Implementation of a Flexible Web Framework for Simulating Python System Models

Demonstrated on Solar Electric Water Heating System Model

In partial fulfillment of the requirements for the degree

Master of Science

at the department of

Electrical and Computer Engineering

of the

Technical University of Munich

Advisor (external)	Dipl.-Ing. Milica Grahovac
	Lawrence Berkeley National Laboratory (California, USA)
Advisor (TUM)	Prof. Dr. rer. nat. Thomas Hamacher
	Chair of Renewable and Sustainable Energy Systems
Submitted by:	Hannes Bohnengel
	110 Channel St.
	San Francisco, CA-94158, USA
Submitted on:	Munich, March 29, 2019

Abstract

In the scope of this Master's Thesis a flexible web framework based on *Django* has been implemented. With this web framework complex, Python-based system models can be deployed through a web browser. This includes the configuration of component parameters, the set-up of input data like weather data or load profiles, the invocation of system simulations, as well as an interactive visualization of the results for further analyses. The benefits of the web framework have been demonstrated on the example of a Solar Water Heating (SWH) system model developed at Lawrence Berkeley National Laboratory (LBNL) in California [1]. As additional research adjacent to the SWH project, the *Solar Electric System* has been implemented within this thesis. The results of this thesis include a simplified approach for conducting system simulations of Python system models, leading to shorter development cycles and a refined method for analyzing simulation results.

Kurzfassung

Im Rahmen dieser Masterarbeit wurde ein flexibles *Web Framework* basierend auf *Django* entwickelt. Mit diesem *Web Framework* ist es möglich komplexe, auf Python basierende Systemmodelle durch einen *Web Browser* einzusetzen. Dies umfasst die Konfiguration von Komponenten-Parametern, die Definition von Eingabedaten wie Wetterdaten oder Lastprofile, das Starten von Systemsimulationen und die interaktive Visualisierung von Simulationsergebnissen für weitere Analysen. Die Vorteile des *Web Frameworks* wurden anhand des Beispiels eines Solar-Warmwasser (*Solar Water Heating* (SWH)) Systemmodels, das am Lawrence Berkeley National Laboratory (LBNL) in Kalifornien [1] entwickelt wurde, demonstriert. Als ergänzende Forschungsarbeit zum SWH Projekt wurde das *Solar Electric System* im Rahmen dieser Arbeit entwickelt. Zu den Ergebnissen dieser Arbeit gehört ein vereinfachter Prozess zur Durchführung von Systemsimulationen von Python Systemmodellen, was zu beschleunigten Entwicklungszyklen und verbesserten Analysemethoden der Simulationsergebnisse führt.

Statement of Academic Integrity

I, *Hannes Bohnengel*, hereby confirm that the attached thesis,

Implementation of a Flexible Web Framework for Simulating Python System Models

was written independently by me without the use of any sources or aids beyond those cited, and all passages and ideas taken from other sources are indicated in the text and given the corresponding citation.

I confirm to respect the "Code of Conduct for Safeguarding Good Academic Practice and Procedures in Cases of Academic Misconduct at Technische Universität München, 2015", as can be read on the website of the Equal Opportunity Office of TUM.

I agree to the further use of my work and its results (including programs produced and methods used) for research and instructional purposes.

I have not previously submitted this thesis for academic credit.

March 29, 2019

Date

[illegible]

Signature (Hannes Bohnengel)

Declaration for the Transfer of the Thesis

I agree to the transfer of this thesis to:

- Students currently or in future writing their thesis at the chair:

  ☐ Flat rate by employees

  ☑ Only after particular prior consultation.

- Present or future employees at the chair:

  ☐ Flat rate by employees

  ☑ Only after particular prior consultation.

My copyright and personal right of use remain unaffected.

March 29, 2019	[illegible]
Date	**Signature** (Hannes Bohnengel)

Contents

1 Introduction

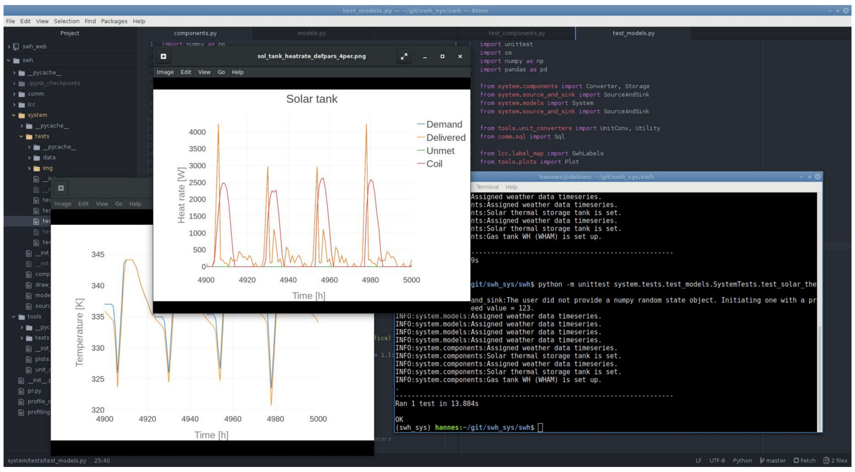
This chapter introduces the topic and provides an overview about the motivation of this Master's Thesis. In addition to that, its structure and contents are explained further.

1.1 Motivation

On June 27, 2018, Python 3.7 was released about 17 years after its very first appearance in February 1991 [2], [3]. As of March 18, 2019 there are 172,467 projects, 1,252,979 releases backed by 312,427 users [4] and according to [5] "*Python has a solid claim to being the fastest-growing major programming language*". Just to name a few of the most significant reasons for this: The clean code syntax with its dynamic type system is leading to a better readability and maintainability, even in large code bases. Due to the fact that Python is an interpreted programming language, the code does not have to be recompiled after every edit, it can just be executed and the effects can be seen immediately. Compared to other general purpose programming languages, Python comes with a rich set of standard libraries which enables the fast development of powerful applications with less lines of code. Lastly, there is a huge community around the extensive number of Python-based open-source projects found in the web covering topics like machine learning, web development, multimedia processing, blockchain, and data science [6].

Although there are various software tools like *Jupyter Notebook* [7] or *PyCharm* [8] available in order to simplify the development of Python projects, usually the command line interface and some kind of text editor are used rather frequently in order to edit parameters and run simulations. This switching back and forth between multiple user interfaces takes the developer unnecessary time.

In Figure 1.1 the current development framework as used in the Solar Water Heating (SWH) project developed by a team at Lawrence Berkeley National Laboratory (LBNL) in California, can be seen [1]. It shows a code editor in the background, a terminal on the right side and two static images with simulation results on the left side. In order to run a simulation with different parameters, the user would have to navigate to the respective file in the code editor, edit the source code, invoke the simulation in the terminal (by typing in a rather long command) and open the generated images showing plots of time series. Especially when doing this iteratively, a lot of time is spent on setting up the simulation. Also, the user needs to be aware where exactly which parameter is defined in the source code and what commands to run different types of simulations.



```
import unittest
import os
import numpy as np
import pandas as pd

from system.components import Converter, Storage
from system.source_and_sink import SourceAndSink
from system.models import System
from system.source_and_sink import SourceAndSink

from tools.unit_converters import UnitConv, Utility
from comm.sql import Sql

from lcc.label_map import SwhLabels
from tools.plots import Plot
```

```
Assigned weather data timeseries.
nts:Assigned weather data timeseries.
nts:Solar thermal storage tank is set.
nts:Assigned weather data timeseries.
nts:Solar thermal storage tank is set.
nts:Gas tank WH (WHAM) is set up.

--------------------------------------------------
9s


git/swh_sys/swh$ python -m unittest system.tests.test_models.SystemTests.test_solar_the

and_sink:The user did not provide a numpy random state object. Initiating one with a pr
eed value = 123.
INFO:system.models:Assigned weather data timeseries.
INFO:system.models:Assigned weather data timeseries.
INFO:system.models:Assigned weather data timeseries.
INFO:system.models:Assigned weather data timeseries.
INFO:system.components:Assigned weather data timeseries.
INFO:system.components:Solar thermal storage tank is set.
INFO:system.components:Assigned weather data timeseries.
INFO:system.components:Solar thermal storage tank is set.
INFO:system.components:Gas tank WH (WHAM) is set up.
.
----------------------------------------------------------------------
Ran 1 test in 13.884s

OK
(swh_sys) hannes:~/git/swh_sys/swh$
```
test_models.py — ~/git/swh_sys/swh — Atom
File Edit View Selection Find Packages Help
Project
components.py
models.py
test_components.py
test_models.py
sol_tank_heatrate_defpars_4per.png
Image Edit View Go Help
Solar tank
Heat rate [W]
Time [h]
Demand
Delivered
Unmet
Coil
Temperature [K]
hannes@debian: ~/git/swh_sys/swh
Terminal Help
LF UTF-8 Python master Fetch 2 files

Figure 1.1: Python SWH modeling framework developed by the team at LBNL [1]

One approach to simplify the mentioned work flow, is to reduce the number of user interfaces to one, the web browser. Figure 1.2 shows a mock-up of a browser-based user interface, where the user can set up a system configuration, invoke its simulation and analyze the results in an interactive way. By enabling this, the user not only saves time, but also the overall work flow is greatly simplified. Compared to the above seen framework, no knowledge of the source code is necessary.

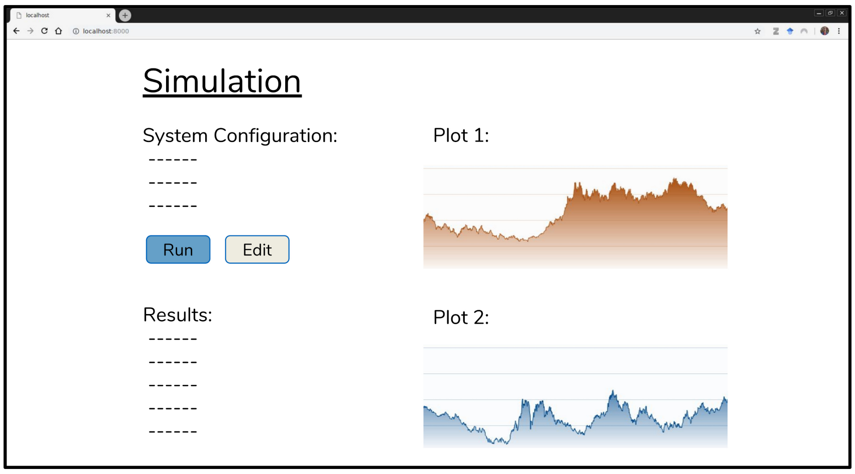


Figure 1.2: Browser based interactive interface (mock-up)

Another big advantage of a browser-based user interface is the option of sharing the access to the system model with or demonstrating it to collaborators or project stakeholders, even across the web, if the application is hosted on a server with access to the Internet.

Since this new browser-based interface could just be plugged into the already existing system model, no alterations to the system code base are necessary. Due to the flexibility of existing open-source web frameworks, it would be possible to use the browser-based user interface during the development of complex Python system models and speed up their development-validation cycle.

1.2 Goals and Requirements

The overall goal of this thesis is to develop a flexible web framework in order to simplify the development of complex Python system models in terms of generating an enhanced usability, reducing the time spent on development-validation iterations and improved analyzability of simulation results. In the following this is broken down into more specific subgoals and requirements of the flexible web framework:

- One requirement is to minimize the interaction with the project code base as much as possible. That means, all system parameters need to be accessible through the front-end of the web framework. Also, the user should be able to start system simulations and see their results without having to switch the user-interface.
- Analyzing simulation results is an important part of the development process, so by using modern web technologies, the web framework has to offer the user an interactive visualization to extract insights about the simulated system's functionality and performance.
- Since the development of complex system models is generally conducted over a longer period of time and system design and requirements change during the project life time, the implemented web framework needs to be adaptable and scalable with no or minimum changes of its source code. This enables the developers to focus on the core aspects of their work.

- By using a web browser as primary user-interface, the usability of the web framework should not depend on the user's platform. This includes the three major operating systems *Windows*, *macOS* and *Linux*. Generally, also mobile platforms like smartphones or tablets are expected to be usable as client of the flexible web framework, but in the scope of this thesis, this is not included as a necessary requirement.
- The web framework should provide a simple user interface, where no detailed knowledge of the incorporated system model is necessary. This way it is possible to use the web framework as a demonstration platform of a Python system model to all collaborators and stakeholders involved in the project.

1.3 Thesis Structure

This thesis has been structured in five chapters. Chapter 1 introduces the topic and the motivation why it is worth being dealt with. The project goals and requirements are defined and a general structure of the thesis is being provided.

In Chapter 2, the author explains the basic functionality of the chosen open-source framework *Django* which is used as a platform to implement the flexible web framework as main part of the thesis. Additionally, the SWH project is being introduced including the project goals, the system modeling architecture and a short overview about how solar billing works in California for customers of the *Pacific Gas and Electric (PGE) Company*.

Chapter 3 provides the details about the model and system implementation of the *Solar Electric System* which has been carried out as additional research adjacent to the SWH project conducted at LBNL [1].

The main work of this thesis is presented in Chapter 4, the implementation of the flexible web framework. First the project structure is introduced before the models are explained in further details. The chapter concludes with providing instructions on how to set up and use the web framework.

Chapter 5 is summarizing the results of this thesis and outlines a list of potential features for consideration during future work on the web framework.

2 Essentials

In this chapter the web framework *Django* [9] is introduced and some insights about its general structure and function are outlined. Furthermore, an overview about the SWH project is provided and it is explained how solar billing works for customers of PGE in California.

2.1 Django Web Framework

Citing directly from the developers [9] "*Django* is a high-level Python Web framework that encourages rapid development and clean, pragmatic design." It belongs to one of the widely used Python-based full-stack web frameworks. As of March 18, 2019 there are 5337 websites based on *Django* [10] among which are popular sites like *Instagram*, *Mozilla*, *Pinterest* and *National Geographic* [9].

Django follows the Model View Controller (MVC) architectural pattern, meaning that a user interfacing system can be divided into three main parts. According to the authors of [11] the model describes the state and behavior of the central components of the application. Views are responsible for "*everything graphical*", i.e. the layer the user is interacting with. The interconnection between a model and a view is defined in a controller, handling data exchange and processing.

Figure 2.1 shows the general structure of a *Django* project consisting of two applications (indicated in green and orange). Every application is built up independently and conceptually follows the MVC paradigm. Though, in *Django* it is referred to as Model Template View (MTV) pattern, where a *Django* view is representing the role of the controller and a *Django* template provides the function of a view according to the original definition [12]. Since *Django* is a web framework with the purpose of building web applications, the browser is the central user interface and next to the template a Uniform Resource Location (URL) dispatcher makes up the front-end layer.

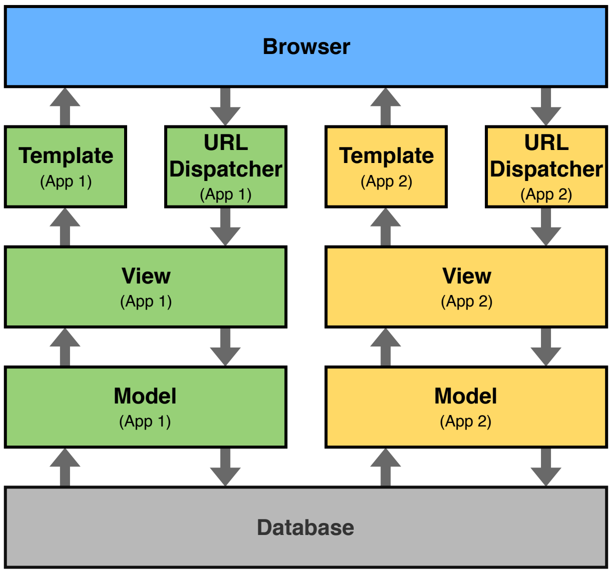


Figure 2.1: *Django* project architecture [13]

To handle a Hypertext Transfer Protocol (HTTP) request on a *Django*-powered site, the URL dispatcher, which is more or less a mapping between URLs and view functions, entirely defined in Python code, derives which view function to invoke [14]. In this view function the required data is retrieved from the model and if it is found in the database, the content of the page is returned by using *Django's* template system [15]. If the content is not found, the HTTP response code "`404 - Not Found`" is returned. *Django's* template system is not required to be used, but offers a powerful way to dynamically generate Hypertext Markup Language (HTML) content, which can be rendered in the browser. This is done, by defining a static part of the HTML content and a dynamically created part by using a special syntax. This way, the final HTML content as rendered in the browser is compiled on-demand by *Django* including the data retrieved from the model [16].

In order to avoid complicated database queries, which vary for different database back-ends, *Django* models offer a unique interface to store data, independently from the used database back-end. By inheriting from the Python class `django.db.models.Model` provided by *Django's* core framework, every *Django* model class has access to specific model fields and functions. Every model object is mapped to a table in the database by *Django's* inbuilt Object Relational Mapper (ORM). This way, the user is able to describe the layout of the database, exclusively by using Python classes and their attributes [17].

2.2 Solar Water Heating Project

2.2.1 Overview

The SWH project is conducted at LBNL in Berkeley, California and has been commissioned by the California Energy Commission (CEC) [1]. At the time of writing it is still under development and the final report has not been published yet. Part of the project is to investigate the benefits of scale for community compared to individual end-use infrastructure of residential SWH systems in California. Scale is meant for both the design of the SWH system, which includes the sizing and the performance parameters of the used components, and the number and size of the supplied households, which is represented by the hourly hot water draw profile. The project outcomes are expected to provide insights on how to optimally deploy SWH systems in California in order to reduce over-all costs and greenhouse gas emissions.

In [18] the authors develop multiple scenarios to decarbonize residential water heating in California and highlight Heat Pump Water Heaters (HPWHs) and solar thermal water heaters as one of the most promising water heating technologies. Both of theses systems have been chosen by the team at LBNL as example SWH systems for being modeled and further analyzed. In the scope of the SWH project (and thus also in this thesis) the system based on a HPWH is being referred to as *Solar Electric System* and is described in more detail in Section 3. The system based on a thermal water heating system is called *Solar Thermal System* and is not being discussed in more detail in this thesis.

2.2.2 System Modeling

In the scope of the SWH project, a Python-based system model has been developed. Similar as in [19], [20], and [21] the system is split into functional components, each modeled through simplified, empirical model equations. These components have been categorized into three classes according to their main purpose. Every component which is converting energy from one form to another belongs to the *Converter* class. The *Storage* class describes components used for storing energy and the *Distribution* class contains components involved in the distribution of energy (e.g. in the form of hot water or electricity). In Table 2.1 an overview of the classes and a representative set of their components is given, where the components which are used to model the *Solar Electric System* are highlighted yellow.

Table 2.1: System component classes (*Solar Electric System* components are highlighted) [1]

Class	**Component**
Converter	Photovoltaic Panel
	Heat Pump
	Solar Collector
	Gas Burner
	Electric Resistance Heater
Storage	Heat Pump Tank
	Solar Thermal Tank
Distribution	Inverter
	Solar Pump
	Distribution Pump
	Piping

The system model is provided with external input data which consists of weather data (containing the solar irradiation, the dry and wet bulb air temperature, and the main inlet water temperature) and the load profile[1] representing the consumer hot water demand. During the system simulation the performance of the chosen SWH system is computed on an hourly basis over the period of one calendar year.

2.2.3 Solar Billing – True-Up Statement

As a customer of PGE, one of the major Investor Owned Utilities (IOU) in California [22] you are entitled to feed back over-produced solar electricity, which has been generated by a Photovoltaic (PV) system on your property, back to the electric grid. In the annual *True-Up* statement the consumed electricity is compared against the generated electricity and the customer only has to pay the difference. In the case more electricity has been

[1]The exact hourly amount of drawn hot water depending on the household size is derived using a model developed by the project team at LBNL [1].

produced than consumed, i.e. more has been fed into than pulled from the utility grid, the customer is compensated for this amount as part of the Net Surplus Compensation (NSC) program. However, the compensation rate per kilowatt-hour is only between 2 and 4 cents, which is determined by the *California Public Utilities Commission* [23]. Compared to the rate in the range of about 13 to 23 cents, customers have to pay per kilowatt-hour (depending on their total annual electricity usage) [24], it is hardly economic to install a PV system which is over-sized with respect to the household's consumption. If the amount of generated electricity stays below the consumed, the customer receives about the same value per kilowatt-hour he would have had to pay for, if pulled from the grid [25], which is much higher than the rate for over-produced electricity. In addition to this, no matter if the customer generates or produces more electricity, monthly minimum delivery charges will have to be paid for being connected to the grid (they will be credited on the annual bill) [26].

In order to benefit from receiving an annual *True-Up* statement, customers of PGE have to be admitted for the *Standard Net Energy Metering (NEM) Interconnection Agreement*, which ensures a "*safe and reliable interconnection to the grid*" [27]. If admitted, a NEM device is installed. This device basically is an electric power meter which is able to spin backwards. That way the amount of generated electricity fed back into the grid is subtracted from the consumed energy and the total net energy amount can be derived.

3 Solar Electric System Implementation

This chapter describes the implementation work representing additional research adjacent to the SWH project conducted at LBNL [1]. Specifically, details about the implementation of the component models as part of the *Solar Electric System* as well as the system-level implementation of the system itself are being explained.

3.1 System Overview

Figure 3.1 shows the system-level structure of the solar electric system. The blue-shaded elements have been implemented in the scope of this thesis and will be further described in the following sections.

