Ktor

Ktor is a Kotlin-specific framework for building asynchronous client- and server-side web applications.

- Ktor Basics
- Ktor Architecture & Lifecycle
- Kotlin Basics
- Ktor REST API
- Ktor Authentication and Authorization
- Signing JSON Tokens with RSA

Ktor Basics

Ktor is a Kotlin based asynchronous web framework.

Backends need to be versatile and scalable. Developers should have a 'microservice mindset' to create more maintainable backend services.

Why Ktor

- Kotlin based - concise and fun language to use
- Lightweight - low server start up time
- Asyncronous through co-routines - does NOT create a thread per request/response - scalable
- Runs on JVM and uses Gradle - runs on basic infrastructure (i.e. a docker container)
- Well documented and Easy to use (At least compared to Ruby and PHP). Jetbrains always does a good job with their documentation.
- Modular - default project starts with one module
- Rich in features - autentication, data access, content negotiation, & more
- Flexible - currently supports 4 different servlet containers: Tomcat, Netty, Jetty, and CIO
- In development since 2015 and is well-maintained

What is a Servlet Container?

- Web Container / Servlet Container / Servlet Engine : is used to manage the components like Servlets, JSP. It is a part of the web server.
- Web Server / HTTP Server: A server which is capable of handling HTTP requests, sent by a client and respond back with a HTTP response.
- Application Server / App Server: can handle all application operations between users and an organization's back end business applications or databases.It is frequently viewed as part of a three-tier application with: Presentation tier, logic tier,Data tier

Which to Choose?

Tomcat

- The most popular open source project under Apache
- Very well documented
- JSP parsing is very fast
- Flexible and scales easy
- Spring Boot uses Tomcat by default

Jetty

- Uses less memory and is more lightweight thus offering speed and scalabillity
- Small and efficent with low maintaince costs
- Can server as a asyncronous server with some effort
- Open source with good community and support. Originated from the Eclipse foundation
- Widely used, though less so than Tomcat
- Built into several frameworks; GWT, JRuby, Grails, Scala/Lift & more
- If you want a light HTTP servlet container use Jetty.

Netty

- An **asynchronous**, event-driven, non-blocking I/O, network application framework.
- You can write your own servlet container, but technically Netty in itself is a server framwork not a container like Jetty
- Greatly simplifies network programming such as TCP and UDP socker servers.
- If you deal a lot with network protocols and want it to be non-blocking use Netty (usually for high-performance cases).

CIO

- Stands for Configuration Interface Object
- Made by Oracle. Very poorly documented in my opinion. can't find any example usages for Java Servlets outside the documentation.
- The benifit of this one is is runs on Android and JVM
- I think Ktor support for CIO is still flakey as well. I would avoid this one.

For Ktor the buzzword is *asynchronous*, and I plan on experimenting with socket programming, so my choice is Netty.

Note: I think there is also support for Apache and Android as a buit in http client engine but the documentation doesn't go into great detail

Starting a Project

Either go to start.ktor.io or use the Intellij plugin to generate a project template. Select whatever add-ons you want but always make sure 'routing' is selected

Other add-ons I would concider basic:

- Status Pages - Allow the server to respond to thrown exceptions
- CallLogging - Logs Client Requests
- CSS/HTML DSL - You'll need these if you plan on hosting your html directly off ktor.
- PartialContent - Adds support for breaking up content and using paging
- Authentication Basic - Good starting point for security if you don't feel like setting up ldap
- ContentNegotation - Automatic type conversion
- GSON - Helper library for handling JSON data

The Kotlin Coroutine

The Kotin corourine is an asynchronous non-blocking job that can run on the same thread as other coroutines.

- Coroutine is acheived in Ktor using the kotlinx.coroutines libraray
- Runs in a context - many coroutines can share context and thread pool
- Ktor uses coroutines through the framework, which is why it preforms so well.
- A coroutine function is decorated with "suspend"
- Other suspend functions can only be run from suspend functions
- A coroutine is usually start with the launch function in the DEFAULT context
- A coroutine can be tested or run by the main function by using the runBlocking function

Installation and Configuration of Features

- Most features require a Gradle/Maven dependecy
- use the `install` function. Configure it with a trailing Lambda as an argument.

```
    install(CallLogging) {
        level = Level.INFO
        filter { call -> call.request.path().startsWith("/") }
    }
    install(Routing) {
        get("/") {
            call.respondText("Good evening World")
        }
    }
```

Common Features

- A common feature has one or more helper functions
- Routing is a common feature
- For a feature with helper functions, we can configure this feature after calling it's install function
- For example:

```
install(Routing) {
    get("/") {
        call.respondText("Good evening World")
    }
}
```

can be seperated into `install(Routing)` and

```
routing {
    get("/") {
        call.respondText("Good evening World")
    }
}
```

Although, this is a bad example, because Routing is installed by default so there's no need for `install(Routing)`

Custom Features

You can build and install your own custom features. Most features intercept the pipeline at the Application.Features phase. That's all I'm going to say about that.

