



Published in Bits and Pieces



Fernando Doglio

Follow

Jun 1, 2021 · 12 min read · Listen



Save



Sharing Types Between Backend and Frontend with the BFF Pattern

Given its implementation, it might as well be a love story between a service and a client

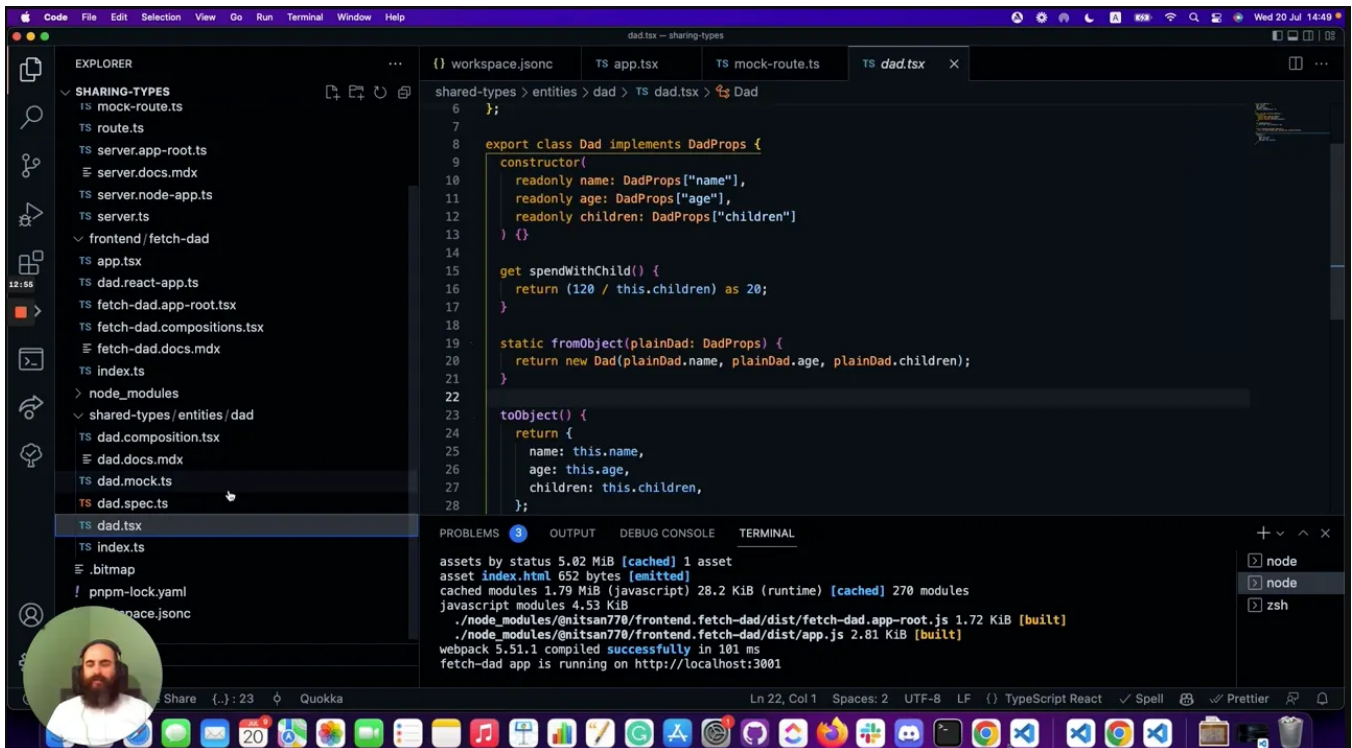


Image by [Cheryl Holt](#) from [Pixabay](#).

Important — The recommended way to share types

We highly recommend sharing types between your frontend and backend using a component. It's much easier, and it ensures nothing in your code will break even as APIs and code are changing. You're welcome :)

Watch:



Code and tutorial:

Sharing Types Between Your Frontend and Backend Applications

Your backend API has been updated to return data of a new type.
The frontend team must be informed to update...

bit.cloud

A contract between a back-end service and a front-end consumer (or client) is usually everything there is to join both worlds. That contract can take the form of a REST API specification, a GraphQL endpoint, or anything really, as long as it tells both parties what to expect from the other.

