BetterProgramming

Follow 198K Followers

Bit Harmony: A New Tool To Create and Collaborate on Independent Node.js Components

The latest open source tool you need for your development workflow

Fernando Doglio Apr 2 · 11 min read

Photo by Christina Morillo from Pexels.

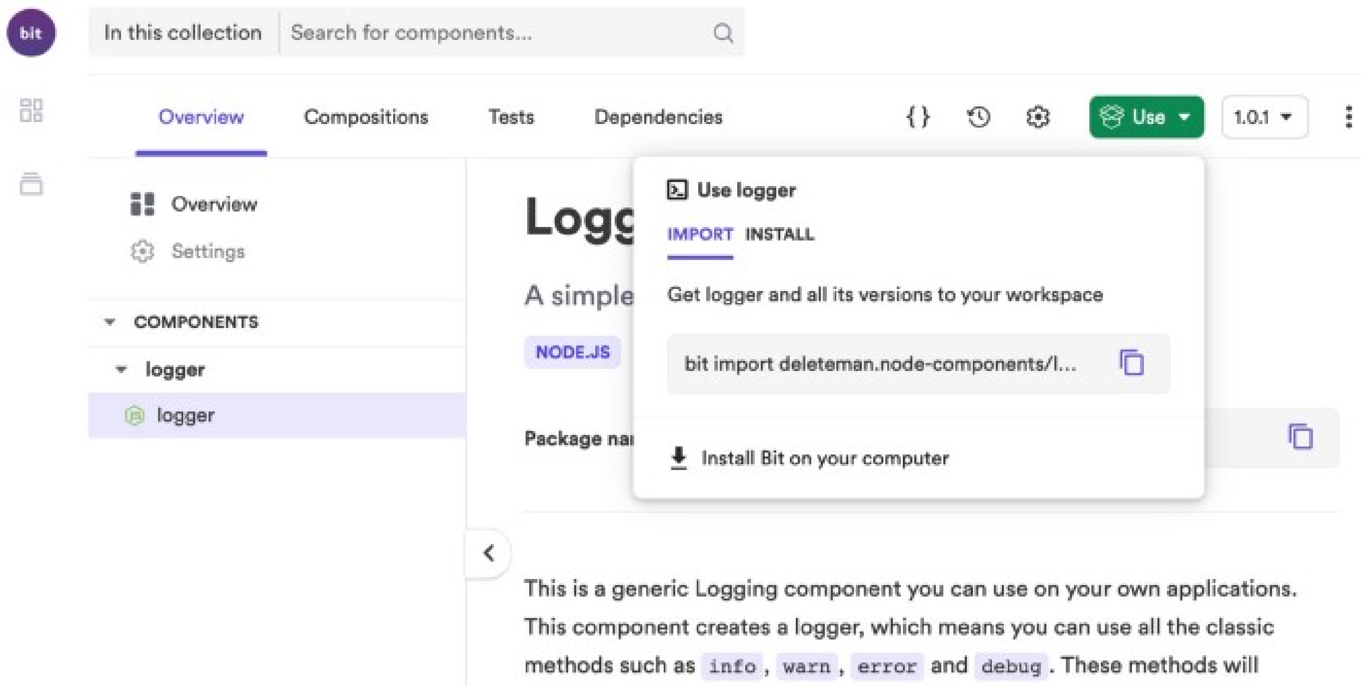
Everyone knows that if you want to share a piece of Node.js code as an individual component with other people, you need to create a package and publish it somewhere — ideally on NPM where people can find it, install it, and use it.

There is nothing wrong with that workflow. It’s been working so far and it’s perfect when you only care about people using it. But what if you want more? What if your intention is for others to use your code, update it as they see fit, and publish it back? This is not such an uncommon use case if you think about companies trying to have standard modules to use across multiple projects.

The only issue? With the currently available tools, you’d need a hybrid between NPM and Git to pull it off. That or a complex workflow. You pick.

And what if you’re dealing with a textbook case of a monorepo gone wrong? A single repository with the code base for your enterprise platform and every single in-house extension around it there as well. Every developer and their uncle are working on that repo, so you either set up a very complex branching model and pray your devs follow it or live your life solving merge conflicts and explaining how Git works to developers. This is not a scenario you want to be in (trust me, I've been there).

Luckily for you, Bit's newest version, code-named Harmony, is here to help.



Get logger and all its versions to your workspace

```
bit import deleteman.node-components/l...
```

This is a generic Logging component you can use on your own applications. This component creates a logger, which means you can use all the classic methods such as `info`, `warn`, `error` and `debug`. These methods will

My Node component exported to a remote scope on Bit.dev

What Is Bit?

If you haven't heard of it yet, Bit is an open-source tool (with native integration to Bit.dev's remote hosting platform) that helps you create and share atomic components. What does this mean? That you can build independent components from scratch or extract sections of your code and share them as independent components on a Bit server (e.g. Bit.dev).

