Reactive Applications

with Angular 2

Understand how to build fully reactive features in Angular 2

Agenda

- The Reactive Big Picture
- Observables and RxJS
- Immutable Operations
- Reactive State and @ngrx/store
- Reactive Async
- Reactive Data Models

The Reactive Sample Project

- A **RESTful** master-detail web application that communicates to a local REST API using **json-server**
- A **reactive** master-detail web application that uses **@ngrx/store**
- We will be making the **widgets** feature reactive
- Feel free to use the existing code as a reference point
- Please explore! Don't be afraid to try new things!

ANGULAR 2 with NGRX

Item 1

This is a description

Item 2

This is a description

Item 3

This is a lovely item

Create New Item

Item Name

Enter a name

Item Description

Enter a description

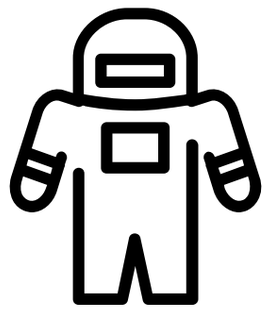
CANCEL SAVE

http://bit.ly/fem-ng2-ngrx-app

http://onehungrymind.com/fem-examples/

Pre-Challenges

- Download and run the sample application
- Wire up the **widgets** component to the **widgets-list** and **widget-details** components via **@Input** and **@Output**
- Connect the **widgets** service to communicate with RESTful api using the **HTTP** module and **Observable.toPromise**



The Reactive Big Picture

The Reactive Big Picture

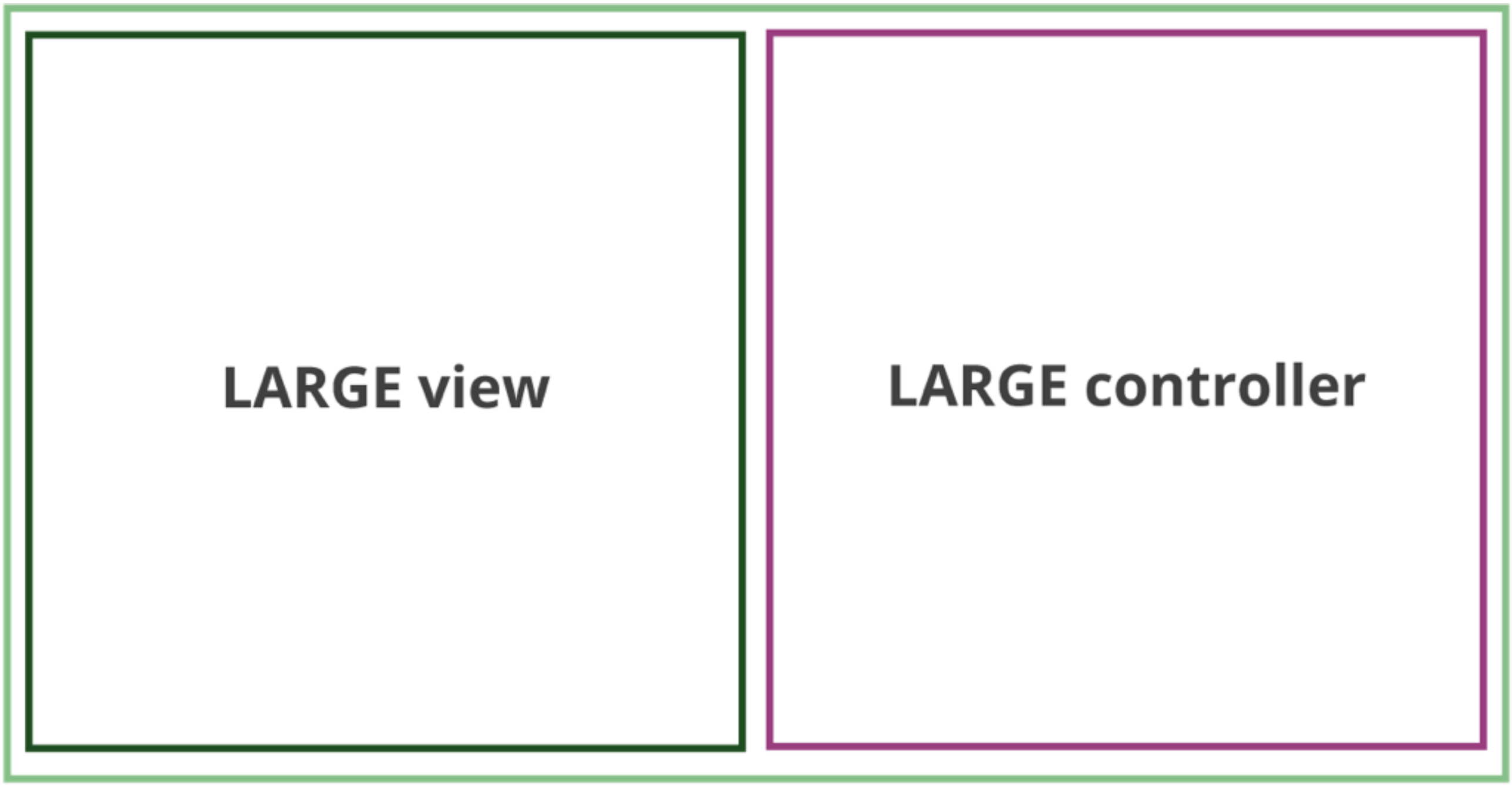
- The Reactive Sample Project
- Angular History Lesson
- Why Reactive?
- Reactive Angular 2
- Enter Redux

