Getting Started with IBM Bluemix Hands-On Workshop

Module 5: Maximizing the Value of Bluemix

Table of Contents

Workshop overview

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

You'll 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 word *FizzBuzz*.

For example, for the range 1 .. 20, the response is:

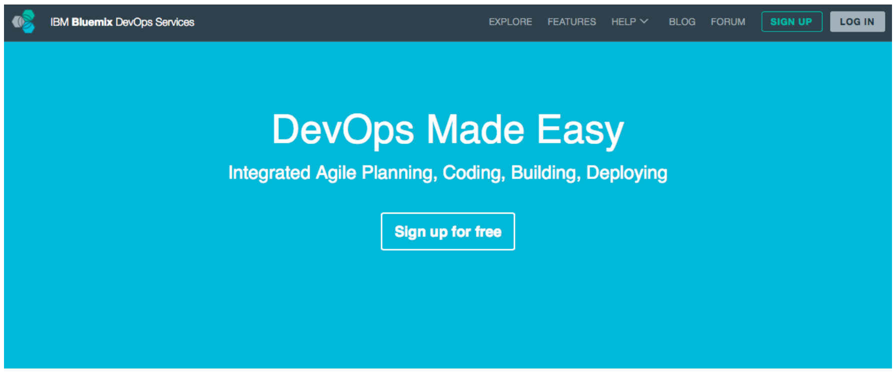
```
1, 2, Fizz, 4, Buzz, Fizz, 7, 8, Fizz, Buzz, 11, Fizz, 13, 14,
FizzBuzz, 16, 17, Fizz, 19, Buzz
```

You will use Node.js as the runtime and use test-driven development practices to create the solution.

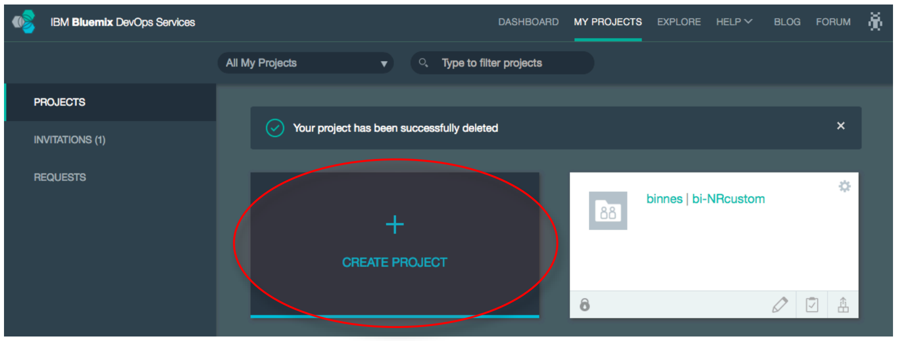
Exercise 5a: Creating the project source code repository

Before you write any code, you must have the correct tooling and a source code repository. For this exercise, you'll use the Git service from IBM DevOps Services for Bluemix.

1. Log in to DevOps Services for Bluemix: http://hub.jazz.net.

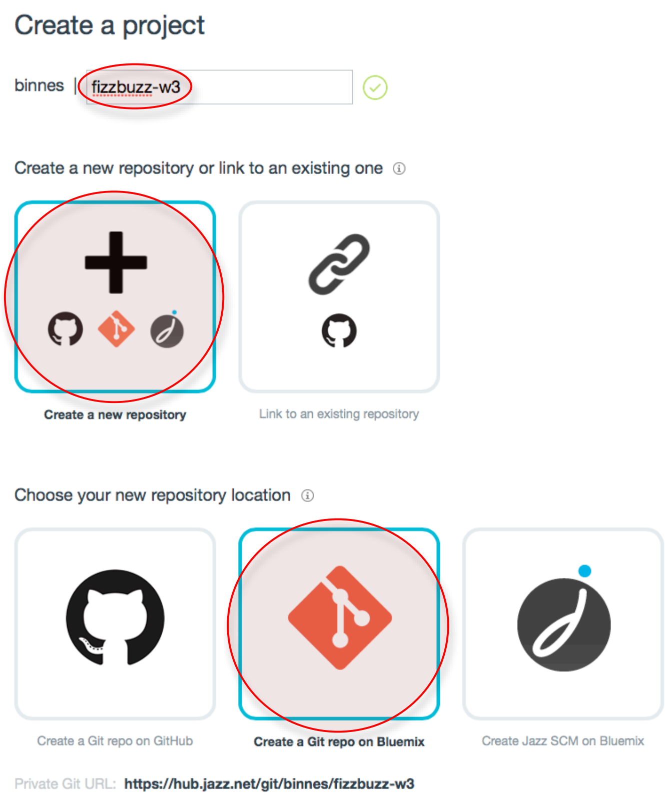

IBM Bluemix DevOps Services
EXPLORE
FEATURES
HELP
BLOG
FORUM
SIGN UP
LOG IN
DevOps Made Easy
Integrated Agile Planning, Coding, Building, Deploying
Sign up for free

2. From your MyProjects page, click **CREATE PROJECT**.



Your project has been successfully deleted

3. Create a Git repository on Bluemix:
   a. Enter a project name.
   b. Create a new repository.
   c. Create a Git repo on Bluemix.


Create a project

binnes | fizzbuzz-w3

Create a new repository or link to an existing one

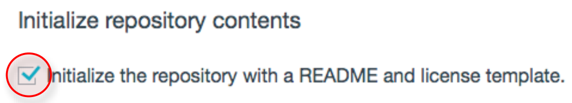
Create a new repository
Link to an existing repository

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. Select to initialize the repository

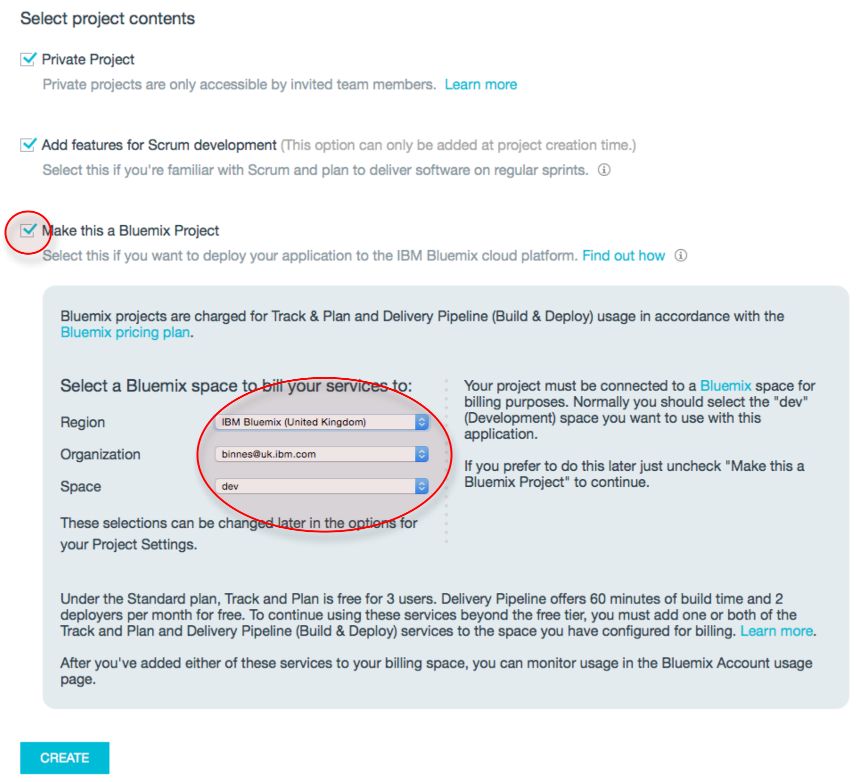

Initialize repository contents

✓ Initialize the repository with a README and license template.

5. If you want to share your project and make it searchable to the public, do not select the **Private Project** checkbox.

6. If you want to use the Track and Plan tool to support agile methodology, add features for Scrum development.

7. Select the checkbox to make the project a Bluemix project. Then, provide the details of the Bluemix space that the application will be deployed to.



Select project contents

✓ Private Project
Private 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 Project
Select 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	IBM Bluemix (United Kingdom)
Organization	binnes@uk.ibm.com
Space	dev

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.

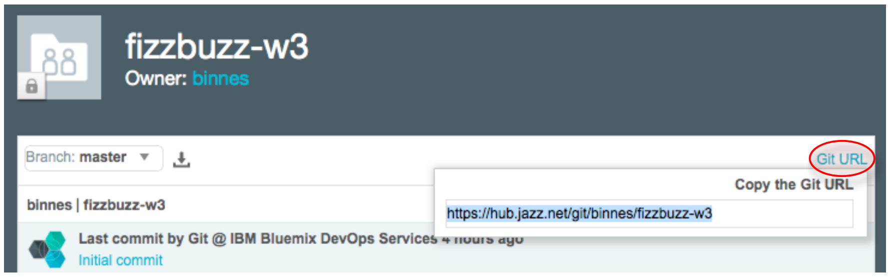
Under the Standard plan, Track and Plan is free for 3 users. Delivery Pipeline offers 60 minutes of build time and 2 deployers per month for free. To continue using these services beyond the free tier, you must add one or both of the Track and Plan and Delivery Pipeline (Build & Deploy) services to the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

8. Click **CREATE** to create the project.

The code repository has been created, and you can create a new Eclipse project that is based on the repository.

