Redux reducer actions

Pavel Lasarev
paullasarev@gmail.com

Agenda

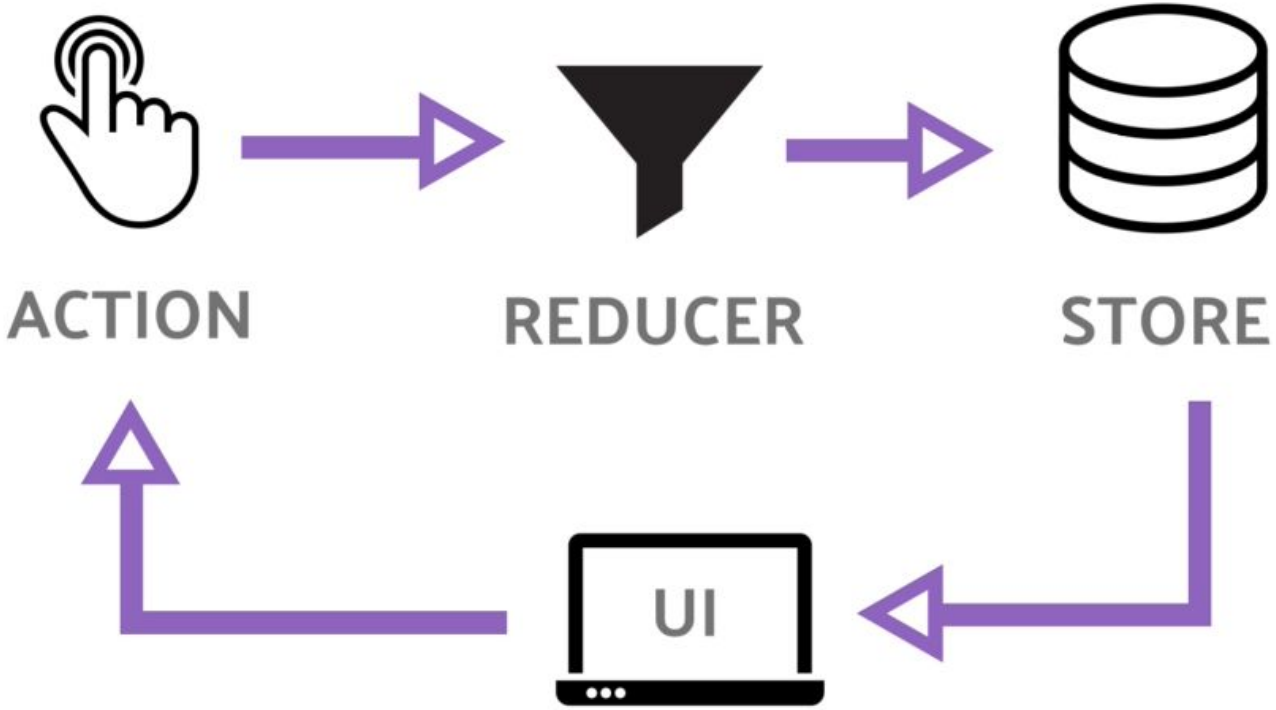
- redux flow
- redux-saga effects
- redux-saga drawbacks
  - plumber code
  - unit testing is awful
- example task: load extra info from API
  - redux-saga vs redux-reducer-actions
- redux-reducer-actions
  - reducer actions
  - use root state
  - attach to store
  - options
- redux-reducer-actions in production
- useful redux libs
- questions?

