|  |  |
| --- | --- |
|  | Overview  This section of the API Reference Architecture contains information on some foundational development concepts.  ❊❊❊ Also refer to [Cloud Native](https://docs.google.com/document/u/0/d/1OYjI5x-PCJj6Qvof6_VcRt25yJ2FGcRMuyz7Q5sdvQ4/edit) patterns and principles ❊❊❊  [Security](#_k3a4uo33blxx)  [Security Considerations](#_6bbsnci83ovc)  [APIs and Risk](#_k5nesb6udyqx)  [Types of Attacks to Consider](#_p45ee4h4vx4x)  [Parameters](#_ioedzk6vg01g)  [Identity](#_sggj7byt92ku)  [Man-in-the-Middle](#_t04w1af9e5is)  [Security Strategies](#_emteycey8y89)  [Validate Parameters](#_rfsly88n5fg4)  [Apply Explicit Threat Detection](#_r01ah09x1m7d)  [Transport Protocol](#_tdhn5jj3hodh)  [Strong Authentication and Authorization](#_eogf4nb9nu0m)  [Secure API Architectures](#_k7rk67lyag4e)  [Reuse vs Independence](#_111kx3o)  [Privacy](#_jwtc3qmcuvr3)  [Reliability or Availability](#_4ocrycjirkig)  [Performance by Design](#_34p6s2j3vwc2)  [Supportable Systems](#_2ddxqma523wv)  [Traceability](#_nbhfjdvq3n4x)  [Benefits of Traceability](#_qa4jw98wviww)  [Observability](#_3oj3du5o2kr)  [Documentation](#_ho2xp17jrhkw)  [API should be documented clearly stating:](#_kji49s3d46cn)  [Benefits](#_pg9g7bsntilw)  [Discovery](#_frcc4d1jx32)  [Clean, Readable Code](#_58thlp5oa6hq)  [Data Management, Quality and Accuracy](#_206ipza)  [Data Governance](#_1fczbl19dnae)  [Eventual Data Consistency and Persistence](#_3sq4tyimcc17)  [References](#_w4l1qo8rz09d) |

# Security

## Security Considerations

Any Architecture must provide for the security of the enterprise and an individual’s information. The privacy and security of an individual's information is regulated by the government and cannot be compromised. Security must be addressed at all levels and in all components of the architecture. If any one component is not secured it could be exploited for malicious intent.

APIs allow for an open architecture and the ability to share data and capabilities across the enterprise.

The design of the API interface, its boundary, and granularity describe the scope of how much of a back-end system a consumer can access. An API’s URL often provides a clear indication of the API’s capability. The benefit of APIs to expose capabilities of the enterprise in a reusable and easy to consume manner also makes APIs perfect targets for hackers. This is one reason an enterprise that adopts an API architecture must adopt an architecture that is secure in design, implementation, and operation.

All APIs and the supporting infrastructure must be designed for security. A well-defined security strategy enables an organization to pursue an API strategy more safely and securely as well as realize the benefits of agile integration. APIs allow the development of an open architecture where capabilities and data are shared between applications. APIs provide a view of the data that the application wants to make accessible to other APIs within the enterprise.

## APIs and Risk

The APIs’ benefit of providing a design that provides an indication of its capability also provides insight into the underlying implementation of the application. These details that might otherwise be hidden in the web application are now exposed and can provide hackers with clues that could lead to attack opportunities that might otherwise have been overlooked. APIs tend to be extremely clear and self-documenting at their best, providing insight into internal objects and even internal database structure which can be valuable information for hackers.

Increased visibility is not the only risk APIs introduce. Typically there is an increase in the number of requests across more instances which means there are more instances to exploit.

## Types of Attacks to Consider

There are a number of types of attacks that can be used by hackers. The following is not an exhaustive list, as hackers are always discovering new techniques, but rather some more common types of attacks to be aware of and to design a security strategy to prevent.

Details on how to design and develop for security is provided in the Design Best Practices and Development Best Practices documents.

### Parameters

Parameter attacks exploit the data sent into an API, includingURL, query parameters, HTTP headers, and/or post content. The most common of this type is SQL injection where an attempt is made to manipulate the system by providing inputs that exploit the behaviour of databases and other components.

These attacks usually result when developers do not carefully validate inputs. In order to make APIs easy to consume and self-documenting, API design usually makes parameters and their use clear through their names. The parameter names provide a window into the API’s use to the attacker.

Parameter attacks certainly are not new, these attacks have threatened the web for years. However, because developers are often accustomed to their web frameworks doing validation for them in the web application world, they may neglect to do thorough validation for their APIs. The same risks that impact web applications also apply to APIs, and the same precautions need to be taken to mitigate this threat.

