Kotlin Cheat Booklet

I. Function & Keyword Reference

This section lists key functions, keywords, and constructs highlighted in the lectures, along with their purpose.

Language Basics & Control Flow

- fun main(args: Array<String>): The entry point for Kotlin applications.
- println(message: Any?): Prints the given message to the standard output.
- readLine(): String?: Reads a line of text from the standard input.
- val: Declares a read-only (immutable) variable.
- var: Declares a mutable variable.
- if (condition) {...} else {...}: Conditional execution. Can be used as an expression.
- when (value) { condition -> result ... else -> default }: Multi-branch conditional. Can be used as an expression.
- for (item in collection) {...}: Iterates over a collection or range.
- while (condition) {...}: Loop that executes as long as the condition is true.
- do {...} while (condition): Loop that executes at least once, then as long as the condition is true.
- return: Exits a function, optionally with a value.
- .. (rangeTo): Creates a range (e.g., 1..10).
- until: Creates a range excluding the end value (e.g., 1 until 10).
- downTo: Creates a reversed range (e.g., 10 downTo 1).
- step: Specifies the step in a range (e.g., 1..10 step 2).
- ?: (Elvis operator): Provides a default value if an expression is null (e.g., val length = name?.length ?: 0).
- !! (Not-null assertion): Converts a nullable type to its non-null counterpart, throwing NullPointerException if the value is null.
- is: Checks if an object is an instance of a certain type (e.g., obj is String).
- as: Performs a type cast. Can throw ClassCastException (e.g., val s = obj as String).
- as?: Performs a safe type cast, returning null if the cast is not possible.

Collections & Data Structures (Creation & Basic Ops)

- listOf<T>(...): Creates an immutable list.
- mutableListOf<T>(...): Creates a mutable list.
- setOf<T>(...): Creates an immutable set.
- mutableSetOf<T>(...): Creates a mutable set.
- mapOf<K, V>(key1 to value1, ...): Creates an immutable map.
- mutableMapOf<K, V>(...): Creates a mutable map.
- .size: Property to get the number of elements in a collection.

- .add(element): Adds an element to a mutable collection.
- .remove(element): Removes an element from a mutable collection.
- .first(): Returns the first element. Throws if empty.
- .last(): Returns the last element. Throws if empty.
- .firstOrNull(): Returns the first element, or null if empty.
- .lastOrNull(): Returns the last element, or null if empty.
- get(index) or [index]: Accesses an element at a specific index in a list or map.
- put(key, value) or map[key] = value: Adds or updates an entry in a mutable map.
- containsKey(key): Checks if a map contains a specific key.
- containsValue(value): Checks if a map contains a specific value.
- isEmpty(): Checks if a collection is empty.
- iterator(): Returns an iterator for a collection.
- hasNext() (Iterator): Checks if there are more elements.
- next() (Iterator): Returns the next element.
- remove() (MutableIterator): Removes the last element returned by next().

Object-Oriented Programming (OOP)

- class ClassName(...): Declares a class.
- constructor(...): Declares a primary or secondary constructor.
- init {...}: Initializer block, executed when an instance is created.
- get(): Defines a custom getter for a property.
- set(value) {...}: Defines a custom setter for a property (field keyword refers to the backing field).
- override: Marks a member as overriding a member from a superclass or interface.
- open: Allows a class or member to be subclassed or overridden. By default, classes and members are final.
- abstract: Declares an abstract class or member that must be implemented by subclasses.
- interface InterfaceName {...}: Declares an interface.
- super: Refers to the superclass implementation.
- this: Refers to the current instance.
- data class: Creates a class primarily to hold data, automatically generating equals(), hashCode(), toString(), copy(), and componentN() functions.
- object ObjectName: Declares a singleton object.
- companion object {...}: Declares an object tied to a class, allowing access to its members via the class name (similar to static members in Java).
- enum class EnumName(...): Declares an enumeration class.
- sealed class: Restricts class hierarchies, all direct subclasses must be declared in the same file. Useful with when expressions for exhaustive checks.
- operator fun functionName(...): Overloads an operator (e.g., plus for +, invoke for ()).
- infix fun Type.functionName(param): ReturnType: Allows calling a function with infix notation (e.g., a functionName b).
- public, private, protected, internal: Visibility modifiers.

Functions & Lambdas

- fun functionName(param: Type): ReturnType {...}: Defines a function.
- (params) -> ReturnType: Lambda expression syntax.
- ::functionName: Function reference.
- Type.() -> ReturnType: Lambda with receiver. this inside the lambda refers to an instance of Type.
- inline fun: Suggests the compiler to inline the function and its lambda arguments at the call site.
- noinline: Modifies a lambda parameter of an inline function to prevent it from being inlined.
- crossinline: Modifies a lambda parameter of an inline function to forbid non-local returns but still allow inlining.
- suspend fun: Declares a suspending function, which can be paused and resumed later. Used in coroutines.

Null Safety & Error Handling

- T?: Declares a nullable type.
- throw Exception(...): Throws an exception.
- try {...} catch (e: ExceptionType) {...} finally {...}: Handles exceptions.
- error(message: Any): Throws an IllegalStateException with the given message.
- check(value: Boolean) { lazyMessage }: Throws an IllegalStateException if the value is false.
- require(value: Boolean) { lazyMessage }: Throws an IllegalArgumentException if the value is false.
- requireNotNull(value: T?) { lazyMessage }: Throws an IllegalArgumentException if the value is null.
- TODO(reason: String): Throws a NotImplementedError.
- runCatching { ... }: Executes a block and returns a Result object (success or failure).
- Result.isSuccess, Result.isFailure, Result.getOrNull(), Result.exceptionOrNull(), Result.getOrThrow(): Methods to interact with Result.

Generics & Variance

- class MyClass<T>: Declares a generic class with type parameter T.
- fun <T> myFunction(param: T): T: Declares a generic function.
- T : UpperBound: Specifies an upper bound for a type parameter.
- out T (Covariance): Marks a type parameter as covariant. Producer<Derived> is a subtype of Producer<Base>.
- in T (Contravariance): Marks a type parameter as contravariant. Consumer<Base> is a subtype of Consumer<Derived>.
- * (Star projection): Used when type arguments are unknown or don't matter (e.g., List<*>).

