Home

Welcome to the home page for Alemba's RESTful API. This is your online home for all information on the API.

Why RESTful?

A RESTful API is an Application Programming Interface that follows REST (or Representational State Transfer) principles to allow one system to manipulate data on another. REST is a widely used protocol that provides a wide range of benefits over older protocols, such as performance, scalability and simplicity.

How is this different from the original vFire API?

The original WCF-based (Windows Communication Foundation) API is procedural – it exposes a limited set of operations that manipulate a limited set of entities, and you need to know the name of the operation that you want to perform in order to use it. By contrast, the RESTful API is entity centric – it exposes all the entities within vFire, and the verbs that can be used to manipulate those entities follow the same standard pattern. The RESTful API is also self-documenting, in that you can discover more detail about how to use the API from the API itself.

The result is a more flexible and more intuitive way of working with vFire data.

How do I get it?

If you have vFire 9.7 or later installed, you already have it. It comes as part of the standard install, and does not require additional installation, licensing or cost to use.

What you get

- 3 web services in 2 web sites in 2 app pools:
    - Alemba.Web
        - Authorization service – allows you to login and provides a token to use for all subsequent API calls
        - The API Explorer (see below)
    - Alemba.API – the main API
- 38 new tables of metadata in the vFire database
- API Explorer – an interactive UI for understanding the API, including live search facilities as well as detailed technical information regarding all the available entities and actions
- This guide.

Getting ready to use the API

The web services should be automatically configured as part of the install. However you are advised to check the settings in IIS, to confirm that the App Pool is configured for automatic recycling out of hours. You must install a suitable SSL certificate and enable SSL bindings on Alemba.Web and Alemba.API.

The base url will be **<servername>/<VirtualDirectory>/Alemba.Web**

That's it.

Getting Started

The best way to familiarize yourself with the API is through the API Explorer. This gives you an overview of the entities covered by the API, their properties, and what actions are supported per entity. It also provides a visual interface for viewing live data. For details see the API Explorer Guide.

Once you are familiar with the API, you can start using the API programmatically. See the Programmers' Guide for technical details.

For information on how to use the API to achieve business level tasks, such as raising a Service Order, see the Cook Book.

API Explorer Guide

What it does

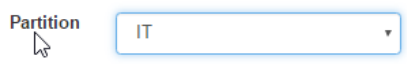
The API Explorer allows you to explore all the entities within the API. For each entity, you can

- See the attributes of all the entity's properties
- See all the actions that can be performed
- Browse the data in your test database

Logging in

You can log into the API Explorer using your vFire Id and Password, and specifying the "scope" – whether you are logging in as an Analyst or User. Both Analysts and User can log in and use the API, subject to the relevant privileges.

On login, the logged in user's name is displayed on the left. On the right you can select the current Partition, if the system is partitioned.

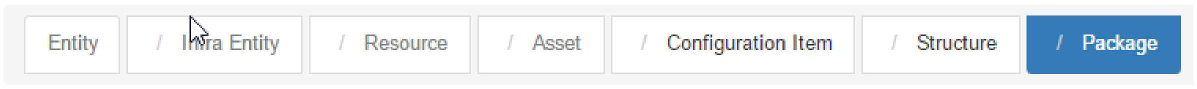

Partition: IT

Navigation

By default, the API Explorer opens at the Call entity (see API structure). All of the available entities appear in a tree in the Entities panel on the left. Physical entities appear in blue, logical ones (which you cannot select) in grey. Clicking on any of these relocates you at that object. You can expand or contract the tree to explore the entities.

Alternatively, use the Search field above the tree, which returns a list of matches that is refined as you type. If the item is currently not visible in the tree, it expands to show where that entity is located in the hierarchy.

Your current position in the entity hierarchy is reflected in the breadcrumb trail above that panel. Physical entities are in black, logical ones are greyed out. You can click on any physical entity in the breadcrumb trail to go to that entity. Currently selected entity is shown in blue.

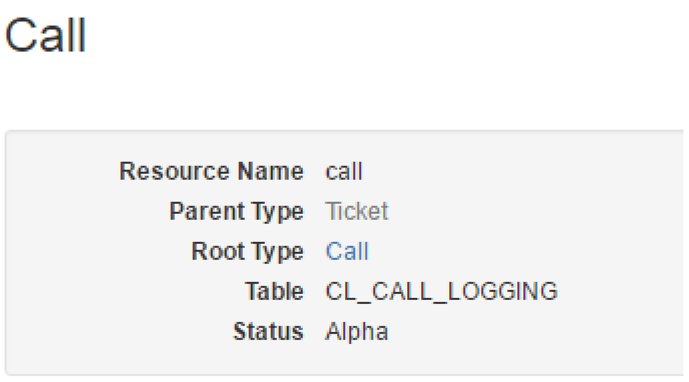

Entity / Infra Entity / Resource / Asset / Configuration Item / Structure / Package

Entity details

The rest of the API Explorer shows details about the selected entity.

Base information

At the top is the entity name. Below this, there is a brief description of the entity plus core information about the entity itself:



Call

Resource Name	call
Parent Type	Ticket
Root Type	Call
Table	CL_CALL_LOGGING
Status	Alpha

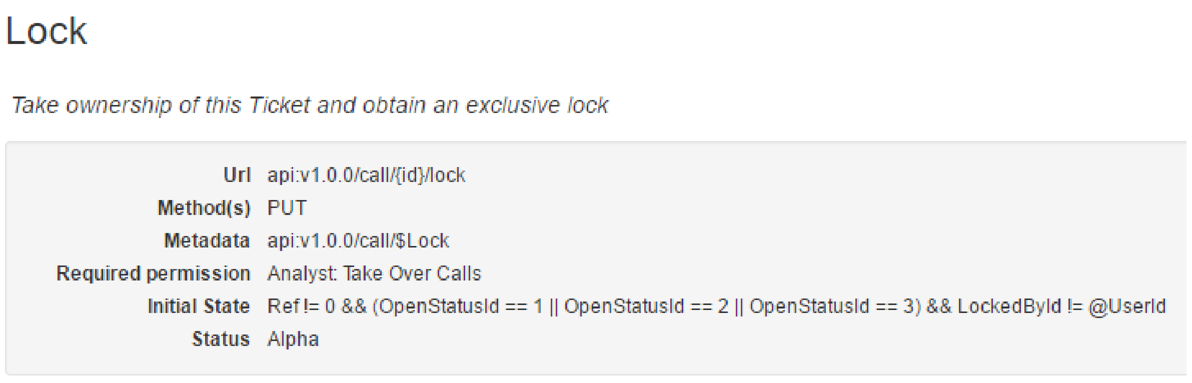
• (Description)	A brief summary of what the entity is
• Resource Name	Name used in url. If none, this is an abstract entity – a logical grouping for child entities, which is not accessible via the API
• Parent Type	Entity type of immediate parent
• Root Type	Entity type of entity from which this ultimately derives
• Table	If present, name of underlying database table
• Status	Indicates whether this entity is officially supported yet

Actions

Below the entity details, there is a list of actions for that entity. Selecting one will display information about the action.

Action

At the top is the action name and following this is further information about the action itself:



Lock

Take ownership of this Ticket and obtain an exclusive lock

Url	api:v1.0.0/call/{id}/lock
Method(s)	PUT
Metadata	api:v1.0.0/call/$Lock
Required permission	Analyst: Take Over Calls
Initial State	Ref != 0 && (OpenStatusId == 1 || OpenStatusId == 2 || OpenStatusId == 3) && LockedById != @UserId
Status	Alpha

Action information includes:

• (Description)	A brief summary of what this action does
• Url	The URL to invoke this action
• Method(s)	The supported HTTP method(s).
• Metadata	The link to the metadata for the action

• Required permission	Privilege needed to invoke this action, for both Analysts & Users
• Initial State	The conditions that need to be fulfilled for this action to be usable e.g. a call that you want to Update must either be in a New state, or must be in an Open* state and locked by you.
• Status	Indicates whether this action is officially supported for this entity yet

Below this, there may be between two and five further sections, depending on the selected action.

