

# Веб- программирование: Браузерные API

# Содержание

---

01 вом

02 Хранение

03 Скрипты

04 Performance API

05 Графика

06 Отслеживание изменений

07 Системное взаимодействие

08 Медиа API

# Определение

Браузерные API (Application Programming Interface) — это набор готовых программных интерфейсов, встроенных в веб-браузер, которые позволяют разработчикам взаимодействовать с различными функциями и возможностями браузера через JavaScript.



<https://developer.mozilla.org/en-US/docs/Web/API>

# ВОМ

# ВОМ

ВОМ (Browser Object Model) — это набор объектов, предоставляемых браузером, которые позволяют взаимодействовать с окном браузера и его элементами. В отличие от DOM, который работает с элементами страницы, ВОМ фокусируется на самом браузере.

# Основные объекты

- Window — главный объект модели
- Location — управление URL
- History — управление историей браузера
- Navigator — информация о браузере
- Screen — информация об экране
- Document — связь с DOM

# Window: свойства

- innerWidth и innerHeight — размеры окна
- outerWidth и outerHeight — полный размер окна
- screenX и screenY — координаты окна
- closed — проверка закрытия окна

# Window: методы

- `alert()`, `confirm()`, `prompt()` — диалоговые окна
- `setTimeout()`, `clearTimeout()` — таймеры
- `setInterval()`, `clearInterval()` — интервалы
- `open()`, `close()` — управление окнами

# Location: свойства

- href — полный URL
- protocol — протокол
- host — хост
- pathname — путь
- search — параметры запроса
- hash — якорь

# Location: методы

- assign() — загрузка нового URL
- reload() — перезагрузка страницы
- replace() — замена текущей страницы

# History: методы

- back() — переход назад
- forward() — переход вперед
- go(n) — переход на n страниц

# **Navigator: свойства**

- userAgent — информация о клиенте
- appVersion — версия браузера
- platform — платформа
- language — язык

# Screen: свойства

- width и height — размеры экрана
- availWidth и availHeight — доступные размеры
- colorDepth — глубина цвета
- pixelDepth — глубина пикселя

# Дока



<https://doka.guide/js/bom/>

# Хранение: Storage и IndexedDB

# Local Storage

Local Storage — это механизм хранения данных в браузере пользователя, который позволяет сохранять информацию на длительный срок. Данные хранятся локально и не отправляются на сервер.

- Постоянное хранение данных (не удаляется после закрытия браузера)
- Объем хранилища: около 5-10 МБ на домен
- Формат хранения: пары ключ-значение
- Тип данных: только строки (другие типы автоматически преобразуются)
- Same-origin policy: данные доступны только для своего домена

# Основные методы

- `setItem(key, value)` — сохранение данных
- `getItem(key)` — получение данных
- `removeItem(key)` — удаление данных
- `clear()` — очистка всего хранилища
- `length` — получение количества элементов
- `key(index)` — получение ключа по индексу

# Пример с простыми данными

```
// Сохранение строки
localStorage.setItem('username', 'Anya');

// Получение строки
const user = localStorage.getItem('username');
console.log(user) // Anya

// Сохранение числа
localStorage.setItem('age', 31);

// удаление
localStorage.removeItem('age')
```

# Пример с объектом

```
const user = {  
    name: 'Anyा',  
    age: 31  
};  
  
// Сериализация объекта в JSON  
localStorage.setItem('user', JSON.stringify(user));  
  
// Десериализация из JSON  
const retrievedUser =  
JSON.parse(localStorage.getItem('user'));
```

# Дока



<https://doka.guide/js/local-storage/>

# Session Storage

Session Storage — это механизм временного хранения данных в браузере, который автоматически удаляет информацию после закрытия вкладки или окна браузера.

