Dependency Injection

As our programs grow in size, parts of the app need to communicate with other modules. When module `A` requires module `B` to run, we say that `B` is a *dependency* of `A`.

One of the most common ways to get access to dependencies is to simply `import` a file. For instance, in this hypothetical module we might do the following:

```
// in A.ts
import {B} from 'B'; // a dependency!

B.foo(); // using B
```

In many cases, simply `import`ing code is sufficient, but other times we need to provide dependencies in a more sophisticated way. For instance, we may want to:

- substitute out the implementation of `B` for `MockB` during testing
- share a *single instance* of the `B` class across our whole app (e.g. the *Singleton* pattern)
- create a *new instance* of the `B` class every time it is used (e.g. the *Factory* pattern)

Dependency Injection can solve these problems.

Dependency Injection (DI) is a system to make parts of our program accessible to other parts of the program - and we can configure how that happens.

One way to think about "the injector" is as a replacement for the `new` operator. That is, instead of using the language-provided `new` operator, Dependency Injection let's us configure *how* objects are created.

The term Dependency Injection is used to describe both a design pattern (used in many different frameworks) and also the *specific implementation* of DI that is built-in to Angular.

The major benefit of using Dependency Injection is that the client component needn't be aware of ***how* to create the dependencies**. All the client component needs to know is how to *interact* with those dependencies. This is all very abstract, so let's dive in to some code.

How to use this chapter

This chapter is a tour of Angular DI system and concepts. You can find the code for this chapter in `code/dependency-injection`.

While reading this chapter, run the demo project by changing into the project directory and running:

```
npm install
npm start
```

As a preview, to get Dependency Injection to work involves configuration in your `NgModules`. It can feel a bit confusing at first to figure out "where" things are coming from.

The example code has full, runnable examples with all of the context. So if you feel lost, we'd encourage you to checkout the sample code alongside reading this chapter.

Injections Example: `PriceService`

Let's imagine we're building a store that has `Products` and we need to calculate the final price of that product after sales tax. In order to calculate the full price for this product, we use a `PriceService` that takes as input:

- the **base price** of the `Product` and
- the **state** we're selling it to.

and then returns the final price of the `Product`, plus tax:

code/dependency-injection/src/app/price-service-demo/price.service.1.ts

```
export class PriceService {
  constructor() { }

  calculateTotalPrice(basePrice: number, state: string) {
    // e.g. Imgine that in our "real" application we're
    // accessing a real database of state sales tax amounts
    const tax = Math.random();

    return basePrice + tax;
  }

}
```

In this service, the `calculateTotalPrice` function will take the `basePrice` of a product and the `state` and return the total price of product.

Say we want to use this service on our `Product` model. Here's how it could look without dependency injection:

code/dependency-injection/src/app/price-service-demo/product.model.1.ts

```
import { PriceService } from './price.service';

export class Product {
  service: PriceService;
  basePrice: number;

  constructor(basePrice: number) {
    this.service = new PriceService(); // <-- create directly ("hardcoded")
    this.basePrice = basePrice;
  }

  totalPrice(state: string) {
    return this.service.calculateTotalPrice(this.basePrice, state);
  }
}
```

Now imagine we need to write a test for this `Product` class. We could write a test like this:

```
import { Product } from './product';

describe('Product', () => {

  let product;

  beforeEach(() => {
    product = new Product(11);
  });

  describe('price', () => {
    it('is calculated based on the basePrice and the state', () => {
      expect(product.totalPrice('FL')).toBe(11.66); // <-- hmmm
    });
  })

});
```

The problem with this test is that we don't actually know what the exact value for tax in Florida (`'FL'`) is going to be. Even if we implemented the `PriceService` the 'real' way by calling an API or calling a database, we have the problem that:

- The API needs to be available (or the database needs to be running) and
- We need to know the exact Florida tax at the time we write the test.

What should we do if we want to test the `price` method of the `Product` *without* relying on this external resource? In this case we often *mock* the `PriceService`. For example, if we know the *interface* of a `PriceService`, we could write a `MockPriceService` which will always give us a predictable calculation (and not be reliant on a database or API).

