1. API CALLS

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

```
{
    "description": "Delete a record",
    "id": "/grape/delete_record",
    "name": "GrapeDeleteRecord",
    "sqlfunc": "grape.delete_record",
    "type": "object",
    "properties": {
        "tablename": {"type": "string", "description": "Table name to delete the record from"},
        "schema": {"type": "string", "description": "Schema name of table"},
        "filter": {"type": "json", "description": "Key-value pair"}
    },
    "example": {"schema": "public"},
    "return": {
        "type": "object",
        "properties": {
            "return": {"type": "json", "description": "The returning values"},
            "status": {"enum": ["ERROR", "OK"], "description": "Status indicator"},
            "message": {"type": "string", "description": "Error message"},
            "code": {"type": "number", "description": "Error code"},
            "error": {"type": "json", "description": "Error details"}
        }
    }
}
```

Usually, each API call lives in it's own 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.

The purpose of this JSON structure

Grape classifies between 3 different types of API calls:

1. Filesystem request: the browser requests a file from the filesystem (for example an HTML, CSS, JS or image file)
2. Database API request: the API handler is implemented as a database function, accepting and returning a JSON object
3. JavaScript API request: the API handler is implemented as a JavaScript function, and handled completely in the Node.JS environment

1.1 FS Request

The first and most simple is a request for a file on the filesystem. A request that does not accept JSON, and does not start with /download, will fall under this category. This includes the initial call for index.html. Files are searched for in the directories listed in the Grape config setting *public_directories*, in the order that it is specified in the setting. The first file found matching the path (relative to any public directory) and filename is served.

1.2 DB API requests

Database API calls are the most commonly used API calls. The logic for the function is typically implemented as a function in PostgreSQL. The function being called in the database accepts a JSON parameter, and returns a JSON object with the result.

