Angular Fundamentals

Angular Developer Series

Peter Munro

Angular Fundamentals

Angular Fundamentals

Single Page Applications (SPAs)

Angular Fundamentals

Drawbacks of Page-Based Sites

- Typically pages on same site share common features
  - same header, footer, side bar
- Reloading entire page is slow
  - requires multiple requests (HTML, images, JS and CSS)
  - even if coming from cache browser has to check and load them
  - browser has to redraw entire page, rendering and layout can be slow
- Bad user experience
  - white screen of death while page navigation takes place

Single Page Applications (SPAs)

- Load a single html page
  - add, hide and display DOM elements dynamically
- Use JavaScript for interactivity
  - handles user interactions, business logic
  - uses XHR to send and request data to and from server
  - transforms data from server into DOM

Deep Linking

- Often need shareable linking to launch into specific view
  - SPA's only have single URL so need a way to store and share view
- Use # in URL to add state without reloading page
  - e.g. http://example.com/address/index.html/#user/abc1
  - most frameworks provide in built URL router
- URL Router
  - handles parsing URL on initial page load
  - notified when URL changes
  - tells a view that it should be rendered

HTML5 History API

History API enables:

1. client to maintain state even across (local) URL changes
2. a URL without using #
3. use of the back and forwards buttons

See MDN

Angular

Angular Fundamentals

Angular Introduction

- Open Source Web App Framework from Google
  - announced at ng-Europe conference Sept 2014
- Angular is not a version upgrade
  - it is a complete rewrite
  - with significant differences from Angular 1.x
- Angular 2+
  - website: angular.io
  - just called 'Angular'
- Angular 1.x
  - old website: https://angularjs.org/
  - referred to as 'Angular 1' or 'AngularJS'

Angular Releases

The schedule looks like this:

VERSION	DATE
4	March 2017
5	October 2017
6	May 2018
7	September/October 2018
8	March/April 2019

VERSION DATE

- Aim for backward compatibility with Angular 2
- Further releases about every 6 months
- See the
  - blog post
  - Release Schedule

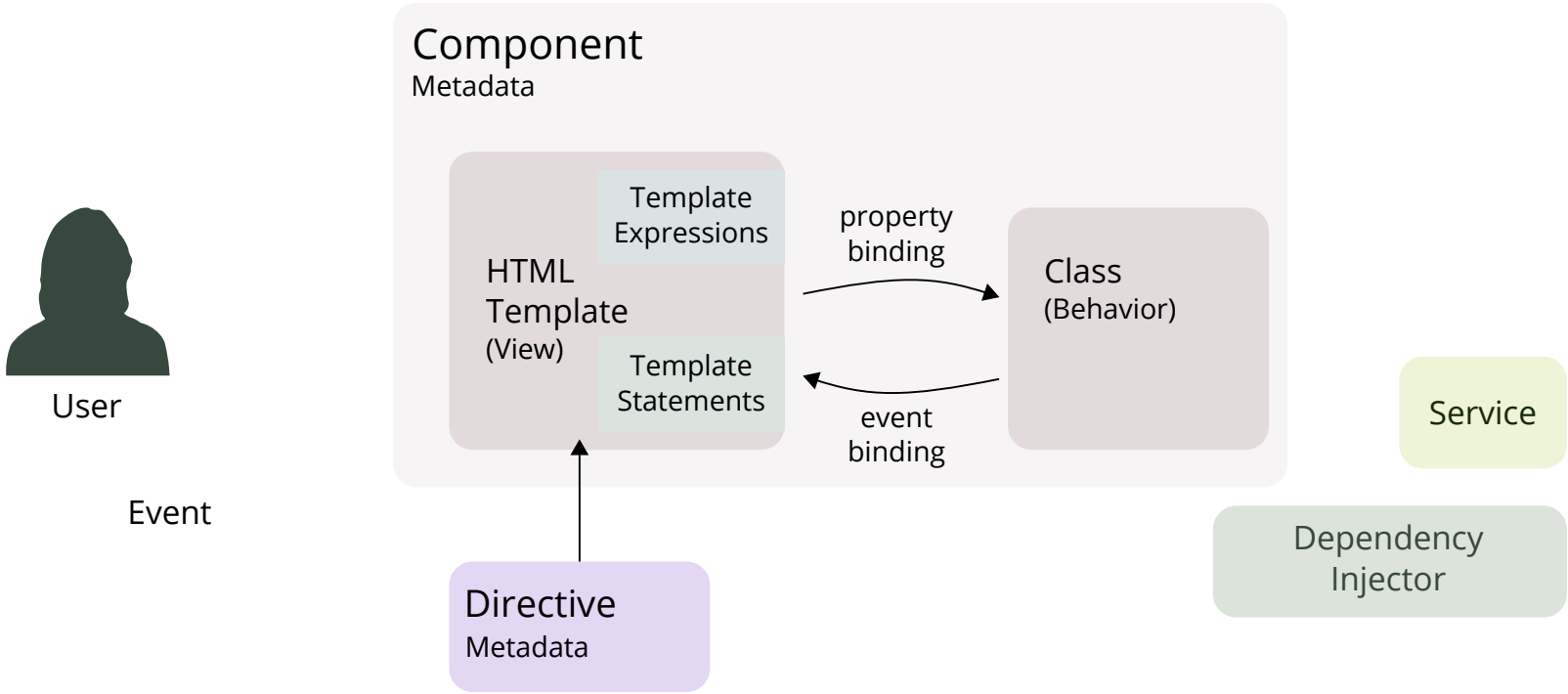
Angular Language Choice

- Angular allows a choice of languages
  - TypeScript (default and preferred)
  - ES6/ES5 JavaScript
  - Dart
- Typed superset of JavaScript
  - compiles to plain JavaScript
  - addresses many weaknesses of JavaScript
  - supports compile time type checking
  - `.ts` file extension used with TypeScript
- See http://www.typescriptlang.org/

Getting Started with Angular

Angular Fundamentals

Overview of Angular



Using Angular CLI to Scaffold a Project

Angular CLI

- makes developing Angular apps much easier
- provides development environment with short feedback loop
- can generate production builds too
- provides support for testing

Useful Resources

- Angular
  - https://angular.io/docs/ts/latest/
- Angular Tutorials
  - https://angular.io/docs/ts/latest/tutorial
  - http://www.tutorialspoint.com/angular2
- Angular API
  - https://angular.io/docs/ts/latest/api
- Curated list of resources
  - https://www.npmjs.com/package/awesome-angular2

Exercise

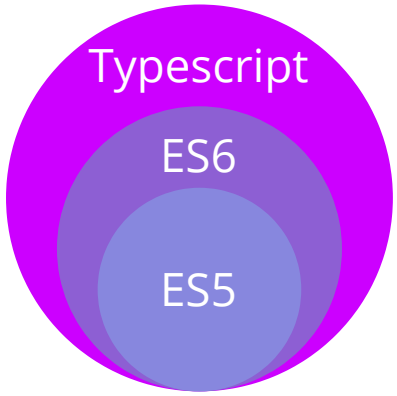
Getting Started

Typescript Primer

Angular Fundamentals

What's Typescript?

TypeScript is a typed superset of JavaScript:



TypeScript is typically compiled to JavaScript:

Typescript's Goals

TypeScript goals:

- Provide an optional type system for JavaScript
- Provide planned features from future JavaScript editions to current JavaScript engines

Why TypeScript?

1. Introduces a compile-time step
    - Helps catch errors earlier
2. Type Checking
    - Ditto
3. Edit-Time Support
    - IDEs and Code Editors have plugins
    - Syntax checking and Intellisense

Compiling Plain JavaScript - 1

Even with plain JavaScript, the TypeScript compiler can reveal errors:

JavaScript

```
let s = 'hello';
let i = s + 3;
i++;
```

```
$ tsc why-typescript-1.ts
why-typescript-1.ts(3,1): error TS2356: An arithmetic operand must be of type 'any', 'nu
$
```

Compiling Plain JavaScript - 2

TypeScript even understands the difference between mouse events and key events:

JavaScript

```
let el = document.getElementById('input1');

el.addEventListener('mousedown', event => {
  console.log(event.key);
});
```

```
$ tsc why-typescript-2.ts
why-typescript-2.ts(4,21): error TS2339: Property 'key' does not exist on type 'MouseEve
$
```

Type Inference

We haven't explicitly specified the types we're using

- TypeScript is *inferring* the types from the code

IDE Support

If your IDE supports TypeScript, you can catch errors even earlier:

Installing TypeScript

To install:

```
$ npm install -g typescript
```

Check version:

```
$ tsc -v
Version 2.4.1
```

Compiling TypeScript

To compile:

```
$ tsc app.ts                    # single file
$ tsc main.ts worker.ts         # multiple files
$ tsc main.ts worker.ts --watch # multiple files with change watcher
```

- Many more options available - see `tsc -h`

Compiling TypeScript - Options

Here's an example in which we specify two options:

```
$ tsc --target ES5 --experimentalDecorators myapp.ts
```

- `target ES5` tells the compiler to generate ES5 code
- `experimentalDecorators` enables language support for decorators

Compiling TypeScript - `tsconfig.json`

There may be many options

- To make it easier, you can specifiy them in `tsconfig.json`
- This one corresponds to the previous example:

```
{
  "compilerOptions": {
    "target": "ES5",
    "experimentalDecorators": true
  }
}
```

- Place your `tsconfig.json` at the root of your source tree

A more complex `tsconfig.json`

```
{
  "compilerOptions": {
    "module": "system",
    "noImplicitAny": true,
    "removeComments": true,
    "preserveConstEnums": true,
    "outDir": "../dist/out-tsc",
    "sourceMap": true
  },
  "include": [
    "src/**/*"
  ],
  "exclude": [
    "node_modules",
    "**/*.spec.ts"
  ]
}
```

Type Annotations

Your JavaScript is TypeScript

- Plain JS is still emitted, even with type errors
  - Old code still works
  - You can add TypeScript to your codebase progressively

Declarations and Declaration Files

Many libraries are written in pure JavaScript

- for these, we can use 'type definition files'
  - they supply types for third party libraries
- a huge number are available at DefinitelyTyped
  - to consume, see consumption

Declaring Variables

To declare a variable, we can do one of these:

- Declare the type and value in one statement:

```
let s: string = 'hello';
```

- Declare the type only:

```
let s: string;      // value is set to undefined
```

- Declare value only:

```
let s = 'hello';    // type is inferred from value
```

