# State management in react applications

## Perex

Managing state as application’s complexity grows can easily become too tricky. However there are some rules, methods and tools, which can make this task easier and keep the architecture of the system neat and readable. Let’s tame the beast in the next few lines.

## Types of state

To be able to address the topic, common issues and their solutions, we need first some clarification of what we are dealing with. There is some common language on the topic already settled.

State from the point of view the methods of its persistence and its purpose can be divided into following types:

* + Persistent
  + Client
  + Server
  + Ui state
  + transient state
  + Router state

#### Persistent State

Persistent state is everything, the client can see after the authentication process kicks in. It is mostly stored on the server and is retrieved after every login and the following initialization of the application.

This state is shared by the other two types of state. Client state and the Server State.

#### Client State

This state is everything stored on the individual client the application runs at. It is mostly initialized by the fetch of the server state. But it mostly differs from the server state in some details. Not everything on the server is necessarily present in the client state. On top of that, Client state can be personalized by web browser storage capabilities (Session Storage, Local Storage, Indexed DB).

#### Server State

This is everything that is supposed to have lifespan independent of the lifespan of the client session. Server state usually makes up substantive part of the Persistent state.

#### UI state

Mostly short-lived state of the individual parts of the application. In React application, it’s mostly implemented as state of the component – useState hook of functional component or setState of class-based component. Sometimes can be partly stored in the client or on the server and be a part of the persistent state.

#### Transient State

This state comes from the users interaction with the application. It is often bound to the particular client of user account. For example when you watch clip on the youtube and get back to it later on, you will be offered to start from the point you stopped watching last time. However, if you pass the link to the clip to your friend who never watched it before, he will be offered to start watching from the very beginning of the clip.

#### Router State

This is state of the router of the application. Parameters, query values and path to page are stored in it. It is mostly handled by libraries in SPA environment. In React applications, the most popular – at least counting the NPM weekly downloads – is react-router.

## Introduction of sample application in state one

I created a simplistic application for the purpose of this article, which is hosted on github:

<https://github.com/PetrBorak/state-management-react>

To make it run on your machine, reproduce the following steps in terminal:

* git clone <https://github.com/PetrBorak/state-management-react.git>
* cd [folder]
* npm install
* npm run start
* npm server

Of course, I suppose, you have already Node.js and git present on your machine.

The application contains, on top of what the create-react-app will produce, the following main part, which we will work with here:

* The todos component
  + - * + state-management-react/src/todosComponent
* The backend service
  + - * + state-management-react/src/backend

#### TODO Component

The Todo component handles the fetch of the todos and their rendering. At this phase, it contains the following code:

Code >

**import *React***, { ReactDOM, *useEffect*, *useState* } **from 'react'  
import** { *fetchTodos*, getTodos } **from "../backend/backend"**;  
  
**export const** *Todos* = (props) => {  
 **const** [todos, setTodos] = *useState*([])  
 **const** filterSelection = (ev) => *fetchTodos*(ev.**target**.**value**).then((result) => {  
 setTodos(result)  
 });  
 *useEffect*(() => {  
 *fetchTodos*().then((result) => {  
 setTodos(result)  
 })  
 }, []);  
  
 **return** (  
 <**div**>  
 <**h2**>HELLO WORLD FROM TODOS COMPONENT</**h2**>  
 <**select onChange=**{filterSelection}>  
 <**option value=**{**'ALL'**}>ALL</**option**>  
 <**option value=**{**'DONE'**}>DONE</**option**>  
 <**option value=**{**'WAITING'**}>WAITING</**option**>  
 </**select**>  
 <**h2**>TODOS:</**h2**>  
 {todos.map((todo)=><**div**><**h3**>{todo.**title**}</**h3**>{todo.**body**}<**input type='checkbox' checked=**{todo.**state**===**'DONE'**}/></**div**>)}  
 </**div**>  
 )  
}

It renders the individual todos, their title and checkbox indicating, wheter the todo is waiting for competition or is already done.

The component is functional and uses useEffect and useState hooks:

* UseEffect fetches the todos, when the component mounts
* UseState stores the fetched todos for the component

More about functional hooks can be read here: <https://reactjs.org/docs/hooks-intro.html>

#### Backend Service

Backend service only handles the fetching of the todos and creates a request string based on parameters passed to it.

Here is the code for it:

Code >

**export const** *fetchTodos* = (params) => *fetch*(**`http://localhost:4000/todos**${params ? **'?filter='** + params : **''**}**`**).then((result) => result.json());

##### Drawbacks

Well, the application is fully functional now. It fetches data and parses them properly. Everything seems perfectly reasonable. However, there are a few drawbacks to this solution.

