Coroutines and Reactive Programming – friends or foes?

Konrad Kamiński
Allegro.pl

```
suspend fun getUser(userId: Int): User?
fun getDefaultUserName(userId: Int): String

val name = getUser(userId)?.name ?: getDefaultUserName(userId)
println(name)
```

```
fun getUser(userId: Int): Mono<User>
fun getDefaultUserName(userId: Int): String

getUser(userId)
  .map { it.name }
  .switchIfEmpty(Mono.fromCallable { getDefaultUserName(userId) })
  .subscribe { name -> println(name) }
```

```
suspend fun getUser(userId: Int): User?
fun getDefaultUserName(userId: Int): String

val name = getUser(userId)?.name ?: getDefaultUserName(userId)
println(name)
```

```
fun getUser(userId: Int): Mono<User>
fun getDefaultUserName(userId: Int): String

getUser(userId)
  .map { it.name }
  .switchIfEmpty(Mono.fromCallable { getDefaultUserName(userId) })
  .subscribe { name -> println(name) }
```

```
suspend fun getUser(userId: Int): User?
fun getDefaultUserName(userId: Int): String

val name = getUser(userId)?.name ?: getDefaultUserName(userId)
println(name)
```

```
fun getUser(userId: Int): Mono<User>
fun getDefaultUserName(userId: Int): String

getUser(userId)
  .map { it.name }
  .switchIfEmpty(Mono.fromCallable { getDefaultUserName(userId) })
  .subscribe { name -> println(name) }
```

Sequential code

```
suspend fun getUser(userId: Int): User
fun getAccount(accountId: Int): Account

suspend fun getAccountNo(userId: Int): String =
  getAccount(getUser(userId).accountId).accountNo
```

```
suspend fun getUser(userId: Int): User
fun getAccount(accountId: Int): Account

suspend fun getAccountNo(userId: Int): String =
  getAccount(getUser(userId).accountId).accountNo
```

```
fun getUser(userId: Int): Mono<User>
fun getAccount(accountId: Int): Account

fun getAccountNo(userId: Int): Mono<String> =
  getUser(userId)
    .map { getAccount(it.accountId).accountNo }
```

```
suspend fun getUser(userId: Int): User
fun getAccount(accountId: Int): Account

suspend fun getAccountNo(userId: Int): String =
  getAccount(getUser(userId).accountId).accountNo
```

```
suspend fun getUser(userId: Int): User
suspend fun getAccount(accountId: Int): Account

suspend fun getAccountNo(userId: Int): String =
  getAccount(getUser(userId).accountId).accountNo
```

```
suspend fun getUser(userId: Int): User
suspend fun getAccount(accountId: Int): Account

suspend fun getAccountNo(userId: Int): String =
  getAccount(getUser(userId).accountId).accountNo
```

```
fun getUser(userId: Int): Mono<User>
fun getAccount(accountId: Int): Account

fun getAccountNo(userId: Int): Mono<String> =
  getUser(userId)
    .map { getAccount(it.accountId).accountNo }
```

```
suspend fun getUser(userId: Int): User
suspend fun getAccount(accountId: Int): Account

suspend fun getAccountNo(userId: Int): String =
  getAccount(getUser(userId).accountId).accountNo
```

```
fun getUser(userId: Int): Mono<User>
fun getAccount(accountId: Int): Mono<Account>

fun getAccountNo(userId: Int): Mono<String> =
  getUser(userId)
    .map { getAccount(it.accountId).accountNo }
```

```
suspend fun getUser(userId: Int): User
suspend fun getAccount(accountId: Int): Account

suspend fun getAccountNo(userId: Int): String =
  getAccount(getUser(userId).accountId).accountNo
```

```
fun getUser(userId: Int): Mono<User>
fun getAccount(accountId: Int): Mono<Account>

fun getAccountNo(userId: Int): Mono<String> =
  getUser(userId)
    .map { getAccount(it.accountId).map(Account::accountNo) }
```

```
suspend fun getUser(userId: Int): User
suspend fun getAccount(accountId: Int): Account

suspend fun getAccountNo(userId: Int): String =
  getAccount(getUser(userId).accountId).accountNo
```

```
fun getUser(userId: Int): Mono<User>
fun getAccount(accountId: Int): Mono<Account>

fun getAccountNo(userId: Int): Mono<String> =
  getUser(userId)
    .flatMap { getAccount(it.accountId).map(Account::accountNo) }
```

