You are in: DevOps Technical Workshop Wiki > V2 Lab IV - A Nonfunctional Hope > V2 Lab 4 - Steps 2A, 2B & 2C

V2 Lab 4 - Steps 2A, 2B & 2C

Like | Updated 8 December 2017 by Roy Mitchley | Tags: *None*

Atom	`Anything in this box needs to be edited in Atom Text`

Terminal	`This box contains lists of commands that should executed in order on the Terminal`

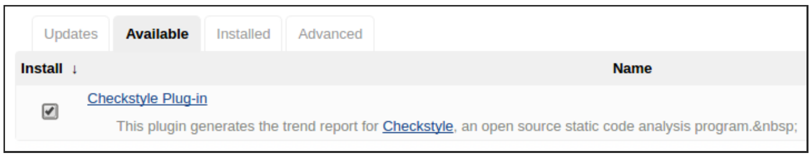
(A) Using jshint in Jenkins

The objective of this lab is to add jshint to our builds to run static code analysis of our build ensure coding standards are met.

JSHINT (http://jshint.com/) is a code quality tool which analyses JavaScript code for common errors. It can be run both client and server side. A `.jshintrc` file is defined outlining the rules which are applied when running the check. One such rule is `"unused": true` , this rule means ensure there are no variables that are unused in the code base ie defined but never needed.

The files for each are stored in `./client/.jshintrc` AND `./server/.jshintrc` within the `todolist` app. Grunt is the task runner that will be used to run the linting for each side of the stack so it can be read in by Jenkins.

– On Jenkins, go to *Manage Jenkins* > *Manage Plugins* and look for the `Checkstyle Plug-in`



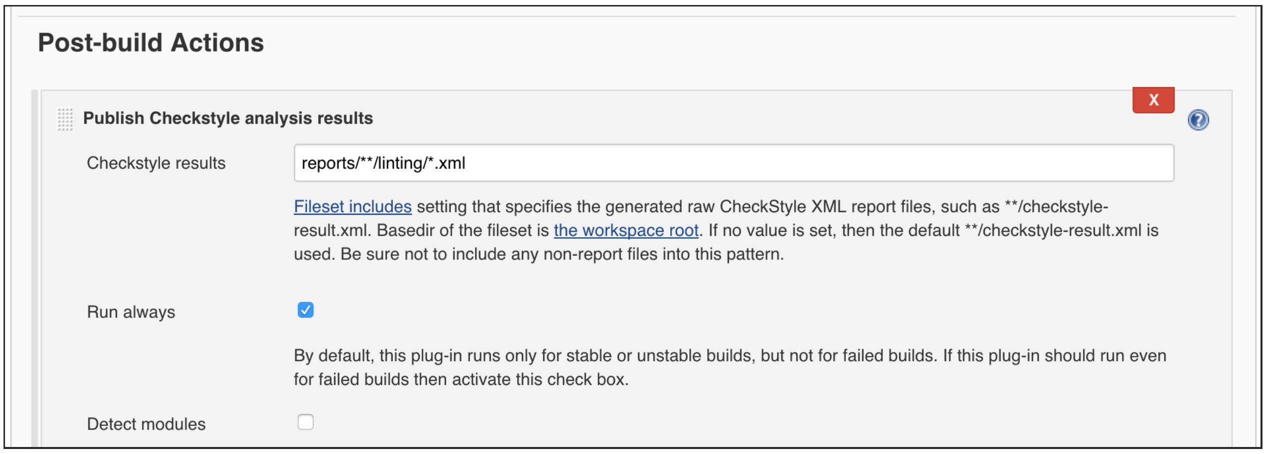
Install ↓	Name
✓	Checkstyle Plug-in This plugin generates the trend report for Checkstyle, an open source static code analysis program.

– Go to the configuration for the *todolist-build* and in the *Execute Shell Step* add jshint steps so that the block looks as follows:

```
npm install

bower install

grunt jshint:ci_client

grunt jshint:ci_server

grunt test:client

grunt test:server-jenkins

grunt build
```

– Add *Publish Checkstyle analysis result* in the *Post-Build action* for the *todolist-build.* Add the string below to the results tab to read in the results of the jshint command and tick the *Run Always* checkbox.

```
reports/**/linting/*.xml
```

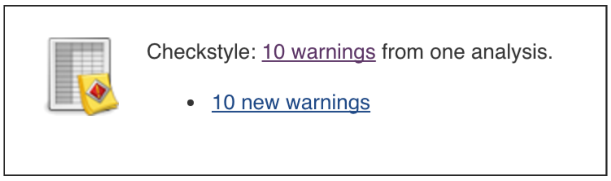


Post-build Actions

Publish Checkstyle analysis results

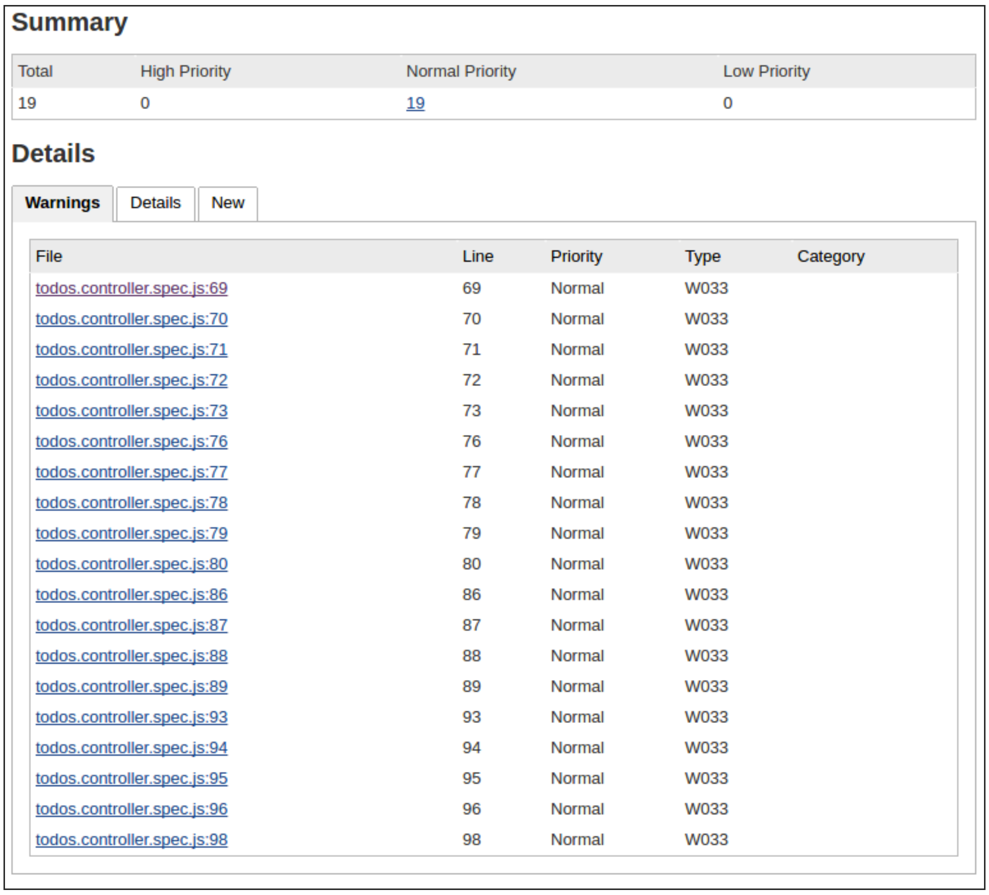
Checkstyle results	reports/**/linting/*.xml
	Fileset includes setting that specifies the generated raw CheckStyle XML report files, such as **/checkstyle-result.xml. Basedir of the fileset is the workspace root. If no value is set, then the default **/checkstyle-result.xml is used. Be sure not to include any non-report files into this pattern.
Run always	✓
	By default, this plug-in runs only for stable or unstable builds, but not for failed builds. If this plug-in should run even for failed builds then activate this check box.
Detect modules	☐

– Run the build and on the results page for that job run you should see some errors on the console. Follow the warnings link to view the errors.



