Learn to Build Reactive Apps with Angular 2

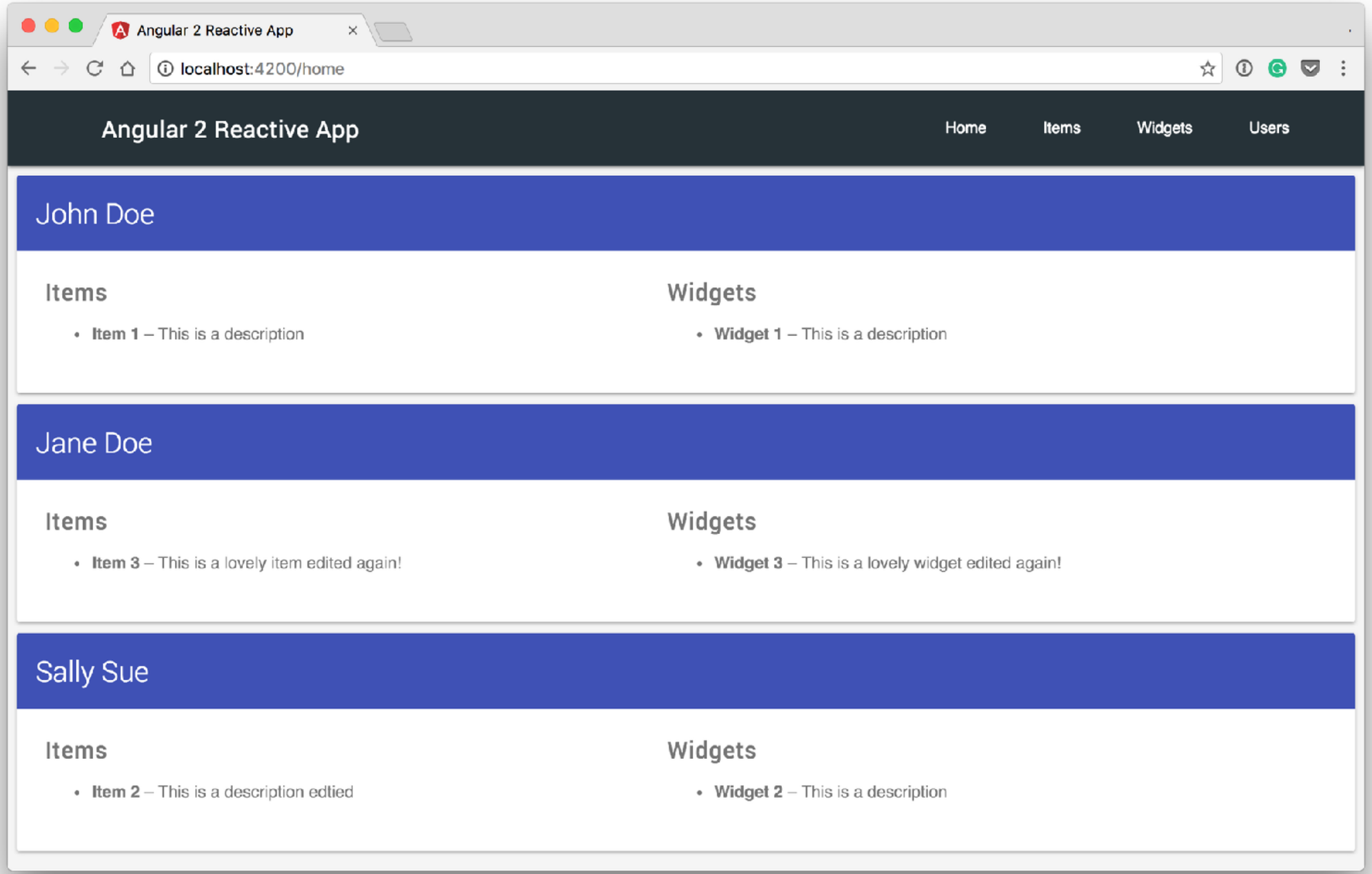
Strong grasp on how to **construct reactive** features in Angular 2

Agenda

- The Demo Application
- Why Reactive Angular?
- Redux Primer
- Immutable Operations
- The Observable Stream
- Reactive Data

Getting **Started**

https://github.com/onehungrymind/ng2-reactive-app


Angular 2 Reactive App
localhost:4200/home
Angular 2 Reactive App
Home
Items
Widgets
Users
John Doe
Items
Item 1 – This is a description
Widgets
Widget 1 – This is a description
Jane Doe
Items
Item 3 – This is a lovely item edited again!
Widgets
Widget 3 – This is a lovely widget edited again!
Sally Sue
Items
Item 2 – This is a description edtied
Widgets
Widget 2 – This is a description

The Demo Application

- 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**
- Each exercise has a **start** branch and a **solution** branch
- Feel free to use the existing **items** feature as a reference point
- Please explore! Don't be afraid to try new things!

Challenges

- Make sure you can run the application

Why **Reactive** Angular

Why Reactive Angular?

- What is Reactive?
- Why Reactive?
- Reactive Angular

In the context of this workshop, reactive programming is when we react to data being streamed to us over time

The biggest problem in the development and maintenance of large-scale software systems is complexity — large systems are hard to understand.

Out of the Tarpit - Ben Mosely Peter Marks

We believe that the major contributor to this complexity in many systems is the handling of state and the burden that this adds when trying to analyse and reason about the system. Other closely related contributors are code volume, and explicit concern with the flow of control through the system.

Out of the Tarpit - Ben Mosely Peter Marks

Complexity and **Purgatory**

```
class ItemsComponent {
  total: number = 0;
  currentCategory: string = 'cool';
  inbound(item) {
    const newTotal: number;
    switch(this.currentCategory) {
      case 'fun':
        // calculate total based on fun factor
        break;
      case 'cool':
        // calculate total based on cool factor
        break;
      case 'dangerous':
        // calculate total based on dangerous factor
        break;
      default:
        // do nothing at all
    }
    return newTotal;
  }
```

```
class ItemsComponent {
  total: number = 0;
  currentCategory: string = 'cool';
  inbound(item) {
    const newTotal: number;
    switch(this.currentCategory) {
      case 'fun':
        // calculate total based on fun factor
        break;
      case 'cool':
        // calculate total based on cool factor
        break;
      case 'dangerous':
        // calculate total based on dangerous factor
        break;
      default:
        // do nothing at all
    }
    return newTotal;
  }
```

```
const itemsComponents = new ItemsComponent();
const myItem = {name:'My Item'};
itemsComponents.inbound(myItem); // Some result

itemsComponents.currentCategory = 'fun'; // Changing state
itemsComponents.inbound(myItem); // Same parameter but different result

itemsComponents.currentCategory = 'cool'; // Changing state
itemsComponents.inbound(myItem); // Same parameter but different result

itemsComponents.currentCategory = 'dangerous'; // Changing state
itemsComponents.inbound(myItem); // Same parameter but different result
```

```
class ItemsComponent {
  total: number = 0;
  currentCategory: string = 'cool';
  currentAgeGroup: string = 'child';
  inbound(item) {
    const newTotal: number;
    switch(this.currentCategory) {
      //...
      case 'dangerous':
        if(this.currentAgeGroup !== 'child') {
          // calculate total based on dangerous factor
          this.currentCategory = 'dangerous';
        } else {
          // calculate total based on alternate dangerous factor
        }
        break;
      default:
      // do nothing at all
    }
    return newTotal;
  }
}
```

State
Management

```
class Inventory {
  ledger = { total: 1200 };
}
class ItemsComponent {
  ledger: any;
  constructor(private inventory:Inventory) {
    this.ledger = inventory.ledger;
  }
  add(x) { this.ledger.total += x; }
}
class WidgetsComponent {
  ledger: any;
  constructor(private inventory:Inventory) {
    this.ledger = inventory.ledger;
  }
  add(x) { this.ledger.total += x; }
}
```

```
class Inventory {
  ledger = { total: 1200 };
}
class ItemsComponent {
  ledger: any;
  constructor(private inventory:Inventory) {
    this.ledger = inventory.ledger;
  }
  add(x) { this.ledger.total += x; }
}
class WidgetsComponent {
  ledger: any;
  constructor(private inventory:Inventory) {
    this.ledger = inventory.ledger;
  }
  add(x) { this.ledger.total += x; }
}
```

Controlling
Flow

```
function doWork() {
  return $http.post('url')
    .then(function(response){
      if(response.data.success)
        return response.data;
      else
        return $q.reject('some error occurred');
    })
}
doWork().then(console.log, console.error);
```

