

Trendyol Group

trendyol^{.com}

trendyol
tech

trendyol
express



Asenkron ve Paralel Programlama, Kotlin Coroutine on Server-Side

Caner Patır

Software Engineer @trendyoltech



github.com/canerpatir



[@canerpatir](https://medium.com/@canerpatir)

İçerik

- Asenkron ve paralel programlama
 - CPU yoğun vs I/O yoğun işler
 - Non-blocking I/O
 - Reactive programlama
- Kotlin ile asenkron programlama
 - Coroutine
 - Thread vs coroutine
 - Motivasyon
 - avoid callback hell
 - **suspend** functions
 - Coroutine builders ve Dispatchers
 - Parallel decomposition
 - Kotlin ve spring boot reactive stack
- Değer
- Case Study - Accelerate your service with Kotlin Coroutine, Spring Boot Reactive Stack
- Kaynaklar

Parallel vs Concurrent vs Multi-threaded vs Async ?



Parallel == Multi-threaded vs Concurrent == Async ?

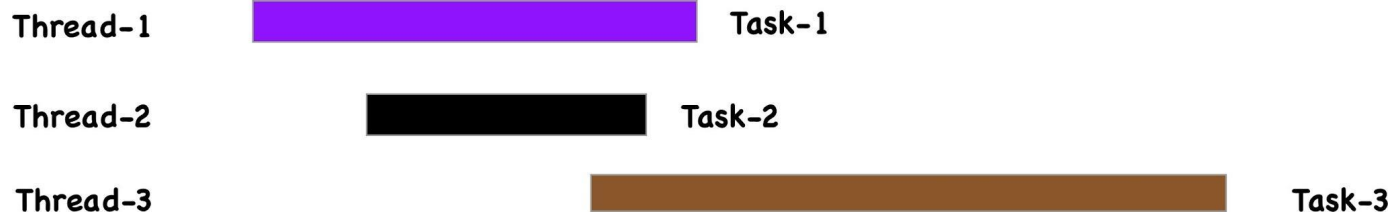
- Synchronous
 - Birden fazla işin farklı zaman dilimlerinde art arda icra edilmesi
- Parallelism (Multi-threading)
 - Birden fazla işin, farklı işlem birimleri(**thread**) üzerinde aynı zaman diliminde icra edilmesi
 - Multi CPU(ya da core) ortamlar gerekir
- Asynchronous (Concurrency)
 - Birden fazla işin örtüşen zaman dilimlerinde, birbirini engellemeden icra edilmesi
 - Context-switching ile işler belli bir senkronizasyonla örtüşen zamanlı olarak işlenir
 - Tek CPU ortamda da concurrency sağlanabilir (örn: NodeJs)

Parallel == Multi-threaded vs Concurrent == Async ?

- Synchronous



- Multi-threaded = Parallelism



- Asynchronous = Concurrency

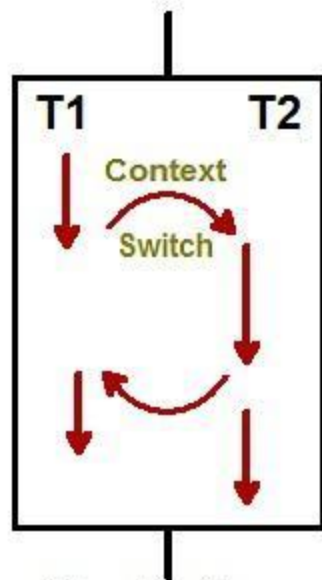




ForGIFs.com

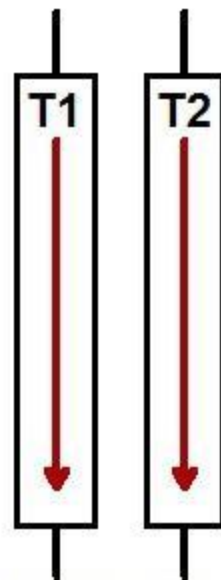


Concurrent



Single Core

Parallel



Multiple Core

S: Ne zaman asenkron ne zaman paralel yaklaşım tercih ederiz ?

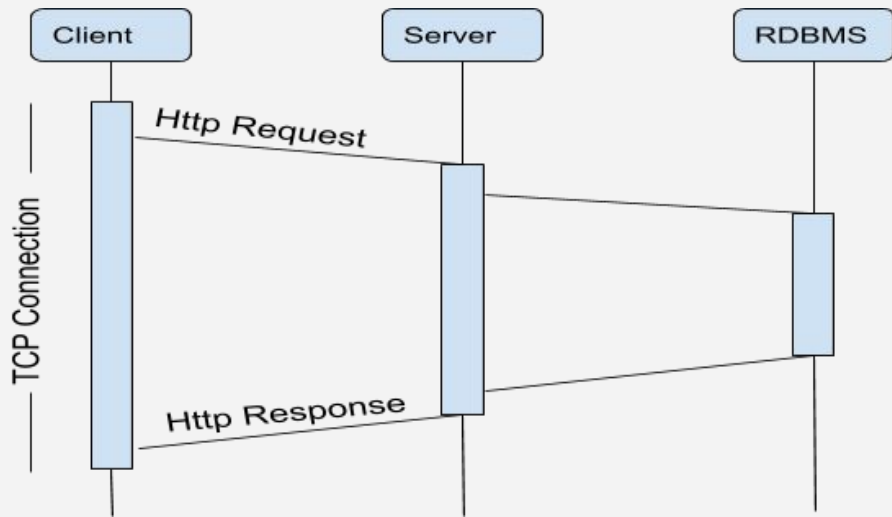
CPU yoğun vs IO yoğun işler

- CPU yoğun işler
 - Process, zamanının çoğunu CPU'yu kullanarak geçirir
 - Başka bir deyişle yoğun aritmetik ve mantık operasyonu içerir
 - Görsel işleme, hash, kriptografi vs.
- IO yoğun işler
 - Process, zamanın çoğunu I/O operasyonları ile geçirir
 - CPU kullanım süreleri düşüktür
 - DB'den yaptığımız okuma yazmalar, bütün network operasyonları (http çağrıları, TCP write/read vs)

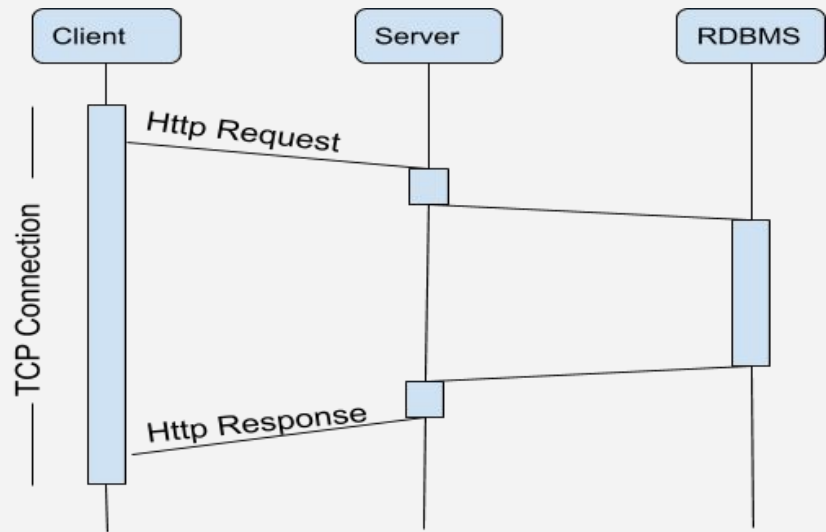
Non-blocking I/O

- I/O işlemi (DB write/read, API call vs.) esnasında thread'in bloklanmayarak başka görevlere tahsis edilmesi ve CPU'nun yüksek verimlilikle kullanılması esasına dayalı bir yaklaşımdır.