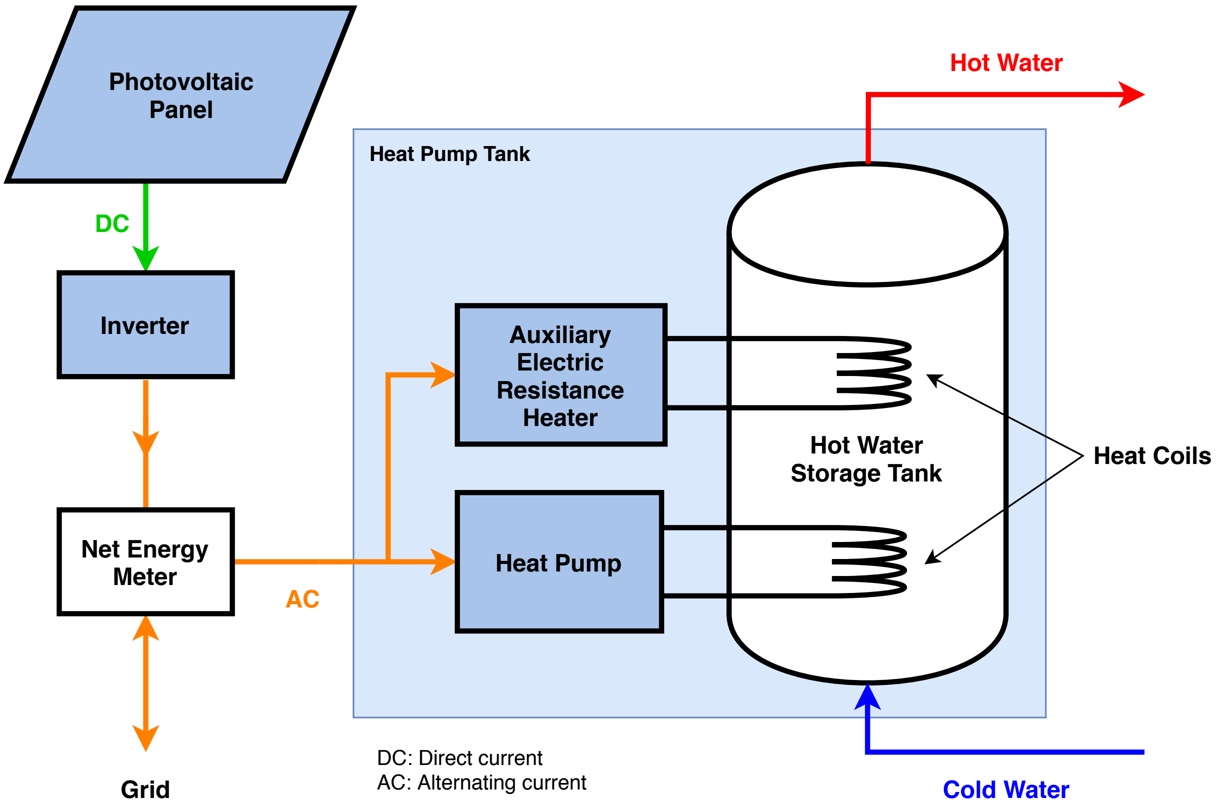


Figure 3.1: Solar electric system structure

The typical operation of the system looks like this: During the day, solar irradiation is absorbed by the photovoltaic panel resulting in the generation of electric energy in the form of Direct Current (DC). To further use this energy it is inverted to Alternating Current (AC) by an inverter. Now it can be used to power typical household appliances like refrigerators or washing machines. Since there might be situations, where the amount of electric power consumption is smaller than the generated, the surplus can be fed into the power supply grid using a NEM device (see Section 2.2.3).

Until here, the system has transferred the energy coming from solar irradiation into electric energy, thus the name *Solar Electric System*. To heat up the water with electricity as efficiently as possible, a HPWH is used. Heat pumps make use of the same principle as refrigerators, but only in the other direction (see Section 3.2.3).

In order to be able to meet peak demands, an auxiliary electric resistance heater is used. This can be either placed in the tank to heat up the stored hot water or deployed as an instantaneous (or through flow) heater before the warm water outlet. However the goal is to minimize the usage of the auxiliary heater to a minimum, due to its significant lower efficiency compared to the heat pump. Usually, HPWHs are provided as a system including a hot water storage tank and a backup electric resistance heater [28].

3.2 Component Models

In the literature, many different models to simulate the performance of the following component models are available, ranging from a high level of detail to more generalized models. Since the focus of this project rather lies on the system-level simulation and the fast exploration of the performance of different system configurations than on highly precise component-level simulation, the models chosen for the used components only capture as much detail as necessary.

3.2.1 Photovoltaic

To calculate the electric power output of a PV module for a given solar irradiation, the model *PVSimple* [29] published in the open-source Modelica Buildings Library [30] has been chosen.

Since the performance parameters of PV modules can be given in a different format either as a combination of the effective panel area $A_{pannel,\,eff}$ and the PV module conversion efficiency η_{pv} or as peak performance $P_{pv,\,peak}$ for a given reference irradiation I_{ref}, Equation 3.2 has been derived from Equation 3.1 by using the relation seen in Equation 3.3 [19, p.35].

$$P_{pv,\,ac} = A_{panel,\,eff} \cdot \eta_{pv} \cdot I \cdot \eta_{dc,\,ac} \tag{3.1}$$

$$P_{pv,\,ac} = \frac{P_{pv,\,peak}}{I_{ref}} \cdot I \cdot \eta_{dc,\,ac} \tag{3.2}$$

$$A_{panel,\,eff} \cdot \eta_{pv} = \frac{P_{pv,\,peak}}{I_{ref}} \tag{3.3}$$

Multiplying either of the mentioned pair of performance parameters with the current total solar irradiation I (the sum of direct and diffuse irradiation, assumed to be perpendicular to the panel surface) and the conversion efficiency of the inverter $\eta_{dc,\,ac}$ results in the electrical power generated by the PV after inversion $P_{pv,\,ac}$.

To validate the accuracy of this model, the results are compared to the *PVWatts* model [31] which is part of a software tool called System Advisor Model (SAM) published by the National Renewable Energy Laboratory (NREL) [32]. SAM is a software tool, to run performance and cost simulations on a collection of renewable energy systems.

First the SAM *PVWatts* model is configured as shown in Table 3.1. To accommodate for different household sizes, the validation is conducted for a PV panel with a rated peak power of 1 kW in case (1) and 4 kW in case (2) (rated at a reference irradiation of 1000 $\frac{W}{m^2}$). Figure A.1 in Appendix A shows a screenshot of the system design view of SAM with all parameters of the *PVWatts* model.

The solar irradiation is retrieved from NREL's climate data set Typical Meteorological Year (TMY3) with San Francisco International Airport as location.

Table 3.1: SAM PVWatts parameters

Parameter	**(1)**	**(2)**
System nameplate size	1 kW	4 kW
Module type	standard	
DC to AC ratio	1.2	
Inverter efficiency	96 %	
Total system losses	14.08 %	

Table 3.2: SWH PVSimple parameters

Parameter	**(1)**	**(2)**
$P_{pv,peak}$	1 kW	4 kW
I_{ref}	1000 $\frac{W}{m^2}$	
$A_{panel,eff}$	6.67 m^2	26.67 m^2
η_{pv}	15 %	
$\eta_{dc,ac}$	82.48 %	

Then, after having set up the *PVWatts* model, the simulation can be invoked starting on January 1 at 00:00 and ending at December 31 at 23:00 of the same year with a resolution of 1 hour. After that, the required results are exported to a CSV file which includes an hourly time stamp, the solar beam irradiation in $\frac{W}{m^2}$ and the resulting output power of the *PVWatts* model in kW for every hour of the whole time period.

Now the performance parameters of the *PVSimple* model need to be set up according to the parameters of the SAM *PVWatts* model. The system nameplate size is equivalent to $P_{pv,peak}$ defined under Standard Test Conditions (STC) which include a reference irradiation of 1000 $\frac{W}{m^2}$. The SAM desktop software offers a help system which provides a detailed documentation of all model parameters, including a description of the different module types for the *PVWatts* model. For the *standard* type a panel efficiency of 15 % can be found there. When using the relation shown in Equation 3.3 the effective panel area $A_{panel,eff}$ can be calculated for both cases (1) and (2). When consulting the SAM documentation the recommended size for the inverter efficiency is 96 % and the total system losses are 14.08 %. A detailed composition of these system losses can be seen in Figure A.1 in Appendix A. To incorporate the same system losses for both models, the inverter efficiency of the *PVSimple* model is defined in a way to include both inverter efficiency and total system losses (see Equation 3.4).

$$\eta_{ac,dc} = 0.96 \cdot (1.0 - 0.1408) = 0.8248 \tag{3.4}$$

After running the simulation of the *PVSimple* model with the given performance parameters and the solar irradiation data imported from the simulation results of the SAM *PVWatts* model, the generated power of both models can be compared. In Figure 3.2 two duration curves are shown and Figure 3.3 visualizes the direct correlation of both models.

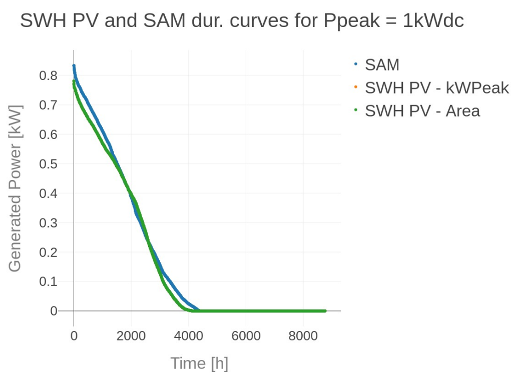


Figure 3.2: Duration curves for case (1) [1]

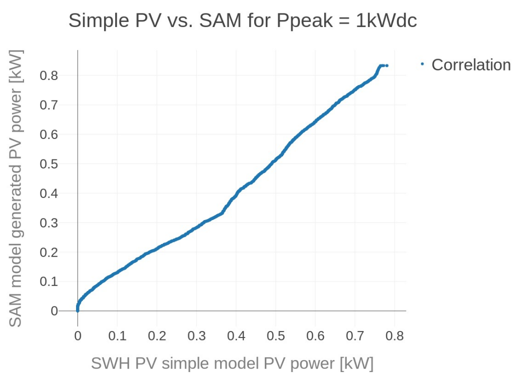


Figure 3.3: Direct correlation for case (1) [1]

The relative error over the cumulative annual sum for both cases is 6.6 %, which is well in an acceptable range, considering the simple design of the *PVSimple* model and the project precision requirements. To optimize this even further, the value of $\eta_{ac,\,dc}$ is adjusted to 85 % which yields in a relative error below 4 %.

In order to choose which combination of performance parameters (either $A_{panel,\,eff}$ and η_{pv} or $P_{pv,\,peak}$ and I_{ref}) is used for the simulation, the Python function of the PV model is being passed a boolean flag called `use_p_peak` as input argument. The function returns both the generated power before and after the DC/AC conversion $P_{pv,\,ac}$ and $P_{pv,\,dc}$. This way, potential DC-powered components can be added to the system and be powered by the PV panel as well.

3.2.2 Inverter

Implementing the inverter as separate component has mainly the reason to stick to a consistent modular system architecture style. Its only performance parameter is the conversion efficiency $\eta_{dc,\,ac}$ which is being defined to incorporate all system losses related to conversion (e.g. cable losses). Equation 3.5 shows the relation between AC and DC power generated by the PV model.

$$P_{pv,\,ac} = P_{pv,\,dc} \cdot \eta_{dc,\,ac} \tag{3.5}$$

3.2.3 Heat Pump

Heat pumps have been around since the first refrigerators hit the market, since they are based on the same working principle. Instead of generating heat, they move it from a warmer space to a cooler and thus are able to operate with efficiencies well above 200 %, also called Coefficient of Performance (COP). Figure 3.4 shows the basic cycle of a heat pump. The evaporator (3) acts as a heat exchanger between the refrigerant and the surrounding air. When entering it, the refrigerant is a low-pressured liquid. As its boiling point is below the temperature of the surrounding air, it evaporates thereby absorbing thermal energy. Then it enters the electrical powered compressor (4) as low-pressured gas. Through the compression process the temperature of the gas rises and it now enters the condenser (1) which respectively acts as a heat exchanger between the refrigerant and the water (or air) which is to be heated. Since the temperature here is lower than the boiling point of the refrigerant, it condenses and releases the stored thermal energy. As cooled, high-pressured liquid it reaches the expansion valve (2) where it passes through as low-pressured liquid and the cycle starts over [33].

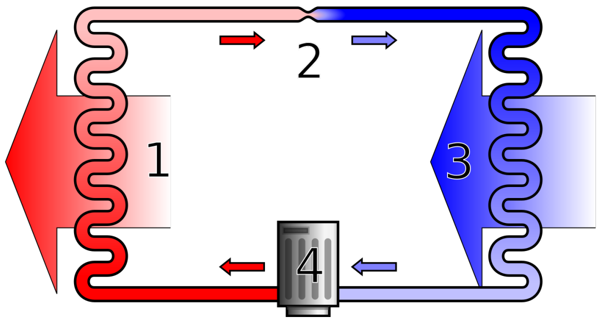


Figure 3.4: Heat pump working principle [34]

For the implementation of the heat pump model used in this project an empirical model presented in [28] has been used. In this report researchers of NREL have evaluated the performance of five HPWH units under different conditions in a laboratory environment. The measured performance data has been used to develop a simulation model using bi-quadratic curve fits. Equation 3.6 shows the calculation of the performance factor $f_{hp,perf}$, for a given wet bulb temperature T_{wb} and average tank water temperature T_{wa}.

$$f_{hp,perf} = C1 + C2 \cdot T_{wb} + C3 \cdot {T_{wb}}^2 + C4 \cdot T_{wa} + C5 \cdot {T_{wa}}^2 + C6 \cdot T_{wa} \tag{3.6}$$

Both temperature values have to be provided in Celsius in order for the equation to work correctly. The coefficients $C1$ through $C6$ are provided in [28] and can be found in Appendix B for either a given rated heating capacity $Q_{hp,rated}$ or rated COP $COP_{hp,rated}$.

In order to calculate the effective heating capacity Q_{hp} or effective COP COP_{hp} of the heat pump, first the performance factor using the respective coefficients (either from Table B.1 or B.2) has to be calculated and then multiplied with the rated performance $Q_{hp,rated}$ or $COP_{hp,rated}$ (retrieved from Table B.3) respectively (see Equation 3.7 and 3.8) [28].

$$Q_{hp} = f_{hp,perf,Q} \cdot Q_{hp,rated} \tag{3.7}$$

$$COP_{hp} = f_{hp,perf,COP} \cdot COP_{hp,rated} \tag{3.8}$$

After having done so, the electric power consumed by the heat pump P_{hp} can be derived, using the relation shown in Equation 3.9 [35]. These calculations are necessary, since the heat pump's performance is strongly dependent on the current wet bulb ambient air temperature and the average tank water temperature and both of these temperatures change continuously through the course of operation.

$$P_{hp} = \frac{Q_{hp}}{COP_{hp}} \tag{3.9}$$

In order to validate the implemented model of the heat pump, its performance over a given range has been plotted as can be seen in Figure 3.5. In this plot the resulting COP_{hp} for Unit A (see Appendix B) has been computed according to Equation 3.6 and 3.8 for different tank and wet bulb temperature values. In the plot, the average tank water temperature T_{wa} is denoted as "T_tank" and the wet bulb temperature T_{wb} as "T_wb". It can be seen, that the resulting COP is highest, for a low tank temperature and a high wet bulb temperature. Specifically it exceeds 6 for $T_{wa} = 20\ °C$ and $T_{wb} = 35\ °C$ and drops to 1 for $T_{wa} = 70\ °C$ and $T_{wb} = 0\ °C$. However, these two edge cases are not representative, since they don't reflect the normal use-case of the heat pump, when deployed for water heating. Assuming that the tank temperature is kept at around 60 $°C$, the COP is ranging between 1.7 and 2.0 for a wet bulb temperature between 10 and 20 $°C$, which can be considered as a typical use-case.

When comparing the performance results plotted in Figure 3.5 with the empirical test results Equation 3.6 is based on (see [28, p.16]), a very high correlation can be noted, which validates the correct functionality of the implemented heat pump model.

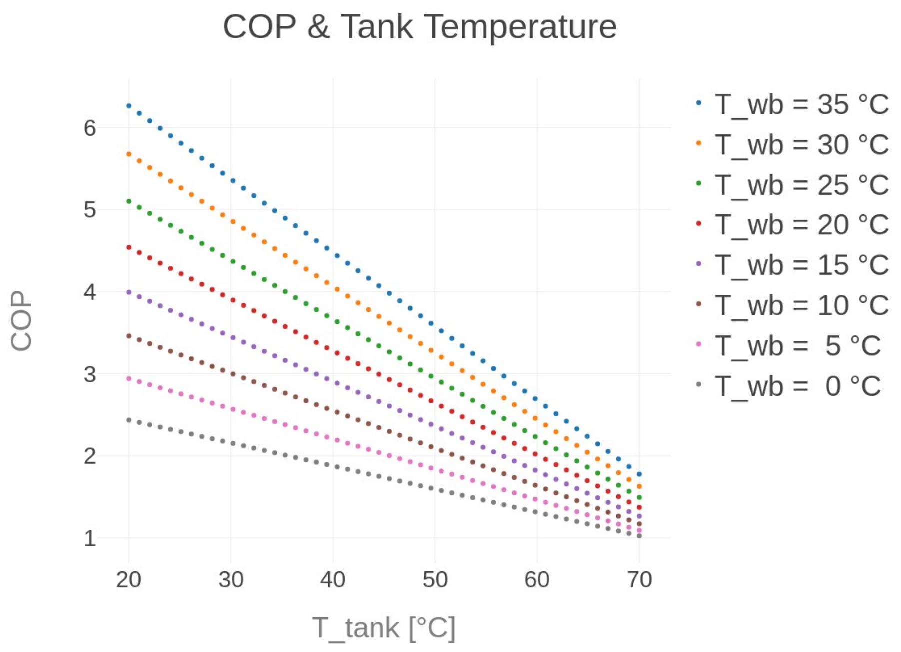


Figure 3.5: Heat pump performance validation

In the scope of the SWH project, two weather data sources can be used, either the TMY3 dataset [36] or one provided by the CEC [37]. Since the heat pump model requires the wet bulb temperature as input parameter and the TMY3 dataset only contains the dry bulb temperature, an approximation provided by [38] has been used to calculate the wet bulb temperature, when using the TMY3 dataset as weather source. The author of [38] is presenting an equation to calculate the wet bulb temperature T_{wb} only depending on the dry bulb ambient air temperature T and the relative air humidity RH[1] (see Equation 3.10). The resulting error in the calculated wet bulb temperature stays below $\pm 1\ °C$ for a valid range of 5 % to 99 % for RH and $-20\ °C$ to $50\ °C$ for T at standard sea level pressure of 101.325 kPa [38].

$$\begin{aligned} T_{wb} &= T \cdot \arctan(0.151977 \cdot (RH + 8.313659)^{1/2}) \\ &+ \arctan(T + RH) - atan(RH - 1.676331) \\ &+ 0.00391838 \cdot RH^{3/2} \cdot \arctan(0.023101 \cdot RH) - 4.686035 \end{aligned} \tag{3.10}$$

[1]The value for the relative air humidity has to be provided as number between 0 and 100, e.g. 61.45% as 61.45

3.2.4 Electric Resistance Heater

Electric resistance heaters (regardless if heating water or air) operate with an efficiency of 100 %. That means all the consumed electric power is converted into thermal energy. But, since the used electricity is usually generated using coal, oil or gas generators which have an efficiency of about 30 % and there are also losses for transporting the electricity to the consumer, the overall efficiency is much lower than the initial, promising 100 % [39].

In the scope of this project the electric resistance heater is merely used as a backup heater, in case the main heating system is not capable of meeting an occurring peak demand. Its performance parameters are the nominal heating capacity $Q_{el,nom}$ and the efficiency η_{el}. The efficiency parameter is mainly implemented as a formality, since as stated before the efficiency is always 100 %. However, if any other system losses would have to be taken into account, this parameter could be adjusted respectively. In Equation 3.11 the calculation of the consumed electricity P_{el} depending on a given nominal heating capacity and conversion efficiency is shown.

$$P_{el} = \frac{Q_{el,nom}}{\eta_{el}} \tag{3.11}$$

The model is implemented in a way, that it is given the current heating demand as input argument and depending on its nominal heating capacity returns the delivered heating capacity, the unmet heating capacity and the used electric power. In the scope of this project, this model can be used both for an in-tank backup heater as well as for an instantaneous through flow heater.

3.2.5 Heat Pump Tank

The heat pump tank is divided into three separate functional components, the heat pump as primary heat source, the electric resistance heater as backup heat source and the hot water storage tank. Both the heat pump and the electric resistance heater have been modeled in the scope of this project (see the proceeding sections). The model of the hot water tank has been implemented by the team at LBNL and has been used for implementing the heat pump tank component [1].

In the tank model basically the current, accumulated heat loss and gain are used to calculate the resulting tank water temperature for the next time step of the simulation. As heat loss the thermal losses of the upper and the lower tank volume are considered as well as the thermal losses of the hot water piping and the loss resulting from taping a defined volume of hot water. The heat gain is provided by the heat pump and the electric resistance heater and can be zero in the case, that the water temperature reaches the defined maximum of the tank. However, this control is implemented on system-level (see Section 3.3) and not in the tank model.

As soon as both total heat loss and gain are defined, they are used to compute the thermal dynamics using a model of the tank implemented by the team at LBNL [1] which returns the new state of the tank.

3.3 Solar Electric System Model

In contrast to the component models the solar electric system model is not simulated for a single time step (by default one hour), but for a whole calendar year. The climate data is provided, as mentioned earlier, either by the TMY3 dataset [36] or by a dataset of the CEC [37], whereas the hot water load profiles are based on research conducted at LBNL [1].

