



PLATINUM **SOFTWARE**

PlatSoft Grape Manual

Developer's guide to using the PlatSoft Grape Framework

Table of Contents

1. Grape	Page 5
1.1. Quickstart	Page 5
1.2. HTTP Listener Worker	Page 6
1.3. Resource handlers	Page 6
1.4. API Calls	Page 6
1.4.1. Registering API calls	Page 6
1.4.2. Fields of JSON Schema API call	Page 7
1.4.3. Input Validation	Page 8
1.4.3.1. Examples	
1.4.4. Database API Calls	Page 8
1.4.5. JavaScript Handlers	Page 9
1.4.6. Access control on API calls	Page 10
1.4.7. Generating API documentation	Page 10
1.4.8. Consuming an API call	Page 10
1.4.8.1. Using HTTP Authorization	
1.5. Authentication	Page 11
1.5.1. Generating user's encryption and decryption key from password	Page 11
1.5.2. TGT Request	Page 11
1.5.3. User auth_info field	Page 12
1.5.4. User IP Filter	Page 12
1.5.5. Adding TLS/SSL support	Page 13
1.5.6. Configuration options	Page 13
1.6. User Management	Page 13
1.6.1. Access Roles	Page 13
1.6.2. Creating initial users	Page 14
1.7. Usage	Page 14
1.8. Grape configuration files	Page 14
1.9. Project file structure	Page 15
1.10. Database	Page 15
1.10.1. Defining the database structure	Page 15
1.10.1.1. Database model	
1.10.1.2. Database functions	
1.10.1.3. Initial data	
1.10.1.4. Indexes	
1.10.1.5. Naming conventions	
1.10.1.6. Pitfalls to avoid	
1.10.1.7. Schemas	
1.10.2. Using NOTIFY/LISTEN	Page 16
1.11. grape-db-setup	Page 16
1.11.1. Installation	Page 17
1.11.2. Usage	Page 17
1.11.3. File order	Page 17
1.11.4. Manifest files	Page 17
1.12. Log directory	Page 18
1.12.1. Log files	Page 18
1.12.2. grape.pid	Page 18
1.12.3. grape.fifo	Page 18
1.13. Grape settings	Page 18

1.13.1. Known Grape settings	Page 19
1.14. Additional Functionality	Page 19
1.14.1. Grape Notifications	Page 19
1.14.1.1. Defining notification functions	
1.14.1.2. Notifications in GrapeUI	
1.15. Additional Modules	Page 20
1.15.1. Available Modules	Page 20
1.15.2. Generating PDF Files	Page 20
1.15.2.1. XML-producing SQL function	
1.15.2.2. XSL file	
1.15.2.3. API handler	
1.15.3. Sending Emails	Page 22
1.15.3.1. Setup	
1.15.3.2. Send an email from PostgreSQL	
1.15.3.3. Send an email through an API call	
1.15.3.4. Custom headers	
1.15.3.5. Email templates	
1.15.3.6. Example	
1.15.4. Workers	Page 24
1.15.4.1. Anatomy of a worker	
1.15.4.2. Registering a worker	
1.16. Reference	Page 24
1.17. Grape class reference	Page 24
1.17.1. Methods	Page 24
1.17.1.1. constructor (config, config, ...)	
1.17.1.2. addWorker (options)	
1.17.1.3. start ()	
1.17.2. Members	Page 25
1.17.3. Events	Page 25
1.17.3.1. worker	
1.17.3.2. worker-*	
1.17.3.3. worker-*-created	
1.17.3.4. creating_pidfile	
1.17.3.5. pidfile_created	
1.18. Standardized Error Codes	Page 25
1.19. grape-manage	Page 25
1.19.1. Adding new commands	Page 26
1.19.2. Bash auto completion	Page 26
1.20. Built-in API calls	Page 26
1.20.1. GrapeDeleteRecord	Page 26
1.20.2. GrapeInsertRecord	Page 27
1.20.3. GrapeListQuery	Page 28
1.20.4. GrapeSendMail	Page 28
1.20.5. GrapeUpdateRecord	Page 29
1.20.6. GrapeUserSave	Page 29
1.20.7. download_public_js_files	Page 30
1.21. Grape SQL Functions	Page 30
1.21.1. API result functions	Page 30
1.21.2. Data importing functions	Page 31
1.21.3. JSON helpers	Page 32

1.21.4. List query	Page 33
1.21.5. Table Operations	Page 34
1.21.6. Reports	Page 34
1.21.7. User related functions	Page 35
1.21.8. Session related functions	Page 35
1.21.9. Other utility functions	Page 37
1.22. Grape SQL Views	Page 37
1.22.1. User Related Views	Page 37
1.22.2. Session Related Views	Page 37
1.22.3. Other Utility Views	Page 37
1.23. Grape configuration reference	Page 38

1. GRAPE

Grape is a web framework, created at Platinum Software. It provides a way to define and create server-side functionality for web applications (for example API calls, worker process management, DB connectivity, etc).

