# Архитектурный анализ: Анатомия Hash-Map и управление памятью в Enterprise.NET

Учебный модуль: Неделя 4, День 1

Тип документа: Теоретическое руководство (Deep Dive)

Целевая аудитория: Unity Developers transitioning to High-Load Backend

## Введение: Смена парадигмы от игрового цикла к конвейеру запросов

Переход от разработки на Unity к архитектуре корпоративных бэкендов на.NET Core требует не просто изучения новых API, но и фундаментального сдвига в ментальной модели управления памятью и вычислительными ресурсами. В среде GameDev (Unity) инженерное мышление формируется вокруг «Игрового цикла» (Game Loop) — жесткого временного бюджета (обычно 16.6 мс для 60 FPS), в рамках которого код должен выполниться без задержек. Это порождает паттерны, направленные на минимизацию аллокаций в каждом кадре («Zero Allocation in Update»), избегание коллекций в горячих путях и предпочтение ручного управления массивами для предотвращения пауз Garbage Collector (GC Spikes).

В среде Enterprise Backend (.NET 8/9, ASP.NET Core) парадигма меняется кардинально. Здесь мы оперируем не кадрами, а **запросами** (Requests). Метрикой успеха становится не стабильный FPS, а **Пропускная способность** (Throughput — RPS) и **Латентность P99** (время ответа для 99% пользователей). В высоконагруженной системе, обрабатывающей 50,000 запросов в секунду, каждая структура данных подвергается экстремальному давлению. Неэффективный поиск в словаре, лишнее копирование памяти или неправильная стратегия хеширования, умноженные на тысячи параллельных потоков, могут привести к исчерпанию пула потоков (ThreadPool Starvation) и каскадному отказу системы.

Данный документ представляет собой исчерпывающий технический анализ внутренней архитектуры System.Collections.Generic.Dictionary<TKey, TValue> — краеугольного камня любой бэкенд-системы. Мы отойдем от поверхностного использования методов Add и TryGetValue и спустимся на уровень организации памяти CLR (Common Language Runtime), работы процессорного кэша и алгоритмической оптимизации в последних версиях.NET.

## 1. Теоретические основы и проблема отображения

Словарь в.NET является реализацией абстрактного типа данных **Hash Table** (хеш-таблица). Его основная цель — обеспечить выполнение операций вставки, удаления и поиска за константное время $O(1)$ в среднем случае.2

### 1.1 Фундаментальная проблема адресации

Задача любой хеш-таблицы — отобразить бесконечное (или очень большое) множество возможных ключей $U$ (universe of keys) в конечное, ограниченное адресное пространство памяти — массив индексов размером $M$.

$$h(k): U \rightarrow \{0, 1,..., M-1\}$$

Этот процесс в реализации Microsoft.NET происходит в два этапа, понимание которых критично для производительности:

1. **Генерация хеша (Hashing):** Метод GetHashCode() преобразует объект ключа в 32-битное знаковое целое число (int). Это «сырой» хеш, который теоретически может принимать $2^{32}$ значений.
2. **Редукция (Reduction):** Внутренний механизм словаря преобразует этот 32-битный int в валидный индекс массива (bucket index), который должен находиться в диапазоне $[0, Capacity - 1]$.

### 1.2 Неизбежность коллизий и принцип Дирихле

Поскольку размер массива $M$ всегда меньше количества возможных вариантов ключей (например, количество возможных строк бесконечно), согласно принципу Дирихле (Pigeonhole Principle), неизбежна ситуация, когда два разных ключа $k\_1$ и $k\_2$ отображаются в одну и ту же ячейку массива:

$$h(k\_1) = h(k\_2), \text{ где } k\_1 \neq k\_2$$

Производительность словаря в бэкенде напрямую зависит от того, насколько эффективно решаются эти коллизии. В отличие от наивных реализаций, которые могут использовать открытую адресацию (Open Addressing),.NET использует метод **цепочек (Chaining)**, но с уникальной оптимизацией по памяти, которую мы разберем далее.1

