Content

API Style

Resquests

URL structure

All endpoints must be prefixed with /v1/. Pattern: `/v1/...`

Collections of resources are referenced by their resource name (plural)
Pattern: `/v1/:resource name`
Example: `/v1/accounts`

Individual resources are referenced their resource name (plural) followed by the guid
Pattern: `/v1/:resource name/:guid`
Example: `/v1/accounts/25fe21b8-8de2-40d0-93b0-c819101d1a11`

Resources

A resource represents an individual object within the system, such as an account or a branch. It is represented as a JSON object.

A resource **MUST** contain the following fields:

- `guid` : a unique identifier for the resource
- `created_at` : an ISO8601 compatible date and time that the resource was created
- `updated_at` : an ISO8601 compatible date and time that the resource was last updated

A resource **may** contain additional fields which are the attributes describing the resource.

A resource **MUST** include a `self` link object in the `links` field.

Example

```
{
  "guid": "a-b-c",
  "created_at": "2015-07-06T23:22:56Z",
  "updated_at": "2015-07-08T23:22:56Z",

  "name": "school account",
  "description": "an example account",

  "links": {
    "self": {
      "href": "/v1/accounts/a-b-c"
    }
  }
}
```

HTTP Methods

GET

Used to retrieve a single resource or a list of resources.

- GET requests **may** include query parameters
- GET requests **must NOT** include a request body

Example

Show individual resource: `GET /v1/accounts/:guid`

Responses (Resource)

Scenario	Code	Body
Authorized User	200	Resource
Unauthorized User	404	Error

List collection of resources: `GET /v1/accounts/`

Responses (Collection)

Scenario	Code	Body
User With Complete Visibility	200	List of All Resources
User With Partial Visibility	200	List of Visible Resources
User With No Visibility	200	Empty List

POST

Used to create a resource.

- POST requests **must NOT** include query parameters
- POST requests **may** include a request body

Examples

Create a resource:

```
POST /v1/accounts/

{
  "customer_guid": "ab09cd29-9420-f021-g20d-123431420768"
  "account_number": "12345678",
  "brach_guid": "guid-bd7369a8-deed-ff1a-2315-77410293a922"
,  ...
}
```

Responses

Scenario	Code(s)	Body
Authorized User (sync)	201	Created Resource
Authorized User (async)	202	Empty w/ Location Header -> Job
Read-only User	403	Error
Unauthorized User	403	Error

PUT

Used to trigger an action, basically to update a resource

- PUT requests **must NOT** include query parameters
- PUT requests **may** include a request body
- PUT requests **MUST be** idempotent

Examples

- Trigger an action:

```
MISSING...
```

- Partially update a resource:

```
PUT /v1/accounts/:guid

{
  "alias": "new_alias_name"
}
```

Responses

Scenario	Code(s)	Body
Authorized User (sync)	200	Empty
Authorized User (async)	202	Empty w/ Location Header -> Job
Read-only User	403	Error
Unauthorized User	404	Error

DELETE

Used to delete a resource.

- DELETE requests **may** include query parameters
- DELETE requests **must NOT** include a request body

Examples

Delete a resource:

```
DELETE /v1/accounts/:guid
```

Responses

Scenario	Code(s)	Body
Authorized User (sync)	204	N/A
Authorized User (async)	202	Empty w/ Location Header -> Job
Read-only User	403	Error
Unauthorized User	404	Error
Missing Resource	404	Error

Links

Links provide URLs to relationships and actions for a resource. Links are represented as a JSON object.

Actions

Query Parameters

Filtering

Relationships

Nested Resources

Including Related Resources

This is a mechanism for including multiple related resources in a single response.

