

ACCELERATING THROUGH
ANGULAR 2

Level 5

Services

Section 1

Revisiting Our Mock Data

car-parts.component.ts

TypeScript

```
import { Component } from '@angular/core';
import { CarPart } from './car-part';
import { CARPARTS } from './mocks';

...
})
export class CarPartsComponent {
  carParts: CarPart[];

  ngOnInit() {
    this.carParts = CARPARTS;
  }
  ...
}
```

mocks.ts

TypeScript

```
import { CarPart } from './car-part';

export const CARPARTS: CarPart[] = [{
  "id": 1,
  "name": "Super Tires",
  "description": "These tires are the very best",
  "inStock": 5,
  "price": 4.99
}, { ... }, { ... }];
```

We are loading our CarParts by importing our mock file, but this isn't the best solution for working with data.

Why This Isn't the Best Method

- We'd need to import the mocks on every file that needs the data. If the way we access this data changes, we'd have to change it everywhere.
- It's not easy to switch between real and mock data.
- This sort of data loading is best left to service classes.

car-parts.component.ts

TypeScript

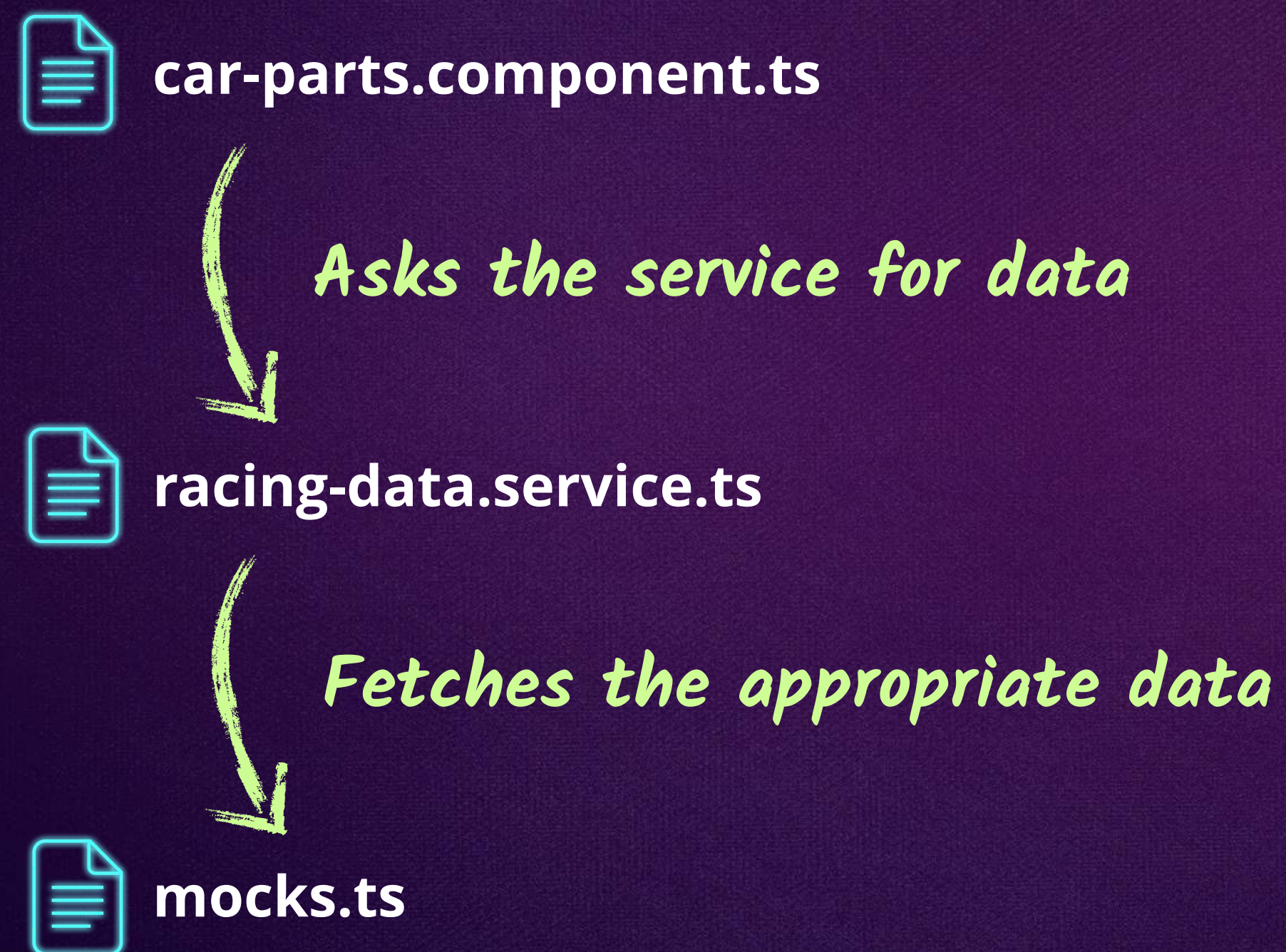
```
import { Component } from '@angular/core';
import { CarPart } from '../car-part';
import { CARPARTS } from '../mocks';

...
})
export class CarPartsComponent {
  carParts: CarPart[];

  ngOnInit() {
    this.carParts = CARPARTS;
  }
  ...
}
```