Here's the interface for `IPriceService`:

code/dependency-injection/src/app/price-service-demo/price-service.interface.ts

```
1  export interface IPriceService {
2    calculateTotalPrice(basePrice: number, state: string): number;
3  }
```

This interface defines just one function: `calculateTotalPrice`. Now we can write a `MockPriceService` that conforms to this interface, which we will use only for our tests:

code/dependency-injection/src/app/price-service-demo/price.service.mock.ts

```
1  import { IPriceService } from './price-service.interface';
2
3  export class MockPriceService implements IPriceService {
4    calculateTotalPrice(basePrice: number, state: string) {
5      if (state === 'FL') {
6        return basePrice + 0.66; // it's always 66 cents!
7      }
8
9      return basePrice;
10   }
11 }
```

Now, just because we've written a `MockPriceService` doesn't mean our `Product` will use it. In order to use this service, we need to modify our `Product` class:

code/dependency-injection/src/app/price-service-demo/product.model.ts

```
import { IPriceService } from './price-service.interface';

export class Product {
  service: IPriceService;
  basePrice: number;

  constructor(service: IPriceService, basePrice: number) {
    this.service = service; // <-- passed in as an argument!
    this.basePrice = basePrice;
  }

  totalPrice(state: string) {
    return this.service.calculateTotalPrice(this.basePrice, state);
  }
}
```

Now, when creating a `Product` **the client using the `Product` class becomes responsible for deciding which concrete implementation of the `PriceService` is going to be given to the new instance.**

And with this change, we can tweak our test slightly and get rid of the dependency on the unpredictable `PriceService`:

code/dependency-injection/src/app/price-service-demo/product.spec.ts

```
import { Product } from './product.model';
import { MockPriceService } from './price.service.mock';

describe('Product', () => {
  let product;

  beforeEach(() => {
    const service = new MockPriceService();
    product = new Product(service, 11.00);
  });

  describe('price', () => {
    it('is calculated based on the basePrice and the state', () => {
      expect(product.totalPrice('FL')).toBe(11.66);
    });
  });
});
```

We also get the bonus of having confidence that we're testing the `Product` class *in isolation*. That is, we're making sure that our class works with a predictable dependency.

While the predictability is nice, it's a bit laborious to pass a concrete implementation of a service every time we want a new `Product`. Thankfully, Angular's DI library helps us deal with that problem, too. More on that below.

Within Angular's DI system, instead of directly `import`ing and creating a `new` instance of a class, instead we will:

- Register the "dependency" with Angular
- Describe *how* the dependency will be *injected*
- Inject the dependency

One benefit of this model is that the dependency *implementation* can be swapped at run-time (as in our mocking example above). But another significant benefit is that we can configure **how the dependency is created**.

That is, often in the case of program-wide services, we may want to have **only one instance** - that is, a Singleton. With DI we're able to configure Singletons easily.

A third use-case for DI is for configuration or environment-specific variables. For instance, we might define a "constant" `API_URL`, but then inject a different value in production vs. development.

Let's learn how to create our own services and the different ways of injecting them.

