**Параллелизм**

**Какие инструменты для работы с потоками в .NET ?**

В .NET есть прямой доступ работы с потомками через класс Thread, он даёт доступ не напрямую к потокам, а обёртке (Менедж Тредами).

Находится в пространстве имён систем треадинг, где естьмонитор – это наш lock оператор и ещё там есть обёртки над мьютексом и симафором

Threadpool

Когда платформа .NET получает запрос (запрос может быть вызовом метода или функции из любого приложения). Для этого запроса обработки создается объект потока. Когда объект потока создается, для этого объекта потока выделяются некоторые ресурсы, например память.

**Пул потоков в C#** — это не что иное, как набор потоков, которые можно повторно использовать для выполнения задач в фоновом режиме.

Обычно пул потоков требуется, когда у нас создается несколько потоков для выполнения ряда задач, организованных в очередь. Как правило, у нас больше задач, чем потоков.

Для Thread есть метод ThreadPool .QueueUserWorkItem

Для Task есть метод Task .Factory.StartNew(Action)

WaitAll & WhenAll

**Task.WaitAll** - блокирует текущий поток, пока все остальные задачи не завершат выполнение. фактически блокирует и ожидает завершения всех других задач.

**WaitAny -** Он ждет, пока завершится хотя бы одна из массива задач.

**Task.WhenAll** - используется для создания задачи, которая будет завершена тогда и только тогда, когда все остальные задачи будут завершены. вы получите незавершенный объект задачи. Однако это не заблокирует, а позволит программе выполняться. С помощю него можно обрабатывать исключения.

На практике мы используем TPL более высокую абстракцию это Task и операторами async/await которы с ним работают

**Если на сервере не достаточно свободных потоков для обработки запроса возникает 503 ошибка**

Parallel LINQ (AsParallel())

Так же есть **PLINQ** который может распараллелитьцепочкукоманд на нужное количество потоков, он сам разберётся сколько в нашей коллекции элементов и как это оптимально распараллелить на доступные в системе ресурсы. Источник данных разделяется на

1

сегменты, и каждый сегмент обрабатывается в отдельном потоке. Это позволяет произвести запрос на многоядерных машинах намного быстрее, **но** параллельная обработка потенциально требует больших затрат ресурсов, то PLINK в этом случае может выбрать последовательную обработку, если она не требует больших затрат ресурсов (если коллекция небольшая)

**Момент:** предположим у нас есть коллекция которую мы распараллелили далее мы что-то делам с данными коллекции и через форич выводим, для вывода нужно будет собрать данные со всех потоков и склеить их. Для оптимизации мы можем использовать **ForAll** который принимает делегат Action и выводит данные в том же потоке в котором они обрабатываться. Тот же принцип с сортировкой используем **AsOrdered**

**Чем опасны потоки Dead Lock и Race Condition в .NET ?**

Race Condition – состояние гонки. Когда программа выполняется 1000 раз нормальна на 1001 возникает ошибка. Это результат «приходит» планировщика потоков рантайма, ОС, ядра. Для этого есть блокировки lock

Dead Lock – это когда поток 1 ждёт ресурс занятый потоком 2, а тот в сою очередь ждёт ресурс занятый потоком 2, а первому чтобы закончить нужны оба ресурса. Они будут ждать друг друга вечно.

Для диагностики можно использовать легирование, но это не даст 100% гарантии результату.

**Какие отменять таски ?**

В .NET Framework для резкой отмены выполнения треда был метод abort(), его выпилили в коре.

Для тасок есть метод Cancel(). Для этого нам надо создать объект CancellationTokenSource далее взять его токен и записать в CancellationToken. Метод кенсл вызывается у объекта cancelTokenSource и меняет значения токена IsCancellationRequested далее в программе делаем проверку если это значение истинно делаем определённые действия и вызываем метод Dispose()

Я знаю, что многие методы LINQ имеют принимающий параметр CancellationToken и скорее всего делает внутри себя проверку на IsCancellationRequested

Отслеживать работу можно как-то через IProgress интерфейс, хз как

**Как результат выдавать при отмене таски ?**

1) При получении сигнала отмены выйти из метода задачи, например, с помощью оператора return или построив логику метода соответствующим образом. Но следует учитывать, что в этом случае задача перейдет в состояние TaskStatus.RanToCompletion, а не в состояние TaskStatus.Canceled.

2

2) При получении сигнала отмены сгенерировать исключение OperationCanceledException, вызвав у токена метод **ThrowIfCancellationRequested**(). После этого задача перейдет в состояние TaskStatus.Canceled.

3) Обработчик отмены **Register**() вызывается на токене. Принимает Action

Данная ошибка запишется в AggregateException нам надо будет перебрать InnerExceptions и найти TaskCanceledException

**Почему плохо использовать Task без await и async void. Когда можно?**

В том случае когда нам надо выполнить какую-то работу не дожидаясь её выполнения, правда надо обрабатывать исключения + логирование в том методе

**Как связаны Threads и Tasks?** Таска выполняется на пуле потоков.

**Таск это** - высокоуровневая абстракция над некой задаче которую надо выполнить асинхронноиспользовавтаск ран вы запустите делегат туда переданыйна каком-то потоке из пула

**Поток** – это способ выполнить задачу

В .NET если какой-то поток завис(действительно что-то большое обрабатывает) он не будет мешать остальным, если на сервере есть свободные потоки

Таска может выполнятся на одном потоке. На одном потоке может быть много тасок, они переиспользуют потоки

**Различия между Task и Threads?**

Такс – просто набор программных инструкций, которые загружаются в память. Для выполнения задач используется пул потоков

Поток — это основная единица использования ЦП , состоящая из счетчика программ, стека и набора регистров. Поток выполнения — это наименьшая последовательность запрограммированных инструкций, которой планировщик может управлять независимо

**Асинхронный метод, как и обычный, может использовать любое количество параметров или не использовать их вообще. Однако асинхронный метод не может определять параметры с модификаторами out, ref и in.**

Различия:

3

1) Задача более абстрактна, чем потоки. Всегда рекомендуется использовать задачи вместо потока, поскольку он создается в пуле потоков, в котором уже есть созданные системой потоки для повышения производительности.

2) Задача может вернуть результат. Не существует прямого механизма возврата результата из потока.

3) Задача поддерживает отмену с помощью токенов отмены. Но Thread нет.

4) Вы можете прикрепить задачу к родительской задаче, таким образом, вы можете решить, будет ли родительская или дочерняя задача существовать первой.

5) При использовании потока, если мы получаем исключение в долго выполняющемся методе, невозможно перехватить исключение в родительской функции, но то же самое можно легко поймать, если мы используем задачи.

6) Вы можете легко строить цепочки задач. Вы можете указать, когда задача должна запускаться после предыдущей задачи, и вы можете указать, должно ли быть переключение контекста синхронизации. Это дает вам прекрасную возможность запустить длительную задачу в фоновом режиме, а затем задачу обновления пользовательского интерфейса в потоке пользовательского интерфейса.

7) Задача по умолчанию является фоновой задачей. У вас не может быть задачи переднего плана. С другой стороны, поток может быть фоновым или передним планом.

8) Планировщик задач по умолчанию будет использовать пул потоков, поэтому некоторые задачи могут не запускаться до тех пор, пока не будут завершены другие ожидающие задачи. Если вы используете Thread напрямую, каждое использование будет запускать новый Thread.

