

---

**taxasoft-ghini**

**Apr 11, 2021**



---

## Contents:

---

<b>1</b>	<b>introduction</b>	<b>1</b>
1.1	serving . . . . .	1
1.2	fallback . . . . .	1
1.3	showcasing . . . . .	1
<b>2</b>	<b>participating</b>	<b>3</b>
2.1	your account . . . . .	3
2.2	organize your pictures . . . . .	4
2.3	bulk inserts . . . . .	4
2.4	adding images . . . . .	4
2.5	validating . . . . .	6
<b>3</b>	<b>searching</b>	<b>9</b>
<b>4</b>	<b>technical documentation</b>	<b>11</b>
4.1	site installation . . . . .	11
4.2	run locally . . . . .	12
4.3	publishing the site . . . . .	12
4.4	rest-api . . . . .	13
4.5	importing from ghini.desktop . . . . .	15
<b>5</b>	<b>Indices and tables</b>	<b>17</b>



Ghini server is a AGPL software, aiming at letting professional and amateur botanists share knowledge, in the form of plant images, localization of plant observations, taxonomic identifications.

Ghini server follows established best practices for botanical collections, so that it can successfully be used by botanical institutions needing a strong database foundation.

The combination ghini.server + ghini.web is the natural successor of ghini.desktop, a GPL desktop program. Even if the interface are very similar, they are based on different technologies, and are not compatible at the database level. There might come a ghini.desktop version compatible with ghini.server.

## 1.1 serving

Installation of a ghini.server site amounts to installing a standard django service. This is a rather technical task, so please either look for and refer to the corresponding documentation, or ask for advise and support.

## 1.2 fallback

Please refer to ghini.desktop

## 1.3 showcasing

Ghini server's initial goal is to showcase itself in the form of geographic botanical collections. If you have a use case and want to participate, please contact the Ghini team. As of now, there's the following projects:

- [cuaderno](#), a botanist's collection handbook.
- [almaghreb](#), wild plants in the Atlas Region.
- [tanager](#), a small privately held botanical garden. The data in the database is by far not the complete garden's collection.

- *caribe*, wild plants in the Caribbean Area.

Participants get access to ghini's editing facilities and the django admin interface to the database, while anonymous visitors can browse and query the data from the ghini interface.

This page explains how to contribute data to a running ghini site.

## 2.1 your account

You need an account for the site, it needs to be enabled, and to make you part of the staff. You do not need superpowers to contribute data.

### Permissions

Active

Designates whether this user should be treated as active. Unselect this instead of deleting accounts.

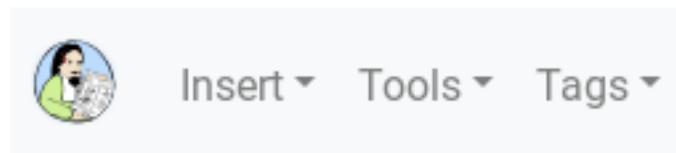
Staff status

Designates whether the user can log into this admin site.

Superuser status

Designates that this user has all permissions without explicitly assigning them.

When logged in as staff member, you will see the *Insert* menu next to the Ghini icon, before the *Tools* menu,



moreover, in the Results page, right-clicking on any results row will show the context menu associated to the row type.

## Arecoideae Beilschmied

Areaceae - 0 verifications; 0 subtaxa

Edit

Add sub taxon

Add verification

Delete

If you do not see these, you are being handled by the software as a visitor, and you need to  and log back in with your full permissions.

## 2.2 organize your pictures

The software requires that you organize your plant images by physical plant, and by accession. Please make sure your data follows this order before you start.

A running definition of a database accession for plants photographed in the wild, it is a group of nearby plants, clearly of the same species and variety, very likely of the same age, and clearly belonging to the same population. Decide for yourself if you want to use the term accession as a synonym for population.

A running definition of a database plant for the same case, that would be a group of plants within the same accession, difficult to separate from one another. They all share the same geographic coordinates, and you can provide a reasonable estimate of the number of individual plants composing the group.

## 2.3 bulk inserts

If you have a large amount of accessions that you want to document with plant images, you may request a bulk insertion. Please write a friendly email to Mario Frasca <ghini@anche.no>, stating your needs. It makes sense if you have more than, say, twenty accessions.

## 2.4 adding images

The basic logic is: first have all your accessions in place, which you either do manually, or with a request for bulk insertion, then navigate to [the URL for adding plant images](#).

