Angular

The modern web developer’s platform

Kevin Anidjar

Angular Tech Leader @ Edenred

Former B2C Platform CTO @ TotalEnergies

DAY 1

NX	Introduction Monorepo Command-Line Interface Structure
Lazy-loading	Introduction Implementation Preloading Strategy
Optimization	Standalone Components ChangeDetectorStrategy / ChangeDetectorRef ngFor* trackBy Budgets RxJS Subscriptions
RouteGuards & Resolvers	Introduction CanActivate / CanActiveChild CanDeactivate Pre-fetching resources

DAY 2

NgRx (Redux flow)	Introduction State management (store) Side-Effects Debugging
Reactive Forms	Custom controls Async Validators Custom validators
Cypress	Introduction Write Tests
Server-side rendering	Introduction Usage

Introduction

Nx is a **build system** with monorepo support and plugins extensions

Nx plugins, although very useful for many projects, **are completely optional**

Nx **caches the output of any previously run command** such as testing and building, so it can **replay the cached results instead of rerunning it**

Nx Cloud allows you to **share the computation cache across everyone in your team and CI**

Deep integration with tools like Cypress, Storybook, and Jest.

Monorepo

A monorepo is a **single git repository** that **holds the source code for multiple applications and libraries**, along with the tooling for them.

Share code and visibility: reuse validation code, UI components, and types across the codebase. Reuse code between the backend, the frontend, and utility libraries.

Atomic changes: you can change a button component in a shared library and the applications that use that component in the same commit.

Single set of dependencies: use a single version of all third-party dependencies, reducing inconsistencies between applications.

Command-line interface (CLI)

```
npm install -g nx
```

The **Nx CLI** provides commands to operate and manage the different parts of the codebase. These commands fall into three categories.

1. Acting on Code (executors)

The nx run command executes a target on a single project. For convenience, you can also run nx [target] [project] which is an alias to nx run [project]:[target].

```
nx run my-js-app:build
nx build my-js-app
```

```
nx run-many --target=build --projects=app1,app2
nx run-many --target=test --all
```

Command-line interface (CLI)

2. Modifying Code (generators)

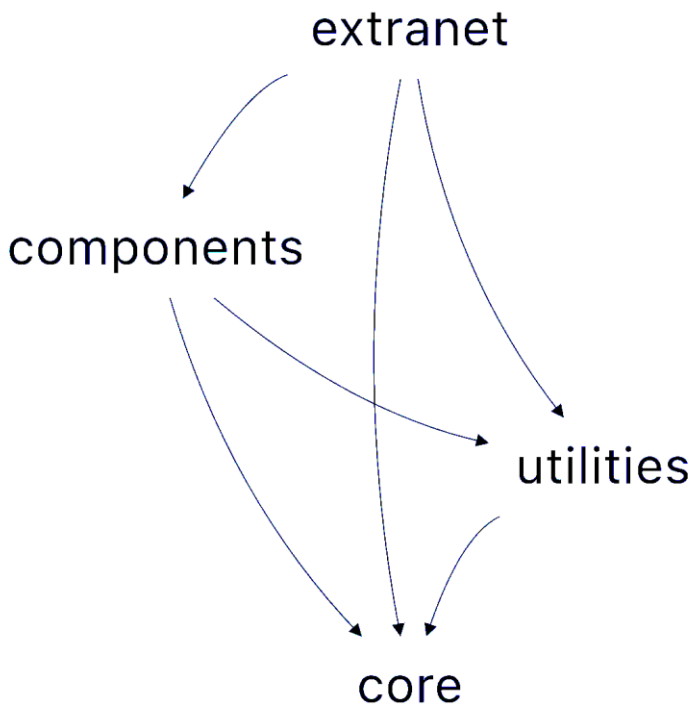
```
nx generate @nrwl/angular:library my-lib
nx generate @nrwl/angular:component my-component
```

3. Understanding the codebase

Nx creates and maintains a project graph between projects based on import statements in your code and uses that information to run executors only on the affected projects

Command-line interface (CLI)

```
nx graph extranet
```



Project	Dependencies
Extranet	Components, Utilities, Core
Components	Utilities, Core
Utilities	Core

Command-line interface (CLI)

Create a new workspace

```
npx create-nx-workspace --preset=angular
```

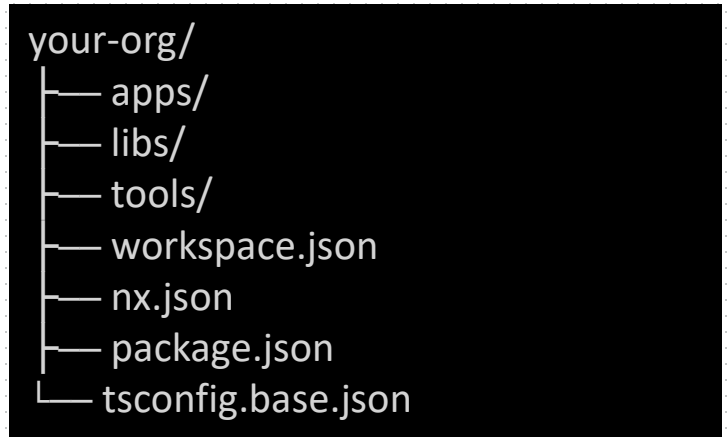
An Angular workspace will be created including the following dependencies:

Dependency	Usage
Storybook	Component building
Jest	Unit tests
Cypress	e2e
ESLint	Linting

Structure

/apps/ contains the application projects.

/libs/ contains the library projects. Each library defines its own external API so that boundaries between libraries remain clear.



/tools/ contains scripts that act on your code base. This could be database scripts, custom executors, or workspace generators.

/workspace.json lists every projects in your workspace

/nx.json configures the Nx CLI itself. It tells Nx what needs to be cached, how to run tasks etc.

Lazy-loading

Introduction

By default, modules are **eagerly loaded**, which means that as soon as the application loads, so do **all the modules**, whether or not they are immediately necessary.

For large applications with lot of routes, consider lazy loading

Lazy-loading is a design pattern that **loads modules as needed**.

It helps keep initial bundle sizes smaller, which in turn helps **decrease load times**.

Introduction

Initial Chunk Files	Names	Size
main.f3bb0d415bd537ee.js	main	4.01 MB
styles.81e8e2d8d261377c.css	styles	357.01 kB
scripts.302de51e5f7a435d.js	scripts	223.34 kB
polyfills.7a242ff82af945ea.js	polyfills	36.21 kB
runtime.34bf4dc7d495d042.js	runtime	4.75 kB
	Initial Total	**4.62 MB**

Lazy Chunk Files	Names	Size
7763.a6fa85846f309262.js	-	253.59 kB
9175.be8542562e0f6e2d.js	-	250.34 kB
3307.329c7a27c66c2077.js	-	229.33 kB
2923.8622aba1448ca428.js	-	209.15 kB
4794.5538bc46638cdf61.js	-	195.39 kB
9539.654fc5de5389db91.js	-	186.91 kB
368.8cd761cdfff904f9.js	-	176.78 kB
8591.bdefe8e9f6612749.js	-	155.33 kB
5209.b8287e8fd70c968d.js	-	147.28 kB
8604.bde0dfe892465e02.js	-	130.14 kB
8248.a4a2c431a6eb6d2d.js	-	126.74 kB
3002.0244ba46e6e04246.js	-	120.76 kB
8556.90fd053b9222e325.js	-	119.15 kB
3599.0e35d077f3c3d052.js	-	115.47 kB
3974.0259119688552e09.js	-	111.18 kB
6107.0cbd6db7280473df.js	-	102.41 kB
2158.418501fdd75305df.js	-	98.17 kB
778.948ab9c5c56f3296.js	-	97.04 kB

2957.eff2e9499ad3d50a.js	-	96.96 kB
314.dc9a86b21ddb9dbf.js	-	96.59 kB
4176.1e8db90e874cbe2a.js	-	94.68 kB
8582.fc26be0ac1e11f7b.js	-	85.25 kB
7035.0cb500e3664eb7df.js	-	80.63 kB
3845.6864a373c53fde97.js	-	79.38 kB
5564.1afd8384aeef93a6.js	-	77.09 kB
9711.4538c94a0edaa8cf.js	-	72.83 kB
9942.ceb02da789143836.js	-	67.69 kB
6382.9b82cb92cbc488b3.js	-	60.60 kB
6766.5d5c12600a3b7287.js	-	58.95 kB
6796.d60a7942877db2ca.js	-	44.58 kB
5316.be2725d906f67192.js	-	36.06 kB
7175.0cc891b1fc7c9809.js	-	35.70 kB
9556.c69a04c11c9b74fb.js	-	27.44 kB
9456.9d0a5785a1d3937b.js	-	24.95 kB
9654.33273f675672244e.js	-	23.56 kB
5563.0112936bc4805d78.js	-	22.80 kB
2094.a9a0da5c6395b80c.js	-	19.08 kB
9160.17259b04768806f9.js	-	18.70 kB
4330.0384671131177e6a.js	-	15.90 kB
6079.3e18c97d2dbc7c41.js	-	15.28 kB
8844.e4d3faef8d454e4a.js	-	14.92 kB
6741.09a24afc02391428.js	-	13.82 kB

Without lazy-chunks, « main.js » would weight more than 8 MB.

