Frontend-Entwicklung mit Angular

Mail: jonas.bandi@ivorycode.com

Twitter: @jbandi

AGENDA

DAY 1

- Intro
- Angular CLI
- Angular Components
- Angular Routing
- Basic Constructs
- ToDo App

DAY 2

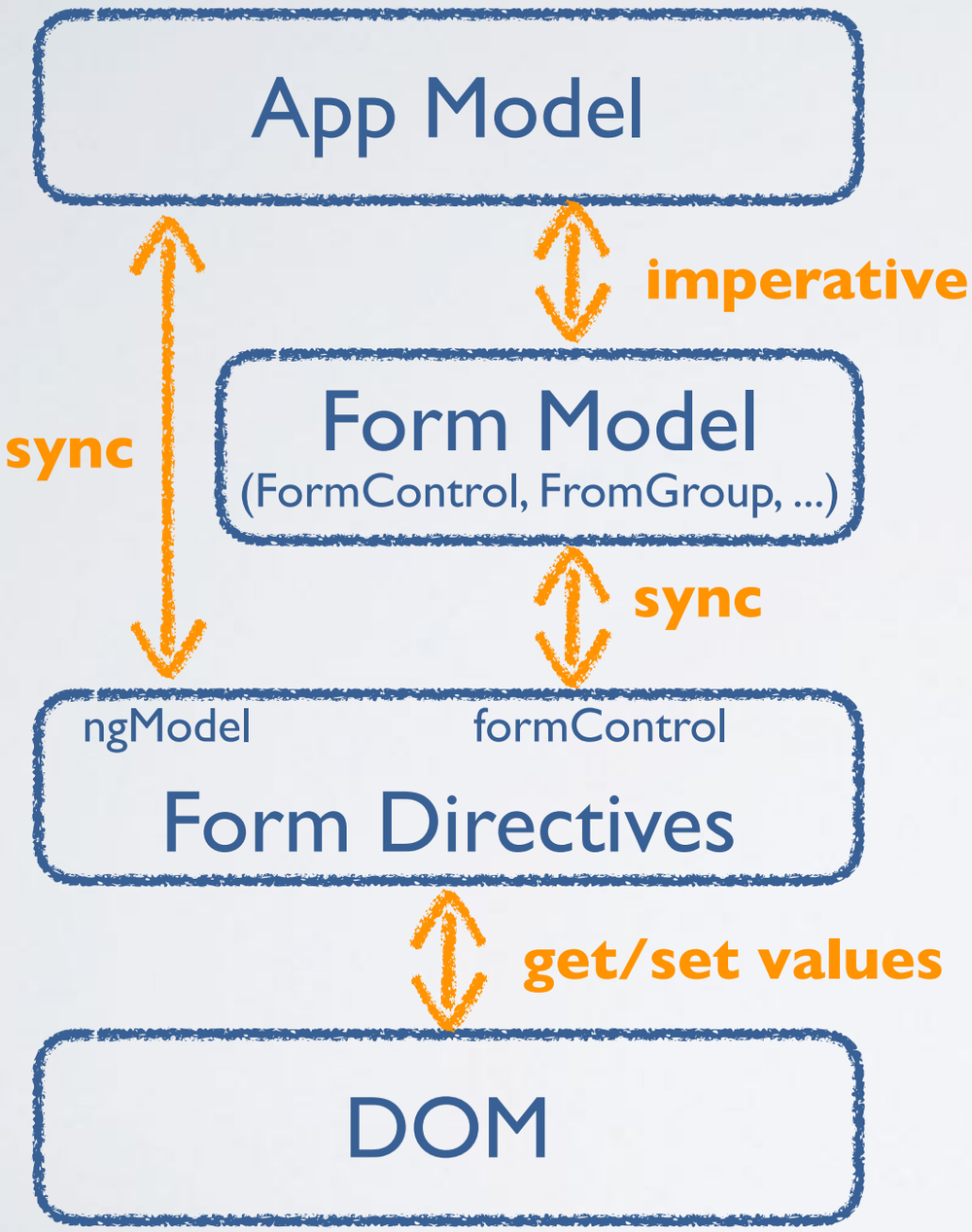
- Angular Forms
- UI Constructs Part 2
- Backend Access
- Modularization & Routing Part 2
- State Management

SIDE-TRACK

- Modern JavaScript Development
- JavaScript: ES5 & ES2015+
- TypeScript

Angular Forms

Angular Forms



application object
provided by the app developer

ui independent representation of the form
building blocks provided by Angular

directives connecting the form model to the DOM
provided by Angular

representation of the UI in the browser

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`, `ngSubmit` …

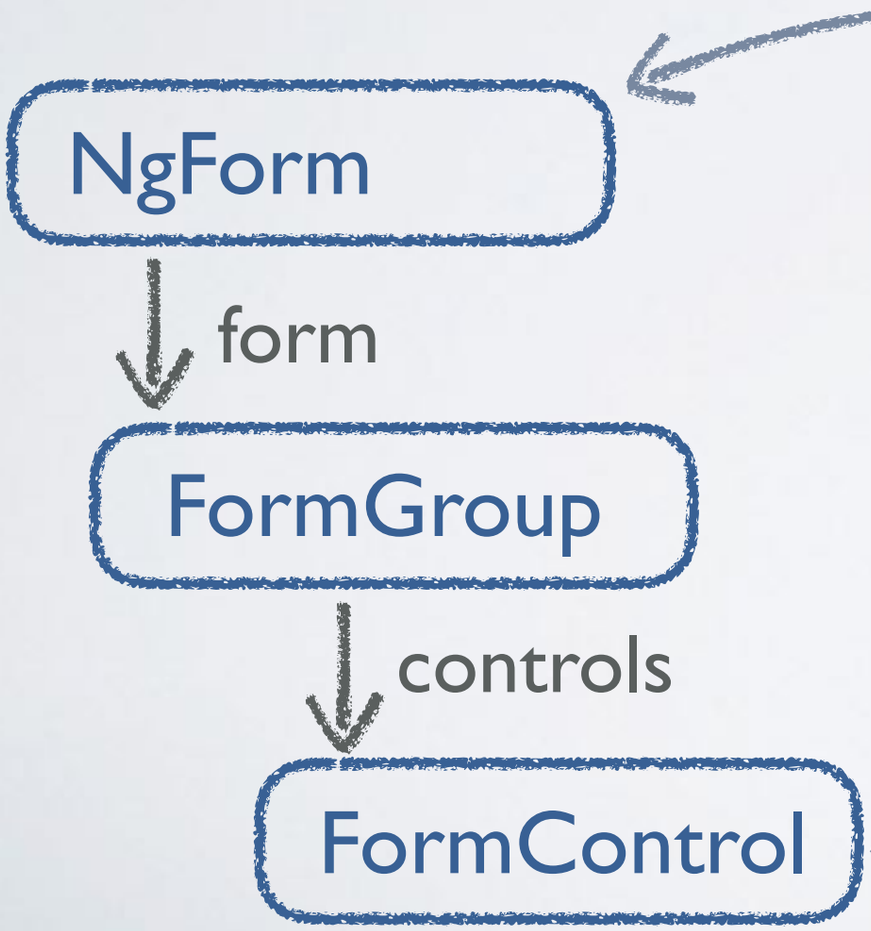
```
<form #formRef="ngForm" (ngSubmit)="onSubmit(formRef.value)">

    <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.

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>
```

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(theForm.value)">

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

```
@Component({
    ...
})
export class Form3Component {
    private theForm: FormGroup;

    constructor(formBuilder: FormBuilder){
        this.theForm = formBuilder.group({
            'name': ['']
        });
    }
    ...

}
```

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

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

```
constructor(formBuilder: FormBuilder){
    this.myForm = formBuilder.group({
        'name': [''],
    });
}
```

FormGroup

controls

FormControl

```
<form [formGroup]="myForm">
  <input name="username"
    type="text" required
    formControlName="name"
</form>
```

