# Учебный план: Архитектура коллекций и управление памятью в Enterprise.NET (Неделя 4)

## Введение и Методологическое Обоснование

Переход квалифицированного Unity-разработчика в сферу Enterprise-бэкенда требует фундаментальной перестройки ментальной модели работы с данными. В среде GameDev (Unity) основной паттерн взаимодействия с памятью диктуется игровым циклом (Game Loop) и жесткими ограничениями на аллокации в кадре (Zero Allocation in Update). Разработчики привыкают избегать коллекций и LINQ в горячих путях выполнения кода, предпочитая массивы и ручное управление индексами, чтобы предотвратить "фризы" (GC Spikes) во время рендеринга.1

Однако в среде.NET Core Backend (ASP.NET Core) парадигма меняется. Здесь мы оперируем понятиями пропускной способности (Throughput), конкурентного доступа (Concurrency) и масштабируемости. Коллекции становятся не просто контейнерами данных, а фундаментальными строительными блоками архитектуры. Ошибочный выбор между List<T> и LinkedList<T>, непонимание внутреннего устройства Dictionary или неправильная стратегия блокировок в ConcurrentDictionary могут привести к деградации производительности всей системы под нагрузкой.

Четвертая неделя обучения посвящена глубокому анализу (Deep Dive) внутреннего устройства CLR (Common Language Runtime) в контексте структур данных. Мы отойдем от поверхностного использования API и спустимся на уровень управления памятью, кэш-линий процессора и алгоритмической сложности.

Программа рассчитана на 20 часов интенсивного погружения (5 дней по 4 часа), сочетая теоретический анализ исходного кода платформы.NET с практическими задачами, имитирующими реальные высоконагруженные сценарии.

## Глобальный Промт для AI-Ментора (GEM-bot)

Ниже представлен специализированный системный промт, который необходимо использовать для инициализации контекста вашего AI-ассистента (Gemini/GPT) на всю текущую неделю. Это обеспечит релевантность ответов и поддержание необходимого уровня технической глубины.

**System Role:** Ты — Senior.NET Architect и эксперт по Performance Engineering с специализацией на внутреннем устройстве CLR (Common Language Runtime) и High-Load системах. Ты менторишь опытного Unity-разработчика (C# Middle/Senior), который переходит в Enterprise Backend разработку.

**Твои задачи:**

1. **Глубокий технический анализ:** Не давай поверхностных определений. Объясняй работу коллекций на уровне раскладки памяти (Memory Layout), работы Garbage Collector (Gen0/1/2, LOH), стоимости аллокаций и влияния на CPU Cache (Spatial/Temporal Locality).
2. **Сравнительный анализ:** Постоянно проводи параллели между подходами Unity (Single-threaded, Frame-based) и Backend (Multi-threaded, Request-based). Объясняй, почему привычные паттерны Unity (например, избегание foreach) могут быть неактуальны или вредны на сервере, и наоборот.
3. **Архитектурная строгость:** Требуй обоснования выбора каждой структуры данных. Задавай вопросы: "Почему здесь ConcurrentDictionary, а не MemoryCache?", "Как этот код поведет себя при 10k RPS?".
4. **Код-ревью:** Анализируй код с точки зрения потокобезопасности (Thread Safety), ложного разделения (False Sharing) и асимптотической сложности (Big O).

**Контекст недели:** Мы изучаем архитектуру коллекций (Dictionary, LinkedList, Stack/Queue), конкурентные коллекции (System.Collections.Concurrent), неизменяемые коллекции (Immutable/Frozen) и стратегии кэширования (LRU/LFU).

## День 1: Анатомия Hash-Map и Внутреннее устройство Dictionary

В современной бэкенд-разработке словарь (Dictionary<TKey, TValue>) является, пожалуй, самой критически важной структурой данных. Он используется повсеместно: от механизмов маршрутизации HTTP-запросов и хранения сессий до построения индексов в базах данных in-memory. Понимание того, как словарь работает "под капотом", необходимо для предотвращения коллизий, оптимизации потребления памяти и защиты от атак типа Hash Flooding.

