САНКТ-ПЕТЕРБУРГСКИЙ НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ ИТМО

Дисциплина: Фронт-энд разработка

Отчет
Лабораторная Работа №3
“Реализация приложения на ReactJS”

Выполнил:
Стукалов Артем

Группа
K33392

Проверил:
Добряков Д. И.

Санкт-Петербург

2023 г.

Задача

Мигрировать ранее написанный сайт на фреймворк Vue.JS. Минимальные требования:

- Должен быть подключён роутер
- Должна быть реализована работа с внешним API
- Разумное деление на компоненты

Ход работы

В рамках данной работы будет рассмотрена немного другая задача. Разберем как написать базовое приложение на React JS со следующими аспектами:

1) Самописный стор наподобие redux с возможностью асинхронных эффектов и интеграцией с внешними зависимостями
2) Реализация работы с внешними зависимостями. Как пример будет рассмотрена удобная обертка для работы с сетью

Общие требования

Дабы не создавать “приложение-оркестр” позаботимся следующими требованиями:

1) Приложение может быть интегрировано в любое другое
2) Есть возможность управлять сторон приложения из вне приложения
3) Приложение не должно самостоятельно ходить в сеть, а делать это лишь через проксирующий класс. Так мы сможем самостоятельно управлять трафиком приложения.
4) Если реализация какой-то части приложения не зависит от выбранного фреймворка, такие части должны быть реализованы вне приложения и переданы в него с использованием dependency инъекций.

Самописный стор

Для начала разберемся с минимальным набором определений.

State - общее состояние всего стора. ПРедставляет из себя обычный объект. Является и мутабельным.

Dispatch - функция позволяющая вызвать какое-то действие(action)

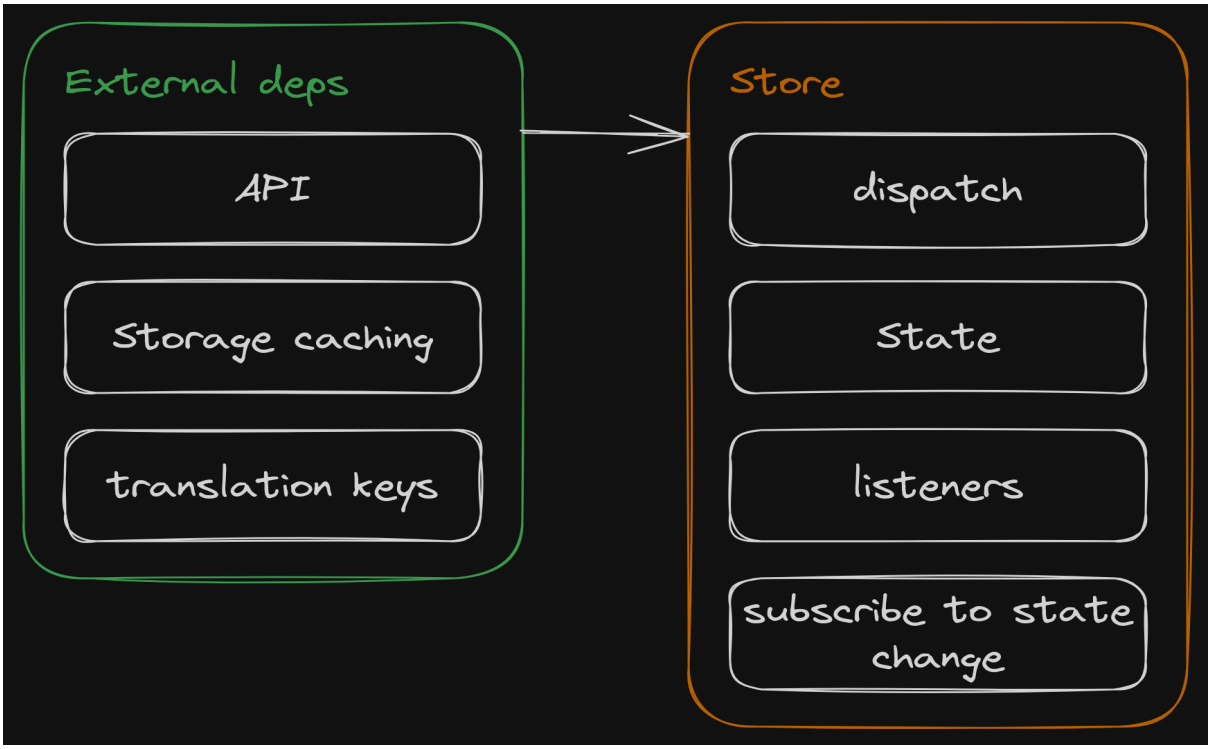
Action - действие, которое может изменять значение стейта и вызывать другие действия используя эффекты.