Grape is installed as an NPM library, and then included in projects. A typical startup file (index.js) looks like this:

```
var Grape = require("ps-grape");

var app = new Grape.grape(
  "../config/defaults.json",
  "../config/local.json"
);

app.start();
```

When the Grape object is created, it recursively reads all the config files provided to the constructor. In the example above, it will read `../config/defaults.json` and `../config/local.json`. Configuration files can also include other configuration files.

Grape spawns a number of processes when `Grape.start()` is called. These processes perform various tasks, but arguably the most important process, aside from the master process, is the [HTTP Listener](#) process. More than one instance of a worker process can be created, and it is possible to add custom workers (see [Workers](#) for more information).

The [HTTP Listeners](#) provides a lot of functionality that is typically associated with a web framework, and in Grape it is implemented as an Express application.

[Interprocess communication](#) and watchdog functionality is implemented in the master process. If a worker process dies, the master process will restart the worker - if the error is not fatal. The IPC functionality also provides a shared memory cache between processes.

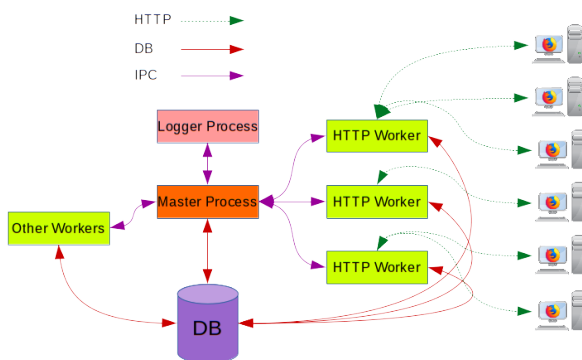


Fig. 1: Grape communications

1.1 Quickstart

The quickest way to get started with Grape is to copy the skel directory from the grape-samples repository:

```
$ git clone git:///grape-samples
$ cp -R grape-samples/skel/ ~/demo/
$ npm set registry http://npm.platsoft.net:4873/
$ npm install -g ps-grape
$ cd ~/demo/app
$ npm install
$ cd ..
$ grape-db-setup db/initial.manifest
```

```
$ node index.js
```

Now you should be able to connect to <http://localhost:9000/>

1.2 HTTP Listener Worker

This built-in worker provides most of the web functionality that is associated with Grape. The process in the process table is called **httplistener**.

The HTTP listener will listen on a port specified in the [port](#) configuration option for incoming connections. The port is usually set in one of the project's configuration files. If SSL is enabled (configuration options `use_https`, `sslkey` and `sslcert`), the connection will be served over HTTPS.

When the HTTP listener receives a new HTTP request, the following steps are performed:

1. The request is logged
2. The body parsers are called (JSON, Multipart, XML, etc)
3. The requested resource is looked up and matched to a resource handler
4. The connection's session is checked and validated against active sessions
5. The resource and requested operation is validated against the session
6. The resource handler is executed (API call executed, file fetched from disk, etc)

1.3 Resource handlers

HTTP requests are requests for operations to be performed on a specific resource. The resource is identified by the request's URL, and the operation by the request's method (for example GET or POST). For example, consider the following HTTP request:

```
GET /index.html HTTP/1.1
Host: www.example.com
```

The client (browser) is asking the server to retrieve and send the data associated with the resource `/index.html`.

Each resource has a handler associated with it. In Grape, there are two default resource handlers built-in:

1. The static file handler; and
2. [the API call handler](#)

When Grape receives the HTTP request illustrated above, it will identify it as a static file (because it is a file that exists in one of the public directories), and send the file accordingly. If the request can be matched to an API call, the API handler will be called instead.

1.4 API Calls

1.4.1 Registering API calls

API calls are defined in JSON files, and created when a Grape object is created. The files can be found in a projects **api/** directory, and looks like this:

```
{
  "description": "Calculate the square root of a number",
  "url": "/maths/sqrt",
  "name": "SquareRoot",
  "sqlfunc": "maths_sqrt",
  "method": "POST",
  "validate": "(value:i)",
  "roles": ["guest", "all"],
  "result": {
    "type": "object",
    "properties": {
```

```

        "result": {"type": "number", "description": "The resulting number"},
        "status": {"enum": ["ERROR", "OK"]}
    }
}

```

In the example above, the application will listen for a **POST** request to **/maths/sqrt**, and call the PostgreSQL function **maths_sqrt**, sending the request's body as a JSON parameter.

