



Introduction State management

Building a custom RxJS Store



Peter Kassenaar
info@kassenaar.com



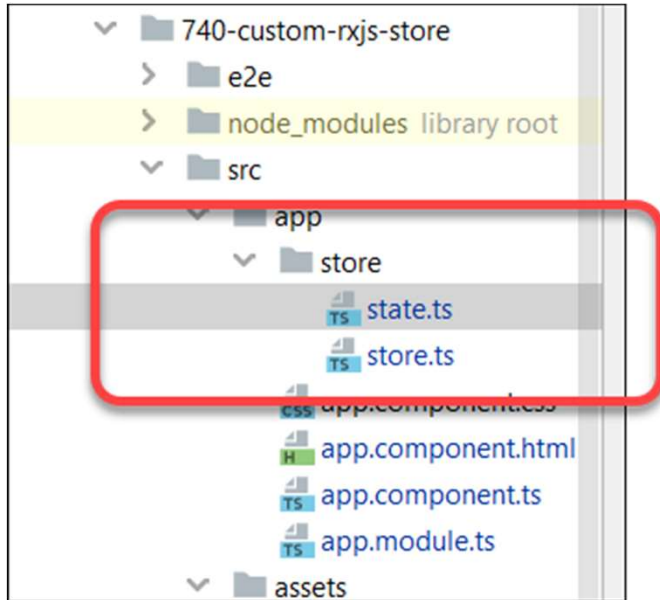
Building a store from scratch

Using observables, standard RxJS techniques and custom code, without a library

Steps

1. Create a `../store` folder and `store.ts` file, add it to the module
2. Create interface `State` for the data you want to 'store' (duh...)
3. Create a constant `state` of type `State`
4. Create a subject of type `BehaviorSubject` with type `State`, initialize it with initial `state`.
5. Expose the subject as an observable
6. Create `.set()` method and `.select()` methods

Step 1 – create store, and Step 2) state



We are now creating a *simple store*, for one application, with one (1) module.

With `@ngrx/store` things can get – way – more complex. This example is to demonstrate how store concept works.

```
// state.ts
export interface Todo {
  id: number;
  name: string;
  done: boolean;
}
export interface State {
  todos: Todo[];
  // other slices of the store
}
```

Step 3 – create state, and Step 4) BehaviorSubject

```
// store.ts
import {State} from './state';

const state: State = {
  todos: undefined
};

export class Store {
  // use behaviorsubject to create a subject with initial state
  // the last value is also passed to new subscribers.
  // The behaviorsubject holds the data (i.e. state)
  private subject = new BehaviorSubject<State>(state);
}
```


So the state is initially a list of undefined `Todo`'s.

We're going to set them later from the code. Of course you can fetch them from a backend and so on.

We use `BehaviorSubject` to create **initial state**. A `Subject` cannot do that.

On BehaviorSubject<Type>()

- BehaviorSubject can hold a variable (i.e. state), where a Subject can not.
- New subscribers get a copy of that data, i.e. the last emitted state, which of course is very useful in a store scenario.
- You pass a new piece of data to the BehaviorSubject with the `.next()` method.

 Learn RxJS

- Introduction
- LEARN RXJS
- Operators >
- Subjects >
 - AsyncSubject
 - BehaviorSubject**
 - ReplaySubject
 - Subject
- Recipes >
- Concepts >

Example 1: Simple BehaviorSubject

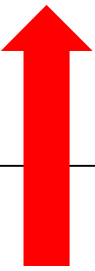
(Stackblitz)

```
1 // RxJS v6+
2 import { BehaviorSubject } from 'rxjs';
3
4 const subject = new BehaviorSubject(123);
5
6 // two new subscribers will get initial value => output: 123, 123
7 subject.subscribe(console.log);
8 subject.subscribe(console.log);
9
10 // two subscribers will get new value => output: 456, 456
11 subject.next(456);
12
13 // new subscriber will get latest value (456) => output: 456
14 subject.subscribe(console.log);
15
16 // all three subscribers will get new value => output: 789, 789, 789
17 subject.next(789);
18
19 // output: 123, 123, 456, 456, 456, 789, 789, 789
```