### Теоретический модуль [Изучение Hash-Map и Dictionary в .NET день 1](https://docs.google.com/document/d/1BciLOhYbMeJEWPpeK--ioJNDHXq-GWbgpfdErSFoOnk/edit?usp=drive_link)

#### 1. Внутренняя архитектура Dictionary

В отличие от наивных представлений, Dictionary в.NET не хранит данные как простой список пар KeyValuePair. Реализация базируется на двух основных массивах для обеспечения производительности O(1) и эффективного использования памяти 2:

1. **int buckets**: Массив целых чисел, представляющий собой "корзины" хеш-таблицы. Индекс в этом массиве вычисляется на основе хеш-кода ключа. Значение в ячейке указывает на индекс первого элемента в массиве entries.
2. **Entry entries**: Массив структур, содержащий фактические данные. Использование массива структур (вместо классов) критично для производительности, так как это улучшает локальность данных (Data Locality) и снижает нагрузку на GC.3

Структура Entry выглядит (упрощенно) следующим образом:

C#

private struct Entry {  
 public int hashCode; // Кэшированный хеш (для ускорения ресайза и проверок)  
 public int next; // Индекс следующего элемента в цепочке коллизий (-1, если конец)  
 public TKey key; // Ключ  
 public TValue value; // Значение  
}

Такая организация памяти позволяет.NET Dictionary быть чрезвычайно быстрым. При добавлении элемента, если соответствующий bucket уже занят, происходит коллизия..NET разрешает коллизии методом **цепочек (Chaining)**, но, в отличие от классической реализации со связным списком узлов в куче, цепочка формируется внутри того же массива entries через индексы next. Это устраняет необходимость в дополнительных аллокациях объектов-узлов при вставке, что является существенным преимуществом перед реализациями в других языках (например, Java до версий с Red-Black Tree).3

#### 2. Стратегия ресайзинга (Resizing) и простые числа

Когда словарь заполняется (достигается capacity), происходит перераспределение памяти (Resize). Критически важным аспектом реализации.NET является выбор нового размера массива. В то время как многие хеш-таблицы используют степени двойки (Power of 2) для быстрого вычисления индекса через побитовое AND (например, hash & (size - 1)),.NET традиционно использует **простые числа (Prime Numbers)**.5

| **Стратегия** | **Механизм вычисления индекса** | **Преимущества** | **Недостатки** |
| --- | --- | --- | --- |
| **Power of 2** | index = hashCode & (size - 1) | Очень быстрая операция (1 такт CPU). | Требует качественной функции распределения хеша (Avalanche effect). Плохой хеш приводит к множеству коллизий. |
| **Prime Number** (.NET) | index = hashCode % size | Снижает вероятность коллизий даже при "слабых" хеш-функциях, так как простое число не имеет общих делителей с паттернами в данных. | Операция деления/остатка (DIV/IDIV) значительно медленнее побитовых операций на уровне CPU (может занимать десятки тактов).5 |

Этот выбор в пользу простых чисел делает Dictionary в C# более устойчивым к плохим пользовательским реализациям GetHashCode(), но ценой некоторого снижения "сырой" производительности вставки/поиска.6

#### 3. Hash Flooding и безопасность

Понимание устройства словаря открывает глаза на уязвимость типа "Hash Flooding DoS". Если злоумышленник знает алгоритм хеширования и размер таблицы, он может сгенерировать тысячи ключей, которые попадут в один и тот же bucket. Это превратит словарь в связный список, деградируя сложность операций с O(1) до O(n), что приведет к отказу в обслуживании (CPU 100%). Для защиты от этого в.NET внедрена рандомизация хеширования строк (Randomized String Hashing), которая включается при обнаружении большого количества коллизий.7

### Промт для глубокого исследования (Theory Generation)

"Проведи детальный анализ исходного кода класса Dictionary<TKey, TValue> в последней версии.NET (GitHub dotnet/runtime). Опиши алгоритм метода FindEntry. Как именно используется массив buckets для навигации по массиву entries? Сравни стратегию ресайзинга в.NET Framework 4.8 (использование простых чисел) и оптимизации в.NET 8 (использование fastmod или других техник). Объясни, почему удаление элемента (Remove) не приводит к физическому сжатию массива entries, а создает 'цепочку свободных слотов' (freeList). В чем опасность утечки памяти при частом добавлении и удалении элементов в долгоживущий словарь?"

### Практическое задание: "Zero-Allocation Hash Map"

**Цель:** Создать упрощенную, но высокопроизводительную хеш-таблицу, которая минимизирует нагрузку на GC, используя структуры.

