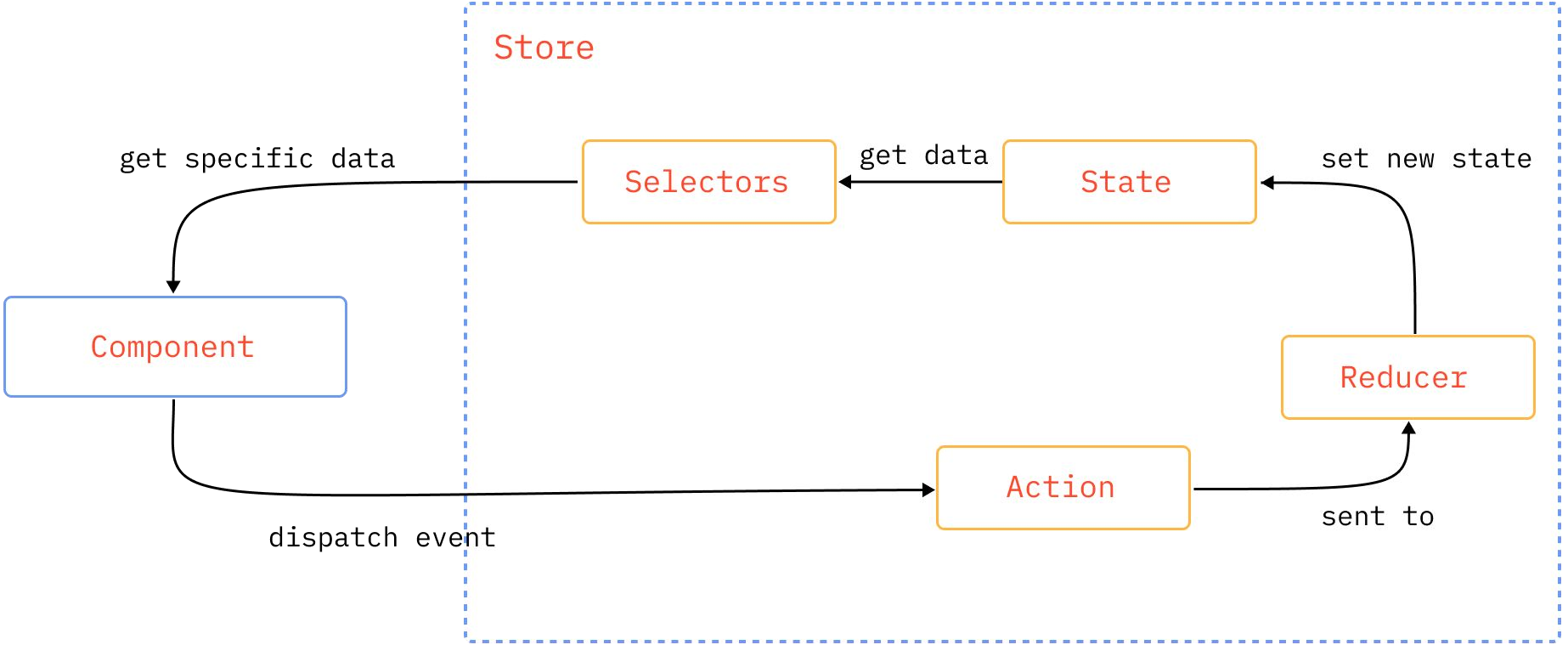
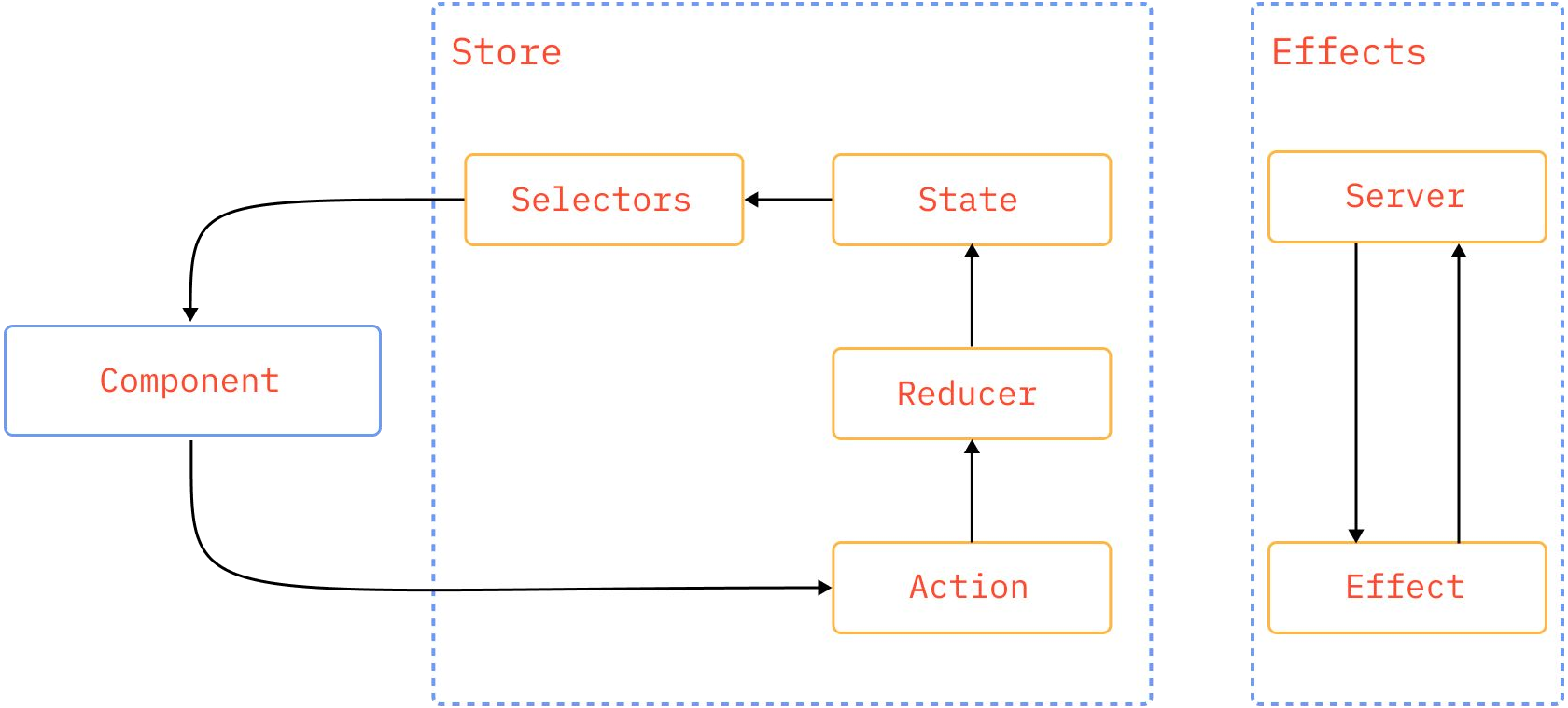