Checkstyle: 10 warnings from one analysis.

- 10 new warnings



Summary

Total	High Priority	Normal Priority	Low Priority
19	0	19	0

Details

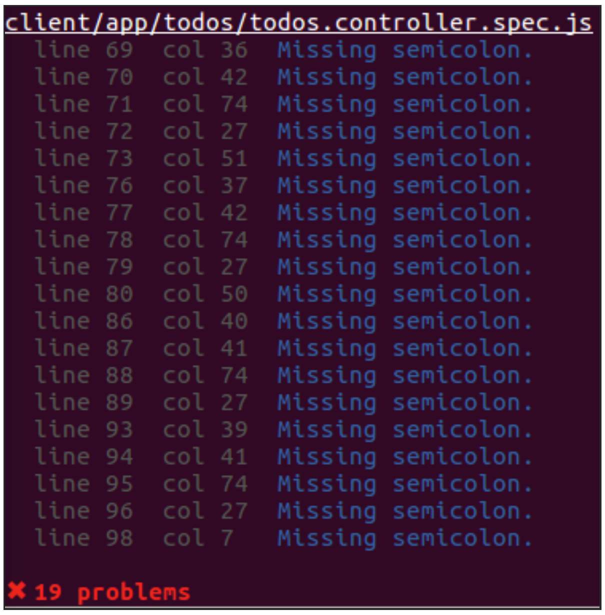
Warnings | Details | New

File	Line	Priority	Type	Category
todos.controller.spec.js:69	69	Normal	W033	
todos.controller.spec.js:70	70	Normal	W033	
todos.controller.spec.js:71	71	Normal	W033	
todos.controller.spec.js:72	72	Normal	W033	
todos.controller.spec.js:73	73	Normal	W033	
todos.controller.spec.js:76	76	Normal	W033	
todos.controller.spec.js:77	77	Normal	W033	
todos.controller.spec.js:78	78	Normal	W033	
todos.controller.spec.js:79	79	Normal	W033	
todos.controller.spec.js:80	80	Normal	W033	
todos.controller.spec.js:86	86	Normal	W033	
todos.controller.spec.js:87	87	Normal	W033	
todos.controller.spec.js:88	88	Normal	W033	
todos.controller.spec.js:89	89	Normal	W033	
todos.controller.spec.js:93	93	Normal	W033	
todos.controller.spec.js:94	94	Normal	W033	
todos.controller.spec.js:95	95	Normal	W033	
todos.controller.spec.js:96	96	Normal	W033	
todos.controller.spec.js:98	98	Normal	W033	

– Running `grunt` locally, the `~/todolist/` project will also produce the same result. Fix the errors by going to the line identified(s) and

Terminal

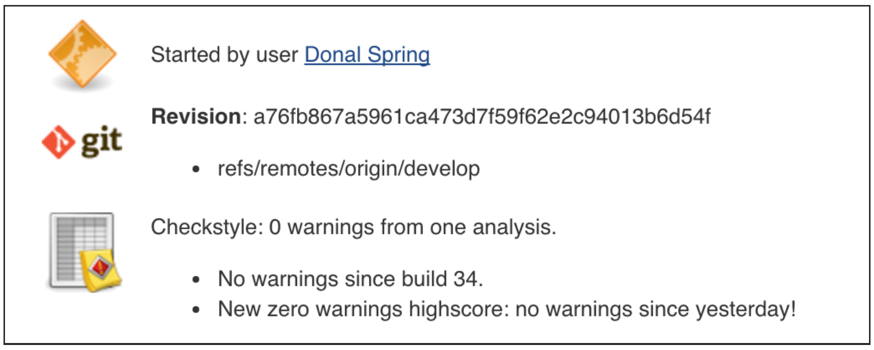
```
cd ~/todolist/
grunt
```



```
client/app/todos/todos.controller.spec.js
  line 69   col 36  Missing semicolon.
  line 70   col 42  Missing semicolon.
  line 71   col 74  Missing semicolon.
  line 72   col 27  Missing semicolon.
  line 73   col 51  Missing semicolon.
  line 76   col 37  Missing semicolon.
  line 77   col 42  Missing semicolon.
  line 78   col 74  Missing semicolon.
  line 79   col 27  Missing semicolon.
  line 80   col 50  Missing semicolon.
  line 86   col 40  Missing semicolon.
  line 87   col 41  Missing semicolon.
  line 88   col 74  Missing semicolon.
  line 89   col 27  Missing semicolon.
  line 93   col 39  Missing semicolon.
  line 94   col 41  Missing semicolon.
  line 95   col 74  Missing semicolon.
  line 96   col 27  Missing semicolon.
  line 98   col 7   Missing semicolon.

✖ 19 problems
```

– After fixing the changes, commit the code and watch the build run again.

– Run the build and on the results homepage you should see the report showing no errors similar to below.



Started by user Donal Spring

Revision: a76fb867a5961ca473d7f59f62e2c94013b6d54f

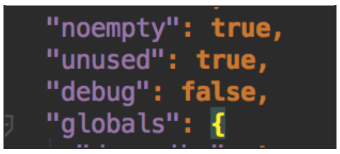
- refs/remotes/origin/develop

Checkstyle: 0 warnings from one analysis.

- No warnings since build 34.
- New zero warnings highscore: no warnings since yesterday!

