Web 3

Redux

HO
GENT

Gebruikte technologieën

- Redux DevTools
- Redux
- React-Redux
- Redux Toolkit

HO
GENT

State management

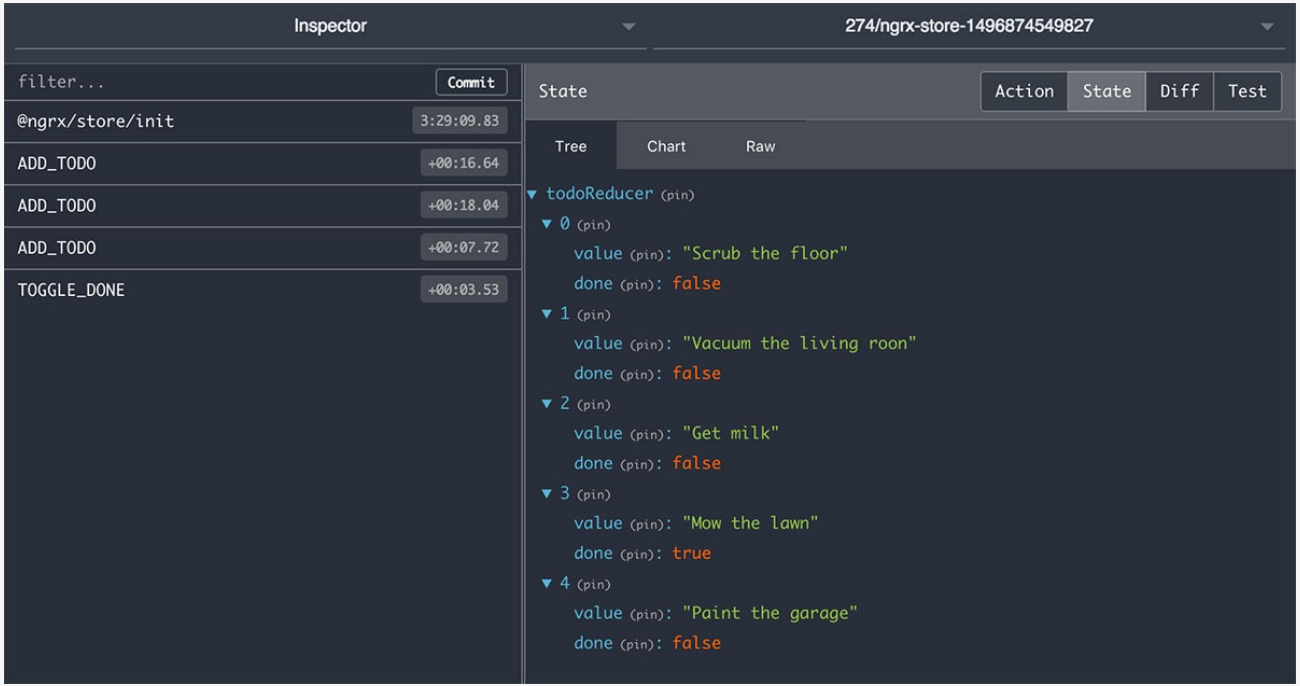
- **Tot nu** toe hebben we steeds onze state rechtstreeks **in** de React components gezet.
- Als onze applicatie **groter** wordt, moet onze state **uit** de React components gehaald worden
- Een populaire state management library is **Redux**.
- Facebook had een applicatie architectuur ontwikkeld nl. Flux. Redux is hierop gebaseerd en maakt het ons makkelijker voor de state management.

Redux DevTools

- De Redux DevTools zorgt ervoor dat we onze state kunnen bekijken als ook de wijzigingen die uitgevoerd zijn en dergelijke
- Onmisbaar als we met state aan het werken zijn om te debuggen
- Beschikbaar voor Chrome, Firefox en Edge
- Rechtermuisklik en dan kiezen voor Redux DevTools

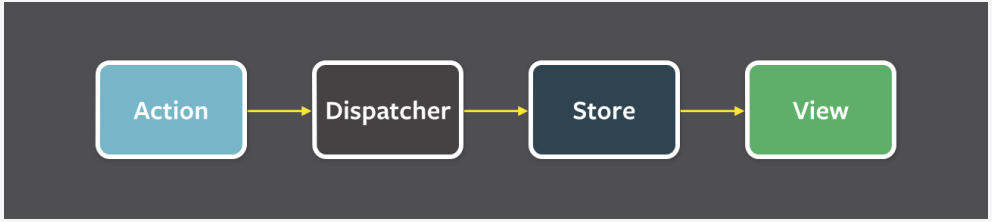
https://chrome.google.com/webstore/detail/redux-devtools/lmhkpmbekcpmknklioeibfkpmmfibljd
https://addons.mozilla.org/en-US/firefox/addon/reduxdevtools/

