# Working with components and directives

Introduction to Angular 6

# Angular CLI

- Angular CLI is useful throughout the development process for a variety of tasks:
  - Generating new components, services, routes and more with code scaffolding
  - Providing a standardized project folder structure
  - Bootstrapping your application
  - Serving the application
  - Running the tests (both unit and end-to-end)
  - Creating a production grade build for distribution with environment specific configuration

# ng new

- ng new command takes a few arguments that allow you to customize the application generated to your preference.
  - Whether you want to use vanilla CSS or SCSS or any other CSS framework
  - Whether you want to generate a routing module
  - Whether you want inline styles/templates
  - Whether you want a common prefix to all components

# Running an app

- There are technically two ways to run an application
- Running it in development mode
  - where the Angular CLI compiles the changes as it happens and refreshes our UI
- Running it in production mode
  - with an optimal compiled build, served via static files

# Key files generated in project

- Key files that the Angular CLI generates
- main.js
  - the transpiled code that is specific to your application,
- vendor.js
  - includes all the third-party libraries and frameworks you depend on (including Angular).
- styles.js
  - is a compilation of all the CSS styles that are needed for your application
- polyfills.js
  - includes all the polyfills needed for supporting some capabilities in older browsers
- runtime.js
  - file with webpack utilities and loaders that is needed for bootstrapping the application

# Loading Angular

- Loading is triggered by a main request to the server.
  - When we open any URL in our browser, the first request is made to server running within ng serve
  - This initial request is satisfied by an HTML page (index.html)
  - Contains the marker element for the root component
- The part that loads the core Angular scripts and our application code is inserted dynamically at runtime by ng serve
  - This combines all the vendor libraries, application code, styles, and inline templates into individual bundles
  - injects them into index.html to be loaded as soon as the page renders in the browser.
- The web application runs as JavaScript
  - ng serve transpiles Angular Typescript into Javascript for the browser to load.

# main.ts

- identifies which Angular module is to be loaded when the application starts.
  - It can also change application-level configuration.
- Its main aim is to point the Angular framework at the core module of your application
- This triggers the rest of your application source code from that point

# app.module.ts

- This root module file provides the core configuration of your application
  - Loads all the relevant and necessary dependencies
- Declares which components will be used within your application
  - Marks the main entry point component of your application

# Module metadata

- An NgModule is defined by a class decorated with @NgModule().
  - The @NgModule() decorator is a function that takes a single metadata object, whose properties describe the module.
- declarations:
  - The components, directives, and pipes that belong to this NgModule.
  - For any component to be used within the context of a module, it has to be imported into your module file and declared here.
  - This ensures that the component is visible to all other components within the module.
  - The Angular CLI will automatically add your component or directive to the module when you create a component through it.

# Module metadata

- exports:
  - The subset of declarations that should be visible and usable in the component templates of other NgModules.
- imports:
  - Other modules whose exported classes and services are needed by component templates declared in this NgModule.
  - You must export a component to allow it to be accessed outside of the module where the component is declared
- providers:
  - Creators of services that this NgModule contributes to the global collection of services
- bootstrap:
  - The bootstrap array defines the root component that acts as the entry point to your application
  - Allows Angular to know what element to look for in index.html.

# Root component (AppComponent)

- The app-selector
  - is a CSS selector that identifies how Angular finds this particular component in any HTML page
  - Usually element selectors, but can also be any CSS selector, from a CSS class to an attribute as well
- Examples:
- selector: 'app-stock-item'
  - component being used as <app-stock-item></app-stock-item> in the HTML.
- selector: '.app-stock-item'
  - component being used as a CSS class like  in the HTML
- selector: '[app-stock-item]'
  - component being used as an attribute on an existing element like <div app-stock-item> in the HTML

# Root component (AppComponent)

- templateUrl
  - This is the path to the HTML used to render this component
  - The path passed pass to the templateUrl attribute is relative to the path of the component
  - can also use inline templates instead of specifying a templateUrl.
  - This allows the component to contain all the information instead of splitting it across HTML and TypeScript code.
  - Use multiline templates using the ` (backtick) symbol when defining inline templates.
- Angular builds does not load the template by URL at runtime.
  - Instead, it precompiles a build and ensures that the template is inlined as part of the build process.

