DEVX360

Sipho Sehlapelo
Kesley Hamann
Sibusiso Mngomezulu
Lwando Msindo
David Musa-Aisien

DevX360

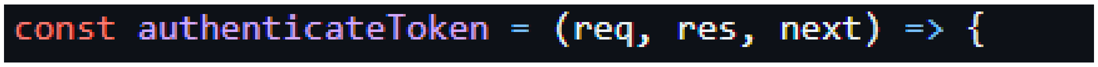
Table of Contents

Coding Standards

These standards ensure a consistent, maintainable, and collaborative codebase across all team members. By aligning on naming, formatting, and structure, we reduce bugs, speed up onboarding, and improve long-term maintainability.

Naming Conventions: These are the set of rules to be used for choosing the names of functions, variables and other entities in code.

- Functions, variables, and file names use camelCase.
  - The first letter of the entity name will begin with a small letter. Every logical word afterwards in the variable name will start with a capital letter.

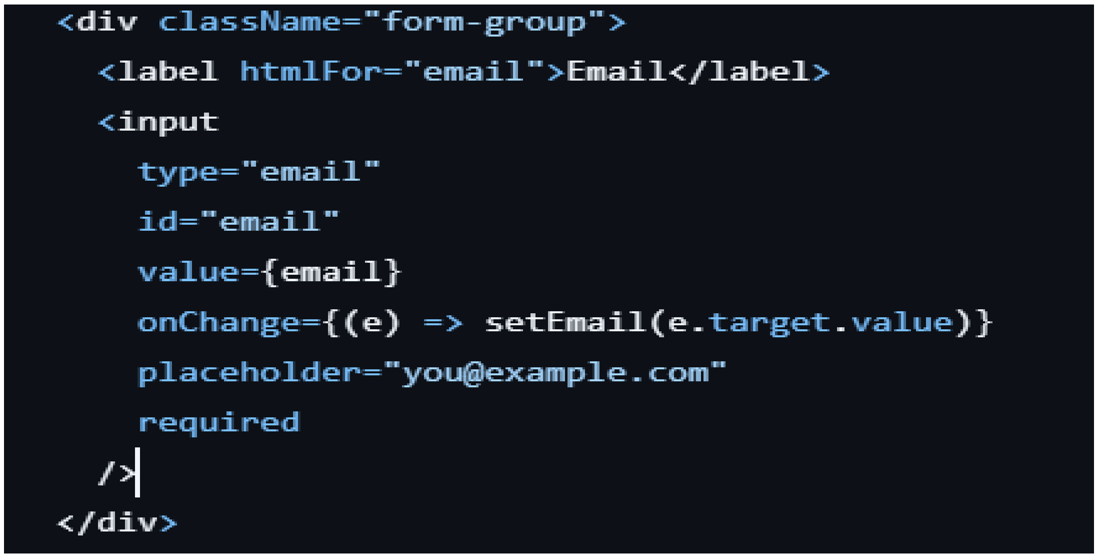


```
const authenticateToken = (req, res, next) => {
```

- React components use PascalCase.
  - The first letter of each compound word in a variable is capitalized.

- EXAMPLE: HandleError, VariableName etc.

- React class names use kebab-case.
  - Each word in the variable name is separated by a hypen symbol as shown below:

