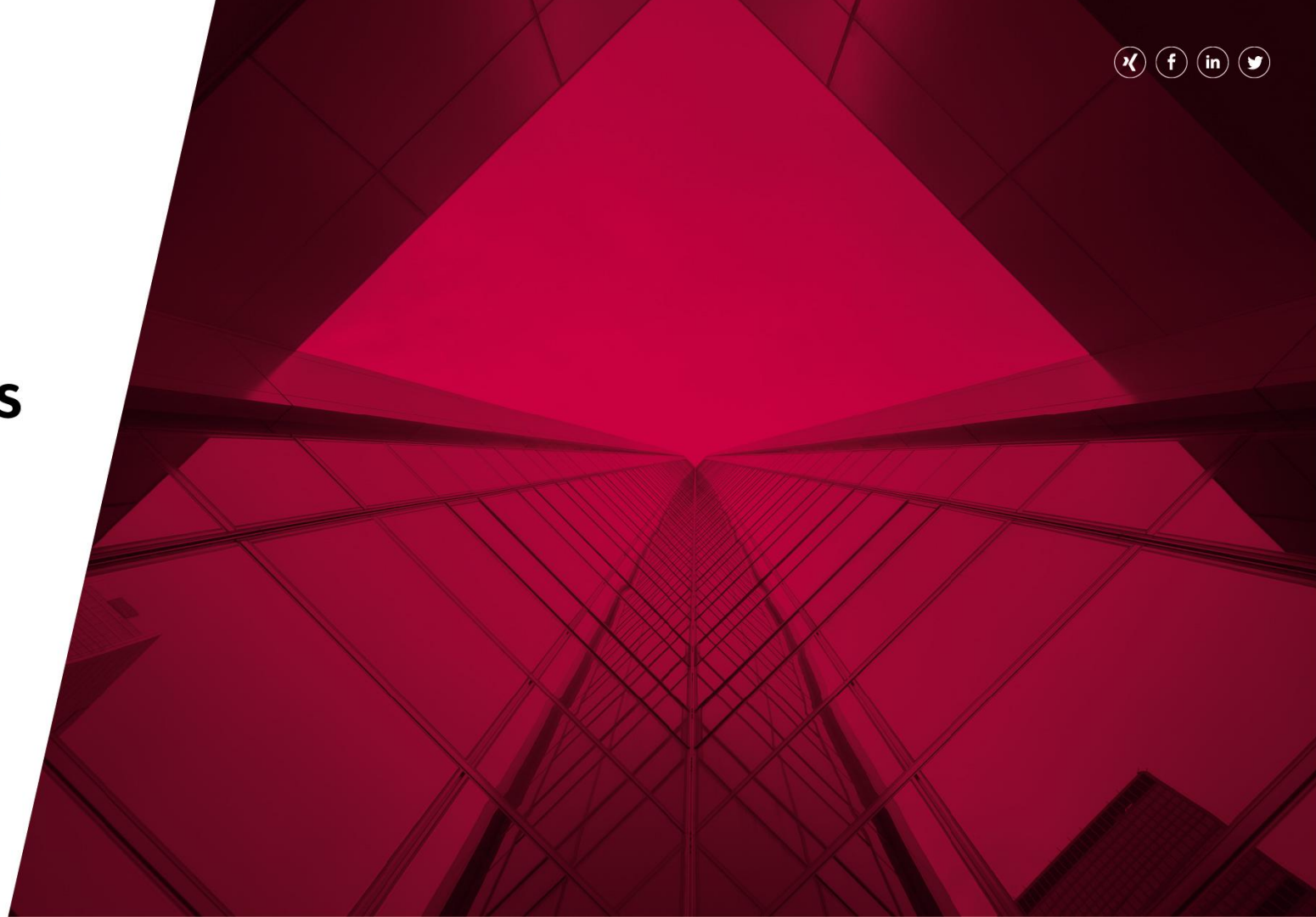
ANGULAR
ARCHITECTS
INSIDE KNOWLEDGE



Workshop

Angular State Management

ANGULAR
ARCHITECTS
INSIDE KNOWLEDGE

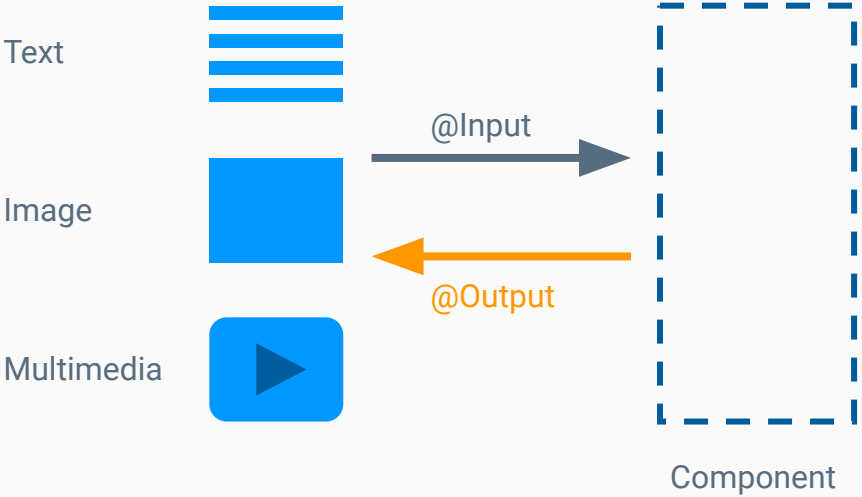
Divide & Conquer

Presentational- & Container-Components

ANGULAR
ARCHITECTS
INSIDE KNOWLEDGE

Presentational Components

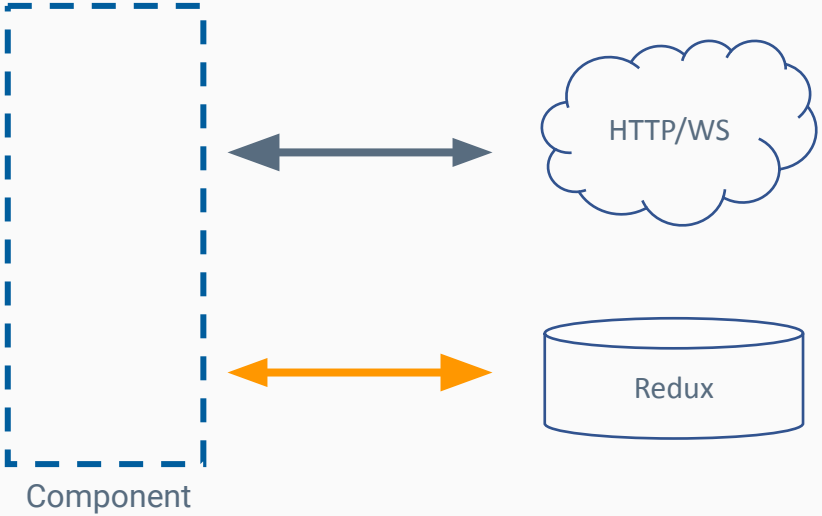
- Are reusable building blocks
- Do not consume data services
- Only use @Input and @Output
- Isolate the presentation logic



ANGULAR ARCHITECTS
INSIDE KNOWLEDGE

Container Components

- Orchestrate components
  - Host Components
  - Setup Communication
- Provide data from APIs
- Made for a certain use case