<https://www.learnrxjs.io/learn-rxjs/subjects/behaviorsubject>

Step 5 - Expose the subject as an observable

```
// store.ts
export class Store {
  // use behaviorsubject to create a subject with initial state
  // the last value is also passed to new subscribers.
  // The behaviorsubject holds the data (i.e. state)
  private subject = new BehaviorSubject<State>(state);
  private store = this.subject.asObservable()
    .pipe(
      distinctUntilChanged() // make it a little bit smoother, don't overnotify the subscribers
    );
}
```



The `store` is the variable we expose to the outer world later on, so components and services deal with an observable instead of a subject directly

Step 6 – create `.set()` and `.select()` methods

- Also create a helper `get` property that returns the current value of the state

```
// store.ts
export class Store {
  ...
  // internal helper function, return the current
  // value of the subject.
  get value(): any {
    return this.subject.value;
  }

  // set a new piece in the store. Update the
  // current store, using the spread operator (favor immutability)
  set(name: string, payload: any): void {
    this.subject.next({
      ...this.value, [name]: payload
    });
  }

  // select a slice from the store, use pluck to only fetch the
  // requested branch of the json-tree from the store
  select<T>(name: string): Observable<T> {
    return this.store.pipe(
      pluck(name),
    );
  }
}
```

Getter (internal)

Setter, using `.next()`

Selector, using `.pluck()`

Dynamically set the name of the property in the store. If it doesn't exist, it creates it

Access our store, only return the selected slice

This is all we need to do to create a reactive store!

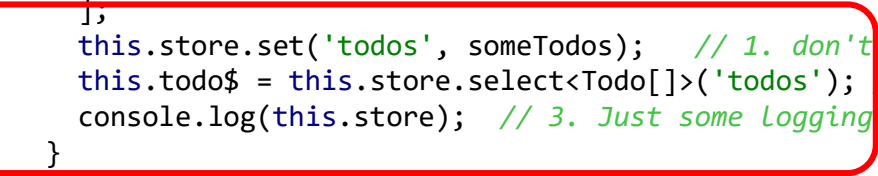

Use the store in the component

- Import the store in the component
 - set data, retrieve that data and bind it to the UI

```
// app.component.ts
export class AppComponent implements OnInit {
  todo$: Observable<any>;

  constructor(private store: Store) {
  }

  ngOnInit(): void {
    // just some dummy data in the store, you can fetch this from a backend of course
    const someTodos: Todo[] = [
      {id: 1, name: 'Get breakfast', done: false},
      {id: 2, name: 'Go coding', done: false},
      {id: 3, name: 'Attend meeting', done: false},
    ],
    this.store.set('todos', someTodos); // 1. don't set a component property directly! Instead, set data in the store
    this.todo$ = this.store.select<Todo[]>('todos'); // 2. Fetch data from the store, assign it to local property
    console.log(this.store); // 3. Just some logging, to see if the store works
  }
}
```



Note: **no callbacks here**. Everything is **reactive** (as one would expect from a reactive store)

Result

```
<div class="container">
  <h1>Custom Store</h1>
  <ul class="list-group">
    <li class="list-group-item"
      *ngFor="let todo of todos" | async">
      {{ todo.id }} - {{ todo.name }}
    </li>
  </ul>
  <hr>
</div>
```

The screenshot shows a web browser at localhost:4200 displaying a web application titled "Custom Store". The application has a list of three items: "1 - Get breakfast", "2 - Go coding", and "3 - Attend meeting". To the right of the browser, the developer console is open, showing the state of the application. A red box highlights the state of the "Store" component, which contains an array of three todos. A red arrow points from the list in the browser to the state in the console.

Custom Store

- 1 - Get breakfast
- 2 - Go coding
- 3 - Attend meeting

current store: {todos: Array(3)} store.ts:35

▼ Store {subject: BehaviorSubject, store: Observable} app.component.ts:34

value: (...)

subject: BehaviorSubject {isScalar: false, observers: Array(1), closed: false, isStop...

store: Observable {isScalar: false, source: Observable, operator: DistinctUntilChange...