- Declare neither value nor type:

```
let s;              // type is any; value is undefined
```

Classes

To define a class:

```
class Point {
  x: number;
  y: number;
  constructor(x: number, y: number) {
    this.x = x;
    this.y = y;
  }
  add(point: Point) {
    return new Point(this.x + point.x, this.y + point.y);
  }
}
```

And to use it:

JavaScript

```
let p1: Point = new Point(0, 10);
let p2 = new Point(10, 20);
let p3 = p1.add(p2);
```

Static Members

TypeScript supports the proposed static members:

```
class Point {
  static instances = 0;
  x: number;
  y: number;
  constructor(x: number, y: number) {
    Point.instances++;
    this.x = x;
    this.y = y;
  }
  add(point: Point) {
    return new Point(this.x + point.x, this.y + point.y);
  }
}
```

Inheritance

We can inherit from a base class like this:

```
class Point3D extends Point {
  z: number;

  constructor(x: number, y: number, z: number) {
    super(x, y);
    this.z = z;
  }
  add(point: Point3D) {
    var point2D = super.add(point);
    return new Point3D(point2D.x, point2D.y, this.z + point.z);
  }
}
```

Interfaces

- TypeScript extends JavaScript and supports interfaces:

JavaScript

```
interface ClockInterface {
  currentTime: Date,
  setTime(d: Date): void
}

class Clock implements ClockInterface {
  currentTime: Date;
  constructor(d: Date) {
    currentTime = d;
  }
  setTime(d: Date) {
    currentTime = d;
  }
}
```

Access Modifiers

TypeScript supports a variety of access modifiers:

```
class Foo {
  public x: number;
  private y: number;
  protected z: number;
}
```

MODIFIER	ACCESSIBLE WHERE?
`public`	Anywhere (default)
`private`	Only within the class
`protected`	Within the class and its subclasses

MODIFIER ACCESSIBLE WHERE?

Constructor Parameter Properties

Instead of this...

```
class Person {
  private firstName: string;
  private lastName: string;

  constructor(firstName: string, lastNa
    this.firstName = firstName;
    this.lastName = lastName;
  }
}
```

- If you allows this:

```
class Person {
    constructor(private firstName: stri
    }
}
```

prefix a constructor argument with an accessibility modifier, TypeScript creates the property and initializes it from the argument

Generics

You can define generic classes and functions like this:

```
class MyClass<T> {
 public count: T;
 print(x: T) {
    console.log(x);
 }
}
```

Now to instantiate it:

JavaScript

```
let gn = new MyClass<number>();
gn.count = 0;
```

Generic Constraints

You can also specify constraints:

```
class Logger<T extends Lengthwise>(arg: T): T {}
```

- This says that `Logger` must be instantiated with a type or subtype of `Lengthwise`

Enum

Enumerated types let you assign identifiers to numeric constants:

```
enum Direction {
  Up, Down, Left, Right
}
let x = Direction.Up;
```

```
enum Compass{
  North = 0, East = 90, South = 180, West = 270
}
let y = Direction.North;
```

Functions

```
function add(firstName: string, lastName: string): string {
  return firstName + " " + lastName;
}
```

Functions - Optional Parameters

- In JS, parameters are optional
  - In TS, they're mandatory unless nullable
- optional parameters are postfixed with '?':

```
function add(firstName: string, lastName?: string): string {
  return firstName + (lastName ? " " + lastName : '');
}
```

Functions - Default Parameters (ES6)

Functions can declare default parameter values:

JavaScript

```
function hello(who: string = 'world') {
  console.log(`Hello, ${who}!`);
}
```

Functions - Rest Parameters (ES6)

- Rest parameters
  - allow a variable length set of arguments

Arrow Functions

Arrow Functions are similar to lambdas in other languages:

JavaScript

```
let numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9];
let oddNumbers = numbers.filter(number => {
  return number % 2;
});
```

Or more simply:

JavaScript

```
let numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9];
let oddNumbers = numbers.filter(number => number % 2);
```

Functions as Parameters

- Functions can be passed as parameters to other functions
    - A function that accepts or returns another function is called a *Higher Order Function*

Here's an example of how such a parameter is defined in TypeScript:

```
function runner(func: (n: string) => void) {
  func("Hello World!");
}
```

Decorators

Decorators enable you to annotate and modify classes, functions and properties:

A decorated class:

```
@sealed
class Greeter {
  ...
}
```

The decorator definition:

JavaScript

```
function sealed(constructor: Function)
    Object.seal(constructor);
    Object.seal(constructor.prototype);
}
```

Decorators can be attached to class declarations, methods, accessors, properties, or parameters

- Decorators can also take parameters:

```
@isTestable(true)
class MyClass { }
```

Duck Typing

- Question: are two different types *type compatible*?
  - Can I use one type in a place expecting the other?
- TypeScript offers "duck typing" or *structural typing*
  - We say types are related to each other if they have common members

Example

```
interface Action {
  type: string
}
```

```
let a: Action = {
  type: "literal"
};

class NotAnAction {
  type: string;
  constructor() {
    this.type = "Constructor function (
  }
}
a = new NotAnAction(); // valid TypeScr
```

Constants

Constant values can't be reassigned:

```
const MAX: number = 100;
```

- you must provide a value
- `const` is block scoped

With an object, it's the variable reference that's constant, not the object:

JavaScript

```
const p = new Person('John');
p = new Person('Bob'); // ERROR
```

- Can't reassign `p`

JavaScript

```
const p = new Person('John');
p.name = 'Bob'; // Legal
```

- But can change properties of the object `p` refers to

TypeScript Resources

Useful Resources

- TypeScript Tutorial
- ECMAScript 6 (ES 6)
- Quick Start Guide
- Dr Dobbs TypeScript Introduction

Components

Angular Fundamentals

Views and Templates

Defining a Component

To define a component:

- create a component class
- specify metadata by decorating it with `@Component`:

app.component.ts

```
import { Component } from "@angular/core";

@Component({
  selector: "app-root",
  templateUrl: "./app.component.html",
  styleUrls: ["./app.component.css"],
})
export class AppComponent {
  message = "App works!";
}
```

app.component.html

Creating and Adding a New Component

To create and add a new component to your app:

1. Create the component class file
    - For example, `mything.component.ts`
    - Write the component and template
2. Add the component to your application's module
3. Insert the component's selector where you want that component to appear

Common `@Component` Properties

PROPERTY	DESCRIPTION
`selector`	css selector that identifies this component in a template
`providers`	list of providers available to this component and its children
`template`	inline-defined template for the view
`templateUrl`	url to an external file containing a template for the view
`styles`	inline-defined styles to be applied to this component's view
`styleUrls`	list of url sheets to be applied to this component's view
`exports`	list of components to export from module
PROPERTY	DESCRIPTION

```
@Component({
  selector: "app-root",
  templateUrl: "./app.component.html",
  styleUrls: ["./app.component.css"],
  providers: [AppStateService]
}) export class AppComponent { ... }
```

Component Templates

Templates can be defined either:

- inline

JavaScript

```
@Component({
  selector: 'app-root',
  template: `
    <h2>App Root</h2>
    <div>{{ name }}</div>
  `
})
export class AppComponent {...}
```

- Or externally in a separate file

```
@Component({
  selector: 'app-root',
  templateUrl: './app.component.html'
})
export class AppComponent {...}
```

- app.component.html

```
<h2>App Root</h2>
<div>{{ name }}</div>
```

Component View Styles

Styles can be defined either:

- Inline within the Component declaration
  - via the `styles` property

JavaScript

```
@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styles: [`
  .selected {
    background-color: #CFD8DC;
    color: white;
  }
  `]
})
export class AppComponent {...}
```

- Externally to the Component
  - via the `styleUrls` property

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

Components and Modules

- A component must belong to an `NgModule`
  - Place it in the `declarations` property

```
import { BrowserModule } from '@angular/platform-browser';
import { NgModule } from '@angular/core';
import { FormsModule } from '@angular/forms';

import { AppComponent } from './app.component';

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

Components and Data Members

- Components can define data members
  - may be variable or constant
  - that can be referenced within a template – can be used with one or two way binding

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

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html'
})
export class AppComponent  {
    name = 'John';
}
```

app.component.html

HTML

```
<h1>Welcome {{name}}</h1>
<p> Hello {{name}} </p>
```

Property Binding

Angular Fundamentals

What's Data Binding?

Data Binding connects the UI to the business logic

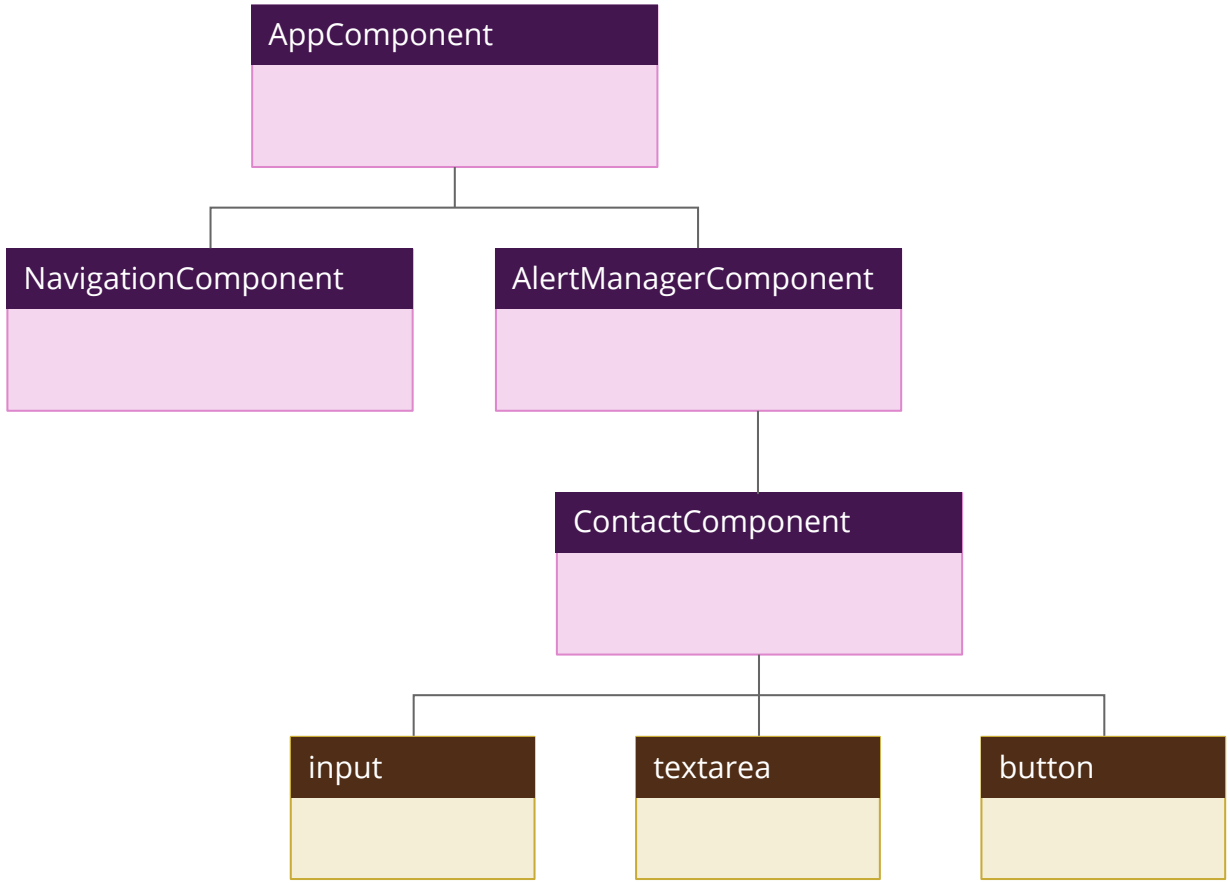
- We avoid manual DOM updates
  - Instead, we declare our needs to the *data binding layer*
- We just say "bind this UI thing to that data thing"
  - When data changes, the framework ensures the UI showing that data is kept in sync

