Frontend-Development with Angular

Mail: jonas.bandi@ivorycode.com

Twitter: @jbandi

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 a JavaScript object.
There is no need for deserialization.
Note: The payload consists of plain JavaScript objects, not class instances!

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

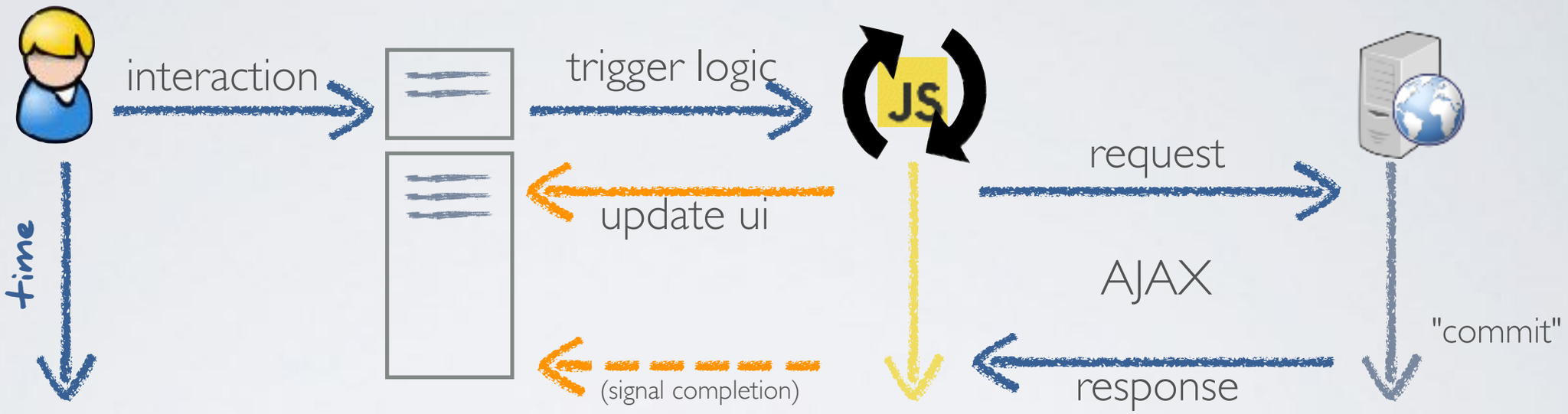
ToDo[]

EXERCISES

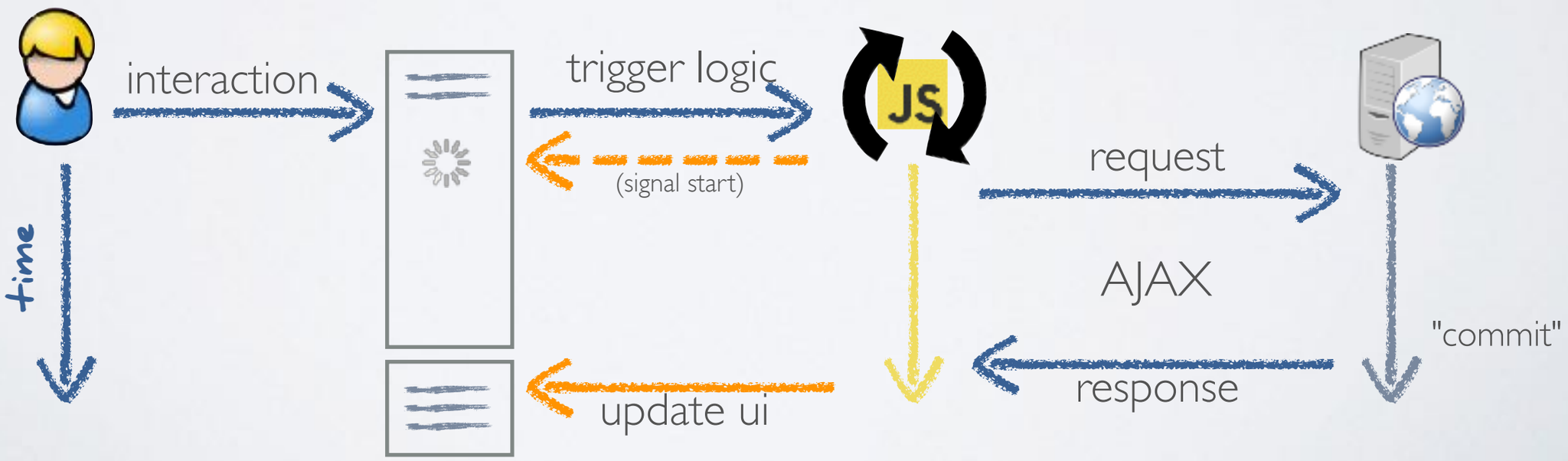
Exercise 7: ToDo – Backend Access

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
https://blog.angularindepth.com/top-10-ways-to-use-interceptors-in-angular-db450f8a62d6

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

JavaScript object

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

Type-Safe Backend Access

Java to TypeScript Generator

https://github.com/vojtechhabarta/typescript-generator

```java
public class Person {
    public String name;
    public int age;
    public boolean hasChildren;
    public List<String> tags;
    public Map<String, String> emails;
}
```

```typescript
interface Person {
    name: string;
    age: number;
    hasChildren: boolean;
    tags: string[];
    emails: { [index: string]: string };
}
```

Very easy to configure in a Maven or Gradle build.
Can detect DTOs automatically for JAX-RS or DTOs can be specified via pattern.
Somhow configurable (mapping, naming ...)
Limitation: One single generated file / module containing all the types.

Alternative: https://github.com/swagger-api/swagger-codegen

```
swagger-codegen generate -i http://petstore.swagger.io/v2/swagger.json -l typescript-angular -o .
```

Generated code is very different for: `typescript-angular`, `typescript-fetch`, `typescript-node` ...

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)!

Reasons for multiple NgModules

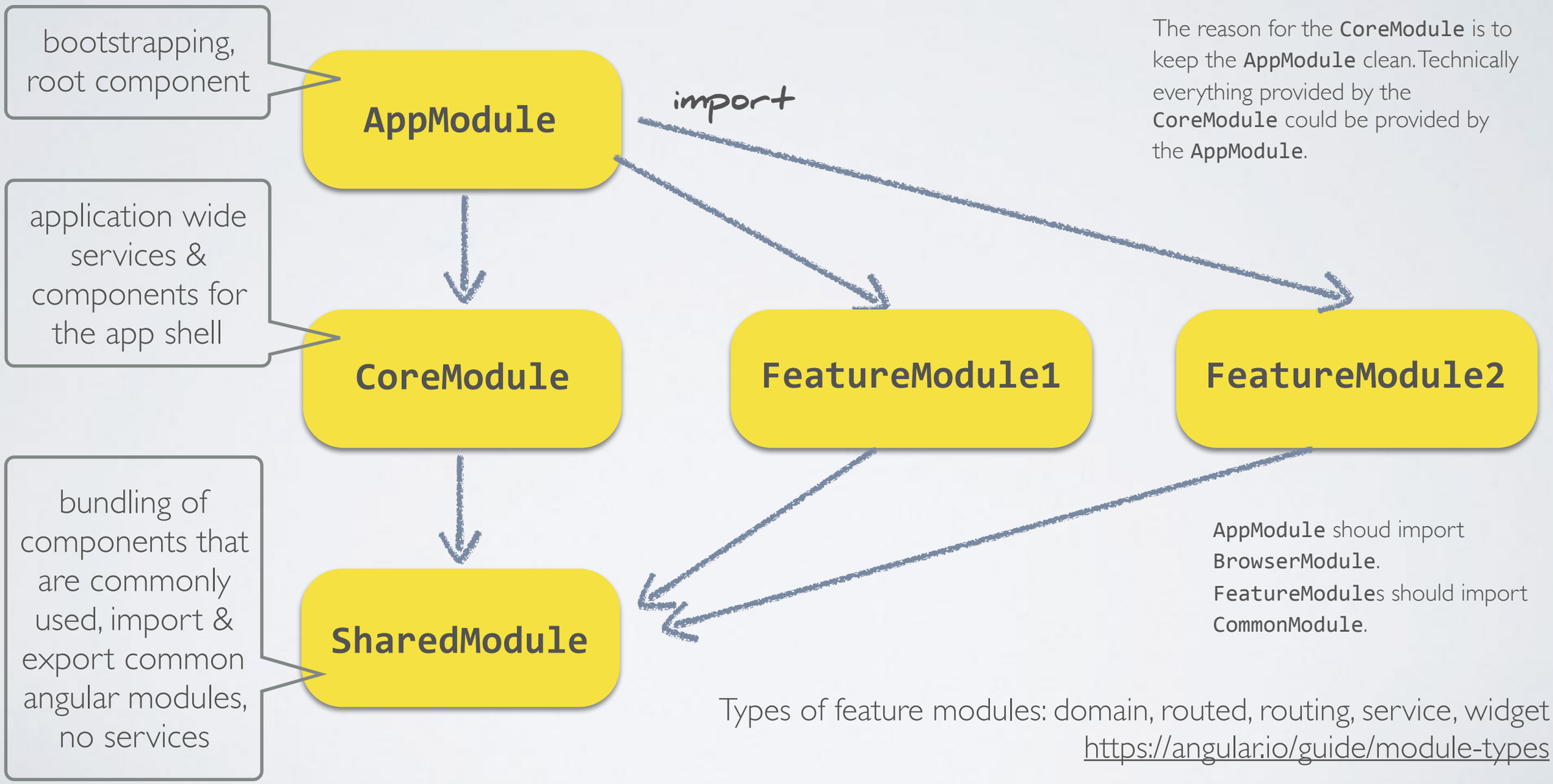
- Encapsulation:
  A NgModule can have "internal" and exported components, directives & pipes. Only exported constructs can be used from other NgModules.
  Note: Services are not encapusalted and always available "globally".

- 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`:

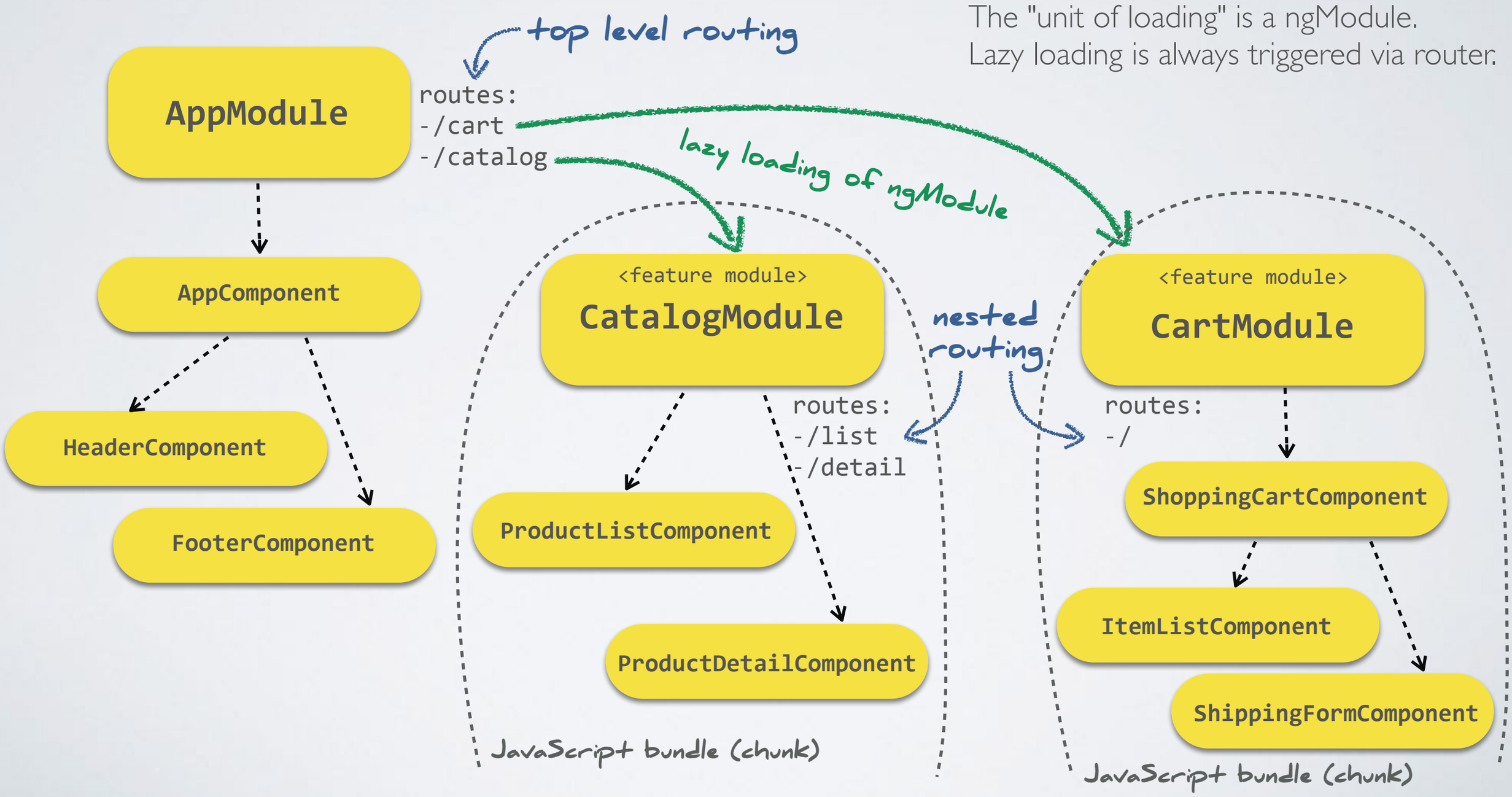
- https://angular.io/guide/singleton-services
- https://angular.io/guide/ngmodule-faq#what-is-the-forroot-method

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.

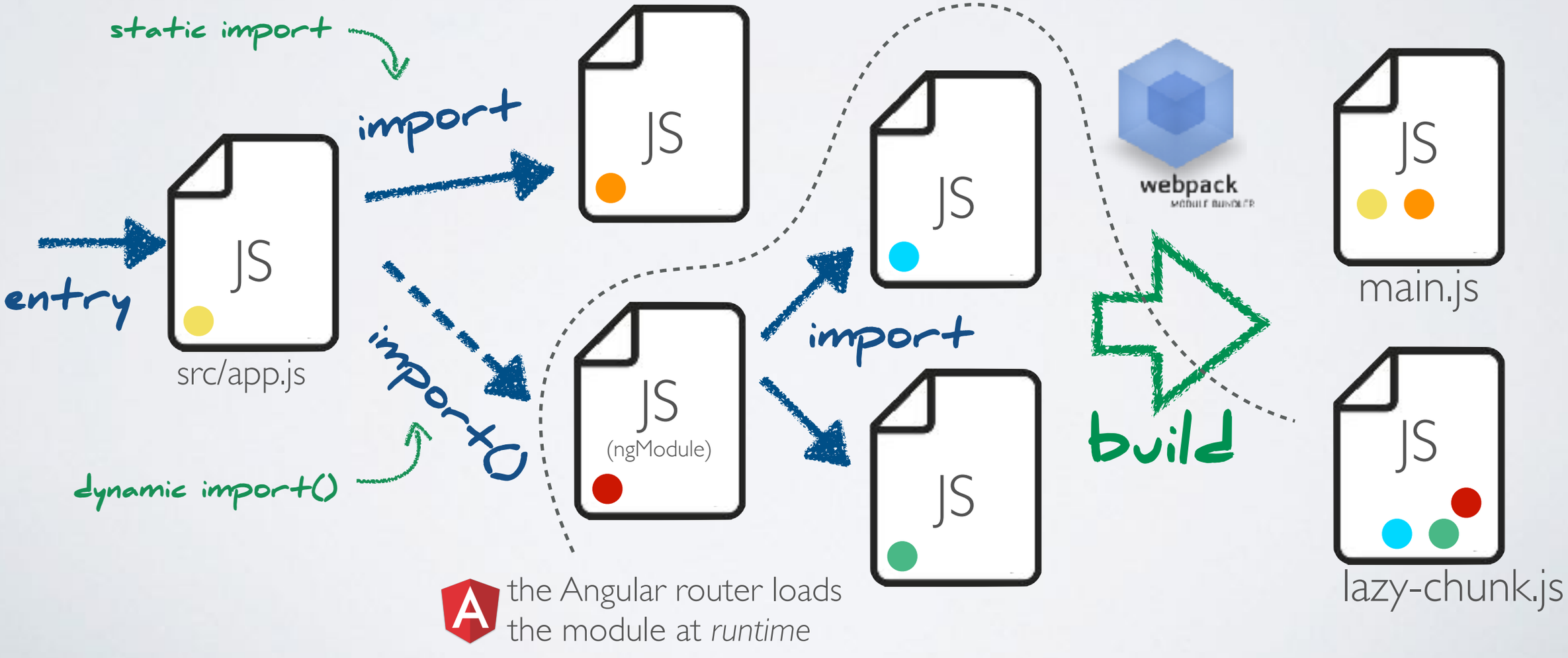
The "unit of loading" is a ngModule.
Lazy loading is always triggered via router.



Lazy Loading

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

const routes: Routes = [
  {path: '', pathMatch: 'full', redirectTo: 'home'},
  {path: 'home', component: HomeComponent},
  {path: 'catalog', loadChildren: () => import('./catalog/catalog.module')
                                          .then(m => m.CatalogModule)},
];
```

using "dynamic import"
(JavaScript standard)



Lazy Loading Syntax

Lazy loading syntax in Angular 8 and later:

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

const routes: Routes = [
  {path: '', pathMatch: 'full', redirectTo: 'home'},
  {path: 'home', component: HomeComponent},
  {path: 'catalog', loadChildren: () => import('./catalog/catalog.module')
                                          .then(m => m.CatalogModule)},
];
```

using "dynamic import" (JavaScript standard)

Lazy loading syntax before Angular 8:

```
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 string for webpack to create a lazy chunk

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

The Angular CLI build uses a "magic string" to declare lazy loadable modules. It uses a custom webpack loader to create the separate chunks.

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
- hero-loader: https://www.npmjs.com/package/@herodevs/hero-loader

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

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: () => import('./todos/components/todo-screen/todo-screen.module').then(m => m.ToDoScreenModule) },
  { path: 'done', loadChildren: () => import('./todos/components/done-screen/done-todos.module').then(m => m.DoneScreenModule) },
  { path: 'details', loadChildren: () => import('./todos/components/detail-screen/detail-screen.module').then(m => m.DetailScreenModule) }
];
```

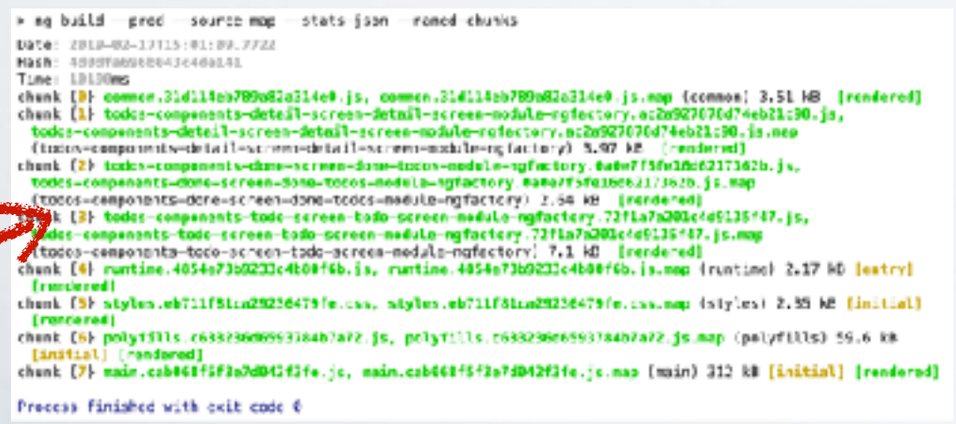
good: configuring lazy loading

`app.module.ts`

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

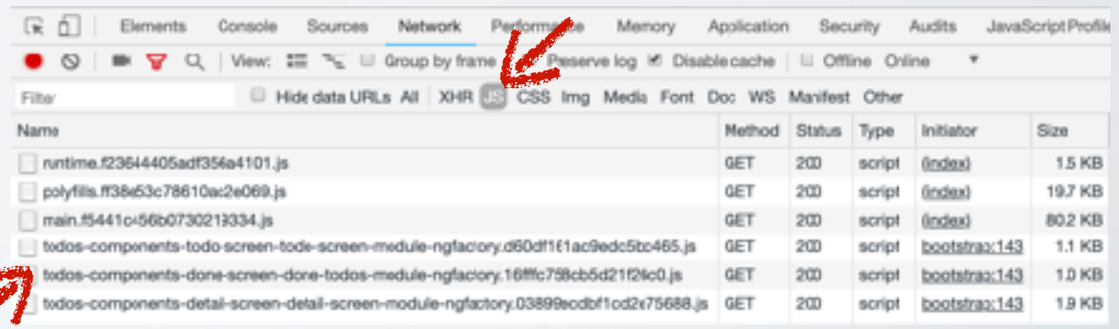
bad: referencing the lazy modules from the app module

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



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!

"Real-World" Setup with the CLI

Single App with several NgModules

Create app with root module:

```
ng new --routing --skip-tests --inline-template --inline-style --style scss starter-app -p aw
```

Create core modules:

```
ng g m core --routing
ng g c core/core-component
ng g s core/core-service1
```

Create feature modules:

```
ng g m home --routing
ng g c home/home-shell
ng g c home/welcome
ng g s home/home-service
```

```
ng g m catalog --routing
ng g c catalog/catalog-shell
```

```
ng g m cart --routing
ng g c cart/cart-shell
```

Create shared modules:

```
ng g m shared1
ng g c shared1/shared1-component1
ng g c shared1/shared1-component2
ng g s shared1/shared1-service1
```

```
ng g m shared2
ng g c shared2/shared2-component1
ng g c shared2/shared2-component2
ng g s shared2/shared2-service1
```

"Real-World" Setup with the CLI

Single App with several NgModules

app module routing:

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

feature module routings:

```
const routes: Routes = [
  {path: '', component: HomeShellComponent},
  {path: 'welcome', component: WelcomeComponent},
];
```

```
const routes: Routes = [
  {path: '', pathMatch: 'full', redirectTo: 'list'},
  {path: 'list', component: CatalogShellComponent},
];
```

Angular CLI Build Options

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

Optimizing an Angular build a complex problem with many paramters and it is hardly documented!