Angular History Lesson

tiny app == tiny view + tiny controller

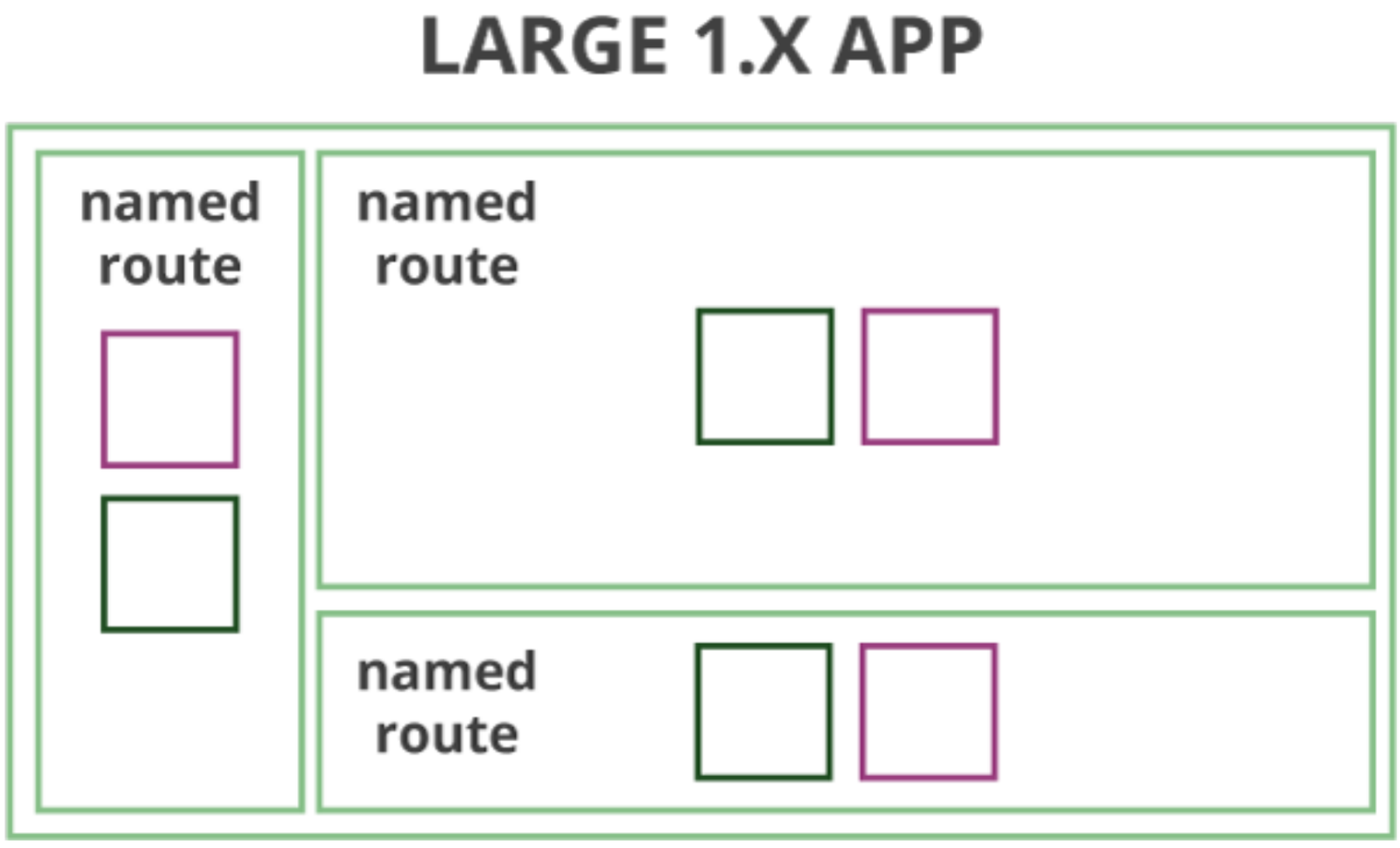
Hello Angular 1.x

Let's Get Serious

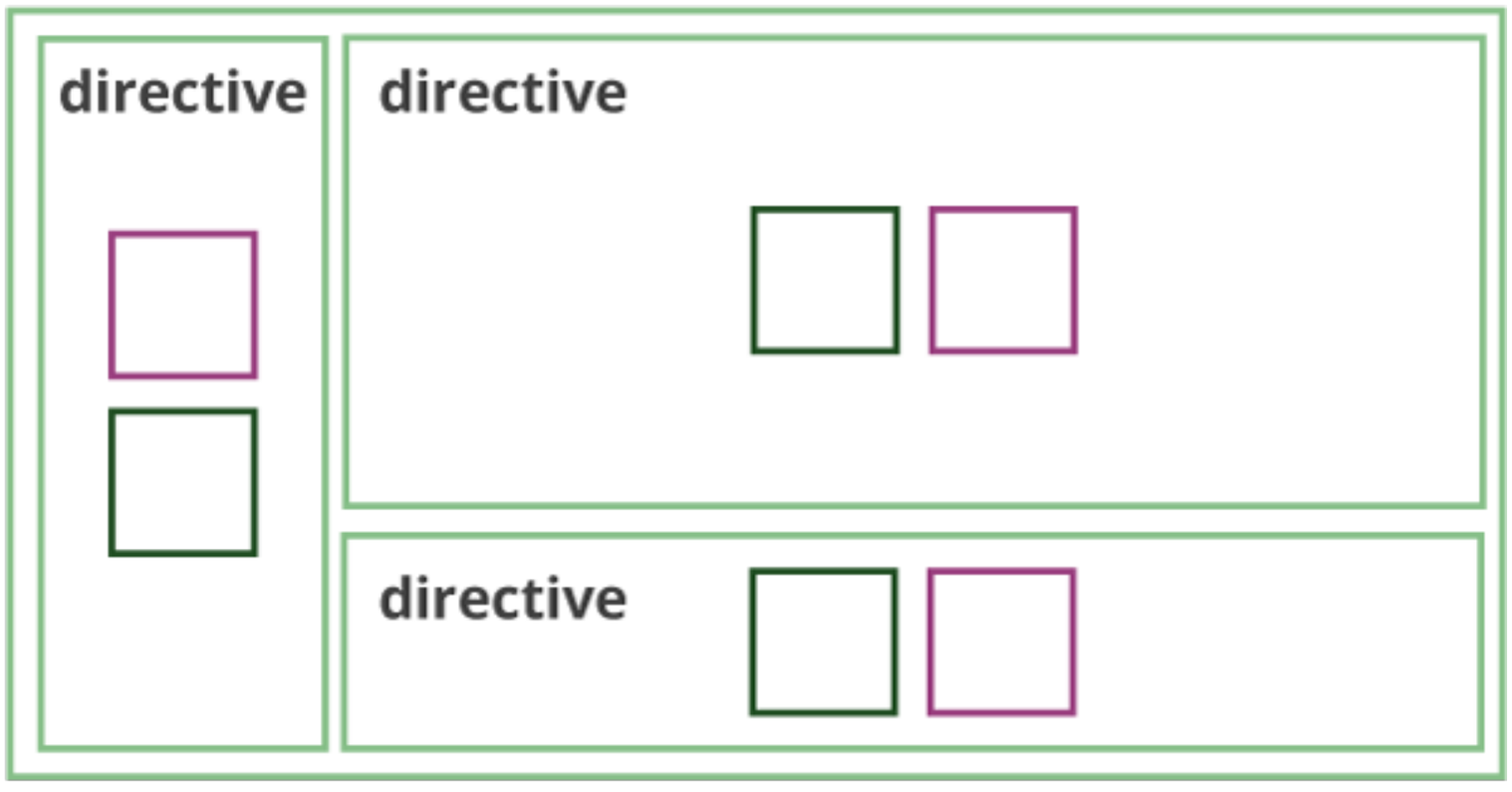


Let's Get Realistic

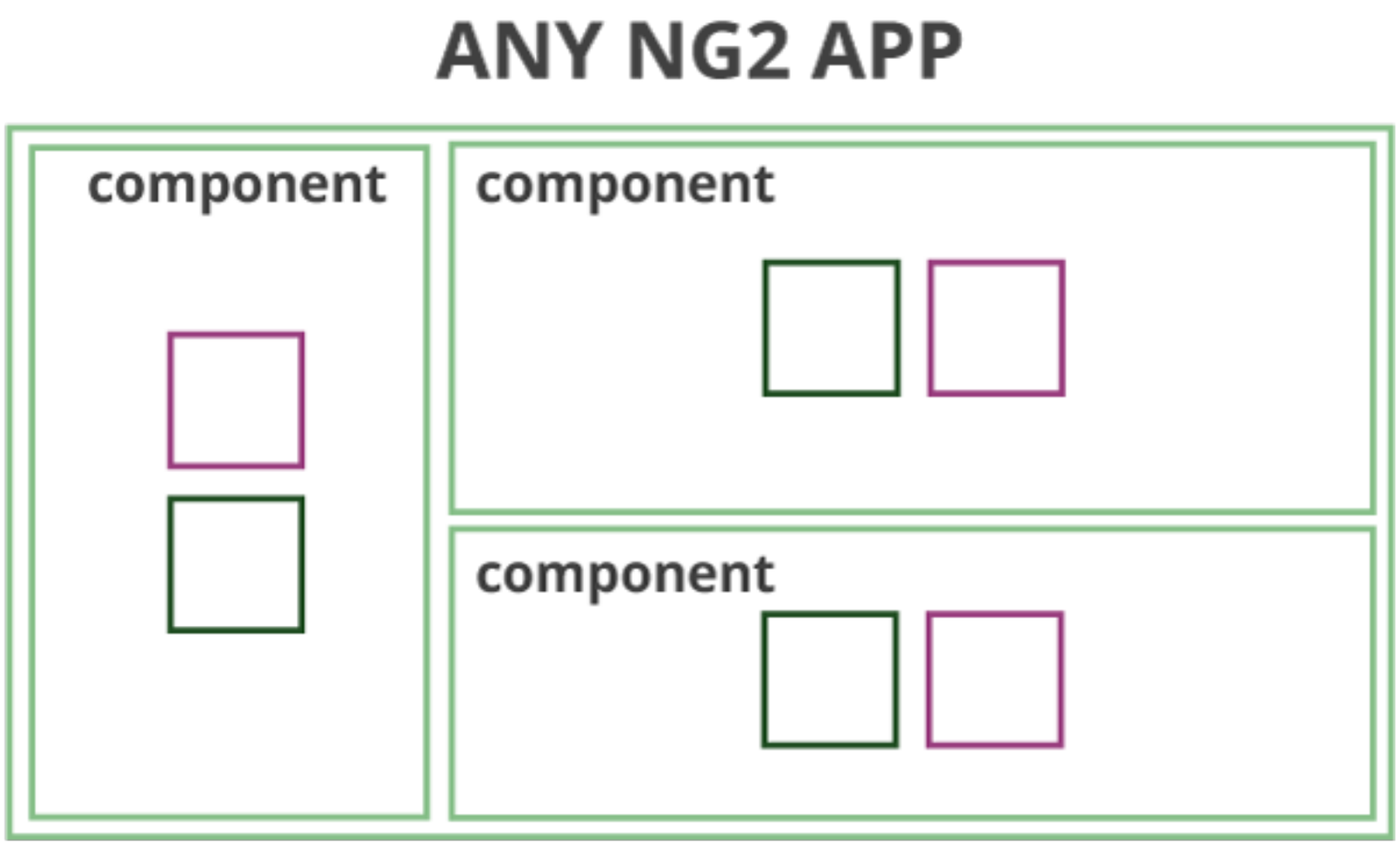
Two Solid Approaches



Named Routes

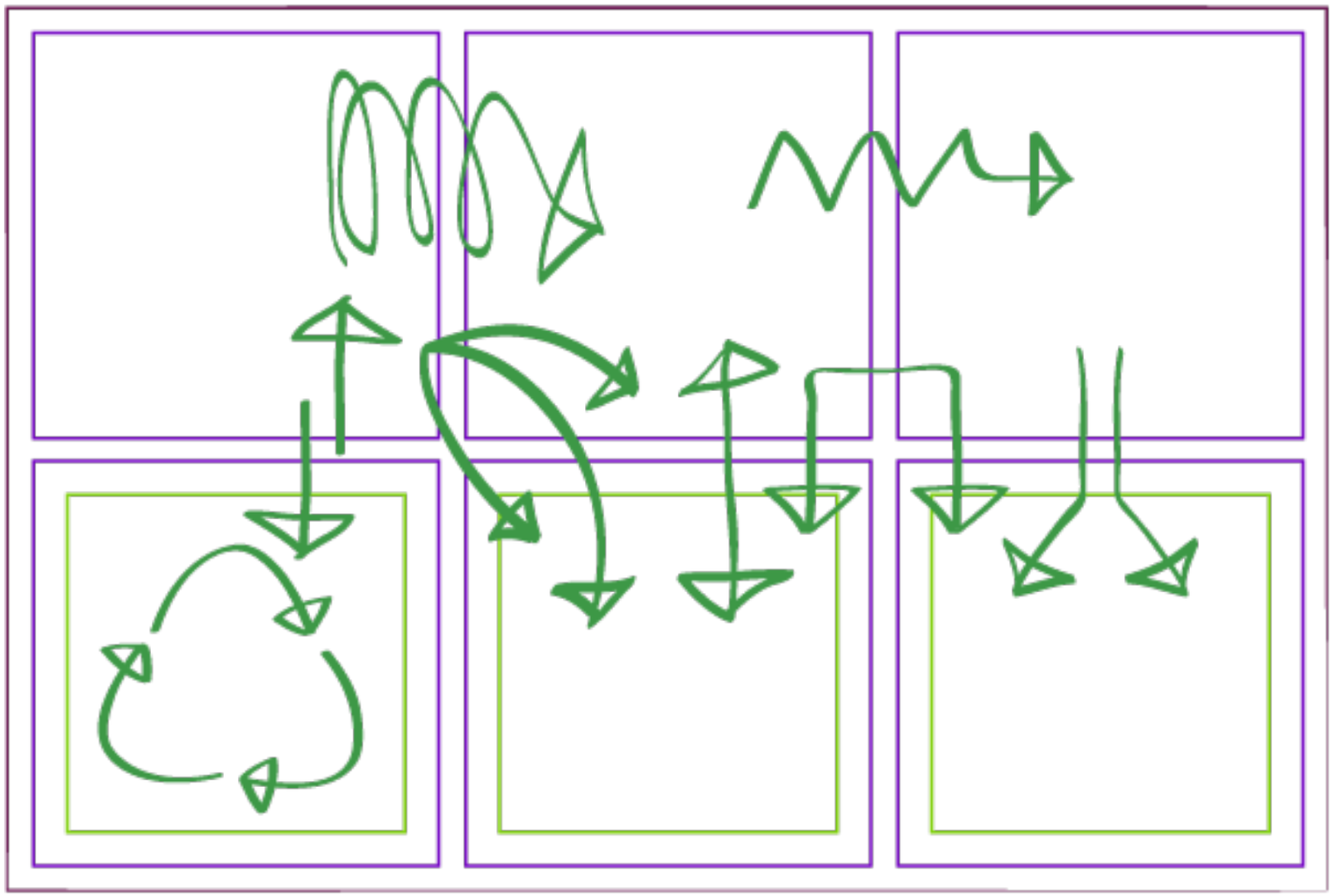


Directives



Components

Small Problem...



State Everywhere!

