



**Session**

# **Web-API mit async/await: Wobei hilft's und wie funktioniert's?**

**Sebastian Gingter**

<https://thinktecture.com/sebastian-gingter>

@phoenixhawk

Consultant & Erklärbar 🐼

# Sebastian Gingter

- Consultant & Erklärbär 🐻💬 bei der Thinktecture AG
- Fokus auf
  - Flexible und skalierbare Backend-Architekturen
  - Identity und Access-Management
  - Entwicklerproduktivität
  - Softwarequalität
  - Alles rund um .NET Core
- [sebastian.gingter@thinktecture.com](mailto:sebastian.gingter@thinktecture.com)
- Twitter: @phoenixhawk

# Special Day “Modern Business Applications”

Thema	Sprecher	Datum, Uhrzeit
<b>Web-APIs mit async/await: Wobei hilft's und wie funktioniert's?</b>	Sebastian Gingter	DI, 22. Februar 2022, 10.45 bis 11.45
<b>Web Components in modernen Web-Apps: Hype oder Heilsbringer?</b>	Peter Kröner	DI, 22. Februar 2022, 12.15 bis 13.15
<b>Produktivitäts-PWAs mit Angular auf Desktopniveau</b>	Christian Liebel	DI, 22. Februar 2022, 14.30 bis 15.30
<b>Blazor WebAssembly in der Praxis: 5 Dinge, die Sie kennen sollten</b>	Christian Weyer	DI, 22. Februar 2022, 16.00 bis 17.00
<b>Serverless Kubernetes mit Azure Container Apps</b>	Thorsten Hans	DI, 22. Februar 2022, 18.00 bis 19.00

# Was Euch erwartet (und was nicht):

- Viel Hintergrund-Infos & Theorie
- Tiefes Verständnis was “das Zeug eigentlich macht”
- und warum
- Eine kurze Zeitreise zurück ins letzte Jahrtausend
- Eher weniger Praxisbeispiele

# Inhalte

- Vom Prozess über Threads ...
- ... über synchronen vs. asynchronen I/O ...
- ... zu Tasks und dem .NET Threadpool
- `async` & `await`
- Tips & Tricks & Demos
- Fazit

# Macht `async/await` meine API schneller?

Nein, aber (viel) effizienter!

# Prozesse und Threads

# Prozess und Thread(s)

- Warum und wie?
- Prozesse und Threads sind frühe Virtualisierungskonzepte
- Prozess -> virtualisiert RAM
  - Abstrahiert den Zugriff einer Anwendung auf Speicherbereiche
- Thread(s) -> virtualisiert CPU
  - Abstrahiert den Zugriff einer Anwendung auf die CPU(-Zeit)
  - Threads haben Speicher- & Ausführungs-Overhead



# Threads

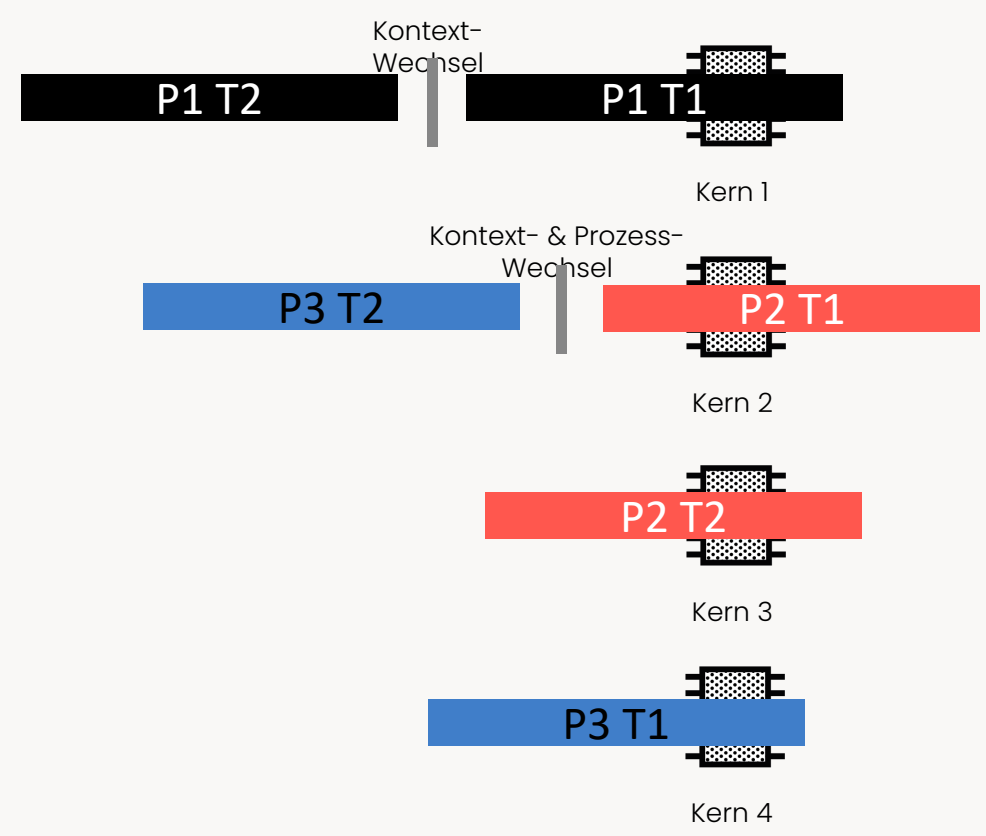
- Gleichgrosse Zeitscheiben (Quantum) pro Zuweisung
  - 20 ms auf Desktop-PCs, 60 ms für Prozess mit Window-Fokus\*
  - 120 ms auf Server-SKUs\*
- Thread läuft bis Quantum-Ende (preempted), ausser:
  - Er terminiert (ist fertig)
  - Er gibt freiwillig Rechenzeit ab (yield / Sleep)
  - Er blockiert / entered wait-state, z.B. bei OS-API-Call (Synchronous I/O)
  - Prozessor-Affinity wird geändert und erlaubt weiterlaufen nicht