This, however, is a love story between a Node.js back-end and a React front-end. Living in seemingly different worlds they found a common language to communicate with each other, but that wasn't enough, misunderstandings were still happening, sometimes one would expect the other to say something that the latter wouldn't know how to express. It wasn't until a few years ago, with the generalization of TypeScript (and TypeScript types) that they both started speaking a common language.

Let's check out what the BFF pattern is (no, it's not the Best Friends Forever pattern, as cool as that might sound) and how TS Types can help us craft a solid contract between back and front.

The BFF pattern

Open in app ↗

Get unlimited access



funnier.

Anyway, the point of this pattern is to turn the back-end and the front-end into best friends, even when they don't necessarily see eye to eye. You see (no pun intended), when you develop a back-end microservice that is consumed by multiple client applications, the API needs to be standard, essentially, the same API for all clients. This in theory simplifies adoption and makes it easier for new developers to learn how to use your service.



That being said, there are times when your client applications aren't able/willing to follow your API's contract for whatever reason. This causes extra logic to be added on the front-end to parse and transform your response into something they can actually use.

And here is where the BFF comes into play, instead of having your client apps talking directly to your microservice, they'll talk to a proxy service, which takes care of sending the request to the intended service and translating the response into something the front-end can understand.

Is the BFF pattern better than no pattern? For starters:

1. The client application remains dumb, which is something we tend to want for security and ease of adoption reasons. The dumber the client app has to be, the faster other teams will be able to create client applications. Additionally, the core business logic and the data wrangling needed are all kept secret on the back-end.
2. The added delay from the extra connection on the back-end is expected to be smaller than the impact from the extra resource consumption on the front-end.

But is it perfect? I mean:

1. The architecture of your system is more complex due to yet another moving part (i.e another service). And this example only covers one BFF, but you could potentially build one BFF  652 |  3 | ... application.
2. There is a tight implicit coupling between the back-end and front-end due to this BFF service. Granted, the point of the BFF is to make sure the contract between back and front is exactly what the client needs, reducing the chances of future changes to a minimum. However, those chances are never zero, and a modification either on the back-end or front-end implies a direct change on the other. So, coupling.

Can you live with these negative points? Then the BFF is for you. Are they too much of a problem or are you using a monolithic architecture? Then forget about it, move on and implement the BFF once you're working with 20th-century tech (at least).

What about shared types?

Shared types are a specialization of the BFF pattern specifically for TypeScript, which allows you to share the same type definition between front-end and back-end code.

This is fantastic because you're literally avoiding definition problems between two teams by providing both with a single source of truth.

For this to work, you need to find a way to turn your type definition into a separate module that can be imported by both parties. This can be done in many ways, to be

honest:

- If you have a monorepo you can simply import the definition from a generic path to both projects. This approach however couples the life-cycle of your types to both projects at the same time because by making a change in a file (i.e the type definition), you're affecting the entire monorepo. Granted, with good enough git-foo you can work around those problems, but it's a bit headache.
- You can turn your type definition into an NPM module, publish it, and then have both, your front-end and back-end code install it as a dependency. This is great after the fact, but during initial development, you'll have to publish the module for things to work, otherwise, you'll have to have local import paths and then refactor them into absolute ones. And even after the initial development is done, any changes on the local version of your types module will need to be published before any of the other projects can see it. This can become a bit of a hustle.

Finally, we have a 3rd and more interesting option that involves using Bit to build the whole thing.

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 independent components. These are components (or modules) that are independently developed, versioned, and collaborated on.

teambit/bit

Bit is a tool for composing modern applications of independent components. It extends the benefits of micro-services to...

github.com

Bit: The platform for the modular web

Bit is a standard infrastructure for components. It's everything your teams need to enjoy autonomous development...

bit.dev

You can either author new independent components from scratch or gradually extract components from an existing codebase.

