Gravity Development Guidelines

This is a document meant to educate and aid developers on the use of the Gravity development framework, providing guidelines on how to customize and further develop it. This guide will be updated as more features are added.

Basic info, Stack description

What is Gravity?

Gravity is a blockchain powered full-stack web development framework that allows for the quick development of apps whose information can be stored and retrieved into the Jupiter blockchain maintained by Sigwo Technologies. This allows an effective means of encryption and backup for sensitive information.

The block under which Gravity is built on is the Jupiter Blockchain, a fork of the public NXT blockchain. Because of this, all nxt blockchain api calls work and perform the exact same functions in Jupiter. The NXT api guide (https://nxtwiki.org/wiki/The_Nxt_API) is thus a great resource to use when looking to develop new features in Gravity that depend on interactions with the Blockchain.

Stack and required plugin information

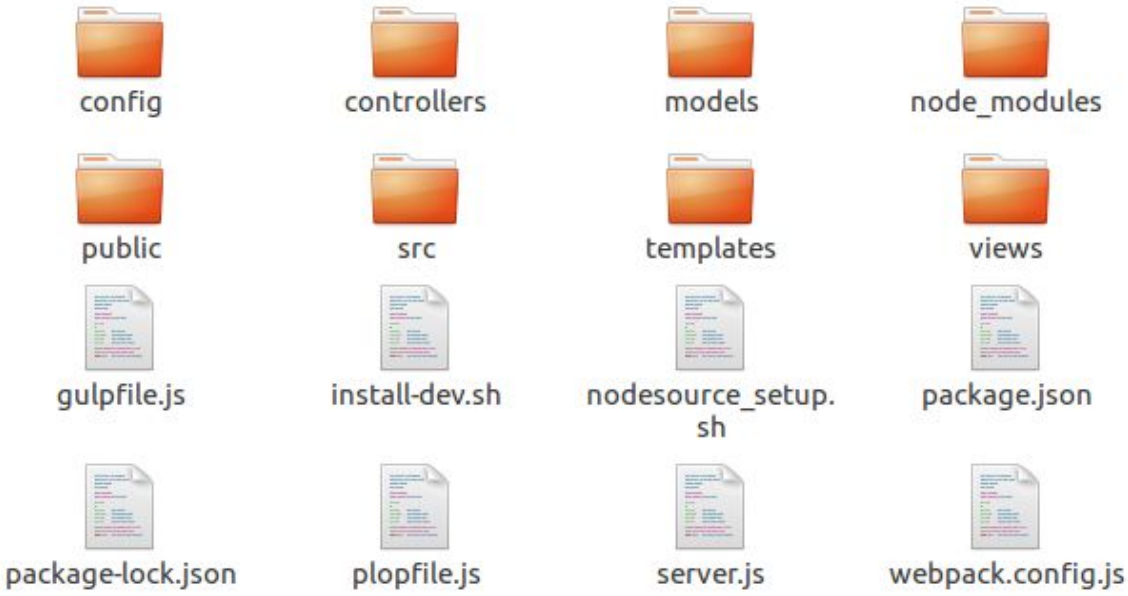
NodeJs was the framework chosen to write Gravity's code and manage its backend. In order to power Gravity, a number of packages are required to run the app. The list of packages can be seen under the 'packages.json' file in the root directory. The main mentions of this packages are the 'Express' and 'React' packages. The first package is a backend framework that expands on Node's features and is heavily used in all backend related code. The second package is a robust front-end framework that allows for faster development of better user experiences. The entirety of the Gravity front-end is being written as React Components.

In addition to React as the main front-end development tool, Bootstrap is used as a styling framework to quickly develop mobile-compatible pages.

To facilitate the quick development of the app, Gravity makes heavy use of both the Plop and Gulp node libraries, which specialize in generating templates and running tasks.

File arrangements

Below is a screenshot of the typical file arrangement of a Gravity powered app.



Architecturally, Gravity mainly follows a Model-view-controller (MVC) pattern with a clear separation between files that handle data mapping, routing and and front-end rendering (models, controllers, views/src folders respectively).