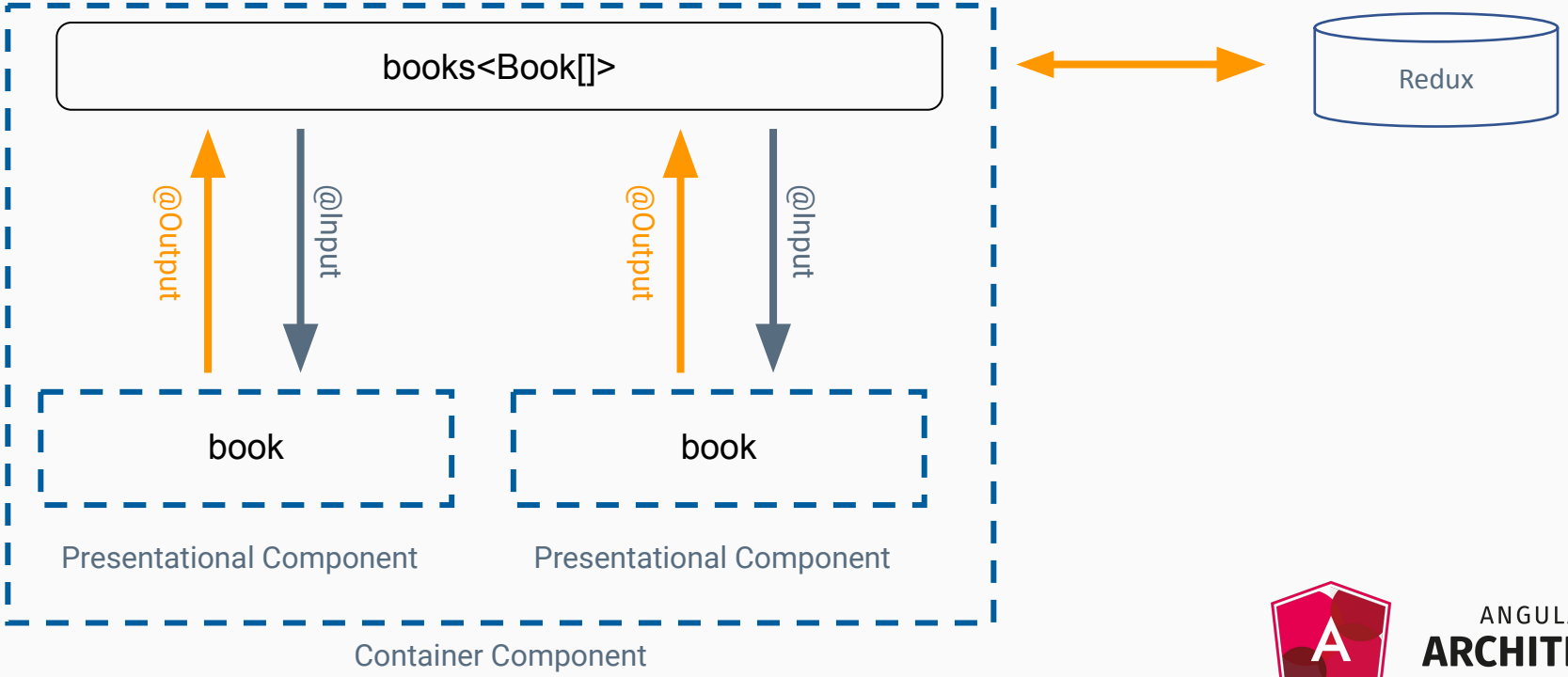


ANGULAR
ARCHITECTS
INSIDE KNOWLEDGE

Example

- Application with two views
  - Home-screen with featured Books
  - Category-list with Books
- We need the same presentation of a book in both views
- We should create a reusable component

Example



ANGULAR ARCHITECTS
INSIDE KNOWLEDGE

Container vs. Presentational

We **don't necessarily need to extract all** the rendering logic of every component we build into a separate presentation component.

It's more about **building the components that make the most sense**.

Redux

Architecture for centralised State Management

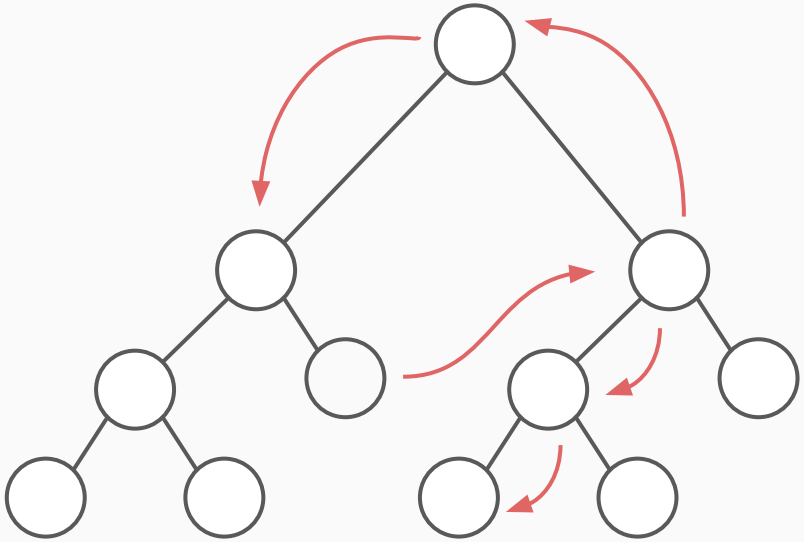
ANGULAR
ARCHITECTS
INSIDE KNOWLEDGE

Helps to manage the state of your application

Why?

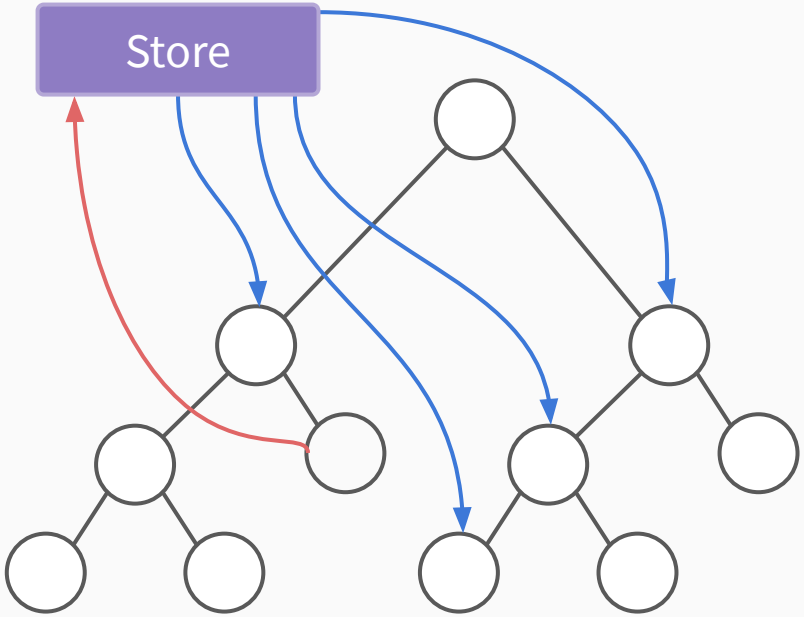
- Unidirectional data flow
- Predictable state changes and rendering
- Alternative UIs while reusing most of the business logic.

Why?



