



Welcome to **DevGeekWeek**

We will start in a few minutes



C:\ngrx new




Redux with Signal Stores

Hi, I'm...



Kobi Hari

- Freelancer
- Developer, Instructor and Consultant
- Angular, Async development, .NET core

 hari@applicolors.com

 <https://github.com/kobi2294>

 <https://www.youtube.com/@kobihari-applicolors1464>



Angular + NgRx



DevGeekWeek 2024
INSPIRED BY INNOVATION



JOHN BRYCE
תלמדו הייטק. זה עובד!
a matrix company

We have a GitHub Repository!

[kobi-hari-courses/2406-geek-week-ngrx](https://github.com/kobi-hari-courses/2406-geek-week-ngrx)



Commits		
#	Link	Description
1	Initial application	Created an empty application using the angular CLI: ng new redux-pop-quiz
2	Added Material Design	Added angular material using the cli: ng add @angular/material
3	Custom Theme	Defined the theme palettes in partial _common.scss and generated the material styles in styles.scss
4	Tested theme	Used some angular material components like icon and button to test the new theme
5	Grid, Flex layout	Created layout where main content is either grid or flex for easier layout management
6	CSS Variables	Added CSS variables to the theme

Session Summary

Commits

Application Setup

#	Link	Description
1	Initial application	Created an empty application using the angular CLI: ng new redux-pop-quiz
2	Added Material Design	Added angular material using the cli: ng add @angular/material
3	Custom Theme	Defined the theme palettes in partial _common.scss and generated the material styles in styles.scss
4	Tested theme	Used some angular material components like icon and button to test the new theme
5	Grid, Flex layout	Created layout where main content is either grid or flex for easier layout management
6	CSS Variables	Added CSS variables to the theme

Project Code with Commits Summary

Building the UI

#	Link	Description
7	Toolbar	Added material toolbar component

 [presentations](#)

 [projects](#)

 [LICENSE](#)

 [README.md](#)