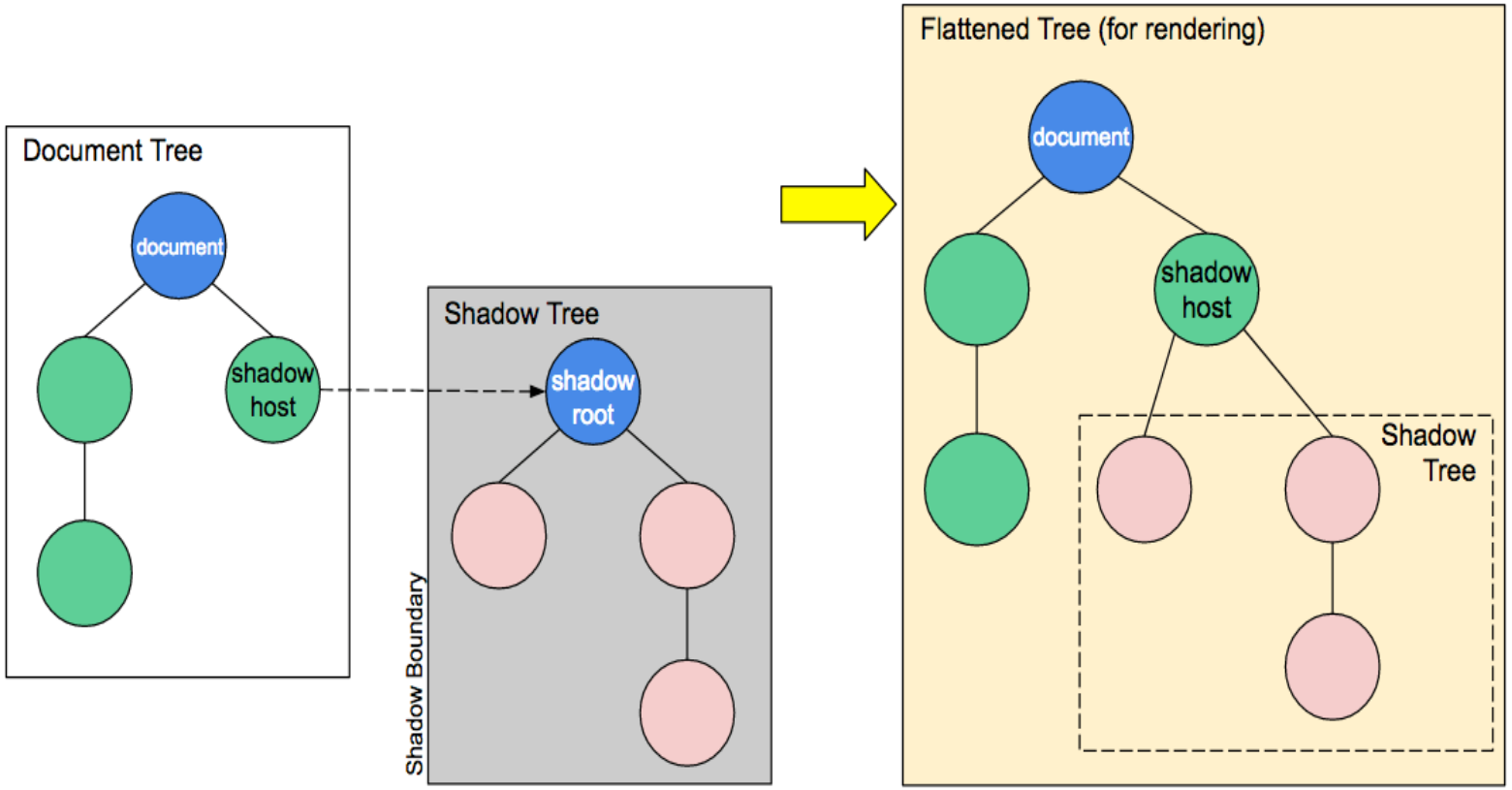
# Root component (AppComponent)

