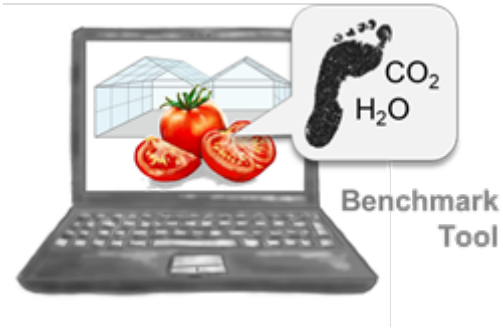
HOCHSCHULE
WEIHENSTEPHAN-TRIESDORF
UNIVERSITY OF APPLIED SCIENCES

Faculty of Horticulture and Food Technology



Maintanence Guide

Benchmark Tool for greenhouse cultivation

Author: Adrian Wild
Date: February 2023

Contents

1 Technology

For the development of the web application, various technologies have been used. For further development, it is crucial to use the same versions for these technologies.

1.1 Back end

The whole back end is written in Python version 3.9. Django was used as the framework for the web application itself. Django is a full-stack Python framework for web applications, which means that it provides functionality for front-end and back-end development. For further information see Django website `https://www.djangoproject.com/`. All packages used by the back end are saved in the *requirements.txt* and *environment.yml* files. The conda and pip package managers are used for package management.

1.1.1 Used packages

django: 4.0.3
djangorestframework: 3.13.1
pandas: 1.4.2
django-rest-knox: 4.2.0
openpyxl: 3.0.9
numpy: 1.22.3
psycop2: 2.9.3

1.2 Front end

The front end is written in TypeScript version 4.6.3. It uses the React framework to build the user interface. The package manager npm is used to include front-end packages in the web application. For the runtime environment, Node.js is used. The used packages are saved in the *package.json* file.

1.2.1 Used packages

@babel/runtime: 7.17.9
@emotion/react: 11.9.0
@emotion/styled: 11.8.1
@mui/icons-material: 5.6.2
@mui/material: 5.6.3
@mui/x-date-pickers: *
@reduxjs/toolkit: 1.8.1
axios: 0.27.2
chart.js: 3.7.1
chartjs-plugin-zoom: 1.2.1
date-fns: 2.29.1
js-cookie: 3.0.1
react-chartjs-2: 4.1.0
react-redux: 8.0.1
react-router-dom: 6.3.0
redux: 4.2.0
redux-thunk: 2.4.1

There are also multiple `devDependencies`, which are packages that are required by the above packages. They can be also found in the *package.json* file.

2 Development Setup

To continue the development of the web application, the project needs to be set up. For this, the following steps need to be done (also described in *README.md*):

2.1 Prerequisites

1. Install Python 3.9:
   - Anaconda Distribution: `https://docs.anaconda.com/anaconda/install/`
   - Standard Python: `https://www.python.org/downloads/`
2. Install Node.js 16:
   - Node.js: `https://nodejs.org/en/download/`

2.2 Getting started

1. Clone the repository.
2. Set up a virtual environment. A virtual environment for python ensures a synchronized development environment. This can be done like this:
   - Download conda: `https://docs.conda.io/projects/conda/en/latest/user-guide/install/download.html`
   - Open a command prompt with conda available on your PATH or an anaconda prompt.
   - Navigate to your project folder.
   - Create the virtual environment using conda: `conda env create --file environment.yml`
   - Make the environment available to your IDE. This step depends on your IDE.
3. Install Node.js dependencies: For this, navigate in a command prompt to the frontend folder and type in:

   ```
   npm install
   ```

4. The development server can be started with this command:

   ```
   cd path/to/project\\
   python manage.py runserver
   ```

   And in a separate command prompt:

   ```
   cd path/to/project/frontend\\
   npm run dev
   ```

3 Backend application

3.1 Structure

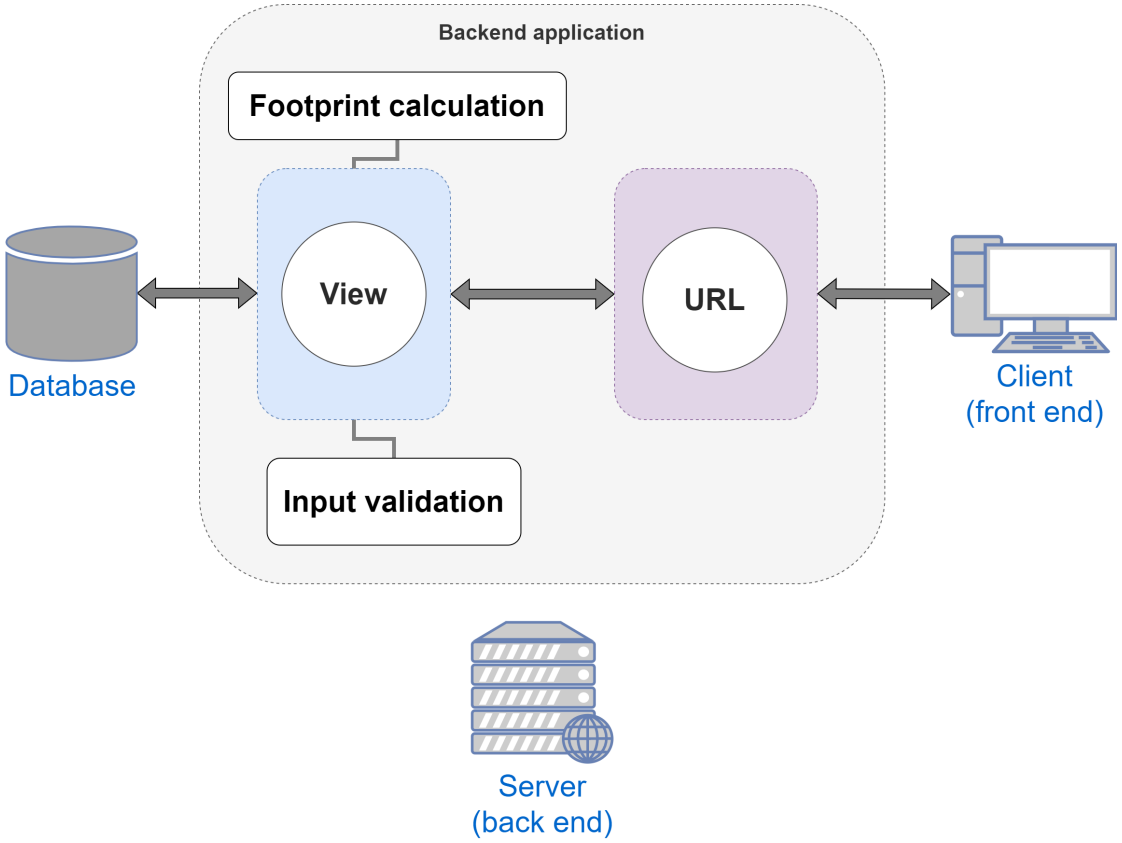


Figure 3.1: Backend application

The above diagram shows the structure of the back end. The view is the heart of the back end. It handles all actions related to greenhouse data. For that, it provides a REST API for the front end. The respective endpoints of the view can be addressed through an HTTP request from the front end. The view accesses external functions to perform its tasks. In addition, the view accesses a database in order to be able to perform its services.