Blocking I/O



Non-blocking I/O

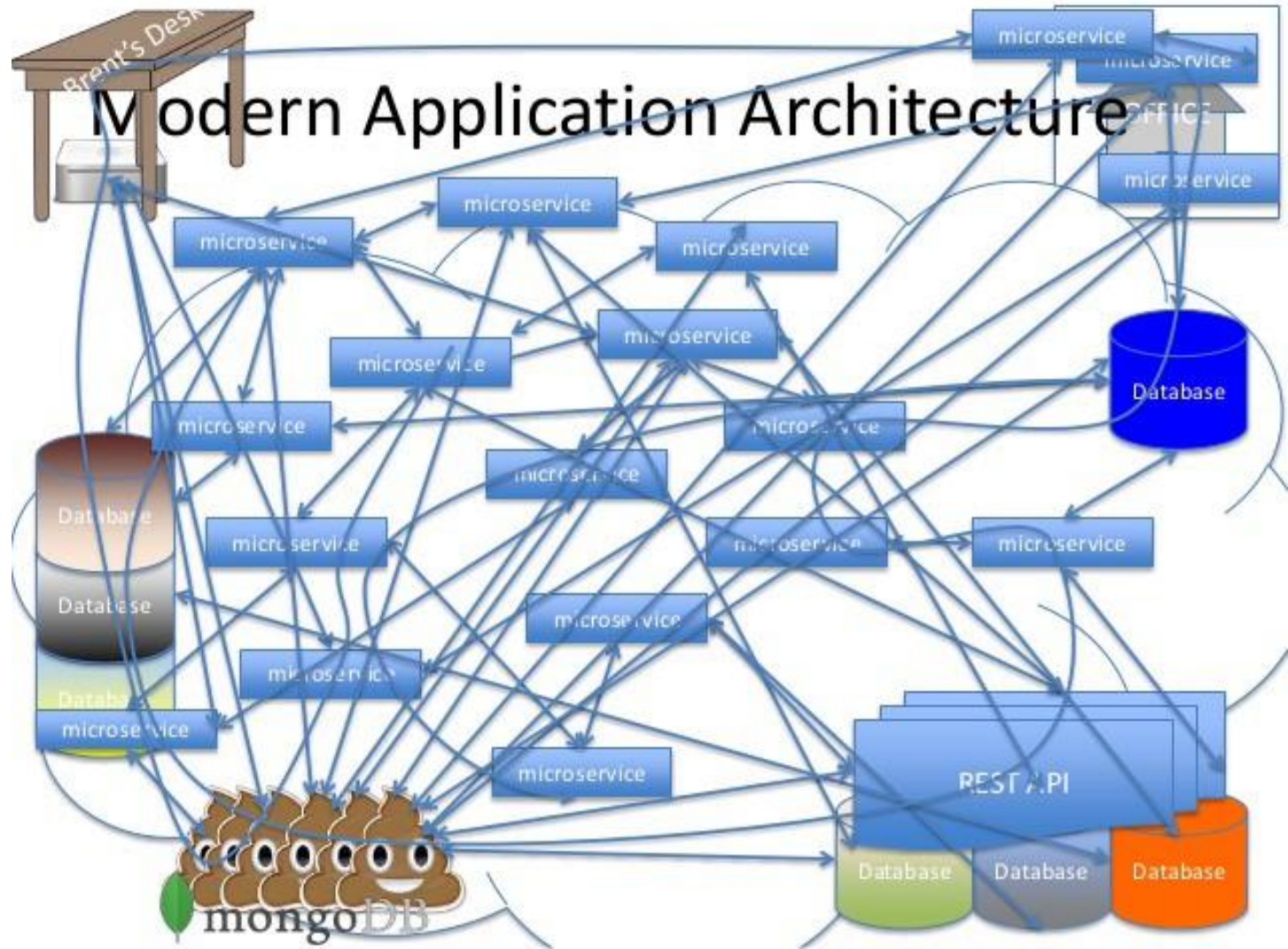


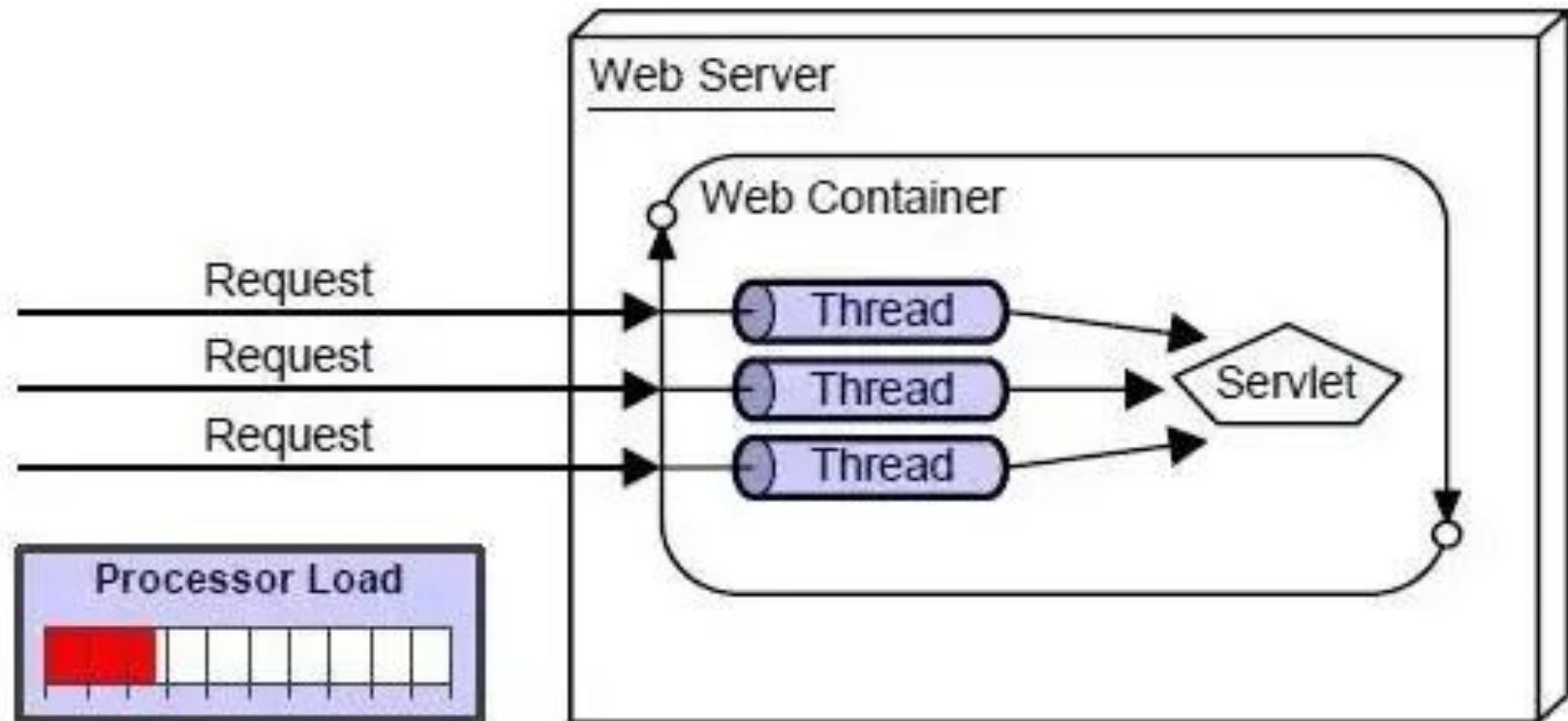
S: Ne zaman asenkron ne zaman paralel yaklaşım tercih ederiz ?