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/

EXERCISES

Exercise 5 – Forms

Advanced 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(el: ElementRef) {
       el.nativeElement.style.backgroundColor = 'yellow';
    }
}
```

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>
```

https://angular.io/docs/ts/latest/api/#!?apiFilter=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/docs/ts/latest/guide/pipes.html

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() myChild;
  doReset(){ myChild.reset() }
}
```

Nested Content

(aka "Transclusion")

A component can have nested content:

parent-template.html

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

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

container.html

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


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

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

Access to components in content in parent 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-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.

Backend Access

Backend Access

Angular provides the **`HttpClient`** service for backend access:

```
this.httpClient
    .get(backendUrl)
    .subscribe(
              data => this.data = data,
              error => this.error = error
        );
```

Use the **`async`** pipe to bind the latest value of an observable or a promise:

```
this.data = this.httpClient
                  .get(backendUrl);
```

```
<pre>{{data | async | json}}</pre>
```

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

The observables of **`http`** *do* complete, therefore you don't need to unsubscribe.

Fallback to promises:

```
this._http
    .get(backendUrl)
    .toPromise()
    .then(...)
    .catch(...)
```

There is also a Jsonp service.

https://angular.io/docs/ts/latest/guide/server-communication.html

Including RxJS

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 uses RxJS 5.5 which introduces "lettable operators":

```
import { range } from 'rxjs/observable/range';
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 you use them in.

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

EXERCISES

Exercise 5 – Backend Access

Modularization

Modularization

NgModules can depend on other NgModules.

There is always *one root* module.

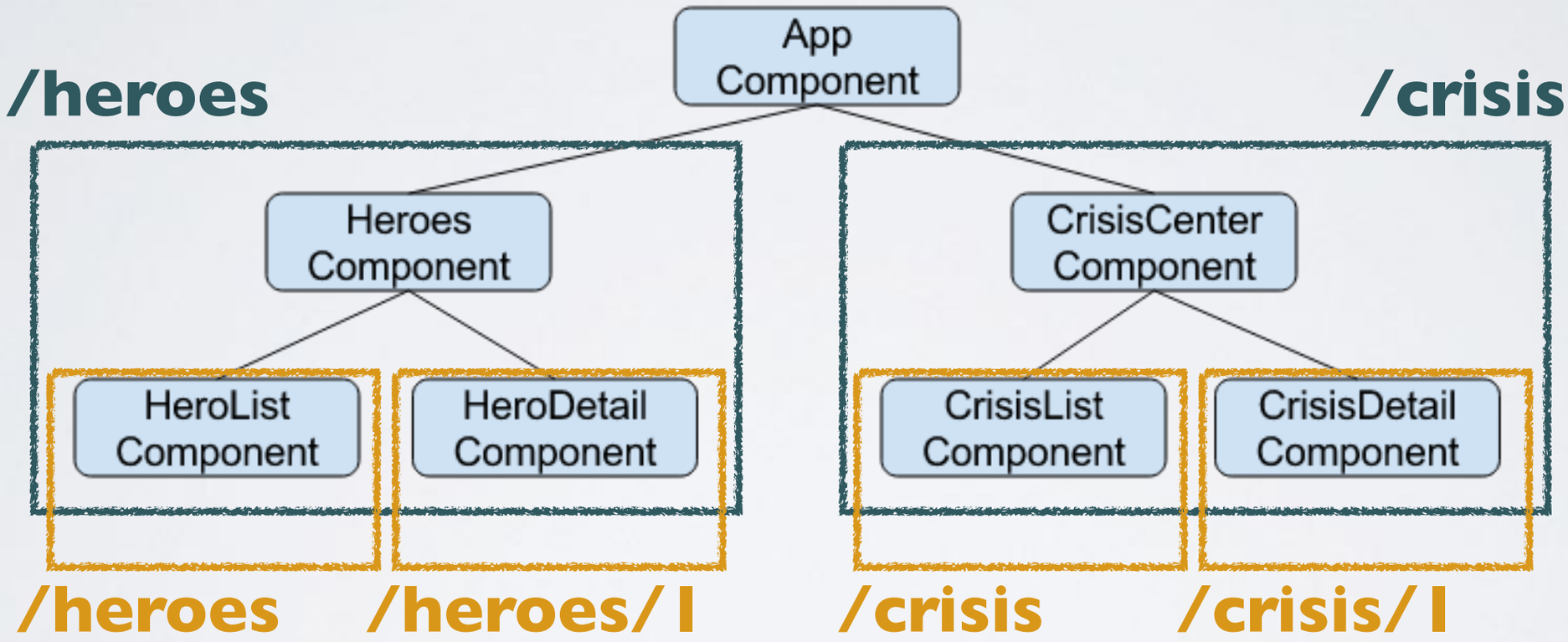
```
@NgModule({
    imports: [BrowserModule,
              FeatureModule],
    declarations: [AppComponent],
    bootstrap: [AppComponent]
})
export class AppModule {}
```

Exported components can be used in the importing module.

```
@NgModule({
    imports: [CommonModule],
    declarations: [ChildComponent],
    exports: [ChildComponent]
  })
export class FeatureModule {}
```

Hierarchical Routing

In a component based approach, each view is implemented by one or more components.

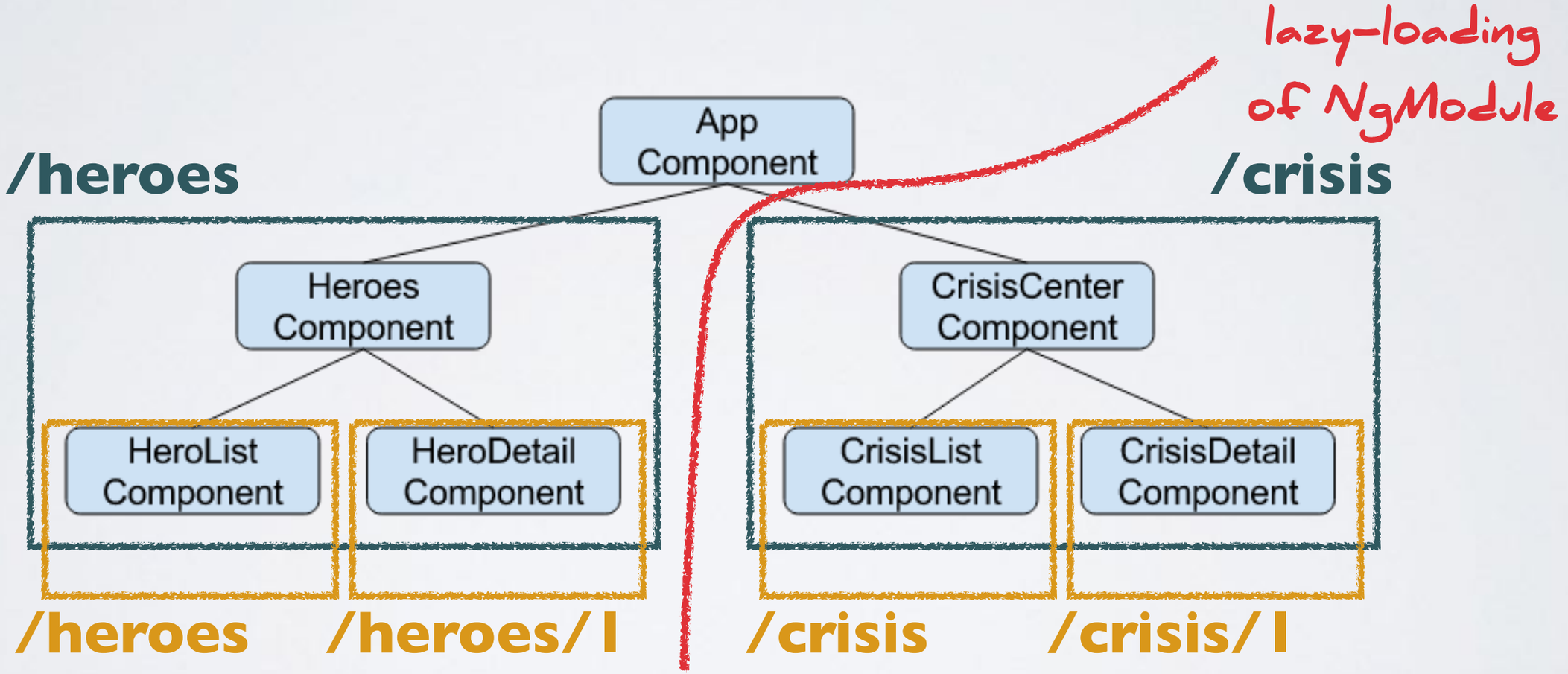


Routing is hierarchical: a parent component can associate a url with a child component.