3.2 API

In the following, the communication between the front end and back end will be explained in more detail. All requests arrive in *url.py*, which forwards them via the url-patterns defined there to the corresponding class. These classes are defined in the *backend/api* folder. There, first of all, the class definition is used to check whether the user is authorized to access this interface. The logic is Django-internal, but it can be summarized as comparing the authorization token sent in the request with the authorization tables to see if the token is currently active. If it is, the process is started in the class.

3.2.1 GetCalculatedCO2Footprint

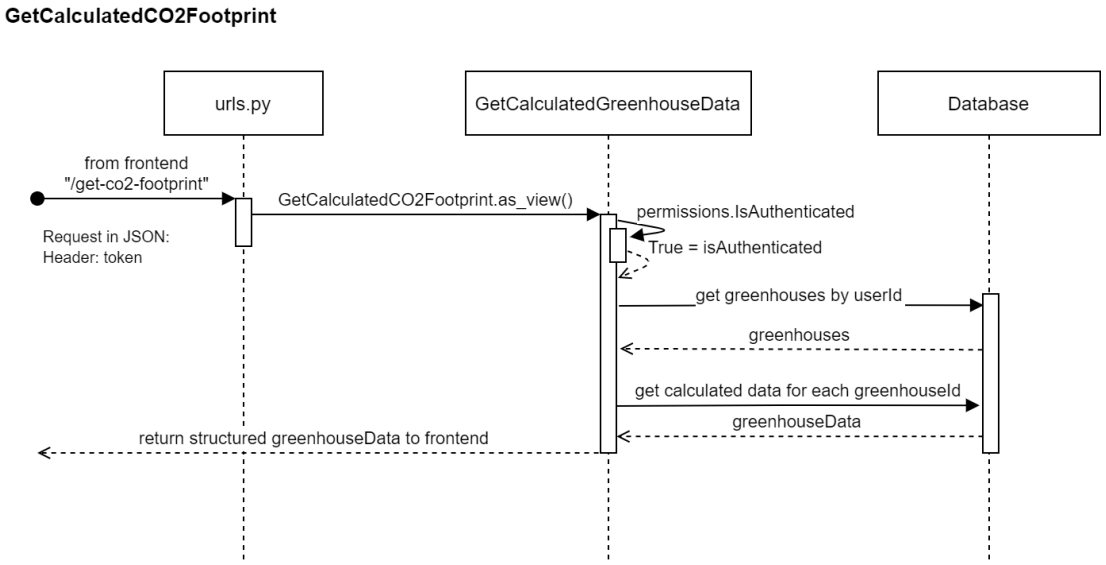


Figure 3.2: GetCalculatedCO2Footprint sequence diagram

This endpoint is there to deliver the calculated CO_2 footprint in a structured JSON format to the front end. From the front end, a request is made to the URL slug /get-co2-footprint. The authorization token of the user has to be included in the header of the request. In the following, at each point where a variable is "checked", the variable is checked for correct and valid content and in case of an error, a corresponding response with code 400 is sent to the front end. This context is implied by "checked" from now on. In the first step, the names and IDs of the calculated values are read from the `Calculations` database table. Then all greenhouses are read from the database table `Greenhouses` via the userId and it is checked whether there exist greenhouses. With the help of these greenhouses, the dictionary structure for the front end is created. All required values, in this case, the `GreenhouseData` and `Results`, are read from the tables and arranged in dictionaries and lists. The JSON string consists of a list of dictionaries. Each dictionary is used to display a plot in the front end.

3.2.2 GetCalculatedH2OFootprint

This endpoint works in analogy to the GetCalculatedCO2Footprint. The difference is that not every data set contains an H_2O footprint and different values are retrieved from the `Results` table. Furthermore, two additional dictionaries for the direct water footprint are generated.