This presentation and other Useful Links



Angular + NgRx



Our Agenda



The Redux Pattern



NgRx Store – Selectors, Actions, Reducers



NgRx Effects – Reactivity On Steroids

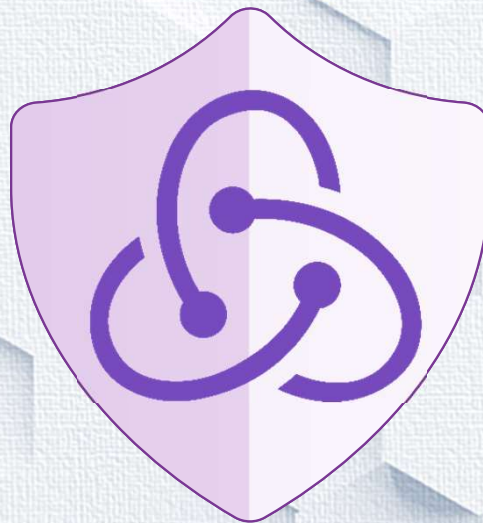


NgRx Signals Store – The new approach





Angular + NgRx



The “Redux” Pattern





Angular + NgRx



The Redux Pattern



Single Point of Truth



Immutability

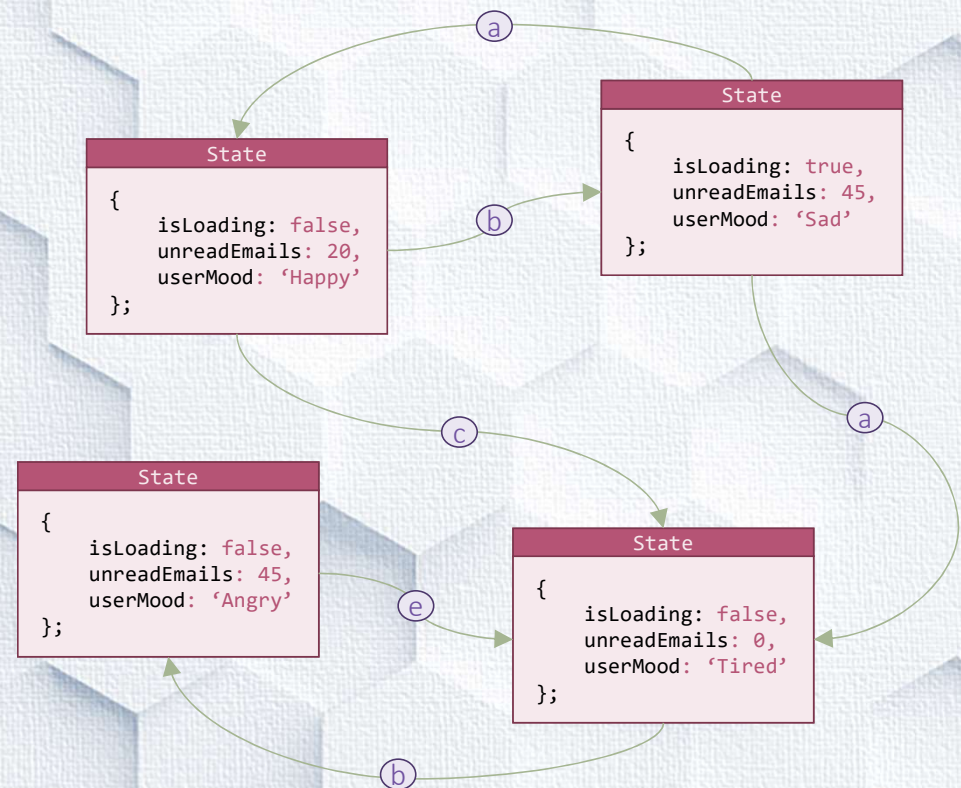


Pure functions - Reducers



What is a “State”

- **State** is an object holding all the changeable data in the application
- Any change to data means that you move from one **state** to another
- **State** is replaced as a result of an **Action**





Angular + NgRx



NgRx Store

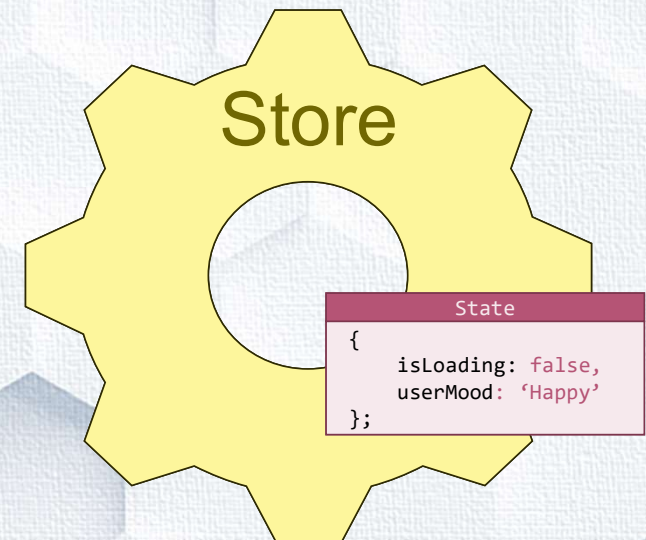
Selectors, Actions and Reducers





The Store

- The store is a service that holds the state
- The structure of the state is private (!)
- Reading or Updating the state is done indirectly using **Selectors** and **Actions**