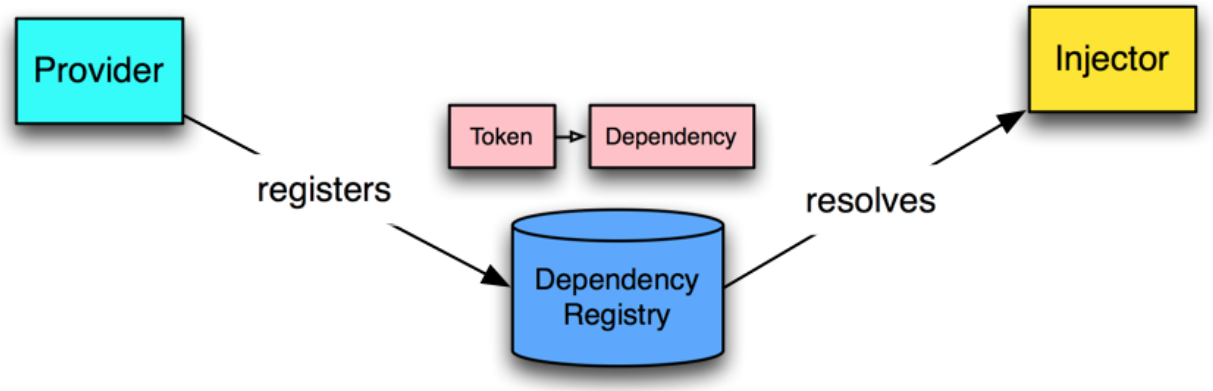
Dependency Injection Parts

To register a dependency we have to **bind it** to something that will **identify that dependency**. This identification is called the dependency **token**. For instance, if we want to register the URL of an API, we can use the string `API_URL` as the token. Similarly, if we're registering a class, we can use the class itself as its **token** as we'll see below.

Dependency injection in Angular has three pieces:

- the **Provider** (also often referred to as a binding) maps a *token* (that can be a string or a class) to a list of dependencies. It tells Angular how to create an object, given a token.
- the **Injector** that holds a set of bindings and is responsible for resolving dependencies and injecting them when creating objects
- the **Dependency** that is what's being injected

We can think of the role of each piece as illustrated below:



Dependency Injection

A way of thinking about this is that when we configure DI we specify **what** is being injected and **how** it will be resolved.

Playing with an Injector

Above with our `Product` and `PriceService` we **manually** created the `PriceService` using the `new` operator. This mimics what Angular itself does.

Angular uses an *injector* to **resolve** a dependency and **create the instance**. This is done for us behind the scenes, but as an exercise, it's useful to explore what's happening. It can be enlightening to use the injector manually, because we can see what Angular does for us behind the scenes.

Let's **manually use the injector** in our component to resolve and create a service. (After we've resolved a dependency manually, we'll show the typical, easy way of injecting dependencies.)

One of the common use-cases for services is to have a 'global' Singleton object. For instance, we might have a `UserService` which contains the information for the currently logged in user. Many different components will want to have logic based on the current user, so this is a good case for a service.

Here's a basic `UserService` that stores the user object as a property:

code/dependency-injection/src/app/services/user.service.ts

```
import { Injectable } from '@angular/core';

@Injectable()
export class UserService {
  user: any;

  setUser(newUser) {
    this.user = newUser;
  }

  getUser(): any {
    return this.user;
  }
}
```

Say we want to create a toy sign-in form:

code/dependency-injection/src/app/user-demo/user-demo.component.html

```
<div>
  <p
    *ngIf="userName"
    class="welcome">
    Welcome: {{ userName }}!
  </p>
  <button
    (click)="signIn()"
    class="ui button"
    >Sign In
  </button>
</div>
```

Above, we click the "Sign In" button to call the `signIn()` function (which we'll define in a moment). If we have a `userName`, we'll display a greeting.

Simple Sign In Button

Now let's implement this functionality in our component by **using the injector directly**.

code/dependency-injection/src/app/user-demo/user-demo.injector.component.ts

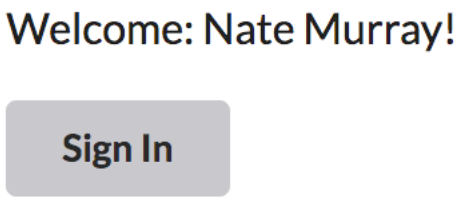
```
import {
  Component,
  ReflectiveInjector
 } from '@angular/core';

 import { UserService } from '../services/user.service';

@Component({
  selector: 'app-injector-demo',
  templateUrl: './user-demo.component.html',
  styleUrls: ['./user-demo.component.css']
})
export class UserDemoInjectorComponent {
  userName: string;
  userService: UserService;

  constructor() {
    // Create an _injector_ and ask for it to resolve and create a UserService
    const injector: any = ReflectiveInjector.resolveAndCreate([UserService]);

    // use the injector to **get the instance** of the UserService
    this.userService = injector.get(UserService);
  }

```

```
25    signIn(): void {
26      // when we sign in, set the user
27      // this mimics filling out a login form
28      this.userService.setUser({
29        name: 'Nate Murray'
30      });
31
32      // now **read** the user name from the service
33      this.userName = this.userService.getUser().name;
34      console.log('User name is: ', this.userName);
35    }
36  }
```

This starts as a basic component: we have a selector, template, and CSS. Note that we have two properties: `userName`, which holds the currently logged-in user's name and `userService`, which holds a reference to the `UserService`.

