# Communication Protocol Description

Chris Morin <chris.morin2@gmail.com>

## Table of Contents



# 1. Introduction

Communication between users and organization applications will consist of them passing YAML objects back and forth (henceforth called dG objects). In the context of a single transaction, the user wishing to acces an account hosted by the organization application will be called the client and the organization application will be called the server. The specific medium used to pass these objects isn't of great importance, so we'll assume it's occurring over https.

The client and server use their keys to open a session and once handshaking is complete, the client sends a Request dG object. This Request object contains the information relating to what the client wants from the server. The server processes the request by ensuring the client has the relevant permissions and performing the desired action if so. The server then replies with a Response dG object which states whether the action was a success or not and contains data related to the nature of the request (e.g. if the request was a request to read a certain value, the response would contain said value).

So far we have seen two dG object types: requests and responses. A third type is used by servers to publicize the structure of their user accounts along with details about the nature of the data in them. These objects are called templates.

All dG objects conform to the following structure. Any element preceded by a tilde is optional.

```
---
[template | request | response]:
        <object content>

author ID: <ID of user who authored this document>
author key: <author's public key>
signature: <signature of author>
...
```

This document will detail how these 3 dG objects are structured and why they are structures as such.

# 2. Template Object

Template objects are publicly available from organization applications and allow users to see the structure and content of an organization's accounts without even needing to have their own account. Template objects conform the process of users exploring what kind of data they have access to in their account and how they can interact with in.

## 2.1. Structure

Template objects are dG objects so they adhere to the structure detailed above. The object content section (the only part that varies between object types) of template types is detailed below.

```
organization name: <org name>
short_description: <description of account purpose>
~long_description: <longer description>
data_map:
        <data map>

actions:
        <list of actions>
```

## 2.2. Data Map

A data map looks like this.

```
<data 1 name>:
        <data_item 1>

~<more data_items>
```

A data_item conforms to the following structure.

```
type: "inode | list | regular | image | binary blob"
~access_policy: <access policy>
is_optional: "true | false"
~description: <description of data>
~children: <data map of children items or single item>
```

type
: Type of the data_item. These are the first types but others could be added. inode simply means that the type isn't a value but contains children which themselves are data_items. data_items of type regular are items which conform to YAML's built-in types (i.e. numbers, strings, associative array, list).

access_policy
: The access policy describes who has access to data and how the owner can grant access. The syntax of the access policy is detailed in a later section.

is_optional
: Some data_items aren't necessarily present in all accounts. This field specifies whether the data_item might not be in an account.

description
: Description of the data_item.

children
: Only inodes have a children field. In the case of inodes, children field contains a data map with child data_items inside. In the case of lists, the children field contains a single data item. A list can have several data items of the same form and the child data item specifies this form. Some organization applications could have large and complex account structures. Putting all of this in a single file and transmitting it whenever anyone wants to know about how the account is structured would impractical. Several template files can be used to describes different parts of the data structure. In the case where a inode type doesn't have a children field, the children can be accessed in the remote file. How a client fetches other template will be described later.

## Access policy

By default, the account owner will have read access to all fields while the organization which hosts the account will have read and write access. All other users have no access. The access policy section details how permissions on the specific piece of data differ from this. All permissions are inherited recursively by any children data items.

The access policy will consist of a list of user ID/permission pairs. A user ID permission pair looks like this.

```
<user ID number> : <permissions>
```

The permission part can be *r*, *rw* or the probably seldom used *w* (write access without read access). The user ID number will have two aliases: "owner" and "host". These will make the common case of changing the account owner's or the hosting organization's permissions more legible. The access policy also shows what kind of permissions a user can grant. This is specified after the user's permissions as a *g* followed by the specific permissions they can grant. A possible access policy field could look like this:

```
access_policy:
        - owner: rw gr
        - 7d4f4e363e: r
        - 4e277a5539: rw
```

**Note**

I might add another permission called *a* which causes the host to send an alert to the user ID when the data field is changed.

# 2.3. Actions

Actions allow users to interact with accounts in ways not easily possible with simple reads and writes. An example could be scheduling a doctor's appointment; it would feel unnatural to schedule an appointment by writing to certain data locations.

