YOU DON'T KNOW MOBX STATE TREE

Berlin | 20-21 November 2018

Max Gallo | @_maxgallo

{codemotion}

HI 👋
I'M MAX GALLO

About me: 🍝 💻 🇬🇧 🎶 🏍 📷 ✈️ ✍️

Principal Engineer @ DAZN

twitter: @_maxgallo
more: maxgallo.io

AGENDA

» Part One: MobX

» Part Two: MobX State Tree

» Part Three: Designing a Reactive Project

PART ONE

MOBX

MOBX

» State Management Library

» Helps decoupling View from Business Logic

» Transparent Reactive Programming

» Flexible / Unopinionated

MOBX OBSERVABLES & REACTIONS

```
import { observable, autorun} from 'mobx';

const album = observable({
    title: 'Californication',
    playCount: 0,
});

autorun(() => console.log(`New play count: ${album.playCount}`))
// New play count: 0

album.playCount = 1; // New play count: 1

album.playCount = 24; // New play count: 24
```

MOBX COMPUTED VALUES

```
import { observable, autorun, computed} from 'mobx';

const album = observable({
    title: 'Californication',
    playCount: 0,
});

const all = computed(() => album.title + album.playCount);

autorun(() => console.log(all))
// Californication0

album.playCount = 1;          // Californication1
album.title = 'OkComputer'; // OkComputer1
album.playCount = 24;         // OkComputer24
```

MOBX RECAP

Observable state

Mutable Application State

Computed Values

Automatically derived values, lazily evaluated

Reactions

Side effects like autorun or updating a React component

MOBX ❤️ REACT

```
const album = observable({
    title: 'Californication',
    playCount: 0,
});

@observable
class MyComponent extends React.Component {
    render() {
        return `Californication album.playCount`;
    }
}

album.playCount++     // ----> React Render
album.playCount = 9   // ----> React Render
```

PART TWO
MOBX STATE TREE

MOBX STATE TREE

» Powered by MobX

» Runtime typed Application State

» Opinionated & ready to use

» Relies on the concept of Trees (Stores)

WHAT'S A TREE/STORE ?

```
import { types } from 'mobx-state-tree';

const AlbumStore = types
    .model('Album', {                   // mobx observable
        title: types.string,
        rating: types.integer,
    })
    .views(self => ({
        get isGood() {                   // mobx computed
            return self.rating >= 7
        }
    }));

const okComputer = AlbumStore.create(
    { title: 'Ok Computer', rating: 8}
);

console.log(okComputer.isGood); // true
```

MOBX STATE TREE STORES

```
const AlbumStore = types.model('Album',...)


const MusicLibraryStore = types
    .model('MusicLibrary', {
        albums: types.array(AlbumStore), // -------> Model
    })
    .views(self => ({
        get goodAlbums() { // ---------------------> View
            return self.albums.filter(x => x.isGood)
        }
    }))
    .actions(self => ({
        addAlbum(album) { // ----------------------> Action
          self.albums.push(album);
        },
    }));

const musicLibrary = MusicLibraryStore.create(
    { albums: [ { title: 'Rumours', rating: 9} ] }
);

musicLibrary.addAlbum({ title: 'Harvest', rating: 8 });
musicLibrary.addAlbum({ title: 'Ten', rating: 9 });
```

Model

» Mutable observable state

» Runtime type information

» Could contain other trees

Views

MobX computed values

Actions

The only way to update the model

MOBX STATE TREE

HOW TO CONNECT THE STORES WITH THE VIEW ?

```
/** ------------- App.js -------------------- **/

import { Provider }    from 'mobx-react'
import App             from './App.js'
import ShopStore       from './Shop.store.js'
import NavigationStore from './Navigation.store.js'

const shopStore = ShopStore.create()

ReactDOM.render(
    <Provider
        shop={shopStore}
        navigation={navigationStore}
    >
        <App />
    </Provider>,
    document.getElementById('root')
)
```

```
/** ---------- CheckoutView.js ---------- **/

import React                from 'react'
import { inject, observer } from 'mobx-react'

@inject('shop') @observer
class CheckoutView extends React.Component {
    render() {
        const { shop } = this.props;
        return (
            Amount:{ shop.checkountAmount }
        );
    }
}
```