- styleURLs
  - A given component can have multiple styles attached to it, provided as an input array
  - This allows you to pull in component-specific CSS, as well as potentially any other common CSS that needs to be applied to it.
  - Similar to templates, you can either inline your CSS using the styles attribute

# Shadow DOM

- HTML, CSS, and JavaScript have a default tendency to be global in the context of the current page
  - ID given to an element can easily clash with another element somewhere else on the page (particularly with CSS)
- Shadow DOM fixes this by scoping HTML DOM and CSS
  - It provides the ability to have scoped styling to a component
  - thus preventing the styles from leaking out and affecting the rest of the application
  - also the ability to isolate and make the DOM encapsulated
  - is one of the four Web Component standards: HTML Templates, Shadow DOM, Custom elements and HTML Imports
- Whenever we create a component
  - Angular puts it's template into a shadowRoot, which is the Shadow DOM of that particular component

# Shadow DOM


Document Tree
document
shadow host
Shadow Tree
shadow root
Shadow Boundary
Flattened Tree (for rendering)
document
shadow host
Shadow Tree

# Style encapsulation

- Angular promotes encapsulation and isolation of styles.
- By default, the styles you define and use in one component will not affect/impact any other parent or child component.
- To change whether styles take effect locally or globally
  - use the encapsulation attribute on the Component decorator, which takes one of three values:

# Style encapsulation

- ViewEncapsulation.Emulated
  - This the default, where Angular creates shimmed CSS to emulate the behavior that shadow DOMs and shadow roots provide.
- ViewEncapsulation.Native
  - This is the ideal, where Angular will use shadow DOM and roots
  - This will only work on browsers and platforms that natively support it.
- ViewEncapsulation.None
  - Uses global CSS, without any encapsulation
  - Is a good way of applying common styles to all child components
  - However adds the risk of polluting the global CSS namespace and having unintentional effects.

# Generating components

- The Angular CLI generate command generates relevant files for a new component:
  - The component definition
  - The corresponding template definition
  - The styles for the component
  - The skeleton unit tests for the component
- Also updates the original app module so that Angular application recognizes the new module.

# Working with components

- Recommended convention to follow whenever you are working with components
  - The filename starts with the name of the item you are creating
  - This is followed by the type of element it is (in this case, a component)
  - Finally, the relevant extension
- Responsibilities of the component class:
  - Load and hold all the data necessary for rendering the component
  - Handle and process any events that may arise from any element in the component
- To use this component in our application
  - create an element that matches the selector defined anywhere inside our main app component
  - If we have a hierarchy of components we can choose to use it in any of their templates as well

# OnInit

- Angular gives us hooks into the lifecycle of a component
- OnInit
  - Angular's OnInit hook is executed after the component is created by the Angular framework
  - generally recommended to do any initialization work of a component in the OnInit hook
  - makes it easier to test the functionality of the rest of the component without necessarily triggering the initialization flow every time
- ngOnInit
  - When you want to hook on the initialization phase of a component
  - implement the OnInit interface and then implement the ngOnInit function in the component

# Interpolation for HTML

- Uses double-curly notation ({{ }})
  - evaluates the expression between the curly braces as per the component backing it
  - and then renders the result as a string in place in the HTML.
  - The expressions can be slightly more complex.
- One-way data binding automatically updates the UI based on values in the component
  - and then keeps them updated  as the value changes in the component