This is how we manage state at the moment

State management with Redux

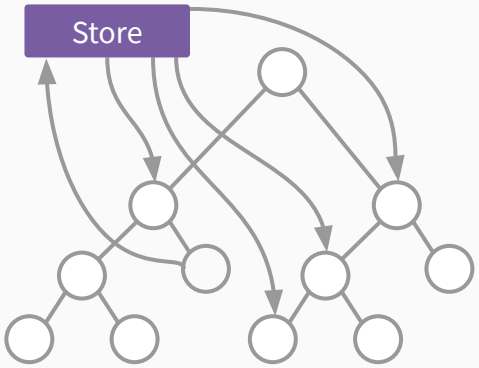


Everything is dispatched from and to **one global store**

ANGULAR ARCHITECTS
INSIDE KNOWLEDGE

Redux principles

1. The Store is the single source of truth
2. The Store is read only
3. Pure functions mutate application state

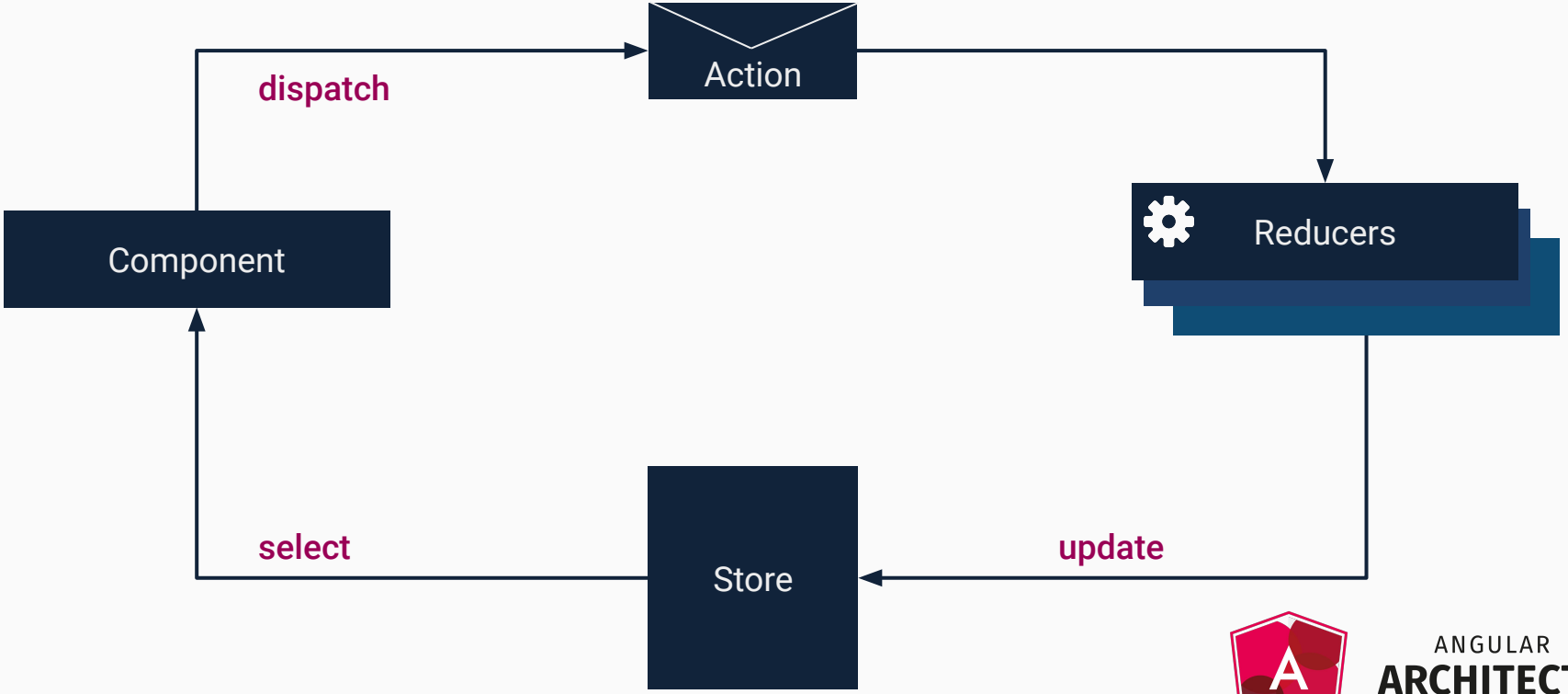


State management with Redux

Redux asks you to:

- Describe application state as plain objects and arrays
- Describe changes in the system as plain objects
- Describe the logic for handling changes as *pure functions*

Redux Architecture

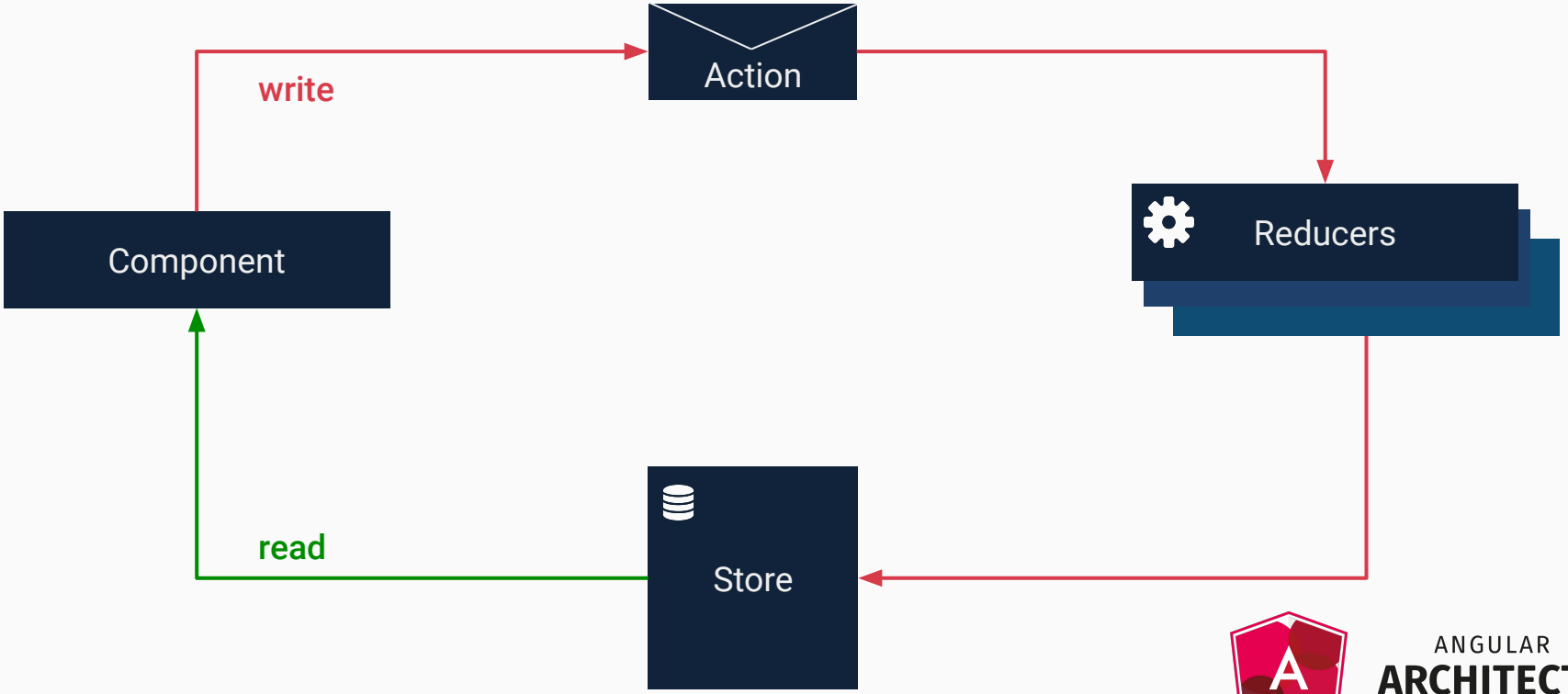


ANGULAR
ARCHITECTS
INSIDE KNOWLEDGE

Redux Architecture | Flow

- Component dispatches an action.
- Action passes multiple reducer functions.
- A reducer that is responsible for the action can mutate the store
  - Mutation is done by creating a new state.
  - The existing state cannot be changed since it is **read only**.
- Component can select several parts from the store.

Redux Architecture



ANGULAR
ARCHITECTS
INSIDE KNOWLEDGE

Redux is CQS

- Redux is about **Command-Query-Segregation** (CQS).
- **Reading** and **writing** logic are seperated.
- This leads to smaller but also more decentralized code.

Action

- Consider an action to be an **envelope** you send to a certain recipient.
- The **recipient** will be a reducer function.

Action anatomy

```
{
  type: 'My unique action type'

}
```

An action is an object that must
Have a unique type.