Autoreload

- Detects changes on the classpath, meaning changes are made without having to restart the server. Very handy when making HTML changes
- Autoreload is an experimental feature only available with JDK8+. It can be enabled with Intellij or within the Gradle buiild
- Of course there is a cost to preformance. Do not use this feature in production or while benchmarking
- application.conf
  - Chose folders to watch for classpath changes: `watch = [/module1, /module2]`
  - Usually we only deal with one module, so an example conf would be something like:

```
tor {
deployment {
port = 8080
port = ${?PORT}
watch = [ / ]
}
application {
modules = [cpm.exmple.ApplicationKt.module]
}

``
```

Gradle: Recompile

- Gradle can recompile the project on code changes: `gradle -t installDist`
- This compiles and listens for source code changes
- Using Gradle gives the most flexible autoreload setup, but requires JDK8+

Call Logging - Log All Incoming Requests

- Enables us to choose specific routes for which we want to enable logging. You can have as many filters as you would like.

```
install(CallLogging) {
level = Level.INFO
filter { call -> call.request.path().startsWith("/mysection1") }
}
```

Metrics Statistics on the Usage of Endpoints

- There is a feature called DropwizardMetrics that monitors performance statistics like:
  - Number of threads
  - Number of Calls per endpoint
  - Memory Usage
  - And more...
- It can be configured to log statistics to a file or other places like JMX
- JXM Reporter allows us to expose metrics to the JMX so we can use the JConsole or jvisualvm to view metrics
- Slf4j Reporter outputs the metrics in the log every X seconds
- The following dependencies must be implemented:

```
implementation("io.ktor:ktor-metrics:$ktor_version")
implementation("io.dropwizard.metrics:metrics-jmx:4.0.0")
```

Then install the feature:

```
install(DropwizardMetrics) {
    Slf4jReporter.forRegistry(registry)
        .outputTo(log)
        .convertRatesTo(TimeUnit.SECONDS)
```

```
        .convertDurationsTo(TimeUnit.MILLISECONDS)
        .build()
        .start(15, TimeUnit.SECONDS)

    JmxReporter.forRegistry(registry)
        .convertRatesTo(TimeUnit.SECONDS)
        .convertDurationsTo(TimeUnit.MILLISECONDS)
        .build()
        .start()
}
```

- This information is very useful for debugging performance
- Run `jconsole` in the termnial and select the running process to see visualizatons on performance 3ffb8c0b3893bc3cc5d520cca391670b.png

Ktor Architecture & Lifecycle

Architecture

a75ce66dd598d66738b87023e85a44b7.png

- EngineMain Class
  - Used to run the application
  - Loads application.conf file
  - Supported Engines:
    - CIO: io.ktor.server.cio.EngineMain.main
    - Jetty: io.ktor.server.jetty.EngineMain.main
    - Netty: io.ktor.server.netty.EngineMain.main
    - Tomcat: io.ktor.server.tomcat.EngineMain.main
- ApplicationEngine
  - In charge of running the application
  - Uses the configuration to decide which ports to listen to
- ApplicationEngineEnvironment
  - Immutable
  - Contains a Classloader, Configuration, Logger, Monitor (event bus for a port and application information that can be subscribed to), Connectors, Monitor, and installed Modules
- ApplicationCallPipeline - Contains application phases which can be intercepted
- Contains a context: ApplicationCall class
  - This class has a reference to the application, the request, the response, the attributes, and parameters
  - Phases:
    - Setup: Prepares the call and processes and attributes
    - Monitoring: Logging metrics and error handling
    - Features: Infrasructure features - most features intercept at this phase
    - Call: Processes a call and sends a response
    - Fallback: Handles unprocessed calls
  - Monitor (Event Bus):
    - Raises application events
    - Enables us to subscribe to the following events:
      - ApplicationStarting
      - ApplicationStarted
      - ApplicationStopPreparing
      - ApplicationStopping
      - ApplicationStopped

    - In between these events a database could be cleaned or emails could be sent, for ex

Ktor Routes

- Routing is a built in feature that helps us structure the page request handling
- Information about the request is extracted like the header and request parameters
- Routes are matched up against the extracted information and the route configuration
- Route functions:
    - route(HttpMethod.Get, path) { do something.. }
    - Shortcut functions: get, post, put, delete, head, and options
    - Use trailing lambdas to create the response
- Routing tree enables us to setup complex nested routes
- Builder functions can be combined and nested
- We can trace why a certain route was chosen with the trace function `trace { application.log.trace(it.buildText()) }`
- Path segments
    - Optional: `/greeting/{myParamId?}` - If the path segment exists the paramater myParamId will be set to the value
    - Wildcard: `/weather/*/asia` - Mathes a path starting with weather and ending with asia
    - Tailcard: `/weather/{myParamId...}` - myParamId will be set to the rest of the URL. Can also be used without the parameter (`/weather/{...}`)
- If there are multiple path matches the route of "highest qaulity" will be chosen
    - If the header has an "Accepts"-key to perfer a type of content:

```
accept(ContextType.Text.Plain) { ... }
accept(ContextType.Text.Html) { ... }
accept(ContextType.Application.Json) { ... }
```

Builder Functions

- route(path) - segments are on the path and context is within the lambda, which could contain more routes
- method(verb) - segments on HTTP method
- param(name, value) - segments on query parameter
- param(name) - segments on query parameter key
- optionalParam(name) - segments a query parameter if it exists
- header(name, value) - segments on header content Example:

```
    routing {
□□
        route("/weather") {
            route("/asia") {
                // this will only execute if the specified systemtoken is present
                header("systemtoken", "weathersystem") {
                    handle {
                        call.respondText("The weather is sunny")
                    }
                }
            }
            route("/europe", HttpMethod.Get) {
                // if the parameter name is not present call the other handle function
                param("name") {
                    handle {
                        var name = call.parameters.get("name")
                        call.respondText("The weather is $name")
                    }
                }
                handle {
                    call.respondText("The weather is rainy")
                }
            }
            route("/usa") {
                get {
                    call.respondText("The weather is rainy")
                }
            }
        }
□}
```

