

## Project Report

### Description

The main goal of this project was to create an interactive and easy to use web app that allows firefighters, or other agencies view and analyze all the wildfires currently happening in the United States. This app makes an API call to a wildfire database from the National Interagency Fire Center<sup>1</sup> and displays the data to a user on an interactive map of the U.S. and on a dashboard page. We used the Streamlit<sup>2</sup> library to create the front end and used their servers for free hosting as well. While the user will only see the front-end interface, most of the coding was done in the backend in `classes.py`. Here you will find a variety of classes which will be explained in further detail. The main data structure used in this project was the `EventStack` class. As can be inferred from the name, this class is an implementation of a stack with special functionality that is used to store wildfire event information received from the API call. Wildfire events are inserted into the stack based on a priority index. This index was determined by taking the acres covered by a wildfire and dividing it by the total time elapsed since the fire began:

$$Priority\ Index = \frac{Acres}{Time\ Elapsed}$$

### Significance

As of right now, there are over 1,000 ongoing wildfires in the U.S. With all these fires happening at once, it can be difficult to keep track of important information. This project seeks to aid this issue by creating a hub of wildfires and their data. Users of this app can view all wildfires in this U.S. as well as get information on individual fires. This app also presents useful statistics on these fires to the user. The stats that get presented are total count of wildfires, total acreage covered by wildfires, and the estimated suppression cost to put out every fire. We believe this app would be most useful to firefighting agencies trying to figure out which fires should be allocated the most resources. This can also help local governments decide what budgets should be assigned to their fire departments to help combat these fires.

<sup>1</sup> <https://data-nifc.opendata.arcgis.com/datasets/nifc::wfigs-2025-interagency-fire-perimeters-to-date/about>

<sup>2</sup> <https://streamlit.io/>

## Usage

There are two options for users who want to use this project, one is to go to the live demo at <https://wildfiredashboard.streamlit.app>. The other option is to install it on your local machine along with the required packages. We will go over the usage of both options.

### Live Demo Option

To access the live demo for the app, simply go to the following link:

<https://wildfiredashboard.streamlit.app> and click “start app.” Please allow up to one minute for the app to start. Once the app has started, you will be at the home page where you will see an interactive map of all wildfires currently in the U.S. and its territories. On the map, you will see various points each representing a wildfire. The larger points represent larger fires, and a color gradient assigns different colors to different sizes of fires as well for easier distinction between points. You are able to zoom and scroll anywhere throughout the map, and hovering over a point will give you a couple key pieces of information on the fire: `InitialLatitude`, latitude of the fire; `InitialLongitude`, longitude of the fire; `ID`, unique ID number of the wildfire event; `Acres`, acreage covered by the fire. Below the map you can see the basic statistics mentioned earlier: Wildfire count, acres covered, and estimated suppression cost. One important thing to note is that the size of the points does not reflect the actual size of the fire in real life. Difference in sizing exists only for distinction between points.

Using the side panel, you can navigate to the User Dashboard page. At the top of the page, you will see metrics for the top 3 highest priority fires, along with their name, ID, acreage covered, and estimated suppression cost. Under these metrics, you will see a table presenting a complete list of wildfires and their associated data. Here you will find the same meaningful information as before, along with more specific data like City, State, and County, as well as the cause of the fire if available. Streamlit offer special functionality with their data tables, allowing you the user to search for an entry via any field like ID or name. You are even able to download all the data in the table as a CSV file by simple clicking on the download icon in the top left. Under this table is the “Get Fire by ID” section. Here as the name implies, you are able to type the ID of any fire into the box and get specific information regarding the fire.

## Installation Option

If you choose to install the app on your own machine, you can run the following commands to clone into the GitHub repository, create a virtual machine, and run the app.

