Coding Standards Documentation

Marito Multilingual Terminology PWA

Team Name: Velox

August 20, 2025

Contents

1. Introduction

This document establishes coding standards for the Marito project, a progressive web application for multilingual lexicons, term banks, and glossaries. These standards ensure consistency, maintainability, and quality across the entire codebase.

1.1. Project Stack

- **Frontend**: React + TypeScript + Vite + Tailwind CSS
- **Backend**: Python + FastAPI + SQLAlchemy + PostgreSQL
- **Infrastructure**: Docker + GitHub Actions

2. General Principles

2.1. Code Quality

- Write clean, readable, and maintainable code
- Use meaningful naming conventions
- Keep functions and classes focused on single responsibilities

2.2. Consistency

- Use consistent formatting across all files
- Follow established patterns within the codebase
- Use automated tooling for code formatting and linting

2.3. Performance

- Optimize for readability first, then performance
- Minimize network requests and database queries
- Implement proper caching strategies

2.4. Security

- Follow security best practices
- Never commit sensitive information
- Validate all user inputs
- Use proper authentication and authorization

3. Frontend Standards (React/TypeScript)

3.1. File Structure and Naming

```
frontend/
|-- .env                              # Environment variables
|-- components.json                   # UI component configurations
|-- eslint.config.js                  # ESLint configuration
|-- index.html                        # Main HTML entry point
|-- jest.config.js                    # Jest testing configuration
|-- lint-staged.config.js             # Lint-staged configuration
|-- package.json                      # Dependencies and scripts
|-- README.md                         # Project documentation
|-- setupTests.ts                     # Test setup configuration
|-- tailwind.config.js                # Tailwind CSS configuration
|-- tsconfig.app.json                 # TypeScript app configuration
|-- tsconfig.json                     # TypeScript configuration
|-- tsconfig.node.json                # TypeScript node configuration
|-- vite.config.ts                    # Vite configuration
|
|-- src/                              # Source code
|   |-- App.css                       # App-specific styles
|   |-- App.tsx                       # Main app component
|   |-- config.ts                     # Application configuration
|   |-- custom.d.ts                   # Custom type definitions
|   |-- i18n.tsx                      # Internationalization setup
|   |-- index.css                     # Global CSS
|   |-- main.tsx                      # Application entry point
|   |-- sw.ts                         # Service worker
|   |-- vite-env.d.ts                 # Vite environment types
|   |
|   |-- components/                   # Reusable UI components
|   |-- hooks/                        # Custom React hooks
|   |-- lib/                          # Library code and utilities
|   |-- pages/                        # Page components
|   |-- styles/                       # CSS and style utilities
|   |-- types/                        # TypeScript type definitions
|   `-- utils/                        # Utility functions
|
|-- public/                           # Static assets
|-- Tests/                            # Test files
|-- dev-dist/                         # Development distribution
|-- dist/                             # Production distribution
`-- __mocks__/                        # Mock files for testing
```

3.2. Naming Conventions

Our frontend follows consistent naming patterns to ensure code readability and maintainability:

- **Components (including Pages)**: Use PascalCase for all React components (`LoginPage.tsx`, `LanguageSwitcher.tsx`, `SearchPage.tsx`). This convention makes it immediately clear that a file contains a React component and distinguishes it from utility functions. Pages are a specific type of React component representing full page views.

- **Utilities**: Use camelCase for utility functions (`termLanguageUtils.ts`, `exportUtils.ts`). These are general-purpose functions that can be used across the application.
- **Constants**: Use UPPER_SNAKE_CASE for application constants (`API_ENDPOINTS`, `DEFAULT_TIMEOUT`). This makes constants easily distinguishable from variables.
- **Types/Interfaces**: Use PascalCase for TypeScript types and interfaces (`Term`, `TermTranslations`). This aligns with TypeScript conventions and distinguishes types from values.
- **Variables and Functions**: Use camelCase for all variables and functions (`determineTermLangua` `searchTermsInDb`, `isLoading`, `handleSubmit`). This is the standard JavaScript convention.
- **Files**: Use camelCase for utility files and PascalCase for component files. This creates a clear distinction between different types of files in the codebase.