**Задача почти всегда лучший вариант; он предоставляет гораздо более мощный API и позволяет избежать ненужной траты потоков ОС . Задача — это абстракция, поэтому ее намного проще использовать.**

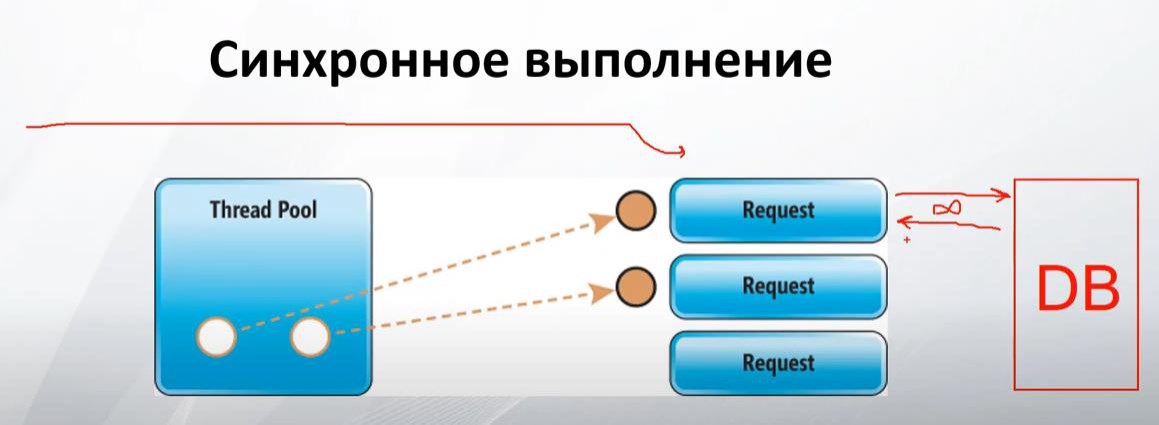
**Разница между асинхронностью и многопоточностью ?**

Многопоточность — параллельное выполнение, асинхронность — логическая оптимизация выполнения, которая может работать и в одном, и во многих потоках.

При синхронной работе, когда на сервер приходит запрос, ему выделяется свободный поток с пула который будет сопровождать и обрабатывать его. к примеру запрос на получения списка всех юзеров который обращается к базе и если обращение будет синхронно на время выполнения поток будет заблокирован(а если будет много таких запросов, они вызовут 503 ошибку), но если обращение будет асинхронно поток может временно освобождаться и обрабатывать новые запросы. При это когда база выдаст ответ этот или другой поток вернутся и смогут закончить запрос

**При асинхронном программировании мы не ускоряем работу одного отдельного запроса(поток не участвует при непосредственной работой с бд), но при этом мы масштабируем приложение и грамотно распределяем ресурсы сервера и как следствие ускорение приложения, то есть большее число запросов в единицу времени. Одни запрос к серверу – один запрос на обработку**

4



**Async/Await – операторы**

При использовании ключевого слова async мы создаём конечный автоман(State машину)

Конечный автомат - это место, где будет выполняться задача, и он будет ссылаться на то, какой поток инициировал эту задачу, поэтому, когда эта задача будет завершена, конечный автомат уведомит вызывающий поток о завершении задачи.

При этом async не делает метод асинхронным если нет await ибо такой код будет выполнятся синхронно, а ресурсы на создания конечного автомата будут выделены. Количество автоматов будет всегда одно и = 1.

await сообщает компилятору, что метод не может быть продолжен пока не завершится асинхронный процесс await. Так же он возвращает контроль над управлением наверх вызывающему async методу вплоть до освобождения потока и возвращения его в пулл

**Что возвращает метод async ?**

Task – представляет собой операцию, которая обычно выполняется асинхронно

Task<T> - возвращает тип T

Task – не возвращает ничего по умолчанию void. Можем получить состояние, ошибки

void – тяжело тестировать, отлавливать исключения, проверять состояние операции(либо проверить ошибку )

ValueTask<T> - структура

Таски управляются конечным автоматом, который был создан компилятором, когда мы пометили метод как async

5

**ValueTask & Task**

ValueTask во многом аналогично применению Task<T> за исключением некоторых различий в работе с памятью. Это структура.Поэтому применение ValueTask вместо Task приводит к копированию большего количества данных и соответственно создает некоторые дополнительные издержки.

Преимуществом ValueTask перед Task является то, что данный тип позволяет избежать дополнительных выделений памяти в хипе

ValueTask обычно применяется, когда результат асинхронной операции уже имеется.

**Правила написания асинхронного кода:**

1) Начинаем писать с нижнего уровня и далее вверх по структуре, а не с контроллера. Например со слоя работы с данными -> слоя бизнес логики -> контролеры

2) Добавляем в конце названия метода Async

3) При работе с EF следует везде использовать асинхронные аналоги методов. Рекомендация от майкрософт

**Когда стоит использовать:**

1) Операции input/output – работа с базой данных, файловой системой, сеть 2) Сложные расчёты, сильно нагружающие CPU

**Свойства класса Task:**

1) AsyncState: возвращает объект состояния задачи

2) CurrentId: возвращает идентификатор текущей задачи (статическое свойство) 3) Id: возвращает идентификатор текущей задачи

4) Exception: возвращает объект исключения, возникшего при выполнении задачи 5) Status: возвращает статус задачи:

a. Canceled: задача отменена

b. Created: задача создана, но еще не запущена

c. Faulted: в процессе работы задачи произошло исключение d. RanToCompletion: задача успешно завершена

e. Running: задача запущена, но еще не завершена

f. WaitingForActivation: задача ожидает активации и постановки в график выполнения

g. WaitingForChildrenToComplete: задача завершена и теперь ожидает заврешения прикрепленных к ней дочерних задач

h. WaitingToRun: задача поставлена в график выполнения, но еще не начала свое выполнение

6) IsCompleted: возвращает true, если задача завершена

7) IsCanceled: возвращает true, если задача была отменена

8) IsFaulted: возвращает true, если задача завершилась при возникновении исключения 9) IsCompletedSuccessfully: возвращает true, если задача завершилась успешно

6

Если необходимо, чтобы вложенная задача выполнялась как часть внешней, необходимо использовать значение TaskCreationOptions.AttachedToParent

ContinueWith – задача продолжения. позволяют определить задачи, которые выполняются после завершения других задач. Таким образом мы можем определить условия их вызова, передать из предыдущей задачи в следующую некоторые данные.

ContinueWith & await - совершенно разные вещи. Await не имеет отношения к ContinueWith, и более того, ломает его.

Такое поведение обусловлено особенностью механизма async/await — после прерывания метода из него возвращается незавершенная задача, что интерпретируется механизмом ContinueWith как завершение метода. Далее стартует следующий метод цепочки, однако после возврата результата — стейт-машина первого метода запустит вторую часть метода, и оба метода продолжат выполнение параллельно.

Из-за этого может получится что запуститься первая задача ContinueWith будет считать её как завершённой и запустит вторую задачу, хотя первая ещё не завершилась и далее завершится первая.

Класс Parallel - предназначен для упрощения параллельного выполнения кода.

Одним из методов, позволяющих параллельное выполнение задач, является метод **Invoke**. Метод Parallel.Invoke в качестве параметра принимает массив объектов Action. И таким образом, при наличии нескольких ядер на целевой машине данные методы будут выполняться параллельно на различных ядрах.

