# Отчет об архитектурной трансформации: Теория пятого дня (Неделя 11)

## Продвинутые паттерны асинхронности, сетевая безопасность и архитектура клиентских приложений

### 1. Введение: Архитектурный контекст перехода на клиентскую сторону

В рамках комплексной программы профессиональной переподготовки Unity-разработчика в Enterprise Backend инженера, пятый день одиннадцатой недели представляет собой критический этап интеграции накопленных знаний. Если предыдущие дни были посвящены изоляции среды выполнения JavaScript (Runtime), изучению цикла событий (Event Loop) и системы типов, то теория пятого дня выводит разработчика за пределы изолированной песочницы браузера к взаимодействию с внешним миром.

Для Senior C# разработчика этот переход характеризуется фундаментальной сменой парадигмы: от прямого управления сокетами, потоками и памятью к работе через высокоуровневые абстракции браузера, налагающие строгие ограничения безопасности и управления ресурсами. В среде Unity или ASP.NET Core разработчик оперирует в доверенной среде (Trusted Environment). В клиентской веб-разработке код исполняется во враждебной среде браузера пользователя, что требует принципиально иных подходов к безопасности (XSS, CORS) и управлению производительностью (Debounce, Throttle).

Данный отчет представляет собой исчерпывающую техническую спецификацию, разработанную для формирования ментальной модели Fullstack-архитектора. Мы деконструируем API браузера, проводя прямые параллели с внутренним устройством CLR (Common Language Runtime), паттернами Unity Engine и системными механизмами.NET, чтобы обеспечить максимально безболезненный перенос компетенций.

### 2. Управление частотой событий и производительность UI

#### 2.1. Проблема высокой частоты дискретизации событий

В экосистеме Unity разработчик привык к детерминированному циклу обновлений. Метод Update() вызывается один раз за кадр, и частота вызовов жестко привязана к Frame Rate (обычно 60 FPS). В браузере события ввода (Input Events) — такие как scroll, resize, mousemove или input — генерируются аппаратно и могут поступать в Event Loop с частотой, значительно превышающей частоту обновления экрана, или, наоборот, быть хаотичными.

Прямая обработка каждого такого события (например, пересчет DOM или отправка запроса на сервер при каждом нажатии клавиши) приводит к двум критическим проблемам:

1. **Layout Thrashing (Трэшинг макета):** Частые манипуляции с DOM форсируют синхронный пересчет геометрии страницы (Reflow), блокируя главный поток.1
2. **Network Flooding:** Отправка HTTP-запроса на каждый символ ввода (input event) создает эффект DDoS-атаки на собственный бэкенд.

Для решения этих задач используются паттерны ограничения частоты вызовов: **Debounce** и **Throttle**. Для C#-разработчика, привыкшего к корутинам и FixedUpdate, эти паттерны требуют переосмысления управления временем выполнения.

#### 2.2. Сравнительный анализ: Unity Coroutines vs JS Patterns

| **Характеристика** | **Throttle (JavaScript)** | **Debounce (JavaScript)** | **Unity Analogy (C#)** |
| --- | --- | --- | --- |
| **Логика запуска** | Гарантированное выполнение не чаще чем раз в N мс. | Выполнение только после того, как поток событий прекратится на N мс. | Update() с таймером vs Coroutine с отменой. |
| **Сценарий использования** | Скроллинг, изменение размера окна (Resize), анимация при движении мыши. | Поиск (Typeahead), валидация форм, сохранение драфта. | FixedUpdate (физика) vs Ввод текста в GUI. |
| **Механизм ожидания** | Игнорирование вызовов в период "остывания". | Сброс таймера при каждом новом событии. | Time.time check vs StopCoroutine(). |
| **Гарантия исполнения** | Регулярная (промежуточные результаты важны). | Отложенная (важен только конечный результат). | Плавное движение vs Завершение ввода. |

Throttle (Ограничение пропускной способности)

Этот паттерн гарантирует, что функция вызывается с фиксированной периодичностью, независимо от того, насколько часто срабатывает событие. Это критически важно для таких задач, как бесконечная прокрутка (Infinite Scroll), где необходимо проверять позицию скролла, но делать это на каждый пиксель сдвига слишком дорого.2

