The Ultimate Guide to Angular Evolution

How Each Angular Version Impacts Efficiency, DX, UX and App Performance

HOUSE OF ANGULAR

created in cooperation with

angular.love

Table of Contents

Preface

This ebook is a combination of Angular roadmap and changelog, supplemented with clear explanations and use cases of all relevant changes in recent years. It brings together all the new features in one place and puts them in a broader context of Angular's evolution. Regardless of whether you are interested in learning about a specific functionality or want to stay up to date with all the changes and their impact on the framework, this text has got you covered.

Introduction

Angular is gaining momentum we've never seen before. Since 2021, we've been getting a lot of new features on a regular basis, and the pace of change is not slowing down. This is a positive for Angular apps, but it also means you need to dedicate an amount of time every year to keep track of these changes. It's crucial to stay up to date with Angular's changes and new features in order to provide your users with a quality, high-performing application and in-house developers with a good developer experience.

Regularly updating the Angular version will make your application easier to maintain thanks to features and tools that make it easier to develop, test, and improve your application. This translates directly into several benefits for your business, including improved app performance, better security, increased developer efficiency, and better user experience.

This ebook will serve as your guide to previous Angular versions, the changes and new features they brought. As a bonus it provides expert insights and predictions of what the framework will look like in the future.

We will be focusing on versions 14 and up. This is because the earlier period of Angular development was marked by significant internal changes (related to the migration of the compiler and rendering engine to Ivy), which, at the same time, meant much less new functionality and changes to the framework's API. To visualize this, we can try to divide the lifespan of Angular so far into three phases:

	1st Phase	2nd Phase	3rd Phase
Angular major version	2–9	9–13	13+
Period	2016–2019	2019–2021	2021–now*
Summary	Framework foundation	Migration to IVY	Post migration-to-IVY boom

*Now - the time of publication, Q1 2024

Angular along with other frameworks like React and Vue, popularized sin-gle-page applications and built a strong community around them. The first versions of Angular were very thoroughly battle-tested. During that period the limitations of the legacy build and rendering pipeline called "View Engine" were learned. After that Angular core developers decided to completely redesign it and gave it a new code name called "Ivy". Complete migration took a few years to finally abandon the previous solution.

Ivy engine introduced a number of new opportunities and brought us into the 3rd Phase, a kind of "boom" of new functionalities and long-term shifts in Angular mental models. In the following pages, we will look at all of these in detail.

How this book is organized

The technical content is grouped by major release versions of Angular (starting with version 14, up to version 17, and beyond). Sometimes, however, functionalities were introduced between major releases or introduced gradually and improved over several versions. While we wanted to follow the chronological order as closely as possible, there may be some deviations that do not disrupt the broader context of changes to the framework.

Our approach to presenting this content is systematic and reader-friendly, ensuring that you can both grasp the technicalities and see the broader picture of each feature's impact. We structured all functionalities into three distinct sections:

Challenge - This section outlines the specific problem or need that the new feature addresses. Understanding the challenge provides context and highlights the significance of the feature in real-world scenarios.

Solution - Here, we delve into the technical details of the feature. We describe how it works, its implementation, and any variations or configurations that are relevant.

Benefits - This part showcases the advantages of using the feature. We focus on how it contributes to business values.

To further enhance your reading and learning experience, each feature is categorized using labels. That way, you have the option of reading this ebook cover-to-cover to get a full picture of Angular's evolution or jump to specific sections that are most relevant to your immediate needs. The labels and structured format make it easy to find and focus on the features that are most pertinent to your current projects or interests.

The four labels are:

Performance

Performance enhancements in frontend development are crucial for both the end-user experience and the efficiency of development processes. These enhancements focus on optimized JavaScript execution, efficient resource loading, and minimized browser reflow and repaint, resulting in quicker page loads and smoother web element interactions. Additionally, these improvements can streamline the development workflow and CI/CD pipeline, enabling quicker builds, more efficient testing, and faster deployment cycles. We use this label to mark such beneficial changes in performance.

Dev Experience

Improving developer experience can increase productivity and code quality. When developers have better tools, processes, and documentation, they can implement features more quickly, reliably and in a more maintainable way. They are more motivated to work on your project and stay in your team for a longer time. This label

marks changes that have a positive impact on the developer's experience.

UX

Focusing on user experience is crucial, because it leads to products that are more intuitive, accessible and enjoyable to use. This can increase user engagement, satisfaction and loyalty, which are important factors for the success of any software application. This label marks changes that improve user experience in Angular projects.

Efficiency

By increasing the speed at which new features and fixes are delivered, companies can respond more quickly to market changes and user feedback, leading to a competitive edge and improved customer satisfaction. This label marks changes that speed up the developer's work and increase his/her efficiency.

We hope this book serves as a valuable resource in your journey with Angular, whether you're a seasoned developer or just starting out. Happy reading, and happy coding!

Typed Forms

Standalone API

Angular v14

release date: **06.2022**

Before version 14, a significant goal was accomplished with the final deprecation of the legacy View Engine. This change, along with aligning all internal tools with Ivy, not only simplified but often enabled the introduction of innovations in Angular. It also led to easier framework maintenance, reduced codebase complexity, and smaller bundle sizes.

This was the first wave of improvements made possible by Ivy. The best examples are the Standalone API (in developer preview), Typed Forms and better developer experience related to debugging applications, both in the CLI and in the browser.

The introduction of standalone components streamlines the development process, reducing the need for additional configurations. Enhanced type safety in forms ensures error-resistant applications, which is especially beneficial for complex use cases. Features like streamlined page title accessibility improve user experience and SEO, while extended developer diagnostics offer actionable performance insights. With support for the latest TypeScript and ECMAScript standards, Angular 14 provides developers with a more powerful, efficient and flexible toolset for building advanced web applications.

Standalone API (developer preview)

Performance **Dev Experience** **Efficiency**

Challenge:

Although ECMAScript has a native module system supported in all modern browsers, Angular delivers its own module system, which is based on NgModules.

NgModule is a structure that describes how to create an injector and how to compile a component's template at runtime. It includes definitions of components, directives, pipes, and service providers that will be added to the application dependency injectors. The goals of NgModules are to organize the application and extend it with capabilities from external libraries.

This solution was met with some criticism from the beginning, because it was considered too complicated and illegible. The complex connections between modules and their providers, unclear dependencies between components, and unclear NullInjectorErrors were some of the reasons a simplified alternative needed to be provided.

Solution:

In version 14 (developer preview), the Angular Team made NgModules optional with full backward compatibility. Creating components, directives and pipes as standalone is possible by setting the "standalone" flag to true in their decorators.

```
@Component({
  selector: 'footer',
  template: '<ng-content></ng-content>',
  standalone: true,
})
export class FooterComponent {}
```

Standalone components can be imported by another standalone component or module or used within a routing declaration.

```
//in module
@NgModule({
  imports: [FooterComponent],
})
export class Module {}

//in component
@Component({
  selector: main,
  template: '<ng-content></ng-content>',
  standalone: true,
imports: [FooterComponent ]
})
export class Component {}

//routing
export const ADMIN_ROUTES: Route[] = [
  {path: 'footer', component: FooterComponent}
];
```

This transition makes components, directives and pipes self-contained. NgModule is no longer the smallest building reusable block. This results in many positive outcomes, including:

• a component no longer needs to be defined in its own NgModule and can be reused independently,

• reading components is much easier, as the reader doesn't have to track a component's module to understand its dependencies,

• tracking implicit dependencies on NgModules context is very costly for tools and makes it difficult to optimize generated code. With the standalone API the tools can be optimized,

• the API for dynamic loading and rendering components is simpler, since we no longer have to care about a component's module when using ViewContainer-Ref.createComponent(...).

Angular's main ApplicationModule is no exception. It also became optional, making it possible to bootstrap application directly with a standalone component using the bootstrapComponent function:

```
import { Component, bootstrapComponent } from '@angular/core';

@Component({
  selector: 'my-app',
  standalone: true,
  template: '<h1>Hi there!</h1>',
})
export class AppComponent {}

bootstrapComponent(AppComponent);
```

How does it affect the routing? The new API allows to lazy load standalone paths directly, with no need for NgModule:

```
RouterModule.forRoot([
 {
   path: '/path/to/standalone/component',
   loadComponent: () => import('./default-standalone.cmp')
 }
]);
```

There is also no need for NgModule while testing standalone components. This is how the test case can look like

```
const fixture = TestBed.createStandaloneComponent(MyComponent, {...});
```

At this point, in developer preview mode, the change is not production-ready. It has some problems to overcome. (Spoiler: Angular overcomes them in Angular version 15). Still, it presents a major mental model change that turns standalone components into basic building blocks of an application.

The Angular Core Team also made sure that everything was compatible with the existing module-based ecosystem, meaning all existing libraries should work as-is.

Benefits:

This revolution **boosts the developer experience** by:

- reducing boilerplate,
- making reading component code (especially dependencies) easier,
- removing the whole concept of an extra module system based on NgModules,
- improving compilation time, and
- lowering the entry barrier for novice developers.

Reduced bundle size and Tree-Shaking (thanks to function-based API) also improves the performance. More details and benefits are described in chapter "Angular 15", where Standalone API becomes Stable.

Expert Opinion:

Standalone APIs will make the authoring and building of Angular apps much simpler, especially for beginners to the Angular framework. The concept of Angular modules had been complex to explain and many developers who wanted to learn Angular were confused. Thankfully Standalone APIs will change that.

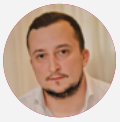
~ Aristeidis Bampakos
Google Developer Expert

From my perspective, the introduction of the Standalone API has significantly simplified the learning curve for Angular. In the past, learners had to grapple with understanding the complex NgModule relationships, including figuring out what to export, where to import, and dealing with the intricacies of eager and lazy-loaded routing. Standalone API, however, empowers developers to concentrate solely on the component they are working on, without the need to concern themselves with the intricacies of the NgModule. The extension of this approach to Pipes and Directives is like the icing on the cake.

Furthermore, Angular CLI provides support for creating projects with a focus on the standalone component-based approach, which is a valuable feature. There are also several open-source libraries available to facilitate the migration of applications from NgModule-based architecture to a standalone component-based one, making this transition smoother and more accessible.

~ Balram Chavan
Google Developer Expert

Standalone APIs introduced an important shift towards simplifying application development in Angular. They are the basis for many powerful features that have been introduced recently.



~ Marko Stanimirović
Google Developer Expert

Typed forms

Dev Experience

Challenge:

Before Angular version 14, many APIs related to Reactive Forms (FormGroup, FormControl, FormArray, etc.) included the usage of "any" type. This resulted in poor type safety, bad support for coding tools, codebase inconsistency and problematic refactoring.

Solution:

In response to the challenges mentioned above, the Angular Team introduced Typed Forms. Existing reactive forms were extended to include generic types to enforce type safety for form controls, groups, and arrays. By applying types, you can catch potential errors at compile time, making your codebase more robust and maintainable.

Example of reactive control with a generic type:

```
const nameFormControl = new FormControl<string>('');
// type of nameFormControl.value is string | null
```

The reason control values are nullable is because of the control.reset() method. If you don't pass an argument, it will change the value to null. However, it's possible to change this behavior by setting the new flag called 'nonNullable':

```
const nameFormControl = new FormControl<string>('', { nonNullable: true });
// type of nameFormControl.value is string

nameFormControl.reset();
// reset method will change value to empty string
```

Things get more interesting when we inspect the behavior of complex controls, like FormGroup:

```
const myForm = new FormGroup({
  name: new FormControl('', { nonNullable: true }),
});
// type of myForm.value.name is string | undefined
// type of myForm.getRawValue().name is string
```

The possibly undefined type comes from the fact that name control might be disabled (and not included in form value object).

For the use cases where we don't know all control keys beforehand, like when controls are added dynamically, the Angular Team added the new FormRecord class.

```
const myForm = new FormRecord<FormControl<string>>({ });
myForm.addControl('foo', new FormControl<string>('bar', { nonNullable: true
}));
// type of myForm.value is  Partial<{[key: string]: string}>
```

Migrating to Angular 14 will not break any existing untyped forms, because all occurrences of forms classes will be automatically replaced by their untyped versions:

```
const login = new UntypedFormGroup({
   email: new UntypedFormControl(''),
   password: new UntypedFormControl(''),
});
```

You can incrementally migrate to Typed Forms by removing the "Untyped" prefix in your application.

Benefits:

Leveraging TypedForms in Angular:

- **Leads to more readable, maintainable, and reliable code:** this feature enhances type safety and development efficiency by providing compile-time checking and improved IDE support for autocompletion and error detection. Discrepancies in data types or structures are caught early, and refactoring becomes safer and more straightforward.

• Facilitates cleaner, more concise code and streamline validation processes, aligning well with Angular's design philosophy for a more cohesive development experience.

Inject function

Dev Experience | Efficiency

Challenge:

So far, when it comes to dependency injection, we have been limited to injections through the constructor. This brings some limitations associated with reusability (as we could not reuse the constructor definition), testing, and a general separation of concerns. For example, classes required a knowledge of how to create their dependencies instead of focusing purely on their main responsibilities.

Solution:

The 'inject' utility function allows us to retrieve an instance of dependency from the Angular Dependency Injection System outside of a class constructor. This function has to be called in an injection context, that is one of the following:

- a constructor of the component, directive, pipe, injectable service or NgModule,
- an initializer for fields of such classes,
- a factory function (in a 'useFactory' object of a provider or an injectable),
- an InjectionToken's factory, or
- a function within a stack frame that is run in an injection context.

Let's take a look at an example of usages inside a component:

```
@Component({...})
class HomeComponent {
  private readonly dependencyA: DependencyA;
  private readonly dependencyB: DependencyB = inject(DependencyB);

  constructor() {
    this.dependencyA = inject(DependencyA);
  }
}
```

When you want to verify if you're in an injection context, you can use a helper function called assertInInjectionContext:

```
function getService(): FooService {
  assertInInjectionContext(getService);
  return inject(FooService);
}
```

This feature allows us to create many reusable DI-dependent functions like the following:

```
function getProductId(): Observable<string> {
  return inject(ActivatedRoute).paramMap.pipe(
    map((params) => params.get('productId'))
  );
}


@Component({...})
class ProductDetails  {
   private readonly productId$ = getProductId();
  ...
}
```

Benefits:

The inject function in Angular:

- **Provides flexibility:** allows developers to access service instances and other dependencies outside of class constructors, making code more modular and testable.
- **Streamlines the process of dependency retrieval:** promotes cleaner and more maintainable code by abstracting the complexity of dependency management and injection.

CDK Dialog and Menu

Dev Experience | Efficiency | UX

Challenge:

Before the introduction of the CDK, styling certain components available in Angular Material was difficult due to their ready-made design.In such a situation, it was necessary to overwrite the styles of a given component, which could be problematic.

Solution:

Thanks to the CDK, it is possible to create unstyled dialogs and menus and customize them by ourselves.

Open dialog is called by an open method with a component or with a TemplateRef representing the dialog content and returns a DialogRef instance.

```
const dialogRef = dialog.open(DialogComponent, {
  height: '300px',
  width: '500px',
  panelClass: 'empty-dialog',
});
```

cdkMenu from CdkMenuModule provides directives to create custom menu interactions based on the WAI ARIA specification. It's possible to create your own design or use ready-made classes from directives to make it easier to add custom styles.