Threads

```
suspend fun getUser(userId: Int): User
val userServiceContext = newFixedThreadPoolContext(5, "user")

suspend fun getUserName(userId: Int): String =
  withContext(userServiceContext) {
    getUser(userId).name
  }
```

```
suspend fun getUserName(userId: Int): String
suspend fun calculateEncryptionKey(name: String): String
val encryptionContext = newFixedThreadPoolContext(5, "encryption")

suspend fun getUserEncryptionKey(userId: Int): String =
  withContext(encryptionContext) {
    calculateEncryptionKey(getUserName(userId))
  }
```

```
suspend fun getUserName(userId: Int): String
suspend fun calculateEncryptionKey(name: String): String
val encryptionContext = newFixedThreadPoolContext(5, "encryption")

suspend fun getUserEncryptionKey(userId: Int): String =
  withContext(encryptionContext) {
    calculateEncryptionKey(getUserName(userId))
  }
```

```
fun getUser(userId: Int): Mono<User>
val userScheduler = Schedulers.newParallel("user", 5)

fun getUserName(userId: Int): Mono<String> =
  getUser(userId)
    .map { user -> user.name }
    .subscribeOn(userScheduler)
```

```
suspend fun getUserName(userId: Int): String
suspend fun calculateEncryptionKey(name: String): String
val encryptionContext = newFixedThreadPoolContext(5, "encryption")

suspend fun getUserEncryptionKey(userId: Int): String =
  withContext(encryptionContext) {
    calculateEncryptionKey(getUserName(userId))
  }
```

```
fun getUserName(userId: Int): Mono<String>
fun calculateEncryptionKey(name: String): String
val encryptionScheduler = Schedulers.newParallel("encryption", 5)

fun getUserEncryptionKey(name: String): Mono<String> =
  getUserName(userId)
    .publishOn(encryptionScheduler)
    .map { user -> calculateEncryptionKey(user) }
```

