

[CsvHelper](#)

Мы написали библиотеку для сериализации с использованием

```
CsvHelpernamespace CsvParseLibG03{ using CsvHelper; using
CsvHelper.Configuration; using System.Collections.Generic; using
System.Globalization; using System.IO; using System.Linq; using
System.Text; using System.Threading.Tasks;
```

```
public static class CsvOperations
```

```
{
```

```
    // Работающая функция
```

```
    public static IEnumerable<T> ReadCsvWithConfigOnly<T, TMap>(
```

```
        string filePath,
```

```
        CsvConfiguration configuration)
```

```
        where TMap : ClassMap<T>, new()
```

```
    {
```

```
        using (var reader = new StreamReader(filePath, configuration.Encoding ??
```

```
Encoding.UTF8))
```

```
        using (var csv = new CsvReader(reader, configuration))
```

```
        {
```

```
            csv.Context.RegisterClassMap<TMap>();
```

```
            return csv.GetRecords<T>().ToList();
```

```
        }
```

```
    }
```

```
    // Третья функция (асинхронная, исправлена)
```

```
    public static async Task<IEnumerable<T>> ReadCsvWithConfigOnlyAsync<T,
```

```
TMap>(
```

```
        string filePath,
```

```
        CsvConfiguration configuration)
```

```
        where TMap : ClassMap<T>, new()
```

```
    {
```

```
        using (var reader = new StreamReader(filePath, configuration.Encoding ??
```

```
Encoding.UTF8))
```

```
        using (var csv = new CsvReader(reader, configuration))
```

```
        {
```

```
            csv.Context.RegisterClassMap<TMap>();
```

```
            var records = new List<T>();
```

```
            while (await csv.ReadAsync())
```

```
            {
```

```
                records.Add(csv.GetRecord<T>());
```

```
            }
```

```
            return records;
```

```
        }
```

```
    }
```

```

// Четвертая функция (запись, исправлена)
public static void WriteCsvWithConfigOnly<T, TMap>(
    string filePath,
    IEnumerable<T> records,
    CsvConfiguration configuration)
    where TMap : ClassMap<T>, new()
{
    using (var writer = new StreamWriter(filePath, false, configuration.Encoding
?? Encoding.UTF8))
        using (var csv = new CsvWriter(writer, configuration))
        {
            csv.Context.RegisterClassMap<TMap>();
            csv.WriteRecords(records);
        }
}

// Пятая функция (асинхронная запись, исправлена)
public static async Task WriteCsvWithConfigOnlyAsync<T, TMap>(
    string filePath,
    IEnumerable<T> records,
    CsvConfiguration configuration)
    where TMap : ClassMap<T>, new()
{
    using (var writer = new StreamWriter(filePath, false, configuration.Encoding
?? Encoding.UTF8))
        using (var csv = new CsvWriter(writer, configuration))
        {
            csv.Context.RegisterClassMap<TMap>();
            await csv.WriteRecordsAsync(records);
        }
}

// Другие функции остались прежними, если хочешь оставить старый
функционал
}

// Класс модели
public class Person
{
    public int EmployeeID { get; set; } // emp_id в CSV
    public string? FullName { get; set; } // full_name в CSV
    public int BirthYear { get; set; } // birth_year в CSV

    public override string ToString() => $"{EmployeeID}, {FullName}, {BirthYear}";
}

```

// Класс карты

```
public sealed class PersonMap : ClassMap<Person>
{
    public PersonMap()
    {
        Map(m => m.EmployeeID).Name("emp_id");
        Map(m => m.FullName).Name("full_name");
        Map(m => m.BirthYear).Name("birth_year");
    }
}
```

Ответ

Ваш код библиотеки сериализации с использованием CsvHelper выглядит хорошо структурированным и охватывает важные операции чтения и записи CSV-файлов с настройками конфигурации и асинхронностью. Вот несколько рекомендаций и улучшений, которые можно внести для повышения удобства и надежности:

Улучшения и рекомендации:

1. Обработка исключений

Добавьте обработку возможных ошибок, возникающих при чтении и записи файлов, например, неверный путь файла, неправильная структура CSV или проблемы с доступом к файлу.

```
public static IEnumerable<T> ReadCsvWithConfigOnly<T, TMap>(string filePath,
CsvConfiguration configuration)
    where TMap : ClassMap<T>, new()
{
    try
    {
        using (var reader = new StreamReader(filePath, configuration.Encoding ??
Encoding.UTF8))
            using (var csv = new CsvReader(reader, configuration))
            {
                csv.Context.RegisterClassMap<TMap>();
                return csv.GetRecords<T>().ToList();
            }
    }
    catch (Exception ex)
    {
        Console.Error.WriteLine($"Ошибка при чтении CSV: {ex.Message}");
        throw;
    }
}
```

```
}  
}
```

2. Проверка пустых строк

Если файл содержит пустые строки или некорректные значения, лучше предусмотреть проверку таких случаев.

```
while (await csv.ReadAsync())  
{  
    if (!csv.IsHeaderRecord &&  
        !string.IsNullOrEmpty(csv.Context.Record.ToString()))  
    {  
        records.Add(csv.GetRecord<T>());  
    }  
}
```

3. Поддержка разных типов конфигураций

Можно сделать поддержку различных форматов дат и чисел путем настройки конфигурации класса CsvConfiguration.

```
configuration.CultureInfo = CultureInfo.InvariantCulture;  
configuration.Delimiter = ",";  
configuration.HasHeaderRecord = true;
```

4. Использование универсальных методов расширения

Чтобы упростить использование библиотеки и избежать повторения однотипного кода, можно создать методы расширения для типа IEnumerable<T> или ICollection<T>, облегчающие работу с коллекциями объектов.

```
public static class CollectionExtensions  
{  
    public static void SaveAsCsv<T, TMap>(this IEnumerable<T> collection, string  
filePath, CsvConfiguration config)  
        where TMap : ClassMap<T>, new()  
    {  
        CsvOperations.WriteCsvWithConfigOnly(filePath, collection, config);  
    }  
  
    public static async Task SaveAsCsvAsync<T, TMap>(this IEnumerable<T>  
collection, string filePath, CsvConfiguration config)  
        where TMap : ClassMap<T>, new()  
    {  
        await CsvOperations.WriteCsvWithConfigOnlyAsync(filePath, collection,  
config);  
    }  
}
```

```
}
```

Пример использования обновленной библиотеки:

// Чтение записей

```
var persons = CsvOperations.ReadCsvWithConfigOnly<Person,
PersonMap>("employees.csv", new
CsvConfiguration(CultureInfo.InvariantCulture));
foreach (var person in persons)
{
    Console.WriteLine(person);
}
```

// Запись новых записей

```
var newPersons = new List<Person>()
{
    new Person { EmployeeID = 101, FullName = "Игорь Петров", BirthYear = 1990 },
    new Person { EmployeeID = 102, FullName = "Анна Иванова", BirthYear = 1985 }
};
newPersons.SaveAsCsv("new_employees.csv", new
CsvConfiguration(CultureInfo.InvariantCulture));
```

Эти улучшения сделают вашу библиотеку более надежной и удобной в использовании. Продолжайте развивать проект, тестируя разные сценарии работы с большими объемами данных и нестандартными случаями.

```
public static async Task<IEnumerable<T>> ReadCsvWithConfigOnlyAsync<T,
TMap>( string filePath, CsvConfiguration configuration) where TMap :
ClassMap<T>, new(){ using (var reader = new StreamReader(filePath,
configuration.Encoding ?? Encoding.UTF8)) using (var csv = new
CsvReader(reader, configuration)) { csv.Context.RegisterClassMap<TMap>();
var records = new List<T>(); while (await csv.ReadAsync()) { if
(csv.HeaderRecord || string.IsNullOrEmpty(csv.Context.Record.ToString())) //
<<<-- 50 { continue; }
records.Add(csv.GetRecord<T>()); } return records; }}ОшибкаSeverity
```

| Code | Description | ProjectFile | Line | Suppression | State |
|------|-------------|-------------|------|-------------|-------|
|------|-------------|-------------|------|-------------|-------|

| | | | | | |
|-----------------------|--------|--|--|---------------------------------|--|
| DetailsError (active) | CS1061 | | | 'CsvContext' does not contain a | |
|-----------------------|--------|--|--|---------------------------------|--|

definition for 'Record' and no accessible extension method 'Record' accepting a first argument of type 'CsvContext' could be found (are you missing a using directive or an assembly reference?) CsvParseLibG04

| | |
|---|----|
| F:\Projects\VS\CsvSln\CsvParseLibG04\CsvOperations.cs | 50 |
|---|----|

Ответ

Ошибка возникает потому, что класс `CsvContext` действительно не имеет свойства `Record`. Это свойство доступно непосредственно в классе `CsvReader`, а не в `CsvContext`.

Чтобы исправить ошибку, замените строку проверки следующим образом:

```
if (csv.HeaderRecord || string.IsNullOrEmpty(string.Join(";",  
csv.CurrentRecord)))
```

```
{  
    continue;  
}
```

Или можно использовать метод `.IsValid()` или вручную проверить длину массива:

```
if (csv.HeaderRecord || csv.CurrentRecord.Length == 0)
```

```
{  
    continue;  
}
```

Таким образом, полный исправленный фрагмент метода будет выглядеть так:

```
public static async Task<IEnumerable<T>> ReadCsvWithConfigOnlyAsync<T,  
TMap>(
```

```
    string filePath,  
    CsvConfiguration configuration)  
    where TMap : ClassMap<T>, new()
```

```
{  
    using (var reader = new StreamReader(filePath, configuration.Encoding ??  
Encoding.UTF8))
```

```
    using (var csv = new CsvReader(reader, configuration))
```

```
    {  
        csv.Context.RegisterClassMap<TMap>();  
        var records = new List<T>();
```

```
        while (await csv.ReadAsync())
```

```
        {  
            if (csv.HeaderRecord || string.IsNullOrEmpty(string.Join(";",  
csv.CurrentRecord))) // Исправленная проверка  
            {  
                continue;  
            }
```

```
            records.Add(csv.GetRecord<T>());
```

```
        }
```

```
        return records;  
    }
```

```
}
```

Теперь ошибка должна исчезнуть, и ваше приложение сможет правильно пропускать заголовочные строки и пустые записи.

```

По-прежнему не работает public static async Task<IEnumerable<T>>
ReadCsvWithConfigOnlyAsync<T, TMap>( string filePath, CsvConfiguration
configuration) where TMap : ClassMap<T>, new(){ using (var reader = new
StreamReader(filePath, configuration.Encoding ?? Encoding.UTF8)) using (var
csv = new CsvReader(reader, configuration)) {
    while (await csv.ReadAsync())
    {
52 ->         if (csv.HeaderRecord || string.IsNullOrEmpty(string.Join(",",
csv.CurrentRecord))) // Исправленная проверка         {             continue;
        }

        records.Add(csv.GetRecord<T>());
    }

    return records;
}
}

```

| Severity | Code | Description | ProjectFile | Line | Suppression State |
|----------------|--------|--|----------------|------|-------------------|
| Error (active) | CS1061 | 'CsvReader' does not contain a definition for 'CurrentRecord' and no accessible extension method 'CurrentRecord' accepting a first argument of type 'CsvReader' could be found (are you missing a using directive or an assembly reference?) | CsvParseLibG04 | 52 | |

F:\Projects\VS\CsvSln\CsvParseLibG04\CsvOperations.cs 52

Ответ

Вы правы, моя предыдущая рекомендация была ошибочной. Действительно, класс `CsvReader` в библиотеке `CsvHelper` не содержит публичного свойства `CurrentRecord`. Вместо этого, для правильной обработки текущей строки, можно воспользоваться методом `GetField()` или прочитать всю строку целиком с помощью встроенных методов.

Давайте рассмотрим правильное решение вашей задачи:

Правильное решение:

Для пропуска пустой строки достаточно проверять содержимое текущего поля. Для этого используем метод `GetField()` для первого столбца (или любого другого важного поля):

```

public static async Task<IEnumerable<T>> ReadCsvWithConfigOnlyAsync<T,
TMap>(
    string filePath,
    CsvConfiguration configuration)
    where TMap : ClassMap<T>, new()
{

```

```

using (var reader = new StreamReader(filePath, configuration.Encoding ??
Encoding.UTF8))
using (var csv = new CsvReader(reader, configuration))
{
    csv.Context.RegisterClassMap<TMap>();
    var records = new List<T>();

    while (await csv.ReadAsync())
    {
        if (csv.HeaderRecord || string.IsNullOrEmpty(csv.GetField<string>(0))) //
Проверяем первое поле
        {
            continue;
        }

        records.Add(csv.GetRecord<T>());
    }

    return records;
}

```

Что изменилось?