3.3. TypeScript Configuration

Our TypeScript configuration emphasizes strict type checking to catch errors early and improve code quality. The `tsconfig.app.json` enforces several important rules:

- **Strict Type Checking**: We enable strict mode to catch potential runtime errors at compile time. This includes strict null checks, strict function types, and strict property initialization.
- **No Unused Code**: The configuration flags unused local variables and parameters, helping maintain clean code and preventing dead code accumulation.
- **Exhaustive Switch Cases**: We prevent fallthrough cases in switch statements, ensuring all possible cases are handled explicitly.

3.3.1 Type Safety Guidelines

- Always provide explicit types for function parameters and return values rather than relying on type inference
- Avoid the `any` type completely; use `unknown` for truly unknown types or create specific union types
- Use interfaces to define object shapes and types for unions or primitive combinations
- Implement comprehensive error handling with properly typed error objects that include error codes and context

3.4. React Component Standards

Our React components follow functional programming principles with hooks, promoting consistency and maintainability across the application.

- **Component Architecture**: All components are functional components using React hooks. This approach provides better performance optimization opportunities and aligns with modern React development practices.

- **Props Interface Definition**: Every component must have a clearly defined props interface declared above the component. This serves as documentation and enables better IDE support with autocomplete and type checking.
- **Prop Handling**: We prefer optional props over default props where possible. Optional props with TypeScript provide better type safety and make component APIs more explicit about what is required versus optional.
- **Component Size and Responsibility**: Components should follow the single responsibility principle. If a component becomes too large or handles multiple concerns, it should be broken down into smaller, focused components.
- **Performance Optimization**: Use React.memo() judiciously for components that receive complex objects as props or render frequently. However, avoid premature optimization and measure performance impact before applying memoization.

3.5. Styling Standards

Our styling approach centers on Tailwind CSS for consistent, maintainable, and responsive design across the application.

- **Tailwind CSS Usage**: We use Tailwind's utility-first approach to build custom designs without writing custom CSS. This approach promotes consistency, reduces CSS bundle size, and makes it easier to maintain responsive designs.
- **Color System**: Our custom color palette is defined in the Tailwind configuration and reflects the project's branding. Use these predefined colors consistently across the application rather than hardcoding color values.
- **Responsive Design**: All components should be responsive by default. Use Tailwind's responsive prefixes (`sm:`, `md:`, `lg:`, `xl:`) to create designs that work across all device sizes.
- **Dark Mode Support**: The application supports dark mode through Tailwind's `dark:` variant. When styling components, consider both light and dark mode appearances.

3.5.1 Component Styling Guidelines

- Group related utility classes together for better readability
- Use consistent spacing scales provided by Tailwind
- Prefer composition over custom CSS whenever possible
- Document any custom utility classes or components in the design system

3.6. Code Quality Tools

Our frontend development relies on several automated tools to maintain code quality and consistency across the team.

- **ESLint Configuration**: We use ESLint with TypeScript-specific rules to catch potential errors, enforce coding standards, and maintain consistency. The configuration extends strict type-checked rules to catch type-related issues early in development.
- **Prettier Integration**: Prettier handles all code formatting automatically, ensuring consistent code style across the team. It's configured to run on save and as part of the pre-commit hooks.

3.6.1 Formatting Standards

- Use semicolons for statement termination
- Prefer single quotes for strings
- Maintain 80 character line width for readability
- Use 2-space indentation for consistency with React ecosystem

3.6.2 Tool Integration

All quality tools are integrated into the development workflow through VS Code settings, pre-commit hooks, and CI/CD pipelines to catch issues before they reach production.

3.7. State Management

Our state management strategy emphasizes simplicity and follows React's built-in patterns for most use cases.

- **Local State with Hooks**: For component-specific state, we use React's `useState` and `useReducer` hooks. This approach keeps state close to where it's used and makes components easier to understand and test.
- **Custom Hooks for Complex Logic**: When state logic becomes complex or needs to be shared across components, we extract it into custom hooks. This promotes reusability and keeps components focused on rendering.

3.7.1 State Organization Principles

- Keep state as local as possible to the components that need it
- Use custom hooks to share stateful logic between components
- Prefer composition over complex state management libraries for most use cases
- Consider using Context API only for truly global state that many components need
- Always include error states in your state management

3.8. Code Structure and Formatting

- **Indentation and Spacing**: We use 2-space indentation consistently throughout the frontend codebase. This aligns with React ecosystem standards and improves readability in deeply nested JSX structures.
- **Single Statement Principle**: Each line should contain only one statement or operation. This makes code easier to read, debug, and maintain. It also makes version control diffs more meaningful.
- **Method Chaining**: When chaining methods, break lines appropriately to maintain readability. Each method in a chain should typically be on its own line with proper indentation.

