Need For Engineering Documentation

Table Of Contents

Problem Statement

Hyland's Technology organization is facing significant challenges due to the lack of standardized practices for engineering documentation. Engineers currently rely on ad hoc approaches, resulting in inconsistent and fragmented documentation that is difficult to locate and often varies in format, structure, and detail. This has created knowledge silos, where critical information resides with individuals rather than being accessible across the organization. Consequently, new hires struggle to onboard efficiently, and cross-functional teams, including solution engineers, services, and tech support, face challenges leveraging engineering outputs. The absence of automation and integration tools for documentation further exacerbates these issues, requiring manual effort and introducing inconsistencies. Additionally, poorly documented or undocumented systems slow developer productivity, as engineers must spend significant time interpreting existing code and systems. These challenges increase technical debt, hinder scalability, and slow time-to-market for new solutions. Addressing these issues by implementing clear standards, tools, and workflows for documentation would foster better collaboration, enhance developer productivity, and improve system maintainability.

Business Values in Creating Engineering Documentation

Investing in standardized engineering documentation early as Technology moves into our new value/work streams enhances operational efficiency and scalability. By addressing knowledge silos and improving accessibility, documentation reduces dependencies, accelerates onboarding, and supports collaboration across teams. It minimizes technical debt, simplifies maintenance, and fosters innovation by providing clarity around code and systems. Automating updates ensures documentation evolves with the code, creating a sustainable process that supports long-term growth. Additionally, high-quality documentation strengthens relationships with customers and partners while signaling a commitment to excellence, reinforcing your organization's ability to adapt and succeed.

Key Objectives Of Creating Engineering Documentation

- **Accelerate Development and Delivery:** Streamline workflows to reduce time spent interpreting code and systems, enabling faster delivery of features and products.
- **Enhance Knowledge Sharing and Collaboration:** Create accessible documentation to eliminate knowledge silos and improve cross-functional collaboration among engineering, solution engineers, services, and support teams.
- **Improve Onboarding Efficiency:** Provide clear and consistent documentation to reduce onboarding time for new team members, ensuring they become productive faster.
- **Automate Documentation Maintenance:** Integrate documentation generation and updates into CI/CD pipelines to ensure it evolves alongside code and remains reliable.
- **Support Long-Term Scalability:** Build processes and practices that grow with the organization, enabling sustained operational excellence and adaptability.

Expected Outcome To Achieve

- **Accelerate Development and Delivery**
  - **Outcome:** Decrease time spent on understanding existing code and systems.
  - **Metric:** Average time to complete new feature development decreases by 20% within six months.
- **Enhance Knowledge Sharing and Collaboration**
  - **Outcome:** Improve accessibility and usability of documentation across teams.
  - **Metric:** 90% of team members report finding relevant documentation in under five minutes during quarterly surveys.
- **Improve Onboarding Efficiency**
  - **Outcome:** Reduce time-to-productivity for new hires.
  - **Metric:** Average onboarding time decreases by 30% within the first year of implementation.
- **Automate Documentation Maintenance**
  - **Outcome:** Ensure documentation evolves with code changes and reduces manual effort.
  - **Metric:** 90% of repositories integrate automated documentation generation and updates within CI/CD pipelines within the first year.
- **Support Long-Term Scalability**
  - **Outcome:** Establish scalable processes that adapt as the organization grows.
  - **Metric:** 100% compliance with documentation standards across repositories within a year; measurable efficiency gains in team operations as the organization scales (e.g., onboarding, project throughput).

ReadMe Documents Learning Path:

General Path: Apply to all types of Tech Stacks

1. Read through parent readme document to understand the need of Engineering Documentation and Code Standards to successfully develop a project in line with Modern Development Practices for documentation.

2. Read through Guiding Principles For Engineering Documentation Standards to understand the principles which we have used to develop the projects present the repository.

3. Read through OpenAPI - An Introduction to understand key elements of OpenAPI.

4. Read through OpenAPI Documentation Standards - understanding of which is very important for creating OpenAPI documentations.