- Временное хранение данных только на период активной сессии
- Объем хранилища: около 5-10 МБ на домен
- Формат хранения: пары ключ-значение
- Тип данных: только строки
- Сохранение при перезагрузке: данные остаются после обновления страницы
- Изоляция сессий: каждая вкладка имеет собственное хранилище

# Основные методы

- `setItem(key, value)` — сохранение данных
- `getItem(key)` — получение данных
- `removeItem(key)` — удаление данных
- `clear()` — очистка всего хранилища
- `length` — получение количества элементов
- `key(index)` — получение ключа по индексу

# Пример с простыми данными

```
// Сохранение данных
sessionStorage.setItem('cartItem', 'product1');

// Получение данных
const item =
sessionStorage.getItem('cartItem');

// Удаление данных
sessionStorage.removeItem('cartItem');
```

# Дока



<https://doka.guide/js/session-storage/>

# Сравнение Session и Local Storage

Характеристика	Session Storage	Local Storage
Срок хранения	Пока вкладка открыта	Не ограничен
Объём	5 - 10 МБ	5 - 10 МБ
Тип данных	Строки	Строки
Уровень изоляции	Сессия	Домен

# IndexedDB

IndexedDB — это встроенная в браузер объектно-ориентированная база данных, позволяющая хранить большие объемы структурированных данных непосредственно на стороне клиента.

- Оффлайн-работа: работает без подключения к сети
- Высокая производительность: быстрый доступ к данным
- Масштабируемость: поддержка больших объемов информации
- Безопасность: данные изолированы по доменам

# Пример

```
// открываем соединение с базой
// 1 - это версия
const request = window.indexedDB.open("MyTestDatabase", 1);

request.onerror = (event) => {
    // как-то обрабатываем ошибку
};

request.onsuccess = (event) => {
    // как-то обрабатываем успех
};
```

## Пример

```
request.onupgradeneeded = (event) => {
    const db = event.target.result;
    const objectStore = db.createObjectStore(
        "customers",
        { keyPath: "id" },
    );
    objectStore.createIndex("name", "name", { unique: false });

    objectStore.transaction.oncomplete = () => {
        const customerObjectStore = db
            .transaction("customers", "readwrite")
            .objectStore("customers");
        customerObjectStore.add({ id: "111", name: "Bill", age: 35 });
    };
};
```

# Дока



<https://doka.guide/tools/browsers-storages/#indexeddb>

# Скрипты: WebWorkers и ServiceWorkers

# WebWorkers

Web Workers — это механизм, позволяющий выполнять JavaScript-код в отдельном потоке параллельно с основным потоком браузера.

- Отдельный поток: код выполняется в изолированном окружении
- Параллельное выполнение: не блокирует основной поток и рендеринг страницы
- Обмен сообщениями: взаимодействие через механизм `postMessage`
- Изоляция памяти: основной поток и воркер не делят память

# Ограничения

- Нет доступа к DOM
- Нет доступа к window
- Нет прямого доступа к localStorage
- Работает только с JavaScript-файлами

# Применение

- Выполнение тяжелых вычислений
- Обработка больших данных
- Криптографические операции
- Парсинг больших JSON
- Работа с файлами

# Пример: создание

```
// index.js
const worker = new Worker('worker.js');

worker.postMessage({ data: 'some data' });

worker.onmessage = (event) => {
    console.log('Ответ от воркера:', event.data);
};

worker.terminate();
```

# Пример: использование

```
// worker.js
self.onmessage = (event) => {
    const data = event.data;

    const result = heavyComputation(data);

    self.postMessage(result);
};
```

# Дока



<https://doka.guide/js/web-workers/>

# ServiceWorkers

Service Workers — это специальный тип веб-воркеров, работающий в отдельном потоке браузера, который позволяет:

- Обеспечивать работу приложения в офлайн-режиме
- Управлять кэшированием ресурсов
- Отправлять push-уведомления
- Синхронизировать данные в фоновом режиме
- Перехватывать сетевые запросы

# Особенности

- Работают в отдельном потоке
- Не имеют доступа к DOM
- Не могут использовать XHR или файлы cookie
- Должны обслуживаться через HTTPS

# Основные события

- install — вызывается при установке
- activate — срабатывает при активации
- fetch — происходит при запросе ресурса
- push — при получении push-уведомления
- sync — при восстановлении соединения