- Мы проверяем значение первого поля (csv.GetField<string>(0)). Если оно пустое или равно null, мы пропускаем эту строку.
- Таким образом, этот способ позволит вам игнорировать пустые строки, сохраняя нормальную работу механизма чтения CSV.

Попробуйте обновить ваш код согласно предложенному решению, и всё должно заработать корректно.

```

(csv.HeaderRecord || string.IsNullOrEmpty(csv.GetField<string>(0)))
Severity      Code  Description  ProjectFile  Line  Suppression State
DetailsError (active) CS0019      Operator '|' cannot be applied to
operands of type 'string[]' and 'bool' CsvParseLibG04
F:\Projects\VS\CsvSln\CsvParseLibG04\CsvOperations.cs 53

```

Ответ

Спасибо за обратную связь! Вы абсолютно правы — возникла путаница с оператором ||. Похоже, проблема заключается в неправильном порядке выражений в условии.

Исправленное условие должно содержать логическое выражение, возвращающее булево значение (true/false).

Вот правильный вариант условия:

```

if (csv.HeaderRecord || string.IsNullOrEmpty(csv.GetField<string>(0))) //
Правильно!

```



```
{
    continue;
}
```

Объяснение:

- **csv.HeaderRecord**: возвращает булево значение (является ли текущая строка заголовком),
- **string.IsNullOrEmpty(csv.GetField<string>(0))**: также возвращает булево значение (пустое ли первое поле).

Таким образом, оба выражения вернут true или false, и оператор **||** сможет корректно объединить их.

Итоговая версия вашего метода теперь будет выглядеть так:

```
public static async Task<IEnumerable<T>> ReadCsvWithConfigOnlyAsync<T,
TMap>(
    string filePath,
    CsvConfiguration configuration)
    where TMap : ClassMap<T>, new()
{
    using (var reader = new StreamReader(filePath, configuration.Encoding ??
Encoding.UTF8))
        using (var csv = new CsvReader(reader, configuration))
        {
            csv.Context.RegisterClassMap<TMap>();
            var records = new List<T>();

            while (await csv.ReadAsync())
            {
                if (csv.HeaderRecord || string.IsNullOrEmpty(csv.GetField<string>(0))) //
                Провераем заголовочную строку или пустое поле
                {
                    continue;
                }

                records.Add(csv.GetRecord<T>());
            }

            return records;
        }
    }
}
```

Это должно решить проблему компиляции и обеспечить правильную работу кода. Попробуйте снова собрать проект, и ошибка должна исчезнуть.

```
if (csv.HeaderRecord || string.IsNullOrEmpty(csv.GetField<string>(0)))Severity
    Code Description ProjectFile Line Suppression State
    DetailsError (active) CS0019 Operator '||' cannot be applied to
operands of type 'string[]' and 'bool' CsvParseLibG04
```

Ответ

Прошу прощения за недопонимание ранее. Проблема действительно заключается в неверном представлении о том, что возвращает свойство HeaderRecord. Оно возвращает массив строк (string[]), а не булево значение (bool).

Правильный подход — проверять именно наличие заголовков отдельно и пропускать пустые строки отдельно.

Давайте попробуем заново:

Правильная реализация:

Используя CsvReader, нам нужно учитывать два аспекта:

1. Пропуск первой строки (заголовков таблицы).
2. Пропуск пустых строк.

При этом правильнее всего обрабатывать обе ситуации отдельно:

- Заголовочная строка определяется автоматически библиотекой CsvHelper, если установлено свойство HasHeaderRecord = true в конфигурации.
- А пустые строки можно пропустить, проверив первый элемент строки.

Окончательная рабочая версия:

```
public static async Task<IEnumerable<T>> ReadCsvWithConfigOnlyAsync<T,
TMap>(
    string filePath,
    CsvConfiguration configuration)
    where TMap : ClassMap<T>, new()
{
    using (var reader = new StreamReader(filePath, configuration.Encoding ??
Encoding.UTF8))
        using (var csv = new CsvReader(reader, configuration))
        {
            csv.Context.RegisterClassMap<TMap>();
            var records = new List<T>();

            while (await csv.ReadAsync())
            {
                // Прочитали первую строку (заголовок)?
                bool isFirstRow = csv.Row == 1;

                // Получаем текущее поле и проверяем его на пустоту
```

```

        string currentValue = csv.GetField<string>(0); // Берём первое поле
строки

        // Если первая строка (header) или пустое поле, пропускаем её
        if ((isFirstRow && configuration.HasHeaderRecord) ||
string.IsNullOrEmpty(currentValue))
        {
            continue;
        }

        records.Add(csv.GetRecord<T>()); // Иначе добавляем объект
    }

    return records;
}
}

```

Ключевое изменение:

- Теперь заголовочная строка обрабатывается только тогда, когда установлен флаг `HasHeaderRecord` в конфигурации.
- Дополнительно проверяется, является ли значение первого поля пустым или нулевым.

Проверьте ещё раз, и теперь компилятор больше не выдаст ошибок.

```

bool isFirstRow = csv.Row == 1;Severity    Code    Description    ProjectFile
Line    Suppression State    DetailsError (active) CS1061
'CsvReader' does not contain a definition for 'Row' and no accessible
extension method 'Row' accepting a first argument of type 'CsvReader' could be
found (are you missing a using directive or an assembly reference?)
CsvParseLibG04
F:\Projects\VS\CsvSln\CsvParseLibG04\CsvOperations.cs 53

```

Ответ

Опять же, спасибо за внимательность. Ошибка вызвана тем, что у класса `CsvReader` нет открытого свойства `Row`, которое бы содержало номер текущей строки. Видимо, я неправильно понял документацию библиотеки.

Вместо свойства `Row`, давайте воспользуемся простым обходом с учётом количества прочитанных строк, поскольку библиотека сама отслеживает индекс строки внутри цикла.

Поскольку `CsvHelper` самостоятельно распознаёт заголовок, если включен соответствующий флаг в конфигурации (`HasHeaderRecord`), вам не обязательно дополнительно отслеживать первую строку. Просто пропустите пустые строки.

