API Design Guidelines and Standards

Timothy S. Hilgenberg

Version 0.9.0, August, 2022

Table of Contents

Colophon

The purpose of this document is to provide API designers and developers with guidance on how to design and build high quality, domain driven, easy to consumer API. These same principals should be used during the governance process to insure consistency in design and the adherence to standards.

This book is a very early draft of this manuscript

This book is being written in AsciiDoc using Visual Studio Code. I'm just learning this tool, so I apologize in advance for any bad formatting issues with early versions of the book.

All comments welcome

Early Working Draft Version 0.9.0 — June 2022

Overview

What is an API?

The term API is an abbreviation for Application Programming Interface.

According to Wikipedia:
Application Programming Interface (API) is a set of subroutine definitions, protocols, and tools for building application software. In general terms, it is a set of clearly defined methods of communication between various software components. A good API makes it easier to develop a computer program by providing all the building blocks, which are then put together by the programmer.

The term API and service are often used interchangeably. A service is typically some action, retrieving or updating data, against a business entity or object. There are many types of services or actions that could be performed in applications. Using HR as an example domain, here are some examples:

- Getting domain related data → Person Health Coverage or Person 401k Balances
- Executing a business process or transaction → Health Care Annual Enrollment
- Supporting page marshalling and rendering > Fund Transfer Page marshalling
- Providing translation support for links and text > Building a page

The design standard is to follow an API First philosophy:

- Everyone MUST follow the API First principle.
- The API first principle is an extension of contract-first principle. Therefore, a development of an API MUST always start with API design without any upfront coding activities.
- API design specification (e.g., description, schema) is the master of truth, not the API implementation.
- API implementation MUST always be compliant to particular API design which represents the contract between API, and it's consumer.
- Approved API Design, represented by its API Description or schema, MUST represent the contract between API stakeholder, implementers, and consumers.
- An update to the corresponding contract (API Design) MUST be implemented and approved before any change to an API MUST.

At the highest level, API can be separated by types. These types would be:

Domain	Representing domain business objects without any display attributes. These are considered **facts** about the entities without regard to how they are displayed. In an MVC world, a pure model. Example: /person
Channel	Representing a channel (mobile, voice, web) component, which could be a mashup of domain **facts** plus display related fields. Example: /personDashboard, /someWidget, /textResolver
Command	Representing a command or action that doesn't have a true domain object or 'bounded context' associated with it. Example: /startSessionForPerson, /loginPerson

The scope of the service will generally be centered on the major business entities. From a design perspective, we will be following a design approach called Domain Driven Design described by Eric Evans. The goal is to create services around the major business entities, organized by "bounded contexts"

The unit of deployment for a service is the combination of URL routes and HTTP verbs that are logically associated together and will be:

- All the URL path routes associated with the service
- All the HTTP verbs need to act on the resources (GET, PUT, POST)

The goal is to keep the service scope to do one thing well. This allows the service to be easily reusable and deployable. In general, the scope of the service should center on a business entity. However, it is acceptable to have multiple logical service implementations deployed into a single executable.

OpenAPI Specification

Every API MUST be described using an API description format. The API description format used MUST be the OpenAPI Specification (formerly known as Swagger Specification) version 3.0. Every API description MUST be approved by API Governance and published in an appropriate API design repository. The spec SHOULD also be stored in version control system in the same repository as the API implementation. A code repository will not be created unless it has been approved by API Governance. _Any time a new URL and HTTP method is designed, it SHOULD go through the governance process for review.

Quotes

*An API signature should read like the **elevator speech** of the operations function/feature/purpose - should be domain related, not a technical implementation. It can be*

read from right to left to make sure it makes good sense - Tim Hilgenberg

APIs should be easy to use and hard to misuse. It should be easy to do simple things, possible to do complex things; and impossible, or at least difficult, to do wrong things - Joshua Bloch

Keep APIs free of implementation details. They confuse users and inhibit flexibility to evolve. It isn't always obvious what an implementation detail. Be wary of overspecification - Joshua Bloch

HTTP API General Design Standards

The following section provides the design standards for creating an HTTP REST based API. These are the more technical specifications for HTTP related API and are more implementation related. These can apply to any domain and are specific to HTTP. There is a separate section below that discusses the guidelines related to designing good domain API.

HTTP Methods/Verbs

REST API follow the standard HTTP methods (GET, PUT, POST, DELETE) used in general web site browser□s implementations. Using this model, it provides a level of connectedness, simplicity and standardization followed in the HTTP web environment.

GET SHOULD be used to retrieve data from the resource data store for inquiries. If the business object is not being mutated, this verb should be used. This is similar to a SQL SELECT.

- There may be situations when the API request data is large and complex. If this case the POST method (overloaded POST) may be appropriate. In this case one would put the verb **get** prepended to the resource name. Example: POST /api/<Resource>/get
- If this occurs, the designer should explore all possibilities to reduce the complexity of the request data.
- Personal Identification Information (PII) data should never be path variable on the URL. If PII data, like a person identifier, is required input into a GET call, a POST method should be considered. In this case, the data is placed in the request and would then be encrypted.
- *Add note about **meta** and add section later*

POST SHOULD be used to create new resources in the data store. As mentioned above, there may be special circumstance where a GET is implemented as a POST

PUT SHOULD be used to update resources in the data store. There may be situations where the PUT may not be sufficient to describe the update action that needs to take place. This will probably occur in business process update situations, where an additional □action□ needs to be provided. In this case, the additional **action** can be part of the request body in the PUT or a path variable If a path variable, the action would be placed at the end of the path

DELETE SHOULD be used to delete a resource in the data store.

