Schedule 1: Annex C

Design Principles and Policies

26th July 2024 and

Version 1.02

Contents

DESIGN PRINCIPLES OVERVIEW

The RECCo Design Authority ensures that all digital services and solutions align with RECCo's strategic goals and objectives. This authority encompasses five critical design functions:

1. Technology

2. Application

3. Data

4. Business

5. Enterprise

All design principles outlined ensure relevance and alignment with the criticality of the solutions. The DXP portal and API Gateway are considered mission-critical solutions, while the rest are classified as business-critical. Non-live environments represent the third criticality level. Solutions may move from non-live to mission-critical status throughout their lifecycle. These principles provide a framework for design but emphasise collaboration with RECCo to finalise designs.

The key focus areas of these design principles are Scalability, Availability, Performance, and Reliability. By prioritising these four core principles, RECCo aims to build robust and efficient digital solutions that can support the dynamic needs of the organisation and its stakeholders.

Scalability:

- Ensure systems are elastic and can scale seamlessly to accommodate growth and changing demands without compromising performance.

Availability:

- Design architectures to provide high availability, ensuring systems are always accessible and operational.

Performance:

- Optimise systems for high performance, ensuring fast response times and efficient resource utilisation.

Reliability:

- Implement measures to ensure the reliability of data within the solution, focusing on error handling, data validation, and maintaining data integrity.

During implementation, RECCo may ask the Digital Service Partner to demonstrate how the solution, platform, or tools align with these design principles. This ensures that all components meet the required standards and contribute to the overall strategic objectives of RECCo.

These foundational principles guide the RECCo Design Authority in shaping a resilient and future-proof IT landscape that drives sustainable business success.

DESIGN FUNCTIONS

1. Technology Design Principles

Overview

The Technology Design Principles guide the development and management of the technological infrastructure within RECCo. These principles ensure that all technology-related initiatives support scalability, availability, performance, and reliability, aligning with RECCo's strategic goals and criticality levels.

Principles

- Seamlessly Scalable:

Scale seamlessly to accommodate growing demands and changing business needs. This includes the ability to scale without causing disruptions to services.

- Resilient Platforms:

Implement redundant components, failover mechanisms, and disaster recovery plans to minimise downtime and maintain business continuity.

- On Demand Elasticity:

Design platforms to be elastic, allowing resources to scale up or down automatically in response to fluctuations in demand.

- Observability & Telemetry:

Monitor infrastructure performance metrics and user experience in real-time to identify performance bottlenecks and optimise resource allocation, including reporting.

- Multi Cloud:

Adopt a multi-cloud strategy to be cloud-agnostic, where possible, enabling seamless migration and workload portability across multiple cloud environments.

- Cloud Native Architectures:

Embrace patterns such as microservices, serverless computing, and event-driven architectures to maximise agility, scalability, and innovation.

By following these principles, RECCo ensures that its technological foundation is robust, scalable, and adaptable, capable of supporting the organisation's evolving needs and strategic goals.

2. Application Design Principles

Overview

The Application Design Principles guide the development and management of applications within RECCo. These principles ensure that all applications are designed to be available, scalable, performant, and reliable, aligning with RECCo's strategic goals and criticality levels.

Principles

- Simplified UX:

Prioritise user experience by designing intuitive and user-centred interfaces. Ensure that every aspect of the design—from interface layout to interaction flows—is optimised for ease of use.

- Flexibility at its Core:

Design applications to be adaptable to changing business requirements and technological advancements. This ensures minimal rework or disruption when modifications are needed.

- Maintainability:

Develop applications with maintainability in mind. Use clean code, clear documentation, and standard practices to facilitate ongoing development and troubleshooting.

- Interoperability:

Ensure applications support open standards and APIs for seamless integration with other systems and platforms. This promotes easy data exchange and interoperability.

- Microservices:

Adopt a microservices architecture where possible. Break down applications into small, independent services that can be developed, deployed, and scaled independently.

- Meets Performance Needs:

Optimise applications for performance, ensuring fast response times and efficient resource utilisation. Applications should deliver a seamless user experience under varying load conditions.

- Technical Fit:

Ensure applications are compatible with RECCo's preferred platforms, ideally cloud-enabled and following a SaaS model. This compatibility supports scalability and modern deployment practices.

- Adoption/Track Record:

Choose application solutions with a proven track record and ensure there is suitable ongoing support. This guarantees reliability and continued performance.

By adhering to these principles, RECCo ensures that its applications are robust, user-friendly, and capable of evolving with the organisation's needs, supporting long-term strategic goals and operational efficiency.

3. Data Design Principles

Overview

The Data Design Principles guide the management and utilisation of data within RECCo. These principles ensure that data is reliable, accurate, and secure, supporting scalability, availability, performance, and reliability in alignment with RECCo's strategic goals and criticality levels.

Principles

- Data Quality:

Ensure data is accurate, consistent, complete, and relevant to support decision-making and business processes. High-quality data is essential for reliable analytics and reporting.