First, clone into the repository by typing the following command into your terminal:

```
git clone https://github.com/drkohlbek/WildfireDashboard.git
```

Next, create a virtual environment (optional):

```
python -m venv ./  
source ./bin/activate
```

Install the required packages:

```
pip install -r requirements.txt
```

or, for Mac:

```
pip3 install -r requirements.txt
```

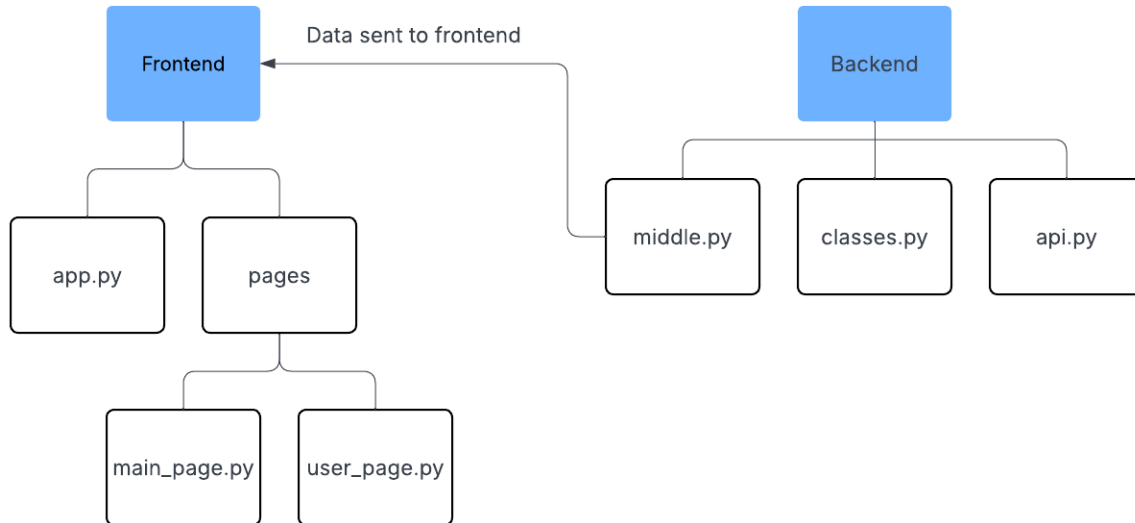
Finally, start the app:

```
streamlit run app.py
```

This final command will start a local web server on port 8501. Go to your browser and type in localhost:8501 to view the app. From here, refer to the instructions under “Live Demo Option” to use the app.

## Code Structure

The files in this project are split up into two main groups, frontend, and backend. Each group contains multiple files to handle various parts of the website.



We will explain each group individually, starting with the backend.

### Backend

Here you will find the underlying functionality that manages the data used in this project. The file `api.py` runs a simple API call using the Python requests library to the National Interagency Fire Center database. From here, we retrieve the json response and store it in a variable `data` to be used in `middle.py`.

```
1 import requests
2
3 # initial API call, see https://data-nifc.opendata.arcgis.com/datasets/nifc::wfigs-2025-interagency-fire-perimeters-to-date/about
4 API = "https://services3.arcgis.com/T4QMspbfLg3qTGWY/arcgis/rest/services/WFIGS_Interagency_Perimeters_YearToDate/FeatureServer/0/query?outFields=*&where=1%3D1&f=geojson"
5 response = requests.get(API)
6 data = response.json()
```

The next file is `classes.py`. Here you will find 3 classes, `Event`, `EventStack`, and `EventUtils`. `Event` is a class to represent wildfire events, and the attributes for this class is the data received from the database. `EventStack` is the main data structure that is used throughout the project. It is a stack with special functionality that our app to get a wildfire event by its ID, and to get the top three wildfire events based on their priority. `EventUtils` is a class full of static methods that are used throughout each class and other files in this project. The specific functionality of each of these classes will be described in more detail in the next section.

Finally, `middle.py` is a simple middleware file to instantiate an instance of the `EventStack` class as well as other data and send it to the frontend. This file is the “bridge” that connects the backend to the frontend in this application.

## Frontend

We will start with the files located under the pages folder. First is `main_page.py`. As the name implies, this is the main page of the application and what the user will land on when they go to the URL for the app. Here we Streamlit's built in Plotly<sup>3</sup> functionality to display an interactive map of our wildfire data. From `middle.py`, we import `stats`, the basic statistics of our data, and `figure`, the map figure generated by the Plotly library in `classes.py`.

