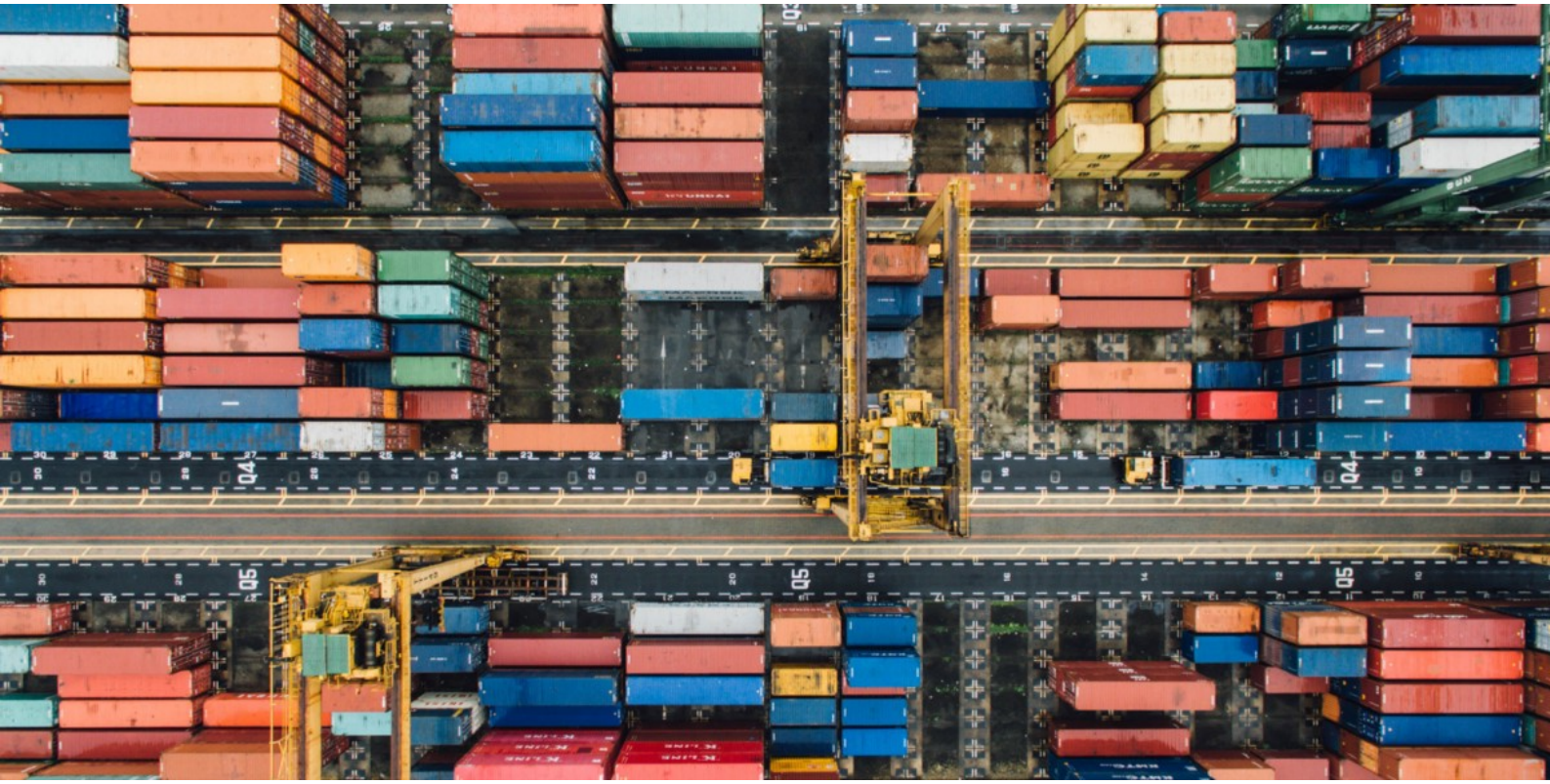
Container Components with Angular

Lars Gyrup Brink Nielsen Follow

Nov 6, 2018 · 14 min read



Standardised shipping containers. Photo by chuttersnap on Unsplash.

With the Model-View-Presenter design pattern it is easy to use any application state management library or pattern whether its a redux-like state container like the NgRx Store or simply plain old services as in the “Tour of Heroes” Angular tutorial.