A typical menu consists directives:

- cdkMenuTriggerFor - trigger element to open ng-template with menu
- cdkMenu - create menu content after click on the trigger
- cdkMenuItem - create and add item to menu

```
<button [cdkMenuTriggerFor]="menu">Open menu</button>

<ng-template #menu>
  <div cdkMenu>
    <button cdkMenuItem>Item 1</button>
    <button cdkMenuItem>Item 2</button>
    <button cdkMenuItem>Item 3</button>
  </div>
</ng-template>
```

Benefits:

Angular CDK:

- **Provides a set of tools** to build feature-packed and high-quality Angular components.
- **Simplifies common pattern and behavior implementation.**

Setting the page title

Dev Experience | Efficiency

Challenge:

Angular provides a Title service that allows modifying the title of a current HTML document. To use it, you have to inject it. It is completely independent from routing, so any linking to the data in the routing configuration requires complex logic using Angular Router.

Solution:

Angular 14 offers a new feature - TitleStrategy - to set unique page titles using the new Route.title property in the Angular Router. The title property takes over the title of the page after routing navigation.

```
export const routes: Routes = [
  {
    path: 'home',
    title: 'Home Page',
    loadComponent: () =>
      import('./home/home.component').then((m) => m.HomeComponent),
  }
];
```

It is also possible to provide a custom TitleStrategy to apply more complex logic behind a page title.

```
const routes: Routes = [{
  path: 'home',
  component: HomeComponent
}, {
  path: 'about',
  component: AboutComponent,
  title: 'About Me'  // <-- Page title
}];

@Injectable()

export class TemplatePageTitleStrategy extends TitleStrategy {
  constructor(private readonly title: Title) {
    super();
  }
  override updateTitle(routerState: RouterStateSnapshot) {
    const title = this.buildTitle(routerState);
    if (title !== undefined) {
      this.title.setTitle(`My App - ${title}`);
    } else {
      this.title.setTitle(`My App - Home`);
    }
  }
}

@NgModule({
  ...
  providers: [{provide: TitleStrategy,  useClass: TemplatePageTitleStrategy}]

})
export class MainModule {}
```

Benefits:

This feature is:

• **A new convenient way for manipulating a page title:** with no dependency injection and reactivity overhead.

ENVIRONMENT_INITIALIZER Injection Token

Dev Experience | Efficiency

Challenge:

It was possible to use a class with NgModule decorator to run initialization logic, i.e.:

```
@NgModule(...)
export class LazyModule {
  constructor(configService: ConfigService) {
    configService.init();
  }
}
```

But in the absence of NgModules, some Standalone API counterpart was needed.

Solution:

The Environment injector is a more generalized version of the module injector, introduced together with Standalone APIs in Angular v14. ENVIRONMENT_INITIALIZER is a token for initialization functions that will run during the construction time of an environment injector.

When we navigate to a lazy loaded route, a new environment injector is also created for that route. Then we can provide ENVIRONMENT_INITIALIZER functions that will be executed upon such navigation. Inside the initialization function, you can use the inject(...) function and perform any logic you need when the application is bootstrapped or lazy loaded content is instantiated.

Initialization functions on a standalone application bootstrap:

```
bootstrapApplication(AppComponent, {
  providers: [
    {
      provide: ENVIRONMENT_INITIALIZER,
      multi: true,
      useValue: () => inject(ConfigurationService).init(),
    },
  ],
});
```

Initialization functions on lazy loaded routes:

```
export const lazyRoutes: Routes = [
 {
  path: '',
  component: FooComponent,
  providers: [
   {
    provide: ENVIRONMENT_INITIALIZER,
    multi: true,
    useValue: () => inject(BarService).activate(),
   },
  ],
 },
];
```

Benefits:

This Injection token:

• **Increases efficiency:** we can place our initialization logic for the entire application or a specific lazily loaded route in a way that is compatible with Standalone APIs and NgModules.

• **Improves developer experience:** this part of the process is now more clear and convenient to carry out.

Binding to protected component members

Dev Experience

Challenge:

Only public members of component were accessible in its component template. It meant that every binding in the template was also a part of the component's public API. Component classes exposed too much and violated encapsulation.

Component's public API is relevant when we reference components programmatically, i.e. via ViewChild decorator:

```
@ViewChild(MyComponent) myComponent: MyComponent;
```

Solution:

In Angular v14 it is possible to use fields marked as protected in the template.

```
@Component({
  selector: 'my-component',
  template: '{{ message }}',  // Now compiles!
})
export class MyComponent {
  protected message: string = 'Hello world';
}
```

Benefits:

With this change, we get:

- **An improved developer experience:** the overall encapsulation is improved. We can now encapsulate (hide) methods and properties that we use inside a component's template but don't want to expose them anywhere else.

Angular extended diagnostics

Dev Experience **Efficiency**

Challenge:

Sometimes in the Angular codebase, there are potential anomalies that are not straightforward bugs. For example, when they don't cause a compilation error and meet all syntax requirements. At the same time, objections might occur, and they might not necessarily reflect what the programmer had in mind.

Let's take a look at the following example:

```
<component ([foo])="bar"></component>
```

This is a valid Angular example, but it is not a standard two-way data binding. Instead, it is a one-way data binding for an event (output) called "[foo]". The valid two-way data binding looks like this:

```
<component [(foo)]="bar"></component>
```

Solution:

Angular 13.2.0 brought us a new functionality called Extended Diagnostics. It's a tool built into the compilation process of Angular view templates (it's part of the compiler itself), and it doesn't require any additional infrastructure or scripts. – It simply works out of the box, and with the ng serve during the transpilation process.

Its task is to detect potential anomalies just like the one mentioned above. It serves as a kind of additional linter for angular view template syntax. We enable it inside the tsconfig file in the angularCompilerOptions section.

```
{
  "angularCompilerOptions": {
    "strictTemplates": true,
    "extendedDiagnostics": {
      "checks": {
        "invalidBananaInBox": "error"
      },
    "defaultCategory": "error"
  }
}
```

The list of currently available diagnostics is available in the official documentation: https://angular.io/extended-diagnostics. The Angular Team plans to add new diagnostics in minor releases of the framework. New diagnostics and new bugs may appear along with version upgrades, so by default, detected anomalies are returned as warnings. This can be controlled with the "angularCompilerOptions.extended-Diagnostics.defaultCategory" field in the tsconfig file.

Ideas for new diagnostics can be submitted via the github feature requests: https://github.com/angular/angular/issues/new?template=2-feature-request.yaml

Benefits:

These extended diagnostics result in:

- **Improved code security and reliability:** with no extra effort and/or cost.

ESM Application Build (experimental)

Dev Experience | Performance

Challenge:

The standard Angular bundler is considered quite slow by developers. The Angular Team tested various other approaches that can speed up the package-building process.

Solution:

Angular version 14 introduces an experimental feature that leverages the esbuild-based build system for the "ng build" command. This experimental build system compiles pure ECMAScript Modules (ESM) output.

To enable it, use the following snippet in the angular.json config file:

```
"builder": "@angular-devkit/build-angular:browser-esbuild"
```

Esbuild itself is an extremely fast JavaScript bundler and minifier written in the Go language that's designed for modern web development. It supports heavy parallel processing and might significantly outperform competitors such as Webpack. Its support in Angular framework will be developed in the upcoming releases.

Benefits:

Introducing a new esbuild-based build system is:

- **A step towards faster build-time performance:** including both initial builds and incremental builds.
- **An opening to new tools:** thanks to ESM, which includes dynamic import expressions for lazy module loading support.

TS Typescript/Node.js support

Support for Node.js v12 and Typescript older than 4.6 has been removed. Angular 14 supports TS v4.7 and targets ES2020 by default, meaning initial bundle size is reduced. Here are the new feature examples in the now-supported Typescript:

Enhanced Awaited type (available in 4.5)

Dev Experience

Challenge:

Developers who frequently work with Promises, especially with async/await syntax, sometimes want to explicitly describe the type of value returned by the resolved Promise. For regular synchronous functions, there is the ReturnType<FnType> utility, but before TypeScript 4.5, there was no counterpart for asynchronous functions.

Solution:

A new utility type called Awaited was introduced in TypeScript 4.5. It unwraps promise-like "thenables" without relying on PromiseLike, and it does it recursively.

```
// Name = string
type Name = Awaited<Promise<string>>;


// Age = number
type Age = Awaited<Promise<Promise<number>>>;


// Foo = boolean | number
type Foo = Awaited<boolean | Promise<number>>;
```

Benefits:

Extra explicit typing of async functions:

- **Positively affects the type-safeness**, and
- **Improves code readability.**

Template string types as discriminants (available in 4.5)

Dev Experience

Challenge:

TypeScript could not correctly use template string types to narrow the type in discriminated unions. In such cases, the exact type could not be inferred therefore, typing support was limited.

Solution:

With this feature, Typescript is now able to use template string literals as discriminants. The following example used to fail, but now successfully type-checks:

```
export interface Success {
  type: `${string}Success`;
  body: string;
}


export interface Error {
  type: `${string}Error`;
  message: string
}


export function handler(r: Success | Error) {
  if (r.type === "HttpSuccess") {
     const token = r.body; // correct!
  }
}
```

Benefits:

This change brings:

• **More flexibility:** when defining types and more intelligent type inference by TypeScript transpiler.

Control Flow Analysis for Destructured Discriminated Unions (Available in 4.6)

Dev Experience

Challenge:

If we had a discriminated union and we tried to destructure it, we could no longer narrow its members using discriminator property.

Solution:

Since TypeScript version 4.6, it is possible to narrow destructed discriminated object properties. The following example explains such a case:

```
type Action =
  | { kind: “NumberContents”; payload: number }
  | { kind: “StringContents”; payload: string };
function processAction(action: Action) {
  const { kind, payload } = action;
  if (kind === “NumberContents”) {
  // payloadis narrowed to number
    let num = payload * 2;
    // …
  } else if (kind === “StringContents”) {
  // payload is narrowed to string
    const str = payload.trim();
    // …
  }
}
```

Benefits:

We gain more flexibility when using and mixing discriminated unions and object destruction.

Allows code in Constructors before super() (Available in 4.6)

Dev Experience

Challenge:

In JavaScript classes, it’s necessary to invoke super() before using “this” keyword. TypeScript also has this rule, although it used to be excessively strict in ensuring it. In TypeScript, it used to be considered an error to have any code at the start of a constructor if the class containing it had any property initializers.

Solution:

From now on we can place a code inside the constructor, before calling “super()”. Note that it is still mandatory to call super() before referring to the “this” keyword.

```
class Base {
  // ...
}
class Derived extends Base {
  someProperty = true;
  constructor() {
    doSomeStuff(); // do any logic, but don't refer 'this' yet
    super();
  }
}
```

Benefits:

The unnecessary limitation has been removed and the transpiler still validates the code correctly, giving more freedom to the programmer.

Directive Composition API

Image Directive

Standalone API

Angular v15

release date: **11.2022**

Angular v15 introduces significant improvements, phasing out legacy systems, enhancing developer experience and optimizing performance. Standalone APIs are now stable, supporting simpler development practices and ensuring compatibility with core libraries. The release includes more efficient bundling with tree-shakable standalone APIs for Router and HttpClient and an ngSrc image directive for smarter data fetching and improved performance.

Standalone API (Stable)

Performance **Dev Experience** **Efficiency**

Challenge:

The problems with NgModules and the benefits of the Standalone API were presented in the Angular 14 chapter. The solution at that time had disadvantages related to not being in a stable version, as well as shortcomings regarding providers, adapting many basic modules and benefiting from abandoning modules.

Solution:

Starting with version 15, the standalone APIs drop the developer preview label and become stable. Thanks to this, we get the green light to safely utilize them in our applications, including production.

Let's explore the major changes around Angular Routing. First, we received a new type of guard – canMatch. So what's the difference between this new one, canLoad, which tells us whether we can load a route that references a lazily loaded module, and canActivate/canActivateChild, which tells us whether we can activate a child route/route? CanMatch works, in a sense, at a different, "earlier" stage and decides whether the current url can be matched against a given route. This means it can play a similar role to both canLoad (which it will eventually replace) and canActivate.* However when it returns false, subsequent routing configuration entries are processed.

This means we gain a new possibility – defining a route with the same path, but navigating to different places based on the logic implemented by the guard. This is useful, for example, when handling different user roles, or conditionally loading another version of the feature based on feature flags. Here's a usage example:

```
class CanMatchSettings implements CanMatch {
 constructor(private currentUser: User) {}

 canMatch(route: Route, segments: UrlSegment[]): boolean {
  return this.currentUser.isAdmin;
 }
}

const routes: Routes = [
 {
  path: 'settings',
  canMatch: [CanMatchSettings],
  loadComponent: () =>
   import('./admin-settings/admin-settings.component').then(
    (v) => v.AdminSettingsComponent
   ),
 },
 {
  path: 'settings',
  loadComponent: () =>
   import('./user-settings/user-settings.component').then(
    (v) => v.UserSettingsComponent
   ),
 },
];
```

The Router API has been fully adjusted to the standalone approach, so we no longer need to use the Router Module. Instead, we get a whole set of alternative APIs, which also have the advantage of being easily treeshakeable:

```
const routes: Routes = [...];

bootstrapApplication(AppComponent, {
 providers: [
  provideRouter(
   routes,
   withDebugTracing(),
   withPreloading(PreloadAllModules)
  ),
 ],
});
```

Another new feature is support for functional guards and resolvers. This means that it is possible to implement them in the form of plain functions, so we can say goodbye to the classes being the only option here. The introduction of this concept triggered many reactions among the community, both positive and negative. Some developers see this as a new direction of the framework. Personal preferences aside, one thing cannot be denied – it requires much less boilerplate. What is more, it is now very easy to create higher-order, parametrized functions returning a properly configured version of the guard based on that. A usage example, which you can easily compare with the earlier example, looks as follows:

```
const routes: Routes = [
{
   path: 'settings',
   canMatch: [() => inject(User).isAdmin],
   loadComponent: () =>
    import('./admin-settings/admin-settings.component').then(
      (v) => v.AdminSettingsComponent
    ),
 },
 {
   path: 'settings',
   loadComponent: () =>
    import('./user-settings/user-settings.component').then(
      (v) => v.UserSettingsComponent
    ),
 },
];
```

A small, but lovely, improvement also appeared in the syntax for importing lazy-loaded paths. It is possible to omit the ".then(...)" part if we use the default export in the target file.

```
@Component({
 standalone: true,
 ...
})
export default class MyComponent { ... }

{
 path: 'home',
 loadComponent: () => import('./my-component'),
}
```

Every lazy-loaded route creates a separate injector, just like every lazy loaded ngModule created its own injector. The counterpart of the "providers" array in the NgModule configuration is now moved to the route configuration:

```
{
  path: 'admin',
  providers: [AdminService],
  children: [
    {path: 'users', component: AdminUsersCmp},
    {path: 'teams', component: AdminTeamsCmp},
  ],
}
```

Providers declared in the route are available for the component declared at the same level and for all its descendants.