Tech Stack Specific Path:

1. Tech Stack: .NET 8
    - Read through Documentation Standards For Code Comments
    - Read through <a DotNet8ProjectInstallationInstruction.md">Project Information and Installation Instructions

🟥 **The Tech Stack section is still under process and will be updated as other tech stack projects are added**

OpenAPI - An Introduction

OpenAPI

OpenAPI is a specification for defining APIs (Application Programming Interfaces) that allows developers to describe their APIs in a standard, machine-readable format. The primary goal of OpenAPI is to create clear, concise, and easy-to-understand documentation for APIs. Here are the key elements and standards associated with OpenAPI documentation:

Key Elements Of OpenAPI

1. OpenAPI Specification (OAS) Versioning

- The OpenAPI specification is versioned, and each version comes with new features and improvements. The latest version (as of January 2025) is OpenAPI 3.1.
- You should always ensure your documentation aligns with the version you're using. OpenAPI 3.x has several improvements over OpenAPI 2.0 (Swagger), including better support for JSON Schema and additional fields.

2. Structure of OpenAPI Documentation OpenAPI documents are usually written in YAML or JSON formats and contain several key sections. A typical structure includes:

- openapi: The version of the OpenAPI specification used (e.g., "3.1.0"). info: Contains metadata about the API, such as its title, description, version, contact information, and license.
- servers: Describes the available API servers, including URLs and any relevant variables.
- paths: Defines the available API endpoints (URLs) and the supported HTTP methods (GET, POST, PUT, DELETE, etc.). Each endpoint has details on parameters, request bodies, responses, etc.
- components: Defines reusable elements, such as:
  - schemas: Data models (usually represented in JSON Schema format) used in requests and responses.
  - parameters: Reusable parameters for endpoints.
  - responses: Predefined responses that can be referenced across multiple endpoints.
  - securitySchemes: Authentication mechanisms like OAuth, API keys, etc.
- security: Specifies the security requirements for the API.
- tags: Groupings of operations (e.g., "User" or "Product") to help organize the API.
- externalDocs: Link to external documentation or resources.

3. Describing Operations and Endpoints Each API endpoint is described under the paths object. For example, a GET request to fetch a list of users might look like:

```yaml
paths:
```

```yaml
  /users:
    get:
      summary: Retrieve all users
      operationId: getUsers
      tags:
        - Users
      parameters:
        - name: page
          in: query
          description: Page number for pagination
          required: false
          schema:
            type: integer
      responses:
        '200':
          description: A list of users
          content:
            application/json:
              schema:
                type: array
                items:
                  $ref: '#/components/schemas/User'
```

This specifies:

- GET /users: A GET request to the /users endpoint.
- summary: A brief description of the operation.
- operationId: A unique name for the operation, useful for generating code or documentation.
- parameters: Query parameters or body content that the operation expects.
- responses: HTTP responses and their respective data.

4. Schemas and Data Types The components section defines reusable schemas that describe the structure of the request and response bodies. These schemas are often written in JSON Schema format. For example:

```yaml
components:
  schemas:
    User:
      type: object
      properties:
        id:
          type: integer
        name:
```

```
          type: string
        email:
          type: string
      required:
        - id
        - name
        - email
```

This defines a User object with an id, name, and email, marking all of them as required.

5. Parameters Parameters are used to describe the inputs that an API method accepts. They can be passed via:

- path: Part of the URL (e.g., /users/{userId}).
- query: URL query parameters (e.g., /users?page=2).
- header: HTTP headers.
- cookie: Cookies.
- Parameters are defined with type, description, and constraints.

6. Responses and Status Codes The responses section describes the possible responses for an operation, including HTTP status codes and the response content type.

```
yaml

responses:
  '200':
    description: Successful operation
    content:
      application/json:
        schema:
          $ref: '#/components/schemas/User'
  '404':
    description: User not found
'''

**7. Security**
OpenAPI supports various authentication mechanisms (e.g., API keys, OAuth2, JWT). These are
defined under securitySchemes in the components section.

Example using an API key for security:
```yaml
yaml

components:
  securitySchemes:
```
```

```
    api_key:
      type: apiKey
      in: header
      name: X-API-KEY

security:
  - api_key: []
```

8. Documentation Tools There are many tools that can generate user-friendly documentation from OpenAPI specifications:

- **Swagger UI:** A popular tool for displaying interactive API documentation.
- **Redoc:** Another open-source tool for generating attractive documentation.
- **Postman:** Allows you to import OpenAPI specs and interact with your API.

