UPHILL CONF

Frontend-Development with Angular

Mail: jonas.bandi@ivorycode.com

Twitter: @jbandi

ABOUT ME

Jonas Bandi
jonas.bandi@ivorycode.com
Twitter: @jbandi

- Freelancer, in den letzten 5 Jahren vor allem in Projekten im Spannungsfeld zwischen modernen Webentwicklung und traditionellen Geschäftsanwendungen.
- Dozent an der Berner Fachhochschule seit 2007
- In-House Kurse: Web-Technologien im Enterprise
  UBS, Postfinance, Mobiliar, BIT, SBB ...

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

http://ivorycode.com/#schulung

AGENDA

- **Intro**
- **Angular CLI**
- **Components Basics**
- **Component Architecture**
- **Backend Access**
- **NgModules**
  Change Detection / State Management

Material

Git Repository:
`https://github.com/jbandi/ng-uphill-2019.git`

Initial Checkout:
`git clone https://github.com/jbandi/ng-uphill-2019.git`

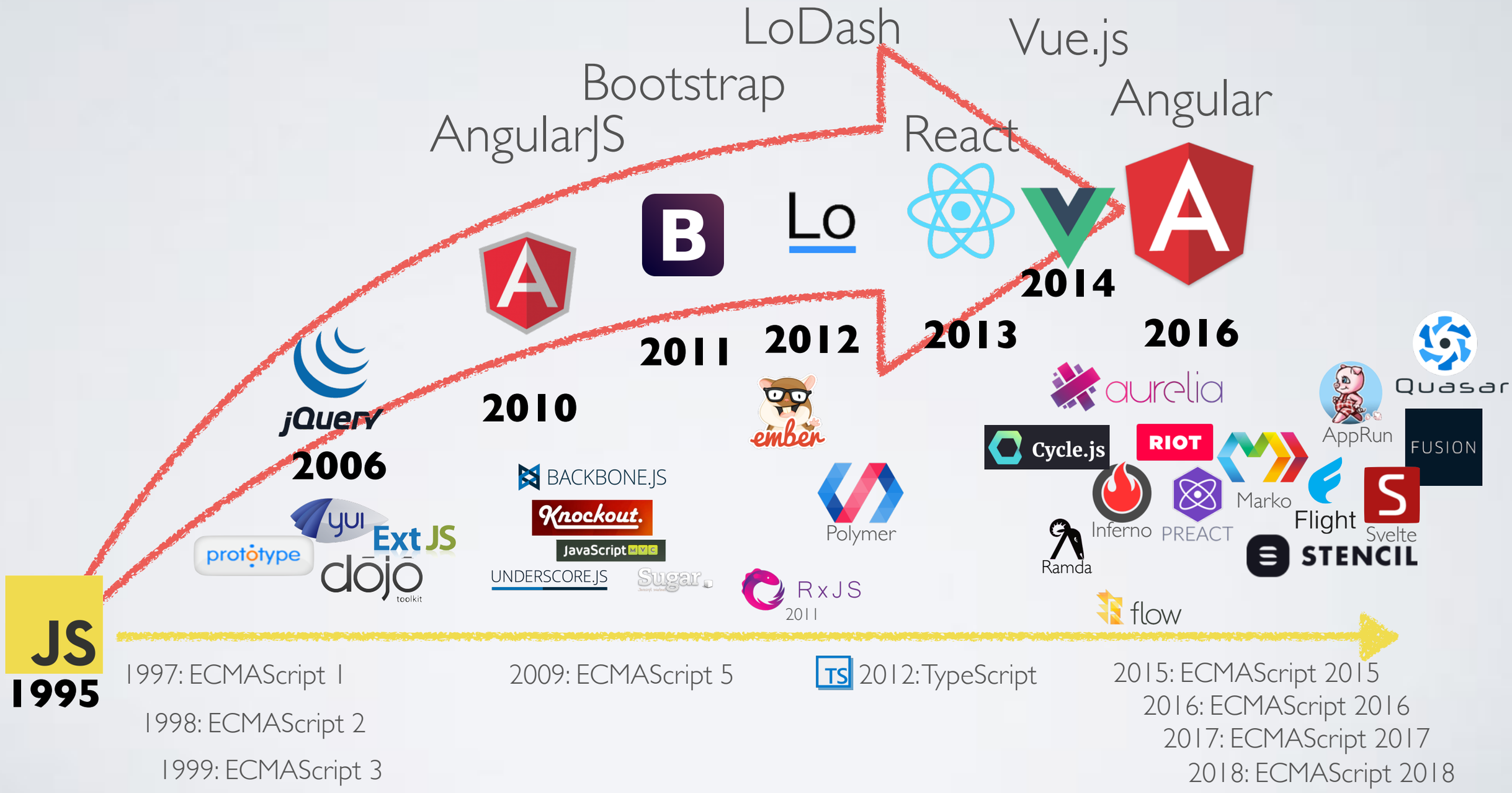
Update: `git pull`

```
git reset --hard
git clean -dfx
git pull
```

(discard all local changes)

Slides & Exercises: `<checkout>/00-CourseMaterial`

The Frontend JavaScript Ecosystem



Angular is a new implementation of the concepts behind AngularJS ...

... for the modern web.

... but Angular is not an update to AngularJS.

Angular is **built upon** the modern web:

- shadow dom
- web workers

Angular is **built for** the modern web:

- mobile browsers
- modern browsers
- server-side rendering

Angular **improves** over AngularJS:

- faster
- easier to use & learn
- built on proven best practices (i.e. components, unidirectional data flow …)

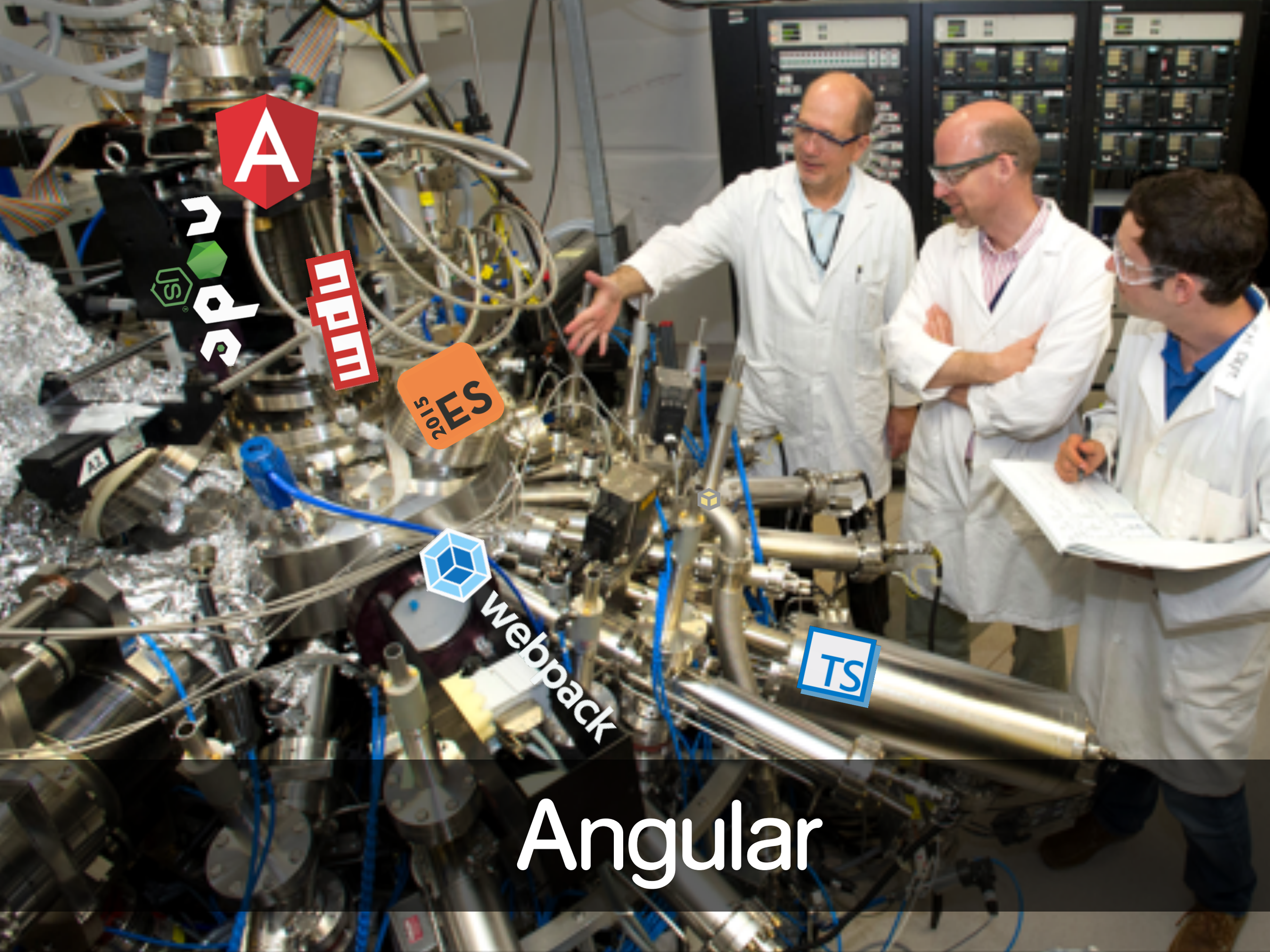
Angular JS

AngularJS

```
(function () {
    'use strict';

    var app = angular.module('todoApp');
    app.controller('todoController', ToDoController);

    ToDoController.$inject = ['toDoService'];
    function ToDoController(todoService) {
        var ctrl = this;

        ctrl.newToDo = new ToDoItem();
        ctrl.todos = todoService.getToDos();

        ctrl.addToDo = function () {
            ctrl.newToDo.created = new Date();
            todoService.addToDo(ctrl.newToDo);
            ctrl.newToDo = new ToDoItem();
        };

        ctrl.removeToDo = function (todo) {
            todoService.removeToDo(todo);
        };
    }
})();
```

Angular

```
import {Component,
  EventEmitter, Output} from '@angular/core';
import {ToDo} from '../../model/todo.model';

@Component({
  selector: 'td-new-todo',
  templateUrl: './new-todo.component.html',
})
export class NewTodoComponent {

  public newToDo: ToDo;

  constructor() {
    this.newToDo = new ToDo();
  }

  @Output() addToDo = new EventEmitter<ToDo>();

  onAddToDo(): void {
    this.addToDo.emit(this.newToDo);
    this.newToDo = new ToDo();
  }
}
```



Angular

Angular aspires to be a Platform

"One framework. Mobile & desktop"

classic web-apps for desktops

WebPlatform.org

ANGULAR MOBILE

progressive web-apps for mobile
(web workers, cache, push, offline)

installed mobile apps
(hybrid - web-ui)

installed mobile apps
(hybrid - native UI)

ANGULAR UNIVERSAL

server side rendering
https://angular.io/guide/universal

dev tooling

ANGULAR CLI

https://github.com/angular/angular-cli

https://angularconsole.com/

installed desktop apps

https://github.com/NathanWalker/angular-seed-advanced

Recent Versions

Angular 5	November 2017	• Mainly performance improvements. • RxJS 5.5. • New features in Forms.
Angular 6	May 2018	• Alignment of project version numbers: Angular, CLI, Material, CDK • Dependency upgrades: RxJS 6, WebPack 4 • CLI: new structure, update & add, support for libraries • Tree Shakable Services, • Angular Elements (initial support)
Angular 7	October 2018	• No relevant changes in the Angular framework. • Updates in Angular CLI & Angular CDK.
Angular 8	planned for May 2019	• CLI: Diferential serving • Beta: Ivy - new rendering engine (major performance improvements, better debugability, foundation for new features) • new syntax for lazy loading modules • ???

What is Angular?

- Angular is a framework for dynamic web applications (aka. Single Page Applications)
- A framework to dynamically create & manipulate the DOM and linking it to application functionality.
- Angular is a HTML processor
- Angular lets you extend HTML with domain concepts

Getting started quickly with Angular CLI

Execute the following commands in a terminal:

```
npm install -g @angular/cli

ng new awesome-ng --defaults
cd awesome-ng
npm start
```

https://angular.io/cli/

Using `npx` without installing the cli globally:

```
npx @angular/cli@6 new old-app
cd old-app
npm start
```

(`npx` is available since npm v5.2)

EXERCISES

Exercise 1 - Create an Angular App

Dissecting an Angular App

```
npm start
npm run build -- --prod
npm test
npm e2e
npm run lint
```

Project Configuration:

- package.json
- tsconfig.json
- angular.json
- polyfills.ts

App Component:

- /src/app

Bootstrapping:

- main.ts
- index.html
  (src vs. runtime / including es2015-polyfills.js at runtime)

Angular CLI commands: https://angular.io/cli
Angular CLI build options: https://angular.io/cli/build

Angular CLI Build Setup

ES2015+/TypeScript

Write Code

TypeScript

Compile

TypeScript is compiled to ES5

webpack
MODULE BUNDLER

Bundle

Create coarse-grained & optimised bundles from many files and 3rd-party libraries

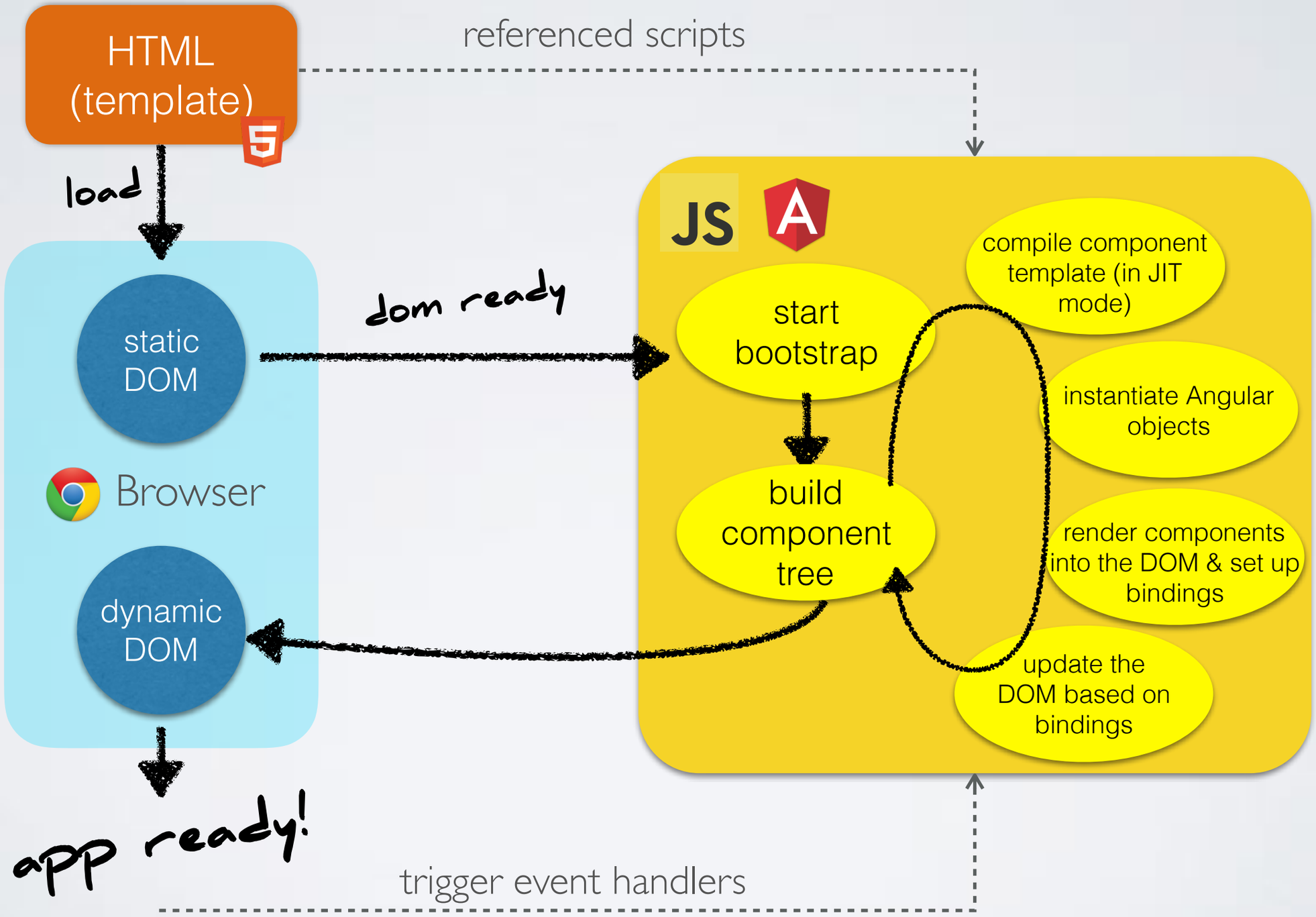
ES5

Run in Browser

Load pure JavaScript bundles.

Modern JavaScript Build: https://medium.com/the-node-js-collection/modern-javascript-explained-for-dinosaurs-f695e9747b70

How Angular Works



How Angular Works

"Angular is a HTML processor"

template

```
<ul class="todo-list" >
    <li *ngFor="let todo of todos">
        {{todo.title}}
        <button class="remove-button"
            (click)="removeToDo(todo)">
            x
        </button>
    </li>
</ul>
```

components
services
...

interact

parse

generate

manipulate

handle events

DOM

Bootstrapping

src/index.html

html shell

```
<body>

  <app-root>
    Loading...
  </app-root>
  <script src=…>
  </script>
```

src/main.ts

bootstrap-script

```
platformBrowserDynamic()
  .bootstrapModule(AppModule)
```

root module

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

src/app/app.module.ts

root component

```
@Component({
    selector: 'app-root',
    template: `Hello world`
})
export class AppComponent {}
```

src/app/app.component.ts

My concerns with the Angular CLI

Setting up an optimized Angular project from scratch is very hard!

I definitely recommend using the Angular CLI!

The CLI is also a risk:
It "hides" the whole toolchain (npm dependencies, webpack ...). If you find out later that the CLI setup does not fit your project, it is very difficult to set up an equivalent toolchain.

Example: https://medium.jonasbandi.net/angular-cli-and-moment-js-a-recipe-for-disaster-and-how-to-fix-it-163a79180173

Documentation of the CLI is sparse. Documentation how to set up an Angular build process without the CLI is even sparser.

Angular CLI used to offer the `ng eject` command to get the config files.
Unfortunately it was disabled in v6 and removed in v7.

ngx-build-plus allows to extend/override the webpack configuration of Angular CLI:
https://github.com/manfredsteyer/ngx-build-plus

Setting up an Angular project without the CLI:
https://medium.freecodecamp.org/how-to-configure-webpack-4-with-angular-7-a-complete-guide-9a23c879f471

Configuring the Angular CLI

Config file: **`angular.json`**

The defaults can be set via **`ng config`**:

```
ng config schematics.@schematics/angular.component.spec false
ng config schematics.@schematics/angular.component.styleext scss

ng config schematics.@schematics/angular.component.inlineTemplate true
ng config schematics.@schematics/angular.component.inlineStyle true
```

(unfortunately no documentation of all the configs, you have too look into the schema of angular.json: https://github.com/angular/angular-cli/wiki/angular-workspace)

CLI: Creating a Production Build

On the commandline using **ng** directly:

```
ng build --prod
```

On the commandline via **npm**:

```
npm run build -- --prod
```

Configuration via **package.json**:

```
"scripts": {
  ...
  "build": "ng build --prod"
}
```

The build output is the **dist** directory. It contains all assets necessary to deploy and run the SPA.

Serving a prod build in development:

```
ng serve --prod
```

```
npx serve -s dist/awesome-ng
```

```
npx gzip-cli dist/awesome-ng/*.js
npx http-server dist/awesome-ng -g
```

Further configuration of the build:
https://angular.io/cli/build

IE Support:
JS Polyfills & CSS Autoprefixer

Angular assumes a modern browser.
Older browsers have to be patched to appear like a modern browser.
The Angular CLI prepares polyfills to make Angular work in IE.

For Angular CLI 7.3+: IE support works out of the box
In `index.html` there is a script tag: `<script src="es2015-polyfills.js" nomodule>`
Configured by setting `"es5BrowserSupport": true` in `angular.json` (architect->build->options)

For Angular CLI < 7.3:
Include the follwing lines in
`src/polyfills.ts`

```
import 'core-js/es6/symbol';
import 'core-js/es6/object';
import 'core-js/es6/function';
import 'core-js/es6/parse-int';
import 'core-js/es6/parse-float';
import 'core-js/es6/number';
import 'core-js/es6/math';
import 'core-js/es6/string';
import 'core-js/es6/date';
import 'core-js/es6/array';
import 'core-js/es6/regexp';
import 'core-js/es6/map';
import 'core-js/es6/weak-map';
import 'core-js/es6/set';
```

For better CSS support, Autoprefixer should be configured to add vendor prefixes for IE:.