1. Action Parameters

For each action that can accept them, details are shown for each property and parameter that can be set as part of the action. This includes:

- (Label) User-friendly name. If Required, followed by an asterisk
- (Description) Textual explanation of property's purpose
- Path The path that identifies the location of the value in the object. For example, the path $action.Description translates to the JSON object { "$action": { "Description": "…" } }
- Data Type One of the supported Data Types
- Display Type An indication of Display Type
- Max Length Shown for fields if they have a fixed length
- Required Indicates a mandatory property or parameter
- Readonly Indicates if the property is readonly
- Default Value Value that will be used is none suppliedRules Details any special conditions that apply to the parameter. Examples include the Capitalization rule on Call Ref, or RequiredIfNotNull on IPK Status (both on Call.Create). Attributes of the rule include:
- (Description)
- Type – e.g. RequiredIfTrue
- Source – Field whose state impacts Target
- Target – Field impacted by state of Source
- Phase – Before Patch and Before Commit. In an update action, the API validates the input (Before Patch), retrieves the existing record and patches it, then validates the changes (Before Commit) before saving
- Scope – Client, Client/Server, or Server. Where the Scope is Client or Client/Server this is a hint at suggested client side behaviour. Where the Scope is Server this is an indicator at server side business rules.

2. Example Request

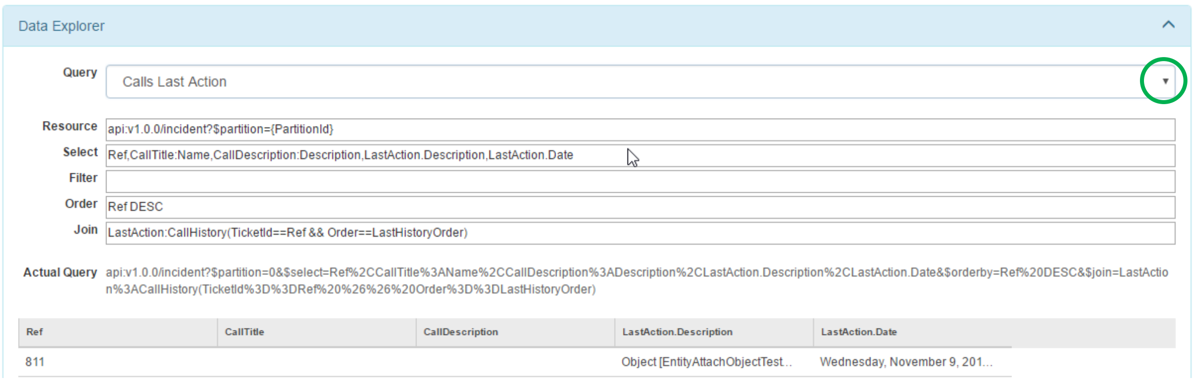
An example of the JSON (JavaScript Object Notation) that you would use to perform that action. This may include a section called $action (as in the Defer action), which can include parameters – values that control behaviour, as opposed to directly setting properties on that object.

3. Example Response

An example of the JSON typically returned on successfully performing an action, including mock data.

4. Data Explorer

For Search actions, this allows you to see data for the entity.



Query: Calls Last Action

Resource: api:v1.0.0/incident?$partition={PartitionId}

Select: Ref,CallTitle:Name,CallDescription:Description,LastAction.Description,LastAction.Date

Filter:

Order: Ref DESC

Join: LastAction:CallHistory(TicketId==Ref && Order==LastHistoryOrder)

Actual Query: api:v1.0.0/incident?$partition=0&$select=Ref%2CCallTitle%3AName%2CCallDescription%3ADescription%2CLastAction.Description%2CLastAction.Date&$orderby=Ref%20DESC&$join=LastAction%3ACallHistory(TicketId%3D%3DRef%20%26%26%20Order%3D%3DLastHistoryOrder)

Ref	CallTitle	CallDescription	LastAction.Description	LastAction.Date
811			Object [EntityAttachObjectTest...	Wednesday, November 9, 201...

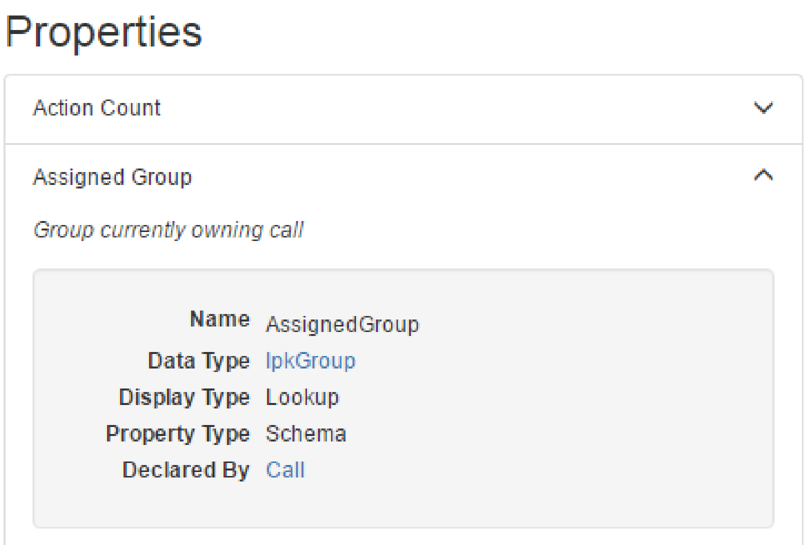
• Query	Build your own query ("Custom") or, where available, select a shipped example query
• Resource	The element of the url that identifies this version of the entity
• Select	The properties to return. A UI for the $select clause (see Searching)
• Filter	The properties to filter by. A UI for the $filter clause (see Searching)
• Order	The properties to order by. A UI for the $orderby clause (see Sorting)
• Left Join	A way to link to tables that are not already linked as part of the schema. A UI for the $leftJoin clause (see Joining)
• Inner Join	A way to link to tables that are not already linked as part of the schema. A UI for the $innerJoin clause (see Joining)
• Actual Query	The statement that is generated from the above parameters
• (Results table)	The results. By default just shows the first 10 fields. More data retrieved on demand as you scroll down.

5. Augmenters

A list of the augmenters that are relevant to this entity and action. The following attributes are shown per augmenter: Name, Description, Example, Version, and Augmenter Type.

Properties

The vertical panel on the far right shows the properties that are available for the entity and their attributes, in the context of the selected action.



Group currently owning call

Name	AssignedGroup
Data Type	IpkGroup
Display Type	Lookup
Property Type	Schema
Declared By	Call

• (Label)	this is the user-friendly display name
• (Description)	a brief description of the property
• Name	the name of the property, as used by the API programmer
• Data Type	one of the supported data types – see here. If the Data Type is another Entity Type, the value can be used as a link to that Entity Type
• Display Type	one of the supported display types – see here
• Property Type	Contains: • ”Schema” for out-of-the-box properties, or • ”Extension” for fields created using Designer
• Declared By	This is the entity type from which this property is inherited.

Note that if you add new fields using vFire Core Designer, you will need to restart the Alemba.API web service to make them appear in the API Explorer.

Programmers’ Guide

API structure

The RESTful API is built on top of a schema that encompasses the primary data entities in the vFire system. Additionally a number of logical entities have been added that allow similar entities to be grouped together. The hierarchical structure allows sub-entities to inherit common properties, allowing for consistent meaning and behaviour.

