

Bear Season: Building a Fake Stock Market in Functional Kotlin

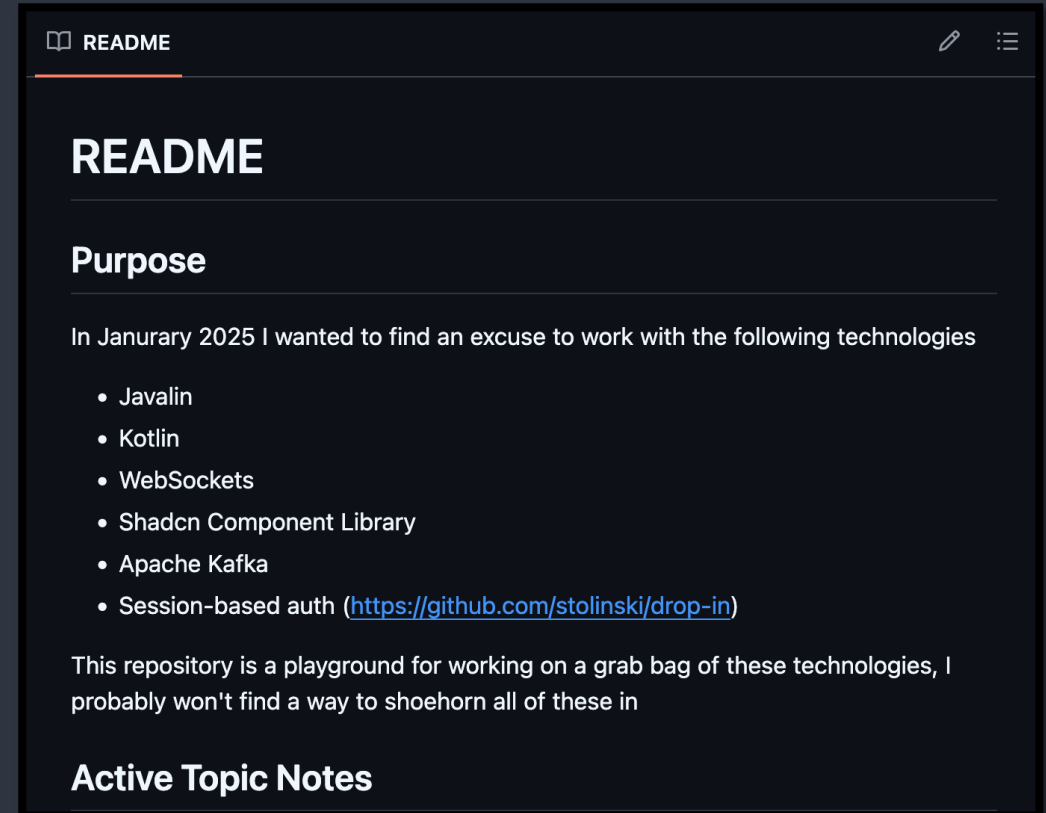
Iain Schmitt

SPS Tech Jam 2025


demo.iainschmitt.com

My exchange worked backwards from some technologies I wanted to work with, not a normal way of picking a project

- Javalin: similar to ASP.NET minimal API
- Kafka: used in Assortment
- Kotlin: F# Kafka transferability concerns, used at SPS, no 'mixed paradigm' issues
- WebSockets: Not another pure REST API side project!
- Session-based auth: wanted to do it once
- Shadcn: less frontend emphasis, nice components otherwise



Finding out about patio11's Stockfighter fake stock exchange piqued my interest and checked a few boxes for an actual project



Kalzumeus

Developing In Stockfighter With No Trading Experience

[Starfighter](#) is a company which makes fun programming challenges. One of our goals is inspiring engineers to take a whack at problems they might assume are “too difficult for me.” Both sets of levels for our first game, Stockfighter, give copious opportunities for this: one set has you do algorithmic trading and one set has you do low-level C and assembly coding, reverse engineering, and security research.