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

- You don't have to physically extract the code to version it, share it, and collaborate on it, independently. 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 that point on. 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 their 'remote scope' (remote component hosting).

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 downloads the code and copies it into your working directory. It also generates a corresponding package in your `node_modules` directory. This package is regenerated as you change the source code (in the working directory). That way you're able to consume it using an absolute path that works in all contexts (and will even work if you choose to install the component's package without importing it).

And that is what we're going to be taking advantage of today, the fact that you can turn a local setup into a multi-repo situation without worrying too much about anything really.

Building a shared type module

As a practical way of showing the beauty that is the shared type pattern, I'm going to build 3 components:

- A type definition, of course. This is the one that the other two components will be using on their side. This component will also export a mock with fake data used by the front-end component when there is no connectivity to the back-end.

This last part is merely for example purposes, you can probably find better ways to solve that problem on your side.

- A back-end service that exports a function ready to create an HTTP web server and serve a fixed response for every request. Given this is an example the service is going to be very simple, however, it should be enough to prove the point. As part of the code, it'll use the shared type definition to make sure the response has the right format.
- A React component that will pull the data from the service (or use the mocked data) and render it on the page. Of course, as part of its code this component will also be using the shared type definition.

Let's get building.

The initial setup

Assuming you've installed Bit and that you've logged in on the platform already, please take 2 minutes to create a collection (also known as 'scope') and call it "bff" (you can call it whatever you like, that's the name I went with).

With that in mind, initialize the workspace:

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

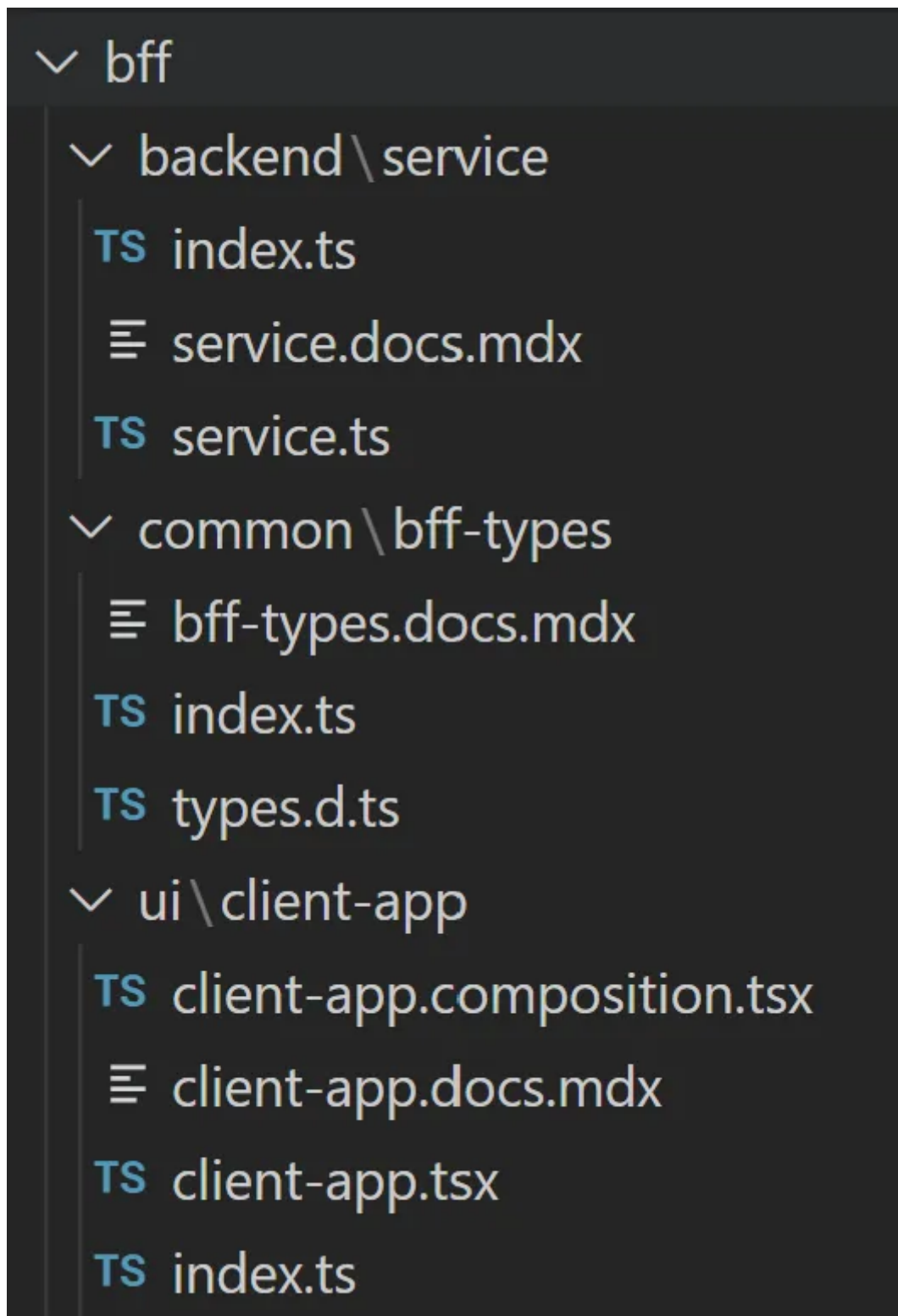
After that, edit the `workspace.jsonc` file and make sure the `defaultScope` value is set to `[username].bff` in my case that looks like `"deleteman.bff"` .