3.8.1 Comment Guidelines

- Use single-line comments for brief explanations or TODOs
- Write multi-line comments using JSDoc format for functions and complex logic
- Avoid obvious comments that simply restate what the code does
- Focus comments on explaining why something is done, not what is being done

3.9. Error Handling

Comprehensive error handling is crucial for providing a good user experience and maintaining application stability.

- **Error Boundaries**: Implement error boundaries to catch and handle React component errors gracefully. This prevents the entire application from crashing when individual components encounter errors.
- **Async Operation Handling**: Always wrap async operations in try-catch blocks. Handle different types of errors appropriately and provide meaningful feedback to users.
- **Error State Management**: Include error states in your component state management. Components should be able to display loading, success, and error states appropriately.
- **User-Friendly Messages**: Convert technical error messages into user-friendly language. Users shouldn't see raw API errors or technical jargon.
- **Error Logging and Monitoring**: Log errors with sufficient context for debugging while avoiding logging sensitive user information. In production, integrate with monitoring services to track and respond to errors proactively.
- **Error Propagation**: Handle errors at the appropriate level. Some errors should be handled locally, while others need to be propagated up to parent components or global error handlers.

4. Backend Standards (Python/FastAPI)

4.1. File Structure and Naming

```
backend/
|-- docker-compose.yml                  # Service orchestration
|-- Makefile                            # Build and deployment commands
|-- mypy.ini                            # MyPy configuration
|-- pyproject.toml                      # Project configuration
|-- requirements.txt                    # Common dependencies
|-- run_tests.sh                        # Test runner script
|-- run_tests_linux.sh                  # Linux test runner script
|-- local_mavito_fallback.db            # Local SQLite database
|
|-- mavito-common-lib/                  # Shared utilities and models
|   |-- pyproject.toml                  # Package configuration
|   |-- mavito_common/                  # Common package
|   `-- mavito_common_lib.egg-info/     # Package metadata
|
|-- alembic-service/                    # Database migration service
|   |-- Dockerfile                      # Container definition
|   |-- entrypoint.sh                   # Service entry script
|   |-- pyproject.toml                  # Service configuration
|   `-- requirements.txt                # Service dependencies
|
|-- analytics-service/                  # Analytics and reporting
|   |-- Dockerfile                      # Container definition
|   |-- pyproject.toml                  # Service configuration
|   |-- requirements.txt                # Service dependencies
|   |-- app/                            # Service code
|   `-- Mock_Data/                      # Test data
|
|-- auth-service/                       # Authentication service
|   |-- Dockerfile                      # Container definition
|   |-- pyproject.toml                  # Service configuration
|   |-- requirements.txt                # Service dependencies
|   |-- sa-key.json/                    # Service account key
|   |-- scripts/                        # Utility scripts
|   |-- Mock_Data/                      # Test data
|   `-- app/                            # Service code
|       |-- api/                        # API endpoints
|       |-- crud/                       # Database operations
|       |-- tests/                      # Service tests
|       |-- utils/                      # Utility functions
|       |-- main.py                     # Service entry point
|       `-- __init__.py                 # Package initialization
|
|-- comment-service/                    # Comment handling service
|   |-- Dockerfile                      # Container definition
|   |-- pyproject.toml                  # Service configuration
|   |-- requirements.txt                # Service dependencies
|   |-- app/                            # Service code
|   `-- __init__.py                     # Package initialization
|
|-- feedback-service/                   # Feedback handling service
|   |-- Dockerfile                      # Container definition
|   |-- pyproject.toml                  # Service configuration
```

```
|   |-- requirements.txt                # Service dependencies
|   `-- app/                            # Service code
|
|-- glossary-service/                   # Glossary management
|   |-- Dockerfile                      # Container definition
|   |-- pyproject.toml                  # Service configuration
|   |-- requirements.txt                # Service dependencies
|   |-- scripts/                        # Utility scripts
|   |-- Mock_Data/                      # Test data
|   `-- app/                            # Service code
|
|-- linguist-application-service/       # Linguist application handling
|   |-- Dockerfile                      # Container definition
|   |-- pyproject.toml                  # Service configuration
|   |-- requirements.txt                # Service dependencies
|   |-- sa-key.json/                    # Service account key
|   |-- app/                            # Service code
|   `-- __init__.py                     # Package initialization
|
|-- search-service/                     # Search functionality
|   |-- Dockerfile                      # Container definition
|   |-- pyproject.toml                  # Service configuration
|   |-- requirements.txt                # Service dependencies
|   |-- app/                            # Service code
|   `-- Mock_Data/                      # Test data
|
|-- vote-service/                       # Voting system
|-- workspace-service/                  # Workspace management
|
|-- migrations/                         # Database migrations
|   |-- alembic.ini                     # Alembic configuration
|   `-- alembic/                        # Alembic migrations
|
|-- Mock_Data/                          # Development data
|   |-- communityActivity.json          # Community activity data
|   |-- fakeUsers.py                    # User generation script
|   |-- multilingual_statistical_terminology_clean.json
|   |-- recentTerms.json                # Recent terms data
|   |-- termData.json                   # Term definitions
|   `-- users.json                      # Sample users
|
`-- dsfsi_sa_key.json/                  # Service account key
```