## 2. Внутренняя архитектура памяти: Buckets и Entries

Ключевая ошибка Unity-разработчиков при переходе на бэкенд — представление о Словаре как о списке объектов KeyValuePair, разбросанных по куче (Managed Heap). Реальность гораздо сложнее и эффективнее. Для обеспечения локальности данных (Data Locality) и снижения нагрузки на GC, Dictionary<TKey, TValue> использует **два параллельных массива**.1

Взглянем на упрощенное представление полей класса (основано на исходном коде CoreCLR):

C#

public class Dictionary<TKey, TValue>  
{  
 private int \_buckets; // Индексная таблица  
 private Entry \_entries; // Хранилище данных  
 private int \_count; // Текущее количество элементов  
 private int \_freeList; // Голова списка свободных слотов  
 private int \_freeCount; // Количество свободных слотов  
 private int \_version; // Версионность для защиты итераторов  
 //...  
}

### 2.1 Массив \_buckets: Индексная карта

Массив \_buckets — это массив целых чисел (int). Его размер определяет текущую **емкость** (Capacity) хеш-таблицы.

* **Роль:** Служит точкой входа для поиска. Индекс в этом массиве вычисляется на основе хеш-кода ключа.
* **Значение:** Каждая ячейка хранит **индекс** первого элемента цепочки коллизий в массиве \_entries.
* **Семантика:** Значение -1 (или 0 в новейших версиях с 1-based индексацией) означает, что корзина пуста.

Этот массив экстремально компактен. Поскольку он содержит только int (Value Type), при загрузке в кэш процессора (L1 Cache) в одну кэш-линию (64 байта) помещается 16 индексов. Это обеспечивает сверхбыстрое сканирование.9

### 2.2 Массив \_entries: Монолитное хранилище

Массив \_entries — это то место, где живут данные. В отличие от классических реализаций Linked List, где каждый узел — это отдельный объект в куче, здесь используется массив структур Entry.

#### Структура Entry:

C#

private struct Entry  
{  
 public int hashCode; // Кэшированный хеш (31 бит)  
 public int next; // Индекс следующего элемента (-1, если конец)  
 public TKey key; // Ключ  
 public TValue value; // Значение  
}

**Анализ структуры Entry:**

1. **hashCode:** Словарь **кэширует** результат GetHashCode(). Это критически важная оптимизация. При ресайзе (расширении словаря) необходимо пересчитать индексы корзин для *всех* элементов. Благодаря сохраненному хешу, рантайму не нужно заново вызывать пользовательский (потенциально медленный) метод GetHashCode(). Также это поле используется для "быстрой проверки" при поиске перед вызовом Equals.6
2. **next:** Это аналог указателя в связном списке. Но вместо ссылки на объект в памяти, это **индекс** в том же массиве \_entries. Это реализует так называемое "Сцепление внутри массива" (Array-Based Chaining).
3. **key / value:** Дженерики. Если TKey и TValue — это структуры (например, int), то Entry остается Plain Old Data (POD) структурой, и весь массив \_entries представляет собой плотный блок памяти без ссылок, что идеально для GC.11

### 2.3 Сравнительный анализ с классическим LinkedList (Unity context)

В Unity разработчики часто используют List<T> вместо LinkedList<T>, потому что последний создает объект-обертку (Node) для каждого элемента, вызывая фрагментацию памяти.

Реализация Dictionary в.NET решает эту проблему гениально:

* **Отсутствие аллокаций узлов:** При добавлении элемента (Add) не создается новый объект Node. Данные просто записываются в следующую свободную ячейку массива \_entries (структура).
* **Локальность данных (Spatial Locality):** Логически элементы могут образовывать цепочку (entry -> entry -> entry), но физически они находятся в одном непрерывном блоке памяти. Вероятность того, что entry уже находится в кэше процессора после чтения entry, значительно выше, чем при переходе по ссылке в куче.1

## 3. Алгоритмическая эволюция: Resizing и FastMod

Одной из самых глубоких тем, отличающих "старый".NET Framework от современного.NET 8, является математика вычисления индекса корзины. Это напрямую влияет на производительность в High-Load сценариях.