Angular CLI bundling options: `--prod`

```
--common-chunk
--vendor-chunk
--build-optimizer
--named-chunks
--optimization
--source-map
--stats-json
```

Some bundling options are also configured in `angular.json`

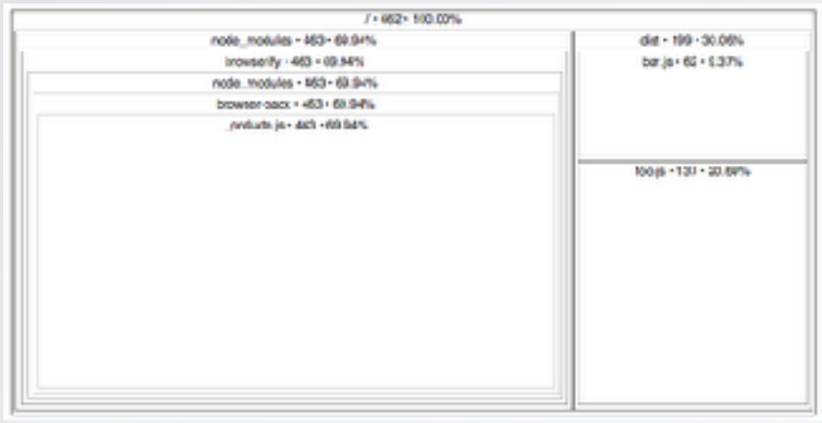
Analyzing Bundles

Angular CLI bundling options:

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

```
--prod
--common-chunk
--vendor-chunk
--build-optimizer
--named-chunks
```

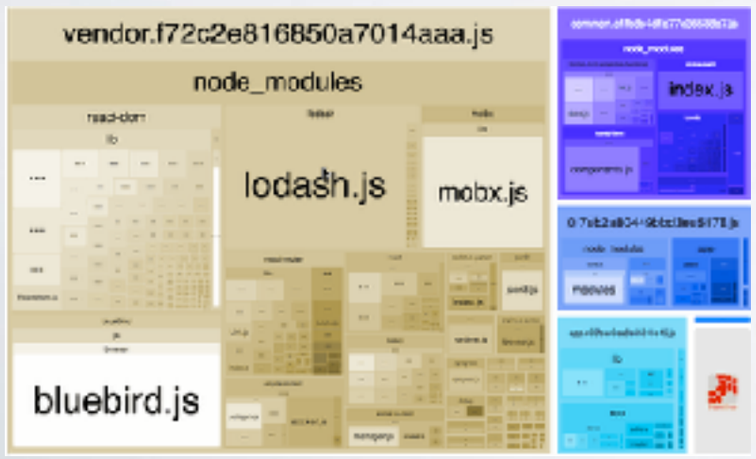
source-map-explorer:
https://www.npmjs.com/package/source-map-explorer




```
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: Performance Budgets

JavaScript bundle size does matter!

https://medium.com/@addyosmani/the-cost-of-javascript-in-2018-7d8950fbb5d4

Keep an eye on your bundle size, reducing bundle size late in the project can be a big pain.

angular.json

```
{
  ...
"configurations": {
    "production": {
          ...
          "budgets": [
            {
              "type": "initial",,
              "maximumWarning": "2mb",
              "maximumError": "5mb"
            }
          ]
```

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.
Starting from Angular CLI 7 a budget is scaffolded.

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

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
Currently not possibe: Budget for lazy chunks: https://github.com/angular/angular-cli/issues/11019

Potential for Confusion

In Angular we have different hierarchical constructs:

Component-Tree	Defined by the programmer. Child components are instantiated in templates
NgModule Hierarchy	Defined by the programmer: - eager loaded modules: Root NgModule imports other NgModules. - lazy loaded modules are referenced in the routing configuration
Provider Hierarchy	Derived by Angular from the component tree and the lazy loaded modules.
Bundle/Chunk Hierarchy	Primarily definded by lazy loaded modules. Webpack extracts common chunks independently of the NgModule hierarchy.

Typical Bundling Problems

Problematic:

importing full rxjs

```
import * as rxjs from 'rxjs';
console.log(rxjs);
```

importing wrong constructs from rxjs

```
import {filter} from 'rxjs/internal/operators';
console.log(filter);
```

importing full momentjs

```
import * as moment from 'moment';
console.log(moment);
```

importing full component libraries

```
import * as primeng from 'primeng/primeng';
console.log(primeng);
```

Better:

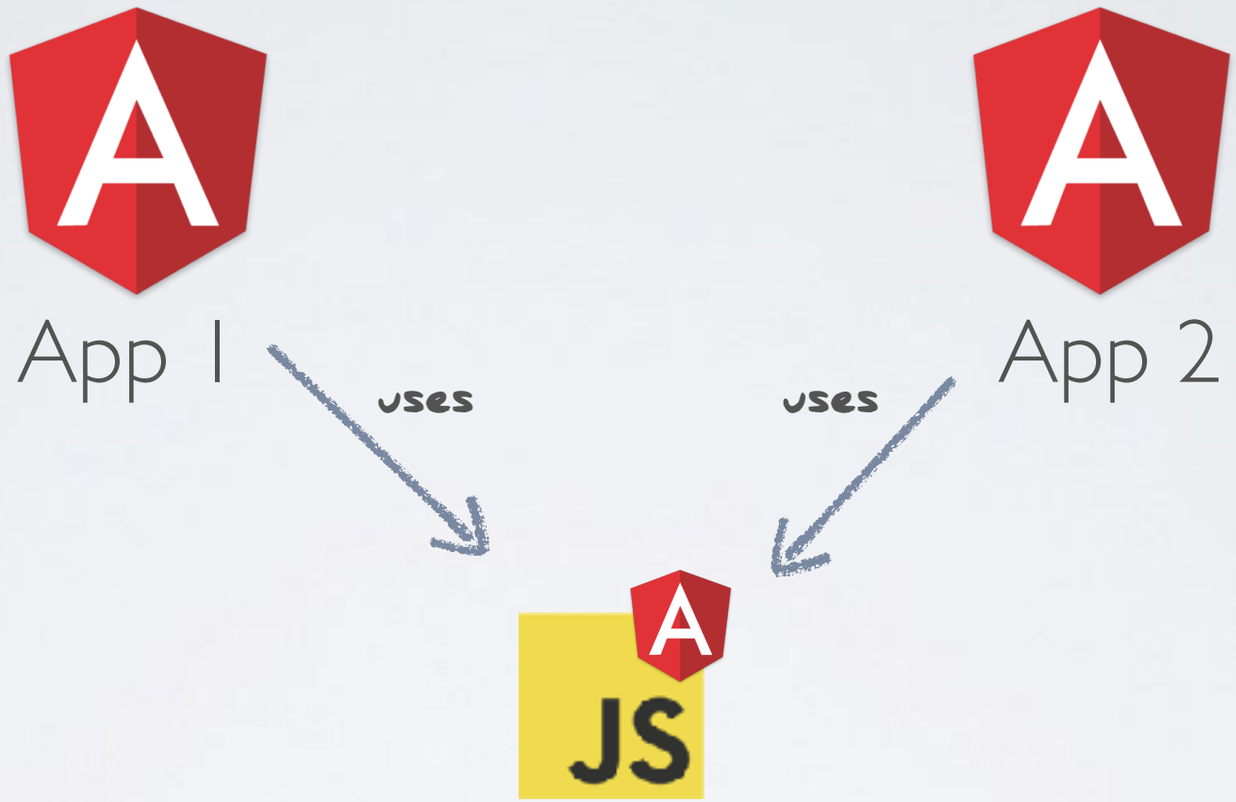
```
import {map} from 'rxjs/operators'
console.log(map);
```

```
import {filter} from 'rxjs/operators';
console.log(filter);
```

use `date-fns` or extend the webpack config with `ngx-build-plus`

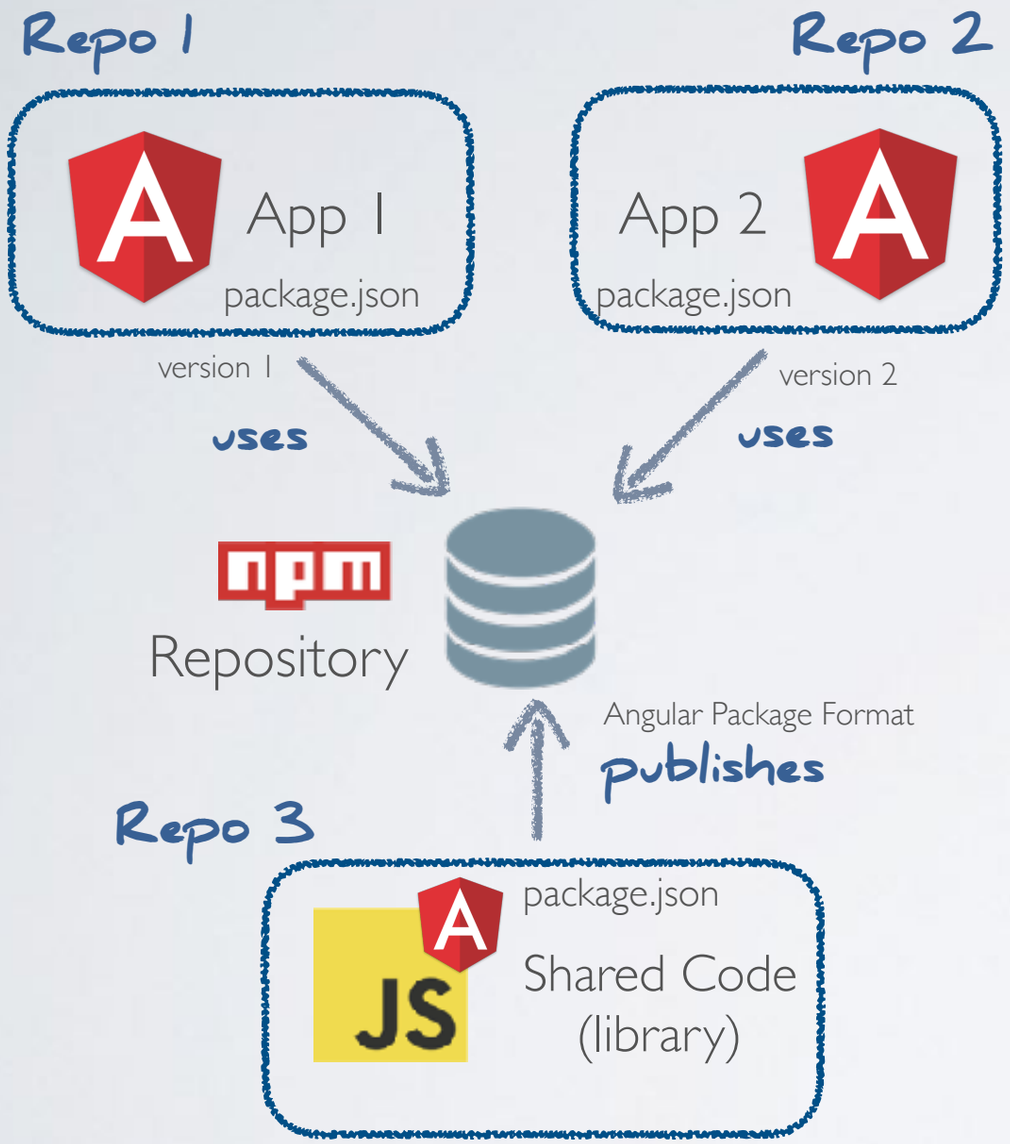
```
import {MatListModule, MatSidenavModule}
            from 'primeng/primeng';
```

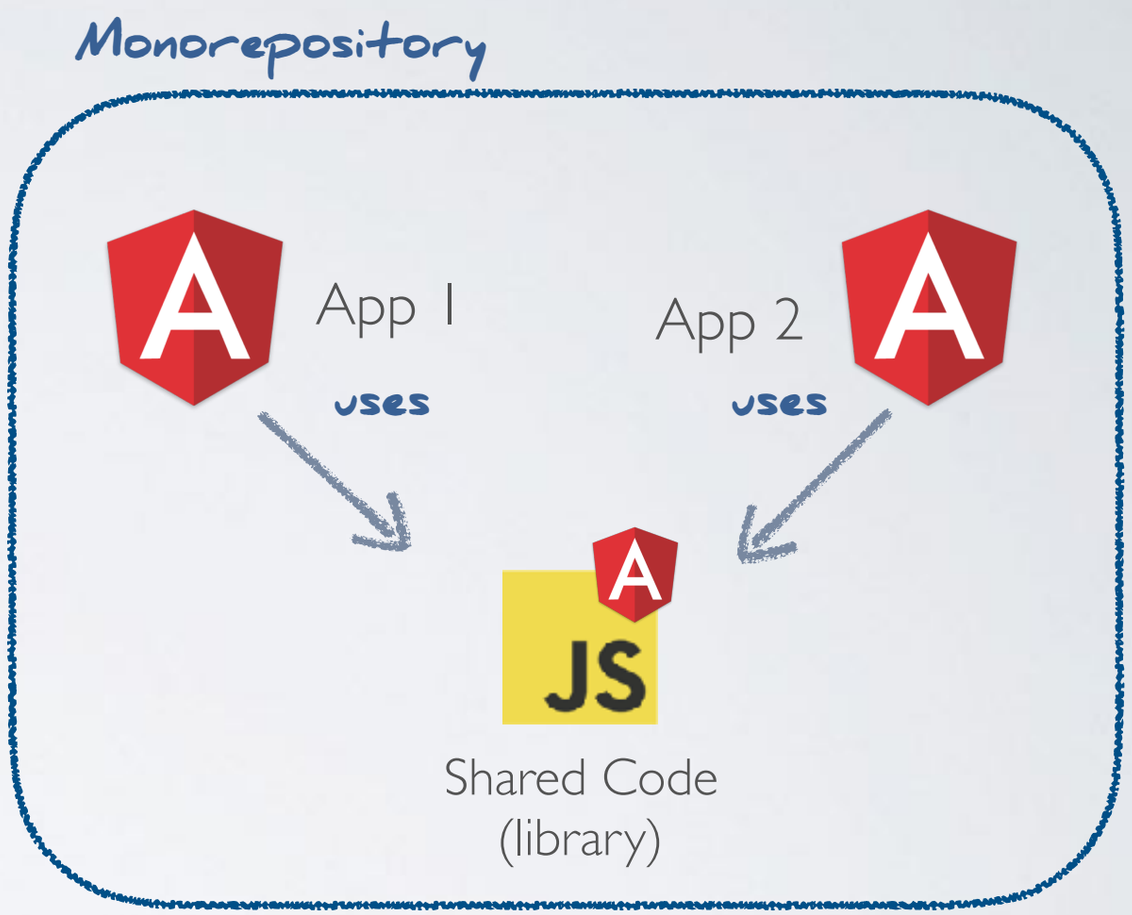
Apps & Libraries



Questions:
How many deployment artifacts (javascript bundles)?
How many development artifacts (javascript packages)?
What are the release cycles of the different artifacts? Are they coupled?
Source code organization?
Versioning schema?

Apps & Libraries





Note: Angular CLI 6 does support libraries ...

Nrwl Extensions for Angular
https://nrwl.io/nx

Lerna: A tool for managing JavaScript projects with multiple packages.
https://nrwl.io/nx

Yarn workspaces
https://yarnpkg.com/en/docs/workspaces

Packagr: https://github.com/dherges/ng-packagr
Generator: https://github.com/jvandemo/generator-angular2-library

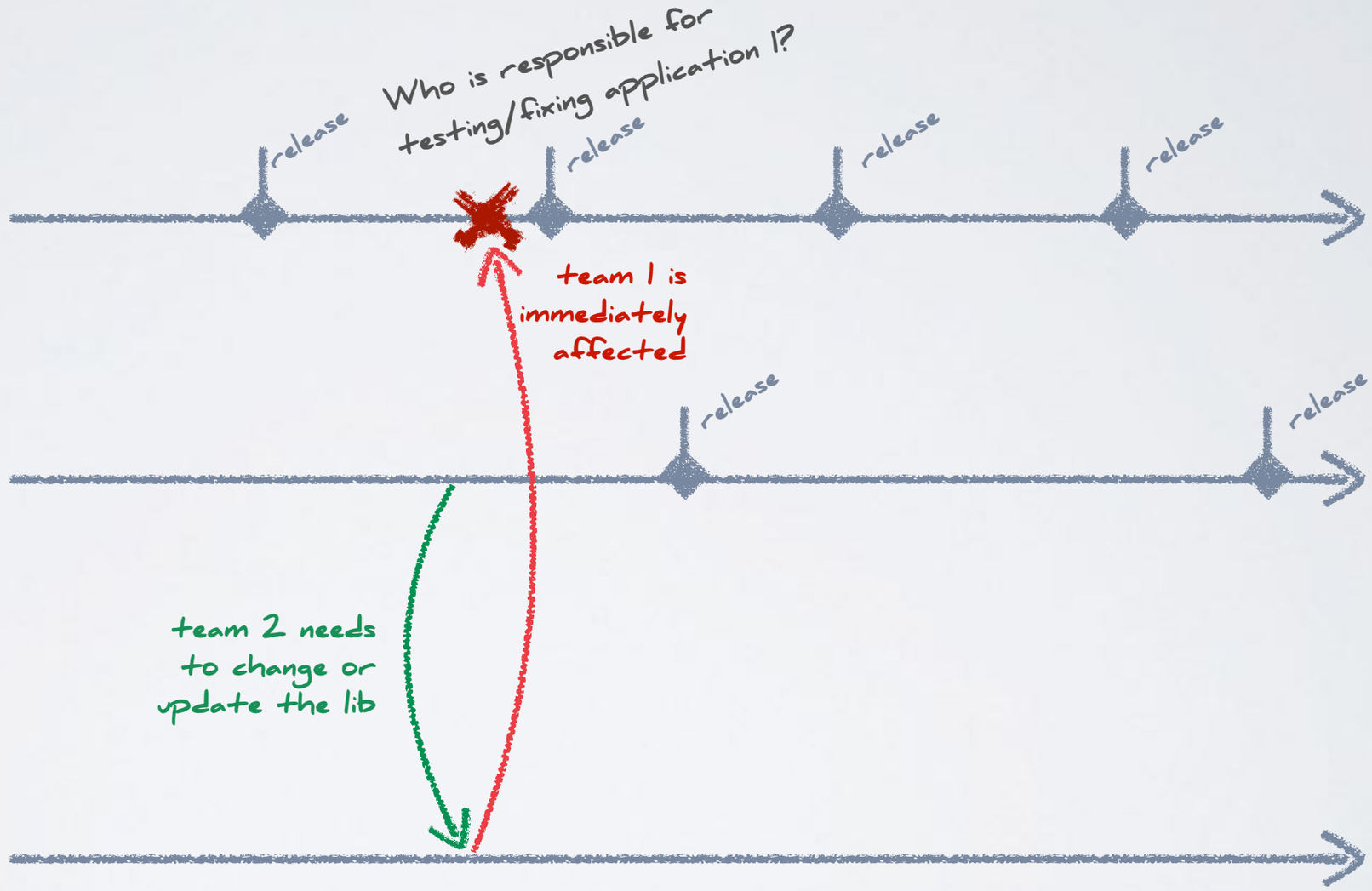
Implications of the Mono-Repo

"Think about the granularity of development & deployment artifacts"

Application 1
- implemented by team 1
- driven by org-unit 1

Application 2
- implemented by team 2
- driven by org-unit 2

shared lib
- inhouse lib
- 3rd party lib
  (ie. Angular, React)



The consequence is coupling between App1 and App 2:
- teams have to coordinate changes to the shared lib.
- development & release schedules of applications are coupled

Multi-App Setup with the CLI

Workspace: several apps with shared libraries

Create app with root module:

```
ng new --createApplication=false --routing --skip-tests --inline-template --inline-style
                                                          --style scss -p aw multi-project
```

```
ng g application --routing --skip-tests --inline-template --inline-style --style scss first-app -p fa
```

```
ng g application --routing --skip-tests --inline-template --inline-style --style scss second-app -p sa
```

```
rm -rf src
rm -rf e2e
```

```
ng serve first-app
ng serve second-app --port=4201
```

```
ng g library first-lib
```

```
ng g c --project first-lib first-shared1
```

```
ng build first-lib
```

In a project module:

```
import { FirstLibModule } from 'first-lib';
```

In a project component template:

```
<lib-first-lib></lib-first-lib>
```

Nx: An extension to the Angular CLI

Nrwl Extensions for Angular
https://nrwl.io/nx

- an extension for the the Angular CLI implementing the monorepo-style development
- a collection of runtime libraries, linters, and code generators

Routing Part 2

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.

Note: Its not necessary to unsubscribe in this specifc case, since Angular destroys the component and the correspinding **`ActivatedRoute`** together.

https://angular.io/guide/router#activated-route

Programmatic Navigation

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

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

Navigate with a *link parameters array*:

(a *link parameter array* can also be passed to the **`routerLink`** directive)

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

```
this.router.navigate(['/details', 3, 'person', 2]);
```
→ `/details/3/person/2`

```
this.router.navigate(['/details'],
                {queryParams: {id:5, foo:'foo1'}});
```
→ `/details?id=5&foo='foo'`

