Service infrastructure

Python module and dependencies

The service backend is all handled by python modules all packaged in this repo. The modules can be installed using pip directly from the main branch of the repo. It is recommended to deploy the service in an Ubuntu machine. The DSSAT model doesn't need to be installed as it is included in one of the dependencies of the project. Before setting up the python environment, Postgres and GDAL must be installed in the system. After Postgres and GDAL are installed, the service python package can be installed directly from the repo using the next commands:

```
git clone https://github.com/SERVIR/dssat_service
cd dssat_service
pip install .
pip install GDAL==$(gdal-config --version)
```

The previous steps must successfully install the `dssatservice` python package. Any error during the installation is likely due to missing system dependencies. In addition, to download and ingest the ERA5 data, the CDS API must be configured. This tutorial walks you through.

The modules in `dssatservice` handle the setting-up and operation of the service. There are four main modules, each handles different operations in the service. The `dssat` module handles the model run. The `database` module handles all the database operations. The `data` module handles the data downloading, transformation, and ingestion operations. Finally, the `ui` module handles the interaction between the user and the service; therefore, it has some classes that manage the django session, store user defined parameters, and generate the plots. All modules and functions are well documented using docstrings.

Database

The database is a PostgreSQL database with the PostGIS extension. PostGIS extends the capabilities of the PostgreSQL relational database by adding support for storing, indexing, and querying geospatial data. The database stores all the data necessary for the service operation. It stores: weather data (historical and forecast), soil profiles, administrative boundaries, cultivar options (options shown to the user), other data needed. Each country or domain has its own schema which contains all the tables. There are two types of tables here identified: read-only tables, which are created only once, and no new data is added to them, one example are the soil tables; and read and write tables, which need to be regularly updated, such as the tables containing weather data. The service is successfully deployed, and simulations can be run only

after all tables have been created and populated. If any table is missing the model run will fail at some point.

The python modules contain the functions needed to create and populate the database. The entire process of setting up and populating the database is described in the setting up the service infrastructure section.

Read-only tables

Once these tables are created the data is ingested only once. No new data is supposed to be added to the tables after the domain (schema) is created and the data is loaded in the service database. Next table lists the read-only tables, along with their columns and what each table is for. The **bold column names** are indexes of that table.

Table 1.1. Read-only tables in the database.

Table	Columns	What for
admin	**gid (INT)** **Geom (POLYGON)** **Admin1 (STRING)** var1 (ANY) var2 (ANY) …. varN (ANY)	Contains the administrative subdivisions. It will also contain static variables associated with each subdivision. This could include for example custom planting dates or cultivars.
cultivar_options	**admin1 (STRING)** **maturity_type(STRING)** cultivar(STRING) season_length (INT)	It contains the cultivar options for the app. Cultivars are the different varieties of the same crop that can be planted. The cultivar is defined by a 6 character code (cultivar column). Each admin1 will have five cultivar options, one per maturity type: Very short, short, medium, long, and very long. The cultivar column has the DSSAT cultivar code. Season_length column is the average season length for that cultivar in that admin1. These cultivars could be different from those used for the operative forecast.
era5_clim/prism_clim	**rid (INT)** rast (RASTER) **month (INT)** **variable (STRING)**	Contains monthly climatologies. Tmin, Tmax: average T amplitude: average, std This is used to bias-correct the forecast dataset based on the reanalysis climatologies.
soil	**geom (POINT)** soil (STRING)	Contains the soil profiles. It also contains different crop mask bits for

	cropmask1 (BOOLEAN) **cropmask2 (BOOLEAN)**	each soil profile.
static (optional)	**rid (INT)** **rast (RASTER)** **par (STRING**)	It contains rasters with static model parameters. - TAV (Average annual soil temperature) - TAMP (Amplitude of the annual soil temperature function) **One must be careful of providing a raster that includes the entire area**

Read and write tables

These tables are created once, and are updated as new data is available. This includes tables containing the latest yield forecast, historical weather data (reanalysis), and weather forecast data (forecast). The climatology table is not updated on a frequent basis, instead it is updated any time the climatology reference period wants to be updated. The climatology table (era5_clim) can only be created after the reanalysis tables (era5_tmin, era5_tmax, etc.) are all populated with at least one continuous year of data.

Table 1.2. Read and write tables in the database.