PATCH is not used to do partial resource updates. PUT should be used to update a resource (Note: This may need to be revisited if PATCH becomes more popular for partial updates)

OPTION and **HEAD** are not used in the API

API Naming Conventions

General Naming Rules

- Use American English
- Don't use acronyms
- Stay away from abbreviations
- Avoid generic terms like util, data, info, or text as part of any name
  - Be specific as possible and prefix with the entity name (personId, widgetText)
  - Words like *common* or *util* should be avoided
- Use camelCase not snakeCase ('_') or dashes ('-'') unless stated otherwise
  - Dashes between words is not permitted unless approved - see below for permitted uses of dashes in names

Qualities of a good name

- A good name is concise. A good name need not be the shortest it can possibly be, but a good name should waste no space on things which are extraneous. Good names have a high signal to noise ratio.
- A good name is descriptive. A good name should describe the application of a variable or constant,*not* their contents. A good name should describe the result of an API, or behavior of a REST Operation, *not* its implementation. A good name should describe the purpose of a API, *not* its contents. The more accurately a name describes the thing it identifies, the better the name.
- A good name should be predictable. You should be able to infer the way a service will be used from its name alone. This is a function of choosing descriptive names, but it also about following tradition.

Naming Proverbs/Quotes

- *Poor naming is symptomatic of poor design* - Dave Cheney
- *Good naming is like a good joke, if you have to explain it, it's not funny* - Dave Cheney
- *Names matter - Strive of intelligibility, consistency, and symmetry. Every API is a little*

> *language, and people must learn to read and write it. If you get an API right, code will read like prose.* - Joshua Bloch

URI Design Standards

URI Standard Components

The following components make up a valid URI for calling the REST service:

Table 1. URI Component Table

Component	Description
Server endpoint	Generally, the Gateway URL. For gateway implementations, this is the API gateway server endpoint
'api'	Constant
Major Version Number ('V2' or 'V3')	If omitted, then it is considered Version 1. This is only the major version of the API. Minor versions are not permitted in the URI
'channel' (Optional)	Constant If the API is categorized as a 'channel' API
Service Name	This is the name of the implementation service. * For container based environments, like Docker, this is the container(docker) service name * For the container based environments, the recommendation for the service name is it MUST be all lower case * It is ok to treat this name as singular. For example, '/enrollment/enrollments' is permissible.

Component	Description
REST Resource	For Domain and Channel API, Pair combinations consisting of: * **Domain Entity** - should be plural * **Primary Key** - used to identify a specific resource. If omitted, the entire collection is being considered and not an individual business object based on that key. This primarily for GET calls For command API: **Command name** See section below for design guidance
Platform Name (Optional)	Used when the domain resource is common across platforms and the URI needs to be routed to that specific platform. Not used if platform agnostic

Examples:

- Service Routing → http:/<gatewayEndpoint>/api/v2/enrollment/enrollments
- Platform → http:/<gatewayEndpoint>/api/v2/enrollment/enrollments/platformX
- Multiple Resources & Primary Keys → http:/<gatewayEndpoint>/api/v2/enrollment/personEnrollments/{businessProcessReferenceid}/plan/{planId}/aPlatform

URI Naming Standards

- Every URI MUST follow the general rules for camelCase naming.
  - Hyphens (-) SHOULD NOT be used to delimit combined words
- A URI MUST NOT end with a trailing slash (/).
  - Example: A well-formed URL Example: /plan/1234
- In addition to general naming Rules, URI Variable names MUST follow the RFC6570 specification. That Is, the variable names can consist only from ALPHA / DIGIT / "_"
- Per RFC6570 Hyphen (-) is NOT legal URI variable name character.

Example :A well-formed URI Template Variable Example: /enrollment/personEnrollments/{businessProcessReferenceNumber}/plan

- HTTP Headers
  - Every HTTP Header SHOULD NOT use Hyphenated-Pascal-Case.
  - A custom HTTP Header SHOULD NOT start with X- (RFC6648)

Query Parameters and Path Fragments

- Every URI query parameter or fragment MUST follow the general rules.
- Also, they MUST NOT clash with the reserved query parameter names. The following are reserved words:
  - offset - used for collection pagination
  - limit - used for collection pagination
  - first - ??
  - last - ??