Example: 02-basic-constructs -> 02-forms

Lazy Loading

A route can point to a NgModule which can be lazy-loaded when needed.



Example: 02-basic-constructs -> 03-backend-access

EXERCISES

Exercise 7 – Modularization

Routing

Route Params

Route definitions can contain rout parameters:

```
{ path: 'detail/:id', component: TodoDetailsComponent},
```

Route parameters can be accessed via the `ActivatedRoute` service:

```
constructor(private route: ActivatedRoute){};
```

The ActivatedRoute exposes many properties (url, paramMap, queryParamMap …).

```
const myId = this.route.snapshot.paramMap.get('id');
```

Snapshot contains the initial values at the time the component is created.

```
this.route.paramMap.subscribe(paramMap => this.myId = paramMap.get('id'));
```

The observable paramMap pushes new values, when the url changes.

https://angular.io/guide/router#activatedroute-the-one-stop-shop-for-route-information

Programmatic Navigation

Programmatic navigation happens via the **`Router`** service:

```
constructor(private router: Router){};
```

Navigate with a *link parameters array*:

```
this.router.navigate(['/details', 5]);
```

```
this.router.navigate(['/details', {id:5}]);
```

…or via Url:

```
this.router.navigateByUrl('/details/5');
```

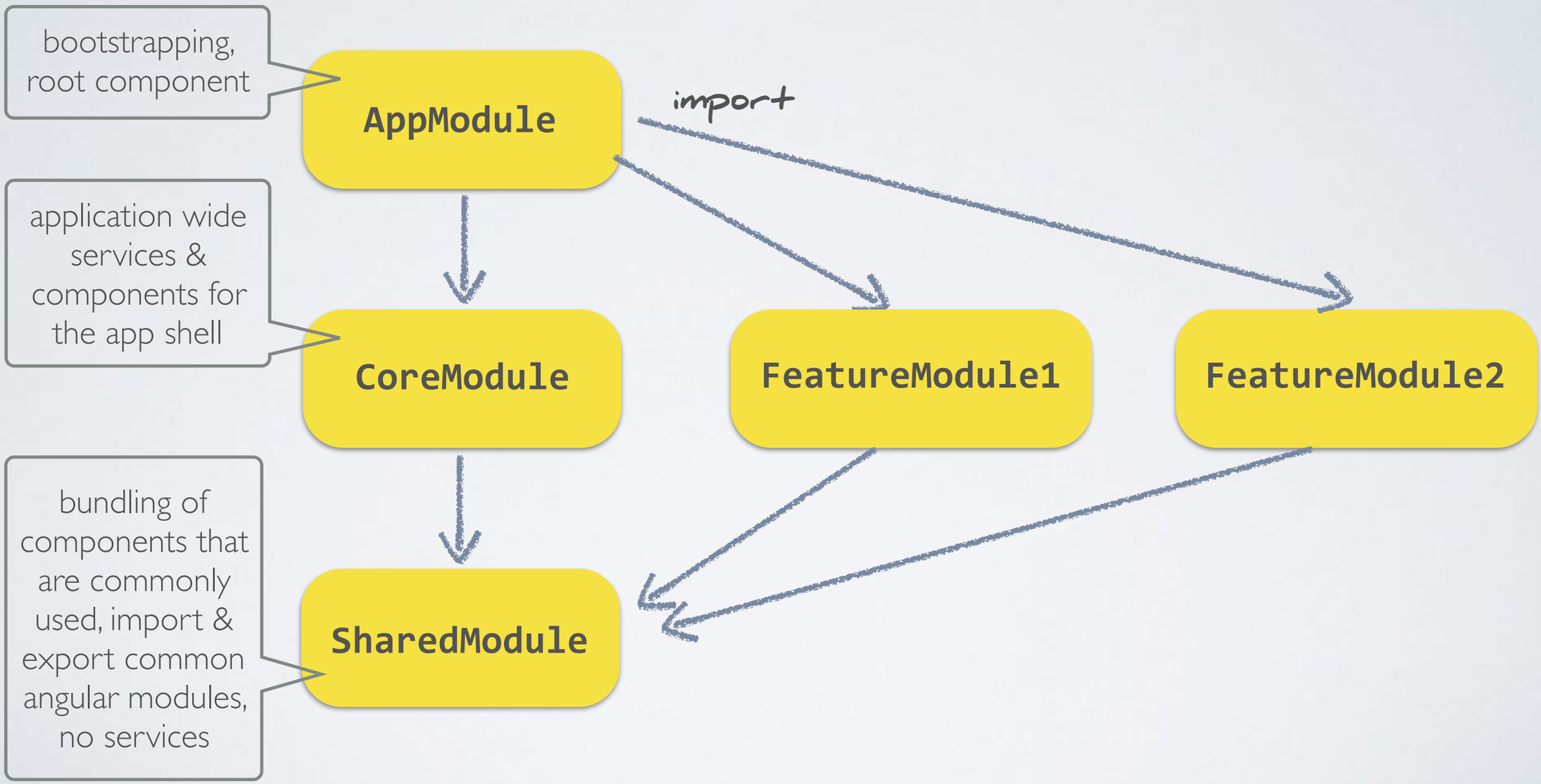
https://angular.io/guide/router#navigating-back-to-the-list-component

EXERCISES

Exercise 8 – Routing

Structuring an App into Modules

feature modules, shared module, core module



https://angular.io/guide/styleguide

https://medium.com/@tomastrajan/6-best-practices-pro-tips-for-angular-cli-better-developer-experience-7b328bc9db81