### 3.1 Эра простых чисел (Legacy:.NET Framework -.NET Core 2.0)

Традиционно размер массива \_buckets (Capacity) всегда выбирался как **простое число** (Prime Number).

Механизм:

При заполнении словаря происходит ресайз. Новый размер вычисляется как Size \* 2, после чего ищется ближайшее следующее простое число из заранее вычисленной таблицы (3, 7, 11, 17, 23,...).

Индекс вычислялся по формуле:

int bucketIndex = (hashCode & 0x7FFFFFFF) % primeSize;

Обоснование:

Использование модуля от простого числа позволяет равномерно распределить элементы по корзинам, даже если хеш-функция пользователя плохая (например, возвращает числа с паттернами, кратные 4 или 10). Простое число "разбивает" эти паттерны, снижая количество коллизий.14

Проблема:

Операция деления с остатком (% или процессорная инструкция IDIV) — одна из самых медленных арифметических операций CPU. Она может занимать 30-50 тактов процессора, в то время как сложение или битовый сдвиг занимают < 1 такта. В горячих циклах (например, join таблиц в памяти) это становилось узким местом.1

### 3.2 Эра FastMod и Vectorization (Modern:.NET 7/8)

В современных версиях.NET (.NET 7,.NET 8) команда Performance Engineering внедрила оптимизацию **FastMod** (автор Daniel Lemire).

Механизм FastMod:

Вместо использования инструкции DIV, рантаум использует трюк с умножением в высших разрядах. Формула (упрощенно) выглядит так:

uint index = (uint)(((ulong)hashCode \* (ulong)size) >> 32);

Эта операция использует только умножение и битовый сдвиг, что работает значительно быстрее. Однако, этот метод требует, чтобы распределение хешей было качественным. Поэтому в.NET 8 используются гибридные стратегии. В некоторых случаях (например, FrozenDictionary) размер может быть степенью двойки, что позволяет использовать сверхбыструю маску (hash & (size - 1)), но требует идеального хеширования.15

Влияние на код:

В исходном коде.NET 8 можно увидеть использование HashHelpers.FastMod. Для Unity-разработчика это сигнал: обновление рантайма само по себе ускоряет все операции со словарями на 20-30% без изменения кода игры/приложения, просто за счет более эффективной арифметики индексов.14

## 4. Жизненный цикл операции Add: Пошаговый разбор

Рассмотрим, что происходит "под капотом" при вызове dictionary.Add("Key", "Value"). Это позволит понять, откуда берутся коллизии и как работает цепочка.

Предположим состояние словаря:

* \_buckets размером 3: [-1, -1, -1]
* \_entries размером 3.
* \_count = 0.

### Шаг 1: Добавление первого элемента (No Collision)

1. Вызывается GetHashCode("Key1"). Допустим, результат 10.
2. Вычисляется индекс корзины: 10 % 3 = 1.
3. Проверяем \_buckets. Там -1 (пусто).
4. Записываем данные в \_entries (первый свободный слот по счетчику \_count):
   * hashCode = 10
   * next = -1 (мы указываем на то, что было в бакете ранее)
   * key = "Key1", value =...
5. Обновляем \_buckets = 0. Теперь корзина 1 указывает на индекс 0 в массиве entries.
6. \_count увеличивается до 1.