Reflection

- instance::class: Gets the KClass reference for an instance.
- ClassName::class: Gets the KClass reference for a class.
- instance.javaClass or ClassName::class.java: Gets the Java Class reference.
- Class.forName(name: String): Loads a class by its fully qualified name.
- .simpleName, .qualifiedName, .members, .constructors, .supertypes: KClass properties.
- .declaredFields, .declaredMethods, .getDeclaredField(), .getDeclaredMethod(): Java Class methods.
- field.get(instance), field.set(instance, value): Accessing field values via reflection.
- method.invoke(instance, args): Calling methods via reflection.
- constructor.newInstance(args): Creating instances via reflection.
- typeof<T>(): (from kotlin.reflect) Gets KType for a type, preserving generic arguments.

Concurrency: Threads

- thread(start: Boolean, isDaemon: Boolean, ... block: () -> Unit): Thread: Creates and optionally starts a new JVM thread.
- Thread.currentThread(): Gets the current thread instance.
- Thread.sleep(millis: Long): Pauses the current thread for a specified duration.
- thread.interrupt(): Interrupts a thread.
- Thread.interrupted(): Checks if the current thread has been interrupted and clears the interrupted status.
- thread.isInterrupted(): Checks if a thread has been interrupted without clearing the status.
- thread.join(): Waits for a thread to terminate.
- Thread.yield(): Hints to the scheduler that the current thread is willing to yield its current use of a processor.
- Runnable: Interface for tasks that can be executed by a thread.
- synchronized(lockObject) {...}: A block of code that can only be executed by one thread at a time using the lockObject's monitor.
- @Synchronized: Annotates a method to be synchronized on the instance (this) or class object.
- Lock: Interface for more flexible locking mechanisms (e.g., ReentrantLock).
- lock.lock(), lock.unlock(), lock.tryLock(): Basic Lock operations.
- ThreadLocal<T>: Provides thread-local variables.
- AtomicInteger, AtomicLong, AtomicReference, etc.: Classes for atomic operations.
- compareAndSet(expect, update) (Atomic types): Atomically sets the value to update if the current value == expect.
- Collections.synchronizedList/Set/Map(collection): Wraps a collection to make it thread-safe (access synchronized).
- ConcurrentHashMap, ConcurrentLinkedQueue: Thread-safe collection implementations.
- Thread.startVirtualThread { ... }: Starts a new virtual thread (Project Loom feature).

Concurrency: Executors

- ExecutorService: Interface for managing thread pools.
- Executors.newFixedThreadPool(nThreads: Int): Creates a thread pool with a fixed number of threads.
- Executors.newCachedThreadPool(): Creates a thread pool that creates new threads as needed and reuses idle ones.
- Executors.newSingleThreadExecutor(): Creates an executor that uses a single worker thread.
- Executors.newScheduledThreadPool(corePoolSize: Int): Creates a thread pool that can schedule commands to run after a given delay, or to execute periodically.
- executor.submit(task: Callable<T> / Runnable): Submits a task for execution.
- executor.shutdown(): Initiates an orderly shutdown in which previously submitted tasks are executed, but no new tasks will be accepted.
- executor.shutdownNow(): Attempts to stop all actively executing tasks, halts the processing of waiting tasks, and returns a list of the tasks that were awaiting execution.
- executor.awaitTermination(timeout: Long, unit: TimeUnit): Blocks until all tasks have completed execution after a shutdown request, or the timeout occurs.
- ForkJoinPool: An ExecutorService for running ForkJoinTasks, suitable for divide-and-conquer algorithms.
- RecursiveTask<V> / RecursiveAction: Base classes for tasks run in a ForkJoinPool.
- task.fork(): Asynchronously executes this task in the pool the task is running in.
- task.join(): Returns the result of the computation when it is done.

Concurrency: Coroutines (kotlinx.coroutines)

- runBlocking<T> { ... }: Runs a new coroutine and **blocks** the current thread until its completion. Used for bridging blocking code to coroutines, mainly in main functions or tests.
- launch { ... }: Launches a new coroutine without blocking the current thread. Returns a Job.
- async<T> { ... }: Launches a new coroutine that computes a result. Returns a Deferred<T>.
- Job: Represents a coroutine. Can be used to cancel it or wait for its completion.
- job.join(): Suspends the coroutine until this job is complete.
- job.cancel(): Cancels the job.
- Deferred<T>: A Job that has a result.
- deferred.await(): T: Suspends the coroutine until the computation is complete and returns the result.
- CoroutineScope: Defines the scope for new coroutines. Coroutines launched in a scope are children of that scope's job.
- coroutineContext: An indexed set of elements, primarily holding the Job and CoroutineDispatcher.
- Dispatchers.Default: Backed by a shared pool of threads on the JVM. Used for

CPU-intensive work.

- Dispatchers.IO: Backed by a shared pool of on-demand created threads. Used for I/O-intensive blocking operations.
- Dispatchers.Main: A dispatcher that is confined to the main thread (e.g., UI thread in Android/JavaFX). Needs a specific implementation.
- Dispatchers.Unconfined: Starts the coroutine in the caller thread, but only until the first suspension point. After suspension, it resumes in whatever thread is used by the suspending function.
- withContext(context: CoroutineContext) { ... }: Calls the specified suspending block with a given coroutine context, suspends until it completes, and returns the result. Used for switching dispatchers.
- suspendCoroutine<T> { continuation -> ... }: Obtains the current Continuation instance and suspends the currently running coroutine. Used to adapt callback-based APIs.
- continuation.resume(value: T): Resumes the coroutine with a successful result.
- continuation.resumeWithException(exception: Throwable): Resumes the coroutine with an exception.
- delay(timeMillis: Long): Suspends the coroutine for a given time.
- withTimeout(timeMillis: Long) { ... }: Runs a block with a timeout. Throws TimeoutCancellationException if timeout is exceeded.
- withTimeoutOrNull(timeMillis: Long) { ... }: Runs a block with a timeout. Returns null if timeout is exceeded.
- yield() (coroutine context): Yields the thread (or thread pool) of the current coroutine dispatcher to other coroutines to run.

