Netcracker

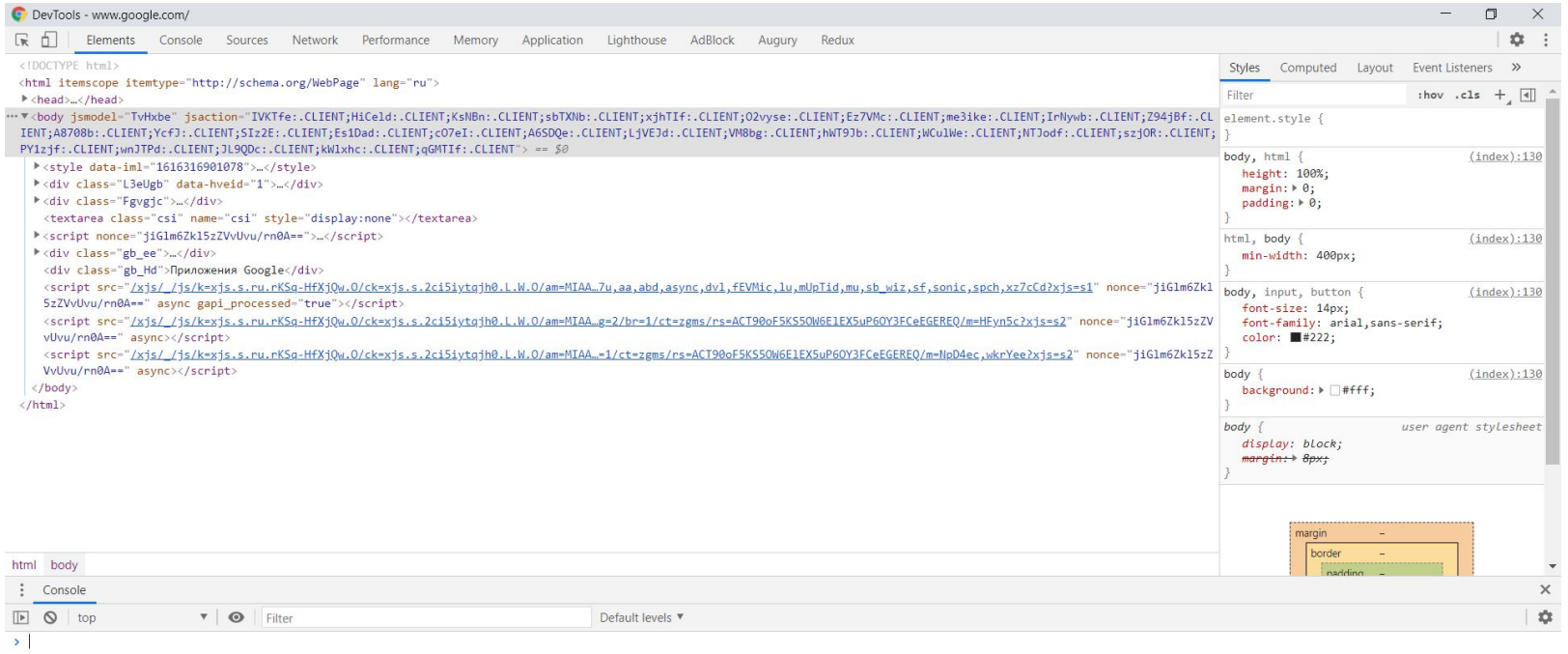
JavaScript

Netcracker

Dev tools

Отладка кода в dev tools

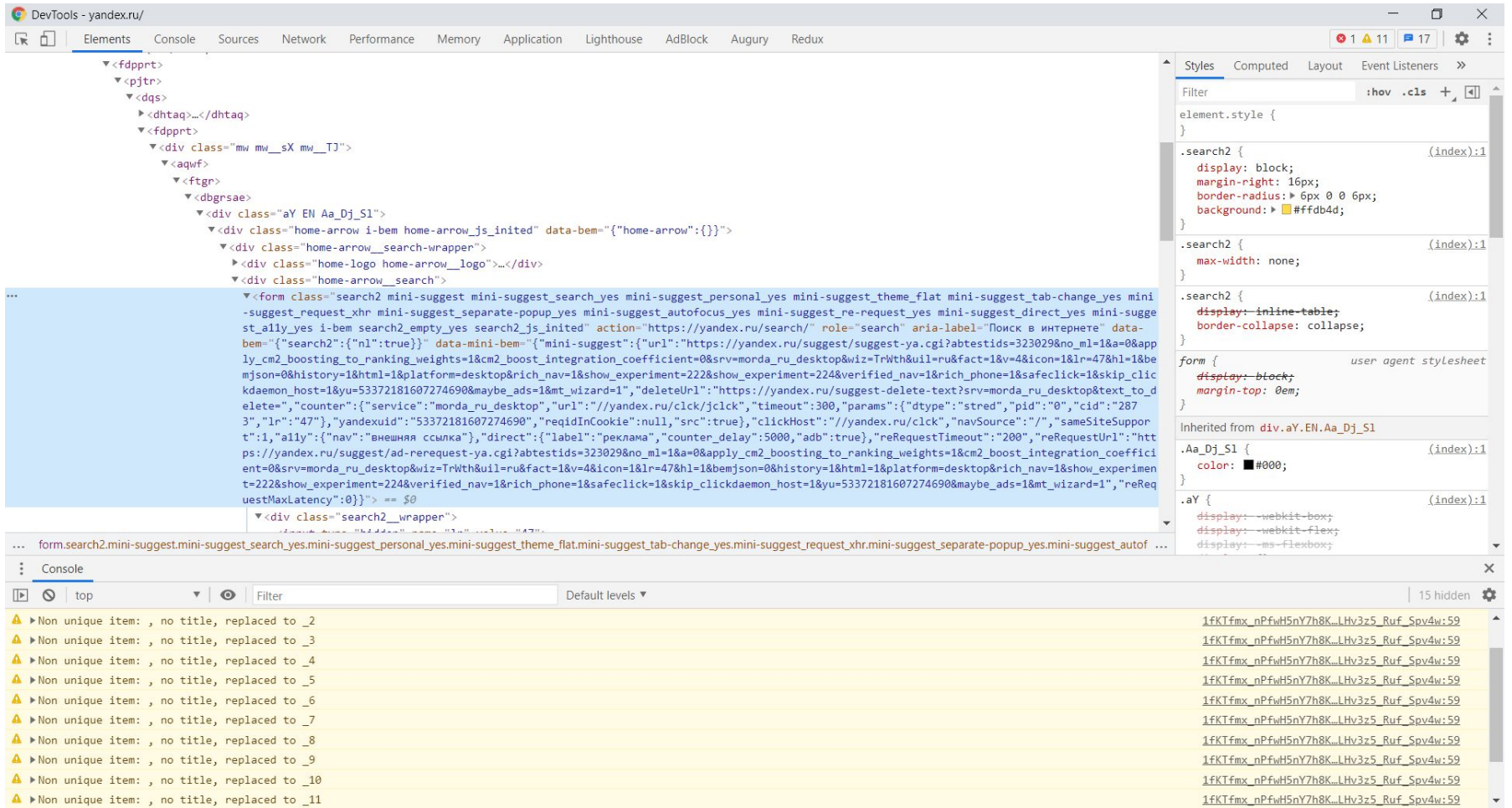
Все современные браузеры поддерживают «инструменты разработчика».
Исправление ошибок с их помощью намного проще и быстрее.