\* Zahlen für Windows, Quelle: "Windows Internals", Solomin, Russinovich et al.  
Jeffrey Richter sagt hingegen: 30 ms in Windows, Linux hat andere Zeiten...

# Prozesse & Threads

- Erzeugen
  - Thread Kernel-Objekt Speicher allokieren (ca. 1 KB) & initialisieren
  - User Mode Data (Thread Environment Block) (4KB) allokieren & initialisieren
    - Exception handling chain, GDI/OpenGL hooks etc.
  - Stacks
    - Kernel-Mode (12/24 KB) und User-Mode (1 MB) allokieren & initialisieren
  - DLL Thread attach notifications (250-300 DLLs...)
- Beenden
  - Speicher (s.o.) freigeben
  - DLL Thread detach notifications (250-300 DLLs...)

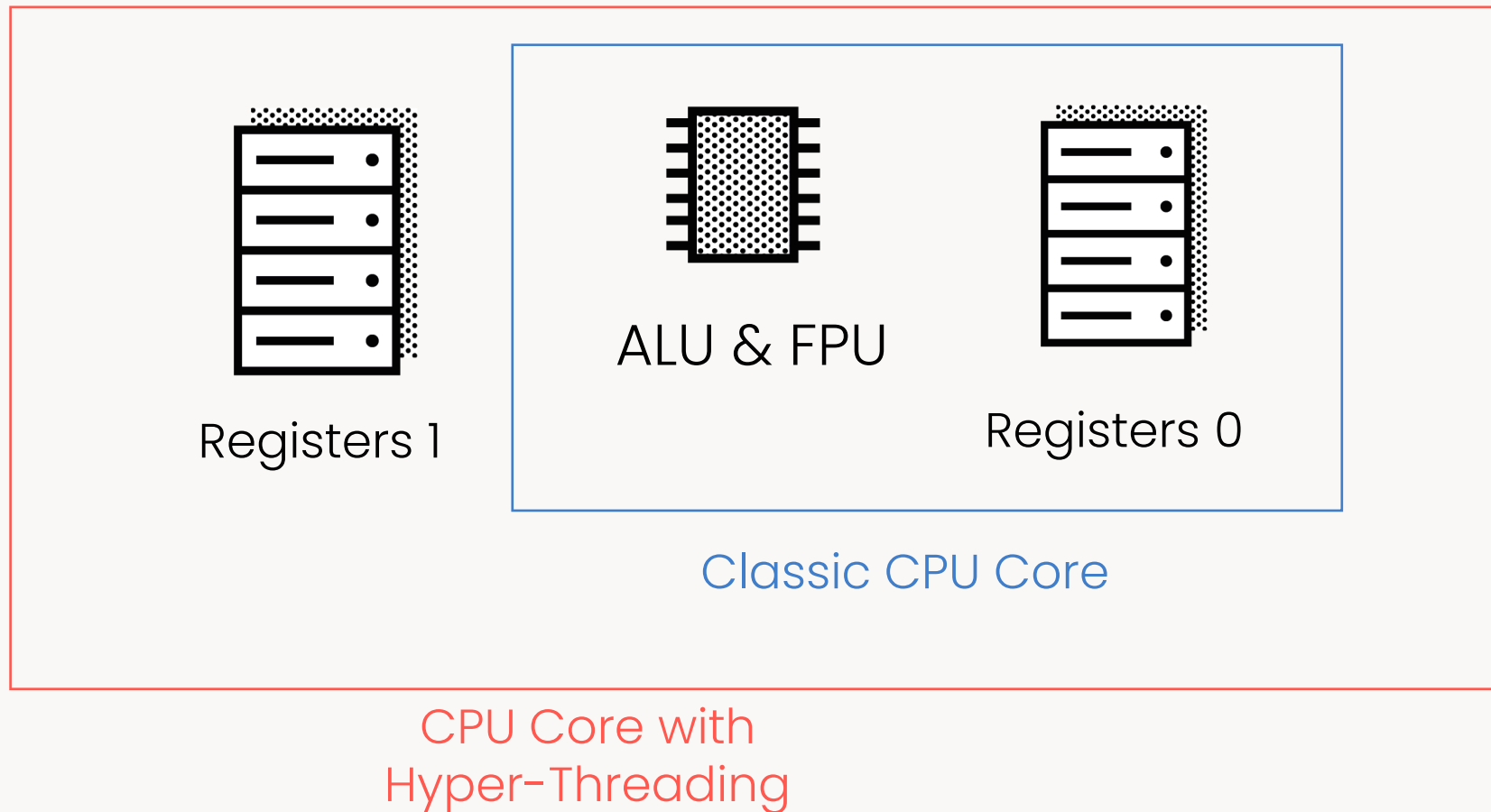
# Threads & deren Overhead



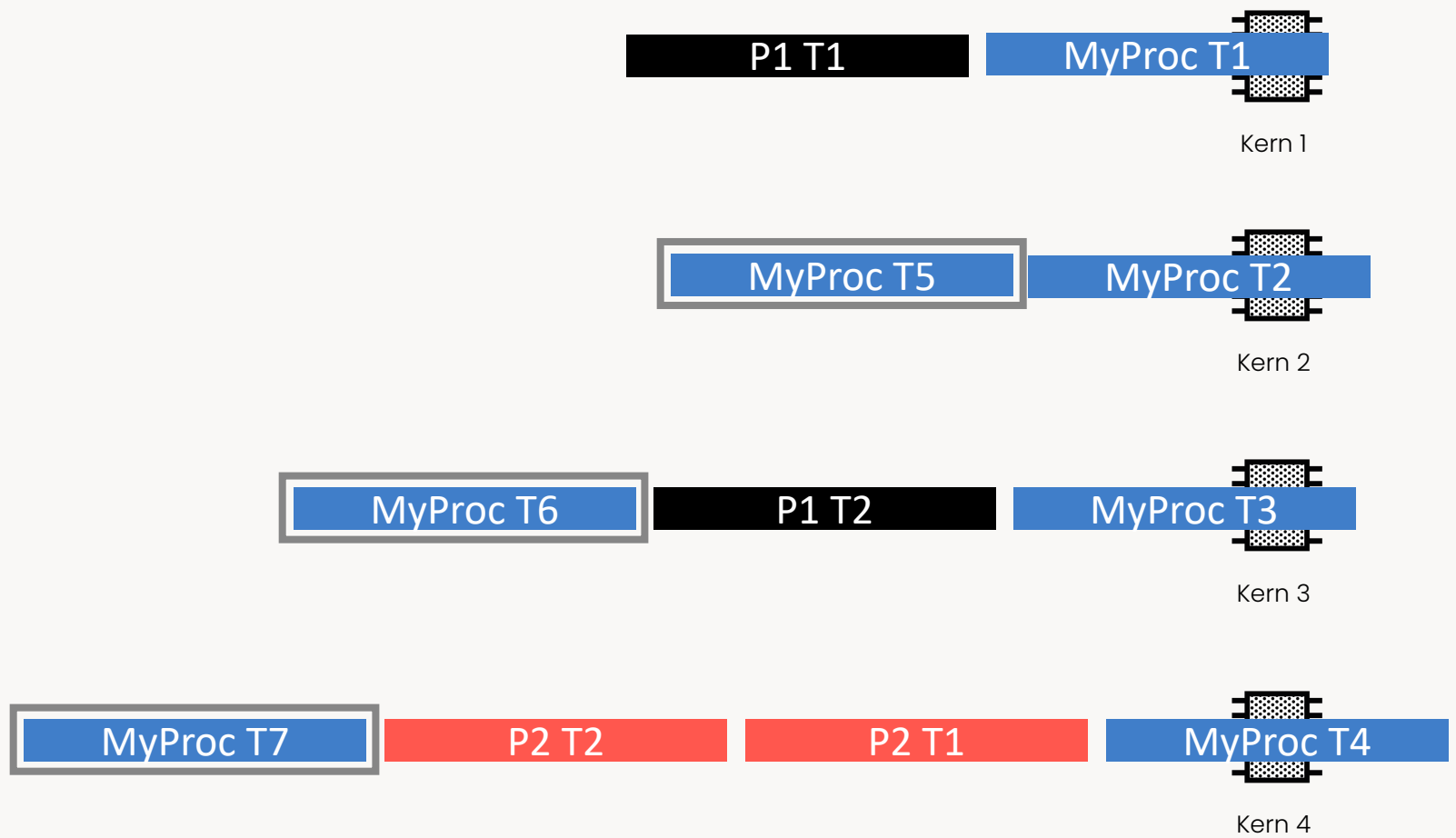
# Threads & deren Overhead

- Context switch
  - Register auslesen und in Thread Kernel-Objekt ablegen
  - Nächsten Thread aussuchen
    - Wenn anderer Prozess: Adressraum austauschen (Stack & Heap)
  - Register von neuen Thread Kernel-Objekt auslesen und auf CPU setzen
  - Schedule-Counter hochzählen etc.
