Overview

Services

Dependency Injection

Component Lifecycle Hooks

Services

Services

A Service provides anything our application needs. It often shares data or functions between other Angular features

Services

Angular 1

- Factories
- Services
- Providers
- Constants
- Values

Angular 2

- Class

vehicle.service.ts

```
@Injectable()
export class VehicleService {
  getVehicles() {
    return [
      new Vehicle(10, 'Millenium Falcon'),
      new Vehicle(12, 'X-Wing Fighter'),
      new Vehicle(14, 'TIE Fighter')
    ];
  }
}
```

Service is simply a class

Service

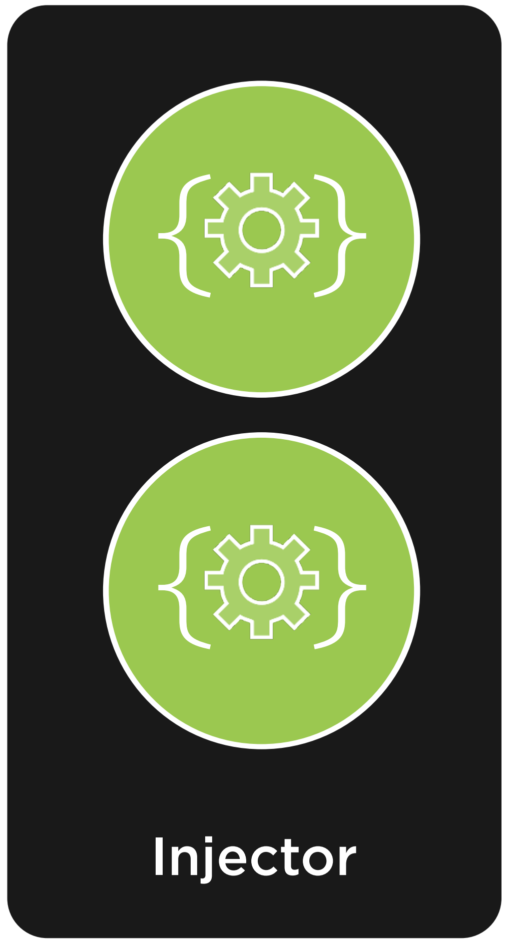
Provides something of value

Shared data or logic

e.g. Data, logger, exception handler, or message service

Demo

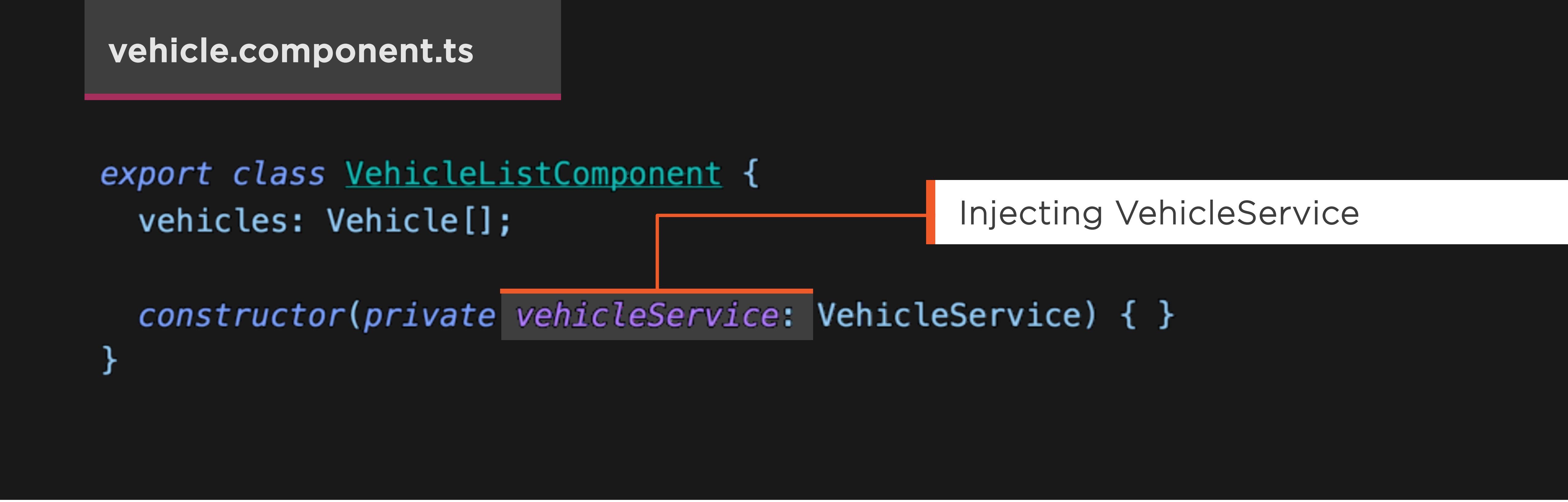
Services



Dependency Injection

Dependency Injection

Dependency Injection is how we provide an instance of a class to another Angular feature