Important primitive: order book shows price and quantity of buy, sell orders for a single asset

- Bid: existing offers to buy
- Ask: existing offers to sell
- Quote: highest bid/smallest ask, 71¢/74¢
- Spread: difference between lowest ask and highest bid
- Limit orders vs. market orders: interaction with order book

Example Order Book for MNAD

		Price	Volume
Asks	Offers to Sell MNAD	76¢	200
		75¢	15
		74¢	10
3¢ spread			
Bids	Offers to Buy MNAD	71¢	3
		70¢	5

What will happen if someone submits a limit order to buy 2 shares of MNAD at 72¢?

		Price	Volume
Asks	Offers to Sell MNAD	76¢	200
		75¢	15
		74¢	10
3¢ spread			
Bids	Offers to Buy MNAD	71¢	3
		70¢	5

		Price	Volume
Asks	Offers to Sell MNAD	76¢	200
		75¢	15
		74¢	10
2¢ spread			
Bids	Offers to Buy MNAD	72¢	2
		71¢	3
		70¢	5

What will happen if someone submits a limit order to buy 4 shares of MNAD at 74¢?

		Price	Volume
Asks	Offers to Sell MNAD	76¢	200
		75¢	15
		74¢	10
2¢ spread			
Bids	Offers to Buy MNAD	72¢	2
		71¢	3
		70¢	5

		Price	Volume
Asks	Offers to Sell MNAD	76¢	200
		75¢	15
		74¢	6
2¢ spread			
Bids	Offers to Buy MNAD	72¢	2
		71¢	3
		70¢	5

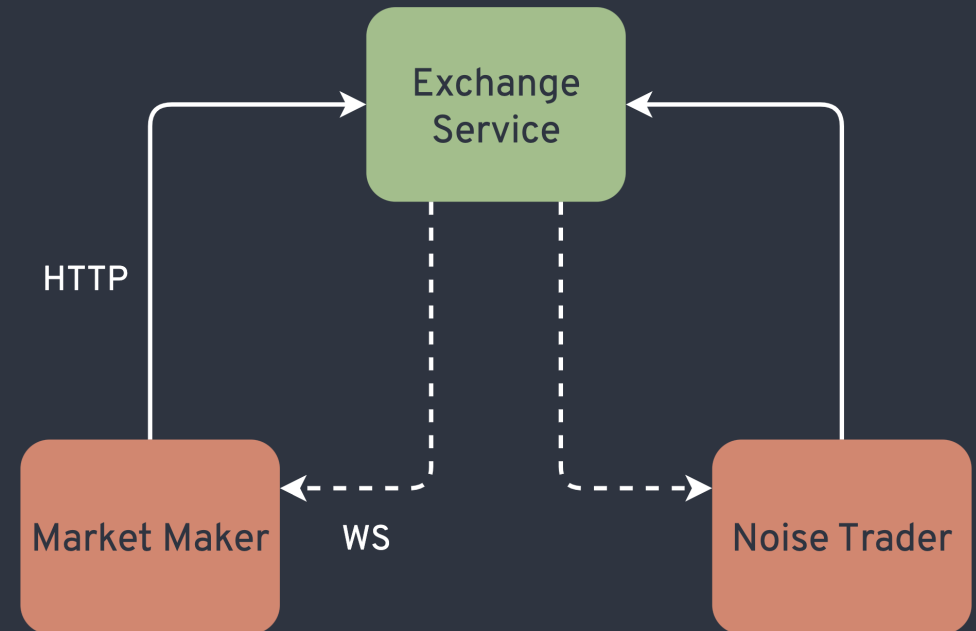
What will happen if someone submits a market order to sell 2 shares of MNAD?

