PLATINUMSOFTWARE

PlatSoft Grape Manual

Developer's guide to using the PlatSoft Grape Framework

Table of Contents

1. Getting Started Page 4
   1.1. Structure of a Grape Application Page 4
   1.1.1. Directory layout Page 4
   1.2. Authentication and access control Page 4
   1.3. API Calls Page 5
   1.3.1. Consuming an API call Page 5
   1.3.2. Creating an API call Page 5
2. How to send emails from Grape in PostgreSQL Page 5
   2.1. Setup Page 5
   2.2. Send an email Page 6
   2.3. Email templates Page 6
   2.4. Example Page 7
3. Known Grape settings Page 7
4. Grape config file Page 8
5. Standardized Error Codes Page 9
6. Grape SQL Functions Page 9
   6.1. API result functions Page 9
   6.1.1. api_success(_keys TEXT[], _values TEXT[], _types TEXT[]) Page 9
   6.1.2. api_success(_keys TEXT[], _values INTEGER[]) Page 9
   6.1.3. api_success(_key1 TEXT, _val1 INTEGER) Page 9
   6.1.4. api_success(_key1 TEXT, _val1 INTEGER, _key2 TEXT, _val2 INTEGER) Page 10
   6.1.5. api_success(_key1 TEXT, _val1 JSON) Page 10
   6.1.6. api_success() Page 10
   6.2. Data importing functions Page 10
   6.3. JSON helpers Page 10
   6.3.1. json2xml(_data JSON, _root TEXT) Page 10
   6.3.2. json_diff (_old JSONB, _new JSONB) Page 10
   6.3.3. json_diff (_old JSON, _new JSON) Page 10
   6.3.4. json_object_diff (_old JSONB, _new JSONB) Page 10
   6.3.5. json_array_diff (_old JSONB, _new JSONB) Page 10
   6.3.6. json_to_composite_type_text(target_schema TEXT, target_type TEXT, data JSON) Page 10
   6.3.7. json_to_composite_type(target_schema TEXT, target_type TEXT, data JSON) Page 10
   6.3.8. cast_json_array_to_int_array (JSON) Page 10
   6.3.9. cast_json_array_to_text_array (JSON) Page 11
   6.4. List query Page 11
   6.4.1. list_query(JSON) Page 11
   6.4.2. list_query_whitelist_add(_schema text, _tables TEXT[], _roles TEXT[]) Page 11
   6.4.3. list_query_whitelist_delete(_schema TEXT, _tablename TEXT) Page 11
   6.5. Process Page 11
   6.5.1. start_process (_process_id INTEGER, _param JSON) Page 11
   6.5.2. start_process (_process_name TEXT, _param JSON) Page 11
   6.5.3. start_process (JSON) Page 11
   6.5.4. list_processes (JSON) Page 12
   6.5.5. schedule_info (JSON) Page 12
   6.6. Reports Page 12
   6.6.1. save_report ( ) Page 12

6.6.2. save_report ( ) Page 12
6.6.3. save_report (JSON) Page 12
6.6.4. execute_report (_report_id INTEGER, _parameters JSON) Page 12
6.6.5. execute_report (JSON) Page 12
6.7. User and session related functions Page 12
6.7.1. toggle_user (JSON) Page 12
6.7.2. user_save (JSON) Page 12
6.7.3. user_save_password (JSON) Page 12
6.7.4. username (_user_id INTEGER) Page 12
6.7.5. user_id_from_name (_username TEXT) Page 12
6.7.6. user_id_from_fullnames(_fullnames TEXT) Page 12
6.7.7. username_from_fullnames(_fullnames TEXT) Page 12
6.7.8. hash_user_password (_user_id INTEGER) Page 12
6.7.9. current_user_roles() Page 13
6.7.10. hash_user_password (_username TEXT) Page 13
6.7.11. current_user_id() Page 13
6.7.12. check_session_access (_session_id TEXT, _check_path TEXT, _check_method TEXT) Page 13
6.7.13. set_session_user_id (JSON) Page 13
6.7.14. session_insert (JSON) Page 13
6.7.15. logout (JSON) Page 13
6.8. Other utility functions Page 13

1. GETTING STARTED

1.1 Structure of a Grape Application

1.1.1 Directory layout

The layout of a typical project is as follows:

- *api/* - API routes to be loaded and registered by Node on startup
- *db/* - Database-related files (mostly SQL)
  - *schema/* - The pgModeler dbm file, and exported SQL
  - *function/* - Files containing general stored procedures
  - *process/* - Files containing stored procedures 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
- *public/* - Public (HTML, Frontend JavaScript and CSS files). See Grape Frontend Development for more information regarding the contents of this directory
- *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
- *default_config.js* - Grape config file containing product-specific settings
- *email_templates* - Email templates

1.2 Authentication and access control

Grape manages authentication, sessions, users, passwords and provide access control. Sessions are tracked using a session ID. Session IDs can be 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 is sent to the server using a custom header ***X-SessionID*** in the HTTP requests. Before an API call is executed, Grape will check the validity of the session ID, and do access control on the path against the user's roles.