1. **Реализация Core:**
   * Создайте структуру Entry<K, V>, содержащую HashCode, Next (индекс), Key, Value.
   * Реализуйте класс LightweightDictionary<K, V>, использующий int buckets и Entry entries.
   * Реализуйте метод Add с разрешением коллизий через индексы (без создания объектов узлов).
2. **Бенчмаркинг:**
   * Используя BenchmarkDotNet, сравните производительность вашей реализации и стандартного Dictionary на операциях Add и TryGetValue (100,000 элементов).
   * **Ключевое требование:** Ваша реализация TryGetValue должна показывать **0 Bytes Allocated** в отчете бенчмарка.
3. **Стресс-тест хеширования:**
   * Создайте пользовательский тип ключа BadHashKey, у которого GetHashCode() всегда возвращает константу (например, return 42;).
   * Замерьте падение производительности при вставке 10,000 элементов. Это продемонстрирует вырождение хеш-таблицы в связный список.

## День 2: Иерархия памяти, Локальность и Списки

Unity-разработчики часто стоят перед выбором: List или LinkedList. В теории алгоритмов (Big O Notation) вставка в середину связного списка — это O(1), а в массив — O(n). Однако на практике в бэкенде ситуация часто обратная из-за архитектуры современных процессоров. Этот день посвящен тому, как "железо" влияет на выбор коллекций.

### Теоретический модуль [Теория обучения: Иерархия памяти и коллекции день 2](https://docs.google.com/document/d/1qx2B4PpzGtKYT-4aUEW9AXxRkPcnS6ocWqsRsozkJnc/edit?usp=drive_link)

#### 1. CPU Caches и Spatial Locality

Процессор не работает с оперативной памятью (RAM) напрямую побайтово. Данные загружаются блоками, называемыми **Cache Lines** (обычно 64 байта), в L1/L2/L3 кэши.

* **List (Array-backed):** Массивы располагаются в памяти непрерывно (Contiguous memory). При чтении первого элемента массива, процессор загружает в кэш сразу пачку соседних элементов. Последующий доступ к ним происходит мгновенно (за наносекунды). Это называется высокой пространственной локальностью (Spatial Locality).9
* **LinkedList:** Узлы связного списка — это отдельные объекты в куче (Heap). Они могут быть разбросаны по памяти хаотично (фрагментация). Переход по ссылке node.Next часто ведет в область памяти, которой нет в кэше, вызывая **Cache Miss**. Процессор вынужден простаивать сотни тактов, ожидая подгрузки данных из RAM.11

#### 2. Реальная производительность

Исследования и бенчмарки показывают, что итерация по LinkedList может быть в десятки раз медленнее, чем по List, именно из-за Cache Misses. Более того, даже операция вставки в середину списка (которая теоретически O(1)) проигрывает массиву (List.Insert - O(n)) на малых и средних объемах данных (до тысяч элементов). Причина в том, что копирование памяти в массиве (memmove) — это крайне оптимизированная операция, использующая SIMD-инструкции процессора, в то время как траверс списка для поиска места вставки упирается в латентность памяти.12

#### 3. Span и Memory

Современный.NET предоставляет инструменты для безопасной работы с памятью без аллокаций. Span<T> — это ref struct, обеспечивающая типобезопасное представление непрерывной области памяти (будь то массив, стек или нативный хип). Использование Span позволяет работать с подмножествами коллекций (Slicing) без создания копий, что критически важно для высоконагруженных парсеров и обработки I/O.13

### Промт для глубокого исследования (Theory Generation)

"Подготовь технический отчет о влиянии 'CPU Cache Misses' на производительность коллекций в.NET. Используй данные из статьи 'Performance of Array vs. Linked List on Modern Computers' (DZone/StackOverflow). Объясни, почему 'Pointer Chasing' в LinkedList убивает производительность ILP (Instruction Level Parallelism). Как ref struct и Span<T> позволяют избежать аллокаций в куче? Приведи пример кода, где замена Substring на AsSpan().Slice дает прирост производительности."

### Практическое задание: "Анализатор Cache Misses"

**Цель:** Эмпирически доказать преимущество массивов над связными списками в сценариях последовательного доступа.

1. **Бенчмарк итерации:**
   * Сгенерируйте List<int> и LinkedList<int> размером в 1,000,000 элементов.
   * Напишите бенчмарк суммирования всех элементов. Сравните время выполнения.
2. **Бенчмарк вставки:**
   * Сравните List.Insert(count/2, item) и LinkedList.AddAfter(node, item) (предполагая, что узел уже найден) и сценарий, где узел нужно найти.
   * Определите пороговое значение N (размер коллекции), при котором LinkedList начинает выигрывать у List (если вообще начинает).
3. **Работа со Span:**
   * Напишите парсер простой строки (например, "User:John;Age:30;Role:Admin"), который извлекает значения.
   * Версия 1: Использовать String.Split (создает массивы строк).
   * Версия 2: Использовать ReadOnlySpan<char> и метод Slice (Zero Allocation).