11. Code Generation Tools like **Swagger Codegen** and **OpenAPI Generator** can automatically generate client libraries, server stubs, and API documentation based on OpenAPI specifications.

OpenAPI Documentation Standards

Creating effective OpenAPI documentation is essential for ensuring that your API is easy to understand, use, and maintain. OpenAPI (formerly Swagger) is a widely used specification for describing RESTful APIs. Developers should follow certain standards to ensure their OpenAPI documentation is clear, consistent, and useful to other developers and consumers.

Here are the key standard practices for developers when creating OpenAPI documentation:

1. Adhere to the OpenAPI Specification (OAS) Version

- Version Consistency: Use the latest stable version of the OpenAPI Specification (currently OpenAPI 3.x). Older versions may lack critical features and may be incompatible with modern tools.
- Schema Versioning: Clearly specify the version of your OpenAPI document in the openapi field. This helps maintain compatibility with different tools and libraries.
- Example:

```yaml
openapi: 3.0.0
```

2. Provide Clear and Comprehensive Descriptions

- API Overview: Include a brief summary and description of what your API does in the info object. This helps consumers quickly understand its purpose.

```yaml
info:
  title: "My API"
  description: "This API allows users to manage and interact with resources."
  version: "1.0.0"
```

- Endpoint Descriptions: Provide clear descriptions of each endpoint and the intended use case. Ensure each HTTP method (GET, POST, etc.) has a meaningful description.
- Parameter Descriptions: For each query, path, header, and request body parameter, provide a detailed description of its purpose and usage.

```yaml
paths:
```

```yaml
  /users/{userId}:
    get:
      summary: "Retrieve user by ID"
      description: "Fetches user details by their unique ID."
      parameters:
        - name: "userId"
          in: "path"
          description: "ID of the user to fetch"
          required: true
          schema:
            type: "string"
```

3. Use Meaningful Naming Conventions

- Endpoints: Use consistent and RESTful naming conventions. For example, use plural nouns for resource names (/users, /products) and standard HTTP methods for actions (GET, POST, PUT, DELETE).
- Path Parameters: Ensure that path parameters have descriptive names that match the resource they represent.

```yaml
/users/{userId}/orders/{orderId}:
  get:
    summary: "Get a specific order for a user"
    parameters:
      - name: "userId"
        in: "path"
        description: "User identifier"
        required: true
      - name: "orderId"
        in: "path"
        description: "Order identifier"
        required: true
```

4. Utilize Request and Response Examples

- Example Requests: Provide examples of both request and response bodies to show how users should structure their data.

```yaml
requestBody:
  content:
```

```
    application/json:
      schema:
        type: object
        properties:
          name:
            type: string
      examples:
        example1:
          value:
            name: "John Doe"
```

- Example Responses: Include example responses for different HTTP status codes (e.g., 200, 404, 500). This helps users understand the kind of responses they can expect from the API.

```
yaml

responses:
  '200':
    description: "Successful response"
    content:
      application/json:
        schema:
          type: object
          properties:
            id:
              type: string
            name:
              type: string
        examples:
          success:
            value:
              id: "123"
              name: "John Doe"
```

5. Use Proper HTTP Status Codes

- Ensure that your API uses the correct HTTP status codes. For example:
  - 200 OK for successful requests.
  - 400 Bad Request for validation errors or missing parameters.
  - 404 Not Found for non-existing resources.
  - 500 Internal Server Error for server issues.

```
yaml
```

```
responses:
  '400':
    description: "Bad request"
  '404':
    description: "Not found"
  '500':
    description: "Internal server error"
```

6. Define and Document Data Models Clearly

- Schema Definitions: Use the components section to define reusable data models and schemas. This avoids redundancy and keeps the documentation DRY (Don't Repeat Yourself).

```
yaml

components:
  schemas:
    User:
      type: object
      properties:
        id:
          type: string
        name:
          type: string
    Order:
      type: object
      properties:
        orderId:
          type: string
        productName:
          type: string
```

