

Improve Performance With Lazy Components



Oliver Flagg [Follow](#)

Jan 31 · 5 min read



Image by Pixabay.com

Laziness is a virtue in software development. Well, at runtime ;) Lazy-loading e.g. is a well established technique in front-end development. Considering the sheer size of today's apps and websites loading all resources upfront would lead to a disappointing user experience.

But what about the performance of components on an active page or view? Imagine a component that renders animations or continuously fetches data from the server to update a chart. Does it make sense to do all that work if the component is not visible? The tab could be hidden or the component could be located in a part of the page that is currently not visible. That's the case we will examine in this article.

The concepts presented here can be applied to apps written in with any framework or library. The component examples will focus on Angular though.

The stream of visibility

There are two APIs that allow us to determine whether an element is visible or not. The first one is the page visibility API. It allows you to check if the page is currently visible by reading the property `Document.hidden`. It also emits a `visibilitychange` event when, well, the visibility changes. Let's create a stream that emits the visibility of a page:

```
pageVisible$ = fromEvent(document, 'visibilitychange')
  .pipe(
    map(e => !document.hidden)
  );
```

Nothing exciting here. One issue with this simple approach is that any subscriber won't get the current state of the page's visibility. To fix that we add another stream and combine both. We can use `defer` to get the current state at subscription time:

```
pageVisible$ = concat(
  defer(() => of(!document.hidden)),
  fromEvent(document, 'visibilitychange')
  .pipe(
    map(e => !document.hidden)
  )
);
```

Now every time we subscribe to the stream we immediately get the current visibility state and we get notified when that changes.

. . .

The second useful, well, even more useful, API is the Intersection Observer API. It checks an element's visibility on the page and informs us about changes. Not only is it far more performant than other methods, like listening to scroll events, the events provide us with useful additional information. We can also fine-tune the behavior such as adding thresholds or margins. That would allow us to get informed when the element is not yet visible but close.

Let's create another stream:

```
const element = document.getElementById('foo');

const elementVisible$ = Observable.create(observer => {
  const intersectionObserver = new
IntersectionObserver(entries => {
    observer.next(entries);
  })

  intersectionObserver.observe(element);

  return () => { intersectionObserver.disconnect(); });

}).pipe (
  flatMap(entries => entries),
  map(entry => entry.isIntersecting),
  distinctUntilChanged()
);
```

. . .

Now it's time to combine these event streams and make them reusable. That's more tricky than it may seem at first. An element can only be seen by a user if both the tab is visible and the element lies (at least partially) within the viewport. So whenever one of the streams emits a value we need to check if both this value and the latest one from the other stream are both `true`. We can use `combineLatest` with a result selector to achieve that:

```
componentInSight$ = combineLatest(
  pageVisible$,
  elementVisible$
  (pageVisible, elementVisible) => pageVisible &&
  elementVisible
);
```

Voila! One stream that emits `true` as soon as an element can be seen and `false` as soon as it disappears from a user's view.

. . .

Let's put that functionality inside a service. If you remove the decorators and work with native elements instead of `ElementRef` the service can be used with any framework, not just Angular.

```

1  @Injectable()
2  export class VisibilityService {
3
4      private pageVisible$: Observable<boolean>;
5
6      constructor(@Inject(DOCUMENT) document: any) {
7          this.pageVisible$ = concat(
8              defer(() => of(!document.hidden)),
9              fromEvent(document, 'visibilitychange')
10                 .pipe(
11                     map(e => !document.hidden)
12                 )
13          );
14      }
15
16      elementInSight(element: ElementRef):Observable<boolean>
17
18          const elementVisible$ = Observable.create(observer => {
19              const intersectionObserver = new IntersectionObserver(
20                  observer.next(entries);
21              });
22
23              intersectionObserver.observe(element.nativeElement);
24
25              return () => { intersectionObserver.disconnect();
26
27              }
28          ).pipe (
29              flatMap(entries => entries),
30              map(entry => entry.isIntersecting),

```

The Lazy Component

Let's start with our first use case, a component that periodically fetches data from a server.

```

1  @Component({
2    selector: 'price',
3    template: `<h1>{{price$ | async}}</h1>`
4  })
5  export class PriceComponent implements OnInit {
6
7    price$: Observable<number>;
8
9    constructor(private priceService: PriceService) {}
10
11    ngOnInit() {
12
13

```

Every two seconds we get a new price (presumably by making an http request) and the page is updated. This happens whether the component is visible or not.

