BridgeApi

BridgeApi is a sophisticated library that facilitates seamless communication between JavaScript frontend code and Kotlin backend logic using JSBridge. It provides a REST API-like interface for JavaScript developers while enabling backend developers to implement logic using familiar MVC patterns.

Features

- Seamless integration between JavaScript and Kotlin
- REST API-like interface for frontend developers
- MVC pattern support for backend developers
- Easy-to-use API for both frontend and backend
- Customizable error handling and request interceptors
- WebView integration for Android applications

Backend (Kotlin)

Getting Started

1. Add the BridgeApi dependency to your project:

```
implementation("io.github.shiniseong:bridge-api:1.1.6")
```

2. Set up your router in your main activity:

```
class MainActivity : ComponentActivity() {
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContent {
            TestBridgeApiTheme {
                Surface(modifier = Modifier.fillMaxSize(), color = MaterialTheme.colorScheme.background) {
                    WebViewScreen()
                }
            }
        }
    }
}
```

3. Implement the WebViewScreen composable:

```
@SuppressLint("SetJavaScriptEnabled")
@Composable
fun WebViewScreen() {
    val mUrl = "http://192.168.219.100:5173/"
    AndroidView(
        factory = {
            WebView(it).apply {
                settings.javaScriptEnabled = true
                webViewClient = WebViewClient()
                settings.loadWithOverviewMode = true
                settings.useWideViewPort = true
                settings.setSupportZoom(true)
                loadUrl(mUrl)
                val bridgeApi = BridgeApi(this)
                addJavascriptInterface(bridgeApi, "BridgeApi")
            }
        },
        update = {
            it.loadUrl(mUrl)
        }
    )
}
```

4. Create your controllers:

```
class UserController(private val userService: UserService) {
    @Get("/:id")
```

```
    fun getUserById(@PathVariable("id") id: Long): ApiCommonResponse {
        val user = userService.getUserById(id)
        return ApiCommonResponse(
            status = 0,
            message = "success",
            data = user.toResDto(),
        )
    }

    // ... other controller methods
}
```

5. Set up your router:

```
val router = BridgeRouter.builder().apply {
    setSerializer(objectMapper)
    registerAllErrorHandlers(listOf(serviceExceptionHandler, universalExceptionHandler))
    registerDecorator(AuthTestInterceptor())
    registerDecorator(LoggingTestInterceptor())
    registerController("api/v1/users", userController)
}.build()
```

Key Concepts

WebView Integration

BridgeApi uses WebView to facilitate communication between JavaScript and Kotlin. Here are the key components:

```
private fun WebView.resolveAsyncPromise(promiseId: String, responseJsonString: String) {
    post { evaluateJavascript(generateResolveScript(promiseId, responseJsonString), null) }
}

private fun WebView.rejectAsyncPromise(promiseId: String, errorJsonString: String) {
    post { evaluateJavascript(generateRejectScript(promiseId, errorJsonString), null) }
}

class BridgeApi(private val webView: WebView) {
    @JavascriptInterface
    fun bridgeRequest(promiseId: String, apiCommonRequestJsonString: String) {
        CoroutineScope(Dispatchers.Default).launch {
            try {
                Log.d("BridgeApi", "bridgeRequest: $apiCommonRequestJsonString")
                val result = router.bridgeRequest(apiCommonRequestJsonString)
                Log.d("BridgeApi", "bridgeRequest result: $result")
                webView.resolveAsyncPromise(promiseId, result)
            } catch (e: Exception) {
                Log.e("BridgeApi", "bridgeRequest error: ${e.message}", e)
                webView.rejectAsyncPromise(promiseId, e.serializeToJson())
            }
        }
    }
}
```

These functions handle the asynchronous communication between JavaScript and Kotlin, resolving or rejecting promises based on the result of the bridge request.

Controllers

Controllers handle incoming requests and return responses. Use annotations to define routes and request methods:

```
@Get("/:id")
fun getUserById(@PathVariable("id") id: Long): ApiCommonResponse {
    // Implementation
}

@Post("")
fun createUser(@JsonBody newUser: UserReqDto): ApiCommonResponse {
    // Implementation
}
```

Decorators

Decorators allow you to add middleware-like functionality:

```
class AuthTestInterceptor : ServiceDecorator() {
    override suspend fun serve(ctx: RequestContext): BridgeResponse {
        if (ctx.headers["Authorization"] != "Example Bearer token")
            println("AuthTestInterceptor: Unauthorized")

        return unwrap().serve(ctx)
    }
}
```

Error Handling

Custom error handlers can be registered to handle specific exceptions:

```
val universalExceptionHandler = ErrorHandler { throwable ->
    ApiCommonResponse(
        status = -99,
        message = throwable.message ?: "Unknown error",
        data = ErrorData(throwable),
    )
}
```

Frontend (TypeScript)

Getting Started

1. Install the BridgeApi client library:

```
npm install bridge-api-client-ts
```

2. Create a BridgeApi instance:

```
const customBridgeApi = BridgeApi.create({
    headers: {"Authorization": "Test Bearer token"},
    responseErrorHandler: (e) => {
        console.error(e)
        return Promise.reject(e);
    },
    timeout: 100,
})
```

3. Use the API to make requests:

```
const handleTestTimeout = async () => {
    setLoading(true);
    try {
        const res = await customBridgeApi.get('/api/v1/users/test/time-out');
        setResult(JSON.stringify(res, null, 2));
    } catch (e) {
        console.error("Error:", e);
        setResult(`Error: ${e.message}`);
    } finally {
        setLoading(false);
    }
}
```

Key Concepts

Making Requests

BridgeApi provides methods for different HTTP verbs:

```
// GET request
const user = await customBridgeApi.get('/api/v1/users/1');

// POST request
const newUser = await customBridgeApi.post('/api/v1/users', { name: 'John Doe', age: 30 });

// PATCH request
const updatedUser = await customBridgeApi.patch('/api/v1/users/1', { age: 31 });

// DELETE request
```

```
await customBridgeApi.delete('/api/v1/users/1');
```

Error Handling

You can provide a custom error handler when creating the BridgeApi instance:

```
const customBridgeApi = BridgeApi.create({
    responseErrorHandler: (e) => {
        console.error("Custom error handler:", e);
        return Promise.reject(e);
    },
})
```

License