G1: Enforce Modular Boundaries

G1: Enforce Modular Boundaries

In this section, we present one of the concrete guidelines proposed for evolving a modular monolith application. G1 focuses on enforcing clear modular boundaries. The guideline aims to prevent unintended coupling in the codebase, preserve encapsulation, and ensure that dependencies between modules remain explicit and verifiable through build- and test-time checks.

Intent and Rationale

This research proposes that modularity in a monolith should be treated as an enforceable architectural property rather than an assumed outcome of refactoring. Without enforcement, boundaries tend to degrade over time through convenience-driven imports, shared utilities, and implicit wiring, which gradually reintroduces tight coupling. G1 therefore treats boundary enforcement as a preventive control, making coupling explicit, reviewable, and auditable, enabling independent evolution of modules and reducing the cost of architectural change.

Conceptual Overview

Define clear boundaries around each bounded-context module so that:

- Each module encapsulates its internal implementation and feature-level organization.
- Dependencies between modules occur only through explicitly declared and verifiable interfaces.
- Unintended coupling is prevented, enabling independent evolution of modules within a single deployable system.

Applicability Conditions and Scope

G1 is applicable when a system is decomposed into logical modules within a single codebase. Inter-module dependencies must be explicitly declared and automatically verifiable, independent of packaging, deployment, or distribution decisions. This guideline assumes a two-level decomposition strategy:

- *Bounded Context Modules:* The codebase is decomposed into bounded-context modules (for example, under a `modules/` root), and each bounded context is treated as a first-class module boundary.

- *Package-by-Feature Within Modules:* Inside each module, code is organized by feature (use cases, domain concepts, aggregates) to localize behavior and reduce scattering, without weakening the outer module boundary.

G1 focuses on enforcing the *outer* boundary between bounded-context modules. The internal package-by-feature organization is treated as a complementary structuring mechanism, and its detailed rules are addressed in subsequent topics.

Objectives

- *Clear Declaration and Isolation:* Ensure each module is clearly identified (for example, via naming conventions, package control, or a descriptor) and remains isolated unless an explicit dependency is declared.
- *Encapsulation:* Keep internal classes, data, and resources hidden within a module.
- *Controlled Dependency:* Allow one module to depend on another only when explicitly declared.
- *Visibility Management:* Declare which services, events, or commands a module exposes or consumes.
- *Early Verification:* Detect boundary violations at build or test time rather than at runtime.
- *Isolation Guarantees:* Optionally forbid certain modules from ever depending on each other.

Key Principles

- *Provided Interface vs. Internal Components:*
    - *Provided Interface:* Public classes, services, APIs, or events explicitly exposed to other modules.
    - *Internal Components:* Classes, utilities, or data that should remain private within the module.
- *Explicit Declarations:* All dependencies on other modules must be declared in a central descriptor to provide auto-verifiability (for example, annotation, configuration file, or YAML).
- *Forbidden Dependencies:* Declare modules that must never depend on each other to enforce stronger isolation where necessary.
- *Inter-Module Communication:* Use either synchronous calls via public interfaces or asynchronous event-driven channels, and avoid direct references to another module's internals.
- *Boundary-First Discipline:* Package-by-feature improves cohesion within a bounded context, but it does not replace module boundaries. A feature slice must not become a cross-module dependency mechanism (for example, by importing internal feature code from other modules).

Implementation Mechanisms

This research proposes implementing G1 through a combination of (i) structural encapsulation, (ii) explicit dependency declaration, and (iii) automated verification:

- *Structural encapsulation:* Separate public API packages from internal packages, and restrict visibility so that only the declared API is importable across module boundaries.
- *Declarative dependency model:* Maintain a module descriptor that defines which modules are required and which are forbidden, and keep it version-controlled with boundary changes.
- *Automated verification:* Fail the build when undeclared cross-module references are detected, and optionally verify event subscriptions and command usage against declared dependencies.

Common Failure Modes and Anti-Patterns

The following failure modes are frequently observed in codebases that lack explicit boundary enforcement. Each anti-pattern weakens modular isolation and increases the cost of future architectural change.

