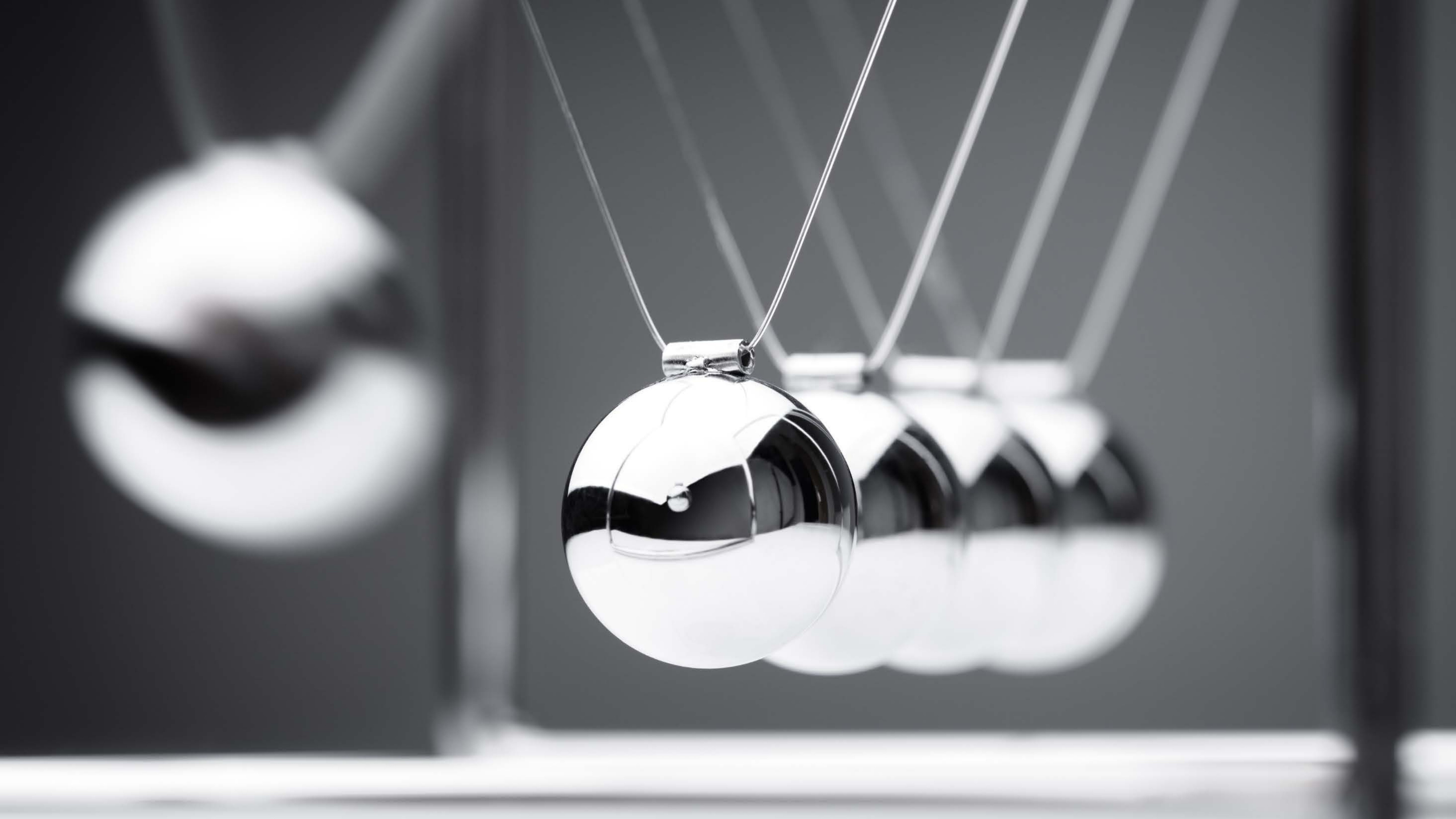
Working with Effects

Duncan Hunter
CONSULTANT | SPEAKER | AUTHOR

@dunchunter duncanhunter.com.au



Module Overview

Why use effects?

