Individual Progress Report

By

Nikhil Mathew Sebastian

Team 10

October 2021

Contents

1 Introduction

My team (Team 10) has taken up the idea sponsored by Professor Arijit Banerjee, which is to build an advanced interface box for the research solar panels on the roof of the ECEB building. The goal of this interface box is to add a layer of protection to the currently loosely monitored panels, while also providing an interface through which researchers can remotely access solar panel data and remotely configure the monitored solar panels and hence allow for a larger research potential of these panels. The end solution for this project will be a physical box that will be placed on each solar panel, and then to have a central, external, wireless portal through which all observations and configurations can be relayed.

As the lone computer engineer in Team 10, I have taken up sole responsibility of almost all of the software aspects of our project. This includes 3 key components – setting up a front-end user-facing portal called "Research Hub" through which researchers can access solar panel data and configure observable panel sections, establishing an external server for the back-end of the portal which can send panel configuration settings and receive real-time panel data wirelessly and also have it connected to the portal and the interface box set-up to accomplish this, and finally the programming of our on-board microcontroller which is the "brain" of our interface box to allow for user interaction and real-time updates.

While I have contributed design perspectives of software restrictions and high-level details of our electronics side and the PCB, the large majority of my work and contributions are through the functionality of the 3 aforementioned components. To ensure that work moves forward and that isolated roadblocks for any one component does not slow down the entire software architecture, I am working on all 3 of these components parallelly. This also works best for testing and integration due to the "build up"/ layered structure of all these components together. The status of the 3 components are as follows:

1. The structure of the front-end user portal is completed, and what is left to be done is improve on the bare-bones visuals, add the planned data visualization, and connect the back-end databases for real-time data updates and configuration updates.
2. While the design of the back-end server and the resources for the same have been identified (planning completed), some of the implementation is left to be done because I need more detail from the microcontroller side and that was delayed since we did not have our ESP-32 board. I have the code structure and implementation scheme ready to go with this in mind.
3. Due to the delay in obtaining the ESP-32 microcontroller there has been no physical programming of the board, but here also I have the code structure and implementation scheme ready to go to set the board up. *(Note: we obtained the board a few days ago so this is the immediate next/current task)*

2 Design

2.1 Changes since Design Review

Following our Design Review session and Design Document review, the only real change to the plan for our software aspects of the project was to change the original 3-prong approach that included building the front-end GUI and back-end server mostly from scratch, and instead utilize the Django framework to build these two components. This is because Django already has many available libraries and dependencies that make it easier to develop a more sophisticated solution while also not having to worry about overheads with too much granularity.

Because of this, the first core task that was taken up was figuring out the Django framework (further detailed in Section 2.2), since I had lost some time researching and working on the original approach.

2.2 Django Framework: "Research Hub" Portal

2.2.1 Panel Data and User Interface

The most important part of the user-facing side of our project and its software is how the user will be able to interact with the observed solar panel data, the layout of multiple solar panels (since scalability is a key requirement), and how the user can input different configuration settings.

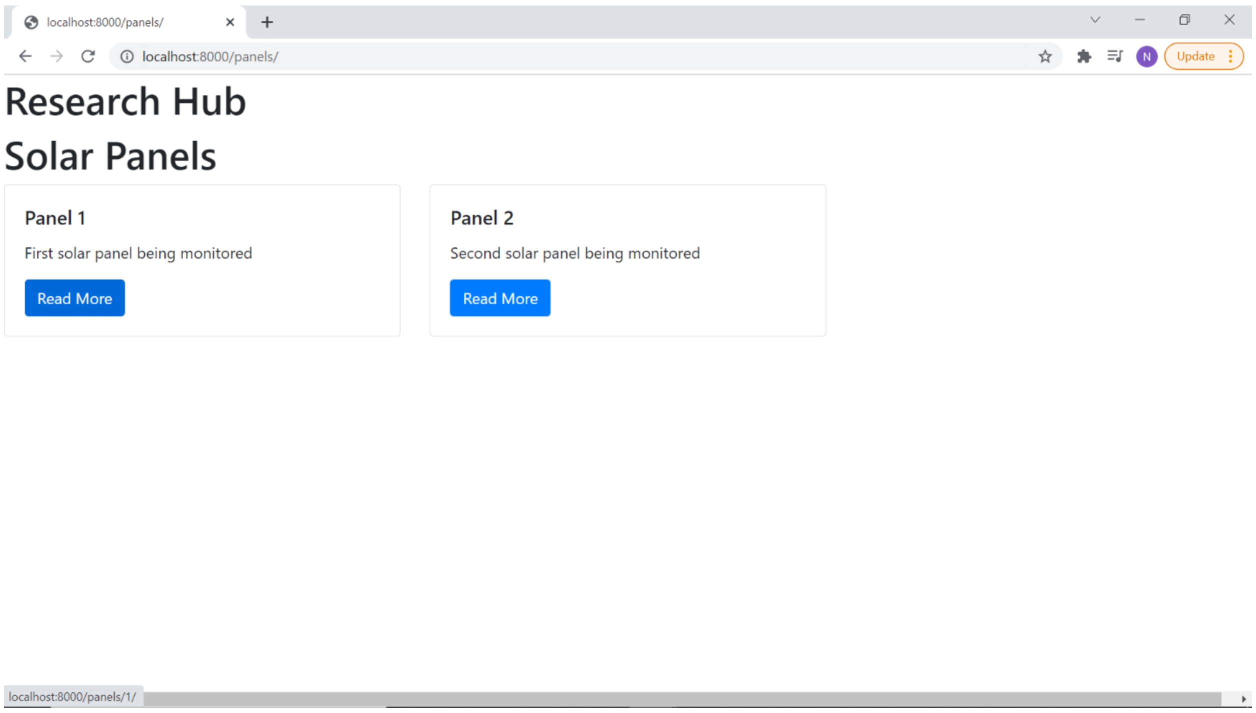
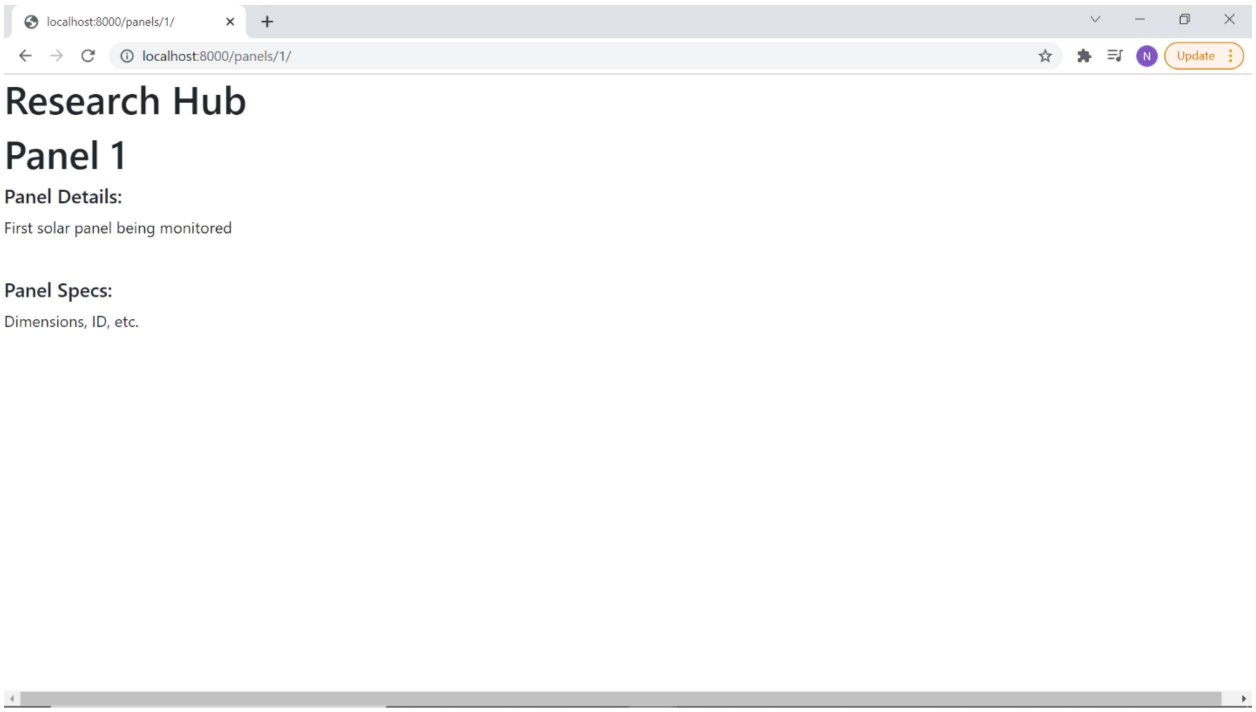