C: IO yoğun işlerde **async**, CPU yoğun işlerde **parallelism**




Modern Application Architecture





S: Peki bir IO operasyonu nasıl asenkron hale getirilir ?

A close-up shot of a man with dark, curly hair and a mustache, wearing a maroon shirt. He is holding a black flip phone to his ear with his left hand, which also has a gold ring on the ring finger and a black wristband. He has a serious expression. The background is an outdoor setting with a building on the left, green hills in the distance, and a cloudy sky.

I'll call back soon, all right?

Reactive Programlama

- Kod akışı yerine, olayları (event) takip ettiğimiz bir programlama modelidir.
- HTTP istekleri, notifikasyonlar, sensor verileri, kullanıcı hareketleri (click stream) , bellekteki bir değişkenin değerinin değişmesi gibi konular olaylara örnektir
- Reaktif uygulamalar aynı anda birden çok olayı takip ederek pozisyon alabilme yeteneğine sahiptir
- Bu yönüyle asenkron programlama için uygun bir modeldir

JVM ekosistemindeki reactive oyuncular

Library level



RxJava



Project
Reactor

Framework level



VERT.X



Kotlin

```

public class JavaArticle {
    private final int id;
    private final String title;
    private final String content;
    private final int likes;

    public JavaArticle(final int id, final String title, final String content, final int likes) {
        this.id = id;
        this.title = title;
        this.content = content;
        this.likes = likes;
    }

    public int getId() {
        return id;
    }

    public String getTitle() {
        return title;
    }

    public String getContent() {
        return content;
    }

    public int getLikes() {
        return likes;
    }

    @Override
    public boolean equals(final Object o) {
        if (this == o) return true;
        if (o == null || getClass() != o.getClass()) return false;

        JavaArticle that = (JavaArticle) o;

        if (id != that.id) return false;
        if (likes != that.likes) return false;
        if (!title.equals(that.title)) return false;
        return content.equals(that.content);
    }

    @Override
    public int hashCode() {
        int result = id;
        result = 31 * result + title.hashCode();
        result = 31 * result + content.hashCode();
        result = 31 * result + likes;
        return result;
    }

    @Override
    public String toString() {
        return "JavaArticle{" +
            "id=" + id +
            ", title='" + title + '\'' +
            ", content='" + content + '\'' +
            ", likes=" + likes +
            '}';
    }
}

```

```

data class KotlinArticle (
    val id: Int,
    val title: String,
    val content: String,
    val likes: Int
)

```

```
package com.example.chrisnielsen.myapplication;

import java.util.List;

public class Test {

    public Test(int index, List<String> myList) {
        this._index = index;
        this._myList = myList;
    }

    private int _index;
    public void setIndex(int value) {
        _index = value;
    }
    public int getIndex() {
        return _index;
    }

    private List<String> _myList;
    public void setMyList(List<String> value) {
        _myList = value;
    }
    public List<String> getMyList() {
        return _myList;
    }

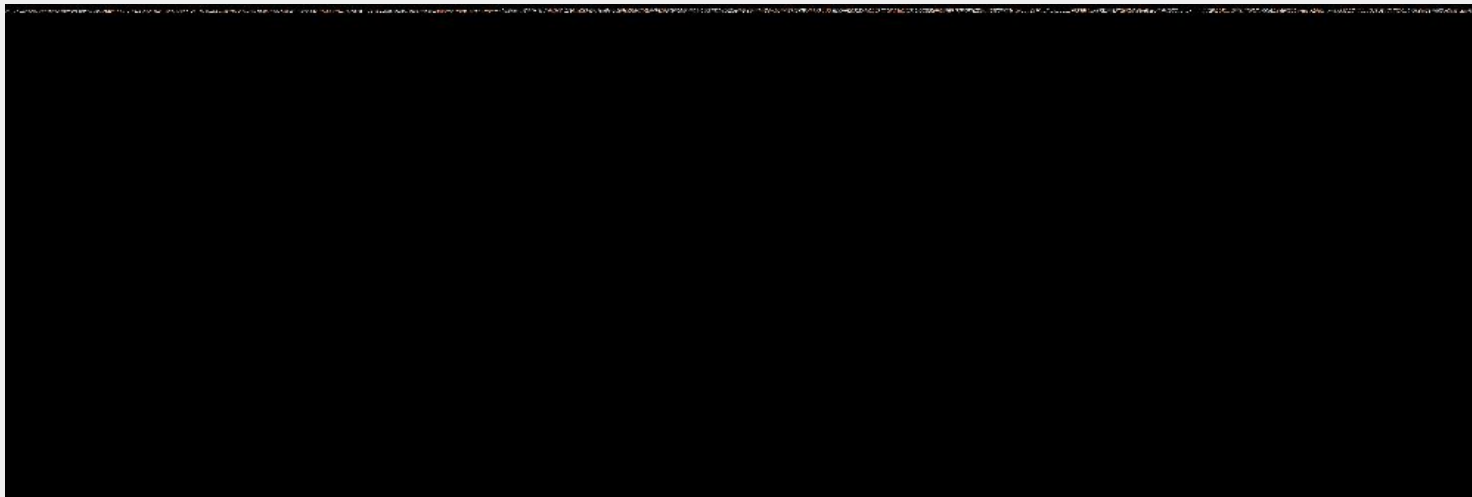
    public void startProgress() {
        Runnable runnable = new Runnable() {
            @Override
            public void run() {
                for (int i = 0; i <= 10; i++) {
                    _index = i;
                }
            }
        };
        new Thread(runnable).start();
    }
}
```



Kotlin

Kotlin Coroutine?