3.2.3 GetOptionGroupValues

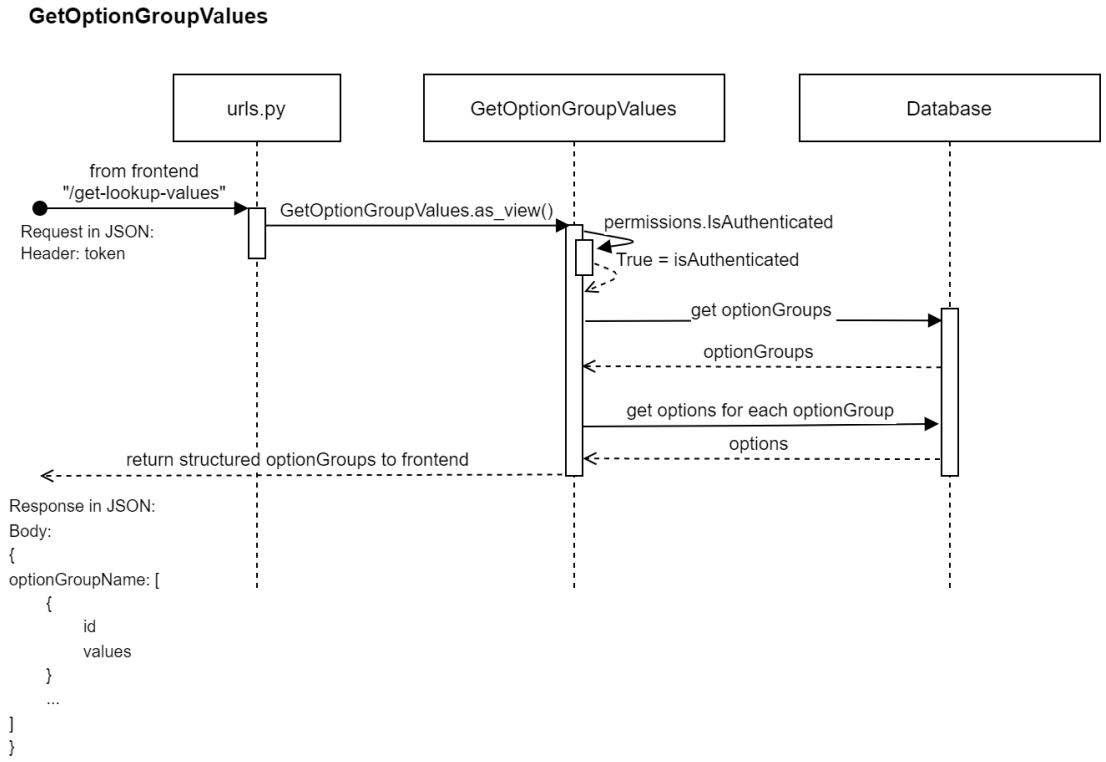


Figure 3.3: GetOptionGroupValues sequence diagram

This endpoint is there to deliver the `Options` of the `OptionGroups` to the front end. They are used on the input page. A request to the URL /get-lookup-values is called from the front end. The authorization token of the user is specified in the header of the request. First, all the `OptionGroups` are read from the corresponding table and checked for existing values. Afterward, with these `OptionGroups`, a structure is built, which is understandable for the front end. In addition, the options are read from the database. The JSON string consists of a list, which contains the individual group objects, which contain the group ID and the respective values of the group.

The structure of the response looks like this:

```
{
    <OptionGroup>: [
            {
                id: <id>,
                value: <value>
            },
            ...
    ],
    ...
}
```

3.2.4 GetUnitValues

Because there are fields where a user can select a unit, there is also a need to deliver those from the database to the front end. The `Options` of an `OptionGroup` can have different units. For example, the `OptionGroup` "Energietraeger" has the `Options` Biogas and Braunkohle. "Biogas" has the units kwh and m3, but "Braunkohle" has the units kwh and kg. Because of that the units are linked to the `Options` and not to the `OptionGroups`. The endpoint works the same as GetOptionGroupValues, but the JSON structure that is generated is a bit different. The structure looks like this:

```
{
    measures: {
        <Measurement>: [
            {
                id: <id>,
                value: <value>
            },
            ...
        ]
    },
    selections: {
        <OptionGroup>: {
            <Option>: [
                {
                    id: <id>,
                    value: <value>
                },
                ...
            ],
            ...
        }
    }

}
```

3.2.5 GetDatasetSummary

This API endpoint delivers metadata about the user's data sets. It is used by the profile page. The structure of the JSON string looks like this:

```
[
    {
        greenhouse_name: <greenhouse_name>,
        data: [
            {
                datasetId: <dataset-id>
                label: <label>,
                co2Footprint: <co2 footprint>,
                h2oFootrpint: <h2o footprint>
            },
            ...
            }
        ]
    }
]
```

3.2.6 GetDatasets

This endpoint is there for retrieving all greenhouse data for every greenhouse of a user. It is used by the input page to automatically fill out the input form with the content of one of the data sets. This also gets done when a user wants to edit a data set on the profile page. The JSON structure looks like this:

```
[
    {
        greenhouse_specs: "[<greenhouse_id>,<greenhouse_name>]",
        greenhouse_datasets: [
            {
                greenhouse_data_id: <dataset-id>,
                <all measures and selections>
            },
            ...
        ]
    },
    ...
]
```

3.2.7 CreateGreenhouseData

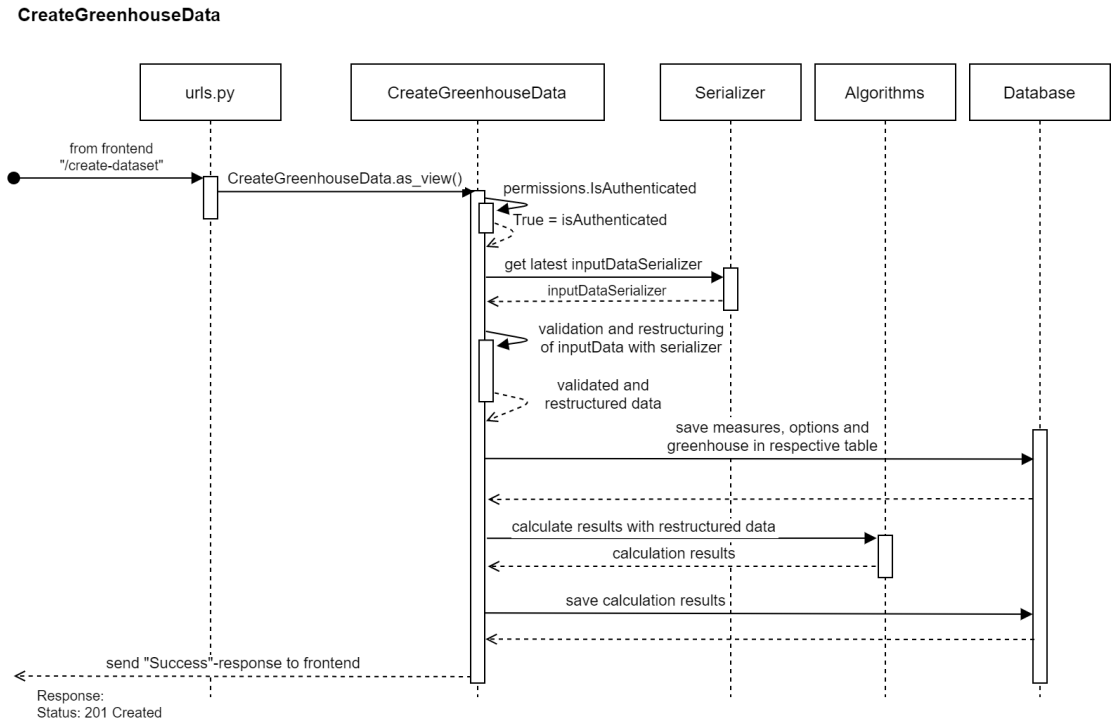


Figure 3.4: CreateGreenhouseData sequence diagram

This endpoint creates a new greenhouse data set in the database.
A request to the URL /create-greenhouse-data is called from the front end. In the header of the request, the `Authorisation` token of the user is included. In addition, a JSON string is provided in the request body, which assigns the `measurement_name` to its input value and the `option_name` to its input value.
At the beginning of the process, the latest version of the `InputDataSerializer` is created. For a more detailed explanation of its structure and operation, see the serializer chapter. Next, the request data is passed to the `InputDataSerializer` so that it can validate the data and transform it into a suitable dictionary structure. If the serializer throws an error, the entire process is aborted and an error response with code 400 is sent to the front end. Once the new structure has been validated, the new data can be saved. For this purpose, a new greenhouse is created, if the name of the greenhouse does not yet exist. Likewise, all measures and selections are saved with their input values in the respective tables. After the input data has been saved, the data is passed on to the calculation algorithms defined in the *calcFootprints.py* file. After the calculations are done, the view receives the results in form of a dictionary and saves them in the database. Finally, a response with Code 201 Created is sent to the front end.