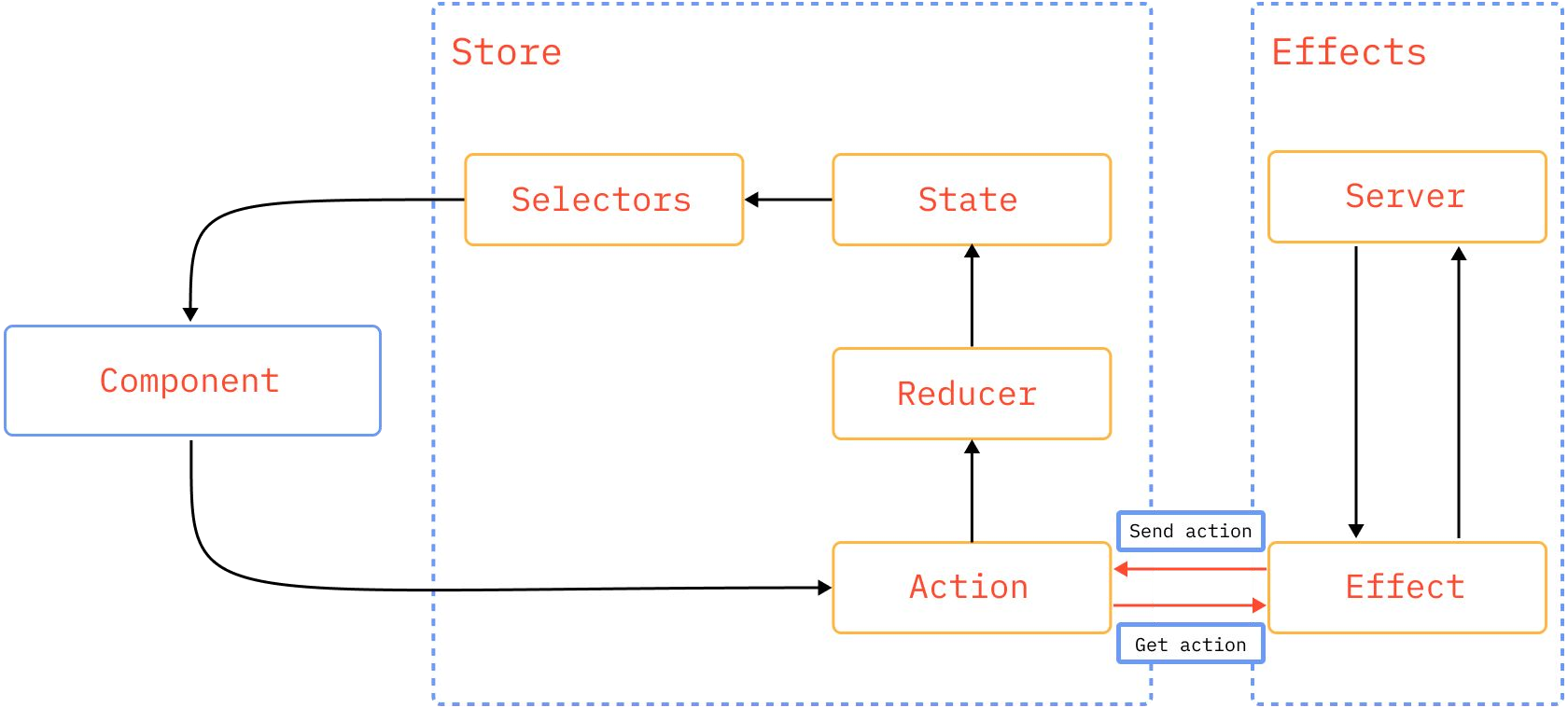
State Management Part 2