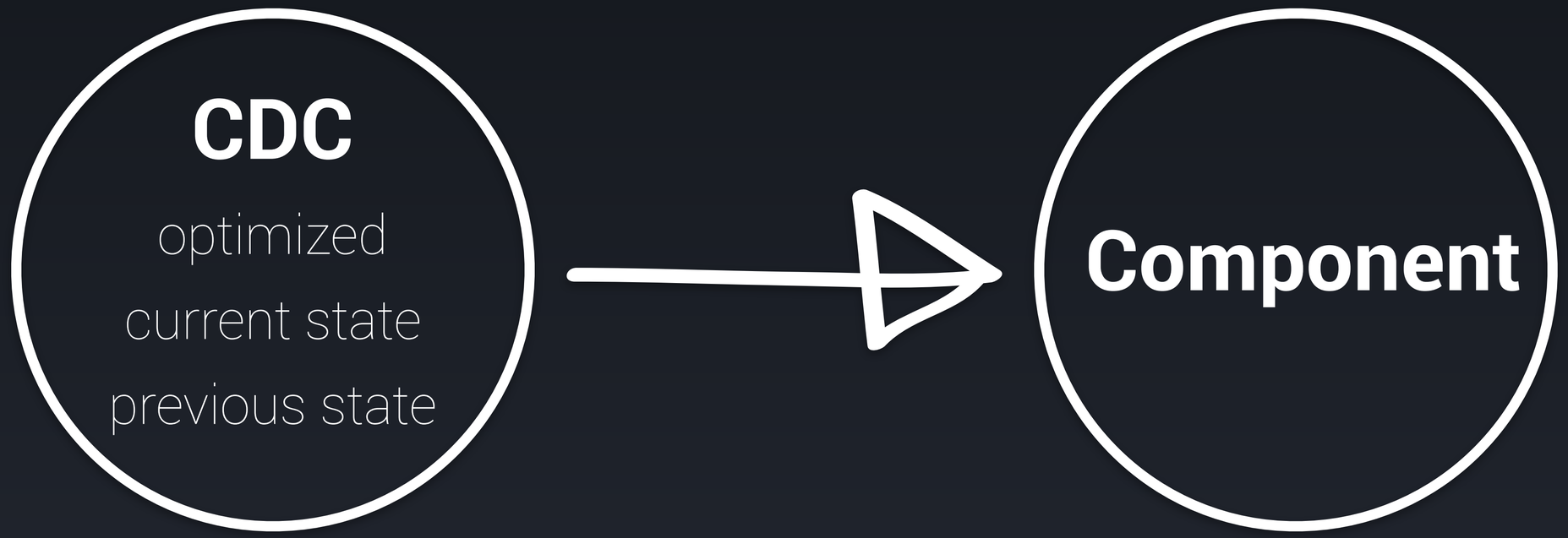
```
var retriesCount = 0;
function doWork() {
  return $http.post('url')
    .then(function(response){
      if(response.data.success)
        return response.data;
      else
        return $q.reject('some error occurred');
    })
    .then(null, function(reason){
      if(retriesCount++ < 3)
        return doWork();
      else
        return $q.reject(reason);
    });
}
doWork().then(console.log, console.error);
```

What do we **do**!?!?!

Reactive Angular

- Angular 2 is ships with mechanisms to help us effectively manage state, control flow and reduce code volume
- The Axis of Awesome consists of better change detection, observable support and the async pipe
- By leveraging observables and redux via @ngrx/store, we can significantly reduce complexity in our applications

