



# Task базирано програмиране в C#



Асинхронно програмиране за  
напреднали



# Защо Task?

---

- Изпълнява се асинхронно.
- Може да съдържа в себе си вложени Task-ове.
- Поддържа continuation - взаимосвързани Task-ове.
- Поддържа cancellation - при определени условия може да спрете Task.
- Оптимално използва системните ресурси.
- Има механизъм за обработка на грешки.
- Използва се лесно, когато се познават важните детайли.

# Какво е Task в C#?

---

- Това е асинхронна операция.
- Дефинира се като нормален Action в C# - чрез lambda expression, delegate или метод.
- Може да получава параметри и да връща резултат.
- Има статус, с който може да се следи текущото състояние.
- Има AggregationException, който държи информация за всички грешки.
- Управлява се от Task Scheduler.

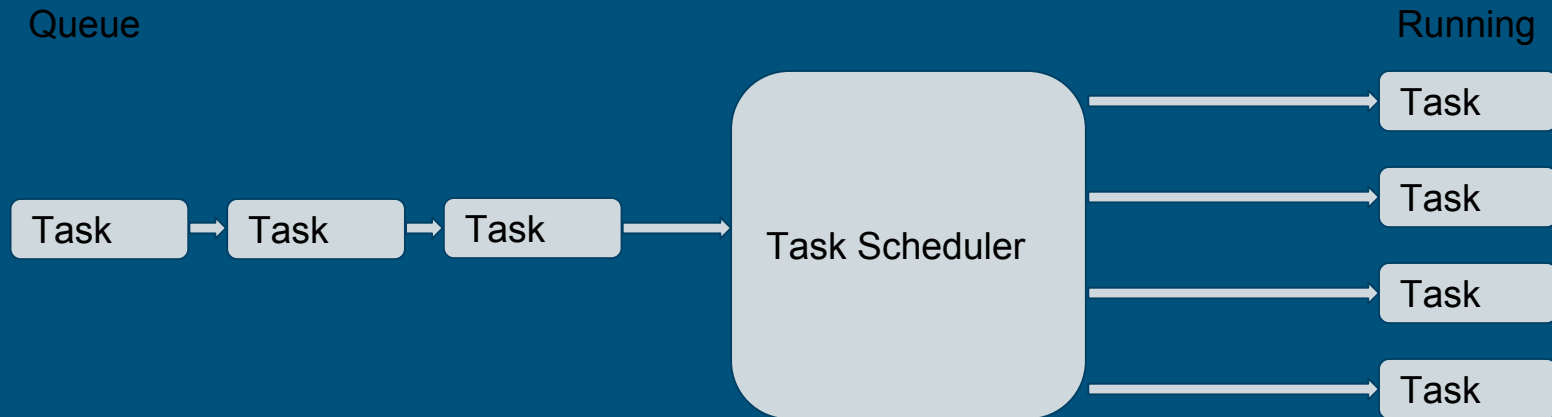
# Къде е подходящо да се използват Tasks?

---

- За продължителни блокови операции - изчисления, обработки на данни и т.н.
- За изчакване на отдалечени мрежови операции.
- За синхронизиране на няколко задачи, които очакват данни една от друга.

# Как работи всичко това?

---



# Създаване на Task-ове

---

- Стартиране на Task през lambda expression
  - Стартиране с delegate и предаване на параметър (винаги е object!)
  - Връщане на резултат от Task
- 
- <https://dotnetfiddle.net/BMbQ0I>

# Изчакване на Task

---

- Синхронно или асинхронно.
  - Един по един, група или един от няколко.
  - Със или без time out.
  - Проверка на статуса
  - Обработка на грешки
- 
- <https://dotnetfiddle.net/u6yE2f>

# Task Cancellation

---

- Използва се CancellationTokenSource.
- Той се подава като параметър.
- Вътре в Task-а може да се проверява cancellation token.
- При сетването му Task-ът може изобщо да не се стартира.
- Task-ът трябва да има достъп до token-а и да го проверява, ако трябва да бъде прекъснат.
- При успешен cancellation се хвърля exception, който е добре да се обработи.
- <https://dotnetfiddle.net/zjkg1l>