Figure 1: Screenshot of the current base page for navigating to monitored solar panels

In the entire Django project, an application called "panels" is dedicated to this interaction with panel-specific data and settings. Figure 1 shows the working base for the first landing page for users to access solar panel data. From a design point of view, the goal was to isolate panel-specific data because the scalability should eventually allow for viewing all 60 available research solar panels. This is why this initial landing page has been thought of as a GUI "page of contents" from where the user can navigate to specific monitored panels.

To build these webpages and to be able to store data to display on a website, we normally need databases and a database management system made through a SQL back-end to store and edit data. But Django abstracts this through its built-in Object Relational Mapper (ORM). An ORM is a program that allows you to create classes that correspond to database tables - class attributes correspond to columns, and instances of the classes correspond to rows in the database. When you're using an ORM, the classes you build that represent database tables are referred to as "models", and in Django they live in the *models.py* module of each Django application (in this case "panels). [1]

In my "panels" application, I built a model for the base of each solar panel's data in the corresponding *models.py*. I then differentiated the above "contents" layout and the panel-specific layout given below by adding HTML files for each specific model and isolating them by URL. All of this was done in the project *views.py* module.



Research Hub

Panel 1

Panel Details:

First solar panel being monitored

Panel Specs:

Dimensions, ID, etc.

Figure 2: Screenshot of current dedicated page for panel-specific data and settings