Pagination

- Query Parameters The following are reserved words are used to position the request in the collection:
  - offset - Position in the cursor for the request (Alternate: page)
  - limit - Maximum number to return (Alternate: per-page)
  - first - ??
  - last - ??
- Response Fields
  - Total - Total Number in cursor
  - Per-Page - Number of resources in response
  - Page - usedPage number in cursor
  - Link - Used if following HATEOUS

Additional Special Cases

- Overloaded POSTs - Add '/get' to end pf the URL path
- Data Views of large business objects - Special views of domain API - Add 'view/<viewName>"
- If the view input is considered optional, it would be best to make it a Query Parameter,

not a Path Parameter

- Query Parameters for GET inputs (Optional)
  - POST and PUT - Http Request Body contains inputs
  - Primarily used for non-primary key retrieval/filtering (where predicate)

API HTTP Response Codes

While designing a REST API, DON'T just use 200 for success or 404 for error. Every error message needs to be customized as NOT to reveal any unnecessary information. H ere are some guidelines to consider for each REST API status return code. Proper error handle may help to validate the incoming requests and better identify the potential security risks.

Table 2. HTTP Status Code - ErrorTable

HTTP Status Code	Description
200 - OK	Request was successful
201	Request was successful and resource created
204	Request was successful and no content is returned
400 - Bad Request	The request is malformed, such as message body format error.
401 - Unauthorized	Wrong or no authentication ID/password provided.
402 - For async requests ????	
403 - Forbidden	It's used when the authentication succeeded but authenticated user doesn't have permission to the requested resource
404 - Not Found	When a non existent resource is requested
405 - Method Not Allowed	The error checking for unexpected HTTP method. For example, the RestAPI is expecting HTTP GET, but HTTP PUT is used.
409 - Conflict	
422	There is something invalid or missing in the request body.

HTTP Status Code	Description
429 - Too Many	The error is used when there may be DOS attack detected or the request is rejected due to rate limiting
500	Server encountered an unexpected condition which prevented it from fulfilling the request
503	Server timed out when processing the request

Domain API Design Guidelines

At the highest level, the APIs are separated into types. These types would be:

Domain Representing domain business objects fields without any display fields. In an MVC world, a pure model.
Example: /person

Channel representing a channel component, which could be a mashup of domain fields plus display related fields.
Example: /personDashboard, /someWidget, /textResolver

Command representing a command or action that doesn't have a 'bounded context' associated with it. Example: /startSessionForPerson, /loginPerson

Initially, we will only denote the 'channel' in the URL name

The Domain and Channel API are noun based. So, to distinguish domain API from channel API, we will add the path name of '/channel' to the URL.

- If the URL REST resource is a noun and does not have the word channel in the path, it is considered a domain API.
- If the URL REST resource is a noun and does have the word channel in the path, it is considered a channel API.
- If the URL REST resource starts with a verb, it is considered a command API.

The path words '/api' will denote these URLs are api based versus a general Ux web requests and will be placed between the Base URL Context and the Rest Resource Name

The last level would be what we consider the REST resource or the bounded context.

API version would be part of the URL and is to be put between the 'api' path and the resource name path

Rest Resource Design

The Rest Resource or 'bounded context' will identify the business object related to the desired action.

The format for the REST resource is a series (one of more) of the following:
"bounded context name"/{primary key}

Bounded contexts will have a single entity root identified by a primary identity key(s). If key is considered PII data, the {primary key} or {id} would be placed in an encrypted session token

- Only bounded contexts and ids can be part of the resource. If the business object is split up and still has the same id as the parent level, we will not split the name.
  - For example, personHealth would not be /persons/health, since health is not a bounded context with a different key than person
- If the bounded context is an association between two bounded contexts, one should have multiple pairs, one for each primary key of the association.
- If a component does not have a primary id that meets the bounded context rules, it should not be in the REST resource
- In the URL, the "bounded context name" should be plural. This positions the API to return collections for the bounded context

API Design Naming

For domain API, it should be would the high level independent business objects and one nested level lower:

- Primary - person, plan, client, activity, person Activity
- Subcategory , same id - personWealth, personHealth

For channel API , it could be the UX component or widget object:

- Examples: personDashboard. contribution Change Conversation

For command API, it would represent the verb or action being taken:

- Examples: logonPerson, startSessionForPerson, search

Design Considerations

- Rest resource names should be plural and end with an 's' to denote a collection. No need to define the true plural (People for persons)
  - If {id} is defined, only the document associated with the id will be retrieved
  - If no {id} is defined, the entire collection is considered.
  - Any filtering would be defined in a query parameter
- Except for command API, only the 4 HTTP verbs can be used - GET,PUT,POST, DELETE
  - Resource names should not contain any verbs (get, update, revise)

    - There are some exceptions described below
- Resource names will follow domain driven design modeling approach for defining REST resource
    - If a framework ins being used, will favor following a domain design taxonomy over a framework's REST approach
    - Will not follow a Odata/SQL like language-based approach to get data. This is where the Rest API is treated like a declarative SQL statement. This approach is more complex and provides way to much flexibility
    - Makes it hard to create a concrete specification
    - Opens up some potential data security issues and access control to data
- If the request is bringing back a collection, we will use Cursor based processing names placed in query parameters (paging, skipping).
    - For example, if a 'clients' API is bringing back a list, one would navigate through the list, by skipping 'x' number of records → /clients?skip=20;page=10 would bring back 10 objects, starting with the 21st object in the list

Command API Design

Guiding Principals

- Any command API should end with a verb-oriented resource name.
- Two types:
    - Algorithmic - POST - No change in state
    - Status or State change - PUT - Changing the state of an entity, not the descriptive attributes

Channel API Design

This chapter is only appropriate is the company is planning on using API to drive the rendering of of the Ux interface.

