DEPENDENCY INJECTION

Developer Preview Only - some details may change

Angular's dependency injection system creates and delivers dependent services "just-in-time".

Dependency Injection is an important application design pattern. 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".

In this chapter we'll learn what DI is, why we want it. Then we'll learn how to use it in an Angular app.

All code in this chapter is available as a live example on the web.

Why Dependency Injection?

Let's start with the following code.

app/car/car.ts (no di)

```
export class Car {

  public engine: Engine;
  public tires: Tires;
  public description = 'No DI';

  constructor() {
    this.engine = new Engine();
    this.tires = new Tires();
  }

  // Method using the engine and tires
  drive() {
    return `${this.description} car with ` +
      `${this.engine.cylinders} cylinders and ${this.tires.make} tires.`
  }
}
```

Our `Car` creates everything it needs inside its constructor. What's the problem?

The problem is that our `Car` class is brittle, inflexible, and hard to test.

Our `Car` needs an engine and tires. Instead of asking for them, the `Car` constructor creates its own copies by "new-ing" them from the very specific classes, `Engine` and `Tires`.

What if the `Engine` class evolves and its constructor requires a parameter? Our `Car` is broken and stays broken until we rewrite it along the lines of `this.engine = new Engine(theNewParameter)`. We didn't care about `Engine` constructor parameters when we first wrote `Car`. We don't really care about them now. But we'll *have* to start caring because when the definion of `Engine` changes, our `Car` class must change. That makes `Car` brittle.

What if we want to put a different brand of tires on our `Car`. Too bad. We're locked into whatever brand the `Tires` class creates. That makes our `Car` inflexible.

Right now each new car gets its own engine. It can't share an engine with other cars. While that makes sense for an automobile engine, we can think of other dependencies that should be shared ... like the onboard wireless connection to the manufacturer's service center. Our `Car` lacks the flexibility to share services that have been created previously for other consumers.

When we write tests for our `Car` we're at the mercy of its hidden dependencies. Is it even possible to create a new `Engine` in a test environment? What does `Engine`itself depend upon? What does that dependency depend on? Will a new instance of `Engine` make an asynchronous call to the server? We certainly don't want that going on during our tests.

What if our `Car` should flash a warning signal when tire pressure is low. How do we confirm that it actually does flash a warning if we can't swap in low-pressure tires during the test?

We have no control over the car's hidden dependencies. When we can't control the dependencies, a class become difficult to test.

How can we make `Car` more robust, more flexible, and more testable?

That's super easy. We probably already know what to do. We change our `Car` constructor to this:

```
public description = 'DI';
```

```
constructor(public engine: Engine, public tires: Tires) { }
```

See what happened? We moved the definition of the dependencies to the constructor. Our `Car` class no longer creates an engine or tires. It just consumes them.

> We also leverage TypeScript's constructor syntax for declaring parameters and properties simultaneously.

Now we create a car by passing the engine and tires to the constructor.

Simple Car

```
// Simple car with 4 cylinders and Flintstone tires.
var car = new Car(new Engine(), new Tires());
```

How cool is that? The definition of the engine and tire dependencies are decoupled from the `Car` class itself. We can pass in any kind of engine or tires we like, as long as they conform to the general API requirements of an engine or tires.

If someone extends the `Engine` class, that is not `Car`'s problem.

> The consumer of `Car` has the problem. The consumer must update the car creation code to something like:
>
> Super Car

```
class Engine2 {
  constructor(public cylinders: number) { }
}
// Super car with 12 cylinders and Flintstone tires.
var bigCylinders = 12;
var car = new Car(new Engine2(bigCylinders), new Tires());
```

The critical point is this: `Car` itself did not have to change. We'll take care of the consumer's problem soon enough.

The `Car` class is much easier to test because we are in complete control of its dependencies. We can pass mocks to the constructor that do exactly what we want them to do during each test:

Test Car

```
class MockEngine extends Engine { cylinders = 8; }
class MockTires  extends Tires  { make = "YokoGoodStone"; }

// Test car with 8 cylinders and YokoGoodStone tires.
var car = new Car(new MockEngine(), new MockTires());
```

We just learned what Dependency Injection is.

It's a coding pattern in which a class receives its dependencies from external sources rather than creating them itself.