9. In DevOps Services console, select the Git URL link and copy the URL. Use Ctrl+C or Cmd C.



fizzbuzz-w3
Owner: binnes

Branch: master

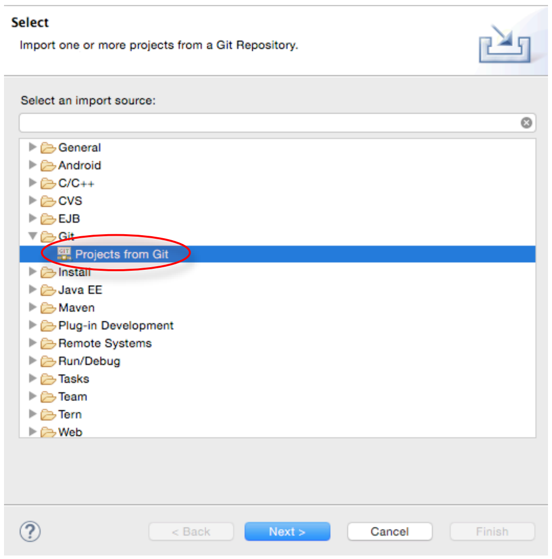
binnes | fizzbuzz-w3

Last commit by Git @ IBM Bluemix DevOps Services 4 hours ago
Initial commit

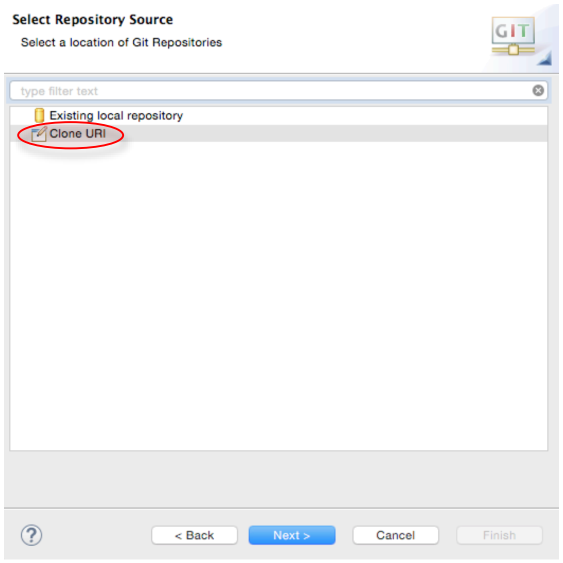
Git URL
Copy the Git URL
https://hub.jazz.net/git/binnes/fizzbuzz-w3

10. Start Eclipse and then from the main menu, click **File > Import**.

11. From the Select dialog, click **Git > Project from Git** and then click **Next.**



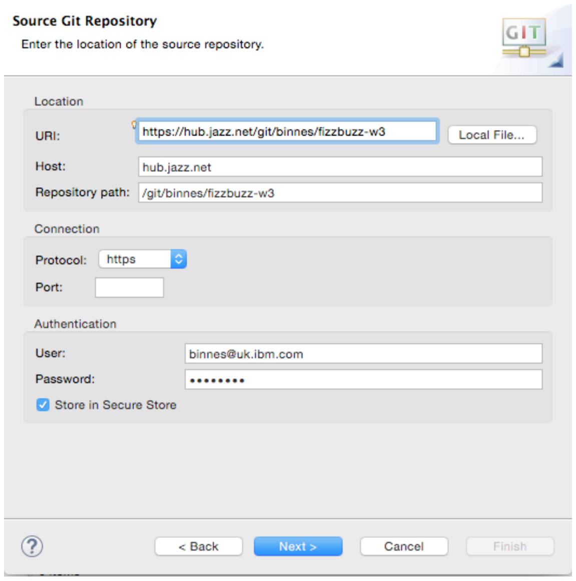
12. In the Git dialog, select Clone URI and then click **Next**.



13. In the next dialog, paste the Git repository URL into the **URI Location** field. This will automatically populate the Host and Repository path.

14. Enter your DevOps services user name and password. Optionally, select to store the Git credentials in a secure store in Eclipse. You will need to provide additional information to initialize the secure store if this is the first time that you store credentials.

15. Click **Next**.


Source Git Repository

Enter the location of the source repository.

Location

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

Host: hub.jazz.net

Repository path: /git/binnes/fizzbuzz-w3

Connection

Protocol: https

Port:

Authentication

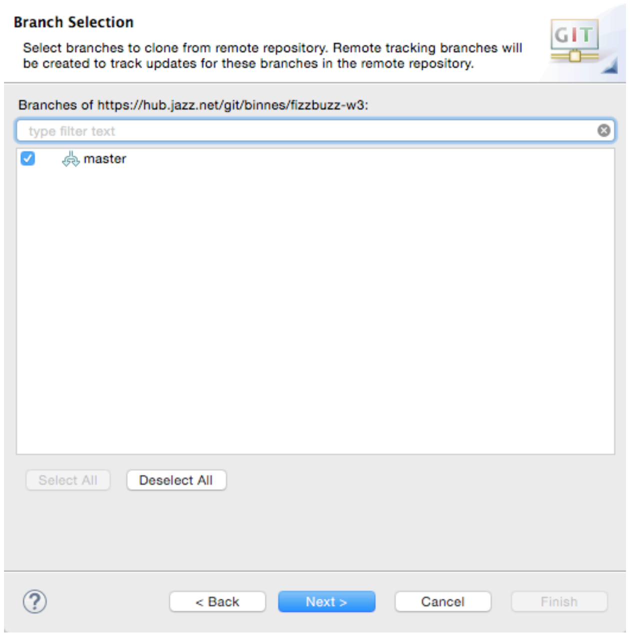
User: binnes@uk.ibm.com

Password: ••••••••

☑ Store in Secure Store

Eclipse will query the Git repository and list the next branch that's available. Because you just created the repository, only the master branch exists.

16. Leave the master branch selected and click **Next**.


Branch Selection

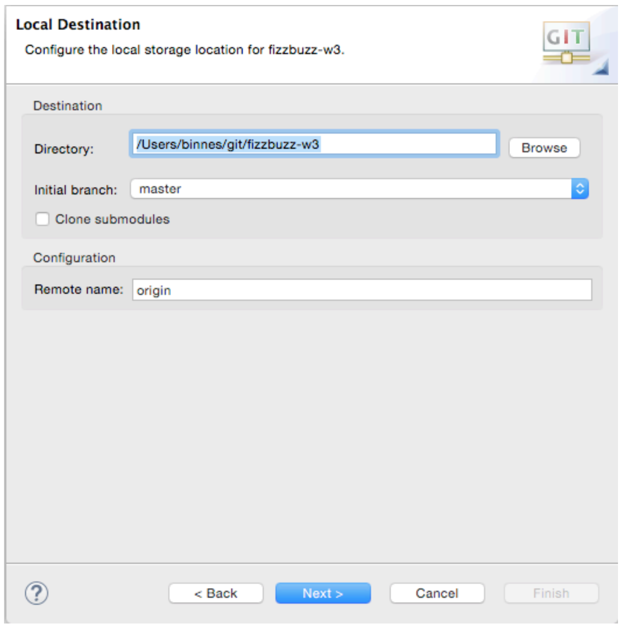
Select branches to clone from remote repository. Remote tracking branches will be created to track updates for these branches in the remote repository.

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

☑ master

The local destination is then chosen. The default location is `<user home directory>/git/<project name>`. Use suggested default location. If you want to change it, you must remember the location because you will need it later in the exercise.

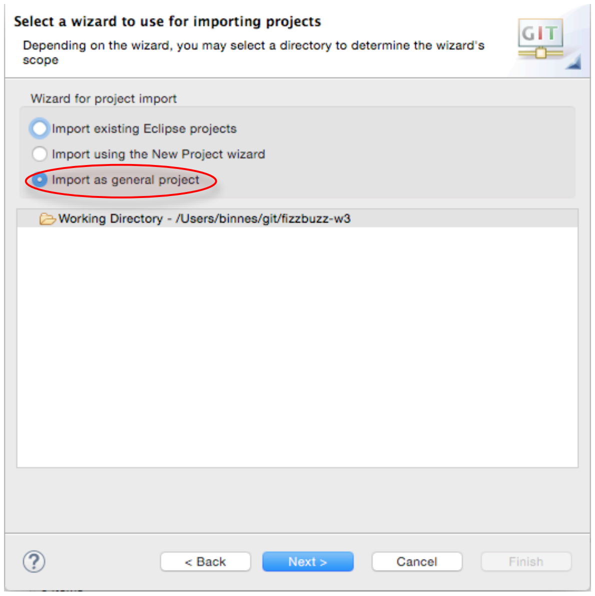
17. Click **Next**.


Local Destination
Configure the local storage location for fizzbuzz-w3.

Destination
Directory: /Users/binnes/git/fizzbuzz-w3
Initial branch: master
Clone submodules

Configuration
Remote name: origin

18. Select the wizard to import as a general project and then click **Next**.

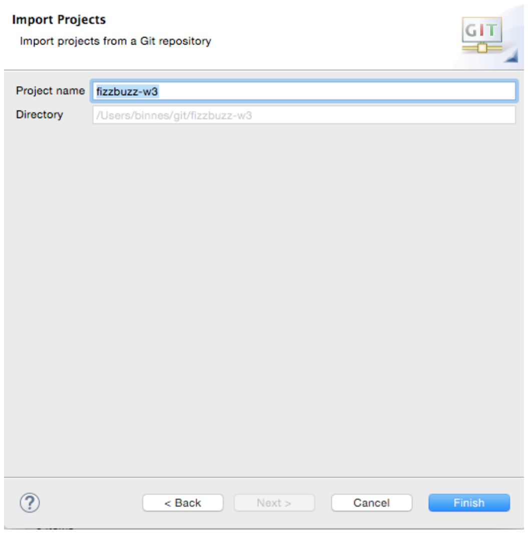

Select a wizard to use for importing projects
Depending on the wizard, you may select a directory to determine the wizard's scope

Wizard for project import
Import existing Eclipse projects
Import using the New Project wizard
[✓ Import as general project]

Working Directory - /Users/binnes/git/fizzbuzz-w3

19. In the Import Projects dialog, enter an Eclipse project name. Use the same name as the DevOps service project name, but you can change if needed.

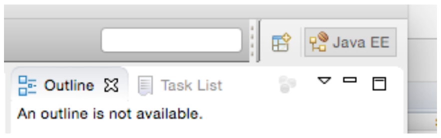
20. Click **Finish**.



Project name: fizzbuzz-w3
Directory: /Users/binnes/git/fizzbuzz-w3

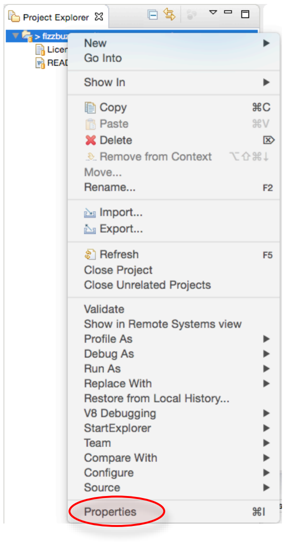
You now have an Eclipse project linked to your DevOps services Git repository. To complete the project setup, configure Eclipse to support JavaScript development.

21. Switch Eclipse to the JavaScript perspective:
    a. Click **Add Perspective** at the top right of the Eclipse window.