- Reference these models in your endpoints.

```
yaml

paths:
  /users/{userId}:
    get:
      summary: "Get user by ID"
      responses:
        '200':
          description: "User found"
          content:
```

```
        application/json:
          schema:
            $ref: "#/components/schemas/User"
```

7. Use Security Schemes and Authentication

- If your API requires authentication, define the security schemes in the components section and reference them in the security field.

```
yaml

components:
  securitySchemes:
    apiKey:
      type: apiKey
      in: header
      name: X-API-Key

security:
  - apiKey: []
```

8. Use Descriptive Tags for Organizing Endpoints

- Group related endpoints under meaningful tags to improve navigation and organization in the documentation.

```
yaml

paths:
  /users:
    get:
      tags:
        - "User Operations"
      summary: "Get all users"
```

9. Make Use of OpenAPI Tools

- Leverage tools like Swagger UI, Redoc, and Postman to generate interactive documentation and explore API endpoints.
- Use validation tools to ensure your OpenAPI documents are well-formed and follow the specification.

10. Version Your API

- Clearly indicate API versioning in both the URL (e.g., /v1/users) and in the info object. This helps avoid confusion as your API evolves over time.

11. Provide Helpful Error Handling Document all possible error responses, including common error types, response formats, and how users can address them.

```yaml
responses:
  '400':
    description: "Bad request due to missing parameters"
    content:
      application/json:
        schema:
          $ref: "#/components/schemas/Error"
```

By following these standards, developers can create OpenAPI documentation that is clear, consistent, and helpful to others working with their API, leading to better maintainability and a more streamlined development process.

Contribution Standards

- Provide clear guidelines for anyone interested in contributing to the project. This can include:
- Forking the repo, cloning, and creating branches.
- Code style, testing requirements, or any conventions to follow.
- Pull request process.
- **Example:**
    - Fork the repository and create a new branch for your changes.
    - Ensure all tests pass before submitting a pull request.
    - Follow the code style guidelines (indentation, naming conventions, etc.).
    - Submit a pull request with a detailed description of the changes you made.

DotNetOpenAPI Project Information and Installation Instructions

This project is an ASP.NET Core application with integrated Swagger for API documentation.

Prerequisites

- .NET 8.0 SDK
- Node.js (if you have any frontend dependencies)
- Visual Studio or any other IDE of your choice

Installation

1. **Clone the repository:**

```
git clone https://github.com/yourusername/DotNetOpenAPI.git
```

2. **Restore dependencies:**

```
dotnet restore
```

3. **Build the project:**

```
dotnet build --configuration Release
```

4. **Run the project:**

```
dotnet run --configuration Release
```

The application will start and listen on `http://localhost:5108` by default.

Swagger Documentation

Swagger is used to generate API documentation. Once the application is running, you can access the Swagger UI at: http://localhost:5108/swagger/index.html

To view OpenAPI documentation you can either use browser or download the documentation under your project as .json file.

- View via browser: http://localhost:5108/swagger/v1/swagger.json

- Download document as swagger.json file: In Visual Studio Terminal, run the command: curl http://localhost:5108/swagger/v1/swagger.json

Running in GitHub Actions

This project includes a GitHub Actions workflow to build the project and generate the Swagger documentation. The workflow is defined in `.github/workflows/build-and-generate-swagger.yml`.

Example GitHub Actions Workflow

name: Build and Generate Swagger on: push: branches: - main pull_request: branches: - main jobs: build: runs-on: ubuntu-latest steps:

- name: Checkout code uses: actions/checkout@v2
- name: Set up .NET uses: actions/setup-dotnet@v1 with: dotnet-version: '8.0.x'
- name: Restore dependencies run: dotnet restore
- name: Build run: dotnet build --configuration Release
- name: Publish run: dotnet publish --configuration Release --output ./publish
- name: Start ASP.NET Core application env: ASPNETCORE_URLS: http://localhost:5001 run: | nohup dotnet ./publish/DotNetOpenAPI.dll & sleep 10

  # Check if the application is running

  until curl -s http://localhost:5001/swagger/v1/swagger.json; do echo "Waiting for the application to start..." sleep 5 done

- name: Download swagger.json run: curl http://localhost:5001/swagger/v1/swagger.json -o swagger.json
- name: Upload swagger.json uses: actions/upload-artifact@v4 with: name: swagger-json path: swagger.json

Contributing

Contributions are welcome! Please open an issue or submit a pull request for any changes.