What is channel content?

- Channel field/content is a description or explanation of a domain 'fact':
- Domain data transformed Into the Client’s 'Communication Voice'
- Content being provided In the preferred language of the person
- Rendered in the appropriate device capabilities or medium
  - Web Browser
  - Mobile Native Application
  - Voice Assistant (Alexa)
  - Communication - Paper, Email, Text

The above should be maintained by the channel platform layer, except for content related to: * Legally bound messages that cannot be changed or excluded * Small descriptive attributes that give human connection to an internally created identifier

Guiding Principals

- The domain API should deliver raw data facts, independent of the channel or device they are rendered on
- The channel API is responsible for what is shown, where and how it’s displayed and what content to communicate
  - In addition, it is responsible for gather user input to drive business processes
- With the trend to more voice assistant, one should ask how the domain fact would be rendered in a voice mode. This helps focuses on the domain data and less on how it is being rendered in the device. For example, a voice system doesn’t leverage any formatting of zip codes, phone number or currency. The rendering device channel API would be where the device aspects are injected.

Channel Messages and Ux Content

- In general, all content should be stored, managed and retrieved by the channel API
- The channel API should support different languages and the formatting of country values

(currency, dates, etc)

- If the data is currently in an external content management system, then it is the responsibility of the channel API to retrieve and manage.
- If any piece of content being shown to the user is greater the 30 characters, then it is the responsibility of the channel/device to manage.
  - The domain API will provide a identifier or 'messageLabel' to link which piece of content to assemble and show. The actual content will be the responsibility of the channel API.
- The only expectations are:
  - Business object descriptions (i.e. plan descriptions, etc). These will be provided by the domain API, if it is defined in the domain as a description, then it would be provided by the domain layer.
  - Footnotes or Legally required content required by law

What situations would create a Channel API?

The following situations would lend themselves to creating a channel API versus a domain API:

Data Marshalling for UI widgets/components

- This type of API is created when there is a need to combine domain data with display and navigational directives. The marshalling of these sources is a precursor to rendering and passed on to components like Angular for rendering.
- The sources could be domain API, data from UX configuration data bases and calls to third party.
- Since these API are going over a network, the granularity of the 'widgets' is important. We probably want to avoid API for smaller Ux controls and even tiles. The proper granularity is:
  - A collection of control creating something like a dashboard or navigational component
  - A key UX page, like a Home Page or Domain Landing Page
  - A series of pages that make up a business process conversation or drill down inquiry flow
- The bounded context and primary key for these components are likely to be:
  - ComponentID for major UX components
  - PageId for key UX pages
  - Conversation or FlowId for multi-page flows

- The pattern here might be a combination of:
- WidgetConfiguration service call for configuration (text, links, expressions) for the widget being displayed, plus
- A domain call leveraged by the widget
- Then the service would be responsible for mashing the two components together.

Domain API aggregation

- This type of API is created when there is a need to aggregation of two or more domain bounded contexts. At the domain level, the resource will be very strictly organized around primary 'bounded contexts' and their primary keys. For example, if there is a desire to combine some person data and a person's defined contribution data, a channel API can be created that would call both of the domain API and provide an aggregated response. There are two types of aggregation:
  - Aggregation across multiple bounded contexts with the same primary key, PersonId to PersonId
  - Aggregation with two different bounded context with a connected key. For example, an API that would combine a person plan participation with a plan's provisions - PersonId+PlanId to PlanId
- Special channel aggregation views for 3rd parties and partners
  - These API would be similar to the Domain API or Aggregated Domain API, but would be for external 3rd parties and partners. In addition, selected special components could also have their own set of channel or 'integration' API.

Content Management API

- These API would be centered on the needs of Content Management. The types of content that could be provided as services would be:
  - Text resolvers for client specific symbolic replacement
  - Link resolvers
  - Document/Content for Language translation
  - Others
- The key concern with these API is granularity. It doesn't make sense to make a single call for a single word or link. In these cases, it might be best to create an underlying library, such that the API can process a group of these smaller functions. Either the text or links can be:
  - logically grouped and processed together (Asset Groups, WidgetConfiguration)or
  - Be part of some large context, like a Landing Page or page

Domain Driven Modeling for API Design

The domain API deliver raw data facts, independent of the channel or device they are rendered on

Domain Driven Design Theory

From a domain business perspective, there are three types of Business Objects. They typically follow general data base design strategies.

1. Fundamental Entity - Primary independent business objects like, person, plan or client. Will have its own single value primary keys
2. Dependent Entity - The entity's existence is dependent on the existence of a Fundamental entity. Typically, has a foreign key of the Fundamental entity
3. Associative Entity - It is the relationship formed by the joining of two Fundamental entities. It may have it's own independent key, but will always have foreign keys to the Fundamental entities.

