Server-Sent Events (SSE)

Fra `while (true)` til Kotlin Flows

En moderne tilnærming til real-time kommunikasjon

Agenda

- **Hva er SSE?**: WebSockets, nesten. Eller?
- **Problemet**: While-løkker og kompleksitet
- **Løsningen**: SSE med Kotlin Flows
- **Hva er Flows?**: Asynkrone lister og patterns
- **I praksis**: Versjonspolling og KI-søk
- **Arkitektur**: RØDDIG, testbar kode
- **Resultatet**: Gevinster og veien videre

Hva er Server-Sent Events?

SSE er en web-standard for å sende data fra server til klient i real-time.

- **En-veis kommunikasjon**: Server → Klient
- **Automatisk gjenkobling**: Hvis tilkoblingen brytes
- **Innebygd i moderne nettlesere**: EventSource API
- **Lettere enn WebSockets**: For mange bruksområder

```
// Frontend (TypeScript)
const eventSource = new EventSource("/sse");
eventSource.addEventListener("AISearchReady", (event) => {
  console.log("AI søk ferdig:", event.data);
});
```

Vår faktiske implementasjon i frontend er litt mer sofistikert, men det over er egentlig alt som strengt tatt trengs.

SSE med Ktor: Enkelt og greit

Her er en minimal implementasjon av SSE-endepunkt i Ktor:

```
// Enkel SSE route i Ktor
sse("/sse") {
    // Send en enkel "Hello, World!" event
    send(ServerSentEvent(
        event = "Hello, World!",
    ))
}
```

Viktige detaljer:

- `sse()`-funksjonen setter automatisk riktig content-type og headers
- `ServerSentEvent` er Ktors innebygde klasse for SSE-meldinger
- `event`-parameter setter event-typen
- `data`-parameter inneholder meldingsdata (valgfritt)
- Ktor håndterer automatisk riktig formatering og flushing

Før: While-løkker og kompleksitet

Den gamle tilnærmingen hadde flere utfordringer:

```
// Gammel tilnærming (forenklet)
var updPrincipal = getUpdatedPrincipal()
var clientOpen = true
while (updPrincipal != null && clientOpen) {
    val meta = createMeta(true)
    clientOpen = sendMetaEvent(this, meta)
    if (clientOpen) {
        delay(30000)
        updPrincipal = getUpdatedPrincipal()
    }
}

if (clientOpen) {
    sendMetaEvent(this, createMeta(true))
}
```

Problemer:

- Imperativ programmering - Vanskelig å følge logikken
- Manuell tilstandshåndtering - Lett å glemme opprydding
- Kompleks feilhåndtering - Spredd utover mange steder
- Vanskelig å teste - Tett koblet logikk

Nå: Kotlin Flows og komposisjon

Moderne tilnærming med deklarativ programmering:

```
// Heartbeat flow
val heartBeatEvents = flow {
    while (currentCoroutineContext().isActive) {
        delay(HEARTBEAT_INTERVAL_MS)
        emit(ProtoServerSentEvent.Heartbeat)
    }
}

// Frontend version events
val frontendVersionEvents = frontendVersionPoller.latestAvailableVersion
    .filterNotNull()
    .map { version ->
        createMetaEventWithFrontendVersion(latestFrontendVersion = version)
    }.distinctUntilChanged()

// Session-specific events
val (sessionEvents, cleanUpSessionEvents) =
    serverEventEmitter.eventsForSession(sessionId)

// Kombiner alle flows
handleSseEventsUntilLogoutOrDisconnect(
    sessionId, uniqueId, sseSession,
    frontendVersionEvents, heartBeatEvents, sessionEvents
)
```