Лектор: Петър Маламов







Effects

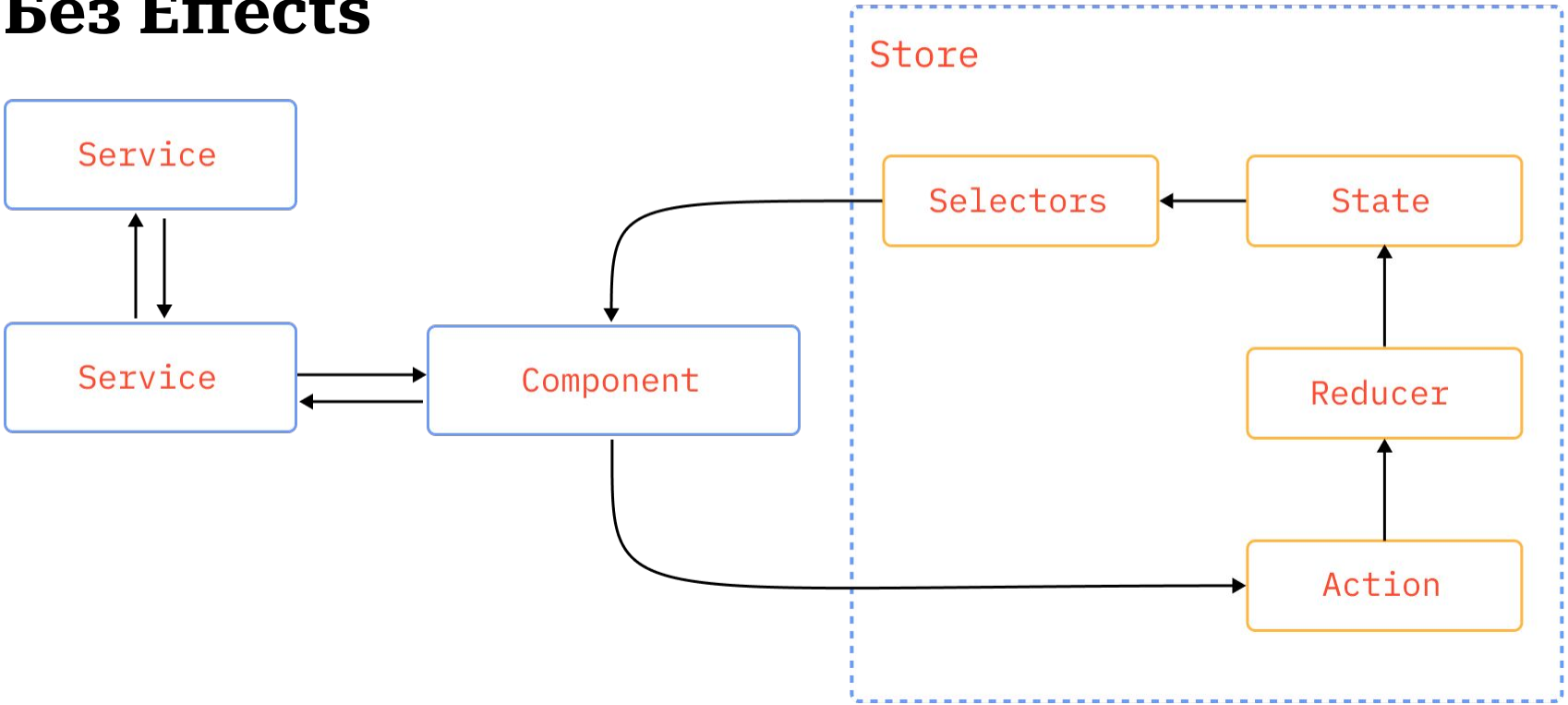
Какво е Effect?

Механизъм за управление на странични ефекти.

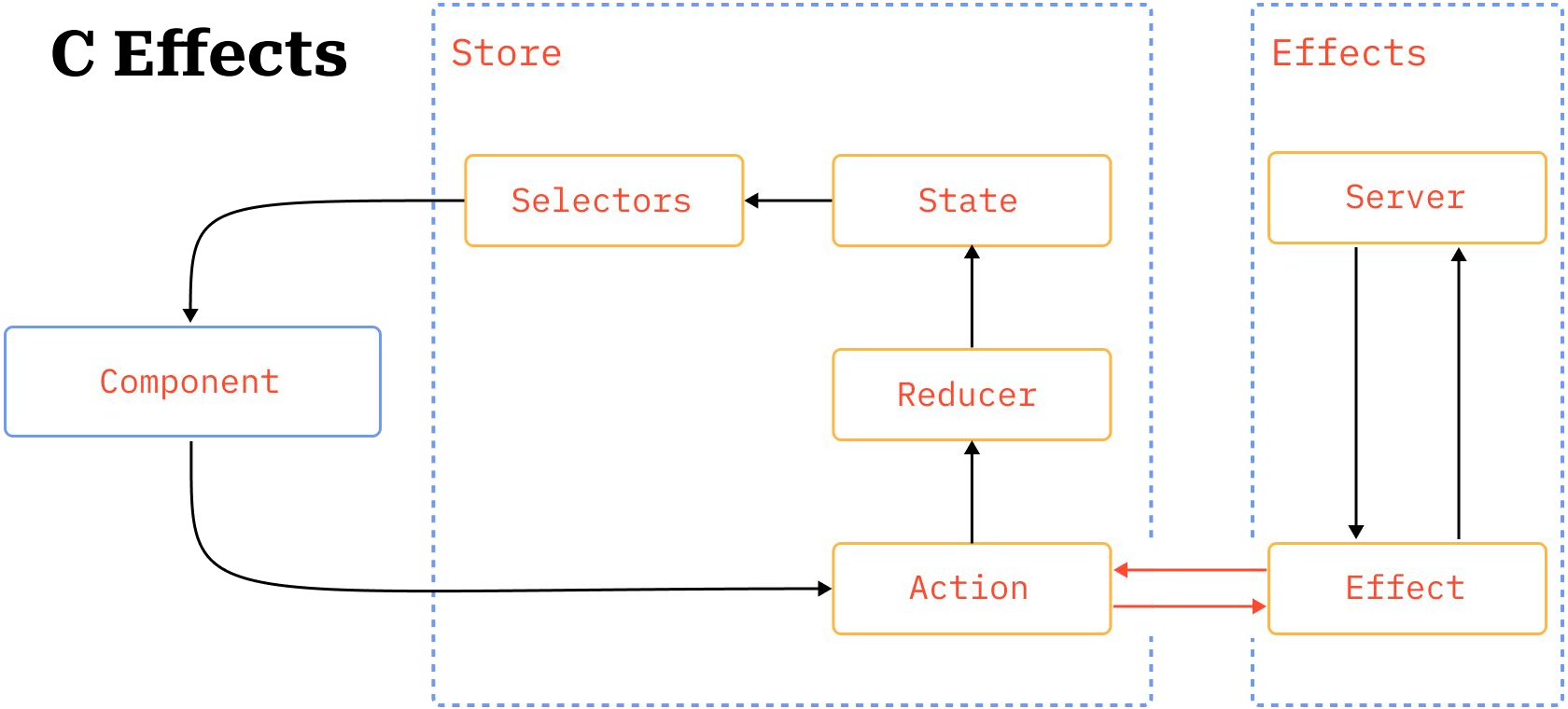
Позволява логиката за взаимодействие с външни системи да бъде изолирана извън компонентите.

