

**Тяжело в учении,  
сложно при внедрении,  
быстро в использовании**

**Павел Дадыкин**

ROGII



**Frontend  
Conf 2024**

# Обо мне

## Павел Дадыкин

- 9 лет во фронтенде
- 2 года тимлид на проекте StarLite Web
- Первый раз на FrontendConf



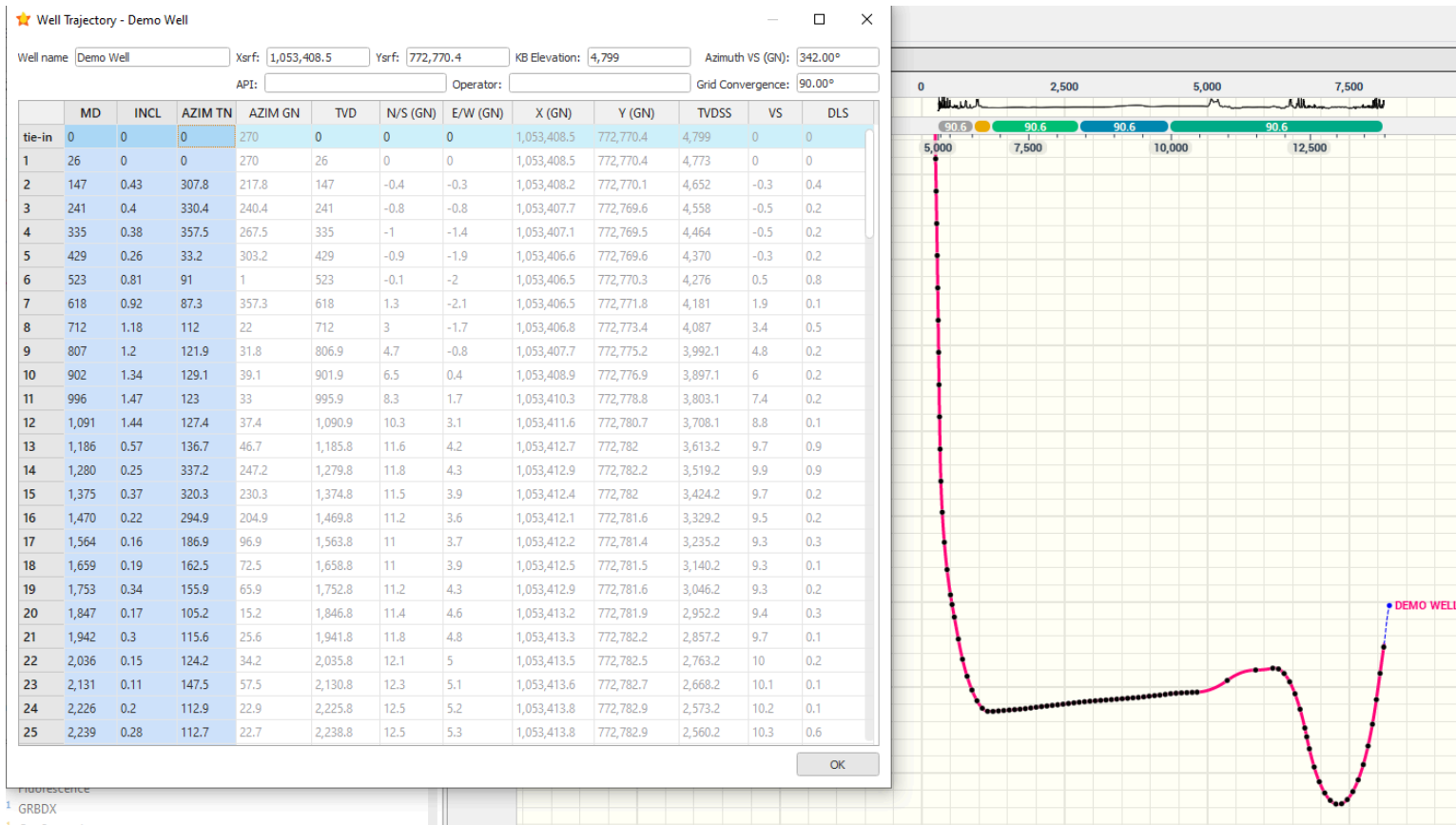
# О чём доклад

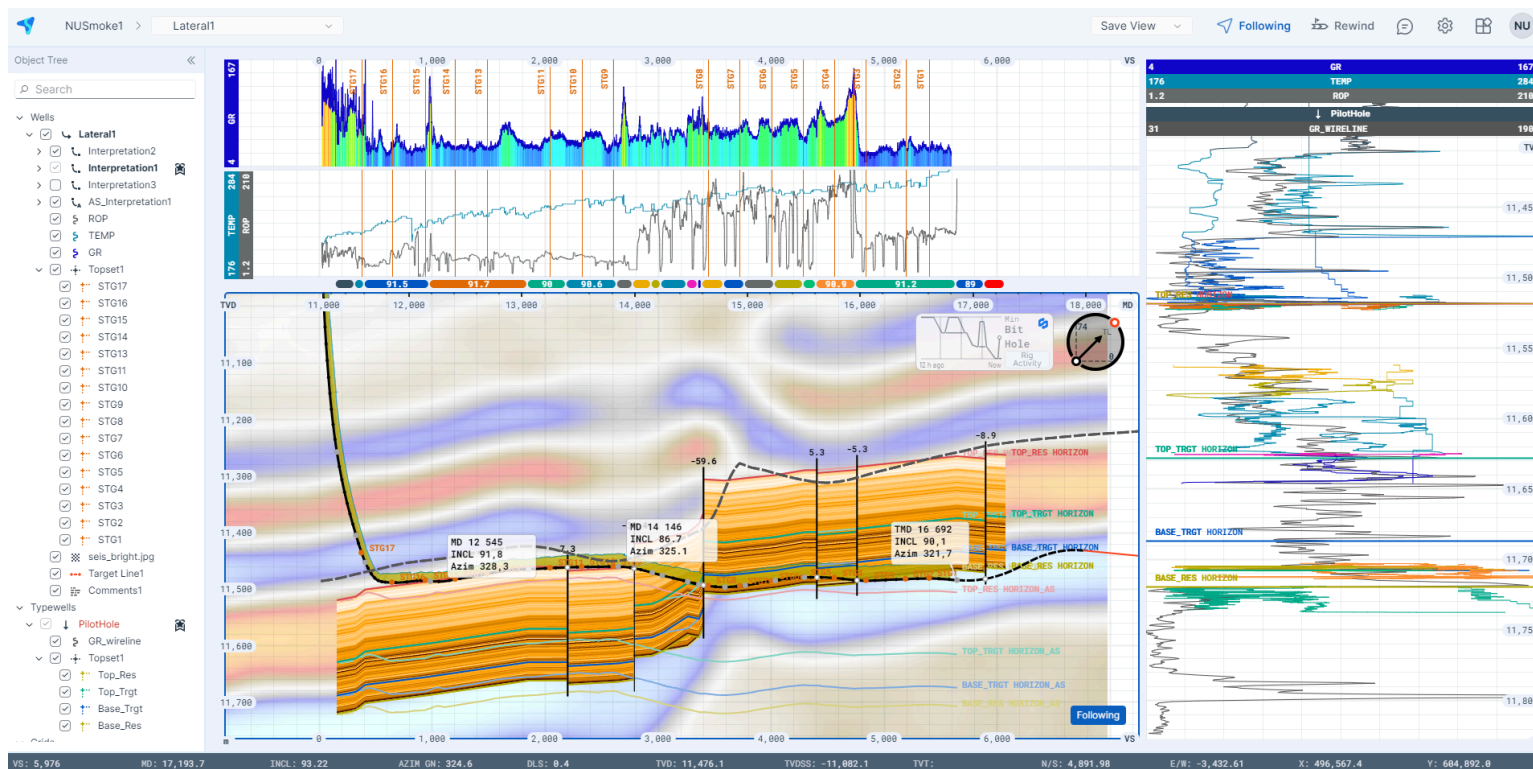
- История внедрения WebAssembly на реальном проекте
- Подводные камни и решение проблем
- Стоит ли вам использовать WebAssembly в своих проектах?