Sequences & Collection Operations

- sequence<T> { yield(value) ... }: Creates a lazily evaluated sequence.
- yield(value: T) (SequenceScope): Produces a value in a sequence builder.
- .asSequence(): Creates a lazy Sequence from an Iterable.
- **Transformation:**
  - map { transform -> ... }: Returns a list containing the results of applying the given transform function to each element.
  - mapIndexed { index, element -> ... }: Similar to map, but includes the element's index.
  - mapNotNull { transform -> ... }: Applies transform and filters out null results.
  - flatMap { transform -> ... }: Transforms each element into an iterable/sequence and flattens the results into a single list.
  - flatten(): Flattens a collection of collections into a single collection.
  - zip(otherCollection): Returns a list of pairs, where pairs are formed from elements with the same index from both collections.
- **Filtering:**
  - filter { predicate -> ... }: Returns a list containing only elements matching the given predicate.
  - filterNot { predicate -> ... }: Returns a list containing only elements not matching

the given predicate.
    - filterIsInstance<T>(): Returns a list containing only elements of the specified type.
    - filterNotNull(): Returns a list containing only non-null elements.
- **Taking/Dropping:**
    - take(n: Int): Returns a list containing the first n elements.
    - takeLast(n: Int): Returns a list containing the last n elements.
    - drop(n: Int): Returns a list containing all elements except the first n.
    - dropLast(n: Int): Returns a list containing all elements except the last n.
    - takeWhile { predicate -> ... }: Returns a list containing the first elements that satisfy the predicate.
    - dropWhile { predicate -> ... }: Returns a list containing all elements except the first ones that satisfy the predicate.
- **Element Access & Finding:**
    - distinct(): Returns a list containing only distinct elements.
    - distinctBy { selector -> ... }: Returns a list containing elements for which the selector returns unique values.
    - find { predicate -> ... } (or firstOrNull { predicate -> ... }): Returns the first element matching the predicate, or null.
    - first { predicate -> ... }: Returns the first element matching the predicate. Throws if no such element is found.
    - last { predicate -> ... }: Returns the last element matching the predicate. Throws if no such element is found.
    - single { predicate -> ... }: Returns the single element matching the predicate. Throws if none or more than one element matches.
    - singleOrNull { predicate -> ... }: Returns the single element matching the predicate, or null if none or more than one matches.
    - elementAt(index: Int): Returns the element at the given index.
    - elementAtOrNull(index: Int): Returns the element at the given index, or null if out of bounds.
    - indexOf(element: T): Returns the first index of the element, or -1 if not found.
    - lastIndexOf(element: T): Returns the last index of the element, or -1 if not found.
- **Aggregation:**
    - count { predicate -> ... }: Returns the number of elements matching the predicate (or total count if no predicate).
    - sum(): Returns the sum of elements (for numeric types).
    - sumOf { selector -> ... }: Returns the sum of values returned by the selector function.
    - average(): Returns the average of elements (for numeric types).
    - minOrNull(), maxOrNull(): Returns the minimum or maximum element, or null if empty.
    - minByOrNull { selector -> ... }, maxByOrNull { selector -> ... }: Returns the element for which the selector returns the minimum/maximum value.
    - fold(initial) { acc, element -> ... }: Accumulates value starting with initial value and

applying operation from left to right.
    - reduce { acc, element -> ... }: Accumulates value starting with the first element and applying operation from left to right.
    - foldRight(initial) { element, acc -> ... }, reduceRight { element, acc -> ... }: Similar to fold/reduce but from right to left.
- **Grouping & Associating:**
    - groupBy { keySelector -> ... }: Groups elements by the key returned by keySelector.
    - associateBy { keySelector -> ... }: Creates a map where keys are generated by keySelector and values are the elements themselves (last one wins for duplicate keys).
    - associateWith { valueSelector -> ... }: Creates a map where elements are keys and values are generated by valueSelector.
    - associate { transform -> ... }: Creates a map by transforming elements into Pairs.
- **Partitioning:**
    - partition { predicate -> ... }: Splits the collection into a Pair of lists: one with elements matching the predicate, and one with the rest.
- **Ordering:**
    - sorted(): Returns a list of elements sorted according to their natural sort order.
    - sortedBy { selector -> ... }: Returns a list of elements sorted according to the natural sort order of the values returned by the selector.
    - sortedDescending(), sortedByDescending { selector -> ... }: Similar but in descending order.
    - sortedWith(comparator: Comparator<T>): Returns a list of elements sorted according to the given comparator.
    - compareBy({ selector1 }, { selector2 }): Creates a comparator for multi-level sorting.
    - reversed(): Returns a list with elements in reversed order.
- **Set Operations:**
    - union(other: Iterable<T>): Returns a set containing all distinct elements from both collections.
    - intersect(other: Iterable<T>): Returns a set containing only elements present in both collections.
    - subtract(other: Iterable<T>): Returns a set containing elements from the original collection not present in the other.
- **Utility:**
    - joinToString(separator: String, prefix: String, postfix: String, transform: (T) -> CharSequence): Creates a string from all the elements.
    - onEach { action -> ... }: Performs the given action on each element and returns the collection itself (useful for debugging or intermediate steps).
    - shuffled(): Returns a list with elements in a random order.
    - random(): Returns a random element from the collection.

Domain-Specific Languages (DSLs)

- @DslMarker: Annotation used to control the scope of receivers in DSLs, preventing implicit access to outer receivers.
- Builder functions (e.g., html { ... }, json { ... }): Typically higher-order functions with receiver lambdas to create a DSL structure.

Kotlin Multiplatform (KMP)

- expect: Keyword used in common code to declare a platform-specific function, class, property, or typealias. The actual implementation is provided by platform modules.
- actual: Keyword used in platform-specific modules (e.g., androidMain, iosMain) to provide the concrete implementation for an expect declaration in common code.

Compose Multiplatform (UI)

- @Composable: Annotation that marks a function as a UI component builder. Composable functions describe the UI's structure and appearance. They can call other composable functions.
- @Preview: Annotation to show a preview of a composable function in Android Studio.
  - name, showBackground, device, uiMode (e.g., UI_MODE_NIGHT_YES): Parameters to customize the preview.
- MaterialTheme { ... }: Applies Material Design styling (colors, typography, shapes) to its composable children.
  - lightColors(), darkColors(): Define color palettes.
- Surface { ... }: A basic container that provides background color, elevation, and shape.
- **Layout Composables:**
  - Column { ... }: Arranges children vertically.
    - verticalArrangement: Controls spacing and alignment along the main axis (e.g., Arrangement.Top, Arrangement.Center, Arrangement.SpaceBetween).
    - horizontalAlignment: Controls alignment along the cross axis (e.g., Alignment.Start, Alignment.CenterHorizontally).
  - Row { ... }: Arranges children horizontally.
    - horizontalArrangement: Controls spacing and alignment along the main axis.
    - verticalAlignment: Controls alignment along the cross axis.
  - Box { ... }: Stacks children on top of each other. Useful for overlays or positioning elements with specific alignments.
    - contentAlignment: Controls alignment of children within the Box (e.g., Alignment.Center).
  - LazyColumn { items(...) { ... } }: Displays a vertically scrolling list of items efficiently, only composing and laying out visible items.
  - LazyRow { items(...) { ... } }: Displays a horizontally scrolling list of items efficiently.
- **Basic UI Components:**