Cool! But what about that poor consumer? Anyone who wants a `Car` must now create all three parts: the `Car`, `Engine`, and

`Tires`. The `Car` class shed its problems at the consumer's expense. We need something that takes care of assembling these parts for us.

We could write a giant class to do that:

app/car/car-factory.ts

```
import {Engine, Tires, Car} from './car';

export class CarFactory {

  createCar() {
    let car = new Car(this.createEngine(), this.createTires());
    car.description = 'Factory';
    return car;
  }

  createEngine() {
    return new Engine();
  }

  createTires() {
    return new Tires();
  }
}
```

It's not so bad now with only three creation methods. But maintaining it will be hairy as the application grows. This `SuperFactory` is going to become a huge spider web of interdependent factory methods!

Wouldn't it be nice if we could simply list the things we want to build without having to define which dependency gets injected into what?

This is where the Dependency Injection Framework comes into play. Imagine the framework had something called an `Injector`. We register some classes with this `Injector` and it figures out how to create them.

When we need a `Car`, we simply ask the `Injector` to get it for us and we're good to go.

```
var car = injector.get(Car);
```

Everyone wins. The `Car` knows nothing about creating an `Engine` or `Tires`. The consumer knows nothing about creating a `Car`. We don't have a gigantic factory class to maintain. Both `Car` and consumer simply ask for what they need and the `Injector` delivers.

This is what a **Dependency Injection Framework** is all about.

Now that we know what Dependency Injection is and appreciate its benefits, let's see how it is implemented in Angular.

Angular Dependency Injection

Angular ships with its own Dependency Injection framework. This framework can also be used as a standalone module by other applications and frameworks.

That sounds nice. What does it do for us when building components in Angular? Let's see, one step at a time.

We'll begin with a simplified version of the `HeroesComponent` that we built in the The Tour of Heroes.

```
import { Component }          from 'angular2/core';
import { HeroListComponent }  from './hero-list.component';

@Component({
  selector: 'my-heroes',
  template: `
  <h2>Heroes</h2>
  <hero-list></hero-list>
  `,
  directives:[HeroListComponent]
})
export class HeroesComponent { }
```

The `HeroesComponent` is the root component of the *Heroes* feature area. It governs all the child components of this area. In our stripped down version there is only one child, `HeroListComponent`, dedicated to displaying a list of heroes.

> Do we really need so many files? Of course not! We're going *beyond* the strictly necessary in order to illustrate patterns that will serve us well in real applications. With each file we can make one or two points rather than crowd them all into one file.

Right now it gets heroes from `HEREOES`, an in-memory collection, defined in another file and imported by this component. That may suffice in the early stages of development but it's far from ideal. As soon as we try to test this component or want to get our heroes data from a remote server, we'll have to change the implementation of `heroes` and fix every other use of the `HEROES` mock data.

Let's make a service that hides how we get hero data.

> Write this service in its own file. See this note to understand why.

app/heroes/hero.service.ts

```
import {Hero}   from './hero';
import {HEROES} from './mock-heroes';

export class HeroService {
  getHeroes() { return HEROES;  }
}
```

Our `HeroService` exposes a `getHeroes()` method that returns the same mock data as before but none of its consumers need to know that.

> We aren't even pretending this is a real service. If we were actually getting data from a remote server, the API would have to be asynchronous, perhaps returning ES2015 promises We'd also have to rewrite the way components consume our service. This is important but not important to our current story.

A service is nothing more than a class in Angular 2. It remains nothing more than a class until we register it with an Angular injector.

Configuring the Injector

We don't have to create an Angular injector. Angular creates an application-wide injector for us during the bootstrap process.

app/main.ts (bootstrap)

```
bootstrap(AppComponent);
```

We do have to configure the injector by registering the **providers** that create the services we need in our application. We'll explain what providers are later in this chapter. Before we do, let's see an example of provider registration during bootstrapping:

```
// Injecting services in bootstrap works but is discouraged
bootstrap(AppComponent, [HeroService]);
```

The injector now knows about our `HeroService`. An instance of our `HeroService` will be available for injection across our entire application.

Of course we can't help wondering about that comment telling us not to do it this way. It *will* work. It's just not a best practice. The bootstrap provider option is intended for configuring and overriding Angular's own pre-registered services.