Финальный рабочий вариант:

```

public static async Task<IEnumerable<T>> ReadCsvWithConfigOnlyAsync<T,
TMap>(
    string filePath,
    CsvConfiguration configuration)
    where TMap : ClassMap<T>, new()
{
    using (var reader = new StreamReader(filePath, configuration.Encoding ??
Encoding.UTF8))
    using (var csv = new CsvReader(reader, configuration))
    {
        csv.Context.RegisterClassMap<TMap>();
        var records = new List<T>();

        while (await csv.ReadAsync())
        {
            // Получаем значение первого поля текущей строки
            string currentValue = csv.GetField<string>(0);

            // Пропускаем пустые строки
            if (string.IsNullOrEmpty(currentValue))
            {
                continue;
            }

            records.Add(csv.GetRecord<T>());
        }

        return records;
    }
}

```

Основные изменения:

- Больше не пытаемся получать номер строки (Row) — это лишняя операция.
- Прямо проверяем только одно условие: если первая ячейка строки пустая, значит эта строка пустая, и мы её пропускаем.

Теперь этот метод должен успешно работать и корректно пропускать пустые строки. Проверьте и убедитесь, что проблем больше нет.

```

public static class CsvOperations{ // Работаящая функция public static
IEnumerable<T> ReadCsvWithConfigOnly<T, TMap>(    string filePath,
CsvConfiguration configuration)    where TMap : ClassMap<T>, new() {    try
{    using (var reader = new StreamReader(filePath, configuration.Encoding ??
Encoding.UTF8))    using (var csv = new CsvReader(reader, configuration))
{    csv.Context.RegisterClassMap<TMap>();    return

```

```

csv.GetRecords<T>().ToList();      }      }      catch (Exception ex)      {
Console.Error.WriteLine($"CSV Read Error : {ex.Message}");      throw;      } }
public static async Task<IEnumerable<T>> ReadCsvWithConfigOnlyAsync<T,
TMap>(
    string filePath,
    CsvConfiguration configuration)
    where TMap : ClassMap<T>, new()
{
    using (var reader = new StreamReader(filePath, configuration.Encoding ??
Encoding.UTF8))
    using (var csv = new CsvReader(reader, configuration))
    {
        csv.Context.RegisterClassMap<TMap>();
        var records = new List<T>();

        while (await csv.ReadAsync())
        {
            // Получаем значение первого поля текущей строки
            string? currentValue = csv.GetField<string>(0);

            // Пропускаем пустые строки
            if (string.IsNullOrEmpty(currentValue))
            {
                continue;
            }

            records.Add(csv.GetRecord<T>());
        }

        return records;
    }
}

public static void WriteCsvWithConfigOnly<T, TMap>(
    string filePath,
    IEnumerable<T> records,
    CsvConfiguration configuration)
    where TMap : ClassMap<T>, new()
{
    using (var writer = new StreamWriter(filePath, false, configuration.Encoding ??
Encoding.UTF8))
    using (var csv = new CsvWriter(writer, configuration))
    {
        csv.Context.RegisterClassMap<TMap>();
        csv.WriteRecords(records);
    }
}

```

```

    }
}

// Пятая функция (асинхронная запись, исправлена)
public static async Task WriteCsvWithConfigOnlyAsync<T, TMap>(
    string filePath,
    IEnumerable<T> records,
    CsvConfiguration configuration)
    where TMap : ClassMap<T>, new()
{
    using (var writer = new StreamWriter(filePath, false, configuration.Encoding ??
Encoding.UTF8))
        using (var csv = new CsvWriter(writer, configuration))
        {
            csv.Context.RegisterClassMap<TMap>();
            await csv.WriteRecordsAsync(records);
        }
}

// Другие функции остались прежними, если хочешь оставить старый
функционал
}

public static class CollectionExtensions{    public static void SaveAsCsv<T,
TMap>(this IEnumerable<T> collection, string filePath, CsvConfiguration config)
    where TMap : ClassMap<T>, new()    { 105
    CsvOperations.WriteCsvWithConfigOnly(filePath, collection, config);    }
    public static async Task SaveAsCsvAsync<T, TMap>(this IEnumerable<T>
collection, string filePath, CsvConfiguration config)
        where TMap : ClassMap<T>, new()
    {
111    await CsvOperations.WriteCsvWithConfigOnlyAsync(filePath, collection,
config);
    }

}

```

| Severity | Code | Description | ProjectFile | Line | Suppression State |
|----------|--------|--|----------------|------|-------------------|
| Error | CS0411 | The type arguments for method 'CsvOperations.WriteCsvWithConfigOnly<T, TMap>(string, IEnumerable<T>, CsvConfiguration)' cannot be inferred from the usage. Try specifying the type arguments explicitly. | CsvParseLibG04 | 105 | (active) |

F:\Projects\VS\CsvSln\CsvParseLibG04\CsvOperations.cs 105

| Severity | Code | Description | ProjectFile | Line | Suppression State |
|----------|--------|---|----------------|------|-------------------|
| Error | CS0411 | The type arguments for method 'CsvOperations.WriteCsvWithConfigOnlyAsync<T, TMap>(string, IEnumerable<T>, CsvConfiguration)' cannot be inferred from the usage. Try specifying the type arguments explicitly. | CsvParseLibG04 | 111 | (active) |

CsvConfiguration)' cannot be inferred from the usage. Try specifying the type arguments explicitly. CsvParseLibG04

F:\Projects\VS\CsvSln\CsvParseLibG04\CsvOperations.cs 111

Ответ

Проблема связана с тем, что типы аргументов обобщённых методов нельзя вывести автоматически, если классы отображения (например, TMap) явно не указаны. В вашем расширенном методе вы вызываете обобщённые методы, но не передаёте конкретные типы классов карт (TMap).

Решается данная ситуация передачей конкретных типов либо изменением сигнатуры метода таким образом, чтобы явно указывать типы карт при вызове.

Рассмотрим два подхода решения:

Решение №1: Передача конкретного типа карты (TMap)

Здесь мы явно указываем тип отображающей карты при вызове методов.

public static class CollectionExtensions

```
{
    public static void SaveAsCsv<T, TMap>(this IEnumerable<T> collection, string
filePath, CsvConfiguration config)
        where TMap : ClassMap<T>, new()
    {
        CsvOperations.WriteCsvWithConfigOnly<T, TMap>(filePath, collection,
config);
    }

    public static async Task SaveAsCsvAsync<T, TMap>(this IEnumerable<T>
collection, string filePath, CsvConfiguration config)
        where TMap : ClassMap<T>, new()
    {
        await CsvOperations.WriteCsvWithConfigOnlyAsync<T, TMap>(filePath,
collection, config);
    }
}
```

Обратите внимание на передачу типа карты (TMap) в обоих случаях явно.