4.2. Naming Conventions

Our backend follows Python's PEP 8 naming conventions with additional project-specific guidelines:

- **Files**: Use snake_case for all Python files (`crud_search.py`, `main.py`). This follows Python conventions and improves readability.
- **Classes**: Use PascalCase for class names (`Term`, `TermVote`). This distinguishes classes from functions and variables.
- **Functions**: Use snake_case for all function names (`search_terms_in_db`, `determine_term_langua` This aligns with Python standards and improves code consistency.

- **Variables**: Use snake_case for all variables (`user_data`, `is_active`, `email_address`). This creates consistency across the codebase.
- **Constants**: Use UPPER_SNAKE_CASE for constants (`MAX_RETRY_COUNT`, `DEFAULT_TIMEOUT`, `API_VERSION`). This makes constants easily identifiable.

4.3. Python Code Standards

Our Python codebase emphasizes type safety, clear documentation, and modern async/await patterns for optimal performance.

- **Type Hints**: Type hints are mandatory for all function parameters and return values. This improves code reliability, enables better IDE support, and makes the codebase more maintainable as it grows.
- **Documentation**: Every public function and class must include comprehensive docstrings following Google or NumPy docstring format. This serves as both documentation and helps with API generation.
- **Async Programming**: All database operations and external API calls use async/await patterns to prevent blocking and improve application performance under load.
- **Exception Handling**: Use specific exception types rather than catching generic exceptions. This enables more targeted error handling and better debugging information.
- **Code Organization**: Use dataclasses or Pydantic models for structured data. This provides automatic validation, serialization, and clear data contracts between different parts of the application.

4.4. FastAPI Standards

Our FastAPI implementation follows RESTful principles and emphasizes clear, self-documenting APIs with comprehensive error handling.

- **Route Organization**: Group related routes using APIRouter to maintain clean separation of concerns. This makes the codebase more modular and easier to maintain as the application grows.
- **Response Models**: Always specify explicit response models using Pydantic schemas. This ensures API responses are consistent, well-documented, and automatically validated.
- **Status Codes**: Use appropriate HTTP status codes for different scenarios. This helps clients understand the outcome of their requests and handle responses appropriately.
- **Dependency Injection**: Leverage FastAPI's dependency injection system for database sessions, authentication, and other shared resources. This promotes loose coupling and makes testing easier.
- **API Documentation**: FastAPI automatically generates API documentation, but ensure your route functions have clear docstrings and properly typed parameters to enhance the generated docs.

4.5. Database Models

Our database models use SQLAlchemy's modern declarative approach with proper type annotations and relationships.

- **Modern SQLAlchemy Syntax**: We use SQLAlchemy 2.0+ with `Mapped` type annotations for better type safety and IDE support. This provides clearer model definitions and better error detection.
- **UUID Primary Keys**: All models use UUID primary keys for better security and to avoid enumeration attacks. UUIDs also make it easier to merge data from different sources.
- **Proper Indexing**: Add database indexes on frequently queried columns, especially foreign keys and columns used in WHERE clauses. This significantly improves query performance.
- **Timestamps**: Include created_at and updated_at timestamps on all models to track data lifecycle. Use timezone-aware timestamps for consistency across different deployment environments.
- **Nullable vs Non-Nullable**: Be explicit about which fields can be null. This prevents data integrity issues and makes the schema requirements clear to other developers.