Property Binding

- A component or DOM element exposes *properties*
  - Properties are often initialized from *attributes* in HTML

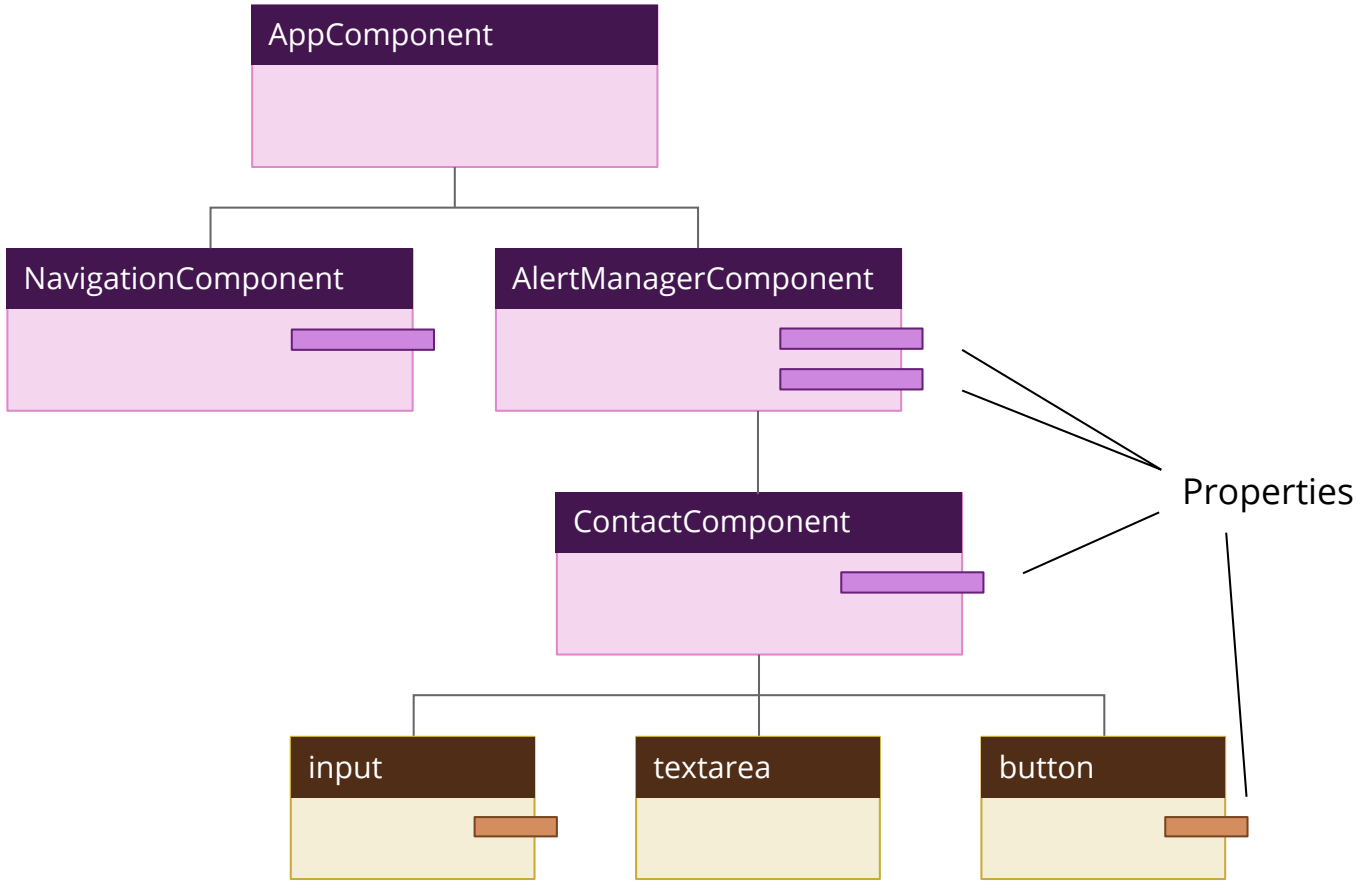
The Component Hierarchy

As you build your app with components, a hierarchy forms:



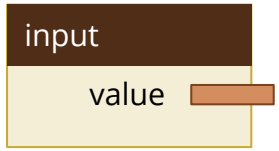
Properties

Each component or DOM element has *properties*:



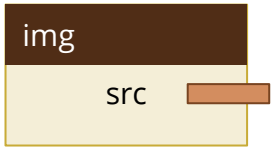
Examples of Properties

An `<input>` element has an in-memory `value` property:



- it contains the value currently entered in the input box

An `<img>` element has an in-memory `src` property:



- the image is displayed from the URL provided to the `src` property

HTML attributes vs. DOM properties

- We don't bind to HTML attributes
  - We bind to DOM properties instead

HTML ATTRIBUTE	DOM PROPERTY
Exists in HTML page/template	Exists 'live' in browser memory (DOM)
Specifies *initial* value	Holds current value
Can't be updated over time	Changes over app's lifetime

Example

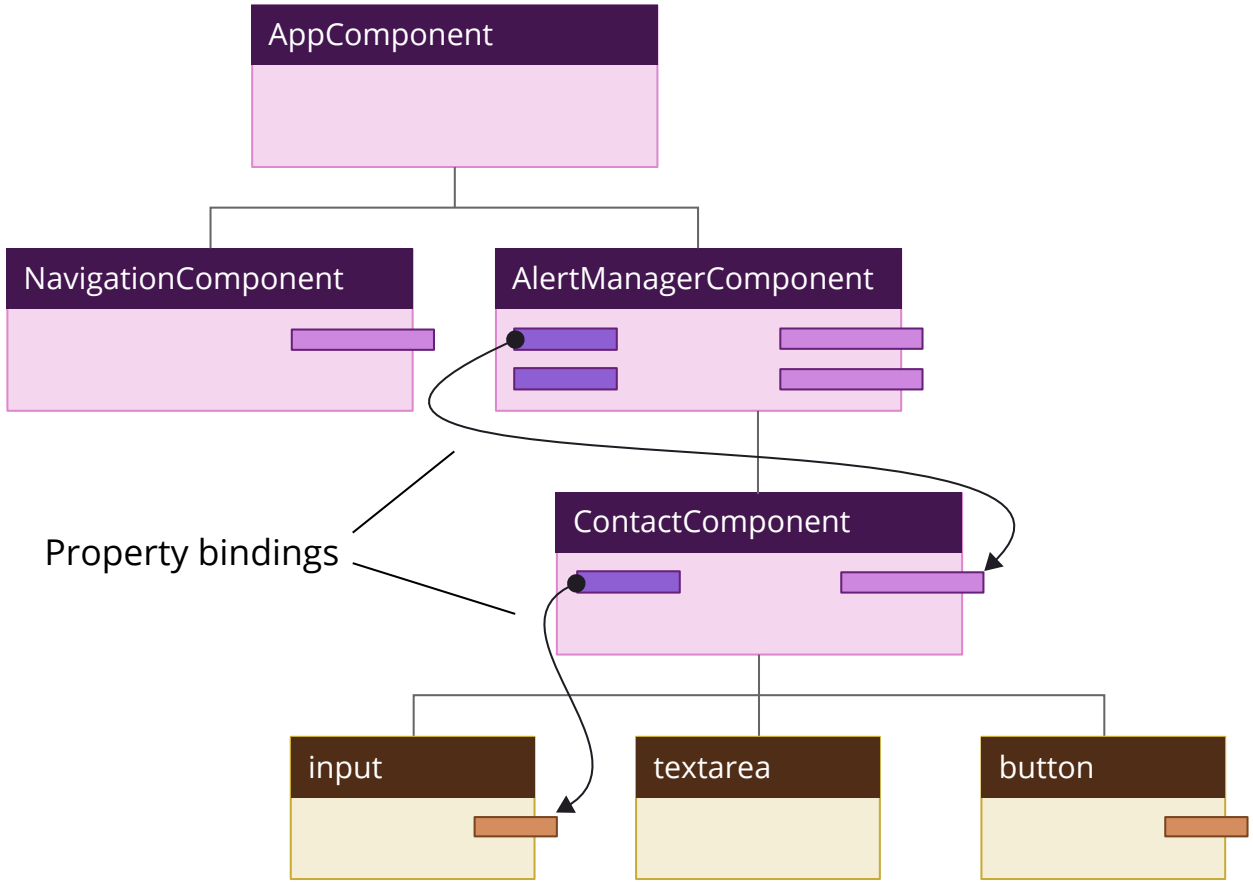
HTML

```
<input type="text" value="Dave">
```

- Attribute `value` just initializes the same property, `value`
- Typing text into the input box updates the `value` property, but not the `value` attribute

Property Bindings

Property bindings enable our components to pass data to each other:



The Component

The (simple) component looks like this:

JavaScript

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

@Component({
  selector: 'my',
  template: `
    <a [href]="site.url" [textContent]="site.name"></a>
  `,
})
export class MyComponent {
  site = {
    name: 'Internet Movie Database',
    url: 'https://www.imdb.com/'
  };
}
```

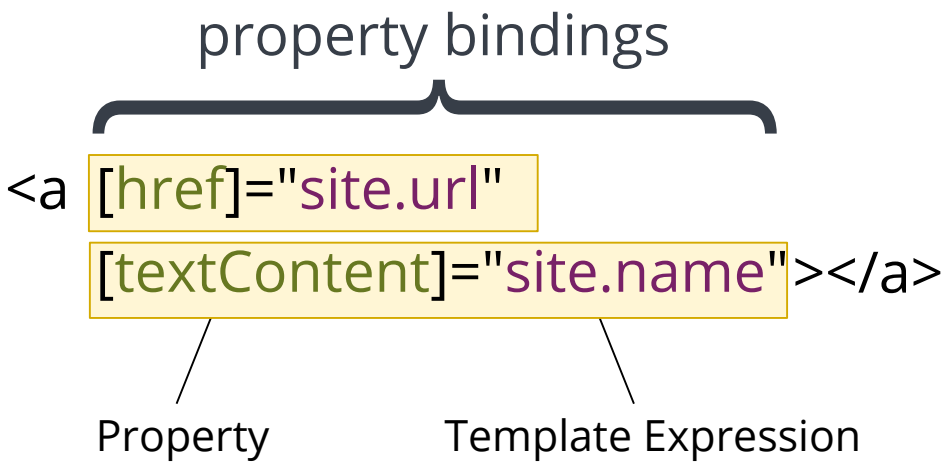
Specifying a Property Binding -- 1

This anchor tag specifies two *property bindings*:

```
<a  [href]="site.url"
    [textContent]="site.name"></a>
```

Specifying a Property Binding -- 2

These bind a property to a *template expression*:

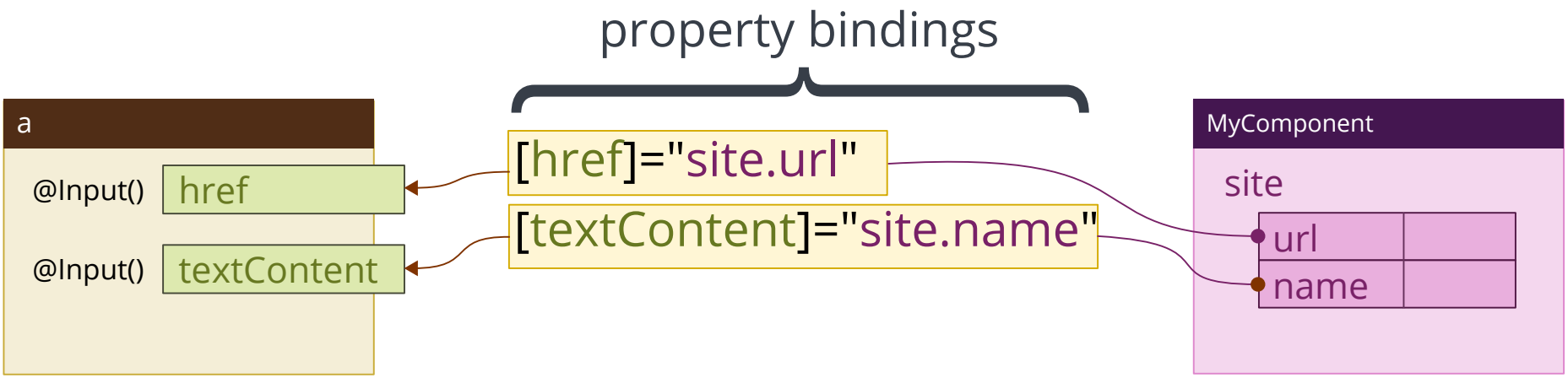


Property A property is in-memory and belongs to a component or DOM element

Template Expression A template expression is similar to a JavaScript expression

Specifying a Property Binding -- 3

Here are the two items that are bound:

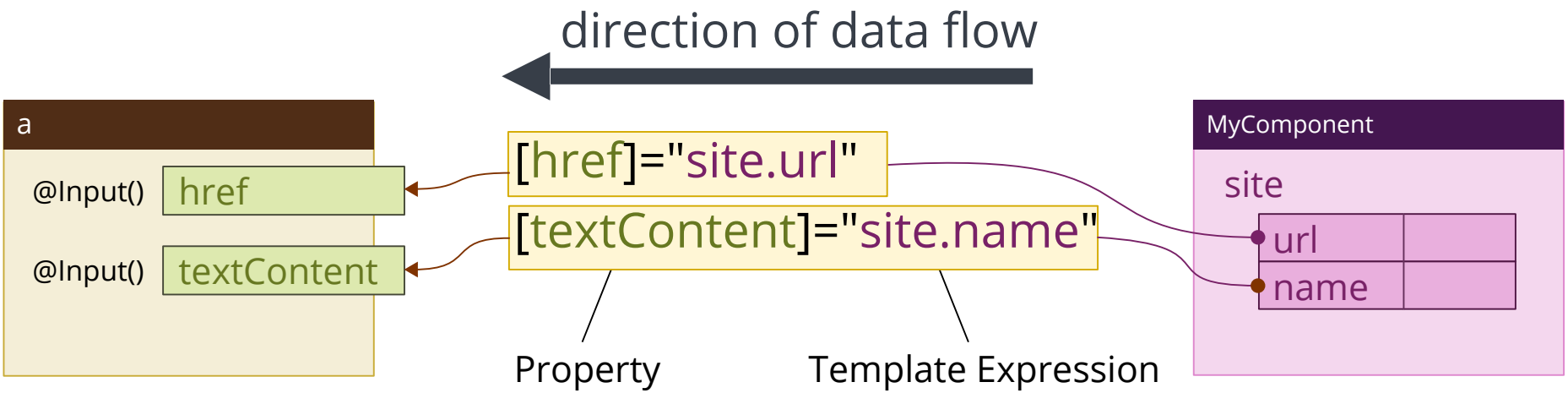


This says:

- have the `href` property always reflect the current value of *MyComponent*'s `site.url` value; and
- have the `textContent` property (the visible link) always reflect the current value of *MyComponent*'s `site.name` value

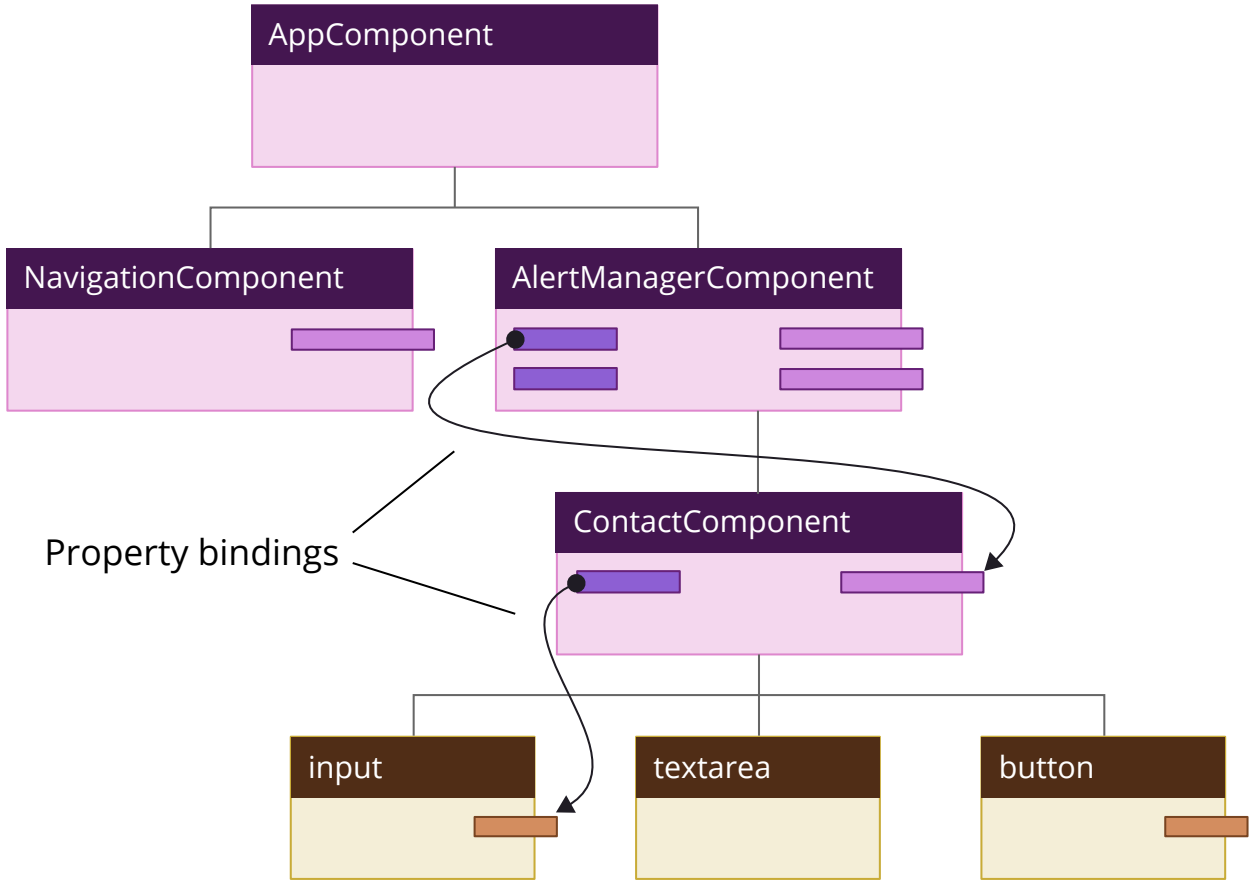
Specifying a Property Binding -- 4

Data flows from component to a target element or component:



Data Flows Downwards

So property bindings are a way of passing data downwards, from parent components to their children:



More Examples of Property Binding

Have an image source synchronized to a URL:

```
<img [src]="myPhotoUrl">
```

Bind a blog post title to a heading's contents:

```
<h1 [textContent]="blogpost.title"></h1>
```

Only enable a submit button when the `isValid` flag is true:

```
<button [disabled]="!isValid" type="submit">Submit</button>
```

Provide a `Person` object to our `ContactDetailsComponent`:

```
<contact-details [person]="selectedPerson"></contact-details>
```

"bind-" Syntax

As an alternative, Angular also supports this syntax:

```
<img bind-src="myPhotoUrl">
```

This is equivalent to:

```
<img [src]="myPhotoUrl">
```

Interpolation vs Property Binding

- Angular converts interpolation into property binding
- However:
    - Interpolation can be very expressive
    - It only converts to *strings*

In this example, these do the same thing:

```
<img src="{{myPhotoUrl}}">
<img [src]="myPhotoUrl">
```

Types of Property Binding

There are three types of property binding. 1. Element property 2. Directive property 3. Component property

Kinds of Data Binding

Angular supports these kinds of data binding:

DIRECTION	DATA UPDATES	SYNTAX	TYPE
One-way	From data source to view target	`{{template expression}}` `[target]=expression`	Interpolation Property Attribute Class Style
One-way	From view target to data source	`(target)="statement"` `on-target="statement"`	Event
Two-way		`[(target)]="expression"`	Two-way
DIRECTION	DATA UPDATES	SYNTAX	TYPE

Handling Events

Angular Fundamentals

Handling Button Click Events

To handle a button click event:

JavaScript

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

@Component({
  selector: 'alert-button',
  template: `
    <button (click)="showAlert()">Alert me</button>
  `,
})
export class AlertButtonComponent {
  showAlert() {
    alert('Button clicked');
  }
}
```

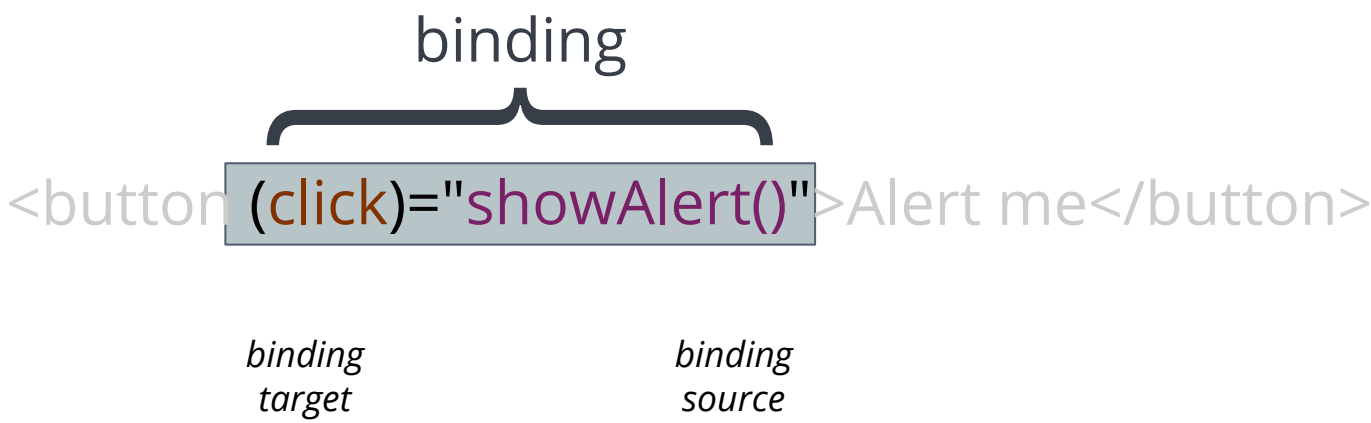
Data Binding -- 1

In this snippet from the previous slide:

```
<button (click)="showAlert()">Alert me</button>
```

Data Binding -- 2

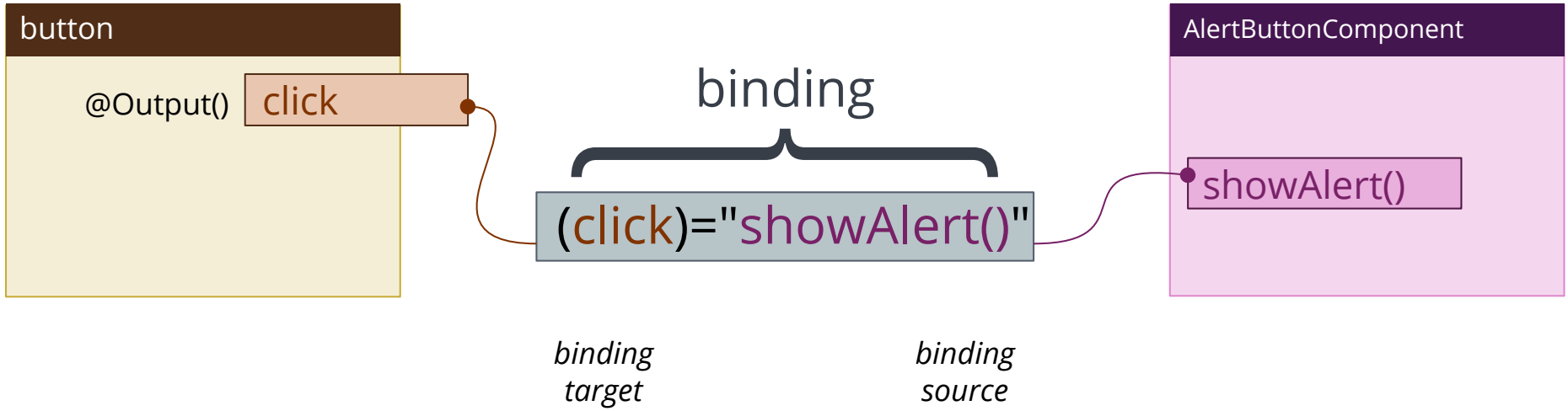
This expression is a *binding*:



- the `(click)="showAlert()"` is a *data binding*
  - more specifically, an *output binding*:

Data Binding -- 3

It means "bind this target to this source":



This means:

- "when the button raises a click event..."
  - "call the showAlert() method in this component's class"

The `$event` Parameter

To pass data to the method, we use the special `$event` parameter:

JavaScript

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

@Component({
  selector: 'alert-button',
  template: `<button (click)="showAlert($event)">Alert me</button>`,
})
export class AlertButtonComponent {
  showAlert(e) {
    alert('Button clicked');
    console.log(e);
  }
}
```

- The `$event` name refers to the data emitted by the binding target

Common DOM Events

- Common events are listed on MDN:
  - https://developer.mozilla.org/en-US/docs/Web/Events
- Your components can generate events too
  - More later...

Event Methods

Just as in the standard DOM, Angular supports the following methods:

```
myEventHandler(event) {
  event.preventDefault();
  event.stopPropagation();
}
```

- `preventDefault()` prevents the default action
  - for a `<a href="url">`, the default is 'go to url';
  - for a `<button type="submit">`, the default is to submit the form
- `stopPropagation()` stops the event propagating up the element hierarchy

Directives

Angular Fundamentals

Directives

- HTML has a finite, limited vocabulary
  - `<ul>` renders an unordered list
  - `<video>` renders a video
- Why can't we *extend* HTML to add our own, so they could:
  - modify the structure of our HTML?
  - add custom attributes to elements?
  - modify CSS styles programmatically?

What is a Directive?

- Directives enable us to extend HTML
  - add new attributes, tags etc
  - components are really one class of custom directive
- Enable reusable data and functions
  - e.g. `ngClick`
- Simplify template syntax

Categories of Directive

Three main types of directive:

Component Directive	Attribute Directive	Structural Directive
Examples	**Examples**	**Examples**
MyApp	NgStyle NgClass NgForm NgModel	NgIf NgFor

- Component Directive
  - a directive with an HTML template
- Attribute Directive
  - changes the behavior or appearance of a component or element
- Structural Directive
  - alters layout by adding, removing or replacing elements in DOM

Attribute Directives

Angular Fundamentals

Attribute Directive

- Changes appearance or behavior of native DOM elements
  - Typically independent of the DOM element
- Examples of attribute directives
  - `ngClass`
  - `ngStyle`
  - both work on any element
- May be applied in response to user input, service data etc.

`NgStyle` Directive

- Built in directive
  - used to modify the style attribute of an element
- Can be used within a component's template
  - can bind a directive in similar manner to a property
  - e.g. `<p [ngStyle] ="{'color': 'blue'}">Some test</p>`
- Can generate style information from component data
  - allows style to change dynamically
  - e.g. `<p [ngStyle] ="componentStyles">Some test</p>`

```
export class StyleExampleComponent {
  borderStyle = 'black';

  componentStyles = {
    color: 'blue',
    'border-color': this.borderStyle,
  };
}
```

Styling Content -- 1

This content is rendered by the child component:

JavaScript

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

@Component({
  selector: 'app-root',
  template: `
    <app-style-example>Style me!</app-style-example>
  `
})
export class AppComponent {}
```

- To render the content, use `<ng-content></ng-content>`

Styling Content -- 2

JavaScript

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

@Component({
  selector: 'app-style-example',
  template: `
    <p [ngStyle]="alertStyles">
        <ng-content></ng-content>
    </p>`
})
export class StyleExampleComponent {
  borderStyle = 'red green blue yellow';

  alertStyles = {
    color: 'blue',
    'border-style': 'solid',
    'border-color': this.borderStyle
  };
}
```

`NgClass` Directive

- Changes the class attribute bound to an element
  - or a component
- Can bind a string directly to the attribute
  - similar to adding a HTML class attribute
  - here used to wrap some `ng-content`

`NgClass` Directive

- You can apply a set of CSS classes from an array:

JavaScript

```
@Component({
  selector: 'app-class',
  template: `
    <p [ngClass]="['warning', 'big']">
      <ng-content></ng-content>
    </p>`
})
export class StylingUsingArrayDemo {}
```

NgClass Directive

- By using a JS object, you can apply CSS classes conditionally:

JavaScript

```
@Component({
  selector: 'app-class',
  template: `
    <p [ngClass]="mystyles">
      <ng-content></ng-content>
    </p>`
})
export class StylingUsingObjectDemo {
  flat: boolean = true;
  mystyles = { card: true, dark: false, flat: flat };
}
```

Structural Directives

Angular Fundamentals

Structural Directives

Structural Directives modify the DOM structure

For example:

- to display content depending on a condition
- to repeat content, displaying one piece per item in a list

`ngIf` Example

This paragraph will render only if the condition is truthy:

HTML

