Microsoft Azure

Dependency Injection in JavaScript 101

Jeremy Likness Jan 2 *Updated on Mar 05, 2019* • 6 min read

#javascript #angular #react #dependencyinjection

In my article and presentation "The 3 D's of Modern Web Development" I explain what I believe are critical elements for success in modern JavaScript frameworks.

> *Wait, you didn't see my presentation?* That's OK ... when you have just under an hour of time to invest, I believe you will receive value by watching it here.

Dependency Injection is one of those elements. I find developers often struggle to understand what it is, how it works, and why it's even necessary.

I learn by doing and hope a simple code example will help explain. To begin with, I wrote a very small application that assembles and runs a car. The dependencies look like this:

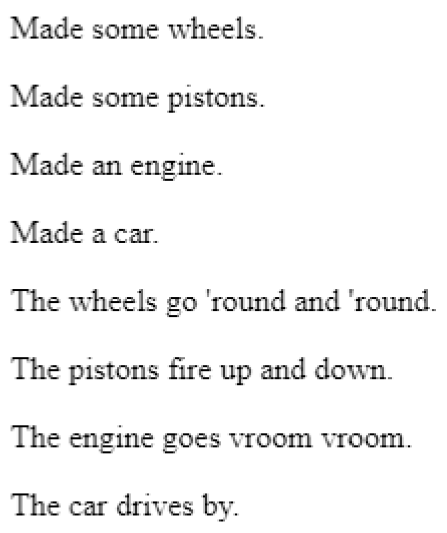
```
Car
|
|--Engine
|   |
|   |--Pistons
|
|--Wheels
```

Think of the parts as dependencies between components. You can see the code and run it interactively here: https://jsfiddle.net/jeremylikness/gzt6o1L5/.

```
function Wheels() {
  this.action = () => log("The wheels go 'round and 'round.");
  log("Made some wheels.");
}

function Pistons() {
  this.action = () => log("The pistons fire up and down.");
  log("Made some pistons.");
}

function Engine() {
  this.pistons = new Pistons();
  this.action = () => {
    this.pistons.action();
    log("The engine goes vroom vroom.");
  };
  log("Made an engine.");
}

function Car() {
  this.wheels = new Wheels();
  this.engine = new Engine();
  this.action = () => {
    this.wheels.action();
    this.engine.action();
    log("The car drives by.");
  };
  log("Made a car.");
}

var car = new Car();
car.action();
```

index.js hosted with ❤ by **GitHub** **view raw**

The output should be what you expected.



```
Made some wheels.

Made some pistons.

Made an engine.

Made a car.

The wheels go 'round and 'round.

The pistons fire up and down.

The engine goes vroom vroom.

The car drives by.
```

Great! So far, we have something that works, and we didn't even have to install a fancy framework. So, what's the problem?

The code works but is very simple. The problems come into play in a much larger application. Imagine having hundreds of components with dependencies ... now you will run into some issues:

1. The components depend directly on each other. If you attempt break each component (wheel, piston, etc.) into its own file, you will have to ensure everything is included in the right order for it to work. If you create or include the engine before defining the piston, the code will fail.
2. You cannot develop components in parallel. The tight coupling means it's not possible to have a developer working on engines while another is working on pistons. (For that

matter, you can't easily make an empty set of objects as placeholders for pistons while you work on engines).

3. The components create their own dependencies so there is no way to effectively test them without dependencies. You can't easily swap out "piston" with "test piston." In web apps this is important for unit tests. For example, you want to be able to mock web API calls rather than make real HTTP requests in your tests.

A little of refactoring will solve the third problem. Have you heard of a pattern called *Inversion of Control*? It is a simple pattern. Right now, the components are in control of their own dependencies. Let's invert that, so the components are no longer in control. We'll create the dependencies elsewhere and inject them. Inversion of control removes the direct dependencies, and dependency injection is how instances are passed to components.