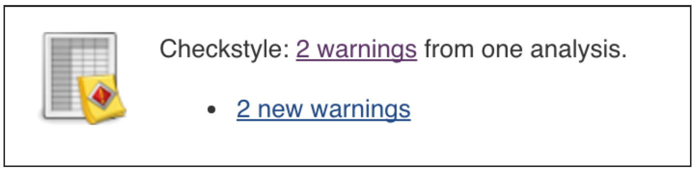
– Next we will introduce a new rule to the linting for server side. Open the `~/todolist/server/.jshintrc` file in Atom. Add this line into the JSON as shown below

Atom	Open the `~/todolist/server/.jshintrc` and set `"unused": true,`



```
"noempty": true,
"unused": true,
"debug": false,
"globals": {
```

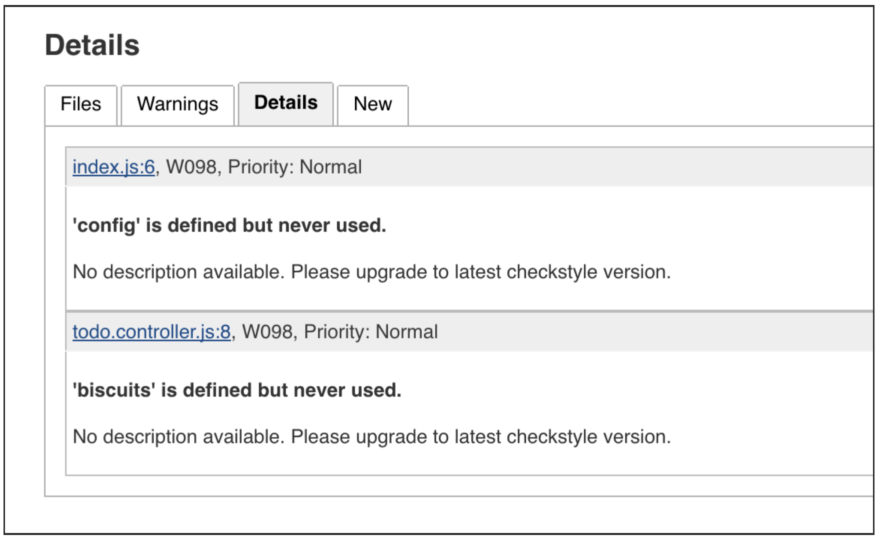
– Commit your code and monitor the build. You should now see some Warning appear for the Checkstyle.



Checkstyle: 2 warnings from one analysis.

- 2 new warnings

– Select the Warnings and fix the issues. Use the details tab to show you where / what to fix. If variables are unused they should be removed (hint hint). Commit your code fix and watch the build pass



Details

Files | Warnings | Details | New

index.js:6, W098, Priority: Normal

'config' is defined but never used.

No description available. Please upgrade to latest checkstyle version.

todo.controller.js:8, W098, Priority: Normal

'biscuits' is defined but never used.

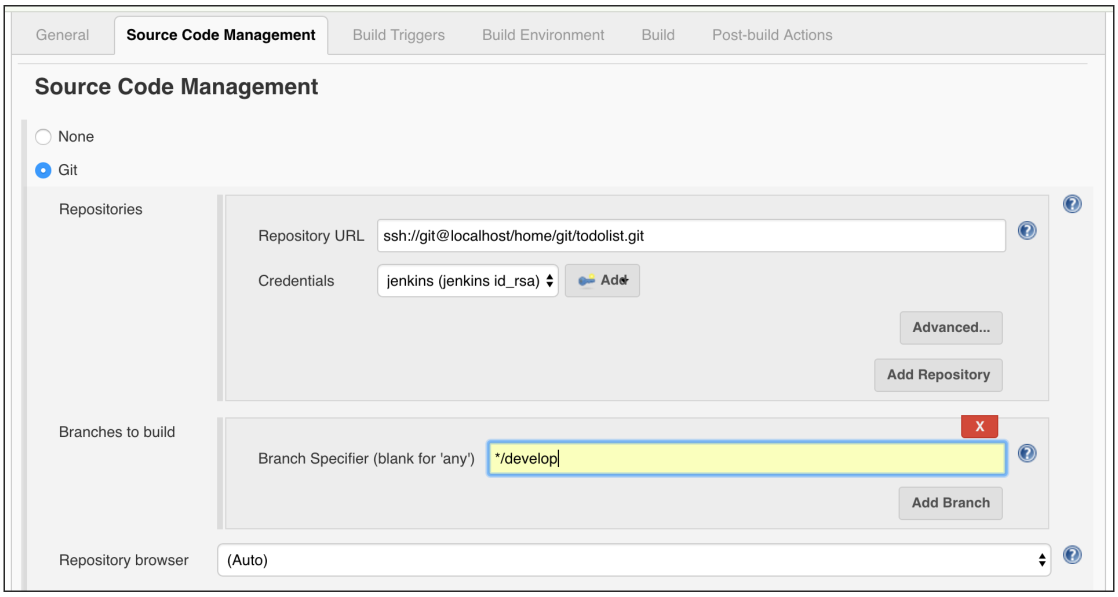
No description available. Please upgrade to latest checkstyle version.

(B) Performance testing using Jenkins

Automating performance metrics at early phases of project development can lead to spotting errors sooner and more importantly fixing them while they are still small problems. This lab will simulate using Jenkins to spot performance degradation in a nightly run test job. This job is run nightly as we want to keep the feedback loop short for our builds but still get the feedback from performance issues at least daily.

– On Jenkins, create a *New Item* (*Freestyle*) `nightly-perf-test-si`

– On the Source Code Management tab, checkout the source code for the app as done before `ssh://git@git.server/home/git/todolist.git`



Source Code Management

None

Git [✓]

Repositories

Repository URL: ssh://git@localhost/home/git/todolist.git

Credentials: jenkins (jenkins id_rsa)

Branches to build

Branch Specifier (blank for 'any'): */develop

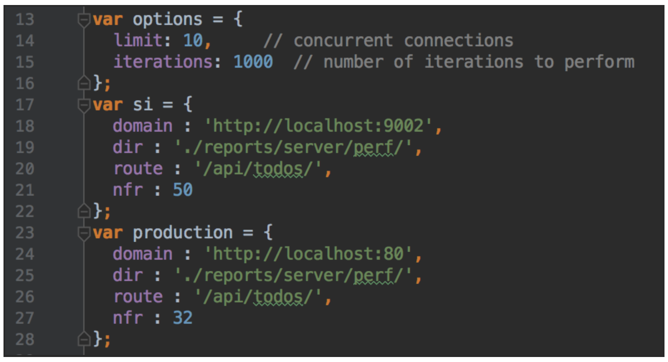
Repository browser: (Auto)

– Run the following in the *Add Build Step* > *Execute Shell.* The `set +e` below is used to tell jenkins not to exit when it receives a non 0 return code. This code is then captured and the next perf-test is run and the response code (`rc1 / rc2`) is captured. The error exit is then turned off (`set -e`) and exit command is called with the bitwise response code from both runs ie if either failed, the build will fail.

```
npm install
set +e
grunt perf-test:si:create
rc1=$?
grunt perf-test:si:show
rc2=$?
set -e
exit $(($rc1 | $rc2))
```

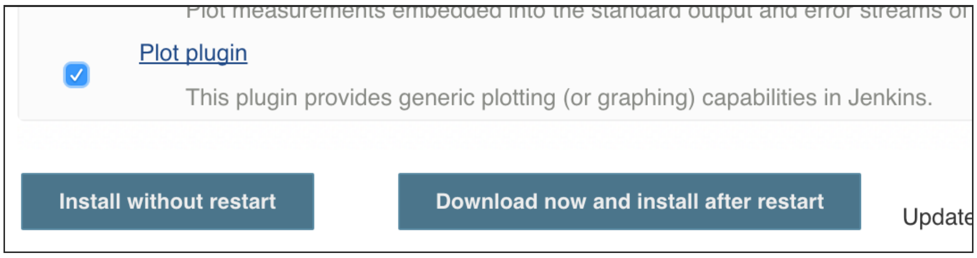
– Add ANSI output to the job (Gnome-terminal). Then *Save / Apply*

