GovStack

Building Block Definition
Information Mediator

Developed by: Aleksander Reitsakas, Taylor Downs, and Dr. Ramkumar Permachanahalli in cooperation with GIZ, ITU, DIAL, and the Government of Estonia
Reviewed by: Fergal Marrinan, Aare Lapõnin, Neil Roy, Surendrasingh Sucharia

GovStack

Table of Contents

GovStack

1 Version History

Version	Author	Comment
1.0.0	Aleksander Reitsakas, Taylor Downs, Ramkumar P.S.	Initial revision
1.0.1	Aleksander Reitsakas	External links updated
1.1.0	Aleksander Reitsakas, Taylor Downs, Ramkumar P.S. Reviewers: Neil Roy, Aare Lapõnin, Fergal Marrinan, Surendrasingh Sucharia.	Applied feedback from technical review, addressed formatting issues

2 Description

The functional requirements section lists the technical capabilities that this building block should have. These requirements should be sufficient to deliver all functionality that is listed in the Key Digital Functionalities section.

The information mediator (IM) building block (BB) provides a gateway for exchange of data and services among GovStack Building blocks through open-API rest based interfaces to ensure interoperability and implementation of standards. The IM provides mechanisms for applications/BBs to publish and consume services and event notifications among other GovStack BBs.

IM services act as a channel through which BBs and external applications can connect to services exposed by other BBs such as registry services, identity services and payment services. The IM BB provides a second service, as a broadcasting channel for notification of events among the connected applications in a publisher-subscriber (PubSub) model. The IM BB also maintains a log of transactions (e.g., requests, events), as well as handling communication errors between BBs and/or other applications via the PubSub service. This component may employ other core components, such as registries, repositories, etc. By allowing different applications to exchange information, it can act as a mechanism to encourage or enforce best practices, data standards around PubSub, and data-sharing policies in cross-facility work-flows among business processes.

2.1 Use Case Applicability

The Information Mediator, through the “Service Access Layer” and the “Pubsub Layer” enables *all* use cases that make use of HTTP requests and the OpenAPI specification.

For this exercise, we have tested against the use cases within the “Postpartum and Infant Care” and “Unconditional Social Cash Transfer” user journeys.

2.2 What Is *Out* of Scope

The IM BB does NOT handle communication between human users and applications.

The IM BB does NOT handle ingress and egress access from external applications to and from GovStack’s internal components. (This may be handled by a public API Gateway in the domain of the Security BB.)

2.3 Future Scope

2.3.1 Consider broader existing standards. See section on Gov.UK’s API Management Strategy Document.

3 Terminology

3.1 Building block

Building blocks (BBs) are software modules that can be deployed and combined in a standardized manner. Each building block is capable of working independently, but they can be combined to do much more.

Building blocks are composable, interoperable software modules that can be used across a variety of use cases. They are standards-based, preferably open-source, and designed for scale.

Each building block exposes a set of services in the form of REST APIs that can be consumed by other building blocks or applications.

Ingress access is access from external applications to GovStack building blocks and applications.

Egress access is access from within GovStack BBs and applications to external applications

3.2 Use case

A **use case** is a piece of functionality described as a sequence of actions (steps) to achieve a specific goal in a specific context of usage. E.g., in one use case the IM BB may be used to let a BB access a service provided by another BB; in another use case it may be used to relay an event notification from one BB to several other BBs via PubSub.

Each use case may involve a collection of modules, or BBs. A relatively small set of these BBs can be readily applied to a wide variety of use cases in low-resource settings.

3.3 API[1]

An application programming interface (**API**) is a connection between computers or between computer programs. It is a type of software interface, offering a service to other pieces of

[1] https://en.wikipedia.org/wiki/API

software. A document or standard that describes how to build such a connection or interface is called an API specification. A computer system that meets this standard is said to implement or expose an API. The term API may refer either to the specification or to the implementation.

3.4 Webhook[2]

A **webhook** (also called a web callback or HTTP push API) is a way for an application to provide other applications with real-time information. A webhook delivers data to other applications as it happens, meaning you get data immediately.

In a sense, a "webhook" is just a way of making an HTTP post to a URL when *something important* happens.

For example, consider "PatientTracker v2", a fictional application that registers new patients. The PatientTracker v2 admin will create a *webhook* that pushes data from their application to 7 different applications that depend on this event to do something, each having some external URL whenever a new patient is created inside my "PatientTracker v2" application.

- So, without **PubSub**, to notify multiple different services) about a "new patient", one would need to create multiple different webhooks (with corresponding URLs) in the Patient Tracker v2 application. Also, this implies that Patient Tracker V2 knows, apriori, which other applications have to be informed based on an event, which creates a nightmare of change management as needs evolve. Also, each application such as the "Patient Tracker V2" will have to add and maintain its own set of webhooks for each event it generates. This mapping of "who to inform if this event happens" may change with time.
- With a **PubSub** layer, it needs only 1 webhook which sends data (when a new patient is created in the Patient Tracker v2 application) to the IM BB url for handling PubSub events (e.g., IM.com/pubsub/inbox) and the PubSub layer of the BB will then handle sending this information to all 7 applications if they are active subscribers for this event.

Webhooks can also be used *without* PubSub to facilitate an asynchronous style of communication. The consumer can request information from the provider, but expect a 202/Accepted status code and then another (delayed) POST request to a specific URL via a webhook configured on the provider's side. However this is sub-optimal and useful only in situations where one can tolerate inordinate delays in a response to a POST await.

3.5 Service Access

3.5.1 Member

A **member** is an organization which is authorized to communicate via the IM for a particular GovStack implementation.

3.5.2 Application

An **application** is a running instance containing one or more BB instances and zero or more use case implementations. An application uses an IM building block to communicate with other BBs or applications. An application typically has a single responsibility.

[2] https://sendgrid.com/blog/whats-webhook

3.5.3 Service

A **service** is a minimal piece of functionality provided by a building block or use case implementation. A service can be local (inside one application) or remote (between different applications). Remote services are consumed using REST protocol and they are described by OpenAPI specification. For example, the "registration" service might be accessed at a particular URL and allow a requester to "register a patient" by sending a POST request with proper patient data.

3.5.4 Security Server

A security server is the main piece of software that is responsible for implementing the "service access" layer of the Information Mediator. This software acts as a gateway, and is responsible for mediating requests between various members, applications, and services.

3.6 PubSub[3]

In software architecture, publish–subscribe (**PubSub**) is a messaging pattern where senders of messages, called publishers, do not program the messages to be sent directly to specific receivers, called subscribers, but instead categorize published messages into classes without knowledge of which subscribers, if any, there may be. Similarly, subscribers express interest in one or more classes and only receive messages that are of interest, without knowledge of which publishers, if any, there are.

PubSub is a sibling of the message queue paradigm, and is typically one part of a larger message-oriented middleware system. Most messaging systems support both the PubSub and message queue models in their API; e.g. Java Message Service (JMS), which is especially useful to handle multiple retries in case a recipient application is offline due to connectivity issues, in a typical low resource region.

This pattern provides greater network scalability and a more dynamic network topology, with a resulting decreased flexibility to modify the publisher and the structure of the published data.

3.6.1 Publisher

A **Publisher** produces events and sends them to rooms. Each event has an event type associated with it. Publishers can produce events of different types.

3.6.2 Room

A **Room** is an PubSub entity that handles the distribution of events. Each Room has a set of connected event types. (E.g., the "birth" Room might contain three event types: "new_birth", "birth_complication", and "infant_death".)

3.6.3 Subscriber

A **Subscriber** can process events of a certain event type. Subscribers are independent of each other and their business logic is different (as rule). Each subscriber processes events from their own perspective.

[3] https://en.wikipedia.org/wiki/Publish%E2%80%93subscribe_pattern

4 Key Digital Functionalities

At a high level, these are the key functionalities that must or should be provided by any software implementation that plays the role of an Information Mediator. These key digital functionalities have been cross-checked and validated with those provided by the OpenHIE community (see below for details) and the "OpenHIE" column in the yellow table provides a cross-reference to the corresponding OpenHIE specification.

4.1 Summary

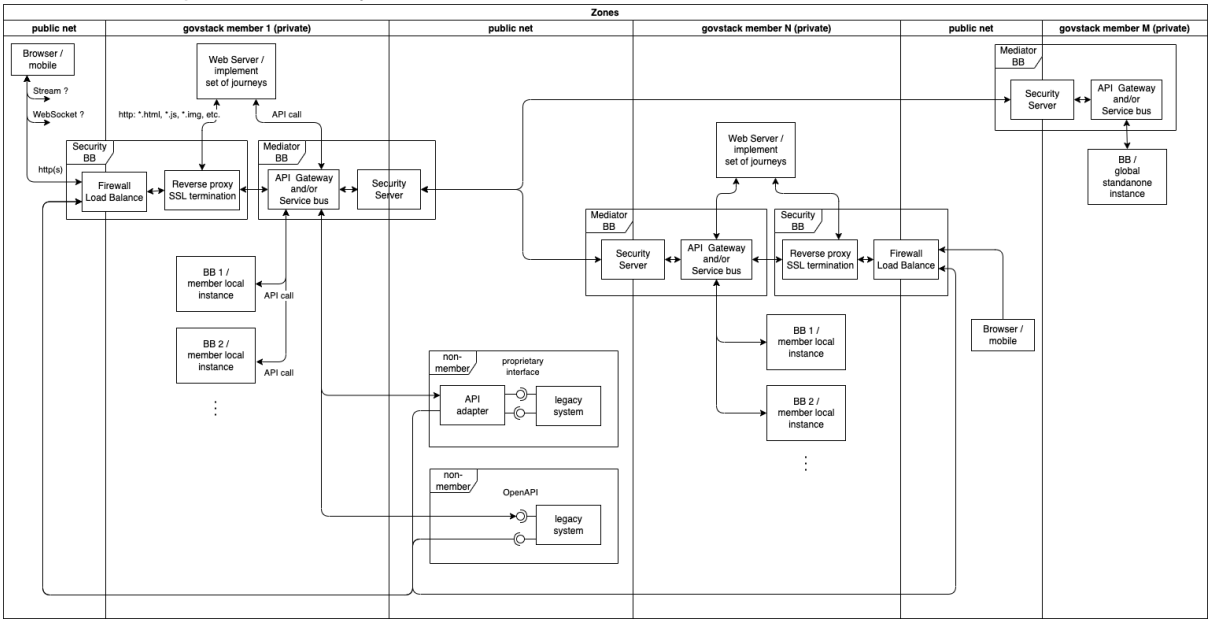
In a nutshell, the Information Mediator building block is responsible for providing (1) a managed facility through which different GovStack BBs and applications may communicate securely with each other and (2) a facility through which applications may publish and subscribe to different events identified by unique message types, enabling more efficient and resilient communication and application design.

Note that the IM is not responsible for manipulating the payloads sent to and from various applications—in a sense, it is both the postal service and the roads/bridges/train tracks—but it does not read the contents of your mail.

4.2 Overall design

GovStack facilitates creation of an ecosystem of applications meant to fulfill different e-governance tasks. Applications are software solutions consisting of some specific part which is actually implementation of some user journeys and a set of standard Building Blocks implementing predefined standardized parts of the job. Different parts of the application interact with each other preferably by means of REST based API. This interaction is built around the API Gateway of the application.

Interaction between applications which are participants of the GovStack ecosystem is implemented by Mediator BB Security Servers. Interaction with software solutions outside of GovStack goes in inward direction through firewall/load balancer/reverse proxy and in outward direction through API Gateway.



Draw.io source in github:
https://github.com/GovStackWorkingGroup/BuildingBlockAPI/blob/main/IM/diagrams/global-picture.drawio.png

4.3 Specific Functionalities

Please note that the source/master-copy of these tables are maintained on this Google sheet and may be updated by clicking the "refresh" button below.

4.3.1 Key Digital Functionalities Table (KDFs)

GovStack provided an initial set of key digital functionalities as recommendations for the IM building block. The following table provides updated KDFs, striking-out and refining those that are no longer required and/or in scope.

ID	Explanation	Status	Bucket	Open HIE?	Comments
KDF1	Routes requests to the correct provider after necessary message transformation functionalities and protocol conversion	REQUIRED	Service Access	IOLF-2	The IM does not do message transformation or protocol conversion.
KDF2	Connects the service requestor to the service provider and its underlying solution platforms, realizing the requested service	REQUIRED	Service Access	IOLF-2	Maybe IOLF-6, but not orchestration
KDF3	Discovers services and, at runtime, to support the virtualization of services, so that changes to end-points can occur without impact to service consumers and service providers	REQUIRED	Service Access	---	
KDF4	Supports the handling of transactions and associated communication errors and exceptions	n/a	Workflow	IOLF-10	Note that this has been shifted to the Workflow BB.
KDF5	Enforces access privileges and other security policies	REQUIRED	Service Access	IOLF-1/15	
KDF6	Maintains service invocation history and monitors and tracks service invocations	REQUIRED	Service Access	IOLF-3	
KDF7	Provides broadcast/multicast capabilities to facilitate faster, more resilient application design	RECOMMENDED	PubSub		Note that "caching" has been REMOVED from this requirement.
KDF8	Translates data from one format to another, and interoperates with handshake protocols to enable interoperability between different ICT Building Blocks duplications	REQUIRED	Workflow	IOLF-5	Note that this has been shifted to the Workflow BB.

