Frontend-Development with Angular

Mail: jonas.bandi@ivorycode.com

Twitter: @jbandi

Templates

Template Reference Variables

Template reference variables can be used inside a template. They reference a DOM element or an Angular component/directive.

references the DOM element

```
<input #nameInput>
<button (click)="updateName(nameInput.value)">Update!</button>
```

accessing the DOM element property

references the component instance

```
<aw-greeter #greeterComponent></aw-greeter>
<button (click)="greeterComponent.reset()">Reset!</button>
```

calling component method

directive

references the directive

```
<input [(ngModel)]="name" #nameControl="ngModel" required>
Debug: {{nameControl.valid)}}
```

acessing the property of the directive

https://angular.io/guide/user-input
https://angular.io/guide/template-syntax#template-reference-variables--var-

Structural Directives

Structural directives shape or reshape the DOM's structure.

The host element of the directive is used as a template that is instantiated by Angular.

The three common, built-in structural directives:
`*ngFor, *ngIf, *ngSwitchCase`

```
<ul>
  <li *ngFor="let character of characters">
    <span>{{character.firstName}}</span>
    <span>{{character.lastName}}</span>
    <span *ngIf="showDistrict">{{character.district}}</span>
    <span [ngSwitch]="character.emotion">
      <span *ngSwitchCase="Emotion.InLove">❤️</span>
      <span *ngSwitchCase="Emotion.Angry">🔪</span>
      <span *ngSwitchCase="Emotion.Sad">💧</span>
      <span *ngSwitchDefault>⁉️</span>
    </span>
  </li>
</ul>
```

https://angular.io/guide/structural-directives

Structural Directives

The `*` is part of the name of the structural directive.

Structural directives are a *microsyntax* that expands into **`<ng-template>`** elements.

```
<ul>
    <ng-template ngFor let-character [ngForOf]="characters">
        <li>
            <span>{{character.firstName}}</span>
            <span>{{character.lastName}}</span>
            <ng-template [ngIf]="showDistrict">
                <span>{{character.district}}</span>
            </ng-template>
        </li>
    </ng-template>
</ul>
```

`<ng-template>` and **`<ng-container>`** are Angular elements used to render html. They can also be used directly.

https://angular.io/guide/structural-directives

*ngIf & else

```
<span *ngIf="character.isInLove; else elseBlock">❤️</span>
<ng-template #elseBlock>⁉️</ng-template>
```

```
<span *ngIf="character.isInLove; then thenBlock; else elseBlock"></span>
<ng-template #thenBlock>❤️</ng-template>
<ng-template #elseBlock>⁉️</ng-template>
```

https://angular.io/api/common/NgIf

*ngFor & trackBy

***ngFor** re-creates the DOM elements when the bound objects change.

With a **trackBy** function Angular can reuse DOM elements.

```
<ul>
  <li *ngFor="let character of characters; trackBy:trackByFirstName">
    <span>{{character.firstName}}</span>
    <span>{{character.lastName}}</span>
    <span *ngIf="showDistrict">{{character.district}}</span>
  </li>
</ul>
```

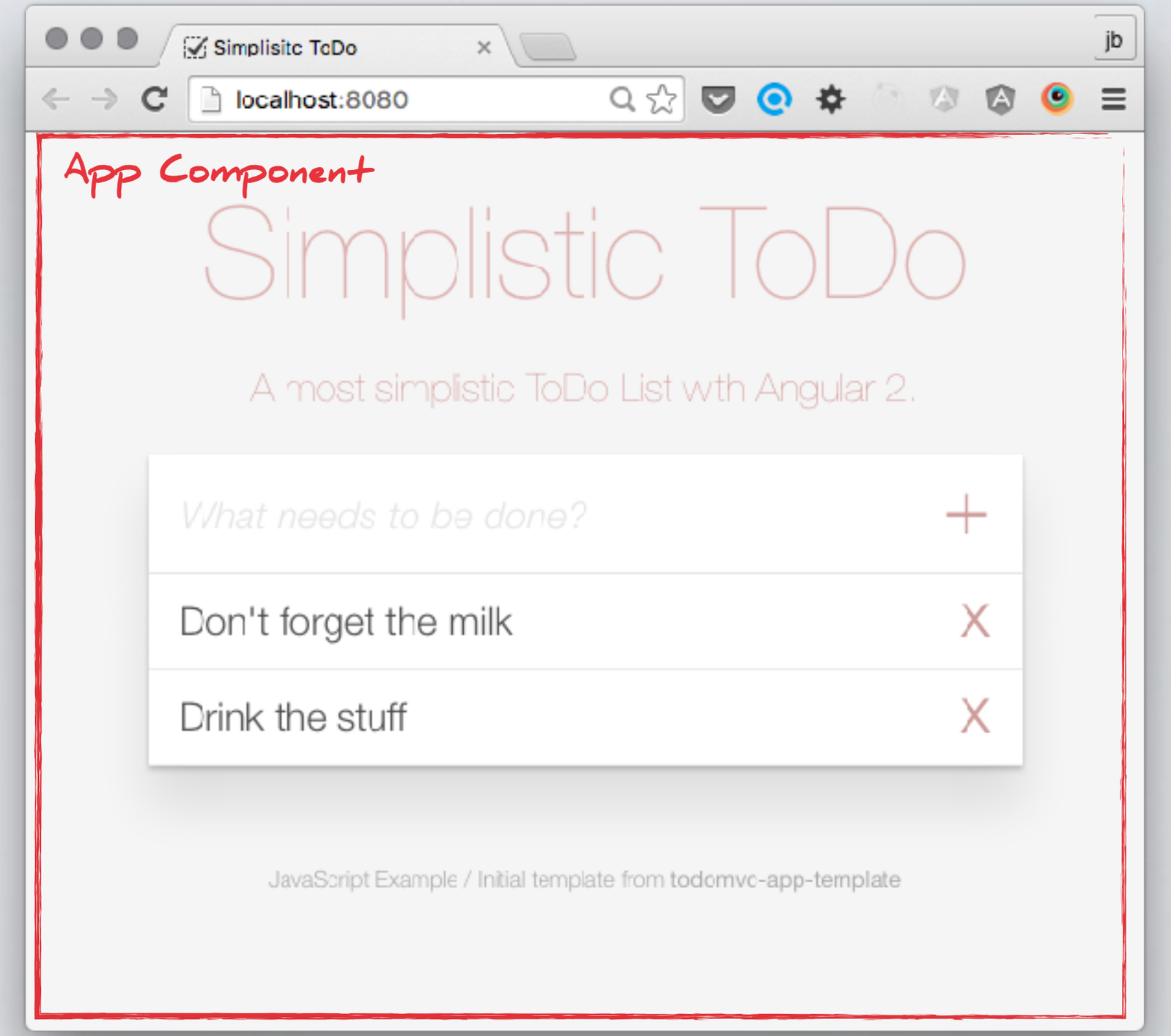
```
@Component({ ... })
export class StructuralDirectivesComponent {

  ...

  trackByFirstName(index: number, character: ICharacter) {
    return character.firstName;
  }
}
```

https://angular.io/guide/template-syntax#ngfor-with-trackby

Components: The Main Building Blocks



EXERCISES

Exercise 4.1 – ToDo App –Single Component

Component Lifecycle Hooks

Angular calls specific methods on a component during its life-cycle:

`ngOnChanges`	before **`ngOnInit`** and when a data-bound input property value changes.
`ngOnInit`	after the first **`ngOnChanges`**. Note: `@Input()` properties are set in contrast to the constructor
`ngDoCheck`	during every Angular change detection cycle.
`ngAfterContentInit`	after projecting content into the component. Note: `@ContentChild()` properties are set
`ngAfterContentChecked`	after every check of projected component content. Note: Can be used to get a changed value from a `@ContentChild()` property
`ngAfterViewInit`	after initializing the component's views and child views Note: `@ViewChild()` properties are set (especially elements of dynamic templates using i.e. `*ngFor`)
`ngAfterViewChecked`	after every check of the component's views and child views. Note: Can be used to get a changed value from a `@ViewChild()` property
`ngOnDestroy`	just before Angular destroys the directive/component.

Angular provides matching TypeScript interfaces: **`OnInit, AfterViewInit`** …

https://angular.io/guide/lifecycle-hooks

Services

Services are "Injectables"

"Service" is a broad category encompassing any value, function or feature that our application needs.

A service is typically a class with a narrow, well-defined purpose. It should do something specific and do it well.

It typically provides data and/or logic for other Angular constructs.

The easiest way to use services is to declare them as singletons:

```
@Injectable({providedIn: 'root'})
export class DataService {
  ...
}
```

Note: `{providedIn: 'root'}` is only available since Angular 6. Alternatively `providers` can be declared in modules and components.

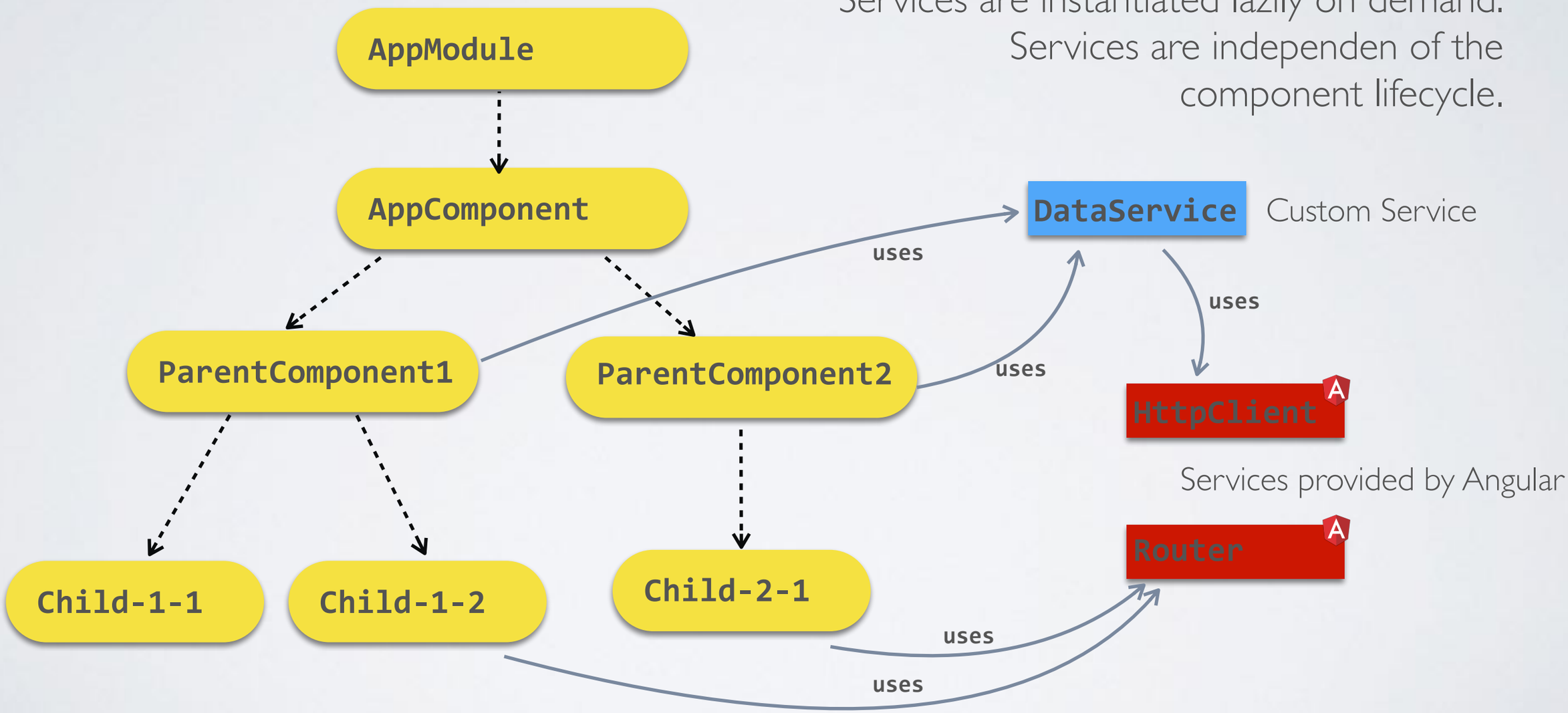
Services then can be requested via dependency-injection:

```
@Component({
    selector: 'my-component',
    template: 'path/template.html')
})
export class MyComponent {
  constructor(
     private dataService: DataService
  ){}
  ...
}
```