Event-typer

Type-safe events med sealed interface:

```
sealed interface ProtoServerSentEvent {
    val event: String?
    fun serialize(): String?
    fun plainEvent() = ServerSentEvent(event = event, data = serialize())
    val shouldTriggerAuthCheck: Boolean
}

// Heartbeat for å holde tilkobling levende
data object Heartbeat : ProtoServerSentEvent {
    override val event: String = "Heartbeat"
    override fun serialize() = null
    override val shouldTriggerAuthCheck = true
}

// KI-søk ferdig
data class AISearchReady(val searchId: String) : ProtoServerSentEvent {
    override val event = "AISearchReady"
    override fun serialize() = searchId
    override val shouldTriggerAuthCheck = false
}

// Metadata om versjonsnumre
@Serializable
data class Meta(
    @SerialName("revision") val latestBackendVersion: String,
    @SerialName("latestFrontendVersion") val latestFrontendVersion: String
) : ProtoServerSentEvent {
    override val event: String = "Meta"
    override fun serialize() = Json.encodeToString(serializer(), this)
    override val shouldTriggerAuthCheck = false
}
```

Hva er Kotlin Flows?

En asynkron liste som emitter verdier over tid

Tenk på forskjellen mellom:

```
// Liste - alle verdier tilgjengelig med en gang
val numbers: List<Int> = listOf(1, 2, 3, 4, 5)

// Sequence - lazy evaluering, men fortsatt synkron
val sequence: Sequence<Int> = sequenceOf(1, 2, 3, 4, 5)
    .map { it * 2 }

// Flow - asynkron "liste" som emitter verdier over tid
val flow: Flow<Int> = flow {
    emit(1)
    delay(1000)
    emit(2)
    delay(1000)
    emit(3)
}
```

Flow = Asynkron + Cold + Composable

Flow patterns du bør kjenne

🔄 Produsere verdier

```
// Timer flow
flow {
    var counter = 0
    while (currentCoroutineContext().isActive) {
        emit(counter++)
        delay(1000)
    }
}

// StateFlow - hot flow med state
val _state = MutableStateFlow(initialValue)
val state: StateFlow<T> = _state.asStateFlow()
```

🔀 Transformere

```
flow.map { it * 2 }
    .filter { it > 5 }
    .distinctUntilChanged()
    .take(10)
```

🤝 Kombinere

```
// Merge - alle verdier fra alle flows
merge(flow1, flow2, flow3)

// Combine - kombinere latest fra hver flow
combine(flow1, flow2) { a, b -> a + b }
```

Bruksområde 1: Frontend versjon poller

Automatisk oppdatering ved ny deploy

```
@Singleton
class FrontendVersionPoller @Inject constructor(
    application: Application,
    private val client: HttpClient
) : AutoCloseable {
    private val refreshInterval: Duration = 10.seconds
    private val scope = CoroutineScope(SupervisorJob() + Dispatchers.IO)

    private val domain = when {
        application.developmentMode -> "http://localhost:1234"
        ServerUtil.isProductionServer() -> "https://pro2.lovdata.no"
        else -> "https://stage-pro2.lovdata.no"
    }

    private val _latestAvailableVersion = MutableStateFlow<String?>(null)
    val latestAvailableVersion: StateFlow<String?> = _latestAvailableVersion.asStateFlow()

    private val job = scope.launch {
        while (isActive) {
            runCatching {
                client.get("$domain/.version?${UUID.randomUUID()}")
                    .bodyAsText().trim()
            }.onSuccess {
                _latestAvailableVersion.value = it
            }.onFailure {
                log.warn("Failed to fetch version")
            }
            delay(duration = refreshInterval)
        }
    }
}
```

Versjon polling (fortsettelse)

Konvertering til SSE events

```
// I StatusRoutes - konverter StateFlow til event flow
val frontendVersionEvents = frontendVersionPoller.latestAvailableVersion
    .filterNotNull()
    .map { version ->
        createMetaEventWithFrontendVersion(latestFrontendVersion = version)
    }.distinctUntilChanged()

private fun createMetaEventWithFrontendVersion(latestFrontendVersion: String) =
    ProtoServerSentEvent.Meta(
        latestBackendVersion = backendVersion,
        buildDate = buildDate,
        latestFrontendVersion = latestFrontendVersion
    )
```

Resultat: Alle tilkoblede klienter får automatisk beskjed når det er en ny frontend-versjon tilgjengelig.

