Angular elements overview

Angular elements are Angular components packaged as *custom elements* (also called Web Components), a web standard for defining new HTML elements in a framework-agnostic way.

Custom elements are a Web Platform feature currently supported by Chrome, Firefox, Opera, and Safari, and available in other browsers through polyfills (see Browser Support). A custom element extends HTML by allowing you to define a tag whose content is created and controlled by JavaScript code. The browser maintains a `CustomElementRegistry` of defined custom elements, which maps an instantiable JavaScript class to an HTML tag.

The `@angular/elements` package exports a `createCustomElement()` API that provides a bridge

from Angular's component interface and change detection functionality to the built-in DOM API.

Transforming a component to a custom element makes all of the required Angular infrastructure available to the browser. Creating a custom element is simple and straightforward, and automatically connects your component-defined view with change detection and data binding, mapping Angular functionality to the corresponding native HTML equivalents.

> We are working on custom elements that can be used by web apps built on other frameworks. A minimal, self-contained version of the Angular framework will be injected as a service to support the component's change-detection and data-binding functionality. For more about the direction of development, check out this video presentation.

Using custom elements

Custom elements bootstrap themselves - they start automatically when they are added to the DOM, and are automatically destroyed when removed from the DOM. Once a custom element is added to the DOM for any page, it looks and behaves like any other HTML element, and does not require any special knowledge of Angular terms or usage conventions.

- **Easy dynamic content in an Angular app**
  Transforming a component to a custom element provides an easy path to creating dynamic HTML content in your Angular app. HTML content that you add directly to the DOM in an Angular app is normally displayed without Angular processing, unless you define a *dynamic component*, adding your own code to connect the HTML tag to your app data, and participate in change detection. With a custom element, all of that wiring is taken care of automatically.

- **Content-rich applications**
  If you have a content-rich app, such as the Angular app that presents this documentation, custom elements let you give your content providers sophisticated Angular functionality without requiring knowledge of Angular. For example, an Angular guide like this one is added directly to the DOM by the Angular navigation tools, but can include special elements like `<code-snippet>` that perform complex operations. All you need to tell your content provider is the syntax of your custom

element. They don't need to know anything about Angular, or anything about your component's data structures or implementation.

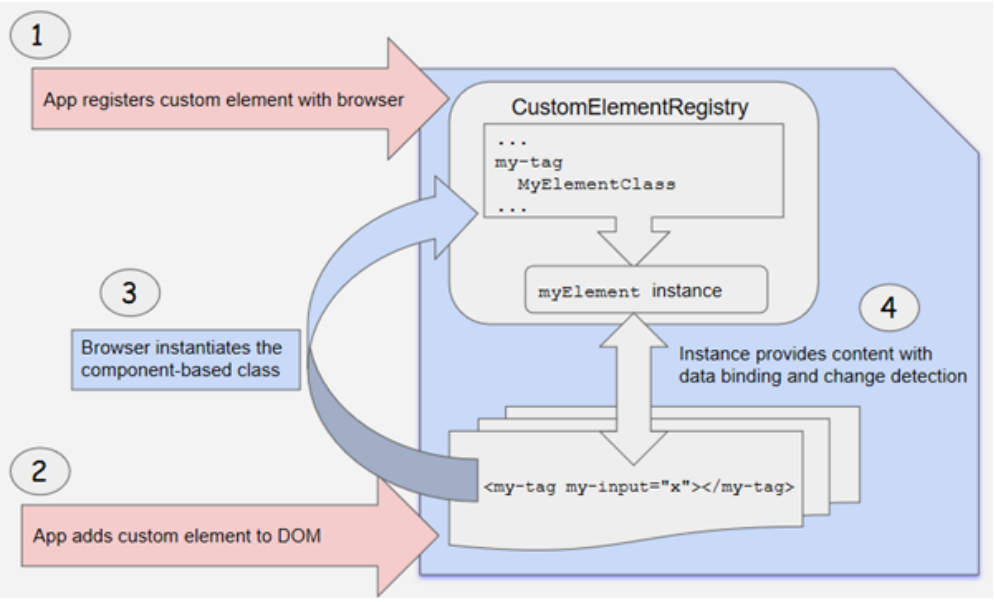
How it works

Use the `createCustomElement()` function to convert a component into a class that can be registered with the browser as a custom element. After you register your configured class with the browser's custom-element registry, you can use the new element just like a built-in HTML element in content that you add directly into the DOM:

```
<my-popup message="Use Angular!"></my-popup>
```

When your custom element is placed on a page, the browser creates an instance of the registered class and adds it to the DOM. The content is provided by the component's template, which uses Angular

template syntax, and is rendered using the component and DOM data. Input properties in the component correspond to input attributes for the element.



