# Введение в Effector

В этом месяце вы познакомились с несколькими способами управления состоянием: от базовых механизмов библиотеки React (Props drilling и React Context API) до отдельной библиотеки Redux, с помощью которой можно описать сложные системы хранения и модификации состояния. Помимо Redux существуют и другие решения, которые позволяют описывать логику состояния приложения. В этом дополнительном материале мы расскажем об одном из них — библиотеке [Effector](https://effector.dev/).

## Что такое Effector

Effector — реактивный стейт-менеджер, основанный на теории графов. Библиотека была создана как альтернатива Redux. Но есть и концептуальное отличие Effector от Redux — подход множественных сторов (multi-store). Если в Redux вы обычно создаёте одно большое глобальное состояние, то в Effector, наоборот, дробите стейт на множество атомарных сторов и храните эти сущности отдельно. Такой подход значительно упрощает взаимодействие с данными: если вы измените что-то в одном сторе, это никак не повлияет на другой стор.

Кроме этого, объекты, которые генерирует Effector, представляют собой вершины связных графов — все вычисления происходят в строгой последовательности, что полностью исключает такие неприятные моменты, как гонку состояний.

Но и это не всё. У библиотеки Effector — масса других преимуществ, к примеру:

* высокая производительность;
* небольшой размер: ~7KB (GZIP);
* простой, но очень ёмкий API: для начала работы потребуется изучить всего три сущности;
* отличные возможности типизации: есть поддержка TypeScript и Flow;
* независимость от фреймворков: Effector можно применять в любом JS-приложении.

Если коротко описать суть работы с Effector, в этой библиотеке всё сводится к двум вещам:

* созданию событий, которые вызываются извне или внутри бизнес-логики,
* описанию слушателей, которые ждут возникновения события и реагируют на это — запрашивают данные с сервера, меняют значение стора, запускают сайд-эффекты.

## Как изучать Effector

Чтобы начать работать с этой библиотекой, достаточно изучить три основные сущности, которые в Effector принято называть «юнитами» (англ. Unit): Event, Store и Effect. С их помощью можно описать довольно сложную логику работы с состоянием.

Но есть один важный нюанс: во время знакомства с Effector вам потребуется полностью абстрагироваться от библиотеки Redux, которую вы изучили ранее. В этих библиотеках применяются разные подходы к работе с данными, поэтому если рассматривать Effector через призму опыта работы с Redux, можно легко запутаться.

Если вы готовы, переходите к следующему уроку.

**Добавление Effector в проект**

Effector — независимая от фреймворков библиотека. Это значит, что вы можете использовать её в любом JavaScript-проекте и с различными фреймворками. Чтобы добавить Effector в проект, достаточно установить одну зависимость:

Скопировать кодBASH

npm install --save effector

Но чтобы использовать Effector вместе с React было удобнее, существует специальная библиотека — effector-react. Она содержит, к примеру, полезный хук useStore(), который позволяет получать актуальное значение состояния в функциональном компоненте. Установить эту библиотеку можно с помощью такой команды:

Скопировать кодBASH

npm install --save effector-react

Всё. Теперь ваш React-проект готов к эффективному менеджменту состояний. И никаких дополнительных настроек не требуется.

# Юниты

Большая часть работы с Effector заключается в создании различных юнитов и описании взаимодействий между ними. Юнит (англ. Unit) — тип данных, представляющий собой вершину графа. Всего в Effector существует четыре типа юнитов, но в этой теме мы рассмотрим лишь три из них — этого достаточно для полноценного описания бизнес-логики сложного приложения.

* Event — функция, которая вызывается в качестве реакции на какое-либо событие.
* Store — объект, в котором хранятся данные.
* Effect — объект для описания асинхронных сайд-эффектов.

Рассмотрим их поближе.

## Event

Event (или событие) отражает намерение изменить состояние и вызывается в качестве реакции на какое-либо событие. Например, пользователь кликнул по кнопке добавления товара в корзину, на что интерфейс должен соответствующе отреагировать. Для этого в обработчике клика можно вызвать Event, что послужит сигналом для бизнес-логики, которая отвечает за состояние корзины.

Чтобы создать Event, следует воспользоваться функцией createEvent():

Скопировать кодJSX

const addToCart = createEvent();

Event нужно использовать как вызов функции. В параметры можно передавать любое значение, например ID товара, который нужно положить в корзину:

Скопировать кодJSX

const addToCart = createEvent();

addToCart(35);

Но в данном случае ничего не произойдёт, так как на это событие ещё никто не подписан. Создание событий обретёт смысл тогда, когда что-то будет ожидать эти события и как-то на них реагировать. Например, подписки хранилища.