Когато effects се използват заедно със Store, намаляват отговорностите на компонента.

Без Effects



C Effects



Как се създава Effect?

За да създадем ефект, използваме функцията **createEffect**, която приема observable и връща ново observable, съдържащ нов action.

Този observable може идва от **Actions service**-а и представлява поток от всички действия, изпратени към store-а.

Чрез оператора **ofType** филтрираме само тези действия, които са от интерес за конкретния ефект.

Как се създава Effect?

Ефектите могат да бъдат създадени по два начина:

- Като Injectable class
- Като функция

Създаване като Injectable class

TS effects.ts

```ts
@Injectable()
export class PostsEffects {
  constructor(private http: HttpClient, private actions$: Actions) {}

  loadPosts$ = createEffect(() =>
    this.actions$.pipe(
      ofType(loadPosts),
      exhaustMap(() =>
        this.http.get('https://jsonplaceholder.typicode.com/posts').pipe(
          tap((data) => {
            console.log(data);
          }),
          map((posts) => loadPostsSuccess({ posts }))
        )
      )
    )
  );
}
```

Създаване като функция

TS effects.ts

```ts
export const loadPostsEffect = createEffect(
  (actions$ = inject(Actions), http = inject(HttpClient)) => {
    return actions$.pipe(
      ofType(loadPosts),
      exhaustMap(() =>
        http.get('https://jsonplaceholder.typicode.com/posts').pipe(
          tap((data) => {
            console.log('function', data);
          }),
          map((posts) => loadPostsSuccess({ posts }))
        )
      )
    );
  },
  {
    functional: true,
  }
);
```

За да инжектираме services в ефекта, използваме inject.

Трябва да добавим флага functional за да може да ползваме ефекта като функция

Важно!

- Когато ефектите използват **Actions service** като източник, те слушат за всеки action, изпратен към store, като обработват само тези, които отговарят на критериите, зададени от оператора ofType.

- След като даден ефект обработи action, той трябва да изпрати нов action, за да актуализира store-а.

- **createEffect** може да приема всякакъв observable поток, при условие че той генерира нов action.

TS effects.ts

```ts
@Injectable()
export class PostsEffects {
  constructor(private http: HttpClient, private actions$: Actions) {}

  loadPosts$ = createEffect(() =>
    fromEvent(document, 'click').pipe(
      exhaustMap(() =>
        this.http.get('https://jsonplaceholder.typicode.com/posts').pipe(
          tap((data) => {
            console.log(data);
          }),
          map((posts) => loadPostsSuccess({ posts })),
          catchError(() => of(loadPostsError()))
        )
      )
    )
  );
}
```

При възникване на грешка

Effects са изградени върху observable потоци от RxJS и работят до възникване на грешка или завършване на потока.

За да продължат да работят при грешка или завършване, ефектите трябва да бъдат поставени в "flattening" оператори като **mergeMap**, **concatMap** или **exhaustMap**, които осигуряват тяхната стабилност.

TS effects.ts

```ts
@Injectable()
export class PostsEffects {
  constructor(private http: HttpClient, private actions$: Actions) {}

  loadPosts$ = createEffect(() =>
    this.actions$.pipe(
      ofType(loadPosts),
      exhaustMap(() =>
        this.http.get('https://jsonplaceholder.typicode.com/2posts').pipe(
          tap((data) => {
            console.log(data);
          }),
          map((posts) => loadPostsSuccess({ posts }))
        )
      ),
      catchError(() => of(loadPostsError()))
    )
  );
}
```

Ще приключи слушането за други actions

TS effects.ts

```ts
@Injectable()
export class PostsEffects {
  constructor(private http: HttpClient, private actions$: Actions) {}

  loadPosts$ = createEffect(() =>
    this.actions$.pipe(
      ofType(loadPosts),
      exhaustMap(() =>
        this.http.get('https://jsonplaceholder.typicode.com/2posts').pipe(
          tap((data) => {
            console.log(data);
          }),
          map((posts) => loadPostsSuccess({ posts })),
          catchError(() => of(loadPostsError()))
        )
      )
    )
  );
}
```

Ще продължи да слуша за други actions

Регистриране на Effects

За да може един ефект да извършва странични действия и да актуализира съответния state, е необходимо първо да го регистрираме в приложението.

Има два основни типа регистрация:

- **Root регистрация** - Започват да работят веднага, за да гарантират, че слушат за всички actions.
- **Feature регистрация -** Започват да слушат за actions веднага след добавянето на feature модула.

TS app.config.ts

```ts
export const appConfig: ApplicationConfig = {
  providers: [
    provideHttpClient(),
    provideRouter(routes),
    provideStore(),
    provideState({ name: 'counter', reducer: counterReducer }),

    provideEffects(effects),

  ],
};
```

Root регистрация

TS app.routes.ts

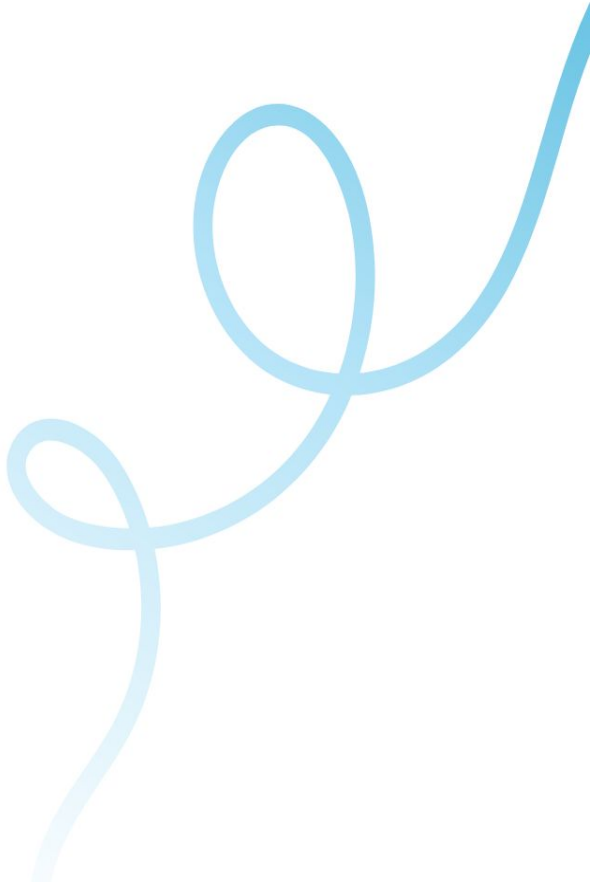
```ts
export const routes: Routes = [
  {
    path: 'posts',
    providers: [provideEffects(effects)],
  },
];
```