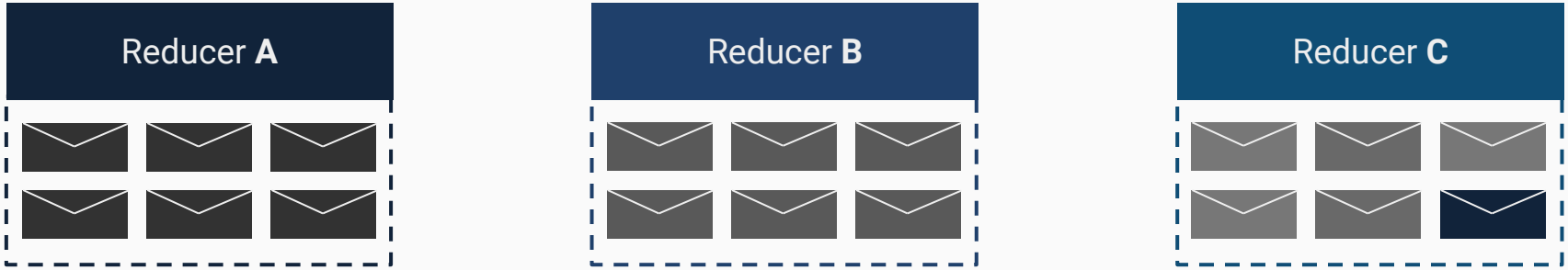
Action anatomy

```
{
  type: 'My unique action type',
  payload: { /* some optional payload */ }
}
```

The payload is optional.

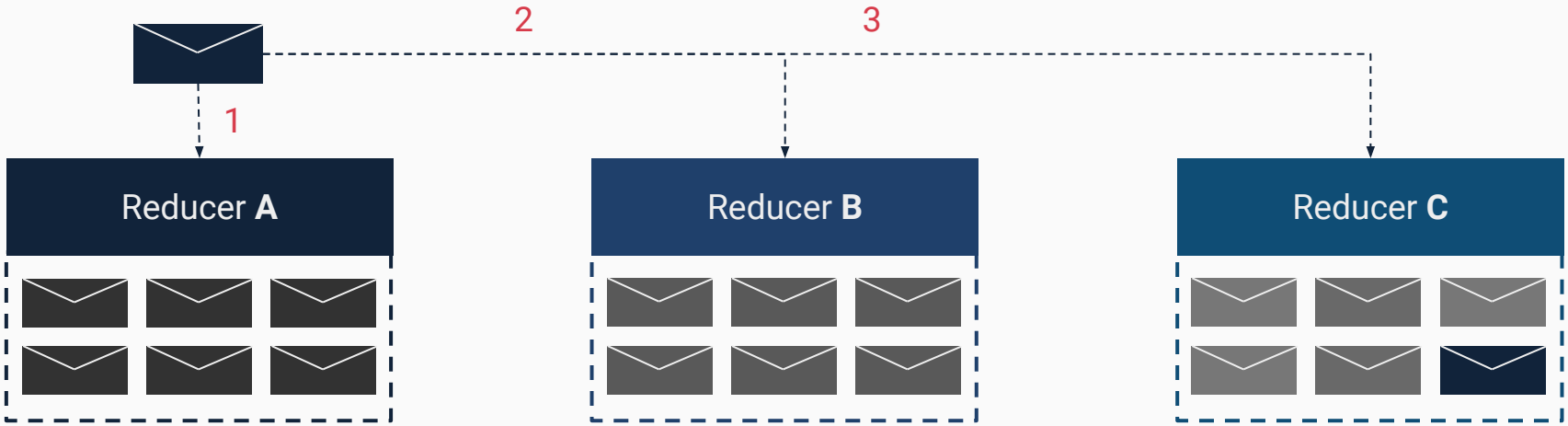
Reducer

- Each reducer is responsible for a certain amount of actions.



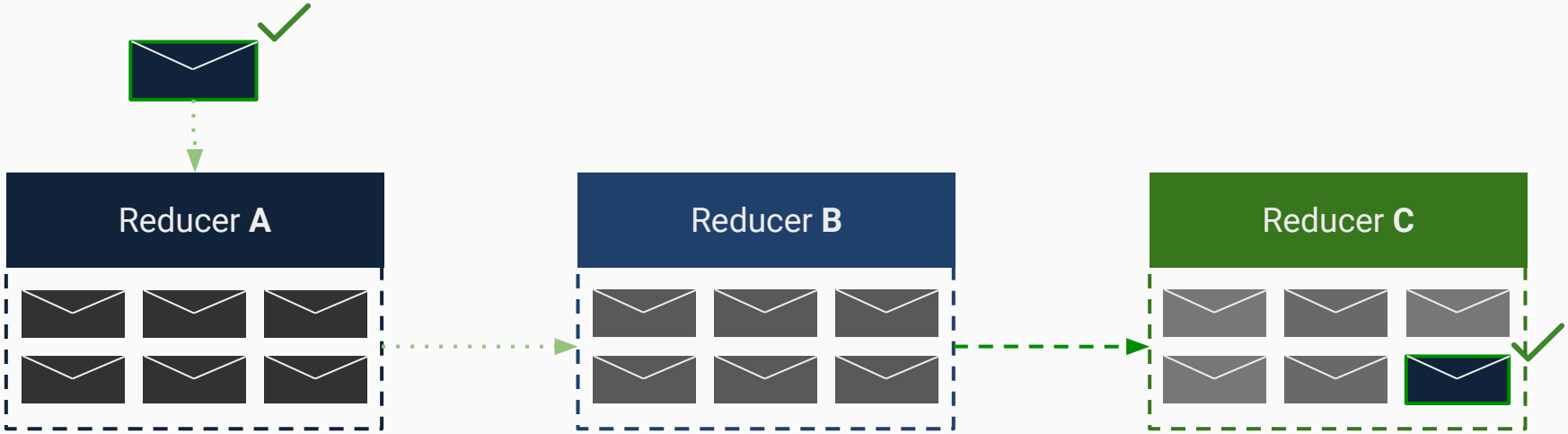
Reducer

- An action passes each reducer.



Reducer

- If the action type matches, a state update is executed.



Store

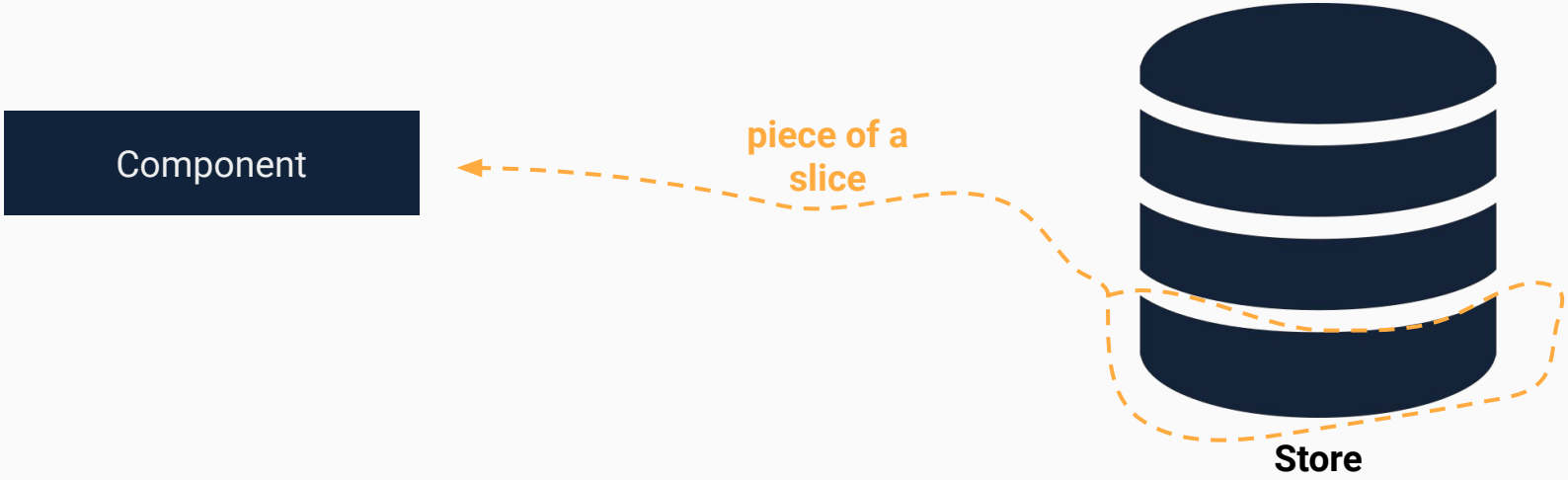
<code>

State is stored in-memory.