- Threadler tarafından işletilen, gerektiğinde suspend edilebilen asenkron görev parçalarıdır
- Gerektiğinde suspend edilebilirler
- Suspend edildiklerinde CPU'yu bloklamazlar



**Coroutines are Kotlin lightweight threads
allowing to write non-blocking code in an
imperative way**



Light-weight thread ?

Nereden esinlendi ?

- **async/await TPL - C#**
- **goroutine ve channel - Go**

Coroutine != Thread

Thread → Blocking

Coroutine → Suspending

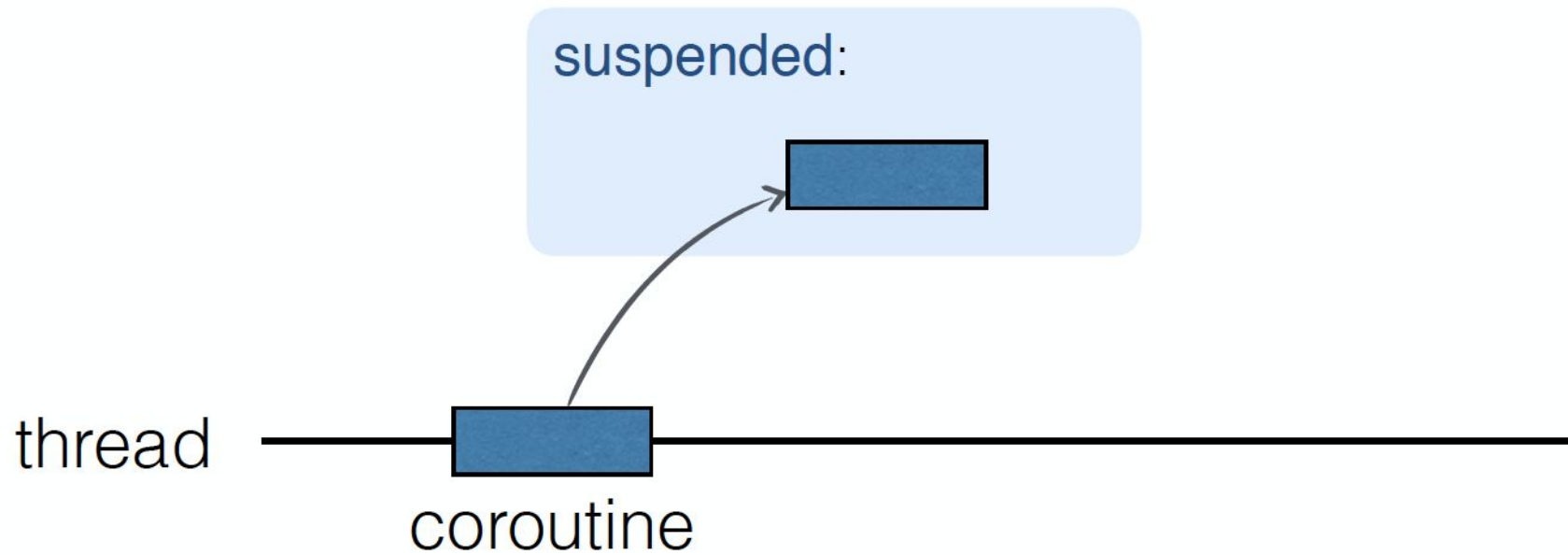
Context switching

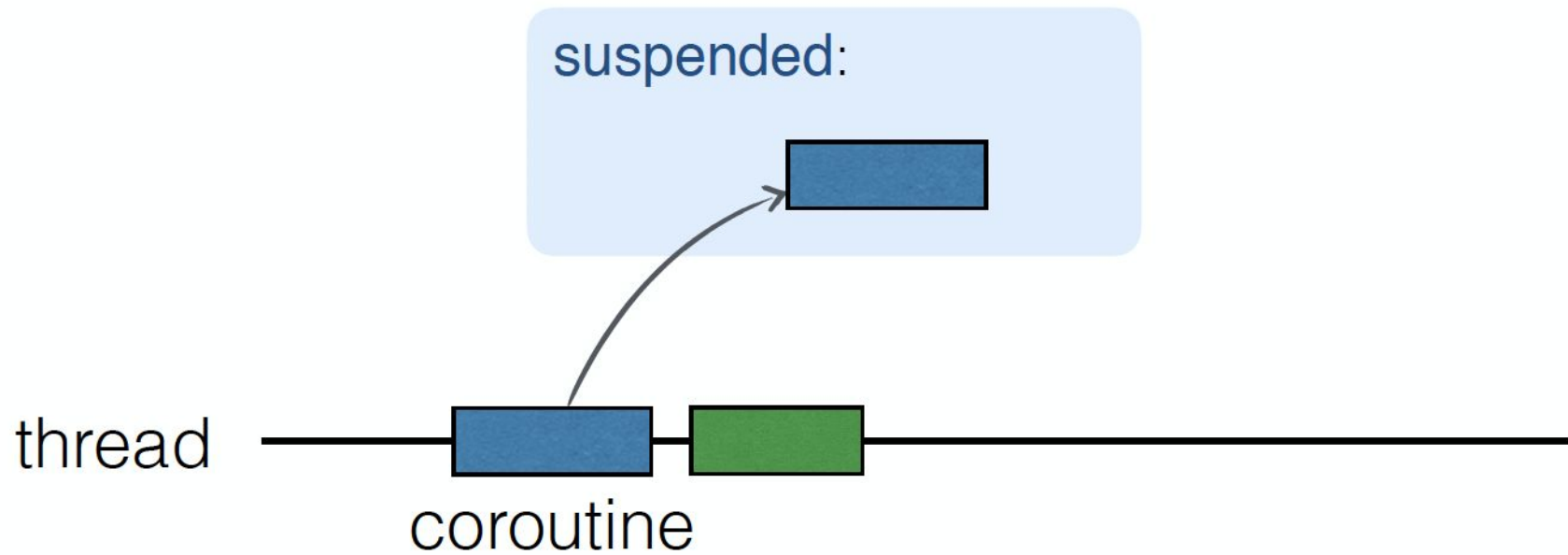
- Coroutine uzun sürecek bir I/O operasyonu başladığında **suspend** edilir
- Coroutine suspend edildiğinde thread başka bir coroutine i işletmeye başlar
- I/O işlemi bittiğinde ilgili coroutine **herhangi bir thread** tarafından devralınıp hayatına devam eder
- Bu devir teslim sürecine context switching denir