Install @ngrx/effects

Define an effect

Register an effect

Use an effect

Unsubscribe from observables

Exception handling in effects

NgRx Effects Library

Manages side effects to keep components pure

Effects Keep Components Pure

Component

```
constructor(
   private store: Store<State>,
   private productService: ProductService
) { }


ngOnInit() {
   this.productService.getProducts().subscribe(
      products => this.store.dispatch(
        new productActions.Load()
      )
}
```

Reducers Are Pure Functions

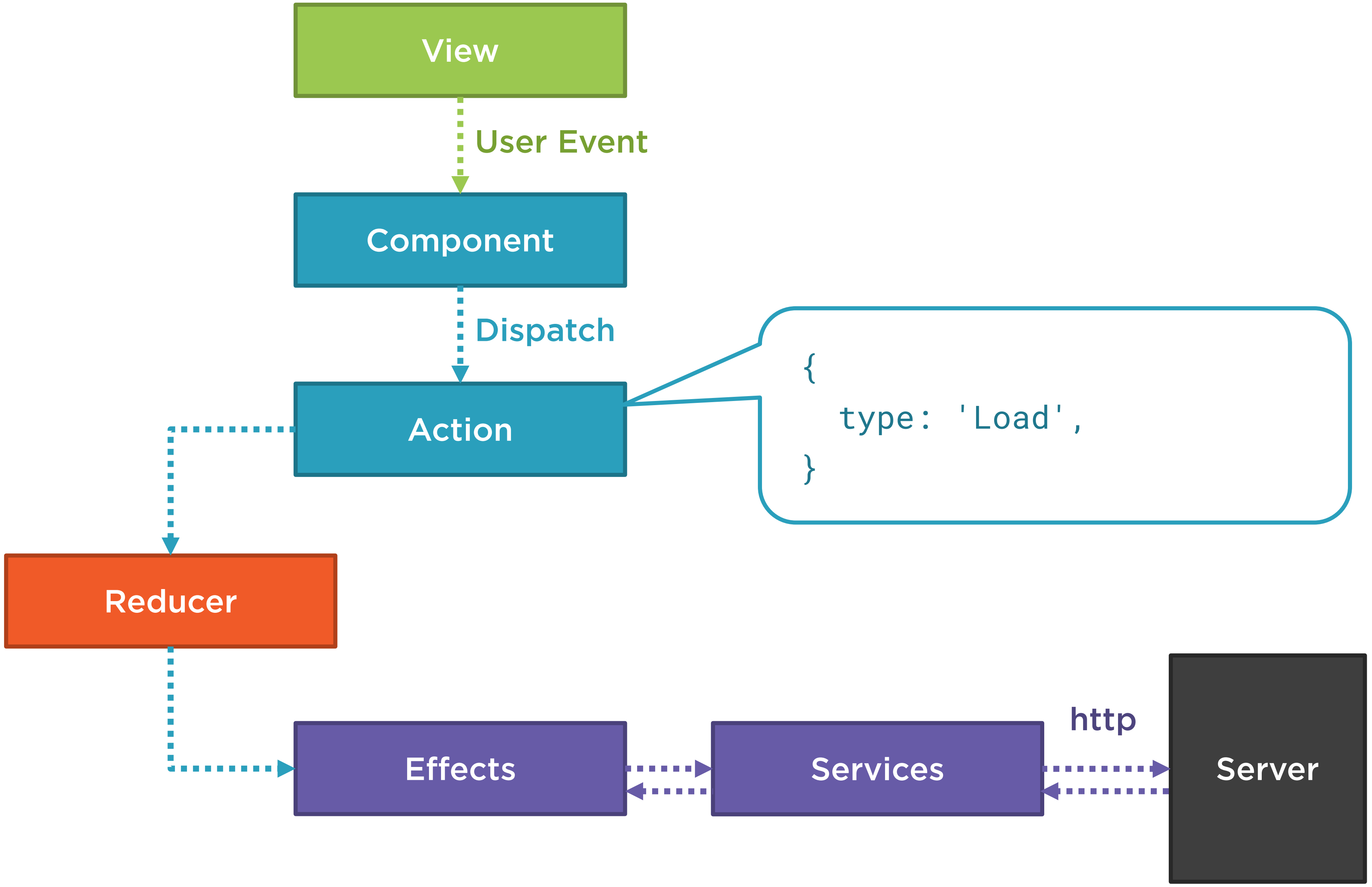
Reducer

```
switch(action.type) {
  case ProductActionTypes.Load:

~~    return this.productService.getProducts().subscribe(~~
~~       products => this.store.dispatch(~~
~~         new productActions.Load()~~
~~    )~~

}
```

