DI Container Location: Why Outside the 4 Layers?

Location: `src/container.ts`
Previous Location: `src/infrastructure/di/container.ts`
Decision: Move outside the 4-layer architecture

The Question

"The DI container wires everything together. Where does it belong?"

The Answer: Nowhere (and Everywhere)

The DI container is a **bootstrap concern** that doesn't fit cleanly into any of the 4 layers:

- ❌ **Not Domain** - Domain has no dependencies, doesn't wire anything
- ❌ **Not Application** - Application defines use cases, not wiring
- ❌ **Not Infrastructure** - Infrastructure implements interfaces, doesn't coordinate layers
- ❌ **Not Presentation** - Presentation consumes dependencies, doesn't define them

The DI container touches ALL layers - it's the glue that binds them together.

Why It Was in Infrastructure (Before)

Original reasoning:

- "DI is a technical concern, so it's infrastructure"
- Infrastructure seemed like a catch-all for "technical stuff"

Problems with this:

- Infrastructure layer should **implement** interfaces, not coordinate layers
- Container imports from ALL layers (Domain, Application, Infrastructure, Presentation)
- Creates confusion: "Why does Infrastructure import from Infrastructure?"

Why It Should Be Outside (After)

Reason 1: It's a Bootstrap Concern

The container is called **once** at app startup to wire everything together:

```
// src/index.js (or App.tsx)
import "./container"; // ← Wires everything up
import { App } from "./App";

// Container has run, dependencies are registered
// Now the app can start
```

This is **bootstrap code**, like:

- Environment configuration
- Logger initialization
- Global error handlers

These don't belong in a layer—they're "setup" before layers even matter.

Reason 2: It Imports from All Layers

```
// src/container.ts
import { LifeContextRepository } from "@/application/repositories/LifeContextRepository"; // Application
import { LifeContextRepositoryImplFirebase } from "@/infrastructure/firebase/..."; // Infrastructure
import { GetLifeContextUseCase } from "@/application/useCases/lifeContext/GetLifeContext"; // Application
import { SanityClient } from "@/infrastructure/sanity/SanityClient"; // Infrastructure
```

No layer should import from all other layers. That's a sign it doesn't belong in a layer.

Reason 3: Clean Architecture Principle

From Uncle Bob's Clean Architecture:

> "The Main component is the ultimate detail—the lowest-level policy. It is the initial entry point of the system. Nothing, other than the operating system, depends on it."

The container IS the "Main" component:

- It's the entry point for dependency wiring
- Nothing depends on it (it's imported once, at startup)
- It depends on everything (to wire them together)

In Clean Architecture diagrams, "Main" sits **outside** the circles.

Comparison to Other Frameworks

Spring Boot (Java)

```
src/
├── main/
│   ├── Application.java         ← Bootstrap (like container.ts)
│   ├── domain/
│   ├── application/
│   └── infrastructure/
```

Application.java is outside the layers - it wires them together.

NestJS (TypeScript)

```
src/
├── main.ts                      ← Bootstrap (like container.ts)
├── app.module.ts                ← Wiring (like container.ts)
├── modules/
│   ├── users/
│   ├── products/
```

main.ts and app.module.ts are bootstrap concerns, not business logic.

Your App (React Native)

```
src/
├── container.ts                 ← Bootstrap / Wiring
├── old/                         ← Legacy code
├── domain/
├── application/
├── infrastructure/
└── presentation/
```

What Goes in container.ts?

✅ DO Put Here

- Repository registrations

```
container.registerSingleton<LifeContextRepository>(
  "LifeContextRepository",
  LifeContextRepositoryImplFirebase,
);
```

- Use case registrations

```
container.register<GetLifeContextUseCase>(GetLifeContextUseCase, {
  useClass: GetLifeContextUseCase,
});
```

- Service registrations

```
container.registerSingleton<AnalyticsService>(
  "AnalyticsService",
  FirebaseAnalyticsService,
);
```

❌ DON'T Put Here

- Business logic (belongs in Application/Domain)
- Repository implementations (belong in Infrastructure)
- React components (belong in Presentation)
- Configuration values (put in `config/` or env vars)

File Size

Current container.ts: ~200 lines

Is that too large?