Метод Parallel.**For** позволяет выполнять итерации цикла параллельно. Первый параметр метода задает начальный индекс элемента в цикле, а второй параметр - конечный индекс. Третий параметр - делегат Action - указывает на метод, который будет выполняться один раз за итерацию

Метод Parallel.**ForEach** осуществляет итерацию по коллекции, реализующей интерфейс IEnumerable, подобно циклу foreach, только осуществляет параллельное выполнение перебора. Первый параметр представляет перебираемую коллекцию, а второй параметр -делегат, выполняющийся один раз за итерацию для каждого перебираемого элемента коллекции.На выходе метод возвращает структуру **ParallelLoopResult**, которая содержит информацию о выполнении цикла. В методах Parallel.ForEach и Parallel.For мы также можем, не дожидаясь окончания цикла, выйти из него ParallelLoopState.Breake()

**IsCompleted**: определяет, завершилось ли полное выполнение параллельного цикла

**LowestBreakIteration**: возвращает индекс, на котором произошло прерывание работы цикла

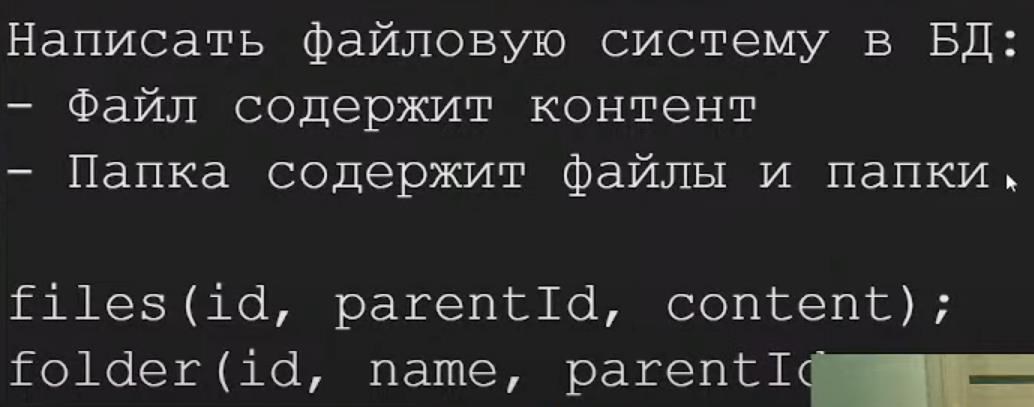
**Отмена параллельных операций Parallel**

7

Методы имеют перегруженный вариант с принимаемым параметром ParallelOptions у которого можно установить токен.

ConfigureAwait - данный метод принадлежит классуTask и позволяет указать, необходимо ли нам выполнять продолжение в том же контексте, где была вызвана асинхронная операция. По умолчанию, без использования этого метода, контекст запоминается и продолжение ведется в нем с помощью упомянутого метода Post. Стандартное веб-приложение ASP.NET Core **не имеет каких-то уникальных контекстов**

8

**Бази даних**

Задача на создание файловой системы

**Индексы помогают только при Select**

По умолчанию любой запрос выполняется full scan – чтобы сформировать ответ бд должна пройти все строчки.

Мы создаём индекс на определённое поле. Пускай это будет имя юзера. Далее к примеру, мы ищем пользователя Олег, поиск уже будет происходить по столбцу имя пользователя, а все имена будут отсортированы, далее первая буква имени будет сравниваться с символом в середине отсортированной записи, далее определяем с какой стороны нам надо дальше искать и далее делам по такому же принципу берём средний элемент же отобранной стороны и сравниваем. Таким образом мы не проверяем всё символы. Сложность log2n. **Индексы работают по отсортированным данным**(B-Tree)

Есть разные виды индексов HASH, когда колонка индексов хешируется и сравнивается их hash

Селективность индекса. Чем меньше количеству строк соответствует значение атрибута – тем выше селективность. Такие атрибуты следует использовать в начале индекса, ибо уже в начале отсеется больше записей. Можем проверить с помощью Group By на поле таблицы.

Атрибуты в **Order by добавляем в самый конец индекса**, они работают в самом конце

В начале атрибут с высокой селективностью, далее остальные атрибуты и в конце атрибут сортировки

В MSSQL индексы могут работать медленнее чем full scan

**Предостережение**:

1) не нужно заранее создавать индексы. Они создаются под конкретную задачу. 2) удалять неиспользуваемые индексы

3) не использовать на небольших объёмах данных (до несколько тысяч записей) 4) исходите из медленных запросов – создавайте под них индексы

9

**Недостатки**:

**1)** Изменение в бд. Индексы это структура данных которая так же храниться в базе и связана с основной таблицей. Когда меняется основная таблица, меняются все индексы, а если их много, то на их обновление требуется много времени. К примеру b-tree надо отсортировать опять все записи в столбце индекса.

**Плюсы:**

**1)** Увеличивается скорость запросов на чтение

**Когда стоит использовать** ?

1) В большинстве баз данных индексируется primary key, в некоторых рекомендуется foreign key и уникальные поля

2) колонки которые часто используются в where именно на равно(=) 3) group by - всегда

4) order by

5) Части в поле ON при join

**Когда стоит не использовать** ?

1) не использовать при **having(работает уже с прочитанными данными)** 2) с полями, которые часто изменяются (**update**)

3) insert – под вопросом, если мы его один раз вставили и больше не трогаем, то ок 4) НЕЛЬЗЯ ИСПОЛЬЗОВАТЬ ВЫЧЕСЛЕНИЯ НАД ПОЛЯМИ ФИЛТРА(+-)

Кластерный & не кластерный индексы

Кластерный индекс – его суть хранить таблицу отсортированную по какому-то ключу и это и есть наша таблица. Он может быть только один.

Не кластерный индекс – он хранится отдельно в виде какого-то хитрого дерева и лишь ссылается на эту таблицу. Может быть много.Содержат только те столбцы (ключевые), по которым определен данный индекс, а также содержит указатель на строки с реальными данными в таблице.

**Типы индексов**

1) Составной – может включить до 16 столбцов в индекс, но их общая длина ограничена 900 байтами. Как кластеризованный, так и некластеризованный индексы могут быть составными.

2) Уникальный индекс – автоматически является первичный ключ

3) Покрывающий индекс - позволяет конкретному запросу сразу получить все необходимые данные с листьев индекса без дополнительных обращений к записям самой таблицы.

10

**Транзакции –** последовательность операторов SQL, которые выполняются как единое целое(всё или ничего), которая не прерывается другими пользователями базы(не меняет состояние).

**Транзакции работают с DML операторами Insert, Update, Delete, Read**

**ACID–** требования к транзакционнойсистеме обеспечивающее надёжнуюпредсказуемую её работу

**А** –Атомарность. Гарантия что ни одна операция не будет выполнена частично, либо все либо ни одной

**С** – Согласованность(Консистентность). В транзакции не могут зафиксированы не корректные результаты, только валидные. К примеру нарушение Constraints(ограничения например что поле не должно быть нулл) либо указать ид которого нет в базе

**I** – Изолированность. Транзакции должны быть изолированы друг от друга и при параллельном выполнении результат не должен меняться.

**D** – Стойкость. Изменения успешно выполненной транзакции должны быть в любом случае сохранены

Чтобы создать транзакцию нам надо её задекларировать **DECLARE** @TransactionName VARCHAR(20) = 'Transaction1'; далее выполняемый код помещаем в **BEGIN TRAN** @TransactionName и выполняем. Для отката выполняем **ROLLBACK** **TRAN** @TransactionName.