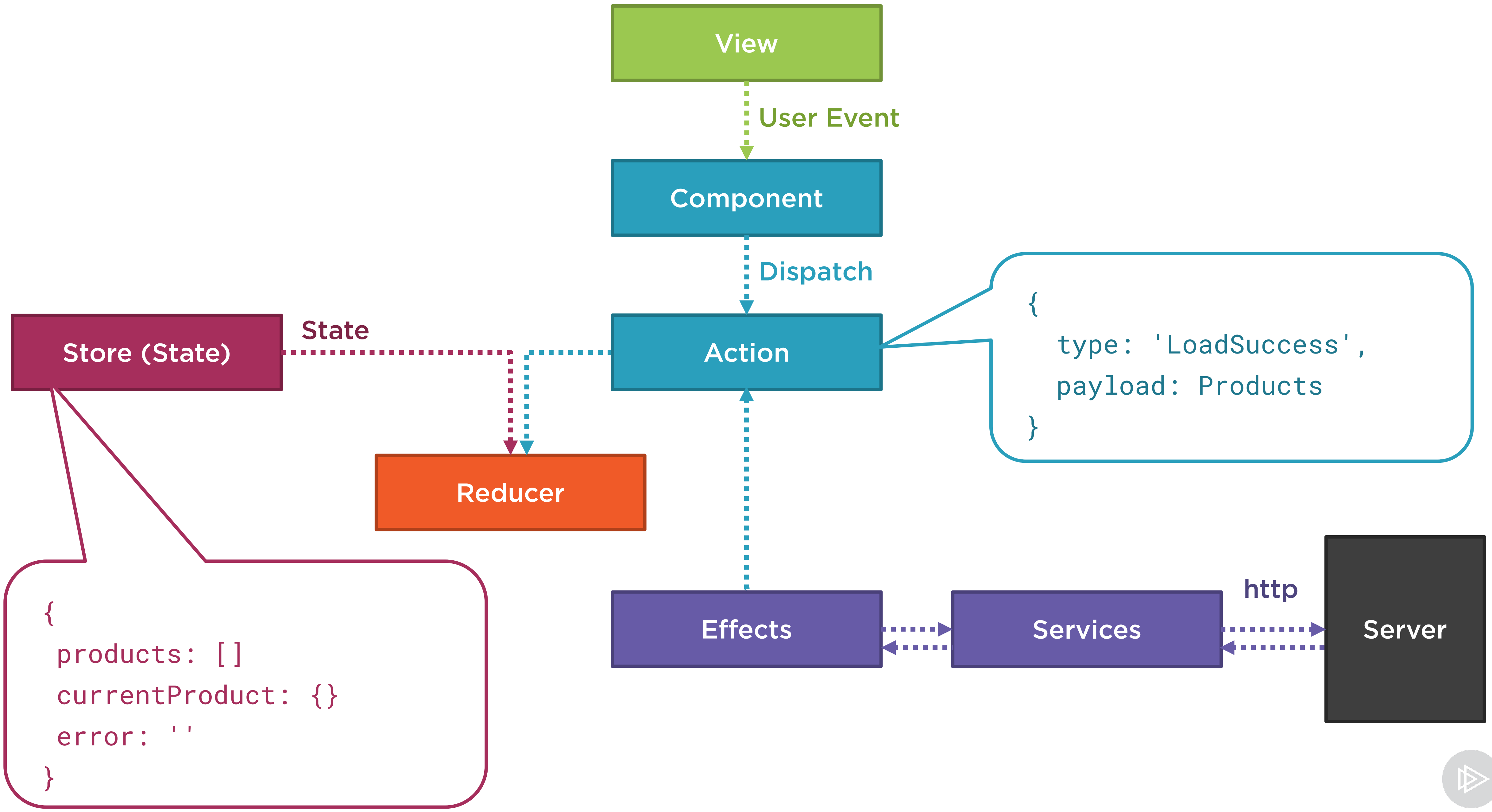
Effects Take Actions and Dispatch Actions