		Price	Volume
Asks	Offers to Sell MNAD	76¢	200
		75¢	15
		74¢	6
2¢ spread			
Bids	Offers to Buy MNAD	72¢	2
		71¢	3
		70¢	5

		Price	Volume
Asks	Offers to Sell MNAD	76¢	200
		75¢	15
		74¢	6
3¢ spread			
Bids	Offers to Buy MNAD	72¢	2
		71¢	3
		70¢	5

Most important “moving parts” of the model are the exchange and the market maker

- Exchange
 - Source of truth, authentication for clients
 - Market, limit orders, order cancelation
 - Pushes quote changes, other messages over WebSockets to clients
- Noise Traders
 - Client service
 - Submit random buy, sell market orders
- Market Maker
 - Client service like the noise traders
 - Places limit orders for noise traders to trade against
 - Reacts to quote changes



Market makers have no perspective on the value of an asset, and my market maker has very simple logic

Order Book

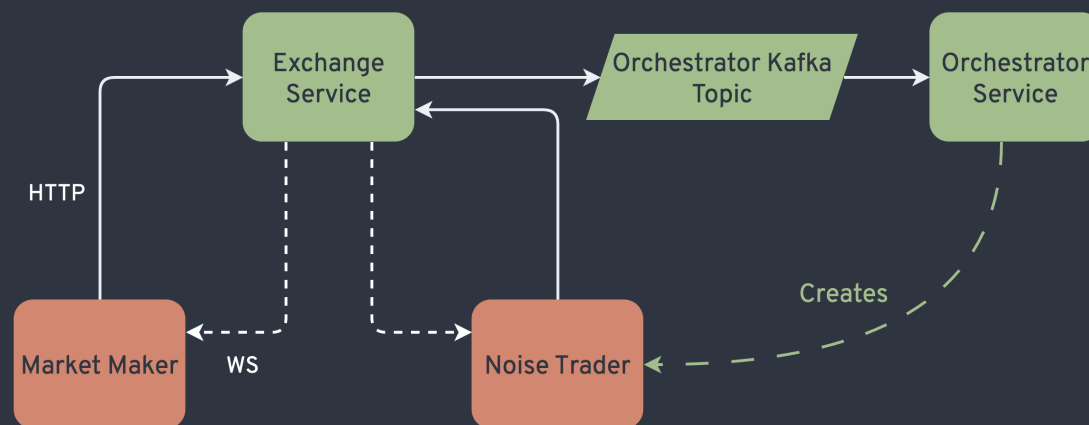
	Price	Volume
Asks	76¢	200
	75¢	15
	74¢	10
3¢ spread		
Bids	71¢	3
	70¢	5

Market Making Logic

1. Market maker selects initial quote, submits buy & sell limit orders
2. Wait for WebSocket message with new quote
3. When new quote received, cancel all of its orders
 1. If quote shifts higher, submit higher buy & sell limit orders: 71/74 to 72/75
 2. If quote shifts lower, submit lower buy & sell limit orders: 71/74 to 70/73
 3. Can widen spreads as necessary
4. This is pretty easy to game

