AEGIS Development Standards

1. Coding Conventions

1.1 Naming Conventions

We follow clear and descriptive naming practices to improve readability and maintainability across the codebase.

Variables

Use `camelCase` for variable names:

```
caseId := uuid.New()
evidenceList := []Evidence{}
```

Constants

Use `SNAKE_CASE` for constants:

```
const MAX_RETRIES = 5
const API_TIMEOUT_MS = 3000
```

Functions

Function names follow `camelCase` and clearly describe their purpose:

```
func generateReport(caseId uuid.UUID) error {...}
func validateInput(input string) bool {...}
```

-

Files

General Go files: `snake_case`

```
evidence_service.go
case_repository.go
```

-

React components: `PascalCase`

```
 EvidenceCard.tsx
CaseTable.tsx
```

-

1.2 Formatting

- Use **4 spaces** for indentation.
- Stick to **one statement per line**. Avoid grouping variables or key-value pairs unless trivial.
- Prettier is enforced in the CI pipeline to maintain consistent formatting.

Example: Correct formatting

```
const userName = "Retshepile";
const caseStatus = "open";

function createEvidence(evidenceId: string): void {
    console.log(`Creating evidence ${evidenceId}`);
}
```

Example: Avoid this

```
const userName = "Retshepile", caseStatus = "open";
```

1.3 Comments

- Keep comments **short and focused**.
- Use `//` for single-line comments and `/* */` for multi-line explanations.
- Comments should explain **why** something exists, not **what** the code does (the code should be self-explanatory).

Example

```
// Fetch all evidence associated with a case before creating a
report
evidenceList := getEvidenceByCase(caseId)
```

2. Error Handling

- Detect and handle errors **early**.
- Return or log **clear, actionable messages**.
- Recover gracefully when possible (clean state, notify user, no crash).

Example

```
file, err := os.Open(filePath)
if err != nil {
    log.Printf("Failed to open evidence file %s: %v", filePath, err)
    return err
}
```

3. Testing

3.1 Test Types

We test at multiple levels to ensure confidence across the system:

Unit Tests

- Test individual components and logic.
- Use Testify for Go: test files follow the `*_test.go` pattern.

```
func TestGenerateReport(t *testing.T) {...}
```

Integration Tests

- Test how modules work together.
- Run using Cypress.

End-to-End (E2E) Tests

- Simulate full user flows in the app.
- Run using Cypress.

3.2 Coverage

- Aim for **80%+ coverage** on critical modules.
- See the Testing Specification document for detailed guidance.

4. Git Workflow

4.1 Branching Strategy

We follow **GitFlow** for structured development:

Primary Branches

- `main`: Production-ready code. No direct commits.
- `develop`: Integration branch for completed features.

Supporting Branches

- `feature/*`: New features. Branch off `develop`, merge back into `develop`.
- `hotfix/*`: Urgent fixes. Branch off `main`, merge into both `main` and `develop`.
- `config/*`: Infrastructure, CI/CD, or project-wide config changes.

4.2 Branch Naming Conventions

- Use **lowercase, hyphenated** names that describe the purpose.
  Examples

```
feature/user-authentication
hotfix/fix-logout-crash
config/update-eslint-rules
```

- Avoid vague or overly long names. Stick to alphanumeric characters and single hyphens.

5. Code Reviews and CI

5.1 Pull Requests

- New work starts from `develop` and merges back once complete.
- Pull requests trigger **automatic linting** checks.
- Code must **pass linting and tests** before review.

5.2 Linting

- We use **ESLint** for JavaScript/TypeScript code.
- Rules match the coding conventions above.
- Linting runs automatically in **GitHub Actions**.

Example of a failing lint error

```
error  'userName' is defined but never used  no-unused-vars
```