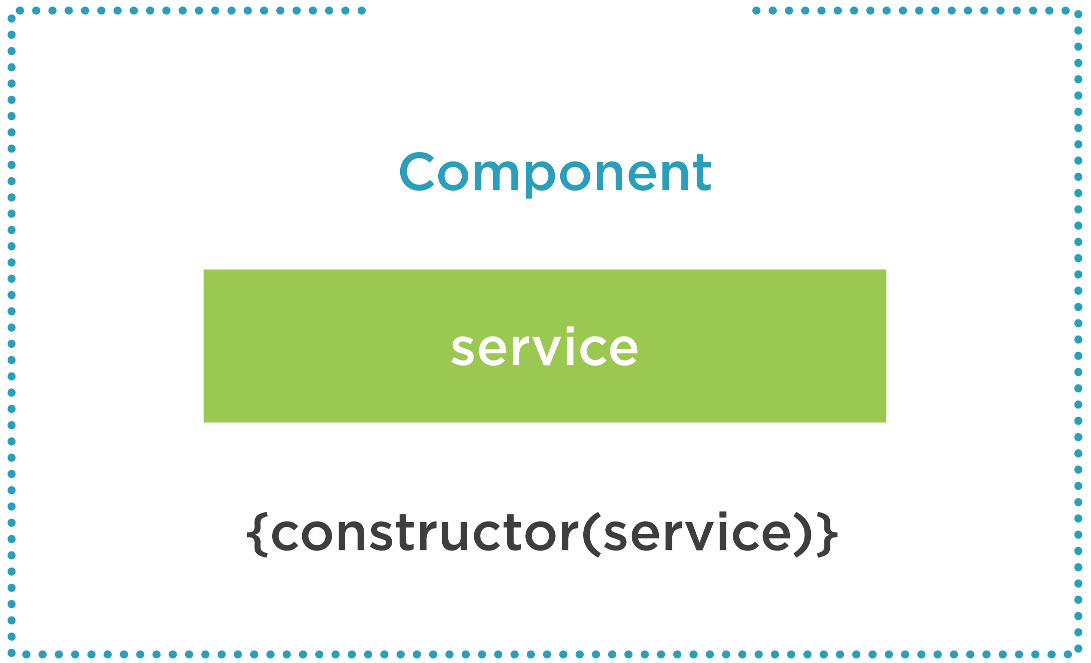

vehicle.component.ts

```
export class VehicleListComponent {
  vehicles: Vehicle[];

  constructor(private vehicleService: VehicleService) { }
}
```

Injecting a Service into a Component

Locates the service in an Angular injector

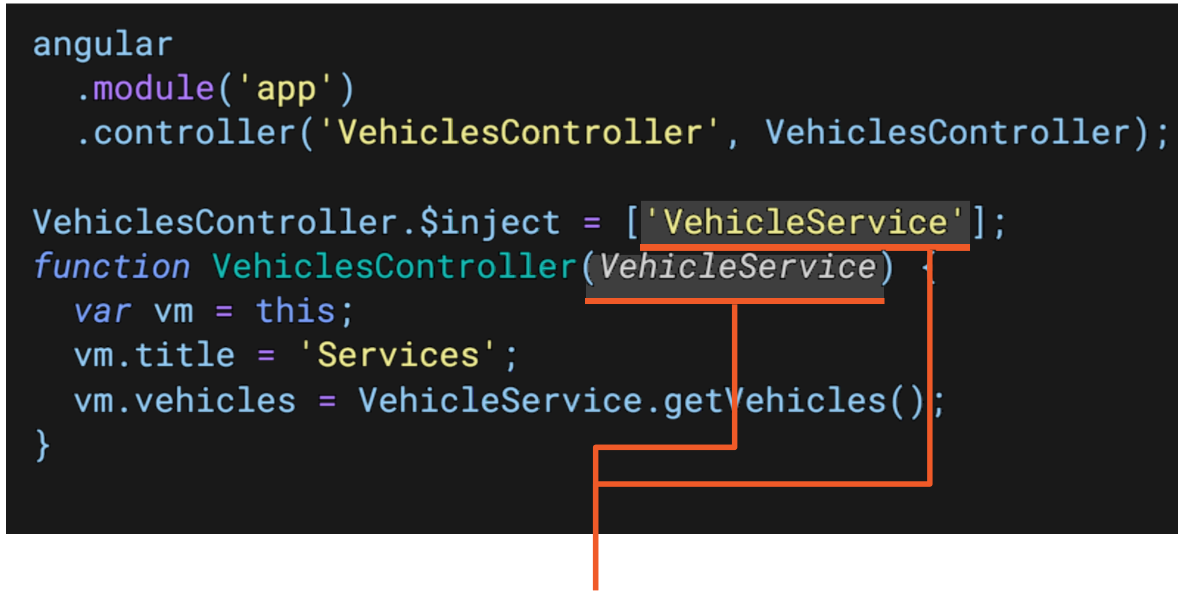
Injects the service into the constructor