- *Convenience-Driven Imports:* Developers import internal classes from another module because the path is shorter or the type is already available. Over time, these imports create a web of hidden coupling that boundary checks cannot detect if enforcement is absent. This anti-pattern is captured by an increase in C_{leak} and a decrease in ρ_{api}.
- *Shared Utility Creep:* A `shared/` or `common/` package grows to include domain-specific logic that belongs inside a bounded context. Modules become coupled through this shared layer rather than through explicit contracts. This is reflected in rising C_{undecl} when shared utilities reference module internals.
- *Implicit Wiring via Framework Magic:* Dependency injection containers, auto-configuration, or classpath scanning silently wire modules together without any declared dependency. Boundary checks based on import analysis miss these couplings entirely. This failure mode surfaces through declining P_{iso} when module isolation tests fail due to undeclared runtime dependencies.
- *Event Handlers Registered Outside the Composition Root:* When event subscriptions are scattered across module initializers rather than centralized in a single wiring location, the integration surface becomes invisible to architectural review. This increases C_{event} and makes the system's collaboration topology difficult to audit.
- *Boundary Enforcement Treated as Optional:* When boundary verification gates are not part of the mandatory CI pipeline, violations accumulate silently. Teams discover the erosion only when a refactoring or extraction attempt reveals that modules are far more coupled than the architecture documentation suggests.

Metrics and Verification

- *Undeclared Dependency Reference Count:* Number of static references from module A to module B where $A \rightarrow B$ is not declared in the module descriptor.
- *Forbidden Dependency Reference Count:* Number of static references from module A to module B where $A \rightarrow B$ is explicitly forbidden.
- *API-Only Dependency Ratio:* For all cross-module references, the proportion that targets the provider module's declared public API surface (interfaces, exported packages, published events) rather than internal packages.
- *Encapsulation Leakage Count:* Number of external references to internal symbols (classes, packages, resources) that are not part of the provider module's declared public surface.
- *Module Isolation Test Pass Rate:* Percentage of module-scoped tests that execute with only the module and its declared dependencies available (for example, module-level context loading). Failures indicate hidden wiring to undeclared modules, even when imports look compliant.
- *Event Subscription Boundary Violations (if events are used):* Number of event handlers in module A consuming events from module B without a declared $A \rightarrow B$ dependency (or without a declared event subscription, depending on the enforcement model).
- *Boundary Bypass Surface Count (optional):* Number of occurrences of mechanisms that can bypass static boundaries (for example, reflection-based access, service locator patterns, dynamic class loading) within cross-module interaction paths. This metric is intended as a risk indicator even when no violations are detected statically.

Documentation Guidelines

- *Module Descriptor:* Maintain a machine-readable descriptor (YAML, annotation, or code) that centralizes *requires* and *forbids* entries for each module.
- *Change Log:* Whenever a boundary is added, removed, or modified, update the descriptor with version and author metadata.

Tooling Capabilities Checklist

Any open-source or proprietary tool used to enforce modular boundaries should support:

- *Module Discovery:* Automatically identify modules via conventions (for example, module folders, annotations) or explicit configuration.

- *Boundary Verification:* Perform automated checks during build/test to detect undeclared cross-module references and fail on violations.
- *Event-Driven Enforcement:* Track published events and subscribers, enforcing that only modules with declared subscriptions handle specific events.
- *Isolated Testing:* Allow tests to load only a given module (and its declared dependencies), failing early if the module tries to wire code from undeclared modules.
- *Documentation Artifacts:* Generate up-to-date artifacts or reports for inclusion in architecture documentation or CI/CD feedback.
- *Runtime Validation (Optional):* Optionally perform runtime checks to catch dynamic boundary violations (for example, reflection) that compile-time checks might miss.

Literature Support Commentary

Although modularity is widely recognized as essential, much of the research treats it as an outcome of refactoring rather than a design criterion. Few works propose proactive structural mechanisms for enforcing modularity in monoliths, and those that do often lack empirical validation. This gap motivates G1 as a guideline that operationalizes modular boundaries through explicit declarations and automated verification. Industry experience reports reinforce this need: Shopify's engineering team documented how their large-scale monolith was restructured around enforced modular boundaries to sustain development velocity , and Jovanovic reports similar findings from rewriting a legacy system as a modular monolith, noting that boundary enforcement was the decisive factor in controlling complexity.