Sample Request: `curl -H "systemtoken: weathersystem" -X GET "localhost:8080/weather/asia"`

Calling 3rd Party REST Services

- The HttpClient can be installed with different types of engines
- We can configure the HttpClient to deserialize a JSON response to an instance of a class with Gson or Jackson
- Ex using Apache:

```
val client = HttpClient(Apache) {
    install(JsonFeature) {
        serializer = GsonSerializer()
    }
}
```

- Supported Engines:
  - Apache - "io.ktor:ktor-client-apache:$ktor_version"
    - Supports HTTP/1.1 and 2
  - CIO - "io.ktor:ktor-client-cio:$ktor_version"
    - Supports HTTP/1.x
  - Jetty - "io.ktor:ktor-client-jetty:$ktor_version"
    - Supports HTTP/2
- If no engine is speicified, the default engine will be used (if any availble)
- When running in JVM, a ServiceLoader will look for an engine on the classpath and choose by sorting in alphabetical order
- On native systems (IOS, Android) an engine will be found by static linkage

Testing with the MockEngine

- The MockEngine can be used to choose a static response for a given URL, great for Unit Testing

```
val client = HttpClient(MockEngine)
{
    engine {
        addHandler { request ->
            when (request.url.fullUrl) {
                "https://example.org/" -> {
                    val responseHeaders = headersOf("Content-Type" to lostOf(ContentType.Text.Plain.toString()))
                    respond("Hello, world", headers = responseHeaders)
                }
                else -> error("Unhandled ${request.url.fullUrl}")
            }
        }
    }
```

```
}
```

Kotlin Basics

Kotlin Class Extensions

- Extends the functionality of an existing class

```
fun Int.addFive() : Int {
    return this + 5
}
```

- Does not actaully change the code of the class
- Provides a function that can be called on instances of the class

When are Class Extensions Useful in Ktor?

- To seperate business specific routes and logic from the rest of the routes

Coroutine Contexts

- DEFAULT: Number of threads = number of CPI cores - use this for calculations or if you are uncertain about which context to use
- IO: Number of threads = 64 or number of cores (whichever is larger) - use this for rest communication or storing data to a file or database
- MAIN: Number of threads = 1 - is mainly used in android apps to interact with user interface

Sample Code:

```
import kotlinx.coroutines.*
import kotlin.random.Random

fun main(args: Array<String>) = runBlocking {
    // 64 Threads in IO
    withContext(Dispatchers.IO) {
```

```
        repeat (100_000) { // 100_000 = 100,000
            launch {
                firstcoroutine(it) // 'it' will be the current iteration
            }
        }
        println("End of withContext")
    }
    println("End of main function")
}

suspend fun firstcoroutine(id: Int) {
    delay(Random.nextLong()%2000) // The delay is a random number less than 2 seconds
    println("first $id")
}
```

Running this code gives an output something like:

```
first 0
first 1
first 2
first 5
first 6
first 7
first 8
first 10
first 9
first 11
first 13
...
```

Notice how the sequence falls out of order? This is threading and Kotlin coroutines in action.

Ktor REST API

REST - Respresentational State Transfer

- The path (URL) is the route to a resource
- A resource is any kind of logical object in the business model
- HTTP is most often used as transport protocol
  - GET - retrieve data
  - PUT - create or update an object
  - POST - submit data or update an object
  - DELETE
  - HEAD - get only the header (no body)
  - OPTIONS - queries which methods are possible
  - PATCH - updates a partial resource
- JSON is always the preferred data format, it's just so darn pretty
- API - Application program interface
- HTTPS encrypts communications between server and client
- Is often made secure with basic or OAUTH2 authenication

Ktor and Hypermedia

Hypermedia is an extension of the term hypertext, it is a medium of delivering information to a browser that could include graphics, audio, video, plaintext, and hyperlinks.

- HATEOAS - Hypermedia as the Engine of Application State
- Ktor does not have a standard/built-in feature that generates a HATEOAS response yet

Content Negotiation

- Register (map) content converters with content types

```
install(ContentNegotiation) {
    register(TheContentType,  TheContentTypeConverter()) {
        //configure converter
    }
}
```

- ContentNegotation feature is part of standard ktor library
- Example Converters:
  - GSON converter - io.ktor:ktor-gson
  - Jackson converter - io.ktor:ktor-jackson
  - There is no standard XML converter, but you could define one by importing com.fasterxml.jackson.dataformat:jackson-dataformat-xml
- Use "Content-Type" header to find the correct receive converter
- Use the "Accept" header to find the matching send converter

Custom Converters

- To write a custom converter implement the interface ContentConverter
  - Implement method convertForSend
  - Implement method convertForReceive

```
interface ContentConverter {
    suspend fun convertForSend(context: PipelineContent<Any, ApplicationCall>, contentType: ContentType, value:
Any): Any?
    suspend fun convertForReceive(context: PipelineContext<ApplicationReceiveRequest, ApplicationCall>): Any?
}
```