Languages

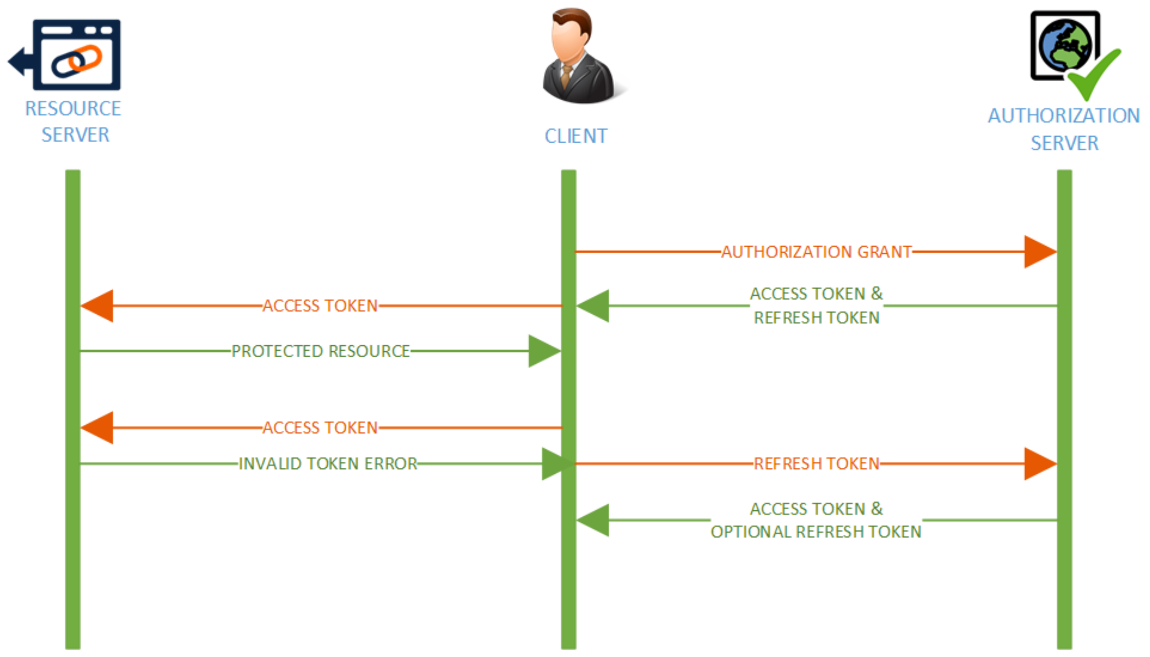
The API is platform independent - it can be accessed from any kind of web client, using a range of languages. The API Explorer provides language neutral examples of the structures sent and received as part of a web request/response, for each action, for each entity.

Authentication

The API uses standard oAuth 2.0 authentication, via /alemba.web/oauth/login.

The authentication workflow issues a short-lived Access Token and a longer-lived Refresh Token on Login. The Access Token is used for authentication on the REST API. The Refresh Token is used to renew the Access Token and therefore maintain the session. The Refresh Token allocates a vFire Core Session and consumes a licence.

The process is illustrated below, flowing from the top downwards.



When the Refresh Token is used to renew an Access Token, a new Refresh Token is also issued, and the vFire Core Session is extended. The used Refresh Token becomes invalid and should be discarded. Clients should store the new Refresh Token for subsequent usage. If the vFire Core Session expires or is removed, the Refresh Token will become invalid. If the Refresh Token expires, the vFire Core Session is terminated.

The single use Refresh Token and short lived Access Token ensures that compromised tokens quickly become invalid - protecting the security of individual users, and preventing unauthorized Access Token reuse.

The Access Token must be presented in the Authorization header of the HTTP request:

Authorization: Bearer <Access Token>

Note that the Authorization service supports the following oAuth 2.0 Grant Types:

- Password - Authenticate using a username and password
- client_credentials - Authenticate using integrated security
- refresh_token - Authenticate using an existing Refresh Token

Logging in

Below is an example of the code used to login with a username and password:

```
function passwordLogin() {
        var args = {
            client_id: "clientid",
            grant_type: "password",
            scope: "scope",
            password: "username",
            username: "password"
        };
        var xhr = $.ajax({
            url: 'alemba.web/oauth/login',
            type: "POST",
            data: args,
            contentType: 'application/x-www-form-urlencoded'
        });
        xhr.done(onGrantSuccess).fail(function (err) { return 
onGrantFailure(err, "password"); });
        return xhr;
    }
```

Or, to refresh your access token:

```
function refreshTokenLogin(refreshToken) {
        var args = {
            client_id: "clientid",
            grant_type: "refresh_token",
            scope: "scope",
            refresh_token: refreshToken
        };
        var xhr = $.ajax({
            url: 'alemba.web/oauth/login',
            type: 'POST',
            data: args,
            contentType: 'application/x-www-form-urlencoded'
        });
        xhr.done(onGrantSuccess).fail(function (err) {
            if (err.status == 401) {
            }
            else {
                onGrantFailure(err, "refresh_token");
            }
        });
        return xhr;
    }
```

Note that **scope** should be set to **session-type:Analyst** or **session-type:User**. It is case sensitive.

Login Responses

For successful Logins the response will be:

```
{
    expires_in: number,     // number of seconds until access_token expiry
    access_token: string,   // token used for data access
    refresh_token: string,  // token used for access_token renewal
    scope: string,          // The actual scope of the token

}
```

Note that there is a new refresh_token in the response. The old one will no longer work and must be discarded. It is acceptable to renew your access_token before it is due to expire, but you must not do so with every request.

If the authorization request is not successful, clients can expect to receive a suitable HTTP response code and JSON formatted data containing an error code.

The response data may also include error_description, which gives the developer a clue as to the precise cause of the failure.

```
{
    error: string,               // one of the oauth 2.0 error codes
    error_description: string,   // a description of the error if applicable
}
```

In these cases, the response is deliberately vague so as to protect the integrity of the authorization server.

HTTP Status Code	Error Code	Reasons
401	invalid_client	The client_id is incorrect or the client is not enabled
400	invalid_grant	The credentials are not correct, the user is not allowed to login

If you receive a 401 response, it is because your access token has expired, and you must refresh it using the refresh token. If you receive a 401 response when using a refresh token, you must login again with username and password.

Logging out

You can logout as follows:

```
    function logout() {
        var deferred = $.Deferred();
        var args = {
            token: exports.grant.refresh_token
        };
        var xhr = $.ajax({
            url: 'alemba.web/oauth/login',
            type: "POST",
            data: args,
            contentType: 'application/x-www-form-urlencoded',
            headers: {
                "Authorization": "Bearer " + exports.grant.access_token
            }
        });
        xhr.done(function () {
            //Logout success
            deferred.resolve();
        }).fail(function (err) {
            switch (err.status) {
                case 404:
                    deferred.resolve();
                    break;
                case 400:
                    deferred.reject("Invalid token");
                    break;
                case 401:

refreshTokenLogin(exports.grant.refresh_token).done(function () {
                        //We've successfully refreshed the access token
                        //Now we can try to invalidate the refresh token
again
                        logout().done(function () {
                            deferred.resolve();
                        }).fail(function (err) {
                            //Logout is still not working.
                            //The session may still be active and may still
be consuming a license.
```

```
                            //The session can be terminated by an 
administrator from logon control, so this error should be reported
                            deferred.reject(err);
                        });
                    }).fail(function () {
                        //If you cant log in its because the refresh token 
has expired.
                        //This can be considered a successful logout
                        deferred.resolve();
                    });
                    break;
                case 403: //Not allowed, probably because the refresh token 
is not related to the access token
                default:
                    deferred.reject(err.responseJSON.Message);
                    break;
            }
        });
        return deferred.promise();
    }
```

Logout Responses

Logout will give one of the following responses:

200: Logged out successfully.

400: Token invalid or missing.

401: Not authorized because it's not been possible to validate your ownership of the refresh token. In this case you must refresh the access token and try again.

403: That refresh token doesn't belong to you.

404: The refresh token is valid but has already been removed. Maybe you logged in elsewhere.

