Upgrading State Management from RxJS to NgRx

Ervis Trupja

Software Engineer, Author

@ervis_trupja | www.dotnethow.net

What Is NgRx and Why Use It?

NgRx provides structured state management for Angular by using RxJS to handle all application data as a single, predictable stream.

The Problem with Simple State Management

State changes are scattered throughout the application

Difficult to track what caused a state change

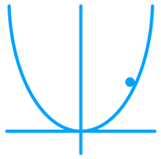
No clear separation between data fetching, state updates, side effects

Hard to implement features like undo/redo

Challenging to debug complex state interactions

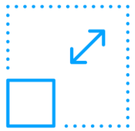
Poor performance with large state objects

Why NgRx?



Predictability: State changes follow a strict unidirectional flow

Debugging: Tools for tracking and debugging state changes



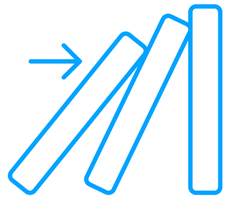
Scalability: Consistent patterns for large apps

Testing: Simplifies testing individual components

Performance: Optimized for Angular’s change detection

NgRx Key Concepts/Components

Actions



Effects

Reducers

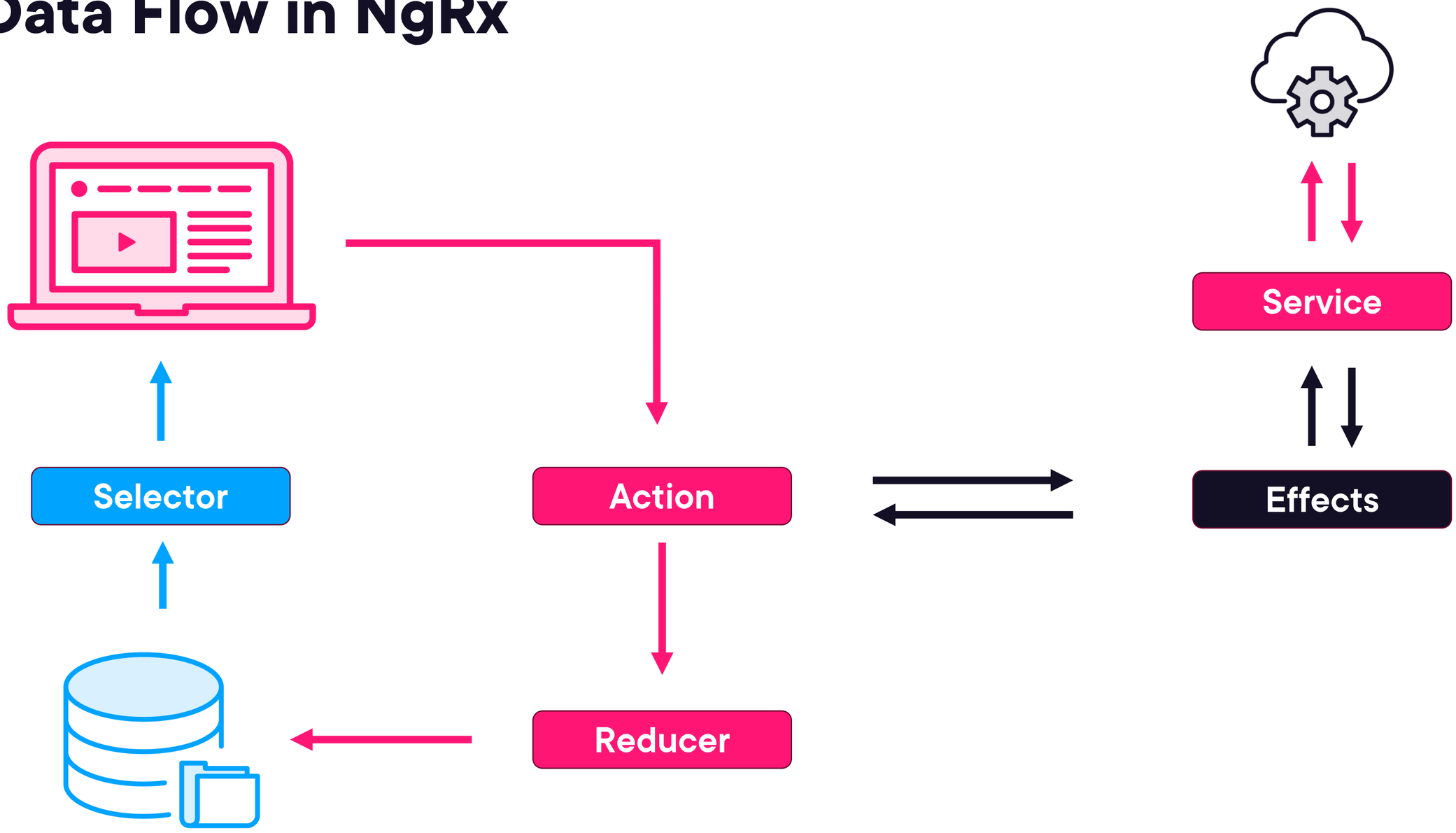
Store

Selectors

Data Flow in NgRx

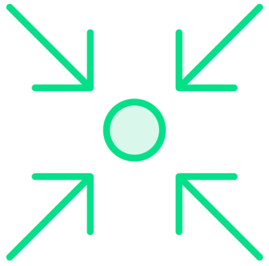