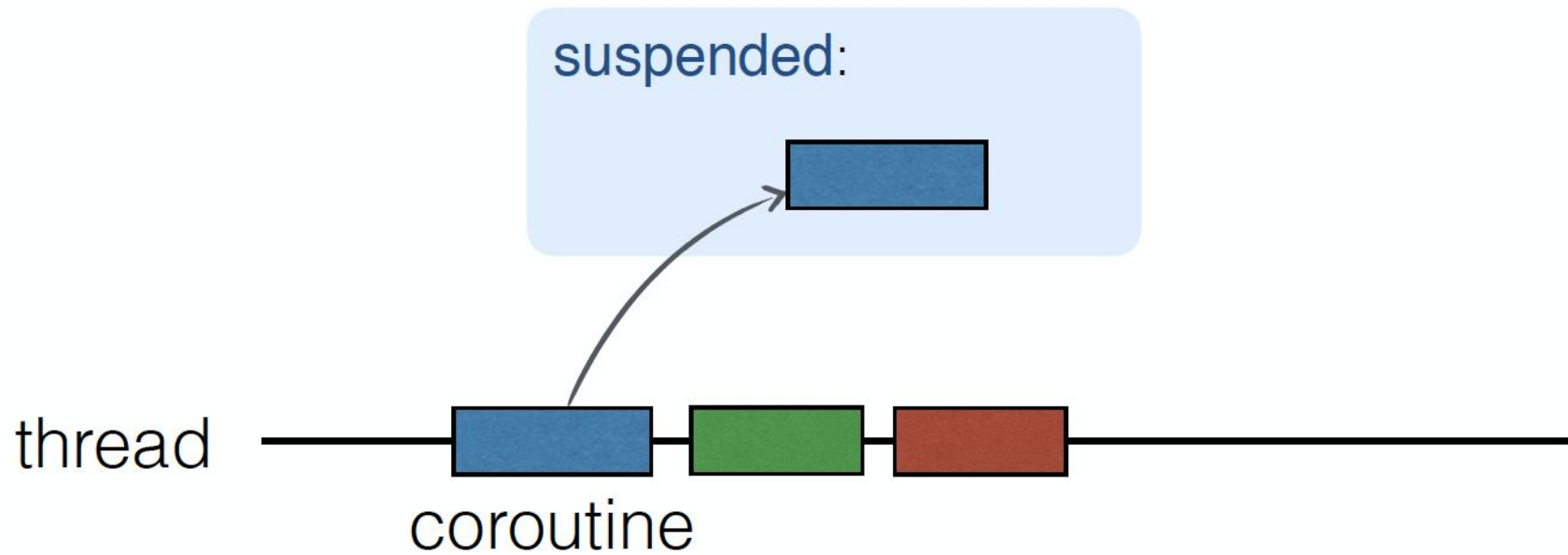
thread

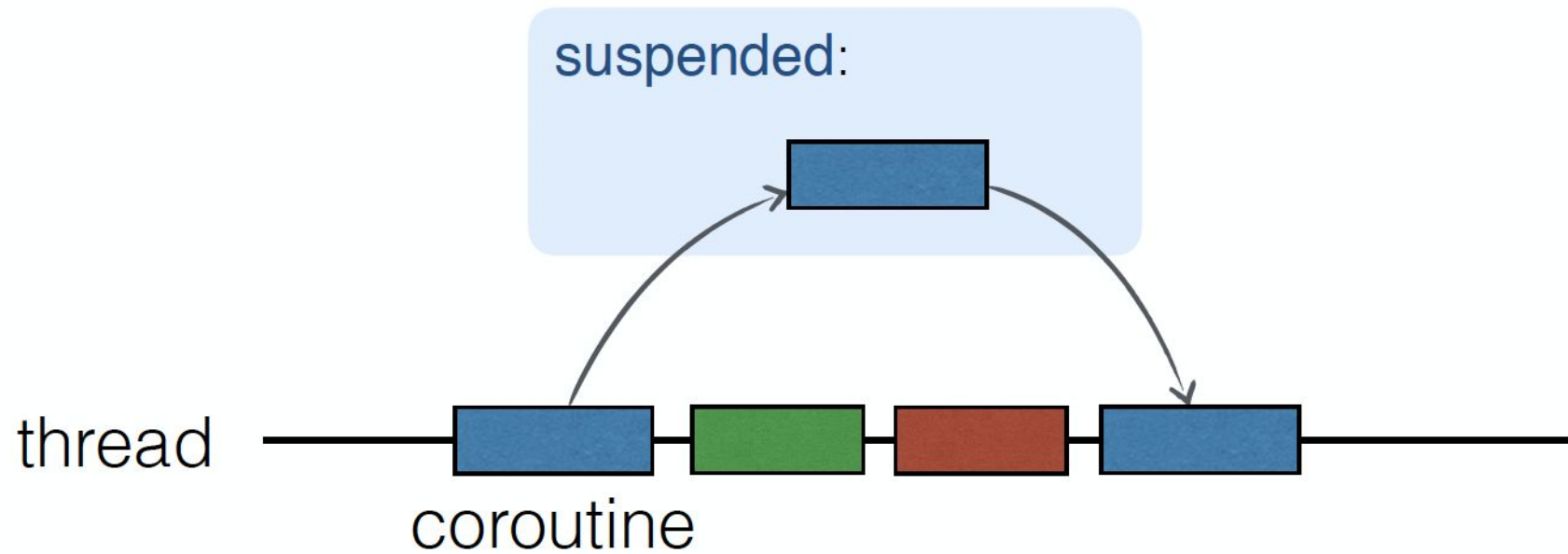


coroutine









Motivasyon

- Asenkron modelin ürettiği, kullanıcı deneyimi, kaynak optimizasyonu gibi değerlerden faydalanmak
- Bu değerlerden faydalanırken alışık olduğumuz **sequential declarative** kodlama tarzından vazgeçmemek
- “**Callback hell**” den kaçınmak (bkz. continuous passing style programming)

callback hell

```
fun getProduct(id: Long): Product {...}  
fun getCategory(id: Long): Category {...}  
fun saveProduct(product: Product) {...}  
  
fun updateProductCategory(id: Long, categoryId: Long) {  
    val product = getProduct(id)  
    val category = getCategory(categoryId)  
    category.validate()  
    product.category = category  
    saveProduct(product)  
}
```

callback hell - CompletableFuture

```
fun getProduct(id: Long): CompletableFuture<Product> {...}
fun getCategory(id: Long): CompletableFuture<Category> {...}
fun saveProduct(product: Product): CompletableFuture<Void> {...}

fun updateProductCategory(id: Long, categoryId: Long) {
    getProduct(id)
        .thenApply { product ->
            getCategory(categoryId).thenAccept { category ->
                category.validate()
                product.category = category
            }
            saveProduct(product)
        }
    }.join()
}
```

callback hell - rxJava

```
fun getProduct(id: Long): Single<Product> {...}
fun getCategory(id: Long): Single<Category> {...}
fun saveProduct(product: Product): Single<Void> {...}

fun updateProductCategory(id: Long, categoryId: Long) {
    getProduct(id)
        .subscribeOn(Schedulers.io())
        .flatMap { product ->
            getCategory(categoryId)
                .map { category ->
                    category.validate()
                    product.category = category
                    return@map product
                }
        }
        .doOnSuccess { saveProduct(it) }
        .subscribe()
}
```

