Come to the backend side
We have Kotlin!

Ktor for backend development

(and cookies)

Julien Salvi

FOSDEM 21

Julien Salvi

Senior Android Engineer @ aircall

10 years into Android!
PAUG, Punk and IPAs!

@JulienSalvi

Table of Contents

01 **Introduction**
A Ktor overview

02 **Setup Ktor project**
First steps with Ktor

03 **Build your web service**
Ktor features, Postgres, Exposed…

04 **Share code thanks to Swagger**
A source of truth for your API

01

Introduction

A Ktor overview

A Ktor overview

- Developed and maintained by **JetBrains**
- Stable release in **2018**
- Latest release is **1.5.0**
- **Asynchronous framework** for building web services, web apps and more written in Kotlin .

A Ktor overview

- Ktor is **lightweight**. Use the features you want with easy configuration
- Ktor is **extensible**. Configurable pipeline where you can extend what your need
- Ktor is **multiplatform**. Use it on mobile, for backends or web apps
- Ktor is **asynchronous**. Thanks to Kotlin coroutines we can build high scalable microservices

Let's build a web service

Unleash the power of Ktor

Let's build a tiny web service

- Setup default **routes**
- Use Ktor Features
- Setup a **Postgres** database that runs with **docker-compose**
- Use **Exposed** for dealing with the database, **Valiktor** for data validation
- Use **Swagger** as a single source of truth

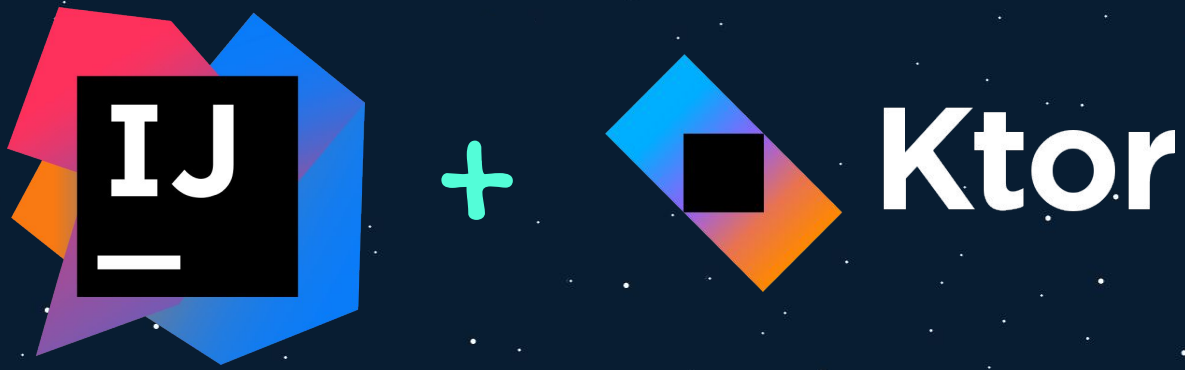
Star Wars Cooking API

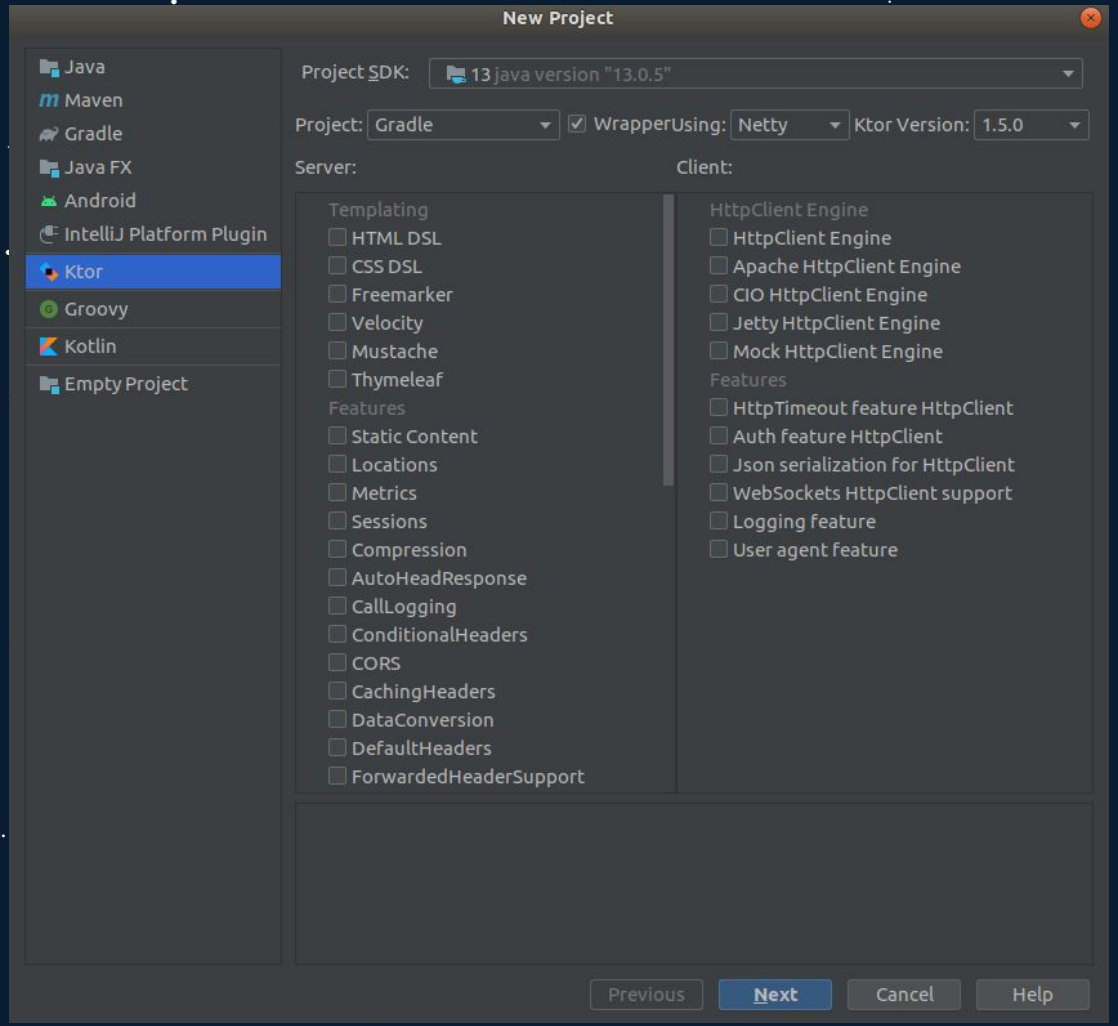
(because it's cool!)