```
this.router.navigate(['/details', {id:5, foo:'foo1'}]);
```
→ `/details;id=5;foo='foo'`

Note: this is called *Matrix URL* notation and can be used to pass optional parameters...

…or via Url:

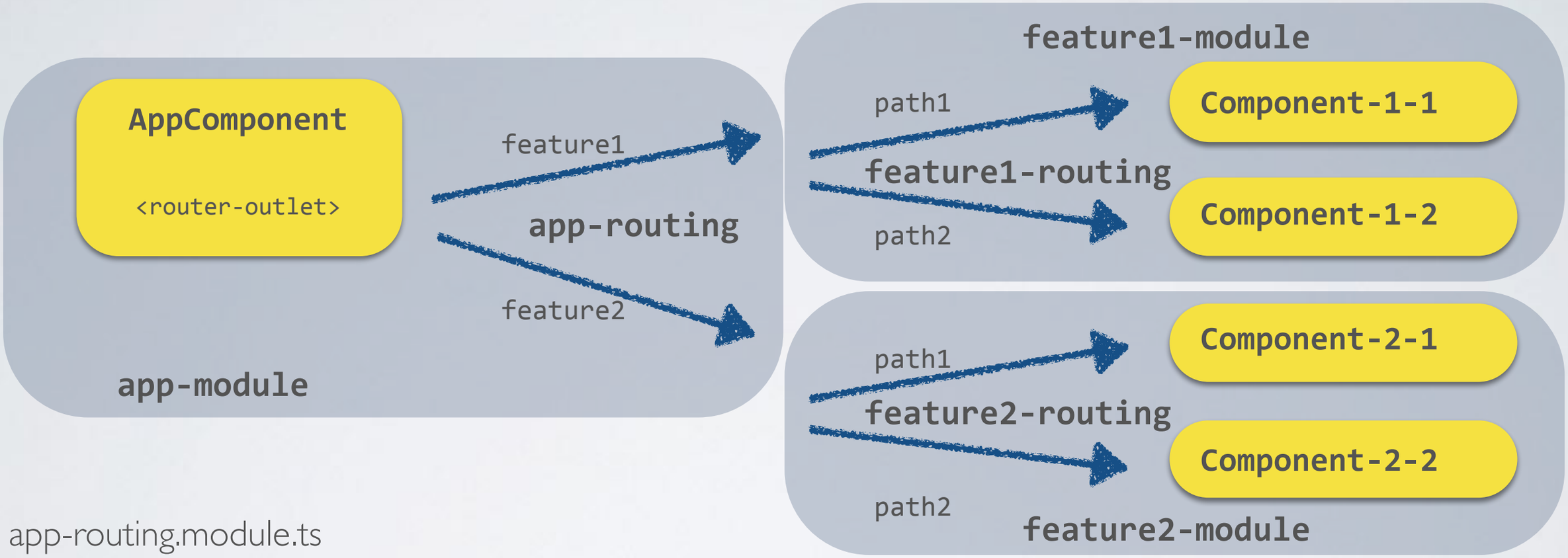
```
this.router.navigateByUrl('/details/5');
```

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

Hierarchical Routing

Feature modules can define their own routes.

Does only work with lazy-loaded modules :-(



app-routing.module.ts

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

home-routing.module.ts

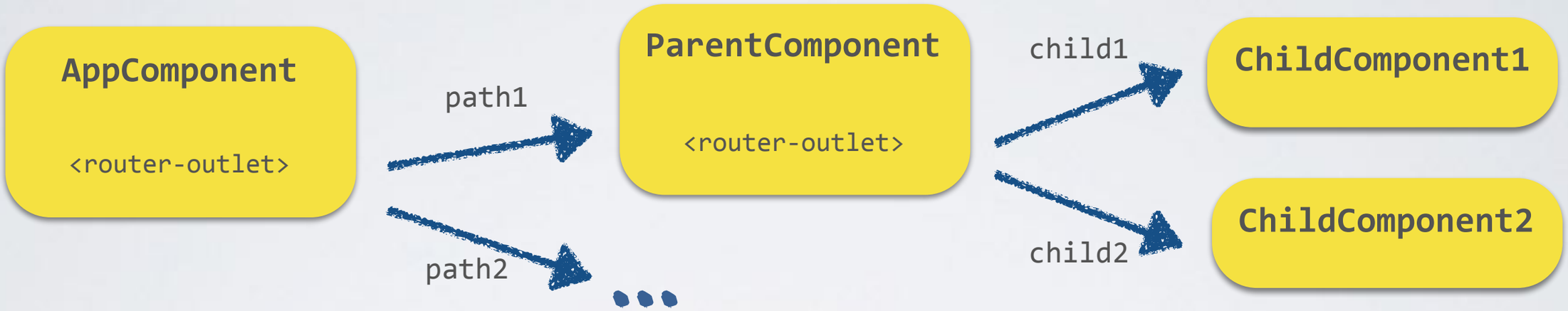
```
const routes: Routes = [
  {path: '', component: HomeShellComponent},
  {path: 'welcome', component: WelcomeComponent},
];
```

catalog-routing.module.ts

```
const routes: Routes = [
  {path: '', pathMatch: 'full', redirectTo: 'list'},
  {path: 'list', component: CatalogShellComponent},
];
```

Child Routing

Enabling a component tree with nested **<router-outlet>**.



app-routing.module.ts

```
const routes: Routes = [
  {path: 'path1', component: ParentComponent,
     children: [
       {path:'', redirectTo: 'child1', pathMatch: 'full' },
       {path:child1, component: ChildComponent1 },
       {path:child2, component: ChildComponent2 },
     ]
  },
  ...
];
```

Multiple Routes

Angular provides named outlets to display multiple routes in different outlets at the same time.

```
{path: 'multiple-routes', component: MultipleRoutesParentComponent,
    children: [
      {path: 'child1', component: MultipleRoutesChild1Component},
      {path: 'child2', component: MultipleRoutesChild2Component},
      {path: 'side1', component: MultipleRoutesChild2Component, outlet: 'side'},
      {path: 'side2', component: MultipleRoutesChild1Component, outlet: 'side'},
    ]
  },
```

```
<div style="display: flex">
  <div>
    <h3>Main Content</h3>
    <router-outlet></router-outlet>
  </div>

  <div>
    <h4>Side Content</h4>
    <router-outlet name="side"></router-outlet>
  </div>
</div>
```

Link parameter array:

```
[{outlets:{side:['side1']}}]
```

```
[{outlets:{primary: 'child1',side:['side1']}}]
```

examples:

```
<a [routerLink]="[{outlets:{side:['side1']}}]">Side 1</a>
```

```
this.router.navigate([{outlets:{side:['side1']}}]);
```

Resulting url: **`http://localhost:4444/routing/multiple-routes/(child1//side:side1)`**

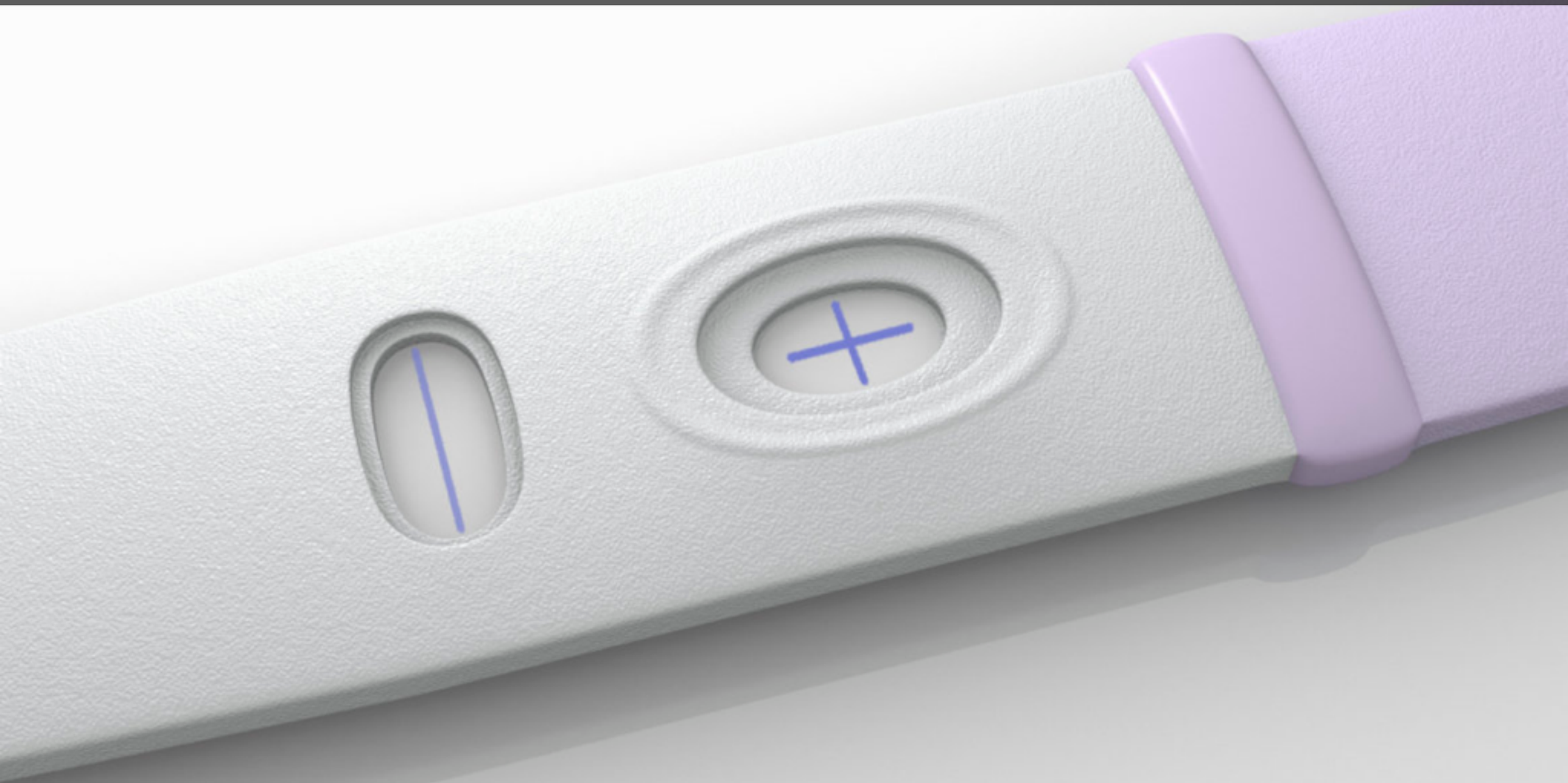
https://angular.io/guide/router#displaying-multiple-routes-in-named-outlets

Dynamic Routing

https://medium.com/@DenysVuika/dynamic-routes-with-angular-6fda03b7fa2c

Detail: die dynamischen Komponenten müssen noch zusätzlich im app.module.ts unter entryComponents hinzugefügt werden,

Testing



Test Framework

API for writing tests in code.
Run the tests in the browser.

Jasmine · mocha · chai · QUnit · tape · Jest

Node based test-runner

Run JavaScript tests in Node.js
No DOM is provided.
(use jsdom as a DOM implementation for node)

Jasmine · mocha · Jest · tape

Advanced Test Runner

Run tests in multiple browsers (including PhantomJS)
Provide different reporters (coverage, junit …)
Run test continuously.
Automatically launch browsers.

KARMA · Testem

End-to-End Testing

Automate browsers via Selenium

Protractor (end to end testing for AngularJS) · Nightwatch.js · cypress

JS Test Execution

(where to run tests)

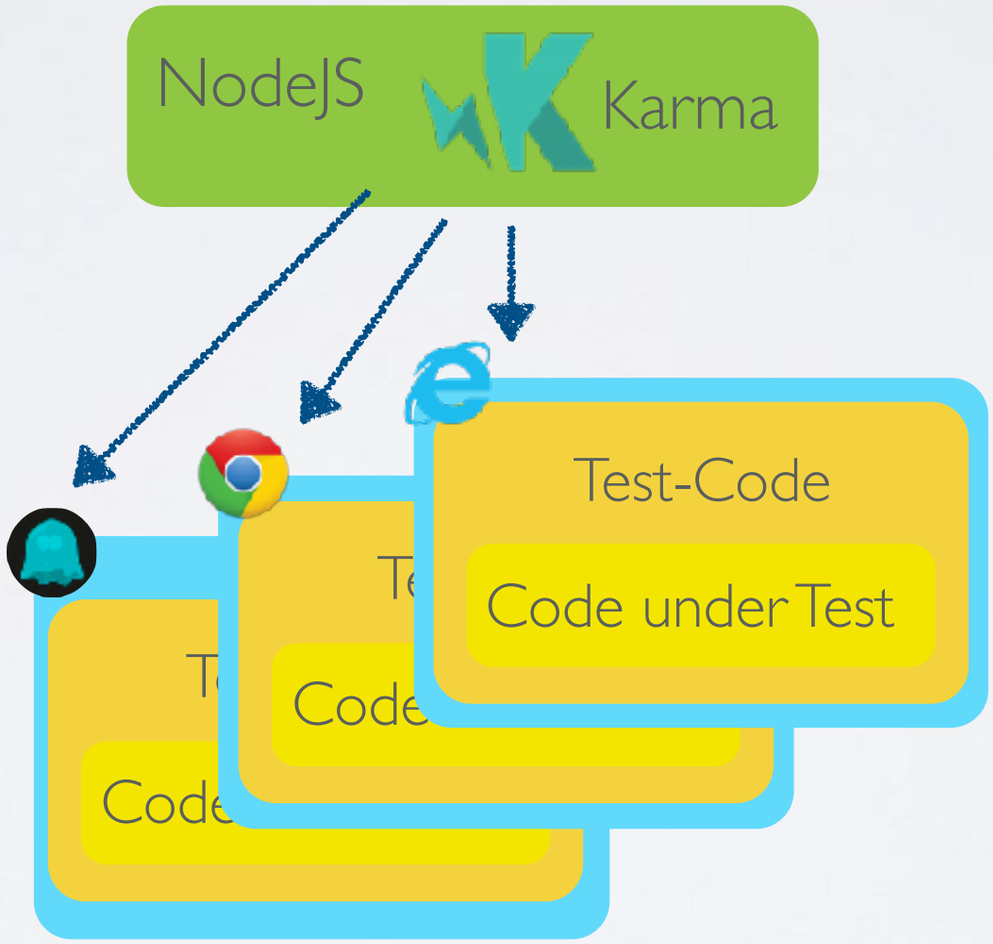
Pure Node-Testing

(unit-tests)

Run tests in node.
Optionally use jsdom to fake DOM.

Karma

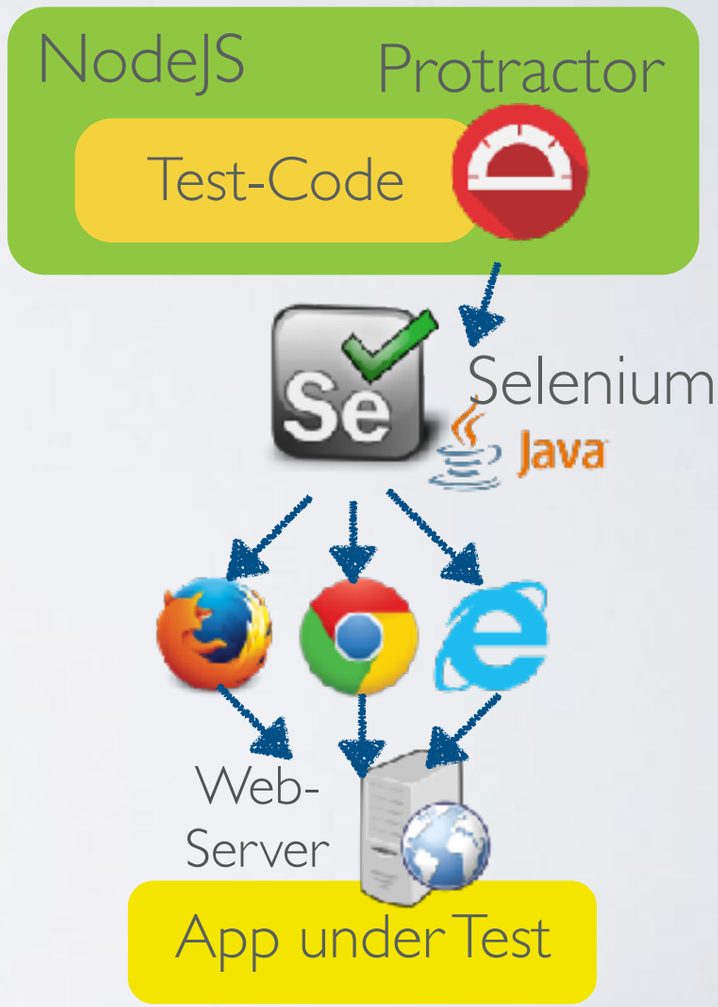
(unit-tests)



Run tests in browser.
Optionaly use PhantomJS as headless browser (easy installation, speed).

Protractor

(end-to-end tests, e2e)



Script a browser to interact with a deployed app.

Jasmine

https://jasmine.github.io/

structure:

```
describe("Demo", () => {
  let sut;

  beforeEach(function () {
    sut = {};
    sut.mvValue = false;
  });

  it(...);
});
```

run a single test: `fit(...)`

ignore test: `xit(...)`

synchronous test:

```
it("simple assertion should succeed", () => {
  sut.myValue = true;

  expect(sut.myValue).toBe(true);
});
```

asynchronous test:

sinaling test finished

```
it("aysnc assertion should succeed", (done) => {
  setTimeout(() => sut.myValue = true, 1000);

  setTimeout(() => {
    expect(sut.myValue).toBe(true);
    done();
  }, 6000);
}, 10000);
```

overriding test timeout

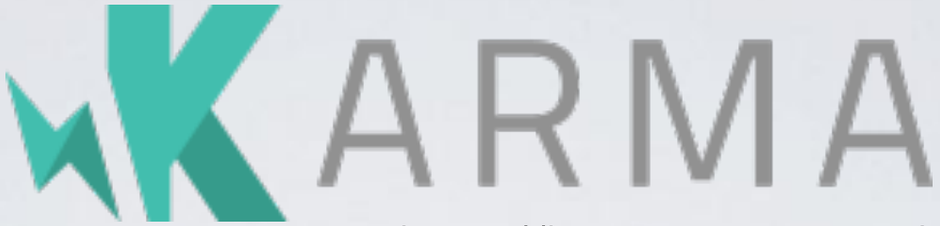
Matchers: `toBe()`, `toEqual()`, `toBeFalsy()`, `toContain()` ...

https://jasmine.github.io/api/3.3/matchers.html

https://jasmine.github.io/api/3.3/Spy.html

Mocking:

```
const httpClientSpy = jasmine.createSpyObj('httpClient', ['get']);
httpClientSpy.get.and.returnValue(of(testData));
```

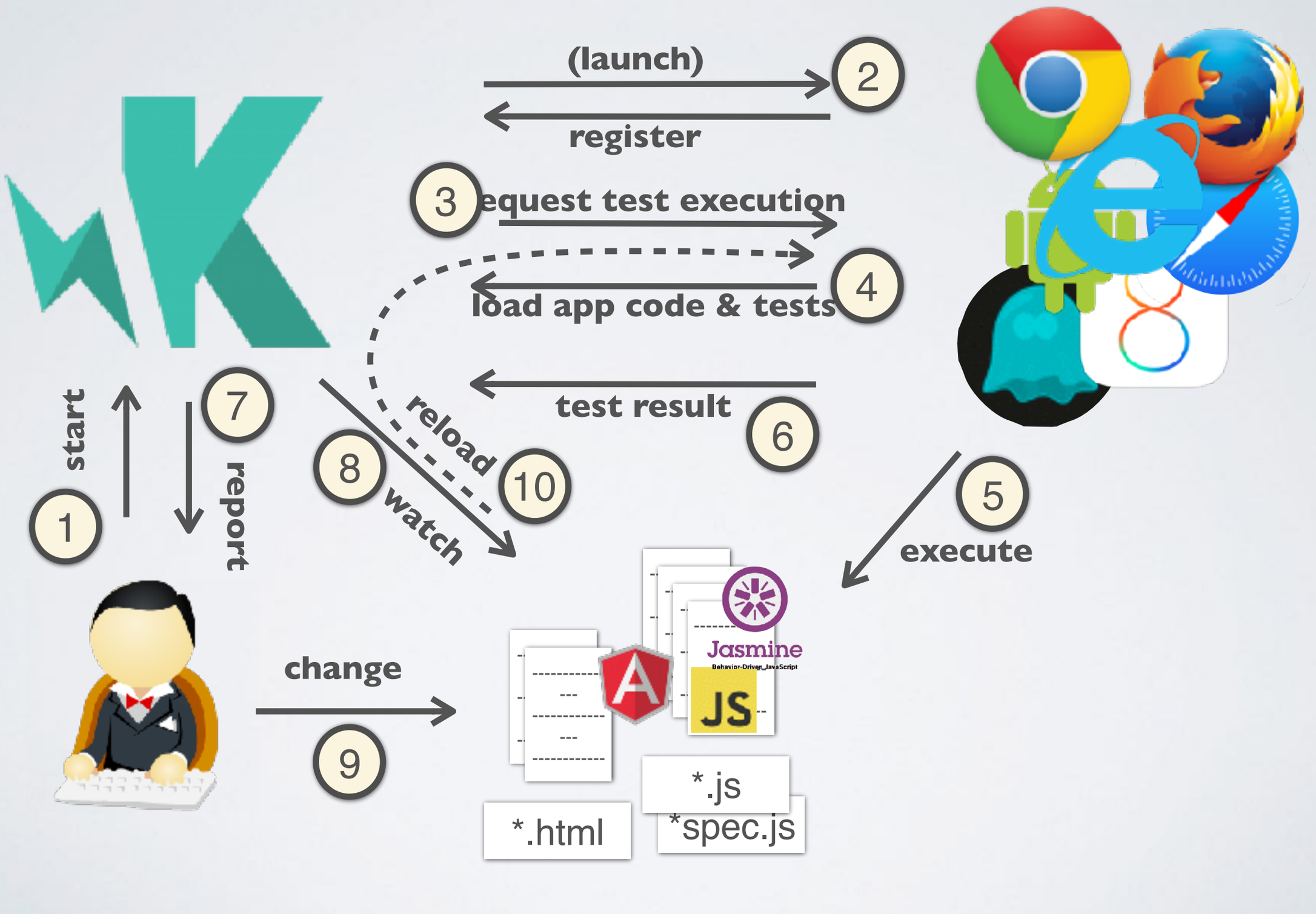


http://karma-runner.github.io/

- Karma is typically used for unit-test
- Karma is a "test runner"
  - ... but tests are executed in a browser
- Tests are written with a JavaScript test framework (provides the API):
  - Karma supports Jasmine, Mocha, QUnit ...
  - The Angular CLI uses Jasmine
- Installing and using Karma:


```
npm install -g karma-cli
npm install karma
karma init
karma start
```

KARMA


(launch)
2
register
3
equest test execution
load app code & tests
4
test result
6
5
execute
reload
10
8
watch
7
report
start
1
change
9
Jasmine
Behavior-Driven JavaScript
JS
*.js
*spec.js
*.html

Protractor

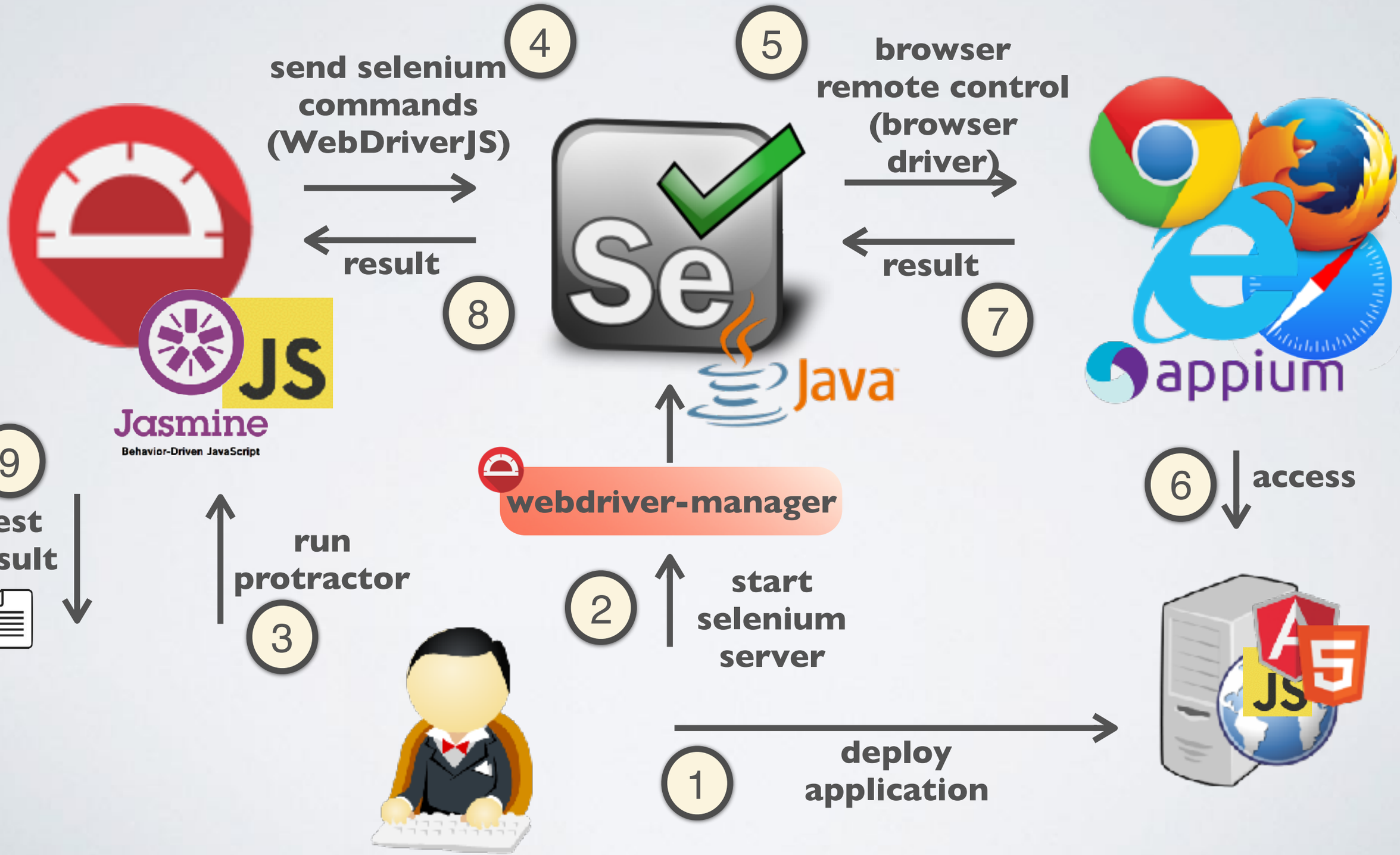
end to end testing for AngularJS

- Protractor is a "end-to-end test framework"
  - ... it is built on top of Selenium/WebDriverJS
- Tests are written with a JavaScript test framework (provides the API):
  - Jasmine, Mocha, Cucumber ...
- Protractor is used for end-to-end tests
  - Browser automation against a deployed application
- Installing and using Protractor:

```
npm install -g protractor
webdriver-manager update
webdriver-manager start
protractor protractor.conf.js
```

Protractor

end to end testing for AngularJS



Angular Test Types

isolated:
just JavaScript objects

shallow:
component with template without child components

integrated:
component with template including child components

e2e:
starting application & browser application

unit-test

KARMA

integration-test

Protractor
end to end testing for AngularJS

Angular CLI Test Infrastructure

Angular CLI: Running Unit Tests (https://angular.io/cli/test)

Configuration: `src/karma.conf.js`

Executing in the continuous build:

```
ng test --watch=false --codeCoverage=true --browsers=ChromeHeadless
```

Adding the junit reporter:

```
npm i -D karma-junit-reporter
```

and configure it in `karma.conf.js`

Angular CLI: Running e2e Tests

Configuration: `e2e/protractor.conf.js`

```
ng e2e
```

Using headless Chrome:

```
capabilities: {
    chromeOptions: {
      args: [ "--headless" ]
    },
    'browserName': 'chrome'
  },
```

Angular Test Infrastructure

https://angular.io/guide/testing

```
import {TestBed, async, fakeAsync} from '@angular/core/testing';
import {NO_ERRORS_SCHEMA} from '@angular/core';
```

`TestBed`: emulates a NgModule for the test

```
TestBed.configureTestingModule({
        declarations: [Comp]
      });
fixture = TestBed.createComponent(Comp);
```

`async` utility: make the test wait until all async operations are finished

```
it('is async', fakeAsync(() => {
      tick(1000);
      expect(...).toEqual(...);
    })
  );
```

`fakeAsync` utility: faking time with a virtual clock

```
it('is async', async(() => {
     return fixture.whenStable()
        .then(() => {
            expect(...).toEqual(...);
      });
}));
```

`NO_ERRORS_SCHEMA` configures Angular to ignore unknown elements/attributes in the template.

```
TestBed.configureTestingModule({
          declarations: [AppComponent],
          schemas: [ NO_ERRORS_SCHEMA ]
        });
```

Dependency Injection

Dependency Injection

With dependency injection we can provide objects to other Angular constructs.

Consumers declare dependencies in their constructor.

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

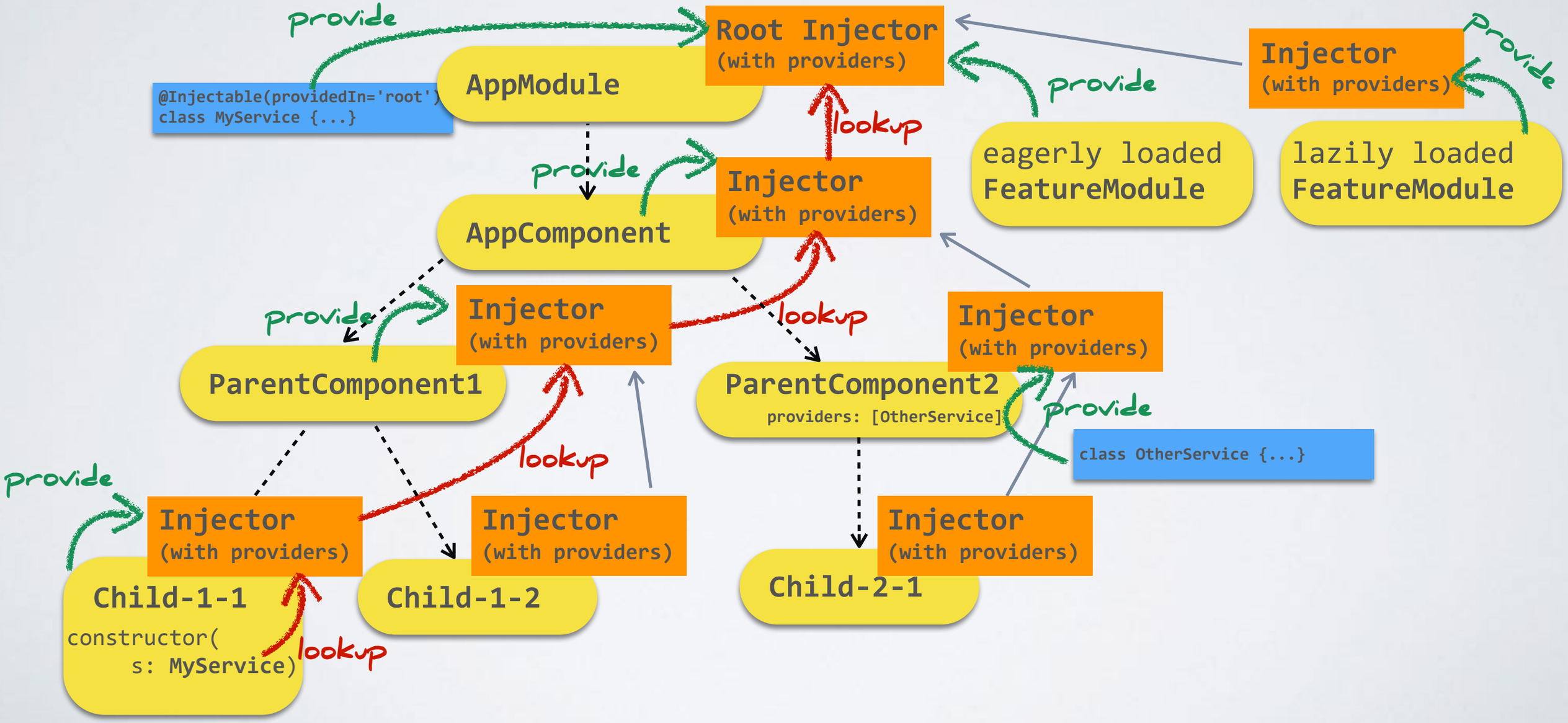
```
@Injectable({providedIn: 'root'})
export class DataService {
  constructor(){}
}
```

Dependency injection relies on TypeScript metadata:
`tsconfig.json` must contain `emitDecoratorMetadata:true`

https://angular.io/guide/dependency-injection

Dependency Injection

Services are provided via *injectors*. Angular builds a hierarchy of injectors parallel to the application's component tree.



https://angular.io/guide/hierarchical-dependency-injection

Dependency Injection

There are different mechanism to register services with an injector:

Services can be registerd in the root injector:

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

(introduced in Angular 6)

Services can be registerd in a module:

```
import { UserModule } from './user.module';
@Injectable({providedIn: UserModule})
export class DataService {}
```

(introduced in Angular 6)

Note: You need to introduce another module to avoid circular dependencies with this approach! See [1].

A NgModule can register singleton services:

```
import DataService from '../path/DataService'
@NgModule({
    ...
    providers: [DataService]
})
export class AppModule {}
```

The service will be registered in the root injector if the module is *eager loaded*.
The service will be registered in a new injector if the module is *lazy loaded*.

A component can register services for itself and its component subtree:

```
import DataService from '../path/DataService'
@Component({
    ...
    providers: [DataService]
})
export class MyComponent {}
```

https://angular.io/guide/providers
https://angular.io/guide/hierarchical-dependency-injection

[1]: https://www.softwarearchitekt.at/post/2018/05/06/the-new-treeshakable-providers-api-in-angular-why-how-and-cycles.aspx

Dependency Injection

Dependencies can have dependencies.

```
export class FirstService {
  ...
}
```

```
@Injectable()
export class SecondService {
  constructor(private firstService: FirstService){}
}
```

```
@Component({
    selector: 'my-component',
    template: 'path/template.html'),
    providers: [SecondService],
})
export class MyComponent {
  constructor(private secondService: SecondService){}
  ...
}
```

Technically `@Injectable()` is only needed if the server itself has dependencies.
However it is a good practice to add it to every service as default.

Dependency Injection Providers

Short form:

```
providers: [MyService]
```

Class provider:

```
providers: [{provide: MyService, useClass: MyService}]
```

token

provider definition

remember: this can't be an interface!

Alias provider:

```
providers: [{provide: MyOldService, useClass: MyNewService}]
```

Value provider:

```
const theServiceInstance = { ... }
providers: [{provide: MyService, useValue: theServiceInstance}]
```

Factory provider:

```
function theServiceFactory(service1: Service1, service2: Service2){
   ...
   return new MyService(...);
}

providers: [{provide: MyService,
             useFactory: theServiceFactory,
             deps: [Service1, Service2]
           }]
```

Injector tokens:

```
import { NG_VALIDATORS } from '@angular/forms';
providers: [{provide: NG_VALIDATORS, useExisting: CapitalLetterValidatorDirective, multi: true}]
```

injector token

https://angular.io/guide/dependency-injection-providers
https://angular.io/api/core/InjectionToken

Configuration via Dependency Injection (examples)

Configuring hash routing:

```
import {HashLocationStrategy, LocationStrategy} from '@angular/common';
...
providers: [{provide: LocationStrategy, useClass: HashLocationStrategy}],
```

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

Registering a custom validator:

```
import { NG_VALIDATORS } from '@angular/forms';
...
providers: [{provide: NG_VALIDATORS, useExisting: CapitalLetterValidatorDirective, multi: true}]
```

Registering a http interceptor:

```
import { HTTP_INTERCEPTORS } from '@angular/common/http';
import { TokenInterceptor } from './...';
...
providers: [{ provide: HTTP_INTERCEPTORS, useClass: TokenInterceptor, multi: true }}]
```

Registering a global error handler:

```
import { HTTP_INTERCEPTORS } from '@angular/common/http';
import { MyErrorHandler } from './...';
...
providers: [{ provide: ErrorHandler, useClass: MyErrorHandler }}]
```

Internationalization

Internationalization

Angular v7 has official i18n: https://angular.io/guide/i18n
- the app has to be reloaded to switch language, no language switching at runtime
- you have to build & deploy the app in each language separately (translations are happening at build time)
- No translations in code, just in the template
- No direct json support (support for XLIFF or XMB)

"runtime i18n" is promised for a future version of Angular (v7 or later, coupled to "Ivy" ...):
- https://github.com/angular/angular/issues/16477
- https://www.youtube.com/watch?v=miG-ghJhFPc

ngx-translate is an open-source alternative:
- http://www.ngx-translate.com/ / https://github.com/ngx-translate
- language switching at runtime
- supports translations in code
- json support
- https://github.com/ngx-translate/core/issues/495
- https://stackblitz.com/github/ngx-translate/example
- The future of ngx-translate is unclear: https://github.com/ngx-translate/core/issues/783

ngx-translate

https://github.com/ngx-translate/core

```
npm install @ngx-translate/core
npm install @ngx-translate/http-loader
```

Setup:

```
export function HttpLoaderFactory(http: HttpClient) {
  return new TranslateHttpLoader(http);
}
@NgModule({
    imports: [
        BrowserModule,
        TranslateModule.forRoot({loader: {
          provide: TranslateLoader,
          useFactory: HttpLoaderFactory,
          deps: [HttpClient]
      }})
    ],
    bootstrap: [AppComponent]
})
export class AppModule { }
```

```
@NgModule({
    imports: [TranslateModule]
export class FeatureModule { }
```

Usage:

```
<h3>{{ 'HELLO' | translate }}</h3>
```

```
<h3 translate>HELLO</h3>
```

Hidden Treasures

Loading initial Data

Loading data before the Angular application is bootstrapped.

Note: Do not use **`environment.ts`** to store context data (i.e. backend-url ...) in the source code!

Using the `APP_INITIALIZER` provider:

```
@NgModule({
  declarations: [ ... ],
  imports: [ ... ],
  providers: [{
     provide: APP_INITIALIZER,
     multi: true,
     useFactory: loadInitialData,
     deps: [HttpClient]
  }],
  bootstrap: [AppComponent]
})
```

```
export function loadInitialData(
                httpClient: HttpClient) {
  return () => {
    return httpClient.get(URL)
      .toPromise()
      .then(response => {
        console.log('Loaded data: ', response);
        // we could save the data into a service
       });
  };
}
```

Demo:

`10-demos/src/app/app.module.ts`
`10-demos/src/app/init.ts`

https://davembush.github.io/where-to-store-angular-configurations/

Global Error Handling

Using the `ErrorHandler` provider:

```
import {GlobalErrorHandler} from './...'

@NgModule({
  declarations: [ ... ],
  imports: [ ... ],
  providers: [{
     provide: ErrorHandler,
     useClass: GlobalErrorHandler
  }],
  bootstrap: [AppComponent]
})
```

```
@Injectable()
export class GlobalErrorHandler
                implements ErrorHandler {
  handleError(error: any): void {

    alert('Sorry, we fucked up ....');
    location.href = '/';
     // IMPORTANT:
     // If the error is not re-thrown then
     // it is swallowed!
     // throw error;
   }
}
```

Demo:

`10-demos/src/app/app.module.ts`
`10-demos/src/app/global-error-handler.service.ts`

https://medium.com/@amcdnl/global-error-handling-with-angular2-6b992bdfb59c


The Future
NEXT EXIT

Angular Ivy

The Next Generation Angular Renderer

Announced in early 2018. Not part of Angular 7.

A big internal rewrite of Angular.
Promised to be completely API compliant.

("you can just switch to the Ivy Renderer with a flag")

Advantages:

- *smaller* bundles ("Hello World" in 2.7kb)
- *faster* compiles
- better readability/*simpler* debugging of generated template code
- enabling meta-programming like higher order components

Some new Angular features/APIs are now coupled to Ivy:

- runtime internationalization
- lazy loading of components

Try it:

```
ng new iyv-app --experimentalIvy=true
ng serve --aot
```

https://is-angular-ivy-ready.firebaseapp.com/#/status

Build System: Bazel

Google does not use Webpack nor the Angular CLI.

The *ABC* initiative in the Angular projects aims to align the Google tooling with the "open source" ecosystem.
See: http://g.co/ng/abc

Bazel is the open source project coming out of a internal build tool at Google called *Blaze*.

Bazel

https://bazel.build/

A feature of Bazel is that fine grained build artifacts can be specified (i.e. each component is a build artifact). These artifacts can be cached (i.e. on disk) which enables very fast incremental builds and also to parallelize builds (i.e. with build farms)

Example Angular project: https://github.com/alexeagle/angular-bazel-example
Example React project: https://github.com/thelgevold/react-bazel-example/tree/large-app

Change Detection

Rich Harris
@Rich_Harris

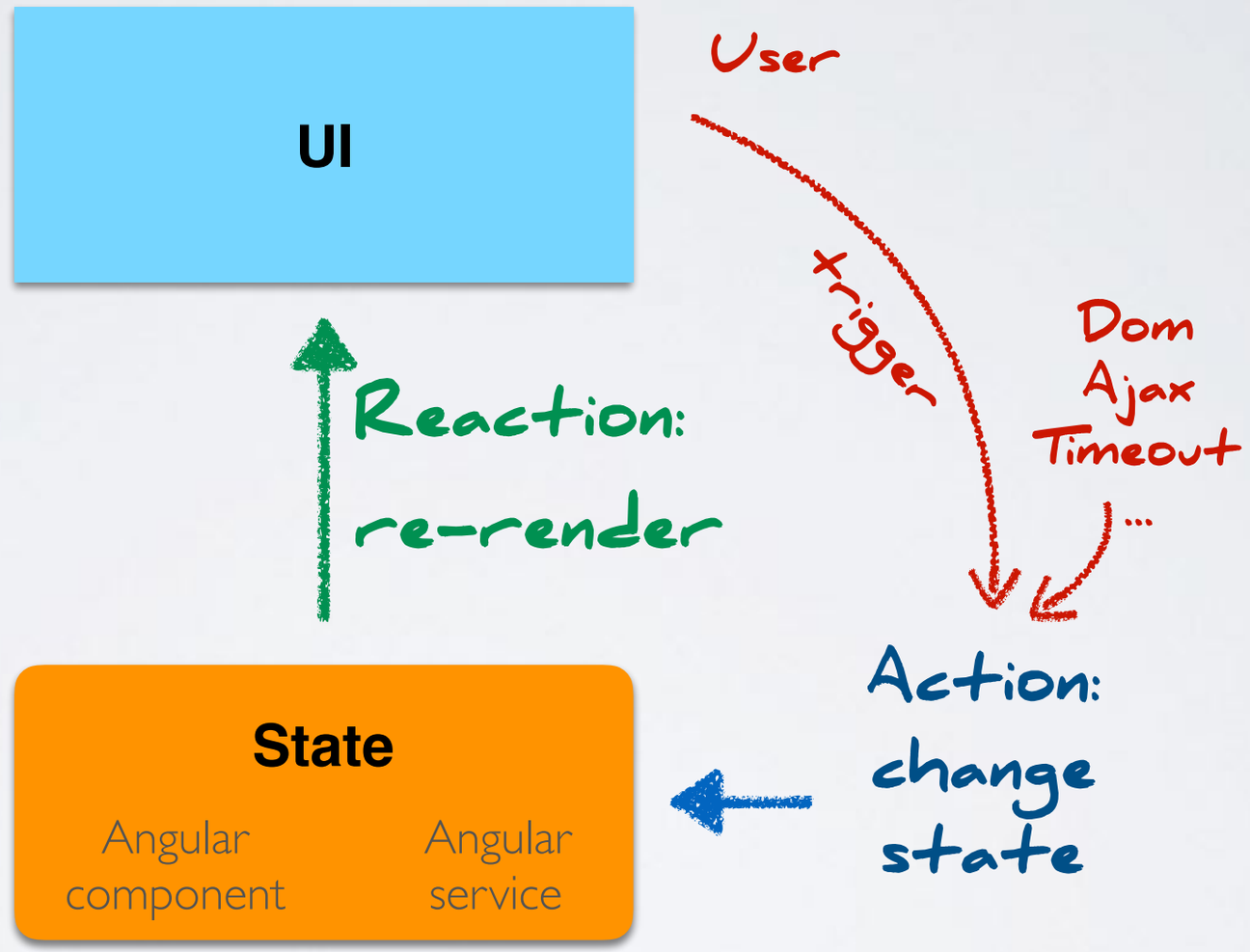
Following

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

State is Managed in JavaScript

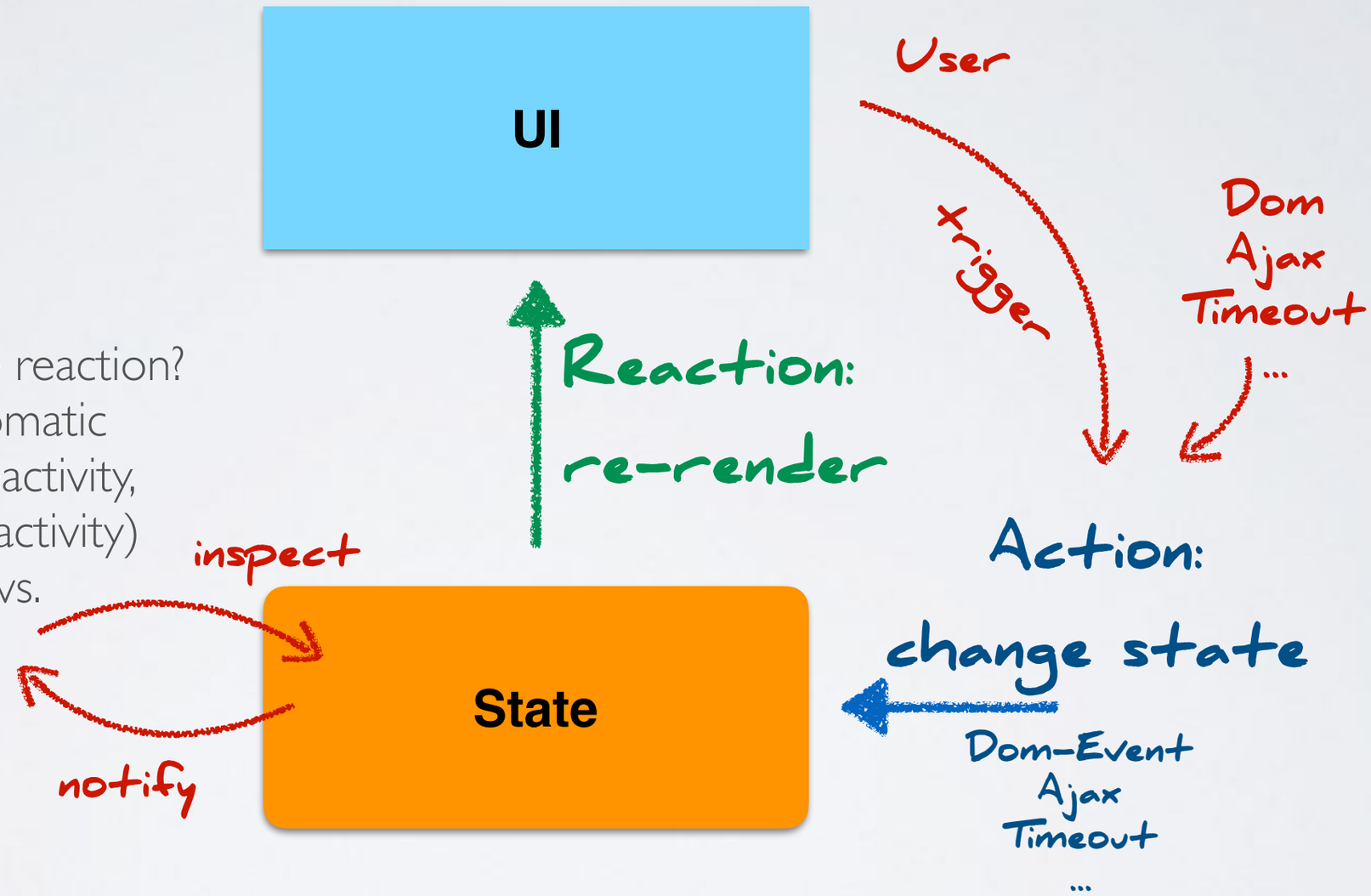


Evan You - Reactivity in Frontend JavaScript Frameworks: https://www.youtube.com/watch?v=r4pNEdIt_I4
http://teropa.info/blog/2015/03/02/change-and-its-detection-in-javascript-frameworks.html

"Reactivity" in UIs

Change Detection:
How to trigger the reaction?

- explicit vs. automatic (transparent reactivity, unobtrusive reactivity)
- dirty-checking vs. notification



Evan You - Reactivity in Frontend JavaScript Frameworks: https://www.youtube.com/watch?v=r4pNEdIt_I4
Front end development and change detection: https://www.youtube.com/watch?v=1i8klHov3vA
http://teropa.info/blog/2015/03/02/change-and-its-detection-in-javascript-frameworks.html

Change Tracking ... What and Why?

Traditional "DOM-centric" applications

UI = state

Browsers have "built-in" change tracking: If the DOM is changed, the UI is re-rendered.

With modern SPA architectures (MVC, MVP, Components …) the client state is represented as JavaScript objects.

UI = f(state)

When to call?

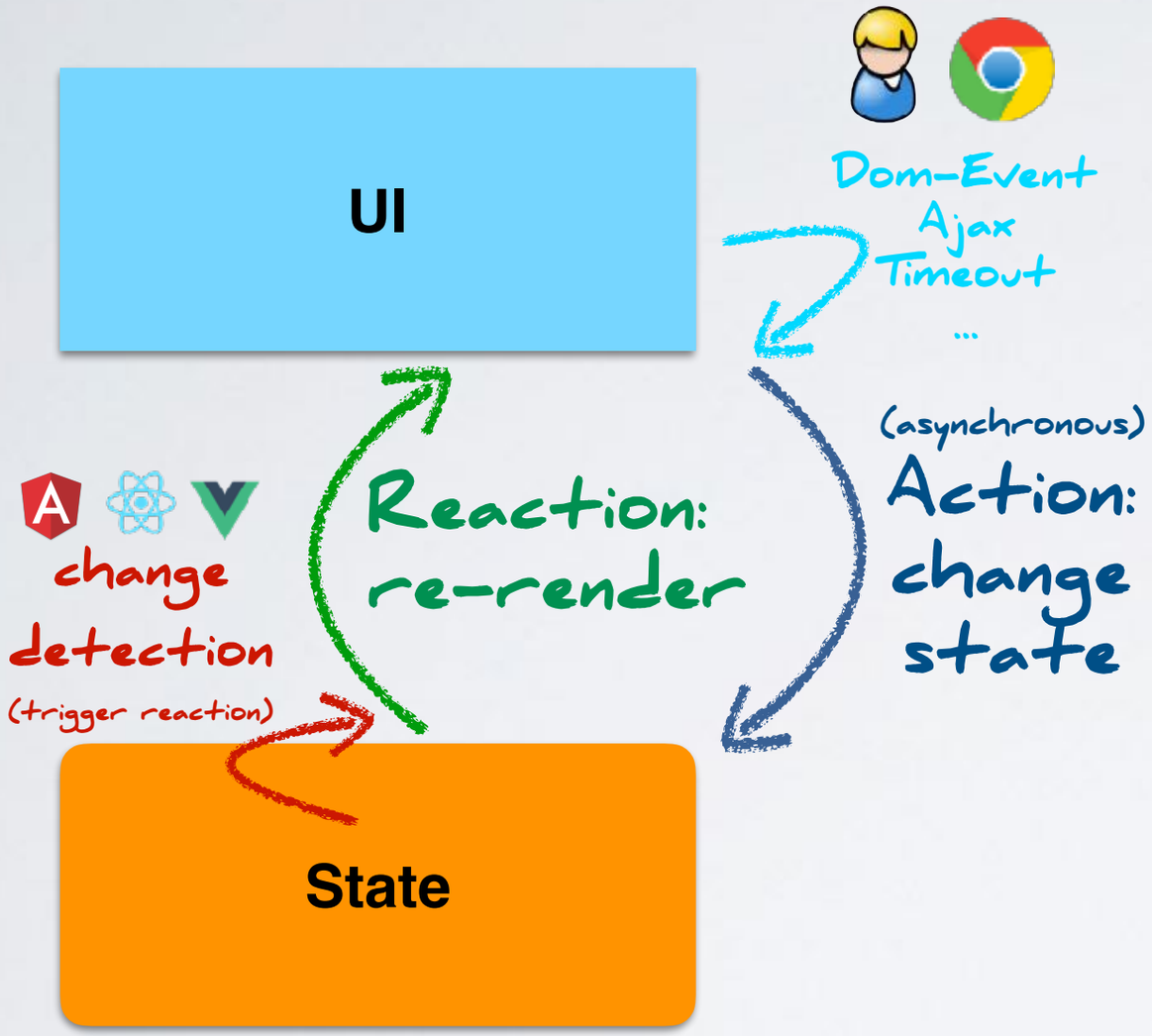
The UI that you can see and manipulate on screen is the result of painting a visual representation of data.

Reactive programming in general addresses the question: How to deal with change over time?

The UI should (automatically) re-render when the state changes.

https://blog.danlew.net/2017/07/27/an-introduction-to-functional-reactive-programming/

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 in order to notify the app when a potential change happens. It triggers the Angular change-detection algorithm which implements dirty checking.
In Angular changes are often explicitly modeled as streams of events/data with RxJS Observables.

React: Change Detection is explicit via `setState`

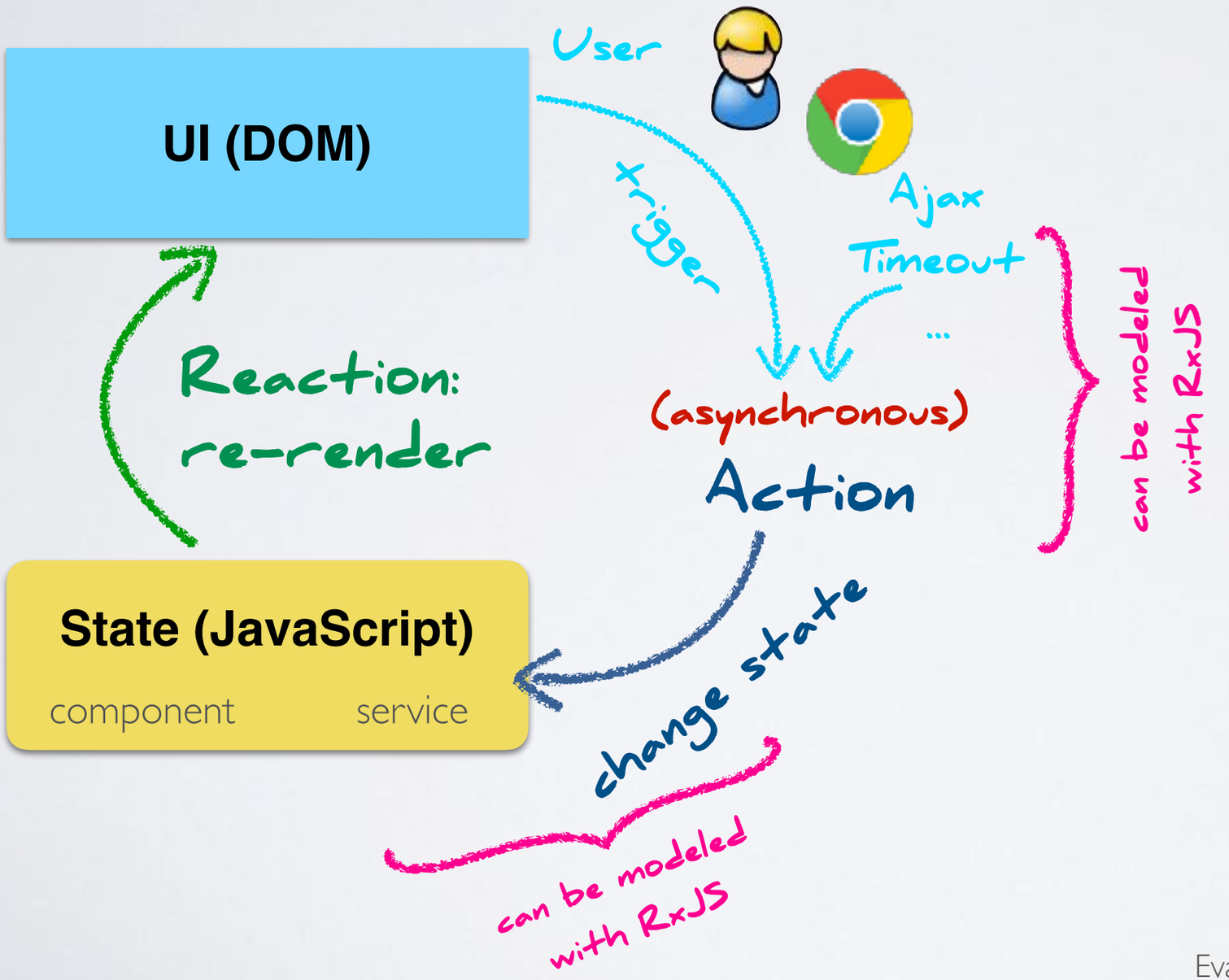
`setState` triggers a complete rendering of the component and its child components into the virtual DOM (this can be optimized).

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

Reactivity in Angular

In Angular we have two concepts how reactivity is typically realised:

Default Angular change detection is a form of *transparent reactivity*. It makes reactivity an *implicit characteristic* of your program.

Vue.js or MobX are other frameworks/libraries that provide transparent reactivity.

Modeling Reactivity explicitly with RxJS is a form of *explicit reactivity*.

You can leverage the power of RxJS for statemanagement.
Libraries like NgRx or Akita expose state changes as Observables.

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

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.

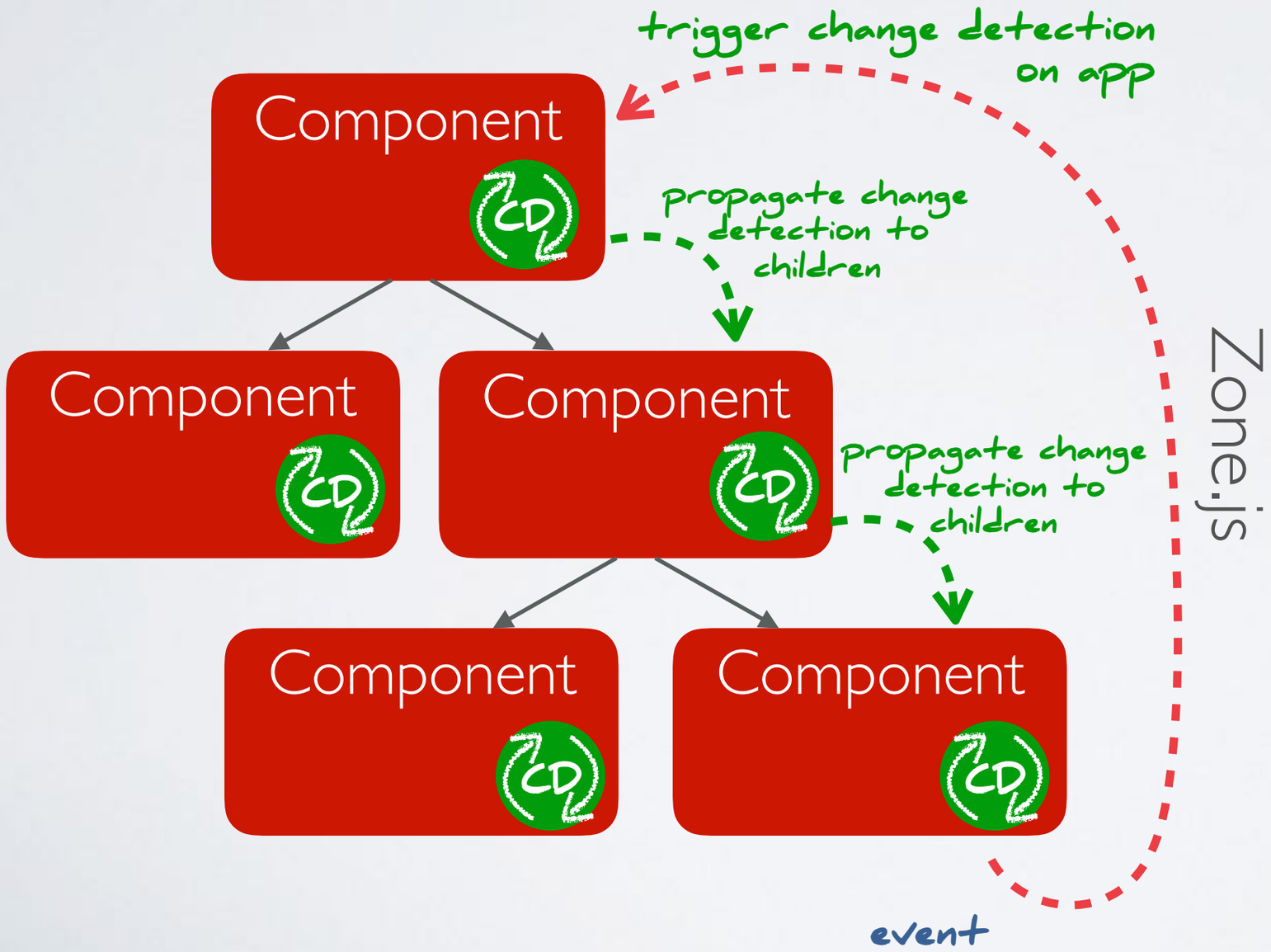
More details: https://github.com/angular/zone.js/blob/master/STANDARD-APIS.md

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.

Demos in: 10-demos/src/app/20-change-tracking

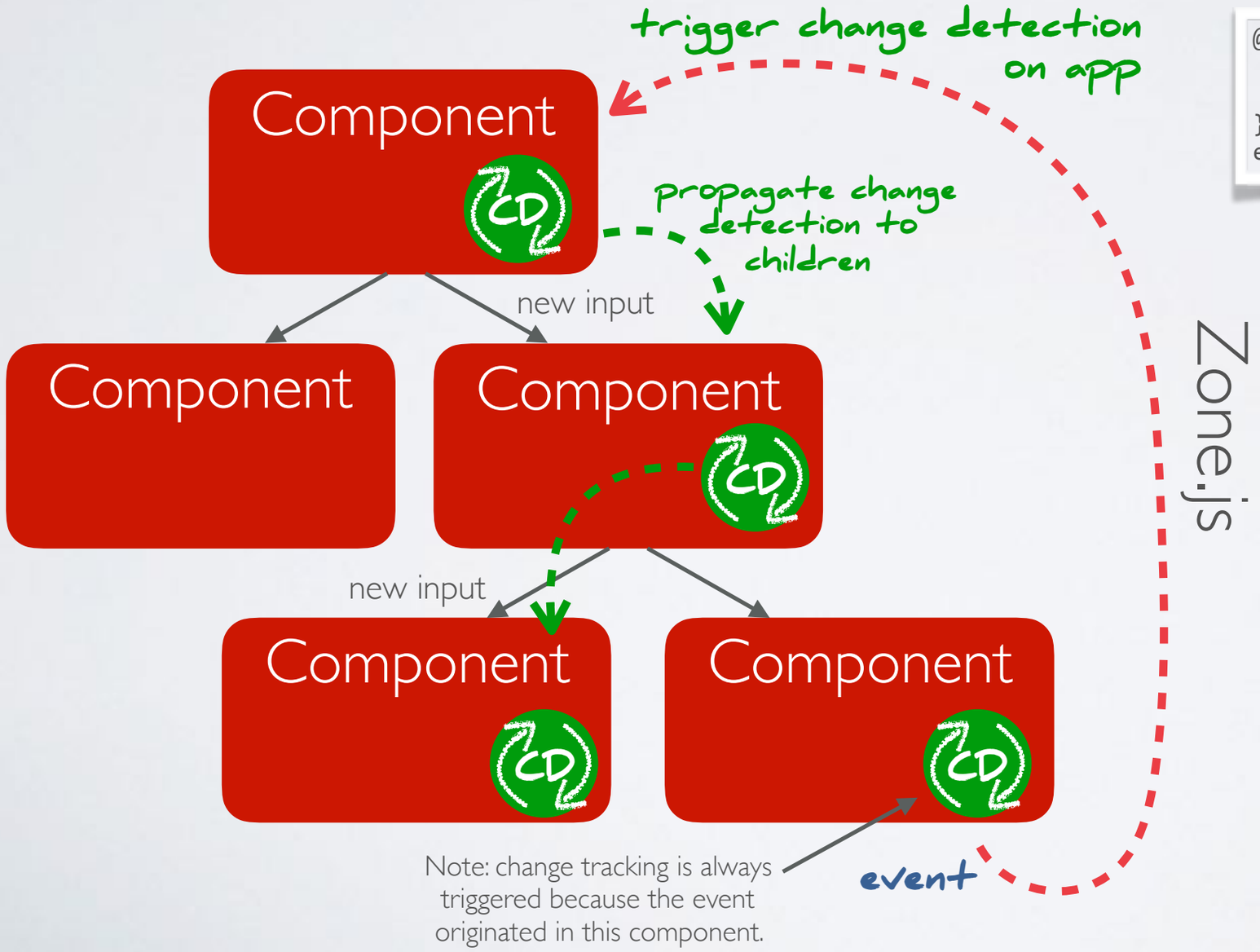
https://blog.angularindepth.com/everything-you-need-to-know-about-the-expressionchangedafterithasbeencheckederror-error-e3fd9ce7dbb4

TODO:

- Advice: Dont use setters & getters in Angular components

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 bound observable emits / An `async` pipe receives an event
- A DOM event of the component fires
- An `@Output` property/`EventEmitter` of the component fires
- 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()`)

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

Change Detection API

- `ApplicationRef.tick()`

  Triggers full change detection cycle of the application.

- `ChangeDetectorRef.detach()`

  The component is not checked for changes any more.

- `ChangeDetectorRef.markForCheck()`

  Marks the associated component as changed so that it gets checked by change detection in the next cycle.

- `Zone.run()`

  Runs a callback in the Angular zone, so that Angular can trigger change detection when the callback has finished.

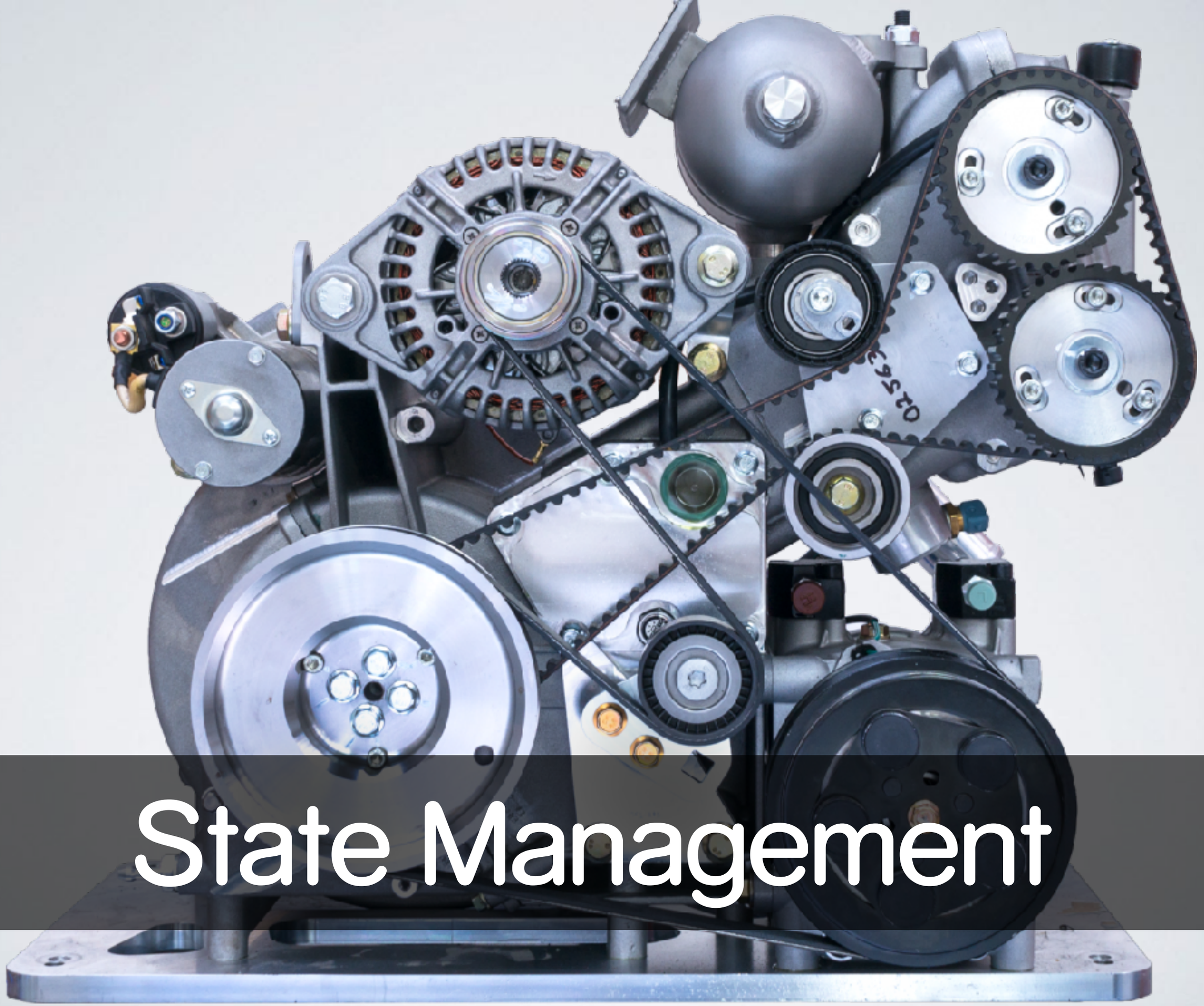
API: Disable Zones