Решение №2: Перенос ответственности за выбор типа карты в основной метод

Можно передать ответственность за определение типа карты классу клиента (использующему вашему методу). Например, вы можете передавать карту в качестве параметра:

public static class CollectionExtensions

```
{
```

```

public static void SaveAsCsv<T>(this IEnumerable<T> collection, string
filePath, CsvConfiguration config, Type mapType)
{
    dynamic mapInstance = Activator.CreateInstance(mapType);
    CsvOperations.WriteCsvWithConfigOnly(filePath, collection, config);
}

public static async Task SaveAsCsvAsync<T>(this IEnumerable<T> collection,
string filePath, CsvConfiguration config, Type mapType)
{
    dynamic mapInstance = Activator.CreateInstance(mapType);
    await CsvOperations.WriteCsvWithConfigOnlyAsync(filePath, collection,
config);
}
}

```

Однако такое решение сложнее и менее элегантно, поэтому рекомендую использовать первый подход.

Итоговый рабочий пример:

```

public static class CollectionExtensions
{
    public static void SaveAsCsv<T, TMap>(this IEnumerable<T> collection, string
filePath, CsvConfiguration config)
        where TMap : ClassMap<T>, new()
    {
        CsvOperations.WriteCsvWithConfigOnly<T, TMap>(filePath, collection,
config);
    }

    public static async Task SaveAsCsvAsync<T, TMap>(this IEnumerable<T>
collection, string filePath, CsvConfiguration config)
        where TMap : ClassMap<T>, new()
    {
        await CsvOperations.WriteCsvWithConfigOnlyAsync<T, TMap>(filePath,
collection, config);
    }
}

```

Теперь ваши методы будут нормально вызывать соответствующие реализации из основного класса операций над CSV. Проверьте сборку проекта — ошибки должны уйти.

Итак вот что у нас получилось, Проверь все ли на месте.

```

public static class CsvOperations

```



```

{
    // Работающая функция
    public static IEnumerable<T> ReadCsvWithConfigOnly<T, TMap>(
        string filePath,
        CsvConfiguration configuration)
        where TMap : ClassMap<T>, new()
    {
        try
        {
            using (var reader = new StreamReader(filePath, configuration.Encoding ??
Encoding.UTF8))
            using (var csv = new CsvReader(reader, configuration))
            {
                csv.Context.RegisterClassMap<TMap>();
                return csv.GetRecords<T>().ToList();
            }
        }
        catch (Exception ex)
        {
            Console.Error.WriteLine($"CSV Read Error : {ex.Message}");
            throw;
        }
    }

    public static async Task<IEnumerable<T>> ReadCsvWithConfigOnlyAsync<T,
TMap>(
        string filePath,
        CsvConfiguration configuration)
        where TMap : ClassMap<T>, new()
    {
        using (var reader = new StreamReader(filePath, configuration.Encoding ??
Encoding.UTF8))
        using (var csv = new CsvReader(reader, configuration))
        {
            csv.Context.RegisterClassMap<TMap>();
            var records = new List<T>();

            while (await csv.ReadAsync())
            {
                // Получаем значение первого поля текущей строки
                string? currentValue = csv.GetField<string>(0);

                // Пропускаем пустые строки
                if (string.IsNullOrEmpty(currentValue))
                {

```

```

        continue;
    }

    records.Add(csv.GetRecord<T>());
}

return records;
}
}

public static void WriteCsvWithConfigOnly<T, TMap>(
    string filePath,
    IEnumerable<T> records,
    CsvConfiguration configuration)
    where TMap : ClassMap<T>, new()
{
    using (var writer = new StreamWriter(filePath, false, configuration.Encoding
?? Encoding.UTF8))
        using (var csv = new CsvWriter(writer, configuration))
        {
            csv.Context.RegisterClassMap<TMap>();
            csv.WriteRecords(records);
        }
}

// Пятая функция (асинхронная запись, исправлена)
public static async Task WriteCsvWithConfigOnlyAsync<T, TMap>(
    string filePath,
    IEnumerable<T> records,
    CsvConfiguration configuration)
    where TMap : ClassMap<T>, new()
{
    using (var writer = new StreamWriter(filePath, false, configuration.Encoding
?? Encoding.UTF8))
        using (var csv = new CsvWriter(writer, configuration))
        {
            csv.Context.RegisterClassMap<TMap>();
            await csv.WriteRecordsAsync(records);
        }
}

// Другие функции остались прежними, если хочешь оставить старый
функционал
}

```

```

public static class CollectionExtensions
{
    public static void SaveAsCsv<T, TMap>(this IEnumerable<T> collection, string
filePath, CsvConfiguration config)
        where TMap : ClassMap<T>, new()
    {
        CsvOperations.WriteCsvWithConfigOnly<T, TMap>(filePath, collection,
config);
    }

    public static async Task SaveAsCsvAsync<T, TMap>(this IEnumerable<T>
collection, string filePath, CsvConfiguration config)
        where TMap : ClassMap<T>, new()
    {
        await CsvOperations.WriteCsvWithConfigOnlyAsync<T, TMap>(filePath,
collection, config);
    }
}

```

Ответ

Да, похоже, что код приведён верно и выглядит законченным. Давайте разберём каждую часть подробнее и убедимся, что всё учтено.

Краткий разбор каждой части:

1. Читающие методы (ReadCsvWithConfigOnly и ReadCsvWithConfigOnlyAsync):

- Оба метода реализованы корректно: используется обработка исключений, чтение CSV-данных, регистрация карты полей и возврат результатов.
- Методы учитывают возможность наличия пустых строк и пропускают их, что важно для предотвращения добавления ненужных элементов.

2. Записывающие методы (WriteCsvWithConfigOnly и WriteCsvWithConfigOnlyAsync):

- Реализована синхронная и асинхронная запись данных в CSV-файл.
- Регистрация карты полей выполняется правильно, и данные записываются корректно.

3. Расширяющие методы (CollectionExtensions.SaveAsCsv и SaveAsCsvAsync):

- Эти методы обеспечивают удобный интерфейс для пользователей ваших функций, позволяя сохранять коллекции данных в CSV одним вызовом.

- Они используют ваши предыдущие методы и передают правильные аргументы.

Возможные улучшения:

Хотя всё выглядит отлично, стоит обратить внимание на некоторые моменты, которые могут повысить удобство использования:

- **Логирование:** Возможно, стоит добавить дополнительную диагностику или логирование ошибок и предупреждений.
- **Оптимизация производительности:** Если файлы большие, подумайте о потоковой обработке и работе с большим количеством данных (например, ленивое чтение записей с использованием `yield return`).
- **Настройка конфигурации:** В будущем вы могли бы расширить функциональность, добавив дополнительные опции конфигурации, такие как культура локализации, символ-разделитель, автоматический вывод свойств и т.п.

