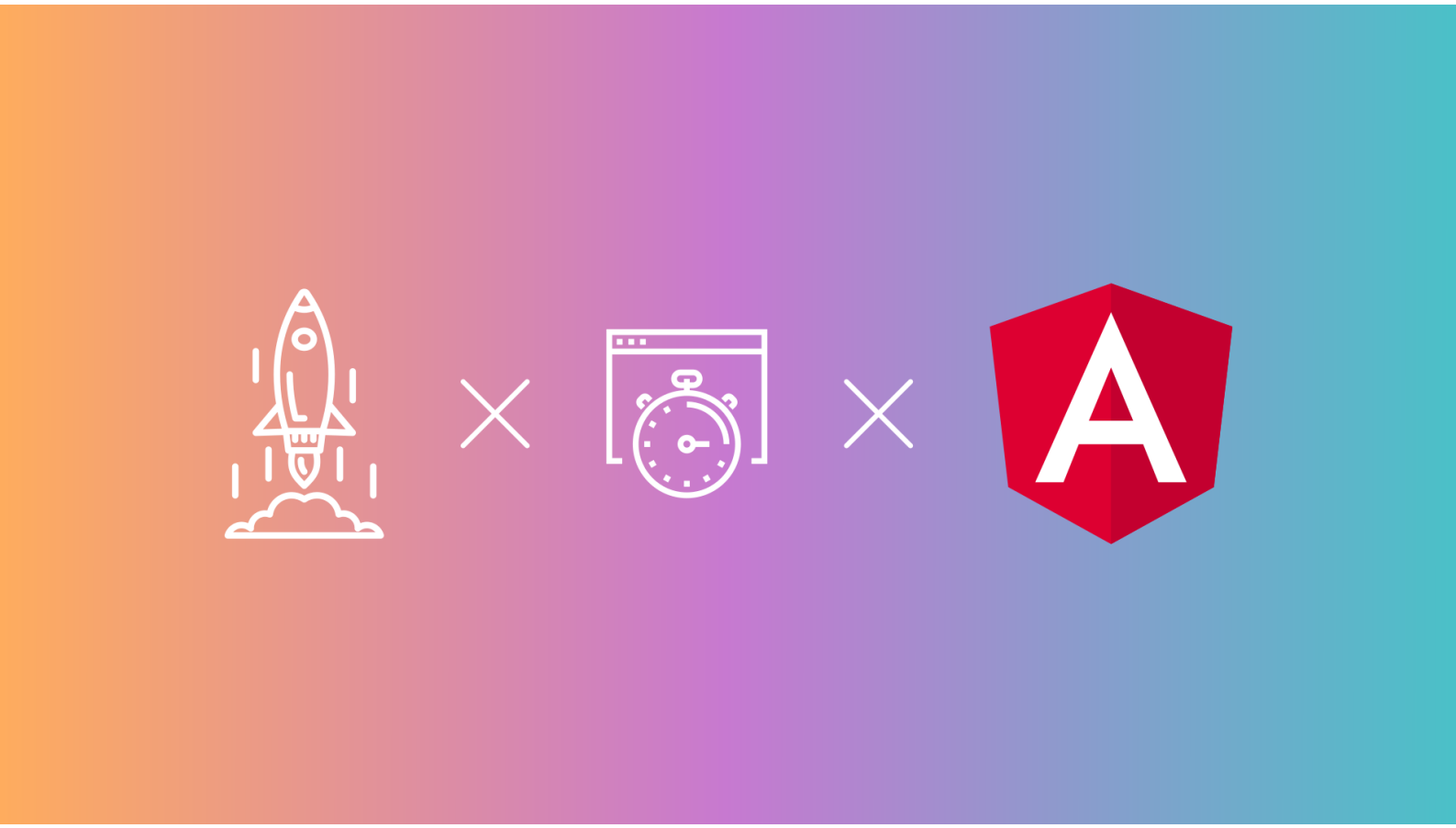


Icon Courtesy: Start-Up By Edwin, Multiply By Kero, Angular by Simon04, and Performance by Maxim Basinski

I changed my implementation of an EXTREMELY deeply nested Angular Reactive Form and you won't believe what happened 🤯

Do you have a deeply nested data structure? Do you have to build a form around it? Is this form really slow to work with? Are your users complaining about it? Do you need an escape? WAIT! I'm here to guide you home.

Siddharth Ajmera Follow

Feb 19 · 13 min read ★

🐼 TL;DR:

1. Analyze a reactive form to identify the scopes for refactoring.

2. Modularize the code by refactoring for better separation of concerns.
3. Refactor the way the Reactive Form is generated in TypeScript.
4. Identifying the source of performance lag.
5. Enhance the performance by refactoring the form template.
6. **Starting Point—StackBlitz Project**
7. **End Goal—StackBlitz Project**

. . .

The inspiration 🔥 to write this article came from a **Stack Overflow *question***, the title of which read:

> *"Angular 7, Reactive Form slow response when has large data"*

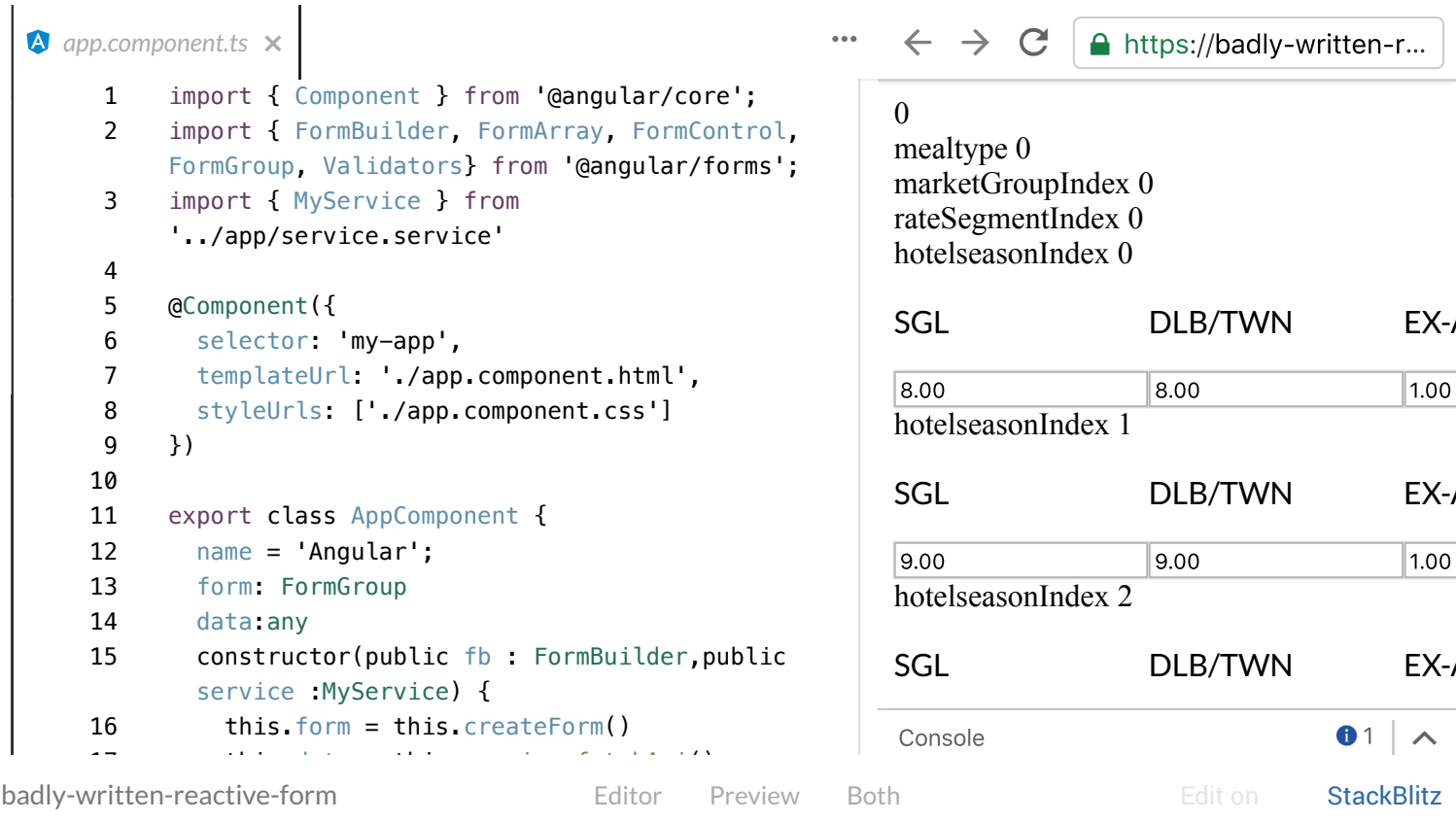
I did answer the question there. But then I thought why not write a detailed medium article about it. So here I am, doing just that.