Fundamental Entity

The following is the best practice design for primary independent business entities. A primary independent entity is a business domain object with the following characteristics:

- The entity has a single primary key that uniquely defines the entity
- Examples:
  - Person → Primary Key: personId
  - PersonBusinessProcess or PersonActivity → Primary Key: businessProcessReferenceNumber or actRefNum
  - Activity → Primary Key: ActId
  - Document → Primary Key: documentId **

URI Design Best Practices

Figure out how to indent

If you are retrieving a specific entity based on its primary key. Will return one instance of the entity: GET /businessObjects/{primaryKey}

If you are retrieving a collection of entities based on a group of fields to filter the entire collection. This is considered a non-primary key 'where predicate', where the fields become

query parameters: GET /businessObjects?queryParm1=value1&queryParm2=value

If you want to retrieve the entire collection. Use 'limit' and 'offset' query parameters if you need to take a 'cursor' approach for large collections GET /businessObjects

If you want to update a single entity: PUT /businessObjects/{primaryKey}

If you want to create a single entity. Any pre-determined primary key should be put in the request body of POST POST /businessObjects

If you want to delete a single entity. DELETE /businessObjects/{primaryKey}

Rules

- All resource names should be plural
- All names (resource, query parameters, form fields) should be camel case.
- We highly discourage doing the following with the URL /businessObjects
  - Bulk insert (POSTs)
  - Bulk update (PUTs)
  - Bulk deletes (DELETEs)

Dependent Entity

A Dependent Entity is created when the is a parent-child relationship. The Dependent Entity□s existence is dependent on the existence of the parent. It can not stand alone

Two types of Parent-to-Child Relationships: * One Parent to Many Children * One Parent to only One Child

Rules

- They follow all the parent rules
- If there is a collection of dependents for a given parent,
  - All children entities can be part of the request/response of the parent API
  - If the request is based on a single child, the key of the child should be an optional query parameter for GETs and the request for PUTs/POSTs/Deletes. The key should not be a path parameter.

Associative Entity

An Associative Entity is created by the union of two Fundamental Entities. The relationship is typically Many-to-Many, but it can be One-to-One. These entities act like a Fundamental entity , but will have both primary keys.

Rules

- They follow all the Fundamental Entity Rules
- In general, both primary keys of the Fundamental Entities can be on path parameters
- The entity itself can have dependent entities from the join and follow the above dependent rules
- There are situations where the API can navigate from one side of the relationship.
  - In this case, the other side is a collection of that entity with the primary key as a key into that map.
  - For example, in a Person-Plan Associative relationship, if only the person primary key is known, an API can bring back the collection of Plans, with the PlanId the key of that collection.

Special Case API Situations

Asynchronous Task -→ Work in Progress If an API operation is asynchronous, but a client would like to track its progress, the response to such an asynchronous operation MUST always return. A HTTP response status code of 202 - Accepted together with an application/json representation of a new task-tracking resource.

Task Tracking Resource

- The task-tracking resource SHOULD convey the information about the status of an asynchronous task.
- Retrieval of such a resource using the HTTP GET Request Method SHOULD be designed as follows:
  - Task is Still Processing
    - Return 200 OK and representation of the current status. **Task Successfully Completed
    - Return 303
  - Task Failed
    - Return 200 OK with the problem detail information in the response header on the task has failed.

Design Note

- The asynchronous operation task-tracking resource can be either polled by client or the client might initially provide a callback to be executed when the operation finishes.
- In the case of callback, the API and its client MUST agree on what HTTP method and request format is used for the callback invitation.

Batch Operations

Processing Similar Resources

An operation that needs to process several related resources in bulk SHOULD use a collection resource with the appropriate HTTP Request Method. When processing an existing resource, the request message body MUST contain the URLs of the respective resources being processed.

Results of Bulk Operation

Every bulk operation MUST be atomic and treated as any other operation.

The server must implement bulk requests as atomic. If the request is for creating ten addresses, the server should create all ten addresses before returning a successful response code. The server should not commit changes partially in the case of failures.

DO NOT USE "POST Tunneling." Every API MUST avoid tunneling multiple HTTP Request using one POST request. Instead, provide an application-specific resource to process the batch request.

Non-atomic Bulk Operations Non-atomic bulk operations are strongly discouraged as they bring additional burden and confusion to the client and are difficult to consume, debug, maintain and evolve over the time.

The suggestion is to split a non-atomic operation into several atomic operations. The cost of few more calls will be greatly outweighed but the cleaner design, clarity and easier maintainability.

However, in such an operation has to be provided such a non-atomic bulk operation MUST conform to the following guidelines.

Non-atomic bulk operation MUST return a success status code (e.g. 200 OK) only if every and all sub-operation succeeded.

If any single one sub-operation fails the whole non-atomic bulk operation MUST return the respective 4xx or 5xx status code.

In the case of a failure the response MUST contain the problem detail information about every sub-operation that has failed.

The client MUST be aware that the operation is non-atomic and the even the operation might have failed some sub-operations were processed successfully.

Search Requests

A search (filter) operation on a collection resource SHOULD be defined as safe, idempotent

and cacheable, therefore using the GET HTTP request method.

Every search parameter SHOULD be provided in the form of a query parameter. In the case of search parameters being mutually exclusive or require the presence of another parameter, the explanation MUST be part of operation's description.

Alternative Design

When it would be beneficial (e.g. one of the filter queries is more common than another one) a separate resource for the particular query SHOULD be provided. In such a case, the pivotal search parameter MAY be in the form of a path variable.