We need a better way to manage state

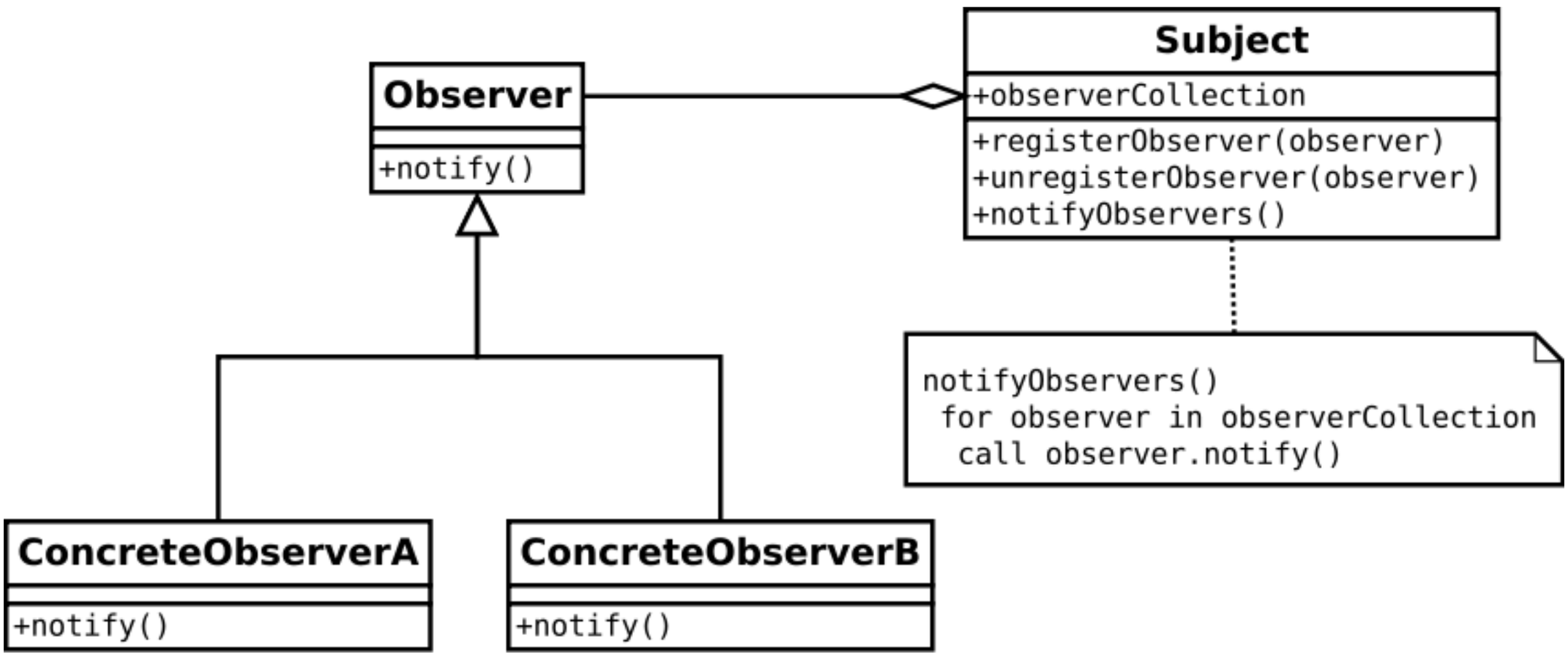
What if we only had to manage state in **ONE** place?

What if we could **RELIABLY** push new state to our app?

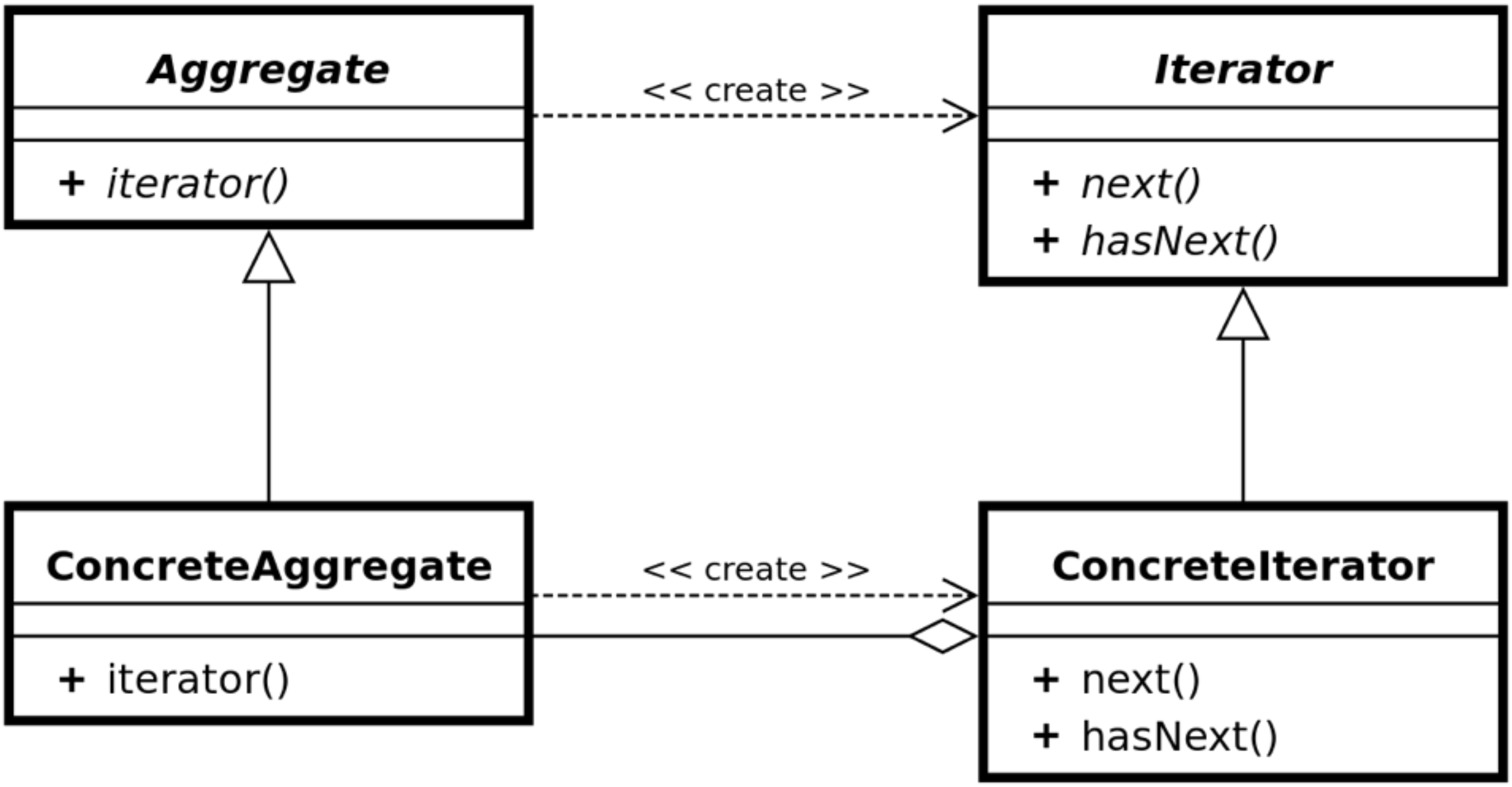
What if we could dramatically **SIMPLIFY** handling user interactions?

Why Reactive?

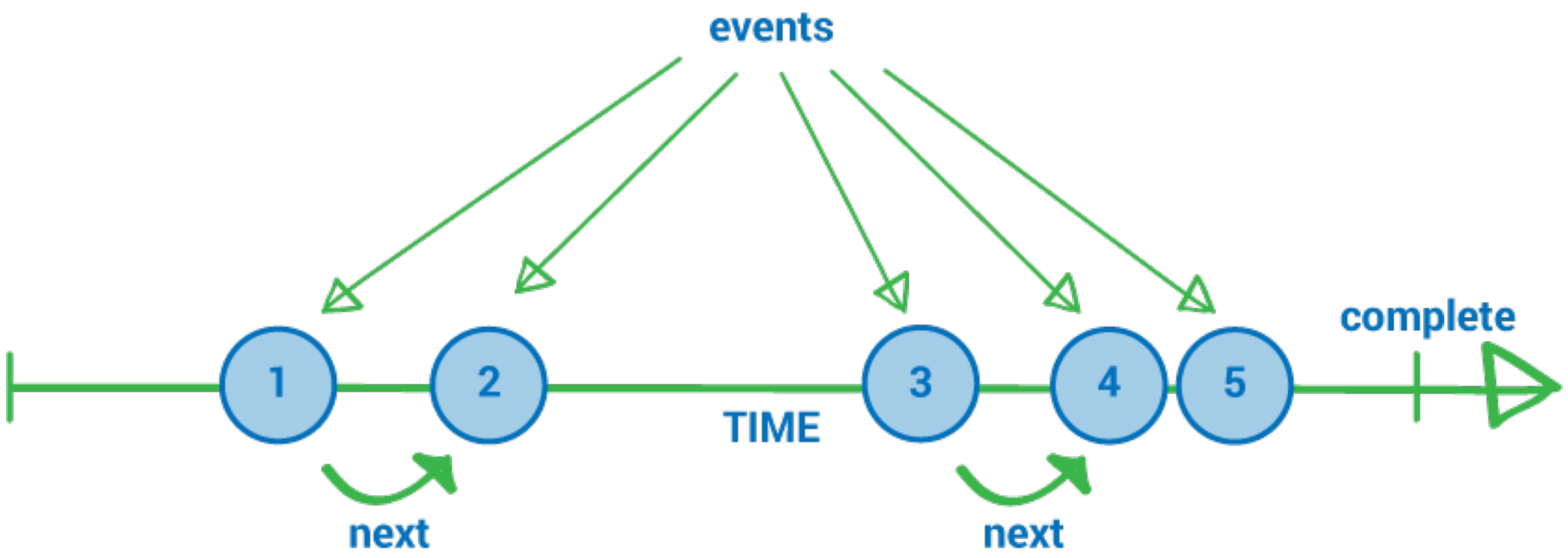
- In the context of this workshop, reactive programming is when we react to data being streamed to us over time
- The atomic building block for this is the observable object which is an extension of the Observer Pattern



Observer Pattern



Iterator Pattern



Observable Sequence

	SINGLE	MULTIPLE
SYNCHRONOUS	Function	Enumerable
ASYNCHRONOUS	Promise	Observable

Time + Value

	SINGLE	MULTIPLE
PULL	Function	Enumerable
PUSH	Promise	Observable

Value Consumption

Reactive Angular 2

- Observables are a core part of Angular 2
- Async pipes make binding to observables as easy as binding to primitives
- Observables and immutability alleviate the burden of change detection

```
this.http.get(BASE_URL)
  .map(res => res.json())
  .map(payload => ({ type: 'ADD_ITEMS', payload }))
  .subscribe(action => this.store.dispatch(action));
```

Observable Sequence

```
<div class="mdl-cell mdl-cell--6-col">
  <items-list [items]="items | async"
    (selected)="selectItem($event)" (deleted)="deleteItem($event)">
  </items-list>
</div>
<div class="mdl-cell mdl-cell--6-col">
  <item-detail
    (saved)="saveItem($event)" (cancelled)="resetItem($event)"
    [item]="selectedItem | async">Select an Item</item-detail>
</div>
```