Introducing Services

Services are used to organize and share code across your app, and they're usually where we create our data access methods.



First, let's create the simplest service, and then we'll learn something called dependency injection to make it even more powerful.

Writing Our First Service

racing-data.service.ts

TypeScript

```
import { CARPARTS } from './mocks';

export class RacingDataService {
  getCarParts() {
    return CARPARTS;
  }
}
```

car-parts.component.ts

TypeScript

```
import { RacingDataService } from './racing-data.service';
...
export class CarPartsComponent {
  carParts: CarPart[];

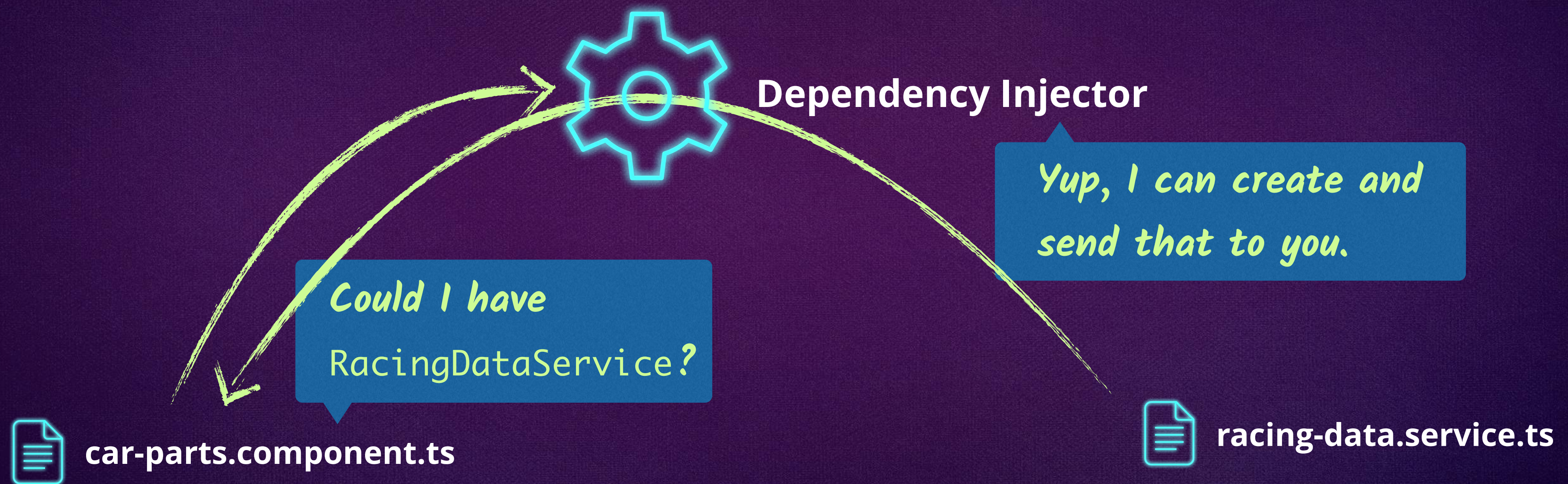
  ngOnInit() {
    let racingDataService = new RacingDataService();
    this.carParts = racingDataService.getCarParts();
  }
}
```