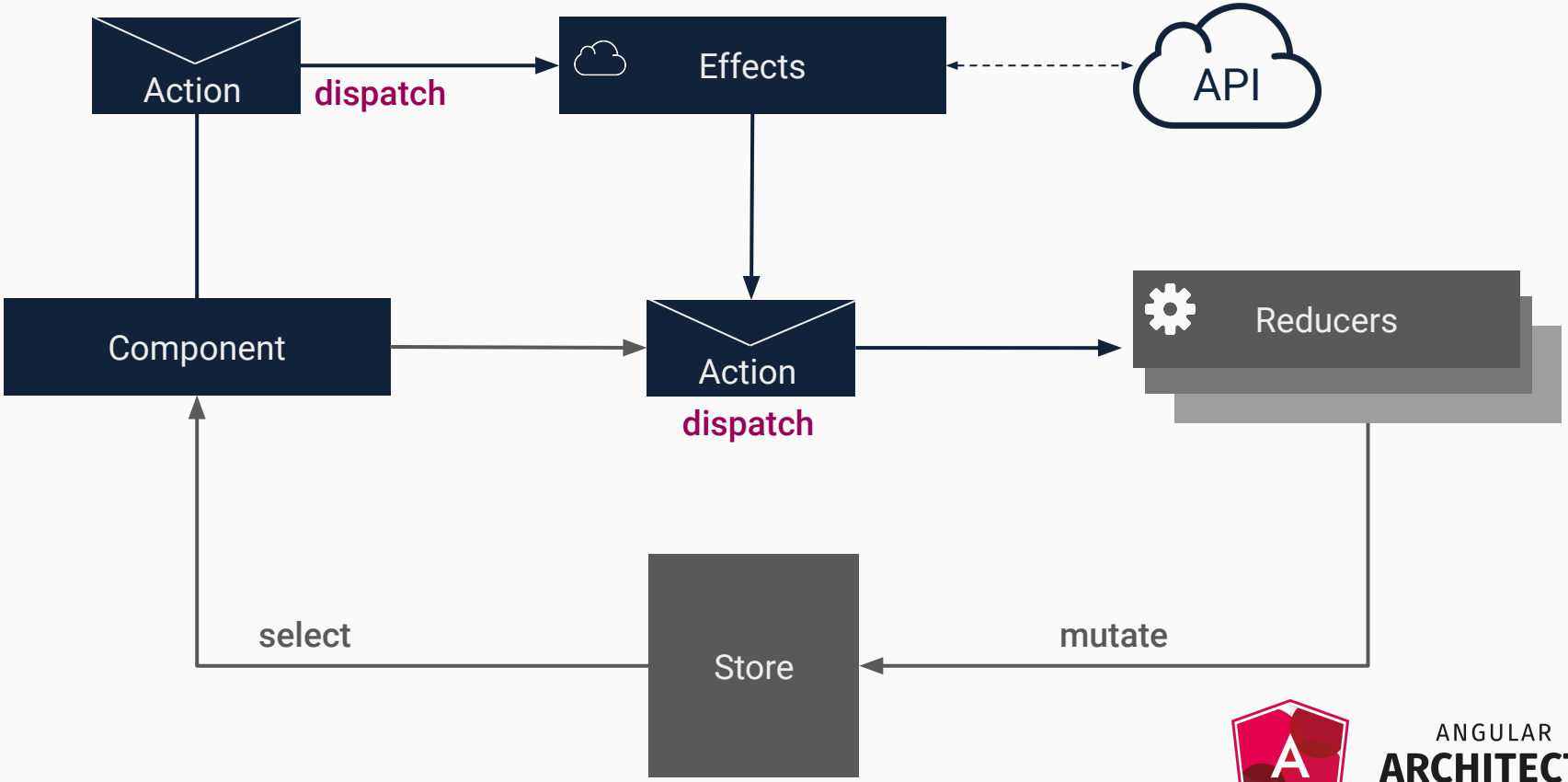
```
{
  featureA: {
    slice1: { … }
    slice2: { … }
  },
  featureB: {
    slice1: { … }
    slice2: { … }
  }
}
```

Store

- Components can select the information they need



Isolate side effects



ANGULAR
ARCHITECTS
INSIDE KNOWLEDGE

Redux Store

Redux uses a single store to manage everything.

```
{
    books {
        collection,
        Search
        currentBook
    }
}
```

*POJO = Plain Old JavaScript Object

Actions

Redux Action

<code>

A redux action is an object with a type and an optional payload that describes a state change

```
{
  type: 'ADD_BOOK',
  book: {...}
}
```

Action Creators

ANGULAR
ARCHITECTS
INSIDE KNOWLEDGE

Action Creator

<code>

An action creator is a function that returns an action

```
function addBook(book) {
  return {
    type: 'ADD_BOOK',
    book
  }
}
```

Reducers

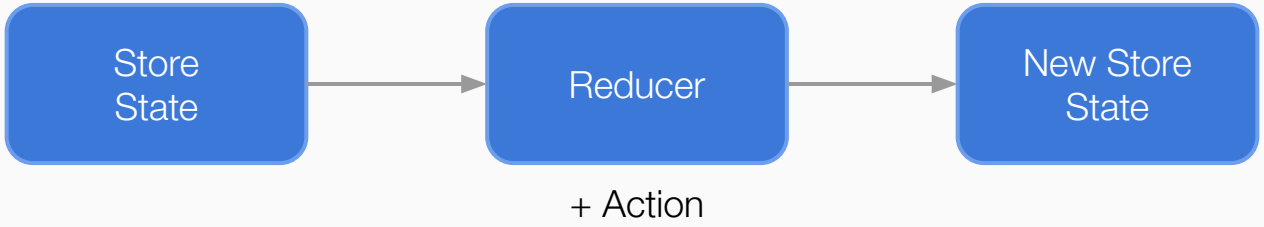
A Reducer transfers the

store to another state

ANGULAR
ARCHITECTS
INSIDE KNOWLEDGE

Reducers

A reducer takes a state and an action and always returns a **new state**.



Reducer

<code>

A reducer implementing the actual state change for an action type

```
function reducer(state, action) {
  switch(action.type) {
    case 'ADD_BOOK':
      let newState = { ...state }; // shallow copy of the state
      newState.books = [...state.books, action.book];
      return newState;
    case ...
  }
}
```

