Guidance for Programs to Create API Strategy

Contents

1. Purpose 2
2. Intended Audience ..... **Error! Bookmark not defined.**
3. Scope 3
4. Instructions 3
   - T.1 Program Application and Data API Vision 3
   - T.2 Key Data and Services 4
   - T.3 Key Stakeholders 4
   - T.4 Driving Architectures 4
   - T.5 API Governance 4
   - T.6 API Technical Details/Key Elements 6
   - T.7 Exceptions 10
5. Further Resources 10

{Program} API Strategy Template 11

1. Introduction

Application Programming Interfaces or APIs are used to provide simplified, reusable integration patterns between a user and a system or from system to system. APIs along with their supporting infrastructure (e.g., API Catalog, API Governance) allow systems to rapidly discover and access partner/enterprise data or capabilities.

The intended audience of this document is Program Offices and their technical leads. This is to provide a vehicle for the PMO to set their vision and strategy for implementing APIs and overall interoperability for the applications they are going to build and sharing of data. This API Strategy will also communicate to the stakeholder community their approach and allow any course corrections need to align programs with Service, Enterprise and Joint development efforts.

2. Purpose

The current state of the DoD must transform from a closed architectural approach which ignores the potential of collaboration and integration of systems. We need to move from a culture with a "One and Done" mentality, and lack of forethought using special, sometimes "hard-wired" network segments, VPNs, point-to-point interfaces and brute force transforms to a culture that anticipates software integration through standards-based APIs.

The DoD acquisition process has recently moved to agile software pathways needed to speed application development but still struggles to feed these data hungry efforts. This makes a solid API strategy starting point to interoperability and transform Net Readiness KPP from net-centric to data-centric which reflects modern software integration equities, shifting from a static to a dynamic and evolutionary model. The current interface control document and requirements approaches cause thousands of applications to recreate what industry relies on APIs to solve, causing the DoD to spend unneeded resources and take in some cases years to update data types and formats.

API first is an approach to development in which your APIs are considered first-class citizens. This means that everything you're developing is developed with the end goal of API consumption in mind. In API first development, APIs are no longer an afterthought — they're a differentiator.

In code-first development, you focus on building the service and its resources. Your developers are making the decisions and creating the API. With code-first development, there's always a risk of not delivering what the consumer needs. API first, synonymous with REST, is just one of a variety of approaches.

With API first development, your focus shifts to consumer needs via a feedback loop, then building out the service. This helps you make sure you're building the right API the first time — every time.

API first is still relevant today. And many businesses are still leveraging API first strategies. But it's not as innovative of an approach as it used to be. We've heard from several businesses embarking on digital transformations who are considering a different paradigm — data first.

A data first approach allows you to ensure that data gets to where it needs to go. It goes beyond point-to-point APIs and ensures that data is available to those who need to access it at any given time.

Data is the most important asset a business owns. By taking a data first approach, you build your strategy around a variety of API patterns that include REST, SOAP (yes, it is still very prevalent), GraphQL, and RPC.

The need to rapidly feed our data hungry applications is only the start of what the DoD needs to solve to stay competitive against our near-peer adversaries. We are falling behind other global AI superpowers in AI/ML due to the lack of data extensibility across our environments. We need to mirror industry leads such as Google, Microsoft, Amazon and IBM that rely on large-scale AI/ML models that are accelerated via APIs. In addition to the loss of countless man-hours and taxpayer's dollars, an API first strategy would slash years from our competitors' advantage. Overall, victory in future battlefields isn't going to the combatant with the largest inventory of ordnance, but rather to the side who can put real time data into the warfighter's hands.

3. Scope

All Application programs on the software pathway. All other Software Acquisition programs (as well as any program developing and deploying software) are strongly encouraged to produce an API Strategy to drive programs' interoperability approach.

4. Instructions

The intent of the API Strategy is to encourage the program to think about its approach to interoperability prior to starting development, so why the template has may topic areas, the intent is this document would start at establishing vision and initial plans to facilitate understanding by stakeholder. This strategy should be updated as the program evolves.