Что можно делать с помощью dev tools?

1. Искать и устранять дефекты вёрстки «на лету»
2. Разбирать иерархию наследования каскадных стилей
3. Отлаживать JS и изменять его в режиме реального времени
4. Динамически проверять тестовые решения прямо в консоли
5. Исследовать загружаемые ресурсы
6. Просматривать отображение приложения в различных разрешениях
7. Анализировать проблемы безопасности
8. Анализировать утечки памяти

Elements



Elements

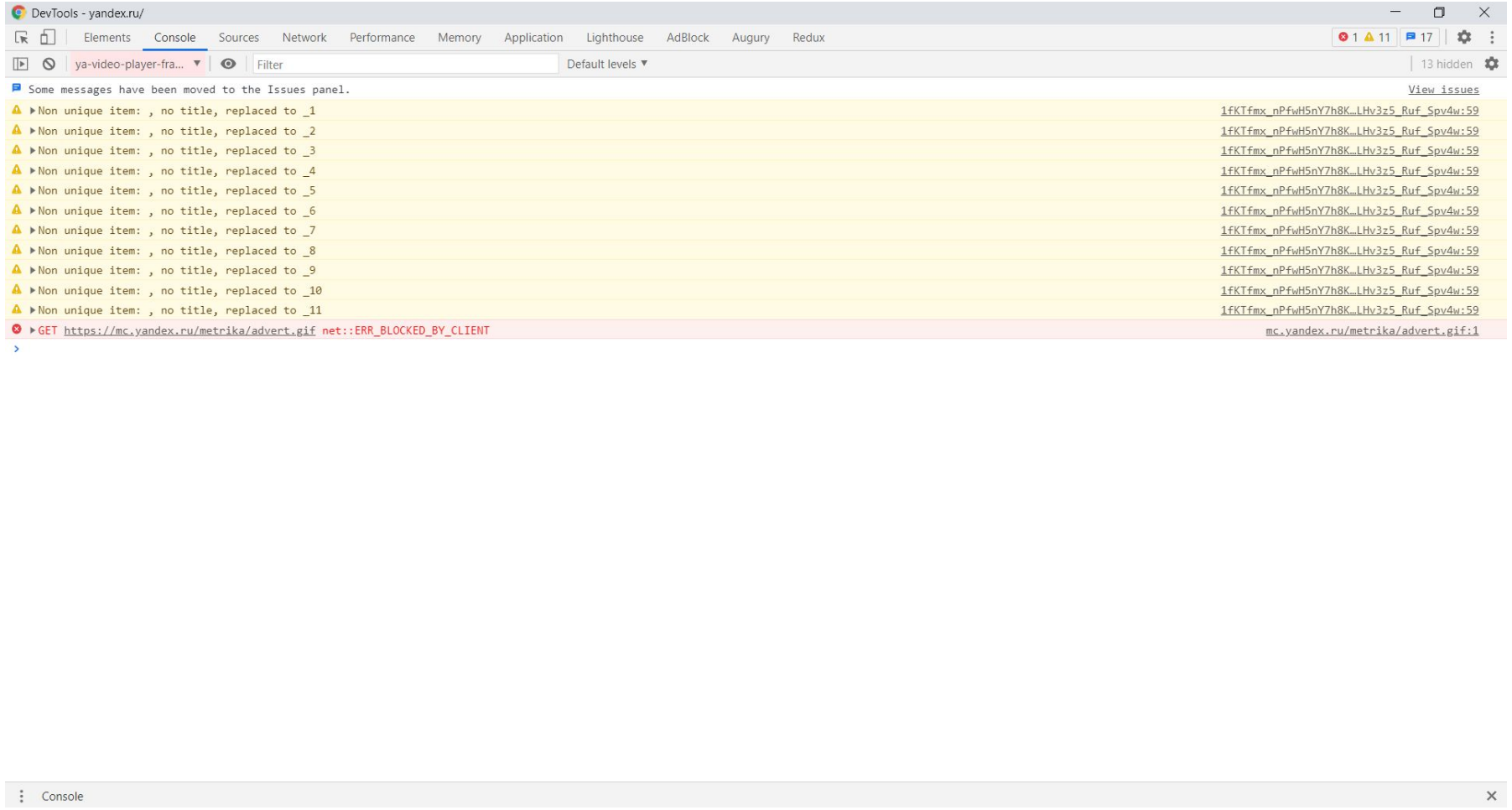
Главный раздел, где находится html разметка документа. Позволяет:

- Инспектировать html
- Находить и исправлять верстку
- Добавлять или удалять атрибуты
- Искать нужные элементы

Справа – вся необходимая информация по CSS и взаимодействию с элементами.

- Добавлять и удалять стили
- Исследовать иерархию наследования CSS
- Искать в каком компоненте был добавлен текущий стиль
- В подтабе Event Listeners можно исследовать зарегистрированные подписки на элемент.

Console



Console

Незаменимый инструмент в отладке кода. Позволяет онлайн выполнять куски кода. Находясь на точке остановки позволяет обращаться к переменным в текущем окружении.

Консоль открывается клавишей **Esc** или переходом в раздел **Console**.

Самая часто используемая команда: **console.log()** позволяет выводить в консоль переменные/текст/объекты и тп.

Есть ещё одна полезная команда **console.dir()**. Она позволяет выводить объект с его структурой, прототипом и тд.