Components & Services

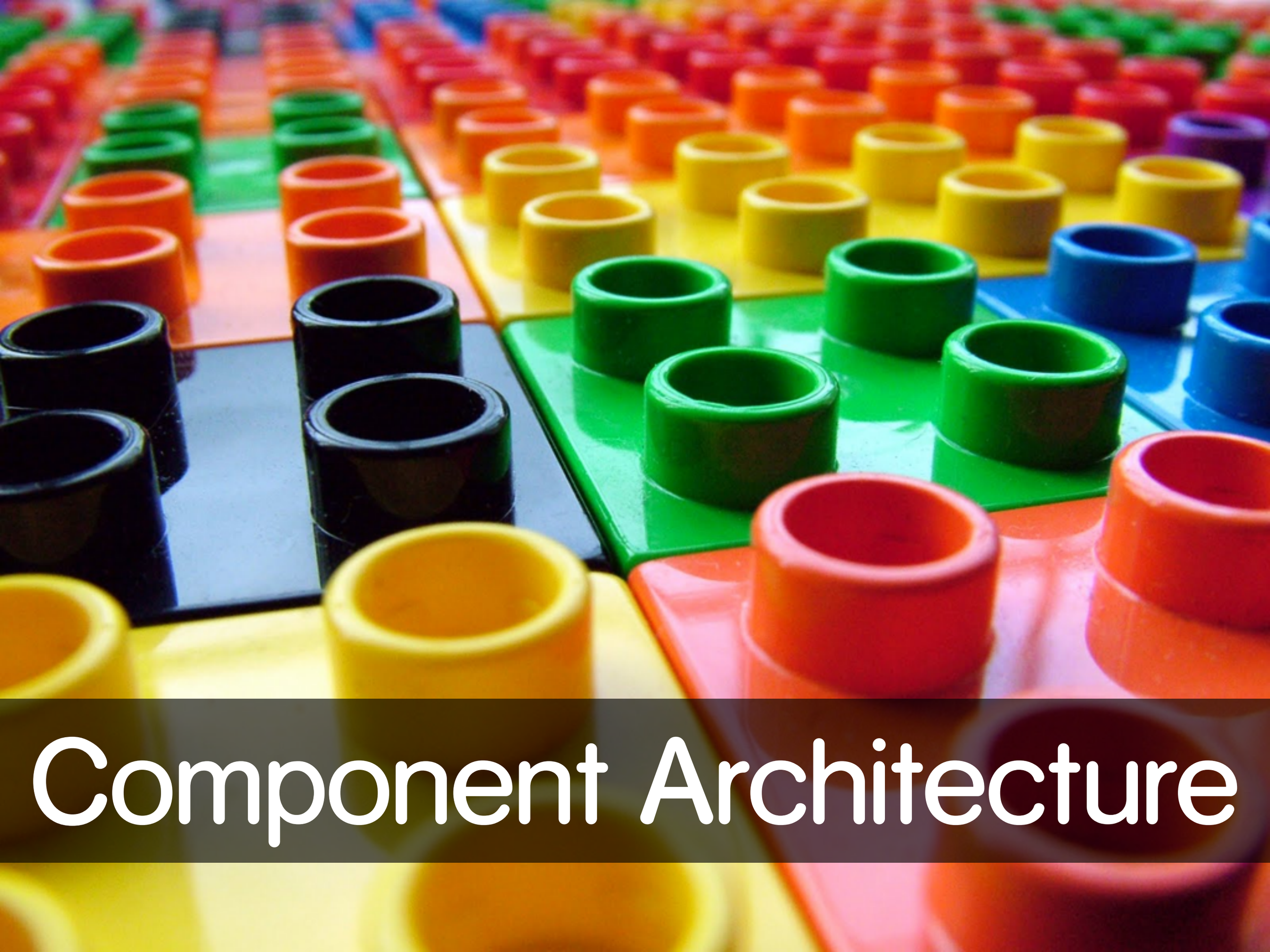
Services are objects outside of the component tree.

Services are instantiated by Angular.
Services are instantiated lazily on demand.
Services are independen of the component lifecycle.



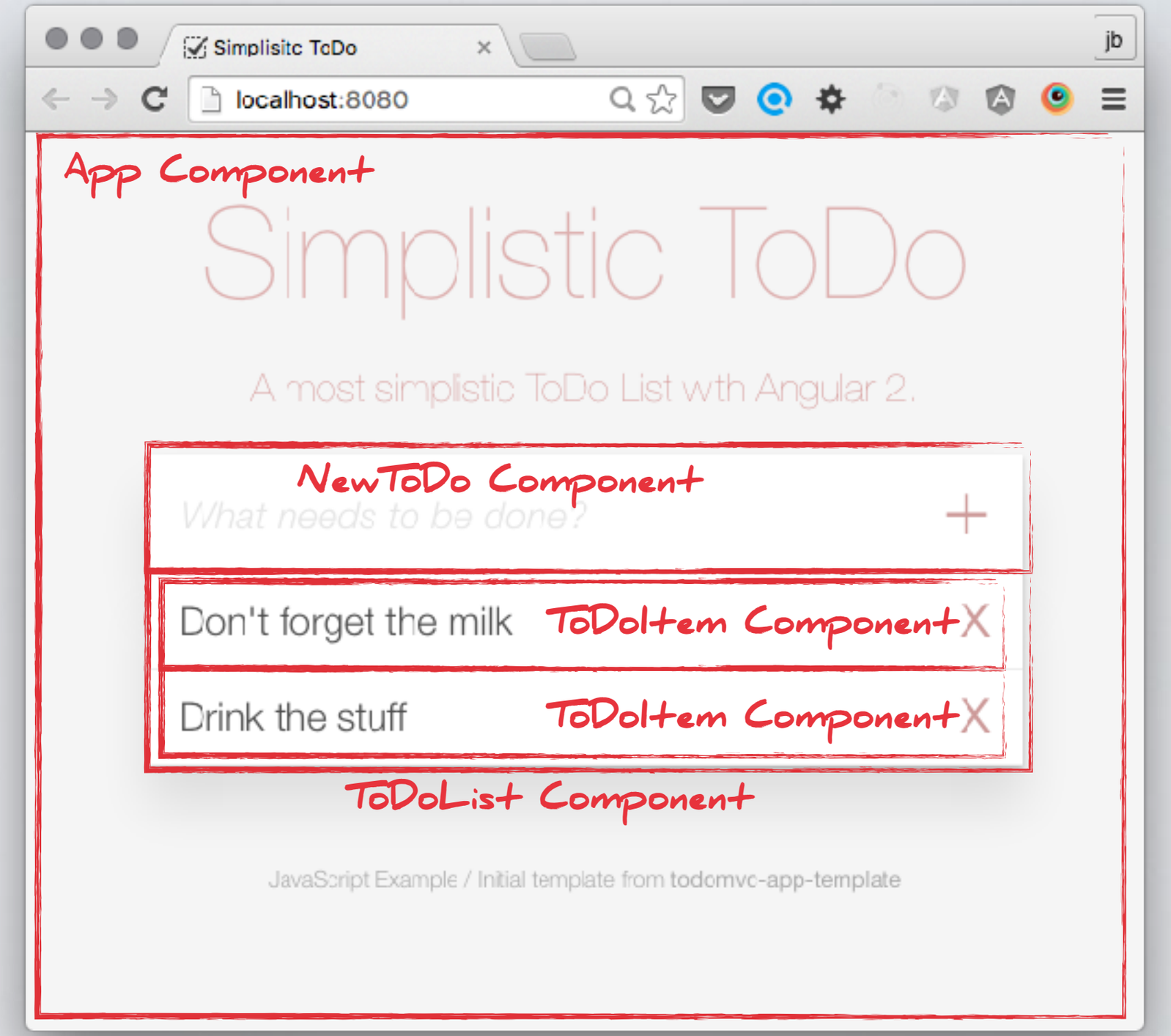
Services

- A service in Angular is just an object.
- There are built-in services provided by Angular:
  - `Router`, `ActivatedRoute`, `Location`, `Title`, `HttpClient` …
- You should write your own custom services …
  - … to structure your application.
  - … to share and reuse functionallity.
  - … to manage and share state outside of the component tree.



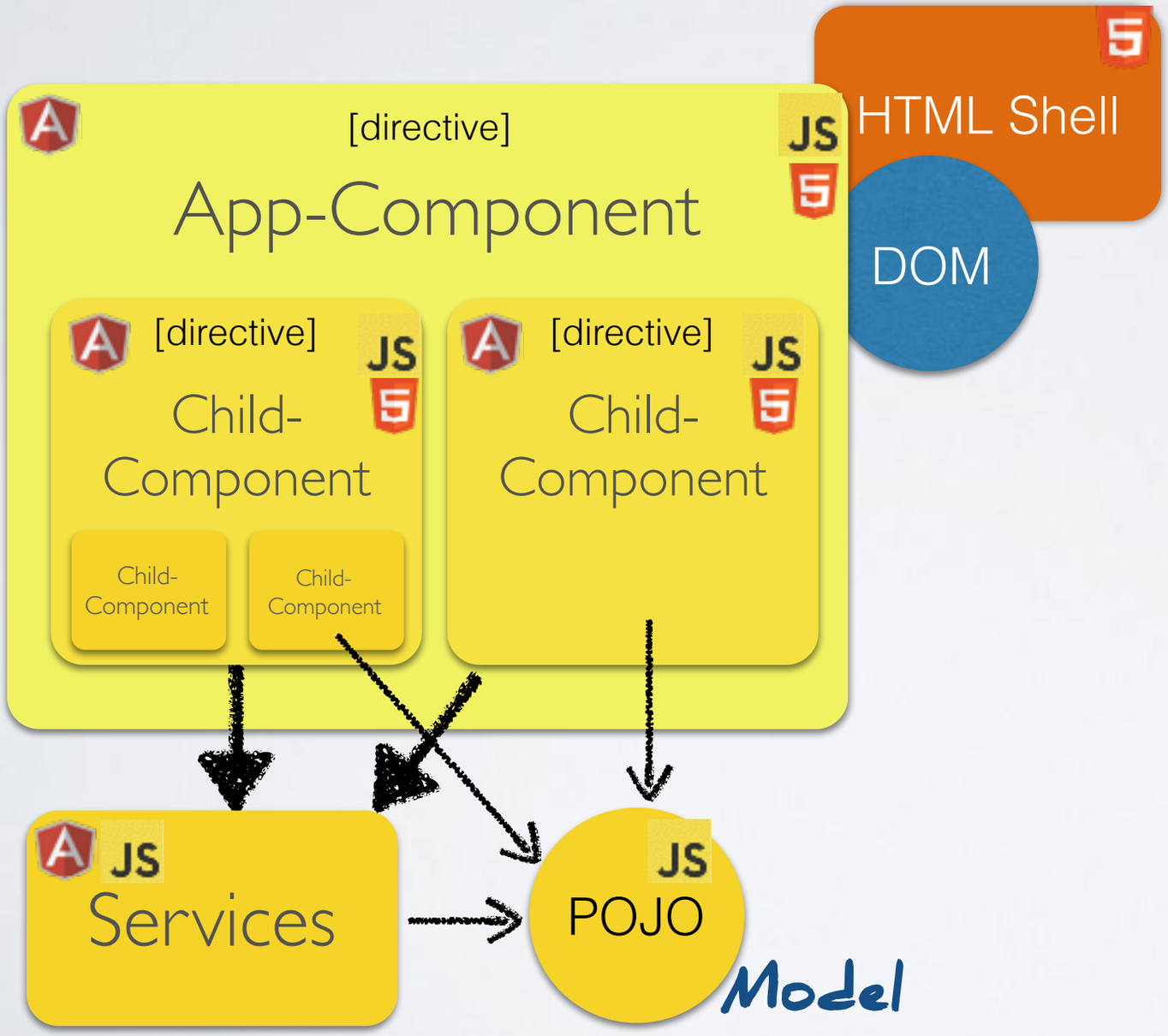
Component Architecture

Components: The Main Building Blocks



The Component Architecture

An app consists of a tree of components



Components are Angular directives, that are processed when compiling the html

Each component consists of a template and a class.