Advanced State Management

ngrx

MobX

Redux

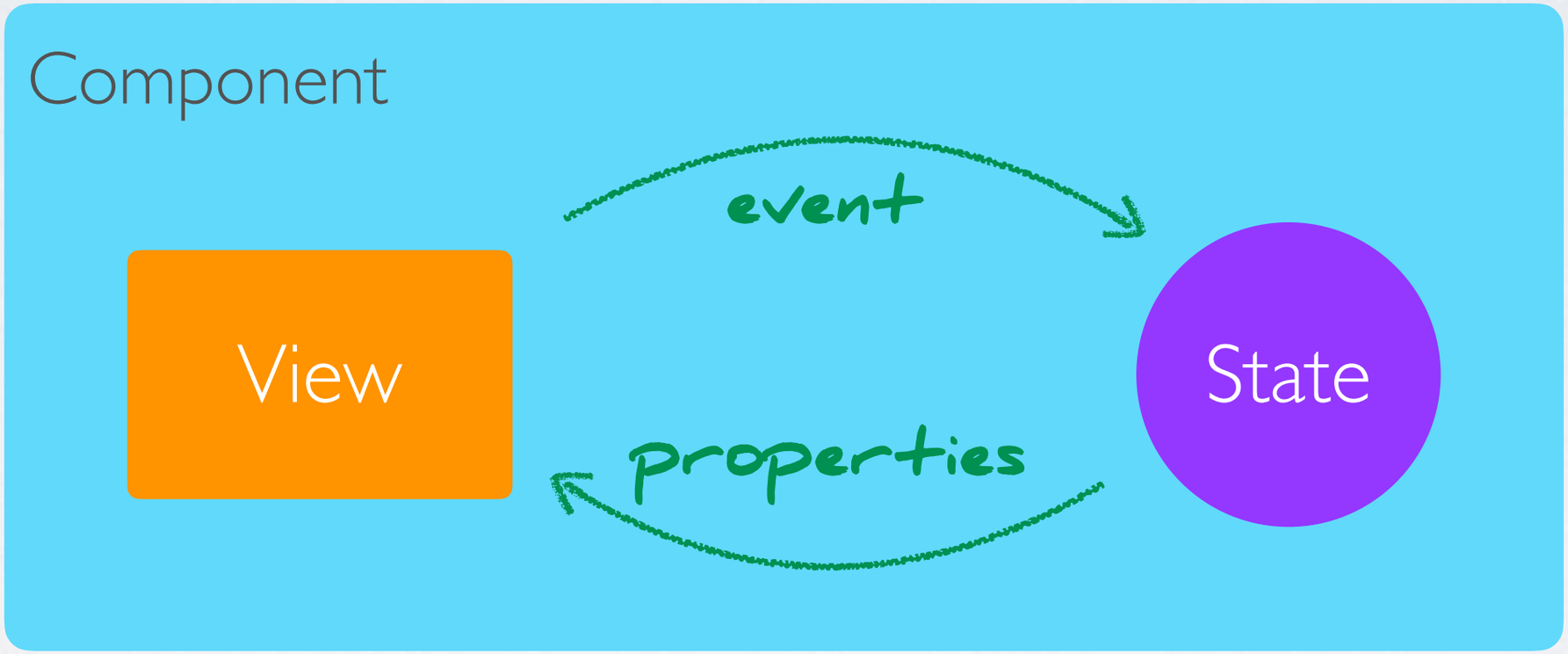
Managing state in a complex web application is not a solved problem! New ideas and implementations are popping up all the time ...

https://stateofjs.com/2017/state-management/results

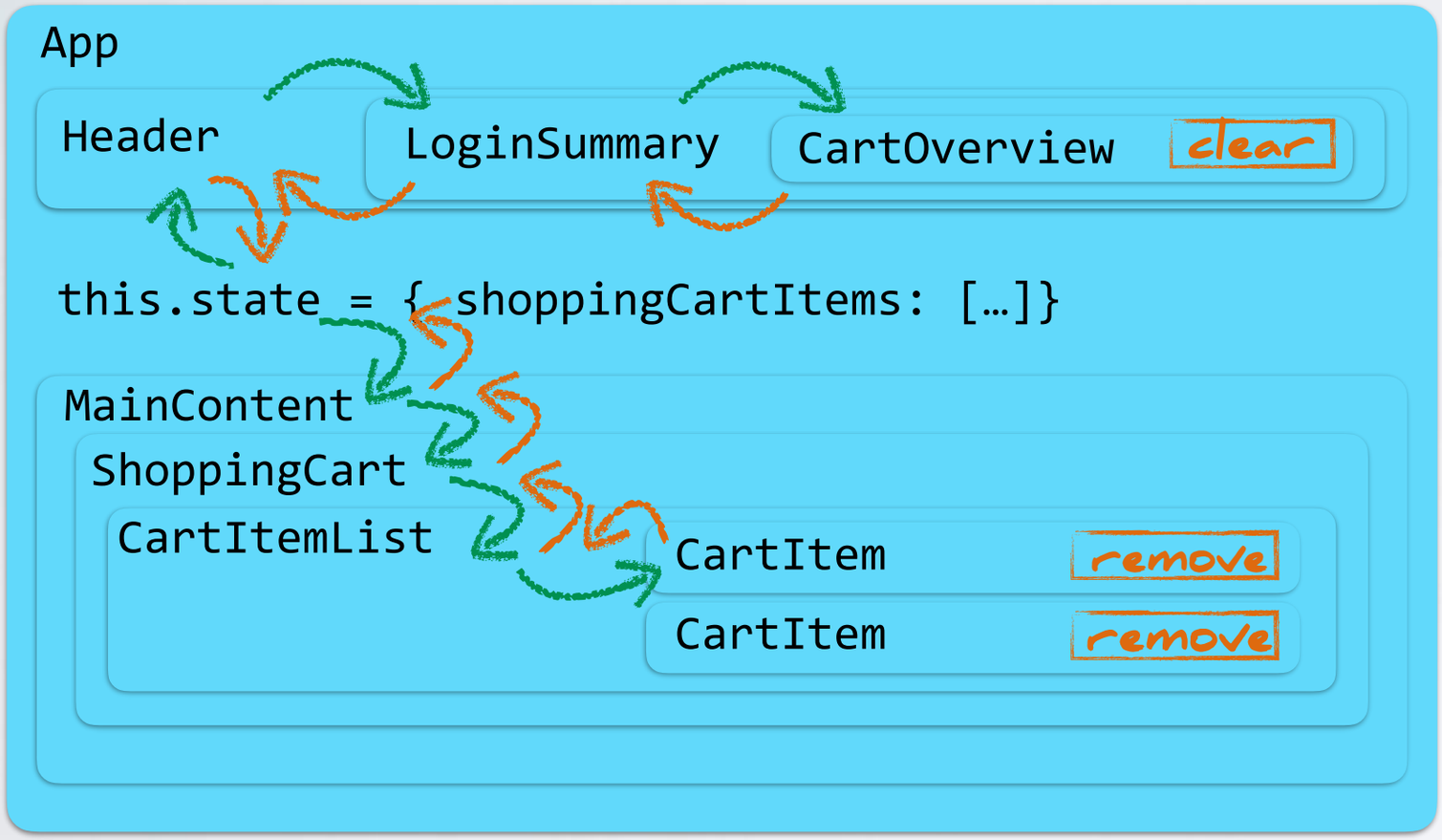
MobX State Tree: https://github.com/mobxjs/mobx-state-tree
Immer: https://github.com/mweststrate/immer
Satchel: https://github.com/Microsoft/satcheljs
...

A single component

Managing state in a component is simple:



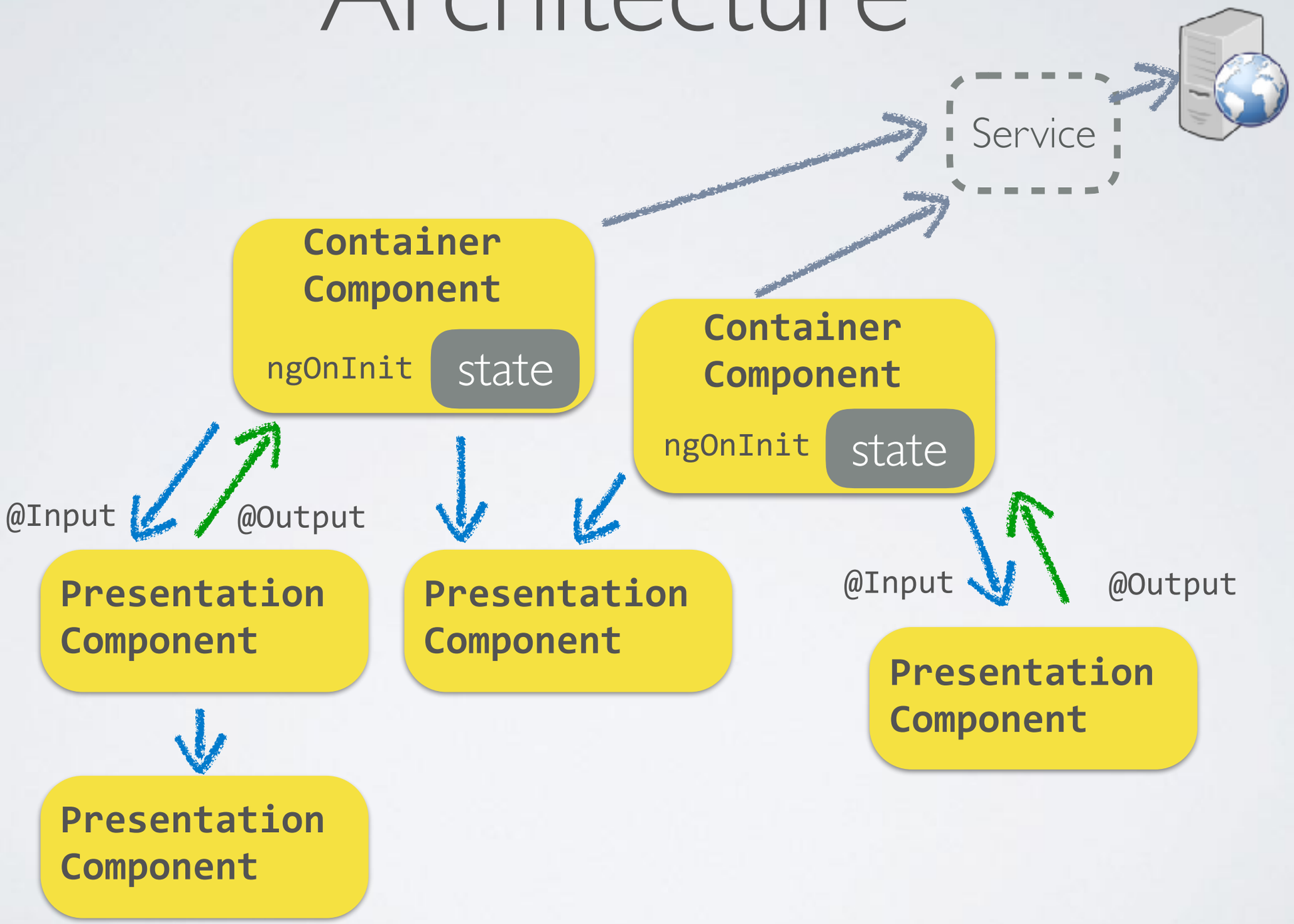
State Management



Challenges when managing state in a component tree:

- Multiple components may depend on the same piece of state.
- Different components may need to mutate the same piece of state.

State Management: Component Architecture



https://www.robinwieruch.de/learn-react-before-using-redux/

Container vs. Presentation Components

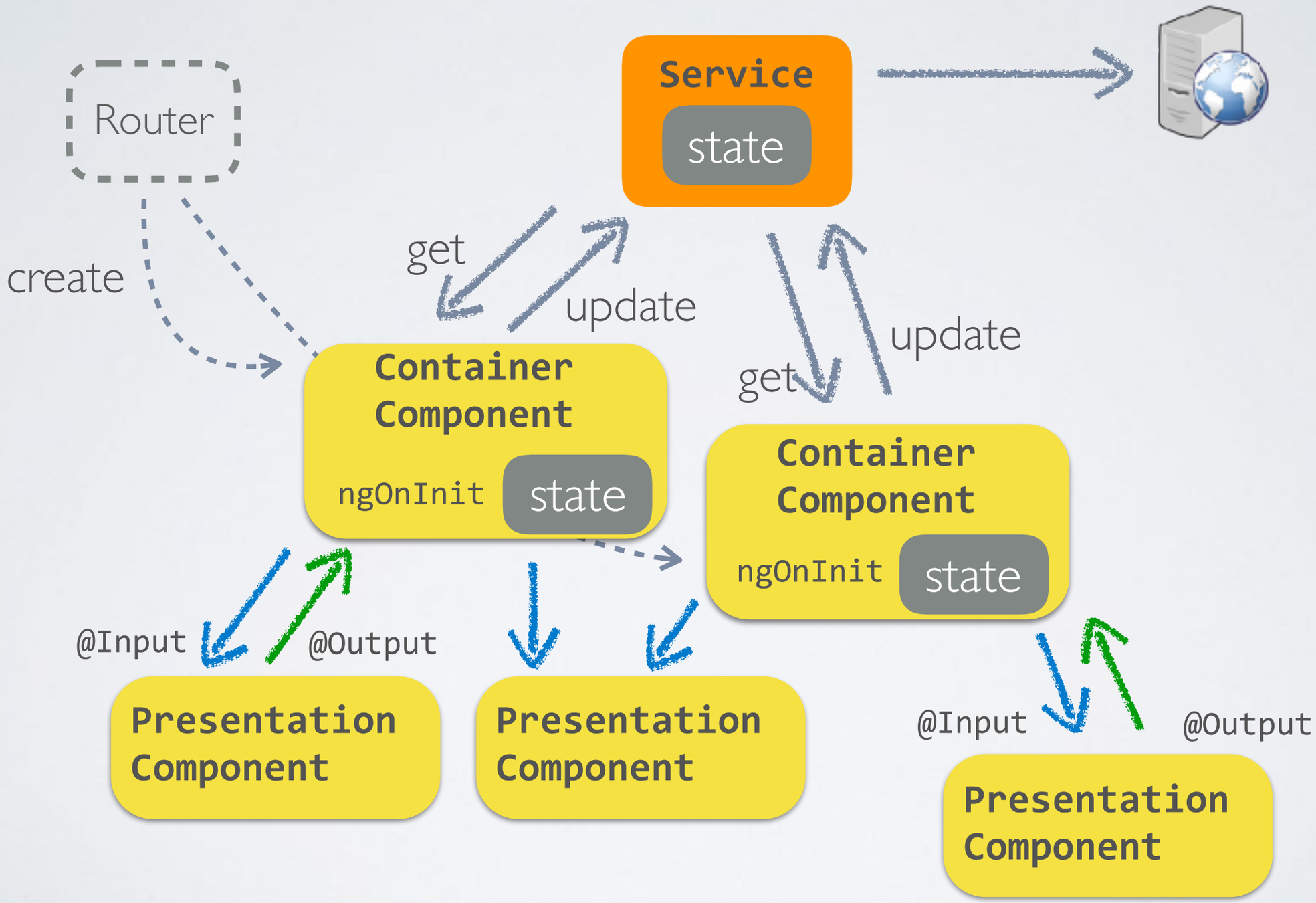
Application should be decomposed in container- and presentation components:

Container	**Presentation**
Little to no markup	Mostly markup
Pass data and actions down	Receive data & actions via props
typically stateful / manage state	mostly stateless
	better reusability