- Data Lifecycle Management:

Define processes for managing data throughout its lifecycle, including data acquisition, storage, retention, archival, and disposal. Proper lifecycle management ensures data remains useful and compliant over time.

- Single Source of Truth:

Establish a single source of truth for key data entities (e.g., customers, products) to ensure consistency and accuracy across all systems and services. This minimises data discrepancies and errors.

- Data Centric & Analytics Driven:

Design systems to be data-centric, leveraging intelligent analytics to unlock the full value of data. This includes building in reporting capabilities and key metric tracking to support data-driven decision-making.

- Data Governance:

Implement policies, processes, and procedures for effective data management. This includes defining data ownership, stewardship, and compliance to ensure data integrity and accountability.

- Data Security:

Protect data confidentiality, integrity, and availability through robust security measures. This includes encryption, access controls, and authentication mechanisms to safeguard data from unauthorised access and breaches.

By adhering to these principles, RECCo ensures that its data management practices support the organisation's strategic objectives, providing a reliable foundation for informed decision-making and operational excellence.

4. Business Design Principles

Overview

The Business Design Principles guide the alignment of technological solutions with RECCo's business objectives and strategies. These principles ensure that all business processes and systems are designed to support scalability, availability, performance, and reliability, aligning with RECCo's strategic goals and criticality levels.

Principles

- Alignment with Business Goals:

Ensure that all architectures and solutions support RECCo's business strategies and deliver measurable value. Solutions should be designed to enhance business growth and efficiency.

- Innovation:

Encourage creative and innovative solutions within established best practices. This drives continuous improvement and keeps RECCo at the forefront of technological advancements.

- Governance:

Provide a consistent framework for making architectural and business decisions. This ensures coherence and alignment with organisational objectives, policies, and standards.

- Quality:

Promote high-quality architectures by considering critical attributes such as performance, scalability, security, and maintainability. Quality should be integral to all business solutions.

- Risk Mitigation:

Provide guidelines for addressing common challenges and pitfalls, reducing the likelihood of architectural flaws and technical debt. This includes proactive identification and mitigation of business risks.

By adhering to these principles, RECCo ensures that its business processes and systems are robust, efficient, and aligned with the organisation's long-term strategic goals. This alignment facilitates sustainable growth and operational excellence.

5. Enterprise Design Principles

Overview

The Enterprise Design Principles guide the overall architectural framework and strategic alignment of technological initiatives within RECCo. These principles ensure that the enterprise architecture supports scalability, availability, performance, and reliability, aligning with RECCo's strategic goals and criticality levels.

Principles

- Cloud First:

Prioritise cloud-native architectures and services for scalability, agility, and cost-effectiveness. Leverage cloud computing resources to support dynamic workloads, rapid innovation, and global scalability while optimising resource utilisation and reducing infrastructure overhead.

- Secure by Design:

Emphasise security as a fundamental aspect of architecture from the outset. Integrate security controls, encryption, access management, and threat detection mechanisms into every layer of the architecture to protect data and systems from cyber threats and vulnerabilities.

- Compliance & Governance:

Incorporate compliance and governance requirements into architecture design. Adhere to industry regulations, standards, and best practices to ensure data privacy, regulatory compliance, and ethical use of technology while maintaining transparency, accountability, and trust with stakeholders and customers.

- Resilient & Fault Tolerant:

Architect for resilience and fault tolerance by implementing redundancy, failover mechanisms, and disaster recovery strategies. Ensure high availability, reliability, and business continuity, minimising downtime, service disruptions, and data loss in the face of failures or disruptions.

- Open Standards & Modularity:

Design architectures with open interfaces and standards to facilitate seamless integration with external systems, partners, and third-party services. Promote flexibility by breaking down systems into smaller, interoperable modules to help RECCo adapt more easily to changing requirements.

- Clear Ownership Within:

Retain ownership to ensure clarity, accountability, control, and alignment around assets. This is essential for maximising value and mitigating risks. Effective asset management enables RECCo to achieve its strategic objectives and drive success in digital initiatives.

By adhering to these principles, RECCo ensures that its enterprise architecture is robust, scalable, and adaptable, capable of supporting the organisation's evolving needs and strategic goals.

6. Role of the Design Authority (DA)

Overview

The Design Authority (DA) in RECCo is responsible for ensuring that any new solution, platform, or tool introduced to the RECCo digital ecosystem, or any existing one moving up or down in criticality, adheres to the established design principles and strategic goals of RECCo. Operating with a streamlined governance structure, the DA involves key stakeholders in reviewing technical decisions to ensure alignment with business objectives.

The DA plays a critical role in fostering cross-functional collaboration and adapting quickly to changes, promoting consistency and mitigating risks. This approach ensures the integrity and alignment of technological initiatives with the organisation's objectives.

The key benefits of the DA include alignment with business goals, risk mitigation, consistency in technical solutions, efficient resource allocation, and enhanced collaboration and communication. These benefits contribute to driving the right level of change to support RECCo's vision.

