API Design & Implementation Best Practices

Sreedhar Sonnati
Integration Architect

API deployment requires concise planning and strategy. Through best practices, you can position your business for success in digital transformation.

XTIVIA

WHITE PAPER

Digital transformation is happening everywhere with the involvement of mobile and cloud technologies. Application Programming Interfaces (APIs), once seen as developers' tools, are now being exposed to the market. However, digital transformation is not an easy task. It involves the ability to bring in multiple technologies to create distinctive and disruptive services to the market. To make this happen, we must be able to retrieve data/information from disparate sources and provide them to multiple consumers in various formats.

While navigating through Digital Transformation, most organizations were not effectively using the best features of Rest API standards, Best Practices, and the flexibilities of it. They ended up with the same issues they had in their old environment and wasted a lot of time and resources to resolve those issues. Many organizations are not following the best practices and key concepts.

Section One:

Understanding Issues in API Design and Implementation

A majority of APIs are built with Monolithic Architecture, which has all business functionalities in a single API. This has drawbacks and is pretty high risk; it's hard to change and maintain, and it takes time to deliver to the market. If any changes to the source and target systems are required, we need to rewrite the entire functionality, as it's rigid and sticks to serve only one function that cannot be reused. It's not flexible, reusable, nor scalable, cross-functional, or extensible, and it can't be used for multiple consumers at the same time. This is why we need to build more APIs to serve multiple consumers to get similar functionality since each source has a different set of input and output parameters. Most of the implementation will be redundant in this case.

APIs serve enterprises by communicating with different systems because it's hard to troubleshoot when there are connectivity issues, system failures, or unavailability of systems. If any of the systems are altered or retired, we need to rewrite the API and put effort to test the entire functionality again and again, which is time-consuming, costly, and impacts the existing users as well.

Another factor we've identified is multiple domains related to functionalities of the organization that have been incorporated within the same API. Another drawback is if any domain-related changes happen, we need to test all other domains to guarantee functionality.

Examples of API Implementation Issues

We've seen the below issues with the Coarse-Grained API implementation at one of our customer sites. They built an API that deals with customer-related CRUD operations, tokenization of credit card transactions, and customer loyalty operations dealing with the JSON data format.

After implementing this API, we received the following requirements:

1. Another application is looking to perform customer-related operations with XML as a data format.
2. Mobile applications and web applications are looking for a different set of request and response fields.
3. Some applications need enhanced output fields by communicating dependent services.
4. In the future, we need to update the customer-related CRUD operations to Salesforce along with the existing system.
5. Other Internal APIs need tokenization functionality in their individual APIs and the same for loyalty points as well.
6. In the long-term perspective, they want to change the back-end storage from Oracle to Salesforce for customer and loyalty data.

For all of the above scenarios, we can't leverage the existing API implementation. Instead, for each situation, we need to build a brand-new API that's serving similar business functionality.

Examples of API Security Issues

Most implementations were just done for Security Socket Layer (SSL) Offloading and did not apply any other security constraints at the external and internal gateway level. This means that their system is not robust and is susceptible to crashes from external attackers. If proper security constraints were not in place, we may get DoS (Denial-of-Service) Attacks, Anti-farming issues, and the API then becomes unresponsive for users. As access permissions are not defined with user roles, everyone can access all API functions, which causes unnecessary data traffic, makes the API unresponsive with overload, and negatively impacts the business. Since API validation is not in place, we can face issues like code injections, malicious entity declarations, and parser attacks as well.

Examples of Insufficient API Policies

Most of the APIs were not having Authentication, Authorization, threat protection, or Quality of Service-related policies applied to their APIs. If authentication and authorization policies are not defined, intruders can access the API and steal the information. Since there is no control over the number of requests made to the API, the API will be overloaded with calls and API performance will be degraded. In some cases, APIs are unresponsive and "out of memory" issues will occur. Without applied threat protection policies, the API will not be safe from malware, risky file types, or website/malicious network traffic.

Insufficient API Versioning

Most of the Organizations are not maintaining the API versioning. Instead, they are building the API's on-demand as opposed to leveraging existing APIs that serve the same functionalities. If we don't have versioning, you will see duplicate and redundant APIs in the system. If any changes need to be applied, we will need to rekey the multiple versions, which is a time-consuming process and requires manual maintenance, which is not a good practice.

Let's explore a scenario: we built the customer API version 1.0, which is saving customer data to Oracle in the backend. As requested by the customer, we decided to change the backend from Oracle to Salesforce. The customer expects both old and new versions running for some time. If we did not maintain the versioning, we cannot meet the customer's needs. In the meantime, if any issues happened in Version 1.0 of the API, we don't have control, nor can we fix the issue immediately.

