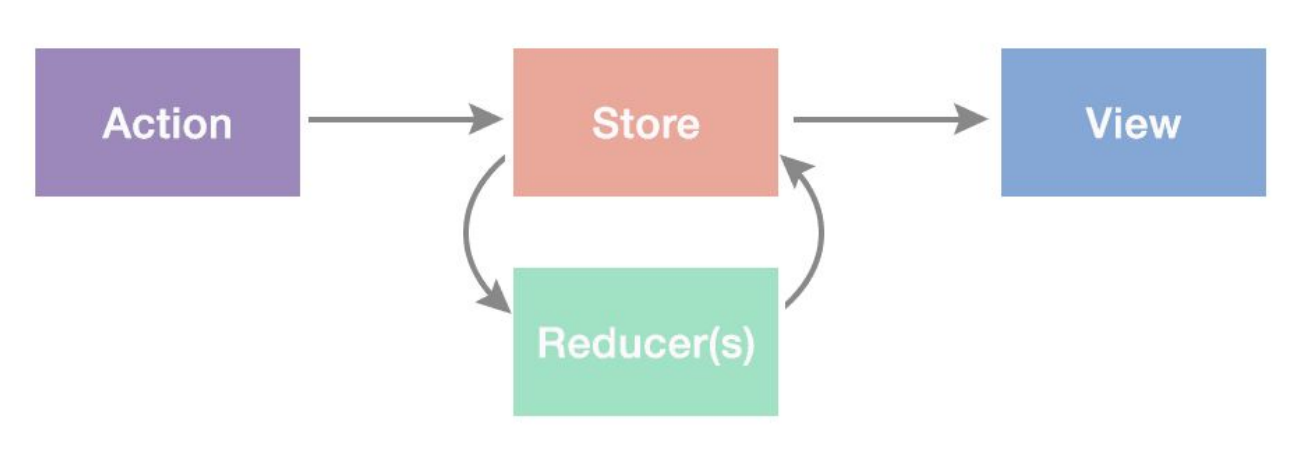
React third party libs.

adapt

Redux

Redux is a predictable state container for JavaScript apps.

It helps you write applications that behave consistently, run in different environments (client, server, and native), and are easy to test.



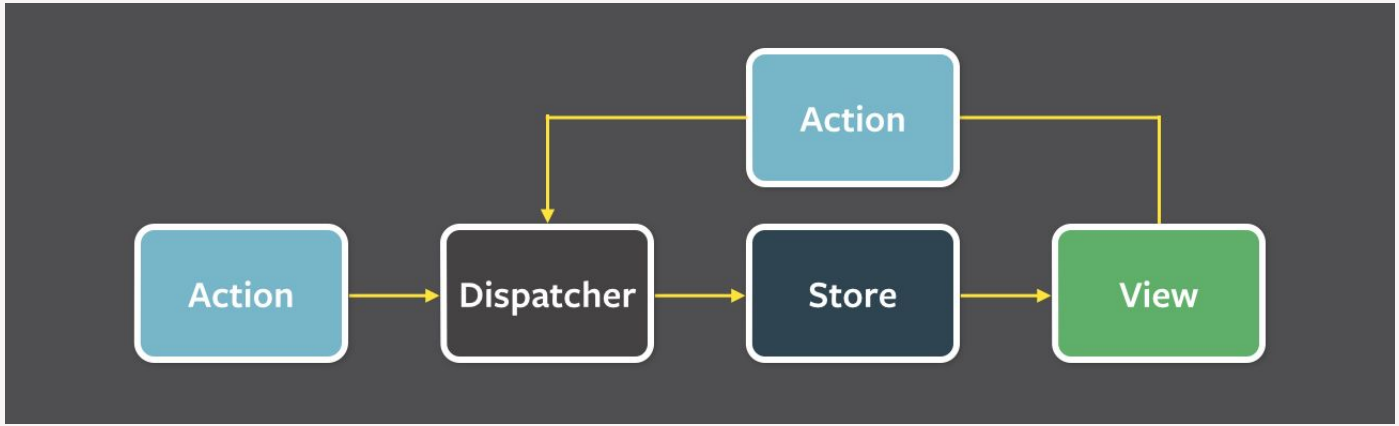
adapt

Redux basically are

Its “global” variable managing center, which notify views about any changes. Also its ensure unidirectional data flow, which prevents undesirable behaviour and changes is “variable” state.