You will be presented with this form:

---

Home › Garden › Plant images › Add plant image

## Add plant image

<b>Plant:</b>	<input type="text" value=""/>	 
<b>Height:</b>	<input type="text" value="1"/>	 
<b>Width:</b>	<input type="text" value="1"/>	 
<b>Image:</b>	<input type="button" value="Browse..."/>	No file selected.

In this form you basically repeat: choose a plant from the database, ignore height and width, choose a picture from your file system (this will provide the correct values for height and width), then click on one of the three *Save* buttons:

<input type="button" value="Save and add another"/>	<input type="button" value="Save and continue editing"/>	<input type="button" value="SAVE"/>
---	--	-------------------------------------

If you are adding an image for a plant already represented in the database, simply start typing the plant accession number, and the software will present you the plants matching the text you are typing. Each time the image is relative to a new plant, you need to add the database object that describes the plant in question. You do so by clicking on the + button next to the plant chooser widget.

This will open a new window,

Add plant

Accession:    

Code:

Location:    

Quantity:   

Geometry:



Here you choose your **accession**, indicate a numerical sequential value for the plant **code** within the accession (you need to so manually), select or add a plant **location** (it would be the name of the area for your observations), provide an estimate of the **quantity** of individual plants in the plant group, and use the **geometry** map to indicate an approximate plant position. You do so by activating the *draw a marker* widget, then clicking on the approximate spot in the map. Zoom in or zoom out, and peruse the other widgets as necessary.

Do not forget to *Save* your changes.

Back to your Plant Image insertion window, browse to your image, confirm by *save* or *save and add another*.

## 2.5 validating

A different activity is relative to validating accessions with taxonomic information. You can do so from [the admin interface](#),

Add verification

Accession:  ✎ +

Taxon:  ✎ + ✕

Qualifier:

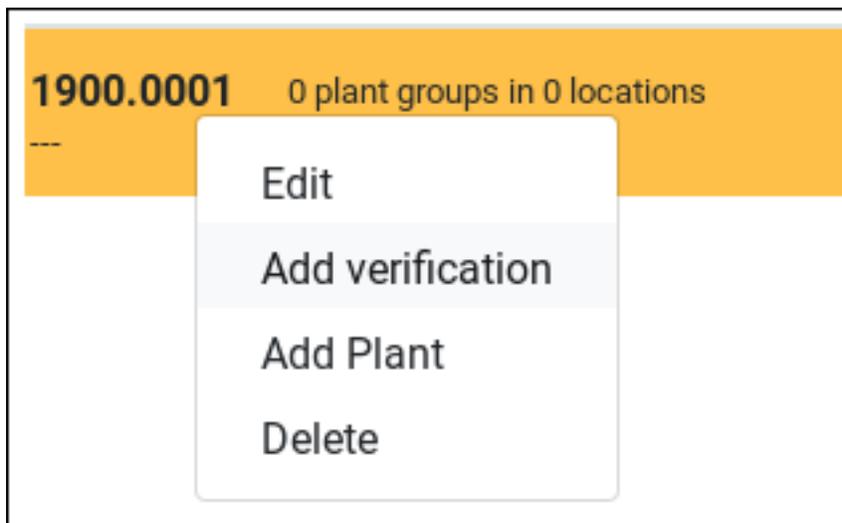
Seq:  ▲ ▼

Level:

Contact:  ✎ + ✕

Date:  Today 📅  
Note: You are 5 hours behind server time.

but you can also use the results window, activating the context menu for the accession you want to validate.





ghini exposes several search strategies, the most simple of which allows you to enter values, or <TERMS>, and ghini will attempt matching them against the data:

**terms** <TERMS>

Filter, based on the value of the default selection fields of any of the search domains. <TERMS> is a space-separated list of values. A match is found if one of the default search fields in one of the default search *domain* is found equal to one of the given terms.

The filter succeeds on an element of a domain if all the terms match for that element.

If prefixed with the optional keyword `or`, the filter succeeds if any one of the terms matches.

The **terms** strategy can be seen as a shortcut to the following, where we limit the search to a specific <DOMAIN>.

**domain** <DOMAIN> <op> <TERM>

Filter based on the value of the default selection fields of the specified <DOMAIN>, applying a specific comparison.

<DOMAIN> is one of the data types, that's `accession`, `plant`, `taxon`, etc.

<op> is a comparison operator, that's `=`, `<=`, etc.

The **domain** strategy can be seen as a shortcut to the following, in which we explicitly mention the <field> to be matched.

**single field** <DOMAIN> . <field> <op> <TERM>

Filter, based on the value of a single field of the specified domain.