  - Text(text = "..."): Displays text.
  - Button(onClick = { ... }) { Text("...") }: A clickable button.
  - TextField(value = state, onValueChange = { state = it }, label = { Text("...") }): An input field for text.
  - Image(painter = painterResource(...), contentDescription = "..."): Displays an image. painterResource is often used for loading drawable resources.
  - Checkbox(checked = state, onCheckedChange = { state = it }): A checkbox.
  - Card { ... }: A Material Design card, often used to group related content with elevation.
- **Modifiers (Modifier)**:
  - Used to decorate or add behavior to composables (e.g., Modifier.padding(16.dp), Modifier.fillMaxWidth(), Modifier.background(Color.Blue), Modifier.clickable { ... }, Modifier.weight(1f)).
  - Chained together to apply multiple transformations.
  - .dp: Density-independent pixels, a unit for specifying dimensions and spacing.
- **State Management:**
  - remember { mutableStateOf(initialValue) }: Creates and remembers a mutable state object. When the state's value changes, composables that read this state are recomposed.
    - var myState by remember { mutableStateOf(...) }: Using property delegation for more concise state access.
  - rememberSaveable { mutableStateOf(...) }: Similar to remember, but also saves and restores state across activity/process recreation.
  - **State Hoisting:** Pattern of moving state up the composable tree to make components more reusable and testable. Stateless composables receive state and callbacks from their parents.
- **ViewModel (androidx.lifecycle.ViewModel):**
  - A class designed to store and manage UI-related data in a lifecycle-conscious way. Survives configuration changes like screen rotations.
  - Often used with StateFlow or LiveData to expose state to Composables.
  - viewModel.count.collectAsState(): Collects a StateFlow from a ViewModel and represents it as a Compose State.
- **Navigation (androidx.navigation:navigation-compose):**
  - rememberNavController(): Creates and remembers a NavController.
  - NavHost(navController, startDestination = "route") { composable("route") { ScreenComposable(...) } ... }: Defines the navigation graph.
  - navController.navigate("destination_route"): Navigates to a new screen.
  - navController.popBackStack(): Navigates back.
  - navArgument("argName") { type = NavType.IntType }: Defines arguments for a route.
  - navDeepLink { uriPattern = "..." }: Defines deep links for a route.

Networking & Backend (Ktor)

- **Sockets (java.net.Socket, java.net.ServerSocket):**
  - ServerSocket(port): Creates a server socket that listens for incoming connections on a specific port.
  - serverSocket.accept(): Blocks until a client connects, then returns a Socket for communication with that client.
  - Socket(host, port): Creates a client socket and connects to a server.
  - socket.getInputStream(), socket.getOutputStream(): Get streams for reading from and writing to the socket.
- **Ktor (Web Framework):**
  - embeddedServer(Netty, port = 8080, module = Application::module): Starts a Ktor server using a specified engine (e.g., Netty) and configuration module.
  - Application.module(): An extension function where you configure your Ktor application (routing, plugins).
  - **Routing:**
    - routing { get("/") { call.respondText("Hello") } ... }: Defines HTTP routes and handlers.
    - get(path) { ... }, post(path) { ... }, put(path) { ... }, delete(path) { ... }: Define handlers for specific HTTP methods and paths.
    - call.respondText(text), call.respond(obj), call.respond(HttpStatusCode, message): Sends a response to the client.
    - call.receive<Type>(): Receives and deserializes the request body into an object of Type.
    - call.parameters["name"]: Accesses path parameters (e.g., /users/{name}).
    - call.request.queryParameters["name"]: Accesses URL query parameters (e.g., /search?q=name).
  - **Plugins:**
    - install(PluginFeature) { ...configuration... }: Installs and configures a Ktor plugin.
    - **ContentNegotiation**: Handles serialization/deserialization of request/response bodies (e.g., using kotlinx.serialization.json()).
    - **Routing**: Core plugin for defining routes.
    - **Authentication**: Adds authentication mechanisms (e.g., basic, jwt).
      - authenticate("authName") { route(...) }: Protects routes with a specific authentication configuration.
      - call.principal<UserIdPrincipal>(): Retrieves the authenticated principal.
    - **StatusPages**: Configures custom responses for errors or specific HTTP status codes (e.g., 404 Not Found).
      - exception<Throwable> { call, cause -> ... }: Handles specific exceptions.
      - status(HttpStatusCode.NotFound) { call, status -> ... }: Handles specific status codes.
    - **CallLogging**: Logs incoming requests and their responses.

        - **CORS (Cross-Origin Resource Sharing)**: Configures rules for cross-origin requests.
        - **MicrometerMetrics**: Integrates with Micrometer for application metrics.
            - appMicrometerRegistry.scrape(): Provides metrics in Prometheus format.
    - **Testing (testApplication { ... }):**
        - createClient { install(ContentNegotiation) { json() } }: Creates a test client for making requests to the application.
        - client.get("/path"), client.post("/path") { setBody(obj) }: Perform HTTP requests in tests.
        - response.status, response.body<Type>(): Assert on the response.
- **Exposed (SQL Library):**
    - Database.connect(url, driver, user, password): Connects to a database.
    - object Tasks : Table("tableName") { ... }: Defines a table schema.
        - integer("colName"), varchar("colName", length), bool("colName"), .autoIncrement(), .default(value).
    - transaction(database) { ... }: Executes a block of code within a database transaction.
    - SchemaUtils.create(TableObject): Creates the table in the database if it doesn't exist.
    - Table.insert { it[column] = value ... }: Inserts a new row. Returns generated keys if applicable.
    - Table.select { expression }: Selects rows matching a condition.
        - (Column eq value), (Column less value), etc. for conditions.
    - Table.update({ condition }) { it[column] = newValue }: Updates rows.
    - Table.deleteWhere { condition }: Deletes rows.
    - ResultRow.get(Column) or row[Column]: Accesses column values from a query result.

kotlinx.serialization

- @Serializable: Marks a class as serializable, enabling the compiler plugin to generate a serializer for it.
- Json { ... }: Builder for configuring Json format (e.g., prettyPrint, ignoreUnknownKeys, encodeDefaults).
- Json.encodeToString(serializer, value) or Json.encodeToString(value) (with reified type): Serializes an object to a JSON string.
- Json.decodeFromString(serializer, string) or Json.decodeFromString<T>(string): Deserializes a JSON string to an object.
- T.serializer(): Retrieves the generated serializer for class T.
- @SerialName("name_in_json"): Specifies a different name for a property in the serialized form.
- @Transient: Excludes a property from serialization.
- @Required: Marks a property as required during deserialization, even if it has a default

value (useful for ensuring presence in input).

