# Angular Technical Interview

Written By Eamon O'Tuathail

http://www.clipcode.net

One step often needed when starting a software engineering project is completing the development team. Some client engineering personnel may already be in place, but often additional developers will need to be hired/selected. We believe a small team of experienced developers will always out-perform a larger team of less experienced developers, so picking really good team members is a critical step in getting a new project off the ground. To help with this, Clipcode can provide an experienced software architect to act as a technical interviewer and carry out detailed technical interviews with suitable candidates.

This can be provided as a standalone service, or as part of Clipcode's Angular Application Kickstart, where we provide client development teams with an experienced software architect to work on-site with them during the initial stage of an Angular 5 or Angular 6 project, with the aim of helping out in any practical way to ensure the successful launch of the project.

The following is a collection of sample questions demonstrating the technical level of knowledge concerning Angular we will be expecting during our interviews. (Since this document will be placed on our website, these will not actually be the questions we ask in a real interview - we have other questions for that).

- o -

**Let's start with bootstrapping. When you use Angular CLI to create a new project, it generates code to bootstrap a single component called AppComponent which becomes the root for all sub-components. However, Angular is well capable of bootstrapping multiple components (having multiple independent root components, each with their own tree of sub-components). What steps are needed in order to bootstrap two components?**

Create a new component using:

```
ng g component second
```

Note the selector used in second.component.ts (Angular CLI sets it to 'app-second' and we can change it to anything we want). Use this selector in index.html as an additional root component:

```
<body>
  <app-root></app-root>
  <p>some text goes here</p>
  <app-second></app-second>
</body>
```

Edit the bootstrap array in app.module.ts. Angular CLI will already have added `SecondComponent` to the declarations array (assuming `--skip-import` was not used), and now we need to add it to the bootstrap array, like so:

```
@NgModule({
  declarations: [
    AppComponent,
    SecondComponent
  ],
  ..
  bootstrap: [AppComponent, SecondComponent]
})
export class AppModule { }
```

Remember, components, directives and pipes go in `declarations`. Services go in `providers`. The `imports` array is for imported `NgModule`s. The exports array is a list of exported directives, pipes and NgModules. The `entryComponents` entry (not often used) is for dynamically loaded components. The `schemas` entry is for custom HTML tags. The optional `id` entry is an ID to be used if this module is to be registered with `getModuleFactory()`.

Angular's `NgModule`s are used for constructs that Angular needs to know about (e.g. for dependency injection or template syntax). For all other types, just use normal TypeScript `import`.

Useful Link: https://angular.io/guide/bootstrapping

**How would you create and use a lazily-loaded NgModule?**

Create an additional routed module and a routing module using this Angular CLI command:

```
ng generate module myxyzfeature --routing
```

Three things must happen to enable lazy-loading:

- The main `NgModule` should not in any way reference a type from the lazily-loaded module (e.g. using a TypeScript `import` statement). If this were to occur, the referenced `NgModule` will be eagerly loaded.
- The lazily-loaded-routing-module must call `RouterModule.forChild` (and definitely not `forRoot`). This ensures the router service itself is only loaded once.
- The path for the route uses the `loadChildren` parameter (and not `children`, which is used for child route definitions that are not lazily loaded) and passes in a string with the file name and module class name where the `NgModule` is to be located. The `#` character separates them.

```
{
  path: 'my-path',
  loadChildren: 'app/myxyzfeature.module#MyXyzFeatureModule',
},
```

Useful Link: https://angular.io/guide/lazy-loading-ngmodules

**There are three ways the Angular Router can load modules – the first two are eagerly and lazily. What is the third way? How would you use it? What are your usage recommendations for each of the three module loading techniques?**

The third way is preloading. This is a variant of lazily loading but gives you more fine-grained control over how and when modules are loaded. Eager is good for modules you definitely need. Lazy is good for seldom-used modules or large modules that are not always needed. Preloading lets you specify – via a `preloadingStrategy` and perhaps a `CanLoad` guard, which of the lazily loaded modules should be loaded and when.

So if your code has this:

```
@NgModule({
  imports: [RouterModule.forRoot(routes, {
    preloadingStrategy: PreloadAllModules
  })],
  exports: [RouterModule]
})
```

then all optional modules will actually be preloaded. However, as the app evolves and perhaps gets larger, `CanLoad` guards can selectively decide which of the optional modules are to be loaded early.