Next, we have `user_page.py`. This page presents the user dashboard that contains a variety of useful information. Firstly, the user is presented with 3 metrics of the top three highest priority fires in the U.S., imported from `middle.py`. Then, our data is turned into a data frame using the Pandas<sup>4</sup> library and presented to the user using `streamlit.dataframe`. At the bottom of the page, we present functionality to search for a wildfire event by its ID, then we use `EventStack.get_by_ID()` to retrieve the wildfire event and present its data to the user.

Finally, we have `app.py`. This is a simple file that is needed to run the Streamlit app using `streamlit run app.py`. It retrieves the previously described pages and loads them into the main app and runs it using `pg.run()`.

<sup>3</sup> <https://plotly.com/>

## Functionality and Testing

Here we will describe in more detail the classes and functions found in `classes.py` and some testing to showcase their functionality.

### Event class

`Event` is a class to store information about a wildfire event with data received from the API call in `api.py`. The photo below lists the attributes of this class and a brief description of each attribute:

```
Attributes
-----
ID: int
| Unique ID for wildfire event
IncidentName: str
| Name of wildfire event
Acres: float
| Total acres covered by fire of the event
CreateDate: esriFieldTypeDate/int
| Date of event creation, eventually converted into Python datetime.datetime
CurrentDate: datetime.datetime
| Date the program is run, aka the current date
FireBehaviorGeneral: str
| General behavior of the fire (minimal, moderate, high)
FireCause: str
| Cause (if known) of the fire
FireCauseSpecific: str
| More specific cause of the fire
IncidentShortDescription: str
| Short description of the fire
InitialLatitude: float
| Initial latitude extracted by EventUtils.get_coordinates()
InitialLongitude: float
| Initial longitude extracted by EventUtils.get_coordinates()
State: str
| Starting state
City: str
| Starting city of the fire
County: str
| Starting county of the fire
Priority: float
| Priority of the fire, see EventUtils.determine_priority() for more info.
| User is also able to set this value manually.
```

Now we will explain each of the methods found in this class along with testing some functionality of the class.

```
Event.__init__()
```

Simple Python init method to insatiate attributes of the class. The attribute `location_info` has special logic to extract the city, state, and county of a wildfire event from its latitude and longitude attributes using `EventUtils.get_reverse_geocode()` (explained later).

```
# Run reverse_geocode and extract location info
location_info = EventUtils.get_reverse_geocode(
    InitialLatitude, InitialLongitude)
if location_info != None:
    self.City = location_info["city"]
    self.State = location_info["state"]
    try:
        self.County = location_info["county"]
    except KeyError:
        self.County = None
else:
    self.City = None
    self.State = None
    self.County = None
```

As can be seen, safe cases are developed in the logic in the case `get_reverse_geocode()` does not return the required information.

The class also has functionality if a user wants to manual specify the priority of a wildfire event. If no priority is specified, `EventUtils.determine_priority()` is called to create the priority index.

```
# Allow user to override priority
if Priority != None:
    self.Priority = Priority
else:
    self.Priority = EventUtils.determine_priority(self)
```

For the `self.Cost` attribute, we simply multiply `self.Acres` by a constant `COST_PER_ACRE` determined by calculating the average suppression cost of for wildfires in 2023.

```
# Using numbers from https://www.nifc.gov/fire-information/statistics/suppression-costs,
# the total cost per acre to suppress wildfires in 2023 was ~$1,177
COST_PER_ACRE = 1177

# Determine estimated fire suppression costs
if self.Acres != None:
    self.Cost = self.Acres * COST_PER_ACRE
else:
    You, last week • First Commit ...
    self.Cost = None
```

To test functionality of the class, we create an instance of `Event` using dummy data:

```
my_event = Event(22, "Test Event", 5544, 1738366578452, "Fast", "Human",
                "Human lit a cigarette in a forest", "Restaurant worker smoked on his break and caused a fire", 30, 30, None)
```