No. It's a list of registrations. As your app grows, this file grows proportionally.

Organization strategies:

Option 1: Keep Single File (Recommended)

```
// src/container.ts
import { container } from "tsyringe";

// Domain (none - no dependencies)

// Application
import { GetLifeContextUseCase } from "@/application/useCases/...";
container.register(GetLifeContextUseCase, { useClass: GetLifeContextUseCase });

// Infrastructure
import { LifeContextRepositoryImplFirebase } from "@/infrastructure/firebase/...";
container.registerSingleton<LifeContextRepository>(
  "LifeContextRepository",
  LifeContextRepositoryImplFirebase,
);

// Export for use
export { container };
```

Pros:

- ✅ All wiring in one place
- ✅ Easy to see what's registered
- ✅ Grep-able

Option 2: Split by Layer (If Very Large)

```
src/
├── container/
│   ├── index.ts              ← Re-exports everything
│   ├── repositories.ts       ← Infrastructure registrations
│   ├── useCases.ts           ← Application use cases
│   └── services.ts           ← Application services
```

```ts
// src/container/index.ts
import "./repositories";
import "./useCases";
import "./services";

export { container } from "tsyringe";
```

Only split if container.ts exceeds ~300-400 lines.

Legacy Code in src/old/

Why Move Legacy Code?

During the migration, you'll have two architectures:

- **New:** Clean Architecture (4 layers)
- **Old:** Legacy screens, services, etc.

Problem: Mixing them creates confusion:

```
src/
├── screens/          ← Old? New? Both?
├── services/         ← Old? New? Both?
├── presentation/     ← Only new
```

Solution: Segregate the old:

```
src/
├── old/              ← Everything being phased out
│   ├── screens/
│   ├── services/
│   ├── store/
│   └── ...
├── domain/           ← New architecture
├── application/
├── infrastructure/
└── presentation/
```

What Goes in src/old/?

Move here:

- ✅ Legacy screens (screens/)
- ✅ Legacy services (services/)
- ✅ Legacy store (store/)
- ✅ Legacy hooks (hooks/)
- ✅ Legacy modules (modules/)
- ✅ Anything not following Clean Architecture

Keep outside old/:

- ✅ Domain layer (new)
- ✅ Application layer (new)
- ✅ Infrastructure layer (new)
- ✅ Presentation layer (new)
- ✅ Resources (res/, assets/)
- ✅ Config files

Migration Strategy

Phase 1: Move legacy code

```
mkdir src/old
mv src/screens src/old/
mv src/services src/old/
mv src/store src/old/
mv src/hooks src/old/
mv src/modules src/old/
```

Phase 2: Update imports

```
// Old import
import { SomeService } from "@/services/SomeService";

// New import
import { SomeService } from "@/old/services/SomeService";
```

Phase 3: Gradually migrate

- Refactor screens one-by-one
- Move to `presentation/components/` or `presentation/screens/`
- Delete from `old/` when done

Phase 4: Remove old/ folder when empty

FAQ

Q: "Won't container.ts get huge?"

A: It will grow with your app, but that's normal. 200-400 lines is fine. Split into subfiles if it exceeds 500 lines.

Q: "Should I put configuration in container.ts?"

A: No. Put config in `src/config/` or environment variables. Container just wires things up.

Q: "Can I import container from layers?"

A: Yes! Layers can resolve dependencies from container:

```
// presentation/viewModels/useLifeContext.ts
import { container } from "tsyringe";
const useCase = container.resolve(GetLifeContextUseCase);
```

Q: "Does container belong to a layer conceptually?"

A: No. It's **orthogonal** to layers—it's the mechanism that makes layers work together.