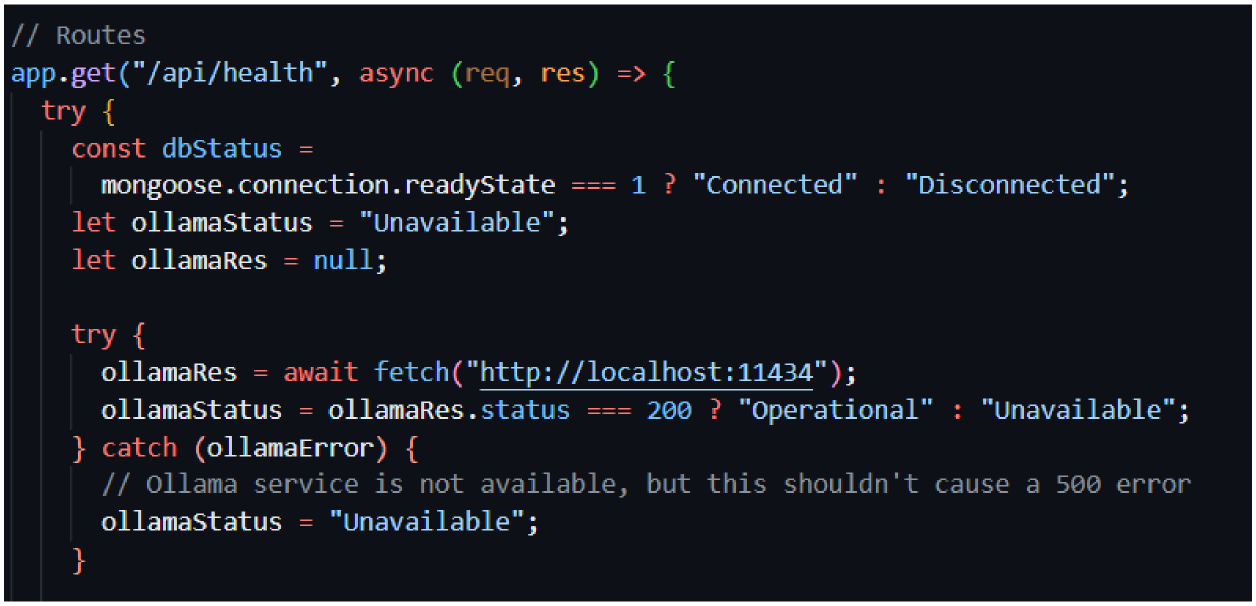


```
<div className="form-group">
  <label htmlFor="email">Email</label>
  <input
    type="email"
    id="email"
    value={email}
    onChange={(e) => setEmail(e.target.value)}
    placeholder="you@example.com"
    required
  />
</div>
```

- Descriptive function and file names used across all components.
- Assigned unique names to each used variable. As stated before the variable would have a descriptive/meaningful name and we would avoid re-assigning this variable and re-using it for multiple functions.

Comments

- Meaningful, descriptive comments are added and will serve as a form of documentation within the code, explaining the logic, functionality and/or purpose of specific code sections. This can break down complex sections of code into understandable pieces of information. This will improve the ease of integration of our code as well as, to an extent, improve the accessibility of our code to untrained users.
- Comments documenting the main purpose of a function will be placed just directly above or along-side the function definition as shown below:



```
// Routes
app.get("/api/health", async (req, res) => {
  try {
    const dbStatus =
      mongoose.connection.readyState === 1 ? "Connected" : "Disconnected";
    let ollamaStatus = "Unavailable";
    let ollamaRes = null;

    try {
      ollamaRes = await fetch("http://localhost:11434");
      ollamaStatus = ollamaRes.status === 200 ? "Operational" : "Unavailable";
    } catch (ollamaError) {
      // Ollama service is not available, but this shouldn't cause a 500 error
      ollamaStatus = "Unavailable";
    }
```

Code Review

- Various software such as ESLint and Prettier as well manual methods such as peer review will be used for Quality Assurance of our code enforcing industry standards as well readable, sensible formatting of our code. All significant pull requests (determined by internal discussion) require approval from at least one team member.

Test Coverage Standards

- We maintain a minimum of 80% unit test coverage across all modules. Jest is used as the test runner, and code is not merged until tests pass via CI.
- All code will be thoroughly tested using a combination of unit, integration and e2e tests.
- Tests will be archived in the _tests_ folder in their respective branches on the repository: Frontend tests will be stored in the “feature/frontend” branch. Backend tests will be stored in the “feature/ai-analysis” branch.

Formalized Exception Handling

- Code in the API service routes was kept within try-catch blocks with appropriate responses to errors being added.
- All backend service routes are enclosed in try-catch blocks.
- Meaningful and consistent error messages are returned with appropriate HTTP status codes.