У транзакции есть мозможность делать откат до определённой точки которую мы создаём с помощью **SAVE TRANSACTION** firstPoint и делать **ROLLBACK** firstPoint и откатяться изменения до firstPoint, если не указать **ROLLBACK** точку отката, он откатит всё. Если мы используем **COMMIT** на транзакции уже откатить не получится, данные сохранены.

**Уровни изоляции**

Чтобы установить уровень изоляции транзакции перед бигином надо написать

SET TRANSACTION ISOLATION LEVEL REPEATABLE READ;

1) Read Uncommitted – транзакции видят результаты других незавершённых транзакций. Если какой-топользовательчто-то меняет (незавершил, **НЕ ЗАКОМИТИЛ**,неоткати) это увидят все другие транзакции, которые работаю с тем же диапазоном данных. 0 изоляции.

2) Read Committed(по умолчанию) - параллельно исполняющиеся транзакции видят только зафиксированные изменения других транзакций

Если какой-то пользователь что-то меняет (**ЗАКОМИТИЛ**) это увидят все другие транзакции, которые работаюс тем же диапазоном данных. Если ониидут параллельно.

11

Недостаток, есть 2 запроса – одни добавляет письмо, второй возвращает количество писем и количество вернулось раньше добавления, как итог – вернётся одно количество писем – а по факту на 1 больше

3) Repeatable Read(репитбл рид) – не видит изменения Update и Delete, но видны результаты Insert. Фантомное чтение. Если какой-то пользователь что-то удалил или изменил (**ЗАКОМИТИЛ**), то это не увидят все другие транзакции, которые работаю с тем же диапазоном данных.

4) Seriaziable(сериалайзбл) – самый высокий и тяжёлый уровень изоляции. Блокирует чтение. Реализует принцип I в ACID, но он тяжёлый и из-за него падает производительность. Он тупо формирует очередь с транзакций, что приведёт к медленной работе.

По сути транзакции нам нужны для обработки важных данных. К примеру переводов с карты на карту.

**В** **ef** **core** **нам** **надо** **создать** **переменную** **типа** **using** **var** **transaction** **= context.Database.BeginTransaction();**

**И у нас будут доступные методы Commit(), Rollback(), RollbackToSavepoint(), CreateSavepoint()**

**Execution plan**

После выполнения запроса механизм обработки запросов быстро создает несколько планов выполнения и выбирает тот, который возвращает результаты с наилучшей производительностью. Есть два типа планов выполнения, чтобы быть конкретными:

1) Предполагаемый план выполнения - тип плана выполнения является просто предположением обработчика запросов о том, какие конкретные шаги должны быть задействованы при возврате результатов. Он часто генерируется до того, как запрос был выполнен.

2) Фактический план выполнения —план выполнения создается после выполнения запроса. Он показывает фактические операции и шаги, связанные с выполнением запроса. Это может или не может отличаться от предполагаемого плана выполнения

Мы можем разделить план выполнения на следующие пять шагов:

1) Сканирование кластерного индекса

2) Поток данных при сканировании кластеризованного индекса ( стрелка ) 3) Оператор сортировки

4) Поток данных от оператора сортировки ( стрелка ) 5) Выбор оператора

12

Жизненный цикл запроса

1) Вначале выражения LINQ обрабатываются Entity Framework Core, и на их основе создается объект запроса в тойформе, в которойонможет обрабатываться провайдером базы данных.

2) Созданный объект запроса кэшируется, что позволяет не пересоздавать его при повторном его выполнении.

3) Затем этот объект запроса передается провайдеру базы данных, который транслирует его на язык, понятный для базы данных (например, SQL).

4) База данных обрабатывает запрос и возвращает определенный результат.

5) EF Core получает результат обработки, и дальше его действия зависят от того, отслеживаются ли результаты запроса или нет.

Если запрос является отслеживаемым, то десь есть два альтернативных варианта:

1) Если данные, полученные из бд, представляют объекты, которые уже отслеживаются, то есть они уже есть в контексте, то EF возвращает те объекты, которые уже есть в контексте.

2) Если данные, полученные из бд, представляют объекты, которые еще не отслеживаться, их нет в контексте, то EF создает по этим данным новые объекты, добавляет в контекст, начинает их отслеживать и возвращает их пользователю.

Если запрос является не отслеживаемым, то есть отслеживание отключено с помощью метода **AsNoTracking**() то EF создает по этим данным новые объекты и возвращает их пользователю. В отличие от отслеживаемых запросов созданные объекты не добавляются в контекст и не отслеживаются.

Для отслеживаемых объектов всегда создается экземпляр из модели в кэше, а то что мы создаем другие экземпляры для использования в коде просто получают ссылку на объект в кеше(если там есть уже эта сущность)

По умолчанию все запросы, которые возвращают объекты классов моделей являются отслеживаемыми. Когда контекст данных извлекает данные из базы данных, Entity Framework помещает извлеченные объекты в кэш и отслеживает изменения, которые происходят с этими объектами вплоть до использования метода SaveChanges()/SaveChangesAsync(), который фиксирует все изменения в базе данных. Но нам не всегда необходимо отслеживать изменения.

13

**Типы данных, коллекции и структуры данных**

**В чем разница между IEnumerable и IQueryable?**

IQueryable интерфейс расширяет интерфейс IEnumerable добавляя туда 3 новых свойства.

IQueryProvider Provider – интервейс который содержит методы для создания квери CreateQuery и Execute который возвращает TResult

Expression Expressionб, Type ElementType

IQueryable используется в основном и ef core или других ORM для трансляции Expression из одного языка в другой, который будет собран через цепочку LINQ вызовов.

**IEnumerable** - представляет набор данных в памяти и может перемещаться по этим данным только вперед. Запрос, представленный объектом IEnumerable, выполняется немедленно и полностью, поэтому получение данных приложением происходит быстро.

При выполнении запроса IEnumerable загружает **все данные**, и если нам надо выполнить их фильтрацию, то сама фильтрация происходит на **стороне клиента**.

**IQueryable** - предоставляет **удаленный доступ** к базе данных и позволяет перемещаться по данным как в прямом порядке от начала до конца, так и в обратном порядке. В **IQueryable не все методы могут транслироваться.**

По сути IEnumerable запрос не будет транслироваться, а будет стягивание базы данных(работа с коллекцией в памяти), а IQueryable позволит обращаться с помощью IQueryProvider и транслировать запрос в sql и выполнять всё расчёты, выборку и так далее на стороне базы данных

Если разработчику нужен **весь набор** возвращаемых данных, то лучше использовать **IEnumerable**, предоставляющий максимальную скорость. Если же нам не нужен весь набор, а то только **некоторые отфильтрованные данные**, то лучше применять **IQueryable**.

**Как работают IObservable и IObserve? – данные интерфейсы реализуют паттерн наблюдатель.**

**IObservable** – реализует наблюдаемый объект. Он должен принимать в конструкторе список наблюдателей Observe и реализовать метод интерфейса для подписки **Subscribe**.

**IObserve** - реализует наблюдатель объект. Он должен реализовать методы **OnCompleted, OnError, OnNext**. Так же можно самому написать метод который принимает список Наблюдаемых объектов и на них подписываться.

14

**Разница IEnumerable, ICollection и IList ?**

**IEnumerable** - используется, когда мы хотим перебирать наши классы, используя цикл foreach.

**ICollection** - унаследован от интерфейсаIEnumerable. Имеет метода добавления, удаления, копирования, очистки и сравнение. Имеет поле Count

**IList** - реализует интерфейсы ICollection и IEnumerable. Имеет методы для работы по индексу IndexOf, Insert, RemoveAt. **Самый функциональный**

**В чем разница между массивом (T[]) и списком (List<T>) ?**

List<T> - по сути это оболочка над массивом T[].

