Services

Introduction to Services

Services are a type of component in Angular that are specifically designed to create reusable logic that can be injected into multiple components. This makes them ideally suited for the task of compartmentalizing API interactions, and so that is their most common use case.

When you use a service to hold your API interactions, you can easily add additional features to your API methods such as caching, and allow your components to subscribe to a single instance of an API call, preventing unnecessary trips to the server.

A common usage scenario for a service in Angular is to fetch data from a data source. This could be a database running on a server somewhere, or it could be any other form of data that you Angular application needs. In the lab for this topic, you will create a service to retrieve simple data from a web site, in JSON format.

Creating a Service

The very first thing you do, after designing your service, is to create one using the Angular CLI. This is accomplished with a simple command:

```
ng generate service service_name
```

This single command use the Angular CLI, calls the generate command to generate a service and gives it a name of service_name. You should, of course, ensure that you are using descriptive names for your services.

This will generate a file called service_name.service.ts in the src/app folder of your project.

If you open that folder, you should see the following:

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

@Injectable()
export class service_nameService {

  constructor() { }

}
```

Notice that our service imports Injectable from @angular/core, which allows this service to be injectable into other components. Recall that we want our service to be usable by other components and the Angular pattern recommendations is to make the services injectable so they can be used across components, share data, and be a singleton for instantiation when a component needs their functionality.

Using a Service

Once you have a service created, and it is an injectable component, you can make use of it in your components. One way to do this is to add it to the providers: section in your component, as shown here:

```
@NgModule({
  declarations: [
    AppComponent
  ],
  imports: [
    BrowserModule,
    HttpClientModule
  ],
  providers: [service_name],
  bootstrap: [AppComponent]
})
export class AppModule { }
```

Note that this code segment is not complete as we are missing the imports but the it is intended to show where we would place our service so that the Angular framework will inject it into our component.

You will also need to ensure that the service is imported in the app module as well. This will be covered in the tutorial labs where you will create a search service, inject it into your components, and then utilize in the application, to return data from a web site.

Tutorial Lab Creating Services

To Create a Service

In this lab, we will be creating a Service to hold our method for accessing the GitHub Search API call that we manually executed in the previous lab.

1. Open up a command prompt or terminal.

2. Navigate to your angular-fundamentals folder.

3. Run the following command:

```
ng generate service git-search
```

This will generate a file called git-search.service.ts in the src/app folder of your project. Note that we used the same name as when we generated the interface. You will note that the Angular generator automatically appends the type of object you are generating to the filename (e.g. service, component, module, etc.). This is useful as you can name related files similarly and keep those files in a single directory without worrying about naming conflicts.

4. Open up your git-search.service.ts file. You should see the following:

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

@Injectable()
export class GitSearchService {

  constructor() { }
```

```
}
```

You can see that the Injectable decorator has been imported from @angular/core. This allows you to inject this service into other components. You can use this decorator on other types of classes, but it should always be included in a service.

Now we have our service generated and ready to go. In our next lesson, we will take this service and inject the interface we created in the previous lab.

To Import an Interface into Your Service

Now that we have our service ready to go, we can import the interface we made in the previous task. When we do this, we can give the Service the shape of the expected output of our API call, and every component that we subsequently inject our Service into will be provided with that information as well. This will especially come in handy later when we are building out the views for our application.

Let's go ahead and import the Interface into our Service.

1. Open up your git-search.service.ts file in Visual Studio Code.

2. Go ahead and open up your git-search.ts file (the file containing your interface) as well.

3. Take a look at the top line of your git-search.ts file. You will see the following:

```
export interface GitSearch {
```

Using the export statement in this fashion creates an export object with GitSearch as a parameter. It will need to be imported using curly braces in an import statement {}. Curly braces allow you to have multiple exports in the same file, and to import them separately on a single import statement.

4. Let's add the import statement to the top line of git-search.service.ts.

```
import { GitSearch } from './git-search'
```

As the git-search.ts file that implements the interface is in the same directory as the git-search.service.ts file, you can use the relative directory path ./ to indicate that the file is in the same directory. You also do not need to include the .ts extension on the end of the file as TypeScript assumes this. Note that the name of the interface in the import statement matches the name of the interface in the export statement in git-search.ts. As we are importing types using the curly braces {} syntax, you need to ensure that the names in the import match the names being exported.