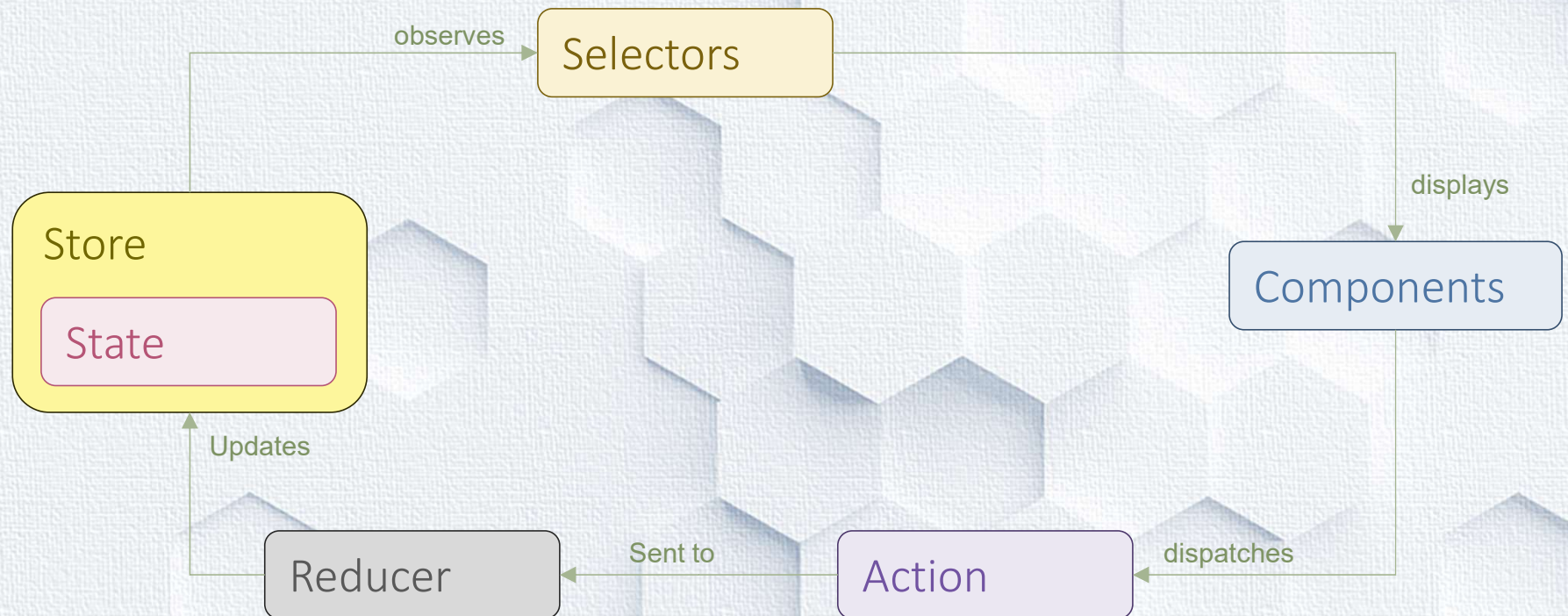




Angular + NgRx



The Circle of Life





Angular + NgRx



Demo - Setting up the Store

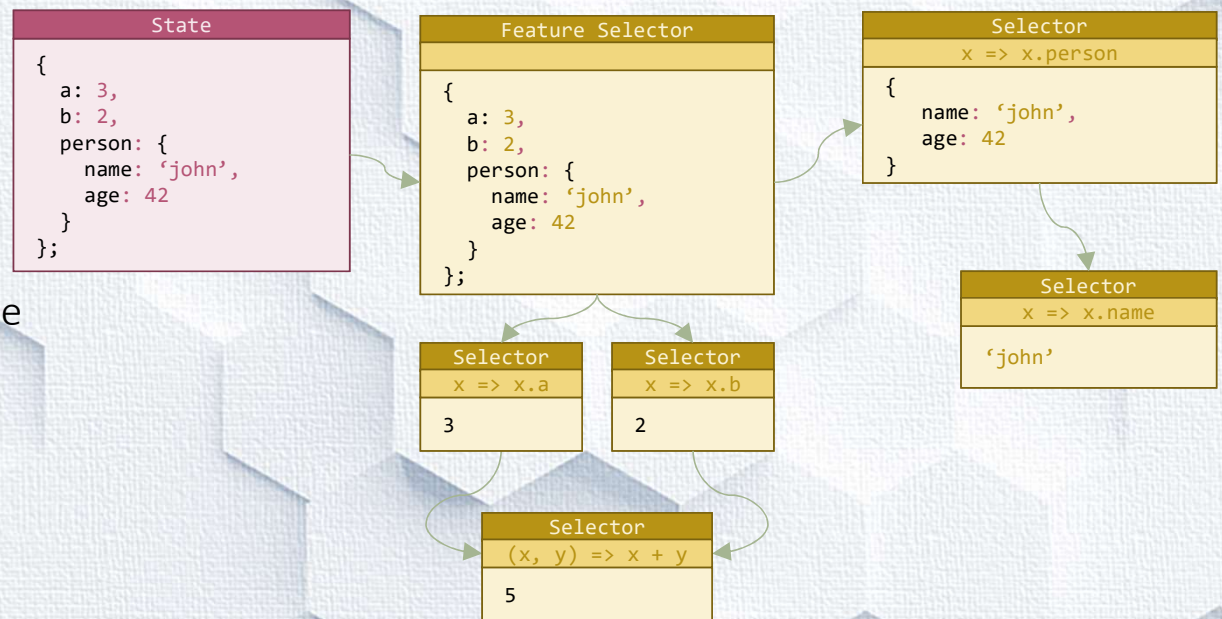


- ✓ Installing the packages
- ✓ Creating State and Store
- ✓ Instrumenting The dev-tools
- ✓ Creating features
- ✓ Providing the feature





Selectors



- Selectors select a piece of the state
- A selector is based on:
 - Parent selector (or many)
 - Projection function
- They can also select “calculated expression” of the state



Selecting from the store

- Selectors are used to create observables
- These observables are memoized and optimized

 `store.select(`

Selector
<code>x => x.person.name</code>

`)`



`Observable<string>`





Angular + NgRx

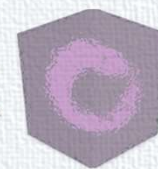


Demo – Using Selectors



- ✓ Atomic Selectors
- ✓ Adding Extra Selectors
- ✓ Deriving Selectors from multiple sources
- ✓ Using selectors in Components