T.1 Program Application and Data API Vision

Identify the key services that can be offered through APIs. This highlights the main capabilities of the application component.

The purpose of any **vision statement** is to ensure that the organization is aligned to a common goal. Visions are a key component of how we communicate. They convey why each organization exists and how value is added to their ecosystem.

The vision statement for your data should support and be consistent with your organization's overall vision. It provides the answer to "Why do we care about data?" as well as "Why do the data organization and processes exist?"

A data vision statement provides alignment for the short and long term. It is the baseline or "North Star" for your data strategy. The absence of a data vision usually leads to a series of problem issues and costly clean-up efforts.

T.2 Key Data and Services

Identify use cases for each API along with endpoints for each use case. These use cases provide a basis to create an API Roadmap. Your API roadmap should address all the types of APIs you intend on creating—internal, partner, or public. It should also include key objectives, such as digital transformation or APIs as products.

T.3 Key Stakeholders

Determine the API stakeholders within the organization. Ensure that all stakeholders share a common vision and provide their inputs on the API design. Collaborate consistently with API stakeholders and consumers to ensure that they can easily use the API.

You need to figure out where you want your API program to go before applying governance, understand the data domain, mission and capability needs, identify who works on creating APIs and track their workflow, and interview leaders and stakeholders who will help you map out the objectives and issues of your API program.

T.4 Driving Architectures

Describe operational environment (within mission context and how it may be accessed offline or out of mission). Will the application be cloud based or ran on local systems? What networks and as well as operational considerations such as Denied, Degraded, intermittent or Limited (DDIL) communications. In addition, API architecture must be considered.

API architecture refers to the technical framework of developing a software interface that exposes backend data and application functionality for use in external applications. An API architecture consists of components for external interfacing, traffic control, runtime execution of business logic, and data access.

API security is the process of protecting APIs from attacks. Because APIs are very commonly used, and because they enable access to sensitive software functions and data, they are becoming a primary target for attackers. API security is a key component of modern web application security.

T.5 API Governance

API governance is the of applying rules and policies to your APIs. You create processes that standardize various aspects of your APIs. The major activities that API governance includes are:

- Program Level Governance
- Service and/or Enterprise Governance
- External Communication
- Hosting
- Security
- Lifecycle
- Testing

Program/Service and/or Enterprise Governance needs to describe management approaches and minimum features to ensure standardization across the program and associated developments. This should include:

- Descriptions
- API Contracts
- Designs
- Protocols
- Quality Reviews

This standardization helps ensure that all your APIs remain consistent even when different developers design and build them. An effective API governance program helps organizations ensure that every API is high-quality and discoverable—whether they create a few APIs or a few thousand.

Achieving consistency across your APIs means that you can reuse existing components. And part of governance is tracking APIs and making them so that developers can easily find existing API artifacts and reuse them in future designs. This means that developers only have to build components once and won't end up duplicating code.

When you enforce standards, you prevent developers from having to reinvent the wheel. They can spend more of their time on tasks that benefit the business, like building new services. API governance also helps keep everyone involved in your API program on the same page. When stakeholders have misunderstandings about API goals or designs, it can cause API programs to fail. Security must be built into to the foundation of the API development and deployments, here is a checklist of things to consider.

API Security Checklist

- Authentication: Use highly robust, approved authentication standards (e.g., JWT, OAUTH2).
- Authorization: Incorporate role-based access control (RBAC) and attribute-based access control (ABAC).
- API Access: Limit number of requests (e.g., per minute), Throttling, and enforce Quotas to avoid DDoS/brute-force attacks. Use approved encryption (e.g., TLS1.2) to protect server side data in flight and protect against Man-In-the-Middle (MITM) attacks. Use defensive code techniques to validate passed parameters against known exploits (e.g., file manipulation, SQL injection)
- Requests: Use the proper HTTP method according to the operation: GET (read), POST (create), PUT/PATCH (replace/update), and DELETE (to delete a record).
- Processing: Check if all the endpoints are protected behind authentication to avoid broken authentication process.
- Responses: Ensure request responses are using industry standard HTTPS status codes (e.g., 404 Not Found, 405 Method Not Allowed, 413 Payload Too Large). Use pagination to limit the

maximum number of results per request to limit performance spikes across your system infrastructure (e.g., compute, network).