Effect - асинхронная функция, в рамках которой имеется доступ к внешним модулям. Может вернуть набор действий, которые будут выполнены после завершения эффекта.

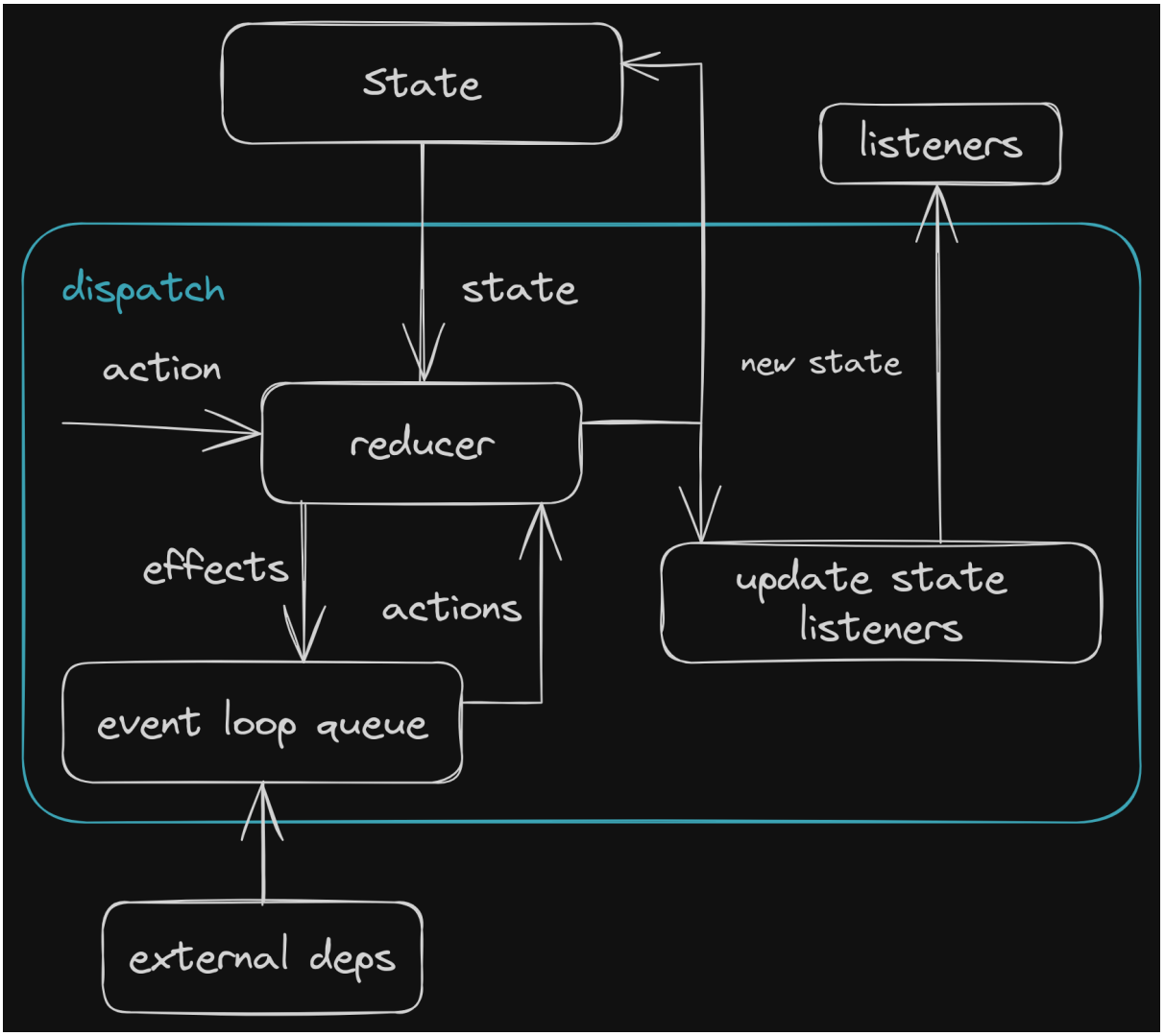
StateSelector - react хук, позволяющий считать часть данных из стейта и подписаться на изменения только этой части стейта.