Table	**Columns**	**What for**
latest_forecast	**gid (INT)** **Geom (POLYGON)** **Admin1 (STRING)** Obs_avg (INT) Pred (INT) Pred_cat (INT) planting_p (STRING) Ref_period (STRING) Nitro_rate (INT) Urea_rate (INT)	It contains the geometries and forecast information. Obs_avg column is the average observed yield in the reference period (kg/ha). Ref_period is the period used as reference for yield, e.g. '2000-2020'. The reference period and yield are used to assign the predicted category (Low, Normal, High; pred_cat column). Plantain_p and Nitro_rate are the planting window and nitrogen rate used to produce the forecast. Urea_rate is just the Urea equivalent of Nitro_rate. The creation of this table is explained in the Routine scripts section.
latest_forecast_overview	**Admin1 (STRING)** **Run (INT)** devPhase (FLOAT) stressWatPho (FLOAT) stressWatGro (FLOAT) stressNitPhto (FLOAT) stressNitGro (FLOAT)	It contains the stress table from the DSSAT Overview file for all ensemble members. The DSSAT Overview file (OVERVIEW.OUT) is a text file generated after a successful run of the model. The creation of this table is explained in the Routine scripts

	…..	section.
latest_forecast_results	**Admin1 (STRING)** **Run (INT)** HARWT (INT) MAT (INT) FLO (INT) ….	It contains the DSSAT run results for the end of the season for all ensemble members. The creation of this table is explained in the Routine scripts section.
era5/prism_tmin, era5/prism_tmax, era5/prism_srad, era5/prism_rain	**rid (INT)** **rast (RASTER)** **fdate (DATE**)	Contains data from the AgERA5 or PRISM dataset for tmin, tmax, srad, and rain. As PRISM does not have daily srad, AgERA5 srad is ingested.
nmme_tmin, nmme_tmax, nmme_rain	**rid (INT)** **rast (RASTER)** **fdate (DATE)** **ens (INT)**	Contains data from the NMME dataset for the tmax, tmin, and rain. srad is not available for NMME data, therefore, it is estimated during the model run.

Service Operation

Setting up the service database

After all the `dssatservice` Python package is installed in the environment, the next step is to set the database. All tables are created and populated using a function from `dssatservice`.

Creating the db

Postgres must be installed at this point. The necessary extensions can be installed and the database created using the following lines of code in the terminal:

```
sudo add-apt-repository ppa:ubuntugis/ppa
sudo apt-get update
sudo apt-get install postgresql-14-postgis-3
sudo apt-get install postgresql-14-postgis-3-scripts
sudo apt-get install postgis
sudo -u postgres createuser -s $(whoami)
createdb dssatserv
echo "create extension postgis;" | psql -d dssatserv
echo "create extension postgis_topology;" | psql -d dssatserv
echo "create extension postgis_raster;" | psql -d dssatserv
```

After creating the database, we have to create the tables and ingest the data. Now, the db population process involves: creating one schema per domain (country), creating and populating the read-only tables, and creating and populating the read and write tables. The steps to populate the entire db will be described next. The process must be followed in the order next established. The debug.py file contains one example of setting up the service for Rwanda. All the input data needed to set the service up for the example is in the data_example.zip file in the github repo.

All the functions to populate the database are in the `database` and `data.ingest` modules. In this document, the aliases for the `database` and `data.ingest` modules will be "db" and "ing" respectively. Also, the *con* parameter in all the functions, is an open connection object to the database. That connection can be instantiated using the `psycopg2.connect` function (Example).

admin table (code example)

The admin table contains the geometries of the domain or country. It is created when a new country or domain is added to the database using the `db.add_country` function.

Table 2.1. Details on the function to create admin table

Function	`db.add_country(`
	`  con:pg.extensions.connection,`
	`  name:str,`
	`  shapefile:str,`
	`  admin1:str="admin1"`
	`)`

What it does	It creates the *name* schema and the *name.admin* table. *name* is the name of the country or domain.The *name.admin* table will contain the geometries of the *shapefile* file. The *admin1* parameter indicates what is the name of the field on the attribute table of the *shapefile* that has the name of each administrative unit. This function also creates some of the other tables of the database. These include: reanalysis tables (era5_*), soil table, and cultivars table. Therefore, you won't have to create those tables yourself.
Comments on the input files	*shapefile* is the path to the shapefile of the country to add.

soil table (code example)

The soil table is created in the previous step. However, the table is empty, and the soil data must be ingested. The soil data is ingested using the `ing.ingest_soil` function.

