# Отчет об архитектурной трансформации: Стратегия обучения JavaScript Runtime и асинхронным паттернам (Неделя 11)

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

В рамках комплексной программы профессиональной переподготовки Unity-разработчика в Enterprise Backend инженера, 11-я неделя представляет собой критически важный этап, который можно охарактеризовать как «инверсию ментальной модели исполнения». Согласно исходному плану обучения 1, предыдущие десять недель были посвящены углубленному изучению экосистемы.NET, управлению памятью CLR (Common Language Runtime) и многопоточности на уровне ядра ОС. Теперь, переходя к фазе «Frontend-мост», инженер сталкивается с необходимостью освоить среду выполнения JavaScript, которая фундаментально отличается от привычного детерминированного мира игровых циклов Unity или пулов потоков.NET.

Этот отчет представляет собой детально проработанный план обучения, рассчитанный на 20 часов интенсивной работы. Однако это не просто расписание занятий; это глубокий сравнительный анализ архитектурных паттернов, призванный связать существующий опыт работы с C# и Unity с реалиями браузерного исполнения кода. Основная сложность для Senior Unity-разработчика заключается не в синтаксисе JavaScript (который, как и C#, принадлежит к C-подобному семейству 2), а в понимании однопоточной модели конкурентности, основанной на цикле событий (Event Loop), и динамической природы управления памятью.

В отличие от Unity, где разработчик имеет прямой контроль над Update() и FixedUpdate() циклами, и в отличие от ASP.NET Core, где параллелизм достигается за счет использования множества потоков, JavaScript в браузере оперирует в условиях жестких ограничений UI-потока. Любая блокирующая операция здесь фатальна для пользовательского опыта, что требует переосмысления подходов к асинхронности. Если в C# Task.Delay или Thread.Sleep приостанавливают поток, возвращая его в пул, то в JS аналогичные операции реализуются через отложенные сообщения в очередях задач.3

Ниже представлен исчерпывающий план и методическое руководство, структурированное по пяти учебным дням. Каждый раздел сопровождается глубоким теоретическим обоснованием, проводящим параллели с.NET, и практическими заданиями, направленными на формирование новых нейронных связей у разработчика.

## 2. Глобальный системный промт для AI-Ментора (GEM-Bot)

Для обеспечения максимальной эффективности процесса самообразования и эмуляции парного программирования с опытным наставником, на данной неделе внедряется специализированный системный промт. Этот инструмент настроен на трансформацию стандартных ответов языковой модели в формат архитектурного ревью, используя терминологию, понятную C#-разработчику.

**Промт для интеграции (Скопировать в контекст LLM):**

**System Prompt Override: The Senior Polyglot Architect Persona**

**Роль:** Ты — Senior Principal Architect с 15-летним опытом в.NET (CLR Internals, High-Performance Computing) и Modern Web (V8 Internals, Browser Rendering Optimization). Твоя текущая задача — менторить Senior Unity-разработчика, который переходит в Fullstack Enterprise разработку.

**Твоя ментальная модель и стиль коммуникации:**

1. **Принцип компаративного анализа:** Объясняя любую концепцию JavaScript, ты обязан начинать с аналогии в C# или Unity API.
   * *Пример:* Если объясняешь Event Loop, сравнивай его с CoroutineScheduler в Unity, но акцентируй внимание на отсутствии истинного параллелизма.
   * *Пример:* Объясняя Prototypal Inheritance, сравнивай это с виртуальными таблицами методов (vtable), но динамически изменяемыми в Runtime, в отличие от статической компиляции C#.
2. **Техническая глубина:** Используй терминологию системного уровня. Не говори "JS однопоточный", говори "V8 использует один Call Stack и делегирует I/O операции через libuv (в Node) или Web APIs (в браузере)". Говори о Stack, Heap, Hidden Classes (Shapes), Garbage Collection Generations.
3. **Code Review Mode:** Если студент присылает код, написанный в "C# стиле" (например, избыточные классы, блокирующие вызовы, игнорирование замыканий), ты должен жестко критиковать это как анти-паттерн в JS контексте.
4. **Отсечение магии:** На вопрос "как сделать X" сначала объясни физику процесса (memory layout, execution context), и только потом давай синтаксис.