Service is injected into the Component's constructor

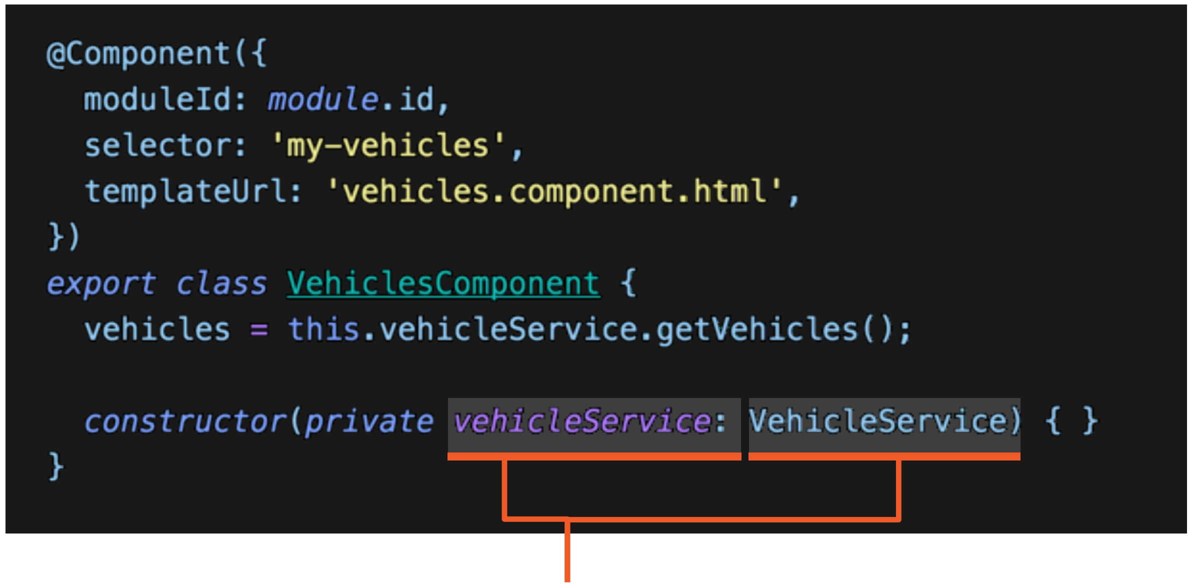
Dependency Injection Then and Now

Angular 1

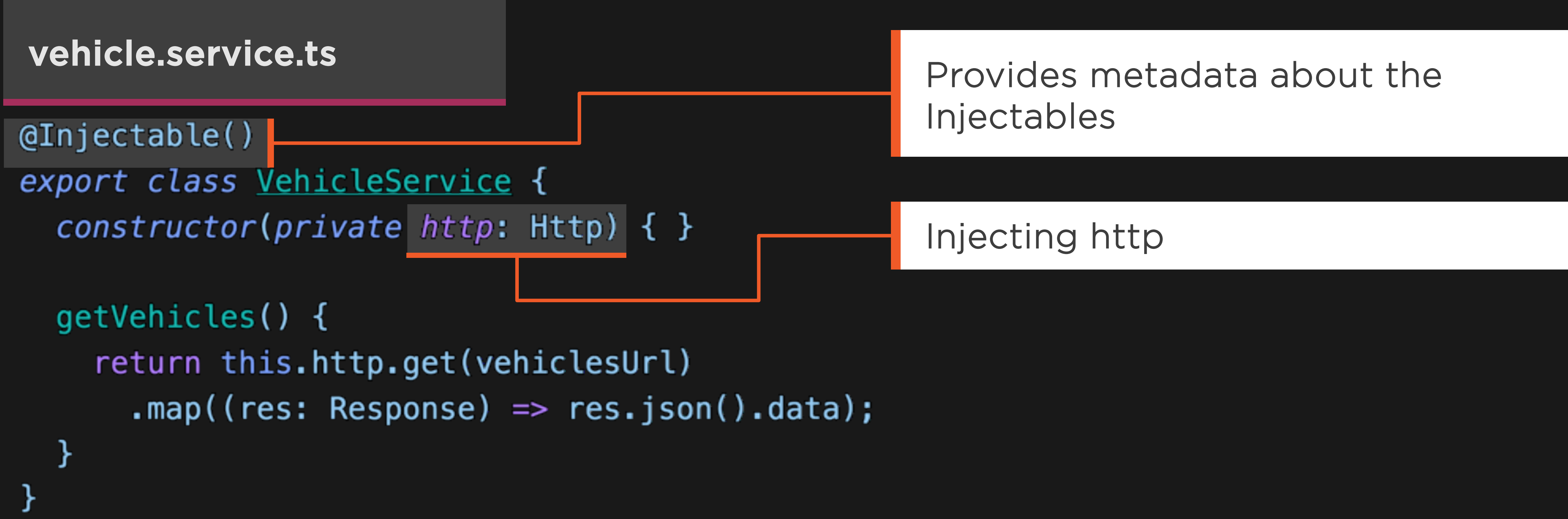


```
angular
  .module('app')
  .controller('VehiclesController', VehiclesController);

VehiclesController.$inject = ['VehicleService'];
function VehiclesController(VehicleService) {
  var vm = this;
  vm.title = 'Services';
  vm.vehicles = VehicleService.getVehicles();
}
```

Angular 2



```
@Component({
  moduleId: module.id,
  selector: 'my-vehicles',
  templateUrl: 'vehicles.component.html',
})
export class VehiclesComponent {
  vehicles = this.vehicleService.getVehicles();

  constructor(private vehicleService: VehicleService) { }
}
```


vehicle.service.ts

```
@Injectable()
export class VehicleService {
  constructor(private http: Http) { }

  getVehicles() {
    return this.http.get(vehiclesUrl)
      .map((res: Response) => res.json().data);
  }
}
```