Redux Influences

Redux evolves the ideas of Flux, but avoids its complexity by taking cues from Elm. Even if you haven't used Flux or Elm, Redux only takes a few minutes to get started with.



adapt

Redux Actions

Actions are payloads of information that send data from your application to your store. They are the only source of information for the store. You send them to the store using store.dispatch().

```
{
  type: ADD_TODO,
  text: 'Build my first Redux app'
}
```

adapt

Redux Action Creators

Action creators are functions that create actions.

```
function addTodo(text) {
  return {
    type: ADD_TODO,
    text
  }
}
```

Redux Reducers

Reducers specify how the application's state changes in response to actions sent to the store. Remember that actions only describe what happened, but don't describe how the application's state changes.

```
function todoApp(state = initialState, action) {
  switch (action.type) {
    case SET_VISIBILITY_FILTER:
      return Object.assign({}, state, {
        visibilityFilter: action.filter
      })
    default:
      return state
  }
}
```

adapt

Redux Reducers - Rules

1. We don't mutate the state.
2. We return the previous state in the default case. It's important to return the previous state for any unknown action.

Redux Store

The Store is the object that brings them together. The store has the following responsibilities:

- Holds application state;
- Allows access to state via getState();
- Allows state to be updated via dispatch(action);
- Registers listeners via subscribe(listener);
- Handles unregistering of listeners via the function returned by subscribe(listener).

Example

Exercise

React Router

React Router is a collection of navigational components that compose declaratively with your application.

React Router: Routers

At the core of every React Router application should be a router component. For web projects, react-router-dom provides <BrowserRouter> and <HashRouter> routers. Both of these will create a specialized history object for you. Generally speaking, you should use a <BrowserRouter> if you have a server that responds to requests and a <HashRouter> if you are using a static file server.

React Router: Routers

```
import { BrowserRouter } from "react-router-dom";
ReactDOM.render(
  <BrowserRouter>
    <App />
  </BrowserRouter>,
  holder
);
```

React Router: Matching

There are two route matching components: <Route> and <Switch>.

Route matching is done by comparing a <Route>'s path prop to the current location’s pathname. When a <Route> matches it will render its content and when it does not match, it will render null. A <Route> with no path will always match.

You can include a <Route> anywhere that you want to render content based on the location. It will often make sense to list a number of possible <Route>s next to each other. The <Switch> component is used to group <Route>s together.

React Router: Routers

```
import { Route, Switch } from "react-router-dom";

// when location = { pathname: '/about' }
<Route path='/about' component={About}/> // renders <About/>
<Route path='/contact' component={Contact}/> // renders null
<Route component={Always}/> // renders <Always/>

<Switch>
  <Route exact path="/" component={Home} />
  <Route path="/about" component={About} />
  <Route path="/contact" component={Contact} />
</Switch>

<Switch>
  <Route exact path="/" component={Home} />
  <Route path="/about" component={About} />
  <Route path="/contact" component={Contact} />
  {/* when none of the above match, <NoMatch> will be rendered */}
  <Route component={NoMatch} />
</Switch>
```

adapt

React Router: Rendering Props

You have three prop choices for how you render a component for a given <Route>: **component**, **render**, and **children**.

Component should be used when you have an existing component (either a React.Component or a stateless functional component) that you want to render. **Render**, which takes an inline function, should only be used when you have to pass in-scope variables to the component you want to render. You should not use the component prop with an inline function to pass in-scope variables because you will get undesired component unmounts/remounts.

React Router: Rendering Props

```
const Home = () => <div>Home</div>;

const App = () => {
  const someVariable = true;

  return (
    <Switch>
      {/* these are good */}
      <Route exact path="/" component={Home} />
      <Route
        path="/about"
        render={props => <About {...props} extra={someVariable} />}
      />
      {/* do not do this */}
      <Route
        path="/contact"
        component={props => <Contact {...props} extra={someVariable} />}
      />
    </Switch>
  );
};
```

adapt

React Router: Navigation

React Router provides a <Link> component to create links in your application. Wherever you render a <Link>, an anchor (<a>) will be rendered in your application's HTML.