A list of actions looks like this

- <action name 1> : <action 1> ~- <more actions>

An action is structured as so:

```
parameters: <parameters>
caller_list: <list of users who can call the action>
~description: <description of action>
```

The parameter field is a map of key value pairs which allow the action caller to pass some data. It’s structured as below.

```
parameters:
        <param 1>: <description of param 1>
        ~<more parameters>
```

Example actions for the Quebec automobile society:

```
actions:
        schedule_test:
                parameters:
                        time: time of the test
                        location:  >
                                test can take place in Dorval (DORVAL) or in
                                Montreal (MONTREAL).
                        type: is it a WRITTEN test or a PRACTICAL test
                        payment: authorization of transfer of $100 to the SAAQ

caller_list: owner
description: schedule a test to obtain driver licence.

add_car:
        parameters:
                used_or_new: is the car USED or is it NEW
                purchase_form: form obtained when vehicle was purchased
                payment: authorization of transfer of $200 to the SAAQ

remove_car:
        parameters:
                car_ID: ID of the car on record you'd like to remove
                selling_form: (optional) form from car sale
```

# 2.4. Example

To put it all together, let’s look at a simple example template. We’ll build off the actions we used above.

```
organization name: SAAQ
short_description:  >
        The Société de l'assurance automobile du Québec or SAAQ
        (English: Quebec Automobile Insurance Corporation), is a
        Quebec crown corporation responsible for licensing drivers
        and vehicles in the province and providing public auto
        insurance

data_map:
        vehicles:
                type: list
                is_optional: false
                description: list of cars user own or has owned
                children:
```

```
                        car_ID: car ID
                        make: make of car
                        model: car model
                        year: year of car model
                        color: color of car
                        car_purchase_date: date car was purchased by account owner
                        date_of_disposal: data account owner lost possession of car

driving_points:
        type: regular
        is_optional: false
        description:  >
                points account owner has and the total points
                available to him in the form of POINTS_LEFT / TOTAL
                POINTS
        access_policy:
                owner: r gr
                        # the owner can grant access to they
                        # points as many insurance companies use this
                        # data for premium calculation.

actions:
        schedule_test:
                parameters:
                        time: time of the test
                        location:  >
                                test can take place in Dorval (DORVAL) or in
                                Montreal (MONTREAL).
                        type: is it a WRITTEN test or a PRACTICAL test
                        payment: authorization of transfer of $100 to the SAAQ

caller_list: owner
description: schedule a test to obtain driver licence.

add_car:
        parameters:
                used_or_new: is the car USED or is it NEW
                purchase_form: form obtained when vehicle was purchased
                payment: authorization of transfer of $200 to the SAAQ

remove_car:
        parameters:
                car_ID: ID of the car on record you'd like to remove
                selling_form: (optional) form from car sale
```

# 3. Request

When a user wants to interact with an organization, they create a request and send it to the organization’s server. The request can be to read some data, write some data or call an action.

## 3.1. Structure

A request object consists of a list of requests and is structured as follows.

```
request:
        - <request item 1>
        ~- <possibly more request items>
```

There are three request item types. Read, write and action call type. A read or write request items is structured like this:

```
req_type: "read | write"
data_path: <path to data>
~new_value: <new data to be written in case of write>
```

The data_path takes on the value of the path to the data to read/write. As seen in the template section, account data in organized in a tree. A path is described as it is in most programming languages with a series of names separated by periods. lists are indexed with brackets (C-style).

An action call request looks like this:

```
req_type: "action call"
action_name: <name of action>
~paramaters: <map of parameter name/values>
```

## 3.2. Example

To make things more concrete, let’s say an individual has just bought a new car and also wants to check up on how many points they have left on their licence. They would create a request object that looks like this.

```
---
request:
        - req_type: read
          data_path: driving_points
```

- req_type: action call action_name: add_car parameters: used_or_new: NEW purchase_form: date_bought: … dealer: … … payment: payment_method: credit …

  ```
  author ID: 5e7a255c43
  author key: 23252d22736b7a257538355a7c31233f273e763a37685027433852455d
  signature: 6e3521214a5e3f496a563d642b21595344516d67716a6e463a70713764
  ...
  ```

# 4. Response

When a client makes a request, the server will reply with a response. The purpose of this reports on the success of the request and provides any relevant data (such as data requested in a read).

## 4.1. Structure

The response object contains a response item for each request item. Naturally, they will have similar structures.

```
response:
        - <response item 1>
        ~- <possibly more response items>
```

A response item:

```
resp_type: <same as corresponding request>
was_succesful: "true | false"
```

```
~failure_cause: ""
~value_read: <value of data item in case of read>
~action_call_resp: <action call might need to return specific data>
```

# 4.2. Example

Let’s look at a possible response to the request example seen above.

```
---
response:
        - resp_type: read
          was_succesful: true
          value_read: 2 / 10
```

- resp_type: action call was_successful: false failure_cause: dealer could not confirm car purchase

```
author ID: 5e7a255c43
author key: 23252d22736b7a257538355a7c31233f273e763a37685027433852455d
signature: 6e3521214a5e3f496a563d642b21595344516d67716a6e463a70713764
...
```