Similarly to Router, HttpClient has also been adapted to the module-less approach:

```
bootstrapApplication(AppComponent, {
  providers: [
    provideHttpClient(
      withXsrfConfiguration({
        cookieName: 'MY-XSRF-TOKEN',
        headerName: 'X-MY-XSRF-TOKEN',
      })
    ),
  ],
});
```

The transition to Standalone API also affected Angular interceptors:

```
bootstrapApplication(AppComponent, {
 providers: [
  provideHttpClient(
   withInterceptors([
    (request, next) => {
     console.log('Url: ', request.urlWithParams);
     return next(request);
    },
   ])
  ),
 ],
```

During migration to functional interceptors, you can still use your class-based interceptors configured using a multi-provider by adding withInterceptorsFromDi utility function as follows:

```
bootstrapApplication(AppComponent, {
 providers: [
  provideHttpClient(withInterceptorsFromDi()),
  {
   provide: HTTP_INTERCEPTORS,
   useClass: YourClassBasedHttpInterceptor,
   multi: true,
  },
 ],
});
```

If in your standalone app, you need any providers from third-party libraries that are available only inside NgModules, you can use the importProvidersFrom utility function:

```
bootstrapApplication(AppComponent, {
 providers: [importProvidersFrom(MatDialogModule)],
});
```

It is also worth mentioning that the shift towards a module-less approach simplifies the creation of dynamic components that are self-contained and no longer need their Ngmodules.

```
@Component({...})
export class MyComponent {
  constructor(private readonly viewContainerRef: ViewContainerRef) {}
  create(): void {
    this.viewContainerRef.createComponent(OtherComponent);
  }
}
```

Benefits:

The proposed module-less approach:

- **Simplifies the process of application development:** reduces the reliance on NgModules and minimizes boilerplate code.
- **Enables faster development cycles and a cleaner codebase:** provides a more direct and streamlined API for key aspects like bootstrapping, routing, and dynamic component instantiation.
- **Improves app performance:** thanks to the shift towards a providers-first approach, which enhances tree-shakability.

Expert Opinion:

The standalone API makes many things easier. Especially since you now have to provide everything you need in one component, and it does not magically come from somewhere. This makes the topic of DependencyInjection much easier to understand. This also made the lazy loading of components possible. My tip for this: Declare the component class as a default export so that you don't have to resolve the promise yourself during lazy loading. This makes the dynamic import shorter.

~ David Muellerchen
Google Developer Expert

Directive composition API

Efficiency | Dev Experience

Challenge:

One of the most wanted features in the framework was the ability to reuse directives and apply their behavior to other directives or components.

Up to this point, there have been few possibilities to partially achieve a similar result, such as the use of inheritance, where the main limitation is that only one base class can be used.

Another idea used, for example, by Angular Material, is the use of TypeScript mixins. But this forces a specific approach to the code shared this way, which heavily complicates implementation and doesn't allow for the use of Angular APIs in mixins.

Solution:

The Directive Composition API in Angular is a feature that allows directives to be applied to a different directive or a component's host element directly from within the component's TypeScript class. The only major restriction is that only standalone directives can be applied to our directives/components, which on the other hand, don't need to be standalone.

```
@Component({
 selector: 'my-component',
 templateUrl: './my-component.html',
 hostDirectives: [
  {
   directive: NgClass,
  },
  {
   directive: CdkDrag,
   inputs: ['data'],
   outputs: ['moved: dragged'],
  },
 ],
 standalone: true,
})
export class MyComponent {}
```

The above piece of code applies the NgClass and CdkDrag directives to our component. The first one doesn't expose any inputs or outputs, so you won't be able to use them in the template where our component is used. The second, on the other hand, exposes both input and output, with an alias defined for output. Therefore, the use of our component could look as follows:

```
<my-component [data]="myData" (dragged)=onDragged($event)>
</my-component>
```

But can we control the behavior of applied directives from inside the component? This is possible using the inject function, which allows us to inject the instance of the directive into the component and manipulate its properties. It looks like this:

```
@Component({
  selector: 'my-component',
  templateUrl: './my-component.html',
  hostDirectives: [
    {
      directive: NgClass,
    },
  ],
  standalone: true,
})
export class MyComponent {
  private ngClassDirective = inject(NgClass);
  someCallback(): void {
    this.ngClassDirective.ngClass = 'my-class';
  }
}
```

Benefits:

Directive Composition API:

- **Improves the developer experience:** enhances code modularity by allowing developers to encapsulate and reuse behaviors across different components and directives.
- **Leads to a cleaner and more organized codebase**
- **Makes the maintenance and updating of the application more efficient.**

• **Simplifies the implementation process:** it applies directives directly to the host element from within the TypeScript class, reducing the need for complex configurations and boilerplate code in the templates.

Image directive

Efficiency **Performance** **UX**

Challenge:

The app may take a long time to load in the browser due to the way images are loaded. This can be especially noticeable when the website contains a lot of multimedia.

Solution:

In collaboration with the Aurora team, the Angular team has introduced the NgOptimizedImage to enhance image optimization and incorporate best practices for image loading performance. It is a standalone directive designed to boost image loading efficiency. It became a stable feature in Angular version 15.

To activate iNgOptimizedImage, simply replace the image's src attribute with ngSrc.

```
<img ngSrc="cat.jpg">
```

If the LCP image is shown, a good way to prioritize its loading is to use property called "priority".

```
<img ngSrc="cat.jpg" priority>
```

Thank for that 'priority' applies three optimizations:

- Sets fetchpriority=high - gets resources in the optimal order and prioritizes the image
- Sets loading=eager - lazy-loading images
- Automatically generates a preload link element if it uses SSR.

NgOptimizedImage requires us to specify a height and width for the image or attribute 'fill' to prevent image-related layout changes. But if the 'fill' attribute is used, it's necessary to set the parent element 'position: relative/fixed/absolute'. When using CDN images, it is possible to compress images and convert them on demand to formats such

as WebP or AVIF.
The NgOptimizedImage image directive also offers other solutions to improve application performance when loading images such as:

- supporting resource hints - preloading critical assets
- possibility to preload the resource for all routes instead of manual addition of preload resource hint.

Benefits:

Using the NgOptimizedImage directive and ngSrc attribute is very simple, almost cost-free, and can significantly improve an application's performance, SEO and core web vitals.

MDC-based components

Efficiency | Dev Experience | UX

Challenge:

The current version of the Angular Material library was loosely linked to the official Material Design specification. All the styles and behaviors were reinterpreted and reimplemented to Angular style. With the arrival of Material Design Components for Web (MDC), the library became outdated It became clear that new Angular Material should be strongly and directly based on official MDC design tokens. These include values like colors, fonts and measurements.

Solution:

In Angular Material version 15, a significant migration took place. Many components are now being refactored to be based on Material Design Components for Web (MDC). Various components have been refactored, leading to changes in styles, APIs and even complete rewrites for some. Components like form-field, chips, slider and list have significant changes in their APIs to integrate with MDC. There are library-wide changes affecting component size, color, spacing, shadows and animations to improve spec-compliance and accessibility. Theming changes include updates to default typography levels and themeable density. Each component has specific changes, including style updates, element structure and API modifications.

Benefits:

These changes offer several benefits, such as:

- **Improved accessibility**
- **Better adherence to the Material Design spec**
- **Faster adoption of future Material Design versions:** due to shared infrastructure

CDK Listbox

Efficiency | Dev Experience | UX

Challenge:

Creating a typical listbox with accessibility support, keyboard events support, multiselection and correct scroll behavior, as well as satisfying WAI ARIA listbox pattern requirements is a time-consuming task.

Solution:

The newly added component to the Angular CDK library meets all the above guidelines while being a fully customizable solution.

The code example:

```
<label class="example-listbox-label" id="example-appointment-label">
  Appointment Time
</label>
<ul cdkListbox
    [cdkListboxValue]="appointment"
    [cdkListboxCompareWith]="compareDate"
    (cdkListboxValueChange)="appointment = $event.value"
    aria-labelledby="example-appointment-label"
    class="example-listbox">
  <li *ngFor="let time of slots"
      [cdkOption]="time"
      class="example-option">
    {{formatTime(time)}}
  </li>
```

Benefits:

Using a ready-to-use, safe, tested and trustworthy solution with WAI ARIA standards directly from the creators of Angular saves developers a significant amount of time.

Improved stack traces

Dev Experience

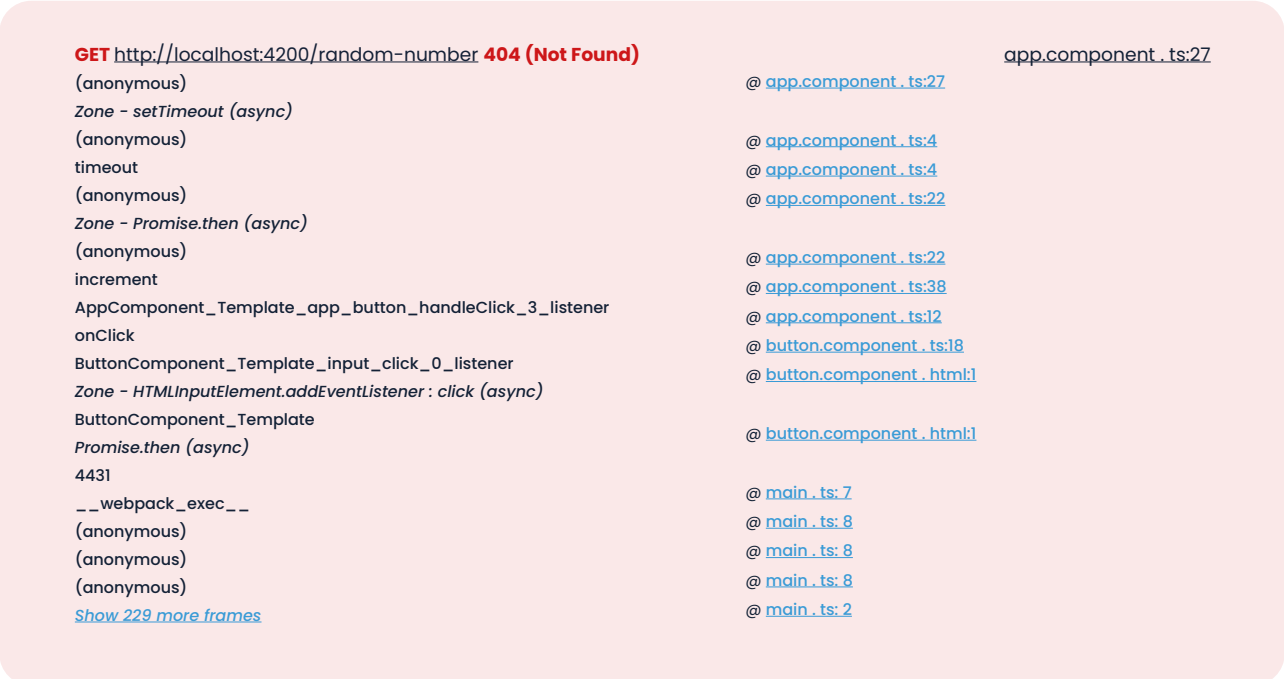
Challenge:

Up until now, the stack traces presented in CLI and browser devtools were quite obscured by external functions (i.e. from webpack or node_modules). It was hard to investigate the execution order when the trace included many lines from outside the code written by the programmer.

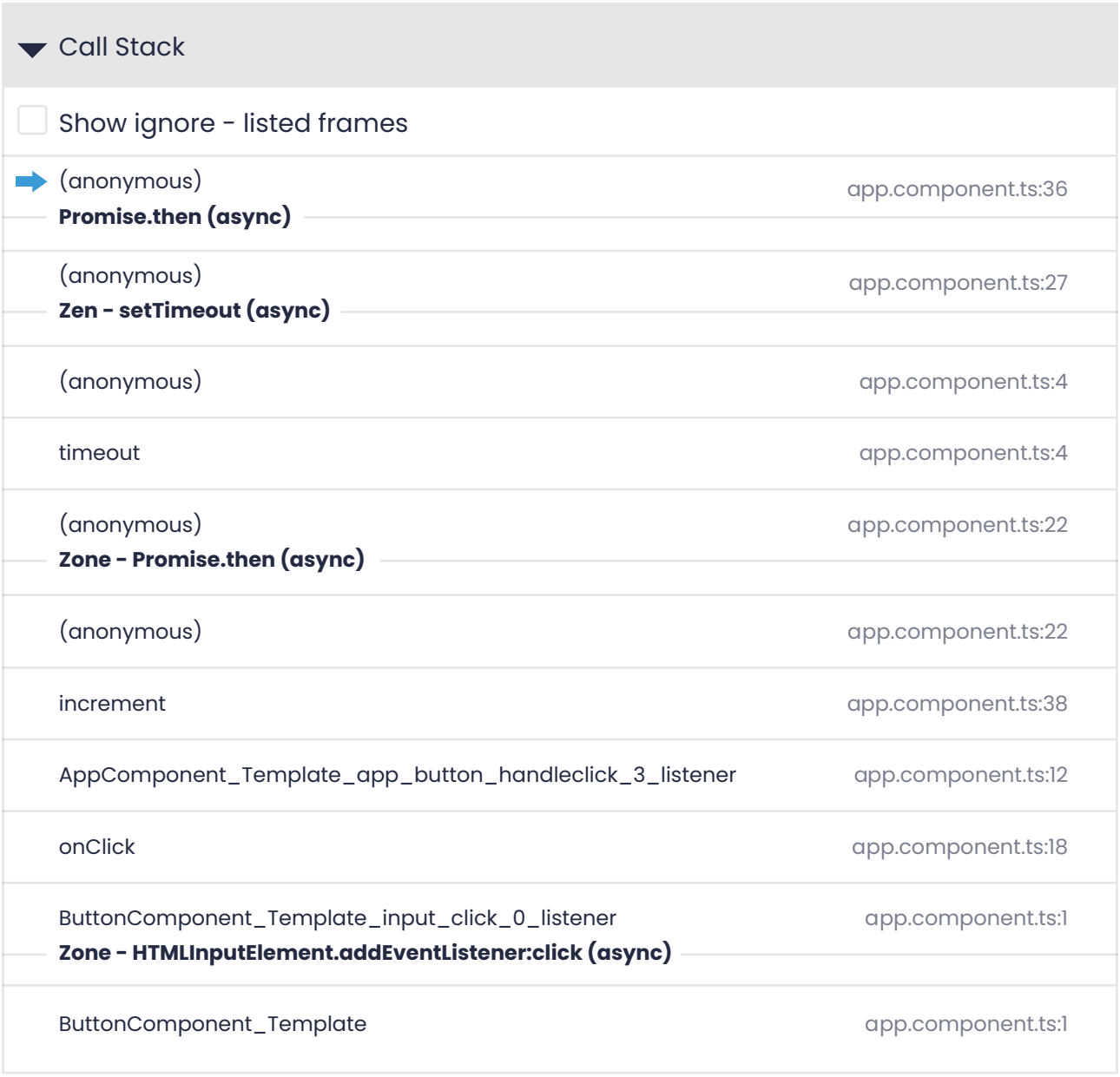
Solution:

Improvements in this area are possible thanks to the cooperation of the Angular Team and the Chrome team. The goal was to mark scripts as "external" and thus exclude them in the development tool functions(e.g. stack traces). Starting with Angular 14.1, the contents of the node_modules and webpack folders are marked in this way. It is worth mentioning that this mechanism is available to all developers, so authors of other frameworks can use it as well.