### Identity

Identity attacks exploit flaws in authentication, authorization, and session tracking. In particular, many of these are the result of migrating bad practices from the web world into API development.

User identity is the logical identity of a user of the system. It is used by systems to identify and distinguish between users that access the system. Identity information must be secured at all times: at rest and in transport.

APIs also introduce the concept of application identity, which is a key that uniquely identifies the application that is calling the API. This key is referred to as an API key. API keys are replicated across every instance of an application with the intent to manage consumer use of the API through policies like rate limiting. API keys are difficult to completely secure especially if they are hidden within the applications. API keys are not credentials and should not be considered such.

### Man-in-the-Middle

These attacks intercept legitimate transactions and exploit unsigned and/or unencrypted data being sent between the client and the server. They can reveal confidential information (such as personal data), alter a transaction in flight, or even replay legitimate transactions. A man-in-the-middle attack describes a situation in which an attacker sits in between a sender and a receiver of information. They may do this transparently, or they may explicitly pose as one party or the other, but in both cases they use this position to exploit the exchange of unsigned or unencrypted data.

APIs that are not properly configured using a secure transport protocol are highly vulnerable to this form of attack. In the API world, the stakes are higher and transport protection is essential to secure data, sessions, and access to functionality.

## Security Strategies

### Validate Parameters

The first step for any resilient API implementation is to sanitize all incoming data to confirm that it is valid and will not cause harm. The single most effective defense against parameter manipulation and injection attacks is to validate all incoming data against a strict schema,

which provides a description of what are considered permissible inputs to the system. Schema validation should be as restrictive as possible, using typing, ranges, sets, and even explicit white listing whenever possible.

The API reference architecture consists of the application of the TMF data model. These well-defined data models provide a description of the API parameters and inputs for the core API data model. All extensions applied to an API should also be documented. The TELUS API data model will be managed within the TELUS data management system providing management, governance and discoverability of API data models. Governing the API data models also provides for security governance to ensure APIs are not exposing confidential information to a security risk.

### Apply Explicit Threat Detection

Good schema validation can protect against many injection attacks, but consider also explicit scanning for common attack signatures. SQL injection or script injection attacks often betray themselves by following common patterns that are easy to spot by scanning raw input.

Consider also that attacks may take other forms, such as a denial of service (DoS). We have networking infrastructure that can spot and mitigate network-level DoS assaults, but it is important to check for DoS attacks that exploit parameters. Very large messages, heavily nested data structures, or overly complex data structures can all result in an effective denial-of-service attack that needlessly consumes resources on an affected API server. Apply virus detection to all potentially risky encoded content. APIs involved in file transfer should decode base64 attachments and submit these to server-grade virus scanning before persisting to a file system where they could be inadvertently activated.

### Transport Protocol

SSL/TLS provides integrity on all data exchanged between a client and a server, including important access tokens such as those used in OAuth. It optionally provides client-side authentication using certificates, which is important in many environments.

### Strong Authentication and Authorization

User and application identity are concepts that must be implemented and managed separately.

Authorization should be applied across all levels of identity including incoming IP address, access time, device identification, geolocation, etc. OAuth should be applied for user-centric API authorization. Security should be applied using proven solutions and not created in house.

### Secure API Architectures

Secure APIs from any type of intrusion by separating API implementation and API security into distinct tiers. This follows the architectural principle of separation of concerns and allows each component to apply the correct solution for its specific concern. This approach allows the API developer to focus completely on the application domain, ensuring that each API is well-designed and promotes integration between different apps. Security is then addressed by the security expert who can focus solely on identity, threats, and data security.

Security for the APIs is provided at a number of levels:

* All APIs are registered within the API repository
* All APIs are secured by the API gateway with the application of rigid security protocols and policies
* All implementations are secured to prevent direct access
* All data is secured both at rest and in transport

# Reuse vs Independence

* using existing databases and components
* shared libraries

# Privacy

# Reliability or Availability

Microservice architecture can improve the reliability and availability of functions through their characteristics. Microservices are autonomous and independently deployable. They depend on their own data and they communicate through events rather than being dependent on any other component.

When one service orchestrates or relies on another to provide a response in a sequence of requests it makes it dependent on the reliability, availability, and performance of others. A microservice architecture and the decoupling of one API from another is further augmented by the use of choreography. This architecture can improve reliability, availability and performance.

Design and development best practices for reliability and availability are contained within the Design Best Practices, Development Best Practices, and supported by the Architecture Patterns proofs of concept.

## Performance by Design

# Supportable Systems

## Traceability

*“Traceability is the capability to trace something. In some cases, it is interpreted as the ability to verify the history, location, or application of an item by means of documented recorded identification.”*

In order to contribute to the goal of supportable systems API calls and their relationship to other components should be traceable using a unique identifier. APIs and the API lifecycle Management Platform should provide the capability to keep track of the use and provide the ability to trace along a value chain.

