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}")
            }
        }
    }
}
```

Revision #1
Created 17 April 2022 00:03:34 by Elkip
Updated 17 April 2022 01:02:09 by Elkip