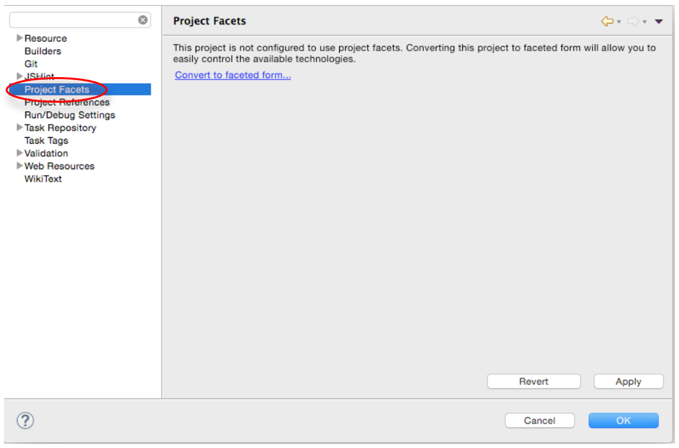


    b. Select **JavaScript** from the list.

22. Enable the NodeJS facet on the project:
    a. Right-click the project name and click **Properties**.



    b. In the Project Facet dialog, click **Project Facets** and then click **Convert to faceted form**.

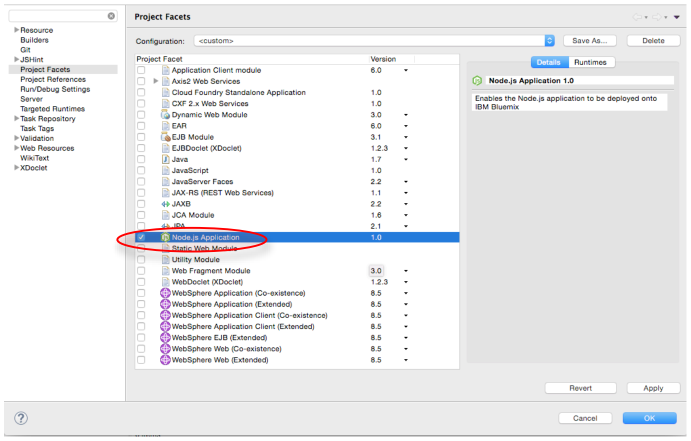


Project Facets

This project is not configured to use project facets. Converting this project to faceted form will allow you to easily control the available technologies.

Convert to faceted form...

c. When you see the list of facets, click **Node.js Application**. Click **OK**.



Project Facets

Configuration: <custom>

	Project Facet	Version
☐	Application Client module	6.0
☐	Axis2 Web Services	
☐	Cloud Foundry Standalone Application	1.0
☐	CXF 2.x Web Services	1.0
☐	Dynamic Web Module	3.0
☐	EAR	6.0
☐	EJB Module	3.1
☐	EJBDoclet (XDoclet)	1.2.3
☐	Java	1.7
☐	JavaScript	1.0
☐	JavaServer Faces	2.2
☐	JAX-RS (REST Web Services)	1.1
☐	JAXB	2.2
☐	JCA Module	1.6
☐	JPA	2.1
✓	Node.js Application	1.0
☐	Static Web Module	
☐	Utility Module	
☐	Web Fragment Module	3.0
☐	WebDoclet (XDoclet)	1.2.3
☐	WebSphere Application (Co-existence)	8.5
☐	WebSphere Application (Extended)	8.5
☐	WebSphere Application Client (Co-existence)	8.5
☐	WebSphere Application Client (Extended)	8.5
☐	WebSphere EJB (Extended)	8.5
☐	WebSphere Web (Co-existence)	8.5
☐	WebSphere Web (Extended)	8.5

Node.js Application 1.0

Enables the Node.js application to be deployed onto IBM Bluemix

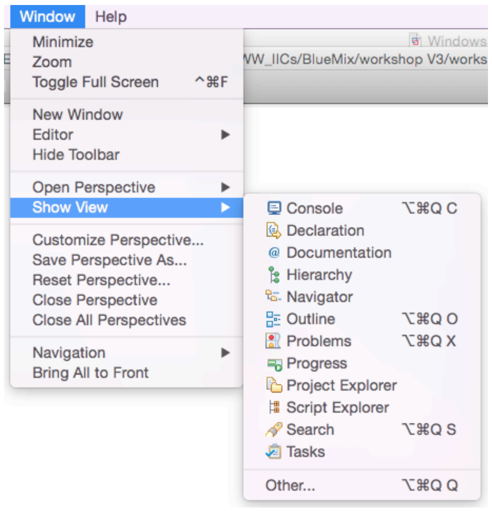
Adding the facet automatically creates a sample application in the project.

d. Remove the `app.js` and `package.json` files. Right-click each file, click **Delete**, and confirm the deletion.

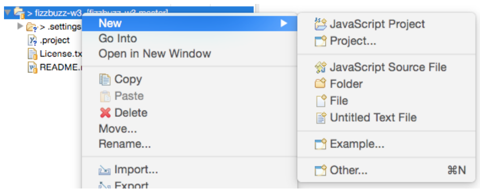
The project shows that there are changes to the local copy of the project. These are the Eclipse project setting files. Don't save these as part of the source code.

23. Add a navigator view to see all the files in the project:

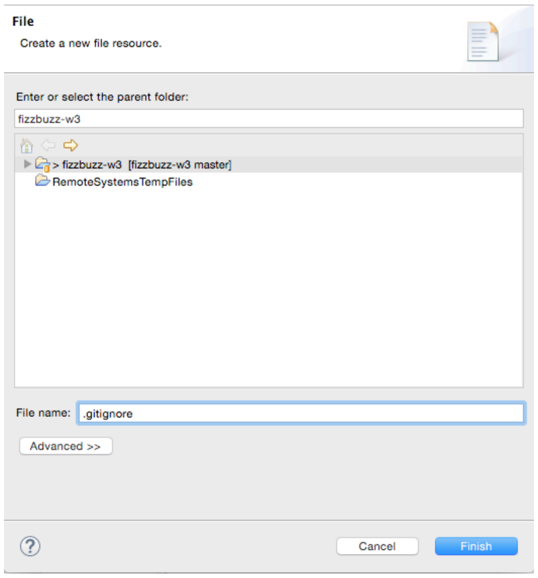
a. Click **Window > Show View** and then click **Navigator**.



b. Right-click the project name in the Navigator view and click **New > File**.



c. Name the file `.gitignore` and then click **Finish**.


File

Create a new file resource.

Enter or select the parent folder:

fizzbuzz-w3

> fizzbuzz-w3 [fizzbuzz-w3 master]

RemoteSystemsTempFiles

File name: .gitignore

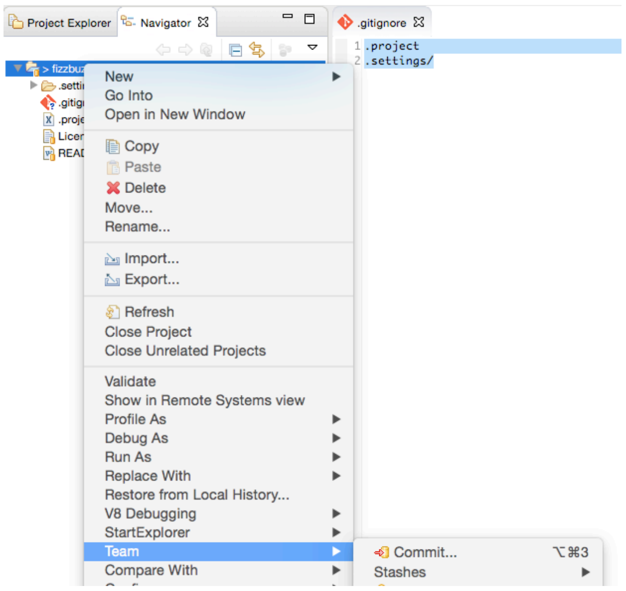
d. Add the following two lines to the `.gitignore` file:

Windows: Leave the slash as specified; do not change it to `.settings\`:

```
.project
.settings/
```

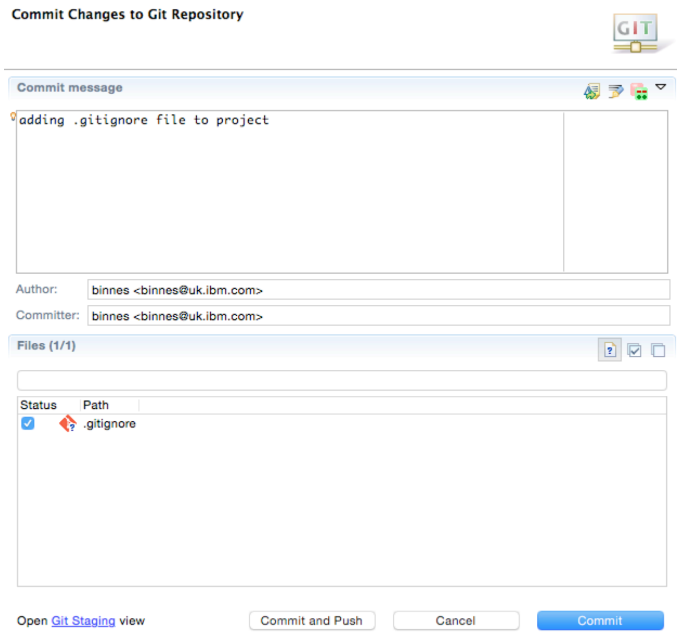
The project still shows that there are changes. This occurs because the `.gitignore` file is a new file that is not committed and pushed to the master branch on the Git server.

24. Commit and push the `.gitignore` file:
    a. Right-click the project name and click **Team > Commit**.



```
.project
.settings/
```

    b. In the Commit Changes dialog, enter a commit message and select the `.gitignore` file. Then, click **Commit and Push**.



Commit Changes to Git Repository

Commit message

```
adding .gitignore file to project
```

Author: binnes <binnes@uk.ibm.com>

Committer: binnes <binnes@uk.ibm.com>

Files (1/1)

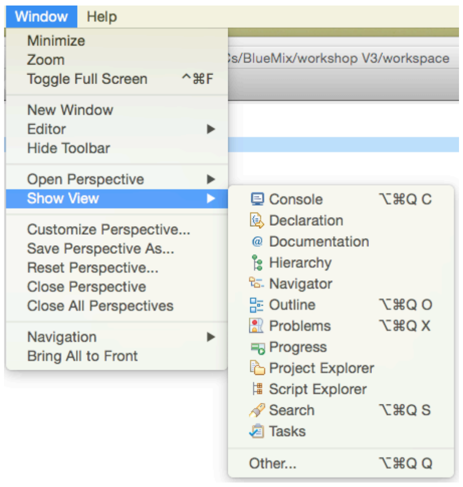
Status	Path
✓	.gitignore

    c. Close the confirmation dialog by clicking **OK**.