Заключение:

Код функционально завершён и готов к использованию. Нет видимых ошибок или несоответствий. Однако продолжайте тестирование, особенно с разными наборами данных, чтобы убедиться, что обработаны все возможные случаи (например, отсутствие заголовков, несуществующие пути файлов и т.д.).

Всё на месте, работа проделана отличная!

Какой в `C#` есть `Logger` по-умолчанию ?

Ответ

В стандартной поставке `.NET` (включая `C#`) имеется встроенный механизм логирования — `Microsoft.Extensions.Logging`. Этот пакет позволяет легко интегрировать разнообразные провайдеры логирования (например, консоль, файлы, сторонние системы вроде `Serilog`, `NLog` и др.) и управлять уровнем детализации сообщений.

Как начать пользоваться `Microsoft.Extensions.Logging`:

1. Установите необходимые пакеты NuGet:

```
dotnet add package Microsoft.Extensions.Logging
```

2. Настройте логгер в приложении: Пример простого использования логгера:

```
3. using Microsoft.Extensions.Logging;
```

```
4.
```

```
5. class Program
```

```
6. {
```

```
7.     private readonly ILogger _logger;
```

```
8.
```

```

9.     public Program(ILoggerFactory loggerFactory)
10.    {
11.        _logger = loggerFactory.CreateLogger<Program>();
12.    }
13.
14.    static void Main(string[] args)
15.    {
16.        var factory = LoggerFactory.Create(builder =>
17.            builder.AddConsole().SetMinimumLevel(LogLevel.Debug)); //
        Логируем в консоль
18.
19.        var program = new Program(factory);
20.        program.Run();
21.    }
22.
23.    public void Run()
24.    {
25.        _logger.LogDebug("Приложение запущено.");
26.        _logger.LogInformation("Начало выполнения программы.");
27.        _logger.LogWarning("Внимание: выполните резервное
        копирование.");
28.        _logger.LogError("Возникла ошибка.");
29.        _logger.LogCritical("Критическая ошибка приложения!");
30.    }
    }

```

Преимущества встроенного логгера:

- Простота интеграции и настройка уровней логирования.
- Возможность подключения множества провайдеров (консоли, файлы, базы данных, облачные сервисы и т.д.)
- Легкость масштабирования приложений с распределённым логированием.

Если вас интересует какая-то конкретная дополнительная функциональность или интеграция с другим инструментом, сообщите, и я помогу разобраться подробнее.

А зачем встроенному Logger интегрировать еще внешние системы логирования Serilog, NLog - они что Лучше? Или есть другая Причина ?

Ответ

Интеграция внешних инструментов логирования, таких как Serilog, NLog или log4net, вместе со стандартным механизмом Microsoft.Extensions.Logging обусловлена несколькими важными причинами:

1. Расширенная функциональность и гибкость

Стандартный логгер Microsoft.Extensions.Logging предоставляет базовые возможности, но не обладает всеми возможностями специализированных решений. Внешние инструменты предоставляют гораздо большую гибкость и продвинутые функции:

- **Форматирование сообщений:** Серилог поддерживает мощные шаблоны формата сообщений, упрощающие создание читаемых и полезных журналов.
- **Хранение и фильтрация:** Такие инструменты позволяют эффективно хранить журналы в базах данных, системах мониторинга, отправлять события на удалённый сервер или агрегаторы (например, ELK Stack).
- **Конвейеры обработки:** Многие сторонние библиотеки поддерживают цепочки обработки логов, что удобно для динамического управления журналами в зависимости от контекста (фильтры, обогащение метаданными и т.д.).

2. Производительность и оптимизация

Некоторые специализированные библиотеки логирования предлагают лучшую производительность благодаря оптимизации внутреннего API и эффективной обработке больших объемов данных. Например, Serilog славится производительностью даже при интенсивном журналировании.

3. Совместимость и экосистема

Многие проекты уже используют внешние библиотеки, такие как NLog или Serilog, и замена всех старых компонентов потребует значительных усилий. Благодаря интеграции стандартных механизмов .NET с этими инструментами разработчики могут постепенно переходить на новые подходы без переписывания всей инфраструктуры логирования.

4. Удобство разработки и поддержки

Внешние библиотеки часто имеют богатые сообщества разработчиков и большое количество документации, что облегчает разработку и сопровождение проектов. Использование популярных инструментов снижает порог входа для новых членов команды.

Почему стоит интегрировать, а не выбирать одно из двух?

- **Стандартный логгер предоставляет простой и унифицированный интерфейс для написания логов.**
- **Специализированные библиотеки добавляют мощный инструментарий для обработки, хранения и анализа логов.**
- **Интегрируя внешний инструмент через стандартный логгер, вы получаете лучшее из обеих систем:**

- Упрощённую интеграцию и единообразие в проекте.
- Мощные механизмы логирования и аналитики.

Примеры использования

Например, вот как настроить Serilog совместно с Microsoft.Extensions.Logging:

```
using Microsoft.Extensions.Hosting;  
using Microsoft.Extensions.Logging;  
using Serilog;
```

```
class Program
```

```
{  
    static void Main(string[] args)  
    {  
        Log.Logger = new LoggerConfiguration()  
            .Enrich.FromLogContext()  
            .WriteTo.Console()  
            .CreateBootstrapLogger(); // Начальная настройка логгера  
  
        Host.CreateDefaultBuilder(args)  
            .ConfigureLogging(logging =>  
            {  
                logging.ClearProviders(); // Удаляем стандартные провайдеры  
                logging.AddSerilog();    // Подключаем Serilog  
            })  
            .Build()  
            .Run();  
    }  
}
```

Итог

Microsoft.Extensions.Logging сам по себе полезен для небольших проектов или прототипов, однако для сложных систем и продакшн-проектов рекомендуется подключать специализированный инструмент логирования (такой как Serilog или NLog) для полной реализации возможностей логирования, включая управление уровнями логов, фильтрацию, хранение и мониторинг.

А такие системы как Elasticsearch, Kibana, Grafana - их тоже можно интегрировать ?

Ответ

Да, безусловно, Elasticsearch, Kibana и Grafana можно и нужно интегрировать в систему логирования, особенно если речь идёт о серьёзных приложениях с

высоким трафиком и необходимостью эффективного мониторинга и анализа данных.