Container components sit at the boundary of the presentational layer and integrate our UI with the application state. They serve two main purposes:

- Container components supply a data flow for presentation.
- Container components translate component-specific events to application state commands or *actions* to put it in Redux/NgRx Store terms.

Container components can also integrate UI to other non-presentational layers like I/O or messaging.

In this article we will go through the process of extracting a container component from a mixed component.

• • •

Most of the terms used in this article are explained in the introductory article “Model-View-Presenter with Angular”:

• • •

Container Components

We call them *container components* because they *contain* all the state needed for the child components in their view. Additionally, they exclusively *contain* child components in their view—no presentational content. The template of a container component is made up entirely of child components and data bindings.

Another useful way to think of container components is that they—like *shipping containers*—are entirely self-contained and can be moved arbitrarily around in component templates since they have no input or output properties.

Container components address the issue of bucket brigading events and properties through several layers of the component tree—a phenomenon known as *prop drilling* in the React community.

Simple Example

We start out with the `DashboardComponent` from the Tour of Heroes tutorial.

```
import { Component, OnInit } from '@angular/core';

import { Hero } from '../hero';
import { HeroService } from '../hero.service';

@Component({
  selector: 'app-dashboard',
  styleUrls: ['./dashboard.component.css'],
  templateUrl: './dashboard.component.html',
})
export class DashboardComponent implements OnInit {
  heroes: Hero[] = [];

  constructor(private heroService: HeroService) {}

  ngOnInit() {
```

Dashboard: Mixed component model. Open in new tab.

Identify Mixed Concerns

We see that this component has mixed concerns that span multiple horizontal layers in our app as described in the introductory article.

Search this file...

Horizontal layer	Examples
Business logic	Application-specific logic, domain logic, validation
Persistence	WebStorage, IndexedDB, WebSQL, HTTP, WebSoc
Messaging	WebRTC, WebSocket, Server-Sent Events
I/O	Web Bluetooth, WebUSB, NFC, camera, micropho
Presentation	DOM manipulation, event listeners

Horizontal layers in a web application. Open in new tab.

First of all, it is concerned with presentation. It has an array of heroes which are displayed in its template.

```
<h3>Top Heroes</h3>
<div class="grid grid-pad">
  <a *ngFor="let hero of heroes" class="col-1-4"
      routerLink="/detail/{{hero.id}}">
    <div class="module hero">
      <h4>{{hero.name}}</h4>
    </div>
  </a>
```

Dashboard: Mixed component template. Open in new tab.

While presentation is a valid concern of a UI component, this mixed component is also tightly coupled to state management. In an NgRx application, this component could have injected a `Store` and queried for a piece of the application state with a state selector. In Tour of Heroes, it injects a `HeroService` and queries the heroes state through an observable, then slices a subset of the array and stores a reference in its `heroes` property.

Lifecycle Hook

It is worth pointing out that our mixed dashboard component hooks into the `OnInit` moment of its lifecycle. This is where it subscribes to the observable returned by `HeroService#getHeroes`. It is a proper place to do so, since subscribing to an observable triggers a side effect which we do not want in the constructor or a property initialiser.

In particular, an HTTP request is sent when we subscribe to the observable returned by `HeroService#getHeroes`. By keeping asynchronous code out of constructors and property initialisers, we make our components easier to test and reason about.

. . .

If you feel unsure about the basic concepts of RxJS observables, read "Angular—Introduction to Reactive Extensions (RxJS)" by Gerard Sans:

Angular—Introduction to Reactive Extensions (RxJS)

How to use observable sequences in AngularJS

medium.com

. . .

Splitting a Mixed Component

To separate the multilayer concerns of the mixed component, we split it into two components—a container component and a presentational component.

The container component is responsible for integrating the UI with the non-presentational layers of our application such as the *application state management* and *persistence* layers.

Once we have identified the non-presentational logic in the mixed component, we create the container component by isolating and extracting this logic almost entirely by cutting source code from the mixed component model and pasting it into the container component model.

```
1  import { Component, OnInit } from '@angular/core';
2
3  import { Hero } from '../hero';
4  import { HeroService } from '../hero.service';
5
6  @Component({
7    selector: 'app-dashboard',
8    styleUrls: ['./dashboard.component.css'],
9    templateUrl: './dashboard.component.html',
10 })
11 export class DashboardComponent implements OnInit {
12   heroes: Hero[] = [];
13
14   constructor(private heroService: HeroService) {}
15
16   ngOnInit() {
```

Dashboard: Initial mixed component model. Open in new tab.