While that sounds an awful lot like NPM, there are some major differences:

- You don't have to extract the code to share it. You can export a component directly from inside your repository. Bit allows you to identify a section of your code as a component and treat it independently from the rest of your project. This, in turn, helps you simplify the sharing process since you don't have to set up a separate repo and rework the way you import those files into your project.

- People importing your components (as opposed to just installing them) can also collaborate on them, modify them, and export them back to the registry. This is incredibly powerful if you're working as a group of teams inside the same organization because you're able to collaborate on the same tool across teams without having to work on a separate project to do it. Importing a Bit component brings down the code and copies it into your working directory (instead of a pesky `npm_modules` folder where you can't do anything with it).

And now with Bit's latest version, team collaboration is easier and faster thanks to additions such as development environments that you can pre-configure and share with everyone so you're all working with the same configuration.

Authoring a Node Component

Notice how I said "component" and not "package." Bit's concept of a component is different from that of a package in the sense that it contains so much more than just the code you're sharing.

If you were to build a standard Node package, you'd want to use tools such as:

- NPM to install dependencies and, of course, share the final result with the community.
- Mocha, Just, or some other testing framework. This is not a hard requirement, but you definitely want to test your code if you expect someone else to trust it.
- Git. Again, not a specifically hard requirement, but it's up there with the big ones. Normally, if you're sharing your packages for others to use, you'll also want them to let you know if there are issues with it or even contribute their ideas and improvements. Git allows you to this by publishing your code in places such as GitHub or Bitbucket.

- A linter. Normally something you'd want to have — especially if you expect people to contribute to your work. This will help make sure every piece of code they add is properly written.

And I could probably keep going, but I think these groups help show that building a package — the proper way, at least — is not trivial or feasible with just your IDE.

However, thanks to Harmony's environments, we can begin to worry less about those tools and focus more on actually building our components.

Let's quickly look at the process you'd go through when extracting a piece of your code into a component and how you'd share it with other teams using Bit Harmony.

Getting started

For a detailed step-by-step guide on how to install it and create your first component, you can check out their official documentation.

Otherwise, I'll assume you've already installed Bit on your system and are ready to get started.

What I'm going to be showing you here is how to extract a piece of code and turn it into a component without affecting your current project. And for that, I'll work with one of my older open source projects, jWhisper (a JSON-WSP compatible library I built a while ago). This is to show you that you don't need to structure your projects in any special way.

After installing Bit on your system, logging in with the CLI tool, and setting up your account on Bit.dev (again, check out their tutorial for those steps), the first thing you need to do is to create a Scope (also known as a collection), which is where your components will live.

You can see here that I've created mine and called it "node-components." I've also marked the "Start with Harmony" checkbox. I created it under my own user instead of my organization, but that's just me.

Now that we got that out of the way, let's get back to our code and create our workspace, which is the first thing you'd want to do. And that is done with a very basic command inside your project's folder:

```
$ bit init --harmony
```

The `--harmony` flag is very important. Otherwise, you'd be creating the workspace with the older version. What you have to care about right now is that two files just got created for you:

1. The `.bitmap` file. Don't get confused by the name. While it sounds like an image, it's just a JSON file where Bit keeps track of all your components and maps their names to their physical locations. You won't have to update this file manually, so just ignore it for now.

2. `workspace.jsonc`. This is the one you want to open. Here, you'll configure everything you need for your project. We'll see more about it in a second. Right now, you need to edit the `defaultScope` key to `yourusername.your-scope`. In my case, that would be `"deleteman/node-components"`.

Let's now take a closer look at what environments are and what we can get from them.

Using environments

An environment is essentially a set of pre-configured tools for you to use during development. If you've used Bit in the past, this is a new concept.

Thanks to environments, you can configure the type of project you're building and the tools you're using to do so (i.e. the package manager required, the linter, the testing library, and more). This is because from now on, the only tool you'll have to worry about is Bit.

Through environments, Bit is abstracting the need for the individual tools and giving you a single point of entry: its CLI.

- Do you need to build your code? Not a problem. You can configure the build pipeline or use the default one.
- Do you need to install your dependencies? Not an issue. Through environments, you can configure the default package manager for your project (be it npm, pnpm, yarn, or whatever) and Bit will take care of calling it when required.
- Do you need to run a linter before building your project? Great, you can configure your favorite linter and add it to the build pipeline provided by the environment.

The entire lifecycle of your component is handled by Bit, and the tools to do so are configured on the environment.

The way you configure your environment is by editing the `workspace.jsonc` file under the `teambit.workspace/variants` section. Make it look like this:

```
"teambit.workspace/variants": {
    "*": {
      "teambit.harmony/node": { }
    }
  }
```

Here, we're telling Bit that we're using the "node" environment for *all* components. But we could use different environments for different folders, which lets you export and handle different default settings inside the same project (hello, flexibility!).

By choosing the "node" environment, we're picking the default behavior Bit understands is best for a Node.js project. Essentially, it picks Jest as the default testing library, understanding that all files ending with `*.spec.*` and `*.test.*` are unit tests. It'll use TypeScript as its default compiler and a few other defaults. You can view the entire list.