Redux-saga

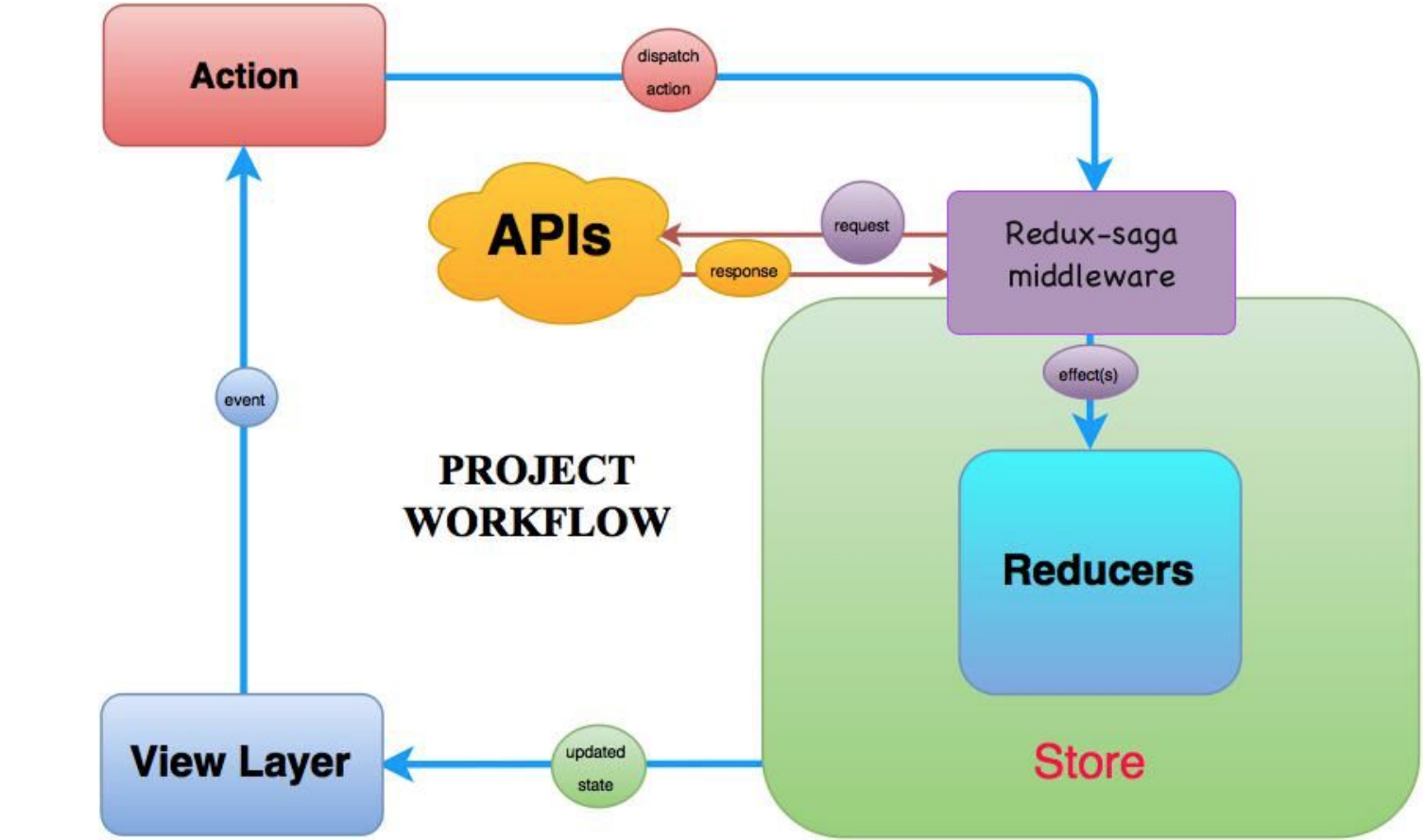
redux-saga is a library that aims to make application side effects (i.e. asynchronous things like data fetching and impure things like accessing the browser cache) easier to manage, more efficient to execute, easy to test, and better at handling failures.

