Kraze: Kotlin HTTP Networking Library

Overview

Kraze is a modern, Kotlin-first HTTP networking library built on top of OkHttp. It provides a flexible, type-safe API for making HTTP requests with built-in support for multiple serialization libraries and advanced networking features.

Table of Contents

Features

- 🎯 Fluent and type-safe API
- 🔄 Multiple serialization support
  - Gson
  - Jackson
  - Kotlinx Serialization
  - Moshi
- 🔐 Built-in authentication providers
- 📝 Configurable logging
- 🌐 WebSocket support
- ⚡ Built on top of OkHttp
- 🔧 Highly extensible
- 📦 Lightweight and performant

Installation

Add the following to your `build.gradle.kts`:

```
// Core library
implementation("com.developersyndicate.kraze:kraze:1.0.0-alpha")

// Choose one or more serialization modules
implementation("com.developersyndicate.kraze:kraze-gson:1.0.0-alpha")
implementation("com.developersyndicate.kraze:kraze-jackson:1.0.0-alpha")
implementation("com.developersyndicate.kraze:kraze-kotlinx-serialization:1.0.0-alpha")
implementation("com.developersyndicate.kraze:kraze-moshi:1.0.0-alpha")
```

Core Components

NetworkClient

The primary class for making HTTP requests with support for:

- Multiple HTTP methods
- Serialization
- Authentication
- Logging
- Flexible configuration

Serialization Modules

- **kraze-gson**: Google's Gson integration
- **kraze-jackson**: Jackson integration
- **kraze-kotlinx-serialization**: Kotlinx Serialization integration
- **kraze-moshi**: Square's Moshi integration

Getting Started

Basic Client Creation

```
val client = krazeClient {
    baseUrl("https://api.example.com")
    logLevel(KrazeLoggingInterceptor.Level.BASIC)
    serializer(MoshiSerialization()) // Choose your preferred serializer
}
```

HTTP Request Methods

GET Request

```
// Simple GET request
client.get<ResponseType>("endpoint").onSuccess { response ->
    println(response)
}.onFailure { error ->
    println(error)
}

// GET with query parameters
client.get<ResponseType>("endpoint") {
    queryParam("key", "value")
    header("Authorization", "Bearer token")
}
```

POST Request

```
client.post<ResponseType>("endpoint") {
    body(requestBody)
    header("Content-Type", "application/json")
}
```

PUT and DELETE Requests

```
client.put<ResponseType>("endpoint") {
    body(updateData)
}

client.delete<ResponseType>("endpoint") {
    queryParam("id", "123")
}
```

Serialization

1. Gson Serialization

```
val client = krazeClient {
    serializer(GsonSerialization())
}
```

2. Jackson Serialization

```
val client = krazeClient {
    serializer(JacksonSerialization())
}
```

3. Kotlinx Serialization

```
@Serializable
data class User(val name: String, val age: Int)

val client = krazeClient {
    serializer(KotlinxSerialization())
}
```

4. Moshi Serialization

```
val client = krazeClient {
    serializer(MoshiSerialization())
}
```

Authentication

Basic Authentication

```
val client = krazeClient {
    authenticator(BasicAuthenticationProvider("username", "password"))
}
```

Token Authentication

```
val client = krazeClient {
    authenticator(TokenAuthenticationProvider("your-token"))
}
```

Logging

Logging Levels

```
krazeClient {
    logLevel(KrazeLoggingInterceptor.Level.NONE)    // No logging
    logLevel(KrazeLoggingInterceptor.Level.BASIC)   // Basic request/response info
    logLevel(KrazeLoggingInterceptor.Level.HEADERS) // Include headers
    logLevel(KrazeLoggingInterceptor.Level.BODY)    // Include request/response bodies
}
```

WebSocket Support

```
val webSocket = krazeWebSocket("/chat", client) {
    headers {
        "Authorization" to "Bearer token"
    }

    onOpen { webSocket, response ->
        println("Connection opened")
    }

    onMessage { webSocket, text ->
        println("Message received: $text")
    }

    onClosing { webSocket, code, reason ->
        println("Connection closing")
    }

    onFailure { webSocket, throwable, response ->
        println("Error: ${throwable.message}")
    }
}

// Send message
webSocket.send("Hello!")
```

Advanced Configuration

Timeouts and Connection Pooling

```
krazeClient {
    connectTimeout(30)  // seconds
    readTimeout(30)     // seconds
    writeTimeout(30)    // seconds
    connectionPool(ConnectionPool(5, 5, TimeUnit.MINUTES))
}
```