. . .

✅ Prerequisites:

1. Basics of Angular. If you're new to Angular, I have a free YouTube Playlist on Angular.
2. Familiarity with Reactive Forms in Angular. If you're not comfortable with Reactive Forms, I'd highly recommend this video.
3. I'd also suggest you respond to this article with a comment 💬 if you have trouble understanding any part of it.

This is where we start 👶



Here's the Starting Point of our Code

If we have a look at the above StackBlitz. There are 2 major issues with it:

1. The code is not as per the standards suggested by the **Angular Style Guide**.
2. The App is also lagging in performance when it comes to updates, as well as the initial loads. You can either do a Performance Audit to figure that out. Or just select a particular field and type something in it(or maybe just hammer your keyboard until you see it 😅). It should definitely feel laggy to you.

In this article, we'll target to improve the above App in three parts:

1. We'll first make the code modular.
2. Then we'll refactor the code to align it with the Angular Style Guide.
3. Finally, we'll identify the cause of this lag, refactor the code even further, and fix the App's performance in the process.

So without further ado, let's get started with Step 1.

. . .

STEP 1 — Make the code Modular 👭

Just by having a look at the above StackBlitz, you can infer that the code is not at all structured properly. So we'll start by doing just that.

From what I understand, we need to work with 3 files here:

1. **Service:** The responsibility of this service would be to fetch the data and create a form populated with it. We'll move the extra code in here to other files.
2. **Interface:** We'll be creating the interface for data models to properly model and structure the data that we're getting from the AJAX call.
3. **Data JSON:** We'll be creating a json file. This file would contain the json response that is present as of now in the service file's `fetchApi` itself. Ideally, the data that we're expecting here would come from a REST API. But for the sake of this article, we'll be making a json file. We'll give it a relevant name(`hotel.json` for eg., as it is essentially a file that holds data for a hotel). We'll also keep it in the assets folder so that it could be bundled 📦 as a part of our Angular App. Once this is done, we'll be able to hit `/assets/hotel.json` to get the data.

So let's start right with it

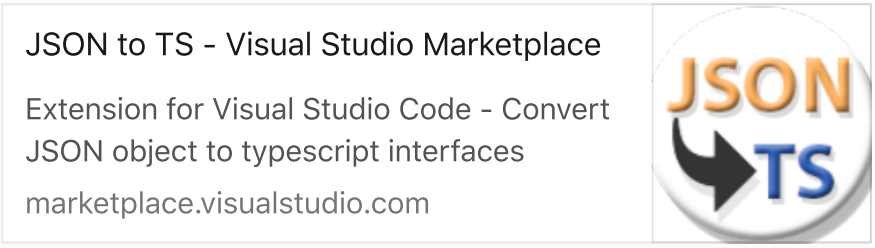
1. Generating the interfaces:

Judging from the json response that we already have in the `fetchApi` method, these would be the interfaces that we'd have:

```ts
export interface Hotel {
  id: string;
  currencyId: string;
  hotelYearId: string;
  priceTaxTypeId: string;
  code: string;
  name: string;
  createBy: string;
  createDate: string;
  lastUpdateBy: string;
  lastUpdateDate: string;
  remark: string;
  internalRemark: string;
  roomTypes: RoomType[];
}

export interface RoomType {
  chk: boolean;
  roomTypeId: string;
  mealTypes: MealType[];
}

export interface MealType {
  chk: boolean;
  mealTypeId: string;
  marketGroups: MarketGroup[];
}

export interface MarketGroup {
  chk: boolean;
  markets: Market[];
  rateSegments: RateSegment[];
}

export interface Market {
  marketId: string;
}

export interface RateSegment {
  chk: boolean;
  rateSegmentId: string;
```

The Interfaces that we created for the Data Models based on the JSON Data

You can generate this manually. But there's also a VSCode Plugin 🔌 that you can leverage to do it for you. I'm linking it below just in case you need it.



Also, we're not building classes for these data models. We're creating interfaces, as that's what's recommended by the **Angular Style Guide**:

> *" Consider using an interface for data models. "*

2. Moving the JSON Data to its own file :

Let's cut the JSON data from the service and move it to a `hotel.json` file.

```
{
  "id": "bef2dd35-6165-48e4-bb73-0f4a6e8cba43",
  "currencyId": "233aadd2-5d16-4df9-8b3d-a632a90cc746"
  "hotelYearId": "713b1389-24f2-4818-aa81-48d82e98ff5b
  "priceTaxTypeId": "00000000-0000-0000-0000-000000000
  "code": "WS2",
  "name": "Wholesale 2",
  "createBy": "system",
  "createDate": "2019-01-26T14:49:31.080Z",
  "lastUpdateBy": "userUpdate",
  "lastUpdateDate": "2019-01-28T11:11:40.541Z",
  "remark": "",
  "internalRemark": "",
  "roomTypes": [
    {
      "chk": true,
      "roomTypeId": "3daf6074-4279-4ef7-a3ae-92e567668
      "mealTypes": [
        {
          "chk": true,
          "mealTypeId": "8ac6b3d1-9f81-4fe3-8bac-96a38
          "marketGroups": [
            {
              "chk": true,
              "markets": [
                {
                  "marketId": "ffffffff-ffff-ffff-ffff
                }
              ],
              "rateSegments": [
                {
                  "chk": true,
                  "rateSegmentId": "00000000-0000-0000
                  "hotelSeasons": [
                    {
                      "chk": true,
                      "hotelSeasonId": "4cf7013a-cf05-
                      "rates": [
                        {
                          "rateCodeId": "00000000-0000
                          "cancellationPolicyId": "16c
                          "dayFlag": "1234567",
                          "singlePrice": "8,100.00",
                          "doublePrice": "8,100.00",
```

```
45                          "xbedPrice": "1,000.00",
46                          "xbedChildPrice": "500.00",
47                          "bfPrice": "400.00",
48                          "bfChildPrice": "200.00",
49                          "unitMonth": "0.00",
50                          "unitDay": "0.00",
51                          "minStay": 0
52                      }
```

A **newly created** file to store the JSON Data for a Hotel

This file will reside at the path `src/assets/` .

3. Fixing the `fetchApi` method in the service:

Now that we've moved the data from the service file to `/src/assets/hotel.json` the `fetchApi` method is broken. We'll have to return something from the method to fix it. That's where the **`HttpClient`** in Angular comes into the picture. We'll import the **`HttpClientModule`** in our `@NgModule` and then add it to the **`imports`** array. Something like this:

```
...
import { HttpClientModule } from '@angular/common/http';
...

@NgModule({
  imports: [..., HttpClientModule, ...],
  ...
})
export class AppModule { }
```

Then you can simply use `HttpClient` by importing it in the service file and injecting it as a Dependency. Something like this:

```
...
import { HttpClient } from '@angular/common/http';

@Injectable()
export class MyService {

  constructor(
    private http: HttpClient,
    ...
  ) {}
```

```
  ...

  fetchApi() {
    return this.http.get('/assets/hotel.json');
  }

}
```

And that concludes Step 1. We've made our code modular so far module. 🎉

. . .