Контекст Недели 11:

Мы фокусируемся на "Vanilla JavaScript" и "Browser Internals". Никаких фреймворков (React/Angular) на этой неделе.

Ключевые темы: Execution Context, Hoisting (аналогия с C# pre-compiler/IL generation), Event Loop (Microtasks vs Macrotasks), DOM Manipulation (аналог Scene Graph manipulation).

**Твоя цель:** Разрушить иллюзию, что JS — это "простой скриптовый язык", и показать его как сложную, событийно-ориентированную систему с уникальной моделью памяти.

## 3. Детальный план обучения: День за днем

### День 1: Среда выполнения (Runtime) и Система типов

Первый день посвящен деконструкции мифов о простоте JavaScript и погружению в низкоуровневое устройство движка V8. Для разработчика, привыкшего к строгой статической типизации C# и компиляции в Intermediate Language (IL), динамическая природа JS может показаться хаотичной. Задача этого дня — показать систему в этом хаосе, объяснив механизмы JIT-компиляции и управления памятью.

#### Архитектурный анализ: CLR vs V8

В.NET CLR управление памятью и типами происходит строго детерминированно. Типы известны на этапе компиляции (за исключением dynamic), и память аллоцируется в стеке (для Value Types) или в куче (для Reference Types). В V8 (движок JS в Chrome и Node.js) ситуация иная: все является объектом, но движок пытается оптимизировать хранение.

Критически важно понять концепцию Hidden Classes (Shapes). Когда в C# вы объявляете класс, его структура фиксирована в памяти. В JS, добавляя свойство к объекту, вы меняете его "форму". V8 создает скрытые классы на лету, чтобы оптимизировать доступ к полям. Понимание этого механизма позволяет писать JS-код, который работает почти так же быстро, как C#.2

Другой важный аспект — **Scope (Область видимости)**. В C# область видимости переменной жестко привязана к блоку кода. В JS, до стандарта ES6 (появление let и const), существовал только функциональный скоуп. Механизм **Hoisting** (поднятие переменных) часто сбивает с толку C#-разработчиков, так как создает иллюзию, что переменная доступна до её объявления. Это можно сравнить с тем, как компилятор C# генерирует поля для замыканий в начале метода, но в JS это поведение языка, а не деталь реализации компилятора.5

#### Темы теории (День 1)

1. **V8 Engine Pipeline:** Parsing -> AST -> Ignition (Interpreter) -> TurboFan (Optimizing Compiler). Как V8 оптимизирует динамический код.
2. **Memory Model:** Stack vs Heap в JS. Хранение примитивов (Smi - Small Integers) и объектов. Сравнение с Boxing/Unboxing в C#.
3. **Type Coercion (Неявное приведение):** Таблицы истинности. Почему ==!. Сравнение с операторами перегрузки в C#.
4. **Execution Context & Scope:** Global, Function, Block scopes. Hoisting и Temporal Dead Zone. Аналогии с var в C#.
5. **Reference Types:** Мутабельность объектов и массивов. Поверхностное (Shallow) и глубокое (Deep) копирование.

#### Промт для глубокого исследования (Day 1 Theory Generation)

Задание для AI:

Подготовь технический отчет на тему: "Сравнительный анализ Runtime архитектуры:.NET CLR против Google V8".

**Обязательные разделы:**

1. **Система типов:** Объясни разницу между "Nominal Typing" (C#) и "Duck Typing" (JS). Как V8 использует Hidden Classes (Shapes) и Inline Caches (IC) для эмуляции производительности статических типов?
2. **Управление памятью:** Сравни Garbage Collection в.NET (Generational GC: Gen0, Gen1, Gen2, LOH) и в V8 (Orinoco: Scavenger для Young Generation и Mark-Sweep-Compact для Old Generation).
3. **Hoisting vs IL:** Проведи параллель между механизмом Hoisting в JS и тем, как C# компилятор преобразует локальные переменные в IL-код. Покажи пример, где var в JS ведет себя иначе, чем var (type inference) в C#.
4. **Практические последствия:** Почему изменение "формы" объекта (добавление свойств после инициализации) убивает производительность в V8 (Deoptimization)?