- CI/CD Monitoring and Auditing: Ensure system infrastructure is operating nominally. Ensure all critical errors are being logged and alerted for rapid response and recovery. Review your design and implementation with unit/integration tests coverage. Use a code review process and do not allow self-approval.

T.6 API Technical Details/Key Elements

API Specification: An API specification tells developers what an API does. The API specifications can be written in a machine-readable standardized syntax (e.g., OpenAPI, AsynchAPI). In addition, it's helpful to generate human-readable reference documentation based on the API specification. If your organization follows an approach to building APIs, then you can easily incorporate API specifications into your governance process.

Comprehensive Style Guide: Governance is about creating consistency, and one of the best ways to ensure consistency across your APIs is to create a complete style guide. Your style guide should eventually cover nearly every aspect of an API. Start your style guide out small and build it out over time. You should include things like architectural style, API description format, naming conventions, and error handling. We plan on launching soon (check back in a few months) to help make creating, sharing, and automating style guides easier.

Central API Catalog: The more extensive your API portfolio becomes, the harder it gets to discover your existing APIs. You need to create a catalog that contains all your API design assets and dependencies. With a catalog in place, developers can search for and find APIs currently in use. You can include in the catalog collections of assets, such as API descriptions and JSON schemas. You can't use what you can't find, so making your APIs discoverable is crucial to effective governance.

Reusable API Components: In addition to a catalog of existing APIs, governance programs can make individual components of APIs intentionally browsable. To encourage consistency in API designs, you can include headers, query parameters, models, and even pieces of models in a library. There are a number of ways these can be stored (including Stoplight's Design Libraries-COMING SOON), most important is that they are discoverable. You want new APIs to be able to build on the team's previous work.

SLAs: Many teams building internal APIs may not think of service-level agreements as necessary. However, your governance program should distinguish the APIs that are mission-critical from those that can have downtime. How you maintain and improve your APIs could be determined by how available it needs to be. Consider classifying a tier of service for each API and make it clear who is responsible for maintaining that level of service.

API Guidance: The API Guidance document specifies the API coding guideline and best practices that need to be adhered to during the development. Include all APIs in the contract document. Check for consistency in endpoint names, error codes, and URLs in all the APIs.

API Metrics and Telemetry:

Infrastructure Metrics

Many of these metrics are the focus of Application Performance Monitoring (APM) tools and infrastructure monitoring companies like Datadog.

Uptime

While one of the most basic metrics, uptime or availability is the gold standard for measuring the availability of a service. Many enterprise agreements include a SLA (Service Level Agreement), and uptime is usually rolled up into that. Many times, you'll hear terms like triple 9's or four 9's which is a measure of how much uptime vs downtime there is per year.

CPU Usage

CPU usage is one of the most classic performance metrics that can be a proxy to application responsiveness. High Server CPU usage can mean the server or virtual machine is oversubscribed and overloaded or it can mean a performance bug in your application such as too many spinlocks. Infrastructure engineers use CPU usage (along with its sister metric, memory percentage) for resource planning and measuring overall health. Certain types of applications like high bandwidth proxy services and API gateways naturally have higher CPU usage than other metrics along with workloads that involve heavy floating point math such as video encoding and machine learning workloads.

Memory Usage

Like CPU usage, memory usage is also a good proxy for measuring resource utilization as CPU and memory capacity are physical resources unlike a metric which may be more configuration dependent. A VM with extremely low memory usage can either be downsized or have additional services allocated to that VM to consume additional memory. On the flip side, high memory usage can be an indicator of servers overloaded. Traditionally, big data queries/stream processing and production databases consume much more memory than CPU. In fact, the size of memory per VM is a good indicator for how long your batch query can take as more memory available can reduce checkpointing, network synchronization, and paging to disk. When looking at memory usage, you should also look at the number of page faults and I/O ops. An easy mistake to make is an application that's configured to allocate at a maximum only a small fraction of available physical memory which can cause artificially high page virtual memory thrashing.