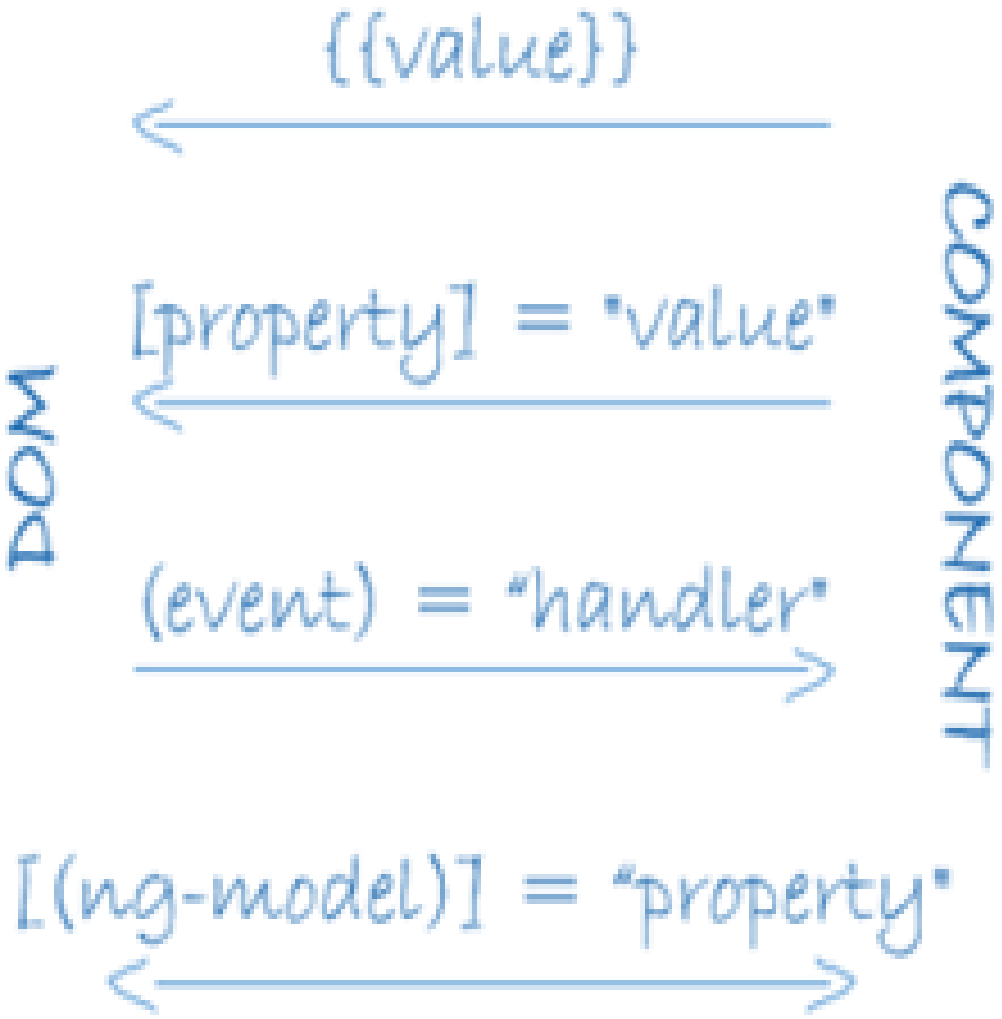
# Binding to DOM properties

- [] syntax is used with any DOM property on an element
  - to bind one way from the component to the UI
  - allows us to modify the content and the behavior of the HTML that is rendered in the browser
- When you bind to the class property
  - it overrides the existing value of the property.
- Can also bind to other HTML properties
  - E.g. src property of an img tag, or disabled property of input and button

# Event binding

- the left part of the equals symbol refers to the event we are binding to
- the right part of the equals symbol refers to the template statement that Angular should execute whenever the event is triggered
- Angular also gives access to the underlying DOM event
  - by giving access to a special variable $event.
  - Can use this to hook onto other standard DOM events that are triggered, like focus, blur, submit, etc