4.3.2 “Out of Scope” Functional Requirements

Initially, the following table of high-level functional requirements was provided to the Information Mediation working group, but all have since been identified as “out of scope” because they fall into the realm of the Workflow BB or the Security BB.

ID	Explanation	Status	Bucket	OpenHIE?	Comments
F1	Map data structures and fields from the identification system to the registration system and vice versa.	NOT REQUIRED	Workflow	IOLF-5	Moved to workflow.
F2	Hold authentication and credentials for each system	NOT REQUIRED	Workflow	IOLF-1/13/14	Moved to workflow.
F3	Allow the definition of steps for a particular transaction	NOT REQUIRED	Workflow	IOLF-6	Moved to workflow.
F4	Provide an API for both systems to access – and execute all necessary steps for that transaction (including error handling, retries, and notifications)	NOT REQUIRED	Workflow	IOLF-4/5/9/10/11	Moved to workflow. This is a number of features lumped together.
F5	Provide an API for external systems to access Gov-stack BBs	NOT REQUIRED	API Gateway+ IAM	IOLF-4/5/9/10/11	Moved to Security BB.

4.3.3 Existing Standards

4.3.3.1 OpenHIE Comparison

For comparison with existing specifications, we have engaged with OpenHIE — a global community focused on health systems interoperability. While the remit of the Information Mediator is domain independent, we have drawn from OpenHIE because (a) a number of the initial use cases are health-related and (b) the OpenHIE community has developed a mature, well-thought-through, multi-stakeholder specification which can serve as an example (though limited) for GovStack’s purposes.

Below are the key Information mediation requirements from their OHIE architecture specification. Note that the “GovStack?” column provides a cross-reference *back* to the above GovStack specifications.

ID	Explanation	Status	GovStack?	Comments
IOLF-1	The system should provide a central point of access for the services of the HIE. For example, this interface will provide access to the CR, PR, FR and SHR. This central point of access simplifies security management and provides a single entry point into the HIE.	Recommended	KDF2	
IOLF-2	The system shall provide routing functions that allow messages to be routed to the correct service provider systems within the HIE.	Requirement	KDF1	
IOLF-3	The system shall provide a central logging mechanism for the messages sent through the exchange. This function should log copies of the messages that travel through the interoperability layer for audit and reporting purposes.	Requirement	KDF8	
IOLF-4	The system should allow for the rerunning of failed transactions at a central level, alleviating the need for Point-of-Service systems to resend data, for example, in the event of a problem with an infrastructure component.	Recommended	F4	

IOLF-5	Should support transformation of messages that travel from the interoperability layer to service provider systems and vice versa if the service provider is not able to communicate in the required format, i.e. provides implementation specific adapters to transform messages from the interoperability layer's internal form to a form that the service provider expects (e.g., SHR, CR, PR).	Recommended	F1/F4	
IOLF-6	The system should allow for the routing of messages to the appropriate architecture component or external Point-of-Service system. • Performs orchestration tasks for complex transactions to take the burden off client systems. This orchestration may contact multiple service providers within the HIE on a client's behalf and compile an appropriate response to return to the client. • Examples of orchestration could be the execution of a care plan or the validation of elements (such as identifiers or codes) in a message against other service providers within the HIE (e.g., PR, CR, FR, TS). • Orchestration tasks are those that are required to complete the current transaction and therefore must be executed timeously as the transaction cannot continue without these steps.	Recommended	KDF2	
IOLF-7	The systems shall include an interface into which a workflow engine can be connected. • This workflow engine should be able to keep track of the long running state of a patient's care and would be able to perform actions based on this context (such as sending alerts) to improve patient care. • This workflow engine is out of scope for an Interoperability Layer. However, the Interoperability Layer is expected to expose an interface to allow this sort of systems to be implemented.	Requirement	n/a	Note that this has been shifted to the Workflow BB.
IOLF-8	The system should support the ability to be extended by allowing additional mediation functions to be added or removed as they are needed.	Recommendation		
IOLF-9	The system shall support a mechanism for error management and tracking, e.g. a console for viewing failed transactions.	Requirement	F4	
IOLF-10	The system shall allow for failed transactions to be grouped by error type and reason so that errors can be rectified efficiently by finding the root cause of the error, fixing the problem, and re-running those transactions.	Requirement	F4	
IOLF-11	The system should support the ability for a user to re-run errored transactions through the HIE once the reason for their failure has been rectified.	Recommendation	F4	
IOLF-12	The system shall provide authorized users with a view of metrics for monitoring the flow of messages through the HIE.	Requirement		
IOLF-13	The system shall manage the security of the HIE through authentication (identity verification), authorization (permission to interact with specified HIE components) and encryption and decryption of messages.	Requirement	F2/KDF5	
IOLF-14	The system shall support Authentication and Authorization of systems trying to send data to the HIE.	Requirement	F2/KDF5	
IOLF-15	The system should support the encryption of data in flight (when not on a physically secure network) and at rest (whenever data is stored, e.g. when transactions are stored for logging).	Recommendation		
IOLF-16	The system should capture monitoring statistics, such as transaction loads and performance metrics, and provides a view of these for monitoring the flow of messages through the HIE.	Recommendation		

4.3.3.2 Gov.UK's API Management Strategy Document

In future iterations of this specification, we may take into consideration more broad API-management standards which include multiple domains, such as those proposed by the UK Government in their https://www.gov.uk/guidance/defining-an-api-management-strategy document.

5 Cross-Cutting Requirements

The cross-cutting requirements described in this section are an extension of the cross-cutting requirements defined in the Architecture Blueprint document and the Security BB definition. This section will describe any additional cross-cutting requirements that apply to this building block or differences with the Architecture Blueprint.

5.1 Architecture Blueprint

Topic Number in Architecture Document	Name of Requirement in Architect Document	Remarks
5.5	MUST only use TIOBE top 25 languages	SHOULD use TIOBE top 25 languages (Not MUST) Shell may be used for scripting. (Shell holds language ranks between 50 and 100.)
5.8	APIs MUST be idempotent	Application APIs will contain POST endpoints which are not idempotent. GET/PUT/DELETE APIs are idempotent relative to Mediator BB, but idempotent-ness of intermediated services depend on the service provider and cannot be specified at this level.
5.17	MUST provide configuration only through the environment	Configuration not only through environment Configuration is mainly in configuration files. Environment variables could also be used. No configuration in code.
5.21	Databases MUST not include business logic	Databases SHOULD not include business logic We propose that this is a design recommendation which intends to make business logic all live in a clearly visible and accessible location. This may not always be followed, as the benefits of stored procedures (e.g., in reducing DB transaction round-trips, etc.) may sometimes outweigh this general design recommendation.
5.26	MUST be asynchronous first	SHOULD be asynchronous first The Service Access Layer of IM is synchronous first. The PubSub Layer is asynchronous

5.32	MUST use standardized configuration	See 5.17
5.34	MUST use standardized data formats for interchange	Standard formats are used for communication with other BB. Inside BB non-standard protocols can be used.
5.38	MUST use web hooks for callbacks	OpenAPI spec 3.0 supported in the first version. Moving to OpenAPI 3.1 is planned in the future.

5.2 Security Specification

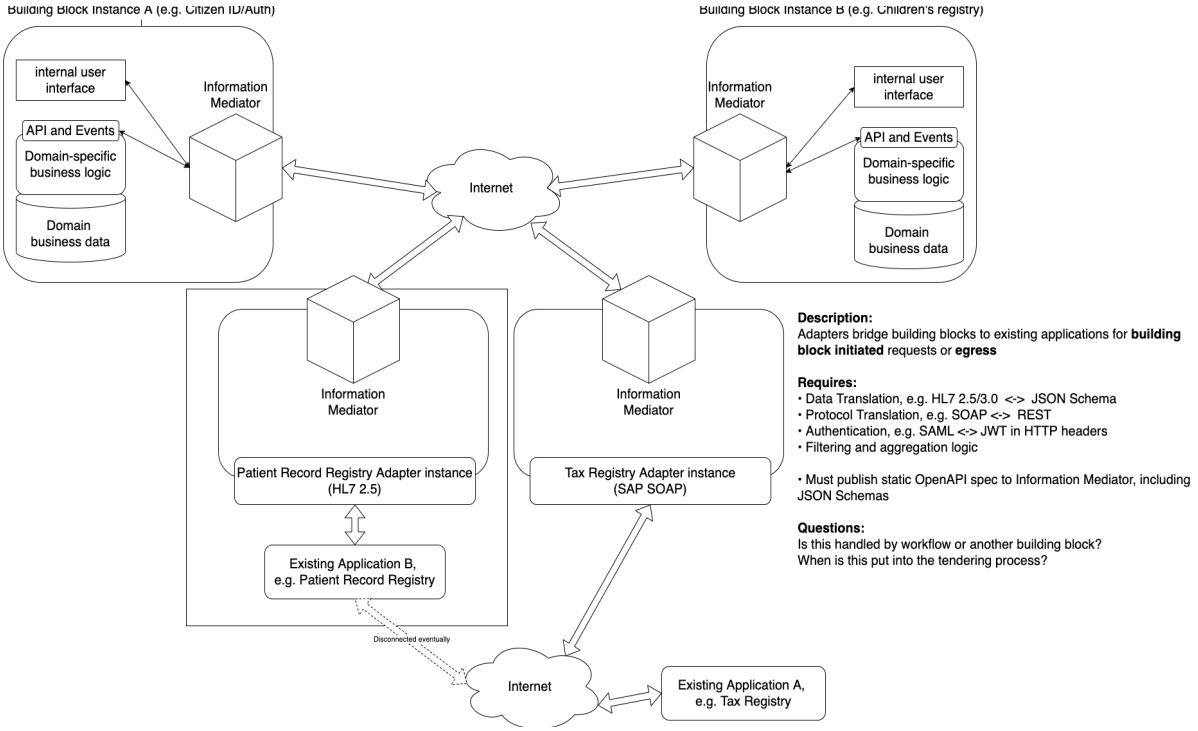
Topic Number in Security Document	Name of Requirement in Security Document	Remarks
	MUST allow the provisioning/management of admin users via the SSO/IAM application specified by the Security BB.	

6 Functional Requirements

The functional requirements section lists the technical capabilities that this building block should have. These requirements should be sufficient to deliver all functionality that is listed in the Key Digital Functionalities section.

These functional requirements do not define specific APIs - they provide a list of information about functionality that must be implemented within the building block.

6.1 Service Access Layer



Editable version (github repo / image - link)

6.1.1 Administrative Interface

1. There must be an administrative portal that allows an IM administrator (with appropriate authentication) to register/deregister members, applications, and services.
    a. OpenIAM (or other IAM solution) must be able to create/edit/delete admin users for the IM interface.
        i. Security BB Spec 1.0.1 section 6-1
2. There must be a security-server level interface that allows an administrator for a single security server to manage members, applications, and services that live under a single security server.

6.1.2 Administrative User Roles

1. There must be different types of administrative roles which provide different levels of access:
    a. System Administrator — is responsible for the installation, configuration, and maintenance of BB
    b. Service Administrator — determine which services are available and what are access rights
    c. Security officer — must be able to manage key settings, keys, and certificates
    d. Registration Officer – registration/removal of members/applications
    e. PubSub manager — must be able to manually add/remove PubSub subscriptions and manage PubSub event "types" (with associated formats/standards)

6.1.3 Registration

1. Registering a ***member/organization*** requires:
    a. That the new **member** provide:

i. Organization Name (entity that may provide or consume services of IM BB or its member applications)
ii. Certificate of member/organization signing key issued by Certificate Authority which is trusted by the state (e.g., In Estonia, the "Estonian CA" keeps a list of valid certificates.)
iii. Access to signing key (HSM or Token storing a private key)
iv. The public IP address of this security server

b. That the **administrator** verifies and accepts the request for registration.

2. Registering an ***application***[4] requires:
   a. That the new **member** provide:
      i. Application Name
      ii. Certificate of transport key of application (all subsequent requests will use the private key associated with this certificate to secure connection
      iii. **Note that when configuring the application to interact with the Information Mediator, the public certificate of the Information Mediator (which is made available) must be added to a list of known/trusted certs of application.
   b. That the **administrator** verifies and accepts the request for registration.
3. Registering a ***service***[5] requires:
   a. That the owner of the application provide:
      i. OpenAPI 3.0 specification document (see example, includes base paths like "/api/patients", "/api/visits", etc.)
      ii. Description
      iii. List of enabled endpoints of OpenAPI spec (the requirement is that we can enable/disable endpoints)
4. Managing a list of **allowed consumers for services** requires:
   a. Application developers MAY access the directory service (described below) to select the services that they want to consume.
   b. The request/approval/addition of allowed consumers process is a business-first process with manual steps. (The hard part is negotiating data sharing agreements and signing contracts, when that is completed the Information Mediator administrator can easily modify the allowed consumers list manually.)
   c. An application must specify which member/application/service they want to access.
   d. The provider of that service must decide if the consumer is allowed.
   e. Once approved, the requesting application will be added to the list of allowed applications for the requested service.
      i. The list of consuming applications may change.
      ii. When the service is first designed and made available, there may be no consumers on the access list.
      iii. When a consumer decides that they want to access a service, if it is determined allowable they must be added to this list.

