Repository and GitHub Management Guide

This guide provides comprehensive instructions for managing the neozipkit open source library repository, including GitHub settings, CI/CD workflows, version management, publishing procedures, and maintenance tasks.

Table of Contents

---

1. Repository Overview

Repository Structure

The neozipkit repository is organized into several key directories:

```
neozipkit/
├── src/                    # Source TypeScript code
│   ├── core/              # Core ZIP functionality
│   ├── browser/           # Browser-specific implementations
│   ├── server/            # Server-specific implementations
│   ├── blockchain/        # Blockchain integration (NFT, OTS)
│   ├── encryption/        # Encryption utilities
│   └── types/             # TypeScript type definitions
├── dist/                  # Production build output (tracked in git)
├── dev-dist/              # Development build output (ignored by git)
├── contracts/             # Smart contracts for blockchain features
│   ├── src/               # Solidity source files
│   ├── scripts/           # Deployment and verification scripts
│   ├── ABI.txt            # Deployed contract ABI
│   └── Bytecode.txt       # Deployed contract bytecode
├── examples/              # Example code demonstrating library usage
│   ├── create-zip.ts      # ZIP creation example
│   ├── extract-zip.ts     # ZIP extraction example
│   ├── list-zip.ts        # ZIP listing example
│   └── blockchain-tokenize.ts  # Blockchain tokenization example
├── scripts/               # Build and utility scripts
│   ├── check-branch.js    # Branch protection script
│   ├── auto-build.js      # Smart build selection
│   ├── update-version.js  # Version management
│   └── create-browser-entry.js  # Browser bundle creation
├── .github/
│   └── workflows/         # GitHub Actions workflows
│       ├── ci.yml         # Continuous integration
│       └── publish.yml    # Automated publishing
└── package.json           # Package configuration and scripts
```

Key Directories

`src/` - Source TypeScript code

- All library code is written in TypeScript
- Organized by platform (browser/server) and functionality
- Core functionality is platform-agnostic
- Platform-specific code is in separate directories

`dist/` - Production Build Output

- Compiled JavaScript and TypeScript declarations
- Tracked in git (required for npm package)
- Generated by `yarn build` command
- Used for npm publishing

`dev-dist/` - Development Build Output

- Development builds for local testing
- Ignored by git (not committed)
- Generated by `yarn dev:build` command
- Used during development on feature branches

`contracts/` - Smart Contracts

- Isolated Solidity contracts directory
- Has its own `package.json` and dependencies
- Contains deployment and verification scripts
- See `contracts/README.md` for contract-specific documentation

`examples/` - Example Code

- Basic examples demonstrating core functionality
- Can be run with `ts-node` directly
- See `examples/README.md` for usage instructions

`scripts/` - Build Scripts

- Utility scripts for build automation
- Branch protection and CI/CD helpers
- Version management automation

Dual Build System

The repository uses a dual build system to support both production and development workflows:

- **Production Build (`dist/`)**:
  - Used for npm publishing
  - Tracked in git
  - Protected in CI (main branch only)
  - Generated by `yarn build`
- **Development Build (`dev-dist/`)**:
  - Used for local development
  - Ignored by git
  - Available on all branches
  - Generated by `yarn dev:build`

See `DEV_BUILD.md` for detailed information about the development build system.

Related Documentation

- `README.md` - Main project documentation and API reference
- `VERSION_MANAGEMENT.md` - Version management procedures
- `DEV_BUILD.md` - Development build system documentation
- `BRANCH_PROTECTION_ANALYSIS.md` - Branch protection strategy
- `contracts/README.md` - Smart contract documentation
- `examples/README.md` - Example code documentation

---

2. GitHub Repository Settings

Repository Visibility

The repository should be set to **Public** to allow open source contributions:

1. Go to **Settings** → **General**
2. Scroll to **Danger Zone**
3. Ensure repository is set to **Public**

Default Branch

The default branch should be set to **main** (or **master** if using legacy naming):

1. Go to **Settings** → **Branches**
2. Under **Default branch**, select **main**
3. Click **Update** and confirm

Branch Protection Rules

Configure branch protection for the main branch:

1. Go to **Settings** → **Branches**
2. Click **Add rule** or edit existing rule for `main`
3. Configure the following settings:

Required Settings:

- ✅ **Require a pull request before merging**
  - Require approvals: 1 (or more as needed)
  - Dismiss stale pull request approvals when new commits are pushed
  - Require review from Code Owners (if using CODEOWNERS file)
- ✅ **Require status checks to pass before merging**
  - Require branches to be up to date before merging
  - Status checks to require:
    - `build` (from CI workflow)
    - Any other required checks
- ✅ **Require conversation resolution before merging**
  - All comments and review feedback must be addressed
- ✅ **Do not allow bypassing the above settings**
  - Even administrators should follow these rules