Key Responsibilities

- UX Focused:

Prioritise user experience and usability in technical solutions. Ensure that solutions are intuitive, accessible, user-friendly, and meet the needs and expectations of RECCo end-users.

- Governance & Compliance:

Establish and enforce governance mechanisms to ensure that technical decisions adhere to established standards, policies, and regulatory requirements. Maintain compliance with industry regulations and best practices.

- Risk Management:

Identify, assess, and mitigate technical risks associated with IT projects and initiatives. Proactively address potential vulnerabilities, dependencies, and failure points to ensure the reliability, security, and resilience of technical solutions.

- Embrace Technology:

Stay abreast of emerging technologies and industry trends. Integrate innovative solutions into technical designs and encourage the exploration of new technologies such as artificial intelligence, microservices, APIs, and open standards to foster innovation.

- Clear Communication:

Foster collaboration among stakeholders, including business leaders, project managers, architects, developers, and operations teams. Ensure transparent communication and shared understanding of technical decisions and solutions.

- Continuous Improvement:

Embrace a culture of continuous improvement, innovation, and learning. Encourage experimentation, knowledge sharing, and feedback loops to drive innovation and evolve technical capabilities over time.

- Strategic Alignment:

Ensure that technical decisions and solutions are closely aligned with RECCo's business objectives, strategies, and priorities. Prioritise initiatives that deliver tangible value, support business growth, and enhance user experience.

Implementation Oversight

During the implementation of new solutions, RECCo may request the Digital Service Partner to demonstrate how the proposed solution, platform, or tool aligns with the established design principles. This process ensures that all components meet the required standards and contribute to the overall strategic objectives of RECCo.

By fulfilling these responsibilities, the DA ensures that RECCo's technological landscape is robust, scalable, and aligned with the organisation's long-term strategic goals, driving sustainable business success and operational excellence.

API MANAGEMENT POLICY

1. Overview

This document describes the standards required to be implemented for all APIs for which RECCo is responsible.

In summary, APIs will:

- be secure
- be designed using a micro-service architecture[1]
- follow recognised industry standards and best-practice
- avoid breaking changes
- be documented

Note 1: This specifically refers to new APIs. The API management policy applies to all APIs used to interact with REC services.

2. Standards

This section describes the standards that shall be applied to REC APIs. The standards cover protocols, rate limiting and quota management, API versioning and management.

2.1.API Protocol

In this section a distinction is made between customer facing APIs and internal APIs. Customer facing APIs are those the service users (REC Parties, TPIs and end consumers) interact with directly, whereas Internal APIs are those that operate between the API Gateway and REC services, or between REC services.

2.1.1. Customer Facing APIs

REC service users shall interact with RESTful APIs by default. Deviations from this standard shall require approval from RECCo's Enterprise Architect.

APIs used between an API Gateway and a REC service, or between REC services shall conform to the RESTful standard. Deviations shall be permitted via approval from RECCo's Enterprise Architect, and:

- Any API Gateway that is North of the API can perform a protocol translation to the service's API protocol.
- The protocol satisfies the API Protocol Security constraints described in 2.1.2 below.

2.1.2. API Protocol Security

RECCo shall not permit the use of any API Protocol that:

- Out of the box has no security controls,

- Has not been in commercial use for at least 3 years,
- Is on RECCo's API protocol retirement list.

The only exception to this rule is for existing REC services. However, APIs permitted by the exception will need to be replaced at the service's next procurement.

2.1.3. Blocking or Non-Blocking

A blocking API is one that processes synchronous requests, whereas a non-blocking one processes asynchronous requests. APIs shall be non-blocking when:

- The service that receives the API requests has an availability level higher than 99.99%
- The service is expected to receive via the API a high volume of requests (1000 TPS+) and be capable of responding to them.
- Services processing API requests shall be single threaded and asynchronous when service performance requirements imply the service will be more reliable.

2.2. Rate Limiting

APIs shall be rate limited without exception, furthermore:

- A standard rate limit shall be applied to all users of the API, and:
- API users shall be able to apply for a temporary increase in their rate limit.
- API users shall be able to apply for a permanent increase in their rate limit.
- Service providers shall not allow a rate limit to be set for an individual API user that has a detrimental impact on other users.
- Service providers shall be able to charge for an increase in rate limits.
- An exponential back off policy shall be applied to API users who exceed their rate limit. The API user is responsible for initiating delays in request submissions when a 'too many requests' response is received.
- It is the responsibility of the API user to apply an increased delay in the submission of requests when a 'too many requests' response is received.
- Where an API allows batched requests to be submitted:
- There shall be an upper limit to the number of requests that can be included,
- Where a batch contains *N* requests then, for example:
- The rate limit is 200 requests/second, then,
- When N is 50, and 120 requests have already been submitted, the total number of requests submitted is 170 and the batch is accepted.
- When N is 50, and 190 requests have already been submitted, the total number of requests submitted is 240, the batch is rejected and a `too many requests` (429) response is provided.