# Пример: регистрация сервис-воркера

```
window.addEventListener('load', () => {
  if ('serviceWorker' in navigator) {
    navigator.serviceWorker.register('/sw.js')
      .then(() => {
        console.log('Service Worker зарегистрирован');
      })
      .catch(err => {
        console.error('Ошибка регистрации:', err);
      });
  }
});
```

# Пример: работа с кешем

```
// sw.js
self.addEventListener('install', event => {
  event.waitUntil(
    caches.open('v1')
      .then(cache => cache.addAll(['/style/.css']))
  );
});

self.addEventListener('fetch', event => {
  event.respondWith(
    caches.match(event.request)
      .then(response => response || fetch(event.request));
  );
});
```

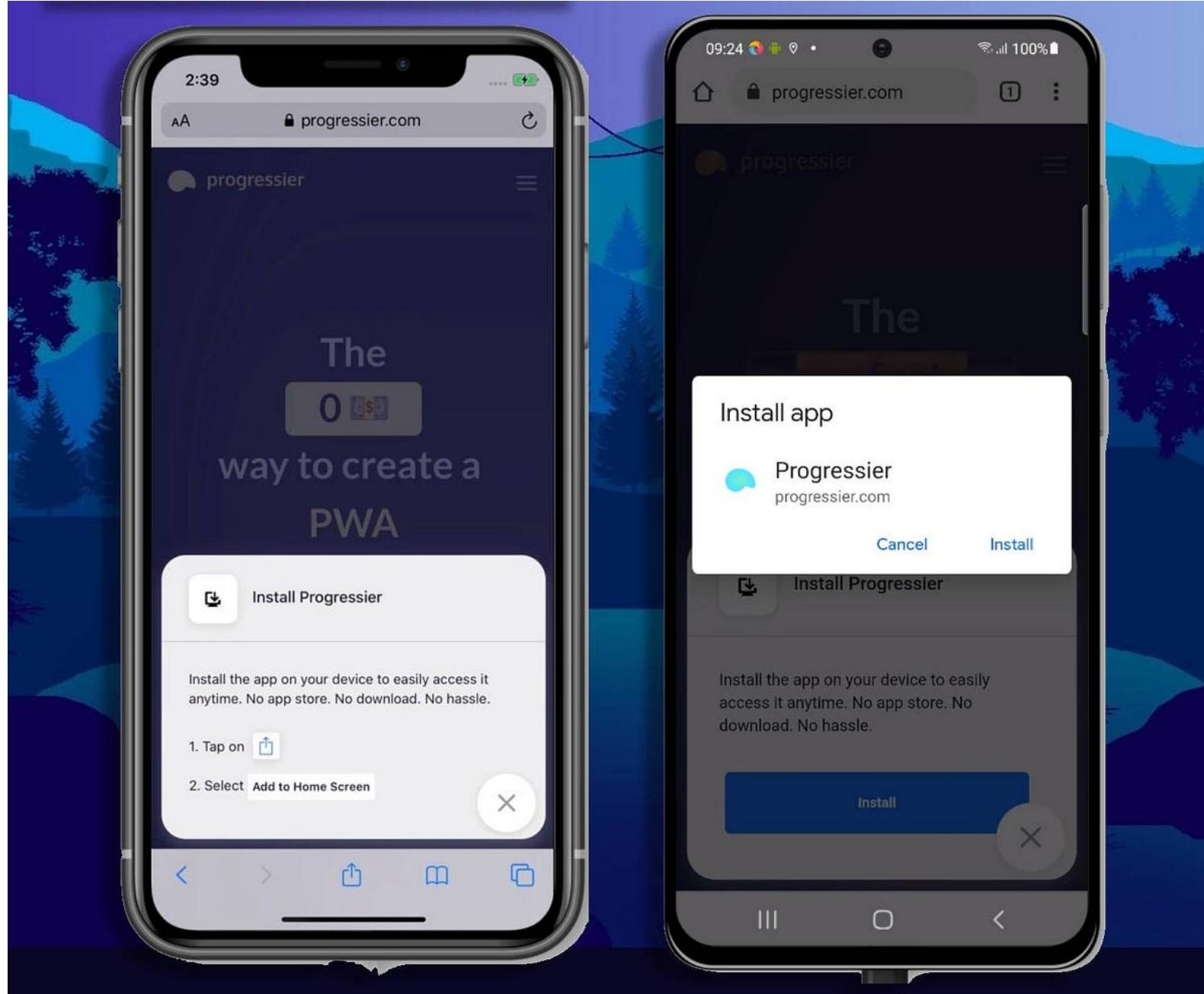
# MDN



[https://developer.mozilla.org/en-US/docs/Web/API/Service\\_Worker\\_API](https://developer.mozilla.org/en-US/docs/Web/API/Service_Worker_API)

# PWA

PWA — это прогрессивное веб-приложение, которое объединяет лучшие черты веб-сайтов и мобильных приложений. Оно доступно на любом устройстве через браузер и может устанавливаться на главный экран.



# Особенности

- Прогрессивность: адаптация под возможности устройства
- Доступность: работа на любых современных устройствах
- Безопасность: обязательное использование HTTPS
- Оффлайн-доступ: возможность работы без интернета
- Быстрая загрузка: оптимизированная производительность

# Преимущества

- Универсальность: работает на всех устройствах
- Экономия: не требует разработки отдельных версий для платформ
- Автообновление: приложение обновляется автоматически
- Оффлайн-режим: доступ к контенту без интернета
- Уведомления: push-сообщения даже при закрытом приложении
- Безопасность: защита данных через HTTPS
- Простая установка: без магазинов приложений

# Ограничения

- Поддержка браузеров: не все браузеры одинаково поддерживают
- Доступ к железу: ограниченный доступ к функциям устройства