Besides the three above term matching strategies, we have a rather complex and powerful SQL-like search strategy.

**sql-like** <DOMAIN> `where` <COMPLEX-QUERY>

This is the most generic and powerful search. You give a search domain, then specify an expression to be matched. The literal string `where` in second position is what triggers usage of this strategy.

COMPLEX-QUERY is an expression, composed of boolean tests on fields, either of the domain or of a connected domain (think of `accession.plants.images`), tested with an operator (think of `=`, `like`, `contains`, `against` values (think of a string, or a number). Boolean tests can be combined with `and`, `or`, `not`, and parentheses.

For domains that specify a geometry (for example, `plant`, where the `geometry` field is the current plant location) you can use the clause `geometry in area`, where you specify the area by selecting it in the **map** page.

Please consider that ghini will fall back to one of the above more generic search strategies if the query is somehow incorrect. This will most likely return an empty result set.

**depending** <query> | depending

On any of the previous search strategies, you can append the query modifier `| depending`. This changes the resulting query-set, applying the *depending* function to each of the elements in the original result.

Logged in users can use the `ghini.server` API to run these queries, or use the `ghini.web` interface to enter them and have the results nicely organized in the various `ghini.web` tabs.

## 4.1 site installation

The installation for `ghini.server` just the standard installation procedure for any python3 program: first of all make sure you *can* create virtual environments, then create and activate the virtual environment where you will install the software, and finally install the software, either from the global Python Package Index, or from a custom source.

---

**Note:** Please do not even think of asking me how to install and run `ghini.server` on Windows. Even if it is definitely possible (as said above, `ghini.server` is a standard Python package), it is just as definitely out of my sphere of interest, helping you with using Windows for serving data over the internet, it's just “*not done*”, not only in my opinion. You can **use** `ghini` from a Windows client, even using the Microsoft Edge browser, but here I will only explain how to install `ghini` server on a unix server, and specifically GNU/Linux.

---

Get yourself a recent Python version. Now that Debian has released its 10th version with Python3.7, there's no excuse for staying on anything older than that. Anyhow, the software works fine with Python3.5 (the one which came with Debian 9) and is continuously tested against Python 3.5, 3.6 and 3.7.

The other non-automatic step is installing `virtualenv` and `pip`. Look for the details somewhere else if you think you need more details. If you're on Debian, `virtualenv` is contained in the package `python3-venv`, while `pip` gets automatically installed when you create your virtual environment. Keep in mind that the command to create virtual environment is the not-too-mnemonic `python3 -m venv`:

```
sudo apt-get install python3-venv
python3 -m venv ~/.virtualenvs/ghini/
. ~/.virtualenvs/ghini/bin/activate
pip install --upgrade pip
```

---

**Note:** At the time of writing, `ghini.server` isn't yet published as a PyPI package, so even if we describe the PyPI method, it might not work yet.

---

Decide whether to download the packaged software from the Python Package Index, PyPI, or use the github version.

Installing from PyPI is as easy as:

```
pip install ghini.server
```

If using github, you will need `git`, then you must decide whether you use your own fork, or the official repository, and choose the branch you want to check out.

In short and in practice:

```
mkdir -p ~/Local/github/Ghini
cd ~/Local/github/Ghini
( git clone git@github.com:Ghini/server.git ||
  git clone https://github.com/Ghini/server.git )
cd server
git checkout ghini-3.2
pip install -r requirements.txt
```

In either case, `pip install` takes care of all dependencies, and you need take no further steps.

## 4.2 run locally

As it is standard practice with any Django program, after installing the software, you need to set up the database, subsequently to migrate it, create the users you need, import your data, then you can run the site and navigate to it.

The first step, to set up the database, you do this by tweaking the `ghini/settings.py` file, or by copying it to a custom named settings file, respecting the format `ghini/settings_<name>.py`. You would do the first if you only need one site, and the second if you plan running multiple instances of `ghini.server`.

With the virtual environment active, run the command `./manage.py migrate`. It will load your default settings, or you append the option to the command: `--settings ghini.settings_<name>`.

To create users, the easiest would be to enter the django shell, then to do something along the lines:

```
from django.contrib.auth.models import User
me, _ = User.objects.get_or_create(username='mario')
me.set_password('mario') # or maybe something safer
me.save
```

The standard way to run your site in development mode is to just run `./manage.py`, requesting the `runserver` command. Specify on which port to serve your site, then navigate locally to it.

All the above would work in Windows as well, the next steps are about going in production, for which we advise the use of `nginx` combined with `uwsgi`, and a unix system.

