



PLATINUM **SOFTWARE**

## PlatSoft Grape Manual

Developer's guide to using the PlatSoft Grape Framework

# Table of Contents

|   |         |
|---|---------|
| 1. Grape  | Page 4  |
| 1.1. Architecture                               | Page 5  |
| 1.2. Getting started using grape_init           | Page 5  |
| 1.2.1. Installing packages                      | Page 5  |
| 1.2.1.1. Install Grape-UI components            |         |
| 1.3. Directory layout                           | Page 6  |
| 1.4. API Calls                                  | Page 6  |
| 1.4.1. FS Request                               | Page 6  |
| 1.4.2. DB API requests                          | Page 6  |
| 1.4.2.1. Registering the API call using schemas |         |
| 1.4.2.1.1. Validation                           |         |
| 1.4.2.1.2. Generating documentation             |         |
| 1.4.2.2. JavaScript API handlers                |         |
| 1.4.2.2.1. Generating documentation             |         |
| 1.4.2.3. Non-JSON returning API calls           |         |
| 1.4.3. Access control                           | Page 11 |
| 1.4.4. Consuming an API call                    | Page 11 |
| 1.4.4.1. Using HTTP Authorization               |         |
| 1.5. Database                                   | Page 12 |
| 1.5.1. Defining the database structure          | Page 12 |
| 1.5.1.1. Database model                         |         |
| 1.5.1.2. Database functions                     |         |
| 1.5.1.3. Initial data                           |         |
| 1.5.1.4. Indexes                                |         |
| 1.5.1.5. Naming conventions                     |         |
| 1.5.1.6. Pitfalls to avoid                      |         |
| 1.5.1.7. Schemas                                |         |
| 1.5.2. Using NOTIFY/LISTEN                      | Page 13 |
| 1.6. grape-db-setup                             | Page 13 |
| 1.6.1. File order                               | Page 14 |
| 1.6.2. Manifest files                           | Page 14 |
| 1.7. Log directory                              | Page 14 |
| 1.7.1. Log files                                | Page 14 |
| 1.7.2. grape.pid                                | Page 15 |
| 1.7.3. grape.fifo                               | Page 15 |
| 1.8. Sending Emails                             | Page 15 |
| 1.8.1. Setup                                    | Page 15 |
| 1.8.2. Send an email from PostgreSQL            | Page 15 |
| 1.8.3. Send an email through an API call        | Page 15 |
| 1.8.4. Custom headers                           | Page 15 |
| 1.8.5. Email templates                          | Page 15 |
| 1.8.6. Example                                  | Page 17 |
| 1.9. Generating PDF Files                       | Page 17 |
| 1.9.1. XML-producing SQL function               | Page 18 |
| 1.9.2. XSL file                                 | Page 18 |
| 1.9.3. API handler                              | Page 18 |
| 1.10. User IP Filter                            | Page 18 |
| 1.11. Grape Notifications                       | Page 18 |

|   |         |
|---|---------|
| 1.11.1. Defining notification functions | Page 19 |
| 1.11.2. Notifications in GrapeUI        | Page 19 |
| 1.12. Grape settings                    | Page 19 |
| 1.12.1. Known Grape settings            | Page 19 |
| 1.13. Grape config file                 | Page 20 |
| 1.14. Standardized Error Codes          | Page 22 |
| 1.15. grape-manage                      | Page 22 |
| 1.15.1. Adding new commands             | Page 22 |
| 1.15.2. Bash auto completion            | Page 23 |
| 1.16. Built-in API calls                | Page 23 |
| 1.16.1. GrapeDeleteRecord               | Page 23 |
| 1.16.2. GrapeInsertRecord               | Page 24 |
| 1.16.3. GrapeListQuery                  | Page 24 |
| 1.16.4. GrapeSendMail                   | Page 25 |
| 1.16.5. GrapeUpdateRecord               | Page 25 |
| 1.16.6. GrapeUserSave                   | Page 26 |
| 1.16.7. download_public_js_files        | Page 27 |
| 1.17. Grape SQL Functions               | Page 27 |
| 1.17.1. API result functions            | Page 27 |
| 1.17.2. Data importing functions        | Page 28 |
| 1.17.3. JSON helpers                    | Page 29 |
| 1.17.4. List query                      | Page 30 |
| 1.17.5. Table Operations                | Page 31 |
| 1.17.6. Reports                         | Page 31 |
| 1.17.7. User related functions          | Page 32 |
| 1.17.8. Session related functions       | Page 32 |
| 1.17.9. Other utility functions         | Page 34 |
| 1.18. Grape SQL Views                   | Page 34 |
| 1.18.1. User Related Views              | Page 34 |
| 1.18.2. Session Related Views           | Page 34 |
| 1.18.3. Other Utility Views             | Page 34 |

# 1. GRAPE

**Note!** This section deals mostly with the backend functionality of systems. For information on the user interface, see "Grape User Interface"

Grape is a framework and set of processes developed internally by Platinum Software. It is implemented as an Express.js app. A Grape application provides an HTTP/S interface for web clients to connect to.

Grape deals with the following in an application:

- [Access control](#)
- Session management
- Database connections
- Relaying [API calls](#) to handlers (DB functions or Javascript)
- Serving static files

Other functionality provided for developers, are:

- [Sending emails](#) (generated with templates)
- [Data import \(XLS/CSV\) and processing](#)
- [PlatSoft Background Worker](#)
- [Generating PDF files from XML \(using XSL stylesheets and Apache FOP\)](#)
- [Sending system notifications](#)