- Link: https://angular.io/guide/router#preloading-background-loading-of-feature-areas

**Angular supports multiple router-outlets at the same time – either nested or not. Explain how to achieve both and what impact they have on the URL (the browser session still only has a single URL).**

When multiple router-outlets are used in the same hosting HTML file, then the primary outlet is un-named, the other(s) require a name to be used. Here we have one called secondary:

```
<router-outlet></router-outlet>
<router-outlet name="secondary"></router-outlet>
```

The routes in the routes array intended for the named outlets must use that name:

```
{
    path: 'notepad',
    component: ScribblerComponent,
    outlet: 'secondary'
},
```

To display content in the named outlet, a `RouterLink` needs to be added that activates that path:

```
<a [routerLink]="[{ outlets: { secondary: ['notepad'] } }]">Show my
notepad</a>
```

When it comes time to remove the secondary routed outlets, call `router.navigate` with the outlet name and pass in the null value:

```
this.router.navigate([{ outlets: { secondary: null }}]);
```

When secondary outlets are active, the URL will display extra data within parentheses:

```
https://www.clipcode.net/demo/myapp(secondary:notepad)
```

A different approach is to have multiple router-outlets, one at the root and the other inside a view which is the target of a route. The route table is configured using the `children` parameter, which reflects the desired nesting hierarchy:

```
const MyRoutes: Routes = [
  {
    path: 'my-component', component: MyComponent,
    children: [
      {
        path: '', component: MyistComponent,
        children: [
          {
            path: ':id', component: MyInfoComponent
          } ] } ] };
```

**Angular CLI is a wonderful tool to create new Angular projects and new constructs (e.g. modules, components, services) within those projects. Explain the steps needed to use Angular CLI to create a new project that uses a different version of an important package (e.g. TypeScript) compared to what Angular CLI automatically adds to package.json. What is the role of package-lock.json?**

By default, `ng new <PROJECT_NAME>` creates a new Angular project with a package.json file and runs `npm install` which downloads packages listed in package.json. It also creates a package-lock.json file:

- https://docs.npmjs.com/files/package-locks

which *"describes an exact, and more importantly reproducible node_modules tree. Once it's present, any future installation will base its work off this file, instead of recalculating dependency versions off package.json."*

If we wish to edit package.json before npm install, then we need to run `ng new` with the `--skip-install` option:

```
ng new demo --skip-install
```

This creates a new Angular project with the files but does not run npm install and no package-lock.json exists. After this, we can manually edit the created package.json:

```
"devDependencies": {
    ..
    "typescript": "^2.6.2"
  }
```

Then we run npm install manually.

**Discuss Angular's dependency injection and TypeScript interfaces. Is there anything specific to TypeScript interfaces that an Angular application developer planning to use them with Angular's dependency injection should be aware of?**

JavaScript does not support interfaces. When TypeScript code that use interfaces gets transpiled to JavaScript, the TypeScript interface disappear. Hence TypeScript interfaces cannot be used as the provider token in Angular dependency injection.

The easy way around this is to define an injection token - e.g. `FRIENDLY_CONFIG` - and in the constructor of the injectable, use the `@Inject` metadata - e.g. `@Inject(FRIENDLY_CONFIG)`:

```
interface Friendly { sayHi();}

class SpecialistService implements Friendly {
  sayHi(){console.log('hello from SpecialistService!');}
}

export const FRIENDLY_CONFIG =
     new InjectionToken<Friendly>('my-friendly-instance');

@Injectable()
class SimpleProvider {
  constructor(@Inject(FRIENDLY_CONFIG)  private ssvr : Friendly){}
  ..
}
```

This needs to be registered with an injector either with a `useValue`:

```
export const myfriend = new SpecialistService();

@Component({
  ..
  providers: [
    SimpleProvider,
    {provide: FRIENDLY_CONFIG, useValue: myfriend}
  ]
})
export class FourthComponent implements OnInit { .. }
```

.. or a `useFactory` parameter:

```
function friendlyFactory(): Friendly { return myfriend; }

@Component({
  ..
  providers: [
    SimpleProvider,
    {provide: FRIENDLY_CONFIG, useFactory: friendlyFactory}
  ]
})
export class FourthComponent implements OnInit { .. }
```

**Angular's dependency injection allows you three choices when deciding on the cardinality of the injected instances:**

- **firstly, inject zero or one instance,**
- **secondly, inject exactly one instance, or**
- **thirdly inject zero, one or more instances.**

**How is this achieved?**

Short answer: use `@Optional`, default, `multi:true`.