Base RESTful API methods

The following HTTP verbs are used by the API to perform the listed actions.

- Create POST
- Read GET
- Update PUT
- Delete DELETE

(Note that neither soft nor hard deletion have been implemented in the first release)

A programmatically discoverable API

The API can be used to return information about itself, in the form of hypermedia – machine readable descriptions and links to further similar information, allowing a developer to progressively explore the breadth and depth of the API for themselves. What is more, these descriptions are used by the API itself, guaranteeing that this "documentation" is always current.

For example, to discover root level information on the scope of the API, invoke …

http://localhost/alemba.api/api?$metadata&$options

Note that this response will also be returned for any request which does not specify a resource, for example:

GET http://localhost/alemba.api

GET http://localhost/alemba.api/api

GET http://localhost/alemba.api/api/v1

The metadata will be returned in JSON format and will contain links to the metadata for top level entities exposed by the API, for example:

```
{
    "_links": {
        "Approval": [
            {
                "_self": "api:v1/approval/$metadata"
            }
        ],
        "Call": [
            {
                "_self": "api:v1/call/$metadata"
            }
        ],
        ...
    },
    "description": "A description of the API, the links at this level and
the ResourceDescriptor response type."
}
```

All metadata responses may include the following properties
"children":
An array of descendant types for the current response.e.g.Call may list children including Incident
The metadata for each child only includes "_self"

"description": A description of the current metadata response

"name": The name of the entity that is the subject of the metadata response

"properties":
An array of property descriptions for the subject entity.
These properties provide the minimum information required to understand the data model and basic constraints.

Each entry in the array may include the following properties.
"name":
The name of the entity property.

"displayName":

The default display name of the property.This could be used in table column headers or form fields.

"type":
A description of the data type of this property.
The type property is a complex type which has the following properties

"displayTypes":
The suggested display types for this property

"dataType":
The type of data this property represents

"class":
The kind of property

"description":
A description of the purpose of the property.

"usage":
Internal when the property is used for internal business logic, otherwise Public

"isKey":
The property represents the unique identifier (primary key) for the entity

"noSearch":
When true, this property is not supported in searches

"defaultValue":
An indication of the default value for this property

"length":
The maximum length of this field.Text fields only.

"uppercase":
When true, this indicates that the Text value will be capitalized.Non capitalized input may cause validation errors in a future release.

"status":
Indicates the current release status of the subject entity. "Alpha", "Beta", "GA"

"_actions":
A hash map of action name and an array of action metadata

"_context":
A reference to the entity metadata of a record or of the metadata of the parent of the subject entity

"_links":
A hash map of name and an array of link metadata.These links must be requested using the http verb GET

"_self":
  A reference to the current response

link and action metadata will always include a link to _self and will often include a "href" property
The "href" may be templated, as denoted by the syntax {id}. The templated values must be replaced by the client.
e.g. {id} in "api:v1/call/{id}" should be replaced with the Ref of a call.
{id} always indicates the primary key field for the target entity. All other entity properties may also be referenced in the template. e.g. {Partition} where Partition is the name of the property in the entity referenced by "_context".

"_context" and "_self" will always define a medialink to an API resource. For brevity, the links are prefixed with "api:"
Clients should replace this prefix with the actual API base url
e.g. Given an api base url of http://web-server/core-system/alemba.api/api, api:v1/call/$metadata should be interpretted as http://web-server/core-system/alemba.api/api/v1/call/$metadata
All API medialinks can be invoked using the $options suffix.

These links can then be followed to explore further details about the entity, what it is and what it can do.

All API entities support simple and predictable RESTful actions. e.g.

"api:v1/call" supports GET for searching and POST for create

"api:v1/call/1" supports PUT for update and GET to get that instance of a call.

Discovering the entity actions

Many entities also support more complex actions, such as forward.

These actions are typically accessed using **"api:v1/call/1/forward"**

Details of the required inputs and supported HTTP method can be found in the metadata for that action.

For example the Call Create action metadata can be accessed with

http://localhost/alemba.api/api/call/$create?$options

This metadata gives you info about the Create action, including a list of mandatory and optional properties and parameters. $create in the above example can be substituted with any action that is supported by the Call entity.

What is more, you can use the same principles not just as an aid to programming, but at runtime too. The metadata that is returned about a specific object contains links for the list of actions *for that object instance in its current state.* (An exception is the Search action, which will only return links to the relevant Get action for each record.) For example you will only see the Reopen action if a Call is in a closed state.

Searching

The Rest API supports expressive searching of most entities

The query parameters and syntax applies to all Search actions for all entities.

To start, it is possible to simple request a resource using HTTP GET.

```
GET api:v1/call
```

This will return a reference to all accessible calls in JSON format

```
{
  "results": [
    {
      "_context": "api:v1/call/$metadata",
      "_self": "api:v1/call/3"
    },
    {
      "_context": "api:v1/incident/$metadata",
      "_self": "api:v1/incident/4"
    }
  ],
  "_self": "api:v1/call?$top=2147483647",
  "__count": 275
}
```

The response contains the following properties

- "results": This is an array of search results. Each result will always include a _context url (so you know what it is) and a _self url (so you know how to get that item).
- "_self": This is a url refering to the current response

Notice that the _self url includes a query string parameter

```
$top=2147483647
```

The search actually ran without any row limit. It will try to return every row (accessible to the current session).

This query string parameter is added to the response as a hint.

Paging

The Rest API supports flexible paging of search results.

$top accepts a positive signed integer value (Int32) and is used to define a row count.

```
GET api:v1/call?$top=30
```

This will limit the response to the top 30 records

$orderby accepts a comma separated list of property names and optional sort direction (see $select for more details on property names) and is used to define the order of the results

```
GET api:v1/call?$orderby=Ref
```

This will return all Calls ordered by Ref

By default the orderby clause will be applied in ascending order, but this can be overriden

`GET api:v1/call?$orderby=Ref` **`asc`**

This will return all Calls ordered by Ref in ascending order

`GET api:v1/call?$orderby=Ref` **`desc`**

This will return all Calls ordered by Ref in descending order

$skip accepts a positive signed integer value (Int32) and is used to define a number of rows to skip

e.g. `GET api:v1/call?$top=30`

This will limit the response to the top 30 records

and `GET api:v1/call?`**`$skip=30`**`&$top=30`

This will skip the first 30 records and return the next 30 (ie rows 31 to 60)

These parameters can be combined (in any order) to control page size and contents

e.g. `GET api:v1/call?$top=30&$skip=30&$orderby=Ref desc`

This will return the top 30 Calls ordered by Ref in descending order

The API also supports counting, which can be used to calculate the total number of pages

$count must be set to true and is used to instruct the api to return a count only

`GET api:v1/call?`**`$count=true`**

This will return a text/plain response containing a number which represents the total number of rows.

Alternatively, **$inlinecount** can be used to have the count be returned with a set of results

`GET api:v1/call?$top=30&`**`$inlinecount=true`**

```
{
  "results": [
    {
      "_context": "api:v1/call/$metadata",
      "_self": "api:v1/call/3"
    },
    {
      "_context": "api:v1/incident/$metadata",
      "_self": "api:v1/incident/4"
    }
  ],
  "_self": "api:v1/call?$top=2147483647",
  "__count": 275
}
```
Note that "__count" is included in the response body.

Best Practice

Use of these paging features is critical for individual client and application wide performance and should be utilized by all API consumers for all searches.

Even where it is assumed that there are only a handful of records.

Note, that using $inlinecount results in two executions of the database query. One to get the count and one to get the result set.

Therefore it is important that this parameter is not added to every request

Selecting columns

The Rest API supports configurable column selection in search results.