6.1.4 Accessing Services

1. To make a request to another service via the Information Mediator, an application **MUST**:
   a. Using REST, make a valid HTTP request to the local Information Mediator security server with headers which identifies itself at the application level.
   b. The components of the request must be:
      i. Security server URL
      ii. API version
      iii. Instance (e.g., Country)
      iv. Domain of member
      v. Member (e.g., Ministry of X)

[4] All applications have their own TLS certificate
[5] These are the API endpoints

vi. Application
vii. Service (OpenAPI file)
viii. Path
    1. Endpoint
    2. Query parameters

Example *ONLY*

Sample GET request:

`url-of-local-information-mediator-security-server/r1/INDIA/GOV/ministry-of-agriculture-karnataka/market-linkages-app/inventory-service/v1/check-level/apples?fresh=true`

Response: *{ data: 7 }*

Sample POST request:

`url-of-local-information-mediator-security-server/r1/INDIA/PVT/tata-buyers-corp-karnataka/small-farmer-buyers-app/inventory-service/v1/supply/apples`

With body: {"qtyAvailable": 4}

Response: *{ result: "Stock level report created." }*

How to interpret the above request paths:

`SECURITY-SERVER-URL/r1/INSTANCE/DOMAIN/MEMBER/APPLICATION/SERVICE/PATH`

   c. Note that all applications are making requests to the security server, which runs on their own network, *rather than* making requests to other applications directly over the public internet. (This is one of the main points of the security server and Information Mediator architecture.)
2. The Information Mediator signs & sends the request from application A to the security server for application B.
   a. Replace: "`url-of-local-information-mediator-security-server/r1/INDIA`" with "`https://api.moh.kn.in/security-server`", for example.
   b. The payload includes: `/GOV/ministry-of-agriculture-karnataka/market-linkages-app/inventory-service/v1/check-level/apples?fresh=true`
3. Security server for application B receives the request and validates the signature then forwards it to the application/service/endpoint.
4. The application/service/endpoint responds to this GET with `{ "status": 200, "body": { "apples": 47 } }`
5. Security server for application B signs the response and sends it back to the security server for application A.
6. Security server for application A validates signature and forwards response to application A.
7. Note that this is all synchronous. Application A's GET request is open/unresponded-to until step 6.

6.1.5 Directory Service

1. At development time, to see which organizations are available on GovStack, the administrator of application A sends a GET request to the security server: `url-of-security-server/listClients[?instanceId=INDIA]`
   a. Response is an array of organizations with descriptions.
   b. Parameter `instanceId` is needed only if a federated GovStack ecosystem is requested.
2. At development time, to learn which applications are available, the administrator/developer at application A sends a GET request to the security server `url-of-security-server/r1/INDIA/GOV/MEMBER/APPLICATION/{listMethods || allowedMethods}`
   a. Response is an array of services (either all services, or services that the requester is authorized to access via "allowedMethods").

3. At development time, to learn about an available service, administrator at application A sends a GET request to the security server:
`url-of-security-server/r1/INDIA/GOV/MEMBER/APPLICATION/getOpenApi?serviceCode=SERVICE`
    a. Response is an OpenAPI specification, detailing the endpoints and requirements for that service/API.
4. A view layer allowing for easy exploration of ALL clients, applications, and services should be provided. (Note that, "under the hood", this layer may make use of the APIs described above or be implemented via a separate API.)

6.2 Pub-Sub Layer

Any *implementation* of a PubSub layer is subjective—it will contain required shapes for requests and specific endpoints provided by the application implementing the PubSub functionality. The requirements below do not prescribe a specific implementation, rather they outline the required functionality that may be implemented in a number of common ways.

6.2.1 Adherence to Key Concepts

1. The Publisher does not know about Subscribers (but the Room does know.)
2. Events of the same type can be generated by different producers.
3. The Subscriber does not know about Producers.
    a. There was a previous concern that the disconnect between producer and consumer would lead to legal issues (specifically, that authority or "blame" could not be allocated when one had to validate the origin of a particular event).
    b. This has been addressed with "rooms". For example, the room containing the "birth" event type is the responsibility of the Ministry of Health (MoH), and the MoH is responsible for ensuring that all "birth event" producers are valid, and that all "birth event" subscribers are valid.
    c. So, if you care about "birth events", they may be sourced from lots of different hospitals/clinics/mobile-apps, but their validation and distribution is now the sole responsibility of the MoH. (I.e., You can trust the MoH to ensure the provenance of a birth event.)
4. The Room is responsible for the event lifecycle.

6.2.2 Facilitating Publish/Subscribe

1. The Room publishes event types and sets up GovStack service for accepting events of this type; in general, a single room might host multiple event types. Depending on the need, an implementation may also allocate each Room for a specific event type, if it helps separate subscribers. For e.g. in India, "Ministry Of Home Affairs" can be "Owner" of a Room where events of type "emergency" events are published by "citizens" or "entities" and entities such as "ambulance services", "Fire-fighting services", Hospitals, etc., can be Subscribers to this Room. In this scenario, an event of type "Fire emergency" can be published by any citizen enrolled into this Room. The Ministry can choose to have one "emergency" Room in each town, and enroll Subscribers relevant to respective regions.
2. Publishers and Subscribers discover available Rooms and event types.
3. The Subscriber requests that events of certain types be delivered to them.
4. The Subscriber specifies the desired **delivery mode** (push/pull). The Room and Subscriber conclude the delivery contract. Note that:
    a. **push** delivery mode is when the Room sends events via webhook to the Subscriber's API
    b. **pull** delivery mode (optional) means that the Room keeps a queue of events and the Subscriber can check that queue
5. At minimum, this BB should provide for a push delivery mode. It may also provide a pull mode based on convenience.

6. The Publisher and the Room have a data access contract to establish trust for accessing information.
    a. This is **akin to the standard trust contract** in the Information Mediator—the consumer must have rights to access a certain API.
    b. Publishers have to sign a contract with the Room owner's consent to gain access to specific Rooms and events. In the contract, Publishers declare what type of message will be published in a given Room.
7. The Publisher generates an event:
    a. The Publisher makes a POST call to the Room service of a particular event type.
    b. The Room stores the event and replies with the event id (e.g. uuid). Event id is generated by Room or is taken from the original event dataset if provided by the publisher.
    c. The Room sends an immediate acknowledgement to the Publisher.
8. The Room distributes an event as follows:
    a. A reference to publisher and event id is added in the event dataset.
    b. For each Subscriber:
        i. (alt) If the mode is 'push', make a POST call to Subscriber GovStack service of the event type;
        ii. (alt) If the mode is 'pull', enqueue an event for request from the Subscriber;
            1. There is a queue of events waiting to be processed per the Subscriber, such that Subscriber might periodically check to see events waiting in their own queue, process those events, and clear the queue.
            2. A pull mechanism is essential for resilience to network dropouts and traffic load balance at servers and differentiating urgent/emergency events from normal events. (This can be decided during implementation.)
9. (opt - if mode is PULL) The Subscriber pulls an event:
    a. The Subscriber makes a GET call to the Room service of the particular event type.
10. (opt) The Subscriber requests event details. Some event details may have more restricted regulations for handling and may be not included in event type. In this case, the Subscriber requests these details directly from the publisher by making a GET call to the referenced Publisher with event id as a parameter.
    a. This call implies the existence of an associated contract between the Subscriber and the Publisher.
11. The Information Mediator building block creates a log of all messages published and distributed.
12. This necessitates three end points to be declared per event type PubSub instance:
    a. an endpoint URI to be registered by every Subscriber on a per event type basis.
    b. (optional) an endpoint URI on the PubSub where all pull requests come from various Subscribers.
    c. an endpoint on the Room to send messages for publication.
13. An enrollment process of Subscribers and Publishers should capture these details.
14. If an event payload is very large, then it is recommended to just publish the "event" and let Subscribers get full details directly from the Publishers as needed. It will simplify event payloads, reduce TAT, storage and bandwidth significantly.

6.2.3 Defining Message Types

1. PubSub requires global agreement on event/message "types" (e.g., "newPatient" or "paymentCreated") and the PubSub layer must provide a mechanism for registering event types.
2. For each event type, a JSON schema for the required payload shape must be defined to facilitate consumption of those events.
3. An interface for registering and viewing event types must be provided.

6.2.4 Broadcasting a Message

1. An application must be able to make a POST request with a valid JSON payload as the body (see below) and specify the message type to a Room provided by the PubSub layer service.

6.2.5 Maintaining and Displaying Message Logs

1. All broadcast messages and deliveries/delivery statuses must be added to log.
2. There must be a possibility to search and view status of messages, for example, a message with type X and from 7 subscribers message had been successfully delivered to all 7.

6.2.6 Retrying Messages

1. Published messages SHOULD be stored so that delivery may be retried if certain Subscribers are offline.
    a. Subscribers should be able to *configure* their retry strategies, overriding the default exponential backoff on retryable errors. (See Google Pub/Sub for example: https://cloud.google.com/pubsub/docs/handling-failures)
2. If an active subscription exists but a 502 response is received when forwarding a message to that service, the PubSub layer SHOULD retry N times with a standard backoff. (An exponential backoff may be the default approach.)
    a. The backoff and retry strategy SHOULD be configurable by an administrator with access to the PubSub layer application.
3. If a message cannot be delivered the PubDub layer SHOULD drop that message.

6.2.7 Registering/Updating/Deleting a Subscription

1. The PubSub layer should provide an admin UI to help create/manage subscriptions with the data below.
2. The PubSub layer should allow an administrator to view a list of active subscriptions.
3. For registration via API, an application must be able to make a POST request to a service exposed by the PubSub layer which defines which endpoints certain event types should be sent to.
4. For services that provide an OpenAPI v3 webhooks specification, the PubSub layer should be able to create necessary webhooks on those applications via API.
5. See below for a JSON description of the active subscriptions for an application.

```
{
  securityServer: nk888,
  member: "112 (Ministry of Public Engagement) hl7-2.5",
  application: 'Patient Portal', // WHO I AM...
  subscriptions: [ // AND WHAT I WANT TO BE NOTIFIED ABOUT...
    { type: "newPatient", service: '/openfn/inbox/uuid' },
    { type: "newMedication", service: '/api/approvedMedsListing' },
  ]
}
```

6.2.8 Message Receipt/Delivery Logging and Audit Trail Generation

1. All events received and delivered MUST have a unique ID.
2. All events received by the PubSub layer MUST be logged or added to a log sync and those log entries MUST contain event metadata including the sender, timestamp, and event type, but MAY *not* include the event payload.
3. All event delivery attempts MUST be logged or added to a log sync

4. For every event message received, the IM sends back an acknowledgement with the ID of event to the respective Publisher

6.3 Logging

1. The Information Mediator BB maintains a message log.
    a. The purpose of the message log is to provide means to prove the reception of a regular request or response message to a third party. Messages exchanged between IM BB are signed and encrypted. For every regular request and response, the security server produces a complete signed and timestamped document. At minimum, the log must store metadata that identifies a specific message, a status of transaction carried out on that message by IM BB, along with source ID and date time stamp.
2. The information Mediator BB has full audit trail capabilities.
    a. The Information Mediator BB keeps an audit log. The audit log events are generated by the user interface when the administrator changes the system's state or configuration. The administrator actions are logged regardless of whether the outcome was a success or a failure.The system must be capable of emitting statistical reports for a given organization, application or service or consumer and status.

6.4 Monitoring

1. Operational monitoring
    a. Operational monitoring provides details about the request exchange,such as the ID-s of the client and the service, various attributes of the message read from the message header, request and response timestamps, sizes etc., but not the actual payload of messages.
    b. The operational monitoring daemon collects and shares operational monitoring data of the Information Mediator BB as part of request exchange, shares this data, and calculates and shares health statistics (the timestamps and number of successful/unsuccessful requests, various metrics of the duration and message size of the requests, etc.).
    c. The operational monitoring daemon makes operational and health data available to the owner of the security server, regular clients and the central monitoring client via the security server. (For example, local health data may be made available for external monitoring systems.)
    d. The owner of the security server and the central monitoring client are able to query the records of all clients. For a regular client, only the records associated with that client are available.
2. Environmental monitoring
    a. Environmental monitoring provides details of the security servers such as operating system, memory, disk space, CPU load, traffic load, running processes and installed packages, etc. in a chosen date range.
    b. Environmental monitoring provides a standard endpoint which can be accessed with a-client (e.g. Java's jconsole application if using JMX).
    c. It is possible to limit what allowed non-owners can request via environmental monitoring data request. The security server owner will always get the full data set as requested.