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 can live in multiple directories. The list of directories it is found in, is defined in the configuration option "**api_directories**". This option is 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 the HTTP Listener have been created, each of the directories found in this option is traversed, and the API calls found in the files are created.

1.4.2 Fields of JSON Schema API call

The following fields are recognized in a schema object:

```

{
  "description": "API call description",
  "url": "URL of the call",
  "name": "CamelCase name for this call",
  "sqlfunc": "SQL function",
  "sqlfunc_type": "Specify jsonb here for JSONB functions. Defaults to JSON",
  "jsfile": "JavaScript file to execute"
  "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 to generate documentation},
  "validate": "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,
  "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"
}
}

```

1.4.3 Input 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 datetime

Field modifiers:

- * optional is true (default = false)
- **E** Empty becomes null
- **0** Nullable

1.4.3.1 Examples

The validation string `(batch_id: i, products: [s], qty: f*, labreport: (description:s))` means the object being validated must conform to the following rules:

- It must have a **batch_id** field with an integer value
- It must have a **products** field, which is an array of strings
- It can optionally have a **qty** field, of type float
- **labreport** is a nested object, with one field, **description**, which must be a string

Note! More examples are available in the grape repository, in the [examples/](#) subdirectory

1.4.4 Database API Calls

If an API call definition contains a **sqlfunc** field, it is named a "Database API call". The PostgreSQL function named in the **sqlfunc** field will be called when the API call is executed.

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 or JSONB object. **sqlfunc**type should be set to "JSONB" if the function accepts a JSONB parameter.

The object that is sent to the SQL function is created from the request's body (in the case of POST), or the request's query parameters (in the case of GET calls).

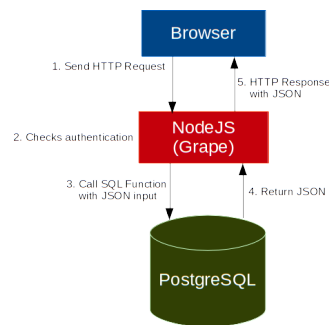


Fig. 2: Anatomy of a DB API

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;

```

1.4.5 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 "jsfile" field in the JSON schema. The JavaScript file path should be specified relative to the JSON file path.

The JavaScript file should return a function, for example:

```

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

```

The equivalent maths_sqrt function, implemented in JavaScript instead of SQL, will look like this:

```

module.exports = function() {
    return function(req, res) {
        var v = parseFloat(req.body.value);
        res.json({status: 'OK', result: Maths.sqrt()}).end();
    }
}

```

When writing API calls in JavaScript, you have access to the following fields:

- **req.app** - the grape express application

- **res.locals.db** - database connection
- **res.locals.session** - session information. If the a valid session is available, `res.locals.session.session_id` will be set
- **res.locals.session.user_roles** - list of roles associated with the current session

1.4.6 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 valid 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 sets of access roles for the request to be allowed. If no overlap exists, the request is denied.

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**.

See [Authentication](#) for more information on how to get a Session ID.

1.4.7 Generating API 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)
```

1.4.8 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.

1.4.8.1 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 Authentication

Before some resources in the system can be used, a session ID needs to be generated and sent with HTTP requests. The following steps are used to obtain a session ID for a user. This login procedure was designed to prevent any passwords from being sent over the network.

1. Firstly, the client (browser) retrieves the service name of the server we are trying to log into. [GET /grape/setting/service_name]
2. It then sends a [TGT \(Ticket Granting Ticket\) Request](#)
3. If the reply is an ERROR with code -400, the user is prompted for a TOTP (One time pin). The TGT Request is resent with the field `totp_status` set to 'y'
4. A decryption key is generated from the user's password, and the parameters received in the TGT Reply (See [Generating encryption and decryption key](#))
5. The message is then decrypted, and the TGT extracted
6. A service ticket is requested, using the TGT, service name and username (**POST /grape/auth/service_request**)
7. The session ID is obtained by sending the service ticket to **POST /grape/login_with_ticket**

1.5.1 Generating user's encryption and decryption key from password

The format of the field `grape.user.password` is built up as follows: `ALGO-ROUNDS:SALT:KEY`

- **ALGO** defaults to **sha256**. This is controlled by the grape setting [password_hashing_rounds](#)
- **ROUNDS** defaults to **1000**
- **SALT** is a randomly generated character string (20 characters)

The key is calculated by repeating the following process **ROUNDS** times:

1. Concatenate the result of the previous run (use an empty string on the first iteration), with the salt and user's real password
2. Calculate the SHA-256 (or **ALGO**) digest for the answer of (1)
3. Hex-encode the result from (2)

If a TOTP is used, the key hex-encoded SHA256 digest of the previously calculated key concatenated with the TOTP is used (`HEX(SHA256(KEY + OTP))`). Messages are then encrypted and decrypted using AES-CBC with PKCS padding.