The Root Files

- **Server.js:** This is the most important file of the app as it glues all the packages and files together. It was setup in such a way that it runs the app based entirely on the contents of the folders detailed above. Because of this reason, it does not need to be modified unless a new package is loaded in the app and it needs to run on every folder.
- **Package.json:** As mentioned previously, this file contains all the app dependencies. When downloading the code for its repository, use the 'npm install' (or 'sudo npm install' if necessary) command to install all the dependencies of Gravity in your local environment. It will automatically use those in the package.json file to install them. Read more on npm to know how to add new packages (you don't need to write them on this file directly for them to be install, typing npm install 'name of package' and pressing enter in your terminal automatically adds the new package to the package.json file).
- **Package-lock.json:** File generated when running npm install. No need to touch this.
- **Webpack.config.js:** Webpack is used to compile the React Components of Gravity into a specific file ('bundle.js'). What you need to know is that the React Components that you write won't actually be loaded by the app when a page is loaded, instead, 'bundle.js' will be called and provide the webpage with the behaviour described in the components. This file basically tells Webpack to load the React Components (more on them on later

file descriptions) into bundle.js. This file is absolutely necessary at the moment but don't need to edit it unless planning on doing a front-end overhaul.

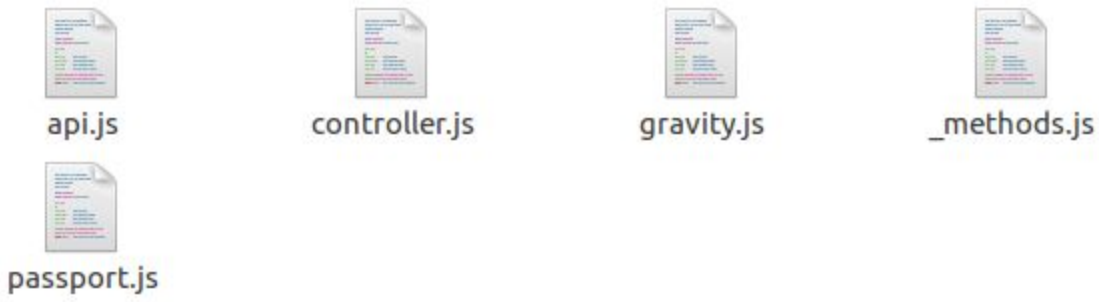
- **.env(Important for Development!):** The .env file is not present in the above screenshot but it is necessary when hosting gravity in your local computer for development purposes. You would just need to add it to the root file and name it .env (yes, with the period in the front). This file would hold the environmental variables needed by the Gravity app. These are variables with sensitive data that should not be stored in the code, but in the server on which the Gravity app runs.
- **.gravity.js(Important for Development):** The .gravity file is also not present in the above screenshot but it is necessary when developing your app. This file should contain the same variables as the .env file but on a Javascript format. This is so that you are able to run some of the commands needed to work on your applications (such as database table creation).
- **plopfile.js(Important for Development):** This is the file that handles file generation based on templates.

Node_modules

These folders is generated when loading packages for the app (via npm install or bower) and contains all the dependency files. **You should NOT modify these folders or any of its files in any manner. It contains none of the custom code used by Gravity.**

Config

This folder contains all the files related to the general configuration of the application and which are directly loaded in the server.js file. The following are the files that would always be in a Gravity app.

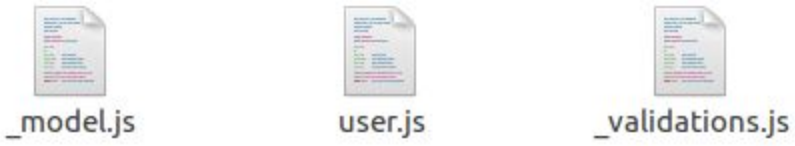


- **api.js:** This file contains the general api endpoints that can be used to retrieve data from the blockchain in a general manner. More on this in upcoming sections.
- **controller.js:** This file contains the routing rules that will be employed by the application. For example, they redirect the user 2FA screen when needed or prevent unlogged users to access pages that require authentication.

- **gravity.js:** This is an extremely **important file as it loads the app with all the functions the gravity site will use to record data into the blockchain as well as generating important files for development.**
- **_methods:** This file contains certain methods used by the configuration files and that are too large to be placed in the other files without sacrificing code legibility.
- **passport.js:** This file contains the code related to the user authentication rules that the app will use. The package Passport handles most of the heavy lifting in the authentication process. This file **contains all the logic involved in validating a user on login and storing user information at signup**. Api calls to Jupiter are included during signup so to automatically store certain information in the blockchain and perform certain actions related to it (generating an alias for the user in the blockchain for example).