In our component's `constructor` we are using a static method from `ReflectiveInjector` called `resolveAndCreate`. That method is responsible for creating a new injector. The parameter we pass in is an array with all the *injectable things* we want this new injector to *know*. In our case, we just wanted it to know about the `UserService` injectable.

The `ReflectiveInjector` is a concrete implementation of `Injector` that uses *reflection* to look up the proper parameter types. While there are other injectors that are possible `ReflectiveInjector` is the "normal" injector we'll be using in most apps.



Signed In

Providing Dependencies with `NgModule`

While it's interesting to see how an `injector` is created directly, that isn't the typical way we'd use injections. Instead, what we'd normally do is

- use `NgModule` to *register* what we'll inject – these are called *providers* and
- use decorators (generally on a `constructor`) to specify **what we're injecting**

By doing these two steps **Angular** will manage creating the injector and resolving the dependencies.

Let's convert our `UserService` to be *injectable* as a singleton across our app. First, we're going to add it to the `providers` key of our `NgModule`:

code/dependency-injection/src/app/user-demo/user-demo.module.ts

```
1  import { NgModule } from '@angular/core';
2  import { CommonModule } from '@angular/common';
3
4  // imported here
5  import { UserService } from '../services/user.service';
6
7  @NgModule({
8    imports: [
9      CommonModule
10   ],
11   providers: [
12     UserService // <-- added right here
13   ],
14   declarations: []
15 })
16 export class UserDemoModule { }
```

Now we can inject `UserService` into our component like this:

code/dependency-injection/src/app/user-demo/user-demo.component.ts

```
import { Component, OnInit } from '@angular/core';

import { UserService } from '../services/user.service';

@Component({
  selector: 'app-user-demo',
  templateUrl: './user-demo.component.html',
  styleUrls: ['./user-demo.component.css']
})
export class UserDemoComponent {
  userName: string;
  // removed `userService` because of constructor shorthand below

  // Angular will inject the singleton instance of `UserService` here.
  // We set it as a property with `private`.
  constructor(private userService: UserService) {
    // empty because we don't have to do anything else!
  }

  // below is the same...
  signIn(): void {
    // when we sign in, set the user
    // this mimics filling out a login form
    this.userService.setUser({
      name: 'Nate Murray'
    });

    // now **read** the user name from the service
    this.userName = this.userService.getUser().name;
    console.log('User name is: ', this.userName);
  }
}
```

Notice in the `constructor` above that we have made `userService: UserService` an argument to the `UserDemoComponent`. When this component is created on our page **Angular will resolve and inject the `UserService` singleton**. What's great about this is that because Angular is managing the instance, we don't have to worry about doing it ourselves. Every class that injects the `UserService` will receive the same singleton.

Providers are the Key

It's important to know that when we put the `UserService` on the constructor of the `UserDemoComponent`, Angular knows what to inject (and how) **because we listed `UserService` in the `providers` key of our `NgModule`.

It **does not** inject arbitrary classes. You **must** configure an `NgModule` for DI to work.

We've been talking a lot about Singleton services, but we can inject things in lots of other ways. Let's take a look.

Providers

There are several ways we can configure resolving injected dependencies in Angular. For instance we can:

- Inject a (singleton) instance of a class (as we've seen)
- Inject a **value**
- **Call any function** and inject the return value of that function

Let's look into detail at how we create each one:

Using a Class

As we've discussed, injecting a singleton instance of a class is probably the most common type of injection.

When we put the class itself into the list of `providers` like this:

```
1   providers: [ UserService ]
```

This tells Angular that we want to provide a singleton *instance* of `UserService` whenever `UserService` is injected. Because this pattern is so common, the class by itself is actually shorthand notation for the following, equivalent configuration:

```
1   providers: [
2     { provide: UserService, useClass: UserService }
3   ]
```

What's interesting to note is that the object configuration with `provide` takes **two** keys. `provide` is the *token* that we use **to identify the injection** and the second `useClass` is **how and what** to inject.