Implementation

CustomersRoutingModule

```
const routes: Routes = [
 {
  path: '',
  component: CustomersComponent
 }
];
```

Define routes in CustomersRoutingModule

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

Import RouterModule with routes

Implementation

AppRoutingModule

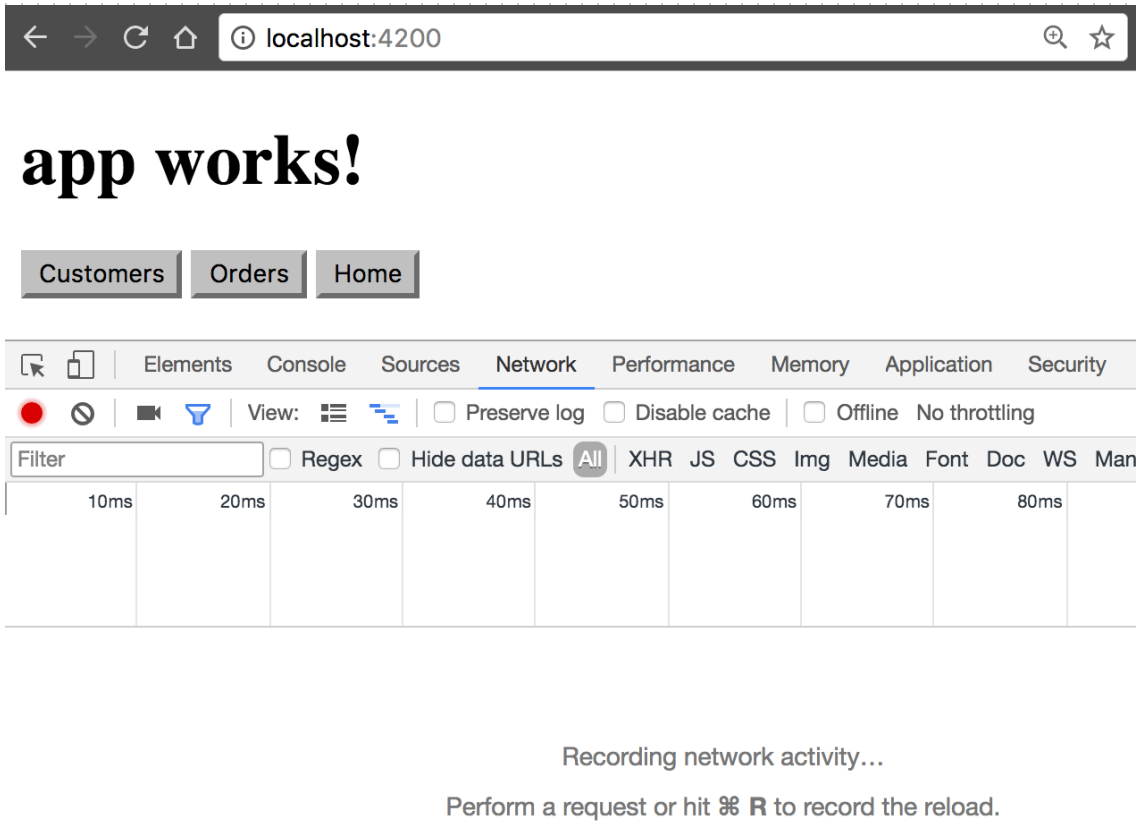
```
const routes: Routes = [
 {
   path: 'customers',
   loadChildren: () => import('./customers/customers.module').then(m => m.CustomersModule)
 }
];
```

Import lazy-loaded modules

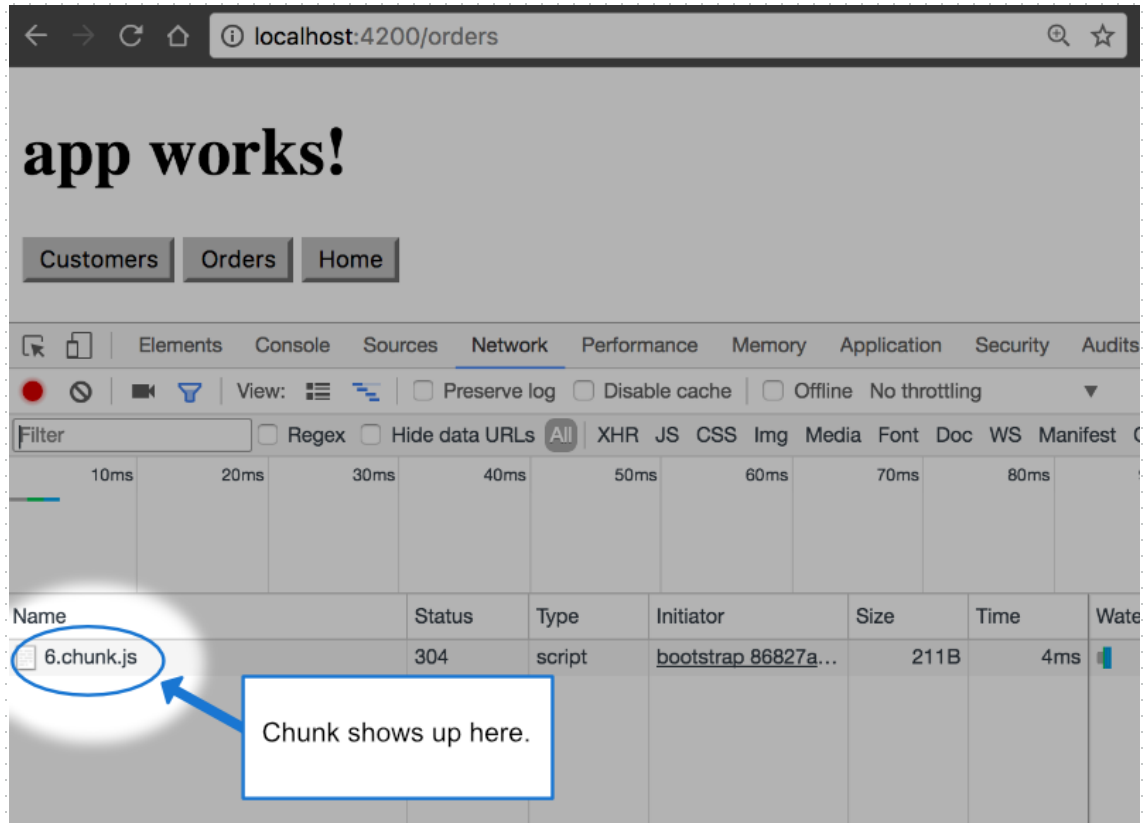
```
@NgModule({
imports: [RouterModule.forRoot(routes)],
exports: [RouterModule]
})
export class AppRoutingModule { }
```

Customer module is lazy-loaded and available on « http://localhost:4200/customers »

Implementation



Before clicking « Orders » button
No module loaded



After clicking « Orders » button
OrdersModule is loaded

Preloading Strategy

Preloading is a mechanism to further optimize loading time by loading modules in the background.

This technique is used with lazy loading, so you need to configure lazy loading before using any preloading strategies.

Built-in preloading strategies

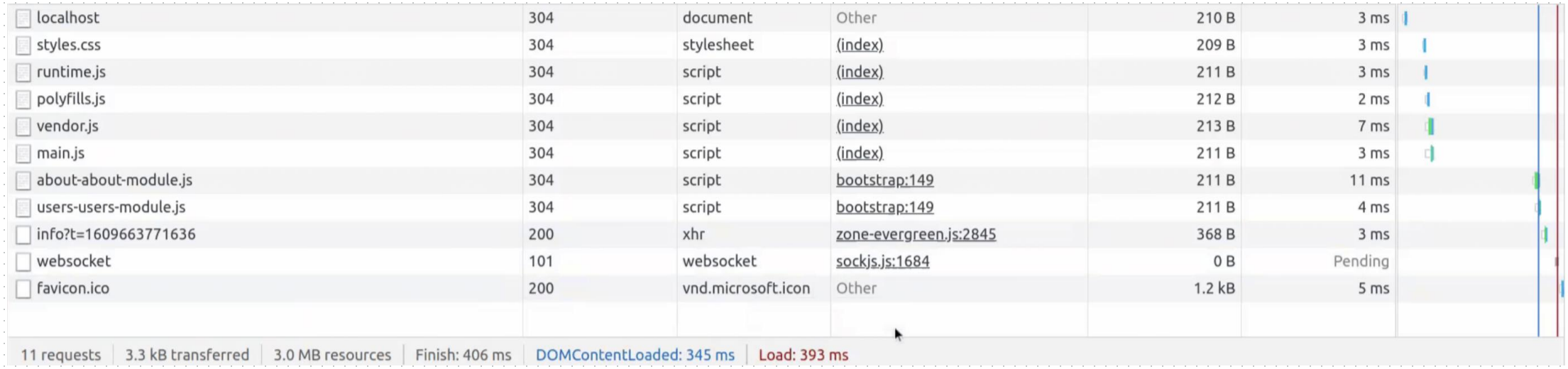
NoPreloading (default) or PreloadAllModules.

Custom preloading strategies

Preload after some time, preload based on network quality, load required modules first, frequently used second, etc..

Preloading Strategy

PreloadAllModules



localhost	304	document	Other	210 B	3 ms
styles.css	304	stylesheet	(index)	209 B	3 ms
runtime.js	304	script	(index)	211 B	3 ms
polyfills.js	304	script	(index)	212 B	2 ms
vendor.js	304	script	(index)	213 B	7 ms
main.js	304	script	(index)	211 B	3 ms
about-about-module.js	304	script	bootstrap:149	211 B	11 ms
users-users-module.js	304	script	bootstrap:149	211 B	4 ms
info?t=1609663771636	200	xhr	zone-evergreen.js:2845	368 B	3 ms
websocket	101	websocket	sockjs.js:1684	0 B	Pending
favicon.ico	200	vnd.microsoft.icon	Other	1.2 kB	5 ms

11 requests | 3.3 kB transferred | 3.0 MB resources | Finish: 406 ms | DOMContentLoaded: 345 ms | Load: 393 ms

When using the PreloadAllModules strategy, Angular loads all the modules in the background after the eager modules are loaded.

Preloading Strategy

Custom Preloading Strategies

If the number of modules in your application is large, preloading all the modules might not be the best solution. In that case, we can configure custom preloading strategies.

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

Provide the router with your « CustomPreloadingStrategyService »

Preloading Strategy

```
const routes: Routes = [
  {path: 'about', data: { preload: true }, loadChildren: () => import('./about/about.module').then(m => m.AboutModule)},
  {path: 'users', loadChildren: () => import('./users/users.module').then(m => m.UsersModule)},
];
```

Add a « preload » property to routes we want to preload

```
@Injectable()
export class CustomPreloadingStrategyService implements PreloadingStrategy {
  public preload(route: Route, preloadRoute: () => Observable<any>): Observable<any> {
    if (route.data?.preload) {
      return preloadRoute();
    }
    return of(null);
  }
}
```

Preload only routes with truthy preload property

Let’s code!

Open and read « apps/1-angular-lazyloading/README.md »

Optimization

Standalone Components

Angular classes marked as standalone do not need to be declared in a module.

