Template type checking

Overview of template type checking

Just as TypeScript catches type errors in your code, Angular checks the expressions and bindings within the templates of your application and can report any type errors it finds. Angular currently has three modes of doing this, depending on the value of the `fullTemplateTypeCheck` and `strictTemplates` flags in the TypeScript configuration file.

Basic mode

In the most basic type-checking mode, with the `fullTemplateTypeCheck` flag set to `false`, Angular validates only top-level expressions in a template.

If you write `<map [city]="user.address.city">`, the compiler verifies the following:

- `user` is a property on the component class.
- `user` is an object with an address property.
- `user.address` is an object with a city property.

The compiler does not verify that the value of `user.address.city` is assignable to the city input of the `<map>` component.

The compiler also has some major limitations in this mode:

- Importantly, it doesn't check embedded views, such as `*ngIf`, `*ngFor`, other `<ng-template>` embedded view.
- It doesn't figure out the types of `#refs`, the results of pipes, or the type of `$event` in event bindings.

In many cases, these things end up as type `any`, which can cause subsequent parts of the expression to go unchecked.

Full mode

If the `fullTemplateTypeCheck` flag is set to `true`, Angular is more aggressive in its type-checking within templates. In particular:

- Embedded views (such as those within an `*ngIf` or `*ngFor`) are checked.
- Pipes have the correct return type.
- Local references to directives and pipes have the correct type (except for any generic parameters, which will be `any`).

The following still have type `any`.

- Local references to DOM elements.
- The `$event` object.
- Safe navigation expressions.

The `fullTemplateTypeCheck` flag has been deprecated in Angular 13. The `strictTemplates` family of compiler options should be used instead.

{@a strict-mode}

Strict mode

Angular maintains the behavior of the `fullTemplateTypeCheck` flag, and introduces a third "strict mode". Strict mode is a superset of full mode, and is accessed by setting the `strictTemplates` flag to true. This flag supersedes the `fullTemplateTypeCheck` flag. In strict mode, Angular uses checks that go beyond the version 8 type-checker. Note that strict mode is only available if using Ivy.

In addition to the full mode behavior, Angular does the following:

- Verifies that component/directive bindings are assignable to their `@Input()` s.
- Obeys TypeScript's `strictNullChecks` flag when validating the preceding mode.
- Infers the correct type of components/directives, including generics.
- Infers template context types where configured (for example, allowing correct type-checking of `NgFor`).
- Infers the correct type of `$event` in component/directive, DOM, and animation event bindings.
- Infers the correct type of local references to DOM elements, based on the tag name (for example, the type that `document.createElement` would return for that tag).

Checking of `*ngFor`

The three modes of type-checking treat embedded views differently. Consider the following example.

interface User { name: string; address: { city: string; state: string; } }

```
<div *ngFor="let user of users">
  <h2>{{config.title}}</h2>
  <span>City: {{user.address.city}}</span>
</div>
```

The `<h2>` and the `<span>` are in the `*ngFor` embedded view. In basic mode, Angular doesn't check either of them. However, in full mode, Angular checks that `config` and `user` exist and assumes a type of `any`. In strict mode, Angular knows that the `user` in the `<span>` has a type of `User`, and that `address` is an object with a `city` property of type `string`.

{@a troubleshooting-template-errors}

Troubleshooting template errors

With strict mode, you might encounter template errors that didn't arise in either of the previous modes. These errors often represent genuine type mismatches in the templates that were not caught by the previous tooling. If this is the case, the error message should make it clear where in the template the problem occurs.

There can also be false positives when the typings of an Angular library are either incomplete or incorrect, or when the typings don't quite line up with expectations as in the following cases.

- When a library's typings are wrong or incomplete (for example, missing `null | undefined` if the library was not written with `strictNullChecks` in mind).
- When a library's input types are too narrow and the library hasn't added appropriate metadata for Angular to figure this out. This usually occurs with disabled or other common Boolean inputs used as attributes, for example, `<input disabled>`.
- When using `$event.target` for DOM events (because of the possibility of event bubbling, `$event.target` in the DOM typings doesn't have the type you might expect).

In case of a false positive like these, there are a few options:

- Use the `$any()` type-cast function in certain contexts to opt out of type-checking for a part of the expression.
- Disable strict checks entirely by setting `strictTemplates: false` in the application's TypeScript configuration file, `tsconfig.json`.
- Disable certain type-checking operations individually, while maintaining strictness in other aspects, by setting a *strictness flag* to `false`.
- If you want to use `strictTemplates` and `strictNullChecks` together, opt out of strict null type checking specifically for input bindings using `strictNullInputTypes`.