###### State is adhoc

Keeping the persistent state, or parts of it in the individual components is a real risk for the application and can incur substantional danger of design rot.

We are more or less OK with this solution, but as the complexity of the application grows the danger will become more and more obvious. One of the dangers is pretty tricky. Namely the accidental mutation in combination with side effects.

Mutation hides changes, and hidden changes brings non-determinism into the application. We do not want non-deterministic behavior in the app.

To avoid problems with this type of problem, there was evolved system of tools, that handle state in the unidirectional, readable way and keep the mutation away with the help of combination of pure functions and transaction records about the changes done in the state.

Yep, your ques is probably right. I am talking about Redux right now. I won’t elaborate more about why it’s so useful for the architecture of the app. You can always read more, for example in this article:

<https://redux.js.org/introduction/getting-started>

Actually, the first sentence of this article is very good summary of my previous words:

“Redux is predictable state container for Javascript apps.”

### Connection of redux to the application

To avoid the aforementioned risks, let’s implement the TodosComponent with redux. We move the ad-hoc state to Redux container and amend the fetchTodos function. This function now calls the backend and, in case of successful fetch updates the Redux store with new set of Todos.

To do that, we need to define the Store by adding the following files to our solution:

* state-management-react/src/store
* state-management-react/src/store/todos
  + - * + here are the action creators and reducers for our store
* state-management-react/src/store/todos/todos.js
  + - * + the file with reducers and initial state
* state-management-react/src/store/todos/todosActions.js
  + - * + the file with action definitions and action creators

Lets look at the content of this new files briefly:

#### state-management-react/src/store/todos

**const** initialStateTodos = {  
 **todos**: []  
}  
  
**export const** *todosReducer* = (state = initialStateTodos, action) => {  
 **switch**(action.**type**){  
 **case 'FETCH\_PENDING'**:  
 **return** {...state, **loading**: **true**}  
 **case 'FETCH\_SUCCESS'**:  
 **return** {...state, **todos**: action.**payload**}  
 **case 'FETCH\_ERROR'**:  
 **return** {...state, **error**: **true**}  
 **default**:  
 **return** state;  
 }  
}

#### state-management-react/src/store/todos/todosActions.js

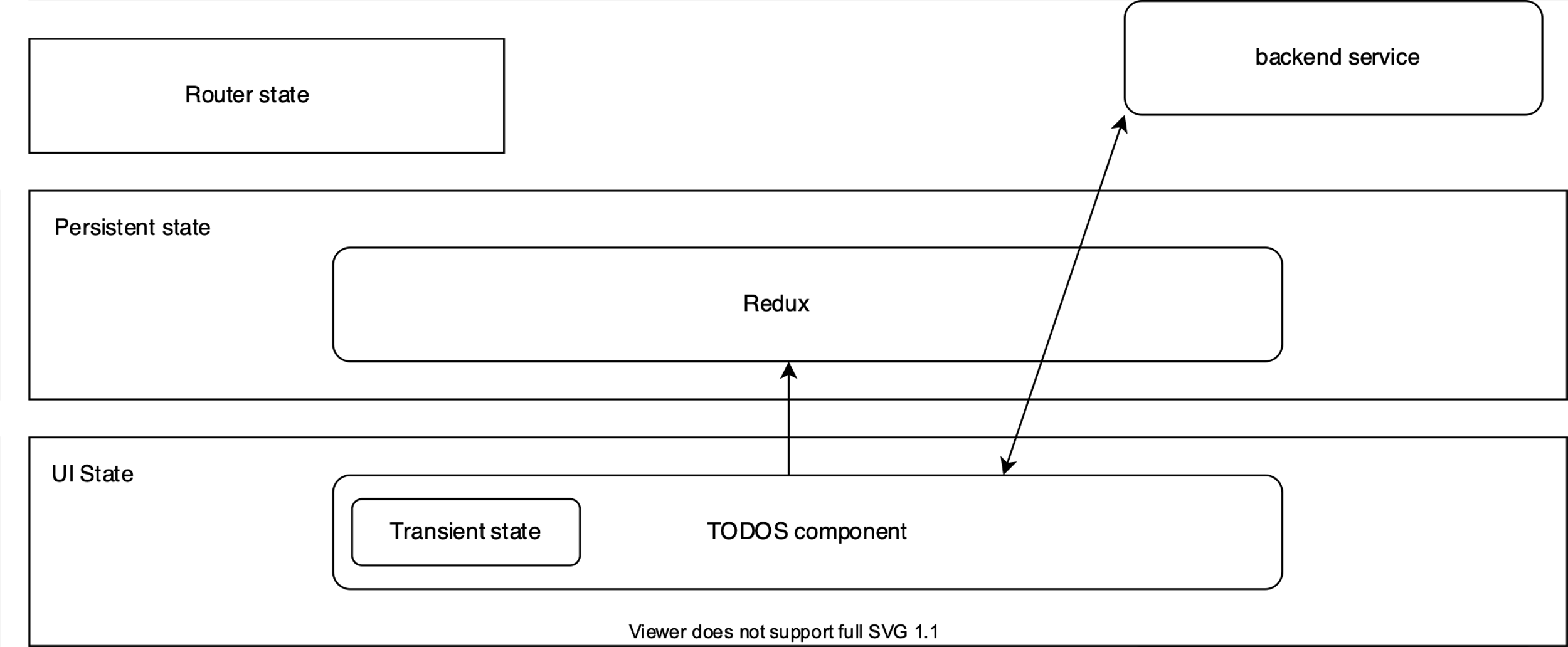
**export const** *todosActionCreatorFetchSuccess* = (payload) => ({  
 **type**: **'FETCH\_SUCCESS'**,  
 **payload**: payload  
})  
  