5. When you are finished, your service should now look like this:

```
import { Injectable } from '@angular/core';
import { GitSearch} from './git-search'
@Injectable()
export class GitSearchService {

  constructor() { }

}
```

Our interface is now imported into our service and ready to go. In our next task we will create a method that will eventually be used to call the API.

To Create a Service Method

n the previous task, we created a service and imported our GitSearch Interface. Since we have our service created and our interface imported into it, we are now going to do two things: prepare a method to be used within our service to access the GitHub API, and set up caching within that method. Caching within a service is relatively easy, and allows you to be able to save data to be used later, preventing subsequent API calls and allowing those repeated calls to be delivered much quicker.

Note that at the end of this task we will have our method set up to be used, but it won't be talking to the API just yet. In the next lab we will be taking this method and adding the HttpClient to access the GitHub Search API.

1. Open up your git-search.service.ts file.

2. Above our constructor function, let's add an empty array to contain the cached values. This will set up a cache for our service the first time it is injected.

```
import { Injectable } from '@angular/core';
import { GitSearch } from './git-search'
@Injectable()
export class GitSearchService {
   cachedValues: Array<{
       [query: string]: GitSearch
    }> = [];
 constructor() {
  }

}
```

You can see we went ahead and described the shape of the objects we are going to be storing inside of the array. In this case the key of each object in the array will be the query passed to the function - so it subsequently has a type of string - and the value which will be the actual output from the API - marked with the type of GitSearch, which is using our previously imported interface.

3. Let's go ahead and add our method next.

```
import { Injectable } from '@angular/core';
import { GitSearch } from './git-search'
@Injectable()
export class GitSearchService {
   cachedValues: Array<{
      [query: string]: GitSearch
   }> = [];
 constructor() {

  }
```

```
  gitSearch = (query: string) => {

  }
}
```

The gitSearch function we have just created is scoped to the this variable of the service, so if we were referencing it from within the service we would reference this.gitSearch. Outside of the service, in other components, it would be referenced as GitSearchService.gitSearch.

4. Next we are going to create a Promise within our function to get resolved by either the cache value or the API call value.

```
gitSearch = (query: string) => {
   let promise = new Promise((resolve, reject) => {

   })
   return promise;
  }
```

Note the two parameters of the Promise - resolve and reject. You will call resolve with successful data calls, whereas reject would be used for manual error handling.

5. Inside of our promise, we want to add a check to see if we want to use the cached value instead of querying for the response.

```
 gitSearch = (query: string) => {
   let promise = new Promise((resolve, reject) => {
      if (this.cachedValues[query]) {
         resolve(this.cachedValues[query])
      }
   })
   return promise;
 }
```

This will check the array that we created earlier for cached values with the same search query as what is being currently searched. This will get returned immediately without querying the API.

6. If a value isn't cached, we will want to return the call from the API. For now, we'll just put a direct resolve call into an else block.

```
 gitSearch = (query: string) => {
   let promise = new Promise((resolve, reject) => {
      if (this.cachedValues[query]) {
         resolve(this.cachedValues[query])
      }
      else {
         resolve("Placeholder");
      }
   })
   return promise;
 }
```

Note: What we have demonstrated here is a very naive caching method that caches the values for the lifetime of the application instance. For long running applications or APIs where the return values could

change frequently (i.e. stock price) this approach would be problematic. However, this approach could be extended - for example - by storing the time the query was last executed. The code could then check to see if the cached data was older than a predefined limit (say 10 minutes). If the cached data had expired, it could then be discarded, a new API call made, and the new data saved in the cache and returned.

7. When you're finished, you should have something that looks like this:

```
import { Injectable } from '@angular/core';
import { GitSearch } from './git-search'
@Injectable()
export class GitSearchService {
   cachedValues: Array<{
        [query: string]: GitSearch
   }> = [];
  constructor() {

  }

  gitSearch = (query: string) => {
    let promise = new Promise((resolve, reject) => {
       if (this.cachedValues[query]) {
          resolve(this.cachedValues[query])
       }
       else {
          resolve("Placeholder");
       }
    })
    return promise;
  }
}
```

So now we have a service created, our methods ready, and our interfaces injected. In our next lab, we will be replacing the placeholder resolve call with a real API call.