Actions

- Action is a simple JSON
- It must contain a “**type**” property
- It may contain additional properties (called the “**payload**”)
- NgRx provides “Action Creators”
- Actions describe **events**.
- By convention the **type** should look like this:
‘[SOURCE] name’

Action

```
{  
  type: '[USER] set loading',  
  value: true  
};
```

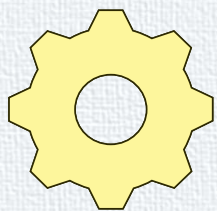



Angular + NgRx



Dispatching Actions

- Use the **Store** to dispatch actions



store.dispatch

```
Action
{
  type: '[USER] set loading',
  value: true
};
```





Action Groups

- NgRx 16 now has a new concept for actions: **Action Group**
- **Action Groups** are used to group together actions from the same source
- Angular uses Typescript Templates to pull some neat compiler tricks



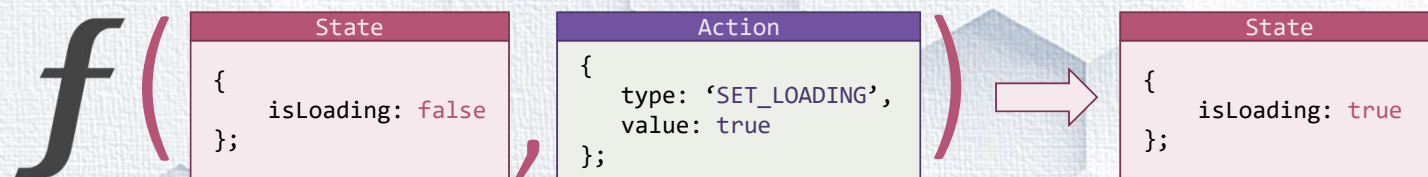


Angular + NgRx



Reducers

- Reducer is a “Pure Function”
- It calculates new state from old state, and an action





