Manual Heading

HM Vibha Towers, 5th Floor, No. 66/5-25, Lasker Hosur Road, Adugodi, above Star Hyper,
Bengaluru, Karnataka 560030

Disclaimer

This document contains confidential and proprietary data/information/material of Perfios Software Solutions Private Limited. No data/information/material contained in this document shall be copied, reproduced, republished, distributed, transmitted, downloaded, transferred, broadcasted in any form and, /or exploited commercially or otherwise in any manner whatsoever without obtaining the express prior written consent of Perfios Software Solutions Private Limited.

Revision History

Version	Issued Date	Description	Status	Remarks	Author
1.0	07-Sep-2025	Home Loan – Product Information			Deepu

Perfios Confidential

Contents

Perfios Confidential

1 Introduction

Perfios Software Technologies has automated the onboarding journey for **Application Name** applicants to the Union Bank of India (UBI). The following document serves as a Product information manual.

--- title: Compoents sidebar_position: 3 ---

1.1 Components of openAPI

The major componets are:

- Parameters
- requestBody
- responsesBody

1.2 Parameters

It define the inputs an API operation can accept, other than the body payload. They specify how clients can send values to an endpoint.

In OpenAPI, you can declare parameters for each operation (get, post, etc.) or reuse them via components/parameters.

1.2.1 Types of parameters

Parameter Location	**\`in:\` value**	**Example**
Query parameter	\`query\`	\`/users?limit=10\`
Path parameter	\`path\`	\`/users/{id}\`
Header parameter	\`header\`	\`X-Request-ID: abc123\`
Cookie parameter	\`cookie\`	Sent as HTTP cookie

1.2.2 Example

```yaml parameters: # Path parameter

- name: id

in: path description: Unique identifier for the user required: true schema: type: integer

# Query parameter

- name: expand

in: query description: Additional related data to include required: false schema: type: string enum: [posts, settings]
```

# Header parameter

- name: X-Request-ID

in: header description: Optional request ID for tracing required: false schema: type: string

# Cookie parameter

- name: session_id

in: cookie description: Session identifier required: false schema: type: string ```

1.2.3 Parameter Attributes

Field	Description
`name`	Name of the parameter
`in`	Location: `query`, `path`, `header`, or `cookie`
`required`	`true` or `false`
`description`	Optional explanation
`schema`	Data type, constraints, enums
`example`	Example value
`deprecated`	Mark a parameter as deprecated

1.2.4 Best practices of parameters

```yaml ✅nly path parameters must always be required: true.

✅ Keep header and cookie params optional unless absolutely necessary.

✅ Describe how each parameter affects the response in description. ```

## 1.3 requestBody

In OpenAPI, the requestBody describes the content of the HTTP request body that clients send to the API.

It can specify:

```yaml Content type(s) supported (application/json, multipart/form-data, etc.)

Schema of the body

Examples of the body ```

It’s not a parameter, because it isn’t sent in the query/path/header/cookie — it’s the actual body of the HTTP request.

### 1.3.1 Examples

```yaml post: summary: Create a new user description: Creates a new user record in the system. requestBody: description: User data to create required: true content: application/json: schema: type: object properties: name: type: string description: Full name of the user email: type: string format: email description: Email address of the user age: type: integer description: Age of the user required:

- name
- email

example: name: John Doe email: john.doe@example.com age: 30 ```

### 1.3.2 Attributes

| **Field** | **Description** |
|---|---|
| `description` | Optional text explaining what the request body is |
| `required` | `true` or `false` |
| `content` | Defines media types and schema(s) |
| `example` | Single example of a valid body |
| `examples` | Multiple named examples (optional) |

### 1.3.3 Typical case of requestBody

| **HTTP Method** | **Use of Body** |
|---|---|
| `POST` | Create a new resource |
| `PUT` | Replace a resource |
| `PATCH` | Partially update a resource |
| `DELETE` | Rare, but can send a body in some cases |

### 1.3.4 Best practices of requestBody
```

```yaml ✅ Always set required: true if the endpoint fails without a body.

✅ Use $ref to reuse schemas and avoid duplication.

✅ Include examples in your spec to help consumers understand the payload format.

✅ Use proper media types (JSON is standard for REST APIs). ```

## 1.4 responseBody

The responses object defines all possible HTTP responses for an operation, keyed by status code.