Request/Response Entities Design Guidelines

JSON Format Conventions

Any JSON-based message MUST conform to the following rules:

- All JSON field names MUST follow the Naming Conventions (camelCase, American English, etc.)
- Field names MUST be ASCII alpha num characters,
  - Underscore (_) or dollar sign ($) should only be used in special approved cases
- Boolean fields MUST NOT be of null value
- Fields with null value SHOULD be omitted
- Empty arrays and objects SHOULD NOT be null (use [] or {} instead)
- Array field names SHOULD be plural (e.g. "plans": [])

Common Data Types

The following standards apply to common data types:

- Date and Time Format
  - Date and Time MUST always conform to the ISO 8601 format e.g.: 2017-06-21T14:07:17Z (date time) or 2017-06-21 (date), it MUST use the UTC (without time offsets).
- Duration Format
  - Duration format MUST conform to the ISO 8601 standard e.g.: P3Y6M4DT12H30M5S (three years, six months, four days, twelve hours, thirty minutes, and five seconds).
- Time Interval Format
  - Time Interval format MUST conform to the ISO 8601 standard e.g.: 2007-03-01T13:00:00Z/2008-05-11T15:30:00Z.
- Numbers will be provided in the domain API as real numbers with imbedded decimal points
- Language Code Format
  - Language codes MUST conform to the ISO 639 e.g.: en for English.
- Country Code Format

    - Country codes MUST conform to the ISO 3166-1 alpha-2 e.g.: DE for Germany.
- Currency Format o Currency codes MUST conform to the ISO 4217 e.g.: EUR for Euro.
- Field Formatting
    - In general, formatting should be done by the channel based on language and locale. Fields like:
        - Zipcode
        - Currency
        - Dates
            - The only exceptions are:
        - Full Name formatted
        - Phone number formatted

Simple Math

- Numbers will be provided in the domain API as real numbers with imbedded decimal points
- The channel API should be responsible for simple math - addition, subtraction. Multiplication and division
    - Given floating point issues in the programming model, it would be allowable to provide some complex calculation involving multiplication and division.
    - The channel API should be responsible for simple data and time math, where domain policy and variation by client do not apply

Separation of Concerns

Every API using HTTP/S API MUST precisely follows the concern separation of an HTTP message:

- A resource identifier - URI MUST be used to indicate identity only
- HTTP request method MUST be used to communicate the action semantics (intent and safety)
- HTTP response status code MUST be used to communicate the information about the result of the attempt to understand and satisfy the request
- HTTP message body MUST be used to transfer the message content
- HTTP message headers MUST be used to transfer the metadata about the message and its content
- URI query parameter SHOULD NOT be used to transfer metadata

Example 1

The rule:

A resource identifierURI MUST be used to indicate identity only implies there MUST NOT be any information about the representation media type, version of the resource or anything else in the URI.

For example, URIs /greeting.json or /v2.1.3/greeting are illegal as they are not used for identification of a resource only but they convey the information about representation format or version. URIs are not meant to carry any other information but the identifier of the resource.

Example 2

The rule:

HTTP message body MUST be used to transfer the message content implies an HTTP GET request MUST NOT use HTTP message body to identify the resource.

For example, a request:

GET /greeting HTTP/1.1 Content-Type: application/json ... { "filter": "string" "depth": 3 }

is not acceptable (ignoring the fact that HTTP GET method shouldn't have the body). To express identity use URI and query parameters instead e.g. /greeting?filter=string&depth=3 .

Keep things simple while designing by separating the concerns between the different parts of the request and response cycle. Keeping simple rules here allows for greater focus on larger and harder problems.

Requests and responses will be made to address a particular resource or collection. Use the path to indicate identity, the body to transfer the contents and headers to communicate metadata. Query params may be used as a means to pass header information also in edge cases, but headers are preferred as they are more flexible and can convey more diverse information. - Heroku HTTP API Design Guide

Client/3rd party/Partner API Considerations

When dealing with a client or 3rd Party or partner, who would like to consumer any domain-based API, the following best practices apply:

- In general, there is only one domain inquiry API for each domain.
  - The current list of domains is person, worker, person+worker, DC, DB, HM and RA - Reimbursement Account.
  - There may be reasons for size of API response and performance considerations that we would create subset views of the full set of inquiry data for a domain

- If we decide to create a 'view' of the domain API for a channel, 3rd party, partner or client, we will create a channel faCade API that would call the domain API and transform/map that data into the response of that channel API. This will create two types of API:
  - General subset channel views for any consumer
  - Client/Partner Specific channel view
- If we want to restrict/provide a view to a specific client or partner, a client/partner specific channel API would need to be created and we would apply our new Access Control such that only that consumer can access that view.

Versioning

The REST API is not a static entity. As one adds more domain data attributes and additional source platforms, the APIs for a given resource will continue to evolve. As one makes changes to the API request and response structures, one needs to be mindful of how it affects the consumers of the API. The company needs to have a versioning strategy that allows them to evolve the API quickly without forcing major changes on the consumers. As the number of consumer start to grow, this will be ever more important. We do not want to be in a position where the consumers must make change based on a schema version change. The goal is to decouple the consumer/provider development chain, allowing the consumers to upgrade at their own pace and the API to evolve independent of the consumers.