$select accepts a comma separated list of property paths to include in the search results

```
GET api:v1/call?$select=Ref,Description
```

This instructs the API to include Ref and Description in the search results

```
{
  "results": [
    {
      "Ref": 3,
      "Description": "Microsoft Windows 2000 needs to be installed across
all client machines.",
      "_context": "api:v1/call/$metadata",
      "_self": "api:v1/call/3"
    },
    {
      "Ref": 4,
      "Description": "Cannot access intranet.",
      "_context": "api:v1/incident/$metadata",
      "_self": "api:v1/incident/4"
    }
  ],
  "_self": "api:v1/call?$select=Ref,Description&$top=2147483647"
}
```

Note that as well as retrieving properties from the entity, you can directly retrieve properties from related entities. This is extremely powerful. In the old WCF API, you would first need to get the raw foreign key ref from the main entity and then do a lookup on the related one, or you would have to write your own custom query, including joining objects and ensuring that the related object is not locked. The RESTful API is built on an underlying schema that takes care of these complexities and allows you to get the data you need in the simplest possible way. In fact you can traverse the entity relationships as far as you need, so getting a call's service's user's email can be achieved with

```
GET api:v1/call?$select=Ref,Description,Priority
```

This will add the Priority ref to the results

```
{
  "Ref": 4,
  "Description": "Cannot access intranet.",
  "Priority": 3,
  "_context": "api:v1/incident/$metadata",
  "_self": "api:v1/incident/4"
}
```

It is possible to select linked fields from relation type properties

```
GET api:v1/call?$select=Ref,Description,Priority.Name
```

```
{
  "Ref": 4,
  "Description": "Cannot access intranet.",
  "Priority": {
    "Name": "Priority 3",
    "_context": "api:v1/call-priority/$metadata",
    "_self": "api:v1/call-priority/3"
  },
  "_context": "api:v1/incident/$metadata",
  "_self": "api:v1/incident/4"
}
```

Notice that the linked entity includes metadata links

This also applies to linked fields from linked relations.

```
GET api:v1/call?$select=Ref,Description,Service.Location.Name
```

Returns

```
{
  "Ref": 4,
  "Description": "Cannot access intranet.",
  "Service": {
        "Location": {
          "Name": "San Francisco",
          "_context": "api:v1/location/$metadata",
          "_self": "api:v1/location/9"
        },
        "_context": "api:v1/service/$metadata",
        "_self": "api:v1/service/1"
  },
  "_context": "api:v1/incident/$metadata",
  "_self": "api:v1/incident/4"
}
```

Property paths in $select can be aliased using a prefix to simplify the response

```
GET api:v1/call?$select=Ref,Description,LocationName:Service.Location.Name
```

```
{
  "Ref": 4,
  "Description": "Cannot access intranet.",
  "LocationName": "San Francisco",
  "_context": "api:v1/incident/$metadata",
  "_self": "api:v1/incident/4"
}
```

$select will also accept named Extension Augmenters

```
GET api:v1/call?$select=Ref,@AssignmentState
```

This will add the computed assignment state to the response

```
{
  "Ref": 4,
  "AssignmentState": "Assigned to Me",
  "_context": "api:v1/incident/$metadata",
  "_self": "api:v1/incident/4"
}
```

To help with orientation during development, $select will also accept *. This will return all properties for the entity.

$select=*

Best Practice

$select=* is only intended for development and should not be used in production.

Selecting values from extension fields is supported, but carries a significant overhead and so should be avoided.

Filtering

The Rest API supports expressive filtering of search results

$filter accepts a C# LINQ style predicate which is translated to parameterized SQL and applied as a search filter

`GET api:v1/call?$filter=Priority==1`
This would return all accessible calls where the Priority is equal to 1

Equality Operators and Methods

All data types support basic equality comparison

== Is equal to

= Is equal to

!= Not equal to

Binary data types support basic equality comparison but in practice, this can only be used to compare the property value with null.

`GET api:v1/call/1/attachment?$filter=BinaryData!=null`

Boolean data types support basic equality operators

When comparing with true or false, the right hand side of a boolean property equality expression can be omitted.

Binary equality expressions can also be negated with !

`GET api:v1/person?$filter=IsLoggedIn==true`

is equivelant to `GET api:v1/person?$filter=IsLoggedIn`

or `GET api:v1/person?$filter=IsLoggedIn==false`

is equivelant to `GET api:v1/person?$filter=!IsLoggedIn`

DateTime data type filters must be used with one of the applicable augmenters

`GET api:v1/call?$filter=CreatedDate>@DateTime(2017-01-01T00:00:00.000+1)`

This will return all Calls which were created after Midnight on January 1st 2017 (UTC+1)

Note that the date value must be expressed in ISO8601 format

The @Now augmenter can be used to compare a date value with the current time.

```
GET api:v1/call?$filter=CreatedDate==@Now
```

This is most useful where a query will be designed and then subsequently reused.

The @NowOffset augmenter can be used to compare a date value with the current time and an offset expressed in days hours and minutes

```
GET api:v1/call?$filter=CreatedDate>@NowOffset(0,-1,-30)
```

This will return Calls created in the last 0 days, 1 hours and 30 minutes (in the last hour and a half).

As with all filters, the expressions can be combined using logical And (`&&`) and Or (`||`) operators

```
GET api:v1/call?$filter=CreatedDate>=@DateTime(2017-01-
01T00:00:00.000+1)&&CreatedDate<=@NowOffset(0,0,-30)
```

This will return Calls created between Midnight on January 1st 2017 (UTC+1) and half an hour ago.

Dates do not have to be defined in UTC format, but MUST include the timezone.

If no timezone is supplied, dates are assumed to be in local server time.

Text and **RichText** data types support basic equality and can also be used with some string comparison methods

The Contains method can be used to match records where a string property contains a word or phrase

```
GET api:v1/call?$filter=ShortDescription.Contains("email")
```

The StartsWith method can be used to match records where a string property starts with a word or phrase

```
GET api:v1/call?$filter=ShortDescription.StartsWith("email")
```

The EndsWith method can be used to match records where a string property ends with a word or phrase

```
GET api:v1/call?$filter=ShortDescription.EndsWith("email")
```

As with all filters, the expressions can be combined using logical And (`&&`) and Or (`||`) operators

```
GET api:v1/call
?$filter=hortDescription.Contains("email")||ShortDescription.StartsWith("ou
tlook")
```

Number data types support more complex equality comparison operators

>	Greater than
<	Less than
>=	Greater than or equal to
<=	Less than or equal to

```
GET api:v1/call?$filter=Number1>=3
```

Combining Expressions

Filter expressions can be combined using logical And (`&&`) and Or (`||`) operators and can be grouped using parentheses `(` and `)`

```
GET api:v1/call?$filter=((Number1>=3
||Number2==1)&&(Priority==3||Priority==1))||(CreatedDate>@NowOffset(0,0,-
30)&&@IsAssignedToMe)
```

Note that query string parameter values must be url encoded

```
GET api:v1/call
?$filter=((Number1%3E%3D3%E2%80%8B%7C%7CNumber2%3D%3D1)%26%26(Priority%3D%3
D3%7C%7CPriority%3D%3D1))%7C%7C(CreatedDate%3E%40NowOffset(0,0,-
30)%E2%80%8B%E2%80%8B%E2%80%8B%26%26%40IsAssignedToMe)
```

Joining

You can use **$join** to link to entities that are not already linked as part of the schema definition. This is the case with entities that have multi-column keys e.g. CallHistory, which has a composite key of Ref and LastHistoryOrder. Once you create such a join, you will want to refer to properties of that joined entity, so part of the definition of the join is an alias for that join, in the format

Alias:TargetEntity(TargetProperty==SourceProperty)

… where the clause in brackets can occur multiple times. For example:

&$join=LastAction:CallHistory(TicketId==Ref && Order==LastHistoryOrder)

You can then use the alias in your select clause just as if it were a property of the entity with a Data Type that is the target entity, e.g. **LastAction.Description**.

Note that this technique should be used sparingly, because as with extension fields the relationship is not indexed and so may result in reduced performance.