Optional but Recommended:

- ✅ **Require linear history** (prevents merge commits)
- ✅ **Include administrators** (apply rules to all users)

- ✅ **Restrict who can push to matching branches** (if using teams)

Required Status Checks

The following status checks should be required before merging:

- `build` - From `.github/workflows/ci.yml`
- Any other automated tests or linting checks

To configure:

1. In branch protection rules, under **Require status checks to pass**
2. Check the boxes for required checks
3. Ensure **Require branches to be up to date before merging** is enabled

Pull Request Requirements

Configure pull request settings:

1. Go to **Settings** → **General** → **Pull Requests**
2. Configure:
   - ✅ **Allow merge commits** (or prefer squash/rebase)
   - ✅ **Allow squash merging** (recommended for clean history)
   - ✅ **Allow rebase merging** (optional)
   - ✅ **Always suggest updating pull request branches**
   - ✅ **Allow auto-merge** (optional, for automated merging)

Secrets Configuration

GitHub Secrets are used to store sensitive information for CI/CD workflows.

Required Secrets

`NPM_TOKEN` - Required for publishing to npm:

1. Go to **Settings** → **Secrets and variables** → **Actions**
2. Click **New repository secret**
3. Name: `NPM_TOKEN`
4. Value: Your npm authentication token
   - Generate at: https://www.npmjs.com/settings/[username]/tokens
   - Select **Automation** token type
   - Grant **Publish** permission
5. Click **Add secret**

`GITHUB_TOKEN` - Automatically provided by GitHub Actions:

- No configuration needed
- Automatically available in workflows
- Has permissions to create releases and comments

Optional Secrets

If using additional services, you may need:

- `ETHERSCAN_API_KEY` - For contract verification (if automating)
- `ALCHEMY_API_KEY` - For blockchain RPC access (if needed)
- Any other service-specific API keys

Permissions

Collaborator Access Levels

Configure collaborator permissions:

1. Go to **Settings** → **Collaborators and teams**
2. Add collaborators with appropriate access:
    - **Read** - For external contributors (view only)
    - **Triage** - Can manage issues and PRs
    - **Write** - Can push to branches (use sparingly)
    - **Maintain** - Can manage repository settings (use for trusted maintainers)
    - **Admin** - Full access (use only for repository owners)

Team Permissions

If using GitHub Teams:

1. Go to **Settings** → **Collaborators and teams**
2. Create teams with appropriate permissions
3. Assign team members
4. Grant team access to repository

External Contributor Permissions

External contributors (non-collaborators) can:

- Fork the repository
- Create branches in their fork
- Submit pull requests
- Open issues
- Comment on issues and PRs

They cannot:

- Push directly to the repository
- Merge pull requests
- Access secrets
- Modify repository settings

---

3. CI/CD Workflows

The repository uses GitHub Actions for continuous integration and deployment.

Workflow Files

All workflows are located in `.github/workflows/`:

- `ci.yml` - Continuous integration (build and test)
- `publish.yml` - Automated publishing to npm

CI Workflow (`ci.yml`)

Purpose: Build and verify the codebase on pull requests and main branch pushes.

Triggers:

- Pull requests targeting `main` or `master`
- Pushes to `main` or `master` branches

What It Does:

1. Checks out the code
2. Sets up Node.js 20 with Yarn caching
3. Installs dependencies with `yarn install --frozen-lockfile`
4. Builds the project:
    - Main branch → Production build (`yarn build:ci` → `dist/`)

    - PR branches → Development build (`yarn dev:build` → `dev-dist/`)
5. Verifies build artifacts exist

Key Features:

- Uses Yarn package manager
- Caches dependencies for faster builds
- Different build targets based on branch
- Fails if build artifacts are missing

Monitoring:

- View workflow runs in **Actions** tab
- Check status on pull request page
- Review logs for build errors

Publish Workflow (`publish.yml`)

Purpose: Automatically publish to npm and create GitHub releases when version tags are pushed.