The approach is to implement a major version/minor version strategy. To accomplish this, we will be focused on backward compatibility.

If the API structures need to change and the change is backward compatible to the consumer, then this will constitute a minor version change.

If the API structures need to change and the change is not backward compatible to the consumer, then this will constitute a major version change.

From a naming perspective, we will follow the naming best practice of {major version}.{minor version}. Since we are using JSON, we will have more flexibility

Minor Version

For a change to be considered a minor version, it must be backward compatible. If so, it should have no material impact on the consumer, unless they want to consume the new changes. Our goal is to encourage minor versions and avoid major versions. Since we are using JSON, we will have more flexibility on creating minor versions over the current SOAP WSDL operation specification.

A minor version can be implemented if it meets the following requirements:

- On request fields, the field can be considered 'optional'
- Fields are moved or added within a sub document section (JSON Object: person, contact) or moved within the same type of structure - flat/non array or array/map
- A field is deprecated but not removed.

If a minor version is created,

- The major release number will remain the same
- The endpoint URL will remain the same

- It will be added to the version numbering as a 'dot' release, adding one to the prior minor release number
- The schema version field inside the response body will be changed to indicate the new version number

Major Version

If a data change breaks backward compatibility with any consumer, it will create a new major version. Our goal is to not force a consumer to make a change at the same time the new version is deployed. The guiding principle of a major version is to clean up any issues cause by the creation of redundant fields and deprecated fields which should be removed. The goal is to not have more than once per year, primarily focused on cleanup.

A major version must be implemented if it meets the following requirements:

- On request fields, the field is considered 'required'
- A field changes its data type (e.g. number to string)
- Should never change - Direction is to create a new field
- A field is removed
- Deprecated field should be removed after two or more minor releases
- Once a year, cleaning up stuff
- Fields are moved from a flat structure to an array/map structure or array to flat.

If a major version is created,

- The major release number will increase by one
- A new endpoint URL will be created with the major version number - 'V2'
- The minor version number will be set to zero
- The response Version inside the response body will be changes to indicate the new version number, will denote the complete version number, major and minor

Rules for Extending

Any modification to an existing API MUST avoid breaking changes and MUST maintain backward compatibility.

Any change to an API MUST follow the following Rules for Extending:

- You MUST NOT take anything away (related: Minimal Surface Principle, Robustness Principle)

- You MUST NOT change processing rules
- You MUST NOT make optional things required
- Anything you add MUST be optional (related Robustness Principle)

API version implementation - URL based.

The initial POV will be to:

- Put the major version on the base URL between the base endpoint and the resource name.
  - /{BaseURL}/api/{version number}/{resource}?{query parameters}
- The major version and minor version will be denoted on the response of the API response structure
- It will contain the {major version} dot{ minor version} (e.g. 2.1, 3.2)
  - Like Maven, do we need additional attributes, like Release Candidate or Snapshot with a date timestamp
  - If a combination of URL route and HTTP verb is being deprecated (i.e still working, but has an end of life date), then the API response should provide that information.
- If in a future version, the API (route and Verb) are disabled (i.e no implementation), then the service should return an HTTP status code of: 410 Gone

API Security

Security stuff goes here

Non Standard REST situations

Overloaded POST for GETS Query Requests with Large Inputs

While HTTP doesn't impose any limit on the length of a URI, some implementation of server or client might have difficulties handling long URIs (usually URIs with many or large query parameters)

Every endpoint with such a URI MUST use the HTTP POST Request Method and send the query string in HTTP Request Message body.

Recommendation: Add the word /get to the end to denote it is truly a GET

Since this operation is safe and idempotent, using the POST method violates the HTTP protocol semantics and results in loss of cache-ability.

Additional Topics

Robustness

Every API implementation and API consumer MUST follow Postel's law:

Be conservative in what you send, be liberal in what you accept. - John Postel

That is, send the necessary minimum and be tolerant as possible while consuming another service (tolerant reader).

Minimal API Surface

- Every API design MUST aim for a minimal API surface without sacrificing on product requirements.
- API design SHOULD NOT include unnecessary resources, relations, actions or data.
- API design SHOULD NOT add functionality until deemed necessary (YAGNI principle).

Input Validation

Everything you know about input validation applies to RESTful web services, but add 10% because automated tools can easily fuzz your interfaces for hours on end at high velocity Help the user input high-quality data into your web services, such as ensuring a Zip code makes sense for the supplied address, or the date makes sense. If not, reject that input. Also, make sure that the output encoding is robust for your application. Some other specific forms of input validations need to be implemented:

- Secure parsing: Use a secure parser for parsing the incoming messages. If you are using XML, make sure to use a parser that is NOT VULNERABLE to XXE and similar attacks.
- Strong typing: It's difficult to perform most attacks if the only allowed values are true or false, or a number, or one of a small number of acceptable values. Strongly type incoming data as quickly as possible.
- Validate incoming content-types: When POSTing or PUTting new data, the client will specify the Content-Type (e.g. application/xml or application/json) of the incoming data. The server SHOULD NEVER assume the Content-Type; it SHOULD ALWAYS check that the Content-Type header and the content are the same types. A lack of Content-Type header or an unexpected Content-Type header SHOULD result in the server rejecting the content with a 406 Not Acceptable response.
- Validate response types: It is common for REST services to allow multiple response types (e.g. application/xml or application/json, and the client specifies the preferred order of response types by the Accept header in the request. DO NOT simply copy the Accept header to the Content-type header of the response. Reject the request (ideally with a 406 Not Acceptable response) if the Accept header does not specifically contain one of the allowable types. Because there are many MIME types for the typical response types, it's

important to document for clients specifically which MIME types should be used.