Zone.js

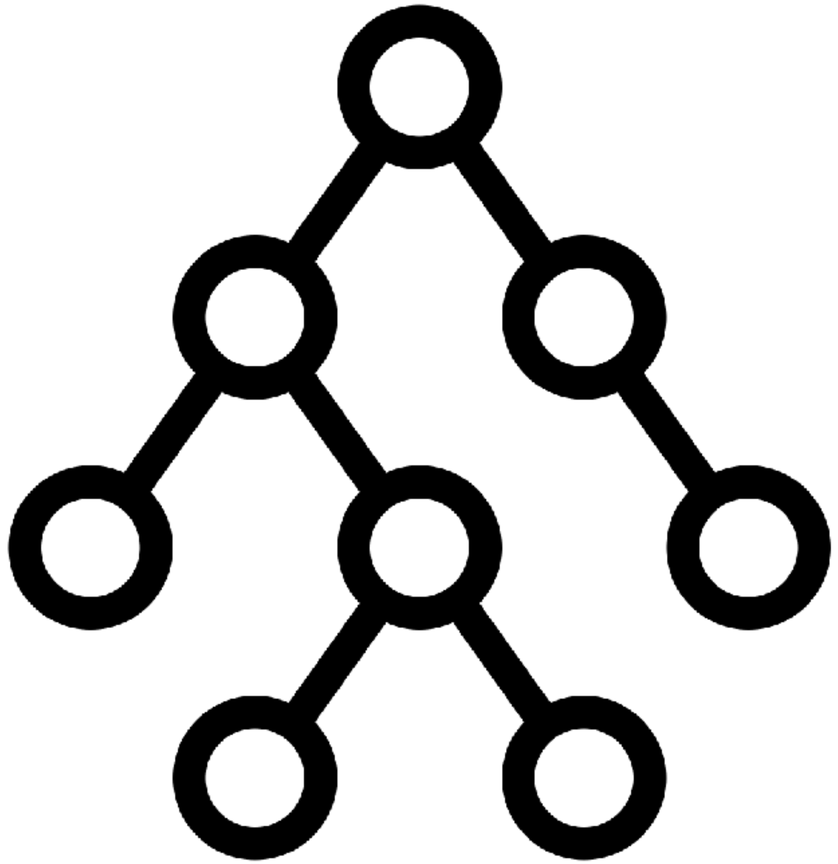


Change Detection Classes

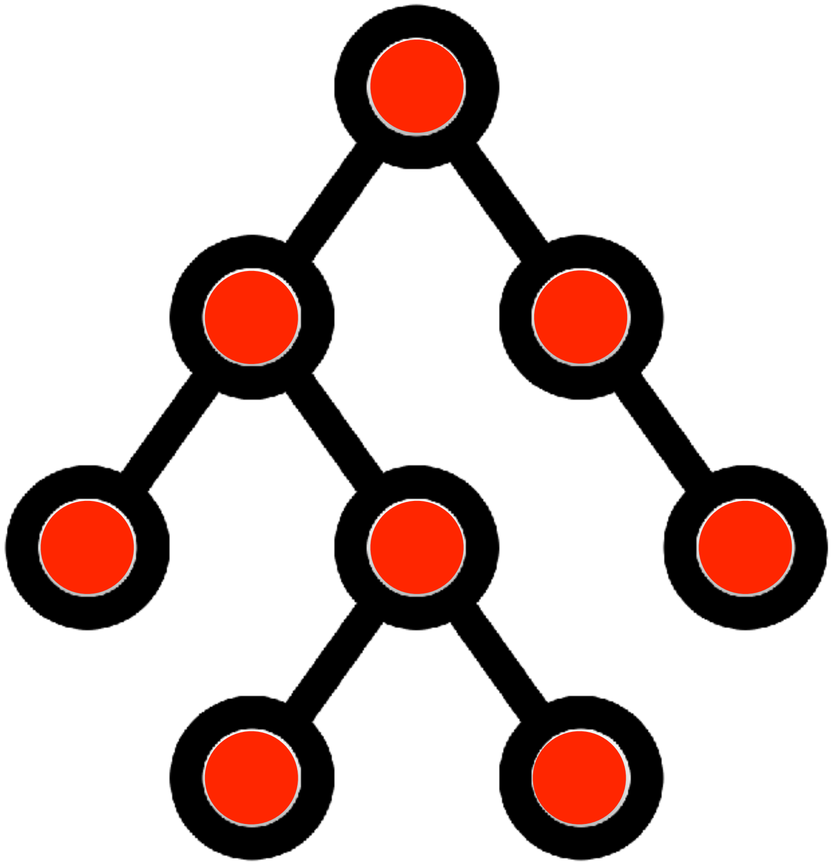
3 -10x Faster

Change Detection Strategy

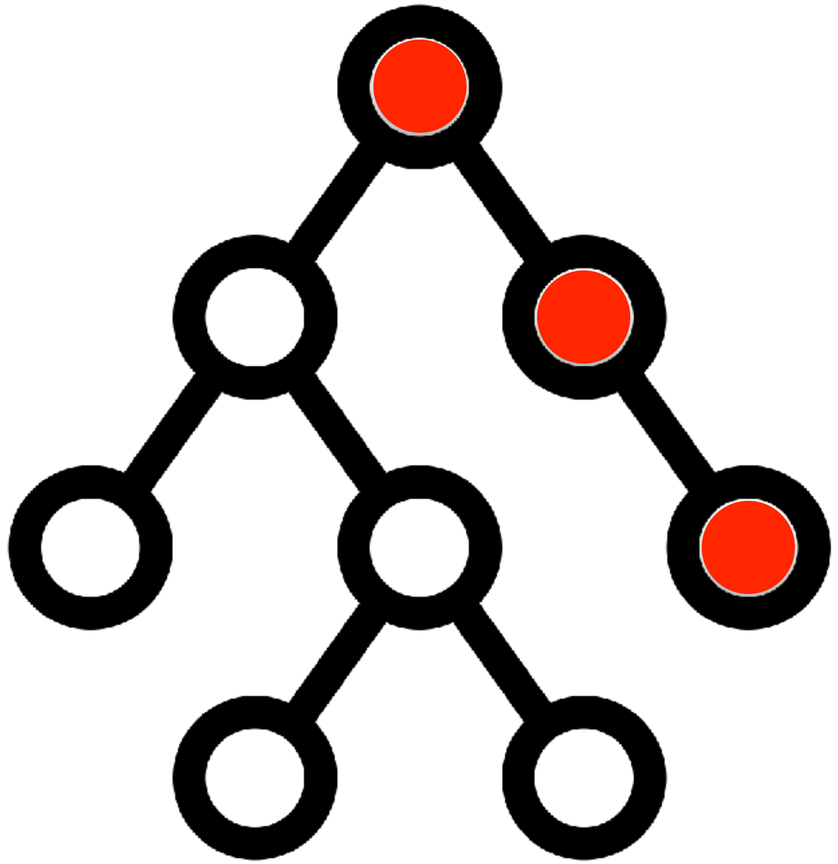
- We can control how Angular will respond when it detects changes within the application
- By default, Angular will always detect changes and respond on all nodes
- We can set the change detection strategy to onPush meaning that Angular will only check for change
- We are essentially turning off change detection for a component branch in our application which has serious performance implications



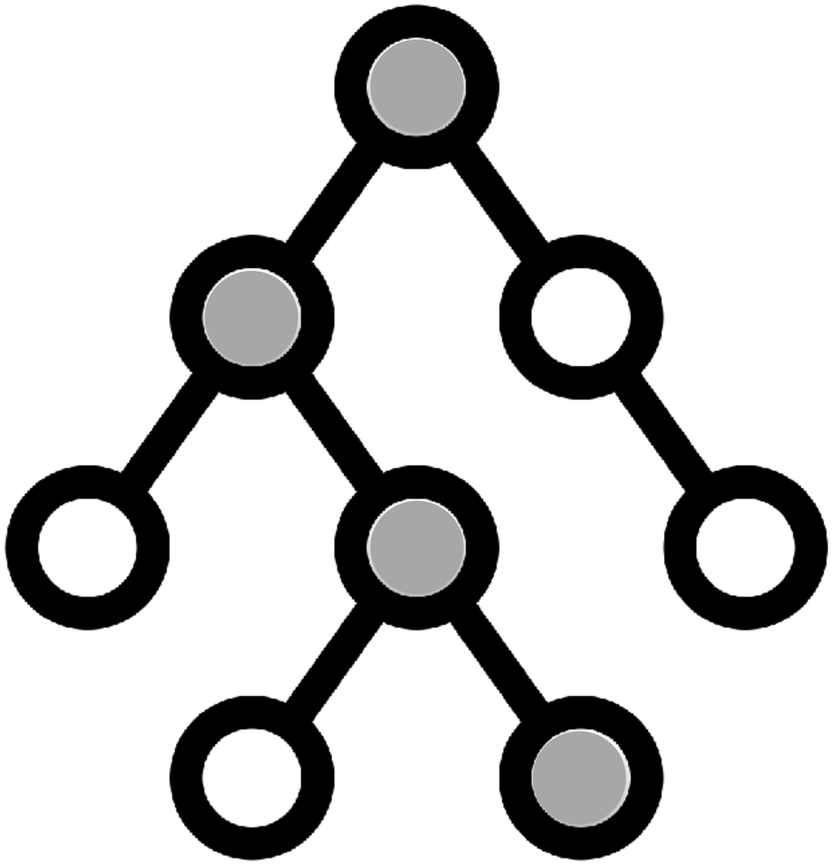
Detecting Change



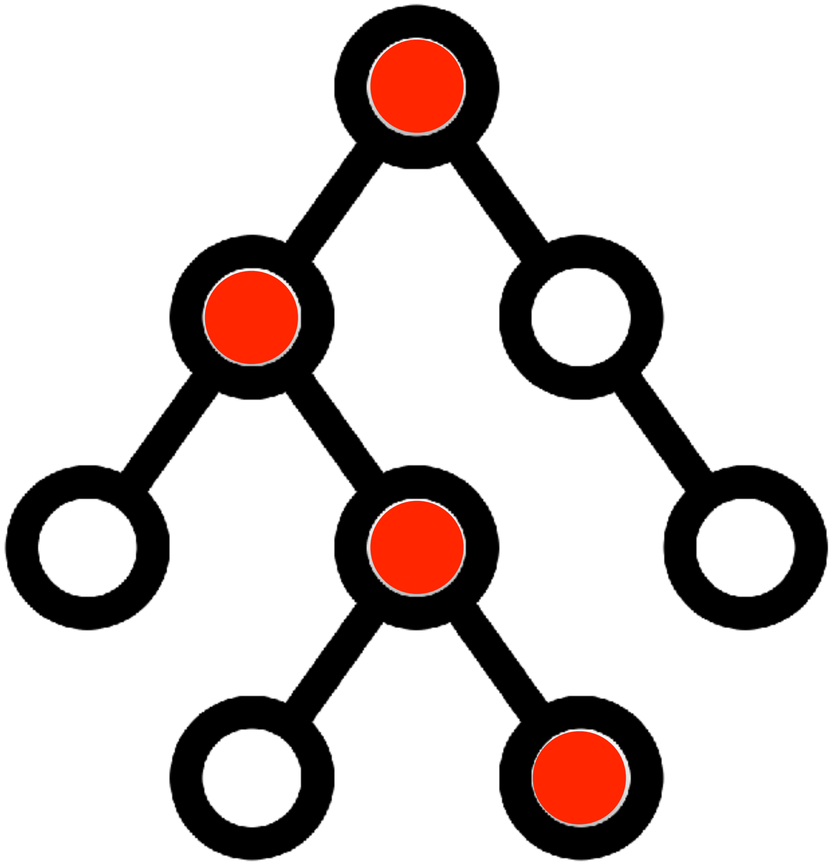
Default Change Detection



OnPush Change Detection



Observables



Observables

```
@Component({
  selector: 'app-widgets',
  templateUrl: './widgets.component.html',
  styleUrls: ['./widgets.component.css'],
  changeDetection: ChangeDetectionStrategy.OnPush
})
export class WidgetsComponent implements OnInit { }
```

ChangeDetectionStrategy.onPush

```
Observable.fromEvent(document, 'click')
  .map(event => 100)
  .startWith(5)
  .subscribe(coolness => {
    this.coolness = coolness;
  });
```

Will Not Update

```
constructor(private cd: ChangeDetectorRef) {}

ngOnInit() {
  Observable.fromEvent(document, 'click')
    .map(event => 100)
    .startWith(5)
    .subscribe(coolness => {
      this.coolness = coolness;
      this.cd.detectChanges();
    });
}
```

Will Update

Async Pipe

The async pipe allows us to bind directly to an asynchronous primitive in our template
The async pipe will automatically unwrap the value for us
The async pipe will automatically unsubscribe from the observable when the DOM element is removed

```
<app-items-list [items]="items$ | async"
                (selected)="selectItem($event)"
                (deleted)="deleteItem($event)">
</app-items-list>
```

async pipe

Challenges

- Take a few minutes and explore the **master** branch of the sample application
- Where are we using observables?
- Where are we leveraging a change detection strategy?
- Where are we using the async pipe?