- ✓ We've decoupled our data.
- ✗ Classes using this service must know how to create a RacingDataService.
- ✗ We'll be creating a new RacingDataService every time we need to fetch car parts.
- ✗ It'll be harder to switch between a mock service and a real one.

Introducing Dependency Injection

When you run an Angular 2 application, it creates a dependency injector. An injector is in charge of knowing how to create and send things.



The injector knows how to inject our dependencies.

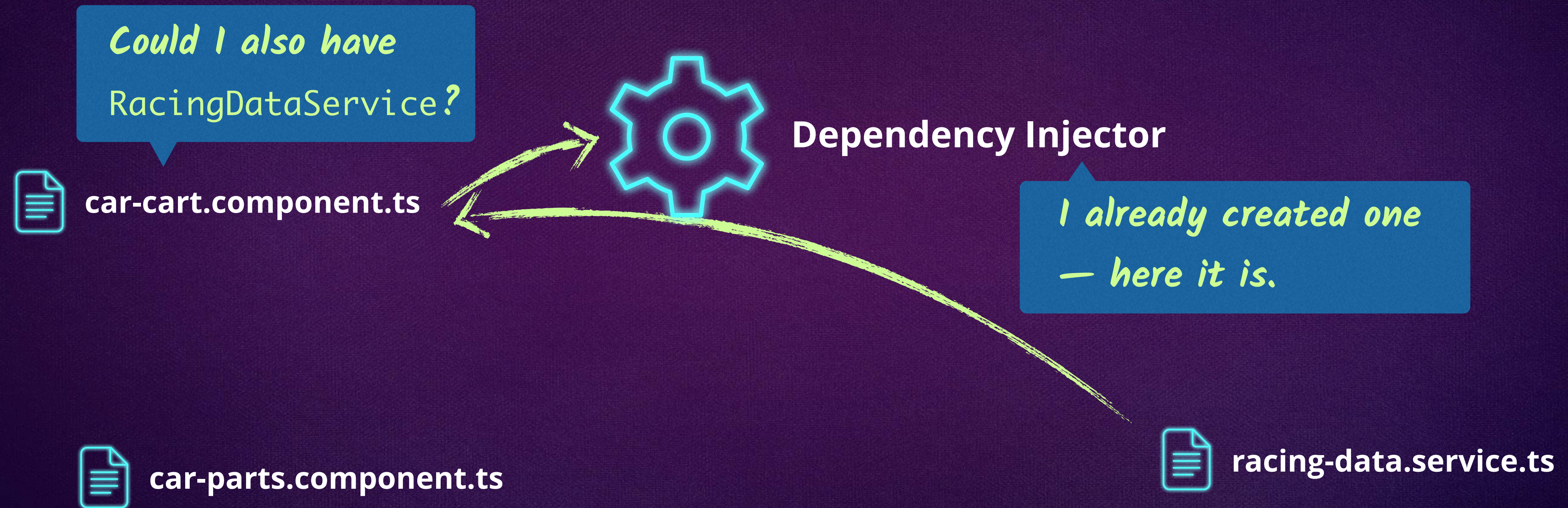
Create (if needed) and send

Classes we depend on

ACCELERATING THROUGH
ANGULAR 2

Re-injecting Dependencies

If the injector already created a service, we can have it resend the same service.

