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 lets 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.

INJECTIONS EXAMPLE: PRICESERVICE

Let's imagine we're building a store that has `Product`s 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:

```
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

```
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).

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.

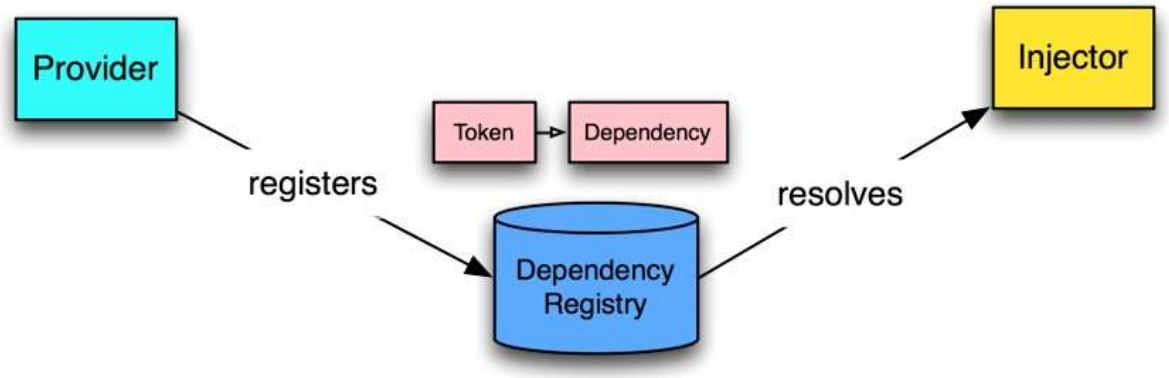
A third use-case for DI is for configuration or environment-specific variables. For instance, we might define a "constant" `API_URL`, but then inject a different value in production vs. development.

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



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:

```
@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:

```
<div>
<p
*ngIf="userName" 4   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.

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

```
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]); 20
// use the injector to **get the instance** of the UserService
this.userService = injector.get(UserService);
}

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);
}
}
```

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.

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`:

```
@NgModule({
      imports: [
            CommonModule
      ],
      providers: [
            UserService // <-- added right here
      ],
      declarations: []
})
export class UserDemoModule { }
```

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

```
@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 13
// 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:

1. Inject a (singleton) instance of a class (as we've seen)
2. Inject a value
3. Call any function and inject the return value of that function