Effects

Effects take an action, do some work and dispatch a new action

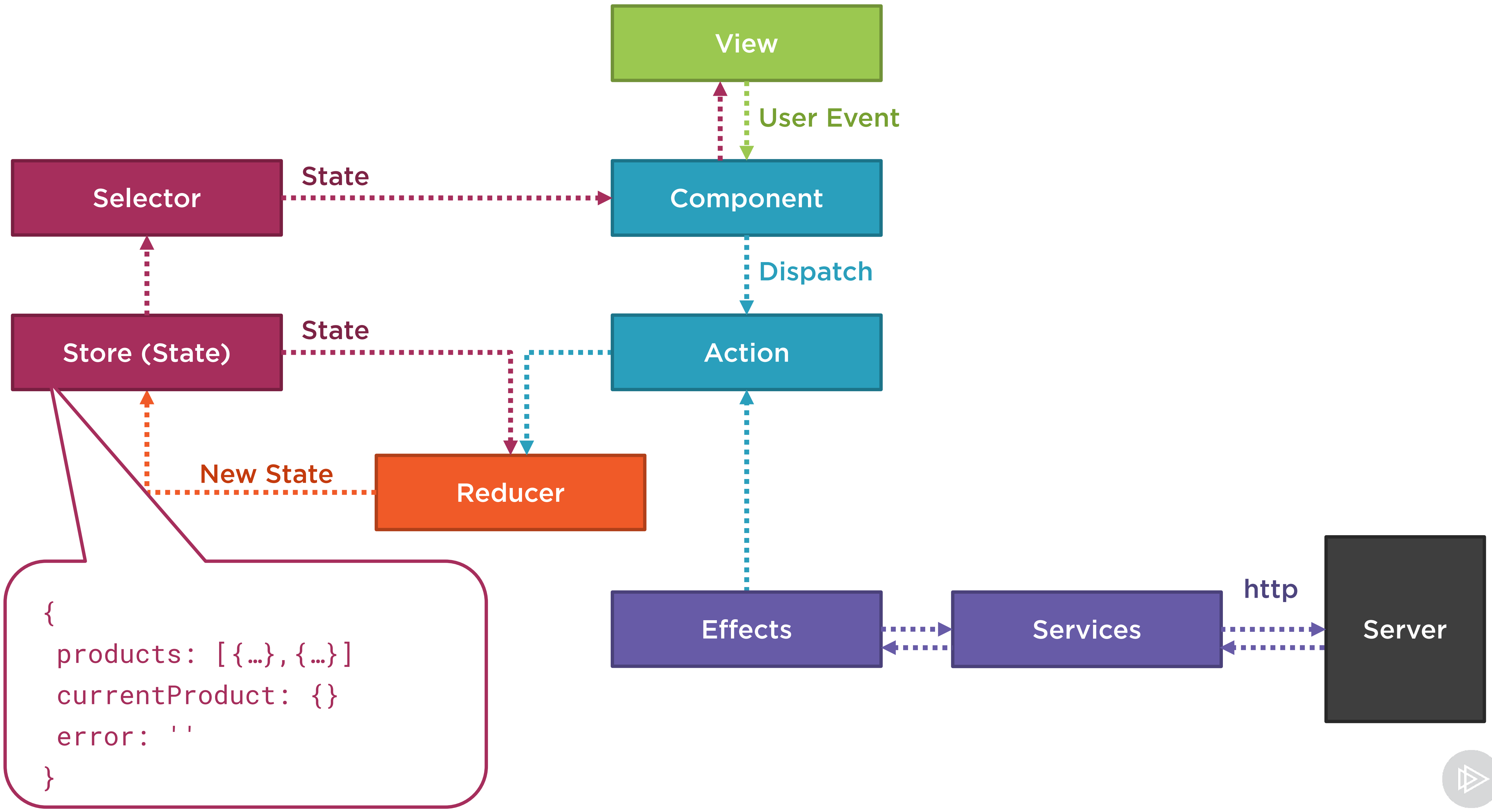

View
User Event
Component
Dispatch
Action
Reducer
Effects
Services
http
Server

```
{
  type: 'Load',
}
```



```
{
  type: 'LoadSuccess',
  payload: Products
}
```

```
{
 products: []
 currentProduct: {}
 error: ''
}
```



```
{
 products: [{…},{…}]
 currentProduct: {}
 error: ''
}
```

Benefits of Effects

Keep components pure

Isolate side effects

Easier to test

Defining an Effect

Create service

```
@Injectable()
export class ProductEffects {







}
```

Defining an Effect

Create service

```
@Injectable()
export class ProductEffects {

    constructor(private actions$:Actions,
                                                    ) { }







}
```

Defining an Effect

Create service

```
@Injectable()
export class ProductEffects {

   constructor(private actions$:Actions,
               private productService: ProductService) { }








}
```

Defining an Effect

Define effect

```
@Injectable()
export class ProductEffects {

    constructor(private actions$:Actions,
                private productService: ProductService) { }

    @Effect()
    loadProducts$





}
```

Defining an Effect

Define effect

```
@Injectable()
export class ProductEffects {

    constructor(private actions$:Actions,
                private productService: ProductService) { }

    @Effect()
    loadProducts$ = this.actions$



}
```

Defining an Effect

Define effect

```
@Injectable()
export class ProductEffects {

    constructor(private actions$:Actions,
                private productService: ProductService) { }

    @Effect()
    loadProducts$ = this.actions$.pipe(






    );
}
```

