We live in a microservices world, and this world is here to stay. Back-end developers need to dive into [Domain-Driven Design](https://en.wikipedia.org/wiki/Domain-driven_design), write stateless, resilient, highly available services, keep data synchronized through [Change Data Capture](https://en.wikipedia.org/wiki/Change_data_capture), handle network failure, deal with authentication, authorization, [JWT](https://jwt.io/)... and expose a beautiful [Hypermedia API](https://smartbear.com/learn/api-design/what-is-hypermedia/) so the front-end developers can add a great user interface to it.

Good! But what about the front-end?

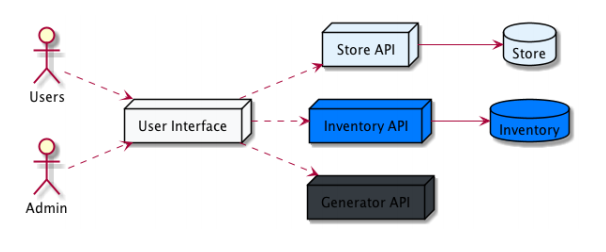
Let's say we have several microservices, several teams, several APIs and, therefore, several user interfaces. How can you create an integrated and consistent user interface so your users don't actually see that there are as many user interfaces as there are Microservices? In this post, I'm implementing the [client-side UI composition design pattern](https://microservices.io/patterns/ui/client-side-ui-composition.html) using Angular Libraries.

**A Microservices Architecture**

Let's say we have a CD BookStore application that allows customers to buy CDs and books online, as well as administrators to check the inventory. We also have a complex ISBN number generator to deal with. If we model this application into microservices, we might end up splitting it into 3 separate microservices:

* Store: Deals with displaying information on CDs and books, allows users to buy them.
* Inventory: Allows administrators to check the availability of an item in the warehouse and buy new items if needed.
* Generator: Generates ISBN numbers each time a new book is created.

We end up with two roles (user and admin) interacting with these 3 APIs through a single user interface:

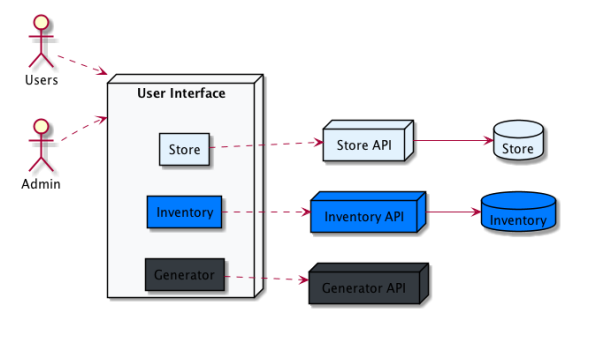


*Client-side UI composition.*

In a perfect world, these three microservices are independent, developed by three different teams and, therefore, have three different user interfaces, each released at its own pace. On one hand, we have three different UIs, and on the other, we want our users to navigate into what they see as a single and integrated application. There are several ways of doing it, and one way I'm describing here is called the Client-side UI composition design pattern:

**Problem:** How to implement a UI screen or page that displays data from multiple services?

**Solution:** Each team develops a client-side UI component, such an Angular component, that implements the region of the page/screen for their service. A UI team is responsible for implementing the page skeletons that build pages/screens by composing multiple, service-specific UI components. The idea is to aggregate, on the client side, our three user interfaces into a single one and end up with something that looks like this:



Aggregating several UIs into a single one works better if they use compatible technologies, of course. In this example, I am using Angular and only Angular to create the 3 user-interfaces and the page skeleton. Of course, [Web Components](https://www.webcomponents.org/) would be a perfect fit for this use case, but that's not the purpose of this article.

**Angular Libraries**

One novelty with Angular CLI 6 is the ability to easily [create libraries](https://github.com/angular/angular-cli/wiki/stories-create-library). Coming back to our architecture, we could say the page skeleton is the Angular application (the CDBookStore application), and then, each microservice user interface is a library (Store, Inventory, Generator).

