**// MapGet()** Регистрирует обработчик для HTTP-запросов методом GET. GET-запросы используются для получения данных с сервера.

**// MapPost()** Регистрирует обработчик для HTTP-запросов методом POST. POST-запросы используются для отправки данных на сервер

**// MapPut()** Регистрирует обработчик для HTTP-запросов методом PUT. PUT-запросы обычно используются для обновления существующих ресурсов на сервере.

**// MapDelete()**  Регистрирует обработчик для HTTP-запросов методом DELETE. DELETE-запросы используются для удаления данных на сервере

**// MapFallback()** Регистрирует резервный маршрут, который выполняется, если ни один из других маршрутов не совпал с запросом. Это полезно для обработки запросов на несуществующие пути.

**// UseExceptionHandler()** Настраивает глобальный обработчик исключений для приложения. Это middleware, которое перехватывает все необработанные исключения и перенаправляет запрос на указанный маршрут для обработки ошибки.

**// UseStaticFiles()** Настраивает middleware для обслуживания статических файлов (например, CSS, JavaScript, изображения).

**// IResult** IResult — это интерфейс, представляющий результат выполнения HTTP-запроса. Он используется для возврата ответа от сервера клиенту. В ASP.NET Core это абстракция, которая позволяет возвращать различные типы HTTP-ответов

**// Exception** Exception — это базовый класс для всех исключений в .NET. Исключения используются для обработки ошибок, которые возникают во время выполнения программы.

**// IDisposable** IDisposable — это интерфейс, который используется для освобождения неуправляемых ресурсов (например, файлов, сетевых подключений, баз данных). Он содержит один метод — Dispose(), который вызывается для очистки ресурсов.

**// Results. Problem и RFC7231, section 6.6.1** Results.Problem() — это метод, который возвращает объект IResult, представляющий HTTP-ответ с детализированной информацией об ошибке. Этот метод соответствует стандарту RFC 7231, section 6.6.1, который описывает формат HTTP-ответов с ошибками. Результат ответа оперд

**// HttpContext** HttpContext — это объект, который представляет контекст HTTP-запроса и ответа. Он содержит всю информацию о текущем запросе (например, заголовки, тело запроса, путь) и позволяет управлять ответом (например, устанавливать статус код, писать в тело ответа).

Данные о знаменитостях сохраняются в файле JSON, путь к которому задаётся через BasePath и JSONFileName. В памяти данные хранятся в списке \_celebrities.

Репозиторий (Repository) — это класс, который управляет данными о знаменитостях. Он предоставляет методы для:

* Загрузки данных из файла (LoadData).
* Сохранения данных в файл (SaveChanges).
* Добавления, обновления, удаления и поиска знаменитостей.

Репозиторий создаётся один раз с помощью статического метода Create:

**Один раз вызываем и показать, как репозиторий вызывается**

using (IRepository repository = Repository.Create("Data"))

Если соединение с данными (например, файлом) обрывается, можно использовать интерфейс IDisposable для освобождения ресурсов. В данном коде метод Dispose пустой, так как нет неуправляемых ресурсов.

DAL — это слой доступа к данным, который отвечает за взаимодействие с хранилищем данных (база данных, файлы, API и т.д.). Его основные задачи:

1. **Изоляция бизнес-логики** от деталей работы с данными.
2. **Упрощение тестирования**: можно подменить реальный DAL на mock-объект.
3. **Централизация логики доступа к данным**: все операции с данными выполняются в одном месте.

Паттерн Repository — это способ организации доступа к данным, который абстрагирует работу с хранилищем. В вашем коде:

* **Интерфейс IRepository** определяет контракт для работы с данными.
* **Класс Repository** реализует этот контракт, работая с данными в формате JSON.
* **Исключения создаются с помощью ключевого слова throw.**
* После throw указывается объект исключения.
* Исключение создается с помощью конструктора соответствующего класса.
* В конструктор передается сообщение об ошибке, которое будет доступно через свойство Message.