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

---

Revision #1
Created 16 April 2022 23:57:49 by Elkip
Updated 17 April 2022 01:02:09 by Elkip