A component can be composed from other components.

similar approaches: React, VueJS, Web-Components/Polymer

Nested Components

A parent component can use child components in its template:

```
<div>
    <aw-child-list></aw-child-list>
</div>
```

To use a child component in a template, the child component must be declared or imported in the corresponding NgModule:

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

Using a Component as a Directive

index.html

```
<html>
<head> … </head>
<body>
  <my-app>Loading...</my-app>
</body>

</html>
```

parent component

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

```
<div>
  <h1>App</h1>
  <child-component></child-component>
</div>
```

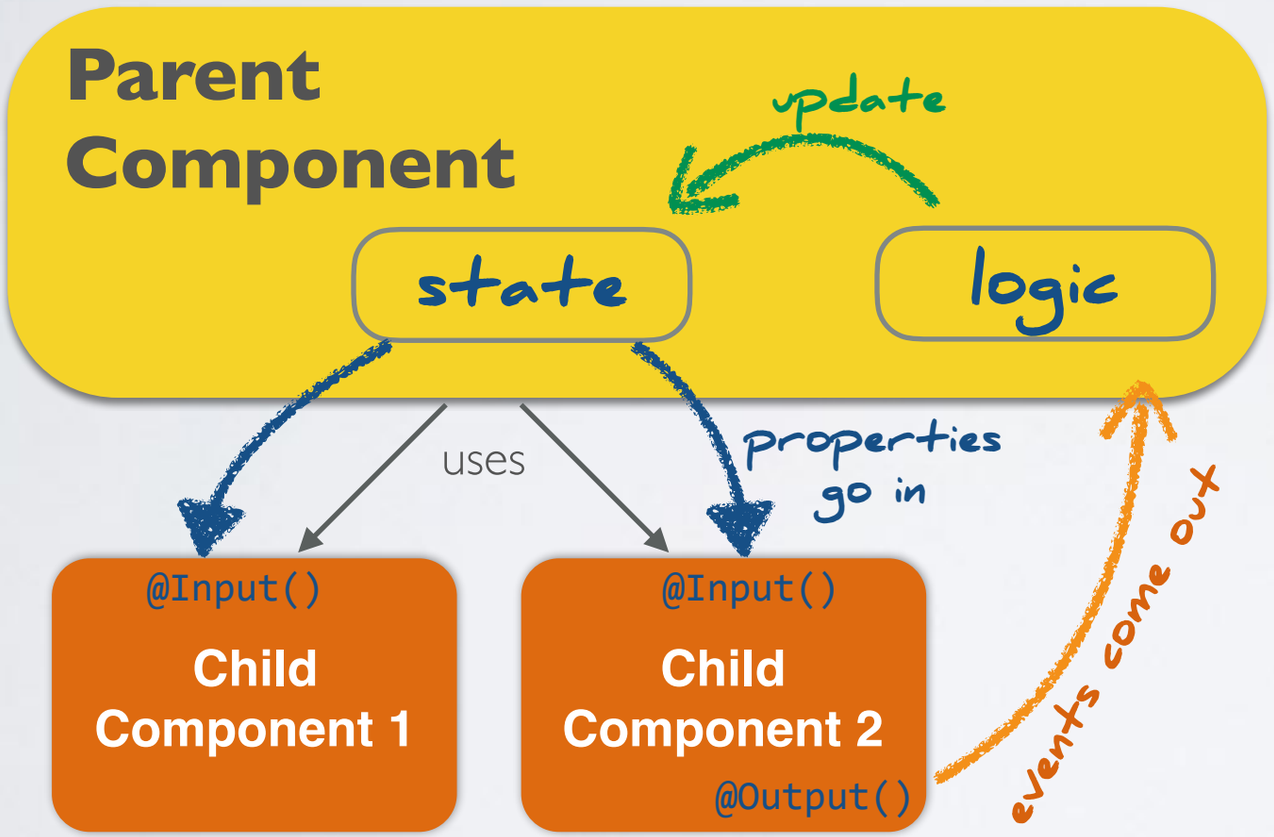
```
@NgModule({
  declarations: [AppComponent,
                 ChildComponent],
  imports     : [BrowserModule],
  bootstrap   : [AppComponent]
})
export class AppModule {}
```

```
@Component({
  selector: 'child-component',
  templateUrl: 'child.html',
})
export class ChildComponent { }
```

child component

Unidirectional Data-Flow

State should be explicitly owned by one component.
State should *never be duplicated* in multiple components.
(sometimes it is tricky to detect the *minimal state* from which other state properties can be *derived*)



- A parent component passes state to children as properties. Children are re-rendered when these properties change.
- Children should not edit state of their parent
- Children "notify" parents (events, callbacks …)

Angular formalises **unidirectional data-flow** with `@Input()` and `@Output()` properties.

Input- & Output-Properties

Angular property- and event-binding to DOM elements:

template.html

```
<p [innerText]="name" [style.color]="color"></p>
<input type="text" (change)="onChange($event)">
```

Input- and Output properties apply the same concept to application-specific data:

parent-template.html

```
<my-child [children]="persons"
    (addPerson)="onAddPerson($event)">
</my-child>
```

parent-component.ts

```
@Component({ ... })
export class ParentComponent {
  persons = [];
  onAddPerson(person){ ... }
}
```

child-template.html

```
<button (click)="onAddPerson()">Add</button>
<ul>
    <li *ngFor="let child of children">
        ...
    </li>
</ul>
```

child-component.ts

```
@Component({ ... })
export class ChildComponent {
  @Input() children;
  @Output() addPerson = new EventEmitter();
  onAddPerson(){addPerson.emit(...)}
}
```

Container vs. Presentation Components

A pattern for "Separtion of Concerns"

Applications can be decomposed in container- and presentation components:

Container	**Presentation**
little to no markup	mostly markup
pass data and handle events	receive data & actions via props
typically stateful / manage state	mostly stateless
	better reusability

aka: *Smart*- vs. *Dumb/Pure* Components

https://medium.com/@dan_abramov/smart-and-dumb-components-7ca2f9a7c7d0

Pattern: Container vs. Presentation Components

- dumb components have no side effects
- dumb components primary rely on `@input` and `@output` properties. They do not know more of their context.
- dumb components can have local state
  - if a component communicates with a service, it is probably a smart component

Try to have as many dumb components and as few smart components as possible in your application. This makes dataflow and side-effects of your applications very explicit and predictable.

https://medium.com/@jtomaszewski/how-to-write-good-composable-and-pure-components-in-angular-2-1756945c0f5b
https://toddmotto.com/stateful-stateless-components
https://medium.com/curated-by-versett/building-maintainable-angular-2-applications-5b9ec4b463a1
https://blog.angular-university.io/angular-2-smart-components-vs-presentation-components-whats-the-difference-when-to-use-each-and-why/
https://medium.com/@dan_abramov/smart-and-dumb-components-7ca2f9a7c7d0

EXERCISES

Exercise 4.2 & 4.3 – ToDo App – Component Architecture

More UI Concepts

Attribute Directives

Attribute directives are attached to elements:

```
<p awHighlight>I am green with envy!</p>
```

In the implementation you can access the element.

```
@Directive({ selector: '[awHighlight]' })
export class HighlightDirective {

  private defaultColor = 'yellow';

  constructor(private el: ElementRef, private renderer: Renderer2) {
    this.renderer
          .setStyle(this.el.nativeElement, 'background-color', color);
  }
}
```

https://angular.io/docs/ts/latest/guide/attribute-directives.html

Pipes

Pipes transform displayed values within a template.

```
<div> {{time | date}} </div>
```

Angular offers many built-in pipes:

`date, uppercase, lowercase, currency, percent, number, replace, slice, json …`

Pipes can be parameterized and chained:

```
<div> {{time | date:'longDate' | uppercase }} </div>
```

Note: There is no **`filter`** or **`orderBy`** pipe like in AngularJS
https://angular.io/guide/pipes#no-filter-pipe

https://angular.io/api?type=pipe

Custom Pipes

Custom pipes are implemented by a class with a **`transform`** method that is annotated with **`@Pipe`**:

```
@Pipe({name: 'camelCase'})
export class CamelCasePipe implements PipeTransform {
    transform(value:string, args:any[]):string { … }
}
```

Implementing **`PipeTransform`** is optional.

A component must declare the custom pipes he uses:

```
<div>
{{message | camelCase:true}}
</div>
```

https://angular.io/guide/pipes

Nested Components: Child-Access

Templates can declare and use local variables:

template.html

```
<my-child #child></my-child>
<button (click)="child.reset()">Reset</button>
```

Components can get access to child components via **`@ViewChild()`** and **`@ViewChildren()`**

template.html

```
<my-child></my-child>
<button (click)="doReset()">Reset</button>
```

component.ts

```
@Component({...})
export class ParentComponent{
  @ViewChild('child', {static: false}) myChild;
  doReset(){ myChild.reset() }
}
```

https://angular.io/api/core/ViewChild

Content Projection

(aka "Transclusion")

A component can have nested content:

parent.component.html

```
<my-container>
  <div>I am nested!</div>
</my-container>
```

result in the DOM

```
<div>Container Header</div>
<div>I am nested!</div>
<div>Container Footer</div>
```

The component has access to the nested content in its template via **`<ng-content>`**

container.component.html

```
<div>Container Header</div>
<ng-content></ng-content>
<div>Container Footer</div>
```

Selection of content: **`<ng-content select="selector">`**

Access to elements in the content of a component via **`@ContentChild()`** and **`@ContentChildren()`**

Nested Components vs. Nested Content

`@ViewChild` vs. `@ContentChild`

app.component.html

```
<my-parent></my-parent>
```

parent.component.html

```
<div>
  <h1>Hello from parent!</h1>
  <my-child></my-child>
</div>
```

child.component.html

```
<div>
  <h2>Hello from child!</h2>
</div>
```

app.component.html

```
<my-parent>
  <h1>Hello from content!</h1>
  <my-child></my-child>
</my-parent>
```

parent.component.html

```
<div>
  <h1>Hello from parent!</h1>
  <ng-content></ng-content>
</div>
```

Styling Components

With **Component Styles** Angular enables a more modular design than regular stylesheets.

```
@Component({
  selector: 'hero-app',
  template: `
    <h1>Tour of Heroes</h1>
    <hero-app-main [hero]=hero></hero-app-main>`,
  styles: ['h1 { font-weight: normal; }'],
})
```

instead of `styles` we can use `styleUrls`

Component styles are local to the component and can't be changed from outside.

Component styles are co-located to the component which leads to a better project structure.

Two-Way Binding in Depth

The "*Banana In A Box*" Syntax:

```
<aw-sizer [(size)]="fontSizePx"></aw-sizer>
```

Syntactic sugar for:

```
<aw-sizer [size]="fontSizePx" (sizeChange)="fontSizePx=$event"></aw-sizer>
```

The "generic" two-way binding is based of a convention of the naming of a property and the corresponding change event:

```
size <-> sizeChange
```

However native HTML elements do not follow this naming convention. The `NgModel` directive is a bridge that enables two-way binding to form elements:

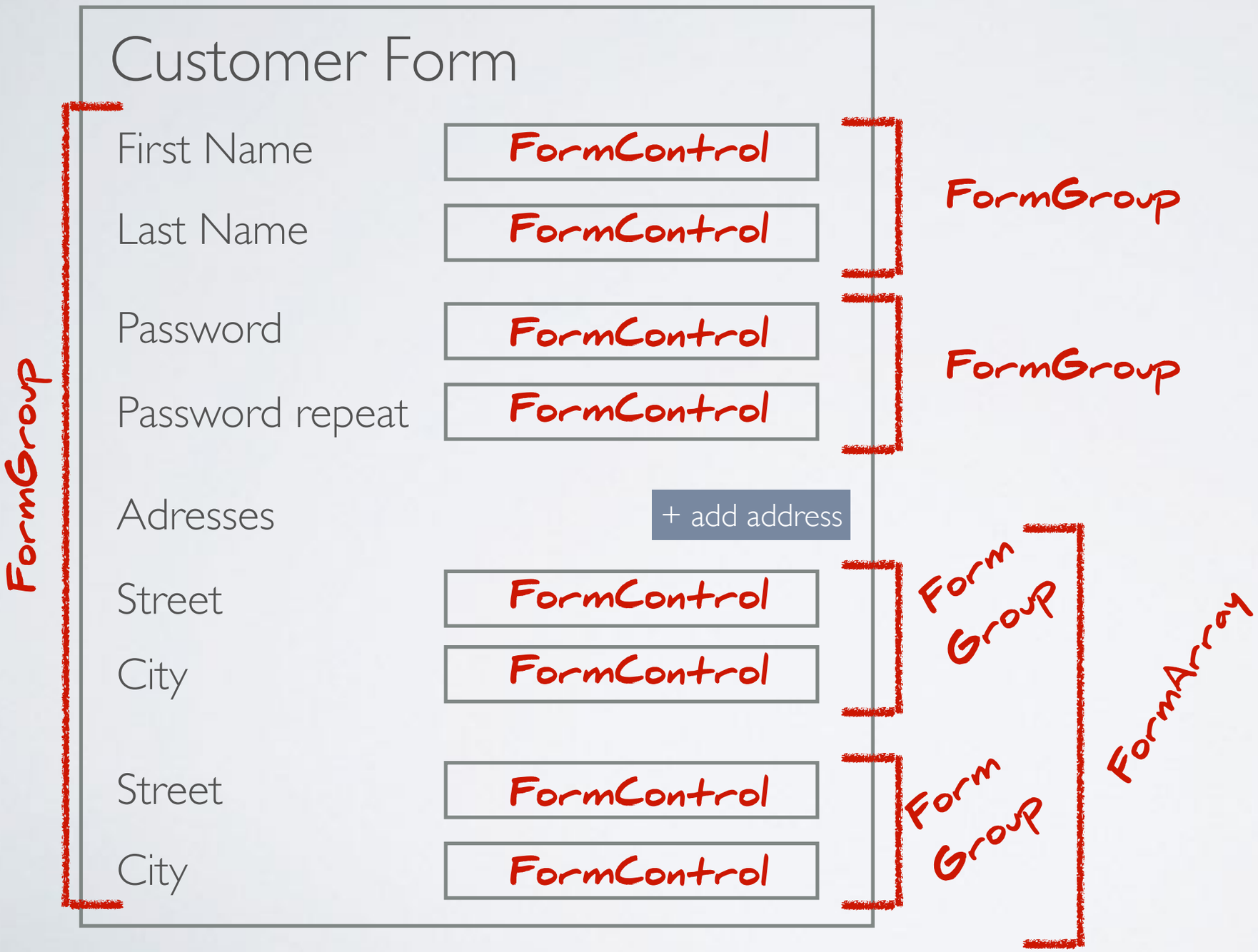
```
<input [(ngModel)]="myValue"/>
```

https://angular.io/guide/template-syntax#two-way-binding---

Angular Forms

The Form Controls

Angular provides *form controls* to model an ui-independent representation of a form.