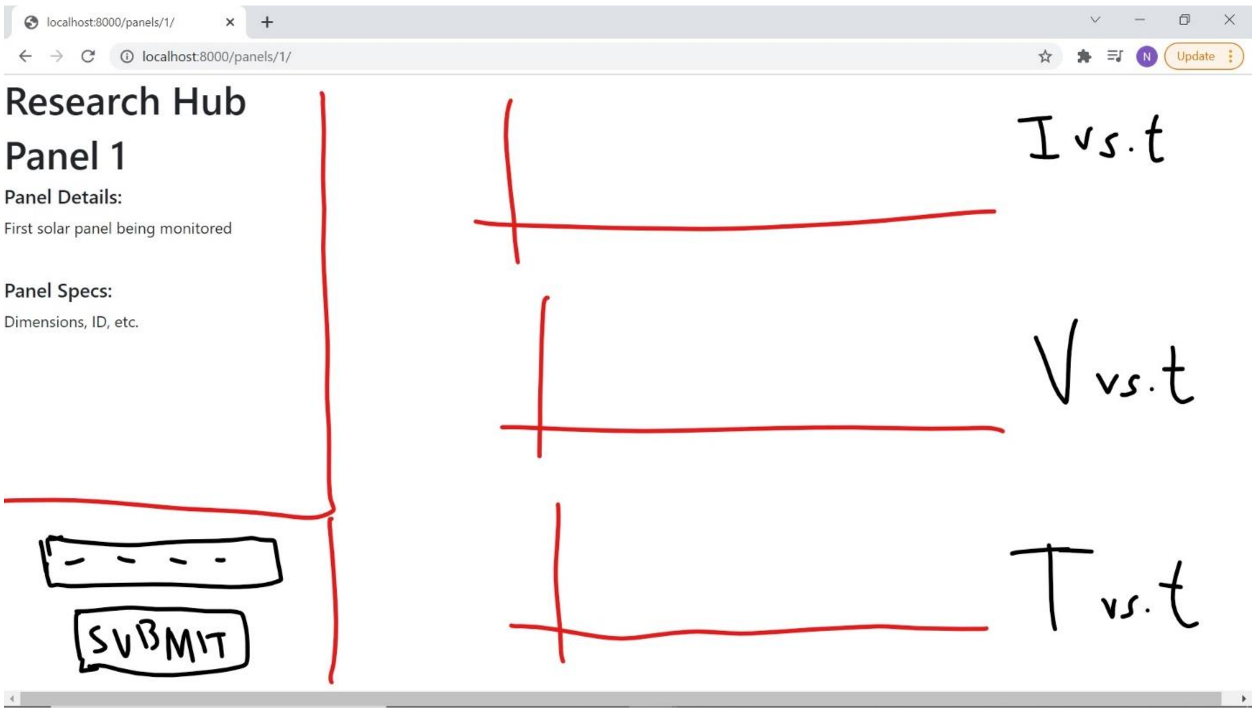


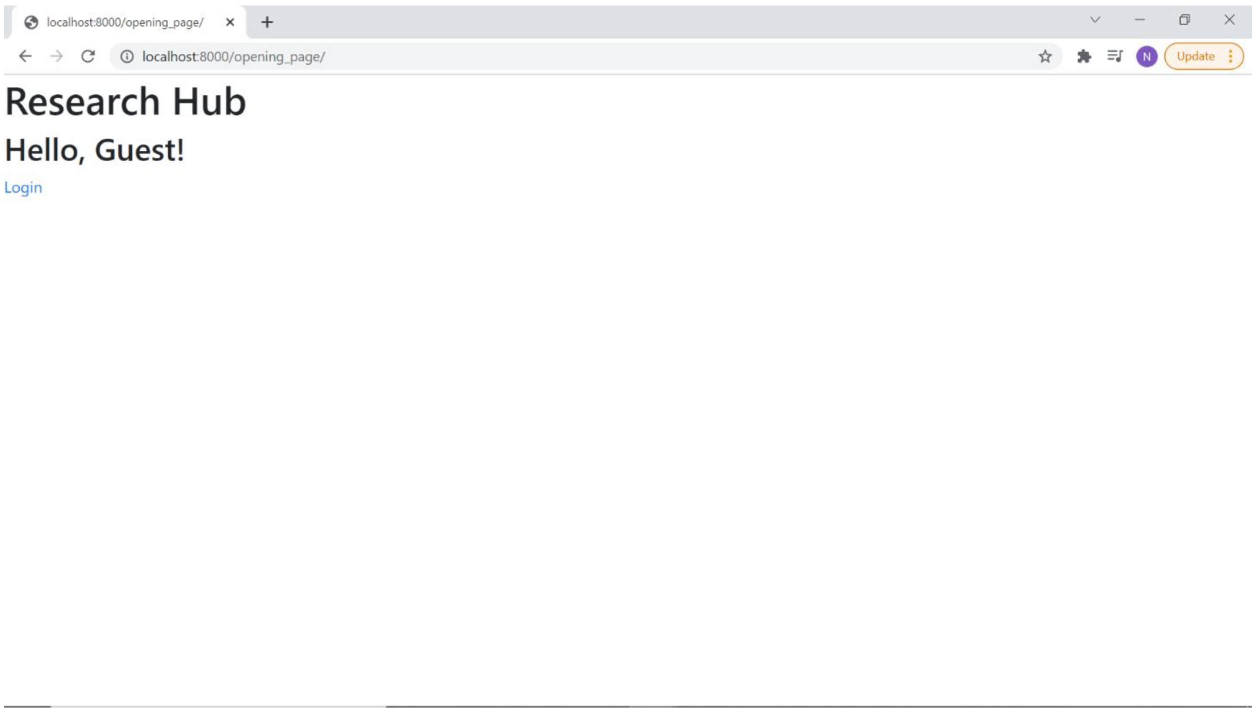
Figure 3: Rough drawn-over design of how visualizations and configuration option will be added

As seen in Figure 2, the current follow-up page dedicated to individual solar panels does not include a lot of information, mainly due to the lack of functioning data from the interface box – both monitored data and a channel for configuration input. This is further unsupported since the back-end server is not fully set up either. Rather than add dummy images and drop-boxes, I thought it better to have a cleaner base on which I can layer GUI elements. Figure 3 provides a rough overview of what the finished product would look like – additional details regarding each panel can be added near where data is currently shown, the real-time monitoring of the 3 critical solar panel data outputs will be plotted in a time-series manner using a majority of the space on the right, and the bottom-left of the page can have the configuration settings as a dropdown menu with the 3 possible options (will update graphs in real-time). Note that I have already completed the set-up and import for Bootstrap CDN in the *base.html* template for the portal website. Bootstrap will allow for easy and sleek styling of these simple webpages rather than manual CSS scripting. [2]

Due to known recurring issues with project graphing software like GraphViz [3], I was not able to produce a module-level/application-grouped graph of the Django framework built so far. But I could create a visualization of the module-file tree for my Django project, which can be seen in Appendix A. I also generated a .dot file with the Dot digraph of the Django project, which can be seen in Appendix B.

2.2.2 Security: Login/Logout Features

The other key aspect of the Research Hub portal is of course security. Considering that safety is one of the motivations behind this project, we also need to make sure that this portal, that gives away research information and also the ability to edit configuration setting, can only be accessed by verified ECE personnel. With this in mind, I was able to develop the following basic layout for a login access approach:

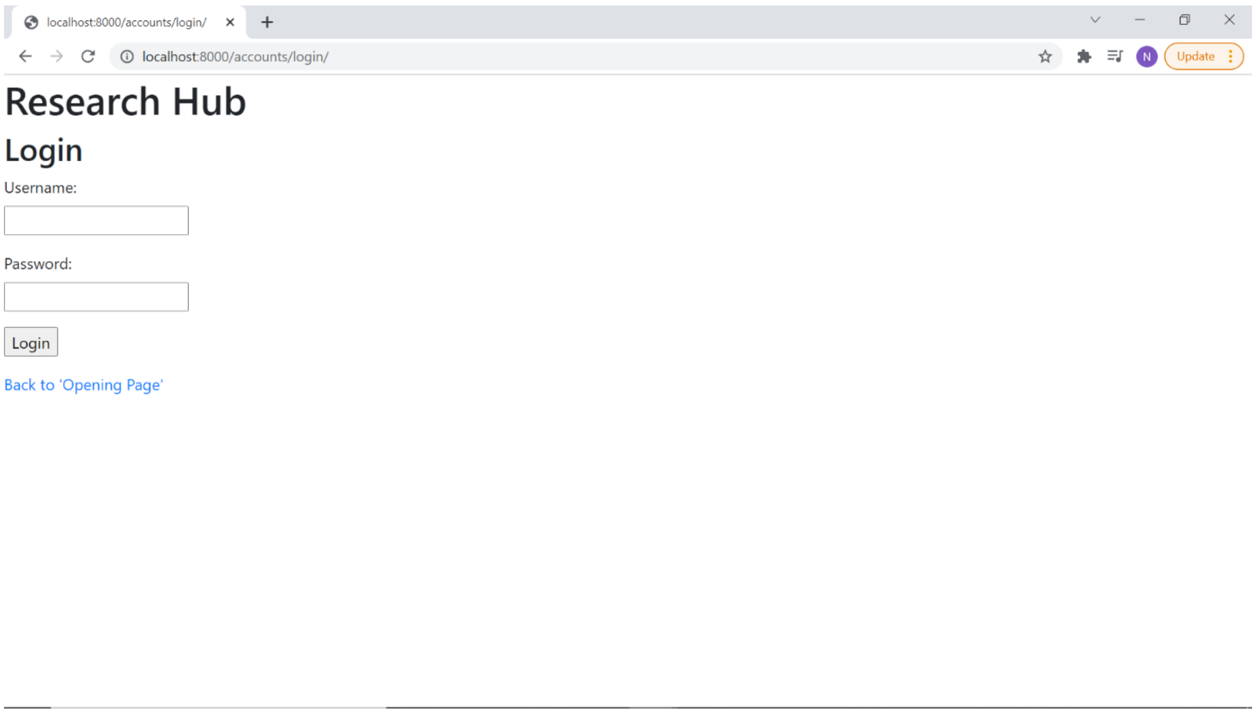

localhost:8000/opening_page/

Research Hub

Hello, Guest!