Async Pipe

Enter Redux

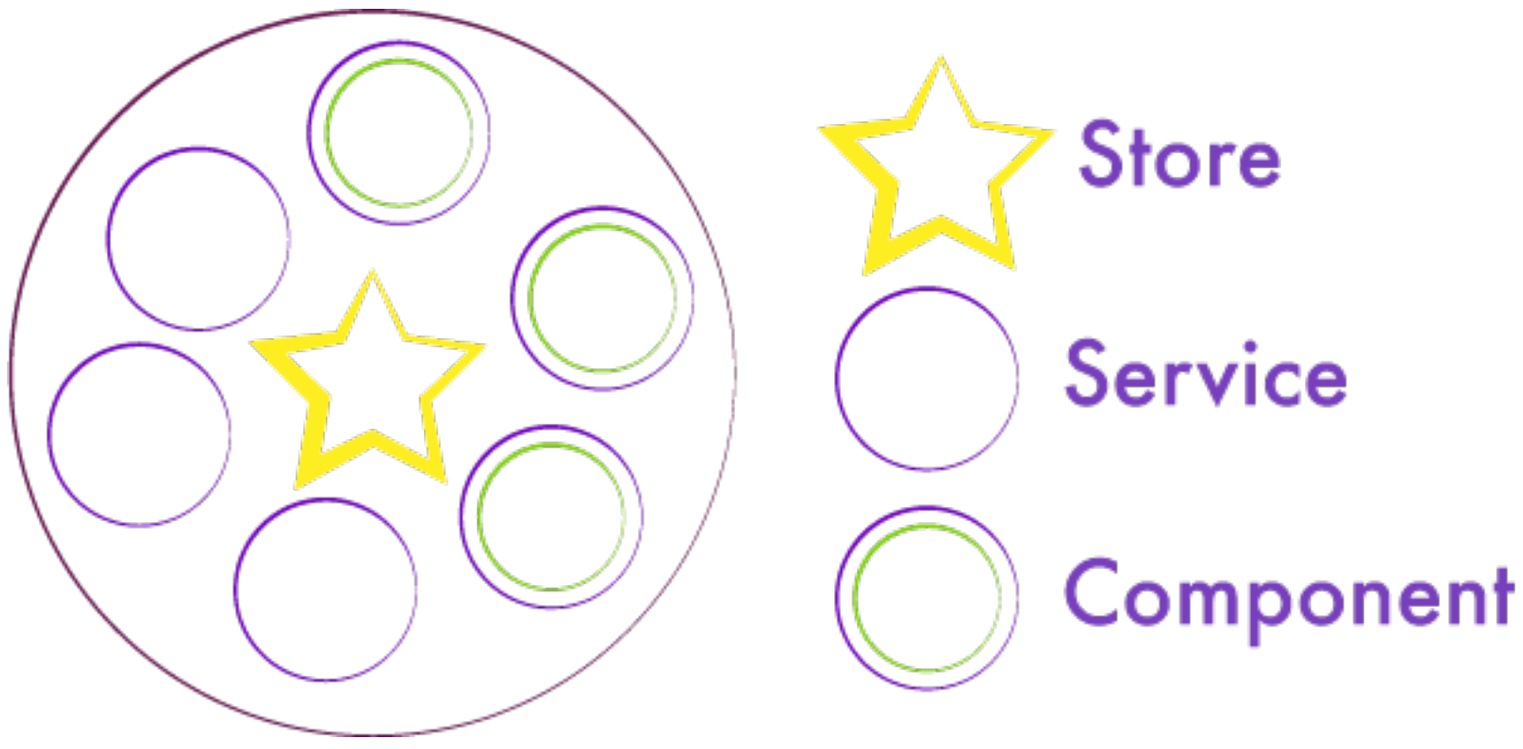
- Single, immutable state tree
- State flows down
- Events flow up
- No more managing parts of state in separate controllers and services

Redux is a library but more importantly it is a **pattern**

Required Viewing

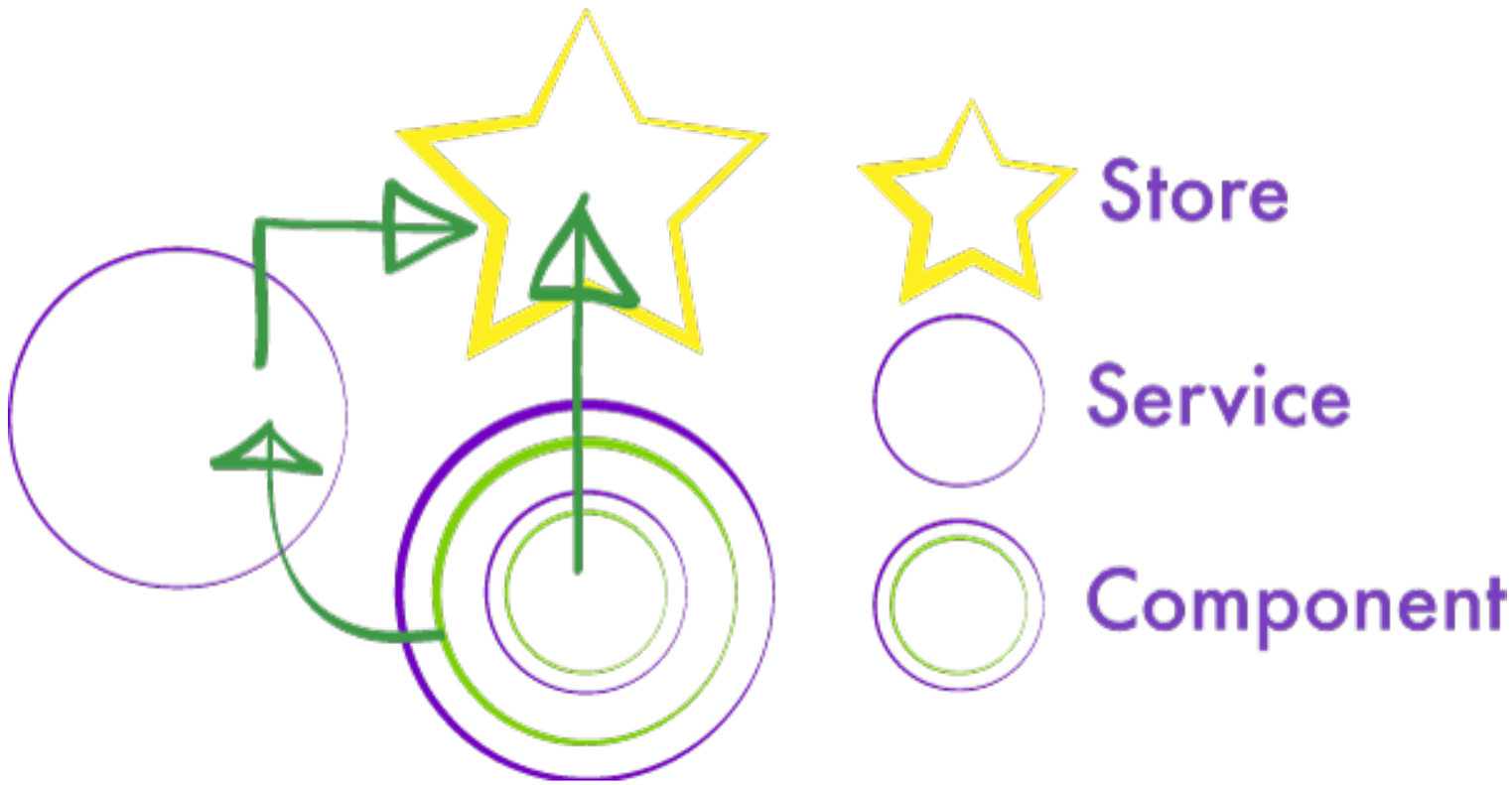
Getting Started with Redux by Dan Abramov

https://egghead.io/series/getting-started-with-redux

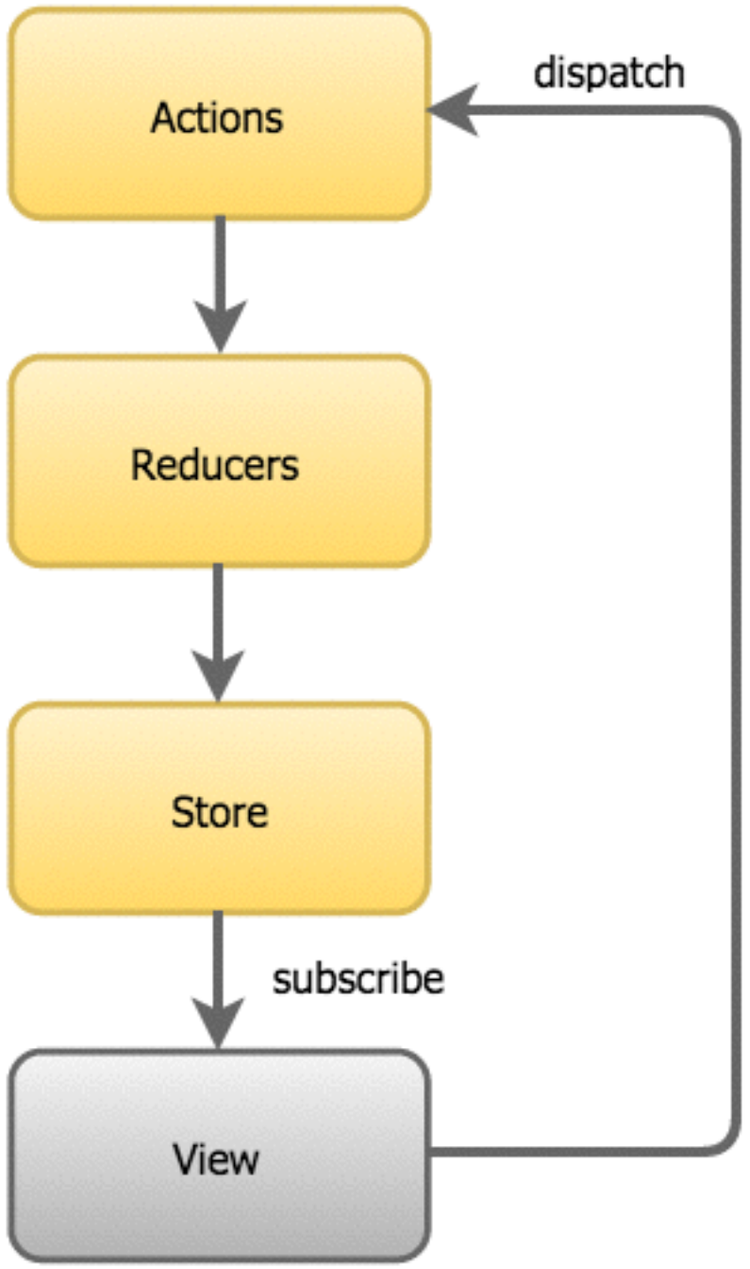


Single State Tree

State Flows Down



Events Flows Up

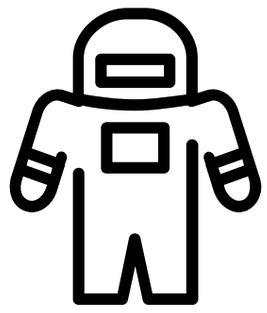


All Together Now!

Demonstration

Challenges

- Download and run the sample application
- Identify the major reactive components
- Where are we using observables?
- Where are we using async pipes?