An example with the jackson XML library:

```
class XmlConverter : ContentConverter {
    override suspend fun convertForSend(
        context: PipelineContext<Any, ApplicationCall>,
        contentType: ContentType,
        value: Any
    ): Any? {
        val xmlMapper = XmlMapper()
        val xml = xmlMapper.writeValueAsString(value)
        return TextContent(xml, contentType.withCharset(context.call.suitableCharset()))
    }

    override suspend fun convertForReceive(context: PipelineContext<ApplicationReceiveRequest,
```

```
ApplicationCall>): Any? {
        val request = context.subject
        val channel = request.value as? ByteReadChannel ?: return null
        val reader = channel.toInputStream().reader(context.call.request.contentCharset() ?: Charsets.UTF_8)
        val type = request.typeInfo
        val xmlMapper = XmlMapper()
        val xml = reader
        val result: Any? = xmlMapper.readValue(xml, type.javaClass)
        return result
    }
}
```

And then set that in the ContentNegotiator

```
    // If nothing is specified in the header the request will use XML
    install(ContentNegotiation) {
        gson {
        }
        register(ContentType.Application.Xml, XmlConverter())
    }
```

By setting up both json and xml we can set the perfered response type in the header: `curl -H "Accept: application/json" "http://localhost:8080/spaceship"` `curl -H "Accept: application/xml" "http://localhost:8080/spaceship"`

Serialization

- We can enable serialization features in both Gson and Jackson to configure the output. There are many serialization features, I will not list them all here, explore thhe class SerializationFeature for details.
- The main difference between GSON and Jackson is which serialization features are enabled by default. Ex:
  - Jackson prints out null values by default.
  - Jackson does not enable support for java.time.* by default, one must add the seperate dependency and register the module

```
    install(ContentNegotiation) {
        // If this line is active the request will use XML, unless the header specifies otherwise
        //register(ContentType.Application.Xml, XmlConverter())
```

```
    jackson {
        registerModule(JavaTimeModule())
        enable(SerializationFeature.INDENT_OUTPUT)
        enable(SerializationFeature.WRITE_SINGLE_ELEM_ARRAYS_UNWRAPPED)
    }
}
```

- Annotations can be used to determine how a single field should behave
  - @JsonProperty - Set configuration to a single field
  - @JsonFormat - To determine the datetime format of a single field
  - @JsonInclude - Set if null values should be included

How Data Is Posted to the Route Endpoint

- The raw body data is collected from the request object

```
val channel: ByteReadChannel = call.receiveChannel()
val text: String = call.receiveText()
// receiveStream() is synchronous and blocks the thread
val inputStream: InputStream = call.receiveStream()
val multipart: MultiPartData = call.receiveMultipart()
```

- Form Parameters can be extracted with the function `call.recieveParameters`
- Cookies sent in the header from the client can be accessed with `val cookies: RequestCookies = call.request.cookies` or can be accessed individually with `val specificCookie: String? = request.cookies["specificCookie"]`
- Multiparts are good for files uploads

```
val multipart = call.receiveMultipart()
multipart.forEachPart { ..}
```

Routes with Path Variables

- In a route, we can get the path variables with call.parameters.get(...)

- The Locations feature maps the variables from a class definition. As of right now it is still an experimental feature, it has been for a while
- Requires implementing "io.ktor:ktor-locations:$ktor_version" and then `install(Locations) {}`
- The path variables are mapped by creating a class and annotation it with `@Location("/myLocation/{mypathvar}")`
- Variables are in { } and it needs to match an argument in the primary constructor
- Then the route needs to be mapped using generics with the class we annotated with Location

```
get<MyLocation> {
call.respondText("${it.mypathvar}")
}
```

Nested Routes

- Nested Routes can be created with nested inner classes

```
@Location("/book/{category}")
data class Book(val category: String) {
@Location("/{author}")
data class Author(val book: Book, val author: String)

@Location("/list")
data class List(val book: Book)
}
```

Routes with Request Parameters

- Request or query parameters are most often used to describe paging and sorting when dealing with many rows of data
- It is the extra parameter after the "?" in a URL `https://localhost:8080/book/list?sortby=author&asc=1`
- Multiple request parameters are separated with the "&" -sign
- Key and value are separated with the "=" sign
- We can use Locations to map request parameters to class fields. This is done by adding extra constructor arguments: `@Location("/listbooks") data class List(val sortBy: String, val asc: Int)` A call to the following might look like: `http://localhost:8080/article/flowers/list?sortBy=author,releasedate&asc=1`
- These arguments can be anything as long as they do not match path variables
- All request parameters are optional.

Working with Headers

Retrieve Headers from Request

- The request can be accessed on the call, when we are setting up the routes with call.request, using the following: `val headerValue: String? = call.request.headers.get("MyHeaderName")`
- Mutiple values for a header key can be accessed with: `val multipleValues: List<String>? = request.headers.getAll("MyHeaderWithMultipleValues")`
- Convienience functions can access standard headers on a request.