Caching

```
krazeClient {
    cache(File("cache_dir"), 10L * 1024 * 1024) // 10 MB cache
}
```

Error Handling

```
client.get<ResponseType>("endpoint")
    .onSuccess { response ->
        // Handle success
    }
    .onFailure { error ->
        when (error) {
            is NetworkError -> // Handle network error
            is SerializationError -> // Handle serialization error
            else -> // Handle other errors
        }
    }
```

Compatibility

- **Kotlin Version**: 1.8+ (Recommended 2.1.0)
- **Java Version**: Java 11+

- **Platforms**:
  - JVM
  - Android
  - Multiplatform (with limitations)

Best Practices

1. Always handle both success and failure cases
2. Choose serialization library based on project needs
3. Use appropriate log levels
4. Close WebSocket connections when no longer needed
5. Configure timeouts and connection pools for optimal performance

License

Acknowledgments

Built on top of OkHttp by Square, with gratitude to the open-source community.

Detailed Examples

Complete HTTP Methods Examples

GET Requests

```
// Basic GET request
val response1 = client.get<UserResponse>("users/1")

// GET with query parameters
val response2 = client.get<UserListResponse>("users") {
    queryParam("page", "1")
    queryParam("limit", "10")
    queryParam("sort", "name")
}

// GET with headers
val response3 = client.get<SecureResponse>("secure/data") {
    header("Authorization", "Bearer token123")
    header("Accept", "application/json")
}

// GET with error handling
client.get<UserProfile>("users/profile") {
    queryParam("id", userId)
}.onSuccess { profile ->
    println("Profile loaded: ${profile.name}")
}.onFailure { error ->
    when (error) {
        is IOException -> println("Network error: ${error.message}")
        else -> println("Other error: ${error.message}")
    }
}
```

POST Requests

```
// Simple POST with body
data class CreateUserRequest(val name: String, val email: String)

val newUser = CreateUserRequest("John Doe", "john@example.com")
client.post<UserResponse>("users") {
    body(newUser)
}

// POST with form data
client.post<UploadResponse>("upload") {
    multipartField("description", "Profile picture")
    multipartFile(
        "image",
        "profile.jpg",
        imageFile,
        MediaType.parse("image/jpeg")!!
    )
}

// POST with custom headers and error handling
val loginRequest = LoginRequest("username", "password")
client.post<LoginResponse>("auth/login") {
    body(loginRequest)
    header("Client-Version", "1.0.0")
}.onSuccess { response ->
    saveToken(response.token)
}.onFailure { error ->
    handleLoginError(error)
}
```

PUT Requests

```
// Update user profile
data class UpdateProfileRequest(val name: String, val bio: String)

val updateData = UpdateProfileRequest("John Smith", "Software Developer")
client.put<ProfileResponse>("users/profile") {
    body(updateData)
    header("Authorization", "Bearer token123")
}

// PUT with query parameters
client.put<DocumentResponse>("documents") {
    queryParam("version", "2")
    body(documentData)
}
```

DELETE Requests

```
// Simple DELETE
client.delete<DeleteResponse>("users/123")

// DELETE with confirmation body
client.delete<DeleteResponse>("accounts") {
    body(DeleteConfirmation("yes-delete-my-account"))
}

// DELETE with query parameters
client.delete<BatchDeleteResponse>("posts") {
    queryParam("ids", "1,2,3,4")
    header("Authorization", "Bearer token123")
}
```

Serialization Examples

Using Gson

```
data class User(
    val id: Int,
    val name: String,
    val email: String
)

val client = krazeClient {
    baseUrl("https://api.example.com")
    serializer(GsonSerialization())
}

// Automatic serialization/deserialization
client.post<User>("users") {
    body(User(1, "John", "john@example.com"))
}.onSuccess { user ->
    println("Created user: ${user.name}")
}
```

Using Kotlinx Serialization

```
@Serializable
data class Product(
    val id: Int,
    val name: String,
    val price: Double
)

val client = krazeClient {
    baseUrl("https://api.example.com")
    serializer(KotlinxSerialization())
}

client.get<List<Product>>("products").onSuccess { products ->
    products.forEach { println("${it.name}: $${it.price}") }
}
```