Table 2.2. Details on the function to ingest soil data

Function	`ing.ingest_soil(`
	`  con:pg.extensions.connection,`
	`  schema:str,`
	`  soilfile:str,`
	`  mask1:str=None,`
	`  mask2:str=None`
	`)`

What it does	It takes the *soilfile* DSSAT .SOL file and ingest it in the *schema.soil* table. The *soilfile* must include all the soil profiles of the domain or country, as it is defined in the Global High-Resolution Soil Profile Database for Crop Modeling Applications. The *soilfiles* for each country should be taken from that database. The function does not verify the match between the soil points and the domain geometries. Then, it must be verified that all the administrative units in the domain have soil profiles defined within their boundaries. Each soil profile occupies one row in the table. Each profile is represented as a point geometry. The coordinates of each point are defined in the *soilfile* (Look at this example). A masking field is also created in the table. This is done using two cropmasks. The path to the cropmask tiffs are defined in *mask1* and *mask2*. If mask1, and mask2 are not passed, then all soils profiles in the file will be considered as Agricultural areas. The idea of having two cropmasks is to be flexible when masking out pixels. This was thought to allow taking soil samples from places that are not predominantly cropland areas. Then, in our case, *mask1* is a cropland product, while *mask2* is a mask excluding forests, cities, barren, and ice land cover categories (places where crops can't be grown). Then, mask1 is more restrictive than mask2. The framework is designed to take samples using *mask1* if possible. If no points can't be sampled using *mask1*, then *mask2* is used.
Comments on the input files	*soilfile* is the path to the .SOL file for the country. It must be taken from the Global High-Resolution Soil Profile Database for Crop Modeling Applications, or at least it must follow the same format. *mask1* is the path to the more restrictive cropmask raster. *mask2* is the path to the less restrictive cropmask raster. The cropmask rasters should be in the same resolution as the soil data. By default it is 10km.

static table (code example)

This table stores static spatial information (parameters) in the form of rasters. It is created and populated using the `ingest_static` function. The name of the static variable is passed in the *parname* parameter of the function. This table is optional. Up to this point, only the TAV and TAMP DSSAT weather parameters are uploaded into the table. These optional parameters (TAV and TAMP) allow a better definition of some assumptions of the model. The TAV and TAMP rasters can be generated using the ERA5 data with this GEE script.

Table 2.3. Details on the function to ingest static data

Function	`ing.ingest_static(`
	`  con:pg.extensions.connection,`
	`  schema:str,`
	`  rast:str,`
	`  parname:str`
	`)`

What it does	It creates the *schema.static* table if it does not exist, and ingest the raster in the *rast* path, under the *parname* name.
Comments on the input files	*rast* is the path to the raster file to be ingested. It must be in the same resolution as the soil and weather data (10 km). It must be verified that the raster covers all the area defined in *schema.admin*.

era5/prism tables (code example)

These tables store the historical weather data. The name of the table follows the era5/prism_VAR format, where the first term is the name of the reanalysis dataset, and VAR is the name of the variable. Those tables are created with the admin table (`db.add_country`) for the ERA5 reanalysis dataset. The ERA5 data is ingested using the `ing.ingest_era5_series` function, and the PRISM data using the `ing.ingest_prism_series` function. If a different weather dataset is to be ingested, then a similar function must be developed for that reanalysis dataset.

When setting the service up, it is recommended to download the past 20 years of weather data for each domain or country.

Table 2.4. Details on the function to ingest the ERA5 and PRISM historical weather data.

Function	`ing.ingest_era5_series(`
	`  con:pg.extensions.connection,`
	`  schema:str,`
	`  datefrom:datetime,`
	`  dateto:datetime`
	`)`
	`ing.ingest_prism_series(`
	`  con:pg.extensions.connection,`
	`  schema:str,`
	`  datefrom:datetime,`
	`  dateto:datetime`
	`)`

What it does	It ingest the weather data in the next tables: *schema.era5_tmax*,

	schema.era5_tmin, *schema.era5_rain*, and *schema.era5_srad* if ERA5 data is ingested, and the *schema.prism_tmax*, *schema.prism_tmin*, *schema.prism_rain*, and *schema.prism_srad* if PRISM data is ingested. The ingested data covers the period between *datefrom* to *dateto*. For this function to work, the cds API must be configured in the environment. Here are some instructions to do it.
Comments on the input files	