- Разработка продуктов для нефтегазовой индустрии
- Geoscience-решения
- Много математических расчётов
- Desktop, Web, iOS, Android, SDK, Public API

# Что за расчёты?





# Эволюция продуктов ROGII

1. Расчёты написаны для Desktop App (на C++)
2. Хранение данных в Cloud
3. Повторение расчётов в Web, iOS, Android

# Проблемы

- Дублирование расчётов на разных платформах
- Низкая производительность при вычислениях
- Сложности добавления/изменения расчётов



# Цель

- Внедрить в приложение расчёты, написанные на C++
- Сохранить при этом текущий стек технологий:
  - React
  - **Typescript**
  - **Webpack**
  - **Web workers**
  - Canvas
  - WebGL

# WebAssembly (WASM)

- Формат байт-кода, исполняемого современными браузерами
- Может взаимодействовать с JavaScript



# Плюсы WASM

- Обеспечивает высокую скорость исполнения
- Поддержка всеми браузерами
- Компиляция в WASM доступна для множества языков  
(C, **C++**, C#, Rust, Elixir, Erlang, Go, TypeScript, D, Kotlin)

# Минусы WASM

- Чёрный ящик
- Внешняя типизация
- Конвертация данных  $JS \rightarrow WASM \rightarrow JS$

# Доклады про WASM

- [Разработка под WebAssembly: реальные грабли и примеры](#)  
[\(Андрей Нагих, FC 2018\)](#)
- [RUST + WEBASSEMBLY \(Илья Барышников, FC 2019\)](#)
- [WebAssembly SPA-фреймворки \(FC 2020\)](#)

# Тяжело в учении

Примеры использования WASM в сети (конец 2021 года):

- A + B
- Factorial
- Fibonacci



**Как же скомпилировать C++ в WASM?**

# Компиляция в WASM

- [Emscripten](#)
- [Binaryen](#)
- [Компиляция C в WebAssembly без Emscripten](#)



# Emscripten

- Компиляция из C/C++/LLVM-language в WebAssembly
- Запуск этого кода в Web/Node.js
- Поддержка библиотек C/C++



# Документация Emscripten

The screenshot displays the GitHub web interface for the `emscripten-core/emscripten` repository, specifically the `src/settings.js` file. The left sidebar shows the file tree with `settings.js` selected. The main area shows the code for `settings.js`, which is 2260 lines long (2021 loc) and 103 KB in size. The code includes various debugging settings and runtime options, such as `RUNTIME_DEBUG`, `LEGACY_RUNTIME`, and `OFFSCREEN_FRAMEBUFFER_FORBID_VAO_PATH`.

```
2138 // This setting is enabled by default if any of the following debugging settings
2139 // are enabled:
2140 // - PTHREADS_DEBUG
2141 // - DYLINK_DEBUG
2142 // - LIBRARY_DEBUG
2143 // - GL_DEBUG
2144 // - OPENAL_DEBUG
2145 // - EXCEPTION_DEBUG
2146 // - SYSCALL_DEBUG
2147 // - WEBSOCKET_DEBUG
2148 // - SOCKET_DEBUG
2149 // - FETCH_DEBUG
2150 // [link]
2151 var RUNTIME_DEBUG = false;
2152
2153 // Include JS library symbols that were previously part of the default runtime.
2154 // Without this, such symbols can be made available by adding them to
2155 // DEFAULT_LIBRARY_FUNCS_TO_INCLUDE, or via the dependencies of another JS
2156 // library symbol.
2157 var LEGACY_RUNTIME = false;
2158
2159 // User-defined functions to wrap with signature conversion, which take or return
2160 // pointer argument. Only affects MEMORY64=1 builds, see create_pointer_conversion_wrappers
2161 // in emscripten.py for details.
2162 // Use _ for non-pointer arguments, p for pointer/153 arguments, and P for optional pointer/153 values.
2163 // Example use -sSIGNATURE_CONVERSIONS=someFunction:_p,anotherFunction:p
2164 // [link]
2165 var SIGNATURE_CONVERSIONS = [];
2166
2167 //=====
2168 // Internal, used for testing only, from here
2169 //=====
2170
2171 // Internal (testing only): Disables the blitOffscreenFramebuffer VAO path.
2172 // [link]
2173 var OFFSCREEN_FRAMEBUFFER_FORBID_VAO_PATH = false;
2174
2175 // Internal (testing only): Forces memory growing to fail.
2176 // [link]
2177 var TEST_MEMORY_GROWTH_FAILS = false;
```