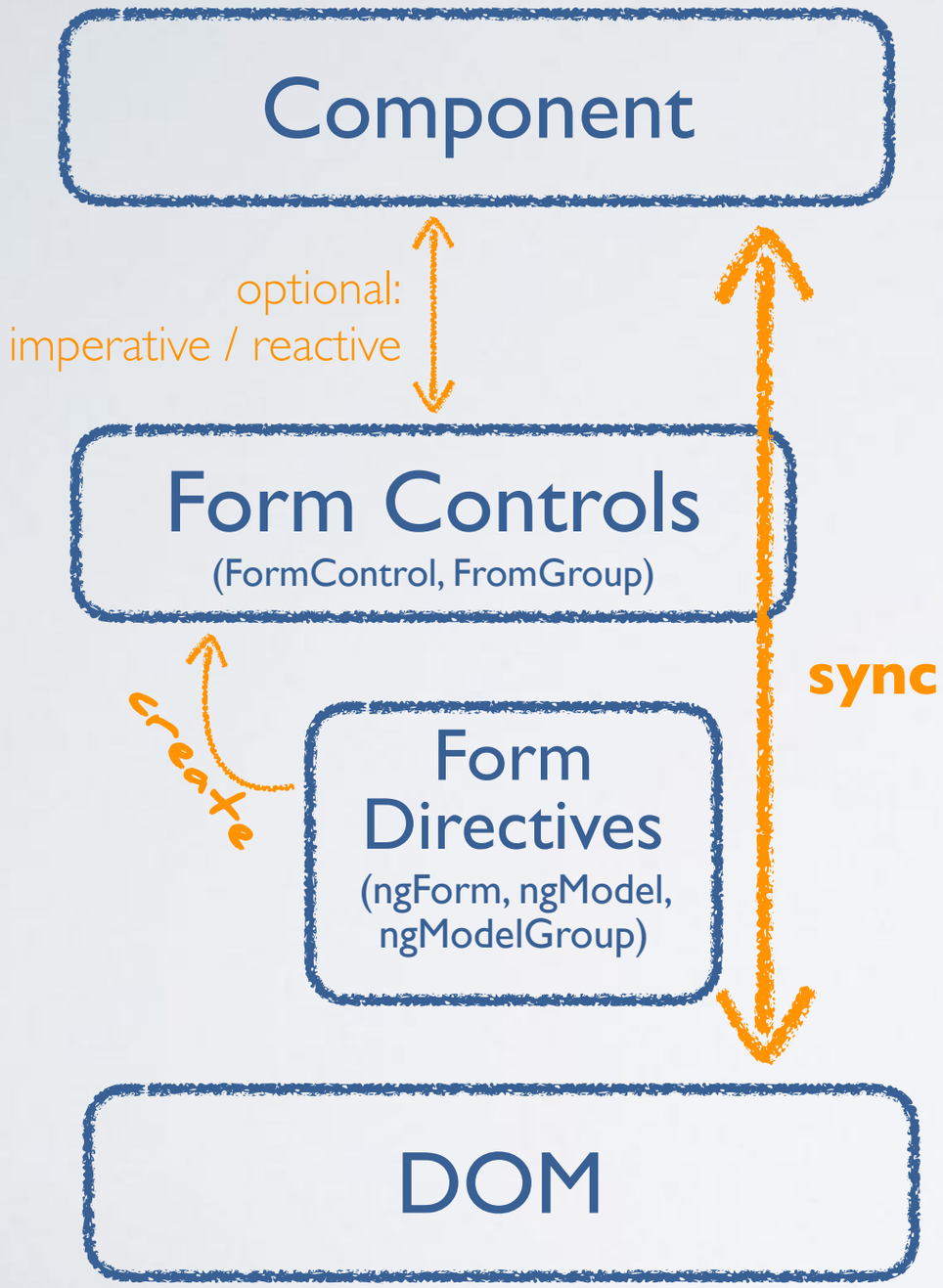
With *template driven* forms `FormGroup`s and `FormControl`s are created transparently by Angular.

With *reactive forms* `FormGroup`s, `FormArray`s and `FormControl`s are created explicitly in the component.

The Form Controls

FormControl	Representing an actual control: input, checkbox, radio button, select ... A `FormControl` can have validators. Angular maintains state for touched, dirty and validation. https://angular.io/api/forms/FormControl
FormGroup	The full form is represented as a from group. A form group typically contains `FormControl`s but can also be composed of other `FormGroup`s or `FormArray`s. Access by name. Perform cross field validation. Tracks the value and validity state of a group of `FormControl` instances. Can be observed for aggregated changes. https://angular.io/api/forms/FormGroup
FormArray	Tracks the value and validity state of an array of `FormControl`, `FormGroup` or `FormArray` instances. Access by index. Can be observed for aggregated changes. https://angular.io/api/forms/FormArray

Angular Template Driven Forms



Angular **`@Component`** class
provided by the app developer

ui independent representation of the form
building blocks provided by Angular and instantiated by Angular

directives connecting the form controls to the DOM
provided by Angular used by the developer

representation of the UI in the browser

https://angular.io/guide/forms-overview#data-flow-in-template-driven-forms
https://blog.nrwl.io/angular-forms-in-depth-ecb7c58166b5

Template Driven Forms

With template driven forms angular creates a form model based on the template.

Angular provides the `FormsModule` via `@angular/forms`

This module provides directives to enhance forms for databinding, validation and change tracking:

`ngModel`, `ngModelGroup`, `ngForm` …

```
<form #formRef="ngForm" (ngSubmit)="onSubmit(formRef.value)">

    <fieldset ngModelGroup="login">
        <input name="username" type="text" required
               [(ngModel)]="username" #usernameRef="ngModel">
        <div *ngIf="usernameRef.errors?.required">This field is required</div>
        <input name="password" type="password" ngModel>
    </fieldset>
    <button type="submit">Submit!</button>
</form>
```

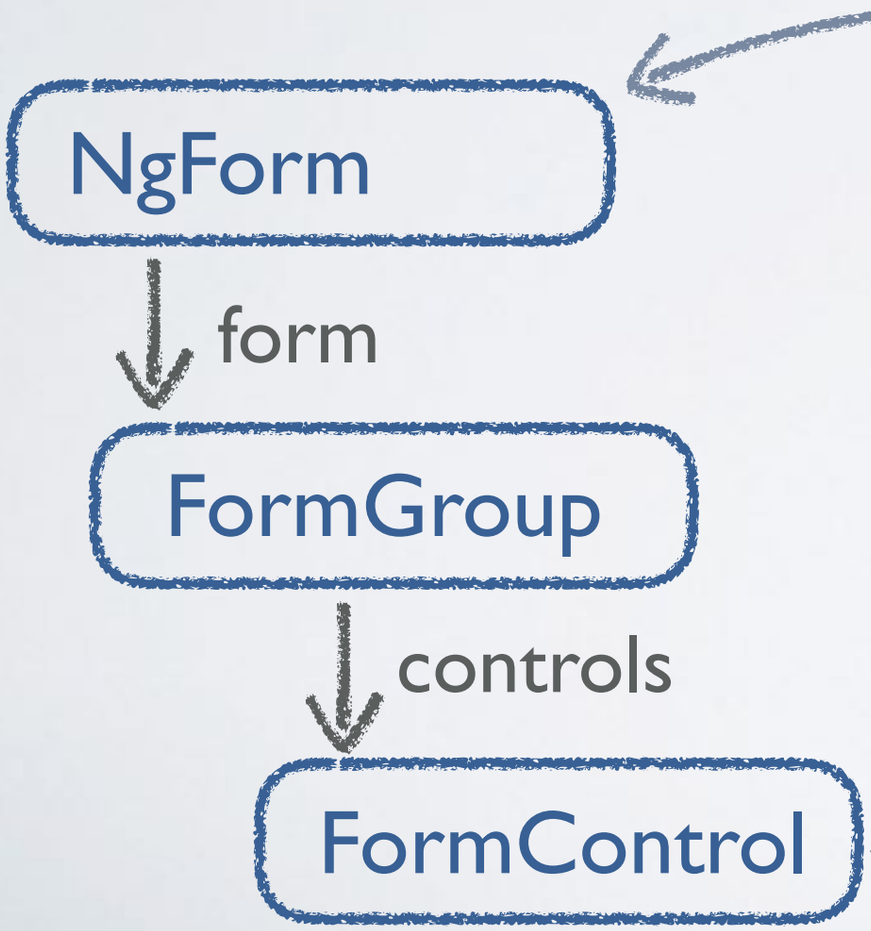
Note: The `name` attribute must be set on any form field. Angular needs a name to create the form model.

Note: You could also handle the native `submit` event, but handling ngSubmit `ngSubmit` ensures that the form doesn't submit when the handler code throws an error.

https://angular.io/docs/ts/latest/guide/forms.html

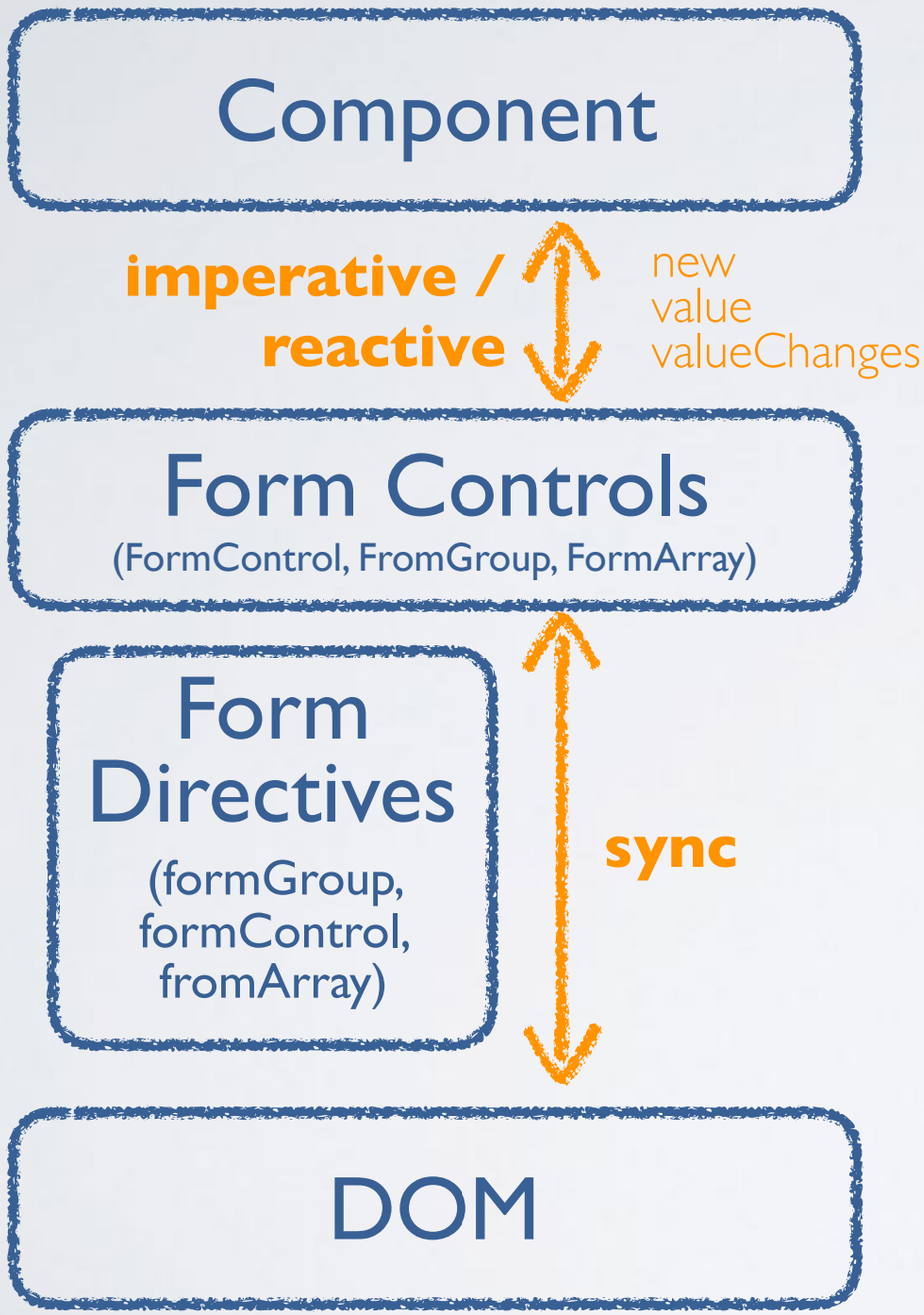
Template Driven Forms

- Angular automatically attaches a **`ngForm`** to each **`<form>`**
- **`ngModel`** and **`ngModelGroup`** create the form model and attach it to the **`ngForm`**
- **`ngForm`**, **`ngModel`** and **`ngModelGroup`** expose the form model




```
<form #formRef="ngForm">
  <input name="username"
    type="text" required
    [(ngModel)]="username"
    #usernameRef="ngModel">
</form>
```

Angular Reactive Forms



Angular **`@Component`** class
provided by the app developer

ui independent representation of the form
building blocks provided by Angular and instantiated by the developer

directives connecting the form controls to the DOM
provided by Angular used by the developer

representation of the UI in the browser

https://angular.io/guide/forms-overview#data-flow-in-reactive-forms
https://blog.nrwl.io/angular-forms-in-depth-ecb7c58166b5

Reactive Forms

With reactive forms, the form is explicitly modelled in code and the UI binds to that model.