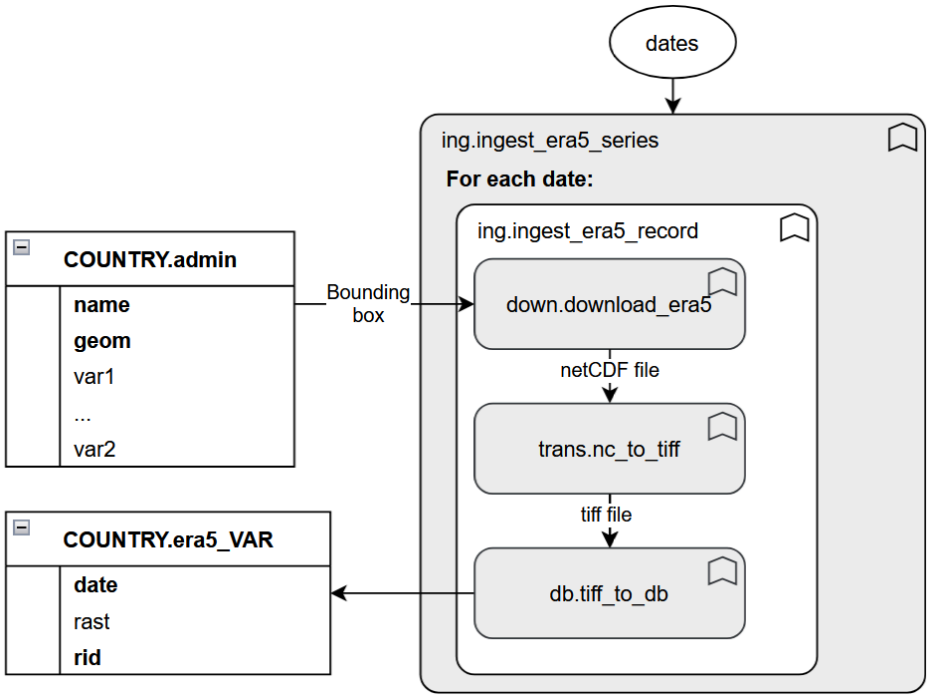


Figure 1. Schematic of the ingest_era5_series function.

era5_clim table (code example)

It stores the monthly temperature climatology for the country or region. This information is used for the bias adjustment of the NMME forecast data. It is created and populated using the `db.calculate_climatology` function. Before running this function the database must have at least one complete year of data. It would be recommended to have at least 20-30 years of data for the climatology to be representative.

Table 2.5. Details on the function to create the ERA5 monthly climatology.

Function	`ing.calculate_climatology(`
	`  con:pg.extensions.connection,`
	`  schema:str,`
	`  weather_table:str='era5'`
	`)`

What it does	It calculates the monthly averages of the average temperature ([tmax+tmin]/2) and daily temperature range (tmax-tmin) using all the historical ERA5 or PRISM data available for the domain. It saves the data in the *schema.era5_clim* table if 'era5' is indicated as the *weather_table* and *schema.prism_clim* if 'prism' is indicated as the *weather_table*. This function can be re-run to recalculate the climatologies as new data is available. If a different historical weather dataset is implemented, then this must be modified in this function.
Comments on the input files	

Test end-of-season run (code example)

At this point all the data needed to run an end-of-season simulation is available in the table. An end-of-season simulation is a simulation run when the season has already ended, therefore, all the historical data is available. Regardless of the weather data source, the `dssat.run_spatial_dssat` function is used to run the simulation. The function will decide the weather data source depending on the data availability. If all the data to reach the end of season is available in the ERA5/PRISM tables, then it will use only ERA5/PRISM data. If the ERA5/PRISM data might not be enough to reach the end of the season, then a combination of ERA5/PRISM and NNME data is used.

In this case, we will be testing a scenario where the data in the ERA5 or PRISM tables is enough to get to the end of the season. The simplified workflow of the function is illustrated in Figure 2. The data workflow in the "Get Weather Series" block differs between end-of-season and in-season runs. In the case of end-of-season, that block would be a single query to the ERA5 tables.

Table 2.6. Details on the function to run the DSSAT model for a defined admin unit.

Function	`dssat.run_spatial_dssat(`
	`  con=pg.extensions.connection,`
	`  schema:str,`
	`  admin1:str,`
	`  plantingdate:datetime,`
	`  cultivar:str,`
	`  nitrogen:list[tuple],`
	`  overview:bool=False,`
	`  nens:int=50,`
	`  all_random:bool=True,`
	`  return_input:bool=False,`
	`  weather_table:str='era5'`

	)
What it does	It runs DSSAT spatially for the *admin1* admin unit in the *schema* region/country. The planting date is defined in the *plantingdate* parameter. The cultivar code is defined in the *cultivar* parameter. The cultivar can be any cultivar in the DSSAT cultivar file. Note that it is not constrained to the cultivars in the cultivar_options table. The nitrogen applications are defined in the *nitrogen* parameter as a list of tuples, where each element in the list is one fertilizer application, and the tuple is a days-after-planting and nitrogen-rate pair. For example, two applications of 20kg/ha of nitrogen at planting and 50 days after planting would be passed as [(0, 20), (50, 20)]. The *nens* parameter is the number of ensembles to consider during the simulation. It can take any value from 1 to 99. When *all_random* is set to True, the weather and soil pixels are randomly sampled (Soil and Weather can be from different pixels). If it is set to *False* then the Soil and Weather pairs are matched to the same pixel. By default the function returns the DSSAT startdad output as DataFrame. This DataFrame contains all the end of season results: final yield, above ground biomass, season length, total precipitation, etc. When the *overview* parameter is set to True, it also returns the environmental stress summary from the OVERVIEW.OUT file as a DataFrame. The *weather_table* parameter defines the weather source to be used (era5 or prism). The *return_input* option is set to True the function won't run the model, instead it will return a list with the input Soil and Weather files.
Comments on the input files	

