Exercise 5.b - Setting up development tooling

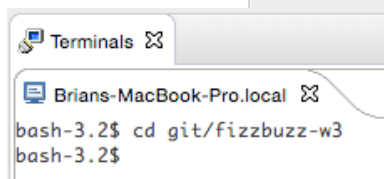
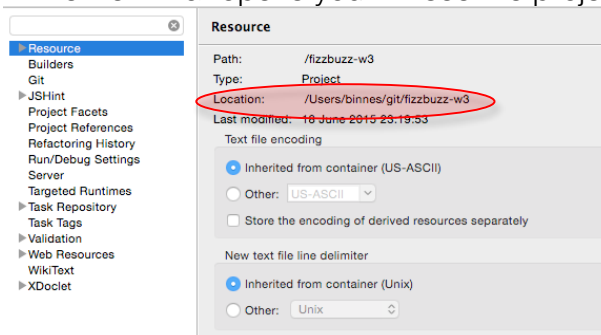
Using appropriate developer tooling with features to automatically spot errors, provide code assist etc. can make a developer more productive and help identify possible problems. In the previous step a new project was created that will now be used. The first step is to ensure the Javascript tooling enabled and the test framework is created.

Create the Javascript project:

1. In the terminals view change to the directory containing the project
  - Mac default `cd <user home>/git/<project name>`
  - Windows default `cd <user home>\git\<project name>`
  - If you want to verify where your project is located - right click the project name in the Project Explorer view then select properties:



- In the view that opens you will see the project location on the resources tab:



- Run **'npm init'** in the terminals window and answer the prompts as shown in the screenshot below. Note Windows users use '/' instead of '\' when specifying the test command.
  - If the terminals window is too small double clicking the tab will bring it full screen, double clicking again will return it to its original size and location.

```
bash-3.2$ npm init
This utility will walk you through creating a package.json file.
It only covers the most common items, and tries to guess sane defaults.
```

See `npm help json` for definitive documentation on these fields and exactly what they do.

Use `npm install <pkg> --save` afterwards to install a package and save it as a dependency in the package.json file.

Press ^C at any time to quit.

name: (fizzbuzz-w3)

version: (0.0.0)

description: REST API to calculate FizzBuzz values for a given range

entry point: (index.js) server.js

test command: node\_modules/.bin/mocha

git repository: (https://hub.jazz.net/git/binnes/fizzbuzz-w3)

keywords:

author: Brian Innes

license: (ISC)

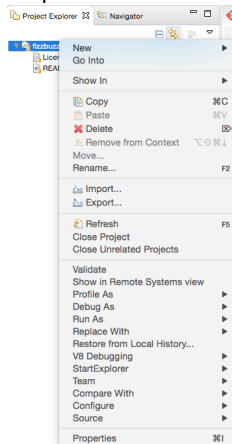
About to write to /Users/binnes/git/fizzbuzz-w3/package.json:

```
{
  "name": "fizzbuzz-w3",
  "version": "0.0.0",
  "description": "REST API to calculate FizzBuzz values for a given range",
  "main": "server.js",
  "scripts": {
    "test": "node_modules/.bin/mocha"
  },
  "repository": {
    "type": "git",
    "url": "https://hub.jazz.net/git/binnes/fizzbuzz-w3"
  },
  "author": "Brian Innes",
  "license": "ISC"
}
```

Is this ok? (yes) yes

```
bash-3.2$
```

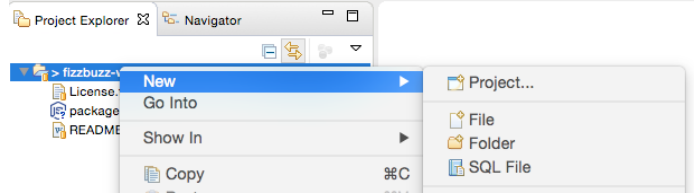
- To bring the created files into Eclipse right click the project name in the Project Explorer view and select Refresh from the menu or use the F5 shortcut key:



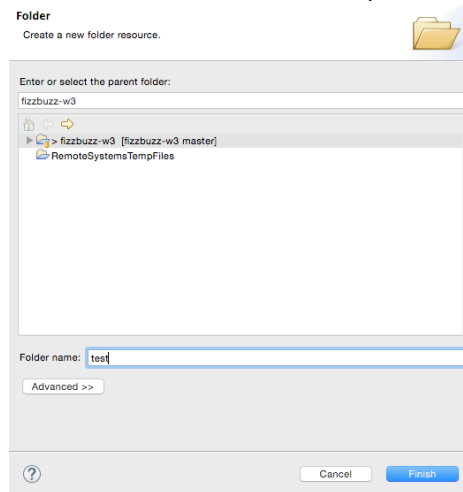
- If the wizard truncated to test script to "mocha" open the package.json file in the Eclipse editor and change it back to "node\_modules/.bin/mocha"

- Tests are expected to be in a directory named test so create a test directory

- Right click the project name in the Project Explorer view and select New -> Folder:



- Name the folder test then press the 'Finish' button:



- To be able to run tests the mocha framework needs to be installed

- In the Terminals view enter **npm install mocha --save-dev**
- Select the project name in the Project Explorer and then F5 to refresh the project in Eclipse and bring in the new files

- Test that everything works as expected by running a test

- In the Terminals view enter **npm test**

```
bash-3.2$ npm test

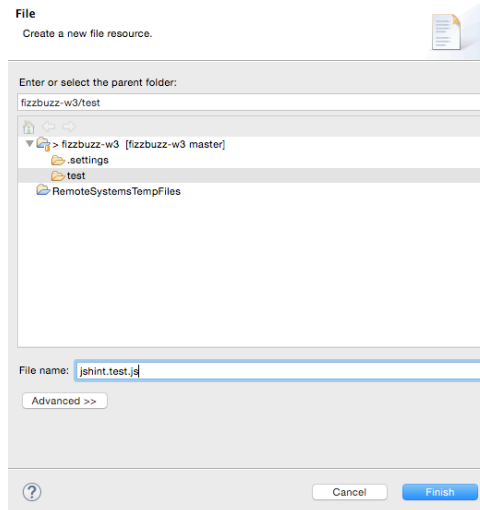
> fizzbuzz-w3@0.0.0 test /Users/binnes/git/fizzbuzz-w3
> mocha

  0 passing (4ms)
```

- There are no tests defined, but we see the test framework ran
- JSHint is a lint program for Javascript. It provides static analysis of code. It is good practice to have this running in developer tooling and many development teams choose to include it as part of their automated test suite. Mocha can run JSHint
  - In the Terminals view enter **npm install mocha-jshint --save-dev**



- Select the test directory of the project in the Project Explorer view then right click and select New -> File, name the file jshint.test.js



- In the file enter the following content then save the file (from the menu File -> Save or Ctrl+S or Cmd+S):  
`require('mocha-jshint') ();`
- Rerun the tests, **npm test** in the Terminals view.

There are many errors from the imported modules in node\_modules directory. This is code outside our control, so should be excluded from tests. JSHint can be configured to not check files using a .jshintignore file

6. Create a .jshintignore file

- right click on the project name in the Project Explorer view New -> File
- add the following content to the file:

```
node_modules
test
```

- rerun the tests **npm test** in the Terminals view

```
bash-3.2$ npm test

> fizzbuzz-w3@0.0.0 test /Users/binnes/git/fizzbuzz-w3
> mocha

jshint
  ✓ should pass for working directory (38ms)

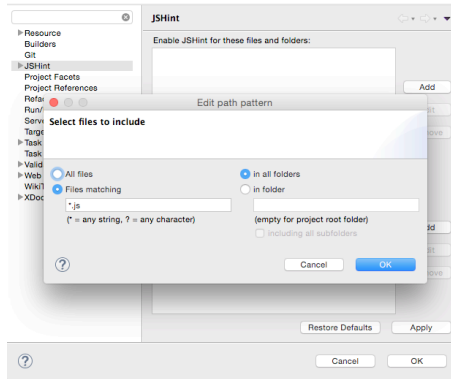
1 passing (45ms)
```

JSHint is now enabled in the test suite, but ideally the code editor should provide feedback of problems. JSHint is included in Eclipse, but needs to be configured:

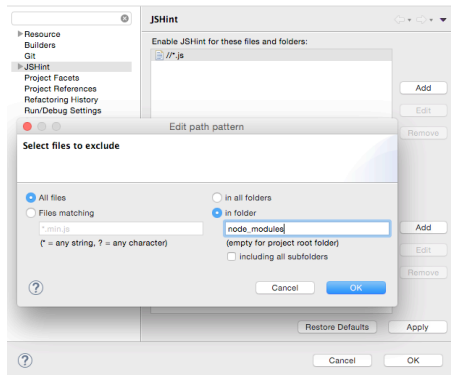
7. Enable JSHint in Eclipse:

- Right click the project name in the Project Explorer view and select properties
- In the dialog that appears select JSHint

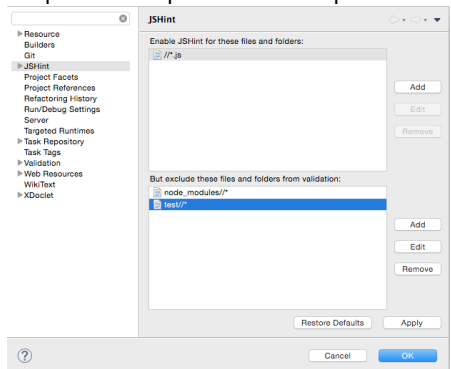
- Select the Add button next to the 'Enable JSHint for these files and folders:' box



- Accept the defaults as shown above and select the 'OK' button
- Select the Add button next to the 'But exclude these files and folders from validation:' box
- Select All files then select in folder and add the node\_modules folder then check the 'include all subfolders checkbox:



- Repeat the previous step to exclude the content of the test folder too:

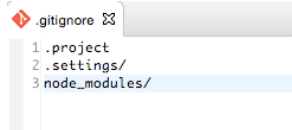


- Select 'OK' to make the changes and close the dialog.

