



Gemeente Haarlem

Angular - Maatwerk



Peter Kassenaar
info@kassenaar.com



NgRx Effects

Talking to external resources and dispatching a new action upon result



For Instance (when to use ngrx/effects):

- Talking to a RESTFul server == a *side effect* with implications on the store.
 - Hence the name, `ngrx/effects`
 - So, it is a *side effects* model for `ngrx/store`
- *Listen* for `ngrx/store` actions
- *Isolate* effects from components and reducers
- Communicate *outside* of Angular, notify the store when changes are complete
 - Perfect for Asynchronounous operations!



What is @ngrx/effects?

"@ngrx/effects is part of the NgRx state management library.

*It lets you **listen for dispatched actions** and perform **side effects** (like HTTP requests, logging, navigation, etc.) **outside** of components and reducers. Reducers must be pure functions—no side effects allowed.*

Effects solve the problem of handling async logic in a clean, testable, and reactive way."

Latest information: @ngrx.io



The screenshot displays the @ngrx.io website with a purple header. The navigation bar includes links for DOCS, WORKSHOPS, BLOG, and SPONSOR, along with a search bar and social media icons. The left sidebar lists various NGRX modules, with '@ngrx/effects' highlighted by a red circle. The main content area is titled '@ngrx/effects' and includes an introduction, key concepts, and installation instructions. The right sidebar shows a table of contents for the '@ngrx/effects' guide.

@ngrx/effects

Effects are an RxJS powered side effect model for *Store*. Effects use streams to provide *new sources* of actions to reduce state based on external interactions such as network requests, web socket messages and time-based events.

Introduction

In a service-based Angular application, components are responsible for interacting with external resources directly through services. Instead, effects provide a way to interact with those services and isolate them from the components. Effects are where you handle tasks such as fetching data, long-running tasks that produce multiple events, and other external interactions where your components don't need explicit knowledge of these interactions.

Key Concepts

- Effects isolate side effects from components, allowing for more *pure* components that select state and dispatch actions.
- Effects are long-running services that listen to an observable of *every* action dispatched from the *Store*.
- Effects filter those actions based on the type of action they are interested in. This is done by using an operator.
- Effects perform tasks, which are synchronous or asynchronous and return a new action.

Installation

Detailed installation instructions can be found on the [Installation](#) page.

Comparison with Component-Based Side Effects

In a service-based application, your components interact with data through many different services that expose data through properties and methods. These services may depend on other services that manage other sets of data. Your components consume these services to perform tasks, giving your components many responsibilities.

Imagine that your application manages movies. Here is a component that fetches and displays a list of movies.

@ngrx/effects

- Introduction
- Key Concepts
- Installation
- Comparison with Component-Based Side Effects
- Writing Effects
- Handling Errors
- Functional Effects
- Registering Effects
 - Alternative Way of Registering Effects
- Incorporating State
- Using Other Observable Sources for Effects

