ANGULAR ARCHITECTURE

Angular Architecture - Smart Components vs Presentational Components

ANGULAR UNIVERSITY
11 JAN 2021

This post is part of the ongoing Angular Architecture series, where we cover common design problems and solutions at the level of the View Layer and the Service layer. Here is the full series:

- View Layer Architecture - Smart Components vs Presentational Components
- View Layer Architecture - Container vs Presentational Components Common Pitfalls

- Service Layer Architecture - Redux and Ngrx Store - When to Use a Store And Why?
- Service Layer Architecture - Ngrx Store - An Architecture Guide

Introduction to View Layer Architecture

When building an Angular application, one the most frequent questions that we are faced with right at the beginning is: how do we structure our application?

The immediate answer that we might find is: we just split everything into components! But we then quickly find out that there is more to it than that:

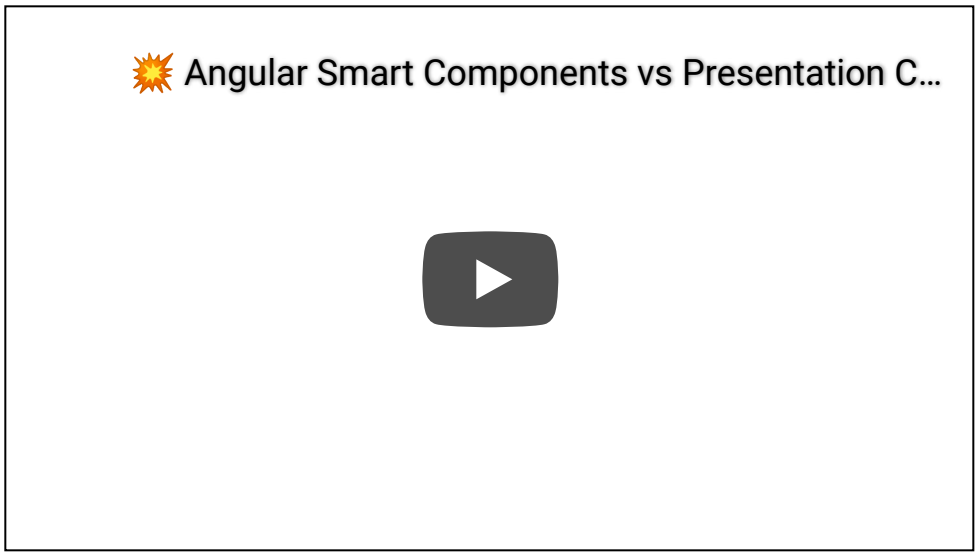
- what types of components are there?
- how should components interact?
- should I inject services into any component?
- how do I make my components reusable across views?

We will try to answer these and other questions by splitting components essentially into two types of specialized components (but there is more to it):

- Smart Components: also know sometimes as application-level components, or container components
- Presentation Components: also known sometimes as pure components or dumb components

Let's find out the differences between these two types of components, and when should we use each and why!

describe below (Subscribe in YouTube to get similar videos):



Splitting an application into different types of components

In order to understand the difference between the two types of components, let's start with a simple application, where the separation is not yet present.

We started building the Home screen of an application and we have added multiple features to a single template:

```
@Component({
  selector: 'app-home',
  template: `
    <h2>All Lessons</h2>
    <h4>Total Lessons: {{lessons?.length}}</h4>

    <div class="lessons-list-container v-h-center-block-parent">
```

```
11                  <td class="lesson-title"> {{lesson.description}} </td>
12                  <td class="duration">
13                      <i class="md-icon duration-icon">access_time</i>
14                      <span>{{lesson.duration}}</span>
15                  </td>
16              </tr>
17              </tbody>
18          </table>
19      </div>
20  `,
21    styleUrls: ['./home.component.css']
22  })
23  export class HomeComponent implements OnInit {
24
25      lessons: Lesson[];
26
27    constructor(private lessonsService: LessonsService) {
28    }
29
30    ngOnInit() {
31        this.lessonsService.findAllLessons()
32            .pipe(
33                tap(console.log)
34             )
35            .subscribe(
36                lessons => this.allLessons = lessons
37            );
38    }
39
40    selectLesson(lesson) {
41      ...
42    }
43
44  }
```

01-home.ts hosted with ❤ by GitHub view raw

Understanding the Problem

table containing a list of lessons.