Request context

```
suspend fun getUser(userId: Int): User
val requestId: ThreadLocal<String>
suspend fun loggingGetUser (userId: Int): User =
  log("${requestId.get ()}: $userId").let { getUser(userId) }

suspend fun getUserName(userId: Int): String =
```

```
suspend fun getUser(userId: Int): User
val requestId: ThreadLocal<String>
suspend fun loggingGetUser (userId: Int): User =
  log("${requestId.get ()}: $userId").let { getUser(userId) }

suspend fun getUserName(userId: Int): String =

    loggingGetUser(userId).name
```

```
suspend fun getUser(userId: Int): User
val requestId: ThreadLocal<String>
suspend fun loggingGetUser (userId: Int): User =
  log("${requestId.get ()}: $userId").let { getUser(userId) }

suspend fun getUserName(userId: Int): String =
  withContext(requestId.asContextElement()) {
    loggingGetUser(userId).name
  }
```

```
suspend fun getUser(userId: Int): User
val requestId: ThreadLocal<String>
suspend fun loggingGetUser (userId: Int): User =
  log("${requestId.get ()}: $userId").let { getUser(userId) }

suspend fun getUserName(userId: Int): String =
  withContext(requestId.asContextElement()) {
    loggingGetUser(userId).name
  }
```

```
fun getUser(userId: Int): Mono<User>
val requestId: ThreadLocal<String>
fun loggingGetUser(userId: Int): Mono<User> = getUser(userId).map { user ->

    log("${requestId.get()}: $userId").let { user }
  }
```

```
suspend fun getUser(userId: Int): User
val requestId: ThreadLocal<String>
suspend fun loggingGetUser (userId: Int): User =
  log("${requestId.get ()}: $userId").let { getUser(userId) }

suspend fun getUserName(userId: Int): String =
  withContext(requestId.asContextElement()) {
    loggingGetUser(userId).name
  }
```

```
fun getUser(userId: Int): Mono<User>
val requestId: ThreadLocal<String>
fun loggingGetUser(userId: Int): Mono<User> = getUser(userId).flatMap { user ->
    Mono.subscriberContext().map { context ->
      log("${context.get<String>("reqId")}: $userId").let { user }
  }}
fun getUserName(userId: Int): Mono<String> =
```

```
suspend fun getUser(userId: Int): User
val requestId: ThreadLocal<String>
suspend fun loggingGetUser (userId: Int): User =
  log("${requestId.get ()}: $userId").let { getUser(userId) }

suspend fun getUserName(userId: Int): String =
  withContext(requestId.asContextElement()) {
    loggingGetUser(userId).name
  }
```

```
fun getUser(userId: Int): Mono<User>
val requestId: ThreadLocal<String>
fun loggingGetUser(userId: Int): Mono<User> = getUser(userId).flatMap { user ->
    Mono.subscriberContext().map { context ->
      log("${context.get<String>("reqId")}: $userId").let { user }
  }}
fun getUserName(userId: Int): Mono<String> =
  loggingGetUser(userId).map { it.name }
```

```
suspend fun getUser(userId: Int): User
val requestId: ThreadLocal<String>
suspend fun loggingGetUser (userId: Int): User =
  log("${requestId.get ()}: $userId").let { getUser(userId) }

suspend fun getUserName(userId: Int): String =
  withContext(requestId.asContextElement()) {
    loggingGetUser(userId).name
  }
```

```
fun getUser(userId: Int): Mono<User>
val requestId: ThreadLocal<String>
fun loggingGetUser(userId: Int): Mono<User> = getUser(userId).flatMap { user ->
    Mono.subscriberContext().map { context ->
      log("${context.get<String>("reqId")}: $userId").let { user }
  }}
fun getUserName(userId: Int): Mono<String> =
  loggingGetUser(userId).map { it.name }
    .subscriberContext(Context.of("reqId", requestId.get()))
```

Exception handling

```
suspend fun getUser(userId: Int): User

suspend fun getUserName(userId: Int): String =
  try {
    getUser(userId).name
  }  catch (e: UserNotFoundException) {
    "Unknown: $userId"
  }
```

```kotlin
suspend fun getUser(userId: Int): User

suspend fun getUserName(userId: Int): String =
  try {
    getUser(userId).name
  }  catch (e: UserNotFoundException) {
    "Unknown: $userId"
  }
```

```kotlin
fun getUser(userId: Int): Mono<User>

fun getUserName(userId: Int): Mono<String> =
  getUser(userId)
    .map { it.name }
    .onErrorReturn("Unknown: $userId")
    // onErrorResume
    // onErrorMap
```