**Дефолтной конфигурации Emscripten  
не существует!**

# Флаги компиляции

01. `-s DYNAMIC_EXECUTION=0 -s ENVIRONMENT=worker`

02. `-s MODULARIZE=1 -s ALLOW_MEMORY_GROWTH=1 --bind`

- Выключаем использование eval
- **Задаем выполнение в веб-воркере**
- Оборачиваем в модульный код (без глобальных переменных)
- Разрешаем автоматический рост памяти
- Говорим привязать все функции к js

**Где же взять описание функций?**

# Генерация документации

## Doxygen

### ◆ calculateTHLProjectionFor3DPoint()

```
double calculateTHLProjectionFor3DPoint ( CGeomathCalculatedTrajectoryPointCollection calculatedTrajectory,  
                                          CGeomath3DPoint point,  
                                          CGeomathWellHead wellHead  
                                          )
```

calculates THL-projection for 3D point

#### Parameters

- [in] **calculatedTrajectory** Calculated points of the trajectory. Can be obtained by calling calculateTrajectory.
- [in] **point** Point which thl is needed.
- [in] **wellHead** Wellhead data of the lateral, in the projection of which calculations are performed.

#### Returns




Returns THL-projection.

**Как сделать удобной отладку ошибок?**

# Разные сборки (через stake)




## Debug

- 👎 Большой вес файла
- 👎 Работает медленнее
- 👍 Детальное описание ошибок

| Name  | Size      |
|---|-----------|
|  package.json                  | 182 bytes |
|  starsteermath_javascript.js   | 113.5 kB  |
|  starsteermath_javascript.wasm | 32.1 MB   |

## Release

- 👍 Маленький вес файла
- 👍 Работает быстро
- 👎 Сложно отлаживать

| Name  | Size      |
|---|-----------|
|  package.json                  | 184 bytes |
|  starsteermath_javascript.js   | 37.8 kB   |
|  starsteermath_javascript.wasm | 387.7 kB  |



# Утечки памяти

- Добавили дополнительный флаг в сборку
- Он передаёт флаг `-fsanitize=address -g2` компилятору
- Вызов функции для принудительного сбора статистики

# Как реализовать версионирование?

# Версионирование

Emscripten на выходе даёт 2 файла:

- Бинарный .wasm
- Обвязка в виде .js-файла



# Версионирование

Добавить в начало JS-файла комментарий

01. /\*

02.   Version: 1.3.0

03.   Commit: 681def4aefb99c9de5357617dc138b66e6ae1d0b

04.   Pipeline: 103990

05. \*/

# Версионирование

- Используем [Nexus](#)
- Собираем в npm-пакет
- Учитываем вариант сборки (Debug/Release)

01. "dependencies": {

02.     "@nexus-npm/wasm": "0.9.0-Release",

03. }

# Подключение WASM к проекту

# Как настроить Webpack?

# Webpack

[webassembly-loader](#)

[wasm-loader](#)

Repository

github.com/ballercat/wasm-loader

Homepage

github.com/ballercat/wasm-loader#rea...

Weekly Downloads

8,286



Version

1.3.0

License

MIT

Unpacked Size

24.3 kB

Total Files

11

Issues

0

Pull Requests

0

Last publish

6 years ago



# Webpack 4 file-loader

```
01. {  
02.   test: /\.wasm$/,  
03.   type: "javascript/auto",  
04.   use: [{  
05.     loader: "file-loader",  
06.     options: { name: "wasm/[name].[hash].[ext]" }  
07.   }]  
08. }
```

# Загрузка WASM в приложение

```
01. import wasm from 'math-js.wasm';  
02.  
03. const response = await client.get(wasm, {  
04.     responseType: 'arraybuffer'  
05. });  
06.  
07. const wasmBinary = response.data;
```

# Webpack 5

Asset Modules на замену file-loader

syncWebAssembly / asyncWebAssembly

**Как добавить типизацию?**

# Typescript

- [@types/emscripten](https://github.com/microsoft/typescript) для вспомогательных методов
- Описание структур и функций вручную, используя Doxygen
- Автоматическая генерация **.ts** из **.hpp**

# CSP Error

“*Refused to compile or instantiate WebAssembly module because 'unsafe-eval' is not an allowed source of script in the following Content Security Policy directive: "script-src https:".*

Content-Security-Policy: script-src 'wasm-unsafe-eval'