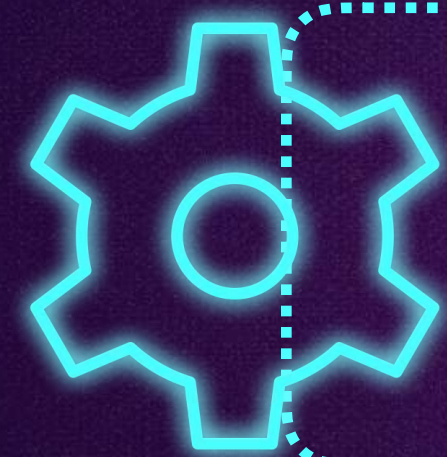


How Does an Injector Know What It Can Inject?

We must tell the injector what it can inject by registering “providers” with it.

These are the providers I have — they tell me what I can create and send.

Dependency Injector



rac`ing-data.service.ts`
another.service.ts
api.service.ts

There are three steps to make this all work with `RacingDataService`:

1. Add the injectable decorator to `RacingDataService`.
2. Let our injector know about our service by naming it as a provider.
3. Inject the dependency into our `car-parts.component.ts`.

Step 1: Add the Injectable Decorator

We need to turn our service into something that can be safely used by our dependency injector.

racng-data.service.ts

TypeScript

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

```
@Injectable()
```

Don't forget the parentheses!

```
export class RacingDataService {  
  getCarParts() {  
    return CARPARTS;  
  }  
}
```


Step 2: Let Our Injector Know About the Service

We want all our subcomponents to have access to `RacingDataService`. To do this, we register it as a provider at the top level — specifically, `AppComponent`.

app.component.ts

TypeScript

```
...
import { RacingDataService } from './racing-data.service';

@Component({
  selector: 'my-app',
  template: `...`,
  directives: [CarPartsComponent],
  providers: [RacingDataService]
})
```

Now all subcomponents can ask for (inject) our `RacingDataService` when they need it, and an instance of `RacingDataService` will either be delivered if it exists, or it will be created and delivered.

Step 3: Injecting the Dependency

car-parts.component.ts

TypeScript

...

```
import { RacingDataService } from './racing-data.service';
```

```
@Component({ ... })
```

```
export class CarPartsComponent {  
  carParts: CarPart[];
```

```
  constructor(private racingDataService: RacingDataService) { }  
}
```

We don't need to add RacingDataService as a provider since it's a subcomponent.

Means TypeScript automatically defines component properties based on the parameters. The generated JavaScript is:

```
function CarPartsComponent(racingDataService) {  
  this.racingDataService = racingDataService;
```

TypeScript syntax for setting the type of the parameter. This is what identifies that the RacingDataService should be injected into this component.

Using the Service to Fetch Our carParts

car-parts.component.ts

TypeScript

```
...
import { RacingDataService } from './racing-data.service';

@Component({ ... })
export class CarPartsComponent {
  carParts: CarPart[];

  constructor(private racingDataService: RacingDataService) { }

  ngOnInit() {
    this.carParts = this.racingDataService.getCarParts();
  }
}
```


Now our app is more scalable and testable.

- ✓ Scalable because our dependencies aren't strongly tied to our classes.
- ✓ Testable because it'd be easy to mock services when we test the component.

Our App Still Works

ULTRA RACING

There are 9 total parts in stock.