Standalone components specify their dependencies directly instead of getting them through a module

```
@Component({
  standalone: true,
  selector: 'standalone-component',
  imports: [AnotherStandaloneComponent],
  template: `<another-standalone-component></another-standalone-component>`,
})
export class StandaloneComponent {
}
```

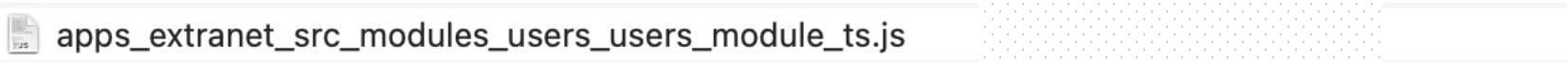
Standalone Components

```
@NgModule({
 imports: [
  RouterModule.forChild([
   {
    path: 'standalone-component',
    loadComponent: () =>
     import('./components/standalone.component').then(m => m.StandaloneComponent)
   }
  ])
 ]
})
class MyModule {
}
```

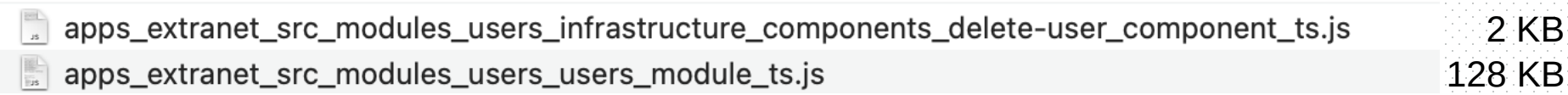
Import component as a lazy-loaded module

Standalone Components

Standard component embedded in « Users » module with many other components



Lazy-loaded standalone component



DeleteUserComponent only weights 2kb and can be loaded instantly when needed.

ChangeDetectorStrategy / ChangeDetectorRef

Change detection is the process through which Angular checks to see whether your application state has changed, and if any DOM node needs to be updated

By default, Angular performs change detection on all components (from top to bottom) **every time something changes in your app**.

A change can occur from a user event or data received from a network request.

Change detection is very performant, but as an app gets more complex and the amount of components grows, change detection will have to perform more and more work.

One solution is to use the **OnPush change detection strategy** for specific components.

This will instruct Angular to run change detection on these components and their sub-tree only **when new references are passed to them** versus when data is mutated.

ChangeDetectorStrategy / ChangeDetectorRef

Default change detection

Angular decides if the view needs to be updated by **comparing all the template expression values before and after** the occurrence of an event, for all components of the component tree

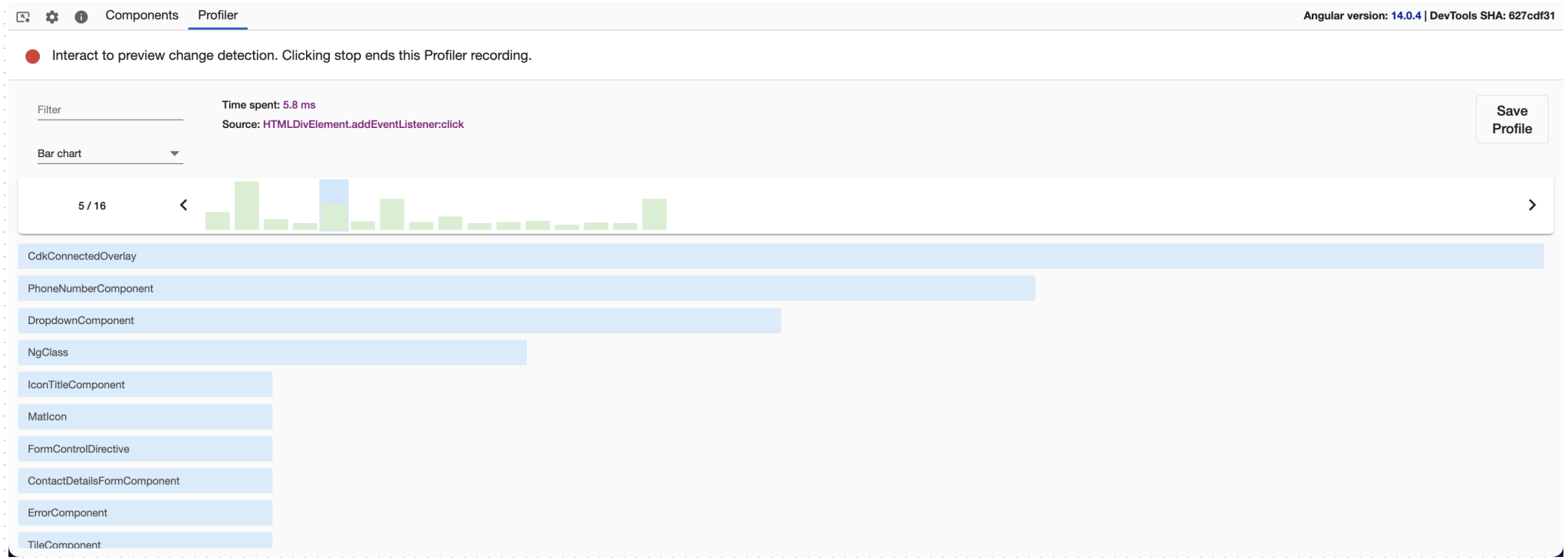
By default, Angular **does not do deep object comparison to detect changes**, it only takes into account **properties used by the template**

```
@Component({
  selector: 'todo-item’,
  template: `<span>{{todo.owner.firstname}} {{todo.description}} {{todo.done}}</span>`
})
export class TodoItem {
  @Input() todo: Todo;
}
```

ChangeDetectorStrategy / ChangeDetectorRef

Angular DevTools

You can use the Angular DevTools extension to track change detections.



You can see a sequence of bars, each one of them symbolizing change detection cycles in your app. The taller a bar is, the longer your application has spent in this cycle

ChangeDetectorStrategy / ChangeDetectorRef

OnPush strategy

```
@Component({
  selector: ‘app-todo’,
  template: `<span>{{todo.owner.firstname}} {{todo.description}} {{todo.completed}}</span>`,
  changeDetection: ChangeDetectionStrategy.OnPush
})
export class TodoComponent{
  @Input() todo: Todo;

  public constuctor(private _cdr: ChangeDetectorRef) {}

  public detechChanges() {
    this._cdr.detectChanges();
  }
}
```

We set the changeDetection option to « ChangeDetectionStrategy.OnPush ». Change detection automatically triggers on the component and its children when the reference of the input changes or manually via detectChanges()

Let’s code!

Open and read « apps/2-angular-changedetection/README.md »

*ngFor trackBy

The *ngFor directive needs to **uniquely identify items** in the iterable to correctly perform DOM updates when items in the iterable are reordered, new items are added, or existing items are removed.

In all of these scenarios it is usually desirable to **only update the DOM elements associated with the items affected by the change**.

The *ngFor directive allows to pass a trackBy function telling the directive **to track items on a specific property**.

By default, the directive use the « Object.is() » method to tell whether an item has changed or not.

*ngFor trackBy

```
<li *ngFor="let user of users; let index = index; trackBy: trackById">
  {{ user.id }} {{ user.name }}
</li>
```

```
public trackById(index: number, user: User): number {
  return user.id;
}
```

When an update occurs, the directive will check if an id has changed. If so, the list will be **deleted from the DOM then recreated**.

Otherwise, only the **properties that have changed will be updated**.

Let’s code!

Open and read « apps/3-angular-trackby/README.md »

Budgets

Angular budgets allow us to configure expected sizes of bundles

We can configure thresholds for conditions when we want to receive a warning or even fail build with an error if the bundle size gets too out of control

Angular budgets are defined in the angular.json file. Budgets are defined per project which makes sense because every app in a workspace has different needs.

It only makes sense to define budgets for the production builds. Prod build creates bundles with « true size » after applying all optimizations like tree-shaking and code minimization.

Budgets

```
{
 "configurations": {
  "production": {
   "budgets": [
    {
     "type": "bundle",
     "name": "vendor",
     "baseline": "750kb",
     "warning": "100kb",
     "error": "200kb"
    }
   ]
  }
 }
}
```

The **warning** and **error** properties specify how much can the bundle size deviate from its baseline.

For example bundle with the baseline of **750kb** will trigger warning of **100kb** only if its size **is more than 850kb or less than 650kb**. The warning then acts as a both min and max threshold.

Budgets

```
angular-ngrx-material-starter (master) $ npm run build:prod

> angular-ngrx-material-starter@6.5.1 build:prod C:\projects\github\angular-ngrx-material-start
er
> ng build --prod --build-optimizer --vendor-chunk --common-chunk


Date: 2018-08-01T09:37:56.671Z
Hash: bce583cc4aaf16e0fb0b
Time: 82541ms
chunk {0} 0.b1d72e084945c020e0a5.js () 71.3 kB  [rendered]
chunk {1} runtime.b289b5dc646a31db2380.js (runtime) 1.84 kB [entry] [rendered]
chunk {2} vendor.886e90f7c9277f3ac754.js (vendor) 969 kB [initial] [rendered]
chunk {3} styles.434d1d046e291450295f.css (styles) 287 kB [initial] [rendered]
chunk {4} polyfills.a8dc43efeccde4fe998f.js (polyfills) 166 kB [initial] [rendered]
chunk {5} main.9fa19b4dc6789e3c549c.js (main) 130 kB [initial] [rendered]

WARNING in budgets, maximum exceeded for vendor. Budget 850 kB was exceeded by 119 kB.

ERROR in budgets, maximum exceeded for vendor. Budget 950 kB was exceeded by 19.1 kB.
```

Build fails when budget exceeds error threshold

RxJS subscriptions

When an Observable emits a new value, its Observers execute code that was set up during the subscription.

```
obs$.subscribe(data => doSomething(data));
```

If we do not manage this subscription, every time obs$ emits a new value, doSomething will be called. Also, when the user navigates to a new view, doSomething will still be called, potentially causing unexpected results and errors

Subscription management is about knowing when to complete or unsubscribe from an Observable.

We can refer to this management of subscriptions as cleaning up active subscriptions.