The mental model is that a saga is like a separate thread in your application that's solely responsible for **side effects**. redux-saga is a redux middleware, which means this thread can be started, paused and cancelled from the main application with normal redux actions, it has access to the full redux application state and it can dispatch redux actions as well

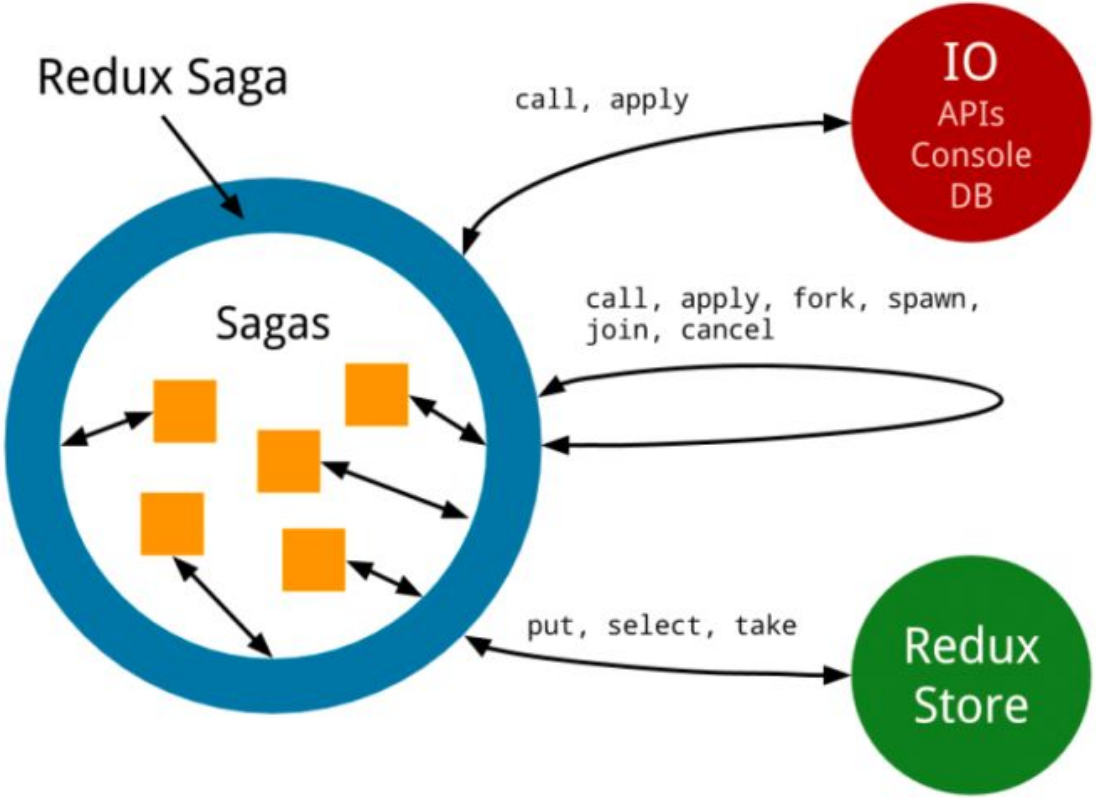
Redux flow



redux-saga flow



redux-saga


Redux Saga
call, apply
IO
APIs
Console
DB
Sagas
call, apply, fork, spawn,
join, cancel
put, select, take
Redux
Store

Redux code (common)

```
// action creators
export const GET_ITEM = 'GET_ITEM';
export const GET_ITEM_SUCCESS = GET_ITEM_SUCCESS';

export const getItem = (id) => ({
  type: GET_ITEM,
  request: {
     url: `/api/item/${id}`,
     method: 'GET',
     meta: { id },
  },
})
```

```
export const GET_FILE = 'GET_FILE';
export const GET_FILE = 'GET_FILE_SUCCESS';

export const getFile = (id) => ({
  type: GET_FILE,
  request: {
     url: `/api/file/${id}`,
     method: 'GET',
     meta: { id },
  },
})
```

redux-saga code

```
// reducer
export const reducer(state, action) {
  switch(action.type) {
     case GET_ITEM_SUCCESS: {
       return {
         ..state,
         item: action.payload,
       };
     }
     case GET_FILE_SUCCESS: {
       return {
         ..state,
         file: action.payload,
       };
     }
     default:
       return state;
  }
}
```

```
// effects

export function* onGetItem(action) {
  const { payload: { fileId } } = action;
  yield put(getFile(fileId));
}

export function* itemSaga() {
  takeEvery(GET_ITEM_SUCCESS, onGetItem);
}

// root saga
export function* rootSaga() {
  spawn(itemSaga);
}
```