## 1.1 Architecture

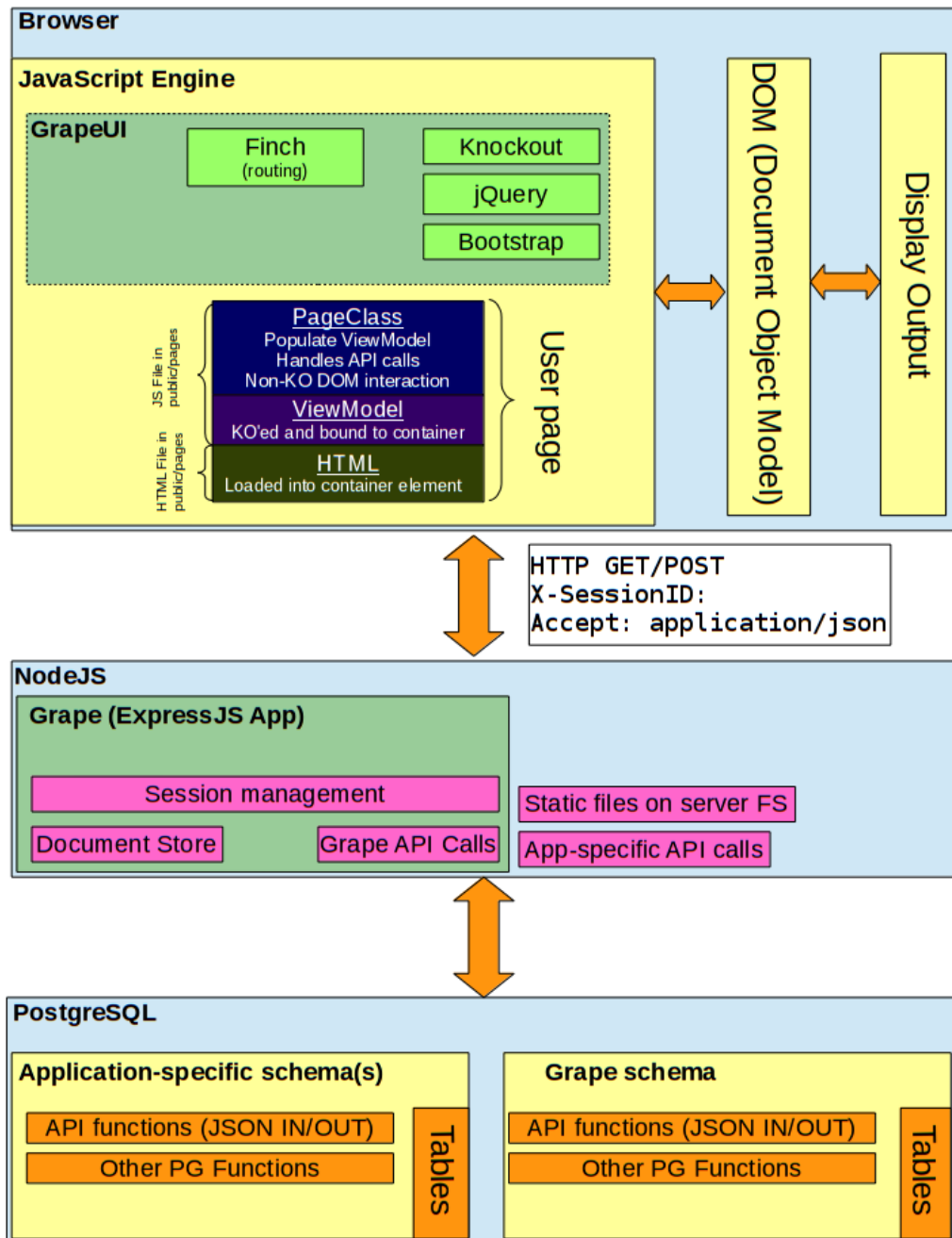


Fig. 1: Grape Architecture

## 1.2 Getting started using grape\_init

**Note!** This feature is only available in Grape 1.0.4 and above

`grape_init` is a script that can be used to bootstrap a new application. After installing, simply run `grape_init` in a terminal and follow the instructions on screen.

### 1.2.1 Installing packages

1. If NVM is not installed, install [Node Version Manager](#)
2. If NodeJS is not install, install the latest [NodeJS](#) LTS version
3. `npm install -g grape-ui`

4. `npm install -g grape-tools`

### 1.2.1.1 Install Grape-UI components

Run the following commands in a terminal from within the project you have just created to have a reference to the grape-ui folder:

1. `grape-ui -l (list components)`
2. `grape-ui -i [component name] e.g. grape-ui -i user_management`

## 1.3 Directory layout

Most Grape applications will have the following directories/files:

- [\*api/\*](#) - API routes to be loaded and registered by Node on startup
- [\*db/\*](#) - Database-related files (mostly SQL)
- [\*public/\*](#) - Public (HTML, Frontend JavaScript and CSS files). See Grape Frontend Development for more information regarding the contents of this directory
- [\*grape-ui/\*](#) - Public (HTML, Frontend JavaScript and CSS files) for GrapeUI components
- [\*scripts/\*](#) - Scripts
- [\*log/\*](#) - Log files
- [\*node\\_modules/\*](#) - This directory is automatically generated by npm when installing modules
- [\*config.js\*](#) - Grape config file (should not be checked into the repo)
- [\*config.js.example\*](#) - Example Grape config file
- [\*index.js\*](#) - Application entry point
- [\*default\\_config.js\*](#) - Grape config file containing product-specific settings
- [\*email\\_templates\*](#) - Email templates

## 1.4 API Calls

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

### 1.4.1 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.

### 1.4.2 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.

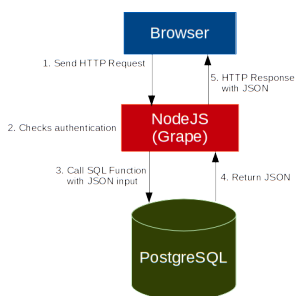


Fig. 2: Anatomy of a DB API

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.

### 1.4.2.1 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",

```

```

"sqlfunc": "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 false),
"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"] }
    }
  }
}

```



### 1.4.2.1.1 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

### 1.4.2.1.2 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
```

### 1.4.2.2 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
 *
 */
```

#### 1.4.2.2.1 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
```

### 1.4.2.3 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**.

### 1.4.3 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`.

Grape includes the following roles and access paths by default:

| ROLE  | DESCRIPTION         | PATHS ALLOWED   |
|-------|---------------------|---|
| guest | No or invalid login | <ul style="list-style-type: none"> <li>/grape/login</li> </ul>  |
| all   | All logged in users | <ul style="list-style-type: none"> <li>/lookup/*</li> <li>/grape/list</li> <li>/grape/api_list</li> </ul> |
| admin | Administrator       | <ul style="list-style-type: none"> <li>* (all paths allowed)</li> </ul>                                   |

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`

### 1.4.4 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

### 1.4.4.1 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.

## 1.5 Database

Grape interacts heavily with a database. Users, sessions and other data is stored in the database. Functions are defined here, and most business logic happen in the database. The following subdirectories can be found in a project's `db/` directory.

- `schema/` - The database schema file (**.dbm file** - created in [pgModeler](#), and exported DDL to SQL files).
- `function/` - Directories and files containing database functions.
- `process/` - Files containing database functions related to [background processes](#).
- `data/` - Files containing initial data for the system.
- `deployments/` - Containing subdirectories for specific deployments. See the section Deployments for more information regarding this.

### 1.5.1 Defining the database structure

#### 1.5.1.1 Database model

pgModeler is a database modelling tool specifically designed for PostgreSQL. The model are saved in a DBM (\*.dbm) file, and exported to a \*.sql file. Usually, both **.dbm** files and the corresponding **.sql** files are added to the repositories. The sql files are always generated by pgModeler and should never be edited manually. The DBM file is saved in `db/schema/PROJECTNAME.dbm`. From within pgModeler, the SQL should be exported to a file `db/schema/PROJECTNAME.sql`. When the database is created, this file will be loaded.

#### 1.5.1.2 Database functions

Database functions (like API call handlers or business logic) usually resides in `db/function/`. Files should be properly named, and functions grouped together in files, in a way that is understandable to someone who is not familiar with the code. File names should be all lowercase, and words split by a underscore. For example, `file_name.sql`.

#### 1.5.1.3 Initial data

Initial data for the system usually resides in `db/data/`. These sql functions includes data such as user/access roles, settings and whitelist tables specific to the system.

#### 1.5.1.4 Indexes

**Note!** Remember that primary key columns automatically have indexes assigned to them.

Always make indexes on the following columns:

- Any column that is referenced by a foreign key.
- Any column that references a foreign key.
- Any column that is used often (in the WHERE clause of queries).

#### 1.5.1.5 Naming conventions

- Keep table names lowercase, with words separated by a underscore. For example, `table_name`.

- If a relation (table) has an id/unique column (which should be the case in most tables), add '\_id' to the table name. For example, table policy has an ID column called **policy\_id**.

**Note!** There are security risks involved in using an auto-incrementing column where the value of such a column is exposed to clients, or untrusted personnel.

- Use a 'v\_' prefix for views. For example, **v\_policy**
- Use a 'mv\_' prefix for materialized views. For example, **mv\_policy\_address**
- Use a '\_idx' suffix for indexes. For example, **policy\_id\_idx**
- Use a '\_fk' suffix for foreign key constraints, with an abbreviation of the tablename as prefix. For example, **po\_policy\_id\_fk**

### 1.5.1.6 Pitfalls to avoid

- Avoid multi-column indexes (and primary keys) unless it is really suitable for the situation.
- Avoid too many indexes on a table.