#### Практические задания (День 1)

**Цель:** Почувствовать "пластичность" типов JS и научиться писать код, дружелюбный к оптимизатору V8.

1. **Лабораторная работа "Type Coercion Lab":**
   * Создать набор функций, принимающих аргументы any.
   * Реализовать "калькулятор", который корректно обрабатывает сложение массива с числом, булева значения со строкой и т.д., предсказывая результат.
   * *Задача:* Объяснить результат выражения (! +)[+] + (! +)[+!+]. (Это классический пример "WTF JS", который заставляет понять приведение к примитивам).
2. **Задание "Optimized Objects":**
   * Написать скрипт, создающий 100,000 объектов двумя способами:
     1. Добавляя свойства динамически (obj.a = 1; obj.b = 2).
     2. Инициализируя всё в конструкторе/литерале.
   * Замерить производительность доступа к полям. Это демонстрирует работу Hidden Classes, аналогичную структурам в C#.
3. **Анализ Hoisting:**
   * Написать код с использованием var внутри циклов и условий, предсказать вывод.
   * Переписать код на let/const, объяснив разницу в Block Scope (аналог {} блоков в C#).6

### День 2: Асинхронность и Event Loop (Сдвиг парадигмы)

Этот день является самым сложным для понимания при переходе с C#. В.NET многопоточность реальна: потоки ОС исполняются на разных ядрах CPU. В JS поток один. Весь параллелизм — это иллюзия, создаваемая быстрым переключением задач в цикле событий.

#### Архитектурный анализ: ThreadPool vs Event Loop

В C# для I/O операций используется Task, который (обычно) полагается на IOCP (I/O Completion Ports) операционной системы. Когда задача завершается, поток из ThreadPool подхватывает продолжение (Continuation).

В JS модель иная: есть один поток исполнения. Если вы напишете while(true), вы "убьете" всё приложение (включая рендеринг UI). Асинхронность достигается за счет Event Loop.

Ключевое различие, которое должен усвоить архитектор:

* В C# await означает "возможно, я вернусь в другом потоке" (если нет SynchronizationContext).
* В JS await означает "я вернусь в том же потоке, но позже, когда стек освободится".7

Важно разделить очереди задач: **Macrotasks** (setTimeout, I/O) и **Microtasks** (Promise.then, queueMicrotask). Микрозадачи имеют приоритет и могут блокировать цикл событий, вызывая голодание (Starvation) макрозадач (например, рендеринга).9

| **Характеристика** | **C# ThreadPool** | **JS Event Loop** |
| --- | --- | --- |
| **Параллелизм** | Истинный (Multi-core) | Кооперативный (Single-core) |
| **Блокировка** | Блокирует один поток | Блокирует всё приложение |
| **Приоритеты** | Thread Priority | Microtask > Macrotask |
| **Механизм ожидания** | Context Switch | Polling / Callback Queue |

#### Темы теории (День 2)

1. **Single Thread Model:** Call Stack и Heap. Почему нет lock и Monitor.
2. **Web APIs:** Роль браузера (Timer API, Fetch, DOM Events) как внешнего исполнителя асинхронных операций.
3. **The Event Loop Mechanics:** Алгоритм работы: Check Stack -> Execute Microtasks -> Render -> Execute Macrotask.10
4. **Promises vs Tasks:** Внутреннее устройство Promise. States (Pending, Fulfilled, Rejected). Сравнение с System.Threading.Tasks.Task.
5. **Async/Await Implementation:** Генераторы и их роль в реализации async. Сравнение с State Machine в C#.

#### Промт для глубокого исследования (Day 2 Theory Generation)

Задание для AI:

Разработай глубокое руководство "Асинхронная архитектура: От C# Task к JS Promise".

**Ключевые вопросы для анализа:**

1. **Event Loop Visualized:** Опиши пошагово, что происходит в движке, когда вызывается setTimeout(fn, 0). Почему это не гарантирует исполнение через 0мс? Сравни это с Task.Yield() в C#.
2. **Microtask Queue Starvation:** Объясни, как бесконечный цикл промисов (Promise.resolve().then(loop)) может повесить вкладку браузера, не давая сработать кликам мыши (Macrotask). Есть ли аналог этой проблемы в.NET ThreadPool?
3. **Exception Handling:** Сравни распространение исключений в async/await стеке C# и в цепочке Promise JS. Почему "неперехваченный промис" (Unhandled Rejection) может положить Node.js процесс, но игнорируется (часто) в браузере?
4. **Аналогия с Unity:** Сравни JS Event Loop с жизненным циклом кадра в Unity (Update -> Coroutines -> LateUpdate -> Render). Где в этой схеме находятся Microtasks?.14

