ktor App Servers

ktor

- ktor is a app server framework from Jetbrains (the IntelliJ people) written in Kotlin
- Concurrency is implemented by having most callbacks be suspend functions (coroutines) that can be yielded and resumed
- “host implementations” (ie the slow IO calls) use nonblocking APIs

ktor design

- The framework is a “pipeline” that follows requests through the system
- “Interceptors” can be hooked in at various points to perform tasks/modify the request/response along the way
- There is a default pipeline that we'll use which gives us basic request processing phases, and we'll add a few plugins which get run throughout the pipeline

Routing

- This is probably the plugin we'll deal with the most
- It allows us to attach a new handler for a given URL path
- Here's the basic syntax:

```
routing {
    get("/my/path") { handlerCode() }
    post("/some/other/path") { handlerCode() }
}
```

- There's a bunch of ways to set it up, but basically we want to call the above code when we're initializing the application
- We can have many separate `routing` blocks, so we can organize our code basically however we want

More complex routing

- We can have parameterized routes like `/users/{id}/email`
- To access the id in our handler we can use `call.parameters["id"]` which returns a `String`?
- You can also do nested routing like:

```
routing {
    route("/student"){
        route("/grades"){
            get { handler for "/student/grades" }
        }
        route("/address"){
            get { handler for "/student/address" }
            post { handler for POST to "/student/address" }
}}}
```

Sending responses

- There's a family of `call.respond*` methods which let you respond in various formats (string, raw bytes, automatically serialized format like json, etc)
- `call.response` lets you manipulate the status code, response headers, etc

Serialization

- The `ContentNegotiation` plugin and its friends let you automatically serialize/deserialize objects from requests/to responses
- You need to annotate classes as `@Serializable` to opt in to this

```
@Serializable
data class Customer(val id: Int, val firstName: String, val lastName: String) post("/customer") {
    val customer = call.receive<customer>()
    customerStorage.add(customer)
    call.respondText("Customer stored correctly", status = HttpStatusCode.Created)
}
```

- This would automically deserialize a JSON formatted customer in a POST request

“Resources” for type safety

- The `Resources` plugin adds type safety to routing (making sure parameters are the right types)
- First you define an annotated `Resource` class representing a top-level path (like "/students”)
- It's nested members are nested paths (a “grades” member would correspond to "/students/grades”)
- There are overloads of the routing functions (`get`, `post`, etc) which take these types as template parameters, which will make sure types check before calling your handlers

Database stuff: exposed

- Exposed is a jetbrains kotlin library sort of like Room
- It works with any JDBC comptible DB (basically any RMDBs)

DB Entities

- We define out DB schemas by defining kotlin singleton `objects` that inherit from `Table`
- Fields are defined using a SQL-like syntax

```
object SomeTable: Table(){
    val id = integer("id").auto_increment()
    val book = reference("book_id", Book.id)
    override val primaryKey = PrimaryKey(arrayOf(id, book))
}
```

Queries

- There are 2 different APIs but we'll look at a sort of LINQ based API
- It's basically a direct translation of SQL syntax, embedded in Kotlin, implemented as a “domain specific language”
- DSLs let you write your own languages that fit within a host language
- Since it's compiled by Kotlin, our queries will be type checked even though they look like SQL

Example

“Book Liking App”