STEP 2 — Refactor the code

Now that we're done with the first part of the whole process, it's now time for refactoring. In this step, we'll take the code that we have as of now, and we'll polish it to greatness.

1. Refactoring the Service:

So we already have the data, and we need to populate our Reactive Form with that data. But for that, we need to have a form in the first place. But this form should be capable enough to deal with the data that we're passing it and then generate the form accordingly.

We'll inject **FormBuilder** as a dependency and use it to generate the form. We'll also use an **Array's** `map` method to transform an Object into a `FormGroup`.

Let's write a method in the service, that's going to generate the form(`getHotelForm`). This method will also accept the data on the basis of which, this form is going to be generated. We'll also set the json data that we got from the API as the default value of the form.

To tackle the deeply nested object arrays in our API response, we'll create methods that would generate those `FormGroup` s for each item in the array. Internally these methods will call `map` on the arrays in the JSON data and map each item in the Array to a corresponding `FormGroup` . So we'll have to implement multiple methods for each `FormGroup` type.

Each of these `generateX` method is returning a `FormGroup` setting it's default values as the parameter that it was accepting. Following this particular signature:

```
generateX(valueParam) {
  const formGroupName = this.fb.group({
    fieldOne: [valueParam.fieldOne, Validators.Required],
    ...,
    arrayField:
this.fb.array(valueParam.someArray.map(itemInArray =>
generateY(itemInArray)))
  });
  return formGroupName;
}
```

Let me show you the code so that it's better for you to understand the gibberish I'm throwing at you😜 :

```
import { Injectable } from '@angular/core';
import { HttpClient } from '@angular/common/http';
import { FormBuilder, Validators } from '@angular/for
import { map } from 'rxjs/operators';

import { Hotel, RoomType, MealType, MarketGroup, Mark

@Injectable()
export class UtilService {

  constructor(
    private readonly fb: FormBuilder,
    private readonly http: HttpClient
  ) { }

  getHotelForm() {
    return this.getHotel().pipe(
      map((apiResponse: any) => this.fb.group({
        id: [apiResponse.id, Validators.required],
        currencyId: [apiResponse.currencyId, Validato
        hotelYearId: [apiResponse.hotelYearId, Valida
        priceTaxTypeId: [apiResponse.priceTaxTypeId,
        code: [apiResponse.code, Validators.required]
        name: [apiResponse.name, Validators.required]
        createBy: [apiResponse.createBy, Validators.r
        createDate: [apiResponse.createDate, Validato
        lastUpdateBy: [apiResponse.lastUpdateBy, Vali
        lastUpdateDate: [apiResponse.lastUpdateDate,
        remark: [apiResponse.remark, Validators.requi
        internalRemark: [apiResponse.internalRemark,
        roomTypes: this.fb.array(apiResponse.roomType
      }))
    );
  }

  private getHotel() {
    return this.http.get('/assets/hotel.json');
  }

  private generateRoomTypeForm(roomType: RoomType) {

    const roomTypeForm = this.fb.group({
      chk: [roomType.chk, Validators.required],
      roomTypeId: [roomType.roomTypeId, Validators.re
      mealTypes: this.fb.array(roomType.mealTypes.map
```

```
45        mealTypes: this.fb.array(roomType.mealTypes.map
46      });
47
48      return roomTypeForm;
49    }
50
51    private generateMealTypeForm(mealType: MealType) {
52
53      const mealTypeForm = this.fb.group({
54        chk: [mealType.chk, Validators.required],
55        mealTypeId: [mealType.mealTypeId, Validators.re
56        marketGroups: this.fb.array(mealType.marketGrou
57      });
58
59      return mealTypeForm;
60    }
61
62    private generateMarketGroupForm(marketGroup: Market
63
64      const marketGroupForm = this.fb.group({
65        chk: [marketGroup.chk, Validators.required],
66        markets: this.fb.array(marketGroup.markets.map(
67        rateSegments: this.fb.array(marketGroup.rateSeg
68      });
69
70      return marketGroupForm;
71    }
72
73    private generateMarketForm(market: Market) {
```

Here's the Refactored Service

Following are a few things that we've done here:

1. I've changed the name of the service to `UtilService`. We should *always make sure that names* that we give to properties and methods and everything else when it comes to programming, *makes sense*.
2. In favor of the above reason, I've also changed the name of the `fetchApi` method to `getHotel`.
3. I've created 8 new methods for the purpose of generating the form and different parts of it. `getHotelForm` will call the `getHotel` method. It would return an `Observable` wrapping the

hotel data in response. We can then `pipe` through it and transform it into a `FormGroup` by using the `map` operator. The response object of the type `Hotel` would be set as the fields' default values to that of the corresponding keys in the object.

4. `getHotelForm` is accompanied by 7 other methods to generate child `FormGroup`s for each different type of object in each different array. Do notice the names of these methods and they should be pretty straightforward. I'm also planning on creating a video 📹 about this that should be more descriptive. So stay tuned for that.

5. Finally, the `getHotelForm` method will return an `Observable` wrapping a `FormGroup` that is populated with the json data that we're getting.

2. Refactoring the Component Class:

Now that we've changed the names of a few methods, we don't really need big and heavy methods like `createForm` and the code in `ngOnInit`. Everything is done by the service itself, we are already getting the form as an `Observable` from the `getHotelForm` method. We also changed the name of the service itself, that would also be broken now in the Component. So we'll have to fix that as well.

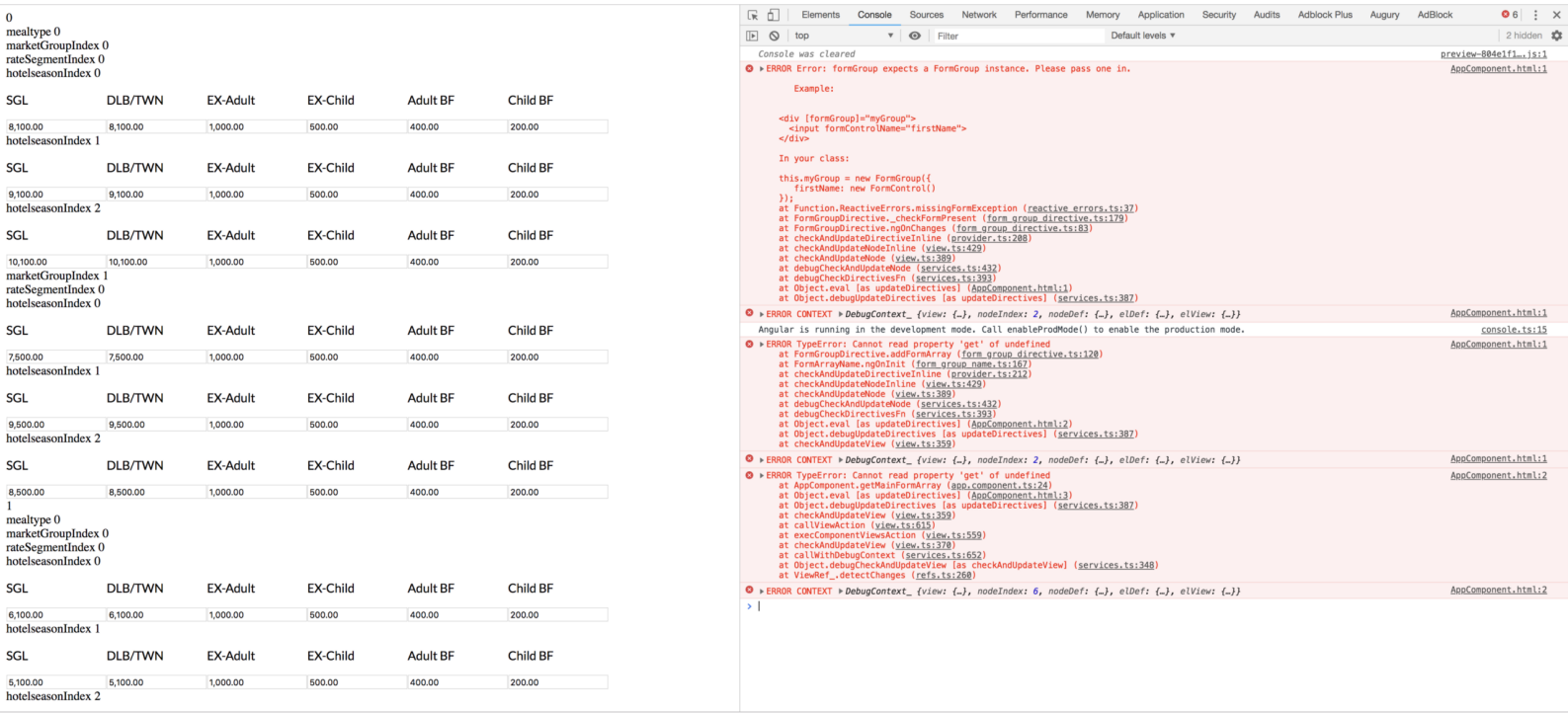
So to refactor this code, we'll copy the code from the `constructor` of the Component Class and replace the current code in `ngOnInit` with it. We'll also get rid of the `createForm` method. And we'll also refactor the constructor to just inject `UtilService` as a dependency. This is how it should look like after refactoring:

```
...
import { FormGroup, FormArray } from '@angular/forms';
import { UtilService } from '../app/util.service';

@Component({...})
export class AppComponent {

  form: FormGroup;

  constructor(private readonly service: UtilService) {}

  ngOnInit() {
    this.service.getHotelForm()
      .subscribe(hotelForm => this.form = hotelForm);
  }

  ...
```

```
}
```