```yaml Status code (200, 400, 404, 500, etc.) Data format (application/json, application/xml, etc.) Schema of the response data Example(s) of response data ```

### 1.4.1 Example's

```yaml responses: '200': description: User found content: application/json: schema: type: object properties: id: type: integer example: 1258 name: type: string example: David J email: type: string example: davidjxyz@gmail.com example: id: 123 name: John Doe email: john.doe@example.com '404': description: User not found content: application/json: schema: type: object properties: error: type: string example: error: User with id 1258 not found ```

### 1.4.2 Key Fields of responseBody

| Field | Description |
|---|---|
| `description` | Mandatory short text about what this response means |
| `content` | Media type(s) and body schema |
| `headers` | Optional — response headers |

### 1.4.3 Best practices of responsesBody

```yaml ✅ Always document common status codes: 200, 201, 400, 401, 404, 500.

✅ Include an example for every response type.

☑ Use clear, human-readable description.

☑ Define error formats (standard error response schema). ```

--- title: Welcome sidebar_position: 1 ---

<img src="/img/cove-page-image.jpg" alt="cover_page" style={{ maxWidth: "500px", margin: "0 auto", boxShadow: '0 4px 12px rgba(0, 0, 0, 0.3)', borderRadius: '2rem', alignItem: 'center'}} />

--- id: introduction title: Introduction sidebar_position: 2 ---

## 1.5 What is OpenAPI?

OpenAPI is a **standard specification** for defining RESTful APIs — how they work, what endpoints they have, what inputs/outputs they expect — all written in a **machine-readable format** (YAML or JSON).

### 1.5.1 in other words

#### 1.5.1.1 If you're building or consuming an API, the OpenAPI document serves as

- It's **contract** between the API provider and the consumer.
- A **blueprint** of the API.
- A source of truth for **developers, testers, and clients**.
- A way to generate documentation, SDKs, mock servers, and more.

## 1.6 Why openAPI?

- **Clear communication:** All teams (frontend, backend, QA, external clients) can understand the API without ambiguity.

- **Automation:** Tools like Swagger UI, Redoc, and Postman can read the file and give you documentation, code, tests.

- **Validation:** You can test if your API responses actually match the spec.

- **Mocking:** Generate fake APIs before backend is ready.

## 1.7 What is an OpenAPI Document?

An OpenAPI document is the YAML (or JSON) file where you describe your API, such as:

- API version & metadata
- Server URLs
- Authentication methods
- Paths (endpoints), HTTP methods
- Request parameters, request bodies
- Response schemas, HTTP status codes
- Example requests & responses
- Tags & grouping

### 1.7.1 API version & metadata

- Every API needs a **name, version number,** and **description** to identify it.
- Optional metadata like contact info, license, terms of service helps consumers.

Ymal format example:

info: title: Sample API version: 1.0.0 description: This is a sample API.

### 1.7.2 Server URLs

- Defines the base URL of your API.
- Consumers need to know where to send requests (prod, dev, staging servers can all be listed).

ymal format example:

servers:

- -url: https://api.example.com/v1

description: Production Server

### 1.7.3 Authentication methods

- Defines how clients authenticate.
- Could be:
- API keys
- OAuth 2.0
- JWT tokens
- HTTP Basic Auth
- Without this, clients wouldn’t know how to gain access.