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

import { Hero } from '../hero';

@Component({
  selector: 'app-dashboard',
  templateUrl: './dashboard.component.html',
  styleUrls: [ './dashboard.component.css' ]
})
```

Dashboard: Mixed component model after extracting a container component. Open in new tab.

After moving the logic to the container component, a few steps remain to turn the mixed component into a presentational component. These steps are explained in detail in an upcoming article and include renaming the tag name and matching the data binding API to the one we expect to use in the container component template.

Isolate and Extract Layer Integrations

```
import { ChangeDetectionStrategy, Component } from '@a
import { Observable } from 'rxjs';
import { map } from 'rxjs/operators';

import { Hero } from '../hero';
import { HeroService } from '../hero.service';

@Component({
  changeDetection: ChangeDetectionStrategy.OnPush,
  selector: 'app-dashboard',
  templateUrl: './dashboard.container.html',
})
export class DashboardContainerComponent {
```

Dashboard: Container component model. Open in new tab.

We extract the `HeroService` dependency and create a stream of data that matches the data flow in the mixed dashboard component. This is the `topHeroes$` observable property which adds a pipeline of operations on top of the observable returned by `HeroService#getHeroes`.

Our top heroes stream emits a value after the observable from the hero service does so, but only when it is observed—when a subscription has been created. We map over the emitted array of heroes to get the subset of heroes that we present to our users.

Connect the Presentational Component Using Data Bindings

After extracting the application state integration logic, we can—for now—consider the dashboard component a presentational component and assume that it will have a `heroes` input property as seen in the template of the dashboard container component.

The final step in extracting a container component is to connect it to the resulting presentational component through *data bindings*, i.e. property bindings and event bindings in the container component template.

