# Введение в WebSocket

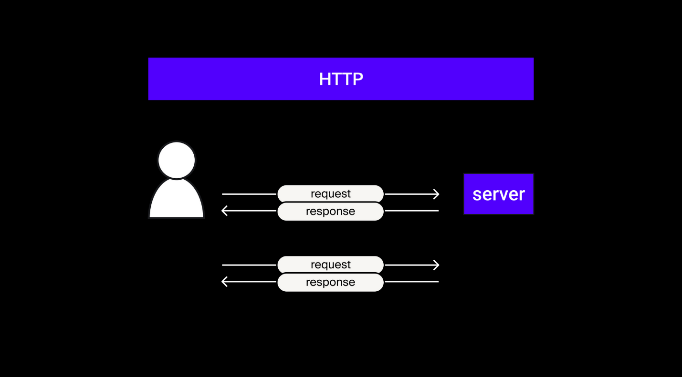
В этой теме вам предстоит познакомиться с новым протоколом передачи данных — WebSocket. Вы наверняка с ним уже сталкивались на просторах интернета, просто не задумывались об этом. Протокол WebSocket обычно применяют для реализации чатов, финансовых приложений (рынков ценных бумаг, облигаций), игровых платформ или других инструментов для совместной работы.

Протокол WebSocket создаёт постоянное соединение, с помощью которого можно непрерывно обмениваться данными без необходимости создавать новые запросы.

## Обмен данными по HTTP и WebSocket

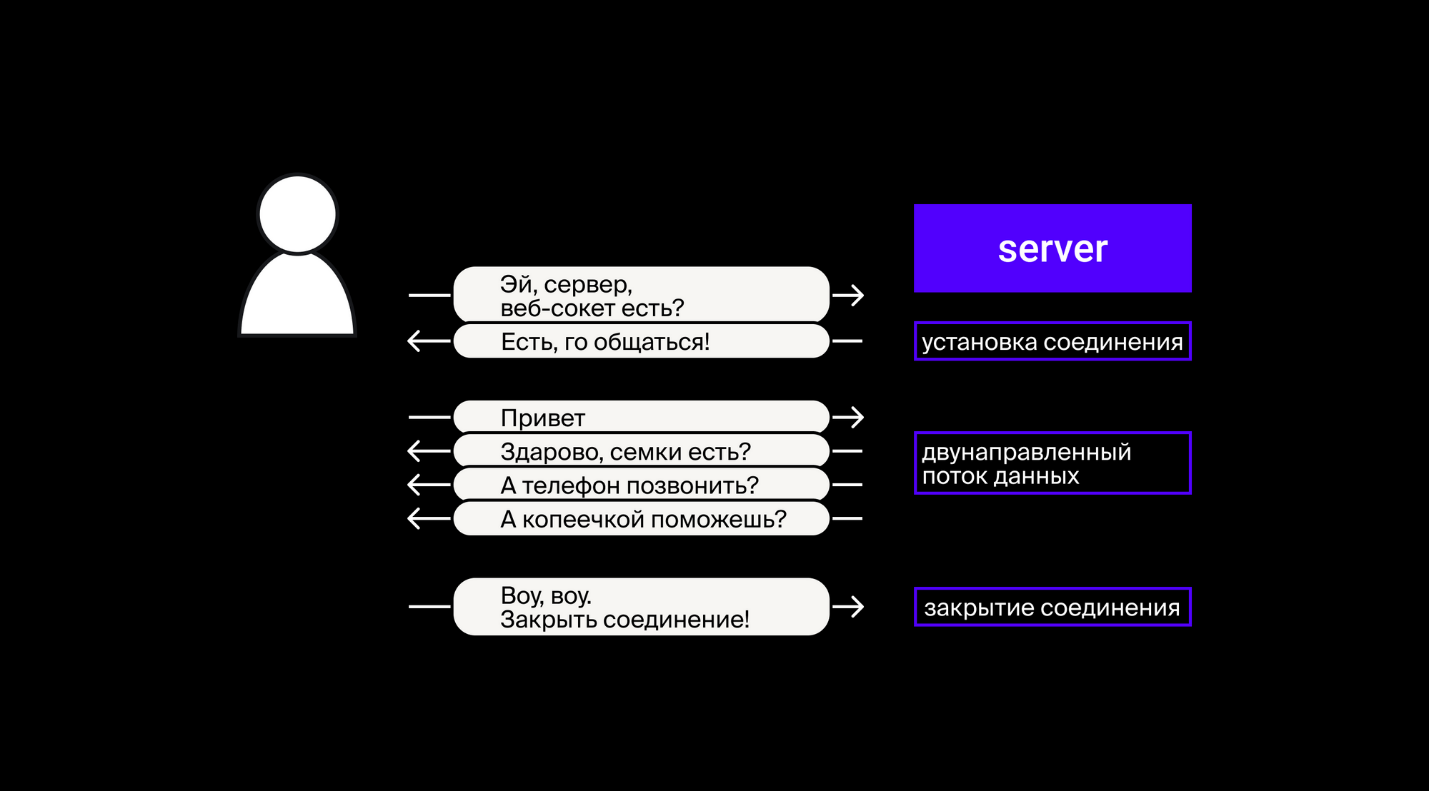
Сравним, как происходит обмен данными по протоколу WebSocket с уже хорошо знакомым нам HTTP.

Схема обмена данными по HTTP выглядит так — клиент отправляет запрос на сервер и получает от него ответ:



С веб-сокетом дела обстоят иначе:

1. Сначала необходимо установить соединение. Клиент отправляет HTTP-запрос, чтобы узнать, поддерживает ли сервер WebSocket.
2. Если сервер поддерживает сокет, то устанавливается постоянное соединение, в котором происходит обмен данными. При этом сервер может присылать клиенту сообщения без запроса (например, пуш-уведомления от приложений).
3. В любой момент времени любая из сторон может закрыть соединение.