The result is a stack trace that omits scripts that are probably not in the developer's area of interest when an error is shown in the console. Also, the code that belongs to excluded scripts is skipped when you're debugging and iterating over subsequent instructions in the code.



The second new feature worth mentioning is stack trace linking for asynchronous operations. The code executed following the completion of an asynchronous action can now be properly linked with the code initiating this action (e.g. a click and call to the server). This is possible thanks to the introduction of the so-called Async Stack Tagging API mechanism in Chrome.



Benefits:

Both sync and async traces have readable form and provide more information to the programmer. Therefore these improvements:

- **Boost developer experience**
- **Ease the debugging process**

Auto-imports in language service

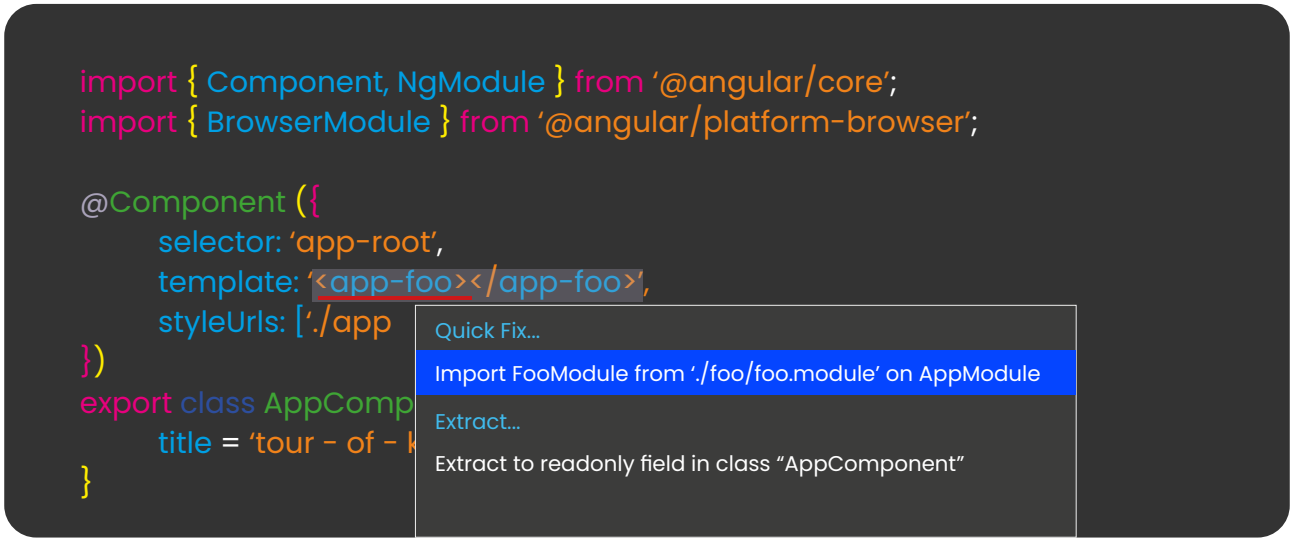
Efficiency | Dev Experience

Challenge:

With Standalone API, we frequently need to add extra imports to the Component metadata, because the only components, directives and pipes available in the template are ones we explicitly imported.

Solution:

The Angular Language Service is a tool utilized by all code editors to enhance errors, hints and navigation between Angular templates. The new DX-related improvement, a part of language service, allows us to automatically import components whose selectors were used in another component's template. This applies to both standalone and module-based components.



Benefits:

This tool is utilized by IDEs, and it significantly simplifies and speeds up the import of components, directives and pipes using keyboard shortcuts, enabling the programmer to focus on other tasks.

Typescript/Node.js support

Angular 14.2 added support for TypeSript 4.8, Node.js v18 and Angular 15.1 added support for Typescript 4.9. According to the official documentation, it includes, among other things, a number of performance improvements that can reduce build and hot reload times of TypeScript projects by up to 40%. Let's take a look at other new features worth mentioning.

The satisfies Operator (available in 4.9)

Dev Experience

Challenge:

When we define an expression, TypeScript infers the most general type for it:

```
const person = { name: 'John' };
// typeof person: { name: string }
```

In the example above, the typeof "name" property is not narrowed to const literal "John" but generalized to any string value. In some cases, we would like to ensure that the expression matches a certain type while also ensuring the most specific type at the same time.

Solution:

TypeScript version 4.9 introduced a new operator called "satisfies." It validates whether the type of an expression matches a type without changing the resulting type of that expression.

```
type Colors = 'red' | 'green' | 'blue';
// Ensure that we have exactly the keys from 'Colors'.
const myColors = {
  red: 'yes',
  green: false,
  blue: 'maybe',
  orange: false,
  // error - "orange" is not part of "Colors" union type
} satisfies Record<Colors, unknown>;
/**
 * typeof myColors is not Record<Colors, unknown>
 *  but {"red": string, "green": boolean, "blue": "string"}
 *  All the information about the 'red', 'green', and 'blue'
 *  properties are retained.
 */
const isGreen: boolean = myColors.green;
//typeof myColors.green is not unknown despite "satisfies" type check
```

This operator can be combined with "as const" to infer type from constant values and check them against a broader type at the same time:

```
const friends = [
  { name: 'John' },
  { name: 'Paul' },
] as const satisfies readonly { name: string }[];
```

In the example above, expression is type-checked with the satisfies operator, but the inherited type for friends const comes directly from the value.

Satisfies is somewhat similar to "as" operator. The difference is that with "as," the programmer is responsible for the type safety and with "satisfies", TypeScript validates the type we assert automatically.

```
type Red = 'red';
const x: Red = 'blue' as Red;
// everything ok
const y: Red = 'blue' satisfies Red;
// error: Type '"blue"' does not satisfy the expected type '"red"'
```

Benefits:

The new operator improves the TypeScript developer experience as it enforces type safety without the loss of information (i.e. via type widening). By replacing the "as" operator with "satisfies," we also reduce the reliance on untrustworthy type assertions. The code readability is also improved in some cases.

Checks For Equality on NaN (available in 4.9)

Dev Experience

Challenge:

In IEEE-754 floats standard, supported by JavaScript and TypeScript, nothing is ever equal to NaN, the"not a number" value. However, it is a common mistake to check it with someValue === NaN, instead of using the built-in Number.isNaN function.

Solution:

Typescript no longer allows direct comparisons against NaN and suggests Number.is-NaN instead:

```
function validate(someValue: number) {
  return someValue !== NaN;
  //     ~~~~~~~~~~~~~~~~~
  // error: This condition will always return 'true'.
  //        Did you mean '!Number.isNaN(someValue)'?
}
```

Benefits:

Thanks to the new restriction, we avoid an obvious error and are clearly informed about an error.

Esbuild

Non-destructive Hydration

Signals

Angular v16

release date: **05.2023**

Angular version 16 brings substantial advancements, focusing on creating a more modern and efficient framework. The introduction of signals marks the beginning of a refined change detection mechanism, while non-destructive hydration sets the stage for advanced hydration scenarios crucial for public web platforms.

Enhancements in development ease are achieved through mandatory inputs, routing parameter bindings and the new DestroyRef. Furthermore, the Angular CLI now supports creating applications with standalone components, and the experimental esbuild-based builder offers a substantial speed boost in application building.

Signals library (developer preview)

Performance **Dev Experience** **Efficiency**

Challenge:

Angular applications can face performance issues due to inefficient change detection mechanisms, resulting in an excessive number of computations to determine what has changed in the application state. Additionally, developers might struggle with a more complex mental model for understanding reactivity, data flow, RxJS and dependencies within the application, making it harder to write, maintain, and optimize their code efficiently.

Solution:

In Angular 16, we received a developer preview of a brand new library containing a primitive that represents a reactive value–the Angular signals library.

Angular's signals library introduces a way to define reactive values and establish explicit dependencies between them within an application. Unlike RxJS, observables that push changes through a stream of values, signals allow for a more direct and straightforward way to declare reactive state, compute derived values and handle side-effects, offering a different approach to managing reactivity in Angular applications.

A signal in Angular is essentially a wrapper around a value, which notifies interested consumers (e.g., components, functions) when that value changes. Signals can hold any type of value, ranging from simple primitives like numbers and strings, to more complex data structures such as objects or arrays.

Here are a few reasons behind Angular signals implementation:

- Angular can keep track of which signals are read in the view. This lets you know which components need to be refreshed due to a change in state.
- The ability to synchronously read a value that is always available.
- Reading a value does not trigger side effects.

• No glitches that would cause inconsistencies in reading the state.

• Automatic and dynamic dependency tracking, and thus no explicit subscriptions. That frees us from having to manage them to avoid memory leaks.

• The possibility to use signals outside of components, which works well with the Dependency Injection.

• The ability to write code in a declarative way.

This is how we can define signals and express dependencies between them:

```
@Component({...})
export class App {
 firstName = signal('Ash');
 lastName = signal('Ketchum');
 fullName = computed(() => `${this.firstName()} ${this.lastName()}`);
 setName(newName: string): void {
  this.firstName.set(newName);
 }
}
```

'firstName' and 'lastName' are writable signals, meaning they provide an API for updating their values directly. 'fullName' is a readonly signal that depends on the other signals and is recalculated every time any dependent signal changes its value.

In many cases, it is useful to define a side effect. This is a call to code that changes state outside its local context, such as sending an http request or synchronizing two independent data models. To create an effect we can use "effect" function:

```
effect(() => {
 console.log(`The current value of my signal is: ${mySignal()}`);
});
```

The Angular Team also provides a bridge between signals and RxJS, placed in the @angular/core/rxjs-interop library. It is possible to convert a signal to observable and an observable to a signal.

```
import { toObservable, toSignal } from '@angular/core/rxjs-interop';
@Component({...})
export class App {
  firstName = signal('Ash');
  firstName$ = toObservable(this.firstName);

  lastName$ = of('Ketchum');
  lastName = toSignal(this.lastName$, '');
}
```

The real revolution will come when the Angular Team releases signal-components. These components will work without zoneJS and will have their own change detection strategy based solely on signals. It will be possible to update DOM elements with surgical precision, without unnecessary traversal of the component tree.

It's also worth mentioning that external libraries like ngrx also implement support for signals. The NgRx store will have its signal-based counterpart called SignalStore.

```
import { signalState } from '@ngrx/signals';
const state = signalState({
  user: {
    firstName: 'John',
    lastName: 'Smith',
  },
  foo: 'bar',
  numbers: [1, 2, 3],
});
console.log(state()); // { user: { firstName: 'John', lastName: 'Smith' }, foo: 'bar' }
console.log(state.user()); // { firstName: 'John', lastName: 'Smith' }
console.log(state.user.firstName()); // 'John'
```

Benefits:

The introduction of a new experimental reactive primitive is a promise of a simplified mental model for reactivity in Angular. Its goals are to lower the learning curve, reduce the number of errors and make the whole development process intuitive and straightforward. With the future arrival of signal-based components, significant performance improvements are expected.

Expert Opinion:

I think signals will have a dramatic impact on many sides of the framework from performance to bundle size improvements. This will be especially visible when signals are integrated into the change detection mechanism. It will make the process much more efficient and performant because, unlike the current zone.js-based implementation, signals will allow Angular to know in which exact component a change occurred and update only the affected components. Besides that, the change detection mechanism will become much simpler and unified, because we will have only one predictable change detection strategy, which will also have a positive impact on the learning curve. Finally, we will be able to reduce bundle size by removing the zone.js dependency, which will not be needed for the new change detection mechanism. And this is just one of the examples. I think that we are currently seeing only the tip of the iceberg.

~ Dmytro Mezhenskyi
Google Developer Expert

The Angular signal feature is currently in its Developer Preview stage. Once this feature becomes stable, it is poised to introduce an entirely new approach to writing Angular applications. The elimination of zone.js will free developers from concerns related to change detection cycles and performance issues, allowing them to concentrate more on their business logic. This is exemplified by the transition from familiar [(ngModel)] binding syntax to the new signal and effect syntax, which requires developers to adapt and update their projects. Despite these changes, I firmly believe that this innovation holds great promise in advancing the framework to the next level.

~ Balram Chavan
Google Developer Expert

SSR Hydration (developer preview)

Performance **UX**

Challenge:

By default, Angular renders applications only in a browser. If any web crawler tries to index the page, it receives an almost empty HTML file and is unable to navigate and find searchable content. This is one of the reasons why we might want to implement Server-Side Rendering, the process of rendering our site on the server side and returning the whole HTML structure.

A server-side rendered HTML is displayed in the browser, while the Angular app bootstraps in the background, reusing the information available in server-generated HTML. The HTML then destroys DOM and replaces it with with a newly rendered and fully interactive Angular app.

Destroying and replacing an application's DOM may result in a visible UI flicker and has a negative impact on Core Web Vitals, such as FID, LCP and CLS.

Solution:

Angular introduced hydration, a new feature currently available for developer preview. We can enable it as follows:

```
import { bootstrapApplication, provideClientHydration, } from '@angular/
platform-browser';
...

bootstrapApplication(RootCmp, {
  providers: [provideClientHydration()]
});
```

It enables reusing server-side rendered HTML in the browser without destroying it. Angular matches the existing DOM elements with application structure at runtime. This results in a performance improvement in Core Web Vitals and a better SEO performance.

There is also an option to skip hydration for some components, or rather component trees, if they're not compatible with hydration. For example, manipulating DOM directly with browser APIs. Use one of the following options to enable hydration:

#1

```
<test-component ngSkipHydration />
```

#2

```
@Component({
 ...
 host: {ngSkipHydration: 'true'},
})
class TestComponent {}
```

There are also some other improvements added for SSR. These include new standalone-friendly utility functions to provide SSR capabilities to an application (provideServerRendering()) and HTTP transfer state (withTransferCache()).

If your Angular application uses SSR, you should definitely check this feature out. More details are available in the official documentation: https://angular.io/guide/hydration

Benefits:

Full hydration brings:

- **Better UX:** faster loading and interactivity, no more flickering
- **Enhanced SEO:** in comparison to destructive hydration

Expert Opinion:

The Angular team has been deeply committed to the ongoing refinement of Server-Side Rendering (SSR) capabilities. Over time, they've introduced a series of enhancements to SSR applications, aiming to provide a smoother and more efficient experience for both developers and end-users. The team's focus has been on achieving the goal of a fully hydrated application, which means that the application is not just server-rendered but also preloaded with the necessary data and components, offering a seamless and responsive user experience.

This concerted effort to improve SSR reflects the Angular team's dedication to staying at the forefront of web development technology. It's an exciting journey of innovation, and it's intriguing to anticipate how these ongoing efforts will shape the landscape of Angular in the future. The advancements in SSR hold the promise of further enhancing the performance, SEO-friendliness, and overall user experience of Angular applications, making them even more competitive and appealing in the dynamic world of web development.

~ Balram Chavan
Google Developer Expert

Vite-powered dev server

Performance **Dev Experience**

Challenge:

Previous experimental features related to the Angular building systems affected only the building process and not the development process, especially in the case of the development server with hot-reload.

Solution:

The esbuild-based build system for Angular CLI entered developer preview in version 16, with the goal of significantly speeding up the build process. Early tests showed improvements of over 72% in cold production builds. With this update, Vite is utilized as the development server, while esbuild enhances both development and production builds for faster performance. However, it's important to note that Angular CLI exclusively uses Vite as a development server. This is due to the Angular compiler's need to maintain a dependency graph between components, requiring a different compilation model.