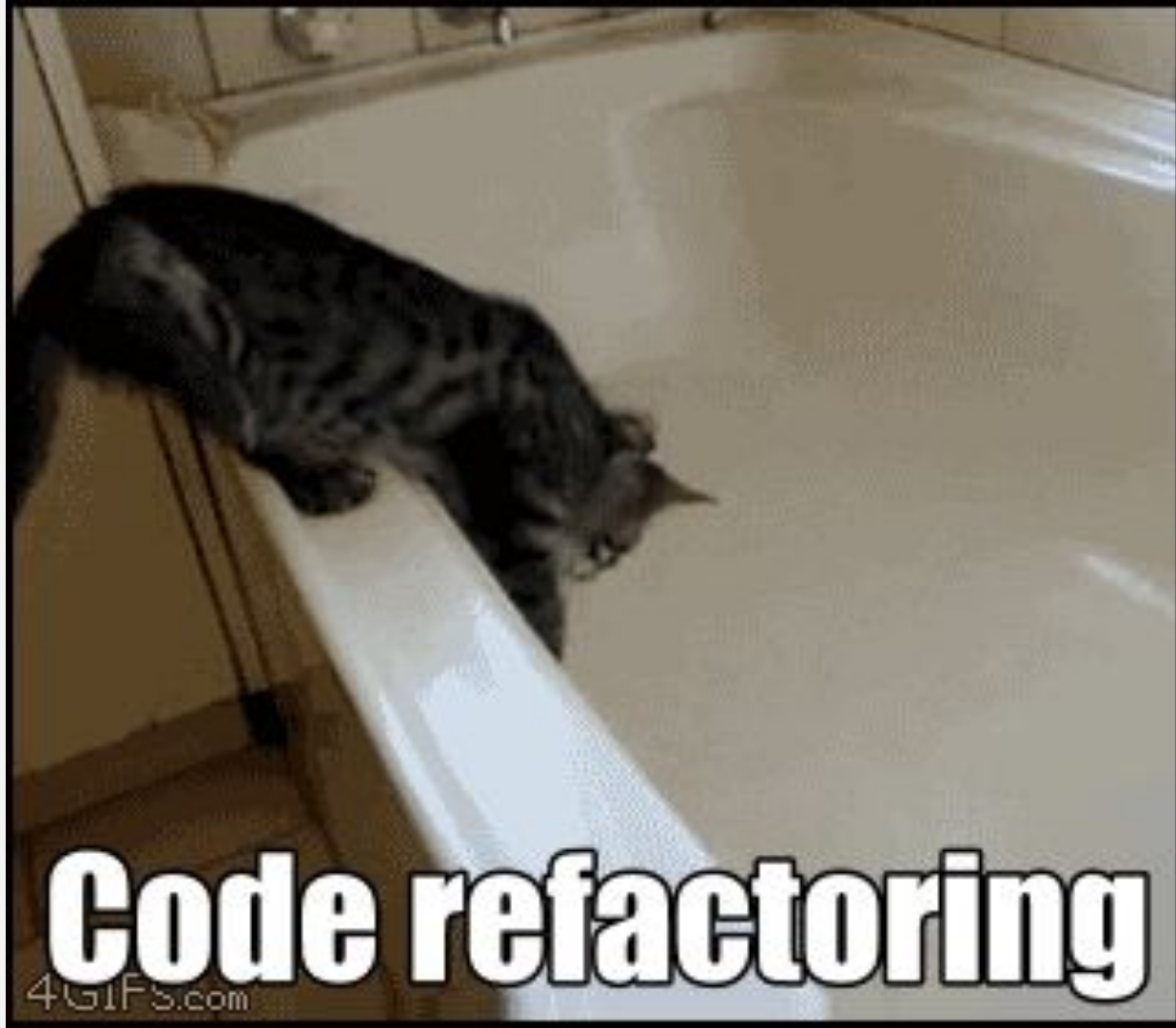
:)

```
fun getProduct(id: Long): Product {...}  
fun getCategory(id: Long): Category {...}  
fun saveProduct(product: Product) {...}  
  
fun updateProductCategory(id: Long, categoryId: Long) {  
    val product = getProduct(id)  
    val category = getCategory(categoryId)  
    category.validate()  
    product.category = category  
    saveProduct(product)  
}
```



```
1 var asyncJavaScript = function(err, callback) {  
2   callback(function(err, callback) {  
3     callback(function(err, callback) {  
4       callback(function(err, callback) {  
5         callback(function(err, callback) {  
6           callback(function(err, callback) {  
7             callback(function(err, callback) {  
8               callback(function(err, callback) {  
9                 console.error('CALLBACK HELL');  
10              });  
11            });  
12          });  
13        });  
      });  
    });  
  });  
}
```

When you really want it to be synchronous !!



Avoid callback hell - Coroutine (async/await)

```
fun getProduct(id: Long): Deferred<Product> {...}  
fun getCategory(id: Long): Deferred<Category> {...}  
fun saveProduct(product: Product): Deferred<Void> {...}
```

```
fun updateProductCategory(id: Long, categoryId: Long) = async {  
    val product = getProduct(id).await()  
    val category = getCategory(categoryId).await()  
    category.validate()  
    product.category = category  
    saveProduct(product).await()  
}
```

Avoid callback hell - Coroutine

```
fun getProduct(id: Long): Deferred<Product> {...}  
fun getCategory(id: Long): Deferred<Category> {...}  
fun saveProduct(product: Product): Deferred<Void> {...}  
  
fun updateProductCategory(id: Long, categoryId: Long) = async {  
    val product = getProduct(id).await()  
    val category = getCategory(categoryId).await()  
    category.validate()  
    product.category = category  
    saveProduct(product).await()  
}
```

Avoid callback hell - Coroutine suspend function

```
suspend fun getProduct(id: Long): Product {...}  
suspend fun getCategory(id: Long): Category {...}  
suspend fun saveProduct(product: Product) {...}
```

```
suspend fun updateProductCategory(id: Long, categoryId: Long) {  
    val product = getProduct(id)  
    val category = getCategory(categoryId)  
    category.validate()  
    product.category = category  
    saveProduct(product)  
}
```

Avoid callback hell - Coroutine suspend function

```
fun getProduct(id: Long): Product {...}  
fun getCategory(id: Long): Category {...}  
fun saveProduct(product: Product) {...}
```

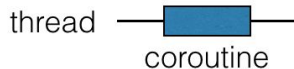
```
fun updateProductCategory(id: Long, categoryId: Long) {  
    val product = getProduct(id)  
    val category = getCategory(categoryId)  
    category.validate()  
    product.category = category  
    saveProduct(product)  
}
```

```
suspend fun getProduct(id: Long): Product {...}  
suspend fun getCategory(id: Long): Category {...}  
suspend fun saveProduct(product: Product) {...}
```

```
suspend fun updateProductCategory(id: Long, categoryId: Long) {  
    val product = getProduct(id)  
    val category = getCategory(categoryId)  
    category.validate()  
    product.category = category  
    saveProduct(product)  
}
```

suspend functions

- Coroutine'lerin belli suspension pointlerde context switchinge girer
- Bu noktaları, derleyiciye geliştirici deklare etmek durumundadır
- Bu deklarasyon, ilgili methodu suspend keywordü ile etiketleyerek sağlanır



suspend functions

```
18 suspend fun someSuspendableOperation(): String {  
19     delay(1000L) // representation of IO operation like going to DB  
20     return "hello"  
21 }
```


suspend functions

- **suspend** functionlar diğer dillerin asenkron yazımdan farklı olarak, geriye özel veri tipleri dönmek zorunda değildir
 - Java — Future, Flux, Mono, Single, Observable ve binlercesi :)
 - Javascript— Promise
 - C# — Task

```
suspend fun someSuspendableOperation(): String
```

suspend functions

- **suspend** fonksiyon, normal fonksiyonlar içerisinde call edilemez

```
14 ▶ fun main(args: Array<String>){  
15   → someSuspendableOperation()  
16 }  
17  
18 suspend fun someSuspendableOperation(): String {  
19   → delay(1000L) // representation of IO operation like going to DB  
20   return "hello"  
21 }
```

