NeutronX Technical Architecture (v1.0)

This document defines the finalized technical architecture for **NeutronX**, the Flutter-first Dart backend framework with shared DTO support and monorepo-native design.

---

1. Core Principles & USP

1.1 Flutter-First Backend

NeutronX is designed to work seamlessly with Flutter applications, enabling:

- Shared DTOs and models
- Shared validation rules
- Monorepo development between `/apps/backend` and `/apps/mobile`
- Compile-time sync between backend responses and Flutter parsing

1.2 Modular Architecture

NeutronX is composed of independent units:

- **Core Runtime** (HTTP, router, middleware)
- **Plugin System** (database, auth, caching, rate-limit)
- **CLI** (`neutron`)
- **Extensions** (controllers, annotations, codegen – future)

1.3 Stability & Flexibility

- Shelf-style middleware
- Router objects + nesting (controller-ready)
- No global state
- Fully testable units

---

2. High-Level Architecture Overview

NeutronX is built on four layers:

```
+----------------------------+
|  Controller Layer (*)      |  <-- Optional, later: annotations/codegen
+----------------------------+
|  Router Layer              |  <-- Router objects, nested routers
+----------------------------+
|  Middleware Pipeline      |  <-- Shelf-style Handler/Middleware
+---------------------------+
|  Core HTTP Runtime       |  <-- dart:io server, Request/Response APIs
+---------------------------+
```

(*) Will be added in a future release without breaking changes.

---

3. Core HTTP Runtime

**3.1 Use of **dart\:io

NeutronX binds directly to `HttpServer` for:

- Maximum performance
- Full control over connection lifecycle
- Clean extension to WebSockets/upgrades in future

3.2 Request API (Finalized)

Key properties:

- `method`, `uri`, `path`
- `params`, `query`, `headers`, `cookies`
- Shared context map for middleware
- Cached body parsing

Body helpers:

- `bodyBytes()`
- `body()`
- `json()`
- `parseJson<T>()`

3.3 Response API (Finalized)

Immutable response with factories:

- `.text()`
- `.json()`
- `.bytes()`
- `.redirect()`
- `.empty()`

Supports:

- UTF-8 JSON/text
- DTO encoding for shared Flutter models
- `copyWith()` (middleware override)

3.4 Request Lifecycle

```
HttpRequest → Request.fromHttpRequest → middleware pipeline → Router.handler →
Response → HttpResponse
```

---

4. Middleware System

4.1 Shelf-Style Middleware (Final)

```
typedef Handler = Future<Response> Function(Request req);
typedef Middleware = Handler Function(Handler next);
```

4.2 Execution Flow (Onion Model)

```
logging → cors → auth → router → handler
```

4.3 Examples

- Logging Middleware
- Auth Middleware
- Error Handler Wrapper
- Rate Limiting (future plugin)

---

5. Router Architecture

5.1 Router Goals

- Modular: Each feature has its own router
- Nestable: `mount('/users', usersRouter)`
- Pattern matching for path params (`:id`)
- Future-ready for controller annotations

5.2 Router API (Final)

```
router.get('/path', handler);
router.post('/path', handler);
router.put('/path', handler);
router.delete('/path', handler);
router.patch('/path', handler);

router.mount('/prefix', anotherRouter);
```

5.3 Matching Engine

Supports:

- Static paths
- `:param` dynamic segments
- Query parsing
- Mount precedence over local routes

5.4 Router Handler Output

Router exposes a final:

```
Handler get handler;
```

Which plugs directly into the middleware chain.

---

6. Future Controller Layer (Annotation-Based)

6.1 Separation of Concerns

- Controllers live in `neutronx_controller` package
- Router remains the core mechanism
- Controller compiler generates router bindings

6.2 Example (Future)

```
@Controller('/users')
class UsersController {
  @Get('/')
  Future<List<UserDto>> list() {...}
}
```

compiled into:

```
router.get('/', wrap(controller.list));
```

6.3 Guard, Role, and Metadata Support

Router will later store metadata for:

- Roles
- Guards
- Documentation
- OpenAPI generation

---

7. Monorepo + Shared DTO Architecture

7.1 Monorepo Structure

```
root/
  apps/
    backend/        # NeutronX backend
    mobile/         # Flutter app
  packages/
    models/         # Shared DTOs (UserDto, AuthDto, etc.)
    utils/          # Shared helpers (optional)
```

7.2 Shared DTO Package

- Pure Dart package
- Used by both backend and Flutter
- Uses `json_serializable` or manual `.toJson()`

7.3 Sync Between Backend and Flutter

- Backend returns DTO from shared package → Flutter decodes same class
- Compile-time breakage when fields change → prevents runtime bugs

7.4 API Contracts

NeutronX encourages type-safe contracts by:

- Returning DTOs (`Response.json(dto.toJson())`)
- Using `parseJson<T>()` for typed input