02

Setup a Ktor project

First steps with Ktor


IJ
+
Ktor


New Project
Java
Maven
Gradle
Java FX
Android
IntelliJ Platform Plugin
Ktor
Groovy
Kotlin
Empty Project
Project SDK:
13 java version "13.0.5"
Project:
Gradle
WrapperUsing:
Netty
Ktor Version:
1.5.0
Server:
Templating
HTML DSL
CSS DSL
Freemarker
Velocity
Mustache
Thymeleaf
Features
Static Content
Locations
Metrics
Sessions
Compression
AutoHeadResponse
CallLogging
ConditionalHeaders
CORS
CachingHeaders
DataConversion
DefaultHeaders
ForwardedHeaderSupport
Client:
HttpClient Engine
HttpClient Engine
Apache HttpClient Engine
CIO HttpClient Engine
Jetty HttpClient Engine
Mock HttpClient Engine
Features
HttpTimeout feature HttpClient
Auth feature HttpClient
Json serialization for HttpClient
WebSockets HttpClient support
Logging feature
User agent feature
Previous
Next
Cancel
Help

Minimal Setup

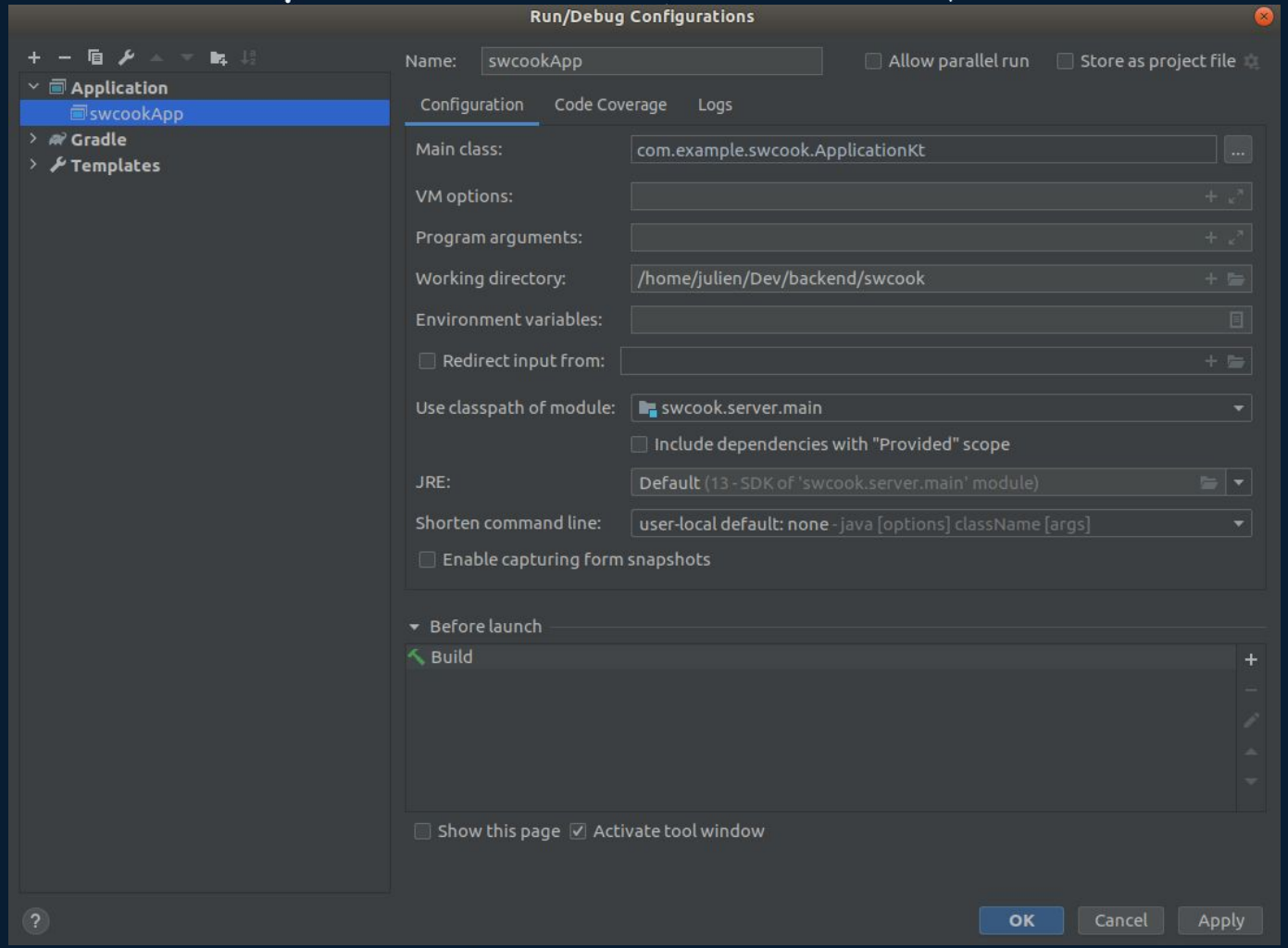
```
// In Application.kt your main class

fun main(args: Array<String>): Unit = io.ktor.server.netty.EngineMain.main(args)

@KtorExperimentalLocationsAPI
@KtorExperimentalAPI
@Suppress("unused") // Referenced in application.conf
@kotlin.jvm.JvmOverloads
fun Application.module(testing: Boolean = false) {

    routing {
        get("/") {
            call.respondText("Server is running 🏃")
        }
    }
}
```