Feature регистрация

Важно!

Регистрирането на един и същи ефект многократно няма да доведе до многократно изпълнение.

Упражнение



Entity

Какво е NgRx Entity

Библиотека, която предоставя удобни инструменти за работа с колекции от обекти.

Тя улеснява управлението на state-а, като осигурява автоматично добавяне, обновяване, изтриване и нормализиране на обекти чрез уникални идентификатори(ids).

С помощта на **entity** можем да обработваме данни по ефективен и организиран начин, като се възползваме от вградените функции и оптимизации за работа с колекции.

Entity State

Представлява предварително дефиниран интерфейс за state-а на колекция от обекти. Този интерфейс дефинира как трябва да изглежда state-а на колекцията в store-а.

Интерфейсът на Entity State съдържа две основни части:

- **entities** - dictionary от обектите в колекцията, индексирани по id.
- **ids** - масив, който съдържа само идентификаторите на обектите в колекцията.

Важно!

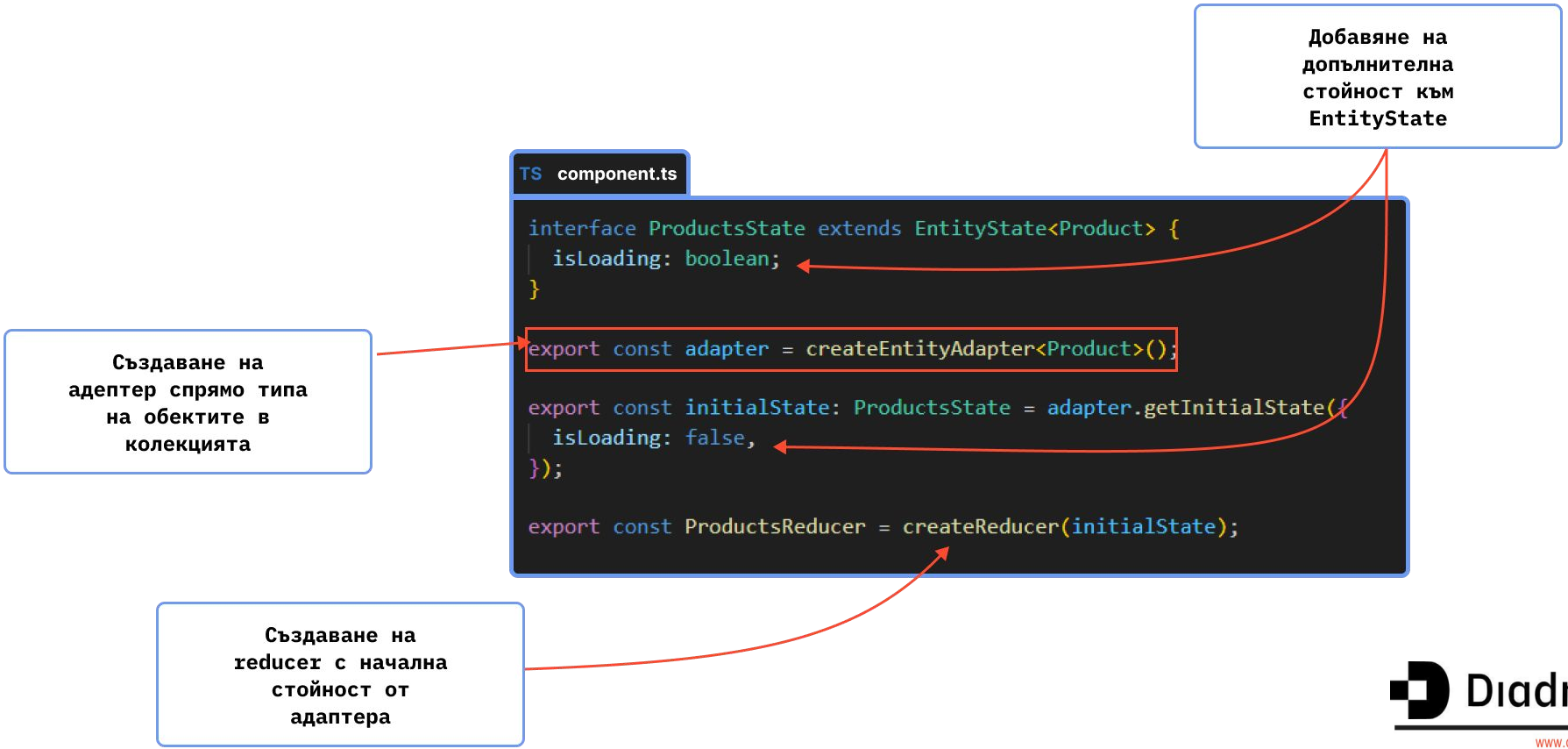
Всеки обект в колекцията трябва да има уникално поле, което го идентифицира (обикновено id).

Използване на NgRx Entity

За да създадем колекция от обекти, използваме функцията **createEntityAdapter**, която предоставя методи за управление на елементите в нея.