При подключении по WebSocket используется один из протоколов: открытый (незащищённый) и зашифрованный.

## Открытый (ws://) и зашифрованный (wss://) протоколы

Протоколы в WebSocket соответствуют открытому http:// и зашифрованному https:// и записываются так: ws:// и wss://.

Вид протокола, по которому будет открываться WebSocket, в первую очередь зависит от сервера. Это значит, что если на стороне клиента мы выбираем зашифрованный протокол, а сервер его не поддерживает, сокет не будет работать. Поэтому на стороне клиента нужно всегда выбирать тот вид протокола, который поддерживается сервером.

Скопировать кодJSX

// открытый протокол

const ws = new WebSocket("ws://norma.nomoreparties.space/");

// зашифрованный протокол

const ws = new WebSocket("wss://norma.nomoreparties.space/");

Если в браузере открыть DevTools и посмотреть на запрос во вкладке Network, то среди заголовков запроса вы найдете те, которые сигнализируют серверу, что клиент хочет использовать веб-сокет как протокол передачи данных:

* Connection: Upgrade — указывает, что клиент хочет изменить протокол;
* Upgrade: websocket — указывается запрашиваемый протокол;
* Sec-WebSocket-Key: блаблабла== — случайный ключ для обеспечения безопастности;
* Sec-WebSocket-Version: 13 — версия протокола WebSocket.

Если сервер поддерживает WebSocket, клиент получит ответ со статусом 101 Switching Protocols и такими заголовками:

* Sec-WebSocket-Accept: newблаблабла== — перекодированный Sec-WebSocket-Key. Этот заголовок нужен, чтобы клиент (браузер) мог убедиться, что ответ соответствует запросу.
* Connection: Upgrade;
* Upgrade: websocket.

## Альтернативы WebSocket

WebSocket — не единственный в своём роде. Его альтернативой считают протокол server-sent events (SSE). Для работы с ним используется класс EventSource.

В целом его поведение очень похоже на поведение WebSocket за несколькими исключениями:

* это однонаправленный канал — данные посылает только сервер;
* отправляться может только текст;
* протокол — HTTP.

Протокол server-sent events на практике применяют гораздо реже, чем WebSocket. Но о нём полезно знать для расширения кругозора. С более подробной информацией о SSE вы можете ознакомиться [по ссылке](https://developer.mozilla.org/en-US/docs/Web/API/EventSource), а вот о WebSocket — уже в следующем уроке!

# Открытие и закрытие соединения

В этом уроке вы узнаете, как открывать и закрывать соединение. Кроме этого, научитесь определять тип состояния подключения.

## Открытие соединения

Для открытия соединения требуется создать объект WebSocket. Это можно сделать так:

Скопировать кодJSX

const ws = new WebSocket("ws://norma.nomoreparties.space/")

При создании объекта класса WebSocket не требуется никаких дополнительных параметров, кроме адреса API WebSocket с типом подключения ws или wss. О них мы говорили [в предыдущем уроке](https://praktikum.yandex.ru/trainer/react/lesson/d35a0419-85ee-4011-961e-dede83f55715).

## Состояние соединения

Объект ws содержит свойство readyState:

Скопировать кодJSX

const ws = new WebSocket("ws://norma.nomoreparties.space/")

console.log(ws.readyState)

А ещё у свойства readyState объекта ws есть четыре значения:

* 0 (CONNECTING): соединение ещё не установлено. WebSocket переходит в это состояние после создания объекта класса, но до начала обмена данными.
* 1 (OPEN): обмен данными. Свойство readyState принимает это состояние после того, как соединение установлено.
* 2 (CLOSING): соединение закрывается. Свойство readyState переходит в это состояние после начала закрытия соединения.
* 3 (CLOSED): соединение закрыто. Значение readyState принимает последнее состояние после закрытия соединения.

## Закрытие соединения

Право на закрытие соединения имеют как браузер, так и сервер. Это происходит с помощью вызова метода close:

Скопировать кодJSX

const ws = new WebSocket("ws://norma.nomoreparties.space/")

ws.close('code', 'reason')

У метода есть два необязательных параметра: code и reason.

code — специальный код закрытие. Это может быть любое число, но есть принятая конвенция кодов. Вот несколько примеров ([обо всех типах можно прочитать по ссылке](https://datatracker.ietf.org/doc/html/rfc6455#section-7.4.1)):

| **Close code (uint16)** | **Codename** | **Description** |
| --- | --- | --- |
| 0 - 999 |  | Обычно не используется. |
| 1000 | CLOSE\_NORMAL | Нормальное закрытие без ошибок. |
| 1001 | CLOSE\_GOING\_AWAY | Страница с открытым сокетом была закрыта клиентом. |
| 1002 | CLOSE\_PROTOCOL\_ERROR | Ошибка внутри протокола фрейма. |
| 1003 | CLOSE\_UNSUPPORTED | Неправильный тип данных. |
| 1004 |  | Зарезервированный тип ошибки. |
| 1005 | CLOSED\_NO\_STATUS | Закрыто без определённого статуса. |

reason — строка, которая описывает причину закрытия соединения.

При закрытии соединения срабатывает функция onclose — подробнее разберём её в следующем уроке.

# События Websocket

В предыдущем уроке мы говорили о том, как отрывать и закрывать соединение. Чтобы работать с WebSocket, нужно разобраться в его событиях, на которые можно «подписаться».

Все события, которые происходят с WebSocket, доступны по соответствующему ключу из из объекта WebSocket. К примеру так выглядят события onopen и onmessage:

Скопировать кодTSX

const ws = new WebSocket("ws://norma.nomoreparties.space/");

ws.onopen

ws.onmessage

Чтобы добавить метод, который будет отвечать за соответствующее событие, нужно приравнять необходимый метод этому событию, например:

Скопировать кодTSX

const ws = new WebSocket("ws://norma.nomoreparties.space/");

ws.onopen = (event: Event) => {

console.log("Соединение установлено")

}

Мы добавили метод для обработки события onopen, который будет выводить сообщение в консоль в случае успешного открытия WebSocket. Вне зависимости от того, на какое событие подписан метод, его параметр — всегда event.