Observables and RxJS

Observables and RxJS

- Reactive with Observables
- Data Flow with Observables
- What is RxJS?
- RxJS and Observables
- Most Common RxJS Operators
- Async Pipes

Reactive with Observables

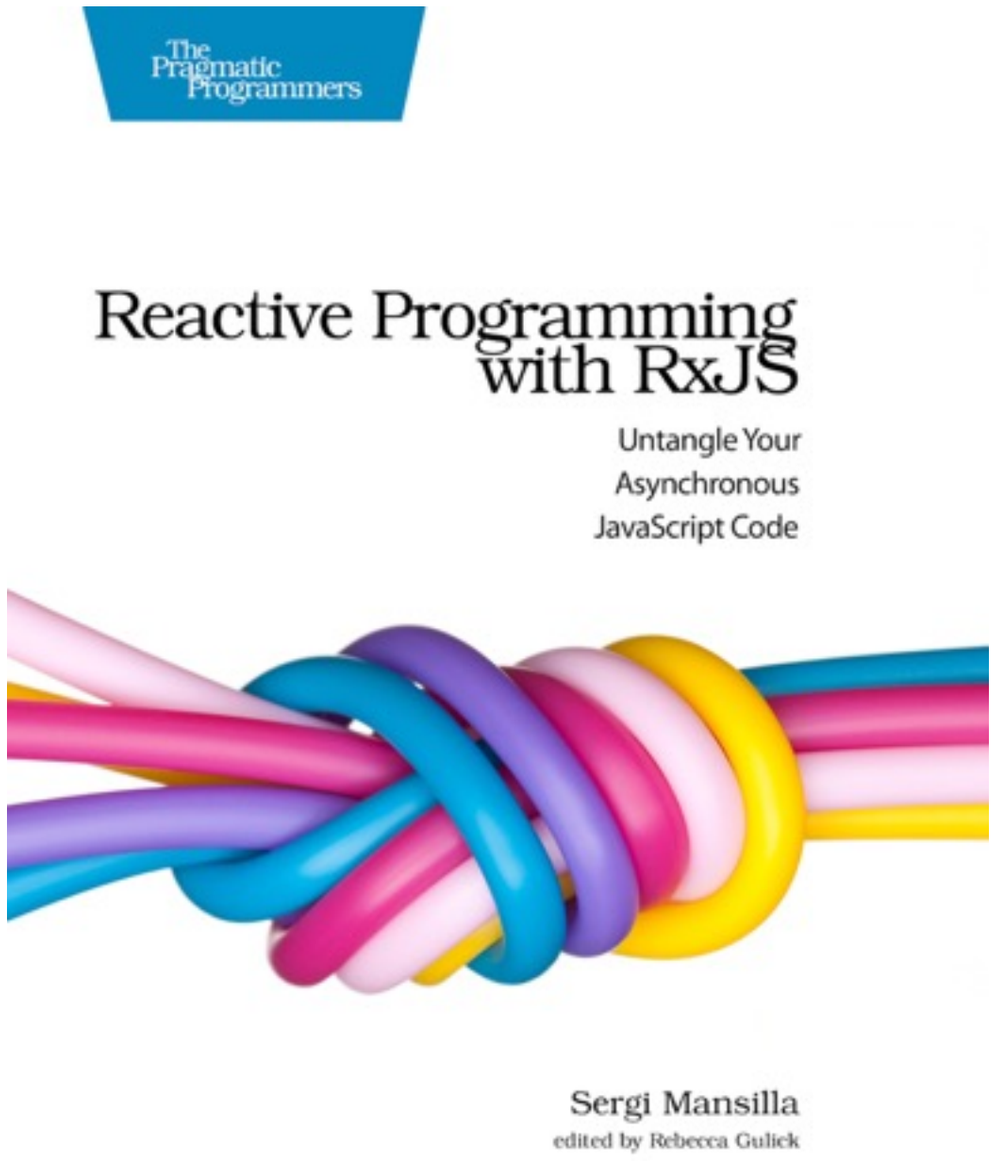
- In the observer pattern, an object (called the subject), maintains a list of its dependents, called observers, and notifies them automatically of any state changes, usually by calling one of their methods.
- This represents a push strategy as opposed to a pull (or polling) strategy
- An observer doesn’t have to constantly poll the subject for changes, the subject “pushes” notifications to the observer

RxJS

- What is RxJS?
- RxJS and Observables
- Most Common RxJS Operators
- RxJS Examples

What is RxJS?

- A set of libraries to compose asynchronous and event-based programs using observable collections
- A **TON** of operators that allow you transform an observable stream
- This is generally where the learning curve gets steep



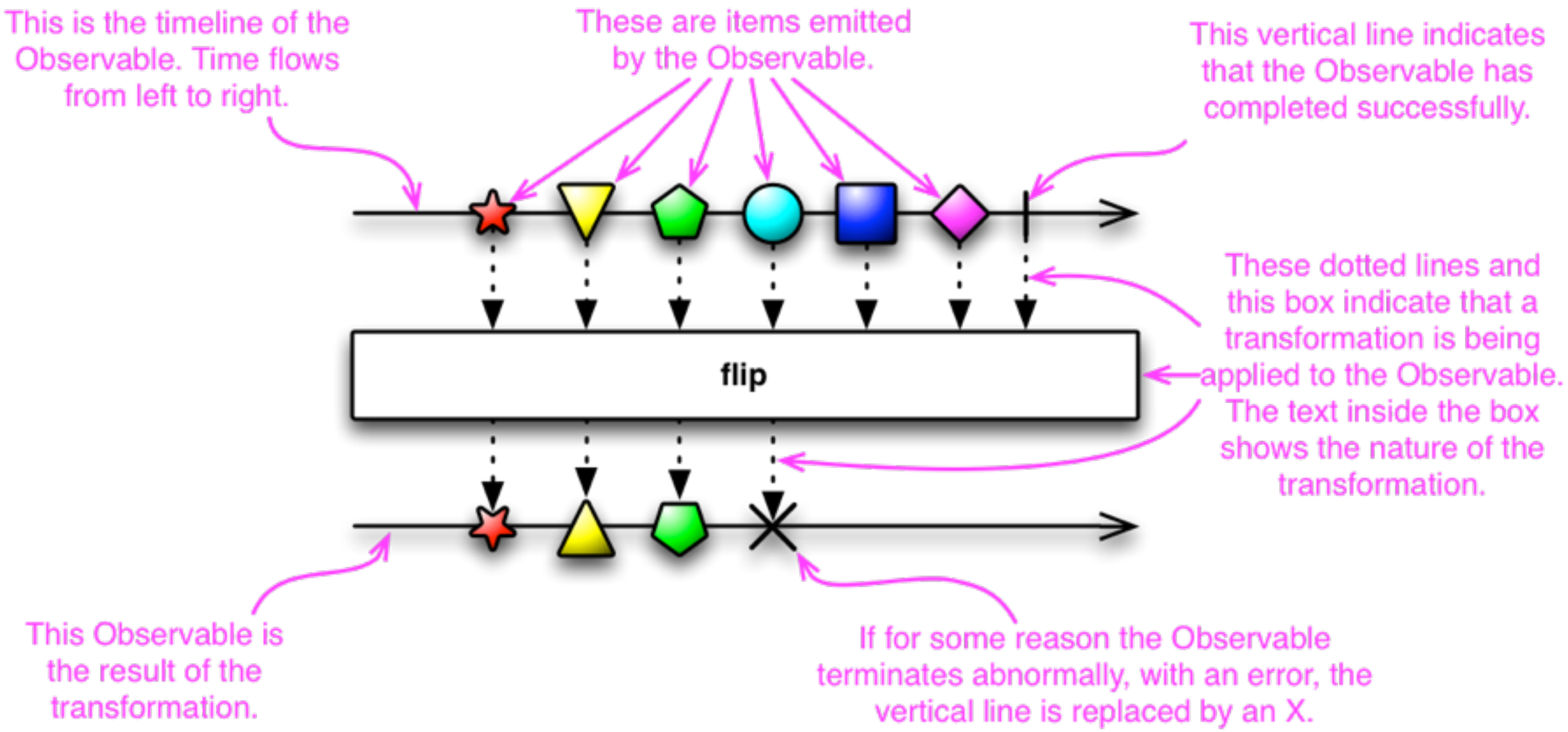
Required Reading

Reactive Programming with RxJS

https://pragprog.com/book/smreactjs/reactive-programming-with-rxjs

```
/* Get stock data somehow */
const source = getAsyncStockData();

const subscription = source
  .filter(quote => quote.price > 30)
  .map(quote => quote.price)
  .subscribe(
    price => console.log(`Prices higher than $30: ${price}`),
    err => console.log(`Something went wrong: ${err.message}`)
);

/* When we're done */
subscription.dispose();
```

Basic Example

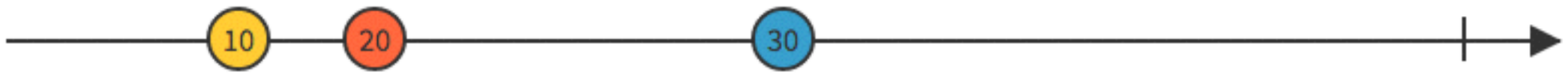

This is the timeline of the Observable. Time flows from left to right.
These are items emitted by the Observable.
This vertical line indicates that the Observable has completed successfully.
flip
These dotted lines and this box indicate that a transformation is being applied to the Observable. The text inside the box shows the nature of the transformation.
This Observable is the result of the transformation.
If for some reason the Observable terminates abnormally, with an error, the vertical line is replaced by an X.

Marbles

Most Common RxJS Operators

- map
- filter
- scan
- debounce
- distinctUntilChanged
- combineLatest
- flatMap

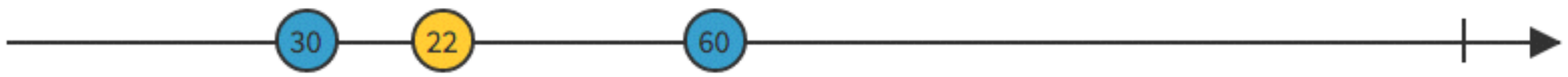

```
map(x => 10 * x)
```



```
// Array
var numbers = [1, 2, 3];
var roots = numbers.map(Math.sqrt);
// roots is now [1, 4, 9], numbers is still [1, 2, 3]

// Observable
var source = Observable.range(1, 3)
  .map(x => x * x);

var subscription = source.subscribe(
  x => console.log('Next: ' + x),
  err => console.log('Error: ' + err),
  () => console.log('Completed'));

// => Next: 1
// => Next: 4
// => Next: 9
// => Completed
```

map


```
filter(x => x > 10)
```



```
// Array
var filtered = [12, 5, 8, 130, 44].filter(x => x >= 10);
// filtered is [12, 130, 44]

// Observable
var source = Observable.range(0, 5)
  .filter(x => x % 2 === 0);

var subscription = source.subscribe(
  x => console.log('Next: ' + x),
  err => console.log('Error: ' + err),
  () => console.log('Completed'));

// => Next: 0
// => Next: 2
// => Next: 4
// => Completed
```