In terms of Angular CLI commands, this is what we have:

1

# Main app called CDBookStore

2

$ ng new cdbookstore --directory cdbookstore

3

/

4

-routing

5

true

6

# The 3 libraries for our 3 microservices UI

7

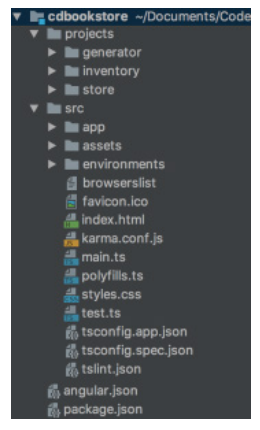
$ ng generate library store --prefix str

8

$ ng generate library inventory --prefix inv

9

$ ng generate library generator --prefix gen

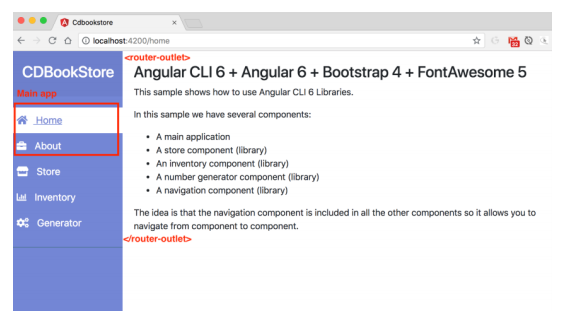


Once you've executed these commands, you will get the following directory structure:

* src/app/ is the good old Angular structure. In this directory, we will have the page skeleton of the CDBookStore application. As you will see, this skeleton page is only a sidebar to navigate from one Microservice UI to another.
* projects/ is the directory where all the libraries end up
* projects/generator: the UI for the Generator microservice.
* projects/inventory: the UI for the Inventory microservice.
* projects/store: the UI for the Store microservice.

**The Skeleton Page**

The skeleton page is there to aggregate all the other components. Basically, it's only a sidebar menu (allowing us to navigate from one Microservice UI to another) and a <router-outlet>. It is where you could have login/logout and other user preferences. It looks like this:



In terms of code, despite using [Bootstrap](https://getbootstrap.com/), there is nothing special. What's interesting is the way the routes are defined. The [AppRouting Module](https://github.com/agoncal/agoncal-sample-angular/blob/master/cdbookstore/src/app/app-routing.module.ts" \t "_blank) only defines the routes of the main application (here, the Home and About menus).

1

const routes: Routes = [

2

{path: '', component: HomeComponent},

3

{path: 'home', component: HomeComponent},

4

{path: 'about', component: AboutComponent},

5

];

6

@NgModule({

7

imports: [RouterModule.forRoot(routes)],

8

exports: [RouterModule]

9

})

10

export class AppRoutingModule { }

In the HTML side of the side menu bar, we can find the navigation directive and the router:

1

<ul class="list-unstyled components">

2

<li>

3

<a [routerLink]="['/home']">

4

<i class="fas fa-home"></i>

5

Home

6

</a>

7

<a [routerLink]="['/str']">

8

<i class="fas fa-store"></i>

9

Store

10

</a>

11

<a [routerLink]="['/inv']">

12

<i class="fas fa-chart-bar"></i>

13

Inventory

14

</a>

15

<a [routerLink]="['/gen']">

16

<i class="fas fa-cogs"></i>

17

Generator

18

</a>

19

</li>

20

</ul>

21

<!-- Page Content -->

22

<div id="content">

23

<router-outlet></router-outlet>

24

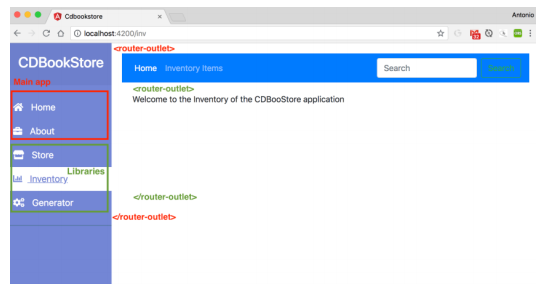
</div>

As you can see in the code above, the routerLink directive is used to navigate to the main app components as well as library components. The trick here is that /home is defined in the routing module, but not /str, /inv or /gen. These routes are defined in the libraries themselves.

**Disclaimer:**I am a terrible web designer/developer. If someone reading this post knows about jQuery and Angular and want to give me a hand, I would love to have the sidebar be collapsible. I even [created an issue](https://github.com/agoncal/agoncal-sample-angular/issues/1) for you!