But there will likely be other parts of the application where this functionality is also needed, for example let's say that we have another screen which presents the table of contents of a given course.

In that screen we also want to display a list of lessons, but only the lessons that belong to that course. In that case, we would need something very similar to what we have implemented in the Home screen.

And we shouldn't just copy-paste this across components, we should create a reusable component, right?

Let's create a Presentation Component

What we will want to do in this situation is to extract the table part of the screen into a separate component, let's call it the `LessonsListComponent`:

```
@Component({
  selector: 'lessons-list',
  template: `
      <table class="table lessons-list card card-strong">
          <tbody>
          <tr *ngFor="let lesson of lessons" (click)="selectLesson(lesson)">
              <td class="lesson-title"> {{lesson.description}} </td>
              <td class="duration">
                  <i class="md-icon duration-icon">access_time</i>
                  <span>{{lesson.duration}}</span>
              </td>
          </tr>
          </tbody>
      </table>
  `,
  styleUrls: ['./lessons-list.component.css']
})
export class LessonsListComponent {

  @Input()
  lessons: Lesson[];

  @Output('lesson')
  lessonEmitter = new EventEmitter<Lesson>();

    selectLesson(lesson:Lesson) {
        this.lessonEmitter.emit(lesson);
    }

}
```

02-lessons-list.ts hosted with ❤ by GitHub view raw

Now let's take a closer look at this component: it does not have the lessons service injected into it via its constructor. Instead, it receives the

come from:

- the lessons might be a list of all lessons available
- or the lessons might be a list of all the lessons of a given course
- or even the lessons might be a page in any given list of a search

We could reuse this component in all of these scenarios, because the lessons list component does not know where the data comes from. The responsibility of the component is purely to present the data to the user and not to fetch it from a particular location.

This is why we usually call to this type of component a Presentation Component. But what happened to the Home Component?

Let's create a Smart Component

If we go back to the Home component, this is what it will look like after refactoring:

```
4
5   @Component({
6     selector: 'app-home',
7     template: `
8         <h2>All Lessons</h2>
9         <h4>Total Lessons: {{lessons?.length}}</h4>
10
11        <div class="lessons-list-container v-h-center-block-parent">
12            <lessons-list [lessons]="lessons" (lesson)="selectLesson($event)">
13        </div>
14  `,
15    styleUrls: ['./home.component.css']
16  })
17  export class HomeComponent implements OnInit {
18
19      lessons: Lesson[];
20
21    constructor(private lessonsService: LessonsService) {
22    }
23
24    ngOnInit() {
25       ...
26    }
27
28    selectLesson(lesson) {
29      ...
30    }
31
32  }
```

03-home-component-refactored.ts hosted with ❤ by GitHub view raw

how to retrieve the lessons list from a service, and what type of list this is (if these lessons are the lessons of a course, etc.).

But the Home component does not know how to *present* the lessons to the user.

What type of component is the Home component?

Let's give a name to this type of components similar to the home component, which is an application-specific component: let's call this a Smart Component.

This type of component is inherently tied to the application itself, so as we can see it receives in the constructor some dependencies that are application-specific, like the `LessonsService`.

> *It would be very hard to use this component in another application.*

The top level component of our view will likely always be a smart component. Even if we transfer the loading of the data to a router data resolver, the component would still at least have to have the `ActivatedRoute` service injected into it.

So we want to implement the top-level smart component by composing it internally using a set of presentation components. And it's as simple as that. Or is it?

Typical interaction between smart components and presentation components

The example that we see here is very frequent, we have the smart

utput .

In this case, we are using the custom `lesson` event to indicate that we have selected a given lesson in the list.

Using `@Output` the presentation component remains isolated from the smart component via a clearly defined interface:

- the lessons list presentation component only knows that it emitted an event, but does not know what are the receivers of the event or what will the receivers do in response to the event
- the home screen smart component subscribes to the `lesson` custom event and reacts to the event, but it does not know what triggered the event. Did the user double click on the lessons list or did the user click a view button? This is transparent to the smart component.

So this is all clear and simple, what could go wrong here?

A clear way to split smart vs presentation components?

With this, we might at this point conclude that building our application is as simple as making all top level components smart components, and build them by using a local tree of presentation components.