SUPER TIRES

These tires are the very best

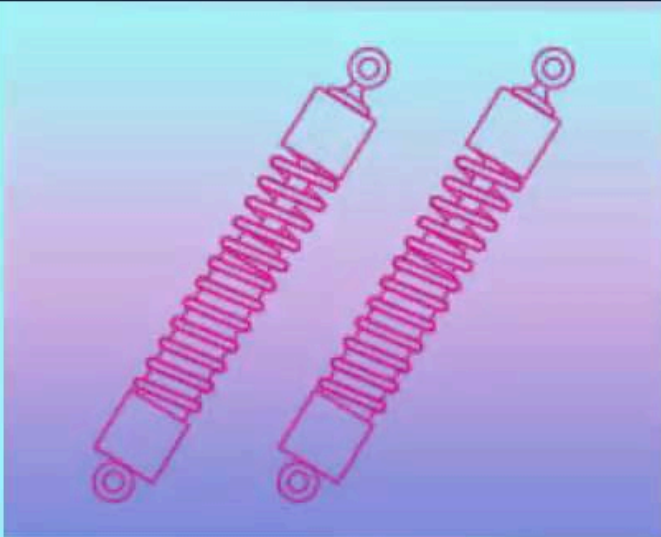
5 in Stock

€4.99

-

0

+



REINFORCED SHOCKS

Shocks made from kryptonite


4 in Stock

€9.99

-

0

+



PADDED SEATS

Super soft seats for a smooth ride

Out of Stock

€24.99

-

0

+

ACCELERATING THROUGH
ANGULAR 2

What'd We Learn?

- Services are used to organize and share code across your app, and they're usually where you create your data access methods.
- We use dependency injection to create and send services to the classes that need them.
- We give our dependency injector providers so that it knows what classes it can create and send for us.
- We ask for dependencies by specifying them in our class constructor.