Angular provides the `ReactiveFormsModule` via `@angular/forms`
This module provides directives to bind form-elements to a form-model: `formGroup`, `formControlName`, `formControl` …

```
<form [formGroup]="theForm"
      (ngSubmit)="onSubmit()">

    <div>
        <label for="name">Name</label>
        <input type="text"
            id="name"
            placeholder="Name"
            formControlName="name">
    </div>
</form>
```

```
@Component({
   ...
})
export class FormComponent {
 private theForm: FormGroup;

 constructor(){
   this.theForm = new FormGroup({
      'name': new FormControl(),
      'password': new FormControl()
    });
 }
 ...
}
```

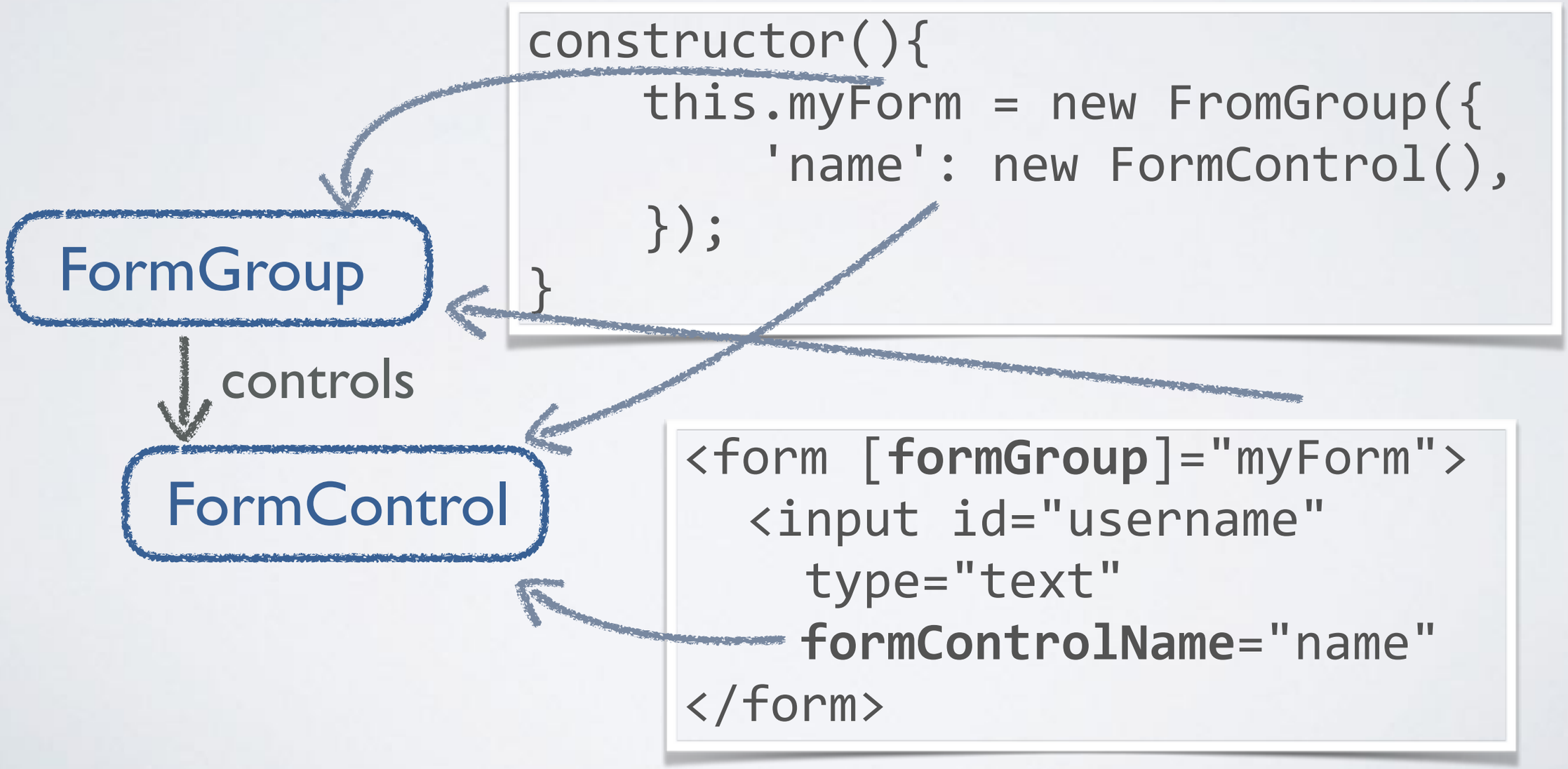
Note: The `name` attribute defines the key in the value property of the form. If `name` is omitted, `formControlName` is used as key.

Note: The `FormBuilder` service provides a fluent API for creating FormGroup, FromArray & FormControl

http://blog.thoughtram.io/angular/2016/06/22/model-driven-forms-in-angular-2.html
http://blog.angular-university.io/introduction-to-angular-2-forms-template-driven-vs-model-driven/

Reactive Forms

- The form model is explicitly created in the component
- `formGroup` and `formControl` bind the form model to the template



EXERCISES

Exercise 5 – Forms

Dynamic Forms

Dynamically create a reactive form and the matching template from a from description.

component.ts

```
export class DynamicFormComponent implements OnInit {
  formDescription = staticFormDescription; // could be loaded from DB
  form: FormGroup = new FormGroup({});
  formTemplateFields = [];

  ngOnInit() {
    for (const [fieldName, fieldDescription] of Object.entries(this.formDescription)) {
        // add templata metadata to formTemplateFields
    }
  }
}
```

component.html

```
<form [formGroup]="form">
  <div *ngFor="let field of formTemplateFields">
    <div [ngSwitch]="field.type">

      <!-- create fomr field and connect it to the from instance -->

    </div>
  </div>
</form>
```

Guide: https://angular.io/guide/dynamic-form

Library: https://github.com/formly-js/ngx-formly

Big Picture: Form Validation

Angular form validation is bound to single forms.

If you have validation that spans multiple forms (i.e. a wizard with multiple pages that each contain a form), then further validation has typically to be implemented in the component / model.

Often it's then unavoidable to have some duplication betweet this custom validation code and the form validation.

Template Driven vs. Reactive Forms

- The form model is directly bound to the app model
- Validation driven by template
- declarative style is simpler for simple forms and static scenarios
- Testing only possible with a DOM

- The form model is an explicit layer holding state
- Validation driven by code
- Better for implementing dynamic scenarios and complex cross-field validation
- Better testability (unit-tests, i.e. validation logic)

http://blog.angular-university.io/introduction-to-angular-2-forms-template-driven-vs-model-driven/

Custom Form Control

Using a custom form control:

```
<form>
  ...
  <aw-star-rating name="rating" [(ngModel)]="rating"></aw-star-rating>
  ...
</form>
```

custom form control

```
<form [formGroup]="myForm">
  ...
  <aw-star-rating formControlName="rating"></aw-star-rating>
  ...
</form>
```

Implementing a custom form control:

```
export class StarRatingComponent implements ControlValueAccessor {
    writeValue(obj: any){...}
    registerOnChange(fn: any){...}
    registerOnTouched(fn: any){...}
    setDisabledState(isDisabled: boolean){...}
    ...
}
```

A custom form control must implement `ControlValueAccessor`.

A custom form control must be registered with Angular:

```
providers: [
  {
    provide: NG_VALUE_ACCESSOR,
    useExisting: forwardRef(() =>
                 StarRatingComponent),
    multi: true
  }
]
```

https://angular.io/api/forms/ControlValueAccessor

Form Composition

Form controls can be created by nested components.

```
export class ParentComponent implements OnInit {
    myForm: FormGroup;
    ...
}
```

```
<aw-form-child
      [address]="myForm.controls['address']">
</aw-form-child>
```

top-level form group is passed to the child component

```
export class FormChildComponent implements OnInit
{
    @Input() address: FormGroup;
    ...
}
```

the parent component creates the top-level form controls

the child component creates the nested form controls and adds them to the parent from group.

Form Composition

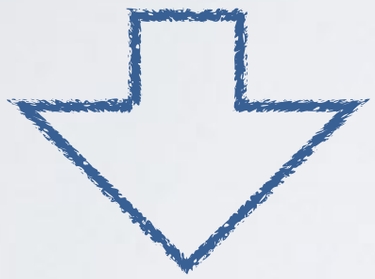
Custom form controls can be used to wrap nested form sections.

```
export class ParentComponent implements OnInit {
    myForm: FormGroup;
    ...
}
```

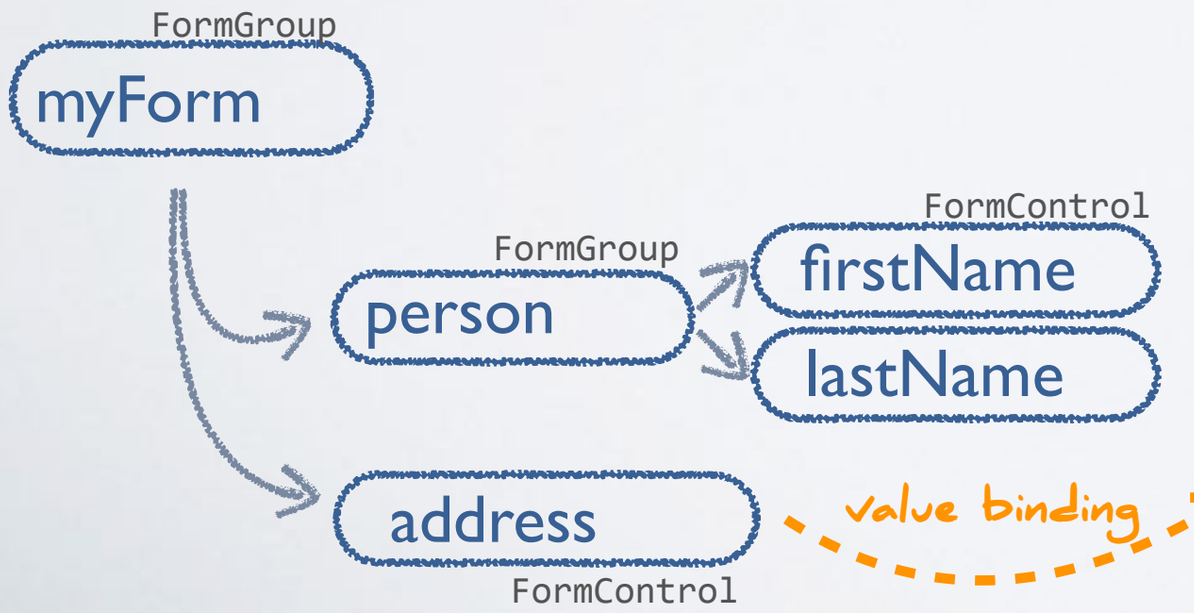
```
<aw-form-child
      formControlName="address">
</aw-form-child>
```

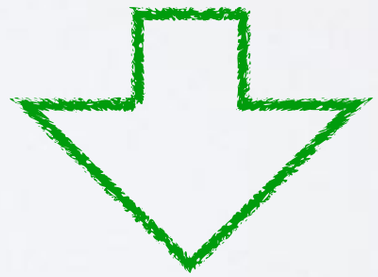
child component is a custom form control

```
export class FormChildComponent
        implements OnInit, ControlValueAccessor
 {
    address: FormGroup;
    ...
}
```