Defining an Effect

Filter actions

```
@Injectable()
export class ProductEffects {

   constructor(private actions$:Actions,
               private productService: ProductService) { }

   @Effect()
   loadProducts$ = this.actions$.pipe(
     ofType(ProductActionTypes.Load),





   );
}
```

Defining an Effect

```
@Injectable()
export class ProductEffec

    constructor(private a
                private pr

    @Effect()
    loadProducts$ = this.actions$.pipe(
      ofType(ProductActionTypes.Load),




    );
}
```

Filter actions

```
export enum ProductActionTypes {
  Load ='[Product] Load',
  LoadSuccess ='[Product] Load Success',
  LoadFail ='[Product] Load Fail'
}
```

Defining an Effect

Map

```
@Injectable()
export class ProductEffects {

    constructor(private actions$:Actions,
                private productService: ProductService) { }

    @Effect()
    loadProducts$ = this.actions$.pipe(
      ofType(ProductActionTypes.Load),
      mergeMap(action =>



      )
    );
}
```

Defining an Effect

Call service

```
@Injectable()
export class ProductEffects {

  constructor(private actions$:Actions,
              private productService: ProductService) { }

  @Effect()
  loadProducts$ = this.actions$.pipe(
    ofType(ProductActionTypes.Load),
    mergeMap(action =>
      this.productService.getProducts().pipe(


    )
  );
}
```

Defining an Effect

```
@Injectable()
export class ProductEffects {

   constructor(private actions$:Actions,
               private productService: ProductService) { }

   @Effect()
   loadProducts$ = this.actions$.pipe(
     ofType(ProductActionTypes.Load),
     mergeMap(action =>
       this.productService.getProducts().pipe(
        map(products => (new LoadSuccess(products)))
       )
     )
   );
}
```

Return action

Defining an Effect

Create service

Define effect

Filter actions

Map

Call service

Return new action

```
@Injectable()
export class ProductEffects {

   constructor(private productService: ProductService,
               private actions$:Actions) { }

   @Effect()
   loadProducts$ = this.actions$.pipe(
     ofType(ProductActionTypes.Load),
     mergeMap(action =>
       this.productService.getProducts().pipe(
        map(products => (new LoadSuccess(products)))
       )
     )
   );
}
```

Defining an Effect

Take an action

Do some work

Return a new action

```
@Injectable()
export class ProductEffects {

   constructor(private productService: ProductService,
               private actions$:Actions) { }

   @Effect()
   loadProducts$ = this.actions$.pipe(
     ofType(ProductActionTypes.Load),
     mergeMap(action =>
       this.productService.getProducts().pipe(
        map(products => (new LoadSuccess(products)))
       )
     )
   );
}
```

Demo

Install and define an effect

RxJS Operators

```
@Effect()
loadProducts$ = this.actions$.pipe(
  ofType(ProductActionTypes.Load),
  mergeMap(action =>
   this.productService.getProducts().pipe(
    map(products => (new LoadSuccess(products)))
   )
  )
);
```

RxJS Operators

```
@Effect()
loadProducts$ = this.actions$.pipe(
  ofType(ProductActionTypes.Load),
  mergeMap(action =>
   this.productService.getProducts().pipe(
    map(products => (new LoadSuccess(products)))
   )
  )
);
```

RxJS Operators

```
@Effect()
loadProducts$ = this.actions$.pipe(
  ofType(ProductActionTypes.Load),
  mergeMap(action =>
   this.productService.getProducts().pipe(
    map(products => (new LoadSuccess(products)))
   )
  )
);
```

RxJS Operators

```
@Effect()
loadProducts$ = this.actions$.pipe(
  ofType(ProductActionTypes.Load),
  switchMap(action =>
   this.productService.getProducts().pipe(
    map(products => (new LoadSuccess(products)))
   )
  )
);
```

RxJS Operators

switchMap	Cancels the current subscription/request and can cause race condition **Use for get requests or cancelable requests like searches**
concatMap	Runs subscriptions/requests in order and is less performant **Use for get, post and put requests when order is important**
mergeMap	Runs subscriptions/requests in parallel **Use for put, post and delete methods when order is not important**
exhaustMap	Ignores all subsequent subscriptions/requests until it completes **Use for login when you do not want more requests until the initial one is complete**

Registering an Effect

App Module

```
@NgModule({
 imports:[
  ...
  StoreModule.forRoot({}),
  EffectsModule.forRoot([]),
 ],
 declarations:[...],
 bootstrap:[...]
})
export class AppModule{ }
```