The <NavLink> is a special type of <Link> that can style itself as "active" when its to prop matches the current location.

Any time that you want to force navigation, you can render a <Redirect>. When a <Redirect> renders, it will navigate using its to prop.

React Router: Navigation

```
<Link to="/">Home</Link>
// <a href='/'>Home</a>


// location = { pathname: '/react' }
<NavLink to="/react" activeClassName="hurray">
  React
</NavLink>
// <a href='/react' className='hurray'>React</a>

<Redirect to="/login" />
```

adapt

Example

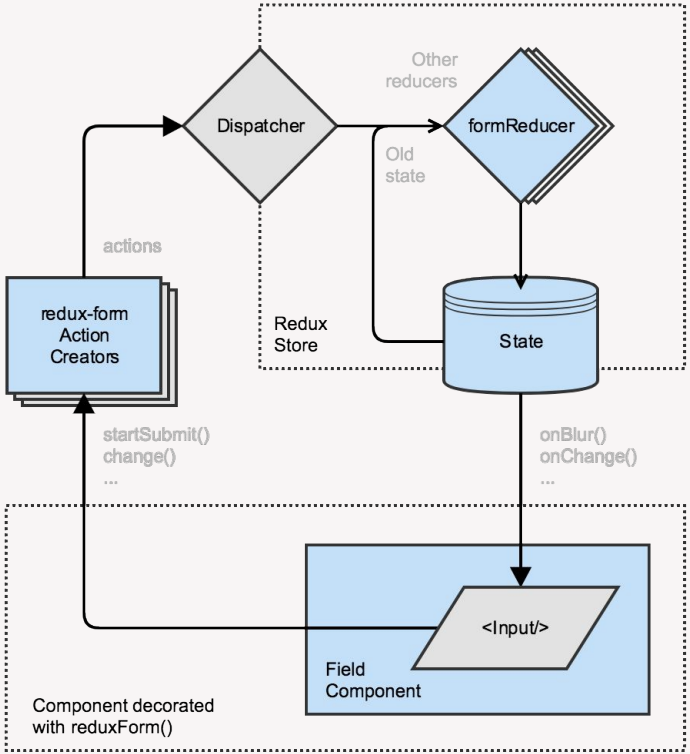
Exercise

Redux Form

Easy way to store your form data in store and control all its aspects and ensuring unidirectional data flow.

It also provides: hooks for validation and formatting handlers, various properties and action creators.

Redux Form



1. User clicks on the input,
2. "Focus action" is dispatched,
3. formReducer updates the corresponding state slice,
4. The state is then passed back to the input.

adapt

Redux Form: Reducer

The store should know how to handle actions coming from the form components. To enable this, we need to pass the formReducer to your store. It serves for all of your form components, so you only have to pass it once.

```
import { createStore, combineReducers } from 'redux'
import { reducer as formReducer } from 'redux-form'

const rootReducer = combineReducers({
  // ...your other reducers here
  // you have to pass formReducer under 'form' key,
  // for custom keys look up the docs for 'getFormState'
  form: formReducer
})

const store = createStore(rootReducer)
```

adapt

Redux Form: Form component

To make your form component communicate with the store, we need to wrap it with reduxForm(). It will provide the props about the form state and function to handle the submit process.

```
import React from 'react'
import { Field, reduxForm } from 'redux-form'

let ContactForm = props => {
  const { handleSubmit } = props
  return <form onSubmit={handleSubmit}>{/* form body*/}</form>
}

ContactForm = reduxForm({
  // a unique name for the form
  form: 'contact'
})(ContactForm)

export default ContactForm
```

adapt

Redux Form: Field component

The <Field/> component connects each input to the store. The basic usage goes as follows:

```
<Field name="inputName" component="input" type="text" />
```

adapt

Redux Form: Submitting

```
import React from 'react'
import ContactForm from './ContactForm'

class ContactPage extends React.Component {
  submit = values => {
    // print the form values to the console
    console.log(values)
  }
  render() {
    return <ContactForm onSubmit={this.submit} />
  }
}
```

Example

Exercise

Questions?