At this point Jenkins can now run the performance tests. The task for the performance tests is stored in the `./tasks/perf-test.js` directory/file of the project. The top of the file has the configurable components of the tests being run. The `limit` variable is the amount of concurrent connections fired off to the rest endpoint & the `iterations` is the number of requests to fire off. The `nfr` for each environment is defined in the individual blocks for each environment along with the `domain` / `routes` that are being tested. The response times (`stats`) are printed out to the console for the job and also some printed to file for interpretation by Jenkins.



```
var options = {
    limit: 10,      // concurrent connections
    iterations: 1000  // number of iterations to perform
};
var si = {
    domain : 'http://localhost:9002',
    dir : './reports/server/perf/',
    route : '/api/todos/',
    nfr : 50
};
var production = {
    domain : 'http://localhost:80',
    dir : './reports/server/perf/',
    route : '/api/todos/',
    nfr : 32
};
```

– To make the stats meaningful to Jenkins, install the `Plot Plugin`



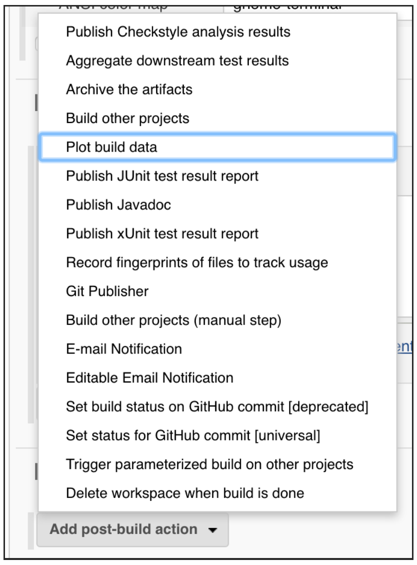
Plot measurements embedded into the standard output and error streams of

✓ Plot plugin

This plugin provides generic plotting (or graphing) capabilities in Jenkins.

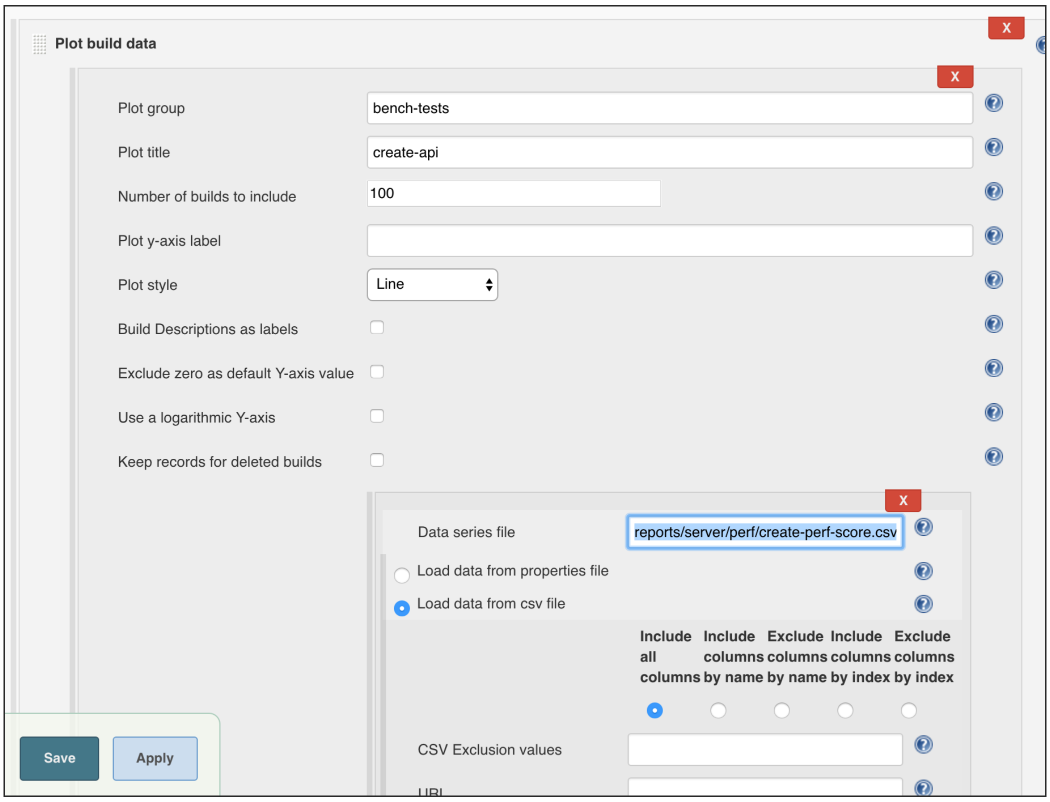
– To make the stats meaningful to Jenkins, install the `Plot Plugin`

– Once installed, go back to the `nightly-perf-test-si` job to configure the reading in the results. Add a *Post-build Action* > *Plot Build Data*



Publish Checkstyle analysis results
Aggregate downstream test results
Archive the artifacts
Build other projects
Plot build data
Publish JUnit test result report
Publish Javadoc
Publish xUnit test result report
Record fingerprints of files to track usage
Git Publisher
Build other projects (manual step)
E-mail Notification
Editable Email Notification
Set build status on GitHub commit [deprecated]
Set status for GitHub commit [universal]
Trigger parameterized build on other projects
Delete workspace when build is done
Add post-build action

– On the new dialog, name the *Plot group* eg `benchmark-tests` and add `create-api` as the *Plot title*. Set the *Number of Builds to Include* to a large number like `100`. Set the *Data Series file* to be `reports/server/perf/create-perf-score.csv` and mark the *Load data from CSV* checkbox. *Apply* those changes


Plot build data

Plot group	bench-tests
Plot title	create-api
Number of builds to include	100
Plot y-axis label	
Plot style	Line
Build Descriptions as labels	☐
Exclude zero as default Y-axis value	☐
Use a logarithmic Y-axis	☐
Keep records for deleted builds	☐