  - Nach dem Switch hat die CPU Cache misses und muss den Cache neu befüllen
  - Nach ein paar ms das gleiche Spiel wieder
  - Hyperthreading hilft, den Overhead zu minimieren

# Hyperthreading



# Threads & deren Anzahl



# CPU & Zeit

oder: Wenn ein CPU-Cycle eine Sekunde wäre

- Object Allocation: 15ns ~1 min
- Quantum: 20ms ~2 Jahre
- Server-Quantum: ~12 Jahre

1 CPU cycle	0.3 ns	1 s
Level 1 cache access	0.9 ns	3 s
Level 2 cache access	2.8 ns	9 s
Level 3 cache access	12.9 ns	43 s
Main memory access	120 ns	6 min
Solid-state disk I/O	50-150 µs	2-6 days
Rotational disk I/O	1-10 ms	1-12 months
Internet: SF to NYC	40 ms	4 years
Internet: SF to UK	81 ms	8 years
Internet: SF to Australia	183 ms	19 years
OS virtualization reboot	4 s	423 years
SCSI command time-out	30 s	3000 years
Hardware virtualization reboot	40 s	4000 years
Physical system reboot	5 m	32 millenia

# Zusammenfassung: Prozesse & Threads


- Was ist ein Thread?
- Overhead von Threads
  - Erzeugen
  - Beenden
  - Kontextwechsel