### 1.5.1.7 Schemas

By default, grape applications will use the following schemas:

- public
- grape
- proc

## 1.5.2 Using NOTIFY/LISTEN

Grape includes a utility function, **new\_notify\_handler(channel\_name, callback)**, which will keep track of callbacks for database NOTIFY events.

## 1.6 grape-db-setup

**grape-db-setup** is a tool used to load SQL files into a database. This is typically done during initial creation of a database, or when applying patches to a database. **grape-db-setup** accepts the following options:

- **-d, --dburi [dburi]** Connection parameters for the target database. If dburi is not specified, the generated SQL will be printed to stdout.
- **-s, --superdburi [superdburi]** Connection parameters that will be used when creating and dropping the database.
- **-c, --create** Create the database before attempting to create objects.
- **-r, --drop** Drop and recreate the database before attempting to create objects.
- **-i, --continue** Continue processing when an error occurs (by default, processing will stop).
- **-e, --schema** The default schema to use when creating objects (defaults to "public"). If "none" is specified, search\_path will not be set.

In addition to the above mentioned options, one or more files/directories should be provided. Each of these additional arguments will be processed:

- If the entry is a directory, the files in this directory will be recursively processed and **.sql** files be included in the output.
- If the entry is a **.sql** file, the file will be included in the output.
- If the entry is a **.manifest** file, the file will be read and lines will be processed accordingly.

## 1.6.1 File order

**grape-db-setup** loads SQL files alphabetically. Sometimes it might be necessary to explicitly change this load order. This can be done by adding a **-- Require:** pre-processing instruction in the SQL file. The file that is required, will be loaded before the file containing the instruction. For example, having the following in an SQL file called **file.sql**, will force the loading of **other\_file.sql** before **file.sql**.

SQL files can contain a:

```
-- Require: other_file.sql
```

## 1.6.2 Manifest files

Manifest files contains a list of files (sql and other manifest files) and directories. If a manifest file is provided to **grape-db-setup**, each line will be processed. Manifest files can contain comments using the **#** character.

```
# Load grape files
../node_modules/ps-grape/db/initial.manifest

db/function/
```

Most projects will include 2 manifest files usually:

- **initial.manifest** - To be used during initial creation of database. This file will create the database schema and tables (as exported from pgModeler).
- **functions.manifest** - To be used when recreating functions. This file will load all PL/pgSQL functions and views.

Additionally, deployments can include manifest files in order to apply changes to target live databases.

## 1.7 Log directory

The log directory is usually called **log** and is located in the project's base directory.

**Note!** None of the files that are created in the log directory should be checked in the git repository, as it will be automatically created.

### 1.7.1 Log files

Log filenames follows the following convention: **<CHANNEL>-<LEVEL>-<YYYYMMDD>.log**, for example **db-error-20170419.log**. Symbolic links will be created in the log directory, pointing to the latest of each of the log files.

The available levels are:

- **debug** - debug messages. This will not be logged if debug is false
- **info** - informational messages
- **warn** - warning messages
- **error** - error messages
- **trace** - trace messages

The available channels are:

- **api** - messages related to API calls
- **app** - messages related to the application
- **session** - session messages (login, logout, grants, etc)
- **db** - database messages (db errors, calls, etc)
- **comms** - comms messages

## 1.7.2 grape.pid

This is a special file containing the PID (process ID) of the running grape process.

## 1.7.3 grape.fifo

This FIFO is used for communication between application processes.

# 1.8 Sending Emails

## 1.8.1 Setup

The following needs to be set up in order to send emails from within SQL functions:

1. **smtp** settings in config

```
smtp: {
  host: 'mail.platsoft.net',
  from: 'Merlot <merlot-live@platsoft.net>',
  secureConnection: true,
  port: 465,
  auth: {
    user: 'username',
    pass: 'password'
  }
}
```

2. **email\_template\_directory** in config containing templates, typically `__dirname + '/email_templates'`

## 1.8.2 Send an email from PostgreSQL

Call `grape.send_email(to TEXT, template TEXT, data JSON, headers JSON)` to send an email. The values of the parameters should be as follows:

- **to** Email address of receiver
- **template** Template name (see **Email templates** below)
- **data** Template data
- **headers** Optional - this is an array of additional headers to include in the email. For example: `{'X-Key1': 'value', 'X-Key2': 'value'}`

## 1.8.3 Send an email through an API call

The API call [GrapeSendMail \(/grape/send\\_mail\)](#) can be called from a client application to send an email. See [GrapeSendMail](#) for more information.

## 1.8.4 Custom headers

Headers defined in the "headers" input will be used to set SMTP headers. Commonly used custom headers:

- **From:** Change the From field
- **Reply-To:** Set the Reply-To field

## 1.8.5 Email templates

Email templates live in the **email\_template\_directory** defined in the app's config. Each template has 4 files (each starting with the specified **templatename**):

- **templatename.subject** - To generate the subject

- **templatename.text** - To generate the plain-text body of the email
- **templatename.html** - To generate the HTML body of the email
- **templatename.attachments** - To generate a list of attachments to include in the email

Underscore's template engine is used. The data sent to **grape.send\_email(to, template, data)** is accessible inside the template files. For example, if an email is called with the following data:

```
{ "firstname": "Piet" }
```

The field `firstname` is accessible inside of the templates using `<%= firstname %>`



## 1.8.6 Example

A typical welcome email will have the following templates (assuming the template name is **welcome**):

**welcome.subject:**

```
Hi <%= firstname %>! Welcome to <%= product_name %>
```

**welcome.text:**

```
Hi <%= firstname %>!  
Welcome to <%= product_name %>.  
  
Your login details are as follows:  
Username: <%= username %>  
Password: <%= password %>  
  
Goodbye
```

The HTML file is optional, and follows the same pattern.

This template must be called with a JSON object containing at least fields **firstname**, **product\_name**, **username** and **password**. The send this email, call the `grape.send_email` function:

```
SELECT grape.send_email('piet@platsoft.net', 'welcome',  
'{"firstname": "Piet",  
"product_name": "Some System",  
"username": "Piet",  
"password": "Piet123"}'::JSON);
```

## 1.9 Generating PDF Files

To generate a PDF document in a Grape application, we use [Apache FOP](#). The generation process works as follows:

1. A database function returning XML text is defined
2. An XSL file is created (in [app.config.xsl\\_directory](#)) that transforms the output XML of the database function into XSL-FO
3. An API handler for the specific document type is defined, for example `/download/policy_document/:policy_id`
4. The API handler calls `app.get('pdfgenerator').generate_and_stream_xml` to generate and stream the PDF

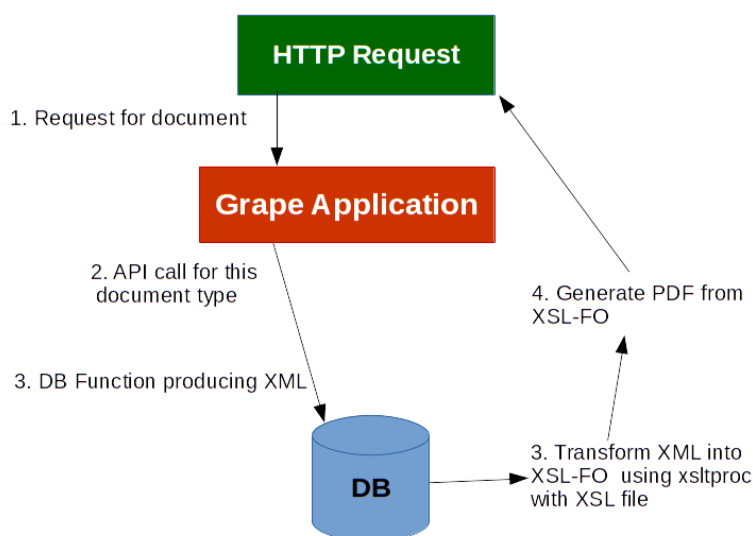


Fig. 3: PDF Generation Process

### 1.9.1 XML-producing SQL function

A function returning TEXT containing XML.