Login

Figure 4: Screenshot of the basic Welcome page that prompts login


localhost:8000/accounts/login/

Research Hub

Login

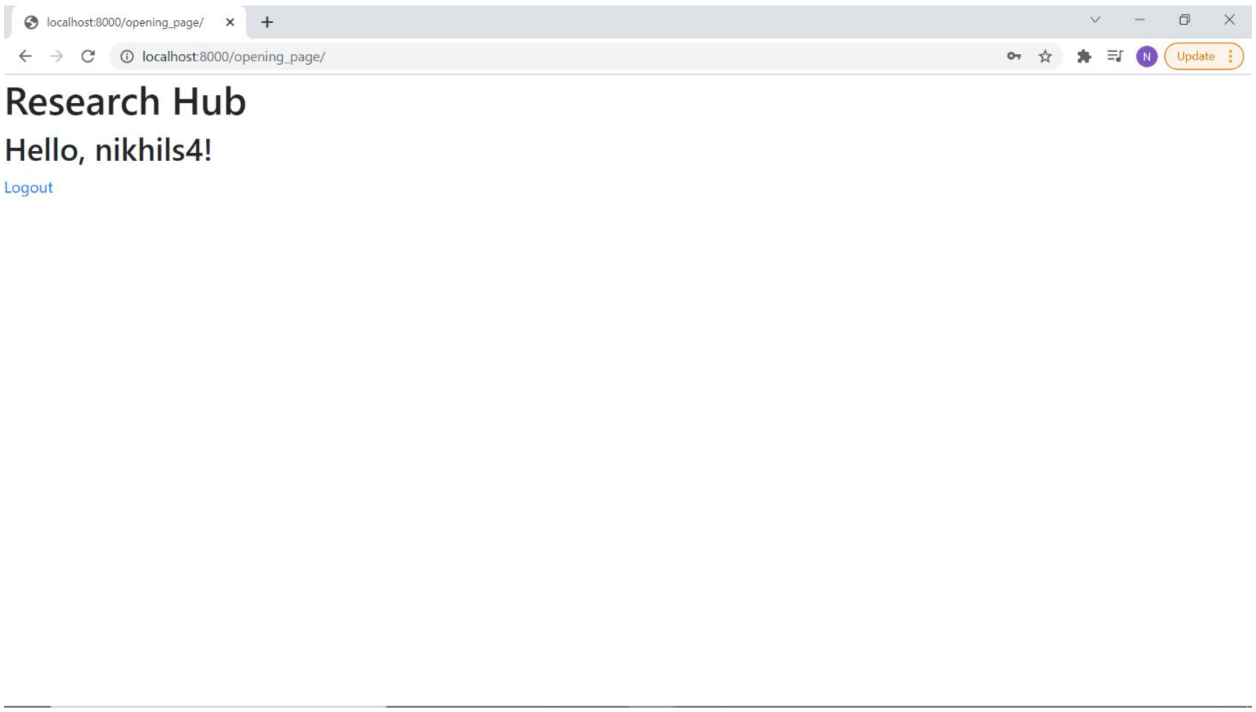
Username:

Password:

Login

Back to 'Opening Page'

Figure 5: Screenshot of the Login page itself


localhost:8000/opening_page/

Research Hub

Hello, nikhils4!

Logout

Figure 6: Screenshot of confirmation page that displays username (and allows for logout)

To allow for modular testing from a design point of view, this account accessing application ("users") is separated from the "panels" application from Section 2.2.1. This is why I put together the simple HTML script to allow for the username to be displayed on the landing page if the authentication was successful (elaborated on in Section 3.1/Verification). After connecting the application to it's configured *views.py* file and the base webpage, the rest of the login was achieved by adding the URLs provided by the Django authentication system [4]. This gave me access to URLs like "accounts/login/" and "accounts/logout/" and also has URLs for password resets, etc., all of which is integrated from a URL point of view. After this I was able to write the HTML script for the logging-in page, as well as tweak the confirmation page to allow for circling back (easy verification) and add the logout functionality. An important aspect of the current login sequence is that it accounts for CSRF tokens (protection against the most common web attack – CSRF), hence setting up the start of the web security countermeasures that I need to implement. Django also has documentation for encryption and other security measures, which I will build on top of the existing project along with the mentioned account management options.

Another aspect of the Django framework that was explored through this was the Django superuser or it's administration settings. As the owner I was able to create this user with access to forms and tables throughout the project. It is using this that I have currently set up account access for just the super user/myself (username: nikhils4), but the admin platform allows for easy and direct editing of forms and databases which will be crucial as we move on to more complicated tasks like managing a list of authorized users. [5]