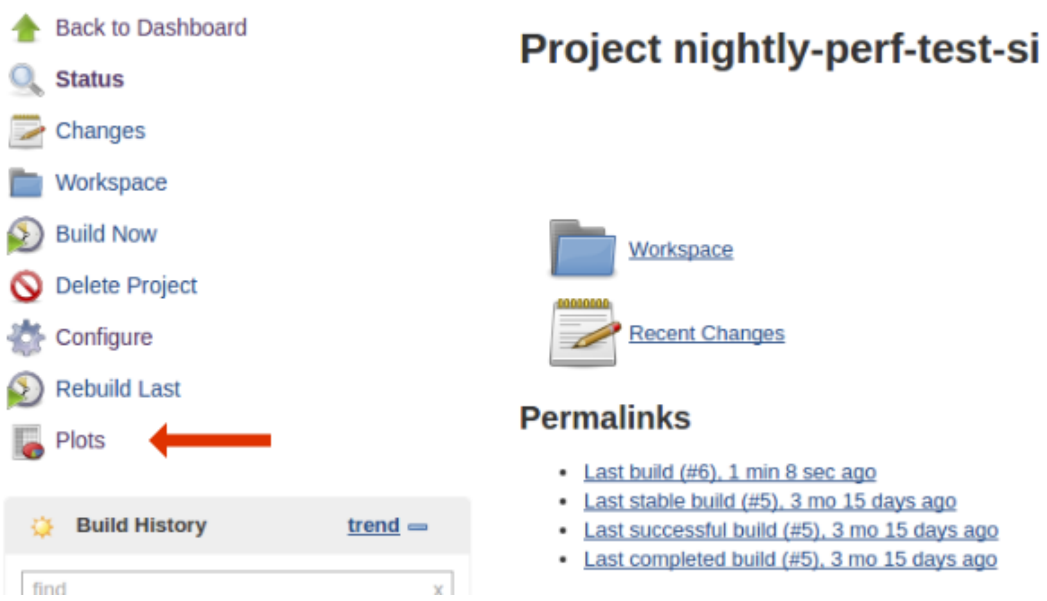
Data series file	reports/server/perf/create-perf-score.csv
Load data from properties file	○
Load data from csv file	●

Include all columns	Include columns by name	Exclude columns by name	Include columns by index	Exclude columns by index
●	○	○	○	○

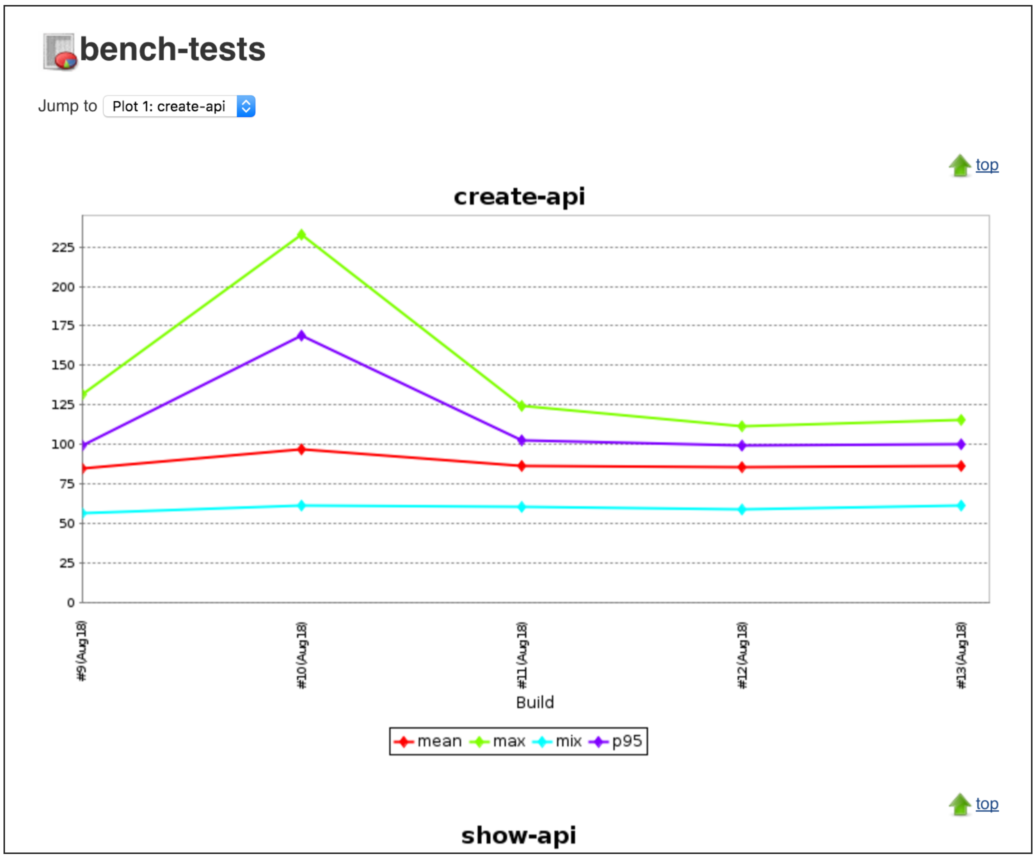
CSV Exclusion values

– Hit *Add Plot* to add another plot. Fill out the information again but this time setting the *Plot title* to `show-api`. Keep the *Plot group* the same as before: `bench-tests`. Set the *Data Series file* to be `reports/server/perf/show-perf-score.csv` and mark the *Load data from CSV* checkbox. *Save* those changes and run the job. Run it a few times to start to generate the data points on the plot. Note: It is possible that the `npm install` will fail on the first execution - simply run the build again.

In the nightly-perf-test-si project you should see a menu option called Plots.


Back to Dashboard
Status
Changes
Workspace
Build Now
Delete Project
Configure
Rebuild Last
Plots
Build History
trend
find
Project nightly-perf-test-si
Workspace
Recent Changes
Permalinks
Last build (#6), 1 min 8 sec ago
Last stable build (#5), 3 mo 15 days ago
Last successful build (#5), 3 mo 15 days ago
Last completed build (#5), 3 mo 15 days ago

When you open this you should see a graph like this once you have run some tests.



To simulate how this kind of reporting is useful to us, we will implement some data transformation in our API module. This data transformation will be written as non performant code (Sam Code) and the result on Jenkins will be monitored.

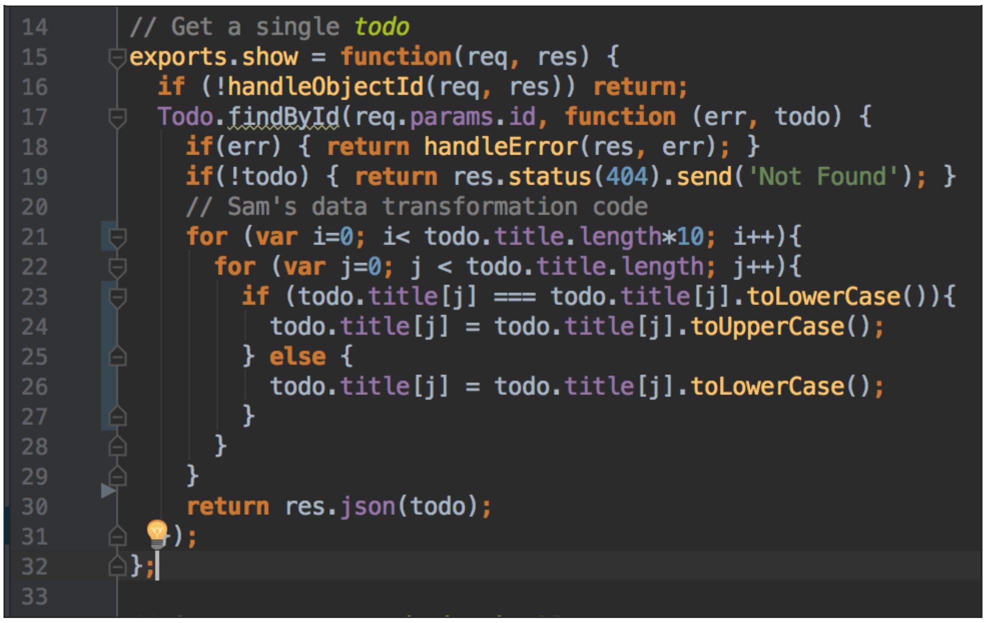
– Open the project app code in Atom and add the (Sam Code)

Terminal	`cd ~/todolist && atom .`

Atom	Navigate to the todo api files `server/api/todo/todo.controller.js`. Navigate to the `exports.show` function (`// Get a single todo` is written above it) Add in Sam's non performant code above the `return res.json(todo);` as shown below.