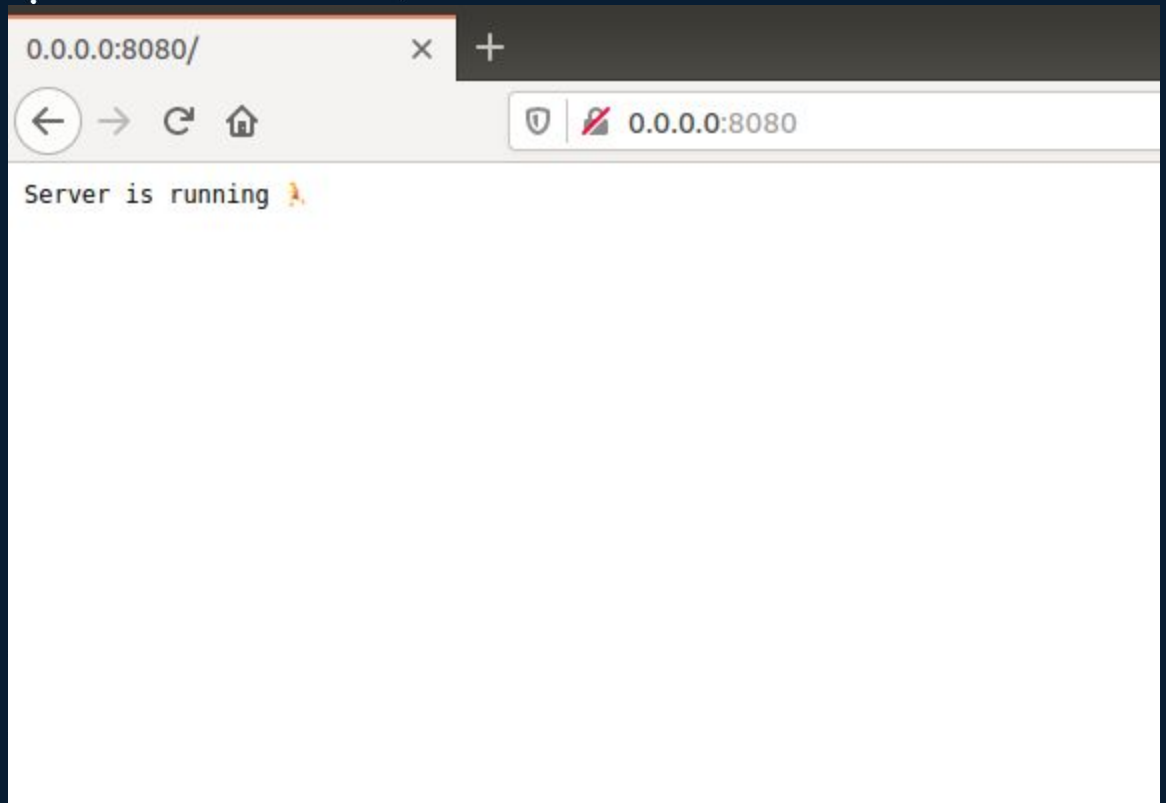
Minimal Setup

```
// In Application.kt your main class

fun main(args: Array<String>): Unit = io.ktor.server.netty.EngineMain.main(args)

@KtorExperimentalLocationsAPI
@KtorExperimentalAPI
@Suppress("unused") // Referenced in application.conf
@kotlin.jvm.JvmOverloads
fun Application.module(testing: Boolean = false) {

    routing {
        get("/") {
            call.respondText("Server is running 🏃")
        }
    }
}
```

Minimal Setup

```
// In application.conf in the resources folder

ktor {
    environment = dev
    environment = ${?KTOR_ENVIRONMENT}

    deployment {
        port = 8080
        port = ${?KTOR_PORT} # Override config via environment variable
    }

    application {
        modules = [ com.example.swcook.ApplicationKt.module ]
    }
}
```