## День 3: Многопоточность и Конкурентные коллекции

В Unity многопоточность часто ограничивается Coroutines или Job System, но основной API движка однопоточен. В бэкенде параллелизм — это норма. Использование обычного Dictionary в многопоточной среде приведет к непредсказуемым ошибкам или крашам. Однако использование lock на каждую операцию убивает производительность. Решение — специализированные конкурентные коллекции.

### Теоретический модуль [Создание документа по многопоточности .NET день 3](https://docs.google.com/document/d/13lq11UIzDe0LgX5E9eFMnutCr60adi1WU-At3kMNpBI/edit?usp=drive_link)

#### 1. ConcurrentDictionary: Fine-Grained Locking и Lock-Free Reads

ConcurrentDictionary — это шедевр инженерной мысли.NET. Он решает проблему глобальной блокировки, используя **Fine-Grained Locking** (мелкогранулярные блокировки).

* Словарь разделен на несколько сегментов (по умолчанию определяется количеством ядер CPU). Каждый сегмент имеет свой независимый замок (lock). Запись в один сегмент не блокирует запись в другие.15
* **Lock-Free Reads:** Самое важное свойство — чтение (TryGetValue, индексатор) выполняется **без блокировок**. Это достигается за счет использования инструкций Volatile и Memory Barriers, гарантирующих, что поток увидит актуальное состояние памяти без захвата монитора.17 Это делает ConcurrentDictionary идеальным для кэшей (Read-Heavy workloads).

#### 2. Паттерн Producer-Consumer: BlockingCollection vs Channels

Для организации асинхронной обработки задач (очереди сообщений) используются специализированные структуры:

* **BlockingCollection:** Классическая реализация. Оборачивает ConcurrentQueue. Потоки-потребители блокируются (уходят в ожидание), если очередь пуста, и просыпаются при появлении данных. Это синхронный, блокирующий подход.19
* **System.Threading.Channels:** Современная (начиная с.NET Core 3.0), высокопроизводительная альтернатива. Каналы поддерживают async/await из коробки. ChannelReader.ReadAsync возвращает ValueTask, что позволяет потоку не блокироваться, а возвращаться в ThreadPool во время ожидания данных. Это значительно эффективнее для I/O-bound задач и масштабируемости.21

#### 3. Проблема False Sharing

При высоконагруженной параллельной работе может возникнуть эффект "ложного разделения" (False Sharing). Если два потока часто пишут в разные переменные, которые случайно оказались в одной Cache Line (64 байта), ядра процессора будут постоянно инвалидировать кэши друг друга (Cache Coherency Protocol). Это приводит к резкому падению производительности, несмотря на отсутствие явных блокировок. Решение — добавление "пустот" (padding) между полями.23

### Промт для глубокого исследования (Theory Generation)

"Сравни архитектуру BlockingCollection и System.Threading.Channels. Почему Channels считаются более подходящими для современной асинхронной модели.NET? Объясни понятие 'Backpressure' в контексте Bounded Channels. Как ConcurrentDictionary реализует 'Lock-Free Reads' с использованием Volatile.Read? Напиши пример структуры с Padding-ом для предотвращения False Sharing."

### Практическое задание: "Сервер обработки заказов"

**Цель:** Реализовать масштабируемую систему обработки задач по паттерну Producer-Consumer, избегая блокировок потоков.

1. **Ingestion Layer (Producer):**
   * Создайте пул задач (Producers), которые имитируют поступление заказов (Order).
   * Используйте Channel.CreateBounded<Order>(capacity) для ограничения очереди. Продемонстрируйте, как Producer "ждет" (асинхронно), если очередь переполнена (Backpressure).
2. **Processing Layer (Consumer):**
   * Запустите несколько воркеров (Consumers), читающих из канала (Reader.ReadAllAsync()).
   * Воркеры должны обрабатывать заказ (имитация задержки Task.Delay).
3. **State Management:**
   * Используйте ConcurrentDictionary<Guid, OrderStatus> для отслеживания статуса заказов.
   * Реализуйте метод получения статуса, который читает из словаря (Lock-Free) параллельно с записью.
4. **Graceful Shutdown:**
   * Реализуйте корректную остановку сервера: Producer вызывает Writer.Complete(), Consumers дочитывают оставшиеся сообщения и завершают работу.