#### Практические задания (День 2)

**Цель:** Научиться управлять потоком исполнения в однопоточной среде и избегать блокировок UI.

1. **Задание "Event Loop Predictor":**
   * Дана сложная последовательность вложенных setTimeout, Promise.resolve, queueMicrotask и синхронного кода.
   * *Задача:* Расписать порядок вывода в консоль до запуска кода. Объяснить каждое решение (почему микротаск выполнился раньше таймера с 0 задержкой).16
2. **Задание "Non-blocking Calculation":**
   * Реализовать тяжелую вычислительную задачу (например, поиск простых чисел до 10 млн).
   * При синхронном запуске UI "фризится".
   * *Решение:* Разбить задачу на чанки (Chunking) и использовать паттерн "Scheduler" через setTimeout(..., 0) или requestIdleCallback, чтобы давать браузеру "подышать" (отрисовать кадры) между вычислениями. Это прямой аналог распределения нагрузки в Update() на несколько кадров в Unity.
3. **Задание "Async Retry & Timeout":**
   * Написать утилиту, которая пытается загрузить ресурс (Fetch), делая 3 попытки с экспоненциальной задержкой, но прерывает операцию, если общее время превысило 5 секунд. Использовать Promise.race.

### День 3: DOM, Рендеринг и Виртуализация

Для Unity-разработчика DOM (Document Object Model) проще всего представить как граф сцены (Scene Graph), где каждый HTML-тег — это GameObject. Однако DOM — это чрезвычайно тяжелая структура данных. Манипуляции с ней вызывают пересчет геометрии (Reflow) и перерисовку (Repaint), что по стоимости сопоставимо с Draw Calls и Physics Recalculation в Unity.

#### Архитектурный анализ: Scene Graph vs DOM Tree

В Unity изменение Transform.position помечает объект как "грязный" (dirty), и пересчет происходит в конце кадра. В браузере чтение свойства (например, offsetWidth) сразу после записи (width = '100px') форсирует синхронный пересчет макета (Forced Synchronous Layout), что катастрофически снижает FPS.18

Архитектор должен понимать паттерны минимизации обращений к DOM: использование DocumentFragment (аналог оффскрин буфера или инстанцирования префаба перед добавлением в сцену) и виртуализации списков (аналог пулинга объектов в ScrollRect).

#### Темы теории (День 3)

1. **DOM API:** Обход дерева, селекторы (querySelector). Сравнение производительности с GetComponent и FindObject.
2. **Browser Rendering Pipeline:** Parse HTML -> DOM Tree + CSSOM -> Render Tree -> Layout (Reflow) -> Paint -> Composite.
3. **Event Delegation:** Всплытие (Bubbling) и Перехват (Capturing). Почему один лиснер на родителе лучше 1000 лиснеров на детях (оптимизация памяти).
4. **Virtual DOM (Concept):** Почему React быстрее прямых манипуляций (введение в тему следующей недели).

#### Промт для глубокого исследования (Day 3 Theory Generation)

Задание для AI:

Подготовь анализ производительности рендеринга: "DOM Manipulation Patterns: Best Practices for High-Performance UI".

**Точки анализа:**

1. **Reflow/Repaint Triggers:** Составь таблицу действий, вызывающих дорогой пересчет макета. Сравни это с перестроением Canvas в Unity (Canvas.ForceUpdateCanvases()).
2. **DocumentFragment:** Объясни, как использование фрагментов снижает количество перерисовок. Сравни с батчингом (Batching) в графике.
3. **Event Delegation:** Объясни механику диспетчеризации событий в браузере. Как использование target и currentTarget позволяет экономить память, избегая создания тысяч замыканий для обработчиков событий?19
4. **Shadow DOM:** Краткий обзор инкапсуляции стилей и разметки (Web Components).