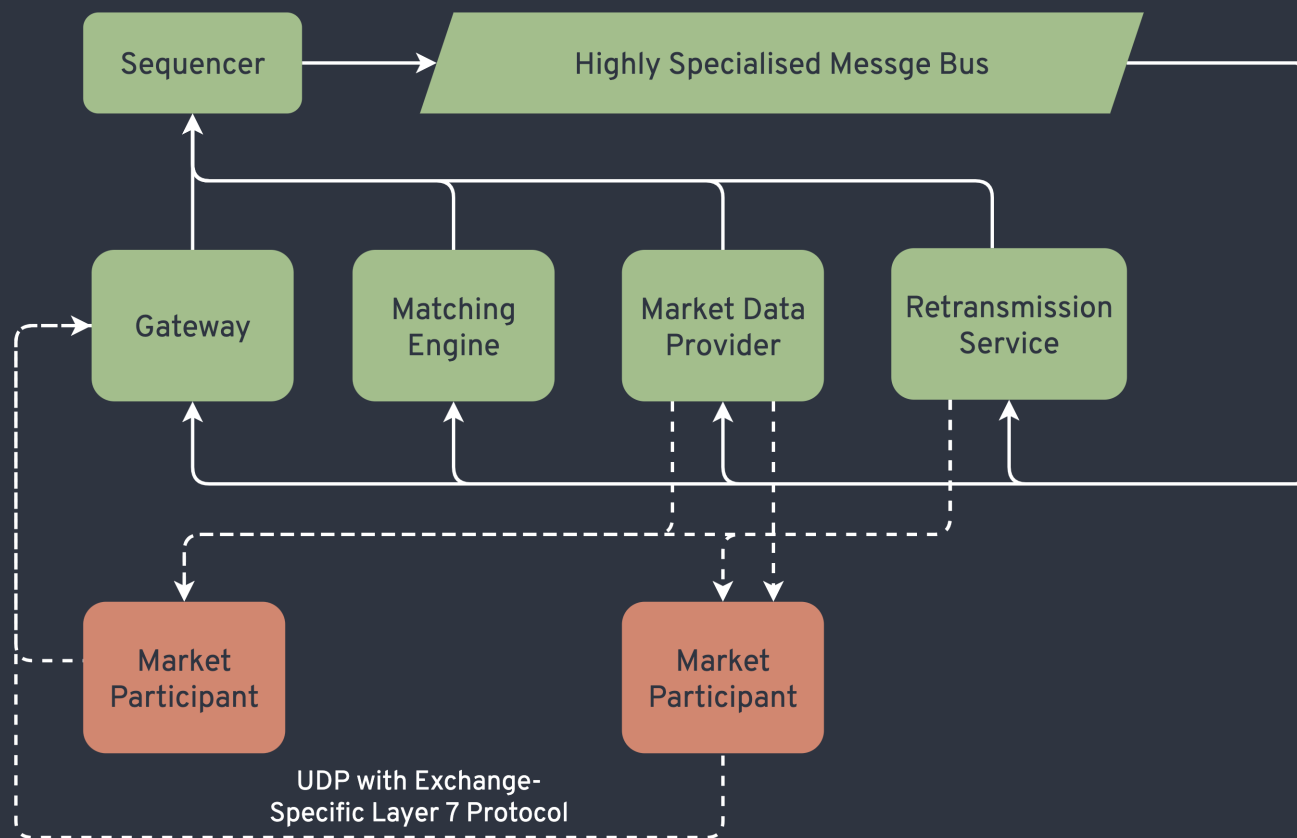
Because noise traders gradually lose credits, new ones need to be seeded from market maker funds

		Price	Volume
Asks		76¢	200
		75¢	15
		74¢	10
3¢ spread			
Bids		71¢	3
		70¢	5



Difficult to exaggerate how much more sophisticated most real exchanges are by comparison

- Millions of live orders, millions of messages per second
- Sequencer: single point of failure, ~100 lines of C
- Can't multithread as much as you'd think
- "How to Build an Exchange" talks from Brian Nigito, Rachel Wonnacott
- Protocols: OUCH, SOUP, SIP, ITCH, etc.



Takeaway: Kotlin has a small learning curve, fantastic language to work with

- Null safety
- Coroutines
- Concise classes
- Extension properties & functions
- Unit type vs. void
- Single-expression functions
- Overall more syntax, no static keyword

```
val a: String = "abc"
val b: String? = null

class Orchestrator(
    private val orchestratorEmail: String,
    private val password: String,
    private val ticker: Ticker,
    kafkaConfig: KafkaSSLConfig
) {
    private val consumer =
        AppKafkaConsumer(kafkaConfig, "test-consumer-group")
        //...
}

delay(1.seconds)

private inline fun <T> withSemaphore(semaphore:
    Semaphore, action: () -> T): T

fun close() = connection.close()
```

Takeaway: Would be nice to have more language-level functional programming support, but Arrow is very powerful