RxJS subscriptions

Unsubscribing manually

```
private _subscription : Subscription;

public ngOnInit(): void {
  this._subscription = obs$.subscribe(data => this._doSomething(data));
}

public ngOnDestroy(): void {
  this._subscription.unsubscribe();
}
```

RxJS subscriptions

Unsubscribing multiple subscriptions manually

```
private _subscription: Subscription;

public ngOnInit(): void {
  this._subscription = obs$.subscribe(data => this._doSomething(data));

  this._subscription.add(
     obs$.subscribe(data => this._doSomethingElse(data))
  );
}

  public ngOnDestroy(): void {
    this._subscription.unsubscribe();
  }
```

RxJS subscriptions

Unsubscribing with operators automatically

```
private _subscription: Subscription;

public ngOnInit(): void {
  obs$.pipe(first()).subscribe(data => this._doSomething(data));

  obs$.pipe(take(1)).subscribe(data => this._doSomethingElse(data));
}
```

In both cases, will emit the first emitted value then complete

RxJS subscriptions

Unsubscribing with operators automatically

```
private _notifier$: Subject<null> new Subject();

public ngOnInit(): void {
  obs$.pipe(takeUntil(this._notifier$)).subscribe(data => this._doSomething(data));
  obs$.pipe(takeUntil(this._notifier$)).subscribe(data => this._doSomethingElse(data));
}

public ngOnDestroy(): void {
  this._notifier$.next();
  this_notifier$.complete();
}
```

Will emit values until the component is destroyed

RxJS subscriptions

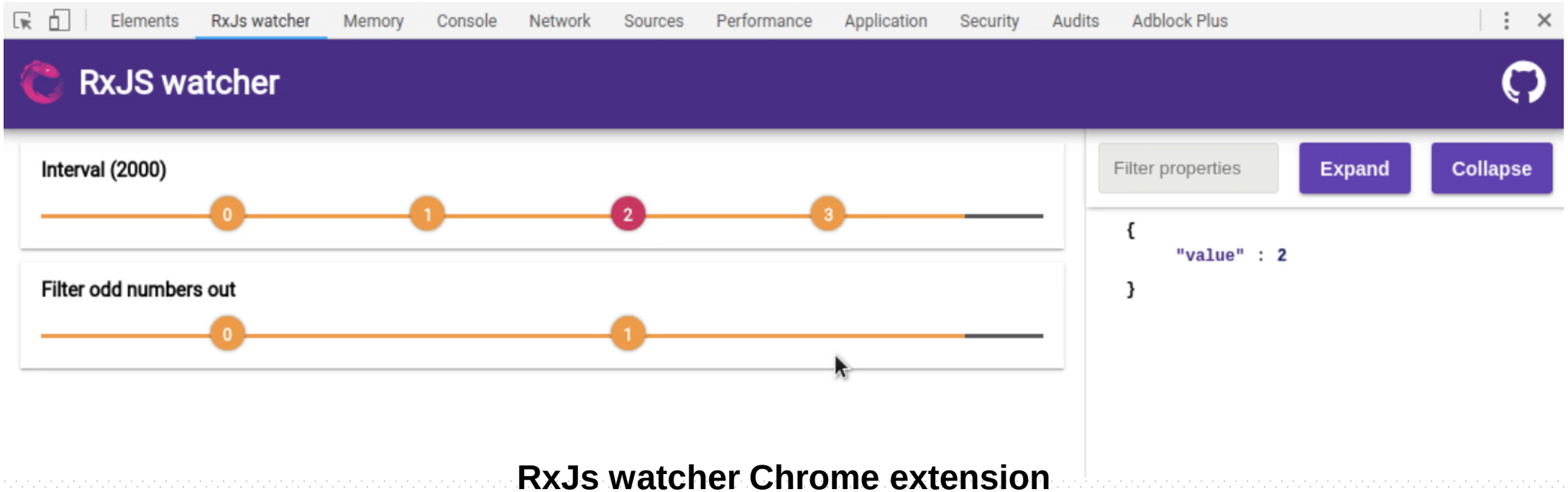
Unsubscribing with operators automatically

```
obs$.pipe(
  takeWhile((data) => data.length === 0)
)
.subscribe(data => this._doSomething(data));
```

Will emit values until condition is met

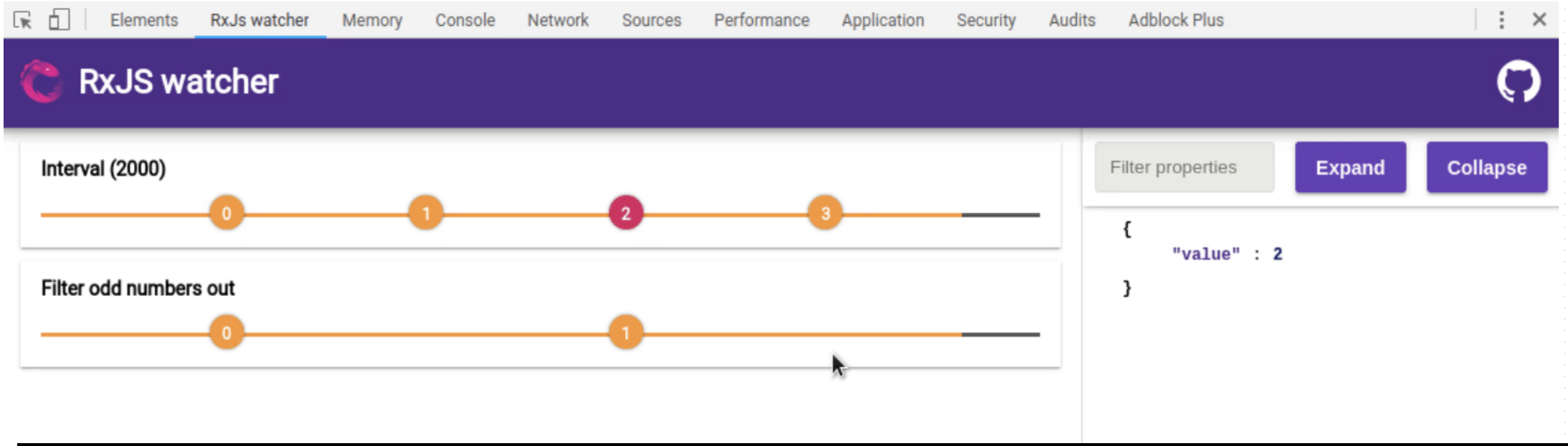
RxJS subscriptions

Visualize Rxjs observables in Chrome



RxJs watcher Chrome extension

RxJS subscriptions




```
import { watch } from "rxjs-watcher";

interval(2000).pipe(
  watch("Interval (2000)", 10),
  filter(v => v % 2 === 0),
  watch("Filter odd numbers out", 10),
).subscribe();
```

Let’s code!

Open and read « apps/4-angular-rxjs-subscriptions/README.md »

Route Guards & Resolvers

Guards

Use route guards to prevent users from navigating to parts of an application without authorization.

Guard	Usage
CanActivate	Guard deciding if a route can be activated
CanActivateChild	Guard deciding if a child route can be activated.
CanDeactivate	Guard deciding if a route can be deactivated
CanLoad	Guard deciding if a route can be loaded

Guards

```
@Injectable()
class IsAdminGuard implements CanActivate {
  constructor(private _user: User, private _router: Router) {}

  public canActivate(
    route: ActivatedRouteSnapshot,
    state: RouterStateSnapshot
  ): boolean | UrlTree {
    return this._user.isAdmin() || this._router.parseUrl('/home’);
  }
}
```

If user is admin, route is activated, otherwise redirect the user to /home

Guards

```
RouterModule.forRoot([
 {
   path: 'finances',
   canActivate: [IsAdminGuard],
   canActivateChild: [HasAccountingRightsGuard],
   component: FinancesPageComponent,
   children: [
    {
      path: ‘transactions',
      component: TransactionsComponent
    },
    {
      path: 'invoices',
      component: InvoicesComponent
    }
   ]
 }
])
```

CanActivate / CanActivateChild

Parent route can be activated if **IsAdminGuard** returns true

Children routes can be activated if **both IsAdminGuard and HasAccountingRightsGuard** return true

Guards

```
RouterModule.forRoot([
 {
  path: 'finances',
  canActivate: [IsAdminGuard],
  component: FinancesPageComponent,
  children: [
   {
    path: ‘transactions',
    component: TransactionsComponent,
    canActivate: [HasAccountingRightsGuard]
   },
   {
    path: 'invoices',
    component: InvoicesComponent,
    canActivate: [HasAccountingRightsGuard]
   }
  ]
 }
])
```

CanActivate / CanActivateChild

Parent route can be activated if **IsAdminGuard** returns true

Children routes can be activated if **both IsAdminGuard and HasAccountingRightsGuard** return true

Guards

```
RouterModule.forRoot([
 {
  path: 'finances',
  canActivate: [IsAdminGuard],
  component: FinancesPageComponent,
  children: [
   {
    path: ‘transactions',
    component: TransactionsComponent,
    canDeactivate: [HasSelectedTransactionGuard]
   },
   {
    path: 'invoices',
    component: InvoicesComponent,
    canActivate: [HasAccountingRightsGuard]
   }
  ]
 }
])
```

CanDeactivate

« transactions » route can be deactivated if **HasSelectedTransactionGuard** returns true

Resolvers

Resolvers let the application fetch remote data from the server before the route we want to reach is activated.

```
@Injectable()
export class ProductsResolver implements Resolve<Observable< Product[]>> {

  public constructor(private _productService: ProductService) {}

  public resolve(route: ActivatedRouteSnapshot): Observable<Product[]> {
    return this._productService.getAll$().pipe(
      catchError(error => of([]))
    );
  }
}
```

Fetch and return an array of products. In case of error, return an empty array

Resolvers

```
RouterModule.forRoot([
 {
   path: 'products’,
   component: ProductsComponent,
   resolve: { products: ProductsResolver }
 }
])
```

Declare resolver in route

Resolvers

```
export class ProductsComponent implements OnInit {

  private _products: Product[];

  public constructor(private activatedRoute: ActivatedRoute) {}

  public ngOnInit(): void {
    this._products = this.activatedRoute.snapshot.data.products;
  }

}
```

Access the products from the ActivatedRoute instance

Let’s code!

Open and read « apps/5-angular-guards-resolvers/README.md »

NgRx

Introduction

NgRx is a global state management for Angular applications, inspired by Redux

NgRx is made up of **5 main components** - Store, Actions, Reducers, Selectors, and Effects.

Store

The Store in NgRx acts as the application's single source of truth. It reflects the current state of the app.

Actions

They express unique events that happen in our application. Actions are how the application communicates with NgRx to tell it what to do.

Introduction

Reducers

Reducers are responsible for handling transitions between states.

They react to the Actions dispatched and executes a pure function to update the Store.

