NGMODULES

Angular apps are modular and Angular has its own modularity system called NgModules

As our code grows, it makes sense to be able to split code into their own packages. Angular has already made a couple of Modules for us, but we can also create our own.

Every Angular app has at least one NgModule class, the root module. But we can create new modules, called *feature modules*.

An NgModule is defined by a class decorated with @NgModule().

The decorator takes multiple properties:

- declarations. Components and directives belonging to this module.
- exports. Declarations that should be visible to other modules.
- imports. Other modules to load.
- providers. Services to load.
- bootstrap. Main application view, the root component. Only the root NgModule should have this.

```
@NgModule({
  declarations: [
      DashboardComponent,
      WidgetComponent
  ],
  providers: [DashboardService],
  imports: [
  CommonModule,
  HttpClientModule
  ],
  exports: [
    DashboardComponent
  ]
})
```

Creating a new feature module is simple. Just add a new folder and create a file called modulename.module.ts.

```
import { NgModule } from '@angular/core';
import { CommonModule } from '@angular/common';
@NgModule({
  declarations: [
  ],
  providers: [],
  imports: [CommonModule],
  exports: []
})
export class DashboardModule { }
```

By default all modules shoild import the **CommonModule**.

To load the new Module, just add it to your imports in another module.

```
#app.module.ts
Only the root NgModule
@NgModule({
  declarations: [],
  imports: [
      BrowserModule,
      DashboardModule
    ]
})
```

BUILT IN MODULES

Apart from creating our own modules, angular has already created a couple that solves common problems.

HttpClientModule allows us to inject a HTTP-service to make async requests. First, add HttpClientModule to your imports in app.module. Then inject HttpClient in a service or component.

```
constructor(private http: HttpClient) {
  this.http.get('/api/users').subscribe((data) => {
  });
}
```

FormsModule provides us with helper utils for handling form input.

OBSERVABLES & RXJS

Observables provide support for passing messages between publishers and subscribers in your application They come bundled with a library called **RxJS** which Angular uses extensively.

Observables are similar to Promises in the sense that they are async. They differ in a few points,

- Observables can have multiple listeners
- Observables can emit data multiple times

We can wrap promises into observables

```
import { from } from 'rxjs';

// Create an Observable out of a promise
const data = from(fetch('/api/endpoint'));

data.subscribe({
  next(response) { console.log(response); },
  error(err) { console.error('Error: ' + err); },
  complete() { console.log('Completed'); }
});
```

We have already used observables when we created an **EventEmitter** in our @Output. In Angular, EventEmitters are a special type of observables.

We can use the **next** method on a observable to emit data to all listeners

```
observable.subscribe((data) => {
    console.log(data); // 100
});
observable.next(100);
```

Another common Observable is **BehaviorSubject**. They can emit data using **next** and get the last value using **getValue**

```
import { BehaviorSubject } from 'rxjs';

const subject = new BehaviorSubject();
subject.subscribe((data) => {
    console.log(data);
});
subject.next(100); // Logs 100
subject.next(200); // Logs 200
subject.getValue() // Logs 200
```

Combining Services and Observables

We can use Observables to create emitters from our services, allowing our components to listen to changes in our data structure.

Lets imagine a EventService

```
# EventService
class EventService {
    events: BehaviorSubject<number> = new BehaviorSubject<numb
    constructor() {
        setInterval(() => {
            this.events.next(Math.random())
        }, 1000);
    }
    getEventStream() {
        return this.events;
    }
}
```

Now we can subscribe for new events in a component

```
# EventComponent
class EventComponent implements OnInit {
    lastEvent = null;
    constructor(private eventService: EventService) {
        this.eventService.getEventStream().subscribe(
        (event) => {
            this.lastEvent = event;
        }

    }
}
```

This allows us to build reactive applications, where we update our components state based on changed in our data layer

MORE ABOUT COMPONENT DESIGN

When designing applications with complex structure it can be useful to split the responsibility of components. This presents a couple of questions:

- what types of components are there?
- how should components interact?

We often split components into two types.

- **Container Components**, also known as **Smart** Components
- **Presentation Components**, also known as **Dumb** Components

A container or smart component is concerned with how things should work. It inject services, fetches data and passes it to its host of dumb components.

A dumb component’s task is to display data. It does not inject services. It only contains logic needed for displaying items.

A dumb component only communicates with its parent component, by using @Input and @Output properties.

Designing applications by combining smart and dumb components with modules and services gives us a easy, scalable structure.