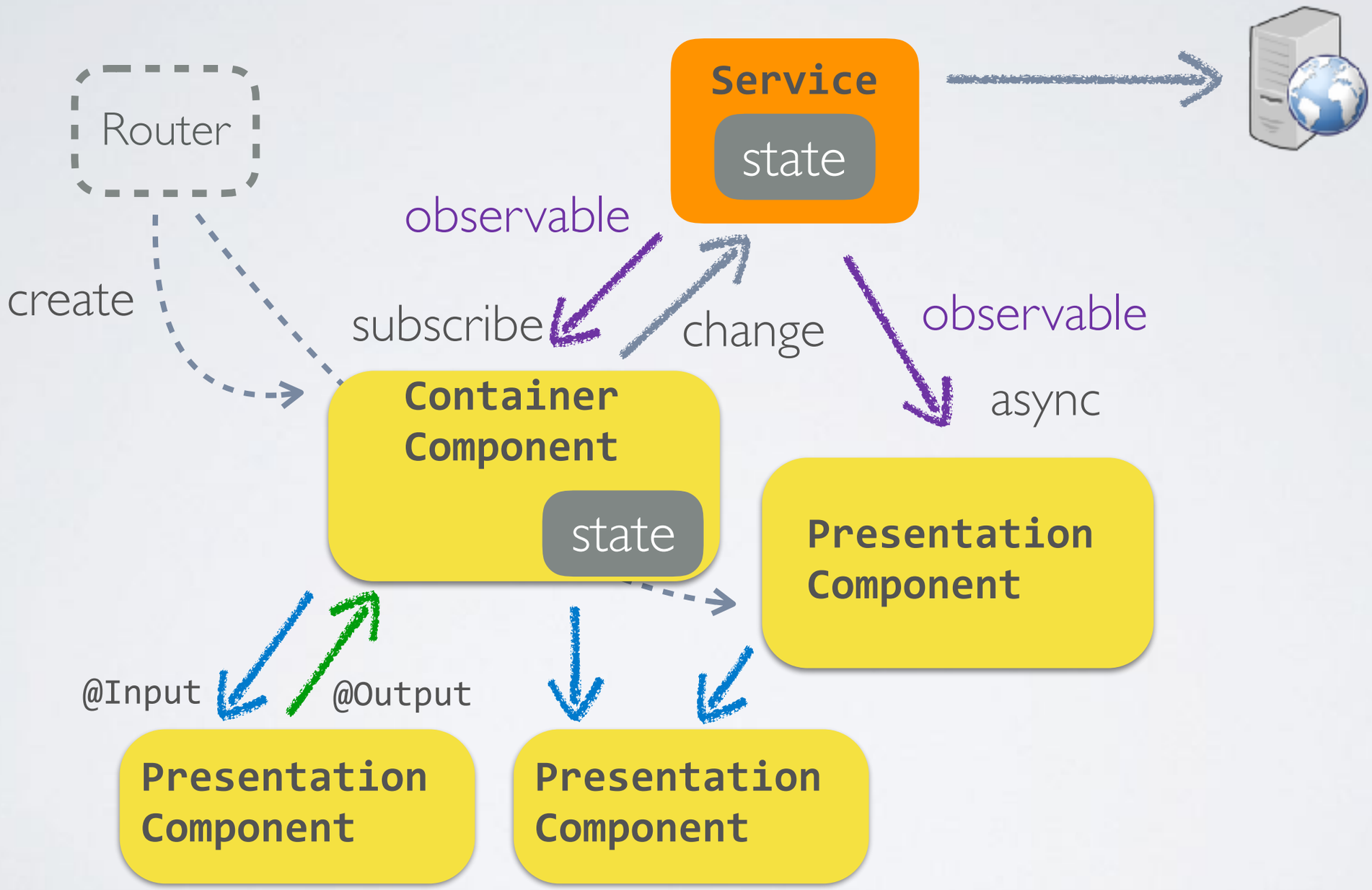
aka: Smart- vs. Dumb Components

https://medium.com/@dan_abramov/smart-and-dumb-components-7ca2f9a7c7d0
https://medium.com/curated-by-versett/building-maintainable-angular-2-applications-5b9ec4b463a1

State Management: Passive Service



State Management: Observable Data Service

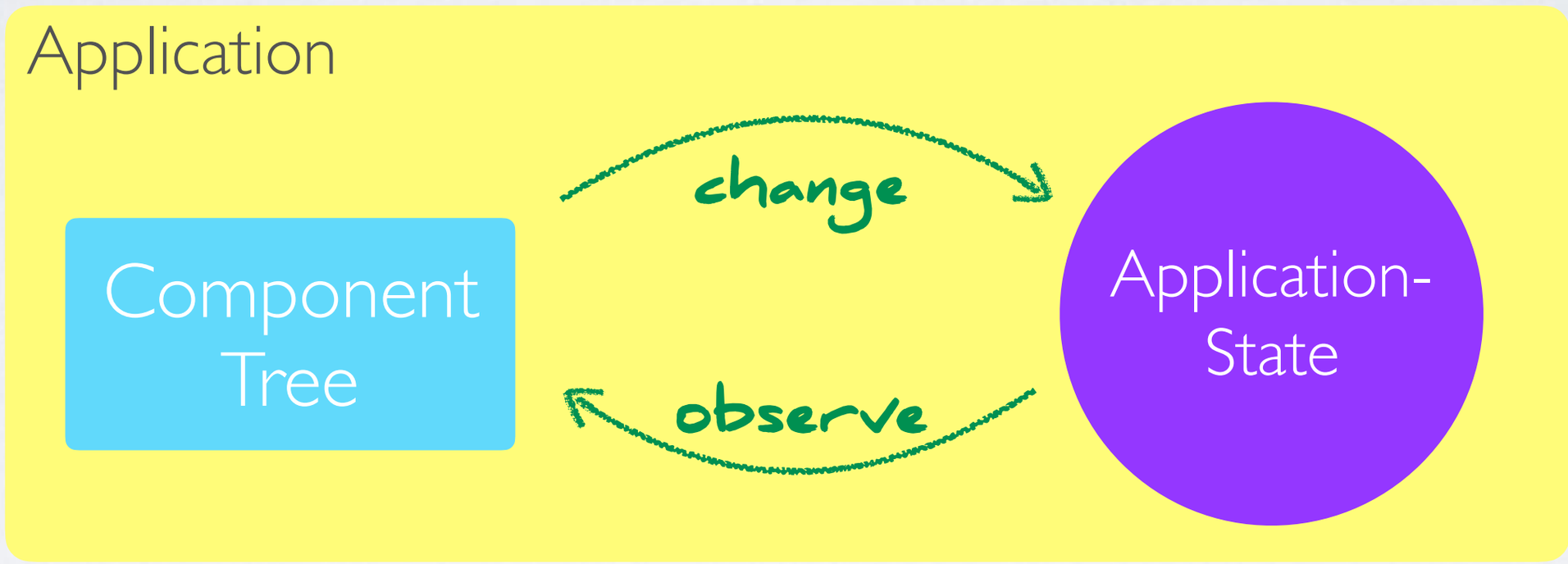


https://coryrylan.com/blog/angular-observable-data-services

https://blog.angular-university.io/how-to-build-angular2-apps-using-rxjs-observable-data-services-pitfalls-to-avoid/