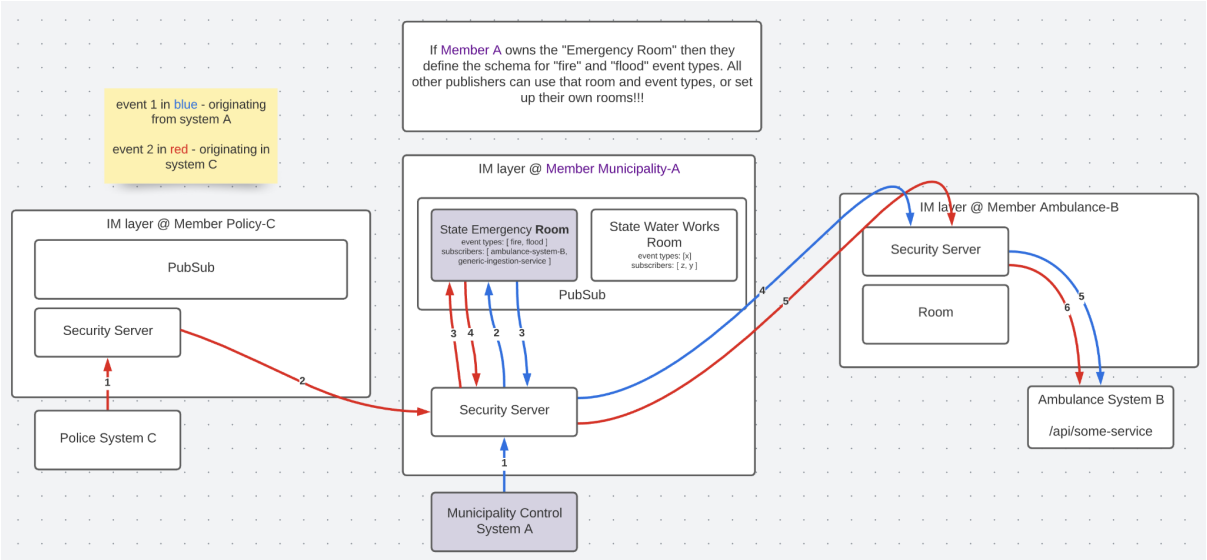
6.5 Scaling/Throughput

1. The Information Mediator BB supports provider-side high availability setup via a so-called internal load balancing mechanism. The setup works so that the same member / member class / member code / application / service code is configured on multiple security servers and the IM BB will then route the request to the server that responds the fastest.
2. Busy production systems may need scalable performance in addition to high availability. The IM BB supports external load balancing mechanisms to address both of these problems simultaneously. A load balancer is added in front of a security server cluster to route the requests based on a selected algorithm.
3. The team in charge of deploying the security server application on their hardware must consider the network infrastructure including a load balancer, _____, etc. The requirements for network infrastructure must be handled/defined in a (yet to exist) “Network BB”.

7 Requirements Clarification

7.1 Understanding Rooms

To understand Rooms, consider the following example: If the "birth" event type is the responsibility of the MoH, then the MoH is responsible for the Room containing the “birth” event type. Also, if MoH is running an instance of a security server on their infrastructure, they are also now running an instance of the Room software (for "birth", "sickness", and "visit" event types) behind that security server. There could be multiple Rooms under a single owner, and each Room might hold multiple event types. This gives the “owner” of an “event type” the authority to decide exactly who is allowed to publish events of that type and who is allowed to subscribe to events of that type.



https://lucid.app/lucidspark/ae9dba58-c15d-43b2-b8ef-9d15f6bd746c/edit - The above diagram shows push delivery mode.

Note how in the above diagram, “Ambulance-B” is a IM BB “member” that has subscribed its “Ambulance System B” REST service (`/api/some-service`) to “fire” events in the “State Emergency Room”.

8 Performance Requirements

This section specifies the basic parameters that an implementing government might use to establish performance requirements for scalability, throughput, and response times when reasonable/necessary. For example:

1. Minimum Throughput = 100 requests/sec
2. Maximum Latency = 1sec
3. Concurrency = 1000 concurrent requests
4. All solutions MUST be able to monitor and report, including but not limited to, resource consumption, throughput, latency, average latency, queue depth/backlogs, etc.
    a. All of these indicators MUST be available through an administrative API.
    b. Ideally, all BBs should be able to run a “monitoring agent” which handles reporting out logs, requests, BB-specific indicators, etc. to a monitoring service (e.g., https://www.datadoghq.com/)
    c. The local monitoring agent should be configurable via web interface.
5. Retries and backoff strategies must be configurable.
6. There are specific “Scaling/Throughput” requirements in the functional requirements section.

9 Data Structures

The resource model shows the relationship between data objects that are used by the Information Mediator building block.

The data elements provide detail for the resource model. All data element schemas can be viewed, commented on, and modified in the schemas section of the IM BB Github repository.

9.1 Standards

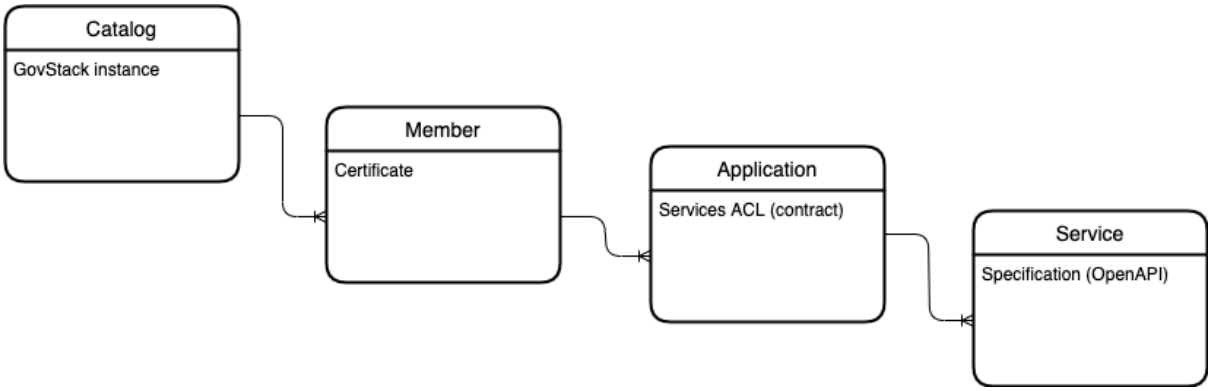
The following standards are applicable to data structures in the BB:

1. Requests will be made using HTTPS.
2. Request bodies will be valid JSON.
3. The description of specific services and the data formats they require will be specified in an OpenAPI 3.1 specification.

9.2 Service Access Layer

9.2.1 Resource Model

The Mediator BB key element is Service. The Service is used by a consuming building block or an application and offered by a provider building block or Application. Both Provider and Consumer MUST be Members of Mediator BB. Members of the mediator BB can be an organization (governmental or not, business or not) or a person (citizen as a rule).



Draw.io source in github:
https://github.com/GovStackWorkingGroup/BuildingBlockAPI/blob/main/IM/diagrams/Mediator-BB-entities.drawio.png.

To become a Member of Mediator BB participant must fulfil declared requirements and apply for onboarding. In the process of onboarding a Member is registered with Mediator BB and gets credentials to connect to Mediator BB. Normally it is done the way that a Member provides a certificate of recognized CA and requests signed with this certificate are considered as legitimate requests of the Member. A member entity can access the PubSub configuration and register a room to publish its own event type as a publisher through its own admin.

Members can browse a directory of Services available in Mediator BB. Each service is described in OpenAPI.

9.2.2 Data Elements

9.2.2.1 Member

Member:
https://github.com/GovStackWorkingGroup/BuildingBlockAPI/blob/main/IM/schemas/member.json

Member:
- Member Class enumeration
- Member Code string

Signing Key:
- Token enumeration (where to put)
- Name string
- Name of CA enumeration

9.2.2.2 Application

Application:
https://github.com/GovStackWorkingGroup/BuildingBlockAPI/blob/main/IM/schemas/application.json

Application:
- Connection type enum http/https
- TLS certificate cert

9.2.2.3 Service

Service:
https://github.com/GovStackWorkingGroup/BuildingBlockAPI/blob/main/IM/schemas/service.json

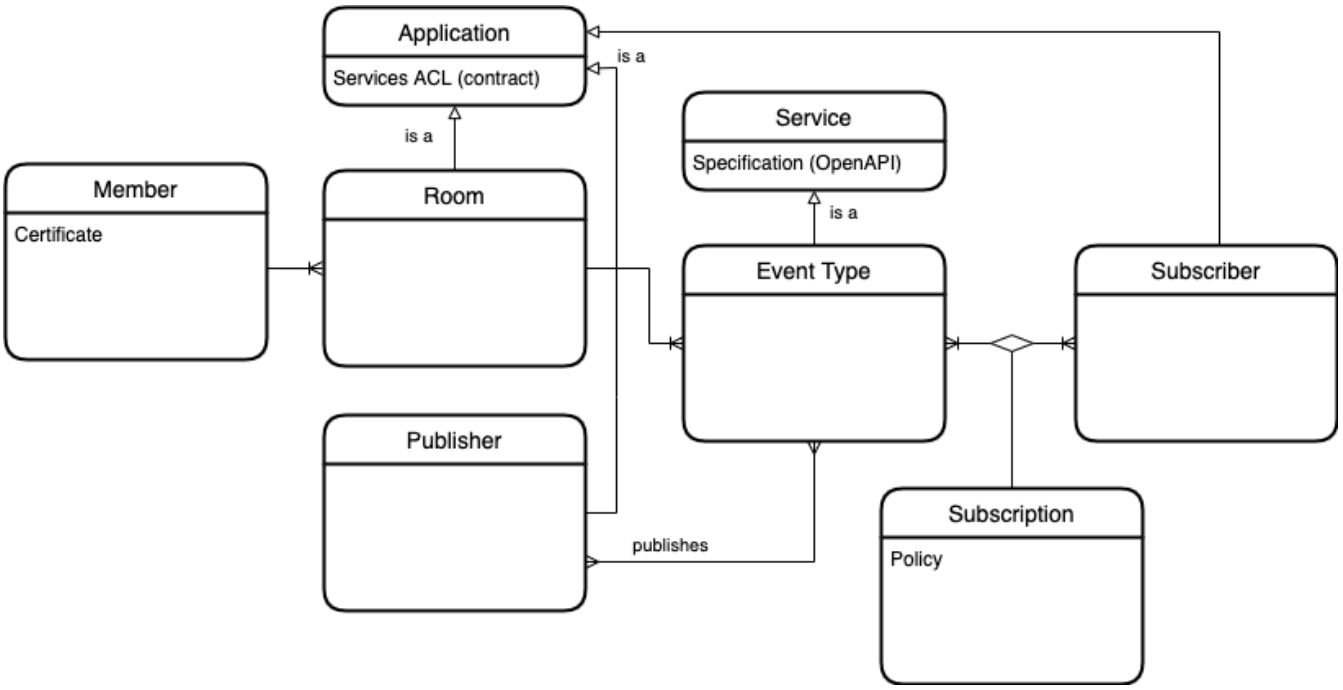
Services:

Service Code	string		
Description URL		url	OpenAPI spec
Service URL		url	base address for endpoint (prefix)
Access Rights	ACL		

9.3 PubSub Layer

9.3.1 Resource Model

Resource Model is extension of Access Layer model:



Draw.io source in github:
https://github.com/GovStackWorkingGroup/BuildingBlockAPI/blob/main/IM/diagrams/Mediator-BB-PubSub-entities.drawio.png

9.3.2 Data Elements

9.3.2.1 Event

An **event** is a set of data described by event type. Each event has an id, and this is most likely a uuid.

9.3.2.2 Event Type

An **event type** is a service described by OpenAPI. Each event type is owned by a certain authority. (E.g. the MoH might own the "new_birth" event type and define its schema.)

9.3.2.3 Publisher

A **publisher** is a GovStack application that produces events and sends them to Rooms.

9.3.2.4 Room

A **room** is a GovStack application that handles the distribution of events. Each Room has a set of connected event types. (E.g., the "birth" room might contain three event types: "new_birth", "birth_complication", and "infant_death".) A room is located in the member's local IM BB implementation and the member is responsible for all types of events in that particular room.

9.3.2.5 Subscriber

A **subscriber** is a GovStack application that can process events. Subscribers are independent of each other and their business logic is different (as rule). Each subscriber processes events from their own perspective.

Schema description:
https://github.com/GovStackWorkingGroup/BuildingBlockAPI/blob/main/IM/schemas/broadcast.json

10 Service APIs

This section describes external APIs that must be implemented by the IM building block. Additional APIs may be implemented by the building block (all APIs must adhere to the standards and protocols defined), but the listed APIs define a minimal set that must be provided by any implementation.