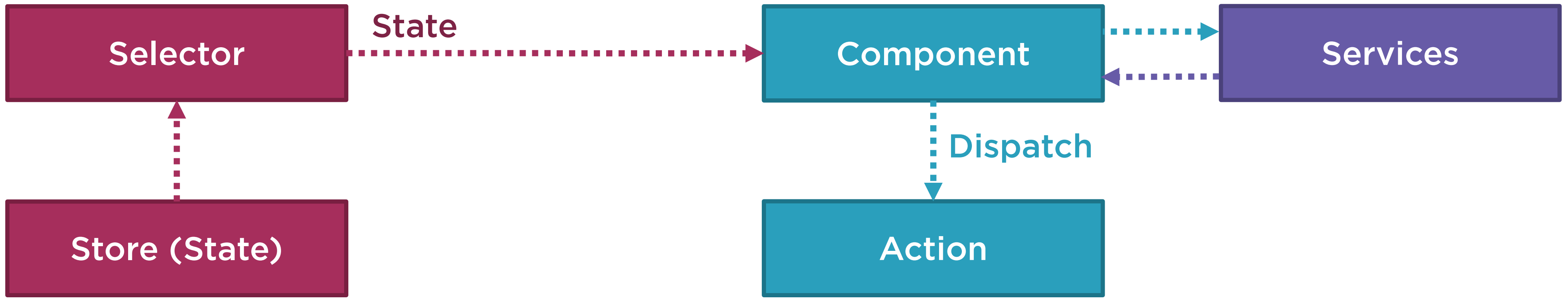
Product Module

```
@NgModule({
 imports:[
    ...
  StoreModule.forFeature('products',reducer),
  EffectsModule.forFeature([ProductEffects])
 ],
 declarations:[...],
 providers:[...]
})
export class ProductModule{ }
```

Demo

Registering an effect

Using Effects



Using Effects

Inject the store

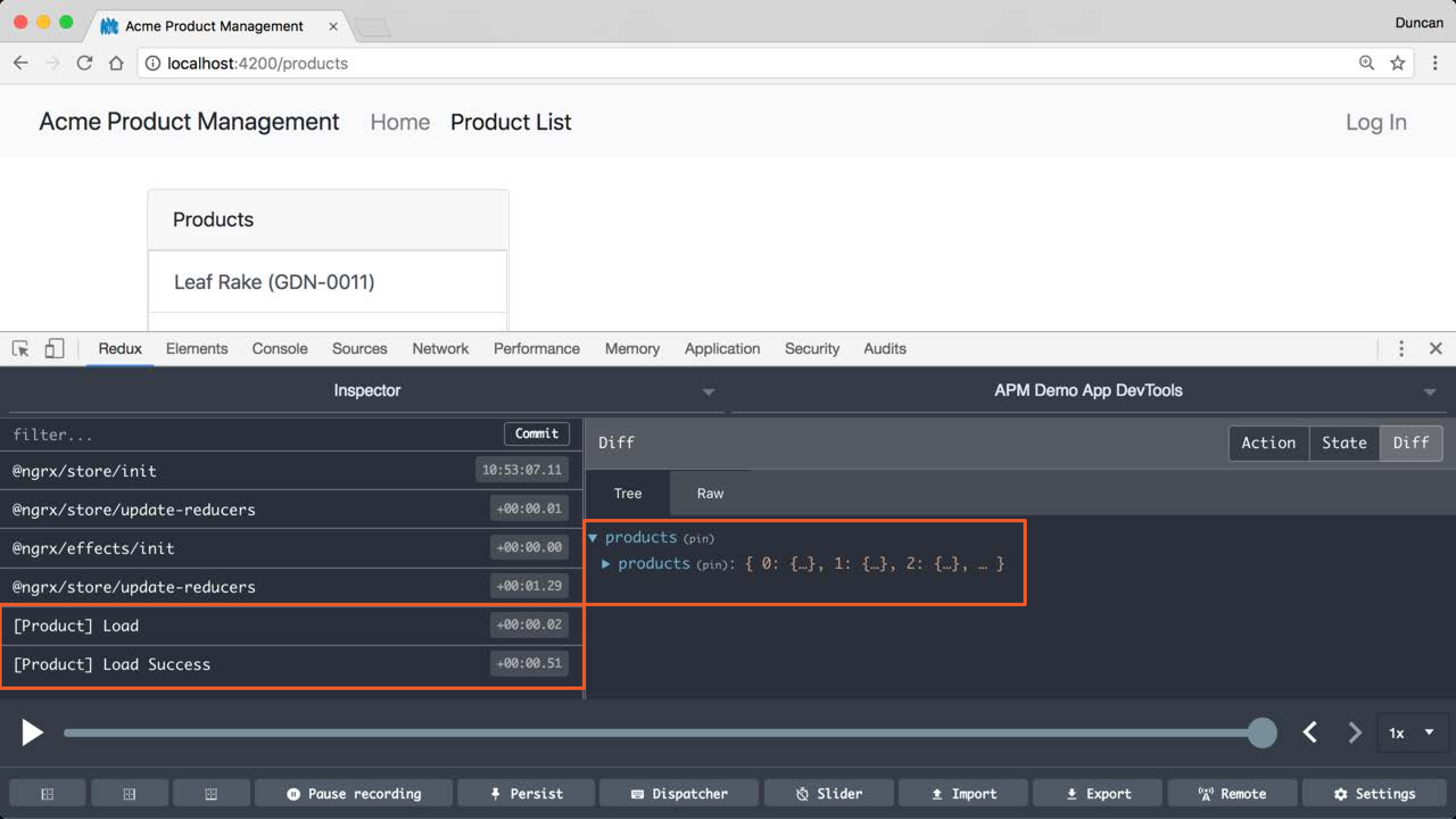
```
constructor(private store: Store<fromProducts.State>){}
```