## День 4: Алгоритмы кэширования и Управление вытеснением

Кэширование — самый эффективный способ повышения производительности, но, как гласит известная цитата, "инвалидация кэша — одна из двух самых сложных проблем в CS". Без правильной стратегии вытеснения (Eviction Policy) кэш превращается в утечку памяти.

### Теоретический модуль [теория недели 4 дня 4](https://docs.google.com/document/d/1OVRbLH7Zhy06JiO4v-75LPFoZzNmvmAVTW1NMW39-Y0/edit?usp=drive_link)

#### 1. Алгоритм LRU (Least Recently Used)

Классический LRU-кэш требует быстрого доступа (O(1)) и быстрого обновления порядка использования. Это достигается комбинацией двух структур 25:

* Dictionary<Key, Node>: Для мгновенного поиска узла по ключу.
* Doubly LinkedList<Node>: Для хранения порядка. При доступе к элементу он перемещается в "голову" списка. При переполнении удаляется элемент из "хвоста" (самый давно неиспользованный).

Стандартный LinkedList в.NET плох для этой задачи, так как он аллоцирует новый объект LinkedListNode при каждом добавлении. Для High-Performance решений часто пишут кастомный двусвязный список поверх массива (аналогично внутренностям Dictionary), чтобы снизить давление на GC.26

#### 2. IMemoryCache и MemoryCache

ASP.NET Core предоставляет стандартную реализацию IMemoryCache (из Microsoft.Extensions.Caching.Memory).

* Она построена поверх ConcurrentDictionary.
* Поддерживает **Sliding Expiration** (продление жизни при доступе) и **Absolute Expiration**.
* **Важный нюанс:** Очистка просроченных элементов (Expiration Scan) происходит не по таймеру (это было бы дорого), а лениво при операциях доступа или при срабатывании триггеров. Это может привести к тому, что "мертвые" объекты живут дольше ожидаемого, занимая память.27
* MemoryCache уведомляет о удалении элементов через PostEvictionDelegate, что позволяет реализовать сложную логику освобождения ресурсов.28

#### 3. Bloom Filters

Это вероятностная структура данных, используемая для сверхбыстрой проверки принадлежности элемента множеству. Она очень компактна (битовый массив). Блум-фильтр может сказать "элемента точно нет" или "элемент, возможно, есть".

* **Use Case:** Защита базы данных от запросов несуществующих ключей (Cache Penetration). Перед тем как идти в БД (если в кэше пусто), проверяем Блум-фильтр. Если он говорит "нет", мы сразу возвращаем 404, не нагружая БД.29

### Промт для глубокого исследования (Theory Generation)

"Разработай архитектуру 'Thread-Safe LRU Cache'. Как обеспечить атомарность операций при перемещении узла в LinkedList и обновлении Dictionary? Сравни использование глобального lock (coarse-grained locking) и ReaderWriterLockSlim для этой задачи. В чем преимущества и недостатки IMemoryCache по сравнению с самописным LRU? Как работают 'Bloom Filters' и какая математика стоит за расчетом вероятности ложноположительного срабатывания?"

### Практическое задание: "Реализация Thread-Safe LRU Cache"

**Цель:** Написать собственный механизм кэширования, понимая принципы синхронизации сложных составных структур.

1. **Структура данных:**
   * Реализуйте класс LRUCache<K, V> с фиксированной емкостью (capacity).
   * Используйте Dictionary для хранения данных и кастомный двусвязный список (класс Node { Prev, Next, Value }) для отслеживания порядка.
2. **Синхронизация:**
   * Оберните операции Get и Put в блокировки. Используйте object lock для простоты или ReaderWriterLockSlim, если хотите оптимизировать чтение.
   * **Логика Put:** Если ключ существует -> обновить значение, переместить в голову. Если нет -> добавить в голову. Если переполнение -> удалить хвост, удалить из словаря.
   * **Логика Get:** Если существует -> переместить в голову, вернуть значение.
3. **Тестирование конкурентности:**
   * Напишите тест, запускающий 10 параллельных потоков. Одни постоянно пишут новые ключи, другие читают. Убедитесь, что размер кэша никогда не превышает capacity и список не разрывается (целостность ссылок Prev/Next).

## День 5: Modern.NET (.NET 8) и Продвинутые техники

Развитие платформы.NET идет стремительно. То, что было актуально в.NET Framework 4.8, может быть устаревшим в.NET 8. Этот день посвящен новейшим инструментам оптимизации.