- XML input validation: XML-based services MUST ensure that they are protected against common XML-based attacks by using secure XML-parsing. This typically means protecting against XML External Entity attacks, XML-signature wrapping etc (Note XML is not a standard protocol at this time. JSON is preferred)

Safe Methods

As per HTTP specification, the GET and HEAD methods should be used only for retrieval of resource representations - and they do not update/delete the resource on the server. Both methods are said to be considered 'safe'. This allows user agents to represent other methods, such as POST, PUT and DELETE, in a special way, so that the user is made aware of the fact that a possibly unsafe action is being requested - and they can update/delete the resource on the server and so should be used carefully.

Idempotent Methods

The term idempotent is used more comprehensively to describe an operation that will produce the same results if executed once or multiple times. This is a beneficial property in many situations, as it means that a transaction can be repeated or retried as often as necessary without causing unintended effects. With non-idempotent operations, the algorithm may have to keep track of whether the operation was already performed or not. In HTTP specification, The methods GET, HEAD, PUT and DELETE are declared idempotent methods. Other methods OPTIONS and TRACE SHOULD NOT have side effects, so both are also inherently idempotent.

Appendix A: API Glossary

API An application program interface, which defines a collection of "REST Operations" centered around a logical business concept - domain object, domain transaction or channel component. Mostly focused on a REST protocol and interface design. A deployed service in Docker is typically based on a single API .

REST Operation A combination of a web resource URL and HTTP Method (GET, POST, PUT, Delete) A logical functional service, feature or component. This sometimes called a service or operation.

Channel API A REST API based on the display and flow component in the various channels. The resources are a mashup of domain facts obtained from the domain API and channel related components, called "widget". It is not the HTML displayed in the device, but the input into the creation of HTML leading to the rendering on the device. The objects managed by these API are primarily the display 'widget' presented to the user. They do not perform any inquiry or transactions, delegating that work to the domain API.

Domain API A REST API based on the business's domain objects, independent of channel. These API support the processing core of the business domain, supporting both inquiry and transaction needs. The high level business objects include person, worker, and participation experience in benefits plans. They also include the business process and supporting documents that manage a person's benefits. * Inquiry * Transaction

Microservice An architectural approach for service design based on smaller components and deployment independence

Deployed Service A Spring Boot base code base deployed in a Docker based container based primarily on a API or subset of an API

API Protocols * REST - HTTP/JSON * SOAP - HTTP/WSDL/XML

Appendix B: 12 Factor App design

Taken from web site : https://12factor.net/

Introduction In the modern era, software is commonly delivered as a service: called web apps, or software-as-a-service. The twelve-factor app is a methodology for building software-as-a-service apps that: * Use declarative formats for setup automation, to minimize time and cost for new developers joining the project; * Have a clean contract with the underlying operating system, offering maximum portability between execution environments; * Are suitable for deployment on modern cloud platforms, obviating the need for servers and systems administration; * Minimize divergence between development and production, enabling continuous deployment for maximum agility; * And can scale up without significant changes to tooling, architecture, or development practices. The twelve-factor methodology can be applied to apps written in any programming language, and which use any combination of backing services (database, queue, memory cache, etc).

Background The contributors to this document have been directly involved in the development and deployment of hundreds of apps, and indirectly witnessed the development, operation, and scaling of hundreds of thousands of apps via our work on the Heroku platform. This document synthesizes all of our experience and observations on a wide variety of software-as-a-service apps in the wild. It is a triangulation on ideal practices for app development, paying particular attention to the dynamics of the organic growth of an app over time, the dynamics of collaboration between developers working on the app s codebase, and avoiding the cost of software erosion.

Our motivation is to raise awareness of some systemic problems we ve seen in modern application development, to provide a shared vocabulary for discussing those problems, and to offer a set of broad conceptual solutions to those problems with accompanying terminology. The format is inspired by Martin Fowler s books Patterns of Enterprise Application Architecture and Refactoring. Who should read this document? Any developer building applications which run as a service. Ops engineers who deploy or manage such applications.

The Twelve Factors I. Codebase One codebase tracked in revision control, many deploys

II. Dependencies Explicitly declare and isolate dependencies

III. Config Store config in the environment

IV. Backing services Treat backing services as attached resources

1. Build, release, run Strictly separate build and run stages

VI. Processes Execute the app as one or more stateless processes

VII. Port binding Export services via port binding

VIII. Concurrency Scale out via the process model

IX. Disposability Maximize robustness with fast startup and graceful shutdown

1. Dev/prod parity Keep development, staging, and production as similar as possible

XI. Logs Treat logs as event streams

XII. Admin processes Run admin/management tasks as one-off processes

DRAFT Version 1.0