You can enable esbuild and Vite by updating your angular.json file:

```
...
"architect": {
  "build": {
    "builder": "@angular-devkit/build-angular:browser-esbuild",
     ...
```

Benefits:

Improvements in the Angular building system bring:

- **An improved developer experience**
- Reduced time and costin heavy CI/CD processes

Please note that these changes are still in experimental mode.

Required inputs

Dev Experience

Challenge:

Until Angular 16, it was not possible to mark inputs as required, but there was a common workaround for this using a component selector:

```
@Component({
  selector: 'app-test-component[title]', // note attribute selector here
  template: '{{ title }}',
})
export class TestComponent {
  @Input()
  title!: string;
}
```

Unfortunately, this was far from ideal. The first thing was that we polluted the component selector. We always had to put all required input names to the selector, which was especially problematic during refactors. It also resulted in a malfunction of the auto-importing mechanisms in IDEs. And second, forgetting to provide the value for an input marked this way meant the error was not very accurate, as there was no match at all for such an "incomplete" selector. You can see this here:

ERROR

src/app/app.component.html:1:1 - error NG8001: 'app-test-component' is not a known element:

1.If 'app-test-component' is an Angular component, then verify that it is part of this module.

2. If 'app-test-component' is a Web Component then add 'CUSTOM_ELEMENTS_SCHEMA" to the @NgModule.schemas' of this component to suppress this message.

1.<app-test-component></app-test-component>

~~~~~~~~~~~~~~~~

src/app/app.component.ts:5:16

5 templateURL: './app.component.html',

~~~~~~~~~~~~~~~~

Error occurs in the template of component AppComponent.

Solution:

The new feature allows us to explicitly mark the input as required, either in the @Input decorator:

```
@Input({required: true}) title!: string;
```

or the @Component decorator inputs array:

```
@Component({
 ...
 inputs: [
  {name: 'title', required: true}
 ]
})
```

However, the new solution has two serious drawbacks. One of them is that it only works in AOT compilation and not in JIT. The other drawback is that this feature still suffers from the strictPropertyInitialization TypeScript compilation flag, which is enabled by default in Angular. TypeScript will raise an error for this property since it was automatically declared as non-nullable but not initialized in the constructor or inline.

ERROR

src/app/test-component/test-component.component.ts:9:3 - error TS2564: Property 'title' has no initializer and is not definitely assigned in the constructor.

9 title: string;
~ ~ ~

This means that you still need to disable this check here e.g. by explicitly marking this property with a non-null assertion operator, even though it has to be provided in the consumer template:

```
@Input({required: true}) title!: string;
```

Benefits:

The existing workarounds are replaced by a new solution with a very simple and straightforward syntax.

Input transform function

Dev Experience

Challenge:

Angular interprets all static HTML attributes as strings. For example, if boolean attributes are considered true when they are present on a DOM node,i.e. “disabled,” Angular, by default, interprets them as strings.

Solution:

Since the release of Angular 16.1, we can provide an optional transform function for the @Input decorator. This allows us to simplify the process of transforming values and discontinue the use of setters and getters that have been used for this purpose so far. Transform function:

```
function toNumber(value: string | number): number {
  return isNumber(value) ? value : parseInt(value);
}


@Component({
  selector: ‘app-foo’,
  template: ``,
  standalone: true,
})
export class FooComponent {
  @Input({ transform: toNumber }) width: number;
}
```

Usage in template:

```
<app-foo width=”100” />
<app-foo [width]=”100” />
```

Additionally, Angular offers us two built-in transform functions that we can use:

```
import { booleanAttribute, numberAttribute } from '@angular/core';

// Transforms a value (typically a string) to a boolean.
@Input({ transform: booleanAttribute }) status!: boolean;

// Transforms a value (typically a string) to a number
@Input({ transform: numberAttribute }) id!: number;
```

Benefits:

Transform functions are pure functions, so they improve readability, reusability and testability.

Router data input bindings

Dev Experience **Efficiency**

Challenge:

Obtaining router-related data inside components is not very comfortable. We have to inject an ActivatedRoute token, subscribe to its properties, manage these subscriptions to ensure no memory leaks, and so on.

Solution:

Angular version 16 brings another interesting feature related to component inputs: the ability to bind them directly to the current route variables, such as path params and query params. This eliminates the need to inject ActivatedRoute into the component in order to use router data.

With this feature, data is being bound only to the routable components present in the routing configuration, and inputs of children components used in the templates of routable routes are not affected. Route data is matched with inputs by name,or input alias name if present. This is done so there is more than one piece of data that can potentially be put as the input value.

The list below shows the precedence of data being bound to input property if the names are the same:

1. Resolved route data
2. Static data
3. Optional/matrix params
4. Path params
5. Query params

There is much more complexity if we consider “inheriting” data from places like the parent route configuration and the parent component presence.

To enable this feature, we have to use a dedicated function called with ComponentInputBinding.

Example route definition:

```
bootstrapApplication(AppComponent, {
  providers: [
    provideRouter(
      [
        {
          path: 'example/:id',
          data: {
            bar: true,
          },
          resolve: { baz: () => 'example string' },
          loadComponent: () => import('./app/todo/todo.component'),
        },
      ],
      withComponentInputBinding()
    ),
  ],
});
```

Exinput binding (input names must match route param names or their aliases):

```
@Component({
  selector: 'example-cmp',
  standalone: true,
  template: '',
})
export default class ExampleComponent {
  @Input() id!: string;
  @Input() bar!: boolean;
  @Input() baz!: string;
}
```

Benefits:

Router data input bindings simplifies the task of transmitting router data to the routed components, consequently diminishing the requirement for repetitive code and dealing with extra RxJS subscriptions.

Injectable DestroyRef and takeUntilDestroyed

Dev Experience

Challenge:

The cleanup process was a common problem Angular programmers had to deal with, especially when we wanted to implement a component/directive lifecycle with long RxJS subscriptions. The standard way to do this was to use an OnDestroy lifecycle hook to complete all relevant streams and execute other cleanup actions. However, this caused some overhead and unwanted boilerplate code. Another way to achieve this implementation was to use third-party solutions like @ngneat/until-destroy, but as opposed to built-in ones, they can introduce compatibility issues and less seamless integration.

Solution:

In the new Angular version, we receive a brand new injectable for registering cleanup callback functions that will be executed when a corresponding injector is destroyed.

```
const destroyRef = inject(DestroyRef);


// register a destroy callback
const unregisterFn = destroyRef.onDestroy(() => doSomethingOnDestroy());


// stop the destroy callback from executing if needed
unregisterFn();
```

Such an injectable is allowed to introduce yet another feature: the takeUntilDestroyed operator. This operator uses the DestroyRef injectable underneath and is a safe way to avoid memory leaks in Angular apps.

```
import { takeUntilDestroyed } from '@angular/core/rxjs-interop';


@Component({
  ...
})
export default class MyComponent {
  constructor() {
    interval(3000).pipe(takeUntilDestroyed()).subscribe(
      value => console.log(value)
    );
  }
}
```

Benefits:

The new operator is a convenient way to avoid memory leaks without dealing with OnDestroy lifecycle hooks and subscription references. DestroyRef also allows you to deal with other cleanup tasks in any way that fits your needs.

Self-closing tags

Dev Experience

Challenge:

When we don't use a content projection mechanism, we are not placing any content between tags of the Angular components we use in templates. HTML specification contains many self-closing tags for nodes without inner HTML (i.e. <img />, <input />,
), but in Angular, we always had to repeat the same name in opening and closing tags, as follows:

```
<app-your-component-name [foo]="bar">
</app-your-component-name>
```

Solution:

Self-closing tags is a highly requested feature that arrived in Angular version 16. You can use self-closing tags for your components.

This feature is optional and backward compatible, so you can continue to use the standard approach, especially when using content projection.

```
<app-your-component-name [foo]="bar">
```

Benefits:

Angular template syntax gets easier and more readable.

runInInjectionContext

Dev Experience

Challenge:

There was no convenient way to inject dependencies outside of construction time, like after user interaction or conditionally after resolving some asynchronous value.

Solution:

An “inject” function is a function that Injects a token from a current injection context. We mostly use it during the construction time of classes being instantiated by the DI system. Examples of classes include components, pipes and injectable services.

There is, however, a possibility to call “inject” at any time of the life cycle if we use the “runInInjectionContext” function. This allows us to instantiate dependency on demand after a user interaction.

```
@Injectable({
  providedIn: 'root',
})
export class MyService {
  private injector = inject(EnvironmentInjector);
  someMethod(): void {
    runInInjectionContext(this.injector, () => {
      const otherService = inject(OtherService);
    });
  }
}
```

Benefits:

We get a possibility to inject dependencies at any time, as long as we have a reference to the injector. This approach can improve application performance if we combine it with lazy loading.

Standalone API CLI support

Dev Experience | Efficiency

Challenge:

There was no way to generate a new project as a standalone from the beginning. We had to manually remove AppModule and rewrite the bootstrapping process to a standalone version with a bootstrapApplication function call. All components, directives and pipes in such a project were generated using built-in code generators in a non-standalone version by default.

Solution:

Angular version 16 introduced a new flag for CLI command with which we can generate a new Angular project.

```
ng new --standalone [name]
```

The project output with such a flag is simpler compared to the old ngModule-based version. Generators for components, directives and pipes in this project will also create standalone versions by default.

Benefits:

Kickoff of a new standalone project is made very easy, increasing overall project efficiency.

TS Typescript/Node.js support

Angular v16 brings support for TypeScript 5.0, and Angular v16.1 for Typescript 5.1. This results in further computation time, memory usage and package size optimizations, as well as many feature enhancements.

ECMAScript decorators (available in 5.0)

Dev Experience

Challenge:

Typescript supported custom experimental decorators for years. They're familiar to every Angular Developer, as we use them to define things like modules, components, injectable services, pipes, directives, inputs, outputs and injections in constructor arguments. They were implemented long before TC39 decided on a decorator's standard, and now there is a mismatch between the old TypeScript and the new ECMAScript decorator. Such mismatch can cause compatibility and integration issues for developers.

Solution:

TypeScript 5.0 supports the new official ECMAScript decorators. This results in better typing, like the ClassMethodDecoratorContext type which impacts meta information like method name, access modifier and the extra function addInitializer

This is an example of a fully typed ECMAScript decorator in Typescript 5.0:

```
function loggedMethod<This, Args extends any[], Return>(
  target: (this: This, ...args: Args) => Return,
  context: ClassMethodDecoratorContext<
    This,
    (this: This, ...args: Args) => Return
  >
) {
  const methodName = String(context.name);
  function replacementMethod(this: This, ...args: Args): Return {
    console.log(`LOG: Entering method '${methodName}'.`);
    const result = target.call(this, ...args);
    console.log(`LOG: Exiting method '${methodName}'.`);
    return result;
  }
  return replacementMethod;
}
```

There is one important difference between the old and new implementation - decorators used in the constructor parameters will not work, as the standard doesn't support the following code:

```
constructor(@Optional() public myService: MyService) {}
```

So we need to use the inject function instead:

```
myService = inject(MyService, { optional: true });
```

Benefits:

We're up to date with the official ECMAScript standard with improved typings. We no longer need to rely on the TypeScript custom implementation without its JavaScript counterpart. Instead, we can use a solution that will be directly supported by all modern browsers.

Extending multiple TS configuration files

Dev Experience

Challenge:

Typescript supported custom experimental decorators for years. They're familiar to every It is uncomfortable to compose TypeScript configuration files with a single file extension, especially in a mono-repository where multiple applications or libraries are involved. This process requires us to create a chains of extensions as follows:

```
// tsconfig1.json
{
  “compilerOptions”: {
     “strictNullChecks”: true
  }
}
// tsconfig2.json
{
“extends”: “./tsconfig1.json”,
  “compilerOptions”: {
     “noImplicitAny”: true
  }
}
// tsconfig.json
{
  “extends”:”./tsconfig2.json”,
  “files”: [“./index.ts”]
}
```

Solution:

Typescript 5.0 allows us to extend multiple tsconfig files.

```
// tsconfig1.json
{
  “compilerOptions”: {
     “strictNullChecks”: true
  }
}
// tsconfig2.json
{
  “compilerOptions”: {
     “noImplicitAny”: true
  }
}
// tsconfig.json
{
  "extends": ["./tsconfig1.json", "./tsconfig2.json"],
  "files": ["./index.ts"]
}
```

Benefits:

Extending multiple TS configuration files can simplify the configuration of your Angular workspace by streamlining the management of compiler options across multiple projects.

All enums as Union enums (available in 5.0)

Dev Experience

Challenge:

Before Typescript 5.0, the language required all constant enum members to be string/numeric literals, like 1,2, 300, “foo” or “bar,” for an enum to make it a union enum. In such a case, union enum members become types as well. That means that certain members can only have the value of an enum member. It also means that enum types themselves effectively become a union of each enum member.

Solution:

In TypeScript 5.0, all enums, even the ones with computed members, are converted into union enums. All members of all enums can be referenced as types and all enums can be successfully narrowed.

The following example with computed enum members is correctly interpreted in TypeScript 5.0 and above:

```
const prefix = 'data';

const enum Routes {
  Parts = `${prefix}/parts`,
  Invoices = `${prefix}/invoices`,
}
```

Benefits:

This feature brings more flexibility when defining types and more intelligent type inference by the TypeScript transpiler.

Unrelated Types for Getters and Setters

Dev Experience

Challenge:

In a getter/setter pair, the get type had to be a subtype of the set type, which potentially limited the ability to return more specific or transformed data types through getters.

Solution:

Since the release of TypeScript version 5.1, we can specify two completely unrelated types for a getter/setter pair.

```
interface CSSStyleRule {
  // ...
  /** Always reads as a `CSSStyleDeclaration` */
  get style(): CSSStyleDeclaration;
  /** Can only write a `string` here. */
  set style(newValue: string);
  // ...
}
```

Benefits:

The developer gains more flexibility by being able to cover edge cases that require different types.

Esbuild

Signals

New Control Flow

Deferred Loading

Angular v17

release date: **11.2023**

Angular 17 marks a pivotal update in the framework's evolution, introducing streamlined features and setting a robust foundation for future advancements in signals and template syntax. In this version, Angular also continues its journey towards enhancing performance and improving SSR.

Signals library (stable)

Performance **Dev Experience** **Efficiency**

Note:

Signals are introduced and described in the "Angular 16" chapter. Here we will focus on the changes that have occurred since then.

Challenge:

The Angular signals library, in its developer preview phase, encountered some modifications and adjustments due to feedback from the Angular community. Various Angular developers raised doubts related to mutability and the change notification system in the initial proposal.

Solution:

In Angular 17, signals became stable, so we could confidently use them in commercial applications. At the end of the developer preview, we received two significant changes.

The first change was the removal of the 'mutate' function that was previously used to mutate the value stored by the signal. The mutate function skipped comparing values because its purpose was to modify the value of the signal by design. Now, it is recommended to use only the update function.

The second change had to do with the implementation of the default function for comparing signal values.

