export class EntityCollectionServiceBase<

T,

S$ extends EntitySelectors$<T> = EntitySelectors$<T>

> implements EntityCollectionService<T> {

readonly dispatcher: EntityDispatcher<T>;

readonly selectors: EntitySelectors<T>;

readonly selectors$: S$;

constructor(

public readonly entityName: string,

serviceElementsFactory: EntityCollectionServiceElementsFactory

) {

entityName = entityName.trim();

const { dispatcher, selectors, selectors$ } = serviceElementsFactory.create<

T,

S$

>(entityName);

this.entityName = entityName;

this.dispatcher = dispatcher;

this.guard = dispatcher.guard;

this.selectId = dispatcher.selectId;

this.toUpdate = dispatcher.toUpdate;

this.selectors = selectors;

this.selectors$ = selectors$;

this.collection$ = selectors$.collection$;

this.count$ = selectors$.count$;

this.entities$ = selectors$.entities$;

this.entityActions$ = selectors$.entityActions$;

this.entityMap$ = selectors$.entityMap$;

this.errors$ = selectors$.errors$;

this.filter$ = selectors$.filter$;

this.filteredEntities$ = selectors$.filteredEntities$;

this.keys$ = selectors$.keys$;

this.loaded$ = selectors$.loaded$;

this.loading$ = selectors$.loading$;

this.changeState$ = selectors$.changeState$;

}

createEntityAction<P = any>(

op: EntityOp,

data?: P,

options?: EntityActionOptions

): EntityAction<P> {

return this.dispatcher.createEntityAction(op, data, options);

}

createAndDispatch<P = any>(

op: EntityOp,

data?: P,

options?: EntityActionOptions

): EntityAction<P> {

return this.dispatcher.createAndDispatch(op, data, options);

}

dispatch(action: Action): Action {

return this.dispatcher.dispatch(action);

}

get store() {

return this.dispatcher.store;

}

guard: EntityActionGuard<T>;

selectId: IdSelector<T>;

toUpdate: (entity: Partial<T>) => Update<T>;

add(entity: T, options?: EntityActionOptions): Observable<T> {

return this.dispatcher.add(entity, options);

}

cancel(

correlationId: any,

reason?: string,

options?: EntityActionOptions

): void {

this.dispatcher.cancel(correlationId, reason, options);

}

delete(entity: T, options?: EntityActionOptions): Observable<number | string>;

delete(

key: number | string,

options?: EntityActionOptions

): Observable<number | string>;

delete(

arg: number | string | T,

options?: EntityActionOptions

): Observable<number | string> {

return this.dispatcher.delete(arg as any, options);

}

getAll(options?: EntityActionOptions): Observable<T[]> {

return this.dispatcher.getAll(options);

}

getByKey(key: any, options?: EntityActionOptions): Observable<T> {

return this.dispatcher.getByKey(key, options);

}

getWithQuery(

queryParams: QueryParams | string,

options?: EntityActionOptions

): Observable<T[]> {

return this.dispatcher.getWithQuery(queryParams, options);

}

load(options?: EntityActionOptions): Observable<T[]> {

return this.dispatcher.load(options);

}

update(entity: Partial<T>, options?: EntityActionOptions): Observable<T> {

return this.dispatcher.update(entity, options);

}

upsert(entity: T, options?: EntityActionOptions): Observable<T> {

return this.dispatcher.upsert(entity, options);

}

addAllToCache(entities: T[], options?: EntityActionOptions): void {

this.dispatcher.addAllToCache(entities, options);

}

addOneToCache(entity: T, options?: EntityActionOptions): void {

this.dispatcher.addOneToCache(entity, options);

}

addManyToCache(entities: T[], options?: EntityActionOptions): void {

this.dispatcher.addManyToCache(entities, options);

}

clearCache(): void {

this.dispatcher.clearCache();

}

removeOneFromCache(entity: T, options?: EntityActionOptions): void;

removeOneFromCache(key: number | string, options?: EntityActionOptions): void;

removeOneFromCache(

arg: (number | string) | T,

options?: EntityActionOptions

): void {

this.dispatcher.removeOneFromCache(arg as any, options);

}

removeManyFromCache(entities: T[], options?: EntityActionOptions): void;

removeManyFromCache(

keys: (number | string)[],

options?: EntityActionOptions

): void;

removeManyFromCache(

args: (number | string)[] | T[],

options?: EntityActionOptions

): void {

this.dispatcher.removeManyFromCache(args as any[], options);

}

updateOneInCache(entity: Partial<T>, options?: EntityActionOptions): void {

// update entity might be a partial of T but must at least have its key.

// pass the Update<T> structure as the payload

this.dispatcher.updateOneInCache(entity, options);

}

updateManyInCache(

entities: Partial<T>[],

options?: EntityActionOptions

): void {

this.dispatcher.updateManyInCache(entities, options);

}

upsertOneInCache(entity: Partial<T>, options?: EntityActionOptions): void {

this.dispatcher.upsertOneInCache(entity, options);

}

upsertManyInCache(

entities: Partial<T>[],

options?: EntityActionOptions

): void {

this.dispatcher.upsertManyInCache(entities, options);

}

setFilter(pattern: any): void {

this.dispatcher.setFilter(pattern);

}

setLoaded(isLoaded: boolean): void {

this.dispatcher.setLoaded(!!isLoaded);

}

setLoading(isLoading: boolean): void {

this.dispatcher.setLoading(!!isLoading);

}

collection$: Observable<EntityCollection<T>> | Store<EntityCollection<T>>;

count$: Observable<number> | Store<number>;

entities$: Observable<T[]> | Store<T[]>;

entityActions$: Observable<EntityAction>;

entityMap$: Observable<Dictionary<T>> | Store<Dictionary<T>>;

errors$: Observable<EntityAction>;

filter$: Observable<any> | Store<any>;

filteredEntities$: Observable<T[]> | Store<T[]>;

keys$: Observable<string[] | number[]> | Store<string[] | number[]>;

loaded$: Observable<boolean> | Store<boolean>;

loading$: Observable<boolean> | Store<boolean>;

changeState$: Observable<ChangeStateMap<T>> | Store<ChangeStateMap<T>>;

}