## Store

Store (или хранилище) — объект, в котором хранится состояние приложения. Это состояние иммутабельно, а для его обновления нужно передать в Store значение, которое не равно текущему и не является undefined.

Создать хранилище так же просто, как и событие, — нужно вызвать функцию createStore():

Скопировать кодJSX

const $cart = createStore([]);

В качестве параметра функция createStore() принимает начальное значение состояния. Это обязательно, ведь значение Store не может быть undefined.

Также обратите внимание на префикс $ в названии хранилища. Ставить этот префикс не обязательно, но в сообществе разработчиков принято обозначать хранилища состояний таким символом — это помогает визуально определить тип сущности.

## Effect

Effect (или эффект) — контейнер для выполнения асинхронных действий. С его помощью удобно делать запросы на сервер, отслеживать состояние и успешность выполнения запросов.

Чтобы создать эффект, достаточно воспользоваться функцией createEffect():

Скопировать кодJSX

const getProductsFx = createEffect(

async () => {

const response = await fetch(`https://store.com/products-list`);

return response.json();

}

);

В качестве параметра createEffect() принимает любую асинхронную функцию (handler).

Эффекты принято обозначать постфиксом Fx, но, как и в случае с хранилищем, делать это необязательно.

Объект Effect имеет несколько полезных полей-событий, которые помогают следить за жизненным циклом эффекта и реагировать на возникающие ситуации. Каждое из этих полей возвращает Event:

* done — вызывается после успешного выполнения асинхронной функции эффекта;
* doneData — вызывается после успешного выполнения асинхронной функции эффекта и возвращает результат вызова функции;
* fail — вызывается после возникновения ошибки во время выполнения асинхронной функции эффекта;
* failData — вызывается после возникновения ошибки во время выполнения асинхронной функции эффекта и возвращает тело ошибки;
* finally — вызывается в любом случае и возвращает статус, результат и параметры выполнения асинхронной функции эффекта;
* pending — возвращает boolean Store, который равен true, пока асинхронная функция эффекта не завершит выполнение;
* inFlight — похож на pending, но возвращает number Store, который зависит от количества незавершённых вызовов асинхронной функции эффекта.

Вы познакомились с основными юнитами библиотеки Effector. Теперь самое время применить эти знания на практике. Увидимся в следующем уроке.

# Инициализация хранилища

Вы познакомились с юнитами Event, Store и Effect, пора научиться описывать взаимодействия между ними. В этом уроке разберём несколько способов модификации стора и научимся выводить актуальное значение хранилища в React-компоненте.

## Создание счётчика

Напишем с помощью Effector обычный счётчик. Для начала создадим стор, в котором будет храниться актуальное значение счётчика:

Скопировать кодJSX

// хранилище счётчика с начальным значением 0

const $count = createStore(0);

Для модификации состояния счётчика воспользуемся событиями:

Скопировать кодJSX

const increment = createEvent();

const decrement = createEvent();

Но события отражают лишь намерение изменить состояние, служат чем-то вроде «сигналов», которые можно вызвать из интерфейса. Сами по себе они ничего не модифицируют. Чтобы изменить значение стора с помощью событий, нужно научить хранилище на них реагировать. Это можно сделать с помощью механизма подписок.

### Создание подписки. Метод Store.on()

Чтобы модифицировать значение хранилища, нужно создать подписку на событие. Таким образом, Store будет слушать определённый Event и соответствующе реагировать, когда Event будет вызван.

Для создания подписки в юните Store есть специальный метод on(), который принимает два аргумента: trigger и reducer.

Триггером может быть Event, Effect или другой Store.

Редьюсер — функция, которая принимает параметры state и params, где state — текущее состояние хранилища, а params — параметры, переданные через вызов юнита. Внутри функции reducer производятся вычисления, необходимые для обновления состояния. Редьюсер обязательно должен вернуть новое состояние хранилища. Внутри редьюсера ни в коем случае нельзя мутировать стейт (изменять состояние напрямую) — это приведёт к ошибкам.