- Batch requests are processed at a lower priority than single requests.
- Rate limits shall be provided in the API documentation.
- API batch limits shall be provided in the API documentation.

2.3.Quota Management

API Quota limitations shall:

Be for a fixed period of time, e.g. number per hour

- Be configurable:
- The API user shall be able to request:
- For a temporary increase in quota limits
- For a permanent increase in quota limits
- API users shall be able to monitor their API quota limit and usage through API response headers.
- Individual API user quota limits shall not be set to a level that has a detrimental impact on other API users.

API users may be charged additional fees when their quota limit exceeds the standard rate.

3. API Versioning

API versioning shall be applied when the protocol allows incompatible changes to be made to an API. If a breaking change can be made for an API used by a service, and the protocol does not allow versioning, the protocol shall not be used.

3.1.API Versioning Format

APIs shall be versioned using a *major.minor.patch* and conform to the SMVER (Semantic Versioning protocol). Hence:

- The *major* number shall be incremented when an incompatible change is applied,
- The *minor* number shall be incremented when a backwards compatible change is applied,
- The *patch* number shall be incremented when a backwards compatible bug fix is applied, and:
    - Shall be set to 0 when there is a change in the major or minor number.

3.1.1. Version Retention

A maximum of N-2 versions shall be operational, with version N-3 being supported for a period of 6 months, after which the version is retired. Upon retirement, documentation referring to the retired version shall be amended, and the reference(s) removed or updated.

Examples

The following table shows version updates for a breaking change, non-breaking change and a patch. It shows the versions supported and operational after the amendment.

Version	Action	New Version	Versions Supported	Versions to Retire
2.4.22	Bug fix	2.4.23	2.4.23, 2.3, 2.2	N/A
2.4.23	Compatible Change	2.5.0	2.5.0, 2.4,2.3	2.2
2.5.0	Incompatible Change	3.0.0	2.5.0,2.4.23	2.3

3.1.2. Communication of API Version Retirement

API users shall be notified when a version of the API shall be retired 6 months before its retirement date. Subsequent communications shall be made 4, 2 and 1 month before retirement of the API version.

3.1.3. Which API Version?

- When a request's header:
- Does not include a reference to the API version to use, then the latest API version shall be used to process the request,
- When the request includes a reference to the API version to use, then:
    - The request shall be rejected when the version is not valid,
    - The request shall be rejected when the version has been retired,
    - The request shall be processed when the API version is still in use.

3.1.4. Documentation

Documentation of customer facing APIs shall conform to the OpenAPI standard (preferably >= v3, minimum v2), and:

- Shall include descriptions of the attributes passed to the API
- Shall include API metadata, for example standard rate and quota limits
- The data types of the attributes passed to the API
- The attributes returned via the API and their data type
- The response and error codes returned by the API
- Worked examples

- Provide context around the use of API features.

API documentation shall not contain references or descriptions of functionality, or APIs that are not or no longer supported.

4. API Security

4.1.API Request Validation

A REC service receiving a request via an API shall:

- Validate the data types of the attributes passed to it and reject a request when a data type is not valid or indeterminate.
- Validate the value of the data attributes passed and reject a request when one value is not correct.
- Reject requests where values for optional fields are provided, and:
- The optional field is not relevant to the request,
- Not used for the type of request being made.

4.2.Authentication

4.2.1. Business to Consumer

Authentication of consumers to APIs shall use established protocols like Open ID Connect - requirement to use MFA depends on the nature of data and level of risk e.g. personal data must only be provided over a connection authenticated using MFA.

4.2.2. Business to Business

Business to business authentication shall use mutual Transport Layer Security (mTLS). The version of TLS used shall be v1.2 or above.

4.2.3. Administrative Access

Authentication of administrators to APIs and associated Gateways users shall use established protocols like Open ID Connect and require MFA.

4.2.4. Authorisation

4.2.4.1. Role Based Access Control

RBAC controls shall be used to determine if an API user is allowed to use an API or a particular API request.

4.2.4.2. Administrative Authorisation

Authorisation of administrative users shall be granted using OAuth 2.0's Authorisation Grant with PKCE

4.3. Digital Signature

By default, request payloads to services hosting confidential data shall be digitally signed, and:

- Where a request passes through RECCo's API Gateway, the gateway shall validate the payload's signature.
- The receiving service shall validate the requests digital signature.
- When a digital signature cannot be validated, the associated request shall be rejected

4.4. TLS Configuration

TLS configurations for APIs shall conform to the guidelines described in 'NIST Special Publication 800-52 - Guidelines for the Selection, Configuration, and Use of Transport Layer Security (TLS) Implementations'

APIs shall be configured to use TLS 1.3 – Note, TLS shall automatically use TLS 1.2 where the other service cannot converse using TLS 1.3.

4.5. Storage of an API's Private Key

Private keys shall be stored in a key store satisfying FIPS 140-2 standards.