Case sensitivity

Note that everything after any of the $ functions above is case sensitive.

Security

Handling partitions

Where entities are Partitioned, the results returned are automatically limited to those in partitions accessible to the current session. Therefore it is not necessary to apply partition filtering, however if desired a specific partition can be specified in one of two ways.

1. $filter=Partition==1
   This will return records where the Partition Ref equals 1. The filter will be applied even if the entity is not partitioned
2. $partition=1
   <u>This is the recommended method </u>and will work as above, but will only apply the partition filter if the entity is actually partitioned. This will also account for variable partitioning of Asset types e.g api/v1/asset?$partition=1 will return a combination of Services, ConfigurationItems, etc. where Service may be partitioned, but ConfigurationItem is not.

Data Types

The following data types are exposed by the API:

RESTful API	**Equivalent in WCF API**
Binary	Byte Array
Boolean	Boolean, Yes/No
DateTime	Date/Time
Number	Integer (including Double and Float)
Text	String
RichText	String

Data types can also be entities. For example, the Service properties on a Call has a Data Type of Service (Lookup in the metadata json). This means that it contains the key value of the associated object – click on the circle icon in the API Explorer to see what that is. (This is equivalent to the WCF data type “Lookup”.)

Note that Boolean property standardises underlying data inconsistencies. Across various tables, flags are stored as `"Y"`, `"YES"`, `"T"`, `"TRUE"`, `"ON"`, `"1"`, `"P"`, `"A"` – for all of these, the API will return **true**, and conversely will convert **true** to appropriate values on PUT and POST.

Display Types

The following table shows the types that are supported, and the related data types.

<u>Display Type</u>	<u>Data Type</u>
Checkbox	Boolean
DatePicker	DateTime
DateTimePicker	DateTime

Lookup	entity
MultiLookup	entity
MultiSelect	entity
ListBox	entity
Numeric	Number
Password	Text
RichText	RichText
Select	entity
Text	Text
TextArea	RichText
TieredSelect	Entity

Action Versioning

Each release of vFire includes a version of the API that is built to work with that release. The API will have its own version number which is not directly related to the vFire release version number.

Within each release, there may be new actions, extensions to existing actions, and fixes. Each action will have its own version number, in the format **major.minor.patch**[1], starting at 1.0.0 for actions that are officially released, which will be incremented independently to reflect how it has been changed, as follows:

- major = incompatible API change made
- minor = functionality added in a backwards-compatible manner
- patch = backwards-compatible bug fixes

The benefit of this approach is that users of previous versions of the API can gain a clear understanding of whether any of the parts of the API that they are using have changed, and the nature of that change, simply by comparing version numbers of each used action.

You should therefore build the appropriate version number into the url for each action, depending on how you want to handle potential changes.

If you only specify the major version – v1 – this means that:

- your code should continue to function without modification
- you will be able to take advantage of any functional extensions
- you will use the latest patches

If you specify the major and minor versions – v1.0 – this means that:

- your code should continue to function without modification
- you will use the latest patches

If you specify all three parts – v1.0.0 – this means that:

- your code should continue to function without modification
- you will always use that specific version of that action

[1] The use of a format with specific *meaning* associated with the increase of each part of the version number is known as *semantic* versioning, see http://semver.org/

Alpha functionality

Note that within any release individual entities and/or actions may be flagged with a status of Alpha. These are **pre-release**, and are provided without warranty and subject to change. **Alemba takes no responsibility for any adverse effects as a result of their use.**

Augmenters

The API includes a set of inbuilt functions that simplify the retrieval of complex data using Search actions. This allows API users to easily select, view and manipulate data using business-level concepts, rather than dealing with the low level data that sometimes needs gathering from many sources to provide that information. These functions are known as augmenters. These can be used as virtual variables, for example in a filter clause. So if you only want to see calls that have breached, you would say:

GET http://…/api/v1.0/call?$filter=@SlmBreached

There are a number of different types of Augmenters:

- Condition Augmenters – for use with $filter, encapsulating complex search conditions (including joins).
- Function Augmenters – provide additional functionality to the queries and are entity-independent
- Token Augmenters – return simple values for use in query filters and are entity-independent
- Extension Augmenters – computed properties, properties selected from other tables with join, etc.

The full list of available Augmenters is documented within the API Explorer.

Error Handling

The API will return errors with an appropriate HTTP Status Code and a message with the following structure:

Message:	string - This is the error message
Type:	string - This is the type of the error – usually just the Exception type name
SubStatus:	string - This is a string which will help to narrow down the reason for the HTTP Status Code

Current SubStatus values are:

```
None, ResourceNotFound, RecordNotFound, LinkedRecordNotFound,
NotSupported, NotImplemented, NotAllowed
```

Programmatically, you should rely upon the HTTP Status Code and the SubStatus.

The HTTP Status Codes that are explicitly returned are: 200, 304, 400, 401, 403, 404, 405, 415 and 500.

Note that often a 404 response will include a message body (as described above). This is so that you can tell the difference between a badly formed url and a non-existent record.

Cook Book

These pages contain information about how to use the API to create well-formed Alemba objects, i.e. objects in the vFire database that contain the data and relationships necessary for those objects to behave in a manner that is compatible with objects created by the suite of vFire applications.

The role of the Alemba rules engine

The business logic that determines how objects behave after creation is contained in the Alemba rules engine (also known as Infra Rules). Some examples of what it does are:

- The initial routing of calls
- The creation and progression of tasks for new requests
- The application of SLAs and other agreements
- Creating history entries

This engine is invoked when objects are created via the API, so the rules therein will be automatically applied, just as they are for objects created via vFire interfaces. This simplifies using the API in a large number of cases – all the API user has to do is to set the values that will trigger those rules and the engine will take care of the rest. The key properties and parameters that impact behaviour are covered in the recipes.

Processes that don't use the rules engine

However there are some processes which must operate across multiple transactions and require user interaction. In these cases, the API user has more work to do – to create things in the right order, and to link the objects correctly. One such example is the creation of Service Orders - details for how to do this are here.

Implications of the difference between a WCF and RESTful API

In the old WCF (Windows Communication Foundation) API, some transactions include parameters that not only impact the main subject of the transaction, but also create, link or update other related objects. This kind of hybrid action is not appropriate in an API whose nature is to enable state transfers, rather than to implement compound rules-driven behaviour. The same ends can be achieved in other ways, and these are described in the cookbook.

Cooking principles

Lock before you update

The following example refers to Call, but applies to all lockable items, such as Assets and Tickets (Calls, Requests and Tasks) and their sub-types.

Before performing an update in vFire Core, you need to Take Action in Core, which as well as assigning the call to yourself. The Core user interface also locks the call, until you close the call screen. You need to follow equivalent principles when using the API to ensure consistent operation with the Core application

Note that in the API, the actions available to you will depend on the state of the record, and the subset of actions that you *should* attempt are subject to your privileges.

When you Get a record …

- If it is not locked,
  - the only PUT/POST action described will be Lock.
- If it is locked by someone else,

        - the lock action may fail. The response will tell you who has locked that record.
          e.g {
            "Message": "Record locked by Jessica Mercato",
            "SubStatus": "None",
            "Type": "HttpStatusException"
          }
- If it is already locked by you,
    - all PUT/POST actions that are appropriate to the record in its current state will be described including Unlock (but not Lock)

Other actions may work, even if they are not listed for the current state of the entity. But there is a higher likelihood that they will fail.