In the implementation of the solar electric system model, first the component models are initialized with their performance parameters, then the climate data and the load profile is used to iteratively calculate the state of the system for each time step. In order to control the status of the heat pump, the control scheme visualized in Figure 3.6 has been implemented.

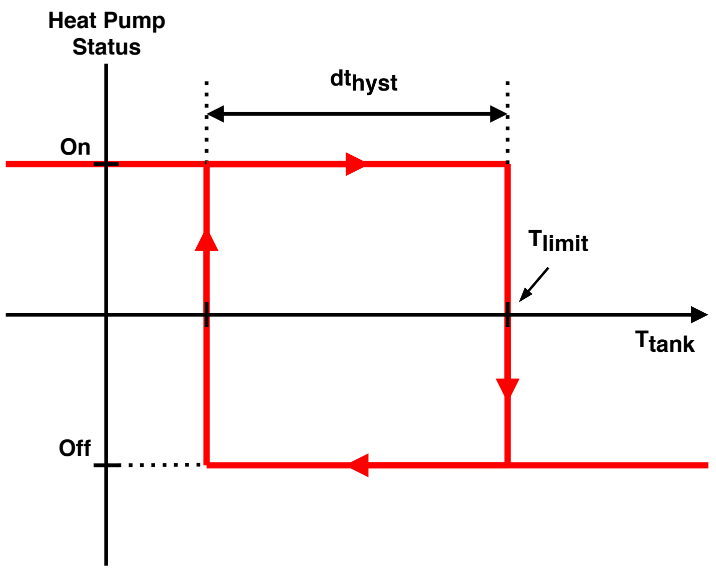


Figure 3.6: Heat pump control cycle

As soon as the tank temperature T_{tank} reaches the defined maximum temperature T_{limit}, the heat pump is switched off, which results in a heat gain delivered to the heat pump tank of zero. While the tank water is cooling off, due to unavoidable heat loss, the heat pump stays off until the water temperature falls below the threshold $T_{limit} - dt_{hyst}$. This way it is ensured, that the dumped heat is minimized and the heat pump is not switched on and off unnecessarily often. In the scope of this project, dt_{hyst} has been set to $5°C$.

After the time series representing the state development have been computed, the array holding the unmet heating capacity is fed into the electric resistance model as heat demand. The electric resistance heater model returns its electricity usage, its delivered heating capacity and the unmet heating capacity. If the nominal heating capacity of the electric resistance heater is not sufficient in order to fulfill its given demand, there remains an unmet heating capacity, which is stored as system-level unmet heating capacity.

The time series hold values either representing heating capacity, generated electric power, and consumed electric power (in *Watt*) or temperature (in *Kelvin*). For the computation of the annual totals, values with the unit *Watt* are summed up which results in annual totals with the unit *Watt-hour* and the temperature values are combined respectively as an average temperature over the whole simulation period.

Since the PV panel may generate electricity when none or less is consumed by the household, there may remain a surplus of electricity which can be fed back into the grid. From the perspective of the grid, the amount fed back into it is the difference between generated and consumed electricity, but in order to calculate a representive solar fraction of the solar electric system, it is essential how the electricity is used. Figure 3.7 shows the logic according to which the generated electricity is allocated to each heat source of the solar electric system and how the remaining surplus is compiled.

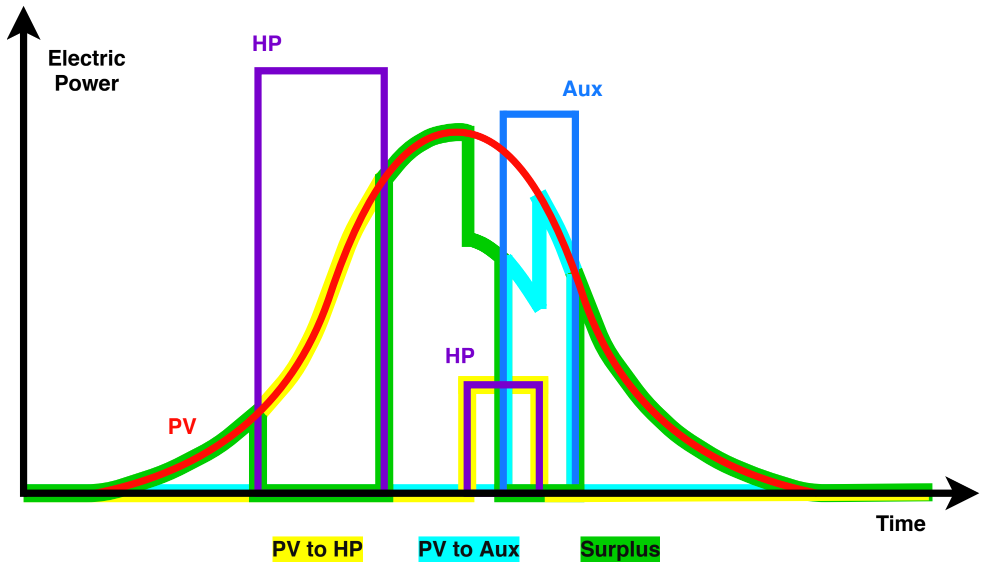


Figure 3.7: Electricity generation, usage and surplus

The red graph in Figure 3.7 labeled "*PV*" represents the electricity generated by the PV panel. The purple graph labeled "*HP*" shows the electricity usage of the heat pump and the blue graph labeled "*Aux*" that of the auxiliary electric resistance heater. Due to its high efficiency the heat pump shall be used as primary heat source. Thus the available solar electricity is initially used to power the heat pump. The yellow shaded segments indicate how much PV generated electricity is used to power the heat pump. If the heat pump consumes more than provided by the PV panel, the remaining amount has to be pulled from the grid. If it is vice versa, there remains a surplus (indicated by the green shaded segments) which can either be fed back into the grid or used to power the auxiliary electric resistance heater (or other electric appliances in the household). For the solar electric system the surplus is only used to power the electric resistance heater. The amount of electricity to power it coming from the PV panel is represented by the turquoise shaded segments in Figure 3.7.

In order to compute the total annual solar fraction of the solar electric system, first the relative amount of PV generated electricity used to power each the heat pump and the electric resistance heater are calculated. This is done by dividing $P_{pv,\,hp,\,total}$ representing the total electricity generated by the PV panel used to power the heat pump by the heat pump's total electricity usage $P_{hp,\,total}$ (see Equation 3.12 and Equation 3.13 for the electric resistance heater respectively).

$$fraction_{pv,\,hp} = \frac{P_{pv,\,hp,\,total}}{P_{hp,\,total}} \qquad (3.12)$$

$$fraction_{pv,\,el} = \frac{P_{pv,\,el,\,total}}{P_{el,\,total}} \qquad (3.13)$$

The total solar fraction $fraction_{solar}$ is representing the relative amount of delivered heat capacity which has been generated by utilizing solar irradiation. To calculate it, the total delivered heat capacity generated by a solar driven heat source is divided by the total heat demand $Q_{dem,\,total}$ (see Equation 3.14). The total delivered heat capacity is the sum of $Q_{hp,\,total} \cdot fraction_{pv,\,hp}$ which represents the total heat capacity generated by the heat pump while being powered by the PV panel and $Q_{el,\,total} \cdot fraction_{pv,\,el}$ for the electric resistance heater respectively.

$$fraction_{solar} = \frac{Q_{hp,\,total} \cdot fraction_{pv,\,hp} + Q_{el,\,total} \cdot fraction_{pv,\,el}}{Q_{dem,\,total}} \qquad (3.14)$$

4 Web Framework Implementation

In this chapter the *Django*-based web framework is described in greater detail. This includes the high-level project architecture, an in-depth outline of its sub-parts as well as the set-up and usage of the framework.

4.1 Project Structure

The project has been developed using *Django's* application-based architecture (see Figure 2.1 on page 6), meaning that all application-specific files are located in the application's folder. By doing so, the potential reuse for other *Django* projects is ensured. The source code has two folders in its root directory. The `doc` folder contains any project related documents, e.g. the file `django-cheatsheet.txt`, which explains the most common *Django* commands. The `web` folder contains the *Django* project itself. In Figure 4.1 the project directory structure inside the `web` folder is shown. This follows the standard *Django* project structure. The name of the project root directory folder `web` is not relevant to *Django*, it can be named arbitrarily. The `web` folder contains the project database `db.sqlite3`, the file `manage.py` which is a *Django* command line utility and five folders called `data`, `static`, `swhweb`, `system`, and `templates`. In the following the purpose and contents of these folders are explained in more detail in order to provide a thorough understanding of the code structure of this project. Folders and files which don't have relevance for this project (e.g. `__pycache__` or `__init__.py`) have not been included in the project directory structure shown in Figure 4.1.

In the folder `data` external databases and pickled files containing simulation results are stored. They are not included in the application folder, since in future use-cases they might be stored on a different drive in the local network or even an external server.

The folder `static` is not tracked by the Version Control System (VCS) `git` and contains all static content of the website (e.g. images and Cascading Style Sheets (CSS) or JavaScript libraries). By running the command

```
$ python manage.py collectstatic
```

all static files stored in application-level folders are collected by *Django* and copied to the `static` root folder. These folders have to be set in the file `swhweb/settings.py`, which is explained later in this section. The reason for keeping all static files in this folder is, that by doing so, it is possible to reference them uniquely from every page of the website by using the relative path `static/`.

```
web
├── data ...................................... Pickled files and external databases
├── static ................................... Local static files (not tracked by git)
├── swhweb .................................................... Project-level files
│   ├── static .............................. Project-level static files (e.g. css or js)
│   │   ├── css
│   │   ├── icon
│   │   └── js
│   ├── local_settings.py .................... Local settings (not tracked by git)
│   ├── settings.py .................................. Project setting parameters
│   ├── urls.py ............................................... Project-level URLs
│   └── views.py .............................................. Project-level views
├── system ................................................ Application-level files
│   ├── migrations ........................ Status of model changes in the database
│   ├── static .......................................... Application-level static files
│   │   └── js
│   ├── templates ...................................... Application-level templates
│   │   └── system
│   │       ├── component ............................. Component related templates
│   │       │   ├── list.html
│   │       │   ├── detail.html
│   │       │   └── edit.html
│   │       ├── configuration ...................... Configuration related templates
│   │       │   └── home.html
│   │       └── visualization ....................... Visualization related templates
│   │           └── home.html
│   ├── admin.py
│   ├── functions.py .......... Helper functions and reference to SWH project code
│   ├── models.py ......................... Model implementation of the application
│   ├── urls.py ......................................... Application-level URLs
│   └── views.py ........................................ Application-level views
├── templates ............................................ Project-level templates
│   └── home.html
├── db.sqlite3 ................................................ Project database
└── manage.py ...................................... Django command line utility
```

Figure 4.1: Project directory structure

The `swhweb` folder holds all project-level configuration files and has been created after running the following command to create the *Django* project:

```
$ django-admin startproject swhweb
```

Furthermore, it contains another `static` folder. Here, all front-end related files are stored in. These are grouped into the folders `css` for CSS libraries, `icon` with the Favicon (the small symbol appearing in a browser's tab when opening a website), and `js` for JavaScript libraries used for the project website. In the scope of this project the open-source front-end component library *Bootstrap* [40] has been used. In order to do so the libraries

- `bootstrap.min.css`
- `jquery-3.3.1.slim.min.js`
- `popper.min.js`
- `bootstrap.min.js`

have to be made available in the respective HTML file. Figure C.1 and C.2 in Appendix C show how to do this either by importing them directly from a Content Delivery Network (CDN), which requires a connection to the Internet during usage, or by importing them locally. For this project, the local import scheme has been chosen, in order to enable a proper functioning website, even for an offline use-case. Therefore, the URLs provided in the CDNs of the libraries have been opened in a browser, and the raw file content has been copied to a local file respectively for every library.

The file `swhweb/settings.py` contains important project configuration parameters, just to name a few of these [41]:

`SECRET_KEY`: Used for cryptographic signing, which among other use-cases is used for setting and reading signed cookies [42].

`INSTALLED_APPS`: A list of *Django* applications used in the *Django* project. Besides the application built in the scope of this thesis, this list contains applications included in the *Django* core framework (e.g. `admin` for using *Django's* administration site).

`DEBUG`: If this is set to `True`, detailed error messages are also displayed in the browser (e.g. a broken URL), which helps for debugging. This should be set to `False` if the project is in production (i.e. available to the end-user)

`DATABASES`: A Python dictionary containing the settings for the used database backend. For this project a SQLite database is used.

`STATIC_ROOT`: This parameter holds the path to the *Django* project's root `static` folder.

`STATIC_FILES_DIRS`: A Python list containing all paths to the `static` folders, both project-level and application-level. From these paths the *Django* `collectstatic` command will look for static files to be copied to the root `static` folder.

In order to exclude the value of `SECRET_KEY` from being stored on the remote project source code repository, the file `local_settings.py` has been created and excluded from being tracked by the VCS `git`. It is imported at the end of `settings.py` and overwrites settings parameters only on this local machine. In case no file named `local_settings.py` is found, the fall-back values in `settings.py` are used.

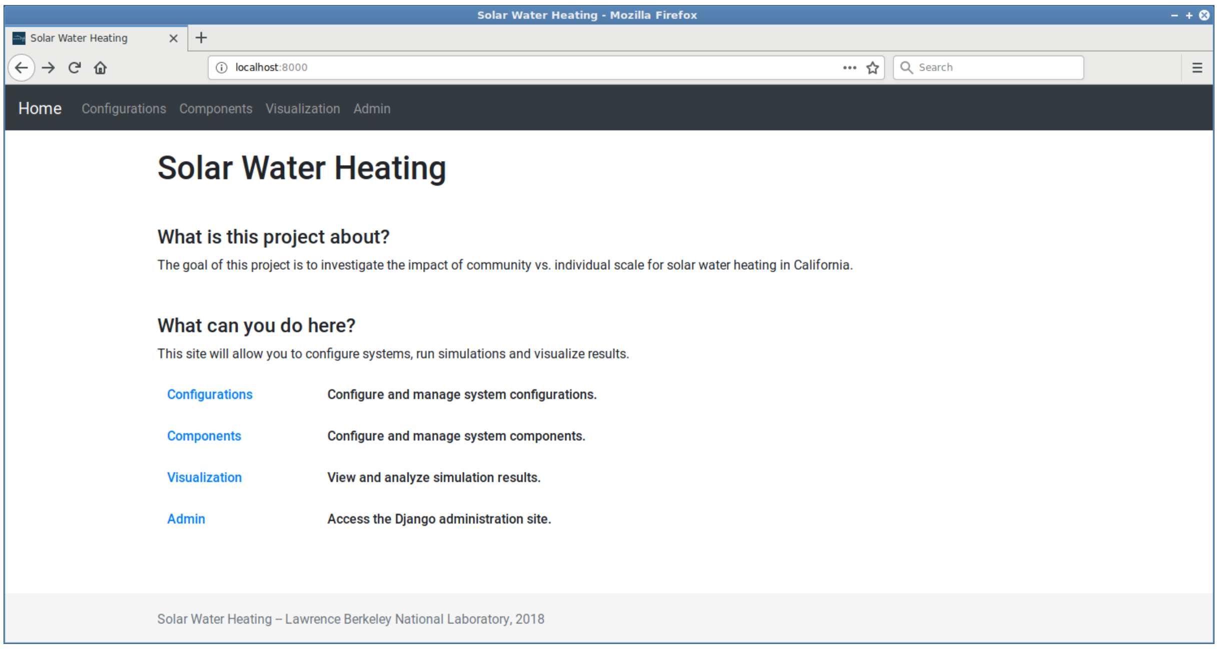
The file `urls.py` can be seen as a list of contents of the website. It routes the entered URLs to the responsible views. In the project-level `urls.py` (stored in the `swhweb` folder) there are only three entries in the Python list `urlpatterns` (see Figure 4.2).

```
1  from django.contrib import admin
2  from django.urls import path, include
3  from . import views
4
5  urlpatterns = [
6      path('admin/', admin.site.urls),
7      path('', views.home, name='home'),
8      path('system/', include('system.urls')),
9  ]
```

Figure 4.2: Code excerpt from `swhweb/urls.py`

During development, the web page is available through `http://localhost:8000` [1] (only on the local machine). If this root URL is entered in the address bar of a browser without appending any path like in line 7 of Figure 4.2, the project-level view `home` is called, which will render the project homepage template (see Figure 4.3). The involved view function is stored in `views.py` in the folder `swhweb` and the template holding the HTML code for the project-level homepage is stored in `home.html` in the folder `templates`.

[1]When running *Windows*, the path `http://127.0.0.1:8000` has to be used.



Solar Water Heating

What is this project about?

The goal of this project is to investigate the impact of community vs. individual scale for solar water heating in California.

What can you do here?

This site will allow you to configure systems, run simulations and visualize results.

Configurations	Configure and manage system configurations.
Components	Configure and manage system components.
Visualization	View and analyze simulation results.
Admin	Access the Django administration site.

Solar Water Heating – Lawrence Berkeley National Laboratory, 2018

Figure 4.3: Landing page of the web framework

The other two paths shown in Figure 4.2 do either direct to the administration site provided by the *Django* core framework, by appending the path `admin` to the root URL like this `http://localhost:8000/admin`, or point towards the application-level `urls.py` file, located in the application folder `system`.

The project front-end makes use of the *Bootstrap* component *Navbar*, which is consistently shown at the top of the page independently what tab the user is navigating to. It contains a link called "*Home*" to the home page on the far left and a link "*Admin*" to the *Django* administration site on the far right. In between the links "*Configurations*", "*Components*", and "*Visualization*" direct to the three application-level pages, explained in Section 4.3.

In the scope of this project the implemented *Django* application has been named "*system*". Respectively, the `system` folder contains all files relevant for the application. The folder `migrations` holds automatically created Python files, which *Django* uses to propagate changes on the models, like the creation of entire models or just renaming a model field, to the project database. See Section 4.3 how to trigger *Django* to create these Python files and how to write the model changes to the database.

As already mentioned the `static` folder contains application-level static content of the website. For the application of this project these are JavaScript libraries and a custom JavaScript function used for visualizing the results (see Section 4.3.5 for more details).

In the `system/templates` folder the application-level *Django* templates can be found organized in respective folders each for the model they belong to. This way, different models can have a template with the same name (e.g. `home.html`).

In order to make a model accessible in *Django's* administration site, it has to be included in the file `admin.py` like shown in Figure 4.4. The admin site provides a powerful tool, it enables authorized users to view, create, delete models and edit their fields. In Appendix D screenshots of the admin homepage, the users configuration page and the model admin pages are illustrated.

```
from django.contrib import admin

# Import the model class
from .models import Component, Configuration, Climate, Household

# Register the model classes for the admin page.
admin.site.register(Component)
admin.site.register(Configuration)
admin.site.register(Climate)
admin.site.register(Household)
```

Figure 4.4: Code excerpt from `system/admin.py`

Besides application-level helper functions, the file `functions.py` contains settings related to the SWH project. In Section 4.2 its contents are described in further detail.

The files `models.py`, `urls.py`, and `views.py` contain the *Django* model implementations, their mapping between URLs and view functions and the implementation of the view functions. All of that is discussed further in the respective subsection in Section 4.3

4.2 Integration of SWH code

In order to keep the code used to build this project's *Django* application separated from content which is specific to the SWH project source code, the file `functions.py` has been created. In here all necessary imports from the SWH project as well as additional helper functions can be found. Besides this, a Python class called `Map` has been created in `functions.py` in order to be used when referring to SWH project source code specific content and classes from within the *Django* application. In Figure 4.5 the member variables of the `Map` class are shown.

```
class Map:
    def __init__(self, components=COMPONENTS, systems=SYSTEMS,
        consumer_labels=CONSUMER_LABELS, system_labels=SYSTEM_LABELS):
        self.components = components
        self.systems = systems
        self._c = consumer_labels
        self._s = system_labels
```

Figure 4.5: Code excerpt from `system/functions.py` (`Map` class member variables)

It can be seen, that the `__init__` function has four default arguments. These are defined just above the `Map` class. The constant `COMPONENTS` has the type of a Python dictionary, with each key representing the component class and each value, being a Python dictionary itself, containing the class implementation and a list of supported component types. The `SYSTEMS` constant has a similar structure. It also is a Python dictionary with each key representing the system type and each value containing the system class implementation, a list of required components and a list of required backup components. In Figure 4.6 both constants can be seen, holding the values not only for the *Solar Electric System* but also for one variant of the *Solar Thermal System*.

The constants `CONSUMER_LABELS` and `SYSTEM_LABELS` are dictionaries imported from the SWH source code containing a mapping between identifiers and human readable names. At the current state of this project, they are only used to provide human readable names of the components, since these are only stored as identifiers, both in `COMPONENTS` and `SYSTEMS`.

```python
# Import swh_sys classes
from swh.system.components import Converter, Storage, Distribution
from swh.system.models import System

# Edit this dictionary to add new components
COMPONENTS = {
    'converter' : { \
        'class' : Converter, \
        'types' : ['hp', 'pv', 'sol_col', 'el_res', 'gas_burn'] \
    },
    'storage' : { \
        'class' : Storage, \
        'types' : ['hp_tank', 'sol_tank'] \
    },
    'distribution' : { \
        'class' : Distribution, \
        'types' : ['inv', 'dist_pump', 'sol_pump', 'piping'] \
    },
}

# Edit this dictionary to add new systems
SYSTEMS = {
    'solar_electric' : { \
        'class' : System, \
        'components' : ['hp', 'pv', 'hp_tank', 'inv', 'piping'], \
        'backup': ['el_res'], \
    },
    'solar_thermal_gas_backup' : { \
        'class' : System, \
        'components' : ['sol_col', 'sol_tank', 'sol_pump', 'piping'], \
        'backup' : ['gas_burn'], \
    },
}
```

Figure 4.6: Code excerpt from `system/functions.py` (components and systems set-up)

Essentially, the purpose of the `Map` class is to provide the content of the `COMPONENTS` and `SYSTEMS` constants in a suitable representation required in different parts of the web framework. As an example, the `Map` class function `get_component_choices` is used to return a Python list with each element being a list itself with the component class name as first element and a list of pairs as second element. These pairs, represented again as a Python list, contain the component identifier and a human readable name of every component associated with the respective component class (see Figure 4.7).