- Конфиденциальность: некоторые операции ограничены

# Ключевые компоненты

- Service Workers — скрипты для управления кэшированием
- Web App Manifest — файл конфигурации приложения
- Push Notifications — система уведомлений

# Doka



<https://doka.guide/tools/pwa/>

# Performance API

# Performance API

Performance API — это набор инструментов браузера для точного измерения времени работы программы и функций. API предоставляет высокоточную метрику времени с разрешением до 5 микросекунд.

Хранит данные измерений в массиве Performance Timeline



<https://doka.guide/js/performance/>

# Основные возможности

- Точное измерение времени выполнения кода
- Анализ производительности веб-приложений
- Мониторинг загрузки ресурсов
- Отслеживание рендеринга страницы
- Профилирование JavaScript-кода

# Типы меток

- Mark — именованная метка времени
- Measure — измерение промежутка между метками
- Navigation — события навигации
- Resource — загрузка ресурсов
- Paint — время отрисовки
- LongTask — длительные задачи

# Основные методы

- `mark()` - создаёт именованную метку
- `measure()` - возвращает разницу во времени между двумя метками
- `now()` - возвращает таймстамп
- `clearMarks()` - удаляет именованные метки
- `clearMeasures()` - удаляет измерения
- `getEntries()` - возвращает все записи из Performance Timeline
- `getEntriesByName()` - возвращает все записи из Performance Timeline с заданным именем
- `getEntriesByType()` - возвращает все записи из Performance Timeline с заданным типом

# Пример

```
// Получение текущего времени
const start = performance.now();

// Создание именованной метки
performance.mark('start');
performance.mark('end');

// Измерение времени между метками
performance.measure('executionTime', 'start', 'end');
```

# Performance Observer

Performance Observer — это API, которое позволяет асинхронно отслеживать различные события производительности веб-приложения в реальном времени и реагировать на них.