3.2.8 UpdateGreenhouseData

This endpoint updates an existing data set in the database. The request body is the same as the one from the endpoint `CreateGreenhouseData`. The only difference is that in this request the header has an additional parameter called `datasetId`. This is the ID of the to-be-updated data set. After validating the data, the view checks, if the data set belongs to the user. If it does, then it will delete the old data set from the database and save the updated one in it. If it doesn't it returns a 400 error response.

3.3 Serializer

3.3.1 InputDataSerializer

This application uses only one serializer called `InputDataSerializer`, which is only needed by the `CreateGreehouseData` and `UpdateGreenhouseData` endpoints. This has mainly the following 2 tasks:

1. Convert the received input data from the front end into Python objects to be able to process the data in the back end.

2. Validate the received data.

A JSON object contains key/value pairs. The keys of the JSON object, in this case, the names of the measurement data, must be defined as class variables in the serializer. The values are defined as so-called fields and assigned to the corresponding class variable. The serializer then checks whether each key in the transferred JSON object has been made known to it and whether the data type of the associated value is correct. At the same time, the deserialization into a Python object takes place. The decimal variables that are entered in the front end and sent to the back end are stored in the database table `Measures`. To make the serializer flexible, the class variables are set according to the content of the database table `Measurements`. Since a measure consists of a decimal value and a unit, no predefined field can be used. Instead, a special field needs to be created that checks the decimal variables (see next section). The same procedure is used for the categorical measurements. Here, the class variables are set according to the content of the table `OptionGroups`. However, it is difficult to select a suitable field/data type since multiple selections with the respective value must be realized for the categories. Therefore, a separate field had to be implemented as well (see next section). Creating the class variables of the `InputDataSerializer` based on the database content requires that a database exists. When executing the first time

```
> python manage.py makemigrations
```

a database does not yet exist, as it is only migrated when executing

```
> python manage.py migrate
```

With makemigrations, the structure of the classes that are used within the project is created. However, since the structure of the `InputDataSerializer` depends on the database content and this does not yet exist, it would lead to an error here. However, this error can only occur when

makemigrations is executed for the first time, which is why this error is caught with a try/except block. Just before the `InputDataSerializer` is called on a POST request, *serializers.py* is executed again and the `InputDataSerializer` thus gets the class variables according to the database content.

3.3.2 Serializer field ListOfTuples

Fields in Django are defined as classes that inherit from serializers.Field. The field ListOfTuples makes the realization of multiple selections with optional values possible by defining the following data structure: [(<int>,<float>,<int>)|(<int>,<float><int><float>)|(<int>,)|(<int)] A Python list of tuples is expected. The tuples must contain at least one integer. Optionally, they can contain another number of type float and a respective unit ID of type integer. They can also have a fourth number of type float. The first number is the option ID, which references the selected option of the user in the database table `Options`. The optional second number contains the associated quantity (e.g. hydropower: 3000 kWh, wind energy: 2000 KWh, ...). The third number is the option unit ID describing the quantity of the second field. The fourth number is another quantity.

3.3.3 Serializer field Tuple

The field Tuple makes the realization of a measure field possible by defining the following data structure: (<float>,<int>) A Python tuple is expected. The tuple must contain one float value at the first position and one integer value at the second position. The first number contains the associated quantity (e.g. Breite: 30). The second number is the measurement unit ID describing the quantity of the first field.

3.4 Database

The database tables used by the Backend application are defined by a Django model in the file called *backend/models.py*. Every class in this file resembles a table.

3.4.1 Schema

The database model is structured in such a way that, as far as possible, all values are variable and independent of each other. A data set of a greenhouse does not consist of one row in a table, but of several rows of a table, which can all be assigned to one data set via the data set ID. Basically, one can say that a user can have several greenhouses. This greenhouse can have several `GreenhouseData`, which represent the data set. All data set values have a fixed name, which is stored in a separate table for the respective value type. These names are stored in the tables `Calculations`, `Measurements` and `OptionGroups`. The connection between the input values and their names takes place in tables that access the name tables via a foreign key. These tables are `Results`, `Measures`, and `Selections`. The entries in these tables also have a connection to the data set via a foreign key. Of particular note is the structure of `OptionGroups`, `Options`, and `Selections`. `OptionGroups` is a grouping that represents an input field in the front end, for example. However, only fixed values in the form of a dropdown are allowed

in this input field. For this reason, the fixed values were stored in the `Options` table so that they can be assigned to `OptionGroup` and thus to the input field. Some of these options can be given a measurement value, which is why another table was necessary to map this option. Overall, this database model makes it much easier to add options for measurement data collection and to store the calculated values. If, for example, new options have to be added in the dropdown, one simply has to add the new options in Options and assign them to the correct OptionGroup. The units are stored in the tables `MeasurementUnits` and `OptionUnits`. They hold the names of the units and are referenced with a foreign key in the `Measures` and `Selections` tables.