- SerializersModule { polymorphic(Base::class) { subclass(Derived::class) { serializer(Derived.serializer()) } } }: Configures polymorphism for serialization.
- KSerializer<T>: Interface for custom serializers.
  - descriptor: SerialDescriptor: Describes the structure of the serialized data.
  - serialize(encoder: Encoder, value: T): Implements serialization logic.
  - deserialize(decoder: Decoder): T: Implements deserialization logic.
- PrimitiveSerialDescriptor(name, kind): Creates a descriptor for primitive types.
- buildClassSerialDescriptor(name) { element<Type>("propName") ... }: Builds a descriptor for class-like structures.
- encoder.encodeXxx(value), decoder.decodeXxx(): Methods on Encoder/Decoder for various types.
- encoder.beginStructure(descriptor).encodeXxxElement(descriptor, index, value).endStructure(descriptor): Pattern for composite serialization.
- decoder.beginStructure(descriptor).decodeXxxElement(descriptor, index).endStructure(descriptor): Pattern for composite deserialization.
- @Contextual: Marks a property for which the serializer should be resolved from the SerializersModule's context.
- Json.parseToJsonElement(string): JsonElement: Parses a string into a generic JsonElement tree.
- buildJsonObject { put(...) }, buildJsonArray { add(...) }: DSL for creating JsonElements programmatically.

kotlinx.datetime

- Clock.System.now(): Instant: Gets the current time as an Instant.
- Instant: Represents a specific moment in time, independent of time zone (usually UTC).
- LocalDate, LocalTime, LocalDateTime: Represent date, time, or date-time without time zone information.
- TimeZone.currentSystemDefault(), TimeZone.UTC, TimeZone.of("Zone/ID"): Represents time zones.
- instant.toLocalDateTime(timeZone: TimeZone): Converts an Instant to LocalDateTime in a given time zone.
- localDateTime.toInstant(timeZone: TimeZone): Converts LocalDateTime to an Instant assuming it's in the given time zone.
- date.plus(period), date.minus(period) (e.g., 1.days, 2.months): Date/time arithmetic using DateTimePeriod or Duration.
- date1.daysUntil(date2), date1.monthsUntil(date2), date1.yearsUntil(date2): Calculates the period between two dates.

kotlinx.benchmark (JMH based)

- @State(Scope.Benchmark): Defines the scope of state objects for benchmarks.
- @BenchmarkMode(Mode.AverageTime): Sets the benchmark mode (e.g., average time, throughput).

- @OutputTimeUnit(TimeUnit.MILLISECONDS): Sets the time unit for reporting results.
- @Warmup(iterations = N, time = T, timeUnit = TU): Configures warmup iterations.
- @Measurement(iterations = N, time = T, timeUnit = TU): Configures measurement iterations.
- @Benchmark: Marks a method as a benchmark to be executed.
- @Param("value1", "value2"): Parameterizes a benchmark with different input values.
- @Setup: Marks a method to be run before each benchmark iteration to set up state.
- @TearDown: Marks a method to be run after each benchmark iteration to clean up state.

II. Theory Overview

1. Kotlin Basics

- **Statically Typed:** Types are checked at compile time, but Kotlin has powerful type inference.
- **Platforms:** Runs on JVM, Android, JavaScript, Native (iOS, macOS, Linux, Windows, WebAssembly).
- **Variables:**
  - val: Immutable (read-only) reference. Value assigned once.
  - var: Mutable (read-write) reference. Value can be reassigned.
  - Type inference allows omitting type declaration if the compiler can infer it.
- **Basic Types:** Int, Long, Short, Byte, Double, Float, Boolean, Char, String. Unsigned types (UInt, ULong, etc.) also exist.
- **Null Safety:** A core feature to prevent NullPointerExceptions.
  - Types are non-nullable by default. To allow null, use ? (e.g., String?).
  - **Safe Calls (?.):** Execute an action only if the value is not null (e.g., name?.length).
  - **Elvis Operator (?:):** Provide a default value if an expression is null (e.g., val len = name?.length ?: 0).
  - **Not-Null Assertion (!!):** Converts a nullable type to non-null, throws NPE if null. Use sparingly.
  - Safe casts (as?).
- **String Templates:** Embed expressions in strings using $ (e.g., "Name: $name", "Length: ${name.length}").
- **Functions:**
  - Declared with fun.
  - Can have default arguments and named arguments.
  - Single-expression functions: fun double(x: Int): Int = x * 2.
  - Extension Functions: Add new functions to existing classes without modifying their source code. this refers to the receiver object.
  - Higher-Order Functions: Functions that take other functions as parameters or return functions.
  - Lambdas: Anonymous functions (e.g., { x, y -> x + y }). If the last argument to a

function is a lambda, it can be moved out of parentheses. If a lambda has only one parameter, it can be implicitly named it.
    - Function Types: e.g., (Int, Int) -> String.
    - Lambdas with Receivers: Lambdas where this refers to a specific receiver object, enabling DSL-like syntax.
- **Control Flow:**
    - if/else: Can be used as expressions.
    - when: Powerful replacement for switch. Can be used as an expression. Can match against values, ranges, types, or arbitrary conditions.
    - for loops: Iterate over anything that provides an iterator (ranges, collections).
    - while and do-while loops.
    - Ranges: 1..5, 1 until 5, 5 downTo 1, 0..10 step 2.
- **Equality:**
    - Structural equality (==, a == b): Checks if a.equals(b) is true.
    - Referential equality (===, a === b): Checks if a and b point to the same object in memory.

2. Object-Oriented Programming (OOP)

- **Classes:** Blueprints for creating objects.
    - **Constructors:** Primary (in class header) and secondary (prefixed with constructor). init blocks for initialization logic.
    - **Properties:** Variables within a class. Can have custom getters/setters. Backing field accessible via field.
    - **Methods:** Functions within a class.
- **Inheritance:**
    - Classes are final by default; use open to allow subclassing. Methods are also final by default; use open to allow overriding.
    - Use : for inheritance and interface implementation.
    - abstract class: Cannot be instantiated; may contain abstract methods (without implementation).
    - override keyword is mandatory for overriding methods/properties.
    - super keyword to access superclass members.
- **Interfaces:** Define contracts. Can contain abstract methods, methods with default implementations, and properties (abstract or with accessors).
    - A class can implement multiple interfaces.
- **Visibility Modifiers:**
    - public: Visible everywhere (default).
    - private: Visible only within the same class (or file for top-level declarations).
    - protected: Visible within the class and its subclasses.
    - internal: Visible within the same module.
- **Any:** The root of the Kotlin class hierarchy. All classes implicitly inherit from Any. Provides equals(), hashCode(), and toString().
- **Data Classes:** Auto-generates equals(), hashCode(), toString(), copy(), and

componentN() (for destructuring declarations). Ideal for classes holding data.