The defaultEquals function implementation in v16 considered any two objects to be different. so that even if we returned references to the same object, all dependent signals were notified of the change.

```ts
// Angular 16 version
export function defaultEquals<T>(a: T, b: T) {
  return (a === null || typeof a !== 'object') && Object.is(a, b);
}
// Angular 17 version
export function defaultEquals<T>(a: T, b: T) {
  return Object.is(a, b);
}
```

New implementation of the defaultEquals function relies solely on Object.is(). As a result, if an object is mutated by using the update() function without changing the reference, other dependent signals will not be notified of the change. To get a signal to notify you about the change, create a new reference of the object with the updated properties (e.g., by using the spread operator) or provide your own implementation of the equals function and then specify it in the signal options.

The upgrade of the API to the stable version does not include the "effect" function, as there is still ongoing discussion about its semantics and potential improvements.

A significant change is that if all components in the application use the OnPush change detection strategy and a new signal value triggers Change Detection in a specific component template, only this component is marked as dirty and its ancestors are notified about the change. This “local” change detection can majorly affect application performance.

It is important to note that Angular 17 still does not include signal-based component implementation, even in the developer preview mode. When this implementation arrives, it is supposed to revolutionize the change detection system, because components will not use zoneJS and DOM recalculation will only happen after the values related to the signal view are changed.

Benefits:

The stabilization of the Angular signals library in Angular 17 lays solid groundwork for future advancements, particularly in moving away from zone.js and towards signal-based components. This shift is a step towards change detection system transformation, which promises enhanced performance and efficiency in application development.

Expert Opinion:

At this juncture, I hold no particularly strong opinion on the signals library. My perpetual quest revolves around enhancing both user experience (UX) and developer experience (DX). Nonetheless, I avoid excessive "hype" concerning the introduction of signals at the framework's core, opting instead for a rational assessment of the potential opportunities signals may herald in the future. A substantial benefit would arise if we could seamlessly transition away from zone.js, aligning with our overarching objective of establishing a viable route toward creating fully zoneless applications (as one of the goals).

~ Artur Androsovych
Angular Expert

Signals are becoming a mainstream technique, being introduced in the most popular frontend frameworks, the reason being that it is the perfect technique for fine-grained synchronous reactivity. For a long time, we used to handle all states in an asynchronous reactive approach using RxJS, whereas state management itself is a synchronous task, such that RxJS is not the right tool by default. This overcomplicated a few use cases and is the reason for unexpected glitching. This is a lot simpler with signals. Additionally, this fine-grained reactive primitive can, and will, be used for fine-grained and zoneless change detection in the future, when signal-based components land in Angular. This is no small change, and will rather change the way we develop reactive Angular applications completely.

~ Stefan Haas
Nx Champion

Signal inputs

Performance **Dev Experience** **Efficiency**

Challenge:

The current implementation of component inputs is not very reactive. In order to listen for changes, we use component lifecycle hooks. That means that if we want the input values to interact with the Signal API, we need to handle the logic manually.

Solution:

Angular version 17.1 enables signal inputs for components. This means we no longer use the decorator. Instead, a dedicated function defines the input:

```
@Component({
  selector: 'my-cmp',
  standalone: true,
  template: `{{ name() }}`,
})
export class MyComponent {
  name = input<string>();
}
```

This mechanism allows you to simplify the reactivity of the component, as it allows you to define computed signals and side effects for input value changes. There is no longer a need to use ngOnChanges lifecycle hook.

```
export class MyComponent {
  name = input<string>();
  helloMessage = computed(() => 'Hello ' + name());

  constructor() {
    effect(() => {
      console.log('logging: ' + this.name());
    })
  }
}
```

The Signal input API has extra capabilities compared to its standard counterpart, such as providing default values, marking input as required, setting aliases, and various transform functions.

```
export class MyComponent {
  // required input
  name = input.required<string>();

  // input with default value
  color = input('red');

  // input with alias
  disabled = input<boolean>(false, { alias: 'isDisabled' });

  // input with transform function
 age = input.required({ transform: numberAttribute });
}
```

Benefits:

Signal inputs mark a significant step towards the deeper integration of signals into the ecosystem and transforms the way we deal with component input changes.

This feature significantly enhances the developer experience and code clarity by streamlining the management of component inputs and facilitating more intuitive coding practices.

New control flow (Developer preview)

Performance **Dev Experience** **Efficiency**

Challenge:

The standard control flow (if/else, switch/case, loop in component templates), based on structural directives, has a few disadvantages. For example, the boilerplate, which is large compared to other frontend frameworks, is necessary even for a single if-else statement.

```
<div *ngIf="condition as value; else elseBlock">{{value}}</div>
<ng-template #elseBlock>Content to render when value is null.</ng-template>
```

The second issue with the standard control flow is the current directives implementation, which is strongly based on the current change detection system. Support for both zone-based and signal-based approaches in directives would greatly complicate the code with no possibility of tree shaking it.

Solution:

The introduction of a new control flow to templates is one of the biggest changes in Angular v17. It is the first step to moving away from built-in structured directives, whose current design would not work in zoneless signal-based applications.

The new syntax is based on a structure called a block. Its appearance significantly differs from what we have seen in templates. We start each block with an "@" prefix and then use a syntax very similar to the one we know from JavaScript.

The change affects the three most commonly used structured directives: ngIf, ngSwitch and ngFor. At this point, there is no planned implementation of custom blocks.

An if/else statement:

```
@if (time < 12) {
  Good morning!
} @else if (time < 17) {
  Good afternoon!
} @else {
  Good evening!
}
```

A switch/case statement:

```
@switch (fruit) {
  @case 'apple' {
    <apple-cmp />
  }
  @case 'banana' {
    <banana-cmp />
  }
  @default {
    <unknown-fruit-cmp />
  }
}
```

A loop statement:

```
<ul>
  @for (item of items; track item.id) {
    <li>{{ item.name }}</li>
  } @empty {
    <li>No items...</li>
  }
</ul>
```

The loop has received the most valuable improvements out of all the structures:

- The @empty block is available, which allows us to display the content when the list we iterate over is empty.
- It is no longer necessary to create a special trackBy function to pass it as an argument. Instead, we only need to specify the unique key of the object to be tracked.
- The @for block requires the use of the track function, which significantly optimizes the process of rendering the list and managing changes without the need for developer interference.

Since we have two ways of using the control flow, you may ask : what will happen to the directives we are familiar with? In version 17, they remain unchanged. But with the arrival of future versions and the exit of the new control flow from the developer preview, they will go into a deprecated state.

However, there is no need to worry about refactoring. The Angular Team has created a scheme that automatically migrates the control flow to the new syntax. In most cases, the scheme should handle the transition to the new syntax without a programmer's interference. To switch to the new syntax, all you need to do is execute the following command:

```
ng generate @angular/core:control-flow
```

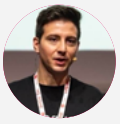
Benefits:

The new control flow in Angular offers a more succinct and readable syntax, significantly reducing boilerplate in templates. It ensures smoother integration with future signal-based change detection systems, enhancing application performance. Additionally, it lays the groundwork for advanced features such as deferred loading, contributing to optimized load times and a better overall user experience.

According to public benchmarks, operations on loops using the new built-in control flow are up to 90% faster.

Expert Opinion:

The new control flow is fabulous. The syntax is quite similar to the JavaScript syntax, and the readability has increased. This new syntax is what we are waiting for. It's also more similar to JSX (not equal), but this can reduce the gap with React, and that can help React developers jump into the Angular environment. With this new approach, we can focus on the view's logic and not on what Angular needs to show our logic correctly; that's a fantastic improvement for the developers and reduces the entry-level in the framework.



~ Luca Del Puppo
Google Developer Expert

The built-in new control flow brings several key improvements:
• No need to import NgIf, NgFor, and more in standalone components.
• Automatic enclosure of code blocks with {}. So no need for <ng-container>.
• Ergonomic and type-checking enhancements
• Improved repeater performance
• A step toward zoneless apps
These changes simplify development, reduce code verbosity, and enhance performance, making Angular even more developer-friendly.

~ Fatima Amzil
Google Developer Expert

Deferred loading (developer preview)

Performance **Dev Experience** **UX**

Challenge:

A lack of lazy loading in a web app can lead to slower page load times and increased bandwidth usage, as all content is loaded at once regardless of its immediate necessity. As long as Angular has lazy-loading modules in routing, there is no simple and convenient way to lazy load individual components, even though it's achievable with dynamic import() and ngComponentOutlet.

Solution:

Angular version 17 introduces the new lazy-loading primitive called "defer," based on the syntax introduced in the new control flow. This extremely useful mechanism allows a controlled delay in the loading of a page's selected elements. That is particularly

important because by using "defer," we can significantly reduce the initial bundle size, which improves the application's loading speed, especially for users with a slower internet connection.

To control when to defer a block's content, use predefined conditions: when and on. You can use them individually or in any combination, depending on when you want to load the content.

"When": defines a logical condition that will load the block's content when it receives a true value.

```
@defer (when condition) {
  <deferred-cmp />
}
```

An important note is that once the block's content has been asynchronously loaded, there is no way to undo that loading. If we want to hide the block's contents, we need to combine the **@defer** block with the **@if** block.

"On": allows us to use predefined triggers which initiate loading.

These predefined triggers are:

- **Idle** - This is the default trigger. The element will be loaded when the browser enters the idle state. The requestIdleCallback method is used to determine when the content will be loaded.
- **Interaction** - Content will be loaded upon user interaction, click, focus, touch, and upon input events, keydown, blur, etc.
- **Immediate** - Loading will occur immediately after the page has been rendered.
- **Timer(x)** - Content will be loaded after X amount of time.
- **Hover** - Content will be loaded when the user hovers the mouse over the area covered by the functionality, which could be the placeholder content or a passed-in-element reference.
- **Viewport** - Content will be loaded when the indicated element appears in the user's view. The Intersection Observer API is used to detect an element's visibility.

```
@defer (on interaction) {
  <deferred-cmp />
}
```

We also have the ability to combine conditions and triggers:

```
@defer (when cond; on interaction, timer(5s)) {
  <deferred-cmp />
}
```

It is worth noting that loading content, as in the case of **when**, is a one-time operation.

"Prefetch": There may be situations where we want to separate the process of fetching content from rendering it on the page. In such a case, we need the prefetch condition. It allows us to specify the moment, using the previously mentioned triggers, when the necessary dependencies will be downloaded. As a result, interaction with this content becomes much faster, resulting in a better UX.

```
@defer (on interaction; prefetch on idle) {
  <deferred-cmp />
}
```

We also have three very useful, optional blocks we can use inside the @defer block:

@placeholder - used to specify the content visible by default until the asynchronously loaded content is activated. Example:

```
@defer (when condition) {
  <deferred-cmp />
}
@placeholder (minimum 2s) {
  <span>There will be deferred content.</span>
}
```

The **"minimum"** condition allows you to specify the minimum time after which the delayed content can be loaded. In the example used above, this means that even if the condition is met immediately, the content will be swapped after 2 seconds.

@loading - the content of this block is displayed when the dependencies are being loaded. Example:

```
@defer {
  <deferred-cmp />
}
@loading (after 100ms; minimum 1s) {
  <span>Content is loading...</span>
}
```

Within this block, we can also use the minimum condition, which works the same way as within the **@placeholder**. It indicates the minimum time for which the block's content will be visible. We can also use the **"after"** condition, which indicates the amount of time it will take for the block's content to appear. If it takes less than 100ms to load, the loader will not appear. Instead, `<deferred-cmp />` will replace it immediately.

@error - represents the content rendered when the deferred loading failed for some reason. Example:

```
@defer (timeout 1s) {
  <deferred-cmp />
}
@error {
  <p>Failed to load the deferred component</p>
  <p>Error: {{ $error.message }}</p>
}
```

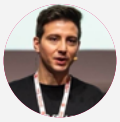
When using the defer block together with the **@error** block, it is possible to use a special **timeout** condition. This condition allows you to set a maximum loading time. If dependencies take longer than the specified time, the contents of the **@error** block will be displayed. Inside this block, the user has access to the **$error** variable, which contains information about the error that occurred during the loading process.

Benefits:

Deferred loading in Angular enhances performance by reducing initial bundle size, speeding up load times, and enabling controlled, asynchronous loading of individual components based on user interaction or other predefined conditions. This leads to a more efficient and user-friendly experience.

Expert Opinion:

Deferred loading opens a new way to reduce the bundle size of every component and introduces a new way to load only essential code needed for the current view. This feature will probably be one of the leading solutions to reduce the page size in a Server Side Render application; it helps to render only the needed parts and skip the dynamic content that isn't required on the page start-up or for the SEO.



~ Luca Del Puppo
Google Developer Expert

*As for the **deferred loading** feature, it's truly incredible! With this feature, we can lazily-load content within an Angular template. It's much more than just simple lazy loading; we can load chunks based on conditions and triggers and even prefetch in advance. This is bound to revolutionize the way we construct our Angular templates.*

~ Fatima Amzil
Google Developer Expert

Inputs Binding with NgComponentOutlet

Dev Experience **Efficiency**

Challenge:

When we used the NgComponentOutlet structural directive to create dynamic components, there was no convenient way to pass the data to them. The usual solution was to create a new injector and transfer data by a custom injection token.

Solution:

In Angular 16.2, we received a possibility to bind data to inputs of a component created with NgComponentOutlet.

```
Component({
 selector: 'my-component',
 imports: [NgComponentOutlet],
 template: `
  <ng-template
*ngComponentOutlet="dynamicComponent; inputs: dynamicComponentIn-
puts;"  />
 `
 standalone: true,
})
class FooComponent {
 readonly dynamicComponent = FooComponent;
 readonly dynamicComponentInputs  = { foo: 'Bar' }
}
```

In the child component from the example above, we have to create @Input with a 'foo' name, or alias and the data will be bound from the parent.

Benefits:

The new function simplifies the data passing process, making it more declarative. This improves code readability and developer experience.

Animation lazy loading

Performance **UX**

Challenge:

Animations are always downloaded while the application is bootstrapped, even though in most cases, animations occur while the user is interacting with an element on the page. This results in increased bundle size.

Solution:

Animation lazy loading is a new functionality in version 17 that solves this problem by introducing the ability to asynchronously load code that is associated with animations.

To start using lazy loading of animations, we need to add provideAnimationsAsync() instead of provideAnimations() to the providers of our application:

```
import { provideAnimationsAsync } from '@angular/platform-browser/animations/async';
bootstrapApplication(AppComponent, {
  providers: [provideAnimationsAsync(), provideRouter(routes)],
});
```

And that's it! Now we just need to make sure that all functions from the @angular/animations module are only imported into dynamically loaded components.

Controlling imports in our application is easy, but the process can become problematic when we do it with libraries. An example of such a library is **@angular/material**, which relies heavily on **@angular/animations**, making it very likely that the module responsible for animations will be pulled into the initial bundle.