#### Практические задания (День 3)

**Цель:** Научиться эффективно манипулировать DOM-деревом без просадок FPS.

1. **Задание "High-Frequency Dashboard":**
   * Создать дашборд, который обновляет 1000 ячеек таблицы данными (симуляция) каждые 100мс.
   * *Оптимизация:* Сравнить обновление через innerHTML (плохо) и точечное обновление textContent конкретных нод.
2. **Задание "Drag & Drop Zone":**
   * Реализовать перетаскивание элементов списка на Vanilla JS.
   * Использовать делегирование событий: повесить обработчики dragstart, drop на контейнер, а не на каждый элемент.
3. **Задание "Infinite Scroll (Manual Implementation)":**
   * Реализовать бесконечную прокрутку. Когда скролл доходит до низа, подгружать новые элементы.
   * *Важно:* Использовать IntersectionObserver (современный API) вместо прослушивания события scroll (которое срабатывает слишком часто).

### День 4: Замыкания (Closures) и Паттерны памяти

Замыкания — это "бедная версия объектов" (или объекты — это "бедная версия замыканий"). Для C#-разработчика важно понять, как замыкания удерживают ссылки на переменные, предотвращая их сборку мусором, и как это соотносится с DisplayClass, генерируемым компилятором Roslyn.

#### Архитектурный анализ: Captured Variables Memory

В C#, когда лямбда захватывает переменную, компилятор создает скрытый класс. В JS замыкания работают через лексическое окружение (Lexical Environment), связанное в цепочку (Scope Chain).

Основная проблема ("Gotcha") для разработчиков C# — поведение переменных цикла. В старых версиях C# и при использовании var в JS (до ES6) переменная цикла захватывалась по ссылке, что приводило к тому, что все замыкания видели конечное значение счетчика. Введение let в JS решило эту проблему, создавая новую привязку для каждой итерации, аналогично поведению foreach в современном C#.6

Также критично понимание контекста this. В C# this всегда указывает на экземпляр класса. В JS this зависит от места вызова (call-site), что часто приводит к потере контекста при передаче методов как колбэков.22

#### Темы теории (День 4)

1. **Lexical Scope & Closure:** Механизм сохранения доступа к внешним переменным.
2. **Memory Leaks:** Как замыкания создают циклические ссылки и утечки памяти (особенно в связке с DOM-элементами).
3. **The this Keyword:** 4 правила привязки: Default, Implicit (obj.method()), Explicit (call, apply, bind), new.
4. **Functional Programming Basics:** Функции высшего порядка (map, filter, reduce) как замена LINQ.
5. **Currying & Partial Application:** Паттерны для создания конфигураций функций.

#### Промт для глубокого исследования (Day 4 Theory Generation)

Задание для AI:

Создать отчет "Memory Management & Closures: Deep Dive for.NET Engineers".

**Детали:**

1. **Captured Variables Internals:** Сравни, как аллоцируется память под захваченные переменные в C# (Heap allocation of DisplayClass) и в JS (Context object allocation).
2. **The Loop Problem:** Проанализируй классическую проблему for (var i...) setTimeout(() => log(i)). Объясни, почему let решает её (Block Scoping).
3. **this vs Delegates:** Объясни, почему в C# Action сохраняет Target, а в JS передача obj.method теряет контекст. Как работает bind под капотом (создание обертки)?.24
4. **LINQ vs Array Methods:** Составь таблицу соответствия методов LINQ и методов массива JS (Select -> map, Where -> filter, Aggregate -> reduce).

#### Практические задания (День 4)

**Цель:** Овладеть функциональными паттернами и контролем контекста.

1. **Задание "Closure Counter & Module":**
   * Реализовать паттерн "Модуль" через замыкание (IIFE), скрыв приватную переменную-счетчик и выставив наружу только методы increment и getValue. Это эмуляция private полей.
2. **Задание "Custom Bind & Polyfills":**
   * Написать собственную реализацию метода .bind(), используя apply и замыкания.
   * Это упражнение форсирует понимание того, как работает явная привязка контекста.
3. **Задание "Data Transformation Pipeline":**
   * Дан массив сложных объектов (лога событий).
   * Используя цепочку filter, map, reduce (аналог LINQ), собрать статистику (например, "количество ошибок по типам за последний час").
   * Написать код в функциональном стиле, без циклов for.

### День 5: Реальные паттерны UI и Интеграция с Backend

Завершающий день теории фокусируется на паттернах, критичных для производительности UI (Debounce/Throttle) и взаимодействии с сервером.