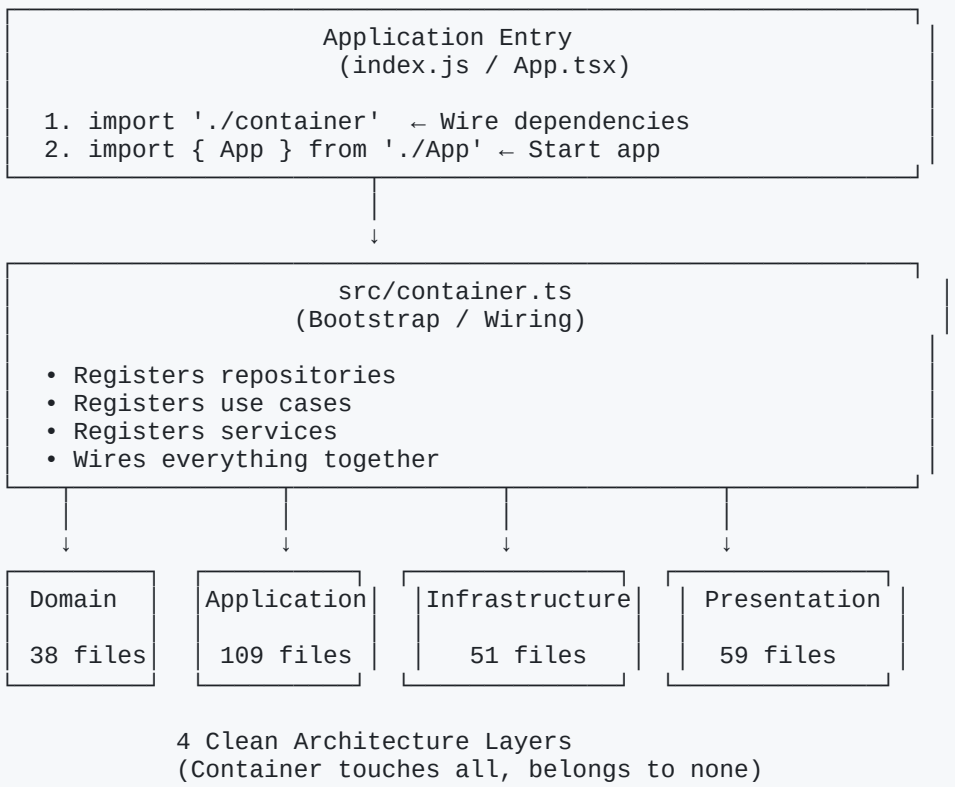
Q: "What if I'm using a framework with built-in DI?"

A: Some frameworks (NestJS, Angular) have built-in DI. In that case, you'd use their module system instead of tsyringe. But the principle is the same: wiring code sits outside business layers.

Q: "Is this 'correct' Clean Architecture?"

A: Yes. The "Main" component (container in our case) is explicitly **outside** the layer structure in Clean Architecture. See Uncle Bob's book, Chapter 21.

Visualizing the Structure



Final Structure

```
src/
├── container.ts              # Bootstrap / DI wiring (outside layers)
│
├── old/                      # Legacy code being phased out
│   ├── screens/              # Old screens
│   ├── services/             # Old services
│   ├── store/                # Old Redux
│   └── ...                   # Other legacy
│
├── domain/                   # Core business rules
├── application/              # Use cases and services
├── infrastructure/           # External service implementations
└── presentation/             # UI and state management
```

Clean separation:

- 4 layers for business logic
- Container for wiring
- Old/ for legacy (temporary)

Summary

Concern	Location	Reason
Business entities	domain/	Core business rules
Business workflows	application/	Use cases
External services	infrastructure/	Firebase, Sanity
UI & state	presentation/	React, Redux
Wiring layers together	container.ts	Bootstrap concern
Legacy code	old/	Being phased out

Key insight: The container doesn't do business logic—it just connects the pieces. It's the "glue," not the "substance."

See Also:

- Architecture Analysis Report - Full architecture details
- Quick Reference - Where to put code
- Clean Architecture (book) - Chapter 21: "The Main Component"

Last Updated: 2025-11-19