Redux Primer

Redux Primer

- What is Redux
- Reducers
- Actions
- Application Store
- store.subscribe
- store.dispatch

Required Viewing

Getting Started with Redux by Dan Abramov

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

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

Enter Redux

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

Single State Tree

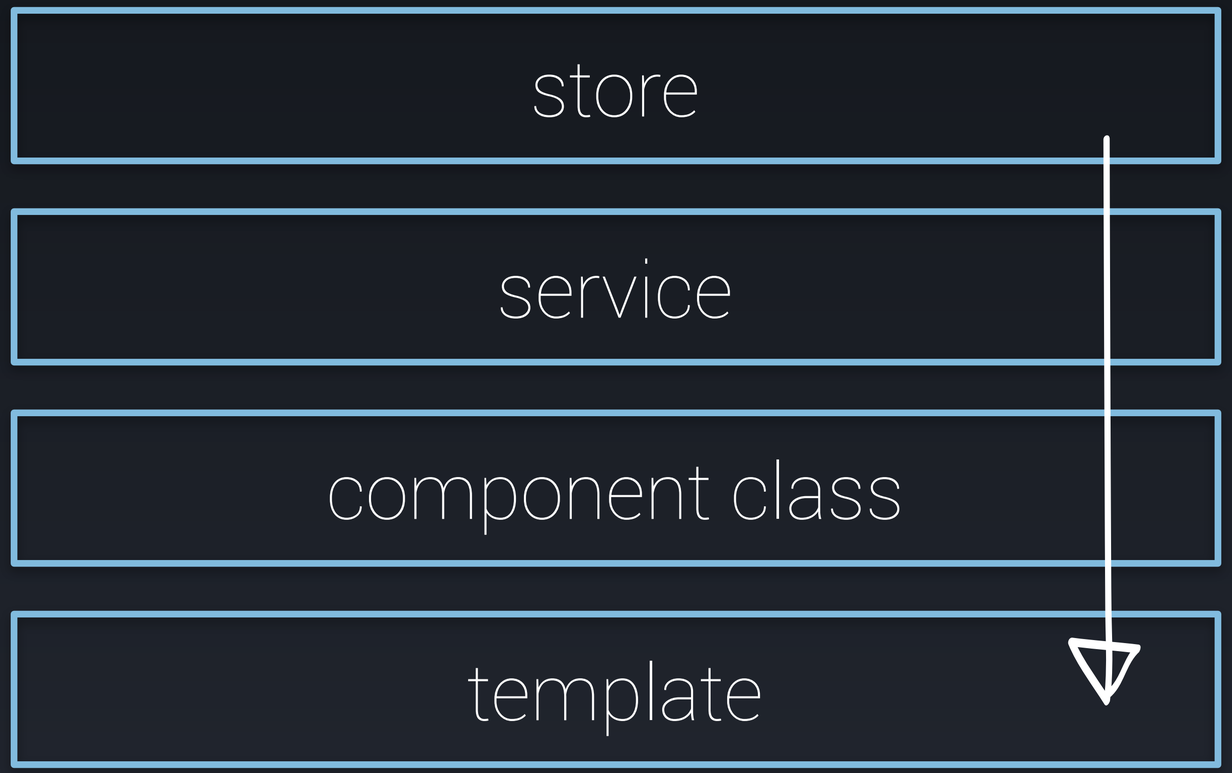
```ts
import { Item, User, Widget } from './shared';

export interface AppStore {
  items: Item[],
  users: User[],
  widgets: Widget[]
}
```

AppStore

```
@NgModule({
  declarations: [ ],
  imports: [
    BrowserModule,
    FormsModule,
    HttpModule,
    Ng2RestAppRoutingModule,
    StoreModule.provideStore({ items, users, widgets })
  ],
  providers: [ItemsService, UsersService, WidgetsService, HomeService],
  bootstrap: [AppComponent]
})
```

StoreModule.provideStore

State Flows Down



```
export declare class Store<T> extends Observable<T> implements Observer<Action> {
    private _dispatcher;
    private _reducer;
    constructor(_dispatcher: Observer<Action>,
      _reducer: Observer<ActionReducer<any>>,
      state$: Observable<any>);
    select: SelectSignature<T>;
    lift<R>(operator: Operator<T, R>): Store<R>;
    replaceReducer(reducer: ActionReducer<any>): void;
    dispatch(action: Action): void;
    next(action: Action): void;
    error(err: any): void;
    complete(): void;
}
```

Store

```ts
export declare class Store<T> extends Observable<T> implements Observer<Action> {
    private _dispatcher;
    private _reducer;
    constructor(_dispatcher: Observer<Action>,
      _reducer: Observer<ActionReducer<any>>,
      state$: Observable<any>);
    select: SelectSignature<T>;
    lift<R>(operator: Operator<T, R>): Store<R>;
    replaceReducer(reducer: ActionReducer<any>): void;
    dispatch(action: Action): void;
    next(action: Action): void;
    error(err: any): void;
    complete(): void;
}
```

Store

```
export class ItemsService {
  items$: Observable<Item[]> = this.store.select('items');

  constructor(
    private http: Http,
    private store: Store<AppStore>
  ) {}

  loadItems() {
    const items: Item[] = [
      {
        'id': 1,
        'name': 'Item 1',
        'description': 'This is a description',
        'user': 1
      }
    ]
    this.store.dispatch({ type: ADD_ITEMS, payload: items });
  }
}
```

this.store

```
export class ItemsService {
  items$: Observable<Item[]> = this.store.select('items');

  constructor(
    private http: Http,
    private store: Store<AppStore>
  ) {}

  loadItems() {
    const items: Item[] = [
      {
        'id': 1,
        'name': 'Item 1',
        'description': 'This is a description',
        'user': 1
      }
    ]
    this.store.dispatch({ type: ADD_ITEMS, payload: items });
  }
}
```

store.select

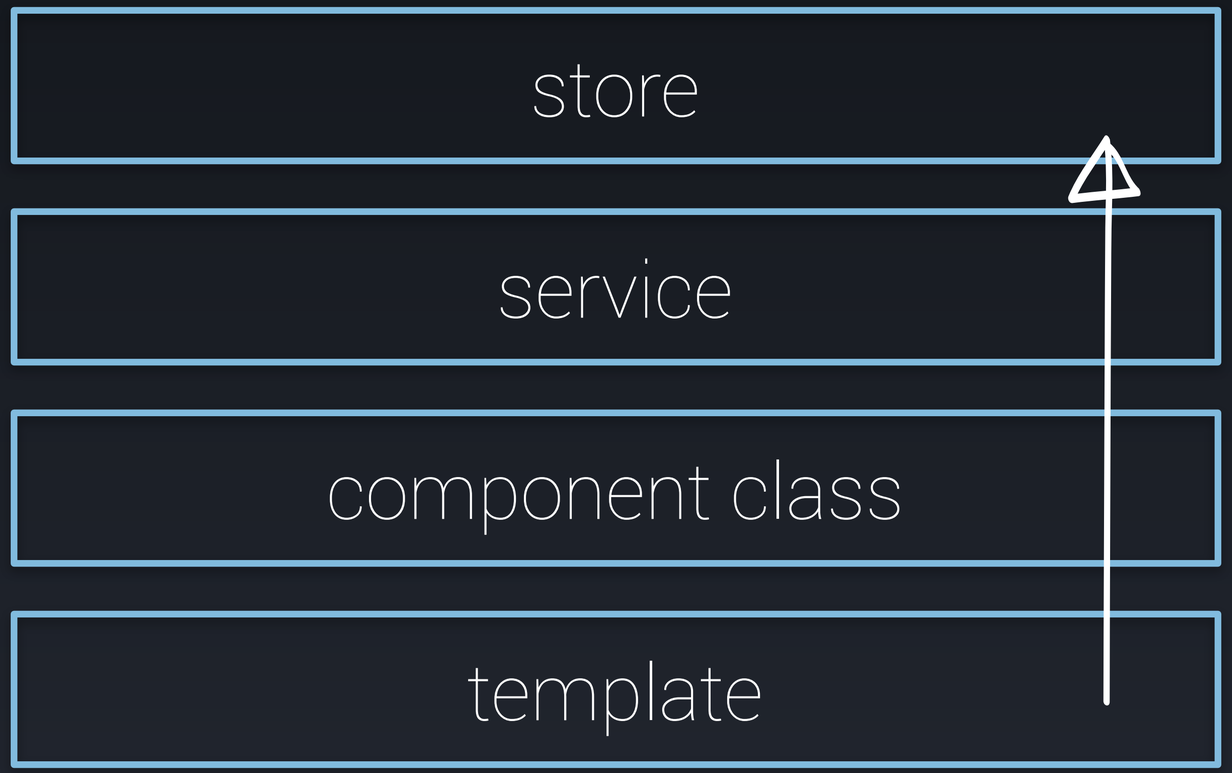
```
export class ItemsComponent implements OnInit {
  items$: Observable<Item[]>;
  selectedItem: Item;

  constructor(
    private itemsService: ItemsService
  ) {}

  ngOnInit() {
    this.items$ = this.itemsService.items$;
    this.itemsService.loadItems();
  }
}
```