```
platformBrowserDynamic()
    .bootstrapModule(AppModule, {ngZone: 'noop'});
```

Change detection must then be swithched to **`OnPush`** everywhere or triggered manually.

Attention:

- some features of Angular stop working i.e. **`ngModel`**
- 3rd party components & libraries might stop working



State Management

State Management

State management in single page applications is a very broad topic and new patterns and libraries are introduced very often.

ngrx

Redux

ngrx-data

Akita

NGXS

MobX

mobx-state-tree

Observable-Store

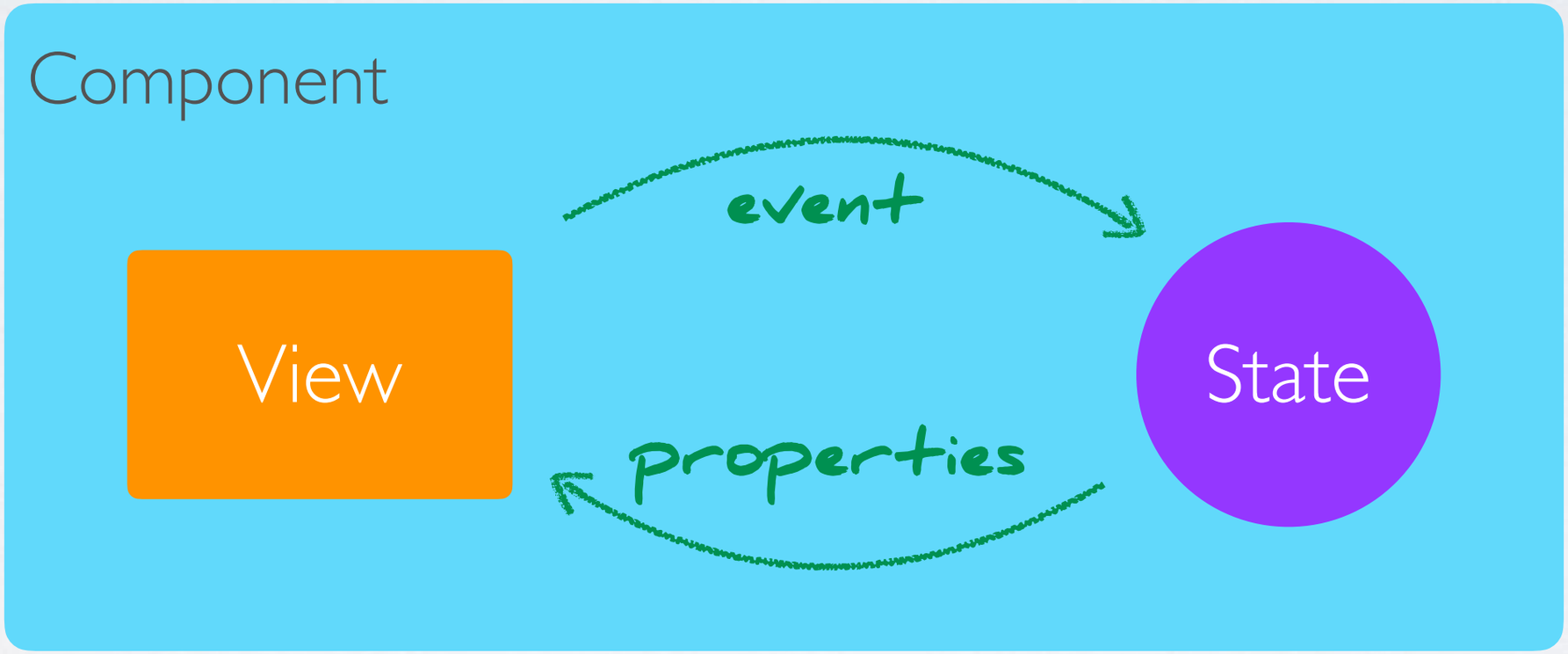
https://github.com/DanWahlin/Observable-Store

Angular Model

https://github.com/angular-extensions/model

A single component

Managing state in a component is simple:



Container vs. Presentation Components

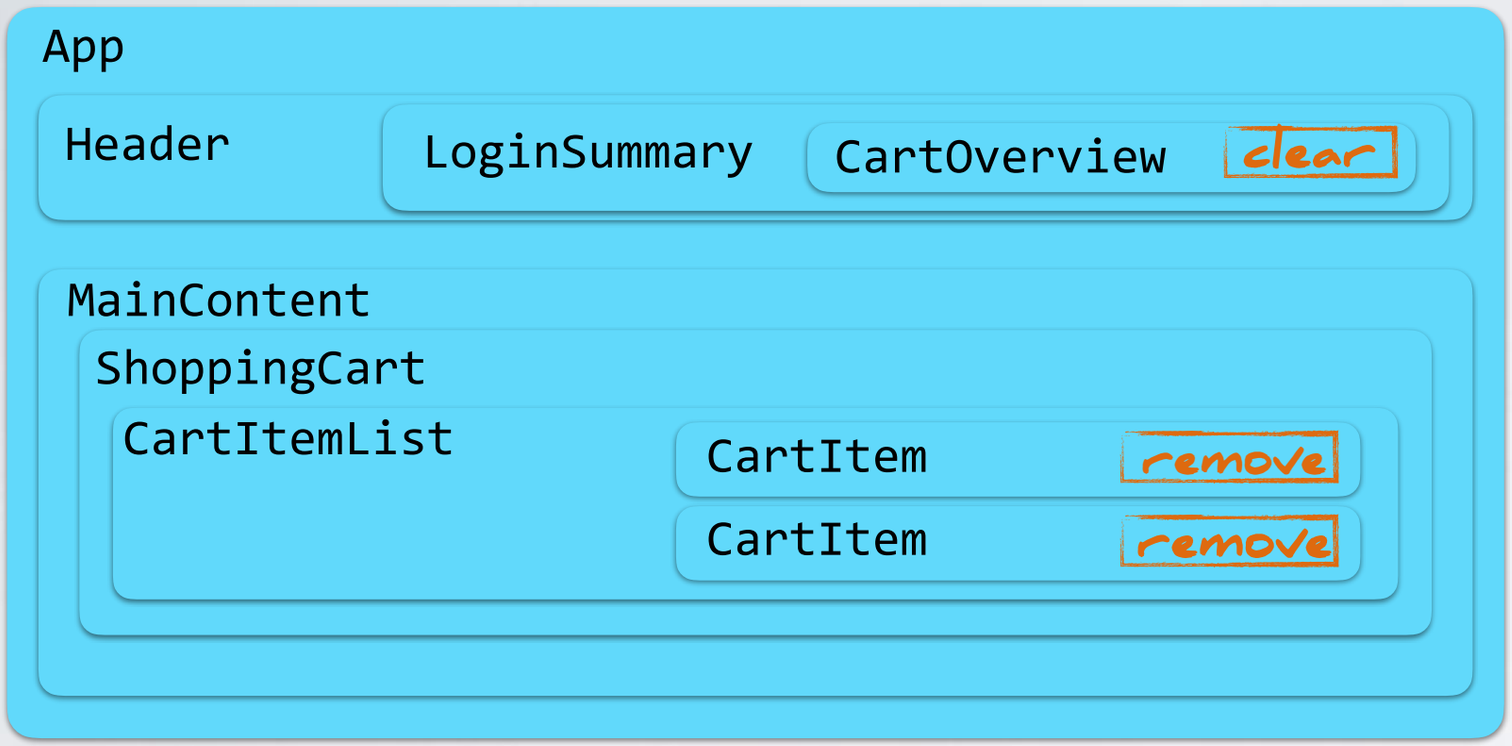
Application should be decomposed in container- and presentation components:

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://toddmotto.com/stateful-stateless-components
https://medium.com/curated-by-versett/building-maintainable-angular-2-applications-5b9ec4b463a1

State Management



CartOverview and CartItemList both are a representation of the current "shopping cart".

Challenges when managing state in a component tree:

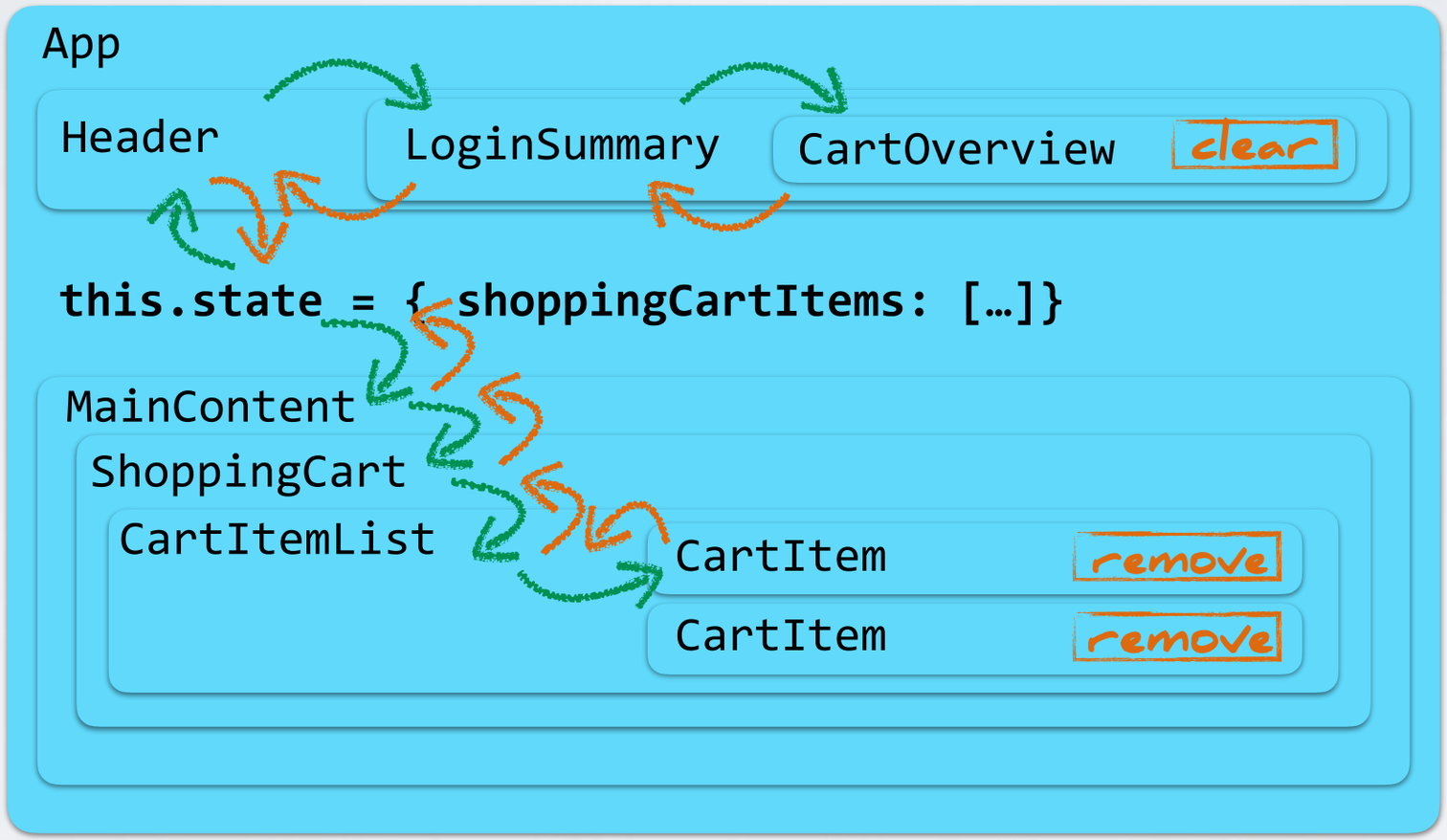
- Multiple components may depend on the same piece of state.
- Different components may need to mutate the same piece of state.

State should never be duplicated between components, there should be a "single source of truth"!

Component Architecture

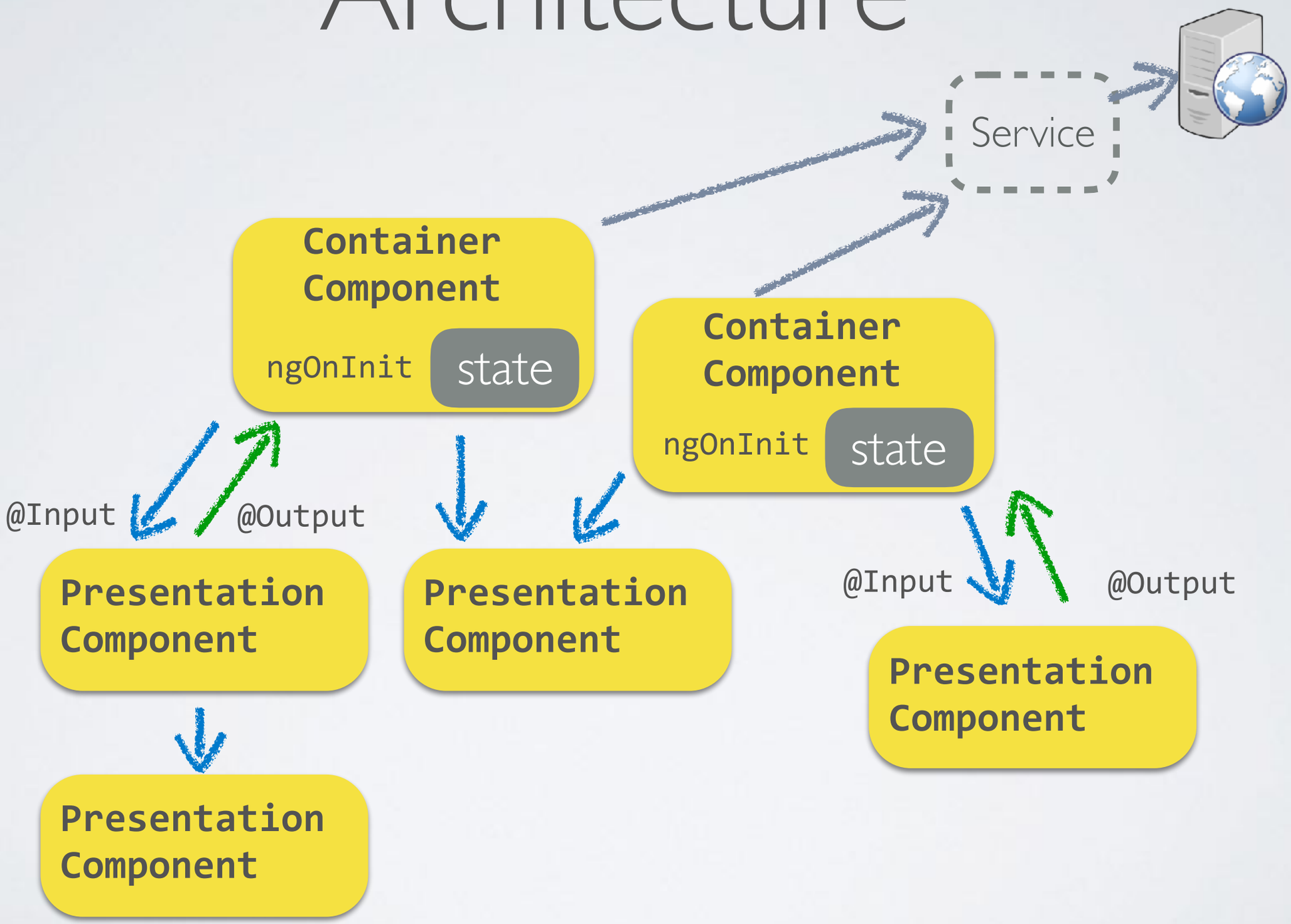
From the component architecture follows the pattern of "lifting state up": if several components need to reflect the same changing data, then the shared state should be liftet up to their closest common ancestor.

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



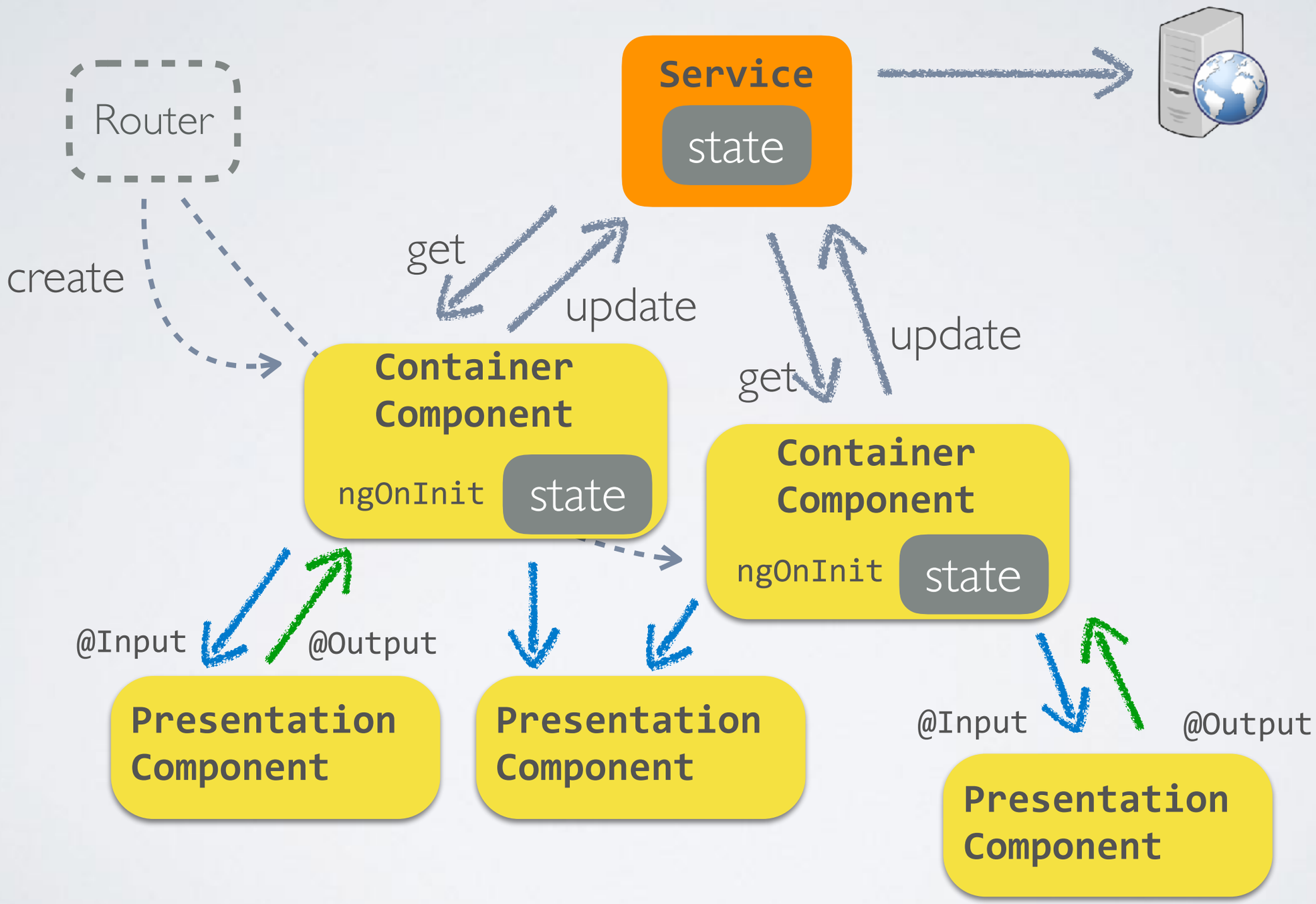
The problem of Angular: a component that is instantiated in a `router-outlet` can not receive `@Input` / `@Output` properties from its parent.