MOBX STATE TREE STORES

DEEP DIVE

» Mutable and Immutable (Snapshots, Time Travelling)

» Composition

» Lifecycle Methods

» Dependency Injection

```
import { getEnv, types } from 'mobx-state-tree';

const MusicLibraryStore = types
    .model('MusicLibrary', {
        albums: types.array(types.string),
    })
    .actions(self => ({
        downloadAlbums() {
            getEnv(self).getAlbumsFromBackend()
                .then(albums => self.albums = albums);
        }
    }));

const getAlbumsFromBackend = function() {
    /** Fetching albums from Backend **/
}

const musicLibrary = MusicLibraryStore.create(
    { albums: ['Who\'s Next'] },  // Initial state
    { getAlbumsFromBackend }      // Environment
);
```

MOBX STATE TREE STORES

DEPENDENCY INJECTION

» Inject anything

» Environment is shared per tree

» Useful for testing

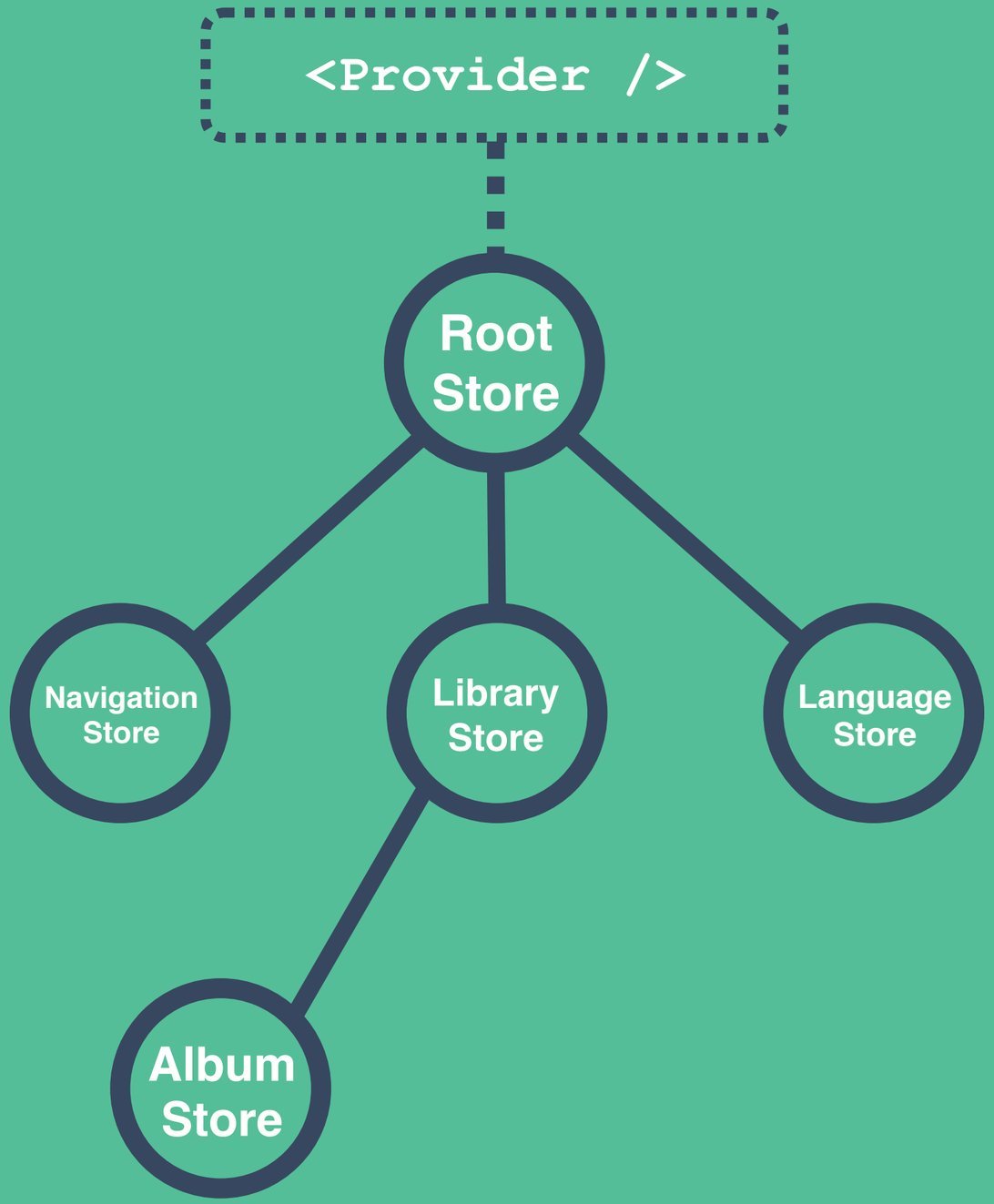
PART THREE

DESIGNING A REACTIVE PROJECT

DESIGNING STORES

1. Shape your Trees
One Root Store vs Multiple Root Stores

2. Stores Communication
How Stores communicate between each other



SHAPE YOUR TREES
ONE ROOT STORE

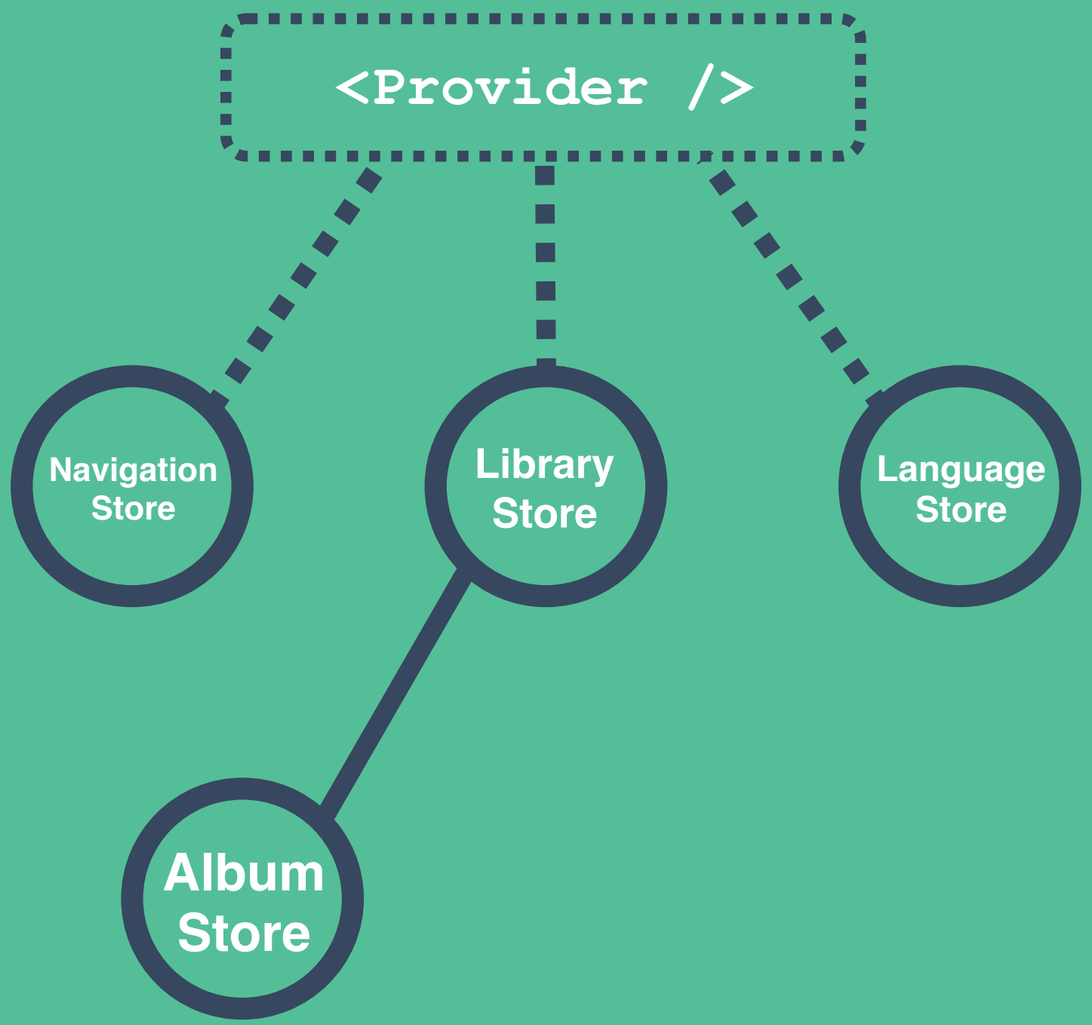
Pros 👍

» Easier to perform actions on everything at once (snapshot, creation, destroy).