4.6. Code Quality Tools

Our backend development workflow integrates multiple tools to ensure code quality, type safety, and consistency.

- **Ruff**: Used for fast Python linting and code analysis. Ruff combines the functionality of multiple tools (flake8, isort, etc.) into a single, fast linter that catches common Python issues.
- **Black**: Handles automatic code formatting to ensure consistent style across the team. Black's opinionated formatting removes debates about code style and ensures consistency.
- **MyPy**: Performs static type checking to catch type-related errors before runtime. This is especially important in a large codebase where type safety prevents many common bugs.
- **Pytest**: Our testing framework for unit tests, integration tests, and API endpoint testing. Pytest's fixture system and async support make it ideal for testing FastAPI applications.

4.6.1 Tool Integration

All tools are configured in `pyproject.toml` and integrated into the development workflow through pre-commit hooks and CI/CD pipelines.

4.7. Code Structure and Formatting

- **Indentation**: Python uses 4-space indentation as specified in PEP 8. This is enforced by Black and provides good readability for nested code structures.
- **Line Length**: We follow the 88-character line limit (Black's default) which is slightly longer than PEP 8's 79 characters but works well with modern displays.
- **Function Organization**: Keep functions focused on single responsibilities. Long functions should be broken down into smaller, more focused functions that are easier to test and understand.
- **Import Organization**: Imports are automatically organized by tools like Ruff, with standard library imports first, followed by third-party imports, then local imports.

4.7.1 Documentation Standards

Use Google-style docstrings for consistency. Include parameter types, return types, and raise information for public APIs.

4.8. Error Handling

Robust error handling is essential for maintaining service reliability and providing meaningful feedback to clients.

- **Exception Hierarchy**: Create a custom exception hierarchy for your application. This allows for more specific error handling and better error categorization.
- **Database Operations**: Always wrap database operations in try-catch blocks with proper rollback mechanisms. Database errors should be handled gracefully without leaving the database in an inconsistent state.
- **Logging Strategy**: Implement structured logging with appropriate log levels. Include relevant context in log messages but avoid logging sensitive information like passwords or tokens.
- **Error Propagation**: Handle errors at the appropriate level. Some errors should be handled locally, while others need to be propagated to calling functions with additional context.
- **User-Facing Errors**: Convert internal errors into user-friendly messages. API responses should provide helpful error information without exposing internal system details.
- **Error Codes**: Use standardized error codes to help clients handle different error conditions programmatically.

5. Database Standards

5.1. Schema Design Principles

Our database design emphasizes clarity, performance, and data integrity through thoughtful schema organization.

- **Table Naming**: Use clear, descriptive table names that immediately convey their purpose. Prefer full words over abbreviations to improve readability and reduce confusion for new team members.
- **Primary Keys**: All tables use UUID primary keys for enhanced security and to prevent enumeration attacks. UUIDs also facilitate data distribution and merging from multiple sources.
- **Indexing Strategy**: Create indexes on frequently queried columns, especially those used in WHERE clauses, JOIN conditions, and ORDER BY clauses. Monitor query performance and add indexes as needed based on actual usage patterns.
- **Data Types**: Choose appropriate data types for each column. Use VARCHAR with reasonable length limits for text fields, and always specify precision for decimal numbers.
- **Constraints**: Implement database-level constraints (NOT NULL, UNIQUE, CHECK) to ensure data integrity at the database level, providing a safety net beyond application-level validation.

5.2. Migration Standards

Database migrations are managed through Alembic to ensure consistent schema changes across all environments.

- **Migration Naming**: Use descriptive names that clearly indicate what the migration does. Include a sequence number or timestamp to maintain order.
- **Reversible Migrations**: Always implement both upgrade and downgrade functions. This allows for safe rollbacks if issues are discovered after deployment.
- **Data Migrations**: When migrations involve data transformation, ensure they handle large datasets efficiently and include proper error handling for data inconsistencies.
- **Testing Migrations**: Test migrations on a copy of production data to ensure they work correctly with real data volumes and edge cases.
- **Documentation**: Include clear comments in migration files explaining complex changes, especially those that might affect application behavior.