Просто автоматически выделяет больший массив при добавлении новых элементов (или сжимает массив при удалении элементов).

Массив лучше использовать, когда мы заранее точно знаем сколько там буде элементов. Так как лист изначально выделят память по 4 элемента, а далее проверяет не заполнил ли лист полностью и добавляет памяти под в два раза большее количество. 4, 8, 16, 32. N\*2

**Где хранятся массивы? Массивы примитивных типов?**

Где будут хранится массивы примитивных типов определяет тип данных который буде хранится в массиве. Если это ссылочный, то в куче, если значимый то в стеке.

Ссылка на массив будет хранится в стеке, если он является локальной переменной или в куче если он является членом класса, так и с другими примитивными типами. Если примитивная переменная хранится в классе, её ссылка будет в классе, а если оналокальная то будет хранится в куче.

В **стек помещаются только локальные переменные** (и сгенерированные компилятором временные переменные), которые не являются закрытыми локальными переменными лямбда-выражения или анонимного метода и не находятся в блоке итератора.

**Когда генерируется дженерик-класс конкретного типа – при выполнении программы**

Во время компиляции, когда универсальный тип или метод компилируется в промежуточный язык Microsoft (MSIL), он содержит метаданные, которые определяют, что он имеет параметры типа. И уже в рантайма он создает специализированный универсальный тип, который заменяемыми в соответствующих местах в MSIL

**Рефлексия** - представляет собой процесс выявления типов во время выполнения приложения.

15

Для управления сборками в пространстве имен System.Reflection имеется класс Assembly.

Чтобы динамически загрузить сборку в приложение, надо использовать статические методы Assembly.LoadFrom() или Assembly.Load().

Assembly asm = Assembly.LoadFrom("MyApp.dll");

Получив все типы сборки с помощью метода GetTypes(), мы опять же можем применить к каждому типу все те методы, которые были рассмотрены в прошлой теме.

Ключевую роль в позднем связывании играет класс System.Activator. С помощью его статического метода Activator.CreateInstance() можно создавать экземпляры заданного типа. Позднее связывание позволяет работать не с ранее указанной сборкой в рантаймею

Поздняя типизация(С#) – это когда тип определяется не в момент компиляции, а вовремя рантайма.

Атрибуты - представляют специальные инструменты, которые позволяют встраивать в сборку дополнительные метаданные.

Атрибуты могут применяться как ко всему типу (классу, интерфейсу и т.д.), так и к отдельным его частям (методу, свойству и т.д.)

Основу атрибутов составляет класс System.Attribute, от которого образованы все остальные классы атрибутов.

**Расскажите о коллекции LinkedList<T>. Чем она отличается от других коллекций?**

LinkedList<T> -это посути цепькоторая хранит данные, следящее значениеипредыдущее в конце и в начале м будет null (пример сцепленные между собой вагоны состава поезда) удобнее использовать при вставке и удалении узлов в разные части, но затратное по памяти. При добавлении меняться ссылки. Поиск осуществляется перебором.

Своими словами он представляет цепь, каждый узел(LinkedListNode) которой хранит, ссылку на следующее и предыдущее значение и значения ячейки. Сама коллекция имеет ссылку на первый узел и последний.

List – **Пересоздаётся**. структура построенная поверх массива. Она хоть и создаётся с запасом, но если мы выполняем вставку к примеру 10 добавлений нормально, а 11 очень долго. Не очень хорош для вставки вначало. Так как лист лежит как масив рядом в памяти он попадает большими кусками в кеши процессора и по нему можно быстро пробежаться. Раз в 100 быстрее при чтении.

LinkedList – **Не пересоздаётся** исползаем чтобы не потерятькакие-то важные данные(есть моменты). Когда надо не часто бегать.

Используем LinkedList

1) При необходимости только последовательный доступ (вперед/назад)

16

2) При необходимости добавить ячейки в начало/середину списка 3) При добавлении со ссылкой на место для вставки.

**Что такое индексатор?**

Индексаторы позволяют индексировать объекты и обращаться к данным по индексу. Фактически с помощью индексаторов мы можем работать с объектами как с массивами.

возвращаемый\_тип this [Тип параметр1, ...]

{ get { ... } set { ... } }

Индексатор должен иметь как минимум один параметр.

К примеру у нас есть компания, а в ней есть свойство сотрудник которое представлено не как список, далее у сотрудника мы создаём индексатор и дальше с полем сотрудником у компании мы можем работать как с массивом.

Индексы могут быть перегружены, быть абстрактными, виртуальными – но не статическими.

StringBuilder – имеет индексатор

**Что такое объективный объект? Какие преимущества дает использование immutable object? Предложите способ реализации его в .NET.**

Неизменяемый(immutable)объект — это объект, который однажды создан и не может измениться. Уже есть один такой тип **String**

Если ваше приложение использует преимущества многопоточности, неизменяемость должна быть фундаментальной частью вашей архитектуры, поскольку неизменяемые объекты по своей сути являются потокобезопасными и невосприимчивы к условиям гонки(race-conditions).

Если мы используем DTO(объекты передачи данных) модели и хотим чтобы они не изменялись мы должны использовать неизменяемые объекты.

Для создания имутабельного типа надо удалить гетеры, установку значения делать только в конструкторе, а ссылочные значения копировать. Сами свойста сделать редонли и приватными, а работать с ними только через буличные переменны только с гетом.

Для работы с имутабельными колекциями, используем System.Collections.Immutable

В C# 9 появились новый ссылочный тип Record который создаёт имутабельные типы, однако нам надо установить ключевое слово **init** вместо гетера в свойстве. Для структры по умолчанию.

Во многим records похожи на обычные классы и структуры, например, они могут абстрактными, их также можно наследовать либо запрещать наследование с помощью оператора sealed.

17

Сравнения в рекордах происходит по значению, а не по ссылке.

Рекорды поддерживают копирование за ключевым словом **with** при этом в инициализаторе мы можем менять копируемые объекты.

**Когда использовать StringBuilder, а когда string? Как работает StringBuilder?**

**StringBuilder & Concat** что быстрее ?

Если мы складываем какую-то строку через Append или через «+», + скомпилируется в вызов метода стринг конкат с 5 параметрами или сколько мы там строк складываем, и он будет уже знать о длиннее и сразу выделит нужное количество памяти, а стринг билдер будет распределять память так как не знает длину.

Система для экономии памяти имеет пул интернированных строк, который может только расширятся но от туда ничего нельзя удалить. Методом **String.Intern()** мы можем добавить строку в этот пул.

Если строка будет оставаться неизменной на протяжении всей программы, используйте объект класса String, поскольку объект **String** неизменяем. Но он потокобезобасный.

Если строка может изменяться (например, много логики и операций при построении строки), то лучшим вариантом будет использование **StringBuilder**. Показывает лучшую производительность, так как новые изменения вносятся в существующий экземпляр

**Какими свойствами должна обладать идеальная хэш-функция?**

Хеш – это называется математическое преобразование информации в короткую, определенной длины строку

1)Хеш функция всегда однонаправленная 2)Фиксированный размер хеша

3) Хеш должен быть уникальный и не содержать коллизий

**Идеальная хеш функция**:

1) Детерминированность – одинаковые входные данные дают одинаковый хеш 2) Скрость

3) Однонаправленность

4) Лавинный эффект. Малейшее изменение несёт полное изменение хеша 5) Отсутствие коллизий

18

**Управление памятью**

**Что такое куча и стек? Различия, принцип работы.**

**Стек** — это специальная область памяти компьютера, в которой хранятся временные переменные, созданные функцией. В стеке переменные объявляются, сохраняются и инициализируются во время выполнения.