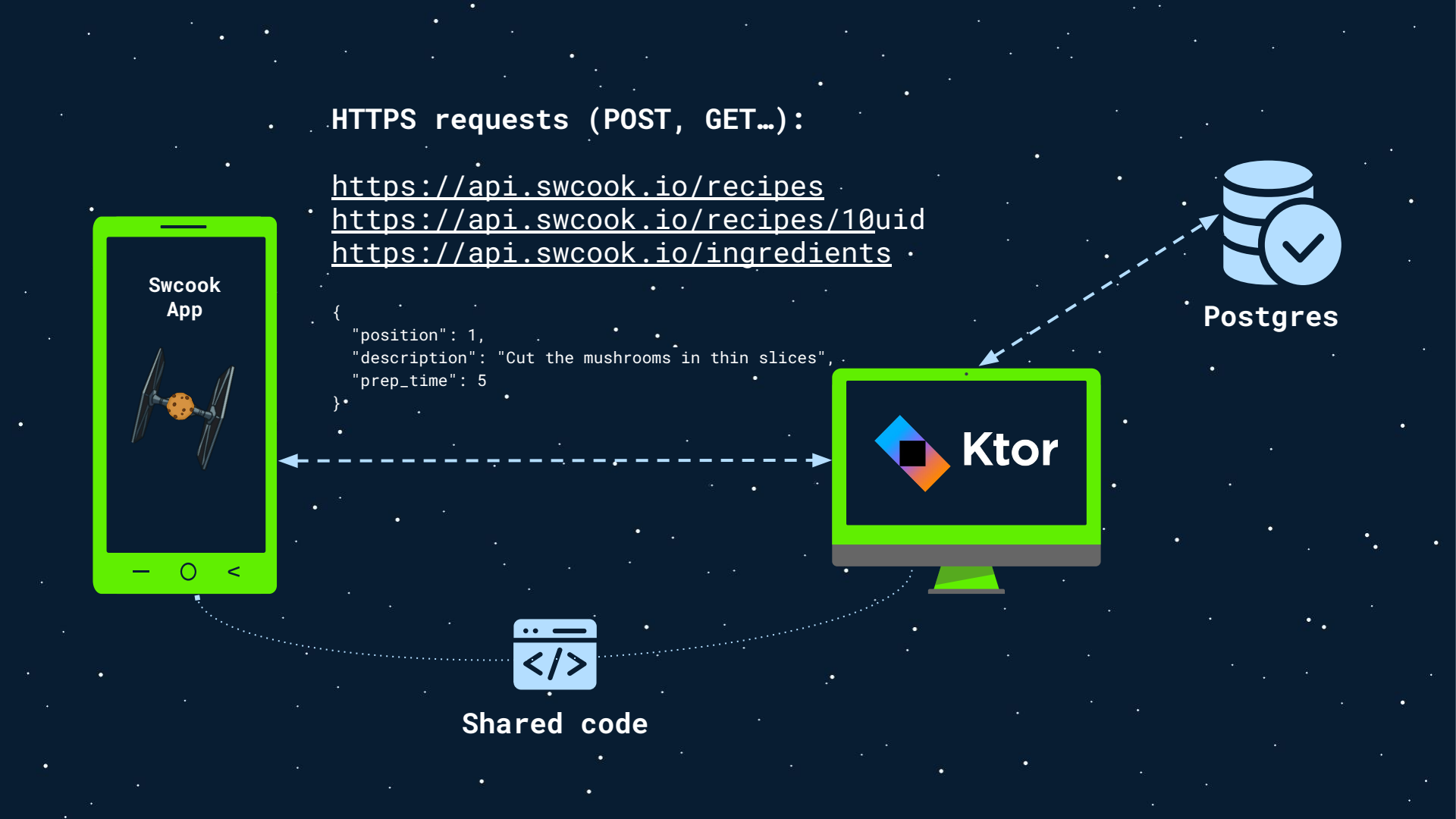

Run/Debug Configurations
Application
swcookApp
Gradle
Templates
Name: swcookApp
Allow parallel run
Store as project file
Configuration
Code Coverage
Logs
Main class: com.example.swcook.ApplicationKt
VM options:
Program arguments:
Working directory: /home/julien/Dev/backend/swcook
Environment variables:
Redirect input from:
Use classpath of module: swcook.server.main
Include dependencies with "Provided" scope
JRE: Default (13 - SDK of 'swcook.server.main' module)
Shorten command line: user-local default: none - java [options] className [args]
Enable capturing form snapshots
Before launch
Build
Show this page
Activate tool window
OK
Cancel
Apply