#### Архитектурный анализ: Optimization & Networking

Обработка ввода пользователя (например, поиск при вводе) требует паттерна Debounce (откладывание вызова до окончания ввода) или Throttle (ограничение частоты вызовов). В Unity это часто решается через таймеры в Update. В JS это функции высшего порядка, декорирующие обработчики событий.25

Взаимодействие с сетью через Fetch API требует понимания промисов и обработки ошибок. Важный аспект — CORS (Cross-Origin Resource Sharing). Это механизм безопасности браузера, который отсутствует в Unity (или работает иначе), и часто становится камнем преткновения при подключении к API.28

#### Темы теории (День 5)

1. **Debounce & Throttle:** Реализация и сценарии использования (Search Input vs Scroll Handler).27
2. **Fetch API:** Конфигурация запросов, заголовки, JSON.
3. **AbortController:** Паттерн отмены асинхронных операций. Прямой аналог CancellationToken в.NET.
4. **CORS:** Preflight requests (OPTIONS). Почему это происходит и как настроить backend.

#### Промт для глубокого исследования (Day 5 Theory Generation)

Задание для AI:

Написать спецификацию "Advanced Client-Side Patterns".

**Аспекты:**

1. **Debounce/Throttle Internals:** Покажи реализацию этих паттернов "с нуля" на JS. Сравни их влияние на Event Loop и количество макрозадач.
2. **AbortController vs CancellationToken:** Сравни API и механику отмены HTTP-запросов. Как отменить fetch при уходе со страницы или новом вводе пользователя?
3. **Security:** XSS (Cross-Site Scripting) при работе с innerHTML. Почему важно использовать textContent или безопасные методы создания DOM-узлов.30

#### Практические задания (День 5)

**Цель:** Собрать знания воедино, реализовав надежный интерфейс.

1. **Задание "Smart Search Input":**
   * Реализовать поле поиска с Debounce (задержка 500мс).
   * При вводе отправлять запрос на фиктивный API (или реальный backend).
   * *Кейс Race Condition:* Если пользователь ввел "А", запрос ушел, потом быстро ввел "АБ", ушел второй запрос. Если ответ на "А" придет ПОЗЖЕ ответа на "АБ", интерфейс покажет старые данные.
   * *Решение:* Использовать AbortController для отмены предыдущего запроса перед отправкой нового.
2. **Задание "Scroll Spy (Throttle)":**
   * Отслеживать позицию скролла и подсвечивать активный пункт меню.
   * Использовать Throttle, чтобы обработчик срабатывал не чаще раза в 100мс, экономя ресурсы CPU.

## 4. Итоговые задания недели (Capstone Projects)

Эти проекты эмулируют реальные задачи Enterprise-разработки, требующие понимания архитектуры.

### Проект А: Высокопроизводительный Virtual Scroll (Vanilla JS)

Бизнес-задача: Отобразить лог событий из 100,000 записей, полученных с бэкенда, без зависания браузера.

Технический вызов: Рендеринг 100k DOM-элементов приведет к краху вкладки (Memory & Layout trashing).

Решение: Реализовать Virtualization (Windowing).31

1. **Алгоритм:**
   * Создать контейнер с высотой, равной totalItems \* itemHeight (чтобы работал нативный скроллбар).
   * Рассчитать, какие индексы элементов видимы в текущем scrollTop.
   * Рендерить только эти элементы (плюс небольшой буфер сверху/снизу), позиционируя их абсолютно (transform: translateY).
   * При скролле не удалять/создавать DOM-ноды, а обновлять их контент и позицию (DOM Recycling), аналогично пулу объектов в Unity.
2. **Метрики:** Использовать Chrome DevTools Performance tab для доказательства стабильных 60 FPS при скролле.

### Проект Б: MVC Архитектура "ToDo Application" (Без фреймворков)

Бизнес-задача: Создать приложение для управления задачами, готовое к масштабированию.

Технический вызов: Реализовать классический паттерн MVC (Model-View-Controller) на чистом JS, чтобы понять, какие проблемы решают современные фреймворки (React/Angular).30

1. **Model:** Класс TodoModel. Хранит данные, работает с localStorage. Реализует паттерн Observer (события onListChanged), чтобы уведомлять подписчиков.
2. **View:** Класс TodoView. Отвечает *только* за DOM. Создает элементы, навешивает обработчики. Не содержит бизнес-логики. Предоставляет метод bindAddTodo(handler).
3. **Controller:** Класс TodoController. Инициализирует Model и View. Подписывается на события View (клики) и вызывает методы Model. Подписывается на события Model и обновляет View.
4. **Функционал:** Добавление, удаление, редактирование, фильтрация (All/Active/Completed).