ExternalDeps - внешний набор зависимостей, базовую реализацию которых приложение предоставляет, но она может быть расширена и дополнена. В дальнейших примерах будет подробно разобран модуль API.

Базово схема работы стора выглядит следующим образом:

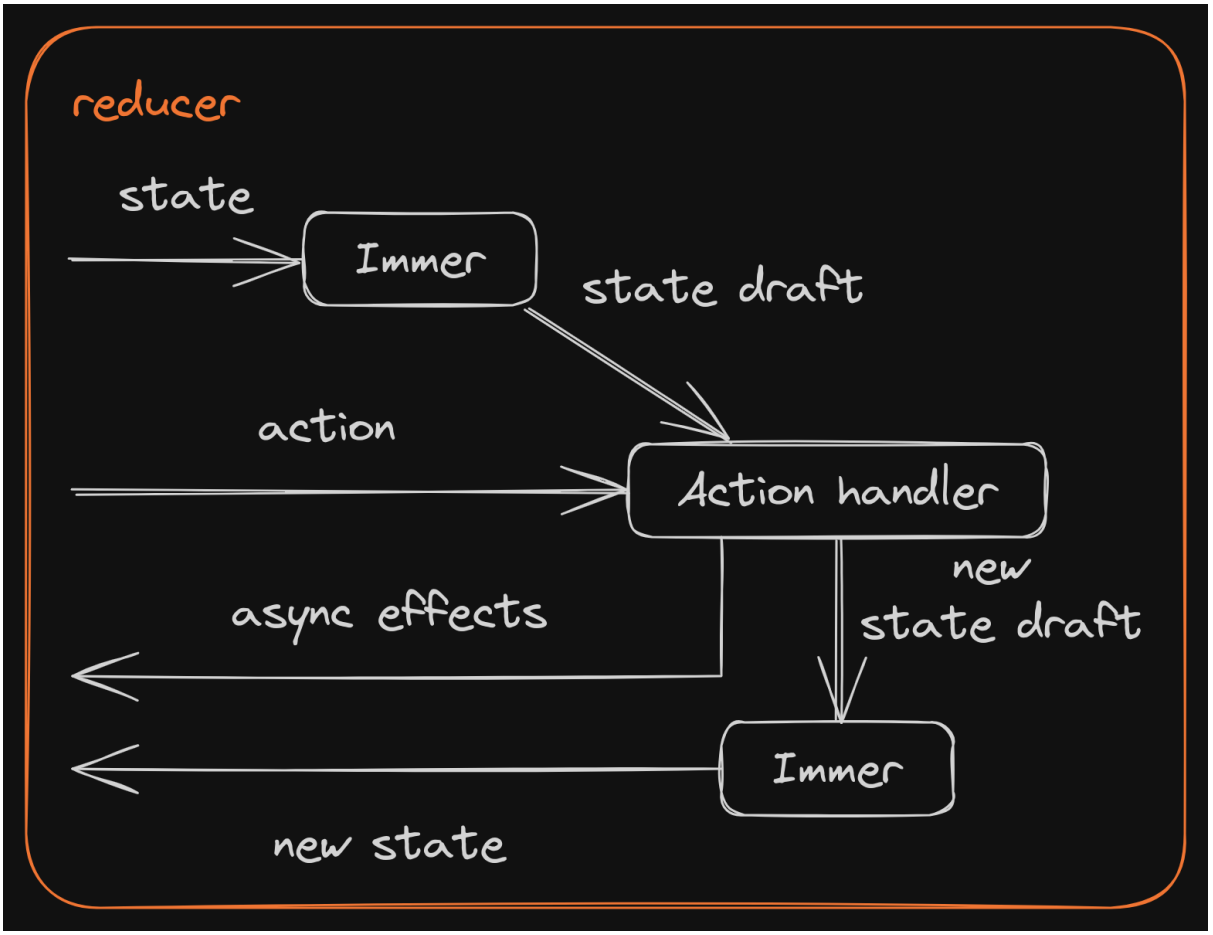


Стор предоставляет 2 функции: dispatch и subscribe(subscribe to state change). Subscribe позволяет подписаться на обновления состояния стора. Схема работы dispatch:



Dispatch принимает в себя некоторый action, который в дальнейшем вместе с текущим стейтом передает в reducer - функцию, которая на основании action изменяет стейт и возвращает новое актуальное значение. Вместе с новым значением стейта также может быть возвращен набор асинхронных эффектов. Dispatch ставит выполнение эффектов в очередь задач event loop, чтобы не блокировать интерфейс пользователя. Также в каждый эффект передаются внешние зависимости(external deps), что позволяет использовать эффекты например для взаимодействия с API.

