LexOrbital Module Template Guide

Complete guide for module development

LexOrbital Core Team

22 novembre 2025

Contents

LexOrbital Module Template Guide

Complete guide for the LexOrbital Module template: creation, development, testing, integration and deployment of modules compatible with the orbital station.

Table of Contents

Overview

The **LexOrbital Module Template** is a standardized template for creating modules compatible with the LexOrbital orbital station. It includes:

- **Complete configuration**: TypeScript, ESLint, Prettier, Husky, Commitlint
- **Pre-configured tests**: Vitest with code coverage
- **CI/CD**: GitHub Actions with quality gates
- **Automatic versioning**: Semantic-release based on Conventional Commits
- **Dockerfile**: Ready-to-use multi-stage containerization
- **Documentation**: Markdown structure → Pandoc (HTML/PDF/DOCX)

Documentation Organization

Technical Sheets (00–07)

The **8 numbered sheets** cover all aspects of module development:

Sheet	Title	Content
00	Quick Start	Installation, configuration, first module
01	Template Structure	File tree, files, organization
02	Module Manifest	`lexorbital.module.json` format (MANDATORY)
03	Development Rules	7 mandatory rules for integration
04	Tests and Quality	Test standards, coverage, tools

Sheet	Title	Content
05	CI/CD Workflow	GitHub Actions pipeline, quality gates
06	Semantic Versioning	Automatic Semantic Versioning
07	Core Integration	Git subtree, docking, discovery
08	Sources	References and resources
09	Manifest Validation	JSON Schema validation and testing

Additional Documentation

- **QUICKSTART.md**: Quick installation and usage guide
- **legal/sources.md**: Sources and references used

Configure the Module

1. **Update `package.json`**: name, description, author
2. **Configure `lexorbital.module.json`**: name, type, role, compatibility
3. **Customize `README.md`**
4. **Develop** your module in `src/`
5. **Test** with `pnpm test`
6. **Commit** with Conventional Commits: `git commit -m "feat: add my feature"`

Mandatory Rules (MANDATORY)

For a module to be integrated into LexOrbital, it **must** comply with:

1. **Conventional Commits** (enforced by Commitlint)
2. **Dockerfile** present and functional
3. **Mandatory tests** (min. healthcheck + functional)
4. **Complete manifest** (`lexorbital.module.json`)
5. **Complete README** with installation instructions
6. **CI compliance** (lint, type-check, test, build)
7. **TypeScript strict mode** enabled

Non-negotiable: These rules are automatically enforced by Husky, Commitlint and CI.

Contributing to Documentation

Add a New Sheet

1. Create a file `docs/NN_sheet-title.md` (NN = 2-digit number)
2. Use the header template:

```
# Sheet #N: Sheet Title {#sheet-N-title}

Summary in 2-3 sentences.

## 1. Sheet Objective

## 2. Key Concepts and Decisions

## 3. Technical Implications

## 4. Implementation Checklist
## 5. Key Takeaways

## 6. Related Links
```

3. Add an entry in the `README.md` table of contents
4. Regenerate the doc: `./scripts/generate-docs.sh`

Update an Existing Sheet

1. Edit the relevant `docs/NN_*.md` file
2. Respect the structure (numbered sections, explicit IDs)
3. Commit with Conventional Commits: `docs(sheet-N): description`
4. Automatically regenerate doc (CI/CD GitHub Actions)

License

This project is licensed under **MIT**. See the LICENSE file at the project root.

Support

For any questions or contributions, consult:

- CONTRIBUTING.md - Contribution guide
- CODE_OF_CONDUCT.md - Code of conduct
- GitHub Issues - Report a bug or propose a feature

Sheet #0: Quick Start

1. Sheet Objective

Provide a step-by-step guide to install, configure, and use the LexOrbital Module template, enabling developers to quickly create modules that comply with station standards.

2. Prerequisites

2.1. Required Software

Tool	Minimum Version	Installation
Node.js	24.11.1	nodejs.org
pnpm	Latest	`npm install -g pnpm`
Git	2.0	git-scm.com
Docker	Latest (optional)	docker.com
Docker Compose	Latest (optional)	Included with Docker Desktop

2.2. Required Knowledge

- **TypeScript**: Intermediate level
- **Node.js**: Basics of npm/pnpm and modules
- **Git**: Commits, branches, git subtree (recommended)
- **Docker**: Optional, for containerization

3. Installation

3.1. Create a Module from the Template

Option 1: Via GitHub UI (recommended)

1. Go to github.com/lexorbital/lexorbital-template-module
2. Click **"Use this template"** → **"Create a new repository"**
3. Name the new repository: `lexorbital-module-<scope>`
    - Example: `lexorbital-module-auth`, `lexorbital-module-dossiers`
4. Choose visibility (Public/Private)
5. Click **"Create repository"**

Option 2: Via CLI

```
# Clone the template
git clone https://github.com/lexorbital/lexorbital-template-module.git lexorbital-module-<scope>
```

```
cd lexorbital-module-<scope>

# Remove Git history from template
rm -rf .git
git init
git add .
git commit -m "chore: initial commit from template"

# Link to new remote repository
git remote add origin git@github.com:your-org/lexorbital-module-<scope>.git
git push -u origin main
```

3.2. Install Dependencies

```
cd lexorbital-module-<scope>
pnpm install
```

This installs:

- **TypeScript** (strict mode)
- **ESLint + Prettier** (quality gates)
- **Vitest** (testing framework)
- **Husky** (git hooks)
- **Commitlint** (conventional commits)
- **Semantic-release** (automatic versioning)

3.3. Configure the Module

Step 1: Update `package.json`

```json
{
  "name": "lexorbital-module-<scope>",
  "version": "0.1.0",
  "description": "Short description of your module",
  "repository": {
    "type": "git",
    "url": "https://github.com/your-org/lexorbital-module-<scope>"
  },
  "author": "Your Name <email@example.com>",
  "keywords": ["lexorbital", "module", "<scope>"]
}
```

Step 2: Configure `lexorbital.module.json`

```json
{
  "name": "lexorbital-module-<scope>",
  "description": "Description of your module",
  "type": "back",
  "version": "0.1.0",
  "entryPoints": {
    "main": "dist/index.js",
    "types": "dist/index.d.ts"
  },
  "lexorbital": {
    "role": "<scope>-module",
    "layer": "back",
    "compatibility": {
      "metaKernel": ">=1.0.0 <2.0.0"
    },
    "tags": ["<scope>", "your-tags"]
  },
  "env": ["ENV_VAR_1", "ENV_VAR_2"],
  "maintainer": {
    "name": "Your Name",
    "contact": "https://github.com/your-org/lexorbital-module-<scope>"
  }
}
```

Important fields:

- `type`: `"back"` (backend), `"front"` (frontend), or `"infra"` (infrastructure)
- `lexorbital.role`: Unique module identifier
- `lexorbital.layer`: Integration layer (`"back"`, `"front"`, `"infra"`)
- `env`: Required environment variables

Step 3: Update `README.md`

~~~
# LexOrbital Module - <Scope>

> Short description of the module.

## Installation

```bash
pnpm install
~~~
```

Configuration

Required environment variables:

- `ENV_VAR_1` - Description
- `ENV_VAR_2` - Description

Usage

```
import { YourService } from "lexorbital-module-<scope>"

const service = new YourService()
```

Documentation

See docs/README.md

```
## 4. Development Commands

### 4.1. Main Commands

```bash
# Development mode (watch mode with hot reload)
```

```
pnpm run dev

# Tests (Vitest)
pnpm test

# Tests in watch mode
pnpm run test:ui

# Production build
pnpm run build

# Linting
pnpm run lint

# Automatic lint + format fix
pnpm run lint:fix

# Formatting (Prettier)
pnpm run format

# TypeScript type checking
pnpm run type-check
```

4.2. Docker Commands (optional)

```
# Build Docker image
pnpm run docker:build

# Start in development mode with Docker
pnpm run docker:dev

# Start with Docker Compose (if infrastructure needed)
docker-compose -f infra/docker-compose.local.yml up
```

5. Setup Validation

Before starting development, validate that everything works:

```
# 1. Verify Git hooks are active
ls -la .husky/
# Should display: pre-commit, commit-msg

# 2. Run tests
pnpm test
#   Should display at least 1 passing test

# 3. Check lint
pnpm run lint
#   No errors

# 4. Build
pnpm run build
#   Should create dist/ folder
# 5. Test a commit (Conventional Commits)
git add .
git commit -m "test: validate setup"
#   The commitlint hook should validate the message
```

If all these steps pass, your environment is ready!

6. First Development

6.1. Starting Structure

The template provides a minimal structure:

```
tests/
   example.test.ts       # Example test (to adapt)
```

6.2. Create Your First Service

```
// src/services/my-service.ts
export class MyService {
  constructor(private readonly config: Config) {}

  async doSomething(): Promise<string> {
    // Your business logic
    return "Hello from MyService"
  }
}
```

6.3. Write the Test

```
// tests/my-service.test.ts
import { describe, it, expect } from "vitest"
import { MyService } from "../src/services/my-service"

describe("MyService", () => {
  it("should return a greeting", async () => {
    const service = new MyService({})
    const result = await service.doSomething()
    expect(result).toBe("Hello from MyService")
  })
})
```

6.4. Commit with Conventional Commits

```
git add src/services/my-service.ts tests/my-service.test.ts
git commit -m "feat: add MyService with basic functionality"
```

The `commitlint` hook will automatically validate the format.

7. Startup Checklist

- ☐ Node.js 24.11.1 installed
- ☐ pnpm installed globally
- ☐ Repository created from template
- ☐ `pnpm install` executed successfully

- ☐ `package.json` updated (name, description, author)
- ☐ `lexorbital.module.json` configured (name, type, role)
- ☐ `README.md` customized
- ☐ Tests pass (`pnpm test`)
- ☐ Lint passes (`pnpm run lint`)
- ☐ Build succeeds (`pnpm run build`)
- ☐ First commit with Conventional Commits validated

8. Next Steps

Once setup is validated, consult:

-
-
-
-

9. Troubleshooting

Error: `pnpm: command not found`

```
npm install -g pnpm
```

Error: Git hooks not triggering

```
npx husky install
chmod +x .husky/pre-commit
chmod +x .husky/commit-msg
```

Error: `commitlint` rejects my commits

Check the format: `type(scope): description`

Valid examples:

- `feat: add new feature`
- `fix: correct bug in service`
- `docs: update README`

Tests fail after installation

```
# Clean and reinstall
rm -rf node_modules pnpm-lock.yaml
pnpm install
pnpm test
```

Sheet #1: Template Structure

1. Sheet Objective

Present the complete file tree of the template, explain the role of each file and folder, and justify organizational choices to maintain consistency across all LexOrbital modules.

2. Complete File Tree