Triggers:

- Push of tags matching pattern `v*` (e.g., `v1.0.0` , `v0.3.1`)

What It Does:

1. Checks out the code
2. Sets up Node.js 20 with npm registry configuration
3. Installs dependencies
4. Builds production version (`yarn build:ci`)
5. Verifies build artifacts
6. Publishes to npm (if tag matches `v*`)
7. Creates GitHub release

Key Features:

- Only runs on version tags
- Requires `NPM_TOKEN` secret
- Automatically creates GitHub releases
- Uses production build only

Publishing Process:

1. Version tag is pushed (e.g., `git push origin v1.0.0`)
2. Workflow automatically triggers
3. Builds and publishes to npm
4. Creates GitHub release with tag name

Monitoring:

- View workflow runs in **Actions** tab
- Check npm package page for new version
- Verify GitHub release was created

Workflow Status

Viewing Workflow Runs:

1. Go to **Actions** tab in GitHub
2. Select workflow from left sidebar
3. Click on a run to see details
4. Expand steps to view logs

Workflow Status on PRs:

- Status checks appear at bottom of PR
- Green checkmark = passed
- Red X = failed
- Yellow circle = in progress

Required Status Checks:

- Must pass before PR can be merged
- Configured in branch protection rules

Troubleshooting Failed Workflows

Common Issues:

1. **Build Failures:**
   - Check TypeScript compilation errors
   - Verify dependencies are up to date
   - Check Node.js version compatibility
2. **Publishing Failures:**
   - Verify `NPM_TOKEN` secret is set correctly
   - Check npm package name and version
   - Ensure version tag format is correct (`v*`)
3. **Dependency Issues:**
   - Run `yarn install` locally to reproduce
   - Check `yarn.lock` for conflicts
   - Verify package.json dependencies

Debugging Steps:

1. Click on failed workflow run
2. Expand failed step to view logs
3. Look for error messages
4. Reproduce locally if possible
5. Fix issue and push changes

---

4. Version Management

Version management is handled through `package.json` as the single source of truth. See `VERSION_MANAGEMENT.md` for complete details.

Semantic Versioning

The project follows Semantic Versioning (SemVer):

- **MAJOR** (1.0.0) - Breaking changes
- **MINOR** (0.1.0) - New features, backward compatible
- **PATCH** (0.0.1) - Bug fixes, backward compatible

Current version format: `MAJOR.MINOR.PATCH` (e.g., `0.3.1`)

Version Update Commands

Use npm scripts to update versions:

```
# Patch release (bug fixes)
yarn version:patch    # 0.3.1 → 0.3.2

# Minor release (new features)
yarn version:minor    # 0.3.1 → 0.4.0

# Major release (breaking changes)
yarn version:major    # 0.3.1 → 1.0.0

# Set specific version
yarn version:set 1.2.3
```

Single Source of Truth

`package.json` is the authoritative source for version information:

```
{
  "name": "@neozip/neozipkit",
  "version": "0.3.1",  // ← This controls ALL version information
  ...
}
```

All version references throughout the codebase automatically read from `package.json`:

- `src/core/version.ts` - Dynamically imports from package.json
- Build output - Reflects package.json version
- Documentation - References stay current

Automatic Version Propagation

When you update the version in `package.json`, the following are automatically updated on the next build:

- ✅ `VERSION.number` - Always matches package.json
- ✅ `VERSION.date` - Current date when built
- ✅ `NEOZIPKIT_INFO.version` - Used throughout the codebase
- ✅ Example files - All version references
- ✅ Compiled output - All built files

No manual updates needed - just update `package.json` and run `yarn build`.

Release Process

Step-by-step release workflow:

1. **Update Version:**

   ```
   yarn version:patch    # or :minor, :major
   # Or manually edit package.json
   ```

2. **Build Production Version:**

   ```
   yarn build
   ```

3. **Test the Build:**

```
# Verify dist/ directory exists
ls -la dist/

# Test import
node -e "const pkg = require('./dist/index.js'); console.log(pkg);"
```

4. **Commit Changes:**

```
git add .
git commit -m "Release v0.3.2"
```

5. **Create Version Tag:**

```
git tag v0.3.2
```

6. **Push Changes and Tag:**

```
git push origin main
git push origin v0.3.2
```

7. **Verify Publication:**
   - Check GitHub Actions for publish workflow
   - Verify npm package page for new version
   - Check GitHub Releases for new release

Creating Version Tags

Version tags must follow the format `v*` (e.g., `v1.0.0`, `v0.3.1`):

```
# Create annotated tag (recommended)
git tag -a v1.0.0 -m "Release version 1.0.0"

# Or create lightweight tag
git tag v1.0.0

# Push tag to trigger publish workflow
git push origin v1.0.0
```

Important: The publish workflow only triggers on tags matching `v*` pattern.