Setting Headers on a Response

- The response can be accessed on the call object when we are setting up the routes with call.response `call.response.header("HeaderName", "HeaderValue")`
- HttpHeaders contain the most common HttpHeaders `call.response.header(HttpHeaders.SetCookie, "CookieValue")`
- There is a DefaultHeaders feature available for installation

```
install(DefaultHeaders) {
header("SystemName", "BookStore")
}
```

Error Handling and Authentication

- Error handling can be done by importing StatusPages. The install function has three main configuration options:
  - Exceptions: Create responses based on exception classes
  - Satus: Create responses based on status code value
  - statusFile: Use html file from classpath as response

```
install(StatusPages) {
    exception<MyCustomException> { cause ->
        call.respond(HttpStatusCode.InternalServerError, "Whoops a Cusom Error Occurred")
        throw cause
    }
}
```

- To prevent recursive stack calls when the same exception is thrown multiple times, each call is only caught by one handler
- It is also possible to redirect the client in the exception handler. That might look something like this:

```
install(StatusPages) {
    exception<HttpRedirectException> { e ->
        call.respondRedirect(e.location, permanent = e.permanent)
    }
}

class HttpRedirectException(val location: String, val permanent: Boolean = false): RuntimeException()
```

- We can also return details from the HTTP repsonse:

```
install(StatusPages) {
    status(HttpStatusCode.NotFound) {
        call.respond(TextContent("${it.value} ${it.description}",
        ContentType.Text.Plain.withCharsets(Charsets.UTF_8), it))
    }
}
```

- And the StatusFile return:

```
install(StatusPages) {
    statusFile(HttpStatusCode.NotFound, HttpStatusCode.Unauthorized, FilePattern = "my-custom-error#.html")
}
// The # above will be filled with the error code number
```

Authentication Concepts

- Authentication - proves the person or system is who they claim
- Authorization - the right to perform an action
- Principle - System or person to be authenticated
- Credentials - Username and password or API key that can be used to prove the identity of a principle
- Realm - Used to give more information in an unauthorized response

Supported Authentication Methods

- Basic - Supply base64 encoded username and password in header
- Form - Username and password sent as form data
- HTTP Digest - MD5 encrypt username and password
- JWT and JWK - JSON Web Tokens
- LDAP within basic Authentication
- OAuth 1a and 2.0
- We can check credentials against values in database or against a constant in the validate function. If it's successful we return a UserIdPrinciple.
- It is reccomended to create a table with usernames and hashed passwords

```
    install(Authentication) {
        basic("myAuth1") {
            realm = " My Realm"
            validate {
                if (it.name == "mike" && it.password == "password")
                    UserIdPrincipal(it.name)
                else null
            }
        }
        basic("myAuth2") {
            realm = "MyOtherRealm"
            validate {
                if(it.password == "${it.name}abc123")
                    UserIdPrincipal(it.name)
                else
                    null
            }
        }
    }

routing {
```

```
        authenticate("myAuth1") {
            get("/secret/weather") {
                val principal = call.principal<UserIdPrincipal>()!!
                call.respondText("Hello ${principal.name} it is secretly going to rain today")
            }
        }

        authenticate("myAuth2") {
            get("/secret/color") {
                val principal = call.principal<UserIdPrincipal>()!!
                call.respondText("Hello ${principal.name}, green is going to be popular tomorrow")
            }
        }
}
```

Routing Interceptors - Check Admin Rights

- An incoming request and outgoing response is called an ApplicationCall
- An ApplicationCall is passed through an ApplicationCallPipeline which consists of a number of interceptors, or it may not have any
- Interceptors are invoked one at a time
- An interceptor can choose to let the next interceptor continue with the ApplicationCall
- An interceptor can choose to finish the ApplicationCall and no more interceptors will receive the call
- The ApplicationCallPipeline consists of phases: 1. Setup 2. Monitoring 3. Features 4. Call 5. Fallback
- An interceptor registers to a specific phase
- Code can be run before and after a pipeline

```
// creating a new interceptor to be called after the call
    val mike = PipelinePhase("Mike")
// This would not work if it was insertPhaseAfter, as the route would have already provided a response
    insertPhaseBefore(ApplicationCallPipeline.Call, mike)

    intercept(ApplicationCallPipeline.Setup) {
        log.info("Setup phase")
```

```
    }
    intercept(ApplicationCallPipeline.Call) {
        log.info("Call phase")
    }
    intercept(ApplicationCallPipeline.Features) {
        log.info("Features phase")
    }
    intercept(ApplicationCallPipeline.Monitoring) {
        log.info("Monitoring phase")
    }

    intercept(mike) {
        log.info("Mike Phase${call.request.uri}")
        if (call.request.uri.contains("mike")) {
            log.info("The uri contains mike")
□□□call.respondText("The Endpoint contains mike")
□□□// finish means the remaining interceptors will not be called
            finish()
        }
    }

    routing {

        get("/something/mike/something") {
            call.respondText("Endpoint handled by route.")
        }
    }
```

Ktor Authentication and Authorization

So to retain the client information after login we have two options:

1. Create a server-side session.
2. Store the session in a Json token on the Client side

With server-side sessions, you will either have to store the session identifier in a database, or else keep it in memory and make sure that the client always hits the same server. Both of these have drawbacks. In the case of the database (or other centralised storage), this becomes a bottleneck and a thing to maintain - essentially an extra query to be done with every request.

With an in-memory solution, you limit your horizontal scaling, and sessions will be affected by network issues (clients roaming between Wifi and mobile data, servers rebooting, etc).

Moving the session to the client means that you remove the dependency on a server-side session, but it imposes its own set of challenges.

- Storing the token securely.
- Transporting it securely.
- JWT sessions can sometimes be hard to invalidate.
- Trusting the client's claim.

And I'm not going to go into much detail on the types of encryption or cryptography. I'll be using RSA-256 encryption, which requires a public and private key. Ktor has a great example of JWT authentication with RSA-256 encryption. I've stolen some of their code below, but I decided to use javax.security rather than the ktor JwkProvider.

Basic Auth

Setting up basic authentication in Ktor is pretty straightforward. What gets more complicated is storing the user session in a token or in-memory session.