0.0.0.0:8080/
0.0.0.0:8080
Server is running

03

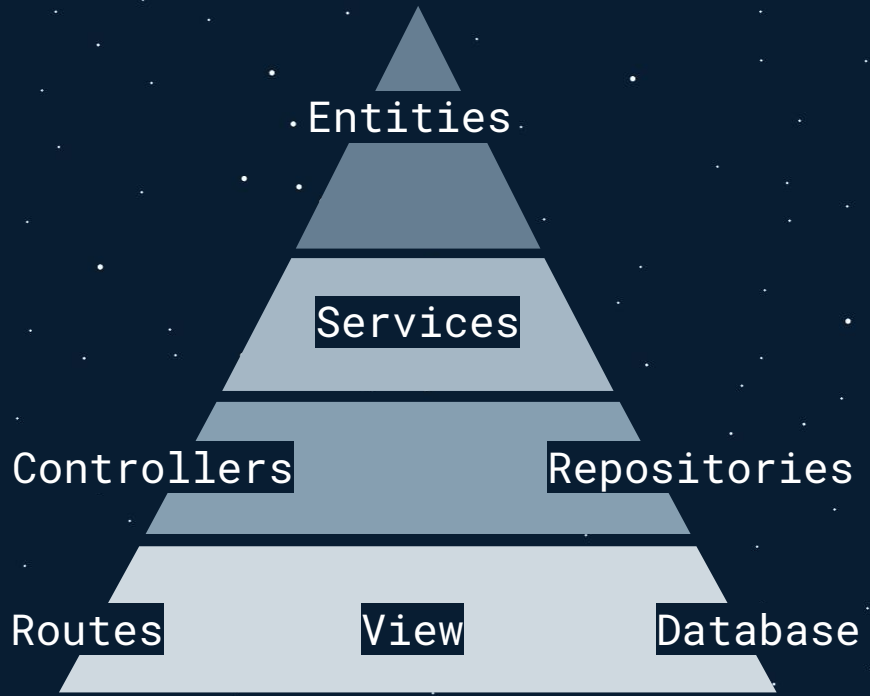
Build your web service

Ktor features, Postegres, Exposed…


HTTPS requests (POST, GET…):
https://api.swcook.io/recipes
https://api.swcook.io/recipes/10uid
https://api.swcook.io/ingredients
{
"position": 1,
"description": "Cut the mushrooms in thin slices",
"prep_time": 5
}
Swcook App
Ktor
Postgres
Shared code

Architecture

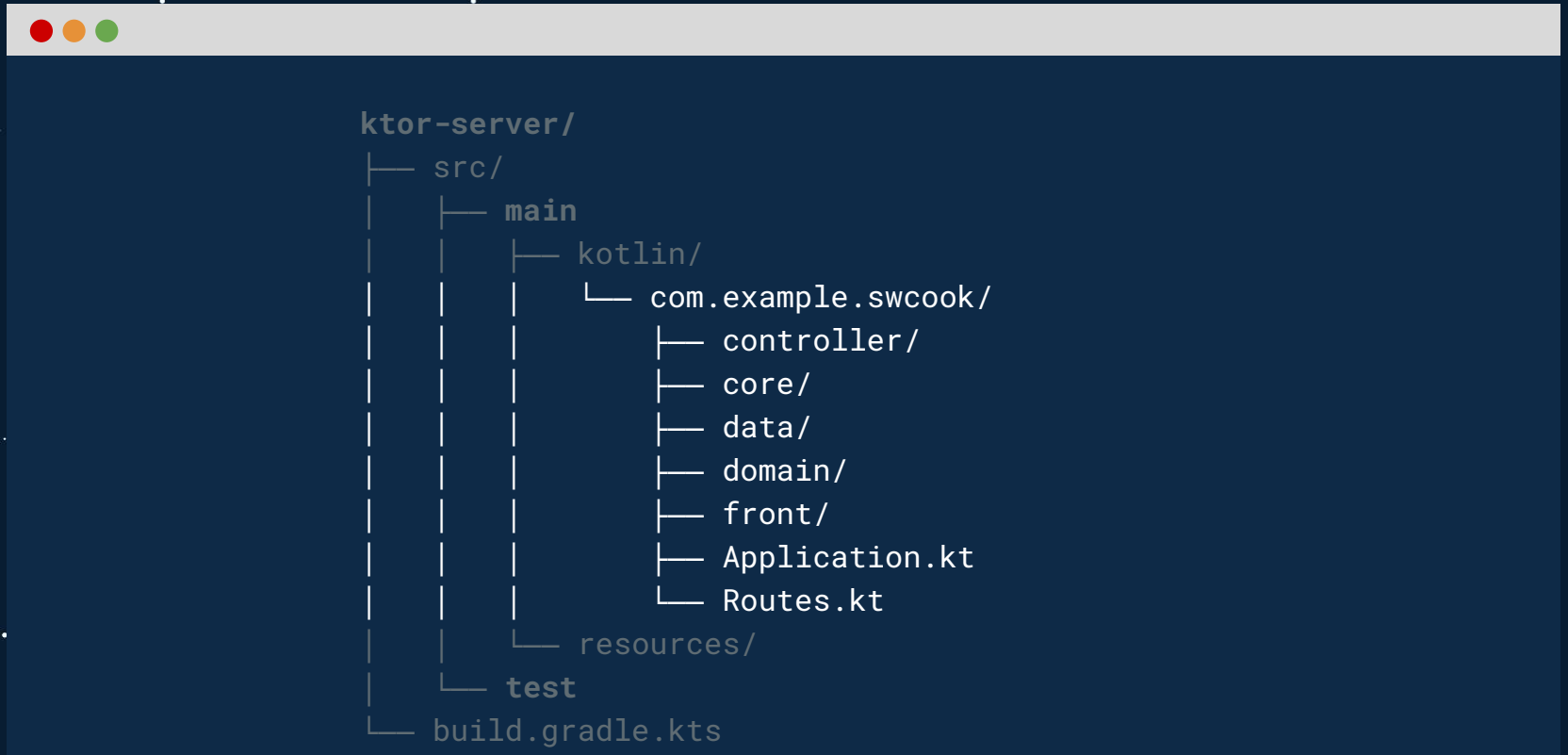
MVC
+
Clean Archi-ish



Project Structure

```
ktor-server/
├── src/
│   ├── main
│   │   ├── kotlin/
│   │   │   └── com.example.swcook/
│   │   │       ├── controller/
│   │   │       ├── core/
│   │   │       ├── data/
│   │   │       ├── domain/
│   │   │       ├── front/
│   │   │       ├── Application.kt
│   │   │       └── Routes.kt
│   │   └── resources/
│   └── test
└── build.gradle.kts
```

