ACADGILD

LEARN. DO. EARN

FRONTEND DEVELOPMENT

(WITH ANGULAR JS)

ACADGILD

Migrating Angular 1 to 2 & Performance Considerations

Session - 17

Agenda

ACADGILD

Sl. No.	Agenda Title
1	Migration From Angular 1.x to 2.0
2	Different Ways of Migration
3	Following The Angular Style Guide
4	Using a Module Loader
5	Migrating to TypeScript
6	Using Component Directives

Sl. No.	Agenda Title
7	Projecting Angular 1 Content into v2 Component
8	Change Detection
9	What Causes Change?
10	Change Detection Strategies
11	Bypassing Components
12	Performance

Migration From Angular 1.x to 2.0

- Having an existing Angular 1 application doesn't mean that we can't begin enjoying everything Angular 2 has to offer. That's because Angular 2 comes with built-in tools for migrating Angular 1 projects over to the Angular 2 platform.

- Some applications will be easier to upgrade than others, and there are ways in which we can make it easier for ourselves. It is possible to prepare and align Angular 1 applications with Angular 2 even before beginning the upgrade process. These preparation steps are all about making the code more decoupled, more maintainable, and up to speed with modern development tools. That means the preparation work will not only make the eventual upgrade easier, but will also generally improve.

Migration From Angular 1.x to 2.0

- ***One of the keys to a successful upgrade*** is to do it incrementally, by running the two frameworks side by side in the same application, and porting Angular 1 components to Angular 2 one by one. This makes it possible to upgrade even large and complex applications without disrupting other business, because the work can be done collaboratively and spread over a period of time. The upgrade module in Angular 2 has been designed to make incremental upgrading seamless.

Different Ways of Migration

ACADGILD

- Upgrading to Angular 2 is quite an easy step to take, but should be made carefully.
- There are two major flavors of Angular 2. Which you use depends on the requirements of your project. The angular team have provided two paths to this:
  - NGFORWARD
  - NGUPGRADE
- ngForward is not a real upgrade framework for Angular 2 but instead we can use it to create Angular 1 apps that look like Angular 2.
- If you still feel uncomfortable upgrading your existing application to Angular 2, you can opt for ngForward

Different Ways of Migration (Contd.)

ACADGILD

- You can either re-write your angular app gradually to look as if it was written in Angular 2 or add features in an Angular 2 manner leaving the existing project untouched. Another benefit that comes with this is that it prepares you and your team for the future even when you choose to hold onto the past for a little bit longer.
- You will be guided through a basic setup to use ngForward but in order to be on track, have a look at the Quick Start for Angular 2. In your existing Angular 1.x (should be 1.3+) app run:
- **npm i --save ng-forward@latest reflect-metadata**

Different Ways of Migration (Contd.)

NGUPGRADE:

- Writing an Angular 1.x app that looks like Angular 2 is not good enough. We need the real stuff. The challenge then becomes that with a large existing Angular 1.x project, it becomes really difficult to re-write all our app to Angular 2, and even using ngForward would not be ideal. This is where ngUpgrade comes to our aid. ngUpgrade is the real stuff.
- Unlike ngForward, ngUpgrade was covered clearly in the Angular 2 docs. If you fall in the category of developers that will take this path, then spare few minutes and digest this.

Different Ways of Migration (Contd.)

- In manual upgradation, we can check out angular 2 official docs, we will follow that approach here:

Preparation

- Following The Angular Style Guide
- Using a Module Loader
- Migrating to TypeScript
- Using Component Directives

Upgrading with The Upgrade Adapter

- How The Upgrade Adapter Works
- Bootstrapping Hybrid Angular 1+2 Applications
- Using Angular 2 Components from Angular 1 Code

Different Ways of Migration (Contd.)