### 1.9.2 XSL file

### 1.9.3 API handler

The API handler will usually make a call to `app.get('pdfgenerator').generate_and_stream_xml(options)`. The options are:

```
{
  res: res, // The HTTP response object
  funcName: 'sql_function', // The SQL function name
  funcParams: [params], // Array of parameters passed to the function
  documentType: 'document_type', // An identifier for this document type. Files will be
                                //created in subdirectories with this name
  baseFileName: 'filename', // A filename without any extension, describing this
                            //document being created
  xslName: 'xsl_filename', // The XSL file to use for transformation
}
```

#### Notes:

- The XSL file must be located in the application's [xsl directory](#). If a [site name](#) is configured, an overriding file can be stored in a subdirectory with the same name (as the sitename)
- The function **funcName** must return an XML document
- An XSL parameter **xsldirname** will be defined and have the value of the directory name that the XSL file is located in. This can be used when images needs to be referenced. For example:

```
<fo:external-graphic content-width="200mm" scaling="uniform">
  <xsl:attribute name="src"><xsl:value-of select="$xsldirname" />/image.png</xsl:attribute>
</fo:external-graphic>
```

## 1.10 User IP Filter

**Note!** This feature was added in Grape 1.0.7

It is possible to prevent users from logging in unless the connection is made from a whitelisted IP address or network. For this to work, the Grape setting [user\\_ip\\_filter](#) needs to be set to **true**, and the user be added to a IP network. The tables `grape.network` and `grape.user_network` is used to define IP networks, and link users to it. The GrapeUI component [User Network Management](#) [TODO] contains pages to administrate these tables.

## 1.11 Grape Notifications

**Note!** This feature was introduced in Grape 1.0.8

Grape supports system notifications sent through requests HTTP headers. This provides a more efficient and convenient way to provide users with updates regarding messages or the system status in general (for example warnings before going into maintenance mode). Notifications will be included if:

1. The **X-Notifications** header is included in the request with a value
2. The [enable\\_notifications](#) grape config option is enabled

### 1.11.1 Defining notification functions

Functions are registered in `grape.notification_function`, and must return a JSONB or NULL. All non-empty results will be sent back to the client (Base64 encoded) in the X-Notifications header. The array contains objects with two fields, **event\_name** (containing the value of the **emitted\_event\_name** field) and **data**, containing the result of the notification function.








| grape.notification_function  |         |        |
|--|---------|--------|
|  notification_function_id | serial  | « pk » |
|  description              | text    |        |
|  function_name            | text    |        |
|  function_schema          | text    |        |
|  active                   | boolean |        |
|  emitted_event_name       | text    |        |
|  nf_active_idx            | index   |        |

Fig. 4: Notification Function Table

### 1.11.2 Notifications in GrapeUI

Add the following code to the JS in your project (`public/js/notification_handler.js`):

```
$(function() {
    window.Grape.Notifications.setup(); // Enable notifications

    window.Grape.Notifications.on('notifications', function(notifications) {
        // Handler for all notifications. notifications will be an array of notifications
    });

    window.Grape.Notifications.on('maintenance_mode', function(notifications) {
        // Handler for notifications with event_emitted_name maintenance_mode
    });
});
```

## 1.12 Grape settings

Grape stores internal settings in the table `grape.setting`. The following functions can be used to read and manipulate these settings:

1. **grape.set\_value (name TEXT, value TEXT)** - sets the value of setting **name** to **value**
2. **grape.get\_value (name TEXT, default\_value TEXT)** - returns the value of the setting **name**, or if it does not exist returns **default\_value**
3. **grape.setting (name TEXT, default\_value TEXT)** - alias for **grape.get\_value (name TEXT, default\_value TEXT)**

### 1.12.1 Known Grape settings

| NAME                | DESCRIPTION   | DEFAULT VALUE |
|---------------------|---|---------------|
| allow_default_paths | If a path is not found and this setting is true, access will be granted | false         |

| NAME                     | DESCRIPTION  | DEFAULT VALUE |
|--------------------------|--|---------------|
| authentication_url       | URL for the server that will grant access to users for this system     | local         |
| dataimport_in_background | Run data import processing functions in the background                 | false         |
| data_upload_schema       | Default schema for data import tables                                  | grape         |
| disable_passwords        | If true, authentication will not check whether the password is correct | false         |
| filter_processes         | Apply role based filtering on processes                                | false         |
| grape_version            | Current Grape version  |               |
| hash_passwords           | Indicate whether passwords in grape.user is hashed                     | false         |
| product_name             | Name of the current system   |               |
| product_uuid             | Unique identifier for the product/system                               |               |
| product_version          | Product version  |               |
| system_url               | URL to access system's frontend  |               |
| service_name             | Local service name (for authentication)                                |               |
| user_ip_filter           | Enable user IP filters   | false         |

## 1.13 Grape config file

The following options are recognized in the config passed to Grape:

| NAME              | DESCRIPTION   | DEFAULT VALUE |
|-------------------|---|---------------|
| dburi             | DB connection settings  |               |
| guest_dburi       | DB connection settings for guest users  |               |
| super_dburi       | DB connection used when setting up database (by grape-db-setup)   |               |
| api_directory     | Directory (or array of directories) containing API files  |               |
| db_definition     | Array containing directories with DB definitions. Subdirectories schema, function, view and data will be traversed when recreating a database using the setup_database tool |               |
| sql_dirs          | Array containing directories with DB definitions. All subdirectories will be recursively read   |               |
| pg_temp_directory | Path to a directory to which both PostgreSQL and the running node process has write access  |               |

| NAME                     | DESCRIPTION  | DEFAULT VALUE                         |
|--------------------------|--|---------------------------------------|
| port                     | Port on which the UI will be available   |                                       |
| http_port                | If this is set, and HTTPS is enabled (use_https), then a normal HTTP server will listen on this port |                                       |
| public_directory         | Directory containing public files  |                                       |
| public_directories       | List of directories containing public files  |                                       |
| debug                    |  | true                                  |
| maxsockets               | Controls the maximum number of sockets supported   | 500                                   |
| instances                | Controls the number of listener applications started   | 1                                     |
| bordeaux_config_file     | Path to Bordeaux config file   | __dirname + '/bordeaux_config.json'   |
| document_store           | Path to document store   |                                       |
| use_https                | Enable or disable HTTPS. sslkey and sslcert need to be set up correctly                              | false                                 |
| session_management       | Enable or disable session management   | true                                  |
| smtp                     | SMTP settings for GrapeMailer  |                                       |
| server_timeout           | The number of milliseconds of inactivity before a socket is presumed to have timed out               | 50000                                 |
| sslkey                   | Path to private SSL key file   | __dirname + '/cert/private.pem'       |
| sslcert                  | Path to private SSL public certificate   | __dirname + '/cert/public_nopass.pem' |
| hr_system                | URL to get access to the Bordeaux system running on Savanna HR system                                | https://192.168.50.86:3999/           |
| email_template_directory | Path to email templates (See GrapeMailer for more information)                                       | __dirname + '/email_templates'        |
| compile_js_dirs          | List of directory names that will be recursed when all JS is being compiled                          | ['pages']                             |
| site_name                | If multiple sites are used (for example different XSL files for documents), specify the site         |                                       |
| xsl_directory            | Directory containing XSL files for PDF generation  |                                       |
| fop                      | Path to the Apache FOP binary (for generation of PDF files)  |                                       |
| ps_bgworker              | Path to ps_bgworker binary   |                                       |

| NAME                 | DESCRIPTION   | DEFAULT VALUE |
|----------------------|---|---------------|
| cache_public_js_dirs | If enabled, the compiled JS files from public directories will not be cached            |               |
| process_name         | Sets the visible process name   |               |
| enable_notifications | Enables the notification subsystem  |               |
| enable_ldap_server   | Enable LDAP server  |               |
| ldap_server_port     | LDAP server port to listen on   |               |
| delayed_response     | Delay all responses with this many milliseconds (can be used to simulate slow networks) | 0             |