Angular + NgRx



Demo – Using Action Groups and Reducers

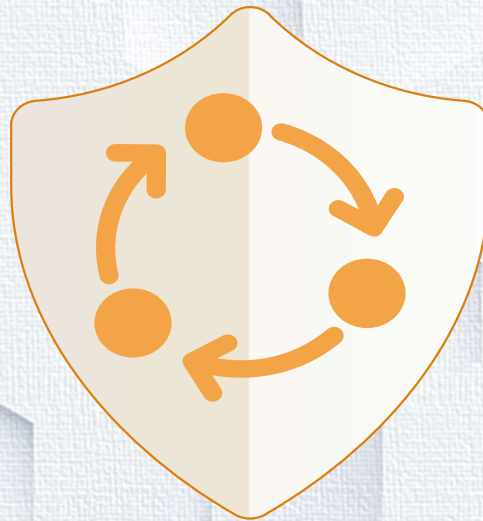


- ✓ Creating Action groups
- ✓ Using the `props` functions to define payload
- ✓ Dispatching actions
- ✓ Implementing the Reducers





Angular + NgRx



NgRx Effects

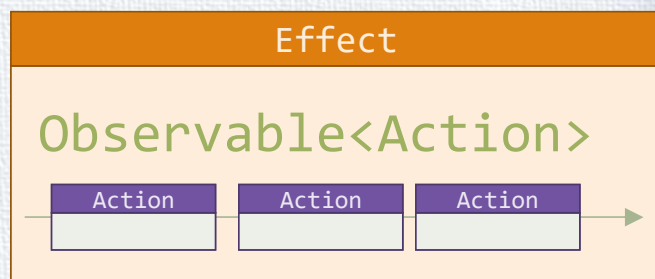
Reactivity On Steroids

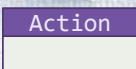




What are “Effects”

- **Effects** are **Observables**
 - That the store subscribes to
 - That yield actions
 - That get dispatched automatically



.subscribe( =>




.dispatch())

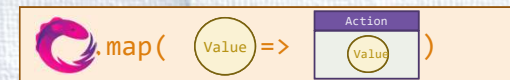
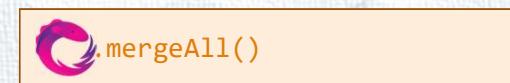
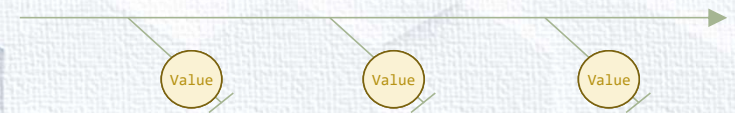
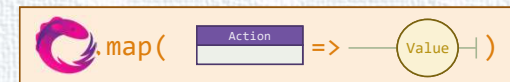
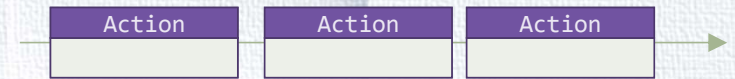


Angular + NgRx



Fun facts about **Effects**

- They are usually created from other observables of **actions**
- They are built using  **RxJS operators**
- They are often mapped to asynchronous **values**
- They often cause.... **Side effects**
(AH HA!!!!)