Это временное запоминающее устройство. Когда вычислительная задача будет завершена, память переменной будет автоматически стерта. Раздел стека в основном содержит методы, локальную переменную и ссылочные переменные

**Куча** — это память, используемая языками программирования для хранения глобальных переменных. По умолчанию все глобальные переменные хранятся в куче памяти. Он поддерживает динамическое выделение памяти. Куча не управляется автоматически для вас и не так жестко управляется процессором. Это больше похоже на свободно плавающую область памяти

**КЛЮЧЕВОЕ ОТЛИЧИЕ**

1) cтек — это линейная структура данных, а куча — иерархическая структура данных.

2) память стека никогда не будет фрагментирована, тогда как память кучи может стать фрагментированной, поскольку блоки памяти сначала выделяются, а затем освобождаются.

3) stack имеет доступ только к локальным переменным, в то время как Heap позволяет вам обращаться к переменным глобально.

4) переменные стека не могут быть изменены, тогда как переменные кучи могут быть изменены.

5) выделение и освобождение стека выполняются инструкциями компилятора, тогда как выделение и освобождение кучи выполняется программистом. (**под вопросом**)

**Минусы стека:**

1) ограничен по памяти

2) переполнение может вызвать аварийную остановку 3) произвольный доступ не возможен

**Минусы кучи:**

1) для вычисления надо больше времени

2) управление памятью в куче сложнее, так как она используется глобально

**Плюсы стека:**

1) переменные локальные автоматически будут автоматически удалены 2) переменные не могут быть изменены.

3) стек используется, когда переменная не используется вне этой функции. 4) быстро обрабатывает данные

19

**Плюсы кучи:**

1) гарбич колектор чистит память

2) позволяет обращаться к переменным глобально 3) не имеет ограничений на размер памяти

Значимые типы хранятся там где они используются. Если они используются как поле ссылочного типа, например переменная инт в классе, она будет хранится в куче. Ссылочные хранятся **всегда** в куче.

**Как работает сборщик мусора?**

GC запускается, когда в одном из поколений нахватает места, когда приложению нахватает памяти , вызов метода Colect

Поколения: 0, 1, 2. 0 – новые объекты, 1 – пережили одну сборку, 2 – более одной сборки.

При нехватке памяти ГК удаляет элементы, на которых больше нет ссылке в 0 интеграции, те на которых есть ссылки дальше с 0 они переходят в первое поколение и тд.

Нельзя явно переместить в другие поколение только 0 сборку.

Есть ещё Лапдж Обджект Хип, где храниться больше объекты от 85 тыс байт. ГК после очисти мусора **сдвигает все объкты один к одному**, так как там были пустые ячейки. Больше данные он так не делает (**Затратно**)

**Три поколения потому что** 0 самые молодые и большинство объектов находятся в 0 (нуждаться в регулярной чистке), во 2 старые объекты, а в первом буферная зона помещаться на одну итерацию. 4 много 2 мало (**быстрый переход с коротко жителя в долгожителя**)

Можно GC.Collect() **указывать какое поколение очищать**, если мы укажем 1 то удалиться 0 и 1, если укажем 2, то очиститься все поколения по необходимости и наш **Лапдж Обджект Хип чиститься**. Чистка второго поколения - **полная сборка мусора**.

На этапе очистки Лапдж Обджект Хип, так как мы большие объекты мы не двигаем, но когда появляться свободные ячейки на их место мы можем записывать данные(выделения объекта, равного или меньшего по размеру). В новых версиях появилось свойство(**LargeObjectHeapCompactionMode**), которое позволяет нам указыать двигать объекты или нет.

**Finalize, IDisposable**

Finalize – метод для очистки неуправляемой памяти.

Когда создаться объекты мы его записываем в очередь финализации. Когда на объект уже нет ссылок ГК смотрит есть ли он в очереди финализации, если объект там есть, то он его перемещает в очередь готовых к завершению и потом поток финализации подписан на ивенты на ГК и как только ГК помещает объекты в очередь готовых к заверению и кидает ИВЕНТ поток финализации удаляет эти объекты. Очищает только неуправляемые объекты.

20

**Используем** когда делаем **обёртку** над работой с низкоуровневым API (unmanage object). Но чтобы иметь какую-то гарантию очистки у наследуемся от класса **CriticalFinalizerObject**

Недостатки:

1) затратный по ресурсам

2) **непредсказуемый** - не знаем когда очиститься ресурсы потому что мы не знаем, когда отработает ГК

**3) он работает только с ссылочными типами** 4) он запускается сам

IDisposable – метод Dispose(). Вызывает очистку ресурсов. Конструкция позволяет выделить область в которой будет находиться объект, класс которого реализует IDisposable и при выходе с этого блока будет срабатывать Dispose(); Ели мы взываем Dispose(), то при наличии диструктора Finalize тоже отработает. Для подавления используем GC.SuppressFinalize();

**Метод Dispose() позволяет нам пометить объект как освобождаемый, в результате чего есть большая вероятность, что он будет уничтожен сборщиком мусора раньше**

**Dispose() нужно вызывать по всех иерархии**

**Используем** когда мы владеем экземпляром объекта, который уже реализует IDisposable. Стрим, конекш к базе. Появились AsycIDisposable

Недостатки:

1) **сам не вызываться**

Зачем нам зарезервировано слово using в C#, если у .NET есть автоматическое управление памятью?

using имеет два назначения:

1) это директива, которая используется для импорта пространств имен в начало файла кода

2) оператор использования. Создаёт одноразовый класс. Мы его использовали и вызвали очистку неявно.

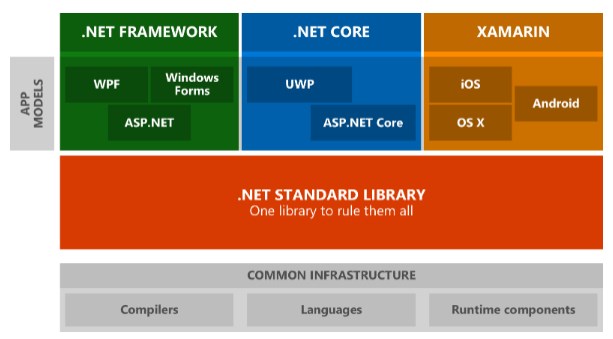
Конструкция юзинг принимает объкты кторые реализуют интерфейс IDisposable и гарантируют, что у них будет вызван метод Dispose().

По сути она реализует конструкцию try finaly под капотом

В C# using операторы определяют новый блок области видимости. Любые переменные, объявленные внутри using оператора, являются локальными для этого блока. В конце этой новой области вызывается метод dispose для этого объекта.

Если возникнет исключения в момент работы конструктора, dispose не будет вызван, так как оператор using ещё не получил ссылку на этот объект, потому что тот упал ещё до создания. Финализатор будет.

21

**Платформа .NET**

Какие исключения нельзя остановить в блоке catch?

StackOverflowException – ошибка перезаполнения стека

ThreadAbortException – возникает когда мы делаем аборт потоку и возникает ошибка. Он может быть перехвачен,но оно будет автоматически вызвано снова в конце блока catch.

Какая разница между .NET Standard Class Library и .NET Core Class Library?