» Unique environment for dependency injection.

Cons 👎

Very easy to create tightly coupled stores



SHAPE YOUR TREES

MULTIPLE ROOT STORES

Pros 👍

Easier to reason by Domain

Cons 👎

» Less immediate to perform actions on everything

» Not single environment for dependency injection

REAL WORLD
STORES COMMUNICATION 

1. Default Approach
2. Actions Wrapper
3. Dependency Injection

STORES COMMUNICATION
DEFAULT APPROACH

```
/** ------------- Root.store.js ---------------- **/

import { types } from 'mobx-state-tree';

const RootStore = types
    .model('RootStore', {
        navStore  : types.maybe(NavStore),
        pageStore : types.maybe(PageStore)
    })

/** ------------- Page.store.js ---------------- **/

import { types, getParent } from 'mobx-state-tree';

const PageStore = types
    .model('PageStore', {
        currentView : types.option(types.string, '')
    })
    .actions(self => ({
        showLoginForm() {
            self.currentView = 'login';
            getParent(self).navStore.setPath('/login')
        },
    }));
```

Each Store access directly other Stores.

» Easier when using a Single Root Store

» Each Store could end up knowing the whole structure ⚠️

STORES COMMUNICATION
ACTIONS WRAPPER

```
import { types, getParent } from 'mobx-state-tree'

const ActionsWrapperStore = types
    .model('ActionsWrapperStore', {})
    .actions(self => ({
        login() {
            authStore.login()
            pageStore.login()
            navigationStore.login()
        },
        goHome() {
            pageStore.showDefault();
            navigationStore.login()
        }
    }));
```

One Store,
to rule them all 🧙‍♂️🌋💍

» Calls directly other Stores

» Friendly interface

» Knows a lot about your App

```
/** ---------- Region.store.js ---------------- **/
const RegionStore = types
    .model('RegionStore', {
        region: types.optional(types.string, 'UK')
    })

/** ---------- Navigation.store.js ------------ **/
import { types, getEnv } from 'mobx-state-tree';

const NavigationStore = types
    .model('NavigationStore', { path: types.string })
    .view(self => ({
        get urlPath() {
            return getEnv(self).regionStore.region
                + '/' + self.path;
        }
    }));

/** ---------- index.js ----------------------- **/
const regionStore = RegionStore.create({});
const navigationStore = NavigationStore.create(
    { path: 'login'},
    { regionStore }
);

console.log(navigationStore.urlPath); // 'UK/login'
```

STORES COMMUNICATION
DEPENDENCY INJECTION

Injecting one or multiple stores into another one.

» You could use it for both Actions and Views

» Circular dependencies while loading could be non-trivial

ONE MORE THING ...