- Disclaimer: this is not idiomatic Kotlin!
- DSLs for `Either`, `Option` types
 - Made possible by Kotlin DSL syntax
 - F# computation expressions comparison
- `Either.catch`: FP idiomatic exception handling
 - No language-level support
 - Other niceties provided here

```
suspend fun getStartingState(  
    exchangeRequestDto: ExchangeRequestDto  
): Either<ClientFailure, StartingState> {  
    return either {  
        val quote =  
            getQuote(exchangeRequestDto).bind()  
        val positions =  
            getUserLongPositions(exchangeRequestDto).bind()  
        val orders =  
            getUserOrders(exchangeRequestDto).bind()  
        StartingState(quote, positions, orders)  
    }  
}
```

Takeaway: Would be nice to have more language-level functional programming support, but Arrow is very powerful

- Disclaimer: this is not idiomatic Kotlin!
- DSLs for `Either`, `Option` types
 - Made possible by Kotlin DSL syntax
 - F# computation expressions comparison
- `Either.catch`: FP idiomatic exception handling
 - No language-level support
 - Other niceties provided here

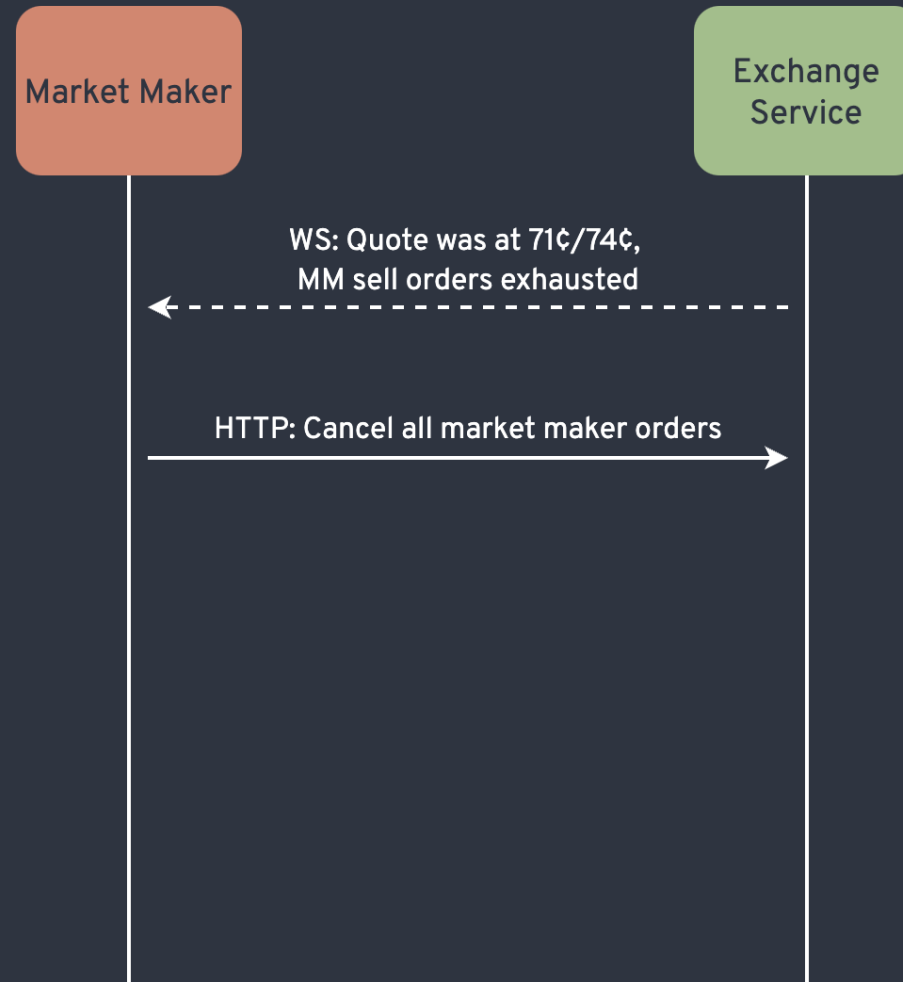
```
either {  
    val passwordHash =  
        BCrypt.hashpw(dto.password, BCrypt.gensalt())  
        ensure(!emailPresent(dto.email)) {  
            raise(400 to "Account for `${dto.email}`  
already exists")  
        }  
    Either.catch {  
        db.query { conn -> //... }  
    }.mapLeft { error -> //... }  
}
```

