Ministry of Housing and Urban Affairs
Government of India

Centre for Digital Governance

Citizen Centric Smart Governance (CCSG)

Specifications, Guidelines and Recommendations for Designing Data Models and APIs - V 2.0

Draft document for discussion

Smart Cities Mission
Ministry of Housing and Urban Affairs (MoHUA)

And

Centre for Digital Governance (CDG)
National Institute of Urban Affairs (NIUA),
New Delhi

November 2020

Specifications, Guidelines and Recommendations for Designing Data Models and APIs

Introduction

This document derives from the NUIS Strategy and Approach Paper and CCSG Open Standards – Strategy and Approach v2.0 and aims to provide granular details of designing CCSG standards.

To achieve maximum mileage from the initiative in a short period, while maintaining the evolvability of standards in a manner that future enhancements are easy to perform, it is necessary to leverage existing tools, techniques and methodologies. The sections below aim to prioritise the key areas of standards creation, lay out the approach to design, and recommend key tools, techniques, and methods to be adopted by CCSG standards creation group.

As categorised in the parent document - CCSG Open Standards – Strategy and Approach v2.0, potential focus areas are:

- Knowledge and Process standards
- Technology standards (Software/Hardware/Data)

This document focuses in detail on guidance for two major parts of **Technology Standards** - **Data Models and APIs**. It describes technology standards at two levels:

- abstract specifications, which translate the parentguidelines into conceptual rules
- design specifications, which further translate the conceptual rules into recommendations, requirements, and prohibitions for Data Model and API designers

Abstract Specifications - Data Models and APIs

This section aims to contextualize and extend the NUIS' key guiding principles described in the NUIS Strategy and Approach paper to the design of Data Models and APIs for CCSG domains. Below is the list of guidelines that Data Models and APIs should follow:

Open

In order to maintain technology and vendor neutrality, Data Models and APIs -

- Should be published under most unrestricted license (Creative Commons:CC0, BY and BY-SA, or MIT)
- Should not assume/require choice of any proprietary technology
- Should be designed and ratified through a multi-stakeholder process
- Should be developed/reviewed/adopted by a group of experts through a consensus-based process

Evolvable

To adapt to changing needs over time, Data Models and APIs -

- Should be versioned with backward compatibility of at least up to 1 major version. The Domain Working Groups (DWGs) should strive to release one major version upgrade every year.
- Should be protocol agnostic, thereby allowing for innovation in protocols, and also support solution use cases over multiple protocols (e.g. governance application over chat interfaces). In other words they should follow The Rule of Least Power[1]. In order to achieve this, Data models and APIs should carry all the necessary information –
    - request metadata e.g. authentication token, device information, signatures
    - response metadata e.g. response signatures, processing status, correlations ids
    - error data e.g. error codes, message etc.
- in its own structures, rather than depending on protocol-specific fields/headers to carry such information.

Extendable

Given the diversity of India's urban systems, it is important to ensure that standards do not limit the ability of solution providers to develop solutions that meet local needs. Further to enable innovation, standards should not be restrictive in their specification and application. Thus

[1]When designing computer systems, one is faced with a choice between using a more or less powerful language for publishing information, for expressing constraints, or for solving some problem. The "Rule of Least Power" suggests choosing the least powerful language suitable for a given purpose. This increases flexibility: the less powerful the language, the more you can do with data stored in that language. For additional information, see https://www.w3.org/2001/tag/doc/leastPower.html

standards need to be extendable to enable ecosystem actors to innovate and build locally relevant solutions. Therefore, Data Models and APIs-

- Should extend from existing international/national standards like National Municipal Accounting Manual (NMAM) wherever possible
- Should allow adding optional business extension elements
- Such extension elements should be clearly documented in the same manner as base models and APIs and made available on a public repository

Minimalistic

To enable ecosystem actors to easily adopt standards while empowering them to innovate, Data Models and APIs -

- Should have minimal mandatory fields in data models
- Should require only most fundamental API operations to enable faster compliance while fulfilling needed functional requirements. Consistent pattern in API operations on various entities makes them simple to adopt.
- Should avoid including attributes/APIs needed for specific solutions that are not yet known to be applicable to wider solutions.