Pure functions are functions that are predictable and have no side effects. Given the same set of inputs, a pure function will always return the same set of outputs.

Selectors

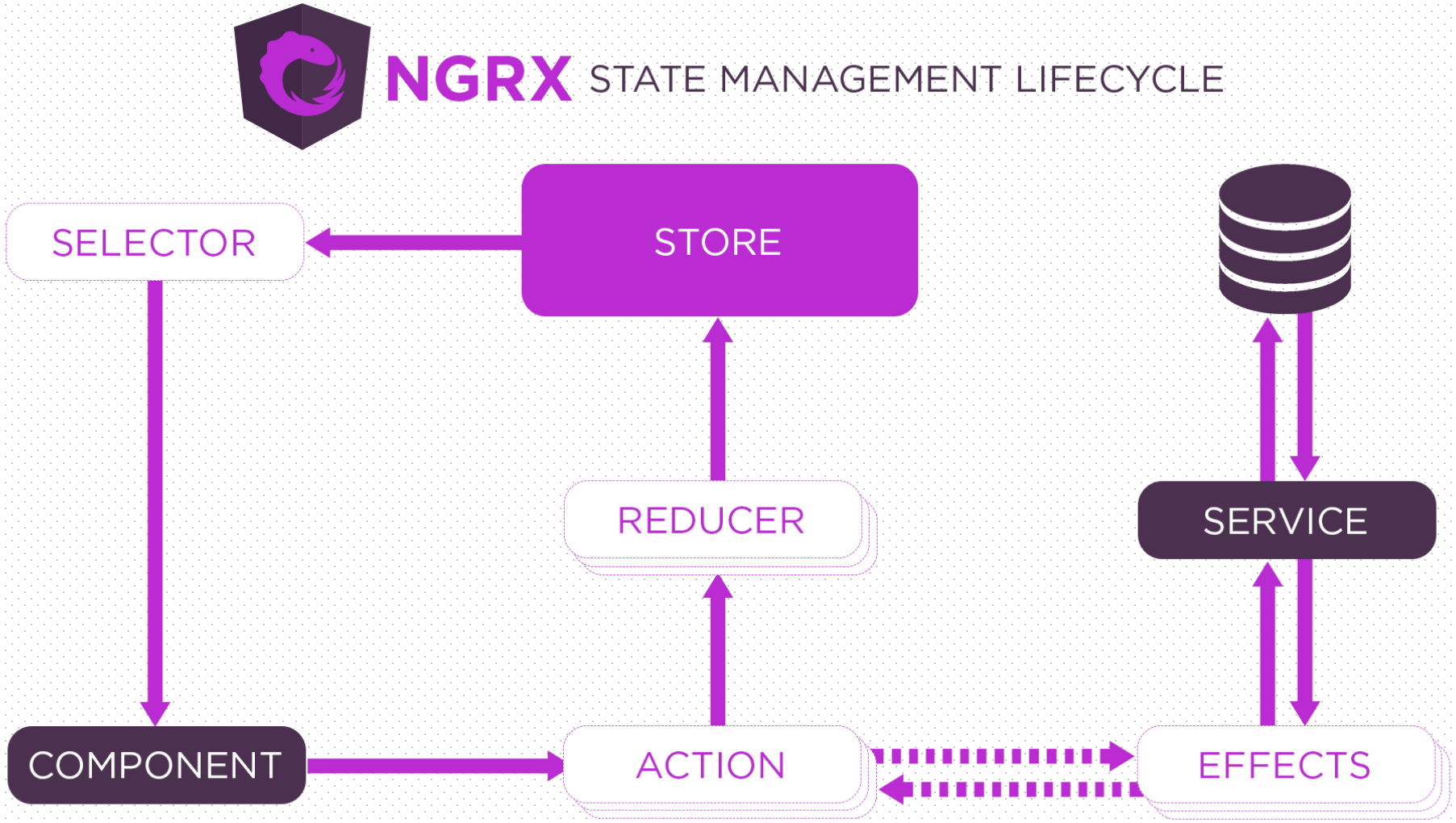
Selectors are pure functions for getting slices of the state from the Store. Selectors are how our application can listen to state changes.

Introduction

Effects

Effects handle the side effects of each Action. These side effects range from communicating with an external API via HTTP when a certain Action is dispatched to dispatching another Action to update another part of the State.

Introduction



Redux Devtools

Debug Redux application's state changes

Chrome extension: [download here]

Store

```
export interface ApplicationState {
  posts: IPost[];
  comments: IComment[];
}
```

Application state definition

```
export const postsSelector = createFeatureSelector<IPost[]>('posts');
export const commentsSelector = createFeatureSelector<IComment[]>('comments');
```

Feature selectors (subtrees)

Store

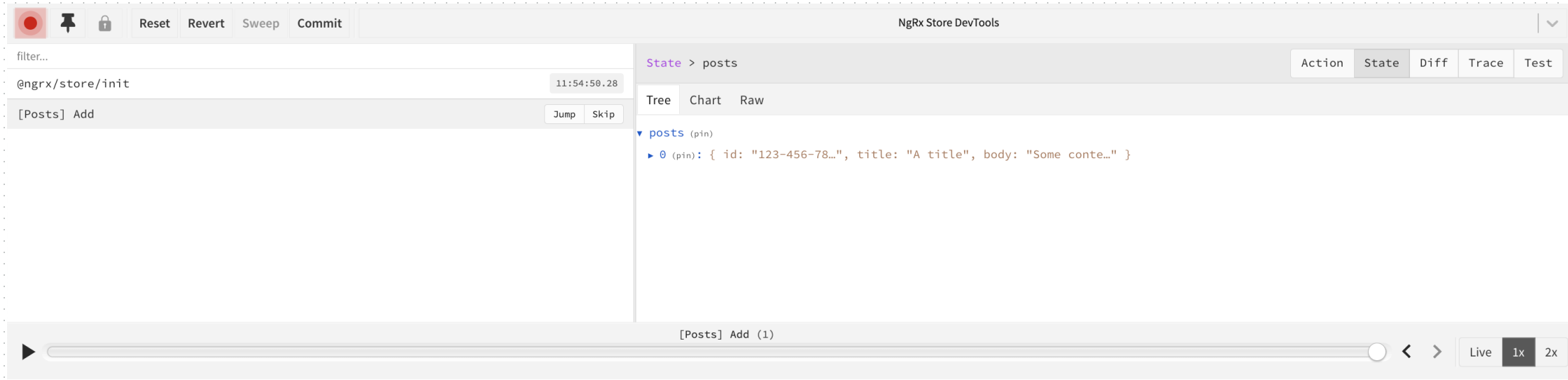
```
export class HomeComponent {

  public posts$: Observable<IPost[]>;
  public comments$: Observable<IComment[]>;

  public constructor(private _store: Store) {
   /**
    * Select posts and comments from the store
    */
    this.posts$ = this._store.select(postsSelector);
    this.comments$ = this._store.select(commentsSelector);
  }

  /**
  * Dispatch action to add a new post
  */
  public addPost(post: IPost) {
    this._store.dispatch(ADD_POST({ post }))
  }
```

Select from store and dispatch ADD_POST action

Store

```
const postsInitialState: IPost[] = [];

const postsReducer = createReducer(postsInitialState,
  on(ADD_POST, (state, { post }) => [...state, post])
);
```

Posts reducer – will handle ADD_POST dispatched action



Let’s code!

Open and read « apps/6-angular-ngrx/README.md »

Effects

```
export class AppComponent {

  public posts$: Observable<IPost[]>;

  public constructor(
    private _http: HttpClient
  ) {
    this.posts$ = this._http.get('https://jsonplaceholder.typicode.com/posts');
  }

}
```

Usual way to load data without effects

Effects

```
this._store.dispatch(LOAD_POSTS());
```

Trigger effect

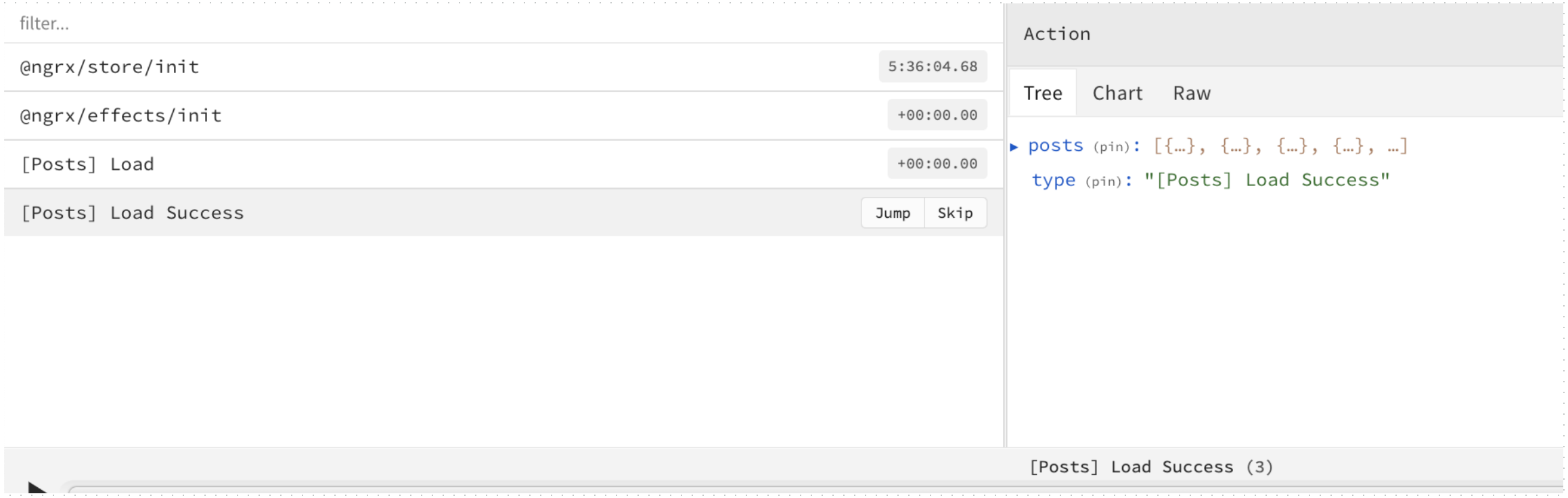
```
public loadPosts$ = createEffect(
  () => this._actions.pipe(
    ofType(LOAD_POSTS),
    exhaustMap(
      () => this._http.get<IPost[]>('https://jsonplaceholder.typicode.com/posts')
        .pipe(
          map((posts) => LOAD_POSTS_SUCCESS({ posts }))
        )
    )
  )
)
```

Load remote posts

```
on(LOAD_POSTS_SUCCESS, (state, { posts }) => posts)
```

Store posts

Effects



« Load » (fetch) then « Load Success » (store) actions are dispatched

Let’s code!

