Learn Continuous Integration with Bitbucket Pipelines

In this tutorial, we'll see how to set up a CI workflow in Bitbucket Pipelines with a simple Node.js example.

BY STEN PITTET

Requirements

Time:

30 minutes

Audience:

You are new to continuous integration and/or Bitbucket Pipelines

Prerequisites:

- Node v4.6 or above to run the sample application
- A terminal to execute bash commands
- Git to manage the repository and push back to Bitbucket Cloud.
- A Bitbucket account

Try it free

Testing software can be an expensive part of the release cycle. Not only do you have to verify that new changes are working as expected, but you also need to make sure that existing features have not broken. This can quickly become a significant burden as the scope of testing increases with each new release. It's also a tedious task for your team as they'll have to manually check basic aspects of your application time and time again.

Test automation exists to solve this issue by eliminating the redundant and annoying aspect of testing. You write a test once, and it can be automatically executed by a testing framework, without the need for human intervention. Then you can go one step further and hook your repository on a continuous integration (CI) service like Bitbucket Pipelines to run your tests automatically on every change that gets pushed to the main repository.

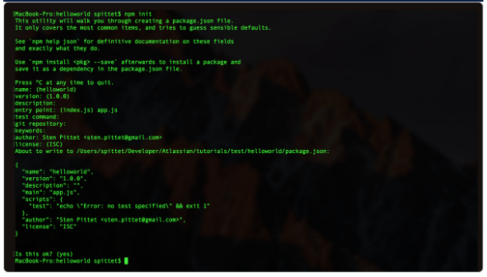
In this tutorial, we'll see how to set up a CI workflow in Bitbucket Pipelines with a simple Node.js example. We'll start by building our app; then we'll look at how we can implement a simple test, and finally, we'll learn how to hook it up to Bitbucket Pipelines.

Step 1: Build a simple Hello World application

To begin our tutorial, we'll start by creating a basic Node.js application that displays "Hello World!" in the browser. It's the same code as the example application in the Express framework, and even if you're not using Node, it should be easy to understand and adapt to your need. If you want to skip this part, you can find the final code of the application in Bitbucket.

Start by creating a directory and run npm init to initialize your new Node project. You can use all the default settings except for the entry point that you need to change to server.js instead of index.js.

```
mkdir helloworld cd helloworld npm init
```



Your npm init settings

If you forgot to change the entry point to server.js don't worry as you can edit it after in your *package.json* file. When you're done your *helloworld* directory should have a single file called package.json, and it should look like this:

package.json

```
{   "name": "helloworld",   "version": "1.0.0",   "descri
```

Now we will install Express which is a web framework for Node. We'll save it as a dependency with the *--save* parameter. It's important to save it explicitly as a dependency because it will allow later Bitbucket Pipelines to know what dependencies needs to be installed to test your application.

```
npm install express --save
```

Create a file called *server.js* and copy and paste the code below to create your *Hello World* application.

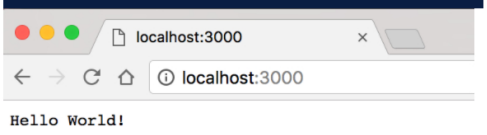
```
var express = require('express') var app = express()   //
```

At this stage your application folder should look like the following.

```
. ├──package.json └── server.js
```

You can now simply start your Node app and go to http://localhost:3000 to see it in action.

```
npm start
```



Our example in action

Step 2: Write a test for our application

Now that we have our application up and running we can start writing a test for it. In this case, we will make sure that it always displays "Hello World!" properly when the user calls the base URL. This is a very basic example, but following the same structure you will be able to add unit tests for your own application, but also do more complex things such has checking authenticating, creating and deleting content, testing permissions.

To do that we'll use a test framework called Mocha and a library called supertest which will help manage HTTP requests in our tests. When comes the time for you to add test automation to your application, take some time to research the right testing framework for you. Depending on the language, the options may vary. Some frameworks are quite established like PHPUnit for PHP but in other cases you might have to explore a bit to understand which test framework will be the best fit for your project. You can find an exhaustive list of unit testing frameworks on Wikipedia but we recommend to approach the developer community of your language to get some recommendations.