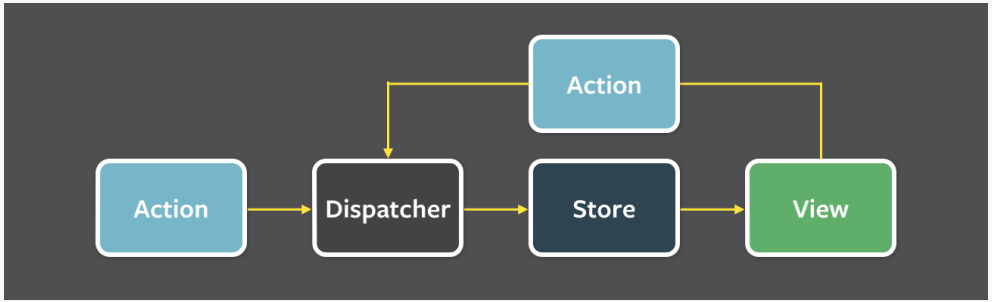
Redux DevTools (2)



HO
GENT

Flux architectuur





HO
GENT

Redux

- “A predictable state container for JS Apps”
- **Voorspelbaar** – consistente applicaties die zowel op client, server en native werken
- **Gecentraliseerd** – de state van de gehele applicatie is gecentraliseerd
- **Debuggable** – maakt het makkelijk om op elk moment te kijken wanneer, waar en waarom de applicatie state veranderd is
- **Flexibel** – werkt met nagenoeg elke UI

https://redux.js.org

HO GENT

Action

- Plain JavaScript Object
- Wordt gebruikt om iets te veranderen in onze state
- Een action wordt steeds gedispatched naar onze store
- Elke wijziging in de state moet worden gedaan via een action dit omdat we een overzicht kunnen bewaren wat er juist gebeurd met onze state

```
{ type: ‘ADD_TODO’, text: ‘Go to swimming pool’ }
{ type: ‘TOGGLE_TODO’, index: 1 }
```

HO
GENT

Dispatcher

- Dispatches een action
- De enige juiste manier om een wijziging in de state uit te voeren
- Onze store heeft dus een dispatch methode waar we een action kunnen meegeven

```
store.dispatch({ type: 'ADD_TODO', text: 'Read the docs' });
```

HO
GENT

Reducer

- Een reducer is een functie die twee argumenten heeft namelijk; de **huidige state**, en een **action**. Deze functie geeft dan een **nieuwe state** terug als resultaat
- Een reducer zal de actions opvangen en dan de state aanpassen naar een nieuwe state
- Elke reducer moet een initiële state hebben

Reducer (2)

```
const initialState = {
   todos: [
        { id: 1, text: 'Learn React', completed: true}
   ]
}

export default appReducer = ((state = initialState, action) => {
   switch (action.type) {
        default:
            return state;
   }
});
```

HO GENT

Store

- De Redux store brengt de **state**, de **actions** en de **reducers** samen
- Belangrijk er is maar **één store** in een Redux applicatie. Als je de state wilt opsplitsen gebruiken we meerdere reducers
- Verschillende verantwoordelijkheden:
  - Houdt de huidige state bij voor de app
  - Verleent toegang tot de huidige state via **`store.getState()`**
  - Zorgt er voor dat we onze state kunnen aanpassen via **`store.dispatch(action)`**
  - …

HO GENT

Store (2)

- We moeten de store nog koppelen aan onze applicatie om gebruik te kunnen maken van de hooks zoals `useSelector` en `useDispatch`
- De store wordt gekoppeld aan de App door middel van de Provider die in het `react-redux` package beschikbaar is

```
import {Provider} from 'react-redux';

<React.StrictMode>
   <Provider store={store}>
        <App />
   </Provider>
</React.StrictMode>
```

HO GENT

State

- De Redux state is een globale state over de hele applicatie
- Niet alle state moet in de Redux store
- Als je een state nodig hebt die bvb. maar door 1 component wordt gebruikt dan moet deze nog steeds in de component gedefinieerd worden met `useState`

State (2)

TIP

In a React + Redux app, your global state should go in the Redux store, and your local state should stay in React components.

If you're not sure where to put something, here are some common rules of thumb for determining what kind of data should be put into Redux:

- Do other parts of the application care about this data?
- Do you need to be able to create further derived data based on this original data?
- Is the same data being used to drive multiple components?
- Is there value to you in being able to restore this state to a given point in time (ie, time travel debugging)?
- Do you want to cache the data (ie, use what's in state if it's already there instead of re-requesting it)?
- Do you want to keep this data consistent while hot-reloading UI components (which may lose their internal state when swapped)?