`Event.__str__()`

`str()` is a Python Dunder method that returns general information about a wildfire event.

```
def __str__(self):
    return f"Information on wildfire {self.ID} in {self.City}, {self.State}:\n \
Incident Name: {self.IncidentName}\n Coverage in Acres: {self.Acres}\n Start Date: {self.convert_date(self.CreateDate)}\n \
Time Elapsed: {str(self.time_elapsed())}\n Fire Behavior: {self.FireBehaviorGeneral}\n \
Cause: {self.FireCause}\n Estimated Cost: {self.Cost}\n Priority: {self.Priority}\n\n \
Description of the Incident:\n {self.IncidentShortDescription}"
```

```
# __str__() demonstration
print(my_event)
print()
```

Code output:

```
● (base) dani@MacBookAir CS_Final % python3 -u testing.py
Information on wildfire 22 in Madīnat Wādī an Naṭrūn, Beheira:
Incident Name: Test Event
Coverage in Acres: 5544
Start Date: 2025-01-31 13:36:18.452000
Time Elapsed: 86 days, 0:14:01.633568
Fire Behavior: Fast
Cause: Human
Estimated Cost: 6525288
Priority: 0.0007460395278975918

Description of the Incident:
Restaurant worker smoked on his break and caused a fire
```



```
Event.convert_date(date)
```

Date data received from the API is in a format called “ersiFieldTypeDate,” This format is an integer representation of a date. The function `convert_date()` takes this integer and converts it to a Python `datetime.datetime` object so it can be used more easily throughout our program. This link was used for reference: <https://pro.arcgis.com/en/pro-app/latest/help/mapping/time/convert-string-or-numeric-time-values-into-data-format.htm>

```
# convert ersi date integer to datetime
def convert_date(self, date):
    """
    convert_date converts ersiFieldTypeDate integer to datetime.datetime object
    ref: https://pro.arcgis.com/en/pro-app/latest/help/mapping/time/convert-string-
    """
    if date != None:
        return datetime.datetime.fromtimestamp((date/1000) - 18000)
    return None
```

```
# convert_date() demonstration
print(my_event.convert_date(my_event.CreateDate))
print()
```

Code output:

```
2025-01-31 13:36:18.452000
```

```
Event.time_elapsed()
```

This function calculates the time from the start of the wildfire to the current date.

```
# return total time since fire began
def time_elapsed(self):
    """
    time_elapsed() calculate and returns the time since the start of the wildfire
    """
    if self.CreateDate != None:
        return self.CurrentDate - self.convert_date(self.CreateDate)
    return None    You, last week • First Commit

# time_elapsed() demonstration
print(my_event.time_elapsed())
print()
```

Code output:

```
86 days, 0:14:01.633568
```

```
Event.return_dict()
```

This function simply returns a Python dictionary representation of the attributes of an `Event` object. This representation becomes useful in other areas of the program when we use Pandas data frames to display information.

```
# return a dict representation of information stored in the object
def return_dict(self):
    """
    return_dict() returns a dict representation of all the attributes of
    the object
    """
    return {
        'ID': self.ID,
        'IncidentName': self.IncidentName,
        'Acres': self.Acres,
        'CreateDate': self.convert_date(self.CreateDate),
        'CurrentDate': self.CurrentDate,
        'State': self.State,
        'City': self.City,
        'County': self.County,
        'Priority': self.Priority,
        'Estimated Cost': self.Cost,
        'InitialLatitude': self.InitialLatitude,
        'InitialLongitude': self.InitialLongitude,
        'FireBehaviorGeneral': self.FireBehaviorGeneral,
        'FireCause': self.FireCause,
        'FireCauseSpecific': self.FireCauseSpecific,
        'IncidentShortDescription': self.IncidentShortDescription,
    }
```

```
# return_dict() demonstration
print(my_event.return_dict())
print()
```

Code output:

```
{'ID': 22, 'IncidentName': 'Test Event', 'Acres': 5544, 'CreateDate': datetime.datetime(2025, 1, 31, 13, 36, 18, 452000), 'CurrentDate': datetime.datetime(2025, 4, 27, 13, 50, 20, 85568), 'State': 'Beheira', 'City': 'Madinat Wadi an Naṭrūn', 'County': None, 'Priority': 0.0007460395278975918, 'Estimated Cost': 6525288, 'InitialLatitude': 30, 'InitialLongitude': 30, 'FireBehaviorGeneral': 'Fast', 'FireCause': 'Human', 'FireCauseSpecific': 'Human lit a cigarette in a forest', 'IncidentShortDescription': 'Restaurant worker smoked on his break and caused a fire'}
```