Security and Privacy Consideration

• **Sensitive Data Transmission**

Only the minimum required sensitive information is extracted and transmitted. Where possible, sensitive information is not sent over the internet. If transmission is necessary, it is encrypted using up-to-date protocols (e.g., HTTPS/TLS).

• **Storage of Sensitive Information**

Sensitive data is stored securely with hashing, salting, and peppering techniques where applicable. We use strong, modern algorithms (e.g., BLAKE2b, SHA-256) to mitigate risks of brute-force and rainbow table attacks.

• **Secret Management**

Tokens, API keys, and other secrets are never hardcoded in the codebase. Instead, they are stored securely in environment variables (.env files) or managed via secret storage solutions. These files are excluded from version control (e.g., .gitignore) to prevent accidental leaks.

• **Role-Based Access Control (RBAC)**

Access to system features is managed through well-defined roles to enforce the principle of least privilege:

- Admin: Full control of the system, manages teams, oversees user activity, and has visibility into all metrics.
- Team Leader: Responsible for managing a specific team, creating projects, and monitoring team-level performance metrics.
- Team Member: Has standard access, can join teams, view their own metrics, and contribute code, but does not have elevated permissions.

• **Code Backups and Recovery**

In the event of system crashes or outages, code is backed up both locally and on the cloud. Redundant backups across multiple systems ensure recoverability.

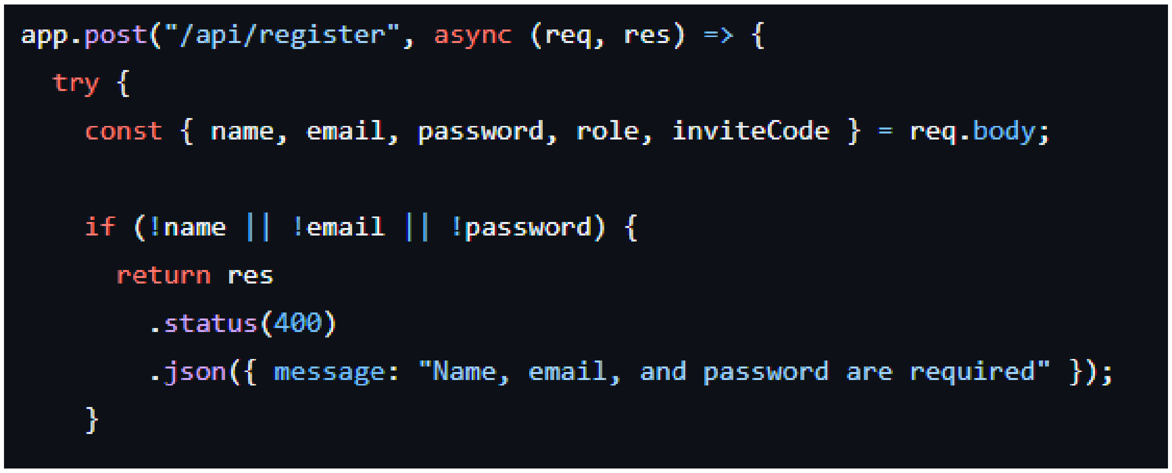
• **Auditing and Monitoring**

Repository activity (commits, PRs, and issues) is continuously tracked. Suspicious activity can be flagged through GitHub's built-in security features (e.g., Dependabot alerts, secret scanning).

• **Coding Style for Security**

Hierarchical indentation is followed to ensure readability, reducing the risk of introducing logic errors (which can lead to vulnerabilities). Each nested block (functions, loops, conditionals) is indented progressively.

