# Dependency Injection ＆ Providers

## Overview

As our application grows beyond one module then we need to deal with the issue of dependencies.

When module A in an application needs module B to run, then module B is a



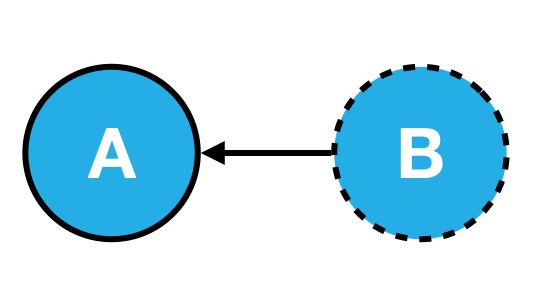
**What is a dependency?**

*dependency*

of module

A

.



Realistically when writing applications we can’t get away from building numerous dependencies between parts of our code.

For example lets imagine we have a class of EmailSender, like so:

class MailChimpService extends EmailService { }

class EmailSender {

emailService: EmailService;

constructor() {

this.emailService = new MailChimpService("APIKEY12345678910");

}

sendEmail(mail: Mail) {

this.emailService.sendEmail(mail);

}

}

emailSender = new EmailSender();

emailSender.sendEmail(mail);

Whats wrong with the above code?

**Inflexible**

Hard to re-use in other configurations.

It hardcodes MailChimpService as the email service that *actually* sends the email.

How would you use this class if you wanted to use another email provider?

**Hard to test**

How can you test the above code?

Calling sendEmail(mail) sends a real email to a real email address using an external service we have no control over.

How do we test that calling sendEmail really sends an email?

**Brittle**

Hard to maintain.

If we changed our API key we need to make sure it’s changed in *every* instance we’ve use the MailChimpService.

Even if we put the API key in a global config variable what if the MailChimpService changed *how* authentication happens and now it uses a *username/password* combination.

The above is described as *tight coupling*. The EmailSender class is said to be *tightly coupled* with the MailChimpService class. This makes the code *inflexible*, *hard to test* and *brittle*.

Now we can’t get away from the fact that the EmailSender class needs the MailChimpService class to function. MailChimpService is a *dependency* of EmailSender.

But we can change the above code so that it’s *easy to reuse*, *easy to test* and *easier to maintain*.

class MailChimpService extends EmailService { }

class EmailSender {

emailService: EmailService;

constructor(emailService: EmailService) {

①

this.emailService = emailService;

}

sendEmail(mail: Mail) {

this.emailService.sendEmail(mail);

}

}

emailSender = new EmailSender(new MailChimpService());

emailSender.sendEmail(mail);

① The emailService is now passed *into* our class via the constructor.

Previously the EmailSender constructor was responsible for creating an *instance* of its dependency the MailChimpService.

Now *something else* is responsible for creating the *instance* of MailChimpService and then passing it into the EmailSender via it’s constructor.

So if we wanted to create an instance of the EmailSender class we now need to pass in all the required dependencies in the constructor.

The dependencies are now said to be *decoupled* from our EmailSender class, how does this help us with our 3 points:

**Flexible/Easier to re-use**

We can *re-use* the EmailSender class but with a different email service.

For example if we wanted to use SendGridService instead of MailChimpService. As long as SendGridService still has a function with the signature sendEmai(mail) we can pass into the EmailSender constructor an instance of SendGridService instead of MailChimpService, like so:

emailSender = new EmailSender(new SendGridService());

**Easier to test**

Following on from the above we can now *test* our EmailSender class much easier.

We can pass in a *dummy* class which doesn’t actually send emails however *does* let us check to see if the sendEmail function was called, like so:

MockedEmailService extends EmailService {

mailSent: boolean = false;

sendEmail(mail: Mail) {

this.mailSent = true;

}

}

let mockService = MockedEmailService()

emailSender = new EmailSender(mockService);

if (mockService.mailSent === true) { ... }

**Easier to maintain**

Since the EmailSender class is not responsible for creating concrete instances of the email service if, for instance, the MailChimpService required some new configuration then the EmailSender class isn’t affected.

As long as the MailChimpService implements the sendEmail function, *how* it’s constructed and functions internally is of no concern to the EmailSender class.

This idea of moving the responsibility of creating concrete instances of dependency’s to something else is called *Inversion of Control*, or *IoC*.

The specific design pattern for implementing IoC above is called *Dependency Injection*, we injected the dependencies of EmailSender in the constructor.

Dependency injection is an important application design pattern it’s used not only in Angular but throughout software development as a whole.

Angular has its own dependency injection framework, and we really can’t build an Angular application without it. It’s used so widely that almost everyone just calls it *DI*.

**Components**

The DI framework in Angular consists of 4 concepts working together:

**Token**

This uniquely identifies something that we want injected. A *dependancy* of our code.

**Dependancy**

The actual code we want injected.

**Provider**

This is a map between a *token* and a list of *dependancies*.

**Injector**

This is a function which when passed a *token* returns a *dependancy* (or a list of dependencies) === Summary

In this section you will learn:

* How the Angular DI framework works under the covers.
* What are injectors & child injectors.
* What function do the @Inject and @Injectable decorators play in the DI framework.
* What are the different types of dependencies we can inject in Angular.
* How to configure DI in Angular with Angular module providers, component providers and component view providers.