### Теоретический модуль [Анализ Архитектуры Коллекций и Кэширования день 5](https://docs.google.com/document/d/1_UeA97sxKKZnBH2PyABDpqUN1ekyvEirdIywnztjugY/edit?usp=drive_link)

#### 1. Frozen Collections (.NET 8)

В.NET 8 представлен namespace System.Collections.Frozen. Это коллекции (FrozenDictionary, FrozenSet), которые являются **неизменяемыми** после создания, но чрезвычайно оптимизированы для **чтения**.

* При создании (ToFrozenDictionary()) происходит анализ ключей и выбор идеальной хеш-функции и структуры бакетов специально под этот набор данных. Это делает создание медленным, но поиск (Lookup) — максимально быстрым.
* **Use Case:** Длительно живущие конфигурации, справочники, таблицы маршрутизации, которые загружаются один раз на старте приложения.31

#### 2. PriorityQueue (.NET 6)

До.NET 6 разработчикам приходилось реализовывать "кучи" (Heaps) вручную. Теперь доступен класс PriorityQueue<TElement, TPriority>, реализующий структуру **Min-Heap** поверх массива.

* Сложность вставки и удаления минимума — O(log n).
* **Use Case:** Планировщики задач, алгоритмы поиска пути (Dijkstra), Rate Limiting (очистка устаревших запросов), обработка сообщений с приоритетами.33

#### 3. Object Pooling (ArrayPool)

Аллокация больших массивов (>= 85 Кб) отправляет их в LOH (Large Object Heap), который собирается редко и ведет к фрагментации. Для решения этой проблемы используется паттерн **Object Pool**.

* ArrayPool<T>.Shared: Позволяет брать массив в аренду (Rent) и возвращать его (Return). Это критически важно для буферов при чтении из сокетов или файлов, позволяя переиспользовать память миллионы раз без нагрузки на GC.14
* ConcurrentBag<T> часто используется как основа для реализации пулов объектов, так как он оптимизирован для сценария, когда поток забирает и возвращает объект локально (Thread Local Storage), минимизируя конкуренцию.37

### Промт для глубокого исследования (Theory Generation)

"Проведи бенчмарк-сравнение Dictionary, ImmutableDictionary и FrozenDictionary (в.NET 8) на скорость чтения (TryGetValue). Почему Frozen быстрее? В каких ситуациях использование ArrayPool может привести к 'Memory Leak' или corruption данных (use-after-return)? Как PriorityQueue реализована внутри (Array-backed Binary Heap) и как работает операция Enqueue/Dequeue?"

### Практическое задание: "Rate Limiter и Object Pool"

1. **Priority Queue Task Scheduler:**
   * Создайте планировщик задач, где каждая задача имеет приоритет (Critical, High, Low).
   * Используйте PriorityQueue<TaskItem, int>.
   * Воркер должен всегда забирать задачу с наивысшим приоритетом (наименьшим значением int).
2. **Оптимизация LRU с пулом:**
   * Вернитесь к коду LRU Cache из Дня 4.
   * Вместо создания новых объектов Node (new Node()) при каждой вставке, реализуйте простой пул узлов (ConcurrentBag<Node>).
   * При удалении элемента из кэша не выбрасывайте узел, а возвращайте его в пул. При добавлении — берите из пула.
   * Измерьте снижение аллокаций памяти.

## Итоговые задания недели (Real-World Projects)

Эти комплексные задания предназначены для консолидации знаний и имитируют реальные задачи при разработке микросервисов.

### Проект А: Распределенный Rate Limiter (Sliding Window)

Контекст: API Gateway должен ограничивать количество запросов от одного IP (например, 100 запросов в минуту). Реализовать алгоритм Sliding Window Log.

Архитектура:

1. **Хранилище:** ConcurrentDictionary<IPAddress, Queue<DateTime>>. Ключ — IP, значение — очередь временных меток запросов.
2. **Алгоритм:** При новом запросе добавлять текущее время в очередь. Затем удалять из начала очереди все метки, которые старше окна (например, 1 минута). Если размер очереди < Limit -> пропустить, иначе -> 429 Too Many Requests.
3. Оптимизация: Поскольку Queue не потокобезопасна, необходимо синхронизировать доступ к значению словаря. Подумайте, как минимизировать блокировку (например, использовать ConcurrentQueue или lock только на конкретный IP).  
   Ссылки: 39

### Проект Б: In-Memory Data Grid (Mini-Redis)

Контекст: Разработать микро-сервис для кэширования данных по TCP.