Unless otherwise noted, each following option is set to the value for `strictTemplates` (`true` when `strictTemplates` is `true` and conversely, the other way around).

Strictness flag	Effect
`strictInputTypes`	Whether the assignability of a binding expression to the `@Input()` field is checked. Also affects the inference of directive generic types.
`strictInputAccessModifiers`	Whether access modifiers such as `private`/`protected`/`readonly` are honored when assigning a binding expression to an `@Input()`. If disabled, the access modifiers of the `@Input` are ignored; only the type is checked. This option is `false` by default, even with `strictTemplates` set to `true`.
`strictNullInputTypes`	Whether `strictNullChecks` is honored when checking `@Input()` bindings (per `strictInputTypes`). Turning this off can be useful when using a library that was not built with `strictNullChecks` in mind.
`strictAttributeTypes`	Whether to check `@Input()` bindings that are made using text attributes. For example, `<input matInput disabled="true">` (setting the `disabled` property to the string `'true'`) vs `<input matInput [disabled]="true">` (setting the `disabled` property to the boolean `true`).
`strictSafeNavigationTypes`	Whether the return type of safe navigation operations (for example, `user?.name`) will be correctly inferred based on the type of `user`). If disabled, `user?.name` will be of type `any`.
`strictDomLocalRefTypes`	Whether local references to DOM elements will have the correct type. If disabled `ref` will be of type `any` for `<input #ref>`.
`strictOutputEventTypes`	Whether `$event` will have the correct type for event bindings to component/directive an `@Output()`, or to animation events. If disabled, it will be `any`.
`strictDomEventTypes`	Whether `$event` will have the correct type for event bindings to DOM events. If disabled, it will be `any`.
`strictContextGenerics`	Whether the type parameters of generic components will be inferred correctly (including any generic bounds). If disabled, any type parameters will be `any`.
`strictLiteralTypes`	Whether object and array literals declared in the template will have their type inferred. If disabled, the type of such literals will be `any`. This flag is `true` when *either* `fullTemplateTypeCheck` or `strictTemplates` is set to `true`.

If you still have issues after troubleshooting with these flags, fall back to full mode by disabling `strictTemplates`.

If that doesn't work, an option of last resort is to turn off full mode entirely with `fullTemplateTypeCheck: false`.

A type-checking error that you cannot resolve with any of the recommended methods can be the result of a bug in the template type-checker itself. If you get errors that require falling back to basic mode, it is likely to be such a bug. If this happens, file an issue so the team can address it.

Inputs and type-checking

The template type checker checks whether a binding expression's type is compatible with that of the corresponding directive input. As an example, consider the following component:

```
export interface User {
  name: string;
}

@Component({
  selector: 'user-detail',
  template: '{{ user.name }}',
})
export class UserDetailComponent {
  @Input() user: User;
}
```

The `AppComponent` template uses this component as follows:

```
@Component({
  selector: 'app-root',
  template: '<user-detail [user]="selectedUser"></user-detail>',
})
export class AppComponent {
  selectedUser: User | null = null;
}
```

Here, during type checking of the template for `AppComponent`, the `[user]="selectedUser"` binding corresponds with the `UserDetailComponent.user` input. Therefore, Angular assigns the `selectedUser` property to `UserDetailComponent.user`, which would result in an error if their types were incompatible. TypeScript checks the assignment according to its type system, obeying flags such as `strictNullChecks` as they are configured in the application.

Avoid run-time type errors by providing more specific in-template type requirements to the template type checker. Make the input type requirements for your own directives as specific as possible by providing template-guard functions in the directive definition. See Improving template type checking for custom directives in this guide.

Strict null checks

When you enable `strictTemplates` and the TypeScript flag `strictNullChecks`, typecheck errors might occur for certain situations that might not easily be avoided. For example:

- A nullable value that is bound to a directive from a library which did not have `strictNullChecks` enabled.

For a library compiled without `strictNullChecks`, its declaration files will not indicate whether a field can be `null` or not. For situations where the library handles `null` correctly, this is problematic, as the compiler will check a nullable value against the declaration files which omit the `null` type. As such, the compiler produces a type-check error because it adheres to `strictNullChecks`.