Open and read « apps/6-angular-ngrx/README.md » (effects section)

Reactive Forms

Custom controls

Angular allows us to create custom controls very easily thanks to the ControlValueInterface, implemented by the component:

```
interface ControlValueAccessor {
 writeValue(obj: any): void
 registerOnChange(fn: any): void
 registerOnTouched(fn: any): void
 setDisabledState(isDisabled: boolean)?: void
}
```

writeValue

Writes a new value to the element. This method is called by the forms API to write to the view when programmatic changes from model to view are requested.

Custom controls

registerOnChange

Registers a callback function that is called when the control's value changes in the UI.

registerOnTouched

Registers a callback function that is called by the forms API on initialization to update the form model on blur.

setDisabledState

Function that is called by the forms API when the control status changes to or from 'DISABLED'. Depending on the status, it enables or disables the appropriate DOM element.

```
export class CounterComponent implements ControlValueAccessor {

  public value: number = 0;
  public disabled = false;
  public onChange: (value: number) => {};
  public onTouched: () => {};

  public increase(): void {
    this.value++;
    this.onChange(this.value);
  }

  public writeValue(value: number): void {
    this.value = value;
  }

  public registerOnChange(fn: (value: number) => {}): void {
    this.onChange = fn;
  }

  public registerOnTouched(fn: () => {}): void {
    this.onTouched = fn;
  }

  public setDisabledState(isDisabled: boolean): void {
    this.disabled = isDisabled;
  }
}
```

counter.component.html

```
<input
  type="text"
  readonly
  [value]="value"
  [disabled]="disabled"
/>
<button
  (click)="increase()"
  [disabled]="disabled">
  Increment
</button>
```

Let’s code!

Open and read « apps/7-angular-reactiveforms/README.md »

Custom validators

Angular provides some built-in validators which are great, but not sufficient for all use cases. Often we need a custom validator that is designed especially for our use case.

You define custom validators for reactive forms with functions

```
export function ValidateUrl(control: AbstractControl) {

  if (!control.value.startsWith('https') || !control.value.includes('.io')) {
    return { invalidUrl: true };
  }

  return null;
}
```

If url is invalid, return error, else return null

```
new FormControl<string | null>(null, [ Validators.required, ValidateUrl ]);
```

Use validator

Custom validators

You define custom validators for template-driven forms with directives

```
@Directive({
  selector: '[appUrlValidator]',
  providers: [{
    provide: NG_VALIDATORS,
    useExisting: ValidateUrlDirective,
    multi: true
  }]
})
export class ValidateUrlDirective implements Validator {
  public validate(control: AbstractControl) {
    if (!control.value.startsWith('https') || !control.value.includes('.io')) {
      return { invalidUrl: true };
    }
    return null;
  }
}
```

```
<input type="text" name="phone [(ngModel)]="phone" appUrlValidator />
```

Async Validators

Occasionally, you may want to validate form input against data that is available through an asynchronous source (eg: HTTP backend)

For instance, checking if a username or email address exists before form submission

You achieve this using Async Validators

Async Validators

```
function usernameExists(): AsyncValidatorFn {
  return (control: AbstractControl): Observable<ValidationErrors | null> => {
    return this._http.get('http://foo.api', {
      params: {
        username: control.value
      }
    }).pipe(
      map((data) => data.result ? { usernameExists: true } : null)
    );
  };
}
```

Call the API, if username exists return an error, else return null

```
new FormControl<string | null>(null, [], [ usernameExists() ]);
```

Use the validator in the FormControl

Let’s code!

Open and read « apps/7-angular-reactiveforms/README.md » (custom validator)

Cypress

Introduction

Cypress is a **front end testing tool**, often compared to Selenium, Protactor, etc.

Users are typically developers or QA engineers building web applications using modern JavaScript frameworks.

Cypress can test **anything** that runs in a browser

```
it('adds todos', () => {
  cy.visit('https://todo.app.com’)
  cy.get('[data-test="new-todo"]’)
   .type('write code{enter}’)
   .type('write tests{enter}’)
  // confirm the application is showing two items
  cy.get('[data-test="todos"]').should('have.length', 2)
})
```

Basic Cypress test

Write tests

Tests written in Cypress are meant to be easy to read and understand

Passing test

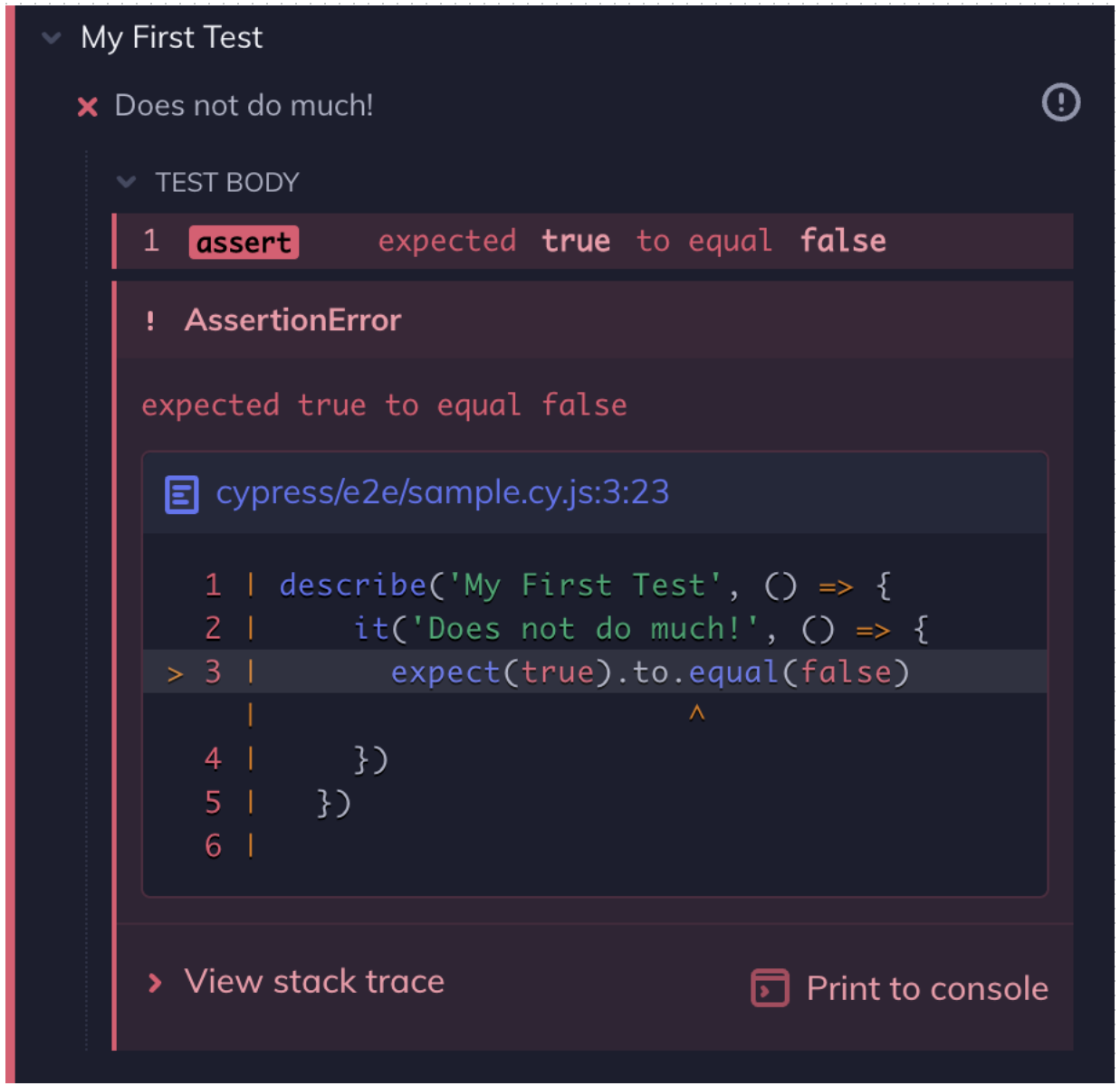
```
describe('My First Test', () => { // Test suite
  it('Does not do much!', () => { // Test
    expect(true).to.equal(true)
  })
})
```

Write tests

Failing test

```
describe('My First Test', () => {
  it('Does not do much!', () => {
    expect(true).to.equal(false)
  })
})
```