G1 Applied: The Modular Tiny Store Example

This section operationalizes Guideline G1 (Enforce Modular Boundaries) using Tiny Store, a small modular monolith maintained as an executable reference for this dissertation.[1] The objective is to show how G1 is applied as an engineering practice, not only as an architectural principle. In Tiny Store, modularity is encoded in the repository structure, integration is wired in a visible composition root, and boundary erosion is surfaced as a failing check. The tutorial therefore connects the guideline to concrete actions: introduce a controlled violation, observe a crisp failure signal, trace it to an unauthorized dependency, and repair the system by restoring an explicit contract between bounded contexts.

The tutorial is written from the perspective of an engineer evolving a monolithic application that contains multiple business capabilities. The goal is to keep bounded contexts isolated, to ensure that any cross-module dependency is intentional and reviewable, and to prevent accidental coupling from accumulating into a refactoring bottleneck.

Two complementary feedback loops are used throughout the steps. The first is `npm run test:boundary`, which validates structural rules and fails when a module imports forbidden internals or bypasses the public surface of another context. The second is `npm run test:integration`, which exercises cross-module behavior end-to-end and confirms that the system remains correct after boundaries are tightened or contracts are introduced. Together, these targets support a workflow that is both practical and repeatable: make a change, observe the result immediately, and keep architectural integrity measurable as the system evolves.

Reader map

This tutorial is designed to be completed in a single sitting. It assumes you can run the repository's boundary and integration targets locally, and it keeps the steps intentionally hands-on. You will first trigger a boundary failure on demand, then locate the exact import or export that caused the violation, and finally fix it by introducing an explicit modular contract rather than importing another context's internals. The expected time investment is 20–30 minutes, and the outcome is a concrete, reproducible ability to detect and remediate boundary erosion using the same mechanisms that can be embedded into a continuous integration pipeline.

Repository Orientation

Tiny Store is the reference codebase used in this dissertation to ground architectural claims in an executable artifact. Its structure is designed to make modularity observable: boundaries are explicit in the directory layout, integration points are concentrated in a few predictable places, and the allowed dependency direction can be verified through automated checks.

The repository is an Nx monorepo: a single workspace that contains multiple projects (applications and libraries) managed under a shared toolchain and an inspectable dependency graph.[2] Deployable applications live under `apps/`, while reusable libraries live under `libs/`. This separation encodes a core rule of the modular monolith: composition roots are allowed to wire dependencies, while modules remain cohesive units that expose controlled integration surfaces.

The system's HTTP boundary is the `apps/api/` project, implemented with Next.js (App Router). REST endpoints under `apps/api/src/app/api/` act as thin adapters that translate requests into application operations. The same project also hosts shared runtime wiring under `apps/api/src/app/lib/`, so the API becomes the main composition root that assembles bounded contexts and connects infrastructure such as persistence and the event bus.

Cross-module integration is intentionally made visible. Listener subscription wiring is centralized in `apps/api/src/app/lib/register-listeners.ts`, which serves as an auditable map of event choreography: which events are subscribed to, which handlers react, and which dependencies exist at the system level. Centralizing subscriptions avoids hidden side effects (for example, listeners registered implicitly during import time) and makes integration decisions reviewable and testable.

Initially, domain behavior was divided into four bounded contexts under `libs/modules/`: `orders`, `inventory`, `payments`, and `shipments`. Each context is an Nx library structured around three concerns. The `domain/` subtree holds aggregates, entities, value objects, domain events, and repository abstractions. The `features/` subtree holds application use cases organized as vertical slices. The `listeners/` subtree reacts to published events and invokes local features, enabling collaboration without direct access to another context's internals.

Modules are consumed through a deliberate public API surface. Each bounded context exposes an entrypoint, typically `modules/<context>/src/index.ts`. This entrypoint is the module's contract: it re-exports only approved symbols (handlers, listener factories, public types, event contracts) and keeps domain internals private. As a result, other projects import capabilities through stable module-level paths (for example, `@tiny-store/modules-orders`) while boundary discipline remains enforceable.