## 5. Заключение: Формирование Fullstack-мышления

Пройдя этот недельный интенсив, разработчик получает не просто навык написания скриптов, а глубокое понимание платформы веб-браузера. Ключевые архитектурные сдвиги, которые должны произойти:

1. **От пулов потоков к циклу событий:** Понимание цены блокировки главного потока и умение разбивать задачи.
2. **От статической памяти к динамической:** Осознание работы скрытых классов V8 и управления памятью в замыканиях.
3. **От прямого управления рендерингом к DOM-операциям:** Навыки минимизации Reflow/Repaint через виртуализацию и фрагменты.

Эти компетенции создают фундамент для следующего этапа — изучения React, где многие из этих концепций (Virtual DOM, Synthetic Events, Hooks) станут абстракциями над изученными низкоуровневыми механизмами.

#### Источники

1. Разработка бэкенда: план обучения C#
2. Tips for JavaScript and TypeScript Developers - A tour of C# | Microsoft Learn, дата последнего обращения: декабря 6, 2025, <https://learn.microsoft.com/en-us/dotnet/csharp/tour-of-csharp/tips-for-javascript-developers>
3. 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>
4. C# vs JavaScript: Which Programming Language Is better For Your Needs? - Ideamotive, дата последнего обращения: декабря 6, 2025, <https://www.ideamotive.co/blog/c-sharp-vs-javascript>
5. Closures in C# vs JavaScript — Same But Different | by Gabi Beyo | Medium, дата последнего обращения: декабря 6, 2025, <https://medium.com/@gabi.beyo/closures-in-c-vs-javascript-same-but-different-65cf5e20daf7>
6. Closure on Loop Variables — C# and JavaScript Comparison | by sidecus - Medium, дата последнего обращения: декабря 6, 2025, <https://medium.com/swlh/closure-on-loop-variables-c-and-javascript-comparison-fc230a8bd211>
7. Understanding Async in JavaScript and C# — A Practical Guide with Code - Medium, дата последнего обращения: декабря 6, 2025, <https://medium.com/@joshi.vignesh/understanding-async-in-javascript-and-c-a-practical-guide-with-code-8933289a88c7>
8. Difference between when the callback runs in JavaScript vs C# - Stack Overflow, дата последнего обращения: декабря 6, 2025, <https://stackoverflow.com/questions/68714621/difference-between-when-the-callback-runs-in-javascript-vs-c-sharp>
9. Understanding JavaScript's Event Loop: Microtasks vs. Macrotasks Explained - Medium, дата последнего обращения: декабря 6, 2025, <https://medium.com/@sahusuneel777/understanding-javascripts-event-loop-microtasks-vs-macrotasks-explained-2f4cf58b0836>
10. How the Event Loop Handles Microtasks and Macrotasks - DEV Community, дата последнего обращения: декабря 6, 2025, <https://dev.to/dinhkhai0201/how-the-event-loop-handles-microtasks-and-macrotasks-4hi7>
11. 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>
12. Understanding the Event Loop in JavaScript: Microtasks, Macrotasks, and Asynchronous Execution - DEV Community, дата последнего обращения: декабря 6, 2025, <https://dev.to/nishanthan-k/understanding-the-event-loop-in-javascript-microtasks-macrotasks-and-asynchronous-execution-3037>
13. What is the difference between the event loop in JavaScript and async non-blocking I/O in Node.js? - Stack Overflow, дата последнего обращения: декабря 6, 2025, <https://stackoverflow.com/questions/70201005/what-is-the-difference-between-the-event-loop-in-javascript-and-async-non-blocki>
14. Broadcast vs Events and CoRoutine vs async Task? - Ask - GameDev.tv, дата последнего обращения: декабря 6, 2025, <https://community.gamedev.tv/t/broadcast-vs-events-and-coroutine-vs-async-task/233481>
15. Coroutines OR Async/await? : r/unity - Reddit, дата последнего обращения: декабря 6, 2025, <https://www.reddit.com/r/unity/comments/zr39d4/coroutines_or_asyncawait/>