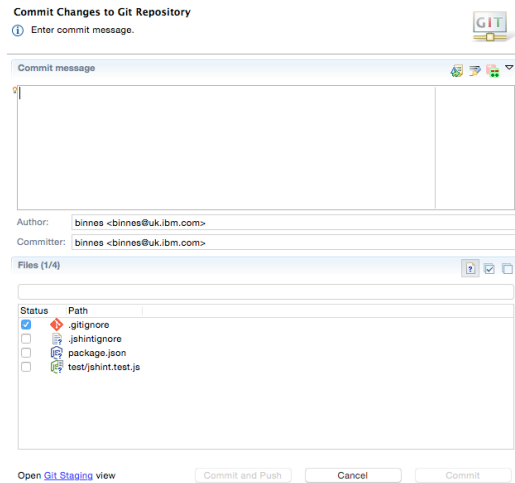
The project is now setup ready to start implementing the REST API, so now would be a good time to commit changes to the Git repo. When it comes to dependencies there is a debate if they should be checked in or re-fetched at build time. To reduce the amount of traffic we need to push to the Git repo for this lab we will exclude the dependencies in node\_modules. Later in the exercise an option to ensure the exact dependencies are fetched each time will be introduced, so we can still ensure there are no accidental or unknown dependencies.

8. Update the Git configuration and commit and push changes to the Git repo.
  - Switch to the Navigator view (needed to see files starting with .)
  - Double click the .gitignore file to edit it

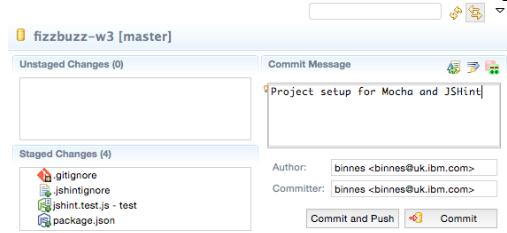
- Add the node\_modules directory to the file then save the file (Ctrl+S or Cmd+S):



- Right Click the project name then select Team -> Commit. Press the link at the bottom right of the dialog to open the Git Staging view



- Drag the 4 files from the Unstaged Changes window to the Staged Changes window then enter a commit message



- Select the 'Commit and Push' button
- Close the confirmation dialog after verify everything completed without error

## Exercise 5.c - Test Driven Development

### Implement divisibleBy

The implementation starts by generating the first test that the implementation must pass. Breaking down the simple problem we have shows the need to have a function to discover if one number is divisible by another number, so the first test will check if 3 is divisible by 3. Before writing the test there is an additional test framework needed. Chai provides the test capability to say "I expect this to be true" or "this should be true", so before writing a test Chai needs to be installed and Mocha configured to use it.

1. Install and configure Chai
  - In the Terminals view enter the command `npm install chai --save-dev`
  - Select the project name in Project Explorer and press F5 to refresh the Eclipse project content.
  - Create a new file in the test folder called `support.js` and add the following content:
 

```
var chai = require("chai");
global.expect = chai.expect;
```
  - Create a new file in the test folder called `mocha.opts` and add the following content:
 

```
--require test/support
```
  - Ensure both files are saved. The files have configured Mocha to use the expect functionality from Chai.
2. Create the first test
  - Create a new file in the test folder called `fizzbuzz.test.js`
    - Right click the test folder in the Project Explorer view then select New -> File
  - Enter code to get access to the code we are about to write to pass the test:
 

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

describe("Fizzbuzz", function() {
  var f = new FizzBuzz();

  describe("divisibleBy()", function() {
    it("when divisible", function() {
      expect(f.divisibleBy(3, 3)).to.be.eql(true);
    });
  });
});
```

Now we have the first test we can write the code to pass the test. Rather than having to manually run the tests there is an option to continually run Mocha, so every time a file is changed the tests are automatically run. This is achieved using the `-w` command line option.

3. Setup Mocha to continually run tests
  - In the Terminals view run the command:
    - `node_modules/.bin/mocha -w` on Mac & Linux systems
    - `node_modules\bin\mocha -w` on Windows systems
    - You will see the tests are failing due to error:
 

```
Error: Cannot find module '../fizzbuzz.js'
```

- Create file in root of project called fizzbuzz.js
- Add the following code to setup to fizzbuzz.js
 

```
var FizzBuzz = function () {
};

module.exports = FizzBuzz;
```
- Save the file, you should now see the tests run with the following result:

```
Fizzbuzz
  divisibleBy()
    1) when divisible

jshint
  ✓ should pass for working directory (99ms)

1 passing (114ms)
1 failing

1) Fizzbuzz divisibleBy() when divisible:
   TypeError: Object [object Object] has no method
   'divisibleBy'
   at Context.<anonymous> (test/fizzbuzz.test.js:8:16)
```

It is now time to implement divisibleBy, but we should only write the minimal code to pass the written test. This could be as simple as returning true, as 3 is divisible by 3!

4. Implement divisibleBy to pass the first test.
  - Add the following code to fizzbuzz.js, above the line starting module.exports...