Inconsistent API Naming Conventions

We have not seen unique naming conventions for each resource, endpoints, query parameters, URI Parameters, Headers, or body elements in the content. Some of the inconsistent examples are below.

- http://api.example.com/inventory_management/managed_entities/{id}/install_script_location - This example is not readable.
- http://api.example.org/My-Folder/my-doc - This example doesn't have the same case.
- http://api.example.com/createcustomer - This example forces creation in the resource name.
- http://api.example.com/searchcustomer/ - This example has "/" at the end.
- http://api.example.com/$payment - This example has special characters
- http://api.example.com/:Id/ - This example is not self-explanatory and not readable.

Section Two:

API Standards and Implementation Best Practices

XTIVIA has years of experience in API standards and implementations. We resolved all these issues by following the best practices listed below. As an implementation partner to various organizations, XTIVIA starts by analyzing the anomalies in the current implementation and applying best practices to facilitate business benefits and grow them exponentially.

API-Led Connectivity Best Practices

We recommend API-Led connectivity to all clients. In our approach, the APIs are based on three distinct layers: System, Process, and Experience. We recommend the following implementation at each layer.

1. System Layer

This is a foundational layer of the three-layered architecture. Here, APIs can be defined for various domains of an organization. For example, Customer operations, Proprietary databases, and all other backend systems. System APIs provide the mechanism to access systems of records and expose data in a canonical format. It defines the contract to describe domain interactions — for example, the Customer domain contains resources with Methods GET, POST, PUT, and DELETE.

System APIs expose an organization's information. They should not be exposed for public use.

- **Advantage**: We can modify the System API logic without affecting the other APIs in the flow. For example, if a System API is using Oracle and, in the future, Oracle needs to be replaced with MySQL, this replacement can be done easily by modifying only the System API without touching anything in the other layers (i.e., Process and Experience).

2. Process Layer

The Process layer APIs orchestrate and choreograph data by calling various System APIs. The orchestration can be aggregating data, splitting data, or routing data. The purpose of Process APIs is to encapsulate the business process independent of system APIs where the data originated.

System APIs expose an organization's information. They should not be exposed for public use.

- **Advantage**: Common business logic can be shared/reused as required — for example, if your organization has a billing process API implemented, it can be reused whenever necessary.

3. Experience Layer

The business process data is consumed with different formats across a range of clients/channels. For example, our billing API exposes data in CSV format, but the client application only accepts XML format or vice versa. The transformation logic would be implemented in the Experience Layer and exposed Experience APIs.

Experience APIs are where data can be reconfigured to meet the needs of multiple consumers. In this layer, we can remove unnecessary methods and expose only essential procedures in Experience APIs.

The Experience APIs are exposed publicly for consumption. Policies can be applied to the APIs, and they can also be monetized to earn potential revenue for the organization.

- **Advantage**: Experience APIs transform data to align with the client's diverse data formats. They can be used to transform data rapidly, thus, decreasing the time to market.

4. Benefits of API-Led Connectivity

API-led connectivity helps organizations improve API performance and application scalability to actualize the following benefits:

- **Transition to Cloud**: The emergence of the cloud has led many organizations to opt for cloud infrastructure but the transition is not always easy with legacy systems. API-led connectivity will not only accelerate adoption but streamline innovation as well.
- **Global Presence**: As business demands expansion, API-led connectivity data and infrastructure can easily unlock necessary accommodations and systems.
- **Scalable**: API-led connectivity will ensure API reusability and enablement of microservices, resulting in quicker application development and integration.
- **Real-Time**: A crucial factor in business growth is responding to customers faster, and API-led connectivity can provide that effortlessly, by delivering real-time customer information to your teams.
- **Bi-modal IT with Agility**: API-led connectivity ensures stability and control over the core system of records, allowing for rapid innovation and iteration of applications to access those systems.
- **Unified Connectivity**: Complete stack of blocks on one platform connectivity, services, and APIs.
- **High Productivity**: Ability to quickly experience new initiatives without any system-level discrepancies.
- **A Platform for the Business**: Leveraging cross-API connectivity offers an environment for the business to become self-sufficient.
- **Hybrid**: Easy transition to cloud and ensures code reusability.
- **Better Estimation**: Ensures the right estimation against any changes in code.

API Security Protocol Best Practices

In past projects, we've applied the following types of security protocols for different clients to meet their organization's security policies and safeguard their APIs.