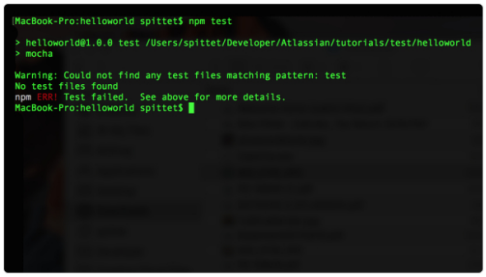
Run the command below in your terminal to install both Mocha and supertest as development dependencies for your application.

```
npm install mocha --save-dev npm install supertest --save
```

In your *package.json* file replace the test script command to invoke *mocha* instead.

```
{   "name": "helloworld",   "version": "1.0.0",   "descri
```

If you run the command *npm* test in your terminal you should get an error that says that no tests could be found. This is expected and we'll see next how to add a basic test.



We're now ready and you can create a test folder in which you're going to add a test file.

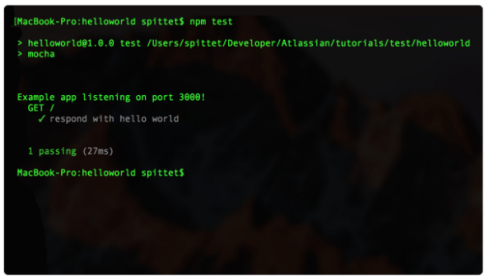
```
mkdir test touch test/test.js
```

Our test will be very simple: it'll only check the presence of the phrase "Hello World!" when the base URL is called.

test.js

```
var request = require('supertest'); var app = require('..
```

If you run *npm* test again in your terminal you should now see that one test is passing.



Congratulations! You now have implemented an automated test for your app. This is just a stepping stone for you to embrace CI and as you build your own application you should look into the **different kind of tests** that you can implement to check the integrity of your system. Just remember that the more complicated the tests are, the more expensive they will be to run.

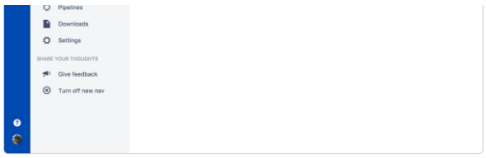
> In this tutorial, we've implemented a feature and then wrote a test for it. It can be interesting to do the opposite and start by writing the tests that should verify your feature. Then you can implement the feature knowing that you already have safeguards to make sure that it's working as expected.

Step 3: Implement your continuous integration workflow with Bitbucket Pipelines

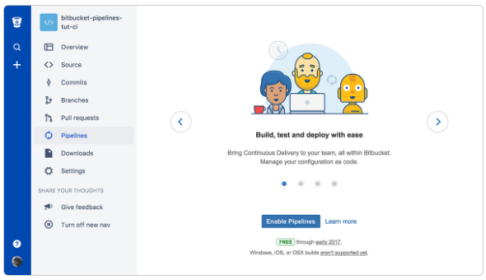
Your test is now scripted and you can run it via the command line. But to start practicing CI you have to make sure that your test suite is run for every new commit. That way developers are alerted as soon as their changes break the application and they can fix the underlying issue right away before moving to their next task. The other advantage of running your tests on every commit is that it can help your team assess the quality of the development – you will be able to visually see how many times new changes break your application, and how quickly things get back to a healthy state.

This may seem daunting but thankfully, automating your tests can easily be done with with Bitbucket Pipelines.

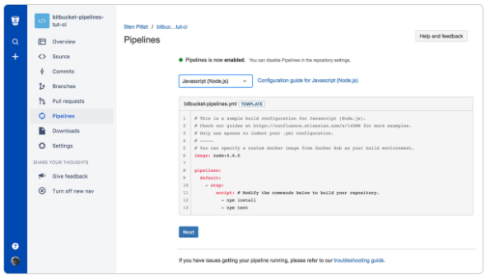
Start by adding your repository to Bitbucket.



Go to Pipelines by clicking on the corresponding menu item in the sidebar.



After enabling Pipelines, pick the Node.js template in the configuration example. This example should have the command *npm install* and *npm test* in the script section and Bitbucket Pipelines will run they just like you would have in your own terminal to install dependencies and run the tests.



You shouldn't need to modify the bitbucket-pipelines.yml file and you can commit it straight to your repository.