In the validation function we can choose how we want to verify the username and password. I like to create a database to store the username and a hashed version of the password, and then validate the user in the repository. Something easier would be to create an in-memory hashtable of

users and passwords, all in the docs.

```
    install(Authentication) {


        basic("auth-basic") {
            validate { credentials ->
                val login = LoginRepo(loginConfig)
                val validation = login.validateUser(credentials.name, credentials.password)
                login.close()
                validation
            }
        }
```

Now when ever I make a request to a route protected by "auth-basic" the application will automatically search the request for a basic authentication credentials.

What I do is have my login route return a RS256 signed JWT token after being signed in, so I just wrap the route in an authentication block.

```
    authenticate("auth-basic") {
        get("/LOGIN") {
            log.info("Starting login sequence")
            val publicKey = jwkProvider.get(jwtConfig.pubKeyId).publicKey
            val keySpecPKCS8 = PKCS8EncodedKeySpec(Base64.getDecoder().decode(jwtConfig.privateKey))
            val privateKey = KeyFactory.getInstance("RSA").generatePrivate(keySpecPKCS8)
            val user = this.call.authentication.principal<LoginEntity>()
            val token = JWT.create()
                .withAudience(jwtConfig.audience)
                .withIssuer(jwtConfig.issuer)
                .withClaim("NAME", user?.name)
                .withClaim("ROLE", user?.role)
                .withClaim("EMAIL", user?.email)
                .withExpiresAt(Date(System.currentTimeMillis() + 60000))
                .sign(Algorithm.RSA256(publicKey as RSAPublicKey, privateKey as RSAPrivateKey))
            call.respond(hashMapOf("token" to token))
        }
    }
```

Note there is some the token is signed with an RSA256 algorithm above, more about that in another chapter. Also I'm returning the token in the response body. This is ok, but the production standard is to store the token in an SSL-encrypted cookie. That way, we don't have to send the

token back and forth or store anything in session storage. Also this reduces the risk of cross-site-scripting. More on that in a second.

JSON Web Tokens

"JWTs are an open standard that defines a way for securely transmitting information between parties as a JSON object."

JWTs are used for quick Authorization, not authentication. Store things like a user's role, email, or other nonsensitive information in the payload. NEVER use a token to store a password.

Ktor handles JWTs passed in the Authorization header using the Bearer scheme like so: `Authorization: Bearer {{auth_token}}`

See my Angular page on JWTs for more info.

The following dependcies are required:

```
implementation "io.ktor:ktor-auth:$ktor_version"
implementation "io.ktor:ktor-auth-jwt:$ktor_version"
```

Authorization Flow:

1. Client makes a POST request with credentials:

```
POST http://localhost:8080/login
Content-Type: application/json

{
  "username": "jetbrains",
  "password": "foobar"
}
```

2. If the credentials are validate the server generates a JSON web token and signs it with the specified algorithm
3. Server sends generated JWT to a client

- A client can now make a request to a protext resource with JSON scheme in the header

```
GET http://localhost:8080/hello
Authorization: Bearer {{auth_token}}
```

4. Server recieves the request and performs the following validations:
   - verify the signature of a JSON object
   - perform additional validations on the JWT payload
5. After validation server responds with contents of protected resource

Installation and Configuration

Add the JWT function to the install Authentciation module, you can define your private key, issuer, audience and realm in application.conf,

```
    val privateKeyString = environment.config.property("jwt.privateKey").getString()
    val issuer = environment.config.property("jwt.issuer").getString()
    val audience = environment.config.property("jwt.audience").getString()
    val myRealm = environment.config.property("jwt.realm").getString()
    val jwkProvider = JwkProviderBuilder(issuer)
        .cached(10, 24, TimeUnit.HOURS)
        .rateLimited(10, 1, TimeUnit.MINUTES)
        .build()
    install(Authentication) {
        jwt("auth-jwt") {
            realm = myRealm
            verifier(jwkProvider, issuer) {
                acceptLeeway(3)
            }
            validate { credential ->
                if (credential.payload.getClaim("username").asString() != "") {
                    JWTPrincipal(credential.payload)
                } else {
                    null
                }
            }
        }
    }
```

Alternatively, there's no downside to generating a new private key every run. In the official Ktor example they create the JWT token directly in the routing function. I perfer to created a seperate class for token generation and validation that looked something like this:

```
class JWTService(private val jwtConfig: JwtConfig, private val jwkProvider: JwkProvider) {

    private val privateKey: PrivateKey

    init {
        val keySpecPKCS8 = PKCS8EncodedKeySpec(Base64.getDecoder().decode(jwtConfig.privateKey))
        privateKey = KeyFactory.getInstance("RSA").generatePrivate(keySpecPKCS8)
    }

    fun generateToken(user: LoginEntity): String = JWT.create()
                .withAudience(jwtConfig.audience)
                .withIssuer(jwtConfig.issuer)
                .withClaim("NAME", user.name)
                .withClaim("ROLE", user.role)
                .withClaim("EMAIL", user.email)
                .withExpiresAt(Date(System.currentTimeMillis() + 60000))
                .sign(Algorithm.RSA256(jwkProvider.get(jwtConfig.pubKeyId).publicKey as RSAPublicKey, privateKey
as RSAPrivateKey))

    fun verifyToken(token: String?): Boolean {
        if (token == null) {
            println("No token found in memory")
            return false
        }
        val payloadJson = validatedToken(token) ?: return false
        return (payloadJson["ROLE"] == jwtConfig.realm)
    }

    fun getLoginEntity(token: String): LoginEntity? {
        val payloadJson = validatedToken(token) ?: return null
        val name = payloadJson["NAME"].toString()
        val email = payloadJson["EMAIL"].toString()
        val role = payloadJson["ROLE"].toString()
        return LoginEntity(name, email, role)
    }

    private fun validatedToken(validateToken: String): JSONObject? {
        try {
            val encodedPayload = JWT.require(
                Algorithm.RSA256(
```

```
                jwkProvider.get(jwtConfig.pubKeyId).publicKey as RSAPublicKey,
                privateKey as RSAPrivateKey
            )
        )
            .build()
            .verify(validateToken)
            .payload
        val payload = String(Base64.getDecoder().decode(encodedPayload))
        val parser = JSONParser()
        return parser.parse(payload) as JSONObject
    } catch (jwtException: JWTVerificationException) {
        println("Failed to verify JWT: " + jwtException.message)
        return null
    } catch (exception: Exception) {
        println("An error occured: " + exception.message)
        return null
    }
  }
}
```