## 1.14 Standardized Error Codes

| CODE | DESCRIPTION              |
|------|--------------------------|
| -1   | Unknown Error            |
| -2   | Permission Denied        |
| -3   | Invalid Input            |
| -5   | Requested data not found |
| -99  | Database Error           |

## 1.15 grape-manage

**grape-manage** is a tool used to provide a common interface for custom scripts and commands that operates in a project's directory. It can be installed using `npm install -g ps-grape`. Commands can be added per-project, and the command should be executed from within the project directory. **grape-manage** will scan `$CWD/cmd` and `$CMD/scripts/cmd/` for commands. The program comes with the following built-in commands:

- **list** - Lists all available commands
- **start** - Starts the Node server
- **stop** - Stops the Node server
- **status** - Prints status information regarding the Node server
- **create\_user** - Create a new user on the system. Run `grape-manage create_user --help` for parameters
- **create\_users** - Create new users on the system from a CSV file. Run `grape-manage create_users -f [FILENAME]`
- **list\_users** - Lists users on the system
- **print\_db\_settings** - Prints DB connection settings for usage in scripts
- **bgworker** - Provides various functions for interacting with [ps\\_bgworker](#)

### 1.15.1 Adding new commands

**grape-manage** will scan `$CWD/cmd` and `$CMD/scripts/cmd/` for commands. Commands are JavaScript files that are require'd, and should export at least an **info** object and a **run** function. The **info** object must contain two parameters: a **description** text field,

and a boolean **db** - which will, if true, provide the **run** function with a database connection. The **run** function is called with 2 parameters: **options** and a **callback**. **options** will contain:

- **db** - Database connection (null if db option passed is false)
- **argv** - Arguments provided to run this script (minus the grape-manage part)
- **options** - The project's config options (read from config.js)
- **funcs** - Some handy utility functions: print\_ok, print\_error, print\_warn and align
- **base\_directory** - Project's base directory

The name of the file will be used as the name of the command. Upon completion, the callback should be called (the command can optionally provide error information). The basic scaffolding for a command is:

```
var GrapeCmd = {};
GrapeCmd.info = {
  description: 'Command description',
  db: true
};

GrapeCmd.run = function(opts, cb) {
  // LOGIC GOES HERE
  // WE HAVE ACCESS TO: opts.db, opts.argv, opts.options, opts.funcs and opts.base_directory
  cb(null);
};

module.exports = GrapeCmd;
```

## 1.15.2 Bash auto completion

Basic auto completion for grape-manage in bash can be achieved by installing the file [grape-manage-bash-completion.sh](#) into `/etc/bash_completion.d/`.

## 1.16 Built-in API calls

### 1.16.1 GrapeDeleteRecord

Delete a record from a table (the table needs to be whitelisted)

**URL:** `/grape/delete_record`

**Method:** POST

**SQL Function:** [grape.delete\\_record](#)

**Input:**

- *JSON object containing the following fields:*
  - **tablename** *STRING* Table name to delete the record from
  - **schema** *STRING* Schema name of table
  - **filter** *JSON* Key-value pair of lookup values (the WHERE clause)

**Example input:**

```
{
  "schema": "public",
  "tablename": "product",
  "filter": {
    "description": "Refining Mist (200ml)"
  }
}
```

```
}
}
```

**Output:**

- *JSON object containing the following fields:*
  - *return* **JSON** On success, the returning values (as specified by the input field returning)
  - *status* **ENUM** Status indicator Possible values: **ERROR, OK**
  - *message* **STRING** Error message
  - *code* **NUMBER** Error code
  - *error* **JSON** Error details

## 1.16.2 GrapeInsertRecord

Inserts a new record into a whitelisted table

**URL:** /grape/insert\_record

**Method:** POST

**SQL Function:** [grape.insert\\_record](#)

**Input:**

- *JSON object containing the following fields:*
  - *tablename* **STRING** Table name to insert record into
  - *schema* **STRING** Schema name of table
  - *values* **JSON** Key-value pair of values for the new record
  - *returning* **STRING** Column value of new record to return (or \* for all)

**Example input:**

```
{
  "schema": "public",
  "tablename": "product",
  "values": {
    "description": "Refining Mist (200ml)"
  },
  "returning": "*"
}
```

**Output:**

- *JSON object containing the following fields:*
  - *return* **JSON** On success, the returning values (as specified by the input field returning)
  - *status* **ENUM** Status indicator Possible values: **ERROR, OK**
  - *message* **STRING** Error message
  - *code* **NUMBER** Error code
  - *error* **JSON** Error details

## 1.16.3 GrapeListQuery

List records from a table or view

**URL:** /grape/list

**Method:** POST

**SQL Function:** [grape.list\\_query](#)



**Input:**

- *JSON object containing the following fields:*
  - *tablename* **STRING** Table or view name
  - *schema* **STRING** Schema name of table or view
  - *sortfield* **STRING** optional Field to order by
  - *limit* **NUMBER** optional Record limit default 50
  - *offset* **NUMBER** optional Record offset default 0
  - *filter* **ARRAY of objects:**
    - *field* **STRING** Field to filter on
    - *operand* **STRING** One of '=', '>', '<', '>=', '<=', 'LIKE', 'ILIKE'
    - *value* **STRING** Filter value

**Example input: Output:**

- *JSON object containing the following fields:*
  - *result\_count* **NUMBER** Number of results returned
  - *offset* **NUMBER** Result offset
  - *limit* **NUMBER** Results limit
  - *records* Array of returned records **ARRAY of JSON objects**
  - *total* **NUMBER** Total number of records in the database (after filter has been applied)

## 1.16.4 GrapeSendMail

Sends an email

**URL:** /grape/send\_mail

**Method:** POST

**SQL Function:** [grape.send\\_email](#)

**Input:**

- *JSON object containing the following fields:*
  - *to* **STRING** Email address of recipient
  - *template* **STRING** Template Name
  - *template\_data* **JSON**
  - *headers* **JSON**

**Example input: Output:**

- *JSON object containing the following fields:*
  - *status* **ENUM** Status indicator Possible values: **ERROR, OK**
  - *message* **STRING** Error message
  - *code* **NUMBER** Error code
  - *error* **JSON**

## 1.16.5 GrapeUpdateRecord

Updates a record in a table (the table needs to be whitelisted)

**URL:** /grape/update\_record

**Method:** POST

**SQL Function:** [grape.update\\_record](#)

**Input:**

- *JSON object containing the following fields:*

- **tablename** **STRING** Name of the table to update
- **schema** **STRING** Schema name of table
- **filter** **JSON** Key-value pair of lookup values (the WHERE clause)
- **values** **JSON** Key-value pair of values for the new data
- **returning** **STRING** Column value of new record to return (or \* for all)