**export const** *todosActionCreatorFetchError* = () => ({  
 **type**: **'FETCH\_ERROR'**,  
})  
  
**export const** *todosActionCreatorFetchPending* = () => ({  
 **type**: **'FETCH\_PENDING'**,  
})

#### And this is how the Todos component looks now:

**import *React***, { *useEffect*, *useCallback* } **from 'react'  
import** { *fetchTodos* } **from "../backend/backend"**;  
**import** {connect} **from "react-redux"**;  
  
**import** { *todosActionCreatorFetchSuccess*, *todosActionCreatorFetchError*, *todosActionCreatorFetchPending*} **from '../store/todos/todosActions'  
  
export const** *Todos* = (props) => {  
 **const** { fetchTodosPending, fetchTodosSuccess, todos } = props;  
  
 **const** fetchAndStore = *useCallback*((param = **''**) => {  
 fetchTodosPending()  
 *fetchTodos*(param).then((result) => {  
 fetchTodosSuccess(result)  
 })  
 }, [fetchTodosSuccess, fetchTodosPending])  
  
 **const** filterSelection = (ev) => fetchAndStore(ev.**target**.**value**)  
  
 *useEffect*(() => {  
 fetchAndStore()  
 }, [fetchAndStore]);  
  
 **return** (  
 <**div**>  
 <**h2**>HELLO WORLD FROM TODOS COMPONENT</**h2**>  
 <**select onChange=**{filterSelection}>  
 <**option value=**{**'ALL'**}>ALL</**option**>  
 <**option value=**{**'DONE'**}>DONE</**option**>  
 <**option value=**{**'WAITING'**}>WAITING</**option**>  
 </**select**>  
 <**h2**>TODOS:</**h2**>  
 {todos.map((todo)=><**div**><**h3**>{todo.**title**}</**h3**>{todo.**body**}<**input type='checkbox' checked=**{todo.**state**===**'DONE'**}/></**div**>)}  
 </**div**>  
 )  
}  
  
**const** mapDispatchToProps = (dispatch) => ({  
 fetchTodosSuccess: (payload) => dispatch(*todosActionCreatorFetchSuccess*(payload)),  
 fetchTodosPending: () => dispatch(*todosActionCreatorFetchPending*()),  
 fetchTodosError: () => dispatch(*todosActionCreatorFetchError*()),  
})  
  
**const** mapStateToProps = (state) => {  
 **debugger  
 return** {  
 **todos**: state.**todos** }  
}  
  
**export default** connect(mapStateToProps, mapDispatchToProps)(*Todos*)

The most important part is that in the fetchAndStore function we now call the individual actions in the store and in case of successful fetch we dispatch the final FETCH\_SUCCESS action with payload containing the individual todos.

This is how our state management in the application looks like right now



### Implementation of redux saga

The component now handles the fetch through the backend service and dispatches appropriate action in the Redux store.

As you can see the router state, though its present in the application in the form of react-router, is now completely isolated.

One of the problems with this is, that we are not able to mirror the state of the parameters in the URL for now. That means, that when putting the application’s filters into some state, we are not able to reproduce this information other way, than setting up the filters by hand everytime the app is loaded.