Console

- **console.warn()** – помечает сообщение жёлтым цветом и значком warning
- **console.error()** – помечает сообщение красным цветом и значком ошибки
- **console.group()** – группирует некие связные данные
- **console.table()** – выводит данные в консоль в виде таблицы
- **console.time()** – позволяет замерять время выполнения операций
- **console.trace()** – выводит в консоль результаты трассировки стека и позволяет судить о том, что произошло в определённом месте программы

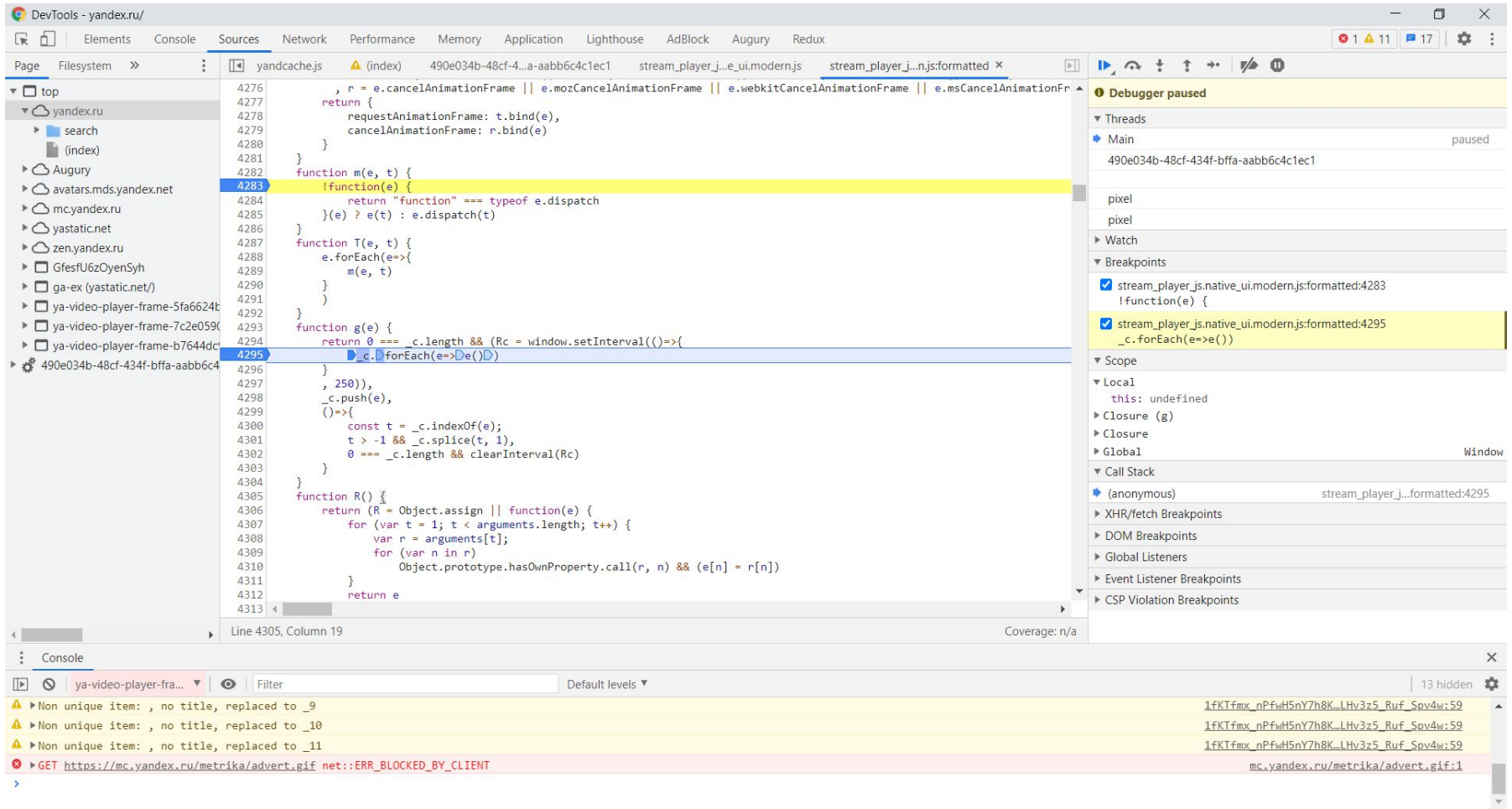
Ошибки

Ошибки при выполнении скрипта выводятся в консоль автоматически.

Добавить в консоль ошибку можно и вручную, с помощью **console.error()**.

Навигироваться на код, где произошла ошибка можно по ссылке, расположенной справа от ошибки.

Sources



Sources

Главный раздел – исходный код позволяет:

- Инспектировать и отлаживать код
- Изменять отдельные куски кода «на лету»