Include the follwing lines in
`src/browserslist`

```
> 0.5%
last 2 versions
Firefox ESR
not dead
IE 9-11
```

https://angular.io/guide/browser-support
https://github.com/postcss/autoprefixer

Updating an Angular CLI project

The Angular projects provides detailed instructions
for updating projects between versions:

https://update.angular.io/

The Angular CLI has the **`update`** command to
update an application and it's dependencies:

```
ng update @angular/cli @angular/core
```

https://angular.io/cli/update

Popular Editors

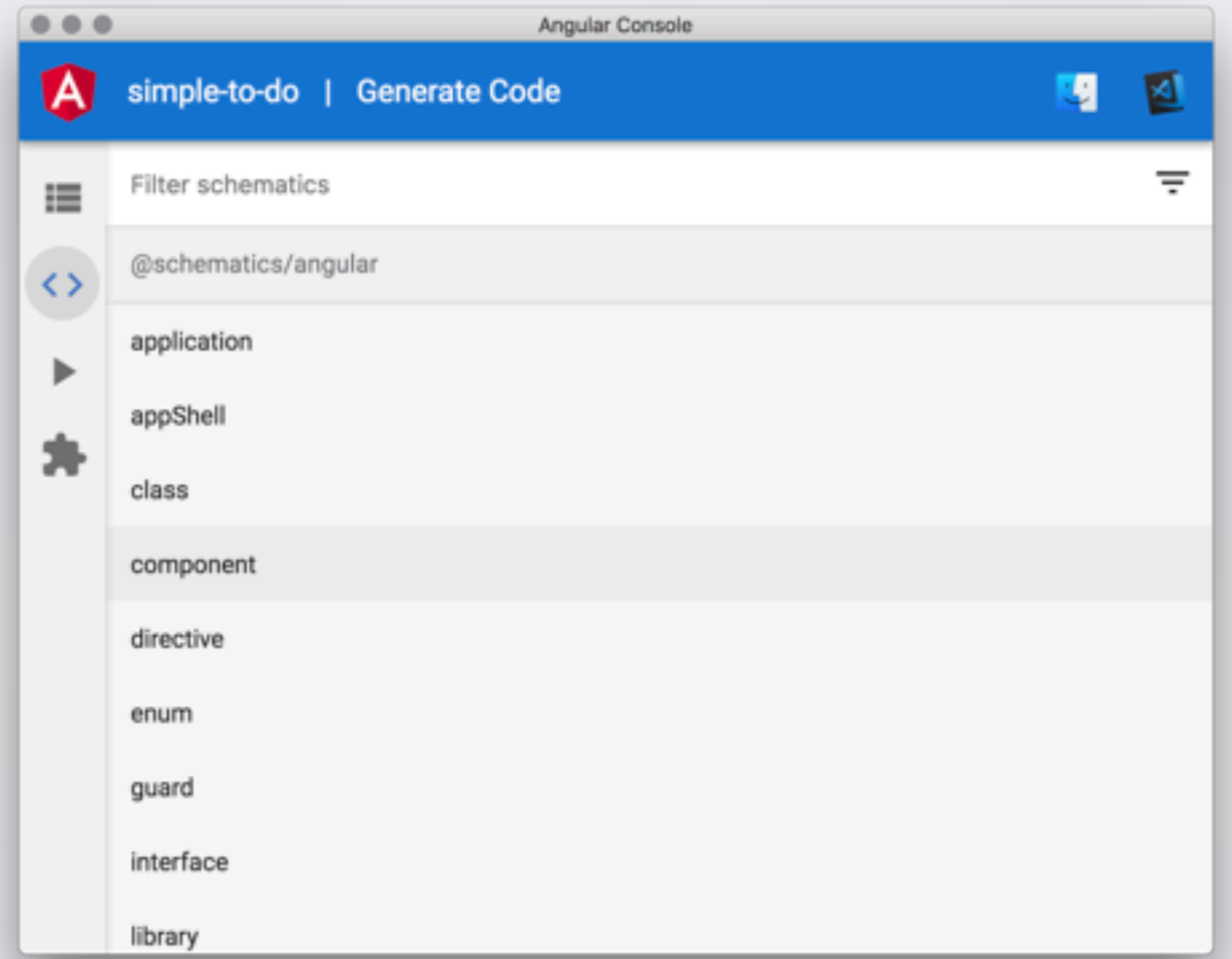
Webstrom & IntelliJ:
Angular 2 TypeScript Live Templates
https://github.com/MrZaYaC/ng2-webstorm-snippets
https://plugins.jetbrains.com/plugin/8395-angular-2-typescript-live-templates

Visual Studio Code:
Angular Essentials
https://github.com/MrZaYaC/ng2-webstorm-snippets

Angular Console

A GUI wrapper around the Angular CLI

https://angularconsole.com/



Debugging

Angular is "just" JavaScript at runtime...

Try the following statements in the console:

```
var helloDomElement = document.getElementsByTagName("app-hello")[0]
var debugElement = ng.probe(helloDomElement)
var helloComponent = debugElement.componentInstance
helloComponent.greetingName = 'Universe'
debugElement.injector.get(ng.coreTokens.ApplicationRef).tick()
```

(the global **ng** is only available in a debug build)

Debugging in Chrome:

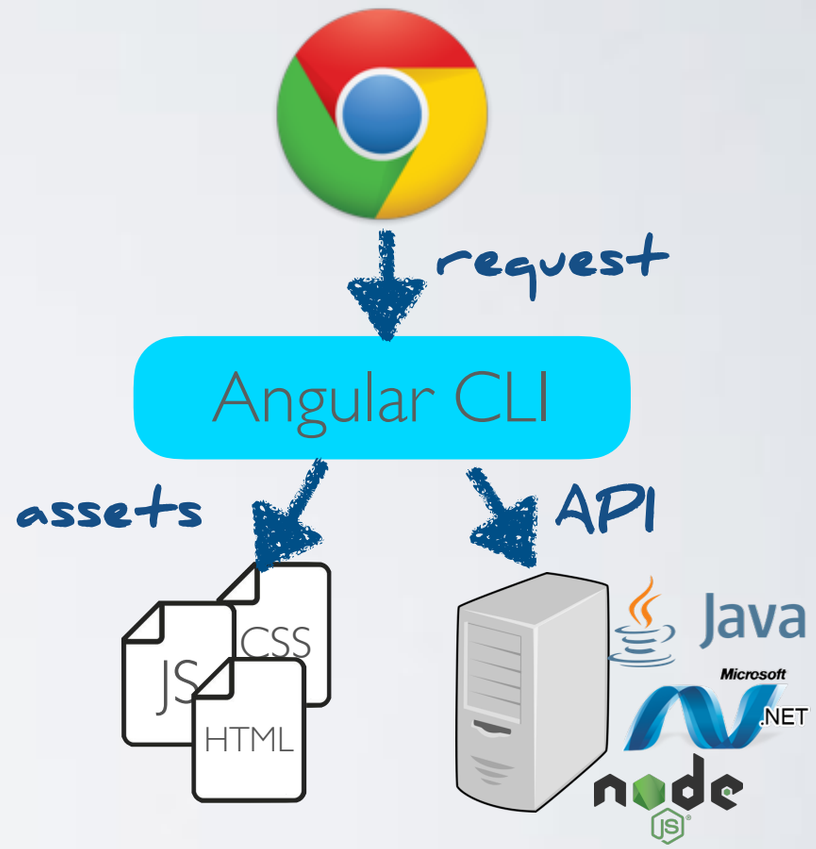
https://augury.angular.io/

CLI: Proxy To Backend

Scenario: Serve your Angular application (static asseets) locally during development but access an API which is hosted on another HTTP endpoint.

proxy.conf.json (file must be added manually)

```
{
  "/api": {
    "target": "http://localhost:3456",
    "pathRewrite": {"^/api" : ""},
    "secure": false
  }
}
```



package.json

```
"start": "ng serve --proxy-config proxy.conf.json",
```

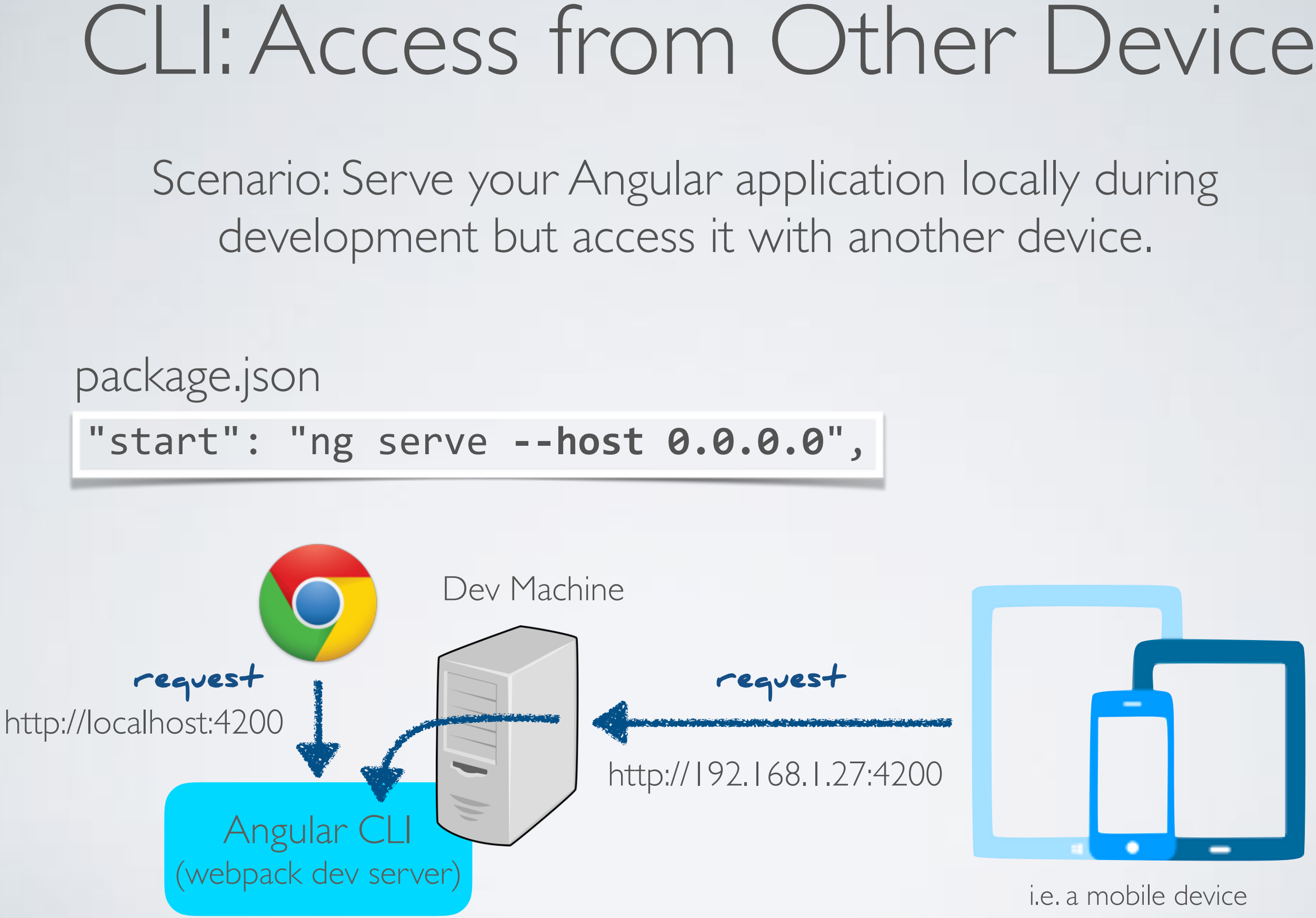
https://github.com/angular/angular-cli/blob/master/docs/documentation/stories/proxy.md

CLI: Access from Other Device

Scenario: Serve your Angular application locally during development but access it with another device.

package.json

```
"start": "ng serve --host 0.0.0.0",
```



CLI: Performance Budgets

angular.json

```
{
  ...
  "configurations": {
      "production": {
          ...
          "budgets": [
            {
              "type": "bundle",
              "name": "main",
              "baseline": "500kb",
              "warning": "50kb",
              "error": "100kb"
            }
          ]
```

With budgets you can control the size and growth of the JavaScript bundles over time.
The build will fail if the error limit is exeeded.

The feature is based on WebPack performance budgets:
https://webpack.js.org/configuration/performance/

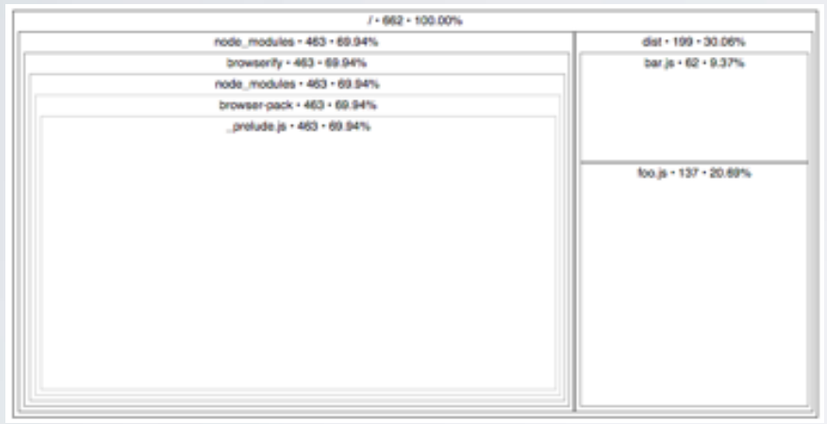
In Angular 7 the CLI generates a performance budget of 2MB (warning) 5MB (error).

The Chrome team at Google suggests a performance budget of 170KB for for the inital JavaScript bundle:
https://medium.com/@addyosmani/the-cost-of-javascript-in-2018-7d8950fbb5d4

https://github.com/angular/angular-cli/blob/master/docs/documentation/stories/budgets.md
https://medium.com/@tomastrajan/how-did-angular-cli-budgets-save-my-day-and-how-they-can-save-yours-300d534aae7a

Analyzing Bundles

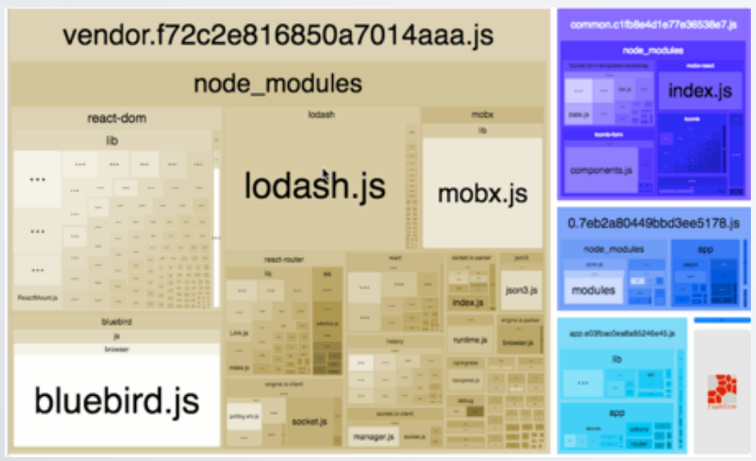
source-map-explorer:
https://www.npmjs.com/package/source-map-explorer




```
npm install -g source-map-explorer
```

```
ng build --prod --source-map
source-map-explorer dist/ng-app/main.XYZ.js
```

webpack-bundle-analyzer:
https://github.com/webpack-contrib/webpack-bundle-analyzer




```
npm install -g webpack-bundle-analyzer
```

```
ng build --prod --stats-json
webpack-bundle-analyzer dist/ng-app/stats.json
```

CLI: Adding Styling

global styles

angular.json

```
{
  ...
  "styles": [
     "src/styles.css",
     "node_modules/bootstrap/dist/css/bootstrap.min.css"
  ]
   ...
}
```

src/styles.css

```
@import '~bootstrap/dist/css/bootstrap.min.css';
```

src/styles.scss

```
@import '~bootstrap/scss/bootstrap.scss';
```

componentent styles

src/app/app.component.ts

```
@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styles: ['body {background-color: green}'],
  encapsulation: ViewEncapsulation.None
})
export class AppComponent {}
```

CLI: Adding 3rd Party Components

Installation:

```
npm install @ng-bootstrap/ng-bootstrap
```

Usage:

src/app/app.module.ts

```
import {NgbModule} from '@ng-bootstrap/ng-bootstrap';

@NgModule({
  declarations: [
    AppComponent
  ],
  imports: [
    BrowserModule,
    FormsModule,
    NgbModule.forRoot()
  ],
  providers: [],
  bootstrap: [AppComponent]
})
export class AppModule { }
```

src/app/app.component.ts

```
<ngb-datepicker></ngb-datepicker>
```

https://ng-bootstrap.github.io/#/home

Angular Components

Angular Components

Components are the main building-block of Angular.

Template + Class + Metadata

= Component

A Simple Component

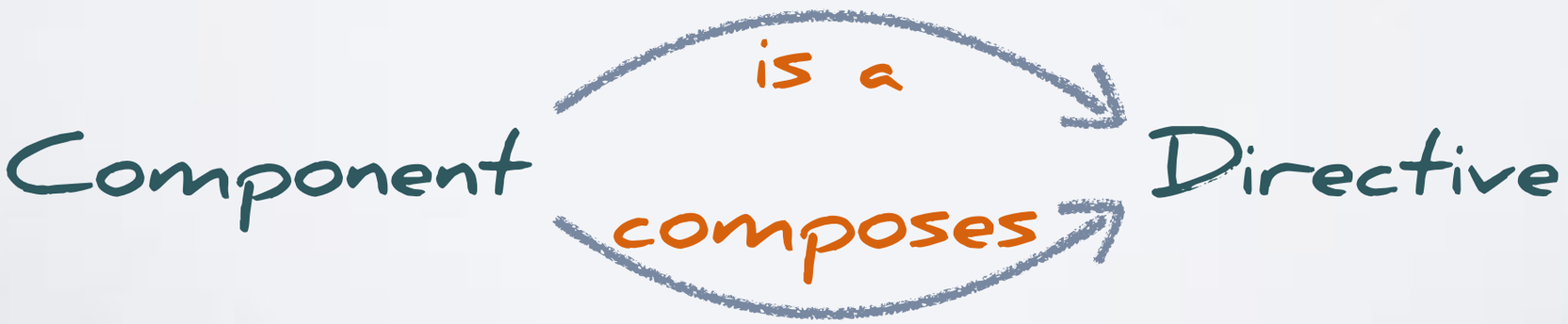
```
import {Component} from 'angular2/core';

@Component({
    selector: 'my-app',
    template: '<h1>My First Angular 2 App</h1>'
})
export class AppComponent { }
```

New components can be generated with the Angular CLI:

```
ng generate component hello
```

Directives & Components

A *component* is a special kind of directive.
A *directive* is a declarative instruction, that is embedded into a template and has a special meaning for the framework.