To keep it simple, I'll just include the code that changed. Notice that instead of directly creating dependencies, the dependencies are now passed into the constructor functions. You can view the entire app and run it interactively here: https://jsfiddle.net/jeremylikness/8r35saz6/

```
function TestPistons() {
  this.action = () => log("The test pistons do nothing.");
  log("Made some test pistons.");
}

function Engine(pistons) {
  this.pistons = pistons;
  this.action = () => {
    this.pistons.action();
    log("The engine goes vroom vroom.");
  };
  log("Made an engine.");
}

```

```
function Car(wheels, engine) {
  this.wheels = wheels;
  this.engine = engine;
  this.action = () => {
    this.wheels.action();
    this.engine.action();
    log("The car drives by.");
  }
  log("Made a car.");
}

var pistons = new Pistons();
var testPistons = new TestPistons();
var wheels = new Wheels();
var engine = new Engine(pistons);
var testEngine = new Engine(testPistons);
var car = new Car(wheels, engine);
car.action();
testEngine.action();
```

index.js hosted with ❤ by **GitHub** **view raw**

Now we've applied the *Inversion of Control* pattern and are doing some simple *Dependency Injection*. However, we still have a problem in a large code base. The previous issues (#1 and #2) have not been addressed. Notice that the objects must be created in the right order. Including or creating them out of order will result in failure. This makes it complicated to develop in parallel or out of sequence (and believe me, it happens with larger teams). A new developer on your team will have to understand all the dependencies to instantiate a component in their own code.

Again, what we can do?

The solution is to bring in an IoC (short for Inversion of Control) container to manage Dependency Injection. There are many types of containers, but here's how they typically work:

- You get one global instance of the container (you can have multiple containers but we'll stick with one to keep it simple)
- You register your components with the container
- You request components from the container, and it handles dependencies for you

First, I'll include a very small library I wrote named jsInject. This is a library I wrote specifically to learn about and understand dependency injection. You can read about it here: Dependency Injection Explained via JavaScript, but I recommend you wait until *after* this article. After you are comfortable with DI and IoC, you can dig deeper to see how I created the container. The library does many things but, in a nutshell, you pass it a label and a constructor function to register a component. If you have dependencies, you pass an array with those dependencies. Here is how I define the `Pistons` class. Notice the code is almost 100% the same as the last iteration, except for the line of code that registers the component.

```
function Pistons() {
    this.action = () => log("The pistons fire up and down.");
    log("Made some pistons.");
  }

  $jsInject.register("pistons", [Pistons]);
```

index.js hosted with ❤ by **GitHub** **view raw**

To get an instance of the class, instead of creating it directly, you "ask" the container for it:

```
var pistons = $jsInject.get("pistons");
```

Easy enough! What's important to understand is that you can now develop in parallel and independently. For example, here is the `Engine` definition. Notice it depends on pistons but doesn't explicitly reference the implementation and simply references the label.

```
function Engine(pistons) {
  this.pistons = pistons;
  this.action = () => {
    this.pistons.action();
    log("The engine goes vroom vroom.");
  };
  log("Made an engine.");
}

$jsInject.register("engine", ["pistons", Engine]);
```

index.js hosted with ❤ by **GitHub** **view raw**

In fact, in the example I created, I define the `Car` and `Engine` classes *before* their dependencies, and it's completely fine! You can see the full example here (the `$$jsInject` library is included at the bottom in minified code): https://jsfiddle.net/jeremylikness/8y0ro5gx/.

The solution works, but there's an added benefit that may not be obvious. In the example I explicitly register a "test engine" with "test pistons." However, you could just as easily register the "pistons" label with the `TestPistons` constructor, and everything would work fine. In fact, I put the registrations with the function definitions for a reason. In a full project, these might be separate components. Imagine if you put the pistons in `pistons.js` and the engine in `engine.js` . You could do something like this:

```
main.js
--engine.js
```

```
--pistons.js
```

That would work to create the engine. Now you want to write unit tests. You implement `TestPiston` in `testPiston.js` like this:

```
function TestPistons() {
  this.action = () => log("The test pistons do nothing.");
  log("Made some test pistons.");
}
$jsInject.register("pistons", [TestPistons]);
```

testPistons.js hosted with ❤ by **GitHub** **view raw**

Notice that you still use the label "pistons" even though you register the `TestPistons` constructor. Now you can set up this:

```
test.js
--engine.js
--testPistons.js
```

Boom! You're golden.