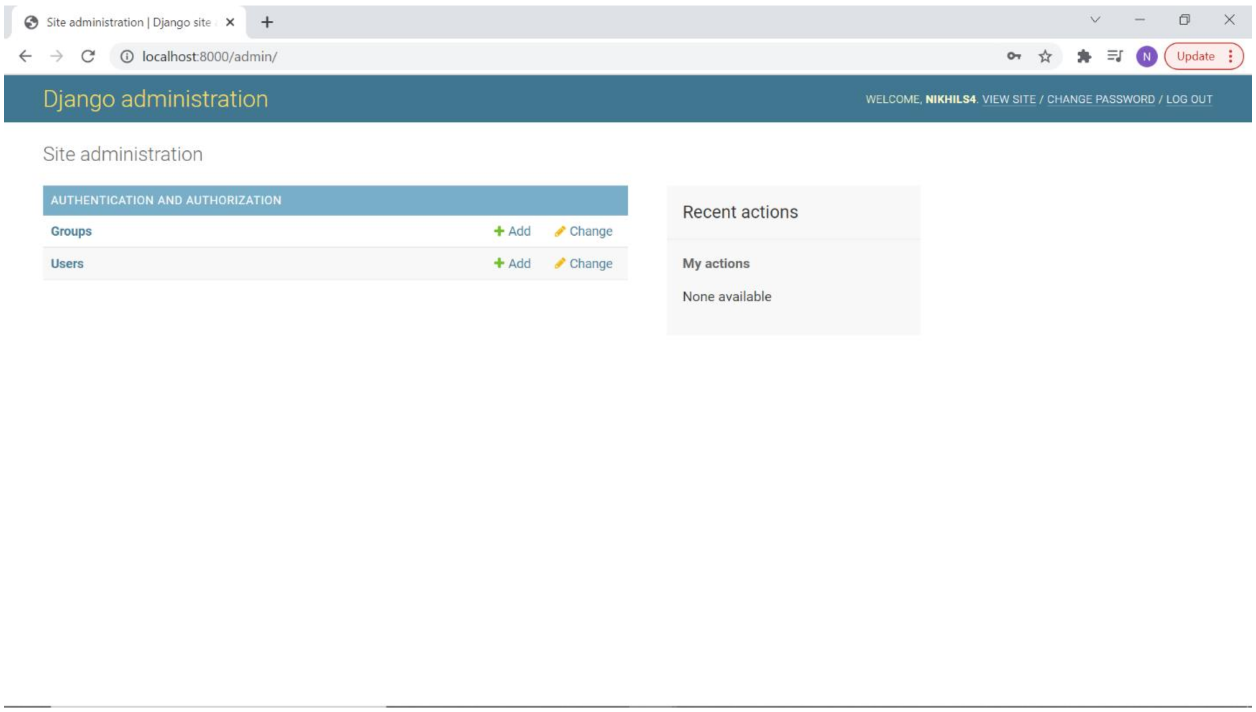


Figure 7: Django superuser/admin home page and its options

At the moment this application is isolated from the "panels" application, while the working state would be that it is the "gateway" to solar panel data. As aforementioned, this has been done for development isolation and modular testing and it will be integrated as the website is built more, especially with regards to completing the configuration of the back-end. Additionally, to ensure an airtight security policy, I will not be providing the option to "create an account" – rather, an offline request for access can be approved by the superuser through administration.

Note again that I will use Bootstrap CDN to improve on the simple visuals of the webpages so far as I add functionalities. And once again I could create a visualization of the module-file tree for my Django project, which can be seen in Appendix A. I also generated a .dot file with the Dot digraph of the Django project, which can be seen in Appendix B.

2.2.3 Back-end: External Server

At the moment, the majority of the back-end has not been directly dealt with, but this is not to say that the basis of it has not been completed. Due to the structure of the Django framework and its in-built ORM capability, every Django application comes with the ability for database management through models and *models.py*. We can even see this in the module-file tree for the Django project in Appendix A – Django creates the main *db.sqlite3* file which is a database file that stores all the data being generated during project/server execution. This coupled with the ORM means we can simply add a REST framework skeleton to the Django project to allow for alignment with HTTP pathways and our back-end will be more or less finished. Note that the reason for the back-end not being fully covered is directly because it needs the Django front facing modules to be good to go before we build on the ORM and databases, otherwise it will be extremely tedious to work around it and backtrack.

That being said, once we complete the functionalities themselves on the front-end of the Django project, we will move forward with the back-end configuration using the Django REST framework, which builds on Django using the REST API for HTTP pathways (which I have experience with). I will then follow the given implementation scheme:

1. Create Serializer: this file will take database data and restructure it to work more easily and efficiently with higher layers of our applications/project like the frontend. For example, the frontend we will create expects the data returned to it from a request to be in a specific format (like JSON or XML) [6]. It will also serve as the API endpoint for REST.
2. Create the Views: by editing the corresponding *views.py* file, we will allow for HTTP requests to be taken in and for responses to be sent out, and vice versa.
3. Create URLs: update the corresponding *urls.py* to allow for valid URLs to be mapped during HTTP communications. (At this point the REST API functionality can pass around the front-end successfully)
4. Options for CORS: we can also configure and allow for Cross-Origin Resource Sharing. While we will not need this specifically since all of our data will be locally hosted or at least a part of the same base webpage, this will allow for future improvements and developments on this project or based on this project.

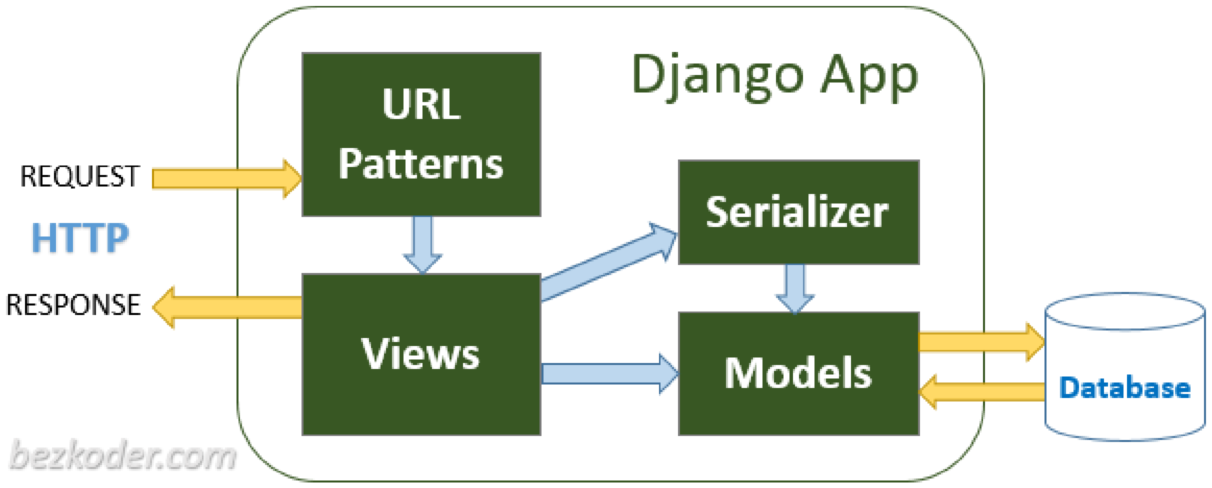


Figure 8: Visualization of the back-end interaction once fully complete [7]

Following this, to allow for a full integration with the front-end, we can use some JavaScript library (say Node or Ember) to have a front-end client. But this can be minimized to the bare requirements as almost all of the communication in our wireless network will be through POST requests to and from the microcontroller and this back-end, with both acting as client-servers.

2.2.4 Public Access: Web Server

The main goal of our project is to have the interface box and remote “Research Hub” ready for researcher use in the ECEB, i.e., we will have a successful solution even without this portal being hosted on the Internet for public access. Nevertheless, this is also a large “additional objective” that we have and which I think I can successfully achieve. This would allow for a more collaborative portal and also for showcasing the interface publicly.

Briefly, this can be achieved by hosting the Django project on a web server. First, we would need to edit production variables in the project – like DEBUG and SECRET_KEY – to allow for successful deploying. We can then host the website on the cloud through a PaaS service like Heroku – after some database configuration and update settings, PaaS Services can handle the rest of the web infrastructure, allowing for our website to run as a stand-alone!

2.3 Microcontroller Programming

Even with the back-end server and front-end webpages working, it is only with wireless communication from the interface box and it's observed values that this project is of any value. It also needs to allow for two-way wireless communication and the ability to send configuration requests for panels to the interface box for processing. This is achieved through the "brain" of our interface box – the ESP-32 microcontroller – by leveraging it's Wi-Fi capability and making it a web client-server.