This approach is applied consistently across JavaScript/React and SON/YAML files.



```
app.post("/api/register", async (req, res) => {
  try {
    const { name, email, password, role, inviteCode } = req.body;

    if (!name || !email || !password) {
      return res
        .status(400)
        .json({ message: "Name, email, and password are required" });
    }
```

Focus on Code Readability

Some simple practices were followed to ensure our code was kept as readable as possible:

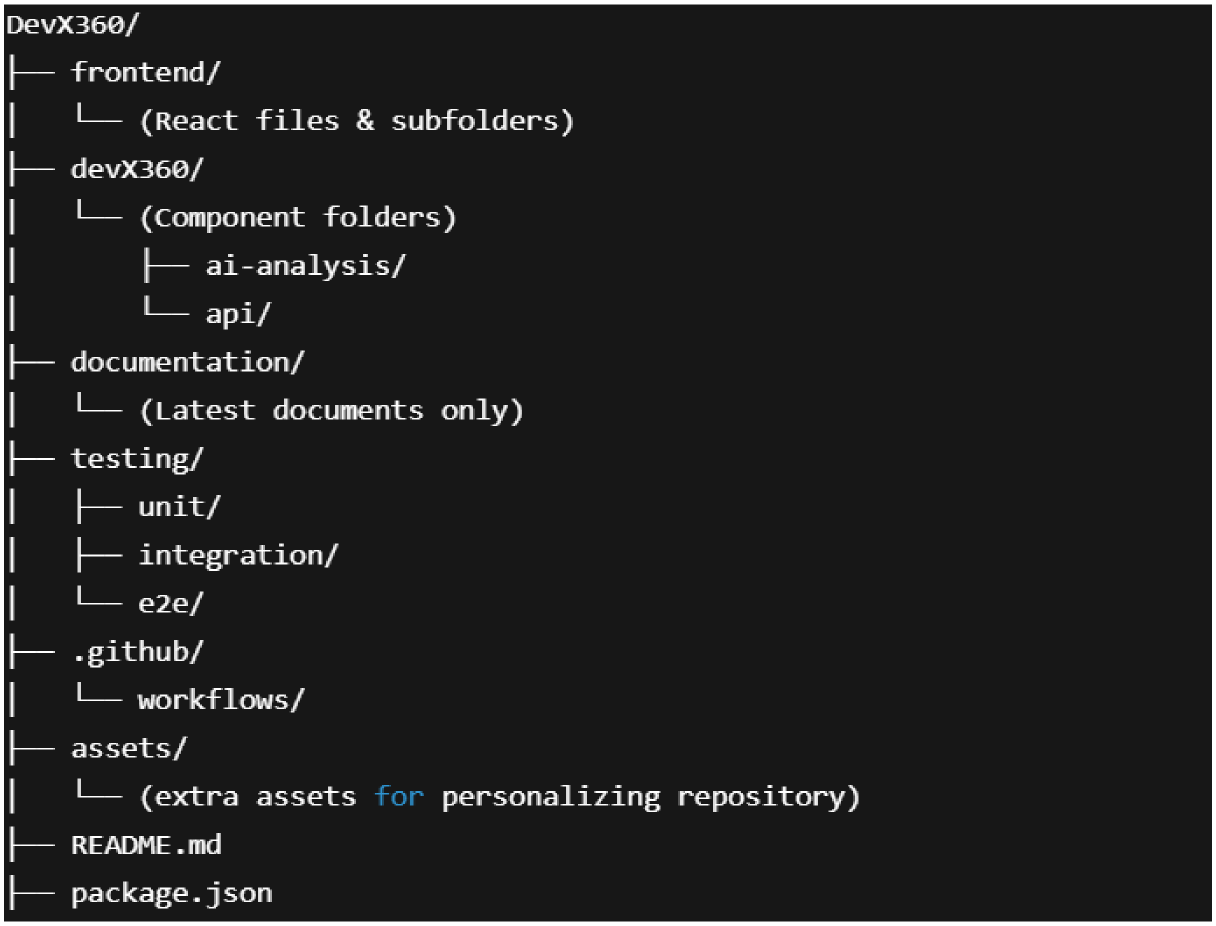
- Write as few lines of code as possible
- Segment related blocks of code into paragraphs
- Avoided the use of unnecessarily lengthy functions and unnecessary helper functions.
- Avoided repetition of code where possible
- Avoided the addition of too many nesting levels which could make code harder to follow

Documentation

- All formal documents will be stored in documentation folders on their respective branches.
- An archive of every document will be stored in a dedicated documentation branch named “feature/documentation”. This will be used for redundancy as well as traceability purposes.