redux-saga unit tests

```
define('item saga', ()=>{
  const action = {
    type: GET_ITEM_SUCCESS,
    payload: { fileId: 42 },
  };

  it ('should process GET_ITEM_SUCCESS', ()=> {
    const gen = itemSaga();
    const getState = gen.next().value;
    const nextState = fork(takeEvery, GET_ITEM_SUCCESS, onGetItem);
    expect(getState).toEqual(nextState);
  });

  it ('should fire GET_FILE on GET_ITEM_SUCCESS', ()=> {
    const gen = onGetItem(action);
    const getState = gen.next().value;
    const nextState = put(getFile(fileId));
    expect(getState).toEqual(nextState);
  });
});
```

Cons:

- tests are extremely verbose
- tests are state oriented
- tests are on the same abstract layer as the code

redux-reducer-actions

```
// reducer
export const reducer(state, action) {
  switch(action.type) {
     case GET_ITEM_SUCCESS: {
       const { payload: { fileId } } = action;
       const actions = [getFile(fileId)];
       return {
         ..state,
         item: action.payload,
         actions,
       };
     }
     case GET_FILE_SUCCESS: {
       return {
         ..state,
         file: action.payload,
       };
     }
     default:
       return state;
  }
}
```

```
// unit tests

define('reducer', ()=>{
  const action = {
    type: GET_ITEM_SUCCESS,
    payload: { fileId: 42 },
  }
  it ('should fire GET_FILE on GET_ITEM_SUCCESS',
    ()=> {
    const state = {};
    const newState = reducer(state, action);
    expect(state.actions).toBeDefined();
    expect(state.actions.length).toBe(1);
    expect(state.actions[0]).toEqual(getFile(42));
  });
});
```

redux-reducer-actions: attach to store

```
// createStore
const sagaMiddleware = createSagaMiddleware();

const middlewares = [sagaMiddleware];
// additional middlewares
// ...
const enhancer = applyMiddleware(...middlewares);

const actionEnchancer = createActionsEnhancer({});

const store = createStore(rootReducer, compose(actionEnchancer, enhancer));

sagaMiddleware.run(rootSaga);

export default function configureStore() {
  return {
    store: {
      ...store,
      runSaga: sagaMiddleware.run,
    },
  };
}
```

redux-reducer-actions: options

```
// log
// log will be used to verbose procecced actions
const isDev = process.env.NODE_ENV !== 'production';
const actionEnchancer = createActionsEnhancer({ log: isDev ? console.log.bind(console)
: null });

// startActionType
// all actions which was fired before the startAction will be queried
//     and processed AFTER the start action
const actionEnchancer = createActionsEnhancer({ startActionType: AUTH_SUCCESS });

// schedule will be used to fire actions in new event loop
// default is window.setTimeout
const actionEnchancer = createActionsEnhancer({ schedule: window.setTimeout });
```

redux-reducer-actions in production

- 2 pet projects
- 2 production projects

Useful redux libs

- combine-section-reducers
- connected-react-router
- redux-persist
- reselect
- redux-saga-requests

combine-section-reducers: Use root state

```
// reducer
export const reducer(state, action, rootState) {
  switch(action.type) {
     case GET_ITEM_SUCCESS: {
       const { payload: { fileId } } = action;
       const { user: { language } } = rootState;
       const actions = [getFile(fileId, language)];
       return {
         ..state,
         item: action.payload,
         actions,
       };
     }
     case GET_FILE_SUCCESS: {
       return {
         ..state,
         file: action.payload,
       };
     }
     default:
       return state;
  }
}
```

```
import combineSectionReducers from
  'combine-section-reducers';

export rootReducer =
combineSectionReducers({
  user,
  item,
  //...
});
```

Questions?

https://github.com/paullasarev/redux-reducer-actions.git