Cypress displays the stack trace and the code frame where the assertion failed

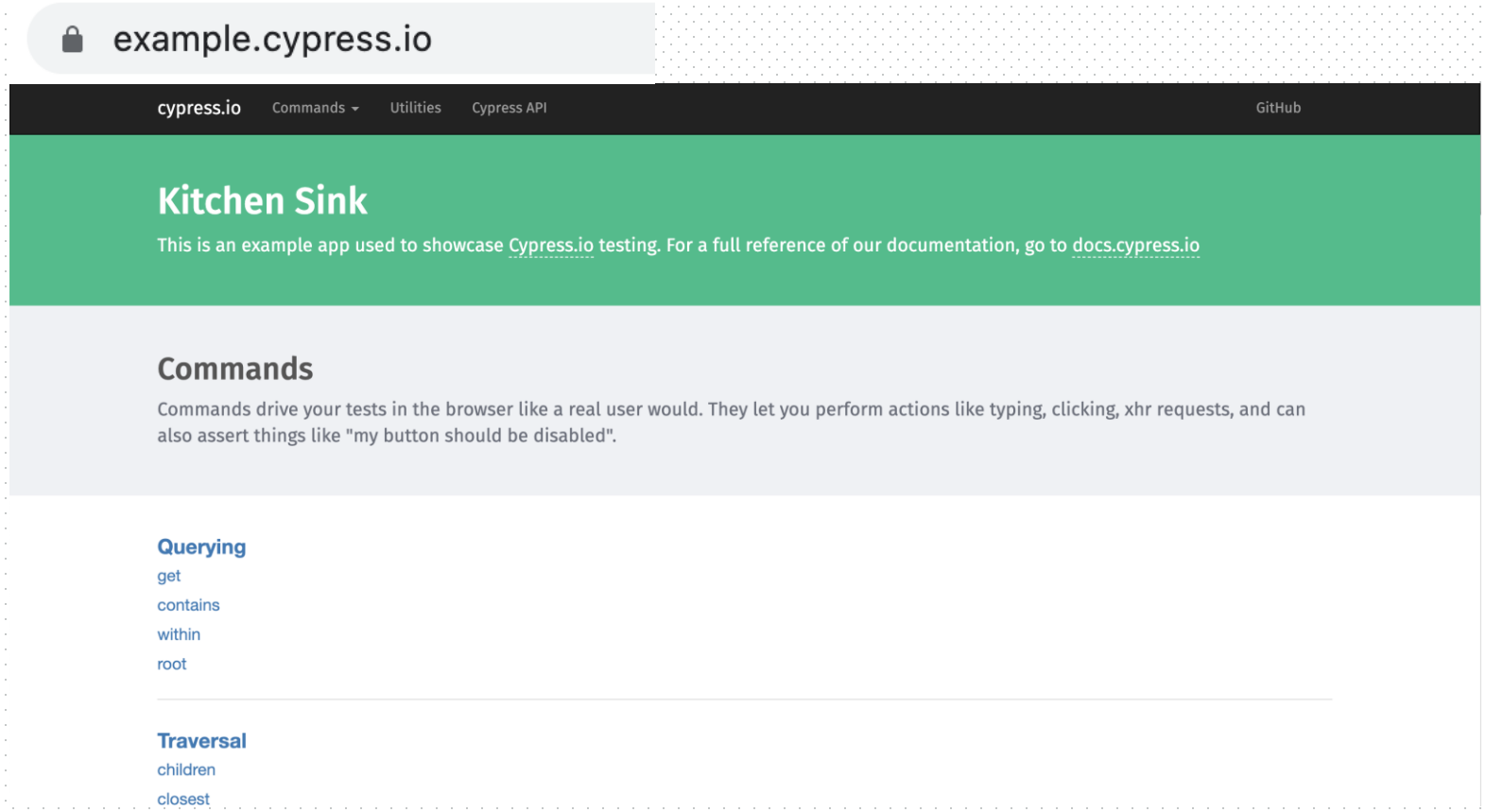


Write tests

```
describe('My First Test', () => {
  it('Gets, types and asserts', () => {
    // 1. Visit https://example.cypress.io
    cy.visit('https://example.cypress.io');

    // 2. Get element containing text "type" and click on it
    cy.contains('type').click();

    // 3. Read URL and assert it includes '/commands/actions'
    cy.url().should('include', '/commands/actions');

    // 4. Get an input, type into it and verify that the value has been updated
    cy.get('.action-email')
      .type('fake@email.com')
      .should('have.value', 'fake@email.com');
  })
})
```

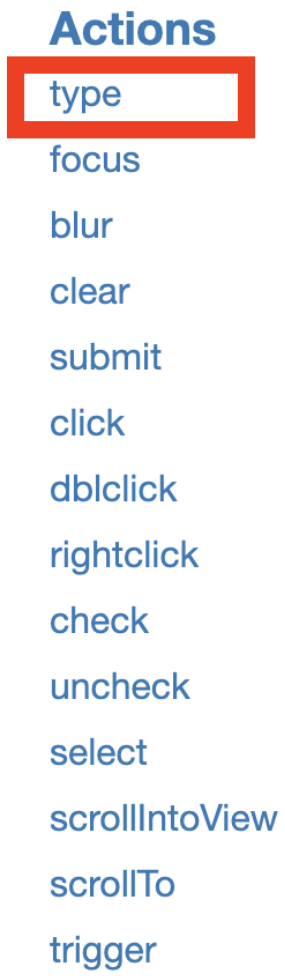
First valuable test

Write tests



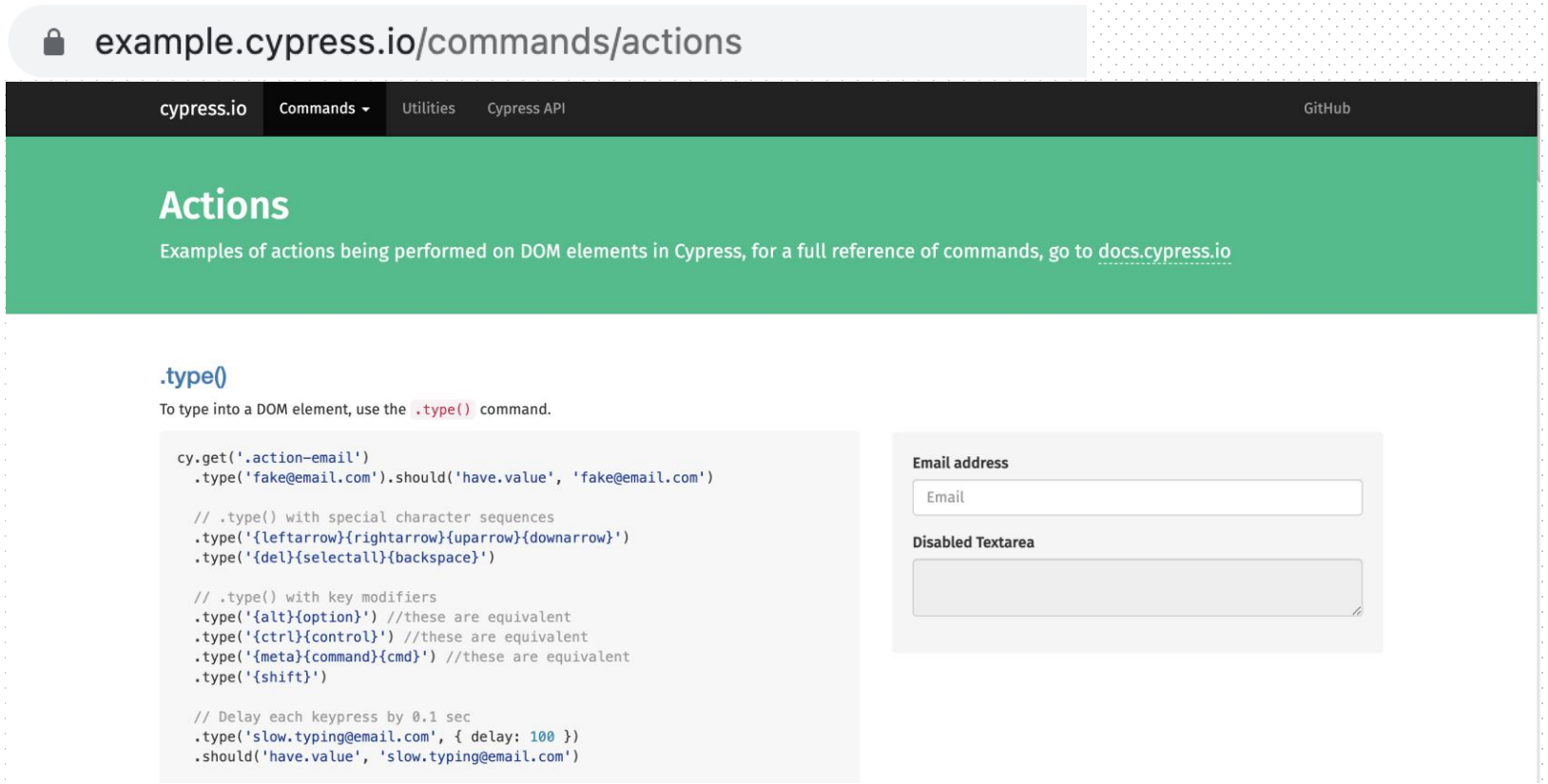
1. Visit https://example.cypress.io

Write tests



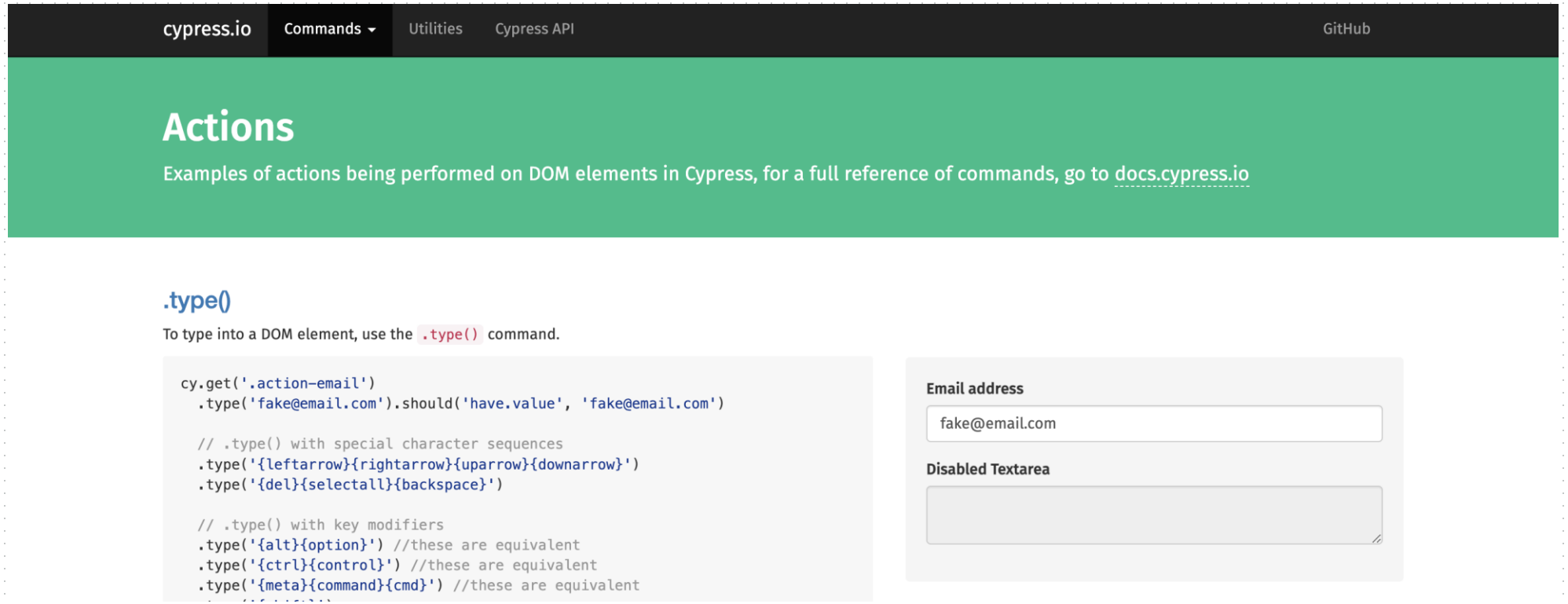
2. Find the element with content « type » and click on it