Release Notes

GitHub releases are automatically created by the publish workflow. To add release notes:

1. Go to **Releases** in GitHub
2. Click **Edit** on the release
3. Add release notes describing changes
4. Save changes

Best Practices:

- List new features
- Document breaking changes
- Include migration guides if needed

- Link to relevant issues and PRs

Version Management Best Practices

1. **Always build after version changes:**

   ```
   yarn version:patch
   yarn build  # ← Required to update all version references
   ```

2. **Use semantic versioning consistently:**

   - Patch for bug fixes
   - Minor for new features
   - Major for breaking changes

3. **Test before releasing:**

   - Build and test locally
   - Run examples if applicable
   - Verify TypeScript compilation

4. **Document breaking changes:**

   - Update README if API changes
   - Add migration notes if needed
   - Update examples if they break

---

5. Publishing Process

This section provides a step-by-step guide for publishing new versions to npm.

Pre-Publishing Checklist

Before publishing, ensure:

- ✅ All tests passing (if applicable)
- ✅ Documentation updated (README.md, examples)
- ✅ Version number correct in package.json
- ✅ Build artifacts verified (dist/ directory)
- ✅ No uncommitted changes
- ✅ All changes committed and pushed
- ✅ Branch is up to date with main

Step-by-Step Publishing Guide

1. Update Version

Choose the appropriate version bump:

```
# Bug fix release
yarn version:patch

# New feature release
yarn version:minor

# Breaking change release
yarn version:major
```

```
# Or set specific version
yarn version:set 1.0.0
```

This updates `package.json` version field.

2. Build Production Version

Build the production package:

```
yarn build
```

This creates the `dist/` directory with compiled JavaScript and TypeScript declarations.

3. Test the Build

Verify the build works correctly:

```
# Check dist/ directory exists
ls -la dist/

# Test importing the package
node -e "const pkg = require('./dist/index.js'); console.log('Build OK:', !!pkg);"

# Verify version in built files (optional)
node -e "const { VERSION } = require('./dist/core/version.js'); console.log('Version:',
VERSION.number);"
```

4. Review Changes

Review what will be published:

```
# See what files will be included
git status

# Review package.json changes
git diff package.json

# Check dist/ contents
ls -R dist/ | head -20
```

5. Commit Changes

Commit the version update and build:

```
git add .
git commit -m "Release v1.0.0"
```

Commit message format: `Release v<VERSION>`

6. Create Version Tag

Create a version tag:

```
git tag v1.0.0
```

Tag format: Must start with `v` followed by version number (e.g., `v1.0.0`, `v0.3.1`)

7. Push Changes and Tag

Push commits and tag to trigger publish workflow:

```
# Push commits
git push origin main

# Push tag (triggers publish workflow)
git push origin v1.0.0
```

8. Monitor Publication

Monitor the publication process:

1. **Check GitHub Actions:**
   - Go to **Actions** tab
   - Find the `publish` workflow run
   - Verify it completes successfully
2. **Verify npm Publication:**
   - Visit: https://www.npmjs.com/package/@neozip/neozipkit
   - Confirm new version appears
   - Check version number matches
3. **Verify GitHub Release:**
   - Go to **Releases** in GitHub
   - Confirm release was created
   - Add release notes if needed

Automated Publishing

The repository uses automated publishing via GitHub Actions:

1. **Push version tag** (e.g., `git push origin v1.0.0`)
2. **Publish workflow triggers** automatically
3. **Workflow builds** production version
4. **Workflow publishes** to npm
5. **Workflow creates** GitHub release

No manual npm publish needed - the workflow handles everything.

Manual Publishing (Not Recommended)

If you need to publish manually (not recommended):

```
# Build first
yarn build

# Publish to npm
npm publish

# Requires NPM_TOKEN or npm login
```

Note: Manual publishing bypasses CI/CD checks and GitHub release creation.

Publishing Troubleshooting

Issue: Publish workflow doesn't trigger

- Verify tag format is `v*` (e.g., `v1.0.0`)
- Check tag was pushed: `git push origin v1.0.0`
- Verify workflow file exists: `.github/workflows/publish.yml`

Issue: npm publish fails

- Check `NPM_TOKEN` secret is set in GitHub
- Verify token has publish permissions
- Check package name matches npm package
- Ensure version doesn't already exist on npm

Issue: Build fails in workflow

- Check TypeScript compilation errors
- Verify dependencies are correct
- Review workflow logs for specific errors

Issue: GitHub release not created

- Check `GITHUB_TOKEN` permissions
- Verify workflow completed successfully
- Check release was created manually if needed

Post-Publishing Tasks

After successful publication:

1. **Update Release Notes:**
    - Go to GitHub Releases
    - Edit the release
    - Add detailed release notes
2. **Announce Release:**
    - Update project changelog (if maintained)
    - Announce on project communication channels
    - Update documentation if needed
3. **Monitor for Issues:**
    - Watch for user reports
    - Monitor npm download statistics
    - Check for any immediate issues

---

6. Branch Strategy

The repository uses a CI-Only Protection approach that allows local development on any branch while protecting production builds in CI/CD. See `BRANCH_PROTECTION_ANALYSIS.md` for detailed analysis.