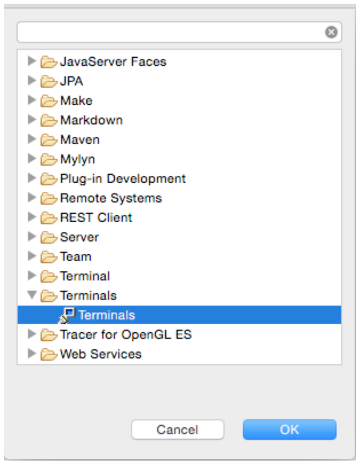
The project will no longer show that there are outstanding changes.

Finally, you should add a new window to access the command line. However, this is not necessary. Instead, you can use a command window outside Eclipse, but it's more convenient to have everything in Eclipse because it provides a better working environment.

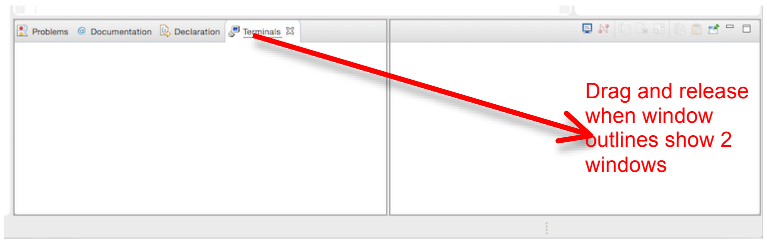
25. Create a terminals view:
    a. From the main menu, click **Window > Show View** and then click **Other**.



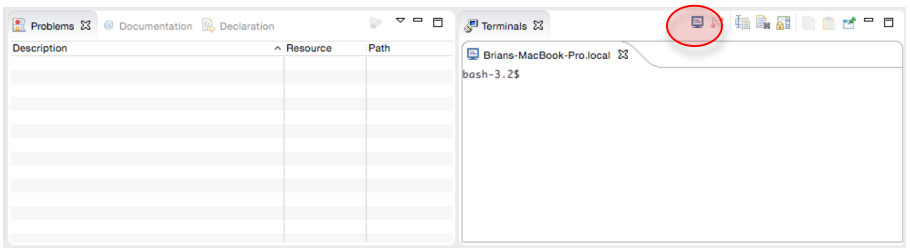
    b. In the dialog, click **Terminals > Terminals** and then click **OK**.



    c. In the Terminals window, drag the tab to split the bottom section of the screen to allow concurrent viewing of the terminal and the problem views.



If a command prompt isn't shown, click the **Open Terminal** icon to start a new terminal. Use the default setting. On some platforms, this happens automatically.


Problems
Documentation
Declaration
Description
Resource
Path
Terminals
Brians-MacBook-Pro.local

```
bash-3.2$
```

Exercise 5b: Setting up development tooling

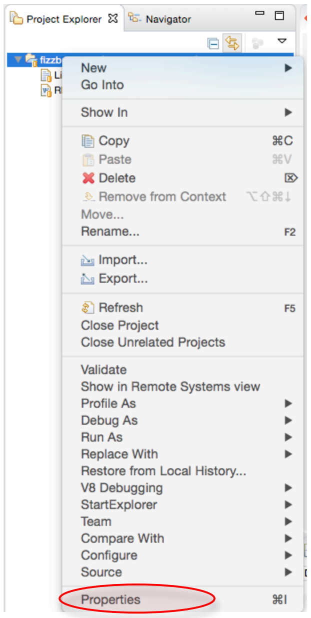
Using appropriate developer tooling with features to automatically check for errors, provide code assist, and so on can make you more productive and help identify possible problems.

You'll use the new project that you created in the previous section.

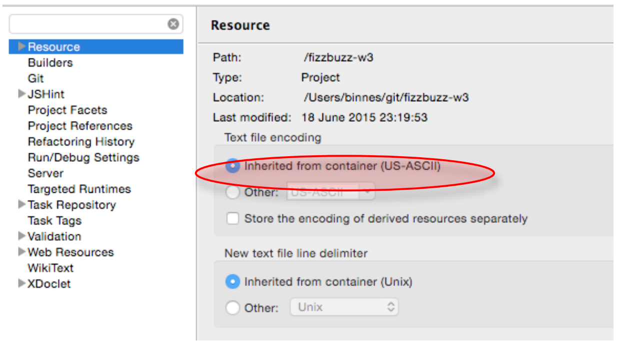
The first step is to ensure the JavaScript tooling is enabled and the test framework is created.

1. Create the JavaScript project:
   a. In the Terminals view, change to the directory that contains the project:
      - **Mac & Linux 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 and then click **Properties**.



      Note the project location on the **Resources** tab:

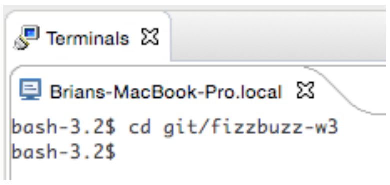


Resource

Path: /fizzbuzz-w3
Type: Project
Location: /Users/binnes/git/fizzbuzz-w3
Last modified: 18 June 2015 23:19:53

Text file encoding
[✓ Inherited from container (US-ASCII)]
Other: US-ASCII
Store the encoding of derived resources separately

New text file line delimiter
[✓ Inherited from container (Unix)]
Other: Unix

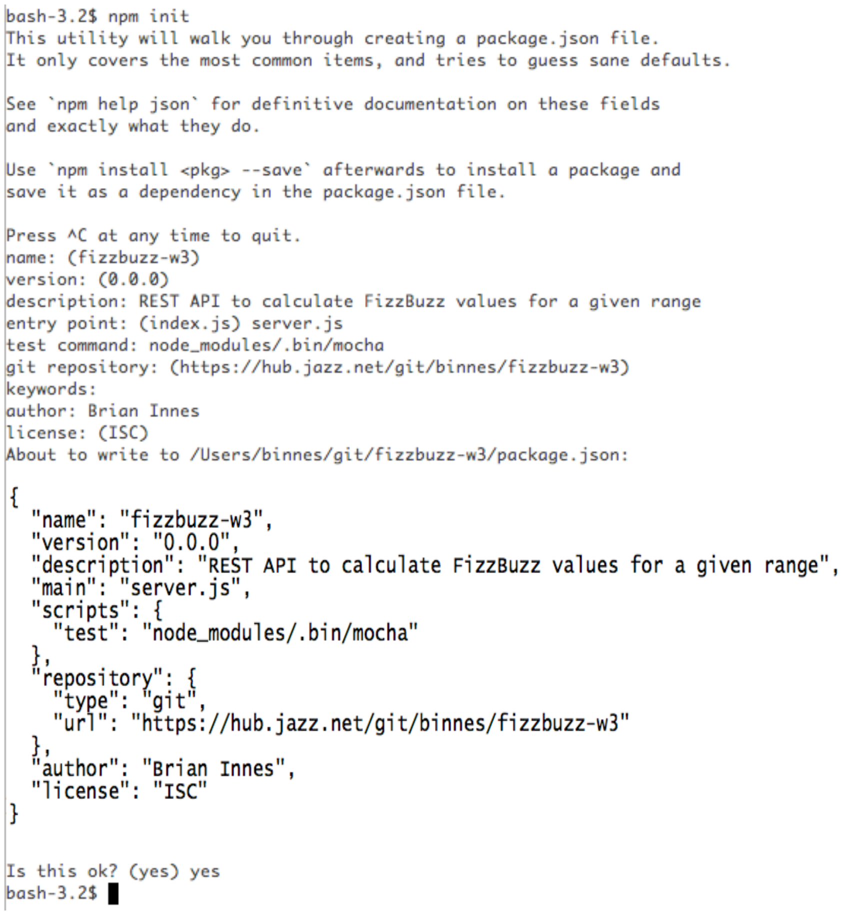


```
bash-3.2$ cd git/fizzbuzz-w3
bash-3.2$
```

b. In the Terminals window, run the **`npm init`** command and answer the prompts:
   - Name - provide a unique name for your project
   - Version - `0.0.0`
   - Description - `REST API to calculate FizzBuzz value for a given range`
   - Entry point - `server.js`
   - Test command - `node_modules/.bin/mocha`
   - Git repository - hit enter to accept the default value
   - Keywords - can leave blank
   - Author - enter your name
   - License - hit enter to accept the default value

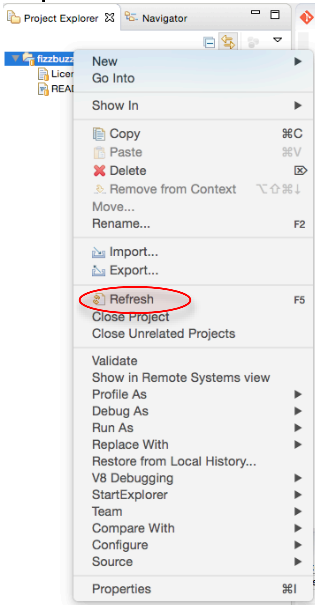
   Windows: Use the forward slash (/) when specifying the test command.

   If the Terminals window is too small, double-click the tab to enlarge it to a 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$
```

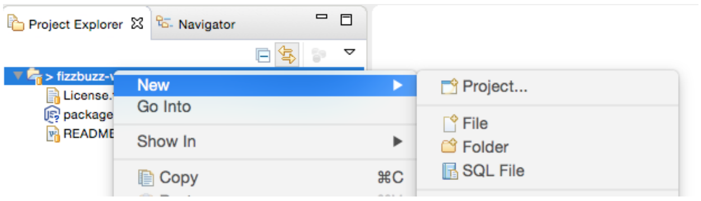
c. To move the created files into Eclipse, right-click the project name in the Project Explorer view and click **Refresh** or press F5.



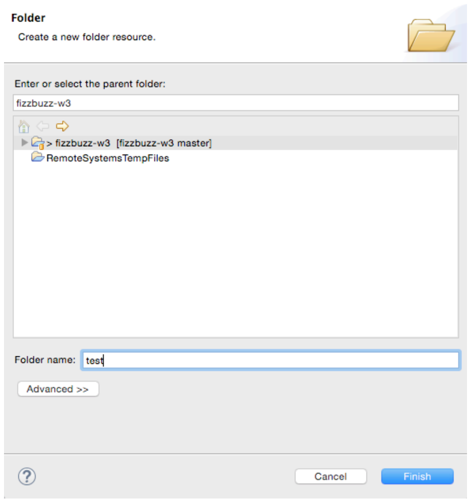
d. If the wizard truncated the test script to `mocha`, open the `package.json` file in the Eclipse editor and change it back to `node_modules/.bin/mocha`.

e. Tests are expected to be in a directory named `test`, so create a test directory:

i. Right-click the project name in the Project Explorer view and click **New > Folder.**



ii. Name the folder `test` and then click **Finish**.


Folder

Create a new folder resource.

Enter or select the parent folder:

fizzbuzz-w3

> fizzbuzz-w3 [fizzbuzz-w3 master]

RemoteSystemsTempFiles

Folder name: test

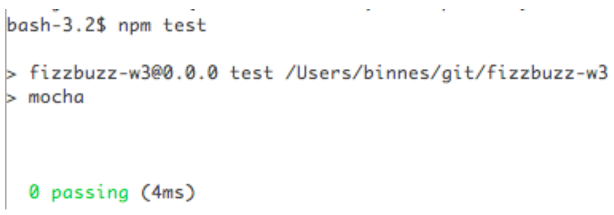
f. To be able to run tests, install the Mocha framework:

i. In the Terminals view, enter `npm install mocha --save-dev`

ii. Select the project name in the Project Explorer and click **Refresh** the project in Eclipse to display the new files.

g. Test that everything works as expected by running a test:

i. 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)
```

Note that the test framework ran even though there are no tests defined.