The preferred approach is to register application providers in application components. Because the `HeroService` will be used within the *Heroes* feature area — and nowhere else — the ideal place to register it is in the top-level `HeroesComponent`.

Registering providers in a component

Here's a revised `HeroesComponent` that registers the `HeroService`.

app/heroes/heroes.component.ts

```
import { Component }           from 'angular2/core';
import { HeroListComponent }   from './hero-list.component';
import { HeroService }         from './hero.service';

@Component({
  selector: 'my-heroes',
  template: `
  <h2>Heroes</h2>
  <hero-list></hero-list>
  `,
  providers:[HeroService],
  directives:[HeroListComponent]
})
export class HeroesComponent { }
```

Look closely at the bottom of the `@Component` metadata:

```
providers:[HeroService],
```

An instance of the `HeroService` is now available for injection in this `HeroesComponent` and all of its child components.

The `HeroesComponent` itself doesn't happen to need the `HeroService`. But its child `HeroListComponent` does so we head there next.

Preparing the *HeroListComponent* for injection

The `HeroListComponent` should get heroes from the injected `HeroService`. Per the dependency injection pattern, the component must "ask for" the service in its constructor as we explained earlier. It's a small change as we see in this comparison:

```
import { Component }   from 'angular2/core';
import { Hero }        from './hero';
import { HeroService } from './hero.service';

@Component({
  selector: 'hero-list',
  template: `
  <div *ngFor="#hero of heroes">
    {{hero.id}} - {{hero.name}}
  </div>
  `,
})
export class HeroListComponent {
  heroes: Hero[];

  constructor(heroService: HeroService) {
    this.heroes = heroService.getHeroes();
  }
}
```

Focus on the constructor

```
    constructor(heroService: HeroService) {
      this.heroes = heroService.getHeroes();
    }
```

Adding a parameter to the constructor isn't all that's happening here.

We're writing in TypeScript and have followed the parameter name with a type annotation, `:HeroService`. The class is also decorated with the `@Component` decorator (scroll up to confirm that fact).

When the TypeScript compiler evaluates this class, it sees the `@Component` decorator and adds class metadata into the generated JavaScript code. Within that metadata lurks the information that associates the `heroService` parameter with the `HeroService` class.

That's how the Angular injector knows to inject an instance of the `HeroService` when it creates a new `HeroListComponent`.

Creating the injector (implicitly)

When we introduced the idea of an injector above, we showed how to create an injector and use it to create a new `Car`.

```
injector = Injector.resolveAndCreate([Car, Engine, Tires, Logger]);
var car = injector.get(Car);
```

We won't find code like that in the Tour of Heroes or any of our other samples. We *could* write code like that if we *had* to which we rarely do. Angular takes care of creating and calling injectors when it creates components for us whether through HTML markup, as in `<hero-list></hero-list>`, or after navigating to a component with the router. Let Angular do its job and we'll enjoy the benefits of automated dependency injection.

Singleton services

Dependencies are singletons within the scope of an injector. In our example, there is a single `HeroService` instance shared among the `HeroesComponent` and its `HeroListComponent` children.

However, Angular DI is an hierarchical injection system which means nested injectors can create their own service instances. Learn more about that in the Hierarchical Injection chapter.

Testing the component

We emphasized earlier that designing a class for dependency injection makes the class easier to test. Listing dependencies as constructor parameters may be all we need to test application parts effectively.

For example, we can create a new `HeroListComponent` with a mock service that we can manipulate under test:

```
let expectedHeroes = [{name: 'A'}, {name: 'B'}]
let mockService = <HeroService> {getHeroes: () => expectedHeroes }

it("should have heroes when HeroListComponent created", () => {
  let hlc = new HeroListComponent(mockService);
  expect(hlc.heroes.length).toEqual(expectedHeroes.length);
})
```

> Learn more in the Testing chapter.

When the service needs a service

Our `HeroService` is very simple. It doesn't have any dependencies of its own.

What if it had a dependency? What if it reported its activities through a logging service? We'd apply the same *constructor injection* pattern, adding a constructor that takes a `logger` parameter.