**Как конвертировать данные JS  $\leftrightarrow$  WASM?**

# Типы данных

## JS

- Массивы объектов
- Автоматическое выделение памяти
- Автоматическая сборка мусора

## C/C++

- `std::vector` / `TypedArray` /  
Динамические массивы
- Требуется выделение памяти
- Нужно освобождать выделенную память



# Конвертация JS ↔ WASM

```
01. const pointer = instance._malloc(size * Float64Array.BYTES_PER_ELEMENT);  
02.  
03. instance.HEAPF64.set(  
04.   new Float64Array(size),  
05.   pointer / Float64Array.BYTES_PER_ELEMENT,  
06. );  
07.  
08. instance._free(trajjectory.mdArray.pointer);
```

# Добавление обёртки

- Добавили метаданные для функций конвертации и высвобождения памяти
- Emscripten сам конвертирует типы в C-структуры
- Передаём сразу массив объектов
- Вызываем `freeCollection(data)`, когда массив не нужен

01. `convertToJSObject<WasmType, JSArrayType>(data)`

02. `convertFromJSObject<JSArrayType, WasmType>(data)`

# Undefined / NaN

В **TypeScript** мы можем написать

```
type X = number | undefined
```

В **C/C++** так нельзя!

Из функций расчёта в этом случае нам возвращается **NaN**

# Undefined → NaN

```
01. // Преобразование для передачи в модуль расчёта
02. point.data ?? NaN
03.
04. // Проверка на наличие значения
05. if (point.data && !isNaN(point.data))
```

# Batch запросов

- Большая часть времени тратится на преобразования из структур данных JS в WASM и обратно
- Лучше выполнить одну большую операцию, чем много маленьких

# Batch запросов

```
01. for (const element of elements) {
```

```
02.   const point = calculatePoint(element.x);
```

```
03.   points.push(point);
```

```
04. }
```

```
01. const xArray = elements.map((element) => element.x);
```

```
02. const points = wasm.calculatePoints(xArray);
```