Тя приема конфигурация с два параметъра:

- **selectId**: Метод за избор идентификатор на колекцията (по подразбиране е id).
- **sortComparer**: Функция за сортиране на колекцията.



Entity Adapter методи

Entity адаптерът предоставя методи за операции върху единични или множество записи в колекцията.

Тези методи могат да променят един или повече записи едновременно.

Всеки метод връща нов state, ако са направени промени, или същия state, ако не са извършени промени.

Entity Adapter методи

- addOne
- setOne
- removeOne
- updateOne
- Други

TS component.ts

```ts
export const addProduct = createAction(
  'Add Product',
  props<{ product: Product }>()
);
export const loadProductsSuccess = createAction(
  'Load Product Success',
  props<{ products: Product[] }>()
);

export const ProductsReducer = createReducer(
  initialState,
  on(addProduct, (state, { product }) => adapter.addOne(product, state)),
  on(loadProductsSuccess, (state, { products }) =>
    adapter.addMany(products, { ...state, isLoading: false })
  )
);
```

Модифицира state-а чрез метода addOne от адаптера.

Промяна на стойност извън EntityState

Извличане на стойности от Entity State

Извличането на стойности от Entity State в NgRx става чрез използването на вградените функции на Entity Adapter.

Методът getSelectors, върнат от създадения адаптер за колекцията, предоставя функции за извличане на информация state-а.

TS component.ts

```ts
export const selectProductsState =
  createFeatureSelector<ProductsState>('products');

export const selectProductIds = createSelector(
  selectProductsState,
  adapter.getSelectors().selectIds
);
export const selectProductEntities = createSelector(
  selectProductsState,
  adapter.getSelectors().selectEntities
);
export const selectAllProducts = createSelector(
  selectProductsState,
  adapter.getSelectors().selectAll
);
```

Основни характеристики:

- Съхранява обектите по техните уникални идентификатори, което позволява бързо и ефективно търсене и актуализиране на елементи
- Предоставя вградените функции за добавяне, обновяване и премахване на обекти значително опростяват операциите върху колекциите.

NgRx + signals

NgRx + signals

NgRx предлага различни подходи за ефективно управление на state-а в приложението чрез използване на signals.

Основните подходи са:

- **Signal State** - минималистичен начин за управление на state-а
- **Signal Store** - цялостно решение за управление на state-а

Signal State

Инструмент, който предлага минималистичен подход за управление на състоянието в компоненти и services.

Създаването му става чрез функцията **signalState**, която приема обект като начална стойност.

При създаване на state-а, функцията връща разширена версия на signal, която притежава всички възможности **read-only signals**

TS component.ts

```ts
export class CounterComponent {
  state = signalState({ count: 0 });

  doubleCount = computed(() => {
    return this.state.count() * 2;
  });

  countEffect = effect(() => {
    console.log('the count is:', this.state.count());
  });
}
```

Важно!

Функцията **signalState** генерира сигнал за всеки атрибут от обекта.

Ако някой атрибут държи като стойност друг обект, тогава за него се генерира **DeepSignal**. Той може да бъде използван като обикновен read-only сигнал, като същевременно всеки вложен атрибут също е сигнал.

За по-добър изпълнение(performance) тези DeepSignals се инициализират lazily, когато някой ги достъпи.

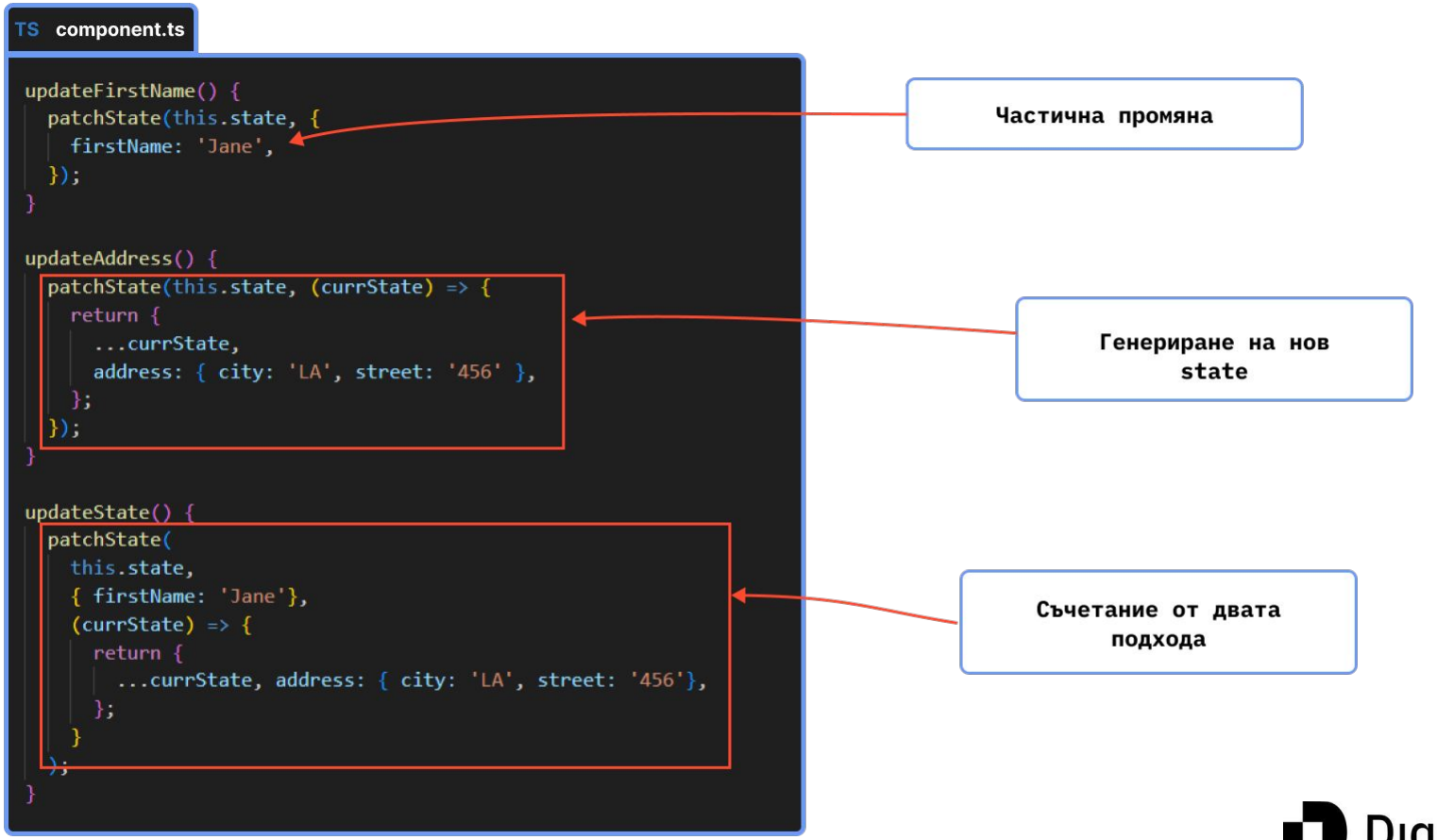
TS component.ts

```ts
export class UserComponent {
  state = signalState({
    firstName: 'John',
    lastName: 'Doe',
    address: { city: 'NY', street: '123' },
  });

  fullName = computed(() => {
    return this.state.firstName() + ' ' + this.state.lastName();
  });

  fullAddress = computed(() => {
    return this.state.address().city + ' ' + this.state.address().street;
  });

  effect = effect(() => {
    console.log('the count is:', this.state.address());
  });
}
```