The project makes event-driven collaboration the primary cross-module trace. Instead of "class-to-class" dependencies, the relevant path is "event → listener → feature." Orders emits domain events, Inventory reacts to reserve stock, and subsequent events drive Payments and Shipments. This choreography is expressed in each context's `domain/events/`, implemented by `listeners/`, and wired in the composition root via `register-listeners.ts`.

Statuses description:

Order created, awaiting inventory check

Inventory reserved, ready for payment

Insufficient stock

Payment successful

Payment declined

Shipment created and dispatched

User cancelled order

Lifecycle of order-related events across bounded contexts.

Shared code is intentionally constrained under `libs/shared/`. The `domain/` library provides domain-neutral primitives, while `infrastructure/` provides reusable mechanisms such as the in-memory event bus and persistence utilities. This split reduces accidental coupling: domain meaning stays inside bounded contexts, and shared libraries remain a small set of stable building blocks rather than a catch-all dependency hub.

Repository-level orientation of the Tiny Store monorepo.

Architecture is enforced as executable checks. The `npm run test:boundary` target validates module dependency rules and detects coupling regressions, while `npm run test:integration` exercises cross-module flows end-to-end. Together, they provide fast feedback on both structural correctness and runtime behavior.

A practical reading order follows the enforced dependency direction: begin at `apps/api/` to see system assembly, then inspect `register-listeners.ts` to identify cross-context collaboration points. Next, enter a bounded context through its `src/index.ts` public surface and trace inward from `features/` into `domain/`. When behavior crosses contexts, follow the published events into `listeners/` and then into the next context's `features/`. This navigation strategy reflects the architectural intent: explicit contracts, visible integration, and boundaries that remain testable as the system scales.

This repository orientation establishes the concrete elements that G1 relies on: bounded contexts exposed through explicit entrypoints, cross-context collaboration expressed as event-driven contracts, and integration wiring centralized in the composition root. With this map in place, the tutorial can move from structure to action. The steps below apply G1 using Tiny Store as an executable reference. Each step states the intent and the expected signal, and when applicable it includes a runnable snippet that reproduces a boundary failure, pinpoints the violating dependency, and validates the fix through the same automated checks that enforce modular boundaries during day-to-day development.

Tutorial: Step-by-Step Application

The steps below apply G1 using Tiny Store as an executable reference. Each step states the intent and the expected signal, and when applicable it includes a runnable snippet.

Steps

- *Step 0: Establish a Clean Baseline:* Confirm that boundary enforcement is active before introducing controlled violations. Run the baseline checks, and verify that all targets pass.

  ```
  npm install
  npm test
  npm run test:boundary
  npm run test:integration
  ```

- *Step 1: Confirm Module Discovery is Unambiguous:* Verify that bounded contexts are mechanically discoverable by path conventions, enabling static boundary checks. In Tiny Store, bounded contexts live under `libs/modules/<context>/`, and shared primitives live under `libs/shared/`.
  - Do: treat `libs/modules/<context>/` as the outer boundary for G1.
  - Don't: create ad-hoc bounded contexts inside `libs/shared/` without enforcement.
- *Step 2: Treat Package-by-Feature as Internal Organization Only:* Confirm that feature slices improve cohesion inside a module while the outer boundary remains contract-based. Feature code (for example

`src/features/`) may depend on `domain/` within the same module, but must not import other modules' features or domain internals.

  - Do: allow `features/` → `domain/` dependencies inside the same module.
  - Don't: allow module *A* feature code to import module *B* feature or domain internals.