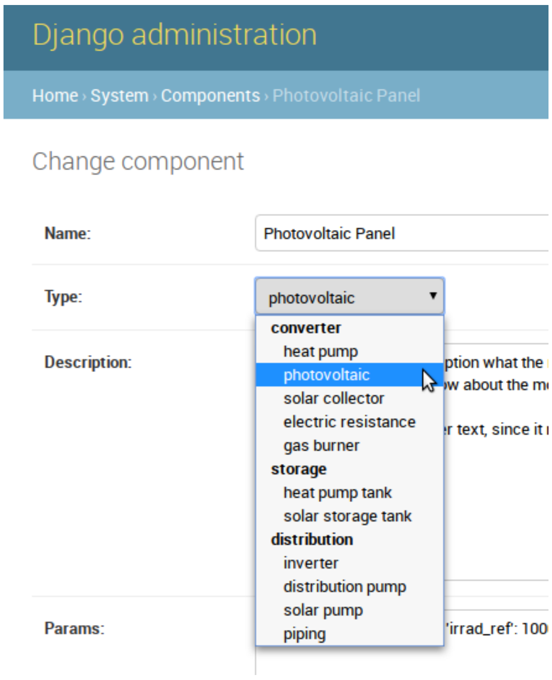
This list is used for providing the `choices` argument for the *Django* model field `type` of the `Component` model class like this:

```python
type = models.CharField(max_length=255, choices=Map().get_component_choices())
```

By doing so, *Django's* administration site of the `Component` model is offering a dropdown menu for the `type` model field as shown in Figure 4.8. When comparing the dropdown menu with the structure of the list shown in Figure 4.7, the same hierarchy can be noticed, according to which each component class has its associated components. For the entries of the dropdown menu, by default the second element of the most inner list is chosen. This is the human readable name of each component retrieved from the label map of the SWH project source code.

```
[ [ 'converter',
    [ ['hp', 'heat pump'],
      ['pv', 'photovoltaic'],
      ['sol_col', 'solar collector'],
      ['el_res', 'electric resistance'],
      ['gas_burn', 'gas burner'] ]
  ],
  [ 'storage',
    [ ['hp_tank', 'heat pump tank'],
      ['sol_tank', 'solar storage tank'] ]
  ],
  [ 'distribution',
    [ ['inv', 'inverter'],
      ['dist_pump', 'distribution pump'],
      ['sol_pump', 'solar pump'],
      ['piping', 'piping'] ]
  ] ]
```

Figure 4.7: Output of `Map` class function `get_component_choices`



Django administration

Home › System › Components › Photovoltaic Panel

Change component

Name: Photovoltaic Panel

Type: photovoltaic

Description: ption what the / w about the m / r text, since it

Params: 'irrad_ref': 100

Figure 4.8: Type choices for `Component` model in *Django* admin page

The advantage of using the `Map` class functions as shown with the example of the function `get_component_choices` earlier in this section, is that the contents of the constants `COMPONENTS` and `SYSTEMS` are not stored in an instance of the `Map` class. Instead, every time the class functions are called, the constants are used as the default arguments in the `__init__` function, so if a new component is added there, it will be reflected in real-time in the web framework without having to restart the *Django* framework. This is ensured by calling the `Map` class functions like this `Map().<class_function()>`, instead of like this `<Map_instance>.<class_function()>`.

4.3 Models

As described in Section 2.1, *Django* uses an ORM to access and describe the database through Python classes, in the *Django* scope referred to as models. In the scope of this project, four models have been created (in the file `system/models.py`), which can be seen in Figure 4.9.

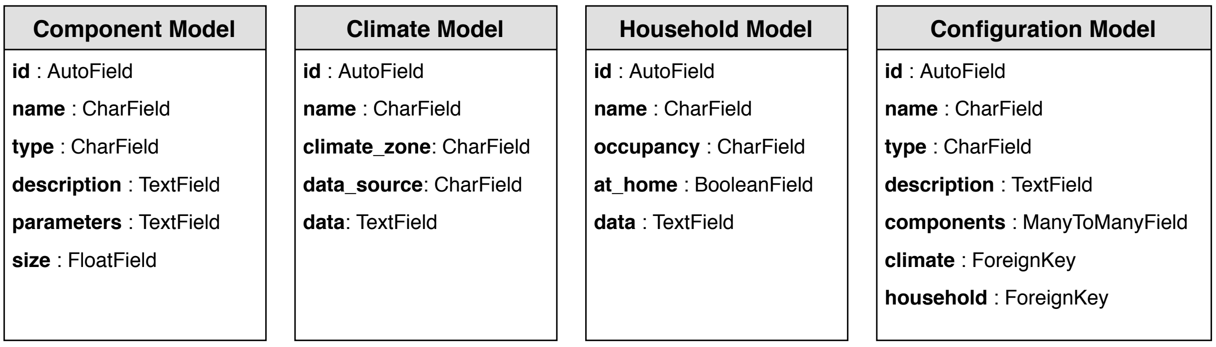


Figure 4.9: *Django* models and their attributes

All of these models are Python classes inheriting from `django.db.models.Model`, which gives them access to *Django* specific model fields. When using one of these fields, *Django* knows to store this attribute in the database. By default every model has an `id` field, which has the type `AutoField`, an automatically increasing Integer used as primary key in the database table (which does not have to be explicitly set in the Python class). In this project, every model has been given a `name` field of the type `CharField` to hold a human readable name of the respective model instance. The purpose of the other fields is explained in the following sections respectively for each model.

When running the following command in a terminal while being in the project's root directory (the `web` folder), *Django* checks if there are any changes to the models.

```
$ python manage.py makemigrations
```

If so, the changes are indicated in the terminal output and *Django* specifig Python files are created in the folder `system/migrations`. These contain all changes to the database and are executed when running the following command. This will result in the migration of the changes to the database.

```
$ python manage.py migrate
```

Every time, a model or one of its attributes, which has the type of a *Django* model field, is created, deleted, or renamed, these two commands have to be executed, in order to migrate the changes to the *Django* project database.

Each model has its own manager class, which can be used to access, edit or delete model instances in a user-defined way. In Figure 4.10 the manager class of the component model can be seen (in lines 4 - 9). It inherits from `django.db.models.Manager` and has a function `create_component()` which instantiates and returns a new object of the `Component` class.

```
1  from django.db import models
2
3  # A manager class for the Component class
4  class ComponentManager(models.Manager):
5
6      # Create a new component object with given parameters
7      def create_component(self, name, type):
8          component = self.create(name=name, type=type)
9          return component
10
11 # Component class
12 class Component(models.Model):
13
14     # Name of the model
15     name = models.CharField(max_length=255, default='undefined')
16
17     # Model manager class
18     objects = ComponentManager()
```

Figure 4.10: Code excerpt from `system/models.py`

Lines 12 - 18 in Figure 4.10 show an excerpt of the `Component` class with its `name` field and the reference to its model manager class. In order to create a new `Component` object the following Python code can be used:

```
new_comp = Component.objects.create_component(name='name', type='type')
```

The above shown example of the `Component` class is equivalent for the other model classes and just demonstrates the structure and usage.

4.3.1 Component

As shown in Table 2.1 on page 8, the SWH system components are grouped into three classes which are `Converter`, `Storage` and `Distribution`. When navigating to the "*Components*" tab, the `comp_list` view is invoked which creates a list of all component types of each component class with a human readable name for each type as set up in the `Map` class in line 10 of Figure 4.11 and retrieves all `Component` objects stored in the database in line 13. It then returns a `HTTPResponse` object containing the respective template (in this example the file `list.html`) and the previously gathered dynamic data as Python dictionary.

```python
 1 from django.shortcuts import render
 2 
 3 from .models import Component
 4 from .functions import Map
 5 
 6 # Show the components home page
 7 def comp_list(request):
 8 
 9     # Get list of component types for each Component class
10     types = Map().get_component_choices()
11 
12     # Retrieve a list of all Component objects from the DB
13     components = Component.objects.all()
14 
15     # Return html page including some dynamic data
16     return render(request,
17                   'system/components/list.html',
18                   {'components' : components, 'types' : types})
```

Figure 4.11: Code excerpt from `system/views.py`

Then, by using *Django's* built-in template language it is possible to dynamically create the actual HTML file which is rendered in the browser. In Figure 4.12 the usage of the template language is shown conceptually for the `list.html` template building the *Components* page. To use constructs like `for` loops or `if` statements, the construct is enframed like this `{% <construct> %}` where `<construct>` has to be replaced with the respective key word. To use a variable passed from the view function, the variable name has to be enframed like this `{{ <variable> }}`, where `<variable>` represents the name of the variable. The *Django* template language is a very powerful tool and also provides features like filters, comments and template inheritance, just to name a few [43].

```
{% for class, values in types %}
   > Create a card for the component class
   > Populate title of card header using {{ class }}
   > Create dropdown menu in card header using {{ values }}
   > Populate card body with components of current component class
   {% for component in components %}
      > Check if current component belongs to current component class
          by comparing {{ component.type }} with entries of {{ values }}
      > If yes, create link for current component using {{ component.id }}
          and {{ component.name }}
   {% endfor %}
{% endfor %}
```

Figure 4.12: Conceptual usage of *Django's* template language in `system/component/list.html`

In the outer `for` loop the current component class (stored in the variable `class`) and its associated list of components (stored in the variable `values`) is retrieved from the `types` list (see Figure 4.7 on page 31) which has been forwarded from the `list` view. Then the value of `class` is used as a title for the card header. The variable `values` contains the list of components for the current component class and can be used to create the dropdown menu, which only contains the components of the current component class. Then the card body is populated with a list of components belonging to the current component class by cycling through the `components` list. This list contains the actual `Component` objects found in the database and is passed along by the `comp_list` view. That means the template has access to the object fields `id`, `type`, and `name`. By comparing the `type` field with each entry of `values` it can be checked if the current `Component` model object belongs to the current component class. If so, the link is created by using the `name` field as link text and the `id` field to create the HTML-native `href` attribute which points to the target URL of the link.

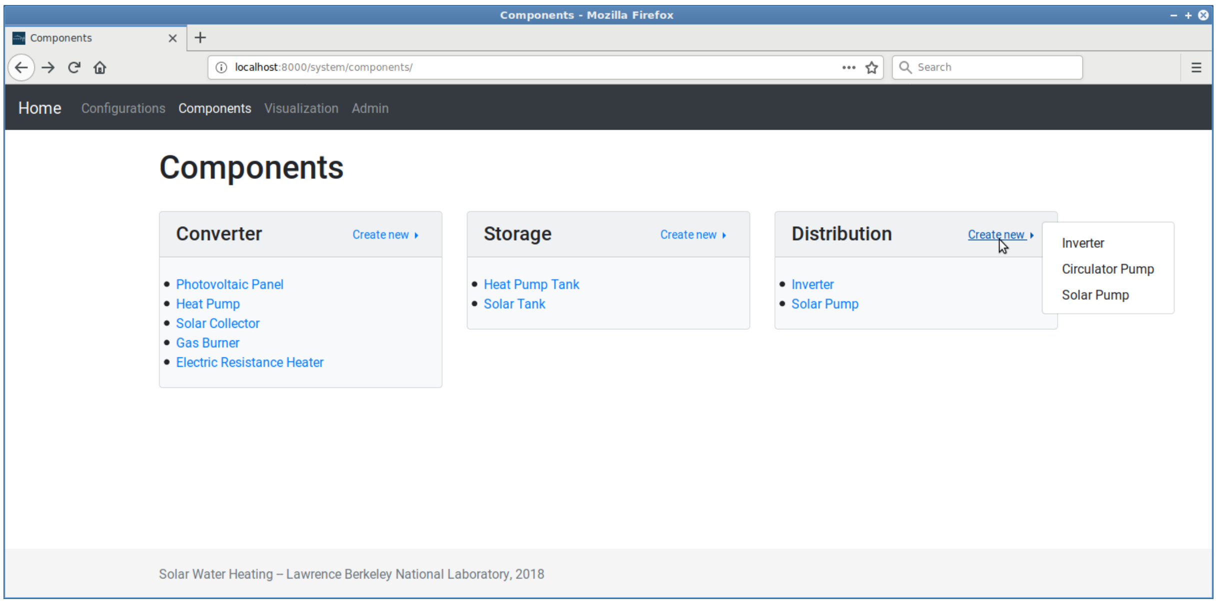
So the *Django* template system translates this line

```
<a href="{% url 'sys:comp_detail' component.id %}">{{ component.name }}</a>
```

to the following, for a given component with the `type` "pv", the `name` "*Photovoltaic Panel*" and the `id` *4*, which can be shown, when selecting the function "View Page Source" after right-clicking on the rendered page in the browser. By using the `url` tag and the name of the view function `'sys:comp_detail'`, *Django* uses the structure defined in the file `system/urls.py` to create the actual URL (see Table 4.1 on page 38).

```
<a href="/system/component/4/">Photovoltaic Panel</a>
```

Figure 4.13 illustrates the resulting web page showing all components categorized in the respective component class. When clicking on "*Create New*" in the card header, as mentioned, a dropdown menu is shown, within only the components are available, which belong to the respective component class.



Components

Converter Create new

- Photovoltaic Panel
- Heat Pump
- Solar Collector
- Gas Burner
- Electric Resistance Heater

Storage Create new

- Heat Pump Tank
- Solar Tank

Distribution Create new

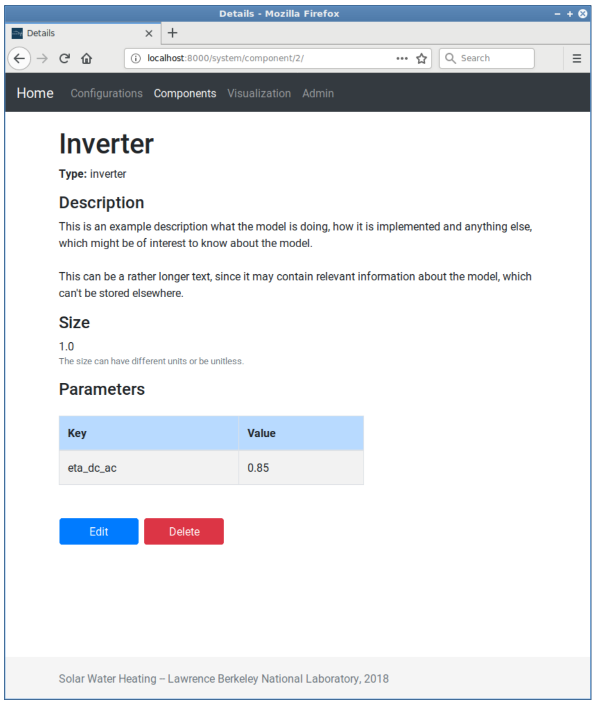
- Inverter
- Solar Pump

Inverter
Circulator Pump
Solar Pump

Solar Water Heating – Lawrence Berkeley National Laboratory, 2018

Figure 4.13: Rendered `list.html` template

By clicking on the component name, the `comp_detail` view of the respective component is called, by directing to the generated URL, like mentioned earlier. In the view, the `Component` object is retrieved from the database by querying for its `id` in order to read all the model attributes. These are then sent to the `detail.html` template, where the details of the component are rendered as shown in Figure 4.14.



Inverter

Type: inverter

Description

This is an example description what the model is doing, how it is implemented and anything else, which might be of interest to know about the model.

This can be a rather longer text, since it may contain relevant information about the model, which can't be stored elsewhere.

Size

1.0

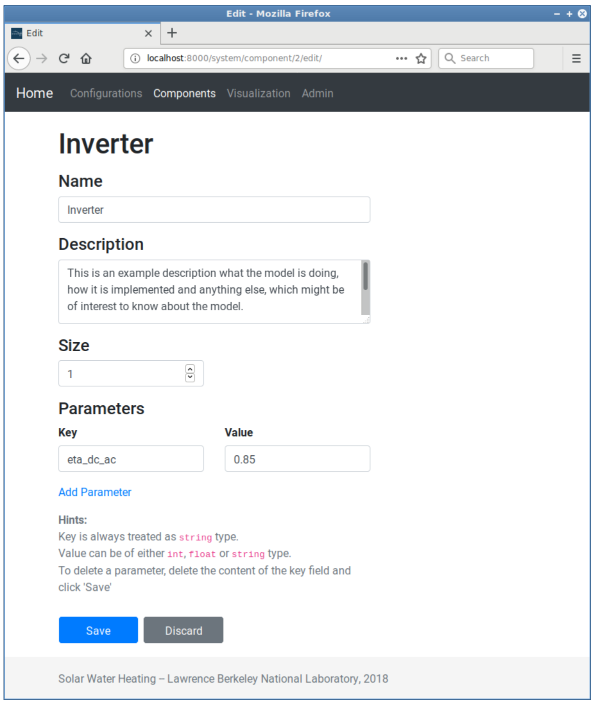
The size can have different units or be unitless.

Parameters

Key	Value
eta_dc_ac	0.85

Solar Water Heating – Lawrence Berkeley National Laboratory, 2018

Figure 4.14: `detail.html` template



Inverter

Name

Inverter

Description

This is an example description what the model is doing, how it is implemented and anything else, which might be of interest to know about the model.

Size

1

Parameters

Key	Value
eta_dc_ac	0.85

Hints:
Key is always treated as `string` type.
Value can be of either `int`, `float` or `string` type.
To delete a parameter, delete the content of the key field and click 'Save'

Solar Water Heating – Lawrence Berkeley National Laboratory, 2018

Figure 4.15: `edit.html` template

The `detail.html` template provides two buttons at the bottom of the page *Edit* and *Delete*. The *Edit* button calls the `comp_edit` view function which does nothing else than retrieving the respective `Component` object from the database and forwarding it to to the `comp_edit` template (shown in Figure 4.15). By clicking the *Delete* button, the current component can be deleted after confirming it in an appearing pop-up. This invokes the view function `comp_delete` which deletes the current `Component` object from the database and redirects to the `list.hml` template.

In the `edit.html` template, the user can edit all model fields except the `type`. All form fields need to contain a value, before clicking on *Save*, otherwise the page is refreshed and a warning is displayed at the top of the page. One exception is if the user wants to delete a parameter. To do so, the *Key* field has to be left empty and after clicking on *Save* the parameter will be deleted. The *Save* button creates a HTTP POST request containing the values of the form fields and calls the `comp_form` template which processes the new values and stores them in the database if they are valid before redirecting to the `detail.html` page. If a value is invalid, the view renders the `edit.html` template and displays the respective warning. A warning message is displayed either if a form field is empty, contains only white spaces or if two or more parameter keys hold the same value.

For adding a new parameter, the user has to click the "*Add Parameter*" action link, which invokes the `comp_add_par` view function and refreshes the page, now with an additional parameter listed. In the `comp_add_par` view, it is checked if the string "*param*" does already exist as parameter key, if yes "*param_1*" (then "*param_2*" and so on) is checked, until it is not present any more. This way it is ensured that the user can add multiple new parameters without having to change the value of the "*Key*" field each time.

Table 4.1 shows an overview of all `Component` views, their relative URLs, and the template they are directing to after they finished execution. What can be seen here, is that not all views have their own template, which is due to the fact, that a view can not only be used to render a new template, but also for data processing while staying on the same page.

Table 4.1: Component views-to-URL mapping

View	**URL**	**Target template**
`comp_list`	`component/`	`list.html`
`comp_detail`	`component/<component_id>/`	`detail.html`
`comp_edit`	`component/<component_id>/edit/`	`edit.html`
`comp_form`	`component/<component_id>/form/`	`detail.html`
`comp_add_par`	`component/<component_id>/add_par/`	`edit.html`
`comp_create`	`component/<component_type>/create/`	`list.html`
`comp_delete`	`component/<component_id>/delete/`	`list.html`

4.3.2 Climate

For the SWH project two climate data sources can be used, the TMY3 dataset [36] and a dataset provided by the CEC [37]. Both can be found in a SQLite database stored in a single file[2] in the folder `data`. The datasets have been preprocessed by the team at LBNL in order to ensure compatibility in terms of scientific units and valid numerical values. As mentioned in the end of Section 3.2.3, the TMY3 dataset initially does not contain the wet bulb temperature and thus these values are approximated based on the work of [38] and appended to the dataset. Additionally as part of the preprocessing, the main water temperature has been added to the datasets.

[2]This file has been created by the team at LBNL [1].

Table 4.2 shows all the climate zones available in California as provided by the two mentioned datasets. The CEC enumerator is just an arbitrary number in which order the climate zones are stored in the database. The TMY3 code is the identifier of the climate zone as defined by NREL.

Table 4.2: Available climate zones in California

Location	**CEC enumerator**	**TMY3 code**
Arcata, CA	01	725945
Santa Rosa, CA	02	724957
Oakland, CA	03	724930
San Jose, CA	04	724945
Santa Maria, CA	05	723940
Torrance, CA	06	722970
San Diego, CA	07	722900
Fullerton, CA	08	722976
Burbank, CA	09	722880
Riverside, CA	10	722869
Red Bluff, CA	11	725910
Sacramento, CA	12	724830
Fresno, CA	13	723890
Palmdale, CA	14	723820
Palm Springs, CA	15	722868
Blue Canyon, CA	16	725845

In order to enable the user to select a climate zone to be used for the system simulation, a `Climate` model has been implemented as part of the web framework. This also ensures, that the climate data is included in the *Django* project database. As the other models, the `Climate` model also has an `id` and `name` field. Additionally, it has a `climate_zone` field which either holds the CEC enumerator or the TMY3 code and the field `data_source` indicating the used climate dataset. The field `data` holds the climate data associated with the location as a string formated Python dictionary.