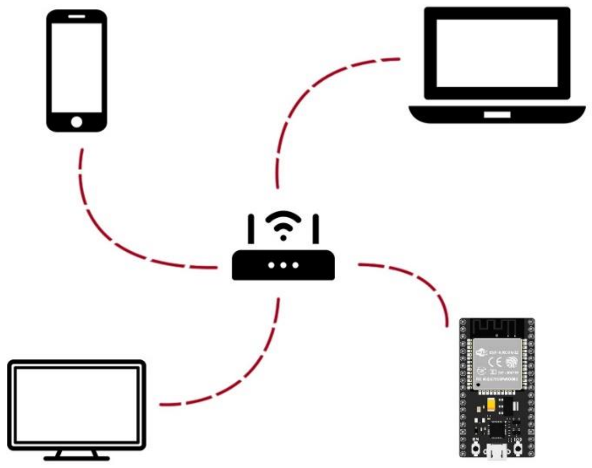


Figure 9: Visualization of network capabilities given a router, electronic receiver, and the ESP-32 in STA mode [8]

The ESP-32 microcontroller can do this through it's Station (STA) Mode – the board that connects to an existing Wi-Fi network (one created by a wireless router) is called a "Station" (STA). In STA mode, the ESP-32 gets its IP from the router it is connected to. With this IP address, it can set up a web server and deliver web pages to all connected devices under the existing Wi-Fi network. [9] In this state the ESP-32 can be configured to both send and receive HTTP requests, and so fulfill our requirement of being able to send and receive periodic POST requests. The high-level implementation scheme that I will follow to set this up through the Arduino IDE is below:

1. We will utilize the *Wifi.h* and *WebServer.h* libraries to implement this

2. Initial setup includes: providing Wi-Fi credentials (SSID and password), setting the port (usually 80), and starting the connection using *WiFi.begin()*
3. Turn the server on and direct it to a HTML handler function: *server.on("/", <handler>)*
4. The structure of the handler function itself will simply invoke the HTML that we define in the script body and return the "200" OK code on completion
5. The HTML itself can be structured as needed based on our data processing
6. Set the communication IP address of the other client-server (Django back-end) so that URL formatting can be used to send and receive POST requests
7. The core HTTP requests and commands will be constantly looped to the Django client-server – these can be made periodic by default but will need editing based on PCB data inflow for real-time data updates

Note that I have not gone into the specifics of constructing POST requests and how to handle received POST inputs either. Once again, this needs some analysis of the data coming in from the hardware side as well as how to parse data to the hardware side of our project. This will be built on as there is progress from my other teammates.

As mentioned in the introduction, since we have finally received the ESP-32 board, this programming is our primary objective so that the rest of the software-side of the project is not held back in any way.

3 Verification

3.1 Django Framework: “Research Hub” Portal

From a verification point of view, the entirety of the Django framework can be split up into two types of functionalities – GUI/user-input based and server-side communication based. So, all the verifications for this component are as follows:

GUI/User-Input Verifications:

Requirement	**Verification**
As per design choice, there is an “index” page for monitored solar panels that works (Complete)	1. Navigation to “/panels” section of the webpage shows the index of monitored solar panels successfully 2. Clicking on any specific solar panel choice redirects user to correct dedicated page successfully
Verification that ORM and Django databases reflect data updates (Complete)	1. Update *models.py* in the “panels” application in anyway (and then complete migration), OR, open ORM shell and update/add panel data 2. Successful if any changes within the project is reflected on the corresponding dedicated solar panel page
Basic functionalities of a webpage are successfully implemented (Complete)	Checks: 1. Arrow clicks to go backward and forward from the current page should work as logically expected 2. Reloading the current page should show no change to static values/elements 3. Direct typing into the URL before or after opening a part of the webpage should redirect you to the correct webpage (URL mapping)
Clicking “Submit” for a configuration change for a specific solar panel should trigger this change	1. Choose an option other than the current configuration and click submit on a specific panel’s webpage 2. The displayed configuration setting should immediately change 3. A POST request should be created with the body corresponding to the updated configuration settings in real-time in the back-end of the “panels” application

	4. Nothing should happen if the configuration chosen is the same as the current one
Authentication for registered users is successful to enter portal (Complete)	1. Try to login with a saved username and password on the webpage 2. Successful if it redirects you to a confirmation page with your username
Login and Logout options/redirects are accurate and successful (Complete)	1. Clicking "Login" on the entry/opening page should redirect the user to a functioning login form 2. Clicking "Logout" once already logged in should log the user out and redirect them to the original entry/opening page
Login page is mapped correctly with regards to the portal and other applications	1. Accessing any part of the website without logging in first should redirect the user to the login form 2. After logging in with authentication, user should be directed to the index of monitored solar panels successfully 3. Logout option should be moved and presented at the panels' index
Password of an accepted user can be changed by the user	1. Offer option of changing password along with "Logout" options for authenticated, logged-in users 2. Redirect this choice to a "New Password" form that updates their password in the database and immediately logs them out on submission

Note: The working website and server execution is a completed activity in itself even if it does not have a specific verification. Similarly, pending tasks of improving webpage style through Bootstrap CDN and adding the GUI elements of the data visualizations and configuration selection panel don't have verifications other than visual but still need to be completed.

Server-side Communication Verifications:
(Note: None have been completed as server-side HTTP work is not complete as aforementioned)

Requirement	**Verification**

Can send POST requests successfully	1. Send a POST request from the Django back-end to the Postman API platform – hardcoded HTTP commands 2. Successful if POST is received and readable on Postman platform
Can receive POST requests successfully	1. Using Postman, send a POST request to the Django back-end's IP address (through API) 2. Successful if POST is received and readable at server-level
Can send POST requests to the ESP-32 board successfully	1. Send a POST request to the ESP-32 client-server – hardcoded HTTP commands 2. Successful if POST is received and readable at board-level
Can receive POST requests from the ESP-32 board successfully	1. Send a POST request from the ESP-32 client-server to the Django back-end 2. Successful if POST is received, readable, and stored in Django databases
Can update panel specific data in real-time from POST requests	1. Receive 1 minute of data packets of real-time information from the interface box wirelessly 2. Check through database logs for past 1 minute after completion 3. Successful if continuous stream of packets recorded and "panels" database updated with the same a. Data also needs to be parsed into the 3 key values to be successful
Can submit a configuration specific to the ESP-32 board	1. Send 1 configuration settings change for 1 solar panel to the interface box wirelessly 2. Successful if request is parsed by the board and relayed (acknowledgement)

