

STATE MANAGEMENT

Ori Calvo, 2017

oric@trainologic.com

<https://trainologic.com>

Assumptions

2

- Component based architecture
 - ▣ Component = tag+code+template+styles
 - ▣ Component is responsible for managing the view/DOM
- Application consists of multiple components
- We want isolated components
 - ▣ Allows for reusability
 - ▣ Easy of maintenance
- Components effect each other

The Challenge

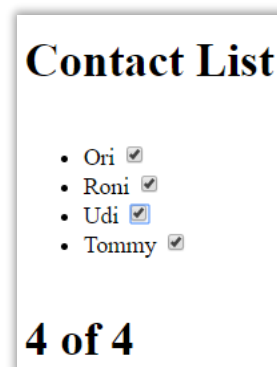
3

- ❑ Keep all components synchronized
- ❑ Minimize DOM writes
- ❑ Minimize change detection
- ❑ Easily understand the flow of a change
- ❑ State modification is atomic
- ❑ State is valid even in the case of an error
- ❑ Restore state from server/client/routing
- ❑ Support undo ?

Scenario

4

- A component which displays a list of contacts
- Each contact can be selected
- Status component which displays the number of selected contacts and the total number
- Selecting/unselecting component effects the status
- The contact list component should be unaware of the status component



Solution 1 – Mediator

5

- Use the parent component to manage the interaction between siblings
- The contact-list component raises an event
- The parent handles the event and updates internal state which is bound to the status component

Solution 1 - Component Mediator

6

```
<h1>Contact List</h1>

<my-contact-list [contacts]="contacts"
                  (selectionChanged)="onSelectionChanged($event)">
</my-contact-list>

<my-status [all]="contacts"
            [selected]="selected">
</my-status>
```

```
export class AppComponent {
  contacts: Contact[];
  selected: Contact[];

  constructor() {
    this.contacts = [...];
    this.selected = [];
  }

  onSelectionChanged($event: SelectionChanged) {...}
}
```

Pros & Cons

7

- Straight forward solution
- Since parent manages the whole state it can be easily duplicated and reused
- Intermediate components must support inputs that are reflection of deep components state
- Intermediate components need to “bubble up” events

Solution 2 – Service Facade

8

- ❑ Remove all inputs & events
- ❑ All components talk to a single service façade
- ❑ The service holds the whole state for all components
- ❑ Each component is bound to the state directly
 - ▣ By holding a reference
- ❑ A component delegates user action to the service
- ❑ The service “fixes” all relevant state

Solution 2 – Service Facade

9

```
export class ContactService {
  contacts: Contact[];
  selected: Contact[];

  constructor() {
    this.contacts = [...];
    this.selected = [];
  }

  changeSelection(contact, selected) {...}
}

export interface Contact {
  id: number;
  name: string;
  selected?: boolean;
}
```

```
export class ContactListComponent {
  constructor(private contactService: ContactService) {
  }

  get contacts() {
    return this.contactService.contacts;
  }
}
```

```
export class ContactListItemComponent {
  @Input() contact: Contact;

  constructor(private contactService: ContactService) {
  }

  changeSelection(contact, selected) {
    this.contactService.changeSelection(contact, selected);
  }
}
```

Pros & Cons

10

- ❑ No need to propagate event/inputs through intermediate components
- ❑ State is more accessible since it resides inside a service which can be injected everywhere
- ❑ Service is singleton → Cannot duplicate components with different state
 - ▣ Can be fixed by using Component's provider
- ❑ Components are stateless → Can remove/recreate component while keeping state
 - ▣ Router scenario

System wide solution ?

11

- Can we apply this technique to the whole application ?
 - ▣ No state inside component
 - ▣ Every component delegates the work to a service
 - ▣ The service is responsible for fixing all related state
 - ▣ Angular is responsible for updating components through data binding
- Yes we can. However ...

System wide solution ?

12

- A component should not be aware of the scope of a change
- Therefore, most components delegate the work to a single root service
- The root service is responsible for “spreading the news”
- Can use
 - ▣ Broadcasting mechanism
 - ▣ Manual invocation

Unidirectional Data Flow

13

- State resides inside services
- Each service may use other service to handle part of the change
- It is important to control the change flow
 - ▣ We don't want cycles
- We can arrange services inside tree and prohibit access between
 - ▣ Sibling services
 - ▣ Child to parent

Reaction

14

- Two components/services might need the exact same data
- Who is the owner ? SRP again ...
- Solution
 - ▣ One service is responsible for modifying the data
 - ▣ The other one only “reacts” to the change
- We need two phases
 - ▣ One for pushing data
 - ▣ Second for synchronizing it
- Both should be unidirectional

Reaction

15

```
export class RootService {  
  state: AppState;  
  
  constructor(private contactService: ContactService,  
               private selectionService: SelectionService,  
               private searchService: SearchService) {  
  }  
  
  refresh() {  
    this.contactService.refresh();  
    this.onContactsLoaded();  
  }  
  
  private onContactsLoaded() {  
    const all = this.state.contacts.all;  
    this.searchService.onContactsLoaded(all);  
    this.selectionService.onContactsLoaded(all)  
  }  
}
```

Reaction

Modification

Can use broadcasting
instead of manual
invocation

Where are we ?