Here is the revision compared to the original.

```
import {Injectable} from 'angular2/core';
import {Hero}       from './hero';
import {HEROES}     from './mock-heroes';
import {Logger}     from '../logger.service';

@Injectable()
export class HeroService {

  constructor(private _logger: Logger) {  }

  getHeroes() {
    this._logger.log('Getting heroes ...')
    return HEROES;
  }
}
```

The constructor now asks for an injected instance of a `Logger` and stores it in a private property called `_logger`. We call that property within our `getHeroes()` method when anyone asks for heroes.

Why *@Injectable*?

Notice the `@Injectable()` decoration above the service class. We haven't seen `@Injectable()` before. As it happens, we could have added it to our first version `HeroService`. We didn't bother because we didn't need it then.

We need it now... now that our service has an injected dependency. We need it because Angular requires constructor parameter metadata in order to inject a `Logger`. As we mentioned earlier, TypeScript *only generates metadata for classes that have a decorator.* .

ALWAYS ADD @INJECTABLE()

We recommend adding `@Injectable()` to every service class, even those that don't have dependencies and, therefore, do not technically require it. Two reasons:

1. *Future proofing*: no need to remember `@Injectable` when we add a dependency later.
2. *Consistency*: all services follow the same rules and we don't have to wonder why a decorator is missing.

The `HeroesComponent` has an injected dependency too. Why don't we add `@Injectable()` to the `HeroesComponent`?

We *can* add it if we really want to. It isn't necessary because the `HeroesComponent` is already decorated with `@Component`. TypeScript generates metadata for *any* class with a decorator and *any* decorator will do.

Always include the parentheses! Always call `@Injectable()`. Our application will fail mysteriously if we forget the parentheses.

Create and register the *Logger* service

We're injecting a Logger into our `HeroService`. We have two remaining steps:

1. Create the Logger service
2. Register it with the application

The logger service implementation is no big deal.

app/logger.service

```
import {Injectable} from 'angular2/core';

@Injectable()
export class Logger {
  logs:string[] = []; // capture logs for testing
  log(message: string){
    this.logs.push(message);
    console.log(message);
  }
}
```

We're likely to need the same logger service everywhere in our application so we put it at the root level of the application in the `app/` folder and we register it in the `providers` array of the metadata for our application root component, `AppComponent`.

app/app.component.ts (providers)

```
providers: [Logger]
```

If we forget to register it, Angular throws an exception when it first looks for the logger:

```
EXCEPTION: No provider for Logger! (HeroListComponent -> HeroService -> Logger)
```

That's Angular telling us that the dependency injector couldn't find the *provider* for the logger. It needed that provider to create a `Logger` to inject into a new `HeroService` which it needed to create and inject into a new `HeroListComponent`.

The chain of creations started with the `Logger` provider. The *provider* is the subject of our next section.

But wait! What if the logger is optional?

Optional dependencies

Our `HeroService` currently requires a `Logger`. What if we could get by without a logger? We'd use it if we had it, ignore it if we didn't. We can do that.

First import the `@Optional()` decorator.

```
import {Optional} from 'angular2/core';
```

Then rewrite the constructor with `@Optional()` decorator preceding the private `_logger` parameter. That tells the injector that `_logger` is optional.

```
log:string;
constructor(@Optional() private _logger:Logger) {  }
```

Be prepared for a null logger. If we don't register one somewhere up the line, the injector will inject `null`. We have a method that logs. What can we do to avoid a null reference exception?

We could substitute a *do-nothing* logger stub so that calling methods continue to work:

```
// No logger? Make one!
if (!this._logger) {
  this._logger = {
    log: (msg:string)=> this._logger.logs.push(msg),
    logs: []
  }
}
```

Obviously we'd take a more sophisticated approach if the logger were optional in multiple locations.

But enough about optional loggers. In our sample application, the `Logger` is required. We must register a `Logger` with the application injector using *providers* as we learn in the next section.

Injector Providers

A provider *provides* the concrete, runtime version of a dependency value. The injector relies on **providers** to create instances of the services that it injects into components and other services.

We must register a service *provider* with the injector or it won't know how to create the service.

Earlier we registered the `Logger` service in the `providers` array of the metadata for the `AppComponent` like this:

```
providers: [Logger]
```

The `providers` array appears to hold a service class. In reality it holds an instance the Provider class that can create that service.