Call the dispatch method

```
this.store.dispatch(new productActions.Load());
```

Select state with selector

```
this.store.pipe(select(fromProduct.getProducts)).subscribe(
  (products: Product[]) => this.products = products
);
```



```
@ngrx/store/init                 10:53:07.11
@ngrx/store/update-reducers      +00:00.01
@ngrx/effects/init               +00:00.00
@ngrx/store/update-reducers      +00:01.29
[Product] Load                   +00:00.02
[Product] Load Success           +00:00.51
```

```
▼ products (pin)
  ▶ products (pin): { 0: {…}, 1: {…}, 2: {…}, … }
```

Demo

Using the effect

You need to unsubscribe from the store.

Unsubscribing from Observables

```
export class ProductListComponent implements OnInit {


 ngOnInit() {
    this.store.pipe(select(fromProduct.getProducts))
    .subscribe((products: Product[]) => this.products = products);
   }
 }






}
```

Unsubscribing from Observables

Add subscriptions property

Push subscriptions onto array

Iterate and unsubscribe from each subscription

```
export class ProductListComponent implements OnInit, OnDestroy {
 subscriptions: Subscription[];

 ngOnInit() {
  this.subscriptions.push(this.store.pipe(
    select(fromProduct.getShowProductCode)
  ).subscribe(showProductCode => this.displayCode = showProductCode));
 }


 ngOnDestroy() {
  this.subscriptions.forEach(
   subscription => subscription.unsubscribe()
  );
 }


}
```

Unsubscribing from Observables

Initialize component Active variable

Add OnDestroy

```
export class ProductListComponent implements OnInit, OnDestroy {
 componentActive = true;

 ngOnInit() {
    this.store.pipe(select(fromProduct.getProducts))
     .subscribe((products: Product[]) => this.products = products);
   }
 }


 ngOnDestroy() {
   this.componentActive = false;
 }


}
```

Unsubscribing from Observables

Unsubscribe with takeWhile

```
export class ProductListComponent implements OnInit, OnDestroy {
 componentActive = true;

 ngOnInit() {
    this.store.pipe(select(fromProduct.getProducts),
     takeWhile(() => this.componentActive))
     .subscribe((products: Product[]) => this.products = products);
   }
 }


 ngOnDestroy() {
   componentActive = false;
 }


}
```

Unsubscribing from Observables

Unsubscribe with takeWhile

```
export class ProductListComponent implements OnInit, OnDestroy {
 componentActive = true;

 ngOnInit() {
    this.store.pipe(select(fromProduct.getProducts),
     takeWhile(() => this.componentActive))
     .subscribe((products: Product[]) => this.products = products);
   }
 }


 ngOnDestroy() {
   componentActive = false;
 }


}
```

Async Pipe

Product List Component

```
this.products$ = this.store.pipe(select(fromProduct.getProducts));
```

Product List View

```
*ngFor="let product of products$ | async"
```

When should you use the async pipe versus subscribing in a component class?