Pure functions

ANGULAR
ARCHITECTS
INSIDE KNOWLEDGE

A pure function **always** returns

the same output for a given input

Why / What you'll learn

Pure functions

- have no side effects
- are easy to reason about
- are easily testable

Pure function

<code>

This is a pure function

```
function(n) {
  return n * n;
}
```

Pure function

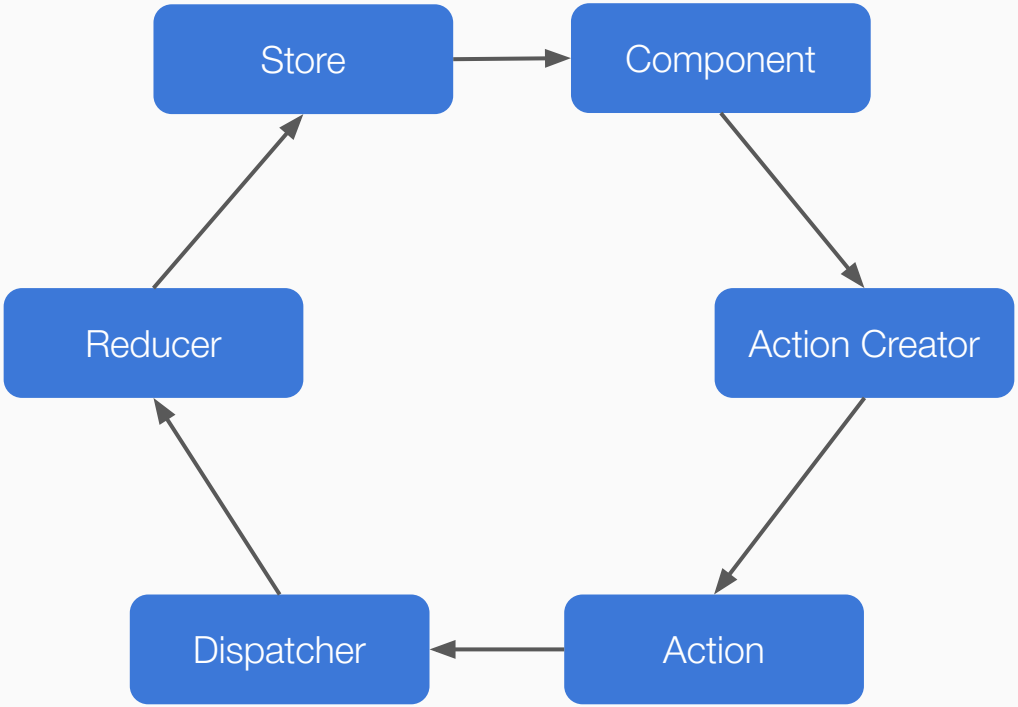
<code>

This is **NOT** a pure function

```
function addMinutes(n) {
  var now = new Date();
  return now.setMinutes(now.getMinutes() + n);
}
```

Complete Redux cycle

Redux cycle



@ngrx/store

Inspired by redux

ANGULAR
ARCHITECTS
INSIDE KNOWLEDGE

@ngrx/store is a controlled state container designed to help write performant, consistent applications on top of Angular

ANGULAR
ARCHITECTS
INSIDE KNOWLEDGE

@ngrx/store - Core Principles

- State is a single immutable data structure
- Actions describe state changes
- Pure functions called reducers take the previous state and the next action to compute the new state
- State accessed with the Store, an observable of state and an observer of actions

Store

ANGULAR
ARCHITECTS
INSIDE KNOWLEDGE

Store

<code>

Short recap

```
{
  featureA: {
    slice1: { … }
    slice2: { … }
  },
  featureB: {
    slice1: { … }
    slice2: { … }
  }
}
```

Store

<code>

Usage of the Store-Service in a component

```
import { Store } from '@ngrx/store';

export class ContainerComponent {
  constructor(private store: Store) {
    // Write
    this.store.dispatch(someAction);

   // Read
   const dataFromStore = this.store.select(/* projector function*/)
  }
}
```

Task

Setup

ANGULAR
ARCHITECTS
INSIDE KNOWLEDGE

Action

Action

<code>

Use action creators

```
import { createAction } from '@ngrx/store';

export const loadBooksStart = createAction('[Book] Load Books Started');
```

Action

<code>

Create action having a payload

```
import { createAction, props } from '@ngrx/store';

export const loadBooksComplete = createAction(
  '[Book] Load Books Completed',
  props<{ books: Book[] }>()
);
```

Actions

Actions are strictly typed simplifying the handling in reducers and effects.

```
const loadBooksComplete: ActionCreator<"[Book] Load Books Completed", (props: {
    books: Book[];
}) => {
    books: Book[];
} & TypedAction<"[Book] Load Books Completed">>
```

```
export const loadBooksComplete = createAction('[Book] Load Books Completed', props<{ books: Book[] }>());
```

ANGULAR ARCHITECTS
INSIDE KNOWLEDGE

Task

Dispatch an Action

Reducer

Store and its reducers

<code>

Each slice is managed by a reducer function

```
{
  featureA: {
    sliceA1: sliceA1Reducer
    sliceA2: sliceA2Reducer
  },
  featureB: {
    sliceB1: sliceB1Reducer
    sliceB2: sliceB2Reducer
  }
  featureC: featureCReducer
}
```

💁‍♀️ A feature can also be represented by a single reducer function.

Reducer setup

<code>

```
import { createReducer, on } from '@ngrx/store';
import { loadBooksStart, loadBooksComplete } from './books.actions';

export const reducer = createReducer(
  initialState,
  on(loadBooksStart, (state, action) => { /* … */ },
  on(loadBooksComplete, (state, action) => { /* … */ }
)
```

Reducer setup

<code>

Registering a reducer in the root state

```
@NgModule({
  imports: [
    //  ...
    StoreModule.forRoot(
      { stateSliceName: reducer },
      { /* options */}
    )
  ]
})
export AppModule { }
```

Reducer setup

<code>

Registering a reducer in a feature

```
@NgModule({
  imports: [
    //  ...
    StoreModule.forFeature('book', bookReducers)
  ]
})
export BookModule { }
```

Task

Reducer

ANGULAR
ARCHITECTS
INSIDE KNOWLEDGE

Store Select

ANGULAR
ARCHITECTS
INSIDE KNOWLEDGE

Store Select

<code>

Inject the Store Service

```
import { Store } from '@ngrx/store';

@Component({ ... })
export class BookListComponent {
  constructor(private store: Store) {}
}
```

Store Select

<code>

Use select projection function

```
import { Store } from '@ngrx/store';

@Component({ ... })
export class BookListComponent {
  constructor(private store: Store) {
    this.books$ = this.store.select(
      state => state['book'].bookCollection.entities
    );
  }
}
```

🤔 magic strings & we have no type-safety here.

Store Select

<code>

Store Service can be typed manually **until** we find a better solution.

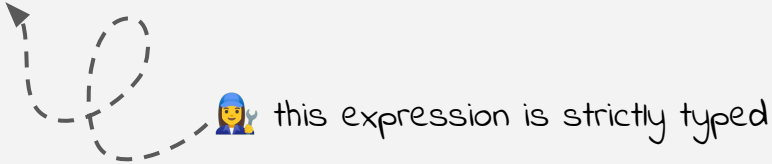
```
import { bookFeatureName, BookCollectionSlice } from '@store/book';

@Component({ ... })
export class BookListComponent {
  constructor(
    private store: Store<
      { [bookFeatureName]: { bookCollection: BookCollectionSlice } }
    >
  ) {}
}
```

ARCHITECTS
INSIDE KNOWLEDGE

Get the Observable

<code>

```
@Component({ ... })
export class BookListComponent {
  constructor(/* ... */) {
    this.books$ = this.store.select(
      state => state[bookFeatureName].bookCollection.entities
    );
  }
}
```



Task

Store Select

ANGULAR
ARCHITECTS
INSIDE KNOWLEDGE

Selector

Selectors are like handy **bookmarks**. They allow you to quickly get data from a certain location in the Store - just like a bookmark allows you to directly navigate to a website.