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	• /grape/login
all	All logged in users	• /lookup/* • /grape/list • /grape/api_list
admin	Administrator	• * (all paths allowed)

1.3 API Calls

1.3.1 Consuming an API call

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**

1.3.2 Creating an API call

A typical Grape API call consists of:

1. A call to register the route in the Node layer
2. A call to connect this route to a function in PostgreSQL
3. A PostgreSQL function, taking in one JSON value and returning a JSON object
4. A PostgreSQL function containing the business logic for the call
5. If applicable, this call should be added to Grape's list of allowed routes

For example, let's create an API call to calculate the square root of a value in the database:

1. When Grape starts up, it reads a directory containing API files. This directory is controlled by setting the ***api_directory*** option.
2. In this directory, add a file to register this API call in.

```
exports = module.exports = function(app) {
    // register the route
    app.get("/maths/sqrt/:input", 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
        {input: req.params.input}, // JSON object as input for this function
        null, // Optional callback (not used here)
        {response: res} // Send the response to res
    );
}
```

2. HOW TO SEND EMAILS FROM GRAPE IN POSTGRESQL

2.1 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'*

2.2 Send an email

Call *grape.send_email (to TEXT, template TEXT, data 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

2.3 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 accessable inside of the templates using `<%= firstname %>`

2.4 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);
```

3. KNOWN GRAPE SETTINGS

NAME	DESCRIPTION	DEFAULT VALUE
hash_passwords	Indicate whether passwords in grape.user is hashed	false
allow_default_paths	If a path is not found and this setting is true, access will be granted	false
grape_version	Current Grape version	
product_name	Name of the current system	
product_uuid	Unique identifier for the product/system	
product_version	Product version	
data_upload_schema	Default schema for data import tables	grape
disable_passwords	If true, authentication will not check whether the password is correct	false
system_url	URL to access system's frontend	

4. 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	
api_directory		
db_definition	Array containing directories with DB definitions. Subdirectories schema, function, view and data will be traversed when recreating	
sql_dirs	Array containing directories with DB definitions. All subdirectories will be recursively read	
pg_temp_directory	Path to a directory to which PostgreSQL has write access	
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 wil 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
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
use_https		
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'

NAME	DESCRIPTION	DEFAULT VALUE
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'

5. STANDARDIZED ERROR CODES

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

6. GRAPE SQL FUNCTIONS

6.1 API result functions

This functions deal 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
api_result_error	*message* TEXT *code* INTEGER *info* JSON	Returns a standardized JSON error object with stats as "ERROR" and the other fields populated. Example: `{"status":"ERROR", "message":"Message", "code": -2, "error": {} }`
api_error	*message* TEXT *code* INTEGER *info* JSON	Overload for api_result_error
api_error		With no arguments, an "Unknown error" message will be generated
api_error_invalid_input		Similar to calling `api_result_error("Invalid input", -2)`

6.1.1 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).

6.1.2 api_success(_keys TEXT[], _values INTEGER[])

Create an API result success JSON object with a list of integer fields as return.

6.1.3 api_success(_key1 TEXT, _val1 INTEGER)

Create an API result success JSON object with one integer field added.

6.1.4 api_success(_key1 TEXT, _val1 INTEGER, _key2 TEXT, _val2 INTEGER)

Create an API result success JSON object with two integer fields added.

6.1.5 api_success(_key1 TEXT, _val1 JSON)

Create an API result success JSON object with a JSON field merged into the result.

6.1.6 api_success()

Returns a API result object with a status field set to "OK".

6.2 Data importing functions

NAME	PARAMETERS	DESCRIPTION
data_import_insert		
data_upload_done		
data_import_row_insert		

6.3 JSON helpers

6.3.1 json2xml(_data JSON, _root TEXT)

Filename: json2xml.sql

6.3.2 json_diff (_old JSONB, _new JSONB)

Filename: json_diff.sql

6.3.3 json_diff (_old JSON, _new JSON)

Filename: json_diff.sql

6.3.4 json_object_diff (_old JSONB, _new JSONB)

Filename: json_diff.sql

6.3.5 json_array_diff (_old JSONB, _new JSONB)

Filename: json_diff.sql

6.3.6 json_to_composite_type_text(target_schema TEXT, target_type TEXT, data JSON)

6.3.7 json_to_composite_type(target_schema TEXT, target_type TEXT, data JSON)

This function will populate a custom type from a JSON object. What sets it apart from the functions available in PostgreSQL, is the fact that it supports complicated multi-level nested objects. **Filename:** json_to_composite_type_text.sql

6.3.8 cast_json_array_to_int_array (JSON)

Provides an implicit cast from JSON to INT[] (cast_json_array_to_int_array.sql).