The majority of functions provided by the IM building block are either defined in the "service access flow" or configured by the administrator via the web UI. There is, however, a "Directory Service" which can provide listings of clients, methods, and available API specs for services on the information mediator. The directory is managed by admins of members. The directory service centralizes and offers knowledge of all enrolled members and their services along with information necessary to bind a third party application as a consumer of that service. These services are described here:

- https://github.com/GovStackWorkingGroup/BuildingBlockAPI/blob/main/IM/govstack_im_service_metadata_api-0.3-swagger.json
- https://github.com/GovStackWorkingGroup/BuildingBlockAPI/tree/main/IM

and changes to the API definitions can be made by submitting a pull request on this repository.

The Swagger variant is available here:
https://app.swaggerhub.com/apis/GovStack/gov-stack_im_service_metadata_api/0.3

The services can be accessed via the following Service APIs:

10.1 Member Discovery API

At development time, to see which organizations are available on GovStack, an administrator of application A sends a GET request to the security server:
`url-of-local-information-mediator-security-server/r1/listClients`

The response is an **array of organizations** with descriptions.

10.2 Service Discovery API

At development time, an administrator at application A sends a GET request to the security server:
`url-of-local-information-mediator-security-server/r1/INDIA/GOV/MEMBER/APPLICATION/{listMethods || allowedMethods}`

The response is an **array of services** (either all services, or services that the requester is authorized to access via "allowedMethods").

10.3 Service Detail API

At development time, to learn about an available service, an administrator at application A sends a GET request to the security server:
`url-of-local-information-mediator-security-server/r1/INDIA/GOV/MEMBER/APPLICATION/getOpenApi?serviceCode=SERVICE`

The response is **an OpenAPI specification**, detailing the endpoints and requirements for that service/API of requested Service of Application.

10.4 Reporting API

At debugging time, to learn about system performance or retrieve an audit log, an administrator may send a request to the reporting API.

The response is <audit trail>, <metrics>, etc.

10.5 Service Access API

The full technical specification on how to call service is presented in the "Detailed Flows" section below 13.1.6 Sending a message from A to B.

The full API definition of all available services can be produced from the set of all available OpenAPI descriptions. For that one need in all occurrences of:
"path": { "<some-value>": … }
prepend <some-value> with
{GovStackInstance}/{memberClass}/{member}/{application}/{service}/
extending this way the path with these details.

10.6 Broadcast API

To broadcast a message to a Room, the service access API must be followed and the service requested must be the service implementing event type.

11 Workflows

A workflow provides a detailed view of how the IM building block will interact with other building blocks to support common use cases. This section lists workflows that this building block must support. Other workflows may be implemented in addition to those listed.

Note that for the Information Mediator, the primary workflow for the Service Access layer is specified in the "Accessing Services" section above. The workflow for the "PubSub layer" is outlined below.

11.1 Broadcast Data to Multiple Services

This workflow describes how data about a particular event is broadcast to multiple services. Note that in this case, and in all PubSub implementations, the sender of the data does not know who will receive the data or what they will do with it. Likewise, the recipient of the data does not know who broadcast the data.

For example, "Clinic System A" wishes to broadcast data about a new birth so that it can be used to trigger asynchronous actions in several other systems. A webhook is configured in Clinic System A that makes an HTTP POST request corresponding to the type of event and some valid JSON in the body to the PubSub provider. The provider maintains a list of active "Subscribers" for that event type. Without manipulating the body, that message is effectively forwarded on to each subscriber, so that REST services in the "Demographics Tracker", "Insurance Registry", and "Early Childhood Education" applications can consume that data and do things based on it.

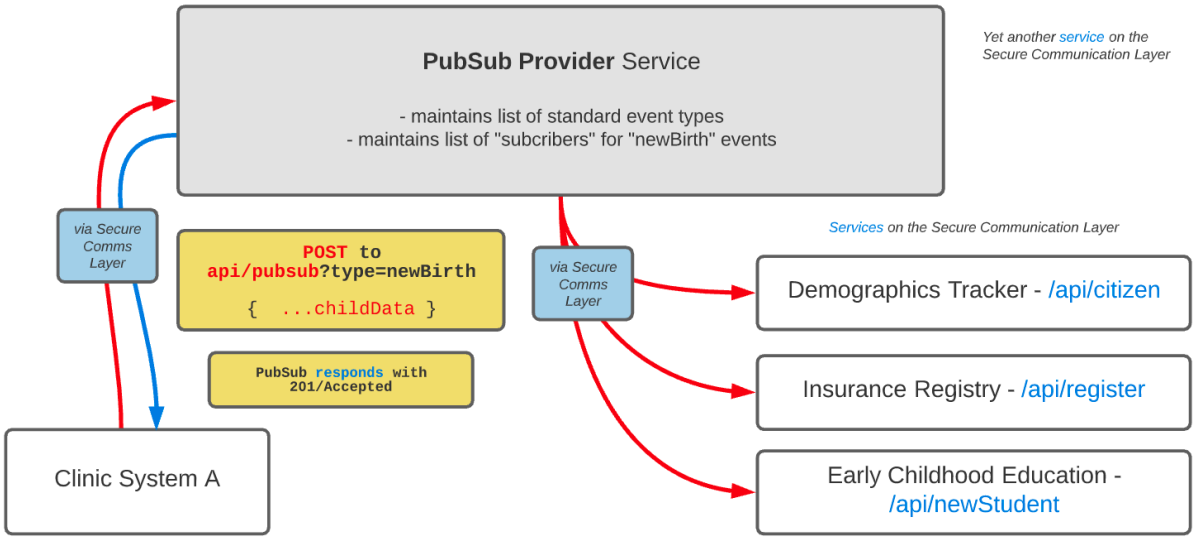
It should be noted that the shape of the payload will be agreed upon beforehand, so that the responsibility of being able to "ingest" a "newBirth" payload from PubSub will fall on the REST service provider. Once an event is ingested, the responsibility to deliver the payload lies with the IM "room" based on the importance given to the event by a Subscriber. Thus, during registration of the Subscriber, parameters such as max queue depth, retries, failsafe mechanisms and error handling have to be configured by each Subscriber as they subscribe to an event type.

11.2 Interaction with Other Building Blocks

This workflow describes multicase interaction between building blocks.

Sequence Diagram:

The sequence diagram shows the flow of data between building blocks for this workflow.



12 Key Decision Log

1. 2021-06-15: Added "PubSub" and "translation" layer *to* the IM scope. They were previously considered out of scope.
2. 2021-06-15: Broke IM specification into 3 segments: (1) Service Access layer; (2) PubSub layer; (3) translation layer.
3. 2021-06-24: Removed "translation" layer from the IM scope. (Max, Ramkumar, Travis, Ingmar, Taylor)
4. 2021-07-15: Decide to push towards a "shareable" v1 and send it to architecture for formal review *before* inviting outside reviewers from Kulli's list of CVs. (Max, Ramkumar, Aleksander, Taylor, Ingmar)
5. 2021-07-22: Move away from "Flux Standard Actions" ({ type: 'newBirth', payload: { ...data }) in favor of a "type" query parameter in the POST request. To broadcast a newBirth payload, make a POST request to /api/pubsub?type=NEW_BIRTH with valid JSON body with the data. This makes it clearer that we will *never* know about or modify the body. (Also leaves the door open for non-JSON bodies.)
6. 2021-08-05: Architecture group present: an API must exist to manage users externally so that an IAM solution can be used to provision and manage IM admin users.
7. 2021-08-05: Architecture group present: OpenAPI specs and JSON schemas for the entities defined in the Github repo should be *real requirements*, not examples/samples. They should only be for a small subset, but they *must* be adhered to by systems which serve this BB.
8. 2021-09-27: Ramkumar feeds back from Architecture Group that there is a need for a monitoring facility that is decoupled from individual BBs. The individual BBs (incl. IM) must determine what is critical for them to have monitored and then their monitoring agent must be able to emit that data or allow an external service to query for it. These indicators may include generic indicators such as network latency, requests/traffic, load, etc. The generic indicators (incl. network traffic indicators) must be available to the monitoring/threat-detection service, but then it's up to that service to configure threat rules: e.g., 10 consecutive failed login attempts for a single API/user should trigger X, Y, Z
9. 2021-09-27: Administrative rights (TD, AR, R)
    a. The **super-admin** (admin for GLOBAL IM system) can
        i. Register members
        ii. Register member applications (approve/deny request)

iii. Activate/suspend access for members and applications (this might be achieved by e.g., revoking a certificate for members). Activation and suspension of member and application access is performed in administrative UI the way similar to handling client state described in section 13.1.2 Security Server Client States

b. [OPTIONAL] The **security server admin** can
   i. Add and supervise the admins for each member (in some cases, a single security server *may* have multiple members—e.g., the Ministry of Health, and the Ministry of Public Health Informatics—and this security server admin may create and delegate responsibilities to the member admins which sit "inside" the single security server instance.)

c. The **member admin** (admin for a member) can
   i. Request registration for applications
   ii. Register application services
   iii. Request event type subscription (we need to design a new target multicast feature)
   iv. Approvie event type subscription (we need to design a new target multicast feature)

10. 2019-09-29: PubSub is not very useful without additional information about who has sent data and who is subscribed to receive it.
    - Extend PubSub so that each event contains:
      i. `{ type: "new_birth", payload, sender }`
      ii. `{ type: "__XYZ__", payload, sender } // everyone subscribes`
      iii. `{ type: "fire_emergency", { data }, sender } // drives API action`
      iv. `{ type: "notification", "Please help me", sender } // notification`
    - The process of subscribing to events now requires a request to subscribe to an event type *from a specific sender* and the sender must agree/authorize.
    - Once a subscription is authorized, the sender cannot "filter" who receives a broadcast. (This is PubSub.) The "filtering" happens on the receiver end.

11. 2019-10-06: Open Question from Ramkumar (noting pushback from AK/TD): Should PubSub event *producers* (senders) be able to choose which subscribers receive each event (from the list of subscribers)? Decided that currently, event publishers will not have any control on who subscribes to the event. So no filtering is possible from the publisher side. Deferring publisher based filtering until such a use case is identified
    - Since we have a fair amount of specification already laid out and since there are very clear, well documented industry standards around pub sub (see below)
      i. https://aws.amazon.com/pub-sub-messaging/
      ii. https://cloud.google.com/pubsub
      iii. https://azure.microsoft.com/en-gb/services/web-pubsub/
    - AND, since any PubSub server/implementation would merely sit "on" the information mediator anyway, as another "sub-building-block" we recommend splitting it out and documenting all of our current progress on that BB *over there* as a version 1 (or even a version 0.1).

12. 2021-11-30: Taylor, Aleksander, Ramkumar - PubSub is back, and will be merged into the document above based on the PubSub Proposal which has been approved by Max and the architecture team.

GovStack

13 Future Considerations

This section is bassed on feedback from reviewers on 2022-02-23.

- We lack adequate examples from domains outside health and this makes the "whole of government" approach which underpins the IM BB unclear.
- We must explain WHAT the specification *IS* before diving in. This document is "a set of requirements for a building block that will enable a certain strategy for application-to-application communication across a government software ecosystem."
- We must explain more clearly that the IM BB *neither* produces nor consumes information—it merely moves information between producers and consumers in a secure, structured fashion.
- We must better articulate the "discovery" services, ensuring that the process of finding members, applications, and available services is clear and obvious.
- We must add a simplified diagram and process description that explains the relationship between the central server and the various security servers—pay careful attention to the "discovery" process and how a copy of the "global configuration" is distributed across the network of security servers.

14 Annexes and Appendices

14.1 Detailed Technical Flows

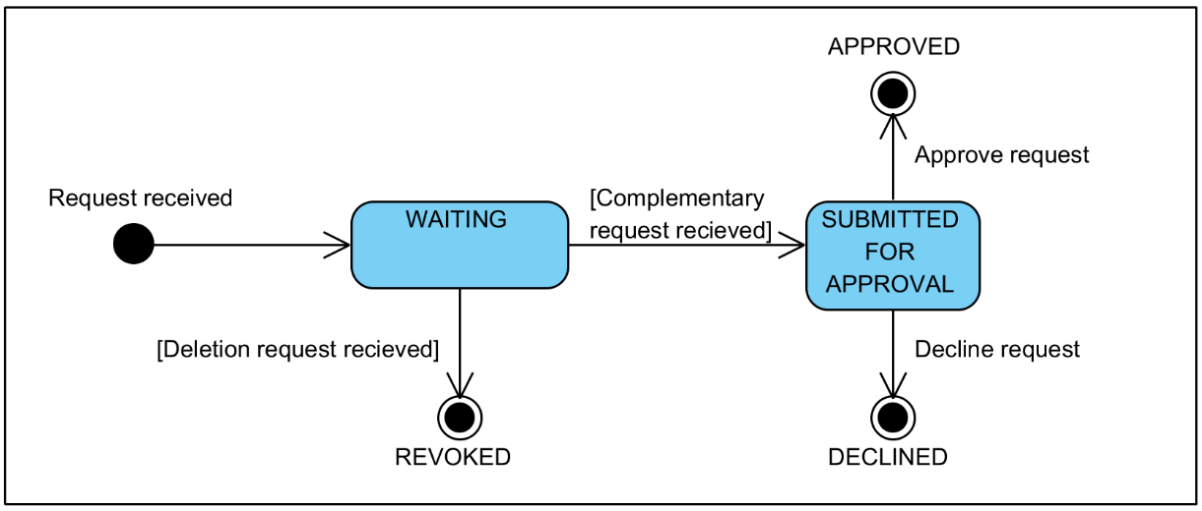
The detailed flows section is a description of how various clients (members, applications, services) are registered and managed on the information mediator.