ACADGILD

- Using Angular 1 Component Directives from Angular 2 Code
- Projecting Angular 1 Content into Angular 2 Components
- Transcluding Angular 2 Content into Angular 1 Component Directives
- Making Angular 1 Dependencies Injectable to Angular 2
- Making Angular 2 Dependencies Injectable to Angular 1

Following The Angular Style Guide

ACADGILD

- **Angular 2** is a reimagined version of the best parts of Angular 1. In that sense, its goals are the same as the Angular Style Guide's: To preserve the good parts of Angular 1, and to avoid the bad parts. There's a lot more to Angular 2 than just that of course, but this does mean that following the style guide helps make your Angular 1 app more closely aligned with Angular 2.

Following The Angular Style Guide (Contd.)

ACADGILD

- There are a few rules in particular that will make it much easier to do an incremental upgrade using the Angular 2 upgrade module:
- The Rule of 1 states that there should be one component per file. This not only makes components easy to navigate and find, but will also allow us to migrate them between languages and frameworks one at a time. In this example application, each controller, component, service, and filter is in its own source file.
- The **Folders-by-Feature** Structure and Modularity rules define similar principles on a higher level of abstraction: Different parts of the application should reside in different directories and Angular modules.

Using a Module Loader

ACADGILD

- When we break application code down into one component per file, we often end up with a project structure with a large number of relatively small files. This is a much neater way to organize things than a small number of large files, but it doesn't work that well if you have to load all those files to the HTML page with <script> tags. Especially when you also have to maintain those tags in the correct order. That's why it's a good idea to start using a module loader.
  - Using a module loader such as SystemJS, Webpack, or Browserify allows us to use the built-in module systems of the TypeScript or ES2015 languages in our apps.
  - We can use the import and export features that explicitly specify what code can and will be shared between different parts of the application.

Migrating to TypeScript

ACADGILD

- If part of our Angular 2 upgrade plan is to also take TypeScript into use, it makes sense to bring in the TypeScript compiler even before the upgrade itself begins. It also means we can start using TypeScript features in our Angular 1 code.

- Since **TypeScript** is a superset of ECMAScript 2015, which in turn is a superset of ECMAScript 5, "**switching**" to TypeScript doesn't necessarily require anything more than installing the TypeScript compiler and switching renaming files from *.js to *.ts. But just doing that is not hugely useful or exciting.

Using Component Directives

ACADGILD

- **In Angular 2**, components are the **main primitive** from which user **interfaces** are **built**. We define the different parts of our UIs as components, and then compose the UI by using components in our templates.
- **Angular 1.5** introduces the ***component API*** that makes it easier to define directives like these. It is a good idea to use this API for component directives for several reasons:
  - It requires less boilerplate code.
  - It enforces the use of component best practices like controllerAs.
  - It has good default values for directive attributes like scope and restrict.

Using Component Directives (Contd.)

ACADGILD

```
export const heroDetail = {
  bindings: {
    hero: '=',
    deleted: '&'
  },
  template: `
    <h2>{{$ctrl.hero.name}} details!</h2>
    <div><label>id: </label>{{$ctrl.hero.id}}</div>
    <button ng-click="$ctrl.onDelete()">Delete</button>
  `,
  controller: function() {
    this.onDelete = () => {
      this.deleted(this.hero);
    };
  }
};
```

Using Component Directives (Contd.)

- Controller lifecycle hook methods $onInit(), $onDestroy(), and $onChanges() are another convenient feature that Angular 1.5 introduces. They all have nearly exact equivalents in Angular 2, so organizing component lifecycle logic around them will ease the eventual Angular 2 upgrade process.

How The Upgrade Adapter Works:

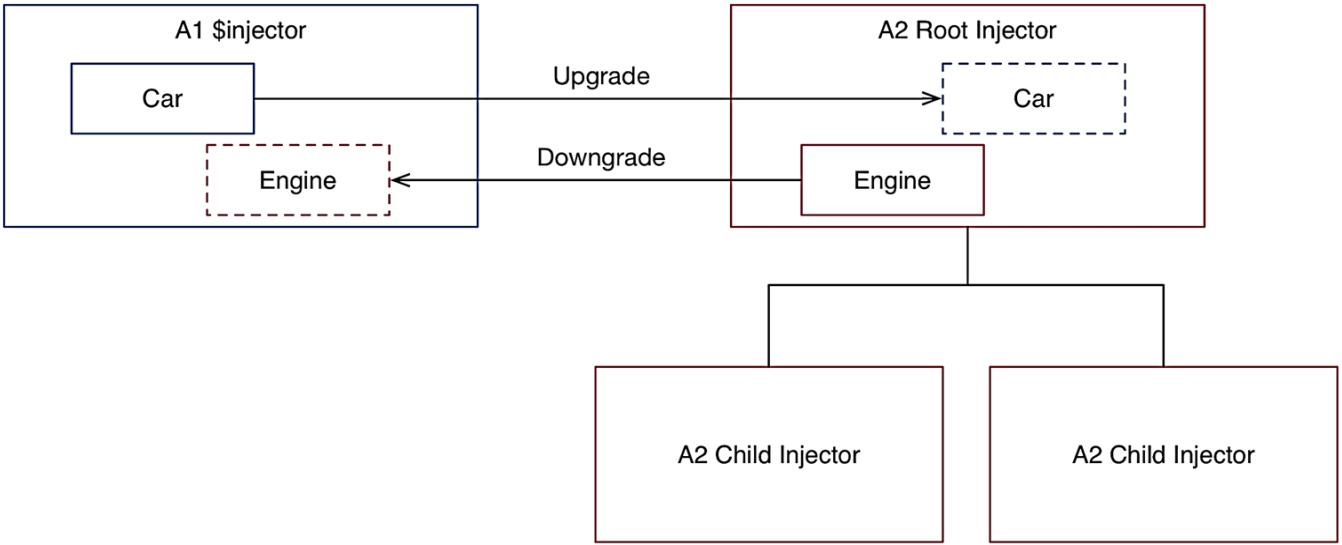
- The primary tool provided by the upgrade module is called the *UpgradeAdapter*. This is a service that can bootstrap and manage hybrid applications that support both Angular 2 and Angular 1 code.

Dependency Injection:

- Dependency injection is front and center in both Angular 1 and Angular 2, but there are some key differences between the two frameworks in how it actually works.

Using Component Directives (Contd.)

ACADGILD



Using Component Directives (Contd.)

- **Change Detection**: Change detection in Angular 1 is all about scope.$apply(). After every event that occurs, scope.$apply() gets called. This is done either automatically by the framework, or in some cases manually by our own code. It is the point in time when change detection occurs and data bindings get updated.

Using Component Directives (Contd.)

ACADGILD

- In Angular 2 things are different. While change detection still occurs after every event, no one needs to call scope.$apply() for that to happen. This is because all Angular 2 code runs inside something called the Angular zone. Angular always knows when the code finishes, so it also knows when it should kick off change detection. The code itself doesn't have to call scope.$apply() or anything like it.

- **Using Angular 2 Components from Angular 1 Code**: Once we're running a hybrid app, we can start the gradual process of upgrading code. One of the more common patterns for doing that is to use an Angular 2 component in an Angular 1 context. This could be a completely new component or one that was previously Angular 1 but has been rewritten for Angular 2.

Using Component Directives

ACADGILD

- Say we have a simple Angular 2 component that shows information about a hero:

```
import { Component } from '@angular/core';
@Component({
  selector: 'hero-detail',
  template: `
    <h2>Windstorm details!</h2>
    <div><label>id: </label>1</div>
  `
})
export class HeroDetailComponent {
}
```

- **Projecting Angular 1 Content into Angular 2 Components**: When we are using a downgraded Angular 2 component from an Angular 1 template, the need may arise to transclude some content into it. This is also possible. While there is no such thing as transclusion in Angular 2, there is a very similar concept called content projection. The UpgradeAdapter is able to make these two features interoperate.

- Angular 2 components that support content projection make use of an <ng-content> tag within them. Here's an example of such a component:

Projecting Angular 1 Content into v2 Component (Contd.)