**Example input:**

```
{
  "schema": "public",
  "tablename": "product",
  "filter": {
    "description": "Refining Mist (200ml)"
  },
  "values": {
    "description": "Refining Mist (250ml)"
  },
  "returning": "*"
}
```

**Output:**

- *JSON object containing the following fields:*
  - **return** **JSON** On success, the returning values (as specified by the input field returning)
  - **status** **ENUM** Status indicator Possible values: **ERROR, OK**
  - **message** **STRING** Error message
  - **code** **NUMBER** Error code
  - **error** **JSON** Error details

### 1.16.6 GrapeUserSave

Save a user, or create one if it does not currently exist

**URL:** /grape/user/save

**Method:** POST

**SQL Function:** [grape.user.save](#)

**Input:**

- *JSON object containing the following fields:*
  - **user\_id** **NUMBER** User ID to update
  - **username** **STRING** Username
  - **fullnames** **STRING** Full names
  - **email** **STRING** Email address of user
  - **password** **STRING** Password
  - **active** **BOOLEAN** Indicate if user is active (can login)
  - **role\_names** **ARRAY of STRING**
  - **employee\_guid** **STRING**
  - **employee\_info** **JSON**

**Example input: Output:**

- *JSON object containing the following fields:*

- *status* **ENUM** Possible values: **ERROR, OK**
- *user\_id* **NUMBER** User ID of new user
- *message* **STRING**
- *code* **NUMBER**
- *error* **JSON**

### 1.16.7 download\_public\_js\_files

The **download\_public\_js\_files** API call is a special API call that will traverse all subdirectories in the public directories (defined by *public\_directories*), with the names defined by *compile\_js\_dirs*. The default values for *compile\_js\_dirs* is **pages**. This means all subdirectories named "pages" will be traversed for JS files, and served through this call.

## 1.17 Grape SQL Functions

### 1.17.1 API result functions

These functions deals with the creation of standardized API results (in JSON format) to be sent back to the API call. They can be found in [api\\_result\\_json.sql](#)

| NAME                         | PARAMETERS  | DESCRIPTION  |
|------------------------------|---|--|
| grape_result_type            | <i>success</i> BOOLEAN<br><i>reason</i> TEXT<br><i>data</i> JSON  | Grape result types.  |
| api_result_error             | <i>message</i> TEXT<br><i>code</i> INTEGER<br><i>error</i> JSON   | Returns a standardized JSON error object with status as "ERROR" and the other fields populated. Example:<br><pre>{"status": "ERROR", "message": "Message", "code": -2, "error": {}}</pre>  |
| api_error                    | <i>message</i> TEXT<br><i>code</i> INTEGER<br><i>error</i> JSON   | Overload for <i>api_result_error</i> .   |
| api_error                    |   | With no arguments, an "Unknown error" message will be generated. Example:<br><pre>{"Unknown error", -1}</pre>  |
| api_error_invalid_input      | <i>info</i> JSON  | Similar to calling <code>api_result_error("Invalid input", -3)</code>  |
| api_error_invalid_field      | <i>name</i> TEXT  | Similar to calling <code>api_result_error("Missing or invalid field: ", -3)</code>   |
| api_error_permission_denied  | <i>info</i> JSON  | Similar to calling <code>api_result_error("Permission denied", -2)</code>  |
| api_error_data_not_found     | <i>info</i> JSON  | Similar to calling <code>api_result_error("Data not found", -5)</code>   |
| api_error_invalid_data_state | <i>info</i> JSON  | Similar to calling <code>api_result_error("The operation requested could not be performed on the data because the data is not in a valid state", -6)</code>  |
| api_success                  | <i>keys</i> TEXT[]<br><i>values</i> TEXT[]<br><i>types</i> TEXT[] | This function will construct a JSON object containing at least one field, "status" with the value "OK". The 3 input parameters should be arrays containing additional keys, values and the associated types ( <i>n/i/number/integer, j/json or nothing for text</i> ). |

| NAME                    | PARAMETERS   | DESCRIPTION   |
|-------------------------|--|---|
| api_success             | <i>keys</i> TEXT<br><i>values</i> INTEGER  |   |
| api_success             | <i>key1</i> TEXT<br><i>val1</i> INTEGER  | Create an API result success JSON object with one integer field added.  |
| api_success             | <i>key1</i> TEXT<br><i>val1</i> INTEGER<br><i>key2</i> TEXT<br><i>val2</i> INTEGER | Create an API result success JSON object with two integer fields added.   |
| api_success             | <i>key1</i> TEXT<br><i>val1</i> JSON   | Create an API result success JSON object with a JSON field merged into the result.  |
| api_success             |  | Returns a API result object with a "status" field set to "OK".  |
| api_success_if_not_null | <i>fieldname</i> TEXT<br><i>data</i> JSON  | Returns success message when data is <b>NOT NULL</b> , otherwise it returns <code>grape.api_error_data_not_found()</code>   |
| api_result              | <i>res</i> <code>grape_result_type</code>  | Returns error message similar to calling <code>api_error(res.reason, -1)</code> if false, otherwise returns success message similar to calling <code>api_success("data", res.data)</code> |

## 1.17.2 Data importing functions

These functions deals with how data importing is handled. They can be found in `data_import.sql`

| NAME                    | PARAMETERS  | DESCRIPTION  |
|-------------------------|---|--|
| upsert_data_import_type | <i>processing_function</i> TEXT<br><i>short_description</i> TEXT<br><i>file_format_info</i> TEXT<br><i>function_schema</i> TEXT<br><i>param_definition</i> JSON | Upsert data import types. If processing_function name is the same, all other values are updated.   |
| estimate_datatype       |   | Overloaded function to estimate the potential datatype of a text value. Function returns the data_type. Example:<br><code>{"NULL", "INTEGER", "NUMERIC", "DATE", "TIMESTAMP", "TIMESTAMPZ"}</code> |
| data_import_insert      |   | API function to insert a data_import entry.<br>Returns success message similar to calling <code>api_success("data_import_id")</code>   |
| data_import_delete      |   | API function to delete a data_import entry. Returns success message similar to calling <code>api_success()</code>  |
| data_import_row_insert  |   | API function to insert a row of JSON into data_import_row.<br>Required field data_import_id must be in the JSON data.  |
| data_import_done        |   | API function to notify server that insertion of all the rows has been completed and timestamp this completion. Returns message similar to calling <code>api_success("data_import_id")</code>       |
| data_import_process     | <i>data_import_id</i> INTEGER   | Internal function to process data_import data. Returns message based on the following data import status:<br><code>0 - Empty</code><br><code>1 - Populated</code>                                  |

| NAME                          | PARAMETERS  | DESCRIPTION  |
|-------------------------------|---|--|
|                               |   | <b>2 - Process started</b><br><b>3 - Some not processed</b><br><b>4 - Processed</b>  |
| data_import_process           |   | API function to process data_import data. Calls internal process function. Returns error message similar to calling <code>api_error("data_import_process failed", -1)</code> if false, otherwise returns success message similar to calling <code>api_success()</code> |
| data_import_test_table_insert |   | API function to create a test table from data_imports data.  |
| data_import_test_table_drop   |   | API function to drop a test table from data_imports data.  |
| data_import_build_result      | <b>status</b> TEXT  | Builds an object in the form of <code>{"result": {"status": "OK"}}</code> for returning from data import functions.  |
| data_import_build_result      | <b>status</b> TEXT<br><b>shared_data</b> JSON             | Builds a object in the form of <code>{"result": {"status": "OK"}, "shared_data": {}}</code> for returning from data import functions.  |
| data_import_build_result      | <b>status</b> TEXT<br><b>shared_data</b> JSONB            | Builds a object in the form of <code>{"result": {"status": "OK"}, "shared_data": {}}</code> for returning from data import functions.  |
| data_import_reset             | <b>data_import_id</b> INTEGER                             | Resets data_import_id's data import status to 1, if populated.   |
| data_import_test_table_alter  |   | API function that returns message similar to calling <code>api_success()</code>  |
| dimport_generic               | <b>data_import</b> grape.data_import<br><b>args</b> JSONB | Example dimport function that does not process the data in any way and allows for a way to create a test table with data that does not need to be processed. Returns message similar to calling <code>data_import_build_result("OK")</code>                            |
| upsert_data_import_type       |   | This function does not actually process the data in any way, but is a way to allow you to import data with which you may create test tables in grape.  |
| proc_process_data_import      |   | Process to process data import files in the background via ps_bgworker.  |