Project Structure



Features and libraries

Features & libraries	Purpose
Locations	Routing
ContentNegotiation	Negotiating media types and serialization
StatusPage	Handle exceptions
DataConversion	Type conversion
Logback	Logging
Koin	Dependency injection
Exposed	ORM framework to deal with the Postgres database
Valiktor	Data validation
Flyway	Database versioning
Moshi	JSON library
HikariCP	JDBC connection pool

Locations

- Create routes that are **type-safe**
- Easy to configuration
- Support **basic types** (Int, String, Boolean…) by default
- Typed methods for defining route handlers: ***get***, ***post***, ***patch***, ***delete…***

Feature: **Locations**

```
implementation("io.ktor:ktor-locations:1.5.0")
```

Routing with Locations

```
object Routes {

    @Location("/recipes")
    class Recipes {
        @Location("/{uid}")
        data class ByUid(val recipes: Recipes, val uid: UUID) {
            @Location("/ingredients")
            data class Ingredients(val app: ByUid) {
                @Location("/{ingredientUid}")
                data class ByUid(val ingredients: Ingredients, val ingredientUid: UUID)
            }
        }
    }

    @Location("/recipes/{uid}/ingredients/{ingredientUid}")
    class RecipeIngredientDetails(val uid: UUID, val ingredientUid: UUID)
}
```

Routing with Locations

```
fun Application.module(testing: Boolean = false) {
    install(Locations)

    routing {
        recipes()
    }
}

// RecipeController.kt
fun Route.recipes() {

    get<Routes.Recipes> { }
    post<Routes.Recipes> { }
    patch<Routes.Recipes.ByUid> { route -> }
    delete<Routes.Recipes.ByUid> { route -> }
}
```

Routing with Locations

```
// GET request with parameters
get<Routes.Recipes> {
    val page = call.parameters["page"].toInt()
    val size = call.parameters["size"].toInt()
    ...
    call.respond(HttpStatusCode.OK, response)
}

// POST request with body
post<Routes.Recipes> {
    val request = call.receive<PostRecipeRequest>()
}

// POST request with multipart data (upload files for example)
post<Routes.Recipes.ByUid> { route ->
    val uid = route.uid
    val multipart = call.receiveMultipart()
}
```

Data Conversion

- Serialize and deserialize a list of values (Date, UUID…)
- Can be combined with **Locations** to support custom types
- Simple configuration
- Part of the core features

Feature:
DataConversion

DataConversion

```
fun Application.module(testing: Boolean = false) {
    install(DataConversion) {
        converters()
    }
}
```

DataConversion

```
fun DataConversion.Configuration.converters() {

    convert<UUID> {
        decode { values, _ ->
            values.singleOrNull()?.let { value -> UUID.fromString(value) }
        }

        encode { value ->
            when (value) {
                null -> emptyList()
                is UUID -> listOf(value.toString())
                else -> throw DataConversionException("Cannot convert $value as UUID")
            }
        }
    }
}
```

Moshi

- **JSON library** for made by Square
- **Deserialize** data received from requests and **serialize** responses
- Lightweight, no HTML escaping, custom type adapters
- Used by **Swagger** gradle plugin 👀

square/**moshi**

```
implementation("com.squareup.moshi:moshi:1.11.0")
```

Valiktor

- **Data validation** for requests
- **Type-safe DSL** to validate objects
- Supports **suspending functions** natively
- Multiple **extensions** (Spring, Joda…)

Github: **Valiktor**

```
implementation("org.valiktor:valiktor-core:0.12.0")
```

Valiktor

```
"""
{
  "position": 1,
  "description": "Cut the mushrooms in thin slices",
  "prep_time": 5
}
"""

@JsonClass(generateAdapter = true)
data class PostStepRequest(
    @Json(name = "position") @field:Json(name = "position") var position: Long,
    @Json(name = "description") @field:Json(name = "description") var description: String,
    @Json(name = "prep_time") @field:Json(name = "prep_time") var prepTime: Long
)
```

Valiktor

```
@Throws(ConstraintViolationException::class)
fun PostStepRequest.validate() {
    validate(this) {
        validate(PostStepRequest::position).isNotNull()
        validate(PostStepRequest::position).isGreaterThanOrEqualTo(0)

        validate(PostStepRequest::description).isNotNull()
        validate(PostStepRequest::description).isNotBlank()

        validate(PostStepRequest::prepTime).isNotNull()
        validate(PostStepRequest::prepTime).isGreaterThanOrEqualTo(0)
    }
}
```

Valiktor

```
post<Routes.Recipes> {
    val request = call.receive<PostRecipeRequest>()
    request.validate()
    val created = recipeService.add(request.recipe.toEntity())
    if (created != null) {
        val response = PostRecipeResponse(recipe = created.renderer())
        call.respond(HttpStatusCode.Created, response)
    } else {
        call.respond(HttpStatusCode.Conflict)
    }
}
```

Error Handling: Status Page

- Properly handle **errors** and **exceptions** in the application
- Three configurations: **exceptions**, **status** and **statusFile**
- Simple configuration
- Part of the core features