* **Аналогия в Unity:** Это классический паттерн ограничения частоты стрельбы или проверки условий внутри Update():  
  C#  
  // Unity C# Throttle Analogy  
  float lastFireTime;  
  float fireRate = 0.1f; // 100ms  
    
  void Update() {  
   if (Input.GetButton("Fire1") && Time.time > lastFireTime + fireRate) {  
   Fire();  
   lastFireTime = Time.time;  
   }  
  }  
    
  В JavaScript реализация полагается на замыкание (Closure), которое хранит состояние lastRan между вызовами функции, так как Update цикла не существует в явном виде.2

Debounce (Отложенное выполнение)

Паттерн гарантирует вызов функции только после наступления "тишины" в потоке событий. Если события продолжают поступать (пользователь печатает), таймер постоянно сбрасывается.

* **Аналогия в Unity:** Представьте корутину, которая запускается при вводе игрока. Если игрок нажимает кнопку снова до завершения таймера, предыдущая корутина останавливается, и запускается новая.  
  C#  
  // Unity C# Debounce Analogy  
  Coroutine searchRoutine;  
    
  public void OnInputChanged(string text) {  
   if (searchRoutine!= null) StopCoroutine(searchRoutine);  
   searchRoutine = StartCoroutine(SearchDelay(text));  
  }  
    
  IEnumerator SearchDelay(string text) {  
   yield return new WaitForSeconds(0.5f);  
   PerformSearch(text);  
  }  
    
  В JavaScript это реализуется через clearTimeout(timerId) и установку нового setTimeout. Механика замыканий позволяет "захватить" ID таймера и управлять им при последующих вызовах той же функции.4

#### 2.3. Внутренняя реализация и управление памятью (V8 vs CLR)

Критическое отличие реализации этих паттернов в JS от C# заключается в **управлении памятью**. В C# для реализации сложной реактивности мы часто используем Rx.NET (Observable.Throttle) или поля класса для хранения состояния таймеров. В Vanilla JS стандартом является использование функций высшего порядка (Higher-Order Functions), возвращающих декорированную функцию.

Анализ аллокаций V8 (Contexts):

Когда создается debounced-функция:

JavaScript

function debounce(func, wait) {  
 let timeout; // Переменная в Scope  
 return function(...args) {  
 const context = this;  
 clearTimeout(timeout);  
 timeout = setTimeout(() => func.apply(context, args), wait);  
 };  
}

Движок V8 аллоцирует специальный объект контекста в куче (Heap), чтобы хранить переменную timeout, так как она используется во вложенной стрелочной функции. Это аналогично тому, как компилятор Roslyn создает скрытый класс (DisplayClass) для замыканий, захватывающих локальные переменные.6 Однако, в отличие от C#, где жизненный цикл объекта часто привязан к MonoBehaviour, в JS ссылка на эту функцию (и, следовательно, на весь контекст) удерживается обработчиком события DOM. Это создает потенциал для утечек памяти, если не удалять обработчики событий (removeEventListener) при уничтожении компонентов ("unmount"), что является аналогом OnDestroy в Unity.8

### 3. Сетевое взаимодействие: Fetch API как замена HttpClient

#### 3.1. Эволюция и архитектурные различия

Для разработчика.NET основным инструментом сетевого взаимодействия является System.Net.Http.HttpClient. В браузере историческим аналогом был XMLHttpRequest (XHR), но современным стандартом является **Fetch API**. Переход с HttpClient на fetch требует понимания нескольких фундаментальных различий в дизайне API и обработке потоков данных.

| **Функция** | **.NET HttpClient** | **JS Fetch API** | **Архитектурное примечание** |
| --- | --- | --- | --- |
| **Асинхронная модель** | Task<HttpResponseMessage> | Promise<Response> | Оба используют модель Future/Promise, но Task в C# более тяжеловесен и поддерживает контекст синхронизации. |
| **Чтение тела** | ReadAsStringAsync() | response.json(), response.text() | В JS методы чтения тела возвращают *новый* Promise, так как тело является потоком (ReadableStream).9 |
| **Конфигурация** | BaseAddress, DefaultRequestHeaders | Объект options в каждом вызове | В JS нет встроенного концепта "экземпляра клиента" с состоянием; обычно пишутся обертки-фабрики. |
| **Обработка ошибок** | EnsureSuccessStatusCode() (опционально) | Ручная проверка response.ok | Fetch *не* выбрасывает исключение при 404/500 ошибках.9 |

#### 3.2. Философия обработки ошибок

Здесь кроется одна из главных ловушек для C# разработчиков. В экосистеме.NET принято, что неудачный HTTP-код (например, 500 Internal Server Error) — это исключительная ситуация, которую часто обрабатывают через try-catch или глобальные фильтры исключений.