NET Standard — это набор API-интерфейсов , которые должны быть реализованы на всех платформах .NET . Это объединяет платформы .NET и предотвращает будущую фрагментацию .

.NET Core, - увеличиват площадь поверхности .NET API, к которой может получить доступ ваша библиотека, и вы можете позволить только приложениям .NET Core быть совместимыми с вашей библиотекой.

Своими словами если мы хотим, чтобы наше приложение могло взаимодействовать с большем количеством типом устройств(мобильное приложение, веб-приложение, десктопное приложение) мы должны использовать NET Standard

Объясните разницу между отложением и немедленным исполнением в LINQ.

Отложенное выполнение переоценивается при каждом выполнении; это называется ленивой оценкой . Это одно из основных преимуществ отложенного выполнения: оно всегда дает вам **самые свежие данные**.

IEnumerable будет возвращать все элементы которые ранее записаны в коллекции, когда мы делаем операцию **ToList()**

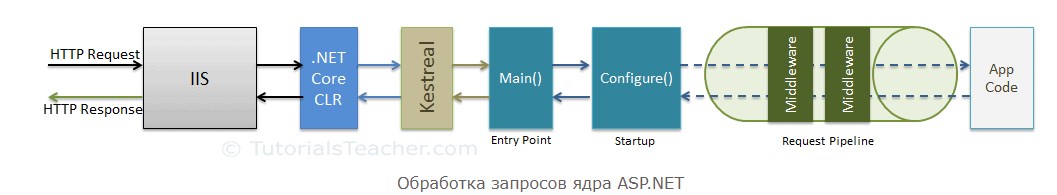
**Преимущества отложенного исполнения:**

1) Это позволяет избежать ненужного выполнения запросов и, следовательно, повышает производительность.

2) Создание запроса и выполнение запроса не связаны друг с другом, поэтому мы можем создать запрос LINQ в несколько этапов.

3) самые свежие данные.

22

**Для чего использовать метод ConfigureServices в Startup.cs?**

**Program -** настраиваете инфраструктуру своего приложения, такую как HTTP-сервер, интеграцию с IIS и источники конфигурации.

**Startup** - определяет, какие компоненты и функции использует ваше приложение такие как middleware. Он настраивает конвейер запросов, который обрабатывает все запросы к приложению. **Это точка входа в приложение**. Он содержит элементы, связанные с конфигурацией приложения.

IWebHostEnvironment - это интерфейс, содержащий информацию, относящуюся к среде веб-хостинга, в которой работает приложение

ConfigureServices – этот метод используется для **настройки служб**, используемых приложением. Когда приложение запрашивается в первый раз, оно вызывает метод ConfigureServices. Этот метод должен быть объявлен с модификатором **общего доступа**, чтобы среда могла читать содержимое из метаданных.

Configure - настраивает конвейер промежуточного программного обеспечения, который контролирует, как приложение обрабатывает HTTP-запросы и отправляет ответ .

**Middleware -** это не что иное, как компонент (класс), который выполняется при каждом запросе в приложении ASP.NET Core.

**Виды**:

1) предоставленное платформой, 2) добавленное через NuGet

3) ваше собственное промежуточное ПО

Для создания своего класса который будет работать в **Middleware** достаточно создать статический класс в нём статический метод который будет принимать IApplicationBuilder и на объекте билдера вызывать UseMiddleware где уже указываем класс-обычный CustomMiddleware в него конструктора принимаем RequestDelegate и далее создаём метод async Task Invoke(HttpContext httpContext). Или используем лямда выражение и пишем сразу в стартапе.

**Run & Use & Map**

**Run –** используется в конце поскольку они завершают конвейер

**Use –** принимает второй параметр делегат (next). используется в связке **await next()**, чтобы по завершению одного действия выполнилось другое. Без **await next(); работает как Run**

**Map –** в основном срабатывает по пути передаваемым первым параметром и вызывает методы контейнера до него.

23

**asp.net core web api request pipeline** – это цепочка **Middleware** которые выполняются один за другим каждый из которых может прерваться.

**Middleware & Action Filter**

**Middleware работает до того как работает asp net и соответственно раньше фильтров.**

**В Middleware мы можем легировать время выполнения написав свой Middleware**

**Middleware используется для отлавливания ошибок** к примеру мы можем мапить свои ошибки на статус коды.

Можно настроить:

Идентификацию – действие когда мы узнаём кто такой пользователь

Авторизация – подтверждение что пользователь есть он, ввод логина и пароля

Авторизация – предоставление каких-то прав

CORS - полное название Cross Origin Resource Sharing. Это стандарт **W3C**, который позволяет серверу выполнять междоменные вызовы из указанных доменов, отклоняя другие. По умолчанию из-за безопасности браузера он не позволяет веб-странице отправлять Ajax-запросы одного домена к другому домену.

Эта безопасность в терминах программного обеспечения называется политикой того же источника и не допускает приостанавливающую атаку сайта для чтения конфиденциальных данных с другого сайта.

Настраиваем с помощью AddCors и довлеем AddPolicy

Какая разница между AddTransient и AddScoped и AddSingleton в ASP.NET Core?

AddSingleton - создает **один экземпляр** сервиса при первом запросе и повторно использует этот экземпляр во всех местах, где требуется эта сервиса.

AddScoped – создаёт один экземпляр сервиса для всех мест, где он требуется на всю обработку запроса.

AddTransient – создаёт каждый раз новый экземпляр сервиса для всех мест, где он требуется.

Kestrel & IIS

Kestrel — это легкий **кроссплатформенный** веб-сервер с **открытым исходным кодом** для ASP.NET Core.

Хотя Kestrel может обслуживать приложение ASP.NET Core самостоятельно, Microsoft рекомендует использовать его вместе с обратным **прокси-сервером**, таким как **IIS** для повышения производительности, безопасности и надежности.

IIS – сервер специфичен для Windows

24

Однако Kestrel **не предоставляет** всех богатых функциональных возможностей полноценного веб-сервера, такого как IIS. Следовательно, мы обычно используем его в качестве **сервера приложений**, а один из вышеперечисленных серверов выступает в качестве **обратного прокси-сервера**

Как в ASP.NET WebAPI настроить кэширование ответов на HTTP-запросы?

Кэширование - представляет собой сохранение данных в специальном месте для более быстрого доступа к ним в будущем

Для кеширования ответов используется атрибут **ResponseCache**

Заголовки кэширования(cache-control)

1) public -кеш может хранить ответ либо на стороне клиента, либо в общем месте.

2) private - только частный кеш настороне клиента может хранитьответ, ноне общийкеш. 3) no-cache -кеш не должен использовать сохраненный ответ для любых запросов.

4) no-store - кеш не должен хранить ответ

Свойства **ResponseCache:**

1) Duration - создаст заголовок, который мы используем для установки продолжительности кеша на 2 минуты (120 секунд)

2) Location – заголовок. Енам **ResponseCacheLocation** a. Any – public

b. Client – private

c. None - браузер не использует кешированный ответ

3) NoStore - указывает, что клиент не должен кэшировать ответ

4) VaryByQueryKeys - можем заставить сервер отправлять новый ответ при изменении указанных параметров строки запроса

25

**Общие**

Какие протоколы сериализации вы знаете и где они применяются

Что такое чистая функция? Какие у нее преимущества?

Чистые функции напоминают математические функции. Они не делают ничего, кроме вычисления результата на основе своего ввода.

1) Он не имеет побочных эффектов . Как и математическая функция, чистая функция возвращает только значение.

