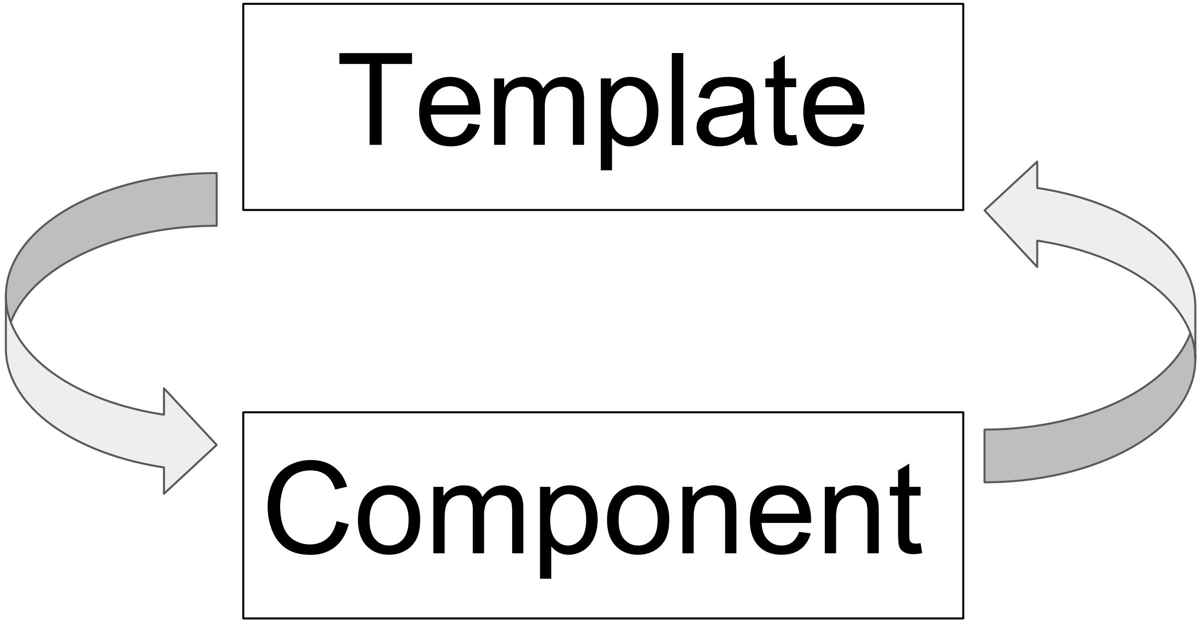
# Angular Templates

angular templating language

## Angular Components



## Angular Components

- Components in angular represent a block of ui
- components are made from class and template
- the class contains the behaviour of the component
- the template contains the UI of the component
- DI is creating class instance and for every class instance a template controlled by that instance
- the template can be defined in the typescript file or in an external file
- if on external file it’s common that the template will have an **html** suffix
- we define a selector for the component
- When DI sees the selector it creates a new instance of the class with a template

## HTML in templates

- Almost all HTML tags are valid template syntax
- Except **<script>, <html>, <body>, <base>**
- You can extend the template language by adding new tags/class/attributes that will map to components and directives
- Let’s start to practice angular templates by creating a new angular project with angular cli

## Interpolation

- syntax: **{{}}**
- Can be placed in:
  - between tags
  - as value to attribute
  - as input to component/directive
- Can contain
  - JS calculation (not using global libraries)
  - public properties from class
  - public methods from class
- Angular will calculate the expression and convert it to string and display it
- the expression should not have side effects
- the expression should be the same if running with same params

## Binding component and template

- the context of a template is the component instance
- the template is binded to the component instance
- 3 types of communication:
  - one way from component to template
  - one way from template to component
  - two way binding

## one way component to template

- component can pass properties and method to the template
- methods should be simple and not too complicated
- one way we saw earlier is with interpolation **{{}}**
- another way to pass data is by binding to DOM properties with the following syntax
    - **<some-tag [domproperty]=”expression”>**
- **some-tag** can also be an angular directive or component we will see example further on
- if you drop the **[]** angular will look at the expression as string and not calculate the expression
- Let’s try and bind to dom property

## Playing with binding properties

- bind to class
- bind to style

## One way from template to component

- templates pass data through events
- you can connect to a dom element event
- you can connect to an angular component or directive event (will demonstrate later)
- the syntax is:
  - **<some-tag (dom-event)=”expression”>**
- expression can call a function on the component
- expression can make assignments
- expression can have side effects
- **$event** will contain the original dom event which you can pass on
- Let’s try and bind dom events

# 2 way binding

- in some cases we want to keep the template in sync with a property in the class and also keep that property in sync with an event from the template
- common use case in forms for example input field we want to keep the value of the input sync with a property in the component
- We can use an angular directive called NgModel form 2 way binding
- that directive belong to the package: **@angular/forms** you need to use **npm** to install that package
- using directives from other modules require us to include the module in our module **imports** array
- The syntax for using **NgModel** directive is:
  - **<input [(ngModel)]="<property>" />**
- We will talk further on ngModel in upcoming classes
- let's try and use 2 way binding

## Can u think of another way?

- is there another way to to achieve the same thing?