Provides metadata about the Injectables

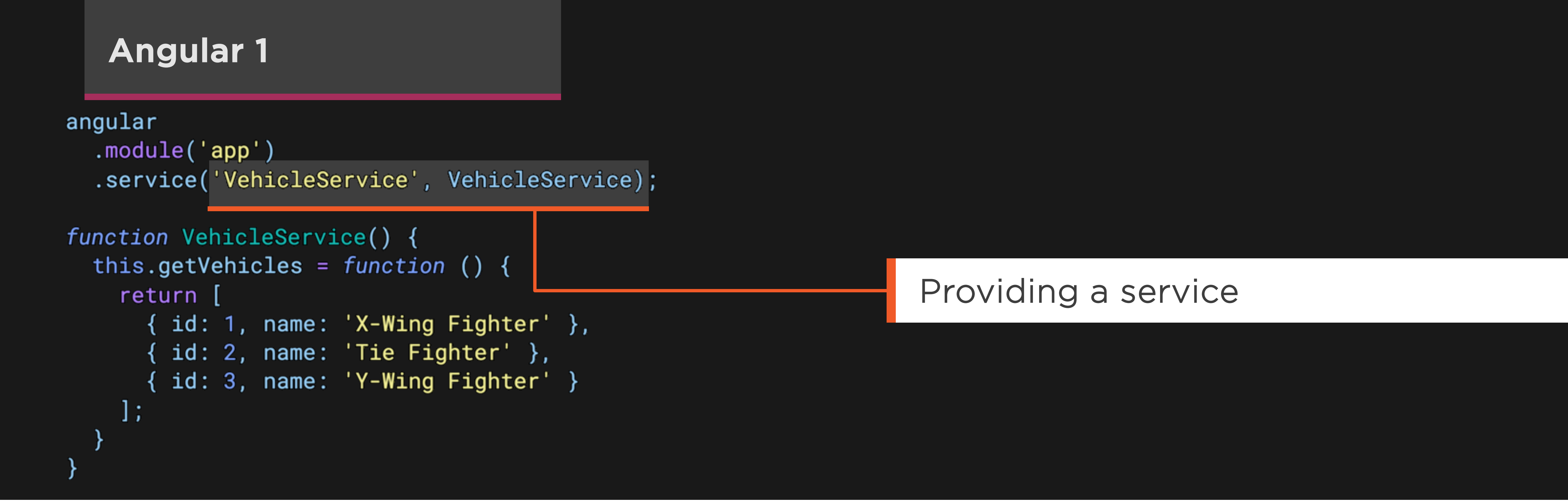
Injecting http

Injecting a Service into a Service

Same concept as injecting into a Component

@Injectable() is similar to Angular 1's $inject

We need to provide the service to an Angular injector


Angular 1

```
angular
  .module('app')
  .service('VehicleService', VehicleService);

function VehicleService() {
  this.getVehicles = function () {
    return [
      { id: 1, name: 'X-Wing Fighter' },
      { id: 2, name: 'Tie Fighter' },
      { id: 3, name: 'Y-Wing Fighter' }
    ];
  }
}
```

