Streaming REST Responses with Kotlin and Ktor

Peter Wall – Tyro Payments – 2023-07-19

First of all...

I would like to pay my respects to the Gadigal people of the Eora nation, as the traditional owners of the land on which we are meeting today.

About me...

My name is **Peter Wall**, and my pronouns are he/him.

I am an engineer in the Banking division of Tyro (I have been here for 4 years) working mainly on the server systems that support Tyro's banking operations.

Earlier in my career I spent 6 years as a Senior Java Architect with Sun Microsystems, both here in Australia and in the US, but despite previously being a strong advocate for Java, I am now a total convert to Kotlin, since first encountering it about 5 years ago.

tyro

About this presentation...

First I will give a quick introduction to Kotlin. I am assuming that the audience at a Java Meet-Up will have Java as your first language, but you will probably also have investigated Kotlin to some extent, so this will just focus on the language features relevant to the demos.

Next, I will give an introduction to Ktor, the framework that we will be using for the topic that all this has been leading up to:

Streaming JSON objects in Kotlin and Ktor.

Kotlin Basics

Some basic Kotlin syntax:

```
fun getCustomer(id: String): Customer {
    val customer = client.getCustomer(id)
    return customer
}
```

- Functions are introduced by `fun`, variables by `var` and values by `val`
- Types are specified following the name, or the parameter list of the `fun`
- Functions, variables and values are public by default
- No semicolons!

Kotlin Types and Nullability

Extending that example:

```
fun getCustomer(id: String): Customer {
    val customer: Customer? = client.getCustomerOrNull(id)
    return customer ?: Customer(id)
}
```

- If the type name is followed by question mark, the value may be `null`
- The `?:` operator takes the left side if it is not `null`, otherwise the right side
- No `new` operator; the constructor just uses the class name

Kotlin Extension Functions

Kotlin allows functions to be declared as members of an existing class:

```
fun Customer.getDisplayName(): String {
    return nickname ?: fullName
}
```

- Extension functions do not allow access to private class members
- They are not polymorphic – you can not declare an extension function on a base class and expect the derived class version to be used

Kotlin Lambdas

A function that takes a lambda parameter:

```
fun select(list: List<Customer>, predicate: (Customer) -> Boolean):
        Customer { /* select customer from list */ }
```

May be called with:

```
val customer = select(list, { customer -> customer.balance < 0 })
```

But the convention is to place the lambda **outside** the parentheses:

```
val customer = select(list) { customer -> customer.balance < 0 }
```

Kotlin Coroutines

Kotlin includes a coroutine mechanism, which allows a large number of lightweight processes to run concurrently, with relatively low resource consumption.

```
launch {
    val customer: Customer = remoteClient.getCustomer(id)
    displayCustomer(customer)
}
```

The lambda supplied to the `launch` function will be executed in a new coroutine, and the functions in that lambda must be **non-blocking**.

Kotlin Non-Blocking Functions

A Kotlin function may be labelled a `suspend fun` to indicate that it is a non-blocking function.

```
suspend fun findCustomer(id: String): Customer {}
```

Such functions may only be invoked within a coroutine context, and they may not perform any blocking operations.

Traditional I/O functions, for example REST client calls or database lookups, are blocking operations, so special non-blocking versions must be used within a `suspend fun`.

Combining several of these concepts...

Consider this function.

```
suspend fun forEachCustomer(consumer: suspend Customer.() -> Unit) {
    /* loop over list of customers, invoking "consumer" for each one */
}
```

The lambda parameter is invoked as a non-blocking extension function:

```
forEachCustomer {
    if (balance < 0)    /* "this" is a "Customer" */
        sendToExceptionReport()
}
```

Ktor

Ktor is a framework for the creation of server and client applications, built around the Kotlin coroutine mechanism.

```
fun main() {
    embeddedServer(Netty, port = 8080) {
        routing {
            get("/") {
                call.respondText("Hello, world!")
            }
        }
    }.start(wait = true)
}
```