3.2 Microcontroller Programming

The microcontroller also has two types of functionalities to verify from its design point of view – on-board network status and HTTP communication with the Django framework. So, all the verifications for this component are as follows:
(Note: None have been completed since the board has only just been received)

Network Status Verifications:

Requirement	**Verification**

Can detect and confirm Wi-Fi connection	Code snippet between starting the Wi-Fi connection and moving on to the server:
	while (WiFi.status() != WL_CONNECTED) {
	  delay(1000);
	  Serial.print(".");
	}
	Serial.println("");
	Serial.println("WiFi connected successfully");

Can verify IP address after successful connection to Wi-Fi	Code snippet after Wi-Fi connection verification snippet:
	Serial.print("Got IP: ");
	Serial.println(WiFi.localIP());

Can verify that turning on and starting of the server happened successfully	Code snippet immediately after turning on the server:
	server.begin();
	Serial.println("HTTP server started");
	delay(100);

Can complete HTTP transfer of HTML scripting successfully	1. I have included the "200" OK code to be processed and returned on HTTP requests 2. Successful completion will print/display the success code

HTTP Communication Verifications:

Requirement	**Verification**
Can send POST requests successfully	1. Send a POST request from the board to the Postman API platform – hardcoded HTTP commands 2. Successful if POST is received and readable on Postman platform
Can receive POST requests successfully	1. Using Postman, send a POST request to the board's IP address (through API) 2. Successful if POST is received and readable on the board
Can send POST requests to the Django framework successfully	1. Send a POST request to the Django back-end client-server – hardcoded HTTP commands 2. Successful if POST is received, readable, and stored in Django databases
Can receive POST requests from the Django framework successfully	1. Send a POST request from the Django back-end client-server – explicit request from a Django application 2. Successful if POST is received and readable on the board

Although not listed here, the microcontroller will also need testing/verification for the connection between PCB functionalities (switching subsystem, data paths, etc.) on the hardware side. My teammates handling the board-level granularities will be handling this too, although I may end up helping with some programming specifics.

3.3 Schedule

Building on the schedule we put together for the Design Document, the following timeline for the rest of the semester has been planned for verification keeping in mind the status so far:

Timeline	**Completed Verifications**
End of Week of 11/01	All Django GUI/user verifications except configuration server trigger, 3 of 6 Django server-side verifications, All microcontroller network verifications, 2 of 4 microcontroller HTTP verifications
End of Week of 11/08	All remaining verifications

Note: Especially for the verifications and tasks for the week of 11/08, I will require the hardware and electronic specifications to be working to complete my objectives

4 Conclusion

So far, I believe I have contributed extremely well to my team considering the resources available. We were unfortunately delayed in submitting our PCB order, and this is sure to affect our planning as the project timeline comes to a close. Similarly, we were also only able to receive our microcontroller board just a few days ago, which directly impeded my ability to program it as I planned. But, all things considered, I have kept moving forward and making progress wherever possible, be it coding or research in preparation or something else.

For the project as a whole, I believe my partners and I have a very amicable workload split. From a grading and functionality point of view, one might argue that I myself shouldering the heavily weighted software aspects (all of them) might be more of a workload. But at the same time, I have seen over the past few weeks that electronics and hardware complexities are extremely time-consuming and confusing, so it could be argued that one person can't shoulder that effort. But that workload is split between both of my teammates! With all of this in mind, I think so far no one in my teammate has felt that they got the short end of the stick, and I am confident that we can work together to make a brilliant project over the final weeks to come!

As we move towards the end of the semester, I do have to accept that some aspects of our project - especially things like the PCB completion status – are behind schedule. But I can't get hung up on this because we still have time, and I am sure that if we put our best efforts we will be able to put together a great project! Looking at Section 3 (Verifications), it is clear that there is work I have to do but I firmly believe I can manage the timeline I have presented (Section 3.3). I think to deliver our project as we initially planned is not overly ambitious but will require good amount of effort from all team members, but we have already discussed this and have put time aside to ensure we succeed.

I can't wait for the remainder of the semester so that we as a team can validate our time and efforts and showcase a great final project!

4.1 Ethical Considerations

Following the IEEE and ACM Codes of Ethics, I believe the following considerations are just a few that I have to hold myself to considering what has been described in this document:

- To be accepting of honest criticism from course staff so that our project comes out in its best form, and be honest in my claims and expectations considering the work at hand [10]
- Remember that this project has a great potential for positive impact after we complete it, and make sure that contributing to society and human well-being stays a core principle of our team [11]
- To honor confidentiality and respect privacy, especially since we are part of a larger research effort through Professor Banerjee and we will be privy to confidential research data through our project [11]
- And, to ensure that I hold not just myself but my teammates as well to our code of ethics, especially as the pressure rises and we reach the close of our project [10]

5 Citations

[1] K. McLaughlin, "An introduction to the Django ORM," *Opensource.com*, 24-Nov-2017. [Online]. Available: https://opensource.com/article/17/11/django-orm. [Accessed: 30-Oct-2021].

[2] J. Thornton and M. Otto, "Bootstrap," *Bootstrap · The most popular HTML, CSS, and JS library in the world.* [Online]. Available: https://getbootstrap.com/. [Accessed: 30-Oct-2021].

[3] "Issues installing pygrahviz 'Fatal error c1083: Cannot open include file: 'graphviz/cgraph.h': No such file or directory,'" *Stack Overflow*, Jan-2020. [Online]. Available: https://stackoverflow.com/questions/59707234/issues-installing-pygrahviz-fatal-error-c1083-cannot-open-include-file-graph. [Accessed: 30-Oct-2021].

[4] "Using the Django authentication system," *Django*. [Online]. Available: https://docs.djangoproject.com/en/3.2/topics/auth/default/. [Accessed: 30-Oct-2021].

[5] "Writing your first Django app, part 2," *Django*. [Online]. Available: https://docs.djangoproject.com/en/1.8/intro/tutorial02/. [Accessed: 30-Oct-2021].

[6] T. Christie, "Serializers," *Serializers - Django REST framework*. [Online]. Available: https://www.django-rest-framework.org/api-guide/serializers/. [Accessed: 30-Oct-2021].

[7] bezkoder, *Django Rest Api Architecture*. BezKoder, 2021.

[8] M. Akbari, *Station Mode*. ElectroPeak.

[9] "Wi-Fi," *ESP-IDF Programming Guide* . [Online]. Available: https://docs.espressif.com/projects/esp-idf/en/latest/esp32/api-reference/network/esp_wifi.html. [Accessed: 30-Oct-2021].

[10] "IEEE code of Ethics," *IEEE*, 2020. [Online]. Available: https://www.ieee.org/about/corporate/governance/p7-8.html. [Accessed: 30-Oct-2021].

[11] "ACM Code of Ethics and Professional Conduct," *Code of Ethics*, 2018. [Online]. Available: https://www.acm.org/code-of-ethics. [Accessed: 30-Oct-2021].