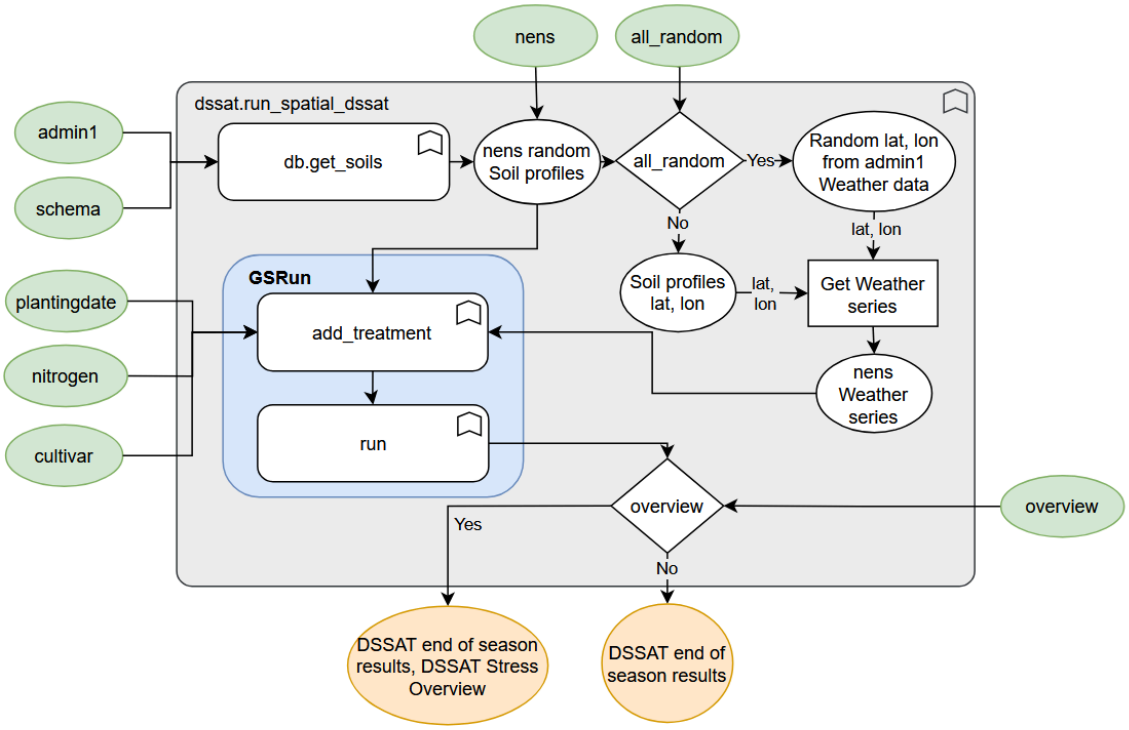


Figure 2. Schematic of the dssat.run_spatial_dssat function.

cultivar_options table (code example)

This table stores the cultivars that will be displayed as an option for each administrative unit in *schema.admin*.

Table 2.7. Details on the function to ingest the cultivar options.

Function	`ing.ingest_cultivars(`
	`  con:pg.extensions.connection,`
	`  schema:str,`
	`  csv:str`
	`)`

What it does	It creates the *schema.cultivar_options* table, and ingests the data from the table at the *csv* path.
Comments on the input files	*csv* is the path to the cultivar options table. It must contain the next columns: admin1, cultivar, maturity_type, and season_length. • admin1: it is the name of the admin unit, it must match the admin1 value for one of the rows of *schema.admin*. • cultivar: DSSAT cultivar code. • maturity_type: the maturity type category. This is what is shown at the web platform. • season_length: the average model season length for that cultivar.

	This list of cultivars are selected after a careful selection. These cultivars are not necessarily the same used for the operation forecast. This will be discussed ahead in this documentation.

nmme tables (code example)

These tables store the NMME climate forecast information. Ingestion is done using the `ingest_nmme` functions. Similar functions must be developed for other climate forecast data sources.

Table 2.8. Details on the function to ingest the NMME climate forecast data.