На практике создание подписки для счётчика выглядит так:

Скопировать кодJSX

const $count = createStore(0);

const increment = createEvent();

const decrement = createEvent();

$count

// подписка на Event increment

.on(increment, (currentCount, incrementValue) => (currentCount += incrementValue))

// подписка на Event decrement

.on(decrement, (currentCount, decrementValue) => (currentCount -= decrementValue));

Этого кода достаточно, чтобы создать простейший счётчик с помощью Effector. Но пока мы лишь описали модель счётчика, теперь научимся выводить информацию на страницу и менять её на лету.

## Интеграция Effector в React

Сперва определим, как будет выглядеть компонент счётчика. Если максимально всё упростить, то получится примерно такой код:

Скопировать кодJSX

function Counter() {

return (

<section>

<h1>Count: 0</h1>

<div>

<button onClick={() => { /\* увеличить значение \*/ }}> + </button>

<button onClick={() => { /\* уменьшить значение \*/ }}> - </button>

</div>

</section>

);

}

Компонент в заголовке показывает текущее состояние и имеет две кнопки для увеличения или уменьшения значения счётчика.

Теперь добавим состояние. Для простоты это можно сделать в том же файле, где находится компонент счётчика. Но дальше в этой теме мы рассмотрим, в каких файлах принято создавать модели.

Скопировать кодJSX

import { createStore, createEvent } from "effector";

const $count = createStore(0);

const increment = createEvent();

const decrement = createEvent();

$count

.on(increment, (currentCount) => (currentCount += 1))

.on(decrement, (currentCount) => (currentCount -= 1));

function Counter() {

return (

<section>

<h1>Count: 0</h1>

<div>

<button onClick={() => { /\* увеличить значение \*/ }}> + </button>

<button onClick={() => { /\* уменьшить значение \*/ }}> - </button>

</div>

</section>

);

}

Обратите внимание, что редьюсеры в подписках никак не обрабатывают params. Это необязательный параметр для редьюсера и сейчас его можно исключить.

На этом этапе уже можно оживить интерфейс. Для получения значения хранилища воспользуемся хуком useStore() из библиотеки effector-react. Чтобы магия заработала, достаточно передать в хук нужное хранилище, на которое будет создана подписка. useStore() вернёт текущее значение стора, а при изменении хранилища компонент будет обновлён автоматически.

С событиями всё гораздо проще — их можно использовать в компонентах напрямую:

Скопировать кодJSX

import { createStore, createEvent } from "effector";

import { useStore } from "effector-react";

const $count = createStore(0);

const increment = createEvent();

const decrement = createEvent();

$count

.on(increment, (currentCount) => (currentCount += 1))

.on(decrement, (currentCount) => (currentCount -= 1));

function Counter() {

*// значение стора $count получено с помощью хука:*

const currentCount = useStore($count);

return (

<section>

{/\* использовать стор в JSX можно как обычную переменную \*/}

<h1>Count: {currentCount}</h1>

<div>

{/\* использовать события можно напрямую \*/}

<button onClick={() => { increment(); }}> + </button>

<button onClick={() => { decrement(); }}> - </button>

</div>

</section>

);

}

Теперь при нажатии на кнопки будут вызваны соответствующие события и значение хранилища изменится.

## Использование параметров в событиях

Предположим, что условия изменились и теперь нам хотелось бы менять счётчик на определённое значение, которое пользователь задаёт сам. К примеру, в компоненте будет не две кнопки, а четыре: две — для изменения счётчика на единицу и ещё две — для изменения счётчика на пять. Реализовать это проще простого, достаточно лишь добавить параметр редьюсера в подписках хранилища, а в интерфейсе передавать нужную величину в виде аргумента события:

Скопировать кодJSX

import { createStore, createEvent } from "effector";

import { useStore } from "effector-react";

const $count = createStore(0);

const increment = createEvent();

const decrement = createEvent();

$count

*// добавим параметры в редьюсеры*

.on(increment, (currentCount, incrementValue) => (currentCount += incrementValue))

.on(decrement, (currentCount, decrementValue) => (currentCount -= decrementValue));

function Counter() {

const currentCount = useStore($count);

return (

<section>

<h1>Count: {currentCount}</h1>

<div>

{/\* в события можно передавать любые аргументы \*/}

<button onClick={() => { increment(1); }}> + 1 </button>

<button onClick={() => { decrement(1); }}> - 1 </button>

</div>

<div>

<button onClick={() => { increment(5); }}> + 5 </button>

<button onClick={() => { decrement(5); }}> - 5 </button>

</div>

</section>

);

}

Отлично, теперь счётчик можно менять на любое значение.