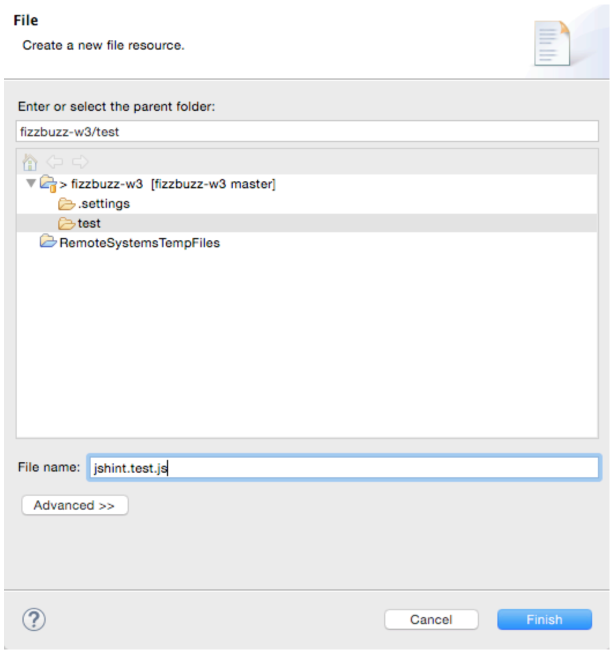
JSHint is a lint program for JavaScript. It provides static analysis of code. It’s a good practice to have this running in your tooling. Many development teams also include it as part of their automated test suite.

h. Run JSHint from Mocha:

i. In the Terminals view, enter the following command:

```
npm install mocha-jshint --save-dev
```

ii. Select the test directory of the project in the Project Explorer view and then right-click and click **New > File**. Name the file `jshint.test.js`.



File

Create a new file resource.

Enter or select the parent folder:

fizzbuzz-w3/test

File name: jshint.test.js

iii. In the file, enter the following content and then save the file by clicking **File > Save** or use these commands:

Windows and Linux: Ctrl+S

Mac: Cmd+S:

```
require('mocha-jshint')();
```

iv. Rerun the tests by entering the following command in the Terminals view:

```
npm test
```

There are many errors from the imported modules in the `node_modules` directory. You do not control this is code, so it should be excluded from tests.

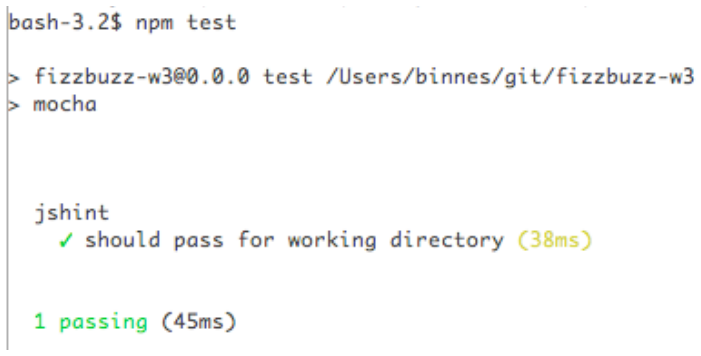
i. Configure JSHint to ignore the files by using a `.jshintignore` file:

  i. Create a `.jshintignore` file by right-clicking the project name in the Project Explorer view and clicking **New > File**.

  ii. Add the following content to the file:

```
node_modules
test
```

  iii. In the Terminals view, rerun the tests: `npm test`



```
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 from the problems.

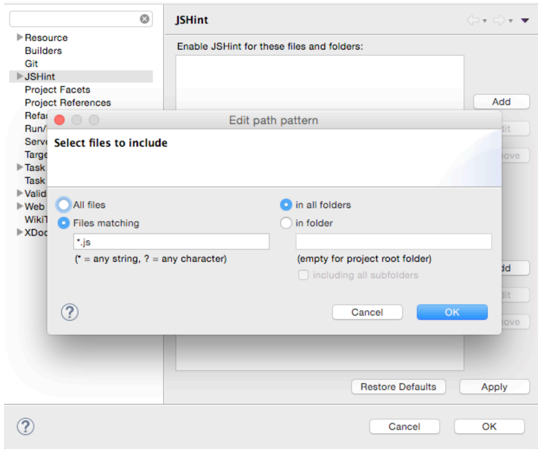
JSHint is included in Eclipse, but you must configure it.

j. Enable JSHint in Eclipse:

  i. Right-click the project name in the Project Explorer view and click **Properties**.

  ii. In the dialog, click **JSHint**.

iii. Click **Add** next to the **Enable JSHint for these files and folders** box.



Enable JSHint for these files and folders:

Select files to include

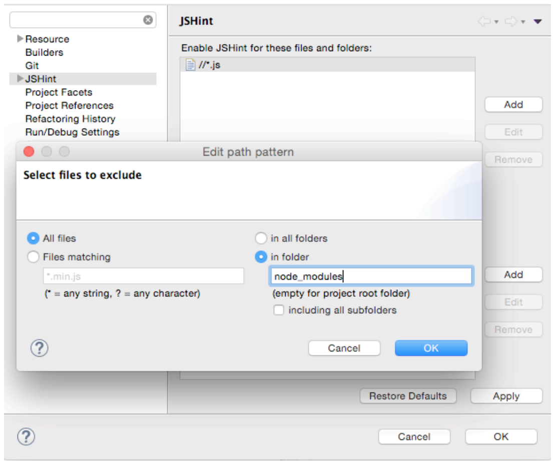
All files · Files matching: *.js · in all folders · in folder: (empty for project root folder) · Including all subfolders

(* = any string, ? = any character)

iv. Accept the defaults as shown above and click **OK**.

v. Click **Add** next to the **But exclude these files and folders from validation** box.

vi. Click **All files** and then select **in folder** and add the node_modules folder. Select the **include all subfolders** checkbox.



Enable JSHint for these files and folders:

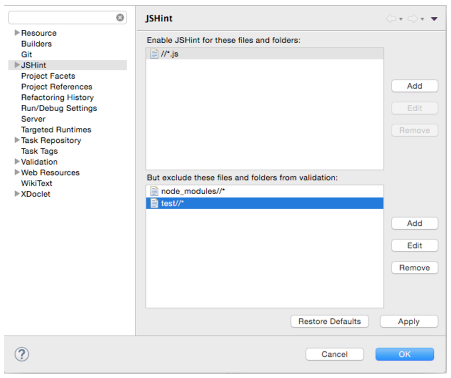
//*.js

Select files to exclude

All files · Files matching: *.min.js · in all folders · in folder: node_modules · (empty for project root folder) · including all subfolders

(* = any string, ? = any character)

vii. Repeat the previous step to exclude the content of the test folder.



Enable JSHint for these files and folders:

//*.js

But exclude these files and folders from validation:

node_modules//*

test//*

viii. Click **OK** to make the changes and close the dialog.

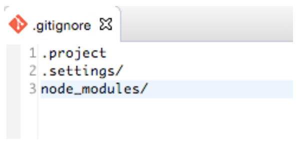
The project is now ready to implement the REST API, so now is a good time to commit changes to the Git repository (repo).

There is some debate about whether dependencies should be checked in or re-fetched at build time. To reduce the amount of traffic that you need to push to the Git repo for this exercise, you will exclude the dependencies in `node_modules`.

k. Update the Git configuration and then commit and push changes to the Git repo:

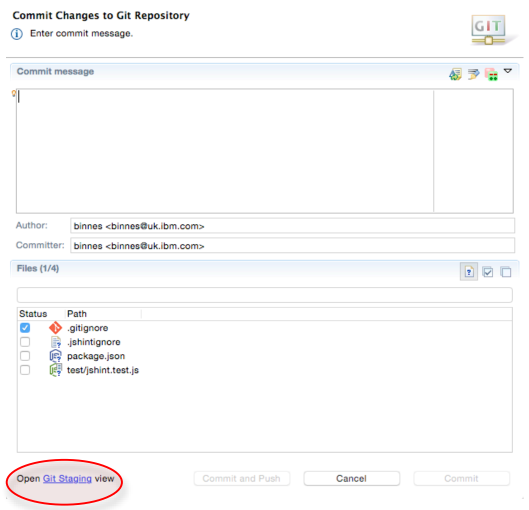
I. Switch to the Navigator view to see files that start with a period (.)

II. Double-click the `.gitignore` file to edit it and add the `node_modules` directory to the file. Then, save the file: Ctrl+S or Cmd+S.



