

# Kotlin JSON-RPC 2.0 Design

---

## Abstract

This document describes the design for the JSON-RPC 2.0 for the kotlinx-rpc.

## Table of Contents

1. Overview	2
2. Conventions	2
3. JSON-RPC	2
3.1. 2.0	2
4. Serialization Strategy	2
5. Core Data Model	2
5.1. Base Message Model	2
5.2. Request & Notification	3
5.3. Response	3
5.4. Error Model	4
5.5. Validations	4
6. Client API Design	4
6.1. Request Method	5
7. Server API Design	5
7.1. Register Functions	5
7.2. Parameters	6
7.3. Call Handling	6
8. Batch Requests	6
9. Backward Compatibility	7
10. Alternatives Considered	7
10.1. Request Parameter Type	7

---

10.2. Notification Handling	7
10.3. Client Method Parameters	8
11. References	8
Author's Address	8

## 1. Overview

This document describes a Kotlin model and client/server API for JSON-RPC 2.0. The goal is to create a light weight design without any code generation needed.

Transport mechanisms, authentication, and specific behavior are intentionally excluded and left to higher-level implementations.

## 2. Conventions

The key words "MUST", "MUST NOT", "SHOULD", "SHOULD NOT", and "MAY" in this document are to be interpreted as described in RFC 2119.

## 3. JSON-RPC

### 3.1. 2.0

## 4. Serialization Strategy

Implementation SHOULD use the core library `kotlinx.serialization`, as it is part of the core libraries and is well-tested and reliable.

It also provides classes such as `JsonElement` or `JsonPrimitive`, which allow for the variable structures of JSON objects while still enabling easy serialization/deserialization.

All examples in this document assume the use of `kotlinx.serialization`. Alternative serialization frameworks may be used at the implementer's discretion.

## 5. Core Data Model

### 5.1. Base Message Model

```
sealed interface JsonRpcObject {  
    val jsonrpc: String?  
}
```

Because JSON-RPC 2.0 enforces a strict structure and a closed set of types for RPC calls, a sealed interface was used to restrict subclass declarations to the same module. The interface defines the `jsonrpc` property, which is mandatory for all JSON-RPC 2.0 objects, but sets it as nullable to allow its use with 1.0. For more information, refer to [Section Backward Compatibility \(Section 9\)](#).

To maintain this restriction, the interface is implemented using data classes only, as they are final by default.

## 5.2. Request & Notification

```
data class JsonRpcRequest (
    override val jsonrpc: String?,
    val method: String,
    val params: JsonElement? = null,
    val id: JsonPrimitive? = null
) : JsonRpcObject
```

Since Section 4 of [JSON-RPC 2.0 \[JSON-RPC-2.0\]](#) defines the `id` field to be either a string or a number, implementations MUST represent it as a `JsonPrimitive`. This is possible because the `id` is not otherwise accessed and is passed through unchanged in the response. Because `JsonPrimitive` also allows boolean values and special null values, implementations MUST perform validations during the initialization of the object to ensure that only strings or numbers are accepted.

As defined in Section 4.1 of the [JSON-RPC 2.0 \[JSON-RPC-2.0\]](#), the difference between a request and a notification is the existence of the `id`. This is modeled by making the `id` property nullable, while enforcing the correct usage through higher-level APIs. The option to have a separate class for notifications was considered. See [Section Notification Handling \(Section 10.2\)](#).

According to the specification, parameters MAY be passed either by position or by name, corresponding to a JSON array or object, respectively. To support both forms, parameters are represented using `JsonElement`. Again, validations are applied to ensure that only `JsonObject` and `JsonArray` values are accepted, as primitive values are not valid parameters. The idea of using stronger types for the parameters was considered; see [Section Request Parameter Type \(Section 10.1\)](#).

## 5.3. Response

```
data class JsonRpcResponse (
    override val jsonrpc: String?,
    val result: JsonElement? = null,
    val error: JsonRpcError? = null,
    val id: JsonPrimitive? = null
) : JsonRpcObject
```

The `result` property MUST be represented as a `JsonElement` whose value is server-defined.

As defined in Section 5 of [JSON-RPC 2.0 \[JSON-RPC-2.0\]](#), either the result or error property MUST be exclusively set. This MUST be validated at initialization as well.

## 5.4. Error Model

```
data class JsonRpcError (  
    val code: Long,  
    val message: String,  
    val data: JsonElement? = null  
)
```

The error code is represented as a Long, since the code MUST be an integer as defined in Section 5.1 of [JSON-RPC 2.0 \[JSON-RPC-2.0\]](#). Using Long accommodates the full integer range representable by a JSON number, as defined in [RFC8259].

The JsonElement was used to allow both primitive and structured values for the data property and was made nullable, as defined in Section 5.1 of [JSON-RPC 2.0 \[JSON-RPC-2.0\]](#).

## 5.5. Validations

Structural specifications that apply to RPC objects are validated during object initialization. Context-dependent validations, such as request/response correlation or result/response semantics, are enforced at higher protocol layers.

## 6. Client API Design

```
interface JsonRpcClient {  
    fun <T> request(  
        method: String,  
        parameters: JsonElement? = null,  
        id: JsonPrimitive? = null,  
        serializer: KSerializer<T>  
    ): T  
  
    fun notify(  
        method: String,  
        parameters: JsonElement? = null  
    )  
}
```

The client SHOULD expose two distinct methods: one for JSON-RPC requests and one for JSON-RPC notifications. This separation provides a clear distinction between request and notification semantics, thereby avoiding the need for additional runtime checks to differentiate between RPC call types.

Each method SHOULD expose all possible properties of the respective RPC call type.