Ktor Information

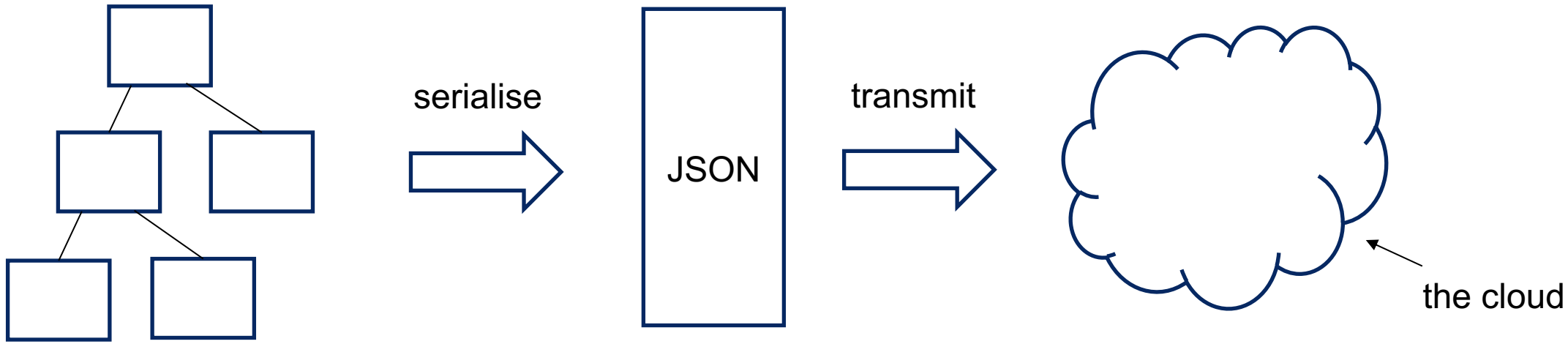
Ktor is a product of JetBrains, the company behind IntelliJ IDEA and Kotlin itself.

For more information, see:

https://ktor.io/

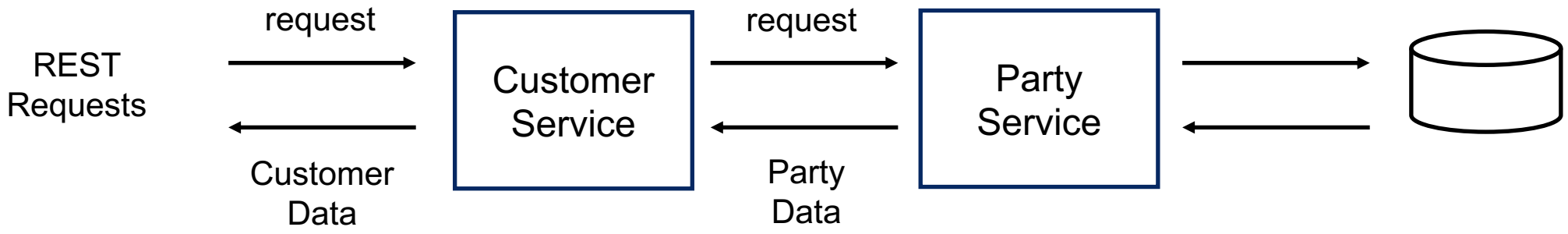
Conventional JSON Output Serialisation

When serialising a JSON response from a REST call, the conventional approach requires serialising the internal form of the response into a JSON string, and then returning the entire string:



The Demonstration Application

For the purposes of these demonstrations, we are imagining a system that has a Party Service which holds information on parties (individuals or organisations) and a Customer Service that retrieves party information from the Party Service, combines it with account information and returns the combined Customer details.



The Endpoint Definition in Ktor

In Ktor, the code for a “get list” endpoint (in a **`Routing`** block) would look something like:

```
get("/party/list/{ids}") {
    val ids = call.parameters["ids"] ?:
            throw IllegalArgumentException("No ids")
    log.info { "GET /party/list/$ids" }
    val list: List<Party> = config.partyService.getList(ids.split('.'))
    call.respond(list)
}
```

With This Service

And the service code is simple, but there may be a lot of complexity hidden behind the "**getParty**" function in this example (multiple database accesses, or REST client calls):

```
suspend fun getList(ids: List<String>): List<Party> {
    return ids.map { id ->
        delay(3000) // to simulate a slow data acquisition process
        getParty(id)
    }
}
```

Demonstration 1

For anyone trying to reproduce this demonstration:

1. Download the project https://github.com/pwall567/ktor-demo-2
2. Build and run the project (will start a server on port 8102).
3. In a terminal window, run: `curl -N http://localhost:8102/party/list/1.2.3.4`
4. The server log will show when each record is retrieved, but the JSON output will be displayed only when all results are available.

Can We Improve On That?

We used a list in that example, and under normal circumstances, we would assemble the entire list and hand it over for serialisation.

But what if each item in the list took a long time to create (multiple database accesses, for example)? With this approach, we would have to wait until the last item in the list was completed before the first was sent to the caller.

Kotlin has coroutines – can we make use of this mechanism to make the data available sooner?