```
.project
.settings/
node_modules/
```

III. Right-click the project name and click **Team > Commit**. Click the link at the bottom right of the dialog to open the Git Staging view.



Commit Changes to Git Repository

Enter commit message.

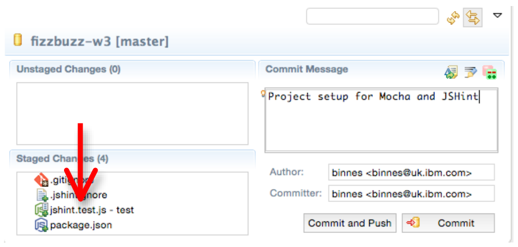
Author: binnes <binnes@uk.ibm.com>

Committer: binnes <binnes@uk.ibm.com>

Files (1/4)

Status	Path
✓	.gitignore
	.jshintignore
	package.json
	test/jshint.test.js

IV. Drag the 4 files from the Unstaged Changes window to the Staged Changes window. Then, enter a commit message.



fizzbuzz-w3 [master]

Project setup for Mocha and JSHint

Author: binnes <binnes@uk.ibm.com>

Committer: binnes <binnes@uk.ibm.com>

V. Click **Commit and Push**.

VI. Verify that the commit completed successfully and then close the confirmation dialog.

Exercise 5c: Test-driven development

Implement the divisibleBy function

To implement the divisibleBy funtion, you generate the first test that the implementation must pass. Solving the problem requires a function to discover whether one number is divisible by another number, so the first test will check whether 3 is divisible by 3.

Before you write the test, you'll add another test framework: Chai. Chai is a behavior-driven development (BDD) and test-driven development (TDD) assertion library for Node.js and a browser that can be paired with any JavaScript testing framework. Chai provides the test capability to say "I expect this to be true" or "this should be true."

Before you write a test, install Chai and then configure Mocha to use Chai:

1. Install and configure Chai:
   a. In the Terminals view, enter the following command:

      ```
      npm install chai --save-dev
      ```

   b. Select the project name in Project Explorer and press F5 to refresh the Eclipse project content.

   c. Create a new file in the test folder named `support.js` and add the following content:

      ```
      var chai = require("chai");
      global.expect = chai.expect;
      ```

   d. Create a new file in the test folder named `mocha.opts` and add the following content:

      ```
      --require test/support
      ```

   e. Save both files.

      The files configure Mocha to use the *expect* functionality from Chai.

2. Create the first test:
   a. Create a new file in the test folder named `fizzbuzz.test.js` by right-clicking the test folder in the Project Explorer view and then click **New > File.**

b. Enter the following code to get access to the code that you 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);
    });
  });
});
```

The first test is now complete, so you can write the code that's needed to pass the test.

Rather than having to manually run the tests, you can specify an option to continually run Mocha. Therefore, every time that a file is changed, the tests are automatically run. Use the `-w` command-line option to run tests continually.

3. Configure Mocha to continually run tests:
   a. In the Terminals view, run the following command:
      - **Mac and Linux**: `node_modules/.bin/mocha -w`
      - **Windows**: `node_modules\.bin\mocha -w`

      You will see that the tests are failing because of an error:
      Error: Cannot find module '../fizzbuzz.js'

   b. In the root of the project, create the file `fizzbuzz.js`.

   c. Add the following code to configure the `fizzbuzz.js` file:

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

module.exports = FizzBuzz;
```

   d. 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)
```

Now, you can implement the divisibleBy function, but you should write only the minimal amount of code to pass the written test. This could be as simple as returning true because 3 is divisible by 3.

4. Implement the divisibleBy function to pass the first test:
   a. Add the following code to the `fizzbuzz.js` file above the line that starts with “`module.exports`”:

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

   b. 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 needed to test when the divisibleBy function should return false. Having the additional test will also require the correct implementation of the divisibleBy function.

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

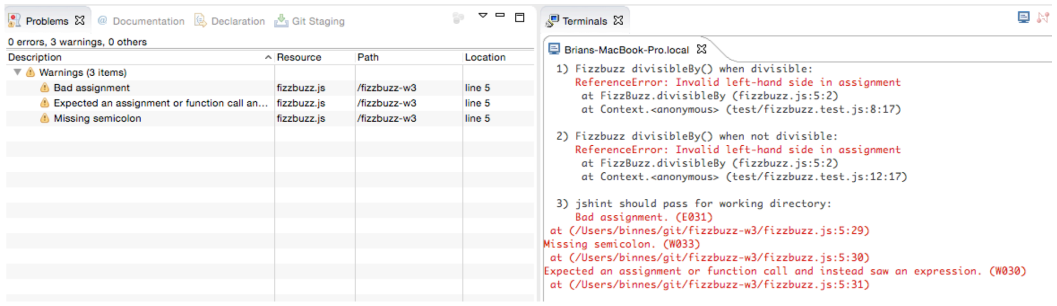
```
  describe("divisibleBy()", function() {
    it("when divisible", function() {
      expect(f.divisibleBy(3, 3)).to.be.eql(true);
    });

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

   b. Save the file. Now, you should have a failing test again.

   c. Implement the divideBy function with the following code. There is a deliberate coding error, so copy the code below as shown:

```
FizzBuzz.prototype.divisibleBy = function(number, divisor) {
      return number % divisor = 0;
};
```

Three errors are reported. You’ve broken the divideBy test for 3 divided by 3: the new test doesn't pass, and you also have JSHint complaining of bad JavaScript code. Notice that Eclipse is also reporting the JSHint issues in the problem panel.



0 errors, 3 warnings, 0 others

Description	Resource	Path	Location
Warnings (3 items)			
Bad assignment	fizzbuzz.js	/fizzbuzz-w3	line 5
Expected an assignment or function call an…	fizzbuzz.js	/fizzbuzz-w3	line 5
Missing semicolon	fizzbuzz.js	/fizzbuzz-w3	line 5

```
  1) Fizzbuzz divisibleBy() when divisible:
     ReferenceError: Invalid left-hand side in assignment
      at FizzBuzz.divisibleBy (fizzbuzz.js:5:2)
      at Context.<anonymous> (test/fizzbuzz.test.js:8:17)

  2) Fizzbuzz divisibleBy() when not divisible:
     ReferenceError: Invalid left-hand side in assignment
      at FizzBuzz.divisibleBy (fizzbuzz.js:5:2)
      at Context.<anonymous> (test/fizzbuzz.test.js:12:17)

  3) jshint should pass for working directory:
     Bad assignment. (E031)
 at (/Users/binnes/git/fizzbuzz-w3/fizzbuzz.js:5:29)
Missing semicolon. (W033)
 at (/Users/binnes/git/fizzbuzz-w3/fizzbuzz.js:5:30)
Expected an assignment or function call and instead saw an expression. (W030)
 at (/Users/binnes/git/fizzbuzz-w3/fizzbuzz.js:5:31)
```

d. Replace the code with the following new code. There is still a deliberate mistake.

```
FizzBuzz.prototype.divisibleBy = function(number, divisor) {
      return number % divisor == 0;
};
```

The functional tests pass, but JSHint is still complaining of bad JavaScript code. This is because in JavaScript there are two comparison operators:

- == does type coercion
- === does *not* do type coercion

Type coercion occurs when a variable is automatically converted to a different type when required, such as changing a number to a string.

e. When you use a static number, use the === operator. Replace the code again to use this code:

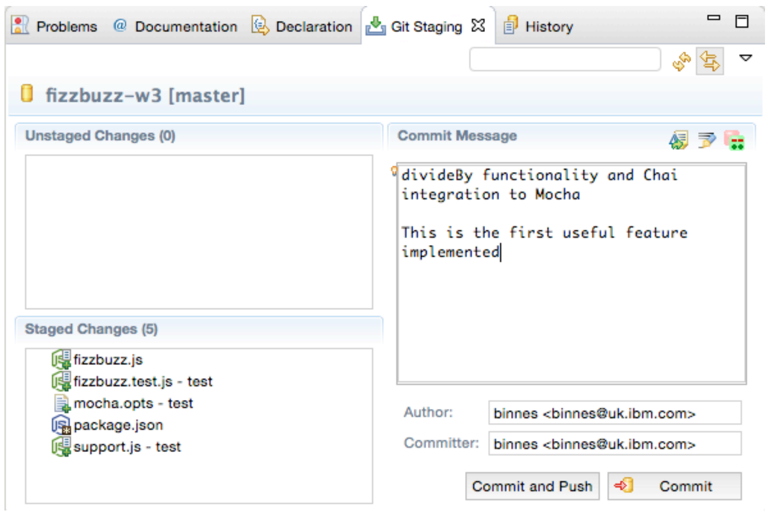
```
FizzBuzz.prototype.divisibleBy = function(number, divisor) {
      return number % divisor === 0;
};
```

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:
   a. In the Git Staging view, select the files in the Unstaged Changes window and drag to the Staged Changes window. Add a comment and then click **Commit and Push**.

      Tip: It’s better to split the configuration of Chai and Mocha into separate commits.



```
divideBy functionality and Chai
integration to Mocha

This is the first useful feature
implemented
```

fizzbuzz.js
fizzbuzz.test.js - test
mocha.opts - test
package.json
support.js - test

Author: binnes <binnes@uk.ibm.com>
Committer: binnes <binnes@uk.ibm.com>

Implement the convertToFizzBuzz function

Now that you have a function to check whether a number is divisible by 3 or 5, you must implement a function to convert a number to its FizzBuzz value.

1. Implement the first test for convertToFizzBuzzZ:
   a. In the `fizzbuzz.test.js` file, add the following 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);
  });
});

   describe("convertToFizzBuzz()", function() {
   });
```

   b. 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.eql("Fizz");
    });
  });
```

   c. In the `fizzbuzz.js` file, add a new function prototype for convertToFizzBuzz and add the code that is required to pass the first test:

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

   d. Add another test for Fizz to test that 6 also returns "Fizz" and then add tests to check that 5 and 10 return "Buzz" by updating the test code in the `fizzbuzz.test.js` file:

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

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

   e. Implement the functionality in the `fizzbuzz.js` file 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.

f. Add tests to check for “FizzBuzz” and a number when that number is 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.eql("Fizz");
    expect(f.convertToFizzBuzz(6)).to.be.eql("Fizz");
  });

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

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

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

  });
});
```

g. To get the tests to pass, complete the convertToFizzBuzz function. 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 the function is complete.

h. Commit the code. Use the Git Staging view to stage, add a comment, and then commit and push the changes to the master repo.

Implement convertRangeToFizzBuzz (introduces Sinon.JS)

The next stage to implementing FizzBuzz is to be able to convert a range of numbers, not just a single number. Ensure that the tests for this function do not repeat the tests for convertToFizzBuzz. You already have tests for that.

The tests need to:

- Verify that when a range is converted, the results are returned in the correct order
- Verify that for a range, you call the convertToFizzBuzz function once for each member of the array

To enable this type of testing, an additional testing capability is needed. You'll use Sinon.JS to provide this capability. Sinon.JS provides a library to help you unit test your code. It supports spies, stubs, and mocks. The library supports multiple browsers and can run on a server using Node.js.

1. Add Sinon.JS and configure the test framework to use it:
   a. In the Terminals view, enter Ctrl+C to stop the Mocha tests. On Windows, answer Y to terminate batch job.

   b. Enter the following command in the Terminals window:

   ```
   npm install sinon --save-dev
   npm install sinon-chai --save-dev
   ```

   c. Refresh the Eclipse project by selecting the project name in the Project Explorer view and pressing F5.

   d. In the Terminals view, restart the Mocha `-w` command.

   Tip: Use the up arrow to scroll through previous commands.