If you've noticed, I've also gotten rid of `FormBuilder` from my imports as I don't really need it anymore.

But now, I'm getting an error on the console. 😅



Error in the Console. Even though the Form is getting displayed properly.

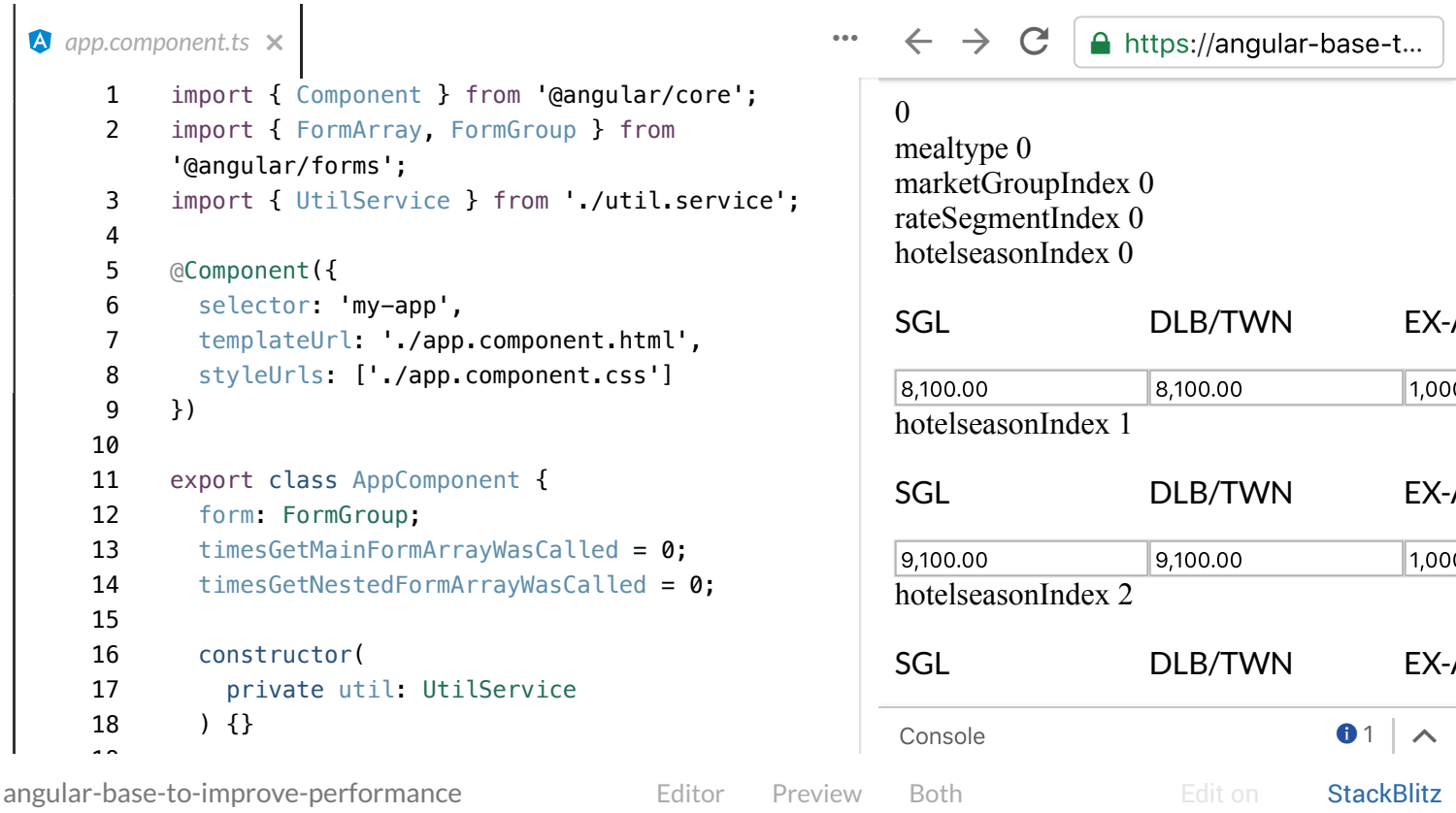
Guess what the reason is? 🤔

Yeap, the reason is `async` hronousity of the `getHotel` method that we're calling in `getHotelForm` on the Service. We're making a call to the json file by calling the `get` method on an `HttpClient` instance. Since this is a network operation, the response would be received asynchronously. But the template will start getting generated parallelly. And at that time `form` would be `undefined` as it would be built only after the API response was received.

So how do we fix it? Well, a small fix would be to put a `*ngIf` on the template.

```
<form *ngIf="form" [formGroup]="form">
  ...
  <pre>{{form.value | json}}</pre>
</form>
```

And this should get you till this point 😃 :



This is the base App on which we will start with the optimization.

. . .

STEP 3 — Optimize the code 🚀

Our code is now well aligned with the Angular Style Guide. But as far as the performance goes, it's not that good. If you want to know how bad the performance really is, just give this exercise a try:

Just type something in any of the fields in the form above. You'll see some lag. Even though the lag is of a few 100 ms, it's still a pretty noticeable lag. And there's a reason for this lag. We'll need to identify the reason in order to fix it, right?