When you Lock a record …

- If the lock does not succeed – (because someone else has locked it in the meantime and you do not have the Take Over Calls privilege)
    - You should retry a limited number of times, and if still failing, either report an error or try again later, according to the nature of the activity
    - If you succeed during a retry, you should Get the record again and check its timestamp (LastActionDate)
        - If changed, validate that it is still in a state where it makes sense to apply your changes, and proceed if so
- If the lock succeeds, all PUT/POST appropriate actions will be available

Now you can perform the desired action

If the action fails (because someone else has locked it in the meantime) …

- Note how many times the action has failed and limit action retries to avoid endlessly looping
- Go back to trying to lock the record as above

If the action succeeds …

- If you need to perform further actions on the same record, proceed with those actions
    - Note that some actions – e.g. Defer, Forward, Close – automatically perform an Unlock too. If you need to perform further actions after this, you may need to lock the object again. The response to any action, also includes a list of suggested actions applicable to the current state.
- If you have finished with the record, you must ensure that it is unlocked – the API will not do this automatically until the current session expires. Although unlocking is implied by some ticket actions (Defer, Forward, Close)

(Note that locked records will eventually be unlocked when the current session expires, just as in vFire Core)

This follows the same pattern as implemented across the vFire product set, and should not cause unexpected behaviour in those products.

Recipes

Creating Calls

To create a call, you must choose a Call type, determine the url for the Call Create action, create a properly formatted Call object, serialize the object in JSON format and submit the object using HTTP

POST.
Finally, the created Call may need to be submitted before it becomes visible to other system users.

Choosing a Call type

Each Call Type may be associated with its own extension fields.
These extension fields may not all be visible on all call types., so the recommended approach to choosing a call type is to derive this from ITIL IPK Tiers or Call Templates - if these are enabled.

IPK Statuses and Streams

In the Core application, users choose a type of call to log by choosing a screen set
This is often determined by selecting an IPK Status and then an IPK Stream, which in turn are mapped to a Screen Set.
It is also possible to link Screen Set to a combination of IPK Status, IPK Stream add Call Problem Type

These Call Type mappings are accessible through either api:v1/ipk-status-stream-to-type or api:v1/ipk-status-stream-type-to-type
Only one of these endpoints will be enabled. The enabled one will allow searching and the other will respond with HTTP Status code 404
To programatically determine which to use, each endpoint can be queried using $options on the end of the query string.

Search the enabled call type mapping entity and choose one according to IPK Status, Stream and/or Call Problem Type

The seach results from each include the usual hypermedia links, and also include "_actions". For example

```
{
  "IpkStatus": {
        "Name": "Incident",
        "_context": "api:v1/ipk-status/$metadata",
        "_self": "api:v1/ipk-status/1"
  },
  "IpkStream": {
        "Name": "Standard",
        "_context": "api:v1/ipk-stream/$metadata",
        "_self": "api:v1/ipk-stream/0"
  },
  "_context": "api:v1/ipk-status-stream-to-type/$metadata",
  "_self": "api:v1/ipk-status-stream-to-
type?$select=IpkStatus,IpkStream&$filter=(IpkStatus%3d%3d1%26%26IpkStream%3
d%3d0%26%26InfraEntityType%3d%3d7)",
  "_actions": {
        "Create": [
          {
                "_self": "api:v1/incident/$Create",
                "href": "api:v1/incident",
                "description": "The resource that should be used to create
a call of this type. IpkStatus and IpkStream from this result should be set
in the new call."
          }
        ]
  }
}
```

Note that "_actions" contains a reference to a Create action.
The Create action includes a hypermedia link to itself and a "href". This "href" is the link to use if you want to Create a Call of this type.
Note also the additional information in "description".

In this example, we would choose to create a call using

```
api:v1/incident
```

Call Templates

In the Self Service Portal, users choose a call template, which in turn is associated with a type of call

Call Templates can be searched using the api:v1/call-template resource and the results will contain references to the appropriate Create action in the hypermedia links for each search result.

```
{
  "Ref": 96,
  "Name": "Default",
  "_context": "api:v1/call-template/$metadata",
  "_self": "api:v1/call-template/96",
  "_actions": {
        "Create": [
          {
               "_self": "api:v1/call/$Create",
               "href": "api:v1/call",
               "description": "The resource that should be used to create
a call of this type. The Template property in the new call should be set to
the Ref of this result."
          }
        ]
  }
}
```

In this example, we would choose to create a call using

```
api:v1/call
```

When choosing a Call type using the API, either of these approaches can be used for any type of user, or you can define your own mechanism.
For example, your user could choose from a list of Call types directly (instead of templates or IPK Tiers)

Creating a Call Object

Developers may choose to take a trial and error approach to call creation.
For example, submitting an empty Object will likely result in a validation error like the following:

```
{
  "Message": "The request is invalid",
  "Type": "FieldValidationException",
  "Errors": {
    "IpkStatus": [
      "Required: IpkStatus must be set when Template is null"
    ],
    "IpkStream": [
      "Required: IpkStream must be set when Template is null"
    ],
    "Partition": [
      "Required"
    ]
  }
}
```
The exact error is entirely dependent upon the configuration of the system.

Call Create Metadata

The metadata for the Call entity describes all of the actions that are available and all of the properties of that entity.
Each action may also contain a description of the allowed inputs for that action.

The Call Create action is no exception and does define applicable inputs and the associated validation rules.

Some fields may be required or readonly depending upon the system configuration.
This configuration is described in the action metadata and is validated by the server.
For example

```
{
    "_context": "api:v1/call/$metadata",
    "_self": "api:v1/call/$Create",
    "href": "api:v1/call",
    "methods": [
        "POST"
    ],
    "inputs": [
        {
            "property": "ConfigurationItem"
        },
        {
            "property": "User"
        },
        {
            "property": "Priority",
            "readonly": true
        },
        {
            "property": "Description",
            "relations": [
                {
                    "behaviors": [
                        {
                            "type": 11,
                            "scope": 1
                        }
                    ],
                    "source": "Description",
                    "target": "DescriptionHtml"
                }
            ]
        },
        {
            "property": "DescriptionHtml",
            "relations": [
                {
                    "behaviors": [
                        {
                            "type": 14,
                            "scope": 1
                        }
                    ],
                    "source": "DescriptionHtml",
                    "target": "Description"
                }
            ]
        },
        {
            "property": "Type"
        },
        {
            "property": "Organization"
        },
        {
            "property": "Template",
            "relations": [
                {
                    "behaviors": [
                        {
                            "type": 16,
                            "scope": 2
                        },
                        {
                            "type": 17,
                            "scope": 2
                        }
                    ],
                    "source": "Template",
                    "target": "IpkStatus"
                },
                {
                    "behaviors": [
                        {
                            "type": 16,
                            "scope": 2
                        },
                        {
                            "type": 17,
                            "scope": 2
                        }
                    ],
                    "source": "Template",
                    "target": "IpkStream"
                },
                {
                    "behaviors": [
                        {
                            "type": 16,
                            "scope": 2
                        }
                    ],
                    "source": "Template",
                    "target": "Name"
                }
            ]
        },
        {
            "property": "IpkStatus",
            "relations": [
                {
                    "behaviors": [
                        {
                            "type": 14,
                            "scope": 2
                        }
                    ],
                    "source": "IpkStatus",
                    "target": "IpkStream"
                }
            ]
        },
        {
            "property": "IpkStream",
            "relations": [
                {
                    "behaviors": [
                        {
                            "type": 14,
                            "scope": 2
                        }
                    ],
                    "source": "IpkStream",
                    "target": "IpkStatus"
                }
            ]
        },
        {
            "property": "Partition",
            "required": true,
            "relations": [
                {
                    "behaviors": [
                        {
                            "type": 15
                        }
                    ],
                    "source": "Partition",
                    "target": "ConfigurationItem"
                },
                {
                    "behaviors": [
                        {
                            "type": 15
                        }
                    ],
                    "source": "Partition",
                    "target": "User"
                },
                {
                    "behaviors": [
                        {
                            "type": 15
                        }
                    ],
                    "source": "Partition",
                    "target": "Priority"
                },

            ]
        }
    ],
    "description": "Create a new record of this type",
    "status": "Alpha"
}
```

The metadata for each action may include "_context", "_self", "href", "description" and "status"

It may also include "methods", which defines the HTTP Methods/Verbs that can be used for the action.