To detect if the component is visible we first need the host element. We get that by letting it be injected. We also let our `VisibilityService` be injected and change the price stream so that it only fetches data when the component is actually visible:

```

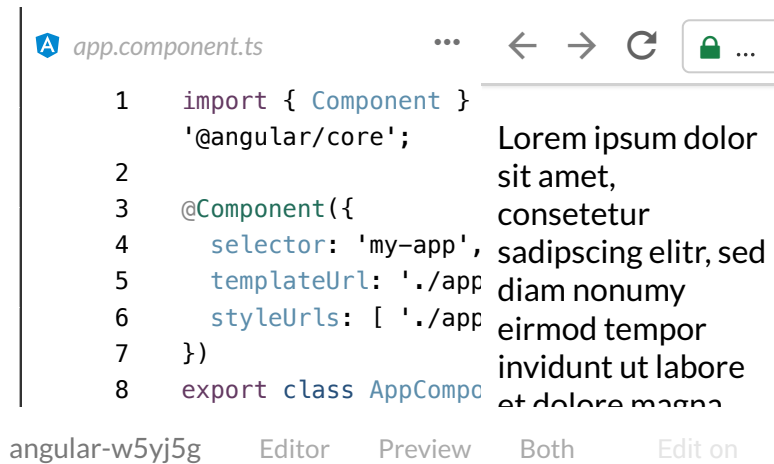
1  @Component({
2    selector: 'price',
3    template: `<h1>{{price$ | async}}</h1>`
4  })
5  export class PriceComponent implements OnInit {
6
7    price$: Observable<number>;
8
9    constructor(private visibilityService: VisibilityService) {}
10
11    ngOnInit() {
12      const inSight$ = this.visibilityService.elementInSight$;
13
14      this.price$ = combineLatest(
15        interval(2000),
16        inSight$

```

We combine the interval and visibility streams and filter all emissions while the component is not visible. Because of the `async` pipe all subscriptions are cancelled automatically.

I have created a stackblitz demo. Resize the window or minimize it and check the console.

<https://stackblitz.com/edit/angular-w5yj5g>



Demo on Stackblitz.

Lazy Creation

What if we want to defer the creation of a component until it's potentially visible? Maybe the creation is expensive or the component uses a large library that we want to load dynamically when the component is created.

The first thing that comes into mind is using `*ngIf`. The problem is, of course, that the element that we want to observe has to exist, which is what we want to prevent for our component.

One possible solution is to observe the parent element. If it's not significantly larger than the lazy component that's a viable option. The alternative is to add a wrapper element, e.g. an empty `div`. To get hold of the element we can use `@ViewChild`.

```
<div #wrapper>
  <hello [name]="name" *ngIf="createComponent | async">
  </hello>
</div>
```

```
export class AppComponent implements OnInit {
```

```
  name = "You";
```

```
  @ViewChild("wrapper") wrapper: ElementRef;
```

```

    createComponent: Observable<boolean>;

    constructor(public visibilityService: VisibilityService)
    {}

    ngOnInit() {

        this.createComponent = this.visibilityService
            .elementInSight(this.wrapper.nativeElement)
            .pipe(filter(visible => visible), take(1));

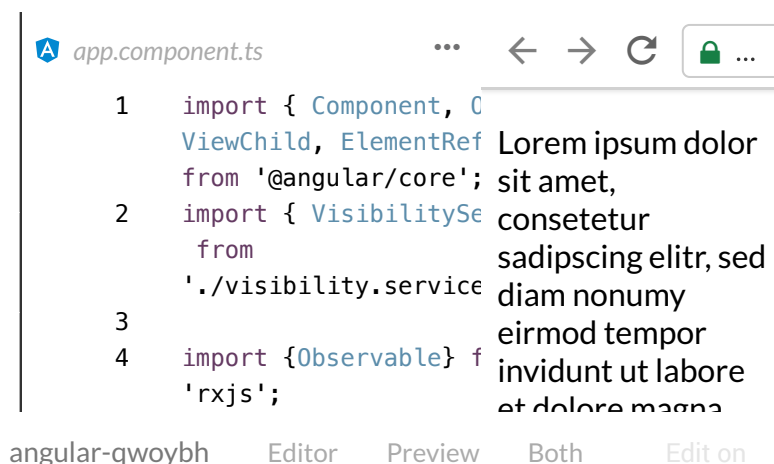
    }

}