Databinding

DOM
(Template)

Interpolation

```
{{propName}}
```

Property Binding

```
[property]="propName"
```

Event Binding

```
(event)="methodName"
```

Two Way Binding

```
[(ngModel)]="propName"
```

(ngModel is provided by the FormsModule)

component

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

Write Your First Component

```
import {Component} from '@angular/core';

@Component({
    selector: 'greeter',
    template: `
<input [(ngModel)]="name"/>
Greetings: {{name}}
`
})
export class GreeterComponent {
  name;
  constructor() {
      this.name = 'John';
    }
}
```

1-way databinding

2-way databinding

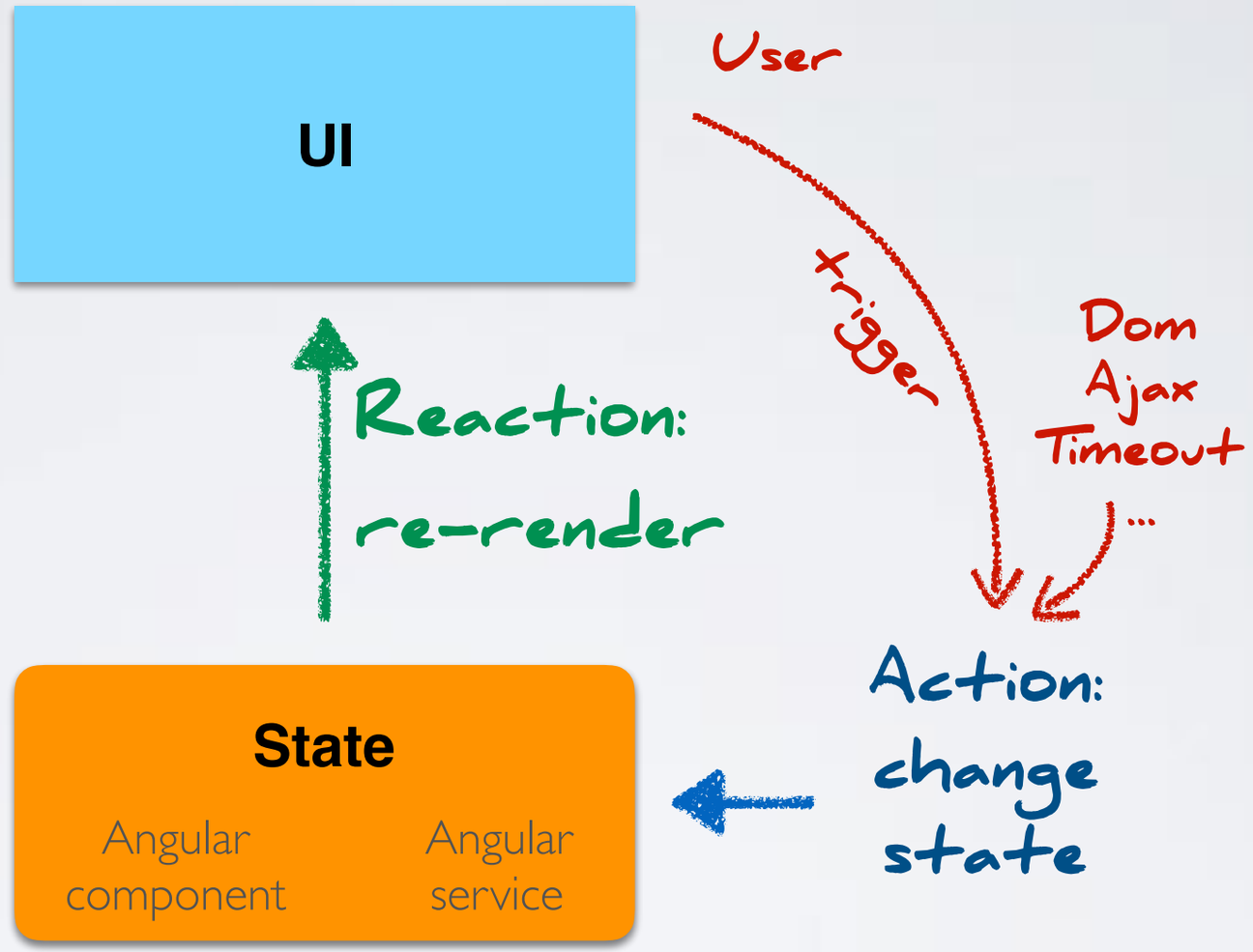
Note: **ngModel** is provided by the **FormsModule**

-> **FormsModule** must be imported in the angular module

EXERCISES

Exercise 2 – Create your first component

State is Managed in JavaScript



Reactivity: Angular reacts on state changes and updates the UI.

Interpolation vs. Property Binding

Interpolation always provides a string:

```
<h3>
  {{title}}
  <img src="{{heroImageUrl}}" style="height:30px">
</h3>
```

Background: Html attributes can't be changed at runtime, DOM properties can be changed.

Angular converts interpolated "attributes" into a property bindings:

```
<input value="{{title + title2}}">
```



```
<input [value]="title + title2">
```

```
<div innerHTML="{{'<h1>Test2</h1>'}}"></div>
```

Note: There is no 'innerHTML' attribute in html just a innerHTML property in the DOM.

Property binding must be used to provide non-string values:

```
<button [disabled]="isDisabled">Try Me</button>
```

Special Element Bindings

CSS class binding (DOM: classList)

```
<p [class]="myClasses"></p>
```

-> `myClasses` evaluates to a string

```
<p [class.is-active]="isActive"></p>
```

-> `isActive` evaluates to a boolean

```
<div [ngClass]="['bold-text', 'green']">array of classes</div>
<div [ngClass]="'italic-text blue'">string of classes</div>
<div [ngClass]="{'small-text': true, 'red': true}”>
   object of classes
</div>
```

CSS style binding (DOM: style property)

```
<p [style.display]="!isActive ? 'none' : null"></p>
```

```
<div [ngStyle]="{'color': color, 'font-size': size>}”
style using ngStyle
</div>
```

`ngClass` and `ngStyle` are directives and should be preferred when setting several classes / inline styles

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

Attribute binding (DOM: setAttribute)

```
<p [attr.role]="role"></p>
```

Special Event Bindings

Angular provides "pseudo-events":

```
<input type="text" (keyup.enter)="handleEnter($event)">
```

```
<input type="text" (keyup.shift.enter)="handleShiftEnter($event)">
```

Unfortunately the supported pseudo events are not documented …?

https://angular.io/guide/user-input#key-event-filtering-with-keyenter

Testing Components

Tests are run via Karma: **`npm run test`**

For unit tests you want to keep the "code under test" small. Angular provides a **`TestBed`** to assemble a module which can only contain the minimal amount of code.

Drawback: The **`TestBed`** must be configured for each test.

When working with external templates, the assembling of the **`TestBed`** must be asynchronous.

The **`TestBed`** can create component fixtures which provide a reference to the component object and the associated DOM element.

https://angular.io/guide/testing

Client Side Routing

Client Side Routing in a SPA

The traditional web is built on the concept of linked documents (URL, bookmarking, back-button …)

In a SPA the document is just the shell for an application. When you navigate away from the document, the application is "stopped".

As a consequence a SPA should "emulate" the traditional user-experience on the web:

- navigate via urls, links & back-button
- bookmarks and deep links

Client-Side Routing in a SPA

http://thedomain:80/app#/first

Browser

Server

initial request
http://thedomain:80/app

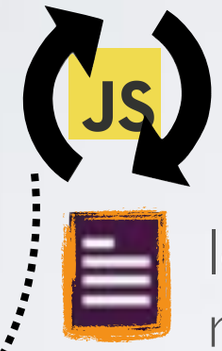
html-shell

html-shell is loaded

assets

optional ajax request
http://thedomain:80/api/data/first

data



"navigate" to "first" screen

load data &
render screen with JS

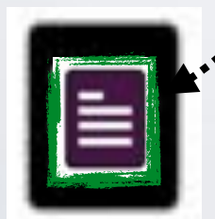
bootstrapping app

http://thedomain:80/app#/**first**

navigation

"navigate" to
"second" screen

load data &
render screen with JS



http://thedomain:80/app#/**second**

optional ajax request
http://thedomain:80/api/data/second

data

Client-Side Routing

Single Page Applications can contain several "pages"/views.

Routing maps URLs to views. aka: “Deep-Linking”

Parts of the URL are evaluated on the client.

Traditional approach: "Hash-Routing" (#)

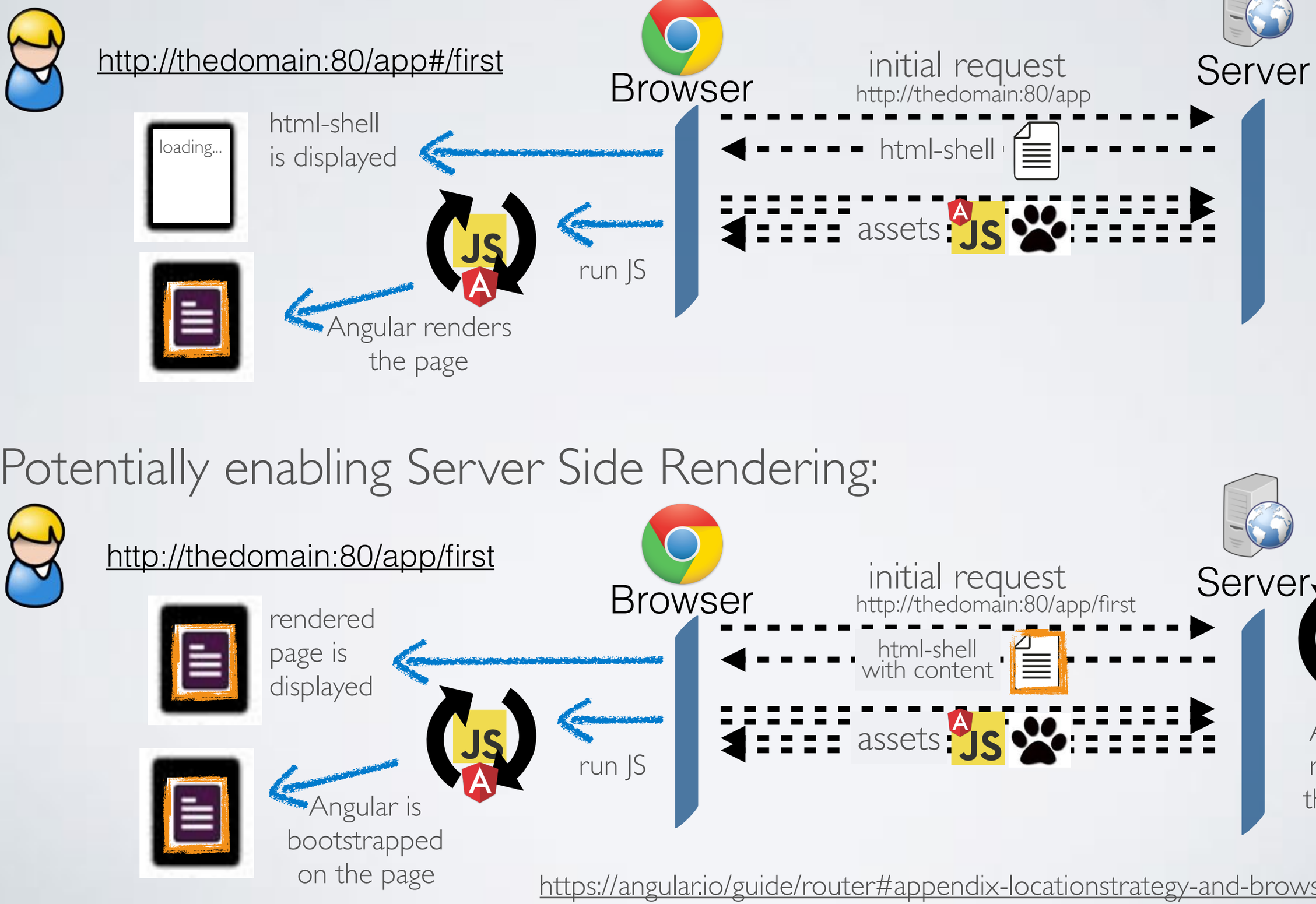
- client-route (part after the **#**) is not sent to the server with the initial http-request
- navigation: browser does not send a request to the server since only the part after the # is changing.

Modern approach: "HTML5 Routing" / "Push-State Routing"

- client-route is sent to the server with the initial http-request
- navigation: url is changed on the client, but the browser does not send a request to the server
- Pitfalls:
  - Server must respond with the "default" shell for unknnown urls (url-rewrite, fallback-url, https://angular.io/guide/deployment#server-configuration)
  - Only supportet in browsers > IE 9 (https://caniuse.com/#feat=history)

Hash-Routing vs. HTML5 Routing

Traditional SPA:



Potentially enabling Server Side Rendering:

https://angular.io/guide/router#appendix-locationstrategy-and-browser-url-styles

Server-Side Rendering of a SPA

The "initial load" of a SPA is always slower than a server rendered application (several roundtrips, JavaScript parse & execute).

Perceived performance can be increased, if the first request (shell) already contains the html representing the initial screen. Current frameworks can rendered on the server to deliver the first screen and then "attach" on the client to provide the functionality.

Implication:
- Node.js on the server!
- Time to first paint increases, time to interaction does not increase!

In many projects server-side rendering is not worth the complexity. For performance optimization there are lower hanging fruits.

Routing in Angular

Angular provides the **`@angular/router`** module, which helps to implement complicated routing scenarios.

With the Angular router module you can map paths to components.

https://angular.io/guide/router

Generating an app with routing with the Angular CLI

```
ng new fantastic-ng --routing --defaults
```

In `src/app/app-routing.module.ts` routes can be configured:

```
const routes: Routes = [
  { path: '', pathMatch: 'full', redirectTo: 'first'  },
  { path: 'first', component: FirstComponent  },
  { path: 'second', component: SecondComponent  }
];
```

The router-module provides the `<router-outlet>` directive which is used in `app.component.html`

Routing

Angular provides the **`@angular/router`** package for client-side routing.

Routing is configured statically when loading a module.
It is *good practice* to configure routing in a separate module, which is imported by the app module.

```
const routes: Routes = [
  { path: '', component: OverviewComponent },
  { path: 'done', component: DoneTodosComponent }
];

@NgModule({
  imports: [RouterModule.forRoot(routes)],
  exports: [RouterModule]
})
export class AppRoutingModule { }
```

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

Angular uses the "HTML5 routing" per default. Use

`RouterModule.forRoot(routes, {useHash: true})`

to get Hash-Routing

http://localhost:8080/app

<head>
</head>

<body>

App Component

...

<router-outlet>

...

</body>

SPA Shell

/app/path1

compontent 1

/app/path2

component 2

Router Directives

The router provides directives for placing a "routed" component into a template.

Directives: `router-outlet`, `routerLink`, `routerLinkActive`

```
<nav>
  <a routerLink="/databinding" routerLinkActive="active">Databinding</a>
  <a routerLink="/pipes" routerLinkActive="active">Pipes</a>
</nav>

<router-outlet></router-outlet>
```

You can also pass a *parameter array* to the routerLink directive: `routerLink="[/details, 5]"`

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/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.

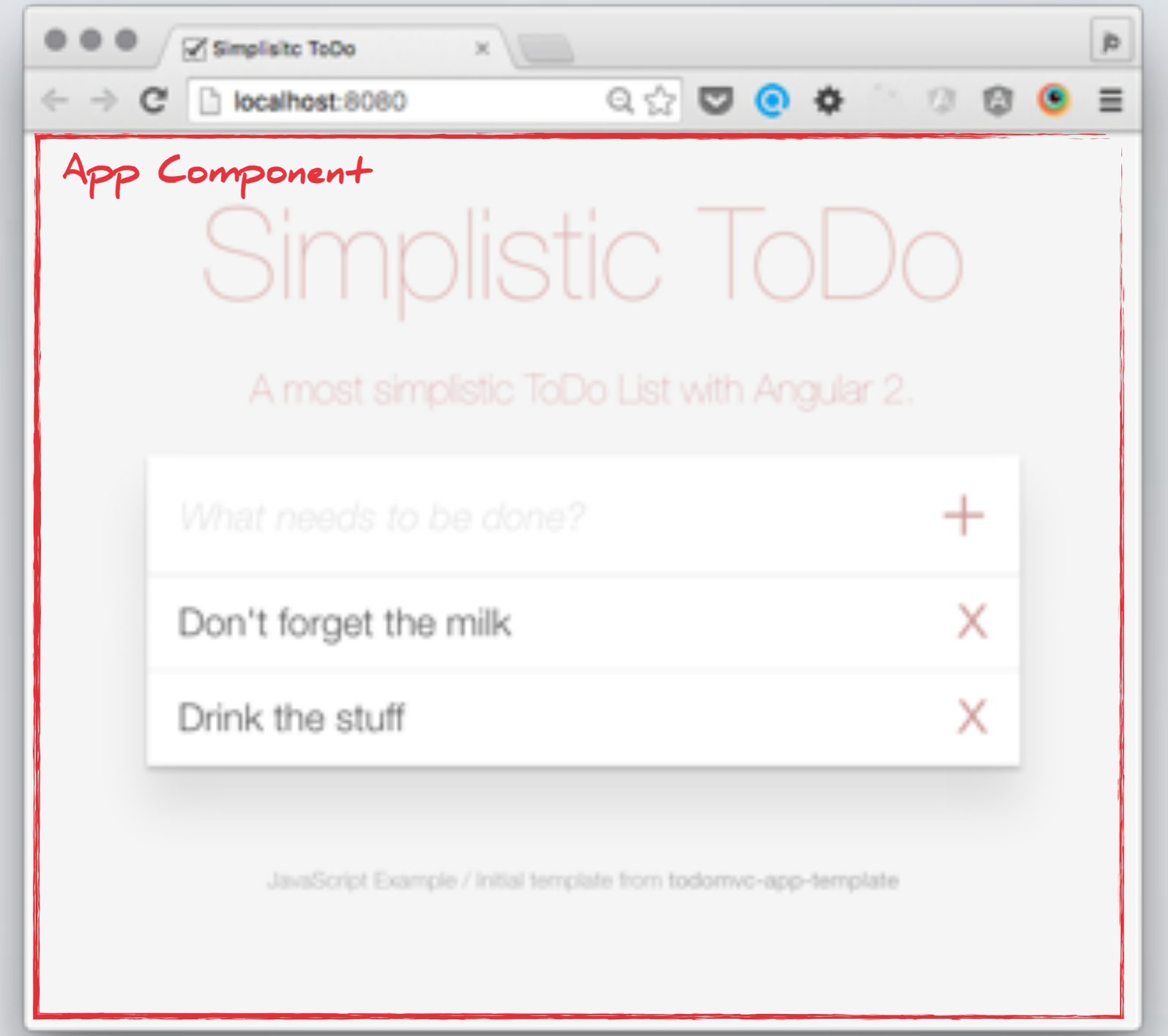
With a **trackBy** function Angular can reuse DOM elements.

```
<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

Demo: 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 offers matching interfaces: **`OnInit, AfterViewInit`** …

They are optional, just for tooling at build time.

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.

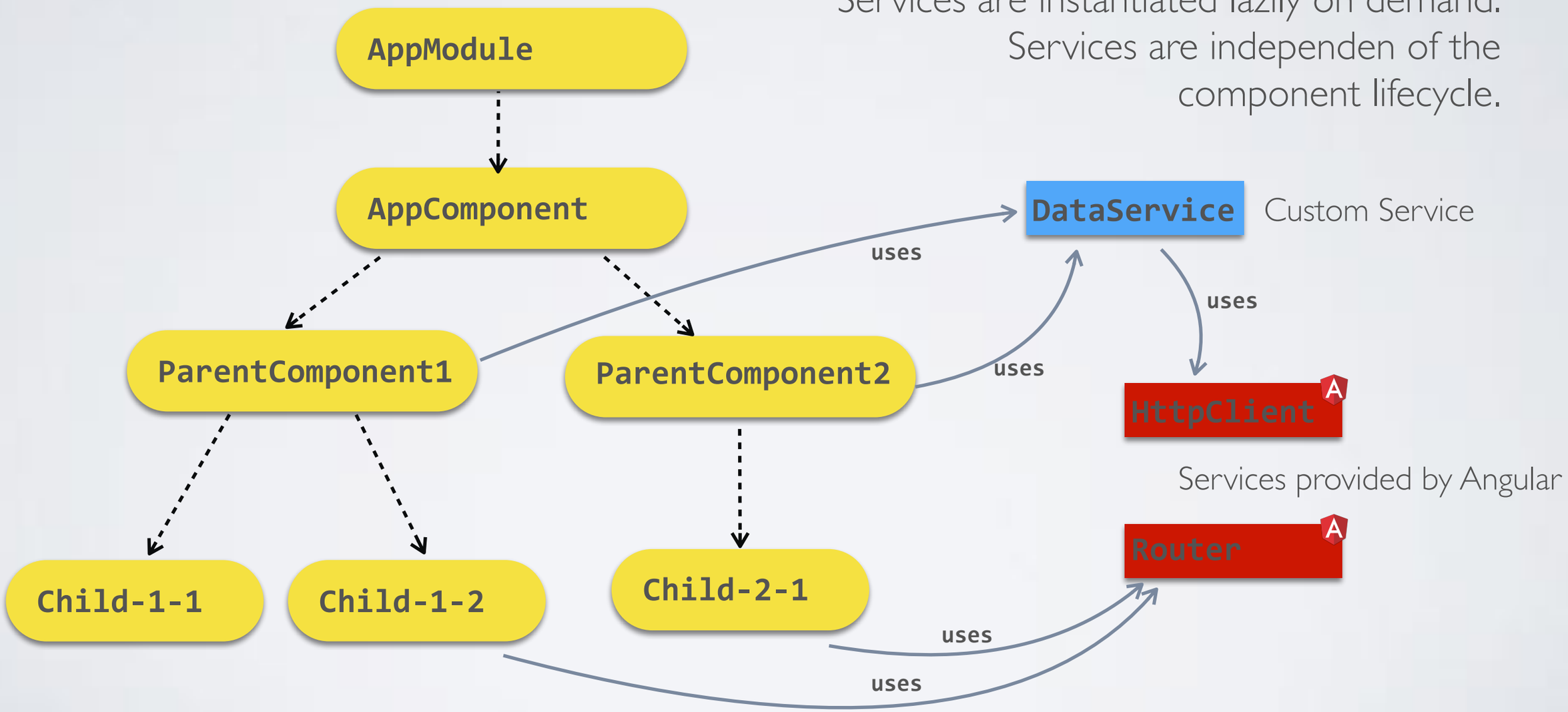
Services then can be requested via dependency-injection:

```
@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.