Feature: **StatusPage**

StatusPage

```
fun Application.module(testing: Boolean = false) {
    install(StatusPage) {
        exceptions()
    }
}
```

StatusPage

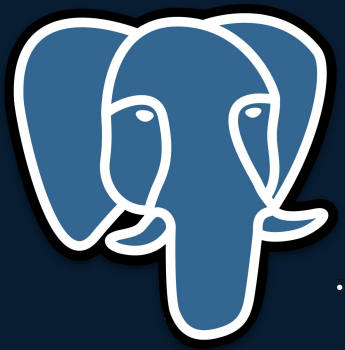
```
fun StatusPages.Configuration.exceptions() {

    exception<ConstraintViolationException> { exception ->
        // Basic ConstraintViolationException handler
        val violations = exception.constraintViolations.map { violation ->
            "${violation.property}:${violation.constraint.name}"
        }
        call.respondText(status = HttpStatusCode.UnprocessableEntity) {
            violations.toString()
        }
    }

    exception<Throwable> { exception ->
        application.log.error("Unhandled exception", exception)
        call.respond(HttpStatusCode.InternalServerError)
    }
}
```

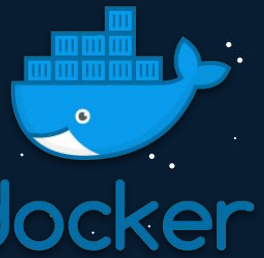
Setup Postgres Database

From Docker with love 🐳

PostgreSQL database with Docker Compose

- Persistent data with **Postgres** database
- Run PostgreSQL database and pgadmin on a **Docker** container thanks to **Docker Compose** tool
- **Compose** is a tool for running multi-container Docker apps
- Use a **YAML file** to configure services





Docker Compose

```yaml
# For development purpose only
version: "3.7"
services:
  postgres:
    image: postgres:12-alpine
    restart: always
    environment:
      POSTGRES_DB: postgres
      POSTGRES_USER: admin
      POSTGRES_PASSWORD: admin
      PGDATA: /var/lib/postgresql/data
    volumes:
      - postgres-data:/var/lib/postgresql/data
    ports:
      - "5432:5432"

volumes:
  postgres-data:
```

Docker Compose

```
# Run/stop docker compose
docker-compose up / down

# Check volumes
docker volume ls

# Remove volume
docker volume rm <volume>
```

Exposed

- **ORM** framework for Kotlin by JetBrains
- Supports **PostgreSQL**, MySQL, H2, MariaDB, SQLite… + **DataSource** JDBC connection with HikariCP
- **DSL** or **DAO** API for querying the database
- Supports basic and advanced **CRUD operations**

Github: **Exposed**

Gradle Dependencies

```
implementation("org.postgresql:postgresql:42.2.16")

implementation("com.zaxxer:HikariCP:3.4.5")

implementation("org.flywaydb:flyway-core:6.5.5")

implementation("org.jetbrains.exposed:exposed-core:0.28.1")
implementation("org.jetbrains.exposed:exposed-dao:0.28.1")
implementation("org.jetbrains.exposed:exposed-jdbc:0.28.1")
implementation("org.jetbrains.exposed:exposed-java-time:0.28.1")
```

Setup database connection

```
# In your application.conf
database {
    connection {
        jdbc = "jdbc:postgresql://localhost:5432/postgres"
        jdbc = ${?DATABASE_JDBC}

        user = admin
        user = ${?DATABASE_USER}

        password = admin
        password = ${?DATABASE_PASSWORD}
    }
}
```

Setup database connection

```
private val appConfig: ApplicationConfig = HoconApplicationConfig(ConfigFactory.load())
lateinit var dataSource: DataSource

val dbConfig = appConfig.config("ktor.database")
val config = HikariConfig()
config.jdbcUrl = dbConfig.property("connection.jdbc").getString()
config.username = dbConfig.property("connection.user").getString()
config.password = dbConfig.property("connection.password").getString()

config.isAutoCommit = false
config.maximumPoolSize = 3
config.transactionIsolation = "TRANSACTION_REPEATABLE_READ"
config.validate()

dataSource = HikariDataSource(config)
//Handle database migrations and versionning
Database.connect(dataSource)
```

Setup Tables and DAO

```
object RecipeTable : ExtendedUUIDTable(name = "recipe") {
    val title: Column<String> = text(name = "title")
    val description: Column<String> = text(name = "description")
    val cookingTime: Column<Int> = integer(name = "cooking_time").default(0)
}

class Recipe(id: EntityID<UUID>) : ExtendedUUIDEntity(id, RecipeTable) {
    companion object : ExtendedUUIDEntityClass<Recipe>(RecipeTable)

    var title by RecipeTable.title
    var description by RecipeTable.description
    var cookingTime by RecipeTable.cookingTime

    val steps by Step referrersOn StepTable.recipe
    var ingredients by Ingredient via RecipeIngredientTable
}
```

CRUD Operations

```
// CREATE
val recipe = Recipe.new {
    title = recipe.title
    description = recipe.description
    cookingTime = recipe.cookingTime
}

// READ
val recipe = Recipe.findById(id) // Ingredient.find { IngredientTable.id eq id }.first()

// UPDATE
val recipe = Recipe.findById(id)
recipe?.title = title

// DELETE
val recipe = Recipe.findById(id)
recipe?.delete()
```