### 1.17.3 JSON helpers

These functions are JSON helpers and can be found in several files: `json2xml.sql`, `json_diff.sql`, `json_to_composite_type_text.sql`, `cast_json_array_to_int_array.sql`, `cast_json_array_to_text_array.sql`

| NAME             | PARAMETERS                           | DESCRIPTION  |
|------------------|--------------------------------------|--|
| json2xml         | <b>data</b> JSON<br><b>root</b> TEXT | Converts JSON object to xml.   |
| json_diff        | <b>old</b> JSONB<br><b>new</b> JSONB | Compares two JSON objects and returns an object containing fields that are different between the two objects. If a field exists in <b>j_old</b> , but not in <b>j_new</b> , it is not included in the results. If a field exists in <b>j_new</b> , but not in <b>j_old</b> , it is included in the results. If a field is different, <b>j_new</b> is chosen. |
| json_diff        | <b>old</b> JSON<br><b>new</b> JSON   |  |
| json_object_diff | <b>old</b> JSONB<br><b>new</b> JSONB | Compares two JSON objects and return any values that exists in <b>_new</b> but not in <b>_old</b> .  |

| NAME                          | PARAMETERS   | DESCRIPTION  |
|-------------------------------|--|--|
| json_array_diff               | <i>old</i> JSONB<br><i>new</i> JSONB                                     | Compare two JSON arrays and return any values that exists in <b>_new</b> but not in <b>_old</b> .  |
| json_to_composite_type_text   | <i>target_schema</i> TEXT<br><i>target_type</i> TEXT<br><i>data</i> JSON | Converts JSON object to composite type text.   |
| json_to_composite_type        | <i>target_schema</i> TEXT<br><i>target_type</i> TEXT<br><i>data</i> JSON | This function will populate a custom type from a JSON object. Multi-level nested objects are supported.  |
| cast_json_array_to_int_array  | JSON data  | Provides an implicit cast from JSON to <b>INT[]</b> ( <a href="#">cast_json_array_to_int_array.sql</a> ).<br><br><pre>#select cast_json_array_to_int_array('[1,2,3] '::JSON); cast_json_array_to_int_array ----- {1, 2, 3}</pre>                 |
| cast_json_array_to_text_array | JSON data  | Provides an implicit cast from JSON to <b>TEXT[]</b> ( <a href="#">cast_json_array_to_text_array.sql</a> ).<br><br><pre>#select cast_json_array_to_text_array('[aa,bb,cc] '::JSON); cast_json_array_to_text_array ----- {"aa", "bb", "cc"}</pre> |

### 1.17.4 List query

Grape's `list_query` call provides an easy way to retrieve rows from a table. Before the contents of a table can be retrieved this way it needs to be added to a whitelist. This functions can be found in [list\\_query.sql](#). The built-in API call to access this function is **/grape/list**. Access control is enforced on tables retrieved.

The **grape.list\_query** function returns rows from a database table. The following input fields are recognized:

- `tablename` TEXT
- `schema` (optional) TEXT
- `sortfield` (optional) TEXT
- `sortorder` (optional) TEXT DESC
- `limit` (optional) INTEGER (DEFAULT 50)
- `offset` (optional) INTEGER (DEFAULT 0)
- `filter` (optional) array of fields:
  - `field` TEXT
  - `operand` TEXT of '=', '>', '<', '>=', '<=', 'LIKE', 'ILIKE', 'IS\_NULL', 'IS\_NOT\_NULL', 'IN'
  - `value` TEXT

The following functions deals with the access control:

| NAME                           | PARAMETERS  | DESCRIPTION  |
|--------------------------------|---|--|
| grape.list_query_whitelist_add | <i>schema</i> TEXT<br><i>tables</i> TEXT[] - A list of table names to allow<br><i>roles</i> TEXT[] - A list of roles to allow | Adds tables to the whitelist for use in grape list_query. Users must be in <b>_roles</b> to be able to access the data in the table. |

| NAME                              | PARAMETERS   | DESCRIPTION                                   |
|-----------------------------------|--|---|
| grape.list_query_whitelist_delete | <i>schema</i> TEXT<br><i>tablename</i> TEXT - A table to remove from allow | Removes a table from the whitelist.           |
| grape.list_query_check_permission | <i>schema</i> TEXT<br><i>tablename</i> TEXT                                | Check permission on a table for current user. |

## 1.17.5 Table Operations

Grape provides three API calls to perform generic DML (INSERT, UPDATE and DELETE) on whitelisted tables.

The API calls are:

- [GrapeInsertRecord](#)
- [GrapeUpdateRecord](#)
- [GrapeDeleteRecord](#)

The SQL function used to whitelist tables, is:

```
grape.table_operation_whitelist_add(schema TEXT, tables TEXT[], roles TEXT[], allowed_operation TEXT) .
```

- *schema* - The schema of the table
- *tables* - An array of table names to add
- *roles* - An array of role names to allow
- *allowed\_operation* - The operation to allow (INSERT, UPDATE or DELETE)

## 1.17.6 Reports

These functions can be found in [reports.sql](#)

| NAME                   | PARAMETERS  | DESCRIPTION   |
|------------------------|---|---|
| save_report            | <i>report_id</i> INTEGER<br><i>name</i> TEXT<br><i>description</i> TEXT<br><i>function_schema</i> TEXT<br><i>function_name</i> TEXT<br><i>input_fields</i> JSON |   |
| save_report            | <i>name</i> TEXT<br><i>function_name</i> TEXT<br><i>description</i> TEXT<br><i>input_fields</i> JSON  |   |
| save_report            | <i>settings</i> JSON  |   |
| execute_report         | <i>report_id</i> INTEGER<br><i>parameters</i> JSON  |   |
| execute_report         | <i>parameters</i> JSON  | JSON object needs name field (with report name) and optional JSON parameters. |
| execute_report_to_file | <i>report_id</i> INTEGER<br><i>reports_executed_id</i> INTEGER<br><i>parameters</i> JSON  | Function to convert a report to a file.                                       |