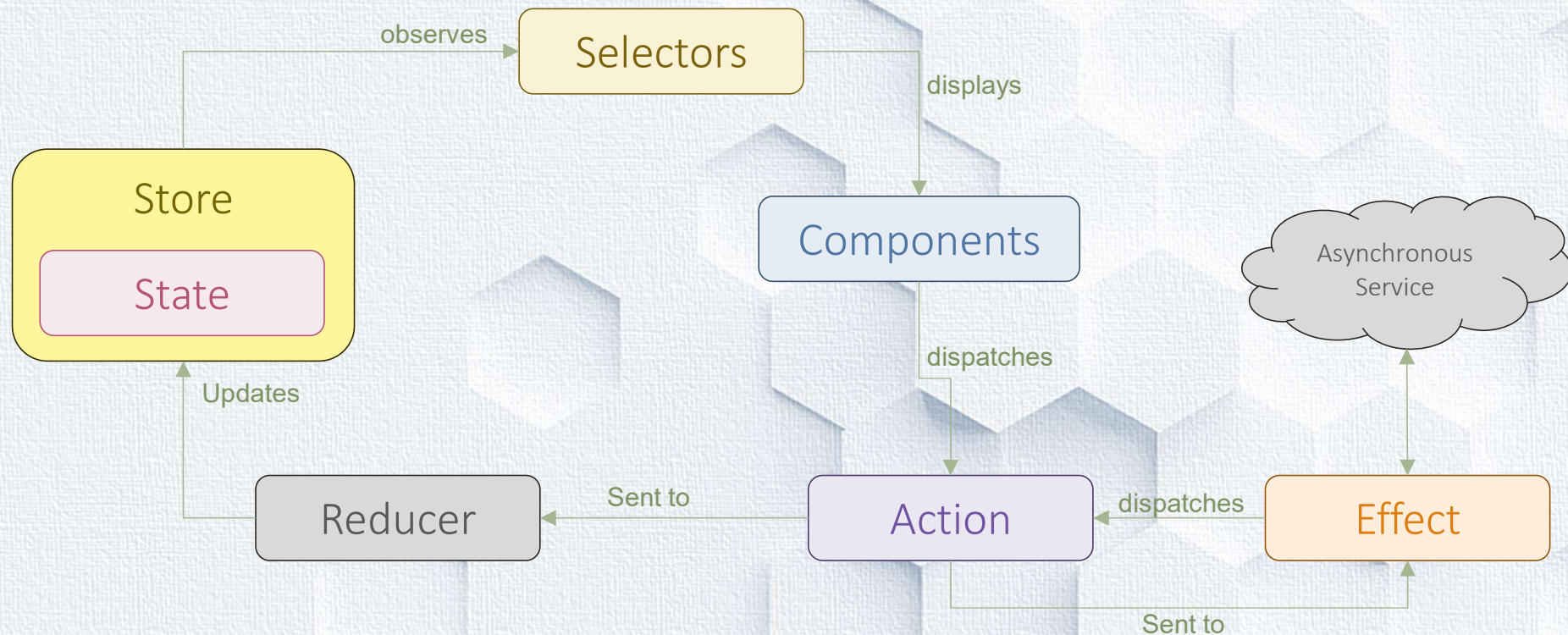




Angular + NgRx



The Circle of Life – with **effects**





Actions drive the app

- **Actions** are not just for the store
- In the “Reactive” thinking, everything that can happen in the application is an **action**.
- **Effects** handle **Actions** by performing operations that are translated into other actions
- **Reducers** handle **Actions** by changing the current state of the application





Angular + NgRx



Demo – Using Effects



- ✓ Installing the @ngrx/effects package
- ✓ Writing Effects as functions
- ✓ Reacting to Actions or State
- ✓ Invoking and Flattening async methods
- ✓ Handling Errors
- ✓ Registering the Effects





Angular + NgRx



NgRx Signal Store

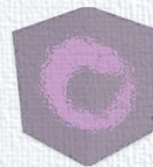
Signal based store as a service





Angular +

NgRx



Signal Store

- Signal Store is a service
- It is provided by a component
- It is usually tied to the life-cycle of the component
- It is like a “baby” store
 - It has “**selectors-like**” signals
 - It has “**reducer-like**” methods
 - It has “**effects-like**” methods
 - It has no **actions** (!!!)
 - It has “**custom features**” (wait and see!!!)
 - It is completely based on functional programming
 - It is based on Signals instead of Observables