Now, you can go back to the terminal and create the first component:

```
$ bit create react-component ui/client-app
```

This will create a new component using the React template, which will include a bit of boilerplate code as well as it'll create the folder structure e need for new components (it should've created a `bff` folder and a `ui/client-app` folder inside it).

The next two components will have to be created manually because Bit still doesn't have templates for them:



Just make sure you replicate the above structure, having our share types inside the `common/bff-types` folder and our service inside the `backend/service` folder.

You must create these `index.ts` files in both, since they'll act as main entry points once we turn these folders into components.

The type definition for this example is going to be straightforward:

Notice I included both files inside the above snippet.

The code for the service is no more complex, to be honest:

Notice I'm always returning a JSON with 2 elements inside and I'm using the type definition to make sure I create the structure to be returned without missing any attribute. Also, the exported function, `newServer` is creating a basic HTTP server (no express or anything) and setting up the required CORS headers for you to properly test the example server from your localhost.

Finally, the react component looks like this:

Lots of code, but really, I'm not doing anything too complex here.

By now you probably noticed the two `import` lines in both, the server component and the react one:

```
import {ServiceResponseType, PersonType, ServiceResponseMocks} from  
'@deleteman/bff.bff-types';
```

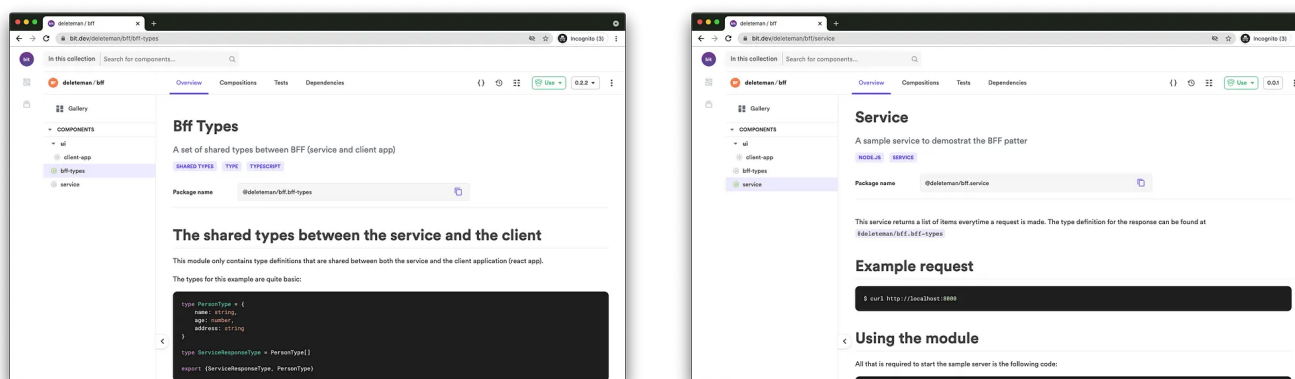
This would normally imply that I've already published the shared types module and now I'm importing it. However, thanks to Bit, that is not required. All I had to do was turn that module into a component, and then Bit will create a symlink inside the