- Warum exakt ein Thread pro Kern ideal ist



# Synchroner vs. Asynchroner I/O

# Synchroner I/O

```
var fs = new FileStream("test.txt", FileMode.OpenOrCreate,  
    | FileAccess.ReadWrite, FileShare.None, 4096, FileOptions.None);  
var data = new byte[1024];  
fs.Read(data, 0, data.Length);
```

- Allokiert Datenstruktur I/O Request Packet (IRP, ~100 bytes)
  - Handle, offset, bytes in file, address byte[]... etc.
- Thread springt in Kernel-Mode
  - IRP wird an Gerätetreiber übergeben
- Thread blockiert während Hardware arbeitet
- 
- Hardware triggert Interrupt auf Bus, Treiber gibt IRP zurück
- Thread wird unblocked (Wait-Bedingung erfüllt)
- Thread wird wieder ge-scheduled und kann weiterlaufen

ReadFile API: <https://docs.microsoft.com/en-us/windows/win32/api/fileapi/nf-fileapi-readfile>

# Asynchroner I/O

```
var fs = new FileStream("test.txt", FileMode.OpenOrCreate,  
    FileAccess.ReadWrite, FileShare.None, 4096, FileOptions.Asynchronous);  
var data = new byte[1024];  
var readingTask = fs.ReadAsync(data, 0, data.Length);
```

