|  |  |
| --- | --- |
|  | Francotyp-Postalia |

Windows IoT – Development Environment Setup

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| # | Version | Date | Authors | Comments/Modifications |
|  | 1.0 | 26.09.2016 | Hendryk Irmischer | initial revision |
|  |  |  |  |  |
|  |  |  |  |  |

Basic starter guide for application development with Angular2.

Table of Contents

[1. Links 2](#_Toc462657980)

[2. Angular2 Best Practices 2](#_Toc462657981)

[3. Creating a sample component 2](#_Toc462657982)

[3.1. Component Parts 2](#_Toc462657983)

[3.2. Passing data to sub components 3](#_Toc462657984)

[3.3. One way bindings 3](#_Toc462657985)

[3.4. Two way bindings 3](#_Toc462657986)

[3.5. Retrieving data from sub components 3](#_Toc462657987)

# Links

* [Angular2 - QuickStart and API Reference](https://angular.io/docs/ts/latest/quickstart.html)
* [Angular 2 Basics](https://www.sitepoint.com/getting-past-hello-world-angular-2/)
* [Angular 2 Components: Inputs and Outputs](https://www.sitepoint.com/angular-2-components-inputs-outputs/)
* [Passing data to and from a nested component in Angular 2](https://www.themarketingtechnologist.co/building-nested-components-in-angular-2/)

# Angular2 Best Practices

* Use dependency injection whenever possible
* Encapsulate as much service functionality as possible into separate service classes
* Think in components: Always split up new features into parts of functionality

# Creating a sample component

## Component Parts

1. View – Declarative Description of our View. The view is typically described in HTML and some sort of template language which Angular2 can interpret.

<settings-component>

<h3>Settings</h3>

<sub-component-1></sub-component-1>

<sub-component-2></sub-component-2>

</settings-component>

1. Controller – Sometimes also referred to as ViewModel it acts as the data layer for the view. Text fields in the view are bound to properties of the controller. Changes to those properties are then automatically propagated to the view by Angular 2. To associate a view with its controller one only needs to decorate the controller class with the ‘@Component’ attribute. For the moment, only the most important component properties are being described:

* *selector: identifier/html tag in the view*
* *templateUrl: url to the template file*
* *providers: Services, that need to be provided to the component via dependency injection*

@Component({

selector: 'settings-component',

templateUrl: 'templates/settings.component.html'

providers: [

SettingsService

]

})

export default class SettingsComponent {

constructor(settingsService: SettingsService) {}

}

1. Services – Although services are not a direct part of components, every reusable piece of functionality should be encapsulated into separate services which are then provided as services to components.
2. Components itself can contain other components

## Passing data to sub components

Let’s say we retrieve all relevant view data in the parent component via some service. We now need to propagate these data to the sub components. We do this, by using Angulars property notation:

<settings-component>

<h3>Settings</h3>

<sub-component-1 [propertyToBeSet]=”settings.value”>

</sub-component-1>

<sub-component-2></sub-component-2>

</settings-component>

We simply tell Angular2 here, that we want to pass the value of the settings object inside our parent component to ‘propertyToBeSet’ inside the sub component. Inside the sub component we then need a property with that exact name and a specific decorator:

@Component({

selector: ‘sub-component-1’,

templateUrl: 'templates/settings/settings.sub1.html'

})

export default class SubComponent1 {

// Provided by parent component

@Input() propertyToBeSet: string;

}

Fields marked with this ‘@Input()’ decorator can be bound to and set from outside the component.

## One way bindings

<input type="text" value=”{{propertyToBeSet}} />

One way bindings or one time bindings are bindings, where values are being set once. There is no additional change tracking.

## Two way bindings

<input type="text" [(ngModel)]=”{{propertyToBeSet}} />

To add change tracking to a binding, we typically use the ‘ngModel’ attribute. Changes in the controller are automatically propagated to the view and vice versa.

## Retrieving data from sub components

To propagate changes back to the parent component we will need another decorator inside our sub component:

@Component({

selector: ‘sub-component-1’,

templateUrl: 'templates/settings/settings.sub1.html'

})

export default class SubComponent1 {

@Input() propertyToBeSet: string;

// Propagates changes to parent component

**@Output()** **handler**: EventEmitter<string> = new EventEmitter<string>();

private button\_onClick() {

this.propertyToBeSet = “newValue”;

this.handler.emit(this.propertyToBeSet);

}

}

The ‘@Output’ decorator ensures, that the parent component is being notified, if a value inside one of the sub components has changed. We just need to ensure, that the corresponding event is being emitted, if a value has been changed. As a little helper, we can make use of setters:

export default class SubComponent1 {

@Input() propertyToBeSet: string;

**public set PropertyToBeSet(val: string) {**

this.propertyToBeSet = val;

// automatically emit changed event

this.handler.emit(this.propertyToBeSet);

**}**

// Propagates changes to parent component

**@Output()** **handler**: EventEmitter<string> = new EventEmitter<string>();

private button\_onClick() {

this.PropertyToBeSet = “newValue”;

}

}

Now, to get a notification from sub components inside the parent, we just need to subscribe to the corresponding changed events of the sub components:

<settings-component>

<h3>Settings</h3>

<sub-component-1 [propertyToBeSet]=”settings.value”

**(handler)=”subComponent1\_OnChanged($event);”**>

</sub-component-1>

<sub-component-2></sub-component-2>

</settings-component>

Now we can handle every change from a subcomponent inside the parent component:

@Component({

selector: 'settings-component',

templateUrl: 'templates/settings.component.html'

providers: [

SettingsService

]

})

export default class SettingsComponent {

constructor(settingsService: SettingsService) {}

**private subComponent1\_OnChanged(newString: string) {**

**this.settings.value = newString;**

**}**

}