There are many ways to *provide* something that looks and behaves like a `Logger`. The `Logger` class itself is an obvious and natural provider - it has the right shape and it's designed to be created. But it's not the only way.

We can configure the injector with alternative providers that can deliver an object that behaves like a `Logger`. We could provide a substitute class. We could provide a logger-like object. We could give it a provider that calls a logger factory function. Any of these approaches might be a good choice under the right circumstances.

What matters is that the injector has a provider to go to when it needs a `Logger`.

The *provide* function

We wrote the `providers` array like this:

```
[Logger]
```

This is actually a short-hand expression for a provider registration that creates a new instance of the Provider class.

```
[new Provider(Logger, {useClass: Logger})]
```

The provide function is the more common and friendlier way to create a `Provider`:

```
[provide(Logger, {useClass: Logger})]
```

In both approaches — `Provider` class and `provide` function — we supply two arguments.

The first is the token that serves as the key for both locating a dependency value and registering the provider.

The second is a provider definition object which we think of as a *recipe* for creating the dependency value. There are many ways to create dependency values ... and many ways to write a recipe.

Alternative Class Providers

Occasionally we'll ask a different class to provide the service. In this example, we tell the injector to return a `BetterLogger` when something asks for the `Logger`.

```
[provide(Logger, {useClass: BetterLogger})]
```

Class provider with dependencies

Maybe an `EvenBetterLogger` could display the user name in the log message. This logger gets the user from the injected `UserService` which happens also to be injected at the application level.

```
@Injectable()
class EvenBetterLogger {
  logs:string[] = [];

  constructor(private _userService: UserService) { }

  log(message:string){
    message = `Message to ${this._userService.user.name}: ${message}.`;
    console.log(message);
    this.logs.push(message);
  }
}
```

Configure it like we did `BetterLogger`.

```
[ UserService,
  provide(Logger, {useClass: EvenBetterLogger}) ]
```

Aliased Class Providers

Suppose there is an old component that depends upon an `OldLogger` class. `OldLogger` has the same interface as the `NewLogger` but for some reason we can't update the old component to use it.

When the *old* component logs a message with `OldLogger`, we want the one singleton instance of `NewLogger` to handle it instead.

The dependency injector should inject that singleton instance when a component asks for either the new or the the old logger. The `OldLogger` should be an alias for `NewLogger`.

We certainly do not want two different `NewLogger` instances in our app. Unfortunately, that's what we get if we try to alias `OldLogger` to `NewLogger` with `useClass`.

```
[ NewLogger,
  // Not aliased! Creates two instances of `NewLogger`
  provide(OldLogger, {useClass:NewLogger}) ]
```

Alias with the `useExisting` option instead.

```
[ NewLogger,
  // Alias OldLogger w/ reference to NewLogger
  provide(OldLogger, {useExisting: NewLogger}) ]
```

Value Providers

Sometimes it's easier to provide a ready-made object rather than ask the injector to create it from a class.

```
// An object in the shape of the logger service
let silentLogger = {
  logs: ['Silent logger says "Shhhhh!". Provided via "useValue"'],
  log: () => {}
}
```

Then we register a provider with this object playing the logger role via the `useValue` option.

```
[provide(Logger, {useValue: silentLogger})]
```

Factory Providers

Sometimes we need to create the dependent value dynamically, based on information we won't have until the last possible moment. Maybe the information keeps changes repeatedly in the course of the browser session..

Suppose also that the injectable service has no independent access to the source of this information.

This situation calls for a **factory provider**.

Let's illustrate by adding a new business requirement: the HeroService must hide *secret* heroes from normal users. Only authorized users should see secret heroes.

Like the `EvenBetterLogger`, the `HeroService` needs a fact about the user. It needs to know if the user is authorized to see secret heroes. That authorization can change during the course of a single application session as when we log in a different user.

Unlike `EvenBetterLogger`, we can't inject the `UserService` into the `HeroService`. The `HeroService` won't have direct access to the user information to decide who is authoriazed and who is not.

> Why? We don't know either. Stuff like this happens.