Retrying

```
suspend fun getUser(userId: Int): User
suspend fun retryingGetUser(userId: Int): User {
  lateinit var ex: Exception
  repeat(5) { counter ->
    try { return getUser(userId) }
    catch (e: Exception) { delay((counter+1L)*500); ex = e }
  }
  throw ex
}
```

```
suspend fun <T> retry(n: Int, delayMillis: Int, fn: suspend () -> T): T {
  lateinit var ex: Exception
  repeat(n) { counter ->
    try { return fn() }
    catch (e: Exception) { delay((counter+1L)*delayMillis); ex = e }
  }
  throw ex
}
```

```
suspend fun getUser(userId: Int): User
suspend fun <T> retry(n: Int, delayMillis: Int, fn: suspend () -> T): T

suspend fun retryingGetUser(userId: Int): User =
  retry(5, 500) {
    getUser(userId)
  }
```

```
suspend fun getUser(userId: Int): User
suspend fun <T> retry(n: Int, delayMillis: Int, fn: suspend () -> T): T

suspend fun retryingGetUser(userId: Int): User =
  retry(5, 500) {
    getUser(userId)
  }
```

```
fun getUser(userId: Int): Mono<User>
fun retryFunc(n: Int, millis: Int, ex: Flux<Throwable>): Flux<Long>

fun retryingGetUser(userId: Int): Mono<User> =
  getUser(userId)
    .retryWhen { exceptions ->
      retryFunc(5, 500, exceptions)
    }
```

```
suspend fun getUser(userId: Int): User
suspend fun <T> retry(n: Int, delayMillis: Int, fn: suspend () -> T): T

suspend fun retryingGetUser(userId: Int): User =
  retry(5, 500) {
    getUser(userId)
  }
```

```
fun retryFunc(n: Int, millis: Int, ex: Flux<Throwable>): Flux<Long> =
  ex.zipWith(Flux.range(1, n), BiFunction { error: Throwable, index: Int ->
       if (index < n) index.toLong()
       else throw Exceptions.propagate(error)
     })
     .flatMap { index ->
       Mono.delay(Duration.ofMillis(index * millis))
     }
```

Concurrent code

```
suspend fun getUser(userId: Int): User
suspend fun getRoles(userId: Int): Roles

suspend fun getUserWithRoles(userId: Int): Pair<User, Roles> = coroutineScope {
  val user: Deferred<User> = async { getUser(userId) }
  val roles: Deferred<Roles> = async { getRoles(userId) }

  user.await() to roles.await()
}
```

```
suspend fun getUser(userId: Int): User
suspend fun getRoles(userId: Int): Roles

suspend fun getUserWithRoles(userId: Int): Pair<User, Roles> = coroutineScope {
  val user: Deferred<User> = async { getUser(userId) }
  val roles: Deferred<Roles> = async { getRoles(userId) }

  user.await() to roles.await()
}
```

```
fun getUser(userId: Int): Mono<User>
fun getRoles(userId: Int): Mono<Roles>

fun getUserWithRoles(userId: Int): Mono<Pair<User, Roles> > =
  getUser(userId)
    .zipWith(getRoles(userId)) { user, roles ->
      user to roles
    }
```

Cancellation

```
suspend fun getUserWithRoles(userId: Int): Pair<User, Roles> = coroutineScope {
  val userDef = async { getUser(userId) }
  val rolesDef = async { getRoles(userId) }

  val user = userDef.await()
  when (user.isBlocked()) {
    true -> { rolesDef.cancel(); user to Roles.EMPTY }
    else -> user to rolesDef.await()
}}
```

```
suspend fun getUserWithRoles(userId: Int): Pair<User, Roles> = coroutineScope {
  val userDef = async { getUser(userId) }
  val rolesDef = async { getRoles(userId) }

  val user = userDef.await()
  when (user.isBlocked()) {
    true -> { rolesDef.cancel(); user to Roles.EMPTY }
    else -> user to rolesDef.await()
}}
```

```
fun getUserWithRoles(userId: Int): Mono<Pair<User, Roles>> {
  val user = getUser(userId).flux().share().single()
  return Mono.first(
    user.flatMap { u ->
      if (u.isBlocked()) Mono.just(u to Roles.EMPTY) else Mono.never() },
    user.zipWith<Roles, Pair<User, Roles>>(getRoles(userId)) { u,r ->
      u to (if (u.isBlocked()) Roles.EMPTY else r) }
  )
}
```

