Services and APIs

Angular provides services to house data, functions, or features that the application might need in multiple components

---

Services

A Service is typically a class with a narrow, well-defined purpose. It should do something specific.

Angular makes a distinction between components and services to increase modularity and reusability.

Services are useful for...

- Providing access to APIs
- Keeping track of data that is shared between components.
- Implementing logic that is used throughout the application.

To create a service, run Angular's command:

```
ng generate service services/service-name
```

This can also be abbreviated to: **`ng g s services/service-name`**

If there isn't a services folder generated it will generate one for you.

The above example would generate 2 files:

- service-name.service.spec.ts
- Service-name.service.ts

A project can have multiple services, and services may depend on one another.

Dependency Injection

In order for a component (or another service) to access data or logic stored in a service, it must be injected with the service. In the components constructor, the desired service(s) can be listed:

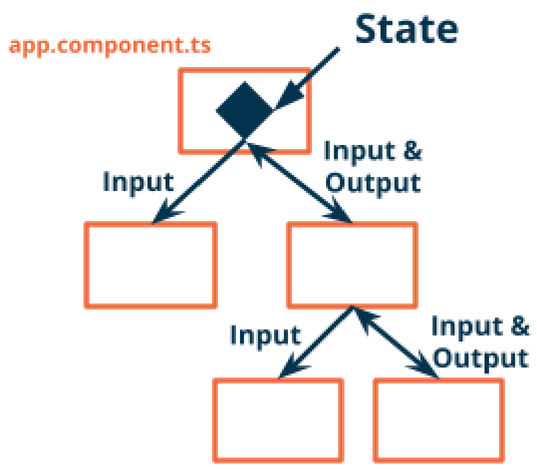
```
constructor(private favoritesService: FavoritesService) { }
```

Anything from this service can now be used locally in the component. Perhaps this Favorites Service included a variable called favoritePokemon that held a string. This would be available with the following dot notation:

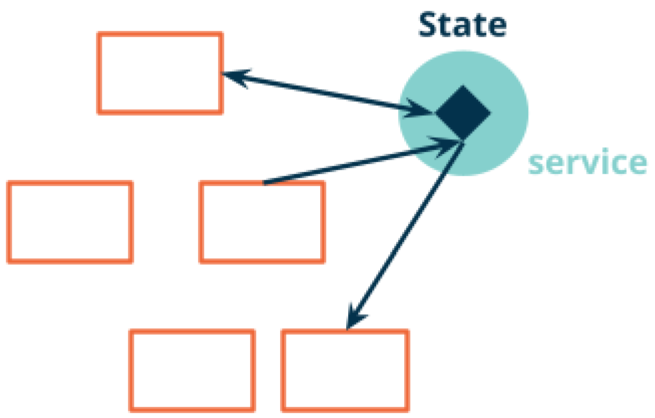
```
this.favoritesService.favoritePokemon
```

Storing data in a service

With Input and Output properties (without a service), state might be stored in a parent component and passed up and down to various components like so:



In contrast, with a service, state can be stored in that service. This means the data can be injected directly into whichever components need it, allowing for easier transferring of data:



NOTE: It is not always better to use a service for this. Each approach may be used in certain circumstances, and both may even be used together.

Suppose we had a service that held a “deck” of Pokemon, called Deck Service. In deck.service.ts, there might be a private variable called deck, that holds an array of Pokemon objects. This service might also hold a method called getDeck() that simply returns the deck.

In deck.service.ts:

```
export class DeckService {
  private deck: PokemonDetails[] = [
    { ... },
    { ... }
  ];

  constructor() {}

  getDeck(): PokemonDetails[] {
    return this.deck;
  }
}
```

And in any component where that deck might be used:

```
export class DeckComponent implements OnInit {

  deck: PokemonDetails[] = [];

  constructor(private deckService: DeckService) { }

  ngOnInit(): void {
    this.deck = this.deckService.getDeck();
  }
}
```

There might even be a method in the service that allows a component to modify the state that lives in the parent:

deck.service.ts:

```
  addToDeck(pokemon: PokemonDetails): void {
    this.deck.push(pokemon);
  }
```

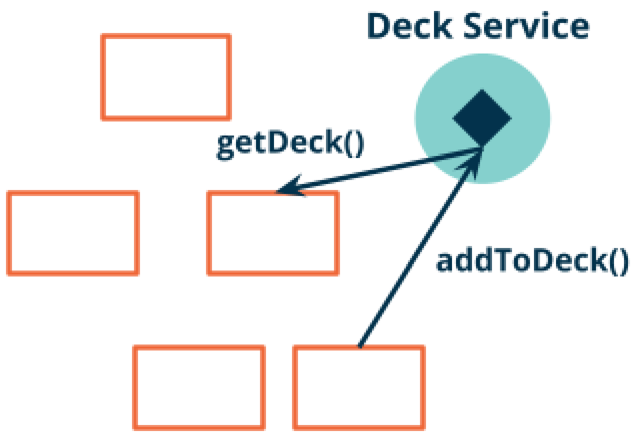
And used in another component:

```
export class PokemonDetailsComponent implements OnInit {

  constructor(
    private deckService: DeckService
  ) {}

  addToDeck(): void {
    if (this.pokemon) {
      this.deckService.addToDeck(this.pokemon);
    }
  }
}
```

To wrap up this example's concept, the Deck Service exists independently, available to any component, and any component injected with the service can receive data (deck) that lives in Deck Service, utilizing getDeck() and can also modify the deck that lives in the service, by utilizing addToDeck().