```
lexorbital-module-<scope>/
  .github/
      workflows/
          ci.yml                  # CI GitHub Actions
  .husky/
      pre-commit                  # Git hook (lint-staged)
      commit-msg                  # Git hook (commitlint)
  docs/
      README.md                   # Documentation index
      [...]                       # Other doc files
  infra/
      docker-compose.local.yml    # Local infrastructure (optional)
  src/
      index.ts                    # Main entry point
      [...]                       # Module source code
  tests/
      example.test.ts             # Tests (Vitest)
  .editorconfig                   # Editor configuration
  .gitignore                      # Git exclusions
  .prettierignore                 # Prettier exclusions
  .prettierrc                     # Prettier configuration
  CHANGELOG.md                    # Version history (auto)
  CODE_OF_CONDUCT.md              # Code of conduct
  commitlint.config.ts            # Commitlint configuration
  CONTRIBUTING.md                 # Contribution guide
  Dockerfile                      # Module Docker image
  eslint.config.cjs               # ESLint configuration (flat config)
  lexorbital.module.json          #   LexOrbital Manifest (MANDATORY)
  LICENSE                         # License (MIT recommended)
  package.json                    # npm/pnpm configuration
  pnpm-lock.yaml                  # pnpm lock file
  README.md                       # Main documentation
  SECURITY.md                     # Security policy
  SUPPORT.md                      # Support and help
  tsconfig.json                 # TypeScript configuration
  vitest.config.ts              # Vitest configuration (optional)
```

3. Main Folders

3.1. `src/` — Source Code

Role: Contains all TypeScript source code for the module.

Recommended Structure:

```
@TODO: Add recommended structure
```

Conventions:

- One file = one responsibility (single responsibility)
- Exports via `index.ts` (barrel exports)
- No code outside `src/` (except config)

3.2. `tests/` — Tests

Role: Contains all module tests (unit, integration, e2e).

Recommended Structure:

```
@TODO: Add recommended structure
```

Conventions:

- Test files: `\*.test.ts`
- Mirrors `src/` structure for unit tests
- At least **1 test** mandatory (healthcheck or functional)

3.3. `docs/` — Documentation

Role: Detailed module documentation (beyond README).

Structure:

```
docs/
  README.md               # Doc index
  00_quick-start.md       # Quick start guide
  01_template-structure.md # This sheet
  [...]                   # Other sheets
  templates/              # Pandoc templates (optional)
     custom.html
```

Conventions:

- Markdown format (`.md`)
- Numbered sheets for Pandoc generation
- README as main index

3.4. `infra/` — Infrastructure

Role: Docker Compose configuration and local infrastructure (databases, caches, etc.).

Example:

```
@TODO: Add example
```

Usage:

```
docker-compose -f infra/docker-compose.local.yml up
```

Note: Infrastructure **must not** be deployed by the module, only for local development.

3.5. `.github/workflows/` — CI/CD

Role: GitHub Actions workflows for continuous integration.

Main File: `ci.yml`

```yaml
name: CI

on:
  push:
    branches: [main, develop]
  pull_request:

jobs:
  test:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v4
      - uses: actions/setup-node@v4
        with:
          node-version: 24
      - run: corepack enable
      - run: pnpm install
      - run: pnpm run lint
      - run: pnpm run type-check
      - run: pnpm test
      - run: pnpm run build
```

Mandatory Steps:

1. Lint (ESLint)
2. Type check (tsc –noEmit)
3. Tests (Vitest)
4. Build (tsc)

3.6. `.husky/` — Git Hooks

Role: Automatic Git hooks to ensure code quality.

`pre-commit`: Lint before commit

```sh
#!/bin/sh
. "$(dirname "$0")/_/husky.sh"

npx lint-staged
```

`.lintstagedrc.json` file (lint-staged configuration):

```json
{
  "*.{ts,tsx}": ["eslint --fix", "prettier --write"],
  "*.{json,md,yml}": ["prettier --write"]
}
```

`commit-msg`: Conventional Commits Validation

```
#!/bin/sh
. "$(dirname "$0")/_/husky.sh"

npx commitlint --edit "$1"
```

4. Configuration Files

4.1. `lexorbital.module.json` (MANDATORY)

Role: Declarative manifest of the module for LexOrbital integration.

See [02_module-manifest] for complete specification.

4.2. `package.json`

Mandatory Scripts:

```
{
  "scripts": {
    "dev": "tsx watch src/index.ts",
    "build": "tsc",
    "test": "vitest run",
    "test:ui": "vitest --ui",
    "lint": "eslint .",
    "lint:fix": "eslint . --fix",
    "format": "prettier --write .",
    "type-check": "tsc --noEmit",
    "prepare": "husky install"
  }
}
```

Standard Dependencies:

- `typescript` (5.0)
- `eslint` + `@typescript-eslint/*`
- `prettier`
- `vitest`
- `husky` + `lint-staged` + `commitlint`

4.3. `tsconfig.json`

Recommended Strict Configuration:

```
{
  "compilerOptions": {
    "target": "ES2022",
    "module": "ESNext",
    "lib": ["ES2022"],
    "moduleResolution": "node",
    "outDir": "./dist",
    "rootDir": "./src",
    "strict": true,
    "esModuleInterop": true,
    "skipLibCheck": true,
    "forceConsistentCasingInFileNames": true,
    "declaration": true,
    "declarationMap": true,
    "sourceMap": true
  },
  "include": ["src/**/*"],
  "exclude": ["node_modules", "dist", "tests"]
}
```

4.4. `eslint.config.cjs` (flat config)

Modern ESLint Configuration:

```
const eslint = require("@eslint/js")
const tseslint = require("typescript-eslint")

module.exports = tseslint.config(eslint.configs.recommended, ...tseslint.configs.recommendedTypeChecked
  languageOptions: {
    parserOptions: {
      project: true,
      tsconfigRootDir: __dirname,
    },
  },
  rules: {
    "@typescript-eslint/no-unused-vars": "error",
    "@typescript-eslint/no-explicit-any": "warn",
  },
})
```

4.5. `.prettierrc`

Prettier Configuration:

```
{
  "semi": true,
  "singleQuote": true,
  "tabWidth": 2,
  "trailingComma": "es5",
  "printWidth": 100
}
```

4.6. `commitlint.config.ts`

Commitlint Configuration:

```
import type { UserConfig } from "@commitlint/types"

const config: UserConfig = {
  extends: ["@commitlint/config-conventional"],
  rules: {
    "type-enum": [2, "always", ["feat", "fix", "docs", "style", "refactor", "perf", "test", "chore", "c
  },
}

export default config
```

4.7. `Dockerfile`

Minimal Multi-Stage Dockerfile:

```dockerfile
# Stage 1: Build
FROM node:24-alpine AS builder

WORKDIR /app

# Enable pnpm
RUN corepack enable

# Copy package files
COPY package.json pnpm-lock.yaml ./

# Install dependencies
RUN pnpm install --frozen-lockfile

# Copy source
COPY . .

# Build
RUN pnpm run build

# Stage 2: Production
FROM node:24-alpine

WORKDIR /app

RUN corepack enable

COPY package.json pnpm-lock.yaml ./
RUN pnpm install --prod --frozen-lockfile

COPY --from=builder /app/dist ./dist

CMD ["node", "dist/index.js"]
```

5. Why This Structure?

5.1. Consistency

All LexOrbital modules share **exactly the same structure**:

- Facilitates navigation for developers
- Enables automation (scripts, CI)
- Guarantees interoperability with Core

5.2. Quality

Integrated tooling ensures:

- Lint-free code (ESLint + Prettier)
- Conventional commits (Commitlint)
- Automatic tests (Vitest + CI)
- Strict types (TypeScript strict mode)

5.3. Autonomy

Each module is **autonomous**:

- Independent Git repository
- Own CI/CD
- SemVer versioning
- Independent lifecycle

6. Compliance Checklist

For a module to be template-compliant:

- ☐ Manifest `lexorbital.module.json` present and valid
- ☐ `package.json` with mandatory scripts (lint, test, build)
- ☐ `tsconfig.json` with strict mode enabled
- ☐ ESLint configured (flat config)
- ☐ Prettier configured
- ☐ Husky + Commitlint installed and active
- ☐ At least 1 test (healthcheck or functional)
- ☐ Dockerfile present (multi-stage recommended)
- ☐ CI GitHub Actions configured (lint, test, build)
- ☐ README.md complete
- ☐ CHANGELOG.md automatically generated

Sheet #2: Module Manifest

The `lexorbital.module.json` file is the **mandatory manifest** for each LexOrbital module. It declares metadata, entry points, compatibility, and module dependencies.

1. Sheet Objective

Specify the exact format of the `lexorbital.module.json` manifest, its role in the LexOrbital ecosystem, and how the Core uses it to discover and validate modules.

2. Manifest Role

2.1. Why a Manifest?

The manifest allows the **Meta-Kernel** of LexOrbital Core to:

- **Discover** modules automatically (scan of `modules/*/lexorbital.module.json`)
- **Validate** compatibility (Core version, dependencies)
- **Load** modules into the correct ring (Ring)
- **Document** architecture automatically (dependency graphs)
- **Configure** required environment variables

2.2. No Manifest, No Integration

A module **without** `lexorbital.module.json`:

- Cannot be docked to the station
- Will not be detected by the Meta-Kernel
- Cannot declare its dependencies
- Cannot be automatically documented

3. Complete Manifest Structure

```
{
  "name": "lexorbital-module-<scope>",
  "description": "Short module description (1-2 sentences)",
  "type": "back",
  "version": "0.1.0",
  "entryPoints": {
    "main": "dist/index.js",
    "types": "dist/index.d.ts"
  },
  "lexorbital": {
    "role": "<scope>-module",
    "layer": "back",
    "orbit": 2,
    "compatibility": {
      "metaKernel": ">=1.0.0 <2.0.0"
    },
    "dependencies": {
      "required": ["config-module", "logger-module"],
      "optional": ["notification-module"]
    },
    "provides": {
      "services": ["MyService", "AnotherService"],
      "events": ["module.event.created", "module.event.updated"],
      "endpoints": ["/api/<scope>"]
    },
    "consumes": {
      "events": ["user.created", "user.deleted"]
    },
    "tags": ["<scope>", "backend", "database"]
  },
  "env": ["DATABASE_URL", "API_KEY"],
  "healthcheck": {
    "endpoint": "/health",
    "interval": 30000
  },
  "maintainer": {
    "name": "Your Name or Organization",
    "contact": "https://github.com/your-org/lexorbital-module-<scope>"
  },
  "repository": {
    "type": "git",
    "url": "https://github.com/your-org/lexorbital-module-<scope>"
  },
  "license": "MIT"
}
```

4. Required Fields

4.1. `name` (string, required)

Format: `lexorbital-module-<scope>`

Examples:

- `lexorbital-module-auth`
- `lexorbital-module-dossiers`
- `lexorbital-module-documents`

Rules:

- Must start with `lexorbital-module-`
- Kebab-case only (lowercase + hyphens)
- Unique in the LexOrbital ecosystem

4.2. `version` (semver, required)

Format: Semantic Versioning (`MAJOR.MINOR.PATCH`)

Examples:

- `0.1.0` (initial development)
- `1.0.0` (first stable version)
- `2.3.5` (mature version)

Rules:

- Must match version in `package.json`
- Managed automatically by `semantic-release`

4.3. `type` (enum, required)

Possible Values:

- `"back"` — Backend module (NestJS, Express, API)
- `"front"` — Frontend module (React, Vue, UI)
- `"infra"` — Infrastructure module (scripts, config, orchestration)

Example:

```json
{
  "type": "back"
}
```

4.4. `entryPoints` (object, required)

Structure:

```json
{
  "entryPoints": {
    "main": "dist/index.js",
    "types": "dist/index.d.ts"
  }
}
```

Fields:

- `main`: Compiled JavaScript file (entry point)
- `types`: TypeScript declaration files (`.d.ts`)

4.5. `lexorbital.role` (string, required)

Role: Unique module identifier in the station.

Format: `<scope>-module`

Examples:

- `auth-module`
- `dossiers-module`
- `documents-module`

4.6. `lexorbital.layer` (enum, required)

Possible Values:

- `"back"` — Backend (Meta-Kernel, services)
- `"front"` — Frontend (React, user interface)
- `"infra"` — Infrastructure (Docker, CI, scripts)

4.7. `lexorbital.compatibility` (object, required)

Structure:

```
{
  "compatibility": {
    "metaKernel": ">=1.0.0 <2.0.0"
  }
}
```

Format: pnpm range (semver)

Examples:

- `">=1.0.0 <2.0.0"` — Compatible with Meta-Kernel 1.x
- `"^1.2.0"` — Compatible with 1.2.0 and above (minor)
- `"~1.2.3"` — Compatible with 1.2.x (patches only)

5. Optional Fields

5.1. `description` (string, optional)

Role: Short module description (1-2 sentences).

Example:

```
{
  "description": "JWT authentication module with OAuth2 and RBAC support"
}
```

5.2. `lexorbital.orbit` (number, optional)

Role: Module orbital ring (0-3).

Values:

- `0` — Meta-Kernel (Core)
- `1` — Core Services (auth, audit, logs)
- `2` — Business Logic (dossiers, documents)
- `3` — Integrations (API Gateway, webhooks)

Example:

```
{
  "orbit": 1
}
```

5.3. `lexorbital.dependencies` (object, optional)

Structure:

```
{
  "dependencies": {
    "required": ["config-module", "logger-module"],
    "optional": ["notification-module"]
  }
}
```

Role: Declares inter-module dependencies.

Validation: The Meta-Kernel verifies that all `required` modules are loaded before initializing this module.

5.4. `lexorbital.provides` (object, optional)

Structure:

```json
{
  "provides": {
    "services": ["AuthService", "TokenService"],
    "events": ["user.login", "user.logout"],
    "endpoints": ["/auth/login", "/auth/logout"]
  }
}
```

Role: Declares what the module exposes (services, events, HTTP endpoints).

Usage: Automatic documentation generation and dependency graphs.

5.5. `lexorbital.consumes` (object, optional)

Structure:

```json
{
  "consumes": {
    "events": ["user.created", "user.deleted"]
  }
}
```

Role: Declares events the module listens to (pub/sub).

Usage: Maps inter-module communication flows.

5.6. `lexorbital.tags` (array, optional)

Role: Tags for search and categorization.

Example:

```json
{
  "tags": ["auth", "security", "jwt", "oauth2"]
}
```

5.7. `env` (array, optional)

Role: List of required environment variables.

Example:

```json
{
  "env": ["DATABASE_URL", "JWT_SECRET", "REDIS_URL"]
}
```

Usage: The Meta-Kernel can validate that all variables are defined at startup.

5.8. `healthcheck` (object, optional)

Structure:

```json
{
  "healthcheck": {
    "endpoint": "/health",
    "interval": 30000
  }
}
```

Role: Configures module health check.

Fields:

- `endpoint`: HTTP endpoint to poll (e.g., `/health`)
- `interval`: Interval in ms (default: 30000 = 30s)

5.9. `maintainer` (object, optional)

Structure:

```json
{
  "maintainer": {
    "name": "LexOrbital Core Team",
    "contact": "https://github.com/lexorbital/lexorbital-module-auth"
  }
}
```

5.10. `repository` (object, optional)

Structure:

```json
{
  "repository": {
    "type": "git",
    "url": "https://github.com/lexorbital/lexorbital-module-auth"
  }
}
```

5.11. `license` (string, optional)

Role: Module license.

Examples:

- `"MIT"` (recommended)
- `"Apache-2.0"`
- `"GPL-3.0"`

6. Manifest Validation

6.1. Validation on Load

The Meta-Kernel validates the manifest on each load:

```typescript
// Pseudo-code validation
function validateManifest(manifest: Manifest): void {
  // Required fields
  if (!manifest.name) throw new Error("Missing required field: name")
  if (!manifest.type) throw new Error("Missing required field: type")
  if (!manifest.version) throw new Error("Missing required field: version")

  // Name format
  if (!manifest.name.startsWith("lexorbital-module-")) {
    throw new Error('Module name must start with "lexorbital-module-"')
  }

  // SemVer version
  if (!semver.valid(manifest.version)) {
    throw new Error("Invalid version format (must be SemVer)")
  }

  // Compatibility
  if (!semver.satisfies(CORE_VERSION, manifest.lexorbital.compatibility.metaKernel)) {
    throw new Error(`Module incompatible with Meta-Kernel version ${CORE_VERSION}`)
  }

  // Dependencies
  for (const dep of manifest.lexorbital.dependencies?.required || []) {
    if (!loadedModules.has(dep)) {
      throw new Error(`Missing required dependency: ${dep}`)
    }
  }
}
```

6.2. Validation with JSON Schema

A JSON Schema is available for automatic validation:

URL: `https://lexorbital.dev/schemas/module-manifest.v1.json`

Usage in Manifest:

```json
{
  "$schema": "https://lexorbital.dev/schemas/module-manifest.v1.json",
  "name": "lexorbital-module-auth",
  ...
}
```

CLI Validation:

```
# Install @exodus/schemasafe
pnpm install --save-dev @exodus/schemasafe

# Validate manifest
npx schemasafe validate lexorbital.module.json
```

7. Complete Examples

7.1. Backend Module (Auth)

```json
{
  "$schema": "https://lexorbital.dev/schemas/module-manifest.v1.json",
  "name": "lexorbital-module-auth",
  "description": "JWT authentication module with OAuth2 and RBAC support",
  "type": "back",
  "version": "1.2.0",
  "entryPoints": {
    "main": "dist/index.js",
    "types": "dist/index.d.ts"
  },
  "lexorbital": {
    "role": "auth-module",
    "layer": "back",
    "orbit": 1,
    "compatibility": {
      "metaKernel": ">=1.0.0 <2.0.0"
    },
    "dependencies": {
      "required": ["config-module", "logger-module"],
      "optional": ["notification-module"]
    },
    "provides": {
      "services": ["AuthService", "RBACService", "TokenService"],
      "events": ["user.login", "user.logout", "token.refreshed"],
      "endpoints": ["/auth/login", "/auth/logout", "/auth/refresh"]
    },
    "consumes": {
      "events": ["user.created", "user.deleted"]
    },
    "tags": ["auth", "jwt", "oauth2", "rbac", "security"]
  },
  "env": ["JWT_SECRET", "JWT_EXPIRY", "DATABASE_URL"],
  "healthcheck": {
    "endpoint": "/auth/health",
    "interval": 30000
  },
  "maintainer": {
    "name": "LexOrbital Core Team",
    "contact": "https://github.com/lexorbital/lexorbital-module-auth"
  },
  "repository": {
    "type": "git",
    "url": "https://github.com/lexorbital/lexorbital-module-auth"
  },
  "license": "MIT"
}
```

7.2. Frontend Module (Console)

```
{
  "name": "lexorbital-module-console",
  "description": "3D control console with Three.js",
  "type": "front",
  "version": "0.5.0",
  "entryPoints": {
    "main": "dist/index.js",
    "types": "dist/index.d.ts"
  },
  "lexorbital": {
    "role": "console-module",
    "layer": "front",
    "orbit": 3,
    "compatibility": {
      "metaKernel": ">=1.0.0 <2.0.0"
    },
    "dependencies": {
      "required": ["api-gateway-module"]
    },
    "tags": ["frontend", "react", "threejs", "console", "ui"]
  },
  "license": "MIT"
}
```

7.3. Infrastructure Module

```
{
  "name": "lexorbital-module-ci-scripts",
  "description": "CI/CD scripts and automation",
  "type": "infra",
  "version": "1.0.0",
  "entryPoints": {
    "main": "dist/index.js"
  },
  "lexorbital": {
    "role": "ci-scripts",
    "layer": "infra",
    "compatibility": {
      "metaKernel": "*"
    },
    "tags": ["ci", "cd", "automation", "scripts"]
  },
  "license": "MIT"
}
```

8. Compliance Checklist

- ☐ `lexorbital.module.json` file present at root
- ☐ `name` field in format `lexorbital-module-<scope>`
- ☐ `version` field in SemVer (matches `package.json`)
- ☐ `type` field defined (`back`, `front`, or `infra`)
- ☐ `entryPoints.main` and `entryPoints.types` defined
- ☐ `lexorbital.role` unique and descriptive
- ☐ `lexorbital.layer` matches type
- ☐ `lexorbital.compatibility.metaKernel` defined
- ☐ `env` variables listed if necessary
- ☐ `healthcheck.endpoint` defined (for back modules)
- ☐ JSON Schema validation passes without errors

Sheet #3: Development Rules

1. Sheet Objective

Define strict development rules to ensure quality, consistency, and interoperability of all LexOrbital modules. No exceptions are tolerated: these rules are **mandatory** for docking.

2. The 7 Golden Rules

Rule 1: Conventional Commits

Status: MANDATORY

Description: All commits must follow the Conventional Commits specification.

Required Format

```
<type>(<scope>): <subject>

[optional body]

[optional footer]
```

Allowed Types

Type	Description	Example
`feat`	New feature	`feat: add OAuth2 support`
`fix`	Bug fix	`fix: correct token expiration`
`refactor`	Refactoring (no feat/fix)	`refactor: extract service logic`
`docs`	Documentation only	`docs: update README`
`test`	Test additions/changes	`test: add unit tests for AuthService`
`chore`	Maintenance, config, deps	`chore: update dependencies`
`perf`	Performance improvement	`perf: optimize database queries`
`ci`	CI/CD modification	`ci: add coverage report`
`revert`	Revert a commit	`revert: revert "feat: add feature X"`

Breaking Changes

For breaking changes:

```
feat!: remove deprecated API endpoint

BREAKING CHANGE: The /api/v1/old endpoint has been removed.
Migrate to /api/v2/new instead.
```

Enforcement

Conventional Commits are **enforced** by:

- **Husky** (hook `commit-msg`)
- **Commitlint** (format validation)
- **CI** (GitHub Actions)

If the commit does not respect the format, it will be **rejected**.

Rule 2: Mandatory Dockerfile

Status: MANDATORY

Description: Each module **must** include a `Dockerfile` for containerization.

Requirements

- **Multi-stage** Dockerfile (build + production)
- Official base image (`node:24-alpine` recommended)
- No secrets in image (use environment variables)
- Lightest possible image
- Non-root user (security)

Dockerfile Template

```dockerfile
# Stage 1: Build
FROM node:24-alpine AS builder

WORKDIR /app

RUN corepack enable

COPY package.json pnpm-lock.yaml ./
RUN pnpm install --frozen-lockfile

COPY . .
RUN pnpm run build

# Stage 2: Production
FROM node:24-alpine

# Non-root user
RUN addgroup -g 1001 -S appuser && \
    adduser -S -u 1001 -G appuser appuser

WORKDIR /app

RUN corepack enable

COPY package.json pnpm-lock.yaml ./
RUN pnpm install --prod --frozen-lockfile

COPY --from=builder /app/dist ./dist
```

```
# Switch to non-root user
USER appuser

CMD ["node", "dist/index.js"]
```

Validation

```
# Build must succeed
docker build -t lexorbital-module-<scope> .

