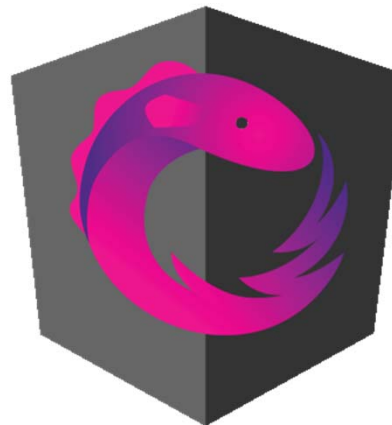
The background is a solid teal color with faint, white, hand-drawn architectural sketches. These sketches include a tall brick wall on the left, a large circular structure with concentric rings in the upper center, a ladder on the left side, and various other geometric shapes and lines scattered throughout. A small crescent moon is visible in the upper right corner.

Reactive Angular with RxJS @ngrx/store – Using @Effect

Peter Kassenaar –
info@kassenaar.com

What is @ngrx/effects for?

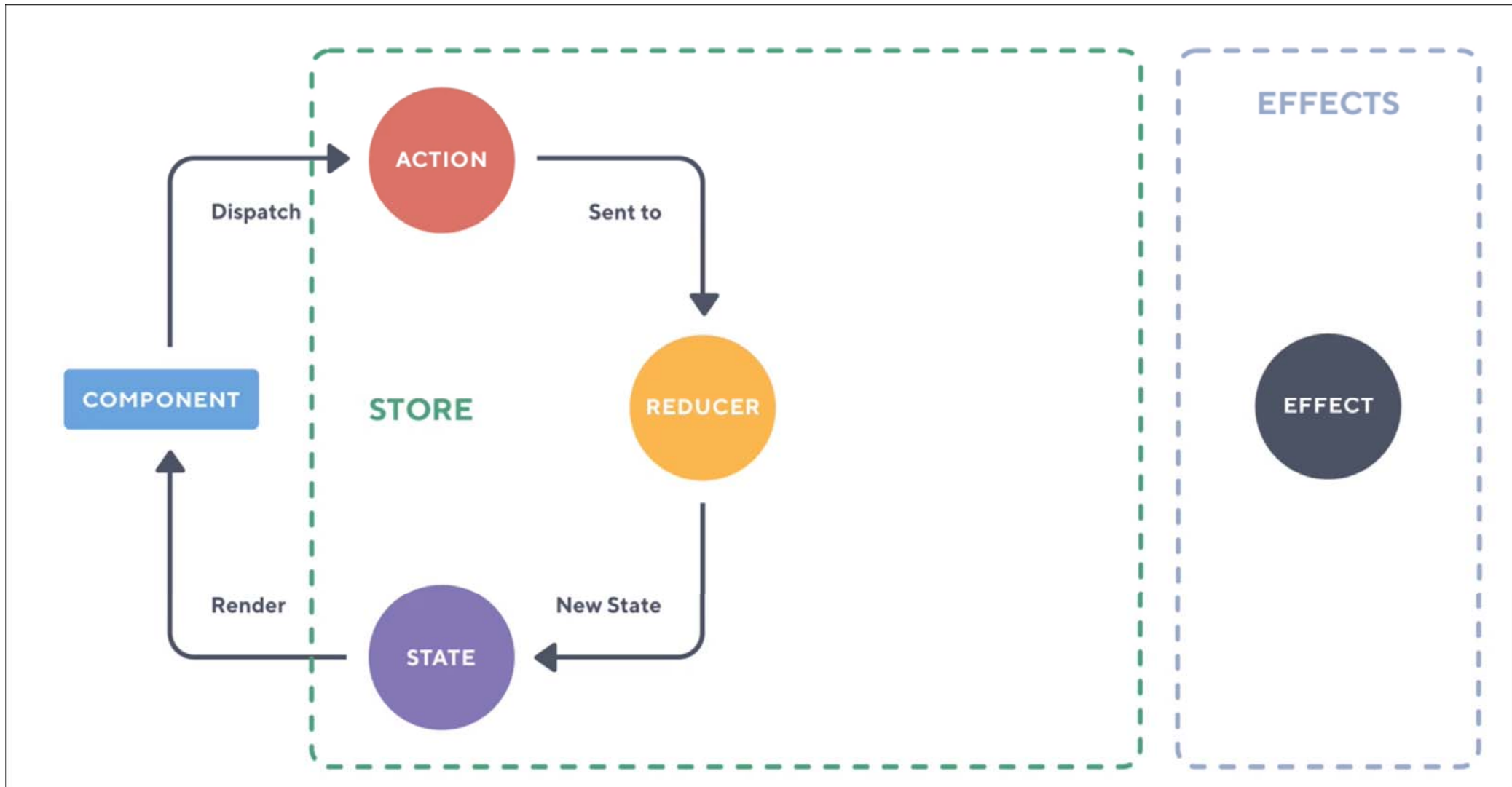
- @ngrx/effects provides an API to model event sources as actions. Effects:
 - *Listen* for actions dispatched from @ngrx/store
 - *Isolate side effects* from components
 - Provide *new sources* of actions to reduce state based on *external interactions* such as network requests, web socket messages and time-based events.