Write tests



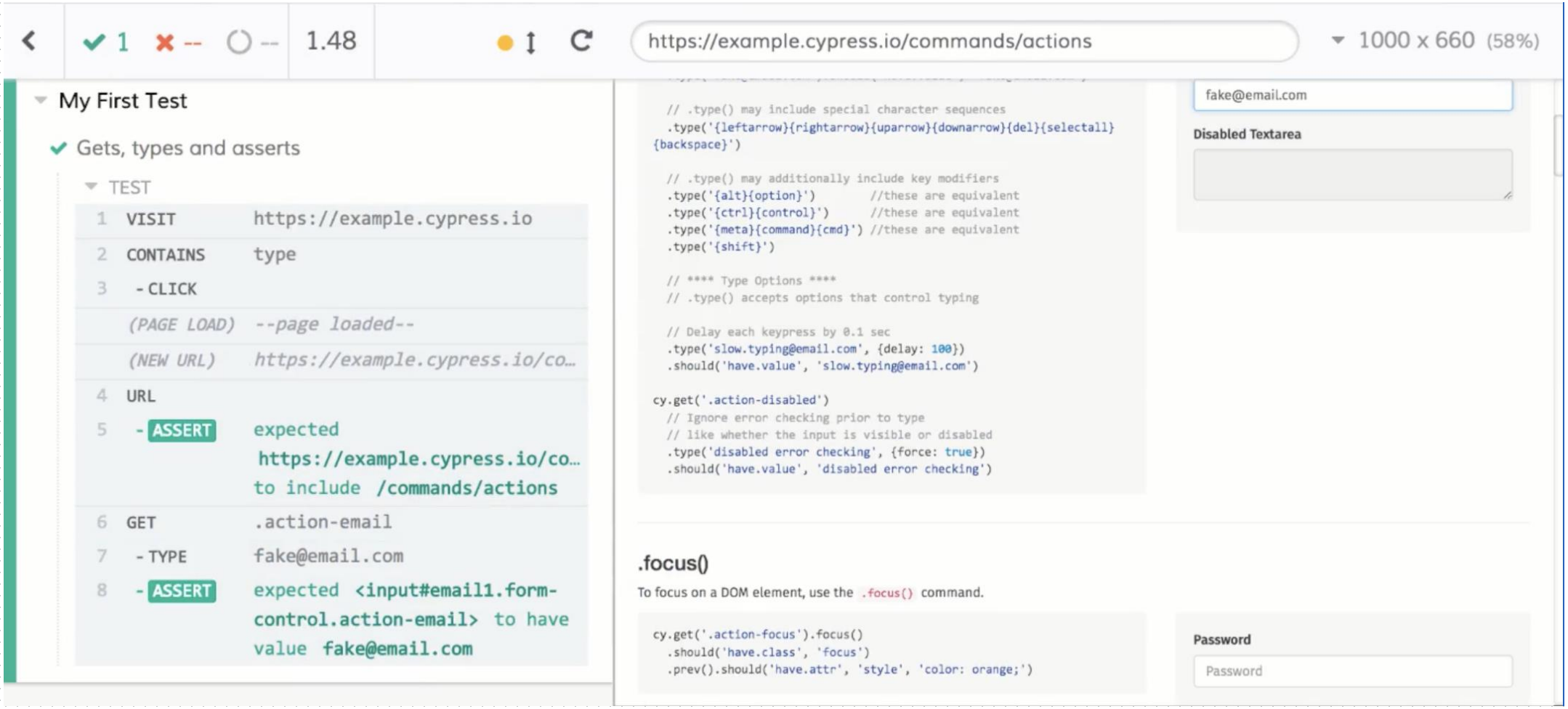
3. Get the URL and assert it includes: /commands/actions

Write tests



4. Get the input with the action-email class, type fake@email.com into the input and assert the input reflects the new value

Write tests



Cypress dashboard - Test report

Write tests

API documentation

https://docs.cypress.io/api/table-of-contents

Let’s code!

Open and read « apps/8-cypress/README.md »

Angular Universal

Server-Side Rendering

Introduction

Angular Universal is a technology that renders **Angular applications on the server**

A normal Angular application **executes in the browser,** rendering pages in the DOM in response to user actions.

Angular Universal executes on the server, **generating static application pages** that later get bootstrapped on the client.

The application generally renders more quickly, giving users a chance **to view the application layout before it becomes fully interactive**.

Introduction

There are three main reasons to create a Universal version of your application.

1. Facilitate web crawlers through search engine optimization (SEO)

Search engines rely on web crawlers to index content and make that content searchable on the web.

These web crawlers might be unable to navigate and index your application as a human user could do.

Angular Universal can generate a static version of your application that is easily searchable, linkable, and navigable without JavaScript. Universal also makes a site preview available because each URL returns a fully rendered page.

Introduction

2. Improve performance on mobile and low-powered devices

Some devices don't support JavaScript or execute JavaScript so poorly that the user experience is unacceptable.

For these cases, you might require a server-rendered, no-JavaScript version of the application.

This version, however limited, might be the only practical alternative for people who otherwise couldn't use the application at all.

Introduction

3. Show the first page quickly

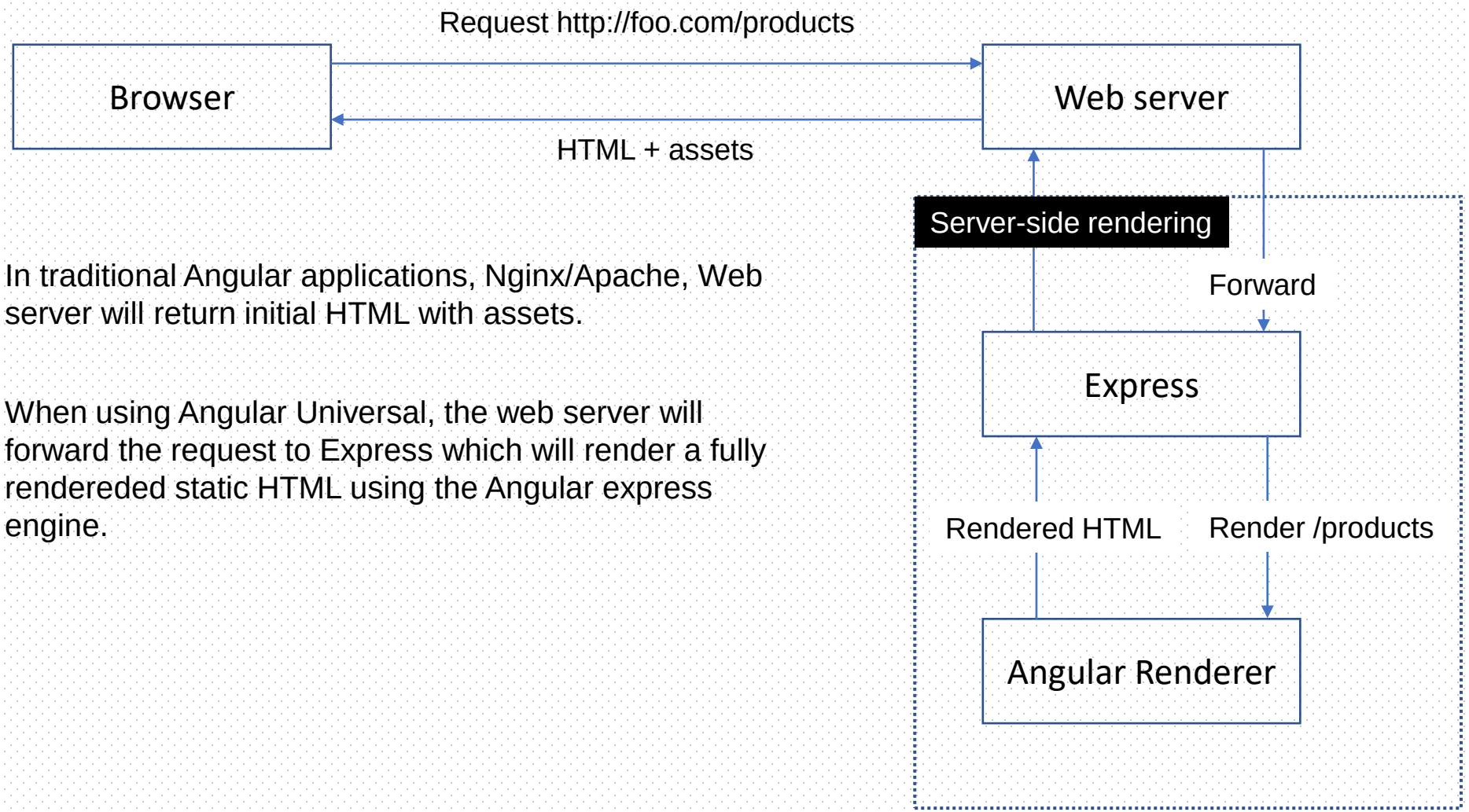
With Angular Universal, you can generate landing pages for the application that look like the complete application. **The pages are pure HTML**, and can display even if JavaScript is disabled.

The pages don't handle browser events, but they do support navigation through the site using routerLink.

In practice, **you'll serve a static version of the landing page to hold the user's attention**. At the same time, you'll load the full Angular application behind it.

The user perceives near-instant performance from the landing page and **gets the full interactive experience after the full application loads**.

Introduction



In traditional Angular applications, Nginx/Apache, Web server will return initial HTML with assets.

When using Angular Universal, the web server will forward the request to Express which will render a fully rendereded static HTML using the Angular express engine.

Usage

Install

```
ng add @nguniversal/express-engine
```

Run dev server. Offers live reload during development, but uses server-side rendering

```
npm run dev:ssr
```

Builds both the server script and the application in production mode. Use this command when you want to build the project for deployment.

```
ng build && ng run app-name:server
```

Live demo

Add Angular Universal to an existing application

Any questions?

Thank you