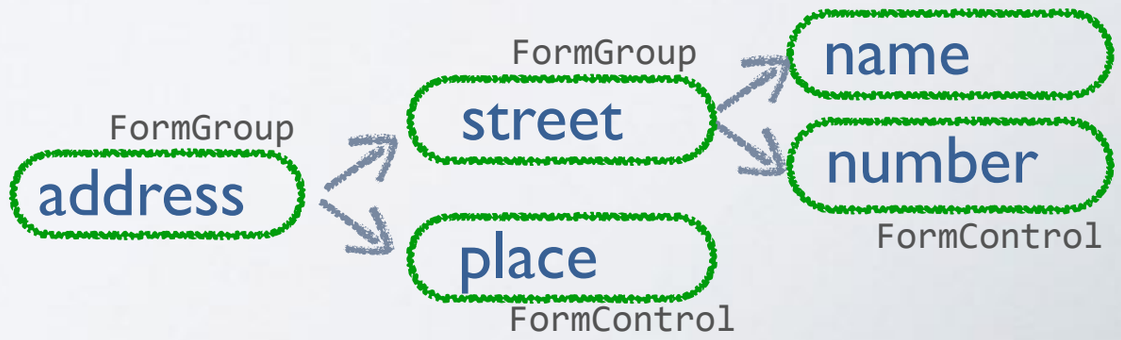


The parent component creates the top-level form controls



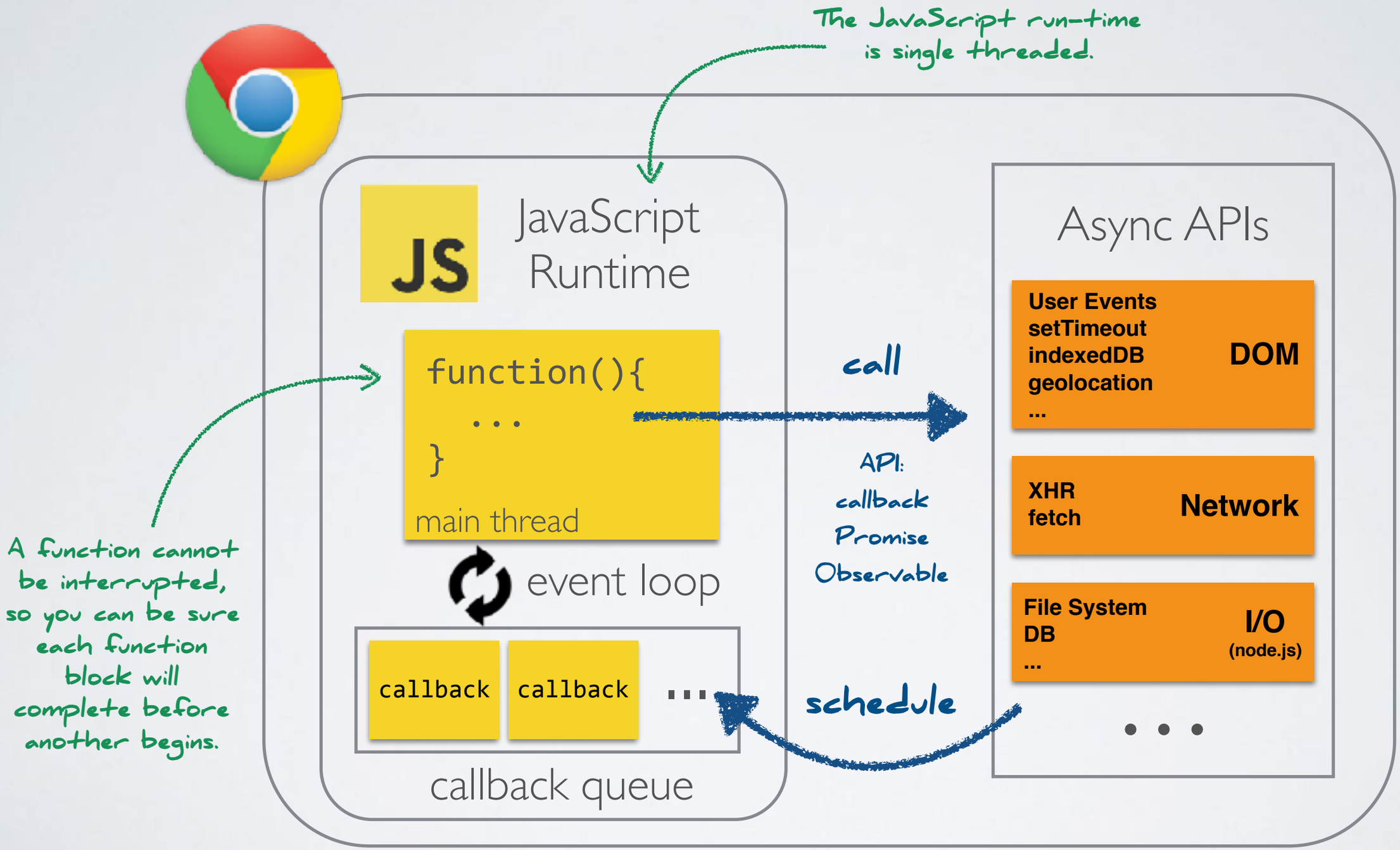


the child component contains a form of which the values are bound to the object of the parent form control.



RxJS / Observables

Concurrency in Javascript: Event Loop



https://www.tedinski.com/2018/10/16/concurrency-vs-parallelism.html

Observables

An observable represents a multi-value push protocol:

	Single	Multiple
Pull	`Function`	Iterator
Push	`Promise`	`Observable`

Typical scenario: an asynchronous stream of events.

```
const observable =
  rxjs.fromEvent(
    document.getElementById('my-button'),'click'
  );

observable.subscribe(x => console.log(x));
```

Observables in ECMAScript

RxJS is the "de-facto" implementation of observables today.

RxJS

http://reactivex.io/rxjs

https://rxjs-dev.firebaseapp.com/

Angular uses RxJS 6

RxJS is a library for composing asynchronous and event-based programs by using observable sequences.

There is a proposal to standardize `Observable` as built-in type in a future version of ECMAScript.

https://github.com/tc39/proposal-observable

There are alternative libraries with similar reactive programming concepts:
Beacon.js, xstream, …

https://baconjs.github.io/
http://staltz.com/xstream/

Angular uses RxJS

RxJS is a peer dependency of Angular:

```
"dependencies": {
  "@angular/animations": "~7.1.0",
  "@angular/common": "~7.1.0",
  "@angular/compiler": "~7.1.0",
  "@angular/core": "~7.1.0",
  "@angular/forms": "~7.1.0",
  "@angular/http": "~7.1.0",
  "@angular/platform-browser": "~7.1.0",
  "@angular/platform-browser-dynamic": "~7.1.0",
  "@angular/router": "~7.1.0",
  "core-js": "^2.5.4",
  "rxjs": "~6.3.3",
  "zone.js": "~0.8.26"
},
```

peer dependency

package.json

Angular uses RXJS heavily internaly.

Angular exposes many APIs based on Observables.

Many 3rd party libraries for Angular are built on top of RxJS

Examples of Angular APIs:

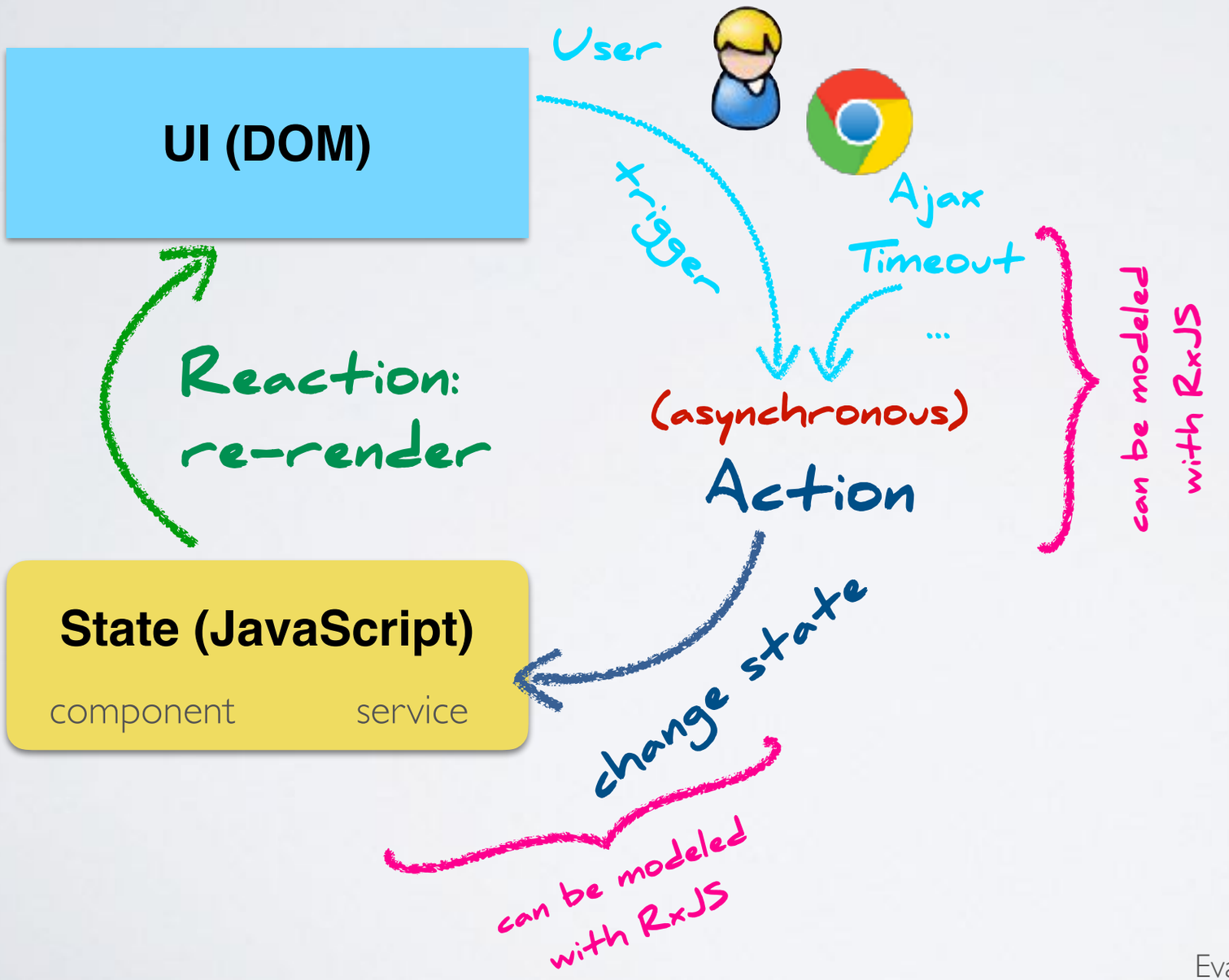
```
httpClient.get(URL)
  .subscribe(
    response => ...
  );
```

```
form.valueChanges
  .subscribe(
    fromData => ...
  );
```

```
route.url
  .subscribe(
    urlSegments => ...
  );
```

Reactivity in a SPA

State is managed in JavaScript.
The UI renders the state and emits events.



There are two mechanisms to implement reactivity in Angular:

- Default Change Detection
- Observables (RxJS)

Evan You - Reactivity in Frontend JavaScript Frameworks:
https://www.youtube.com/watch?v=r4pNEdIt_I4
http://teropa.info/blog/2015/03/02/change-and-its-detection-in-javascript-frameworks.html

Rich Harris

@Rich_Harris

Following

The problem all frameworks are solving is *reactivity*. How does the view react to change?

- React: 'we re-render the world'
- Vue: 'we wrap your data in accessors'
- Svelte: 'we provide an imperative set() method that defeats TypeScript'
- Angular: 'zones' (actually idk 🤷‍♂️)

1:01 PM - 3 Nov 2018

https://twitter.com/Rich_Harris/status/1058751085706383362

What is RxJS?

Reactive programming is an asynchronous programming paradigm concerned with data streams and the propagation of change (Wikipedia).

RxJS (Reactive Extensions for JavaScript) is a library for reactive programming using observables that makes it easier to compose asynchronous or callback-based code.

https://angular.io/guide/rx-library

RxJS is a library

RxJS is a library with an API. It makes it possible to model reactive programming constructs *explicitly in your code*:
Observables, Observers, Subjects, Subscriptions ...

Transparent Reactivity is a concept that makes reactivity an *implicit characteristic* of your program.