Bruksområde 2: KI-søk

Asynkron behandling med callback

KI-søk kan ta lang tid, så vi returnerer umiddelbart med en ID:

```
post("/ai-search") {
    either<RouteError, AISearchResponse> {
        val sessionId = ensureNotNull(call.sessionId) { Unauthorized }
        val userID = call.userPrincipal().bind().userID
        val query = call.bodyObject<AISearchRequest>().bind().query

        // Sjekk om søket allerede finnes
        val (term, exists) = aiCacheRepository.findOrCreateTerm(userID, query).bind()
        if (exists) {
            return@either AISearchResponse(id = term.id)
        }

        // Start asynkron søk med callback
        triggerAndCacheAsyncAiSearch(userID, term) { searchId ->
            // Emit event når søket er ferdig
            serverEventEmitter.sendSessionEvent(
                sessionId = sessionId,
                protoEvent = ProtoServerSentEvent.AISearchReady(searchId)
            )
        }

        AISearchResponse(id = term.id)
    }.also {
        call.respondJson(it, status = HttpStatusCode.Accepted)
    }
}
```

Resultat: Klienten får umiddelbart svar og blir varslet via SSE når søket er ferdig.

Arkitektur: Globale Events

Events som sendes til alle tilkoblede klienter:

```
@Singleton
class ServerEventEmitter {
    private val globalEvents = MutableSharedFlow<ProtoServerSentEvent>(
        extraBufferCapacity = 16,
        onBufferOverflow = BufferOverflow.DROP_OLDEST
    )

    suspend fun sendGlobalEvent(protoEvent: ProtoServerSentEvent) {
        globalEvents.emit(protoEvent)
    }
    /*...*/
}
```

Globale events brukes for:

- Informasjon om nedetid
- ?

Fordeler:

- Effektiv broadcasting til alle klienter
- Ingen session-spesifikk state
- Buffer overflow håndtering beskytter mot slow consumers

Arkitektur: Session Events

Events som sendes til spesifikke brukere/sesjoner:

```
@Singleton
class ServerEventEmitter {
    /*...*/
    private val sessionEventsMap = hashMapOf<String, MutableSharedFlow<ProtoServerSentEvent>>()

    fun eventsForSession(sessionId: String?): Pair<Flow<ProtoServerSentEvent>, () -> Unit> {
      if (sessionId == null) {
        return Pair(emptyFlow()) {}
      }
      val sessionEvents =
        sessionEventsMap.getOrPut(sessionId) {
          MutableSharedFlow<ProtoServerSentEvent>(
            extraBufferCapacity = 16,
            onBufferOverflow = BufferOverflow.DROP_OLDEST
          )
        }

      val closeFunction = {
        sessionEventsMap.remove(sessionId)
        Unit
      }

      return Pair(
        merge(globalEvents, sessionEvents)
          .takeWhile { sessionEventsMap.containsKey(sessionId) },
        closeFunction
      )
    }

    suspend fun sendSessionEvent(sessionId: String, protoEvent: ProtoServerSentEvent) {
        sessionEventsMap[sessionId]?.emit(protoEvent)
            ?: log.warn("No session with id $sessionId")
    }
}
```


Session events brukes for:

Flow-kombinering og livssyklus

Robust håndtering av tilkoblinger

```
private suspend fun handleSseEventsUntilLogoutOrDisconnect(
    sessionId: String?, uniqueId: String, sseSession: ServerSSESession,
    vararg flows: Flow<ProtoServerSentEvent>
) {
    createEventFlowThatTerminatesOnLogoutOrDisconnect(
        authChecker = { event ->
            if (event.shouldTriggerAuthCheck) {
                checkClientAuth(sessionId)
            } else {
                true // Skip auth check for deploy events
            }
        },
        eventSender = { sseSession.sendProtoEvent(it) },
        loggedOutEventGenerator = ::createLoggedOutEvent,
        flows = flows
    ).collect { (clientOpen, clientAuthenticated) ->
        log.debug("Session $sessionId::$uniqueId (open: $clientOpen, auth: $clientAuthenticated)")
    }
}
```