Models

This folder contains the code related to the data that you will be storing in the blockchain. The types and number of files contained in this folder depend on the information that you will be storing, but each 'type' of record should be put in a separate file to keep information properly classified (More on model files in the 'How to create a record' section). There are 3 files that will be in this folder by default though:



- **_model.js:** This is the basic template that every model file is an extension of. **BE CAUTIOUS when if editing this file as it will affect data retrieval and organization.**
- **_validations.js:** This file is used to provide validation rules of the data entered to the blockchain. Because data validation cannot be done at this moment at the blockchain level, validations rules are fed to the _model.js file which **validates data before it is pushed to Jupiter**. Do not edit unless your project needs specific validation rules.
- **user.js:** This is the default user model file. When a 'users' table is created (more on this in later sections), **this file allows for an immediate way to record information from the users of the app such as name, lastname, email, JUP account and other information required such as keys for 2FA authentication.**

Public

This folder contains all the public assets that will be used and displayed in the Gravity app such as styling files, images, javascript files, and external code that you directly want to reference in your webpage. The following is a screenshot of what you typically would find inside this folder:

The most important things to remember is that this is that:

1. The contents in this folder can be directly referenced in your web page as if they were located in the root file. For example, if you want to reference an image in your page, you can do so by calling '/img/name-of-image-file.file-extension'.
2. The app uses scss for styling by default and gulp to update all files.
3. The js folder contains the bundle.js file, which is the compilation of all the code related to the front-end react components. **When making a webpage, '/js/bundle.js' is the link file you must add to load your react components.**

Src

This folders contains all the react-components code that the app will use and which will be compiled into the bundle.js file. **Inside of it you will find a folder called components which contains all the react components that app will use, including the ones that will retrieve your app data your users.** More on this in later sections.

Views

This is the folder that contains the jsx files that are directly involved with front-end rendering of the app. **It contains a Layout folder and a file for each web page that the app will render. The layout folder would have html code that is rendered in all web pages**, for example, a navigation menu and/or footer. These files also contain the html objects with the ids specified in the /src/index.jsx that will render the react components for the app.

Screenshot of the Public folder:

Development with Gravity

Required Environmental variables

A number of environmental variables will be required to run alongside your application in order to run your gravity application. This env variables can be generated via the **npm run gravity:db:create** command, which will create a .env file with the variables along with a .gravity.js file containing a copy of them as well in JSON format. This later file is used by Gravity to run certain commands while your app itself is not running. **The .env and .gravity.js file should be deleted when in production to increase security. In production, use different methods to upload your env variables such as storing them in your /etc/environment file.**

The environmental variables required to run a gravity app are the following (**Keep them all in uppercase**):

- APPNAME => This environmental variable can be used to display the name of it throughout your application
- JUPITERSERVER => This the url or api address of the **jupiter node** your app will use to push and retrieve data from the blockchain.
- APP_ACCOUNT_ADDRESS => This is a Jupiter address that will represent your app's data in the Jupiter blockchain. **This is the address where you will have to send your JUP to in order for your app to save data.**
- APP_ACCOUNT => This is the **passphrase** for the above mentioned account. The passphrase is needed for automation purposes.
- APP_PUBLIC_KEY => This is the **Jupiter public key** associated with the above account. Having makes data recording safer for your app.

- ENCRYPT_ALGORITHM => This represents the algorithm used to encrypt your app's data prior to saving it in the blockchain (2 layer encryption).
- ENCRYPT_PASSWORD => This represents used in the encryption process in the above mentioned scenario. **It is extremely important to keep this password secure as it cannot be changed.**
- SESSION_SECRET => This variable is used by Passport to generate sessions.