## Очистка состояния. Метод Store.reset()

Бывают случаи, когда состояние хранилища нужно сбросить, например при клике на кнопку «Очистить корзину» корзина должна опустеть. Для решения подобных задач Store имеет специальный метод reset(). Этот метод принимает один аргумент trigger либо массив триггеров: [triggerA, triggerB, ...]. В качестве триггеров можно использовать Event, Effect или даже другой Store.

Добавим кнопку сброса счётчика в компонент:

Скопировать кодJSX

import { createStore, createEvent } from "effector";

import { useStore } from "effector-react";

const $count = createStore(0);

const increment = createEvent();

const decrement = createEvent();

*// создадим Event,*

*// с помощью которого сообщим хранилищу о намерении сбросить значение*

const clear = createEvent();

$count

.on(increment, (currentCount, incrementValue) => (currentCount += incrementValue))

.on(decrement, (currentCount, decrementValue) => (currentCount -= decrementValue))

*// создадим подписку на clear, которая будет сбрасывать значение хранилища*

.reset(clear);

function Counter() {

const currentCount = useStore($count);

return (

<section>

<h1>Count: {currentCount}</h1>

<div>

<button onClick={() => { increment(1); }}> + 1 </button>

<button onClick={() => { decrement(1); }}> - 1 </button>

</div>

<div>

<button onClick={() => { increment(5); }}> + 5 </button>

<button onClick={() => { decrement(5); }}> - 5 </button>

</div>

<div>

{/\* вызов события clear ничем не отличается от вызова других событий \*/}

<button onClick={() => { clear(); }}> clear </button>

</div>

</section>

);

}

Всё готово. Теперь, помимо модификации счётчика, его можно легко сбросить. Осталось только найти этому практическое применение.

## Заключение

В этом уроке вы познакомились с основными способами взаимодействия React-компонентов с моделями состояний, описанными с помощью Effector. Вы удивитесь, но в большинстве случаев это всё, что нужно для получения значения хранилища и его модификации.

В следующем уроке глубже погрузимся в юнит Effect — научимся загружать данные и сохранять их в хранилище.

**Работа с данными**

В этом уроке погрузимся в тонкости работы с юнитом Effect: научимся создавать и использовать его.

**Создание эффекта**

Effect — объект-контейнер для асинхронных функций. С помощью эффекта удобно выполнять любые асинхронные вычисления, а самый частый способ его применения — отправка запросов на сервер и обработка ответов. Рассмотрим эффект, который делает запрос к публичному API и получает случайную цитату:

Скопировать кодJSX

const loadQuote = createEffect(async () => {

const res = await fetch("https://api.kanye.rest");

return res.json();

});

Функция createEffect() в качестве обязательного аргумента принимает handler, в котором обычно описывается асинхронная логика.

При вызове эффекта loadQuote() произойдёт запрос к API. Но в таком виде вызов эффекта мало чем отличается от вызова обычной асинхронной функции. Возможности эффекта раскрываются именно при взаимодействии с другими юнитами.

**Как использовать эффекты**

**Добавление в хранилище информации из эффекта.** Просто получить данные с сервера недостаточно, важно где-то их сохранить. Для этого логично воспользоваться Store. Чтобы добавить в хранилище информацию, полученную в эффекте, можно применить параметр Effect.doneData, который послужит сигналом для подписчика хранилища:

Скопировать кодJSX

const loadQuote = createEffect(async () => {

const res = await fetch("https://api.kanye.rest");

return res.json();

});

*// создадим стор*

const $quote = createStore(null);

*// добавим в стор подписку, которая будет слушать событие doneData*

$quote.on(loadQuote.doneData, (\_, data) => data.quote );

После этого данные, полученные в результате запроса, автоматически добавятся в хранилище. Обратите внимание, что в результате каждого вызова редьюсер возвращает новую строку из поля data.quote, — благодаря этому значение в хранилище будет перезаписано. А первый параметр редьюсера заменён на символ подчёркивания \_ из-за того, что при вычислении нового значения хранилища никак не используется предыдущее.