Application Metrics

Request Per Minute (RPM)

RPM (Requests per Minute) is a performance metric often used when comparing HTTP or database servers. Usually, your end to end RPM will be much lower than an advertised RPM, which serves more as an upper bound for a simple "Hello World" API. Since a server will not consider latency incurred for I/O operations to databases, 3rd party services, etc. While some like to brag about their high RPM, an engineering team's goal should be efficiency and attempt

to drive this down. Certain business functions that require many API calls can be combined into fewer number of API calls to reduce this number. Common patterns like batching multiple requests in a single request can be very useful along with ensuring you have a flexible pagination scheme.

Average and Max Latency

One of the most important metrics to track customer experience is API latency or elapsed time. While an increase in infrastructure level metrics like CPU usage may not actually correspond to a drop in user perceived responsiveness, API latency definitely will. Tracking latency by itself may not give you full understanding of why an increase occurred. It's important to track any change on your API such as new API versions being released, new endpoints added, schema changes, and more to get to a root cause why latency increased.

Errors Per Minute

Similar to RPM, Errors per Minute (or error rate) is the number of API calls with non 200 family of status codes per minute and is critical for measuring how buggy and error-prone your API is. In order to track errors per minute, it's important to understand what type of errors are happening. 500 errors could imply bad things are happening with your code whereas many 400 errors could imply user errors from a poorly designed or documented API. This means when designing your API, it's important to use the appropriate HTTP status code.

Product Metrics

APIs are no longer just an engineering term associated with microservices and SOA. APIs as a product is becoming far more common especially among B2B companies who want to one up their competition with new partners and revenue channels. API-driven companies need to look at more than just engineering metrics like errors and latency to understand how their APIs are used (or why they are not being adopted as fast as planned). The role of ensuring the right features are built lies on the API product manager.

API usage growth

For many product managers, API usage (along with unique consumers) is the gold standard to measure API adoption. An API should not be just error free, but growing month over month. Unlike requests per minute, API usage should be measured in longer intervals like days or months to understand real trends. If measuring month-over-month API growth, we recommend choosing 28-days instead as it removes any bias due to weekend vs weekday usage and also differences in number of days per month. For example, February may have only 28 days whereas the month before has a full 31 days causing February to appear to have lower usage.

Unique API consumers

Because a month's increase in API usage may be attributed to just a single customer account, it's important to measure API Monthly Active Users (MAU) or unique consumers of an API. This metric can give you an overall health of new customer acquisition and growth. Many API platform teams correlate API MAU to their web MAU, to get a full product health. If web MAU is growing far faster than API MAU, then this could imply a leaky funnel during integration or

implementation of a new solution. This is especially true when the core product of the company is an API such as for many B2B/SaaS companies. On the other hand, API MAU can be correlated to API usage to understand where that increased API usage came from (New vs. existing customers).

Top customers by API usage

For any company with a focus on B2B, tracking the top API consumers can give you a huge advantage when it comes to understanding how your API is used and where upsell opportunities exist. Many experienced product leaders know that many products exhibit power law dynamics with a handful of power users having a disproportionate amount of usage compared to everyone else.

API retention

Should you spend more money on your product and engineering or put more money into growth? Retention and churn (the opposite of retention) can tell you which path to take. A product with high product retention is closer to product market fit than a product with a churn issue. Unlike subscription retention, product retention tracks the actual usage of a product such as an API. While the two are correlated, they are not the same. In general, product churn is a leading indicator of subscription churn since customers who don't find value in an API may be stuck with a yearly contract while not actively using the API. API retention should be higher than web retention as web retention will include customers who logged in, but didn't necessarily integrate the platform yet. Whereas API retention looks at post-integrated customers.

Time to First Hello World (TTFHW)

TTFHW is an important KPI for not just tracking your API product health, but your overall developer experience aka DX. Especially if your API is an open platform attracting 3rd party developers and partners, you want to ensure they are able to get up and running as soon as possible to their first ah ha moment. TTFHW measures how long it takes from the first visit to your landing page to an MVP integration that makes the first transaction through your API platform. This is a cross-functional metric tracking marketing, documentation and tutorials, to the API itself.