```
<p *ngIf="artist === 'cezanne'">
  Don't be an art critic, but paint, there lies salvation
</p>
```

Built-in Structural Directives

These structural directives are available:

SELECTOR	CLASS	DESCRIPTION
`ngIf`	`NgIf`	Conditionally render content depending on an expression
`ngFor`	`NgForOf`	Render a piece of content once per item of an iterable
`ngSwitch`	`NgSwitch`	Render a piece of content depending on which expression matches

Using `*ngIf` with an `else` clause

Angular 4 supports an `else` clause:

```
<div *ngIf="userObservable | async; else loading; let user">
  Hello {{user.last}}, {{user.first}}!
</div>
<template #loading>Waiting...</template>
```

Iterating with `*ngFor`

- `*ngFor` iterates over an array or any iterable

JavaScript

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

@Component({
  selector: 'ngfor-demo',
  template: `
    <ul>
      <li *ngFor="let fruitItem of fruit">
        {{fruitItem}}
      </li>
    </ul>
  `,
})
export class NgForDemoComponent {
  fruit = ['apple', 'banana', 'custard apple', 'durian', 'fig'];
}
```

`*ngFor` - Importing `CommonModule`

To use `*ngFor` in non-root modules you must import `CommonModule`:

```
...
import { CommonModule } from '@angular/common';
...

@NgModule({
  ...
  imports: [
    CommonModule,     // <-- import in non-root modules
  ],
  ...
})
export class MyModule { }
```

Further `*ngFor` Features

index

Angular exposes the `index` property, which numbers the items from zero:

HTML

```
<li *ngFor="let fruitItem of fruit; let i = index">
  Item {{i}} is: {{fruitItem}}
</li>
```

count

The `count` property maintains the total count of items:

HTML

```
<li *ngFor="let fruitItem of fruit; let i = index; let c = count">
  Item {{i}} is: {{fruitItem}} (count is: {{c}})
</li>
```

Even more `*ngFor` Features

Further properties are available too:

PROPERTY	DESCRIPTION
`first`	True when the item is the *first* item in the iterable
`last`	True when the item is the *last* item in the iterable
`even`	True when the item has an *even index* in the iterable
`odd`	True when the item has an *odd index* in the iterable

Using *ngFor to Style Alternate Rows

To obtain this effect:

Item 0 is: apple [true, false]
Item 1 is: banana [false, true]
Item 2 is: custard apple [true, false]
Item 3 is: durian [false, true]
Item 4 is: fig [true, false]

*ngFor can be used like this:

HTML

```
<li
  *ngFor="let fruitItem of fruit; let i = index; let e = even; let o = odd;"
  [ngClass]="{
    'odd-row': o,
    'even-row': e
  }"
>
  Item {{i}} is: {{fruitItem}} [{{e}}, {{o}}]
</li>
```

Microsyntax

In this expression:

HTML
```
<li *ngFor="let fruitItem of fruit; let i = index; let e = even; let o = odd;" ...>
```

- the expression in quotes is in a "microsyntax"
- the `let i`, `let o` etc declare *template input variables* that you can reference within the template
- the `NgFor` directive maintains a "context object" which it updates as it iterates, which is where `index`, `even` etc come from

Conditional Rendering using `ngSwitch`

- Actually two directives:
  - `ngSwitch` (an attribute directive) indicates the condition
  - `*ngSwitchCase` (a structural directive) indicates options
- Suppose a component has a property called `tab`
  - selects which `<app-tab>` element to render
  - `<app-tab>` is an element indicating a custom (tab) Component

HTML

```
<div [ngSwitch]="tab">
  <app-tab *ngSwitchCase="1">Tab 1</app-tab>
  <app-tab *ngSwitchCase="2">Tab 2</app-tab>
  <app-tab *ngSwitchCase="3">Tab 3</app-tab>
  <app-tab *ngSwitchDefault>Select a tab</app-tab>
</div>
```

Explaining the * Syntax

Angular Fundamentals

The <ng-template> Element

The <ng-template> element looks like this:

HTML

```
<ng-template [ngIf]="artist === 'picasso'">
  <p>
    Art is a lie that makes us realize truth
  </p>
</ng-template>
```

The <ng-template> element:

- doesn't render its content on its own
  - it's just a way of holding content until it is rendered
  - Angular 2 used <template>

The `template` Attribute Directive

The `template` attribute directive allows us to do the same thing:

HTML

```
<div template="ngIf artist === 'picasso'">
  <p>
    Art is a lie that makes us realize truth
  </p>
</div>
```

Translating the * Syntax -- 1

This structural directive...

HTML

```
<p *ngIf="artist === 'picasso'">
  Art is a lie that makes us realize truth
</p>
```

Translating the * Syntax -- 2

first translated to a `template` attribute directive...

HTML

```
<div ng-template="ngIf artist === 'picasso'">
  <p>
    Art is a lie that makes us realize truth
  </p>
</div>
```

Translating the * Syntax -- 3

finally to an <ng-template>:

HTML

```
<ng-template [ngIf]="artist === 'picasso'">
  <p>
    Art is a lie that makes us realize truth
  </p>
</ng-template>
```

Translating the * Syntax -- 4

So the `*` in `*ngFor` is "syntactic sugar"

These two are equivalent:

HTML

```
<ng-template [ngIf]="artist === 'picass
  <p>
    Art is a lie that makes us realize
  </p>
</ng-template>
```

- The `*`

HTML

```
<p *ngIf="artist === 'picasso'">
  Art is a lie that makes us realize tr
</p>
```

means "use the element this directive is attached to *as* the template"

Two-way Binding

Angular Fundamentals

What's "Two-way Binding"?

Two-Way Binding

- updates the display when a property changes; and
- updates a property when a user input changes

Angular 2+

- supports one-way binding only
- emulates the "two-way" binding that was present in AngularJS (v1)
    - it actually uses two one-way bindings

Using "Two-Way" Binding

We can use two-way binding "manually", like this:

HTML

```
<p>You have chosen the username: {{username}}</p>
<input [value]="username" (input)="username = $event.target.value">
```

And in our component:

```
...
export class MyComponent {
  username: string = '';
}
```

- Notice that this is actually two *one-way* bindings

The "Banana-in-a-Box" Syntax

Angular offers a shorter syntax for two-way binding:

HTML

```
<p>You have chosen the username: {{username}}</p>
<input [(ngModel)]="username">
```

- Notice that we use the `ngModel` directive
  - The `NgModel` class maintains a model

The `NgModel` Directive

The "Banana-in-a-Box" is actually equivalent to this:

HTML

```
<p>You have chosen the username: {{username}}</p>
<input [ngModel]="username" (ngModelChange)="username = $event">
```

- The `NgModel` class ensures that updates happen:
  - from model to view (component to template, or `username` to `<input>`)
  - via `[ngModel]="username"`
  - from view to model (template to component, or `<input>` to `username`)
  - via `(ngModelChange)="username = $event"`
  - The `NgModel` class fires an `ngModelChange` event when the model changes

Pipes

Angular Fundamentals

Using Pipes

- Pipes enable data in template expressions to be manipulated
- To use one or more pipes, specify their names in a template expression:

JavaScript

```
@Component({
  selector: 'cats',
  template: `<p>8 out of 10 cats is {{ cats | percent }}</p>`
})
export class CatsComponent {
  cats = 8 / 10;
}
```

Output:

```
8 out of 10 cats is 80%
```

Built-in Pipes

PIPE NAME	CLASS NAME	DESCRIPTION
`currency`	`CurrencyPipe`	transforms number into desired currency
`number`	`DecimalPipe`	format a number as text
`percent`	`PercentPipe`	format a number as a percentage
`date`	`DatePipe`	displays a date in different formats
`slice`	`SlicePipe`	produce a slice or subset of a list or string
PIPE NAME	**CLASS NAME**	**DESCRIPTION**
`uppercase`	`UpperCasePipe`	converts input into *UPPER* case
`lowercase`	`LowerCasePipe`	converts input into *lower* case
`titlecase`	`TitleCasePipe`	converts input into *TitleCase*
`json`	`JsonPipe`	converts a value into a JSON string
`async`	`AsyncPipe`	display a value from an asynchronous primitive

- Pipe name is used in an HTML template e.g. `currency`, `date`
- Each pipe is implemented by a class e.g. `CurrencyPipe`, `DatePipe`

Pipe Parameters

Pipes can take parameters, and also use properties:

```
{{ addressLine1 | slice:0:15 }}     // renders first 15 characters
{{ addressLine1 | slice:0:nChars }} // renders first 'nChars' characters
```

The general form is:

```
expression | pipe1:param1:param2:... | pipe2:param1:param2:... | ...
```

Rendering Dates

The `DatePipe` offers a variety of formatting options:

TEMPLATE EXPRESSION	OUTPUT
`{{ today | date }}`	Oct 1, 2017
`{{ today | date:'medium' }}`	Oct 1, 2017, 8:37:59 PM
`{{ today | date:'shortTime' }}`	8:37 PM
`{{ today | date:'mmss' }}`	3759

The `today` variable is declared as:

JavaScript

```
today = new Date();
```

Rendering Numbers

Some number rendering examples:

TEMPLATE EXPRESSION	OUTPUT
`{{ pi | number }}`	3.142
`{{ pi | number:'3.4-7' }}`	003.1415927

Number Format Expression

- The `digitInfo` expression works like this:
  - `{minIntegerDigits}.{minFractionDigits}-{maxFractionDigits}`

PARAMETER	DESCRIPTION	DEFAULT
`minIntegerDigits`	the minimum number of integer digits to use	1
`minFractionDigits`	the minimum number of digits after fraction	0
`maxFractionDigits`	the maximum number of digits after fraction	3

Rendering Currencies

Some currency rendering examples:

TEMPLATE EXPRESSION	OUTPUT
`{{ tradeValue | currency }}`	USD151.40
`{{ tradeValue | currency:'EUR' }}`	EUR151.40
`{{ tradeValue | currency:'EUR':true }}`	€151.40
`{{ tradeValue | currency:'USD':true:'9.2-2' }}`	$000,000,151.40

General form of the expression:

```
... | currency[:currencyCode[:display[:digitInfo[:locale]]]]
```

- `currencyCode`: the currency to render
- `display`: whether to display a currency symbol (`$`, `€`, `£` etc) or the code (`USD` etc)
  - In Angular 5, `display` is a string with value `code`, `symbol` or `symbol-narrow` (`CAD`, `CA$`, `$`)
- `digitInfo`: same as for `number`

Multiple Pipes

Using multiple pipes is of course valid:

TEMPLATE EXPRESSION	OUTPUT
`{{ tradeValue | currency:'EUR' | lowercase }}`	eur151.40

Passing Data to a Component

Angular Fundamentals

Components and Behavior

- Components can define methods / behavior
  - This behavior can be invoked from the templates

```
import { Component } from "@angular/core";

@Component({
  selector: "counter",
  templateUrl: "./counter.component.html"
})
export class CounterComponent {
  count = 0;

  increment() {
    this.count++;
  }
}
```

- component.component.html

Structuring Applications

Components can be subdivided into two categories:

- Container components
  - application specific
  - higher level
  - typically have access to application's domain model
  - e.g. `LoanValidationComponent`
- Presentation Components
  - responsible for UI rendering
  - or behavior of specific entities supplied to component
  - may be more generic (reusable component)
  - e.g. `TableLayoutRenderer`

Referencing Child Components

Simplest approach: embed child tag in parent template:

- app.component.ts

JavaScript

```
import { Component } from '@angular/core';
@Component({
  selector: 'app-root',
  template: `
    <h2>Parent</h2>
    <hr />
    <child-comp></child-comp>
    <hr />
  `
})
export class AppComponent {}
```

Referencing Child Components

Simplest approach: embed child tag in parent template:

- child.component.ts

JavaScript

```
import { Component } from '@angular/core';
@Component({
  selector: 'child-comp',
  template: `
    <h3>Child Heading</h3>
    <p>Content from the child</p>
  `
})
export class ChildComponent {}
```

Passing Data to Child Components

Parent can provide inputs to child

- app.component.ts

JavaScript

```
@Component({
  selector: 'app-root',
  template: `<h1>Welcome Message</h1>
             <counter count="10"></counter>`
})
export class AppComponent {}
```

- counter.component.ts

```
@Component({
  selector: 'counter',
  template: '<p>Count: {{ count }}</p>'
})
export class CounterComponent {
```

Templates

Angular Fundamentals

Template Syntax

Templates consist of HTML containing embedded:

- template expressions (eg `{{myProperty}}`)

Interpolation Example

HTML

```
<div class="a-section">
    <a href="{{product.url}}">
      <img src="{{product.thumbnailUrl}}"
    </a>
    <h2>
      <a href="{{product.url}}">
        {{product.title}}
      </a>
    </h2>
    <p><span>by {{product.manufacturer}}<
</div>
```

Template Expressions

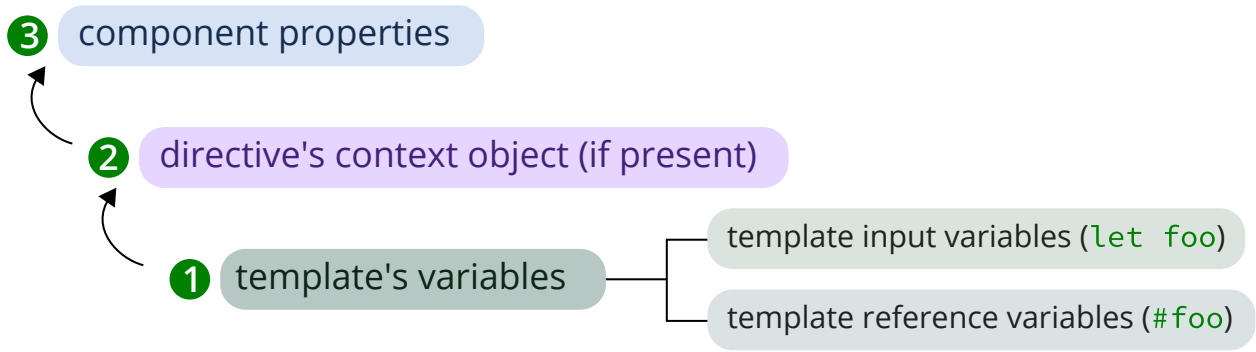
- Template expressions produce a value
  - This value is assigned to the property of a binding target

Expression Context

Where do template expressions get their values from?

- from properties of the corresponding component
- from properties of the template's *context*:
  - a *template input variable* (eg `let product`)
  - a *template reference variable* (eg `#product`)

Identifiers are looked up in this order:



Template Input Variable

- Sometimes we need a 'local' variable within our template
    - Declare it using `let myVariable`

Template:

HTML
```
<div *ngFor="let product of products">{{product.title}}</div>
```

- So `product` here is a variable only visible within the template

Output:

HTML
```
<ul>
  <li>nonda USB-C to USB 3.0 Mini Adapter Aluminum Body</li>
  <li>Apple USB-C to USB Adapter</li>
  <li>AmazonBasics USB 3.0 to 10/100/1000 Gigabit Ethernet Adapter</li>
  <li>Apple Thunderbolt Gigabit Ethernet Adapter (MD463LL/A)</li>
</ul>
```

Template Reference Variable

- Use when you want to reference a DOM element from another part of the same template
- The # syntax means:
  - Take a reference to this DOM element and store it in the variable name provided
  - This variable can now be accessed anywhere in the template
  - The variable is scoped to the template

HTML

```
<input #phone>
<p>{{phone.value}}</p>
```

- This isn't quite enough to do what we want
  - We force Angular to handle events: `<input #phone (keyup)="0">`

Template Expressions don't allow...

OPERATORS	EXAMPLE
assignments	`=`, `+=` etc
instantiation	`new FooConstructor()`
chaining expressions	`;` or `,`
increment or decrement	`++` or `--`
bitwise operators	`

Template Expressions add...

OPERATORS	EXAMPLE
Pipe operators	`` ` ``
Safe navigation operator	`?.`

Template Statements don't allow...

OPERATORS	EXAMPLE
operator assignments	`+=`, `-=` etc
instantiation	`new FooConstructor()`
increment or decrement	`++` or `--`
bitwise operators	`
template expression operators	`

- Also, they can't access globals
  - So no `window`, `document`, `Math.min` etc

Accessing Child Members

Angular Fundamentals

Accessing Child Members

- You can also reference child component members
    - e.g. data member such as instance variable
    - or behavior such as methods
    - need to provide a reference for the child
- app.component.ts

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

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html'
})
export class AppComponent {}
```

- app.component.html

Accessing Child Members

The child component is unaware of the parent's access:

profile.component.ts

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

@Component({
  selector: 'profile',
  templateUrl: 'profile.component.html'
})
export class ProfileComponent {
  name = 'Dave Smith';
}
```

profile.component.html

```
<div style="border: 1px solid black; margin-bottom: 1rem;">
  Profile Name: {{ name }}
</div>
```

Multiple Component Instances

A component may use many instances of the same child component:

app.component.ts

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

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html'
})
export class AppComponent {}
```

app.component.html

HTML

```
<div>
  <h3>Child 1</h3>
  <child></child>
  <h3>Child 2</h3>
  <child message='Welcome'></child>
```

Multiple Component Instances

Here's the child component:

child.component.ts

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

@Component({
  selector: 'child',
  template: '<p>Message is {{ message }}</p>',
})
export class ChildComponent {
  @Input() message: string = 'Hello';
}
```

HTTP and Services

Angular Fundamentals

HTTP Clients in Angular

- Angular's HTTP client has been rewritten in 4.3

ANGULAR VERSION	PACKAGE	MODULE	SERVICE
4.3 -	`@angular/common/http`	`HttpClientModule`	`HttpClient`
2.0 - 4.2	`@angular/http`	`HttpModule`	`Http`

Using `@angular/common/http`

Three steps to making a call:

1. Import the `HTTPClientModule` into your app
2. Inject `HttpClient` into your component
3. Make a request by calling `this.http.get()`

1. Import `HTTPClientModule`

Import the module like this:

```
import { BrowserModule } from '@angular/platform-browser';
import { NgModule } from '@angular/core';
import { HttpClientModule } from '@angular/common/http';  // <-- here
import { AppComponent } from './app.component';

@NgModule({
  declarations: [AppComponent],
  imports: [BrowserModule, HttpClientModule],             // <-- and here
  providers: [],
  bootstrap: [AppComponent]
})
export class AppModule {}
```

2. Inject `HttpClient`

Make `HttpClient` available to your class:

```
import { Component, OnInit } from '@angular/core';
import { HttpClient } from '@angular/common/http';

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

  constructor(private http: HttpClient){
  }
}
```

- Angular's DI framework injects the `HttpClient` instance

Example - Calling the GitHub REST API

The GitHub API offers a root-level request:

```
$ curl https://api.github.com
{
  "current_user_url": "https://api.github.com/user",
  "current_user_authorizations_html_url": "https://github.com/settings/connections/appli
  "authorizations_url": "https://api.github.com/authorizations",
  ...
}
```

Querying the GitHub API

JavaScript

```
import { Component, OnInit } from '@angular/core';
import { HttpClient } from '@angular/common/http';
import { Observable } from 'rxjs/Observable';
import 'rxjs/add/operator/map';
import './gh-api.type';

@Component({
  selector: 'http-demo',
  template: `
  <p>http-demo</p>
  <table>
    <tr *ngFor="let entry of result">
      <td>{{ entry[0] }}</td>
      <td>{{ entry[1] }}</td>
    </tr>
  </table>
  `,
})
```

Querying the GitHub API - 2

JavaScript

```
export class HttpDemoComponent implements OnInit {
  result: [string, string][];

  constructor(private http: HttpClient){
  }

  ngOnInit(): void {
    this.http
      .get('https://api.github.com/')
      .map(obj => Object.entries(obj))
      .subscribe(data => {
        this.result = data;
      })
  }
}
```

Routing

Angular Fundamentals

Why Routing?

We need routing in order to:

- maintain application state
- distinguish between logically separate areas of our app
- allow access to areas based on authorization rules

In addition, routing enables us to:

- bookmark a 'deep' location within an app and return to it later
- share such a URL with others

Client-side Routing Mechanisms

Two techniques are commonly used. With both:

- Page is not reloaded
- Forward and back buttons work

Hash-based Routing

- Anchor name to identify content: `<a name="about"><h1>About</h1></a>`
- Fragment identifier to go to that content: `http://example.com/#/about`

HTML5 History API

- Uses `history.pushState()` and friends to navigate
  - `history.pushState(stateObj, "page 2", "bar.html");`

Routing in Angular

- Angular Router
  - maps URL fragments to components, with or without state
- Routing is supported by `RouterModule`
  - defined in `@angular/router`
- When a route is selected:
  - any previous view is replaced by new view
  - content area is represented by `<router-outlet>`
- Routes defined by the `Routes` array
  - array of routing objects linking paths to components
  - can define a default path

Using the `RouterModule`

- An app can import the `RouterModule` multiple times
  - Once per lazily-loaded bundle
- But there is only ever *one* active router service (singleton)
  - router deals with a global shared resource-location
- Two ways to access the RouterModule...