2) Это последовательно. Его результат определяется только входными аргументами . При одних и тех же входных данных он всегда будет возвращать один и тот же результат.

Что такое dependency injection и зачем оно нужно?

DI - это среда программирования, которая предоставляет вам автоматическое внедрение зависимостей ваших компонентов.

Последний принцип солоди говорит что в типичной многоуровневой архитектуре приложения высокоуровневый компонент не должен напрямую зависеть от низкоуровневого компонента. Вы должны создать абстракцию (например, интерфейс) и сделать так, чтобы оба компонента зависели от этой абстракции.

1) Инверсия управления (IoC): это способ применения **принципа** **инверсии зависимостей**. Инверсия управления — это **фактический механизм**, который позволяет компонентам более высокого уровня зависеть от абстракции, а не от конкретной реализации компонентов более низкого уровня.

2) Внедрение зависимостей : это шаблон проектирования **для реализации инверсии управления**. Это позволяет внедрить конкретную реализацию низкоуровневого компонента в высокоуровневый компонент.

3) КонтейнерIoC :также известныйкак контейнервнедрения зависимостей (DI), этосреда программирования, которая предоставляет вам автоматическое внедрение зависимостей ваших компонентов.

DI можно реализовать самому придерживаясь принципов:

1) Внедрение конструктора : при таком подходе вы создаете экземпляр своей зависимости и передаете его в качестве аргумента конструктору зависимого класса.

2) Внедрение метода :в этом случае вы создаете экземпляр своей зависимости и передаете его определенному методу зависимого класса.

3) Внедрение свойств : этот подход позволяет вам назначить экземпляр вашей зависимости определенному свойству зависимого класса.

IoC необходимо:

1) Регистрация - знать какой тип объекта создавать для конкретной зависимоти

26

2) Разрешение - благодаря этой функции вам не нужно вручную создавать экземпляры объектов для управления зависимостями.

3) Расположение - управляет временем существования зависимостей

Встроенный в .NET Core контейнер IoC реализует IServiceProvider интерфейс. Поэтому, если по какой-то причине вы хотите создать свой собственный контейнер IoC, вам следует реализовать этот интерфейс. В .NET Core зависимости, которыми управляет контейнер, называются службами они **бывают системные и пользовательские**

Что такое cohesion и coupling ?

Связь(coupling) — мера того, насколько модуль (пакет, класс, метод) зависит от других модулей. **Желательно уменьшить связанность или уменьшить степень зависимости** данного модуля от других модулей системы.

Сплоченность(cohesion) — мера того, насколько тесно связаны члены (классы, методы, функциональные возможности внутри метода) модуля с другими членами того же модуля. **Желательно увеличить связность**, так как это указывает на то, что модуль имеет очень конкретную задачу и выполняет только эту задачу.

Что такое IaaS, PaaS, SaaS и каковы отличия между ними?

IaaS - инфраструктура как услуга. IAAS предоставляет доступ к таким ресурсам, как виртуальные машины и виртуальное хранилище (Amazon Web Services)

PaaS - платформа как сервис. предоставляет доступ к среде выполнения для инструментов развертывания и разработки приложений. (Facebook и поисковая система Google.)

SaaS - программное обеспечение как услуга. SAAS дает доступ конечному пользователю. (Веб-сайт MS Office, приложения Facebook и Google)

Статический класс & синглтон

Синглтон может имплементировать интерфейсы, может быть передан как параметр. Он чуть гибче

Статике не надо гарантировать единственность.

События & делегаты

События реализованы поверх делегатов, подписка отписка от методов, потокобезопасные

Лямда & делегаты

Лямда часто является делегатом, но не всегда. Часто используется как дерево выражений в ef core

27

ref & out & in

Все они передают значении по ссылке, актуально для значимых типов, для ссылочных будет указатель на указатель

ref - сообщает компилятору, что объект инициализируется перед входом в функцию (**передаёт ссылку и значение**)

out - сообщает компилятору, что объект будет инициализирован внутри функции. (**передаёт ссылку**)

in - сообщает компилятору, что объект не должен быть изменён.(для ссылочных)

boxing & unboxing

При упаковке создаётся объект типа Object и туда копируется значение

При распаковке в переменную копируется значение с Object

TCP & UDP

TCP – гарантирует доставку целостность и порядок данных. ориентированный на соединение, по IP-сети. Управление потоком определяет, когда данные необходимо отправить повторно, и останавливает поток данных до тех пор, пока предыдущие пакеты не будут успешно переданы.(карточные игры, чаты)

UDP – не гарантирует доставку целостность и порядок данных (для стрименговых платформ, игры гонки, стрелялки). является протоколом без установления соединения. Обычно он используется для связи в реальном времени. Однако UDP никогда не используется для **отправки важных данных**, таких как веб-страницы, информация из базы данных и т. д.; UDP обычно используется для потоковой передачи аудио и видео.

Причина, по которой UDP быстрее, чем TCP, заключается в том, что в нем отсутствует управление потоком или исправление ошибок. На данные, отправляемые через Интернет, влияют коллизии, и могут присутствовать ошибки.

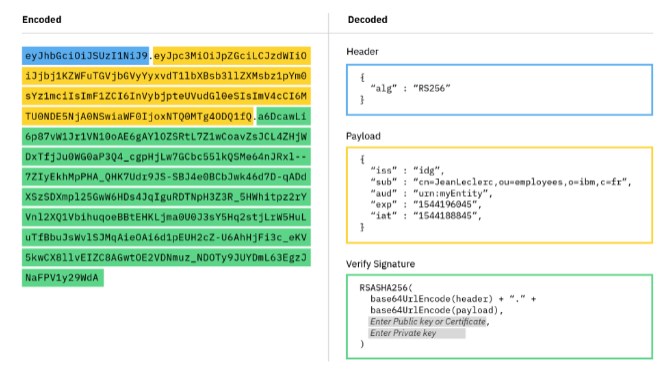
1) TCP ориентирован на соединение и надежен, тогда как UDP - соединение меньше и ненадежно.

2) TCP требует большей обработки на уровне сетевого интерфейса, тогда как в UDP этого нет.

3) TCP использует трехстороннее рукопожатие, управление перегрузкой, управление потоком и другие механизмы для обеспечения надежной передачи.

4) UDP в основном используется в тех случаях, когда задержка пакетов более серьезна, чем потеря пакетов.

28

TLS & SSL

Это протоколы шифрования. TLS новее.

JWT

JWT – подписанный json третей стороной благодаря секретному ключу .

Токен состоит из 3 частей.

Хедер – в закодированном виде хранится алгоритм котором был захеширован токен.

Пейлод - в закодированном виде хранятся данные к примеру клаймы, срок жизни токена, и настройки ввиде издателя токена, потребителя

Сигнатура –состоитиз закодированногохедера + закодированного заголовка + секретного ключа + хешируеться алгоритмом с хедера

Когда мы проверяем токен на сервере мы декодируем хедер чтобы получить алгоритм, декодируем пейлоад чтобы получить срок жизни токена, и потом кодируем всё обратно и хешируем добавляя секретный ключи и сравниваем с хешем который пришёл.

Почему нельзя использовать await в lock

Когда какой-то поток заходит в лок в определённое место записывается каким потоком занят данных блок, это сделано чтобы если мы потом в глубине стека опять попытались обратиться к тому же блоку это произошло мгновенно без всякого лока потому что этот поток уже залочин так как ресурс и так будет у этого потока.

И когда мы используем внутри await мы можем выйти уже на другом потоке

29