But the thing is, it's sometimes not that simple because custom events like `lesson` don't bubble up. So if for example you have a deep tree of components and you want a component multiple levels above to know about the event, the event will not bubble.

custom events don't bubble?

Let's say that instead of only one level of nesting between the lesson list and the home component, we have several levels: the lessons list is inside a collapsible panel which is inside a tab panel.

The lesson list still wants to notify the home component that a lesson was selected via the `lesson` event. But the two components in the middle `TabPanel` and `CollapsiblePanel` are non application-specifc Presentation Components.

> *imagine they were components of the Angular Material library!*

These Presentation-only components do not know about the `lesson` event, so they cannot bubble it up. So how do we implement this, and also why can't custom events simply bubble up?

Why don't custom event bubble up, like it's the case with DOM events like click?

This is not an accident, it's by design and probably to avoid event soup scenarios that the use of solutions similar to a service bus like in AngularJs `$scope.$emit()` and `$scope.$broadcast()` tend to accidentally create.

These type of mechanisms tend to end up creating tight dependencies between different places of the application that should not be aware of each other, also events end up being triggered multiple times or in a sequence that is not apparent while just looking at one file.

So the custom event of a presentation component will only be visible by its parent component and not further up the tree.

the times this is not what we want to implement.

So how do we solve the situation of the lessons list inside the collapsible panel inside the tab panel scenario?

We should still likely create a presentation component for the lessons list. The functionality to present the lessons can be isolated, so the version of `LessonsListComponent` still applies, it's just something that will be used everywhere in the application. But how can this list notify the home component?

For that there are several solutions. One solution that we should look into especially if building a large scale application is to look into solutions like `ngrx/store`.

But even with a store solution, we might not want to inject the store in the presentation component. Because the result of selecting a lesson might not be always to dispatch an event to the store.

To keep the example simple, let's start by creating a specialized store-like service to solve only this lesson selection problem:

```
4
5       private _selected: BehaviorSubject<Lesson> = new BehaviorSubject(null);
6
7       public selected$ = this._selected.asObservable().pipe(filter(lesson =>
8
9
10      select(lesson:Lesson) {
11           this._selected.next(lesson);
12      }
13
14  }
```

04-lesson-selected-service.ts hosted with ❤ by GitHub view raw

If we expose the subject we would give any other part of the application the ability to emit an event on behalf of the service, and that is to be avoided.

So how can this service be used, because we can't inject it in the `LessonsListComponent`, right? We will get to that part, right now let's see first how we can use this new service in the Home component.

Using the new service in Home Component

What the Home component will do is, it will have the new component injected in its constructor:

```
4     lessons: Lesson[];
5
6   constructor(
7         private lessonsService: LessonsService,
8         private lessonSelectedService: LessonSelectedService) {
9   }
10
11  ngOnInit() {
12      ....
13      this.lessonSelectedService.selected$.subscribe(lesson => this.selectLe
14  }
15
16  selectLesson(lesson) {
17    ...
18  }
19
20 }
```

05-use-service-in-home.ts hosted with ❤ by GitHub view raw

As we can see, we have subscribed to the selected$ Observable, which emits new lessons and we trigger the specific logic of the Component to process the selection.

But notice that the Home component does not know about the lessons list, it just knows that some other part of the application triggered a lesson selection. The two parts of the application are still isolated:

- the emitter of the selection does not know about the Home component
- the home component does not know about the lesson

So we have fixed the problem, right? Not yet, because we still don't want to inject the new service in the `LessonListComponent`, this would make it a smart component and we want to keep it a Presentation component. So how to solve this?

How to keep the LessonsListComponent a Presentation Component?

Actually, one way to solve the problem it to make it a Smart Component ;-) We might get to the conclusion that anywhere on the application that this table exists, we always want to trigger a call to the `LessonSelectedService`.

This would make the lessons list component an application specific component which it probably already was anyway. We would probably not ship this component and use it in several applications for example.

So this would solve the problem, and this means that a top level application component like the Home component might be composed of a tree of components that are not only Presentation components.

A Smart Component is not only a top-level component

A Smart Component does not have to be a top level router component only. We can see that there could be other components further down the tree that also get injected a service like `LessonSelectedService`, and don't necessarily get their data only from `@Input()`.