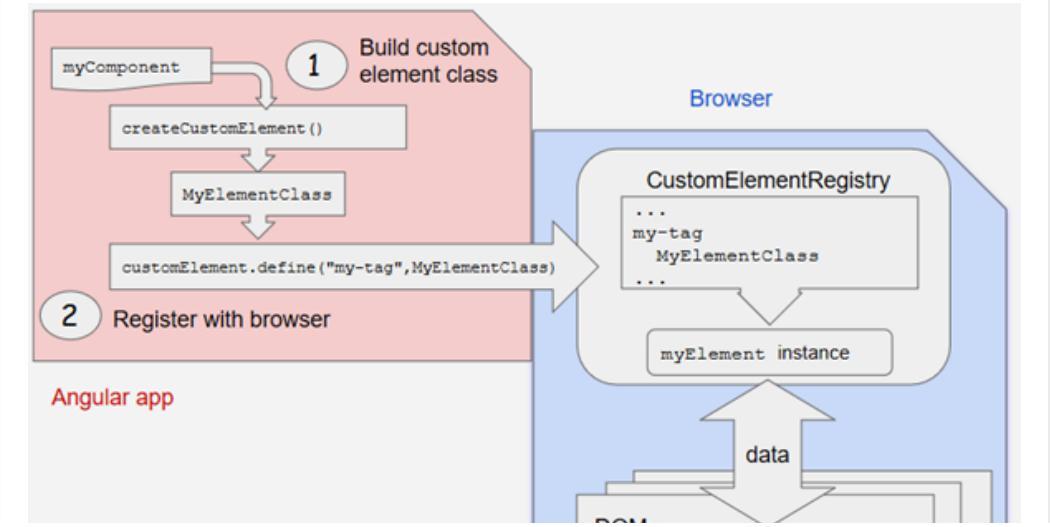
Transforming components to custom elements

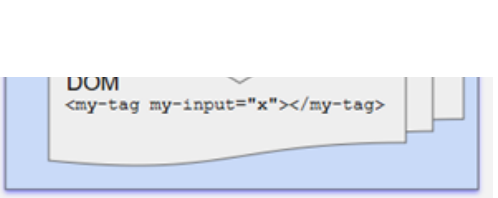
Angular provides the `createCustomElement()` function for converting an Angular component, together with its dependencies, to a custom element.

The function collects the component's observable properties, along with the Angular functionality the browser needs to create and destroy instances, and to detect and respond to changes.

The conversion process implements the `NgElementConstructor` interface, and creates a constructor class that is configured to produce a self-bootstrapping instance of your component.

Use a JavaScript function, `customElements.define()`, to register the configured constructor and its associated custom-element tag with the browser's `CustomElementRegistry`. When the browser encounters the tag for the registered element, it uses the constructor to create a custom-element instance.





Mapping

A custom element *hosts* an Angular component, providing a bridge between the data and logic defined in the component and standard DOM APIs. Component properties and logic maps directly into HTML attributes and the browser's event system.

- The creation API parses the component looking for input properties, and defines corresponding attributes for the custom element. It transforms the property names to make them compatible with custom elements, which do not recognize case distinctions. The resulting attribute names use dash-separated lowercase. For example, for a component with `@Input('myInputProp') inputProp`, the corresponding custom element defines an attribute `my-input-prop`.
- Component outputs are dispatched as HTML Custom Events, with the name of the custom event matching the output name. For example, for a component with `@Output() valueChanged = new EventEmitter()`, the corresponding custom element will dispatch events with the name "valueChanged", and the emitted data will be stored on the event's `detail` property. If you provide an alias, that value is used; for example, `@Output('myClick') clicks = new EventEmitter<string>();` results in dispatch events with the name "myClick".

For more information, see Web Component documentation for Creating custom events.

Browser support for custom elements

The recently-developed custom elements Web Platform feature is currently supported natively in a number of browsers. Support is pending or planned in other browsers.

Browser	Custom Element Support
Chrome	Supported natively.
Opera	Supported natively.
Safari	Supported natively.
Firefox	Supported natively.
Edge	Working on an implementation.

In browsers that support Custom Elements natively, the specification requires developers use ES2015 classes to define Custom Elements - developers can opt-in to this by setting the `target: "es2015"` property in their project's `tsconfig.json`. As Custom Element and ES2015 support may not be available in all browsers, developers can instead choose to use a polyfill to support older browsers and ES5 code.

Use the Angular CLI to automatically set up your project with the correct polyfill: `ng add @angular/elements --name=*your_project_name*`.

- For more information about polyfills, see polyfill documentation.
- For more information about Angular browser support, see Browser Support.

Example: A Popup Service

Previously, when you wanted to add a component to an app at runtime, you had to define a *dynamic component*. The app module would have to list your dynamic component under `entryComponents`, so that the app wouldn't expect it to be present at startup, and then you would have to load it, attach it to an element in the DOM, and wire up all of the dependencies, change detection, and event handling, as described in Dynamic Component Loader.

Using an Angular custom element makes the process much simpler and more transparent, by providing all

of the infrastructure and framework automatically—all you have to do is define the kind of event handling you want. (You do still have to exclude the component from compilation, if you are not going to use it in your app.)