Далее рассмотрим более детальное устройство reducer:



Как уже было сказано раньше reducer принимает в качестве аргументов action и актуальное значение state. Но перед тем как обработать action нам нужно применить immer для того, чтобы получить мутабельный объект стейта. Такое решение связано с тем, что стор опирается на концепцию имутабельности данных, следовательно просто изменить данные в объекте нельзя, нужно полностью скопировать объект и вернуть ссылку на объект с уже измененными данными. Для оптимизации подобного подхода существует библиотека immer, позволяющая работать с имутабельными данными как с мутабельными, но в конце получить новый объект. После проведения изменений возвращает новое состояние и набор эффектов.

Пример реализации dispatch:

```ts
export type Store = Readonly<{
  getState: () => State
  dispatch: Dispatch
  subscribe: Subscribe
}>

export type State = {
  testMap: Map<
    string,
    {
      name: string
    }
  >
}
export type Dispatch = (action: Action) => void
export type InitState = () => State
export type ExternalDeps = {
  // Просто для примера
  api: {
    fetch: (arg: string) => string
  }
}
export type Effect = (externalDeps: ExternalDeps) => Promise<Action>

type Subscribe = (cb: Listener) => () => void
type Listener = (state: State) => void
```

```ts
export function createStore(
  initState: InitState,
  externalDeps: ExternalDeps,
): Store {
  let state = initState()
  let listeners: Listener[] = []

  const getState = () => state

  const dispatch: Dispatch = (action: Action) => {
    const { nextState, effects } = reducer(state, action)

    const hasStateChanged = state !== nextState
    state = nextState

    if (hasStateChanged) {
      listeners.forEach((listener) => listener(nextState))
    }

    if (effects.length !== 0) {
      setTimeout(() => {
        for (const effect of effects) {
          effect(externalDeps).then(dispatch)
        }
      })
    }
  }

  const subscribe: Subscribe = (cb) => {
    listeners.push(cb)

    return () => {
      listeners = listeners.filter((fn) => fn !== cb)
    }
  }

  return { getState, dispatch, subscribe }
}
```

Пример реализации reducer:

```ts
import { createDraft, finishDraft } from 'immer'

// eslint-disable-next-line @typescript-eslint/no-unused-vars
function exhaustivenessCheck(_: never): never {
  throw new Error('Exhaustivness failure! This should never happen.')
}

export type Action = AddNew | AddNew_ReqeustCompleted

type AddNew = {
  type: 'AddNew'
}

type AddNew_ReqeustCompleted = {
  type: 'AddNew_ReqeustCompleted'
  name: string
}
```

```ts
export function reducer(
  prevState: State,
  action: Readonly<Action>,
): {
  nextState: State
  effects: Effect[]
} {
  const draft = createDraft(prevState)
  let effects: Effect[] = []

  switch (action.type) {
    case 'AddNew': {
      effects = [
        // Имитируем поход в АПИ
        async function fetchNewName({ api }) {
          const name = await api.fetch(Math.random().toString())
          return {
            type: 'AddNew_ReqeustCompleted',
            name,
          }
        },
      ]
      break
    }

    case 'AddNew_ReqeustCompleted': {
      const { name } = action
      prevState.testMap.set(name, {
        name,
      })
      break
    }

    default:
      exhaustivenessCheck(action)
  }

  const nextState = finishDraft(draft)
  return {
    nextState,
    effects,
  }
}
```

Можно заметить, что все вышеописанное никак не касается React JS, следовательно весь вышеописанный код является фреймворк агностик кодом и может быть использован с любым фреймворком. Но так как речь идет про React JS, разберем как можно использовать стор с данным фреймворком.

