Change Detection means updating the DOM whenever data is changed. Angular provides two strategies for Change Detection.

In its default strategy, whenever any data is mutated or changed, Angular will run the change detector to update the DOM. In the onPush strategy, Angular will only run the change detector when a new reference is passed to `@Input()` data.

To update the DOM with updated data, Angular provides its own change detector to each component, which is responsible for detecting change and updating the DOM.

Let's say we have a MessageComponent, as listed below:

```
import { Component, Input } from '@angular/core';
@Component({
  selector: 'app-message',
  template: `
    <h2>
    Hey {{person.firstname}} {{person.lastname}} !
    </h2>
    `
})
export class MessageComponent {
  @Input() person;
}
```

In addition, we are using MessageComponent inside AppComponent as shown below:

```
import { Component, OnInit } from '@angular/core';
@Component({
    selector: 'app-root',
    template: `
  <app-message [person]='p'></app-message>
  <button (click)='changeName()'>Change Name</button>
  `
})
export class AppComponent implements OnInit {
    p: any;
    ngOnInit(): void {
        this.p = {
            firstname: 'Brad',
            lastname: 'Cooper'
        };
    }
}
```

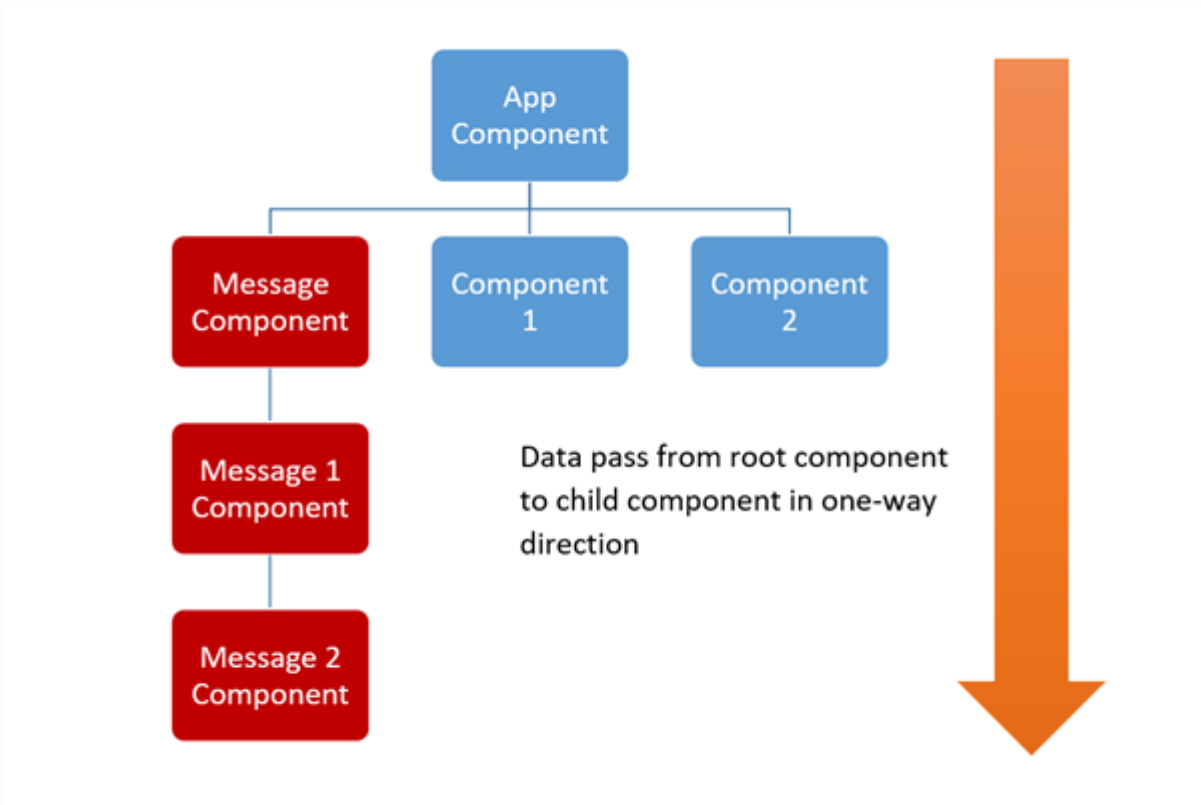
Let us talk through the code: all we are doing is using `MessageComponent` as a child inside `AppComponent` and setting the value of `person` using the property binding. At this point in running the application, you will get the name printed as output.

Next, let's go ahead and update the `firstname` property on the button click in the `AppComponent` class below:

```
changeName() {
    this.p.firstname = 'Foo';
}
```

As soon as we changed the property of mutable object P, Angular fires the change detector to make sure that the DOM (or view) is in sync with the model (in this case, object p). For each property changes, Angular change detector will traverse the component tree and update the DOM.

Let's start with understanding the component tree. An Angular application can be seen as a component tree. It starts with a root component and then goes through to the child components. In Angular, data flows from top to bottom in the component tree.



Whenever the `@Input` type property will be changed, the Angular change detector will start form the root component and traverse all child components to update the DOM. Any changes in the primitive type's property will cause Angular change detection to detect the change and update the DOM.