Takeaway: market maker initially worked itself into a tizzy that showed how important request/response timing was

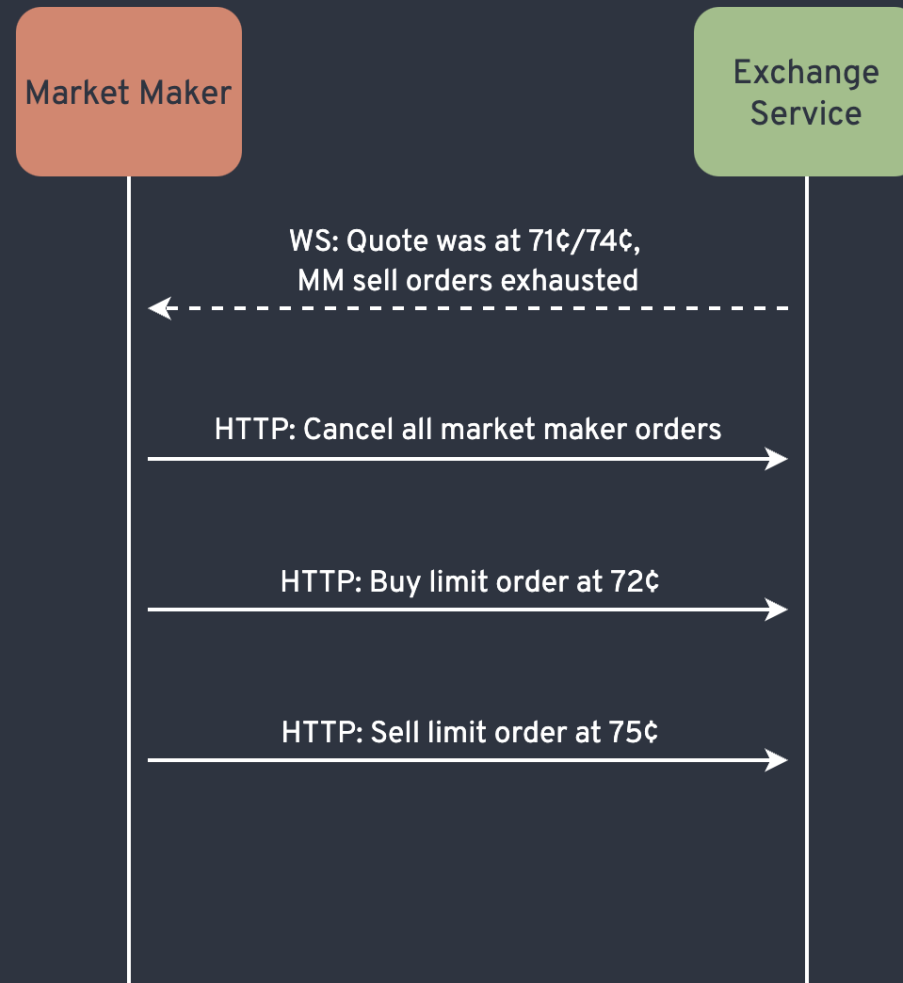
	Price	Volume
Asks	76¢	200
	75¢	15
	74¢	10
3¢ spread		
Bids	71¢	3
	70¢	5

```
data class Quote(  
    val ticker: Ticker,  
    val bid: Int,  
    val ask: Int,  
)
```