```
<app-dashboard-ui
  [heroes]="topHeroes$ | async"
  title="Top Heroes"></app-dashboard-ui>
```

Dashboard: Container component template. Open in new tab.

`app-dashboard-ui` is the tag name of our dashboard component once it has been turned into a presentational component. We connect our `topHeroes$` observable to its `heroes` input property by using the `async` pipe.

I also extracted the heading text from the mixed component and defined it as `title` in the container component template. I will explain when and why we would want to do this in the upcoming article on presentational components.

For now, be satisfied with the immediate benefit that the presentational dashboard component has the potential to be repurposed in a different part of our app with a heading describing a different subset of heroes that we supply to it.

Who Manages the Subscription?

Interestingly enough, we got rid of the `ngOnInit` lifecycle hook. Our container component model prepares the top heroes data stream by piping from an existing observable which causes no side effects, i.e. no subscription.

Where is the subscription initialised now? The answer is that Angular manages the subscription for us. We declaratively instruct Angular to subscribe to the top heroes observable by using the `async` pipe in the container component template.

The result is a subscription that follows the lifecycle of the presentational dashboard component and emits heroes into the `heroes` input property.

We are happy to get rid of manual subscription management since it is tedious and error-prone. If we forget to unsubscribe from an observable that never completes, we can get multiple subscriptions running for the remainder of the application session, resulting in memory leaks.

Data Flows Down From the Container Component

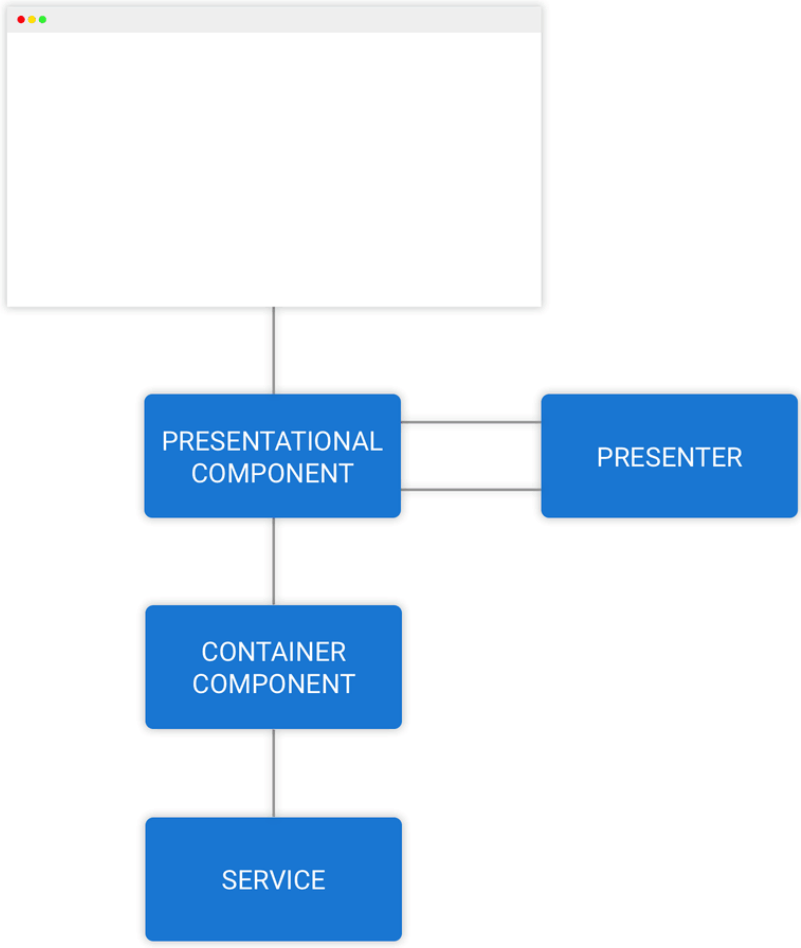


Figure 1. Data flow starting at a service and ending in the DOM. Open in new tab.

Fitting the dashboard feature into the flow diagram of Figure 1, we see how the container component is notified of heroes that it requested from the hero service through an observable.

The container component computes the top heroes which it passes to the presentational component's input property. The heroes array could be passed through a presenter before finally being displayed to the user in the DOM, but the container component is unaware of this since it only knows about the presentational component's data binding API.

Advanced Example

Let us move on to the `HeroesComponent` from Tour of Heroes for a more advanced example.

```
import { Component, OnInit } from '@angular/core';

import { Hero } from '../hero';
import { HeroService } from '../hero.service';

@Component({
  selector: 'app-heroes',
  styleUrls: ['./heroes.component.css'],
  templateUrl: './heroes.component.html',
})
export class HeroesComponent implements OnInit {
  heroes: Hero[];

  constructor(private heroService: HeroService) {}

  ngOnInit() {
    this.getHeroes();
  }

  add(name: string): void {
    name = name.trim();
    if (!name) { return; }
    this.heroService.addHero({ name } as Hero)
      .subscribe(hero => {
        this.heroes.push(hero);
```

Heroes: Mixed component model. Open in new tab.

Isolate Layer Integrations

At first glance, this component might look small, simple and innocent. At closer inspection, it looks like this component has a lot of concerns (pun intended). Like the previous example, the `ngOnInit` lifefycle hook and the `getHeroes` method are concerned with querying for a piece of the application state.

Search this file...

Horizontal layer	Examples
Business logic	Application-specific logic, domain logic, validatior
Persistence	WebStorage, IndexedDB, WebSQL, HTTP, WebSoc
Messaging	WebRTC, WebSocket, Server-Sent Events
I/O	Web Bluetooth, WebUSB, NFC, camera, microphc
Presentation	DOM manipulation, event listeners

Horizontal layers — or system concerns — of a web application. Open in new tab.

The `delete` method deals with persistent state as it replaces the `heroes` property with an array where the deleted hero is filtered out. This method is also concerned with persistence as it deletes a hero from the server state through the hero service.

Finally, the `add` method deals with user interaction as it validates the hero name before creating a hero which is a concern of the persistence and application state layers.

Extract Layer Integrations

Have we got our work cut out for us! Let us get rid of those multilayer system concerns by extracting them into a container component.