# Image must start without error
docker run --rm lexorbital-module-<scope>
```

Rule 3: Mandatory Tests

Status: MANDATORY

Description: Each module must include **at minimum**:

1. **A healthcheck test** (validates the module can start)
2. **A functional test** (tests the main functionality)

Healthcheck Test (example)

```
// tests/healthcheck.test.ts
import { describe, it, expect } from "vitest"

describe("Healthcheck", () => {
  it("should return 200 OK", async () => {
    const response = await fetch("http://localhost:3000/health")
    expect(response.status).toBe(200)
  })
})
```

Functional Test (example)

```
// tests/functional.test.ts
import { describe, it, expect } from "vitest"
import { MyService } from "../src/services/my-service"

describe("MyService", () => {
  it("should perform core functionality", async () => {
    const service = new MyService()
    const result = await service.doSomething()
    expect(result).toBeDefined()
  })
})
```

Test Coverage (recommended)

- **Goal**: 80% coverage
- **Tool**: Vitest with `c8` or `@vitest/coverage-v8`

```
pnpm run coverage
```

Enforcement

- Tests executed in CI (`pnpm test`)
- Build fails if tests fail
- No merge without passing tests

Rule 4: Complete Manifest

Status: MANDATORY

Description: A **complete and valid** `lexorbital.module.json` file is mandatory.

See: [02_module-manifest] for complete specification.

Required Fields

- `name` (format `lexorbital-module-<scope>`)
- `version` (SemVer)
- `type` (`back`, `front`, `infra`)
- `entryPoints.main` and `entryPoints.types`
- `lexorbital.role`
- `lexorbital.layer`
- `lexorbital.compatibility.metaKernel`

Validation

```
# Validate with JSON Schema
npx schemasafe validate lexorbital.module.json
```

Rule 6: CI Compliance

Status: MANDATORY

Description: The module must **pass** all CI tests without error.

Mandatory CI Pipeline

The `.github/workflows/ci.yml` workflow must execute:

1. **Install**: `pnpm install`
2. **Lint**: `pnpm run lint`
3. **Type check**: `pnpm run type-check` (or `tsc --noEmit`)
4. **Tests**: `pnpm test`
5. **Build**: `pnpm run build`

Enforcement

- **No PR can be merged** if CI fails
- **No module can be docked** if CI is not green

Rule 7: TypeScript Strict Mode

Status: MANDATORY

Description: All modules must use TypeScript in **strict mode**.

Enforcement

- `pnpm run type-check` must pass without error
- No `@ts-ignore` or `@ts-expect-error` without justification
- No `any` except exceptional cases (then annotate with `// eslint-disable-line`)

---

3. Best Practices (recommended but not mandatory)

3.1. Keep Modules Focused

One module = one responsibility (Single Responsibility Principle).

Examples:

- `lexorbital-module-auth`: Authentication only
- `lexorbital-module-auth-and-files`: Mixed responsibility

3.2. Minimize Dependencies

Fewer dependencies = fewer conflict risks.

Tips:

- Use Node.js native features when possible
- Avoid large frameworks if a utility is sufficient
- Check licenses of third-party dependencies

3.3. Document All Public APIs

Each exported function/class must have JSDoc.

```
/**
 * Authenticates a user by email/password.
 * @param email - User email
 * @param password - Plain password
 * @returns JWT token if authentication succeeds
 * @throws UnauthorizedException if invalid credentials
 */
async login(email: string, password: string): Promise<string> {
  // ...
}
```

5. Compliance Checklist

For a module to be **compliant**:

- ☐ Commits follow Conventional Commits (enforced by Commitlint)
- ☐ Dockerfile present and functional (multi-stage)
- ☐ At least 1 healthcheck test + 1 functional test
- ☐ Complete and valid `lexorbital.module.json` manifest
- ☐ README.md with all mandatory sections
- ☐ CI passes without error (lint, type-check, test, build)
- ☐ TypeScript strict mode enabled
- ☐ No unjustified ESLint warnings
- ☐ Code formatted with Prettier

6. Consequences of Non-Compliance

Automatic Rejection

A module that does **not** comply with these rules will be **automatically rejected** by:

1. **Git hooks** (if Conventional Commits not respected)
2. **CI** (if tests/lint/build fail)
3. **The Meta-Kernel** (if manifest invalid)

Docking Impossibility

A non-compliant module **cannot be docked** to the LexOrbital station.

PR Blocking

PRs with failing CI are **automatically blocked** (branch protection rules).

Sheet #4: Tests and Quality

1. Sheet Objective

Define mandatory and recommended test standards for LexOrbital modules, with tools, patterns, and metrics to comply with.

2. Mandatory Tests

2.1. Healthcheck Test (MANDATORY)

Objective: Validate that the module can start and respond.

For Backend Modules (HTTP)

```
// tests/healthcheck.test.ts
import { describe, it, expect, beforeAll, afterAll } from "vitest"
import request from "supertest"
import { app } from "../src/app"

describe("Healthcheck", () => {
  let server: any

  beforeAll(async () => {
    server = app.listen(3000)
  })

  afterAll(async () => {
    server.close()
  })

  it("should return 200 OK on /health", async () => {
    const response = await request(app).get("/health")
    expect(response.status).toBe(200)
    expect(response.body).toEqual({
      status: "ok",
      timestamp: expect.any(Number),
    })
  })
})
```

For Frontend Modules (Component)

```
// tests/healthcheck.test.tsx
import { describe, it, expect } from 'vitest';
import { render, screen } from '@testing-library/react';
import { App } from '../src/App';

describe('App Component', () => {
  it('should render without crashing', () => {
    render(<App />);
    expect(screen.getByTestId('app-container')).toBeInTheDocument();
  });
});
```

For Infrastructure Modules

```
// tests/healthcheck.test.ts
import { describe, it, expect } from "vitest"
import { initModule } from "../src/index"

describe("Module Initialization", () => {
  it("should initialize without errors", async () => {
    await expect(initModule()).resolves.not.toThrow()
  })
})
```

2.2. Functional Test (MANDATORY)

Objective: Test at least **one main functionality** of the module.

Example: Authentication Module

```
// tests/functional/auth.test.ts
import { describe, it, expect, beforeEach } from "vitest"
import { AuthService } from "../src/services/auth.service"

describe("AuthService", () => {
  let authService: AuthService

  beforeEach(() => {
    authService = new AuthService({
      jwtSecret: "test-secret",
      jwtExpiry: "1h",
    })
  })

  it("should authenticate valid user", async () => {
    const token = await authService.login("user@example.com", "password123")
    expect(token).toBeDefined()
    expect(typeof token).toBe("string")
  })

  it("should reject invalid credentials", async () => {
    await expect(authService.login("user@example.com", "wrong-password")).rejects.toThrow("Invalid cred
  })
  it("should verify valid token", async () => {
    const token = await authService.login("user@example.com", "password123")
    const payload = await authService.verifyToken(token)
    expect(payload.email).toBe("user@example.com")
  })
})
```

Example: Dossier Management Module

```
// tests/functional/dossiers.test.ts
import { describe, it, expect } from "vitest"
import { DossiersService } from "../src/services/dossiers.service"

describe("DossiersService", () => {
  let service: DossiersService

  beforeEach(() => {
    service = new DossiersService()
  })

  it("should create a new dossier", async () => {
    const dossier = await service.create({
      title: "Test Dossier",
      clientId: "123",
    })

    expect(dossier.id).toBeDefined()
    expect(dossier.title).toBe("Test Dossier")
  })

  it("should retrieve dossier by ID", async () => {
    const created = await service.create({ title: "Test", clientId: "123" })
    const retrieved = await service.findById(created.id)
    expect(retrieved).toEqual(created)
  })
})
```

3. Recommended Test Types

3.1. Unit Tests

Objective: Test isolated functions/classes.

Tool: Vitest

Structure:

```
tests/
  unit/
      services/
          my-service.test.ts
      utils/
          helpers.test.ts
      models/
          user.test.ts
```

Example:

```
// tests/unit/utils/helpers.test.ts
import { describe, it, expect } from "vitest"
import { formatDate, calculateAge } from "../../src/utils/helpers"

describe("Helpers", () => {
  describe("formatDate", () => {
    it("should format date to ISO string", () => {
      const date = new Date("2025-11-22")
      expect(formatDate(date)).toBe("2025-11-22")
    })
  })

  describe("calculateAge", () => {
    it("should calculate age from birthdate", () => {
      const birthdate = new Date("2000-01-01")
      const age = calculateAge(birthdate)
      expect(age).toBeGreaterThanOrEqual(24)
    })
  })
})
```

3.2. Integration Tests

Objective: Test interaction between multiple components.

Structure:

```
tests/
   integration/
       api.test.ts
       database.test.ts
```

Example:

```
// tests/integration/api.test.ts
import { describe, it, expect, beforeAll, afterAll } from "vitest"
import request from "supertest"
import { app } from "../../src/app"
import { db } from "../../src/database"

describe("API Integration", () => {
  beforeAll(async () => {
    await db.connect()
  })

  afterAll(async () => {
    await db.disconnect()
  })

  it("should create and retrieve a user", async () => {
    // Create
    const createResponse = await request(app).post("/api/users").send({ name: "John Doe", email: "john@
    expect(createResponse.status).toBe(201)