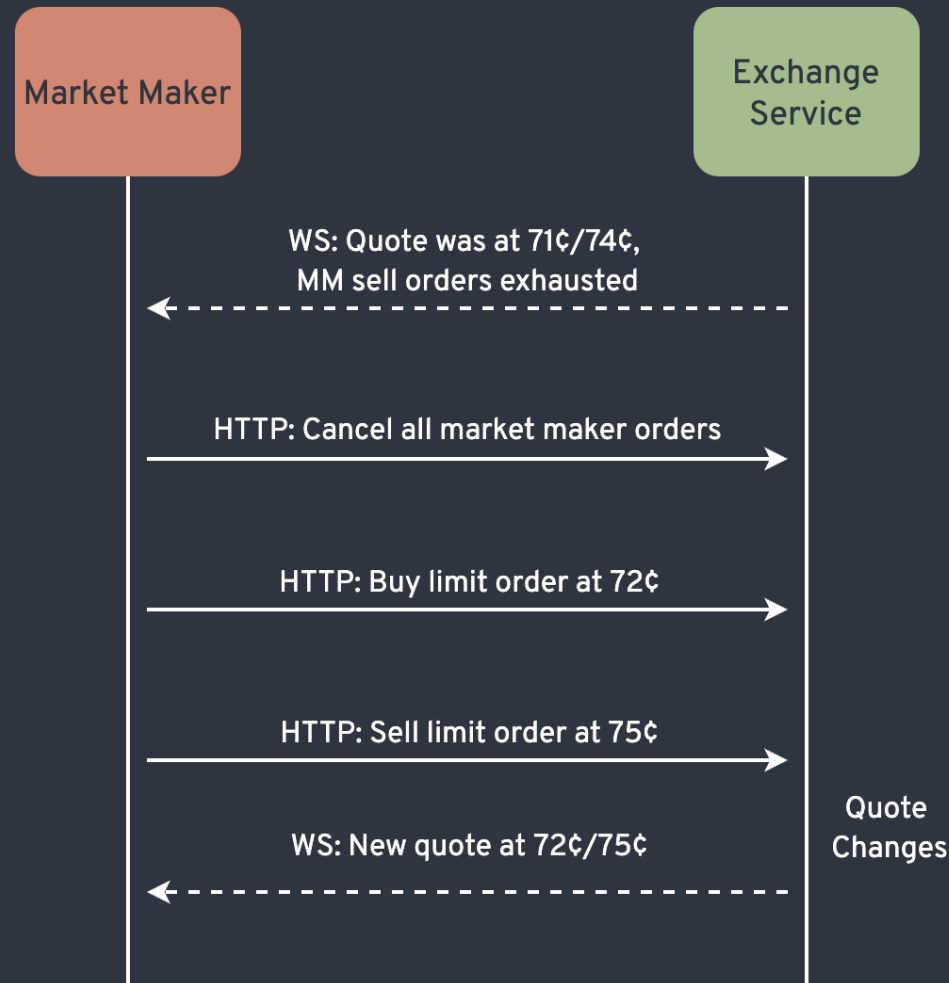

Takeaway: market maker initially worked itself into a tizzy, showed importance of important request/response timing



Takeaway: market maker initially worked itself into a tizzy, showed importance of important request/response timing



Takeaway: market maker initially worked itself into a tizzy, showed importance of important request/response timing



Takeaway: market maker initially worked itself into a tizzy that showed how important request/response timing was

	Price	Volume
Asks	76¢	200
	75¢	15
	74¢	10
3¢ spread		
Bids	71¢	3
	70¢	5

```
data class Quote(  
    val ticker: Ticker,  
    val bid: Int,  
    val ask: Int,  
    val triggerTimestamp:  
        Long,)
```

Workflow takeaway: very helpful to record setbacks in README, and Claude served as my ‘best available reviewer’

tmp.txt
806 lines

TXT

I From the perspective of a very blunt Jane Street trader, provide feedback on the following

The Bad:

- Your "fix" in the market maker is completely broken. You're trying to enforce a spread by manipulating quotes, but you're doing it wrong:

```
kotlin
val bid = midpoint - round((quote.bid - quote.ask) / 2.0).toInt()
```

This makes no mathematical sense. `quote.bid - quote.ask` is always negative for valid quotes, so you're adding to the midpoint, not subtracting.