ACADGILD

```
import { Component, Input } from '@angular/core';
import { Hero } from '../hero';
@Component({
  selector: 'hero-detail',
  template: `
    <h2>{{hero.name}}</h2>
    <div>
      <ng-content></ng-content>
    </div>
  `
})
export class HeroDetailComponent {
  @Input() hero: Hero;
}
```

Projecting Angular 1 Content into v2 Component (Contd.)

ACADGILD

- When using the component from Angular 1, we can supply contents for it. Just like they would be transcluded in Angular 1, they get projected to the location of the <ng-content> tag in Angular 2:

```
<div ng-controller="MainController as mainCtrl">
  <hero-detail [hero]="mainCtrl.hero">
    <!-- Everything here will get projected -->
    <p>{{mainCtrl.hero.description}}</p>
  </hero-detail>
</div>
```

Change Detection

ACADGILD

What's ***Change Detection***?

- The ***basic task*** of change detection is to take the internal state of a program and make it somehow visible to the user interface. This state can be any kind of objects, arrays, primitives, … just any kind of JavaScript data structures.

- This state might end up as paragraphs, forms, links or buttons in the user interface and specifically on the web, it's the Document Object Model (DOM). So basically we take data structures as input and generate DOM output to display it to the user. We call this process rendering.

Change Detection (Contd.)

ACADGILD

- However, it gets trickier when a change happens at runtime. Some time later when the DOM has already been rendered;
- How do we figure out what has changed in our model,

and where do we need to update the DOM?

- Accessing the DOM tree is always expensive, so not only do we need to find out where updates are needed, but we also want to keep that access as tiny as possible.
- This can be tackled in many different ways. One way, for instance, is simply making a http request and re-rendering the whole page. Another approach is the concept of diffing the DOM of the new state with the previous state and only render the difference, which is what ReactJS is doing with Virtual DOM.

What Causes Change?

ACADGILD

- Now that we know what change detection is all about, we might wonder, when exactly can such a change happen? When does Angular know that it has to update the view? Well, let's take a look at the following code:

```
@Component({
  template: `<h1>{{firstname}} {{lastname}}</h1>
    <button (click)="changeName()">Change name</button>`
})
class MyApp {
  firstname:string = 'Avnesh';
  lastname:string = 'Shakya';
  changeName() {
    this.firstname = 'Piyush';
    this.lastname = 'Bagheria';
  }
}
```

Change Detection Strategies

ACADGILD

- **What causes change?** The component above simply displays two **properties** and **provides a method** to change them ***when the button in the template is clicked***. The moment this particular button is clicked is the moment when application state has changed, because it changes the properties of the component. That's the moment we want to update the view.

Here's another one:

```
@Component()
class ContactsApp implements OnInit{
  contacts:Contact[] = [];
  constructor(private http: Http) {}
  ngOnInit() {
    this.http.get('/contacts')
      .map(res => res.json())
      .subscribe(contacts => this.contacts = contacts);
  }
}
```

Change Detection Strategies (Contd.)

- This component holds a list of ***contacts*** and when it initializes, it performs a http request. Once this request comes back, the list gets updated. Again, at this point, our application state has changed so we will want to update the view.
- Basically application state change can be caused by three things:
  - **Events** - click, submit, …
  - **XHR** - Fetching data from a remote server
  - **Timers** - setTimeout(), setInterval()
- They are all ***asynchronous*** which brings us to the conclusion that, basically whenever some asynchronous operation has been performed, our application state might have changed. This is when someone needs to tell Angular to update the view.

Change Detection Strategies (Contd.)

ACADGILD

Who notifies Angular?

- We now know what causes application state change. But what is it that tells Angular, that at this particular moment, the view has to be updated?
- Angular allows us to use native APIs directly. There are no interceptor methods we have to call so Angular gets notified to update the DOM. Is that pure magic?
- As you know that **Zones** take care of this. In fact, Angular comes with its own zone called **NgZone**.
- The short version is, that somewhere in Angular's source code, there's this thing called **ApplicationRef**, which listens to **NgZones onTurnDone** event. Whenever this event is fired, it executes a tick() function which essentially performs change detection.