В **Fetch API** промис переходит в состояние rejected (выбрасывает исключение) **только при сетевом сбое** (DNS error, потеря соединения, CORS error). Если сервер ответил (даже кодом 500 или 404), промис считается fulfilled (выполненным успешно). Это поведение обусловлено тем, что с точки зрения сетевого протокола HTTP транзакция завершилась корректно — ответ получен.9

**Паттерн реализации:**

JavaScript

// Правильная обработка fetch для C# разработчика  
try {  
 const response = await fetch(url);  
 if (!response.ok) { // Аналог!response.IsSuccessStatusCode  
 // Необходимо вручную выбросить ошибку, чтобы попасть в catch  
 throw new Error(`HTTP error! status: ${response.status}`);  
 }  
 const data = await response.json();  
} catch (error) {  
 // Сюда попадем и при сетевой ошибке, и при ручном throw  
 console.error('Fetch failed:', error);  
}

#### 3.3. Потоковая обработка (Streaming)

В C# HttpClient позволяет управлять моментом завершения задачи через HttpCompletionOption.ResponseHeadersRead. Fetch API работает аналогично по умолчанию: промис fetch разрешается, как только получены заголовки ответа. Тело ответа (response.body) представляет собой ReadableStream. Это позволяет начинать обработку данных (например, JSON-парсинг или проигрывание видео) до того, как весь контент загружен по сети, что критично для производительности на мобильных устройствах.10

### 4. Паттерны отмены операций: AbortController

#### 4.1. Сравнительный анализ: CancellationToken vs AbortSignal

В современной асинхронной разработке возможность отмены операции (Cancellation) является обязательным требованием. В.NET это реализовано через унифицированный паттерн CancellationTokenSource (CTS) и CancellationToken. В JavaScript долгое время не было стандарта, пока не был внедрен **AbortController**.

Эти механизмы концептуально идентичны, но имеют различия в реализации, критичные для системного архитектора.12

| **Компонент** | **.NET (C#)** | **JavaScript (Web API)** |
| --- | --- | --- |
| **Инициатор (Trigger)** | CancellationTokenSource | AbortController |
| **Токен (Listener)** | CancellationToken (struct) | AbortSignal (object) |
| **Метод отмены** | cts.Cancel() | controller.abort(reason) |
| **Проверка статуса** | token.IsCancellationRequested | signal.aborted |
| **Механизм подписки** | token.Register(callback) | signal.addEventListener('abort', cb) |
| **Распространение** | Исключение OperationCanceledException | Исключение DOMException (AbortError) |

#### 4.2. Внутренняя механика и управление ресурсами

**В.NET:** CancellationToken — это легковесная структура (struct), что снижает нагрузку на GC при передаче токена вглубь стека вызовов.7 Отмена часто проверяется через поллинг свойства IsCancellationRequested в циклах или через выброс исключения ThrowIfCancellationRequested().

**В V8/Browser:** AbortSignal наследуется от EventTarget. Это означает, что механизм отмены полностью событийно-ориентирован. Когда вызывается controller.abort(), браузер диспетчеризирует событие abort всем подписчикам.

* **Fetch API:** Внутренне подписывается на событие abort сигнала. При срабатывании он немедленно обрывает сетевое соединение на уровне сокетов браузера и переводит промис в состояние rejected с ошибкой AbortError.14
* **Память:** Поскольку AbortSignal использует addEventListener, критически важно понимать, что это создает сильную ссылку. В отличие от C#, где CancellationTokenRegistration реализует IDisposable для очистки, в JS разработчик должен следить за удалением слушателей, если использует AbortSignal для своей логики, а не просто передает его в fetch.

#### 4.3. Паттерн "Race Condition Safe" поиска

Рассмотрим классическую задачу "Live Search" (поиск при вводе). Если пользователь вводит "A", затем быстро "AB", может возникнуть состояние гонки, когда ответ на первый запрос ("A") придет *позже* ответа на второй ("AB"), перезаписав актуальные данные устаревшими.