```

In the template we add a wrapper around the `hello` component and create a template reference variable (`wrapper`). We can then get hold of the element using `@ViewChild("wrapper")`. Next we create a visibility stream that only emits the first true. That way the `hello` component is not being removed from the page when the wrapper is no longer visible.

<https://stackblitz.com/edit/angular-qwoybh>



```

app.component.ts
1  import { Component, C
    ViewChild, ElementRef
    from '@angular/core';
2  import { VisibilitySe
    from
    './visibility.service
3
4  import {Observable} f
    'rxjs';

```

angular-qwoybh Editor Preview Both Edit on

Demo on Stackblitz

Bonus: Visibility directive

While wrappers and observing parent components work well, that approach still requires some code. So let's create a directive that handles everything for us. To keep things simple our directive creates the component when some other element is visible.

```

1  @Directive({ selector: '[visibleWith]'})
2  export class VisibleWith {
3
4      constructor(
5          private templateRef: TemplateRef<any>,
6          private viewContainer: ViewContainerRef,
7          private visibilityService: VisibilityService) { }
8
9      @Input()
10     set visibleWith(element) {
11         this.visibilityService
12             .elementInSight(new ElementRef(element))
13             .pipe(filter(visible => visible), take(1))

```

The directive can be used like this:

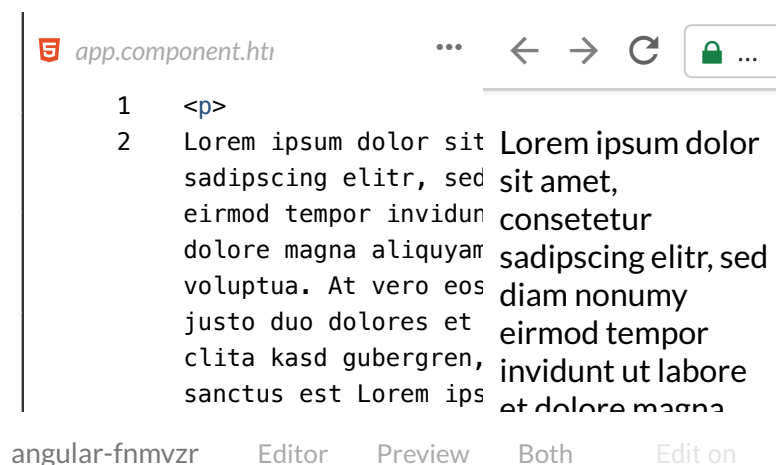
```

<div #wrapper>
<hello [name]="name" *visibleWith="wrapper"></hello>
</div>

```

As you can see the directive expects another (native) element as input. The input setter is similar to what we did in our previous example, except that it creates the component in the subscription handler. The `element` that gets passed in is a native HTML element. Because our service expects an `ElementRef` we need to wrap it.

And here is the demo on Stackblitz



```

1  <p>
2  Lorem ipsum dolor sit Lorem ipsum dolor
  sadipscing elitr, sed sit amet,
  eirmod tempor invidun consetetur
  dolore magna aliquyam sadipscing elitr, sed
  voluptua. At vero eos diam nonumy
  justo duo dolores et eirmod tempor
  clita kasd gubergren, invidunt ut labore
  sanctus est Lorem ips et dolore magna

```

angular-fnmvzr Editor Preview Both Edit on

Conclusion

We have learned quite a few things, such as how to

- Detect the visibility of a page and single elements
- Create an easy to use value stream based on different events
- Perform work only when elements are visible
- Create components only when they would be visible
- Create a structural directive

The performance of web sites and applications is critical for success. Lazy loading has been widely established in the industry. Delaying and avoiding workload is another point to consider and visibility is one determining factor.