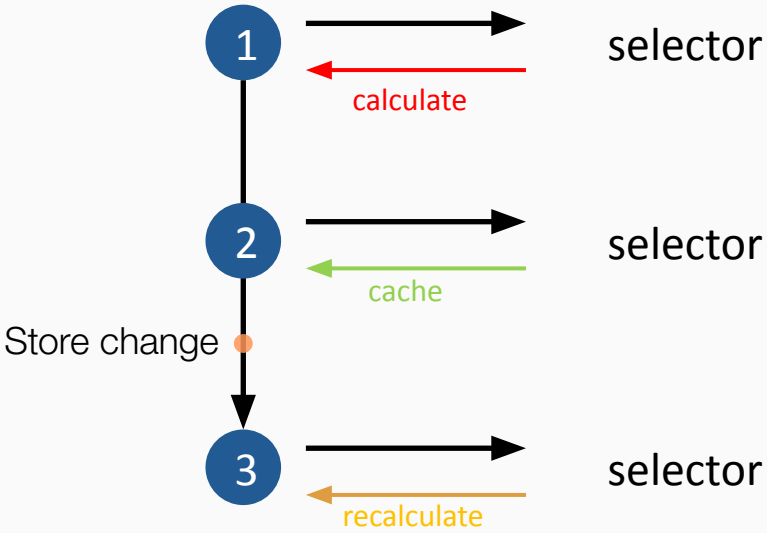
Selectors

<code>

Create Selectors

```
import { createFeatureSelector, createSelector } from '@ngrx/store';

export const bookFeature = createFeatureSelector<
  { bookCollection: BookCollectionSlice }
>('book');

export const bookCollectionSlice = createSelector(
  bookFeature,
  feature => feature.bookCollection
);
```

Selectors are memoized



👩‍🔧 Inline Selectors are not memoized.

ANGULAR ARCHITECTS
INSIDE KNOWLEDGE

Task

Selector

ANGULAR
ARCHITECTS
INSIDE KNOWLEDGE

Selector with Parameters

<code>

Selectors can accept parameters, too

```
export const bookByIsbn = (isbn: string) => createSelector(
  bookCollection, // reuse other selector
  books => books.find(book => book.isbn === isbn));
```

Parameter **books** is the result of the selector **bookCollection.**

Selector with Parameters

<code>

Example

```
this.book$ = this.route.params.pipe(
   switchMap(params => this.store.select(bookByIsbn(params.isbn))),
   filter((book): book is Book => !!book)
);
```

Task

Selector with Parameters

ANGULAR
ARCHITECTS
INSIDE KNOWLEDGE

Organize type information

An `ActionReducerMap` bundles the type information for your feature state.

ActionReducerMap<T>

<code>

Remember strict typing prevents you from typos producing runtime errors.

```
import { ActionReducerMap } from '@ngrx/store';

// Definition of the state shape for a whole feature
export interface BookState {
 bookCollection: BookCollectionSlice;
}

// Use interface of feature state to declare reducer functions
export const bookReducers: ActionReducerMap<BookState> = {
 bookCollection: bookCollectionReducer
};
```

ActionReducerMap<T>

<code>

Derive type definitions in your

```
// Definition of the state shape for a whole feature
export interface BookState {
 bookCollection: BookCollectionSlice;
}

// Use interface of feature state to type the feature selector
export const bookFeature = createFeatureSelector<BookState>(bookFeatureName);
```

ARCHITECTS
INSIDE KNOWLEDGE

ActionReducerMap<T>

<code>

Usage in BookModule

book.feature.ts

```
export const bookFeatureName = 'book';

export interface BookState {
  bookCollection: BookCollectionSlice;
}

export const bookReducers:
ActionReducerMap<BookState> = {
  bookCollection: bookCollectionReducer
};
```

book.module.ts

```
import { bookFeatureName, bookReducers } from '@store/book';

@NgModule({
  imports: [
    StoreModule.forFeature(
      bookFeatureName,
      bookReducers
    ),
    // ...
  ]
})
export class BookModule {}
```

Task

ActionReducerMap<T>

ANGULAR
ARCHITECTS
INSIDE KNOWLEDGE

Effects

ANGULAR
ARCHITECTS
INSIDE KNOWLEDGE

Effects are like event listeners for Actions that are able to run async functions and trigger new Actions.

Effects

- Listen for actions dispatched from @ngrx/store
- Isolate side effects from components
- Provide new sources of actions for external interactions
  - network requests
  - websocket messages
  - time-based events

Use the Effects-Module

<code>

Import the Effects via EffectsModule.forFeature in your feature component

```
import { EffectsModule } from "@ngrx/effects";

@NgModule({
 declarations: [...],
 imports: [
   ...,
   EffectsModule.forFeature([BookCollectionEffects])
 ],
...
```

How to define an Effect

<code>

Listen for the **createBookStart** action and trigger an async operation

```
@Injectable()
export class BookCollectionEffects {
 create = createEffect(() => this.actions$.pipe(
   ofType(createBookStart),
   exhaustMap(action => this.bookApi.create(action.book)),
   map(book => createBookComplete({ book }))
 ));

 constructor(
  private actions$: Actions,
  private bookApi: BookApiService) {}
}
```

How to define an Effect

<code>

Pass action payload to a service

```
@Injectable()
export class BookCollectionEffects {
 create = createEffect(() => this.actions$.pipe(
   ofType(createBookStart)
   exhaustMap(action => this.bookApi.create(action.book),
   map(book => createBookComplete({ book }))
 ));

 constructor(
  private actions$: Actions,
  private bookApi: BookApiService) {}
}
```

How to define an Effect

<code>

You can destructure the action

```
@Injectable()
export class BookCollectionEffects {
 create = createEffect(() => this.actions$.pipe(
   ofType(createBookStart)
   exhaustMap({ book }) => this.bookApi.create(book),
   map(book => createBookComplete({ book }))
 ));

 constructor(
  private actions$: Actions,
  private bookApi: BookApiService) {}
}
```

How to configure an Effect

<code>

```
@Injectable()
export class BookCollectionEffects {
 loadAll = createEffect(
   () => this.actions$.pipe(/* ... */), {
   dispatch: true|false,
   useEffectsErrorHandler: true|false
 });

 // ...
```

Task

Effects I

ANGULAR
ARCHITECTS
INSIDE KNOWLEDGE

Task

Effects II

ANGULAR
ARCHITECTS
INSIDE KNOWLEDGE

Task

Effects III

ANGULAR
ARCHITECTS
INSIDE KNOWLEDGE

Entity

Entity

<code>

EntityState<T> harmonizes state slices dealing with domain models.

```
import { EntityState } from '@ngrx/entity';

export type BookCollectionSlice = EntityState<Book>;

export interface EntityState<T> {
  ids: string[];
  entities: { [id: string]: T };
}
```

Entity

<code>

EntityAdapter for simplifying reducer functions.

```
import { createEntityAdapter } from '@ngrx/entity';

const adapter = createEntityAdapter<Book>();
```

Entity

EntityAdapter provides helper functions for reducers and selectors.

```
adapter.
        addOne
        getInitialState
        getSelectors
        removeAll
        removeMany
        removeOne
        selectId
        sortComparer
        updateMany
        updateOne
        upsertMany
        upsertOne
```

Entity

<code>

EntityAdapter ships with built-in selectors.

```
const adapter = createEntityAdapter<Book>();

export const selectors = adapter.getSelectors(<selector of slice>);
```

Entity

<code>

The target slice needs to be specified in order to get the selectors to work

```
const adapter = createEntityAdapter<Book>();

const bookCollectionSlice = createSelector(
  bookFeature, feature => feature.bookCollection
);

export const selectors = adapter.getSelectors(bookCollectionSlice);
```

Entity

<code>

Selectors being destructured and exported.

```
export const {

  selectAll,       // entities as array
  selectEntities, // entities as dictionary
  selectIds,       // ids as array
  selectTotal      // entity count as number

} = adapter.getSelectors(bookCollectionSelector);
```