- Default change detection in Angular is an example of transparent reactivity.
- MobX is a library that provides transparent reactivity based on observable models

https://github.com/meteor/docs/blob/version-NEXT/long-form/tracker-manual.md

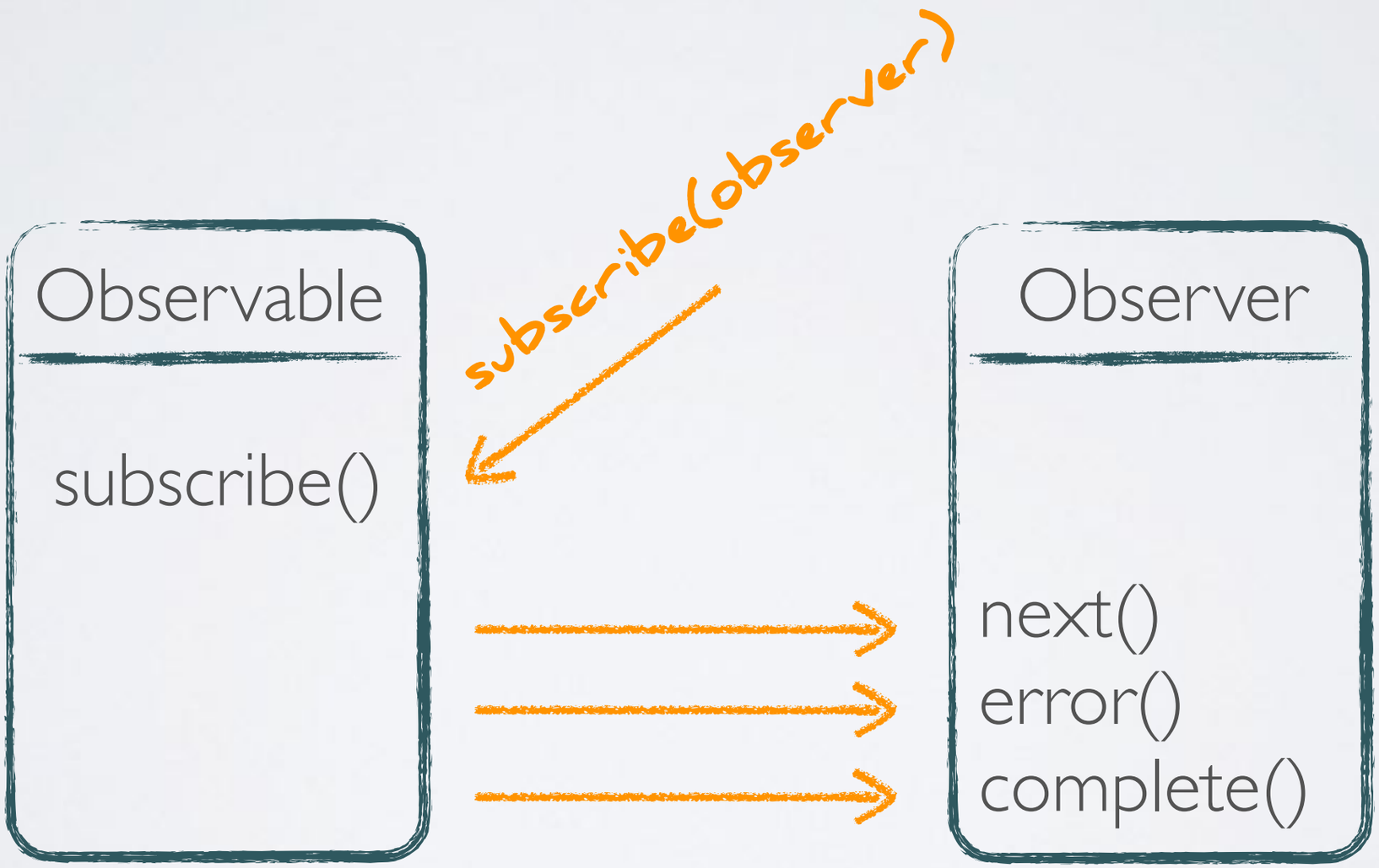
Observables

Observable Characteristics

- Represents any number of values over any amount of time
- Observables are able to deliver values either synchronously or asynchronously
- Observables are lazy
- Observables can be cancelled
- Observables can be composed with higher-order combinators

RxJS:
"The Observer pattern done right"

http://reactivex.io/



Using Observables

Consume an observable:

```
var value$ = startAsyncOperation();
value$.subscribe(
  value => document.getElementById("content").innerText += value,
  error => console.log('Error: ' + error),
  () => console.log('Completed!')
);
```

Provide an observable (using RxJS):

```
function startAsyncOperation(){
   var value$ = new rxjs.Observable(
     observer => {
       setTimeout(() => observer.next("This is first value!"), 1000);
       setTimeout(() => observer.next("This is second value!"), 2000);
       setTimeout(() => observer.next("This is third value!"), 3000);
       setTimeout(() => observer.complete(), 4000);
     });
   return value$;
}
```

Note: In typical application programming you *never* have to create an observable like this.
You either use factory functions or you consume an API that exposes an "event source" in an observable.

Observables vs. Promises

An observable can be an alternative to a promise:
a stream that pushes exactly one result.

```
const observable = new rxjs.Observable(
   observer => setTimeout(
       () => {
            observer.next(42);
            observer.complete();
       },
       2000);
);

observable.subscribe(x => console.log(x));
```

Angular exposes observables for backend access via `HttpClient` service.

Promises vs. Observables

Promise	**Observable**
returns a single value	works with multiple values over time
cancellable via `AbortController`	cancellable
enables the `async/await` syntax of ES2017	supports powerful operators like map, filter & reduce (“lo-dash for async”)

For AJAX calls observables do not provide a clear advantage. But observables can be used consistently for all kind of asnc operations.

https://developer.mozilla.org/en-US/docs/Web/API/AbortController
https://developers.google.com/web/updates/2017/09/abortable-fetch
http://blog.thoughtram.io/angular/2016/01/06/taking-advantage-of-observables-in-angular2.html
Do Observables make sense for http? - https://github.com/angular/angular/issues/5876

Sync vs. Async Observables

sync:

```
of(1, 2, 3, 4, 5)
  .subscribe(
    value => console.log(value),
  );
console.log('Done 1');
```

async:

```
interval(1000)
  .subscribe(
    value => console.log(value),
  );
console.log('Done 2');
```

Creating Observable Streams

Creation Functions:

`of, from, fromEvent, interval, timer ...`

Manual Creation:

```
new Observable(observer => {
  observer.next(1);
  observer.next(2);
  observer.complete();
});
```

```
new Observable(observer => {
  setTimeout(() =>  observer.next(1), 1000);
  setTimeout(() =>  observer.next(2), 2000);
  setTimeout(() =>  observer.complete(), 3000);
});
```

Note: In typical application programming you *never* have to manually create an observable.
You either use factory functions or you consume an API that exposes an "event source" in an observable.

Subscription & Unsubscribe

A subscrition is modeled by a **`Subscription`** object.

```
const subscription = interval(1000)
  .subscribe(
    value => console.log(value),
    error => console.log(error),
    () => console.log('COMPLETED')
  );

setTimeout(() => {
  console.log(subscription.closed);
  subscription.unsubscribe();
}, 4000)
```

There is a potential memory leak, if you are not unsubscribing from an Observable!

Cold vs. Hot Observables

A cold obervable creates the producer:

```
var cold = new Observable((observer) => {
  var producer = new Producer();
  // have observer listen to producer here
});
```

Each subscription creates it's own producer. The producer only starts producing with the subscription.

A hot obervable closes over the producer:

```
var producer = new Producer();
var hot = new Observable((observer) => {
  // have observer listen to producer here
});
```

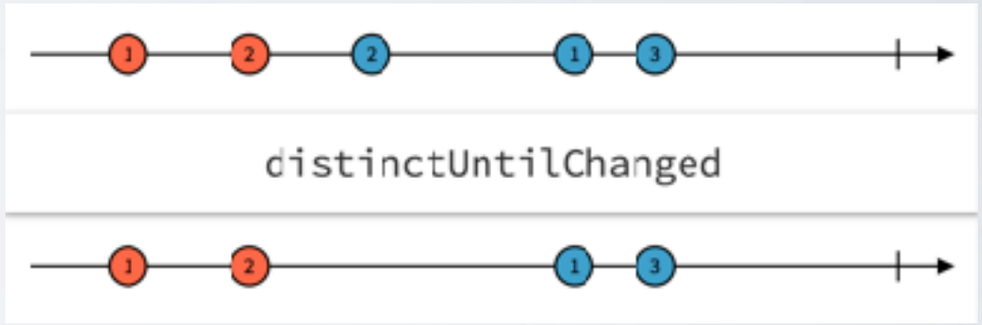
The producer produces values independent of subscriptions.

https://medium.com/@benlesh/hot-vs-cold-observables-f8094ed53339

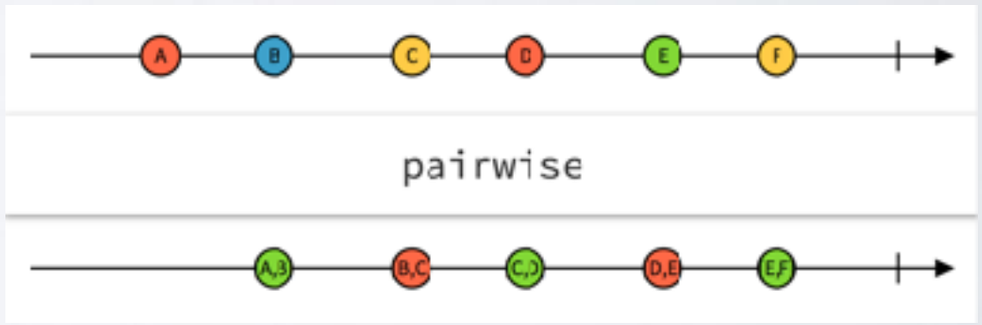
Transformation Operators

```
const {filter, map} = rxjs.operators;
const o1 = rxjs
            .interval(1000)
            .pipe(
                 filter(v => v % 2 === 0),
                 map(v => v * 2)
             );
```

```
const {distinctUntilChanged} = rxjs.operators;
const transformed = o1.
              .pipe(
                 distinctUntilChanged()
              );
```




```
const {pairwise} = rxjs.operators;
const transformed = o1
              .pipe(
                 pairwise()
              );
```




```
transformed.subscribe(
  v => console.log(v)
);
```

http://reactivex.io/documentation/operators
http://rxmarbles.com/

Combination Operators

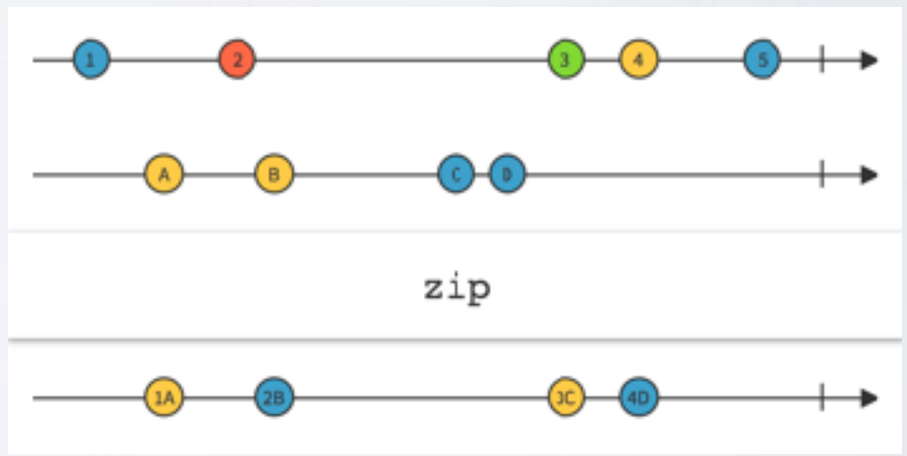
```
const o1 = rxjs.interval(1000);
const o2 = rxjs.interval(1000)
  .delay(500)
  .map(v => v * 1000);
```

```
const combined = rxjs.merge(o1,o2);

combined.subscribe(
  v => console.log(v)
);
```


```
const combined = rxjs.zip(o1,o2);

combined.subscribe(
  v => console.log(v)
);
```



http://reactivex.io/documentation/operators
http://rxmarbles.com/

Higher Order Observables

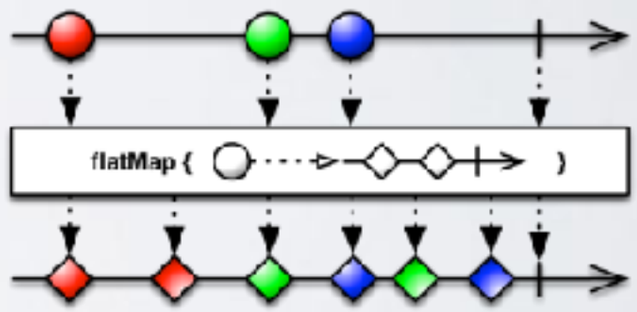
"Observables of Observables"

```
function api1(){
  return rxjs.of(42).delay(2000);
}
function api2(){
  return rxjs.of(43).delay(1000);
}
function api3(){
  return rxjs.of(44).delay(500);
}

stream = rxjs.from([api1, api2, api3]);
```