```ts
import { Component, OnInit } from '@angular/core';

import { Hero } from '../hero';
import { HeroService } from '../hero.service';

@Component({
  selector: 'app-heroes',
  templateUrl: './heroes.container.html',
})
export class HeroesContainerComponent implements OnIni
  heroes: Hero[];

  constructor(private heroService: HeroService) {}

  ngOnInit() {
    this.getHeroes();
  }

  add(name: string): void {
    this.heroService.addHero({ name } as Hero)
      .subscribe(hero => {
        this.heroes.push(hero);
      });
```

Heroes: Container component with mutable state. Open in new tab.

Like in the simple example, we extract the `HeroService` dependency into a container component. We maintain the heroes state in the mutable `heroes` property.

This will work with the default change detection strategy, but we want to improve performance by using the `OnPush` change detection strategy. We need an observable to manage the heroes state.

The hero service returns an observable emitting an array of heroes, but we also need to support additions and removals of heroes. One solution is to create a stateful observable with a `BehaviorSubject` .

However, to use a subject, we need subscribe to the hero service observable which causes a side effect. If the observable did not complete after emitting a single value, we would also have to manage the subscription ourselves to prevent memory leaks.

Additionally, we have to reduce the heroes state when adding or removing a hero. This quickly starts to become complex.

Managing State

To keep track of application state in a reactive way, I created a microlibrary called `rxjs-multi-scan`. The `multiScan` combination operator merges multiple observables through a single scan operation to calculate the current state but with a—usually small—reducer function per observable source. The operator is passed the initial state as its last parameter.

Every odd parameter—except the initial state parameter—is a source observable and its following, even parameter is its reducer function for the scanned state.