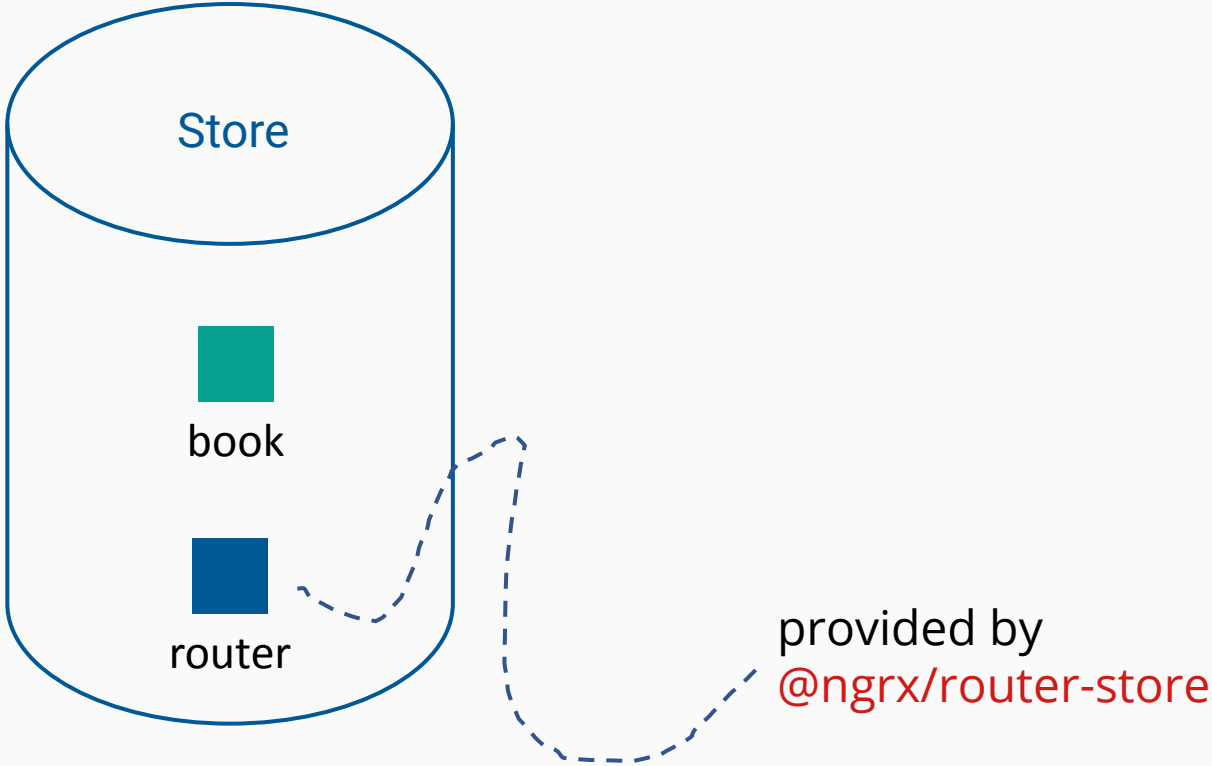
Task

Entity

ANGULAR
ARCHITECTS
INSIDE KNOWLEDGE

Router Store

Router Store



Router Store

<code>

Setup

```
import { StoreRouterConnectingModule } from '@ngrx/router-store';

@NgModule({
  imports: [
    /* ... */
    StoreRouterConnectingModule.forRoot(),
  ]
})
export class AppModule {}
```

Router Store

<code>

Router Store brings its own **reducer**.

```
import {/* ... */, routerReducer } from '@ngrx/router-store';

@NgModule({
  imports: [
    /* ... */
   StoreModule.forRoot(
      { router: routerReducer },
    )
    StoreRouterConnectingModule.forRoot(),
  ]
})
export class AppModule {}
```

Router Store

<code>

Router Store brings its own **selectors**.

```
import { getSelectors, RouterReducerState } from '@ngrx/router-store';

export const selectRouter = createFeatureSelector<RouterReducerState>('router');

export const {
  selectCurrentRoute,   // select the current route
  selectFragment,       // select the current route fragment
  selectQueryParams,    // select the current route query params
  selectQueryParam,     // factory function to select a query param
  selectRouteParams,    // select the current route params
  selectRouteParam,     // factory function to select a route param
  selectRouteData,      // select the current route data
  selectUrl,            // select the current url
} = getSelectors(selectRouter);
```

Router Store

<code>

Router Store brings some **selectors with parameters**.

```
import { getSelectors, RouterReducerState } from '@ngrx/router-store';

export const selectRouter = createFeatureSelector<RouterReducerState>('router');

export const {
  selectCurrentRoute,   // select the current route
  selectFragment,       // select the current route fragment
  selectQueryParams,    // select the current route query params
  selectQueryParam,     // factory function to select a query param
  selectRouteParams,    // select the current route params
  selectRouteParam,     // factory function to select a route param
  selectRouteData,      // select the current route data
  selectUrl,            // select the current url
} = getSelectors(selectRouter);
```

Router Store

<code>

Route selectors can be reused in other selectors.

```
export const bookCollection = createSelector(
  bookFeature, slice => slice.bookCollection.entities
);

export const bookByIsbn = createSelector(
  selectRouteParam('isbn'),
  bookCollection, (isbn, books) => books.find(book => book.isbn === isbn)
);
```

Router Store

<code>

Route selectors can be reused in other selectors.

```
export const bookCollection = createSelector(
  bookFeature, slice => slice.bookCollection.entities
);

export const bookByIsbn = createSelector(
  selectRouteParam('isbn'),
  bookCollection, (isbn, books) => books.find(book => book.isbn === isbn)
);
```

Router Store

<code>

Route selectors can be reused in other selectors.

```
export const bookCollection = createSelector(
  bookFeature, slice => slice.bookCollection.entities
);

export const bookByIsbn = createSelector(
  selectRouteParam('isbn'),
  bookCollection, (isbn, books) => books.find(book => book.isbn === isbn)
);
```

Router Store

<code>

Route selectors can be reused in other selectors.

```
export const bookCollection = createSelector(
  bookFeature, slice => slice.bookCollection.entities
);

export const bookByIsbn = createSelector(
  selectRouteParam('isbn'),
  bookCollection, (isbn, books) => books.find(book => book.isbn === isbn)
);
```

Task

Router Store

ANGULAR
ARCHITECTS
INSIDE KNOWLEDGE

Testing

ANGULAR
ARCHITECTS
INSIDE KNOWLEDGE

The migration to NgRx might brake some of our tests we have written before.

Let‘s fix them

ANGULAR
ARCHITECTS
INSIDE KNOWLEDGE

Component with Store

<code>

Mocking the store

```
import { MockStore, provideMockStore } from '@ngrx/store/testing';

let store: MockStore;

beforeEach(() => {
    TestBed.configureTestingModule({
      declarations: [BookListComponent],
      providers: [provideMockStore()]
    });
});

store = TestBed.inject(MockStore);
```

Component with Store

<code>

Mocking a selector

```
import { bookCollection } from '@store/book';

store.overrideSelector(bookCollection as any, [new BookNa()]);
```

Task

Test | Mock Selector

ANGULAR
ARCHITECTS
INSIDE KNOWLEDGE

Effects

<code>

Setup Environment for Effects

```
TestBed.configureTestingModule({
  imports: [
    RouterTestingModule,                                    // If routing is used
    EffectsModule.forRoot([BookCollectionEffects]),         // Setup Effects and services
    StoreModule.forRoot({}),                                // Setup store
    StoreModule.forFeature(bookFeatureName, bookReducers) // Setup feature for store
  ],
  providers: [{
    provide: BookApiService,
    useFactory: () => bookApiMock
  }] // Setup Service dependencies
});
```

Effects

<code>

Dispatching an action and check the change in the store.

```
describe('When a book was created successfully', () => {
    it('caches the book locally', done => {
      // ...

      store.dispatch(createBookStart({ book }));

      store.select(bookCollection).subscribe(books => {
        expect(books).toContain(book);
        done();
      });
  });
});
```

Task

Test | Action - Effect -
Reducer - Selector

ANGULAR
ARCHITECTS
INSIDE KNOWLEDGE

Reducer

<code>

Since reducers are pure it is a simple unit test

```
const initialState: EntityState<Book> = { entities: {}, ids: [] };

 const action = loadBooksSuccess({book});

 const result = reducer(initialState, action);

 expect(result).toEqual({...initialState, books});
```

Task

Test | Reducer

ANGULAR
ARCHITECTS
INSIDE KNOWLEDGE