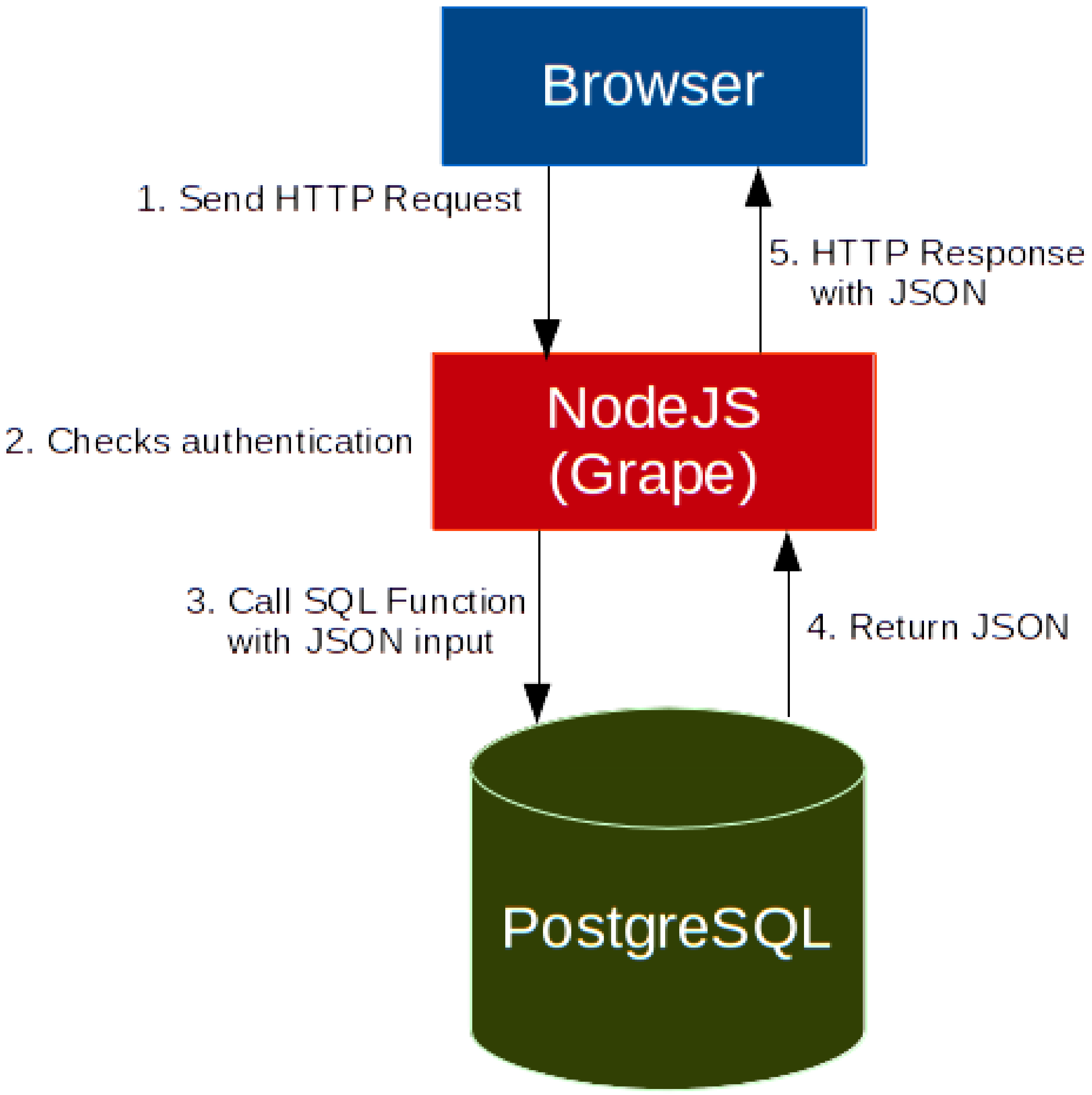

Fig. 1: Anatomy of a DB API

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;
```

The database function needs to be registered against an Express route in GrapeApp. There are two ways to do this: using a schema file, or registering it manually.

1.2.1 Registering the API call using schemas

Note! This method of creating DB API calls was introduced in Grape 1.0.4

APIs can be defined using a format based on the JSON Schema standard (http://json-schema.org), with added fields. On start-up, all the directories in the project's *api_directory* config setting is traversed for JSON files. JSON files are then parsed, and must contain a single object, or an array of objects describing the API calls.

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",
    "type": "Body type - use 'object' for POST methods and 'query' for GET",
    "method": "GET or POST - defaults to POST",
    "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"
    }
}
```

An example of a schema definition for the sqrt function:

```
{
    "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"]}
        }
    }
}
```

1.2.1.1 Validation

If the API call's method is POST, the received body will be validated against the JSON schema (the fields in the *properties*). When the properties field cannot be used (for example in GET calls), an *validate* field can be specified. This field contains a string, in the following format:

```

Validate string syntax:
      begin_expr ::= "(" <begin_expr> "," <param-def> ")"
      param-def ::= <param-name> ":" <data-type> <modifier-list>
      modifier-list ::= <opt-modifier> <modifier-list>
      opt-modifier ::= "*" | "E" | "0" | ""
      data-type ::= "s" | "i" | "f" | "b" | "d" | "t" | "a"

Data types:
      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

1.2.1.2 Generating documentation

In the Viticulture repository, a script called *scripts/build_schema_api_docs.js* can be used to generate VXML documentation for schema-based API calls.

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

  Options:

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

1.2.2 JavaScript API handlers

API calls that are handled by JavaScript functions, can be found in the project's *api directory* (set in the config, usually *api/*). On startup, these directories are traversed for **.js* files and included in the application. API calls are registered using *app.get* and *app.post*. For example, the contents of a JS file in the *api/* directory:

```
exports = module.exports = function(app) {
        app.get("/maths/sqrt/:value", function (req, res) { /* Logic here */ });
}
```

Example of an API call, calling a database function:

```
exports = module.exports = function(app) {
    // register the route
    app.get("/maths/sqrt/:value", api_maths_sqrt);
}
function api_maths_sqrt (req, res)
{
    // call the stored procedure for this API call
    res.locals.db.json_call("maths_sqrt", // the name of the PL/pgSQL function
        {value: req.params.value},    // Build the JSON object as input for this function
        null,                         // Optional callback (not used here)
        {response: res}               // Send the response to res
    );
}
```

API calls should be documented using a standard based on the doxygen style. These comments can be extracted from the code using the *build_jsapi_documentation.js* script in Viticulture to create human-readable documentation. This is especially important for front-end developers, who can then use this documentation to interact with the API calls. All projects should have API documentation ready for clients to use.

A comment block containing documentation for an API call:

```

/**
```

```
 * @api URL  The URL of the API call
 * @desc DESCRIPTION Description of the API call
 * @method GET/POST HTTP Method
 * @sqlfunc SQL_FUNCTION_NAME The SQL function used by this API call
 * @param NAME TYPE DESCRIPTION OPTIONAL DEFAULT When documenting GET requests, specify the
                                                  input parameters using @param
 * @returnsample JSON Example JSON of a successful return call
 * @return DESCRIPTION Description of the return values
 *
 */
```

1.2.2.1 Generating documentation

In the Viticulture repository, a script called *scripts/build_jsapi_documentation.js* can be used to generate VXML documentation for JavaScript API calls.

```
Usage: build_jsapi_documentation [options] directory [directory directory ...]

Options:

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

1.2.3 Non-JSON returning API calls

An API call that is handled by a JavaScript call and needs to return something other than a JSON object, starts with */download*.

1.3 Access control

Grape manages authentication, sessions, users, passwords and provide access control. Sessions are tracked using a session ID. Session IDs are obtained by making a call to *POST /grape/login*, providing a valid username and password. On success, this call will return with the session ID, the user's ID, username, roles and employee GUID. In subsequent calls, the session ID should be sent to the server using a custom header *X-SessionID* in the HTTP requests.

Users and user-related information is stored in grape.user. Users can belong to one or more access roles, stored in grape.access_path. The link-table for these are grape.user_role.

Access control is applied to all API calls. Before the API call is executed, the session is validated. If it cannot be validated, the default role **guest** is used. 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

1.4 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. The following two headers also need to be set on all API calls:

1. Content-type: application/json
2. Accept: application/json

1.4.1 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.