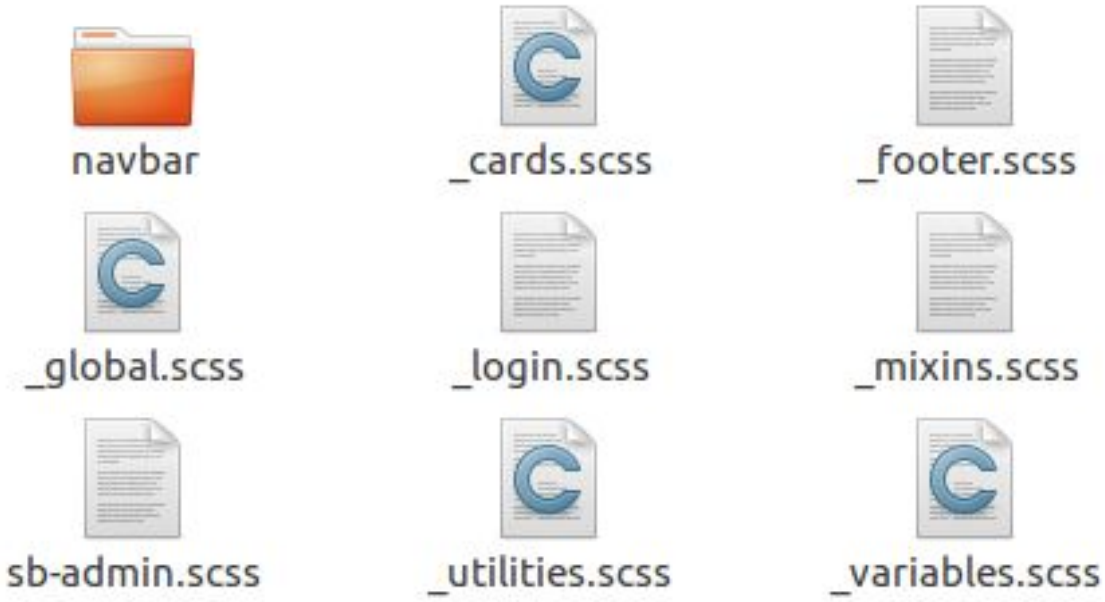
With this environmental variables properly secure, you can retrieve your app's data immediately from the blockchain at any point.

The Frontend

Customizing the Frontend Style

Gravity uses a combination of Gulp, jQuery and the latest version of Bootstrap for custom styles. Bootstrap is the frontend framework that lets us quickly and easily build a beautiful, responsive and interactive web-application. Using Gulp, we take advantage of Bootstrap 4's custom theming as well as the sb-admin theme provided by startbootstrap.com. The original template files have been included for your convenience. Gravity also uses jQuery and jQuery.easing for animation and dynamic website content.

The Bootstrap Theme can be customised using the SCSS files located inside the public folder. These SCSS files are a collection of components used to style the website. Here is a screenshot of what the SCSS folder contains.



Here you will notice the sb-admin.scss, this file is only meant to combine all of our various scss files into one main file to be served to the public. The navbar folder contains all the style components to customize the header of the page in Gravity. The first file you should take a look at is _variables.scss, this sets the colors used throughout the template, this means that we only have to change one file, and the colors will be updated across all files.

The next important file you should review is the _global.scss file as this contains all the global attributes like the body tag, the height and width calculations of each section, and the content-wrapper section, please note that the content-wrapper is where gravity's content will be served. This will be the body content of the page. Navbar is the top header with the links and navigation. The _login.scss is a custom style sheet for the login page. The _footer.scss controls the style for our footer section, and the _mixins.scss are custom variables we can define, and use in other files. The sb-admin.scss is the master file that combines everything into one file so the styles can be served to the website. The _utilities.scss file contains custom variables we can use for applying our own sizes and heights. Finally the _cards.scss file is the custom style for the bootstrap 4 "Cards"component.

Once we have made an alteration to these files, we must run 'gulp' inside the root directory of Gravity to re-compile our files and view our changes live.

In addition to the 'gulp' command, we can also use 'gulp dev' to tell Gulp to watch for any changes, then re-compile everything when the change is detected. It is recommended to run the 'gulp dev' command in a separate terminal screen as this will continue the process in the

background, then you can open another terminal screen to run ‘npm start’ to review your changes.

If you want to set your own custom styles for gravity, it is recommended to create a custom.css file and link it to gravity on the application.jsx file inside views/layout/ Then you can build your own custom css styles to load over-top of the theme files that are already established, this is recommended for quick changes, however for more style control, the SCSS files are the way to go.

We have also included all the original template examples, this way we can easily copy, and paste sections from the template that you wish to your on your site. The examples can be found on the landing page of gravity under the Getting Started tab.

The Backend

Data recording in Gravity

As of the writing of this guide, **information is recorded in the Jupiter blockchain as encrypted messages using the ‘/nxt?requestType=sendMessage’ api call to the blockchain node**. Rather than creating new systems inside Jupiter, we used Nxt’s existing infrastructure to record data. The only requirement for this method is that we have to change the file declaring the constants that the node will use and expand the data limit on message recording. For Gravity, the data limit was extended to 10 times its default.

One important concept for data recording in the blockchain is that an address needs to be created for each data model that needs to be recorded. For example, a specific user address will be used to record User information, another one for payout information, and so on and on. **Gravity records data by sending a JSON object from the model address to the address of the user the data belongs to.** This JSON object **is encrypted twice** first using an encryption method set by the gravity app (more on ’Model files’ and Required environmental variables’ sections), and then by the blockchain itself.

