Seneca
SCHOOL OF INFORMATION AND COMMUNICATIONS TECHNOLOGY

BTI425

Web Programming for Apps and Services

Notes	Weekly
Resources	Graded work
Policies	Standards
Professors & addendum	Code examples

Angular services, more coverage

In Angular, a *service* is a class that provides value and functionality to your app's components. It does not have a user interface. Often its main task is to perform data service operations (e.g. fetch, add, edit, transform, etc.).

A service can be used by *any* of your app's components. Its use promotes a layered system architecture, also known as a separation of concerns. Enables you to write the code once, and use it in many places.

Recently, you had a brief and gentle introduction to an Angular service. This document adds more coverage of this important topic.

Multiple components, and the role of services

Consider a scenario where an app has many components. Some of the components display data that's stored in a database somewhere out there. The data could be fetched in a couple of ways:

1. Wrong way - add the same data-handling code to *each* component (so that multiple components have the same repeated block of data-handling code)
2. Right way - add the data-handling code to a single *service*, and then call the service from each component

A component should be lean and focused - its job is to enable the user experience (for an area of the view) and nothing more. When an app needs general *application logic* that would benefit more than one component, it's time to get a service involved.

Angular helps you follow these principles by making it easy to factor your application logic into services and make those services available to components through dependency injection.

Learning pathway

As noted above, a *service* is often used to perform data service operations. Learning this topic means that we will be learning much about a number of interrelated topics and techniques, including:

- Dependency injection
- HttpClient and asynchrony
- Observable (from RxJS)
- Input directive
- Data binding in component templates
- Structural directives, `*ngFor` and `*ngIf`
- Routing parameters
- Error handling

Supporting documentation

While the idea of a service can be understood and appreciated, it has a number of "moving parts" when properly and fully done.

As a result, there are several documentation sets that will help us get started, and then act as resources for richer or more complex scenarios.

In the official Angular.io documentation set, there are two main sources of information on services.

One is the **TUTORIAL > Services** area. To preview its contents:

- It continues with the *Tour of Heroes* example
- The basics are covered
- Data comes from within the app itself (not from outside)
- Introduces the notion of an asynchronous call to fetch data
- Does show how to add a second service to an app

In summary, the content is minimally useful to us. Its learning pathway is not as clear as we need.

A companion is the **TUTORIAL > HTTP** area. To preview its contents:

- Also continues with the *Tour of Heroes* example
- Comprehensive coverage, maybe too much
- Shows a simulated data server; an in-memory server
- Introduces error-handling
- And two-way data flow (updates back to the web service)

In summary, some of this content is useful. The info nuggets are too widely dispersed however.

Another source of information is the **FUNDAMENTALS > HttpClient** area. To preview its contents:

- Deep dive on HttpClient
- Covers many advanced topics

In summary, some of this content will be useful in the near future.

The community has some quality documentation too. A technology that comes from the community is *Observable*, which is part of the Reactive Extensions project (RxJS). Some links:

Reactive Extensions project home

Rangle.io Angular Training Book, Observables topic

Adding a service to an app

As you have lerned, use the Angular CLI to add a service. In the example below, a service named "DataModelManager" is added to the app:

```
ng g s DataManager --flat --S
```

As you have seen when creating components, a *PascalCase* name is transformed into lower case with "dash" (-) word separators, when it generates the source code files.

A new source code file is created, named `data-manager.service.ts`. Its contents:

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

@Injectable({
  providedIn: 'root'
})
export class DataManagerService {

  constructor() { }
}
```

The `@Injectable()` decorator indicates that this service is intended to be "injected" into another component or service at runtime. (We'll have more to say about "injection" soon.) The decorator's metadata object enables the new service to be available to *every* component in the app.

In the class code, we will add members: Properties to hold state information, and functions to perform tasks.

Configure, then use a service

The services example document will help you get started and successfully create, configure, and use a service, in detail. In this section, we present an overview.

The main configuration task to complete is to write one or more functions, in the service, that can be called by a component. The function will do something with data. Therefore, we also typically need properties to hold data.

Therefore, the simplest implementation of a service is to do the following two tasks:

1. Declare a private field to hold the data
2. Write a public function to deliver the data

Not included in the above is a task that would *materialize* the data. Data can be created new in the service, but more typically data will be fetched from a web service.

Use a service

A service can be used by *any component*. There are typically about four coding tasks to be done in the component class, and then another in its HTML template. The four tasks are:

- **Import statement**: As you would expect, we must `import` the service, to be able to use its members.

- **Constructor parameter**: We "inject" the service into the constructor, as a parameter. More about this in the next section.

- **Property to hold the data**: The main goal of our getting-started work is to work with data. Therefore, a component will need one or more properties to hold the results of a call to a web service resource.

- **Get the data**: In our examples, we will fetch the data when the component is loaded/initialized. Later (but soon), we'll learn how to fetch data as the result of user interaction.

Then, the HTML template coding task will require us to add/edit elements to display/render the data that was fetched.

Dependency injection

Above, you were introduced to the `@Injectable` decorator, which indicates that a service is intended to be "injected" into another component or service at runtime.

The Angular system is has dependency injection (DI) built in. It includes an "injector", which is a module that knows about and maintains a container of **service** instances that it has previously created. A service is created when it is accessed for the first time.

The idea behind dependency injection is simple: You have a component that depends on a service. In the component's code, you do not create that service yourself. Instead, you request one in the component's constructor (as a parameter - see: "Parameter Properties"), and the framework will provide you one. This leads to more decoupled code, which enables testability, and other great things.

During compilation, Angular looks at constructor types and their decorators. This is how services are identified and then maintained by the injector.

HttpClient for web service interaction

HttpClient is Angular's mechanism for communicating with a remote service over the HTTP protocol. While we could use the Fetch API, the Angular team recommends HttpClient.

From the official documentation:

> *The `HttpClient` in `@angular/common/http` offers a simplified client HTTP API for Angular applications that rests on the `XMLHttpRequest` interface exposed by browsers. Additional benefits of `HttpClient` include testability features, typed request and response objects, request and response interception, `Observable` apis, and streamlined error handling.*

The guidance is that a *service* is the usage home for HttpClient. Avoid using HttpClient in a component if it's possible that the request will be used in more than one component.

An HttpClient instance has several methods that you would expect. Most allow us to specify a return type variable (<T>). All will accept a `url` parameter. When using most of them, you will typically also add an `options` parameter:

- `get<T>()` - sends a GET request (and expects a specific return type)
- `post<T>()` - sends a POST request, add headers and the body to the options argument
- `put<T>()` - sends a PUT request (ditto above)
- `delete()` - sends a DELETE request
- …and there are others…

To make HttpClient available everywhere in the app:

1. Open the root AppModule for editing (`app.module.ts`),
2. Import the **HttpClientModule** symbol from @angular/common/http,
3. Add it to the **@NgModule.imports** array.

Using HttpClient

When using **HttpClient** anywhere else in your application (almost always in a `whatever.service.ts` file), be sure to *import **HttpClient*** (and *not* HttpClientModule) into that service or component. For example:

```
import { HttpClient } from "@angular/common/http";
```

The get() function

HttpClient includes a `get<T>()` function. Guess what it does?

This is what we will use in our getting-started examples. In its simplest usage, we do two things:

1. Specify the shape of the data that we're expecting
2. Specify the URL

The `get<T>()` function (overload number 15 in the documentation) returns an *Observable*, to be explained in detail soon. In essence, it is a stream of asynchronous data. The data could be a single object, or a collection. (That's determined by the web service resource.) Think of it as a *Promise*, but it better-serves the needs of Angular apps.

> *The return type is actually a generic `Observable`.*
> *The syntax is `Observable<T>`, where `T` is a placeholder for a type.*
> *Read/skim the generics documentation for more coverage.*
> *Often, it is an observable of an array of something.*
> *For example, an observable of an array of "product" or "customer" objects.*

For example, assume that the web service has a resource URL `/users` that will deliver a collection of user objects.

Next, assume that our data model manager service is responsible for the app's data. Its code will have a method that will call into the web service. For example, it may have a `getUsers()` method similar to this:

```
getUsers(): Observable<User[]> {
  return this.http.get<User[]>(`${this.url}/users`)
}
```

Notice that the return type of the method is `Observable<User[]>`. You can read this return type as an "observable of an array of User objects".

In any component class that is using the data manager service, we extract the data from the `Observable` by using the "subscribe" method. For example:

```
// Assume we have a local "this.users" property already defined,
// and "this.m" is a reference to the data modelmanager service
this.m.getUsers().subscribe(result => this.users = result);
```

The result is that the stream of data - a collection of users, which comes in asynchronously - is transformed into a more familiar array object that we can immediately work with.

Additional note… If you must do multiple tasks in the function that's passed to the `.subscribe()` method, then write the function as a multi-line arrow function; for example:

```
// Assume we have a local "this.users" property already defined,
// and "this.m" is a reference to the data modelmanager service
this.m.getUsers().subscribe(result => {
  // Save the result
  this.users = result;
  // Do some other task (call a function)
  doOtherTask();
  // etc.
});
```

Observable (from RxJS)

Reactive E**x**tensions for **J**ava**S**cript (RxJS) is a library that comes bundled with the Angular development toolchain.

From the documentation:

> *RxJS is a library for reactive programming using Observables, to make it easier to compose asynchronous or callback-based code.*

Above, we suggested that you can think of an `Observable` as a kind of *Promise* that better serves the needs of Angular apps. That's a good starting point.

Then, add on the idea of a *sequence*. For example, let's assume that (like above) we are fetching a collection of `User` objects from a web service. A *Promise* will return that collection. An *Observable* will do that too, inside of the *Observable* wrapper or construct. One small advantage of this, which we will take advantage of later on, is that we can do things to each item (e.g. inspect, transform, etc.) in the sequence as it comes in (or gets delivered).

While it is possible to do a deep dive on `Observable`, we will not need to do so in this course.

Next actions

Open and study the getting started example document.