Since the typical use-case does not include any alterations of the climate data, but only the selection which climate to use for the system simulation, the `Climate` model only has one view function called `climate_populate`. As the name suggests, its purpose is to populate `Climate` model objects in the *Django* project database by reading from the SQLite database provided by the team at LBNL containing the two climate datasets.

4.3.3 Household

Similar to the `Cimate` model the sole purpose of the `Household` model is to hold data which the user can select from for the system simulation. In the scope of the SWH project a consumer load model has been developed by the team at LBNL, which takes the number of occupants per household and a flag indicating if the occupants are at home during the day as input. These input parameters are stored in the `Household` model fields `occupants` and `at_home`. The hourly hot water draw amount is stored in the field `data` as string formatted Python list.

Like the `Climate` model, the `Houshold` model only has one view function which is called `household_populate`. It reads the load profiles from the same database the climate data is stored in and according to the provided number of occupants and `at_home` flag it creates a `Household` model object[3] which is stored in the *Django* project database. To configure which different types of households are available to choose from in the web framework, the constant `HOUSEHOLDS` has been created in the file `system/functions.py`, as shown in Figure 4.16.

```
HOUSEHOLDS = { \
    'Single person household'              : {'occupancy': 1, 'at_home': False},
    'Single person household (at home)'    : {'occupancy': 1, 'at_home': True},
    'Two person household'                 : {'occupancy': 2, 'at_home': False},
    'Two person household (at home)'       : {'occupancy': 2, 'at_home': True},
    'Three person household'               : {'occupancy': 3, 'at_home': False},
    'Three person household (at home)'     : {'occupancy': 3, 'at_home': True},
    'Four person household'                : {'occupancy': 4, 'at_home': False},
    'Four person household (at home)'      : {'occupancy': 4, 'at_home': True},
}
```

Figure 4.16: Code excerpt from `system/functions.py` (household types)

To add different household types, they have to be added to `HOUSEHOLDS`, before rerunning the `populate_households` view function in the `Configuration home.html` template.

[3]The `data` field of this object is populated by using the load model developed by the team at LBNL [1].

4.3.4 Configuration

The `Configuration` model is used to describe a specific system configuration. The `type` field indicates which type the system has. At the time of writing, two different system types are supported: the *Solar Electric System* and the *Solar Thermal System*[4] (with a gas burner as backup heater). To add a new system type, the constants `COMPONENTS` and `SYSTEMS` defined in the file `system/functions.py` need to be edited accordingly (see Section 4.2).

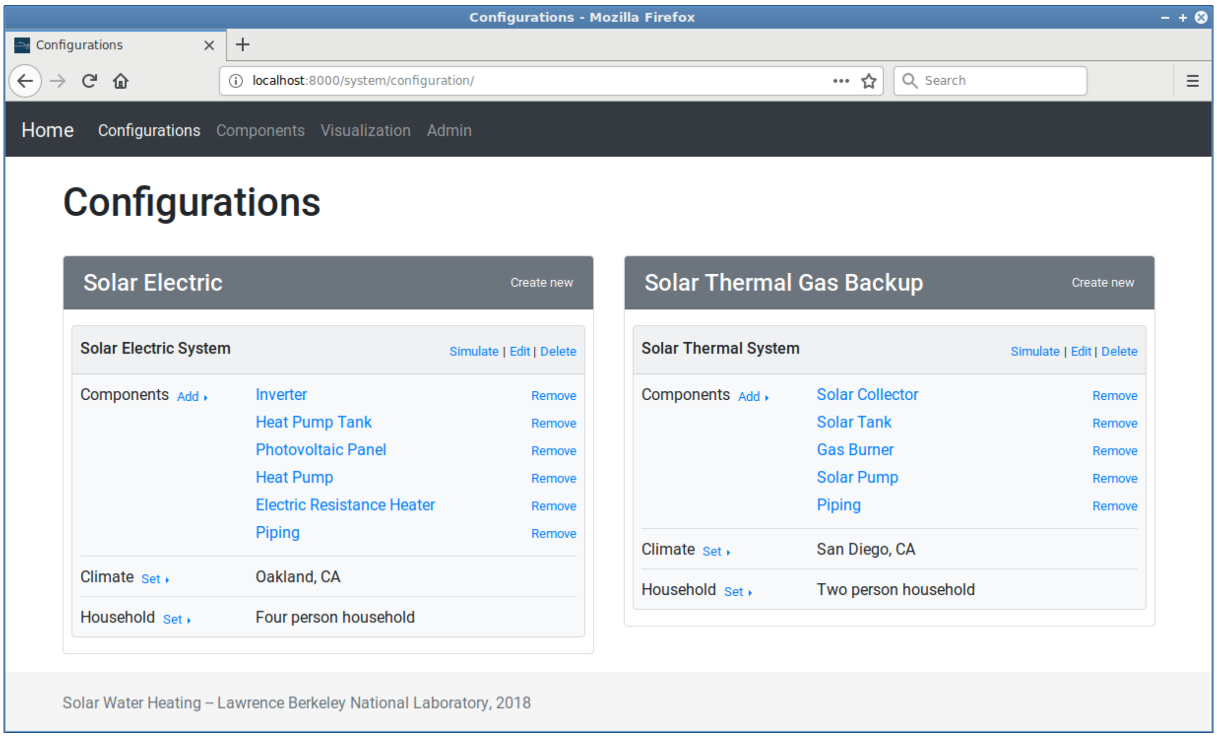
Besides that, the `Configuration` model has three more fields to define the system, the user wants to simulate: the `components` field, the `climate` field, and the `household` field. The `components` field has the *Django* model field type `ManyToManyField`, which defines the relation between objects of the `Configuration` model class and objects of the `Component` model class. By using the relation provided by the `ManyToManyField` field, multiple `Component` model objects can be associated with one or more `Configuration` model objects. It has to be noted, that there is only a reference in the `Configuration` model object and not in the `Component` model object, so the latter is not aware of any association with a `Configuration` model object.

The fields `climate` and `household` are both of the *Django* model field type `ForeignKey`, which represents a reference from one specific `Configuration` model object to one specific `Climate` or `Household` model object. In contrast to the relation to the `Component` model, each `Configuration` model object ever only can be assigned a single climate and household. By using `ForeignKey` instead of `OneToOneField`, it is ensured, that the same `Climate` model object (or `Household` model object respectively) can be assigned to multiple `Configuration` model objects.

In Figure 4.17 the `home.html` template of the `Configuration` model can be seen. When clicking on the "*Configurations*" tab in the navigation bar at the top of the web page, the `config_home` view is called which loads all `Configuration`, `Component`, `Climate`, and `Household` model objects from the project database and forwards them to the `home.html` template.

[4]This system has been completely implemented by the team at LBNL [1].

Similar as seen in the `Component list.html` template, a card for each type is created, where the card header shows the configuration type and an action link "*Create new*" to create a new `Configuration` model object of the respective type. In the card body all `Configuration` model objects found in the database are visualized as a separate card, shaded grey to be distinguishable from the surrounding card body. The card header representing a `Configuration` model object shows its name and three action links: "*Simulate*", "*Edit*", and "*Delete*". When the user clicks on "*Simulate*", the associated system configuration is simulated, which takes a few seconds, before the user is redirected to the visualization template, where the simulation results are visualized. By clicking on "*Edit*" the name of the `Configuration` model object can be edited in an appearing pop-up form, whereas the action link "*Delete*" triggers the deletion of the current `Configuration` model object, after confirming one more time.


Configurations - Mozilla Firefox
Configurations
localhost:8000/system/configuration/
Search
Home
Configurations
Components
Visualization
Admin

Configurations

Solar Electric — Create new

Solar Electric System — Simulate | Edit | Delete

Components Add ▸	Inverter	Remove
	Heat Pump Tank	Remove
	Photovoltaic Panel	Remove
	Heat Pump	Remove
	Electric Resistance Heater	Remove
	Piping	Remove
Climate Set ▸	Oakland, CA	
Household Set ▸	Four person household	

Solar Thermal Gas Backup — Create new

Solar Thermal System — Simulate | Edit | Delete

Components Add ▸	Solar Collector	Remove
	Solar Tank	Remove
	Gas Burner	Remove
	Solar Pump	Remove
	Piping	Remove
Climate Set ▸	San Diego, CA	
Household Set ▸	Two person household	

Solar Water Heating – Lawrence Berkeley National Laboratory, 2018

Figure 4.17: Rendered `configuration/home.html` template

The card body of each `Configuration` model object shows a list of associated components, the assigned climate and the assigned household. If these fields have not been assigned yet, a respective warning is shown in red font. The action link "*Add*" opens a dropdown menu, which shows a list of all available components found in the project database, when being clicked on. When an entry of the dropdown menu is clicked, the respective component is added to the current configuration and can be removed by clicking on the action link "*Remove*" displayed next to its name. Every component can only be added once to a specific configuration. As soon as a climate or household has been assigned to a configuration, it can not be removed again, but only exchanged by another one by clicking on the action link "*Set*" and choosing from the appearing dropdown menu.

In the current version of this project's source code, an action link each for populating the `Climate` and `Household` model objects is shown at the top of the `Configuration` page, if there are no `Climate` and `Household` model objects found in the database. If the database has been populated, the action links are hidden in order to avoid unnecessary duplicates, when running the populate functions multiple times.

Table 4.3 shows an overview of all view functions of the `Configuration` model, their relative URL and the targeted template. In contrast to the `Components` model, the `Configuration` model only uses one template, so all the view functions will render the `Configuration` template `home.html`.

Table 4.3: Configuration views-to-URL mapping

View	URL	Target template
config_home	configuration/	home.html
config_set_name	configuration/<configuration_id>/set_name/	home.html
config_set_climate	configuration/<configuration_id>/set_climate/<climate_id>/	home.html
config_set_household	configuration/<configuration_id>/set_household/<household_id>/	home.html
config_add_component	configuration/<configuration_id>/add_component/<component_id>/	home.html
config_remove_component	configuration/<configuration_id>/remove_component/<component_id>/	home.html
config_create	configuration/<configuration_type>/create/	home.html
config_delete	configuration/<configuration_id>/delete/	home.html
config_invoke	configuration/<configuration_id>/invoke/	home.html[5]

In order to invoke the system simulation, the `Map` class's function `get_system_class` is used to instantiate an object of the SWH `System` class with the given parameters retrieved from the `Configuration` model object's associated components, climate, and household. After that the SWH `System` class's function `simulate` is used to invoke the system simulation. The results are stored in a local `pickle` file in order to be retrievable from the `Visualization` model.

[5]If the simulation is successfully invoked in the view function `config_invoke`, the `home.html` template of the `Visualization` model is being shown.

"*Pickling*" a Python object is the process of converting a binary object into a byte-stream (also known as serializing) with the purpose of being able to store the object on a file system. This mechanism is provided by the Python module `pickle` (available in the Python *Standard Library*) and is specific to Python in order to be independent of external serialization standards. The reverse process is called "*unpickling*" [44].

For every `Configuration` model object a separate `pickle` file is stored in order to separate the simulation results of each configured system. The name for the `pickle` file is built like in the following scheme (represented as Python code)

```
file_name = 'results_' + str(config.type) + '_' + str(config.id) + '.p'
```

which would result in "`results_solar_electric_1.p`" for a `Configuration` model object of type `solar_electric` with the `id` being `1`. This ensures, that for every `Configuration` model object there can only exist a single results file, since every `id` field is unique.

4.3.5 Visualization

The `Visualization` model is technically spoken not a *Django* model like the previously described models, since it does not have an own *Django* model class implementation in `system/models.py`. This is due to the fact, that the visualization of the results is handled in JavaScript except for a couple of Python helper functions and the `Visualization` template defined in `system/templates/visualization/home.html`. And since the results are stored as local files, there is no state to be captured by a *Django* model.

The `view` function first retrieves all `Configuration` model objects from the database in order to be aware what results files to expect. For every file which is stored according to the scheme explained in the proceeding section, an entry in a Python list called `plots` is created. This entry is a Python dictionary formated as shown in Figure 4.18. The `name` key is filled with the `Configuration` model object's `name` field and the `id` key is assigned the `type` and `id` field of the `Configuration` model object (concatenated as string). The `index` key holds the index[6] for the plot, which in the case of this project is a list of timestamps, starting at January 1st at 12 AM and ending at December 31st

[6]The index represents the x-axis of the plot.

at 11:00 PM. The `series` key contains a dictionary with all time series returned from the system simulation. It is created dynamically, so if a new series is added in the SWH project source code, it will be reflected here without any intervention necessary. Its `data` key contains the simulation data as list of all time steps and the `name` key is a human readable name retrieved from the SWH label map, as provided through the `Map` class. For simplicity reasons Figure 4.18 only holds time series with two example data values. The last key called `totals` contains a dictionary holding all annual totals returned from the system simulation.

```
plot_data = {
    'name': 'Solar Electric System', \
    'id': 'solar_electric_1', \
    'index' : ["2018-01-01 00:00:00", "2018-01-01 00:00:00"], \
    'series': {\
        'Q_hp' : {\
            'data': [0.0, 32.4], \
            'name': 'Heat Pump - Delivered'}, \
        'Q_dem' : {\
            'data': [0.0, 33.3], \
            'name': 'Heat Demand'}, \
    }, \
    'totals': {\
        'Sol_Fra': {\
            'name': 'Solar Fraction', \
            'data': 0.53}, \
        'Q_hp' : {\
            'data': 1324.42, \
            'name': 'Heat Pump - Delivered'}, \
    }
}
```

Figure 4.18: Data structure holding plot data

In Figure 4.19 the rendered `Visualization` template is shown. The current version of the web framework will display all available plots each in a card using the full width of the browser window. The card header will show the name of the respective `Configuration` model object on the left side and an action link to show the by default hidden totals in a table on the right side. After clicking on "*Show/Hide Table*" a collapsed table will be expanded in between the card header and the card body which is showing the simulation results as plotted time series.

In the scope of this project, the open-source project *Plotly* [45] has been used. *Plotly* includes interactive graphing libraries both for Python [46] and Javascript [47]. For the `Visualization` model the JavaScript library *Plotly.js* has been used. It is following a declarative design guide, where charts are described as JSON objects [47].

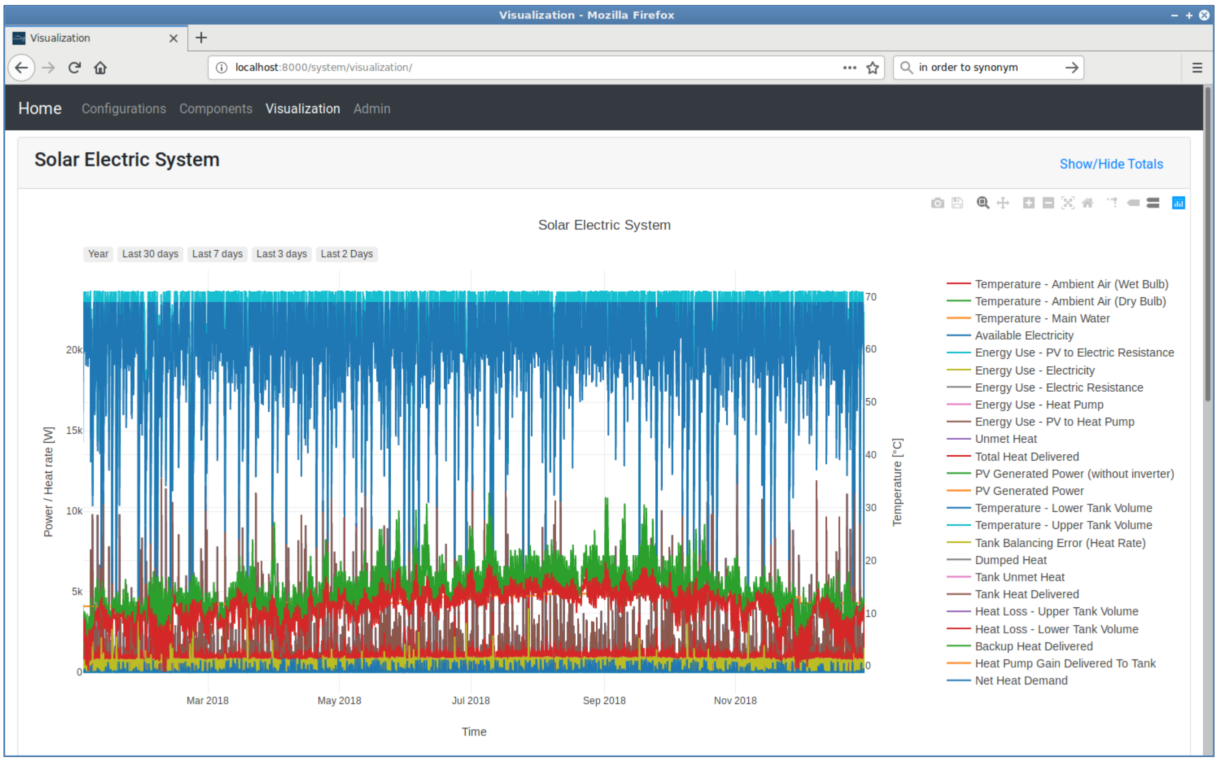


Figure 4.19: Rendered `visualization/home.html` template

The main advantage of using the browser-based graphing library *Plotly* is that the user can interactively manipulate the current view of the plotted graph. As can be seen in the top right corner of the card holding the results graphs in Figure 4.19, *Plotly* provides a tool-bar for the interaction with the graph view[7]. Among those provided tools are the option to download the current view as an image, the user can select areas to zoom in to and reset the zoom, it can be chosen to either show labels of all curves or only the one closest to the cursor and there is an option to export the graphs to an online tool called *Plotly Chart Studio* [48]. On the right side of the plot there is a list of available data series with a color reference to the associated graph in the plot. With a single click

[7]This tool-bar is only shown, when the mouse cursor is moved above the top right corner of the plot.

on the name of a data series the associated graph will be hidden (or shown) in the plot, whereas a double click hides (or shows) all other graphs. This feature is especially useful to analyze a single or multiple selected data series.

Since the time series in this project have two different units *Watt* and *Celsius*, two y-axes are used to show both data types simultaneously. In order to decide which data series is assigned to which y-axis, the label of the data series is being checked. Only if it starts with the string "*Temperature*", it is assigned the right y-axis and additionally its values are converted from *Kelvin* to *Celsius*. *Plotly* enables the user to independently zoom in on each y-axis, by scrolling in and out while keeping the cursor over the respective y-axis. Additionally, the visible segment of an axis can be moved by dragging with the mouse on the axis in one direction. On the top left corner of the plot several buttons can be seen. By clicking on them, predefined time ranges are applied to the x-axis (e.g. the last 30, 7, 3 or 2 days). All these features are very useful to visualize a very specific part of the simulation results, which is essential for the analysis of the simulated system.

The JavaScript code to create the plot using the *Plotly.js* library can be found in the function `plot_results` in the file `swh_plots.js` in the application-level `static` folder of the *Django* project. In the `Visualization` template `home.html` the function is called for every available plot at the end of the file.

4.4 Set-Up

This section contains the instructions on how to get the implemented *Django* project up and running. Generally, it is possible to deploy the project on every machine, which is capable of running Python (since *Django* is implemented in and using Python) and which has a JavaScript-enabled web browser. The following commands are to be exectued in a *Unix* terminal, and have only been tested on a computer running the *Linux* operating system.

In order to ensure that the installation of the required Python packages don't interfere with a potential system-wide Python installation, the "*Python Data Science Distribution*" *Anaconda* is used [49]. *Anaconda* includes a Python package and environment manager, supports both Python version 2 and 3 and is available for all major operating systems (including *Linux*, *Windows* and *macOS*). Instructions on where to download and how to install *Anaconda* can be found in the online documentation (see [50]). One advantage of using *Anaconda* over alternative package and environment manager is that many useful Python libaries like `pandas` or `numpy` are already included in the *Anaconda* installation.

After having installed *Anaconda*, a new environment can be created by using the following command, where `<env_name>` should be replaced with the name of the new environment:

```
$ conda create -n <env_name>
```

After the new environment has been created, it can be activated or deactivated by running one of the following commands, respectively. When the environment is activated, it appears in parentheses in front of the terminal prompt (which usually contains the user name and the name of the machine).

```
$ conda activate <env_name>
$ conda deactivate
```

After having activated the new environment, the installed packages can be shown by running the command `conda list`. In addition to the package names and versions this also prints the installation path of the environment, just for reference. Now the required *Django* Python package needs to be installed by running the following command[8]:

```
$ conda install django
```

Besides the *Django* Python package this will install additional dependencies required by *Django*. To now add the Python system model of the SWH project the following command has to be exectued in the same directory as the root folder of the SWH source code (`<swh_root_folder>` needs to be replaced with the actual folder name):

```
$ pip install -e <swh_root_folder>
```

`pip` is Python's default package manager and provides an "*editable*" installation option by using the `-e` flag. This ensures, that the above created Python environment is referencing

[8]For this project, *Django* version 2.1.7 has been used.

the locally stored source code of the SWH project for the web framework. Since both the web framework and the SWH system model are still in development, changes on both code bases will be reflected when running the *Django* project.