Front-end app

Data Flow in NgRx



The NgRx Solution

NgRx introduces a structured approach to state management based on three core principles:

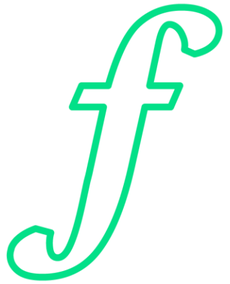


Single source of truth

All application state is stored in one place

State is read-only

State can only be changed through actions



Reducers are functions

Take the current state and an action, returning a new state

What Will We Change?

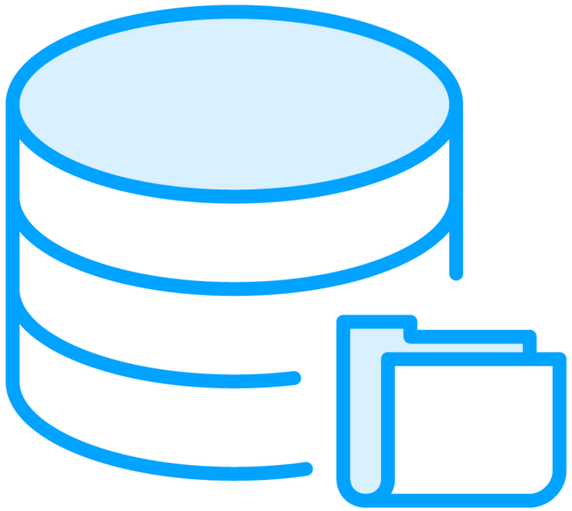
RxJs	vs.	NgRx
Used TaskService with BehaviourSubject		Implemented NgRx store with actions, reducers, selectors, and effects
State changes were handled directly in the service		State changes go through a strict action → reducer → state flow
Components subscribed to service observables		Components dispatch actions and use selectors to access state
No clear separation between data fetching and state management		Clear separation of concerns with dedicated files for each NgRx concept

Setting Up NgRx and Store

NgRx Store

The NgRx Store is a centralized container in Angular that holds your application's entire state. A single instance of the Store is created and shared across your whole application, ensuring a consistent and single source of truth for all data.