Промяна на стойности в signal state

Стойността на signalState е read-only, което означава, че не може да бъде променяна директно.

За да променим стойността, трябва да използваме функцията **patchState**. Тя позволява частична промяна на данните или подаване на функция, която генерира нов state.



Важно!

Когато използваме функция за промяна на state-а, трябва да го правим по immutable начин, като винаги генерираме нова стойност.

Signal Store

Предлага цялостно решение за управление на state-а на приложението.

За да създадем store използваме функцията **signalStore**,която приема последователност от функции(features) на store-а.

При създаване на store-а, функцията връща **injectable service**, които може да бъде навсякъде в приложението.

Създаване на Signal Store

За да създадем store, трябва да подадем на функцията signalStore последователност от features.

Някои от features са:

- **withState**
- **withComputed**
- **withMethods**
- **withHooks**
- **withEntities**

withState

Използва се за добавяне на **state slice** към **store**-а.
Подобно на signalState, тук началният state също трябва да е обект.

За всяка част от state slice автоматично се създава съответен signal.

Функцията withState може да приема и factory фунцкия като входен аргумент. Тя се изпълнява в DI context, което позволява началният state да бъде получен от външен източник.

TS component.ts

```ts
const initialState: ProductsState = {
  products: [],
  isLoading: false,
  filter: { query: '', order: 'asc' },
};

export const ProductsStore = signalStore(
  withState(initialState),
);
```

Използване на Signal Store

При създаване на store-а се създава injectable service, което значи че може да го регистрираме както глобално така и локално.

TS component.ts

```ts
export const ProductsStore = signalStore(
  { providedIn: 'root' },
  withState(initialState)
);
```

Глобално регистриране

TS component.ts

```ts
@Component({
  selector: 'app-products',
  standalone: true,
  imports: [],
  templateUrl: './products.component.html',
  styleUrl: './products.component.css',
  providers: [ProductsStore],
})
export class ProductsComponent {
  state = inject(ProductsStore);
}
```

Локално регистриране

Четене на стойности от Signal Store

Четенето на стойности от Signal Store става чрез използване на автоматично генерираните сигнали за state.

TS component.ts

```ts
export class ProductsComponent {
  state = inject(ProductsStore);

  myEffect = effect(() => {
    console.log('The products are: ', this.state.products());
  });
}
```

withComputed

Използва се за добавяне на computed signals към state-а

Функцията **withComputed** приема factory фунцкия като входен аргумент, която се изпълнява в DI context.

Factory функцията приема инстанцията на store-а като аргумент, въз основа на който създава нови computed signals

TS component.ts

```ts
export const ProductsStore = signalStore(
  withState(initialState),
  withComputed(({ products, filter }) => {
    return {
      filteredProducts: computed(() => {
        return products().filter((p) => p.name.includes(filter.query()));
      }),
    };
  })
);
```

withMethods

Използва се за добавяне на методи към store-а, които могат да актуализират state-а или да изпълняват странични ефекти.

Функцията **withMethods** приема factory фунцкия като входен аргумент, която се изпълнява в DI context.

Factory функцията приема инстанцията на store-а като аргумент и връща обект, съдържащ методи.

TS component.ts

```ts
export const ProductsStore = signalStore(
  withState(initialState),
  withComputed(({ products, filter }) => {···
  }),
  withMethods((state, http = inject(HttpClient)) => {
    return {
      async loadProducts() {
        patchState(state, { isLoading: true });
        const products = await lastValueFrom(
          http.get<Product[]>('https://jsonplaceholder.typicode.com/posts')
        );
        patchState(state, { products, isLoading: false });
      },
    };
  })
);
```

Важно!

По подразбиране state-ът е защитен от промени извън store-а.

За да позволим външни промени на state-а, може да зададем **protectedState**: **false** в конфигурацията на store-а

TS component.ts

```ts
export const ProductsStore = signalStore(
  { protectedState: false },
  withState(initialState),
  withComputed(({ products, filter }) => {···
  }),
  withMethods((state, http = inject(HttpClient)) => {···
  })
);
```

TS component.ts

```ts
export class ProductsComponent {
  state = inject(ProductsStore);

  updateState(book: Book): void {
    patchState(this.state, { isLoading: true });
  }
}
```

withHooks

Използва се за добавяне на функции, чрез които можем да се “закачим” за различни стъпки от жизнения цикъл store-а.