### Шаг 2: Добавление с коллизией (Collision)

1. Вызывается Add("Key2",...). Хеш равен 13.
2. Индекс корзины: 13 % 3 = 1. **Коллизия!** Корзина 1 уже занята (индекс 0).
3. Мы не бежим по цепочке (это было бы $O(N)$). Мы вставляем **в голову** (Head insertion).
4. Записываем данные в \_entries (следующий свободный слот):
   * hashCode = 13
   * next = 0 (значение, которое *было* в \_buckets). Теперь новый элемент указывает на старый.
   * key = "Key2", value =...
5. Обновляем \_buckets = 1. Теперь корзина указывает на новый элемент (индекс 1).

Результат:

Логическая структура: Bucket -> Entry -> Entry -> null.

Физически в памяти: \_entries заполнен последовательно. Такая схема гарантирует $O(1)$ на вставку даже при коллизиях (если не требуется ресайз).6

## 5. Алгоритм поиска: Hot Path Optimization

Метод FindEntry (внутренний метод, вызываемый TryGetValue, ContainsKey, индексатором) — это самый "горячий" участок кода в любом High-Load приложении.

C#

// Псевдокод логики FindEntry в.NET 8  
private int FindEntry(TKey key) {  
 // 1. Быстрая арифметика индекса (FastMod)  
 int hashCode = key.GetHashCode();  
 int i = \_buckets[HashHelpers.FastMod(hashCode, \_buckets.Length,...)];  
  
 // 2. Итерация по цепочке (Loop)  
 while (i >= 0) {  
 ref Entry entry = ref \_entries[i]; // Взятие по ссылке (без копирования!)  
  
 // 3. Проверка хеша (дешевая операция)  
 if (entry.hashCode == hashCode) {  
 // 4. Проверка равенства ключей (дорогая операция)  
 if (EqualityComparer<TKey>.Default.Equals(entry.key, key)) {  
 return i; // Найден  
 }  
 }  
 i = entry.next; // Переход к следующему в цепочке  
 }  
 return -1;  
}

**Критические аспекты производительности:**

1. **Ref Locals:** Использование ref Entry (доступно в C# 7+) позволяет работать с данными прямо в массиве без копирования структуры на стек. Для больших структур-значений (struct) это огромный выигрыш.17
2. **Short-Circuiting:** Проверка entry.hashCode == hashCode выполняется первой. Сравнение двух int занимает 1 такт CPU. Если они не равны, дорогой вызов Equals() (который может быть виртуальным или включать сравнение строк) пропускается. Это объясняет, почему **качество хеш-функции** важнее скорости Equals.9
3. **Devirtualization:** В.NET 8 JIT-компилятор (благодаря PGO - Profile Guided Optimization) способен "девиртуализировать" вызов EqualityComparer<TKey>.Default. Если ключом является int или string, JIT подставляет прямую инструкцию сравнения (cmp или вызов оптимизированного нативного кода), избегая накладных расходов на вызов интерфейса.21

## 6. Удаление и Free List: Управление дырами

Удаление элемента (Remove) в такой архитектуре создает проблему. Мы не можем физически удалить структуру из середины массива \_entries, так как это потребовало бы сдвига всех последующих элементов (операция $O(N)$), что недопустимо для хеш-таблицы.

Вместо этого.NET реализует паттерн **Free List** (Список свободных слотов).

### Механизм удаления

Когда мы удаляем элемент по индексу i:

1. **Unlink:** Мы находим предшественника в цепочке (или бакет) и перебрасываем его next на \_entries[i].next. Элемент логически исключен из цепочки поиска.
2. **Очистка:** Поля key и value в \_entries[i] зануляются (default), чтобы GC мог собрать объекты, на которые они ссылались.
3. **Добавление в Free List:**
   * Поле next удаленного элемента (\_entries[i].next) начинает указывать на *текущую* голову списка свободных (\_freeList).
   * Поле \_freeList обновляется и становится равным i.
   * Счетчик \_freeCount увеличивается.