### Handling the URL as source of true

The state of the filters is thus adhoc and transient. Not to mention, that because we do the new fetch each time the filters change, we have no record of them in the global Redux state.

The state is simply not part of the running application. We won’t change it, because its sufficient for the purpose of this tutorial. The shape of state is always a subject to discussion across the development team.

For now, let’s focus on other problem with recent state of the application.

The URL should be always handled as the primary source of true in the application. The reason for that is, that user can whenever interact with the URL through the omnibox.

For now, for example, once the the user refresh or copy the URL to application, the filters state is back to its default values.

#### Side effects In Redux application

As already mentioned the reducers should be pure functions and the state amended by them should be immutable.

But the way, we fetch the data to the application right now – by calling the backend service straight from the Todos component - is not very good solution either. That’s basically another side effect.

This kind of side effect influences the architecture of the application in a bad way. It cannot be seen right now, as our example is really simple, but again – as the complexity of the application grows, the modularity of it with this ad hoc solution is seriously impacted. The system’s design rot can easily take place. (More about the design rot here: <https://fi.ort.edu.uy/innovaportal/file/2032/1/design_principles.pdf>

)

Luckily for us, we have solutions to this problem. There is no one altogether right solution. Everything about the right architecture is more or less a set of advices, principles and patterns to follow.

So, to keep the architecture unaffected and to avoid the system design rot while scaling up, we should try to shift as much side effects as possible to places dedicated to them.

As this fetch call is basically a side effect closely linked to global state, which we have in Redux, we should move it to the place appropriate to Redux side effects.

One really popular solution for side effects in Redux is Redux Saga.

Again, this article is not intended to elaborate on how the libraries really work. You can read more about Redux Saga here:

<https://redux-saga.js.org/>

The todo component now initialize the fetch from the server trough the backend service. This mean

* Link to redux saga introduction article
* About side effects and middleware in Redux
* Description of the procedure of migrating as depicted in Sample application section

# Sample appilication

# Overview

### state one

Todos application

~~XXX Mark as done~~

* ~~Backend service will call the backend~~
* ~~Backend service will do optimistic update and wont check for the failure of the call~~
* ~~Backend service will keep all the state within itself~~

Filter done and not done

Structure

Backend and its state for todos in one go

* + - * + Fetch is called from the component

Componenent has hard coded filter values

component has hardcoded default values for filters

* + - * + Backend fetches the todos for the filter combination and stores them in its state
        + When filter changes in the component, backend is called and calls the endpoint for the combination of filters and the setup filters

This will not enable passing the combination of filters by copying the url as the filters are not part of it

~~XXXX Filters setup through a call to backend service~~

Filters hardcoded in the component

### State two

Implement redux

* Global state now isolated from the local state previously saved in the component part
* The component now calls backend (fetchArcitles method) and dispatches actions
  + - * + FETCH PENDING
        + FETCH SUCCESS
        + FETCH ERROR
        + FILTER CHANGE
* The backend is used only for the call of the endpoint (for combination of filters)
* The backend is now stateless
* The filter values are stored in redux
* The choosen filter values are now stored in redux
  + Component reads them through connect
  + When change occurs the component dispatches action
    - FILTER CHANGE
    - The reducer changes the choosen filters value
    - The component calls new backend fetch (fetchArcitles method)
  + The backend service is now completely stateless
    - Advantages
      * No mutation
      * No race condition
        + Mutation hides changes

Hidden changes lead to non determinism

Link to Eric Elliot

* State’s shape is the following
  + Todos
  + Filters – actual filters in the application
  + ChosenFilters – choosen combination of filters

## State three

Implement redux saga

* Actions
  + The same for now
  + The fetching of todos goes to the saga
    - New action FETCH TODOS
      * Dispatched by component
    - Saga reads the state of the filters
    - Saga calls the backend
      * + When OK, saga stores dispatches FETCH SUCCESS action
        + When backend error, saga dispatches FETCH ERROR
        + Backend calls the enpoind and is called from the saga
        + Component dispatches the action

Brief description of connect and link

<https://react-redux.js.org/api/connect>

### State four

Implement redux-router-connect

* Params are added to the route components route property
* Connection to the application
* Add saga listener to the router – this will :
  + The component with the change of filters will now
  + Call the router with params
  + The saga listening for route change will detect it and will call the dispatch
    - FILTERS CHANGE - create new action SETUP FILTERS ?
    - FETCH TOODOS