Функциите, чрез които можем да се „закачим“ към жизнения цикъл на store-а, са:

- onInit - след инициализиране на store-а, намира се в DI context
- onDestroy - преди унищожаване на store-а

И двете функции приемат инстанцията на store-а като аргумент.

Създаване чрез
factory функция

TS component.ts

```ts
export const ProductsStore = signalStore(
  withState(initialState),
  withComputed(({ products, filter }) => { ···
  }),
  withMethods((state, http = inject(HttpClient)) => { ···
  }),
  withHooks((store) => {
    const logger = inject(Logger);
    return {
      onInit() {
        console.log('products', store.products());
      },
      onDestroy() {
        logger.warn('Store in being destroyed');
      },
    };
  })
);
```

TS component.ts

```ts
export const ProductsStore = signalStore(
  withState(initialState),
  withComputed(({ products, filter }) => { ···
  }),
  withMethods((state, http = inject(HttpClient)) => { ···
  }),
  withHooks({
    onInit(store) {
      inject(ProductsService).getProducts();
    },
    onDestroy(store) {
      console.log('Store is destroyed');
    },
  })
);
```

При onInit и във
factory функция се
намираме в DI
context

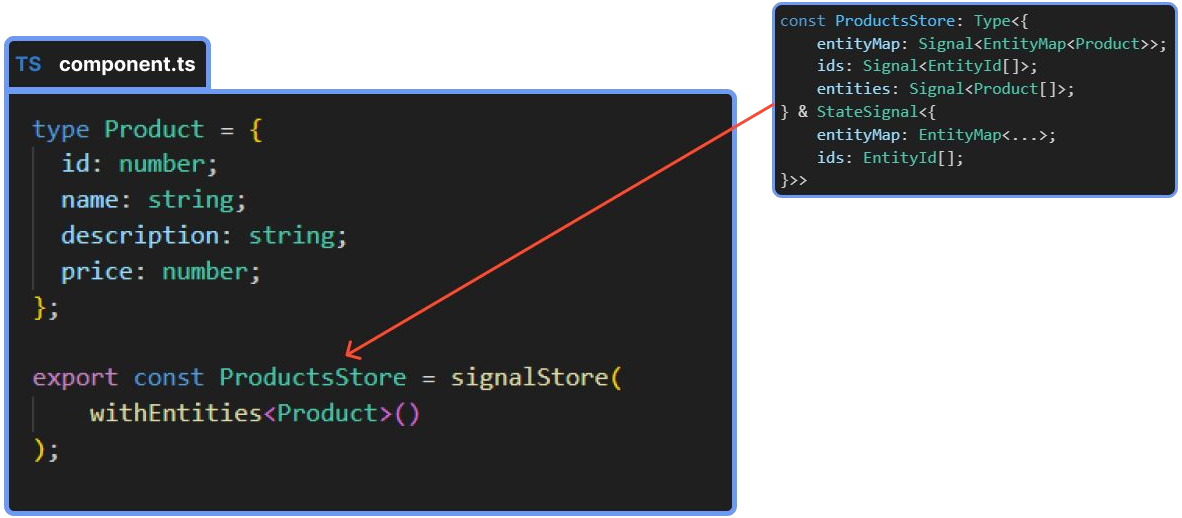
withEntities

Използва се за създаване на прост и ефективен начин за управление на колекции от обекти.

Всеки обект в колекцията трябва задължително да има уникално **ID**.

Функцията **withEntities** добавя към store-а три сигнала:

- ids - масив с всички id-та на обектите
- entityMap - map, където всеки ключ е id и всяка стойност е обект
- entries - масив с всички обекти. Този сигнал е **computed signal**



Модифициране на колекцията

Използвайки withEntities, NgRx ни предоставя предефинирани функции, които можем да използваме заедно с patchState, за да модифицираме колекцията от обекти.

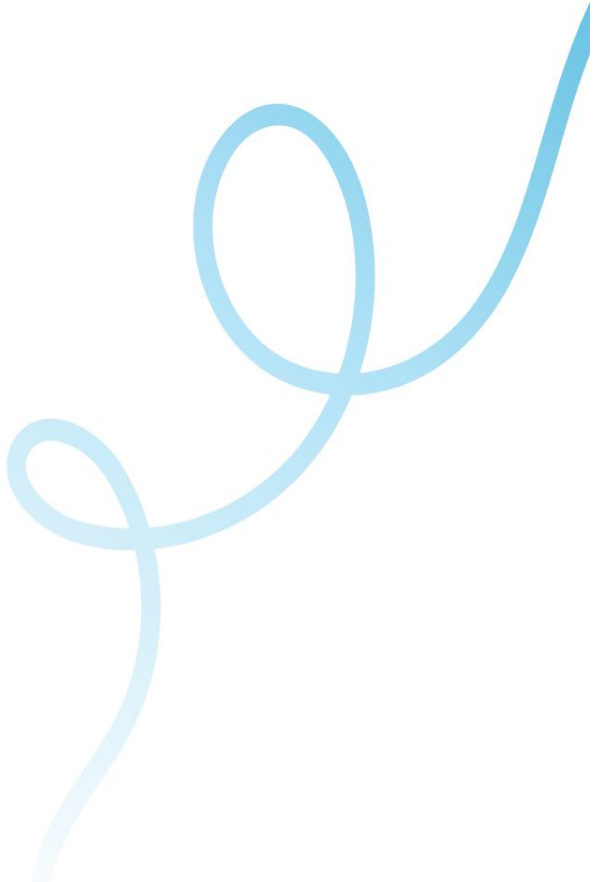
Някои от функциите са:

- addEntity
- addEntities
- updateEntity
- Други

TS component.ts

```ts
export const ProductsStore = signalStore(
  withEntities<Product>(),
  withMethods((store) => ({
    addProduct(product: Product): void {
      patchState(store, addEntity(product));
    },
    removeProduct(productId: number): void {
      patchState(store, removeEntity(productId));
    },
  }))
);
```

Упражнение



Благодаря за вниманието!