Data Consumption

```
<app-items-list [items]="items$ | async"
                (selected)="selectItem($event)"
                (deleted)="deleteItem($event)">
</app-items-list>
```

Data Consumption

Events Flow Up



```ts
export interface ActionReducer<T> {
    (state: T, action: Action): T;
}

export interface Action {
    type: string;
    payload?: any;
}
```

Reducer

```
export const items = (state = [], action) => {
  switch (action.type) {
    case ADD_ITEMS:
      return action.payload;
    default:
      return state;
  }
};
```

Reducer

```
export const ADD_ITEMS = 'ADD_ITEMS';
export const CREATE_ITEM = 'CREATE_ITEM';
export const UPDATE_ITEM = 'UPDATE_ITEM';
export const DELETE_ITEM = 'DELETE_ITEM';

export const items: ActionReducer<Item[]> = (state: Item[] = [], action: Action) => {
  switch (action.type) {
    case ADD_ITEMS:
    case CREATE_ITEM:
    case UPDATE_ITEM:
    case DELETE_ITEM:
    default:
      return state;
  }
};
```

action.type

```
export const items = (state = [], action) => {
  switch (action.type) {
    case ADD_ITEMS:
      return action.payload;
    default:
      return state;
  }
};
```

action.payload

```
export const items: ActionReducer<Item[]> = (state: Item[] = [], action: Action) => {
  switch (action.type) {
    case ADD_ITEMS:
      return action.payload;
    case CREATE_ITEM:
      return [...state, action.payload];
    case UPDATE_ITEM:
      return state.map(item => {
        return item[comparator] === action.payload[comparator]
          ? Object.assign({}, item, action.payload) : item;
      });
    case DELETE_ITEM:
      return state.filter(item => {
        return item[comparator] !== action.payload[comparator];
      });
    default:
      return state;
  }
};
```

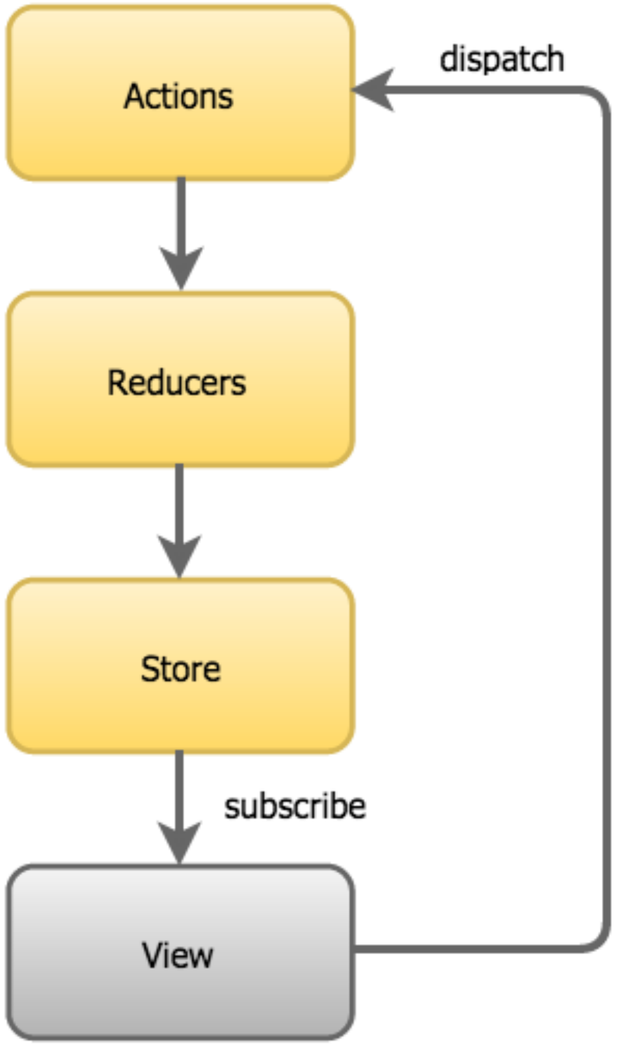
Items Reducer

```
export class ItemsService {
  items$: Observable<Item[]> = this.store.select('items');

  constructor(
    private http: Http,
    private store: Store<AppStore>
  ) {}

  loadItems() {
    const items: Item[] = [
      {
        'id': 1,
        'name': 'Item 1',
        'description': 'This is a description',
        'user': 1
      }
    ]
    this.store.dispatch({ type: ADD_ITEMS, payload: items });
  }
}
```

store.dispatch

```
createItem(item: Item) {
  this.store.dispatch({ type: CREATE_ITEM, payload: item });
}

updateItem(item: Item) {
  this.store.dispatch({ type: UPDATE_ITEM, payload: item });
}

deleteItem(item: Item) {
  this.store.dispatch({ type: DELETE_ITEM, payload: item });
}
```

store.dispatch


Actions
dispatch
Reducers
Store
subscribe
View

Challenges

- Checkout out the **01-redux-primer-start** branch
- Create a **widgets.reducer.ts** file
- Create a **widgets** reducer with an **ADD_WIDGETS** action
- Update the **app.module.ts** file to initialize the the application store
- Update **widgets.service.ts** to use get the widgets collection from the application store

HINT: Use the Items feature as a reference

Immutable Operations

Immutable Operations

- Object.freeze
- Immutable Add
- Immutable Update
- Immutable Delete

```
case CREATE_WIDGET:
  state.push(action.payload);
  return state;
```

Mutable!

```
case UPDATE_WIDGET:
  state.forEach((widget, index) => {
    if (widget[comparator] === action.payload[comparator]) {
      state.splice(index, 1, action.payload);
    }
  });
  return state;
```

Mutable!

```
case DELETE_WIDGET:
  state.forEach((widget, index) => {
    if (widget[comparator] === action.payload[comparator]) {
      state.splice(index, 1);
    }
  });
  return state;
```

Mutable!

```
case CREATE_WIDGET:
  Object.freeze(state);
  state.push(action.payload);
  return state;
```

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.**

Object.freeze

```
case CREATE_ITEM:
  return [...state, action.payload];

case CREATE_ITEM:
  return state.concat(action.payload);
```

The concat() method is used to merge two or more arrays. This method does not change the existing arrays, but instead returns a new array. - MDN

Immutable!

```
case UPDATE_ITEM:
  return state.map(item => {
    return item[comparator] === action.payload[comparator]
      ? Object.assign({}, item, action.payload) : item;
  });
```

The map() method creates a new array with the results of calling a provided function on every element in this array.

The Object.assign() method is used to copy the values of all enumerable own properties from one or more source objects to a target object. It will return the target object.

Immutable!

```
case DELETE_ITEM:
  return state.filter(item => {
    return item[comparator] !== action.payload[comparator];
  });
```

The filter() method creates a new array with all elements that pass the test implemented by the provided function.

Immutable!

Challenges

- Checkout out the **02-immutable-operations-start** branch
- Locate the **widgets.reducer.ts** file
- Replace the **mutable** operations with **immutable** equivalents
- Verify that these operations are immutable using **Object.freeze**

The **Observable** Stream

https://egghead.io/courses/step-by-step-async-javascript-with-rxjs

https://egghead.io/courses/introduction-to-reactive-programming

The Observable Stream

- The Basic Sequence
- Initial Output
- Final Input
- Observable Operators
- Observable.map

State **management**

Controlling flow

Code **volume**

Enter **Observables**

```
return this.http.get(this.URLS.FETCH)
  .map(res => res.json())
  .toPromise();
```

Problem solved!

Observables give us a powerful way to **encapsulate**, **transport** and **transform** data from user interactions to create powerful and immersive experiences.