- *Step 3: Separate Public Surface from Internal Components:* Ensure cross-module usage occurs only through module entrypoints (for example `@tiny-store/modules-orders`), so internals cannot leak by import convenience.

```
/**
 * ✅ DO: import the module via its public entrypoint.
 * The public surface should expose handlers, ports, and event contracts,
 * not domain entities or repositories.
 */
import { PlaceOrderHandler } from '@tiny-store/modules-orders';
```

  The example above is correct because it imports a capability that the Orders module explicitly exposes as part of its contract. The rest of the system depends on what Orders *does* (a use case handler), not on how Orders *implements* its domain model. This is the preferred dependency shape: module-to-module dependencies point to stable entrypoints, and the set of exposed symbols remains small enough to be audited as the system grows.

```
/**
 * ❌ DON'T: entities and repositories should remain internal to the module.
 */
import { Order } from '@tiny-store/modules-orders';
// internal entity
import { OrderRepository } from '@tiny-store/modules-orders';
// internal repository
```

  The code above is incorrect because it couples other contexts to Orders' implementation details. Other modules should not construct, persist, or reason about Orders via internal entities or repositories, even if the import path appears "official." Reactions to order state changes should flow through event contracts and listeners, while requests to Orders should use a use case handler exported by the public surface. When functionality is genuinely cross-cutting, promote it to `libs/shared/` as a domain-neutral primitive instead of importing or duplicating Orders' internals. This preserves module autonomy, keeps invariants local, and prevents boundary checks from being eroded by convenience imports.

- *Step 4: Encode Explicit Declarations (Descriptor Equivalent):* Keep dependency rules explicit and reviewable by expressing *requires* and *forbids* in a single, executable source of truth (in Tiny Store, the boundary test suite plays this role).

```
/**
 * Example only: keep a single source of truth for module boundaries.
 * This can live near libs/shared/testing/src/module-boundaries.ts
 * and be imported by module-boundary.spec.ts.
 */
export const moduleBoundaries = {
  orders: {
    requires: ['shared-domain', 'shared-infrastructure'],
    forbids: ['inventory', 'payments', 'shipments'],
  },
  inventory: {
    requires: ['shared-domain', 'shared-infrastructure'],
    forbids: ['orders', 'payments', 'shipments'],
  },
} as const;
```

- *Step 5: Centralize Cross-Module Wiring in the Composition Root:* Ensure cross-module collaboration is auditable by keeping listener subscription wiring in the API composition root, rather than inside module internals.

```
/**
 * ✅ DO: keep subscriptions centralized in the composition root
 * (apps/api/src/app/lib/register-listeners.ts) so cross-module wiring is auditable.

 * The sub binds an event name (contract) to a handler (reaction).
 */
import { EventBus } from '@tiny-store/shared-infrastructure';
import { OrderPlacedListener } from '@tiny-store/modules-inventory';

const eventBus = EventBus.getInstance();
eventBus.subscribe('OrderPlaced', (event) => {
  return new OrderPlacedListener().handle(event);
});
```

```
/**
 * ❌ DON'T: wiring inside a module makes dependencies implicit and harder to audit.
 * Subscriptions should be registered in the composition root instead.
 */
EventBus.getInstance().subscribe('OrderPlaced', handler);
```

- *Step 6: Enforce Boundary Verification as a First-Class Gate:* Boundary rules only protect the architecture if they run continuously. This step treats boundary verification as a mandatory gate in the local workflow and in CI, so boundary erosion is detected before it reaches runtime. In Tiny Store, the gate is `npm run test:boundary`: it encodes the allowed dependency direction between domains and fails the build whenever a module bypasses a public surface, imports forbidden internals, or introduces

an unauthorized cross-context dependency. The key idea is to make architectural integrity measurable and non-optional, using the same feedback loop that already exists for unit and integration tests.

```
npm run test:boundary

❌ Boundary violation detected

Source project:   libs/modules/orders
Forbidden import: libs/modules/inventory/src/domain/Product.ts
Importing file:   libs/modules/orders/src/features/place-order/use-case.ts

Rule: Modules must import other modules only via their public entrypoint
      (e.g., @tiny-store/modules-inventory), not internal domain files.

Fix:  Replace internal import with an explicit contract:
      - consume a public handler, or
      - react via an event + listener, or
      - move truly shared primitives to libs/shared/.
```

The expected signal is binary: the gate either passes (no violations) or fails with a report that pinpoints the violating dependency.