   Mac and Linux: `node_modules/.bin/mocha -w`
   Windows: `node_modules\.bin\mocha -w`

   e. Modify the `support.js` file in the `test` folder to enable Chai to use Sinon.JS:

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

   chai.use(sinonChai);
   global.expect = chai.expect;
   ```

2. Add the test for the convertRangeToFizzBuzz function:
   a. Add the Describe function below that tests for the convertToFizzBuzzz function:

   ```
     describe("convertRangeToFizzBuzz()", function() {
    });
   ```

   b. Add a test to ensure that 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"]);
      });

   });
```

c. Implement the function to satisfy the test by adding the following code:

```
FizzBuzz.prototype.convertRangeToFizzBuzz = function(start, end) {
  return ["1", "2", "Fizz"];
};
```

d. Add a test to ensure that 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, you use Sinon.JS 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.JS 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();
      });
    });
```

e. Implement the function to pass the tests by adding 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;
};
```

The implementation of FizzBuzz is now complete because the code passes all the tests.

f. Commit the code by using the Git Staging view to stage, provide a comment, and then commit and push the changes to the master repository.

Exercise 5d: Adding the REST API and deploying to Bluemix

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

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

1. Implement a Node.js server by using the Express.js framework:
   a. Create a new file named `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. Because the application requires the Express.js framework, install it and add it to the `packages.json` file. Use the command `npm install`:
   a. In the Terminals view, enter Ctrl+C to stop the Mocha tests.

   b. Enter the following command:

```
npm install express -save
```

   c. Refresh the Eclipse content by selecting the project name in Project Explorer view and then pressing F5.

3. Implement the REST API:
   a. 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)
  });
});
```

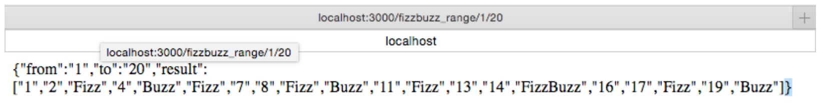
4. Test the API:
   a. Start the NodeJS server by entering the following command in the Terminals view:

   ```
   npm start
   ```

   You should see a response similar to this output:

   ```
   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
   ```

5. Test the API in a browser or command line:
   a. Enter `http://localhost:3000/fizzbuzz_range/1/20` into a browser. You should see the following text in the browser:


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

   Tip: Instead of testing the API with a browser, you can use the command-line tool cURL:

   ```
   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","Buz
   z"]}
   ```

   b. Enter Ctrl+C to terminate the server.

6. Modify code for Bluemix and Cloud Foundry:
   a. Before pushing the code to Bluemix, modify the `server.js` file to ensure that the server listens on the host name and port that is 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";
   ```

   b. In the project, create a new file named `manifest.yml`. Then, enter the following content and change the name so that it’s unique, such as adding your initials at the start of the name:

   ```
   applications:
   ```

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

c. Save the file.

d. In the project, create a new file named `.cfignore`. Then, enter the following content:

```
launchConfigurations/
.git/
node_modules/
npm-debug.log
```

e. Modify the `.gitignore` file so that it has the following content. Use the Navigator view to see files that start with a dot.

```
launchConfigurations/
.project
.settings/
node_modules/
npm-debug.log
```

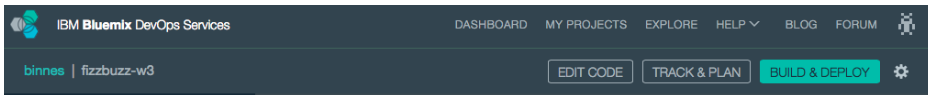
f. Commit the change to the master repo by using the Git Staging view.

Configure a DevOps pipeline to automatically test and deploy code

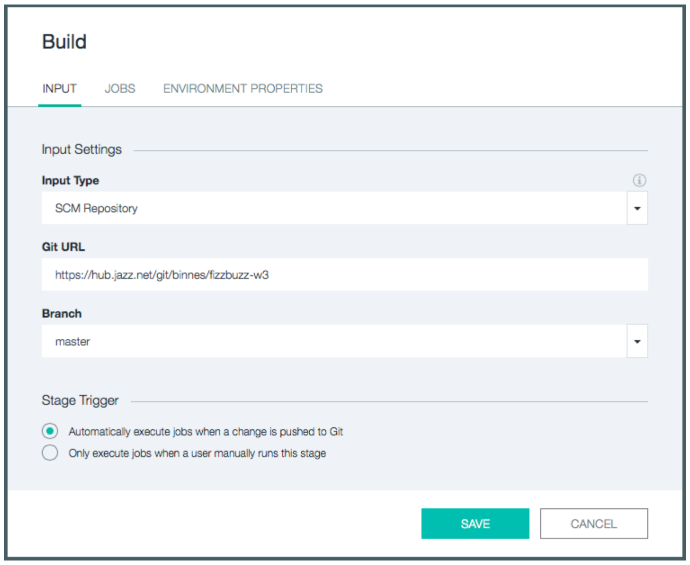
Now that the REST API is working, you must deploy it to IBM Bluemix. Use the Build and Deploy capability in IBM Bluemix DevOps services.

1. Log in to Bluemix DevOps services (https://hub.jazz.net) and from the MY PROJECTS page, select the FizzBuzz project that you created in the previous sections.

2. Build a DevOps pipeline for the project:
   a. Select the **BUILD & DEPLOY** section of the project.



   b. Click **ADD STAGE**.



Build

Input Settings

Input Type: SCM Repository

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

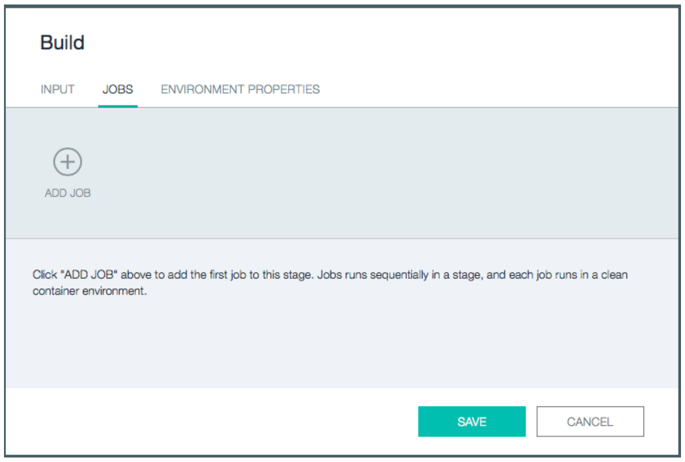
Branch: master

Stage Trigger

- [✓ Automatically execute jobs when a change is pushed to Git]
- Only execute jobs when a user manually runs this stage

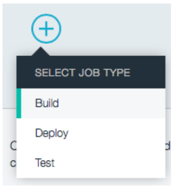
   c. On the INPUT page, name the stage `Build`. Leave the rest of the fields as default values.

   d. Click the **JOBS** tab.



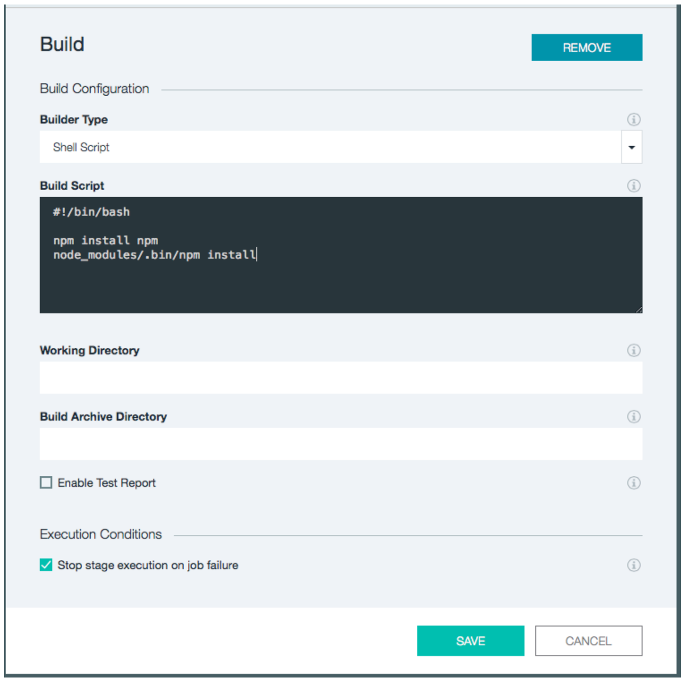
Click "ADD JOB" above to add the first job to this stage. Jobs runs sequentially in a stage, and each job runs in a clean container environment.

e. Click **ADD JOB** and then click **Build** as the job type.



f. For the **Builder Type**, select **Shell Script** and enter the script as shown in the screen capture below. Leave the other fields as default values.

```
#!/bin/bash

npm install npm
node_modules/.bin/npm install
```



Builder Type: Shell Script

Build Script:

```
#!/bin/bash

npm install npm
node_modules/.bin/npm install
```

Working Directory

Build Archive Directory

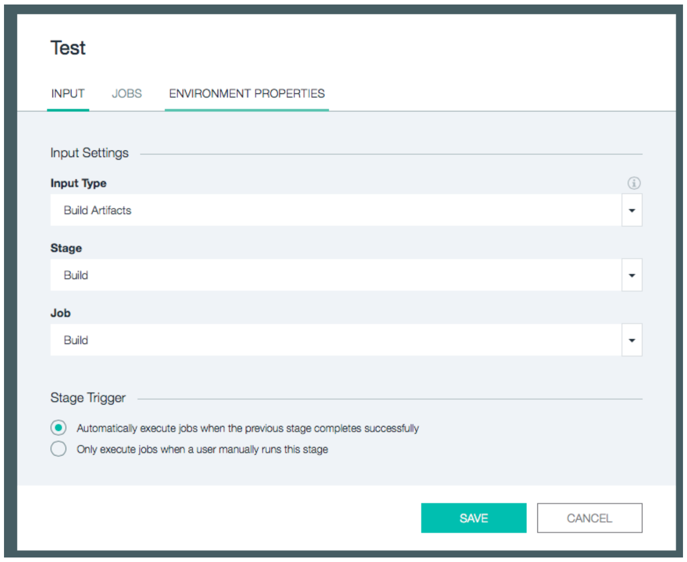
☐ Enable Test Report

✓ Stop stage execution on job failure

g. Click **Save** to save the stage.

h. Click **ADD STAGE** to add another stage in the DevOps pipeline.

i. On the **Input** tab, name the stage `Test.` Then, ensure that the **Input Type** is **Build Artifacts**, the **Stage** is **Build**, and the **Job** is **Build**.