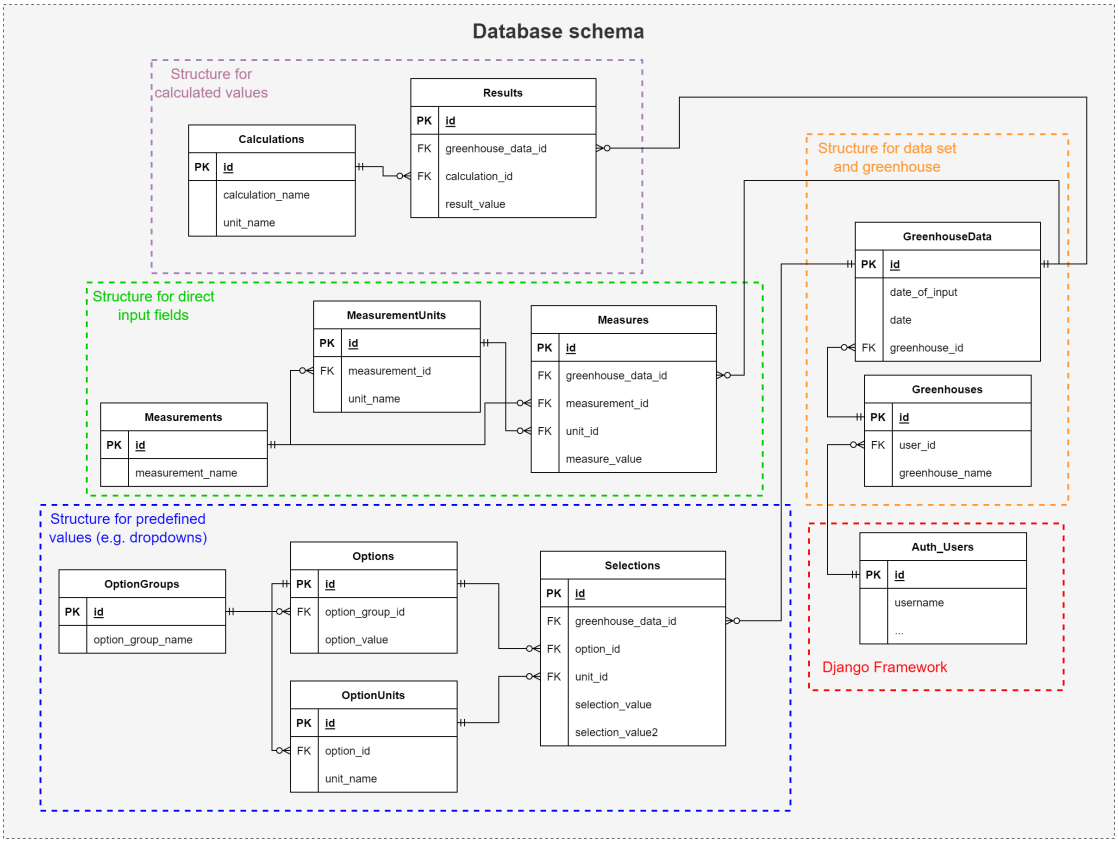


Figure 3.5: Database schema

3.4.2 Fill database

The `Optiongroups`, `Options`, `OptionUnits`, `Measurements`, `MeasurementUnits`, and `Calculations` tables can be automatically filled with a Python script. The script reads the *.csv* files in the *backend/data* folder and saves them into the database. This is only necessary if the database has been wiped. The script for a PostgreSQL database is called: *fillPostgresDatabase.py*. In this script, the credentials for the database need to be adjusted. If an SQLite database is used for developmental purposes, then the script *fillSQLiteDatabase.py* can be used instead.

4 Accounts application

This Django App is also part of the back end and handles all account-related actions.

4.1 Token-Authentication with Django-Rest-Knox

The Python package Django-Rest-Knox is based on the Django Rest Framework and extends the existing authentication functions to be more secure and easier to integrate. The authentication of users is based on tokens, which are sent to the client when logging in. These are kept in an application-internal database table until logout or deleted after a certain time-out period. To be considered authenticated, a client must send the previously received token with a request to the server. The server checks the token for validity and only performs the desired actions if the token is valid. See the user documentation of Django-Rest-Knox for more detailed information: https://james1345.github.io/django-rest-knox

4.2 API

The Accounts application provides a total of eight API endpoints for user authentication and management:

1. auth/user: request user information about the logged-in user (GET request).
2. auth/login: interface for logging in existing users. Expects a username and password in the body of the POST request. Returns user information and the authentication token.
3. auth/register: interface for registering new users. Expects a username, email, password, and site name in the body of the POST request with the following JSON structure:

```
{
    username: MaxMustermann,
    email: MaxMustermann@example.de
    password: MaxPasswortMustermann
    profile: {
        company_name: MusterBetrieb
    }
}
```

   This endpoint sends an email to the user's email address. The email contains a link with 2 URL parameters. One is the Base64 encoded user ID and the other one is a unique token to activate the user's account. The endpoint returns the user information.
4. auth/activate: endpoint for activating a user account. Expects the Base64 encoded user ID as well as a unique token as URL parameters. It sets the .is_active attribute in the Auth_User table for this user to true.

5. auth/forgotpw: endpoint for requesting a password reset email for a user. Like in the registration, this endpoint sends an email to a user's account. That email also contains a user ID and a unique token.

6. auth/resetpw: endpoint for changing the password of a user. Expects the Base64 encoded user ID as well as a unique token as URL parameters. Furthermore, it expects a field called password in the request body. If the token belongs to the user ID, then the password of that user will be changed.

7. auth/delete: endpoint for deleting a user account. If the request comes from a logged-in user, then the user account will be deleted.

8. auth/user: endpoint for sending user information to the front end.

4.3 Secure API endpoints

If an endpoint should be only accessible to authenticated users, then the view needs to be extended with the following code:

```
permission_classes = [
        permissions.IsAuthenticated
]
```

This checks whether a valid authentication token is present in the header of the HTTP request when the view is called. The header must contain the following field, where <auth-token> must be a valid authentication token:

```
Authorization: Token <auth-token>
```

In addition, a so-called CSRF token is required by Django when using POST requests. This is also specified in the header via the following field:

```
X-CSRFToken: <crsf-token>
```

This token does not have to be requested first but is directly integrated into the delivered HTML document of the Django Frontend application. It can be determined there, for example, using the JS-Cookie library and the following code:

```
import Cookies from "js-cookie";
    const csrftoken = Cookies.get('csrftoken');
```

5 Frontend application

5.1 Structure

The Frontend application forms the interface between the back end, which runs on the server, and the user. The Frontend application delivers a simple HTML document and a JavaScript file to the user's browser, which can then provide the necessary functions for data input and visualization. The front end can be roughly divided into four layers, which fulfill different tasks in processing user input. They use appropriate technologies for their purpose:

1. Send and process HTTP requests and responses (red).
2. Application-wide provision of special data at runtime (blue).
3. Application logic and structural design of the website (green).
4. Visualization of data and graphical design of the user interface (yellow/orange).

