УДК 604.5

ИСПОЛЬЗОВАНИЕ JAVASCRIPT-БИБЛИОТЕКИ REDUX ДЛЯ РАЗРАБОТКИ МОБИЛЬНОГО ПРИЛОЖЕНИЯ

К.Е. Колобова, студентка
Научный руководитель – А.А. Тайлакова, ст. преподаватель
Кузбасский государственный технический университет
имени Т. Ф. Горбачёва, г. Кемерово

По мере того, как ваше приложение начинает расширяться, становится всё сложнее следить за текущим его состоянием, отслеживать рендеринг компонентов, следить за потоком изменяющихся данных. Несмотря на то, что в React есть собственный метод управления состоянием, он плохо масштабируется: для того, чтобы получить ответную реакцию на какое-либо событие от независимых компонентов приложения, приходится либо передавать локальное состояние в виде пропсов (props = свойства) дочерним компонентам, либо поднимать его вверх до ближайшего родительского компонента. В обоих случаях делать это не удобно. Код становится грязным и трудночитаемым, а компоненты – зависимыми от нагроможденной вложенности. Это дает повод воспользоваться дополнительными решениями. Одно из таких решений – это библиотека Redux, при использовании которой все состояние приложения становится легкодоступным для всех его компонентов. Что же она из себя представляет?

Redux – это менеджер состояний приложения. В большинстве случаев в связке с ним используют React, однако он не привязан непосредственно к этой библиотеке и может также использоваться с другими JavaScript-библиотеками и фреймворками. Важно отметить, что Redux не является обязательной составляющей вашего приложения, особенно в том случае, если оно небольшое.

Библиотека Redux основана на нескольких концепциях, изучив которые можно гораздо проще решать проблемы с состоянием приложения. Давайте перейдем к их рассмотрению.

Store (хранилище). Это объект, который хранит состояние приложения. Данные из этого объекта могут быть отправлены в любой компонент по требованию.

Пример создания и инициализации хранилища:

```
const rootReducer = combineReducers({
    post: postReducer
})

export default createStore(rootReducer, applyMiddleware(thunk))
```

Пример обновления состояния:

```
const thisPost = {
      date: new Date().toJSON(),
      text: text,
      img: imgRef.current,
      booked: false
    }
    dispatch(addPost(thisPost))
```

Action (действие). Это объект, принимающий в себя свойства, главное из которых – `type`. Action обозначает событие, описывающее, что именно должно произойти. Для передачи информации из этого объекта используется метод `dispatch()`.

```
dispatch({
      type: REMOVE_POST,
      payload: id
})
```

Здесь `REMOVE_POST` является заранее заготовленной константой, импортированной из другого файла:

```
export const REMOVE_POST = 'REMOVE_POST'
```

Action creators (генераторы действий). Это функции, которые создают действия. Они описывают, что произошло, но не описывают то, как изменится состояние приложения.

Например, в приведенной ниже функции `removePost` отправляется запрос к локальной базе данных на удаление элемента по полученному идентификатору.

```
export const removePost = id => async dispatch => {
    await DB.removePost(id)
    dispatch({
        type: REMOVE_POST,
        payload: id
    })
}
```

Reducer (редуктор). Это одна или более функций, которые определяют, каким образом должно измениться состояние приложения в ответ на полученные действия (action). Редуктор не мутирует существующее состояние, а возвращает совершенно новый объект дерева состояний, которым заменяется предыдущий. Редуктор – это чистая функция, поэтому он не должен иметь какие-либо побочные эффекты, а также вызывать функции, результат которых зависит от чего-то кроме их собственных аргументов, например «`new Date.now()`».

Пример использования редуктора в конструкции switch-case:

```
const initialState = {
    allPosts: [],
    bookedPosts: [],
    loading: true
}
export const postReducer = (state = initialState, action) => {
  switch (action.type) {
    case REMOVE_POST:
        return {
            ...state,
            allPosts: state.allPosts.filter(p =>
                    p.id !== action.payload
             ),
            bookedPosts: state.bookedPosts.filter(p =>
                    p.id !== action.payload
             )
        }
    default:
        return state
  }
}
```

Представления (View) – они же компоненты пользовательского интерфейса (UI Components) – отвечают за вывод информации на экран в графическом виде. Это могут быть как полностью сверстанные экраны, так и отдельные элементы интерфейса. Например, таким компонентом может выступать значок для экрана загрузки, который можно будет использовать неоднократно:

```
export const AppLoader = () => {
    return(
        <View style={styles.center}>
            <ActivityIndicator size="large" color={APP_COLORS.CYPRUS} />
        </View>
    )
}
const styles = StyleSheet.create({
    center: {
        flex: 1,
        justifyContent: 'center',
        alignItems: 'center'
    }
})
```

Или стилизованное поле для ввода. Изначально этот компонент имеет свой стиль, однако при вызове, его можно дополнить или даже полностью изменить:

```
export const AppText = props => (
      <Text style={{ ...styles.default, ...props.style }}>
            {props.children}
      </Text>
)
const styles = StyleSheet.create({
    default: {
        fontSize: 14,
        color: APP_COLORS.BLACK,
    }
})
```

View не меняют данные напрямую. При возникновении какого-либо события, или при взаимодействии пользователя с ними, они обращаются к создателям действий (action creators).

Таким образом организуется однонаправленный поток данных. Это хорошо демонстрирует рис. 1.

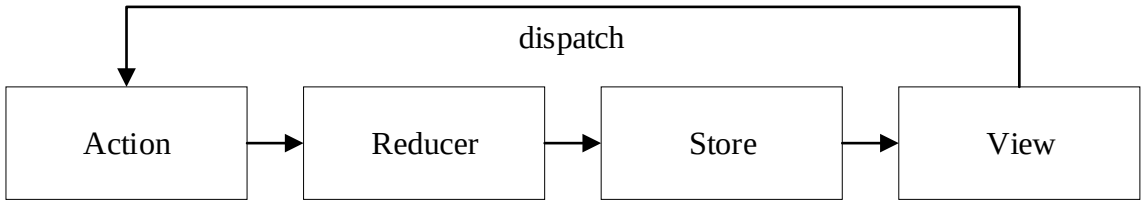

Рисунок 1 Взаимодействие элементов архитектуры Redux

Важное преимущество использования Redux – это упрощение тестирования приложения. Учитывая независимость каждого из компонентов, каждый блок приложения можно подвергнуть независимому модульному тестированию. Ведь функция, которая представляет такой компонент, всегда возвращает одно и то же представление для одних и тех же данных, что делает поведение приложения предсказуемым и, следовательно, легко поддающимся контролю при отслеживании ошибок. Однако не только компоненты пользовательского интерфейса могут быть подвергнуты независимому тестированию, но и редукторы, а также создатели действий.

Список литературы:

1. Udemy – Курс «React Native 2020. Мобильные приложения на JavaScript» [Электронный ресурс] – Режим доступа: https://www.udemy.com/course/react-native-complete-guide/, свободный (дата обращения: 07.02.2021).
2. Репозиторий на GitHub: Redux [Электронный ресурс] – Режим доступа: https://github.com/reduxjs/redux, свободный (дата обращения: 25.03.2021).
3. Redux – API Reference [Электронный ресурс] – Режим доступа: https://redux.js.org/api/api-reference, свободный (дата обращения: 25.03.2021).
4. METANIT.COM – введение в Redux [Электронный ресурс] – Режим доступа: https://metanit.com/web/react/5.3.php, свободный (дата обращения: 25.03.2021).