

Продвинутые Асинхронные Механизмы в JavaScript и node.js




На примере WebSocket

Обо мне

- SDET с 2018
- Senior SDET @b2broker
- Certified node.js application developer (JSNAD 2023)
- автор TG канала @haradkou_sdet
- Иногда ментор и консультирую компании



Agenda

- Websocket ↔
- Сообщения и логика 
- Заккрытие сокета 
- Следим за памятью 

Контекст проекта

- Trading terminal
- Node.js as a client to .NET server
- .NET signalR in server

Трейдинг терминал

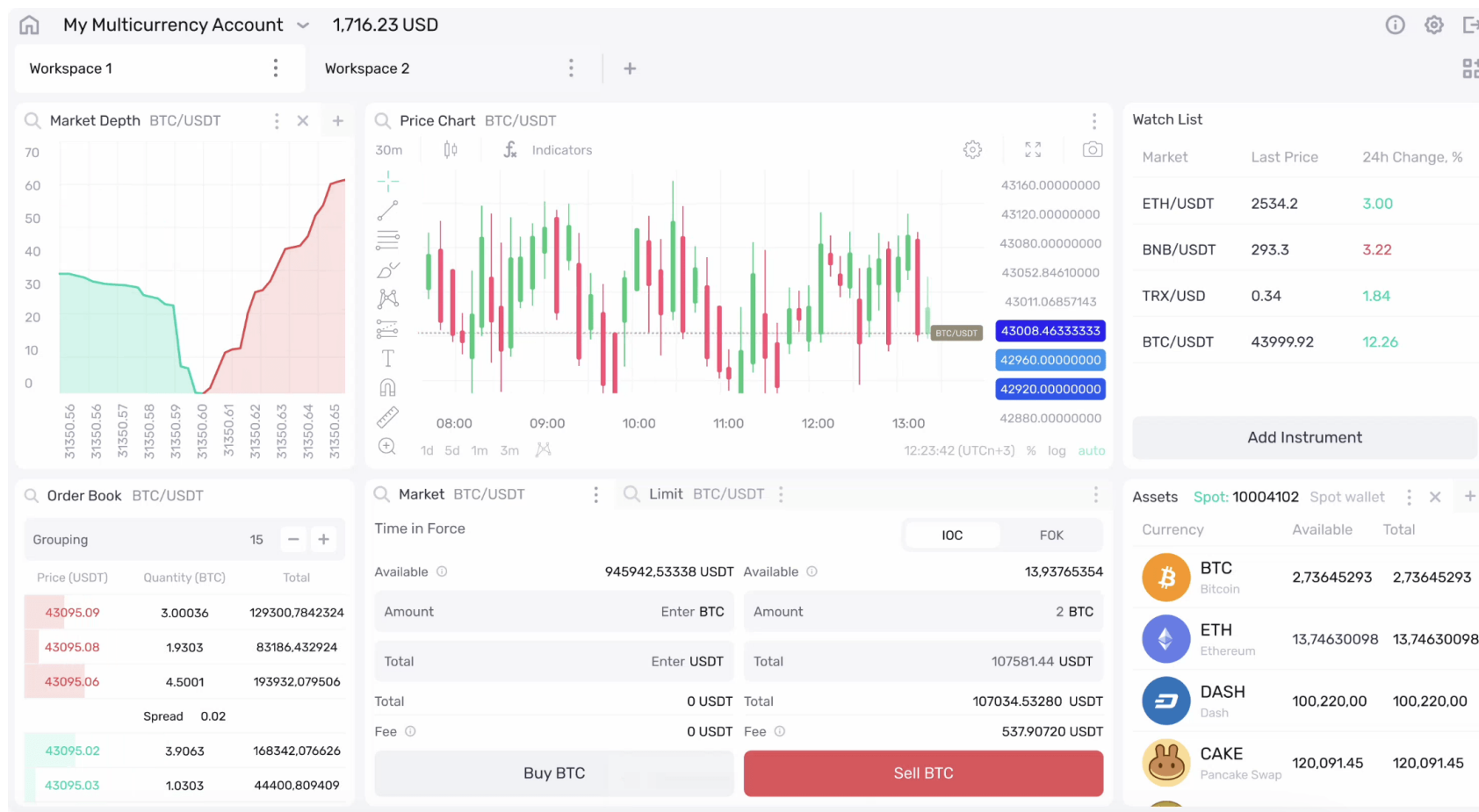


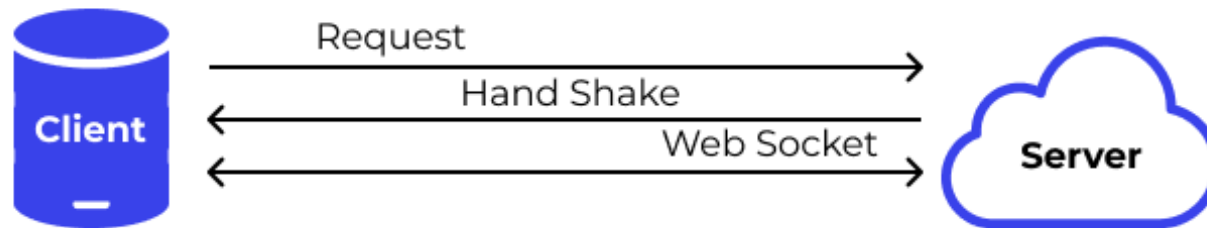
Фото из официального сайта b2broker.com/bbp