Balance Data Privacy with Data Empowerment

To leverage the power of data while ensuring safe usage, Data Models and APIs -

- Should require minimal personally identifiable information (PII) to be collected mandatorily thereby reducing risk to PII data
- Should provide policy based access control to enable creation, modification and sharing of data as needed
- Should include provision for data anonymization and proxy fields for PII and other sensitive data

Provide for non-repudiability

To ensure right attribution for the data, Data Models and APIs -

- Should declare access mechanisms for APIs
- Should provide for digital signatures
- Should provide APIs for accessing data access information

Unbundled

To provide the most fundamental building blocks while ensuring minimalism, extensibility and evolvability, Data Models and APIs -

- Should limit the mandatory information in data models

- Should require minimal foundational APIs. API standards should not mandate composite APIs (ones that can be decomposed into foundational ones)

Design Specifications - Data Models and APIs

This section provides recommendations, rules, and prohibitions for Data Model and API designers, so as to ensure that the data models and APIs they create are compliant with the guiding principles for CCSG Open Standards – Strategy and Approach Paper. The sub-sections below can be mapped on to one or more of the guiding principles; collectively, they ensure that any Data Model and API created as part of the CCSG platform will be smoothly interoperable with the platform as a whole, and in compliance with the guiding principles.

As these sub-sections are written primarily as guidance for Data Model and API designers, a basic level of technical knowledge on the part of the reader is presumed. Readers may refer to the Glossary in the appendix for brief explanations of technical terms.

Given the complexity of various aspects of the urban domain, a few additional rules and requirements - i.e. in addition to the meta-standard - are required to manage Data Models and APIs in keeping with the CCSG open standards. These specifications (rules and requirements) are provided below:

Data Models

1. Data Models should be broken down to simpler fundamental units (models) as far as possible. E.g. A property assessment model is basically Property model and Assessment model with assessment model referring to property model.
2. Data models should include a Universally Unique Identifier (UUID) field which should uniquely identify the object of respective entity type within the respective domain.
3. Data models should require minimal mandatory fields to enable maximal inclusion. As a thumb rule, wherever unsure whether a field is absolutely required in all scenarios, it should be made optional.
4. Data models should extend/ reuse/ adopt international/ national models wherever available/applicable e.g. Open311 for citizen services like grievances and schema.org for general model definitions.
5. Data models should be extensible i.e. they should allow a way to capture extra information that was not initially included during the model design. To achieve this:
    - It may provide a simple map of key-value pairs. Future versions of the data model may choose to create mandatory/optional names attributes in the data models after researching wider applicability of such fields.
    - Data Models should allow for namespacing in field names to indicate the source/reason/category of the extended fields.

6. It is recommended that all timestamps be captured in Unix epoch time. APIs may define display format property to indicate the human readable format most suitable for display.

Multitenancy

Given the variety of organizational contexts in the urban domain, APIs SHOULD specify the organization context in which they are functioning. Knowledge of this information helps in providing relevant controls over creation, modification and access to the data thereby increasing trust in the data fidelity.

To achieve this, all APIs should support multi-tenancy by including tenant id (id for the organization) in their data models. To address the variety of tenant contexts e.g. geographical (ULB, State, regions, zones), functional (departments, sub departments), tenant id should use the concept of namespaces to uniquely identify the organizational unit without any ambiguity. Web addresses on the internet are good examples of such namespace references to the resources on the web. For the purpose of urban APIs, the following namespace pattern is proposed- [**function@]geographicalunit[/resource]**, where

1. **Function** is the department within urban context, e.g. municipal, electricity, etc.
2. **Geographical unit** is namespaced reference to the urban geography, e.g. in.pb.chandigarh (basically **country.state.city**).
3. **Resource** is a resource or sub function with main function is 1, e.g. property tax, or water charges within municipal function.

[ ] indicates that these components are optional. E.g. an API dealing with only municipal context may not need the function/department name to be specified, similarly resource may be deduced from the model context.

Data Modification

In order to keep APIs simple and allow rebundling in various contexts, APIs should be atomic and support most fundamental operations. In order to achieve this -

1. All registries should minimally support following operations -
    a. Create - create a new entity
    b. Update - update an entity, this can include complete replacement as well update of selected fields
    c. Delete - to delete the entity from the system. This operation irrecoverably removes the entity from the system hence special care should be applied in using and implementing this API.
    d. Cancel/Deactivate - to mark an entity as inactive in the systems.
2. All data modification APIs should clearly describe the access control roles and policies (Policy may include the needed workflow information helping in scenarios like consent based data modification). Access/Modification of roles and policies themselves should be access controlled and audited.

3. All data modification API access should require information needed for auditing the operation, including
    a. Who accessed the API, under what access role
    b. When was it accessed
    c. What modifications were performed
4. Audit information should be preserved for the retention period required by the operator's policies. While different operators may choose different retention periods for the audit information, it is recommended that it is at least one year to provide enough window to settle any queries.
5. In case above registry operations do not cover fundamental operations for an entity, providers should clearly mention the business/domain use case and provide additional APIs.
6. APIs should support asynchronous data modification to support application responsiveness and performance.