Требования:

1. **Протокол:** Принимать текстовые команды: SET key value ttl\_sec, GET key.
2. **Core:** Использовать ConcurrentDictionary для хранения данных.
3. **TTL (Time To Live):**
   * Реализовать "ленивую" экспирацию: при GET, если срок действия истек — удалять и возвращать null.
   * Реализовать "активную" экспирацию: фоновая задача (используя PriorityQueue), которая хранит ключи, отсортированные по времени истечения, и удаляет их по таймеру.
4. **Static Data:** Используйте FrozenDictionary для хранения списка поддерживаемых команд и обработчиков, так как этот список неизменен.

#### Источники

1. Разработка бэкенда: план обучения C#
2. Anatomy of the .NET dictionary - Steve Dunn, дата последнего обращения: декабря 3, 2025, <https://dunnhq.com/posts/2024/anatomy-of-the-dotnet-dictionary/>
3. Back to basics: Dictionary part 2, .NET implementation - Mark Vincze, дата последнего обращения: декабря 3, 2025, <https://blog.markvincze.com/back-to-basics-dictionary-part-2-net-implementation/>
4. What type of collision resolution is chosen for HashTable/Dictionary implementation in .net?, дата последнего обращения: декабря 3, 2025, <https://stackoverflow.com/questions/7444765/what-type-of-collision-resolution-is-chosen-for-hashtable-dictionary-implementat>
5. Why .Net dictionaries resize to prime numbers? - Stack Overflow, дата последнего обращения: декабря 3, 2025, <https://stackoverflow.com/questions/4638520/why-net-dictionaries-resize-to-prime-numbers>
6. How Dictionary works in .NET - Coding Bolt, дата последнего обращения: декабря 3, 2025, <https://codingbolt.net/2023/09/25/how-dictionary-works-in-net/>
7. Ensuring Data Integrity with Hash Codes - .NET - Microsoft Learn, дата последнего обращения: декабря 3, 2025, <https://learn.microsoft.com/en-us/dotnet/standard/security/ensuring-data-integrity-with-hash-codes>
8. Why does cache locality matter for array performance? - Stack Overflow, дата последнего обращения: декабря 3, 2025, <https://stackoverflow.com/questions/12065774/why-does-cache-locality-matter-for-array-performance>
9. Why Arrays have better cache locality than Linked list? - GeeksforGeeks, дата последнего обращения: декабря 3, 2025, <https://www.geeksforgeeks.org/dsa/why-arrays-have-better-cache-locality-than-linked-list/>
10. CPU Cache disadvantages of using linked lists in C - Stack Overflow, дата последнего обращения: декабря 3, 2025, <https://stackoverflow.com/questions/40071635/cpu-cache-disadvantages-of-using-linked-lists-in-c>
11. c# - Why is a LinkedList Generally Slower than a List? - Stack Overflow, дата последнего обращения: декабря 3, 2025, <https://stackoverflow.com/questions/5983059/why-is-a-linkedlist-generally-slower-than-a-list>
12. Performance optimisations. Classes vs Structs vs ArrayPool Structs // Nikolay Krondev blog, дата последнего обращения: декабря 3, 2025, <https://www.krondev.net/posts/performance-part1-structs/>
13. The BIG performance difference between ArrayPools in .NET | by Eugene Peshkov | Medium, дата последнего обращения: декабря 3, 2025, <https://medium.com/@epeshk/the-big-performance-difference-between-arraypools-in-net-b25c9fc5e31d>
14. Best Practices for Using ConcurrentDictionary - Eli Arbel, дата последнего обращения: декабря 3, 2025, <https://arbel.net/2013/02/03/best-practices-for-using-concurrentdictionary/>
15. Understanding of ConcurrentDictionary in .NET | by Rasul - Medium, дата последнего обращения: декабря 3, 2025, <https://resulhsn.medium.com/understanding-of-concurrentdictionary-in-net-3434105ba371>
16. ConcurrentDictionary
17. c# - ConcurrentDictionary Object - Reading and writing via different threads - Stack Overflow, дата последнего обращения: декабря 3, 2025, <https://stackoverflow.com/questions/11969570/concurrentdictionary-object-reading-and-writing-via-different-threads>
18. BlockingCollection Overview - .NET - Microsoft Learn, дата последнего обращения: декабря 3, 2025, <https://learn.microsoft.com/en-us/dotnet/standard/collections/thread-safe/blockingcollection-overview>