### Последствия для памяти

Когда мы добавляем *новый* элемент после удаления, словарь сначала проверяет \_freeCount. Если он > 0, он берет индекс из \_freeList и переиспользует эту "дырку" в массиве. Только если свободных слотов нет, он использует \_count (конец массива).8

Важный вывод (Memory Leak Risk):

Массивы \_entries и \_buckets никогда не уменьшаются автоматически. Если вы добавили 1 миллион элементов, а потом удалили 999 999, словарь все еще удерживает в памяти два массива на 1 миллион элементов каждый. В Enterprise-системах долгоживущие словари с высокой волатильностью (много добавлений и удалений) могут привести к фрагментации кучи Large Object Heap (LOH), так как массивы будут занимать > 85 КБ. Решение: использовать TrimExcess() или пересоздавать словарь периодически.17

## 7. Безопасность: Hash Flooding и рандомизация

В отличие от игр, где входные данные контролируются разработчиком, бэкенд открыт внешнему миру. Это создает вектор атаки **Hash Flooding DoS**.

### Механика атаки

Злоумышленник может сгенерировать тысячи строк (например, ключи JSON в POST-запросе), которые будут иметь одинаковый хеш-код.

Если все 10,000 ключей попадут в один бакет:

* Вставка превращается в линейный поиск дубликатов: $1 + 2 +... + N \approx O(N^2)$.
* Загрузка CPU взлетает до 100%, сервер перестает отвечать.

### Защита в.NET

Для защиты от этого в.NET реализована **Рандомизированная хеш-функция строк** (Randomized String Hashing, часто на базе Marvin32).

* При старте приложения (процесса) генерируется случайный 64-битный **Seed**.
* Метод string.GetHashCode() использует этот Seed.
* Следовательно, строка "admin" будет иметь *разные* хеш-коды при каждом перезапуске приложения или в разных доменах приложений. Злоумышленник не может заранее вычислить коллизии, так как не знает Seed.24

В современных версиях.NET (Core) словарь может динамически переключаться на рандомизированный компаратор, если обнаружит, что длина цепочки коллизий превысила пороговое значение (например, 100), тем самым защищаясь от атаки на лету.27

## 8. Сравнительный анализ: Unity vs Enterprise (Сводная таблица)

Этот раздел консолидирует знания для ментора, чтобы наглядно показать разницу подходов.

| **Характеристика** | **Подход Unity (Mono / IL2CPP)** | **Подход Enterprise (.NET 8/9)** | **Причина различия** |
| --- | --- | --- | --- |
| **Выбор коллекции** | Часто List<T> или массивы. Избегание Dictionary в Update. | Dictionary<K,V> — стандарт де-факто для поиска. List.Contains — табу. | В бэкенде N (количество элементов) велико. $O(N)$ недопустимо для Throughput. |
| **Foreach** | Исторический страх (боксинг итератора в старом Mono). | Итератор словаря — struct. foreach не создает мусора (Zero Alloc). | Компилятор C# и JIT в.NET Core оптимизировали итераторы структур.29 |
| **GC Strategy** | Избегать *любых* аллокаций в кадре. | Аллокации в Gen 0 (короткоживущие) — это нормально и дешево. | GC в сервере настроен на Throughput (Server GC), он эффективно чистит Gen 0. Бояться нужно LOH и Gen 2. |
| **Struct vs Class** | Часто Class, так как ссылки удобны. | Struct для ключей/значений, если они малы, для Data Locality. | Массив структур (Entry) повышает плотность данных в кэше CPU.11 |
| **Потокобезопасность** | Один поток (Main Thread). Dictionary безопасен. | Многопоточность. Обычный Dictionary **взрывается** (бесконечный цикл). | Использование ConcurrentDictionary обязательно при общем доступе.30 |

## 9. Продвинутые возможности.NET 8: Взгляд в будущее

Завершая теоретический модуль, важно упомянуть новейшие инструменты, доступные в.NET 8, которые отсутствуют в старых фреймворках.

### 9.1 FrozenDictionary

В пространстве имен System.Collections.Frozen появился FrozenDictionary.