- Allokiert IRP
- Thread springt in Kernel-Mode
  - IRP wird an Gerätetreiber übergeben
- Thread blockiert **NICHT** während Hardware arbeitet
  - Gibt stattdessen einen laufenden Task zurück
- ⌚ ... Hardware triggert Interrupt und gibt IRP zurück
- Task wird auf Completed gestellt



```
while (!readingTask.IsCompleted)  
{  
    WorkOnOtherTasks();  
}  
  
DoSomethingWithData(data);
```

# Demo

# Asynchronität in .NET

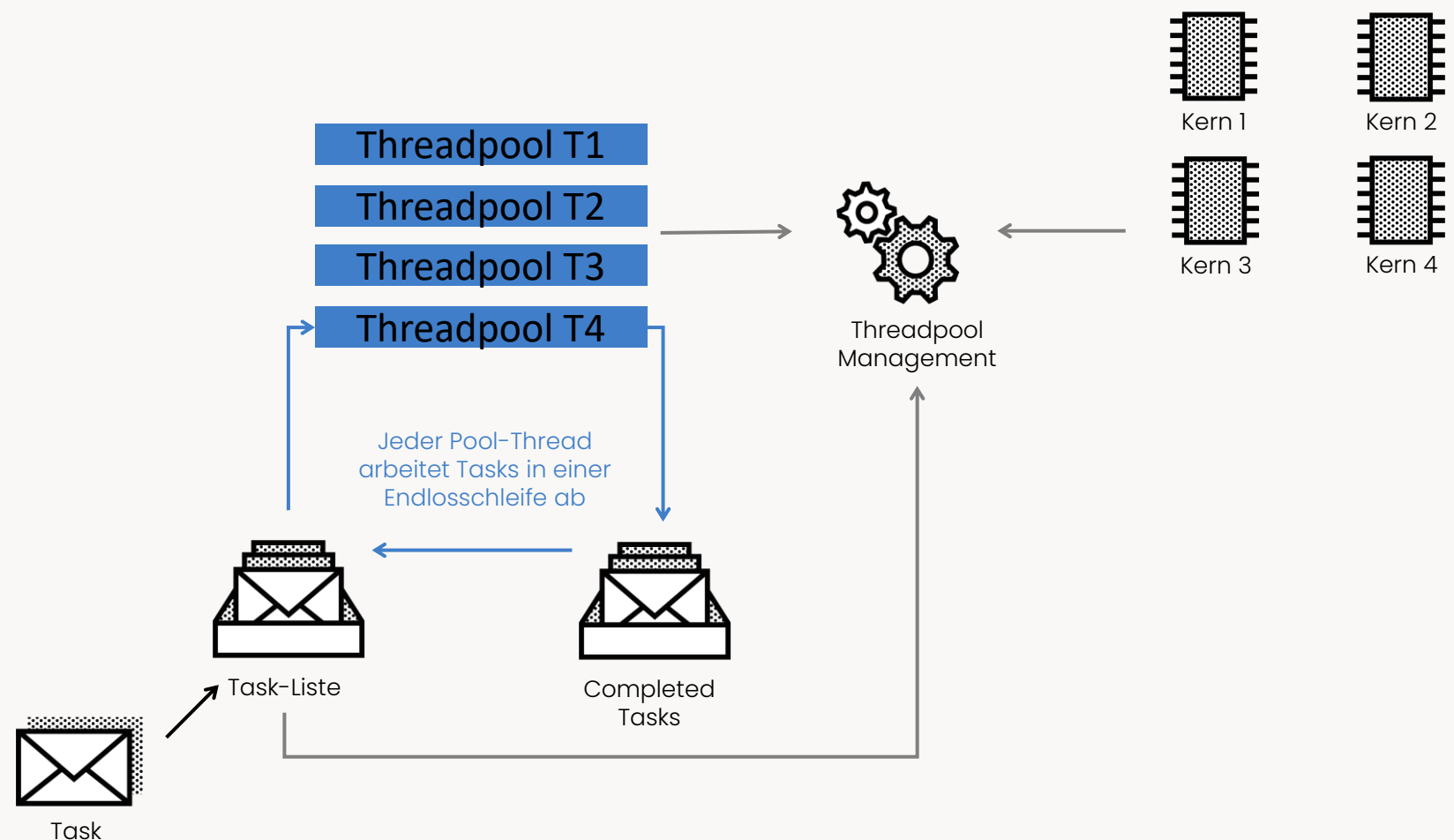
- APM: Asynchronous Programming Model (.NET 1.0)
  - **`IAsyncResult BeginSomething()`**
  - **`EndSomething(result)`**
- EAP: Event-based Asynchronous Pattern (.NET 2)
  - **`DoSomethingAsync()`**
  - **`event SomethingCompleted`**
- TAP: Task-based Asynchronous Pattern (.NET 4)
  - **`Task<T> DoSomethingAsync()`**