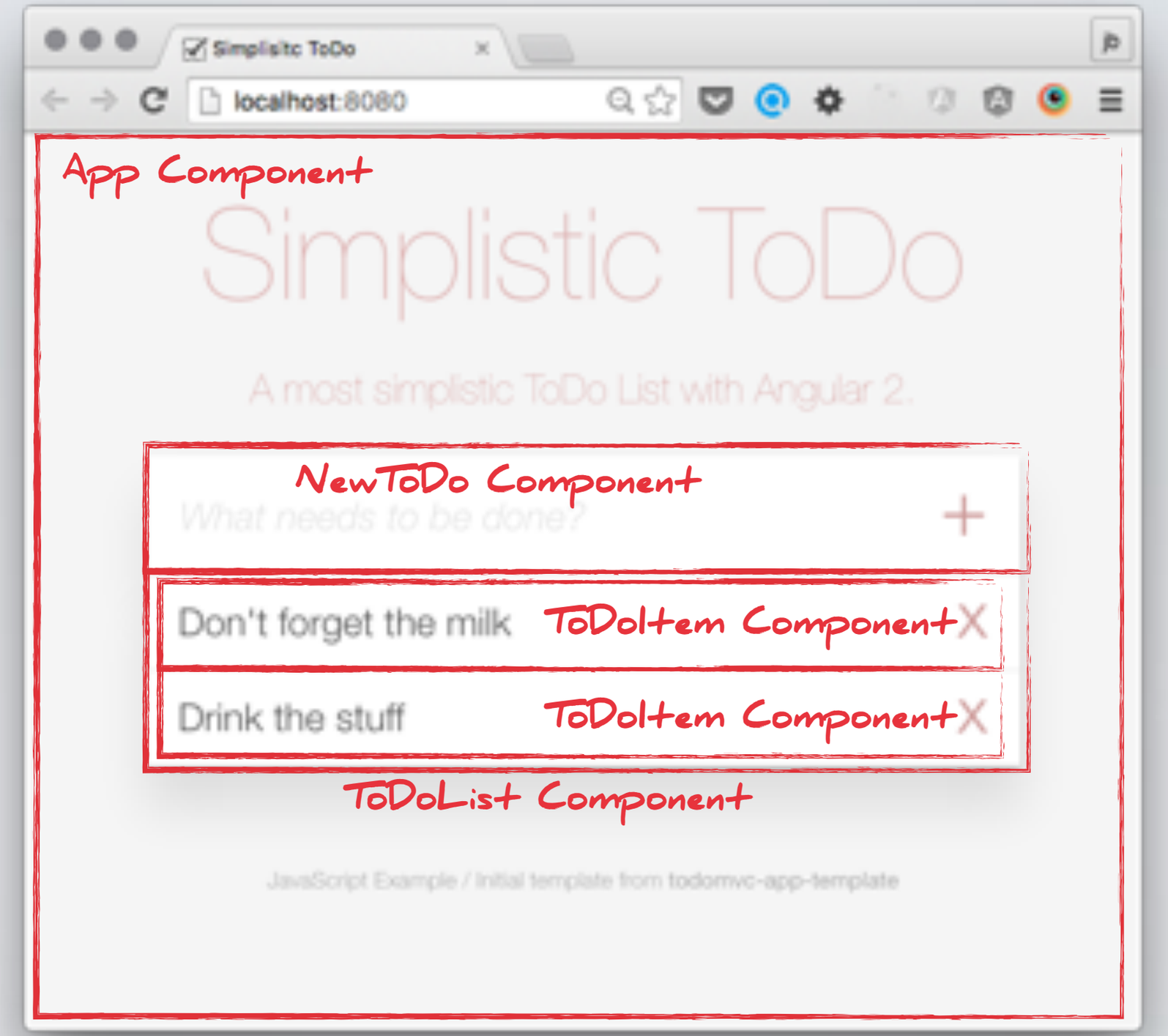


Services

- 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.

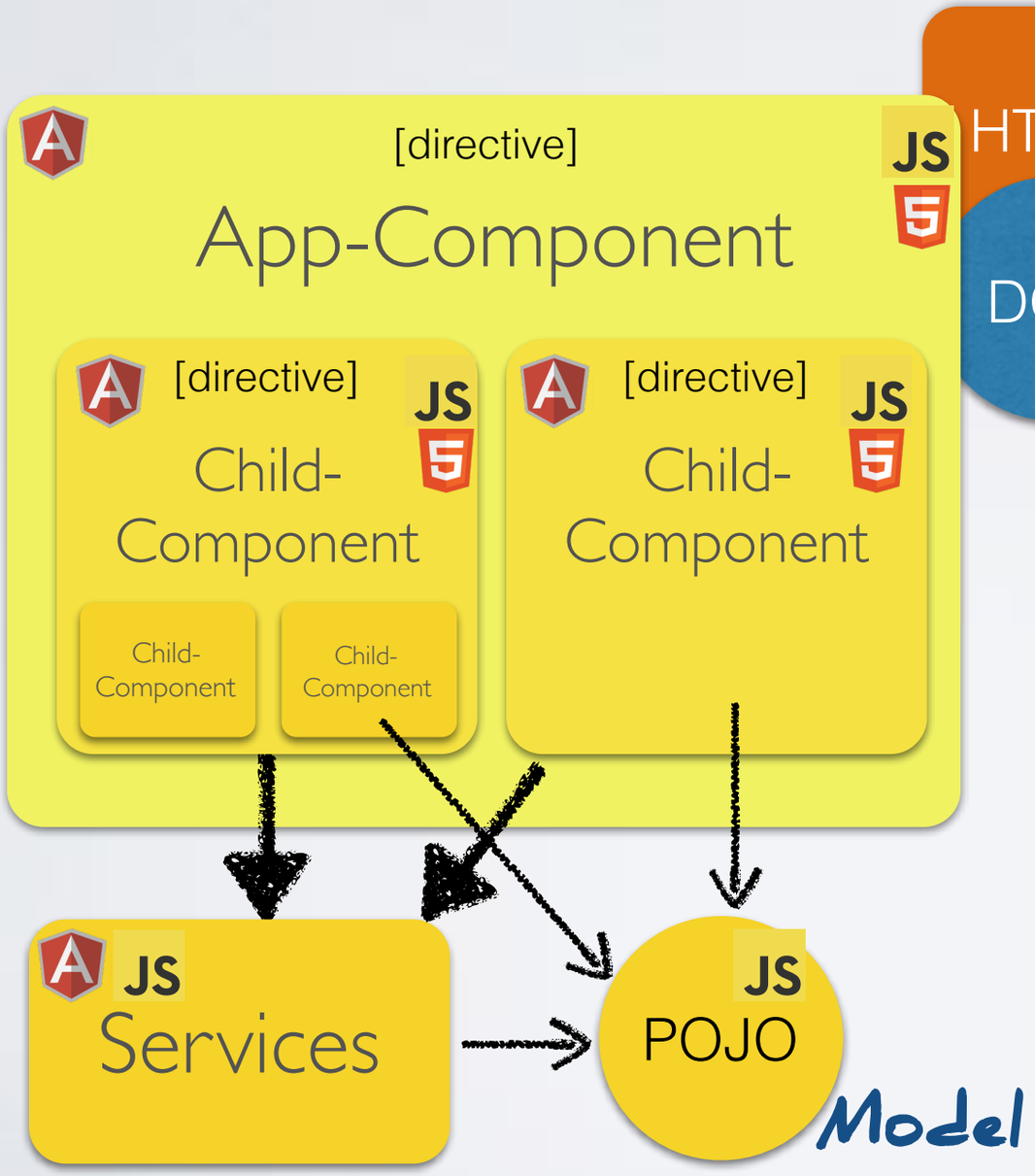
Component Architecture

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 {}
```

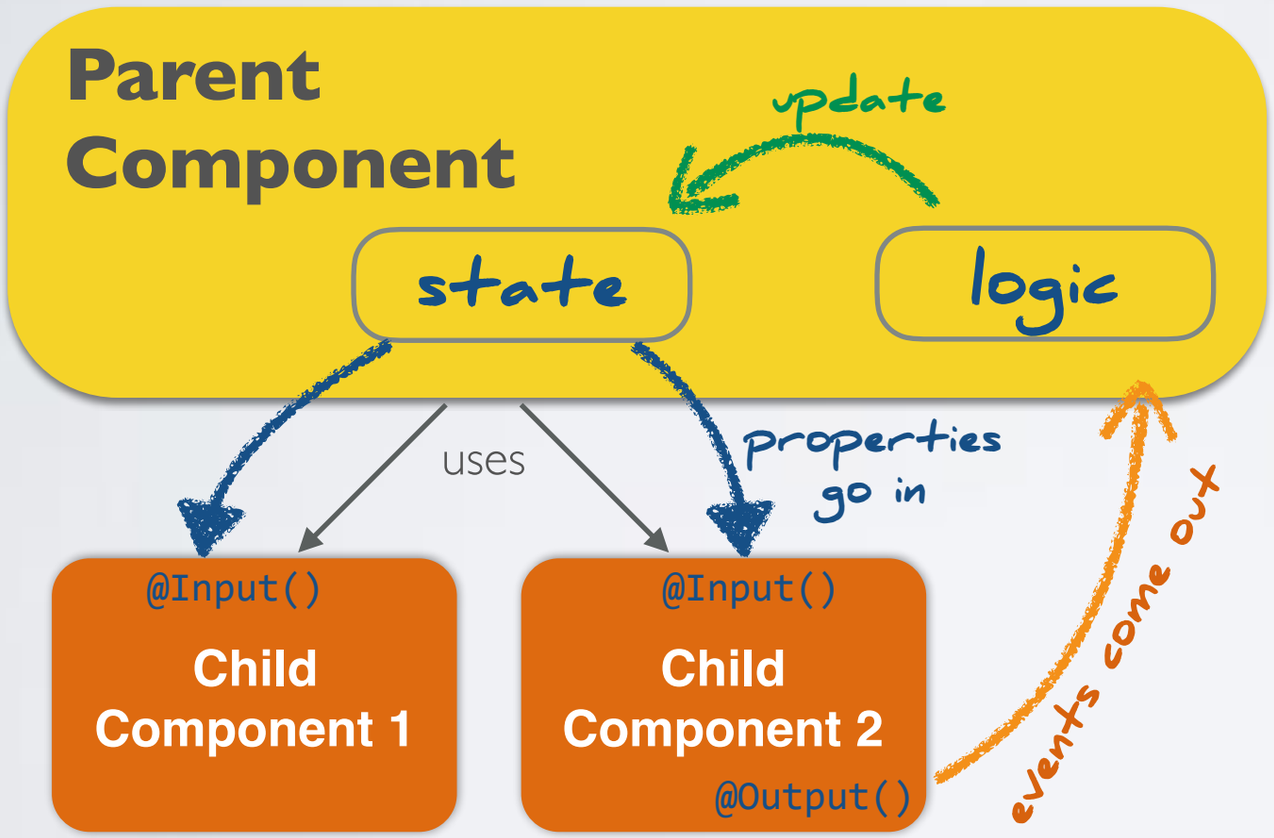
```
@Component({
  selector: 'child-component',
  templateUrl: 'child.html',
})
export class ChildComponent { }
```

child component

Nested Components: 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
- Children should not edit state of their parent
- Children "notify" parents (events, actions …)

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(...)}
}
```

Pattern: Container vs. Presentation Components

Application should 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

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.

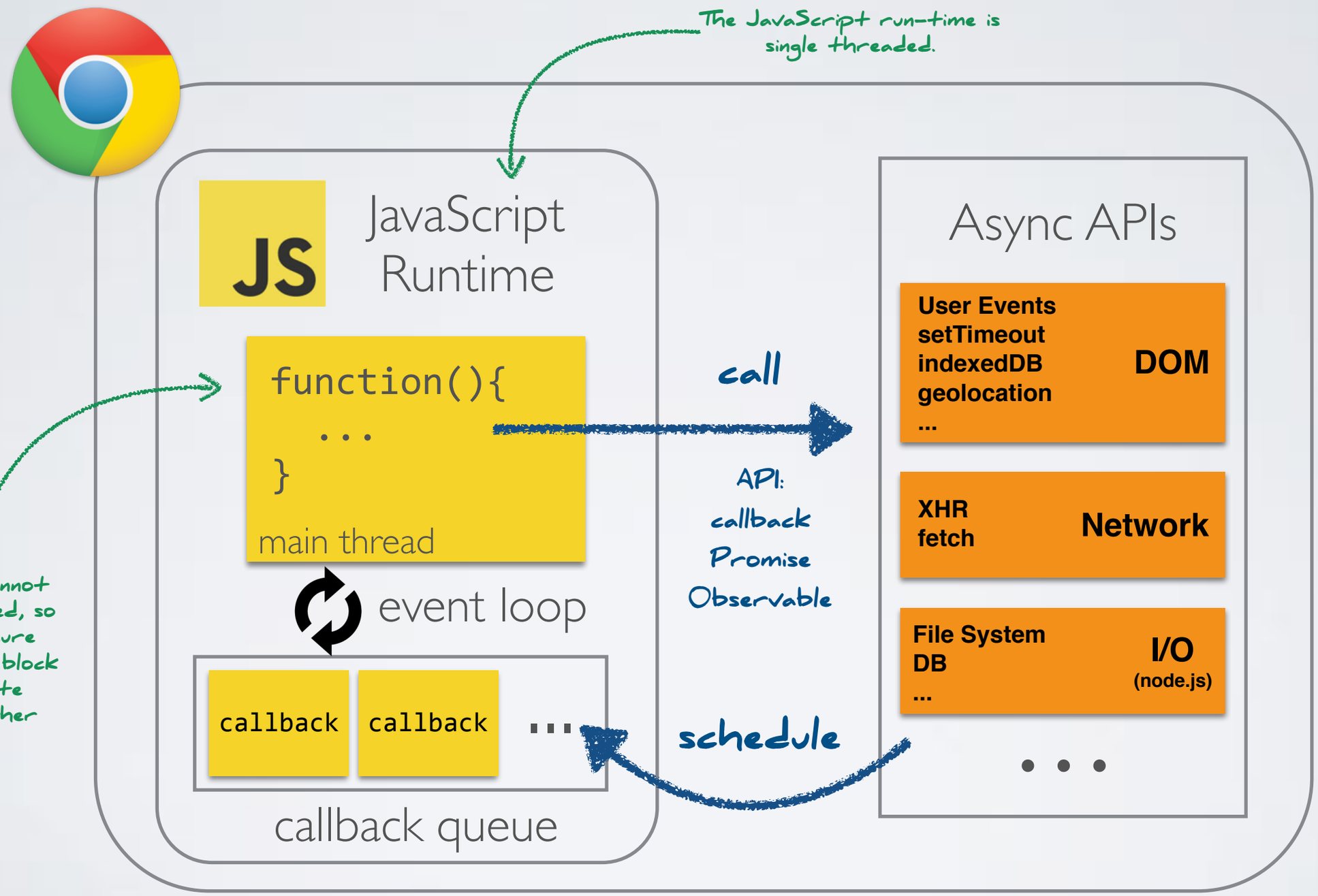
https://medium.com/@jtomaszewski/how-to-write-good-composable-and-pure-components-in-angular-2-1756945c0f5b
https://toddmotto.com/stateful-stateless-components
https://medium.com/curated-by-versett/building-maintainable-angular-2-applications-5b9ec4b463a1
https://blog.angular-university.io/angular-2-smart-components-vs-presentation-components-whats-the-differenc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

EXERCISES

Exercise 3: ToDo App - Component Architecture

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:

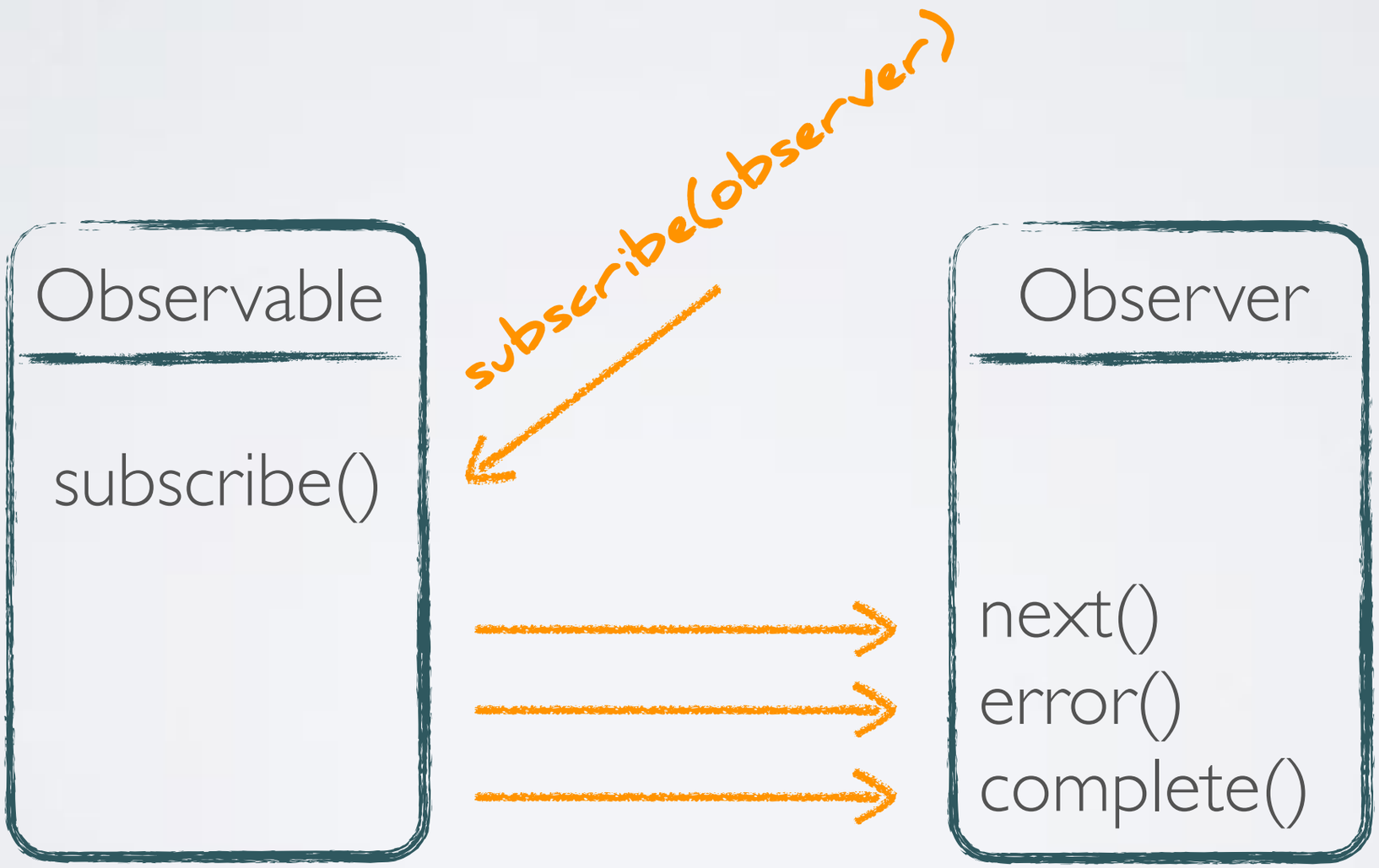
```
httpClient.get(URL)
  .subscribe(
    response => ...
  );
```

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

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

RxJS:
"The Observer pattern done right"

http://reactivex.io/



Using Observables

Consume an Observable:

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

```
const obsever = {
   next(value) {console.log('Received', value)},
   error(error) {console.log('Error!', error)},
   complete() {console.log('Completed!')}
}

var value$ = startAsyncOperation();
value$.subscribe(observer);
```

Provide an Observable (using RxJS):

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 stream" as an observable.

```
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$;
}
```

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.

Subscribe to Observable Streams

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)
```

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
});
```

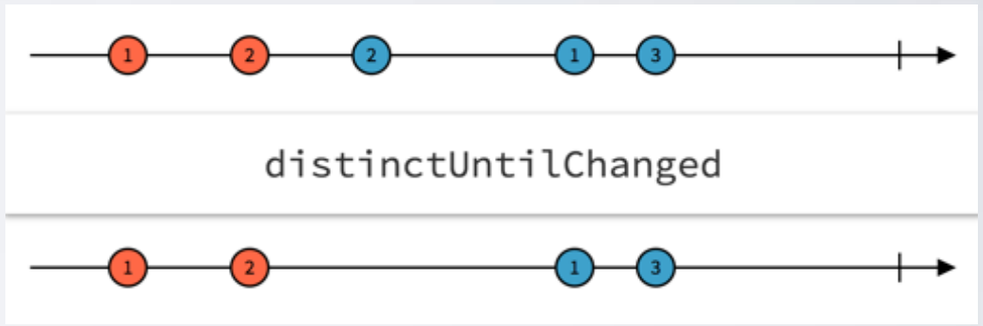
The producer produces values independent of subscriptions.