Dependency Inversion i praksis

Business logic i companion objects - testbar uten DI-frameworks

```
// StatusRoutes.kt - Business critical logic flyttet til companion object
companion object {
    internal suspend fun <T> createEventFlowThatTerminatesOnLogoutOrDisconnect(
        authChecker: suspend (T) -> Boolean,
        eventSender: suspend (T) -> Boolean,
        loggedOutEventGenerator: (() -> T),
        vararg flows: Flow<T>
    ): Flow<Pair<Boolean, Boolean>> =
        merge(*flows)
            .mapIndexed { index, event ->
                if (index == 0 || authChecker(event)) {
                    Pair(event, true)
                } else {
                    Pair(loggedOutEventGenerator(), false)
                }
            }.transformWhile { (event, clientAuthenticated) ->
                val clientOpen = eventSender(event)
                emit(clientOpen to clientAuthenticated)
                clientOpen && clientAuthenticated
            }
}
```

Dependency Inversion: Fordeler

🧪 Testbarhet uten kompleksitet

- Ingen mock-frameworks nødvendig
- Kun funksjoner som parametere
- Business logic isolert fra infrastruktur

🔧 Clean Architecture principper

- Dependency Inversion Principle (DIP)
- Business logic avhenger ikke av detaljer
- Detaljer (HTTP, SSE) avhenger av abstraksjoner

Resultat: Robust kode som er lett å teste og vedlikeholde uten tunge DI-frameworks og rar magi.

Testing: Konkrete eksempler

🧪 Faktiske tester fra SSEFlowCollectionTest.kt

```
@Test
fun `event collection stops after client disconnect`() {
    testSuspend {
        var eventSendShouldSucceed = true
        var lastOutput = ""
        var counter = 1

        StatusRoutes.createEventFlowThatTerminatesOnLogoutOrDisconnect(
            authChecker = { true }, // let all auth checks succeed
            eventSender = { event ->
                lastOutput = event
                eventSendShouldSucceed
            },
            loggedOutEventGenerator = { "logged out event" },
            everlastingHeartbeat
        ).collect { (clientConnected, clientAuthenticated) ->
            if (counter == 11) {
                assertEquals(false, clientConnected)
                assertEquals(true, clientAuthenticated)
                assertEquals("heartbeat", lastOutput)
            }
            counter++
            // simulate disconnect after 10 successful heartbeat sends
            if (counter == 10) {
                eventSendShouldSucceed = false
            }
        }
        assertEquals(11, counter)
    }
}
```

Resultat: Business logic er fullstendig testbar uten å mocke infrastruktur!

Best Practices: Resiliens og ytelse

🛡️ Buffer overflow håndtering

```
MutableSharedFlow<ProtoServerSentEvent>(
    extraBufferCapacity = 16,
    onBufferOverflow = BufferOverflow.DROP_OLDEST
)
```

🔐 Selektiv autentisering

```
val shouldTriggerAuthCheck: Boolean // Per event type
```

🧹 Automatisk opprydding

```
val closeFunction = { sessionEventsMap.remove(sessionId) }
```

📊 Overvåkning

```
private val openSessionCounter = AtomicInteger()
// Log antall åpne sesjoner hvert minutt
```

Fordeler med den nye tilnærmingen

Før (Problemer)	Nå (Løsninger)
Imperative `while`-løkker	Deklarative Flows
Manuell tilstandshåndtering	Automatisk resource management
Spredt feilhåndtering	Sentralisert feilhåndtering
Vanskelig å teste	Enkelt å teste flows
Tight coupling	Loose coupling via events
Kompleks livssyklus	Komposabel arkitektur

Nøkkelgevinst: Vi kan enkelt legge til nye event-typer og flows uten å endre eksisterende kode!

Fremtidige muligheter

🔄 Nye event-typer

- `SystemMaintenance` - Planlagt nedetid
- ?

📈 Utvidelser

- Rate limiting per session?
- Event replay for tapte meldinger
- Metrics og detailed logging
- A/B testing av features

🛠️ Tekniske forbedringer

- Compression av store events
- Custom serialization
- Event batching for høy trafikk

Ting som er verdt å se mer på

- Ytelsesoptimalisering
- Snill load vs viktige events
- Monitoring og debugging
- Frontend implementasjonsdetaljer
- Sammenligning med WebSockets

Takk for oppmerksomheten

Spørsmål om SSE og Kotlin Flows?

🔁 **Server-Sent Events + Kotlin Flows =** 💚