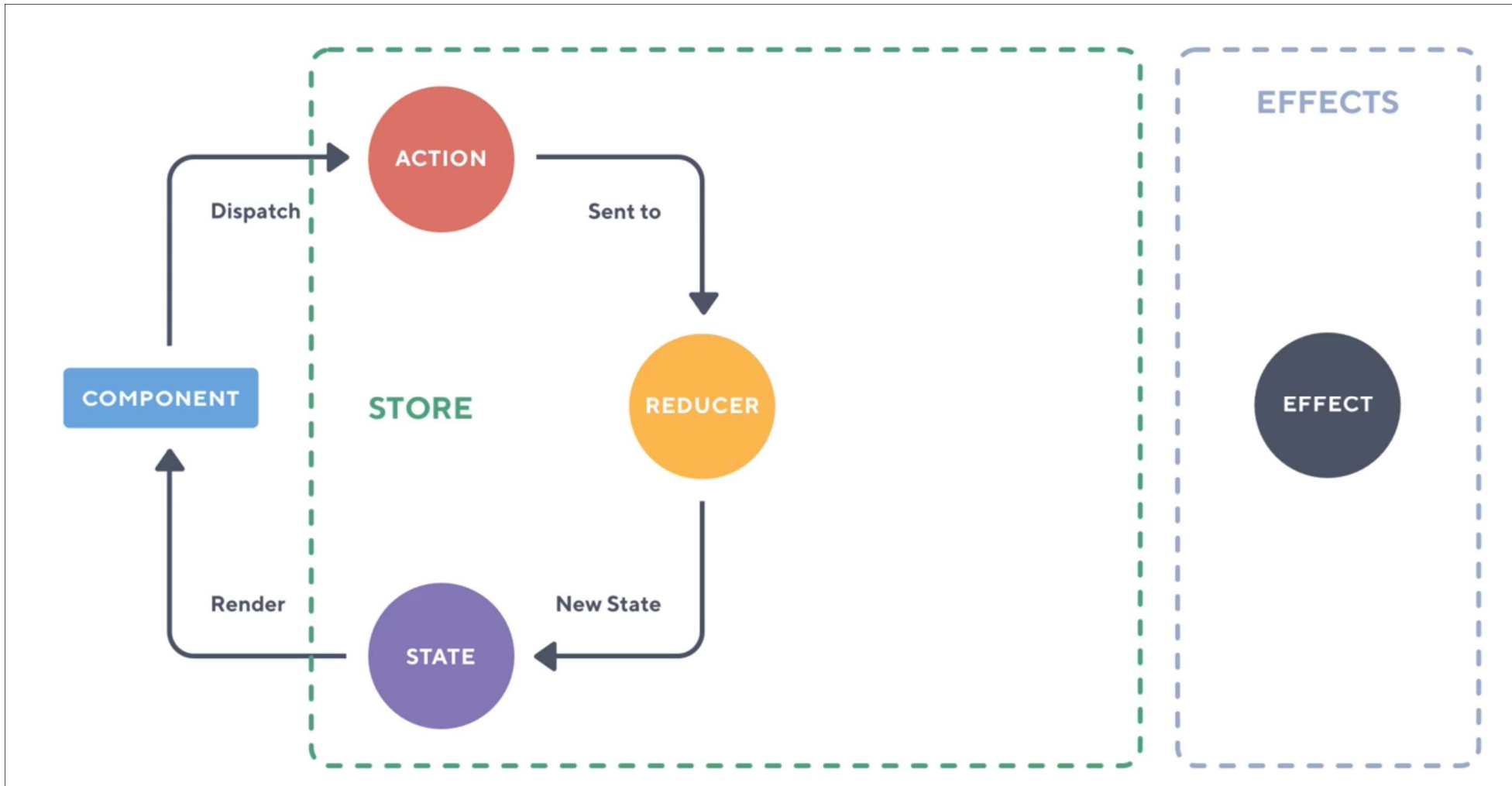
<https://ngrx.io/guide/effects>



Basic flow

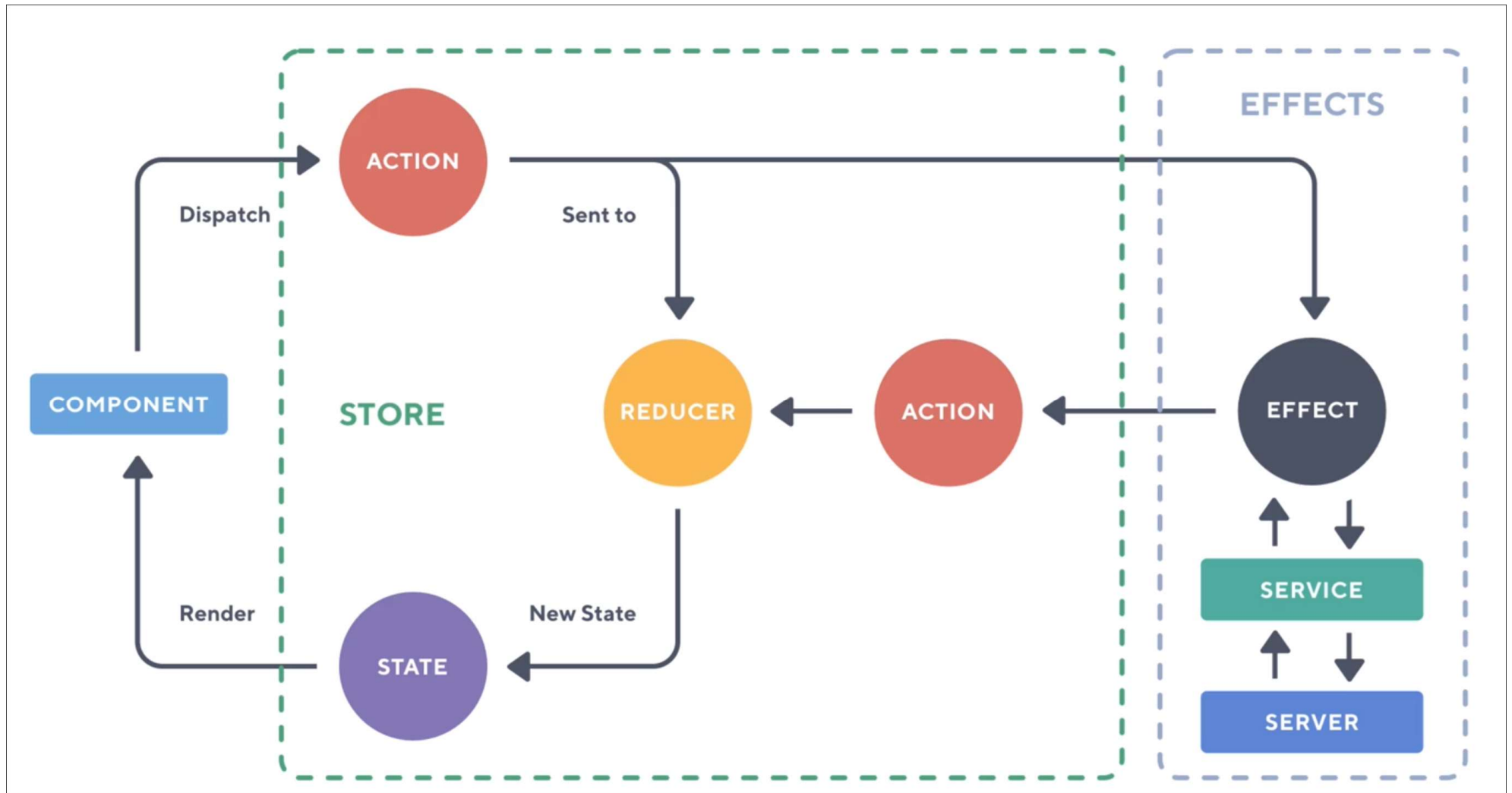
1. User dispatches an action (for example `loadCities`).
2. Effect listens to it, runs some async code (like HTTP call)
3. On success/failure, effect dispatches another action
 - For instance `loadCitiesSuccess` with the cities as payload, or `loadCitiesFailure` with the error as payload