WebSocket Complete Example

```
data class ChatMessage(val user: String, val message: String)
data class StatusUpdate(val type: String, val content: String)

class ChatClient(private val client: NetworkClient) {
    private var webSocket: WebSocket? = null
    private val chatMessages = mutableListOf<ChatMessage>()

    fun connect() {
        webSocket = krazeWebSocket("/chat", client) {
            headers {
                "Authorization" to "Bearer ${getAuthToken()}"
                "User-Agent" to "KrazeChat/1.0"
            }

            onOpen { socket, response ->
                println("Connected to chat server")
                sendSystemMessage("User joined the chat")
            }

            onMessage { socket, text ->
                try {
```

```
                    val message = client.serialization?.decodeFromString(
                        ChatMessage::class,
                        text
                    )
                    message?.let { chatMessages.add(it) }
                    notifyNewMessage(message)
                } catch (e: Exception) {
                    println("Error parsing message: ${e.message}")
                }
            }

            onClosing { socket, code, reason ->
                println("Chat closing: $reason")
                cleanup()
            }

            onClosed { socket, code, reason ->
                println("Chat closed: $reason")
                webSocket = null
            }

            onFailure { socket, throwable, response ->
                println("Chat error: ${throwable.message}")
                handleReconnect()
            }
        }
    }

    fun sendMessage(message: String) {
        val chatMessage = ChatMessage(getCurrentUser(), message)
        val messageJson = client.serialization?.encodeToString(
            ChatMessage::class,
            chatMessage
        )
        webSocket?.send(messageJson ?: return)
    }

    fun disconnect() {
        webSocket?.close(1000, "User disconnected")
        webSocket = null
    }
}

// Usage
val chatClient = ChatClient(krazeClient {
    baseUrl("wss://chat.example.com")
    serializer(KotlinxSerialization())
})

chatClient.connect()
chatClient.sendMessage("Hello, everyone!")
// ... later
chatClient.disconnect()
```

Authentication Examples

Basic Auth with Custom Headers

```
val client = krazeClient {
    baseUrl("https://api.example.com")
    authenticator(BasicAuthenticationProvider("username", "password"))
}

client.get<SecureData>("secure/endpoint") {
    header("X-Custom-Header", "value")
}
```

Token Auth with Refresh

```
class RefreshableTokenProvider(
    private var token: String,
    private val refreshToken: String
) : AuthenticationProvider {
    override fun addAuthenticationHeaders(builder: Request.Builder) {
        builder.addHeader("Authorization", "Bearer $token")
    }

    fun updateToken(newToken: String) {
        token = newToken
    }
}

val tokenProvider = RefreshableTokenProvider(initialToken, refreshToken)
val client = krazeClient {
    baseUrl("https://api.example.com")
    authenticator(tokenProvider)
}

// Automatic token usage
client.get<ProtectedResource>("resource")

// Handle token refresh if needed
client.get<ProtectedResource>("resource").onFailure { error ->
    if (error is UnauthorizedException) {
        // Refresh token
        val newToken = refreshAuthToken(refreshToken)
        tokenProvider.updateToken(newToken)
        // Retry request
        client.get<ProtectedResource>("resource")
    }
}
```

Advanced Configuration Examples

Custom Logging

```
krazeClient {
    logLevel(KrazeLoggingInterceptor.Level.BODY)
    interceptor(CustomLoggingInterceptor())
}

class CustomLoggingInterceptor : Interceptor {
    override fun intercept(chain: Interceptor.Chain): Response {
        val request = chain.request()
        println("⬆ ${request.method} ${request.url}")

        val startTime = System.nanoTime()
        val response = chain.proceed(request)
        val duration = TimeUnit.NANOSECONDS.toMillis(System.nanoTime() - startTime)

        println("⬇ ${response.code} (${duration}ms)")
        return response
    }
}
```

Timeout and Retry Configuration

```
krazeClient {
    connectTimeout(30)
    readTimeout(30)
    writeTimeout(30)

    interceptor(object : Interceptor {
        override fun intercept(chain: Interceptor.Chain): Response {
            var retries = 0
            var response: Response? = null

            while (retries < 3) {
                try {
                    response = chain.proceed(chain.request())
                    if (response.isSuccessful) return response
                    retries++
                } catch (e: IOException) {
                    if (retries == 2) throw e
                    retries++
                }
            }
            return response!!
        }
    })
}
```

Cache Configuration

```
krazeClient {
    cache(
        directory = File("http-cache"),
        maxSize = 50L * 1024L * 1024L // 50 MB
    )

    interceptor(object : Interceptor {
        override fun intercept(chain: Interceptor.Chain): Response {
            val request = chain.request()

            // Customize cache behavior
            val cacheRequest = request.newBuilder()
                .header("Cache-Control", "public, max-age=60") // 1 minute
                .build()

            return chain.proceed(cacheRequest)
        }
    })
}
```

These examples demonstrate the full range of Kraze's capabilities and show how to implement common networking patterns and requirements using the library.