В Unity мы бы проверяли актуальность корутины или использовали бы timestamp. В JS идиоматическим решением является отмена предыдущего запроса через AbortController.15

JavaScript

let currentController = null; // Хранение ссылки на активный контроллер  
  
async function onSearchInput(query) {  
 // 1. Отмена предыдущего запроса (аналог cts.Cancel())  
 if (currentController) {  
 currentController.abort();  
 }  
  
 // 2. Создание нового контроллера  
 currentController = new AbortController();  
 const { signal } = currentController;  
  
 try {  
 const response = await fetch(`/api/search?q=${query}`, { signal });  
 // Если abort вызван здесь, fetch выбросит исключение  
 const data = await response.json();  
 renderResults(data);  
 } catch (error) {  
 // 3. Фильтрация ошибок отмены (аналог catch (OperationCanceledException))  
 if (error.name === 'AbortError') {  
 console.log('Запрос отменен (Stale Request)'); // Игнорируем  
 } else {  
 console.error('Ошибка сети', error); // Реальная ошибка  
 }  
 }  
}

Этот паттерн гарантирует консистентность UI без необходимости ручного сравнения временных меток ответов.

### 5. Безопасность и CORS: Браузер как Операционная Система

#### 5.1. Same-Origin Policy (SOP) vs Unity Editor

Для разработчика десктопных приложений (WPF, WinForms) или игр (Unity Standalone), сетевые запросы практически не ограничены. HttpClient может отправить GET-запрос на любой домен (google.com, my-api.com). В браузере ситуация фундаментально иная.

Браузер действует как изолированная песочница с жесткой политикой безопасности **Same-Origin Policy (SOP)**. По умолчанию скрипт, загруженный с источника Origin A (например, http://localhost:3000), не может читать данные ответа от Origin B (например, https://api.mygame.com), если они отличаются протоколом, доменом или портом.17

**Важное наблюдение:** Unity Editor при тестировании игры игнорирует SOP, так как работает в контексте нативного приложения (Mono/.NET). Однако при билде под **WebGL**, игра начинает работать в контексте браузера, и сетевой стек Unity заменяется на fetch (или XMLHttpRequest), что приводит к внезапным ошибкам доступа к API, которые работали в редакторе.19

#### 5.2. Механизм CORS и Preflight Requests

**CORS (Cross-Origin Resource Sharing)** — это механизм, позволяющий серверам *явно разрешать* исключения из SOP. Это не "баг", который нужно "обойти", а система защиты пользователя.

Особое внимание следует уделить механизму **Preflight Requests**. При выполнении "сложных" запросов (например, POST с JSON-телом или наличие кастомных заголовков, таких как Authorization или X-Unity-Version), браузер перед основным запросом автоматически отправляет предварительный запрос методом **OPTIONS**.21

**Алгоритм Preflight (Сценарий Unity WebGL):**

1. **Unity (JS):** Пытается отправить POST /api/save с заголовком Content-Type: application/json.
2. **Браузер:** Детектирует нестандартный заголовок. Блокирует POST. Отправляет OPTIONS /api/save.
3. **Вопрос браузера:** "Сервер, принимаешь ли ты POST запросы с Content-Type от источника game.com?"
4. **Сервер (ASP.NET Core):** Должен ответить 200 OK с заголовками:
   * Access-Control-Allow-Origin: https://game.com
   * Access-Control-Allow-Methods: POST, OPTIONS
   * Access-Control-Allow-Headers: Content-Type
5. **Браузер:** Если заголовки совпадают, отправляет реальный POST запрос.

**Типичная ошибка:** Разработчики бэкенда часто забывают настроить обработку метода OPTIONS или Middleware CORS (app.UseCors()) в ASP.NET Core, из-за чего WebGL билд падает с ошибкой "CORS header missing", хотя curl и Postman работают (так как они не отправляют Preflight).23

### 6. Работа с DOM: Векторы атак и XSS

#### 6.1. XSS (Cross-Site Scripting) как аналог Remote Code Execution

В мире Unity концепция "инъекции кода" через UI практически отсутствует (если не использовать System.Reflection на пользовательских строках). В вебе **XSS (Cross-Site Scripting)** — это самая распространенная уязвимость, возникающая при трактовке пользовательских данных как кода.