1.5.2 TGT Request

This call is used to request a TGT (Ticket-granting ticket) for a user. The Encrypted TGT are extracted from the reply of this call, to be used in the issuing of service tickets

URL: **/grape/auth/tgt**

Method: **POST**

Fields:

- **username**
- **email**
- **totp** If the user was asked for an OTP, this field should be set to 'y'. If it is not, and the account requires a TOTP, the server will respond with error code -400

Note! Never send the actual OTP or password in any request!

Error codes:

- **-2** NO SUCH USER
- **-3** USER IS CURRENTLY DISABLED
- **-4** PASSWORD IS NOT IN VALID FORMAT
- **-98** USER DOES NOT HAVE VALID GUID

- **-400** USER HAS TOTP ON ACCOUNT BUT TOTP INDICATOR IS NOT SET
- **-500** USER EXISTS ON ANOTHER SERVER

The TGT is encrypted using the server's own private key and can never be read by the user. The Encrypted TGT is added to the reply message, but the message itself (contained in the "data" field) is encrypted using the key generated from the password entered by the user, and the parameters received as part of the reply of this request (SALT, ROUNDS, ALGO).

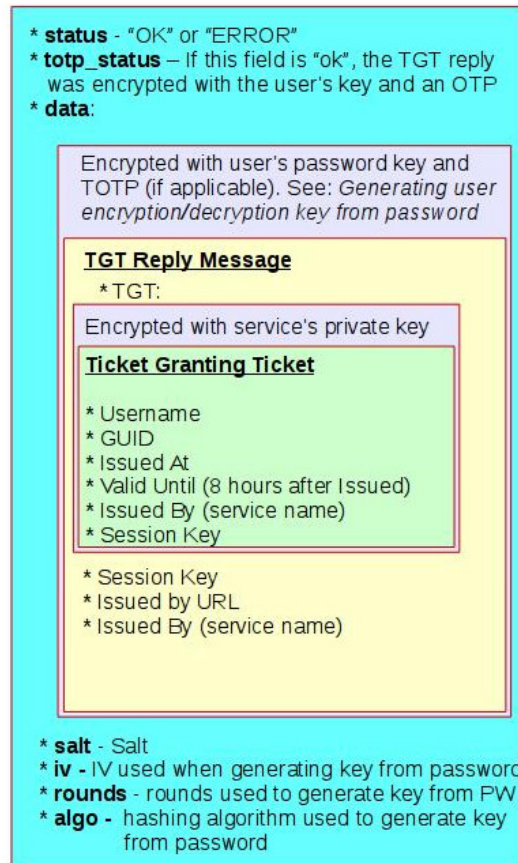


Fig. 3: Reply from POST /grape/auth/tgt

1.5.3 User auth_info field

Fields used:

- totp_status: "pending verification", "ok",
- totp_key
- password_reset_code

1.5.4 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.

1.5.5 Adding TLS/SSL support

If the configuration option **use_https** is set to true, the process will attempt to read the files **sslkey** (containing the path to the SSL key file) and **sslcert** (containing the path to the SSL certificate file). Certificates can be generated with the following command:

```
$ openssl req -x509 -nodes -newkey rsa:2048 -keyout private.pem -out public.pem
```

which will create two files, **private.pem** (used for **sslkey**) and **public.pem** (used for **sslcert**).

1.5.6 Configuration options

The following configuration options are specific to the HTTP Listener Worker:

NAME	DESCRIPTION	DEFAULT VALUE
api_directories	Array of directories containing API files	
cache_public_js_dirs	If enabled, the compiled JS files from public directories will not be cached	
compile_js_dirs	List of directory names that will be recursed when all JS is being compiled	['pages']
delayed_response	Delay all responses with this many milliseconds (this is used to simulate slow networks)	0
enable_notifications	Enables the notification subsystem	
http_port	If this is set, and HTTPS is enabled (use_https), then a normal HTTP server will listen on this port	
instances	Controls the number of HTTP listener applications started	1
maxsockets	Controls the maximum number of sockets supported per HTTP listener process	500
port	Port on which the UI will be available	
public_directories	Array with a list of directories containing public files	
use_https	Enable or disable HTTPS. sslkey and sslcert need to be set up correctly	false
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'

1.6 User Management

Users and user-related information is stored in the table **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**. To see a list of SQL functions that interacts with this tables, jump to [User related functions](#).