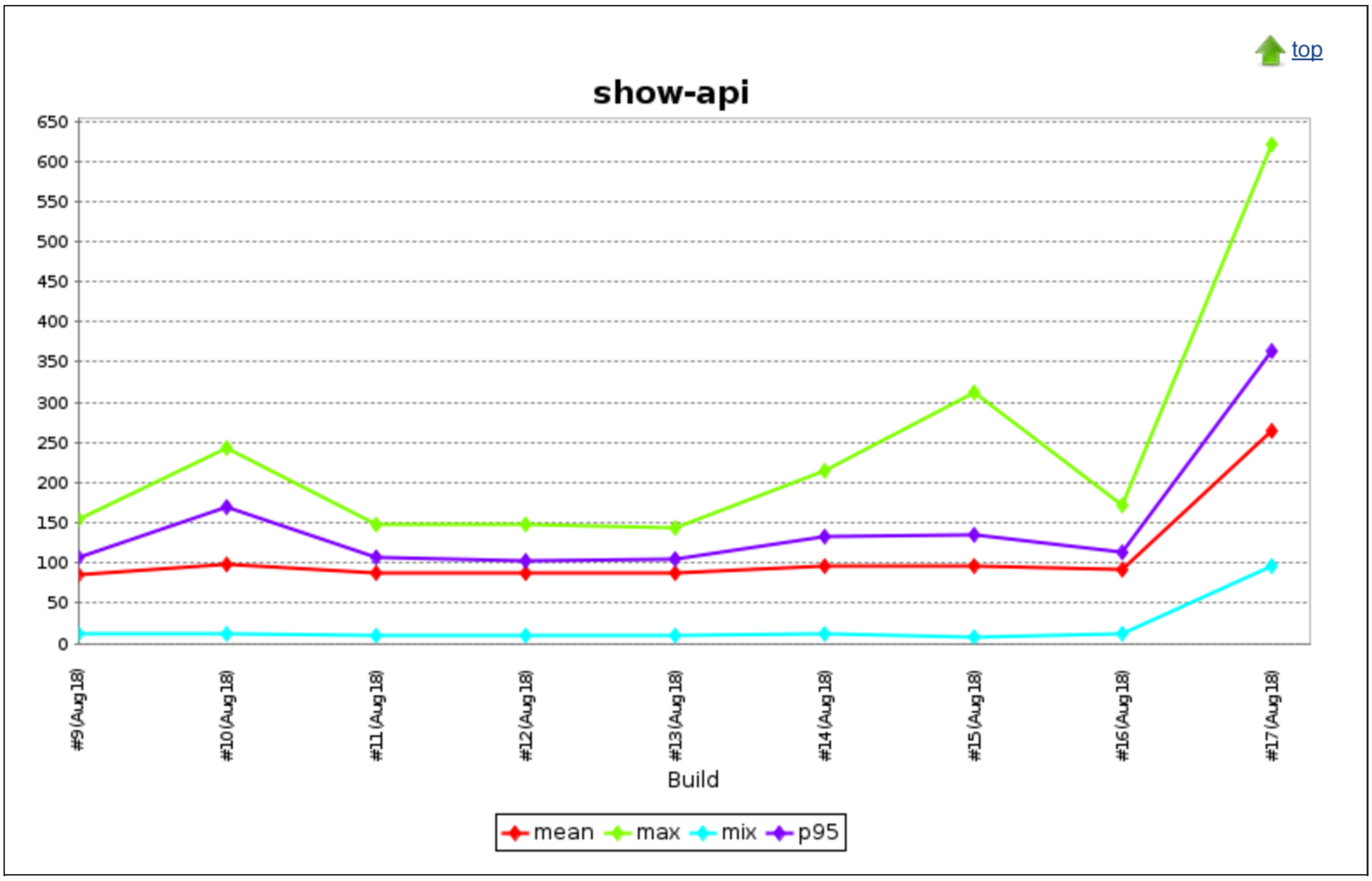
```
// Sam's data transformation code
for (var i=0; i< todo.title.length*10; i++){
  for (var j=0; j < todo.title.length; j++){
    if (todo.title[j] === todo.title[j].toLowerCase()){
      todo.title[j] = todo.title[j].toUpperCase();
    } else {
      todo.title[j] = todo.title[j].toLowerCase();
    }
  }
}
```

The function should now look like this



```
// Get a single todo
exports.show = function(req, res) {
  if (!handleObjectId(req, res)) return;
  Todo.findById(req.params.id, function (err, todo) {
    if(err) { return handleError(res, err); }
    if(!todo) { return res.status(404).send('Not Found'); }
    // Sam's data transformation code
    for (var i=0; i< todo.title.length*10; i++){
      for (var j=0; j < todo.title.length; j++){
        if (todo.title[j] === todo.title[j].toLowerCase()){
          todo.title[j] = todo.title[j].toUpperCase();
        } else {
          todo.title[j] = todo.title[j].toLowerCase();
        }
      }
    }
    return res.json(todo);
  });
};
```

– Commit your code to the repository and wait for the `deploy-si` phase to complete then run a another `nightly-perf-test-si` job.



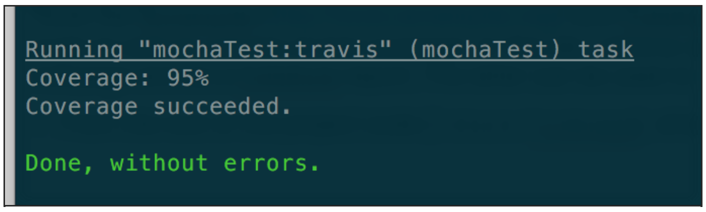
– Fix your changes before moving on to the next lab by either git reverting the previous checkin or removing the bad code.

(C) Using code coverage metrics to analyse the volume and depth of testing

Tools like Sonarqube (http://www.sonarqube.org/) and Cobetura (http://cobertura.github.io/cobertura/) are useful for running static code analysis. They can be used to identify potential problems in code as well as volume of test coverage. In this lab, Jenkins will be configured to read in the results of the coverage metrics for the server side. The reports will be displayed in two ways, as a webpage and as a Cobetura report. The latter can be used to fail the build if certain metrics are not achieved.