Data Search

As one of the most basic and most frequent operations, data search APIs need to be responsive, robust, secure and respect privacy while enabling needed functionality. In order to achieve these search APIs -

1. Search APIs should declare the access control mechanism and role.
2. All search should be audited (Who, when, which API, what criteria) to help with
    a. Improving the product - observability
    b. Audit for illegal uses
    c. Audit for access to PII data whenever demanded in clear
3. Search Criteria - should provide search filter capabilities on-
    a. UUIDs and Unique field combinations of the entities
    b. Entity creation and modification timestamps
    c. Needed directed searches applicable to the domain. This is to prevent bulk access to the data in order to overcome lack of search facility on crucial fields. E.g. Property Tax may need a use case to search the Property and Assessments based on mobile number of the property owner. In absence of providing mobile number as the criteria system may be misused to download all properties with their owner information (Also consider point 2 below for such bulk search needs) - hence such misuse should be avoided by providing directed searches on key parameters.
4. Search Results -
    a. Primary functional role of search APIs is to help the consumer achieve the goal faster while balancing between data access and data privacy/security. In order to achieve this -

i. Search APIs should limit the number of records returned in the search to make searches secure and respect data privacy. While the number of records to return may be configurable it is recommended that API do not return more than 10 records.
ii. Search results should by default Mask or Anonymize or Encrypt the PII or other sensitive data. Access to clear PII data should be provided on individual record basis and under additional access control.
iii. Open Search APIs should never contain PII data in clear.

5. Asynchronous search support - APIs should support asynchronous search capabilities wherein search results are generated and accessed asynchronously. Asynchronous search capabilities can help use cases like
    a. On behalf of search - search is initiated by role X, but the results are delivered to/accessible to role Y
    b. Workflow driven search - Search request is approved by another role before results are accessible/delivered to the originator of the request.
6. Source Field Filter - Search APIs should provide mechanisms to return only needed fields in the response to help save network bandwidth. Without this filter by default APIs should return all fields of its model.
7. Wherever applicable APIs should offer search on the Combination of structured and unstructured data
8. Search APIs should also enable searching audit history of -
    a. Changes to the Data
    b. Searches Audits (as mentioned in point 2 in this section)

Transport Protocol Agnostic Behavior

In order to support a wide variety of use cases the APIs should be protocol agnostic, i.e. they should not depend on communication protocol specific mechanisms (e.g. HTTP headers) to carry needed information. In particular:

1. APIs should not depend on protocol specific verbs to signal operations (e.g. PUT, POST, GET). In order to satisfy this, it is recommended that API specification use the following path scheme to indicate fundamental operations -
    a. Create - /<uri>/_create
    b. Update - /<uri>/_update
    c. Delete - /<uri>/_delete
    d. Cancel/Deactivate - /<uri>/_cancel
    e. Search - /<uri>/_search

    <uri> here refers to the Uniform Resource Identifier as described in RFC 3986 and _ (underscore) is used as a mechanism to indicate the operation name. This path scheme can be extended further for additional operations an API may need to define.

2. Protocol specific layer can be offered as a plug over the protocol agnostic microservice body
3. API requests should always include RequestHeader along with the API's business details in its body. Following should be the main constituents of RequestHeader -
    a. Authentication and Authorization details, e.g. JWT Auth Token, API Key
    b. Client generated Request message Id
    c. Device Details - device id, device signature, device type etc
    d. API info - version, id, path etc
    e. Request Signature (Optional)
    f. Additional Info (Optional) e.g. callback urls, channel, source
4. A Response should always contain ResponseHeader along with the API's business details in its body. Following should be the main constituents of ResponseHeader -
    a. Client's Request Message id
    b. Response Message id from Server
    c. Processing Status
        i. Completed
        ii. Accepted (in case of asynchronous processing)
        iii. Failed
    b. Response Signature (Optional)
    c. Error (Optional) - Error details from the API, actual response object may not be returned
    d. Information (Optional) - Additional information from the API that may be utilized to further optimize the interaction.
    e. Debug (Optional) - Debug information when requested by client.
    f. Additional Info (Optional) e.g. status url (to find out the current status of an asynchronous processing response), additional links to perform special functions like file uploads etc.
5. In case of an Error in handling the operation, API should return the error status in ResponseHeader along with additional error details in the Error object.
6. API standards should allow indicating the need to access debug info in case of error. In which case the ResponseHeader should additionally carry Debug objects with debuggable information of the error. To ensure security and performance, servers should implement this under a policy based control.