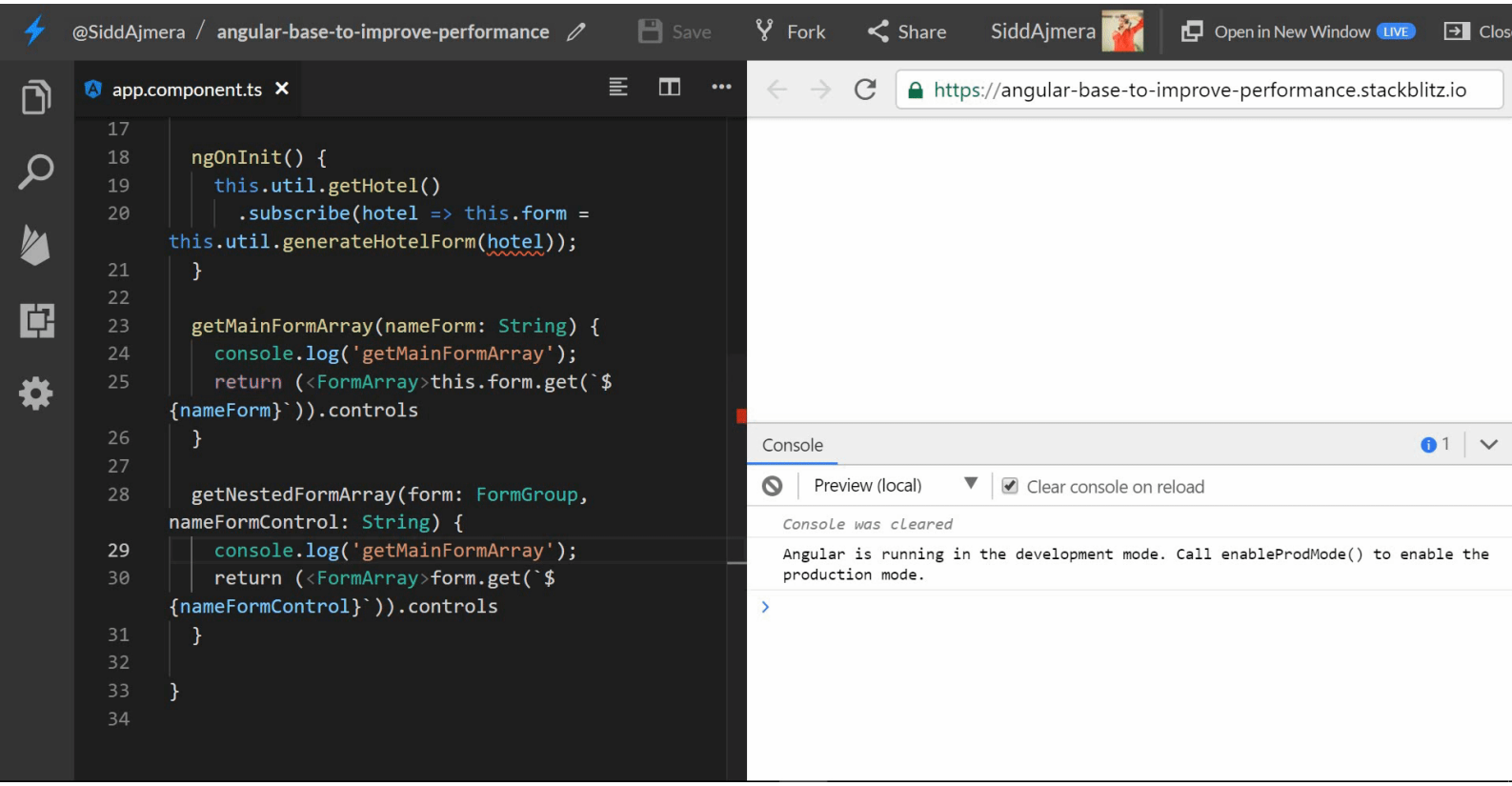
How do I identify the cause of this lag?

Let's try by placing `console.log(…)` in methods `getMainFormArray` and `getNestedFormArray`.

```
import { Component } from '@angular/core';
import { FormArray, FormGroup } from '@angular/forms';

import { UtilService } from './util.service';

@Component({...})
export class AppComponent {

  form: FormGroup;

  constructor(private readonly util: UtilService) {}

  ngOnInit() {
    this.util.getHotelForm()
      .subscribe(hotelForm => this.form = hotelForm);
  }

  getMainFormArray(nameForm: String) {
    console.log('getMainFormArray');
    return
(<FormArray>this.form.get(`${nameForm}`)).controls;
  }

  getNestedFormArray(form: FormGroup, nameFormControl:
String) {
    // console.log('getMainFormArray');
    return
(<FormArray>form.get(`${nameFormControl}`)).controls;
  }

}
```

Now let's check the console and see how many logs were generated. It should be something along the lines of 133.

Now let's do one more thing, let's type in 111111 in any one of the input fields. Blur from that input field. And then remove 111111 from that input field. And now let's check the number of logs. It should now be somewhere around 1401.



See the lag? And the **source** for this lag!

It's quite apparent that the methods that we have in the Component are getting called pretty much every time Angular is performing change detection on the Component. This is also something that Tanner Edwards points out in his lightning talk at ng-conf 2018:

Increasing Performance — more than a pipe dream — Tanner Edwards

As we're well aware, Angular performs change-detection in the following cases:

1. AJAX Calls
2. Timeouts/Intervals
3. DOM Events.

In our case, keyboard taps (DOM Events) are making Angular to detect changes again and again. And since we're calling these methods in the template, Angular's Change Detection will blow up the call stack. And hence, we're getting those thousands of logs resulting in the lag.

So what now? Should I remove the methods? And if I do, how do I get the controls for my *ngFor in the template?

Hmmmm. This is going to be tricky. But `FormGroup` s have something called `controls` . These are the objects that contain the actual `FormGroup` , `FormControl` or `FormArray` .

So instead of calling these getters or methods to get these `FormArray` s in the template, we can use this `controls` object. Something like this:

```
let roomType of form.controls['roomTypes'].controls
```

to get the `roomTypes` `FormArray` 's `controls` that we could then loop over. And we'll replace all those function calls that we have in our Component's Template with the above syntax. Once we're done with it, our Component Template will look something like this:

```
<form *ngIf="form" [formGroup]="form">
  <div formArrayName="roomTypes">
    <div *ngFor="let roomType of form.controls['roomTy
      {{index}}
      <div formArrayName="mealTypes">
        <div *ngFor="let mealType of roomType.controls
          mealtype {{mealtypeIndex}}
          <div formArrayName="marketGroups">
            <div *ngFor="let marketGroup of mealType.c
              marketGroupIndex {{marketGroupIndex}}
              <div formArrayName="rateSegments">
                <div *ngFor="let rateSegment of market
                  rateSegmentIndex {{rateSegmentIndex}
                  <div formArrayName="hotelSeasons">
                    <div class="fifth_border" *ngFor="
                      hotelseasonIndex {{hotelseasonIn
                      <div formArrayName="rates">
                        <div *ngFor="let rate of hotel
                          <div style="display:flex;fle
                            <div>
                              <p>SGL</p>
                              <input class="input text
                            </div>
                            <div>
                              <p>DLB/TWN</p>
                              <input class="input text
                            </div>
                            <div>
                              <p>EX-Adult</p>
                              <input class="input text
                            </div>
                            <div>
                              <p>EX-Child</p>
                              <input class="input text
                            </div>
                            <div>
                              <p>Adult BF</p>
```