## Communication parent child component

- now that our app is made from ui components, we will often have cases where a parent component is hosting a child component
- Let’s examine how communication is made  with child parent components
- For examining this we will create a child component which displays a simple hello from child message

## One way parent to child

- we want to transfer data from the parent to child
- the syntax is:
  - **<my-element [someInput]=”expression”>**
- expression should be without side effects
- without square brackets it will be considered as constant string
- the child component needs to implement **@input** for what he expects to get
- to demonstrate let’s make our child component receive the message he should print from the parent

## @Input

- created a property which can receive value from parent
- decorator factory for a class property
- imported from @angular/core
- Can accept optional string which can change the name of the attribute that is connected to the property
- you can define inputs also in the @Component metadata
  - **inputs: [‘message’]**
  - **inputs: [‘message:message-to-child’]**
- you can intercept changes in two ways:
  - define the property with setter and getter
  - implement the OnChanges lifecycle event (more on the component lifecycle lesson)
- When the property value change angular will re render the component

# From child to parent - events

- common way for child to transfer data to parent is via events
- we can define an event on the child and the parent can subscribe to that event
- the syntax is:
    - **<my-component (some-event)=”expression” >**
- expression can have side effect
- component has to expose a property of type EventEmitter
- The property exposed should have an **@Output** decorator
- component can send data in the event via $event
- to demonstrate we will create a button in the child component
- the button will control the visibility of the message
- the button will also emit an event and will send the **isVisible** property to whomever is subscribing
- the parent will subscribe to that event

## Template reference variable

- With template reference variable you can get dom element or instance of a component or directive
- we can use it to get the instance of the child component
- the variable scope is the template
- the syntax is:
  - **<jb-child #someNameThatWillGetTheInstance …>**
- you can transfer the variable to functions
- you can add a name for the variable using the **exportAs** metadata then the syntax will be:
  - **<jb-child #someNameThatWillGetTheInstance=”nameSetInExportAs” …>**
  - this is useful when there is a component and directive on the same element and you want to select a specific one

## @ViewChild

- property decorator
- takes a class or a string as selectors
- try to match the first element in the view that match the selector and populate the property with that value
- can be used to grab instance of the child component in the parent
- if you have more than one component instance you can place the template reference variable as the string of the decorator and it will find the specific component
- Let’s try and grab the child component instance using the @ViewChild

## 2 way binding child to parent

- Let’s create a text input in the child component
- We will create 2 way binding to that text input
- the parent component can pass value to the child input
- on input change the parent component property will be updated
- to implement two way binding we need to decide of **propertyName**
- attach **@Input() propertyName**
- attach **@Output() propertyNameChange**
- we use the **[(propertyName)]** to attach to it

## Built-in Directives

- angular has some built in directives which you can use
- the directives are defined in the **BrowserModule**
- if you are creating a module which is not root, you will have to add the **CommonModule** in the imports to use those directives
- We will now go over the common built in directives

## ngIf

- Syntax is:
  - **<some-element *ngIf=”expression” >**
- if expression evaluates to true then the component will be placed in the dom
- if expression is false than angular will destroy the element and any child elements it contains
- Directives that alter the dom are prefixed with *
- you can specify a then block with **ng-template**
- you can specify an else block with **ng-template**

# ngFor

- *ngFor is used to iterate on an iterable and display dom element on every iteration
- the syntax is:
  - **<li *ngFor=”let item of iterable”></li>**
  - this will go over the **iterable** property and print **li** on every iteration
- there are some added variable you can get like
  - index, first, last, even, odd
- let’s define an array of strings in the parent component and print that array in the template

## ngClass

- we already learned a syntax with property binding to add a class
- if you want to add a list of classes this way can be uncomfortable
- using the **ngClass** directive you can give the directive a list of classes in an object
- angular will evaluate the object and place the key class if the value is true

## ngStyle

- we already learned a way to add styling to element with property binding
- if the list of styles is too long that syntax can be cumbersome
- using **ngStyle** you can place an object where the keys are the styles and the values are the styles values

## Pipes

- pipes job is to run a transformation function on an input
- pipe syntax is:
  - **{{ value | pipe:param1:param2 }}**
- pipe will recalculate only when the value or the params are changing
- angular common pipes:
  - uppercase, lowercase, json
- you can also create your own pipe by implementing **PipeTransform**
- you have to put the pipes you create in a module

## Safe Navigation Operator (?.)

- Let’s define a property in our parent component
- that property initial value is null
- after a few seconds the property gets an object with title key
- we would like to print that key in the template
- angular will display an error that the object is null and don’t have a title key
- we can place a question mark before the dot and this tells angular to check that property only if the property is not null

## Summary

- Angular templates are based on HTML
- we can expend that HTML with our components directives and pipe
- angular2 replaced the many directives in angular1 with a general syntax to bind event, bind properties, and two way binding.

## EX templates

- Create a component to display a form for creating a new Task
- Create a component that will display the list of tasks
- Create a service that will hold array of tasks
- the form component will add new task to the array
- the list component will display the list of tasks from the array