```
1  import { ChangeDetectionStrategy, Component } from '@a
2  import { noop, Observable, Subject } from 'rxjs';
3  import { multiScan } from 'rxjs-multi-scan';
4
5  import { Hero } from '../hero';
6  import { HeroService } from '../hero.service';
7
8  @Component({
9    changeDetection: ChangeDetectionStrategy.OnPush,
10   selector: 'app-heroes',
11   templateUrl: './heroes.container.html',
12 })
13 export class HeroesContainerComponent {
14   private heroAdd: Subject<Hero> = new Subject();
15   private heroRemove: Subject<Hero> = new Subject();
16
17   heroes$: Observable<Hero[]> = multiScan(
18     this.heroService.getHeroes(),
19     (heroes, loadedHeroes) => [...heroes, ...loadedHer
20     this.heroAdd,
21     (heroes, hero) => [...heroes, hero],
22     this.heroRemove,
23     (heroes, hero) => heroes.filter(h => h !== hero),
24     []);
25
26   constructor(private heroService: HeroService) {}
27
28   add(name: string): void {
```

Heroes: Container component model with observable state. Open in new tab.

In our use case, the initial state is an empty array. When the observable returned by `HeroService#getHeroes` emits an array of heroes, it concatenates them to the current state.

I created an RxJS `Subject` per user interaction—one for adding a hero and one for removing a hero. Whenever a hero is emitted through the private `heroAdd` property, the corresponding reducer function in the `multiScan` operation appends it to the current state.

When a hero is removed, the hero is emitted through the `heroRemove` subject which triggers a filter on the current heroes state to filter the specified hero.

Persistence Update Strategies

We allow the addition or deletion of a hero in the public methods `add` and `delete`. When a hero is added, we use the pessimistic update strategy by first persisting the hero to the server state through the hero service and only on success do we update the persistent state in `heroes$`.

Currently, we do not handle errors when updating the server state. This is seen in that the `error` handler in the `subscribe` observer parameter is `noop`. Say we wanted to display a toast to the user or retry the operation, we would do so in the `error` handler.

When deleting a hero, we apply the optimistic update strategy by first removing the hero from the persistent state followed by deletion from the server state. If the deletion fails, we roll back the persistent state by adding back the hero to `heroes$` through the `heroAdd` subject.

This is an improvement over the initial implementation which did not handle server errors when deleting a hero.

Events Flow Up To the Container Component

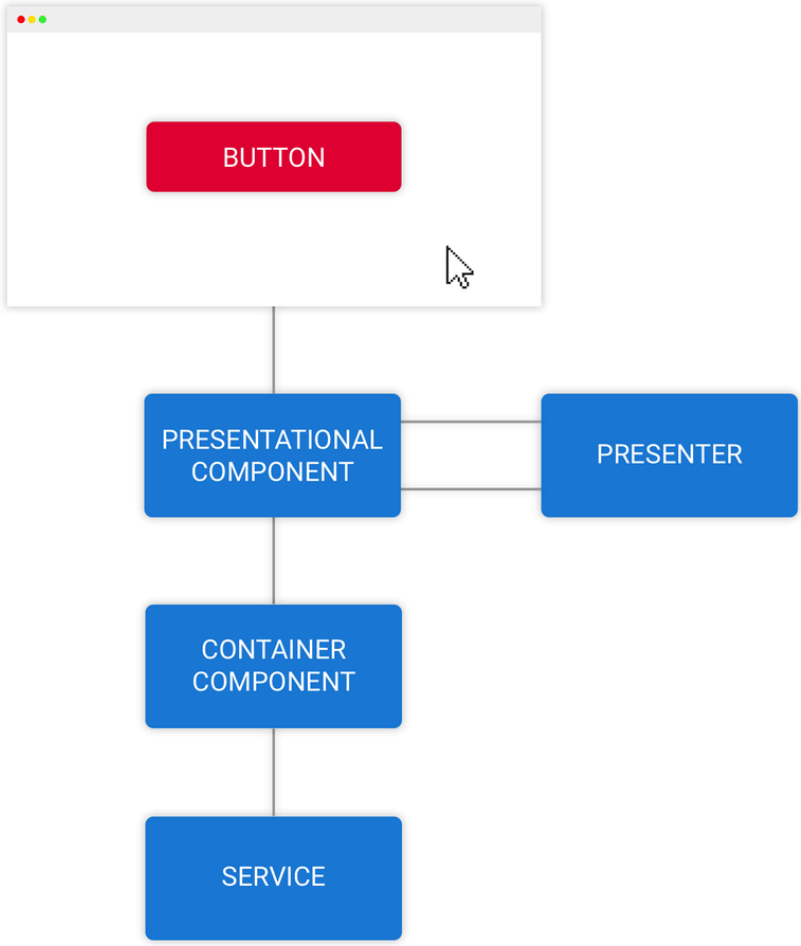


Figure 2. Event flow starting with a user interaction and ending in a service. Open in new tab.

Let us mentally fit the heroes feature into the flow diagram of Figure 2. Visualise how the user enters the hero name and then clicks the *Add* button.

A method on the presentational component model is called with the name of the new hero. The presentational component might delegate user interaction logic to a presenter before it emits the hero name as an event through one of its output properties.

The container component is notified of the emitted hero name which it passes to the hero service and finally updates the persistent state in the container component model.

The updated heroes state notifies the presentational component and the data flow continues as illustrated in Figure 1.

Application State is a Different Concern

It is important to note that while application state can be specific to an application feature, the heroes state is used in multiple areas of Tour of Heroes. As mentioned earlier, it is persistent state that mirrors part of the server state. Ideally, our heroes container component should not be managing persistent state itself, but rather rely on the hero service to do so—or the store in an application that uses NgRx Store.

Despite that the heroes state is managed in a feature-specific container component, it is consistent in the application. This is because the dashboard asks the hero service for the heroes server state every time it is initialised which results in a HTTP request that hydrates (initialises) the persistent state.

In these related articles, we focus on Angular components. In an effort to do so, we will not modify services. If you want to put the heroes state in the hero service where it belongs, you can extract the state management from this container component.

See? Once we separate the concerns, it is easy to isolate a specific type of logic and put it in the application layer that it belongs to.

Working with Immutable Data

In the mixed heroes component, the `Array#push` method was used to add a hero to the heroes state. This mutates the array meaning that a new reference is not created. While this is supported by Angular's default change detection strategy, we opt for performance with the `OnPush` change detection strategy in all our components.

For this strategy to work, we need to emit a fresh array reference whenever a hero is added. We do this by using the spread operator (`...`) in a new array literal to copy heroes from the snapshot (current) value of the heroes and include the additional hero. This new array is emitted to observers of the `heroes$` property.

Leftover Logic

If you follow along in your editor, you might have noticed that we left the validation logic in the mixed heroes component. This is intentional as it is neither concerned with application state nor persistence.

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

import { Hero } from '../hero';

@Component({
  selector: 'app-heroes',
  templateUrl: './heroes.component.html',
  styleUrls: ['./heroes.component.css']
})
export class HeroesComponent {
  heroes: Hero[];

  add(name: string): void {
```

Heroes: Mixed component model after extracting a container component. Open in new tab.