6. API Standards

6.1. RESTful API Design

Our API design follows REST principles to create intuitive, predictable interfaces for client applications.

- **Resource-Based URLs**: Design URLs around resources rather than actions. Use nouns for resource names and leverage HTTP methods to indicate the action being performed.
- **HTTP Methods**: Use appropriate HTTP methods consistently:
  - GET for retrieving data (safe and idempotent)
  - POST for creating new resources
  - PUT for updating entire resources (idempotent)
  - PATCH for partial updates
  - DELETE for removing resources (idempotent)

6.1.1 API Endpoints

Use descriptive, RESTful endpoint patterns that clearly communicate the resource and action:

```
#     Good - actual endpoints from glossary-service
@router.get("/categories", response_model=List[str])
@router.get("/categories/{category_name}/terms")
@router.get("/terms/{term_id}/translations")
@router.get("/search")
@router.post("/translate")
@router.get("/stats")

#     Bad - unclear or non-RESTful
@router.get("/get_user")
@router.post("/do_translation")
@router.put("/update")
```

- **URL Structure**: Follow a logical hierarchy in URL structure. Use nested resources when there's a clear parent-child relationship, but avoid deep nesting (max 2-3 levels).
- **Versioning**: Include API version in the URL path to enable backward compatibility as the API evolves. Start with v1 and increment for breaking changes.
- **Query Parameters**: Use query parameters for filtering, sorting, pagination, and other optional modifications to the base resource request.

6.2. Response Format Standards

All API responses follow a consistent structure to provide predictable interfaces for client applications.

- **Success Responses**: Include the requested data in a `data` field, along with metadata about the response. This structure makes it easy for clients to extract the actual content while also accessing response metadata.
- **Metadata**: Include response metadata such as timestamps, API version, and pagination information where applicable. This helps with debugging and provides context about the response.
- **Content-Type**: Always return appropriate Content-Type headers. Use `application/json` for JSON responses and ensure character encoding is specified.
- **Response Consistency**: Maintain consistent field naming and structure across all endpoints. Use the same patterns for similar operations (e.g., all list endpoints should have similar pagination structures).

6.3. Error Response Standards

Error responses provide clear, actionable information to help clients handle and recover from errors appropriately.

- **Error Structure**: Use a consistent error response structure that includes error codes, human-readable messages, and additional details when helpful.
- **Error Codes**: Implement standardized error codes that clients can use programmatically to handle different error conditions. This enables better error handling logic in client applications.
- **HTTP Status Codes**: Use appropriate HTTP status codes to categorize errors:
  - 400 for client errors (bad request, validation errors)
  - 401 for authentication errors
  - 403 for authorization errors
  - 404 for resource not found
  - 500 for server errors
- **Error Details**: Provide enough detail to help developers debug issues without exposing sensitive system information. Include field-specific validation errors when applicable.

7. Testing Standards

7.1. Frontend Testing Strategy

Our frontend testing approach ensures component reliability, user experience quality, and prevents regressions.

- **Unit Testing with Vitest**: We use Vitest for fast unit testing of individual components and utility functions. Vitest provides excellent TypeScript support and integrates well with our build system.

- **Component Testing**: Focus on testing component behavior rather than implementation details. Test user interactions, state changes, and prop handling. Use React Testing Library's approach of testing how users interact with components.
- **Mock Strategy**: Create focused mocks for external dependencies like API calls. Use dependency injection patterns to make components more testable.

7.1.1 Test Organization

Structure tests to mirror the source code organization. Keep test files close to the code they test for better maintainability. Our current test structure includes:

- **Frontend Tests**:
  - `LoginPage.test.tsx` - Tests login functionality and form validation
  - `SearchPage.test.tsx` - Tests search interface and results display
  - `GlossaryPage.test.tsx` - Tests glossary browsing and term display
  - `Dashboard.test.tsx` - Tests dashboard layout and data visualization
  - `AdminPage.test.tsx` - Tests administrative functions and permissions
- **Backend Tests**:
  - `test_auth.py` - Tests authentication endpoints and security
  - `test_search.py` - Tests search functionality and term retrieval
  - `test_glossary.py` - Tests glossary management operations
  - `test_analytics_unit.py` - Tests analytics calculation functions
  - `test_vote_endpoint.py` - Tests voting system endpoints
- **Test Coverage**: Aim for meaningful test coverage rather than just high percentage coverage. Focus on testing critical user flows and edge cases.