Using a Kotlin “Flow”

tyro

```
get("/party/flow/{ids}") {
    val ids = call.parameters["ids"] ?:
            throw IllegalArgumentException("No ids")
    log.info { "GET /party/flow/$ids" }
    val flow = flow { // this lambda is executed in a new coroutine
        config.partyService.getStream(ids.split('.')) {
            log.info { "Sending ${it.id}" }
            emit(it)
        }
    }
    call.respond(flow) // the "Flow", even though it's still being filled
}
```

And in the Service

The service for this approach is simple:

```
suspend fun getStream(ids: List<String>, consumer: suspend (Party) -> Unit) {
    for (id in ids) {
        delay(3000)
        consumer(getParty(id))
    }
}
```

Each “**Party**” object will be added to the “**Flow**” as it becomes available, and will be serialised and sent in the response immediately.

Demonstration 2

For anyone trying to reproduce this demonstration:

1. Download the project https://github.com/pwall567/ktor-demo-2
2. Build and run the project (will start a server on port 8102).
3. In a terminal window, run: `curl -N http://localhost:8102/party/flow/1.2.3.4`
4. The server log will show when each record is retrieved, and JSON output will be displayed as each party record arrives.

The “kjson-ktor” Library

The ability to deserialise and stream data asynchronously is not part of the standard Ktor library; it comes from “**kjson-ktor**”:

https://github.com/pwall567/kjson-ktor

This library provides adapters allowing the use of the “**kjson**” library with Ktor, and also allows the streaming of asynchronous data as shown here.

Kotlin has two mechanisms for passing data asynchronously between coroutines: “**Channel**” and “**Flow**”. Either mechanism can be used for the serialised output. The terminology and the functions differ, but the principles are the same.

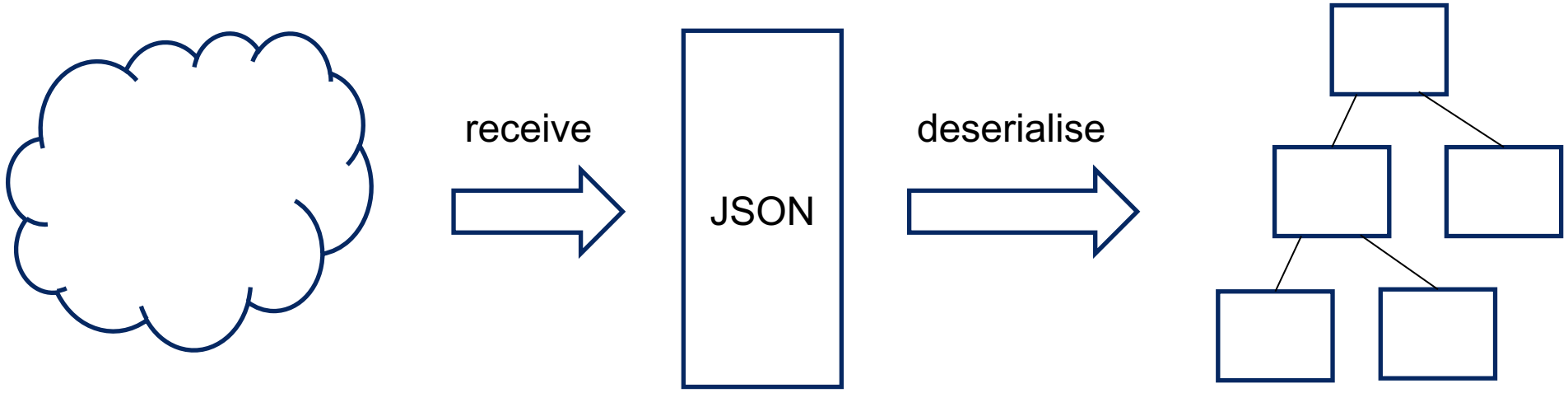
Output of Non-JSON Data

The “**kjson-ktor**” library also includes the function “**respondStream**” to help with the streaming output of non-JSON data. For example, if an endpoint wants to output an opaque token:

```
get("/token/{id}") {
    val id = call.parameters["id"] ?: throw IllegalArgumentException("No id")
    val token: String = tokenisationService.getToken(id)
    call.respondStream(contentType = ContentType.Text.Plain) {
        output(token)
    }
}
```

Serialisation is Easy; Deserialisation is Harder

In most circumstances, JSON input must be read completely into a string, and then parsed into its eventual form:



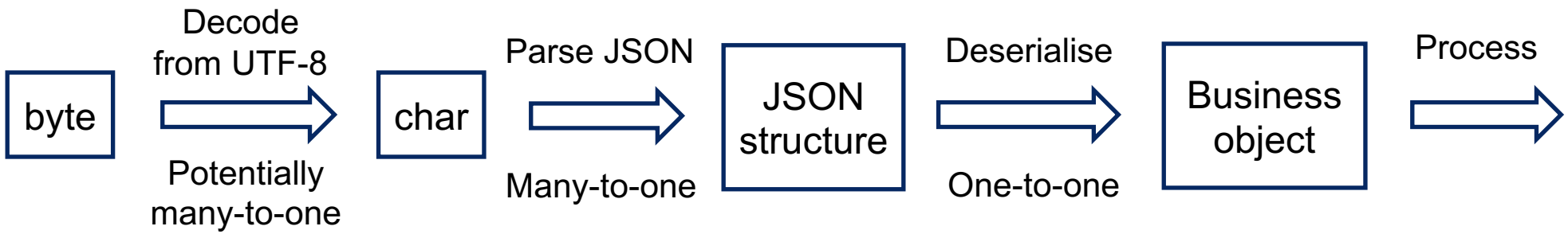
Client Code in Ktor

The Ktor code for the client call in this simple case is straightforward:

```
suspend fun getList(ids: String): List<Party> {
    val response = client.get("$PARTY_SERVER_BASE_URI/party/list/$ids")
    when (response.status) {
        HttpStatusCode.OK -> return response.body()
        else -> throw IllegalStateException("Something went wrong")
    }
}
```

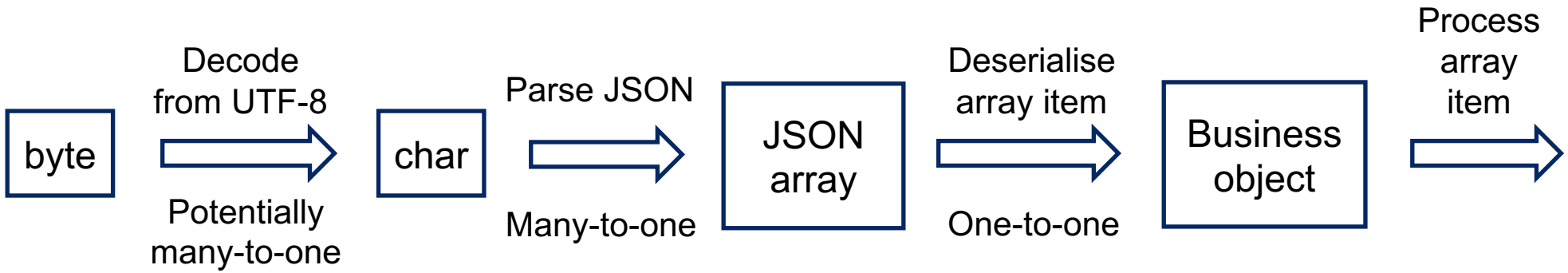
Streaming Input

Streaming input is more complicated. It requires that each byte be decoded from its transmission encoding, parsed into JSON and deserialised into the business object form, all on the fly:



Streaming Array Input

The nature of JSON means that we must always wait until we see the closing brace of an object before we consider the object to be complete. But when the JSON input consists of an array, each item in the array may be processed as soon as that item is completed:



Streaming Asynchronous Input

The reading of asynchronous array input requires an additional function:

```
suspend fun getFlow(ids: String, consumer: suspend (Party) -> Unit) {
    client.receiveStreamJSON<Party>("$PARTY_BASE_URI/party/flow/$ids") {
        log.info { "Received ${it.id}" }
        consumer(it)
    }
}
```

The "**receiveStreamJSON**" function is part of the "**kjson-ktor**" library; it expects the input stream to be a JSON array, deserialises each array item to the required type and passes the object to a lambda function in real time.

`receiveStreamJSON`

The full function signature of **`receiveStreamJSON`** is as follows (note that most parameters have appropriate defaults):

```
suspend inline fun <reified T : Any> HttpClient.receiveStreamJSON(
    urlString: String,                                  // the URL
    method: HttpMethod = HttpMethod.Get,                // the HTTP method
    body: Any = EmptyContent,                           // a possible POST or PUT body
    headers: Headers = Headers.Empty,                   // request headers
    expectedStatus: HttpStatusCode = HttpStatusCode.OK, // the expected status
    config: JSONConfig = JSONConfig.defaultConfig,      // config for JSON conversion
    noinline consumer: suspend (T) -> Unit              // lambda to take each item
)
```

Demonstration 3

For anyone trying to reproduce this demonstration:

1. Download the project https://github.com/pwall567/ktor-demo-2
2. Build and run the project (will start a server on port 8102).
3. Download the project https://github.com/pwall567/ktor-demo-1
4. Build and run the project (will start a server on port 8101).
5. In a terminal window, run: `curl -N http://localhost:8101/customer/flow/1.2.3.4`
6. The server log in each window will show when each record is retrieved, and the JSON output will be displayed as each customer record arrives.