State Management: Component Architecture



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

State Management: Passive Service



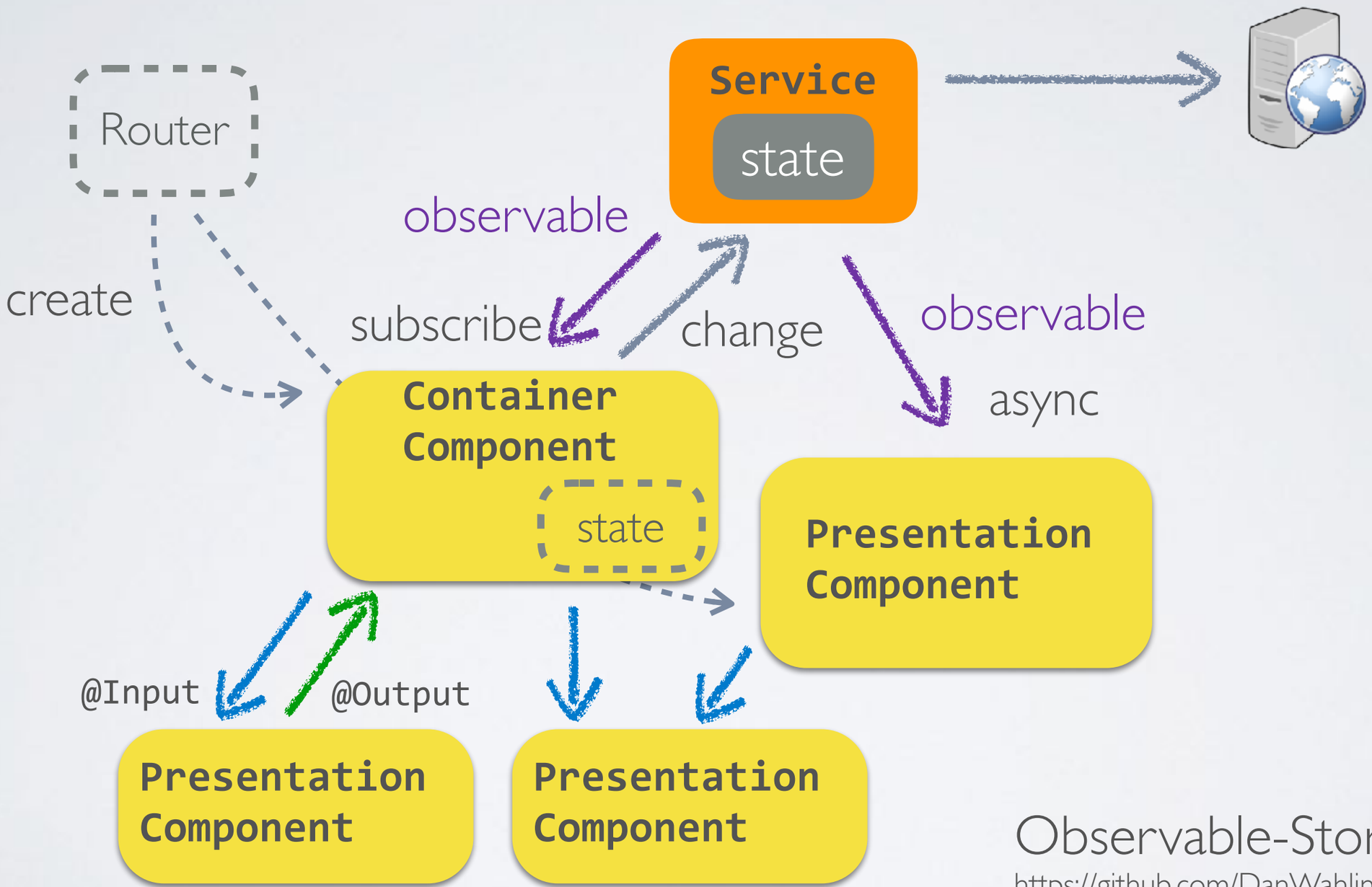
TODO: Simple State Management

https://medium.jonasbandi.net/the-most-simple-state-management-solution-for-angular-1d32706e6f1c

https://twitter.com/shai_reznik/status/1112770263832317953

https://stackblitz.com/edit/angular-sync-state-without-rx
https://stackblitz.com/edit/angular-sync-state-without-rx-perf

Reactive State Management: Observable Data Service



Observable-Store

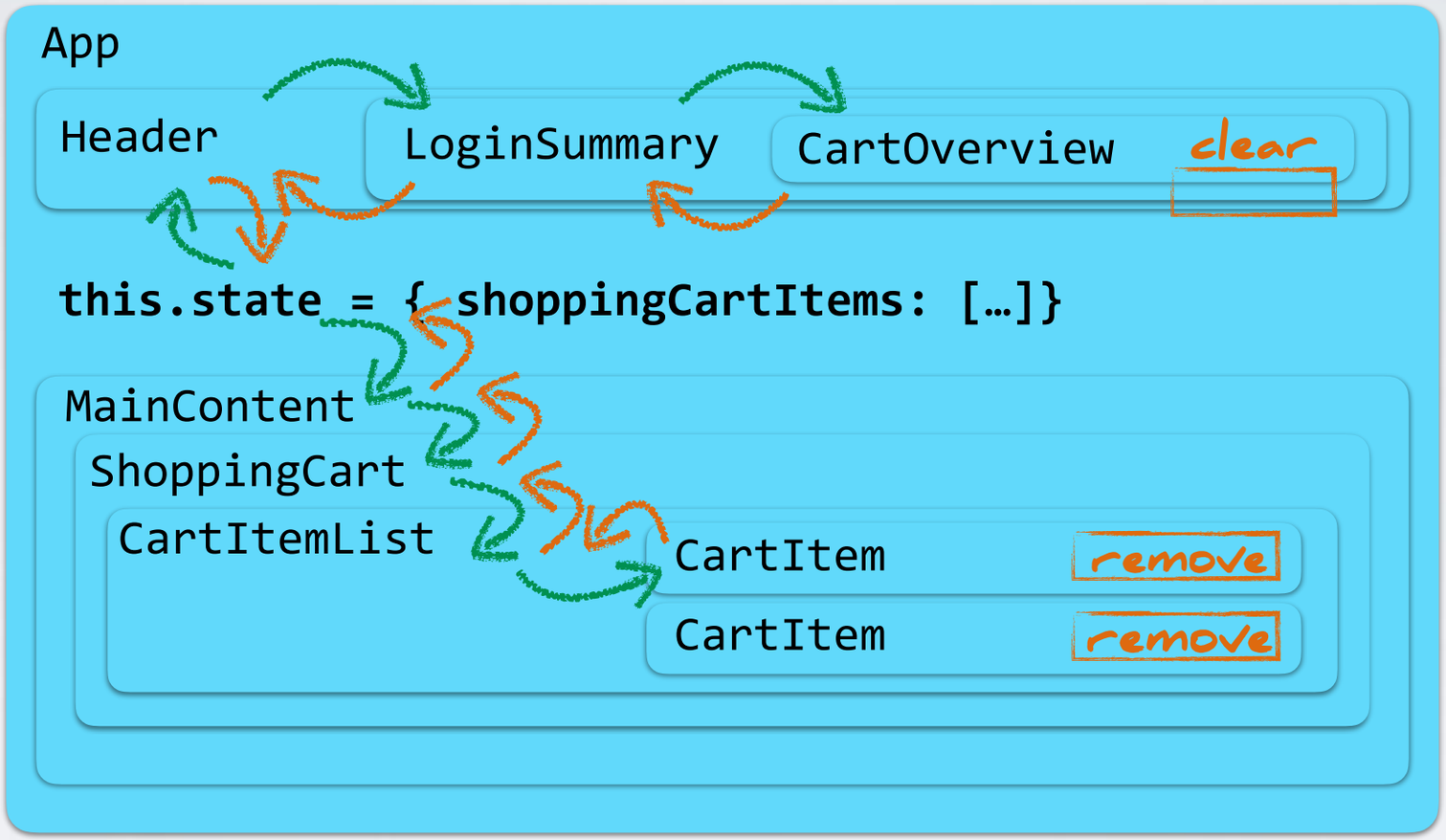
https://github.com/DanWahlin/Observable-Store

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/

"Prop Drilling"

The process you have to go through to pass data and events through the component tree.



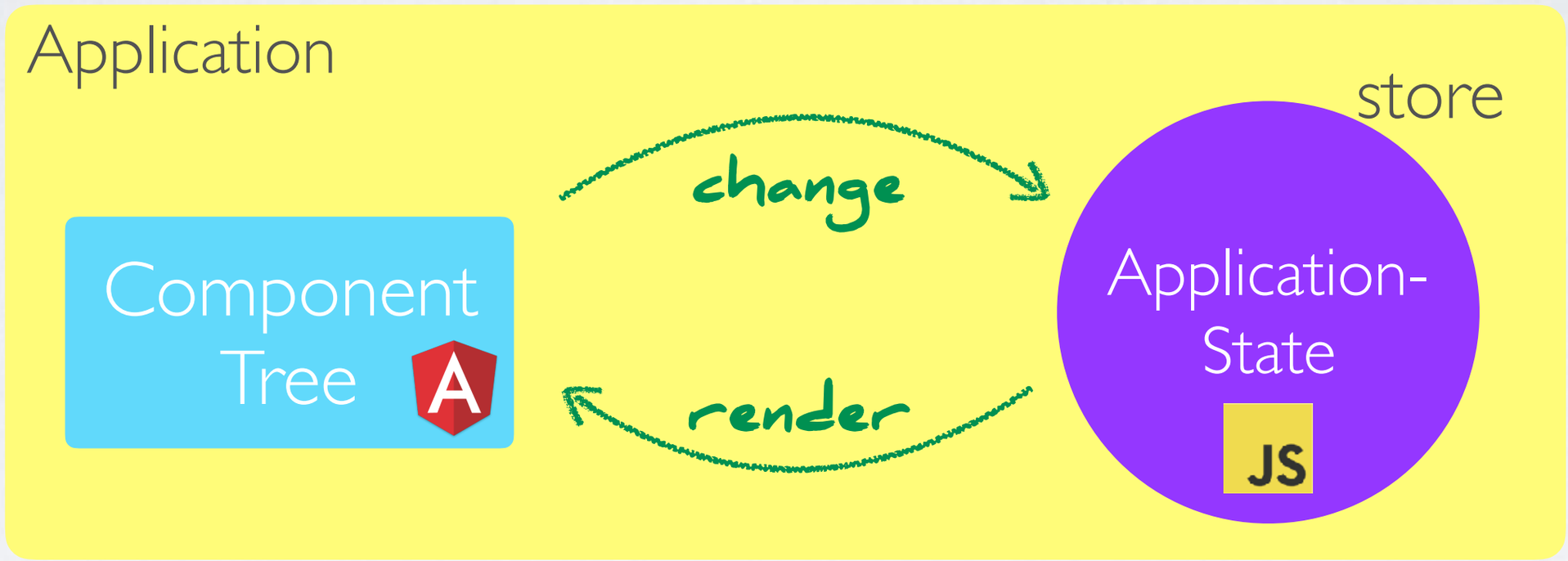
Prop drilling can be a *good thing*: it makes the data-flow very explicit!

Prop drilling can be a *bad thing*: passing data from its holder to a consumer via serveral intermediates is tedious and makes changing the component tree more difficult.

https://blog.kentcdodds.com/prop-drilling-bb62e02cb691

Application with State Container

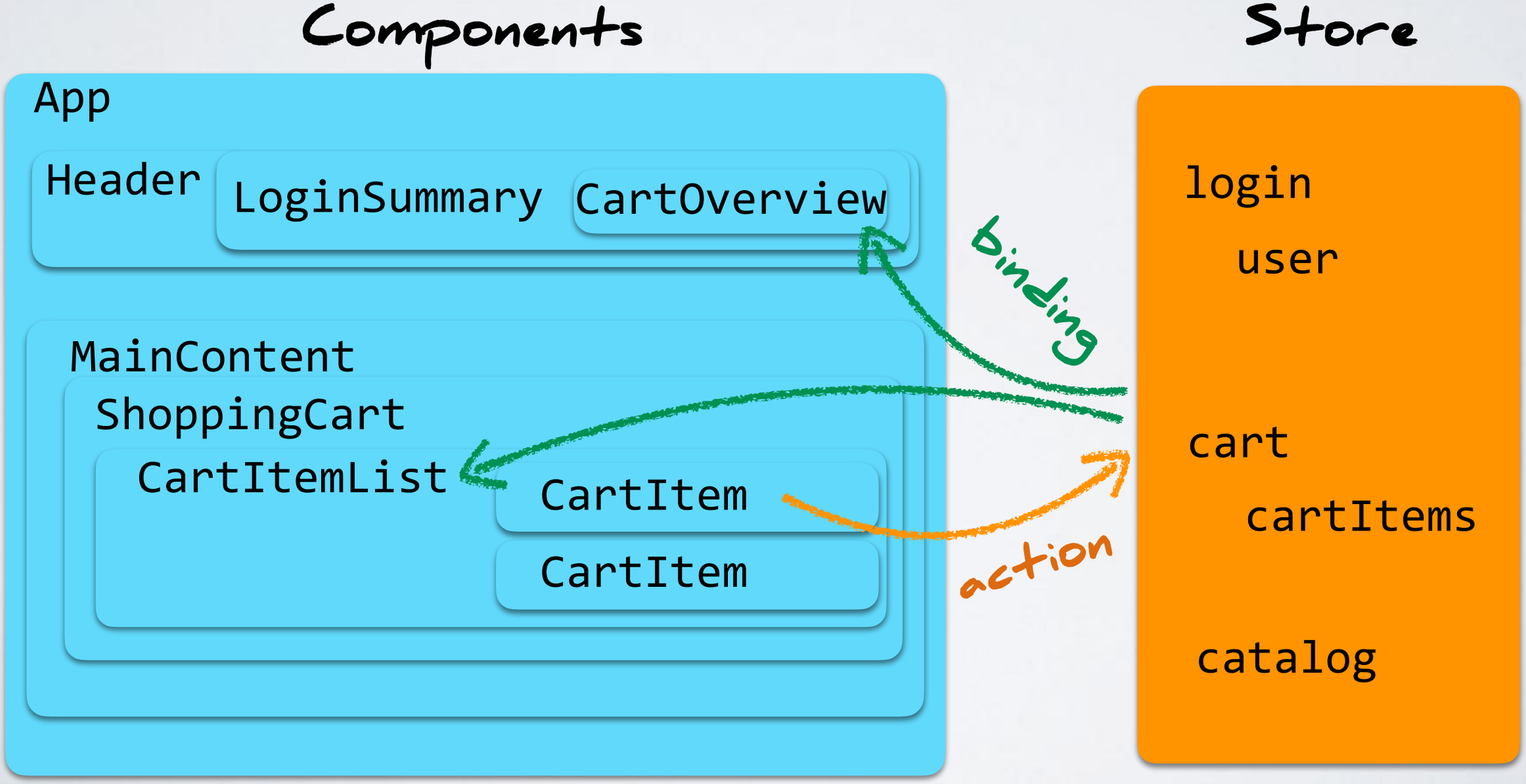
A state container extracts the shared state out of the components, and manages it in a global singleton.



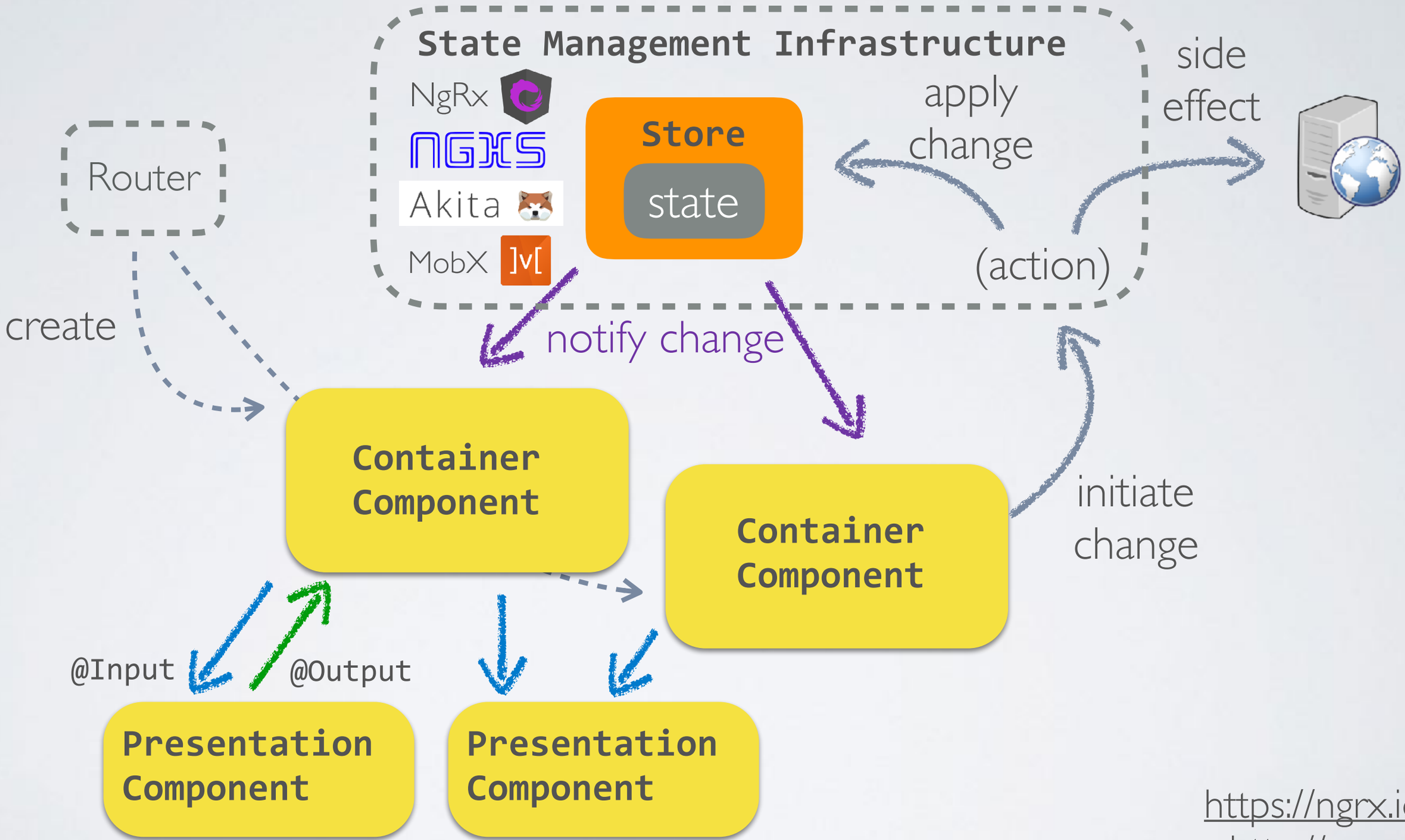
The component tree becomes a big "view", and any component can access the state or trigger actions, no matter where they are in the tree!

Managing State with a State Container

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



State Management: State Container



https://ngrx.io/
http://ngxs.io
https://github.com/datorama/akita
https://mobx.js.org/

Redux Basics: 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.

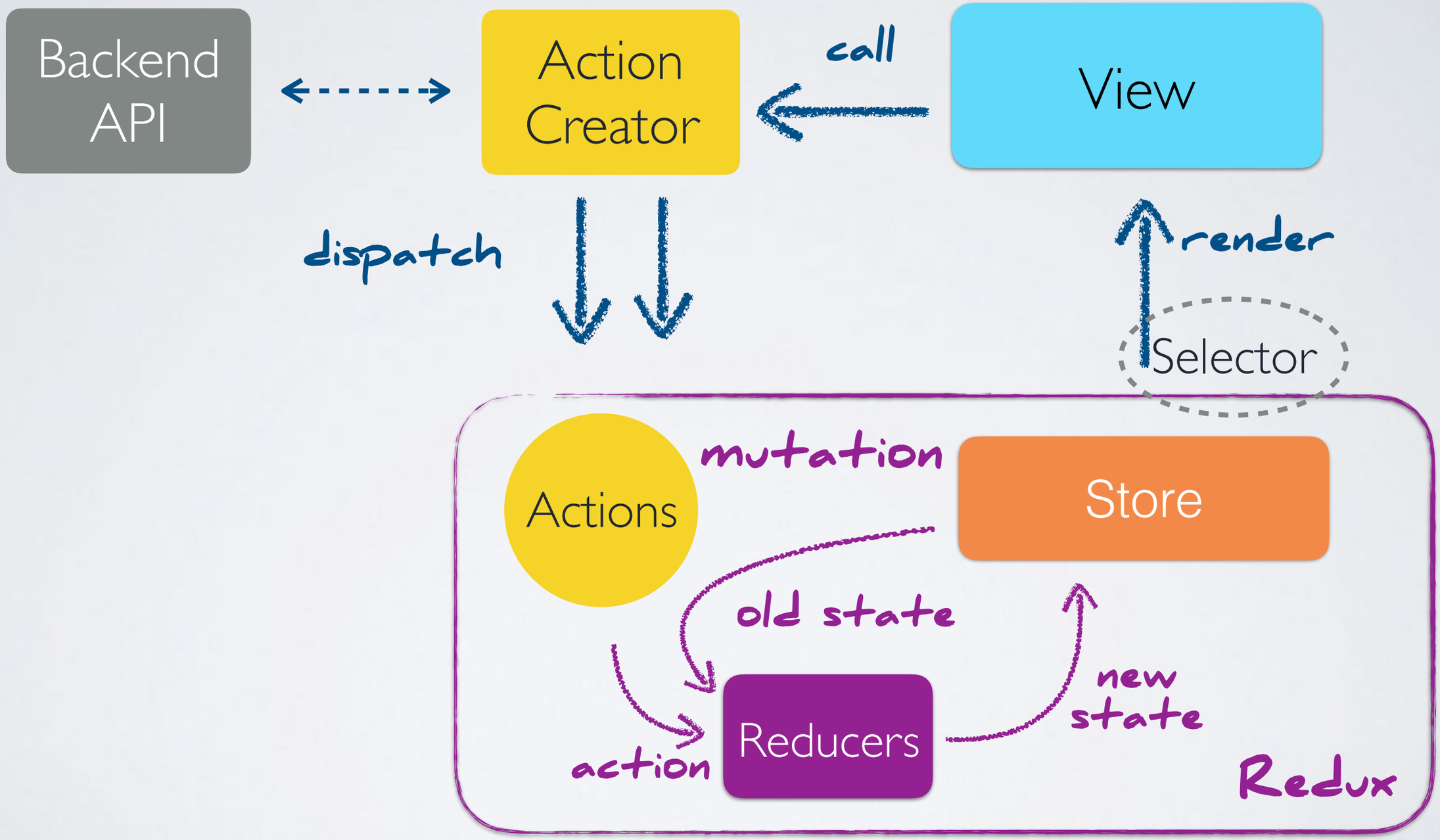
State Reduce

Redux implements store mutations in a "functional way".

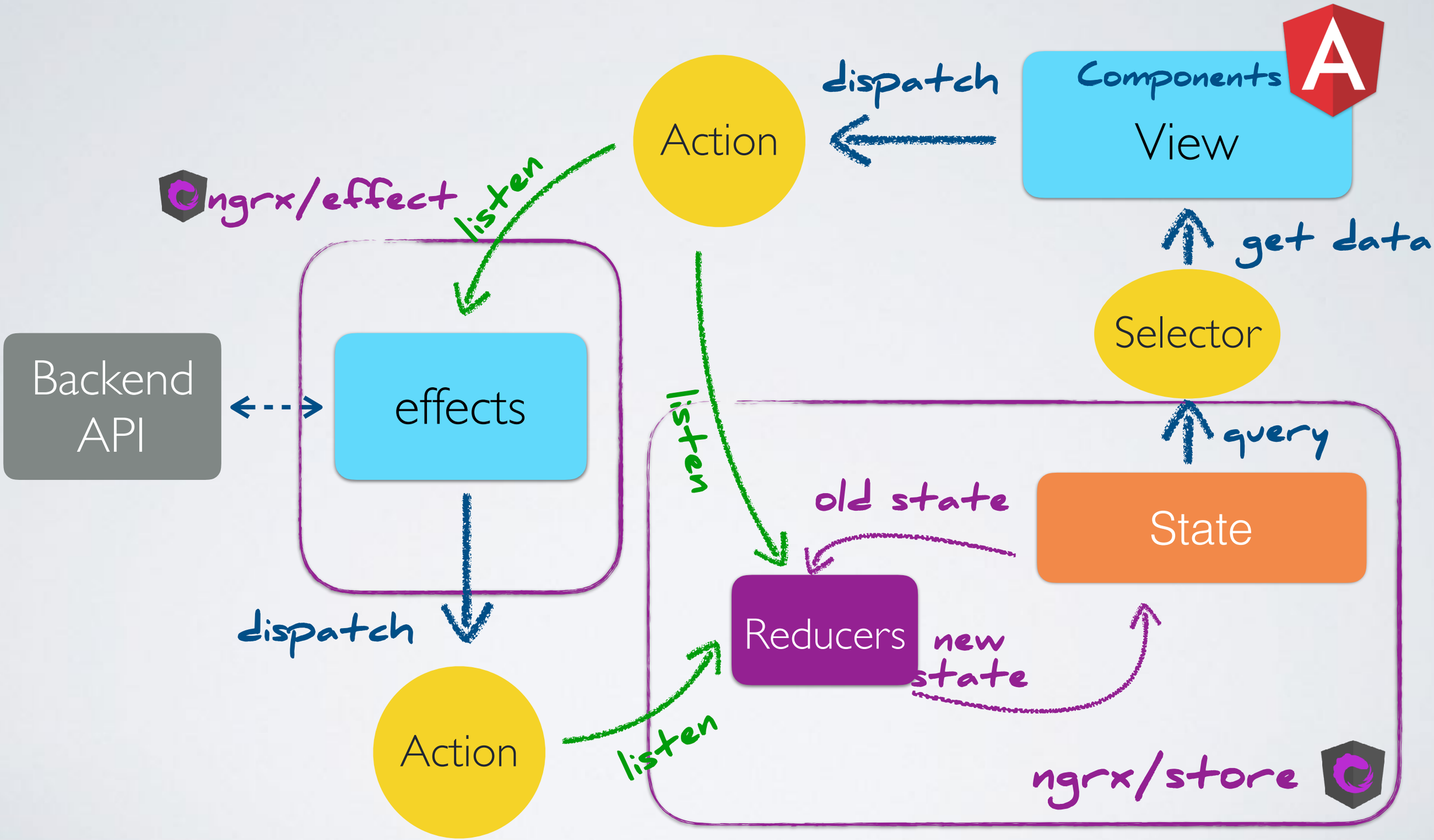
(old state , action) => new state

aka: "reducer function"