Connect the Presentational Component Using Its Data Binding API

The final step is to connect the container component to the presentational component's data binding API in the container component template.

```
<app-heroes-ui
  [heroes]="heroes$ | async"
  title="My Heroes"
  (add)="add($event)"
```

Heroes: Container component template. Open in new tab.

As in the simple example, we connect the `heroes` input property to our observable property by piping it through `async`. This will pass a fresh array reference to the presentational component, every time the heroes state changes.

Remember that when we use the `async` pipe, Angular manages the subscription to the `heroes$` observable for us so that it follows the lifecycle of the presentational component.

Event Bindings

In the presentational heroes component, our users are able to change the application state by adding or removing heroes. We expect the presentational component to emit a hero through an output property every time the user adds or removes a hero, so we connect the `add`

method of the container component to the presentational component's `add` event.

Likewise, we connect the `delete` method to the `remove` event. I named the method `delete` as the intent is to delete the hero from the server state while keeping the persistent state in sync.

While deletion is an intent that can be expected to be handled by a container component, a presentational component should not be concerned with application state except local UI state. It can only emit a component-specific event when the user asks to remove a hero. The `remove` event is translated to a persistence command by the heroes container component which in turn is expected to change the application state. The new state flows down to the presentational component's input properties in the form of a new array reference.

Apply the `OnPush` Change Detection Strategy

When building a container component, we make sure that we are using observables for streaming the application state. At the same time, we work with immutable data structures exclusively in the observables.

This enables us to use the `OnPush` change detection strategy in the container component, since the `async` pipe triggers change detection when values are emitted through an observable. Because a new reference is emitted with each new value when working with immutable data structures, we will also be able to apply the `OnPush` change detection strategy to the presentational components.

Naming and File Structure

We started out with the `HeroesComponent` which had 4 related files:

- The component-specific stylesheet
- The component template
- The component test suite
- The component model

```
heroes
├── heroes.component.css
├── heroes.component.html
├── heroes.component.spec.ts
├── heroes.component.ts
├── heroes.container.html
```

Heroes: Container component file structure. Open in new tab.

We added the `HeroesContainerComponent` and its test suite. A container component rarely has styles, so only 3 additional files are needed.

I chose to keep the files in a single directory and name the container component files similar to the mixed component files but with a `.container` suffix instead of `.component`.

It is important to note that you can name the files, directories and classes whatever you like. This is a design pattern, not a bunch of laws set in stone.

You like inline templates and stylesheets? or maybe separate directories for the mixed component and the container component files? By all means, use whatever makes sense to your team and you.

Summary

To extract a container component from a mixed component, we go through these steps:

1. Isolate and extract integration with non-presentational layers into a container component.
2. Let the container component stream application state through observables.
3. Connect the container component to the presentational component with data bindings.
4. Apply the `OnPush` change detection strategy.

Remember that container components serve two main purposes:

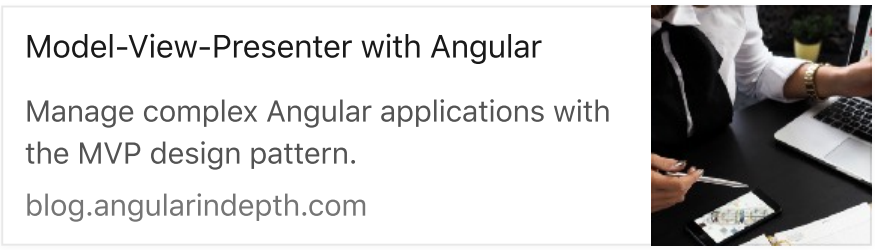
- Container components supply a data flow for presentation.

- Container components translate component-specific events to application state commands—or *actions* to put it in Redux/NgRx Store terms.

One of the big advantages of using container components is increased testability. Continue your study in “Testing Angular Container Components”.