<https://developer.mozilla.org/en-US/docs/Web/API/PerformanceObserver>

# Пример

```
function perfObserver(list, observer) {
  list.getEntries().forEach((entry) => {
    if (entry.entryType === "mark") {
      console.log(`#${entry.name}'s startTime: ${entry.startTime}`);
    }
    if (entry.entryType === "measure") {
      console.log(`#${entry.name}'s duration: ${entry.duration}`);
    }
  });
}
const observer = new PerformanceObserver(perfObserver);
observer.observe({ entryTypes: ["measure", "mark"] });
```

# Графика: Canvas и SVG

# Canvas

Canvas — это HTML-элемент, который предоставляет возможность создавать и манипулировать растровой графикой с помощью JavaScript. Он позволяет рисовать фигуры, линии, текст и изображения прямо в браузере.

```
<canvas id="myCanvas" width="300" height="200"></canvas>
```

# Canvas Context

Canvas Context — это интерфейс, предоставляющий методы и свойства для рисования и манипуляции графикой на HTML Canvas. Он является мостом между HTML-элементом canvas и JavaScript-кодом.

Типы:

- 2D-контекст
- WebGL-контекст — для работы с 3D-графикой

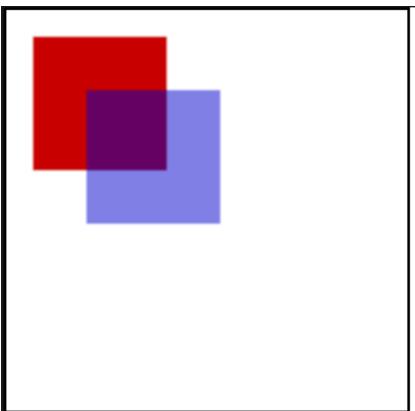
```
const canvas = document.getElementById('myCanvas');
const ctx = canvas.getContext('2d');
```

# Пример

```
const canvas = document.getElementById("canvas");
const ctx = canvas.getContext("2d");

ctx.fillStyle = "rgb(200 0 0)";
ctx.fillRect(10, 10, 50, 50);

ctx.fillStyle = "rgb(0 0 200 / 50%)";
ctx.fillRect(30, 30, 50, 50);
```



# Основные методы: пути и фигуры

- `beginPath()` — начинает новый путь рисования
- `closePath()` — закрывает текущий путь
- `moveTo(x, y)` — перемещает перо в указанную точку
- `lineTo(x, y)` — рисует линию к указанной точке
- `quadraticCurveTo()` — рисует квадратичную кривую Безье
- `bezierCurveTo()` — рисует кубическую кривую Безье
- `arc(x, y, radius, startAngle, endAngle)` — рисует дугу
- `arcTo()` — рисует дугу между двумя линиями
- `rect(x, y, width, height)` — создает прямоугольник

# Основные методы: стили и цвета

- `fillStyle` — устанавливает стиль заливки
- `strokeStyle` — устанавливает стиль контура
- `lineWidth` — задает толщину линии
- `lineCap` — определяет форму концов линий
- `lineJoin` — определяет форму соединения линий
- `miterLimit` — ограничивает длину острых углов
- `createLinearGradient()` — создает линейный градиент
- `createRadialGradient()` — создает радиальный градиент

# Основные методы: рисование

- `fill()` — заполняет фигуру текущим стилем заливки
- `stroke()` — рисует контур фигуры
- `clearRect(x, y, width, height)` — очищает указанную область
- `drawImage()` — рисует изображение
- `putImageData()` — помещает данные изображения
- `getImageData()` — получает данные изображения

# Основные методы: текст

- `font` — устанавливает шрифт
- `textAlign` — выравнивает текст
- `textBaseline` — задает базовую линию текста
- `fillText(text, x, y)` — рисует заполненный текст
- `strokeText(text, x, y)` — рисует контур текста

# Применение

- Игры
- Визуализации данных
- Графические редакторы
- Анимации
- Интерактивные диаграммы

# Ограничения

- Масштабирование: при увеличении может теряться качество
- Сложность: создание сложных графических интерфейсов требует много кода
- Производительность: интенсивная работа может нагружать процессор
- Интерактивность: возможна, но на низком уровне

# Туториал от MDN



[https://developer.mozilla.org/en-US/docs/Web/API/Canvas\\_API/Tutorial](https://developer.mozilla.org/en-US/docs/Web/API/Canvas_API/Tutorial)