Parallel code

```
suspend fun getUserIds(accountId: Int): List<Int>
suspend fun getUser(userId: Int): User

suspend fun getUsers(accountId: Int): List<User> = coroutineScope {
  getUserIds(accountId) // List<Int>


}
```

```
suspend fun getUserIds(accountId: Int): List<Int>
suspend fun getUser(userId: Int): User

suspend fun getUsers(accountId: Int): List<User> = coroutineScope {
  getUserIds(accountId) // List<Int>
    .map { userId -> async { getUser(userId) } } // List<Deferred<User>>

}
```

```
suspend fun getUserIds(accountId: Int): List<Int>
suspend fun getUser(userId: Int): User

suspend fun getUsers(accountId: Int): List<User> = coroutineScope {
  getUserIds(accountId) // List<Int>
    .map { userId -> async { getUser(userId) } } // List<Deferred<User>>
    .awaitAll()
}
```

```
suspend fun getUserIds(accountId: Int): List<Int>
suspend fun getUser(userId: Int): User

suspend fun getUsers(accountId: Int): List<User> = coroutineScope {
  getUserIds(accountId) // List<Int>
    .map { userId -> async { getUser(userId) } } // List<Deferred<User>>
    .awaitAll()
}
```

```
fun getUserIds(accountId: Int): Mono<List<Int>>
fun getUser(userId: Int): Mono<User>

fun getUsers(accountId: Int): Mono<List<User>> =
  getUserIds(accountId).flatMapIterable { it }
    .flatMap { getUser(it) }.collectList()
```

```
suspend fun getUserIds(accountId: Int): List<Int>
suspend fun getUser(userId: Int): User

suspend fun getUsers(accountId: Int): List<User> = coroutineScope {
  getUserIds(accountId) // List<Int>
    .map { userId -> async { getUser(userId) } } // List<Deferred<User>>
    .awaitAll()
}
```

```
fun getUserIds(accountId: Int): Flux<Int>
fun getUser(userId: Int): Mono<User>

fun getUsers(accountId: Int): Flux<User> =
  getUserIds(accountId)
    .flatMap { getUser(it) }
```

Streams

```
suspend fun getUsers(accountId: Int): ReceiveChannel<User>

suspend fun printUsers(accountId: Int) {
  val users = getUsers(accountId)
  while (true) {
    val user = channel.receive()
    println(user.name)
  }
}
```

```
suspend fun getUsers(accountId: Int): ReceiveChannel<User>

suspend fun printUsers(accountId: Int) {
  val users = getUsers(accountId)
  while (true) {
    val user = channel.receiveOrNull()
    if (user != null) println(user.name) else break
  }
}
```

```
suspend fun getUsers(accountId: Int): ReceiveChannel<User>

suspend fun printUsers(accountId: Int) {
  val users = getUsers(accountId)

  users.consumeEach { user ->
    println(user.name)
  }
}
```

```
suspend fun getUsers(accountId: Int): ReceiveChannel<User>

suspend fun printUsers(accountId: Int) {
  val users = getUsers(accountId)

  users.consumeEach { user ->
    println(user.name)
  }
}
```

```
fun getUsers(accountId: Int): Flux<User>

fun printUsers(accountId: Int) {
  getUsers(accountId)
    .subscribe { user ->
      println(user.name)
    }
}
```

Stream generation

```
suspend fun CoroutineScope.getUsers(accountId: Int): ReceiveChannel<User> =
  Channel<User>().apply {
    launch {
      send(user1)
      send(user2)
      close()
    }
  }
```

```
suspend fun CoroutineScope.getUsers(accountId: Int): ReceiveChannel<User> = produce {
  send(user1) // same as channel.send(user1)
  send(user2)
  close()
}
```

```
suspend fun CoroutineScope.getUsers(accountId: Int): ReceiveChannel<User> = produce {
  send(user1) // same as channel.send(user1)
  send(user2)
  close()
}
```

```
fun getUsers(accountId: Int): Flux<User> = Flux.generate { sink ->
  sink.next(user1)
  sink.next(user2)
  sink.complete()
}
```