## 4.3 publishing the site

The above method lets you navigate your data using only the Django server. This is not how you would run a production site, where requests first reach your web server, which takes care to redirect each part of the request to the appropriate code producing that part of the composite response.

Serving `ghini.server` is based on `nginx` and `uwsgi`. We advise an external `nginx` server, so that all incoming web requests reach that first.

Set up `nginx` so that it makes sure the clients are talking the secure `https` protocol.

Most requests will cause a further cascade in simpler requests. The rules in our configuration establish whether `nginx` will satisfy the request itself by serving static data, or if it will redirect the request to a local `uwsgi` socket, behind which there is our Django `ghini.server`.

Run `uwsgi` in `emperor` mode and let it inspect the `uwsgi.d` directory in the `ghini.server` main directory.

Produce or update your `ini` files by running the provided `create-ini-files.sh` script, corresponding to the content of the `/etc/nginx/site-enabled` directory, for all site definitions looking like `ghini.server` instances.

## 4.4 rest-api

We have a main api for interacting with the database.

Each object has its URL, which really identifies the object (e.g.: plant #1 for accession 101 in year 2001):

```
/garden/accession/2001.0101/plant/1/
```

Removing the object's trailing identifier from the URL gives the class URL (e.g.: the plants collection):

```
/garden/accession/2001.0101/plant/
```

The trailing slash is part of the URL, but the server will add it if it's missing.

### collections

We organized the objects in three sections: `taxonomy`, `collection`, `garden`. There might come some day a `herbarium` or `seedbank` section, or we may reorganize in fewer sections, we will see. As of now, we have these collections:

```
/taxonomy/rank/
/taxonomy/taxon/
/collection/accession/
/collection/contact/
/collection/accession/<code>/verification/
/garden/accession/<code>/plant/
/garden/accession/<code>/plant/<code>/propagation/
/garden/location/
```

Verifications and Plants only make sense in combination with an accession, so their collections are behind an accession code. Same for Propagations, which only make sense in relation with the mother plant.

### individual objects

Append a primary key to a collection URL, and you get the URL for an individual within the collection.

As far as their URLs are concerned, `rank`, `taxon`, `contact` have a primary key which is a sequential number, with no semantics.

Accessions have their own accession code, Plants have a sequential plant code within the Accession they belong to, Verifications also have a unique sequential number within the Accession they describe. Propagations have a sequential number within their mother Plant.

---

**Note:** If we generalize the database to model more than one garden, we will need to associate accessions to gardens, we will probably identify gardens with a stub, and will prepend accession urls with a garden stub code. As of now, we only deal with a single garden.

---

### GET and her sisters

Collection URLs implement the `GET` and `POST` verbs, respectively for getting the whole collection (or a selection thereof), and for adding an individual object to the collection. These URLs get a `-list` suffix in their Django name.

Individual URLs implement the `GET`, `PUT` and `DELETE` verbs, with their obvious meanings, applying to the specific individual only. These URLs get a `-detail` suffix in their Django name.

### more URLs

Collections also have an URL for the empty html form, to be populated by the user and posted to the server. The corresponding Django names have suffix `-post-form`.

Individual objects have more entry points, respectively for:

- The populated html form (django suffix `-form`)
- A json data dictionary for the infobox (django suffix `-infobox`)
- A dictionary with several representations for the same object (django suffix `-markup`)
- A json data dictionary with *depending* objects, and the definition of the concept depends on the object. A Location considers the plants located there as its depending objects, a Taxon its subtaxa **and** the accessions verified to it. The result has the same shape as the dictionary returned by a search. (django suffix `-depending`)
- A rendered html page with object pictures (django suffix `-carousel`)

### search API

`filter/` and `get-filter-tokens/` are the main query api entry point. Both expect a `q` parameter, which they interprets according to several search strategies. Search strategies are described in some detail in the user manual.

The result of a `get-filter-tokens/` request is a dictionary, where the keys are the names of the collection in the result, and the values are *tokens*. You get as many tokens as the non-empty collections matching your query.

The next step on the client side is to enter a loop to *cash* your *tokens*. Each invocation of the `cash-token/<token>/` returns you a dictionary with three entries:

- `chunk` holds the list of items.
- `expect` specifies the length of the expected complete set. One possible use is to update a progress bar.
- `done` tells you whether this was the last chunk.

Attempting to cash a token which was already paid in full will provide the empty result. Same will happen if you attempt to cash an invalid token. The empty result is `expect:0, done:True, chunk:[]`.