## 1.17.7 User related functions

| NAME                          | PARAMETERS  | DESCRIPTION   |
|-------------------------------|---|---|
| grape.user_save               | JSON containing:<br><i>user_id</i> INTEGER<br><i>username</i> TEXT<br><i>password</i> TEXT<br><i>email</i> TEXT<br><i>fullnames</i> TEXT<br><i>active</i> BOOLEAN (optional)<br><i>role_names</i> TEXT[]<br><i>employee_guid</i> UUID | Save a user field, or create a new user. API call:<br><b>POST /grape/user/save</b>  |
| grape.new_user                | <i>user_id</i> INTEGER<br><i>rec</i> RECORD<br><i>role_name</i> TEXT  | Creates a new user. Returns the user ID, or error code -1 if it does not exist.   |
| grape.username                | <i>user_id</i> INTEGER  | Returns the username for a user ID, or <b>NULL</b> if it does not exist.  |
| grape.user_id_from_name       | <i>username</i> TEXT  | Returns the user ID for a username, or <b>NULL</b> if it does not exist.  |
| grape.user_id_from_fullnames  | <i>fullnames</i> TEXT   | Returns the user ID for a user found by fullnames, or <b>NULL</b> if it does not exist.   |
| grape.username_from_fullnames | <i>fullnames</i> TEXT   | Returns the username for a user found by fullnames, or <b>NULL</b> if it does not exist.  |
| grape.hash_user_password      | <i>user_id</i> INTEGER  | Hashes a password for user and updates the user table afterwards.<br><ol style="list-style-type: none"><li>If the hash length is the same as the password length and the password starts with a '\$' sign, it is assumed that the password is already hashed and the update is ignored (return -1)</li><li>If grape.setting passwords_hashed isn't true, nothing is done (return -2)</li><li>On success 0 is returned</li></ol> |
| grape.hash_user_password      | <i>username</i> TEXT  | Overload for <b>grape.hash_user_password (user_id INTEGER)</b>  |
| grape.set_user_password       | <i>user_id</i> INTEGER<br><i>password</i> TEXT<br><i>is_hashed</i> BOOLEAN  | Set user password. If the password given to this function is already hashed then <i>is_hashed</i> should be <b>TRUE</b> .   |

## 1.17.8 Session related functions

| NAME                       | PARAMETERS  | DESCRIPTION  |
|----------------------------|---|--|
| grape.current_user_roles   |   | Returns a list of all roles the current user belongs to.   |
| grape.current_user_in_role | <i>role</i> TEXT  | Returns <b>TRUE</b> if the current user belongs to <i>_role</i> .  |
| grape.current_user_in_role | <i>roles</i> TEXT[]   | Returns <b>TRUE</b> if the current user belongs to any of <i>_roles</i> .  |
| grape.current_user_id      |   | Returns the integer value of the current session's "grape.user_id" setting. This is typically set with grape before any API call is called.                              |
| grape.check_session_access | <i>session_id</i> TEXT - Session ID to check for<br><i>check_path</i> TEXT - Access path to check<br><i>check_method</i> TEXT - HTTP method to check (GET/POST) | This function performs access control on an API call (based on the path and session ID). It is automatically called by the express app before any API call is performed: |



| NAME                                     | PARAMETERS                                       | DESCRIPTION  |
|--|--|--|
|  |  | <ol style="list-style-type: none"> <li>1. Check that the path has access control on it. If it cannot be found, the grape setting <code>default_access_allowed</code> is checked, and if true, access will be granted. If not, it will be denied and code 9 will be returned.</li> <li>2. If the path has a role 'guest' granted access to it, everyone will be allowed (even if the session is invalid).</li> <li>3. If the session is invalid, access will be denied and code 1 returned.</li> <li>4. If the path has a role 'all', only, and all, valid sessions will be granted access.</li> <li>5. If the user has access granted to the access path's role, access is granted.</li> <li>6. If all the above fails, access is denied with code 2.</li> </ol>                   |
| grape.session_insert                     | <b>user_id</b> INTEGER<br><b>ip_address</b> TEXT | <p>This function requires <code>user.sql</code>.</p> <ol style="list-style-type: none"> <li>1. Input:             <ul style="list-style-type: none"> <li>• username or email</li> <li>• password</li> <li>• ip_address</li> <li>• persistent TRUE/FALSE (optional)</li> </ul> </li> <li>2. Status:             <ul style="list-style-type: none"> <li>• status = ERROR</li> <li>• code 1 = No such user</li> <li>• code 2 = Wrong password</li> <li>• code 3 = User is inactive</li> <li>• code 4 = IP not allowed</li> </ul> </li> <li>3. On success, status = OK and following fields returned: <b>session_id</b>, <b>user_id</b>, <b>username</b> and <b>user_roles</b>.</li> <li>4. Setting <b>hash_passwords</b> is used to decide if passwords are hashed or not.</li> </ol> |
| grape.logout                             | <b>JSON</b> JSON containing session_id           | API call:<br><a href="#">/grape/logout</a>   |
| grape.session_ping                       | <b>JSON</b> JSON containing session_id           | Checks validity of a session and returns a JSON object containing the session's username, user_id, fullnames, email, GUID and user_roles.<br>API call:<br><a href="#">/grape/session_ping</a>  |
| grape.session_insert                     | <b>username</b> TEXT<br><b>password</b> TEXT     | <p>This function inserts a new session for a valid username and password provided.</p> API call:<br><a href="#">/grape/logout</a>  |
| grape.create_session_from_service_ticket |  | Function which creates a session from the service ticket.  |
| grape.set_password_with_service_ticket   |  | Function which sets the password with the service ticket.  |
| grape.logout                             |  | Function used to logout user from current session.   |
| grape.ping                               |  | Function used to ping current session.   |
| grape.set_session_user_id                | <b>user_id</b> INTEGER                           | Function which sets the <b>user_id</b> for the current session.  |

| NAME                       | PARAMETERS           | DESCRIPTION   |
|----------------------------|----------------------|---|
| grape.set_session_username | <b>username</b> TEXT | Function which sets the <b>user_name</b> for the current session. |

## 1.17.9 Other utility functions

| NAME                   | PARAMETERS                                      | DESCRIPTION   |
|------------------------|---|---|
| month_diff             | <b>_d1</b> DATE<br><b>_d2</b> DATE              | Returns an integer containing the number of months between the two dates provided. If the first parameter is after the second (higher date), the return value will be negative. |
| set_value              | <b>_name</b> TEXT<br><b>_value</b> TEXT         | Sets the value (insert if new, replace if exist) in the key-value pair table <code>grape.setting</code> returning <b>_value</b> .   |
| get_value              | <b>_name</b> TEXT<br><b>_default_value</b> TEXT | Gets the value for setting <b>_name</b> , and if not found it will return <b>_default_value</b> . Defined in <a href="#">setting.sql</a>  |
| generate_uuid          |   | Generates a unique UUID (for example b1086d35-e973-4356-3adc-2eeb6f4963e2). Defined in <a href="#">uuid.sql</a>   |
| array_lowercase        | TEXT[]  |   |
| clean_telephone_number | <b>_tel</b> TEXT                                | Returns a text containing only numbers.   |
| random_string          | <b>length</b> INTEGER                           | Generates a random string of <b>length</b> length. Defined in <a href="#">random_string.sql</a>   |

## 1.18 Grape SQL Views

### 1.18.1 User Related Views

| NAME                   | DESCRIPTION   |
|------------------------|---|
| grape.user_access_role | List of all users, which include user details, passwords and roles assigned to each user. |
| grape.v_active_users   | Returns a list of active users and their roles.   |
| grape.v_ldap_users     | Returns a list of ldap users and their details.   |

### 1.18.2 Session Related Views

| NAME                    | DESCRIPTION   |
|-------------------------|---|
| grape.v_active_sessions | Returns a list of active sessions including user related information. |

### 1.18.3 Other Utility Views

| NAME                        | DESCRIPTION   |
|-----------------------------|---|
| grape.v_process_definitions | List of all processes, which includes important information such as their descriptions, parameters, process types, etc. |
| grape.v_pg_functions        | List of all routine schemas, which include the routine names, parameters and return types.                              |
| grape.v_pg_table_stats      | Statistical information related to all tables.  |
| grape.v_test_table          | Used for testing purposes.  |