Redux Toolkit

- De Redux store brengt de **state**, de **actions** en de **reducers** samen
- Belangrijk er is maar **één store** in een Redux applicatie. Als je de state wilt opsplitsen gebruiken we meerdere reducers
- Verschillende verantwoordelijkheden:
  - Houdt de huidige state bij voor de app
  - Verleent toegang tot de huidige state via **`store.getState()`**
  - Zorgt er voor dat we onze state kunnen aanpassen via **`store.dispatch(action)`**
  - …

HO GENT

Redux Toolkit

configureStore

- Om gebruik te kunnen maken moeten we een store configureren. Maar één store per applicatie
- Best practice:
  - Aanmaken van een **`index.js`** file in de volgende map **`src/store/`**
- We geven de rootReducer mee. Deze rootReducer kan uit meerdere reducers bestaan door middel van **`combineReducers`** (zie code)
- Eventueel een state die we uit onze localStorage halen als **`predefinedState`**

HO GENT

Redux Toolkit

configureStore (2)

- ```
  import {configureStore, combineReducers} from '@reduxjs/toolkit';
  import {reducer as counterReducer} from './counter/slice';


  const rootReducer = combineReducers({
     counter: counterReducer
     …: …
  });

  export const store = configureStore({
     reducer: rootReducer
  });
  ```

HO GENT
  ```

Redux Toolkit

createSlice

- Om een reducer te kunnen meegeven met de store moeten we natuurlijk nog onze reducers aanmaken
- Dit doen we met de **`createSlice`** methode vanuit de Redux Toolkit
- De **`createSlice`** methode maakt ons automatisch onze **action types**, **action creators** en onze **reducers** aan
- We geven een **`name`** mee, de **`initialState`** en de **`reducers`** die nodig zijn

HO GENT

Redux Toolkit

createSlice (2)

```
import { createSlice } from '@reduxjs/toolkit';

const counterSlice = createSlice({
   name: 'counter',
   initialState: { value: 0}          // Value, Object of Array
   reducers: {
        increment: state => state.value + 1;
        incrementByValue: (state, action) => {
             const { payload } = action;
             return state.value + payload;
        }
   }
});

export const {actions, reducer} = counterSlice;
export const {increment, incrementByValue} = actions;
```

Redux Toolkit

createAsyncThunk

- Alle reducers in onze store kunnen enkel maar synchrone code verwerken
- Dit betekent dus als we de store willen wijzigen met data die afkomstig is van een API we dit anders moeten oplossen
- **`createAsyncThunk`** is hiervoor de ideale methode (zie store/repos/slice.js code)
- De **`createAsyncThunk`** methode heeft twee argumenten:
  - De action type string bvb. (‘repos/fetchRepos’)
  - Een asynchrone methode waaruit een response terugkomt

HO GENT

Redux Toolkit

createAsyncThunk (2)

- ```
  import { createAsyncThunk } from '@reduxjs/toolkit';

  export const fetchRepos = createAsyncThunk('repos/fetchRepos', async () =>
  {
     const response = await axios.get('URL');
     return response;
  })

  const repoSlice = createSlice({
     name: 'repos',
     initialState: [],
     extraReducers: {
          [fetchRepos.pending]: state => state = [];
          [fetchRepos.fulfilled]: (state, action) => state =
                                                  action.payload.data;
          [fetchRepos.rejected]: state => state = [];
     }
  });
  ```

HO GENT
  ```

Provider

- Nadat onze store en onze reducers aangemaakt zijn kunnen we nu de store koppelen aan onze React applicatie
- Dit doen we door gebruik te maken van **`Provider`**, wat onderdeel is van het `react-redux` package
- Onze store wordt gekoppeld aan de root van onze React applicatie dus in de `index.js` file of waar de `ReactDOM.render` methode staat

HO GENT

Provider (2)

- ```
  import React from 'react';
  import { Provider } from 'react-redux';

  import { store } from './store';

  ReactDOM.render(
     <React.StrictMode>
         <Provider store={store}>
             <App />
         </Provider>
     </React.StrictMode>
  );
  ```
  ```

React-Redux

useSelector

- Om in een component onze state te kunnen uitlezen maken we gebruiken van de **`useSelector`** Hook van het `react-redux` package
- Met de useSelector hook krijgen we de volledige state terug en wordt onze app gererenderd moesten er wijzigingen zijn in onze data

HO
GENT

React-Redux

useSelector (2)

- ```
  import { useSelector } from 'react-redux';

  const HookComponent = () => {

     const state = useSelector();
     const counterState = useSelector(state => state.counter);
     const testState = useSelector(state => state.test);

     return (

             {counterState.value}

     )
  }
  ```

HO GENT
  ```

React-Redux

useDispatch

- Om onze actions te kunnen dispatchen naar onze store kunnen we gebruik maken van de **`useDispatch`** Hook van het `react-redux` package
- Dispatch wordt gebruik om onze action creator op te roepen op de store
- Deze dispatch kan ook gebruikt worden om een AsyncThunk action creator op te roepen

HO GENT

React-Redux

useDispatch (2)

```
import { useDispatch } from 'react-redux';
import { addOne } from '../store/test/slice';

const HookComponent = () => {

   const dispatch = useDispatch();

   return (
       <div>
           <button onClick={() => dispatch(addOne('one'))} >
               Add one
           </button>
       </div>
   )
}
```

HO GENT