`node_modules` folder pointing to my local copy. This in turn allows me to work on the other components as if everything was already published.

Turning the code into components

Remember, so far we only created one “official” component, the React component. The other two so far are only folders with code, we need Bit to be aware of them, and we can do that with:

```
$ bit add bff/backend/service
$ bit add bff/common/bff-types
```



<https://bit.dev/deleteman/bff>

Now, we’ve told Bit that these two folders contain components as well and that we should be tracking them. This should’ve changed the content of the `.bitmap` file (you don’t need to touch this file, just make sure it as an entry for each module).

One last thing to do before we can call it “done”, is to go back to the `workspace.jsonc` file and edit the `variants` key to make sure we’re using the right environment for each component. Remember, we’re dealing with a react component on one side, a Node.js component and a generic one, so Bit needs to be aware of that. Just open the file and make sure the `teambit.workspace/variants` section looks like this:

```
"teambit.workspace/variants": {
  "bff/ui/client-app": {
    "teambit.react/react": {}
  },
  "bff/common/*": {
    "teambit.react/react": {}
  },
}
```

```
"bff/backend/service": {  
  "teambit.harmony/node": {}  
}  
}
```

Now you can run `bit start` and have the local dev server started. This dev server will give you all the information you need about your components and will show you how their docs will look like once published.

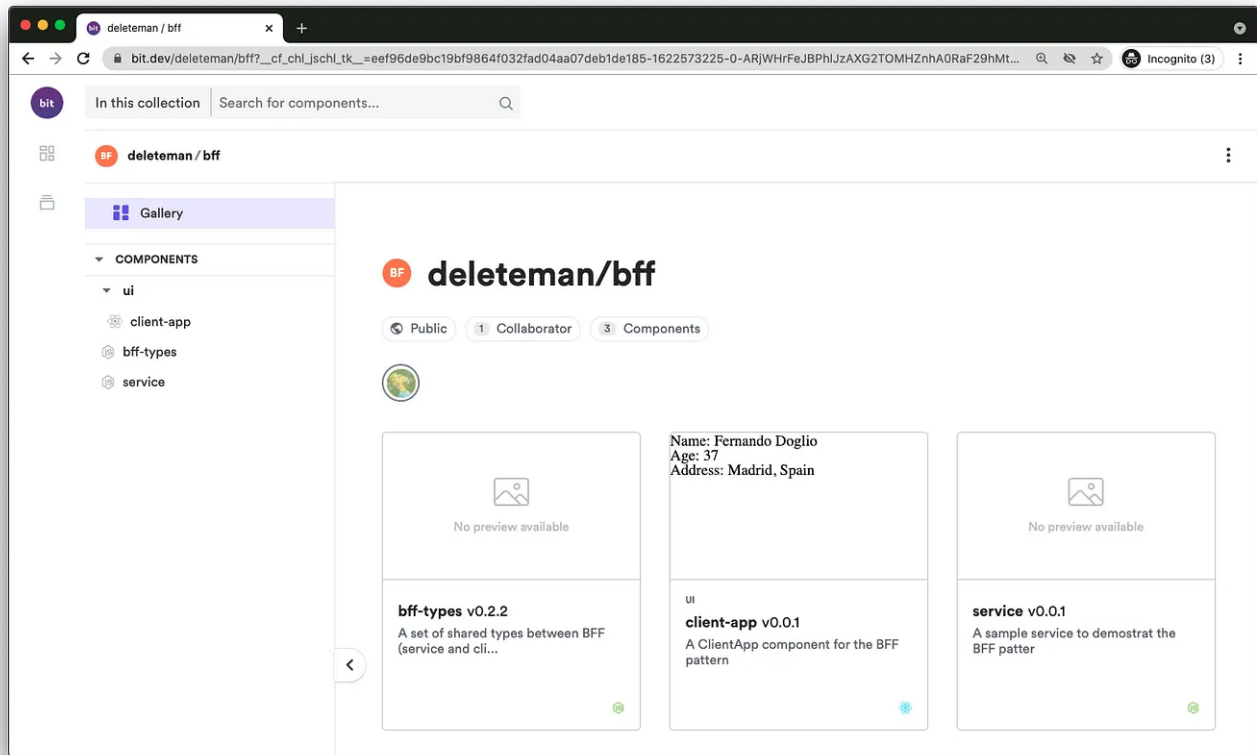
An interesting nugget of information we can get from here is the import URL. You can get it by clicking on a component once the server has started, that is the first thing you'll see and it's autogenerated for you.