Long answer: If the instance is optional, use `@Optional()` in the injectable's constructor:

```
constructor( @Optional() first: FirstType, ..) { .. }
```

If a single instance is required, simply define a parameter with no metadata:

```
constructor( .., second : SecondType, .. ){ .. }
```

If zero or more instances are to be made available, in the provider array use `multi:true`:

```
 {provide: THIRD_CONFIG, useValue: ThirdInstance1, multi: true},
```

.. and in the constructor of the injectable accept an array:

```
constructor( .., @Inject(THIRD_CONFIG)  third:ThirdType[] ) { .. }
```

The following is the full example:

```
import { Component, OnInit, Optional, Inject, InjectionToken, Host }
                                                    from '@angular/core';
class FirstType { firstField : string; }
class SecondType {secondField : string;}

export class ThirdType {
  constructor(private thirdField : string){}
  sayGoodbye(){ console.log('Goodbye - ' + this.thirdField);}
}

export const THIRD_CONFIG = new InjectionToken<ThirdType>('ThirdType');
export const ThirdInstance1 = new ThirdType('alpha');
export const ThirdInstance2 = new ThirdType('beta');

@Component({
  selector: 'app-fifth',
  templateUrl: './fifth.component.html',
  styleUrls: ['./fifth.component.css'],
  providers: [SecondType,
    {provide: THIRD_CONFIG, useValue: ThirdInstance1, multi: true},
    {provide: THIRD_CONFIG, useValue: ThirdInstance2, multi:true}]
})
export class FifthComponent {
  constructor( @Optional() first: FirstType, second : SecondType,
                          @Inject(THIRD_CONFIG)  third:ThirdType[] ) {
    if (third == null) {
      console.log('null discovered');
    } else {
```

```
      console.log(third.length.toString());
      if (third.length > 0)
        third[0].sayGoodbye();
      if (third.length > 1)
        third[1].sayGoodbye();
      }
    }
  }
```

Note for `multi:true` provider configurations, all the providers are from the same injector. In the above case, this will be be component itself. So if we added an additional provider in the NgModule:

```
@NgModule({
  ..
  providers: [
    {provide: THIRD_CONFIG, useValue: ThirdInstance3, multi:true}],
  ..
})
export class AppModule { }
```

this will have no effect, since the providers from the component injector will satisfy the DI request directly.

**Angular developers should keep up to date with new features of the latest Angular release. What version of TypeScript does Angular 5.2 support?**

Angular 5.2 (and Angular 6.0) supports TypeScript 2.6. Specifically, package.json has this entry:

```
  "devDependencies": {
    ..
    "typescript": "2.6.x",
    ..
  }
```

**What is the purpose of Angular 5.2's new `getCurrencySymbol` exported function?**

The Angular Common package support locale data for internationalization support. The `getCurrencySymbol` exported function (new to Angular 5.2) takes in an ISO 4217 currency code (e.g. USD) and returns the currency symbol (e.g. $).

The actual currency data is generated via CLDR:

- https://github.com/angular/angular/blob/db55e86e91fce79a49950bfef0d00eb24315f057/packages/common/src/i18n/currencies.ts

The changelog is the best way to keep up to date with Angular updates:

- https://github.com/angular/angular/blob/master/CHANGELOG.md

**Angular developers should be aware of the evolving standards upon which Angular is built. Explain the organizational structure behind how the HTML standard is evolving. HTML 5.2 introduces the new `WindowOrWorkerGlobalScope` mixin – what is it?**

Two organizations–W3C (https://www.w3.org) & WHATWG (https://whatwg.org)– are involved in progressing the HTML standard. It is essentially the same standard both are writing. They cooperate and the result of their work is to be found in the same set of main browsers - Chrome, Firefox, Edge, Safari – along with many smaller browsers.

WHATWG uses the idea of living standards, that are frequently updated:

> *"The WHATWG standards are described as Living Standards. This means that they are standards that are continuously updated as they receive feedback, either from web developers, browser vendors, tool vendors, or indeed any other interested party. It also means that new features get added to them over time, at a rate intended to keep the standard a little ahead of the implementations, but not so far ahead that the implementations give up." (https://whatwg.org/faq)*

W3C used numbered versions for its standards and have just recently released HTML 5.2:

- https://www.w3.org/TR/html52

There is merit in both approaches to evolving the standard – web developers like fast standard enhancements, yet many (especially larger) enterprises demand precise version numbering so that they can rely on certain features being available to many users. It is best to think of what W3C produces as snapshots of what WHATWG produces (they are not exactly the same, but very similar).

`WindowOrWorkerGlobalScope` is described here:

- https://www.w3.org/TR/html52/webappapis.html#windoworworkerglobalscope-mixin

It is defined as:

```
interface WindowOrWorkerGlobalScope {
  [Replaceable] readonly attribute USVString origin;
  DOMString btoa(DOMString btoa);
  DOMString atob(DOMString atob);
  long setTimeout(..);
  void clearTimeout(optional long handle = 0);
  long setInterval(..);
  void clearInterval(optional long handle = 0);
  Promise<ImageBitmap> createImageBitmap(ImageBitmapSource image);
  Promise<ImageBitmap> createImageBitmap(..);
};
```

It is implemented by both `Window` and `WorkerGlobalScope`:

```
Window implements WindowOrWorkerGlobalScope;
WorkerGlobalScope implements WindowOrWorkerGlobalScope;
```

**RxJS (http://reactivex.io) is a key underlying technology for Angular. Explain the purpose of `Subject` in RxJS.**

A subject is both an observer and an observable. Often the items it received are processed in some way and then passed on.