Suspend function 'someSuspendableOperation' should be called only from a coroutine or another suspend function

[Make main suspend](#) ↵ [More actions...](#) ↵

suspend functions

- Coroutine scope veya başka bir suspend function içerisinde çağrılabilir

```
20 fun normalFunction() {  
21     GlobalScope.async {  
22         someSuspendableOperation()  
23     }  
24 }  
25  
26 suspend fun someSuspendableOperation(): String {  
27     otherSuspendableOperation()  
28     delay(1000L) // representation of IO operation like going to DB  
29     return "hello"  
30 }  
31  
32 suspend fun otherSuspendableOperation() {  
33     delay(500L) // representation of IO operation like going to DB  
34 }
```

Reactive to suspending

- Eğer client kütüphanemiz reactive ise `kotlinx.coroutines` paketi vasıtasıyla reactive tipleri suspending hale çevirebiliyoruz
- `kotlinx.coroutines` ile rxJava, reactor, flow gibi reactive kütüphane tiplerini suspend function a çeviren extension methodları barındırıyor.



Reactive to suspending

```
suspend fun getByld(id: String): GetResult {  
    val monoResult: Mono<GetResult> = collection.get(id)  
    val result: GetResult = monoResult.awaitSingle()  
    return result  
}
```

// `kotlinx.coroutines` paketi içerisinde

```
suspend fun <T> reactor.core.publisher.Mono<T>.awaitSingle(): T { }
```

S: Coroutine nasıl oluşturulur ?

S: Coroutine nasıl oluşturulur ?

C: Coroutine builders

Coroutine builder

- `kotlinx-coroutines-core` kütüphane fonksiyonlarıdır
- `async { ... }`
- `launch { ... }`
- `runBlocking { ... }`

async { ... }

- Geriye değer dönen asenkron operasyonların işletilmesi için kullanılır
- Beklenen sonucu almak için Deferred dönüş tipi kullanılır
- Deferred, Javadaki Future veya Javascript dilindeki Promise in karşılığıdır diyebiliriz
- Sonucu beklemek için Deferred.await methodunu çağırmak gerekir
- Await bir suspend function'dır

```
22 suspend fun coroutineExample() {  
23     println("[${SimpleDateFormat("hh:mm:ss").format(Date())}] coroutine starting")  
24     // starting new async coroutine scope  
25     var deferred1: Deferred<Int> = CoroutineScope(Dispatchers.IO).async { operation() }  
26     val result: Int = deferred1.await()  
27     println("[${SimpleDateFormat("hh:mm:ss").format(Date())}] result is $result")  
28 }  
29  
30 suspend fun operation(): Int {  
31     delay(3000L) // non-blocking operation  
32     println("[${SimpleDateFormat("hh:mm:ss").format(Date())}] operation finished")  
33  
34     return 10  
35 }
```

```
[07:31:56] coroutine starting  
[07:31:59] operation finished  
[07:31:59] result is 10
```

launch { ... }

- Geriye dönüş tipi gerektirmeyen, arka planda işletilecek operasyonlar için kullanılan builder tipidir.
- Fire and forget mantığı ile çalışabilecek işler için uygundur.

```
22 suspend fun coroutineExample() {  
23     CoroutineScope(Dispatchers.IO).launch { operation() }  
24     print("Hello ")  
25 }  
26  
27 suspend fun operation() {  
28     delay(1000L) // non-blocking operation  
29     print("World! ")  
30 }  
31  
32
```

Hello World!

runBlocking { ... }

- Coroutine içerisinde kullanılmamalıdır.
- Genelde blocking tarzda yazılmış koda coroutine scope bağlamak için ya da test amaçlı kullanılır.
- Normal dünya ile suspending dünya arasında köprü görevi görür
- Bütün kod bloğu uçtan uca non-blocking tarzda yazılmışsa kullanımına gerek yoktur.

```
19 ▶ fun main() = runBlocking {  
20     println("[${(SimpleDateFormat("hh:mm:ss")).format(Date())}] operation1 starting")  
21     val result: Int = operation()  
22     println("[${(SimpleDateFormat("hh:mm:ss")).format(Date())}] operation1 finished result is $result")  
23 }  
24  
25 suspend fun operation(): Int {  
26     delay(2000L) // simulated non-blocking operation  
27     return 50  
28 }
```

```
[07:02:57] operation1 starting  
[07:02:59] operation1 finished result is 50
```

parallel decomposition

```
22 suspend fun parallelDecompositionExample() {
23     ↗ val deferred1 = CoroutineScope(Dispatchers.IO).async { operation1() }
24     ↗ val deferred2 = CoroutineScope(Dispatchers.IO).async { operation2() }
25     println("${SimpleDateFormat("hh:mm:ss").format(Date())} Awaiting computations...")
26     ↗ val result = deferred1.await() + deferred2.await()
27     println("${SimpleDateFormat("hh:mm:ss").format(Date())} The result is $result")
28 }
29
30 suspend fun operation1(): Int {
31     ↗ delay(2000L) // simulated computation
32     println("${SimpleDateFormat("hh:mm:ss").format(Date())} operation1 finished")
33     return 50
34 }
35
36 suspend fun operation2(): Int {
37     ↗ delay(1000L)
38     println("${SimpleDateFormat("hh:mm:ss").format(Date())} operation2 finished")
39     return 60
40 }
```

```
[06:37:22] Awaiting computations...
[06:37:23] operation2 finished
[06:37:24] operation1 finished
[06:37:24] The result is 110
```

Dispatchers

- Coroutineler CPU açısından hiçbir anlam ifade etmeyen basit görev abstractionlarıdır
- Dispatcher, coroutinein yürütülmesinden sorumludur
- T anında bir coroutine'i alıp threadpoolda bir thread'e atayarak işletilmesini sağlar ve gerektiğinde coroutine'i suspend ederek threadin başka bir coroutine yürütmesini sağlayabilir
- Dispatcherlar coroutine'leri CPU için anlamlı hale getirmiş olur diyebiliriz.
- RxJava'daki scheduler nesnesinin eşdeğeri