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

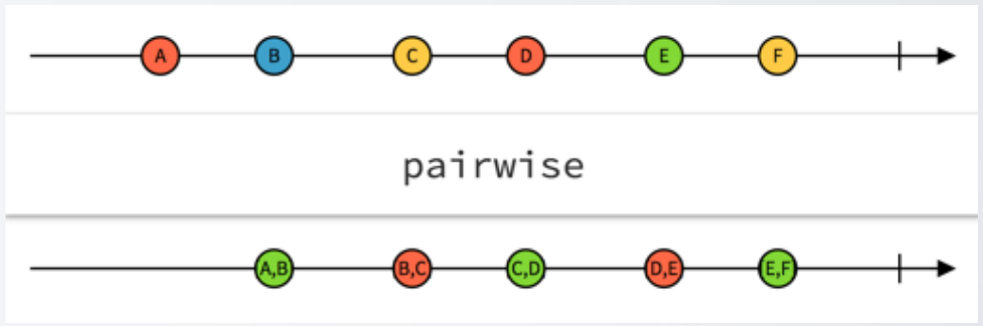
Transformation Operators

```
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()
                );
```



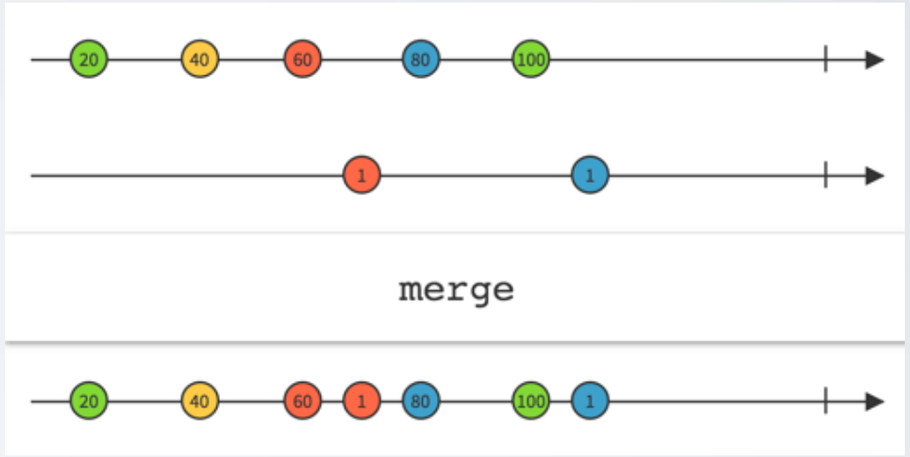

```
transformed.subscribe(
  v => console.log(v)
);
```

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

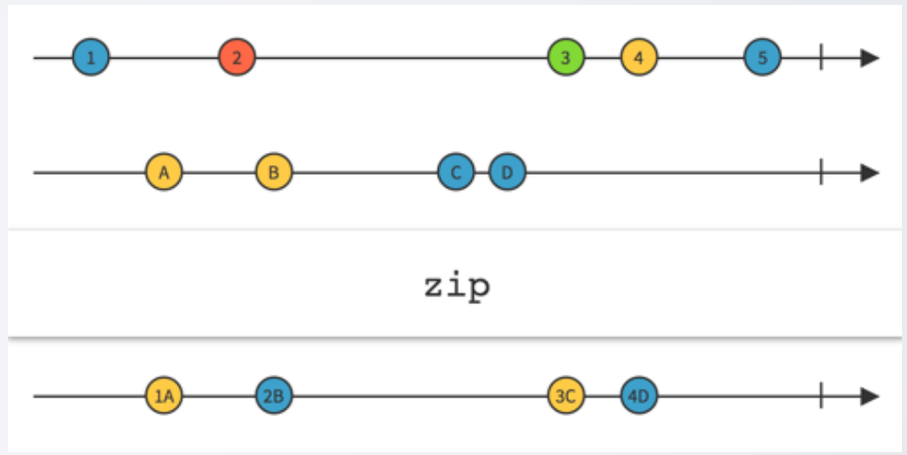
Combination Operators

```
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/

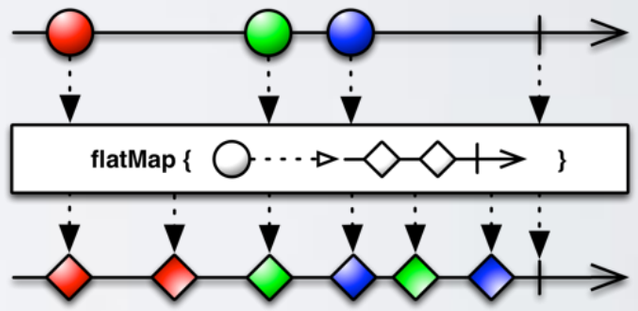
Higher Order Observables

"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]);
```

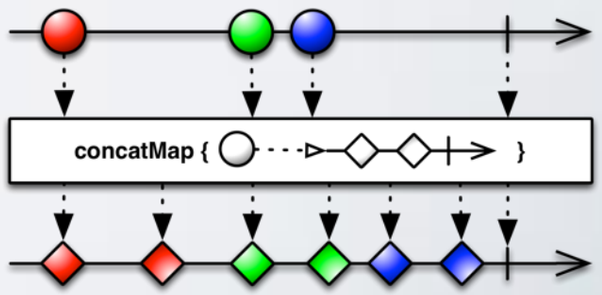
```
stream.flatMap(x => x())
      .subscribe(
          value => console.log(value)
      );
```

out:
44, 43, 42




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

out:
42, 43, 44



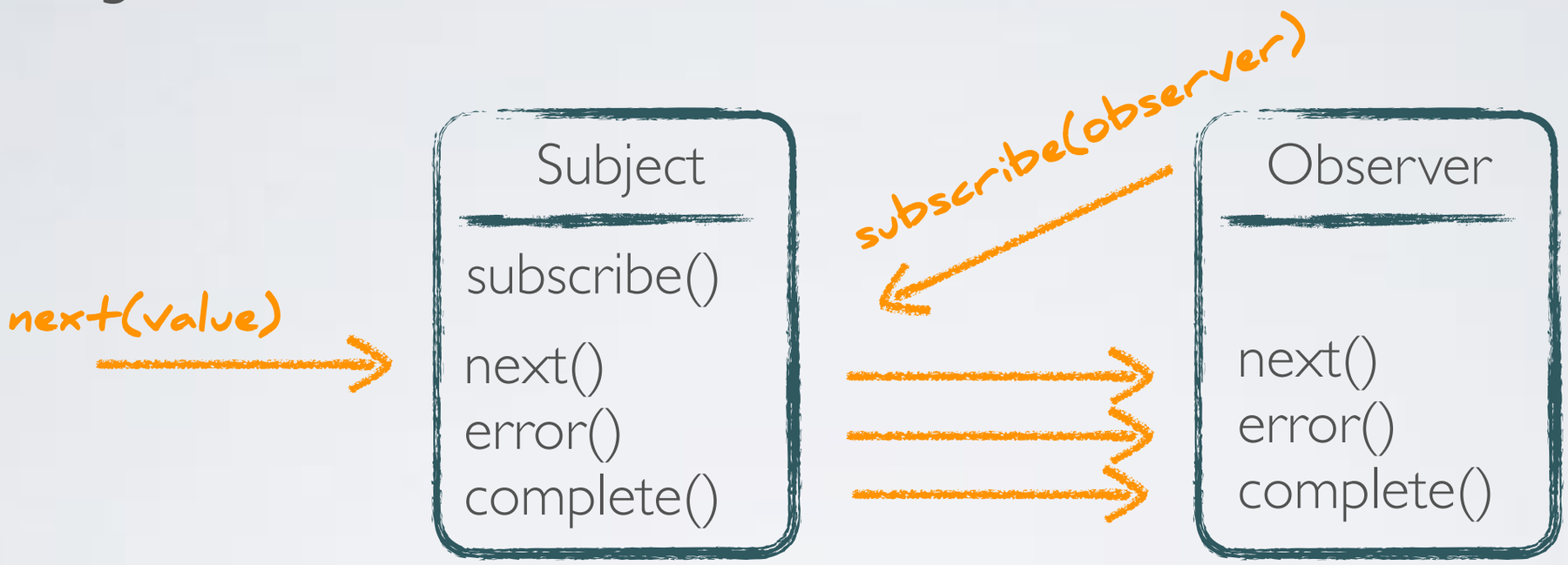
Funny: Switch Map Explained

NgConf 2018 Talk: "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);
```

Angular & RxJS: Unsubscribe Patterns

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

```
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();
  }
}
```

```
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/

Angular Backend Access

Angular 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>
```

The **`httpClient`** provides the typical http methods: **`get`**, **`post`**, **`put`**, **`delete`**

The **`httpClient`** returns the payload of the response as JavaScript object.
Ther is no need for deserialization.

Note: You must unsubscribe from observables that do not complete, else you potentially produce a memory leak! **`async`** pipes unsubscribes automatically.
The observables of **`httpClient`** *do* complete, therefore you don't need to unsubscribe.

Typed Backend Access

`HttpClient` can be used in a typed way, by passing type arguments.

```
interface ToDoData { id: string; title: String;  completed: boolean; }
interface ToDoGetResponse { data: ToDoData[]; }

this.httpClient
    .get<ToDoGetResponse>(backendUrl)
    .subscribe(
                response => this.response = response,
                error => this.error = error
          );
```

a single ToDoGetResponse

The type information is only for the TypeScript compiler at build time. At runtime there is no guarantee or check that the network call really returns objects of the specified type.

You should only use **`interface`** or **`type`** as *type arguments* for the **`HttpClient`**.
Do not use a **`class`** as type argument, since at runtime the response is *never an instance of a class*
(internally **`HttpClient`** just calls JSON.parse()).
If you need class instance in your application, then you have to do the mapping yourself:

```
this.http.get<ToDoGetResponse>(requestUrl)
  .pipe(
    map(response => response.data),
    map(todosDataArray => todosDataArray.map((todoData) => {return new ToDo(todoData);})
  )
  .subscribe( value => this.todos = value );
```

returns an Observable<ToDoGetResponse>

a single ToDoGetResponse

ToDoData[]

ToDoData

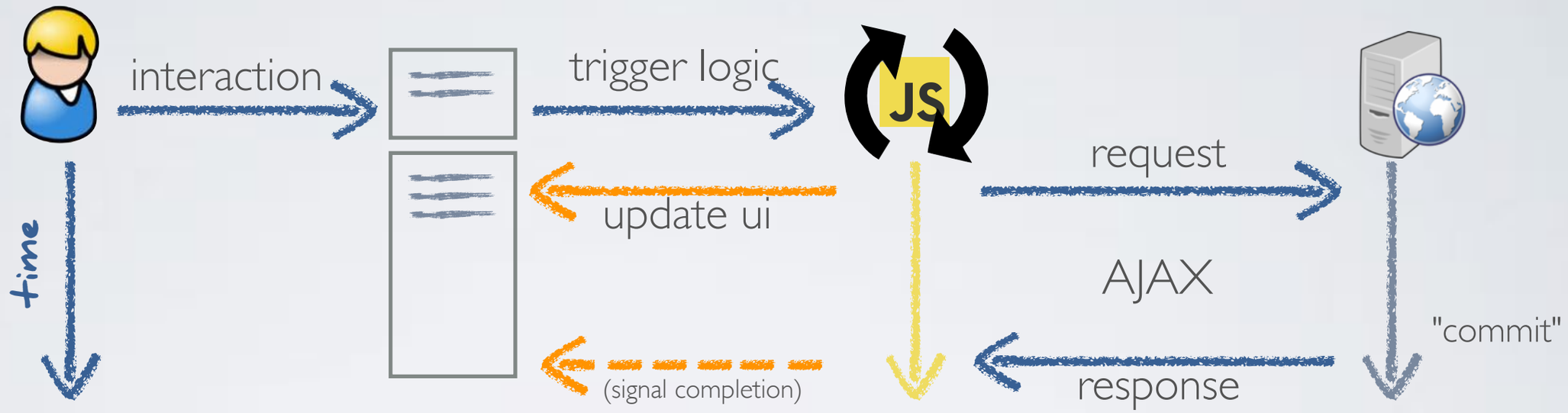
ToDo[]

EXERCISES

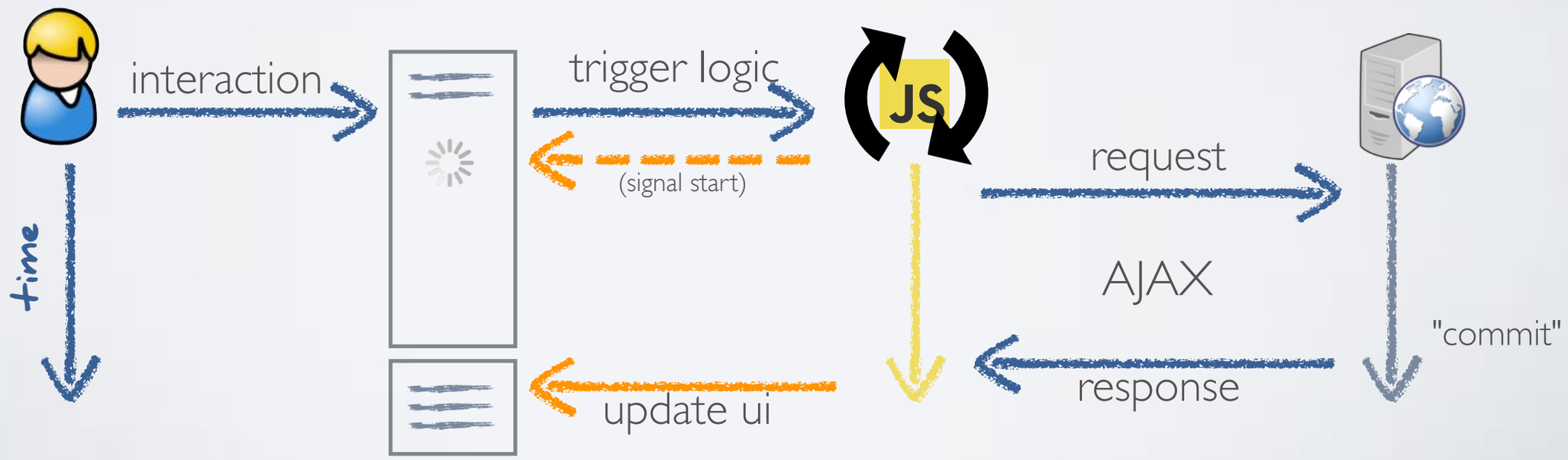
Exercise 4: ToDo – Backend Access

Optimistic UI vs. Pessimistic UI

Optimistic UI:



Pessimistic UI:



https://www.smashingmagazine.com/2016/11/true-lies-of-optimistic-user-interfaces/

Async Pipe Example

With the **async** pipe a template can be bound to the values emitted by an Observable.

The **async** pipe manages the subscription and *unsubscribes automatically* when the component is destroyed.

```
export class MyComponent {
    todos$: Observable<ToDo[]>;
    ngOnInit(){
        this.todos$ = service.loadToDos();
    }
}
```

'todos$' is an Observable

```
<td-todo-list
    [todos]="todos$ | async"
    (removeToDo)="completeToDo($event)">
 </td-todo-list>
```

@Input() 'todos' is an ToDo[]

Attention: each **async** pipe creates a subscription!

This can have undesired effects i.e. with "cold observables" where the values are produced with each subscription.

The following snippet would result in three AJAX requests!

```
<td-todo-list
    *ngIf="todos | async"
    [todos]="todos| async">
</td-todo-list>
<div *ngIf="!(todos | async)">
    <td-spinner></td-spinner>
</div>
```

Bad!

Solution: Using ***ngIf** to declare a variable holding the value

```
<div *ngIf="todos | async as loadedToDos">
    <td-todo-list
        *ngIf="loadedToDos"
        [todos]="loadedToDos">
    </td-todo-list>
    <div *ngIf="!loadedToDos">
      <td-spinner></td-spinner>
    </div>
</div>
```

Good!

HTTP Interceptor

HTTP requests and responses can be intercepted for inspection and transformation.

```
@Injectable()
export class DummyInterceptor implements HttpInterceptor {

  intercept(req: HttpRequest<any>, next: HttpHandler): Observable<HttpEvent<any>>
  {
    req = req.clone({
      setHeaders: {
        XX-DUMMY_HEADER: `DUMMY`
      }
    });
    return next.handle(req);
  }
}
```

app.module.ts

```
providers: [
  { provide: HTTP_INTERCEPTORS, useClass: DummyInterceptor, multi: true },
],
```

https://angular.io/guide/http#intercepting-requests-and-responses

HttpClient: Full Response

HttpClient returns the response payload as JavaScript object:

```
getData(): Observable<MyData> {
  return this.http.get<Config>("https://my-data-endpoint.com/data");
}
```

```
getData().subscribe(
  data => this.myData = data;
)
```

data is a JavaScript object, the TypeScript compiler assumes it is of type MyData

HttpClient can be configured to return the full response:

```
getData(): Observable<HttpResponse<MyData>> {
  return this.http.get<MyData>(
                                "https://my-data-endpoint.com/data",
                                { observe: 'response' }
                               );
}
```

the observable now emits a http response

configuration object

```
getData().subscribe(
  response => {
            this.myData = response.body;
            this.headers = response.headers;
            this.status = response.status;
          }
)
```

http response has different properties

https://angular.io/api/common/http/HttpResponse

HttpClient: Non-JSON data

HttpClient returns the response payload as JavaScript object:

```
getData(): Observable<MyData> {
  return this.http.get<Config>("https://my-data-endpoint.com/data");
}
```

```
getData().subscribe(
  data => this.myData = data;
)
```

data is a JavaScript object, the TypeScript compiler assumes it is of type MyData

HttpClient can be configured to return other content:

```
getData() {
  return this.http.get(
                       "https://my-data-endpoint.com/data.txt",
                       { responseType: 'text' }
                      );
}
```

returns an Observable<string>

configuration object

`responseType` can be: `json`, `text`, `blob`, `arraybuffer`