Here's the Changed Template

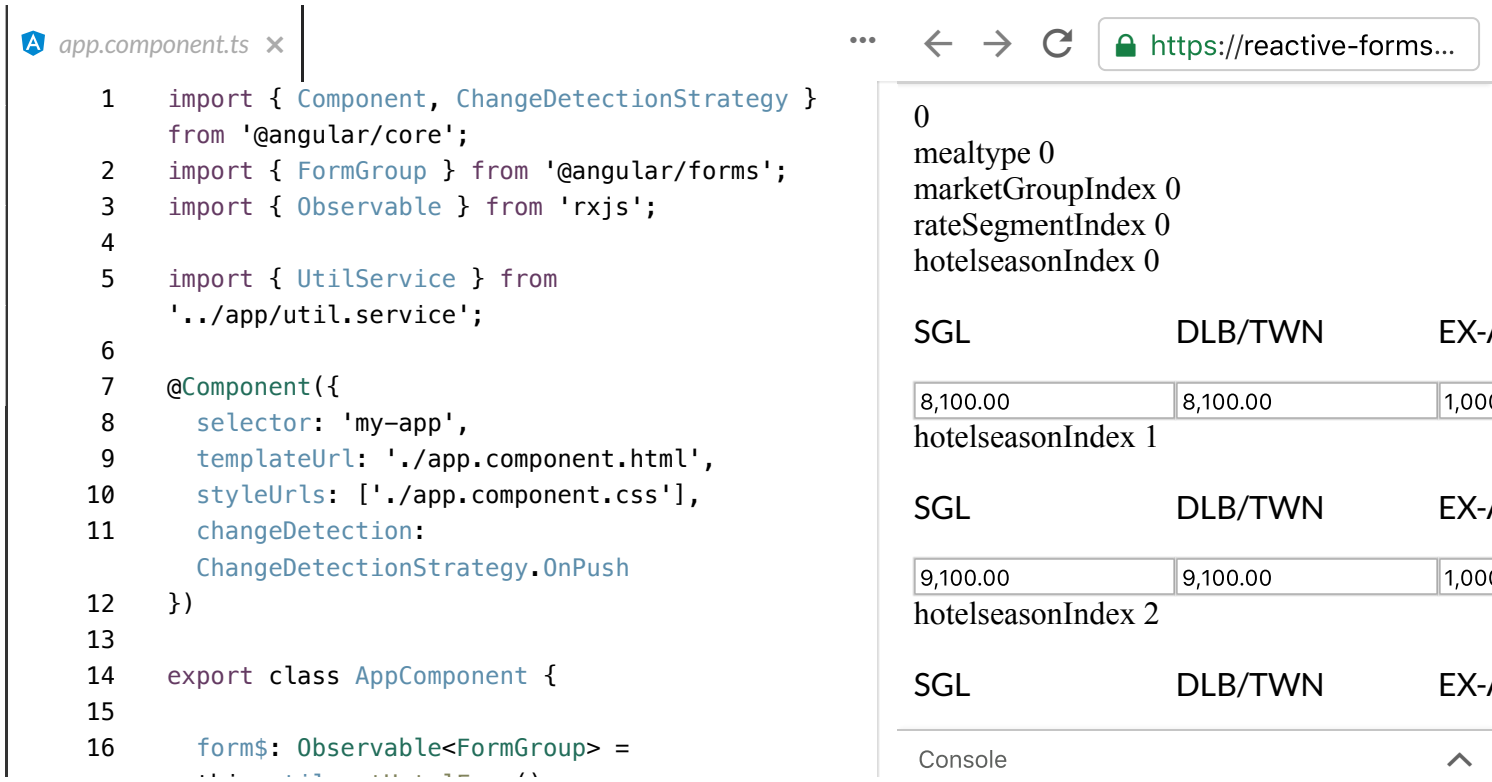
Also now that we don't really call these methods on our Component Class, we can simply get rid of them and reduce the AppComponent to something like this:

```
import { Component } from '@angular/core';
import { FormGroup } from '@angular/forms';
import { Observable } from 'rxjs';

import { UtilService } from '../app/util.service';

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

export class AppComponent {
  form$: Observable<FormGroup> = this.util.getHotelForm();
  constructor(private readonly util: UtilService) {}
}
```

And yeah, since we're not really `subscribe` ing to the Observable now, we will have to use the `async` pipe in our Component template:

```
<form *ngIf="form$ | async as form" [formGroup]="form">
  ...
  <pre>{{form.value | json}}</pre>
</form>
```

Now just test this StackBlitz out and check if you see any difference in the performance:



This is our **Semi-Final Code**

Yeah. Okay! I totally understand if you don't see a noticeable change. The reason is that the initial lag was a few 100 milliseconds. And the updates to the current fields in the above StackBlitz would also take a few milliseconds to update. The difference would be somewhere around a few 100 milliseconds less than what it was before.

So? Are we done here? Is that it?

Not at all. This is just the beginning. You can further improve the performance of this form. How? Well, as suggested by Benedikt L, and Garrett Darnell we can break the `form` even further and create a child component for the `marketGroup` `FormGroup` . And then, we can set the `changeDetectionStrategy` on that child component to `OnPush` . Something like this:

```
import { Component, ChangeDetectionStrategy, Input } from
'@angular/core';
import { FormGroup } from '@angular/forms';

@Component({
  selector: 'market-group-form',
  templateUrl: './market-group-form.component.html',
```

```
  changeDetection: ChangeDetectionStrategy.OnPush
})

export class MarketGroupFormComponent {
  @Input() public marketGroupForm: FormGroup;
}
```

We can simply cut the template for this component from our `app.component.html`:

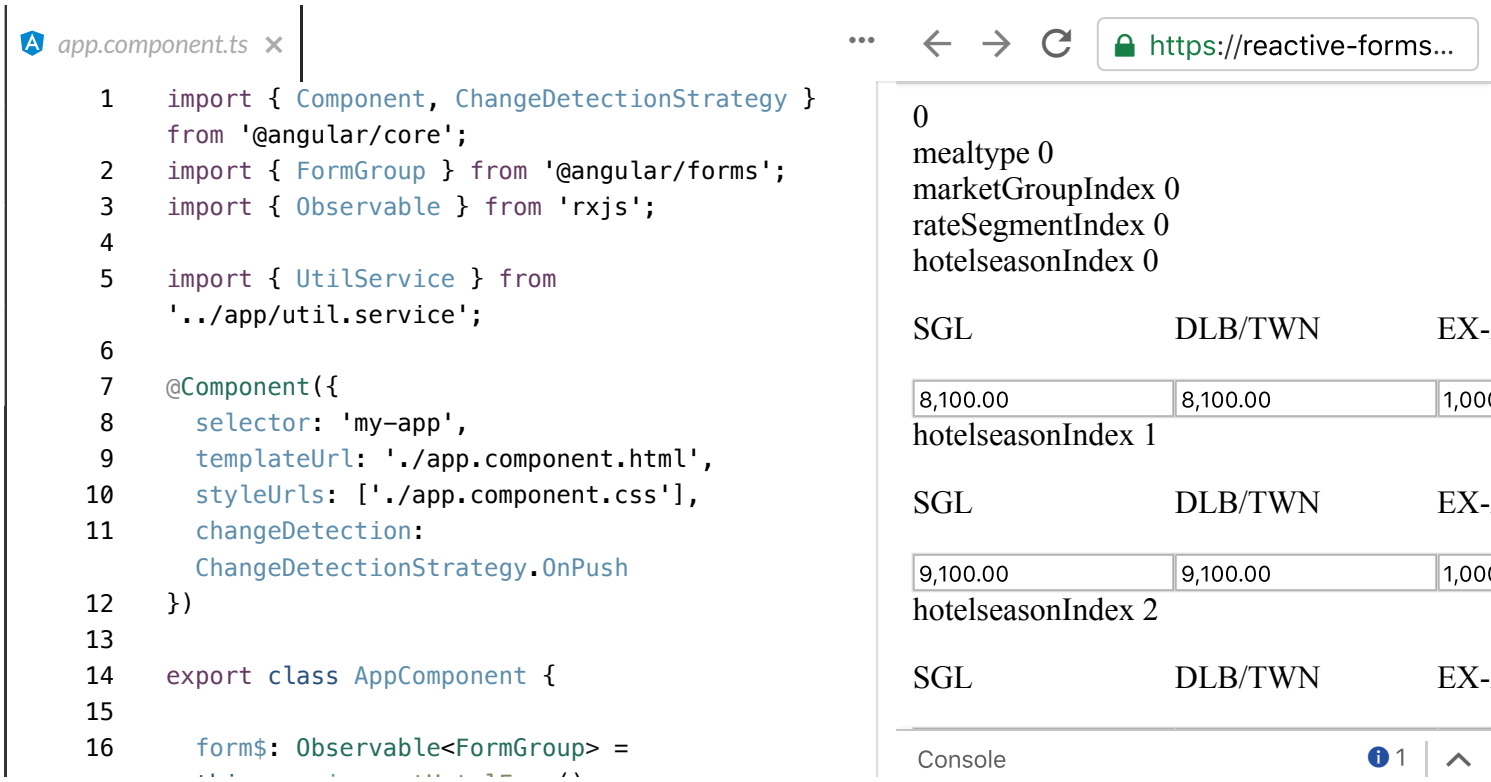
```
<div [formGroup]="marketGroupForm">
  <div formArrayName="rateSegments">
    <div
      *ngFor="let rateSegment of marketGroupForm.contr
      [formGroupName]="rateSegmentIndex">
      rateSegmentIndex {{rateSegmentIndex}}
      <div formArrayName="hotelSeasons">
        <div
          class="fifth_border"
          *ngFor="let hotelseason of rateSegment.contr
          [formGroupName]="hotelseasonIndex">
          hotelseasonIndex {{hotelseasonIndex}}
          <div formArrayName="rates">
            <div
              *ngFor="let rate of hotelseason.controls
              [formGroupName]="rateIndex">
              <div style="display:flex;flex-flow;row">
                <div>
                  <p>SGL</p>
                  <input class="input text_right" type
                </div>
                <div>
                  <p>DLB/TWN</p>
                  <input class="input text_right" type
                </div>
                <div>
                  <p>EX-Adult</p>
                  <input class="input text_right" type
                </div>
                <div>
                  <p>EX-Child</p>
                  <input class="input text_right" type
```