NgRx Store

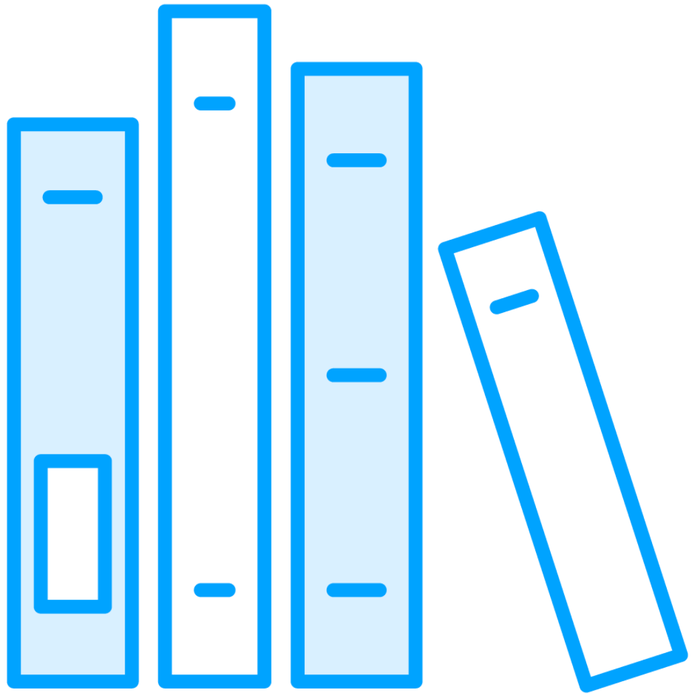


A centralized container for your app's entire state

You get data out of the store using Selectors

You change the state in the store by dispatching Actions

Libraries



@ngrx/store

@ngrx/effects

@ngrx/store-devtools

Demo: Setting up NgRx and Store

Creating Actions

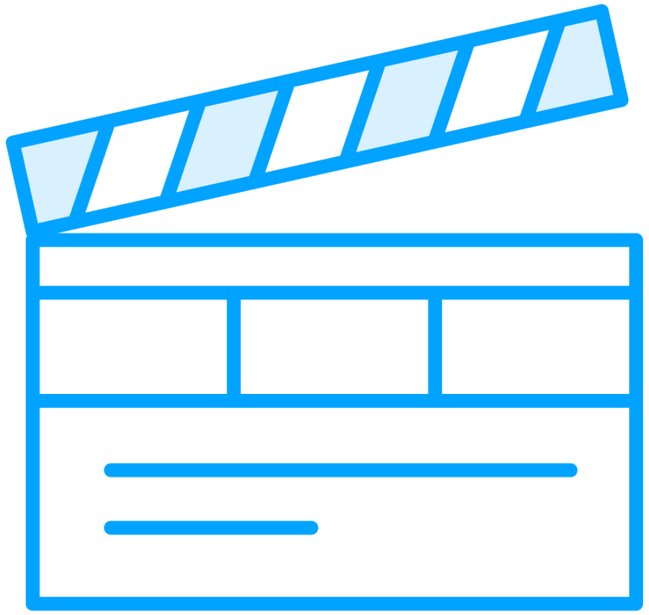
Actions and Reducers

Actions and Reducers are core concepts used to manage and update the application's state in a predictable and scalable way.

Actions are plain objects that describe what happened.

Reducers are functions that update the state based on those Actions

Action



Describes a specific app event

The only source of information for the store

Has a unique type and an optional payload

Actions

Actions are payloads of information that describe events or changes in the application

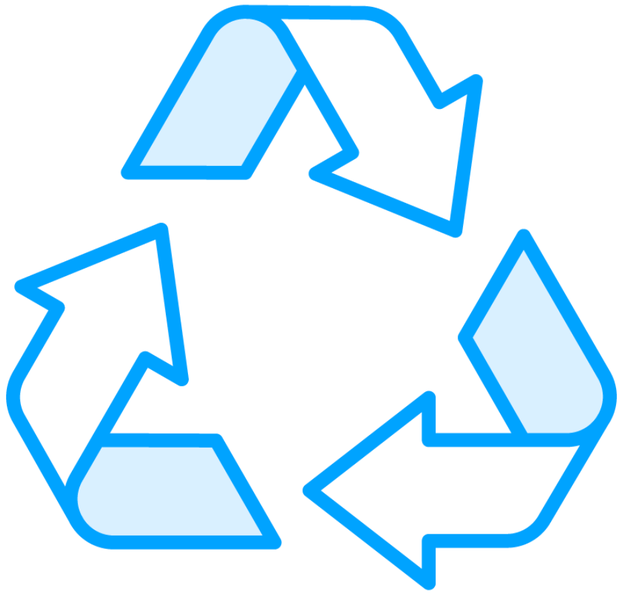
tasks.actions.ts

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

export const loadTasks = createAction(
  '[Tasks] Load Tasks'
);

export const addTask = createAction(
   '[Tasks] Create New Task'
   props<{ task: Task }>()
);
```

Reducer



Defines how the state changes in response to an action

Receives the current state and an action as arguments

Returns a new, completely immutable state object

Reducers

Reducers are pure functions that take the current state and an action as input and return a new state

task.reducer.ts