Branch Protection Strategy

CI-Only Protection (Implemented):

- ✅ **Local Development:** Builds work on any branch locally
- ✅ **CI/CD Protection:** Only main branch builds to `dist/` in CI
- ✅ **Open Source Friendly:** External contributors can build locally
- ✅ **Publishing Protected:** Only main branch can publish to npm

Branch Types

Main Branch (`main` or `master`):

- Production-ready code
- Protected by branch protection rules
- Only builds to `dist/` in CI
- Source of npm publications
- Requires PR reviews before merging

Feature Branches (`feature/*`):

- New features and enhancements
- Can be built locally with `yarn build`
- Build to `dev-dist/` in CI
- Merged via pull requests

Fix Branches (`fix/*`):

- Bug fixes and patches
- Same workflow as feature branches
- Merged via pull requests

Development Branch (`dev`):

- Optional development integration branch
- Not currently used in this repository
- Could be used for staging multiple features

Branch Naming Conventions

Use descriptive branch names:

```
# Features
feature/add-compression-method
feature/blockchain-integration

# Bug fixes
fix/zip-extraction-error
fix/type-definition-issue

# Documentation
docs/update-readme
docs/add-examples

# Refactoring
refactor/improve-error-handling
```

Local Development Workflow

On Any Branch:

```
# Build production version (works locally on any branch)
yarn build

# Build development version (recommended for feature branches)
yarn dev:build
```

```
# Watch mode for development
yarn dev:watch
```

Key Points:

- ✅ `yarn build` works locally on any branch
- ✅ `yarn dev:build` always works
- ✅ No local restrictions on building
- ✅ CI enforces protection, not local scripts

CI/CD Branch Behavior

In GitHub Actions:

- **Main branch pushes:**
  - Builds to `dist/` (production)
  - Runs `yarn build:ci`
  - Protected - only main branch can build to `dist/`
- **PR branches:**
  - Builds to `dev-dist/` (development)
  - Runs `yarn dev:build`
  - No restrictions - all PRs can build
- **Version tags:**
  - Triggers publish workflow
  - Only works from main branch context
  - Publishes to npm and creates GitHub release

Pull Request Workflow

Creating a Pull Request:

1. **Create Feature Branch:**

   ```
   git checkout -b feature/my-feature
   ```

2. **Make Changes:**
   - Edit source files in `src/`
   - Test locally with `yarn dev:build`
   - Commit changes
3. **Push Branch:**

   ```
   git push origin feature/my-feature
   ```

4. **Create Pull Request:**
   - Go to GitHub repository
   - Click **New Pull Request**
   - Select base: `main`, compare: `feature/my-feature`
   - Fill in description
   - Submit PR

PR Review Process:

1. **CI Checks Run:**
    - Build workflow runs automatically
    - Status checks appear on PR
    - Must pass before merging
2. **Code Review:**
    - Maintainers review code
    - Request changes if needed
    - Approve when ready
3. **Merge:**
    - Squash and merge (recommended)
    - Or merge commit
    - Or rebase and merge

Required for Merging:

- ✅ All CI checks passing
- ✅ At least one approval
- ✅ No merge conflicts
- ✅ Branch up to date with main

Merging Strategy

Recommended: Squash and Merge

- Creates single commit on main branch
- Clean commit history
- Easier to revert if needed

Alternative: Rebase and Merge

- Preserves individual commits
- Linear history
- More detailed commit log

Not Recommended: Merge Commit

- Creates merge commit
- Clutters history
- Harder to follow

Branch Protection Rules

See GitHub Repository Settings for detailed branch protection configuration.

Key Rules:

- Require PR before merging
- Require status checks to pass
- Require code review approval
- Require conversation resolution
- Include administrators

---

7. Maintenance Tasks

Regular maintenance keeps the repository healthy and up to date.

Regular Maintenance

Weekly Tasks:

- Review and respond to open issues
- Review and merge approved pull requests
- Monitor CI/CD workflow status
- Check for security advisories

Monthly Tasks:

- Review open issues and prioritize
- Update dependencies (see below)
- Run security audit
- Review and update documentation
- Performance review

Quarterly Tasks:

- Major dependency updates
- Architecture review
- Documentation audit
- Community engagement review

Dependency Updates

Check for Updates:

```
# Check outdated packages
yarn outdated

# Check for security vulnerabilities
yarn audit
```

Update Dependencies:

```
# Update all dependencies to latest (within semver)
yarn upgrade

# Update specific package
yarn upgrade package-name

# Update to latest version (may break semver)
yarn upgrade package-name --latest
```