Effects flow



<https://platform.ultimateangular.com/courses/ngrx-store-effects/lectures/3919211>

Effects flow



<https://platform.ultimateangular.com/courses/ngrx-store-effects/lectures/3919211>



So, an effect...

1. Listens to store `Actions`.
 - **YOU** define **which action** an effect listens to
2. Performs an action (such as talking to a webserver)
3. Dispatches the result to the reducer as a new `Action`.
4. The reducer in turn updates the store/state



Adding @ngrx/effects

```
npm install @ngrx/effects --save
```

OR

```
ng add @ngrx/effects
```

This also :

- Updates `package.json` **and** `npm install`
- Create a `src/app/app.effects.ts` file with an empty `AppEffects` class.
- Create a `src/app/app.effects.spec.ts` file with a basic unit test.
- Update your `src/app/app.module.ts`
 - imports array with `EffectsModule.forRoot([AppEffects])`.



We're doing this manually here, so use

```
npm install @ngrx/effects
```



Adding effects to the module

- Import `EffectsModule` from `@ngrx/effects`
- Use `EffectsModule.forRoot()` for root module
- Use `EffectsModule.forFeature()` for feature module

```
// Effects
import {EffectsModule} from '@ngrx/effects';

// exported effects from general index file
import { effects } from './effects/index';

@NgModule({
  ...
  imports      : [
    ...
    EffectsModule.forRoot(effects),    // array of effects
  ],
  ...
})
export class AppModule {
}
```





Example effect - /260-ngrx-effects

```
@Injectable()
export class CitiesEffects {

  constructor(private actions$: Actions,
               private cityService: CityService) {

  }

  loadCities$ = createEffect(() => this.actions$.pipe(
    // 1. Listen to this specific event (fired from app.component.ts)
    ofType(LoadCitiesViaEffect),
    mergeMap(() => {
      return this.cityService.loadCities() // 2. talk to API
    })
    .pipe(
      map((cities: City[]) => loadCitiesSuccess({cities})), // 3. Dispatch new action
      catchError(() => of(loadCitiesFail())) // 4. catch error and dispatch failure action
    );
  ));
}
```





Difference with service-based approach

- The Service *does not* subscribe.
- Instead it just fetches content from an URL and returns it to the effect
- ...which in turn dispatches a new Action to update the store.

```
// this is now called from the Effect
loadCities() {
  return this.http.get(BASE_URL)
    .pipe(
      tap(res => console.log('We talked to json-server and received: ', res)),
      finalize(() => 'Getting cities complete...')
    );
// Note: when using effects, no more subscriber in the service!
}
```



Workshop

- Start from `/260-ngrx-effects`
 - Use `cities.json`, or another `.json`-file you create yourself
 - Implement the `Effect()`'s for Adding and Removing a city
- OR: Create a blank project:
 - Add `@ngrx/effects` to the project and to the module
 - Use `createEffect()` to load an external resource (your `.json`-file)
 - Notify the store once the resource is loaded and update the UI.