```
FizzBuzz.prototype.divisibleBy = function(number, divisor) {
  return true;
};
```

- Save the file, you should now see all your tests pass

```
Fizzbuzz
  divisibleBy()
    □ when divisible

jshint
  □ should pass for working directory (61ms)

2 passing (67ms)
```

This code is clearly not correct, but it does pass the test. Another test is need to test when divisibleBy should return false - this will also require the correct implementation of the functionality.

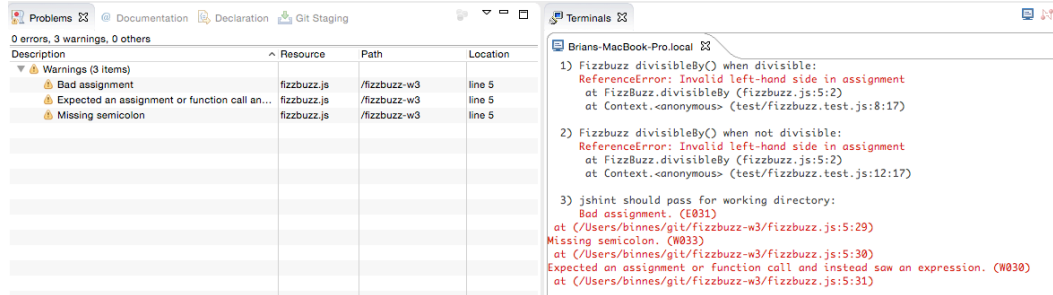
5. Write the test and implement the code when divisibleBy should return false
  - In the fizzbuzz.test.js file add code beneath the existing test to test if 2 is divisible by three. The section of the test for divisibleBy should now look like this:

```
describe("divisibleBy()", function() {
  it("when divisible", function() {
    expect(f.divisibleBy(3, 3)).to.be.eql(true);
  });

  it("when not divisible", function() {
    expect(f.divisibleBy(3, 2)).to.be.eql(false);
  });
});
```

- Save the file, now you should have a failing test again.
- Implement the divideBy function with the following code (There is a deliberate coding error, so copy the code below):  

```
FizzBuzz.prototype.divideBy = function(number, divisor) {  
    return number % divisor = 0;  
};
```
- There are 3 errors reported. We have broken the divideBy test for 3 divided by 3, the new test doesn't pass and we also have JSHint complaining of bad Javascript code. Notice Eclipse is also reporting the JSHint issues in the problem panel.



- Fix the code with the following (and there is still a deliberate mistake):  

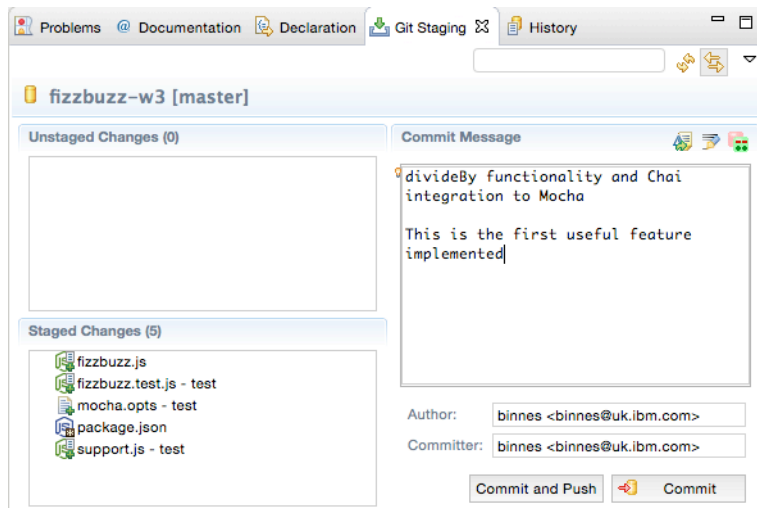
```
FizzBuzz.prototype.divideBy = function(number, divisor) {  
    return number % divisor == 0;  
};
```
- Now we pass the functional tests, but JSHint is still complaining of bad Javascript code. This is because in Javascript there are 2 comparison operators '==' which does type coercion and '===' which does not do type coercion. When using a static number the '===' should be used:  

```
FizzBuzz.prototype.divideBy = function(number, divisor) {  
    return number % divisor === 0;  
};
```
- All the tests now pass

The first function is now implemented and all the tests pass. This is a good time to commit code

6. Commit the code and push to the master repo
  - In the 'Git Staging' view select the files in the 'Unstaged Changes' window and drag to the 'Staged Changes' window. Add a comment then select the 'Commit and Push' button.
    - Note it would be more correct to split out the configuration of Chai and Mocha into a separate commit

## Section 5 - Maximising the value of Bluemix



## Implement convertToFizzBuzz

Now there is a function to check if a number is divisible by 3 or 5 we can implement a function to convert a number to its FizzBuzz value. We need to start with the first test:

1. Implement the first test for convertToFizzBuzz

- In fizzbuzz.test.js add a new test block after the divisibleBy block:

```
describe("divisibleBy()", function() {
  it("when divisible", function() {
    expect(f.divisibleBy(3, 3)).to.be.eql(true);
  });

  it("when not divisible", function() {
    expect(f.divisibleBy(3, 2)).to.be.eql(false);
  });
});
```

- Add the first test to check that 3 is converted to 'Fizz':
- ```
describe("convertToFizzBuzz()", function() {
  it("when divisible by 3", function() {
    expect(f.convertToFizzBuzz(3)).to.be.equal("Fizz");
  });
});
```
- In the fizzbuzz.js add a new function prototype for convertToFizzBuzz and add the code required to pass the first test.

```
FizzBuzz.prototype.convertToFizzBuzz = function(number) {
  return "Fizz";
};
```

- Add another test for Fizz to test 6 also returns 'Fizz' and then add tests to check 5 and 10 return 'Buzz' by adding the following code to fizzbuzz.test.js:

- Implement the functionality in fizzbuzz.js to pass the tests by using the following code:

```
FizzBuzz.prototype.convertToFizzBuzz = function(number) {
  if (this.divisibleBy(number, 3)) {
    return "Fizz";
  }

  if (this.divisibleBy(number, 5)) {
    return "Buzz";
  }
};
```

- All the tests should now pass. Now add tests to check for FizzBuzz and a number when not divisible by either 3 or 5. Use the following code for the tests:

```
describe("convertToFizzBuzz()", function() {
  it("when divisible by 3", function() {
    expect(f.convertToFizzBuzz(3)).to.be.equal("Fizz");
    expect(f.convertToFizzBuzz(6)).to.be.equal("Fizz");
  });

  it("when divisible by 5", function() {
    expect(f.convertToFizzBuzz(5)).to.be.equal("Buzz");
    expect(f.convertToFizzBuzz(10)).to.be.equal("Buzz");
  });
});
```



```

it("when divisible by 15", function() {
  expect(f.convertToFizzBuzz(15)).to.be.equal("FizzBuzz");
  expect(f.convertToFizzBuzz(30)).to.be.equal("FizzBuzz");
});

it("when not divisible by 3, 5 or 15", function() {
  expect(f.convertToFizzBuzz(4)).to.be.equal("4");
  expect(f.convertToFizzBuzz(7)).to.be.equal("7");
});
});

```

- To get the tests to pass, the convertToFizzBuzz function needs to be completed. Use the following code:

```

FizzBuzz.prototype.convertToFizzBuzz =
function(number) {
  if (this.divisibleBy(number, 15)) {
    return "FizzBuzz";
  }

  if (this.divisibleBy(number, 3)) {
    return "Fizz";
  }

  if (this.divisibleBy(number, 5)) {
    return "Buzz";
  }

  return number.toString();
};

```

- All the tests should now pass and we have the function complete. Now is another good time to commit. Use the 'Git Staging' view to stage, add a comment then Commit and Push the changes to the master repo.

## Implement convertRangeToFizzBuzz (introduces Sinon)

The next stage to implementing FizzBuzz is to be able to convert a range of numbers, not just a single number. It is important to ensure the tests for this function do not repeat the tests for convertToFizzBuzz - we already have tests for that. The tests that need to:

- verify when a range is converted the results are returned in the correct order
- verify for a range we call the convertToFizzBuzz function once for each member of the array

To enable this type of testing an additional testing capability is needed. Sinon will be used to provide this capability.

1. Add Sinon and configure test framework to use it
  - In the Terminals view enter Ctrl+C to stop the Mocha tests (on Windows answer Y to terminate batch job)
  - Enter the following command in the Terminals window:
 

```
npm install sinon --save-dev
npm install sinon-chai --save-dev
```
  - Refresh the Eclipse project (select the project name in the Project Explorer view and hit F5)
  - In the Terminals view restart the mocha -w command (up arrow should scroll through previous commands) node\_modules/.bin/mocha -w or node\_modules\bin\mocha -w
  - Modify support.js in the test folder to enable Chai to use Sinon:
 

```
var chai = require("chai");
var sinonChai = require("sinon-chai");

chai.use(sinonChai);
global.expect = chai.expect;
```
2. Add the test for the convertRangeToFizzBuzz function:
  - Add the Describe function below the tests for the convertToFizzBuzz function:
 

```
describe("convertRangeToFizzBuzz()", function() {
});
```
  - Add a test to ensure the results are returned in the correct order:
 

```
describe("convertRangeToFizzBuzz()", function() {
  it("returns in correct order", function() {
    expect(f.convertRangeToFizzBuzz(1, 3)).to.be.eql(["1", "2", "Fizz"]);
  });
});
```
  - Implement the function to satisfy the test with the following code:
 