It is defined here (note this is the new location in RxJS v6 after source tree reorg):

- https://github.com/ReactiveX/rxjs/blob/master/src/internal/Subject.ts

as:

```
export class Subject<T> extends Observable<T> implements ISubscription {  }

  [rxSubscriberSymbol]() {
    return new SubjectSubscriber(this);
  }
  ..
  static create: Function = <T>(destination: Observer<T>,
            source: Observable<T>): AnonymousSubject<T> => {
              return new AnonymousSubject<T>(destination, source);
  }
  lift<R>(operator: Operator<T, R>): Observable<R> {..}
  next(value?: T) {..}
  error(err: any) {..}
  complete() {..}
  unsubscribe() {..}
  ...
}
```

For more details, see:

- http://reactivex.io/documentation/subject.html
- http://xgrommx.github.io/rx-book/content/subjects/index.html

**What does the '*' in '*ngIf do?**

It means "use an ng-template" and is very important for the correct operation of `ngIf`, `ngFor` and similar. For details, see:

- https://angular.io/guide/structural-directives#asterisk

This explains that a line such as:

```
<div *ngIf="hero" class="name">{{hero.name}}</div>
```

becomes:

```
<ng-template [ngIf]="hero">
  <div class="name">{{hero.name}}</div>
</ng-template>
```

**The Angular Renderer API is used to create and interact with a tree of elements (DIV, P, B, TABLE, etc). However, the Renderer API does not itself list what those elements are or any of their security characteristics. So what are they and what information can we access about how to use them? (Hint: "looking at the WHATWG or W3C sites for the HTML 5 specs" is a good start, but not the full answer we are looking for – we are looking for something from inside Angular).**

The SecurityContext enum:

- https://angular.io/api/core/SecurityContext

is defined as:

```
enum SecurityContext {
  NONE: 0
  HTML: 1
  STYLE: 2
  SCRIPT: 3
  URL: 4
  RESOURCE_URL: 5
}
```

and is used to describe the security characteristics of elements.

The schema details from here has info about security context etc:

- https://github.com/angular/angular/blob/master/packages/compiler/src/schema

**What are the best practices for Angular-related security?**

Quoted from:

- https://angular.io/guide/security

*Best Practices*

*Keep current with the latest Angular library releases. We regularly update the Angular libraries, and these updates may fix security defects discovered in previous versions. Check the Angular change log for security-related updates.*

*Don't modify your copy of Angular. Private, customized versions of Angular tend to fall behind the current version and may not include important security fixes and enhancements. Instead, share your Angular improvements with the community and make a pull request.*

*Avoid Angular APIs marked in the documentation as "Security Risk." For more information, see the Trusting safe values section of this page.*

**What is the different between a zone and a web worker? Explain how both could be used by Angular applications. Advise on design considerations on when to pick either approach for your next Angular app.**

Zones and web workers are complimentary techniques which can be used separately or together. Zones are provided by the Zone.js project:

- https://github.com/angular/zone.js

Web workers are part of the standard web platform.

A web worker is a background thread. It has no direct access to the real DOM. However, through a custom polyfill approach, DOM calls in a web worker could be redirected to sending messages over a message bus. A zone subdivides a single thread (either the main browser UI thread, or a web worker). A zone monkey patches certain constructs so that code appears to run in a conceptual isolated zone, and timers registered in that zone have their callbacks in the same zone, and it is noted when the zone event queue is empty (Angular used this as the basis of initiating change detection). Code in a zone running in the main browser UI thread can access the real DOM, whereas code in a zone running in a web worker cannot.

For many Angular application developers, who do not do anything special to interact with zones or web workers, their code runs inside the default Angular NgZone and does not use web workers at all.

More advanced Angular application developers can take steps to execute some code outside the default zone (using NgZone.runOutsideAngular). This avoids excessive change detection activations. They can also use platform-webworker (instead of platform-browser) so that their application runs in a web worker and DOM calls are sent over the message broker to the main UI thread for actual display.

Zones are good when there are limited workloads to be performed, which preferably can be divided into chunks (in between processing each chunk, the browser can process UI events in a timely manner). Care should be taken not to have excessive CPU usage on the main browser UI thread, as this interferes with user interaction.

Today most phones, tablets and even low-end PCs have multiple cores. Having eight cores on a device is common. A zone always runs in a single thread (either the main browser UI thread or a web worker) so it will not fully benefit from multiple cores. Regardless of whether they use zones or not, single-threaded browser apps only use one core (other apps and the OS can use the other cores, so they are not totally wasted). To use multiple cores, a browser app needs to use web workers. So for heavy-duty Angular applications with substantial workloads, it now makes sense to invest the extra effort to use them, via web workers.