16. JavaScript Event Loop -- Visualized! - YouTube, дата последнего обращения: декабря 6, 2025, <https://www.youtube.com/watch?v=okkHnAo8GmE>
17. JavaScript Visualized - Event Loop, Web APIs, (Micro)task Queue - Lydia Hallie, дата последнего обращения: декабря 6, 2025, <https://www.lydiahallie.com/blog/event-loop>
18. Manual: Comparison of UI systems in Unity, дата последнего обращения: декабря 6, 2025, <https://docs.unity3d.com/6000.2/Documentation/Manual/UI-system-compare.html>
19. DOM Manipulation in Vanilla JavaScript - NamasteDev Blogs, дата последнего обращения: декабря 6, 2025, <https://namastedev.com/blog/dom-manipulation-in-vanilla-javascript-7/>
20. Mastering DOM Manipulation in Vanilla JavaScript: The Complete Guide - Rajesh Dhiman, дата последнего обращения: декабря 6, 2025, <https://www.rajeshdhiman.in/blog/mastering-dom-manipulation-advanced-javascript>
21. I wrote an article explaining how closures work in .NET. Thought this might be useful. : r/dotnet - Reddit, дата последнего обращения: декабря 6, 2025, <https://www.reddit.com/r/dotnet/comments/170kgis/i_wrote_an_article_explaining_how_closures_work/>
22. this - JavaScript | MDN, дата последнего обращения: декабря 6, 2025, <https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/this>
23. Breaking down the 'this' keyword in Javascript - LoginRadius, дата последнего обращения: декабря 6, 2025, <https://www.loginradius.com/blog/engineering/breaking-down-this-keyword-in-javascript>
24. Detailed Explanation of Variable Capture in Closures - Stack Overflow, дата последнего обращения: декабря 6, 2025, <https://stackoverflow.com/questions/5438307/detailed-explanation-of-variable-capture-in-closures>
25. Debouncing and Throttling with examples javascript - DEV Community, дата последнего обращения: декабря 6, 2025, <https://dev.to/hemantgovekar/debouncing-and-throttling-with-examples-5dgl>
26. Debounce and Throttle in Real Life Scenarios | by Kfir Zuberi | WalkMe Engineering, дата последнего обращения: декабря 6, 2025, <https://medium.com/walkme-engineering/debounce-and-throttle-in-real-life-scenarios-1cc7e2e38c68>
27. 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>
28. Async/Await in C# vs JavaScript: Same Hoodie, Different Superpowers - DEV Community, дата последнего обращения: декабря 6, 2025, <https://dev.to/pranta/asyncawait-in-c-vs-javascript-same-hoodie-different-superpowers-41jo>
29. JavaScript Performance Optimization: Debounce Vs Throttle Explained - Nile Bits, дата последнего обращения: декабря 6, 2025, <https://www.nilebits.com/blog/2024/07/javascript-debounce-vs-throttle/>
30. Writing a TodoMVC App with Modern Vanilla JavaScript - Frontend Masters, дата последнего обращения: декабря 6, 2025, <https://frontendmasters.com/blog/vanilla-javascript-todomvc/>
31. List Virtualization: How to Smoothly Scroll Through 10000 Items | by Ethan Haller - Medium, дата последнего обращения: декабря 6, 2025, <https://medium.com/@ethanhaller02/list-virtualization-how-to-smoothly-scroll-through-10-000-items-cad39bfa7f3e>
32. Virtualization for Large Lists - DEV Community, дата последнего обращения: декабря 6, 2025, <https://dev.to/maurya-sachin/virtualization-for-large-lists-in8>
33. List Virtualization - Patterns.dev, дата последнего обращения: декабря 6, 2025, <https://www.patterns.dev/vanilla/virtual-lists/>
34. Classic Front End MVC with Vanilla Javascript | by Patrick Ackerman | Medium, дата последнего обращения: декабря 6, 2025, <https://medium.com/@patrickackerman/classic-front-end-mvc-with-vanilla-javascript-7eee550bc702>
35. How to Build a Simple MVC App From Scratch in JavaScript - Tania Rascia, дата последнего обращения: декабря 6, 2025, <https://www.taniarascia.com/javascript-mvc-todo-app/>