Pagination of Related Resources

Requesting Partial Resources

Design

Naming Conventions

In the world of APIs, there are URIs, resources, and representations. A URI is what allows you to access a

resource. What is returned, in response, is a representation of that resource. Since we're a bank, let's take "account" as an example, you may have:

url

`http://.../accounts/12345`

resource

`/accounts`

representation

```
{
    "accounts": [{
        "balance": {
        "available": 350.15,
        "current": 149.85
        },
        "meta": {
            "limit": 500.00,
            "name": "My Credit Card",
            "number": "12345"
        },
        "institutionType": "citi",
        "type": "credit"
    }]
}
```

Citi APIs should only have two URIs that represent a given resource. In the example above, the API URIs to either fetch all accounts or fetch a specific account, respectively, would look something like:

`/accounts` or `/accounts/123456`

Note that many times there are cases where a specific account cannot be fetched without some other relationship tied to it. It's, after all, almost certainly going to be associated with some account owner. In these scenarios, the following syntax should be used:

`/{resource}/{id}/{resource}/{id}`

As a result, our account lookup API would look something like:

`/customers/ABC123/accounts/123456`

For simplicity's sake, resources should rarely go deeper than what is outlined above. But, if necessary, the same `/{resource}/{id}` pattern should be followed.

Lastly, when assigning the resource name, it is better to use concrete names as opposed to making the resource overly abstract. For example, a resource of `/locations` may be too abstract. Does it refer to ATM Locations? Branches? Restrooms?

Rather, more suitable resource names could be `/atmLocations` or `/branchLocations`. The exception, of course, would be if a parent resource is already present in the hierarchy. In that case, one might expect to see something like `/branches/locations`.

For a set of APIs, the key is keeping the number of resources at a manageable level for the end-user. Apigee suggests a target of approximately 12-24 resources names for a set of APIs. If we need to go much beyond that, we should revisit our API taxonomy.

Methods

For those that have developed SOAP services within Citi, the use of verbs in the service definition has been promoted as a best practice, helping to make the service self-descriptive. In the world of REST APIs, this is no longer the case. Keeping simplicity in mind, **using verbs within the resource becomes cumbersome to navigate as an end user and MUST be avoided**. In our account example the list can quickly grow if VERBS are incorporated:

```
/getAccount
/createAccount
/deleteAccount
/updateAccountStatus
/getActiveAccounts
/validateAccount
/getAllAccounts
/updateExpirationDate
/updateActiveCustomers
/increaseCreditLimit
…
```

However, the use of nouns combined with HTTP methods allows us to narrow this down to a single resource and 4 HTTP verbs:

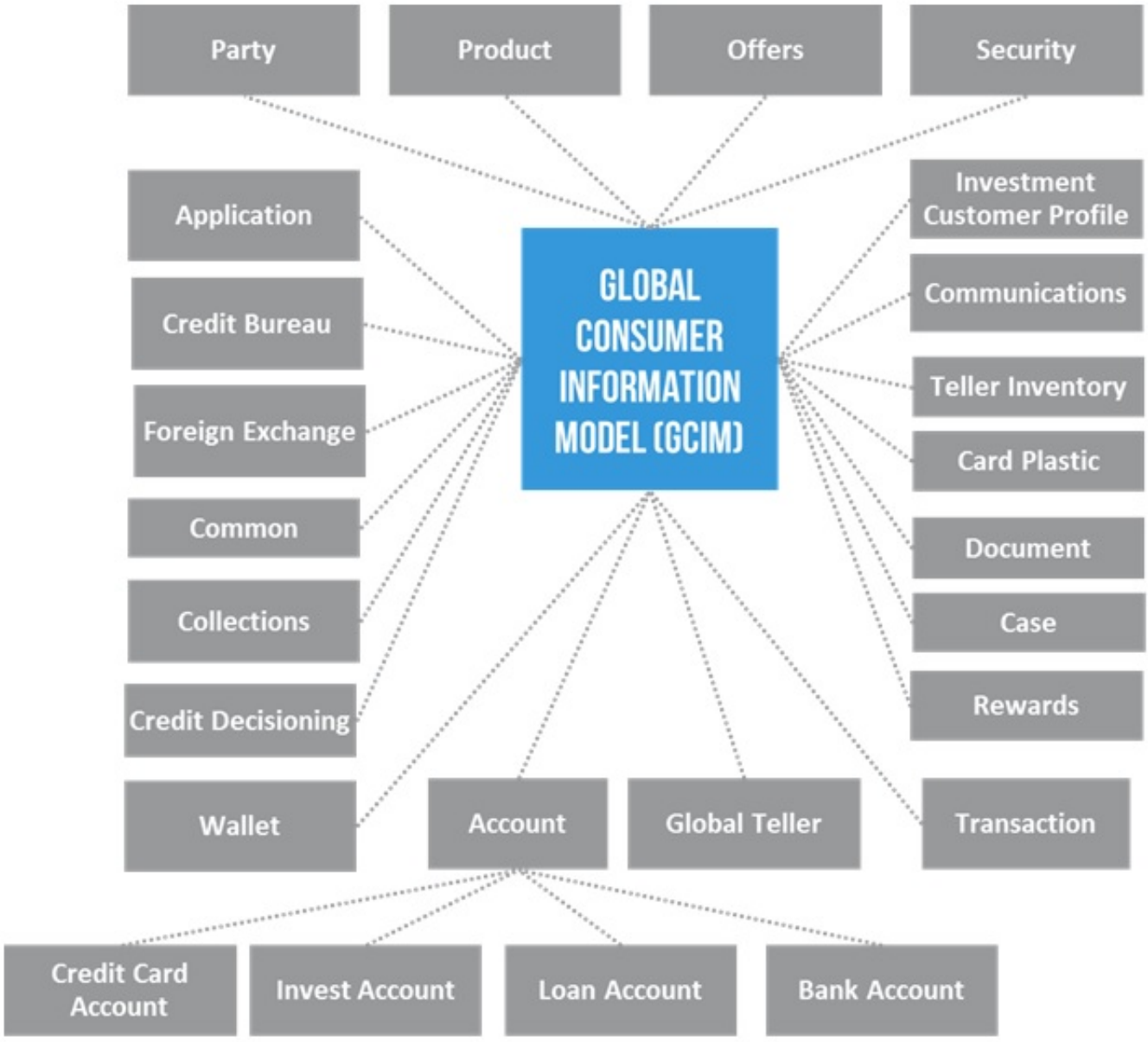
Operation	HTTP Method	Resource	Comments
Create	POST	`/accounts`	Creates an account with the parameters passed in the message body
Read	GET	`/accounts` or `/accounts/{guid}`	Fetches all accounts or a specific account's details, respectively
Update	PUT	`/accounts/{guid}`	Updates an account with the information passed message body (*Full* resource)
Update	PATCH	`/accounts/{guid}`	Updates an account (or accounts) with the information passed message body (*Partial* resource)
Delete	DELETE	`/accounts/{guid}`	Moves the account(s) to a "deleted" status

Consistency across all API

We want to keep them consistency across APIs. If we have a series of APIs which return an "account" object, that object should be the same across those APIs. The only exception would be in the case of the same API being used both publicly and privately to Citi. In this case, only a subset of the "account" elements should be sent to external parties. However, the "account" object returned should be consistent.

We also need to take into account the context of the API request. For example, an `/accounts` request which returns a list of accounts, may return a subset of the "account" object returned when making an `/accounts/1234` call which would return account details.

In order to support Citi's API strategy, the GCIM will be used as the basis for all payloads for newly developed APIs.



Resource and Enumeration Naming Conventions

The purpose of this living section is to define a strawman for standards and conventions to be followed for GCIM aligned objects and elements in Digital APIs. Projects should conform to the standards and if the standards are lacking, the project should raise the gap(s)/deviation to the Information Engineering team. The Information Engineering team will have active involvement in participating in governance processes for all digital APIs and the intent is to evolve the standards to best satisfy the objective of promoting reuse across the Digital APIs.