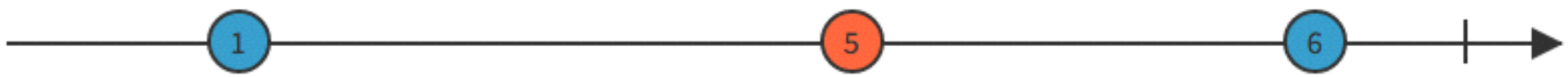
filter


```
scan((x, y) => x + y)
```

```
var source = Observable.range(1, 3)
  .scan((acc, x) => acc + x);

var subscription = source.subscribe(
  x => console.log('Next: ' + x),
  err => console.log('Error: ' + err),
  () => console.log('Completed'));

// => Next: 1
// => Next: 3
// => Next: 6
// => Completed
```

scan

debounce



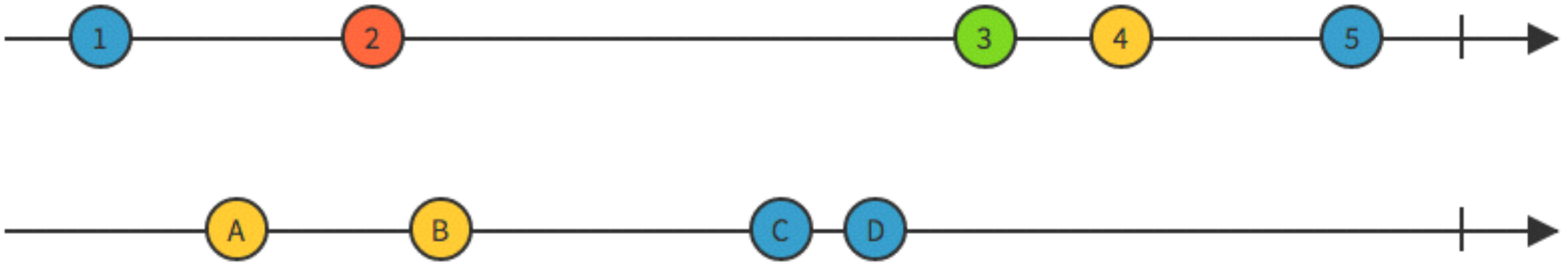
```
var array = [
  800,
  700,
  600,
  500
];

var source = Observable.for(
  array,
  function (x) { return Observable.timer(x) }
  )
  .map(function(x, i) { return i; })
  .debounce(function (x) { return Observable.timer(700); });

var subscription = source.subscribe(
  x => console.log('Next: ' + x),
  err => console.log('Error: ' + err),
  () => console.log('Completed'));

// => Next: 0
// => Next: 3
// => Completed
```

debounce

distinctUntilChanged

```
var source = Observable.of(42, 42, 24, 24)
  .distinctUntilChanged();

var subscription = source.subscribe(
  x => console.log('Next: ' + x),
  err => console.log('Error: ' + err),
  () => console.log('Completed'));

// => Next: 42
// => Next: 24
// => Completed
```

distinctUntilChanged




```
combineLatest((x, y) => "" + x + y)
```

```
var source1 = Observable.interval(100)
  .map(function (i) { return 'First: ' + i; });

var source2 = Observable.interval(150)
  .map(function (i) { return 'Second: ' + i; });

// Combine latest of source1 and source2 whenever either gives a value
var source = Observable.combineLatest(
  source1,
  source2
).take(4);
```

```
var subscription = source.subscribe(
  x => console.log('Next: ' + JSON.stringify(x)),
  err => console.log('Error: ' + err),
  () => console.log('Completed'));

// => Next: ["First: 0","Second: 0"]
// => Next: ["First: 1","Second: 0"]
// => Next: ["First: 1","Second: 1"]
// => Next: ["First: 2","Second: 1"]
// => Completed
```

combineLatest

```
var source = Observable.range(1, 2)
  .flatMap(function (x) {
    return Observable.range(x, 2);
  });

var subscription = source.subscribe(
  x => console.log('Next: ' + x),
  err => console.log('Error: ' + err),
  () => console.log('Completed'));

// => Next: 1
// => Next: 2
// => Next: 2
// => Next: 3
// => Completed
```

flatMap

Async Pipes

- Resolves async data (observables/promises) directly in the template
- Skips the process of having to manually subscribe to async methods in the component and then setting those values for the template to bind to
- No need to subscribe in the component
- We can chain any operators on the observable and leave the template the same

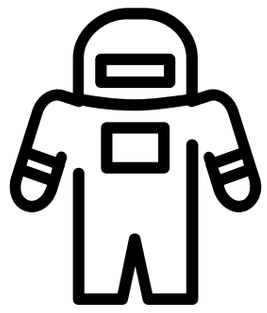
```
@Component({
  selector: 'my-app',
  template: `
  <div>
    <items-list [items]="items | async"
      (selected)="selectItem($event)" (deleted)="deleteItem($event)">
    </items-list>
  </div>
  `,
  directives: [ItemList],
  changeDetection: ChangeDetectionStrategy.OnPush
})
export class App {
  items: Observable<Array<Item>>;

  constructor(private itemsService: ItemsService) {
    this.items = itemsService.items;
  }
}
```

Async Pipes

Demonstration

Challenges

- Convert any **Observable.toPromise** calls to use an observable
- Apply **Observable.map** to your HTTP observable stream
- Apply **Observable.filter** to your HTTP observable stream



Immutable Operations

Immutable Operations

- Why Immutable?
- Avoiding Array Mutations
- Avoiding Object Mutations
- Helpful Immutable Tools

Why Immutable?

- Simplified Application Development
- No Defensive Copying
- Advanced Memoization
- Better Change Detection
- Easier to Test

Avoiding Mutations

- Array.concat
- Array.slice
- ...spread
- Array.map
- Array.filter
- Object.assign

```
export const items = (state: any = [], {type, payload}) => {
  switch (type) {
    case 'ADD_ITEMS':
      return payload;
    case 'CREATE_ITEM':
      return [...state, payload];
    case 'UPDATE_ITEM':
      return state.map(item => {
        return item.id === payload.id ?
          Object.assign({}, item, payload) : item;
      });
    case 'DELETE_ITEM':
      return state.filter(item => {
        return item.id !== payload.id;
      });
    default:
      return state;
  }
};
```

Avoiding Mutations

```
export const items = (state: any = [], {type, payload}) => {
  switch (type) {
    case 'ADD_ITEMS':
      return payload;
    case 'CREATE_ITEM':
      return [...state, payload];
    case 'UPDATE_ITEM':
      return state.map(item => {
        return item.id === payload.id ?
          Object.assign({}, item, payload) : item;
      });
    case 'DELETE_ITEM':
      return state.filter(item => {
        return item.id !== payload.id;
      });
    default:
      return state;
  }
};
```

Avoiding Mutations

```
export const items = (state: any = [], {type, payload}) => {
  switch (type) {
    case 'ADD_ITEMS':
      return payload;
    case 'CREATE_ITEM':
      return [...state, payload];
    case 'UPDATE_ITEM':
      return state.map(item => {
        return item.id === payload.id ?
          Object.assign({}, item, payload) : item;
      });
    case 'DELETE_ITEM':
      return state.filter(item => {
        return item.id !== payload.id;
      });
    default:
      return state;
  }
};
```

Avoiding Mutations

Helpful Immutable Tools

- Object.freeze
- deep-freeze
- eslint-plugin-immutable
- Immutable.js
- Ramda.js

Object.freeze

The **Object.freeze()** method freezes an object: that is, prevents new properties from being added to it; prevents existing properties from being removed; and prevents existing properties, or their enumerability, configurability, or writability, from being changed. **In essence the object is made effectively immutable.** The method returns the object being frozen.

deep-freeze

recursively **Object.freeze()** objects. #micDrop

eslint-plugin-immutable

This is an ESLint plugin to disable all mutation in JavaScript. #micDrop

eslint-plugin-immutable

This is an ESLint plugin to disable all mutation in JavaScript. #micDrop

"This is an ESLint plugin to disable all mutation in JavaScript. Think this is a bit too restrictive? Well if you're using Redux and React, there isn't much reason for your code to be mutating anything. Redux maintains a mutable pointer to your immutable application state, and React manages your DOM state. Your components should be stateless functions, translating data into Virtual DOM objects whenever Redux emits a new state. These ESLint rules explicitly prohibit mutation, effectively forcing you to write code very similar to Elm in React."

Jafar Husain

This is an ESLint plugin to disable all mutation in JavaScript. Think this is a bit too restrictive? Well if you're using **@ngrx** and **Angular 2**, there isn't much reason for your code to be mutating anything. **@ngrx** maintains a mutable pointer to your immutable application state, and **Angular 2** manages your DOM state. Your components should be stateless functions, translating data into DOM objects whenever **@ngrx** emits a new state. These ESLint rules explicitly prohibit mutation, effectively forcing you to write code very similar to Elm in **Angular 2.**

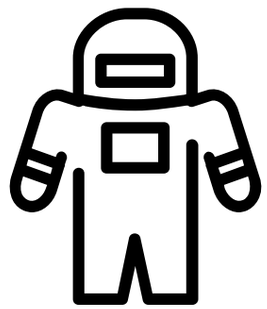
Lukas and Scott

IMMUTABLE

Demonstration

Challenges

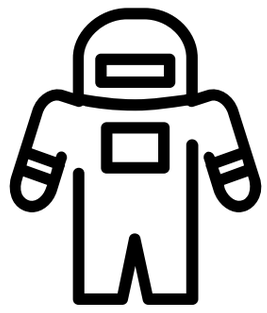
- Create immutable methods in the **widgets service** to create, read, update and delete the widgets collection.



Reactive State with @ngrx/store

Reactive State

- Redux and @ngrx/store
- Store
- Reducers
- Actions
- store.select
- store.dispatch