And then use it in our app.component.html like this:

```
<form
  *ngIf="form$ | async as form"
  [formGroup]="form">
  <div
    formArrayName="roomTypes">
    <div
      *ngFor="let roomType of form.controls['roomTypes
      [formGroupName]="index">
      {{index}}
      <div
        formArrayName="mealTypes">
        <div
          *ngFor="let mealType of roomType.controls['m
          [formGroupName]="mealtypeIndex">
          mealtype {{mealtypeIndex}}
          <div
            formArrayName="marketGroups">
            <div
              *ngFor="let marketGroup of mealType.cont
              [formGroupName]="marketGroupIndex">
```

Notice **Line 22** where we're using **<market-group-form ></market-group-form>**

Ideally, the decision to break this component further down into child component(s) can be taken based on how complex or deeply nested, your form is. On each of these child component(s) you can then specify the `changeDetection` to be `ChangeDetectionStrategy.OnPush` . Angular won't detect any changes on the Child Components if nothing changed there due to this. And this would further improve the performance. And after changing all of that, this is how your final implementation should look like:



This is our **Final Implementation**

Now just test this StackBlitz out and check if you see any difference in the performance. I'm sure you'll notice a significant change this time around.

Well Sidd, do you have any stats to suggest that the refactored code is faster? 🤔

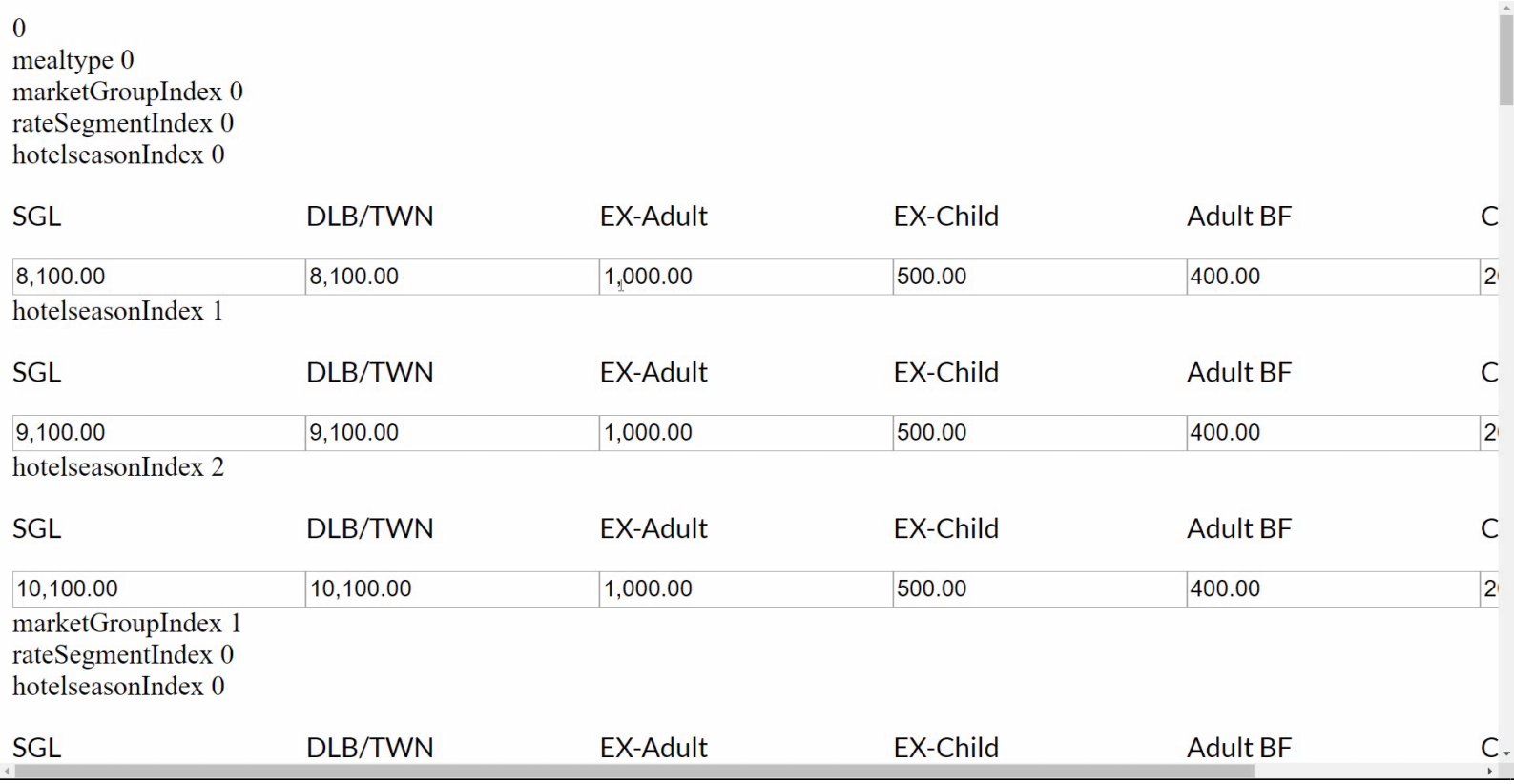
Sure. How about a Performance Audits?

Performance Audits are a pretty neat way of determining the performance of your Apps. I'm using Google Chrome so I'll do it in there.

So we open up the Chrome Debugger tools and then open up the Performance Tab. Once you're there, just do the following:

1. Click on the Record Button.
2. Click on a particular field and type 111111 in there.

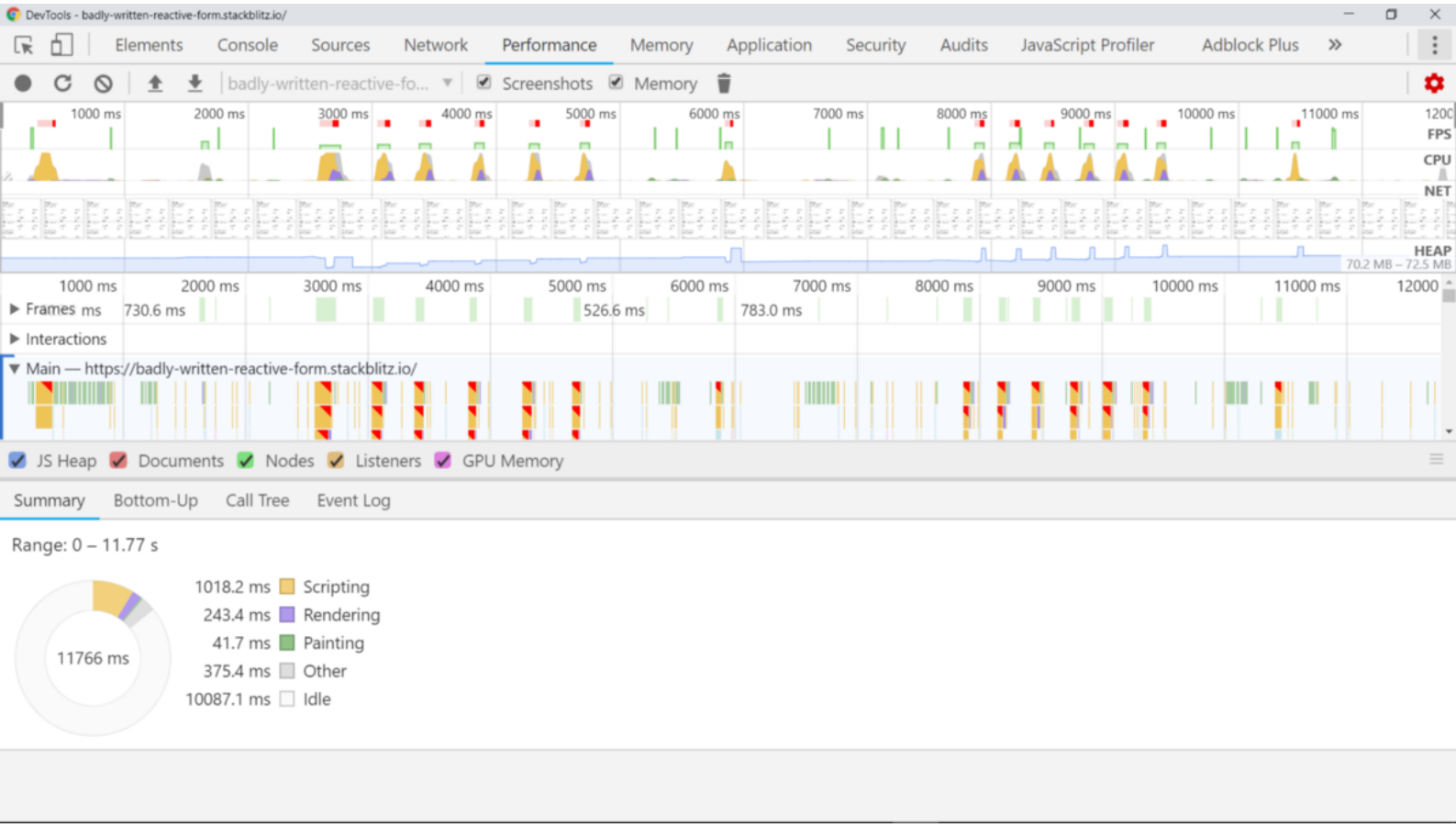
3. Then blur from that field.
4. Click on the same field again and then remove 111111 from there.
5. Click on the Stop button to stop the recording of your Audit Session.



Here's a Gif of the Steps you'd follow to perform one

Follow the above steps for both the **Starting Point** and **End Goal a**nd notice the difference. Following are the stats that I got when I did it:

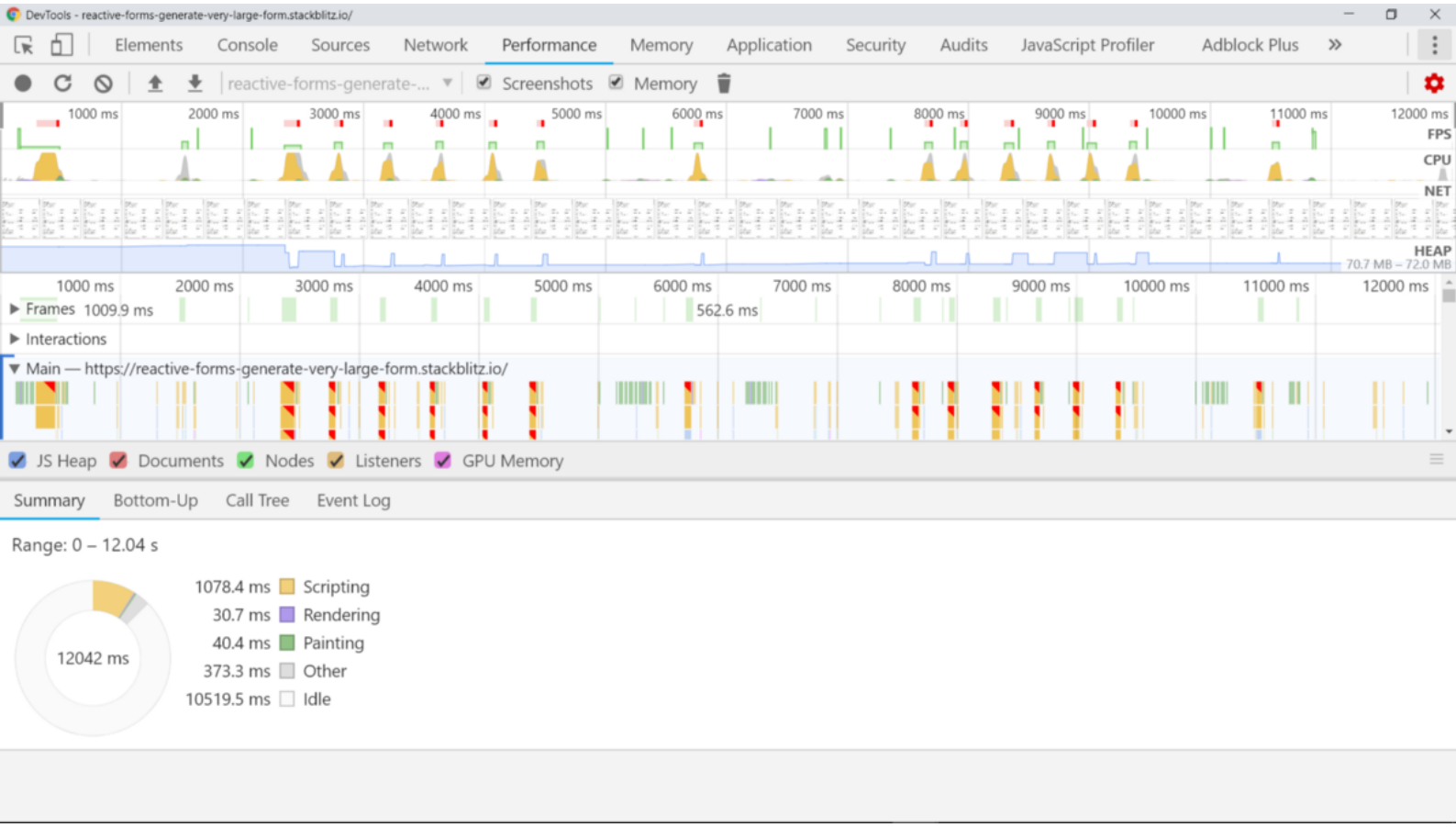
Here's what I got for the Starting Point Example:



These are the stats for the **Starting** Point

Things to note down in the stats are the time taken for Scripting, Rendering and Painting the UI. The Status shows that it took 1018.2ms to script, 268.0ms to render and 62.5ms to paint the UI.

Let's see how the End Goal Example perform:



These are the stats for the **End Goal**

As you can see, rendering(that happens frequently on updates) took almost 8 x less time in the Final Goal. There's a clear performance gain of about 792% as far as the rendering goes. This time, it took 1078.4 ms to script, 30.7 ms to render and 40.4 ms to paint the UI. And this looks pretty significant to me. What do you think?

Whoa, you're still reading 😍

So what did we learn today? We learned that we should never call functions in string interpolation(`{{ fn() }}`) or property binding(`[prop]= "fn()"`) syntax. It makes those bound functions run on every change detection cycle, thereby killing the performance of your app. We also learned how `OnPush` is a very powerful strategy that we can use to improve the performance of our Angular Apps.

I hope you can take this learning home and then improve your past apps, implement these performance improvement strategies in your current apps and keep them in mind for the future app that you'll be building in Angular.

• • •

Closing notes 🎉:

On that note, I'd like to conclude this article. Thanks a lot for reading it till the end. I hope I didn't bore you. 😅

I'm extremely grateful to Alex Okrushko, Michael Karén, Max Koretskyi aka Wizard, and Rajat Badjatya for taking the time to proofread it and providing all the constructive feedback in making this article better.

I hope this article has taught you something new related to Angular and Reactive Forms 👌 . If it did, share this article with your friends who are new to Angular and want to achieve something similar. Also, comment 💬 below 👇 on what you'd like to read about next. **Until next time then.** 👋