DI isn't just good for testing. The IoC container makes it possible to build your components in parallel. Dependencies are defined in a single place instead of throughout your app, and components that depend on other components can easily request them without having to understand the full dependency chain. "Car" can request "engine" without knowing that "engine" depends on "pistons." There is no magic order to include files, because everything gets resolved at run time.

This is a very simple example. For a more advanced solution, take a look at Angular's dependency injection. You can define different registrations (called `Providers`) such as types (via TypeScript), hard-coded values and even factories that are functions that return the desired value. You can also manage *lifetime* or *scope*, for example:

- Always give me the same instance when I request a car (singleton)
- Always give me a new instance when I request a car (factory)

As you can see, although people often use them interchangeably, Inversion of Control (IoC) and Dependency Injection (DI) are related but not the same thing. This example demonstrated how to implement IoC, how to add DI, and how to use an IoC container to solve problems. Do you feel you have a better understanding? Any feedback or questions? Let me know your thoughts in the comments below.

Regards,

Jeremy Likness

Note from the DEV admins:

Now reaching **over 3 million visitors** per month, **DEV** is the fastest growing software development community in the world. We think it's the most awesome and we're working hard to keep it that way.

It's free, open source, devoted to the open web, and will never have popups or a pay wall.

Get Started Now

Jeremy Likness ϟ **+ FOLLOW**

Hi! I am a mentor, author, and international speaker with a passion to empower developers to be their best. I focus on cloud development and specialize in enterprise web development.

@jeremylikness jeremylikness JeremyLikness blog.jeremylikness.com

Microsoft Azure

Any language. Any platform.

@azure aka.ms/dev-to

Add to the discussion

PREVIEW SUBMIT

▼

Heiker Jan 4

Some say that an IoC container isn't necessary in javascript since it already has a module system. I don't exactly agree with those people because when you use a IoC container the functions/classes that you make have a diferent "design". Still, the container does the job of a module system, which make it seem like is a bit redundant.

3 REPLY

▼

Jeremy Likness ϟ Jan 4

Absolutely. Your point is a good one because JavaScript doesn't adhere to the same rules more strictly typed, object-oriented languages do. Angular chose the approach of injecting dependencies but JavaScript (and TypeScript, for that matter) have better support for aspect-oriented programming in my opinion. To illustrate, for C# if I want a logger the common way to do this is to inject it. I pass my logger instance into the component and then I can swap it out as needed. I can't take a static dependency on Logger.Something because then I've have a hard-coded reliance on Logger. In JavaScript, however, I can just call to Logger.Something because "Logger" is changeable depending on what module I load. I could probably be convinced that in the JavaScript world, we can talk about dependencies strictly by discussing modules and not have to bring IoC/DI into the picture at all unless we're using a framework that takes that approach. Might write another article to expand on that in more detail, thanks for taking the time to respond!

♡ 2 REPLY

▼

Jilles van Gurp Jan 23

Testability is a topic that has always been a bit problematic in the javascript world; mainly because there are a lot of inexperienced programmers using it as their first language and because it has a history of people copy pasting fragments around on web pages.

There are plenty of tools to write tests for javascript but frontend js has this tendency to turn into an untestable mess and there is this misguided concept that it is simply impossible, or worse, redundant. One problem is that people focus on the wrong type of tests and don't understand the difference between unit and integration test. You see people trying to run scenarios against a mock that basically fires up the whole application. This is a lot of work and those tests can be very brittle. I've talked to many engineers that got all excited about the prospect of their test library firing up a headless browser; just so they can test the side effects of events on lambdas on the dom. Don't do that. It's stupid. Make your side effects unit testable and you don't need a browser, or a dom.

Unit testing is where the action is. Unit testing is very straightforward provided your code is testable. This requires inversion of control. Any time that you have code that initializes stuff as a side effect of being used, or imported, you are creating a testability problem. You can no longer consider the unit in isolation since it fires up stuff that fires up more stuff, etc.

Lambda functions make this even worse. Now you have a class with a constructor that constructs other things than itself (simple guideline: constructors must not do work) and then inserts lambdas in those things that do the actual things you need to test. All inside a constructor that gets called from another constructor. Add promises to the mix and you have a perfect horror show of asynchronous side effects of a things being created that may or may not fire events, etc. Simulating all of that in a unit test is hard. This code is effectively hard to test.