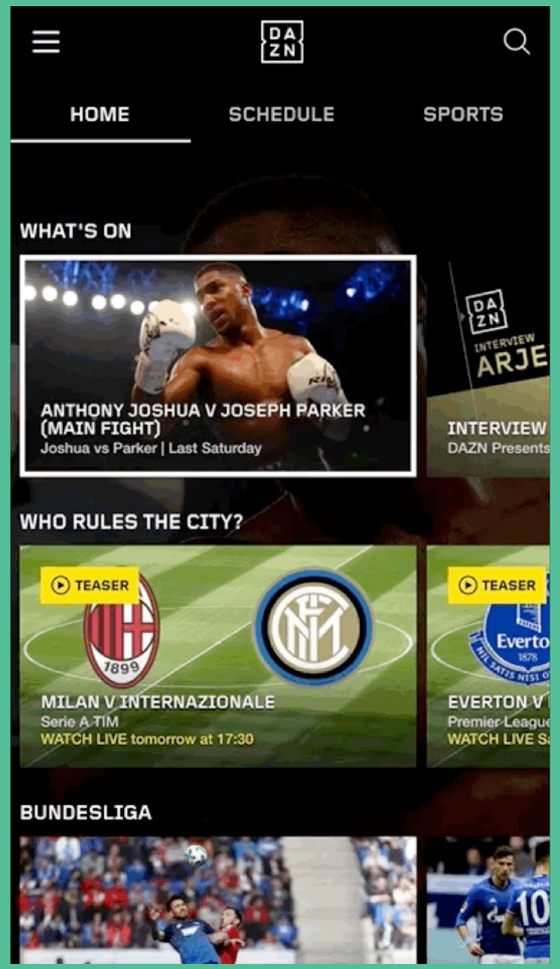
```
const DataStore = types
    .model('DataStore', {
        itemArray: types.array(types.string)
    });

const BigStore = types
    .model('BigStore', {})
    .views(self => ({
        get updatedArray(){
            return self.itemArray.map(x => `big ${x}`);
        }
    }));


const BigDataStore = types.compose(DataStore, BigStore);

const bigDataStore = BigDataStore.create({
    itemArray: ['pen', 'sword']
});

bigDataStore.itemArray    // ['pen', 'sword']
bigDataStore.updatedArray // ['big pen', 'big sword']
```