If you are somewhat too quick in cashing a new token, the `expect` value could still be a large hard-coded value. The correct value is computed in a separate thread, so the server can provide all tokens as soon as possible.

Tokens will expire after some delay in cashing them. This prevents queries to stay active in the system while not any more relevant.

For queries where you expect a small result set (less than ~70 elements), you can may prefer the `filter/` entry point. `filter` short-circuits this process, providing the concrete result at once, in a dictionary having the same external structure as the `get-filter-tokens` result, one list of objects per non-empty collection, and values as the above chunk lists.

One more entry point in this group is `count/`, it accepts the same parameters as `filter` and `get-filter-tokens`, and returns a dictionary with same external structure. The values in this case are the matching query `count()`, plus a grand total under the key `__total__`. You can use this to decide whether to use `filter` or the chunked approach `get-filter-tokens`.

On the server side, executing a search corresponds to constructing one or more queryset. Each element in the queryset is subsequently converted into a dictionary, with the structure:

**inline** The string shown in the result. It may contain html tags.

**twolines** Three elements to be shown in different parts of the client.

**infobox\_url** The url to get the corresponding infobox.

The `inline` and `twolines` entries are meant to be included in the results box. The `infobox_url` provides quick access to the URL where we will get the infobox data, but you can just replace the trailing `infobox/` part and replace with whatever other valid suffix. at the moment of writing, the URLs implemented are *form/*, *markup/*, *depending/*.

## 4.5 importing from ghini.desktop

Please consider this work in progress, try out the instructions, and be prepared to ask for help or to open an issue if the present instructions do not work.

First of all: taxasoft-ghini is not complete, not yet. The current goal is to have it do something useful, and to be visible on-line, it does not (yet) substitute ghini.desktop. Not at all. Expect things to be exciting, but do not expect things to work out of the box.

Got this? Good, now let's see how to copy your ghini.desktop collection into taxasoft-ghini!

### 4.5.1 from ghini.desktop

1. open ghini-1.0
  1. export your (complete) data to csv.
2. close ghini
3. open ghini-1.0 again,
  1. create a new sqlite3 connection,
  2. let ghini create the database.
  3. import the data, this will again initialize the database.
4. close ghini

the result of the above steps is an expendable sqlite3 database: this way whatever we do on it, it has zero impact on your original data.

5. remove all taxonomic information that is not used. we do this straight on the expendable database:

```
sqlite3 ghini.db
delete from genus where id not in (select genus_id from species);
delete from family where id not in (select family_id from genus);
delete from genus_synonym where genus_id not in (select id from genus);
delete from genus_synonym where synonym_id not in (select id from genus);
```

6. consider removing history too, it's not imported anyway:

```
delete from history;
```

7. open ghini.desktop-1.0
  1. export your (reduced) data to csv.
 

this will take a fraction of the time for the previous export.
8. close ghini

## 4.5.2 now to taxasoft-ghini

1. enter the directory of your check-out;
2. activate the virtual environment;
3. move any previous database out of the way;
4. create a new database and initialize it:

```
./manage.py migrate
```

5. consider whether you also want the intermediate taxa, between ranks familia and genus. since importing this information takes rather long, it is not included in the 'migration' command. if you want this data, you must request the import explicitly, with:

```
./manage.py import_genera_derivation
```

have something else to do in the meanwhile, this will take no less than one full hour. on my laptop, writing to a sqlite3 database, it lasts 2 hours.

if you're in a hurry, ask for a partial genus import, limiting to the genera in your trimmed database:

```
./manage.py import_genera_derivation --filter-genera <your genus.txt file>
```

you can repeat the command without filtering, whenever you know you're not going to use the database for a couple of hours.

6. run the command:

```
./manage.py import_desktop <location of second export>
```

this will output as many + as the objects it inserted, as many . as the objects it already found in place. for species, a v is added if the related species is at lower rank.

the genus list in particular, that should be just a sequence of dots. if it is not, it's because you're importing genera that were not created during the previous steps. that's clearly not good and you should review your data.

the opposite goes for the species list: remember that with ghini reloaded fictive species are not any more needed. A dot tells you that the corresponding taxon was found in the database, at some higher rank.

it is normal that importing accessions takes longer: for each object we are creating not only the accession but also the verification object that links the accession to the corresponding taxon.

7. create your superuser:

```
./manage.py createsuperuser
```

8. run your server:

```
./manage.py runserver
```

9. I'm sure there will be errors. please open issues about them, and if you have a solution, propose it.

## CHAPTER 5

---

### Indices and tables

---

- `genindex`
- `modindex`
- `search`