Update Lock File:

After updating dependencies:

```
# Regenerate lock file
yarn install

# Commit changes
git add package.json yarn.lock
git commit -m "chore: update dependencies"
```

Best Practices:

- Update regularly (monthly recommended)
- Test after updates
- Review changelogs for breaking changes
- Update one major dependency at a time
- Commit dependency updates separately from features

Security Audits

Run Security Audit:

```
# Check for vulnerabilities
yarn audit

# Fix automatically (if possible)
yarn audit fix

# Fix with breaking changes
yarn audit fix --force
```

Review Security Advisories:

- Check GitHub Security tab
- Review npm security advisories
- Monitor for critical vulnerabilities
- Update immediately for critical issues

Security Update Process:

1. Identify vulnerable dependency
2. Check for fixed version
3. Update dependency
4. Test thoroughly
5. Create security patch release if needed

TypeScript Updates

Update TypeScript:

```
# Check current version
yarn list typescript

# Update TypeScript
yarn upgrade typescript --latest

# Test compilation
yarn build
```

After TypeScript Update:

- Review any new type errors
- Update type definitions if needed
- Test compilation on all platforms
- Update documentation if API changes

Node.js Version Updates

Check Node.js Version:

The repository targets Node.js 20 (as specified in CI workflows).

Update Node.js:

- Update `.github/workflows/ci.yml` and `publish.yml`
- Update `package.json` engines field (if specified)
- Test with new Node.js version
- Update documentation if needed

Documentation Updates

Regular Documentation Review:

- Update README.md with new features
- Update examples if API changes
- Review and fix broken links
- Add missing documentation
- Update version-specific documentation

Documentation Checklist:

- ✅ README.md is current
- ✅ Examples work correctly
- ✅ API documentation is accurate
- ✅ All links are valid
- ✅ Version information is correct

Issue Management

Regular Issue Review:

- Triage new issues (label, assign, prioritize)
- Respond to questions
- Close resolved issues
- Link related issues
- Create issues for planned work

Issue Labels:

Use labels to organize issues:

- `bug` - Something isn't working
- `enhancement` - New feature or request
- `documentation` - Documentation improvements
- `question` - Questions or discussions
- `good first issue` - Good for new contributors
- `help wanted` - Extra attention needed

Performance Review

Monthly Performance Check:

- Review build times
- Check bundle sizes
- Monitor npm download statistics
- Review CI/CD workflow performance
- Optimize slow processes

Code Quality

Regular Code Review:

- Review code style consistency
- Check for code smells

- Review test coverage (if applicable)
- Refactor as needed
- Update coding standards

---

8. Contributing Guidelines

This section provides guidelines for both external contributors and maintainers.

For External Contributors

Getting Started

1. Fork the Repository:

- Click **Fork** button on GitHub
- Creates a copy in your GitHub account

2. Clone Your Fork:

```
git clone https://github.com/YOUR_USERNAME/neozipkit.git
cd neozipkit
```

3. Add Upstream Remote:

```
git remote add upstream https://github.com/NeoWareInc/neozipkit.git
```

4. Install Dependencies:

```
yarn install
```

Development Setup

Build for Development:

```
# Development build (recommended)
yarn dev:build

# Or watch mode for auto-rebuild
yarn dev:watch
```

Test Your Changes:

```
# Build and test
yarn dev:build

# Run examples (if applicable)
ts-node examples/create-zip.ts
```

Making Changes

1. Create Feature Branch:

```
git checkout -b feature/my-feature
```

2. Make Changes:

- Edit files in `src/` directory
- Follow existing code style
- Add comments for complex logic
- Test your changes

3. Commit Changes:

```
git add .
git commit -m "feat: add new feature description"
```

Commit Message Format:

- `feat:` - New feature
- `fix:` - Bug fix
- `docs:` - Documentation
- `refactor:` - Code refactoring
- `test:` - Tests
- `chore:` - Maintenance

4. Push to Your Fork:

```
git push origin feature/my-feature
```

5. Create Pull Request:

- Go to GitHub repository
- Click **New Pull Request**
- Select your fork and branch
- Fill in description
- Submit PR

Code Style Guidelines

TypeScript:

- Use TypeScript strict mode
- Provide type annotations
- Use interfaces for object shapes
- Follow existing code patterns

Naming Conventions:

- `camelCase` for variables and functions
- `PascalCase` for classes and interfaces
- `UPPER_CASE` for constants
- Descriptive names

Documentation:

- Add JSDoc comments for public APIs
- Document complex algorithms
- Include usage examples

Testing Your Changes

Before Submitting PR:

- ✅ Code compiles without errors
- ✅ Build succeeds (`yarn dev:build`)
- ✅ Examples work (if applicable)