1.6.1 Access Roles

Grape comes built-in with the following assignable access roles:

- admin
- pg_stat
- switch_user

Additionally:

- If a user is logged in, they will always belong to the role "all".

- If a user is not authenticated, they will belong to the role "guest".

1.6.2 Creating initial users

TODO

1.7 Usage

After installing the ps-grape package, Grape can be started with the following code:

```
var Grape = require('ps-grape');
var app = new Grape.grape( 'config.json' );
app.start();
```

The constructor accepts any number of filenames and objects (`config.json` in the above example). Each argument will be read and parsed upon startup. See [Configuration Files](#) for more information on the structure of these files.

Typically, a project will have two configuration files:

1. A file for local configuration (options that is specific to a site/installation), usually named "config.json"
2. A file for project configuration. This file might also include references to other config files, for example when Grape modules are loaded. We usually call this file "defaults.json".

1.8 Grape configuration files

The Grape constructor accepts a list of filenames and objects. When the Grape object is instantiated, all of these files and objects are parsed and merged into a configuration object. This config object is then passed on to the rest of the application.

JSON config files can also have the special option "includes", an array containing paths to other config files to include. This is useful when using Grape modules. An application that creates Grape like this:

```
var Grape = require('ps-grape');
var app = new Grape.grape( 'config/system.json', 'config/locals.json' );
app.start();
```

will cause the application to attempt to read the files **config/system.json** and **config/locals.json**, in that order. One of the files might include all the system-specific settings, for example:

```
{
  "process_name": "demo",
  "port": 9000
}
```

and another, some local settings:

```
{
  "dburi": "pg://user@localhost/demo",
  "superdburi": "pg://user@localhost/postgres"
}
```

Note! Configuration files with sensitive or machine-specific information should never be checked into repositories!

For a reference of commonly-used configuration settings, see [Grape configuration reference](#)

1.9 Project file structure

Most Grape applications will have the following directories/files:

- *app/*
 - *api/* - API schemas to be loaded and registered by Node on startup
 - *index.js* - Application entry point
- *db/* - Database-related files
- *config/*
 - *defaults.json* - Grape config file containing product-specific settings
 - *local.json* - Grape config file containing local-specific
- *public/* - Public (HTML, Frontend JavaScript and CSS files). See Grape Frontend Development for more information regarding the contents of this directory
- *scripts/* - Scripts
- *config.json.example* - Example Grape config file
- *log/* - Log files (should not be checked into the repo)

1.10 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.

1.10.1 Defining the database structure

1.10.1.1 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.

1.10.1.2 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*.

1.10.1.3 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.

1.10.1.4 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).

1.10.1.5 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*

1.10.1.6 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.

1.10.1.7 Schemas

By default, grape applications will use the following schemas:

- public
- grape
- proc

1.10.2 Using NOTIFY/LISTEN

Grape includes a utility function, *new_notify_handler(channel_name, callback)*, which will keep track of callbacks for database NOTIFY events. This should be used in a custom background worker.

Note! When multiple instances of workers have registered a LISTENER, all of them will be invoked when a NOTIFY is called. Only add notification listeners in worker processes, not API call handlers

1.11 grape-db-setup

grape-db-setup is a tool used to load SQL files (from other SQL files, or from manifest files) into a database. This is typically used during initial creation of a database, or when applying patches to a database.

1.11.1 Installation

This tool will be install when grape is installed globally `npm install -g ps-grape`

1.11.2 Usage

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.
- **-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.
- **-a, --readconfig [config.json]** Reads the DBURI from the file provided (the file should have the "dburi" and "superdburi" fields).

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.

If no database connection is defined (through --dburi or --readconfig), the output will be printed to stdout.

1.11.3 File order

When loading files from directories, *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*.

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

1.11.4 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/
```

Manifest files can also be used to call a function with the contents of a JSON file from disk as input.

```
@calljson[b] [Function name] [Filename]
```

For example:

```
@calljson grape.user_save user.json
```

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.

1.12 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.

1.12.1 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

1.12.2 grape.pid

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

1.12.3 grape.fifo

This FIFO is used for communication between application processes.

1.13 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)

1.13.1 Known Grape settings

NAME	DESCRIPTION	DEFAULT VALUE
default_access_allowed	If a path is not found and this setting is true, access will be granted	false
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
password_hashing_rounds		1000
password_hashing_algo		sha256

1.14 Additional Functionality

1.14.1 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

1.14.1.1 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

1.14.1.2 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
    });
});
```

1.15 Additional Modules

Extended functionality is provided as part of Grape modules. A Grape Module can provide additional HTML files, API calls, database structures and functions, and workers. Grape modules are usually installed using npm.