**Отслеживание статуса запроса.** Чтобы отображать информацию в интерфейсе, зачастую требуется отслеживать статус запроса. Для этого достаточно создать ещё один Store, в котором будет храниться информация о состоянии выполнения асинхронной функции, и подписаться на соответствующий параметр эффекта pending:

Скопировать кодJSX

const loadQuote = createEffect(async () => {

const res = await fetch("https://api.kanye.rest");

return res.json();

});

*//...*

*// стор с информацией о статусе запроса*

const $isLoading = createStore(false);

*// параметр pending возвращает true,*

*// если асинхронная функция находится в процессе выполнения*

$isLoading.on(loadQuote.pending, (\_, isPending) => isPending);

При работе с библиотекой Effector гораздо удобнее создавать отдельные хранилища для любых новых данных, чем хранить большой объект с множеством полей. Это позволяет упростить редьюсеры и сократить количество пересчётов стора, что в итоге поможет снизить количество ререндеров компонента.

**Добавление логики ошибки.** Иногда во время выполнения запроса к серверу может возникнуть ошибка. Если это случилось, нужно показать пользователю сообщение с соответствующей информацией. Чтобы добавить такую логику, также можно воспользоваться стором, если сообщения реализованы через эффектор. Но в случае, если сообщения нужно показать через другое API, лучше воспользоваться вспомогательным методом watch(), который есть у каждого юнита. А узнать, что в запросе что-то пошло не так, можно с помощью параметра fail:

Скопировать кодJSX

const loadQuote = createEffect(async () => {

const res = await fetch("https://api.kanye.rest");

return res.json();

});

loadQuote.fail.watch(() => {

alert('При получении данных произошла ошибка.');

});

Теперь, если при выполнении запроса произойдёт ошибка, пользователь об этом узнает.

Метод watch() создаёт так называемый «вотчер», в котором можно описать любую логику сайд-эффектов. Этот метод позволяет как логировать нужную информацию во время разработки, так и настраивать взаимодействие между Effector и другими библиотеками. watch() и другие полезные вспомогательные методы подробнее разберём в следующих уроках.

**Вспомогательные методы. Часть 1**

В предыдущих уроках вы познакомились с основными типами юнитов и разобрали, как их применять на практике. Теперь вы знаете достаточно, чтобы приступить к изучению методов, которые сильно расширяют возможности юнитов.

**Добавление сайд-эффектов. Watch**

Метод watch() полезен как во время разработки — с его помощью можно быстро логировать информацию из любого юнита, так и для написания различных сайд-эффектов и связи Effector с другими библиотеками.

watch() можно вызвать у любого из рассмотренных в этой теме юнитов — Event, Effect и Store. В качестве обязательного аргумента watch() принимает функцию-watcher, параметры которой определяются типом юнита.

Например, если подписаться на событие и при его вызове передать аргумент, это значение можно перехватить внутри вотчера и использовать для описания сайд-эффектов:

Скопировать кодJSX

const event = createEvent();

event.watch((value) => {

console.log('Яндекс', value);

});

event('Практикум');

// => Яндекс Практикум

Если же подписаться на хранилище, вотчер будет получать новое значение хранилища при каждом изменении:

Скопировать кодJSX

const update = createEvent();

const $store = createStore(0);

$store.on(update, (state, value) => state + value);

$store.watch((state) => {

console.log("Current state: ", state);

});

update(1);

update(2);

update(3);

// Current state: 0

// Current state: 1

// Current state: 3

// Current state: 6

Обратите внимание, что в таком случае вотчер отрабатывает даже во время инициализации. Чтобы подписаться только на обновления хранилища, можно воспользоваться встроенным параметром стора updates:

Скопировать кодJSX

const update = createEvent();

const $store = createStore(0);

$store.on(update, (state, value) => state + value);

// применение параметра updates

$store.updates.watch((state) => {

console.log("Current state: ", state);

});

update(1);

// Current state: 1

updates возвращает Event, который триггерится только при обновлении значения хранилища.

**Объединение массива юнитов. Merge**

Метод merge() умеет объединять массив юнитов в одно событие. На вход merge() принимает массив юнитов, а возвращает одно событие:

Скопировать кодJSX

merge(units: Unit[]): Event

merge() может быть полезен, когда нужно одинаково отреагировать на разные события:

Скопировать кодJSX

const event1 = createEvent();

const event2 = createEvent();

const mergedEvent = merge([foo, bar]);

mergedEvent.watch((value) => console.log('Merged event triggered: ', value));

event1(1) *// => Merged event triggered: 1*

event2(2) *// => Merged event triggered: 2*

**Компиляция хранилищ. Combine**

combine() служит для компиляции нескольких хранилищ в одно. Принимает массив юнитов типа Store и функцию, в которой можно объединить все данные в один объект.