As an app developer, **you will need to assign your app a Jupiter account address. This address will act as the main database of your app and will store the address/passphrase/public key information of the tables holding data of your models**. More on this can be found in the ‘Gravity Commands’ section.

The _model.js file

The _model.js file is the main file responsible validating, sending, retrieving, processing, and updating data from the blockchain. Each new model file that is created is or should be an extension of the above file.

Below is a partial screenshot of the _model.js file as of the writing of this guide:

```
var axios = require('axios');
var validate = require('../models/_validations.js');
var gravity = require('../config/gravity.js');
var events = require('events');


class Model {
    constructor(data) {
        //Default values of model
        this.id = null;
        this.record = {};
        this.model = data.model;
        this.table = data.table;
        this.model_params = data.model_params;
        this.data = data.data;
        this.validation_rules = [];
        this.record = this.setRecord();
    }

    setRecord() {…
    }

    generateId(table) {…
    }


    verify() {…
    }
```

At the top you see that we are importing libraries that will be required in all of our models such as **axios** for making calls, **validations** (which provides data validation rules and methods), **gravity** (which provides the methods used to communicate with the blockchain) and **events** (to help divide our code into different steps).

We see that this file is creating a Model class that we export at the end of the code and that it requires **data** (our record's data object) which will come from the other models files such as User when they are called in other parts of the code. We also see that it defines object variables that will be used by the system, such as **model_params**, which lists all the parameters this model uses (ignoring any others that might try to be passed) and the **table** variable, which will contain the name of the table we are recording the model data to. There are several methods in this file, but the most important ones are the **verify()**, **create()** and **update()** methods which are directly used in the data recording process.

verify() is used to **validate the data you wish to record before submitting it to the blockchain** and it is always **called automatically inside the create() and update() methods**. If works by **reading** the **validation_rules** variables initialized in the constructor(and properly defined inside the other model files) and **comparing them with the values given to us in the data parameter**.

create() and update() are pretty self-explanatory **but it has to be indicated that update() works not by modifying the record created by create() but by copying the latest version of that record and updating its values before pushing it to the blockchain**. This is done because the data in the blockchain cannot be deleted if not set to prunable data, **so Gravity retrieves all versions of a record with a specific Id number and sorts from newest to oldest**. This way we can accurately 'update' records and always work with the latest version available.

create() creates the first instance of a specific record and pushes it to the Jupiter blockchain. The **generateId()** method is called inside to give the first record instance a unique id number. This id number if copied by the **update()** method which will push all newer versions of the original record.

Currently, there is no way to delete data. Future versions of gravity, however, will provide a way for records to be 'archived' when they are no longer needed by the app.

Edit this file only if you wish to add global model methods or modify existing methods. If is recommended, however, that you create new model (let's say

‘new_model.js’) that works as an extension of _model.js and make your other model files an extension of this one.

Model files and Data validation

Model files are files that represent what a single record in your database looks like and what types of data it holds. This files are extensions of the _model.js file we covered in the previous section and they connect to the Jupiter blockchain through that file. Below is a copy of what our user model (which comes in all versions of gravity) file looks like by default in gravity.

```
var Model = require('../models/_model.js');
var methods = require('../config/_methods.js');
var bcrypt = require('bcrypt-nodejs');

class User extends Model {
    constructor(data) {
        //Sets model name and table name
        super({
            model: 'user',
            table: 'users',
            model_params: ['id', 'account', 'accounthash', 'email', 'firstname', 'lastname',
                'secret_key', 'twofa_enabled', 'twofa_completed', 'api_key'
            ],
            data: data
        });
        this.public_key = data.public_key;

        //Mandatory method to be called after data
        this.record = this.setRecord();


        this.validation_rules = […
        ]
    }

    setRecord() {…
    }

    generateHash(accounthash) {…
    }

    validPassword(accounthash) {…
    }
    generateKey() {…
    }
}

module.exports = User;
```

As you can see, we load our _model.js file at the top. The **extends Model** part of the code next to the class name makes the file an extension of our main model file. The super()

method is called to pass data to our _model.js file to set variables for the object. **Every model file needs to assign 4 pieces of data through the super method: model, table, model_params and data.**

Model is simply the name of the model in form of a string; this is done to help customize error messages and other things with the actual name of the model.

Table represents the name of the table in the database (you app's address); s**ince all tables of your app are saved in your app address as part of a JSON object, gravity can locate your table by finding the key in the object that matches the string set in the table variable**.

Model_params tells gravity exactly what fields are meant to be saved as part of your record. **If you try to save an object that has a field not included** in the model_params list, it **this field will not be recorded**.