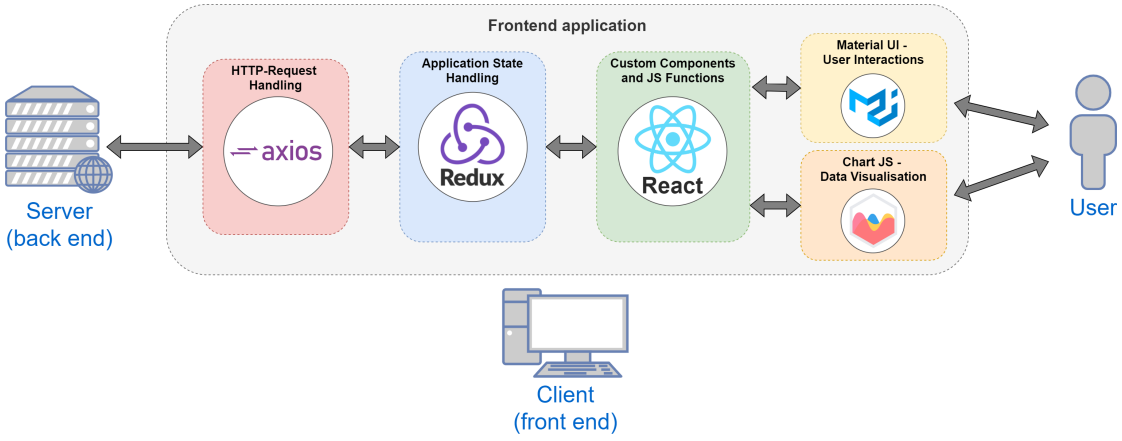


Figure 5.1: Frontend application structure

The user only interacts with the fourth layer, i.e. with the graphical user interface and the components for data visualization (e.g. interactive diagrams). In the case of user input, this layer then calls up functions of the application logic in the third layer, which then triggers the corresponding changes. This includes, for example, requesting data from the server, updating the application state, and sending data to be saved to the server. The third layer also dynamically changes the user interface (e.g. showing or hiding input fields). Any task related to the application state or communication with the server is processed using React Redux (see section 5.2) in layer

two. The structure built on top of React Redux uses Axios (first layer) to make HTTP requests and handle HTTP responses.
The above structure is also reflected in the folder structure of the front end. The diagram below (figure 5.2) shows the dependencies of the individual modules in the front end on each other. The red area covers the structures of layers one and two and the green and yellow areas cover layers three and four. The modules not highlighted in color belong to Django or front-end configuration technologies (e.g. Webpack in the case of bundle.js). The diagram also clearly shows that the actions module alone is responsible for communication with the server.

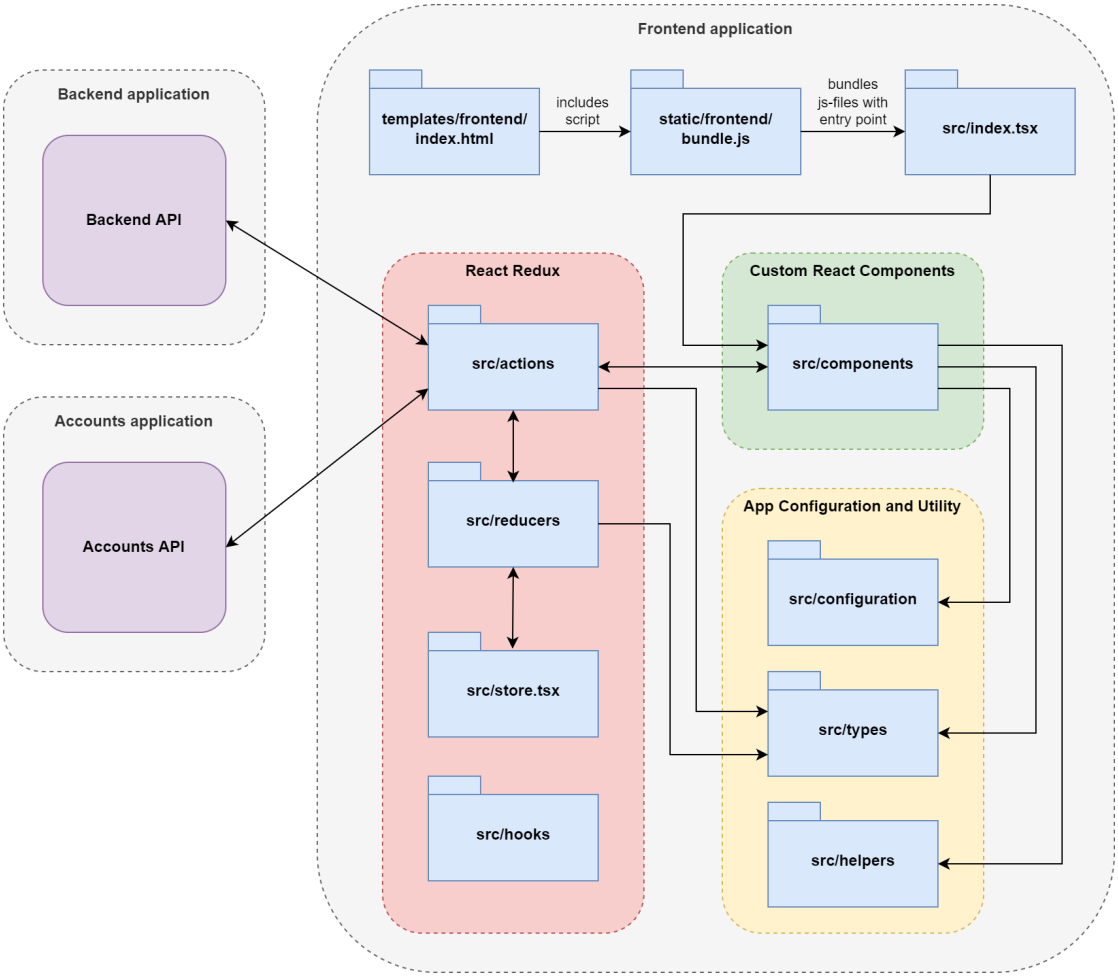


Figure 5.2: Frontend application folder structure

5.2 API calls and React Redux

For communication with the back end, both the back end application and the accounts application provide a corresponding REST API, which is integrated accordingly by the front end. For this purpose, a structure was created using React Redux, which abstracts the individual endpoints from the various application components and also takes over the updating of the application status:

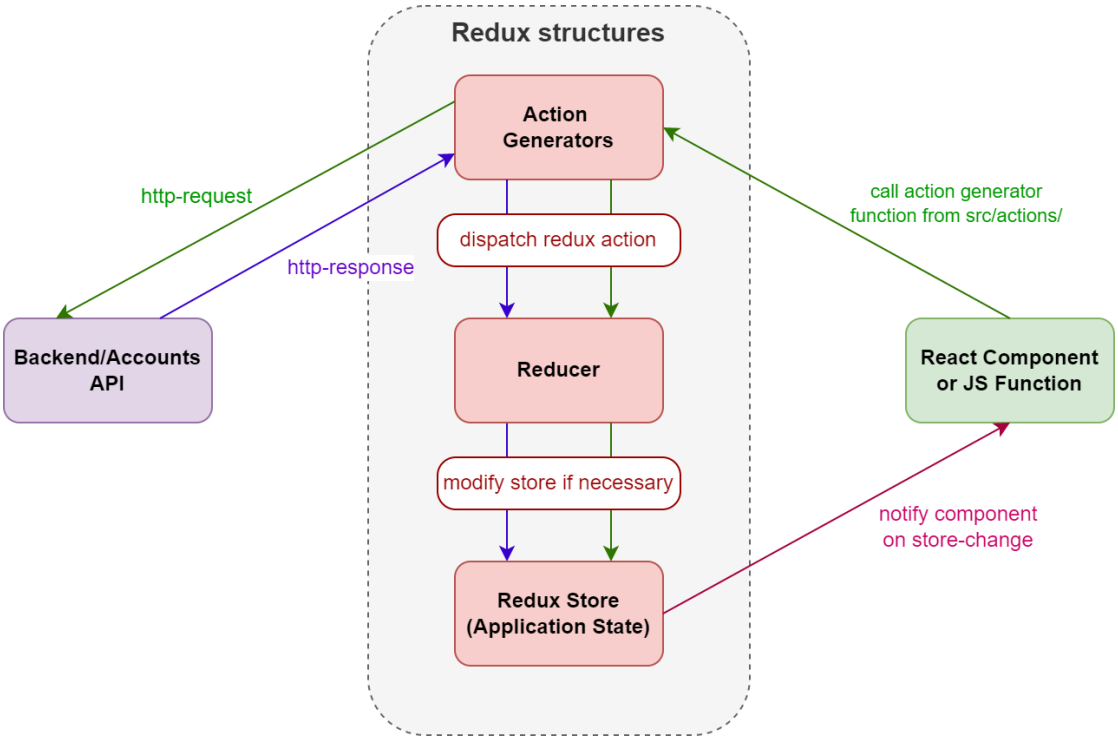


Figure 5.3: Redux flow

The so-called action generators are located in the *src/actions* folder and are responsible for submitting HTTP requests and initiating changes to the state of the application. The reducers, which are defined in the *src/reducers* folder, are responsible for directly changing the state of the application. These are called as soon as a Redux action is to be executed on the Redux store by the action generators. They update the application state (Redux store) based on this. Since this structure means that the actual endpoint calls (API calls) are made in the action generators alone, in the event of an API adaptation, only its integration in the corresponding action generator needs to be adapted so that all components can automatically use the updated endpoint version. For more information on React Redux, see the user documentation: https://redux.js.org/usage/

5.3 Account handling

In order to be able to use the application, a user account is required, which must be created via the registration page. On the page, the company name, email, and password must be entered. If a field is left blank or the passwords entered do not match, a corresponding error message will be displayed. Furthermore, each field has its own restrictions:

- Username: no special characters are allowed.
- Email: no special characters apart from one '@' and multiple '.' are allowed.
- Password: minimum of 8 characters.

A cookie banner also needs to get accepted in order to create an account.
All pages relevant to account management are located in the *src/pages/user* folder. This folder includes these files:

- *PageRegister.tsx*: renders registration page where the user can create his account. The page can be accessed through a link on the login page.
- *PageUserActivation.tsx*: renders a page that is opened when the user clicks on the link in his email to activate his account.
- *PageLogin.tsx*: renders login page. That page calls the login function in the *actions/auth.ts* action generator after the user presses the login button. The action generator calls an endpoint in the back end with the provided credentials and receives the authorization token. The user is logged in once the front end gets that token.
- *PageForgotPW.tsx*: renders forgot password page. That page can be accessed through a link on the login page.
- *PageResetPW.tsx*: renders a page to type in the new password after the user clicked on the password reset link sent to his email.

A so-called authentication token is used to authenticate logged-in users. This is delivered to the front end after successful login and must be sent with every request to secure API endpoints of the back end. For this purpose, the authentication token must be stored in the front end after login for the time of use and must be available application-wide. This is achieved by using the above Redux structure (see section 5.2), whereby the authentication token and other user information are stated in the Redux state after logging in.

5.4 Data input

When a user wants to input data he first gets directed to the `PagePreInputData` component which is located in the *PagePreInputData.tsx* file. There the user selects if he wants to create a new greenhouse or a new data set for an existing greenhouse. If he selects an existing greenhouse, then the newest data set of the selected greenhouse will be transformed into an object of type

`DataToSubmit` and passed to the input page as a prop. If the user creates a new greenhouse, then an empty object of type `DataToSubmit` set will be passed to the input page as a prop.

The data input page consists of the `PageInputData` component, which is located in the *PageInputData.tsx* file. This component contains the state for all subpages and calls the POST endpoint to save all input data. Furthermore, this component fills the state of all subpages with initial values that it received from the `PagePreInputData` component. The individual tabs of the input mask have been moved to their own files, which are located in the *pages/input/subpages* folder. The input fields are rendered in these components. As soon as a value is written into an input field, not only the state of the sub-component is updated, but also the state of the main component with the hook `provide<subpage>()` so that it always contains the current values. All components for the different types of input fields are located in the file *InputFields.tsx*.

5.5 Visualization

The main visualization element is the plot. The footprints are displayed in a variety of bar plots. These are generated with the JavaScript package Chart.js. The actual plotting mechanisms can be found under *src/components/utils/visualization* and are all implemented as functional React components.

5.5.1 CO_2 Footprint

The functional component `PageCO2Footprint` renders the page for the visualization of the CO_2 footprint. It displays a total of four different plots:

1. Total: this is the total footprint value displayed as a stack plot with all categories that a footprint consists of. Each bar represents a data set from one greenhouse. A different greenhouse can be selected with a dropdown field.

2. Normalized: this is the normalized footprint displayed as a stack plot with all categories that a footprint consists of. With a toggle, the mode can be switched between *CO_2 kg / harvest kg* and *CO_2 kg / m2*. This toggle is also included for all the following plots because they all use normalized data. This bar plot displays the data sets of one greenhouse of the user and the best performer data set. This is the data set in the database with the lowest normalized CO_2 footprint.