14.1.1 Registering a Member

A "member" on the information mediator (e.g. "Ministry of Agriculture") is the organization (maybe with their own network) which runs a number of applications (e.g., "Market Linkage", "Rural Agricultural Advisory Services") which have their own services (e.g., "/api/refer :: Create Education Referral Webservice", "/api/inventory-service :: Update Market Inventory").

14.1.2 Security Server Client States

The security server client registration passes following states:



Source in github:
https://github.com/GovStackWorkingGroup/BuildingBlockAPI/blob/main/IM/diagrams/state-machine-diagram-for-registration-requests.png

14.1.3 Adding a Security Server Client

Required Interface: Security Server; Required Role for **Access rights:** Registration Officer

Follow these steps.

1. There is a formal business/legal process which is a precondition for adding a new "client" (Security Server)
    1. It is never one person simply adding a new app or service by their own will. A traceable formal request from an authorized authority of the entity to register this application/service into IM will be there.
    2. This request will need approval from the IM governance authorized person, before it gets into the system. The request and approval documents may be by email or hard copy. In either case a copy of those documents must be mandatory attachments to this registration/registration process of members, apps, services.
    3. Once the formal approval is acquired and properly documented, the technical steps (carried out by an authorized system administrator) are outlined below.
2. In the **CLIENTS** view, click **ADD CLIENT**.
3. In the wizard that opens
    1. Client details page: Select an existing client from the Global list by pressing **SELECT CLIENT** or specify the details of the Client to be added manually and click **NEXT**
    2. Token page: Select the token where you want to add the SIGN key for the new Client. Click **NEXT**
    3. Sign key page: Define a label (optional) for the newly created SIGN key and click **NEXT**

4. CSR details page: Select the Certification Authority (CA) that will issue the certificate in **Certification Service** field and format of the certificate signing request according to the CA's requirements in the **CSR Format** field. Click **NEXT**.

5. Generate CSR page: Define **Organization Name (O)** and click **NEXT**

6. Finish page: click **SUBMIT** and the new client will be added to the Clients list and the new key and CSR will appear in the Keys and Certificates view.

The new client is added to the list of security server clients in the "Saved" state.

14.1.4 Registering an Application

(Adding a Security Server Member Application)

Interface: Security Server; **Access rights:** Registration Officer

Follow these steps.

1. In the **CLIENTS** view in the client list, locate the Service Access gateway member you want to add an application to and click **Add Subsystem** at the end of the row.

2. In the wizard that opens
   2.1. Select an existing application from the Global list by pressing **SELECT SUBSYSTEM** or specify the **Subsystem Code** manually
   2.2. If you wish to register the new application immediately, check the **Register subsystem** checkbox and then click **ADD SUBSYSTEM**.
   (2.3.) If you checked the **Register subsystem** checkbox, a popup will appear asking whether you wish to register the application immediately. In the popup, click **YES**.

The new application is added to the list of security server clients in the "Saved" state.

14.1.5 Registering a Service

Once a member entity and app is registered by IM admin, the admin of that entity can register different services provided by the entity. Otherwise the IM administrator will be inundated by changes across so many services and applications that may use informed
Interface: Security Server; **Access rights:** Service Administrator

After a new REST service is added, the security server displays text "REST" and url for that service.

To add a REST service, follow these steps.

1. Navigate to the CLIENTS tab, click the name of the client for which you wish to add REST service to and click the **SERVICES** tab.

2. Click **ADD REST**. Select whether the URL type is "REST API Base Path" or "OpenAPI 3 Description". Enter the url and service code in the window that opens and click **ADD**.

3. Once the window is closed, the url and the service code are added to the service list. If the added URL type was OpenAPI 3 description, the service description is parsed and endpoints are added under the service. By default, the REST service is added in disabled state.

To see the service details under the REST service

- click the caret on the REST service description row to expand the service details.

Example of service description is in Annex A.

14.1.6 Sending a Message from A to B

(through Service Access gateway)

14.1.6.1 REST Interface

HTTP version 1.1 is used by the protocol as described in [RFC2616]. The consumer member/application is specified using HTTP headers. The service to be called is encoded as part of the HTTP/HTTPS request URL. Here is the generic form of the REST service call.

Request format

{http-request-method} /{protocol-version}/{serviceId}[/path][?query-parameters]

HTTP request headers

X-GovStack-Client: {client}

- **{http-request-method}** can be one of the request methods defined in [RFC7231]. For example GET, POST, PUT and DELETE.
- **{protocol-version}**: specifies the major version of the Service Access gateway Message Protocol for REST. For the initial version r1 MUST be used.
- **{client}**: specifies the member/application that is used as a service client - an entity that initiates the service call. The identifier consists of the following parts: [GovStack instance]/[member class]/[member code]/[application code]. Including the application code is OPTIONAL.
- **{serviceId}** identifies the service that is registered under {provider-application} and invoked by the request. {serviceId} contains the following parts:
    - [GovStack instance]/[member class]/[member code]/[application code]/[service code]. Including the application code is OPTIONAL.
    - The **{serviceId}** is mapped to an actual service URL by the Security Server (see the example below).
- **[path]** contains the relative path to the service to be called
- **[query-parameters]** contains the query parameters to be sent to the service

Here is a practical example of an Service Access gateway REST call.

Request example

GET /r1/INSTANCE/CLASS2/MEMBER2/APPLICATION2/BARSERVICE/v1/bar/zyggy?quu=1

HTTP request headers

X-GovStack-Client: INSTANCE/CLASS1/MEMBER1/APPLICATION1

Breakdown of the request URI:

- **{http-request-method}**: GET
- **{protocol-version}**: /r1
- **{client}**: INSTANCE/CLASS1/MEMBER1/APPLICATION1
- **{serviceId}**: /INSTANCE/CLASS2/MEMBER2/APPLICATION2/BARSERVICE
- **[path]**: /v1/bar/zyggy
- **[query-parameters]**: ?quu=1

Assuming that the serviceId maps to the URL https://barservice.example.org/, the provider will see the request GET https://barservice.example.org/v1/bar/zyggy?quu=1. The reason for naming the service independently of the path is that the same provider could have a fooservice as well (https://fooservice.example.org/), in which case it would be difficult to tell the services apart if the path was the service Id (both services could have paths like "/v1/...") unless the fooservice was attached to a separate application.

On the Security Server side the incoming request URIs MUST be validated. Input strings from the user can't be trusted. Lengths of the strings need to be checked and maximum length or the request URI needs to be limited. Although the URI standard does not specify a maximum size of the URL, most clients enforce an arbitrary limit of 2000 characters. The Security Server implementation MAY do this as well. Sending data that is difficult to express in a hierarchical manner, and especially data that is larger than this 2000 character limit, should be transmitted in the body of the request.

14.1.6.2 Use of HTTP Headers

There is only one mandatory HTTP header in the protocol that needs to be set by the client. Otherwise the use of headers in Service Access gateway REST service calls is OPTIONAL. The mandatory header and the most common optional header types and their operation are described next.

Note. HTTP headers are not case-sensitive. X-GovStack-Client and x-govstack-client are both valid header names.

Mandatory X-GovStack headers in the request

X-GovStack-Client: Specifies the member/application that is used as a service client - an entity that initiates the service call. The identifier consists of the following parts: [GovStack instance]/[member class]/[member code]/[application code]. Including the application code is OPTIONAL. The identifier parts MUST be represented as UTF-8 and encoded using percent encoding.

X-GovStack-Client: INSTANCE/CLASS/MEMBER/APPLICATION

Service Access gateway specific headers returned in the response

The response contains some Service Access gateway specific headers that are set by the provider Security Server. The provider service SHOULD NOT set these headers since in that case they will be overwritten.

- **X-GovStack-Client**: Specifies the member/application that is used as a service client

- **X-GovStack-Service**: Specifies the serviceId that is invoked by the service client
- **X-GovStack-Id**: Unique identifier for this message
- **X-GovStack-Request-Hash**: For responses, this field contains sha-512 encoded hash of the request message
- **X-GovStack-Error**: This header is provided in case there was an error processing the request and it occurred somewhere in Service Access gateway (on the consumer or provider Security Server)

X-GovStack-Request-Id: Unique identifier for the request
X-GovStack-Client: INSTANCE/CLASS/MEMBER/APPLICATION
X-GovStack-Service: INSTANCE/CLASS/MEMBER/APPLICATION/PETSTORE
X-GovStack-Id: fa2e18a5-c2cb-4d09-b994-f57727f7c3fb
X-GovStack-Request-Hash: 4c519cf0-0e5e-4ccf-b72b-8ed6fe289e6e
X-GovStack-Request-Id: f92591a3-6bf0-49b1-987b-0dd78c034cc3

Request hash header

- **X-GovStack-Request-Hash**: For responses, this field SHALL contain the base-64 encoded SHA512(SHA512(headers)+SHA512(body)). If there is no body, then only the headers are included in the calculation i.e. the field contains the base-64 encoded SHA512(headers). This field is automatically filled in by the service provider's Security Server. The field is used to create a strong connection between a request and a response. Thus, it is possible to prove, for example, that a certain registry record is returned in response to a certain query.
- The request hash header MUST be automatically created by the service provider's Security Server and it MUST be verified by the service client's Security Server
- The request message SHOULD NOT contain the request hash header.

The response message returned by a service provider SHOULD NOT contain the request hash header. If the response message contains the request hash header, the service provider's Security Server MUST ignore the field and replace it with the created field.
X-GovStack-Request-Hash:
14sEri8SmLNy/DJyTob0ZddAskmdRy5ZUyhbr33iLkaA+gLpWcivUH16fzbuIs7hhs2AnA4lJDloylihXMlVQA==

Content-Type header

- With REST messages that include the request body it is RECOMMENDED that the content's media type is indicated with this header. Additionally it is RECOMMENDED to use the charset parameter to indicate the character encoding used in the REST message.
- The REST messages originating from the Security Server (e.g. error messages) MUST include the header and indicate the content's type and character encoding with it.
- If Content-Type header is included in the request message by the consumer information system, it MUST be transported unmodified through Service Access gateway to the provider information system

If Content-Type header is included in the response message by the provider information system, it MUST be transported unmodified through Service Access gateway to the consumer information

system
Content-Type: application/json; charset=utf-8
Content-Type: multipart/form-data; boundary=something

In case the service consumer does not provide the Content-Type header (or some of its components), the request message is anyhow passed to the provider service which can decide what to do with it.

Accept header

- It is RECOMMENDED that the service consumer advertises the content types it is able to understand by including the Accept header in the request message.

If Accept header is included in the request message, it MUST be transported unmodified through Service Access gateway to the service provider.
Accept: application/xml

In case the service consumer does not provide the Accept header, the Security Server MUST use the default content-type application/json.

Security Server and X-GovStack extension headers

- **X-GovStack-Security-Server**: To send the request to a specific Security Server this header needs to be included. It contains the following parts
    - [GovStack instance]/[member class]/[member code]/[server code]

Other Service Access gateway extension headers are not defined in this document. Rather they are just contracts between information systems and Service Access gateway handles them like any user defined header.
X-GovStack-Security-Server: INSTANCE/MEMBERCLASS/MEMBERCODE/SERVERCODE

Optional X-GovStack headers

- **X-GovStack-Id**: Unique identifier for this message. It is RECOMMENDED to use universally unique identifiers [UUID]. If X-GovStack-Id is not provided, it SHALL be generated by the consumer Security Server. The provider Security Server SHALL include the X-GovStack-Id header in the response message.

- **X-GovStack-UserId**: User whose action initiated the request. The user ID should be prefixed with two-letter ISO country code (e.g., EE12345678901).

- **X-GovStack-Issue**: Identifies received application, issue or document that was the cause of the service request. This field may be used by the client information system to connect service requests (and responses) to working procedures.