Providing Services in Angular 1

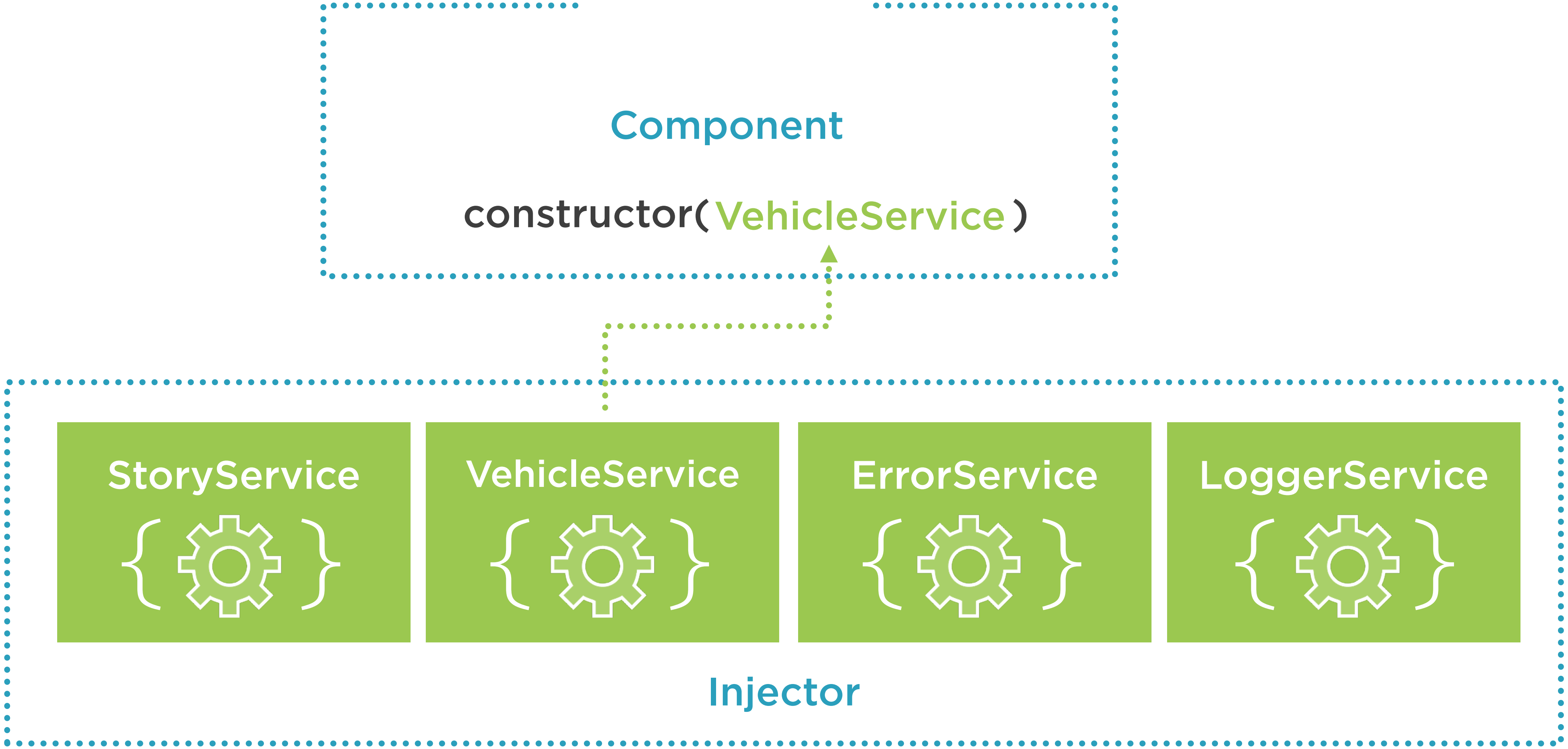
Angular 1 has one global injector

Angular 2 has hierarchical injectors and an injector at the app root

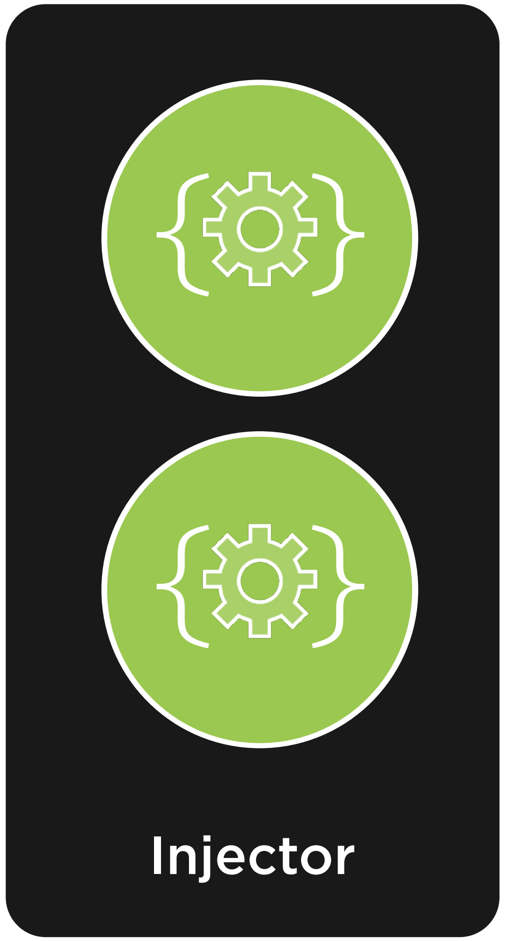
Providing a Service in Angular 2

The Service is now available in the root application injector

```
@NgModule({
  imports: [BrowserModule, FormsModule],
  declarations: [VehiclesComponent],
  providers: [VehicleService],
  bootstrap: [VehiclesComponent],
})
export class AppModule { }
```


Component
constructor(VehicleService)
StoryService
VehicleService
ErrorService
LoggerService
Injector

Injectors



We provide services to Angular's Injectors

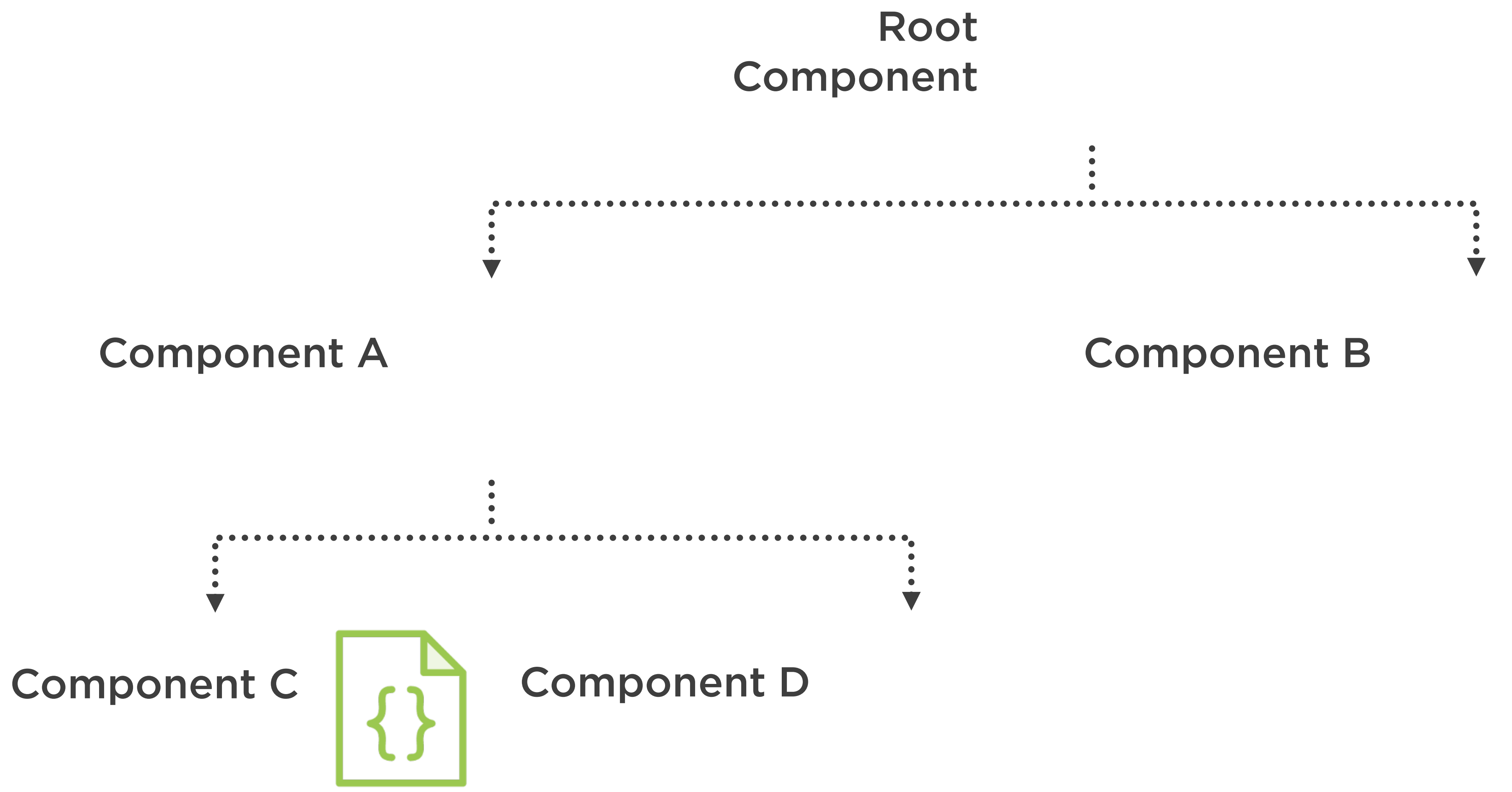
When we inject a service, Angular searches the appropriate injectors for it