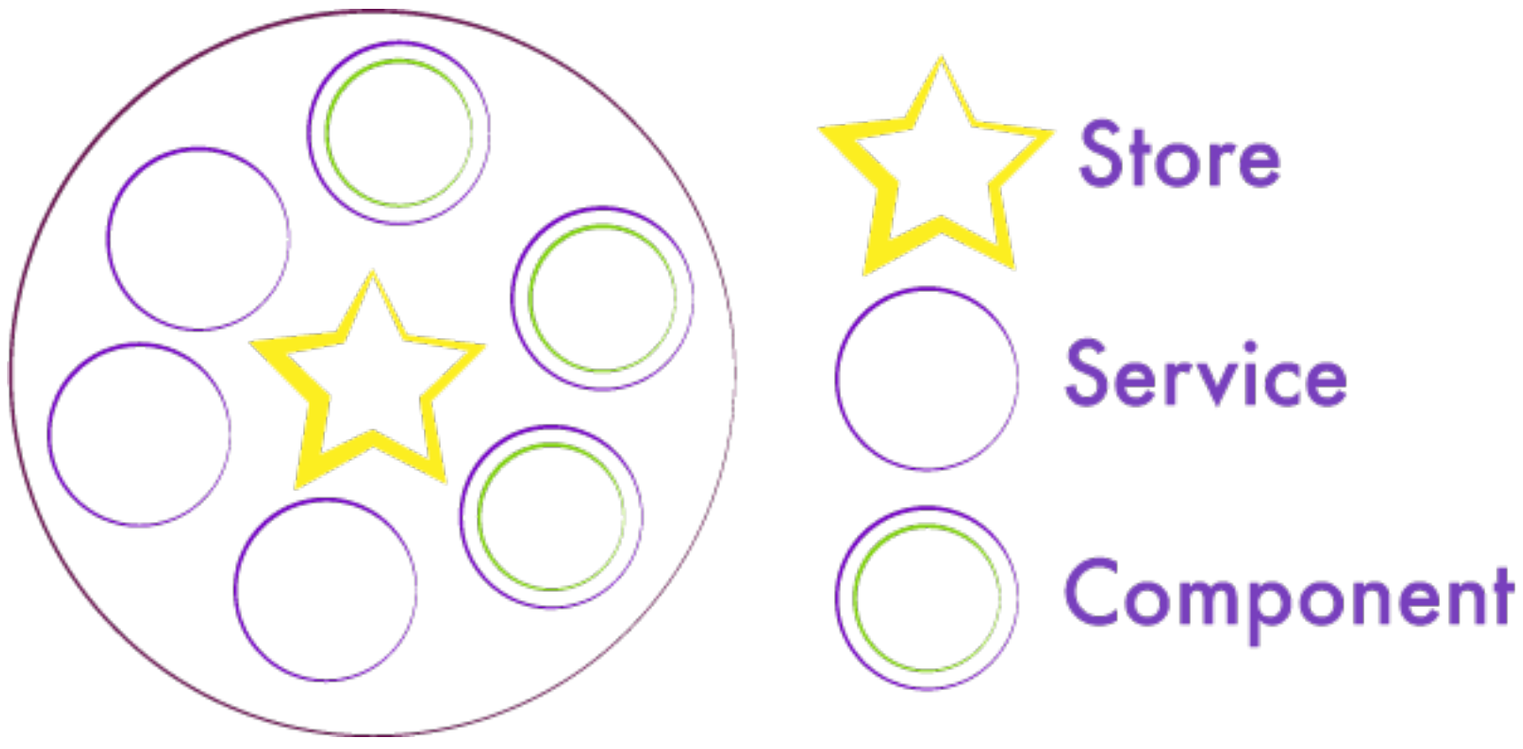


Redux and @ngrx/store

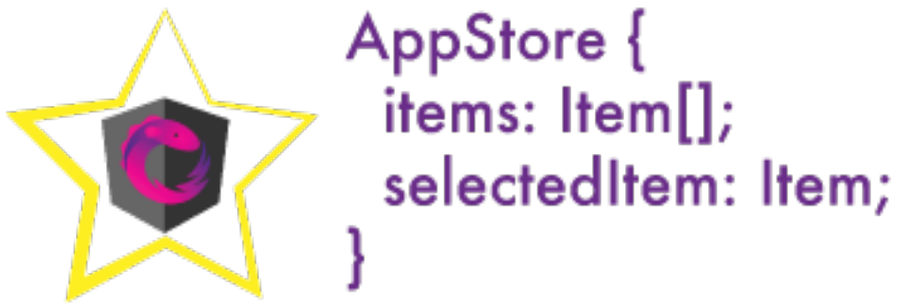
- RxJS powered state management for Angular 2 apps inspired by Redux
- @ngrx/store operates on the same principles as redux
- Slightly different because it uses RxJS
- That means that we can “subscribe” to our state, which means we can use the async pipe to display our state directly in our template

Store

- The store can be thought of as "database" of the application
- State manipulation happens in reducers which are registered with the store
- Takes reducers and provides an observable for the resulting state of each one
- Store can perform pre-reducer and post-reducer methods via middleware



Single State Tree



Single State Tree

```ts
export interface Item {
  id: number;
  name: string;
  description: string;
};
```

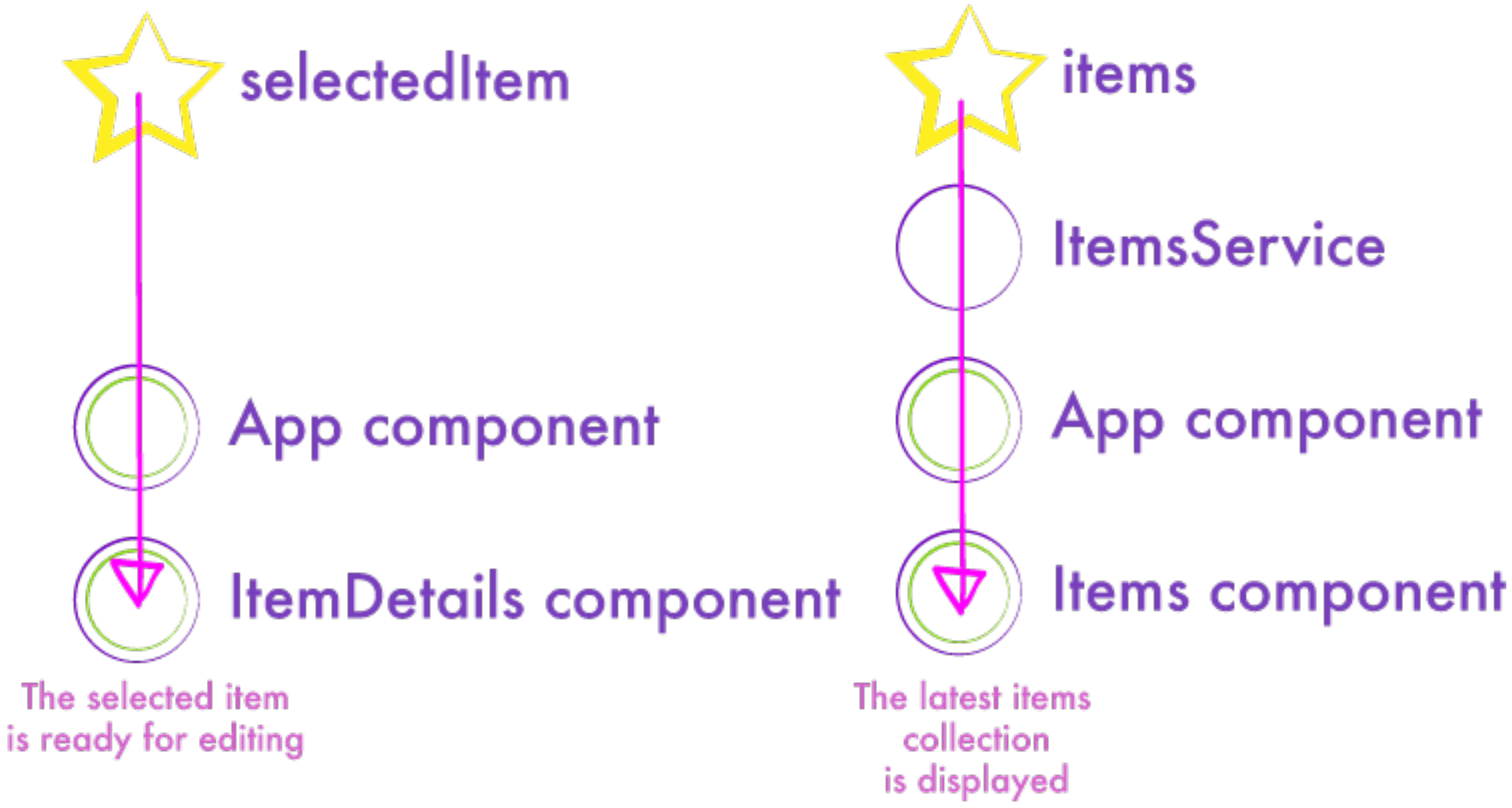
```ts
export interface AppStore {
  items: Item[];
  selectedItem: Item;
};
```

Single State Tree

Reducers

- A method that takes the current state and an action as parameters
- Returns the new state based on the provided action type
- Reducer functions should be pure functions

State Flows Down



State Flows Down

```
export const selectedItem = (state: any = null, {type, payload}) => {
  switch (type) {
    case 'SELECT_ITEM':
      return payload;
    default:
      return state;
  }
};
```

Reducers

provideStore

- Make your reducers available to your application by registering them with **provideStore**
- You can register reducers as well as initial state for reducers

```
import {App} from './src/app';
import {provideStore} from '@ngrx/store';
import {items} from './src/common/stores/items.store';
import {selectedItem} from './src/common/stores/selectedItem.store';

bootstrap(App, [
  provideStore({items, selectedItem})
]);
```

provideStore

store.select

- Returns an observable of the particular data type we want to display
- We can use **combineLatest** to create a subset of multiple data types

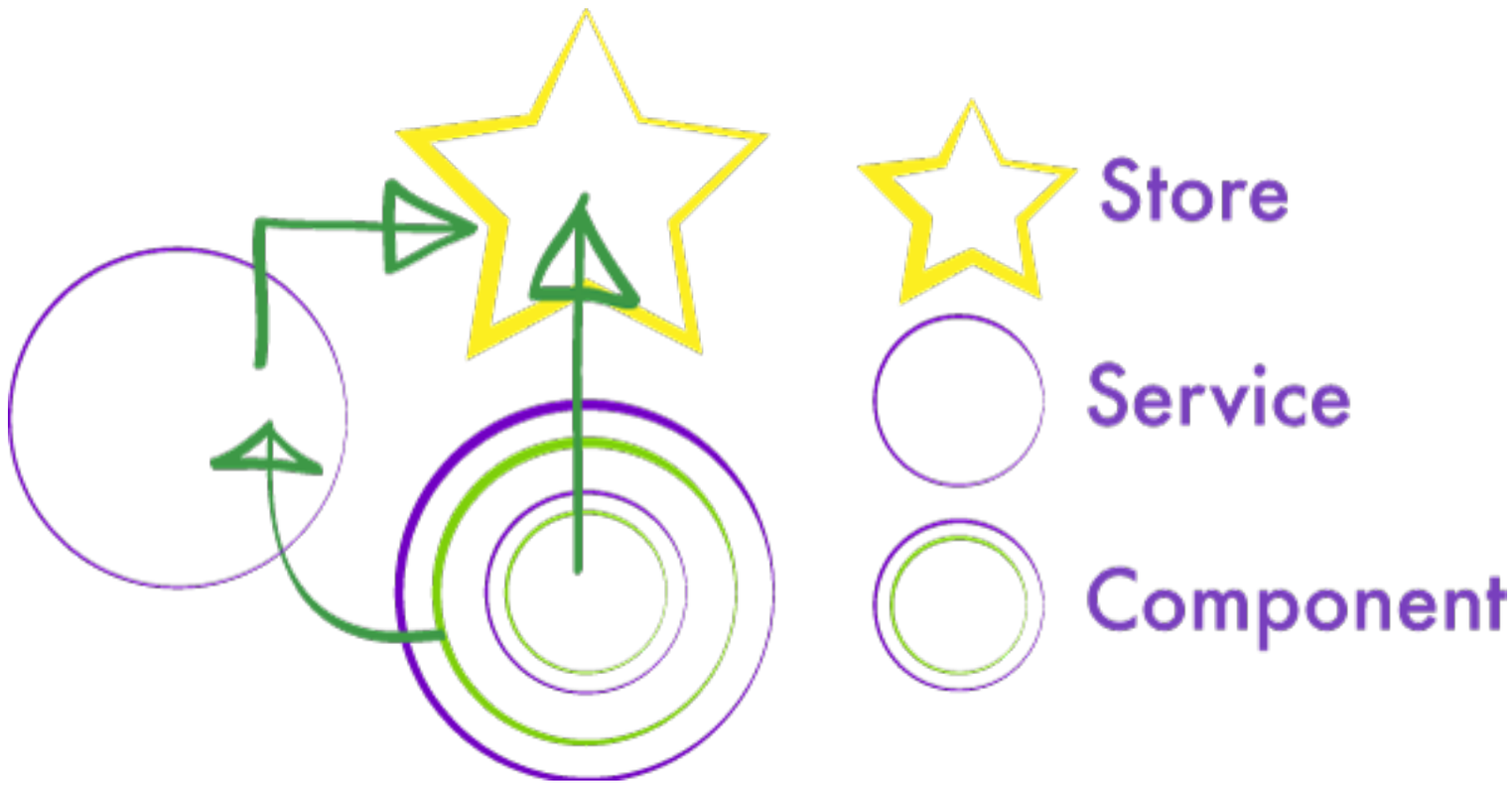
```
// items component
this.selectedItem = store.select('selectedItem');
```

```
// items template
<item-detail
  (saved)="saveItem($event)" (cancelled)="resetItem($event)"
  [item]="selectedItem | async">Select an Item</item-detail>
```