```
FizzBuzz.prototype.convertRangeToFizzBuzz = function(start, end) {
  return ["1", "2", "Fizz"];
};
```
  - Add a test to ensure the convertRangeToFizzBuzz is called the correct number of times for a given range and is called once for each number in the range. To do this we use Sinon to 'spy' on the invocation of the convertToFizzBuzz function. At the top of the fizzbuzz.test.js file add the following line of code to make sinon available:
 

```
var sinon = require("sinon");
```

- The code for the `convertRangeToFizzBuzz` tests is now:
 

```
describe("convertRangeToFizzBuzz()", function() {
  it("returns in correct order", function() {
    expect(f.convertRangeToFizzBuzz(1, 3)).to.be.eql(["1", "2",
    "Fizz"]);
  });

  it("applies FizzBuzz to every number in the range",
  function() {
    var spy = sinon.spy(f, "convertToFizzBuzz");

    f.convertRangeToFizzBuzz(1, 50);

    for (var i = 1; i <= 50; i++) {
      expect(spy.withArgs(i).calledOnce).to.be.eql(true,
      "Expected convertToFizzBuzz to be called with " + i);
    }
    f.convertToFizzBuzz.restore();
  });
});
```
  - Implement the function to pass the tests with the following code:
 

```
FizzBuzz.prototype.convertRangeToFizzBuzz = function(start, end)
{
  var result = [];

  for (var i = start; i <= end; i++) {
    result.push(this.convertToFizzBuzz(i));
  }

  return result;
};
```
3. The implementation of FizzBuzz is now complete as the code passes all the tests. Commit the code:
- Use the Git Staging view to stage, provide a comment then Commit and Push the changes to the master repository.

## Exercise 5.d - Adding the REST API and deploying to Bluemix

Now the FizzBuzz functionality has been created it needs to be exposed as a REST API. The REST API endpoint will respond to an HTTP GET request sent to:

`http(s)://<host>/fizzbuzz_range/:from/:to`

1. Implement a NodeJS server using the express framework
  - Create a new file called `server.js` in the project directory and add the following code to the file:
 

```
var express = require("express");
var app = express();

var server_port = 3000;
var server_host = "localhost";

var server = app.listen(server_port, server_host, function () {

    var host = server.address().address;
    var port = server.address().port;

    console.log("Example app listening at http://%s:%s", host,
port);

});
```
2. The application requires the express framework it must be installed and added to `packages.json` - this can be achieved using `npm install`:
  - In the Terminals view enter `Ctrl+C` to stop the Mocha tests
  - Enter the following command:
 

```
npm install express --save
```
  - Refresh the Eclipse content by selecting the project name in Project Explorer view then pressing `F5`
3. Implement the REST API
  - Enter the following code after the `'var app = express();'` line:
 

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

app.get("/fizzbuzz_range/:from/:to", function (req, res) {
    var fizzbuzz = new FizzBuzz();
    var from = req.params.from;
    var to = req.params.to;

    res.send({
        from: from,
        to: to,
        result: fizzbuzz.convertRangeToFizzBuzz(from, to)
    });
});
```
4. Test the API
  - To start the NodeJS server, enter the following in the Terminals view:
 

```
npm start
```
  - You should see a response similar to:
 

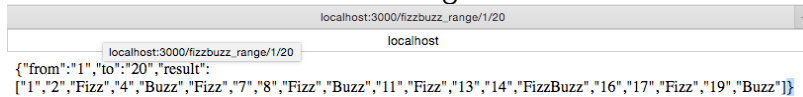
```
npm start
```

```
> fizzbuzz-w3@0.0.0 start /Users/binnes/git/fizzbuzz-w3
> node server.js
```

Example app listening at <http://127.0.0.1:3000>

You can test the API using a browser or command line:

- Enter [http://localhost:3000/fizzbuzz\\_range/1/20](http://localhost:3000/fizzbuzz_range/1/20) into a browser
  - You should see the following in the browser



```
localhost:3000/fizzbuzz_range/1/20
{"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"]}
```

- If you have curl installed on your system, you can test on the command line with the following command:

```
curl -i http://localhost:3000/fizzbuzz_range/1/20
```

```
HTTP/1.1 200 OK
```

```
X-Powered-By: Express
```

```
Content-Type: application/json; charset=utf-8
```

```
Content-Length: 150
```

```
ETag: W/"96-VBtwqT0jYW1L/R6kdGqqKg"
```

```
Date: Mon, 22 Jun 2015 17:43:01 GMT
```

```
Connection: keep-alive
```

```
{ "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" ] }
```

- Enter Ctrl+C to terminate the server

## 5. Modify code for Bluemix/CloudFoundry

- Before pushing the code to Bluemix the server.js file needs to be modified to ensure the server listens on the hostname and port specified in the environment variables. Modify the definitions of the server\_port and server\_host variables to use the VCAP\_APP\_PORT and VCAP\_APP\_HOST environment variables:

```
var server_port = process.env.VCAP_APP_PORT || 3000;
```

```
var server_host = process.env.VCAP_APP_HOST || "localhost";
```