API Tokens Broken Down by Acquisition Channel

API Calls Per Transaction

While more equals better for many product and business metrics, it's important to keep the number of calls per business transaction as low as possible. This metric directly reflects the design of the API. If a new customer has to make 3 different calls and piece the data together, this can mean the API does not have the correct endpoints available. When designing an API, it's important to think in terms of a business transaction or what the customer is trying to achieve rather than just features and endpoints. It may also mean your API is not flexible enough when it comes to filtering and pagination.

API Version Adoption

Many API platform teams may also have many APIs integrations they maintain. Unlike mobile where you just have iOS and Android as the core mobile operating systems, you may have 10's or even hundreds of variations. This can become a maintenance nightmare when rolling out new features. You may selectively roll out critical features to your most popular APIs whereas less critical features may be rolled out to less popular APIs. Measuring API version is also important when it comes to deprecating certain endpoints and features. You wouldn't want to deprecate the endpoint that your highest paying customer is using without some consultation on why they are using it.

T.7 Exceptions

Identify any know exceptions for the program and describe the process for handling exceptions to the use of APIs to expose data or services.

5. Further Resources

OpenAPI Specification Version 3 https://spec.openapis.org/oas/v3.0.3

GSA API Directory https://open.gsa.gov/api/

What is API Governance? 8 Best Practices for API Governance Success
https://www.digitalml.com/api-governance-best-practices/

An API Governance Model for Great APIs https://www.digitalml.com/api-governance-model/

gartners_api_strate
gy_maturi_451168.p

API%20Guidance%2
0v1.7.1-DaaS-v1.3.d

DaTS White Paper
Draft_Finalized Tech

{Program} API Strategy

Programs using the SWP should use APIs to the maximum extent practicable. Programs should capture their API strategy as part of the Acquisition Strategy (interoperability section) or related strategy document. The strategy should outline the high-level plan, which will evolve over time, while supporting documents can capture the technical details. The following is an initial outline/template for SWP programs to consider capturing their API strategy. As always, SWP programs are encouraged to tailor their strategies to the unique aspects of their program and environment.

1.0 Program Application and Data API Vision

Brief description of the programs goals for exposing their applications and data.

2.0 Key Data and Services

Identify key data and service (consumed or produced) based on the CNS. Identify features or restrictions associated with them and describe how APIs will be used to address these.

3.0 Key Stakeholders

Identify key stakeholders for the services and data consumed and produced by the system as well as user or functional communities which may need to be coordinated.

4.0 Driving Architectures

Describe operational environment (within mission context and how it may be accessed offline or out of mission). Will the application be cloud based or ran on local systems? Networks and DDIL considerations. Direct access, use of API gateways, public or partner APIs. Any major security approaches.

5.0 API Governance -how will the APIs be approved and managed

5.1 Program Level

Describe how much autonomy will product teams have on API development. APIs managed with each product branch or maintain in an API environment. If using API contracts, describe how the API contacts will be managed.

5.2 Service and/or Enterprise Governance

Describe any Service or other external governance groups and general workflow for communicating.

5.2 External Communication

Describe if program will leverage a Service or DOD enterprise API solution. Describe the release process for publishing/releasing APIs. Describe how the meta data will be exposed to enable API discovery.

5.3 Hosting

Describe how APIs will be hosted (both internal and external).

5.4 Security

General discussion on key security and classification issues that APIs must address and key security services that will be leveraged. Address how ATOs will be established for external APIs (how to ensure reciprocity).

5.5 Lifecycle

Describe approach for releasing, versioning, compatibility, and removal of APIs.

5.6 Testing

Describe programs approach to test APIs (continuously, with each release, or other). Should include how automation will be used and how externally hosted versions will be addressed.

6.0 API Technical Details/Key Elements

Specify the required features within each API and the program use API contracts to model compliance to SLAs (and/or KPPs) for critical performance needs. Include error and failure handling, telemetry, and metrics.

7.0 Exceptions

Identify any know exceptions for the program and describe the process for handling exceptions to the use of APIs to expose data or services.