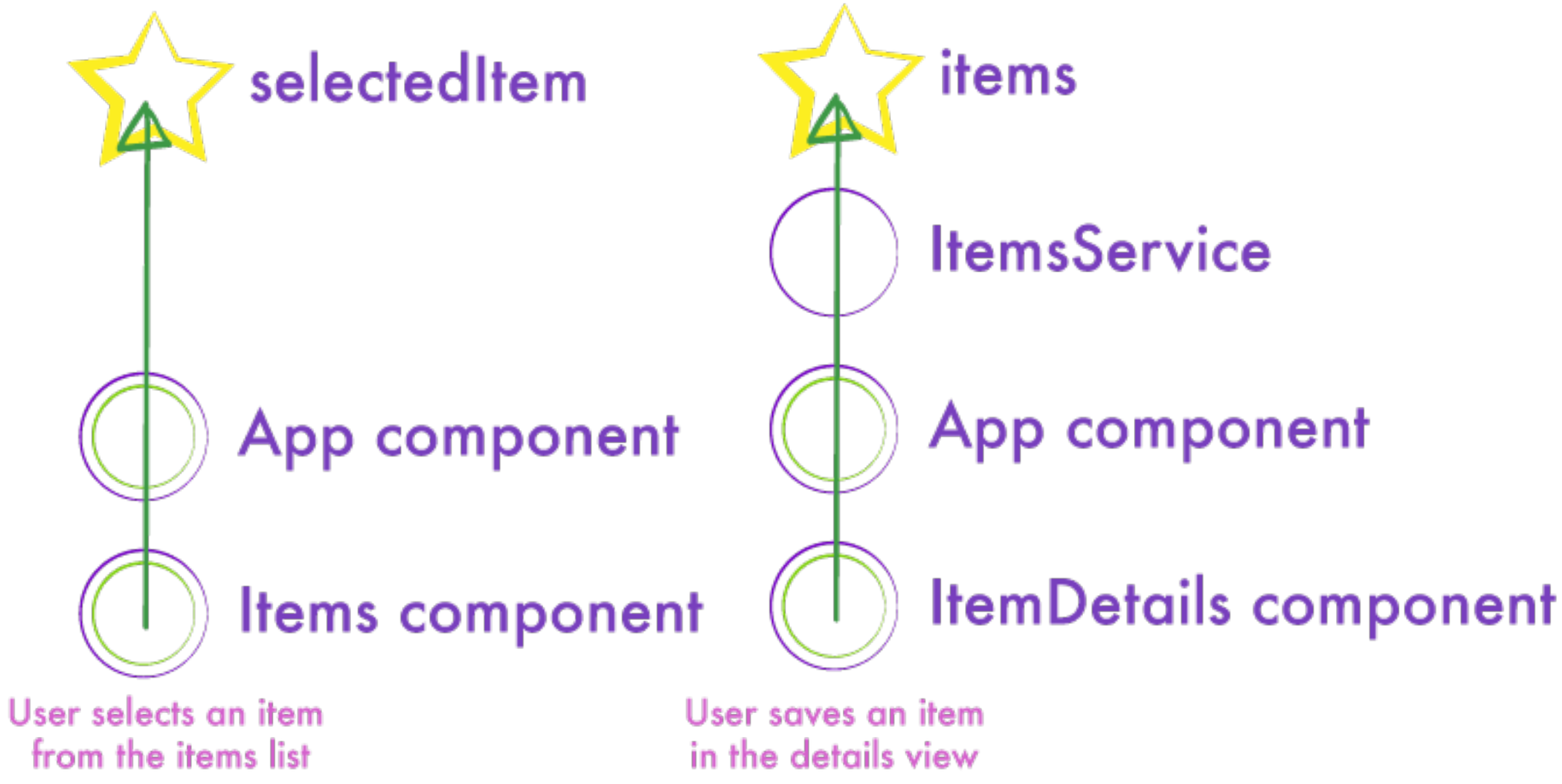
store.select

Actions

- Generally Angular 2 services that dispatch events to the reducer
- Have a type and a payload
- Based on the action type, the reducer will take the payload and return new state



Events Flows Up



Events Flows Up

Actions

- Generally Angular 2 services that dispatch events to the reducer
- Have a type and a payload
- Based on the action type, the reducer will take the payload and return new state

store.dispatch

- Sends an action to the store, which in turn calls the appropriate reducer and updates our selected data type
- Call it straight from the component or from a service

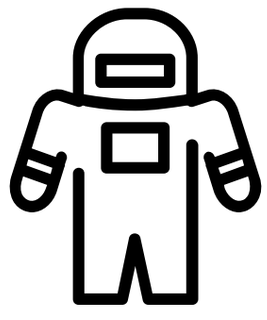
```
selectItem(item: Item) {
  this.store.dispatch({type: 'SELECT_ITEM', payload: item});
}
```

store.dispatch

Demonstration

Challenges

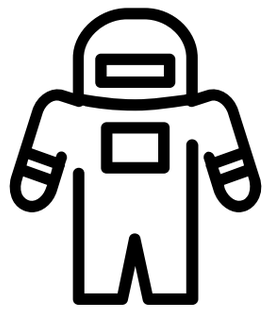
- Create a reducer function for a new data type
- Bootstrap it with the app by providing it to the store
- Pull the new data type into your component by selecting it from the store
- Update the state by dispatching an action to the store
- **BONUS** use **combineLatest** to create a subset of two different data types



Demonstration

Challenges

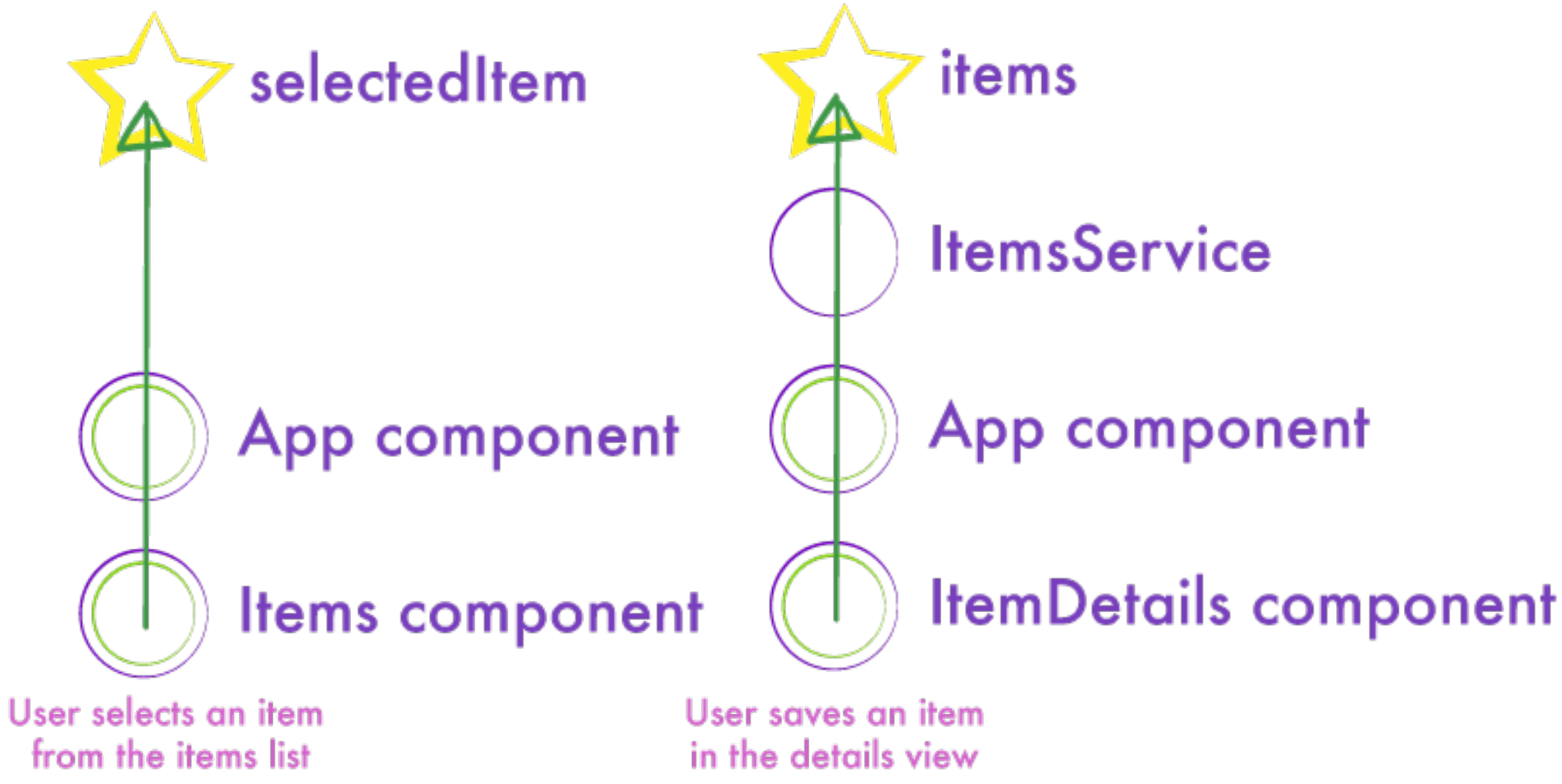
- Create a reducer for **selectedWidget**
- Register the **selectedWidget** reducer with the **provideStore**
- Use **store.select** to get the currently selected widget and display it your view
- Use **store.dispatch** to set the selected widget in the **selectedWidget** reducer



Reactive Async

Reactive Async

- It is inevitable that we will need to perform an asynchronous operation in our application
- We can delegate these operations in a service that is then responsible for dispatching the appropriate event to the reducers



Events Flows Up

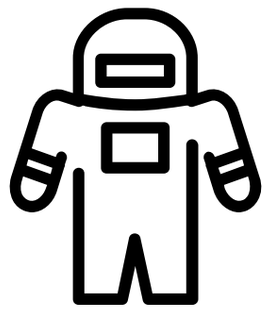
```
@Injectable()
export class ItemsService {
  items: Observable<Array<Item>>;

  constructor(private http: Http, private store: Store<AppStore>) {
    this.items = store.select('items');
  }

  loadItems() {
    this.http.get(BASE_URL)
      .map(res => res.json())
      .map(payload => ({ type: 'ADD_ITEMS', payload }))
      .subscribe(action => this.store.dispatch(action));
  }
}
```

Async Services

Demonstration

Challenges

- Build a **widgets reducer** and register it with the application
- Create a handler in the **widgets reducer** to handle getting all the widgets
- Convert the **widgets service** to reactively handle fetching the **widgets** and dispatching the appropriate event to the **widgets** reducer



Thanks!