X-GovStack-Id: fa2e18a5-c2cb-4d09-b994-f57727f7c3fb
X-GovStack-UserId: EE12345678901
X-GovStack-Issue: MT324223MSD

X-GovStack error header

- **X-GovStack-Error**: This header is provided in case there was an error processing the request and it occurred somewhere in Service Access gateway (on the consumer or provider Security Server). With it the client can easily distinguish between the errors

occurring on provider services and errors on Security Servers. Note that the header does not contain detailed error information but is more like a flag indicator to the interested parties. The header contains only the error type and the more detailed information such as the HTTP response code, error message body etc. need to be read from the response body.
Server.ServerProxy.DatabaseError

User defined headers

User defined HTTP headers (i.e. the headers not mentioned in [https://en.wikipedia.org/wiki/List_of_HTTP_header_fields] or this document) MUST be passed to the recipient unmodified by Security Server.
X-Powered-By: PHP/5.2.17
X-Pingback: https://example.com/xmlrpc.php

Cache headers

Service Access gateway does not cache messages.
Cache headers MUST be passed as-is and the consumer/provider MAY take advantage of this information.
Cache-Control: no-cache, no-store, must-revalidate
Pragma: no-cache

Cross-origin resource sharing

- Security Server is not designed to be a direct proxy for a web front-end. It does not do anything specific to enable cross-origin resource sharing (CORS).

Filtered headers

Some HTTP headers MUST be rewritten by the Security Server. The original value, if any, will not be passed through. The Security Server will either provide a new value or not send the header at all.

- Hop-by-hop headers
    - Connection, Keep-Alive, Proxy-Authenticate, Proxy-Authorization, TE, Trailer, Transfer-Encoding, Upgrade
- Headers that can leak the name or address of the origin host
    - Host
    - User-Agent
    - Server

Specially handled headers

Some HTTP headers are handled by the Security Server and the user should not expect that they are passed through Service Access gateway unmodified.

- Expect
    - Expectation 100 Continue MAY be handled locally at the consumer Security Server. Support for other expectations is OPTIONAL.

- (100 Continue is the only expectation defined by [RFC7231])

- Content-Length
  - The Security Server MAY change the transfer encoding, thus removing or adding a content-length header as necessary.

14.1.6.3 Examples

14.1.6.4 General

The pet store service used in the following examples has an [OPENAPI3] service description file available in Annex A. The most important aspects of the service are described in the text but for more details please refer to the aforementioned service description file.

The examples assume that the serviceId is mapped to https://petstore.niis.org/.

GET Request and Response

REQUEST

Service	Method	Description	Parameters
/pets/{petId}	GET	Finds pet by ID	petId - ID of pet to return

Service called directly

curl -X GET "https://petstore.niis.org/v2/pets/1124" -H "accept: application/json"

Service called through Service Access gateway

curl -X GET "https://{securityserver}/r1/{serviceId}/v2/pets/1124" -H "accept: application/json" -H "X-GovStack-Client: {client}"

Service response


```
{
  "id": 1124,
  "name": "Siddu",
  "photoUrls": [],
  "tags": [],
  "status": "Offline"
}
```


Service response code

200

Service response headers

Content-Type: application/json;charset=utf-8
Date: Thu, 21 Mar 2019 12:36:39 GMT
X-GovStack-id: 29f4d011-ef17-4f2f-9bb1-0452ce17d3f5

X-GovStack-client: DEV/COM/222/TESTCLIENT
X-GovStack-service: DEV/COM/222/TESTSERVICE/petstore
X-GovStack-request-id: f92591a3-6bf0-49b1-987b-0dd78c034cc3
X-GovStack-request-hash:
Xvx9V2U5c5RhDUiXpVLtW7L8vTd5cM2IOBU2n9efEk7/m3ECKyGAp7yTpJpTWpo6HcmwSaGO+cinxMVKjxJTOQ==
Content-Length: 1148

PUT Request and Response

REQUEST

Service	Method	Description	Parameters
/pets/{petId}	PUT	Update an existing pet	body - Pet object that will be updated in the store

Service called directly

curl -X PUT "https://petstore.niis.org/v2/pets/5657082955040009" -H "accept: application/json" -H "Content-Type: application/json" -d '{ "id": 0, "category": { "id": 0, "name": "string" }, "name": "doggie", "photoUrls": ["string"], "tags": [{ "id": 0, "name": "string" }], "status": "available"}'

Service called through Service Access gateway

curl -X PUT "https://{securityserver}/r1/{serviceId}/v2/pets/5657082955040009" -H "accept: application/json" -H "Content-Type: application/json" -H "X-GovStack-Client: {client}" -d '{ "id": 0, "category": { "id": 0, "name": "string" }, "name": "doggie", "photoUrls": ["string"], "tags": [{ "id": 0, "name": "string" }], "status": "available"}'

Service response


```
{
  "id": 5657082955040009,
  "category": {
    "id": 0,
    "name": "string"
  },
  "name": "doggie",
  "photoUrls": [
    "string"
  ],
  "tags": [
    {
      "id": 0,
      "name": "string"
    }
  ],
  "status": "available"
}
```


Service response code

200

Service response headers

Date: Thu, 21 Mar 2019 12:43:33 GMT
X-GovStack-id: acdb2c7a-c705-41c2-b595-4cd62e78316e
X-GovStack-client: DEV/COM/222/TESTCLIENT
X-GovStack-service: DEV/COM/222/TESTSERVICE/petstore
X-GovStack-request-id: f92591a3-6bf0-49b1-987b-0dd78c034cc3
X-GovStack-request-hash:
MOEfTqBjdqYiX3db9hxJ6JvHvCpYqfA6t0Uhdv6g2I2gfMY8ld4CbN8tslj6mUQPXoRaUdPm7NdZeA
YTg6zi+A==
Content-Length: 0

POST Request and Response

REQUEST

Service	Method	Description	Parameters
/pets	POST	Add a new pet to the store	body - Pet object that needs to be added to the store

Service called directly

curl -X POST "https://petstore.niis.org/v2/pets" -H "accept: application/json" -H "Content-Type: application/json" -d '{ "id": 0, "category": { "id": 0, "name": "string" }, "name": "doggie", "photoUrls": ["string"], "tags": [{ "id": 0, "name": "string" }], "status": "available"}'

Service called through Service Access gateway

curl -X POST "https://{securityserver}/r1/{serviceId}/v2/pets" -H "accept: application/json" -H "Content-Type: application/json" -H "X-GovStack-Client: {client}" -d '{ "id": 0, "category": { "id": 0, "name": "string" }, "name": "doggie", "photoUrls": ["string"], "tags": [{ "id": 0, "name": "string" }], "status": "available"}'

Service response


```
{
  "id": 5657082955040122,
  "category": {
    "id": 0,
    "name": "string"
  },
  "name": "doggie",
  "photoUrls": [
    "string"
  ],
  "tags": [
    {
      "id": 0,
      "name": "string"
    }
  ],
```

"status": "available"
}

Service response code

200

Service response headers

Date: Thu, 21 Mar 2019 12:49:38 GMT
X-GovStack-id: dcaaa3a2-a158-41e1-8775-309848052358
X-GovStack-client: DEV/COM/222/TESTCLIENT
X-GovStack-service: DEV/COM/222/TESTSERVICE/petstore
X-GovStack-request-id: f92591a3-6bf0-49b1-987b-0dd78c034cc3
X-GovStack-request-hash:
VCNZdwTxl7m3XC6Mpfw1H6qJUtBcm3Y6tfCvg5b3W/fb2RRXsLF9wftR3u6ElclE+RFaiAN/OkSz02
fAYbNKaw==
Content-Length: 0

DELETE Request and Response

REQUEST

Service	Method	Description	Parameters
/pets/{petId}	DELETE	Deletes a pet	petId - Pet id to delete

Service called directly

curl -X DELETE "https://petstore.niis.org/v2/pets/1124" -H "accept: application/json"

Service called through Service Access gateway

curl -X DELETE "https://{securityserver}/r1/{serviceId}/v2/pets/1124" -H "accept:
application/json" -H "X-GovStack-Client: {client}"

Service response

Service response code

200

Service response headers

Date: Thu, 21 Mar 2019 12:49:38 GMT
X-GovStack-id: 6209d61b-6ab5-4443-a09a-b8d2a7c491b2
X-GovStack-client: DEV/COM/222/TESTCLIENT
X-GovStack-service: DEV/COM/222/TESTSERVICE/petstore
X-GovStack-request-id: f92591a3-6bf0-49b1-987b-0dd78c034cc3
X-GovStack-request-hash:
lQBoldcyuI3BerjHfkleRQ45AyYoFlF7zXSN6yH/RwvTNWEcsTQM18EfqMxYfdkyGGB26oxAjAWv/A
cfmZF7og==

Content-Length: 0

POST Request with Attachments and Response

REQUEST

Service	Method	Description	Parameters
/pets/{petId}/images	POST	uploads an image	• petId - ID of pet to update • additionalMetadata - Additional data to pass to server • file - file to upload

Service called directly

curl -X POST "https://petstore.niis.org/v2/pets/1124/images" -H "accept: application/json" -H "Content-Type: multipart/form-data" -F "file=@A-fluffy-cat-looking-funny-surprised-or-concerned.jpg;type=image/jpeg"

Service called through Service Access gateway

curl -X POST "https://{securityserver}/r1/{serviceId}/v2/pets/1124/images" -H "accept: application/json" -H "Content-Type: multipart/form-data" -H "X-GovStack-client: {client}" -F "file=@A-fluffy-cat-looking-funny-surprised-or-concerned.jpg;type=image/jpeg"

Service response

{
  "code":200,
  "type":null,
  "message":"additionalMetadata: null\nFile uploaded to ./file, 170025 bytes"
}

Service response code

200

Service response headers

Content-Type: application/json;charset=utf-8
Date: Thu, 21 Mar 2019 13:02:29 GMT
X-GovStack-id: 86e081a6-ec16-4b8d-b729-963f9659a80c
X-GovStack-client: DEV/COM/222/TESTCLIENT
X-GovStack-service: DEV/COM/222/TESTSERVICE/petstore
X-GovStack-request-id: f92591a3-6bf0-49b1-987b-0dd78c034cc3
X-GovStack-request-hash: EyclkZAz4WMvbKgnBvd0wUcN4A4w0RZMvugD36ZJ2PpwwGZuMGfxCoO4C0ZC3c4LBGF0rh61vunL3ssZV6TB3Q==
Content-Length: 100

14.1.7 Directory Services (Meta Services)

14.1.7.1 Retrieving List of Service Providers

For retrieving the list of service providers listClients metaservice is used. It can be invoked with simple HTTP GET to right URL. Service output format is controlled with Accept header.

14.1.7.2 Retrieving List of Services

Service Access gateway provides two methods for getting the list of services offered by an Service Access gateway client:

- listMethods lists all REST services offered by a service provider.
- allowedMethods lists all REST services offered by a service provider that the caller has permission to invoke.

Both methods are invoked as regular Service Access gateway REST services.

The serviceId MUST contain the identifier of the target service provider and the value of the serviceCode element MUST be either listMethods or allowedMethods.

Request example

GET /r1/INSTANCE/CLASS2/MEMBER2/APPLICATION2/listMethods

HTTP request headers

X-GovStack-Client: INSTANCE/CLASS1/MEMBER1/APPLICATION1

The body of the response message MUST contain a list of services provided by the service provider (in case of listMethods) or open to the given client (in case of allowedMethods). The response SHALL NOT contain names of the metainfo services.

Annex B contains the OpenAPI description of the REST metadata services.

Annexes C.1 and C.2 contain example response messages for services, respectively.

14.1.7.3 Retrieving the OpenAPI Description of a Service

Service Access gateway provides a metaservice for fetching service descriptions of REST services.

- getOpenAPI returns the OpenAPI service description of a REST service

The method is invoked as regular Service Access gateway REST service.

The serviceId MUST contain the identifier of the target service provider and the value of the serviceCode element MUST be getOpenAPI.

The query parameters must contain serviceCode=xxx where xxx is the service code of the REST service we want to get the service description from.

Request example

GET /r1/INSTANCE/CLASS2/MEMBER2/APPLICATION2/getOpenAPI?serviceCode=listFirms

HTTP request headers

X-GovStack-Client: INSTANCE/CLASS1/MEMBER1/APPLICATION1

The body of the response MUST contain the OpenAPI service description of the REST service indicated by the query parameters.

Annex B contains the OpenAPI description of the REST metadata services.

Annex C.3 contains an example response message for the service.