- Using the `async` pipe with an Observable which you know will emit synchronously.

The `async` pipe currently assumes that the Observable it subscribes to can be asynchronous, which means that it's possible that there is no value available yet. In that case, it still has to return something—which is `null`. In other words, the return type of the `async` pipe includes `null`, which might result in errors in situations where the Observable is known to emit a non-nullable value synchronously.

There are two potential workarounds to the preceding issues:

1. In the template, include the non-null assertion operator `!` at the end of a nullable expression, such as `<user-detail [user]="user!"></user-detail>`.

In this example, the compiler disregards type incompatibilities in nullability, just as in TypeScript code. In the case of the `async` pipe, note that the expression needs to be wrapped in parentheses, as in `<user-detail [user]="(user$ | async)!"></user-detail>`.

1. Disable strict null checks in Angular templates completely.

When `strictTemplates` is enabled, it is still possible to disable certain aspects of type checking. Setting the option `strictNullInputTypes` to `false` disables strict null checks within Angular templates. This flag applies for all components that are part of the application.

Advice for library authors

As a library author, you can take several measures to provide an optimal experience for your users. First, enabling `strictNullChecks` and including `null` in an input's type, as appropriate, communicates to your consumers whether they can provide a nullable value or not. Additionally, it is possible to provide type hints that are specific to the template type checker. See Improving template type checking for custom directives, and Input setter coercion.

{@a input-setter-coercion}

Input setter coercion

Occasionally it is desirable for the `@Input()` of a directive or component to alter the value bound to it, typically using a getter/setter pair for the input. As an example, consider this custom button component:

Consider the following directive:

```
@Component({
  selector: 'submit-button',
  template: `
    <div class="wrapper">
      <button [disabled]="disabled">Submit</button>
    </div>
  `,
})
```

```
class SubmitButton {
  private _disabled: boolean;

  @Input()
  get disabled(): boolean {
    return this._disabled;
  }

  set disabled(value: boolean) {
    this._disabled = value;
  }
}
```

Here, the `disabled` input of the component is being passed on to the `<button>` in the template. All of this works as expected, as long as a `boolean` value is bound to the input. But, suppose a consumer uses this input in the template as an attribute:

```
<submit-button disabled></submit-button>
```

This has the same effect as the binding:

```
<submit-button [disabled]="''"></submit-button>
```

At runtime, the input will be set to the empty string, which is not a `boolean` value. Angular component libraries that deal with this problem often "coerce" the value into the right type in the setter:

```
set disabled(value: boolean) {
  this._disabled = (value === '') || value;
}
```

It would be ideal to change the type of `value` here, from `boolean` to `boolean|''`, to match the set of values which are actually accepted by the setter. TypeScript prior to version 4.3 requires that both the getter and setter have the same type, so if the getter should return a `boolean` then the setter is stuck with the narrower type.

If the consumer has Angular's strictest type checking for templates enabled, this creates a problem: the empty string `''` is not actually assignable to the `disabled` field, which creates a type error when the attribute form is used.

As a workaround for this problem, Angular supports checking a wider, more permissive type for `@Input()` than is declared for the input field itself. Enable this by adding a static property with the `ngAcceptInputType_` prefix to the component class:

```
class SubmitButton {
  private _disabled: boolean;

  @Input()
  get disabled(): boolean {
    return this._disabled;
  }
```

```
  set disabled(value: boolean) {
    this._disabled = (value === '') || value;
  }

  static ngAcceptInputType_disabled: boolean|'';
}
```

Since TypeScript 4.3, the setter could have been declared to accept `boolean|''` as type, making the input setter coercion field obsolete. As such, input setters coercion fields have been deprecated.

This field does not need to have a value. Its existence communicates to the Angular type checker that the `disabled` input should be considered as accepting bindings that match the type `boolean|''`. The suffix should be the `@Input` *field* name.

Care should be taken that if an `ngAcceptInputType_` override is present for a given input, then the setter should be able to handle any values of the overridden type.

Disabling type checking using `$any()`

Disable checking of a binding expression by surrounding the expression in a call to the `$any()` cast pseudo-function. The compiler treats it as a cast to the `any` type just like in TypeScript when a `<any>` or `as any` cast is used.

In the following example, casting `person` to the `any` type suppresses the error `Property address does not exist`.

```
@Component({
  selector: 'my-component',
  template: '{{$any(person).addresss.street}}'
})
class MyComponent {
  person?: Person;
}
```