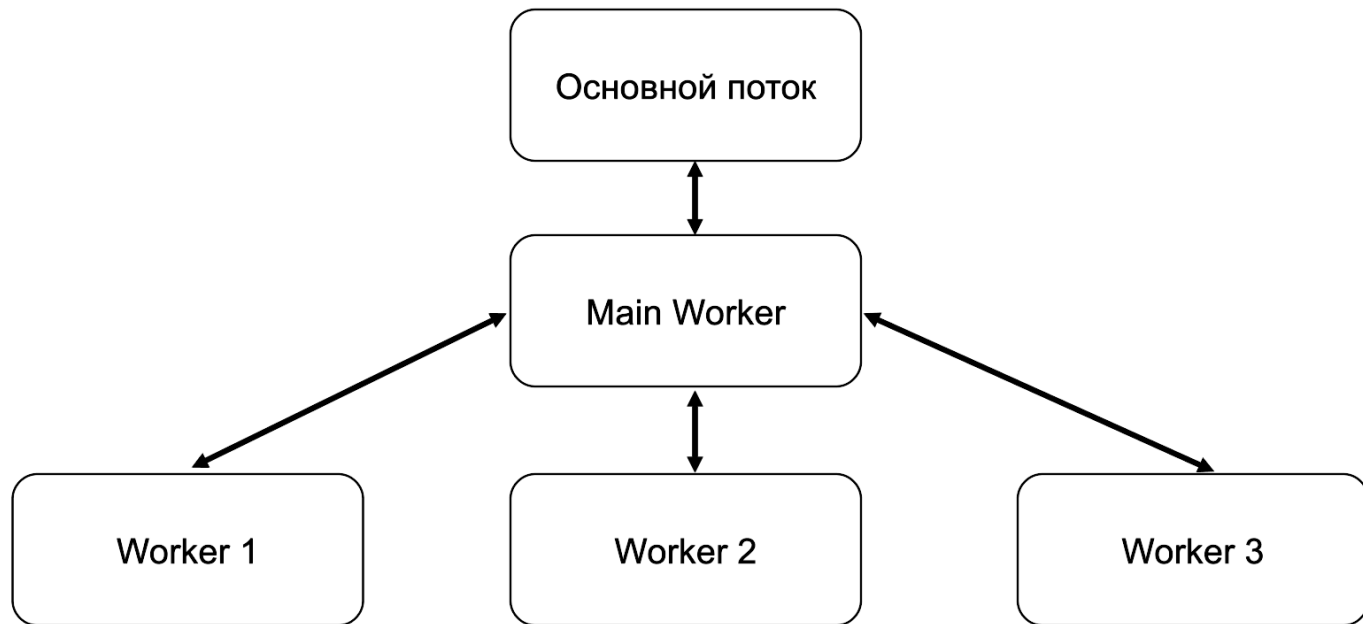
**Как всё подружить с веб-воркерами?**

# Web workers

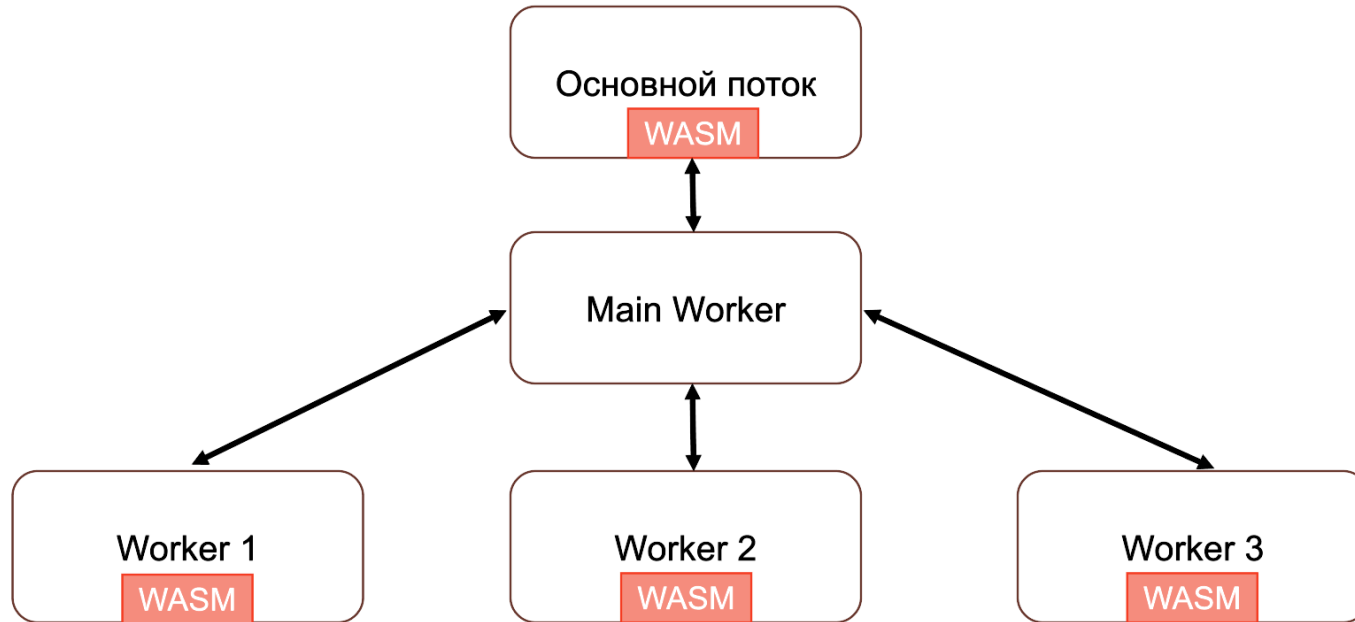
- Механизм, который позволяет скрипту выполняться в фоновом потоке
- Основной поток без блокировки и замедления
- Worker'ы могут запускать другие worker'ы
- Они общаются друг с другом через `postMessage`