19. BlockingCollection and IProducerConsumerCollection - Simple Thread, дата последнего обращения: декабря 3, 2025, <https://www.simplethread.com/blockingcollection-and-iproducerconsumercollection/>
20. BlockingCollection vs. Channels in C#: What's the Difference? | by erhan355 - Medium, дата последнего обращения: декабря 3, 2025, <https://medium.com/@erhan355/blockingcollection-vs-channels-in-c-whats-the-difference-8b742fc332f4>
21. Asynchronous Producer Consumer Pattern in .NET (C#) - DotNetCurry.com, дата последнего обращения: декабря 3, 2025, <https://www.dotnetcurry.com/dotnetcore/1509/async-dotnetcore-pattern>
22. Thread-Safe collections - .NET | Microsoft Learn, дата последнего обращения: декабря 3, 2025, <https://learn.microsoft.com/en-us/dotnet/standard/collections/thread-safe/>
23. .NET Matters: False Sharing | Microsoft Learn, дата последнего обращения: декабря 3, 2025, <https://learn.microsoft.com/en-us/archive/msdn-magazine/2008/october/net-matters-false-sharing>
24. LRU Cache Data Structure | Interview Cake, дата последнего обращения: декабря 3, 2025, <https://www.interviewcake.com/concept/java/lru-cache>
25. Exploring C# LinkedLists via LRU Caches (1 of 3) - Here's the Codez - SoftWx, дата последнего обращения: декабря 3, 2025, <http://blog.softwx.net/2012/06/exploring-linkedlists-via-lru-caches.html>
26. Caching in .NET - Microsoft Learn, дата последнего обращения: декабря 3, 2025, <https://learn.microsoft.com/en-us/dotnet/core/extensions/caching>
27. Cache in-memory in ASP.NET Core - Microsoft Learn, дата последнего обращения: декабря 3, 2025, <https://learn.microsoft.com/en-us/aspnet/core/performance/caching/memory?view=aspnetcore-10.0>
28. Bloom filters: the niche trick behind a 16× faster API : r/programming - Reddit, дата последнего обращения: декабря 3, 2025, <https://www.reddit.com/r/programming/comments/1oyl5k1/bloom_filters_the_niche_trick_behind_a_16_faster/>
29. What is the advantage to using Bloom filters? - Stack Overflow, дата последнего обращения: декабря 3, 2025, <https://stackoverflow.com/questions/4282375/what-is-the-advantage-to-using-bloom-filters>
30. NET 8 — Frozen Collections - Henrique Siebert Domareski, дата последнего обращения: декабря 3, 2025, <https://henriquesd.medium.com/net-8-frozen-collections-404c1d7c5240>
31. Immutable Dictionary and Frozen Dictionary in .NET - DEV Community, дата последнего обращения: декабря 3, 2025, <https://dev.to/moh_moh701/immutable-dictionary-and-frozen-dictionary-in-net-2978>
32. PriorityQueue
33. C# PriorityQueue - GeeksforGeeks, дата последнего обращения: декабря 3, 2025, <https://www.geeksforgeeks.org/c-sharp/c-priorityqueue/>
34. Priority queue in .Net [closed] - Stack Overflow, дата последнего обращения: декабря 3, 2025, <https://stackoverflow.com/questions/102398/priority-queue-in-net>
35. Pooling large arrays with ArrayPool - Adam Sitnik, дата последнего обращения: декабря 3, 2025, <https://adamsitnik.com/Array-Pool/>
36. C# Object Pooling Pattern implementation - Stack Overflow, дата последнего обращения: декабря 3, 2025, <https://stackoverflow.com/questions/2510975/c-sharp-object-pooling-pattern-implementation>
37. Is a ConcurrentBag still the proper choice for backing an Object Pool if you know the access pattern will be unfavorable? - Stack Overflow, дата последнего обращения: декабря 3, 2025, <https://stackoverflow.com/questions/28591492/is-a-concurrentbag-still-the-proper-choice-for-backing-an-object-pool-if-you-kno>
38. Rate limiting middleware in ASP.NET Core - Microsoft Learn, дата последнего обращения: декабря 3, 2025, <https://learn.microsoft.com/en-us/aspnet/core/performance/rate-limit?view=aspnetcore-10.0>
39. Sliding Window Rate Limiting - Design and Implementation - Arpit Bhayani, дата последнего обращения: декабря 3, 2025, <https://arpitbhayani.me/blogs/sliding-window-ratelimiter/>