- Always use HTTPS
- Use Password Hash
- Don't expose information in URLs
- Add timestamp in Request
- Use basic authentication, and if possible, OAuth.
- Employ Input Parameters for validation
- External users should use CA-signed Certificates
- Internal users should use Self-signed Certificates

API Policy Best Practices

The following API policies have served a majority of client needs while providing better performance and stability of APIs. We can categorize the policies below.

1. Security

- **Basic Authentication Policy (Simple)**: Allows access based on a basic authorization mechanism with a single user-password.
- **Basic Authentication Policy (LDAP)**: Allows access based on the basic authorization mechanism, with user-password defined on Lightweight Directory Access Protocol (LDAP).
- **IP Blacklist Policy**: Blocks a single IP address or a range of IP addresses from accessing an API endpoint.
- **IP Whitelist Policy**: Allows a list or range of specified IP addresses to request access.
- **JSON Threat Protection Policy**: Protects against malicious JSON in API requests.
- **XML Threat Protection Policy**: Protects against malicious XML in API requests.
- **JWT Validation**: Validates JWT Token.

2. Compliance

- **Client ID-Enforcement Policy**: Allows access only to authorized client applications.
- **CORS Policy**: Enables access to resources residing in external domains.

3. Quality of Service

- **Rate Limiting Policy**: The Rate Limiting policy limits the number of requests an API accepts within a window of time. The API rejects requests that exceed the limit. You can configure multiple limits with window sizes ranging from milliseconds to years.
- **Rate Limiting Policy (SLA-Based Policy)**: SLA-based rate-limiting restricts the number of requests by application to your API based on the configuration of an SLA tier.
- **Spike Control**: This policy ensures that within any given period of time, no more than the maximum configured requests are processed.

4. Troubleshooting

- **Message Logging**: It logs custom messages using information from incoming requests, responses from the backend, or information from other policies applied to the same API endpoint.

API Versioning Best Practices

API versioning is vital in the modern enterprise, as we are going to build numerous APIs and perform rapid developments to serve different consumers and platforms. XTIVIA recommends the versioning standards below (each organization chooses different methods among these).

- Versioning through URI Path
- Versioning through query parameters
- Versioning through custom headers
- Versioning through content negotiation

API Naming Convention Best Practices

XTIVIA recommends the following API Naming Conventions to be consistent and unique throughout the organization.

- Use nouns to represent resources.
- Use forward-slash (/) to indicate hierarchical relationships.
- Do not use trailing forward-slash (/) in URIs.
- Use a hyphen (-) to improve the readability of URIs.
- Do not use underscores (_).
- Use lowercase letters in URI.
- Never use CRUD function names in URIs.
- Use query component to filter URI Collection
- **Examples**:
  - http://api.example.com/inventory-management/managed-entities/{id}/install-script-location - (Readable)
  - http://api.example.org/my-folder/my-doc - (It has the same case)
  - http://api.example.com/customer - (Use HTTP Post method)
  - http://api.example.com/searchcustomer - (Removed the “/” at the end).
  - http://api.example.com/payment - (Removed the Special Characters).
  - http://api.example.com/Id - (Readable)

Conclusion

To help our clients meet their goals and build safeguarded APIs, XTIVIA recommends these best practices to promote strong security, fine-tuned performance, and API stability. By following these practices, our clients are able to better serve customers without hassle and grow their business with little-to-no issues.

If you have questions on how you can build better APIs or need help with API implementation, please reach out to us at http://www.xtivia.com/contact or info@xtivia.com.

About the Author

Sreedhar Sonnati | Integration Architect, EIM

Sreedhar Sonnati is a dynamic hands-on Integration Architect on the Enterprise Information Management (EIM) team at XTIVIA, Inc. As a collaborative business partner with global IT experience leading complex initiatives, Sreedhar drives transformational change, process automation, and technology innovation. Through strong partnerships with stakeholders, he helps sponsor initiatives that align with business goals resulting in the strategic use of information technology for competitive advantage. For over a decade, he's compiled a blend of IT and business experience through solution architecture, implementation, and API-led connectivity in on-prem, hybrid, and cloud-native environments.

If you can imagine a business outcome, XTIVIA can create it through technology.

XTIVIA does what it takes to ensure customer success through adaptive technology solutions. Our earned reputation is for delivering the right IT solutions and support that meet our customers' specific requirements, regardless of project complexity. Our passion, combined with a dedicated leadership team and unparalleled technical staff, creates customer relationships that stand the test of time.

xtivia.com/blog

linkedin.com/company/XTIVIA

twitter.com/XTIVIA

youtube.com/c/XTIVIA

XTIVIA

www.xtivia.com | (888) 685-3101 ext.2