14.2 Annex A Service Description

```
openapi: 3.0.0
info:
  description: >-
    This is a sample server Petstore server.
  version: 1.0.0
  title: Petstore
  contact:
    email: info@niis.org
  license:
    name: Apache 2.0
    url: 'http://www.apache.org/licenses/LICENSE-2.0.html'
tags:
  - name: pet
    description: Everything about your Pets
    externalDocs:
      description: Find out more
      url: 'https://niis.org'
  - name: store
    description: Access to Petstore orders
  - name: user
    description: Operations about user
    externalDocs:
      description: Find out more about our store
      url: 'https://niis.org'
paths:
  /pets:
    get:
      tags:
        - pet
```

```
    summary: Get pets from store
    description: Search pets
    operationId: getPets
    parameters:
      - name: term
        in: query
        description: search term
        required: false
        schema:
          type: string
    responses:
      '200':
        description: successful operation
        content:
          application/xml:
            schema:
              type: array
              items:
                $ref: '#/components/schemas/Pet'
          application/json:
            schema:
              type: array
              items:
                $ref: '#/components/schemas/Pet'
      '400':
        description: Invalid ID supplied
      '404':
        description: Pet not found
    security:
      - api_key: []
  post:
    tags:
      - pet
    summary: Add a new pet to the store
    description: ''
    operationId: addPet
    responses:
      '201':
        description: pet created
      '405':
        description: Invalid input
    security:
      - petstore_auth:
          - 'write:pets'
          - 'read:pets'
    requestBody:
      $ref: '#/components/requestBodies/Pet'
'/pets/{petId}':
  get:
```

```
    tags:
      - pet
    summary: Find pet by ID
    description: Returns a single pet
    operationId: getPetById
    parameters:
      - name: petId
        in: path
        description: ID of pet to return
        required: true
        schema:
          type: integer
          format: int64
    responses:
      '200':
        description: successful operation
        content:
          application/xml:
            schema:
              $ref: '#/components/schemas/Pet'
          application/json:
            schema:
              $ref: '#/components/schemas/Pet'
      '400':
        description: Invalid ID supplied
      '404':
        description: Pet not found
    security:
      - api_key: []
  put:
    tags:
      - pet
    summary: Update an existing pet
    description: ''
    operationId: updatePet
    parameters:
      - name: petId
        in: path
        description: ID of pet to return
        required: true
        schema:
          type: integer
          format: int64
    responses:
      '200':
        description: Pet updated
      '400':
        description: Invalid ID supplied
      '404':
```

```
        description: Pet not found
      '405':
        description: Validation exception
    security:
      - petstore_auth:
          - 'write:pets'
          - 'read:pets'
    requestBody:
      $ref: '#/components/requestBodies/Pet'
  delete:
    tags:
      - pet
    summary: Deletes a pet
    description: ''
    operationId: deletePet
    parameters:
      - name: api_key
        in: header
        required: false
        schema:
          type: string
      - name: petId
        in: path
        description: Pet id to delete
        required: true
        schema:
          type: integer
          format: int64
    responses:
      '200':
        description: Pet deleted
      '400':
        description: Invalid ID supplied
      '404':
        description: Pet not found
    security:
      - petstore_auth:
          - 'write:pets'
          - 'read:pets'
'/pets/{petId}/images':
  post:
    tags:
      - pet
    summary: Uploads an image
    description: ''
    operationId: uploadFile
    parameters:
      - name: petId
        in: path
```

```
        description: ID of pet to update
        required: true
        schema:
          type: integer
          format: int64
    responses:
      '200':
        description: successful operation
        content:
          application/json:
            schema:
              $ref: '#/components/schemas/ApiResponse'
    security:
      - petstore_auth:
          - 'write:pets'
          - 'read:pets'
    requestBody:
      content:
        multipart/form-data:
          schema:
            type: object
            properties:
              additionalMetadata:
                description: Additional data to pass to server
                type: string
              file:
                description: file to upload
                type: string
                format: binary
/store/inventories:
  get:
    tags:
      - store
    summary: Returns pet inventories by status
    description: Returns a map of status codes to quantities
    operationId: getInventory
    responses:
      '200':
        description: successful operation
        content:
          application/json:
            schema:
              type: object
              additionalProperties:
                type: integer
                format: int32
    security:
      - api_key: []
/store/orders:
```

```
  post:
    tags:
      - store
    summary: Place an order for a pet
    description: ''
    operationId: placeOrder
    responses:
      '200':
        description: successful operation
        content:
          application/xml:
            schema:
              $ref: '#/components/schemas/Order'
          application/json:
            schema:
              $ref: '#/components/schemas/Order'
      '400':
        description: Invalid Order
    requestBody:
      content:
        application/json:
          schema:
            $ref: '#/components/schemas/Order'
      description: order placed for purchasing the pet
      required: true
'/store/orders/{orderId}':
  get:
    tags:
      - store
    summary: Find purchase order by ID
    description: >-
      For valid response try integer IDs with value >= 1 and <= 10.
      Other values will generated exceptions
    operationId: getOrderById
    parameters:
      - name: orderId
        in: path
        description: ID of pet that needs to be fetched
        required: true
        schema:
          type: integer
          format: int64
          minimum: 1
          maximum: 10
    responses:
      '200':
        description: successful operation
        content:
          application/xml:
```

```
            schema:
              $ref: '#/components/schemas/Order'
          application/json:
            schema:
              $ref: '#/components/schemas/Order'
      '400':
        description: Invalid ID supplied
      '404':
        description: Order not found
  delete:
    tags:
      - store
    summary: Delete purchase order by ID
    description: >-
      For valid response try integer IDs with positive integer value.
      Negative or non-integer values will generate API errors
    operationId: deleteOrder
    parameters:
      - name: orderId
        in: path
        description: ID of the order that needs to be deleted
        required: true
        schema:
          type: integer
          format: int64
          minimum: 1
    responses:
      '200':
        description: Purchase order deleted
      '400':
        description: Invalid ID supplied
      '404':
        description: Order not found
/users:
  post:
    tags:
      - user
    summary: Create user
    description: This can only be done by the logged in user.
    operationId: createUser
    responses:
      default:
        description: successful operation
    requestBody:
      content:
        application/json:
          schema:
            $ref: '#/components/schemas/User'
      description: Created user object
```

```
      required: true
/users/login:
  get:
    tags:
      - user
    summary: Logs user into the system
    description: ''
    operationId: loginUser
    parameters:
      - name: username
        in: query
        description: The user name for login
        required: true
        schema:
          type: string
      - name: password
        in: query
        description: The password for login in clear text
        required: true
        schema:
          type: string
    responses:
      '200':
        description: successful operation
        headers:
          X-Rate-Limit:
            description: calls per hour allowed by the user
            schema:
              type: integer
              format: int32
          X-Expires-After:
            description: date in UTC when token expires
            schema:
              type: string
              format: date-time
        content:
          application/xml:
            schema:
              type: string
          application/json:
            schema:
              type: string
      '400':
        description: Invalid username/password supplied
/users/logout:
  get:
    tags:
      - user
    summary: Logs out current logged in user session
```

```
      description: ''
      operationId: logoutUser
      responses:
        default:
          description: successful operation
  '/users/{username}':
    get:
      tags:
        - user
      summary: Get user by user name
      description: ''
      operationId: getUserByName
      parameters:
        - name: username
          in: path
          description: 'The name that needs to be fetched. Use user1 for
testing. '
          required: true
          schema:
            type: string
      responses:
        '200':
          description: successful operation
          content:
            application/xml:
              schema:
                $ref: '#/components/schemas/User'
            application/json:
              schema:
                $ref: '#/components/schemas/User'
        '400':
          description: Invalid username supplied
        '404':
          description: User not found
    put:
      tags:
        - user
      summary: Updated user
      description: This can only be done by the logged in user.
      operationId: updateUser
      parameters:
        - name: username
          in: path
          description: name that need to be updated
          required: true
          schema:
            type: string
      responses:
        '200':
```

```
          description: User updated
        '400':
          description: Invalid user supplied
        '404':
          description: User not found
      requestBody:
        content:
          application/json:
            schema:
              $ref: '#/components/schemas/User'
        description: Updated user object
        required: true
    delete:
      tags:
        - user
      summary: Delete user
      description: This can only be done by the logged in user.
      operationId: deleteUser
      parameters:
        - name: username
          in: path
          description: The name that needs to be deleted
          required: true
          schema:
            type: string
      responses:
        '200':
          description: User deleted
        '400':
          description: Invalid username supplied
        '404':
          description: User not found
externalDocs:
  description: Find out more
  url: 'https://niis.org'
servers:
  - url: 'https://petstore.niis.org/v2'
  - url: 'http://petstore.niis.org/v2'
components:
  requestBodies:
    UserArray:
      content:
        application/json:
          schema:
            type: array
            items:
              $ref: '#/components/schemas/User'
      description: List of user object
      required: true
```

```
    Pet:
      content:
        application/json:
          schema:
            $ref: '#/components/schemas/Pet'
        application/xml:
          schema:
            $ref: '#/components/schemas/Pet'
      description: Pet object that needs to be added to the store
      required: true
  securitySchemes:
    petstore_auth:
      type: oauth2
      flows:
        implicit:
          authorizationUrl: 'http://petstore.niis.org/oauth/dialog'
          scopes:
            'write:pets': modify pets in your account
            'read:pets': read your pets
    api_key:
      type: apiKey
      name: api_key
      in: header
  schemas:
    Order:
      type: object
      properties:
        id:
          type: integer
          format: int64
        petId:
          type: integer
          format: int64
        quantity:
          type: integer
          format: int32
        shipDate:
          type: string
          format: date-time
        status:
          type: string
          description: Order Status
          enum:
            - placed
            - approved
            - delivered
        complete:
          type: boolean
          default: false
```

```
    xml:
      name: Order
  Category:
    type: object
    properties:
      id:
        type: integer
        format: int64
      name:
        type: string
    xml:
      name: Category
  User:
    type: object
    properties:
      id:
        type: integer
        format: int64
      username:
        type: string
      firstName:
        type: string
      lastName:
        type: string
      email:
        type: string
      password:
        type: string
      phone:
        type: string
      userStatus:
        type: integer
        format: int32
        description: User Status
    xml:
      name: User
  Tag:
    type: object
    properties:
      id:
        type: integer
        format: int64
      name:
        type: string
    xml:
      name: Tag
  Pet:
    type: object
    required:
```

```
      - name
      - photoUrls
    properties:
      id:
        type: integer
        format: int64
      category:
        $ref: '#/components/schemas/Category'
      name:
        type: string
        example: doggie
      photoUrls:
        type: array
        xml:
          name: photoUrl
          wrapped: true
        items:
          type: string
      tags:
        type: array
        xml:
          name: tag
          wrapped: true
        items:
          $ref: '#/components/schemas/Tag'
      status:
        type: string
        description: pet status in the store
        enum:
          - available
          - pending
          - sold
    xml:
      name: Pet
  ApiResponse:
    type: object
    properties:
      code:
        type: integer
        format: int32
      type:
        type: string
      message:
        type: string
```

14.3 Annex B Service Descriptions for REST Metadata Services

See https://app.swaggerhub.com/apis/GovStack/gov-stack_im_service_metadata_api/0.3

14.4 Annex C Example Meta Messages

14.4.1 listMethods Response

```
curl -H "accept: application/json" -H
"X-GovStack-Client:INSTANCE/CLASS1/MEMBER1/APPLICATION1"
"https://SECURITYSERVER:443/r1/INSTANCE/CLASS2/MEMBER2/APPLICATION2/listMethods"
{
    "service": [
        {
            "member_class": "CLASS2",
            "member_code": "MEMBER2",
            "object_type": "SERVICE",
            "service_code": "payloadgen",
            "application_code": "APPLICATION2",
            "GovStack_instance": "INSTANCE"
        },
        {
            "member_class": "CLASS2",
            "member_code": "MEMBER2",
            "object_type": "SERVICE",
            "service_code": "kore",
            "application_code": "APPLICATION2",
            "GovStack_instance": "INSTANCE"
        }
    ]
}
```

14.4.2 allowedMethods Response

```
curl -H "accept: application/json" -H
"X-GovStack-Client:INSTANCE/CLASS1/MEMBER1/APPLICATION1"
"https://SECURITYSERVER:443/r1/INSTANCE/CLASS2/MEMBER2/APPLICATION2/allowedMetho
ds"

{
    "service": [
        {
            "member_class": "CLASS2",
            "member_code": "MEMBER2",
            "object_type": "SERVICE",
```

```
            "service_code": "payloadgen",
            "application_code": "APPLICATION2",
            "GovStack_instance": "INSTANCE"
        }
    ]
}
```

14.4.3 getOpenAPI Response

```
curl -H "accept: application/json" -H
"X-GovStack-Client:INSTANCE/CLASS1/MEMBER1/APPLICATION1"
"https://SECURITYSERVER:443/r1/INSTANCE/CLASS2/MEMBER2/APPLICATION2/getOpenAPI?s
erviceCode=listFirms"

openapi: "3.0.0"
info:
  version: 1.0.0
  title: Firm listing
servers:
  - url: https://example.com
paths:
  /firms:
    get:
      summary: List all firms
      operationId: listFirms
      tags:
        - firms
      parameters:
        - name: limit
          in: query
          description: How many items to return at one time (max 100)
          required: false
          schema:
            type: integer
            format: int32
      responses:
        '200':
          description: A paged array of firms
          headers:
            x-next:
              description: A link to the next page of responses
              schema:
                type: string
          content:
            application/json:
              schema:
                $ref: "#/components/schemas/Firms"
        default:
```

```
          description: unexpected error
          content:
            application/json:
              schema:
                $ref: "#/components/schemas/Error"
components:
  schemas:
    Firm:
      required:
        - id
        - name
        - size
        - country
      properties:
        id:
          type: integer
          format: int64
        name:
          type: string
        tag:
          type: string
    Firms:
      type: array
      items:
        $ref: "#/components/schemas/Firm"
    Error:
      required:
        - code
        - message
      properties:
        code:
          type: integer
          format: int32
        message:
          type: string
```