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.")
        }
    }
```

Revision #1
Created 17 April 2022 00:22:01 by Elkip
Updated 17 April 2022 01:02:09 by Elkip