Data represents the actual the JSON object data you are trying to save and which will represent your record. **Data whose keys match parameters in the model_params list will be added to the this.record variable, which is a JSON object.**

this.validation_rules is an array that holds a list of objects used to make sure the data you want to save meet certain standard you want to keep. The following is an example:

```
{
    validate: this.record.firstname,
    attribute_name: 'firstname',
    rules: {
        required: true,
        dataType: 'String'
    }
},
```

The above indicates that we are providing a validation rule for the our record's firstname attribute. **Attribute_name** is a string representation of how you wish this attribute to be represented in validation messages and **rules** contains the actual validations rules. The screenshot shows are we our model's first name attribute is a String data type that is required in order to save the record. To provide further validation, you would need to assign values to other keys recognized by gravity (See 'Data validation keys' section to see supported validation rules).

Finally, **after setting all the initiation data for the model in the constructor, we write down all the specific class functions that the model is going to be using in your app**. Going back to the User model seen before, we see that, among other methods, there is a validPassword() function that user objects will use to validate passwords during login.

Data validation keys

The following are all the keys that gravity currently support:

- **required**: If set to true, this means the attribute needs to be entered by the user.
- **dataType**: This indicates what data type the attribute would fall under. Different validation rules would apply depending of the data type. Current types supported are : String, Integer, String, Boolean (attribute can only be true or false or else an error would be raised) and Email(verifies that the attribute value fits and email format). An error will be raised if an unknown dataType is entered.
- **minLength**: If the datatype of the attribute is String, this indicates the minimum length it should be if added to the requirements hash. An error will be raised if the value of minLength is not an integer or if it is higher than maxLength or if lower than zero..
- **maxLength**: If the datatype of the attribute is String, this indicates the maximum length it should be if added to the requirements hash. An error will be raised if the value of maxLength is not an integer or if lower than minLength or if lower than zero.
- **lessThan**: If the datatype of the attribute is Integer, this indicates the indicates the minimum value the attribute should be if added to the requirements hash. If added, then the value of the attribute will need to be one number higher than the moreThan parameter.
- **moreThan**: If the datatype of the attribute is Integer, this indicates the indicates the maximum value the attribute should be if added to the requirements hash. If added, then the value of the attribute will need to be one number lower than the lessThan parameter.

More **validation rules will be added as the Gravity project progresses**.

Model files are automatically generated through the 'run npm gravity:app:scaffold' command as detailed in the 'Gravity Commands' section but **they can be made manually by following the same structure detailed above.**

Authentication

Gravity natively uses the node package Passport to handle user authentications inside your gravity app. **The passport.js file inside the config folder contains your signup and login methods and has been customized to work alongside the Jupiter blockchain**.

Routing

Basic routes

Routing in Gravity is handled by the files located in the 'controllers' folder. **By default, two files will be included out of the box in gravity: _application.js and account.js.** The first

file contains all the general routes your gravity apps starts with for not authenticated users and the basic user authentication routes. The second file contains specific routes for accounts used by users and 2FA setup and authentication.

In addition to these two default routing files, the 'config' folder contains an additional once called **'api.js**'. This file **is extremely important as it provides generic routes that can be used to retrieve, create and update right away after calling for the 'npm run gravity:app:scaffold' command and creating the table in gravity**.

As mentioned at the beginning of this guide, gravity uses Express to handle requests and responses to the server. A very basic route would work as follows:

```
app.get('/test', function(req, res) {
    res.send({success: true});
});
```

The above route basically catches all requests made to 'your-site.com/test' and returns an object with an object with a key **success** with true **value**. The variable app is defined in server.js file and passed to the route file. This route of course will not return a website, just a response object. If you wish for your route to return a page, you would write the code as follows:

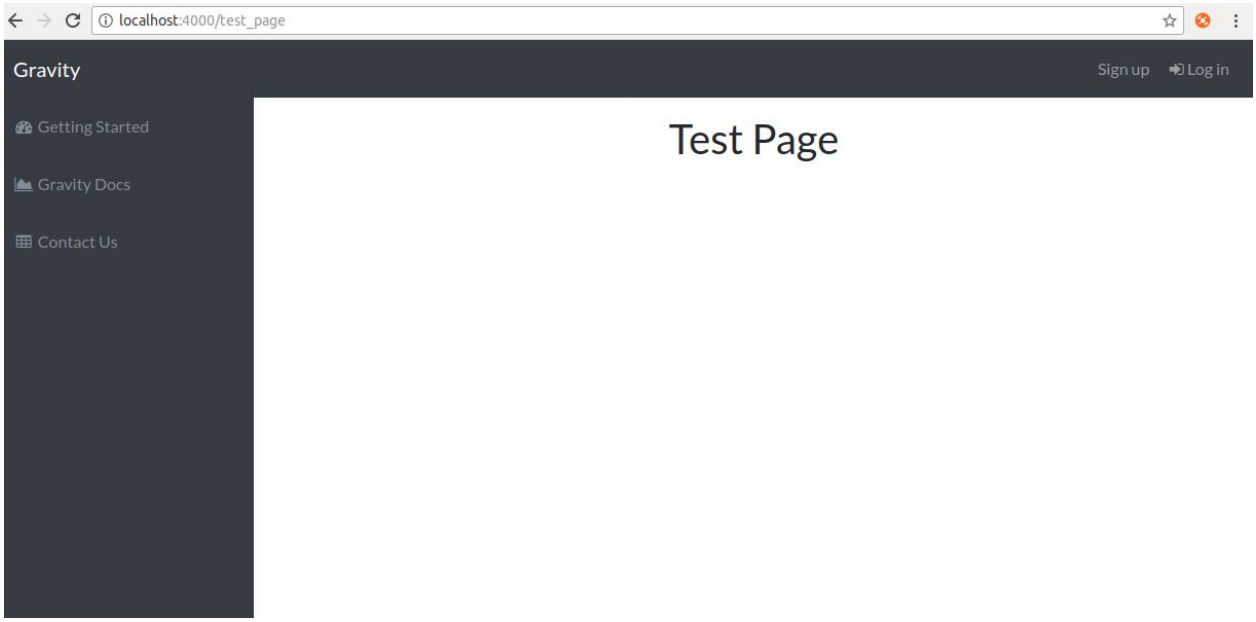
```
app.get('/test_page', (req, res) => {
    var messages = req.session.flash;
    req.session.flash = null;

    const PageFile = require('../views/test_page.jsx');


    var page = ReactDOMServer.renderToString(
        React.createElement(PageFile, { name: 'Gravity - Test Page', dashboard: false, messages: messages })
    );

    res.send(page);
});
```

The above code catches requests made to the '/test_page' endpoint. Once the request is captured, **we load the code that contains the view for that specific page**; in this case, we load test_page.jsx. As mentioned previously, React is gravity's frontend framework and every view file will have the jsx extension. **This code is not pure html code** however, so we use the ReactDOMServer library's **renderToString**() and the React's **createElement**() methods to convert our jsx file into a React element (assigning attached data to it) and then into a string which will be read as pure html code by the browser. We store this converted string into our page variable and send it as a response back to the client.



Route permissions

There are currently three types of non-api Get routes in Gravity: routes for pages that can be accessed by anybody(login, signup, and informational pages), routes for pages that only required you to be logged in (the 2fa page for example), routes for pages that will block the user from accessing them if additional information or clearance is required (for example, if 2fa code is required).This permissions can be found in the controller.js file in the config folder.

We'll first mention how to add **pages that can be accessed by everyone**, which will be referred to as **unlogged pages** for the remainder of the guide. We'll cover the other types of GET pages afterwards.

Unlogged pages and basic route structure

These are pages that can be accessed without authentication of any kind (about pages, signup, login, etc). Our previous screenshot of our test page code is the perfect example. Let's break down its components as it shares mostly the same structure as all page loaders in gravity:

```
app.get('/test_page', (req, res) => {
    var messages = req.session.flash;
    req.session.flash = null;

    const PageFile = require('../views/test_page.jsx');


    var page = ReactDOMServer.renderToString(
        React.createElement(PageFile, { name: 'Gravity - Test Page', dashboard: false, messages: messages })
    );

    res.send(page);
});
```

The first thing that we will always need to include is the 'var messages= req.session.flash' which creates a local variable that will store messages that were stored as a session variable in a previous route (this will occur if a different route redirects to the one you are creating). **After writing that line, you will need to clear the session messages** by declaring the flash session variable null to make sure this message won't be displayed after doing so the first time.

As mentioned before, we load our code for the page and return it as a string that read as html by the browser. The JSON object next to PageFile is the **data** that will be attached to this page. The **props that will always be included in every page route are**:

- **Name**: Which will be the text inside the <title></title> html tags for the page that you will be loading.
- **Dashboard**: This value lets the system know if the page about to be rendered requires the user dashboards or info to be displayed automatically(Go to the Front end section for more details). Set to **false if this is meant to be an unlogged page**.
- **Messages:** This will be where we **allocate the session messages we previously stored in the messages variable** prior to clearing them. We would render this messages in the front-end when the page is accessed.