File Handling

1. File upload/download should also be handled via declared APIs
2. Should use secure file transmission mechanism (like HTTPS) with authentication controls
3. Should support access control and audit mechanisms for storage as well as file access

4. Should provide for file validation mechanisms like file type validations, file size validations etc.

Bulk Data Handling

Bulk Data handling - Most modern systems allow bulk creation/ updation through asynchronous batch processes that leverage existing APIs to handle the create/ update. Therefore,

1. It is recommended that bulk data is handled using implementation specific asynchronous batch layers on top of standard Data Model/APIs designs.
2. Bulk data operations should be able to provide access control and workflow mechanisms of their own in addition to the underlying API layer

API evolution and Versioning

In an evolving system like CCSG standards it is necessary to mark the APIs with semantic versioning to help solution providers implement and publish their compliance for interoperability.

Following are the main strategies to handle API evolutions and required backward compatibility

1. Evolution APIs and Data Models should support N-1 major version backward compatibility. To achieve this
    a. Inclusion of new Mandatory fields should always come with default values thereby not failing the API consumers that have not been upgraded to new version.
    b. Removal of existing mandatory fields should be performed by moving the field to optional in the new version and then optionally removed in further major versions.
2. API path should include API version thereby allowing multiple APIs versions to co-exist in the same time period. Older API paths that are really required to be sunset can return relevant error messages indicating consumers to move to higher versions.
3. Version numbers of APIs should be specified in API version as per OpenAPI 3.0 specifications. OpenAPI 3.0 uses SemVer 2.0 as its versioning scheme. SemVer is a simple set of rules and requirements aka specification that dictates how version numbers are assigned and incremented.

Recommended tools and techniques

Meta-Standard for Specifying Data Model and API Standard

CCSG standards team recommends that APIs are defined using OpenAPI 3.0 Specification. OpenAPI Specification (formerly Swagger Specification) is language-agnostic API description format for APIs. An OpenAPI specification file allows one to describe an entire API, including:

- Data model (entities)

- Available operations on the resources - service endpoints. These can be accompanied with detailed descriptions and usage information.
- Input and output for each operation
- Authentication and authorization information
- Contact information, versions, license, terms of use and other information

OpenAPI allows APIs to describe their own structure, providing the needed flexibility to model complex urban domains.

API specifications in OpenAPI 3.0 are written in YAML or JSON - both these formats are easy to learn and readable to both humans and machines. When properly defined, a consumer can understand and interact with the APIs with a minimal knowledge about their implementation logic.

Recommended tool for using OpenAPI 3.0 - Swagger

OpenAPI specifications also have a set of open source tools built around them called Swagger that help with designing, building, documenting and consuming these APIs thus making it easy for providers to adopt. The major swagger tools include:

- Swagger Editor – browser-based editor where you can write OpenAPI specs.
- Swagger UI – renders OpenAPI specs as interactive API documentation.
- Swagger Codegen – generates server stubs and client libraries from an OpenAPI spec.

OpenAPI and Swagger also have a vibrant ecosystem of other open source and commercial tools that can help in faster adoption of APIs.

Collaboration and Maintenance

Data models and API definitions for a vast and complex domain like CCSG will evolve over a period of time and will require multiple ecosystem actors to collaborate at various point of time. Keeping this in mind they need a modern, auditable and trackable way of working together and versioning which can facilitate various roles like contributor, reviewer etc. In other words, it needs a modern distributed collaboration tool like Open Source developers use.

Git is a free and open source distributed version control system designed to handle projects that require collaboration among diverse and distributed teams. This makes Git ideal for managing collaborative exercises like CCSG API standard development.

GitHub is one of the largest git service providers that is used by developers worldwide to host and manage open source (and private/non open source) projects. Github provides free access for open source projects and a wide variety of tools and interfaces to help non-technical people to leverage the versioning and collaboration capabilities thereby making it highly amenable to manage CCSG standards development. It will also make it more easy and manageable for solution providers to access and develop against various versions of CCSG API standards.

Glossary

1. Anonymisation
2. Application Programming Interface (API)
3. Asynchronous
4. Authentication / Authentication Tokens
5. Backwards compatibility
6. Circular referencing
7. Count
8. Create (and Delete)
9. Creative Commons (License)
10. Data Model
11. Debug
12. Error
13. Git
14. Header
15. HTML
16. Java
17. JSON (and YAML)
18. Metadata
19. MIT (License)
20. OpenAPI
21. Personally Identifiable Information
22. Protocol (and Protocol-Agnostic)
23. Request
24. Response
25. Search
26. Standards Development Group
27. Swagger
28. Tenant / Tenancy
29. Unix Epoch Time
30. UUID
31. Version / Versioning