If you're unhappy with these defaults or they just don't make sense to you, you can override them yourself. Since this is a basic intro article, I'll leave the advanced stuff for the future. We'll go with the defaults for now.

Before moving forward, you'll have to start the dev server that will take care of multiple things, including installing the required files for our newly configured environment to work:

```
$ bit start
```

This command will take a while, but after it is done, you can navigate to `localhost:3000/` to check out the local workspace UI. It'll be empty right now, but you'll start seeing the components when we start adding them.

Setting our code as a component

To share our code as a component, we need to add it as one using Bit's CLI. The important thing to remember is that a component is not just a single file with the code but rather the code, the documentation, the unit tests, and everything in between. This is why our code needs to be inside an individual folder.

In my case, I'm going to be targeting my logger module, which is just an implementation of a Winston logger.

And I’ll add the folder as a component with the following command:

```
$ bit add lib/logger --namespace logger
```

The `namespace` flag is used to group my components in my workspace and, later on, in their remote scope.

You can go back to your local UI now and you’ll see the update:

That’s our component inside the namespace we gave it. Fantastic! However, there seems to be an issue. The red `1,N` indicates there is something wrong with our component.

If you run:

```
$ bit status
```

You’ll notice that Bit has picked up that it needs Winston to work:

We need to manually add that dependency to our component, and we can do that through Bit:

```
$ bit install winston
```

This will install the dependency using the pre-configured package manager (you can see that on `workspace.jsonc` — the default one is `pnmp`). This will also update the `workspace.jsonc` file, adding the new dependency to the `teambit.dependencies/dependency-resolver` key. And if you restart the dev server, you’ll see on your local UI that the red is gone.

Adding tests

At this point, the component is done and ready to be exported. But before we do that, I want to show you how easy it is to add tests to it.

Note: For the following test to work, you'll also need to install the package `@babel/runtime` using the following command:

```
$ bit install  @babel/runtime
```

This will include it as a dependency in your project. Now you can create a simple test file inside the components folder. I called it `logger.spec.ts` :

I'm essentially mocking Winston and checking if my module is calling the basic `createLogger` function. Not the ideal test, but good enough to show you how Bit handles it.

After adding this file, you can then run `$ bit test` and it'll take care of calling Jest (or the testing framework you decided to configure). The best part is that you also get a graphical update on the local UI:

Notice the Tests tab. It shows the results of your files.

Adding documentation

Documenting your components is essential if you're hoping someone will use them someday. The great thing about Bit is that it has some defaults you can use to create the documentation, and it'll add it to its UI (both local and remote).

We'll create a quick `logger.docs.mdx` file inside our component's folder (note that the `*.docs.mdx` extension is what tells Bit this file is documentation only) that looks like this:

```
---
displayName: Logger Component
description: A simple logger component using Winston
labels: ['node.js', 'winston', 'logger']
---
```

```
This is the actual documentation of my module, you can add all the
markdown you want.
```

And this translates to (minus the last line, the screenshot shows the final result after I put some time into documenting the component):

Notice the title, subtitle, and tags added there as well as the bottom line where you could expand the explanation and add examples and code snippets. The “Package name” field is generated by Bit, and it shows you how you can import the new component inside your code. This is because it has created a symbolic link inside the `node_modules` folder pointing to the component’s actual folder. This helps you abstract your code from the actual physical location of your code.

In fact, you could change your JS code into TypeScript, export the component through Bit, and then change the require statement to pull the file using that package name (in this case `@deleteman/node-components.logger.logger`). Your JS code will still work without any flaws using the TS component (this is because Bit compiles the code and saves the JS output inside the proper folder in `node_modules`).

Exporting the module

Last but definitely not least, you are now ready to share your module with the world. You've extracted the code, added tests, managed to create some basic documentation, and now you can simply use the following lines to tag (i.e. add a version of) your component and export it:

```
$ bit tag --all 1.0.0 --message "first version"
$ bit export
```

When running the first command, you're essentially setting the version number of your component and kind of committing all changes (like if you were using Git). During this process, the tests will be triggered and if they fail, the entire process will be canceled. This is to ensure everything that gets tagged actually works.

The second command will take care of exporting everything into Bit.dev servers (notice that you can create your own self-hosted Bit server and use it internally without needing to have your components on the public cloud). You should see the same documentation and list of components you're seeing locally but on your account. See mine, for example.

Conclusion

Congratulations, you've successfully created a reusable component from within your Node.js project. You started with a single file, turned it into TypeScript (potentially), added documentation, a test file, and shared it with the world without leaving the safety of your project.

This tutorial covered the very basics of all the new potential added by Bit Harmony. I'll cover more in-depth topics in upcoming articles, but feel free to leave questions or suggest scenarios and I'll try to cover them in the next one.

Thanks to Anupam Chugh.

Programming JavaScript Nodejs Web Development Angular