4.6. Configuration and Secret storage

API configuration files and secrets shall be retained in a secure vault

4.7. Key and Certificate Rolling

Keys and Certificates can be rolled in a key rolling period.

4.8. Key and Certificate Monitoring

Keys and certificates shall be monitored. Alerts shall be provided when expiry is due in 3, 2 and 1 month, 3, 2 and 1 week, then daily and hourly on the final day until the key or certificate is replaced.

4.9. Key and Certificate Revocation

It shall be possible to revoke keys and certificates with immediate effect once it is known or suspected keys or certificates have been compromised.

4.10. Documentation and Retirement

Documentation of retired APIs or API functionality shall be available as soon as the API or functionality is de-commissioned.

5. Change

The management of APIs shall be auditable from the request of change through to the delivery of change. However, delivery of change is not limited to the deployment of new code to a production environment; It also includes amendments to documentation, security configuration (including RBAC controls), alignment of APIs in test, production and sandbox environments, validation in the production environment through feature flags and progressive deployments.

The introduction of new APIs, changes to existing APIs or their retirement shall only be undertaken when a change has been approved. The impact of change shall be de-risked by:

- Versioning of APIs where the protocol allows,
- Migrating customer facing APIs to protocols that allow versioning,
- The introduction of CI/CD deployments,
- Feature flag/toggle management and progressive deployment,
- Requiring API documentation to be up to date and relevant to APIs operating in a production environment.

To support the progressive deployment process, deployments shall be treated as an auditable security event, as will the setting and resetting of feature flags. Deployments shall:

- Make use of feature flag, and follow progressive release process,
- Deployment shall be seamless deployment and incur zero downtime,
- Include a grace period for API users to implement changes when a breaking change is put in place,
- Only be permitted when the change can be linked to an approved change request.

As a general rule, changes shall be backwards compatible. When this is not possible the service provider shall:

- Inform RECCo an incompatible change must be implemented and prove why that is the case
- Communicate with all service users of an impending breaking change

Changes to APIs shall follow the REC change process, and:

- New APIs shall only be introduced as part of:
    - The delivery of a new service,
    - Addition or removal of functionality,
    - A bug fix.

6. Documentation and Technical Guidance

Documentation available to the API user provides them with key information about the use of the API. It needs:

- To be easily accessible
- To clearly describe its use, and:
    - Input values needed and their type, permitted values to provide where applicable,
    - Output values produced and their type,
    - The APIs meta data, including version number, versions supported and standard rate and quota limits,
    - Response types and error codes.
- To clearly indicate mandatory and optional values to pass, making it clear when to use optional fields.
- To indicate when specific versions of the API shall be retired,
- To use domain relevant terms and provide context,
- To provide worked examples, showing:
    - Submission of requests
    - Positive and negative responses

The provision of the information described above shall go some way to easing the problems developers experience with APIs as some of the guessing will be removed. It will also help service users build more resilient, and more secure services. However, they shall still experience issues, so:

- REC service users shall be able to experiment with APIs in a safe sandbox environment.

7. Development Portal

All APIs, whether they are synchronous or asynchronous, shall be available via a developer portal. In the developer portal an application programmer shall be able to build their applications and evaluate the use of an API. The development portal shall:

- Allow registered users develop applications using REC APIs,
- APIs shall be in line with production,
- Provide access to technical documentation on the use of APIs.

8. API Design

API design shall have an impact on use. Design without user input can result in APIs requiring convoluted processes. Good design shall encourage use of REC services, while bad design shall deter it, so it is an important component of the evolution and development of APIs. API design shall:

- Require an understanding of the ‘customer journey’,
- Require liaison with relevant service users to determine the functionality that needs to be provided,
- Shall be designed to incur a minimum number of API calls to complete the customer journey,
- There shall be parity in API definitions,
- Shall use domain relevant terminology shall be used,
- API users shall be able to trial APIs and recommend changes.

API RELIABILITY POLICY

1. Introduction

API Reliability has many contributing factors, ranging from ease of use and accuracy of supporting documentation, through to replication and redundancy of API infrastructure and routing policy. There are several API resilience patterns, including rate limiting, and shedding, time out, health checks and circuit breakers to name a few. While the implementation of design patterns shall contribute to improved reliability other practices also play an important role in achieving it. These include contract testing, documentation testing and ease of use evaluation. Undertaking these activities shall result in a reduction of errors, thus improving an API's reliability.

The target audiences for this document are project managers involved in the implementation of REC services using APIs and service providers developing and maintaining them. It can also be used as a reference point of good practice should a service's API(s) persistently prove itself unreliable in some way. The document shall describe RECCo's API Reliability Policy that service providers are expected to follow in delivering reliable and resilient APIs. The policy will cover items already commented on in RECCo's API Management policy and are worth repeating in this document. Building reliability through replication and redundancy of infrastructure, for example, are a subset of design patterns and processes that contribute to resilience. Reliability is also a product of design, testing, ease of use evaluation and support.