Operators

```
suspend fun getUsers(accountId: Int): ReceiveChannel<User>

suspend fun getUserNames(accountId: Int): ReceiveChannel<String> =
  getUsers(accountId)
    .filterNot { it.blocked }
    .map { it.name }
```

```
suspend fun getUsers(accountId: Int): ReceiveChannel<User>

suspend fun getUserNames(accountId: Int): ReceiveChannel<String> =
  getUsers(accountId)
    .filterNot { it.blocked }
    .map { it.name }
```

```
fun getUsers(accountId: Int): Flux<User>

fun getUserNames(accountId: Int): Flux<String> =
  getUsers(accountId)
    .filterNot { it.blocked }
    .map { it.name }
```

Custom operator

```
fun <T, R> ReceiveChannel<T>.filterMap (fn: (T) -> R?) = GlobalScope.produce<R> {
  consumeEach { t ->
    val r = fn.invoke(t)
    if (r != null) send(r)
  }
}
```

---

300 lines of code for filter

260 lines of code for map

Backpressure

```
suspend fun CoroutineScope.getUsers(accountId: Int): ReceiveChannel<User> =
  Channel<User>().apply {
    launch {
      send(user1)
      send(user2)
      close()
    }
  }
```

```
val rendezvousChannel = Channel<User>()
val rendezvousChannel = Channel<User>(RENDEZVOUS)
val bufferedChannel = Channel<User>(5)
val unlimitedChannel = Channel<User>(UNLIMITED)
val conflatedChannel = Channel<User>(CONFLATED)
```

```
val rendezvousChannel = Channel<User>()
val rendezvousChannel = Channel<User>(RENDEZVOUS)
val bufferedChannel = Channel<User>(5)
val unlimitedChannel = Channel<User>(UNLIMITED)
val conflatedChannel = Channel<User>(CONFLATED)
```

```
val flux: Flux<User>()

val bufferedFlux = flux.onBackpressureBuffer(bufferSize)
val droppingFlux = flux.onBackpressureDrop()
val latestFlux = flux.onBackpressureLatest()
```

Interoparability

```
fun reactiveGetUser(userId: Int): Mono<User>

suspend fun getUser(userId: Int): User? =
  reactiveGetUser(userId).awaitFirstOrNull()
                         // awaitFirst()
                         // awaitFirstOrDefault(defaultValue: T)
                         // awaitFirstOrElse(defaultValue: () -> T)
```

```
fun reactiveGetUser(userId: Int): Mono<User>

suspend fun getUser(userId: Int): User? =
  reactiveGetUser(userId).awaitFirstOrNull()
                        // awaitFirst()
                        // awaitFirstOrDefault(defaultValue: T)
                        // awaitFirstOrElse(defaultValue: () -> T)
```

```
suspend fun suspendingGetUser(userId: Int): User?

fun CoroutineScope.getUser(userId: Int): Mono<User> = mono {
  suspendingGetUser(userId)
}
```

```
fun getUsersFlux(accountId: Int): Flux<User>

fun getUsers(accountId: Int): ReceiveChannel<User> =
  reactiveGetUsers(accountId).openSubscription()
```

```
fun getUsersFlux(accountId: Int): Flux<User>

fun getUsers(accountId: Int): ReceiveChannel<User> =
  reactiveGetUsers(accountId).openSubscription()
```

```
fun getUsersChannel(accountId: Int): ReceiveChannel<User>

fun CoroutineScope.getUsers(accountId: Int): Flux<User> = flux {
  getUsersChannel(accountId).consumeEach { user ->
    send(user)
  }
}
```

Coroutines or Reactive Programming?

- Sequential code is where coroutines shine
- Concurrent code complexity depends a lot on your use case
- Channels are hot and experimental
- You can mix both solutions

Where to find more information

- Guide to reactive streams with coroutines
  `https://bit.ly/2xNLF1m`
- Corotoutines guide
  `https://bit.ly/2NTUoZS`

Thank you!

Konrad Kamiński
Allegro.pl