License

This project is licensed under the MIT License. See the LICENSE file for details.

Documentation Standards For Code Comments

In .NET 8, the documentation standards for code comments, especially for classes and methods, are based on XML documentation comments. These comments serve as a standardized way to document code for better maintainability, usability, and integration with tools like Visual Studio or other IDEs.

- **Class-level Documentation** At the class level, document the purpose of the class and its key features.
    - **Standards:**
        - **Purpose:** Describe the primary role of the class and its responsibilities.
        - **Dependencies:** List important dependencies or relationships to other classes, components, or systems.
- **Method/Function-level Documentation** Each method should have documentation explaining what it does, its parameters, and its return value. If applicable, include exceptions that the method can throw.
    - **Standards:**
        - **Purpose:** What is the method doing? Why does it exist?
        - Parameters: A brief description of each input parameter, including the type and expected values or range (if applicable).
        - Return Value: Describe what the method returns, its type, and possible return values.
        - Exceptions: Mention any exceptions the method may raise and under what conditions.

Here is a guide on how to properly document classes and methods with XML documentation comments in .NET 8:

General XML Documentation Comments Structure

XML comments use the following syntax:

```
/// <summary>
/// Brief description of what the class, method, or property does.
/// </summary>
/// <remarks>
/// Additional detailed information about the class, method, or property.
/// </remarks>
/// <param name="paramName">Description of the parameter.</param>
/// <returns>Explanation of the return value.</returns>
/// <exception cref="ExceptionType">Condition that causes the exception.</exception>
```

- summary: A concise description of the element (class, method, property).
- remarks: Detailed information or additional notes about the element.

- param: Describes the parameters for methods or constructors. You should document each parameter individually by name.
- returns: Describes what the method returns.
- exception: Explains the exceptions the method might throw.

Class Documentation Example

A class is documented by explaining its purpose, any important details, and how it interacts with other components.

```
/// <summary>
/// Represents a customer in the system.
/// </summary>
/// <remarks>
/// This class contains properties and methods for managing customer data,
/// including their personal details and order history.
/// </remarks>
public class Customer
{
    /// <summary>
    /// Gets or sets the customer's unique identifier.
    /// </summary>
    public int CustomerId { get; set; }

    /// <summary>
    /// Gets or sets the name of the customer.
    /// </summary>
    public string Name { get; set; }

    /// <summary>
    /// Calculates the total amount spent by the customer.
    /// </summary>
    /// <returns>
    /// The total amount spent by the customer.
    /// </returns>
    public decimal GetTotalSpent()
    {
        // Logic to calculate total spent
        return 100.50m;
    }
}
```

Method Documentation Example

Methods should be documented to explain their purpose, parameters, return values, and any exceptions they may throw.

```
/// <summary>
/// Adds a new order for the customer.
/// </summary>
/// <param name="order">The order to add.</param>
/// <exception cref="ArgumentNullException">Thrown when the order is null.</exception>
/// <exception cref="InvalidOperationException">Thrown when the customer is not eligible to
place an order.</exception>
public void AddOrder(Order order)
{
    if (order == null)
    {
        throw new ArgumentNullException(nameof(order), "Order cannot be null.");
    }

    // Logic for adding an order
}
```

Property Documentation Example

Document properties with a short description of what they represent.

```
/// <summary>
/// Gets or sets the customer's email address.
/// </summary>
/// <value>
/// The email address of the customer.
/// </value>
public string Email { get; set; }
```

Constructor Documentation Example

Constructors are documented similarly to methods but with an emphasis on explaining what the constructor is initializing.

```
/// <summary>
/// Initializes a new instance of the <see cref="Customer"/> class.
/// </summary>
/// <param name="customerId">The unique identifier for the customer.</param>
/// <param name="name">The name of the customer.</param>
public Customer(int customerId, string name)
{
```

```
    CustomerId = customerId;
    Name = name;
}
```