7.2. Backend Testing Strategy

Our backend testing ensures API reliability, data integrity, and business logic correctness.

- **Async Testing with Pytest**: We use Pytest with async support for testing FastAPI endpoints and database operations. The async testing capabilities handle our async/await codebase effectively.
- **Test Database**: Use a separate test database or in-memory database for testing to avoid affecting development or production data. Reset the database state between tests to ensure test isolation.
- **API Endpoint Testing**: Test all API endpoints with various input scenarios including valid data, invalid data, edge cases, and error conditions.
- **Integration Testing**: Test the integration between different layers (API, business logic, database) to ensure they work together correctly.
- **Test Fixtures**: Use Pytest fixtures to set up test data and dependencies. This promotes test reusability and maintains clean test code.

7.3. Testing Strategy

- **Unit Tests**: Test individual functions and components
- **Integration Tests**: Test API endpoints and database interactions
- **E2E Tests**: Test complete user workflows
- **Coverage Target**: Minimum 60% code coverage

8. Version Control Standards

8.1. Branch Management Strategy

Our branching strategy supports parallel development while maintaining code quality and release stability.

- **Main Branch**: The `main` branch always contains production-ready code. All code in main should be deployable at any time.
- **Development Branch**: The `develop` branch serves as the integration branch for ongoing development. Feature branches are merged here first for integration testing.
- **Feature Branches**: Use descriptive names that clearly indicate the feature being developed. Include the type of work (feature, bugfix, hotfix) and a brief description.
- **Release Branches**: Create release branches for preparing new versions. This allows for final testing and bug fixes without blocking ongoing development.
- **Hotfix Branches**: For critical production fixes that can't wait for the next regular release cycle.

8.2. Commit Message Standards

Clear, consistent commit messages improve project history readability and help with automated tooling.

- **Conventional Commits**: We follow the Conventional Commits specification for consistent commit messages. This enables automated changelog generation and semantic versioning.

8.2.1 Commit Types

- `feat`: New features or functionality
- `fix`: Bug fixes and error corrections
- `docs`: Documentation changes
- `style`: Code formatting and style changes
- `refactor`: Code restructuring without changing functionality
- `test`: Adding or updating tests

- `chore`: Maintenance tasks and dependency updates

- **Message Structure**: Use a clear, concise description in the imperative mood. Include additional context in the body when necessary for complex changes.

- **Scope**: Optionally include a scope to indicate which part of the codebase is affected (e.g., `feat(auth): add password reset functionality`).

8.3. Pull Request Guidelines

Pull requests are our primary mechanism for code review and ensuring quality before changes reach the main codebase.

- **PR Description**: Provide clear descriptions of what changes were made and why. Include context that will help reviewers understand the purpose and scope of the changes.

- **Change Categories**: Clearly indicate whether the PR contains bug fixes, new features, breaking changes, or documentation updates. This helps reviewers understand the impact and priority.

- **Testing Requirements**: All PRs must include appropriate tests for the changes made. Manual testing should also be documented when automated tests are insufficient.

- **Review Process**: All code must be reviewed by at least two other team members before merging. Reviews should focus on functionality, code quality, security, and adherence to these coding standards.

9. CI/CD Standards

9.1. Pipeline Philosophy

Our CI/CD approach emphasizes automated quality checks, security scanning, and reliable deployment processes through our GitHub Actions workflow (`actions.yml`).

- **Continuous Integration**: Our workflow triggers on pushes to `main`, `master`, and `develop` branches, as well as pull requests. It includes:
  - Backend pipeline with PostgreSQL service containers for testing
  - Automated testing and linting for both frontend and backend
  - Quality gates that must pass before code can be merged

- **Environment Strategy**: Our pipeline uses consistent PostgreSQL containers for testing with proper health checks to ensure database readiness before running tests.

- **Automated Testing**: Each service (auth-service, search-service, glossary-service, analytics-service, vote-service, linguist-application-service) is tested independently with their respective test suites.

9.2. Quality Assurance Requirements

All code changes must meet specific quality criteria before being deployed to production.