▼ value: Object

▼ todos: Array(3)

- 0: {id: 1, name: "Get breakfast", done: false}
- 1: {id: 2, name: "Go coding", done: false}
- 2: {id: 3, name: "Attend meeting", done: false}

length: 3

▼ __proto__: Array(0)

▼ __proto__: Object

constructor: class Store

set: f set(name, payload)

select: f select(name)

get value: f value()

▼ __proto__: Object

Angular is running in the development mode. Call enableProdMode() to enable the production mode. core.js:40480






[WDS] Live Reloading enabled. client:52



Updating the store

Writing new values in the store by writing a custom
`.update()` method

Update the list of Todo's in the store

```
// Update the store, in this case a list of todos
updateTodo(name: string, payload: Todo): void {
  // 1. fetch the correct slice from the store (even if we only have one)
  const value = this.value[name];  Get correct slice
  const newTodos: Todo[] = value.map((todo: Todo) => {  Loop over items, use array mapping (!)
    // 2. Loop over our todos and update the given item
    if (payload.id === todo.id) {
      return {...todo, ...payload};  Return updated item...
    } else {
      return todo;  Or simply return if not applicable
    }
  });
  // 3. Set the store with the new value of newTodos
  this.set(name, newTodos);  Write new array in the store
  // 4. Optional - write state to localStorage, save todos in backend, etc.
}
```

Update the UI and logic for component

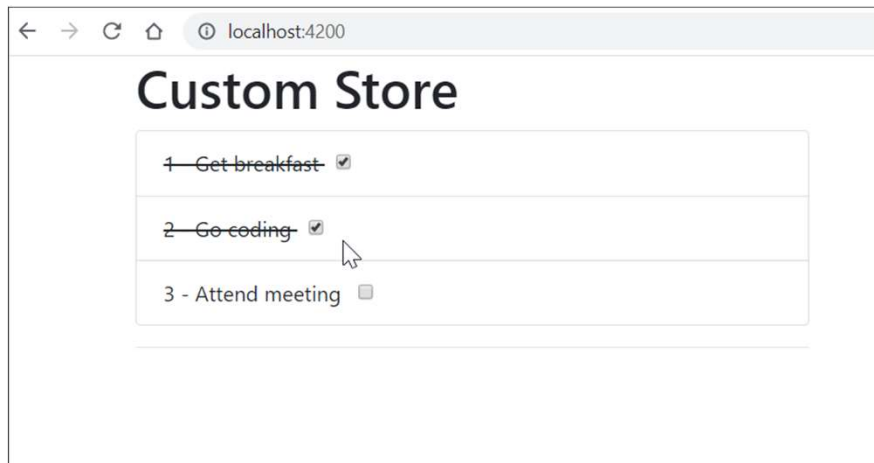
```
<ul class="list-group">
  <li class="list-group-item"
    *ngFor="let todo of todos" | async">
    <span [ngClass]="{'todo-done': todo.done}">
      {{ todo.id }} - {{ todo.name }}
    </span>
    <input type="checkbox" [checked]="todo.done" (change)="updateTodo(todo)">
  </li>
</ul>
```

custom CSS class

Add behavior

```
// update the state of a todo item
updateTodo(todo: Todo) {
  // toggle the state of item
  todo.done = !todo.done;
  this.store.updateTodo('todos', todo);
}
```