One injector for the application root

One injector for the application root

And a hierarchical DI system with a tree of injectors that parallel an application's component tree

Hierarchical Components, Hierarchical Injectors

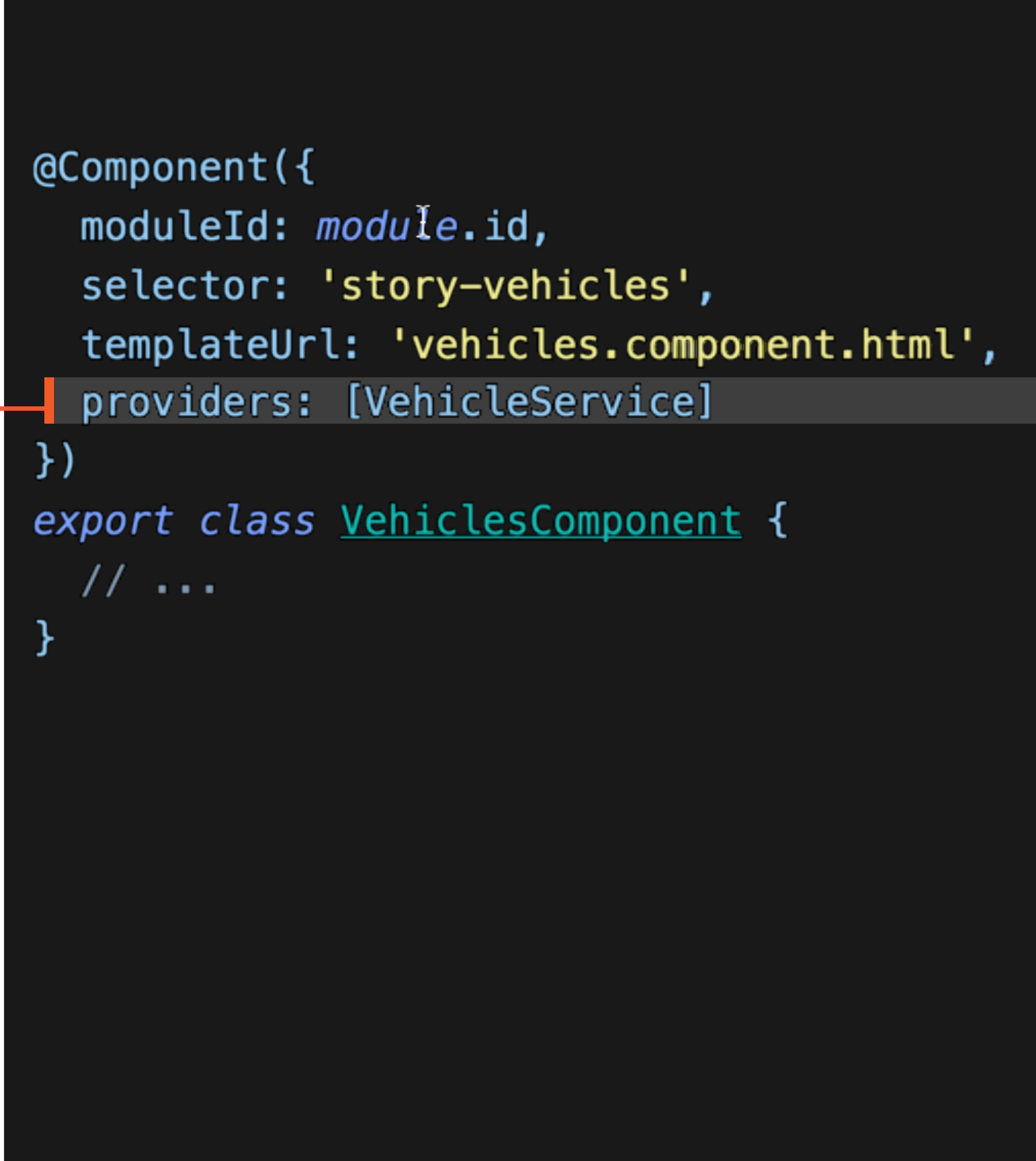


So where do we set the providers?

In the **Component** or an **Angular Module**?