```
getData().subscribe(
  data => this.myText = data
)
```

data is a string

https://angular.io/guide/http#requesting-non-json-data

Structure with NgModules

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.

NgModules provide a level of encapsulation.

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

Don't confuse NgModules with ES2015 modules (import/export)!

Reason for multiple NgModules

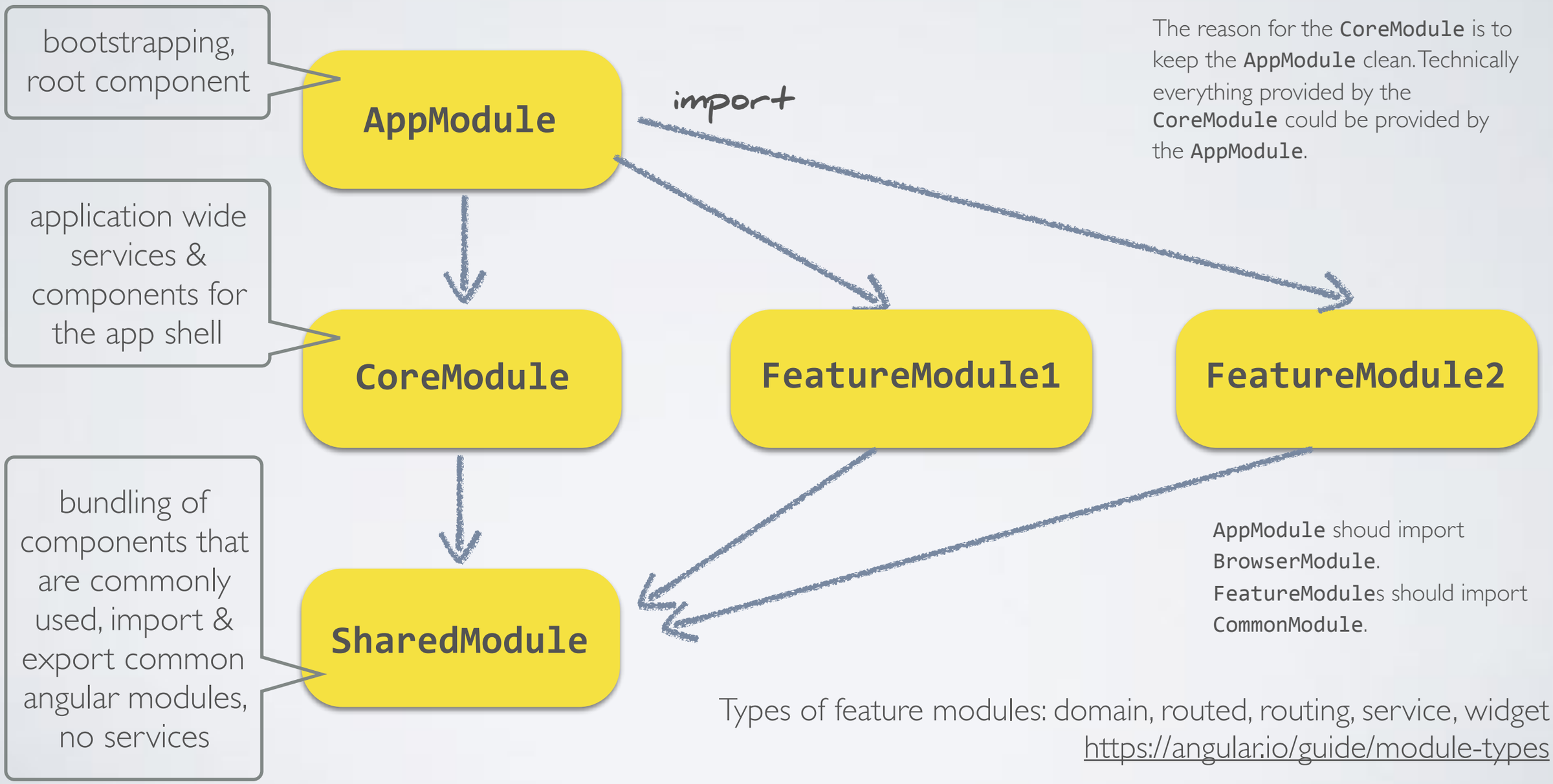
- Encapsulation:
  A NgModule can have "internal" and exported components, directives & pipes. Only exported constructs can be used from other NgModules.
  Note: Services are not encapusalted and always available "globally".

- Lazy-Loading:
  NgModules can bundled seperately and lazy loaded. This improves initial load and startup.

- Re-Use:
  NgModules are the unit for Re-Use. A consuming NgModule imports a providing NgModule and gets all the components/directives.

Structuring an App into NgModules

feature modules, shared module, core module



The reason for the `CoreModule` is to keep the `AppModule` clean. Technically everything provided by the `CoreModule` could be provided by the `AppModule`.

`AppModule` shoud import `BrowserModule`.
`FeatureModule`s should import `CommonModule`.

Types of feature modules: domain, routed, routing, service, widget
https://angular.io/guide/module-types

https://angular.io/guide/styleguide
https://medium.com/@michelestieven/organizing-angular-applications-f0510761d65a
https://medium.com/@tomastrajan/6-best-practices-pro-tips-for-angular-cli-better-developer-experience-7b328bc9db81

Understanding Angular Modules

https://angular.io/guide/module-types

Module provide an encapsulation/isolation for components/directives/pipes.

Module do not provide an encapsulation/isolation for services. They can be injected everywhere, when available via provider hierarchy.

Lazy loaded modules get their own new injector. This injector is a child of the root application injector.

Pattern: `forRoot` / `forChild`:

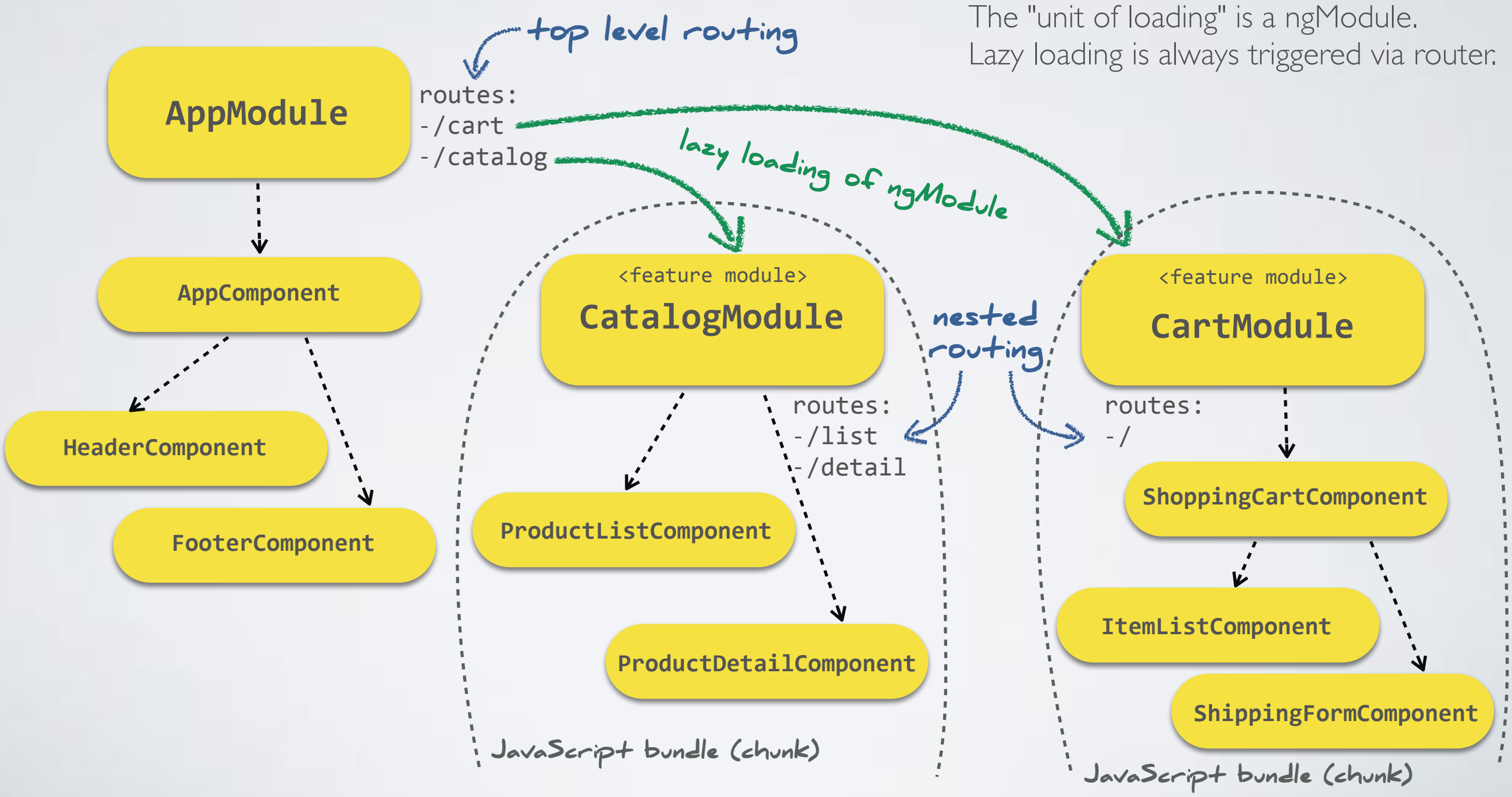
- https://angular.io/guide/singleton-services
- https://angular.io/guide/ngmodule-faq#what-is-the-forroot-method

https://medium.com/@cyrilletuzi/understanding-angular-modules-ngmodule-and-their-scopes-81e4ed6f7407

Lazy Loading

Routes can point to a NgModule which can be lazy-loaded on demand.

The "unit of loading" is a ngModule.
Lazy loading is always triggered via router.

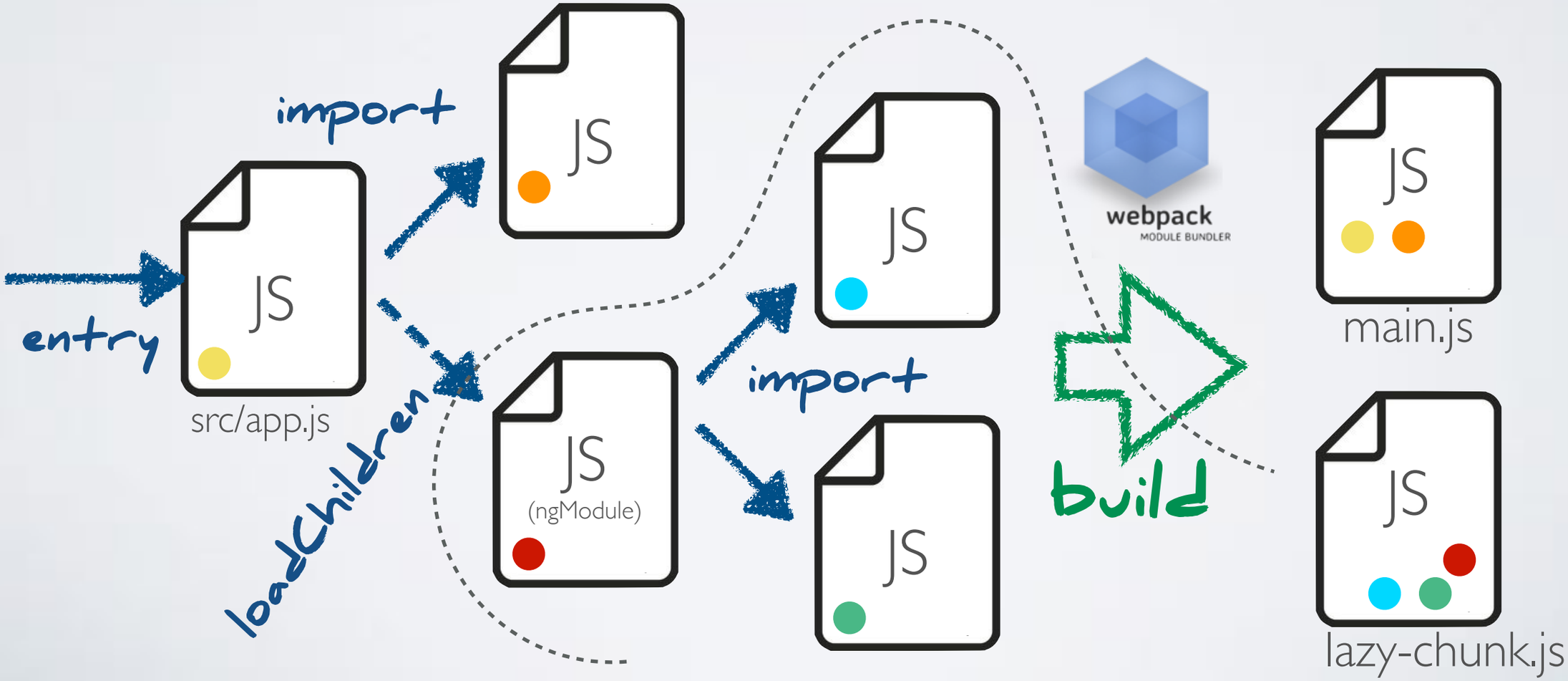


Lazy Loading

```
import HomeComponent from './home.component';

const routes: Routes = [
  {path: '', pathMatch: 'full', redirectTo: 'home'},
  {path: 'home', component: HomeComponent},
  {path: 'catalog', loadChildren: './catalog/catalog.module#CatalogModule'},
];
```

"magic" configuration for webpack to create a lazy chunk

tells Angular which ngModule should be started once the chunk is loaded.



Advanced Lazy-Loading Topics

Lazy-loading can be combined with *pre-loading*:
After the app has started, lazy-loaded modules are loaded even if they are not needed yet. The result is a fast initial load and faster interaction later.

Angular offers the `PreloadAllModules` strategy: https://angular.io/api/router/PreloadAllModules

`ngx-quicklink` downloads lazy-loaded modules associated with visible links on the screen: https://github.com/mgechev/ngx-quicklink

It is technically possible (but complicated) to lazy-load modules/components without the router: (this should become easier with Ivy)

- ngx-loadable: https://github.com/mohammedzamakhan/ngx-loadable
- lazy-af: https://www.npmjs.com/package/@herodevs/lazy-af

Lazy loading of libraries: https://medium.com/@tomastrajan/why-and-how-to-lazy-load-angular-libraries-a3bf1489fe24

Angular 8 should provide a new syntax to lazy-load modules with the "standard" dynamic `import` of JavaScript:

```
loadChildren : () => import('./catalog/catalog.module').then(m => m.CatalogModule)
```

Lazy Loading with ngModules: Pitfalls

It is possible to "prevent" lazy loading by referencing code from a lazy chunk directly or indirectly from your main chunk (sometimes the **`prod`** build fixes this again).

app-routing.module.ts

```
const routes: Routes = [
  { path: '', loadChildren: './todos/components/todo-screen/todo-screen.module#ToDoScreenModule' },
  { path: 'done', loadChildren: './todos/components/done-screen/done-todos.module#DoneScreenModule' },
  { path: 'details', loadChildren: './todos/components/detail-screen/detail-screen.module#DetailScreenModule' }
];
```

good: configuring lazy loading

app.module.ts

```
@NgModule({
  declarations: [
    AppComponent,
  ],
  imports: [
    BrowserModule, FormsModule, HttpClientModule,
    AppRoutingModule,
    ToDoScreenModule, DoneScreenModule,
  ],
  providers: [],
  bootstrap: [AppComponent]
})
export class AppModule {}
```

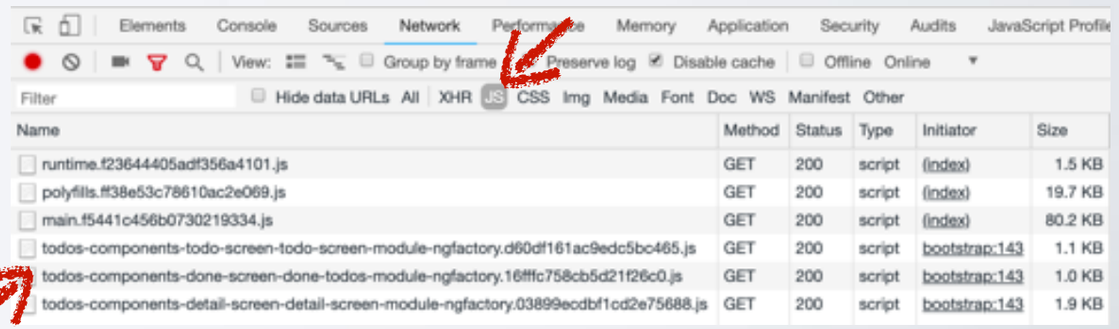
bad: referencing the lazy modules from the app module

Make sure that the chunks are built for **`prod`**, and check the sizes of the chunks.

```
> ng build --prod --source-map --stats-json --named-chunks

Date: 2019-02-17T15:01:09.772Z
Hash: 4d0dfa696b043c4de141
Time: 10190ms
chunk {0} common.31d114eb789a82a314e0.js, common.31d114eb789a82a314e0.js.map (common) 3.51 kB  [rendered]
chunk {1} todos-components-detail-screen-detail-screen-module-ngfactory.ac2a927878d74eb21c90.js,
  todos-components-detail-screen-detail-screen-module-ngfactory.ac2a927878d74eb21c90.js.map
  (todos-components-detail-screen-detail-screen-module-ngfactory) 3.97 kB  [rendered]
chunk {2} todos-components-done-screen-done-todos-module-ngfactory.0a0e7f5fe16d6217362b.js,
  todos-components-done-screen-done-todos-module-ngfactory.0a0e7f5fe16d6217362b.js.map
  (todos-components-done-screen-done-todos-module-ngfactory) 2.64 kB  [rendered]
chunk {3} todos-components-todo-screen-todo-screen-module-ngfactory.72f1a7a201c4d9135f47.js,
  todos-components-todo-screen-todo-screen-module-ngfactory.72f1a7a201c4d9135f47.js.map
  (todos-components-todo-screen-todo-screen-module-ngfactory) 7.1 kB  [rendered]
chunk {4} runtime.4854e73b9233c4b80f6b.js, runtime.4854e73b9233c4b80f6b.js.map (runtime) 2.17 kB [entry]
  [rendered]
chunk {5} styles.eb711f81ca29236479fe.css, styles.eb711f81ca29236479fe.css.map (styles) 2.35 kB [initial]
  [rendered]
chunk {6} polyfills.c633236d6593784b7a72.js, polyfills.c633236d6593784b7a72.js.map (polyfills) 59.6 kB
  [initial] [rendered]
chunk {7} main.cab068f5f3e7d042f3fe.js, main.cab068f5f3e7d042f3fe.js.map (main) 312 kB [initial] [rendered]