У WebSocket есть четыре типа события — разберём каждый из них подробнее.

## Событие onopen

Начнём с события open. Оно вызывается в случае успешного открытия подключения по WebSocket. Подписаться на него можно так:

Скопировать кодTSX

const ws = new WebSocket("ws://norma.nomoreparties.space/");

ws.onopen = (event: Event) => {

console.log("Соединение установлено");

}

Метод — слушатель события onopen — получает event в качестве параметра. Параметр метода содержит много дополнительной информации (метаданные), описание всех событий и методов, подписанных на эти события. Подробно о типе данных параметра event можно почитать [по ссылке.](https://dom.spec.whatwg.org/#interface-event)

Например, по ключу type параметр event возвратит значение 'open':

Скопировать кодTSX

const ws = new WebSocket("ws://norma.nomoreparties.space/");

ws.onopen = (event: Event) => {

console.log("Соединение установлено");

console.log(event.type); *// open*

}

## Событие onmessage

Это событие вызывается каждый раз, когда приходят данные по протоколу. Иначе говоря, когда сервер отправляет сообщение, в браузере вызывается метод, определённый в качестве слушателя метода onmessage объекта класса WebSocket:

Скопировать кодTSX

const ws = new WebSocket("ws://norma.nomoreparties.space/");

ws.onmessage = (event: MessageEvent) => {

console.log(`Получены данные: ${event.data}`)

}

Рассмотрим onmessage на примере чата. Вы уже знаете, что событие обрабатывается в том случае, если сервер отправляет сообщение с данными. Эти данные хранятся в объекте event, их можно получить по ключу data. Подробнее о других полях event можно почитать в описании его типа данных [MessageEvent](https://html.spec.whatwg.org/multipage/comms.html#messageevent).

Например, пользователи в чате обмениваются сообщениями. Когда арендодатель отправляет сообщение — сервер присылает событие 'сообщение', которое будет содержать объект с такими данными:

Скопировать кодJSON

{

"id": 1, // идентификатор пользователя

"name": 'Иванов Василий' // Имя пользователя

"message": 'Привет, дома? Пришли показатели счётчиков, срочно!' // Текст сообщения

}

Чтобы получить доступ к этому объекту — достаточно обратиться к полю data объекта event, который мы передали как параметр в обработчик события:

Скопировать кодTSX

const ws = new WebSocket("ws://norma.nomoreparties.space/");

ws.onmessage = (event: MessageEvent) => {

const userMessage = event.data;

console.log(userMessage); *// {id: 1, name: 'Иванов Василий', message: 'Привет, дома? Пришли показатели счётчиков, срочно!'}*

}

## Событие onclose

Событие onclose вызывается в случае закрытия соединения вне зависимости от того, было оно закрыто сервером или браузером. Например, вы можете общаться по одному небезызвестному видеочату и собеседник закроет окно. Или же множество пользователей участвовали в онлайн-аукционе по продаже утопленных авто и торги закончились. В обоих случаях сработает событие onclose:

Скопировать кодTSX

const ws = new WebSocket("ws://norma.nomoreparties.space/");

ws.onclose = (event: CloseEvent) => {

if (event.wasClean){

console.log('Соединение закрыто корректно');

console.log(`Код закрытия - ${event.code}`);

console.log(`Причина закрытия - ${event.reason}`)

} else {

console.log('Соединение закрыто с кодом - '

}

}

Для onclose параметр event будет типа данных [CloseEvent](https://developer.mozilla.org/en-US/docs/Web/API/CloseEvent). Одно из полей event указывает на закрытие соединения и называется wasClean. Если сокет был закрыт корректно, wasClean возвращает true, а если некорректно, к примеру по причине отсутствия сети, отображается false.

Конвенция кодов ошибок события onclose аналогична тем, о которых мы говорили [в предыдущем уроке](https://practicum.yandex.ru/trainer/react/lesson/17fa740a-b7eb-4fcf-9e27-de0ef1d8b573).

## Событие onerror

Это событие происходит, если во время соединения возникла ошибка:

Скопировать кодTSX

const ws = new WebSocket("ws://norma.nomoreparties.space/");

ws.onerror = (event: Event) => {

console.log(`Ошибка ${event.message}`)

}

Причины события onerror могут быть разные, например, если в процессе подключения прервалось интернет-соединение, или после подключения были отправлены данные, которые вызвали ошибку подключения.

Но не все ошибки вызовут событие onerror — некоторые из них передадутся с событием onclose. Разберёмся, почему так происходит.

Событие onerror сработает только при ошибке: пропало подключение, возникли проблемы с сервером и другие возможные причины, к примеру пользователь отправляет некорректные данные.

Событие onclose сработает при закрытии соединения: зашёл ли пользователь в свой бункер, где отсутствует сеть, или сам прекратил соединение при закрытии чата.

Поэтому onclose в любом случае отработает при закрытии соединения и ошибки он не ловит, он ловит именно закрытие соединение. Тем не менее можно обратиться к полям объекта event и получить информацию об ошибках. Получается, что onclose и onerror — не взаимозаменяемые слушатели.