Now that we have that ready I change the login function to look like so:

```
  authenticate("auth-basic") {
    get(CommonRoutes.LOGIN) {
      log.info("Starting login sequence")
      val user = this.call.authentication.principal<LoginEntity>()!!
      val token = jwtService.generateToken(user)
      // "secure=true" will only work when a valid HTTPS certificate is present!
      val cookie = Cookie("token", token, httpOnly = true, secure = true)
      call.response.cookies.append(cookie)
      call.respond(Response(status = "ok"))
    }
  }

  authenticate("auth-jwt"){
    get("getRole") {
      val entity = jwtService.getLoginEntity(call.request.cookies["token"]!!) ?: LoginEntity("", "", "")
      call.respond(entity)
    }
    get("logout") {
```

```
            call.response.cookies.appendExpired("token")
            call.respond(Response(status = "ok"))
        }
    }
```

As well as implmented cookies, I added a "getRole" route that checks for cookies and returns a user entity if the user is logged in. Using this we can prevent the user from having to login everytime the page is refreshed.

Then we can protect a route with the following syntax:

```
routing {
    authenticate("jwt-auth") {
        get("/") {
            call.respondText("Hello, ${call.principal<UserIdPrincipal>()?.name}!")
        }
    }
}
```

Generating a Self-Signed Certificate

Above I mentioned the best security practice is to SSL-encyrpt our cookies over https. This requires a SSL-certificate. This may be a bit difficult to set up depending on how you are running Ktor. I'm using a docker container with an nginx reverse proxy and a cloudflare domain. I already have an SSL certifate for my domain, but the problem was the my docker was running on http which was causing issues when using `secure` cookies.

In many situtations it is bad practice to use self-signed certificates, but in the case of an LAN address that only we have access to I think it will be okay. TODO: More research on if a reverse proxy exposes the keys on the host.

Ktor has a great library for generating self-signed certificates within a embedded server, but it's labeled as only for testing purposes. Instead I'll generate the ssl certificate with *Let's Encrypt* and store that in a keystore generated manually with `keytool` and add the configuration to `application.conf`.

In a nutshell, steps are as follows:

1. Pulling the Let's Encrypt client (certbot).
2. Generating a certificate for your domain (e.g. example.com)

`./certbot-auto certonly -a standalone -d example.com -d www.example.com`

Things are generated in /etc/letsencrypt/live/example.com. Industry standard is PKCS12 formatted file. Convert the keys to a PKCS12 keystore using OpenSSL as follows:

Open `/etc/letsencrypt/live/example.com` directory.

`openssl pkcs12 -export -in fullchain.pem -inkey privkey.pem -out keystore.p12 -name tomcat -CAfile chain.pem -caname root`

The file keystore.p12 with PKCS12 is now generated in `/etc/letsencrypt/live/example.com.`

It's time to configure your Spring Boot application. Open the application.properties file and put following properties there:

server.port=8443 security.require-ssl=true server.ssl.key-store=/etc/letsencrypt/live/example.com/keystore.p12 server.ssl.key-store-password= server.ssl.keyStoreType=PKCS12 server.ssl.keyAlias=tomcat

Read my blog post for further details and remarks.

Signing JSON Tokens with RSA

RS256 vs HS256

When signing a JSON Web Token (JWT) from the server, two algorithms are supported for signing JSON Web Tokens (JWTs): RS256 and HS256. HS256 is the default for clients and RS256 is the default for APIs. When building applications, it is important to understand the differences between these two algorithms. To begin, HS256 generates a symmetric MAC and RS256 generates an asymmetric signature. Simply put HS256 must share a secret with any client or API that wants to verify the JWT. Like any other symmetric algorithm, the same secret is used for both signing and verifying the JWT. This means there is no way to fully guarantee Auth0 generated the JWT as any client or API with the secret could generate a validly signed JWT. On the other hand, RS256 generates an asymmetric signature, which means a private key must be used to sign the JWT and a different public key must be used to verify the signature. Unlike symmetric algorithms, using RS256 offers assurances that our server is the signer of a JWT since only one party has the private key.