Encapsulate

Transport

Transform

Encapsulate

Transport

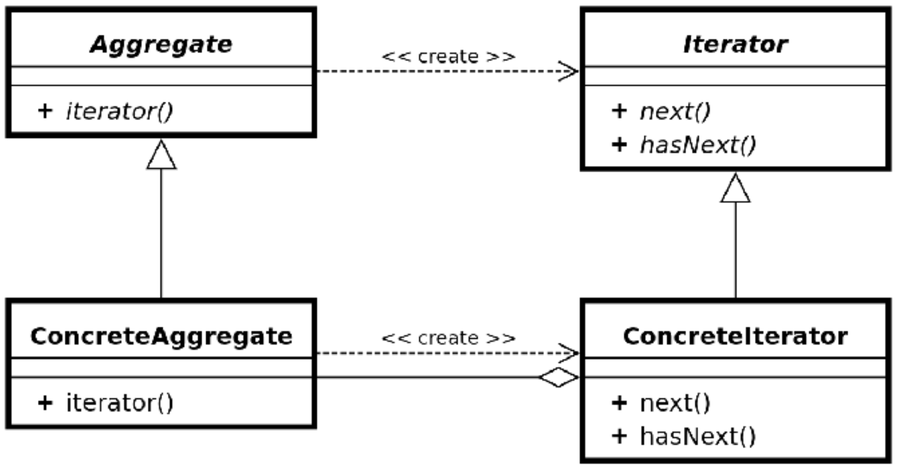
Transform

Encapsulate

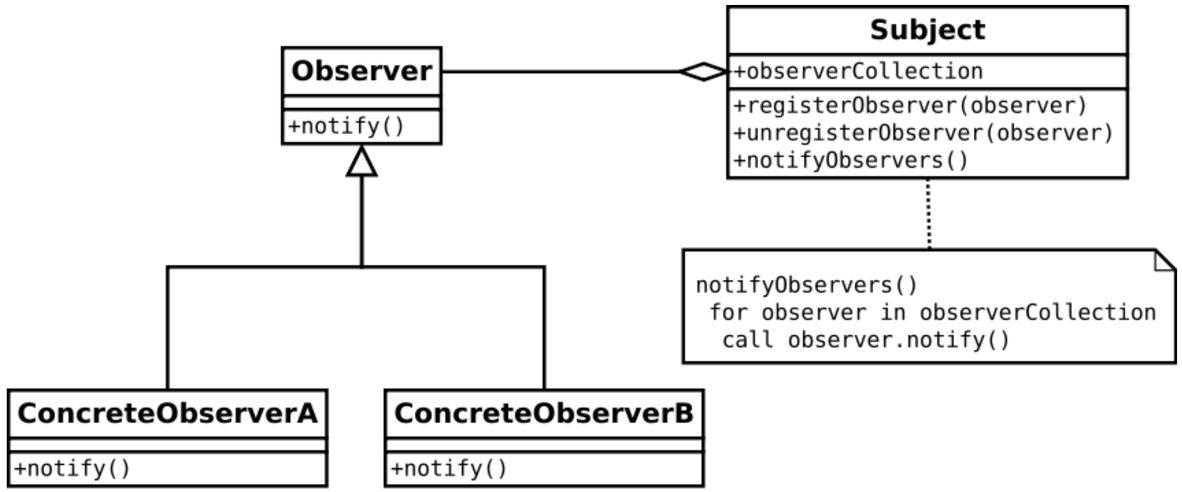
Transport

Transform

Iterator Pattern



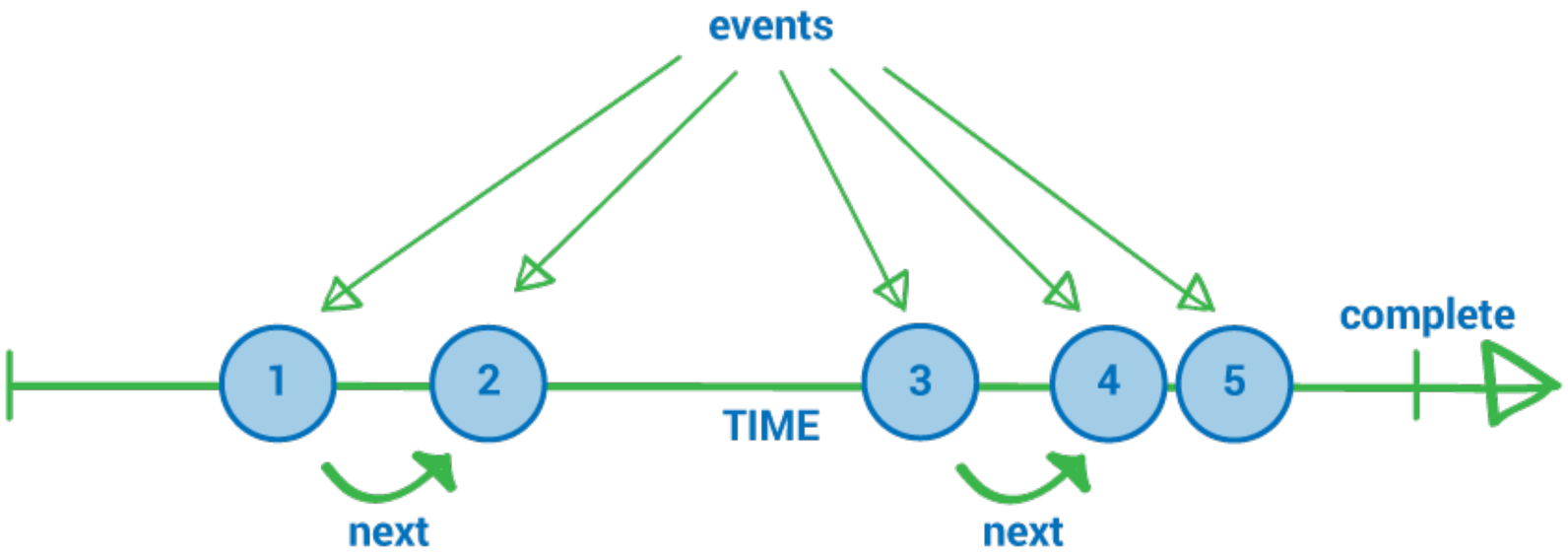
Observer Pattern



State

Communication

Communicate
state over **time**



Observable stream

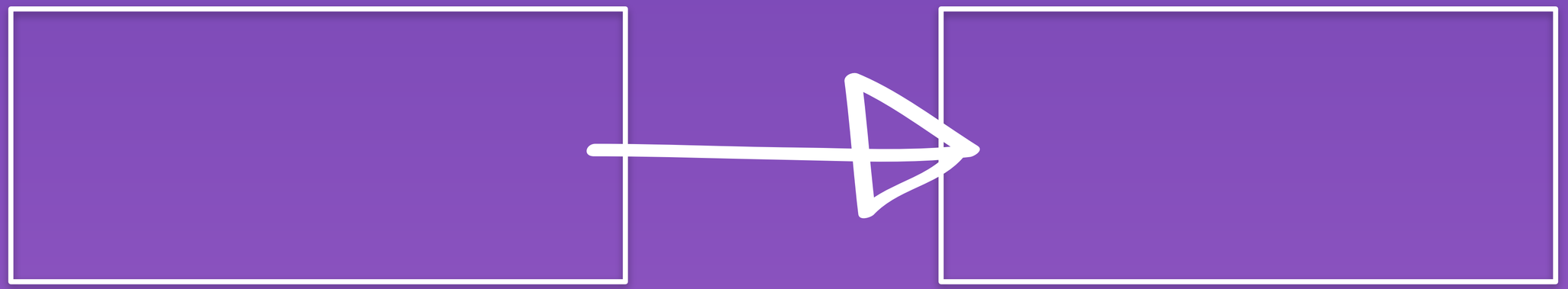
	SINGLE	MULTIPLE
SYNCHRONOUS	Function	Enumerable
ASYNCHRONOUS	Promise	**Observable**