3. Class specific: this plot shows the CO_2 footprint split up into different weight classes of tomato. It uses the data of the most recent data set of one greenhouse.

4. Benchmark: this plot shows the CO_2 footprint of the most recent data set of one greenhouse split up into its categories. Furthermore, it marks the value of the best and worst performers with a cross in the plot.

The first three plots are generated by using the `FootprintPlot` functional component. The fourth plot is generated by using the `BenchmarkPlot` functional component. The last part of

the visualization of the CO_2 footprint is the optimization tab. Here a progress bar will show the user how close his greenhouse is to the perfect greenhouse. It also shows him how well his greenhouse performs in the specific categories that the footprint consists of. Furthermore, it also displays ways to minimize the CO_2 footprint of the greenhouse to the user. The functional component that creates this visualization is called `CO2FootprintOptimisation` and located in the *utils/visualization/CO2FootprintOptimisation.tsx* file.

5.5.2 H_2O Footprint

The H_2O footprint visualization is mostly the same as the CO_2 footprint visualization. The difference in the plots is that different data is used and additional water-specific categories are displayed. There also exists another plot that works like the normalized plot. The difference is that only the water-specific categories are considered in that visualization.

6 Web app extensions

6.1 Creating new API endpoint

Whenever a new API endpoint needs to be created, it should be stored in a new file in the *backend/api* folder. The back end uses the Django rest framework for the API. Because of that every endpoint class gets an `APIView` object passed as an argument. In the class, a get or post function can be created that will be called depending on the type of request. To return something the `Response` class is used. To make the endpoint only available for logged-in users the following code needs to be in the class:

```
permission_classes = [
    permissions.IsAuthenticated,
]
```

This makes the program check whether or not a valid `Authorization` token is set in the request header. To make the endpoint accessible, it has to be added to the `urlpatterns` list in the *backend/urls.py* file. The path of the endpoint will be defined there. All data that is sent from the back end to the front end will be loaded into the Redux state. To achieve that, the endpoints are always accessed through actions. They are located in the *frontend/src/actions* directory. The actions are functions that call the endpoints to send or request data. The reducers contain the state in which the actions can state the retrieved data. They are located in the *frontend/src/reducers* directory.

6.2 Adding new field to input page

The file *InputFields.tsx* holds all components used in the input page. If the extension needs a new type of input field then the component can be created there. To display a new input field in the web app and send the data of the field to the back end to save it into the database more steps need to be followed. An example best shows this. Adding new measure input field `Greenhouse size`:

1. Add a new row to the *measurements.csv* file containing the ID and input field name. This is the name that will be saved into the database. `120,GreenhouseSize`
2. Since the field has a unit, it needs to be added to another file called *measurementunits.csv*. It contains an ID, the unit name, and the ID of the input field (here 120). The row could look like this: `131,m2,120`. That is it for the back end.
3. Extend the `UnitValues` type so that the unit from the field sent from the back end will be saved in the Redux state. `GreenhouseSize: Option[]`.

4. Extend `GreenhouseType` type. This type contains all fields saved in the database. `GreenhouseSize: string`. The type is always string because the data will be always sent as a string containing tuples.

5. Extend the state of the page where the new input field will be shown. The files are in the pages/input directory. `greenhouseSize: MeasureValue | null`.

6. Now, create the field by creating its properties and adding a new input component with the properties to the return statement of the subpage component. The required properties depend on the component used.

7. To send the data to the back end the `submissionData` variable needs to be extended. The value of the input field will be parsed into a different data structure so that it can be sent to the back end:
   `GreenhouseSize: formatMeasureValue(companyInformation?.greenhouseSize).`

8. Just like the `submissionData` the `testData` in the file *tests/EndpointTest.tsx* needs to be extended the same way.

9. The initial state of the input page is stored in a constant called `emptyDataset` inside the file *helpers/InputHelpers.tsx*. It needs to be updated so that the initial state is the same as the defined state. `greenhouseSize: {value: null, unit: null}`

10. The user can change a created dataset afterward. For that, the data set needs to be loaded from the back end and the data structure needs to be parsed to the data structure used by the state of the input page. A function called `fillInputState` in the same file as above does this and needs to be extended.
    `greenhouseSize: parseMeasureTuple(initialDataset.GreenhouseSize)`

6.3 Adding new page

A new page should be created in the pages folder. There lie all the pages that a user can access through the web app. A page usually consists of a functional react component. To make the new page accessible the `pageDefinitions` Array in the *configuration/PageConfig.tsx* file needs to be extended. It can also be configured if the page should be displayed in one of the drawer sections. If there is a need for a new section, then the `Section` enum in the *types/PageConfigTypes.tsx* file, as well as the `AppDrawer` function in the *layout/AppDrawer.tsx* file needs to be extended.

7 Deployment

The deployment process can be separated into two parts. Only part one needs to be executed if the project has already been set up before. If the project is deployed for the first time on the server or if there are changes to the database schema made, then both parts need to be executed.

Part One:

1. Run the *build.sh* script in the project. This script will build a deployable version of the web app. The output of that script lies in the build folder.
2. Connect to the server and delete force the *benchmark-system-gb.tar.gz* file, the *benchmark-system-gb* folder, and the *static* folder if they exist.
3. Transfer the created *benchmark-system-gb.tar.gz* file and the *static* folder to the server.
4. Run the *deploy.sh* script on the server. Scripts on the server can be executed using python3. This will create the *benchmark-system-gb* folder which contains the application.

Part Two:

5. A new virtual environment needs to be created inside the *benchmark-system-gb* folder. This can be done with these commands:

```
sudo pip3 install virtualenv
virtualenv newenv
```

6. It can happen, that the virtual environment doesn't have enough rights to install packages. If that is the case, then the rights of the folder need to be extended.
7. Activate the virtual environment and install all packages:

```
source newenv/bin/activate
pip3 install -r ./requirements.txt
pip3 list # check if packages got installed
```

8. The next step is to connect the web application with the database:

```
python3 manage.py makemigrations
python3 manage.py migrate
```

9. if the database is empty this command can be used to fill the database with all necessary data.

```
python3 manage.py backend/fillPostgresDatabase.py
```

Start Webserver:

The web server can be started with the following command:

```
pkill -f gunicorn
gunicorn -c config/gunicorn/prod.py
```