Verifying RSA256

At the most basic level, the JWKS is a set of keys containing the public keys that should be used to verify any JWT issued by the authorization server. We set this up as a static endpoint on the backend server, something like *https://your-domain/api/.well-known/jwks,json* To create that I added a folder called certs/ to the base project directory and then added the following in a file called 'jwks.json':

```
{
"keys": [
  {
    "alg": "RS256",
    "kty": "RSA",
    "use": "sig",
    "x5c": [
```

```
"MIIC+DCCAeCgAwIBAgIJBIGjYW6hFpn2MA0GCSqGSIb3DQEBBQUAMCMxITAfBgNVBAMTGGN1c3RvbWVyLWRlbW9zLmF1dGgwLmNvbTAeFw0xNjExMjIyMjIyMDVaFw0zMDA4MDEyMjIyMDVaMCMxITAfBgNVBAMTGGN1c3RvbWVyLWRlbW9zLmF1dGgwLmNvbTCCASIwDQYJKoZIhvcNAQEBBQADggEPADCCAQoCggEBAMnjZc5bm/eGlHq09N9HKHahM7Y31P0ul+A2wwP4lSpIwFrWHzxw88/7Dwk9QMc+orGXX95R6av4GF+Es/nG3uK45ooMVMa/hYCh0Mtx3gnSuoTavQEkLzCvSwTqVwzZ+5noukWVqJuMKNwjL77GNcPLY7Xy2/skMCT5bR8UoWaufooQvYq6SyPcRAU4BtdquZRiBT4U5f+4pwNTxSvey7ki50yc1tG49Per/0zA4O6Tlpv8x7Red6m1bCNHt7+Z5nSl3RX/QYyAEUX1a28VcYmR41Osy+o2OUCXYdUAphDaHo4/8rbKTJhlu8jEcc1KoMXAKjgaVZtG/v5ltx6AXY0CAwEAAaMvMC0wDAYDVR0TBAUwAwEB/zAdBgNVHQ4EFgQUQxFG602h1cG+pnyvJoy9pGJJoCswDQYJKoZIhvcNAQEFBQADggEBAGvtCbzGNBUJPLICth3mLsX0Z4z8T8iu4tyoiuAshP/Ry/ZBnFnXmhD8vwgMZ2lTgUWwlrvlgN+fAtYKnwFO2G3BOCFw96Nm8So9sjTda9CCZ3dhoH57F/hVMBB0K6xhklAc0b5ZxUpCIN92v/w+xZoz1XQBHe8ZbRHaP1HpRM4M7DJk2G5cgUCyu3UBvYS41sHvzrxQ3z7vIePRA4WF4bEkfX12gvny0RsPkrbVMXX1Rj9t6V7QXrbPYBAO+43JvDGYawxYVvLhz+BJ45x50GFQmHszfY3BR9TPK8xmMmQwtIvLu1PMttNCs7niCYkSiUv2sc2mlq1i3IashGkkgmo="
    ],
    "n": "yeNlzlub94YgerT030codqEztjfU_S6X4DbDA_iVKkjAWtYfPHDzz_sPCT1Axz6isZdf3lHpq_gYX4Sz-cbe4rjmigxUxr-FgKHQy3HeCdK6hNq9ASQvMK9LBOpXDNn7mei6RZWom4wo3CMvvsY1w8tjtfLb-yQwJPltHxShZq5-ihC9irpLI9xEBTgG12q5lGIFPhTl_7inA1PFK97LuSLnTJzW0bj096v_TMDg7pOWm_zHtF53qbVsI0e3v5nmdKXdFf9BjIARRfVrbxVxiZHjU6zL6jY5QJdh1QCmENoejj_ytspMmGW7yMRxzUqgxcAqOBpVm0b-_mW3HoBdjQ",
    "e": "AQAB",
    "kid": "NjVBRjY5MDlCMUIwNzU4RTA2QzZFMDQ4QzQ2MDAyQjVDNjk1RTM2Qg",
    "x5t": "NjVBRjY5MDlCMUIwNzU4RTA2QzZFMDQ4QzQ2MDAyQjVDNjk1RTM2Qg"
  }
]}
```


- alg: is the algorithm for the key
- kty: is the key type
- use: is how the key was meant to be used. For the example above, sig represents signature verification.
- x5c: is the x509 certificate chain
- kid: is the unique identifier for the key
- x5t: is the thumbprint of the x.509 cert (SHA-1 thumbprint)
- parameter n: Base64 URL encoded string representing the modulus of the RSA Key.
- parameter e: Base64 URL encoded string representing the public exponent of the RSA Key.
- parameter d: Base64 URL encoded string representing the private exponent of the RSA Key.
- parameter p: Base64 URL encoded string representing the secret prime factor of the RSA Key.
- parameter q: Base64 URL encoded string representing the secret prime factor of the RSA Key.
- parameter dp: Base64 URL encoded string representing the first factor CRT exponent of the RSA Key. d mod (p-1)

- parameter dq: Base64 URL encoded string representing the second factor CRT exponentof the RSA Key. d mod (q-1)
- parameter qi: Base64 URL encoded string representing the first CRT coefficient of the RSA Key. q^-1 mod p

And the I added the following routes:

```
static(".well-known") {
    staticRootFolder = File("certs")
    file("jwks.json")
}
```

I recommend the following links:

- Generate JWKS Token
- Change from JWKS and PEM format

Steps for validating the JWT Server-side:

1. Retrieve the JWKS and filter for potential signature verification keys.
2. Extract the JWT from the request's authorization header.
3. Decode the JWT and grab the kid property from the header.
4. Find the signature verification key in the filtered JWKS with a matching kid property.
5. Using the x5c property build a certificate which will be used to verify the JWT signature.
6. Ensure the JWT contains the expected audience, issuer, expiration, etc.