ReplContext

August 17, 2026

ReplContext

Kind: Class

Source: `packages/core/repl/repl-context.ts`

Part of: Core

`ReplContext` encapsulates the runtime context used by the REPL subsystem. It provides output handling through `writeToStdout()`, allowing REPL execution code to send its current output to the standard output stream.

Methods

Method	Signature	Returns
`writeToStdout`	`writeToStdout(text: string)`	`void`

Properties

Property	Type
`logger`	`any`
`debugRegistry`	`Record<ModuleKey, ModuleDebugEntry>`
`globalScope`	`ReplScope`
`nativeFunctions`	`any`

Where it refuses work

- `ReplContext` stops the work with an early return when `moduleName === InternalCoreModule.name`.
- `ReplContext` stops the work with an early return when `stringifiedToken === ApplicationConfig.name || stringifiedToken === moduleRef.metatype.na⋯`.

- `ReplContext` stops the work with an early return when `stringifiedToken === ModuleRef.name`.

Diagram

```
graph LR
  REPL[REPL Command / Evaluator] --> Context[ReplContext]
  Context --> Output[writeToStdout()]
  Output --> Stdout[Process Standard Output]
```

Usage

```
import { ReplContext } from "./repl-context";

// Typically created and managed by the REPL runtime.
declare const context: ReplContext;

// Write the context's current output to standard output.
context.writeToStdout();
```

AI Coding Instructions

- Keep terminal-output behavior centralized in `ReplContext` rather than writing directly to `process.stdout` from REPL command handlers.
- Call `writeToStdout()` only after the context has been populated with the output intended for the user.
- Preserve the existing REPL context lifecycle when adding commands or evaluators; prefer receiving a `ReplContext` instance over creating unrelated output state.
- Avoid coupling REPL evaluation logic to a specific terminal implementation; use the context as the integration boundary for output.

How it works

`ReplContext` is the state holder used to populate and operate the Nest REPL. It retains the application context, a `Logger` named `ReplContext`, a module-debug registry, a null-prototype global scope, and a map of native REPL-function instances. [packages/core/repl/repl-context.ts:29-40]

Its constructor accepts an `INestApplicationContext` and an optional array of `ReplFunction` classes. At runtime, it reads `app.container` through an `any` cast, then immediately builds the module scope and registers built-in plus supplied native functions. Therefore, the supplied application object must expose a container with `getModules()` when constructed. [packages/core/repl/repl-context.ts:39-47] [packages/core/repl/repl-context.ts:53-56]

- `globalScope` starts as `Object.create(null)`, so it has no inherited object properties. [packages/core/repl/repl-context.ts:27,32]
- For every container module except `InternalCoreModule`, the context adds the module metatype to `globalScope` under its class name. If that name is already truthy in the scope, it appends `(${moduleRef.token})` to distinguish the module key. These module properties are enumerable and non-configurable. [packages/core/repl/repl-context.ts:56-74]
- For each module's `providers` and `controllers`, it records token names in `debugRegistry[moduleKey]`. It excludes entries whose stringified token matches `ApplicationConfig`, the module metatype name, or `ModuleRef`; only the first two exclusions also prevent adding the token to `globalScope`. [packages/core/repl/repl-context.ts:77-112]
- Tokens are named as follows: string tokens become quoted strings such as `"token"`; function tokens use `token.name`; other values use `token?.toString()`. [packages/core/repl/repl-context.ts:114-120]
- A token is added to `globalScope` only when no truthy value already exists under that stringified name. The resulting property is enumerable and non-configurable. [packages/core/repl/repl-context.ts:92-99]

The standard `repl()` bootstrap creates this context after initializing the application, then copies `globalScope` property descriptors into Node's `replServer.context`. [packages/core/repl/repl.ts:16-22,25-32] This makes the collected module and injection-token references available in that REPL context. [packages/core/repl/repl-context.ts:65-74,92-99] [packages/core/repl/assign-to-object.util.ts:5-16]

`debugRegistry` has one entry per included module key, with separate `controllers` and `providers` records mapping stringified names to their original injection tokens. [packages/core/repl/repl-context.ts:21-25,77-112] The built-in `debug()` function reads this registry to print either all modules or one selected module; if a requested module key is absent, it logs an error. [packages/core/repl/native-functions/debug-repl-fn.ts:15-36]

The context always registers these built-in function classes: `HelpReplFn`, `GetReplFn`, `ResolveReplFn`, `SelectReplFn`, `DebugReplFn`, and `MethodsReplFn`; optional function classes are appended after them. [packages/core/repl/repl-context.ts:165-184] Each class is constructed with this context and stored in `nativeFunctions` under its declared name. [packages/core/repl/repl-context.ts:122-129] Declared aliases create objects inheriting from the primary function instance, with alias-specific name metadata, and are also added to the map. [packages/core/repl/repl-context.ts:130-142]

Every registered function name, including aliases, is assigned to `globalScope` as an `action` method bound to its function instance. [packages/core/repl/repl-context.ts:145-151] The bound function gets a non-enumerable, non-configurable `help` getter; reading it writes that function's formatted help message to standard output. [packages/core/repl/repl-context.ts:152-162] Function metadata requires a name, description, and signature, with optional aliases; a function class must accept a `ReplContext` and create a `ReplFunction`. [packages/core/repl/repl.interfaces.ts:4-21]

The built-ins use the retained application context for `get`, `resolve`, and `select`, while `methods` resolves string tokens through `app.get()` before scanning prototype method names. [packages/core/repl/native-functions/get-repl-fn.ts:14-16] [packages/core/repl/native-functions/resolve-repl-fn.ts:13-18] [packages/core/repl/native-functions/select-relp-fn.ts:17-19] [packages/core/repl/native-functions/methods-repl-fn.ts:17-30] The built-in `help()` function reads and sorts `nativeFunctions`, then writes each function's name and description. [packages/core/repl/native-functions/help-repl-fn.ts:17-30]

`writeToStdout(text)` directly calls `process.stdout.write(text)`, creating output as its visible side effect. [packages/core/repl/repl-context.ts:49-51]