2. What APIs does the policy apply to?

The policy applies to any API needed to interact with a REC service regardless of the external API end point being housed by an API Gateway, or by direct connection with a service.

It refers to practices known to improve resilience of APIs. While they may not be considered enforceable, avoidance of them will lead to significant issues. Where service providers need to satisfy ISO27001 standards many of the practices referred to will need to be observed to ensure compliance with it. For example, Fuzz testing requires the use of valid and invalid inputs to be used to test APIs and serves to increase security.

When a service provider is not ISO27001 accredited, RECCo shall expect them to be able to demonstrate the practices referred to in this document are observed as part of their existing development process.

RECCo accept all the practices described in this document may not be relevant to all their service's APIs. They will accept their omission in the API development and change process for specific APIs based on requirements and perceived security risks.

3. Documentation and Feedback Loop

For the API user the documentation describing the API, the calls that can be made, the information returned, sent and received is of paramount importance. Its accuracy will contribute towards:

- Successful integration with an API,
- A reduced error rate of service users during development, testing, and in production,
- The success of the API and underlying service.

RECCo's API Management Policy section describes the format API documents should adhere to, their accessibility and maintenance. It is important they accurately and clearly describe how the API is used as an

API's document defines the 'contract'. Consequently, validation of an APIs documentation is often referred to as contract testing.

3.1.Contract Testing

An API contract is documentation that describes an API's intended functionality and constraints. It must clearly state what each request and response should look like, applicable rate limits and other constraints and forms the basis of service-level agreements (SLAs) between producers and consumers. API contract testing helps ensure that new releases don't violate the contract by checking the content and format of requests and responses.

The quality of an API's documentation contributes towards its reliability. Its presentation, wording and relevance of the information provided determines the accuracy and precision with which developers use it. For this reason, APIs used to interact with REC services must be 'contract tested', by users not involved in the design and development of the API. Furthermore, the users doing the contract testing should have no prior knowledge of the API and new or existing functionality as this would skew the results of the testing.

The service provider must be able to produce evidence an API, its additions and changes has been contract tested.

3.2.Feedback Loop

REC API service users can be from organisations both large and small. In a large organisation developers may be able to refer to colleagues for advice, whereas those in smaller organisations will not. With any API there will be use cases and functionality that is more difficult to communicate effectively to everyone. Consequently, service providers need to have a 'feedback' loop for their APIs, allowing users to provide feedback on its documentation or request guidance on using API features.

4. API Testing

APIs need to be tested to make sure they work as intended. Working as intended covers performing the requested task correctly while under normal load, high load and during service failures. Contract testing has already been described in section 3. Other types of testing a service provider must undertake includes:

- Usability testing,
- Performance testing,
- Automated testing,
- Fault testing, and
- Fuzz testing.

One or more omissions of these test phases shall result in a reduction of an API's reliability in some way. The remainder of this section comments on the type of testing that should be undertaken to establish the reliability of an API.

4.1. Usability Testing

Usability testing establishes how easy it is to learn how to use an API accurately, which in turn affects its popularity and contribution to developer productivity. It is different to contract testing. Contract testing confirms the request and response works as stated. Usability concerns itself with the speed of learning, how quickly the user can make use of an API, how easily the user can navigate to the documentation and how the documentation's language assists with the learning.

RECCo require the usability of an API to be assessed at initial implementation and following the addition of new functionality. Evidence of usability testing will need to be retained.

4.2. Performance Testing

REC services will be accessed by users whose business could place additional load on those services accessed by the APIs they use. As with all businesses there will be calendar events, where requests from one or more users can stress the underlying service; There will be media announcements that will also place additional unexpected load on a service in different ways. When excessive load is placed on an API's service:

- For short periods it can be managed by spreading the requests across replicated instances of the service,
- For longer periods the service may have to elastically scale, making it necessary to test service behaviour when this happens,
- Prolonged periods of high throughput can result in additional stress being placed on a system and eventually lead to failure. It is, therefore, necessary to test a service under prolonged load, and
- A burst of activity could break the service. It is therefore necessary to test a service to breaking point to establish first points of failure and understand what the path to failure looks like.

RECCo require APIs and their associated service's performance to be evaluated. Performance tests must be able to demonstrate the service can satisfy performance requirements for a low and high volume of requests. Performance must be evaluated from the point of ingress, through to the point of egress. The environment used to performance test must be identical to the production environment in all aspects. All performance tests need to show performance requirements can be met by the API's service.

4.2.1. Scaling tests

Performance needs to be tested as the load increases and decreases. Scaling tests should show the service is able to process requests for NFRs covering scaling requirements. Scaling tests will need to show performance remains unchanged for small, medium and large loads. Elastic scaling will need to be tested for:

- New services whose anticipated peaks in demand point to elastic scalability being needed, or
- Existing services, where typical throughput is known, and elastic scaling is needed to keep the service afloat.

4.2.2. Spike tests

Elastic scaling is an activity that takes place after a prolonged spike in API requests. Not all peaks in demand will last a long time and will take place in short bursts. Consequently, APIs need to be subjected to spike tests.