1.15.1 Available Modules

- [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](#)

1.15.2 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

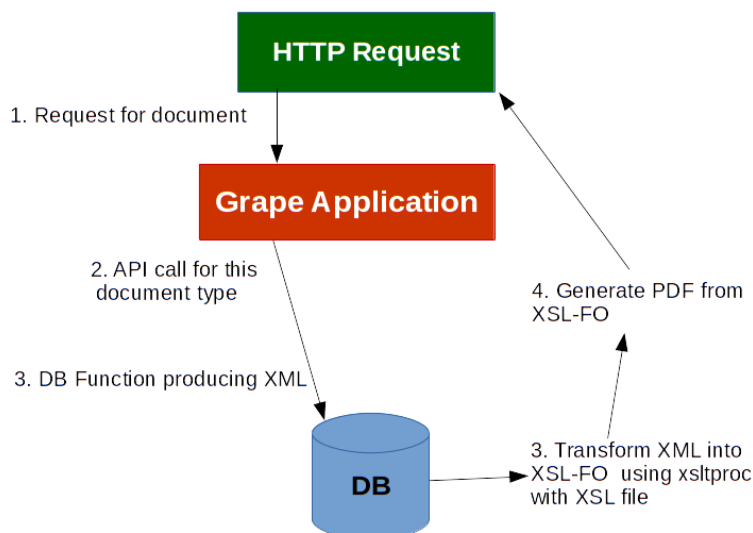


Fig. 5: PDF Generation Process

1.15.2.1 XML-producing SQL function

A function returning TEXT containing XML.

1.15.2.2 XSL file

1.15.2.3 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>

```

1.15.3 Sending Emails

1.15.3.1 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'`

1.15.3.2 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'}`

1.15.3.3 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.

1.15.3.4 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

1.15.3.5 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 %>`

1.15.3.6 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`. To 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);
```

1.15.4 Workers

Conforming to Node's design principle of single-thread multi-process, Grape spawns a number of processes when starting up. By default, 5 instances of the "HTTP Listener" worker is created. These workers implement an Express.js app, and handle incoming HTTP requests. It is possible to add custom workers, or workers defined in Grape modules.

1.15.4.1 Anatomy of a worker

1.15.4.2 Registering a worker

Before [Grape.start\(\)](#) is called, it should be registered using [Grape.addWorker\(options\)](#).

1.16 Reference

1.17 Grape class reference

Below follows a reference for the Grape object (the thing that gets returned from `new Grape.grape()`).

1.17.1 Methods

1.17.1.1 constructor (config, config, ...)

Reads and parses all arguments as configuration objects or files.

1.17.1.2 addWorker (options)

options should include:

- *name* The worker name. After the worker has started up, an event called 'worker-' + *name* will be emitted.
- *instance_count* The number of instances to start. Defaults to 1
- *func* The function to instantiate. Should have a "start" method

1.17.1.3 start ()

The start method verifies that all the correct config options are set, and then creates all the worker processes that has been registered. It also checks if the process PID file exists, and if it does, it sends a SIGINT signal to the process ID found in the file.

1.17.2 Members

1.17.3 Events

1.17.3.1 worker

1.17.3.2 worker-*

1.17.3.3 worker-*-created

1.17.3.4 creating_pidfile

1.17.3.5 pidfile_created

1.18 Standardized Error Codes

CODE	DESCRIPTION
-1	Unknown Error
-2	Permission Denied
-3	Invalid Input
-5	Requested data not found
-98	Configuration Error
-99	Database Error

1.19 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](#)