In addition to the tooling-level gate, it is useful to have a focused automated test that asserts the intended public surface. The objective is not to “test architecture” by runtime imports alone, but to provide a readable specification of what must and must not be available through each module’s entry-point. The example below checks that internal entities are not exposed, while public handlers are available. This test is complementary to `test:boundary`: it documents expectations and protects against accidental re-exports in `src/index.ts`.

```
/**
 * libs/shared/testing/src/module-boundary.spec.ts
 *
 * This test asserts the module public surface (entrypoint exports).
 * It complements test:boundary by preventing accidental re-exports of internals.
 */
describe('Module Public Surface', () => {
  it('does not expose Inventory internals via the module entrypoint', async () => {
    const inventory = await import('@tiny-store/modules-inventory');

    // Internal domain types must NOT be part of the public API surface.
    expect((inventory as any).Product).toBeUndefined();
    expect((inventory as any).InventoryRepository).toBeUndefined();
  });

  it('exposes Orders public handlers via the module entrypoint', async () => {
    const orders = await import('@tiny-store/modules-orders');
```

```
    // Public capabilities are allowed and expected.
    expect((orders as any).PlaceOrderHandler).toBeDefined();
    expect((orders as any).GetOrderHandler).toBeDefined();
  });
});
```

When this gate is treated as first-class, the architectural rules stop being informal conventions and become enforced constraints. In practice, the boundary policy can be summarized as follows: external modules must not access another context's entities, repositories, or internal services; they may depend on public handlers, event contracts, and listener wiring through the composition root; and they may reuse domain-neutral primitives from `libs/shared/`. This is the mechanism that keeps G1 enforceable as the codebase scales.

- *Step 7: Add Module Isolation Checks to Detect Hidden Wiring:* Detect coupling that does not appear as imports by running module-scoped isolation tests that load a module with only its declared dependencies available.

```
/**
 * Example only: the test should fail if the Orders module tries to
 * access undeclared modules during wiring or handler execution.
 */
describe('Orders module isolation', () => {
  it('loads with declared dependencies only', async () => {
    const module = await loadModule('orders', {
      allowed: ['shared-domain', 'shared-infrastructure'],
    });
    expect(module).toBeDefined();
  });
});
```

Exercise Walkthrough: Controlled Violations and Signals

The exercises below intentionally break modular boundaries to demonstrate that enforcement is active and to make failure signals actionable. Each exercise introduces a single, controlled breach, runs the same automated gates used in day-to-day development, and observes the resulting signal. This structure serves two purposes: it confirms that the boundary rules are not merely documented, and it trains the reader to diagnose violations quickly by linking a concrete change to a predictable failure mode.

To keep the summary compact, the table uses a lightweight metric notation. $C_{\bullet}$ denotes a *count* of violations or disallowed references (higher is worse), while $P_{\bullet}$ denotes a *pass rate* (higher is better). In particular, C_{leak} counts public-surface leaks (internal symbols exposed or consumed externally), C_{undecl} counts undeclared cross-module references, C_{forbid} counts references that violate an explicit forbidden dependency rule, P_{iso}

captures the isolation pass rate when module-scoped tests are executed under declared dependencies only, and C_{event} counts boundary violations in event subscription and handling.

Exercise summary

Ex.	What you change	Run	Signal / metric
0	Establish baseline	`test:boundary` `test:integration`	pass (baseline)
1	Export internal symbols (leak)	`test:boundary`	fail, $C_{\text{leak}}\uparrow$
2	Add direct cross-module import	`test:boundary`	fail (or add rule), $C_{\text{undecl}}\uparrow$ / $C_{\text{forbid}}\uparrow$
3	Subscribe inside module (hidden wiring)	`test:boundary` `test:integration`	fail (rule/iso), $P_{\text{iso}}\downarrow$ / $C_{\text{event}}\uparrow$

Exercises

- *Exercise 0: Confirm the Baseline:* Re-run Step 0, and confirm that all checks pass.
- *Exercise 1: Create an Encapsulation Leak by Exporting an Internal Symbol:* Demonstrate that entities and repositories must not be exposed through module entrypoints. Introduce the violation by exporting internal symbols from the Orders module entry-point (for example `libs/modules/orders/src/index.ts`).

```
/**
 * ❌ VIOLATION: exporting internal domain symbols leaks encapsulation.
 */
export * from './domain/entities/order';
export * from './domain/repositories/order-repository';
```

Run enforcement.

```
npm run test:boundary
```

Expected signal: a boundary test fails, and $C_{\text{leak}}\uparrow$. Fix: remove these exports, and expose a contract instead (handler/port/event).