API calls

APIs allow software and devices to communicate across a network, such as the internet. If you're not yet familiar with APIs see Lesson Summary: What is an API?.

Angular has an HttpClient feature that we'll use to make API calls. As a standard practice we'll make our API calls from services.

(If you need some sample API data to play with, here are some APIs that don't require a key...)

- REST Countries (example)
- Grand Circus Demo APIs
- Juneteenth Quotes API
- GitHub (example)
- Random Joke (or ten)
- Pokemon API (example)
- Wikipedia (example)
- Weather.gov API (example)
- Star Wars (example)
- Reddit (example)

Modeling data

When receiving data from an API, it is necessary to model that data with interfaces. There are a couple tools that make this process more seamless:

- Quicktype

    - Paste the JSON from the API.
    - Adjust the name. Select TypeScript.
    - Copy only the interfaces.
- "Paste JSON as Code" A Visual Studio Code plugin
    - Copy the JSON
    - Create an interface
    - Click ctrl + shift + p

Setup

In your app.config.ts you must add provideHttpClient() to the array of providers.

```
export const appConfig: ApplicationConfig = {
  providers: [provideRouter(routes), provideHttpClient()]
};
```

Once you've created your service, you will need to dependency inject the HttpClient to your service or component.

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

Observables

API calls are asynchronous. This means that your program does not stop running while they go out across the internet to fetch data and return. For example, this would not work because "data =" and console.log would both run before and results from the API had come back...

```
let data = callApi();
console.log(data); // THIS DOES NOT WORK. No data yet.
```

To handle this, Angular includes a library called RxJS which has **Observable**s. An Observable is an asynchronous provider of data. We **subscribe** to the Observables.

```
callApi().subscribe(data => { // This runs after data comes back.
    console.log(data); // This works!
}
```

You will see code examples of this in the API Calls section below.

API Calls

In the service we make a method to call the API. This will return an observable of the data the API should return. See Modeling Data.

```
getPokemon():Observable<Pokedex>{
  return this.http.get<Pokedex>("https://pokeapi.co/api/v2/pokemon/?limit=200");
}
```

Consider using string interpolation on your URLs if you have parameters or are reusing the url for multiple different API calls.

```
  baseUrl:string = "https://pokeapi.co/api/v2";

  getPokemon():Observable<Pokedex>{
    return this.http.get<Pokedex>(`${this.baseUrl}/pokemon/?limit=200`);
  }
  getPokemonByName(name:string):Observable<PokemonDetailed>{
    return this.http.get<PokemonDetailed>(`${this.baseUrl}/pokemon/${name}`);
  }
```

Back in the component you want the data in, you will need to first dependency inject the service.

```
constructor(private pokemonService:PokemonService){}
```

With access to the service, we will now call the API call method, then subscribe to the response. This can be done in whatever method you need, but it is most commonly done in ngOnInit.

IF WE DON'T SUBSCRIBE, WE WON'T GET ACCESS TO THE DATA. This is because the request doesn't even start until you subscribe to it.

```
pokedex:Pokedex = {} as Pokedex;

  ngOnInit(){
    this.callApi();
  }

  callApi(){
    this.pokemonService.getPokemon().subscribe((response) => {
        console.log(response); //Optional, but helps with debugging
        this.pokedex = response;
    });
  }
```

You should now have access to the data!

Query parameters

Query parameters may be included directly in the URL string like this.

```
getPokemon(limit: number):Observable<Pokedex>{
  // this can be risky...
  return this.http.get<Pokedex>(`${this.baseUrl}/pokemon/?limit=${limit}`);
}
```

However, this can introduce some risks if the variables may contain invalid URL characters. To safely handle all values, use the **params** option. HttpClient will encode these values safely.

```
getPokemon(query: string, limit: number):Observable<Pokedex>{
  return this.http.get<Pokedex>(`${this.baseUrl}/pokemon/`, {
    params: { query: query, limit: limit }
  });
}
```

Headers

Some APIs may need headers to be included with the request, for example to pass an API key. Include these with the **headers** option.

```
getPokemon():Observable<Pokedex>{
  return this.http.get<Pokedex>(`${this.baseUrl}/pokemon/`, {
    headers: { "X-API-Key": apiKey }
  });
}
```

Additional Resources

- Understanding dependency injection - Angular Guide
- Creating an injectable service - Angular Guide
- Understanding communicating with backend services using HTTP - Angular Guide
- https://app.quicktype.io/?l=ts - Json -> TypeScript
- Paste JSON as Code - Visual Studio Code plugin