After having installed all required dependencies, the installation can be tested by starting the *Django* development server (on `http://localhost:8000` or `http://127.0.0.1:8000` on *Windows* machines) with the following command from the same directory as the file `manage.py` is stored in:

```
$ python manage.py runserver
```

If this command is being executed without any errors, the installation was successful. Only one warning will be displayed, when running the development server for the first time. It states that there are unapplied migrations and also provides a list of them. The reason for this is, that the installed applications listed in the file `settings.py` still need to be migrated to the project database. This can be done, by running the following command:

```
$ python manage.py migrate
```

In order to be able to access the *Django* administration site, a user with the required rights has to be created. After running the following command and providing a user name, an optional email address and a password, a super user (with admin rights) is created:

```
$ python manage.py createsuperuser
```

4.5 Usage

The detailed usage of the different parts of the front-end has already been described in Section 4.3, so this section is intended to provide more details about the system-wide usage and typical work-flow of the web framework.

After having done the set-up (as described in the previous section), the project database does not contain any models. So initially the `Component` and `Configuration` model objects have to be created and the `Climate` and `Household` model objects have to be imported by running their respective populate functions.

The web framework has been designed in order to run system simulations with different system configurations in order to analyze and compare the results. So either an existing configuration can be simulated and analyzed before its components (or climate or household) are changed in order to simulate it again, or multiple system configurations (even of the same system type) can be created and simulated. The latter approach has the advantage that no simulation results are overwritten.

In order to add new component or configuration types to be available for system simulations through the web framework, the constants `COMPONTENTS` and `SYSTEMS` in the file `system/functions.py` have to be edited accordingly. This and the option to add different houshold configurations (as described in Section 4.3.3) are the only two cases, where the user would have to interact with anything outside of the web browser. But since this not expected to be part of the typical usage of the web framework, it can be neglected in terms of overall usability.

In order to deploy the web framework in production mode on a publicly available server, so that it can be accessed not only from the local computer, but basically from every device connected to the Internet with a JavaScript-enabled browser, there are several options. One of them is to host a virtual private server offered by the cloud service provider *DigitalOcean* [51]. Since the deployment is not part of this thesis, this detailed description [52] is referenced and can be consulted in case needed.

5 Results and Outlook

In this chapter, simulation results of the *Solar Electric System* are being discussed, before a comprehensive list of potential features, which should be considered for future work on the web framework, is provided. Finally, the results of the work conducted in this thesis is summarized and final conclusions are being presented.

5.1 Simulation Results of Solar Electric System

In the following the simulation results of the *Solar Electric System* for two different system configurations are shown. The simulations have been conducted for a four person household (not being at home during the day, i.e. `at_home = False`) and *Oakland, CA* as climate zone. The differentiating parameter is the size of the PV panel $A_{panel,eff}$, resulting in two different values for its peak power $P_{pv,peak}$. The sizes have been chosen as shown in case (1) and (2) in Table 3.2 on page 13 as 6.67 m^2 (1) and 26.67 m^2 (2) for $A_{panel,eff}$, which respectively yields in 1.0 kW (1) and 4.0 kW (2) for $P_{pv,peak}$ for the given reference irradiation ($I_{ref} = 1000\ \frac{W}{m^2}$) and panel efficiency ($\eta_{pv} = 15\ \%$). As HPWH Unit D, as shown in Table B.3, has been chosen, which defines the parameters of the heat pump, the heat pump tank, and the electric resistance heater component. In Table E.2 in Appendix E the detailed system configuration and the assigned parameters are shown.

As described in Section 4.3.5 on page 44ff, the simulation results consist of multiple time series and their annual totals. In Table E.1 in Appendix E the full list of annual totals and average temperature values is provided. In the following, a representative selection of them is discussed further, before case (3) and (4) in Table E.1, which have been conducted to analyze the effects of an adjusted heat pump control scheme, are being described.

For both cases (1) and (2) the annual net heat demand for a four person household is $4,606.57\ kWh$ which the system almost completely was able to provide except for an unmet heat capacity of 0.32 kWh which can be neglected. This and the fact that there only has been 23.98 kWh of backup heating capacity delivered by the electric resistance heater throughout the year demonstrates, that with a rated heating capacity of 1.82 kW the heat pump is more than sufficiently sized to supply a four person household.

The main differences for the two cases can be seen in the amount and usage of the generated PV electricity. For case (1) the PV panel is generating $1,612.76$ kWh in one year compared to four times as much with $6,448.63$ kWh for case (2). This correlates with the ratio of panel areas for case (1) and (2), which also is 4. For both cases, the heat pump is consuming $4,002.98$ kWh of electricity in a year, but with $1,696.81$ kWh the system in case (2) has about 70 % more PV generated electricity to power the heat pump than the system in case (1) with $1,015.89$ kWh. For the electric resistance heater this effect is even more significant: In both cases the annual consumption for the electric resistance heater is 23.98 kWh (which is rather low), but in case (2) around a quarter of that (6.17 kWh) is being provided by the PV compared to 1.15 kWh as in case (1). All this is resulting in a considerably distinguished total solar fraction for both cases. While case (1) shows a solar fraction of 34.30 %, the increased panel area in case (2) is resulting in a solar fraction of 57.30 %, which is almost twice as high as in case (1). Another meaningful difference is the total annual surplus of electricity which is available to be used for other appliances in the household or can be fed back into the grid, if a NEM has been installed. The system as configured in case (1) yields 595.72 kWh of over-produced electricity, while in case (2) almost eight times more ($4,745.65$ kWh) electric power is available for other uses.

Which of the cases is more suitable for a given household depends on what the usual electricity consumption of the household for other appliances looks like. For households using a lot of energy for areas like space heating or cooling, light or washers and dryers it might make sense to invest in a larger PV panel.

Another part of the simulation results are the time series, which can be analyzed in interactive plots, created by the open-source graphing library *Plotly*. For simplicity reasons, four screenshots illustrating representative sections will be discussed in the following. On each of these screenshots only the relevant time series have been selected to ensure proper visibility.

Figure 5.1 gives an overview about the heat demand and supply in relation to the tank water temperature for seven days end of June. It can be seen, that the tank temperature (light blue) stays within the range of about 50 and 75 $^{\circ}C$ and drops every time there is a heat demand peak (dark blue). The orange data series shows the heat gain delivered by the heat pump to the hot water tank. It can be clearly noticed, that the heat pump is only switched on, if the tank water temperature drops below a certain limit and is switched off as soon as the defined temperature threshold is exceeded. The pink data series denotes the unmet heat gain, which is always zero.

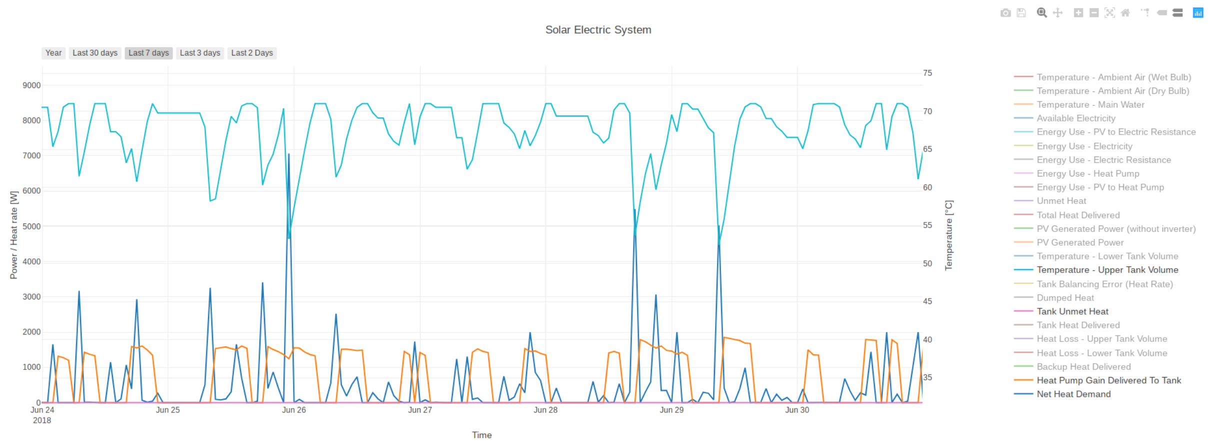


Figure 5.1: Demand and delivered heat for case (1) and (2)

In Figure 5.2 a closer zoom showing about 24 hours in the beginning of July is illustrated. Again it is clearly visibly, that the tank temperature (light blue) is following the heat demand (dark blue, obscured by the brown data series) and falls every time there is a peak. In this plot, the data series of the delivered heat capacity (brown) has been activated and is completely covering the data series of the demand heat capacity, since the demand is always met during this time. Also the data series for the dumped heat capacity (grey) has been activated and shows two smaller peaks at 12:00 PM and 8:00 PM.

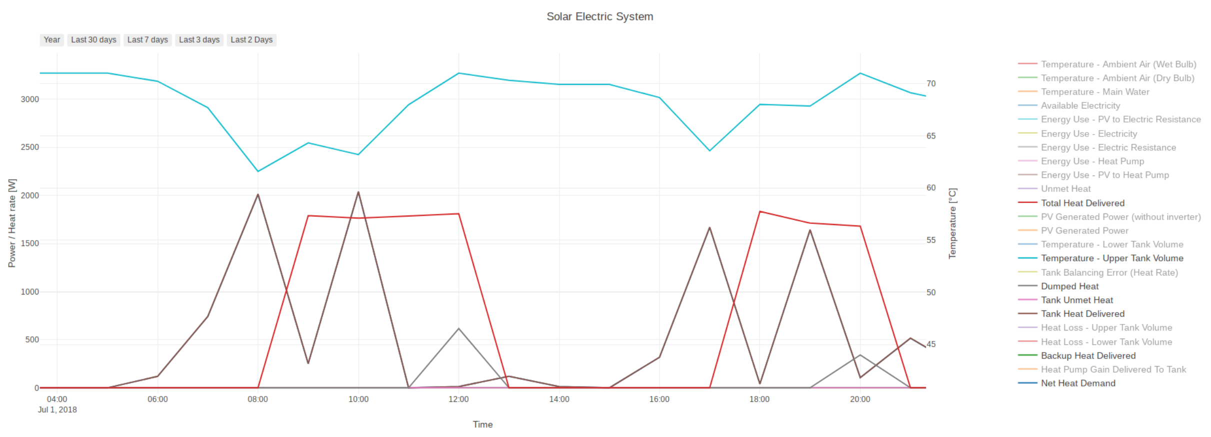


Figure 5.2: Demand and delivered heat for case (1) and (2), close-up

Whereas the previous two plots are valid for both cases (1) and (2), since the demand and the delivered heat capacity are independent of the size of the PV panel, the following two screenshots are taken from different plots. In Figure 5.3 the electricity generation and consumption of case (1) is shown. The orange data series is representing the electricity which is generated by the PV and the pink data series shows the electricity consumption of the heat pump (both in Figure 5.3 and 5.4).

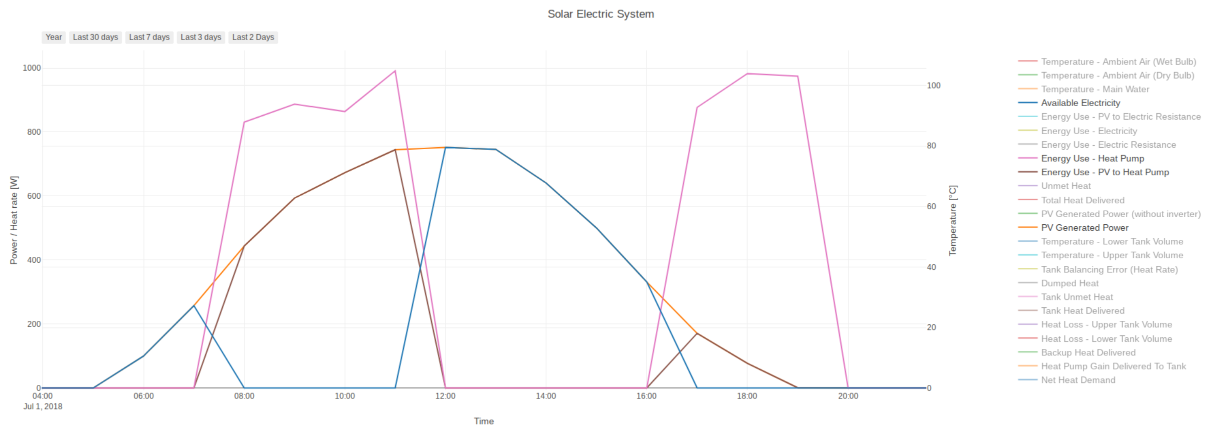


Figure 5.3: Electricity generation and consumption for case (1)

The main difference between case (1) and (2) is, that in case (2) the amount of available electricity is much higher (represented by the dark blue data series) and is almost consistently present for the time when the PV is generating electricity. Also, the amount of electricity coming from the PV used to power the heat pump (brown) is almost always equal to the electricity consumed by the heat pump (pink). Only at around 5:00 PM, where the PV's output power is fading due to the setting sun, the heat pump is not fully powered by the PV for case (2).

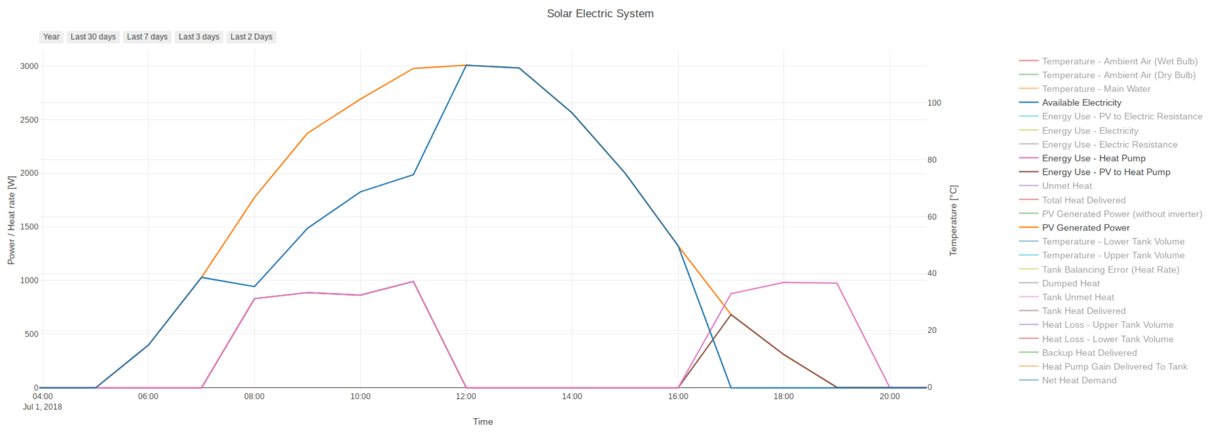


Figure 5.4: Electricity generation and consumption for case (2)

Besides case (1) and (2), Table E.1 also shows the annual totals for case (3) and (4). These cases use the same system configuration as case (1) and (2) respectively, but with a 5 K increased value of the threshold temperature T_{limit}, which is used for the heat pump control scheme illustrated on Figure 3.6 on page 20. The maximum water temperature T_{max}, which the tank is tested for, is used as threshold when to switch off the heat pump. When increasing this threshold, a few effects can be noticed. For once, the unmet heat capacity is dropping to zero and the amount of heat provided by the electric resistance heater (denoted as "*Backup Heat Deliverd*)" is roughly cut in half. On the downside, the amount of dumped heat capacity increases drastically, it is around 13 times more than for case (1) and (2). Also, the total consumed electricity roughly doubles and the amount of available electricity drops. Especially for the comparison of case (1) and (3) the reduced amount of available electricity is significant. Although the solar fraction is rising for an increased threshold temperature as in case (3) and (4) compared to case (1) and (2), the fact, that the overall electricity consumption is roughly doubling is a critical design decision in favor of a lower threshold temperature.

5.2 Next Steps for Web Framework

It has to be pointed out, that the web framework which has been implemented in the scope of this thesis, is still under development. In the version it has during the time of writing, the basic use-case has been covered which was the minimum goal of this thesis. It is possible to set up one or multiple system configurations for both the *Solar Electric* and the *Solar Thermal System*. Also, components can be created and their parameters and sizes can be defined without having to interact with the source code anymore. Both the simulation can be invoked through the browser as well as the results can be viewed, interactively analyzed, and stored using a wide spread open-source graphing library. The user is able to decide which climate the system is simulated in and for which household size it is aimed to be used for.

However, during the development several potential features have been identified, which will be elaborated on in the following. This list can be seen as guideline which next steps can be taken in order to refine the web framework even further:

- Firstly, the author wants to emphasize, that at the time of reading this thesis, some of the following points, will potentially already have been implemented or won't be relevant anymore.
- For the current version of this web framework, there exist several databases, the one used as *Django* project database, the one created by the team at LBNL holding the climate and load data, and an additional one also created by the LBNL project team holding all kinds of information including component performance parameters and sizes used for the terminal-based simulation. In the future, these databases might all be integrated into one single database, potentially the one used as *Django* project database, since this database is required anyway in order for a *Django* project to work. One or multiple new *Django* models could be created to hold the data of the SWH project databases. By doing so, the whole potential of *Django* can be utilized, even if the SWH project is further growing.
- As briefly mentioned in Section 4.5, where the usage of the web framework is explained, the web framework could be deployed on a publicly available server. In order to do so, a couple of things would need to be considered: First of all, a suitable user authentication mechanism would need to be added to the web framework. *Django* is fully equipped with the required building blocks, for further reference about that, see [53]. Secondly, the deployment should not be done by running the development server, but through a proper deployment platform. The *Django* documentation site is advising to use a Web Server Gateway Interface (WSGI) for that and includes a detailed description of how to get started and set one up in [54]. There are many more important aspects to consider when deploying, like using HTTPS instead of HTTP, setting the `DEBUG` parameter to `False` or defining the list `ALLOWED_HOSTS` in the *Django* settings in `settings.py`. A comprehensive checklist can be found in [55].
- The terminal-based simulation of the SWH system model can not only be deployed for an individual household, but also for the community case, where multiple households with a different number of occupants and `at_home` setting (see Section 4.3.3) are being supplied by a single SWH system. At the time of writing the web framework does only support the simulation with an individual household as load.

In order to accomplish the support for the community case, the `Household` and `Configuration` models need to be extended appropriately.

- Since the main focus during the development of the web framework has been put on the implementation of the required features, no automatic testing framework has been integrated yet. The *Django* documentation recommends using `unittest`, which is part of the Python *Standard Library* for this purpose [56].
- In the current version of the web framework, the `parameters` field of the `Components` model and the `data` field of both the `Climate` and `Household` models have been implemented as *Django* model field type `TextField`, by storing the string representation of the data. This method has been chosen due to simplicity reasons and for being able to move forward faster. Using the `parameters` field as an example, a future implementation could be to create a new `Parameter` *Django* model with a `key` and `value` field (and possibly a `type` field) and reference this object from the `Component` model instance. This way, it is not only possible to use the same parameter for multiple components, but also ensure an efficient design of the *Django* project database.

At the time of writing the following issues are not resolved yet and are listed here to give the reader and potential user of the web framework a thorough understanding of the current state of the web framework.

- When setting up the *Django* project on a new computer, all *Django* model objects have to be created from scratch. The reason for that is not clear. Although the *Django* project database has been added to the remote code repository, the migration of model object is not working as supposed to. This topic needs further investigation in order to allow the import of already defined components and configurations, when setting up the *Django* project on a new computer.
- At the current state, there is no sanity check of a `Configuration` model object in place. That means, when hitting the "*Simulate*" button in the `Configuration` template `home.html`, an error appears, if not all necessary components have been assigned to the `Configuration` object model. Also, it is not checked if a `Climate` and `Household` model object have been assigned before running the simulation.

For this, the `Configuraion` model type can be used to derive the list of required components (and backup components) as stored in the `SYSTEMS` constant in the file `functions.py`. The user should not be able to run the simulation, as long as the `Configuration` model object has not been assigned all required components, a climate, and a household.

5.3 Conclusions

In this Master's Thesis a flexible web framework based on *Django* has been implemented in order to simplify the set-up and simulation of complex Python system models. To demonstrate the benefits and functionality of the web framework, the Python system model of the SWH project developed by LBNL [1] has served as an example system. In the scope of this thesis and as adjacent research to the already existing *Solar Thermal System*, the *Solar Electric System* has been implemented both on a component- and system-level, partly based on the work conducted by the team at LBNL.

By using the implemented web framework, it is possible to create component objects and set up their parameters and sizes. Additionally, the user can create system configurations and assign the created components, a climate zone and a household representing a specific hot water load profile, before invoking the system simulation. Through an interactive visualization the web framework allows a thorough analysis of the simulation results. All of that can be done exclusively through a web browser, which essentially simplifies the process compared to as it would be without the web framework.

The web framework is meeting all main requirements listed in Section 1.2, but is still in an early stage of development. Thus, the author can not guarantee an error-free usage of the web framework and therefore has pointed out the most important aspects to be considered for future work in Section 5.2.

Since the web framework is implemented as a *Django*-based application, it can also be used for other system models written in Python, by only adjusting a minimal amount of source code. This way, not only the team at LBNL but also other researchers and developers can benefit from the work conducted in this thesis.

Acknowledgments

First of all, I would like to express special thanks to my mentor and supervisor Milica Grahovac, who not only provided the topic for this Master's Thesis and handled the administrative part at LBNL, but was my resourceful and always approachable go to person throughout the 6 months to discuss ideas. During our weekly meetings and extensive email correspondence she consistently provided insightful suggestions and great support while leaving me a generous amount of flexibility.

Also, I'd like to thank Prof. Thomas Hamacher for giving me this exceptional chance to write my Master's Thesis in the golden state of California, for his great support during the definition phase of the thesis, and his trust in me.

For supporting me in finding a topic for my thesis and finally introducing me to Milica, I would like to thank Thomas Massier, whom I got to know in Singapore during a very insightful research internship at TUMCREATE.