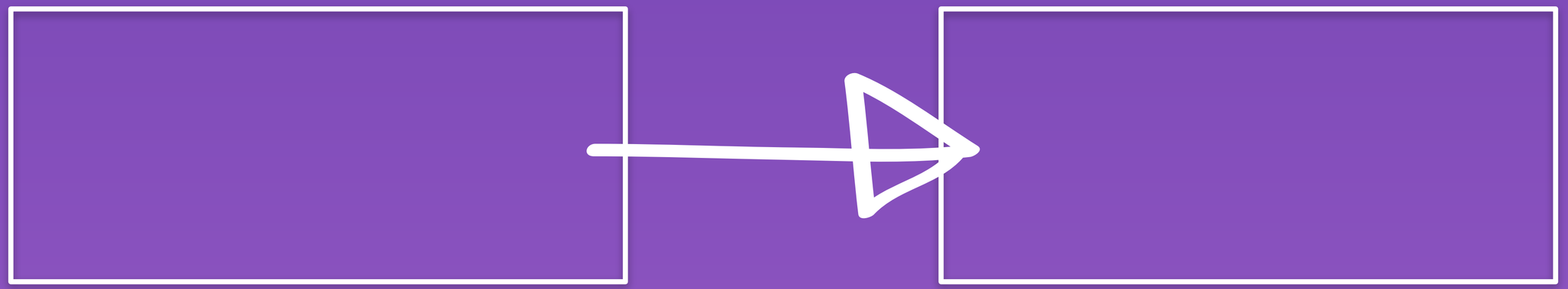
Values over time

	SINGLE	MULTIPLE
PULL	Function	Enumerable
PUSH	Promise	**Observable**

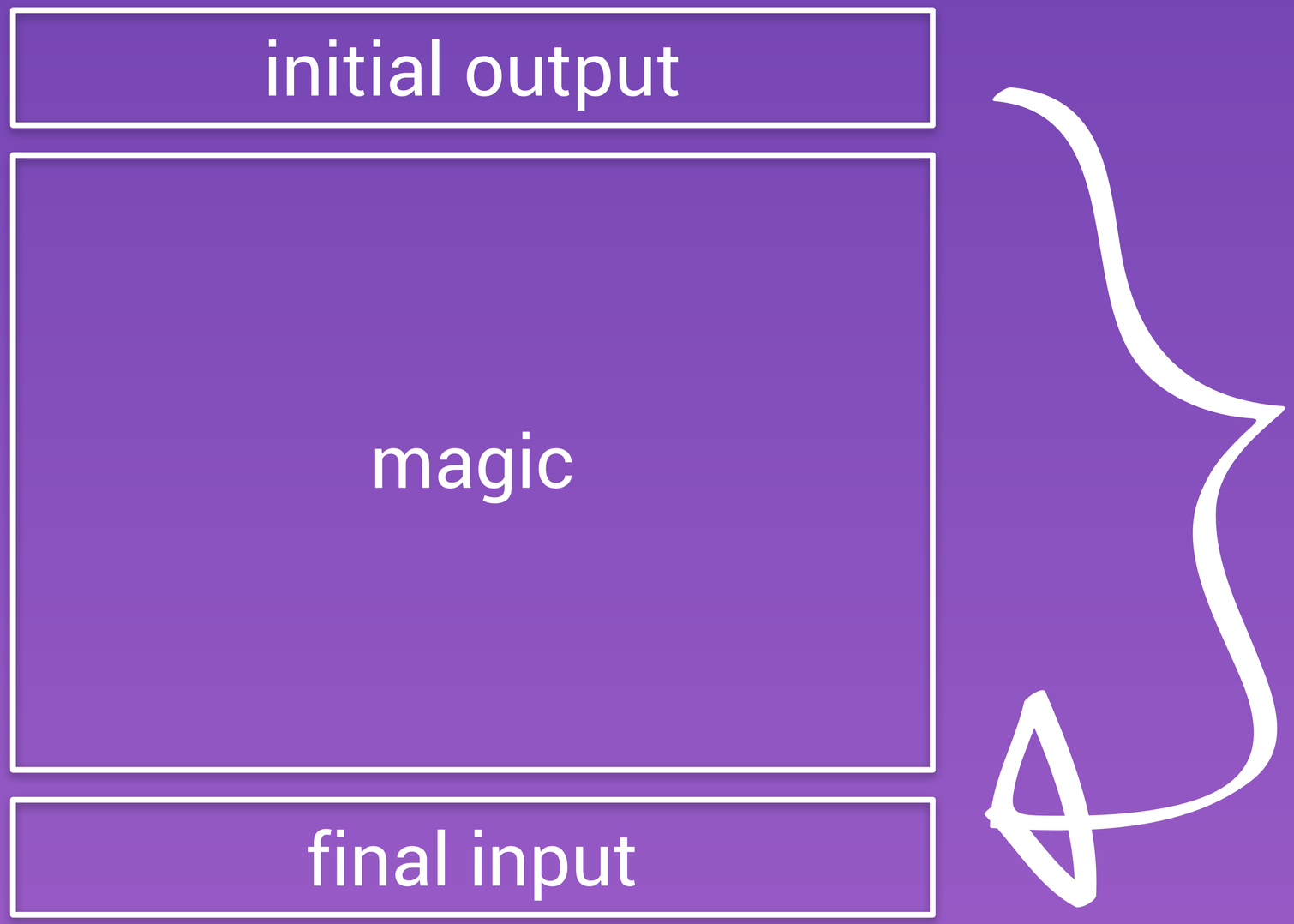
Value consumption

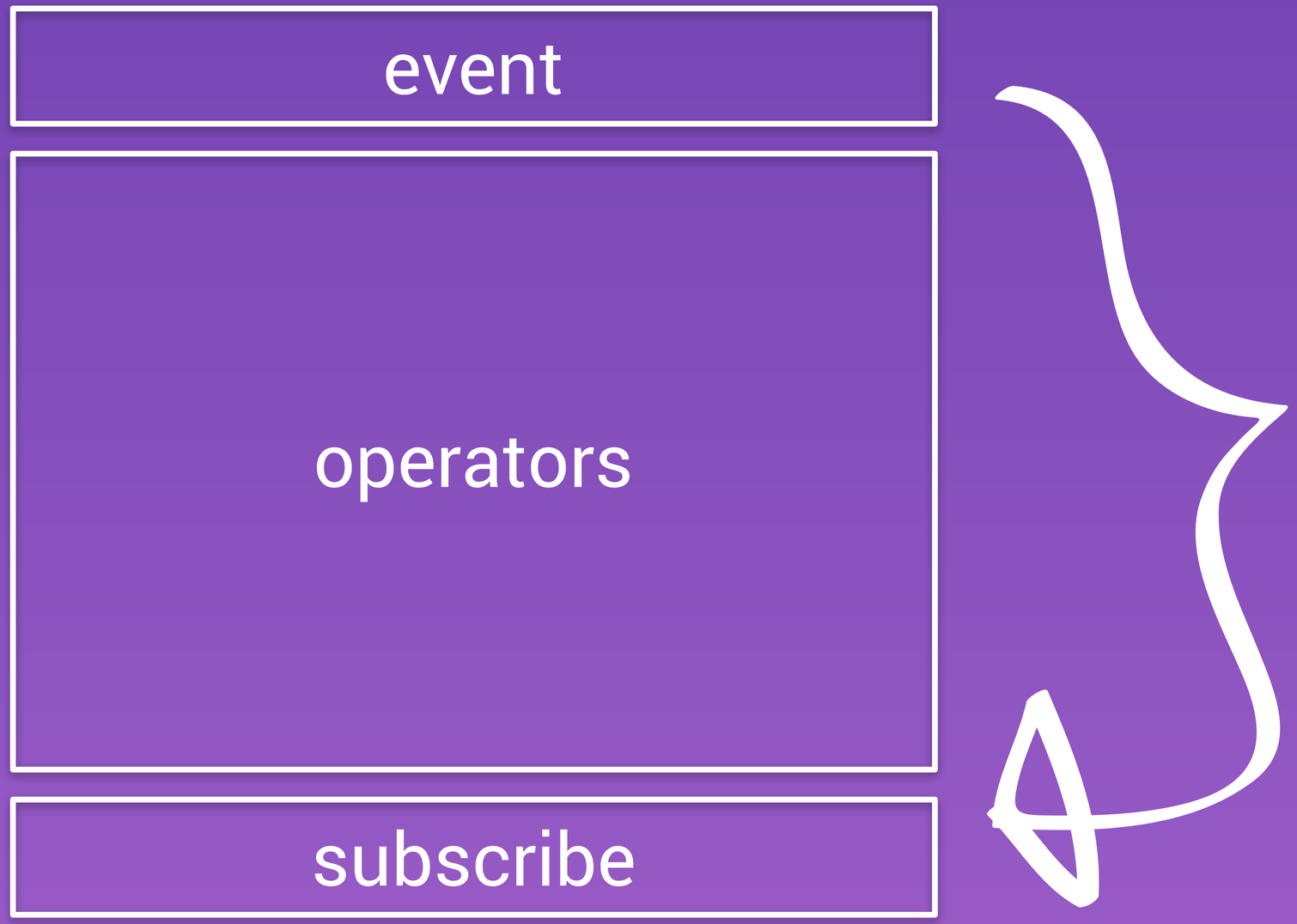
But observables are **hard!!!**


input
output


output
input

The Basic **Sequence**


initial output
magic
final input


event
operators
subscribe

```
@ViewChild('btn') btn;
message: string;

ngOnInit() {
  Observable.fromEvent(this.getNativeElement(this.btn), 'click')
    .subscribe(result => this.message = 'Beast Mode Activated!');
}

getNativeElement(element) {
  return element._elementRef.nativeElement;
}
```

```
@ViewChild('btn') btn;
message: string;

ngOnInit() {
  Observable.fromEvent(this.getNativeElement(this.btn), 'click')
    .subscribe(result => this.message = 'Beast Mode Activated!');
}

getNativeElement(element) {
  return element._elementRef.nativeElement;
}
```

Initial output

```
@ViewChild('btn') btn;
message: string;

ngOnInit() {
  Observable.fromEvent(this.getNativeElement(this.btn), 'click')
    .subscribe(event => this.message = 'Beast Mode Activated!');
}

getNativeElement(element) {
  return element._elementRef.nativeElement;
}
```

