RpcContextCreator

August 18, 2026

RpcContextCreator

Kind: Class

Source: `packages/microservices/context/rpc-context-creator.ts`

Part of: Microservices

`RpcContextCreator` builds the executable wrapper for NestJS microservice RPC handlers. It reflects handler parameter metadata, resolves guards, pipes, interceptors, and exception handling, then returns a function that produces an `Observable` for each incoming RPC request.

Methods

Method	Signature	Returns
`create`	`create(instance: Controller, callback: (...args: unknown[]) => Observable<any>, moduleKey: string, methodName: string, contextId: undefined, inquirerId: string, defaultCallMetadata: Record<string, any>)`	`(...args: any[]) => Promise<Observable<any>>`
`reflectCallbackParamtypes`	`reflectCallbackParamtypes(instance: Controller, callback: (...args: unknown[]) => unknown)`	`unknown[]`
`createGuardsFn`	`createGuardsFn(guards: any[], instance: Controller, callback: (...args: unknown[]) => unknown, contextType: TContext)`	`Function
`getMetadata`	`getMetadata(instance: Controller, methodName: string, defaultCallMetadata: Record<string, any>, contextType: TContext)`	`RpcHandlerMetadata`

Method	Signature	Returns
`exchangeKeysForValues`	`exchangeKeysForValues(keys: string[], metadata: TMetadata, moduleContext: string, paramsFactory: RpcParamsFactory, contextFactory: (args: unknown[]) => ExecutionContextHost)`	`ParamProperties[]`
`createPipesFn`	`createPipesFn(pipes: PipeTransform[], paramsOptions: (ParamProperties & { metatype?: unknown })[])`	`void`
`getParamValue`	`getParamValue(value: T, { metatype, type, data }: { metatype: any; type: any; data: any }, pipes: PipeTransform[])`	`Promise<any>`

Where it refuses work

- `RpcContextCreator` stops the work with `RpcException` when `!canActivate`.
- `RpcContextCreator` stops the work with an early return when `cacheMetadata`.

Diagram

```
graph LR
  A[Incoming RPC message] --> B[RpcContextCreator.create]
  B --> C[Reflect handler parameter metadata]
  C --> D[Create guards, pipes, and interceptors]
  D --> E[Resolve RPC handler arguments]
  E --> F[Invoke controller method]
  F --> G[RpcProxy exception handling]
  G --> H[Observable response]
```

Usage

```
import { lastValueFrom } from 'rxjs';
import { RpcContextCreator } from '@nestjs/microservices/context/rpc-context-creator';

class CustomTransportAdapter {
  constructor(
    private readonly rpcContextCreator: RpcContextCreator,
  ) {}
```

```
  registerHandler(
    controller: object,
    moduleKey: string,
    methodName: string,
  ) {
    const callback = (controller as any)[methodName];

    // Nest internally creates this wrapper when binding RPC controller methods.
    const execute = this.rpcContextCreator.create(
      controller,
      callback,
      moduleKey,
      methodName,
    );

    return async (payload: unknown, rpcContext: unknown) => {
      const response$ = await execute(payload, rpcContext);

      // Convert the handler Observable when the transport needs a Promise.
      return lastValueFrom(response$);
    };
  }
}
```

AI Coding Instructions

- Use `create()` when binding an RPC controller method; it ensures guards, pipes, interceptors, and exception filters are applied consistently.
- Preserve the original controller instance and method name so reflected parameter metadata can be resolved correctly.
- Treat the function returned by `create()` as asynchronous and expect it to resolve to an `Observable`.
- Do not invoke private metadata, guard, or pipe helper methods directly; extend integration behavior through NestJS guards, pipes, interceptors, and exception filters.
- Pass the correct module key and request context when integrating custom transports, especially when request-scoped providers are involved.