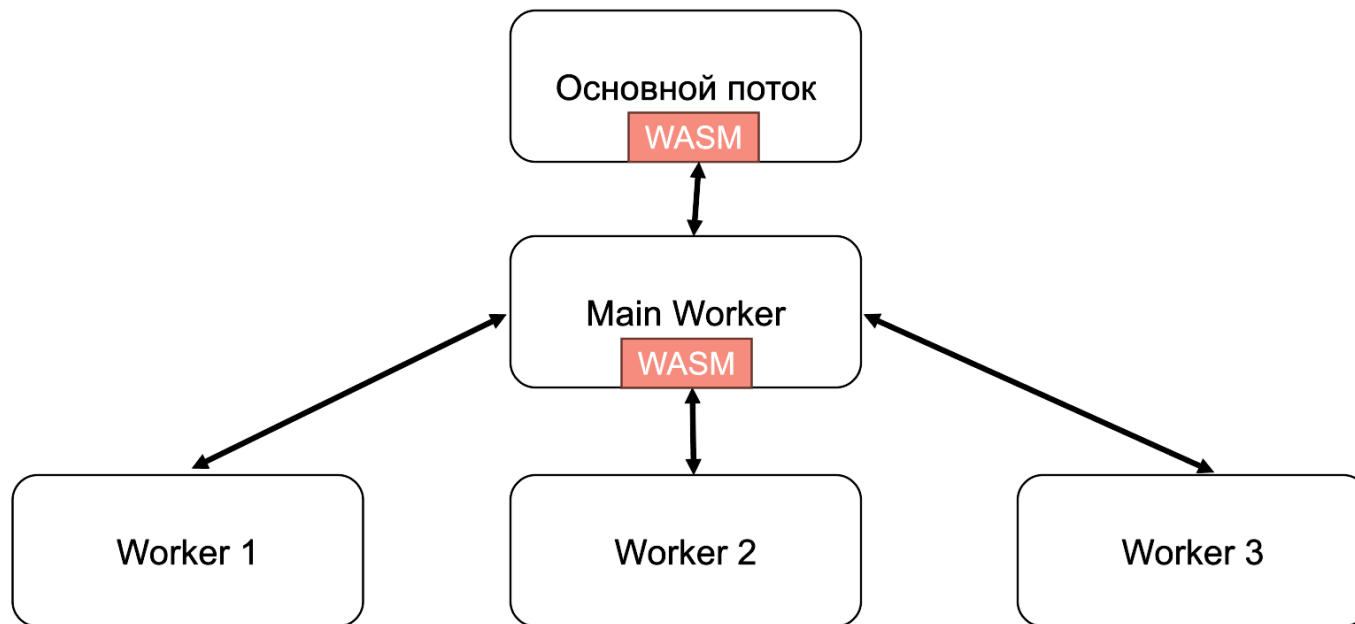
# Web workers в StarLite



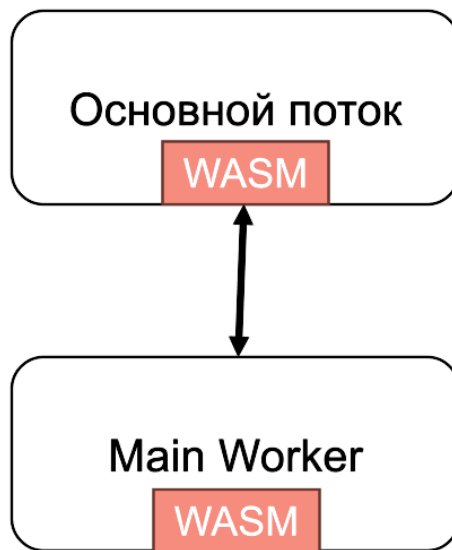
# Web workers & WASM



# Web workers & WASM



# Web workers & WASM



# Как тестировать WASM?

# Тестирование

- Unit-тесты на стороне C++
- Интеграционные тесты QAA

# Jest

```
01. import { readFileSync } from 'fs';  
02.  
03. const wasmBinary = readFileSync(  
04.   'node_modules/path-to-wasm/math.wasm',  
05. );  
06.  
07. instance = await createModule({ wasmBinary });
```

# Результаты



# Плюсы

- Скорость расчётов выросла (в 2 – 10 раз)
- Расчёты на разных платформах совпадают
- Нет дублирования математических расчётов

# Минусы

- Сложность отладки
- Зависимость от другой команды
- Долгая итерация внедрения

# Когда стоит использовать WASM?

- Нужные функции уже написаны на другом языке
- У вас большие объёмы данных, и вам нужен прирост в скорости
- У вас есть ресурсы на внедрение

# Пожалуйста, оставьте СВОЙ ОТЗЫВ

Павел Дадыкин  
ROGII

<https://meloman4eg.github.io/wasm-fc-2024/>

[@meloman4eg](https://github.com/meloman4eg)



**FC**

**Frontend  
Conf 2024**