Change Detection Strategies (Contd.)

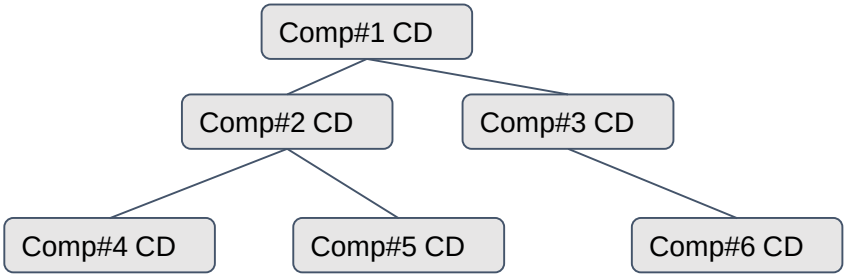
ACADGILD

```
class ApplicationRef {
  changeDetectorRefs:ChangeDetectorRef[] = [];
  constructor(private zone: NgZone) {
    this.zone.onTurnDone
      .subscribe(() => this.zone.run(() => this.tick()));
  }
  tick() {
    this.changeDetectorRefs
      .forEach((ref) => ref.detectChanges());
  }
}
```

Bypassing Components

ACADGILD

- We now know when change detection is ***triggered***, but how is it performed?
- Well, the first thing we need to notice is that, in Angular 2, ***each component has its own change detector***.



Bypassing Components (Contd.)

- This is a significant fact, since this allows us to control, for each component individually, how and when change detection is performed! More on that later.

- Let's assume that somewhere in our component tree an event is fired, maybe a button has been clicked. What happens next? We just learned that zones execute the given handler and notify Angular when the turn is done, which eventually causes Angular to perform change detection.

Bypassing Components (Contd.)

ACADGILD

- Since each component has its own change detector, and an Angular application consists of a component tree, the logical result is that we're having a change detector tree too. This tree can also be viewed as a directed graph where data always flows from top to bottom.

- The reason why data flows from top to bottom, is because change detection is also always performed from top to bottom for every single component, every single time, starting from the root component. This is awesome, as unidirectional data flow is more predictable than cycles. We always know where the data we use in our views comes from, because it can only result from its component.

Bypassing Components (Contd.)

ACADGILD

- Another **interesting observation** is that change detection gets stable after a ***single pass***. Meaning that, if one of our components causes any additional side effects after the first run during change detection, Angular will throw an error.

Performance (Contd.)

ACADGILD

- By default, even if we have to check every single component every single time an event happens, Angular is very fast. It can perform ***hundreds of thousands of checks*** within a couple of milliseconds. This is mainly due to the fact that Angular generates VM friendly code.
- What does that mean? Well, when we said that each component has ***its own change detector***, it's not like there's this single generic thing in Angular that takes care of change detection for each individual component.
- The reason for that is, that it has to be written in a dynamic way, so it can check every component no matter what its model structure looks like. VMs don't like this sort of dynamic code, because they can't optimize it. It's considered polymorphic as the shape of the objects isn't always the same.

Performance (Contd.)

ACADGILD

Angular creates change detector classes at runtime for each component, which are monomorphic, because they know exactly what the shape of the component's model is. VMs can perfectly optimize this code, which makes it very fast to execute. The good thing is that we don't have to care about that too much, because Angular does it automatically.

Smarter Change Detection:

- Again, Angular has to check every component every single time an event happens because… well, maybe the application state has changed. But wouldn't it be great if we could tell Angular to only run change detection for the parts of the application that changed their state?
- ***Yes it would, and in fact we can!*** It turns out there are data structures that give us some guarantees of when something has changed or not - **Immutables** and **Observables**. If we happen to use these structures or types, and we tell Angular about it, change detection can be much much faster. Okay cool, but how so?

Performance (Contd.)