Conventions for Class names

- Class names should be nouns
- Class names should be simple and descriptive
- Class names should follow UpperCamelCase with following considerations:-
    - Wherever standard acronyms are available and applicable, fully defined names or non-standard acronyms/abbreviations for the same should not be used

- Class names should contain characters only. No numeric and special characters are allowed
- Class names should not contain non-approved abbreviations.
- Class names should not exceed 35 characters. If length exceeds 35 characters, then approved abbreviations should be used consistently from last word backward. Approved abbreviations can be found in appendix below
- Class names should not end with 'Request' and 'Response' since these are used to indicate root level classes (see next section)

Example: PartyRole, Address, DestinationAccount

Conventions for Class attribute names

- Class attribute names should be simple and descriptive
- Class attribute names should follow lowerCamelCase with following considerations:-
    - Wherever standard acronyms are available and applicable, fully defined names or non-standard acronyms/abbreviations for the same should not be used e.g. use 'url' *instead of* 'uniformResourceLocator'

- Class attribute names should contain alphanumeric characters only. No special characters are allowed
- Class attribute names should start with an alphabet and never with a numeric character.
- Class attribute names should not contain non-approved abbreviations.
- Class attribute name shouldnot start with the following words: 'get' and 'set'
- If a class attribute represents an array, then the attribute name should depict plural e.g. 'balances'

Example: accountNumber, statusFlag

Conventions for path parameter and query parameter names

- Parameter names should be simple
- Parameter names should follow lowerCamelCase with following considerations:-

    - Wherever standard acronyms are available and applicable, fully defined names or non-standard acronyms/abbreviations for the same should not be used e.g. Use 'url'*instead of* 'uniformResourceLocator'

- Parameter names should contain alphanumeric characters only. No special characters are allowed.
- Parameter names should start with an alphabet and never with a numeric character.
- Parameter names should not contain non-approved abbreviations.
- Query parameters that accept multiple values **MUST** be pluralized e.g. ‘accounts’.
- All query parameters **MUST** be properly url-encoded. If a query parameter value includes the comma (`,`) character, the comma **MUST** be double encoded. Note that for readability purposes, the examples throughout this document do not show encoded query strings.

Example: accountNumber, statusFlag

Data Types

Primitive data types in the Swagger Specification are based on the types supported by the JSON-Schema Draft 4. Models are described using the Schema Object which is a subset of JSON Schema Draft 4.

An additional primitive data type `"file"` used by the Parameter Object and the Response Object] (#responseObject) to set the parameter type or the response as being a file.

Primitives have an optional modifier property `format` . Swagger uses several known formats to more finely define the data type being used. However, the `format` property is an open `string` -valued property, and can have any value to support documentation needs. Formats such as `"email"` , `"uuid"` , etc., can be used even though they are not defined by this specification. Types that are not accompanied by a `format` property follow their definition from the JSON Schema (except for `file` type which is defined above). The formats defined by the Swagger Specification are:

Common Name	`type`	`format`	Comments
integer	`integer`	`int32`	signed 32 bits
long	`integer`	`int64`	signed 64 bits
float	`number`	`float`	
double	`number`	`double`	
string	`string`		
byte	`string`	`byte`	base64 encoded characters
binary	`string`	`binary`	any sequence of octets
boolean	`boolean`		
date	`string`	`date`	As defined by `full-date` - RFC3339
dateTime	`string`	`date-time`	As defined by `date-time` - RFC3339
password	`string`	`password`	Used to hint UIs the input needs to be obscured.

You may have noticed that the examples above use lower camelCase in the resource naming conventions. There are many case variations out in the programming world. We settled on lower camelCase for several reasons:

- camelCase is more familiar to JavaScript developers and easier to parse
- camelCase is better aligned with our Java Object Data Models in use by Citi
- TIBCO best supports camelCase

One List to Rule Them All

- Conventions for path parameter and query parameter names:*
- Parameter names should be simple
- Parameter names should follow lowerCamelCase with following considerations:-
    - Wherever standard acronyms are available and applicable, fully defined names or non-standard acronyms/abbreviations for the same should not be used e.g. Use ‘url’*instead of* ‘uniformResourceLocator’

- Parameter names should contain alphanumeric characters only. No special characters are allowed.
- Parameter names should start with an alphabet and never with a numeric character.