- *Exercise 2: Introduce a Direct Cross-Module Import (Undeclared or Forbidden):* Demonstrate that bounded contexts must not couple via direct imports for convenience. Introduce the violation by adding a direct import in an Orders feature file under `libs/modules/orders/src/features/`.

```
/**
 * ❌ VIOLATION: Orders directly imports Inventory.
 */
import { SomeInventoryHandler } from '@tiny-store/modules-inventory';
```

Run enforcement.

```
npm run test:boundary
```

Expected signal: a boundary check fails if cross-module imports are constrained by the enforcement suite. If it does not fail, treat this as an enforcement coverage gap and extend `module-boundary.spec.ts` to detect forbidden `@tiny-store/modules-*` imports from within another module. Metric impact: $C_{\text{undecl}}\uparrow$ or $C_{\text{forbid}}\uparrow$. Fix (preferred): replace the direct import with event-driven collaboration (publish `OrderPlaced`, react in Inventory, publish `InventoryReserved`).

- *Exercise 3: Hide Event Wiring Inside a Module (Implicit Dependency):* Demonstrate why subscription wiring must remain centralized and auditable. Introduce the violation by placing a subscription call inside a module file (for example under `libs/modules/<context>/src/listeners/`).

```
/**
 * ❌ VIOLATION: hidden event wiring inside a module.
 */
EventBus.getInstance().subscribe('OrderPlaced', (e) => this.handle(e));
```

Run enforcement.

```
npm run test:boundary && npm run test:integration
```

Expected signal: this may not fail immediately if the current enforcement suite does not scan for subscription calls outside the composition root. In that case, enforce a rule that forbids `eventBus.subscribe(` usage outside `apps/api/src/app/lib/register-listeners.ts`, and/or detect it

via module isolation tests. Metric impact: $P_{\text{iso}}\downarrow$ and potentially $C_{\text{event}}\uparrow$. Fix: move the subscription back to `register-listeners.ts`.

Conclusion of the G1 Implementation

This section demonstrated that enforcing modular boundaries in a modular monolith is not a matter of documentation or developer discipline alone. In Tiny Store, G1 is implemented as an executable practice: bounded contexts are exposed through explicit entry-points, cross-context collaboration is expressed through contracts (events, handlers, and listener registration), and integration wiring is kept visible in the composition root. The result is that coupling becomes observable and auditable. When a boundary is violated, the failure signal is immediate and actionable, pointing to the exact dependency that must be removed or formalized.

The walkthrough exercises reinforced the practical distinction between the right and wrong dependency shapes. The right shape is contract-based: modules depend on other modules through public handlers and event contracts, and subscriptions are registered centrally so that integration is reviewable. The wrong shape is convenience-based: importing another module's internals, leaking symbols through entry-points, or hiding wiring inside modules. These shortcuts may appear productive in the short term, but they create architectural debt by turning internal implementation details into system-wide dependencies.

Most importantly, the implementation showed how G1 remains enforceable over time. The boundary verification gate (`npm run test:boundary`), complemented by integration checks (`npm run test:integration`) and public-surface assertions, turns modularity into a measurable constraint. This makes boundary erosion difficult to introduce accidentally and inexpensive to fix when it occurs. As Tiny Store evolves, the same mechanisms scale with it: new modules can be added, contracts can be expanded deliberately, and the architecture can be kept stable without slowing down delivery. In this sense, G1 is not only a guideline but also the foundation that enables progressive scalability by keeping decomposition options open while the system remains a single deployable unit.

With boundaries enforced and coupling kept explicit, the next constraint shifts from *whether* modules can remain isolated to *how* they can be evolved safely. This transition motivates Guideline G2, which focuses on maintainability: preserving clarity of change, controlling complexity growth, and ensuring that modifications remain localized, testable, and low-risk as the codebase and team scale.

---

1. https://github.com/maurcarvalho/tiny-store↩
2. https://nx.dev/concepts/monorepos↩