Function	`ing.ingest_nmme(`
	`  con:pg.extensions.connection,`
	`  schema:str,`
	`  weather_table:str='era5'`
	`)`

What it does	It creates the tables to store the NMME forecast in the *schema* if those don't exist, and then it downloads all ensembles of the latest CCSM4 forecast using the climateserv API. The *weather_table* parameter determines the climatology used to bias adjust the NMME data.
Comments on the input files	

The NMME data available via climateserv only contains mean temperature and precipitation data at 0.5 deg resolution. Therefore, before the model runs that data is internally processed to generate the data needed by DSSAT. This processing includes: bias adjusting and downscaling to 10km; generation of tmax and tmin series assuming tmean = (tmin+tmax)/2 assuming a constant daily temperature range equal to the temperature range from the climatology of that month. This process is described in Figure 2. srad is not saved in the database, instead it is generated when during the model run by using a harmonic series fitted to the previous year data, and an anomaly estimated by a KNN regressor using the precipitation and mean temperature as predictors. This brings a lot of uncertainty to the input, therefore, having a higher quality climate forecast must be a priority in the future development of the service.

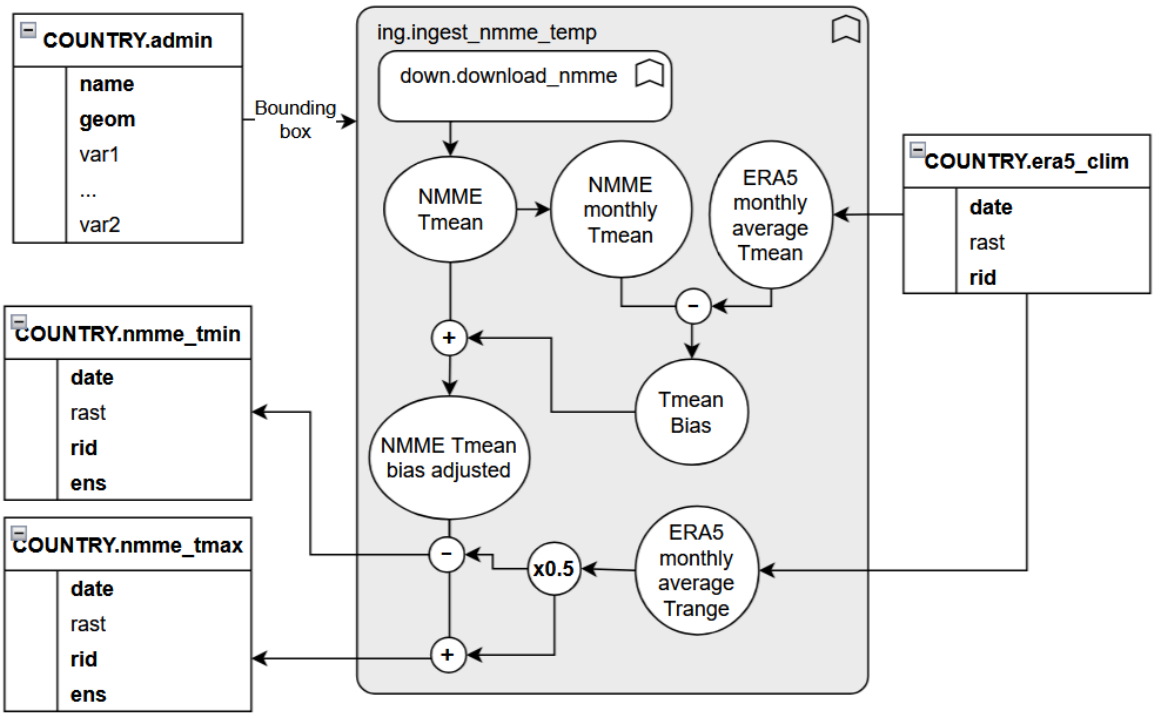


Figure 3. Data pipeline in the `ingest_nmme_temp` function. This function is called by the `ingest_nmme` function.

To complete the tables for the service to be deployed, the forecast tables must be created and populated. Such tables are generated using scripts described ahead in this document. Also, the model must be carefully calibrated before the production of the operational forecasts.

Test in-season run (code example)

At this point all the data needed to run an in-season simulation is available in the table. An in-season simulation is a simulation run when the season hasn't ended, so future weather data will be needed to reach the end of the season. Regardless of the weather data source, the `dssat.run_spatial_dssat` function is used to run the simulation (Table 2.6). The function will decide the weather data source depending on the data availability. If all the data to reach the end of season is available in the ERA5 tables, then it will use only ERA5 data. If the ERA5 data might not be enough to reach the end of the season, then a combination of ERA5 and NNME data is used.

latest_forecast table (code example)

This table contains the results of the latest operational forecast. It includes the geometries and average forecast values of the latest forecast issued. It also contains some values that describe the assumptions to generate the forecast. The `db.add_latest_forecast` function is used to ingest the data in this table from a previously created GeoJSON file. The generation of this file will be discussed ahead in this document (Routine scripts section). This is the table that is directly read by the frontend (the map), therefore, its correct maintenance and update is very important.

Table 2.9. Details on the function to ingest the latest forecast.