Another solution for keeping the LessonsListComponent a Presentation

smart component, which gets injected the `LessonSelectedService`:

```
@Component({
  selector: 'custom-lessons-list',
  template: `
       <lessons-list [lessons]="lessons" (lesson)="selectLesson($event)"></
  `
})
export class CustomLessonsListComponent {

   constructor(private lessonSelectedService: LessonSelectedService) {

   }

   selectLesson(lesson) {
       this.lessonSelectedService.select(lesson);
   }

}
```

06-wrap-presentation-component.ts hosted with ❤ by GitHub view raw

In this we have created a wrapper smart component, and called it `CustomLessonsListComponent`. In this case, we wrapped our own presentation component, but we could be wrapping a component that came from a third party library as well.

Imagine a `MyCustomCountrySelectDropdown`, that wraps a generic dropdown and injects it with the data coming from a concrete service.

How to decide what components to build?

So how to split the application up in many components? Should the header of a page be a component, even if it's used only once?

Organization and readability are alone reasons to create a component even if it's only used in one place. Separating things in smaller files helps to keep to code base maintainable and with the Angular CLI there is no overhead in creating a new component: with a single command, we have a working component where to paste the header in seconds.

How to approach component design

One way to approach this is to avoid defining from the start what will be a component and of what type: we can start by building the top-level component using only plain HTML and third party components.

And only when the template starts getting bigger we start splitting it up into components. If something is used in several parts of the screen and always triggers a given action like calling a store dispatch, we might consider refactoring to a smaller smart component.

If later we realize that we need to present the same data like the smart component that we just created, we can extract the presentation part from it into a presentation component.

The best approach to get a great set of well-designed components is by continuous refactoring, which can be done without burden by using the Angular CLI.

Summary

Angular components can be roughly categorized into two different types:

from. They receive the data via an `@Input()`, and return modified data through `Output()` events

- Smart components: these components are responsible for interacting with the service layer and retrieving the data, which then they pass to presentational components

While building an application, we can look out for opportunities to extract the pure presentation logic into Presentation Components: these only use `@Input` and `Output`, and are useful in case we need to isolate the presentation logic and reuse it.

Communication between smart components at different levels in the component tree or even siblings can be done using a shared service or a store if we want to keep the two components decoupled and unaware of each other.

But we might also want to inject components completely into each other and create a tight coupling, sometimes that is the best solution. In that case injecting the components into each other via for example `@ViewChild` might be the best approach.

Smart vs Presentation Components is a useful distinction

In general the distinction between smart vs presentation components is very useful to keep in mind, but it might not be applied to all the components of the application.

We could have a small component that is both aware of a service and presents some data deeper in the tree, like a lessons list that calls a store upon lesson selection.

Splitting this component up further into a smart and a presentation

Smart vs Presentation Components is more of a mindset to adopt where we ask ourselves:

- would this presentation logic be useful elsewhere in the application?
- would it be useful to split things up further?
- are we creating unintended tight couplings in the application?

We don't necessarily need to extract all the rendering logic of every component we build into a separate presentation component. It's more about building the components that make the most sense to our application at any given time, and refactor if needed in a continuous iterative process made simple by the CLI.

I hope that you enjoyed the post and that it helped getting started with view layer design. Make sure to see the other posts on the Architecture series, linked above!

I invite you to subscribe to our newsletter to get notified when more posts like this come out:

And if you are looking to learn more about Angular application design

If you are just getting started learning Angular, have a look at the Angular for Beginners Course:

Other posts on Angular

If you enjoyed this post, have also a look also at other popular posts that you might find interesting:

- Angular Router - How To Build a Navigation Menu with Bootstrap 4 and Nested Routes

- How to build Angular apps using Observable Data Services - Pitfalls to avoid
- Introduction to Angular Forms - Template Driven vs Model Driven
- Angular ngFor - Learn all Features including trackBy, why is it not only for Arrays ?
- Angular Universal In Practice - How to build SEO Friendly Single Page Apps with Angular
- How does Angular Change Detection Really Work ?
- Typescript 2 Type Definitions Crash Course - Types and Npm, how are they linked ? @types, Compiler Opt-In Types: When To Use Each and Why ?

Name

jmls • 3 years ago

I'm a newbie to the whole rxjs thing - I'm curious about the line

this.lessonSelectedService.selected$.subscribe(lesson => this.selectLesson(lesson));

won't this lead to a memory leak as you are not unsubscribing in ngDestroy ?