In the above code snippet, you will find that on click of the button, the first name in the model will be changed. Then, change detection will be fired to traverse from root to bottom to update the view in MessageComponent.

There could be various reasons for Angular change detector to come into action and start traversing the component tree. They are:

1. Events fired such as button click, etc.
2. AJAX call or XHR requests.
3. Use of JavaScript timer functions such as setTimeOut , SetInterval.

Now, as you see, a single property change can cause change detector to traverse through the whole component tree. Traversing and change detection is a heavy process, which may cause performance degradation of application. Imagine that there are thousands of components in the tree and mutation of any data property can cause change detector to traverse all thousand components to update the DOM. To avoid this, there could be a scenario when you may want to instruct Angular that when change detector should run for a component and its subtree, you can instruct a component's change detector to run only when object references changes instead of mutation of any property by choosing the **onPushChangeDetection** strategy.

You may wish to instruct Angular to run change detection on components and their sub-tree only when new references are passed to them versus when data is simply mutated by setting change detection strategy to onPush.

Let us go back to our example where we are passing an object to `MessageComponent`. In the last example, we just changed the `firstname` property and that causes change detector to run and to update the view of `MessageComponent`. However, now we want change detector to only run when the reference of the passed object is changed instead of just a property value. To do that, let us modify `MessageComponent` to use the OnPush ChangeDetection strategy. To do this set the `changeDetection` property of the `@Component` decorator to `ChangeDetectionStrategy.OnPush` as shown in listing below:

```
import { Component, Input, ChangeDetectionStrategy } from '@angular/core';
@Component({
    selector: 'app-message',
    changeDetection: ChangeDetectionStrategy.OnPush,
    template: `
    <h2>
    Hey {{person.firstname}} {{person.lastname}} !
    </h2>
    `
})
export class MessageComponent {
    @Input() person;
}
```

At this point when you run the application, on the click event of the button in the `AppComponent` change detector will not run for `MessageComponent`, as only a property is being changed and reference is not changing. Since the change detection strategy is set to onPush, now the change detector will only run when the reference of the `@Input` property is changed.

```
changeName() {
    this.p = {
        firstname: 'Foo',
        lastname: 'Kooper'
    };
}
```

In the above code snippet, we are changing the reference of the object instead of just mutating just one property. Now when you run the application, you will find on the click of the button that the DOM is being updated with the new value.

By using **onPush Change Detection**, Angular will only check the tree if the reference passed to the component is changed instead of some property changed in the object. We can summarize that, and use an Immutable Object with **onPush Change Detection** to improve performance and run the change detector for the component tree when the object reference is changed.

We can further improve performance by using RxJS Observables because they emit new values without changing the reference of the object. We can subscribe to the observable for new value and then manually run ChangeDetector.

Let us modify `AppComponent` to pass an observable to `MessageComponent`.

```
import { Component, OnInit } from '@angular/core';
import { BehaviorSubject } from 'rxjs/BehaviorSubject';
@Component({
    selector: 'app-root',
    template: `
  <app-message [person]='data'></app-message>
  <button (click)='changeName()'>Change Name</button>
  `
})
export class AppComponent implements OnInit {
    p: any;
    data: any;
```

```
    ngOnInit(): void {
        this.p = {
            firstname: 'Brad',
            lastname: 'Cooper'
        };
        this.data = new BehaviorSubject(this.p);
    }
    changeName() {
        this.p = {
            firstname: 'Foo',
            lastname: 'Kooper'
        };
        this.data.next(this.p);
    }
}
```

In the code, we are using `BehaviorSubject` to emit the next value as an observable to the `MessageComponent`. We have imported `BehaviorSubject` from RxJS and wrapped object p inside it to create an observable. On the click event of the button, it's fetching the next value in the observable stream and passing to `MessageComponent`.

In the `MessageComponent`, we have to subscribe to the person to read the data.

```
@Input() person: Observable<any>;
_data;
ngOnInit(){
    this.person.subscribe(data => {
        this._data = data;
    });
}
```

Now, on the click of the button, a new value is being created, however, a new reference is not being created as the object is an observable object. Since a new reference is not created, due to onPush ChangeStrategy, Angular is not doing change detection. In this scenario, to update the DOM with the new value of the observable, we have to manually call the change detector as shown below:

```
export class MessageComponent implements OnInit {
    @Input() person: Observable<any>;
    _data;
    constructor(private cd: ChangeDetectorRef) { }
    ngOnInit() {
        this.person.subscribe(data => {
            this._data = data;
            this.cd.markForCheck();
        });
    }
}
```

We have imported the `ChangeDetectorRef` service and injected it, and then called `markForCheck()` manually to cause change detector to run each time observable emits a new value. Now when you run application, and click on the button, the observable will emit a new value and the change detector will update the DOM also, even though a new reference is not getting created.

To summarize:

1. If Angular ChangeDetector is set to default then for any change in any model property, Angular will run change detection traversing the component tree to update the DOM.
2. If Angular ChangeDetetor is set to onPush then Angular will run change detector only when new reference is being passed to the component.
3. If observable is passed to the onPush change detector strategy enabled component then Angular ChangeDetctor has to be called manually to update the DOM.