One means of enabling this capability is to ensure each API within the enterprise can be uniquely identified using a unique identifier. Consumers of the APIs should also be uniquely identified so that the relationship between APIs and their consumers can be understood and reported on.

It would further enhance the Enterprise’s ability to maintain its resource inventory for proactive ticketing and impact analysis by implementing run time discovery of the APIs and Consumers so the enterprise resource inventory could be kept up to date.

### Benefits of Traceability

* Impact Analysis of System Change
* Reduction in Time to resolution
* Proactive Ticketing
* Self-healing
* Value Chain Analysis

## Observability

## Documentation

API documentation should be in a standard format using clear, concise, and familiar language so that stakeholders can easily understand the purpose and specification of the API. An API design should be intuitive, however additional documentation can clarify the purpose and use. Proper documentation reduces the time for consumers to understand and acquire use of the API.

Sufficient documentation also facilitates onboarding new developers, or handoff to new development teams, avoiding the impediment of escalating operational costs and time.

### API should be documented clearly stating:

* API Capability
* Domain context, sub-context of specialization
* UML diagram
* Interface specification
  + Resources
  + Schemas
  + Resource Models
  + Error message specification

### Benefits

* Standard documentation is easier to navigate and understand
* Reduces time to market by making it easy to understand and use
* Reduces system analysis time
* Reduces development time
* Improves Agility
* Increase interoperability

## Discovery

API should be discoverable so that potential consumers can find available APIs. Discovery can be both internal and external. Internal and External discovery may be served by different experiences and may require the application of different policies.

## Clean, Readable Code

Clean, logical code (ideally, based on existing standards) streamlines future development cycles. Developers can easily skim through the interface design to understand included operations and their dependencies, to quickly zero in on what needs updating.

# Data Management, Quality and Accuracy

While APIs unlock the value of enterprise data, they can also improve the management and quality of data that is shared between bounded contexts. All data within a bounded context can be more easily governed and the data consistently exposed to other bounded contexts. As an example, when all Resource Management applications are managed within a Resource Management domain context, the data exposed from the domain context can be a normalized data model and populated with sanitized data. In contrast when all consumers have direct access to any of the Resource management applications, one consumer may use data from one application and another consumer from a different application resulting in inconsistencies of the data model and data exposed to client facing applications. In this case the quality of the data can be inconsistent to the client facing application as well. Data quality directly affects our customer experience.

The adoption of the industry standard Open API data model and well governed extensions will help ensure data quality and accuracy through data model and data governance.

### Data Governance

Data governance will be more critical in a microservice architecture which includes local persistence of data. All API data models including those that represent the events produced and consumed by APIs must be standardized and governed. The metadata of an APIs locally persisted data store should be governed as any other data store. Data governance best practices should be applied as directed by the Data Management Enablement Team.

### Eventual Data Consistency and Persistence

With the application of Microservices, each with their own local persistence, and Event Driven Architecture comes the notion of eventual consistency of an enterprise master data store or legacy data store.

As events are produced and consumed, the local persistence of the microservices are updated so that the changes are immediately available to their consumers. Data persistence to underlying master data is considered to be eventually consistent. This decouples large monolithic data stores from end user applications providing for highly performant applications. This decoupling also allows back end systems to change at a slower rate and provides the ability to address redundant and legacy back end systems.

In the same way that data changes applied in the local persistence of the microservice are sent to the downstream data stores, changes that are applied to the downstream data stores must be reflected in the local persistence of the microservice. For example a microservice in the cloud may have changes to customer data as provided by a system of engagement while an on-prem legacy customer data store may have changes applied from on-prem legacy application that manages customer data.

Event driven Architecture can also be applied to push events up from legacy systems to on cloud microservices. This provides for data consistency and persistence during a transformation to cloud and taking into account some on-prem systems may be required to support different stakeholders or capabilities. The data must be persisted between the two data stores and the events created in cloud and on prem must also be persisted so each system, cloud and on-prem, can react to the events.

# References

* <https://en.wikipedia.org/wiki/ISO/IEC_9126>
* <https://12factor.net/>
* <http://www.cs.utoronto.ca/~matt/csc302/lectures/15_quality_1.pdf>
* [https://coggle.it/ diagram/ Xk0S2jjTUiGCYc6P/t/ product-quality-explained-iso-iec-25010-2011](https://coggle.it/diagram/Xk0S2jjTUiGCYc6P/t/product-quality-explained-iso-iec-25010-2011?present=1)
* <https://www.iso.org/standard/35733.html>
* PL151 - page around solution blueprint - references ISO 25010
  + has been part of TELUS Methods for quite a while
* Dev Ops Evolution team