yaml format basic security example:

securitySchemes:

```yaml BasicAuth: type: http scheme: basic ```

### 1.7.4 Summary

| **Concept** | **What it means** | **Why it matters** |
|---|---|---|
| API version & metadata | Name, version, description of the API | Identifies the API |
| Server URLs | Base URLs (prod/dev) | Tells clients where to call |
| Authentication methods | How to authenticate (API key, OAuth) | Secures the API |
| Paths & HTTP methods | Endpoints and actions | The core of the API |
| Request parameters & bodies | What data the client can send | Defines inputs |
| Response schemas & codes | What the server returns | Defines outputs |
| Example requests/responses | Example payloads | Improves clarity |
| Tags & grouping | Logical categorization | Improves usability |

--- title: Rest vs Open api sidebar_position: 5 ---

**OpenAPI:** a document that describes your REST API — what endpoints exist, what they do, what inputs & outputs they expect — in a standard, machine-readable way.

**REST API:** the actual service — a set of HTTP endpoints that follow REST principles (stateless, resource-oriented, uses verbs like GET/POST/PUT/DELETE).

## 1.8 Difference Between OpenAPI and REST API

| Parameters | OpenAPI | REST API |
|---|---|---|
| ◆ **What is it?** | A **specification/standard for documenting REST APIs.** | A style/architecture for designing APIs. |
| ◆ **Category** | Documentation & contract standard. | API design style. |
| ◆ **Purpose** | To describe a REST API in a machine-readable, standardized way. | To implement a stateless, resource-based API. |
| ◆ **Format** | YAML or JSON document (the OpenAPI document). | API implementation itself (usually HTTP endpoints). |
| ◆ **Example** | \`openapi.yaml\` describing \`/users\`, \`/orders\`, etc. | Server at \`api.example.com\` exposing \`/users\`, \`/orders\`. |
| ◆ **Standard?** | Yes — maintained by OpenAPI Initiative. | Not a standard, but an architectural principle. |
| ◆ **Interactive?** | Enables tools like Swagger UI, Redoc to render interactive docs. | REST API itself is just HTTP responses. |

| | | |
|---|---|---|
| ◆ **Relationship?** | Documents and defines the contract of a REST API. | The actual API you build and deploy. |

## 1.9 Example

You build this REST API:

```yaml GET /users POST /users GET /users/{id} ```

You can define it in OpenAPI like this:

```yaml paths: /users: get: summary: Get users post: summary: Create user /users/{id}: get: summary: Get user by ID ```

--- id: intro title: Introduction sidebar_position: 1 ---

# 2 Welcome

<!-- ### Key activities -->

This is normal text.

- Item 1
- Item 2

## 2.1 Next Section

Some more content.

<!-- hidden debug comment -->

Final line.

The text need to be deleted **here**.

--- title: Workflow sidebar_position: 4 ---

## 2.2 openAPI workflow stages

The below diagram help you to understand the different stages of work flow between starting to deployment of openAPI doc.

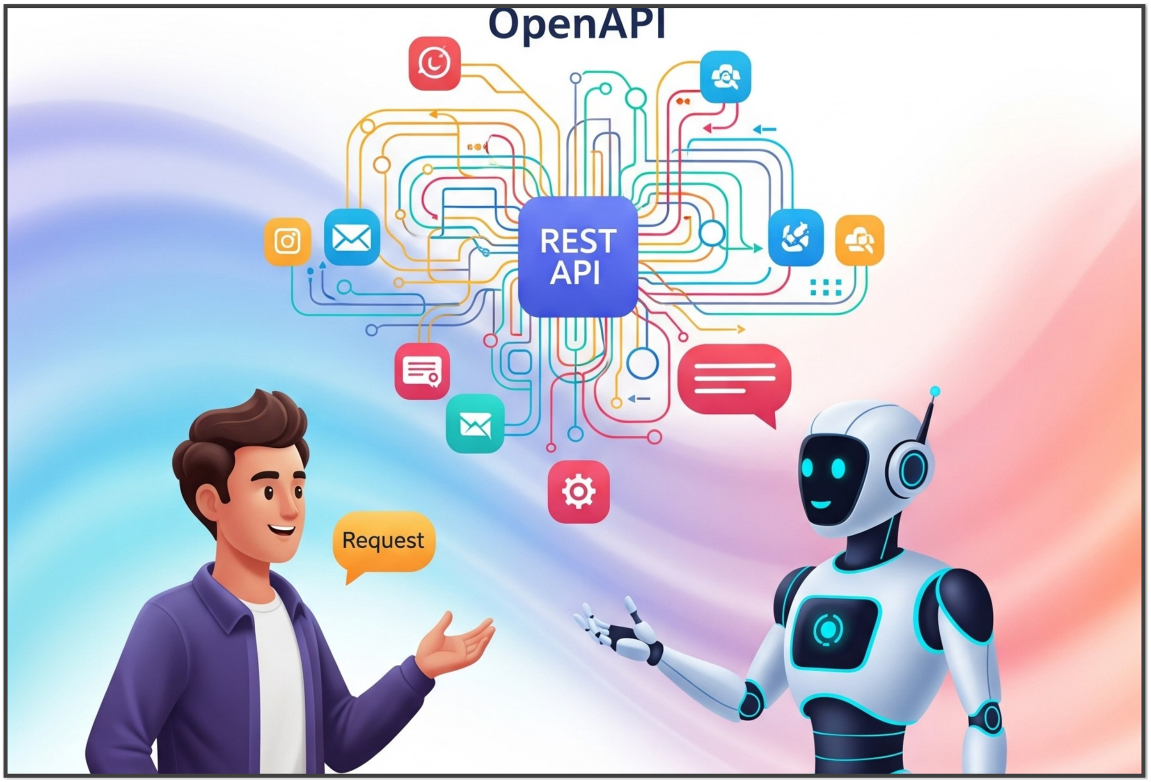
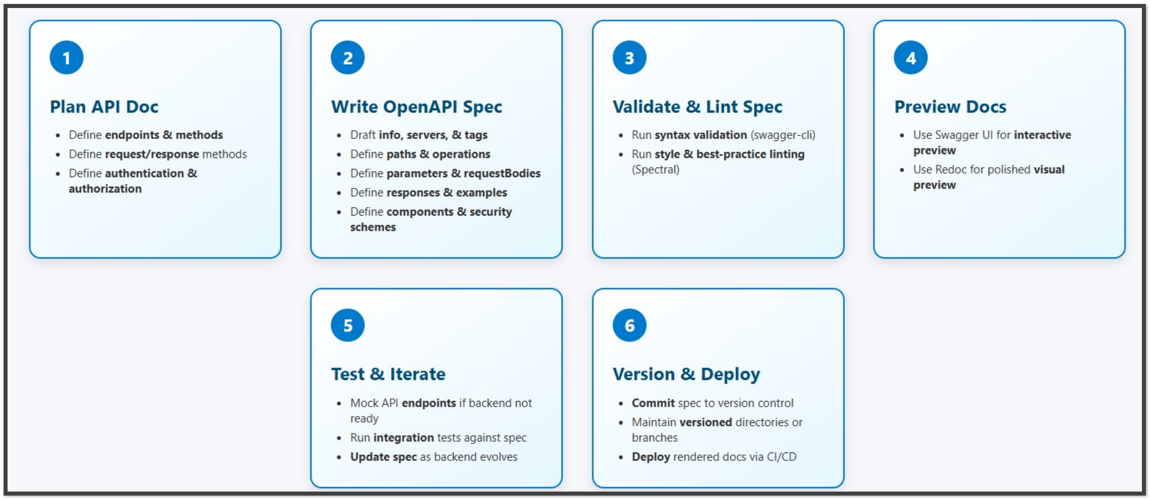


Figure 1

Image: cover-image



1

**Plan API Doc**

- Define **endpoints & methods**
- Define **request/response** methods
- Define **authentication & authorization**

2

**Write OpenAPI Spec**

- Draft **info, servers, & tags**
- Define **paths & operations**
- Define **parameters & requestBodies**
- Define **responses & examples**
- Define **components & security schemes**

3

**Validate & Lint Spec**

- Run **syntax validation** (swagger-cli)
- Run **style & best-practice linting** (Spectral)

4

**Preview Docs**

- Use Swagger UI for **interactive preview**
- Use Redoc for polished **visual preview**

5

**Test & Iterate**

- Mock API **endpoints** if backend not ready
- Run **integration** tests against spec
- **Update spec** as backend evolves

6

**Version & Deploy**

- **Commit** spec to version control
- Maintain **versioned** directories or branches
- **Deploy** rendered docs via CI/CD

Figure 2

Image: openapi-different-stages

Figure 3

Image: human

## 2.3 Plan API doc & Content

Define what the API is supposed to do, and agree on a contract between teams.

<!-- ### Key activities -->

1. Gather business requirements: What functionality should this API expose? **Who will use it**?

2. Identify resources & endpoints: e.g., **/users, /orders**.

3. Define **HTTP methods** for each endpoint: GET, POST, PUT, DELETE.

**Note:**

- Determine **data models & schemas**: What does a **User look like**? What fields are required?

- Specify **authentication & authorization schemes**: API keys, OAuth 2.0, JWT.

- Establish **error handling conventions**: e.g., HTTP 4xx/5xx codes and error body formats.

### 2.3.1 Best Practices

**Note:**

✅ Involve stakeholders early: product owners, developers, QA, external consumers.

✅ Document decisions clearly in a design document before implementing the API.

✅ Use a whiteboard or Figma for initial designs.

## 2.4 Write openApi Spec

Write the API contract in OpenAPI format (YAML or JSON).

<!-- ### Key Activities -->

- Add **metadata**: title, version, description, contact, license.

- Define **servers**: base URLs for production, staging, etc.

- Write **paths & operations**: with parameters, request bodies, responses, tags.

- Create reusable **components**: schemas, responses, parameters.

- Define **security schemes**: and apply them globally or per-operation.

- Add **examples**: for requests and responses to illustrate expected payloads.

### 2.4.1 Best Practice