If we want to check if the animations were downloaded asynchronously, we can build our application with the **--named-chunks** flag. Then we should see **@angular/animations** and **@angular/animations/browser** in separate bundles in the **Lazy Chunk Files** section.

build with asynchronously loaded animations

Initial Chunk Files	Names	Raw Size
chunk-IETGG730.js	-	98.56 kB
main-S3BE5YUD.js	main	77.80 kB
polyfills-W2VU2Y2Q.js	polyfills	33.23 kB
styles-YD2MB7C6.css	styles	43 bytes
	Initial Total	209.63 kB

Lazy Chunk Files	Names	Raw Size
chunk-FBVTU5M2.js	browser	62.20 kB
chunk-D75ILFX5.js	-	3.53 kB
chunk-FW06XWLR.js	animated-component	1.25 kB

build with standard loaded animations

Initial Chunk Files	Names	Raw Size
main-C6GNJMLR.js	main	133.40 kB
chunk-5AHZPM5V.js	-	102.12 kB
polyfills-W2VU2Y2Q.js	polyfills	33.23 kB
styles-YD2MB7C6.css	styles	43 bytes
	Initial Total	268.79 kB

Lazy Chunk Files	Names	Raw Size
chunk-0CFIALD3.js	animated-component	1.23 kB

Benefits:

Lazy loading of any kind of resources enhances user experience by speeding up initial page load times, saving bandwidth, prioritizing critical resources, and potentially improving website performance and SEO.

View Transitions

UX

Challenge:

There is no convenient way to implement smooth animated transitions between pages in Angular.

Solution:

Support for the View Transition API has been introduced. This is a relatively new mechanism that allows us to create smooth and interactive transition effects between different views of a web page. Thanks to this API, we can make changes to the DOM tree while an animation is running between two states.

Adding the View Transition API to our project is very simple.

First, we need to import the 'withViewTransitions' function at the bootstrap point of our application.

```
import { provideRouter, withViewTransitions } from '@angular/router';
bootstrapApplication(AppComponent, {
  providers: [provideRouter(routes, withViewTransitions())],
});
```

Now, the application immediately gains a subtle input and output effect when changing the URL. Of course, we have the ability to create custom animations. In the example below, the transition effect was extended to 2 seconds by using prepared pseudo-elements in the `styles.css` file.

```
/* Screenshot of the view of the page we are leaving */
::view-transition-old(root),
/* Representation of the new page view */
::view-transition-new(root) {
   animation-duration: 2s;
}
```

This example is a very small sample of what we can achieve with this API. If you are interested in the practical use of this mechanism, We encourage you to read the material available in the Chrome browser documentation.

However, it is important to remember that this is a relatively new and experimental feature. This means that it may not be fully supported in some browsers. You can check the level of support for individual browsers on caniuse.

How does this feature work internally? When navigation starts, the API takes a screenshot of the current page and puts it in ::view-transition-old pseudoelement. It also renders a new page, puts it in view-transition-new and plays the animation between them.

Benefits:

The View Transitions API simplifies the creation of smooth and cohesive navigation animations, enhancing user experience by providing a seamless visual transition between different states or views of a web application.

Esbuild + Vite (stable)

Dev Experience | Performance

Challenge:

Until now, Angular's build system primarily relied on Webpack for compiling applications, requiring separate builders for different purposes like production builds, development server, server-side rendering, and prerendering. This led to more complex configurations in angular.json, slower build processes due to repetition of common steps, and less efficient module loading as it did not fully utilize ECMAScript Modules (ESM).

Solution:

In Angular v17, we received the new 'application' builder, which streamlines the build process by using @angular-devkit/build-angular:browser-esbuild as its core. This builder simplifies configurations by consolidating various tasks such as production builds, server-side rendering (SSR), and prerendering into a single, more efficient process. It enhances performance by executing common build steps only once and fully supports ECMAScript Modules (ESM), leading to faster build times and more efficient, modern module loading.

To apply these changes to an existing Angular project, change the builder from:

```
@angular-devkit/build-angular:browser
```

to:

```
@angular-devkit/build-angular:application
```

If your application supports SRR capabilities, you might need a few extra configuration changes (in such case check the official documentation for details).

The new application builder provides the functionality for all of the preexisting builders:

- browser
- prerender
- server
- ssr-dev-server

The usage of Vite build pipeline is encapsulated within the 'dev-server' builder, with no

changes necessary to use new build system.

Benefits:

Thanks to the unified solution, faster build-time performance is also brought to SSR apps. Using the same builder for both types of applications reduces the risk of potential differences resulting from using different bundlers. For new build systems with SSR & SSG, 'ng build' gets its speed increased by up to 87%, and 'ng serve' gets an 80% increase in speed.

SSR Hydration (stable)

Performance **UX**

Challenge:

Full Hydration presented in Angular version 16 was in developer preview mode, which could have stopped us from using it in applications since it was not yet stable.

Solution:

From version 17, the solution is marked as stable, which means it is considered safe for use in a production environment.

To enable it, use the following utility function in your project:

```
import { bootstrapApplication, provideClientHydration, } from '@angular/plat-
form-browser';

...

bootstrapApplication(RootCmp, {
  providers: [provideClientHydration()]
});
```

Benefits:

We can safely enjoy the benefits of hydration such as better UX (faster loading and interactivity, no more flickering) and enhanced SEO in a production environment.

CLI improvements

Dev Experience | Efficiency

Challenge:

The CLI does not take several new features from recent releases into account and uses older solutions by default.

Solution:

In Angular 17, applications generated by 'ng new' will be standalone by default and will use Vite as the default build system. All relevant code generators (i.e. for components, directives, pipes) will produce standalone output as well.

You can use the following migration to convert existing projects to the standalone APIs:

```
ng generate @angular/core:standalone
```

'ng-new' also supports the '--ssr' flag or alternatively includes a prompt asking if we wan't to enable SSR or SSG by default, including hydration.

```
Do you want to enable Server-Side Rendering (SSR) and Static Site Generation
(SSG/Prerendering)? (y/N) y
```

To try out the new built-in control flow syntax in your existing project, you can use the following migration:

```
ng generate @angular/core:control-flow
```

Benefits:

New features in the framework are becoming more accessible, and we get ready-made configurations out-of-the-box right from the start, which speeds up the project launch.

Devtools Dependency Graph

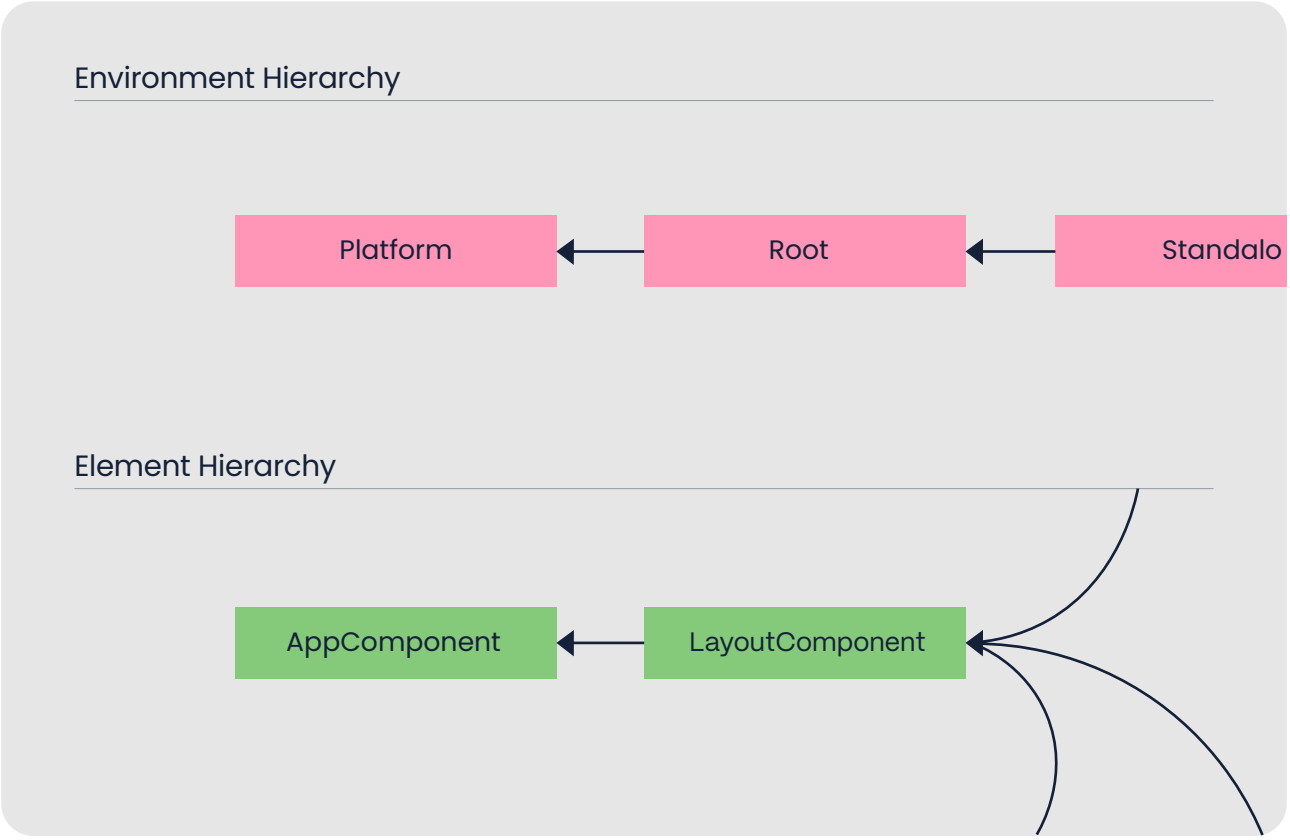
Dev Experience

Challenge:

Debugging dependency injection (DI) has always been problematic. If there were errors, they were always caused by the inability to inject a specific token, and if a token was successfully injected, there was no information about what injection context it came from.

Solution:

The Angular Team implemented brand new debugging APIs related to dependency injection. On top of that, they extended the capabilities of Angular Devtools and made it possible to inspect DI in the runtime. It is possible to verify a component's dependencies in a component inspector, injection tree with a dependency resolution path and providers declared in individual injectors.



Benefits:

The new debugging APIs for dependency injection in Angular helps developers to build more reliable and efficient applications.

Rebranding and introduction of angular.dev

Dev Experience

Challenge:

The branding of Angular has been almost identical since the beginning of AngularJS, which was released over 10 years ago. The framework documentation is also somewhat outdated, often based on ngModules.

Solution:

With the premiere of Angular version 17, we also received a new graphic identification of the framework and a new angular.dev - a home for Angular's new documentation.

Old logo:

New logo:

Angular.dev is a revamp with cutting-edge, interactive documentation. All examples have been reimagined with the standalone API and the latest Angular magic. Plus, we've got guides, tutorials and shiny Angular Playgrounds to code right in our browsers.

Benefits:

The new Angular.dev documentation makes it easier for developers to learn and use Angular.

Typescript/Node.js support

Angular v17 requires Typescript 5.2+ and Node.js v18+. Since 17.1, Angular also supports Typescript 5.3. New versions bring a handful of interesting changes that may be useful in Angular projects.

Explicit Resource Management (available in 5.2)

Dev Experience

Challenge:

There is no convenient way to explicitly manage resources with a specific lifetime. For example, run some cleanup code when a given object reaches the end of its life.

Solution:

The system for managing objects with a specific lifetime will be part of the ECMAScript standard. For Typescript, it was already delivered in version 5.2. This system is inspired by solutions from other programming languages, for example, the "using" syntax from C# or the "try-with-resource" syntax from Java. It allows a programmer to manage the resource declaratively, meaning an object is disposed when execution leaves the scope where object has been defined, or imperatively by calling a Symbol.dispose method manually.

You can create a manageable resource by implementing a new Disposable interface in your class or by creating a function that returns an object implementing Disposable interface. Usually we use such a feature when we need some sort of "clean-up" after creating an object. Common cases include deleting temporary files and closing internet connections.

In the example below, we define a manageable object inside the function scope. When execution leaves the function context, the Symbol.dispose method is automatically called. This happens even if there was an exception during processing. We no longer have to wrap logic into the try/catch/finally block to make sure that clean-up logic is executed in all cases.

```
class Foo implements Disposable {
   [Symbol.dispose]() {
      // “clean-up” code goes here
   }
}

function doSomething() {
  using foo = new Foo();

  // do something
 if(someCondition()) {
    // do another thing
   return;
 }
}
```

This feature also contains support for async disposable resources, Symbol.asyncDispose, and extra methods for dealing with stacks like DisposableStack and AsyncDisposableStack.

Note that this feature is only available if you set the compiler target to es2022 and added “esnext” or “esnext.disposable” to the “lib” array.

```
{
 "compilerOptions": {
    "target": "es2022",
    "lib": ["es2022", "esnext.disposable", "dom"]
 }
}
```

Benefits:

Explicit Resource Management enhances code cleanliness and safety by providing a straightforward and efficient way to manage the lifecycle of resources. This ensures that clean-up or disposal code is automatically executed when the object goes out of scope or encounters an exception, without the need for extensive try/catch/finally blocks.

Named and Anonymous Tuple Elements (available in 5.2)

Dev Experience

Challenge:

Previously, it was not possible to mix labeled and unlabeled elements inside a single tuple, which is an all-or-nothing restriction. It limited flexibility in tuple definitions and led to unnecessary labels or loss of label information, especially in merged or rest element scenarios.

Solution:

Typescript gained support for optional names or labels for each element of the tuple. The all-or-nothing restriction has been removed and labels go as far as preserving the merging with other tuples.

```
// fully labeled tuple
type Coordinates = [latitude: number, longitude: number];

//partially labeled tuple
type RGB = [number, green: number, number];

// all labels are still preserved
type Combined = [...Coordinates, ...RGB];
```

Benefits:

The improvement enables greater flexibility and readability in TypeScript code.

Copying Array Methods (available in 5.2)

Dev Experience

Challenge:

Old and well-known array methods like sort, splice or reverse, all update an array in-place, meaning they mutate the original array. This can lead to unintended side effects and make code harder to understand and debug, especially in complex applications with shared state.

Solution:

TypeScript 5.2 adds definitions of ECMAScript array methods for modifications by copy. All methods that mutate the original array have their non-mutating variants available and ready to use.

- array.reverse() => array.toReversed()
- array.sort(...) => array.toSorted(...)
- array.splice(...) => array.toSpliced(...)
- array[i] = foo => array.with(i, foo)

Benefits:

Using non-mutable array methods over methods that will mutate arrays in place in the JavaScript language have multiple benefits. These include enhancing code predictability, maintainability, and readability by avoiding unintended side effects and ensuring consistent data state across different parts of an application.

The future

Server-side rendering

The future of server-side rendering (SSR) for Angular is very bright. The Angular Team is actively working on improving the SSR experience, and there are a number of exciting new features on the horizon:

- Streamed SSR: This will allow Angular applications to load more quickly, as the server can stream the rendered HTML to the client as it is generated.
- Partial hydration: This will allow Angular to hydrate only the parts of the application that are needed immediately, which can further improve application performance.

Expert Opinion:

In the future Angular SSR technique will be easier to implement, it will not require much configuration and will provide hydration so that Angular apps will be able to resume at some point.

~ Aristeidis Bampakos
Google Developer Expert

Change Detection and Reactivity

Given that right now we have both, RxJS and signals, let's take a look at what the future of reactivity in Angular might look like:

- Signal-based reactivity will become the default reactivity mental model in Angular. This is because signals are more efficient and easier to understand than the current reactivity model based on RxJS observables.
- RxJS will stay with us for more complex implementations, so we can continue managing multiple asynchronous events at once. Combining RxJS with signals will be a new standard,
- With signal-based components, the fine-grained change detection will be widespread, with no extra traversal over the component tree.
- Zone.js will become optional in Angular. This is because signals will not require Zone.js to work, and removing Zone.js can improve performance and reduce bundle size.