* **Сценарий:** Конфигурация, справочники, данные, которые загружаются один раз и многократно читаются (Read-Heavy).
* **Магия:** При создании он анализирует *все* ключи и подбирает идеальную хеш-функцию и размер бакетов специально под этот набор данных, минимизируя коллизии до нуля. Это дает максимальную скорость чтения, превосходящую обычный словарь.8

### 9.2 AlternateLookup (в.NET 9 Preview / Ref Structs)

До.NET 9 поиск в словаре Dictionary<string, int> требовал наличия объекта string. Если у вас был ReadOnlySpan<char> (срез памяти без аллокации), вам приходилось делать ToString(), создавая мусор.

Новая фича AlternateLookup позволяет искать в словаре, используя Span, достигая истинного Zero Allocation даже на уровне API поиска.22

## Заключение

Dictionary<TKey, TValue> в.NET — это не просто контейнер, а сложный инженерный механизм, балансирующий между потреблением памяти, скоростью процессора и безопасностью.

Для Unity-разработчика, переходящего в бэкенд, критически важно понять:

1. **Память:** Словарь — это массивы структур. Это эффективно.
2. **Поиск:** Скорость зависит от GetHashCode. Коллизии убивают производительность.
3. **Безопасность:** Рандомизация хешей защищает сервер от DoS.
4. **Управление:** Словарь не сжимается сам. Удаление оставляет "дырки" в FreeList.

Это знание послужит фундаментом для следующих дней, где мы будем разбирать многопоточные словари (ConcurrentDictionary) и стратегии кэширования.