```yaml ✅ Use tools like Swagger Editor, VS Code plugins for authoring.

✅ Keep the spec in source control alongside code.

✅ Use $ref for reuse and maintainability. ```

## 2.5 Validate & Lint

Ensure your OpenAPI document is correct and follows best practices.

- Run **syntax validation**: check that YAML/JSON is valid and conforms to OpenAPI spec.

- Run a **linter**: check for style conventions, consistency, recommended practices.

Example

```yaml tools: Swagger CLI, Spectral, Redocly CLI. ```

### 2.5.1 Best Practices

```yaml ✅ Integrate validation into CI/CD pipelines to catch errors early.

✅ Define and enforce lint rules to maintain quality and consistency. ```
```

2.6 Preview Documentation

Render your OpenAPI spec into human-friendly documentation.

- Generate docs using:

Swagger UI: interactive “Try It Out” experience.

Redoc: clean, three-panel layout for polished documentation.

ReDocly: This will allow the user to test but its subscription model.

- Share preview links with developers, QA, and stakeholders for feedback.

2.6.1 Best Practice

```yaml ✅ Host your docs on an internal URL or cloud platform.

✅ Customize branding, colors, and layout for external-facing APIs.

✅ Include sample curl commands and SDK examples if possible. ```

## 2.7 Test & Iterate

Use the spec as the source of truth and refine it as the implementation progresses.

- Use **mock servers**: simulate the API based on the spec before backend is implemented.
- Write automated **integration tests** to verify the API matches the spec.
- Run **contract tests** as part of CI.
- Update spec whenever backend implementation changes.

### 2.7.1 Best Practices
```

```yaml ✅ Treat the spec as a living document — update it with every change.

✅ Encourage developers and QA to report mismatches between spec and implementation. ```

## 2.8 Version & Deploy

Version your spec, host the docs, and automate deployment.

- Commit spec to Git with semantic versioning: e.g., v1.0.0, v1.1.0.
- Maintain separate folders/branches for each major version.
- Deploy rendered documentation to:

GitHub / GitLab Pages or Redocly Portal or your own servers

- Tag releases in **GitHub/GitLab** to track changes over time.

## 2.9 Summary

| **Stage** | **Goal** |
|---|---|
| Plan API & Contract | Define the API & align on expectations |
| Write OpenAPI Spec | Document the API in OpenAPI format |
| Validate & Lint | Ensure spec is correct & high quality |
| Preview Docs | Render docs for review & publishing |
| Test & Iterate | Keep spec & implementation in sync |
| Version & Deploy | Manage versions & publish for consumers |
```