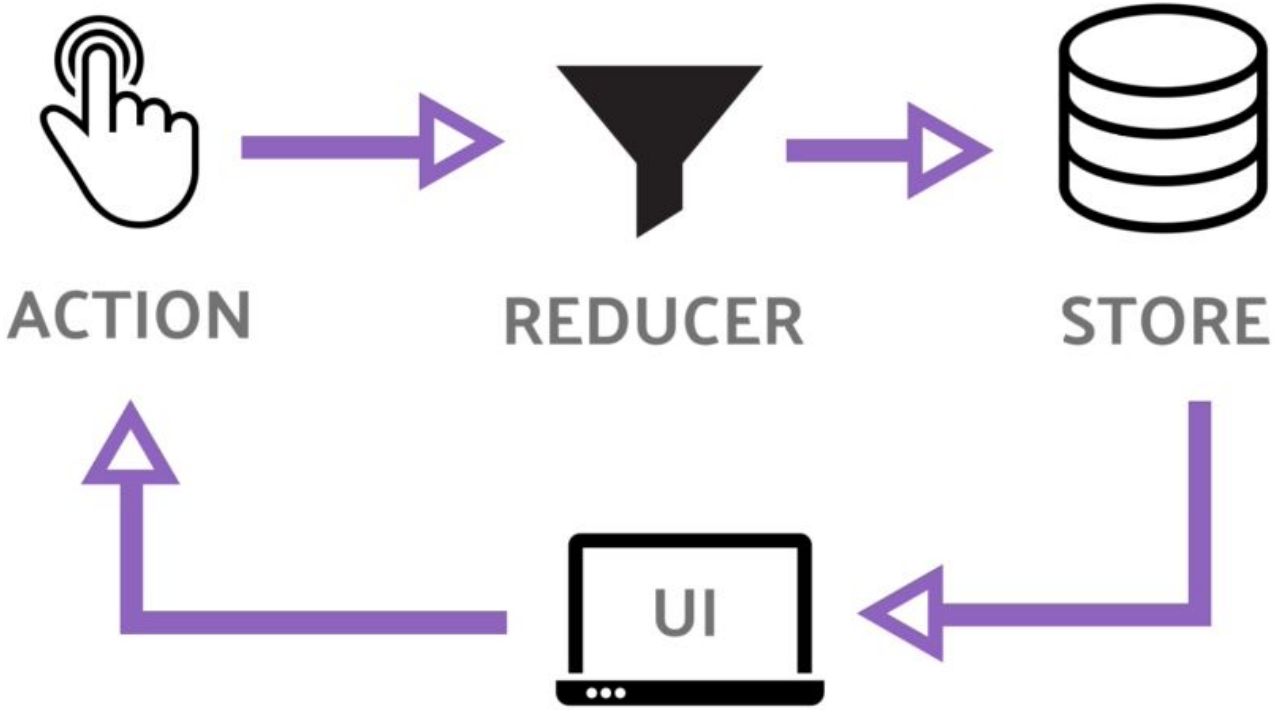
Redux reducer actions

Pavel Lasarev
paullasarev@gmail.com

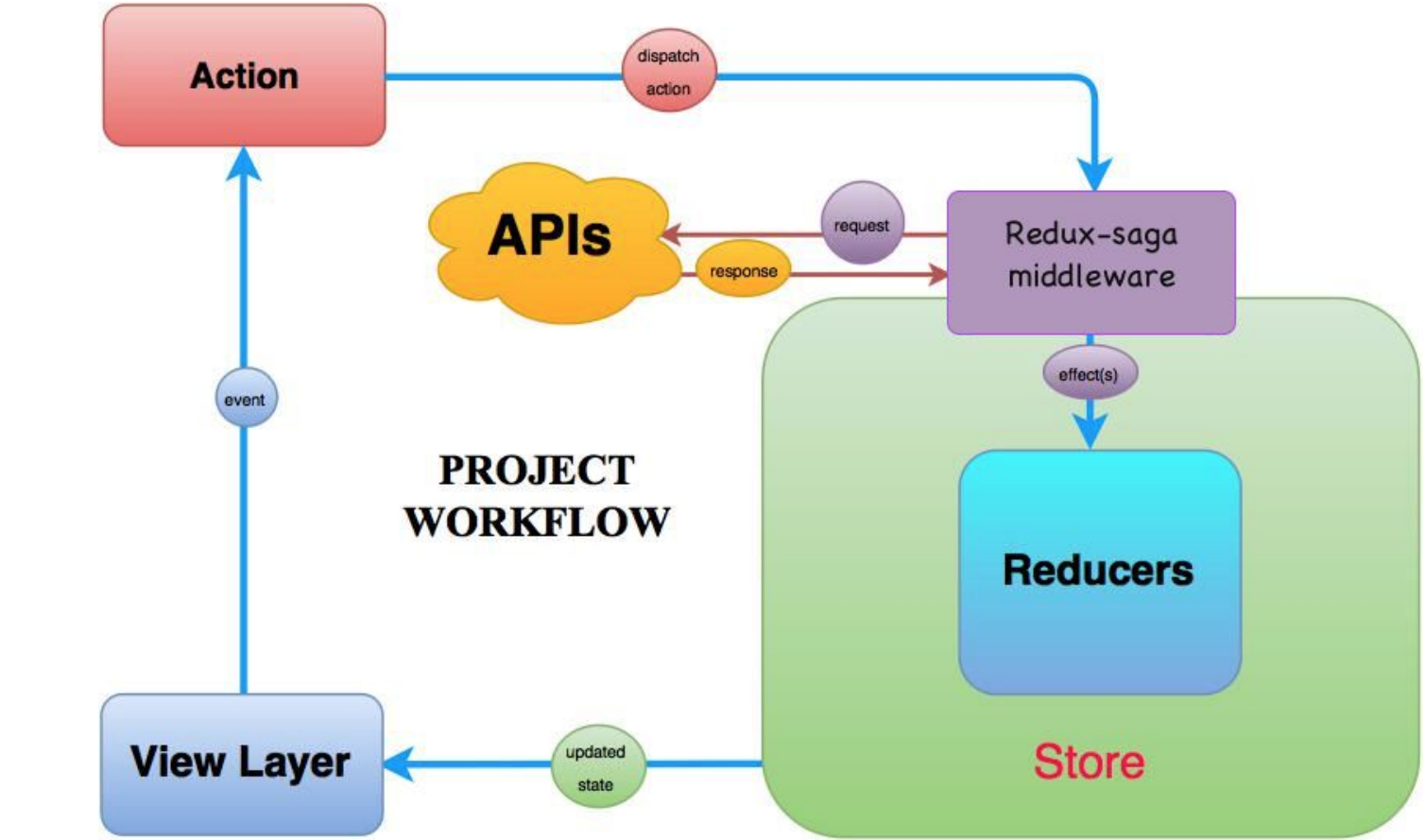
Agenda

- 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
  - attach to store
  - options
- redux-reducer-actions in production
- useful redux libs
- questions?