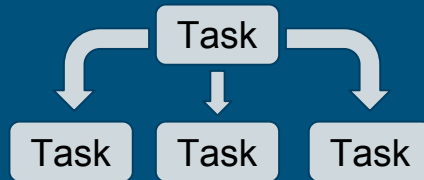


# Task Dependency

Chained Tasks



Parent - Child Tasks



- `ContinueWith` позволява да кажем кога да се изпълни следващия Task
- Опциите са при успех, при канселиране и при грешка.
- <https://dotnetfiddle.net/2uuXME>

# Интеграция с асинхронни API

---

- Възможно е част от старите асинхронни .NET API заявки да се изпълнят чрез Tasks.
- Използва се Task.Factory.FromAsync.
- <https://dotnetfiddle.net/p2fLFW>

# async и await

---

- Това е syntax sugar - свежда се до state machine, която превключва извикванията.
  - Улесняват работата с Task.
  - Задължават викащият метод да върне Task.
  - Ако това не се случи се губи състоянието на викащия Task и тези под него, включително възникналите exceptions.
- 
- <https://dotnetfiddle.net/FVj3f5>

# Task Scheduler

---

- Управлението на задачите може да се пренапише при необходимост.
- В .net има две имплементации:
- ThreadPoolTaskScheduler (Default)
- SynchronizationContextTaskScheduler
- Подават се като параметър на Task, когато се използват явно.

# Synchronization context

---

- Може да се съхраняват и извличат данни в и от него.
- Данните трябва да са [Serializable].
- Данните трябва да са Immutable (да не се променят).

# Мотики - синхронизация

---

- Кодът с `async` и `await` изглежда последователен, но се изпълнява асинхронно.
- Трябва да се следят споделените ресурси.
- Важат всички правила за паралелното програмиране - Lazy инициализации, Interlocked операции, критични секции, семафори и т. н.
- <https://dotnetfiddle.net/oCG00c>

# Мотики - deadlocks

---



# Важно да се знае!

---

- Task-овете не са еквивалентни на Thread-овете.
- Избягвайте `.Wait()` и `.Result`, вместо тях ползвайте `async` и `await`.
- Не пазете данни в контекста на текущия Thread - може да ги загубите след връщане от `await`.
- `async` и `await` прозрачно копират текущия `synchronization context` и го възстановяват, което понякога е ненужно и може да се изключи.
- Това става с `.ConfigureAwait(continueOnCapturedContext:false)`.



# Предимствата на async, await и Tasks

---

- Опростяват кода. Ефективно използват ресурсите на системата.
- Правят прозрачно continuation за всички сценарии - успех, грешка или канселиране.
- Правят ненужни .Wait() и .Result, което предпазва от deadlocks.
- Спестяват работата с нишки, която е трудоемка и скъпа операция.
- await пренасочва управлението към свободен Task и приложението не зависва.
- Имат множество параметри за конфигуриране според нуждите.
- Повечето нови Microsoft APIs са базирани на тях така или иначе.

# Недостатъци на async и await

---

- Веднъж използвате ли ги някъде навътре, трябва да разнасяте async Task и await из всички нива на приложението, което понякога води до сериозен refactoring.
- Възможно е някъде да пропуснете await и компилатора да го премълчи. Ако въпросният Task гръмне, exception-а избива късно, някъде из GC и може да събори цялото приложение.
- Спасението понякога е [AsyncPump](#) - потърсете си го, изпълнява Tasks синхронно. Не е добро решение, но спестява ненавременен refactoring.

# Източници за повече информация

---

- MSDN - разгледайте опциите за стартиране на Task - много са. Има и статии с примери.
- Stack Overflow - BG Mama за програмисти. Има предостатъчно теми, с това кой каква мотика е настъпил и колко голяма цицина е получил.
- Google за всичко останало.

# Гореизложил се:

---

Кирил Костов - софтуерен инженер.

e-mail: [kiril.kostov.varna@gmail.com](mailto:kiril.kostov.varna@gmail.com)

linkedIn: <https://www.linkedin.com/in/kiril-kostov-25930b89/>

Благодаря за вниманието!