16

- ❑ End user clicks a button
- ❑ Component initiates an activity/action
- ❑ A service manages the scope of the activity
- ❑ Application state is changed
- ❑ Some effects/reactions may occur after state modification
- ❑ State is now stable
- ❑ Angular/React detects changes and modifies the DOM

Annotation

17

- Annotations are powerful way to describe code intents

```
export class SelectionService {  
  state: SelectionState;  
  
  @Activity()  
  change(contact: Contact, selected: boolean) {  
    if(selected) {  
      this.state.selected.add(contact);  
    }  
    else {  
      this.state.selected.delete(contact);  
    }  
  }  
  
  @Query()  
  isSelected(contact: Contact) {  
    return this.state.selected.has(contact);  
  }  
  
  @Reaction()  
  onContactsLoaded(all: Contact[]) {  
    this.state.all = all;  
  }  
}
```

Not just description

18

- Decorator can intercept method invocation

```
export function Activity() {  
  return function(target, prop) {  
    const original = target[prop];  
  
    return target[prop] = function() {  
      const name = target.constructor.name + "." + prop;  
      console.log("BEGIN", name);  
  
      const before = performance.now();  
      const retVal = original.apply(this, arguments);  
      const after = performance.now();  
  
      console.log("END", name, (after-before));  
  
      return retVal;  
    }  
  }  
}
```

Optimizing Change Detection

19

- ❑ The previous pattern offers no optimization
- ❑ A single action can change every thing
- ❑ There is no easy way for Angular to understand the scope of a change
- ❑ Therefore, Angular checks every component for each user/async event
 - ▣ Is it really a bad idea ?
 - ▣ Angular needs only a few milliseconds (<10) to check more than 10000 bindings

Hold your horses

20

- Assuming optimization is the art of compromising
- What is the price of optimizing change detection ?
 - ▣ Answer: A restricted state modification logic
- For applications that significantly hold more data than bindings
- The restrictions might incur higher price than the benefit of optimized change detection

Optimizing Change Detection

21

- Angular offer some ways to optimize change detection
- It assumes that data does not change inline
- But rather the whole object is cloned and updated
- Thus, Angular can just examine the reference/input

ChangeDetectionStrategy.OnPush

22

- ❑ Angular skip component's change detection unless one of its input changes
- ❑ An input is “shallow” checked
 - ▣ Only the reference is checked
 - ▣ Not nested fields
- ❑ Therefore we need to use immutable data and clone the data once it changes

Immutability - System wide solution ?

23

- Should we apply the immutability principle cross the whole application ?
- Going that direction means our code changes dramatically
 - ▣ No more inline editing ☹

```
const clone = this.state.all.concat([]);

for(let update of updates) {
  const index = clone.findIndex(c => c.id == update.id);
  if(index !== -1) {
    clone[index] = Object.assign({}, clone[index], update);
  }
}
```

Immutability - Advantages

24

- Immutability offers many advantages
 - ▣ It is easy to detect a change
 - ▣ So even components without Input can be optimized
 - ▣ History data can be saved and restored → Undo is easy to implement
 - ▣ Data never changes → Data is atomic
- However
 - ▣ Cloning object is expensive
 - ▣ Must measure cloning vs. change detection

Where are we ?

25

- Externalizing components state
- Unidirectional data flow
- Single data store
- Immutability
- Modification has side effects/reaction

- Can someone help with that mess ?

Redux

26

- The most popular implementation of the Flux pattern
- Is not strictly Flux compatible
- Has no framework affinity
 - ▣ Although is most popular under the React eco-system
- More than 2M downloads per month

Redux - Ingredients

27

- Store – A single immutable data store
 - ▣ Everyone can subscribe to changes
- Action – Resembles a user action
- Reducer
 - ▣ Responsible for modification
 - ▣ Enforce immutability
 - ▣ Cannot handle asynchronous operations
- Action creator/Thunk
 - ▣ Business logic
 - ▣ Handles asynchronous activities

ngrx

28

- ❑ Same principles of Redux, but
- ❑ Store is observable
 - ▣ Thus, can react to change using reactive programming
- ❑ Effects can be installed
 - ▣ Same role as `redux/thunk`
 - ▣ But again ... reactive programming

Mobx

29

- Adds observable capabilities to existing data
- You define a simple data model
- Using decorators Mobx monkey patches the properties and make them observable
 - ▣ Does it make you a bit nervous ?
- Mobx is aware of any data change and therefore can render component automatically
- Mobx is unopinionated about how user events should be handled

Summary

30

- The principles are important not the frameworks
 - ▣ Unidirectional data flow
 - ▣ Single store
 - ▣ Immutability
 - ▣ Externalize state & work out of components
- Functional/Reactive programming changes some of out patterns