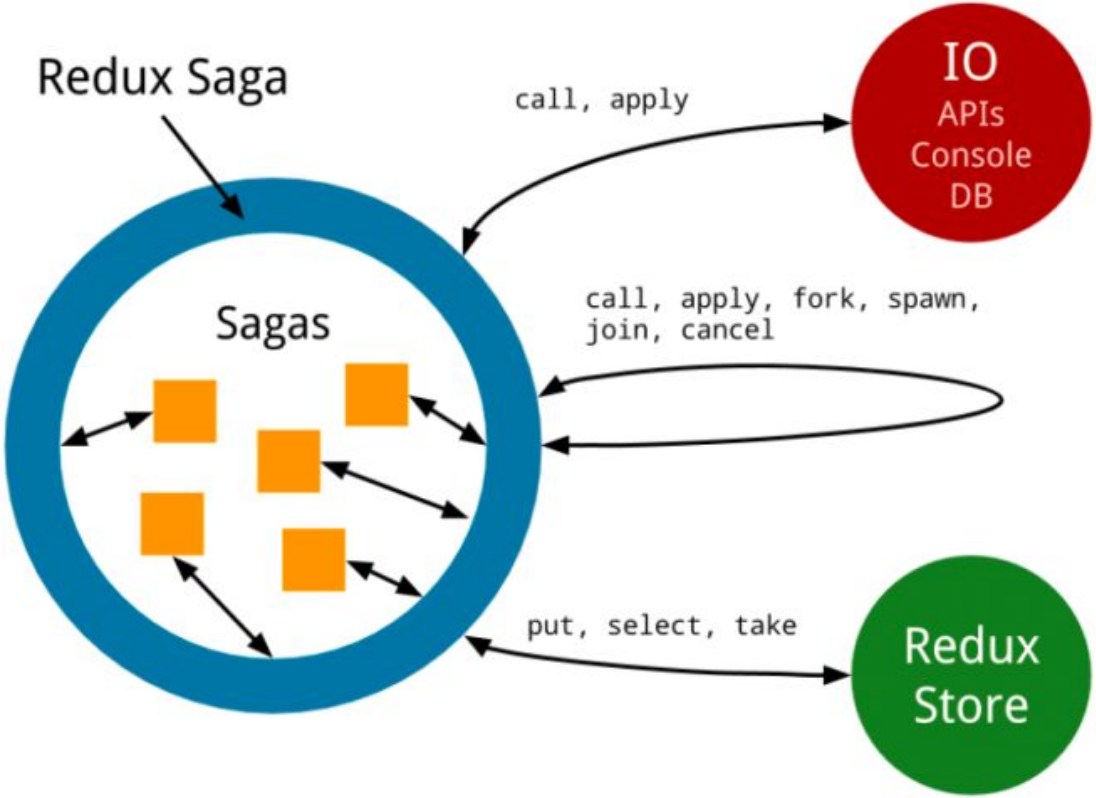
Redux flow



redux-saga flow



redux-saga

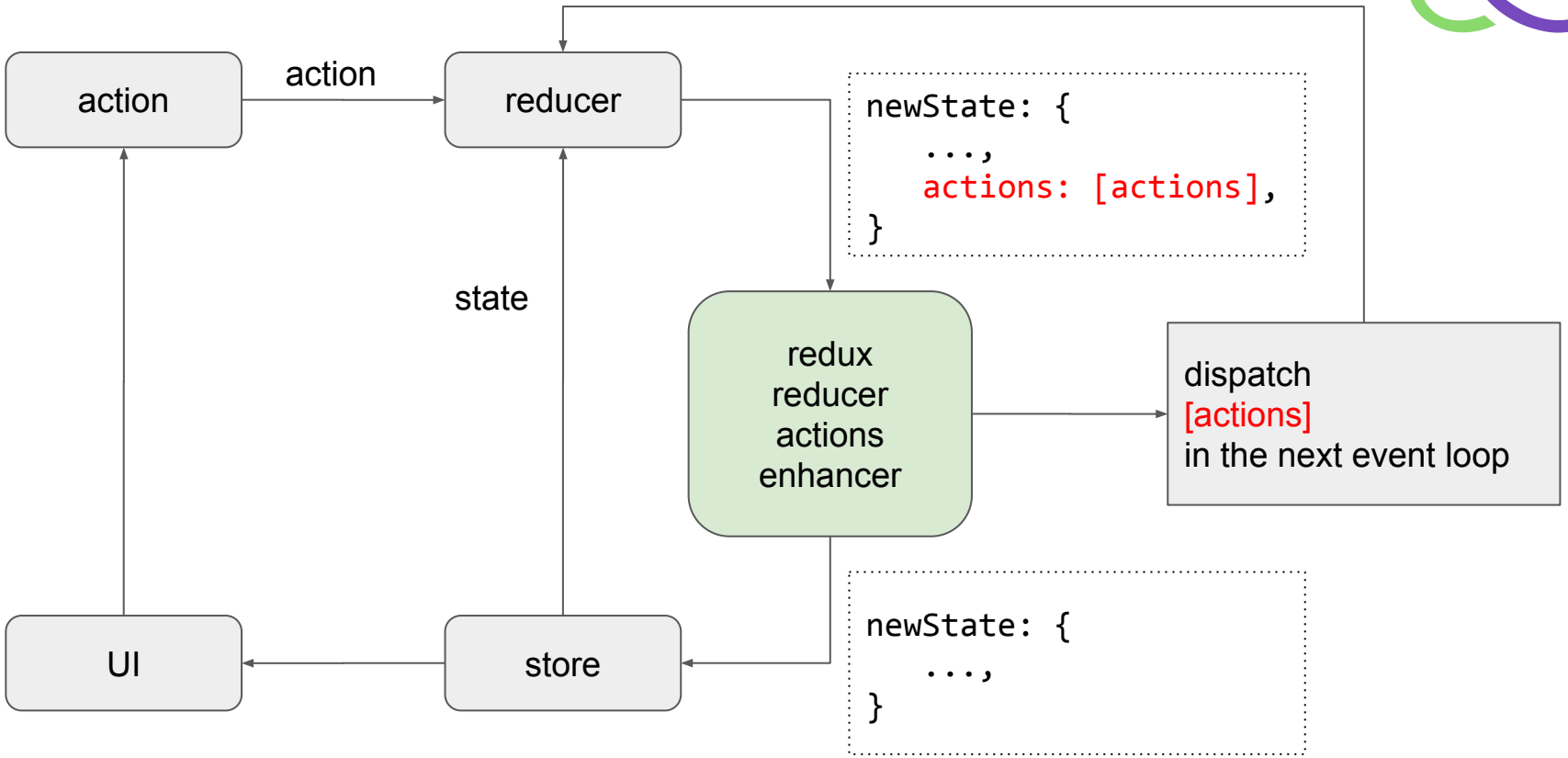

Redux Saga
call, apply
IO
APIs
Console
DB
Sagas
call, apply, fork, spawn,
join, cancel
put, select, take
Redux
Store

Redux-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-reducer-actions flow



redux-reducer-actions

redux-reducer-actions is an Redux store enhancer which allows action generation in reducer. That approach allows to concentrate most logic in one place - reducer.

Due to that fact that reducers are pure functions, it is extremely easy to test this logic and keep code clean and concise.

Actions, processed by **redux-reducer-actions** wouldn't go into the **store**. Rather they will be dispatched in the next event loop throw the standard Redux flow.

Example task: generate an API action
The redux-saga way

Common redux code: actions creators

```
// 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() {
  yield takeEvery(GET_ITEM_SUCCESS,
onGetItem);
}

// root saga
export function* rootSaga() {
  yield 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

Example task: generate an API action
The redux-reducer-actions way

redux-reducer-actions code and tests

```
// 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));
  });
});
```

VS

```
// 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() {
  yield takeEvery(GET_ITEM_SUCCESS, onGetItem);
}

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