## EventStack class

`EventStack` is the main data structure used throughout this program. As the name implies, it is an implementation of a Stack with special functionality to work with wildfire events. Wildfire events are loaded into the stack based on priority, with the highest priority fires being at the top of the stack. This class only has one attribute, `self.events`, which is a list to represent the stack of wildfire events.

To test the `EventStack` class, we will create dummy event objects like before and load them into an instance of the `EventStack` class using `EventUtils.load_stack()`, (explained later).

```
# EventStack testing
"""
first, we'll create more dummy Event objects and store them in an array. Then we will use EventUtils.load_stack()
to load the array and instantiate a stack.
"""
my_event1 = Event(22, "Test Event 1", 5544, 1738366578400, "Fast", "Human",
| | | | | | | | "Human lit a cigarette in a forest", "Restaurant worker smoked on his break and caused a fire", 30, 30, None)
| | | | | | | |
my_event2 = Event(23, "Test Event 2", 200, 1738366578431, "Fast", "Human",
| | | | | | | | "Human lit a cigarette in a dumpster", "Police officer tossed a cigarette in a dumpster and caused a fire", 45, 40, None)
| | | | | | | |
my_event3 = Event(24, "Test Event 3", 1509, 1738366578452, "Fast", "Human",
| | | | | | | | "Human lit a cigarette in a cafe", "Cafe attendee lit a cigarette and caused a fire", 63, 50, None)
| | | | | | | |
my_stack = EventUtils.load_stack([my_event1, my_event2, my_event3])
```

```
__init__(), isEmpty(), push(item), pop(), peek(), and size()
```

The following functions are the basic stack functions you will find in every implementation of a Stack data structure, so we will not go into explicit detail on them, though it is clear how they work from the code and testing outputs.

```
def __init__(self):
    self.events = []

def isEmpty(self):
    return self.events == []

def push(self, item):
    self.events.append(item)

def pop(self):
    return self.events.pop()

def peek(self):
    return self.events[len(self.events)-1]

def size(self):
    return len(self.events)
```

```
# testing basic stack functionality
print( (variable) my_stack: EventStack)
print(my_stack.isEmpty())
print()

print("Popping from the stack")
popped_event = my_stack.pop()
print(popped_event)
print("Length of stack after popping: " + str(my_stack.size()))
print()

print("Pushing to the stack")
my_stack.push(popped_event)
print("Length of stack after pushing: " + str(my_stack.size()))
print()

print("Peeking into the stack")
print(my_stack.peak())
print()
```

Code output:

```
Checking if stack is empty
False

Popping from the stack
Information on wildfire 22 in Madīnat Wādī an Naṭrūn, Beheira:
  Incident Name: Test Event 1
  Coverage in Acres: 5544
  Start Date: 2025-01-31 13:36:18.400000
  Time Elapsed: 86 days, 0:14:01.986484
  Fire Behavior: Fast
  Cause: Human
  Estimated Cost: 6525288
  Priority: 0.0007460394924675404

Description of the Incident:
Restaurant worker smoked on his break and caused a fire
Length of stack after popping: 2

Pushing to the stack
Length of stack after pushing: 3

Peeking into the stack
Information on wildfire 22 in Madīnat Wādī an Naṭrūn, Beheira:
  Incident Name: Test Event 1
  Coverage in Acres: 5544
  Start Date: 2025-01-31 13:36:18.400000
  Time Elapsed: 86 days, 0:14:01.986484
  Fire Behavior: Fast
  Cause: Human
  Estimated Cost: 6525288
  Priority: 0.0007460394924675404

Description of the Incident:
Restaurant worker smoked on his break and caused a fire
```

```
EventStack.get_by_ID(ID)
```