Атака XSS позволяет злоумышленнику внедрить произвольный JavaScript скрипт на страницу, который будет выполнен браузером жертвы в контексте вашего домена. Это дает доступ к кукам, токенам сессии и позволяет выполнять действия от имени пользователя.25

#### 6.2. innerHTML vs textContent: Производительность и Безопасность

В JavaScript существуют два основных способа записи содержимого в DOM-элемент. Для V8 эти операции имеют кардинально разную стоимость.

| **Свойство** | **innerHTML** | **textContent** |
| --- | --- | --- |
| **Механика** | Запускает HTML-парсер. Строит новое DOM-поддерево. Выполняет скрипты (в некоторых контекстах). | Вставляет строку как есть (Plain Text). Экранирует HTML-теги. |
| **Безопасность** | **Опасно (XSS Risk).** Строка <img src=x onerror=alert(1)> приведет к выполнению JS-кода. | **Безопасно.** Теги отобразятся как текст. |
| **Производительность** | **Низкая.** Парсинг HTML — дорогая операция, требующая времени CPU.27 Вызывает Layout Thrashing. | **Высокая.** Минимальные накладные расходы. Оптимизирован в V8 для быстрой вставки текста.28 |

Рекомендация архитектора: Всегда используйте textContent для вывода данных. Использование innerHTML оправдано только в редких случаях (например, рендеринг Rich Text из доверенного источника), и даже тогда данные должны предварительно проходить через санитайзер (например, библиотеку DOMPurify).

В контексте Unity это аналогично разнице между Text.text = str (безопасно) и попыткой скомпилировать пользовательскую строку через CSharpCodeProvider и выполнить её (фатально опасно).29

### 7. Архитектурный паттерн MVC на Vanilla JS

#### 7.1. Отказ от "God Object" и MonoBehaviour

В Unity распространен паттерн, когда один класс (PlayerController : MonoBehaviour) отвечает и за логику перемещения, и за чтение ввода (Input.GetRequest), и за обновление UI, и за воспроизведение звуков. В энтерпрайз веб-разработке такое смешение ответственности (Coupling) недопустимо.

Для закрепления теории пятого дня предлагается реализация паттерна **MVC (Model-View-Controller)** на чистом JS без фреймворков. Это необходимо, чтобы понять проблемы, которые решают React/Angular, такие как синхронизация состояния и DOM.

#### 7.2. Компоненты архитектуры

1. **Model (Данные и Логика):**
   * Аналог POCO (Plain Old CLR Object) классов.
   * Не имеет ссылок на DOM или HTML.
   * Реализует паттерн Observer (аналог C# event Action), чтобы уведомлять подписчиков об изменениях данных.

JavaScript  
class TodoModel {  
 constructor() {  
 this.todos =;  
 this.onTodoListChanged = new Event(); // Custom implementation  
 }  
 addTodo(text) {  
 this.todos.push({ text, complete: false });  
 this.onTodoListChanged.notify(this.todos); // Invoke event  
 }  
}

1. **View (Представление):**
   * Отвечает *только* за манипуляции с DOM.
   * Знает, как найти элементы (querySelector) и как их обновить (textContent).
   * Не содержит бизнес-логики.
   * Выставляет методы для связывания событий, например bindAddTodo(handler), которые вызывают переданный колбэк при клике.
2. **Controller (Связующее звено):**
   * Инициализирует Model и View.
   * Подписывается на события View (UI Events) -> вызывает методы Model.
   * Подписывается на события Model (Data Changes) -> вызывает методы обновления View.