```
import { createReducer, on } from '@ngrx/store';
import { loadTasks, addTask } from ‘./task.actions’;

export const userReducer = createReducer(
  initialState,
  on(loadTasks, (state) => ({ ...state, loading: true })),
  on(addTask, (state, { task }) => ({
    ...state,
    tasks,
    loading: false,
  }))
);
```

Demo: Creating Actions

Setting up application actions

Demo: Creating Reducers

Setting up application reducers

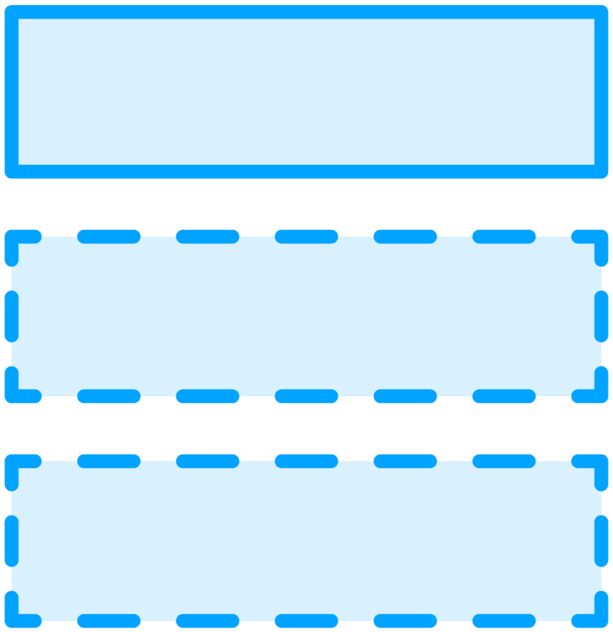
Demo: Mapping Actions with Reducers

Selecting State with Selectors

Selectors

A selector is a function used to read specific data from an application's state. It can also transform or combine this data.

Selector



Selects specific slices of data from the store's state

Can compute new, derived data from state slices

Caches (or "memorizes") results for optimal performance

Selector Types

Basic (state slice selectors)

Derived (computed selectors)

Parameterized (dynamic selectors)

Other:

- Feature selectors
- Combining selectors

Demo: Selecting State with Selectors

Creating application selectors

Replace Service State with NgRx

Replacing services with store, actions, reducers, and selectors

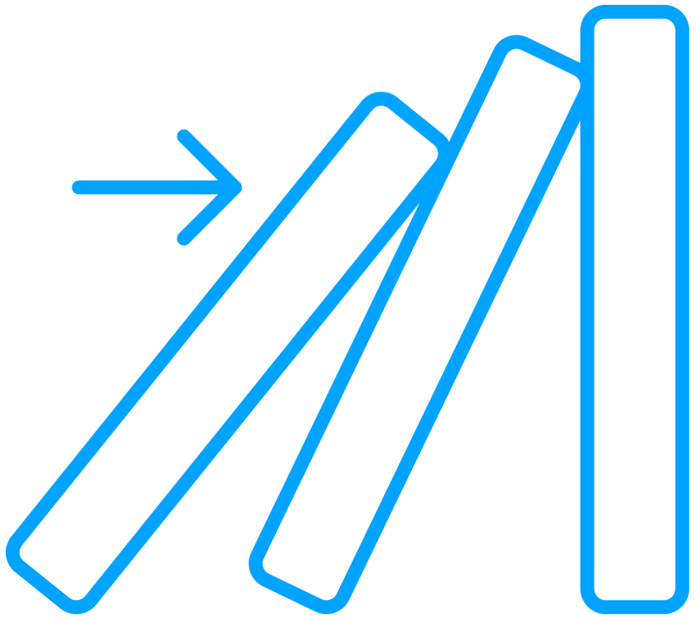
Handling Side Effects with Effects

Side Effects