Function	`db.add_latest_forecast(`
	`  con:pg.extensions.connection,`
	`  schema:str,`
	`  geojson:str`
	`)`

What it does	It creates the *schema.latest_forecast* table if it does not exist, and populates the table with the latest forecast values from *geojson*.
Comments on the input files	*geojson* is a GeoJSON file that contains the administrative units of the country or domain. They should be the same as the ones in the *schema.admin* table. The file must contain the next properties: • admin1: admin name (same as in *schema.admin*) • pred: predicted yield (kg/ha) • pred_cat: predicted category (Very Low, Low, Normal, High, Very High) • obs_avg: Observed average yield (kg/ha) • obs_std: standard deviation of the observed yield (kg/ha) • obs_min: minimum observed yield value (kg/ha) • obs_max: maximum observed yield value (kg/ha) • planting_p: planting window in months. Example (April - June) • ref_period: reference period with observed data. • nitro_rate: nitrogen rate assumed in the forecast (kg N /ha). • urea_rate: nitrogen rate assumed in the forecast converted to Urea rate (kg Urea/ha) • season_nam: name of the season. For example, short rains, long rains, etc.

latest_forecast_results and latest_forecast_overview tables (code example)

Those tables contain the DSSAT standard output and the Overview stress tables for all the DSSAT runs in that forecast. The two tables are necessary in the yield forecast detail section of the frontend. Each table is ingested with the `db.dataframe_to_table` function. This function creates or replaces the *schema.table* table, and populates it with the data in the dataframe passed to the function. The files used to populate these tables are created when running the forecast. This will be explained ahead in this document.

Table 2.10. Details on the function to ingest the latest_forecast_results and latest_forecast_overview tables.

Function	`db.dataframe_to_table(`
	`  con:pg.extensions.connection,`
	`  df:pandas.DataFrame,`
	`  schema:str,`

	`table:str,`
	`index_label:str`
	`)`

What it does	It creates the *schema.latest_forecast* table if it does not exist, and populates the table with the latest forecast values from *geojson*.
Comments on the input files	

Routine scripts

All scripts needed for the service operation are in this repo. The next table lists the scripts in the order they must be run.

Table 2.10. Routine scripts, frequencies, outputs, and prerequisites.

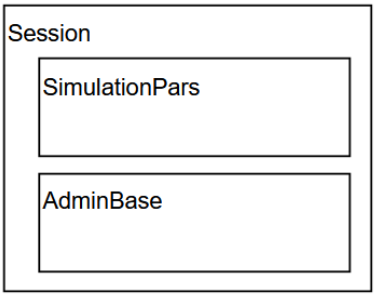
update_weather.py	
Description	Updates the weather data tables if new data is available. It includes both historical data (ERA5) and forecast data (NMME).
Frequency	Weekly
Output	None
Prerequisite	None
run_forecast.py	
Description	Run the forecast using the latest weather data. The program receives three arguments: the country, the season, and the year. For example: `$python run_forecast.py Kenya SHORT_RAINS 2024`
Frequency	Weekly, only if new weather data was ingested that week.
Output	Forecast results (DSSAT standard output) and overview stress table as csv files. These files are the ones used to populate the *schema.latest_forecast_results* and *schema.latest_forecast_overview* tables.
Prerequisite	*update_weather.py* must be run first. All the files and parameters to produce the forecast are defined in the *forecast_params.json* file. This file includes information on geometries, cultivars, nitrogen rates, planting calendars, seasons, etc. The structure of the file, and explanation of the parameters is found on the docstring of the script.
postprocess_forecast.py	
Description	It takes the output of the *run_forecast.py* script and processes it to generate a GeoJSON file with the latest forecast. That GeoJSON file is the one used to populate the *schema.latest_forecast* table. The script is run with three

	arguments; the country name, the season, and the date the latest forecast was produced in the YYYYMMDD format: `$python postprocess_forecast.py Kenya SHORT_RAINS 20241202`
Frequency	Weekly, if a new forecast is produced.
Output	The GeoJSON file that will be used to update the *schema.latest_forecast* table.
Prerequisite	*run_forecast.py*
update_forecast_db.py	
Description	This script updates the forecast tables in the database. The script finds the files created in the previous scripts, and updates the database using those files. The name and location of the files is defined based on the arguments to run the program. The script is run with three arguments; the country name, the season, and the date the latest forecast was produced in the YYYYMMDD format:: `$python update_forecast_db.py Kenya SHORT_RAINS 20241202` When updating the database the program may find issues due to live processes in the database. Then it's better to restart the database service before running this script.
Frequency	Weekly, if a new forecast has been produced
Output	None
Prerequisite	*postprocess_forecast.py* All database processes should be killed. It's better to restart the database service before running the script.