Here we're mapping the `UserService` *class* to the `UserService` *token*. In this case, the name of the class and the token match. This is the common case, but know that the token and the injected thing *aren't required to have the same name.*

As we've seen above, in this case the injector will create a **singleton** behind the scenes and return the same instance every time we inject it . Of course, the first time it is injected, the singleton hasn't been instantiated yet, so when creating the `UserService` instance for the first time, the DI system will trigger the class `constructor` method.

Using a Value

Another way we can use DI is to provide a value, much like we might use a global constant. For instance, we might configure an API Endpoint URL depending on the environment.

To do this, in our `NgModule` `providers`, we use the key `useValue`:

```
  providers: [
    { provide: 'API_URL', useValue: 'http://my.api.com/v1' }
  ]
```

Above, for the `provide` token we're using a *string* of `API_URL`. If we use a string for the `provide` value, Angular can't infer which dependency we're resolving by the type. For instance we **can't** write:

```
// doesn't work - anti-example
export class AnalyticsDemoComponent {
  constructor(apiUrl: 'API_URL') { // <--- this isn't a type, just a string
                                   // if we put `string` that is ambiguous
  }
}
```

So what can we do? In this case, we'll use the `@Inject()` decorator like this:

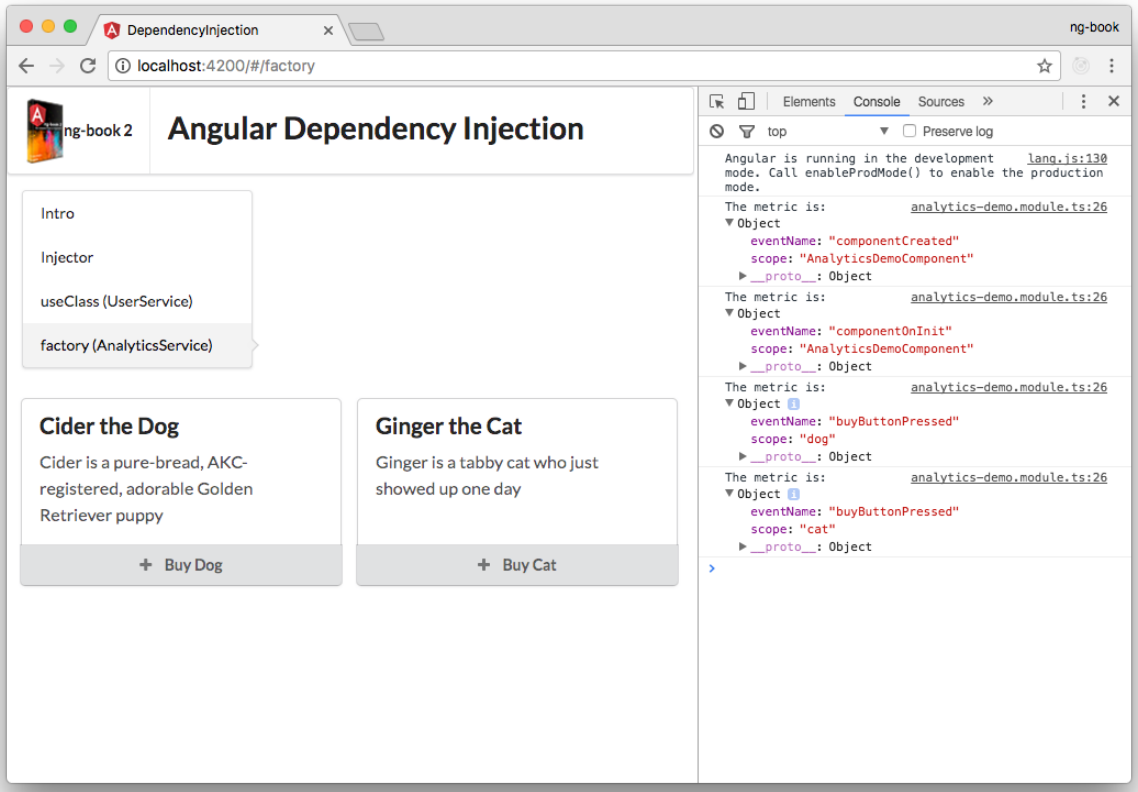
```
import { Inject } from '@angular/core';

export class AnalyticsDemoComponent {
  constructor(@Inject('API_URL') apiUrl: string) {
   // works! do something w/ apiUrl
  }
}
```

Now that we know how to do simple values with `useValue` and Singleton classes with `useClass`, we're ready to talk about the more advanced case: writing configurable services using factories.