Component Subscription vs. Async Pipe

Component Subscription

```
this.productService.getProducts()
.subscribe(
   products => this.products = products
);
```

Async Pipe

```
<div *ngIf="products$ | async">
```

Demo

Unsubscribing from observables

Exception Handling in Effects

```
@Injectable()
export class ProductEffects {

    constructor(private productService: ProductService,
                private actions$:Actions) { }

    @Effect()
    loadProducts$ = this.actions$.pipe(
      ofType(ProductActionTypes.Load),
      mergeMap(action =>
        this.productService.getProducts().pipe(
         map(products => (new LoadSuccess(products)))
        )
      )
    );
}
```

Exception Handling in Effects

```
@Injectable()
export class ProductEffects {

   constructor(private productService: ProductService,
               private actions$:Actions) { }

   @Effect()
   loadProducts$ = this.actions$.pipe(
     ofType(ProductActionTypes.Load),
     mergeMap(action =>
       this.productService.getProducts().pipe(
        map(products => (new LoadSuccess(products)))
       )
     )
   );
}
```

Return new action

Exception Handling in Effects

```
@Injectable()
export class ProductEffects {

   constructor(private productService: ProductService,
               private actions$:Actions) { }

   @Effect()
   loadProducts$ = this.actions$.pipe(
     ofType(ProductActionTypes.Load),
     mergeMap(action =>
       this.productService.getProducts().pipe(
        map(products => (new LoadSuccess(products))),
        catchError(err => of(new LoadFail(err)))
       )
     )
   );
}
```

Return new action

Exception Handling in Effects

```
@Injectable()
export class ProductEffects {

   constructor(private productService: ProductService,
               private actions$:Actions) { }

   @Effect()
   loadProducts$ = this.actions$.pipe(
     ofType(ProductActionTypes.Load),
     mergeMap(action =>
       this.productService.getProducts().pipe(
        map(products => (new LoadSuccess(products))),
        catchError(err => of(new LoadFail(err)))
       )
     )
   );
}
```

Return new action

Exception Handling in Effects

```
@Injectable()
export class ProductEffects {

   constructor(private productService: ProductService,
               private actions$:Actions) { }

   @Effect()
   loadProducts$ = this.actions$.pipe(
     ofType(ProductActionTypes.Load),
     mergeMap(action =>
       this.productService.getProducts().pipe(
        map(products => (new LoadSuccess(products))),
        catchError(err => of(new LoadFail(err)))
       )
     )
   );
}
```

Return new action

Exception Handling in Effects

Add to interface

```
export interface ProductState {
 ...
 error: string;
}
```

Initialize state

```
const initialState: ProductState = {
 ...
 error:''
};
```

Make selector

```
export const getError = createSelector(
 getProductFeatureState,
 state => state.error
);
```

Exception Handling in Effects

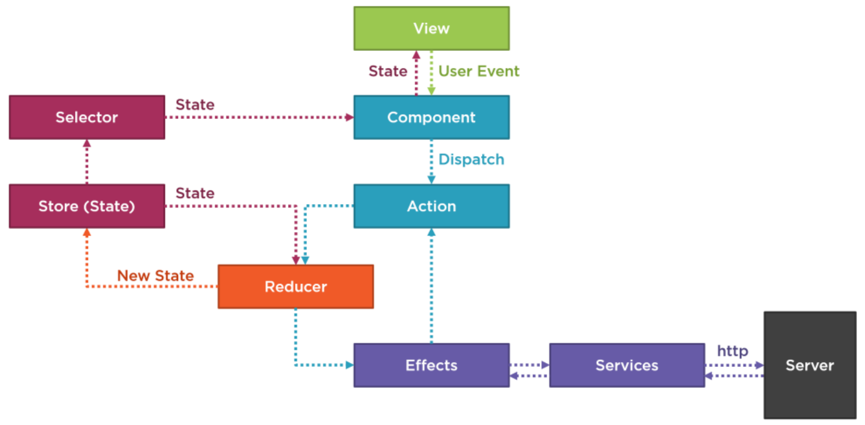
Add case statement

```
case ProductActionTypes.LoadFail: {
 return {
   ...state,
   products:[],
   error: action.payload
 };
}
```

Demo

Add exception handling to effect

Checklist: Using Effects



Install @ngrx/effects

Build the effect to process that action and dispatch the success and fail actions

Initialize the effects module in your root and feature modules

Register your effects in your root or feature modules

Process the success and fail actions in the reducer

Homework

Identify all subscriptions to the store
Hint: Look for "*// TODO: Unsubscribe*"
code comments

Add an OnDestroy lifecycle hook to the component

Initialize a componentActive flag to true

Set the componentActive flag to false in the
ngOnDestory method

Add a takeWhile pipe before the subscribe method
and use the componentActive property as a
predicate in this operator

https://github.com/DeborahK/Angular-NgRx-GettingStarted/tree/master/APM-Demo3