# Property and event binding



# Developing a data model

- Use encapsulation to ensure that our components don't work with lower-level abstractions and properties
  - Model data items as a type in TypeScript
- define an interface or a class with the definition for the particular data model
  - use that consistently throughout the application

# Falsy and truthy values

- In JavaScript, the following values are treated as falsy:
  - undefined
  - null
  - NaN
  - 0
  - "" (any empty string)
  - false (the boolean value)
- Any other value is treated as truthy, including, but not limited to:
  - Any nonzero number
  - Any nonempty string
  - Any nonnull object or array
  - true (the boolean value)

# Attribute Directive: NgClass

- allows us to apply or remove multiple CSS classes simultaneously from an element in HTML
  - For each key in the object that has a truthy value, Angular will add that key as a class to the element
  - Each key in the object that has a falsy value will be removed as a class from that element
- The NgClass directive retains the classes on the element.
- Consider using the NgClass directive when having to apply various different CSS classes on an element conditionally
  - Makes it easy to  reason and understand how and what classes are applied
  - unit test the logic of selecting classes separate from the logic of applying classes to the elements

# Attribute Directive: NgStyle

- works at a CSS style/properties level
  - takes a JSON object and applies it based on the values of the keys.
  - The keys and values it expects are CSS properties and attributes
- It is generally preferable to use the class or NgClass bindings to change the look and feel of your application,

# Alternative to NgClass and NgStyle

- Use a singular version of the class and style binding
  - that adds and removes one particular class/style
- add or remove individual classes based on evaluating a truthy expression in Angular
- However preferable to use the NgClass directive any time you have to deal with more than one or two classes

# Structural directives

- Structural directives use microsyntax
  - a mini-language of sorts that Angular uses to accomplish certain things
- All structural directives in Angular start with an asterisk (*)
- Form of syntactic sugar that is replaced underneath by Angular to a series of steps, resulting in the final view

# Structural Directive: NgIf

- allows you to conditionally hide or show elements in your UI
  - based on whether the expression provided to it to evaluates to a truthy or falsy value
- Removes the element, instead of hiding it via CSS due to performance issues
  - Angular continues to listen and watch for events on all elements in the DOM
  - Removing the element from the DOM reduces any impact on performance
  - especially when the element being removed is a resource-intensive component

# Structural Directive: NgFor

- used for creating multiple elements
  - usually one for each instance of some object in an array.
- The first part of the microsyntax is the for loop
  - create a template instance variable which will be available within the scope of the element created
- The second part creates another template instance variable
  - Used to hold the current index value.

# Working with trackBy

- By default, Angular uses object identity to track additions, removals, and modifications to an array.
- As long as the object reference does not change, Angular will not create new elements for it, and will reuse the old element reference
  - Mainly due to performance reasons, element creation/removal are two of the more costly operations in the browser.
- There are cases when the element reference might change, but you still want to continue using the same element.
  - E.g. when fetching data from server, you don't want to recreate a list unless the items have actually changed

# Working with trackBy

- The trackBy option takes a function that has two arguments: an index and the item.
  - trackBy function is provided to the NgFor directive, it will use the return value of the function to decide how to identify individual elements.
  - Modify *ngFor to pass in an additional attribute in the microsyntax
- This will ensure that Angular calls this function to figure out how to identify individual items, instead of using the object reference.
  - Even if we reload all the list items from the server (thus changing all the object references), Angular will still use the function to decide whether or not to reuse the elements present in the DOM

# Structural Directive: NgSwitch

- Used when you have multiple elements/templates of which one has to be rendered based on conditions
- Used with normal data-binding syntax with the square-bracket notation
- Used along with NgSwitchCase and the NgSwitchDefault directives
  - which would add or remove elements depending on the case.