Это всё, что нужно знать о типах событий WebSocket. В следующем уроке пойдём дальше и разберёмся, как получать и отправлять данные. А пока перейдём к практике.

import { useCallback, useEffect, useRef } from 'react';

export const CONNECTING = 'CONNECTING';

export const OPEN = 'OPEN';

export const CLOSING = 'CLOSING';

export const CLOSED = 'CLOSED';

export const socketStates = {

0: CONNECTING,

1: OPEN,

2: CLOSING,

3: CLOSED

};

export const useSocket = (url, options) => {

const ws = useRef(null);

const connect = useCallback(

token => {

ws.current = new WebSocket(`${url}?token=${token}`);

ws.current.onopen = event => {

if (typeof options.onConnect === 'function') {

options.onConnect(event);

}

};

ws.current.onerror = event => {

if (typeof options.onError === 'function') {

options.onError(event);

}

};

ws.current.onclose = event => {

if (typeof options.onDisconnect === 'function') {

options.onDisconnect(event);

}

};

},

[url, options]

);

useEffect(

() => {

if (ws.current) {

if (typeof options.onConnect === 'function') {

ws.current.onopen = options.onConnect;

}

if (typeof options.onError === 'function') {

ws.current.onerror = options.onError;

}

if (typeof options.onDisconnect === 'function') {

ws.current.onclose = options.onDisconnect;

}

}

},

[options, ws]

);

useEffect(() => {

return () => {

if (ws.current && typeof ws.current.close === 'function') {

ws.current.close();

}

};

}, []);

const sendData = useCallback(message => {}, [ws]);

return { connect, sendData };

};

**Получение и отправка данных**

В этом уроке разберёмся, какие данные можно передавать с помощью WebSocket и как это делать.

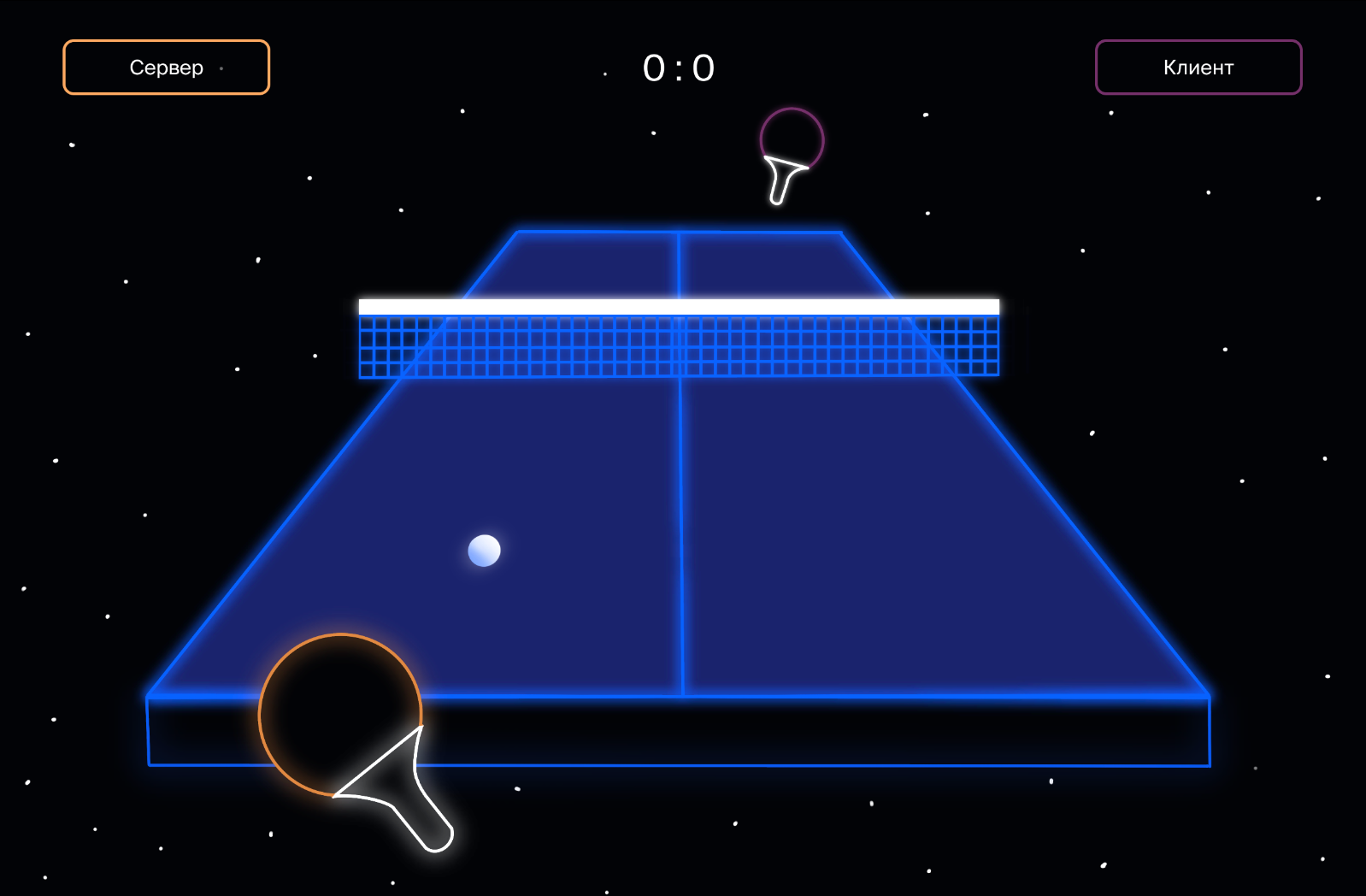
WebSocket оперирует непрерывным обменом данных — назовём их потоком данных. Поток данных в WebSocket состоит из фреймов, или фрагментов данных. Эти данные могут быть отправлены любой из сторон соединения — клиентом или сервером. Рассмотрим виды фреймов и то, как их отправлять.

**Виды фреймов**

Фреймы бывают нескольких видов:

1. Бинарные. Эти фреймы содержат бинарные данные, к примеру Blob и ArrayBuffer, и отправляются обеими сторонами.
2. Текстовые. Они содержат текстовые данные и отправляются обеими сторонами.
3. Пинг-понг. Это сервисный режим, отправляется сервером и используется для поддержания соединения.
4. Закрытие соединения. Этот фрагмент данных используется при закрытии соединения.

При работе с фреймами в браузере мы напрямую взаимодействуем только с бинарными или текстовыми фреймами, остальные фрагменты называются сервисными. Например, пинг-понг фрейм отправляется сервером только для поддержания активного соединения.