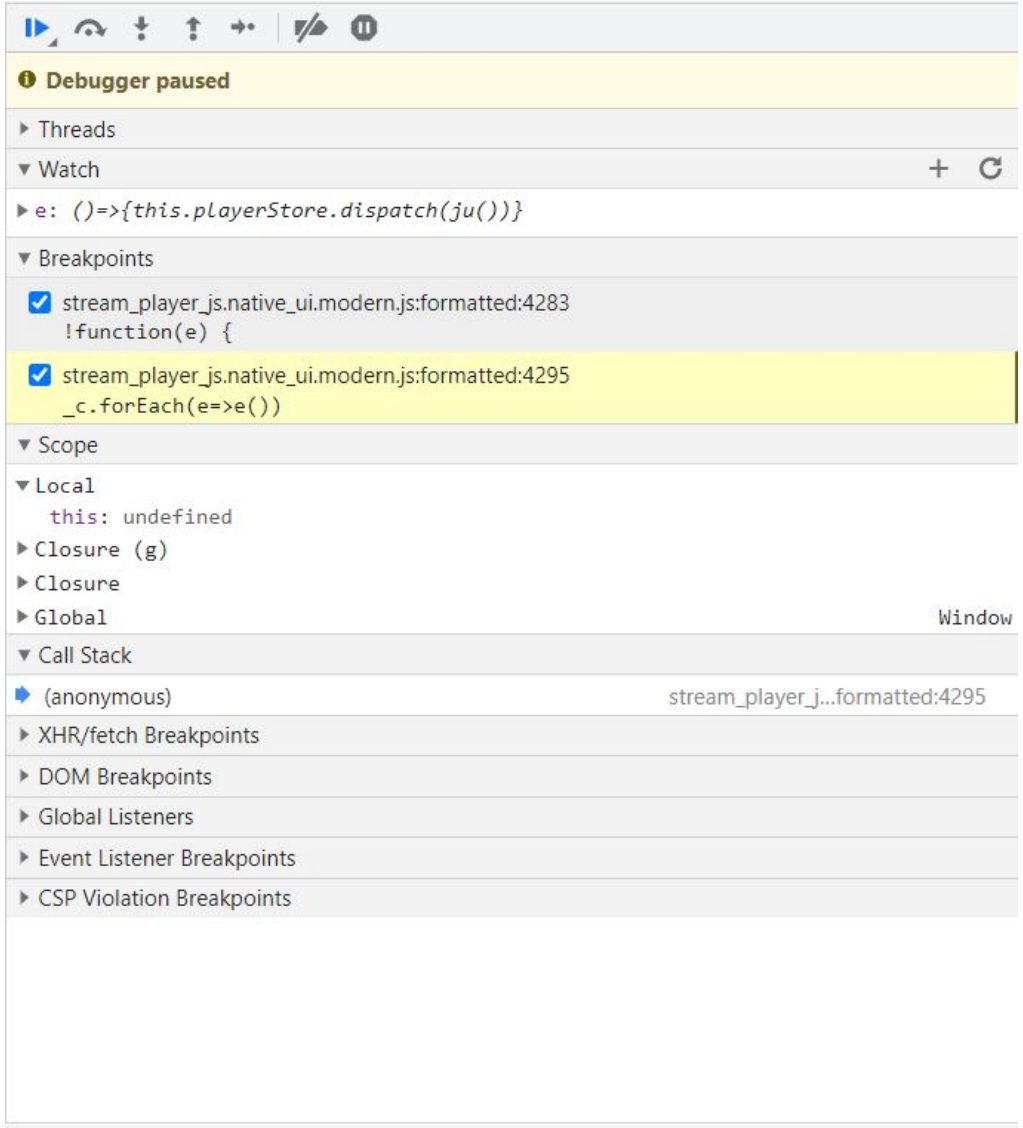
Иерархия исходных файлов (слева, обычно скрыт)

Информационная зона (Включает в себя элементы контроля) позволяет:

- Останавливать/продолжать выполнение приложения
- Контролировать направление, а именно зайти внутрь, пропустить и тп
- Исследовать переменные в текущем окружении
- Исследовать стек вызова методов
- Следить за значениями конкретных переменных
- Следить за точками остановок

Раздел отладки

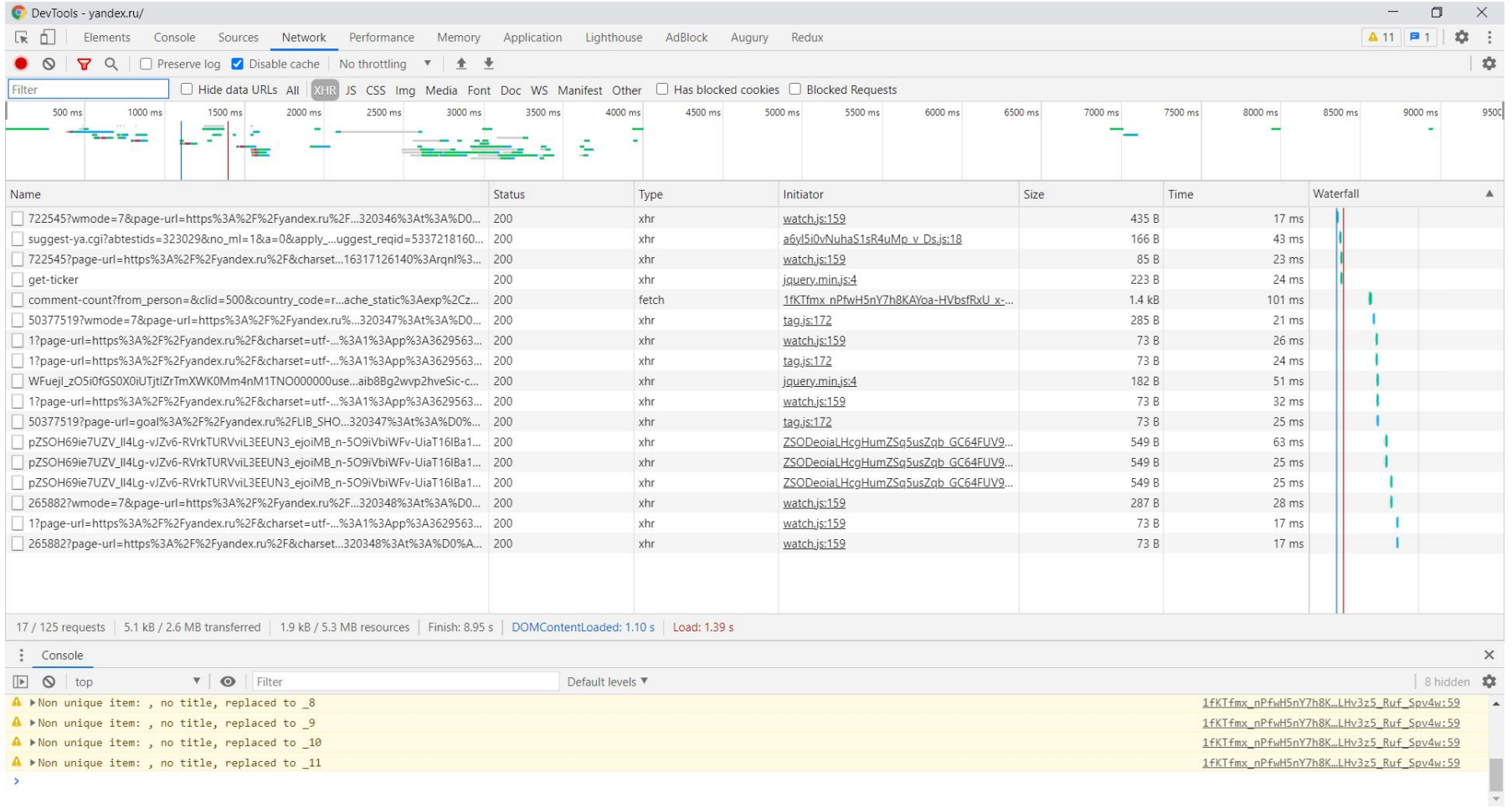
- **Watch** – показывает текущее значение выражений, добавлять выражение нужно вручную
- **Breakpoints** – точки остановок
- **Scope** – переменные окружения
  - Local – локальные переменные текущего окружения, this – ссылка на текущий объект
  - Global – глобальные переменные и функции
- **Call stack** – стек вызовов, все вложенные вызовы, которые привели к текущему месту кода