- Create a new file in the project called manifest.yml then enter the following content - change the name to something that will be unique, such as adding your initials at the start of the name:

```
applications:
```

```
- name: bi-FizzBuzz
```

- Save the file

- Create a new file in the project called .cfignore then enter the following content:

```
launchConfigurations/
```

```
.git/
```

```
node_modules/
```

```
npm-debug.log
```

- Modify the .gitignore file (need to use the Navigator view to see file starting with a dot) so the content now matches this:

```
launchConfigurations/
```

```
.project
```

```
.settings/
```

```
node_modules/
```

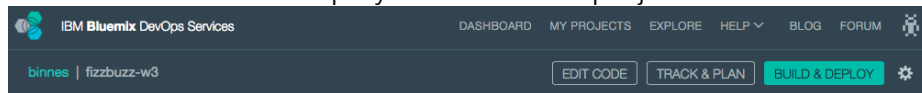
```
npm-debug.log
```

- Commit the change to the master repo using the Git Staging view

## Setup DevOps pipeline to automatically test and deploy code

Now the REST API is working it needs to be deployed to IBM Bluemix. IBM Bluemix DevOps Services provides a Build and Deploy capability.

1. Log onto IBM Bluemix DevOps Services (<https://hub.jazz.net>) and from the MY PROJECTS page select the FizzBuzz project you created in part a).
2. Build a DevOps Pipeline for the project
  - Select the Build and Deploy section of the project



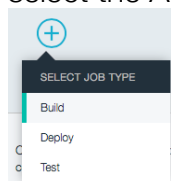
- Select the + ADD STAGE button

 A screenshot of the 'Build' stage configuration form. The 'INPUT' tab is selected. The form has three sections: 'Input Settings', 'Git URL', and 'Branch'. Under 'Input Settings', 'Input Type' is set to 'SCM Repository'. Under 'Git URL', the URL 'https://hub.jazz.net/git/binnes/fizzbuzz-w3' is entered. Under 'Branch', 'master' is selected. The 'Stage Trigger' section has two radio buttons: 'Automatically execute jobs when a change is pushed to Git' (selected) and 'Only execute jobs when a user manually runs this stage'. At the bottom are 'SAVE' and 'CANCEL' buttons.

- On the INPUT page name the stage Build - leave the rest of the fields as default values
- Move to the JOBS tab

 A screenshot of the 'Build' stage configuration form, now in the 'JOBS' tab. The 'INPUT' tab is still selected. The 'JOBS' section shows a large area with a plus icon and the text 'ADD JOB'. Below this, a message states: 'Click "ADD JOB" above to add the first job to this stage. Jobs runs sequentially in a stage, and each job runs in a clean container environment.' At the bottom are 'SAVE' and 'CANCEL' buttons.

- select the ADD JOB button then select 'Build' job type



- Choose Shell Script as Builder Type and enter the script as shown in the screen shot below. Leave the rest of the fields with default values:

**Build** REMOVE

Build Configuration

**Builder Type** ①  
Shell Script

**Build Script** ②

```
#!/bin/bash
npm install npm
node_modules/.bin/npm install
```

**Working Directory** ①

**Build Archive Directory** ①

☐ Enable Test Report ①

**Execution Conditions** ①  
☒ Stop stage execution on job failure

SAVE CANCEL

- Click Save to save the stage
- Click ADD STAGE to add another stage in the DevOps pipeline
- On the Input tab name the stage Test, ensure the Input Type is Build Artifacts from Stage 'Build' and Job 'Build'

**Test**

INPUT JOBS ENVIRONMENT PROPERTIES

**Input Settings**

**Input Type** ①  
Build Artifacts

**Stage**  
Build

**Job**  
Build

**Stage Trigger**

☒ Automatically execute jobs when the previous stage completes successfully  
☐ Only execute jobs when a user manually runs this stage

SAVE CANCEL

- On the JOBS tab select ADD JOB and select Test as the Job Type

- In the Test configuration page ensure Simple type is selected and enter the script as shown on the screen shot

**Test** REMOVE

Test Configuration

**Tester Type** Simple

**Test Command**

```
#!/bin/bash
ARGS=( $@ )

node_modules/.bin/mocha ${ARGS[*]}
```

**Working Directory**

☐ Enable Test Report

**Execution Conditions**

☒ Stop stage execution on job failure

SAVE CANCEL

- Save the stage
- Select to add a third stage, name it Deploy. Ensure Input Type is Build Artifacts from Stage Build and Job Build

**Deploy**

INPUT JOBS ENVIRONMENT PROPERTIES

Input Settings

**Input Type** Build Artifacts

**Stage** Build

**Job** Build

**Stage Trigger**

☒ Automatically execute jobs when the previous stage completes successfully

☐ Only execute jobs when a user manually runs this stage

SAVE CANCEL



- On the JOBS tab select to add a job of type Deploy, the default values should be OK

The screenshot shows the 'Deploy' configuration form. At the top right is a 'REMOVE' button. The form is divided into sections: 'Deploy Configuration', 'Execution Conditions', and 'Deploy Script'. Under 'Deploy Configuration', there are dropdown menus for 'Deployer Type' (Cloud Foundry), 'Target' (IBM Bluemix (United Kingdom) - https://api.eu-gb.bluemix.net), 'Organization' (binnes@uk.ibm.com), 'Space' (dev), and 'Application Name' (fizzbuzz-w3). The 'Deploy Script' section contains a code block with the following content:

```
#!/bin/bash
cf push "${CF_APP}"
# view logs
cf logs "${CF_APP}" --recent
```

Under 'Execution Conditions', there is a checkbox labeled 'Stop stage execution on job failure' which is checked. At the bottom are 'SAVE' and 'CANCEL' buttons.