# SVG

SVG (Scalable Vector Graphics) — это формат векторной графики, основанный на XML, который позволяет создавать и масштабировать изображения без потери качества.

```
<svg width="100" height="100">
  <circle cx="50" cy="50" r="40" fill="red" />
</svg>
```

# Основные элементы

- path — создание произвольных путей
- rect — прямоугольники
- circle — окружности
- ellipse — эллипсы
- line — линии
- polygon — многоугольники
- polyline — ломаные линии

# Особенности

- Можно работать как с обычными html-элементами
- Можно стилизовать через css
- Можно добавлять обработчики событий

# Применение

- Иконки и логотипы
- Инфографика
- Интерактивные диаграммы
- Карты
- Анимации

# Работа через JS

```
// Создание нового SVG элемента
const svgNS = 'http://www.w3.org/2000/svg';
const circle = document.createElementNS(svgNS, 'circle');

// Установка атрибутов
circle.setAttribute('cx', 50);
circle.setAttribute('cy', 50);
circle.setAttribute('r', 40);
circle.setAttribute('fill', 'red');
```

# Работа через JS

```
// Создание нового SVG элемента
const svgNS = 'http://www.w3.org/2000/svg';
const circle = document.createElementNS(svgNS, 'circle');

// Установка атрибутов
circle.setAttribute('cx', 50);
circle.setAttribute('cy', 50);
circle.setAttribute('r', 40);
circle.setAttribute('fill', 'red');
```

# Отслеживание изменений: Intersection и Resize Observers

# Intersection Observer

Intersection Observer — это современный API, который позволяет отслеживать пересечение элементов с видимой областью окна браузера. Он предоставляет эффективный способ мониторинга видимости элементов без постоянного опроса DOM.

# Базовые концепции

- Target — наблюдаемый элемент
- Root — контейнер, относительно которого происходит отслеживание
- Threshold — порог видимости

# Пример

```
const observer = new IntersectionObserver(  
  (entries, observer) => {  
    entries.forEach(entry => {  
      // Обработка пересечения  
    });  
  },  
  {  
    root: null, // null означает viewport  
    threshold: 0.5 // Срабатывание при 50% видимости  
  }  
);  
  
const target = document.querySelector('.element');  
observer.observe(target);  
// Отписка от наблюдения  
observer.unobserve(target);
```

# Свойства объекта Entry

- isIntersecting — булево значение видимости
- intersectionRatio — процент видимости
- boundingRect — границы элемента
- rootBounds — границы корневого элемента

# Doka



<https://doka.guide/js/intersection-observer/>

# Resize Observer

Resize Observer — это современный API, предназначенный для отслеживания изменений размеров элементов DOM в реальном времени. Он предоставляет эффективный способ мониторинга изменений размеров без необходимости постоянного опроса DOM.

# Пример

```
const observer = new ResizeObserver(entries => {
  entries.forEach(entry => {
    const { contentRect } = entry;
    console.log(`  
Ширина: ${contentRect.width},  
Высота: ${contentRect.height}`);
  });
});

const target = document.querySelector('.element');
observer.observe(target);
```

# Свойства объекта Entry

- contentRect — текущие размеры элемента
- borderBoxSize — размеры с учётом границ
- contentBoxSize — внутренние размеры
- target — наблюдаемый элемент

MDN



<https://developer.mozilla.org/en-US/docs/Web/API/ResizeObserver>

# Системное взаимодействие: Clipboard и Notifications

# Clipboard

Clipboard API — это современный веб-стандарт, который позволяет программам и веб-приложениям взаимодействовать с буфером обмена. Он предоставляет безопасный и стандартизованный способ чтения и записи данных в буфер обмена.