Управление отладкой

- **Продолжить выполнение (F8)** – продолжение выполнение скрипта, до новых точек остановки
- **Сделать шаг не заходя внутрь функции (F10)** – выполнение команды, находящейся на текущей строке, если в ней находится вызов функции, не переходит в её реализацию
- **Сделать шаг с заходом в функцию (F11)** – аналогично предыдущему, только с заходом в функцию
- **Сделать шаг до выхода из текущей функции (Shift + F11)** – выполняет скрипт до выхода из текущей функции
- **Отключить все точки остановок (ctrl + F8)** – отключает все точки остановок
- **Отключить/включить автоматическую остановку при ошибке** – включает режим остановки на ошибках

Network

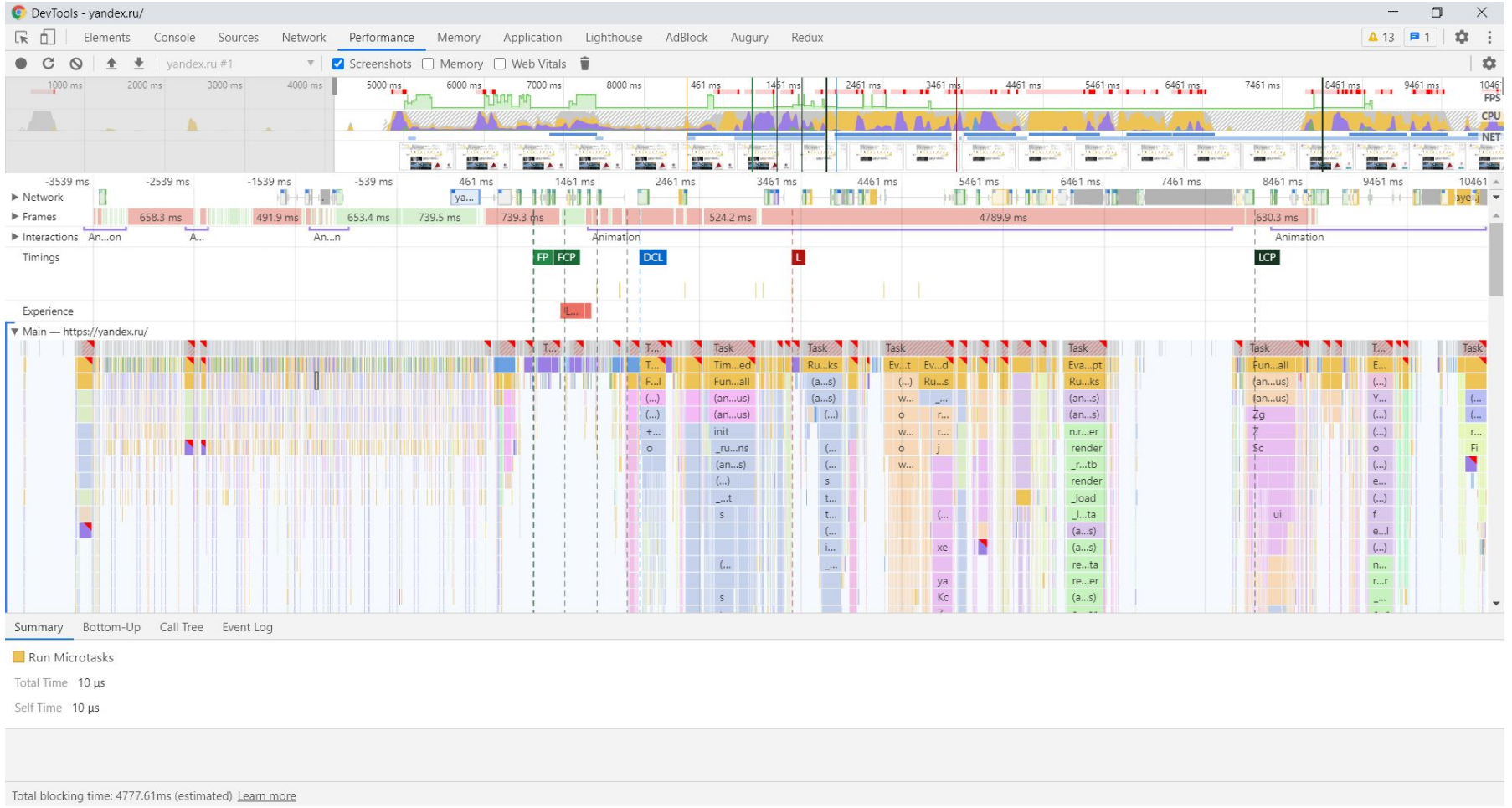


Network

С помощью вкладки Network можно выяснить, сколько времени заняла загрузка страницы, какие ресурсы подключились к странице и многое другое.

Помимо этого можно включить имитацию медленного интернет-соединения (ограничить пропускную способность), отключить кеши или дополнительно отфильтровать ресурсы.