This function gets a wildfire event in an `EventStack` instance by its ID. The function simply iterates one by one through the array `EventStack.events` until it finds an object with the specified ID. Time complexity  $O(n)$ .

```
def get_by_ID(self, ID):
    """
    Simply iterates through the stack to find the event with specified ID.

    Time complexity  $O(n)$ 
    """
    for event in self.events:
        if event.ID == ID:
            return event.return_dict()
```

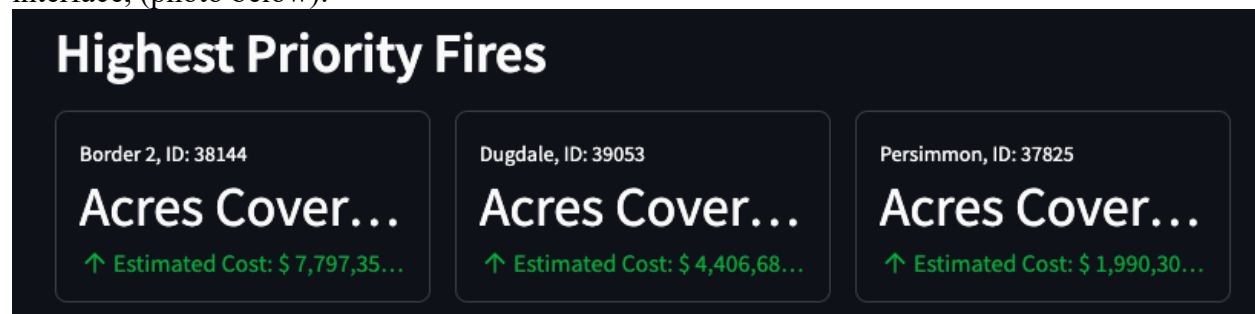
```
# special functionality of the EventStack class
print("get_by_ID() demonstration")
print(my_stack.get_by_ID(24))
print()
```

Code output:

```
get_by_ID() demonstration
{'ID': 24, 'IncidentName': 'Test Event 3', 'Acres': 1509, 'CreateDate': datetime.datetime(2025, 1, 31, 13, 36, 18, 452000), 'CurrentDate': datetime.datetime(2025, 4, 27, 13, 50, 20, 386565), 'State': 'Komi', 'City': 'Mikun', 'County': None, 'Priority': 0.00020306161652215563, 'Estimated Cost': 1776093, 'InitialLatitude': 63, 'InitialLongitude': 50, 'FireBehaviorGeneral': 'Fast', 'FireCause': 'Human', 'FireCauseSpecific': 'Human lit a cigarette in a cafe', 'IncidentShortDescription': 'Cafe attendee lit a cigarette and caused a fire'}
```

```
EventStack.get_top_three()
```

This function simply returns the top three elements in the stack as an array. This functionality was developed so it would be easier to display metrics of the highest priority fires on the user interface, (photo below):



Highest Priority Fires		
Border 2, ID: 38144	Dugdale, ID: 39053	Persimmon, ID: 37825
<b>Acres Cover...</b>	<b>Acres Cover...</b>	<b>Acres Cover...</b>
↑ Estimated Cost: \$ 7,797,35...	↑ Estimated Cost: \$ 4,406,68...	↑ Estimated Cost: \$ 1,990,30...

```
def get_top_three(self):  
    """  
    Returns top 3 elements from stack in an array to be used later  
    in user dashboard.  
    """  
    events = []  
    events.append(self.events[-1])  
    events.append(self.events[-2])  
    events.append(self.events[-3])  
  
    return events
```

```
print("get_top_three() demonstration")  
top_three = my_stack.get_top_three()  
for x in top_three:  
    print(x.IncidentName)  
print()
```

Code output:

```
get_top_three() demonstration  
Test Event 1  
Test Event 3  
Test Event 2
```

## EventUtils class

This is a class that contains a variety of useful static methods that are used throughout the program.

`EventUtils.determine_priority(event)`

This method determines the priority of a wildfire event by measuring the ratio of acres to time elapsed of a wildfire. The following equation gives the priority index for an event:

$$\text{Priority Index} = \frac{\text{Acres}}{\text{Time Elapsed}}$$

For cases where the API does not return an entry for the creation date of a wildfire event, the priority index is equal to the number of acres covered.