---

8. Plugin System Architecture

8.1 Plugin Goals

- Add features without modifying core
- Act as building blocks for DB/auth/cache
- Allow community contributions

8.2 Plugin Interface (Final)

```
abstract class NeutronPlugin {
  String get name;
  Future<void> register(PluginContext ctx);
}
```

8.3 PluginContext Provides

- Root Router
- DI Container (Core)
- Config
- Logger

8.4 Official Plugin Roadmap

- Postgres (`neutronx_postgres`)
- Mongo (`neutronx_mongo`)
- Redis (`neutronx_redis`)
- JWT Auth (`neutronx_auth_jwt`)
- Rate Limit (`neutronx_rate_limit`)

---

9. Dependency Injection (DI) Architecture

9.0 Service Design Philosophy (Stateless Services – v1 Standard)

NeutronX v1 intentionally uses **application-scoped DI**. This means: - Services live for the entire life of the application. - They **cannot hold per-request state** (e.g., current user). - Authenticated user data is always stored in `Request.context` by an `AuthGuard`.

Why stateless services?

- Prevent hidden state and race conditions.
- Avoid per-request object creation overhead.
- Keep the dependency graph clean and acyclic.

- Make unit testing extremely easy.

Correct usage pattern:

```
// AuthGuard attaches user to Request.context
router.post('/cart/add', authGuard((req) async {
  final user = req.context['user'] as User;\  // from middleware
  final body = await req.json();
  return cartService.addItem(user.id, body['item']);
}));
```

Services receive context explicitly:

```
cartService.addItem(userId, item);
```

This results in predictable, testable, high-performance services.

(A future version may introduce **optional request-scoped DI**, but v1 intentionally keeps services stateless.)

---

9. Dependency Injection (DI) Architecture

9.7 DI Flow Explained (Simple + Diagrams)

To clearly understand how DI works across **services → modules → controllers → routers → app**, here are the visualizations:

9.7.1 Big Picture DI Flow

```
+-----------------------------+
|     NeutronContainer        |
|  (register + resolve DI)    |
+---------------+-------------+
                |
                v
+-----------------------------------+
|      Services (business logic)    |
+-----------------------------------+
                |
                v
+-----------------------------------+
|      Modules (feature routers)    |
+-----------------------------------+
                |
                v
+-----------------------------------+
|   Controllers (future, optional)  |
+-----------------------------------+
                |
                v
+-----------------------------------+
|           Routers                 |
+-----------------------------------+
                |
                v
+-----------------------------------+
|      NeutronApp (HTTP runtime)    |
+-----------------------------------+
```

DI always flows from top → bottom. Everything depends on what was registered in the container.

9.7.2 Bootstrapping Flow (App Startup)

```
┌──────────────────────┐
│  NeutronContainer      │
└──────────┬───────────┘
           │ register services
           v
┌──────────────────────┐
│      Services          │  e.g. UsersService
└──────────┬───────────┘
           │ injected into
           v
┌──────────────────────┐
│      Modules           │  e.g. UsersModule
└──────────┬───────────┘
           │ build router
           v
┌──────────────────────┐
│       Routers          │
└──────────┬───────────┘
           │ mounted into
           v
┌──────────────────────┐
│      NeutronApp        │
└──────────────────────┘
```

At startup:

- DI container builds all required services.
- Modules receive services via constructors.
- Modules build routers.
- Routers are mounted into the root router.
- NeutronApp assembles middleware → router → handler.

9.7.3 DI During Requests (Runtime)

```
HTTP Request
     |
     v
Middleware → Router → Handler (services already injected)
```

At runtime, **handlers never call** ``. Everything is injected ahead of time for:

- Speed
- Testability
- Predictability

9.7.4 Future Controller DI Layer

When annotation-based controllers arrive:

```
NeutronContainer
        |
        └── builds UsersController
                 |
                 └── constructor(UsersService)
```

Compiler will generate:

```
void registerUsersControllerRoutes(Router r, NeutronContainer c) {
    final controller = c.get<UsersController>();
    r.get('/', (req) => controller.list());
}
```

Controllers become DI-managed just like services.

9.7.5 DI Summary (Simple Words)

- **Container builds services.**
- **Modules receive services.**
- **Modules build routers.**
- **Routers mount into NeutronApp.**
- **Requests use the pre-built graph.**
- **Future controllers are resolved via the same container.**

This ensures a **clean, stable, testable architecture** with no hidden globals or service locators.

9.1 Goals

- Provide a simple, explicit way to manage dependencies.
- Enable easy mocking and testing of services.
- Avoid global singletons and hard-wired constructors.
- Allow plugins to contribute services into the same container.

NeutronX will ship with a lightweight but powerful DI container as part of **core**, not as a later add-on.