Integrating a user interface

The frontend developer can integrate the package with the user interface as they find it more convenient. One option to integrate the user interface is the `dssatservice.ui` module. This module was designed to handle the user requests in the frontend. The user is expected to run their own simulations using their defined parameters. The `ui.base` module contains the `Session` class. The `Session` class is instantiated once the user has selected the region (admin1) they want to explore. The region and all its data is represented by the `AdminBase` class. Thus, as soon as an `AdminBase` instance is created, the data data for the selected region is loaded into some of the attributes of the `AdminBase` instance. This data includes the reference observed data of the region (max, min, and mean yield), cultivar options, and latest forecast. The simulation parameters defined by the user are stored in a `SimulationPars` instance. The `Session` class has the `run_experiment` method, which runs the model for the selected region and the defined model parameters. The next figure illustrates the dependency between the three classes: The Session class contains the simulation parameters,

which are updated as the user changes their values using the user interface; it also contains the information about the region to be simulated in the adminBase attribute.



The three classes will be described next. This jupyter notebook demonstrates the use of the three classes in a user interface. In the notebook the interface is generated using IPython widgets. In an operative context it would be a web interface.

SimulationPars class

Attributes	
nitrogen_dap: list	A list of integers. As many elements as fertilizer applications/events. Each element represents the day after planting that the fertilization event occurs.
nitrogen_rate: list	A list of floats. It must be as long as nitrogen_dap. Each element is the fertilizer rate (kg/ha) that corresponds to the fertilizer event in nitrogen_dap.
cultivar: str	The DSSAT cultivar code.
planting_date: datetime	The planting date.
irrigation: bool	A bool that defines if irrigation is or not applied during the simulation.

This class is a generic dataclass. It stores the simulation parameters as attributes of the instance. The simulation parameters are tied to the `Session` class as the simPars attribute. Each time the user interacts with the interface and changes the simulation parameters, those are updated in the `Session.simPars` attribute.

AdminBase class

Attributes	
forecast_results: DataFrame	The latest forecast results for the specific region (admin). It is obtained from the latest_forecast_results table.
forecast_overview: DataFrame	The latest forecast overview for the specific region (admin). It is obtained from the latest_forecast_overview

	table.
cultivars: DataFrame	The DSSAT cultivar options for the specific region (admin). The DataFrame contains the cultivar code, maturity type, and season length for that region.
obs_reference: tuple	It is a tuple containing the min, average, and max historical yield for the region.

As soon as the `Session` instance is initialized, the information about the selected region is saved as an `AdminBase` object in the `Session.adminBase` attribute.

Session class

Attributes	
adminBase: AdminBase	The AdminBase instance for the specific region (admin).
simPars: SimulationPars	The simulation parameters selected by the user.
experiment_results: DataFrame	The experiment results. This saves a summary of all the model runs during the session. Each time the model is run, the summary of that simulation will be saved here.
latest_run: DataFrame	The complete results of the latest run.
latest_overview: DataFrame	The complete overview results of the latest run.
Methods	
run_experiment(fakerun: bool)	It runs the model using the current simulation parameters and saves the results in latest_run, and latest_overview. It also updates experiment_results. When fakerun is True, the model does not run. Instead, synthetic data is generated. This could be useful when debugging.

The `Session` class is the one managing all the user interactions. The next box of code illustrates a single on how the class is instantiated as the user select the region, and how the parameters are updated as the user interacts with the frontend:

```
# The user clicks the Uasin Gishu province in Kenya
schema = "kenya"
admin1 = "Uasin Gishu"
session = Session(
    AdminBase(con, schema, admin1)
)
# The user sets 50kg N/ha at planting:
session.simPars.__dict__['nitrogen_dap'] = [0,]
session.simPars.__dict__['nitrogen_rate'] = [50,]
# The user sets the planting date
```

```
session.simPars.__dict__['planting_date'] = datetime(2025, 1, 20)
# The user hits the run simulation button
session.run_experiment(fakerun=False)
```

The `ui.plot` module contains some functions to generate HighChartPlots. The usage of such functions can be explored in the Sample jupyter notebook.

Final notes

- Right now, only ERA5, PRISM and NMME datasets are available for ingestion. NMME is downloaded from Climateserv. To handle new datasets, new functions have to be added to the `dssatservice` package.
- The CDS API was updated by late 2024. After the update, the ERA5 data ingestion is very slow. In the previous CDS API version, small requests were encouraged, so the `dssatservice` was designed that way. This could be a problem during the initial ingestion, but it won't be much of a problem when the system is operative.
- The DSSAT run relies on the spatialDSSAT package. As of now, only Rice and Maize are implemented as an option. You just open an issue in the github repo of the package, and more crops/features could be open on request.