For Instance (when to use ngrx/effects):

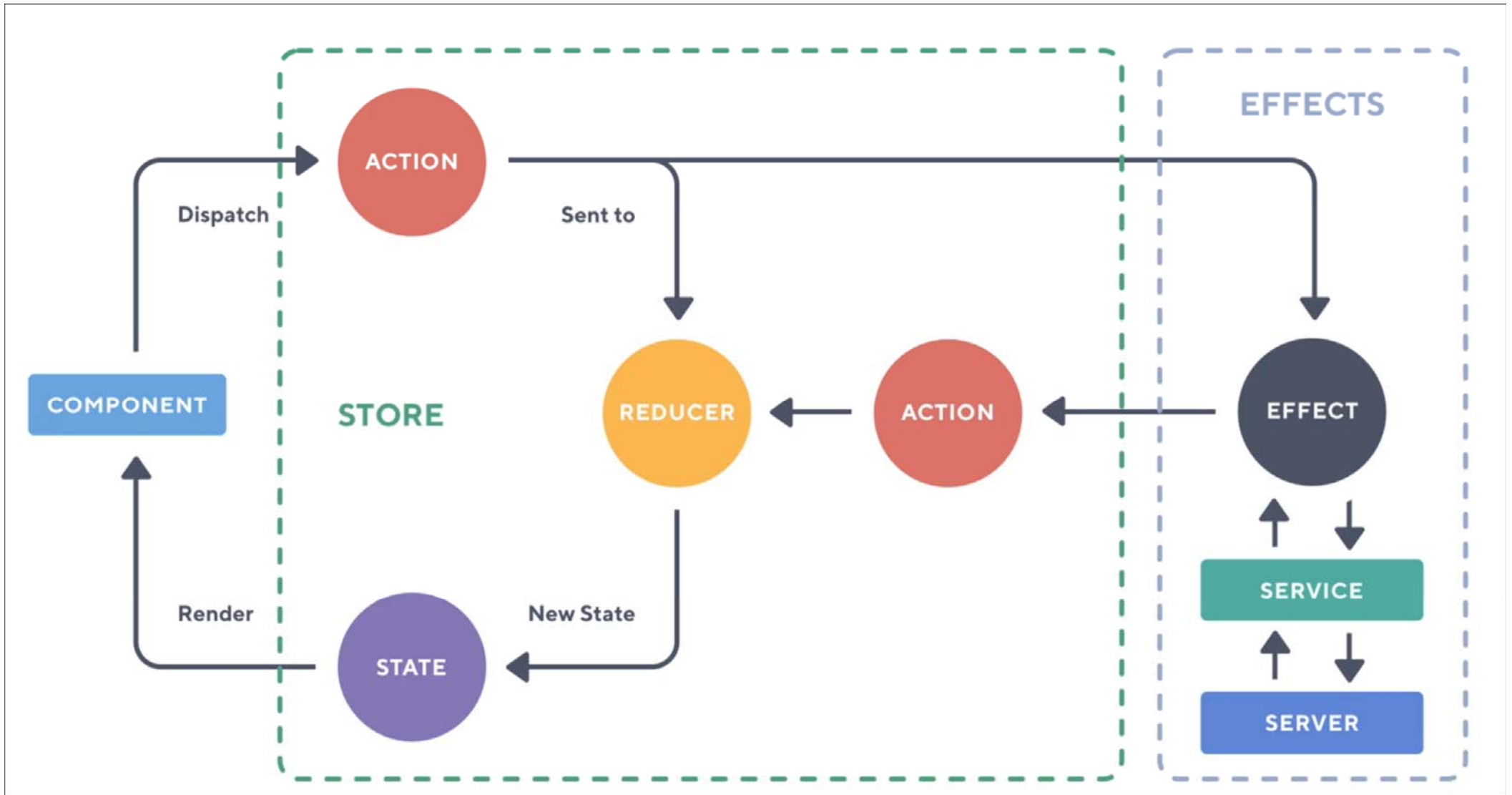
- Talking to a RESTful server == a side effect with implications on the store.
 - Hence the name, ngrx/effects
 - So, it is a **side effects** model for ngrx/store
- **Listen** for ngrx/store actions
- **Isolate** effects from components and reducers
- Communicate **outside** of Angular, notify the store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