6 Appendices

6.1 Appendix A: Django Framework Module-File Structure

```
C: .\445-website\
|  db.sqlite3
|  manage.py
|  my_project.dot
|
├──panels
|  |  admin.py
|  |  apps.py
|  |  models.py
|  |  tests.py
|  |  urls.py
|  |  views.py
|  |  __init__.py
|  |
|  ├──migrations
|  |  |  0001_initial.py
|  |  |  __init__.py
|  |  |
|  |  └──__pycache__
|  |        0001_initial.cpython-310.pyc
|  |        __init__.cpython-310.pyc
|  |
|  ├──templates
|  |     panel_detail.html
|  |     panel_index.html
|  |
|   └───__pycache__
|         admin.cpython-310.pyc
|         apps.cpython-310.pyc
|         models.cpython-310.pyc
|         urls.cpython-310.pyc
|         views.cpython-310.pyc
|         __init__.cpython-310.pyc
|
├───research_portal
|   |  asgi.py
|   |  settings.py
|   |  urls.py
|   |  wsgi.py
|   |  __init__.py
|   |
|   ├───templates
|   |      base.html
|   |
|   └───__pycache__
|         settings.cpython-310.pyc
|         urls.cpython-310.pyc
|         wsgi.cpython-310.pyc
|         __init__.cpython-310.pyc
|
├───team
|   |  admin.py
|   |  apps.py
|   |  models.py
|   |  tests.py
|   |   views.py
|   |   __init__.py
|   |
|   ├───migrations
|   |   |   __init__.py
|   |   |
|   |   └───__pycache__
|   |           __init__.cpython-310.pyc
|   |
|   └───__pycache__
|           admin.cpython-310.pyc
|           apps.cpython-310.pyc
|           models.cpython-310.pyc
|           __init__.cpython-310.pyc
|
└───users
    |   admin.py
    |   apps.py
    |   models.py
    |   tests.py
    |   urls.py
    |   views.py
    |   __init__.py
    |
    ├───migrations
    |   |   __init__.py
    |   |
    |   └───__pycache__
    |           __init__.cpython-310.pyc
```

```
|
├──templates
|   ├──registration
|   |      login.html
|   |
|   └──users
|         opening_page.html
|
└──__pycache__
       admin.cpython-310.pyc
       apps.cpython-310.pyc
       models.cpython-310.pyc
       urls.cpython-310.pyc
       views.cpython-310.pyc
       __init__.cpython-310.pyc
```

6.2 Appendix B: Django Project Visualization .dot Graph

digraph model_graph { // Dotfile by Django-Extensions graph_models // Created: 2021-11-01 12:13 // Cli Options: -a fontname = "Roboto" fontsize = 8 splines = true node [fontname = "Roboto" fontsize = 8 shape = "plaintext"] edge [fontname = "Roboto" fontsize = 8] // Labels django_contrib_admin_models_LogEntry [label=<

LogEntry	
id	**AutoField**
content_type	**ForeignKey (id)**
user	**ForeignKey (id)**
action_flag	PositiveSmallIntegerField
action_time	DateTimeField
change_message	TextField

object_id	TextField
object_repr	CharField

>] django_contrib_auth_models_AbstractUser [label=<

AbstractUser ***<AbstractBaseUser,PermissionsMixin>***	
date_joined	DateTimeField
email	EmailField
first_name	CharField
is_active	BooleanField
is_staff	BooleanField
is_superuser	*BooleanField*
last_login	*DateTimeField*
last_name	CharField
password	*CharField*
username	CharField

>] django_contrib_auth_models_Permission [label=<

Permission	
id	**AutoField**
content_type	**ForeignKey (id)**
codename	CharField
name	CharField

>] django_contrib_auth_models_Group [label=<

Group	
id	**AutoField**
name	CharField

>] django_contrib_auth_models_User [label=<

User ***<AbstractUser>***	
id	**AutoField**
date_joined	*DateTimeField*
email	*EmailField*
first_name	*CharField*
is_active	*BooleanField*
is_staff	*BooleanField*
is_superuser	*BooleanField*
last_login	*DateTimeField*
last_name	*CharField*
password	*CharField*
username	*CharField*

>] django_contrib_contenttypes_models_ContentType [label=<

ContentType	
id	**AutoField**
app_label	CharField
model	CharField

>] django_contrib_sessions_base_session_AbstractBaseSession [label=<

AbstractBaseSession	
expire_date	DateTimeField
session_data	TextField

>] django_contrib_sessions_models_Session [label=<

Session ***<AbstractBaseSession>***	
session_key	***CharField***

expire_date	*DateTimeField*
session_data	*TextField*

>] panels_models_Panel [label=<

Panel	
id	**BigAutoField**
description	TextField
facts	CharField
title	CharField

>] // Relations django_contrib_admin_models_LogEntry -> django_contrib_auth_models_User [label=" user (logentry)"] [arrowhead=none, arrowtail=dot, dir=both]; django_contrib_admin_models_LogEntry -> django_contrib_contenttypes_models_ContentType [label=" content_type (logentry)"] [arrowhead=none, arrowtail=dot, dir=both]; django_contrib_auth_base_user_AbstractBaseUser [label=<

AbstractBaseUser

>] django_contrib_auth_models_AbstractUser -> django_contrib_auth_base_user_AbstractBaseUser [label=" abstract\ninheritance"] [arrowhead=empty, arrowtail=none, dir=both]; django_contrib_auth_models_PermissionsMixin [label=<

PermissionsMixin

>] django_contrib_auth_models_AbstractUser -> django_contrib_auth_models_PermissionsMixin [label=" abstract\ninheritance"] [arrowhead=empty, arrowtail=none, dir=both]; django_contrib_auth_models_Permission -> django_contrib_contenttypes_models_ContentType [label=" content_type (permission)"] [arrowhead=none, arrowtail=dot, dir=both]; django_contrib_auth_models_Group -> django_contrib_auth_models_Permission [label=" permissions (group)"] [arrowhead=dot arrowtail=dot, dir=both]; django_contrib_auth_models_User -> django_contrib_auth_models_Group [label=" groups (user)"] [arrowhead=dot arrowtail=dot, dir=both]; django_contrib_auth_models_User -> django_contrib_auth_models_Permission [label=" user_permissions (user)"] [arrowhead=dot arrowtail=dot, dir=both]; django_contrib_auth_models_User -> django_contrib_auth_models_AbstractUser [label=" abstract\ninheritance"] [arrowhead=empty, arrowtail=none, dir=both]; django_contrib_sessions_models_Session -> django_contrib_sessions_base_session_AbstractBaseSession [label=" abstract\ninheritance"] [arrowhead=empty, arrowtail=none, dir=both]; }