Instead the `HeroService` constructor takes a boolean flag to control display of secret heroes.

app/heroes/hero.service.ts (internals)

```
constructor(
  private _logger: Logger,
  private _isAuthorized: boolean) { }

getHeroes() {
  let auth = this._isAuthorized ? 'authorized ': 'unauthorized';
  this._logger.log(`Getting heroes for ${auth} user.`);
  return HEROES.filter(hero => this._isAuthorized || !hero.isSecret);
}
```

We can inject the `Logger` but we can't inject the boolean `isAuthorized`. We'll have to take over the creation of new instances of this `HeroService` with a factory provider.

A factory provider needs a factory function:

app/heroes/hero.service.provider.ts (factory)

```
let heroServiceFactory = (logger: Logger, userService: UserService) => {
  return new HeroService(logger, userService.user.isAuthorized);
}
```

Although the `HeroService` has no access to the `UserService`, our factory function does.

We inject both the `Logger` and the `UserService` into the factory provider and let the injector pass them along to the factory function:

app/heroes/hero.service.provider.ts (provider)

```
export let heroServiceProvider =
  provide(HeroService, {
    useFactory: heroServiceFactory,
    deps: [Logger, UserService]
  });
```

The `useFactory` field tells Angular that the provider is a factory function whose implementation is the `heroServiceFactory`.

The `deps` property is an array of provider tokens. The `Logger` and `UserService` classes serve as tokens for their own class

> providers. The injector resolves these tokens and injects the corresponding services into the matching factory function parameters.

Notice that we captured the factory provider in an exported variable, `heroServiceProvider`. This extra step makes the factory provider re-usable. We can register our `HeroService` with this variable whereever we need it.

In our sample, we need it only in the `HeroesComponent` where it replaces the previous `HeroService` registration in the metadata `providers` array. Here we see the new and the old implementation side-by-side:

```
import { Component }          from 'angular2/core';
import { HeroListComponent }  from './hero-list.component';
import { heroServiceProvider} from './hero.service.provider';

@Component({
  selector: 'my-heroes',
  template: `
  <h2>Heroes</h2>
  <hero-list></hero-list>
  `,
  providers:[heroServiceProvider],
  directives:[HeroListComponent]
})
export class HeroesComponent { }
```

Dependency Injection Tokens

When we register a provider with an injector we associate that provider with a dependency injection token. The injector maintains an internal *token/provider* map that it references when asked for a dependency. The token is the key to the map.

In all previous examples, the dependency value has been a class *instance* and the class *type* served as its own lookup key. Here we get a `HeroService` directly from the injector by supplying the `HeroService` type as the key/token.

```
heroService:HeroService = this._injector.get(HeroService);
```

We have similar good fortune (in typescript) when we write a constructor that requires an injected class-based dependency. We define a constructor parameter with the `HeroService` class type and Angular knows to inject the service associated with that `HeroService` class token:

```
constructor(heroService: HeroService){ }
```

This is especially convenient when we consider that most dependency values are provided by classes.

Non-class Dependencies

What if the dependency value isn't a class? Sometimes the thing we want to inject is a string, a function, or an object.

Applications often define configuration objects with lots of small facts like the title of the application or the address of a web api endpoint. These configuration objects aren't always instances of a class. They tend to be object hashes like this one:

app/app-config.ts

```
export interface Config {
  apiEndpoint: string,
  title: string
}

export const CONFIG:Config = {
  apiEndpoint: 'api.heroes.com',
  title: 'Dependency Injection'
};
```

We'd like to make this `config` object available for injection. We know we can register an object with a Value Provider. But what do we use for the token? We don't have a class to serve as a token. There is no `Config` class.

Interfaces aren't valid tokens

The `CONFIG` constant has an interface, `Config`. Unfortunately, we cannot use an interface as a token:

```
// FAIL!  Can't use interface as provider token
[provide(Config, {useValue: CONFIG})]
```

```
// FAIL! Can't inject using the interface as the parameter type
constructor(private _config: Config){ }
```

That seems strange if we're used to dependency injection in strongly-typed languages where an interface is the preferred dependency lookup key.

It's not Angular's fault. An interface is a TypeScript design-time artifact. JavaScript doesn't have interfaces. The TypeScript interface disappears from the generated JavaScript. There is no interface type information left for Angular to find at runtime.