Perfomance



Perfomance

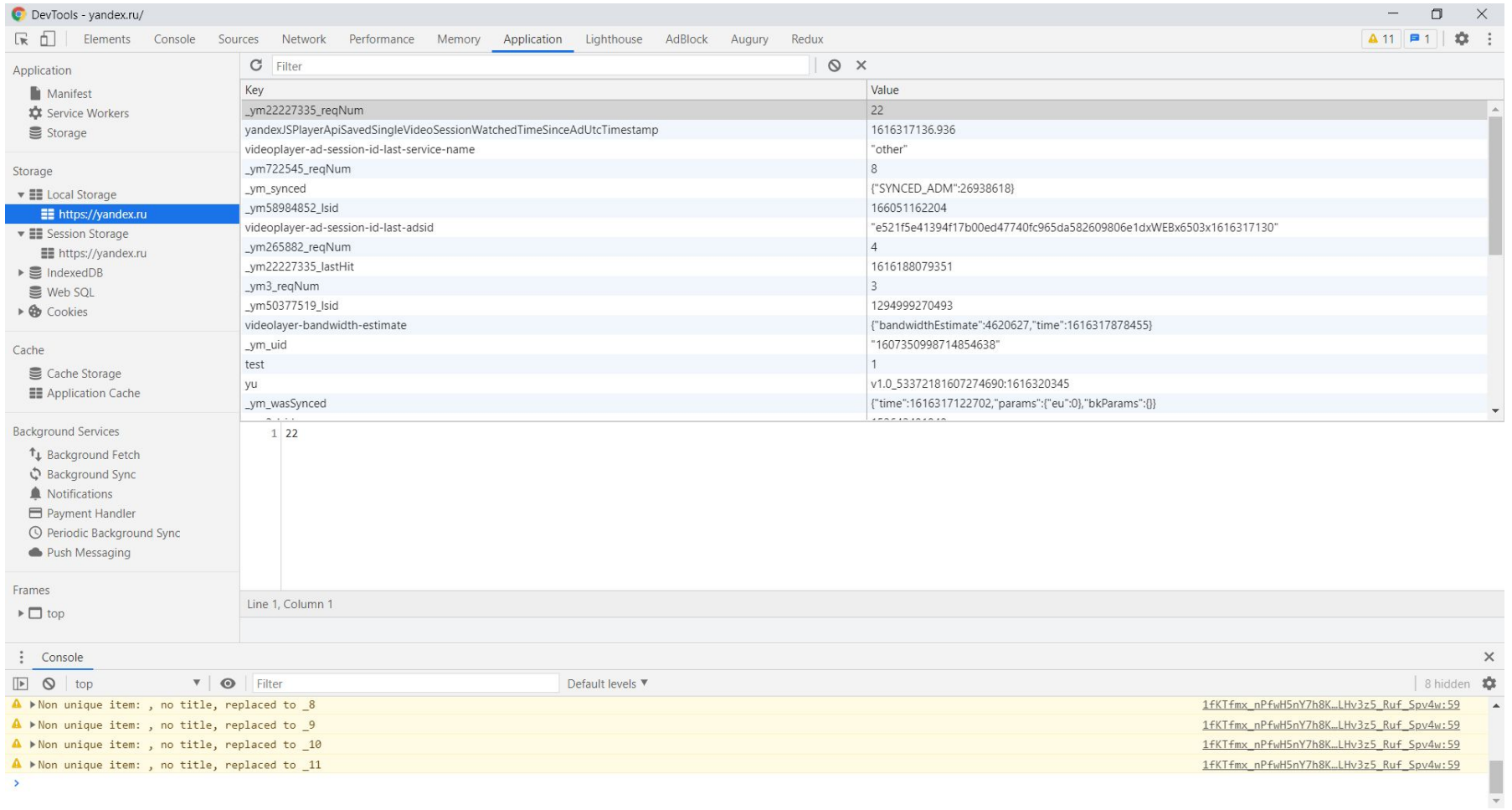
Панель отображает таймлайн использования сети, выполнения JS и загрузки памяти. После первоначального построения будут доступны различные подробные данные о выполнении кода и всем жизненном цикле страницы.

Этот инструмент применяется для улучшения и анализа производительности работы приложения.

Особенности:

- Возможно сделать запись для проведения анализа каждого события
- Возможно просмотреть FPS, загрузку CPU и сетевые запросы.
- Щёлкнув по событию на диаграмме появятся детали
- Можно изменять масштаб таймлайна