ACCELERATING THROUGH
ANGULAR 2

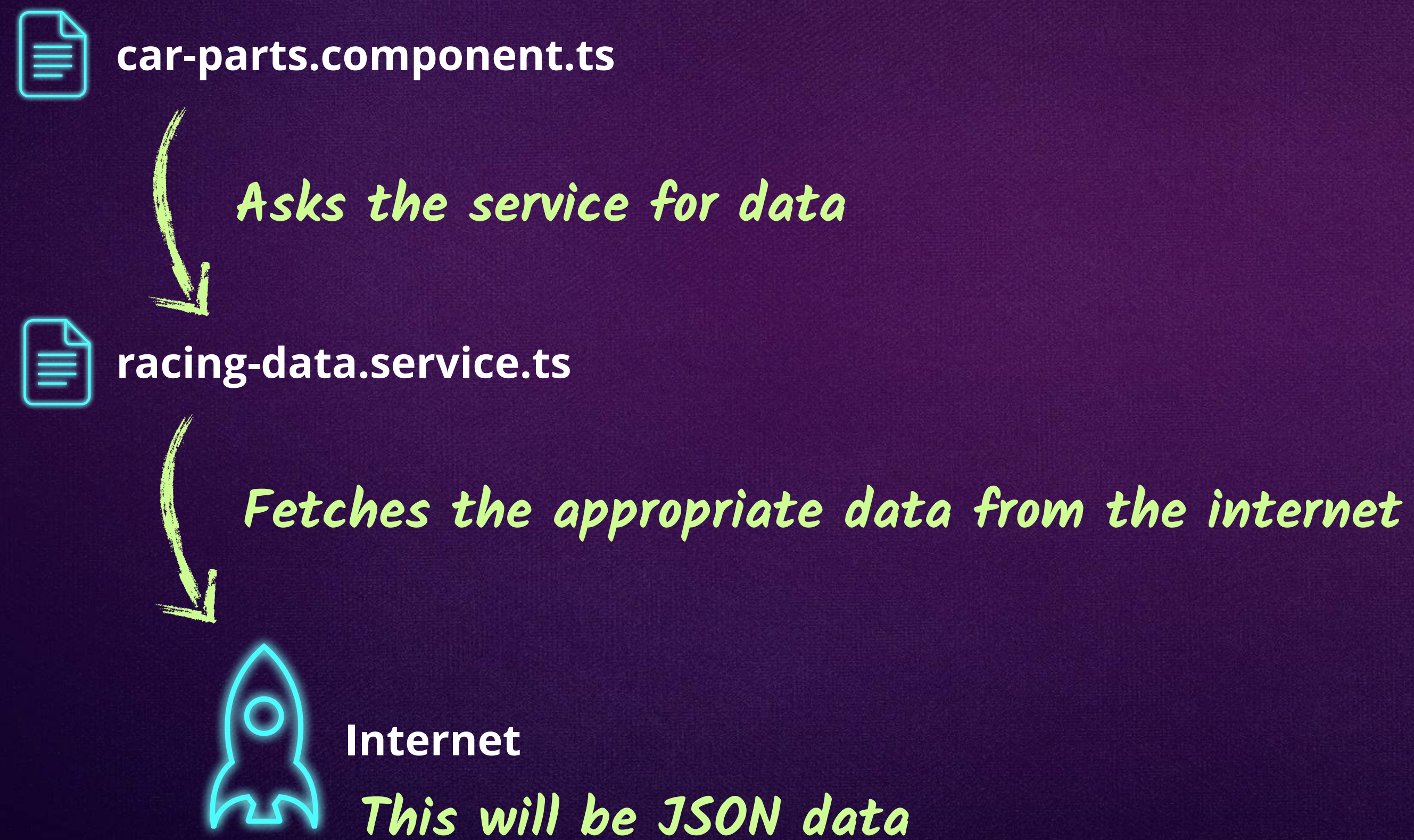
Level 5

Adding Http

Section 2

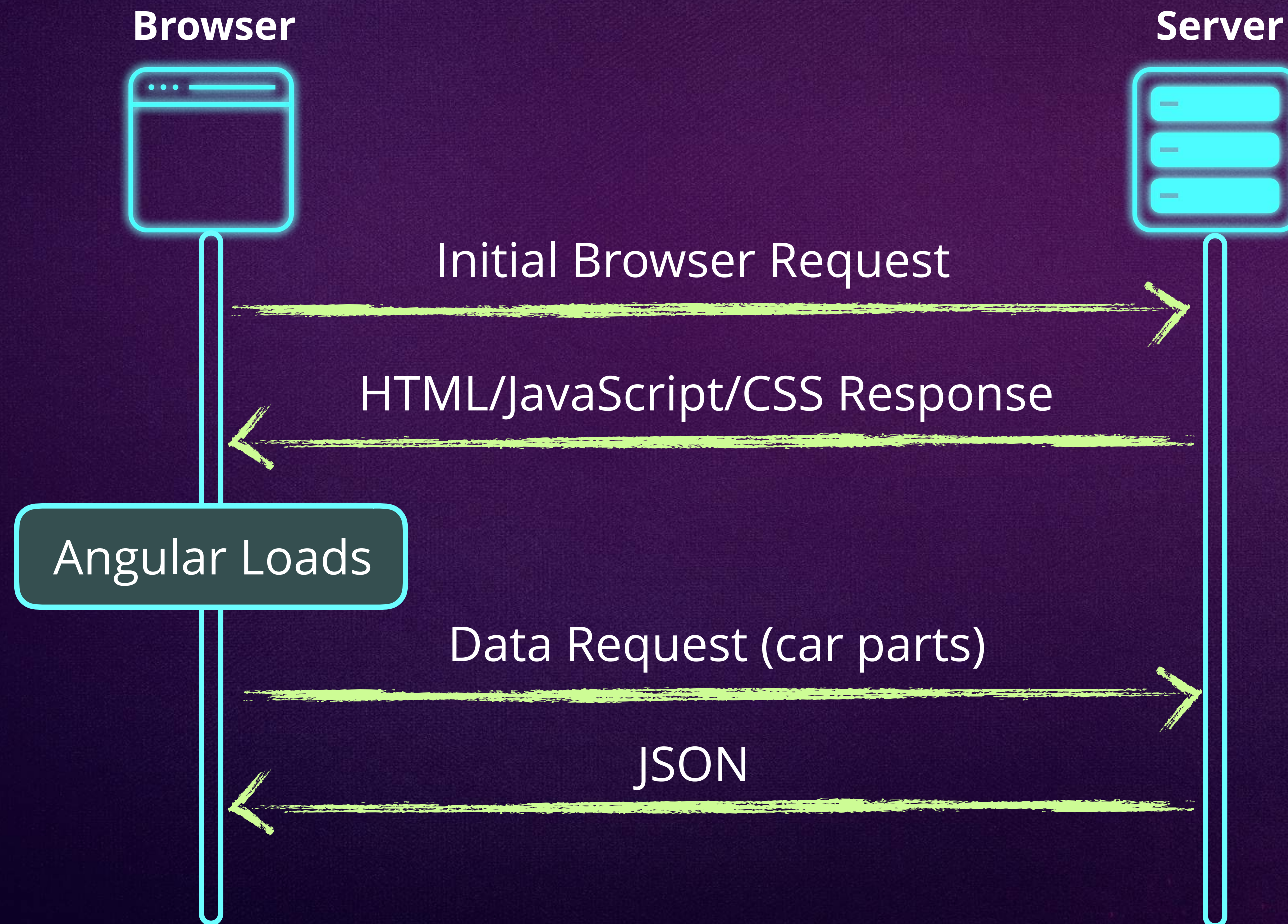
Let's Use the Internet!