Collections

A collection is a list of multiple Resources. A collection is represented as a JSON object.

A collection **MUST** contain a `resources` field. The resources field is an array containing multiple Resources.

A collection **MUST** contain a `pagination` field containing a pagination object.

Example

```
{
  "resources": [
    {
      "guid": "a-b-c",
      "created_at": "2015-07-06T23:22:56Z",
      "updated_at": "2015-07-08T23:22:56Z",

      "links": {
        "self": {
          "href": "/v1/accounts/a-b-c"
        }
      }
    },
    {
      "guid": "d-e-f",
      "created_at": "2015-07-06T23:22:56Z",
      "updated_at": "2015-07-08T23:22:56Z",

      "links": {
        "self": {
          "href": "/v1/accounts/d-e-f"
        }
      }
    }
  ],
  "pagination": {
    "total_results": 2,
    "total_pages": 1,
    "first": {
      "href": "/v1/accounts?page=1&per_page=10"
    },
    "last": {
      "href": "/v1/accounts?page=1&per_page=10"
    },
    "next": null,
    "previous": null
  }
}
```

Pagination

Pagination **may** be used by Collections to limit the number of resources returned at a time. Pagination is requested by a client through the use of query parameters. Pagination is represented as a JSON object.

Pagination **MUST** include a `total_results` field with an integer value of the total number of records in the collection.

Pagination **MUST** include a `total_pages` field with an integer value of the total number of pages in the collection.

Pagination **MUST** include the following fields for pagination links:

- `first` : the first page of resources
- `last` : the last page of resources
- `previous` : the previous page of resources
- `next` : the next page of resources

Pagination links **may** be `null` . For example, if the page currently being displayed is the first page, then `previous` link will be null.

When pagination links contain a URL, they **MUST** be a JSON object with a field named `href` containing a string with the URL for the next page.

The URL **MUST** include all query parameters required to maintain consistency with the original pagination request. For example, if the client requested for the collection to be returned in a specific order direction via a query parameter, then the pagination links must include the proper query parameter to maintain the requested direction.

The following query parameters **MUST** be used for pagination:

- `page` : the page number of resources to return (default: 1)
- `per_page` : the number of resources to return in a paginated collection request (default: 50)
- `order_by` : a field on the resource to order the collection by; each collection may choose a subset of fields that it can be sorted by

When collections are ordered by a subset of fields, each field **MAY** be prepended "-" to indicate descending order direction. If the field is not prepended, the ordering will default to ascending.

If there are additional pagination query parameters, the parameters **MUST** have names that conform to the acceptable query parameter names.

Example

```
"pagination:" {
  "total_results": 20,
  "total_pages": 2,
  "first": {
    "href": "/v1/accounts?order_by=-created_at&page=1&per_page=10"
  },
  "last": {
    "href": "/v1/accounts?order_by=-created_at&page=2&per_page=10"
  },
  "next": {
    "href": "/v1/accounts?order_by=-created_at&page=2&per_page=10"
  },
  "previous": null
}
```

Errors

Status Codes

The HTTP status code returned for errors **MUST** be included in the documented status codes.