Нам нужно обеспечить возможность отслеживать изменения части стейта, для этого нам помогут селекторы. Пример реализации селектора и контект провайдера для стора:

```ts
type StoreProviderProps = {
  store: Store
}

const StoreContext = createContext<Store | null>(null)

export const StoreProvider: FC<StoreProviderProps> = ({ store, children }) => {
  return <StoreContext.Provider value={store}>{children}</StoreContext.Provider>
}

const useStore = (): Store => {
  const store = useContext(StoreContext)
  if (!store) {
    throw new Error('StoreProvider не добавлен в дерево Реакта!')
  }

  return store
}

export const useDispatch = (): Dispatch => {
  return useStore().dispatch
}
```

Главным аспектом тут является хук useSyncExternalStoreWithSelector, который позволяет завязаться на изменения части стора и вызывать процесс рендера только по необходимости.

```ts
type EqualityFn<T> = (a: T, b: T) => boolean

// eslint-disable-next-line @typescript-eslint/no-explicit-any
const refEquality: EqualityFn<any> = (a, b) => a === b

// eslint-disable-next-line @typescript-eslint/no-explicit-any
export const useSelector = <Selector extends (state: State) => any>(
  selector: Selector,
  equalityFn: EqualityFn<ReturnType<Selector>> = refEquality,
): ReturnType<Selector> => {
  const store = useStore()

  const selectedState = useSyncExternalStoreWithSelector(
    store.subscribe,
    store.getState,
    undefined,
    selector,
    equalityFn,
  )

  return selectedState
}
```

Работа с сетью

Существует множество общепризнанных библиотек таких как например axios, но мной будет рассмотрена простейшая реализация АПИ, с которым наше гипотетическое приложение могло бы работать. Также такое АПИ может быть использовано в качестве external deps в примерах выше по реализации стора. Ниже описание примерной реализации с возможностью перезагрузки токена доступа.

Внутри класса API есть очередь запросов. Каждый запрос помимо данных и метода, которые нужно использовать хранит в себе информация о кол-ве ретраев, интервале между ретраями и множитель данного интервала. После вызова метода fetch мы отсылаем запрос в функцию request внутри бесконечного цикла. Если удалось получить ответ, возвращаем его, иначе проверяем тип ошибки. Если это ошибка истекшего токена, то сбрасываем кол-во патраев и перезапрашиваем токен и ждем задержку. Если ошибка известная и это не токен, то просто ждем задержку. В противном случае

выкидываем ошибку выше. Упрощенный код, реализующий данный алгоритм представлен ниже на скриншотах.

```ts
type Requests = {
  'messages.method': {
    params: {}
    response: {}
  }
}

type FetchOptions = {
  retries: number
  retryDelay: number
}

// Используем пустую функцию просто для примера
// eslint-disable-next-line @typescript-eslint/no-unused-vars
const externalDoRequest = async (...data: any): Promise<any> => {}

// Используем пустую функцию просто для примера
// eslint-disable-next-line @typescript-eslint/no-unused-vars
const getToken = (isExpired: boolean): Promise<string> => {
  return new Promise((resolve) => resolve(''))
}

class TokenError extends Error {
  constructor(message: string) {
    super(message)

    Object.setPrototypeOf(this, TokenError.prototype)
  }
}

class RequestError extends Error {
  constructor(message: string) {
    super(message)

    Object.setPrototypeOf(this, RequestError.prototype)
  }
}
```

```
const fetch = async <Method extends keyof Requests>(
  method: Method,
  params: Requests[Method]['params'],
  options: FetchOptions,
): Promise<Requests[Method]['response']> => {
  let fetchAttempts = 0
  let token = await getToken(false)

  while (true) {
    try {
      const res = await externalDoRequest(method, params, token)
      return res
    } catch (err: unknown) {
      fetchAttempts += 1
      if (fetchAttempts === options.retries + 1) {
        throw err
      }

      if (err instanceof TokenError) {
        fetchAttempts = 0
        token = await getToken(true)
        await sleep(options.retryDelay)
        continue
      }

      if (err instanceof RequestError) {
        await sleep(options.retryDelay)
        continue
      }
      throw err
    }
  }
}

```

Выводы:

В рамках данной работы было рассмотрено как можно создать самописный стор для приложения, который в дальнейшем можно улучшать и дополнять, а так же как сделать базовый класс для работы с АПИ. Такой подход отлично подойдет для небольших приложений, а также позволит лучше разобраться как работают все современные библиотеки. Для дальнейшей разработки можно добавить самую популярную библиотеку для роутинга react-router-dom, которая обладает отличной документацией.