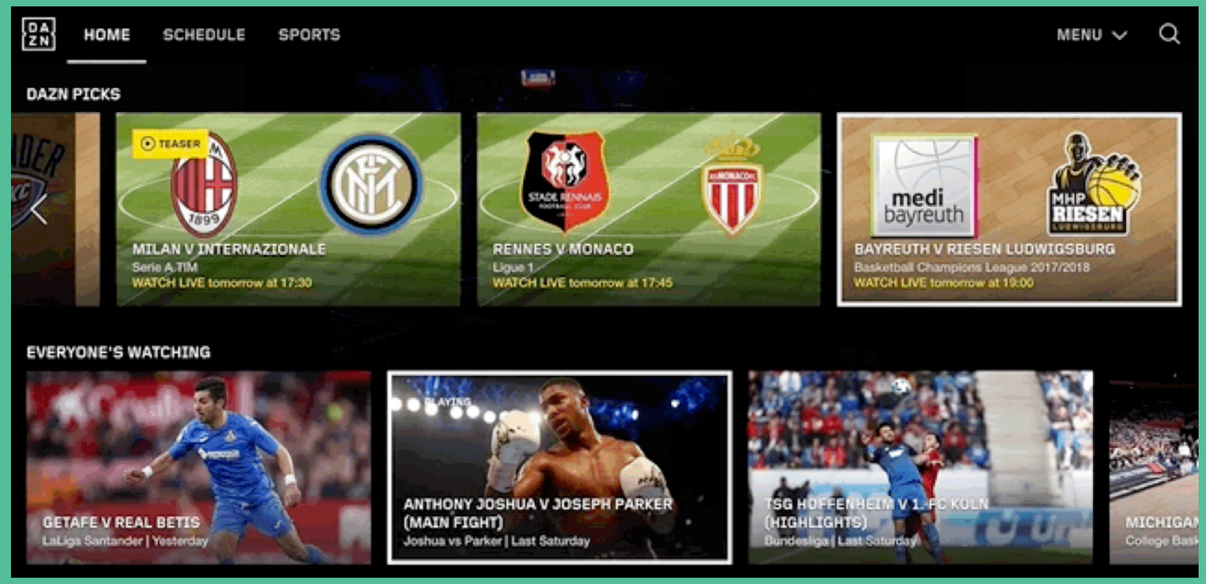
STORE COMPOSITION

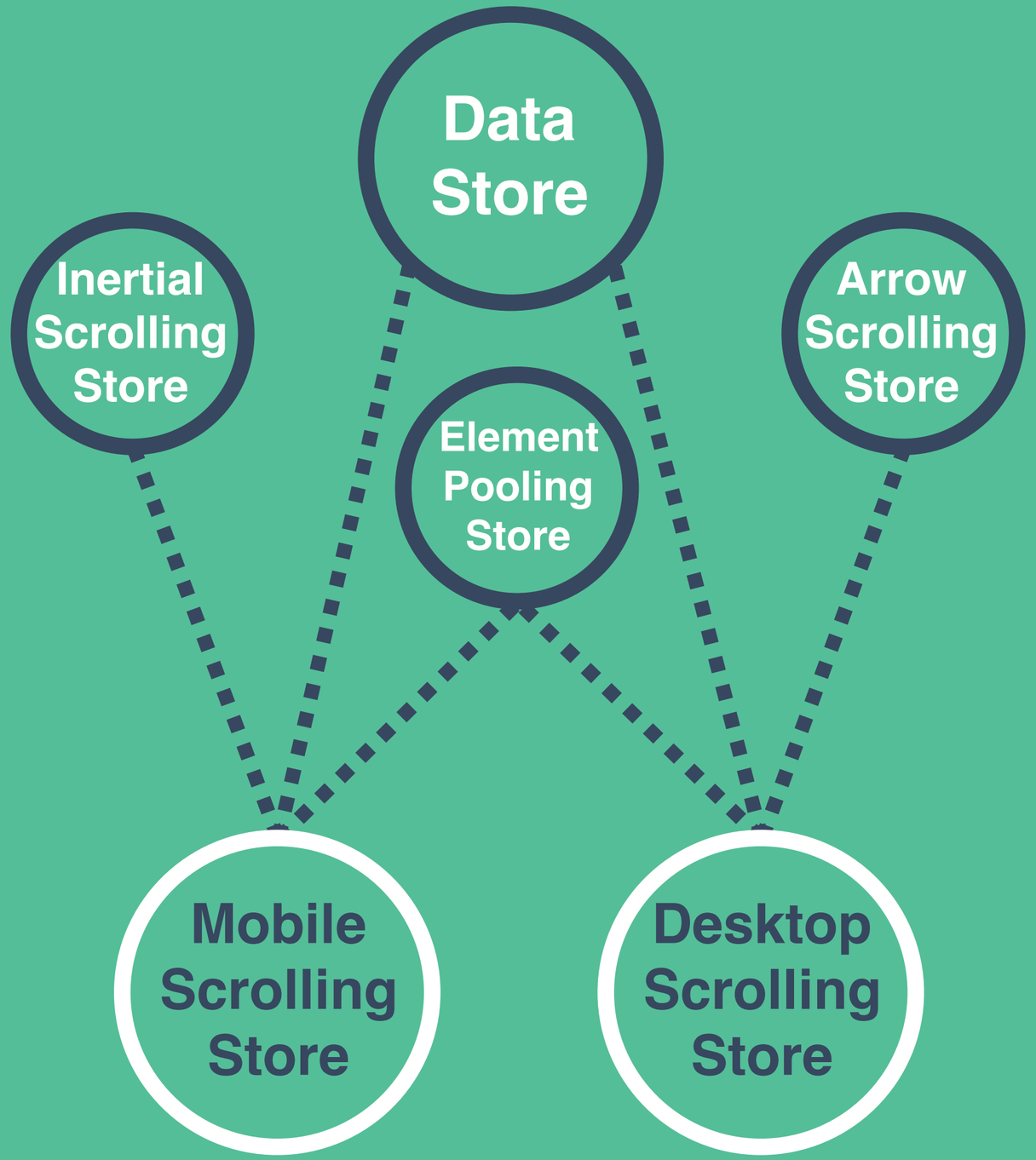
» Separation of Concerns

» Reusability

COMPOSITION
REAL WORLD EXAMPLE







COMPOSITION

REAL WORLD EXAMPLE

CONCLUSIONS

DERIVE EVERYTHING

“ANYTHING THAT CAN BE DERIVED FROM THE APPLICATION STATE, SHOULD BE DERIVED. AUTOMATICALLY”

- Michel Weststrate

TAKEAWAYS

» MobX opens the doors of Reactive Programming

» MobX State Tree provides a structure

» Shape your tree & setup the communication

» Embrace Composition!

» Embrace Reactivity!

THANKS

🤓 github.com/maxgallo/you-dont-know-mobx-state-tree

✉️ hello@maxgallo.io

twitter @_maxgallo

web maxgallo.io