**Ключевое отличие от Unity:** В Unity UI часто обновляется в Update() (polling) — каждый кадр мы проверяем text.text = score.ToString(). В веб-MVC обновление происходит **реактивно (push)**: DOM трогается только тогда, когда модель действительно изменилась. Это критически важно для производительности в браузере, так как DOM-операции на порядки медленнее изменения переменных в памяти.12

### 8. Сводная матрица миграции API (.NET -> Web)

Для ускорения адаптации приводится таблица прямого маппинга концепций.

| **Задача / Концепция** | **.NET / Unity API** | **JavaScript / Web API** | **Комментарий архитектора** |
| --- | --- | --- | --- |
| **HTTP Запрос** | HttpClient.GetAsync() | fetch(url) | JS требует ручной проверки response.ok, так как 404/500 не считаются ошибками транспорта.9 |
| **Отмена операции** | CancellationTokenSource | AbortController | AbortSignal работает через события (EventTarget), а не через состояние структуры.12 |
| **Задержка** | yield return new WaitForSeconds() | setTimeout | В JS таймеры не гарантируют точность (throttling), выполняются в макрозадачах.2 |
| **Потокобезопасность** | lock, Monitor, Interlocked | Не требуется | JS однопоточный. Конкурентность управляется через Event Loop. Нет lock.31 |
| **События** | event Action, UnityEvent | EventTarget, CustomEvent | DOM-события имеют фазы всплытия (Bubbling) и перехвата (Capturing), чего нет в C#. |
| **Рендеринг текста** | Text.text (UGUI) | node.textContent | Использовать innerHTML — аналог eval(), высокий риск XSS.30 |

### 9. Заключение

Теория пятого дня формирует у инженера понимание браузера как сложной, асинхронной и небезопасной среды исполнения. Понимание различий между Macrotask (setTimeout) и Microtask (Promise), умение использовать AbortController для предотвращения гонки данных и знание механизмов CORS для настройки бэкенда — это фундаментальные навыки, отделяющие "кодера" от "архитектора". Эти знания станут базой для изучения React на следующей неделе, где многие из этих механизмов (Virtual DOM, Synthetic Events) являются абстракциями над разобранными здесь низкоуровневыми API.