The Popup Service example app (shown below) defines a component that you can either load dynamically or convert to a custom element.

- `popup.component.ts` defines a simple pop-up element that displays an input message, with some animation and styling.
- `popup.service.ts` creates an injectable service that provides two different ways to invoke the PopupComponent; as a dynamic component, or as a custom element. Notice how much more setup is required for the dynamic-loading method.
- `app.module.ts` adds the PopupComponent in the module's `entryComponents` list, to exclude it from compilation and avoid startup warnings or errors.
- `app.component.ts` defines the app's root component, which uses the PopupService to add the pop-up to the DOM at run time. When the app runs, the root component's constructor converts PopupComponent to a custom element.

For comparison, the demo shows both methods. One button adds the popup using the dynamic-loading

method, and the other uses the custom element. You can see that the result is the same; only the preparation is different.

popup.component.ts *popup.service.ts*

```
import { Component, EventEmitter,
Input, Output } from '@angular/core';
import { animate, state, style,
transition, trigger } from
'@angular/animations';

@Component({
  selector: 'my-popup',
  template: `
    <span>Popup: {{message}}</span>
    <button
(click)="closed.next()">&#x2716;
</button>
  `,
  host: {
    '[@state]': 'state',
  },
  animations: [
    trigger('state', [
```

```
    trigger('state', [
      state('opened', style({transform:
'translateY(0%)'})),
      state('void, closed',
style({transform: 'translateY(100%)',
opacity: 0})),
      transition('* => *',
animate('100ms ease-in')),
    ])
  ],
  styles: [`
    :host {
      position: absolute;
      bottom: 0;
      left: 0;
      right: 0;
      background: #009cff;
      height: 48px;
      padding: 16px;
      display: flex;
      justify-content: space-between;
      align-items: center;
      border-top: 1px solid black;
      font-size: 24px;
    }
```

```
    button {
      border-radius: 50%;
    }
  `]
})
export class PopupComponent {
  private state: 'opened' | 'closed' =
'closed';

  @Input()
  set message(message: string) {
    this._message = message;
    this.state = 'opened';
  }
  get message(): string { return
this._message; }
  _message: string;

  @Output()
  closed = new EventEmitter();
}
```

You can download the full code for the example here.

Typings for custom elements

Generic DOM APIs, such as `document.createElement()` or `document.querySelector()`, return an element type that is appropriate for the specified arguments. For example, calling `document.createElement('a')` will return an `HTMLAnchorElement`, which TypeScript knows has an `href` property. Similarly, `document.createElement('div')` will return an `HTMLDivElement`, which TypeScript knows has no `href` property.

When called with unknown elements, such as a custom element name (`popup-element` in our example), the methods will return a generic type, such as `HTMLElement`, since TypeScript can't infer the correct type of the returned element.

Custom elements created with Angular extend `NgElement` (which in turn extends `HTMLElement`).

Additionally, these custom elements will have a property for each input of the corresponding component. For example, our `popup-element` will have a `message` property of type `string`.

There are a few options if you want to get correct types for your custom elements. Let's assume you create a `my-dialog` custom element based on the following component:

```
@Component(...)
class MyDialog {
  @Input() content: string;
}
```

The most straight forward way to get accurate typings is to cast the return value of the relevant DOM methods to the correct type. For that, you can use the `NgElement` and `WithProperties` types (both exported from `@angular/elements`):

```
const aDialog =
document.createElement('my-dialog') as
NgElement & WithProperties<{content:
string}>;
aDialog.content = 'Hello, world!';
aDialog.content = 123;  // <-- ERROR:
TypeScript knows this should be a
string.
aDialog.body = 'News';  // <-- ERROR:
TypeScript knows there is no `body`
property on `aDialog`.
```

This is a good way to quickly get TypeScript features, such as type checking and autocomplete support, for you custom element. But it can get cumbersome if you need it in several places, because you have to cast the return type on every occurrence.

An alternative way, that only requires defining each custom element's type once, is augmenting the `HTMLElementTagNameMap`, which TypeScript uses to

infer the type of a returned element based on its tag name (for DOM methods such as `document.createElement()`, `document.querySelector()`, etc.):

```
declare global {
  interface HTMLElementTagNameMap {
    'my-dialog': NgElement &
WithProperties<{content: string}>;
    'my-other-element': NgElement &
WithProperties<{foo: 'bar'}>;
    ...
  }
}
```

Now, TypeScript can infer the correct type the same way it does for built-in elements:

```
document.createElement('div')
//--> HTMLDivElement (built-in element)
document.querySelector('foo')
//--> Element        (unknown element)
document.createElement('my-dialog')
//--> NgElement &
WithProperties<{content: string}>
(custom element)
document.querySelector('my-other-
element')  //--> NgElement &
WithProperties<{foo: 'bar'}>
(custom element)
```