This proposal includes `code` which would be an internal unique identifier of a class of error. The method for maintaining a list of these codes and their meanings would need to be determined

```
MISSING
```

Response Codes

Successful Requests

Status Code	Description	Verbs
200 OK	This status **MUST** be returned for synchronous requests that complete successfully and have a response body. This must only be used if there is not a more appropriate 2XX response code.	GET, PATCH, PUT
201 Created	This status **MUST** be returned for synchronous requests that result in the creation of a new resource.	POST
202 Accepted	This status **MUST** be returned for requests that have been successfully accepted and will be asynchronously completed at a later time. See more in the async section.	POST,PATCH,PUT,DELETE
204 No Content	This status **MUST** be returned for synchronous requests that complete successfully and have no response body.	DELETE

Redirection

Status Code	Description	Verbs
302 Found	This status **MUST** be returned when the API Gateway redirects to another location.	GET
303 See Other	This status **MUST** be returned when an async job finishes. It must include a location header containing the resource link. See more in the async section.	GET

Client Errors

Status Code	Description	Verbs
400 Bad Request	This status **MUST** be returned for requests that provide malformed or invalid data. Examples: invalid JSON, unexpected query parameters or request fields.	GET, PATCH, POST, PUT, DELETE
401 Unauthenticated	This status **MUST** be returned if the requested resource requires an authenticated user but there is no OAuth token provided, or the OAuth token provided is invalid.	GET, POST, PATCH, DELETE, PUT
403 Forbidden	This status **MUST** be returned if the request cannot be performed by the user due to lack of permissions. Example: User with read-only permissions to a resource tries to update it.	POST, PATCH, DELETE, PUT
404 Not Found	This status **MUST** be returned if the requested resource does not exist or if the user requesting the resource has insufficient permissions to view the resource.	GET, POST, PATCH, PUT, DELETE
422 Unprocessable Entity	This status **MUST** be returned if the request is semantically valid, but performing the requested operation would result in a invalid state. Example: Attempting to start an app without assigning a droplet.	POST, PATCH, PUT

Server Errors

Status Code	Description
500 Internal Server Error	This status **MUST** be returned when an unexpected error occurs.
502 Bad Gateway	This status **MUST** be returned when an upstream service failure causes a request to fail. Example: Being unable to reach requested service broker.

Asynchronicity

Endpoints are responsible for behaving either asynchronously (return 202) or synchronously (don't return 202).

For async endpoints:

`POST /v1/resource`

The CC will return a 202 with a location header pointing to the job. Depending on the resource, it may also return a skeletal body containing the partial resource.

```
202 Accepted
Location: /v1/jobs/123

{
  "message": "Task queued"
}
```

Before the job has completed, GET requests made to the job endpoint will return 200 with information about the status of the job.

```
GET /v1/jobs/123
```

```
200 OK

{
  "status": "in progress"
}
```

When the job has completed, GET request made to the job endpoint will return 303 and a location header to the resource (assuming it still exists).

```
GET /v1/jobs/123
```

```
303 See Other
Location: /v1/resource/:guid

{}
```

Note that for asynchronous deletes, the redirect location will be to a no-longer-existent resource.

Architecture

Citi initializr (pendent)

Testing

Unit tests

Tests that are executed on the application during the build phase. No integrations with databases / HTTP server stubs etc. take place. Generally speaking your application should have plenty of these to have fast feedback if your features are working fine.

Integration/Smoke tests -

Tests that are executed on the built application during the build phase. Integrations with in memory databases / HTTP server stubs take place. According to the test pyramid, in most cases you should have not too many of these kind of tests.

Smoke tests are executed on a deployed application. The concept of these tests is to check the crucial parts of your application are working properly. If you have 100 features in your application but you gain most money from e.g. 5 features then you could write smoke tests for those 5 features. As you can see we're talking about smoke tests of an application, not of the whole system. In our understanding inside the opinionated pipeline, these tests are executed against an application that is surrounded with stubs.

End to end tests

Tests that are executed on a system composing of multiple applications. The idea of these tests is to check if the tested feature works when the whole system is set up. Due to the fact that it takes a lot of time, effort, resources to maintain such an environment and that often those tests are unreliable (due to many different moving pieces like network database etc.) you should have a handful of those tests. Only for critical parts of your business. Since only production is the key verifier of whether your feature works, some companies don't even want to do those and move directly to deployment to production. When your system contains KPI monitoring and alerting you can quickly react when your deployed application is not behaving properly.