A notification MUST NOT include the id property. If an id is exposed, it MUST be ignored. This could occur if a `JsonRpcRequest` object were passed directly instead of passing the properties as separate parameters. This design option was also considered. See [Section Client Method Parameters \(Section 10.3\)](#).

## 6.1. Request Method

The id parameter SHOULD be nullable, as the id SHOULD NOT be used for direct logic and MAY be generated by the request method. To allow user-defined logic, the id MAY be explicitly provided by the user.

The request method MAY omit the id parameter and manage request identifiers internally.

The method SHOULD return the expected generic type directly and therefore SHOULD accept a `KSerializer` capable of deserializing the result property. This allows the user to work directly with the result object without requiring additional logic. Since the user does not have access to the error object, the method MUST propagate errors to the caller.

## 7. Server API Design

```
interface JsonRpcServer {
    inline fun <reified R> register(
        method: String,
        resultSerializer: KSerializer<R>,
        metadata: RpcMetaData,
        noinline handler: suspend (List<out Any?>) -> R
    )

    method fun registerNotification(
        method: String,
        metadata: RpcMetaData,
        handler: suspend (List<out Any?>) -> Unit
    )

    fun receiveRpcCall(
        request: JsonRpcRequest
    ): JsonRpcResponse?
}
```

### 7.1. Register Functions

The server SHOULD have two registration functions to differentiate between requests and notifications. Since the notification lambda function SHOULD NOT return a result. With that, no serializer for the result is needed.

To ensure multiplatform compatibility, a serializer for the result MUST be provided as an explicit parameter. The same requirement applies to method parameters, for which the `ParameterMetaData` MUST include a corresponding deserializer.

## 7.2. Parameters

```
data class ParameterMetaData<T>(  
    val name: String,  
    val serializer: KSerializer<T>,  
    val isOptional: Boolean = false  
)  
  
data class RpcMetaData {  
    parameterNames: List<ParameterMetaData>  
}
```

To support a variable number and variety of parameters, a handler method **SHOULD** declare a single parameter of type `List<out Any?>`. Since request parameters are represented as a `JsonElement`, they **MUST** first be converted into a `List`.

For a `JsonArray`, this mapping is direct and requires no additional processing. For a `JsonObject`, the mapping is more complex, as object properties **MUST** be associated with specific list indices. To enable this association, additional metadata **MUST** be provided by the handler.

The `RpcMetaData` object therefore **MUST** include a `List` describing the parameters, which is used to map object properties to their corresponding list indices. This list **MUST** be validated to only contain non-empty distinct parameter names.

## 7.3. Call Handling

The server **MUST** implement a method that receives the RPC calls and then calls the corresponding method. The method **MUST** validate the received RPC call before handing it over to the handler.

The `receiveRpcCall` method **MUST** validate all possible errors mentioned in Section 5.1 of [JSON-RPC 2.0 \[JSON-RPC-2.0\]](#).

The `receiveRpcCall` method **MUST** locate the target method using the registered method name and invoke it via the corresponding handler with the deserialized parameters.

When handling a request, a `JsonRpcResult` object **MUST** be created and returned. If the call succeeds, this object **MUST** contain the serialized result using the `resultSerializer`. If an error occurs, it **MUST** instead contain a `JsonRpcError` describing the failure.

## 8. Batch Requests

To accommodate batch requests, the client API **MUST** be extended with a method that accepts a `List < JsonRpcRequest >`. To still have abstract methods for adding requests and notifications, a `BatchRequestBuilder` **SHOULD** be added.

```
interface BatchRequestBuilder {  
    fun addRequest(  
        method: String,  
        parameters: JsonElement? = null,  
        id: JsonPrimitive? = null  
    )  
  
    fun addNotification(  
        method: String,  
        parameters: JsonElement? = null  
    )  
}
```

TODO

## 9. Backward Compatibility

In JSON-RPC 1.0, the property `jsonrpc` is not defined and therefore not mandatory. To accommodate this, the `jsonrpc` property SHOULD be nullable. This allows the client and server to set it to null.

The JSON-RPC 1.0 also only allows parameters to be passed by position. Therefore, the client SHOULD validate the parameter type upon calling a method to ensure this.

## 10. Alternatives Considered

### 10.1. Request Parameter Type

It was considered to represent request parameters using two properties, depending on whether they were passed by position or by name. This would have been represented by a `Map<String, JsonObject>` for named parameters and a `List<JsonObject>` for positional parameters. But the need for a `kotlinx.serialization` class still remains, and thus the class `JsonElement` is used, which already summarizes these possibilities.

### 10.2. Notification Handling

The idea of using a separate data class for notifications was considered.

```
data class JsonRpcNotification (  
    override val jsonrpc: String,  
    val method: String,  
    val params: JsonElement? = null  
) : JsonObject
```

This would allow stricter type handling in the later design of the API. However, this also complicates the case of batch requests, as they may include both requests and notifications within a single array.

Ultimately, this idea was tossed due to the limited benefits. This results in additional validation that needs to be added later on, but since validation is required anyway, this approach mainly introduces additional complexity.

### **10.3. Client Method Parameters**

An alternative design was considered where client API methods accept a single `JsonRpcRequest` object rather than individual parameters. This would allow the request and notify methods to share a common signature, giving callers more flexibility in how requests are constructed.

However, exposing `JsonRpcRequest` at the API boundary would require users to work directly with protocol-level details, such as assigning request identifiers and specifying the protocol version. This increases the risk of constructing invalid messages and necessitates additional validations.

To avoid exposing protocol-specific properties such as `jsonrpc` and `id`, and to prevent invalid states by construction, the client API instead accepts method names and parameters directly.

## **11. References**

**[JSON-RPC-2.0]** Group, J. W., "JSON-RPC 2.0 Specification", URL <https://www.jsonrpc.org/specification>, 2010.

## **Author's Address**

**E. Sljivic**