The action metadata may also define and array of "inputs"

Each input may include the following properties

"property":

The path of the input value. This may be a reference to the name of a property for the current entity "_context" (see [[API Metadata]] or may be a dot separated path

This defines the structure of the object that must be sent in the request body when the action is invoked.

e.g $action.CallActionType describes the following JSON structure

{ "$action": { "CallActionType": value } }

When the "property" path does not refer to an entity property name, all other relevant property values will also be defined. ie "type", "displayType", etc.

"required": A boolean value indicating that the input is required. Omitted unless it is true

"readonly": A boolean value indicating that the input is readonly. Omitted unless it is true.

"relations":

An array of relationships between input fields

Each relation includes "source" and "target" which reference "property" paths within the "inputs" array.

Each relation will also include an array of "behaviours". These define validation level rules.

The behaviors are used in server side validation and transformation rules and are provided so that the client can understand

input requirements without needing to rely upon server side validation.

Behaviours may include the following properties

"phase":

This indicates when the behavior will be applied.

1 indicates that the behavior will be applied after the record has been updated (but before it is committed to the database)

0 indicates that the behavior will be applied before, and therefore will apply to the input only.

"type":

This indicates the type of relation and may be one of the following

11 (CalculateHtmlTextIfNull): Indicates that the HTML text value of the target input should be set. When processed on the server.

12 (RequiredIfTrue): Indicates that the target input is required when the source input is true

13 (ZeroIfNotNull): Indicates that the target input must be set to zero when the source input has a value

14 (RequiredIfNotNull): Indicates that the target input is required when the source input has a value

15 (Dependency): Indicates that the target input is dependent upon the value of the source input. This typically applies to partitioned inputs.

16 (ReadonlyIfNotNull): Indicates that the target input is readonly when the source input has a value

17 (RequiredIfNull): Indicates that the target input is required when the source input does not have a value

18 (ReadonlyIfNull): Indicates that the target input is readonly when the source input does not have a value

"scope":

This indicates where the behavior should be interpreted.

0 indicates that the behavior should be implemented by the client and will be ignored by the server

1 indicates that the behavior can be implemented by the client and will be implemented by the server

2 indicates that the behavior should be implemented by the client and will be implemented by the server

Using the Call Create action metadata, we can easily determine which fields are required, and analyse the more complex relationships between fields.

For the example of creating a Call from a Template, the following ajax call could succeed

```
var create = $.ajax({
        url: metadata.href.replace(/^api:/, "alemba.api/api"),
        data: {
            Template: 96
        },
        method: metadata.methods[0],
        contentType: 'application/json',
        headers: {
            'Authorization': "Bearer " + access_token
        }
});
```

Call Create Response

When successful, the Call Create action will respond with HTTP Status Code 201.
The response body will include the actual changes to the Call record and hypermedia links to the currently applicable actions.

For example

```
{
    "ActualLogDate": "2017-01-23T12:12:41.0000000Z",
    "Template": 96,
    "Ref": 10056,
    "_context": "api:v1/call/$metadata",
    "_self": "api:v1/call/10056",
    "_actions": {
        "Submit": [
            {
                "_self": "api:v1/call/$Submit",
                "href": "api:v1/call/10056/submit"
            }
        ],
        "Update": [
            {
                "_self": "api:v1/call/$Update",
                "href": "api:v1/call/10056"
            }
        ]
    }
}
```

The "_actions" property in the response indicates the actions which are applicable to the created Call *in its current state*. A Call will only be visible to the creator until it has been submitted, therefore if the "Submit" action is listed in the Create response, this action should be invoked before relinquishing responsibility for the Call.

Replicating behaviour of Self Service

If a call is logged in Self Service, fixed rules are applied after saving, to populate certain fields. This has not been replicated in the API, as we will be making such rules configurable in the future. If you wish to replicate the current logic, you need to programmatically do so, following the rules below:

- If LocationId is not set, LocationId is set to the current User's Location Ref (if they have one)
- If OrganizationId is not set, OrganizationId is set to the current User's Organzation Ref (if they have one)
- If the LocationId is still not set, this is set to the first valid Location of: the selected User or Service or CI or Organization of the call.
- If the OrganzationId is still not set, this is set to the first valid Organzation from: the selected User or Service or CI of the call.

Impact of data on post-creation rules

Almost any property of a call may be used by IPK Workflow Rules, and each may have a different impact, according how the rules are set up in your system. The API user should be fully conversant with how these rules are configured, so as to achieve the desired assignment, notification and workflow creation.

Call creation checklist

To summarize, the actions to perform are as follows,

- For Users,
  - Select a Call Template (restricted by partition/security if active)
- For Analysts,
  - Select a Screen Set (and IPK Status, IPK Stream, Problem Type if so configured, restricted by partition/security if active)
- Then for both,
  - Apply variable data
  - Create Call

Creating Requests

The templates retrieved are subject to several security and configuration settings, including:

- partitioning Request Screen Sets and Workflow Templates
- Workflow Management Settings
- login's Workflow Management Role's Template Security settings

Calls and Requests – Determining Initial Assignment

In the WCF API, it is possible to specify the Officer and/or Group to which Calls and Requests are forwarded on creation. These are specified as parameters rather than properties to update. They are ignored if IPK or Workflow rules are in place to set this.

In the RESTful API, it is possible to set these values on the Defer action, rather than the Create, so API users should use both actions, one after the other, to achieve the same thing.

Calls and Requests – Recording Actions

In the WCF API, the transactions to create and update calls include the ability to record “Action and Solution” information, which is largely recorded an associated history record. In the RESTful API, this can be achieved using the Defer action. This may include some of the following parameters:

- $action.Description.Description
- $action.Description.DescriptionHtml
- $action.Description.Title
- $action.ActionType

Creating a Service Order

The items that need to be created for a valid service order are:

- Service Order
- Service Order Items
- A Call or Request per Service Order Item (if you want those items immediately submitted)

Required inputs are: one or more service actions and/or service bundles, and a quantity for each.

To create a new Service Order, a new service order record must first be created (api:v1/service-order/$create)

For each service action in the order:

A Service Order Item linked to the Service Order must be created (api:v1/service-order-item/$create) and a Call or Request must be created which is linked to the Service Order Item.

The type of Call or Request to create is indicated in the metadata links for each service action

Finally, the Service Order must be Submitted (using the Submit action), which will automatically calculate the Order Total and Submit the linked Tickets.

The new dynamic service bundle type is not supported in this version of the API.

The creation of a request will automatically create the necessary tasks and initiate the workflow.

Note that the entity called Order is used for transactions that are part of Asset Management, and is not used for Service Orders.

Closing Calls

The process of closing a call in Core is subject to a number of configurable rules, settings and privileges. Some of these are handled by the rules engine and will be automatically invoked after calling the Close method, others need to be considered by the API user in determining what to pass to the Close method, and whether to call additional methods.

List of Nova related database tables

Nova is driven by metadata, which is stored in the vFire database in new tables. These include:

> _EFMigrationsHistory, AppSetting, Clients, EntityPage, Form, FormDisplayType, FormType, InfraPermission, InfraRole, Label,LabelKey, Language, AlembaSystem, AlembaSystemPage, Page, PageBackLink, Permission, PermissionType, Query, QueryMetadata, QueryParameter, RefreshTokens, Role, RolePermission, ServiceEvent, UserPermission, Widget, WidgetConfig, WidgetConfigLabel, WidgetConfigNavigation, WidgetConfigNavigationCondition, WidgetConfigQuery, WidgetConfigSetting, WidgetContract, WidgetContractParameter, WidgetInstance, WidgetNavigationContractBinding, WidgetQueryContractBinding, WidgetSetting

These tables do not need to be accessed directly to use the API and so are not exposed as entities in the API Explorer.