- ✅ No linter errors
- ✅ Follows code style

Pull Request Guidelines

PR Description Should Include:

- What changes were made
- Why the changes were needed
- How to test the changes
- Screenshots (if UI changes)
- Related issues (if any)

PR Best Practices:

- Keep PRs focused (one feature/fix per PR)
- Keep PRs small when possible
- Respond to review feedback
- Update PR if requested
- Link to related issues

For Maintainers

Review Process

Reviewing Pull Requests:

1. Check CI status (must be passing)
2. Review code changes
3. Test changes locally (if needed)
4. Request changes or approve
5. Merge when ready

Review Checklist:

- ✅ Code follows style guidelines
- ✅ Changes are well-tested
- ✅ Documentation is updated
- ✅ No breaking changes (or documented)
- ✅ CI checks pass

Merging Pull Requests

Merge Options:

- **Squash and merge** (recommended) - Single commit
- **Rebase and merge** - Preserves commits
- **Merge commit** - Not recommended

After Merging:

- Delete feature branch (if applicable)
- Close related issues
- Update documentation if needed

Handling Issues

Issue Triage:

1. Label appropriately
2. Assign if needed
3. Prioritize
4. Respond to questions

5. Close when resolved

Issue Types:

- **Bug Reports:** Reproduce, fix, test
- **Feature Requests:** Evaluate, plan, implement
- **Questions:** Answer or direct to documentation

Release Management

See Publishing Process for detailed release procedures.

Release Responsibilities:

- Update version numbers
- Create release notes
- Publish to npm
- Announce releases
- Monitor for issues

---

9. Security Considerations

Security is critical for an open source library. This section covers secrets management, code security, and best practices.

Secrets Management

Never Commit Secrets

What NOT to Commit:

- API keys
- Private keys
- Passwords
- Authentication tokens
- Environment variables with secrets

Files to Never Commit:

- `.env` files
- `*.key` files
- `*.pem` files (unless public)
- Configuration files with secrets

Using GitHub Secrets

For CI/CD Workflows:

- Store secrets in GitHub Secrets
- Access via `${{ secrets.SECRET_NAME }}`
- Never log secrets in workflow output

Adding Secrets:

1. Go to **Settings** → **Secrets and variables** → **Actions**
2. Click **New repository secret**
3. Enter name and value
4. Click **Add secret**

Required Secrets:

- `NPM_TOKEN` - For npm publishing
- `GITHUB_TOKEN` - Automatically provided

Secret Rotation

Regular Rotation:

- Rotate secrets quarterly (or as needed)
- Rotate immediately if compromised
- Update all workflows using the secret
- Test after rotation

Rotation Process:

1. Generate new secret
2. Update GitHub Secret
3. Test workflow
4. Revoke old secret

Code Security

Dependency Vulnerabilities

Regular Audits:

```
# Check for vulnerabilities
yarn audit

# Fix automatically
yarn audit fix

# Review critical issues
yarn audit --level high
```

Security Update Process:

1. Identify vulnerable dependency
2. Check for patched version
3. Update dependency
4. Test thoroughly
5. Release security patch if needed

Smart Contract Security

For `contracts/` Directory:

- Audit contracts before deployment
- Use established libraries (OpenZeppelin)
- Test thoroughly
- Document security assumptions
- Monitor for vulnerabilities

Contract Deployment:

- Use deterministic deployment
- Verify contracts on block explorers
- Document deployment addresses
- Keep deployment keys secure

Access Control

Repository Access:

- Limit admin access to trusted maintainers
- Use teams for permission management

- Review access regularly
- Remove access when no longer needed

CI/CD Access:

- Only maintainers can modify workflows
- Secrets are protected
- Workflow logs are visible to all
- No secrets in logs

Best Practices

Secure Development

Code Review:

- All changes require review
- Security-sensitive code gets extra scrutiny
- Use automated security scanning
- Review dependency updates

Secure Coding:

- Validate all inputs
- Use parameterized queries (if applicable)
- Avoid eval() and similar functions
- Handle errors securely
- Don't log sensitive data

Security Monitoring

Regular Monitoring:

- Check security advisories monthly
- Monitor npm security alerts
- Review GitHub security tab
- Watch for dependency vulnerabilities

Incident Response:

- Have a security contact
- Document response procedures
- Act quickly on critical issues
- Communicate transparently

Security Resources

Tools:

- `yarn audit` - Dependency vulnerability scanning
- GitHub Security tab - Repository security overview
- npm security advisories - Package vulnerability database

Reporting Security Issues:

- Use GitHub Security Advisories
- Or contact maintainers directly
- Provide detailed information
- Allow time for fix before disclosure

---

10. Troubleshooting

This section covers common issues and their solutions.