Side effects are operations in an application that interact with the outside world, such as making API calls, accessing browser storage, or triggering browser events.

Unlike pure functions (like reducers), side effects depend on external factors and their outcomes can be unpredictable, often involving waiting for external systems.

Effect

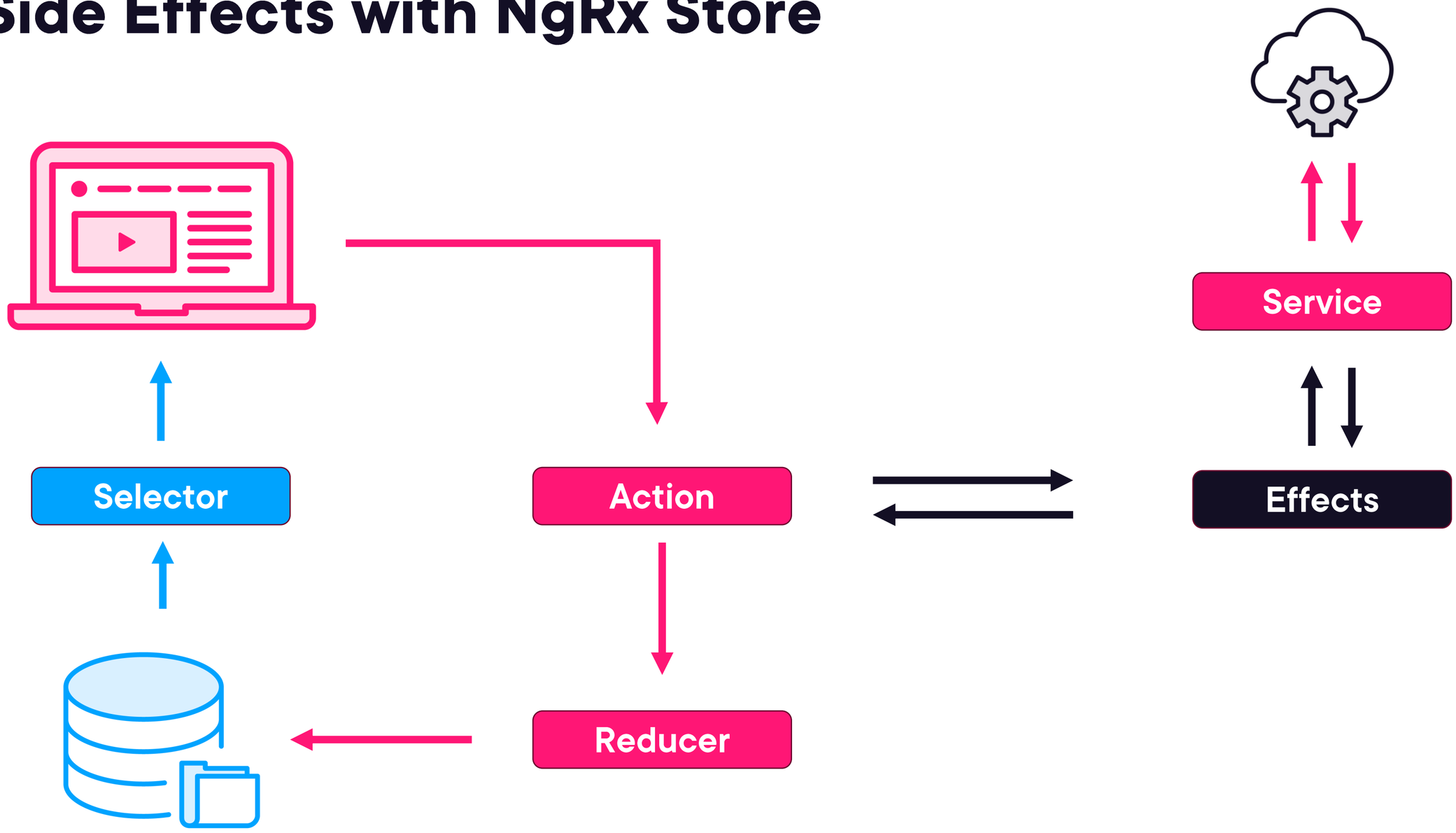


Manages side effects like API calls, timers, or web sockets

Listens for specific actions to know when to trigger its work.

Often dispatches new actions (e.g., success or failure) when its task is done.

Side Effects with NgRx Store



Demo: Handling Side Effects with Effects

Adding a new task with effects

Debugging with Redux DevTools

Redux DevTools

Redux DevTools is a browser extension or standalone tool designed to debug and monitor state management in applications using Redux or Redux-based libraries like NgRx.

Redux DevTools allows you to visualize and interact with the NgRx store's state, actions, and history, making it easier to debug issues like incorrect state updates or unexpected action flows.

How It Works?

Setup and integration

Action history tracking

State inspection

Time-travel debugging

Error detection

Demo: Debugging with Redux DevTools

How Redux DevTools works for debugging