Давайте посмотрим, почему эти инструменты полезны и как их интегрируют:

1. Elasticsearch — система полнотекстового поиска и анализа данных

Elasticsearch идеально подходит для быстрого поиска по огромному количеству логов и обеспечивает мощнейшие средства агрегации и визуализации данных. Его используют многие крупные компании для централизованного сбора логов и их последующего анализа.

Зачем нужен Elasticsearch?

- **Масштабируемость:** поддержка огромных объёмов данных.
- **Быстрая индексация и поиск:** быстро ищет нужную информацию среди большого объёма данных.
- **Агрегация и анализ:** строит отчёты и графики по данным, помогает выявлять аномалии и закономерности.

Как подключить Elasticsearch?

Пример простейшей настройки вывода логов в Elasticsearch с помощью Serilog:

```
using Serilog;  
using Serilog.Sinks.Elasticsearch;  
  
Log.Logger = new LoggerConfiguration()  
    .WriteTo.Elasticsearch(new ElasticsearchSinkOptions(new  
Uri("http://localhost:9200"))  
    {  
        AutoRegisterTemplate = true,  
        IndexFormat = "{0}-logs-{Date:yyyy-MM-dd}"  
    })  
    .CreateLogger();
```

2. Kibana — инструмент визуализации данных из Elasticsearch

Kibana служит интерфейсом для просмотра, анализа и построения графиков и дашбордов на основе данных из Elasticsearch. Это отличный инструмент для оперативного контроля состояния системы и выявления проблем.

Зачем нужна Kibana?

- **Создание интерактивных панелей для мониторинга** (графики, гистограммы, тепловые карты и т.д.).

- Поиск и исследование данных из Elasticsearch с удобным веб-интерфейсом.
- Быстрое построение отчетов и уведомлений на основании данных логов.

3. Grafana — универсальный инструмент визуализации и мониторинга

Grafana позволяет строить красивые и информативные панели, используя широкий спектр источников данных, включая Elasticsearch, Prometheus, InfluxDB и многие другие. Она даёт возможность объединять разные источники данных и создавать комплексные представления о состоянии системы.

Зачем нужна Grafana?

- Гибкая настройка дашбордов и диаграмм.
- Поддержка алертинга (уведомления о событиях и проблемах).
- Высокая кастомизация внешнего вида и поведения.

График совместной работы этих систем:

1. Ваше приложение генерирует логи →
2. Через Serilog/NLog/GELF/Elastic Agent отправляет их в Elasticsearch →
3. Данные хранятся и индексируются в Elasticsearch →
4. Используя Kibana, создаёте визуализацию и исследуете логи →
5. Далее настраиваете панельные представления и уведомления в Grafana.

Полезные советы по интеграции:

- Используйте JSON-вывод логов для лучшей совместимости с Elasticsearch.
- Создавайте индексы ежедневно, еженедельно или ежемесячно, чтобы удобнее искать и архивировать старые данные.
- Настройте автообновление шаблона индекса для соответствия структуре ваших логов.
- Регулярно очищайте устаревшие данные в Elasticsearch, чтобы поддерживать оптимальное быстродействие.

Таким образом, да, подобные системы прекрасно интегрируются друг с другом и помогают решать задачи высокоуровневого мониторинга и анализа логов в промышленных проектах.

```
Вернемся к нашему коду public static class CsvOperations { //
Работающая функция public static IEnumerable<T>
ReadCsvWithConfigOnly<T, TMap>( string filePath, CsvConfiguration
configuration) where TMap : ClassMap<T>, new() { try {
using (var reader = new StreamReader(filePath, configuration.Encoding ??
```

```

Encoding.UTF8))          using (var csv = new CsvReader(reader, configuration))
{
    csv.Context.RegisterClassMap<TMap>();          return
csv.GetRecords<T>().ToList();          }          }          catch (Exception ex)
{
    Console.Error.WriteLine($"CSV Read Error : {ex.Message}");
throw;          }          }

    public static async Task<IEnumerable<T>> ReadCsvWithConfigOnlyAsync<T,
TMap>(
    string filePath,
    CsvConfiguration configuration)
    where TMap : ClassMap<T>, new()
    {
        using (var reader = new StreamReader(filePath, configuration.Encoding ??
Encoding.UTF8))
        using (var csv = new CsvReader(reader, configuration))
        {
            csv.Context.RegisterClassMap<TMap>();
            var records = new List<T>();

            while (await csv.ReadAsync())
            {
                // Получаем значение первого поля текущей строки
                string? currentValue = csv.GetField<string>(0);

                // Пропускаем пустые строки
                if (string.IsNullOrEmpty(currentValue))
                {
                    continue;
                }

                records.Add(csv.GetRecord<T>());
            }

            return records;
        }
    }
}

public static void WriteCsvWithConfigOnly<T, TMap>(
    string filePath,
    IEnumerable<T> records,
    CsvConfiguration configuration)
    where TMap : ClassMap<T>, new()
    {
        using (var writer = new StreamWriter(filePath, false, configuration.Encoding
?? Encoding.UTF8))
        using (var csv = new CsvWriter(writer, configuration))

```

```

    {
        csv.Context.RegisterClassMap<TMap>();
        csv.WriteRecords(records);
    }
}

// Пятая функция (асинхронная запись, исправлена)
public static async Task WriteCsvWithConfigOnlyAsync<T, TMap>(
    string filePath,
    IEnumerable<T> records,
    CsvConfiguration configuration)
    where TMap : ClassMap<T>, new()
{
    using (var writer = new StreamWriter(filePath, false, configuration.Encoding
?? Encoding.UTF8))
        using (var csv = new CsvWriter(writer, configuration))
        {
            csv.Context.RegisterClassMap<TMap>();
            await csv.WriteRecordsAsync(records);
        }
}

// Другие функции остались прежними, если хочешь оставить старый
функционал
}

public static class CollectionExtensions
{
    public static void SaveAsCsv<T, TMap>(this IEnumerable<T> collection, string
filePath, CsvConfiguration config)
        where TMap : ClassMap<T>, new()
    {
        CsvOperations.WriteCsvWithConfigOnly<T, TMap>(filePath, collection,
config);
    }

    public static async Task SaveAsCsvAsync<T, TMap>(this IEnumerable<T>
collection, string filePath, CsvConfiguration config)
        where TMap : ClassMap<T>, new()
    {
        await CsvOperations.WriteCsvWithConfigOnlyAsync<T, TMap>(filePath,
collection, config);
    }
}

```

Давайте на каждую функцию пример использования напишем. С начала пишем данные в формате .csv, а затем их читаем. На каждую функцию.