Application

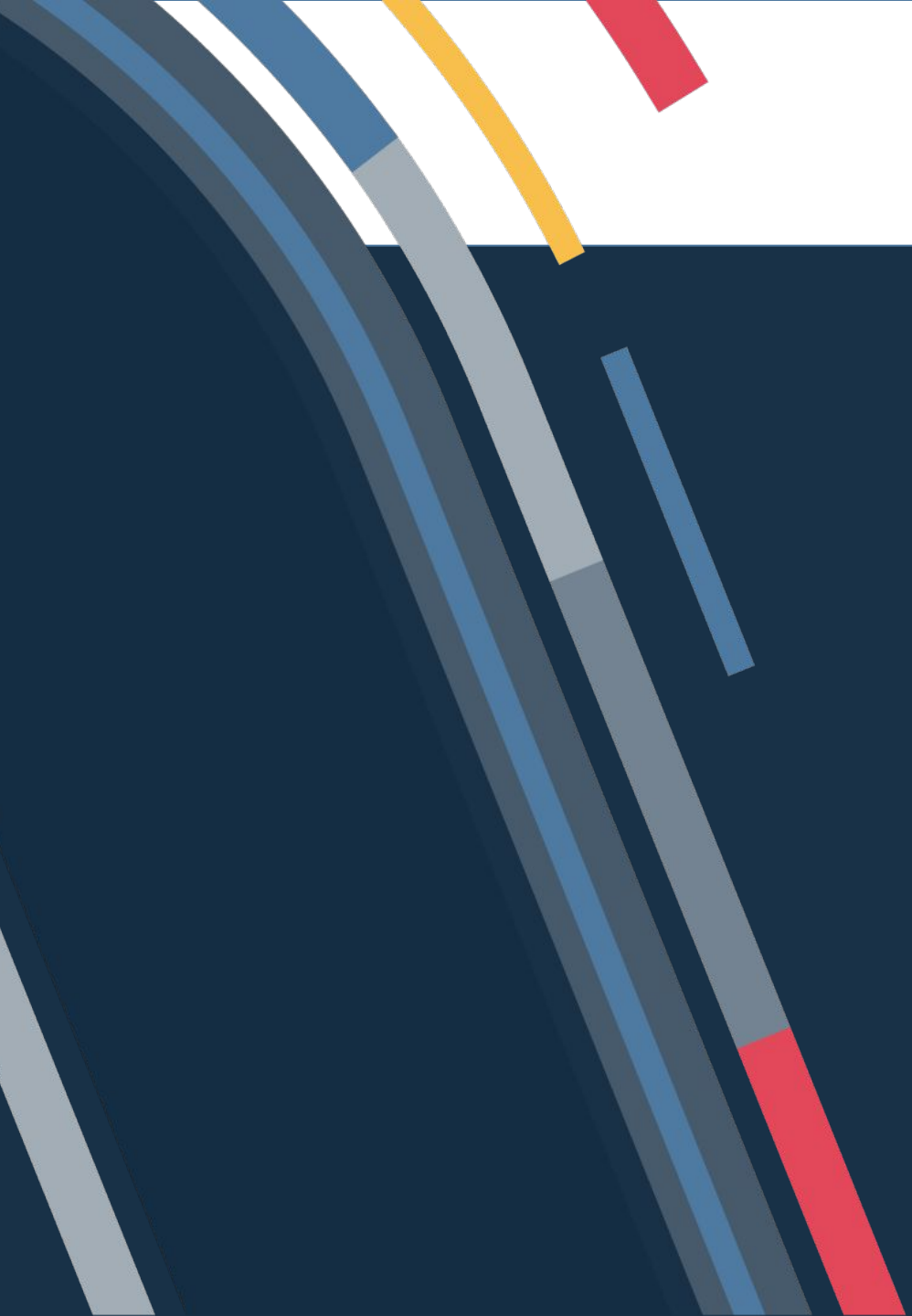


Application

Данная вкладка нужна для инспектирования и очистки всех ресурсов, таких как session storage, local storage, куков, кеша приложения, шрифтов и тд.

Ключевыми возможностями являются:

- Быстрая очистка хранилищ и кеша
- Инспектирование и управление хранилищами
- Инспектирование и удаление файлов cookie



Netcracker

Clean code.
Code style.

Преимущества и признаки чистого кода

- Программист должен легко читать код
- Быстрое понимание кода => быстрое решение задач
  - Добавление новых возможностей
  - Поиск и устранение проблем

Признаки хорошего кода:

- Простой, логичный, понятный
- Чистый и структурированный
- Краткий и лаконичный
- Стилистически единый

Плохой код и откуда он берётся

Признаки плохого кода:

- Непонятный, заумный и сложный
- Запутанный, визуально нагруженный
- Длинный и повторяющийся
- Беспорядочный

Откуда берётся такой код?

- Наследие и сторонний код
- Отсутствие стандартов кодирования
- Различная квалификация сотрудников
- Низкая культура
- Нехватка времени

Правила именования

- Осмысленные имена
  - getSomethingSuperNeededForDrawGraph()
  - getFilteredRows()

- Magic numbers

```
let totalHours =  8*5*4 // плохой пример

const WORKING_HOURS = 8;
const WORKING_DAYS = 5;
const WEEK_IN_MONTH = 4;
let totalHours = WORKING_HOURS * WORKING_DAYS * WEEK_IN_MONTH
```

Правила именования

- Одно слово для каждой концепции
  - fetch/get/retrieve
  - read/load
  - write/upload

- Имена классов и объектов

Должны представлять собой существительные и их комбинации.

  - Customer
  - Account

- Имена методов

Представляют собой глаголы или глагольные словосочетания.

  - getName/setName

Правила именования

- Именование переменных и членов класса

Плохие примеры:
- flag
- _array
- newArray

Хорошие примеры:
- isBlocked
- sortedArray
- currentPosition

Итоги и советы по именованию

Хорошие имена позволяют быстрее ориентироваться в коде и понимать что происходит вокруг.

Хорошее имя:

- Передает назначение
- Имеет разумную длину
- Легко читается
- Быстро вспоминается и ищется
- Не дизинформирует

Функции

- **Длина** – должна быть оптимальной (зачастую это та, которая позволяет разместить функцию на одном экране без пролистывания. Примерно 25-50 строк)
- **Оптимальное количество аргументов** – аргументов не должно быть слишком много (0 – отлично, 1-2 – хорошо, 3 и более уже стоит задуматься)
- **Функциональность** – функция должна выполнять одну операцию. Ничего лишнего.

Комментарии

Комментарии это плохо. Почему? Потому что:

- Они могут быть неактуальными.
- Могут быть избыточными
- Могут запутывать и дезинформировать

Самое плохое в комментариях это закомментированный код!

Комментарии

Исключениями могут быть:

- Комментарии, которые поясняют зачем что-то делается.
- Пояснения в нетривиальных случаях.
- Документация

Рефакторинг

Это изменение внутренней структуры программы без изменений ее видимого поведения с целью облегчить понимание и удешевить модификацию.
М. Фаулер

Цели рефакторинга:

- Достичь лучшего понимания и читаемости кода
- Упростить добавление нового кода
- Улучшить дизайн существующего кода

Процесс рефакторинга:

- Удаление дублирования
- Упрощение сложной логики
- Прояснение непонятного кода

Ключевые моменты

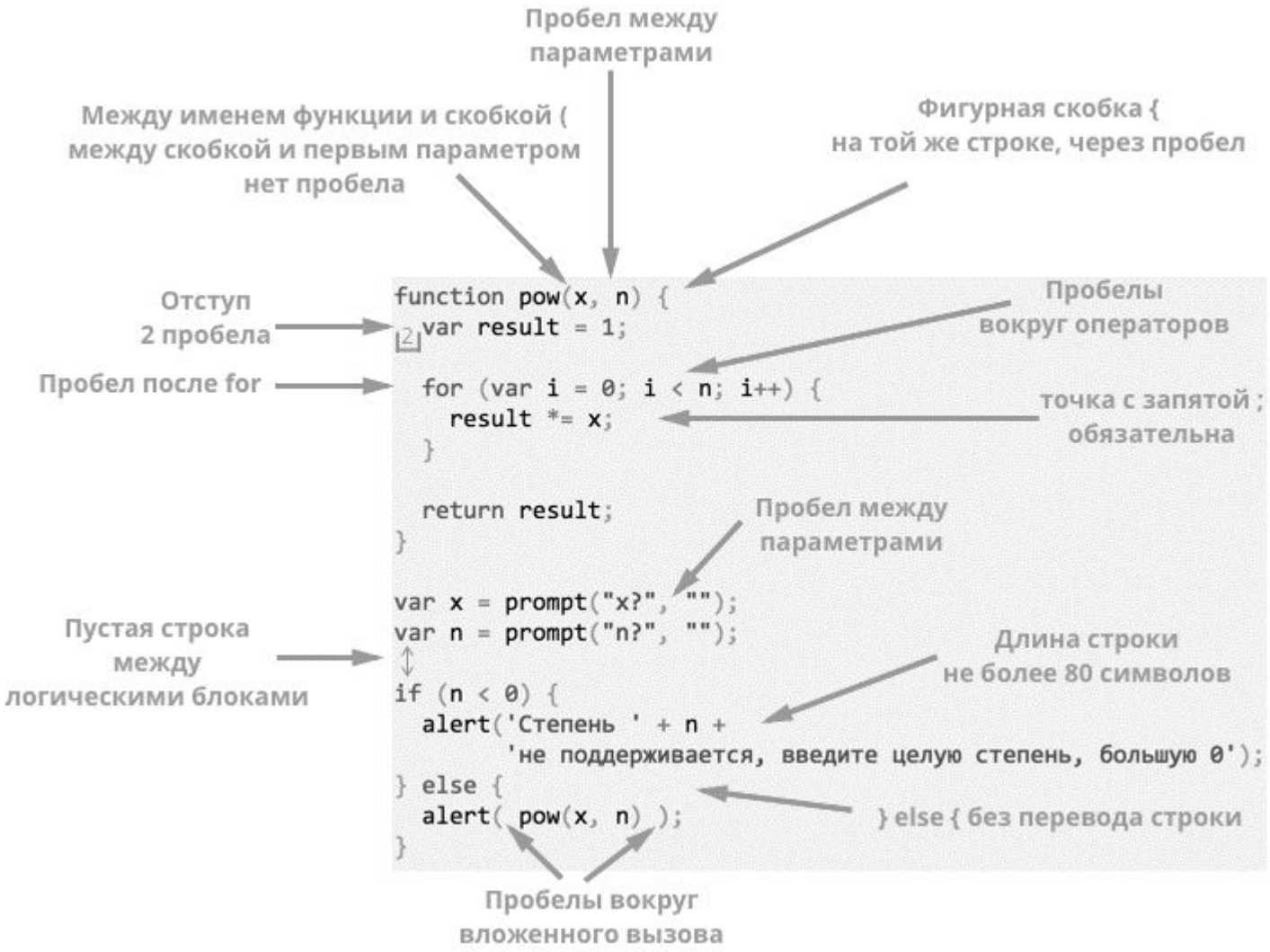
- Некачественный код – смерть проекта
- Качество кода – ответственность разработчика
- Чистка кода – непрерывная активность

Code style

```
if (n < 0) {
  alert('Степень ' + n + ' не поддерживается');
}
```

```
show("Строки" +
     " выровнены" +
     " строго" +
     " одна под другой");
```

```
function pow(x, n) {
  var result = 1;
  //             <--
  for (var i = 0; i < n; i++) {
    result *= x;
  }
  //             <--
  return result;
}
```



Автопроверка code style

Самые известные автоматические средства для проверки стиля кода:

- **JSLint** – средство проверки синтаксиса JS
- **JSHint** – форк от JSLint, управляемый сообществом (более конфигурируемый)
- **TSLint** – средство проверки и форматирования для typescript
- **ESLint** – средство проверки и форматирования синтаксиса JS

ESLint

19 февраля 2019 года вышел официальный пост о прекращении разработки **TSLint**. В нём компания-разработчик советует переходить на **ESLint**.

ESLint обладает гораздо большим количеством описанных правил, покрывает всё, что может быть во фронте. В нём есть правила, которые позволяют проверять код на уровне блоков и выявлять дублирование кода, сложность восприятия и тп. Есть множество плагинов, которые, например, проверяют регулярные выражения.

Netcracker

Перехват и обработка ошибок

Перехват ошибок. Try catch

Это способы «правильно» обрабатывать исключительные ситуации.
Правильность заключается в обёртке кода, где предположительно может быть ошибка, специальной конструкцией **try…catch.**

```
try {
    // потенциально опасный код
} catch (error) { // обработка ошибки }
```

Принцип работы:

1. Выполняется код внутри блока **try**
2. Если ошибок не произошло, блок **catch** игнорируется
3. Если произошла ошибка – выполняется блок **catch**

Перехват ошибок

Цель такой конструкции заключается в необходимости оградить скрипт от возможных ошибок. Другими словами, при возникновении ошибки выполнение скрипта не должно прерываться.

Ошибки могут возникать как автоматически:

- При неправильном использовании переменных
- При обращении к несуществующему полю
- При использовании несуществующих аргументов и тд.

Так могут и создаваться вручную, с помощью ключевого слова **throw**.

Перехват ошибок. Throw

```
Try {
    let err = new Error(“can’t create”);
    err.message = “Can’t resolve”;
    err.name = “ERROR”;
    throw err;
} catch (error) {
    console.dir(error);
}
```

Аргументом error метода **catch** будет являться то, что было «проброшено» после **throw**. В данном случае это будет объект Error.

Перехват ошибок. Объект Error

В случае возникновения ошибки JS генерирует объект, содержащий детали этой ошибки. Для всех встроенных ошибок такой объект имеет несколько основных свойств:

- **name** – имя ошибки
- **message** – сообщение о деталях ошибки
- **stack** – текущий стек вызова. Строка, содержащая информацию о последовательности вложенных вызовов, которые привели к ошибке

Перехват ошибок. Условный блок catch

Условным блоком **catch** называется структура **try…catch** с использованием **if…else**, например:

```
try {
  // код
} catch (error) {
  if (error instanceof TypeError) {
     // обработка исключения TypeError
  } else {
     // обработка остальных исклюений
  }
```

Перехват ошибок. Секция finally

Конструкция **try…catch** может содержать ещё одну секцию – **finally**. Эта секция не обязательна, но если она есть, то она выполняется всегда. Не зависимо от того была ошибка или нет. Секция **finally** используется для того, чтобы завершить начатые операции при любом варианте развития событий.

```
try {
  // пробуем выполнить код
} catch (error) {
  // перехватываем исключение
} finally {
  // выполняем всегда
}
```

Перехват ошибок. Секция finally

```
function func() {
 try {
    return 1;
 } catch (error) {
    // перехватываем исключение
 } finally {
    console.log(“finally”)
 }
}
console.log(func())
```

В данном примере присутствует return, но finally отработает раньше.

Перехват ошибок. Window.onerror

В случае, когда ошибка произошла вне блока **try…catch** или выпала наружу, во внешний код – скрипт упадёт. Чтобы этого избежать можно воспользоваться специальным свойством **window.onerror**. Если в него записать функцию, то она выполнится и получит в аргументах сообщение ошибки, текущий URL и номер строки, откуда выпала ошибка.

```
window.onerror = (message, url, lineNumber) => {
  console.log(“message”)
}
```

Q&A

Thank You


Netcracker