Another problem in many javascript projects is the (ab)use of global variables that are accessed from all over the place. Magic elements in the DOM, a library declaring a const with a singleton instance (aka. a global variable) for which the reference is hardcoded everywhere, magic elements in the DOM that contain data that are passed around via imports. These are problems you have to work around when writing a test when you might want to have an alternate value.

The way out is simple: design for testability. IOC is one of several tools you can use and a crucial one but step 0 is acknowledging that what some people claim is elegant is in fact inherently untestable and therefore something needs to change.

♡ 3 REPLY

▼

Jeremy Likness Jan 23

Thank you for this insightful reply! It makes a lot of sense. I would also add that doing testing the right way (and not just to check a box) transforms the way developers approach code. It's like words: tough for a toddler to communicate with only a few, but an adult can convey very complex thoughts by using the right words as building blocks. Having a test-conscious approach and leveraging patterns like IoC improve your "developer vocabulary" and helps simplify the expression of code.

♡ 1 REPLY

▼

Brian Jan 4

Great post, appreciate the simple explanation.

Just checking, is this line a typo on "piston"?
var piston = $jsInject.get("piston");

Just wondering if it should be "pistons" since that is the label you registered it with or if I'm misunderstanding. Thanks!

♡ 2 REPLY

▼

Jeremy Likness Jan 4

Great catch! It was a typo and I updated the article. Thanks for the good eyes on this.

♡ 1 REPLY

▼

Leon Ormes Aug 6

Jeremy,
Thanks for this post. It actually made me excited like Christmas. I have been trying to work out how and why this is useful and your example clears up all doubts I might have had.
Next, how do I get my team to adopt this pattern? :)

1 REPLY

Jeremy Likness

Aug 9

I always prefer to lead by example and show the value it provides so they *want* to adopt it.

1 REPLY

Rodrigo Assis

Jan 25

Incredible! But one doubt: when you'll use ".register", you've to specify first the label and then the class, right?

1 REPLY

Jeremy Likness

Jan 25

Correct. In the simple DI example you pass a label (string), then an array that should have the function constructor, factory, or instance as the last element.

2 REPLY

Rodrigo Assis

Jan 25

Great! And in this case: "$jsInject.register("engine", ["pistons", Engine]);", you have to pass the "pistons" in the array too because "Engine" receives "pistons" as a parameter, right? And then the last element would be "Engine" cuz is the "current class".

1 THREAD

Jeremy Likness

Jan 25

You got it! The important part of the illustration is that "pistons" is just a label, not an implementation, so you have flexibility to define it however you see fit elsewhere.

2 THREAD

Rodrigo Assis

Jan 25

Oh, I get it! Then I'll use this label on "$jsInject.get()". Amazing!! Thanks ;)

2 REPLY

Sung M. Kim

Jan 3

Thanks for the post Jeremy 👋

A question I've been mulling over while reading.
Can the terms IoC and DI be used interchangeably?

2

REPLY

Jeremy Likness ⚡

Jan 3

My pleasure! The easiest way for me to summarize it is this:

Inversion of Control is a concept/abstraction and Dependency Injection is a specific implementation of that abstraction.

2

REPLY

Sung M. Kim

Jan 3

Ah. so IoC is a blueprint and DI is like a house.

Thanks, Jeremy 👊

1

REPLY

code of conduct - report abuse

Classic DEV Post from Jun 23

What Advice Would You Give Your 20-year-old Self?

Helen Anderson

If you could go back in time, what advice would you give your 20-year-old self?

78 127

Another Post You Might Like

Decorators do not work as you might expect 🤔

Dominic Elm

❤ 74 💬 6

Another Post You Might Like

scrollIntoView is the best thing since sliced bread

Steve Belovarich

Making elements scroll into view used to be hard, especially with animation. No...

❤ 71 💬 6

(a == 1 && a == 2 && a == 3) === true - Wait, hold on...
EmNudge - Nov 26

My Youtube controller Chrome extension
Maroun Baydoun - Nov 26

5 reasons why learning Javascript is a great idea
Duomly - Nov 26

7 best JavaScript projects to master your skills
Areknawo - Nov 26