Pages that require your to be only logged in

These are pages whose only requirement for access are to be logged in. Even if a user has 2FA enabled, for example, they would not need to enter a 2FA verification code to access it. An example of this page would be the 2FA authorization page itself. **It can be viewed only when the user is logged in but does not require additional verifications to be viewed**. The following is the route to the 2FA page:

```
app.get('/2fa_checkup', controller.onlyLoggedIn, (req, res) => {
    var messages = req.session.flash;
    req.session.flash = null;

    //Page for 2fa checkup
    const VerificationPage = require('../views/verification.jsx');

    var check_code = ReactDOMServer.renderToString(
        React.createElement(VerificationPage, {
            name: 'Gravity - Two Factor Verification',
            user: req.user,
            messages: messages,
            dashboard: true
        })
    );

    res.send(check_code);
});
```

As you can see, it follows virtually the same steps as the previous GET page type, the difference being that **a third parameter was added inside the app.get function between the endpoint name and the route function**. This parameter is also a function. **The controller.onlyLoggedIn function redirects a request to the root route ('/') if there's no user logged in**. This is a method from the file config/controller.js file and it is loaded at the top of the route file.

Additionally, you can see that **a new prop was added 'user' which contains the information of the user that made the request to that page** and is stored in the request itself. **Always include user in the props when working on this or the following type of route**.

Pages that may require additional verification

This are the pages where 2fa or any other form of verification should be applied if the user requested those extra security clearances. **The format would be identical to the previous one except that you replace the onlyLoggedIn function with the isLoggedIn function**, which verifies if 1) the user needs to provide 2FA verification and 2) if that verification has been processed. **If verification has not been processed, the user would be redirected to the proper verification page until verification is provided**.

Which permission to use

Which permissions you decide to use will depend your situations but, as a rule of thumb, always grant the highest level of permission (isLoggedIn) to all pages that are meant to be

viewed only by users of your site even if you don't plan on having them implement 2FA. You can always add 2FA or any other form of authentication later on and use the onlyLoggedIn permission to implement any form of verification you want.

Finally, there is also a **isLoggedInIndex for specific rerouting behaviour from the index page.**

Creating new route files

There are **currently two methods that generate route files in Gravity.** The first one is the 'npm run gravity:app:scaffold' which generates all the files are needed to display a page that will get and save records to and from the database, and the 'npm run gravity:app:page' which creates a route file and a view file that matches the route in that file. The following is the full route file that controls that test page we saw earlier:

```
module.exports = function(app, passport, React, ReactDOMServer) {
    var session = require('express-session');
    var flash = require('connect-flash');
    var controller = require('../config/controller.js');

    //============================================================
    // This constains constants needed to connect with Jupiter
    //============================================================

    var axios = require('axios');

    //Loads Gravity module
    var gravity = require('../config/gravity.js');

    app.get('/test_page', (req, res) => {
        var messages = req.session.flash;
        req.session.flash = null;

        const PageFile = require('../views/test_page.jsx');


        var page = ReactDOMServer.renderToString(
            React.createElement(PageFile, { name: 'Gravity - Test Page', dashboard: false, messages: messages })
        );

        res.send(page);
    });
}
```

We display the route code as a function because it is called as such in the server.js file, which passes the app object, our passport configuration for logged pages and our React libraries for page display.The libraries called before our route are common libraries used throughout gravity for route handling, including our controller.js file and our gravity.js file. Not all of this libraries are needed and you are free to add or remove based on your needs.

You **can add new routes manually by adding .js files inside the controller folder that match the above format and** strongly encourage you to **learn** more about both **express** and **axios**, the **two libraries most used libraries** to handle api calls in Gravity.

Routing order

By default, the **server.js file loads every file in the 'controller' folder in alphabetical order, which is why the _appliction.js file will always be the first one to run as long as the dash is kept at the beginning.** The **api.js** routing file mentioned in the previous section is loaded **earlier** than all the other routes in the system since they are **supposed to be used by multiple models within gravity**. Those generic routes contain **exception lists** that tell the system if they are meant to be skipped so that a later route in the load order will pick up the client request.

Make sure that whatever new route you add to your gravity app does not conflict with another route earlier in the order. If it does, you will need to write an exception wherever the conflict originates or move your route directly to an earlier load.