

# Task 2

## Authentication and Authorization

Authentication is currently missing completely from the kotlinx-rpc protocol and depends on the transport layer to secure the endpoints. For example, for Ktor, this would mean that only whole routes can be secured, not at the method level. Currently, workarounds are used that send the authentication token as normal parameters and check them in various places on the server side.

Annotations on the interface or method level would allow us to secure the RPC without having to configure it for the selected transportation.

To access the UserPrincipal without polluting the interface method implementations with an additional context parameter, Kotlin's context parameters would allow accessing the UserPrincipal without explicitly passing it into the method.

Two options could be considered to set the context parameters.

- Generate a proxy that would be called instead of the service implementation. With this, the authentication and authorization can be encapsulated, and the context parameters can be set here and later accessed as needed.
- Add a function handleSecureCall to the KrpcServerService.kt, which would check authorization and then set the context again. This would rather be the quick and dirty solution and may not be extendable in the future in a readable way.

## General Context Parameters

This concept could be extended to provide general support for context parameters in kotlinx-rpc, for example, to access the RPC request context. Incorporating this would require changes to the generator to retrieve the required context parameters, as well as updates to the server to properly set these parameters. These could take over the same ideas as for the authentication context.

## Exception Handling

### Server

In the event of an exception, the server catches it and maps all values of the Throwable to a SerializedException, including the Stacktrace. This exposes internal structures and may pose security risks; therefore, it should be avoided.

### Client

In case of a CallException, the Client propagates the exception directly to the caller of the service method. This means that the service must handle all CallExceptions that may have been thrown by the server. In many instances, a centralized exception handler would make sense to avoid duplicating code. For example, when considering authentication errors.

This also means there has to be a better way to differentiate exception types across the client/server system.

Current code:

```
KrpcCallMessage.CallException -> {
    val cause = message.cause.deserialize()
    channel.close(cause)
    channel.cancel(CancellationException("Call failed", cause))
}
```

The concept is to provide an `RpcExceptionHandler` that acts as an intermediary between the client and the generated RPC methods. By registering this handler on the client, all exceptions from server calls can be intercepted, mapped to domain-specific types, or handled centrally.

This concept could be extended to generally allow interceptors (not only for exceptions) on the client and server sides.

## Overall Error Concept

Looking through the exception handling from both sides, it is not predictable which errors might occur. Possible validation errors would have to be propagated using an exception.

Inspired by this [proposal](#), a better approach might be to treat errors as values instead of throwing them. For example, server-to-client communication could use `Result`, which at least signals that something might go wrong and forces the client to handle it. It's not perfect, because the exact error types aren't fully visible, but it's a step toward clearer and safer code.

If the KEEPAE'0441 proposal is accepted, this could go even further. RPC calls could return rich error types directly in their signatures, showing exactly which errors can happen. Then the generator could create client code that handles each case explicitly, making errors predictable, reducing surprises, and helping developers feel confident that they're covering all possibilities. Overall, it would make error handling in distributed applications much clearer, safer, and easier to work with.

## Missing Serializable

```
@Rpc
class MyService {
    fun foo() : SomeClass
}

data class SomeClass(x: Int)
```

Example from ([547](#)).

Currently, a runtime error is thrown if a non-serializable class is used in an RPC service. This error is inevitable and thus can also be replaced by a compile error. With the experimental idea of `annotationTypeSafetyEnabled`, which ensures that only interfaces with `@RPC` are registered, the concept could be extended to classes used within an RPC service.

In some cases, as `RpclInvokator.call`, the `parameters` parameter could be annotated with `@Serializable`, but I have not checked all usages, and some may need different logic for this.