*Сыграем?*

Важно помнить, что если WebSocket на протяжении определённого времени не получает и не отправляет какие-либо сообщения, браузер может сам закрыть соединение. Поэтому иногда разработчики добавляют пинг-понг фрейм — это помогает поддерживать подключение в «живом» состоянии:

Скопировать кодTSX

// Пример пинг-понг взаимодействия с сервером

const ws = new WebSocket("ws://norma.nomoreparties.space/");

ws.onmessage = (event: MessageEvent) => {

// ...

if(event.data === 'ping'){

ws.send('pong');

}

// ...

}

С типами фреймов разобрались, теперь посмотрим, как их отправлять.

**Отправка данных**

Для отправки данных используется метод send():

Скопировать кодTSX

const ws = new WebSocket("ws://norma.nomoreparties.space/");

const data: string = 'Пример данных';

ws.send(data)

В качестве параметра метод send() принимает параметр с типами данных string | SharedArrayBuffer | ArrayBuffer | Blob | ArrayBufferView.

То есть send принимает текстовые или бинарные данные. Если приходят текстовые данные, то это всегда строка. Если же данные бинарные, то это могут быть Blob, ArrayBuffer,  SharedArrayBuffer, или  ArrayBufferView**.** Почти всегда нам нужно общаться с сервером с помощью объектов. Например, у объекта сообщения может быть тип TMessage с id пользователя, именем пользователя и телом сообщения:

Скопировать кодTSX

type TMessage = {

id: number;

name: string;

message: string;

};

Мы не можем сразу отправить объект с типом TMessage, поэтому важно сериализовать структуру объекта, то есть привести к строке JSON. Затем с помощью метода JSON.stringify отправить строку в метод .send():

Скопировать кодTSX

const ws = new WebSocket("ws://norma.nomoreparties.space/");

const data: TMessage = {

id: 8,

name: "Евгения Сталевар",

message: "Привет, всё норм?"

};

ws.send(JSON.stringify(data))

Данные отправили, теперь пора их получить. Разберёмся, как это сделать.

**Получение данных**

Для примера рассмотрим процесс получения бинарных данных типа ArrayBuffer:

Скопировать кодJSX

const ws = new WebSocket("ws://norma.nomoreparties.space/");

ws.binaryType = "arraybuffer";

ws.onmessage = (event: MessageEvent) => {

if(event.data instanceof ArrayBuffer) {

const view = new DataView(event.data);

console.log(view.getInt32(0));

}

});

Если тип данных ArrayBuffer, для его представления мы можем использовать класс DataView. Это сделает процесс обращения с поступившими данными более удобным.

Мы также воспользовались функций getInt32 класса DataView, чтобы получать 32-битное число на позиции 0. Таким образом мы сможем принять и преобразовать бинарные данные.

Изменить тип данных можно так:

Скопировать кодJSX

ws.bufferType = "arraybuffer"

Как вы знаете, мы общаемся с сервером с помощью объектов. По умолчанию bufferType равняется значению blob. Это значение нам подходит, ведь нам пришла необработанная строка:

Скопировать кодJSX

// ... Другие поля объекта MessageEvent

data: "{\"success\":true,\"message\":\"Привет!\",\"id\":\"5\",\"username\":\"Василий\"}"

И если до этого мы превращали объект в строку, то сейчас придётся сделать наоборот — собрать из строки объект. Это можно сделать с помощью метода JSON.parse():

Скопировать кодTSX

const ws = new WebSocket("ws://norma.nomoreparties.space/");

ws.onmessage = (event: MessageEvent) => {

const { id, message, username }: TMessage = JSON.parse(event.data)

});

В этом уроке мы разобрались с тем, как отправлять и получать данные. Этих знаний достаточно, чтобы научить клиентскую часть приложения «общаться» с сервером. Теперь пора переходить к заданиям.

import { useCallback, useRef, useEffect } from 'react';

export const CONNECTING = 'CONNECTING';

export const OPEN = 'OPEN';

export const CLOSING = 'CLOSING';

export const CLOSED = 'CLOSED';

export const socketStates = {

0: CONNECTING,

1: OPEN,

2: CLOSING,

3: CLOSED

};

export const useSocket = (url, options) => {

const ws = useRef(null);

const connect = useCallback(

token => {

ws.current = new WebSocket(`${url}?token=${token}`);

ws.current.onmessage = event => {

if (typeof options.onMessage === 'function') {

options.onMessage(event);

}

};

ws.current.onopen = event => {

if (typeof options.onConnect === 'function') {

options.onConnect(event);

}

};

ws.current.onerror = event => {

if (typeof options.onError === 'function') {

options.onError(event);

}

};

ws.current.onclose = event => {

if (typeof options.onDisconnect === 'function') {

options.onDisconnect(event);

}

};

},

[url, options]

);

useEffect(

() => {

if (ws.current) {

ws.current.onmessage = event => {

if (typeof options.onMessage === 'function') {

options.onMessage(event);

}

};

if (typeof options.onConnect === 'function') {

ws.current.onopen = options.onConnect;

}

if (typeof options.onError === 'function') {

ws.current.onerror = options.onError;

}

if (typeof options.onDisconnect === 'function') {

ws.current.onclose = options.onDisconnect;

}

}

},

[options, ws]

);

useEffect(() => {

return () => {

if (ws.current && typeof ws.current.close === 'function') {

ws.current.close();

}

};

}, []);

const sendData = useCallback(

message => {

ws.current.send(JSON.stringify(message))

},

[ws]

);

return { connect, sendData };

};

# Отладка и тестирование WebSocket в консоли разработчика

В этом уроке мы продемонстрируем, как можно тестировать WebSocket в консоли разработчика.