- **Enum Classes:** Represent a fixed set of constants.
- **Sealed Classes:** Restrict class hierarchies. All direct subclasses must be in the same file (or same package in Kotlin 1.5+ if the sealed class is in a different module). Useful for representing restricted sets of types, often used with when for exhaustive checks.
- **Objects:** Singleton instances. Can be used for utility classes, constants, or factory patterns.
- **Companion Objects:** Objects declared within a class using companion object. Members can be accessed using the class name, similar to static members in Java/C#.
- **Operator Overloading:** Allows providing custom implementations for predefined operators (e.g., +, *, [], ()).
- **Infix Functions:** Single-parameter extension functions or member functions marked with infix can be called using infix notation (e.g., a myInfixFunction b).
- **Destructuring Declarations:** Allows unpacking objects into multiple variables (e.g., val (name, age) = person). Relies on componentN() functions.

3. Collections Framework

- **Read-only vs. Mutable Collections:**
  - Read-only interfaces (List, Set, Map) provide access but no modification methods.
  - Mutable interfaces (MutableList, MutableSet, MutableMap) extend their read-only counterparts and add modification methods.
- **Collection<T>:** Root interface for collections.
- **List<T>:** Ordered collection, elements accessible by index. Duplicates allowed.
  - Implementations: ArrayList (default for mutableListOf), LinkedList.
- **Set<T>:** Unordered collection of unique elements. equals() and hashCode() are crucial for determining uniqueness.
  - Implementations: HashSet (default for mutableSetOf), LinkedHashSet (maintains insertion order).
- **Map<K, V>:** Collection of key-value pairs. Keys are unique.
  - Implementations: HashMap (default for mutableMapOf), LinkedHashMap (maintains insertion order), TreeMap (sorted by keys).
- **Iterators:** Iterator<T> for traversing collections. MutableIterator<T> allows element removal. ListIterator<T> for lists, allowing bidirectional traversal and index access.
- **Collection Operations:** Kotlin provides a rich set of extension functions for collections (see Function Reference section for examples like map, filter, groupBy, fold, etc.). These operations are generally eager for Iterables.
- **Sequences (Sequence<T>):**
  - Represent lazily evaluated collections. Operations on sequences are intermediate and return new sequences. Terminal operations trigger the computation.
  - Useful for processing large collections or complex data pipelines efficiently, as computations are performed on demand and can avoid creating intermediate collections.

    - Created using asSequence() on an Iterable or builders like sequence { yield(...) }.

4. Generics

- **Type Parameters:** Allow writing code that works with different types without sacrificing type safety (e.g., List<T>).
- **Generic Functions & Classes:** Functions or classes that operate on generic types.
- **Upper Bounds:** Constrain a type parameter to be a subtype of a specific type (e.g., <T : Number>).
- **Variance:** How subtyping of generic types relates to subtyping of their type arguments.
    - **Covariance (out T):** If Dog is a subtype of Animal, then List<Dog> is a subtype of List<Animal>. Used for producers (types that only output T). List is covariant.
    - **Contravariance (in T):** If Dog is a subtype of Animal, then Comparator<Animal> is a subtype of Comparator<Dog>. Used for consumers (types that only input T).
    - **Invariance (no in or out):** MutableList<Dog> is *not* a subtype of MutableList<Animal>. This is the default for generic types that are both producers and consumers.
    - **Use-site Variance (Type Projections):** Specify variance at the point of use, e.g., fun copy(from: List<out T>, to: MutableList<in T>).
- **Star Projection (*):** Represents an unknown type argument (e.g., List<*>). List<*> is like List<out Any?>.

5. Exceptions

- **Throwable:** The base class for all errors and exceptions.
- **Exception Handling:** try-catch-finally blocks. Kotlin does not have checked exceptions.
- **throw:** Used to throw an exception instance.
- **try as an expression:** The value of a try expression is the last expression in the try block or the last expression in a catch block.
- **Helper functions:** error(), check(), require(), requireNotNull(), TODO() for throwing standard exceptions.
- **Result<T> type:** Used with runCatching to represent a success or failure, an alternative to traditional try-catch for some scenarios.

6. Reflection

- **Introspection:** Ability of a program to examine its own structure (classes, methods, fields) at runtime.
- **Kotlin Reflection API (kotlin.reflect):** Provides KClass, KFunction, KProperty, etc. More Kotlin-idiomatic.
- **Java Reflection API (java.lang.reflect):** Also accessible. ::class.java gets the Java Class object.
- **Use Cases:** Frameworks, serialization libraries, dependency injection, tools.
- **Performance:** Reflection is generally slower than direct code execution.
- **Class Loading:** ClassLoader is responsible for loading class files at runtime.

Class.forName() can be used to load a class by name.

7. Concurrency

- **Threads (java.lang.Thread):**
    - Basic unit of concurrency on the JVM.
    - Each thread has its own stack, but shares heap memory with other threads in the same process.
    - **Daemon Threads:** Background threads that don't prevent the JVM from exiting.
    - **Thread States:** NEW, RUNNABLE, BLOCKED, WAITING, TIMED_WAITING, TERMINATED.
    - **Synchronization:** Mechanisms to control access to shared resources to prevent race conditions.
        - synchronized keyword/block: Uses an object's monitor lock.
        - java.util.concurrent.locks.Lock (e.g., ReentrantLock): More flexible locking.
    - **Atomic Operations:** java.util.concurrent.atomic classes (e.g., AtomicInteger) provide lock-free, thread-safe operations on single variables.
    - **ThreadLocal:** Creates variables that are local to each thread.
    - **Concurrent Collections:** Classes in java.util.concurrent (e.g., ConcurrentHashMap) designed for safe concurrent access.
    - **Synchronized Wrappers:** Collections.synchronizedList/Set/Map provide synchronized access to regular collections.
    - **Virtual Threads (Project Loom):** Lightweight threads managed by the JVM, designed for high-throughput I/O-bound tasks. Thread.startVirtualThread { ... }.
- **Executors (java.util.concurrent.ExecutorService):**
    - Framework for managing thread pools and decoupling task submission from execution.
    - Types: Fixed, Cached, Single, Scheduled, ForkJoinPool.
    - Benefits: Thread reuse, controlled concurrency, better resource management.
- **Coroutines (kotlinx.coroutines):**
    - Lightweight concurrency primitive, "lightweight threads". Many coroutines can run on a single thread.
    - **Suspending Functions (suspend fun):** Functions that can be paused and resumed without blocking a thread. Can only be called from other suspending functions or coroutine builders.
    - **Coroutine Builders:** launch, async, runBlocking.
    - **Job:** Handle to a coroutine, allows cancelling or waiting for completion.
    - **Deferred<T>:** A Job with a result, returned by async. await() to get the result.
    - **CoroutineScope:** Defines the lifecycle and context of coroutines. Structured concurrency: child coroutines are automatically cancelled if the parent scope is cancelled.
    - **CoroutineContext:** Holds elements like Job and CoroutineDispatcher.
    - **Dispatchers:** Determine the thread(s) a coroutine runs on (e.g., Default, IO, Main). withContext to switch dispatchers.