Проблемы

- нужно закрывать сокет самим
- тесты параллельные, что может приводить к утечкам памяти
- разнообразная логика на приходящие сообщения

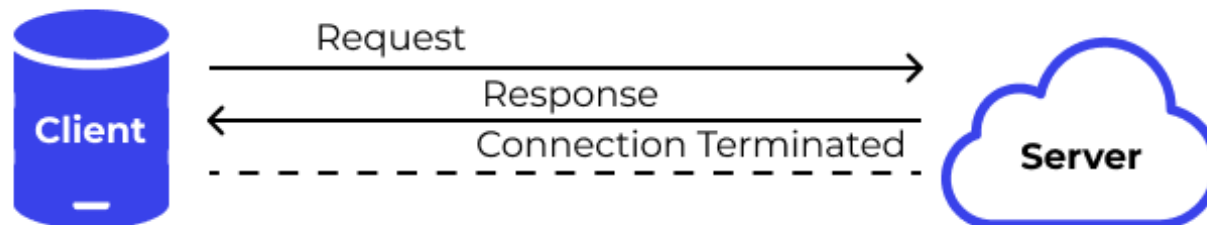
WebSocket

WebSocket Connection



VS

HTTP Connection



WS API

```
1 import WebSocket from 'ws';
2
3 const ws = new WebSocket('ws://www.host.com/path');
4
5 ws.on('open', () => {
6   console.log('socket open!');
7 })
8
9 ws.on('message', (msg) => {
10   console.log('socket message:', msg);
11 })
12
13 ws.on('error', (err) => {
14   console.log('socket error:', err);
15 })
16
17 ws.on('close', () => {
18   console.log('socket close!');
19 })
```


Плюсы ws

- большое комьюнити
- понятная технология
- прост в создании ПОС

Минусы ws

- события добавляют когнитивную нагрузку - нужно помнить о том, что есть подписки
- нужно уметь управлять жизненным циклом
- событие на сообщение может иметь большую логику
 - нет встроенного итератора по сообщениям, например для фильтрации по какому-либо признаку
- логирование\трассировка
 - как понять что освободилась память после закрытия сокета?
 - как не потерять контекст в случае работы с внешними данными

Нужно тюнить!



Набросок

```
1  import ws from 'ws'
2
3  class MyWs {
4      _ws: ws
5      _messages: WsMessage[] = []
6      _events: Function[] = []
7      on(...args): void {}
8      emit(): void {}
9  }
```

Ч1. итератор по сообщениям

проблема: события могут иметь
большую логику обработки

Ч1. Ждем события

```
1 export function waitForEvent<T>(
2   source: EventEmitter,
3   eventName: string
4 ) {
5   return new Promise<T>((resolve, reject) => {
6     const eventHandler = (data: T) => {
7       source.off(eventName, eventHandler)
8       resolve(data)
9     }
10    source.once(eventName, eventHandler)
11  })
12 }
```

41. Итератор

```
1 import ws from 'ws'
2 import waitForEvent from './wait'
3
4 class MyWs {
5   _ws: ws
6   _events: Function[] = []
7   on(event: string, ...args): void {}
8   emit(event: string): void {}
9   async *[Symbol.asyncIterator]() {
10     const message = await waitForEvent(this._ws, 'message')
11     yield message
12   }
13   async *[Symbol.iterator]() {
14     const message = await waitForEvent(this._ws, 'message')
15     yield message
16   }
17 }
```

Ч1. Используем

```
1 import WebSocket from './my-ws'
2
3 async function main(){
4   const ws = new WebSocket(***/)
5   for await (const message of ws) {
6     console.log('Got message!', message)
7     if(message === criteria) break
8   }
9 }
10
11 main()
```


Ч1. Бонус

```
1 import ws from 'ws'
2 import waitForEvent from './wait'
3
4 class MyWs {
5     async *[Symbol.asyncIterator]() { /** реализация */ }
6     async *[Symbol.iterator]() { /** реализация */ }
7
8     messageFilter(cb: (data: Data) => boolean) {
9         for await (const data of this) {
10             const isMatched = cb(data);
11             if(isMatched) yield data;
12             else continue;
13         }
14     }
```

Ч1. Бонус

```
1 import WebSocket from "./my-ws";
2
3 async function main() {
4     const ws = new WebSocket(** */);
5     for await (const message of ws.messageFilter(
6         (data) => typeof data === "string",
7     )) {
8         console.log("Got string message!", message);
9     }
10 }
11
12 main();
```