Для начала, после открытия WebSocket отправим сообщение «Привет»:

Скопировать кодJS

const ws = new WebSocket("ws://norma.nomoreparties.space/");

ws.send("Привет");

После этого добавим колбэк для получения сообщений:

Скопировать кодJS

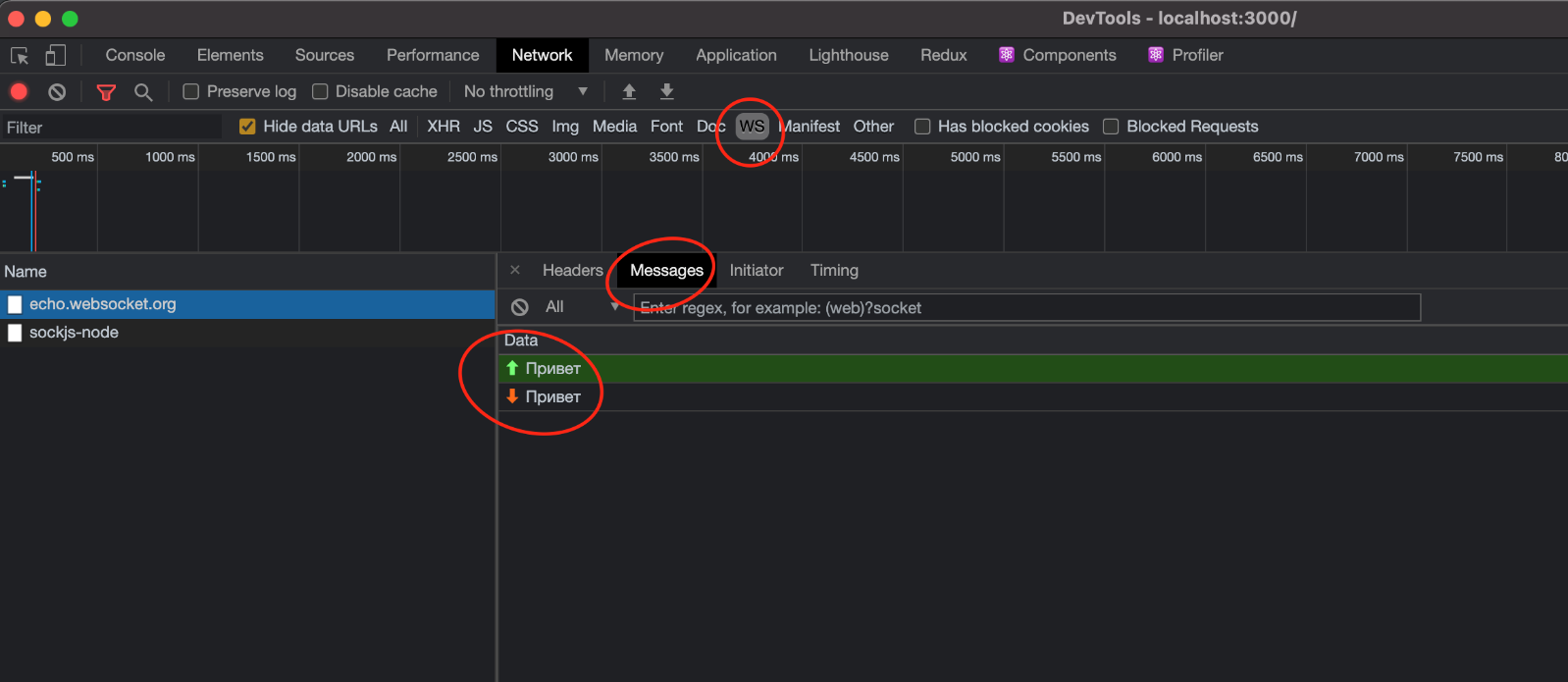
ws.onmessage = (event) => {

console.log(`Получены данныe: ${event.data}`)

}

*// Получены данные: Привет*

Все эти события мы сможем найти в консоли разработчика:

Вкладка WS в консоли разработчика

Во вкладке Network отображаются все запросы и соединения брузера. А вложенная вкладка WS позволяет выбрать только соединения по WebSocket. Во внутренней вкладке Messages можно увидеть все входящие (обозначаются красными стрелками) и исходящие сообщения (помечаются зелёными стрелками).

Как и вся вкладка Network — вкладка WS может помочь в отладке соединения, когда оно некорректно работает. Так же как и во вкладке Network, здесь можно посмотреть статус-коды, адрес по которому было открыто WS-соединение и информацию о данных, которыми обменивается клиент и сервер. Этот инструмент хорошо помогает во время разработки функциональности на WebSocket.

# Принцип реализации WebSocket с Redux

В этой теме поговорим о том, как можно реализовать логику подключения к WebSocket, отправку сообщений с помощью него и обработку ошибок с использованием библиотеки Redux.

Продемонстрируем это на примере простейшего чата, где пользователь будет отправлять сообщение на сервер, а сервер — возвращать его.

Всю логику взаимодействия с WebSocket разместим внутри мидлвара, что позволит разделить зоны ответственности частей приложения и сделать код более читабельным и поддерживаемым.

За подключение к серверу отвечает объект класса WebSocket, который будет создан внутри мидлвара. Все события этого объекта класса будут отправлять экшены в глобальный стейт.

Для получения необходимых значений в приложении создадим селекторы. Структура redux-части проекта будет выглядеть так:

Скопировать кодJSX

redux

├── action-types

│ ├── index.ts

│ └── wsActionTypes.ts

├── middleware

│ ├── index.ts

│ └── socketMiddleware.ts

├── reducers

│ ├── index.ts

│ └── rootReducer.ts

├── selectors

│ ├── getWsConnected.ts

│ ├── index.ts

│ └── wsSelectors.ts

└───store.ts

В следующих уроках разберём мидлвар с инициализацией WebSocket, затем напишем необходимые экшены и редьюсеры, которые пригодятся нам при работе со Store.