- Save the stage
- You can start a new build by pressing the run button on the Build stage

The screenshot shows the 'Pipeline: All Stages' view. It displays three stages: Build, Test, and Deploy. Each stage has a 'STAGE NOT RUN' status. The 'Build' stage shows 'LAST INPUT' as 'Git URL' and 'Not yet run'. The 'Test' stage shows 'LAST INPUT' as 'Stage: Build / Job: Build' and 'Not yet run'. The 'Deploy' stage shows 'LAST INPUT' as 'Stage: Build / Job: Build' and 'Not yet run'. Each stage also has a 'JOBS' section with a 'View logs and history' link and a 'LAST EXECUTION RESULT' section with 'No results'.

- You should see the Build stage start to run, once it completes the Test stage should automatically run then finally the deploy stage.

The screenshot shows the 'Pipeline: All Stages' view after the stages have completed. The 'Build' stage shows 'STAGE PASSED' and 'LAST INPUT' as 'Last commit by binnes 2 hr ago implemented REST API Final code chan...'. The 'Test' stage shows 'STAGE PASSED' and 'LAST INPUT' as 'Stage: Build / Job: Build'. The 'Deploy' stage shows 'STAGE PASSED' and 'LAST INPUT' as 'Stage: Build / Job: Build'. Each stage also has a 'JOBS' section with a 'View logs and history' link and a 'LAST EXECUTION RESULT' section with 'No results'.

- Click the application URL to launch the application. There is no home page set, but modify the URL by adding /fizzbuzz\_range/1/50 to request the range from 1 to 50.

```
fizzbuzz-w3.eu-gb.mybluemix.net/fizzbuzz_range/1/50
{
  "from": "1", "to": "50", "result":
  [
    "1", "2", "Fizz", "4", "Buzz", "Fizz", "7", "8", "Fizz", "Buzz", "11", "Fizz", "13", "14",
    "FizzBuzz", "16", "17", "Fizz", "19", "Buzz", "Fizz", "22", "23", "Fizz", "Buzz", "26",
    "Fizz", "28", "29", "FizzBuzz", "31", "32", "Fizz", "34", "Buzz", "Fizz", "37", "38",
    "Fizz", "Buzz", "41", "Fizz", "43", "44", "FizzBuzz", "46", "47", "Fizz", "49", "Buzz"
  ]
}
```

### 3. Are we really complete?

- There is a bug in the code we just delivered. Part of the skill of testing is to know what to test and in this exercise a common Javascript error was not tested for.
- Test for the range 9 to 50 - you should get no results!
  - This is due to the difference between comparing numbers and comparing strings:
    - 9 < 50 is true for numbers
    - "9" < "50" is false for strings
  - As part of our tests we should ensure the correct type of comparison is taking place
- It turns out some of our tests are not passing the same parameter types as the REST API call. The test is passing a number, but the API call is passing a string. We should rewrite the tests for convertRangeToFizzBuzz passing strings and also use a range that would fail a string comparison but pass a number comparison:

```
describe("convertRangeToFizzBuzz()", function() {
  it("returns in correct order", function() {
    expect(f.convertRangeToFizzBuzz("1",
"3")).toBe.eql(["1", "2", "Fizz"]);
  });

  it("applies FizzBuzz to every number in the range",
function() {
    var spy = sinon.spy(f, "convertToFizzBuzz");

    f.convertRangeToFizzBuzz("9", "50");

    for (var i = 9; i <= 50; i++) {

expect(spy.withArgs(i).calledOnce).toBe.eql(true, "Expected
convertToFizzBuzz to be called with " + i);
    }
    f.convertToFizzBuzz.restore();
  });
});
```

- The 'applies FizzBuzz to every number in the range test' is now failing. To fix it we need to ensure the convertToFizzBuzz function is using numbers not strings:

```
FizzBuzz.prototype.convertRangeToFizzBuzz = function(start, end)
{
  var result = [];
  var from = parseInt(start);
  var to = parseInt(end);
```

```
    for (var i = from; i <= to; i++) {  
        result.push(this.convertToFizzBuzz(i));  
    }  
  
    return result;  
};
```

- Using the 'Git Staging' view, stage, comment and Commit and Push the changes - quickly switch to the IBM Bluemix DevOps Services console for your project in the Build and Deploy settings - you should see a build, test and deploy automatically run