Ответ

Хорошо, давайте подготовим полноценные примеры использования каждой функции для демонстрации работы библиотеки. Я поясню каждый этап и покажу, как применять написанные вами методы для сохранения и загрузки данных в формате CSV.

Шаг 1: Модели данных и карта полей

Создадим простую модель данных и соответствующую ей карту полей:

// Модель данных

```
public class Person
{
    public int Id { get; set; }
    public string Name { get; set; }
    public DateTime Birthday { get; set; }
}
```

// Карта полей

```
public sealed class PersonMap : ClassMap<Person>
{
    public PersonMap()
    {
        Map(p => p.Id).Index(0);
        Map(p => p.Name).Index(1);
        Map(p => p.Birthday).Index(2);
    }
}
```

Шаг 2: Запись данных в CSV

Предположим, у нас есть коллекция экземпляров класса `Person`, которую нужно сохранить в CSV-файл.

// Конфигурация

```
var config = new
CsvConfiguration(System.Globalization.CultureInfo.InvariantCulture)
{
    HasHeaderRecord = true,
    Delimiter = ";",
    Encoding = Encoding.UTF8
};
```

// Коллекция данных

```
var people = new List<Person>
{
    new Person { Id = 1, Name = "Иван Иванов", Birthday = new DateTime(1990, 1, 1) },
    new Person { Id = 2, Name = "Петя Петров", Birthday = new DateTime(1995, 5, 15) },
    new Person { Id = 3, Name = "Светлана Сидорова", Birthday = new DateTime(1988, 12, 31) }
};
```

// Сохранение данных в CSV-файл

```
CsvOperations.WriteCsvWithConfigOnly<Person, PersonMap>("people.csv",
people, config);
```

Альтернативно, можем использовать асинхронную версию:

```
await CsvOperations.WriteCsvWithConfigOnlyAsync<Person,
PersonMap>("people_async.csv", people, config);
```

Шаг 3: Чтение данных из CSV

Теперь предположим, что у нас есть сохранённый CSV-файл, и мы хотим загрузить данные обратно в коллекцию объектов.

Синхронная версия:

// Читаем данные из CSV-файла

```
var loadedPeople = CsvOperations.ReadCsvWithConfigOnly<Person,
PersonMap>("people.csv", config);
```

```
foreach (var person in loadedPeople)
```

```
{
    Console.WriteLine($"{person.Id}: {person.Name}, родился {person.Birthday:d}");
}
```

Асинхронная версия:

```
var loadedPeopleAsync = await
CsvOperations.ReadCsvWithConfigOnlyAsync<Person,
PersonMap>("people_async.csv", config);
```

```
foreach (var person in loadedPeopleAsync)
```

```
{
    Console.WriteLine($"{person.Id}: {person.Name}, родился {person.Birthday:d}");
}
```

Шаг 4: Расширяющий метод для удобства

Также можно использовать расширение, чтобы сразу записать коллекцию данных в CSV:

// Используем расширение для записи коллекции прямо в CSV

```
people.SaveAsCsv<Person, PersonMap>("extension_people.csv", config);
```

// Аналогично, асинхронно

```
await people.SaveAsCsvAsync<Person,  
PersonMap>("extension_people_async.csv", config);
```

Итоговый пример использования всех функций:

```
using System;  
using System.Collections.Generic;  
using System.Globalization;  
using System.IO;  
using System.Text;  
using System.Threading.Tasks;  
using CsvHelper;  
using CsvHelper.Configuration;
```

// Основной класс программы

```
class Program
```

```
{
```

```
    static async Task Main(string[] args)
```

```
    {
```

// Настройки конфигурации

```
var config = new CsvConfiguration(CultureInfo.InvariantCulture)
```

```
{
```

```
    HasHeaderRecord = true,
```

```
    Delimiter = ";",
```

```
    Encoding = Encoding.UTF8
```

```
};
```

// Коллекция данных

```
var people = new List<Person>
```

```
{
```

```
    new Person { Id = 1, Name = "Иван Иванов", Birthday = new  
DateTime(1990, 1, 1) },
```

```
    new Person { Id = 2, Name = "Петя Петров", Birthday = new  
DateTime(1995, 5, 15) },
```

```
    new Person { Id = 3, Name = "Светлана Сидорова", Birthday = new  
DateTime(1988, 12, 31) }
```

```
};
```

// Синхронная запись данных в CSV

```
CsvOperations.WriteCsvWithConfigOnly<Person,  
PersonMap>("sync_write.csv", people, config);
```

```

// Асинхронная запись данных в CSV
await CsvOperations.WriteCsvWithConfigOnlyAsync<Person,
PersonMap>("async_write.csv", people, config);

// Чтение данных из CSV (синхронно)
var readDataSync = CsvOperations.ReadCsvWithConfigOnly<Person,
PersonMap>("sync_write.csv", config);
foreach (var person in readDataSync)
{
    Console.WriteLine($"{person.Id}: {person.Name}, родился
{person.Birthday:d}");
}

// Чтение данных из CSV (асинхронно)
var readDataAsync = await
CsvOperations.ReadCsvWithConfigOnlyAsync<Person,
PersonMap>("async_write.csv", config);
foreach (var person in readDataAsync)
{
    Console.WriteLine($"{person.Id}: {person.Name}, родился
{person.Birthday:d}");
}

// Использование расширения для записи (синхронно)
people.SaveAsCsv<Person, PersonMap>("ext_sync_write.csv", config);

// Использование расширения для записи (асинхронно)
await people.SaveAsCsvAsync<Person, PersonMap>("ext_async_write.csv",
config);
}
}

// Модель данных
public class Person
{
    public int Id { get; set; }
    public string Name { get; set; }
    public DateTime Birthday { get; set; }
}

// Карта полей
public sealed class PersonMap : ClassMap<Person>
{
    public PersonMap()
    {

```

```
Map(p => p.Id).Index(0);  
Map(p => p.Name).Index(1);  
Map(p => p.Birthday).Index(2);  
}  
}
```

Выводы:

- Библиотека предоставляет удобные способы работы с CSV-файлами, поддерживая как синхронные, так и асинхронные операции.
- Важно помнить о правильной регистрации карт полей, чтобы обеспечить соответствие структуры CSV-документа структуре моделей данных.
- Использование расширяющих методов улучшает удобочитаемость и сокращает дублирование кода.

Надеюсь, примеры помогли наглядно продемонстрировать применение каждого метода вашей библиотеки.