# Middleware для работы с WebSocket

В этом уроке разберём процесс создания мидлвара, который отвечает за отправку и получение сообщений, а также за обработку событий закрытия соединения и ошибок. В качестве параметра мидлвар принимает wsUrl, который передаётся в процессе имплементации мидлвара в стор.

Также внутри мидлвара содержится объект класса WebSocket. Он создаётся после того, как мы отправляем экшен WS\_CONNECTION\_START в стор:

Скопировать кодTSX

*// socketMiddleware.ts*

...

if (type === 'WS\_CONNECTION\_START') {

socket = new WebSocket(wsUrl);

}

...

Когда объект класса WebSocket создан, следует подключить функции, которые отвечают за обработку событий.

Сперва реализуем функцию-обработчик события onopen:

Скопировать кодTSX

// socketMiddleware.ts

...

socket.onopen = event => {

dispatch({ type: 'WS\_CONNECTION\_SUCCESS', payload: event });

};

...

Она вызывается после открытия соединения.

Теперь напишем функцию-обработчик события onmessage:

Скопировать кодTSX

// socketMiddleware.ts

socket.onmessage = event => {

const { data } = event;

dispatch({ type: 'WS\_GET\_MESSAGE', payload: data });

};

Эта функция вызывается при получении сообщения от сервера. Сообщение, отправленное сервером, находится в свойстве data объекта event.

Функции-обработчики закрытия соединения и обработки ошибок работают аналогично, их отличает лишь тип экшена.

К примеру, отправка сообщения происходит после отсылки экшена в стор с типом WS\_SEND\_MESSAGE:

Скопировать кодTSX

*// socketMiddleware.ts*

...

if (type === 'WS\_SEND\_MESSAGE') {

const message = payload;

socket.send(JSON.stringify(message));

}

...

Всё готово. Так мидлвар выглядит целиком:

Скопировать кодTSX

*// socketMiddleware.ts*

import type { Middleware, MiddlewareAPI } from 'redux';

import type { AppActions, AppDispatch, RootState } from '../types';

export const socketMiddleware = (wsUrl: string): Middleware => {

return ((store: MiddlewareAPI<AppDispatch, RootState>) => {

let socket: WebSocket | null = null;

return next => (action: AppActions) => {

const { dispatch, getState } = store;

const { type, payload } = action;

if (type === 'WS\_CONNECTION\_START') {

*// объект класса WebSocket*

socket = new WebSocket(wsUrl);

}

if (socket) {

*// функция, которая вызывается при открытии сокета*

socket.onopen = event => {

dispatch({ type: 'WS\_CONNECTION\_SUCCESS', payload: event });

};

*// функция, которая вызывается при ошибке соединения*

socket.onerror = event => {

dispatch({ type: 'WS\_CONNECTION\_ERROR', payload: event });

};

*// функция, которая вызывается при получения события от сервера*

socket.onmessage = event => {

const { data } = event;

dispatch({ type: 'WS\_GET\_MESSAGE', payload: data });

};

*// функция, которая вызывается при закрытии соединения*

socket.onclose = event => {

dispatch({ type: 'WS\_CONNECTION\_CLOSED', payload: event });

};

if (type === 'WS\_SEND\_MESSAGE') {

const message = payload;

*// функция для отправки сообщения на сервер*

socket.send(JSON.stringify(message));

}

}

next(action);

};

}) as Middleware;

};

Мидлвар — функция высшего порядка, для её типизации мы использовали встроенные обобщения redux: Middleware и MiddlewareAPI. В MiddlewareAPI необходимо передать типы dispatch и state нашего хранилища. Эти типы можно получить так:

Скопировать кодTSX

*// src/store/types/index.ts*

export type AppDispatch = typeof store.dispatch;

export type RootState = ReturnType<typeof store.getState>;

У нас получился мидлвар, который принимает на вход параметр URL сокет-соединения. Написанный мидлвар является не совсем унифицированным — мы можем инстанцировать несколько мидлваров для разных сокет-соединений, но типы экшенов сейчас строго прописаны строками, например type === 'WS\_CONNECTION\_START'. В задачах вы встретите socketMiddleware, который принимает два параметра: wsUrl: string и wsActions: TWSStoreActions. С помощью этих параметров мы сделали код унифицированным.

Об инстанцировании мидлвара и конфигурации store мы расскажем в конце темы, а в следующей — опишем необходимые типы экшенов.

**Actions для работы с WebSocket**

В предыдущем уроке мы создали мидлвар для работы с WebSocket. Теперь поговорим о том, какие типы экшенов будем использовать для работы с редьесерами и мидлварами.

Из предыдущей темы вы знаете, как организован процесс создания экземпляра WebSocket. А ещё вы знакомы с целым набором обработчиков событий: onmessage, onclose, onerror, send. Чтобы отобразить это на «языке» Redux, потребуется каждому из методов присвоить свой тип экшена. Определим типы экшенов. Это будут константы со строковыми литеральными типами:

* для создания объекта класса WebSocket:

Скопировать кодTSX

*// wsActionTypes.ts*

export const WS\_CONNECTION\_START: 'WS\_CONNECTION\_START' = 'WS\_CONNECTION\_START';

...

* при успешном соединении:

Скопировать кодTSX

*// wsActionTypes.ts*

...

export const WS\_CONNECTION\_SUCCESS: 'WS\_CONNECTION\_SUCCESS' = 'WS\_CONNECTION\_SUCCESS';

...

* в случае ошибки соединения:

Скопировать кодTSX

*// wsActionTypes.ts*

...

export const WS\_CONNECTION\_ERROR: 'WS\_CONNECTION\_ERROR' = 'WS\_CONNECTION\_ERROR';

...

* при закрытии соединения:

Скопировать кодTSX

*// wsActionTypes.ts*

...

export const WS\_CONNECTION\_CLOSED: 'WS\_CONNECTION\_CLOSED' = 'WS\_CONNECTION\_CLOSED';

...