Build Failures

TypeScript Compilation Errors

Symptoms:

- Build fails with TypeScript errors
- Type errors in console output

Solutions:

```
# Check TypeScript version
yarn list typescript

# Clear build cache
yarn clean
yarn build

# Check for type errors
yarn compile

# Update TypeScript if needed
yarn upgrade typescript
```

Missing Dependencies

Symptoms:

- `Module not found` errors
- Missing package errors

Solutions:

```
# Reinstall dependencies
rm -rf node_modules yarn.lock
yarn install

# Check package.json
cat package.json | grep dependencies

# Verify yarn.lock is committed
git status yarn.lock
```

Build Script Errors

Symptoms:

- Script execution fails
- Permission errors

Solutions:

```
# Check script permissions
chmod +x scripts/*.js

# Run scripts directly
node scripts/check-branch.js
```

```
# Check Node.js version
node --version  # Should be 20+
```

CI/CD Workflow Failures

Workflow Doesn't Trigger

Symptoms:

- Workflow doesn't run on push/PR
- No workflow run appears

Solutions:

- Check workflow file syntax (YAML)
- Verify trigger conditions
- Check branch names match
- Ensure workflow file is in `.github/workflows/`

Build Fails in CI

Symptoms:

- CI build fails
- Different behavior than local

Solutions:

- Reproduce locally first
- Check Node.js version matches
- Verify dependencies are locked (`yarn.lock`)
- Check for environment-specific issues

Publish Workflow Fails

Symptoms:

- npm publish fails
- GitHub release not created

Solutions:

- Verify `NPM_TOKEN` secret is set
- Check token has publish permissions
- Verify tag format is `v*`
- Check package name and version

Publishing Errors

Version Already Exists

Symptoms:

- npm publish fails with version conflict
- Version already published

Solutions:

```
# Check current version
cat package.json | grep version

# Update version
```

```
yarn version:patch  # or :minor, :major

# Verify version doesn't exist
npm view @neozip/neozipkit versions
```

npm Authentication Failed

Symptoms:

- `npm publish` fails with auth error
- Token invalid

Solutions:

- Verify `NPM_TOKEN` secret in GitHub
- Check token hasn't expired
- Regenerate token if needed
- Verify token has publish scope

Version Conflicts

Version Mismatch

Symptoms:

- Version in code doesn't match package.json
- Build shows wrong version

Solutions:

```
# Rebuild after version change
yarn version:patch
yarn build

# Verify version in built files
node -e "const { VERSION } = require('./dist/core/version.js'); console.log(VERSION.number);"
```

Git Tag Conflicts

Symptoms:

- Can't push tag (already exists)
- Tag points to wrong commit

Solutions:

```
# Check existing tags
git tag -l

# Delete local tag
git tag -d v1.0.0

# Delete remote tag (if needed)
git push origin --delete v1.0.0

# Create new tag
git tag v1.0.0
git push origin v1.0.0
```

Dependency Issues

Dependency Resolution Errors

Symptoms:

- `yarn install` fails
- Conflicting dependencies

Solutions:

```
# Clear cache and reinstall
rm -rf node_modules yarn.lock
yarn install

# Check for conflicts
yarn check

# Update yarn
yarn set version stable
```

Peer Dependency Warnings

Symptoms:

- Peer dependency warnings
- Missing peer dependencies

Solutions:

- Install peer dependencies in consuming projects
- Document peer dependencies in README
- Check `peerDependencies` in package.json

Getting Help

Resources

- **GitHub Issues:** Open an issue for bugs or questions
- **Documentation:** Check README.md and other docs
- **Examples:** See `examples/` directory
- **Workflow Logs:** Check GitHub Actions for CI errors

Reporting Issues

When Reporting:

- Include error messages
- Provide steps to reproduce
- Include environment details
- Share relevant code/logs

Issue Template:

```
## Description
Brief description of the issue

## Steps to Reproduce
1. Step one
2. Step two
```

```
## Expected Behavior
What should happen

## Actual Behavior
What actually happens

## Environment
- Node.js version:
- Yarn version:
- OS:

## Additional Context
Any other relevant information
```

Conclusion

This guide provides comprehensive instructions for managing the neozipkit repository. For specific topics, refer to:

- **Version Management:** `VERSION_MANAGEMENT.md`
- **Development Builds:** `DEV_BUILD.md`
- **Branch Protection:** `BRANCH_PROTECTION_ANALYSIS.md`
- **Smart Contracts:** `contracts/README.md`
- **Examples:** `examples/README.md`

For questions or issues, please open a GitHub issue or contact the maintainers.