1. API CALLS

1.1 Registering API calls

API calls are defined in json files that looks like this:

```
{
    "description": "Calculate the square root of a number",
    "id": "/maths/sqrt",
    "name": "SquareRoot",
    "sqlfunc": "maths_sqrt",
    "type": "object",
    "properties": {
      "value": { "type": "number", "description": "The input value for the square root function" }
    },
    "result": {
        "type": "object",
        "properties": {
            "result": {"type": "number", "description": "The resulting number"},
            "status": {"enum": ["ERROR", "OK"]}
        }
    }
}
```

Usually, each API call lives in it's own file. However, it is possible to have an array with multiple API call definitions inside a JSON file.

These files live in multiple directories. The list of directories it is found in, is defined in the configuration option "***api_directories***". This option is usually defined in the configuration files read by Grape upon startup. Each grape module will usually have it's own directory to add to the list. After Grape Express App Instances have been created, each of the directories found in this option is traversed, and all JSON files read and parsed.

1.2 Format of JSON Schema API call

The JSON format used to define APIs are loosely based on JSON Schema standard (http://json-schema.org). The fields in a schema object:

```
{
    "description": "API call description",
    "id": "URL of the call",
    "name": "CamelCase name for this call",
    "sqlfunc": "SQL function",
    "sqlfunctype": "Specify jsonb here for JSONB functions. Defaults to JSON",
    "jsfile": "JavaScript file to execute"
    "type": "Body type - use 'object' for POST methods and 'query' for GET",
    "method": "GET or POST - defaults to POST",
    "roles": "An array of role names that is allowed to execute this request",
    "properties": { Set of properties in this body. Used for POST requests },
    "validation_string": "Validation string. Use this to validate GET/POST input",
    "no_validation": true or false. If this is true, validation will be disabled for this call (default
    "return": {
            "type": "Return type (default to object)"
```

```
            "properties": { Set of properties in this body }
            "example": { Example of a resulting JSON }
        }
}
```

The properties field, which is used with the "object" type, is an associative array with the field name as key, and the field properties as the object. For example:

```
"properties": {
    "number_field": {"type": "number", "description": "Number field 1"},
    "string_field": {"type": "string", "description": "A strings field"},
        "array_field": {"type": "array", "items": {
        "type": "string"
        },
        "description": "List of strings"
    }
}
```

1.3 API Handlers

Grape relays the API request to a handler. This handler can either be a function in the database, or a JavaScript file (TODO: write handlers in any language).

1.3.1 Database Handlers

Database API calls are the most commonly used API calls. The handlers are written as functions/stored procedures in the database. The functions accepts and returns a JSON(B) object. The object is created from the request's body (in case of POST), or the request's query parameters (in the case of GETs). The name of the function is specified in the "***sqlfunc***" field. If the function accepts JSONB (instead of JSON), the value of the field "***sqlfunctype***" should be set to "jsonb".

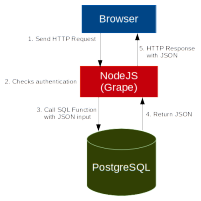

Fig. 1: Anatomy of a DB API

The logic of the function needs to be implemented in PostgreSQL. For example, a function calculating the square root of a number:

```
    /**
     * @api_usage MathsSqrt
     * @api_url POST /maths/sqrt
     */
    CREATE OR REPLACE FUNCTION maths_sqrt (JSON) RETURNS JSON AS $$
    DECLARE
        _value NUMERIC;
        _result NUMERIC;
    BEGIN

        _value := ($1->>'value')::NUMERIC; -- Extract values from JSON
```

```
        _result := sqrt(_value);           -- Calculation

        RETURN grape.api_success('result', _result); -- Build and return JSON object
    END; $$ LANGUAGE plpgsql;
```

1.3.2 JavaScript Handlers

The API handler is implemented as a JavaScript function, and handled completely in the Node.JS environment. The file should be specified using the "***js_file***" field in the JSON schema.

The JavaScript file should return a function, for example:

```
module.exports = function() {
    return function(req, res) {
        // API CODE GOES HERE
        res.end();
    }
}
```

An example of a schema definition for the sqrt function:

1.3.2.1 Validation

If the *validate* field is set in the API definition, validation against the incoming data will happen against this string. The validation string syntax is defined by the following Backus–Naur form:

```
obj-def ::= "(" <param-def-list> ")"
param-def-list ::= <param-def> | <param-def-list> "," <param-def>
param-def ::= <name> ":" <type-info> <modifier-list>
type-info ::= <array-def> | <data-type> | <obj-def>
array-def ::= "[" <data-type> "]" | "[" <obj-def> "]"
data-type ::= "s" | "i" | "f" | "b" | "d" | "t" | "a"
modifier-list ::= <modifier> <modifier-list>
modifier ::= "*" | "E" | "0" | ""
```

The data types that can be used:

- s text
- i integer
- f float
- b boolean
- d date
- t date

Modifiers: * optional is true (default = false) E Empty becomes null 0 Nullable Examples: (batch_labreport_id: i, product_id: i, labreport)

> **Note!** More examples are available in the grape repository, in the **examples/** subdirectory

1.3.2.2 Generating documentation

In the Viticulture repository, a script called *scripts/build_schema_api_docs.js* can be used to generate VXML documentation from the API JSON files.

```
Usage: build_schema_api_docs [options] directory|file [directory|file directory|file ...]

Options:

  -h, --help                  output usage information
  -o, --output [output_file]  Output file (vxml)
```

1.4 Access control on API calls

When an HTTP request is received, Grape attempts to connect a valid session to the request. This is done by looking for an *X-SessionID* header, and additionally looking for a session ID cookie. If a valid session could be identified, the user's access roles are loaded. If no session exists, the role "guest" is used.

Each user has a list of roles assigned to him/her, and each API call has a list of access roles that is allowed to call it. There has to be an overlap between these two set of access roles, otherwise the request is denied. See Authentication for more information.

The list of allowed access roles for each API call is defined in the "*roles*" field in the JSON schema. All users belongs to one or more roles, and always to the role named **all**. API calls are registered in the database (table access_path) by **path**, **method** and the **role** allowed. The **path** is a regular expression, matching the incoming URL of the request.

> **Note!** The SQL function `grape.add_access_path (_path TEXT, _roles TEXT[], _methods TEXT[])` can be used to add a new access path. For initial data, this is kept in the project's db/data/access_paths.sql

1.5 Consuming an API call

The API (built-in API calls and the application's API calls) is available over HTTP/HTTPS on the same port as the frontend. Client-side software can connect to this URL, and can be written in any technology/environment. See access control for details regarding the HTTP header *X-SessionID*.

In order to use an API call, you will need to know the following:

1. The URL of the call. This will look like a typical path, for example "/login"
2. The call method. This will usually be **GET** or **POST**
3. The input parameters. If the call's method is POST, this will be a JSON object containing fields. If it is GET, the input parameters are found in the URL

The API is consumed by making a HTTP request to the API's URL with the appropriate input parameters or body. Usually, session information will be sent in the HTTP headers in the *X-SessionID* header.

1.5.1 Using HTTP Authorization

By default, grape applications support HTTP Authentication (Basic). When using this authentication method, it is not necessary to make a separate call to */grape/login*. The username and password can be included in the HTTP headers. This should only be used over TLS, as the credentials can be reverse-encoded. The "Basic" HTTP authentication scheme is defined in RFC 7617, which transmits credentials as user ID/password pairs, encoded using base64.