# Multiple directives

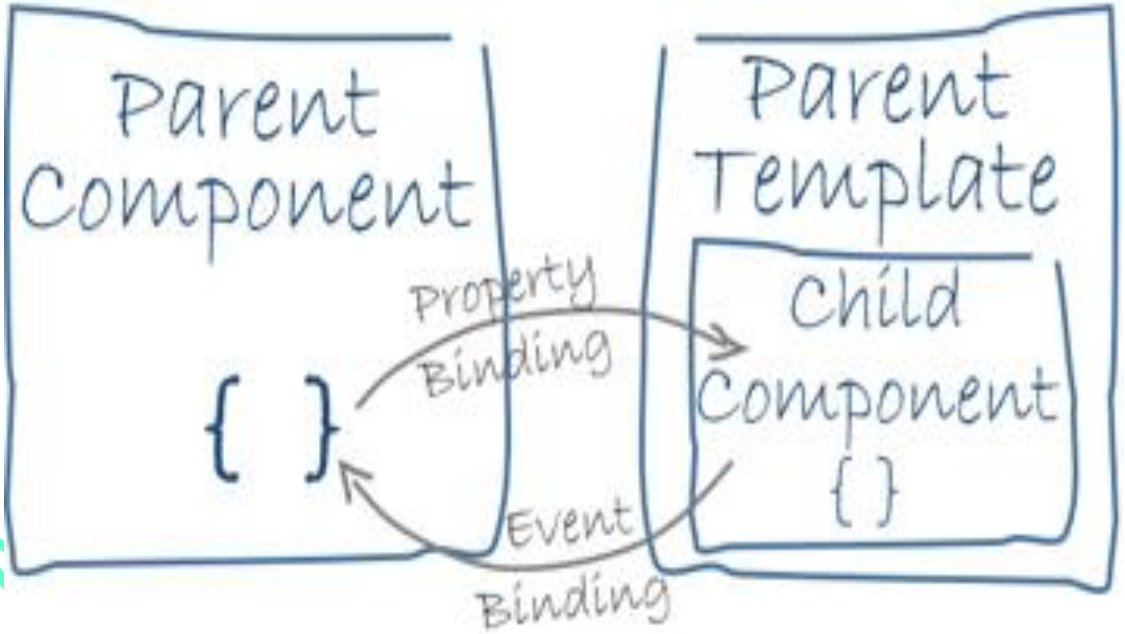
- Angular does not allow multiple directives on the same element
  - Due to ambiguity with regards to the order of execution
- It is recommended to just use wrapping elements (e.g. div)
- This allows you to be explicit about the order in which these structural directives are executed

# Input and output decorators

- Useful to separate the content that a component uses from the component itself
  - This increases its usability
- One of the ways we can make a component reusable is by passing in different inputs to a child component from its parent depending on the use case
- Angular provides hooks to specify each of these through decorators
  - Input and Output
  - They apply at a class member variable level

# Input decorator

- When we add an Input decorator on a member variable
  - it automatically allows you to pass in values to the component for that particular input via Angular's data binding syntax
- The name of the attribute has to match the name of the variable in the component that has been marked as input



# Output decorator

- We can also register and listen for events from a child component
  - Use event binding syntax to register for events
  - Use the Output decorator to accomplish this
- We register an EventEmitter as an output from any component
  - trigger the event using the EventEmitter object
  - any component bound to the event gets a notification and acts accordingly
- need to ensure that the EventEmitter instance is initialized correctly

# Change detection

- By default, Angular applies the ChangeDetectionStrategy.Default mechanism
  - every time Angular notices an event, it will go through each component in the component tree
  - check each of the bindings individually to see if any of the values have changed and need to be updated in the view
- For a very large application, you will have lots of bindings on a given page
  - When a user triggers an event, developer may know for sure that most of the page will not change
  - It possible to give a hint to the Angular change detector to check or not check certain components as you see fit.