- Чтение данных из буфера обмена
- Запись данных в буфер обмена
- Работа с различными форматами данных
- Обработка ошибок и разрешений

# Особенности

- Требуется разрешение пользователя
- Работает только по событию пользователя (click, keypress)
- В некоторых браузерах требуется HTTPS

# Пример: кнопка копирования

```
const copyButton = document.querySelector('.copy-button');
copyButton.addEventListener('click', () => {
  navigator.clipboard.writeText('Текст для копирования')
    .then(() => {
      copyButton.textContent = 'Скопировано!';
    })
    .catch(err => {
      console.error('Ошибка:', err);
    });
});
```

# Notifications

Notification API — это веб-стандарт, который позволяет веб-приложениям отправлять уведомления пользователям, даже когда страница не активна. Это мощный инструмент для поддержания связи с пользователями.

# Параметры

- body — текст уведомления
- icon — иконка уведомления
- tag — уникальный идентификатор
- requireInteraction — требует взаимодействия
- data — дополнительные данные

# Ограничения

- Требуется HTTPS (кроме локального тестирования)
- Необходимость получения разрешения от пользователя
- Ограничения на частоту уведомлений
- Разные политики в разных браузерах

# Пример

```
// Запрос разрешения
Notification.requestPermission()
    .then(permission => {
        if (permission === "granted") {
            // Можно отправлять уведомления
            showNotification();
        }
    });
}

function showNotification() {
    new Notification("Новое сообщение", {
        body: "Вы получили новое сообщение!",
        icon: "/images/icon.png"
    });
}
```

# Медиа: WebRTC

# MediaStream API

MediaStream API — это веб-стандарт, который позволяет работать с потоками аудио и видео в браузере. Он предоставляет возможность захвата, обработки и передачи медиа-контента в реальном времени.

- Захват аудио и видео с устройств
- Создание и управление медиа-потоками
- Обработка медиа-данных
- Передача потоков между пользователями
- Запись медиа-контента

# Компоненты

- MediaStream — основной объект потока
- MediaStreamTrack — отдельные треки (аудио/видео)
- MediaDevices — доступ к устройствам
- MediaStreamEvent — события потока

# Пример: управление потоками

```
const stream = await navigator.mediaDevices.getUserMedia({  
    video: true,  
    audio: true  
});  
  
// Получение списка треков  
const tracks = stream.getTracks();  
  
// Управление треками  
tracks.forEach(track => {  
    track.enabled = false; // Отключение трека  
    track.stop(); // Остановка трека  
});
```

# Пример: параметры захвата потока

```
const constraints = {
  video: {
    width: { min: 1280, ideal: 1920, max: 2560 },
    height: { min: 720, ideal: 1080, max: 1440 },
    facingMode: 'user'
  },
  audio: {
    echoCancellation: true,
    noiseSuppression: true
  }
};
```

# Пример: запись видео

```
const recorder = new MediaRecorder(stream);

recorder.ondataavailable = event => {
    if (event.data.size > 0) {
        const blob = new Blob(
            [event.data],
            { type: 'video/mp4' });
        // Сохранение или отправка файла
    }
};

recorder.start(1000); // Запись каждую секунду
```

# WebRTC

WebRTC (Web Real-Time Communication) — это технология, позволяющая осуществлять передачу аудио и видео в реальном времени между пользователями через веб-браузер без необходимости установки дополнительных плагинов.

# Ограничения

- Необходимость HTTPS
- Зависимость от сетевых условий
- Сложность настройки для сложных топологий
- Потребление ресурсов
- Обязательно разрешение пользователя

# Пример: вывод изображения и звука

```
// Получение доступа к камере и микрофону
navigator.mediaDevices.getUserMedia({
    video: true,
    audio: true
})
    .then(stream => {
        const video = document.querySelector('video');
        video.srcObject = stream;
    })
    .catch(error => {
        console.error('Ошибка доступа к устройствам:', error);
    });
}
```

# Пример: создание соединения

```
// Создание объекта соединения
const peerConnection = new RTCPeerConnection();

// Добавление локального потока
navigator.mediaDevices.getUserMedia({ video: true, audio: true })
    .then(stream => {
        stream.getTracks().forEach(track => {
            peerConnection.addTrack(track, stream);
        });
    });
}
```

# Вопросы