Furthermore, I very much appreciated the emotional support my parents Barbara and Peter Bohnengel offered me during countless phone calls, while I was getting some fresh air in my breaks.

I am very thankful to Edith and Hubert Gerhart for handling the print and submission of my thesis as well as for proofreading it. Not many have the luck to have such supportive and devoted parents-in-law.

Finally, without whom I would not be where I am right now, my warmest thanks go to my wife Stefanie Gerhart for her irreplaceable support, for continuously challenging me to give my best and for her unconditional love and trust during the last 10 years.

List of Figures

List of Tables

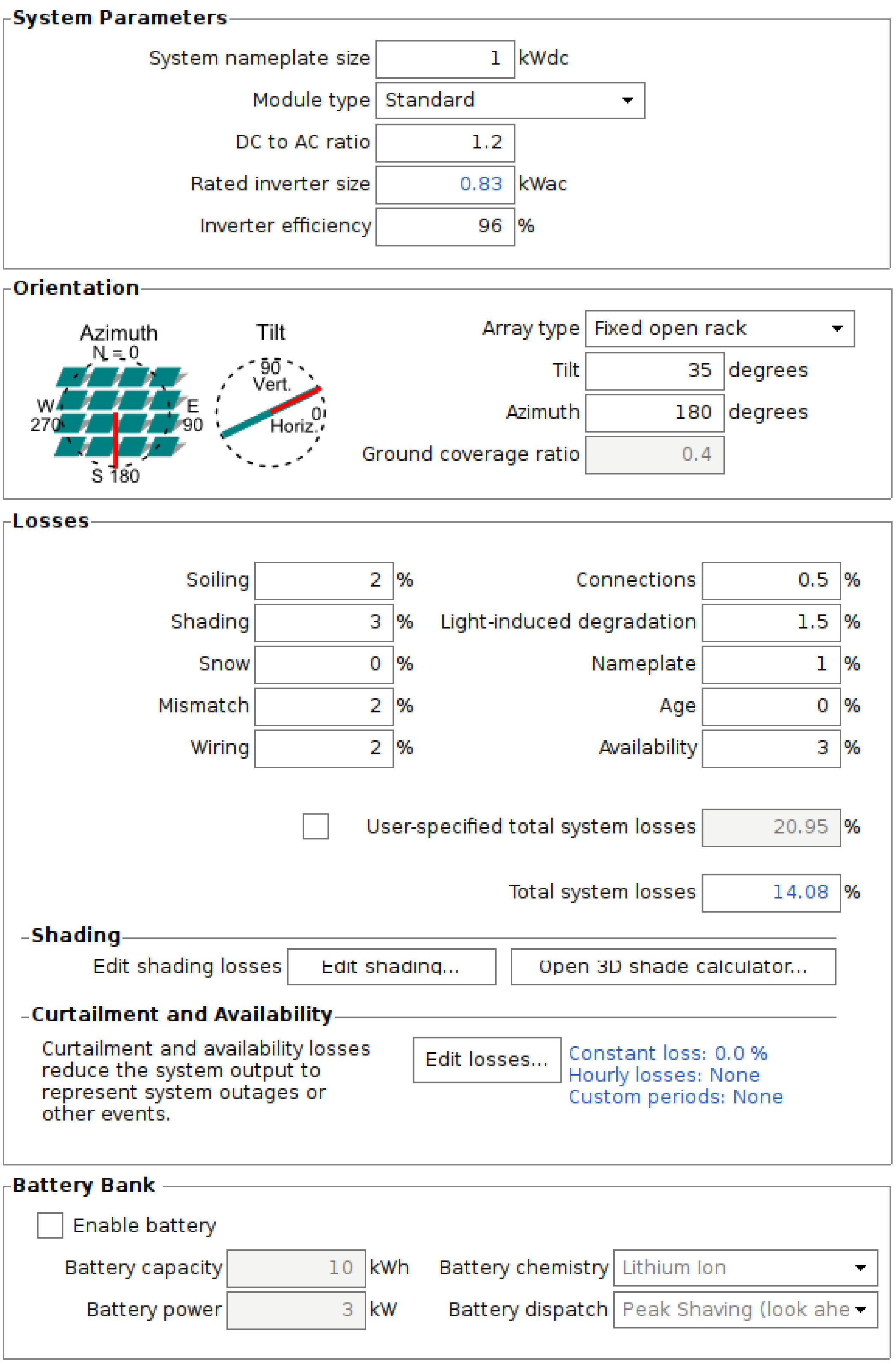
List of Symbols

Symbol	**Meaning**	**Unit**
$A_{panel,eff}$	Effective PV panel area	m^2
COP_{hp}	Coefficient of performance of HP	$-$
$COP_{hp,rated}$	Rated coefficient of performance of HP	$-$
dt_{hyst}	Temperature hysteresis for HP control scheme	$^\circ C$
$f_{hp,perf}$	Performance factor of HP	$-$
I	Current solar irradiation	$\frac{W}{m^2}$
I_{ref}	Reference solar irradiation	$\frac{W}{m^2}$
P_{el}	Consumed electrical power of electric resistance heater	W
P_{hp}	Consumed electrical power of HP	W
$P_{pv,peak}$	Rated peak power of PV	W
$P_{pv,ac}$	Generated electrical power of PV	W
$Q_{dem,total}$	Total heat demand	W
$Q_{el,nom}$	Nominal heating capacity of electric resistance heater	W
Q_{hp}	Effective heating capacity of HP	W
$Q_{hp,rated}$	Rated heating capacity of HP	W
RH	Relative air humidity	$-$
T	Air dry bulb temperature	$^\circ C$
T_{limit}	Maximum temperature for HP control scheme	$^\circ C$
T_{wa}	Main water temperature	$^\circ C$
T_{wb}	Air wet bulb temperature	$^\circ C$
$\eta_{dc,ac}$	DC/AC conversion efficiency	$-$
η_{el}	Electric resistance conversion efficiency	$-$
η_{pv}	PV conversion efficiency	$-$

Abbreviations

AC	Alternating Current
CDN	Content Delivery Network
CEC	California Energy Commission
CSS	Cascading Style Sheets
COP	Coefficient of Performance
DC	Direct Current
HP	Heat Pump
HPWH	Heat Pump Water Heater
HTML	Hypertext Markup Language
HTTP	Hypertext Transfer Protocol
HTTPS	Hypertext Transfer Protocol Secure
JSON	JavaScript Object Notation
IOU	Investor Owned Utility
PGE	Pacific Gas and Electric
PV	Photovoltaic
LBNL	Lawrence Berkeley National Laboratory
MTV	Model Template View
MVC	Model View Controller
NEM	Net Energy Metering
NREL	National Renewable Energy Laboratory
NSC	Net Surplus Compensation
ORM	Object Relational Mapper
SAM	System Advisor Model
STC	Standard Test Conditions
SWH	Solar Water Heating
TMY3	Typical Meteorological Year
URL	Uniform Resource Location
VCS	Version Control System
WSGI	Web Server Gateway Interface

A SAM PVWatts Model Configuration



System Parameters

System nameplate size	1	kWdc
Module type	Standard	
DC to AC ratio	1.2	
Rated inverter size	0.83	kWac
Inverter efficiency	96	%

Orientation

Array type	Fixed open rack	
Tilt	35	degrees
Azimuth	180	degrees
Ground coverage ratio	0.4	

Losses

Soiling	2	%	Connections	0.5	%
Shading	3	%	Light-induced degradation	1.5	%
Snow	0	%	Nameplate	1	%
Mismatch	2	%	Age	0	%
Wiring	2	%	Availability	3	%

☐ User-specified total system losses 20.95 %

Total system losses 14.08 %

Shading

Edit shading losses [Edit shading...] [Open 3D shade calculator...]

Curtailment and Availability

Curtailment and availability losses reduce the system output to represent system outages or other events.

[Edit losses...]

Constant loss: 0.0 %
Hourly losses: None
Custom periods: None

Battery Bank

☐ Enable battery

Battery capacity	10	kWh	Battery chemistry	Lithium Ion
Battery power	3	kW	Battery dispatch	Peak Shaving (look ahe

Figure A.1: SAM PVWatts system design

B Heat Pump Specifications

Table B.1: Coefficients for $COP_{hp,\,rated}$ [28, p. 54]

HPWH Unit	**C1**	**C2**	**C3**	**C4**	**C5**	**C6**
	$[-]$	$[\frac{1}{^{\circ}C}]$	$[\frac{1}{^{\circ}C^2}]$	$[\frac{1}{^{\circ}C}]$	$[\frac{1}{^{\circ}C^2}]$	$[\frac{1}{^{\circ}C^2}]$
Unit A	1.229E+00	5.549E-02	1.139E-04	-1.128E-02	-3.570E-06	-7.234E-04
Unit B	1.192E+00	4.247E-02	-3.795E-04	-1.110E-02	-9.400E-07	-2.657E-04
Unit C	6.945E-02	6.601E-03	1.598E-04	8.842E-04	8.170E-06	3.255E-05
Unit D	9.814E-01	5.334E-02	-2.802E-04	-3.073E-03	-1.384E-04	-2.897E-04
Unit E	2.168E+00	8.124E-02	4.786E-04	-4.870E-02	4.284E-04	-1.499E-03

Table B.2: Coefficients for $Q_{hp,\,rated}$ [28, p. 54]

HPWH Unit	**C1**	**C2**	**C3**	**C4**	**C5**	**C6**
	$[-]$	$[\frac{1}{^{\circ}C}]$	$[\frac{1}{^{\circ}C^2}]$	$[\frac{1}{^{\circ}C}]$	$[\frac{1}{^{\circ}C^2}]$	$[\frac{1}{^{\circ}C^2}]$
Unit A	7.055E-01	3.945E-02	1.433E-04	2.768E-03	-1.069E-04	-2.494E-04
Unit B	5.050E-01	5.116E-02	-2.026E-04	5.444E-03	-1.154E-04	-2.472E-04
Unit C	6.879E-01	1.987E-02	7.659E-04	2.621E-03	5.323E-05	-5.210E-04
Unit D	5.101E-01	3.588E-02	5.563E-05	4.828E-03	-1.348E-04	7.738E-05
Unit E	9.285E-01	4.088E-02	2.737E-04	-3.625E-03	-6.521E-05	-2.986E-04

Table B.3: Rated performance (for $T_{wb} = 14\ ^{\circ}C$ and $T_{wa} = 48.9\ ^{\circ}C$) and tank size [28, pp. 10, 54]

HPWH Unit	**$\mathbf{COP_{hp,\,rated}}$**	**$\mathbf{Q_{hp,\,rated}}$**	**$\mathbf{Q_{aux,\,combined}}$**	**Tank size**
		$[W]$	$[W]$	$[Gallons]$
Unit A	2.43	2350	6500	80
Unit B	2.76	1380	9000	50
Unit C	2.42	2670	4000	50
Unit D	2.77	1820	1700	80
Unit E	2.02	2040	4000	66

C Bootstrap Core Libraries

```
<!-- Offline use case -->
<link href="{% static 'css/bootstrap.min.css' %}" rel="stylesheet">

<!-- Online use case -->
<link rel="stylesheet" href="https://stackpath.bootstrapcdn.com/
   bootstrap/4.3.1/css/bootstrap.min.css" integrity="sha384-
   ggOyR0iXCbMQv3Xipma34MD+dH/1fQ784/j6cY/iJTQUOhcWr7x9JvoRxT2MZw1T"
   crossorigin="anonymous">
```

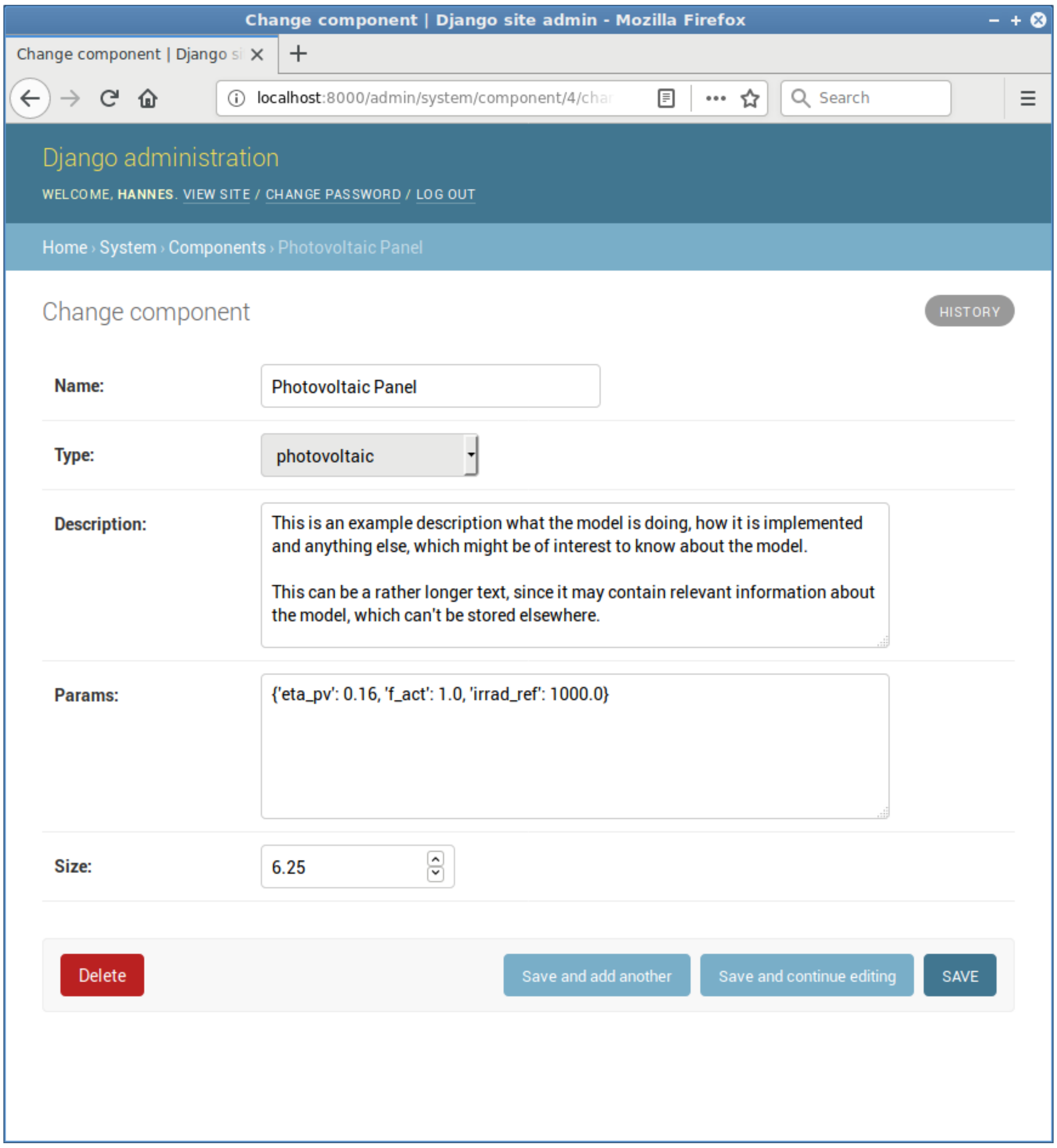
Figure C.1: Import of bootstrap core CSS library

```
<!-- Offline use case -->
<script src="{% static 'js/jquery-slim.min.js' %}"></script>
<script src="{% static 'js/popper.min.js' %}"></script>
<script src="{% static 'js/bootstrap.min.js' %}"></script>

<!-- Online use case -->
<script src="https://code.jquery.com/jquery-3.3.1.slim.min.js" integrity
   ="sha384-q8i/X+965DzO0rT7abK41JStQIAqVgRVzpbzo5smXKp4YfRvH+8
   abtTE1Pi6jizo" crossorigin="anonymous"></script>
<script src="https://cdnjs.cloudflare.com/ajax/libs/popper.js/1.14.7/umd
   /popper.min.js" integrity="sha384-
   UO2eT0CpHqdSJQ6hJty5KVphtPhzWj9WO1clHTMGa3JDZwrnQq4sF86dIHNDz0W1"
   crossorigin="anonymous"></script>
<script src="https://stackpath.bootstrapcdn.com/bootstrap/4.3.1/js/
   bootstrap.min.js" integrity="sha384-
   JjSmVgyd0p3pXB1rRibZUAYoIIy6OrQ6VrjIEaFf/nJGzIxFDsf4x0xIM+B07jRM"
   crossorigin="anonymous"></script>
```

Figure C.2: Import of bootstrap core JavaScript library

D Django Model Administration Pages


Change component | Django site admin - Mozilla Firefox
Change component | Django si
localhost:8000/admin/system/component/4/char
Search
Django administration
WELCOME, HANNES. VIEW SITE / CHANGE PASSWORD / LOG OUT
Home › System › Components › Photovoltaic Panel
HISTORY
Delete
Save and add another
Save and continue editing
SAVE

Change component

Name: Photovoltaic Panel

Type: photovoltaic

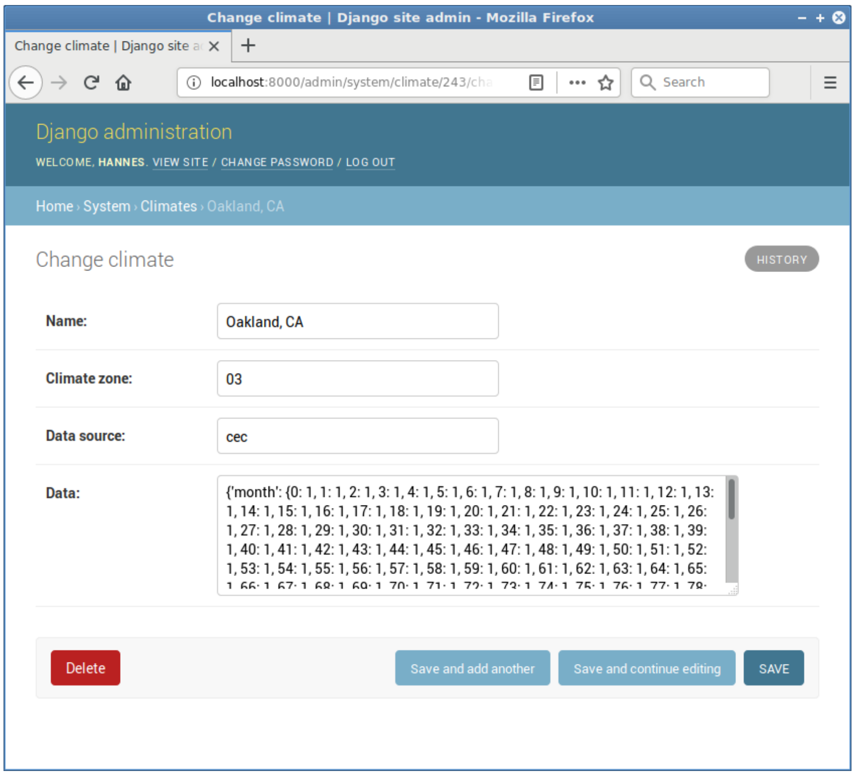
Description: This is an example description what the model is doing, how it is implemented and anything else, which might be of interest to know about the model.

This can be a rather longer text, since it may contain relevant information about the model, which can't be stored elsewhere.