Например, вы можете работать с данными пользователя по отдельности, но перед отправкой данных на сервер собрать все нужные поля в одном объекте:

Скопировать кодJSX

const $firstName = createStore("Keanu");

const $lastName = createStore("Reeves");

const $fullName = combine(

[$firstName, $lastName],

([firstName, lastName]) => `${firstName} ${lastName}`

);

const $user = combine(

[$fullName, $firstName, $lastName],

([fullName, firstName, lastName]) => ({

fullName,

firstName,

lastName

})

);

$user.watch((state) => console.log(state));

*// => { fullName: "Keanu Reeves", firstName: "Keanu", lastName: "Reeves" }*

**Передача событий между юнитами. Forward**

forward() — очень полезный метод, который помогает настраивать связи между различными юнитами. В качестве обязательного аргумента forward() принимает объект с полями from и to, в которые записывают юнит либо массив юнитов.

forward() передаёт событие из одного юнита (from) в другой (to). Так как вызов эффектов напрямую из компонентов считается плохой практикой из-за смешения представления и бизнес-логики, forward() удобно использовать для вызова эффектов с помощью событий. Скажем, вы можете подгрузить данные после того, как страница была смонтирована:

Скопировать кодJSX

const pageMounted = createEvent();

const loadDataFx = createEffect(/\* ... \*/);

forward({

from: pageMounted,

to: loadDataFx

});

Таким же образом можно добавить на страницу кнопку перезагрузки данных и запускать этот процесс с помощью дополнительного события:

Скопировать кодJSX

const pageMounted = createEvent();

const reloadData = createEvent();

const loadDataFx = createEffect(/\* ... \*/);

forward({

from: [pageMounted, reloadData],

to: loadDataFx

});

forward() можно применять не только для событий и эффектов, но и для хранилищ. В этом случае возможны разные вариации связей между юнитами: Event → Effect, Event → Store, Store → Event, Effect → Effect и так далее.

В этом уроке мы рассмотрели четыре вспомогательных метода. С их помощью вы сможете добавлять сайд-эффекты любому юниту, объединять несколько событий в одно, комбинировать хранилища и передавать события от одного юнита другому.

Но это не всё. В следующем уроке разберём ещё три чуть более сложных и не менее полезных метода.

# Вспомогательные методы. Часть 2.

В этом уроке продолжим разбирать вспомогательные методы в Effector.

## Передача событий с условием. Guard

Этот метод похож на forward(), который вы изучили в предыдущем уроке. Разница в том, что при использовании метода guard() можно добавлять условие передачи события. В качестве аргумента Guard принимает объект:

Скопировать кодJSX

guard({ clock?, target?, source, filter }): target

* clock — триггер, событие, которое запускает логику guard(). Необязательное поле;
* target — юнит, который будет вызван, если условие выполнится. Необязательное поле. Если target не передан, guard() возвращает target как результат выполнения;
* source — юнит, из которого будут взяты данные для передачи в target. В случае, если в guard() не передан clock, то source послужит триггером для его выполнения;
* filter — функция, в которой выполняется проверка условия выполнения guard(), возвращает boolean значение.

Guard создан для внедрения условий в бизнес-логику приложения. Например, с его помощью можно ограничивать изменения в сторе:

Скопировать кодJSX

const update = createEvent();

const clear = createEvent();

const $store = createStore(0);

$store

.on(update, (state) => (state += 1))

.reset(clear);

guard({

source: $store,

target: clear,

filter: (value) => value > 2

});

$store.watch((state) => { console.log("Current state: ", state); });

update();

update();

update();

В этом примере guard активируется при каждом изменении $store и проверяет значение стора value. Если значение больше двух, происходит вызов события clear и $store сбрасывается в исходное состояние.

## Передача данных из хранилища в юниты. Sample

sample() — полезный метод, с помощью которого можно взять определённое значение из хранилища и передать его в любой другой юнит.

По своему интерфейсу sample() похож на guard():

Скопировать кодJSX

sample({ clock?, target?, source?, fn? }): target

Метод sample() принимает объект с необязательными полями:

* clock — триггер, событие, которое запускает логику sample();
* target — юнит, который будет вызван в результате работы sample();
* source — юнит, из которого будут взяты данные для передачи в target. В случае, если в sample() не передан clock, то source послужит триггером для его выполнения;
* fn — функция, в которой можно модифицировать данные перед передачей в target.

Обратите внимание, что target можно как вызвать внутри метода, так и вернуть из sample().