Other changes

Further Angular-related changes that may appear soon include:

• Token based theming will be introduced in the Angular Material library. The Angular Team, in collaboration with the Material Design team, will define a token-based theming API. All components will be refactored to be compatible with such an API.

• Signals will gain their own runtime debug tools, as part of Angular DevTools, for signal-based components inspection.

• We might abandon the use of TS/JS decorators, and finally get functional components, due to the transition towards the Standalone function-based API,

• Improved hot-module replacement (HMR) for templates, styles and Typescript code.

Overall

Angular's vision for the future is to create a web development framework that is easy to learn and use, fast, high-performing and scalable. To achieve this vision, the Angular Team is committed to lowering the entry barrier for developers and improving factors such as developer experience, user experience and performance. They actively collaborate with the community and incorporate successful trends and best practices from other frameworks to continuously improve Angular.

What the experts are saying

How important is it to stay up-to-date with all the latest changes in Angular, given the current pace of changes?

Nowadays, Angular is making rapid changes, by introducing standalone APIs, signals, new control flow, etc. Although it is completely possible to not adapt the new changes and for example, keep using NgModules because of the backwards compatibility, it is still recommended to adapt to the changes, because there are, and will be even more, features only available for newer APIs. A great example is host directives, which are only possible with standalone APIs. But don't worry about refactoring, because Angular usually delivers migration schematics to migrate to standalone APIs, or to the new control flow.

~ Stefan Haas
Nx Champion

In this context, a rational approach is derived from the observed behavior of the majority of clients I have encountered. Within the organizations I have collaborated with, those employing Angular-based applications exhibit a certain reluctance when it comes to embracing the latest Angular versions. Their hesitancy arises from their search for compelling reasons to justify such upgrades. At the core of this deliberation lies the end consumer, the user, and their user experience.

For instance, take the case of Angular 15's Standalone API. Its implementation may not manifest any discernible impact on the end user; it could be argued that its primary effects are confined to enhancing the developer experience, and even this judgment is inherently subjective. It's important to acknowledge the formidable challenge posed by the task of updating huge applications that lack comprehensive test coverage. The absence of robust testing engenders substantial risks, rendering the application more susceptible to vulnerability in the face of such alterations.

~ Artur Androsovych
Angular Expert

In my opinion, it is very important. Especially in times of dynamic changes, I think the strategy of frequent and small updates is much more beneficial than waiting until everything settles down. I always invest time to keep my Angular projects updated to benefit from new Angular features and improvements. New features bring new patterns

and unleash better architectural seditions. Also, a simple update can often bring you performance, security, and bundle size improvements for free.

~ Dmytro Mezhenskyi
Google Developer Expert

I know with all the exciting new changes coming out, you may be excited to embrace the latest and greatest. And, pragmatically, I think it's a great way to learn for personal projects or small isolated projects, but when it comes to larger projects involving a lot of developers, I'd urge you to hold back for 2 things: For your team to feel comfortable discussing the change and taking on the work, and for good documentation and best practices to be established. Some of the new concepts are just the beginning of bigger things and we may find reasons to tweak, iterate, or pivot. No need to chase shiny as the goal. On the other hand, Angular has also released security updates, enhancements, and updating their dependencies to ensure the Angular projects we create are secure in recent versions. It just doesn't get all the attention some of the bigger changes get. It's vital to update Angular and dependencies following security guidelines and when enhanced security features are released.

~ J. Alisa Duncan
Google Developer Expert

If your current project is based on Angular 10 or an earlier version, it may start to feel outdated within the rapidly evolving framework landscape. The community is continually striving to stay up-to-date, and you'll often find that answers on platforms like StackOverflow and other websites revolve around the latest Angular frameworks.

Therefore, it is crucial to consider regularly upgrading your enterprise application to a version close to the latest release. From a practical standpoint, I recommend maintaining a gap of -1 or -2 Angular versions from the current one. For instance, if Angular 16 is the current version with Angular 17 on the horizon, I advise companies to upgrade their projects to at least Angular 14 or 15. This approach allows ample time for many enterprise projects with dependencies on various Angular libraries, both open-source and commercial, to catch up and align their libraries with the most recent developments.

~ Balram Chavan
Google Developer Expert

How do the overall changes affect the learning curve? Is it easier to learn Angular in its latest form?

I think that in the future it will definitely become easier. Before, everyone who came to try Angular was hit immediately by complex concepts like RxJS or NgModules. Now, the

learning curve has become more gentle and you have to learn less to build the simplest Angular app. Nowadays, we experience a transition period which is hard for newcomers because they have to learn the old stuff as well as new ones but I think in a year or two things will settle down.

~ Dmytro Mezhenskyi
Google Developer Expert

This question is somewhat complex to answer. If you're just starting to learn Angular now, the process is considerably easier than it was a few years ago. This is because most of the popular web frameworks have embraced TypeScript, and Angular is not the only framework advocating for a typed approach to web development. Notably, features like standalone components, simplified routing APIs, the introduction of new control flow syntax, and improved error reporting have made it more accessible for newcomers to dive into the Angular ecosystem.

In addition, the Angular documentation has grown significantly, offering a wealth of tutorials and advanced concepts. There are official cheat sheets, resources to address common errors and their solutions, an open Discord channel for developers to connect, and a range of Google-sponsored and community-driven events that enable developers to network and learn from one another.

However, it's worth noting that many experienced developers have expressed concerns about the rapid pace of change and the introduction of new features. This sometimes requires them to revisit and potentially rewrite their existing projects, as well as continually learn and adapt to stay current in the Angular landscape.

~ Balram Chavan
Google Developer Expert

I have been able to collect a lot of feedback from my students over the last few years. RxJs and the app structure with the modules were often described as too complicated. Little by little, we are eliminating these pain points.

~ David Muellerchen
Google Developer Expert

The learning curve will be flat for new beginners because they won't have to go through the same pains that experienced developers went through.

When experienced Angular developers learned the old version of Angular, they had to know NgModule and import it to AppModule before using Angular components. To write reactive code, they used Subject to store the state, subscribed the Subject to resolve the Observable, and finally unsubscribed the subscription to avoid a memory leak. These are a few examples of how Angular intimidated beginners in the old days.

Fast forward to 2023, the difficulties of building Angular applications are significantly lower. We can learn standalone components and use them as the building blocks of an application. We can store state in Signal and call APIs to set, update and compute to calculate Signal value. If the requirements require an HTTP request or asynchronous reactivity, developers can learn RxJS and the minimal operators. Learning RxJS can move from the top learning list to a lower position. toSignal and toObservable RxJS-interop can convert between Signal and Observable but they must be used with discretion.

With the new control flow, we should learn @if, @for and @switch instead of the directive counterparts. From my personal experience, @if is easier than ngIf when there are 2 or more else-if. I cannot find a precise and elegant way with ngIf, ngElseIf and ngElse directives. ngFor has trackBy, but it is not compulsory to use whereas @for makes track mandatory and can be used to optimize a long list.

~ Connie Leung
Google Developer Expert

Wow, I feel bad for new Angular developers. They must feel like there's quite a dichotomy. I think the difficulty in learning changes from the learner's past experience. If they are already familiar with the web frameworks out there, they may prefer the new changes, and if they are .NET devs, they may prefer the prior form of Angular. New devs who have to learn how to maintain legacy Angular projects while simultaneously learning new changes for new work are the ones I feel the most for. That said, while learning can cause discomfort for all of us, I'm looking forward to these changes, how the framework grows, and seeing all the cool things we can do with these features.

~ J. Alisa Duncan
Google Developer Expert

What are your expectations for the direction and pace of change in Angular in the future?

In the next few versions, I expect that changes in the framework will reduce a lot of boilerplate code, improve the developer experience and apply ergonomic approaches when it comes to building Angular apps.

~ Aristeidis Bampakos
Google Developer Expert

The Angular team is highly responsive to community feedback, and their recent additions reflect their commitment to improving the framework. Features like standalone components, the inject() function for dependency injection, functional guard routes, and the introduction of new control flow syntax for directives like ngFor and ngIf

demonstrate the team's endeavor to incorporate the best practices from other popular frameworks such as React, Svelte, and Vue. Furthermore, the Angular team is investing more effort into areas like Server-Side Rendering, Hydration, SEO optimization, integrating ESBuild, engaging in discussions around Microfrontends, and more. These initiatives clearly indicate that there are exciting developments on the horizon for Angular.

~ Balram Chavan
Google Developer Expert

There is a brighter future for Angular with all the great provided features and improvements since v14 to v17. It's the renaissance of Angular. These updates bring significant improvements and simplifications, such as standalone components, the removal of the need for modules, and the introduction of functional guards. The latest versions introduce powerful features, including built-in control flow, enhanced performance in v17, and deferred loading, which allows you to achieve a lot with minimal code. The Angular team has hinted at a big surprise in v17, I won't spoil it (hahah), but this surprise will for sure make learning Angular easier and greatly enhance the learning experience.

~ Fatima Amzil
Google Developer Expert

It is the best time to be an Angular developer, and the Angular team works tirelessly to include new and exciting features in each release. Angular 17 has not come out yet, and major Angular podcasters have already scheduled meetings with Angular team members to discuss Angular 18 and beyond.

~ Connie Leung
Google Developer Expert

Thank You

We appreciate you taking the time to read this ebook. Our team at House of Angular is passionate about Angular, and we regularly share educational content that helps businesses and developers gain a better understanding of the Angular framework.

We hope you found this ebook informative and that its contents will help you improve your app's performance, user experience and overall project efficiency.

If you have any questions about Angular or its features, we would be happy to help.

Contact us

Bibliography

Official Typescript Documentation: https://www.typescriptlang.org/
Official Angular Documentation (old): https://angular.io/
Official Angular Documentation (new): https://angular.dev/
Official Angular blog: https://blog.angular.io/
Angular.love blog: https://www.angular.love/en/

About authors

Main Author

Mateusz Dobrowolski
Angular Developer at House of Angular. He takes an active part in the angular.love community, writing expert articles, and sharing his knowledge at Angular meetups.

If you found this e-book useful (or not) please, let us know. Send your feedack to the author: @m-dobrowolski

Special thanks to

Krzysztof Skorupka
Angular Team Leader & Architect at House of Angular. Expert at Angular.love community, blogger, and speaker.

Mateusz Stefańczyk
Angular Team Leader at House of Angular. Expert at Angular.love community, blogger, and speaker.

Przemysław Łęczycki
Angular Team Leader at House of Angular. Expert at Angular.love community, blogger, and speaker.

Dominik Donoch
Angular Developer at House of Angular. Blogger and active member of Angular.love community.

Mateusz Basiński
Angular Developer at House of Angular. Angular.love blogger.

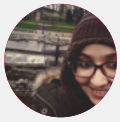
Maja Hendzel
Angular Developer at House of Angular. Angular.love blogger.

Miłosz Rutkowski
Angular Developer at House of Angular. Angular.love blogger.

Dawid Kostka
Angular Developer at House of Angular. Angular.love blogger.

Piotr Wiórek
Blogger and member of Angular.love community.

Invited experts

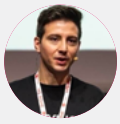


Fatima Amzil
Fatima is a cross-functional frontend technical leader and an expert in Angular. Outside of her professional life, she contributes as a technical writer on Medium, mentor at MyJobGlasses, and an Angular GDE.

Artur Androsovych
Artur is an OSS core developer of various projects as NGXS, NG-ZORRO, single-spa, ngneat (until-destroy code owner), and RxAngular. He was a Google Developer Expert in Angular in 2023. He focuses on runtime performance and has taught teams about Angular internals for the past few years.

Aristeidis Bampakos
Aristeidis is an Angular GDE who works as Web Development Team Lead at Plex-Earth. He is an award-winning author of Learning Angular and Angular Projects books. Currently, he is leading the effort of making Angular accessible to the Greek development community by maintaining the Greek translation of the official Angular documentation.

Balram Chavan
Balram is an Architect specializing in designing and creating scalable, secure solutions for various domains. His expertise spans the cloud, AI and web development landscape. As the author of an Angular framework book, he excels as a team player, an effective communicator, and a mentor.



Luca Del Puppo
Luca is a Senior Software Developer, Microsoft MVP, Google Developer Expert and GitKraken Ambassador. He loves JavaScript and TypeScript. In his free time, I loves studying new technologies, improving himself, creating YouTube content or writing technical articles.

J. Alisa Duncan
Alisa is a Senior Developer Advocate at Okta, full-stack developer, content creator, conference speaker, Pluralsight author, and community builder who

loves the thrill of learning new things. She is a Google Developer Expert in Angular, a Women Techmaker Ambassador, a ngGirls core team member, and a volunteer at community events supporting underrepresented groups entering tech.

Stefan Haas

Stefan is a freelancer and trainer focusing on Angular from Austria. He is the author of NG Journal - The Place Beyond Fundamentals - and an Nx Champion.

Connie Leung

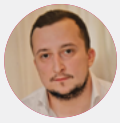
Google Developer Expert for Angular. Software Architect at Diginex, blogger and youtube content creator.

Dmytro Mezhenskyi

Dmytro Mezhenskyi is the founder of the Decoded Frontend YouTube channel, where he shares his knowledge about the Angular framework. As an author, he has created a series of advanced video courses focusing on Angular and GraphQL. With over 10 years of experience in frontend development, Dmytro has been recognized as a Google Developer Expert in Angular and a Microsoft MVP in Web Development.

David Muellerchen

Better known as WebDave, David has been an integral part of the Angular community since 2014. He is a Google Developer Expert, an Angular consultant and trainer, and a frequent speaker at meetups and conferences. He also organizes weekly Angular livestreams and the Hamburg Angular Meetup.



Marko Stanimirović

Marko is a Principal Frontend Engineer at Swiss Marketplace Group. He is also a core member of the NgRx and AnalogJS teams, a Google Developer Expert in Angular, and an organizer of the Angular Belgrade group.

HOUSE OF ANGULAR

House of Angular is a software agency with a focus on developing and supporting Angular applications. They offer a variety of services, including:

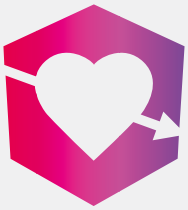
- application development,
- team extension,
- product and team audits,
- and business consulting.

Having worked with Angular since its beginning and with over 10 years in app development, they have become a trusted business partner for companies of different sizes and sectors.

Their goal is to craft the finest frontend solutions using Angular, while also imparting their knowledge and expertise to other users of this framework.

The company is actively involved in nurturing the Angular community. They support key Angular libraries like ngrx, NGXS, and ngneat. Additionally, their team members contribute to open-source Angular projects and spread their expertise through writing articles and giving talks at European meetups.

House of Angular has been recognized for their contributions with the Angular Hero of Community 2021 award.



angular.love

Angular.Love is a community platform for Angular enthusiasts, supportedby House of Angular to facilitate the growth of Angular developers through knowledge-sharing initiatives. It started as a blog where experts published articles about Angular news, features, and best practices. Now angular.love also organizes in-person and online meetups, which frequently feature Google Developer Experts.

Angular developers can find expert insights and expand their Angular knowledge by reading the angular.love blog, attending meetups with talks from experts, and following the community platform's social media pages.

Angular.love community is a recipient of the Angular Hero of Education 2022 award.

Blog | Meetups | YouTube | X (formerly Twitter)

HOUSE OF ANGULAR

created in cooperation with

angular.love