    // Retrieve
    const userId = createResponse.body.id
    const getResponse = await request(app).get(`/api/users/${userId}`)
    expect(getResponse.status).toBe(200)
    expect(getResponse.body.name).toBe("John Doe")
  })
})
```

3.3. E2E Tests (End-to-End)

Objective: Test complete user workflows.

Tool: Playwright (frontend) or Supertest + seed data (backend)

Structure:

```
tests/
  e2e/
      auth-flow.test.ts
      dossier-workflow.test.ts
```

Example:

```
// tests/e2e/auth-flow.test.ts
import { describe, it, expect } from "vitest"
import request from "supertest"
import { app } from "../../src/app"

describe("Authentication Flow (E2E)", () => {
  it("should complete full auth flow", async () => {
    // 1. Register
    const registerResponse = await request(app).post("/auth/register").send({ email: "user@example.com"
    expect(registerResponse.status).toBe(201)

    // 2. Login
    const loginResponse = await request(app).post("/auth/login").send({ email: "user@example.com", passw
    expect(loginResponse.status).toBe(200)
    const { token } = loginResponse.body

    // 3. Access protected route
    const protectedResponse = await request(app).get("/api/profile").set("Authorization", `Bearer ${toke
    expect(protectedResponse.status).toBe(200)
    expect(protectedResponse.body.email).toBe("user@example.com")

    // 4. Logout
    const logoutResponse = await request(app).post("/auth/logout").set("Authorization", `Bearer ${token}
    expect(logoutResponse.status).toBe(200)
  })
})
```

4. Test Tools

4.1. Vitest (mandatory)

Installation:

```
pnpm add -D vitest
```

Configuration: `vitest.config.ts`

```
import { defineConfig } from "vitest/config"

export default defineConfig({
  test: {
    globals: true,
    environment: "node",
    coverage: {
      provider: "v8",
      reporter: ["text", "json", "html", "lcov"],
      exclude: ["**/node_modules/**", "**/dist/**", "**/tests/**", "**/*.config.*"],
      thresholds: {
        lines: 80,
        functions: 80,
        branches: 75,
        statements: 80,
      },
    },
  },
})
```

`package.json` Scripts:

```
{
  "scripts": {
    "test": "vitest run",
    "test:watch": "vitest",
    "test:ui": "vitest --ui",
    "coverage": "vitest run --coverage"
  }
}
```

4.2. Supertest (backend HTTP modules)

Installation:

```
pnpm add -D supertest @types/supertest
```

Usage:

```
import request from "supertest"
import { app } from "../src/app"

await request(app).get("/api/users").expect(200)
```

4.3. React Testing Library (frontend modules)

Installation:

```
pnpm add -D @testing-library/react @testing-library/jest-dom @testing-library/user-event
```

Usage:

```
import { render, screen, fireEvent } from '@testing-library/react';
import { LoginForm } from '../src/components/LoginForm';

test('should submit form', () => {
  render(<LoginForm />);
  const input = screen.getByLabelText('Email');
  fireEvent.change(input, { target: { value: 'user@example.com' } });
  // ...
});
```

5. Test Coverage

5.1. Coverage Goals

Metric	Goal	Minimum Acceptable
Lines	80%	70%
Functions	80%	70%
Branches	75%	65%
Statements	80%	70%

5.2. Generate Coverage Report

```
pnpm run coverage
```

Output:

```
---------------------------|---------|----------|---------|---------|
File                       | % Stmts | % Branch | % Funcs | % Lines |
---------------------------|---------|----------|---------|---------|
All files                  |   85.2  |   78.5   |   82.1  |   85.2  |
 src/services              |   92.3  |   85.7   |   90.0  |   92.3  |
  auth.service.ts          |   95.0  |   88.0   |   92.0  |   95.0  |
 src/utils                 |   78.5  |   70.0   |   75.0  |   78.5  |
  helpers.ts               |   78.5  |   70.0   |   75.0  |   78.5  |
---------------------------|---------|----------|---------|---------|
```

5.3. Display HTML Report

```
open coverage/index.html  # macOS
xdg-open coverage/index.html  # Linux
```

6. Test Patterns

6.1. AAA Pattern (Arrange, Act, Assert)

```
it("should calculate total price", () => {
  // Arrange
  const cart = new ShoppingCart()
  cart.addItem({ name: "Book", price: 10 })
  cart.addItem({ name: "Pen", price: 2 })

  // Act
  const total = cart.getTotal()

  // Assert
  expect(total).toBe(12)
})
```

6.2. Mocking with Vitest

```
import { describe, it, expect, vi } from "vitest"
import { EmailService } from "../src/services/email.service"
import { UserService } from "../src/services/user.service"

describe("UserService", () => {
  it("should send welcome email on user creation", async () => {
    // Mock email service
    const emailService = {
      send: vi.fn(),
    } as any

    const userService = new UserService(emailService)

    await userService.create({ email: "user@example.com", name: "John" })

    expect(emailService.send).toHaveBeenCalledWith({
      to: "user@example.com",
      subject: "Welcome!",
    })
  })
})
```

6.3. Test Fixtures

```
// tests/fixtures/users.ts
export const mockUsers = [
  { id: "1", name: "Alice", email: "alice@example.com" },
  { id: "2", name: "Bob", email: "bob@example.com" },
]

// In tests
import { mockUsers } from "../fixtures/users"

it("should list all users", () => {
  const users = service.getAll()
  expect(users).toEqual(mockUsers)
})
```

7. Quality Gates

7.1. Pre-commit

Before each commit, Husky + lint-staged check:

- Lint-free code (ESLint)
- Formatted code (Prettier)

7.2. CI Pipeline

CI executes:

1. **Lint**: `pnpm run lint`
2. **Type check**: `pnpm run type-check`

3. **Tests**: `pnpm test`
4. **Coverage**: `pnpm run coverage` (with thresholds)
5. **Build**: `pnpm run build`

7.3. Branch Protection

GitHub branch protection rules:

- CI must pass before merge
- At least 1 review required
- No force push
- Linear history

8. Test Checklist

For a module to be compliant:

☐ At least 1 healthcheck test
☐ At least 1 functional test
☐ Unit tests for main services
☐ Coverage 70% (goal 80%)
☐ All tests pass locally (`pnpm test`)
☐ Tests pass in CI
☐ No `it.skip` or `it.only` in production
☐ Fixtures/mocks properly organized

Sheet #5: CI/CD Workflow

1. Sheet Objective

Present the GitHub Actions workflow included in the template, explain each step, and show how to extend it for specific needs.

2. Minimal CI Workflow

2.1. Pipeline Steps

File: `.github/workflows/ci.yml`

Step 1: Checkout Code

```yaml
- uses: actions/checkout@v4
```

Role: Clone the Git repository into the GitHub Actions runner.

Step 2: Setup Node.js

```yaml
- uses: actions/setup-node@v4
  with:
    node-version: 24
```

Role: Install Node.js version 24 (LTS).

Matrix strategy: Allows testing on multiple Node versions (if needed).

Step 3: Enable Corepack

```yaml
- run: corepack enable
```

Role: Enable Corepack to use pnpm without global installation.

Step 4: Install Dependencies

```yaml
- run: pnpm install --frozen-lockfile
```

Role: Install exact dependencies from `pnpm-lock.yaml`.

`--frozen-lockfile`: Fails if lock file is not up to date (ensures reproducibility).

Step 5: Lint

```yaml
- run: pnpm run lint
```

Role: Check code quality with ESLint.

Fails if: ESLint errors detected.

Step 6: Type Check

```
- run: pnpm run type-check
```

Role: Check TypeScript types without generating files.

Command: `tsc --noEmit`

Fails if: Type errors detected.

Step 7: Run Tests

```
- run: pnpm test
```

Role: Execute all tests with Vitest.

Fails if: One or more tests fail.

Step 8: Build

```
- run: pnpm run build
```

Role: Compile TypeScript code to JavaScript.

Fails if: Compilation errors detected.

Step 9: Upload Coverage (optional)

```
- uses: codecov/codecov-action@v3
  with:
    files: ./coverage/coverage-final.json
    fail_ci_if_error: false
```

Role: Send coverage report to Codecov.

Optional: Can be removed if Codecov is not used.

3. Workflow Triggers

3.1. Push on Main Branches

```
on:
  push:
    branches:
      - main
      - develop
```

When: On each push to `main` or `develop`.

Objective: Ensure main branches always remain green.

3.2. Pull Requests

```
on:
  pull_request:
    branches:
      - main
      - develop
```

When: On each PR to `main` or `develop`.

Objective: Validate code before merge.

4. Branch Protection Rules

4.1. GitHub Configuration

To enforce CI, configure branch protection rules:

Settings → Branches → Branch protection rules → Add rule

Recommended Rules for `main`:

- **Require status checks to pass before merging**
  - `quality-checks` (CI job name)
- **Require branches to be up to date before merging**
- **Require approvals** (at least 1 review)
- **Dismiss stale pull request approvals when new commits are pushed**
- **Require linear history** (no merge commits)
- **Do not allow bypassing the above settings** (even for admins)

4.2. Result

With these rules:

- **Impossible to merge** a PR if CI fails
- **Impossible to push directly** to `main` without PR
- **Guarantee** that only tested and validated code reaches production

5. Possible Extensions

5.1. Security Tests (Snyk)

Test that Docker image builds correctly:

```
- name: Build Docker image
  run: docker build -t lexorbital-module-test .

- name: Test Docker image
  run: docker run --rm lexorbital-module-test node --version
```

5.4. pnpm Publication (CD)

Separate workflow for publication:

```
name: Publish

on:
  push:
    tags:
      - "v*"

jobs:
  publish:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v4
      - uses: actions/setup-node@v4
        with:
          node-version: 24
          registry-url: "https://registry.npmjs.org"

      - run: corepack enable
      - run: pnpm install --frozen-lockfile
      - run: pnpm run build

      - name: Publish to pnpm
        run: pnpm publish --no-git-checks
        env:
          NODE_AUTH_TOKEN: ${{ secrets.NPM_TOKEN }}
```

5.5. Semantic Release

Automate versioning and changelogs:

```
- name: Semantic Release
  run: npx semantic-release
  env:
    GITHUB_TOKEN: ${{ secrets.GITHUB_TOKEN }}
    NPM_TOKEN: ${{ secrets.NPM_TOKEN }}
```

6. Deployment (CD)

6.1. Important Rule

Deployment (CD) MUST NOT be added to individual modules.

Reason: Modules are deployed **via the LexOrbital station** (`lexorbital-stack`), not individually.

6.2. Deployment Workflow

Deployment happens at the `lexorbital-core` level:

1. **Module pushed** → Module repo (CI passes)
2. **Subtree update** → `lexorbital-core` pulls updated module
3. **Deploy station** → `lexorbital-stack` deploys entire station

Consequence: Modules only need **CI**, not **CD**.

7. CI Monitoring

7.1. Status Badges

Add a GitHub Actions badge to README:

```
[![CI](https://github.com/your-org/lexorbital-module-<scope>/actions/workflows/ci.yml/badge.svg)](https
```

7.2. Notifications

Configure Slack/Discord notifications:

```
- name: Notify on failure
  if: failure()
  uses: 8398a7/action-slack@v3
  with:
    status: ${{ job.status }}
    text: "CI failed for ${{ github.repository }}"
  env:
    SLACK_WEBHOOK_URL: ${{ secrets.SLACK_WEBHOOK }}
```

8. Optimizations

8.1. pnpm Cache

Speed up dependency installation:

```
- name: Setup pnpm
  uses: pnpm/action-setup@v2
  with:
    version: 8

- name: Get pnpm store directory
  id: pnpm-cache
  run: echo "pnpm_cache_dir=$(pnpm store path)" >> $GITHUB_OUTPUT

- name: Cache pnpm modules
  uses: actions/cache@v3
  with:
    path: ${{ steps.pnpm-cache.outputs.pnpm_cache_dir }}
    key: ${{ runner.os }}-pnpm-${{ hashFiles('**/pnpm-lock.yaml') }}
    restore-keys: |
      ${{ runner.os }}-pnpm-
```

8.2. Parallel Jobs

Run lint and tests in parallel:

```
jobs:
  lint:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v4
      - run: corepack enable
      - run: pnpm install --frozen-lockfile
      - run: pnpm run lint

  test:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v4
      - run: corepack enable
      - run: pnpm install --frozen-lockfile
      - run: pnpm test
```

9. CI Checklist

For a module to be compliant:

- ☐ Workflow `.github/workflows/ci.yml` present
- ☐ Pipeline executes: install, lint, type-check, test, build
- ☐ CI runs on push and PR
- ☐ All steps pass without error
- ☐ Branch protection rules configured
- ☐ Status badge in README (optional)

10. Troubleshooting

Error: `pnpm: command not found`

Solution: Add `corepack enable` before `pnpm install`.

Error: Lock File Out of Date

Solution: Update lock file locally and commit:

```
pnpm install
git add pnpm-lock.yaml
git commit -m "chore: update lock file"
```

Tests Fail in CI but Pass Locally

Possible Causes:

- Missing environment variables
- Timezone difference (UTC in CI vs local)
- Non-reproducible dependencies

Solution: Use `--frozen-lockfile` and define variables in GitHub Secrets.

Sheet #6: Semantic Versioning (SemVer)

Automatic versioning policy based on Semantic Versioning and Conventional Commits, ensuring consistent versions and automatically generated changelogs.

1. Sheet Objective

Explain the automatic versioning system for LexOrbital modules, based on SemVer and powered by Conventional Commits, with automatic changelog generation.

2. Semantic Versioning (SemVer)

2.1. Format

Format: `MAJOR.MINOR.PATCH`

Examples:

- `0.1.0` — Initial development
- `1.0.0` — First stable version
- `1.2.5` — Mature version
- `2.0.0` — Breaking change

2.2. Bump Rules

Version	When to Bump	Example
PATCH	Bug fixes, optimizations	`1.0.0` → `1.0.1`
MINOR	New features (backward compatible)	`1.0.1` → `1.1.0`
MAJOR	Breaking changes (non backward compatible)	`1.1.0` → `2.0.0`

2.3. Pre-releases

For development versions:

- `0.1.0-alpha.1` — Alpha (unstable)
- `0.1.0-beta.1` — Beta (complete features, testing in progress)
- `0.1.0-rc.1` — Release Candidate (ready for release)

3. Conventional Commits → SemVer

3.1. Automatic Mapping

Commit Type	SemVer Impact	Example
`fix:`	**PATCH**	`1.0.0` → `1.0.1`
`feat:`	**MINOR**	`1.0.0` → `1.1.0`
`feat!:` or `BREAKING CHANGE:`	**MAJOR**	`1.0.0` → `2.0.0`
`chore:`, `docs:`, `refactor:`, `test:`	**None**	No release

3.2. Concrete Examples

PATCH (fix)

```
git commit -m "fix: correct token expiration bug"
```

Result: `1.0.0` → `1.0.1`

CHANGELOG:

```
## [1.0.1] - 2025-11-22

### Bug Fixes

- correct token expiration bug ([abc123](https://github.com/...))
```

MINOR (feat)

```
git commit -m "feat: add OAuth2 Google provider"
```

Result: `1.0.1` → `1.1.0`

CHANGELOG:

```
## [1.1.0] - 2025-11-22

### Features

- add OAuth2 Google provider ([def456](https://github.com/...))
```

MAJOR (breaking change)

```
git commit -m "feat!: change AuthService.login() signature

BREAKING CHANGE: login() now requires { email, password } object instead of separate parameters.

Migration: Replace authService.login(email, password) with authService.login({ email, password })"
```

Result: `1.1.0` → `2.0.0`

CHANGELOG:

```
## [2.0.0] - 2025-11-22

###   BREAKING CHANGES

- change AuthService.login() signature
```

```
login() now requires { email, password } object instead of separate parameters.

**Migration:** Replace `authService.login(email, password)` with `authService.login({ email, password }

### Features

- change AuthService.login() signature ([ghi789](https://github.com/...))
```

4. Semantic-release

4.1. Installation

The template already includes `semantic-release`:

```
{
  "devDependencies": {
    "semantic-release": "^22.0.0",
    "@semantic-release/changelog": "^6.0.3",
    "@semantic-release/git": "^10.0.1"
  }
}
```

4.2. Configuration: `.releaserc.json`

```
{
  "branches": ["main"],
  "plugins": [
    "@semantic-release/commit-analyzer",
    "@semantic-release/release-notes-generator",
    "@semantic-release/changelog",
    [
      "@semantic-release/npm",
      {
        "npmPublish": false
      }
    ],
    [
      "@semantic-release/git",
      {
        "assets": ["CHANGELOG.md", "package.json"],
        "message": "chore(release): ${nextRelease.version} [skip ci]\n\n${nextRelease.notes}"
      }
    ],
    "@semantic-release/github"
  ]
}
```

4.3. Plugins

Plugin	Role
`commit-analyzer`	Analyzes commits to determine release type
`release-notes-generator`	Generates release notes from commits
`changelog`	Updates `CHANGELOG.md`

Plugin	Role
`npm`	Publishes to npm (disabled by default with `npmPublish: false`)
`git`	Commits changes (CHANGELOG, package.json) and creates a tag
`github`	Creates a GitHub Release

5. Automatic Workflow

5.1. Release on Push (main)

GitHub Actions Workflow: `.github/workflows/release.yml`

```yaml
name: Release

on:
  push:
    branches:
      - main

jobs:
  release:
    name: Semantic Release
    runs-on: ubuntu-latest

    steps:
      - name: Checkout
        uses: actions/checkout@v4
        with:
          fetch-depth: 0
          persist-credentials: false

      - name: Setup Node.js
        uses: actions/setup-node@v4
        with:
          node-version: 24

      - name: Enable Corepack
        run: corepack enable

      - name: Install dependencies
        run: pnpm install --frozen-lockfile

      - name: Build
        run: pnpm run build

      - name: Semantic Release
        run: npx semantic-release
        env:
          GITHUB_TOKEN: ${{ secrets.GITHUB_TOKEN }}
          NPM_TOKEN: ${{ secrets.NPM_TOKEN }}
```

5.2. Automatic Process

1. **Push to `main`** → `release` workflow triggers

2. **Semantic-release** analyzes commits since last release
3. **Version determination**:
   - `fix:` commits → PATCH
   - `feat:` commits → MINOR
   - Commits with `BREAKING CHANGE` → MAJOR
4. **CHANGELOG generation** (adds new commits)
5. **Version bump** in `package.json`
6. **Commit** changes: `chore(release): X.Y.Z [skip ci]`
7. **Tag creation** Git: `vX.Y.Z`
8. **Push** commit + tag
9. **GitHub Release creation** with release notes

5.3. Result

After a push with `feat: add new feature`:

- `package.json`: `"version": "1.1.0"`
- `CHANGELOG.md` updated with new version
- Git tag: `v1.1.0`
- GitHub Release created automatically

6. Automatic CHANGELOG

6.1. Structure

The `CHANGELOG.md` is automatically generated:

```markdown
# Changelog

All notable changes to this project will be documented in this file.

## [1.1.0] - 2025-11-22

### Features

- add OAuth2 Google provider ([abc123](https://github.com/.../commit/abc123))
- add 2FA support ([def456](https://github.com/.../commit/def456))

### Bug Fixes

- correct token refresh logic ([ghi789](https://github.com/.../commit/ghi789))

## [1.0.1] - 2025-11-20

### Bug Fixes

- fix password hashing issue ([jkl012](https://github.com/.../commit/jkl012))

## [1.0.0] - 2025-11-15

### Features

- initial release with JWT authentication
```

6.2. Commit Groups

Commits are automatically grouped by type:

- **BREAKING CHANGES** — Breaking changes
- **Features** — New features (`feat:`)
- **Bug Fixes** — Fixes (`fix:`)
- **Performance Improvements** — Optimizations (`perf:`)
- **Reverts** — Commit reverts (`revert:`)

7. Manual Release

7.1. Dry Run (simulation)

Simulate a release without publishing:

```
npx semantic-release --dry-run
```

Output:

```
[semantic-release] ›   Start step "analyzeCommits" of plugin "@semantic-release/commit-analyzer"
[semantic-release] ›   Analyzing commit: feat: add new feature
[semantic-release] ›   Completed step "analyzeCommits" of plugin "@semantic-release/commit-analyzer"
[semantic-release] ›   The next release version is 1.1.0
```

7.2. Local Release

Create a release manually (if not in GitHub Actions):

```
npx semantic-release --no-ci
```

Warning: Requires `GITHUB_TOKEN` and `NPM_TOKEN` environment tokens.

8. Branch Management

8.1. Release Branches

By default, semantic-release only releases from `main`:

```
{
  "branches": ["main"]
}
```

8.2. Multi-branches (optional)

To support multiple release channels:

```
{
  "branches": [
    "main",
    {
      "name": "next",
      "prerelease": true
    },
    {
      "name": "beta",
      "prerelease": true
    }
  ]
}
```

Result:

- Push to `main` → `1.0.0`
- Push to `next` → `1.1.0-next.1`
- Push to `beta` → `1.1.0-beta.1`

9. npm Publication (optional)

9.1. Enable Publication

In `.releaserc.json`, remove `"npmPublish": false`:

```json
{
  "plugins": ["@semantic-release/npm"]
}
```

9.2. npm Token

Add the `NPM_TOKEN` secret in GitHub:

Settings → Secrets → Actions → New repository secret

Name: `NPM_TOKEN`
Value: Your npm token (generated on npmjs.com)

9.3. npm Scope

To publish a scoped package (e.g., `@lexorbital/module-auth`):

`package.json`:

```json
{
  "name": "@lexorbital/module-auth",
  "publishConfig": {
    "access": "public"
  }
}
```

10. Best Practices

10.1. Squash Commits

Squash PR commits into a single commit:

```bash
git rebase -i HEAD~5
# Squash all commits except the first
# Edit final message to respect Conventional Commits
```

10.2. Clear Commit Messages

Bad:

```
fix stuff
update code
wip
```

Good:

```
fix: correct token expiration calculation
feat: add support for refresh tokens
refactor: extract auth logic into service
```

10.3. Manual Changelog (if necessary)

If manual notes are needed in CHANGELOG:

```markdown
## [1.0.0] - 2025-11-22

### Migration Guide

This is a major release with breaking changes. Please follow this guide:

1. Update all imports from `auth` to `@lexorbital/auth`
2. Replace `login(email, password)` with `login({ email, password })`
3. Update environment variables (see README)

## 11. Versioning Checklist

- [ ] All commits follow Conventional Commits
- [ ] `semantic-release` configured in `.releaserc.json`
- [ ] GitHub Actions `release.yml` workflow present
- [ ] `GITHUB_TOKEN` secret available (automatic)
- [ ] `NPM_TOKEN` secret configured (if npm publication)
- [ ] CHANGELOG.md automatically generated
- [ ] Git tags automatically created
- [ ] GitHub Releases automatically created
```

Sheet #7: Integration with LexOrbital Core

How modules are discovered, validated, and docked to the LexOrbital Core station via **git subtree**, while maintaining their autonomy and independent repository.

1. Sheet Objective

Explain the module integration process into `lexorbital-core`, the use of git subtree, autonomy rules, and the update workflow.

2. Integration Principle

2.1. Autonomous Modules, Unified Station

Concept: Each module is an **independent Git repository**, but all are **integrated** into `lexorbital-core` to form the complete station.

```
lexorbital-module-auth       ← Independent repository
lexorbital-module-dossiers   ← Independent repository
lexorbital-module-documents  ← Independent repository
           ↓
   lexorbital-core
         modules/
             auth/         ← git subtree of lexorbital-module-auth
             dossiers/     ← git subtree of lexorbital-module-dossiers
             documents/    ← git subtree of lexorbital-module-documents
```

2.2. Why git subtree?

Advantages:

- **Transparency**: Module code is physically present in `lexorbital-core`
- **Simple clone**: `git clone lexorbital-core` is sufficient, no submodules
- **Preserved history**: Module history is merged into Core
- **Autonomy**: Module can evolve independently
- **Contribution**: Easy to push changes upstream

Vs git submodule:

Criterion	git subtree	git submodule
Clone	Simple (`git clone`)	Complex (`--recurse-submodules`)
Code Present	Yes	No (reference only)

Criterion	git subtree	git submodule
Detached State	No	Yes (detached HEAD)
Contribution	Easy (`subtree push`)	Complex

3. Module Docking (Initial Docking)

3.1. git subtree add Command

From the `lexorbital-core` repository:

```
git subtree add \
  --prefix=modules/auth \
  git@github.com:lexorbital/lexorbital-module-auth.git \
  main \
  --squash
```

Parameters:

- `--prefix=modules/auth`: Where to place the module in the monorepo
- `git@github.com:...`: Module repository URL
- `main`: Branch to integrate
- `--squash`: Merge history into a single commit (recommended)

3.2. Docking Script

The Core provides a script to simplify:

`lexorbital-core/scripts/add-module.sh`

```
#!/bin/bash
set -e

MODULE_NAME=$1
MODULE_REPO=$2
MODULE_BRANCH=${3:-main}

if [ -z "$MODULE_NAME" ] || [ -z "$MODULE_REPO" ]; then
  echo "Usage: ./scripts/add-module.sh <module-name> <repo-url> [branch]"
  echo "Example: ./scripts/add-module.sh auth git@github.com:lexorbital/lexorbital-module-auth.git"
  exit 1
fi

echo "   Docking module ${MODULE_NAME}..."

git remote add -f "${MODULE_NAME}-remote" "$MODULE_REPO"
git subtree add --prefix="modules/${MODULE_NAME}" "${MODULE_NAME}-remote" "$MODULE_BRANCH" --squash
git remote remove "${MODULE_NAME}-remote"

echo "  Module ${MODULE_NAME} docked successfully in modules/${MODULE_NAME}"
```

Usage:

```
cd lexorbital-core
./scripts/add-module.sh auth git@github.com:lexorbital/lexorbital-module-auth.git
```

3.3. Post-Docking Verification

After docking, verify:

```bash
# 1. Module is present
ls -la modules/auth/

# 2. Manifest is valid
cat modules/auth/lexorbital.module.json

# 3. Meta-Kernel detects the module
pnpm run start:dev
# Logs: "  Module auth-module loaded successfully"
```

4. Module Update (Pull Upstream)

4.1. git subtree pull Command

From `lexorbital-core`, to update a module:

```bash
git subtree pull \
  --prefix=modules/auth \
  git@github.com:lexorbital/lexorbital-module-auth.git \
  main \
  --squash
```

4.2. Update Script

`lexorbital-core/scripts/update-module.sh`

```bash
#!/bin/bash
set -e

MODULE_NAME=$1
MODULE_REPO=$2
MODULE_BRANCH=${3:-main}

if [ -z "$MODULE_NAME" ] || [ -z "$MODULE_REPO" ]; then
  echo "Usage: ./scripts/update-module.sh <module-name> <repo-url> [branch]"
  exit 1
fi

echo "  Updating module ${MODULE_NAME}..."

git remote add -f "${MODULE_NAME}-remote" "$MODULE_REPO" 2>/dev/null || true
git subtree pull --prefix="modules/${MODULE_NAME}" "${MODULE_NAME}-remote" "$MODULE_BRANCH" --squash
git remote remove "${MODULE_NAME}-remote" 2>/dev/null || true

echo "  Module ${MODULE_NAME} updated successfully"
```

Usage:

```
./scripts/update-module.sh auth git@github.com:lexorbital/lexorbital-module-auth.git
```

4.3. Conflict Resolution

If conflicts occur during pull:

```bash
# 1. Manually resolve conflicts
git status
# Conflicted files listed

# 2. Edit files to resolve conflicts
code modules/auth/src/service.ts

# 3. Mark as resolved
git add modules/auth/src/service.ts

# 4. Continue merge
git commit
```

5. Contributing to a Module (Push Upstream)

5.1. Golden Rule: Do NOT Modify in lexorbital-core

IMPORTANT: Modifications **must** be made in the module repository, **not** in `lexorbital-core/modules/`.

Reason:

- Maintain module autonomy
- Preserve module Git history
- Avoid desynchronization

5.2. Recommended Workflow

Step 1: Clone Module Separately

```bash
git clone git@github.com:lexorbital/lexorbital-module-auth.git
cd lexorbital-module-auth
```

Step 2: Develop in Module

```bash
# Create feature branch
git checkout -b feat/add-oauth2

# Develop
# ... modifications ...

# Commit (Conventional Commits)
git add .
git commit -m "feat: add OAuth2 Google provider"

# Tests
pnpm test

# Push
git push origin feat/add-oauth2
```

Step 3: Pull Request

Create a PR on the module repository (`lexorbital-module-auth`).

Step 4: Merge

Once PR is merged into module `main`.

Step 5: Update Core

```
cd lexorbital-core
./scripts/update-module.sh auth git@github.com:lexorbital/lexorbital-module-auth.git
```

5.3. Exceptional Case: git subtree push

If a modification **absolutely must** be made in `lexorbital-core` (critical hotfix), we can push to the module:

```
git subtree push \
  --prefix=modules/auth \
  git@github.com:lexorbital/lexorbital-module-auth.git \
  hotfix-branch
```

Warning: This command can be **very slow** (it filters entire history).

Alternative Workflow (faster):

```
# 1. Extract subdirectory
git subtree split --prefix=modules/auth -b temp-auth-branch

# 2. Push to module
git push git@github.com:lexorbital/lexorbital-module-auth.git temp-auth-branch:hotfix

# 3. Cleanup
git branch -D temp-auth-branch
```

6. Module Discovery by Meta-Kernel

6.1. Automatic Scan

On startup, the Meta-Kernel scans all subdirectories of `modules/`:

```
// meta-kernel/src/core/module-loader.service.ts
async discoverModules(): Promise<ModuleManifest[]> {
  const modulesDir = path.join(__dirname, '../../../modules');
  const modules: ModuleManifest[] = [];

  const dirs = await fs.readdir(modulesDir, { withFileTypes: true });

  for (const dir of dirs) {
    if (!dir.isDirectory()) continue;

    const manifestPath = path.join(modulesDir, dir.name, 'lexorbital.module.json');

    if (await fs.pathExists(manifestPath)) {
      const manifest = await fs.readJSON(manifestPath);
      modules.push(manifest);
    }
  }
  return modules;
}
```

6.2. Manifest Validation

For each discovered module, the Meta-Kernel validates:

```
validateManifest(manifest: ModuleManifest): void {
  // 1. Required fields
  if (!manifest.name) throw new Error('Missing field: name');
  if (!manifest.version) throw new Error('Missing field: version');
  if (!manifest.type) throw new Error('Missing field: type');

  // 2. Name format
  if (!manifest.name.startsWith('lexorbital-module-')) {
    throw new Error('Module name must start with "lexorbital-module-"');
  }

  // 3. Compatibility
  if (!semver.satisfies(CORE_VERSION, manifest.lexorbital.compatibility.metaKernel)) {
    throw new Error(`Module incompatible with Core version ${CORE_VERSION}`);
  }

  // 4. Dependencies
  for (const dep of manifest.lexorbital.dependencies?.required || []) {
    if (!this.loadedModules.has(dep)) {
      throw new Error(`Missing required dependency: ${dep}`);
    }
  }
}
```

6.3. Dynamic Loading

Once validated, the module is loaded:

```
async loadModule(manifestPath: string): Promise<void> {
  const manifest = await fs.readJSON(manifestPath);
  this.validateManifest(manifest);

  // Dynamic loading (ESM)
  const modulePath = path.dirname(manifestPath);
  const entryPoint = path.join(modulePath, manifest.entryPoints.main);

  const moduleExports = await import(entryPoint);

  // Registration
  this.loadedModules.set(manifest.lexorbital.role, {
    manifest,
    exports: moduleExports,
  });

  console.log(` Module ${manifest.lexorbital.role} loaded successfully`);
}
```

7. Module Undocking

7.1. Command

```
git rm -r modules/auth
git commit -m "chore: remove auth module"
```

7.2. Undocking Script

`lexorbital-core/scripts/remove-module.sh`

```
#!/bin/bash
set -e

MODULE_NAME=$1

if [ -z "$MODULE_NAME" ]; then
  echo "Usage: ./scripts/remove-module.sh <module-name>"
  exit 1
fi

echo "  Undocking module ${MODULE_NAME}..."