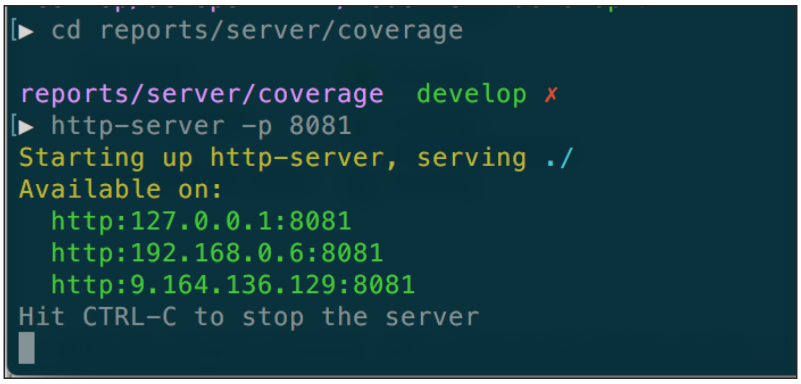
– From the root of the project code (`~/todolist`), when you run `grunt test` you can see the output of the coverage report claiming 95% coverage. Mocha is using Blanket.js to create these stats. There are other options for JavaScript such as Istanbul, which has good integration with another JavaScript testing tool called Jasmine

Terminal	`cd ~/todolist`
	`grunt test`

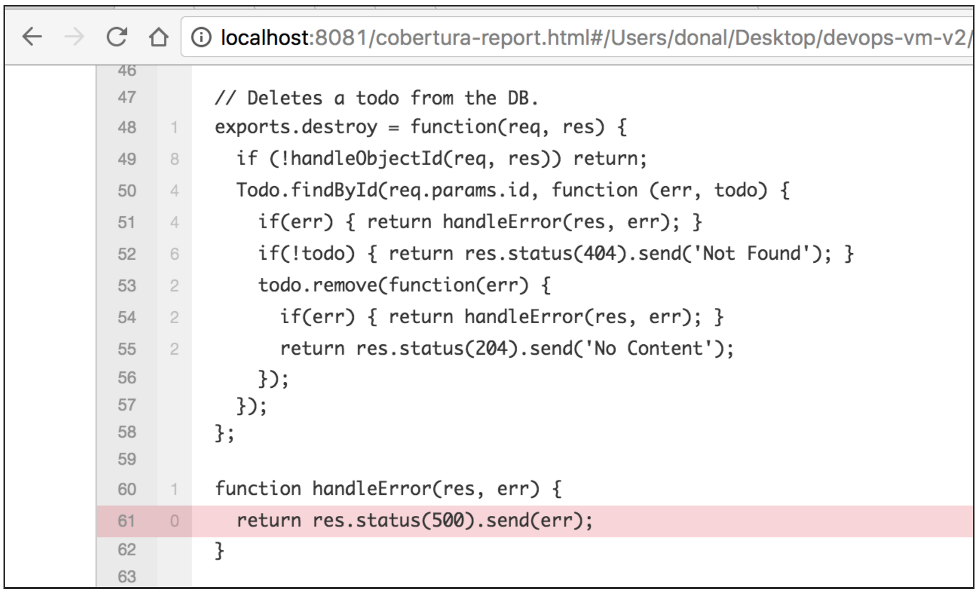

```
Running "mochaTest:travis" (mochaTest) task
Coverage: 95%
Coverage succeeded.

Done, without errors.
```

– The report is also generated as a webpage located in (`reports/server/coverage/coverage.html`). This can be served locally my moving into the directory and running `http-server.` You can see the report in the browser then on http://localhost:8081/cobertura-report.html

Terminal	`cd reports/server/coverage/coverage.html`
	`http-server -p 8081`

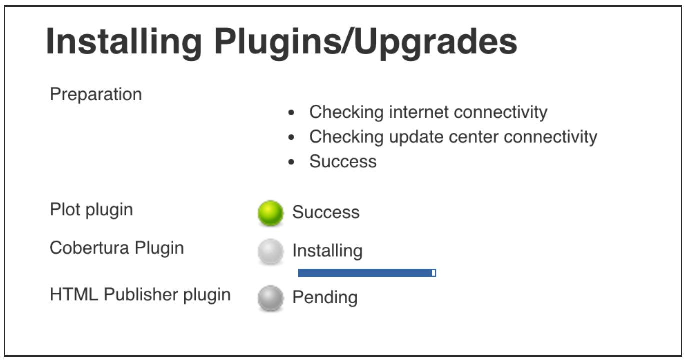

```
▶ cd reports/server/coverage

reports/server/coverage  develop ✗
▶ http-server -p 8081
Starting up http-server, serving ./
Available on:
  http:127.0.0.1:8081
  http:192.168.0.6:8081
  http:9.164.136.129:8081
Hit CTRL-C to stop the server
```

The report shows each file that was tested and the number of times each line of the file was tested. For example, below `handleError()` function is never called so it is highlighted red.


localhost:8081/cobertura-report.html#/Users/donal/Desktop/devops-vm-v2/
```
     // Deletes a todo from the DB.
1    exports.destroy = function(req, res) {
8      if (!handleObjectId(req, res)) return;
4      Todo.findById(req.params.id, function (err, todo) {
4        if(err) { return handleError(res, err); }
6        if(!todo) { return res.status(404).send('Not Found'); }
2        todo.remove(function(err) {
2          if(err) { return handleError(res, err); }
2          return res.status(204).send('No Content');
         });
       });
     };

1    function handleError(res, err) {
0      return res.status(500).send(err);
     }
```

Terminal	To kill the `http-server,` Hit `CTRL and C`

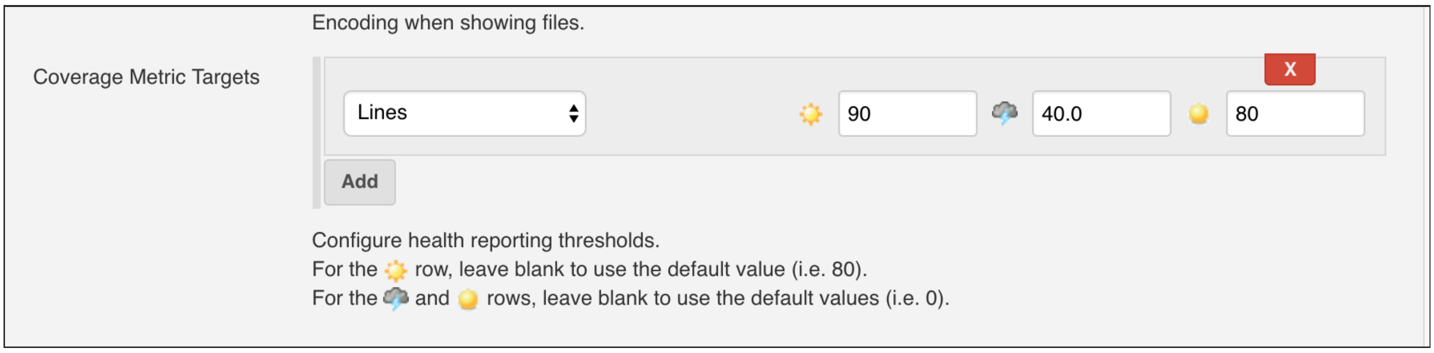
– To get Jenkins interpreting the results, install two plugins `Cobetura Plugin` and `HTML Publisher Plugin`. If you cannot remember how to get to the plugins page it's (from Jenkins homepage) *Manage Jenkins* > *Manage Plugins*.



