Angular 8 Technical Interview

https://www.knowist.ac

The following is a collection of sample questions demonstrating the technical level of knowledge concerning Angular 8 we will be expecting during our interviews.

- o -

Let's start with the most highly anticipated feature of Angular 8 – Ivy. In a sentence, what is it and why should we be interested in it?

Ivy is a powerful new view engine in Angular 8 that is much faster than and produces smaller output compared to the existing view engine, mainly due to much better tree shaking. Kara Erickson of the Angular Team explains it well here:

- https://youtu.be/dIxknqPOWms?t=1817

How would you pass a parameter to the Angular Compiler? Give an example.

The tsconfig.json file can have an "angularCompilerOptions" section that contains parameters intended for the Angular Compiler. Here is an example from:

- https://next.angular.io/guide/ivy

showing use of the `--enable-ivy` command line parameter to `ng new` to create a `angularCompilerOptions` passing in the `enableIvy` switch so that the upcoming Angular 8's Render3 ("Ivy") view engine is used (rather than the default Render2 view engine):

```
"compilerOptions": { ... },
"angularCompilerOptions": {
  "enableIvy": true,
  "allowEmptyCodegenFiles": true,
```

For more details of `angularCompilerOptions`, see here:

- https://angular.io/guide/aot-compiler#angular-compiler-options

Now we'll move on to 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.knowist.ac/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": "~3.1.1"
```

```
    }
```

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 8 support?

Angular 8 supports TypeScript 3.3. Specifically, its [package.json](#) has this entry:

```
  "devDependencies": {
    ..
    "typescript": "~3.3.3333",
    ..
  }
```

What is the purpose of Angular `getCurrencySymbol` exported function?

The Angular Common package support locale data for internationalization support. The `getCurrencySymbol` exported function 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/master/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 are working on HTML 5.3:

- https://www.w3.org/TR/html53/

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/html53/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 difference 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.

Angular Elements and the CDK project are both useful for building components – what is the difference between them?

CDK is the Component Dev Kit.

- https://blog.angular.io/a-component-dev-kit-for-angular-9f06e3b4b3b4

When the Angular Material 2 project was evolving, there was much common functionality among the material design–based components (accessibility, i18n, layout), and so this shared code became the CDK, which is also useful for custom component developers outside the Material team. Source code is here:

- https://github.com/angular/material2/tree/master/src/cdk

Documented here:

- https://material.angular.io/cdk/categories

The result of using the CDK is a component useable from other Angular projects. Angular Elements is for web components and the result of using that is a component callable from everywhere.

- https://en.wikipedia.org/wiki/Web_Components

Imagine an enterprise trying to make a custom component available to client developers from their customers – and these might be using React, jQuery, Angular, some other framework or even straight DOM for their development. Web components is a way of delivering cross-framework components and Angular Elements is a way of building such web components using Angular:

- https://github.com/angular/angular/tree/master/packages/elements

What is the difference between the CDK and Angular DevKit?

They are quite different – the CDK is for constructing Angular components, as just discussed, whereas Angular DevKit is a general set of tools and libraries aimed at Angular developers.

It is located here:

- https://github.com/angular/angular-cli/tree/master/packages/angular_devkit

It contains general tooling needed for developing with Angular.

Schematics is probably the most important part of the new Angular DevKit. It is nicely explained here:

- https://www.npmjs.com/package/@angular-devkit/schematics

    " Schematics are generators that transform an existing filesystem. They can create files, refactor existing files, or move files around. What distinguishes Schematics from other generators, such as Yeoman or Yarn Create, is that schematics are purely descriptive; no changes are applied to the actual filesystem until everything is ready to be committed. There is no side effect, by design, in Schematics.

Really good Angular developers should be really good TypeScript developers. What is the purpose of the ! operator at the end of a TypeScript expression? (note: we are not asking about ! at the beginning of an expression – that is the familiar negation operator).

It is the TypeScript non-null assertion operator. As explained here:

https://www.typescriptlang.org/docs/handbook/release-notes/typescript-2-0.html

    "A new ! post-fix expression operator may be used to assert that its operand is non-null and non-undefined in contexts where the type checker is unable to conclude that fact. Specifically, the operation x! produces a value of the type of x with null and undefined excluded." ([TypeScript release notes](#))

It is used extensively in the Angular source tree – here is one sample usage from:
- https://github.com/angular/angular/blob/master/packages/core/src/render3/jit/directive.ts

as follows:

```
export function directiveMetadata( .. ){
..
return {
    ..
    typeSourceSpan: null !,
    ..
  };
}
```

Angular application developers should have some – even if only high level – awareness of the Angular framework source tree. As measured by the size (in KB) of the src directory inside each Angular framework package, which is the largest package and which is the smallest? Hint: despite what one might think from https://angular.io/api, the largest is not the Core package)

Largest is the Compiler package – whose src sub-directory contains just under 1 MB of code.

Smallest is the platform-webworker-dynamic package – whose src sub-directory contains a mere 1.5 KB.

Note: this is kind of a trick question [but it is fair – let us explain]. A junior Angular application developer who has never explored the Angular source tree will not even be aware that there exists a Compiler package – it is not listed in https://angular.io/api. The API into the Compiler package is actually the Compiler-CLI package, which should really be called Compiler-CLI-And-API, because that is what it is. It is both a CLI, that can be used directly (ngc and i18n), and an API, that can be used by other packages. An example of such API usage is Angular's Bazel package which uses it to call the Compiler from a Bazel build. If we examine:

- https://github.com/angular/angular/blob/master/packages/bazel/src/ngc-wrapped/index.ts

we see this example code:

```
import * as ng from '@angular/compiler-cli';
..
const ngHost =
    ng.createCompilerHost({options: compilerOpts, tsHost: bazelHost});
```

In contrast to the junior developer, a senior Angular application developer (the type this interview is intended to find), will at times have explored some of the Angular source tree and many code paths there fairly quickly lead somewhere inside the Compiler package, so they will be aware of it and know it is very large.

What is the Angular Language Service and how would you us it?

Here is a brief description:

> *"The Angular Language Service is a way to get completions, errors, hints, and navigation inside your Angular templates whether they are external in an HTML file or embedded in annotations/decorators in a string. The Angular Language Service autodetects that you are opening an Angular file, reads your tsconfig.json file, finds all the templates you have in your application, and then provides language services for any templates that you open." https://angular.io/guide/language-service*

To use it, we first install into your editor of choice. For Visual Studio Code, add as described in the above ink. Then we make some sample modification to a *.ts file – such as in app.component.ts we define the `DemoPerson` class **1** and use it in the `AppComponent` class **2**:

```
class DemoPerson {                    1
  name : string;
  age : number;
}

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})
export class AppComponent {
  samplePerson : DemoPerson;          2
}
```

When you examine the corresponding app.component.html file, and start typing `sam` in an interpolation expression, we see:

When we select the `samplePerson` property and try to access one of its two members, we see this, which is really nice:

What is a service worker and how would you create one using Angular?

A service worker is a script that handles caching for a "sometimes connected" web application. Since all web applications can be considered "sometimes connected" - either because of technical error (the network will go down from time to time), or working as planned (imagine a user with a mobile device on a train going through a tunnel – it is expected the Internet connection will drop), then all web applications can benefit from supporting service workers.

As always, a good place to start exploring a new technology is the specification:

● https://w3c.github.io/ServiceWorker/

Angular describes service workers here (and following pages):

● https://next.angular.io/guide/service-worker-getting-started

You can create a new Angular project auto-enabled with service worker functionality simply by adding this switch to ng new:

```
ng new sample --service-worker
```

The result of this extra switch is as follows. It adds this to package.json:

```
"@angular/service-worker": "^8.0.0",
```

and this to .angular-cli.json:

```
"apps": [
    {
      ..
      "serviceWorker": true
    }
```

and it adds a new ngsw-config.json file:

```
{
  "index": "/index.html",
  "assetGroups": [{
    "name": "app",
    "installMode": "prefetch",
    "resources": {
      "files": [
        "/favicon.ico",
        "/index.html"
      ],
      "versionedFiles": [
        "/*.bundle.css",
        "/*.bundle.js",
        "/*.chunk.js"
      ]
    }
  }, {
    "name": "assets",
    "installMode": "lazy",
    "updateMode": "prefetch",
    "resources": {
      "files": [
        "/assets/**"
      ]
    }
  }]
}
```

and lastly it adds this to app.module.ts:

```
import { ServiceWorkerModule } from '@angular/service-worker';
..
@NgModule({
  ..
  imports: [
    ..
    ServiceWorkerModule.register('/ngsw-worker.js',
                                  { enabled: environment.production })
  ],
})
export class AppModule { }
```