    - **Structured Concurrency:** Coroutines are launched within a scope, and their lifetime is bound to that scope. Errors and cancellations propagate.
    - **Cancellation:** Coroutines are cancellable. Suspending functions should be cooperative with cancellation by checking isActive.
    - **Adapting Callbacks:** suspendCoroutine can be used to bridge callback-based APIs to suspending functions.

8. Domain-Specific Languages (DSLs)

- **Internal DSLs:** Created within a host language (Kotlin) using its features to provide a specialized, fluent API for a specific domain.
- **Kotlin Features for DSLs:**
    - Higher-Order Functions & Lambdas
    - Lambdas with Receivers (key for builder patterns)
    - Extension Functions
    - Infix Notation
    - Operator Overloading
    - @DslMarker: Prevents implicit access to outer receivers in nested DSL structures, improving type safety and clarity.
- **Type-Safe Builders:** A common pattern for DSLs, e.g., for HTML, JSON, UI layouts.
- **Benefits:** Improved readability, expressiveness, type safety, reduced boilerplate for domain-specific tasks.
- **Inline Functions in DSLs:** Can improve performance by reducing overhead of lambda objects and function calls, and allow non-local returns from lambdas passed to them.

9. kotlinx Libraries

- **kotlinx.serialization:**
    - Multiplatform serialization library (JSON, Protobuf, CBOR, etc.).
    - Uses a compiler plugin for compile-time type safety and to avoid runtime reflection where possible.
    - @Serializable annotation to make classes serializable.
    - Json object for configuration and performing serialization/deserialization.
    - Supports polymorphic serialization, custom serializers (KSerializer), contextual serialization.
    - JsonElement for working with JSON trees dynamically.
- **kotlinx.datetime:**
    - Multiplatform library for working with dates and times.
    - Core types: Instant, LocalDate, LocalTime, LocalDateTime, TimeZone, Duration, DateTimePeriod.
    - Provides operations for parsing, formatting, arithmetic, and conversions.
- **kotlinx.benchmark:**
    - Multiplatform benchmarking library, often using JMH (Java Microbenchmark Harness) on JVM.
    - Annotations like @Benchmark, @State, @Setup, @Param to define and configure

benchmarks.

10. JVM Platform & Execution

- **JVM (Java Virtual Machine):** Kotlin code for the JVM compiles to Java bytecode, which runs on the JVM.
- **Interoperability:** Seamless interoperation with Java code.
- **Bytecode:** Kotlin compiler generates .class files containing JVM bytecode.
- **Execution:** Bytecode can be interpreted or Just-In-Time (JIT) compiled to native machine code by the JVM for performance.
- **Garbage Collection (GC):** JVM automatically manages memory.
- **Classpath:** Specifies where the JVM should look for class files and libraries.
- **JAR (Java Archive):** A package file format typically used to aggregate many Java class files and associated metadata and resources into one file. -include-runtime option with kotlinc can create a self-contained JAR.

11. Mobile Development with Compose Multiplatform

- **Compose Multiplatform:** A declarative UI toolkit by JetBrains, based on Jetpack Compose, for building UIs across multiple platforms (Android, iOS, Desktop, Web) from a shared Kotlin codebase.
- **Composable Functions (@Composable):**
  - The fundamental building blocks of a Compose UI. They are functions that describe a piece of UI.
  - When their input state changes, they "recompose" (re-run) to update the UI.
  - Cannot contain try-catch blocks directly related to UI logic; error handling is typically managed via state or specific Compose error boundaries if available.
- **State Management:**
  - remember { mutableStateOf(value) }: Remembers a mutable state across recompositions. Changes to this state trigger recomposition of composables that read it.
  - var count by remember { mutableStateOf(0) }: Using property delegation for more concise state handling.
  - **Recomposition:** The process of Compose re-running composable functions when their underlying state or inputs change to update the UI. It's optimized to only update the necessary parts.
  - **State Hoisting:** A pattern where state is lifted from child composables to a common ancestor, making child composables stateless and more reusable. State flows down, events flow up.
  - rememberSaveable: Similar to remember, but the state can survive configuration changes (like screen rotation) and process death on Android.
  - ViewModel (from Android Jetpack, often used with KMP): Manages UI-related data in a lifecycle-aware manner. State can be exposed via StateFlow or LiveData.
  - stateFlow.collectAsState(): Collects values from a StateFlow and represents them as Compose State, triggering recomposition on new emissions.

- **UI Components (Common Examples):**
  - Text("Display text"): For displaying strings.
  - Button(onClick = { /* action */ }) { Text("Click Me") }: A standard button.
  - TextField(value = textState, onValueChange = { textState = it }): For text input.
  - Image(painter = painterResource("drawable_id"), contentDescription = "..."): For displaying images.
  - Checkbox, RadioButton, Switch: For selection controls.
  - Card { ... }: A Material Design surface for grouping content.
- **Layouts:**
  - Column { ... }: Arranges its children vertically.
  - Row { ... }: Arranges its children horizontally.
  - Box { ... }: Stacks children on top of each other, allowing for overlays or specific alignment within its bounds.
  - LazyColumn { items(myList) { item -> Text(item.name) } }: Efficiently displays a vertically scrollable list. Only items currently visible are composed and laid out.
  - LazyRow: Similar to LazyColumn but for horizontal scrolling.
  - **Arrangement** (e.g., Arrangement.SpaceBetween, Arrangement.Center): Controls the distribution of children along the main axis of a Row or Column.
  - **Alignment** (e.g., Alignment.CenterHorizontally, Alignment.Start): Controls the placement of children along the cross axis of a Row or Column, or within a Box.
- **Modifiers (Modifier):**
  - Used to change the appearance, layout, or behavior of composables.
  - Applied by chaining (e.g., Modifier.padding(16.dp).fillMaxWidth().background(Color.Red)).
  - Common uses: padding, size, fillMaxWidth, fillMaxHeight, fillMaxSize, background, border, clickable, weight, offset, verticalScroll/horizontalScroll.
- **@Preview Annotation:**
  - Allows Android Studio to display a preview of a composable function without running the app on a device/emulator.
  - Helpful for quick UI iteration. Can have parameters like name, showBackground, device, uiMode.
- **Navigation (using androidx.navigation:navigation-compose):**
  - NavController: The central API for navigation. rememberNavController() creates an instance.
  - NavHost: A composable that defines a navigation graph. It hosts the current screen.
  - composable("route_name") { MyScreenComposable(...) }: Defines a destination in the NavHost.
  - navController.navigate("target_route"): Navigates to a destination.
  - navController.navigate("target_route/{argument_value}"): Navigates with arguments.
  - navController.popBackStack(): Goes back to the previous screen.
  - Arguments can be defined in the route string (e.g., "profile/{userId}") and