Configurable Services

In the case of the `UserService`, no arguments are required for the `constructor`. But what happens if a service's `constructor` requires arguments? We can implement this by using a *factory* which is a function that can return **any object** when injected.

For instance, let's say we're writing a library for recording user analytics (that is, keeping a record of events of actions a user took on the page). In this scenario, we want to have an `AnalyticsService` with a catch: the `AnalyticsService` should define the interface for *recording* events, **but not the implementation for handling the event**.



Tracking Analytics on the events

Our user may, for instance, want to record these metrics with Google Analytics or they may want to use Optimizely, or some other in-house solution. Let's write an injectable `AnalyticsService` which can take an implementation configuration.

First, a couple of definitions. Let's define a `Metric`:

code/dependency-injection/src/app/analytics-demo/analytics-demo.interface.ts

```
4  export interface Metric {
5    eventName: string;
6    scope: string;
7  }
```

A `Metric` will store an `eventName` and a `scope`. We could use this for say, when a the user `nate` logs-in the `eventName` could be `loggedIn` and the `scope` would be `nate`.

```
// just an example
let metric: Metric = {
  eventName: 'loggedIn',
  scope: 'nate'
}
```

This way we could, in theory, count the number of user logins by counting the events with `eventName` `loggedIn` and count the number of times the specific user `nate` logged in by counting the `loggedIn` events with user `nate`.

We also need to define what an analytics implementation would look like:

code/dependency-injection/src/app/analytics-demo/analytics-demo.interface.ts

```
export interface AnalyticsImplementation {
  recordEvent(metric: Metric): void;
}
```

Here we define an `AnalyticsImplementation` interface to have one function: `recordEvent` which takes a `Metric` as an argument.

Now let's define the `AnalyticsService`:

code/dependency-injection/src/app/services/analytics.service.ts

```
import { Injectable } from '@angular/core';
import {
  Metric,
  AnalyticsImplementation
} from '../analytics-demo/analytics-demo.interface';

@Injectable()
export class AnalyticsService {
  constructor(private implementation: AnalyticsImplementation) {
  }

  record(metric: Metric): void {
    this.implementation.recordEvent(metric);
  }
}
```

Above our `AnalyticsService` defines one method: `record` which accepts a `Metric` and then passes it on to the `implementation`.

Of course, this `AnalyticsService` is a bit trivial and in this case, we probably wouldn't need the indirection. But this same pattern could be used in the case where you had a more advanced `AnalyticsService`. For instance, we could add middleware or broadcast to several implementations.

Notice how its constructor method takes a phrase as a parameter? If we try to use the "regular" `useClass` injection mechanism we would see an error on the browser like:

```
Cannot resolve all parameters for AnalyticsService.
```

This happens because we didn't provide the injector with the implementation necessary for the `constructor`. In order to resolve this problem, we need to configure the provider to **use a factory**.

Using a Factory

So to use our `AnalyticsService`, we need to:

- create an implementation that conforms to `AnalyticsImplementation` and
- add it to `providers` with `useFactory`

Here's how:

code/dependency-injection/src/app/analytics-demo/analytics-demo.module.1.ts

```
import { NgModule } from '@angular/core';
import { CommonModule } from '@angular/common';
import {
  Metric,
  AnalyticsImplementation
} from './analytics-demo.interface';
import { AnalyticsService } from '../services/analytics.service';

@NgModule({
  imports: [
    CommonModule
  ],
  providers: [
    {
      // `AnalyticsService` is the _token_ we use to inject
      // note, the token is the class, but it's just used as an identifier!
      provide: AnalyticsService,

```

```
      // useFactory is a function - whatever is returned from this function
      // will be injected
      useFactory() {

        // create an implementation that will log the event
        const loggingImplementation: AnalyticsImplementation = {
          recordEvent: (metric: Metric): void => {
            console.log('The metric is:', metric);
          }
        };

        // create our new `AnalyticsService` with the implementation
        return new AnalyticsService(loggingImplementation);
      }
    }
  ],
  declarations: [ ]
})
export class AnalyticsDemoModule { }
```

Here in `providers` we're using the syntax:

```
providers: [
  { provide: AnalyticsService, useFactory: () => ... }
]
```

`useFactory` takes a function and **whatever this function returns** will be injected.