41. Бонус Iterator Helpers

```
1 import WebSocket from "./my-ws";
2
3 async function main() {
4     const ws = new WebSocket(** */);
5     for await (
6         const message of ws
7             .take(10)
8             .filter((data) => typeof data === "string")
9     ) {
10         console.log("Got only 10 string messages!", message);
11     }
12 }
13
14 main();
```

Ч2. Автоматическое заккрытие сокета

1. Timeout

- 1. Promise.race + setTimeout

- 2. Abort signal

2. AsyncDispose

42. Timeout

```
1  import ws from 'ws'
2
3  class MyWs {
4      constructor(opts){
5          this.signal = opts.signal ?? null
6
7          if(this.signal) {
8              this.signal.onabort = async (e) => {
9                  this.abort(e)
10             }
11         }
12     }
13
14     abort(reason) {
15         this.close(reason)
16     }
17
18     close(reason) { /* реализация */ }
19 }
```

Ч2. используем Timeout

```
1 import WebSocket from './my-ws'
2
3 async function main(){
4   const signal = AbortSignal.timeout(3000)
5   const ws = new WebSocket({ signal })
6   // ws automatically closes after 3 sec
7   ws.on('message', (msg) => {
8     console.log(msg)
9   })
10 }
11
12 main()
```

(Async) Dispose

- Stage 2
- Typescript 5.2+
- Тоже что и **defer** в Go/Zig/Swift
- По аналогии с **iterator** есть *async* версия

```
1  function loggy(id: string): AsyncDisposable {
2      console.log(`Constructing ${id}`);
3      return {
4          async [Symbol.asyncDispose]() {
5              console.log(`Disposing (async) ${id}`);
6              await doWork();
7          },
8      }
9  }
10 async function func() {
11     await using a = loggy("a");
12     {
13         await using c = loggy("c");
14         await using d = loggy("d");
15     }
16 }
17 func();
18 // Constructing a
19 // Constructing c
20 // Constructing d
21 // Disposing (async) d
22 // Disposing (async) c
23 // Disposing (async) a
```


42. Async Dispose

```
1 import ws from 'ws'
2 import waitForEvent from './wait'
3
4 class MyWs implements AsyncDispose {
5     _ws: ws
6     on(...args): void {}
7     emit(): void {}
8     async close() {}
9
10    async *[Symbol.asyncIterator]() { /** реализация */ }
11    async *[Symbol.iterator]() { /** реализация */ }
12
13    async [Symbol.asyncDispose]() {
14        await this._ws.close()
15    }
16 }
```

Ч2. Используем

```
1 import WebSocket from './my-ws'
2
3 async function main(){
4     await using ws = new WebSocket(** */)
5     for await (const message of ws) {
6         console.log('Got message!', message)
7         if(message === criteria) break
8     }
9     // ws automatically closes here
10 }
11
12 main()
```

Подводные камни

**!Нельзя передать как аргумент
в другие функции!**

ЧЗ. Следим за памятью 🙄🙄

Finalization Registry



Дисклеймер! Эта фича у нас только
планируется

















ЧЗ. Для чего?

- Для трассировки
- Очистка ресурсов
- Менеджмент кэша

Ч3. Минусы

- недетерминированное поведение GC
- не использовать для важных ресурсов!

43. Can I use?

													
	 Chrome	 Edge	 Firefox	 Opera	 Safari	 Chrome Android	 Firefox for Android	 Opera Android	 Safari on iOS	 Samsung Internet	 WebView Android	 Deno	 Node.js
<code>FinalizationRegistry</code>	✓ 84	✓ 84	✓ 79	✓ 70	✓ 14.1	✓ 84	✓ 79	✓ 60	✓ 14.5	✓ 14.0	✓ 84	✓ 1.0	✓ 14.6.0
<code>FinalizationRegistry()</code> constructor	✓ 84	✓ 84	✓ 79	✓ 70	✓ 14.1	✓ 84	✓ 79	✓ 60	✓ 14.5	✓ 14.0	✓ 84	✓ 1.0	✓ 14.6.0

Ч3. Как это выглядит?

```
1 import WebSocket from 'ws';
2
3 // Создаем новый FinalizationRegistry и регистрируем callback функцию
4 const registry = new FinalizationRegistry((ws) => {
5   // send analytics
6 });
7
8 class MyWs {
9   constructor(url) {
10     // Регистрируем объект в FinalizationRegistry
11     registry.register(this, this.ws);
12   }
13
14   close() {
15     if (this.ws) {
16       this.ws.close();
17       registry.unregister(this);
18     }
19     this.ws = null
20   }
21 }
```


Не покрытые темы

- Web Streams
- Async Resource
- Async LocalStorage

Спасибо за внимание!



Список литературы

- [Symbol.iterator](#)
- [Symbol.asyncIterator](#)
- [Iterator helpers](#)
- [Async Dispose](#)
- [Finalization Registry](#)
- [Abort Signal](#)
- [Abort Controller](#)