File and Directory Structure

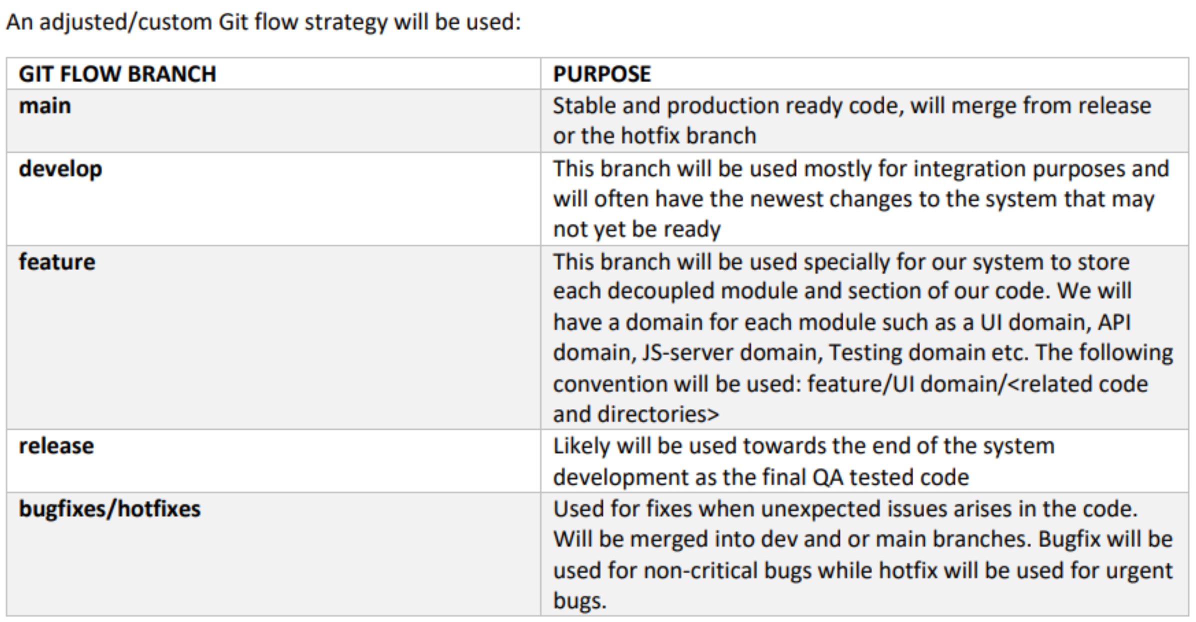
The main branch of our repository is structured as follows:



```
DevX360/
├── frontend/
│   └── (React files & subfolders)
├── devX360/
│   └── (Component folders)
│       ├── ai-analysis/
│       └── api/
├── documentation/
│   └── (Latest documents only)
├── testing/
│   ├── unit/
│   ├── integration/
│   └── e2e/
├── .github/
│   └── workflows/
├── assets/
│   └── (extra assets for personalizing repository)
├── README.md
├── package.json
```

Branching Strategy

The following branching strategy will be used, adopted from our Planning and Role Allocation document.


An adjusted/custom Git flow strategy will be used:

GIT FLOW BRANCH	PURPOSE
main	Stable and production ready code, will merge from release or the hotfix branch
develop	This branch will be used mostly for integration purposes and will often have the newest changes to the system that may not yet be ready
feature	This branch will be used specially for our system to store each decoupled module and section of our code. We will have a domain for each module such as a UI domain, API domain, JS-server domain, Testing domain etc. The following convention will be used: feature/UI domain/<related code and directories>
release	Likely will be used towards the end of the system development as the final QA tested code
bugfixes/hotfixes	Used for fixes when unexpected issues arises in the code. Will be merged into dev and or main branches. Bugfix will be used for non-critical bugs while hotfix will be used for urgent bugs.

Version Control Practices

- Commits must have a clear descriptive message
- Important commits must be reviewed by at least one team member
- Updates to commits are marked with a later version tag to track modification history.
- Small, focused commits. Keep commits atomic — one logical change per commit — so history is easy to read and revert.
- Simple branch names are used.