Also note that we `provide AnalyticsService`. Again, when we use `provide` this way, we're using the class `AnalyticsService` as the *identifying token* of what we're going to inject. (If you wanted to be confusing, you could use a completely separate class, or less-confusingly a string.)

In `useFactory` we're creating an `AnalyticsImplementation` object that has one function: `recordEvent`. `recordEvent` is where we could, in theory, configure **what** happens when an event is recorded. Again, in a real app this would probably send an event to Google Analytics or a custom event logging software.

Lastly, we instantiate our `AnalyticsService` and return it.

Factory Dependencies

Using a factory is the most powerful way to create injectables, because we can do whatever we want within the factory function. Sometimes our factory function will have dependencies of it's own.

Say that we wanted to configure our `AnalyticsImplementation` to make an HTTP request to a particular URL. In order to do this we'd need:

- The Angular `Http` client and
- Our `API_URL` value

Here's how we could set that up:

code/dependency-injection/src/app/analytics-demo/analytics-demo.module.ts

```
import { NgModule } from '@angular/core';
import { CommonModule } from '@angular/common';
import {
  Metric,
  AnalyticsImplementation
} from './analytics-demo.interface';
import { AnalyticsService } from '../services/analytics.service';

// added this ->
import {
  HttpModule,
  Http
} from '@angular/http';

@NgModule({
  imports: [
    CommonModule,
    HttpModule, // <-- added
  ],
  providers: [
    // add our API_URL provider
    { provide: 'API_URL', useValue: 'http://devserver.com' },
    {
      provide: AnalyticsService,

      // add our `deps` to specify the factory depencies
      deps: [ Http, 'API_URL' ],

      // notice we've added arguments here
      // the order matches the deps order
      useFactory(http: Http, apiUrl: string) {

        // create an implementation that will log the event
        const loggingImplementation: AnalyticsImplementation = {
          recordEvent: (metric: Metric): void => {
            console.log('The metric is:', metric);
```

```
            console.log('Sending to: ', apiUrl);
            // ... You'd send the metric using http here ...
          }
        };

        // create our new `AnalyticsService` with the implementation
        return new AnalyticsService(loggingImplementation);
      }
    },
  ],
  declarations: [ ]
})
export class AnalyticsDemoModule { }
```

Here we're importing the `HttpModule`, both in the ES6 `import` (which makes the class *constants* available) and in our `NgModule` `imports` (which makes it available for dependency injection).

We've added an `API_URL` provider, as we did above. And then in our `AnalyticsService` provider, we've added a new key: `deps`. `deps` is an array of injection tokens and these tokens will be resolved and **passed as arguments to the factory function**.

Dependency Injection in Apps

To review, when writing our apps there are three steps we need to take in order to perform an injection:

1. Create the dependency (e.g. the service class)
2. Configure the injection (i.e. register the injection with Angular in our `NgModule`)
3. Declare the dependencies on the **receiving component**

The first thing we do is create the service class, that is, the class that exposes some behavior we want to use. This will be called the *injectable* because it is the *thing* that our components will receive via the *injection*.

Reminder on terminology: a *provider* provides (creates, instantiates, etc.) the *injectable* (the thing you want). In Angular when you want to access an *injectable* you *inject* a dependency into a function (often a `constructor`) and Angular's dependency injection framework will locate it and provide it to you.

As we can see, Dependency Injection provides a powerful way to manage dependencies within our app.

More Resources

- Official Angular DI Docs[49]
- Victor Savkin Compare DI in Angular 1 vs. Angular 2[50]

[49] https://angular.io/docs/ts/latest/guide/dependency-injection.html
[50] http://victorsavkin.com/post/126514197956/dependency-injection-in-angular-1-and-angular-2