- You're using `round()` on integer arithmetic that's already been divided, which is pointless
- The spread enforcement logic assumes you want a fixed spread, but markets don't work that way

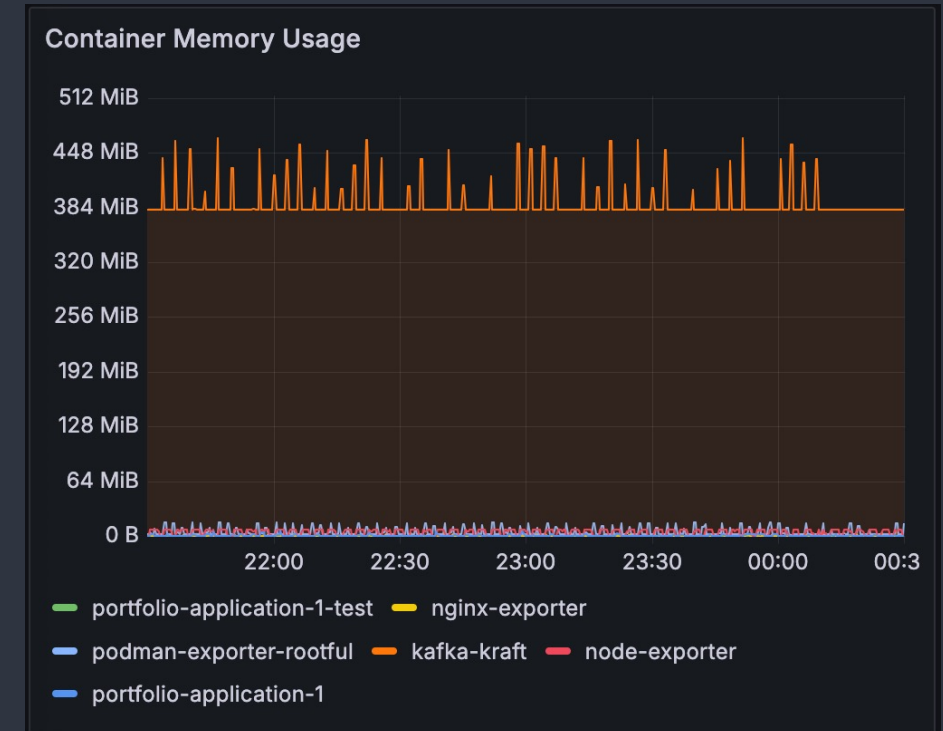
README

ac5e8f8

- The market order matching logic in `getMarketOrderProposal` could benefit from better organization
- The error handling could be more granular, could add:
- Validation errors for order parameters
- Market condition errors (e.g., circuit breakers)
- System state errors (e.g., order book inconsistency)
- `OrderPartialFilled` not utilised
- Consider Arrow's `ValidatedNel` class

Other takeaways: overall satisfied with lessons learned with the technologies used, but some Kafka annoyances

- Concurrency & state
 - Maintaining invariants, large state space challenge
 - Downey's 'Little Book of Semaphores' quite useful
- Javalin
 - Generally fine
 - No coroutine support, Ktor better
- SQLite
 - Easy workflow: checking into Git, rollbacks
 - Missed opportunity to use ksqldb
- Kafka
 - Could not successfully place behind HTTP proxy
 - VPS hosted, single-broker, keystore authentication
 - Inspired my 'Kafka in One File' blog post
 - Memory hog



Final Note and Questions

- Further development possibilities
 - Actual client UI
 - More event stream persistence
 - More order types
 - Find some way to shoehorn F# back in
- Thank you for coming to this talk!
- Questions?