Redux Architecture



ngrx Architecture



https://github.com/ngrx/platform

Redux Programming Model

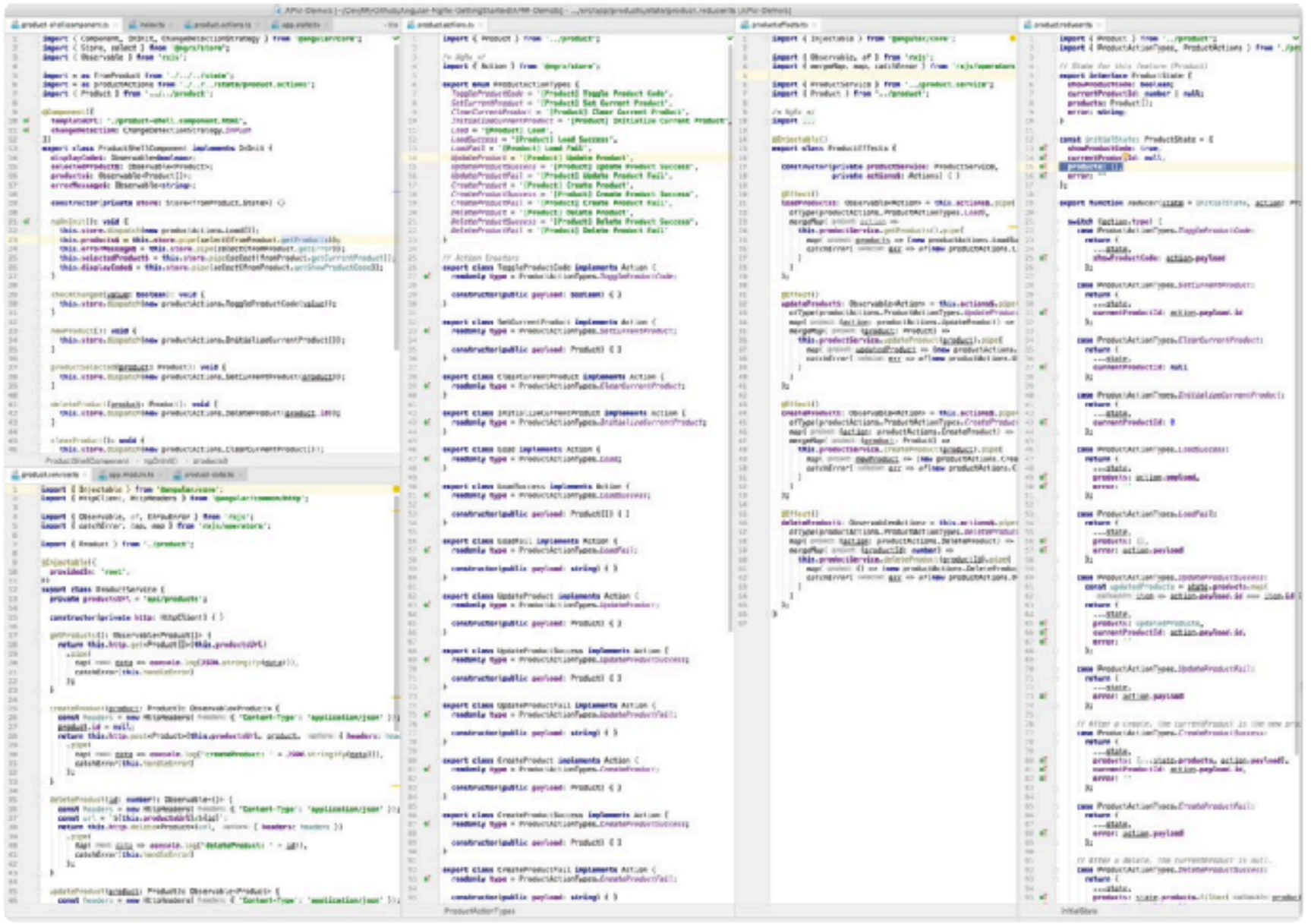
- Single Source of Truth: The state of the entire app is stored in a single object - the *store*

- To change the state, you need to dispatch *actions* that describes what needs to happen.

- You cannot change properties of objects or make changes to existing arrays. In Redux, you must always return a new reference to a new object or a new array.

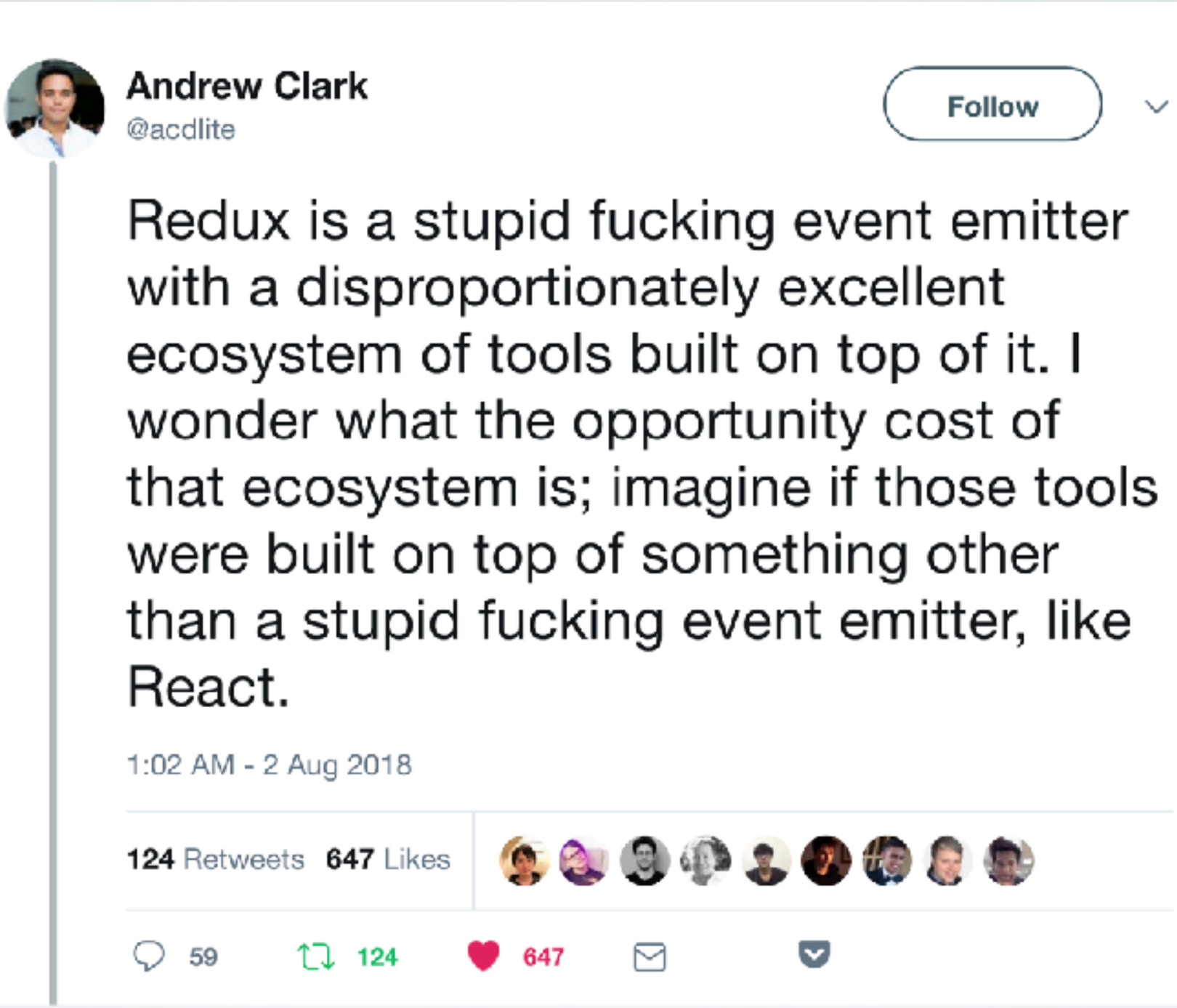
Jonas Bandi
@jbandi

This is what NgRx debugging looks like for me ... I don't think the fragmentation is helpful in any way ...



1:32 PM - 30 Jul 2018

https://twitter.com/jbandi/status/1023954402116358145



Andrew Clark,
one of the original authors of Redux

https://twitter.com/acdlite/status/1024852895814930432

MobX

MobX is a state management library that embraces mutability.
MobX provides a reactivity system that observes object mutations.

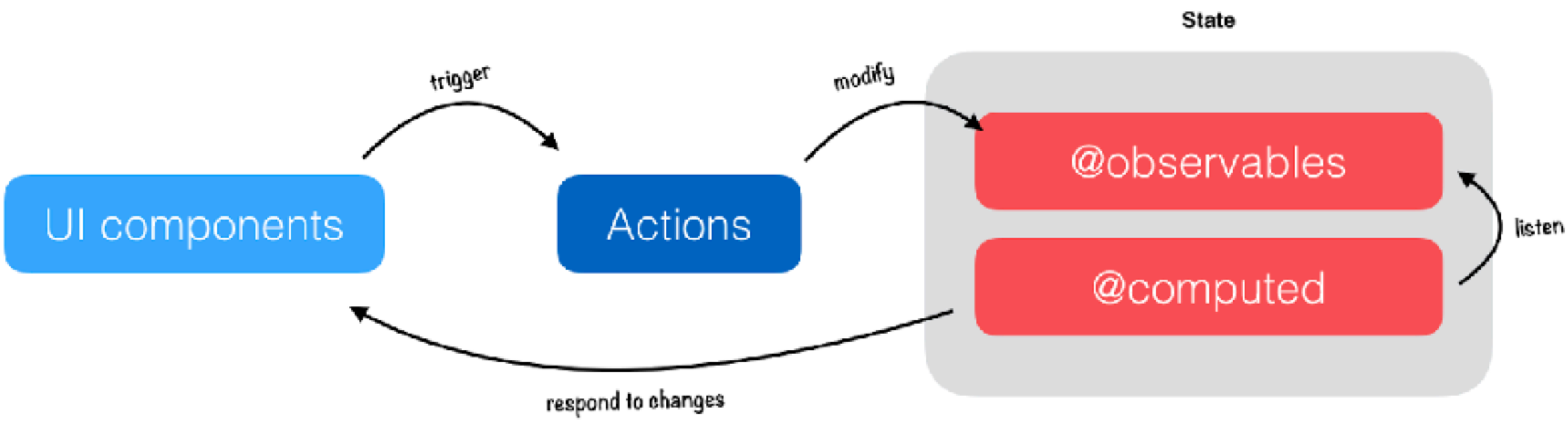
https://mobx.js.org/

Interesting Library: MobX State Tree https://github.com/mobxjs/mobx-state-tree

https://www.robinwieruch.de/redux-mobx-confusion/

MobX Concepts

@observable and @computed make the model reactive:



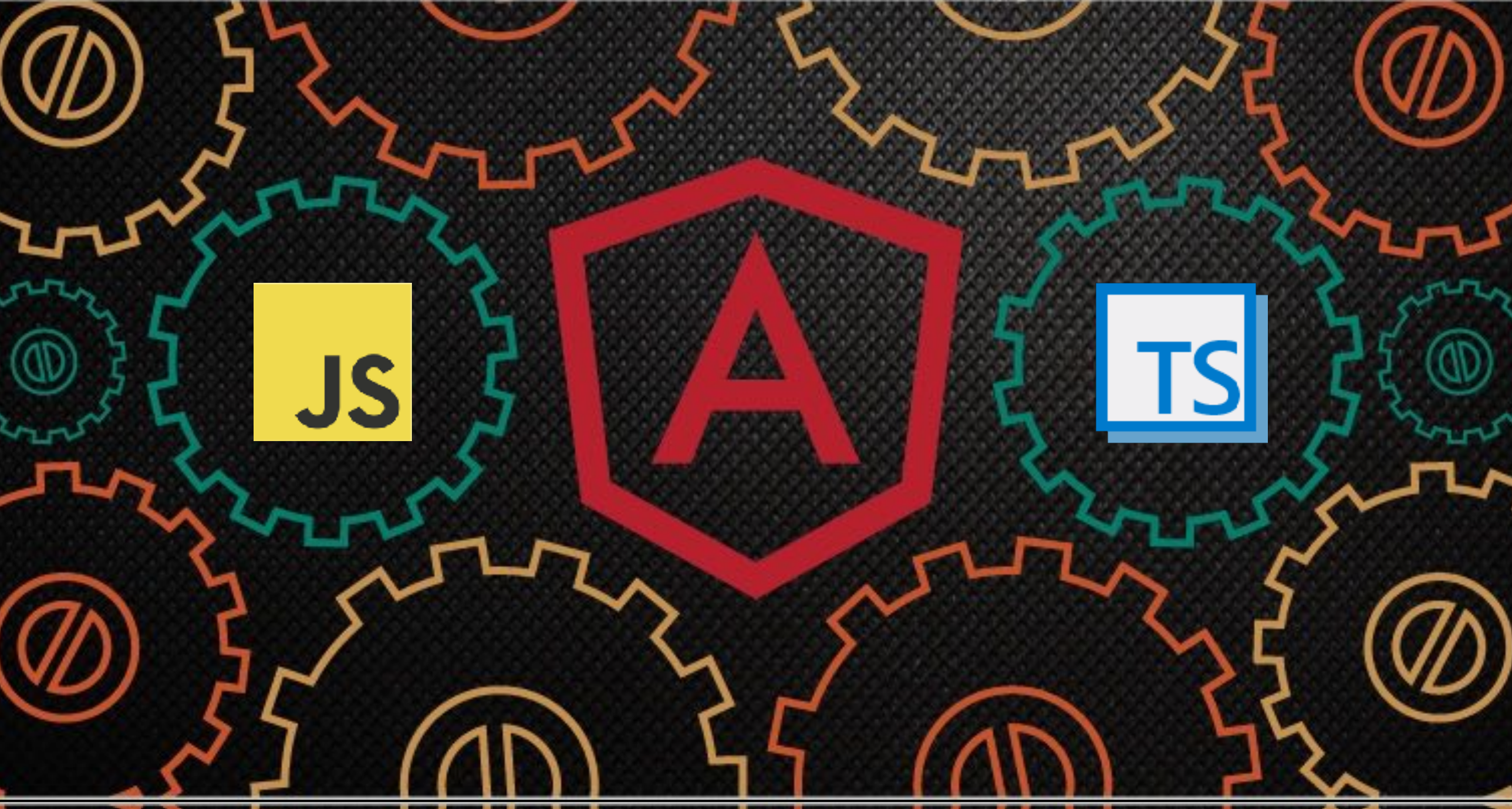
Actions can batch multiple modifications in a "state-change-transaction".

A Reactive Model

```
class Person {
  @observable firstName;
  @observable lastName;

  constructor(firstName, lastName) {
    this.firstName = firstName;
    this.lastName = lastName;
  }

  @computed get fullName() {
    return this.firstName + ' ' + this.lastName;
  }
}
```

```
const person = new Person('John', 'Doe');

const render = () => {
  document.body.innerText = person.fullName;
};

autorun(render);
```

autorun triggers a reaction on each state change



Have fun with Angular!

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

Thank You!

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

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