git rm -r "modules/${MODULE_NAME}"
git commit -m "chore(modules): remove ${MODULE_NAME}"

echo "  Module ${MODULE_NAME} undocked successfully"
```

7.3. Pre-Undocking Checks

- ☐ No other module depends on it (check manifest `dependencies`)
- ☐ Remove module references in Core config
- ☐ Test that station starts without the module

8. Module Replacement

8.1. Use Case

Replace a module with a new implementation (same interface, different implementation).

Example: Replace `auth-module-jwt` with `auth-module-oauth`.

8.2. Workflow

```
# 1. Undock old module
./scripts/remove-module.sh auth-jwt

# 2. Dock new module
./scripts/add-module.sh auth-oauth git@github.com:lexorbital/lexorbital-module-auth-oauth.git

# 3. Update config (if necessary)
# Edit lexorbital-core/.env or config files

# 4. Test
pnpm run start:dev
```

9. Integration Checklist

For a module to be docked:

- ☐ Valid `lexorbital.module.json` manifest
- ☐ Version compatible with Meta-Kernel
- ☐ Required dependencies already loaded
- ☐ Tests pass in module (CI green)
- ☐ Documentation up to date
- ☐ Functional Dockerfile
- ☐ Module docked via `git subtree add`
- ☐ Meta-Kernel detects and loads module without error

Sources and References

This document lists the sources consulted and references used for the creation and development of the LexOrbital Core repository as a whole.

Architecture and Design

Architectural Patterns

- **Modular Architecture Patterns** : Inspiration for the modular system design
- **Microservices Architecture** : Principles applied to module orchestration
- **Plugin Architecture** : Design pattern used for module integration

Orbital Metaphor

- @TODO: Add orbital metaphor

Technologies and Frameworks

- @TODO: Add technologies and frameworks

Infrastructure

- **Docker** : Containerization
  - Official documentation : https://docs.docker.com/
- **Docker Compose** : Container orchestration
  - Official documentation : https://docs.docker.com/compose/

Development Tools

Package Management

- **pnpm** : Package manager
  - Official documentation : https://pnpm.io/

Linting and Formatting

- **ESLint** : JavaScript/TypeScript linter
  - Official documentation : https://eslint.org/
- **Commitlint** : Commit message validation
  - Official documentation : https://commitlint.js.org/

Git

- **Git Subtree** : Module integration
  - Documentation : https://git-scm.com/book/en/v2/Git-Tools-Subtree-Merging

Standards and Conventions

Conventional Commits

- Specification : https://www.conventionalcommits.org/
- Used for commit message standardization

Documentation

- @TODO: Add documentation standards

Licenses and Compliance

Open Source Licenses

- **MIT License** : Recommended license for modules
  - Full text : https://opensource.org/licenses/MIT

Security

- Security best practices for open source projects
- Responsible vulnerability disclosure guidelines

Privacy and Data Protection

- @TODO: Add privacy and data protection best practices

Inspiration and Philosophy

Design Principles

- **Minimalism** : Minimal and essential architecture
- **Modularity** : System composed of autonomous and replaceable modules
- **Security** : Priority on security and compliance
- **Elegance** : Clean and maintainable code

Key Concepts

- **Vessels** : Modules designed as autonomous vessels
- **Orbital System** : Architecture based on an orbital system
- **Law-Driven Core** : Core guided by contracts and rules

Notes

This list is non-exhaustive and will be updated as the project develops. The sources mentioned have served as references for the design, implementation, and documentation of the LexOrbital Template Module system.

Appendix: Module Manifest Validation

Overview

The manifest validation system ensures that your LexOrbital module is properly configured and compliant with ecosystem standards.

Validation Components

1. JSON Schema (`schemas/module-manifest.schema.json`)

A complete JSON Schema that defines:

- **Required** and optional fields
- **Formats** and patterns (semver, URLs, etc.)
- **Enumerations** for types and layers
- **Custom validation** rules
- **Constraints** on default values

2. Automated Tests (`tests/module-manifest.test.ts`)

Test suite that verifies:

- Compliance with JSON schema
- Template customization (no default values)
- Presence of all required fields
- Format validity (semver, URLs)
- Type and layer consistency
- Tag uniqueness

3. Documentation (`schemas/README.md`)

Complete guide on:

- Schema usage
- Validation rules
- Available types and enumerations
- Examples and use cases

Usage

Quick Validation

```bash
# Validate your manifest
pnpm test:manifest

# Detailed validation with error messages
pnpm validate:manifest

# All tests (including manifest)
pnpm test:all
```

IDE Integration

The schema is automatically recognized by modern IDEs thanks to the `$schema` property in `lexorbital.module.json`:

```json
{
  "$schema": "./schemas/module-manifest.schema.json",
  ...
}
```

IDE Features:

- **Autocompletion** of fields
- **IntelliSense** with documentation
- **Real-time validation**
- **Tooltips** with descriptions

CI/CD Validation

Validation tests are automatically executed in the CI/CD pipeline to ensure that:

1. The manifest is valid
2. All required fields are present
3. The template has been customized
4. Formats are correct

Customization Checklist

Before committing your module, make sure you have customized:

Basic Information

- ☐ `name`: Changed from `lexorbital-template-module` to `lexorbital-your-module`
- ☐ `description`: Meaningful description of your module (no reference to "template")
- ☐ `type`: Appropriate type for your module
- ☐ `version`: Semver version (generally `0.1.0` to start)

LexOrbital Configuration

- ☐ `lexorbital.role`: Specific role (not `template-module`)
- ☐ `lexorbital.layer`: Appropriate architectural layer
- ☐ `lexorbital.tags`: Relevant tags for your module
- ☐ `lexorbital.compatibility.metaKernel`: Compatible version

Metadata

- ☐ `maintainer.name`: Your name or organization (not `LexOrbital Core`)
- ☐ `maintainer.contact`: Your email or contact URL
- ☐ `repository.url`: Your repository URL (not the template's)
- ☐ `license`: Appropriate license (default `MIT`)

Entry Points

- ☐ `entryPoints.main`: Main entry point (generally `dist/index.js`)
- ☐ `entryPoints.types`: TypeScript definitions (generally `dist/index.d.ts`)

Types and Layers

Module Types

Type	Description	Example
`utility`	Utilities and helpers	Formatting functions, date helpers
`service`	Business services	Authentication service, API client
`ui-component`	UI components	Buttons, forms, modals
`data-provider`	Data providers	Database connectors, API wrappers
`middleware`	Middlewares	Logging, validation, transformation
`plugin`	Extensible plugins	Extensions, system hooks
`theme`	Visual themes	Styles, visual configurations
`integration`	Third-party integrations	Stripe, SendGrid, AWS
`library`	Generic libraries	Collections, algorithms

Architectural Layers

Layer	Description	Examples
`infrastructure`	Technical infrastructure	Database, caching, logging
`domain`	Business logic	Entities, business rules
`application`	Application coordination	Use cases, services
`presentation`	User interface	Components, views, controllers
`integration`	External integrations	APIs, webhooks, adapters

LexOrbital Template Module Tests

This directory contains tests for the LexOrbital module template.

Test Structure

1. `template-module.test.ts`

Basic tests to verify that the template structure is functional.

Execution:

```
pnpm test
```

2. `module-manifest.test.ts`

Validation tests for the `lexorbital.module.json` manifest.

Execution:

```
pnpm test:manifest
# or
pnpm validate:manifest  # with detailed output
```

Important: Manifest Validation Tests

The tests in `module-manifest.test.ts` **will fail by default** until you have customized your module. This is intentional!

Why do these tests fail?

These tests verify that you have **customized the template** with your own module's information. They fail as long as you use the template's default values:

- `name: "lexorbital-template-module"`
- `lexorbital.role: "template-module"`
- `maintainer.name: "LexOrbital Core"`
- `repository.url: "https://github.com/YohanGH/lexorbital-template-module"`

How to make these tests pass?

1. **Open** `lexorbital.module.json`
2. **Modify** the following fields:

```
{
  "$schema": "./schemas/module-manifest.schema.json",
  "name": "lexorbital-my-awesome-module", //   Change this
  "description": "My awesome module for...", //   Change this
  "type": "service", //   Choose the right type
  "version": "0.1.0",
  "entryPoints": {
    "main": "dist/index.js",
    "types": "dist/index.d.ts"
  },
  "lexorbital": {
    "role": "authentication-service", //   Change this
    "layer": "application", //   Choose the right layer
    "compatibility": {
      "metaKernel": ">=1.0.0 <2.0.0"
    },
    "tags": ["auth", "security"] //   Add your tags
  },
  "maintainer": {
    "name": "Your Name", //   Change this
    "contact": "your.email@example.com" //   Change this
  },
  "repository": {
    "type": "git",
    "url": "https://github.com/your-username/your-repo" //   Change this
  },
  "license": "MIT"
}
```

3. **Rerun** the tests:

```
pnpm test:manifest
```

Example Output with Errors

If you haven't customized the manifest, you'll see:

```
 Module Manifest Customization Issues:

 Please change the 'name' field from 'lexorbital-template-module' to your module name
 Please change 'lexorbital.role' from 'template-module' to your module's actual role
 Please update 'maintainer.name' with your name or organization
 Please update 'repository.url' with your repository URL
  Consider updating 'description' to remove references to 'template'

 See docs/02_module-manifest.md for more information
```

Example Successful Output

Once the manifest is customized:

```
 tests/module-manifest.test.ts > Module Manifest Validation (17 tests) 45ms
   should load the manifest file
   should load the schema file
   should validate the manifest against the JSON schema
   should have a customized module name
   should have a meaningful description
   should have a customized role
```

```
    should have customized maintainer information
    should have a customized repository URL
  ...

Test Files  1 passed (1)
     Tests  17 passed (17)
```

JSON Schema Validation

The JSON schema (`schemas/module-manifest.schema.json`) automatically validates:

Required Fields

- `name`: `lexorbital-*` format (not default)
- `description`: 10-500 characters
- `type`: One of the valid types
- `version`: Valid semver
- `entryPoints`: Entry points to `dist/`
- `lexorbital`: Complete configuration
- `maintainer`: Name and contact
- `repository`: Type and URL
- `license`: License identifier

Valid Module Types

- `utility`, `service`, `ui-component`, `data-provider`
- `middleware`, `plugin`, `theme`, `integration`, `library`

Valid Architectural Layers

- `infrastructure`, `domain`, `application`, `presentation`, `integration`

CI/CD Integration

These tests are automatically executed in the CI/CD pipeline to ensure that:

1. The manifest is valid according to the JSON schema
2. All required fields are present
3. The template has been customized (no default values)
4. Versions follow the semver format
5. URLs and formats are correct

Local Development

Run Manifest Tests Only

```
pnpm test:manifest
```

Need Help?

If the tests fail and you don't understand why:

1. Read the detailed error messages

2. Consult `docs/02_module-manifest.md`
3. Verify that you have modified ALL default values
4. Consult `schemas/README.md` for validation rules

The tests are there to guide you!