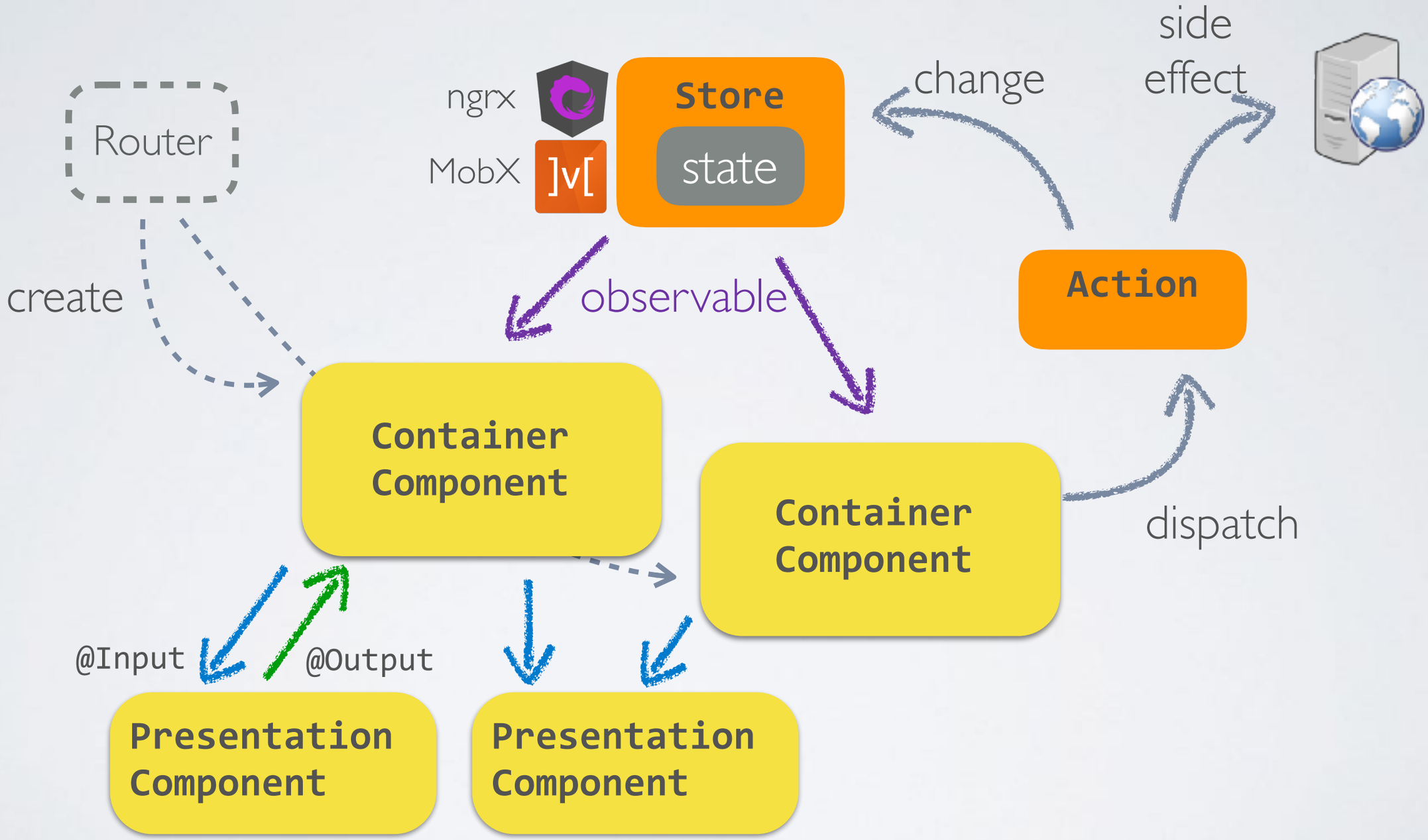
Application with State Container

A state container extracts the shared state out of the components, and manages it in a global singleton.



The component tree becomes a big "view", and any component can access the state or trigger actions, no matter where they are in the tree!

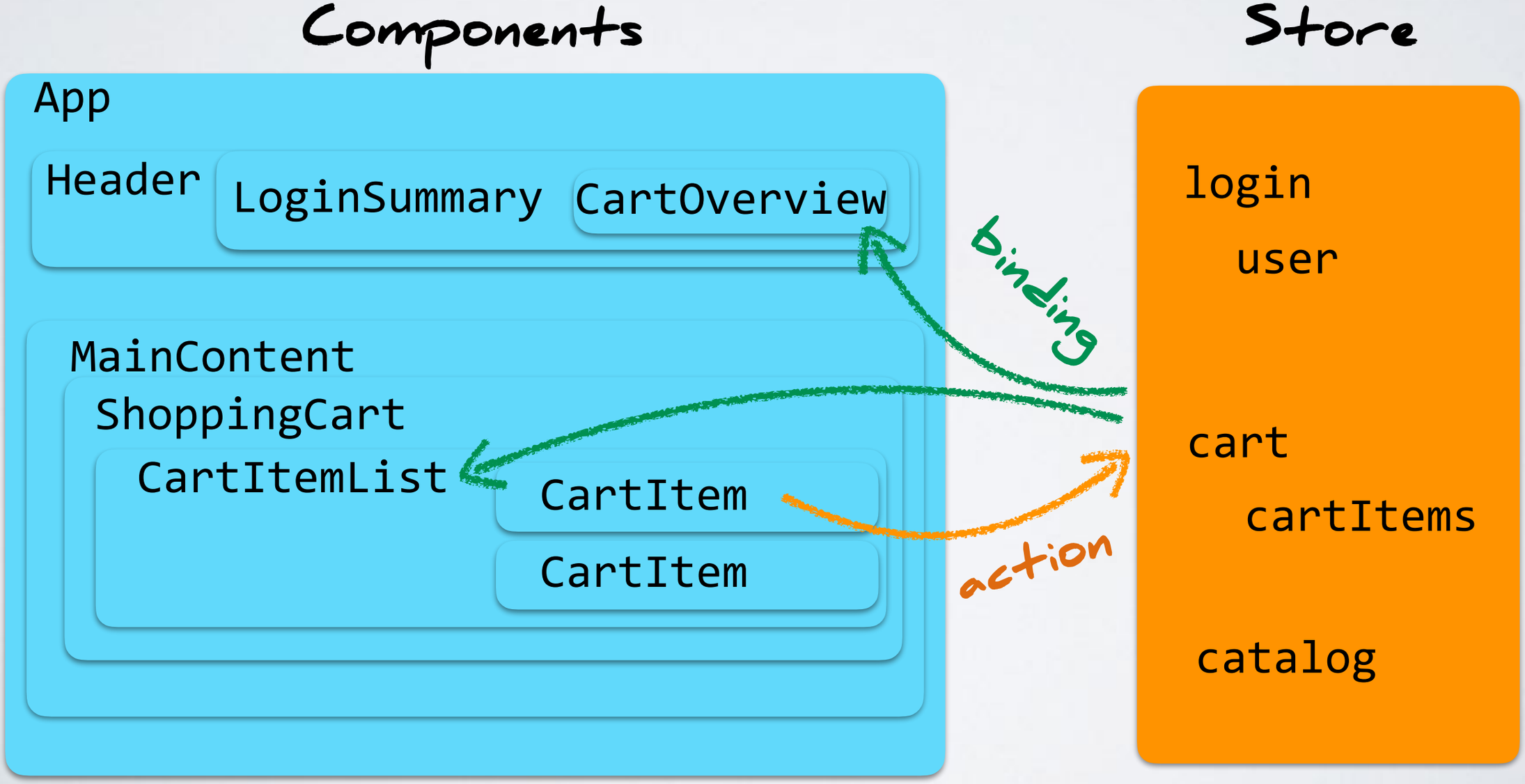
State Management: State Container



ngrx: https://github.com/ngrx/
MobX: https://mobx.js.org/

Managing State with a State Container

State can be managed outside the components.
Components can be bound to state.



Array Reduce

```
var a = [1,2,3,4,5];

var result = a.reduce(
  // reducer function
  (acc, val) => {
    const sum = acc.sum + val;
    const count = acc.count + 1;
    const avg = sum/count;
    return {sum, count, avg};
  },
  // state object
  {sum:0, count:0, avg:0}
);

console.log('Statistics:', result);
```

The reducer function is a pure function.