Final input

```
@ViewChild('btn') btn;
message: string;

ngOnInit() {
  Observable.fromEvent(this.getNativeElement(this.btn), 'click')
    .map(event => 'Beast Mode Activated!')
    .subscribe(result => this.message = result);
}

getNativeElement(element) {
  return element._elementRef.nativeElement;
}
```

Everything in between

```
@ViewChild('btn') btn;
message: string;

ngOnInit() {
  Observable.fromEvent(this.getNativeElement(this.btn), 'click')
    .filter(event => event.shiftKey)
    .map(event => 'Beast Mode Activated!')
    .subscribe(result => this.message = result);
}

getNativeElement(element) {
  return element._elementRef.nativeElement;
}
```

Everything in between

How do we
preserve state
in a stream?

```
<button #right>Right</button>
<div class="container">
   <div #ball class="ball"
      [style.left]="position.x + 'px'"
      [style.top]="position.y + 'px'">
   </div>
</div>
```

```
@ViewChild('right') right;
position: any;

ngOnInit() {
  Observable
    .fromEvent(this.getNativeElement(this.right), 'click')
    .map(event => 10)
    .startWith({x: 100, y: 100})
    .scan((acc, curr) => { return { y: acc.y, x: acc.x + curr}})
    .subscribe(result => this.position = result);
}
```

```
@ViewChild('right') right;
position: any;

ngOnInit() {
  Observable
    .fromEvent(this.getNativeElement(this.right), 'click')
    .map(event => 10)
    .startWith({x: 100, y: 100})
    .scan((acc, curr) => { return { y: acc.y, x: acc.x + curr}})
    .subscribe(result => this.position = result);
}
```

```
@ViewChild('right') right;
position: any;

ngOnInit() {
  Observable
    .fromEvent(this.getNativeElement(this.right), 'click')
    .map(event => 10)
    .startWith({x: 100, y: 100})
    .scan((acc, curr) => { return { y: acc.y, x: acc.x + curr}})
    .subscribe(result => this.position = result);
}
```

```
@ViewChild('right') right;
position: any;

ngOnInit() {
  Observable
    .fromEvent(this.getNativeElement(this.right), 'click')
    .map(event => 10)
    .startWith({x: 100, y: 100})
    .scan((acc, curr) => { return { y: acc.y, x: acc.x + curr}})
    .subscribe(result => this.position = result);
}
```

What if we have
more than one stream?

```
@ViewChild('left') left;
@ViewChild('right') right;
position: any;

ngOnInit() {
  const left$ = Observable.fromEvent(this.getNativeElement(this.left), 'click')
    .map(event => -10);

  const right$ = Observable.fromEvent(this.getNativeElement(this.right), 'click')
    .map(event => 10);

  Observable.merge(left$, right$)
    .startWith({x: 100, y: 100})
    .scan((acc, curr) => { return { y: acc.y, x: acc.x + curr}})
    .subscribe(result => this.position = result);
}
```

```
@ViewChild('left') left;
@ViewChild('right') right;
position: any;

ngOnInit() {
  const left$ = Observable.fromEvent(this.getNativeElement(this.left), 'click')
    .map(event => -10);

  const right$ = Observable.fromEvent(this.getNativeElement(this.right), 'click')
    .map(event => 10);

  Observable.merge(left$, right$)
    .startWith({x: 100, y: 100})
    .scan((acc, curr) => { return { y: acc.y, x: acc.x + curr}})
    .subscribe(result => this.position = result);
}
```

Challenges

Use these four operators to make a clicker that increments and decrements a variable

```
import 'rxjs/add/observable/fromEvent';
import 'rxjs/add/observable/merge';
import 'rxjs/add/operator/startWith';
import 'rxjs/add/operator/scan';
```

Challenges

- Checkout out the **03-observable-stream-start** branch
- Locate the **counter.component.ts** file
- Create an observable stream from a click event using **Observable.fromEvent** as your initial output
- Capture the input of the stream using **Observable.subscribe**

Challenges

- Map the event using **Observable.map**
- Use **Observable.startWith** to start the stream with an initial value
- Use **Observable.scan** to store the state of the stream

Challenges

- Extract the increment stream to a stand alone stream called **increment$**
- Model a **decrement$** stream after the **increment$** stream
- Use **Observable.merge** to combine the **increment$** and **decrement$** stream

Reactive Data

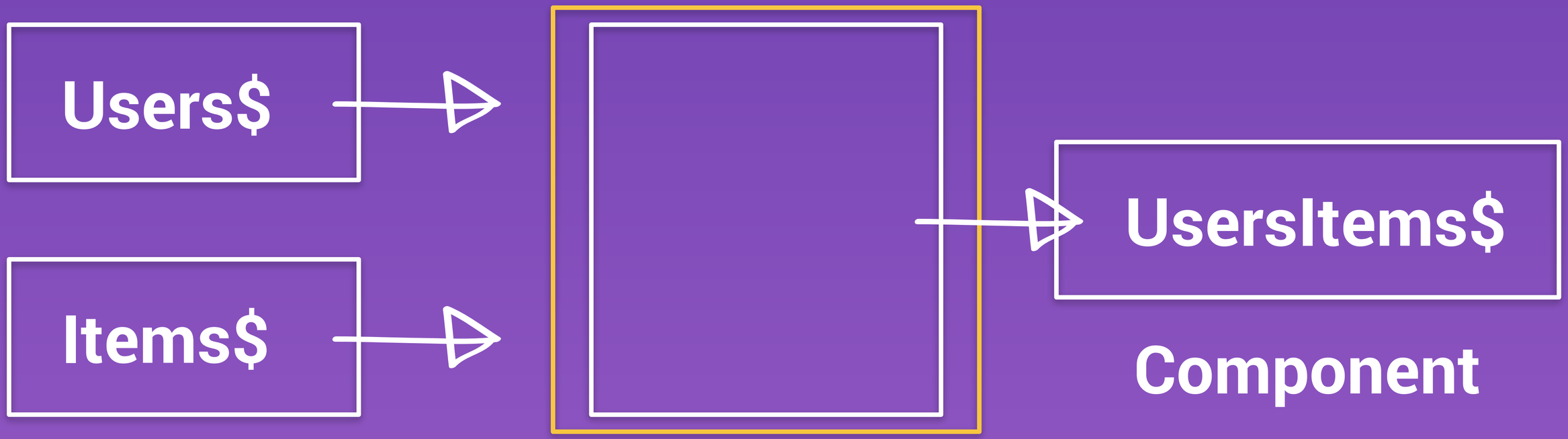
Reactive Data

- Handling Async Operations
- Handling Multiple Models


Event
Async
Operation
Reducer

```
export class ItemsService {
  items$: Observable<Item[]> = this.store.select('items');

  constructor(
    private http: Http,
    private store: Store<AppStore>
  ) {}

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

Async Operations


Users$
Items$
UsersItems$
Component
Observable
.combineLatest

```ts
export interface UserData {
  name: string;
  items: Item[];
  widgets: Widget[];
}
```

Interface

```
export class HomeService {
  items$: Observable<Item[]> = this.itemsService.items$;
  users$: Observable<User[]> = this.usersService.users$;
  widgets$: Observable<Widget[]> = this.widgetsService.widgets$;

  constructor(
    private usersService: UsersService,
    private itemsService: ItemsService,
    private widgetsService: WidgetsService
  ) {
    this.usersService.loadUsers();
    this.itemsService.loadItems();
    this.widgetsService.loadWidgets();
  }
}
```

Individual Streams

```
data$: Observable<UserData[]> = Observable.combineLatest(
  this.users$, this.items$, this.widgets$,
  (users, items, widgets) => {
    return users.map(user => {
      return Object.assign({}, {
        name: user.name,
        items: items.filter(item => item.user === user.id),
        widgets: widgets.filter(widget => widget.user === user.id)
      });
    });
});
```

Observable.combineLatest

```ts
export class HomeComponent {
  data$: Observable<UserData[]> = this.homeService.data$;

  constructor(private homeService: HomeService) { }
}
```

data$

```
<div *ngFor="let userData of data$ | async"></div>
```

async

Challenges

- Checkout out the **05-reactive-data-start** branch
- Locate the **home.service.ts** file
- Consume the **users** collection from the **users** store
- Consume the **items** collection from the **items** store
- Consume the **widgets** collection from the **widgets** store
- Use **Observable.combineLatest** to combine both collections to show the **items** for each **user** and expose it as a **data$** stream
- Display the **data$** stream in the **home** template

@simpulton

Thanks!