1.19.1 Adding new commands

`grape-manage` will scan `$CWD/cmd` and `$CMD/scripts/cmd/` for commands. Commands are JavaScript files that are required, 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;
```

1.19.2 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/`.

1.20 Built-in API calls

1.20.1 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

1.20.2 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

1.20.3 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)

1.20.4 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**

1.20.5 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

1.20.6 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**

1.20.7 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.

1.21 Grape SQL Functions

1.21.1 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 <i>reason</i> TEXT <i>data</i> JSON	Grape result types.
api_result_error	<i>message</i> TEXT <i>code</i> INTEGER <i>error</i> JSON	Returns a standardized JSON error object with status as "ERROR" and the other fields populated. Example: <pre>{"status": "ERROR", "message": "Message", "code": -2, "error": {}}</pre>
api_error	<i>message</i> TEXT <i>code</i> INTEGER <i>error</i> JSON	Overload for <i>api_result_error</i> .
api_error		With no arguments, an "Unknown error" message will be generated. Example: <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>

NAME	PARAMETERS	DESCRIPTION
api_error_invalid_data_state	info 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	keys TEXT[] values TEXT[] types 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 (n/i/number/integer, j/json or nothing for text).
api_success	keys TEXT values INTEGER	
api_success	key1 TEXT val1 INTEGER	Create an API result success JSON object with one integer field added.
api_success	key1 TEXT val1 INTEGER key2 TEXT val2 INTEGER	Create an API result success JSON object with two integer fields added.
api_success	key1 TEXT val1 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	fieldname TEXT data JSON	Returns success message when data is NOT NULL, otherwise it returns <code>grape.api_error_data_not_found()</code>
api_result	res <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>

1.21.2 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	processing_function TEXT short_description TEXT file_format_info TEXT function_schema TEXT param_definition 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: <code>{"NULL", "INTEGER", "NUMERIC", "DATE", "TIMESTAMP", "TIMESTAMPZ"}</code>
data_import_insert		API function to insert a data_import entry. 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. Required field data_import_id must be in the JSON data.

NAME	PARAMETERS	DESCRIPTION
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: 0 - Empty 1 - Populated 2 - Process started 3 - Some not processed 4 - Processed
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	<i>status</i> TEXT	Builds an object in the form of <code>{"result": {"status": "OK"}}</code> for returning from data import functions.
data_import_build_result	<i>status</i> TEXT <i>shared_data</i> 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	<i>status</i> TEXT <i>shared_data</i> JSONB	Builds a object in the form of <code>{"result": {"status": "OK"}, "shared_data": {}}</code> for returning from data import functions.
data_import_reset	<i>data_import_id</i> 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	<i>data_import</i> grape.data_import <i>args</i> 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.

1.21.3 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	<i>data</i> JSON	Converts JSON object to xml.

NAME	PARAMETERS	DESCRIPTION
	<i>root</i> TEXT	
json_diff	<i>old</i> JSONB <i>new</i> JSONB	Compares two JSON objects and returns an object containing fields that are different between the two objects. If a field exists in j_old , but not in j_new , it is not included in the results. If a field exists in j_new , but not in j_old , it is included in the results. If a field is different, j_new is chosen.
json_diff	<i>old</i> JSON <i>new</i> JSON	
json_object_diff	<i>old</i> JSONB <i>new</i> JSONB	Compares two JSON objects and return any values that exists in _new but not in _old .
json_array_diff	<i>old</i> JSONB <i>new</i> JSONB	Compare two JSON arrays and return any values that exists in _new but not in _old .
json_to_composite_type_text	<i>target_schema</i> TEXT <i>target_type</i> TEXT <i>data</i> JSON	Converts JSON object to composite type text.
json_to_composite_type	<i>target_schema</i> TEXT <i>target_type</i> TEXT <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 INT[] (cast_json_array_to_int_array.sql). <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 TEXT[] (cast_json_array_to_text_array.sql). <pre>#select cast_json_array_to_text_array(' [aa,bb,cc] '::JSON); cast_json_array_to_text_array ----- {"aa", "bb", "cc"}</pre>

1.21.4 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 <i>tables</i> TEXT[] - A list of table names to allow <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 _roles to be able to access the data in the table.
grape.list_query_whitelist_delete	<i>schema</i> TEXT <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 <i>tablename</i> TEXT	Check permission on a table for current user.

1.21.5 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)

1.21.6 Reports

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

NAME	PARAMETERS	DESCRIPTION
save_report	<i>report_id</i> INTEGER <i>name</i> TEXT <i>description</i> TEXT <i>function_schema</i> TEXT <i>function_name</i> TEXT <i>input_fields</i> JSON	
save_report	<i>name</i> TEXT <i>function_name</i> TEXT <i>description</i> TEXT <i>input_fields</i> JSON	
save_report	<i>settings</i> JSON	