    accessed from backStackEntry.arguments.
  - Deep links can be specified for routes to allow external navigation into specific app screens.
- **Kotlin Multiplatform (KMP) Integration:**
  - Compose Multiplatform leverages KMP to share UI code.
  - Business logic, data layers, and ViewModels can be written in commonMain.
  - **expect/actual mechanism:**
    - expect declarations in commonMain define an API that platform-specific code must implement.
    - actual implementations in platform-specific source sets (androidMain, iosMain, etc.) provide the concrete behavior for those expect declarations. This is used for accessing platform-specific APIs (like file system, device sensors, specific UI elements if not covered by Compose).
  - Project structure typically includes commonMain, androidMain, iosMain, and potentially desktopMain or jsMain.

12. Backend Development with Ktor

- **Networking Basics:**
  - **IP Address:** Unique identifier for a device on a network (e.g., 127.0.0.1 for localhost).
  - **Ports:** Communication endpoints on a device (0-65535). Common ports include 80 (HTTP), 443 (HTTPS).
  - **Sockets:** Programming interface for network communication.
    - java.net.ServerSocket: Listens for incoming TCP connections on the server.
    - java.net.Socket: Represents a TCP connection endpoint (client or server-side accepted connection).
    - DatagramSocket: For UDP communication (connectionless).
  - **Client-Server Model:** Clients request services/resources from a server.
- **HTTP (HyperText Transfer Protocol):**
  - The foundation of data communication for the World Wide Web.
  - Request-Response protocol.
  - **HTTP Methods:**
    - GET: Retrieve a resource.
    - POST: Submit data to create a new resource.
    - PUT: Update/replace an existing resource completely.
    - PATCH: Partially update an existing resource.
    - DELETE: Delete a resource.
  - **HTTP Status Codes:** Indicate the outcome of a request (e.g., 200 OK, 201 Created, 400 Bad Request, 404 Not Found, 500 Internal Server Error).
  - **Headers:** Provide metadata for requests and responses (e.g., Content-Type, Authorization, Cache-Control).
- **REST (Representational State Transfer):**
  - An architectural style for designing networked applications, commonly used for

web APIs.
    - Principles: Stateless, client-server, uniform interface, resource-based (identified by URLs).
- **Ktor Framework:**
    - A lightweight, asynchronous web framework for Kotlin, built by JetBrains.
    - Modular and extensible through plugins.
    - **Server Setup:**
        - embeddedServer(Engine, port, host, module = Application::moduleName): Starts a Ktor server. Common engines include Netty, CIO.
        - Application.moduleName(): An extension function where the application's logic (routing, plugins) is defined.
    - **Routing:**
        - routing { ... }: Defines the routes for the application.
        - get("/path") { call -> ... }, post(...), put(...), delete(...): Define handlers for HTTP methods.
        - call: Represents an HTTP call (request and response).
        - call.respondText("text"), call.respond(HttpStatusCode.OK, dataObject): Send responses.
        - call.receive<DataType>(): Deserialize the request body into a Kotlin object.
        - Path parameters: get("/items/{id}") { val id = call.parameters["id"] }.
        - Query parameters: get("/items") { val category = call.request.queryParameters["category"] }.
    - **Plugins:** Enhance Ktor's functionality. Installed using install(PluginFeature) { /* config */ }.
        - **ContentNegotiation**: Handles serialization (e.g., object to JSON) and deserialization (e.g., JSON to object). Often used with kotlinx.serialization.
            - json(): Configures kotlinx.serialization for JSON.
        - **StatusPages**: Customizes error handling.
            - exception<MyException> { call, cause -> call.respond(...) }: Handles specific exceptions.
            - status(HttpStatusCode.NotFound) { call, status -> call.respond(...) }: Handles specific HTTP status codes.
        - **Authentication**: Adds security.
            - basic("name") { validate { credentials -> ... UserIdPrincipal(name) } }: Configures basic authentication.
            - authenticate("name") { /* protected routes */ }: Applies authentication to a block of routes.
            - call.principal<UserIdPrincipal>(): Accesses the authenticated user's principal.
        - **CallLogging**: Logs HTTP requests and responses.
        - **CORS**: Configures Cross-Origin Resource Sharing.
        - **MicrometerMetrics**: For exposing application metrics (e.g., for Prometheus).

- **Testing (testApplication { ... }):**
    - Provides an environment to test Ktor applications without deploying them.
    - val client = createClient { install(ContentNegotiation) { json() } }: Creates a test HTTP client.
    - client.get("/path"), client.post("/path") { setBody(data) }: Make requests.
    - Assertions on response.status and response.body<Type>().
- **Database Integration (e.g., with Exposed):**
    - Ktor itself is unopinionated about databases. Libraries like Exposed are used.
    - **Exposed:** A Kotlin SQL library from JetBrains.
        - Table objects define database table schemas.
        - Database.connect(...): Establishes a database connection.
        - transaction(db) { ... }: Executes database operations within a transaction.
        - SchemaUtils.create(MyTable): Creates tables.
        - DSL for CRUD operations: MyTable.insert { ... }, MyTable.select { ... }, MyTable.update { ... }, MyTable.deleteWhere { ... }.
    - **Repository Pattern:** Often used to abstract data access logic, making the application more modular and testable. An interface defines data operations, and an implementation (e.g., ExposedTaskRepository) uses Exposed to interact with the database.
- **Deployment:** Ktor applications can be packaged as JARs and deployed using Docker, on cloud platforms, or in servlet containers. Health checks (/health) and metrics endpoints are important for production.