3 | • Reply • Share ›

Charles Robertson jmls • 2 months ago

You are correct. The subscription does need to be destroyed. I think the author would have added this, if this had been the focus of the tutorial. Obviously, in this example, you would need to explicitly declare a subscription:

this.subscription = this.lessSelectedService…

So, that you can destroy it properly within **ngDestroy**

And there are better ways of destroying subscriptions, as well, like using RxJS operators like **takeUntil**

| • Reply • Share ›

Maria Maria • 6 months ago

What if we need to display the same table with lessons on another (not home) page? Then we will copy paste all code (including injecting services) from Home component to that another page component? Doesn't sound good..

1 | • Reply • Share ›

Peter Grady • a year ago • edited

This is a very nice article - one thing the article points out - which tripped me up some.. Is that smart vs. presentational can really still get you in trouble.

What happened to me is that I ended up with a very large complex smart component sitting doing a lot of work sending various bit of data to many presentation components. Even if you try to use services to abstract away some of the logic - that smart component still has a very large 'interface' of sorts which

It's just doing a lot of stuff - or calling a service which is doing a ton of different things. It's handling click events - its getting data from an external API etc etc. It gets crazy.

The approach I am using to fix this is container type component (associated with a router outlet) and in that we have several smart components - each of them wrapping a few (often one) dumb components. This is in combination with a data store. So the smart components use the data store so this way each component can do a lot less stuff.

Basically you want to keep your smart components small and use more of them - at least in my use case. Presentation components add to code reusability - but don't really help much with code complexity and the single purpose idea.

1 ^ | v • Reply • Share ›

Void • 3 years ago • edited

How I sloved the issue of deeply nested presentation components that need to emit a selection event in one of my projects if I remember correctly was to not only have an input for data but another input with a BehaviorSubject called selectEvent.
The smart component (parent) makes the BehaviorSubject subscribes to it and sends that to the child in its input (the BSubject object). The child then only has to do selectEvent.next(someValue) and your parend will get notified trough the subscribe ! Not to mention you can also use getValue() on the BSubject outside the subscribe if you need the info for whatever reason (post, click on a button that does stuff to selection...). Or you could also subscribe at the child side and use it to notify the child of certain events by doing a next() in the parent this time for stuff like unselect current selection maybe ? Thus creating your personal two way communications channel.
This way there is no need to clutter your project with a service just to share data or do some weird js hacks. ^^
PS: if I consider what the article said, my solution actually doesn't make the presentation component into a smart component and as such is a more "technical" solution to a problem related to angulars principle of not propagating events.

1 ^ | v 1 • Reply • Share ›

Jayaram Kurapati • 2 years ago

For example, I will pass create event from presentation component to container component, then container component will hit an http request, now if any error occurs ? how I should pass to presentation component ? through another @Input property or any other goog practice exists ?

Thanks.

Reply • Share ›

Abhishek Prakash • 2 years ago

Thanks for article. I've tried to explain about Smart components and Dumb components in pretty easy way. In case if someone might be interested.

https://www.shade.codes/dumb-components-and-smart-components/

Reply • Share ›

Eugene Vedensky • 3 years ago

What is the value of the filter call? Won't just return every selected lesson assuming a valid lesson is selected by the client component?

Reply • Share ›

Max Tomasello • 3 years ago

MORE IN ANGULAR ARCHITECTURE

Angular Architecture - Container vs Presentational Components Common Design Pitfalls
13 Apr 2017 – 11 min read

Angular Service Layers: Redux, RxJs and Ngrx Store - When to Use a Store And Why?
22 Nov 2016 – 19 min read

Angular Application Architecture - Building Flux Apps with Redux and Immutable.js
6 Dec 2015 – 11 min read

See all 3 posts →

Angular Service Layers: Redux, RxJS and Ngrx Store - When to Use a Store And Why?

This post is part of the ongoing Angular Architecture series, where we cover common design problems and solutions at the level of the View Layer and the Service layer. Here is the full series: View Layer Architecture - Smart Components vs Presentational Components View

ANGULAR UNIVERSITY
11 JAN 2021 • 19 MIN READ

Top Angular Courses

If you are trying to learn Angular as a beginner, or if you are already experienced and want to take your knowledge to the next level, you probably have already asked yourself what is the best way to learn Angular. We have been teaching

ANGULAR UNIVERSITY
1 JUL 2021 • 15 MIN READ