**The Libraries**

Now when we click on Store, Inventory, or Generator, we want the router to navigate to the library's components:



Notice the parent/child relationship between routers. In the image above, the red router-outlet is the parent and belongs to the main app. The green router-outlet is the child and belongs to the library. Notice that in the store-routing.module.ts, we use str as the main path, and all the other paths are children (using the children keyword):

1

const routes: Routes = [

2

{

3

path: 'str', component: StoreComponent,

4

children:

5

[

6

{path: '', component: HomeComponent},

7

{path: 'home', component: HomeComponent},

8

{path: 'book-list', component:

9

BookListComponent},

10

{path: 'book-detail', component:

11

BookDetailComponent},

12

]

13

},

14

];

15

@NgModule({

16

imports: [RouterModule.forRoot(routes)],

17

exports: [RouterModule]

18

})

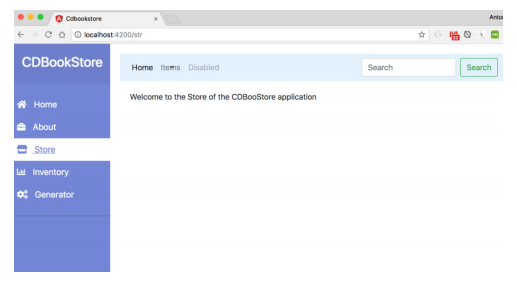
19

export class StoreRoutingModule {

20

}

The beauty of having a parent/child router relationship is that you can benefit from "feature" navigation. This means that if you navigate to /home or /about, you stay in the main application, but if you navigate to /str, /str/home, /str/booklist or /str/book-detail, you navigate to the Store library. You can even do lazy loading on a per feature based (only load the Store library if needed).



The Store library has a navigation bar at the top, therefore, it needs the routerLink directive. Notice that we only need to use the child path [routerLink]="['book-list']" without having to specify /str (no need to [routerLink]="['/str/book-list']"):

1

<nav>

2

<div class="collapse navbar-collapse">

3

<ul class="navbar-nav mr-auto">

4

<li class="nav-item dropdown">

5

<a class="nav-link dropdown-toggle">Items</a>

6

<div class="dropdown-menu" aria-labelledby=

7

"navbarDropdownMenuLink">

8

<a class="dropdown-item"

9

[routerLink]="['book-list']">Books</a>

10

<a class="dropdown-item"

11

[routerLink]="['cd-list']">CDs</a>

12

<a class="dropdown-item"

13

[routerLink]="['dvd-list']">DVDs</a>

14

</div>

15

</li>

16

</ul>

17

</div>

18

</nav>

19

<!-- Page Content -->

20

<div id="content">

21

<router-outlet></router-outlet>

22

</div>

**Pros and Cons**

The [Client-side UI composition design pattern](https://microservices.io/patterns/ui/client-side-ui-composition.html) has some pros and cons. If you already have existing user-interfaces per microservices, and if each one is using totally different technologies (Angular vs React vs Vue.JS vs ...) or different framework versions (Bootstrap 2 vs Bootstrap 4), then aggregating them might be tricky and you might have to rewrite bits and pieces to have them compatible.

But this can actually be beneficial. If you don't have extended teams with hundreds of developers, you might end up being the one moving from one team to another one and will be happy to find (more or less) the same technologies. This is defined in the [Chassis pattern](https://microservices.io/patterns/microservice-chassis.html). And for your end users, the application will have the same look and feel (on mobile phones, laptops, desktops, tablets), which gives a sense of integration.

With the way Angular Libraries are structured, having a monorepo is much easier. Of course, each library can have its own life cycle and be independent, but at the end of the day, it makes it easier to have them on the same repo. I don't know if you [love monorepos](https://medium.com/@Jakeherringbone/you-too-can-love-the-monorepo-d95d1d6fcebe) or not, but some do.

**Conclusion**

We've been architecting and developing microservices for several decades now (yes, it used to be called distributed systems, or distributed services), so we roughly know how to make them work on the back-end. Now, the challenge is to be able to have a user interface which communicates with several microservices, developed by different teams, and at the same time feels consistent and integrated for the end-user. You might not always have this need (see how Amazon has totally different UIs for its services), but if you do, then the [Client-side UI composition design pattern](https://microservices.io/patterns/ui/client-side-ui-composition.html) is a good alternative.