Providing in a Component

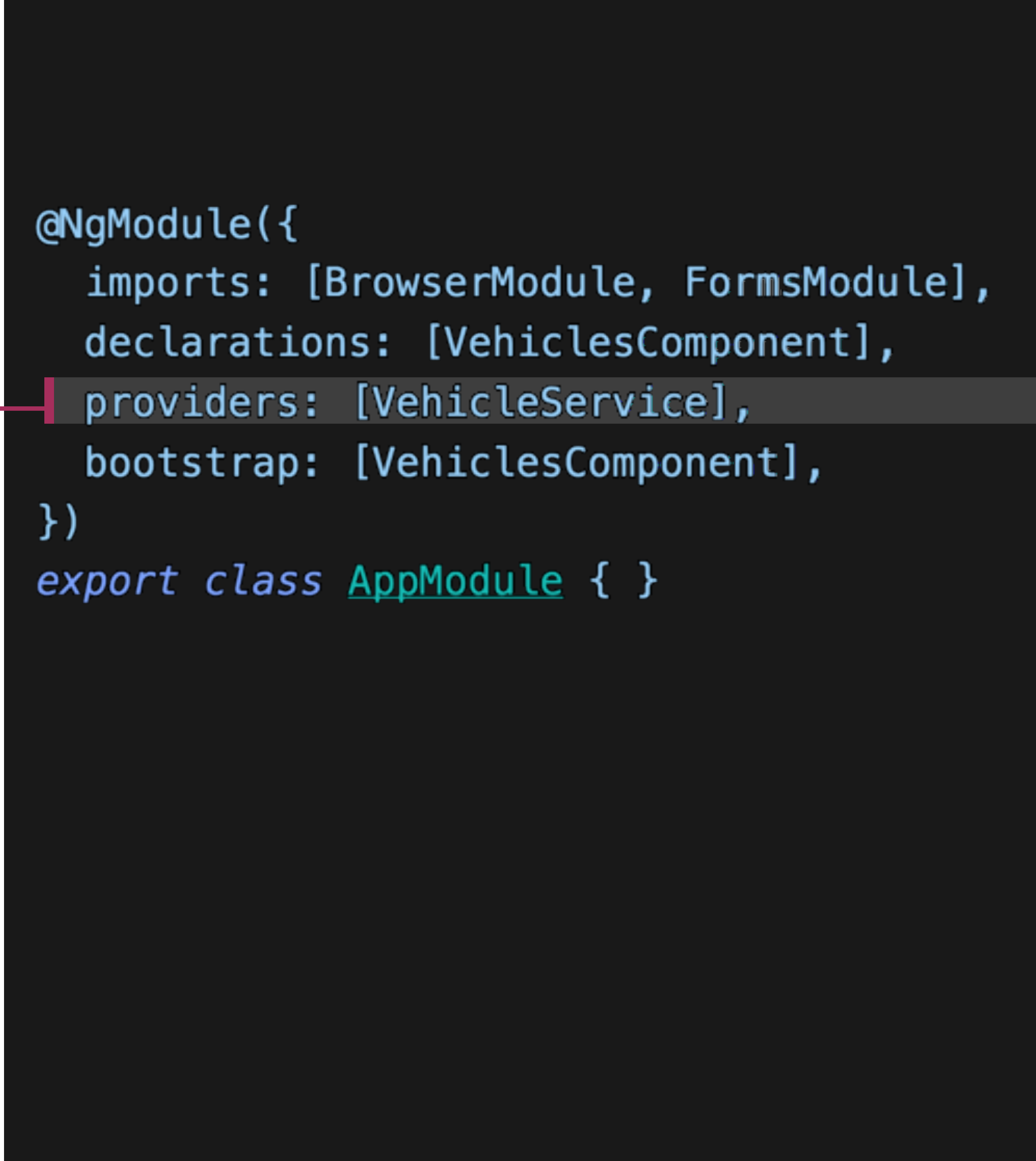
Available to this Component and any in its tree



```
@Component({
  moduleId: module.id,
  selector: 'story-vehicles',
  templateUrl: 'vehicles.component.html',
  providers: [VehicleService]
})
export class VehiclesComponent {
  // ...
}
```

Providing in an Angular Module

Eagerly and lazily-loaded modules and their components can inject the root AppModule services



```
@NgModule({
  imports: [BrowserModule, FormsModule],
  declarations: [VehiclesComponent],
  providers: [VehicleService],
  bootstrap: [VehiclesComponent],
})
export class AppModule { }
```

vehicles.component.ts

```
@Component({
  moduleId: module.id,
  selector: 'story-vehicles',
  templateUrl: 'vehicles.component.html',
  providers: [VehicleService]
})
export class VehiclesComponent {
  // ...
}
```

Providing to Component

app.module.ts

```
@NgModule({
  imports: [BrowserModule, FormsModule],
  declarations: [VehiclesComponent],
  providers: [VehicleService],
  bootstrap: [VehiclesComponent],
})
export class AppModule { }
```

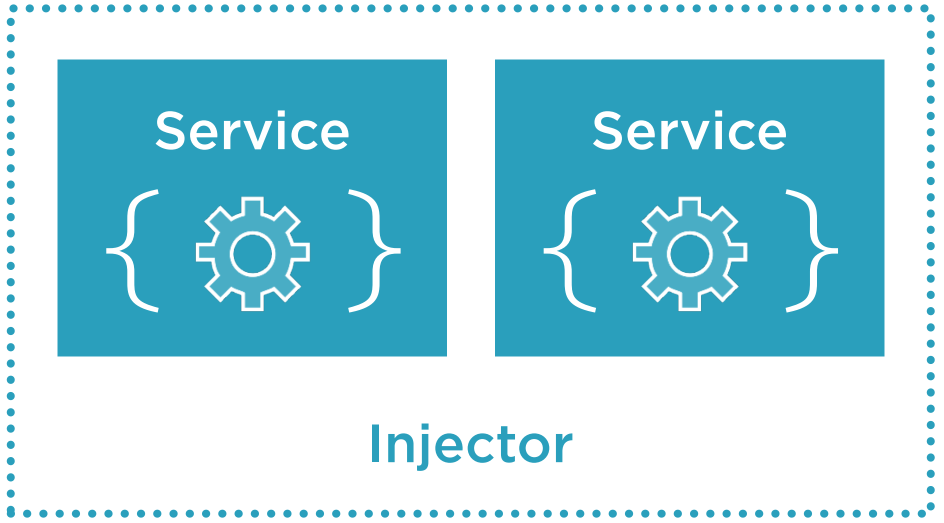
Providing to NgModule

Prefer registering providers in Angular Modules

Provide a service once,
if you want a singleton

Demo

Injectors



Component Lifecycle Hooks

Component Lifecycle Hooks

Lifecycle Hooks allow us to tap into specific moments in the application lifecycle to perform logic.