9.2 NeutronContainer

Core DI type:

- Stores registrations keyed by **type**.
- Supports different lifetimes:
- `registerSingleton<T>(T instance)` – pre-built singleton.
- `registerLazySingleton<T>(T Function(NeutronContainer) factory)` – built once on first `get<T>()`.
- `registerFactory<T>(T Function(NeutronContainer) factory)` – new instance per `get<T>()`.
- Resolves dependencies via `T get<T>()`.

Resolution rules:

- If a singleton exists → return it.
- If a lazy singleton factory exists → build, cache, return.
- If a factory exists → build a fresh instance.
- If nothing registered → throw a clear error.

9.3 Usage in Application Bootstrap

At app startup:

```
final container = NeutronContainer();

container.registerLazySingleton<DbClient>((c) => DbClient.fromEnv());
container.registerFactory<UsersRepository>((c) =>
UsersRepository(c.get<DbClient>()));
container.registerLazySingleton<UsersService>((c) =>
UsersService(c.get<UsersRepository>()));

final usersService = container.get<UsersService>();
final usersRouter = buildUsersRouter(usersService);
```

Handlers and routers receive fully-constructed services, so route code does **not** call the container directly (dependency injection, not service locator in hot paths).

9.4 Usage in Tests

Tests can instantiate a fresh container per test suite:

```
final container = NeutronContainer();
container.registerSingleton<UsersRepository>(FakeUsersRepository());
container.registerLazySingleton<UsersService>((c) =>
UsersService(c.get<UsersRepository>()));

final usersService = container.get<UsersService>();
// Build router/app with this service and test routes.
```

This makes it straightforward to swap real implementations for mocks without changing production code.

9.5 Request Context & Scopes

- The primary scope in v1 is **application scope** (single container for the process).
- Request-specific data (like `user`, `traceId`) should be stored in `Request.context`.
- A future version may add explicit **request scopes** (child containers per request) without breaking the core API.

9.6 Circular Dependency Handling

NeutronX DI enforces **acyclic dependency graphs**.

During resolution, `NeutronContainer` tracks the active resolution stack. If a cycle is detected:

```
A → B → C → A
```

It throws:

```
CircularDependencyError('A → B → C → A')
```

This encourages clean architectural boundaries. If a cycle occurs, you should: - Break responsibilities into smaller services. - Introduce an abstraction/interface. - Move shared logic into a third service.

Circular dependencies are never auto-resolved to avoid unpredictable startup states.

9.7 Plugin Registration Safety & Overrides

Plugins share the same DI container as the application. To prevent accidental service overwrites:

Default Behavior (Safe)

- Registering the same type twice **throws an error**.
- This protects the application from poorly written plugins.

Explicit Overrides

If a developer intentionally wants to override a plugin's service:

```
container.overrideSingleton<AuthService>(CustomAuthService());
```

This API makes intent clear and avoids silent behavior changes.

Type-Based Safety

Service registration is keyed by **type**, not by name. This minimizes collisions between unrelated plugins.

9.6 Integration with Plugin System**

`PluginContext` exposes the same DI container, so plugins can register their own services:

```
class PostgresPlugin extends NeutronPlugin {
  @override
  Future<void> register(PluginContext ctx) async {
    ctx.container.registerLazySingleton<DbClient>((c) => DbClient(ctx.config));
  }
}
```

Any route, controller, or other plugin can then resolve these types via the container at composition time.

---

10. NeutronApp (Runtime Orchestrator)

9.1 Responsibilities

- Hold root router
- Manage middleware chain
- Bind HTTP server
- Adapt Request → Response

9.2 Pipeline Assembly

Reverse fold:

```
Handler finalHandler = router.handler;
for (middleware in middlewares.reversed) {
  finalHandler = middleware(finalHandler);
}
```

9.3 Server Binding

Uses `HttpServer.bind` and pumps every request into `finalHandler`.

---

11. CLI (neutron)

10.1 CLI Capabilities (Phase 1–2)

- `neutron new app_name`
- `neutron dev` (with hot reload optional later)
- `neutron build`
- `neutron generate module users`
- `neutron generate dto User`

10.2 CLI + Monorepo Support

- Create `/packages/models`
- Generate shared DTO templates
- Sync backend + Flutter import paths

---

12. Future Extensions

11.1 Controller/Annotation Layer

- `@Controller`, `@Get`, `@Post`, `@Param`, `@Body`
- Codegen → Router registrations

11.2 OpenAPI Generator

- Generate REST docs
- Generate Flutter API clients automatically

11.3 WebSockets

- Real-time events
- Shared model events with Flutter

11.4 GraphQL Integration

Optional plugin.

---

13. Summary (Architecture Snapshot)

NeutronX core is now defined as:

- Pure `dart:io` runtime
- Request/Response finalized
- Shelf-style middleware
- Nested routers (controller-ready)
- Monorepo-first with shared DTOs
- Plugin-ready architecture
- CLI-powered developer experience

NeutronX is built to be the **official backend companion for Flutter**.