Installing Plugins/Upgrades

Preparation	• Checking internet connectivity • Checking update center connectivity • Success
Plot plugin	Success
Cobertura Plugin	Installing
HTML Publisher plugin	Pending

– With the plugins installed, add a *Post-build Action* on the `todolist-build` job. Add *Publish Cobertura Coverage Report*. Use `reports/server/coverage/cobertura-report.xml` as the path and check the *Fail builds if no reports* option. In the *Advanced* tab add 90% coverage for lines as shown below [40 for the weather and 80 for UNSTABLE Build]. Usually these values would start low and be raised over the life time of a project. Coverage is currently at 95% so raising this to 100 and setting the yellow status to 96 will cause the build Health to change



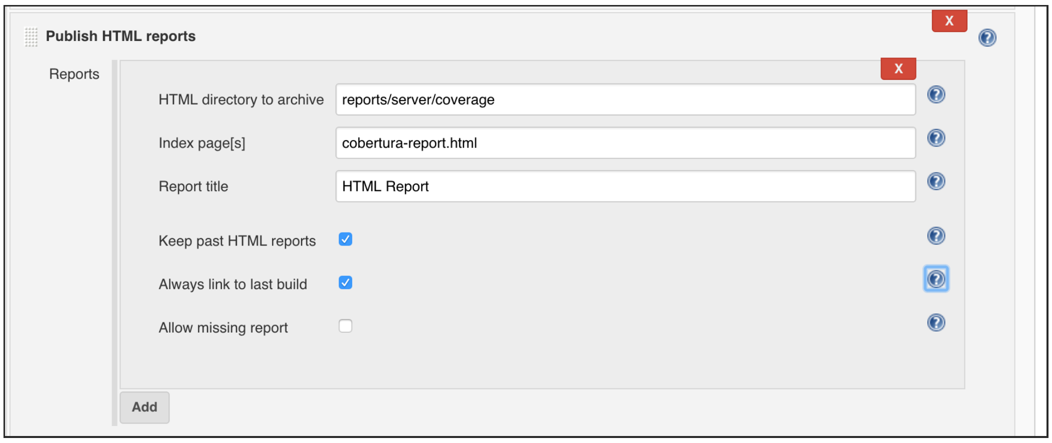
Encoding when showing files.

Coverage Metric Targets

Lines 90 40.0 80

Configure health reporting thresholds.
For the row, leave blank to use the default value (i.e. 80).
For the and rows, leave blank to use the default values (i.e. 0).

– Publish the webreport generated by adding a *Post-build Action* on the `todolist-build` job. Add *Publish HTML Report* and fill in as shown below with the directory set to `reports/server/coverage` and `cobertura-report.html` for the index page.



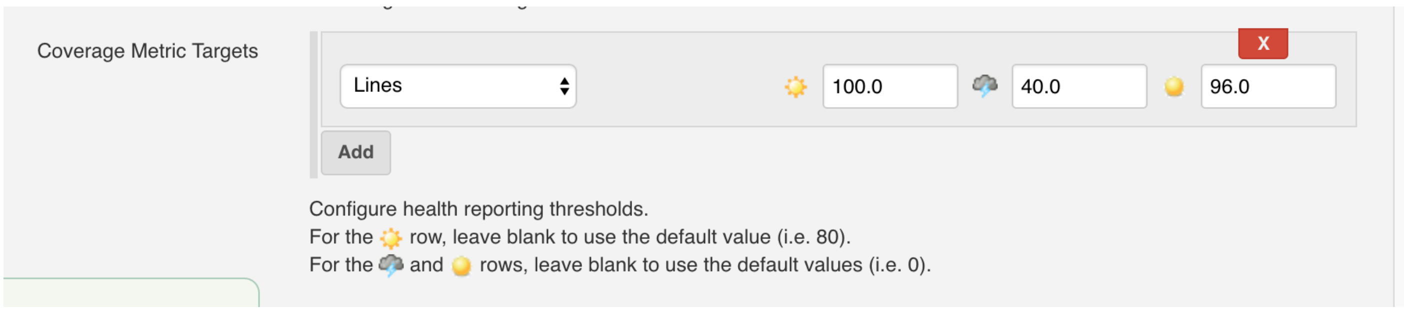
Publish HTML reports

Reports

HTML directory to archive	reports/server/coverage
Index page[s]	cobertura-report.html
Report title	HTML Report
Keep past HTML reports	✓
Always link to last build	✓
Allow missing report	☐

– Finally, Save the configuration and trigger a build. Run twice to see the coverage trend appear on the dashboard. The reports allow you to drill into the configuration and see where exactly there are failures.

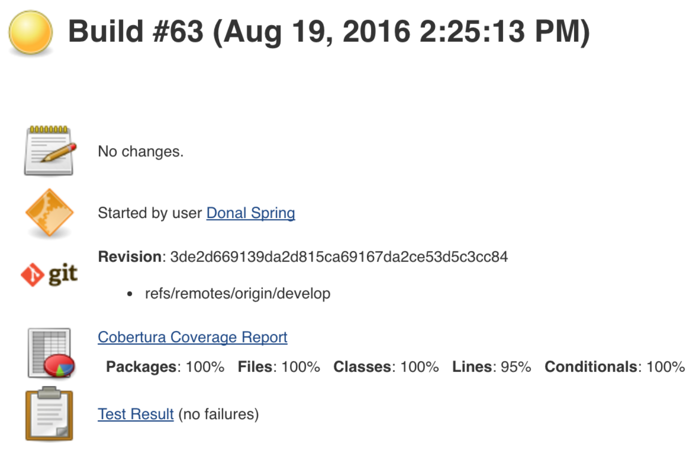
– As in a normal project, this metrics should be increased over time. go back to the Cobertura settings for the todolist-build job. Set the metrics up to 100, 40 and 97 and re-run the build.


Coverage Metric Targets

Lines ☀ 100.0 ⛈ 40.0 ● 96.0

Configure health reporting thresholds.
For the ☀ row, leave blank to use the default value (i.e. 80).
For the ⛈ and ● rows, leave blank to use the default values (i.e. 0).

The build status should change to YELLOW


Build #63 (Aug 19, 2016 2:25:13 PM)

No changes.

Started by user Donal Spring

Revision: 3de2d669139da2d815ca69167da2ce53d5c3cc84

- refs/remotes/origin/develop

Cobertura Coverage Report

Packages: 100% **Files**: 100% **Classes**: 100% **Lines**: 95% **Conditionals**: 100%

Test Result (no failures)

Comments

There are no comments.