Up until now, we've been seeding our car parts with mock data. How might we fetch this from the internet instead?



Welcome to the Real World

When a user visits our web page, the Angular app loads first in our browser, and then it fetches the needed data.



Steps Needed to Use the HTTP Library

1. Create a JSON file with car parts data.

 **car-parts.json**

2. Ensure our application includes the libraries it needs to do Http calls.

 **app.component.ts**

3. Tell our injector about the http provider.

 **racing-data.service.ts**

4. Inject the http dependency into our service and make the http get request.

 **car-parts.component.ts**

5. Listen for data returned by this request.

Step 1: Creating a JSON File

We need to create a JSON data file to load from our service.

car-parts.json

JSON

```
{  
  "data": [  
    {  
      "id": 1,  
      "name": "Super Tires",  
      "description": "These tires are the very best",  
      "inStock": 5,  
      "price": 4.99,  
      "image": "/images/tire.jpg",  
      "featured": false,  
      "quantity": 0  
    },  
    { ... },  
    { ... }  
  ]  
}
```

We wrapped our array in an object to make it feel a little more realistic.

Step 2: Including the HTTP and RxJS Libraries

The HTTP library provides the `get` call we will use to call across the internet.

The RxJS library stands for Reactive Extensions and provides some advance tooling for our http calls.

If you used the 5-minute Quickstart, you have already included these libraries using **SystemJS**.

Step 3: Telling Our Injector It Can Inject HTTP

As you know, the first step in dependency injection is to register providers.

app.component.ts

TypeScript

```
...
import { RacingDataService } from '../racing-data.service';
import { HTTP_PROVIDERS } from '@angular/http';

@Component({
  selector: 'my-app',
  template: `...`,
  directives: [CarPartsComponent],
  providers: [RacingDataService, HTTP_PROVIDERS]
})
```

Now we'll be able to inject the HTTP library when we need it.

Step 4: Injecting HTTP & Using It

Now let's inject http and call out to the internet.

racing-data.service.ts

TypeScript

```
import { Injectable } from '@angular/core';
import { CarPart } from '../car-part';
import { Http } from '@angular/http';
import 'rxjs/add/operator/map';
```

```
@Injectable()
```

```
export class RacingDataService {
```

```
  constructor(private http: Http) { }
```

```
  getCarParts() {
```

```
    return this.http.get('app/car-parts.json')
      .map(response => <CarPart[]>response.json().data);
```

```
  }
```

```
}
```

Injecting HTTP as a dependency

We can do this because our service

class is injectable

There's a lot going on here — let's break it out.

ACCELERATING THROUGH
ANGULAR 2

Stepping Through Our get Request

You might expect get to return a promise, but this returns an observable. Observables give us additional functionality on our http calls — one of which is to treat the return value like an array.

```
getCarParts() {  
  return this.http.get('app/car-parts.json')  
    .map(response => <CarPart[]> response.json().data);  
}
```

For the data returned,
do this to the response.