String tokens

Fortunately, we can register any dependency provider with a string token.

```
// Use string as provider token
[provide('app.config', {useValue: CONFIG})]
```

Now we inject the configuration object into any constructor that needs it with the help of an `@Inject` decorator to tell Angular how to find the configuration dependency value.

```
// @Inject(token) to inject the dependency
constructor(@Inject('app.config') private _config: Config){ }
```

Although it plays no role in dependency injection, the `Config` interface supports strongly-typing of the configuration object within the class.

OpaqueToken

Alternatively, we could define and use an OpaqueToken rather than rely on magic strings that may collide with other developers' string choices.

The definition looks like this:

```
import {OpaqueToken} from 'angular2/core';

export let APP_CONFIG = new OpaqueToken('app.config');
```

Substitute `APP_CONFIG` for the magic string in provider registration and constructor:

```
providers:[provide(APP_CONFIG, {useValue: CONFIG})]
```

```
constructor(@Inject(APP_CONFIG) private _config: Config){ }
```

Here's how we provide and inject the configuration object in our top-level `AppComponent`.

app/app.component.ts (providers)

```
providers: [
  Logger,
  UserService,
  provide(APP_CONFIG, {useValue: CONFIG})
]
```

app/app.component.ts (constructor)

```
constructor(
  @Inject(APP_CONFIG) config:Config,
  private _userService: UserService) {

  this.title = config.title;
}
```

Angular itself uses `OpaqueTokens` to register all of its own non-class dependencies. For example, HTTP_PROVIDERS is the `OpaqueToken` associated with an array of providers for persisting data with the Angular `Http` client.

Internally, the `Provider` turns both the string and the class type into an `OpaqueToken` and keys its *token/provider* map with that.

Next Steps

We learned the basics of Angular Dependency Injection in this chapter.

The Angular Dependency Injection is more capable than we've described. We can learn more about its advanced features, beginning with its support for nested injectors, in the Hierarchical Dependency Injection chapter.

Appendix: Working with injectors directly

We rarely work directly with an injector. Here's an `InjectorComponent` that does.

app/injector.component.ts

```
@Component({
  selector: 'my-injectors',
  template: `
  <h2>Other Injections</h2>
  <div id="car"> {{car.drive()}}</div>
  <div id="hero">{{hero.name}}</div>
  <div id="rodent">{{rodent}}</div>
  `,
  providers: [Car, Engine, Tires,
              heroServiceProvider, Logger]
})
export class InjectorComponent {
  constructor(private _injector: Injector) { }

  car:Car = this._injector.get(Car);

  heroService:HeroService = this._injector.get(HeroService);
  hero = this.heroService.getHeroes()[0];

  get rodent() {
    let rous = this._injector.getOptional(ROUS);
    if (rous) {
      throw new Error('Aaaargh!')
    }
    return "R.O.U.S.'s? I don't think they exist!";
  }
}
```

The `Injector` is itself an injectable service.

In this example, Angular injects the component's own `Injector` into the component's constructor. The component then asks the injected injector for the services it wants.

Note that the services themselves are not injected into the component. They are retrieved by calling `injector.get`.

The `get` method throws an error if it can't resolve the requested service. We can call `getOptional` instead, which we do in one case to retrieve a service (`ROUS`) that isn't registered with this or any ancestor injector.

> The technique we just described is an example of the Service Locator Pattern.
>
> We **avoid** this technique unless we genuinely need it. It encourages a careless grab-bag approach such as we see here. It's difficult to explain, understand, and test. We can't know by inspecting the constructor what this class requires or what it will do. It could acquire services from any ancestor component, not just its own. We're forced to spelunk the implementation to discover what it does.
>
> Framework developers may take this approach when they must acquire services generically and dynamically.

Appendix: Why we recommend one class per file

Having multiple classes in the same file is confusing and best avoided. Developers expect one class per file. Keep them happy.

If we scorn this advice and, say, combine our `HeroService` class with the `HeroesComponent` in the same file, **define the component last!** If we define the component before the service, we'll get a runtime null reference error.

> We actually can define the component first with the help of the `forwardRef()` method as explained in this blog post. But why flirt with trouble? Avoid the problem altogether and define components and services in separate files.

> **Next Step**
>
> Template Syntax