Описать последовательность работы sample() можно так:

* сперва происходит вызов clock;
* затем sample() читает данные в source и обработает их, если нужно, в fn;
* в результате sample() передаёт данные в target.

Самая частая ситуация использования sample() — когда вам нужно вызвать Event с текущим значением какого-либо Store. sample() оптимизирован для таких ситуаций и при использовании этого метода вы можете быть уверены, что данные, полученные из стора, будут актуальны:

Скопировать кодJSX

const update = createEvent();

const targetEvent = createEvent();

const $store = createStore(0);

$store.on(update, (state) => (state += 1));

sample({

source: $store,

clock: update,

target: targetEvent,

fn: (storeValue) => `Event triggered with: ${storeValue}`

});

targetEvent.watch((data) => console.log(data));

update(); // => Event triggered with: 1

update(); // => Event triggered with: 2

update(); // => Event triggered with: 3

Обратите внимание на событие update — оно одновременно запускает обновление хранилища и триггерит sample(), который умеет отслеживать такие ситуации. sample() дожидается всех обновлений в $store и только после этого передаёт управление следующим узлам.

## Работа с эффектами. Attach

Метод attach(), в свою очередь, похож на sample(), но создан специально для работы с эффектами. С помощью этого метода вы можете выполнять обработку аргументов эффекта и вкладывать в него данные из хранилища.

Так выглядит метод attach() в коде:

Скопировать кодJSX

attach({ effect, source?, mapParams? }): newEffect

Он принимает объект с несколькими аргументами и возвращает новый эффект:

* newEffect — эффект, который возвращает attach(). Является триггером, после которого attach() начинает свою работу;
* effect — эффект, который будет вызван в результате работы attach();
* source — Store, из которого будут взяты данные для передачи в effect;
* mapParams — функция, в которой значение, полученное из source, можно обработать перед передачей в effect.

Чаще всего метод attach() применяют для создания общих эффектов, которые можно вызывать через другие эффекты в зависимости от ситуации. Хорошим примером может послужить обёртка над fetch API:

Скопировать кодJSX

const $url = createStore("https://pokeapi.co/api/v2/");

const xhrFx = createEffect(async (url) => {

const res = await fetch(url);

return res.json();

});

xhrFx.doneData.watch((data) => console.log(data));

const fetchPokemonById = attach({

effect: xhrFx,

source: $url,

mapParams: (params, source) => `${source}pokemon/${params}`

});

fetchPokemonById(1);

fetchPokemonById(2);

fetchPokemonById(3);

В этом уроке вы познакомились с тремя методами, которые помогают передавать события по условию, получать актуальные данные из хранилища и передавать их другим юнитам, а также взаимодействовать с эффектами. Это не полный список методов, но этого набора достаточно для описания довольно сложной бизнес-логики.

В следующем уроке разберём архитектурные подходы к разработке приложения с использованием Effector.

# Структура приложения с Effector

Прелесть этой библиотеки в том, что в ней отсутствуют жёсткие правила написания кода. Поэтому вы можете использовать библиотеку так, как вам удобно и познавать архитектурные приёмы на личном опыте. Но для удобства работы с Effector в сообществе разработчиков сложилось несколько общих подходов к написанию бизнес-логики приложения — их мы и разберём в этом уроке.

## Модель

Бизнес-логику приложения в Effector принято разбивать на модели. Модель — структурная единица бизнес-логики, которая представляет собой отдельную сущность либо набор сущностей, объединённых одной областью ответственности. Например, если вы создаёте чат с возможностью регистрации, логику этого приложения можно разделить на такие модели:

* auth — модель с состоянием аутентификации пользователя,
* user — модель с данными о пользователе,
* messages — модель со списком сообщений.

## Структура приложения

Глобальные модели — элементы бизнес-логики, которые необходимы всему приложению или большей части его компонентов. Их принято держать рядом с корнем приложения — в директории /src или в /app:

Скопировать кодJSX

└── app/

├── components/

│ ├── profile/

│ └── messages/

│

└── models/

├── auth/

├── user/

└── messages/

Модели, которые нужны только одному компоненту либо ещё его дочерним компонентам, для удобства можно расположить рядом с этим компонентом:

Скопировать кодJSX

└── app/

├── components/

│ ├── profile/

│ │ └── changeAvatarModal/

│ │ └── model/ // т. к. модель относится только к одному компоненту

│ │ // директорию можно назвать просто "model"

│ └── messages/

│

└── models/

├── auth/

├── user/

└── messages/

## Структура модели

Каждая модель состоит минимум из двух файлов: файла объявления (declaration) и файла инициализации (initialization).