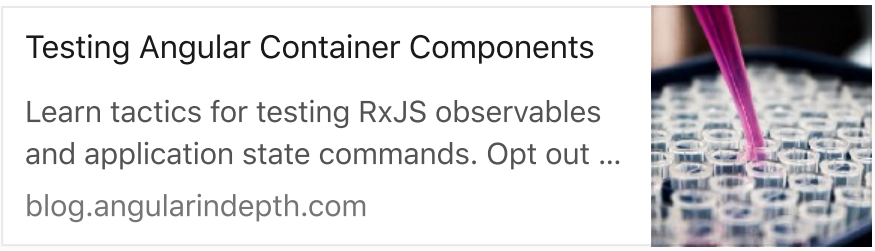
Related Articles

Read the introductory article “Model-View-Presenter with Angular”:



This is also where you will find links to the companion GitHub repository, related articles, and other useful resources.

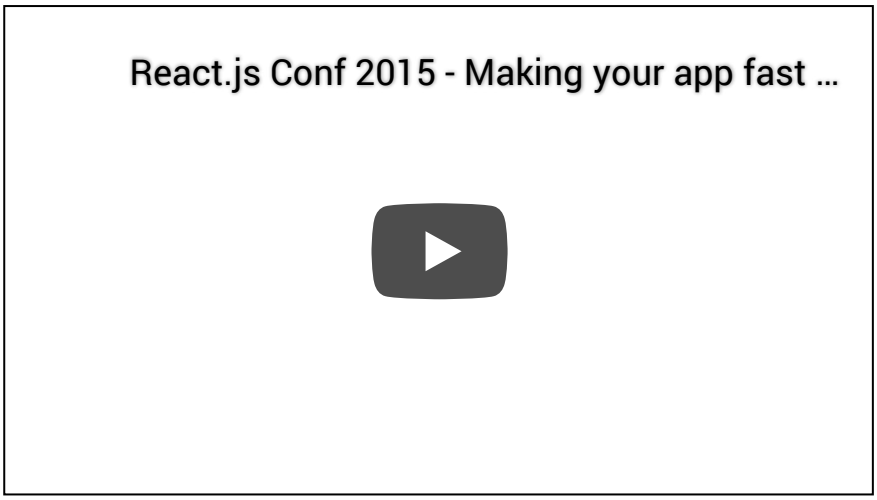
Learn how to test container component logic with blazingly fast unit tests in “Testing Angular Container Components”:



Acknowledgements

Container components have been discussed in the React community for years.

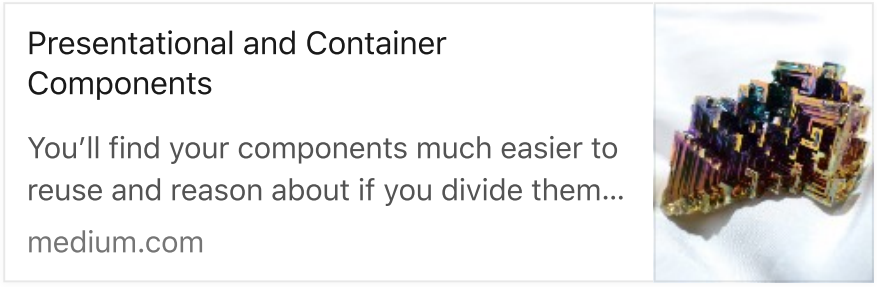
The very first mention of container components is in the talk “Making Your App Fast with High-Performance Components” by Jason Bonta at React Conf 2015:



Making Your App Fast with High-Performance Components, React Conf 2015. Open in new tab.

Michael “chantastic” Chan elaborates a bit and demonstrates a sample component in his 2015 article “Container Components”:

Dan Abramov explains how he divides his React components into container components and presentational components in his 2015 article “Presentational and Container Components”. He continues to discuss related concepts like stateful and stateless components:



Editor

I want to thank you, Max Koretskyi, aka Wizard, for helping me get this article into the best shape possible. I greatly appreciate the time you take to share your experiences about writing for the software development community. Additionally, a big thank you for publishing

my articles so that I can share them with the Angular In Depth audience.

Peer Reviewers

Thank you, dear reviewers, for helping me realise this article. Your feedback has been invaluable!

- Alex Rickabaugh
- Brian Melgaard Hansen
- Craig Spence
- Denise Mauldin
- Kay Khan
- Mahmoud Abduljawad
- Martin Kayser
- Sandra Willford
- Stephen E. Mouritsen Chiang