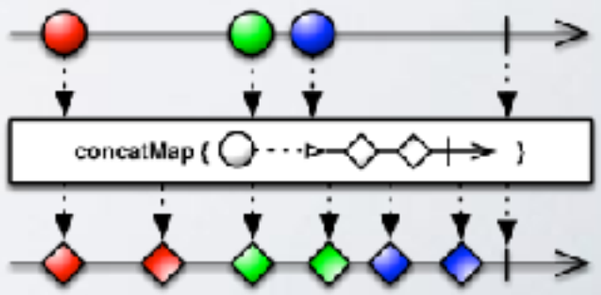
```
stream.flatMap(x => x())
      .subscribe(
          value => console.log(value)
      );
```

out:
44, 43, 42




```
stream.concatMap(x => x())
      .subscribe(
          value => console.log(value)
      );
```

out:
42, 43, 44



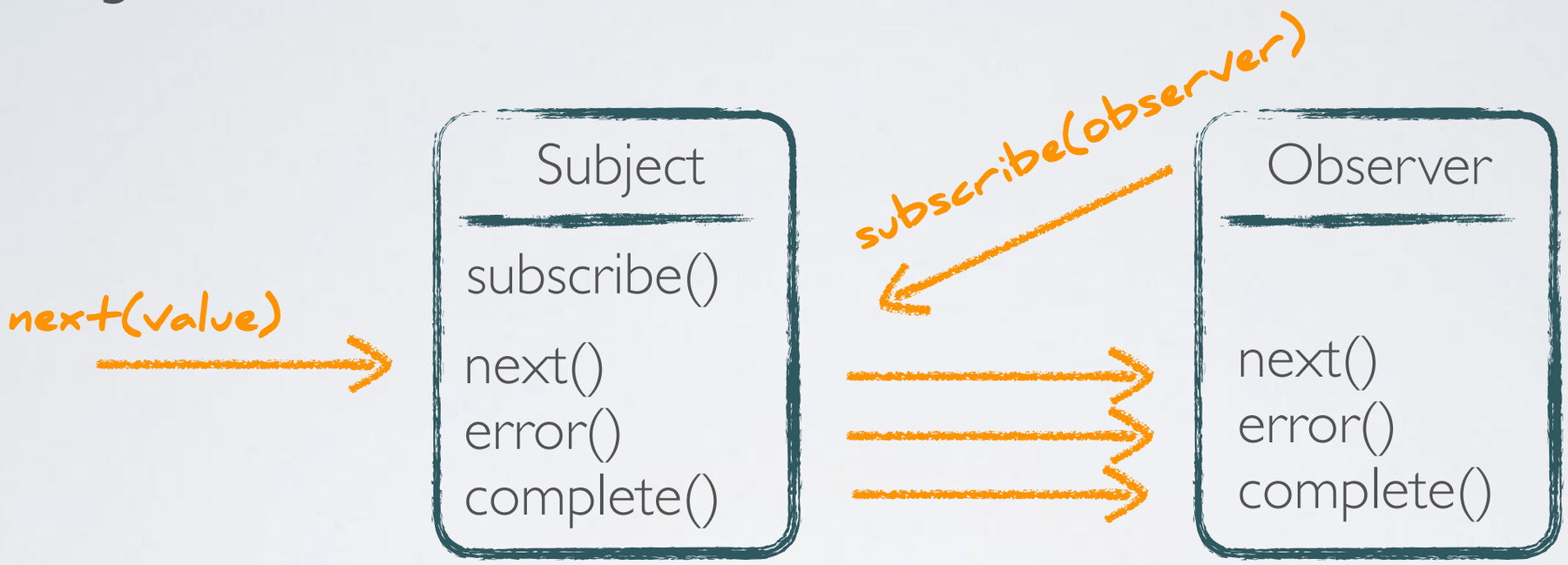
Funny: Switch Map Explained

NgConf 2018: I switched a map and you'll never guess what happened next

https://www.youtube.com/watch?v=rUZ9CjcaCEw

Subject

A **`Subject`** is a **`Observable`** and a **`Observer`** at the same time.



Subjects can be used as "manually triggered" streams.
Subjects can be used for multicasting to several Observers.

```
const subject = new rxjs.Subject();

subject.subscribe(
  value => console.log(value),
);

subject.next(43);
```

BehaviorSubject

A **`BehaviorSubject`** is a special **`Subject`**, which has a notion of "the current value". It stores the latest value emitted to its consumers, and whenever a new **`Observer`** subscribes, it will immediately receive the "current value" from the **`BehaviorSubject`**.

`BehaviorSubjects` are useful for representing "values over time". For instance, an *event stream of birthdays* is a **`Subject`**, but the *stream of a person's age* would be a **`BehaviorSubject`**.

```
const subject = new rxjs.BehaviorSubject(42);

subject.subscribe(
  value => console.log(value),
);

subject.next(43);
```

Scenario: Parallel Fetching

```
const id$ = of(1, 2, 3, 4, 5, 6, 8, 9, 10);

id$
  .pipe(
      map(id => `https://swapi.co/api/people/${id}/`),
      mergeMap(url => ajax.get(url))
  )
  .subscribe(result => console.log(result.response.name));
```

Scenario: Sequential Fetching

```
const id$ = of(1, 2, 3, 4, 5, 6, 8, 9, 10);

id$
  .pipe(
    map(id => `https://swapi.co/api/people/${id}/`),
    concatMap(url => ajax.get(url))
  )
  .subscribe(result => console.log(result.response.name));
```

Scenario: Side-Effect

```
let count = 0;

interval(200)
  .pipe(
    take(10),
    tap(v => count++),
    filter(v => v % 2 === 0)
  )
  .subscribe(
    value => console.log('Even: ' + value),
    null,
    () => console.log('Processed: ' + count)
  );
```

Scenario: Error Handling

```
of(1)
  .pipe(
    map(id => `https://swapi.co/api/ERROR/${id}/`),
    mergeMap(url => ajax.get(url)),
    filter(result => result.nonexistent.property),
    catchError(e => of('Handled', 'More'))
  )
  .subscribe(
    value => console.log(value)
  );
```

Common Task: Retry

```
of(1)
  .pipe(
    mergeMap(id => getData(id)),
    retry(5)
  )
  .subscribe(
    value => console.log(value)
  );
```

Common Task: Debouncing

```
const button = document.getElementById('clicker');

fromEvent(button, 'click')
  .pipe(
    tap(() => console.log('Click received!')),
    debounceTime(1000)
  )
  .subscribe(
    value => console.log('Processing click ...')
  );
```

Scenario: Sharing a Subscription

```
const value$ = interval(200)
  .pipe(
    take(5),
    share()
  );

value$.subscribe(
  value => console.log('Subscription 1:' + value),
);
setTimeout(() => {
  value$.subscribe(
    value => console.log('Subscription 2:' + value),
  )
}, 450);
```

Scenario: Avoid Subjects

```
let count = 0;
const emitter = new Subject();

const button = document.getElementById('clicker');
button.addEventListener('click', increaseAndSignal);

function increaseAndSignal() {
  count++;
  emitter.next(count)
}

emitter
  .subscribe(
    value => console.log('Current count: ' + count),
  );
```

A Subject is a indirection that introduces state and breaks the flow between producer and consumer.

Direct streams from producer to consumer are more explicit.

```
const button = document.getElementById('clicker');
fromEvent(button, 'click')
  .pipe(
    scan((acc, val) => acc + 1, 0)
  )
  .subscribe(
    value => console.log('Current count: ' + value),
  );
```

Scenario: Use switchMap as default flattening strategy

```
const click$ = fromEvent(document, 'click');

const tickWhenClick$ = click$
  .mergeMap(ev => interval(500));

tickWhenClick$.subscribe(function (x) {
  console.log(x)|| displayInPreview(x);
});
```

inner Observable emits forever

never unsubscribes from inner Observables

only subscribes to latest inner Observable

unsubscribes from previous inner Observables

```
const click$ = Rx.Observable
  .fromEvent(document, 'click');

const tickWhenClick$ = click$
  .switchMap(ev => Rx.Observable.interval(500));

tickWhenClick$.subscribe(function (x) {
  console.log(x)|| displayInPreview(x);
});
```

Avoid possible leaks resulting from **`mergeMap`**.

Demo:

Type-ahead Wikipeida search

Talk Recommendation

"Solving Real Problems with RxJs"

https://www.youtube.com/watch?v=W8T3eqUEOSI
https://speakerdeck.com/d3lm/rxjs-patterns-towards-formalising-common-rxjs-patterns
https://github.com/KwintenP/rxjs-recipes-talk

EXERCISES

Exercise 6 – RxJS – Numberguess

RxJS has changed recently ...

RxJS <5.5:

```
import 'rxjs/Rx';
```

```
import { range } from 'rxjs/observable/range';
import 'rxjs/add/operator/filter';
import 'rxjs/add/operator/map';
import 'rxjs/add/operator/catch';

const source$ = range(0, 10);
const stream$ = source$
  .filter(x => x % 2 === 0),
  .map(x => x + x),
  .scan((acc, x) => acc + x, 0);

stream$.subscribe(x => console.log(x))
```

RxJS 5.5:

introduced "pipeable operators":

```
import { range } from 'rxjs/observable/range';
import { map, filter, scan } from 'rxjs/operators';

const source$ = range(0, 10);
const stream$ = source$.pipe(
  filter(x => x % 2 === 0),
  map(x => x + x),
  scan((acc, x) => acc + x, 0)
);

stream$.subscribe(x => console.log(x))
```

exposed global `Rx` when loaded as script:

```
Rx.Observable.range(1,10)
      .subscribe(x => console.log(x));
```

RxJS 6:

renaming of "namespaces":

```
import {range} from 'rxjs'; // all creation methods, utilities
import {map, filter, scan} from 'rxjs/operators';
```

exposes global `rxjs` when loaded as script:

```
rxjs.range(1,10)
        .subscribe(x => console.log(x));
```

https://github.com/ReactiveX/rxjs/blob/master/MIGRATION.md

Including RxJS into an Angular Project

Angular includes only a minimal subset of RxJS.

Angular 4 and earlier: If you want to use more operators of RxJS (like map and filter), you import a module that patches observables (just once in your app):

everything of RxJS … not recommended!

```
import 'rxjs/Rx';
```

only the operators that are needed:

```
import 'rxjs/add/observable/throw';
import 'rxjs/add/operator/filter';
import 'rxjs/add/operator/map';
import 'rxjs/add/operator/catch';
```

Depricated in Angular 5

Angular 5 and later uses RxJS 5.5 or later which provides "pipeable operators":

```
import { Observable, of, range } from 'rxjs';
import { map, filter, scan } from 'rxjs/operators';

const source$ = range(0, 10);
source$.pipe(
  filter(x => x % 2 === 0),
  map(x => x + x),
  scan((acc, x) => acc + x, 0)
)
.subscribe(x => console.log(x))
```

You have to import the operators you need into each file where you use them in.

https://github.com/ReactiveX/rxjs/blob/master/doc/pipeable-operators.md

Angular & RxJS: Unsubscribe Patterns

You must unsubscribe from observables that do not complete, else you potentially produce a memory leak! **async** pipes unsubscribes automatically.

When subscribing to an observable in a component, you almost always arrange to unsubscribe when the component is destroyed.

https://angular.io/guide/router#observable-parammap-and-component-reuse

```
export class CounterComponent
              implements OnInit, OnDestroy {

  name = 'World';
  private unsubscribe = new Subject();

  ngOnInit(){
    interval(1000)
          .pipe(takeUntil(this.unsubscribe))
          .subscribe(
            value => {
              console.log(this.name);
              this.name = '' + value;
            }
          );
  }

  ngOnDestroy() {
   this.unsubscribe.next();
  }
}
```

Should typically be the last operator in the pipe

```
export class CounterComponent
              implements OnInit, OnDestroy {

  name = 'World';
  private alive = true;

  ngOnInit(){
    interval(1000)
          .pipe(takeWhile(() => this.alive))
          .subscribe(
            value => {
              console.log(this.name);
              this.name = '' + value;
            }
          );
  }

  ngOnDestroy() {
   this.alive = false;
  }
}
```

Note: just setting alive=false does not yet unsubscibe!

https://brianflove.com/2016/12/11/anguar-2-unsubscribe-observables/
https://blog.angularindepth.com/the-best-way-to-unsubscribe-rxjs-observable-in-the-angular-applications-d8f9aa42f6a0
https://blog.angularindepth.com/rxjs-avoiding-takeuntil-leaks-fb5182d047ef

Subsink: A Unsubscribe Helper

https://github.com/wardbell/subsink

```
export class CounterComponent
             implements OnInit, OnDestroy {

  name = 'World';
  private subs = new SubSink();

  ngOnInit(){
    ...
    this.subs.sink = observable$.subscribe(...); // easy syntax

    this.subs.add(observable$.subscribe(...)); // if you insist

    this.subs.add( // can add multiple subcriptions
      observable$.subscribe(...),
      anotherObservable$.subscribe(...)
    );
  }

  ngOnDestroy() {
    this.subs.unsubscribe();
  }
}
```

setter