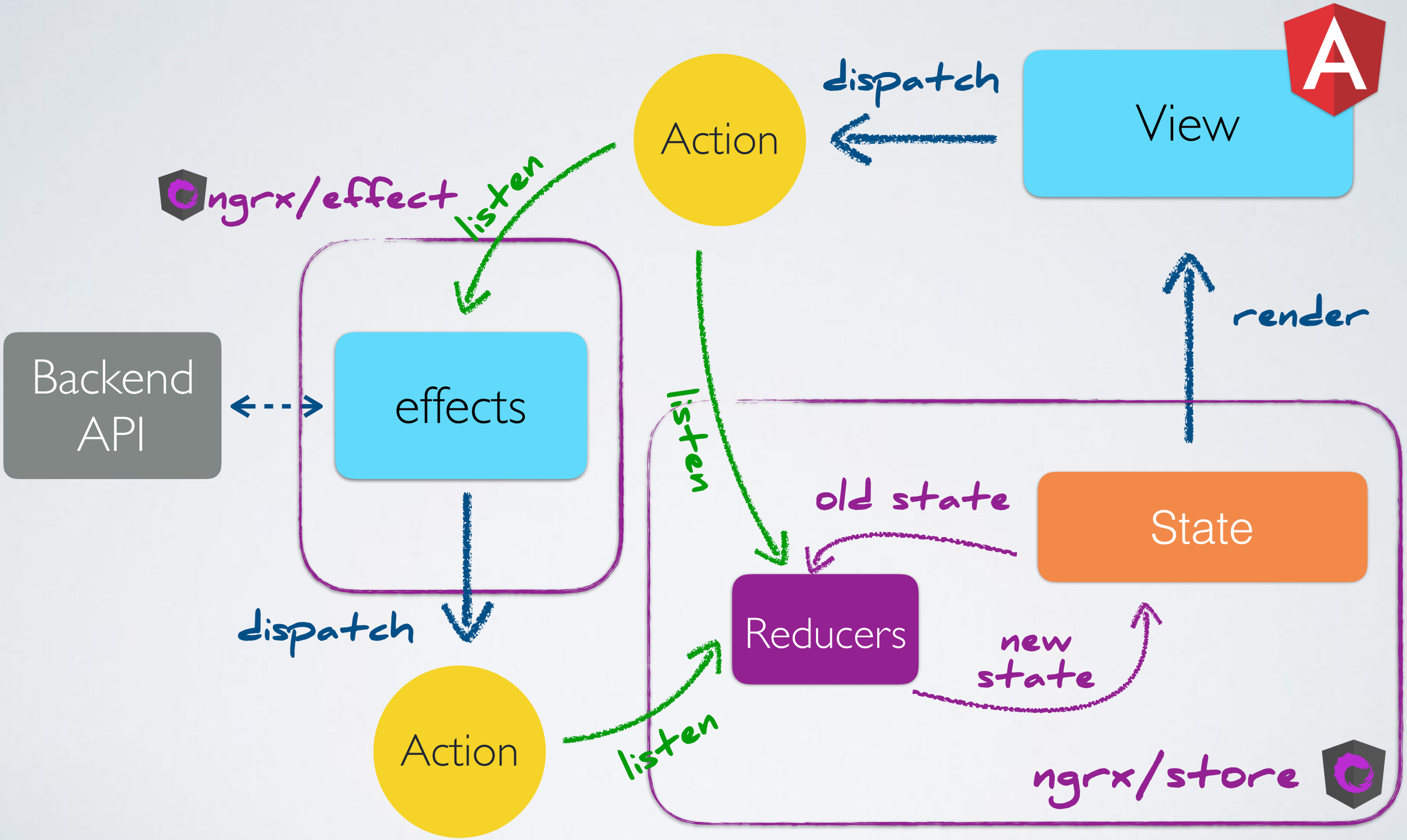
State Reduce

Redux implements store mutations in a "functional way".

(old state , action) => new state

aka: "reducer function"

ngrx Architecture

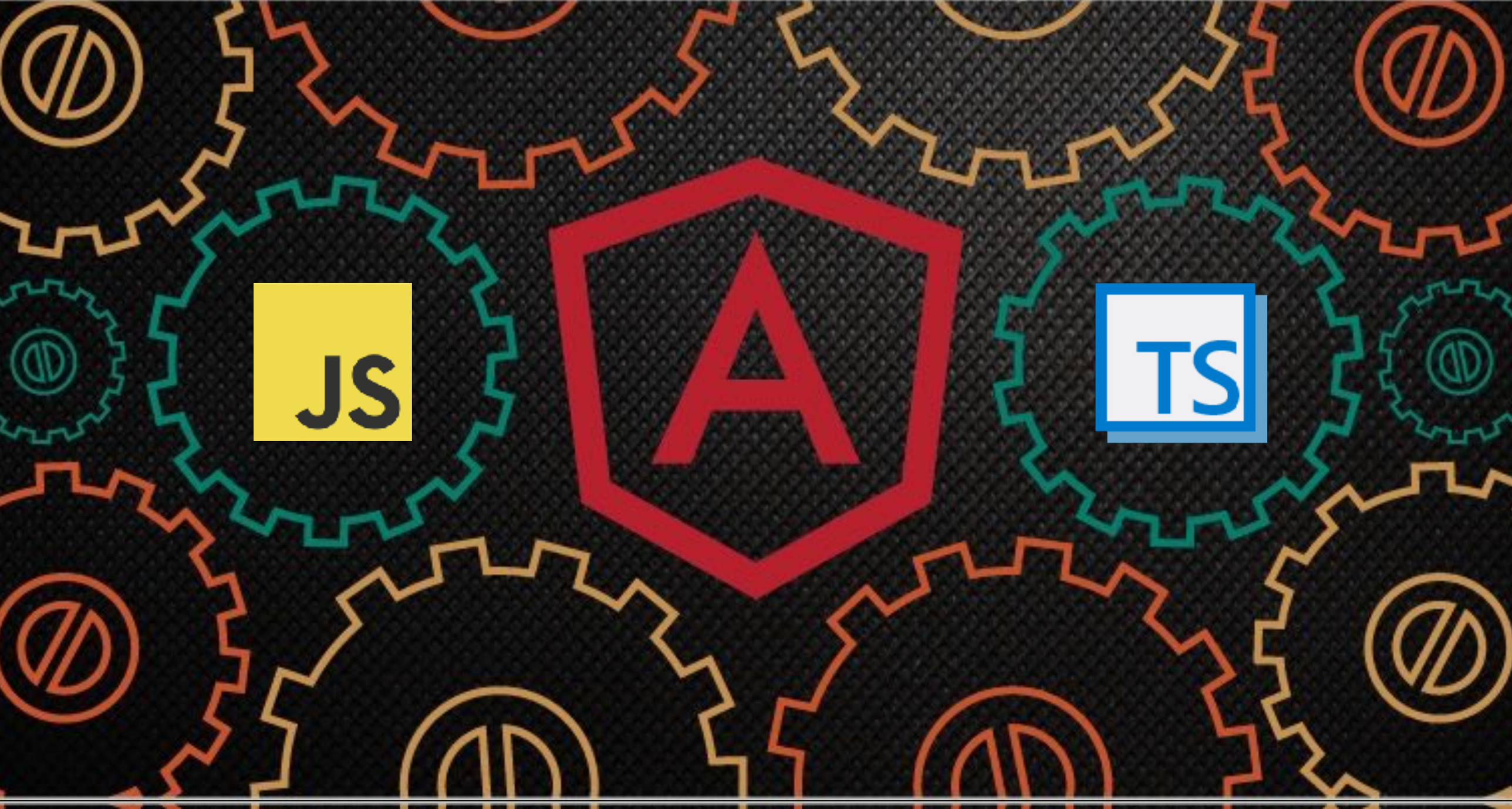


https://github.com/ngrx/platform

MobX

MobX is a state management library that provides a reactive data model and enables mutability.

https://mobx.js.org/
https://github.com/mobxjs/mobx-angular



Have fun with Angular!

Mail: jonas.bandi@gmail.com
Twitter: @jbandi

Alternatives to Angular

No longer a De-Facto Standard

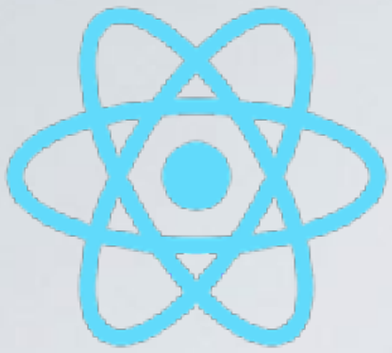
From about 2012 to 2014, AngularJS was almost a de-facto standard for the enterprise.

Today there are viable alternatives.

ThoughtWorks Technology Radar 2017:
"Some of our teams now consider it a solid choice.
Most of our teams, however, still
prefer React, Vue or Ember over Angular."

https://www.thoughtworks.com/radar/languages-and-frameworks

https://medium.com/@chriscordle/why-angular-2-4-is-too-little-too-late-ea86d7fa0bae
https://www.quora.com/Has-Angular-started-to-slowly-die/answer/Michael-Soileau?srid=33NW



React: https://facebook.github.io/react/

- Very popular / big ecosystem
- Project setup has similar complexity as with Angular
- Interesting synergies: ReactNative, React for Windows …

Vue.js: https://vuejs.org

- Very popular / big ecosystem
- Project setup can be much simpler (ES5 style)
- "Powerful enough" for many projects

Aurelia: http://aurelia.io/

- Very "Standard Compliant" and unobtrusive
- Clean separation of concerns
- Commertial Backing / Support

Others:

- AngularJS is still an option! - https://www.ng-book.com/modern-ng1/
- Ember: http://emberjs.com/
- Polymer: https://www.polymer-project.org/

http://todomvc.com/

Vielen Dank!

JavaScript / Angular / React
Schulungen & Coachings,
Project-Setup & Proof-of-Concept:

http://ivorycode.com/#schulung
jonas.bandi@ivorycode.com