```
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);
  });
});
```

```
// 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 usage

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 });
```

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;
  }
}
```

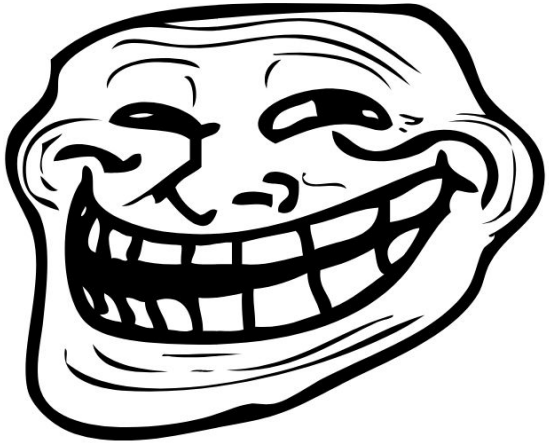
```
import combineSectionReducers from
  'combine-section-reducers';


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

Summary

- redux-reducer-actions pros:
  - allows to concentrate most logic in one place (reducer)
  - extremely easy to test action generation logic
  - keep code clean and concise
- usage in production:
  - 2 pet projects
  - 2 production projects

To dispatch or not not to dispatch?



> Isn't it incorrect to cause side-effects in a reducer?

Yes! Absolutely.

> Doesn't redux-reducer-actions put side-effects in the reducer?

It doesn't. The values returned from the reducer when scheduling an effect only *describe* the effect. Calling the reducer will not cause the effect to run. The value returned by the reducer is just an object that the store knows how to interpret when it is enhanced. You can safely call a reducer in your tests without worrying about waiting for effects to finish and what they will do to your environment.

Questions?

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

Additional materials: useful redux libs

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