- **Code Quality Metrics**:
  - All linting checks must pass with zero warnings or errors
  - All automated tests must pass without exceptions
  - Security scans must show no high or critical severity vulnerabilities
- **Review Requirements**: Every change must be reviewed by at least two other team members who understand the affected code area. Reviews should focus on functionality, security, performance, and maintainability.
- **Testing Standards**: New features must include appropriate unit tests, integration tests, and documentation. Bug fixes should include regression tests to prevent the issue from recurring.
- **Performance Monitoring**: Monitor application performance metrics and ensure changes don't negatively impact response times, memory usage, or other critical performance indicators.

9.3. Deployment and Containerization

Our deployment strategy uses Docker containers for consistency and reliability across environments.

- **Container Strategy**: Use multi-stage Docker builds to optimize image size and build times. Separate build dependencies from runtime dependencies to create lean production images.
- **Environment Configuration**: Use environment variables for configuration that differs between environments. Never hardcode environment-specific values in the application code.
- **Health Checks**: Implement proper health check endpoints for all services. This enables load balancers and orchestration systems to manage service health automatically.
- **Logging and Monitoring**: Ensure all services produce structured logs that can be aggregated and analyzed. Implement metrics collection for monitoring service performance and health.
- **Security**: Run containers with minimal privileges and regularly update base images to address security vulnerabilities. Scan images for known vulnerabilities before deployment.

10. Security Standards

10.1. Authentication and Authorization

Security starts with robust authentication and authorization mechanisms that protect user data and system resources.

- **Authentication Strategy**: Implement strong authentication using industry-standard protocols. Use JWT tokens with appropriate expiration times and refresh token mechanisms for session management.
- **Password Security**: Enforce strong password requirements and use secure hashing algorithms (bcrypt, Argon2) for password storage. Never store passwords in plain text or use weak hashing algorithms.
- **Authorization Models**: Implement role-based access control (RBAC) to ensure users can only access resources they're authorized to use. Use the principle of least privilege when assigning permissions.
- **Session Management**: Implement secure session management with proper token expiration, refresh mechanisms, and logout functionality that invalidates sessions on the server side.

10.2. Input Validation and Data Security

Comprehensive input validation prevents many security vulnerabilities and ensures data integrity.

- **Server-Side Validation**: Always validate input on the server side, even if client-side validation exists. Client-side validation can be bypassed, so server-side validation is the security boundary.
- **Data Sanitization**: Sanitize user input to prevent injection attacks (SQL injection, XSS, etc.). Use parameterized queries for database operations and escape output appropriately.
- **Schema Validation**: Use schema validation libraries (Pydantic for Python, Joi for Node.js) to define and enforce data structure requirements. This provides both security and API contract enforcement.
- **File Upload Security**: If the application handles file uploads, implement strict validation of file types, sizes, and content. Scan uploaded files for malware and store them in secure locations.
- **Rate Limiting**: Implement rate limiting to prevent abuse and DoS attacks. Apply limits at both the API level and for specific operations like login attempts.

10.3. Security Best Practices

Implement comprehensive security measures to protect against common vulnerabilities and attack vectors.

- **HTTPS Everywhere**: Use HTTPS for all communications in production. Implement HTTP Strict Transport Security (HSTS) headers to prevent downgrade attacks.
- **Environment Security**: Never commit sensitive information like API keys, database passwords, or encryption keys to version control. Use environment variables or secure secret management systems.
- **Dependency Management**: Regularly update dependencies to patch known security vulnerabilities. Use automated tools to scan for vulnerable dependencies and prioritize security updates.
- **Error Handling**: Implement secure error handling that doesn't expose sensitive system information to users. Log detailed errors for debugging but return generic error messages to clients.
- **Security Headers**: Implement appropriate security headers (Content Security Policy, X-Frame-Options, etc.) to protect against common web vulnerabilities like XSS and clickjacking.
- **Audit Logging**: Log security-relevant events like authentication attempts, authorization failures, and administrative actions for security monitoring and incident response.

11. Conclusion

These coding standards ensure consistency, maintainability, and quality across the Marito project. All team members must follow these guidelines, and code reviews should verify adherence to these standards. The standards should be regularly reviewed and updated as the project evolves.

For questions or suggestions regarding these standards, please create an issue in the project repository or discuss in team meetings.

Last updated: 13 July 2025
Version: 1.0