#### Источники

1. Throttle vs Debounce in JavaScript: The Complete Guide for Every Developer - Medium, дата последнего обращения: декабря 6, 2025, <https://medium.com/@muhebollah.diu/throttle-vs-debounce-in-javascript-the-complete-guide-for-every-developer-ab0508ada75b>
2. Difference between throttling and debouncing a function - Stack Overflow, дата последнего обращения: декабря 6, 2025, <https://stackoverflow.com/questions/25991367/difference-between-throttling-and-debouncing-a-function>
3. Debounce vs Throttle JavaScript — Real Scroll Demo for Beginners - YouTube, дата последнего обращения: декабря 6, 2025, <https://www.youtube.com/watch?v=n8S4-EDezrc>
4. Update vs Coroutine — Elegance, Performance, and Standards? : r/Unity3D - Reddit, дата последнего обращения: декабря 6, 2025, <https://www.reddit.com/r/Unity3D/comments/evbq0v/update_vs_coroutine_elegance_performance_and/>
5. How Can I Make This Debounce Script Work? Unity3D - Stack Overflow, дата последнего обращения: декабря 6, 2025, <https://stackoverflow.com/questions/75327452/how-can-i-make-this-debounce-script-work-unity3d>
6. .innterHTML vs .textContent : r/learnjavascript - Reddit, дата последнего обращения: декабря 6, 2025, <https://www.reddit.com/r/learnjavascript/comments/18657sp/innterhtml_vs_textcontent/>
7. Why CancellationToken is a struct? - Stack Overflow, дата последнего обращения: декабря 6, 2025, <https://stackoverflow.com/questions/52937589/why-cancellationtoken-is-a-struct>
8. The complete guide to the AbortController API - LogRocket Blog, дата последнего обращения: декабря 6, 2025, <https://blog.logrocket.com/complete-guide-abortcontroller/>
9. Axios and Fetch API? Choosing the Right HTTP Client | Syncfusion Blogs, дата последнего обращения: декабря 6, 2025, <https://www.syncfusion.com/blogs/post/axios-vs-fetch-choose-right-http-client>
10. Fetch API vs. Axios vs. Alova: Which HTTP Client Should You Use in 2025?, дата последнего обращения: декабря 6, 2025, <https://www.freecodecamp.org/news/fetch-api-vs-axios-vs-alova/>
11. HTTP Client in C#: Best Practices for Experts | by iamprovidence - Medium, дата последнего обращения: декабря 6, 2025, <https://medium.com/@iamprovidence/http-client-in-c-best-practices-for-experts-840b36d8f8c4>
12. Do You Really Know AbortController? - DEV Community, дата последнего обращения: декабря 6, 2025, <https://dev.to/leapcell/do-you-really-know-abortcontroller-3628>
13. Escaping the Cancel Button Trap: AbortController and CancellationToken in ASP.NET API, дата последнего обращения: декабря 6, 2025, <https://www.rahulpnath.com/blog/abortcontroller-cancellationtoken-dotnet>
14. AbortController: abort() method - Web APIs - MDN Web Docs, дата последнего обращения: декабря 6, 2025, <https://developer.mozilla.org/en-US/docs/Web/API/AbortController/abort>
15. Canceling Fetch Requests in JavaScript with AbortController - Medium, дата последнего обращения: декабря 6, 2025, <https://medium.com/@AlexanderObregon/canceling-fetch-requests-in-javascript-with-abortcontroller-98c11d2ab54e>
16. Abort Controller and Cancellation Tokens | Cancel Your Work All the Way | ASP NET, дата последнего обращения: декабря 6, 2025, <https://www.youtube.com/watch?v=tSEc5hXx1TM>
17. Understanding CORS: A Developer's Guide - Stack by Convex, дата последнего обращения: декабря 6, 2025, <https://stack.convex.dev/a-developers-guide-to-cors>
18. Cross-Origin Resource Sharing (CORS) - HTTP - MDN Web Docs, дата последнего обращения: декабря 6, 2025, <https://developer.mozilla.org/en-US/docs/Web/HTTP/Guides/CORS>
19. Difference between HttpClient and Unity's UnityWebRequest/WWW API - Stack Overflow, дата последнего обращения: декабря 6, 2025, <https://stackoverflow.com/questions/50160380/difference-between-httpclient-and-unitys-unitywebrequest-www-api>
20. WebGL Networking - Unity - Manual, дата последнего обращения: декабря 6, 2025, <https://docs.unity3d.com/560/Documentation/Manual/webgl-networking.html>
21. Preflight request - Glossary - MDN Web Docs, дата последнего обращения: декабря 6, 2025, <https://developer.mozilla.org/en-US/docs/Glossary/Preflight_request>
22. Understanding CORS and Preflight Requests in APIs | by Bale - Medium, дата последнего обращения: декабря 6, 2025, <https://medium.com/@bloodturtle/understanding-cors-and-preflight-requests-in-apis-e088ae13b417>
23. Cors blocking requests on WebGL build Unity - Stack Overflow, дата последнего обращения: декабря 6, 2025, <https://stackoverflow.com/questions/78588671/cors-blocking-requests-on-webgl-build-unity>
24. CORS, Cookies, Unity and WebGL Builds, дата последнего обращения: декабря 6, 2025, <https://blog.kongregate.com/cors-cookies-unity-and-webgl-builds/>
25. Cross-site scripting (XSS) - Security - MDN Web Docs, дата последнего обращения: декабря 6, 2025, <https://developer.mozilla.org/en-US/docs/Web/Security/Attacks/XSS>
26. What Is Cross-Site Scripting (XSS)? - Palo Alto Networks, дата последнего обращения: декабря 6, 2025, <https://www.paloaltonetworks.com/cyberpedia/xss-cross-site-scripting>
27. The cost of JavaScript in 2019 - V8.dev, дата последнего обращения: декабря 6, 2025, <https://v8.dev/blog/cost-of-javascript-2019>
28. innerText vs innerHtml vs textContent || Difference & When to use each best! - DEV Community, дата последнего обращения: декабря 6, 2025, <https://dev.to/victorusese/innertext-vs-innerhtml-vs-textcontent-difference-when-to-use-each-best-4i8o>
29. Why is it Better to use textContent instead of innerHTML ? - GeeksforGeeks, дата последнего обращения: декабря 6, 2025, <https://www.geeksforgeeks.org/javascript/why-is-it-better-to-use-textcontent-instead-of-innerhtml/>
30. JavaScript innerHTML, innerText, and textContent - DEV Community, дата последнего обращения: декабря 6, 2025, <https://dev.to/4myc/javascript-innerhtml-innertext-and-textcontent-ih>
31. In depth: Microtasks and the JavaScript runtime environment - Web APIs | MDN, дата последнего обращения: декабря 6, 2025, <https://developer.mozilla.org/en-US/docs/Web/API/HTML_DOM_API/Microtask_guide/In_depth>