Effects flow



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

Effects flow



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 {
}
```

Export all effects from index.ts file

Not mandatory – but seen often: export all files from a folder from an `index.ts`-file

```
// effects/index.ts, export everything from this folder  
// TODO: add extra/new effects to this file to auto-export them  
import {CitiesEffects} from './citiesEffects';  
  
export const effects: any[] = [CitiesEffects];  
  
export * from './citiesEffects';
```



Example effect - /225-store-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 ac
      catchError(() => of(loadCitiesFail())) // 4. catch error and dispatch failu
    );
  ));
}
```



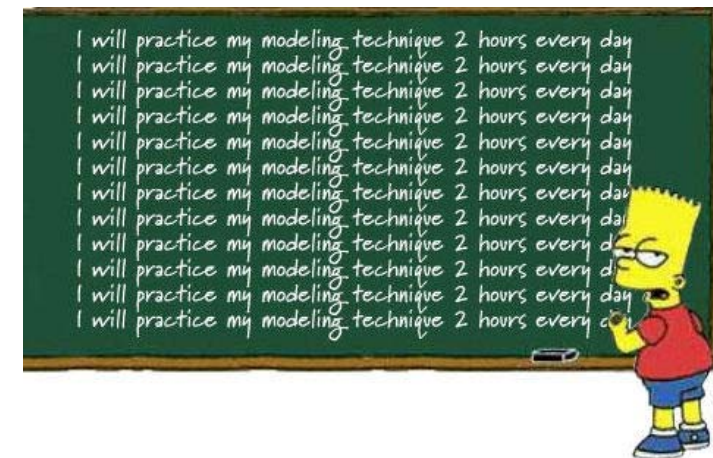
Difference with service-based approach

- 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 `/225-ngrx-store-effect`
 - 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
 - Create an `@Effect()` to load an external resource (your `.json`-file)
 - Notify the store once the resource is loaded and update the UI.



Don't always use switchMap()



<https://twitter.com/victorsavkin/status/963490147328290816>

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

The screenshot shows the official documentation for @ngrx/effects. The page has a purple header with navigation links: GETTING STARTED, DOCS, BLOG, RESOURCES, EVENTS, GITHUB, SPONSOR, and a search bar. A left sidebar lists the documentation structure, with @ngrx/effects expanded. The main content area is titled '@ngrx/effects' and includes a description, an introduction, key concepts, installation instructions, and a comparison with component-based side effects. A right sidebar shows a table of contents for the @ngrx/effects section.

@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

Table of Contents (Right Sidebar):

- @ngrx/effects
 - Introduction
 - Key Concepts
 - Installation
 - Comparison with component-based side effects
 - Writing Effects
 - Handling Errors
 - Registering root effects
 - Registering feature effects
 - Incorporating State