```
# select cast_json_array_to_int_array('[1,2,3]'::JSON);
 cast_json_array_to_int_array
 ------------------------------
```

```
{1,2,3}
```

6.3.9 cast_json_array_to_text_array (JSON)

Provides an implicit cast from JSON to TEXT[] (cast_json_array_to_text_array.sql).

6.4 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

6.4.1 list_query(JSON)

This function returns row from a database table. The following input fields are recognized:

- tablename
- 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 '=', '#x003E;', '#x003C;', '#x003E;=', '#x003C;=', 'LIKE', 'ILIKE', 'IS_NULL', 'IS_NOT_NULL', 'IN'
- value text

6.4.2 list_query_whitelist_add(_schema text, _tables TEXT[], _roles TEXT[])

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

6.4.3 list_query_whitelist_delete(_schema TEXT, _tablename TEXT)

Removes a table from the whitelist.

6.5 Process

This functions deal with tables that are used by ps_bgworker. ps_bgworker provides a way for systems to run tasks and processes in the background and at specific times. This functions can be found in process.sql.

6.5.1 start_process (_process_id INTEGER, _param JSON)

6.5.2 start_process (_process_name TEXT, _param JSON)

Similar to *start_process (_process_id INTEGER, _param JSON)*, but taking a process name as first parameter.

6.5.3 start_process (JSON)

API access function to *start_process (_process_id INTEGER, _param JSON)*

6.5.4 list_processes (JSON)

6.5.5 schedule_info (JSON)

6.6 Reports

This functions can be found in reports.sql.

6.6.1 save_report ( )

6.6.2 save_report ( )

6.6.3 save_report (JSON)

6.6.4 execute_report (_report_id INTEGER, _parameters JSON)

6.6.5 execute_report (JSON)

6.7 User and session related functions

6.7.1 toggle_user (JSON)

Filename: user.sql

6.7.2 user_save (JSON)

Save user field. Also used to add a new user to the system. **API Call:** POST /grape/user/save
Filename: user.sql

6.7.3 user_save_password (JSON)

Filename: user.sql

6.7.4 username (_user_id INTEGER)

Returns a TEXT field containing the username matching the user ID provided
Filename: user.sql

6.7.5 user_id_from_name (_username TEXT)

Returns the user ID (or NULL if not found) for the user matching the username provided.
Filename: user.sql

6.7.6 user_id_from_fullnames(_fullnames TEXT)

Returns an integer containing the user ID matching the full names provided
Filename: user.sql

6.7.7 username_from_fullnames(_fullnames TEXT)

Returns the username for the user matching the fullnames provided
Filename: user.sql

6.7.8 hash_user_password (_user_id INTEGER)

Hashes a password for user and updates the user table afterwards.

- 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 (-1 is returned)

- If grape.setting passwords_hashed isn't true, nothing is done (return -2)
- On success 0 is returned

6.7.9 current_user_roles()

Returns a list of all roles the current user belongs to **Filename:** user.sql
Filename: user.sql

6.7.10 hash_user_password (_username TEXT)

Overload function for ***hash_user_password (_user_id INTEGER)***, taking a username instead of a user ID as input.
Filename: user.sql

6.7.11 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.
Filename: current_user_id.sql

6.7.12 check_session_access (_session_id TEXT, _check_path TEXT, _check_method TEXT)

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:

1. Check that the path has access control on it. If it cannot be found, the grape setting ***default_access_allowed*** 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

Filename: session_access_path.sql

6.7.13 set_session_user_id (JSON)

Sets the current database session's ***grape.user_id*** config variable for use in stored procedures to identify the current user.
Filename: session_access_path.sql
Returns: JSON

6.7.14 session_insert (JSON)

This function inserts a new session for a valid username and password provided.
Filename: session.sql

6.7.15 logout (JSON)

Filename: session.sql

6.8 Other utility functions

NAME	PARAMETERS	DESCRIPTION
month_diff	*_d1* DATE *_d2* 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	*_name* TEXT *_value* TEXT	Sets the value (insert if new, replace if exist) in the key-value pair table grape.setting returning _value.

NAME	PARAMETERS	DESCRIPTION
get_value	***_name*** TEXT ***_default_value*** TEXT	Gets the value for setting ***_name***, and if not found it will return ***_default_value***. 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	***_tel*** TEXT	
random_string	***length*** INTEGER	Generates a random string of ***length*** length. Defined in random_string.sql