Params: {'eta_pv': 0.16, 'f_act': 1.0, 'irrad_ref': 1000.0}

Size: 6.25

Figure D.1: *Django* administration page - Component


Change climate | Django site admin - Mozilla Firefox
Change climate | Django site a
localhost:8000/admin/system/climate/243/cha
Search
Django administration
WELCOME, HANNES. VIEW SITE / CHANGE PASSWORD / LOG OUT
Home › System › Climates › Oakland, CA
Change climate
HISTORY
Delete
Save and add another
Save and continue editing
SAVE

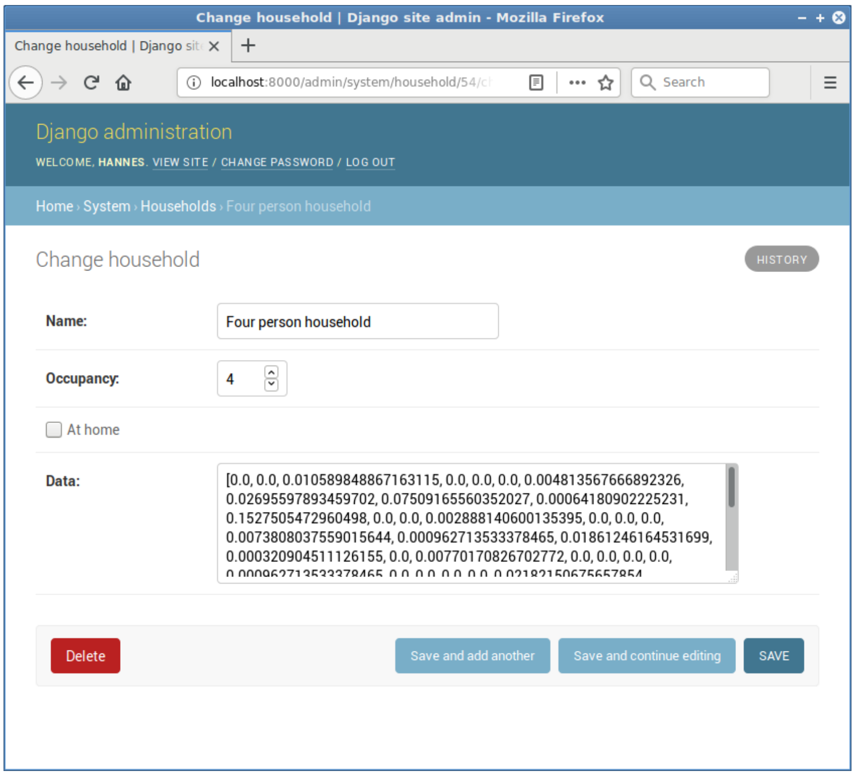
Name: Oakland, CA

Climate zone: 03

Data source: cec

Data:
```
{'month': {0: 1, 1: 1, 2: 1, 3: 1, 4: 1, 5: 1, 6: 1, 7: 1, 8: 1, 9: 1, 10: 1, 11: 1, 12: 1, 13:
1, 14: 1, 15: 1, 16: 1, 17: 1, 18: 1, 19: 1, 20: 1, 21: 1, 22: 1, 23: 1, 24: 1, 25: 1, 26:
1, 27: 1, 28: 1, 29: 1, 30: 1, 31: 1, 32: 1, 33: 1, 34: 1, 35: 1, 36: 1, 37: 1, 38: 1, 39:
1, 40: 1, 41: 1, 42: 1, 43: 1, 44: 1, 45: 1, 46: 1, 47: 1, 48: 1, 49: 1, 50: 1, 51: 1, 52:
1, 53: 1, 54: 1, 55: 1, 56: 1, 57: 1, 58: 1, 59: 1, 60: 1, 61: 1, 62: 1, 63: 1, 64: 1, 65:
1, 66: 1, 67: 1, 68: 1, 69: 1, 70: 1, 71: 1, 72: 1, 73: 1, 74: 1, 75: 1, 76: 1, 77: 1, 78:
```

Figure D.2: *Django* administration page - Climate


Change household | Django site admin - Mozilla Firefox
Change household | Django sit
localhost:8000/admin/system/household/54/c
Search
Django administration
WELCOME, HANNES. VIEW SITE / CHANGE PASSWORD / LOG OUT
Home › System › Households › Four person household
Change household
HISTORY
Delete
Save and add another
Save and continue editing
SAVE

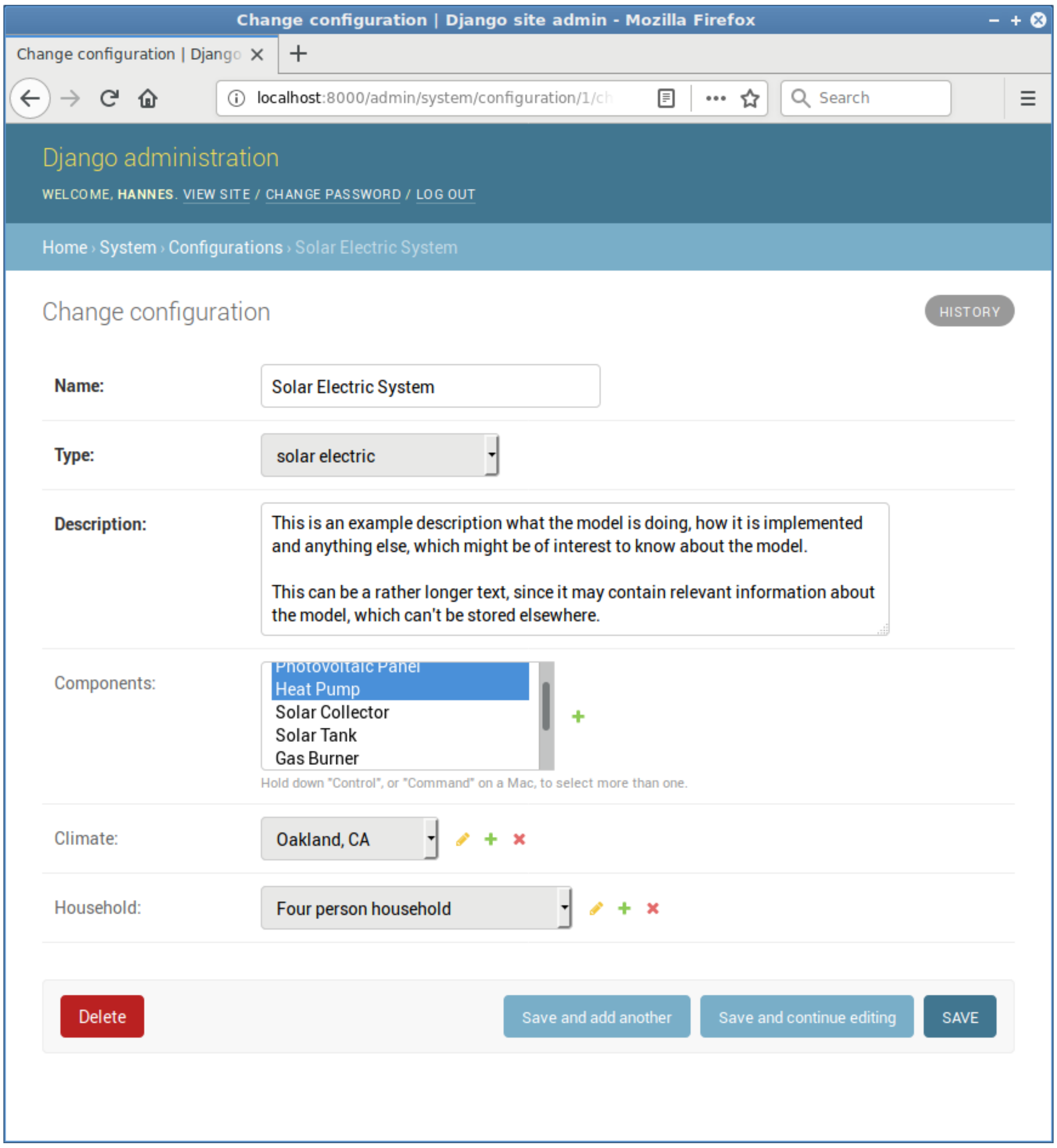
Name: Four person household

Occupancy: 4

☐ At home

Data:
```
[0.0, 0.0, 0.010589848867163115, 0.0, 0.0, 0.0, 0.004813567666892326,
0.026955978934597 02, 0.07509165560352027, 0.00064180902225231,
0.1527505472960498, 0.0, 0.0, 0.002888140600135395, 0.0, 0.0, 0.0,
0.007380803755901 5644, 0.00096271353378465, 0.01861246164531699,
0.000320904511126155, 0.0, 0.007701708267027 72, 0.0, 0.0, 0.0, 0.0,
0.00096271353378465, 0.0, 0.0, 0.0, 0.0, 0.0218215067565 7854
```

Figure D.3: *Django* administration page - Household


Change configuration | Django site admin - Mozilla Firefox
Change configuration | Django
localhost:8000/admin/system/configuration/1/ch
Search
Django administration
WELCOME, HANNES. VIEW SITE / CHANGE PASSWORD / LOG OUT
Home › System › Configurations › Solar Electric System
HISTORY
Delete
Save and add another
Save and continue editing
SAVE

Change configuration

Name:	Solar Electric System
Type:	solar electric
Description:	This is an example description what the model is doing, how it is implemented and anything else, which might be of interest to know about the model. This can be a rather longer text, since it may contain relevant information about the model, which can't be stored elsewhere.
Components:	[✓ Photovoltaic Panel] [✓ Heat Pump] Solar Collector Solar Tank Gas Burner
Climate:	Oakland, CA
Household:	Four person household

Hold down "Control", or "Command" on a Mac, to select more than one.

Figure D.4: *Django* administration page - Configuration

E Solar Electric System - Simulations

Table E.1: Annual totals of solar electric system for different scenarios

	A_pv = 6.67 m²		A_pv = 26.67 m²	
	T_limit = T_max	T_limit = T_max + 5 K	T_limit = T_max	T_limit = T_max + 5 K
	(1)	(3)	(2)	(4)
Solar Fraction [%]	34.30	42.50	57.30	85.70
Net Heat Demand [kWh]	4,606.57	4,606.57	4,606.57	4,606.57
Heat Pump Gain Delivered To Tank [kWh]	6,214.75	10,695.36	6,214.75	10,695.36
Backup Heat Delivered [kWh]	23.98	12.42	23.98	12.42
Heat Loss - Lower Tank Volume [kWh]	601.36	646.64	601.36	646.64
Heat Loss - Upper Tank Volume [kWh]	639.29	671.17	639.29	671.17
Tank Heat Delivered [kWh]	4,595.13	4,607.73	4,595.13	4,607.73
Tank Unmet Heat [kWh]	24.31	12.42	24.31	12.42
Dumped Heat [kWh]	366.53	4,756.65	366.53	4,756.65
Tank Balancing Error (Heat Rate) [kWh]	0.00	0.00	0.00	0.00
PV Generated Power [kWh]	1,612.76	1,612.76	6,448.63	6,448.63
PV Generated Power (DC) [kWh]	1,897.37	1,897.37	7,586.63	7,586.63
Total Heat Delivered [kWh]	6,238.73	10,707.78	6,238.73	10,707.78
Unmet Heat [kWh]	0.32	0.00	0.32	0.00
Energy Use - PV to Heat Pump [kWh]	1,015.89	1,608.96	1,696.81	3,242.48
Energy Use - Heat Pump [kWh]	4,002.98	8,793.94	4,002.98	8,793.94
Energy Use - Electric Resistance [kWh]	23.98	12.42	23.98	12.42
Energy Use - Electricity [kWh]	4,026.96	8,806.35	4,026.96	8,806.35
Energy Use - PV to Electric Resistance [kWh]	1.15	0.49	6.17	2.81
Available Electricity [kWh]	595.72	3.31	4,745.65	3,203.35
Temperature - Upper Tank Volume [°C]	65.87	68.47	65.87	68.47
Temperature - Lower Tank Volume [°C]	62.78	66.47	62.78	66.47
Temperature - Main Water [°C]	12.59	12.59	12.59	12.59
Temperature - Ambient Air (Dry Bulb) [°C]	13.75	13.75	13.75	13.75
Temperature - Ambient Air (Wet Bulb) [°C]	11.13	11.13	11.13	11.13

Table E.2: Solar electric system component parameters

Component	**Parameter**	**Value**
Inverter	$\eta_{dc,\,ac}$	0.85
Photovoltaic Panel	η_{pv}	0.15
	$A_{panel,\,eff}$	6.67 m^2 (1) / 26.67 m^2 (2)
	f_{act}[1]	1.00
	I_{ref}	1000.00 $\frac{W}{m^2}$
Heat Pump[2]	$COP_{hp,\,rated}$	2.77
	$Q_{hp,\,rated}$	1820.00 W
Electric Resistance Heater	η_{el}	1.0
	$Q_{el,\,nom}$	1700.00 W
Heat Pump Tank[3]	$tank\_size$	0.302833 $m^3(= 80.0\ gal)$
	f_upper_vol	0.50
	ins_thi	0.04 m
	$spec\_hea\_con$	0.04 $\frac{W}{mK}$
	t_tap_set	322.04 K
	h_vs_r	6.00
	dt_appr	2.00 K
	t_max_tank	344.15 K
Piping[3]	$pipe\_spec\_hea\_con$	0.03 $\frac{W}{mK}$
	$pipe\_ins\_thick$	0.006 m
	dia_len_slope	0.0002381 m
	dia_len_interc	0.0097619 m

[1] This parameter describes the fraction of the area of actual PV cells compared to the total PV panel area. $f_{act} = 1.00$ means that it is assumed, that there is no area lost (e.g. for mounting frames).

[2] For the coefficients, see Unit D in Table B.1 and B.2.

[3] The empiric models for these components have been implemented by the team at LBNL, so their parameters are not further explained in the scope of this thesis. These parameters have been assigned representative example values (unequal those in the final implementation). For further details see [1].

References

[1] *To be published as a CEC report on Community vs. Individual Scale Solar Water Heating (working title)*. Berkeley, CA: Lawrence Berkeley National Laboratory, 2019.

[2] *Python Documentation - What's new.* [Online]. Available: https://docs.python.org/3/whatsnew/3.7.html (visited on 02/05/2019).

[3] *Python Source Code - History.* [Online]. Available: https://raw.githubusercontent.com/python/cpython/master/Misc/HISTORY (visited on 02/05/2019).

[4] *Python Package Index.* [Online]. Available: https://pypi.org/ (visited on 03/18/2019).

[5] *The Incredible Growth of Python.* [Online]. Available: https://stackoverflow.blog/2017/09/06/incredible-growth-python/ (visited on 02/05/2019).

[6] *Medium - Why You Should Use Python.* [Online]. Available: https://medium.com/@mindfiresolutions.usa/python-7-important-reasons-why-you-should-use-python-5801a98a0d0b (visited on 02/05/2019).

[7] *Project Jupyter.* [Online]. Available: https://jupyter.org/ (visited on 03/12/2019).

[8] *PyCharm - The Python IDE for Professional Developers.* [Online]. Available: https://www.jetbrains.com/pycharm/ (visited on 03/12/2019).

[9] *Django Project.* [Online]. Available: https://www.djangoproject.com/ (visited on 03/14/2019).

[10] *Django Deployment Statistics.* [Online]. Available: https://www.djangosites.org/stats/ (visited on 03/18/2019).

[11] G. E. Krasner and S. T. Pope, "A Cookbook for Using Model-View-Controller User Interface Paradigm in Smalltalk-80," *Journal of Object Oriented Programming*, vol. 1, pp. 26–49, Aug. 1988, ISSN: 0896-8438. [Online]. Available: http://dl.acm.org/citation.cfm?id=50757.50759.

[12] *Django Documentation - FAQ.* [Online]. Available: https://docs.djangoproject.com/en/2.1/faq/general/ (visited on 02/11/2019).

[13] *Django Model View Template Pattern.* [Online]. Available: http://wiki.expertiza.ncsu.edu/index.php/CSC/ECE_517_Fall_2014/ch1a_8_os (visited on 02/11/2019).

[14] *Django Documentation - URLs.* [Online]. Available: https://docs.djangoproject.com/en/2.1/topics/http/urls/ (visited on 02/11/2019).

[15] *Django Documentation - Views.* [Online]. Available: https://docs.djangoproject.com/en/2.1/topics/http/views/ (visited on 02/11/2019).

[16] *Django Documentation - Templates.* [Online]. Available: https://docs.djangoproject.com/en/2.1/topics/templates/ (visited on 02/11/2019).

[17] *Django Documentation - Models.* [Online]. Available: https://docs.djangoproject.com/en/2.1/topics/db/models/ (visited on 02/11/2019).

[18] S. V. Raghavan, M. Wei, and D. M. Kammen, "Scenarios to decarbonize residential water heating in California," *Energy Policy*, vol. 109, pp. 441 –451, 2017, ISSN: 0301-4215. DOI: https://doi.org/10.1016/j.enpol.2017.07.002. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0301421517304329.

[19] M. Grahovac, "Modeling and Optimization of Energy Generation and Storage Systems for Thermal Conditioning of Buildings Targeting Conceptual Building Design," PhD thesis, Dec. 2012. [Online]. Available: https://www.researchgate.net/publication/325256005 (visited on 03/15/2019).

[20] M. Grahovac, P. Liedl, J. Frisch, and P. Tzscheutschler, "On-Off Boiler, Thermal Storage and Solar Collector: Energy Balance Based Model and its Optimization," Oct. 2011. [Online]. Available: https://www.researchgate.net/publication/265288877 (visited on 03/15/2019).

[21] M. Grahovac, "VC Chillers and PV Panels: A Generic Planning Tool Providing the Optimal Dimensions to Minimize Costs or Emissions," Nov. 2012. [Online]. Available: https://www.researchgate.net/publication/325256010 (visited on 03/15/2019).

[22] *Electric Utilities Service Areas.* [Online]. Available: https://www.energy.ca.gov/maps/serviceareas/CA_Electric_IOU.pdf (visited on 03/13/2019).

[23] *PGE - How to read your solar bill.* [Online]. Available: https://www.pge.com/en_US/residential/solar-and-vehicles/green-energy-incentives/solar-and-renewable-metering-and-billing/how-to-read-your-bill/how-to-read-your-bill.page (visited on 03/13/2019).

[24] *PGE - Electric Tariffs.* [Online]. Available: https://www.pge.com/tariffs/electric.shtml (visited on 03/13/2019).

[25] *Alterra Solar - True-Up Breakdown.* [Online]. Available: https://www.allterrasolar.com/making-sense-of-your-annual-bill-after-going-solar/ (visited on 03/13/2019).

[26] *PGE - True-Up Statement Explanation.* [Online]. Available: https://www.pge.com/includes/docs/pdfs/myhome/saveenergymoney/solarenergy/billing-callouts-trueup.pdf (visited on 03/13/2019).

[27] *PGE - NEM Process and Requirements.* [Online]. Available: `https://www.pge.com/en_US/for-our-business-partners/interconnection-renewables/simple-solar-wind/contractor-resources/standard-nem-process-and-requirements.page` (visited on 03/13/2019).

[28] B. Sparn, K. Hudon, and D. Christensen, *Laboratory Performance Evaluation of Residential Integrated Heat Pump Water Heaters.* Golden, Colorado: National Renewable Energy Laboratory, 2014.

[29] *Modelica Buildings Library - Photovoltaic Model.* [Online]. Available: `http://simulationresearch.lbl.gov/modelica/releases/latest/help/Buildings_Electrical_AC_OnePhase_Sources.html#Buildings.Electrical.AC.OnePhase.Sources.PVSimple` (visited on 02/26/2019).

[30] *Modelica Buildings Library.* [Online]. Available: `http://simulationresearch.lbl.gov/modelica/` (visited on 02/26/2019).

[31] *PVWatts Calculator.* National Renewable Energy Laboratory. [Online]. Available: `https://pvwatts.nrel.gov/` (visited on 02/27/2019).

[32] *System Advisor Model Version 2017.9.5 (SAM 2017.9.5).* Golden, CO: National Renewable Energy Laboratory. [Online]. Available: `https://sam.nrel.gov/content/downloads` (visited on 02/26/2019).

[33] *Introduction to Heat Pumps.* [Online]. Available: `https://www.automaticheating.com.au/heat-pumps-explained/` (visited on 02/28/2019).

[34] *Heat Pump Working Principle.* [Online]. Available: `https://commons.wikimedia.org/wiki/File:Heatpump2.svg` (visited on 02/28/2019).

[35] *Coefficient of Performance.* [Online]. Available: `https://us.grundfos.com/service-support/encyclopedia-search/cop-coefficient-ofperformance.html` (visited on 02/28/2019).

[36] *Typical Meteorological Year 3.* [Online]. Available: `https://rredc.nrel.gov/solar/old_data/nsrdb/1991-2005/tmy3/` (visited on 02/28/2019).

[37] *2016 ACM Supporting Content (weather data).* [Online]. Available: `https://www.energy.ca.gov/title24/2016standards/ACM_Supporting_Content/` (visited on 02/28/2019).

[38] R. Stull, "Wet-Bulb Temperature from Relative Humidity and Air Temperature," *Journal of Applied Meteorology and Climatology*, vol. 50, pp. 2267–2269, Nov. 2011. DOI: `10.1175/JAMC-D-11-0143.1`.

[39] *Electric Resistance Heating.* [Online]. Available: `https://www.energy.gov/energysaver/home-heating-systems/electric-resistance-heating` (visited on 02/28/2019).

[40] *Bootstrap - A Front-end Component Library.* [Online]. Available: `https://getbootstrap.com/` (visited on 03/10/2019).

[41] *Django Documentation - Settings.* [Online]. Available: `https://docs.djangoproject.com/en/2.1/ref/settings/` (visited on 03/14/2019).

[42] *Django Documentation - Cryptographic Signing.* [Online]. Available: `https://docs.djangoproject.com/en/2.1/topics/signing/` (visited on 03/14/2019).

[43] *Django Documentation - Template Language.* [Online]. Available: `https://docs.djangoproject.com/en/2.1/ref/templates/language/` (visited on 03/14/2019).

[44] *Python Documentation - Pickle Module.* [Online]. Available: `https://docs.python.org/3/library/pickle.html` (visited on 03/11/2019).

[45] *Plotly.* [Online]. Available: `https://plot.ly/` (visited on 03/12/2019).

[46] *Plotly Python Open Source Graphing Library.* [Online]. Available: `https://plot.ly/python/` (visited on 03/12/2019).

[47] *Plotly Javascript Open Source Graphing Library.* [Online]. Available: `https://plot.ly/javascript/` (visited on 03/12/2019).

[48] *Plotly Chart Studio.* [Online]. Available: `https://plot.ly/create/` (visited on 03/12/2019).

[49] *Anaconda Project.* [Online]. Available: `https://www.anaconda.com/` (visited on 03/12/2019).

[50] *Anaconda Project- Documentation.* [Online]. Available: `https://docs.anaconda.com/anaconda/` (visited on 03/12/2019).

[51] *DigitalOcean.* [Online]. Available: `https://www.digitalocean.com/` (visited on 03/14/2019).

[52] *DigitalOcean - How To Set Up Django with Postgres, Nginx, and Gunicorn on Ubuntu 18.04.* [Online]. Available: `https://www.digitalocean.com/community/tutorials/how-to-set-up-django-with-postgres-nginx-and-gunicorn-on-ubuntu-18-04` (visited on 03/14/2019).

[53] *Django Documentation - User Authentication.* [Online]. Available: `https://docs.djangoproject.com/en/2.1/topics/auth/` (visited on 03/16/2019).

[54] *Django Documentation - Deploy with WSGI.* [Online]. Available: `https://docs.djangoproject.com/en/2.1/howto/deployment/wsgi/` (visited on 03/16/2019).

[55] *Django Documentation - Deployment Checklist.* [Online]. Available: `https://docs.djangoproject.com/en/2.1/howto/deployment/checklist/` (visited on 03/16/2019).

[56] *Django Documentation - Testing.* [Online]. Available: `https://docs.djangoproject.com/en/2.1/topics/testing/` (visited on 03/16/2019).