Using the `RouterModule`

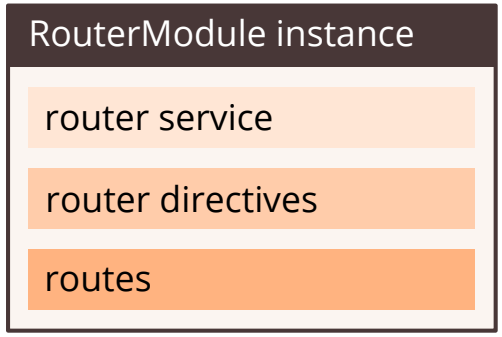
To use within the root module:

```
@NgModule({
  imports: [RouterModule.forRoot(...)]
})
export class AppModule { }
```

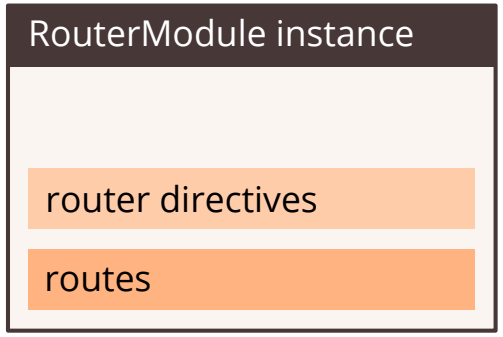
To use within non-root modules:

```
@NgModule({
  imports: [RouterModule.forChild(...)]
}
export class NonRootModule { }
```

RouterModule.forRoot(...)



RouterModule.forChild(...)



Configuring Routes

Routing maps from a URL `path` to a `component` to display:

JavaScript

```
const routes: Routes = [
  { path: '', redirectTo: 'home', pathMatch: 'full' },
  { path: 'find', redirectTo: 'search' },
  { path: 'home', component: HomeComponent },
  { path: 'search', component: SearchComponent },
  { path: '**', component: HomeComponent }
];
```

Default Route

To define a default route:

JavaScript

```
const routes: Routes = [
  { path: '', redirectTo: 'home', pathMatch: 'full' },
  ...
];
```

- A default route applies when the URL has no other path components
- `pathMatch` specifies how much of the path to match
  - `full` means "apply this rule when the rest of the URL matches `path`"
  - `prefix` means "apply this rule when the rest of the URL *starts with* `path`"
  - but with a `path` of `''`, *any* URL would match that so this rule would always apply!

Catch-All Route

To define a catch-all or wildcard route:

JavaScript

```
const routes: Routes = [
  ...
  { path: '**', component: HomeComponent }
];
```

- If the URL doesn't match any other path, this rule applies

Router Outlet

Defines the place where any components rendered via routes should go

HTML

```
<nav>
  ...
</nav>
<div>
  <router-outlet></router-outlet>
</div>
```

Router Links

To link to a component, use the `routerLink` directive:

HTML

```
<a routerLink="/contacts/ben">Link to user Ben</a>
```

Or if you generate the path segments dynamically:

HTML

```
<a [routerLink]="['/contacts', userName]" [queryParams]="{debug: true}" fragment="phone"
  Link to user whose name is in 'userName' value
</a>
```

- If `userName` is "ben", this generates a link to:
  - `/contacts/ben#phone?debug=true`
- The parameter to `routerLink` is just an array of path segments

Navigating Programmatically

You can navigate programmatically:

JavaScript

```
this.router.navigate(['/settings']);
```

- `navigate()` returns a promise
  - the promise resolves to true or false depending on whether the destination was navigated to successfully

A Simple Application using Routes -- 1

First, define the routes:

app.routes.ts

```
import { RouterModule } from '@angular/router';

import { ComponentOne } from './component-one.component';
import { ComponentTwo } from './component-two.component';

export const routes: Routes = [
  { path: '', redirectTo: 'component-one', pathMatch: 'full' },
  { path: 'component-one', component: ComponentOne },
  { path: 'component-two', component: ComponentTwo },
];
```

A Simple Application using Routes -- 2

Second, add `RouterModule` to the root module:

```
import { BrowserModule } from '@angular/platform-browser';
import { NgModule } from '@angular/core';
import { RouterModule } from '@angular/router';   // <-- import RouterModule

import { routes } from './app.routes';            // <-- import the routes

import { AppComponent } from './app.component';
import { ComponentOne } from './component-one.component';
import { ComponentTwo } from './component-two.component';

@NgModule({
  declarations: [ AppComponent, ComponentOne, ComponentTwo ],
  imports: [
    BrowserModule,
    RouterModule.forRoot(routes)                  // <-- add a RouterModule instance
  ],
  bootstrap: [AppComponent]
})
export class AppModule { }
```

A Simple Application using Routes -- 3

Add `routerLink`s and `<router-outlet>`s as appropriate:

app.component.ts

```
import { Component } from '@angular/cor

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
})
export class AppComponent {
```

app.component.html

HTML

```
<nav>
  <a [routerLink]="['/component-one']">
  <a [routerLink]='/component-two'>Comp
</nav>
<h2>Selected View</h2>
<div style="border: 2px solid blue; pad
  <router-outlet></router-outlet>
</div>
```

A Simple Application using Routes -- 4

The components themselves are straightforward:

component-one.component.ts

```
import {Component} from '@angular/core'

@Component({
  selector: 'component-one',
  template: '<p>Component One</p>'
})
export class ComponentOne { }
```

component-two.component.ts

```
import {Component} from '@angular/core'

@Component({
  selector: 'component-two',
  template: '<p>Component Two</p>'
})
export class ComponentTwo { }
```

Route Parameters

Path segments within a URL can be designated as parameters:

```
http://example.com/account/jjones/transaction/6314762
```

Route parameters enable us to pick these up and use them

```
export const routes: Routes = [
  ...
  { path: 'account/:accountId/transaction/:transactionId', component: TransactionViewCom
];
```

- These parameters are made available to the component navigated to

Accessing Route Parameters

To access the route parameters, subscribe to them:

```
import { Component } from '@angular/core';
import { ActivatedRoute } from '@angular/router';          // import ActivatedRoute

@Component({
  selector: 'transaction-view',
  template: '<p>Transaction: <b>ID: {{ id }}</b></p>'
})
export class TransactionViewComponent {
  private sub: any;
  private id;
  constructor(private route: ActivatedRoute) {}

  private ngOnInit() {
    this.sub = this.route.params.subscribe(params => {     // subscribe here
      this.id = +params['transactionId'];
    });
  }
}
```

Navigating with Route Parameters

To navigate specifying route parameters, specify them using the array syntax:

HTML

```
<a routerLink="['/account', 123]">View Account 123</a>
```

Or invoke programmatically:

JavaScript

```
onClick() {
  this.router.navigate( ['/account', 123] );
}
```

Query Parameters

A URL may have optional *query parameters*:

```
http://example.com/account/jjones/transactions?page=3
```

Routing Parameters vs Query Parameters

ROUTING PARAMETERS	QUERY PARAMETERS
Essential to determining route	Non-essential
Included in route definition	Not included

Accessing Query Parameters

To access the query parameters, subscribe to them similar to before:

```
@Component({
  selector: 'query-param-demo',
  template: `<div>Query parameter page #: {{page}}</div>`,
})
export class ComponentOne {
  private sub: any;
  private page:number;
  ...
  ngOnInit() {
    this.sub = this.route.queryParams.subscribe(params => {
      this.page = +params['page'] || 1;
    });
  }
}
```

Navigating with Query Parameters

To navigate using query parameters, specify them via an anchor's `queryParams` property:

HTML

```
<a [routerLink]="['accounts']" [queryParams]="{ page: 6 }">View Accounts page 6</a>
```

And of course you can navigate programmatically too:

```
this.router.navigate(['/products'], { queryParams: { page: pageNumber }});
```

Lazy Loading Modules

- By default all modules are loaded at start up time
    - known as eager loading
- Alternative is to load only when first used
    - known as lazy loading

Why Lazy Load?

Lazy loading offers:

- faster initial app startup
    - due to smaller payload

Setting up Lazy Loading

- Define via route declaration
  - use `loadChildren` and the module name rather than reference
  - not included in parent modules imports

```
import { NgModule } from '@angular/core';
import { CommonModule } from '@angular/common';

export const routes: Routes = [
    { path: '', redirectTo: 'info', pathMatch: 'full' },
    { path: 'settings', component: SettingsComponent },
    { path: 'admin',  loadChildren: './admin/admin.module#AdminModule' }
];
```

Modules

Angular Fundamentals

Why Modules?

Angular Modules help us to:

- organize our app
    - including components, directives and pipes
- extend it with third party libraries

Modules

Angular Modules provide:

- a way to organize an app's pieces
    - including components, directives and pipes
- dependency injection
    - so you can say for example "this module depends on that service instance"

Example Modules

Angular includes many modules 'out-of-the-box':

- `CommonModule` - a module that provides the directives `NgFor`, `NgIf`, `NgStyle` etc, plus common pipes
- `HttpModule` - a module that provides services to perform HTTP requests
- `FormsModule` - a module for data driven forms
- `ReactiveFormsModule` - a module for reactive forms
- `RouterModule` - a module that provides directives and providers for routing

Many third party modules are available too:

- ngx-bootstrap
- ngmodules.org
- Ionic

Use Cases of Modules

- Root Module
  - every app has one
- Feature Module
- Shared Component
- Component

Defining a Module

Modules

- defined as a class decorated with `@NgModule`

```
@NgModule({
  ...
})
export class UserProfileModule { }
```

Module Metadata

- `imports`
  - imports other modules for use within the module
  - `imports: [CommonModule],`
- `declarations`
  - declares components, directives and pipes
  - `declarations: [WidgetOne, WidgetTwo]`
- `exports`
  - exports some classes for use with other components
  - `exports: [WidgetOne, WidgetTwo]`
- `providers`

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

Root Module

- Every application has a root module
  - used to bootstrap the application Root Module
- Root module may be all that is required
  - importing other modules as required
  - implementing any components, pipes, directives and services that are required
- Larger applications will be comprised of several / many modules

Close