Signal State – The baby store

- Define a state: `T`
- Call the `signalState<T>` function
- What you get is the signal of the entire state.
 - But it also has sub property for each sub property of the state.
 - Each such property is a signal of that property
- In the example:
 - `this.state()` returns the entire object (as a signal)
 - `this.state.x()` returns 50 (as a signal)
 - `this.state.y.y1()` returns 10 (as a signal)
- Signal State is a nice utility around signals

```
@Component({selector: 'app-my', template: 'x = {{state.x()}}'})
0 references
export class MyComponent {
  1 reference
  state = signalState({
    x: 50,
    y: {
      y1: 10, y2: 20
    }
  })

  0 references
  changeX() {
    patchState(this.state, state => ({x: state.x + 1}));
  }
}
```




Angular + NgRx



Signal Store

- Signal stores are full blown services
- They have properties, methods.
- You can use them as real stores
- But... they have no class. They are fully functional.
 - You create one by calling the `signalStore` function
 - You enhance it by calling `withXXX` functions as parameters. Each such function builds more functionality into the function.
 - Yes, that was absolutely a valid English sentence in functional programming, get used to it 😊
 - The `signalStore` function, returns a newly created type.
 - You can then use it as injection token.

1 reference

```
export const initialState: QuizState = {  
  questions: QUESTIONS,  
  answers: []  
};
```

3 references

```
export const QuizStore = signalStore(  
  withState(initialState)  
);
```




Consuming the store

- The `signalStore` function, returns a newly created type.
- You can then use it as injection token.
 - Make sure to provide it at the proper level
 - Then just inject the type.
- Just like `signalState`
 - it exposes a set of signals you can bind to.
 - You can call `patchState` to modify it.
 - But don't, at least not like this...

```
@Component({
  selector: 'app-view',
  template: `Number of questions:
  | | | | | {{store.questions().length}}`,
  providers: [QuizStore]
})
0 references
export class ViewComponent {
  0 references
  store = inject(QuizStore);
}
```





Angular + NgRx



Demo – Using Signal Store



- ✓ Installing the `@ngrx/signals` package
- ✓ Creating a signal store service
- ✓ Initializing the state
- ✓ Providing it in the component
- ✓ Injecting it into the component
- ✓ Consuming the state





Computed Signals replace Selectors

- The angular signals allow you to derive **computed** signals from them.
- In signal store, this is how you replace selectors.
- Use the **withComputed** method to define a set of computed signals
- The signals may receive properties already defined as parameters and use them to compute the value

1 reference

```
export const initState = {x: 10, y: 20};
```

0 references

```
export const XyStore = signalStore(  
  withState(initState),  
  withComputed(({x, y}) => ({  
    sum: computed(() => x() + y()),  
    diff: computed(() => x() - y())  
  })))  
)
```





Angular + NgRx



Demo – Signal Store **withComputed**



- ✓ Creating Computed Signals
- ✓ Deriving signals from other signals
- ✓ Binding to computed values





Angular + NgRx



Setting the state

- You can use **patchState** to modify the current state.
 - **patchState** is a function, so it is called independently
 - The first parameter you pass is the **store**
 - Then you have 2 options
 - Pass a partial new state
 - Pass a function that takes the current state and returns a partial new state
- It's all very functional...

```
export class ViewComponent {  
  1 reference  
  store = inject(XyStore);  
  
  0 references  
  incX() {  
    patchState(this.store, state => ({x: state.x + 1}));  
  }  
}
```