JSON Lines

The JSON Lines specification allows multiple JSON values to be specified in a single stream of data, separated by newline characters. For example, events may be logged as a sequence of objects on separate lines:

```
{ "time":"2023-06-24T12:24:10.321+10:00", "eventType":"ACCOUNT_OPEN", "accountNumber":"123456789” }
{ "time":"2023-06-24T12:24:10.321+10:00", "eventType":"DEPOSIT", "accountNumber":"123456789",
  "amount":"1000.00” }
```

The “**kjson-ktor**” library includes functionality to output a stream of data in JSON Lines form.

JSON Lines Output

tyro

The “**`kjson-ktor`**” library includes functionality to output a stream of data in JSON Lines form:

```
get("/party/flow/{ids}") {
    val ids = call.parameters["ids"] ?: throw IllegalArgumentException("No ids")
    log.info { "GET /party/flow/$ids" }
    val flow = flow {
        config.partyService.getStream(ids.split('.')) {
            log.info { "Sending ${it.id}" }
            emit(it)
        }
    }
    call.respondLines(flow) // this is the only line that differs from the array output form
}
```

JSON Lines Input

The function for asynchronous array input has a form that accepts input in JSON Lines form:

```
suspend fun getFlow(ids: String, consumer: suspend (Party) -> Unit) {
    client.receiveStreamJSONLines<Party>("$PARTY_BASE_URI/party/flow/$ids") {
        log.info { "Received ${it.id}" }
        consumer(it)
    }
}
```

Only the function name changes from the array input form.

Demonstration 4

For anyone trying to reproduce this demonstration:

1. Download the project https://github.com/pwall567/ktor-demo-2
2. Build and run the project (will start a server on port 8102).
3. Download the project https://github.com/pwall567/ktor-demo-1
4. Build and run the project (will start a server on port 8101).
5. In a terminal window, run: `curl -N http://localhost:8102/party/lines/1.2.3.4`
6. The server log in each window will show when each record is retrieved, and the JSON output will be displayed in JSON Lines form as each record arrives.

Putting It All Together

Remember the "**`respondStream`**" function to output non-JSON data in a streaming context?

If we connect a *Mustache* template processor to that function, we can:

- read a stream of **`Customer`** objects from one server, which causes
- a streaming read of **`Party`** objects from a second server, and
- each **`Party`** object received from the second server is deserialised and used to create a **`Customer`** object, which
- is passed to a *Mustache* processor, which formats the **`Customer`** into HTML using a template, so that
- the first **`Customer`** object is displayed, formatted, as it arrives, before the last object has been sent from the **`Party`** server.

Demonstration 5

For anyone trying to reproduce this demonstration:

1. Download the project https://github.com/pwall567/ktor-demo-2
2. Build and run the project (will start a server on port 8102).
3. Download the project https://github.com/pwall567/ktor-demo-1
4. Build and run the project (will start a server on port 8101).
5. In a browser window, go to: `http://localhost:8101/display-lines/1.2.3.4`
6. The server log in each window will show when each record is retrieved, and the formatted output will be displayed as each customer record arrives.

And the Code to Achieve This...

```
get("/display/{ids}") {
    val ids = call.parameters["ids"] ?: throw IllegalArgumentException("No ids")
    log.info { "GET /display/$ids" }
    val count = Counter(units = "customer")
    val flow = flow {
        config.customerAccountService.getAccountFlow(ids.split('.')) {
            emit(it)
            count.increment()
        }
    }
    val mustacheContext = mapOf("list" to flow, "count" to count)
    call.respondStream {
        config.mustacheTemplate.coRender(mustacheContext, this)
    }
}
```

Further Exploration

The code for the libraries and the demo projects is available on GitHub, for anyone who wants to explore these concepts further:

https://github.com/pwall567/ktor-demo-1 (the Customer server)

https://github.com/pwall567/ktor-demo-2 (the Party server)

https://github.com/pwall567/kjson-ktor (kjson-ktor library)

https://github.com/pwall567/kjson (kjson library)

Questions?

Thank you