### Файл объявления

Файл объявления принято называть index.js. В этом файле происходит объявление всех юнитов модели. Делать экспорт юнитов принято только в index.js, это помогает создать единую точку взаимодействия с моделью для компонентов.

Типичная структура index.js файла выглядит так:

Скопировать кодJSX

*// app/models/user/index.js*

import { createStore, createEvent, createEffect } from 'effector';

export const $user = createStore(null);

export const loadUser = createEvent();

export const loadUserFx = createEffect(*/\* ... \*/*);

Если в index.js объявлено слишком много юнитов, их можно вынести в отдельные файлы: хранилища в store.js, события в events.js, эффекты в fx.js. В этом случае важно пробросить все экспорты через index.js, чтобы сохранить единую точку взаимодействия.

### Файл инициализации

Файл инициализации принято называть init.js. В этом файле происходит настройка всех взаимодействий между юнитами модели. В файл инициализации можно импортировать все юниты модели, но из него нельзя ничего экспортировать. Только если очень-очень хочется, можно сделать экспорт юнита, но это может стать помехой в будущем.

init.js располагается рядом с index.js и выглядит примерно так:

Скопировать кодJSX

*// app/models/user/init.js*

import { forward } from 'effector';

import { $user, loadUser, loadUserFx } from './index.js';

$user.on(loadUserFx.doneData, (\_, data) => data);

forward({

from: loadUser,

to: loadUserFx

});

loadUserFx.fail.watch(() => console.log(':('));

## Глобальный файл инициализации

Логика работы библиотеки Effector построена так, что все модели можно инициализировать до старта работы с ними. Благодаря процессу инициализации Effector создаёт полный граф зависимостей между юнитами, что позволяет приложению работать корректно. Также инициализация помогает избежать циклических зависимостей.

Чтобы выполнить инициализацию, достаточно правильно организовать импорты init.js файлов:

* в директории models нужно создать файл init.js — это и будет глобальный файл инициализации,
* в глобальном init.js следует импортировать init.js файлы каждой созданной модели:  
  ```jsx // app/models/init.js

import './auth/init.js'; import './user/init.js'; import './messages/init.js'; ```

* глобальный init.js файл нужно импортировать в компоненте App:  
  ```jsx // app/app.js

import './models/init.js';

export function App() { //... ```

Так при старте приложения Effector узнает о всех юнитах, которые применяются в моделях, и построит зависимости между ними.

Для локальных моделей достаточно выполнить импорт модели в компоненте, который использует эту модель:

Скопировать кодJSX

*// app/components/profile/changeAvatarModal/index.js*

import './model/init.js';

export function ChangeAvatarModal() { *//...*

## Заключение

Мы разобрали общие подходы к написанию бизнес-логики приложения с библиотекой Effector. Эти правила помогут вам поддерживать хорошую структуру приложения и послужат основой для экспериментов с архитектурой.

А в следующем заключительном уроке вы сможете изучить уже знакомый вам проект корзины товаров, логика которого переписана на Effector. Код писать не придётся, достаточно просто внимательно с ним ознакомиться.

# Заключение и бонус

В этой теме вы познакомились с основными возможностями библиотеки Effector. Узнали, что такое юниты, и рассмотрели возможности взаимодействия между ними как напрямую, так и с помощью вспомогательных методов.

Мы разобрали далеко не все сущности библиотеки, например не изучили юнит Domain и некоторые вспомогательные методы. Также API представленных в теме элементов гораздо шире, чем те способы применения, которые мы изучили.

Тем не менее полученных знаний должно быть достаточно для решения большей части практических задач, с которыми вы можете столкнуться в повседневной работе. Если вам интересно изучить сущности библиотеки Effector подробнее, ознакомьтесь с [её документацией](https://effector.dev/docs/api/effector/effector/).

Также советуем заглянуть в уютный [чатик русскоязычного сообщества Effector](https://t.me/effector_ru) — там вы можете найти много полезной информации и задать любые интересующие вас вопросы.

В качестве бонуса предлагаем вам изучить уже знакомую корзину товаров, переписанную с применением библиотеки Effector.

Вы можете открыть исходный код как в нашем тренажёре, так и в любой другой среде разработки, для этого мы подготовили архив с проектом:

<https://code.s3.yandex.net/web-developer/archives/effector-task.zip>