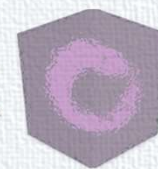

Creating an Updater method

- While the previous method is possible it is **not recommended**
 - We like our updates to be encapsulated in the store, to make sure only valid states are created
- Updater is replacing “**action** + **reducer**”
- The **updater** method creates a function
- Calling the function is like dispatching an **action** that updates the state
- We create method using the **withMethods** function
 - It takes a function that takes the store
 - The function returns an object full of methods
 - These are added to the store

```
export const XyStore = signalStore(  
  withState(initState),  
  withComputed(({x, y}) => ({  
    sum: computed(() => x() + y()),  
    diff: computed(() => x() - y())  
  })),  
  withMethods(store => ({  
    incX() { patchState(store, state => {  
      {x: state.x + 1}}),  
    incY() { patchState(store, state => {  
      {y: state.y + 1}}),  
    reset() { patchState(store, initState)}  
  })))  
)
```




Angular + NgRx



Consuming an Updater

- Updater methods are called like any class method
- Easy...
- Of course, you have this nice feature where all the signals get automatically updated

```
onIncX() {  
  this.store.incX();  
}
```




Angular + NgRx



Demo – Signals store **Updater** methods



- ✓ Using patchState
- ✓ Passing callbacks to patchState
- ✓ Defining an Updater
- ✓ Using the Updater





Creating RxMethods

- Rx Methods are replacement for effects.
- They are asynchronous methods that are triggered like observables.
- Rx methods can be called in various ways.
 - Imperatively – like any other method
 - Reactively by passing an observable, or signal
- They are implemented like an observable that gets a “next” whenever the method is called.

```
withMethods((store, userService = inject(UserService)) => ({
  loadFromServer: rxMethod<string>(  
    pipe(  
      tap(() => patchState(store, {isLoading: true}))  
      exhaustMap(str => userService.getNumber(str)).pipe(  
        tapResponse(  
          next: num => patchState(store, {x: num}),  
          error: console.error,  
          finalize: () => patchState(store, {isLoading: false})  
        )  
      )  
    )  
  )  
}))
```





Consuming rxMethods

- An **rxMethod** is a function just like normal method
- Calling it is like dispatching an action that is handled by an effect
- You can call it imperatively or declaratively
- It may receive
 - Value
 - Observable
 - Signal.

```
myStr = signal('Hi');  
subj$ = new Subject<string>();  
  
ngOnInit() {  
  this.store.loadFromServer(this.myStr);  
  this.store.loadFromServer(this.subj$);  
}  
  
loadAbc() {  
  this.store.loadFromServer('abc');  
}
```





Angular + NgRx



Demo – Signal store **rxMethod**



- ✓ Creating rxMethod
- ✓ Using tapResponse
- ✓ Consuming rxMethods imperatively
- ✓ Consuming rxMethods declaratively





Signal Store Hooks

- You can hook to signal store events just like you can with components and services

```
withHooks({  
  onInit(store) {  
    store.loadFromServer('initial')  
  },  
  onDestroy(store) {  
    console.log('Good bye');  
  }  
})
```





Signal Store Custom Features

- Probably the best feature of signal store is the fact that you can add your own “withXXX” features.

```
export type CallState = 'init' | 'loading' | 'loaded' | { error: string };

export function withCallState() {
  return signalStoreFeature(
    withState<{ callState: CallState }>({ callState: 'init' }),
    withComputed(({ callState }) => ({
      loading: computed(() => callState() === 'loading'),
      loaded: computed(() => callState() === 'loaded'),
      error: computed(() => {
        const state = callState();
        return typeof state === 'object' ? state.error : null
      })
    })),
  );
}
```

```
export const XyStore = signalStore(
  withState(initState),
  withComputed(({x, y}) => ({
    sum: computed(() => x() + y()),
    diff: computed(() => x() - y())
  })),
  withCallState()
);
```





Demo – Signal store `customFeatures`



- ✓ Creating a simple custom feature
- ✓ Realizing that we do not have the dev-tools
- ✓ Understanding how the dev-tools work
- ✓ Let's get crazy
- ✓ Creating withDevTools - a custom feature that connects the store to redux dev-tools



Thanks :)
See you next year