#### Источники

1. План обучения: Архитектура коллекций и кэширование
2. memory - .NET Dictionary, impressively fast but how does it work? - Stack Overflow, дата последнего обращения: декабря 3, 2025, <https://stackoverflow.com/questions/5379871/net-dictionary-impressively-fast-but-how-does-it-work>
3. Exploring the Internals of Dictionary in C# - DEV Community, дата последнего обращения: декабря 3, 2025, <https://dev.to/wirefuture/exploring-the-internals-of-dictionary-in-c-1i5p>
4. Hashtable and Dictionary Collection Types - .NET - Microsoft Learn, дата последнего обращения: декабря 3, 2025, <https://learn.microsoft.com/en-us/dotnet/standard/collections/hashtable-and-dictionary-collection-types>
5. 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>
6. How C# Dictionary Actually Works - Medium, дата последнего обращения: декабря 3, 2025, <https://medium.com/@vosarat1995/how-c-dictionary-actually-works-47f3a156055b>
7. Anatomy of the .NET dictionary - Steve Dunn, дата последнего обращения: декабря 3, 2025, <https://dunnhq.com/posts/2024/anatomy-of-the-dotnet-dictionary/>
8. 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/>
9. .Net Dictionary Internals With Examples | by erhan355 - Medium, дата последнего обращения: декабря 3, 2025, <https://medium.com/@erhan355/net-dictionary-internals-with-examples-8bf1057f802d>
10. How is C# dictionary variable stored in memory? - Stack Overflow, дата последнего обращения: декабря 3, 2025, <https://stackoverflow.com/questions/59985206/how-is-c-sharp-dictionary-variable-stored-in-memory>
11. Dictionary
12. Memory usage of Dictionaries in C# - Stack Overflow, дата последнего обращения: декабря 3, 2025, <https://stackoverflow.com/questions/37578559/memory-usage-of-dictionaries-in-c-sharp>
13. Performance of Dictionary vs List in C# 12 and .NET 8 | by Borjan Vasovski | Medium, дата последнего обращения: декабря 3, 2025, <https://medium.com/@vasovski/performance-of-dictionary-vs-list-in-c-12-and-net-8-1813d18952a5>
14. 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>
15. Performance Improvements in .NET 7 - Microsoft Developer Blogs, дата последнего обращения: декабря 3, 2025, <https://devblogs.microsoft.com/dotnet/performance_improvements_in_net_7/>
16. [Proposal] Vectorized System.Collections.Generic.Dictionary
17. runtime/src/libraries/System.Private.CoreLib/src/System/Collections/Generic/HashSet.cs at main · dotnet/runtime · GitHub, дата последнего обращения: декабря 3, 2025, <https://github.com/dotnet/runtime/blob/main/src/libraries/System.Private.CoreLib/src/System/Collections/Generic/HashSet.cs>
18. 1BRC in .NET among fastest on Linux: My Optimization Journey - Hot For Knowledge, дата последнего обращения: декабря 3, 2025, <https://hotforknowledge.com/2024/01/13/1brc-in-dotnet-among-fastest-on-linux-my-optimization-journey/>
19. HashHelpers.FastMod could use Bmi2.X64.MultiplyNoFlags(ulong, ulong) directly · Issue #113352 · dotnet/runtime - GitHub, дата последнего обращения: декабря 3, 2025, <https://github.com/dotnet/runtime/issues/113352>
20. c# - System Dictionary implementation explanation? - Stack Overflow, дата последнего обращения: декабря 3, 2025, <https://stackoverflow.com/questions/33638060/system-dictionary-implementation-explanation>
21. What's new in .NET 8 runtime - Microsoft Learn, дата последнего обращения: декабря 3, 2025, <https://learn.microsoft.com/en-us/dotnet/core/whats-new/dotnet-8/runtime>
22. Performance Improvements in .NET 8 - Microsoft Developer Blogs, дата последнего обращения: декабря 3, 2025, <https://devblogs.microsoft.com/dotnet/performance-improvements-in-net-8/>
23. Troubleshoot Out of Memory issues (System.OutOfMemoryException) in ASP.NET, дата последнего обращения: декабря 3, 2025, <https://learn.microsoft.com/en-us/troubleshoot/developer/webapps/aspnet/performance/troubleshoot-outofmemoryexception>
24. How does UseRandomizedStringHashAlgorithm help against hash flooding attacks?, дата последнего обращения: декабря 3, 2025, <https://stackoverflow.com/questions/78464528/how-does-userandomizedstringhashalgorithm-help-against-hash-flooding-attacks>
25. String.GetHashCode Method (System) - Microsoft Learn, дата последнего обращения: декабря 3, 2025, <https://learn.microsoft.com/en-us/dotnet/api/system.string.gethashcode?view=net-10.0>
26. What's new in ASP.NET Core in .NET 8 | Microsoft Learn, дата последнего обращения: декабря 3, 2025, <https://learn.microsoft.com/en-us/aspnet/core/release-notes/aspnetcore-8.0?view=aspnetcore-10.0>
27. Protect `Dictionary<,>` against hash collision flood attacks · Issue #4761 · dotnet/runtime, дата последнего обращения: декабря 3, 2025, <https://github.com/dotnet/runtime/issues/4761>
28. [API Proposal]: string.GetHashCodeNonRandomized · Issue #77679 · dotnet/runtime, дата последнего обращения: декабря 3, 2025, <https://github.com/dotnet/runtime/issues/77679>
29. DictionaryEntry Struct (System.Collections) - Microsoft Learn, дата последнего обращения: декабря 3, 2025, <https://learn.microsoft.com/en-us/dotnet/api/system.collections.dictionaryentry?view=net-10.0>
30. Thread-Safe collections - .NET | Microsoft Learn, дата последнего обращения: декабря 3, 2025, <https://learn.microsoft.com/en-us/dotnet/standard/collections/thread-safe/>
31. .NET 8 — FrozenDictionary performance | code-corner.dev, дата последнего обращения: декабря 3, 2025, <https://code-corner.dev/2023/11/08/NET-8-%E2%80%94-FrozenDictionary-performance/>
32. Using structures as composite keys in C# dictionaries, дата последнего обращения: декабря 3, 2025, <https://sii.pl/blog/en/using-structures-as-composite-keys-in-c-dictionaries/>