ACADGILD

Understanding Mutability:

- In order to understand why and how e.g. immutable data structures can help, we need to understand what mutability means. Assume we have the following component:

```
@Component({
  template: '<v-card [vData]="vData"></v-card>'
})

class VCardApp {
  constructor() {
    this.vData = {
      name: 'Avnesh Shakya',
      email: 'avnesh.shakya@gmail.com'
    }
  }
  changeData() {
     this.vData.name = 'Pascal Precht';
  }
}
```

Performance (Contd.)

ACADGILD

- In the last example, we saw that the important part is that changeData() ***mutates*** vData by changing its name property. Eventhough that property is going to be changed, the vData reference itself stays the same.
- ***What happens when change detection is performed***, assuming that some event causes changeData() to be executed? First, vData.name gets changed, and then it's passed to <v-card>. <v-card>'s change detector now checks if the given vData is still the same as before, and yes, it is. **The reference hasn't changed**. However, the name property has changed, so Angular will perform change detection for that object nonetheless.
- ***Because objects are mutable by default in JavaScript (except for primitives)***, Angular has to be conservative and run change detection every single time for every component when an event happens.
- This is where ***immutable data structures*** come into play.

Performance (Contd.)

ACADGILD

Immutable Objects:

- Immutable objects give us the guarantee that objects can't change. That is, if we use immutable objects and we want to make a change on such an object, we'll always get a new reference with that change, as the original object is immutable.
- This pseudo code demonstrates this:

```
var vData = someAPIForImmutables.create({
            name: 'Pascal Precht'
        });
var vData2 = vData.set('name', 'Christoph Burgdorf');
vData === vData2 // false
```

Performance (Contd.)

ACADGILD

- So in last example, **someAPIForImmutables** can be any API we want to use for immutable data structures. However, as we can see, we can't simply change the name property. We'll get a new object with that particular change and this object has a new reference. Or in short: If there's a change, we get a new reference.

Reducing the number of checks:

- Angular can skip entire change detection subtrees when input properties don't change. We just learned that a "change" means "new reference". If we use immutable objects in our Angular app, all we need to do is tell Angular that a component can skip change detection, if its input hasn't changed.
- Let's see how that works by taking a look at <v-card>:

Performance (Contd.)

ACADGILD

```
@Component({
  template: `
    <h2>{{vData.name}}</h2>
    <span>{{vData.email}}</span>
`
})
class VCardCmp {
  @Input() vData;
}
```

- As we can see, VCardCmp only depends on its input properties. We can tell Angular to skip change detection for this component's subtree if none of its inputs changed by setting the change detection strategy to OnPush like this:

Performance (Contd.)

ACADGILD

```
@Component({
  template: `<h2>{{vData.name}}</h2>
    <span>{{vData.email}}</span>`,
  changeDetection: ChangeDetectionStrategy.OnPush
})
class VCardCmp {
  @Input() vData;
}
```

- Now imagine a bigger component tree. We can skip entire subtrees when immutable objects are used and Angular is informed accordingly.

Performance (Contd.)

ACADGILD

Observables:

- Unlike immutable objects, they don't give us new references when a change is made. Instead, they fire events we can subscribe to in order to react to them.
- So, if we use Observables and we want to use OnPush to skip change detector subtrees, but the reference of these objects will never change, how do we deal with that? It turns out Angular has a very smart way to enable paths in the component tree to be checked for certain events, which is exactly what we need in that case.

Performance (Contd.)

ACADGILD

```
@Component({
  template: '{{counter}}',
  changeDetection: ChangeDetectionStrategy.OnPush
})
class CartBadgeCmp {
  @Input() addItemStream:Observable<any>;
  counter = 0;
  ngOnInit() {
    this.addItemStream.subscribe(() => {
      this.counter++; // application state changed
    })
  }
}
```

Performance (Contd.)

ACADGILD

- Let's say we build an e-commerce application with a shopping cart. Whenever a user puts a product into the shopping cart, we want a little counter to show up in our UI, so the user can see the amount of products in the cart.
- CartBadgeCmp does exactly that. It has a counter and an input property addItemStream, which is a stream of events that gets fired, whenever a product is added to the shopping cart.

THANK YOU

For more details contact us at:

Email us at - support@acadgild.com