* при получении сообщения от сервера:

Скопировать кодTSX

*// wsActionTypes.ts*

...

export const WS\_GET\_MESSAGE: 'WS\_GET\_MESSAGE' = 'WS\_GET\_MESSAGE';

...

* для отправки сообщений на сервер:

Скопировать кодTSX

*// wsActionTypes.ts*

...

export const WS\_SEND\_MESSAGE: 'WS\_SEND\_MESSAGE' = 'WS\_SEND\_MESSAGE';

...

Мы определили необходимые типы экшенов — вы уже видели их в примере с мидлварой из предыдущего урока. Типы экшенов понадобятся ещё и в следующем уроке — при создании редьюсеров для работы с WebSocket.

# Reducers для работы с WebSocket

В этом уроке создадим редьюсер, который отвечает за работу с данным, отправленными с сервера, и данными, отправленными клиентом на сервер.

Инициализируем базовое состояние стора для работы с WebSocket. А после этого добавим туда редьюсер. Опишем начальное состояние:

Скопировать кодTSX

// rootReducer.ts

type TWSState = {

wsConnected: boolean;

messages: IMessage[];

error?: Event;

}

const initialState: TWSState = {

wsConnected: false,

messages: []

};

Ключ wsConnected используется для отображения статуса соединения. А по ключу messages мы будем хранить все входящие сообщения. Ключ error будет содержать информацию об ошибке, если получим её от сокета.

Теперь опишем саму функцию редьюсера. Это обычный редьюсер, в котором используется массив для хранения сообщений с сервера и ключ wsConnected — для хранения состояния подключения WebSocket.

При успешном подключении отправляется экшен с типом WS\_CONNECTION\_SUCCESS, а ключу wsConnected присваивается значение true. В случае ошибки подключения отправляется экшен с типом WS\_CONNECTION\_ERROR, а в ключ error передаётся значение из action.payload, которое содержит информацию об ошибке, при этом ключу wsConnected присваивается значение false. Также и при закрытии соединения отправляется экшен с типом WS\_CONNECTION\_CLOSED, а ключ wsConnected получает значение false.

А так редьюсер выглядит целиком:

Скопировать кодTSX

*// rootReducer.ts*

import {

WS\_CONNECTION\_SUCCESS,

WS\_CONNECTION\_ERROR,

WS\_CONNECTION\_CLOSED,

WS\_GET\_MESSAGE

} from '../action-types';

import type { IMessage, TWSActions } from '../types';

type TWSState = {

wsConnected: boolean;

messages: IMessage[];

error?: Event;

}

const initialState = {

wsConnected: false,

messages: []

};

*// Создадим редьюсер для WebSocket*

export const wsReducer = (state = initialState, action: TWSActions) => {

switch (action.type) {

*// Опишем обработку экшена с типом WS\_CONNECTION\_SUCCESS*

*// Установим флаг wsConnected в состояние true*

case WS\_CONNECTION\_SUCCESS:

return {

...state,

error: undefined,

wsConnected: true

};

*// Опишем обработку экшена с типом WS\_CONNECTION\_ERROR*

*// Установим флаг wsConnected в состояние false и передадим ошибку из action.payload*

case WS\_CONNECTION\_ERROR:

return {

...state,

error: action.payload,

wsConnected: false

};

*// Опишем обработку экшена с типом WS\_CONNECTION\_CLOSED, когда соединение закрывается*

*// Установим флаг wsConnected в состояние false*

case WS\_CONNECTION\_CLOSED:

return {

...state,

error: undefined,

wsConnected: false

};

*// Опишем обработку экшена с типом WS\_GET\_MESSAGE*

*// Обработка происходит, когда с сервера возвращаются данные*

*// В messages передадим данные, которые пришли с сервера*

case WS\_GET\_MESSAGE:

return {

...state,

error: undefined,

messages: [...state.messages, action.payload]

};

default:

return state;

}

};

Всё готово — написан мидлвар и создан редьюсер. Самое время перейти к инициализации Store. Разберём это в следующем уроке.

# Конфигурация Store и подключение Middleware

Конфигурация Store — отправная точка при создании Redux-приложения. Именно на этом этапе мы создаём store, добавляем редьюсеры и подключаем мидлвары, которые написали в предыдущих уроках.

Для создания Store воспользуемся [функцией createStore из библиотеки redux](https://www.notion.so/4-reducer-action-d0d747b1477c4f88b8443bdcdf50d764):

Скопировать кодTSX

*// store.ts*

import { createStore } from 'redux';

Также применим редьюсер из файла reducers.js:

Скопировать кодTSX

*// store.ts*

...

import { rootReducer } from './reducers'

А мидлвар возьмём из файла middleware.ts:

Скопировать кодTSX

*// store.ts*

...

import { socketMiddleware } from './middleware'

Инициализируем хранилище с помощью функции createStore. В эту функцию передадим rootReducer. Упакуем createStore в функцию configureStore, которая будет возвращать store:

Скопировать кодJSX

*// store.ts*

import { createStore } from 'redux';

import { rootReducer } from './reducers'

import { socketMiddleware } from './middleware'

export const store = createStore(

rootReducer,

*// Здесь подключим миддлвару socketMiddleware*

);

Для работы с объектом класса WebSocket используем socketMiddleware, который принимает в качестве параметра URL тестового WebSocket. Это единственный мидлвар, который обрабатывает все события объекта класса WebSocket. Для подключения используем applyMiddleware из redux.

Наш файл store.ts будет выглядеть так:

Скопировать кодTSX

*// store.ts*

import { createStore, applyMiddleware } from 'redux';

import { composeWithDevTools } from 'redux-devtools-extension'

import { rootReducer } from './reducers'

import { socketMiddleware } from './middleware'

export const store = createStore(

rootReducer,

composeWithDevTools(applyMiddleware(socketMiddleware('wss://echo.websocket.org')))

);

Вот и все: Store готов, редьюсеры и мидлвары подключены. Теперь сокет-соединение работает корректно, а вся информация собирается в хранилище.