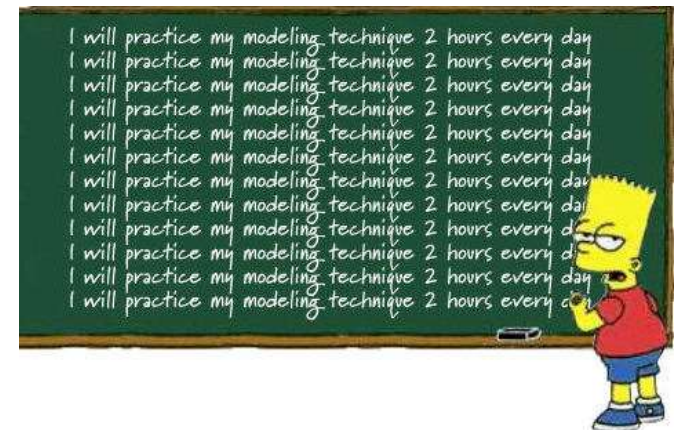
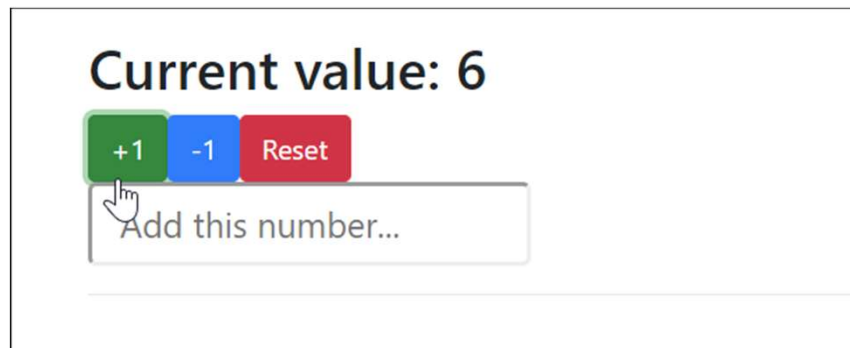
Toggle state and update the store



Result

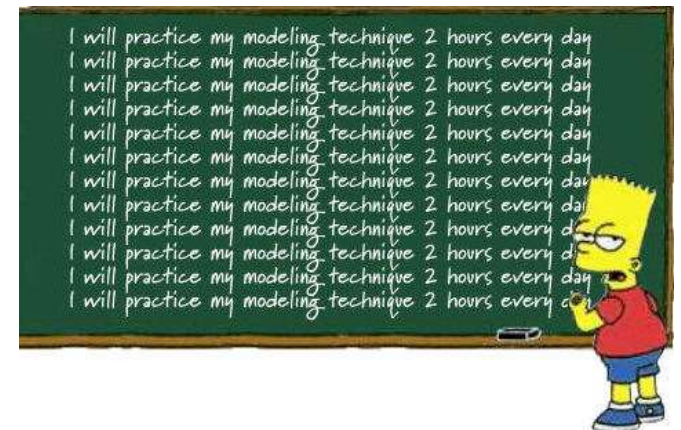
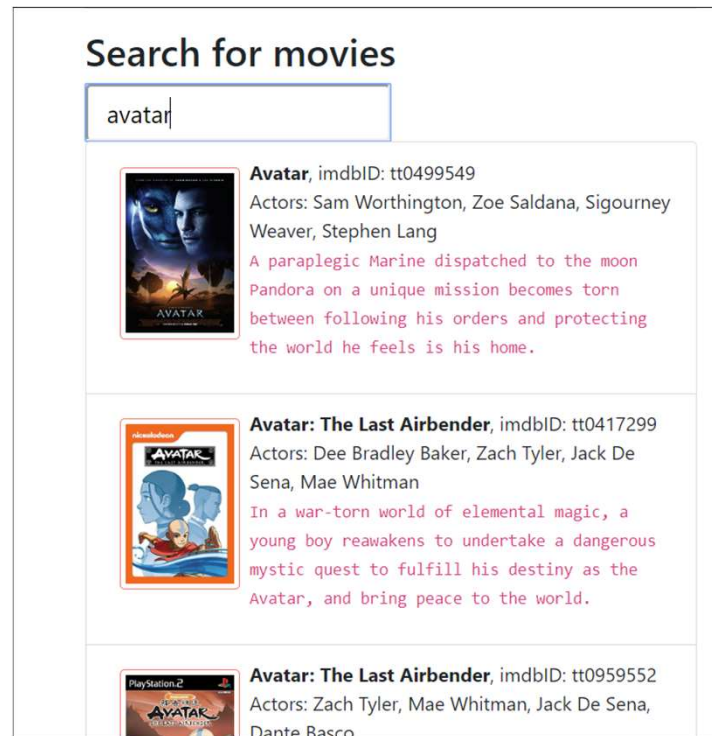
Workshop - 1

- Create a new app, Create a custom store, as described in the slides
- OR: Start from `../740-custom-rxjs-store`
- Create a `counter$` property and add it to the store.
 - In your component: show buttons to `increment()`, `decrement()` and `reset()` the `counter` in the store
 - Add it –for now – to the *same component, for simplicity*
- Some UI and logic is already available in the example, but first try it yourself!