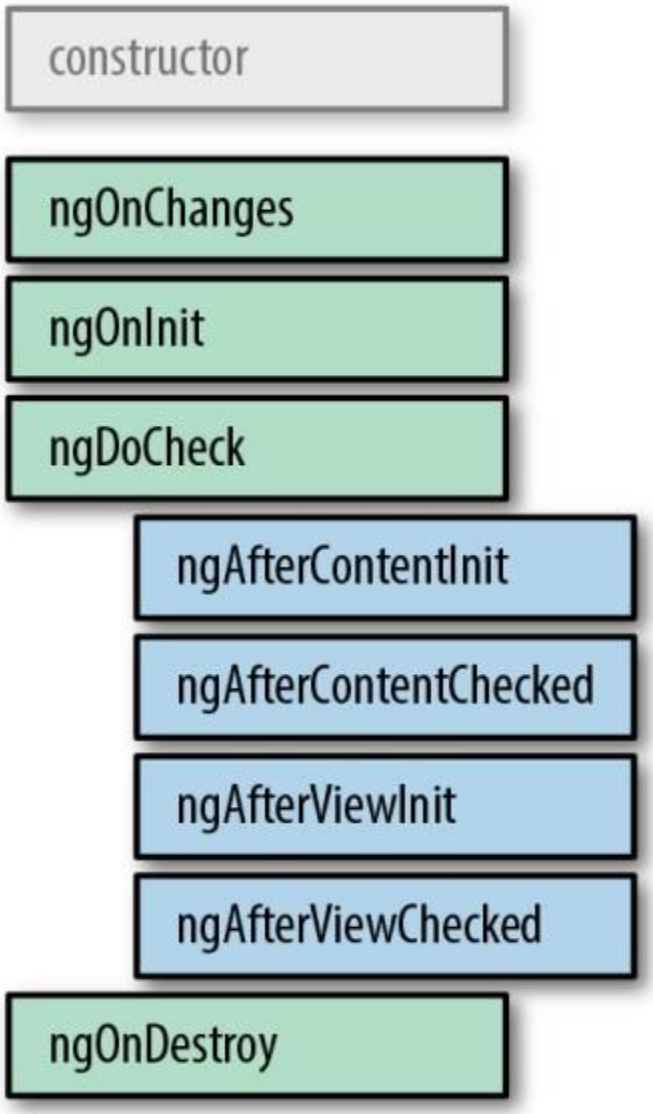
# Change detection

- accomplish this by changing the ChangeDetectionStrategy from the default to ChangeDetectionStrategy.OnPush.
- tells Angular that the bindings for this particular component will need to be checked only based on the Input to this component

# Component life cycle

- Components in Angular have their own life cycle
  - from creation, rendering, changing, to destruction.
- This life cycle executes in preorder tree traversal order, from top to bottom.
- After Angular renders a component, it starts the life cycle for each of its children, and so on until the entire application is rendered.
- There are times when these life cycle events are useful to us in developing our application
  - Angular provides hooks into this life cycle so that we can observe and react as necessary

# Component life cycle



# Component life cycle

- Angular will first call the constructor for any component.
- Any lifecycle hook ending with Init is called only once, when a component is initialized
- Others are called whenever any content changes
- OnDestroy hook is also called only once for a component
- Each of these lifecycle steps comes with an interface that should be implemented when a component need to hook into that step
- Each interface provides a function starting with ng that needs to be implemented
- https://angular.io/guide/lifecycle-hooks

# View projection

- Useful when we want to build components but keep some parts of the UI of the component to be separate
  - The view would be controlled by the user of the component (parent component), while the functionality would be provided by the component itself (child component)
  - It is also possible to project multiple sections and content into child component.
- ViewChildren
  - any child component whose tags/selectors (mostly elements) appear within the template of the component
- ContentChildren
  - any child component that gets projected into the view of the component, but is not directly included in the template within the component.