4.2.3. Soak tests

Performance tests also need to show a service can process a high volume of API requests for a long period of time. These tests should show a service continues to meet performance requirements for the duration of the soak test.

4.2.4. Stress tests

A stress test is used to determine the load or types of loads that can break the service and what that failure looks like. The point of failure for a stress test should not be close to known peak volumes for existing services, SLAs or for performance requirements for the API and its service.

4.2.5. Fault Testing

The most common issues experienced by IT services are network related, particularly in multi-tenanted cloud environments. Given their prevalence, it is diligent to test services ability to cope with network related issues and maintain service. This can be achieved by injecting network faults into a service's network.

Fault testing must be undertaken safely in a test environment.

4.2.6. Fuzz Testing

An APIs service, if it is securely coded, will validate all information passed to it and will reject the following:

- Any request submitted using an invalid set of options,
- Any request submitted whose payload includes invalid data types and data values passed to it.

While that should happen it often does not, which is why Fuzz testing is needed. For an API, Fuzz testing is the submission of API requests, some containing valid requests, others not. Where requests contain invalid data in the payload it should be rejected.

Service providers are required to demonstrate their APIs are subject to Fuzz testing. Fuzz testing should be included in automated test suites.

4.3.Automated Testing

Automated testing of APIs and their services within the development process provides the following benefits:

- Regression testing is a much faster process allowing more to be validated, resulting a de-risking of the test process. There is more confidence a change will work, and therefore a lower risk of reliability problems when the change is deployed,

- Tests shall be validated in the same way every time the automated tests are performed. Tests are not affected by human error, and again this leads to improved reliability of any change, provided the code coverage is high.
- Testers can focus their tests on the changes and do not have to compromise their testing.
- It is common practice now to incorporate automated testing in the development process. A service provider should be able to provide evidence their APIs and services are subjected to a range of automated tests and show the code coverage of the tests is 95% or higher. RECCo expect automated testing to form an integral part of changes delivered via CI/CD pipelines.

5. Responses and Error Responses

Responses for success and failure need to be consistent. Consistency in response format and status is key to client-side processes being able to manage responses appropriately, resulting in better reliability of the API and the client applications. Their consistency is essential to the production of reliable and user-friendly APIs.

RECCo standards require customer facing APIs to be RESTful. Including standard HTTP status codes in responses, additional error codes and meaningful error messages coupled with consistent error handling contributes to the creation of a reliable API and makes it easier for client-side applications to manage error responses.

5.1. HTTP Responses

Responses to requests need to be given with the correct HTTP response for the outcome. 200 series need to be given to indicate success, 300 series for redirection, 400 for client-side errors and 500 responses for server-side errors, conforming to the semantics described in RFC 9110.

5.2. Response Pagination

Payloads included in responses given to API requests can vary in size. Increasing demands placed on APIs and their use in accessing information in reporting systems/data lake is resulting in large payloads being returned. The large payloads returned can put pressure on the API service's network and result in a significant performance degradation, even failure.

Ideally, pagination should be avoided. However, the requirement may make pagination unavoidable, so when responses need to be paginated, it should be managed using next and previous links, rather than a '?page = attribute' as the approach is a more resilient one.

5.3. Error responses

The API's code needs to capture validation, application exceptions and unhandled exceptions. The error responses should adhere to the format referred to in RFC7807, 'The Problem Details Response Format'. Error messages must be meaningful, and the response should provide an indication of action to take. For example, a 429 response could be given along with a message 'Too many requests.'. The response gives a code and a

reason but does not suggest what should be done. Typically, when a response is received the user should back off, and exponential back off should be applied. Hence, the response should also advise the user how long to back off for.

Those developing APIs need to determine where in the service particular responses are actioned. For example, where a service traverses an API Gateway a 429 response is returned to the user by the gateway, not the service downstream of it, with details about backing off included in the response's headers.

Error responses must also be secure, hence:

- No confidential data shall be included in an error response, and
- No information about the service shall be provided in an error response.
- To summarise, The REC require error responses to:
    - Provide a meaningful response code and message to be provided,
    - When a hint to how the recipient should respond to the error is provided guidance it is in the response header, and
    - Personally Identifiable Information (PII) is not included in the response, and
    - Responses exclude information alluding to any component of the underlying service(s).

5.4. Documentation

Error handling and reporting mechanisms of the API need to be documented. Error handling, the error codes, where errors can be expected and additional steps a developer should take on receipt of them need to be documented. This shall result in developers adopting a much more suitable approach in their application's response to errors, thus leading to better reliability.

6. Rate Limiting and Load Shedding

Rate limiting and load shedding achieve the same objective, in that they both result in requests not being processed to keep a service live. The difference between the two is:

- Rate limiting is applied to a user whose frequency of requests over a time interval exceeds a threshold,
- Load shedding is applied when the system is operating under stress and additional requests will likely result in failure.