# .NET Threadpool & Tasks

# Task

- Abstrahiert Threads für uns
- Funktion bzw. Callback
  - mit Metadaten (`ExecutionContext`)
    - z.B. Thread Culture, Principal & Permissions etc.
  - Kann (mit Bedingungen) an andere Tasks gehängt werden
    - `.ContinueWith()`
- “Kleines Stück Code”, das “irgendwo” laufen kann

# Threadpool & Tasks





# ThreadPool

- Der .NET ThreadPool hilft uns beim Thread-Handling
  - Verwaltet Anzahl der Hintergrund-Threads für uns
  - inkl. Erzeugen, Freigeben, Exception/Handling etc.
- Feature, not a bug:
  - Wenn ein ThreadPool-Thread blockiert, wird ein Neuer erzeugt 🍄
- When to Task:
  - Es wird nach Möglichkeit versucht, Tasks direkt auszuführen  
Nur wenn ein Task blockiert / yielded, springt ein anderer Thread ein

# `async & await`

# async & await Keywords

- Lassen unseren Code auf andere Tasks warten ohne zu blockieren
- **async Task<> MyMethodAsync()**
  - Compiler-Magic
  - Erzeugt eine State-Machine aus unserem Code
- **await OtherMethodAsync()**
  - Generiert einen neuen State / Synchronization-Point für jedes await
- Unmittelbar an jedem **await**
  - aufgerufenen Task starten
  - Zustand speichern
  - wenn anderer Task sofort completed, mit dessen Ergebnis weiterlaufen
  - sonst neuen Task zurückgeben, der nach dem aufgerufenen Task läuft (Continuation) und danach wieder unsere State-Machine aufruft

# Demos

# Tips & Tricks

# Tips & Tricks

- Blockieren wenn irgend möglich vermeiden
  - `task.GetAwaiter().GetResult()` → ✗
  - `Thread.Sleep()` → ✗
  - `await Task.Delay()` → ✓
  - `Thread.SpinWait()` (für sehr kurze Wartezeiten, z.B. auf Locks / Mutexe etc.)
  - blocking I/O → ✗
  - async I/O → ✓

# Tips & Tricks

- Wenn möglich & sinnvoll: parallelisieren
  - `var task1 = DoSomethingAsync();`  
`var task2 = DoSomethingElseAsync();`  
`await Task.WhenAll(task1, task2);`
- Threadpool nicht "austrocknen" lassen
  - Longrunning Tasks als solche markieren!
    - `Task.Factory.StartNew(..., TaskCreationOptions.LongRunning);`
    - Erzeugt einen neuen Thread für diesen Task
  - Wenn blockieren sein muss, dann am besten in einem Task
    - Es kommt automatisch ein neuer Thread nach

# Best Practices

- Kein Sync-over-Async
- Kein “async void”, Exceptions können den Prozess beenden
- Konsequenter `xxxAsync` – Suffix verwenden
- `CancellationToken`s immer weiterreichen



# Fazit

# Fazit

- Asynchroner I/O erlaubt es die CPU auszulasten, anstelle sie warten zu lassen
- Web APIs haben meist
  - vorne Netzwerk I/O
  - hinten Netzwerk (Datenbank, Cache, Queues) oder Datei I/O
  - Nicht-blockender asynchroner I/O ist damit i.d.R. von vorne bis hinten möglich
- 1 (lauffähiger) Thread pro Kern ist ideal
- .NET Threadpool handhabt Threads sehr vernünftig

# Fazit

- Viele kleine Tasks lasten die verfügbare CPU-Zeit bestmöglich aus
  - Blocking verschenkt CPU-Zeit
- `async/await` hilft, Code in viele kleine Tasks zu zerlegen
- Spart Ressourcen und damit Kosten
  - Weniger benötigte Hardware oder weniger hohe Azure-Rechnung 😊

# Web-API mit `async/await`:

Wobei hilft's und wie funktioniert's?

think  
tecture

## Slides & Code

zu finden auf

<https://www.thinktecture.com/de/sebastian-gingter>

Sebastian Gingter

[sebastian.gingter@thinktecture.com](mailto:sebastian.gingter@thinktecture.com)

Consultant