Interface

Implement the lifecycle hook's OnInit interface

Lifecycle Hooks

When the Component initializes, the ngOnInit function is executed

```
@Component({
  moduleId: module.id,
  selector: 'story-characters',
  templateUrl: 'characters.component.html',
  styleUrls: ['characters.component.css'],
  providers: [CharacterService]
})
export class CharactersComponent implements OnInit {
  @Output() changed = new EventEmitter<Character>();
  @Input() storyId: number;
  characters: Character[];
  selectedCharacter: Character;

  constructor(private characterService: CharacterService) { }

  ngOnInit() {
    this.characterService.getCharacters(this.storyId)
      .subscribe(characters => this.characters = characters);
  }

  select(selectedCharacter: Character) {
    this.selectedCharacter = selectedCharacter;
    this.changed.emit(selectedCharacter);
  }
}
```

Component Lifecycle Hooks

OnChanges

OnInit

AfterViewInit

OnChanges

OnDestroy

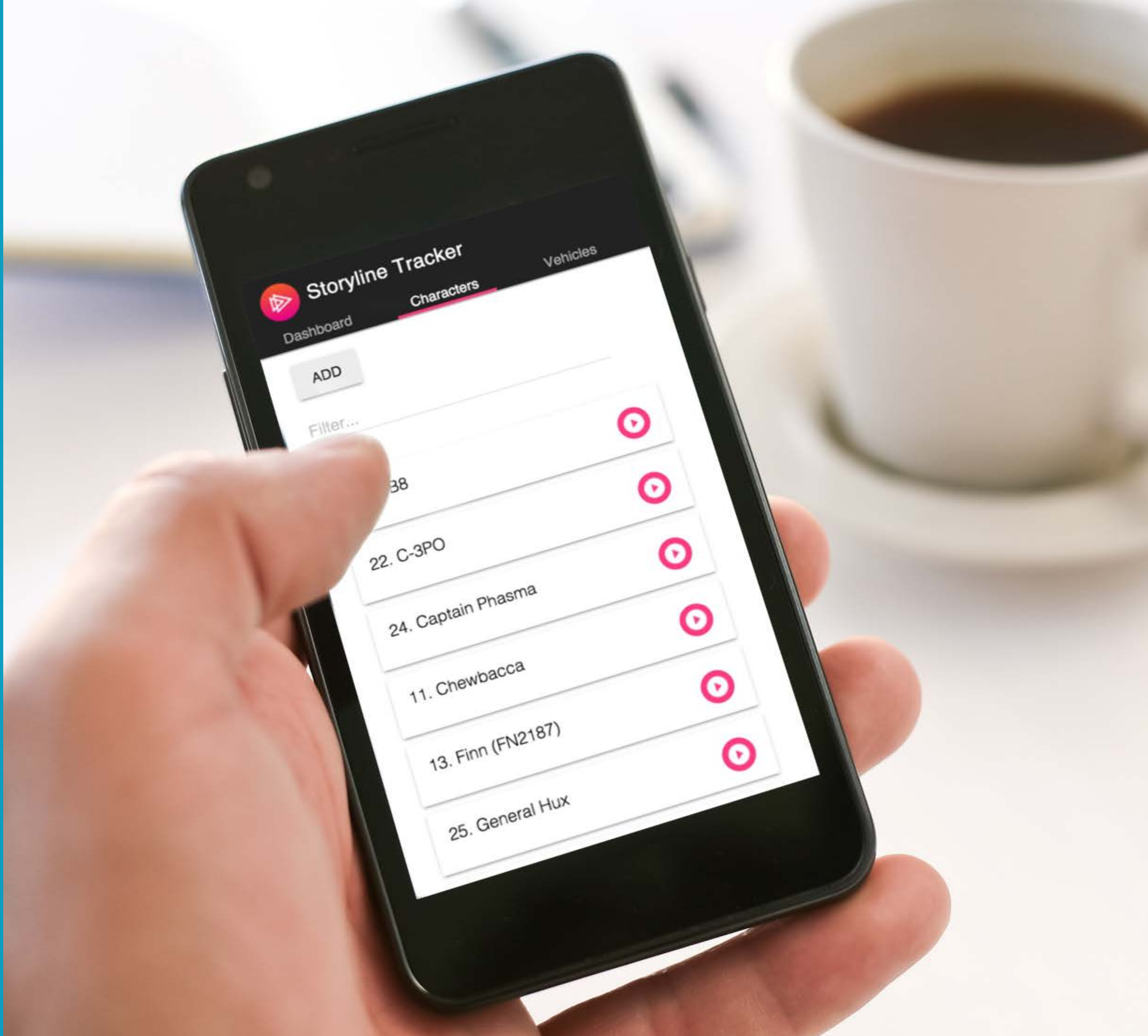
The Lifecycle Interface helps enforce the valid use of a hook

Demo

Component Lifecycle Hooks

Demo

Putting It All Together



Services, DI, and LifeCycle Hooks

Separation with Services

Sharing Instances

Registering with the Injector

Constructor Injection

Tapping into the Component's LifeCycle