Input Type: Build Artifacts

Stage: Build

Job: Build

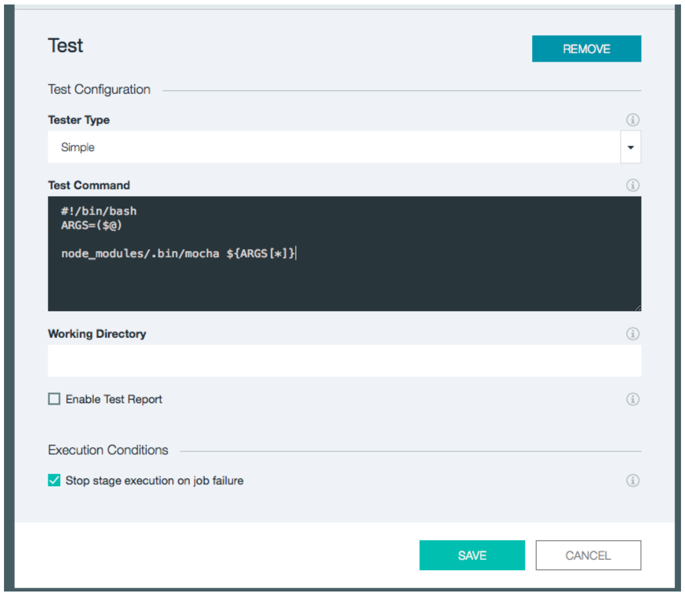
● Automatically execute jobs when the previous stage completes successfully

○ Only execute jobs when a user manually runs this stage

j. On the **JOBS** tab, click **ADD JOB** and click **Test** as the **Job Type**.

k. In the Test Configuration page, ensure that the type is **Simple** and enter the script as shown in the screen capture.

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

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



Tester Type: Simple

Test Command:

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

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

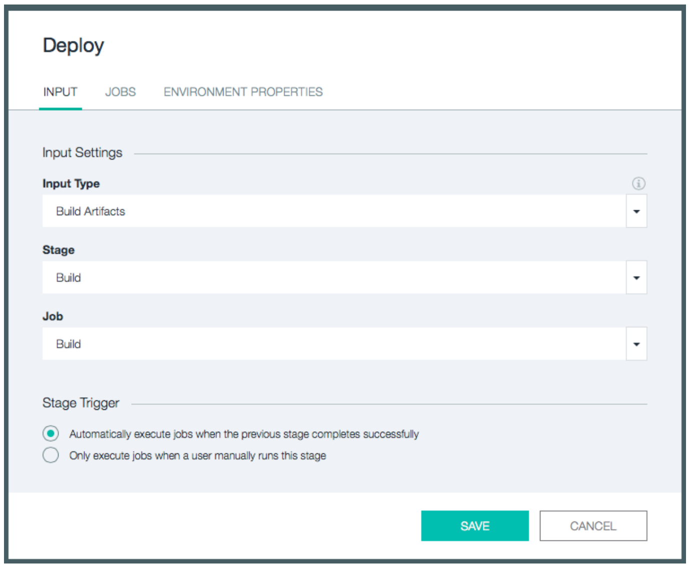
Working Directory:

☐ Enable Test Report

☑ Stop stage execution on job failure

l. Save the stage.

m. Add a third stage and name it `Deploy`. Then, ensure that the **Input Type** is **Build Artifacts**, the **Stage** is **Build**, and the **Job** is **Build**.


Deploy

Input Settings

Input Type: Build Artifacts

Stage: Build

Job: Build

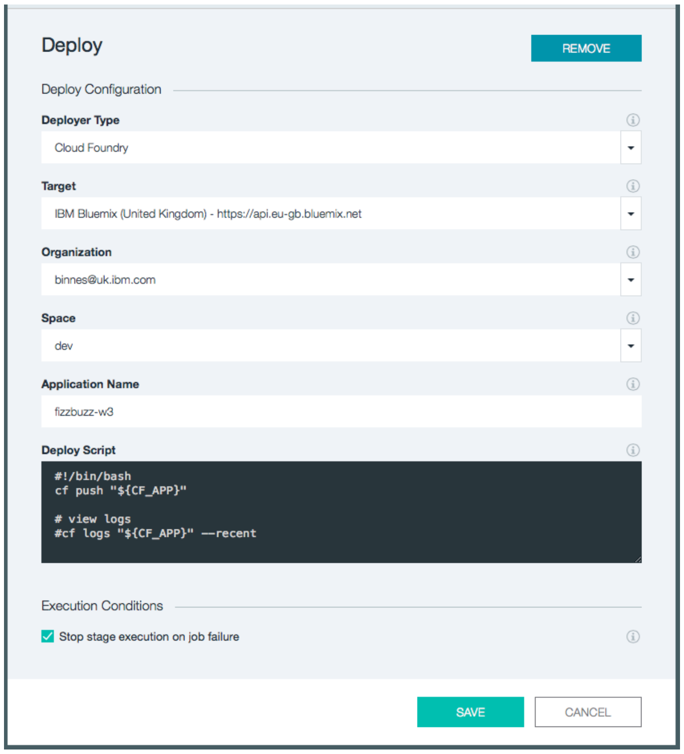
Stage Trigger

(●) Automatically execute jobs when the previous stage completes successfully

( ) Only execute jobs when a user manually runs this stage

n. On the **JOBS** tab, add a job of type Deploy. Use the default values.

```
#!/bin/bash
cf push "${CF_APP}"

# view logs
#cf logs "${CF_APP}" -recent
```


Deploy

Deploy Configuration

Deployer Type: Cloud Foundry

Target: IBM Bluemix (United Kingdom) - https://api.eu-gb.bluemix.net

Organization: binnes@uk.ibm.com

Space: dev

Application Name: fizzbuzz-w3

Deploy Script

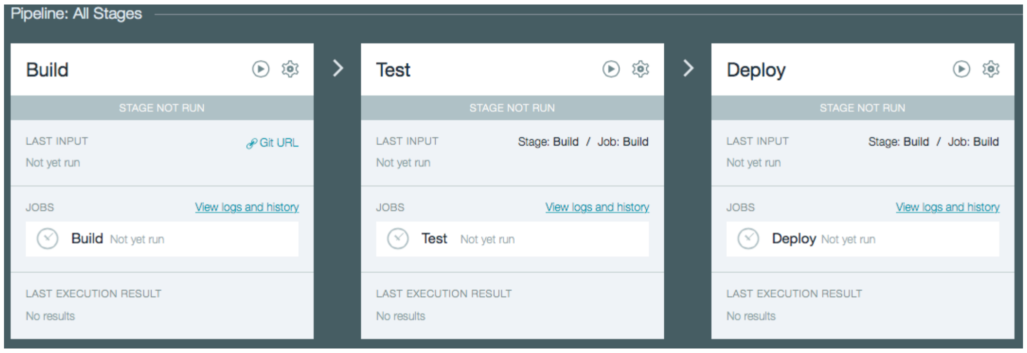
```
#!/bin/bash
cf push "${CF_APP}"

# view logs
#cf logs "${CF_APP}" —recent
```

Execution Conditions

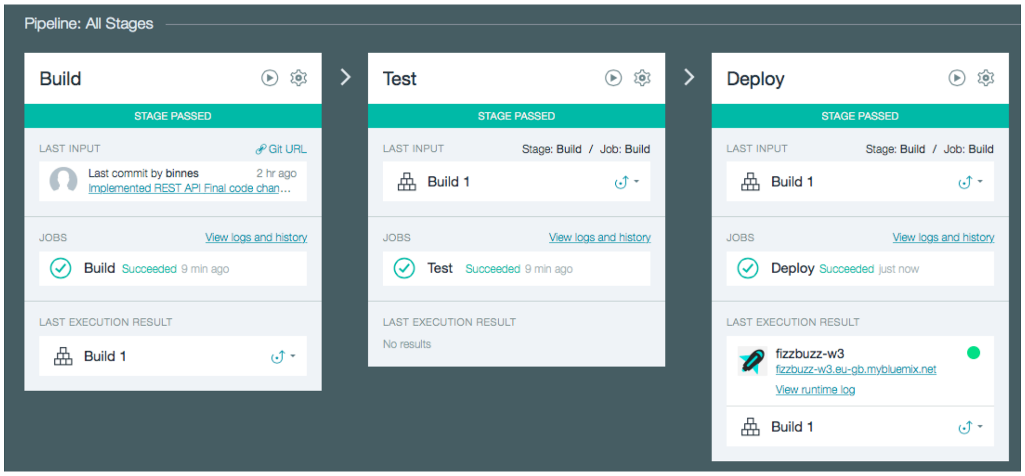
☑ Stop stage execution on job failure

o. Save the stage.

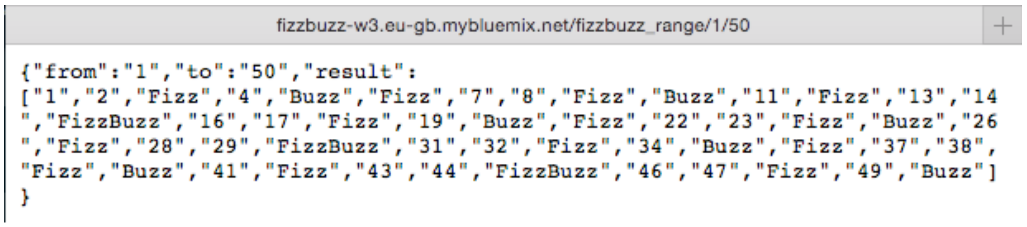
p. Start a new build by clicking **Run** on the Build stage.



You should see the Build stage start to run. After it completes, the Test stage should automatically run and then finally the Deploy stage will run.



q. Click the application URL to launch the application. There is no home page set. However, 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"]
}
```

The code is not yet complete.

A bug in the code was 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.

If you test for the range 9 – 50, you get no results.

This occurs because of the difference between comparing numbers and comparing strings, for example:

- 9 < 50 is true for numbers
- "9" < "50" is false for strings

As part of the tests, ensure the correct type of comparison is taking place.

Some of the 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.

r. Rewrite the test for convertRangeToFizzBuzz to use string parameters. Also use a range that will fail a string comparison but pass a number comparison:

```
  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("9", "50");

              for (var i = 9; i <= 50; i++) {
                expect(spy.withArgs(i).calledOnce).to.be.eql(true,
"Expected convertToFizzBuzz to be called with " + i);
              }
              f.convertToFizzBuzz.restore();
            });
       });
```

The test called “applies FizzBuzz to every number in the range test” is now failing.

s. To fix the problem, ensure that 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;
};
```

t. From the Git Staging view, stage, comment, and commit and push the changes. Then, quickly switch to the IBM Bluemix DevOps Services console for your project in the Build & Deploy settings. You should see a build, test, and deploy being automatically run.