execute_report	<i>report_id</i> INTEGER <i>parameters</i> JSON	

NAME	PARAMETERS	DESCRIPTION
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 <i>reports_executed_id</i> INTEGER <i>parameters</i> JSON	Function to convert a report to a file.

1.21.7 User related functions

NAME	PARAMETERS	DESCRIPTION
grape.user_save	JSON containing: <i>user_id</i> INTEGER <i>username</i> TEXT <i>password</i> TEXT <i>email</i> TEXT <i>fullnames</i> TEXT <i>active</i> BOOLEAN (optional) <i>role_names</i> TEXT[] <i>employee_guid</i> UUID	Save a user field, or create a new user. API call: POST /grape/user/save
grape.new_user	<i>user_id</i> INTEGER <i>rec</i> RECORD <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 <i>NULL</i> if it does not exist.
grape.user_id_from_name	<i>username</i> TEXT	Returns the user ID for a username, or <i>NULL</i> 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 <i>NULL</i> if it does not exist.
grape.username_from_fullnames	<i>fullnames</i> TEXT	Returns the username for a user found by fullnames, or <i>NULL</i> 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. 1. 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) 2. If grape.setting passwords_hashed isn't true, nothing is done (return -2) 3. On success 0 is returned
grape.hash_user_password	<i>username</i> TEXT	Overload for <i>grape.hash_user_password (user_id INTEGER)</i>
grape.set_user_password	<i>user_id</i> INTEGER <i>password</i> TEXT <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 <i>TRUE</i> .

1.21.8 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 <i>TRUE</i> if the current user belongs to <i>_role</i> .
grape.current_user_in_role	<i>roles</i> TEXT[]	Returns <i>TRUE</i> if the current user belongs to any of <i>_roles</i> .

NAME	PARAMETERS	DESCRIPTION
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 <i>check_path</i> TEXT - Access path to check <i>check_method</i> TEXT - HTTP method to check (GET/POST)	<p>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:</p> <ol style="list-style-type: none"> 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. 2. If the path has a role 'guest' granted access to it, everyone will be allowed (even if the session is invalid). 3. If the session is invalid, access will be denied and code 1 returned. 4. If the path has a role 'all', only, and all, valid sessions will be granted access. 5. If the user has access granted to the access path's role, access is granted. 6. If all the above fails, access is denied with code 2.
grape.session_insert	<i>user_id</i> INTEGER <i>ip_address</i> TEXT	<p>This function requires <i>user.sql</i>.</p> <ol style="list-style-type: none"> 1. Input: <ul style="list-style-type: none"> • username or email • password • ip_address • persistant TRUE/FALSE (optional) 2. Status: <ul style="list-style-type: none"> • status = ERROR • code 1 = No such user • code 2 = Wrong password • code 3 = User is inactive • code 4 = IP not allowed 3. On success, status = OK and following fields returned: <i>session_id</i>, <i>user_id</i>, <i>username</i> and <i>user_roles</i>. 4. Setting <i>hash_passwords</i> is used to decide if passwords are hashed or not.
grape.logout	JSON JSON containing session_id	<p>API call:</p> <p>/grape/logout</p>
grape.session_ping	JSON JSON containing session_id	<p>Checks validity of a session and returns a JSON object containing the session's username, user_id, fullnames, email, GUID and user_roles.</p> <p>API call:</p> <p>/grape/session_ping</p>
grape.session_insert	<i>username</i> TEXT <i>password</i> TEXT	<p>This function inserts a new session for a valid username and password provided.</p> <p>API call:</p>

NAME	PARAMETERS	DESCRIPTION
		/grape/logout
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	<i>user_id</i> INTEGER	Function which sets the <i>user_id</i> for the current session.
grape.set_session_username	<i>username</i> TEXT	Function which sets the <i>user_name</i> for the current session.

1.21.9 Other utility functions

NAME	PARAMETERS	DESCRIPTION
month_diff	<i>_d1</i> DATE <i>_d2</i> 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	<i>_name</i> TEXT <i>_value</i> TEXT	Sets the value (insert if new, replace if exist) in the key-value pair table <code>grape.setting</code> returning <i>_value</i> .
get_value	<i>_name</i> TEXT <i>_default_value</i> TEXT	Gets the value for setting <i>_name</i> , and if not found it will return <i>_default_value</i> . Defined in setting.sql
generate_uuid		Generates a unique UUID (for example b1086d35-e973-4356-3adc-2eeb6f4963e2). Defined in uuid.sql
array_lowercase	TEXT[]	
clean_telephone_number	<i>_tel</i> TEXT	Returns a text containing only numbers.
random_string	<i>length</i> INTEGER	Generates a random string of <i>length</i> length. Defined in random_string.sql

1.22 Grape SQL Views

1.22.1 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.

1.22.2 Session Related Views

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

1.22.3 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.

1.23 Grape configuration reference

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)	
debug		true
maxsockets	Controls the maximum number of sockets supported per HTTP listener process	500
pg_temp_directory	Path to a directory to which both PostgreSQL and the running node process has write access	
bordeaux_config_file	Path to Bordeaux config file	__dirname + '/bordeaux_config.json'
document_store	Path to document store	
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
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'
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	
process_name	Sets the visible process name	
enable_ldap_server	Enable LDAP server	
ldap_server_port	LDAP server port to listen on	