6.1. Rate Limits

Rate limiting has long been established as an important mechanism to help prevent DoS (Denial of Service) attacks. They are also known to enhance resilience, by:

- Helping to prevent a service from being overwhelmed,
- Preventing individual users consuming all resources,
- Prevent un-predictable high-volume events causing a service failure.

All APIs for REC services shall be rate limited, and:

- Where API requests allow batched requests, requests in the batch shall be processed within rate limits. For example, where an API has a rate limit of 50 requests/second and a batch contains 51 requests, the first 50 can be submitted in one second and the remaining one in the next,
- Shall have as a minimum a rate limit expressed as a rate per second, and the rate limit set shall be informed by:
    1. The results of performance tests,
    2. Known service performance and resource consumption,
    3. Non-functional performance requirements
- APIs shall have standard rate limit that is applied to its users as a default. API users shall be able to have bespoke rate limits applied but this will incur additional costs to them. An API's standard rate limit shall be included in API documentation. Its back off policy shall also be included in API documentation so users can incorporate processes that allow them to back off gracefully,
- Where an API has different limits applied to different time granularities (chronons), then:
    1. The rate limits shall be smaller for small time granularities. i.e. for rate limits R1 and R2 applied to chronons C1 and C2, if
        a) C1< C2 then R1<R2,
        b) C1=C2 then R1=R2.
    2. The rate limits for the different time granularities shall be documented in the API's documentation together with the backing off policy for each.
- Rate limits applied are not set in stone. Generally, rate limits applied to new services should be conservative in their settings. Once demand is known and service capabilities proven, limits can be increased.

6.2.Load Shedding

Policies for load shedding shall be informed by performance and knowledge of the services business domain. Some API requests will be critical to services while the delay in processing other API requests may have limited or no consequences.

To establish the load shedding policy to apply a service should be load tested to breaking point. The information gathered from the load testing should be used to determine an appropriate load shedding policy that can be applied. The load shedding policy should not impact elastic scaling and should be tested to determine its effectiveness.

Load shedding policy needs to be documented in API documentation so service users are aware it can happen and the form it will take. This will help service users to prepare and implement appropriate activities when such events arise.

When requests are rejected due to load shedding a response should be provided to the service user that informs them the request has been rejected and, as with rate limiting responses, the action they should take. It is the responsibility of an API's underlying service to provide a load shedding response and not a proxy service like an API gateway.

7. Health Checks & Circuit Breaker

Health checks are undertaken to establish the health of a service's components and is achieved by monitoring. Health checks help inform routing of requests.

A circuit break is applied when it is established a service's component is failing, which is determined using health checks. It prevents traffic being sent to the failing components until they have recovered. As a result, it stops requests being sent to a service instance that will end in failure for the request.

Health checks must be tuneable, as should invocation of a circuit breaker

8. Routing Policy

Routing policy can also contribute to the resilience of an API. To distribute requests evenly across service instances a round robin policy could be applied. With such a policy, requests will continually be sent to service instances regardless of their health. This can result in failures to process requests, an increase in requests due to user retries which would add to the problem. For this reason, routing policies should be adopted that provide some guarantees requests will be distributed across service instances that are healthy and performant.

9. Time outs and Idempotence

All IT services shall experience performance issues with the service's network at some point. These can arise from an excessive load placed on the network or some technical issue with one or more network components. The performance issues can lead to requests taking an excessively long time to process or may even appear to not get processed.

9.1.Time Outs

To prevent a backlog of requests building up eventually leading to network resources being consumed, long running API requests should be timed out, i.e. the service has given up waiting for a response, releasing the resources for another request or a retry. However, when a request times out, whether that is invoked by the service or a default of the transport policy, the outcome of the request is not known, it could have succeeded. The calling application may resend the request, which in some circumstances could lead to a need for idempotent processing.

9.2.Idempotence

When an API request times out it is not uncommon for the user to resend the request. When the request does not result in any data amendments the resend has no impact on the underlying data and, therefore, no impact on future requests. However, when an API request is made that will change data, resending the request after a time out risks setting the service's data to incorrect values, which will have knock on effects and require some operational effort to correct. To prevent this from happening, API requests should be put through an idempotent check, i.e. Have we already processed this? Upon performing an idempotent check on an API request:

- The API request shall be processed when the request has not been seen before,
- The API request shall be declined when the request has been seen before and shall include the status of the request (In progress, Completed or Error).

Idempotent checks prevent duplicate updates of data. This has a positive operational impact as corrections to a service's data do not need to be made.

9.3.Time Out Summary

A service's network shall go through peaks and troughs in performance. Services shall also get sudden spikes in use, which can be short term or long lasting. These will result in requests taking some time to complete and may even result in a response to a request not being received. This can and does lead to service failure, so it is considered good practice to manage request timeouts. However, timing out a request does not mean it failed to process; It only means the service waited too long to get a response, so gave up waiting. As stated, timeouts can result in service users duplicating their request, and when the request results in data changes it is advisable to prevent this by having an idempotent policy for data updates.