Transactions

```
// Basic transaction
val ingredient = transaction {
    createIngredient(entity)
}

// Coroutine support with suspend functions
val ingredient = newSuspendedTransaction {
    Ingredient.findById(id)?.toEntity()
}

// Deferred transaction
suspendedTransactionAsync(Dispatchers.IO) {
    Recipe.findById(recipeId)?.ingredients?.map(Ingredient::toEntity)
}
```

Referencing

```
// Many-to-one
object StepTable : ExtendedUUIDTable(name = "step") {
    ...
    val recipe = reference("recipe", RecipeTable)
    val recipe = reference("video", VideoTable).nullable()
}

class Step(id: EntityID<UUID>) : ExtendedUUIDEntity(id, StepTable) {
    ...
    var recipe by Recipe referencedOn StepTable.recipe
    var video by Video optionalReferencedOn StepTable.video
}

class Recipe(id: EntityID<UUID>) : ExtendedUUIDEntity(id, RecipeTable) {
    ...
    val steps by Step referrersOn StepTable.recipe
}
```

Referencing

```
// Many-to-many
object RecipeIngredientTable : Table() {
    val recipe = reference(RecipeTable.tableName, RecipeTable)
    val ingredient = reference(IngredientTable.tableName, IngredientTable)

    override val primaryKey = PrimaryKey(recipe, ingredient)
}

class Recipe(id: EntityID<UUID>) : ExtendedUUIDEntity(id, RecipeTable) {
    ...
    var ingredients by Ingredient via RecipeIngredientTable
}

class Ingredient(id: EntityID<UUID>) : ExtendedUUIDEntity(id, IngredientTable) {
    ...
    var recipes by Recipe via RecipeIngredientTable
}
```

04

Share code thanks to Swagger

A source of truth for your API

Swagger Gradle Codegen

- Gradle plugin to **generate network code** from Swagger file by Nicola Corti
- Swagger relies on **OpenAPI** for RESTful web services, supports **YAML** or **JSON** formats
- Generates **Retrofit** interfaces, uses **Moshi** for serialization and supports **Kotlin coroutines**
- **Having a single source of truth**

Yelp/swagger-gradle-codegen

Swagger module

```
swagger-api/
├── .build/
│   └── com/example/swcook/front
│       ├── apis/
│       ├── models/
│       └── tools
├── api-client.yml
└── build.gradle.kts
```

Swagger module

```
swagger-api/
├── .build/
│   └── com/example/swcook/front
│       ├── apis/
│       ├── models/
│       └── tools
├── api-client.yml
└── build.gradle.kts
```

Swagger: gradle build

```
plugins {
    id("com.yelp.codegen.plugin") version "1.4.1"
}

generateSwagger {
    platform = "kotlin-coroutines"
    packageName = "com.example.swcook.front"
    inputFile = file("./api-client.yml")
    outputDir = file("./build/")
}
```

Swagger: define paths

```yaml
paths:
  /recipes/{recipe_id}:
    get:
      tags:
        - recipes
      summary: Get recipe details
      operationId: "getRecipe"
      parameters:
        - name: recipe_id
          in: path
          required: true
          type: string
          format: uuid
      responses:
        '200':
          description: All recipes details
          schema:
            $ref: '#/definitions/Recipe'
```

Swagger: define models

```yaml
definitions:
  Recipe:
    type: object
    required:
      - id
      - title
      - cooking_time
    properties:
      id:
        type: string
        format: uuid
      title:
        type: string
      description:
        type: string
      cooking_time:
        type: integer
        format: int32
      date_published:
        type: string
        format: date-time
      steps:
        type: array
        items:
          $ref: '#/definitions/Step'
      ingredients:
        type: array
        items:
          $ref: '#/definitions/Ingredient'
```

Swagger: retrofit & models

```kotlin
@JvmSuppressWildcards
interface RecipesApi {

    /**
     * Get recipe details
     * @param recipeId (required)
     */
    @Headers("Content-Type: application/json")
    @GET("/recipes/{recipe_id}")
    suspend fun getRecipe(
        @retrofit2.http.Path("recipe_id") recipeId: UUID
    ): Recipe
}
```

Swagger: retrofit & models

```
@JsonClass(generateAdapter = true)
data class Recipe(
    @Json(name = "id") @field:Json(name = "id") var id: UUID,
    @Json(name = "title") @field:Json(name = "title") var title: String,
    @Json(name = "description") @field:Json(name = "description") var description: String,
    @Json(name = "cooking_time") @field:Json(name = "cooking_time") var cookingTime: Int,
    @Json(name = "date_published") @field:Json(name = "date_published") var datePublished:
ZonedDateTime? = null,
    @Json(name = "steps") @field:Json(name = "steps") var steps: List<Step>? = null,
    @Json(name = "ingredients") @field:Json(name = "ingredients") var ingredients:
List<Ingredient>? = null
)
```

Swagger Gradle Codegen

- **Front models** for your backend and **remote models** for your Android app will be the same
- No more object type surprise!
- Improvements: **kotlinx.serialization** support, **date** type config…
- **Having a single source of truth**

Yelp/swagger-gradle-codegen

Live Demo

(star) Watch the Ktor web service in action !

THANKS!

Do you have any questions?

Have fun with Ktor

CREDITS: This presentation template was created by **Slidesgo**, including icons by **Flaticon**, infographics & images by **Freepik**

@JulienSalvi

FOSDEM 21