Process finished with exit code 0
```

ok!

Check that the chunks are loaded at runtime:



ok!

Check for all "entry" urls of your app, that only the needed chunks are loaded!

Change Detection

Rich Harris
@Rich_Harris

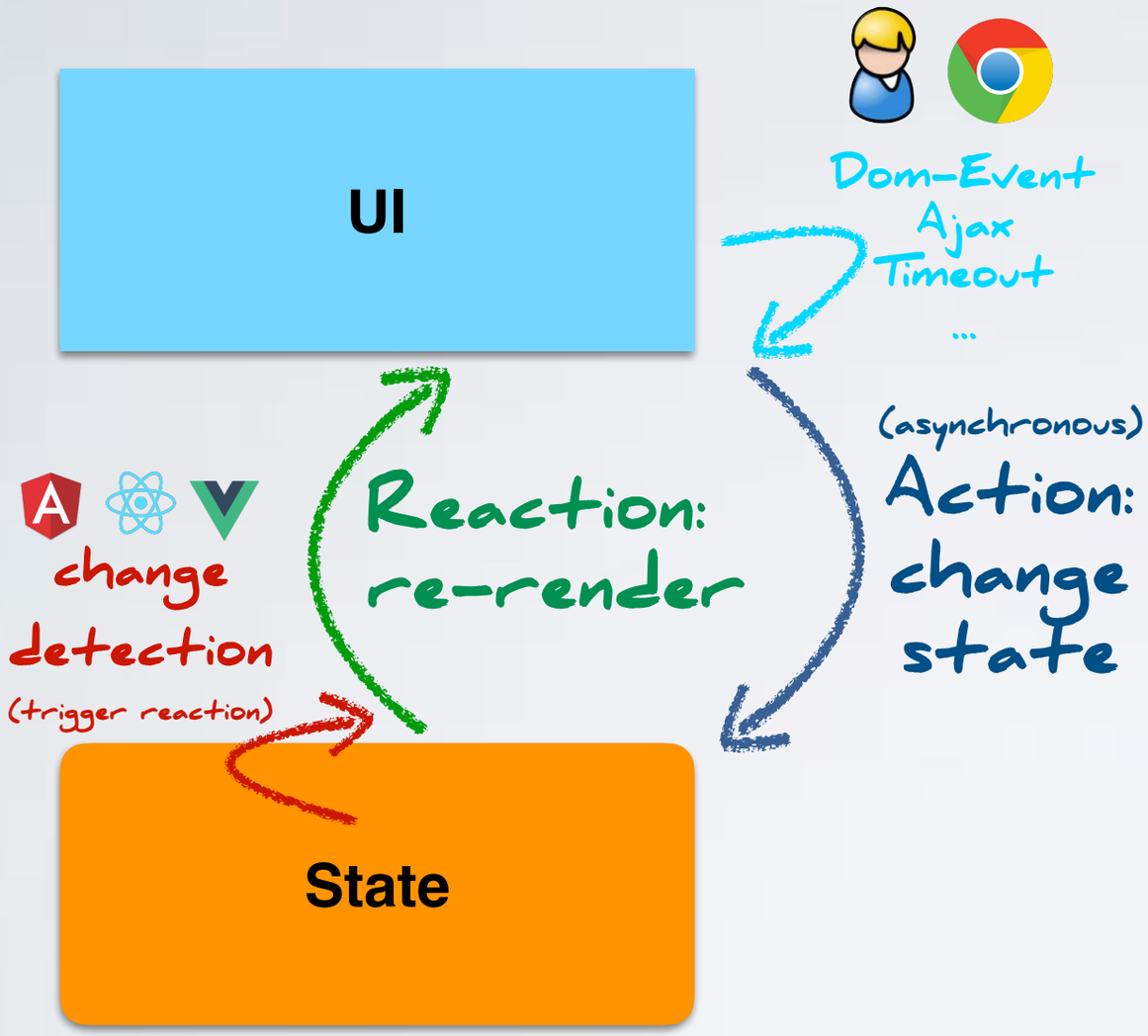
Following

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 🤷)

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

Frameworks & Change Detection



Front end development and change detection (Angular vs. React):
https://www.youtube.com/watch?v=1i8klHov3vA
https://github.com/in-depth-education/change-detection-in-web-frameworks

https://blog.thoughtram.io/angular/2016/02/22/angular-2-change-detection-explained.html
https://vuejs.org/v2/guide/reactivity.html

Angular: Has a "magic" Change Detection

Angular comes with *Zone.js*. This is a library that patches the native Browser APIs so that applications can be notified when a potential change happens. It triggers the Angular change-detection algorithm which performs some form of dirty checking.

React: Change Detection is explicit via `setState`

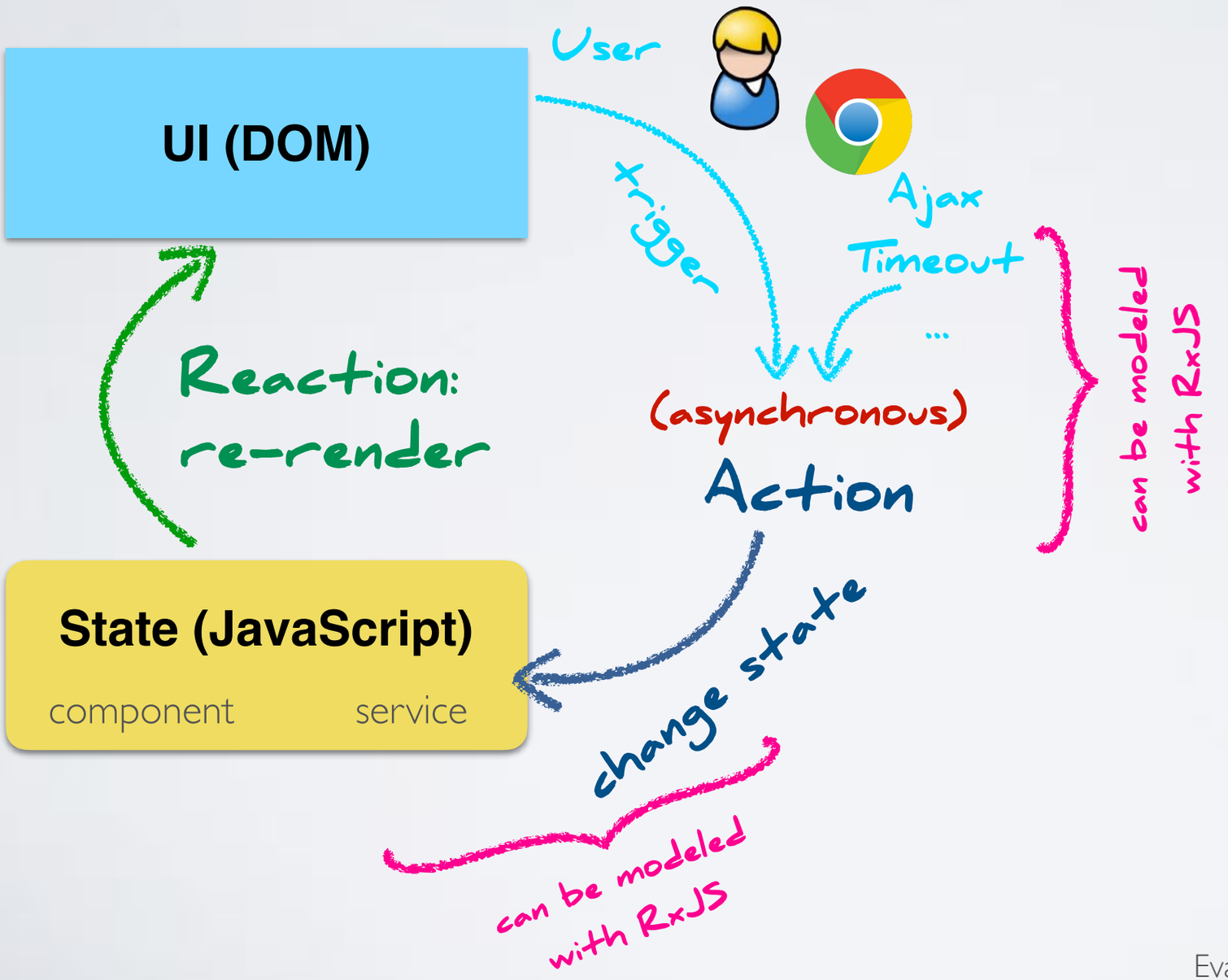
`setState` triggers a complete rendering of the component and its child components into the virtual DOM (this can be optimized).

Vue: The state notifies changes

Vue transparently converts properties to *getter/setters*. This allows dependency-tracking and change-notification.

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

Zone.js

Zone.js is a JavaScript library provided by the Angular project that patches many asynchronous browser APIs. Listeners can then be triggered when these APIs are executed.

Patched APIs (examples): `setTimeout`, `Promise`, `XMLHttpRequest`, `prompt` and DOM events.

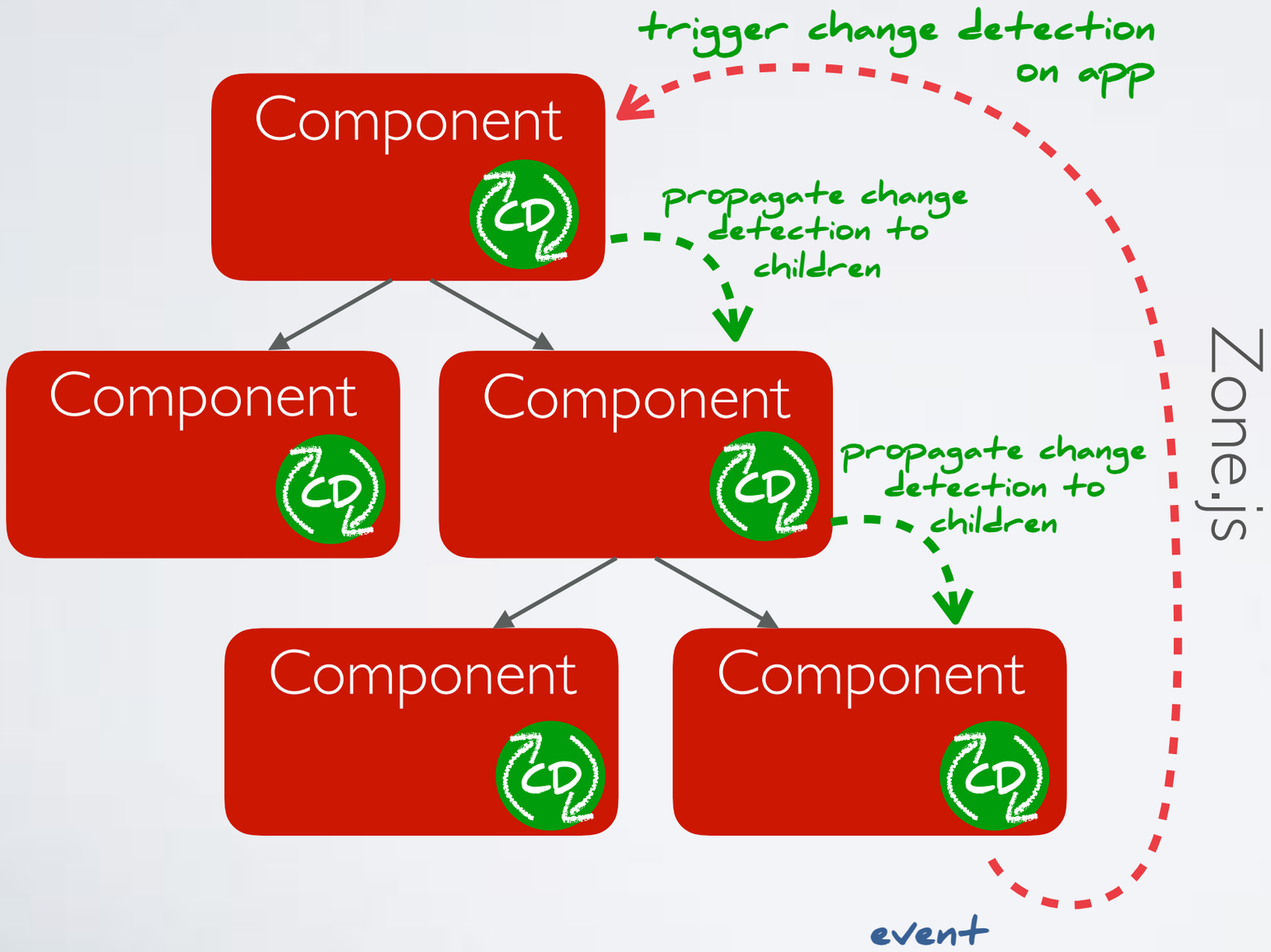
More details: https://github.com/angular/zone.js/blob/master/STANDARD-APIS.md

Angular relies on Zone.js to trigger automatic change detection. Angular is running inside the NgZone (a zone created via Zone.js). When async APIs are executed Angular gets notified when the execution has finished and triggers change detection.

https://github.com/angular/zone.js/

Default: Dirty Tracking

As default Angular detects changes by inspecting the state on all events that could possibly result in state changes.



Change detection is always triggered at the top of the component hirarchy and propagates down to each child.

Every value used in a template is inspected and compared to the previous value.

Checking all components on every possible event can be performance intense.

Unidirectional Data Flow

Angular enforces *unidirectional data flow* from top to bottom of the component tree.

- A child is not allowed to change the state of the parent once the parent changes has been processed.
- This prevents cycles in the change detection.
  (to prevent inconsistencies between state and ui and for better performance)
- In development mode Angular performs a check (a second change detection) to ensure that the component tree has not changed after change detection. If there are changes it throws a `ExpressionChangedAfterItHasBeenCheckedError`.

Change Detection Operations

During change detection Angular performs checks for each component which consists of the following operations performed in the specified order:

- update bound properties for all child components/directives
- call `ngOnInit`, `OnChanges` and `ngDoCheck` lifecycle hooks on all child components/directives
- update DOM for the current component
- run change detection for a child component
- call `ngAfterViewInit` lifecycle hook for all child components/directives

https://blog.angularindepth.com/everything-you-need-to-know-about-the-expressionchangedafterithasbeencheckederror-error-e3fd9ce7dbb4

Demos

Zone.js:

- "Patching" the browser by loading an Angular application manually.
- Zone.js not working with async/await.

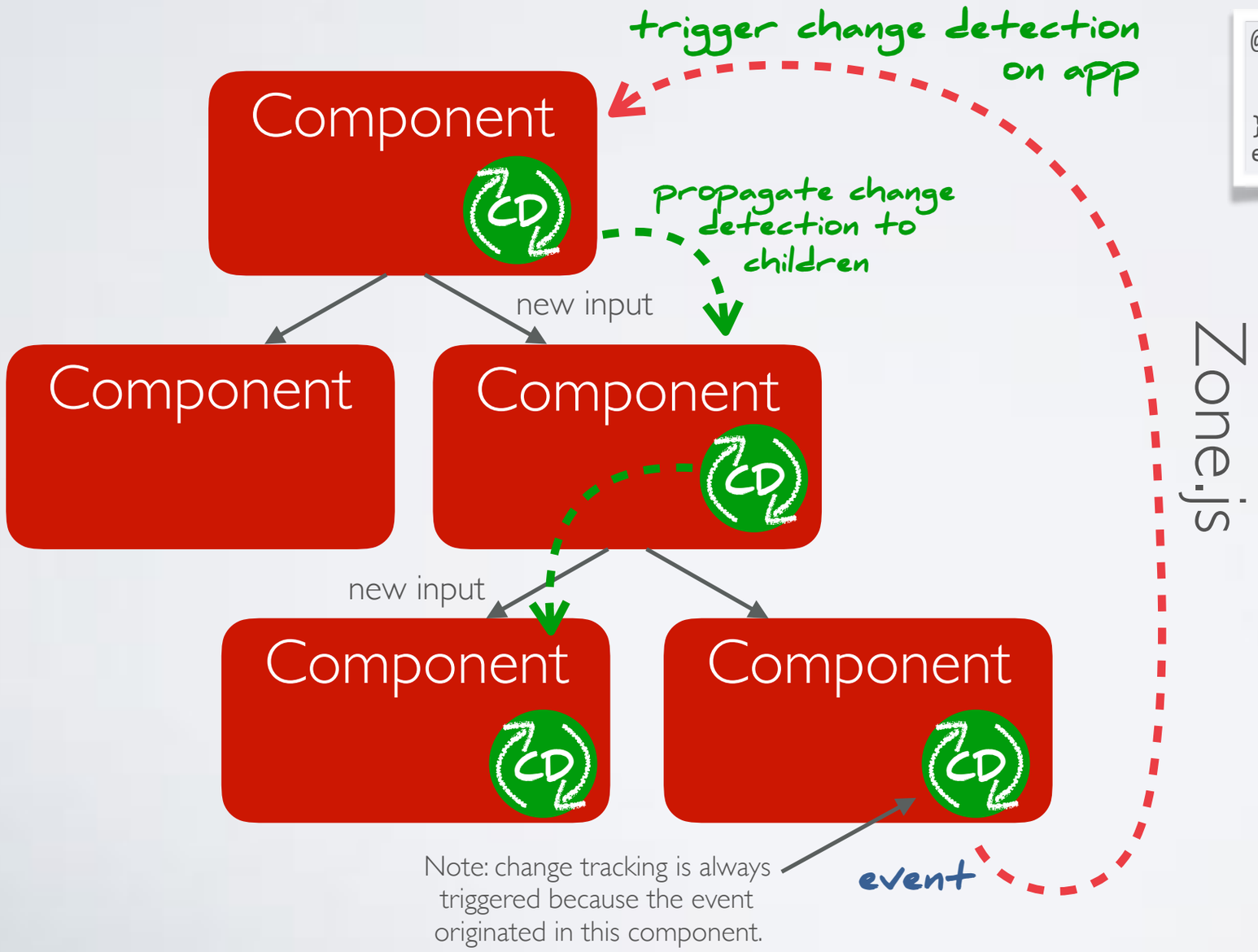
Change Tracking:

- Property access on each action.
- ExpressionChangedAfterItHasBeenCheckedError.

https://blog.angularindepth.com/everything-you-need-to-know-about-the-expressionchangedafterithasbeencheckederror-error-e3fd9ce7dbb4

Change Detection: OnPush

Angular can optimize change detection if we model the state according to a constraint: parents are only allowed to pass *immutable* or *observable* data to their children:




```
@Component({
  changeDetection: ChangeDetectionStrategy.OnPush,
  ...
})
export class MyComponent { ... }
```

Change detection is still triggered at the top of the component hirarchy and propagates down.
But change detection in children is only triggered if the child receives a new input (new object reference via `@Input()` or via Observable).

`OnPush` improves performance by limiting the work Angular performs for change detection.

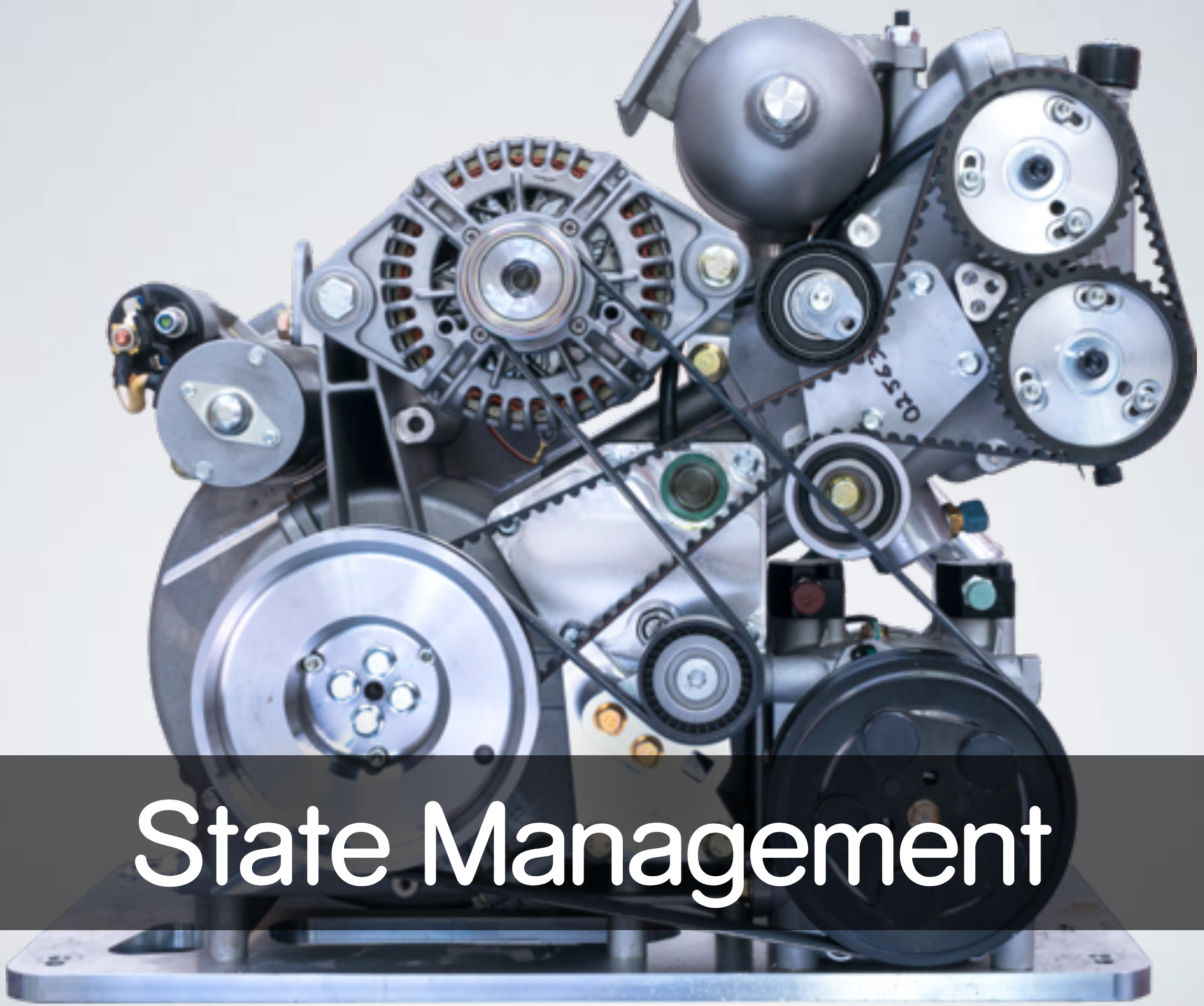
Change Detection: OnPush

With `OnPush` change detection is run when:

- An `@Input` property reference changes
- A DOM event of the component fires
- An `@Output` property/`EventEmitter` of the component fires
- An `async` pipe receives an event
- Change detection is manually invoked via `ChangeDetectionRef`

Benefits of `OnPush`:

- Performance: Change detection is on run on the component when above conditions are met, not on any change somewhere in the application.
- (Make errors visible fast: the UI of presentation components is not updated when they manipulate state that they receive via `@Input()`)

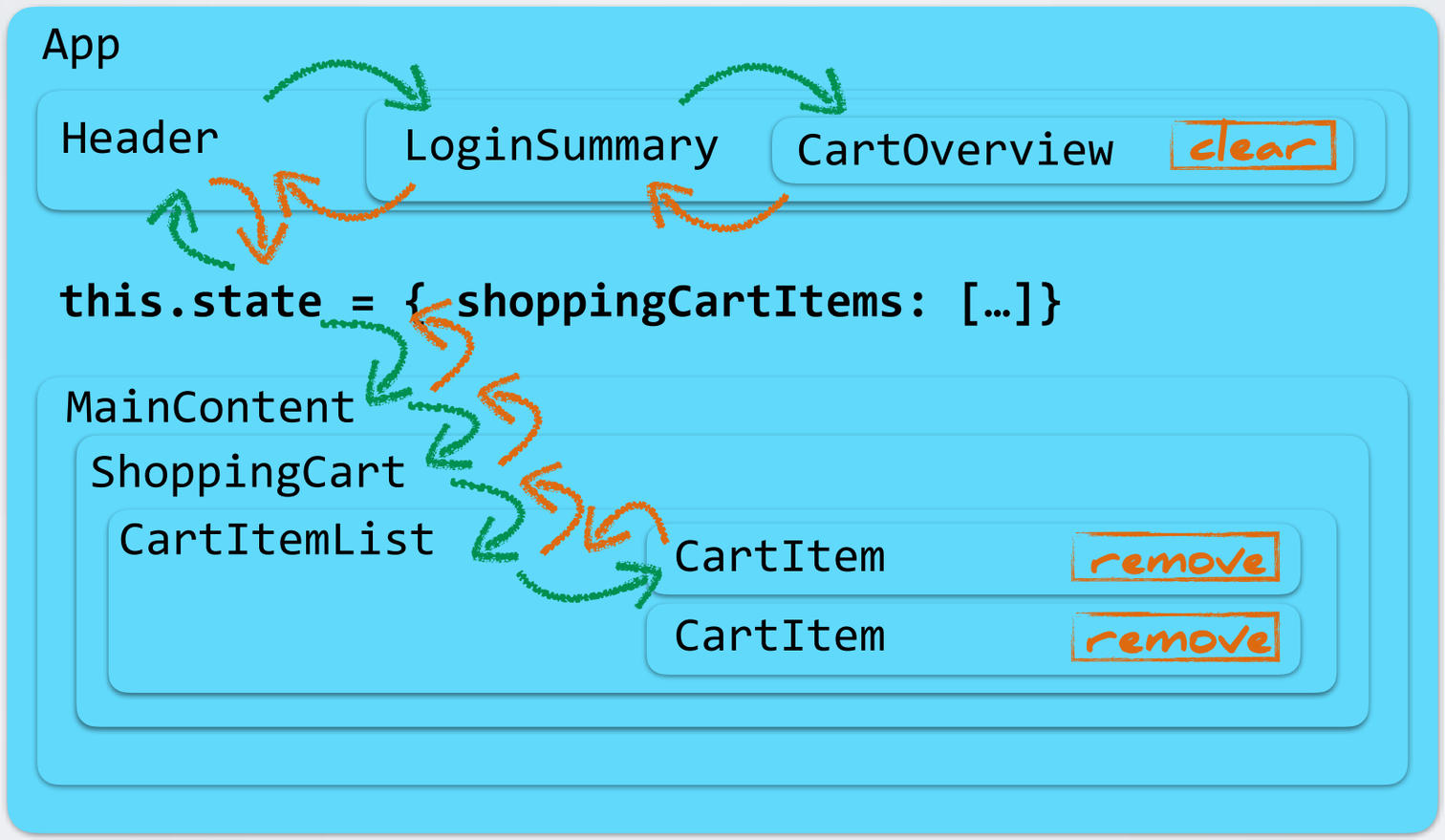


State Management

Component Architecture

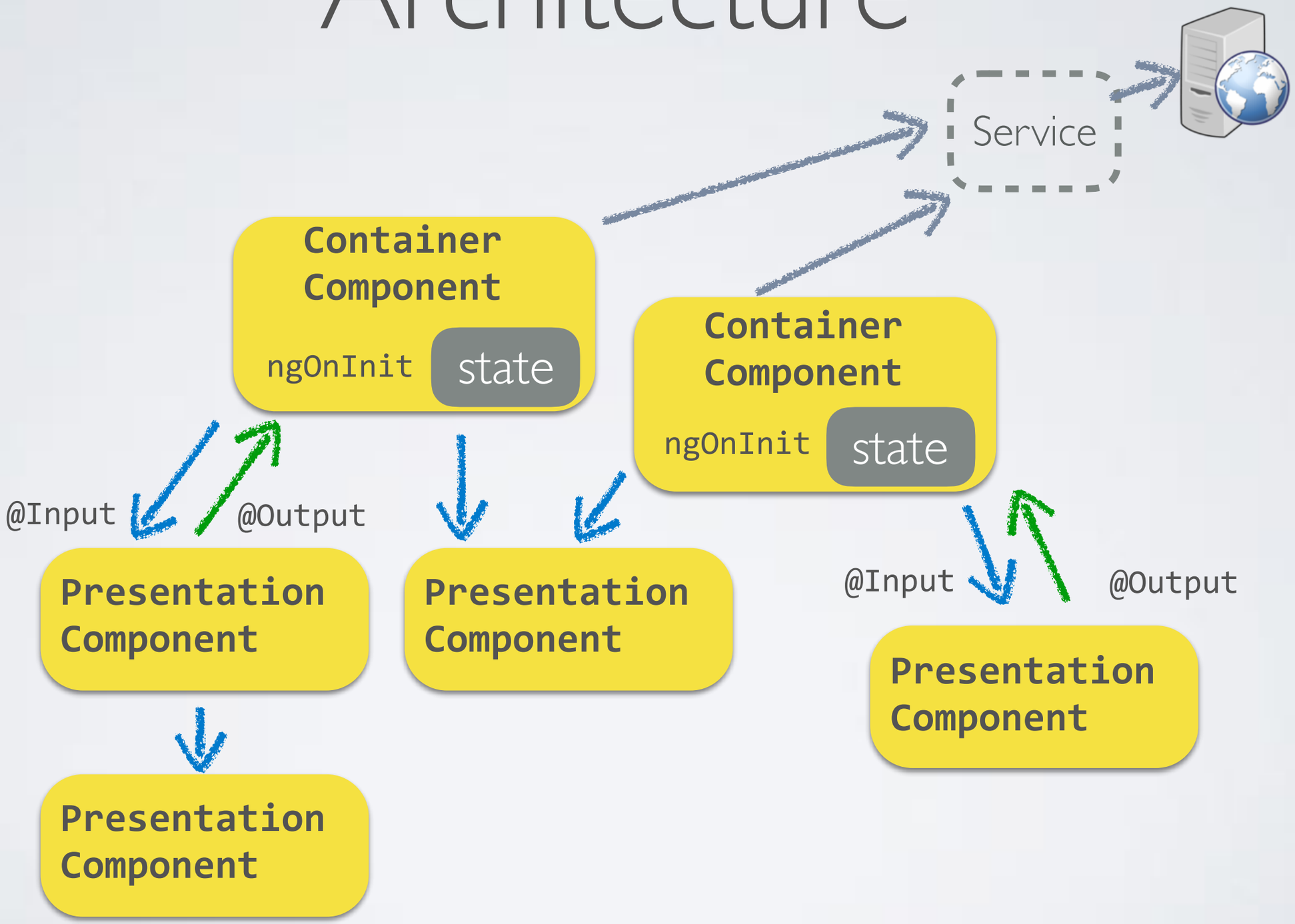
From the component architecture follows the pattern of "lifting state up": if several components need to reflect the same changing data, then the shared state should be liftet up to their closest common ancestor.

https://reactjs.org/docs/lifting-state-up.html



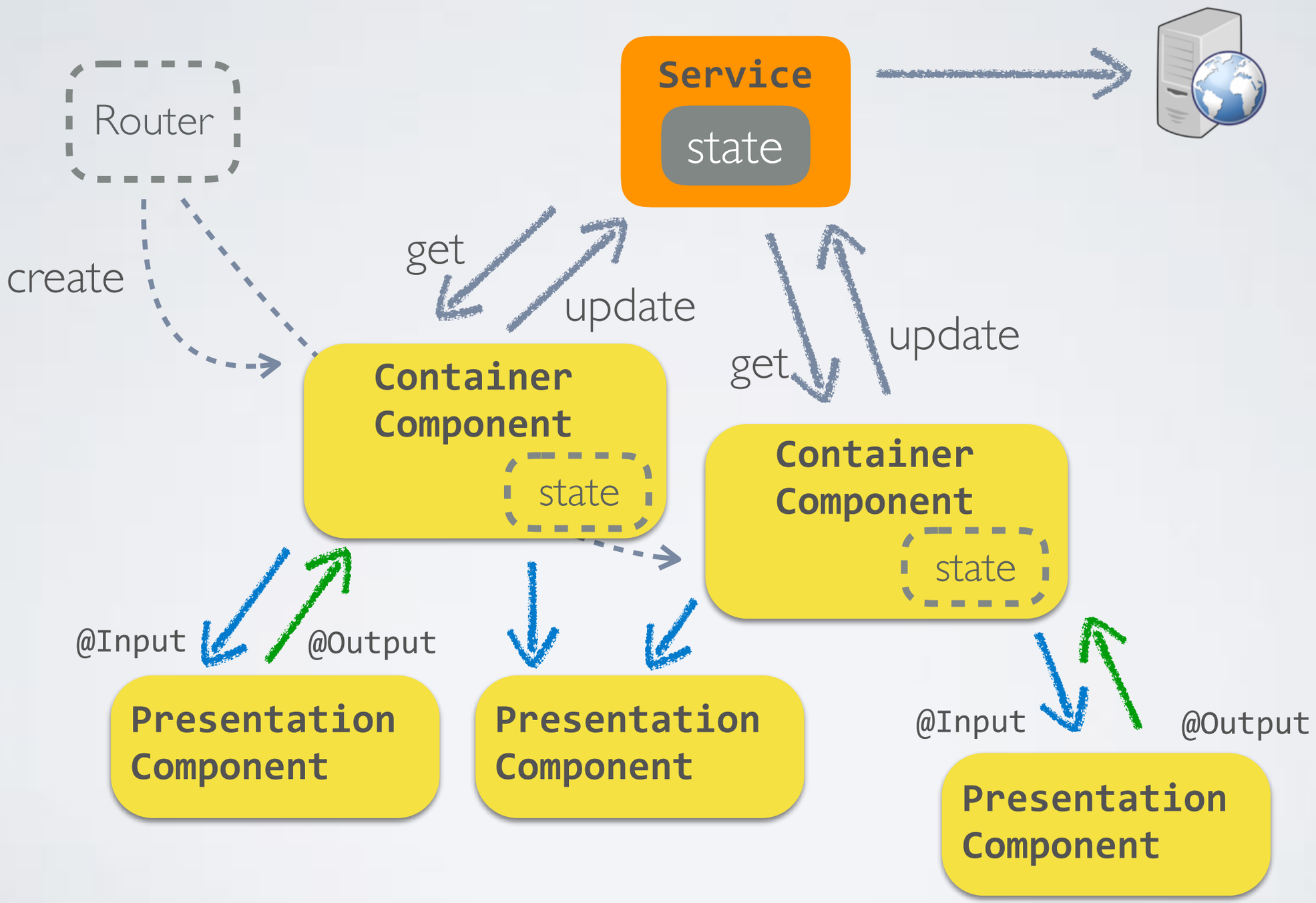
The problem of Angular: a component that is instantiated in a **`router-outlet`** can not receive **`@Input`** / **`@Output`** properties from its parent.

State Management: Component Architecture

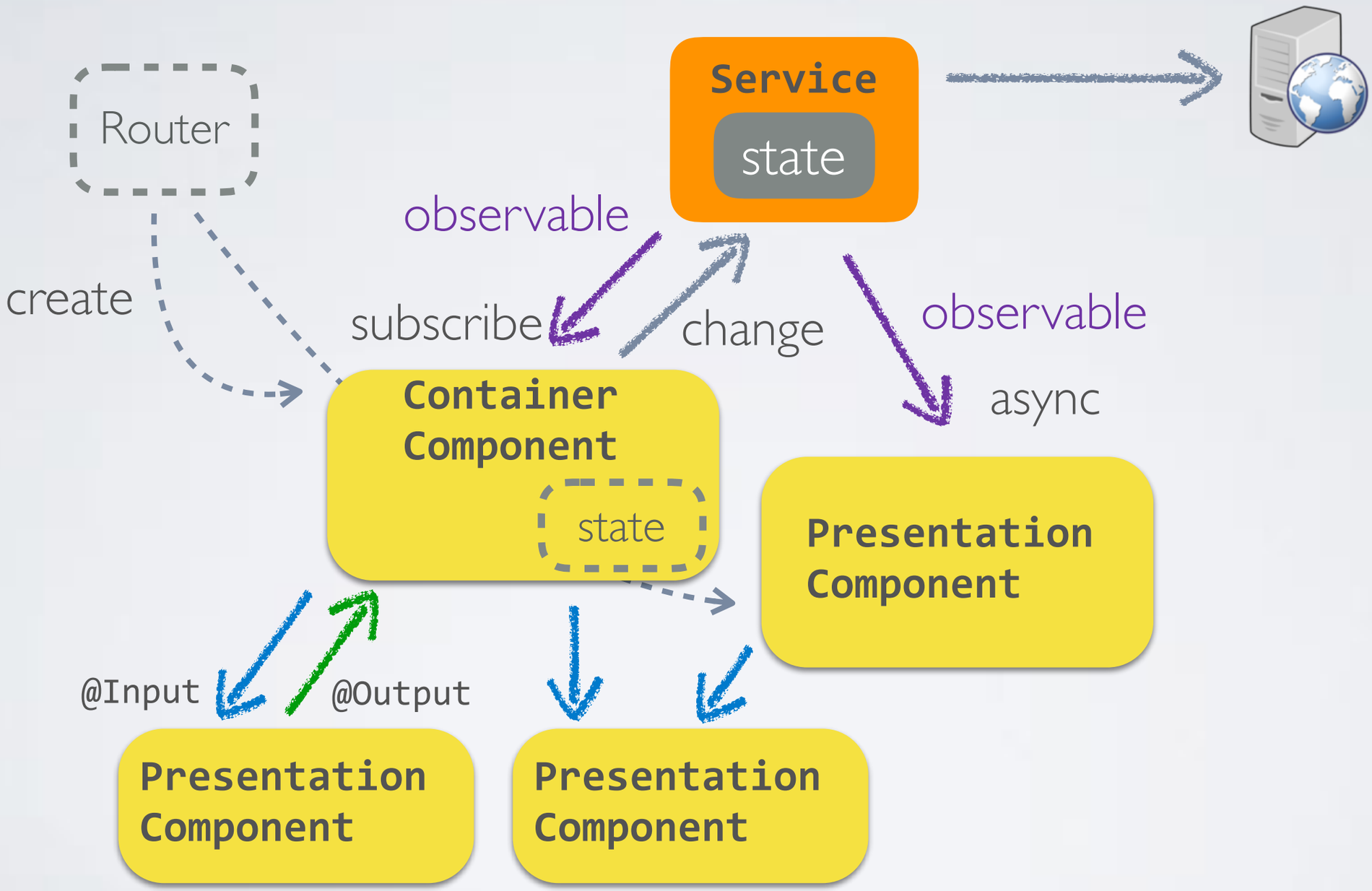


https://www.robinwieruch.de/learn-react-before-using-redux/

State Management: Passive Service



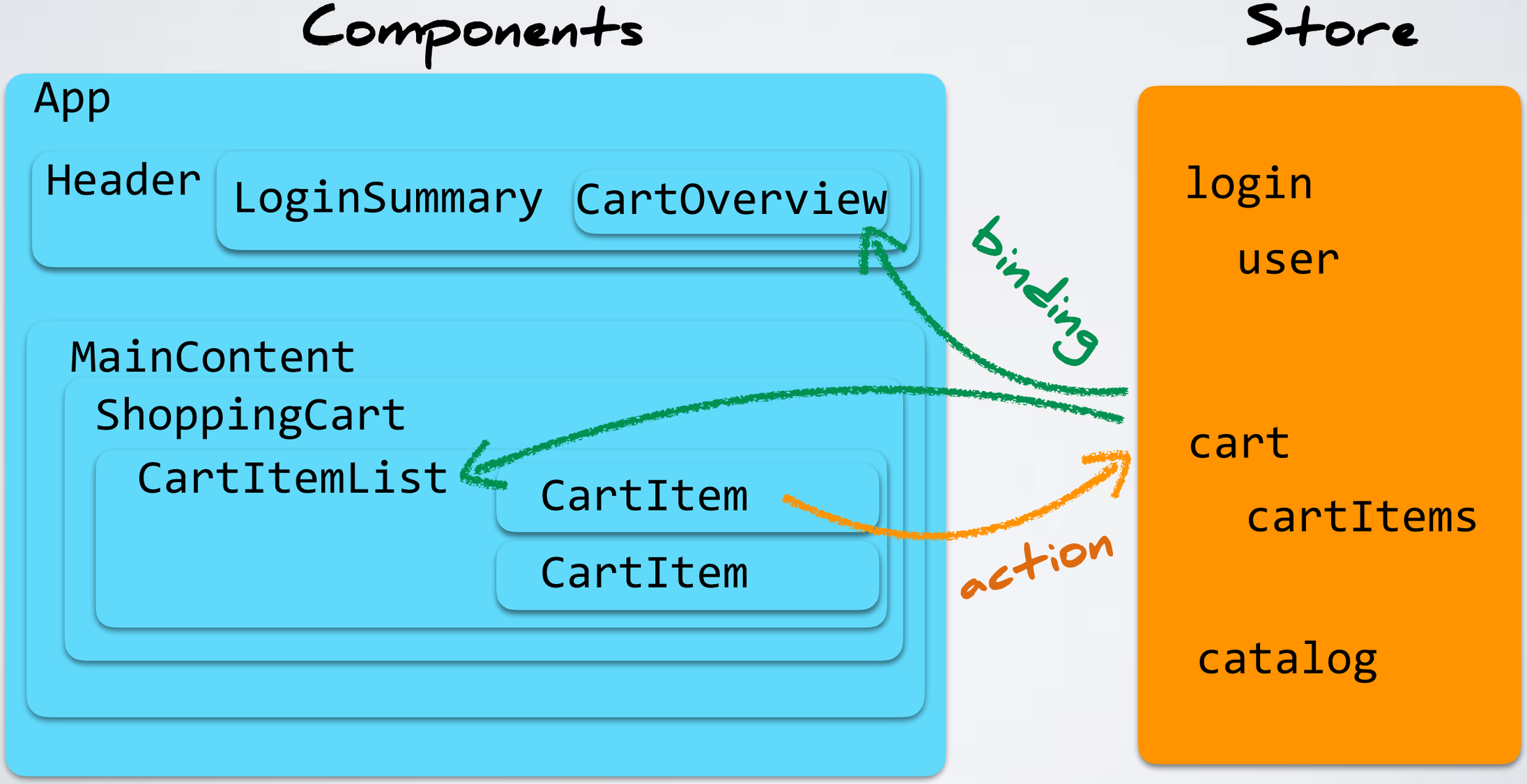
Reactive State Management: Observable Data Service



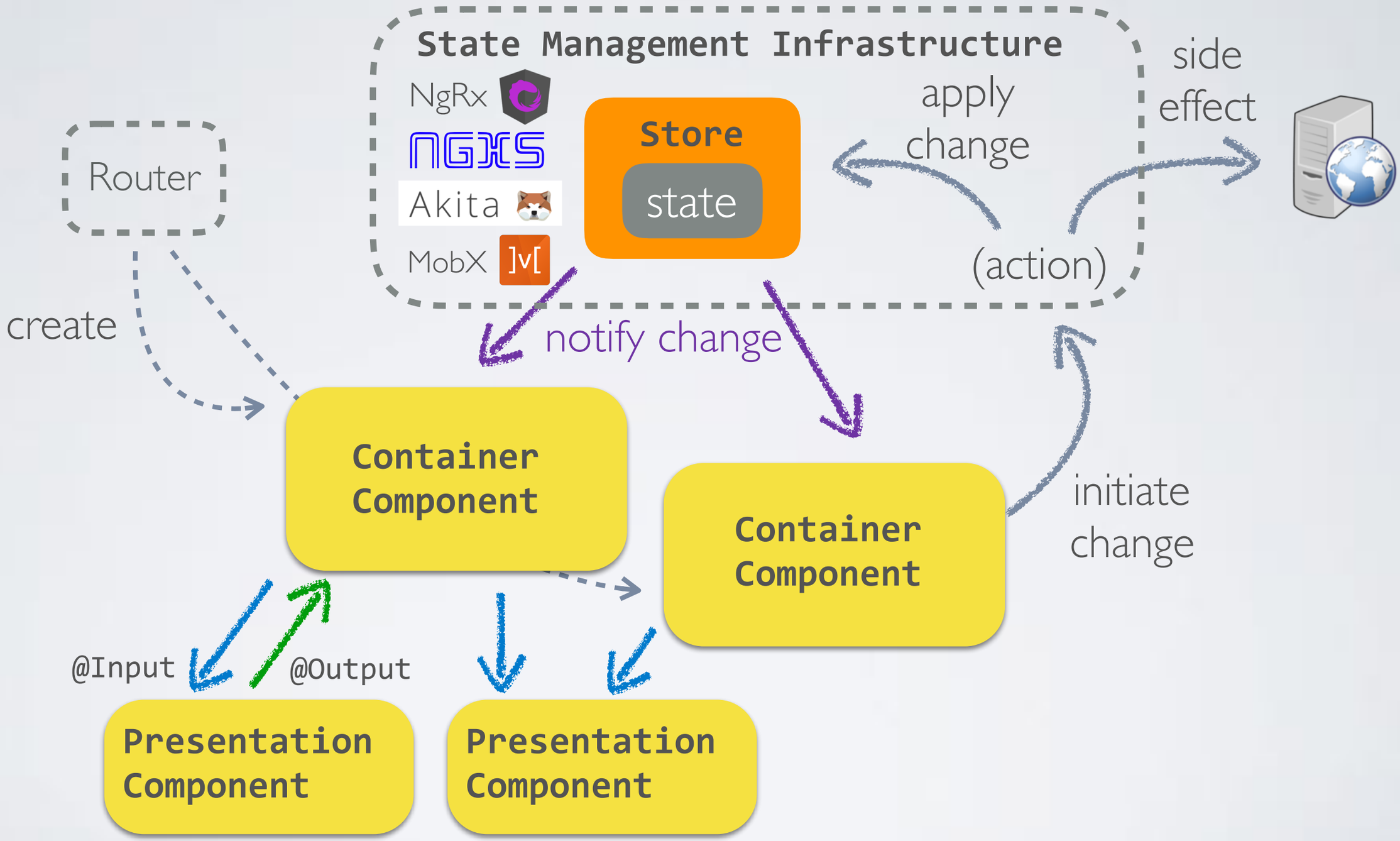
https://coryrylan.com/blog/angular-observable-data-services
https://blog.angular-university.io/how-to-build-angular2-apps-using-rxjs-observable-data-services-pitfalls-to-avoid/

Managing State with a State Container

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



State Management: State Container



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

Have fun with Angular!

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

Alternatives to Angular

No longer a De-Facto Standard!

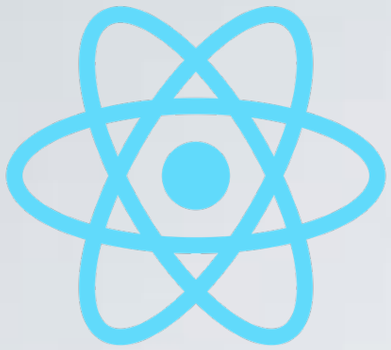
From about 2012 to 2014, AngularJS was almost a de-facto standard for the enterprise.
Today there are viable alternatives.

ThoughtWorks Technology Radar 2017:
"Most of our teams, however, still prefer React, Vue or Ember over Angular."

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

State of JavaScript Survey 2018:

https://2018.stateofjs.com



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

- Very popular / big ecosystem
- Project setup has similar complexity as with Angular
- Very elegant component model & composition patterns
- Interesting synergies: ReactNative, React for Windows …

Vue.js: https://vuejs.org

- Very popular / big ecosystem / driven by the community
- Project setup can be much simpler ("ES5 style")
- Simple solutions for many use-cases

- WebAssembly is on the horizon.
- It will probably trigger another wave of frameworks/innovations.

https://www.youtube.com/watch?v=Qe8UW5543-s

Others:

- Aurelia: http://aurelia.io/
- Ember: http://emberjs.com/
- Polymer: https://www.polymer-project.org/

http://todomvc.com/

Thank You!

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

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