Tells our TypeScript
compiler to treat this like
an array of CarParts.

For each response, parse
the string into JSON.

The array we want is
under the data keyword.


Step 5: Subscribing to the Stream

Since our service now returns an observable object, we need to subscribe to that data stream and tell our component what to do when our data arrives.

car-parts.component.ts

TypeScript

```
...  
export class CarPartsComponent {  
  
    constructor(private racingDataService: RacingDataService) { }  
  
    ngOnInit() {  
        this.racingDataService.getCarParts()  
        .subscribe(carParts => this.carParts = carParts);  
    }  
    ...  
}
```



When carParts arrive on our data stream, set it equal to our local carParts array.

Solving the Problem

In our browser, we get nothing:

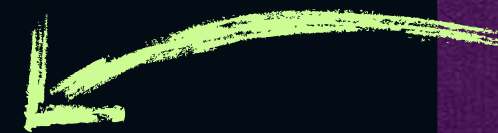


TypeError: Cannot read property 'length' of undefined at CarPartsComponent.totalCarParts

car-parts.component.ts

TypeScript

```
totalCarParts() {  
  let sum = 0;  
  for (let carPart of this.carParts) {  
    sum += carPart.inStock;  
  }  
  return sum;  
}
```



Not an array yet

When our page initially loads, we haven't yet fetched any data, so our carParts is **undefined**.

Checking for Undefined carParts

Let's make sure we only calculate sum if carParts is an array.

car-parts.component.ts

TypeScript


```
totalCarParts() {  
  let sum = 0;  
  
  if (Array.isArray(this.carParts)) {  
    for (let carPart of this.carParts) {  
      sum += carPart.inStock;  
    }  
  }  
  
  return sum;  
}
```



All Working — Now Over the Internet!

ULTRA RACING

There are 9 total parts in stock.



SUPER TIRES

These tires are the very best

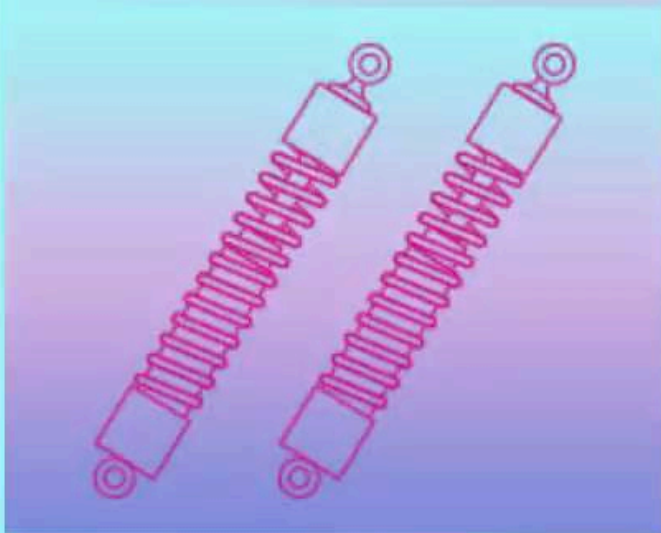
5 in Stock

€4.99

-

0

+



REINFORCED SHOCKS

Shocks made from kryptonite


4 in Stock

€9.99

-

0

+



PADDED SEATS

Super soft seats for a smooth ride

Out of Stock

€24.99

-

0

+

ACCELERATING THROUGH
ANGULAR 2

Last Thoughts on Implementation

1. We didn't do any error handling. If you're writing a production app, you'd want to do this.
2. Since we isolated our network calls as a service, we could easily write a `RacingDataServiceMock` service and inject it when we're testing or developing offline.
3. Observables can be quite powerful and are worth learning about if you are making lots of http calls.

What'd We Learn?

- Angular apps usually load data using service classes after the Angular app is initialized and running.
- We can use the HTTP library through dependency injection.
- Our http calls return an observable, not a promise, which behaves more like an array.

ACCELERATING THROUGH
ANGULAR 2