Extras

August 18, 2026

Extras

Kind: Interface

Source: `packages/core/inspector/interfaces/extras.interface.ts`

Part of: Core

`Extras` groups enhancer definitions discovered by the inspector into two categories: orphaned enhancers and enhancers attached to known targets. It provides a consistent shape for consumers that need to inspect, display, or process enhancer metadata while preserving whether each enhancer is currently associated with an entity.

Properties

Property	Type
`orphanedEnhancers`	`Array<OrphanedEnhancerDefinition>`
`attachedEnhancers`	`Array<AttachedEnhancerDefinition>`

Diagram

```
graph LR
  Extras[Extras]
  Orphaned[orphanedEnhancers<br/>Array&lt;OrphanedEnhancerDefinition&gt;]
  Attached[attachedEnhancers<br/>Array&lt;AttachedEnhancerDefinition&gt;]

  Extras --> Orphaned
  Extras --> Attached
```

Usage

```ts
import type { Extras } from "./interfaces/extras.interface";

function summarizeExtras(extras: Extras): string[] {
  return [
    `Attached enhancers: ${extras.attachedEnhancers.length}`,
    `Orphaned enhancers: ${extras.orphanedEnhancers.length}`,
  ];
}

const extras: Extras = {
  attachedEnhancers: [],
  orphanedEnhancers: [],
};

console.log(summarizeExtras(extras));
```

AI Coding Instructions

- Always provide both `attachedEnhancers` and `orphanedEnhancers`; use empty arrays when no definitions exist.
- Place enhancer definitions in `attachedEnhancers` only when they have a valid inspected target or association.
- Preserve orphaned definitions instead of discarding them, as they may be required for diagnostics or later reconciliation.
- Use the specific `AttachedEnhancerDefinition` and `OrphanedEnhancerDefinition` types rather than mixing both categories into a shared untyped array.

How it works

`Extras` is a TypeScript interface for the `extras` section of a serialized inspector graph. A `SerializedGraphJson` contains `extras: Extras` alongside its nodes, edges, and entrypoints. [serialized-graph-json.interface.ts:8-14]

It requires two arrays:

- `orphanedEnhancers: Array<OrphanedEnhancerDefinition>` records globally registered filters, pipes, interceptors, and guards. Each entry has a `subtype` limited to `'guard'`, `'interceptor'`, `'pipe'`, or `'filter'`, plus a `ref` typed as `unknown`. [extras.interface.ts:10-20] [constants.ts:26-34]
- `attachedEnhancers: Array<AttachedEnhancerDefinition>` records enhancers attached through `APP_PIPE`, `APP_GUARD`, `APP_INTERCEPTOR`, or `APP_FILTER`; each entry contains the associated graph `nodeId`. [extras.interface.ts:3-8] [core/constants.ts:13-28]

`SerializedGraph` initializes both arrays as empty and appends entries through `insertOrphanedEnhancer()` and `insertAttachedEnhancer()`. It includes the same `extras` object in `toJSON()` output; neither insertion method validates entries or removes duplicates. [serialized-graph.ts:26-34] [serialized-graph.ts:111-119] [serialized-graph.ts:125-140]

When application methods register global filters, pipes, interceptors, or guards, each resulting item is added as an orphaned enhancer with its corresponding subtype. Before it reaches the graph, `GraphInspector` replaces the entry's `ref` with `entry.ref.constructor.name`, falling back to `'Object'` when no constructor name is available. [nest-application.ts:403-454] [graph-inspector.ts:86-91]

For attached enhancers, the scanner resolves the registered application provider's wrapper, records it as attached, and applies it through the request-scoped/transient or regular provider path. [scanner.ts:666-703] `GraphInspector.insertAttachedEnhancer()` looks up the wrapper's existing graph node, sets that node's `metadata.global` to `true`, and appends its ID to `attachedEnhancers`; there is no runtime check for a missing node before accessing its metadata. [graph-inspector.ts:93-98]