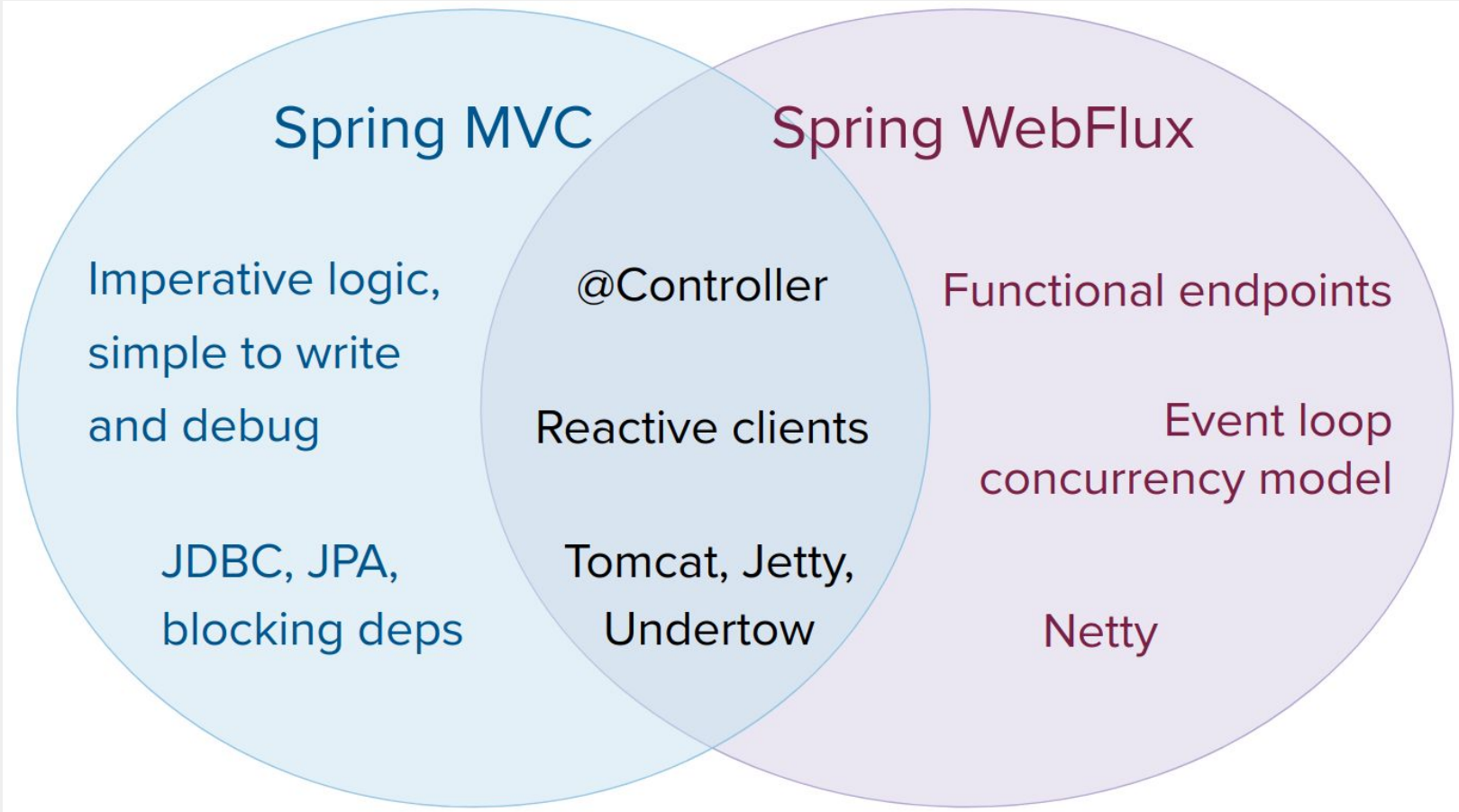
Dispatchers

- **Dispatchers.Default:** Herhangi bir dispatcher belirtilmemişse, tüm coroutine üreticileri tarafından kullanılan varsayılan dispatcherdir. CPU yoğun görevlerin yürütülmesi için uygun seçimdir. (*RxJava'daki Schedulers.Computation*)
- **Dispatchers.IO:** I/O yoğun işlemler için kullanılır. (*RxJava'daki Schedulers.IO*)

Kotlin and Spring Boot



Spring MVC vs Spring Webflux



Spring MVC vs Spring Webflux

Spring MVC

Servlet API

Blocking I/O

Tomcat, Jetty, ...

Spring WebFlux

Spring Web API

Reactor, Reactive Streams

Non-blocking I/O

Netty

Tomcat, Jetty, ...

Kotlin, Spring Boot

Generate a Maven Project ▾ with Kotlin ▾ and Spring Boot 2.0.1 ▾

Project Metadata

Artifact coordinates

Group

com.example



Artifact

demo

Dependencies

Add Spring Boot Starters and dependencies to your application

Search for dependencies

Web, Security, JPA, Actuator, Devtools...

Selected Dependencies

Reactive Web ✕

Reactive MongoDB ✕

Embedded MongoDB ✕

Kotlin, Coroutine, Spring Boot Reactive Stack



Kotlin, Coroutine, Spring Boot Reactive Stack

// Reactive Controllers

@RestController

class ProductController {

@Autowired

lateinit var productRepository: ProductRepository

@GetMapping("/{id}")

fun findOne(@PathVariable id: Int): Mono<Product> {
 return productRepository.getProductById(id)
 }
}

@Repository

class ProductRepository(private val client: DatabaseClient) {

fun getProductById(id: Int): Mono<Product> {
 return client.execute()
 .sql("SELECT * FROM products WHERE id = \$1")
 .bind(0, id)
 .as `(Product::class.java)`
 .fetch()
 .one()
 }
}

// Controllers with Coroutines

@RestController

class ProductControllerCoroutines {

@Autowired

lateinit var productRepository: ProductRepositoryCoroutines

@GetMapping("/{id}")

suspend fun findOne(@PathVariable id: Int): Product? {
 return productRepository.getProductById(id)
 }
}

@Repository

class ProductRepositoryCoroutines(private val client: DatabaseClient) {

suspend fun getProductById(id: Int): Product? =
 client.execute()
 .sql("SELECT * FROM products WHERE id = \$1")
 .bind(0, id)
 .as `(Product::class.java)`
 .fetch()
 .one()
 .awaitFirstOrNull()
}

Sonuç

- Reactive modelin ve non-blocking IO'nun sağladığı değerlerden faydalandık
- Bundan faydalanırken -kotlin coroutine vasıtasıyla- alışık olduğumuz **sequential declarative** kodlama tarzından vazgeçmedik
- “**Callback hell**” gibi bir duruma girmedik, yönetilemeyen kod bloklarından kaçınmış olduk

Kaynaklar

- Migrating a library from RxJava To Coroutines - Mike Nakhimovich
- Coroutines case study - Cleaning up an async API - Tom Hanley
- Deep dive into coroutines on JVM - KotlinConf 2017 - Roman Elizarov
- Blocking threads, suspending coroutines - Medium - Roman Elizarov



Have a nice Kotlin!

...with coroutines

Q & A



Asenkron ve Paralel Programlama, Kotlin Coroutine on Server-Side

Caner Patır

Software Engineer @trendyoltech



github.com/canerpatir



[@canerpatir](https://www.instagram.com/canerpatir)