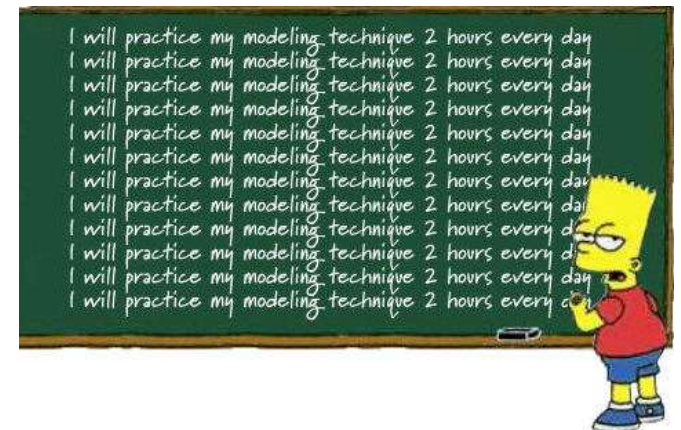
Workshop - 2

- Create a `movies$` property and add it to the store.
- Add a textbox to search for movies, put movies in the store.
- Search for movie details, based on the `imdbID` which is now available.
- Some UI and logic is already available in the example, but first try it yourself!



Optional workshop - 3

- Add the router, (like in `../740-custom-rxjs-store`)
- Make sure that the store contents survive a switch in components.
 - E.g. Movies retain in the store, the counter value is preserved, and so on.
- Tip: *don't reinitialize* the store in the `ngOnInit()` of every component, instead, do it once in `app.component.ts` and work from there.



Simple (KISS) Observable Store by Dan Wahlin

The video player shows a presentation slide titled "Stepping Up: Observable Services to Observable Store" by Dan Wahlin. The slide illustrates a design pattern for an Observable Store. It features a central diagram with four "Service" nodes (red, purple, green, and grey) and eight "Component" nodes (blue). Arrows indicate dependencies: the top-left red Service depends on two Components; the top-right purple Service depends on two Components; the bottom-left orange Service depends on two Components; and the bottom-right grey Service depends on two Components. Additionally, the central green Service depends on two Components, which in turn depend on the top-left red Service and the bottom-left orange Service. The video player interface includes the YouTube logo, a search bar, and a video player with a progress bar at 2:24 / 19:19. The video title is "Stepping Up: Observable Services to Observable Store" and the channel is "ng-conf: Hardwired". The video has 133 likes and a "Subscribe" button is visible.

<https://github.com/DanWahlin/Observable-Store>

Blogpost on Observable Store

The screenshot shows a Dev.to profile page for Steve Whitmore. The header includes a 'DEV' logo, a search bar, a 'Create Post' button, and a notification badge with the number 23. On the left sidebar, there are icons for likes (3), comments (0), and bookmarks (2). The main content area features a large profile picture of Steve Whitmore, a man with dark hair and red glasses. Below the picture, his name 'Steve Whitmore' and the text 'Posted on 25 okt.' are visible. The article title is 'State Management Is Easy Now!? An Introduction to Observable Store', followed by the tags #angular, #architecture, and #javascript. The article text begins with 'Oh application state. The source of so many bugs. React has [Redux](#), Vue has [Pinia](#), and Angular has [NgRx](#) as their most used state management libraries. NgRx is a great solution for folks well versed in Angular-ese and can make tackling problems with enterprise-sized applications way easier. There's a

The right sidebar contains a 'Follow' button and a bio: 'Full-stack dev specializing in frontend technologies and building CI pipelines.' It also lists his location as 'Kansas City, MO', education as 'Some college. Mostly self taught.', work as 'Software Engineer at NAIC', and joined date as '12 sep. 2020'. Below this, there is a section 'More from Steve Whitmore' with two article teasers: 'Take Your WordPress Site Farther With Angular' (tags: #wordpress, #angular, #typescript) and 'Take Your WordPress Site Further With Vue.js' (tags: #wordpress, #php, #vue, #javascript).

<https://dev.to/stevewhitmore/state-management-is-easy-now-an-introduction-to-observable-store-15ij>