Like I said before, by using that URL you don't have to worry about importing it directly or even caring where it points, Bit will solve that dependency for you and you can develop your components under the assumption that this common code is already available.

If you'd like to share this work with others, you can also do that through Bit:

```
$ bit tag --all #1  
$ bit export    #2
```

Step #1 will tag all components, which means you'll give them a version to be exported under. And step #2 will export the components into the global Bit.dev repository. I've shared [mine here](https://blog.bitsrc.io/sharing-types-between-backend-and-frontend-with-the-bff-pattern-553872842463) in case you'd like to take a look.



My remote scope with all shared independent components

What have we done?

Through the use of Bit and the Shared types pattern, we've managed to connect a back-end service with a front-end application making sure they both share the same type definition.

But what else happened there?

1. For starters, we developed and published 3 modules without even having to worry about the TS compiler, WebPack, NPM or any other tool other than Bit for our workflow. That is why I chose Bit and why it is such a useful tool. It abstracted all the steps required to create the 3 components, even when they were working for 3 different environments.
2. We worked with a local dependency which is meant to be an external dependency without even noticing the difference. Every update you make to the local version of the BFF-Types component will be picked up by the other two without having to go through the whole exporting process again.

In the end, the BFF pattern and the shared type patterns are just a way of saying: given a particular use case where a certain degree of coupling is accepted, let's agree on a contract between client and server and overwrite any other generic contract we could've had in the past.

This is between the server and A client, remember that, the BFF couples one single client application (or type of application) to a specialized version of the server. It doesn't negate the generic contract other clients could be using with the original version of the service.

Have you done something like this before? Were you aware of the BFF pattern? Are you OK with the added coupling between front and back or do you prefer to have your services and clients completely separated from each other? Leave a comment below and let's discuss it!

Learn More

The BFF Pattern (Backend for Frontend): An Introduction

Get to know the benefits of using BFF pattern in practice

blog.bitsrc.io

Independent Components: The Web's New Building Blocks

Why everything you know about microservices, micro frontends, monorepos, and even plain old component libraries, is...

blog.bitsrc.io

Creating a Component Library

Create a scalable UI library that others would love to use.

blog.bitsrc.io

--	--

[JavaScript](#)[Typescript](#)[React](#)[Nodejs](#)[Bit](#)

Enjoy the read? Reward the writer.^{Beta}

Your tip will go to Fernando Doglio through a third-party platform of their choice, letting them know you appreciate their story.

Give a tip

Get an email whenever Fernando publishes a new article.

Hey there, did you like what you read? Then subscribe so you can get notified next time I publish a new article! I promise you won't be disappointed!

Emails will be sent to bernardo.learningtechnology@gmail.com. [Not you?](#)



Subscribe