For cases where there is no entry of acres, the priority index is 0.

```
@staticmethod
def determine_priority(event):
    """
    Determine priority of a wildfire event by measureing the ratio of acres
    to time elapsed of a wildfire; a higher ratio means a higher priority.

    Priority index = acres / time elapsed (seconds)

    For cases where their is no entry for CreateDate, return the number of acres
    as the priority.

    For cases where there is no entry for Acres, return 0.
    """
    if event.Acres and event.time_elapsed():
        priority = event.Acres / event.time_elapsed().total_seconds()
        return priority
    elif event.Acres:
        return event.Acres
    return 0
```

```
# EventUtils testing
print(
    f"EventUtils.determine_priority() demo: {EventUtils.determine_priority(my_event1)}")
print()
```

Code output:

```
EventUtils.get_priority() demo: 0.0007460394924675404
```



```
EventUtils.get_coordinates()
```

This function determines the coordinates of a wildfire event by taking the mean value of the coordinates of the perimeter of the wildfire as received by the API. The API gives the coordinates as an n-dimensional array and this function takes the mean and returns a 1-dimensional array with two entries [latitude, longitude].

This function is necessary because many of the entries in the database do not have values for `InitialLatitude` and `InitialLongitude`, however they all have entries for the perimeter of the wildfire.

Another issue is the dimension of coordinate arrays from the API is not consistent across all entries. The workaround we developed is to take only the arrays with `.ndim = 1`.

The input “feature” is simply one wildfire event from the API call. The reason I chose “feature” as the name of the variable is because each wildfire event in the API is stored in a dict called “features.”

Time complexity is  $O(n)$  for each subarray of coordinates.

```
@staticmethod
def get_coordinates(feature):
    """
    Get coordinates of a wildfire event by taking the mean value
    of the coordinates of the perimeter of the event.

    This is necessary since many entries in the database do not have
    attr_InitialLatitude and attr_InitialLongitude values defined.

    Dimension of coordinate arrays is not consistent across all entries. There
    are some entries with multiple 2D arrays in them. A workaround for this is
    to only take the arrays with .ndim = 1 as shown below.
    """
    orig_coords = np.array(
        [x for x in feature['geometry']['coordinates'][0]])
    coords = orig_coords.mean(axis=0)

    if coords.ndim > 1:
        return coords[0]
    return coords
```

```
feature = features[0]
print(
    f"EventUtils.get_coordinates() demo: {EventUtils.get_coordinates(feature)}")
print()
```

Code output:

```
EventUtils.get_coordinates() demo: [-118.54480318  34.07415481]
```

```
EventUtils.quicksort(fires, low=0, high=None) and  
EventUtils.partition(fires, low, high)
```

These two functions are used to create a quicksort algorithm that will be used when loading an array of wildfire events into an `EventStack` instance using `EventUtils.load_stack()`. The functions sort the array based on the priority index of each wildfire object.

The algorithm chooses a pivot point from the array then moves all other values, so the lower priorities are on the left side of the pivot element. The algorithm then recursively does the same operation on both sub arrays on either side of the pivot until the array is sorted.

The time complexity of the quicksort algorithm is  $O(n^2)$  in the worst case and  $O(n \log n)$  in the average case.

```
@staticmethod  
def partition(fires, low, high):  
    """  
    partition() creates partitions to be used in quicksort()  
    (parameter) fires: Any  
    """  
    pivot = fires[high]  
    i = low - 1  
  
    for j in range(low, high):  
        if fires[j].Priority <= pivot.Priority:  
            i += 1  
            fires[i], fires[j] = fires[j], fires[i]  
  
    fires[i+1], fires[high] = fires[high], fires[i+1]  
    return i+1  
  
def quicksort(fires, low=0, high=None):  
    """  
    quicksort() is a function that takes an array of fires and sorts it  
    by priority index. This function is called when instantiating the EventStack class.  
  
    The quicksort algorithm itself intakes an array of values and chooses a value as  
    the pivot. Then it moves all other values so the lower priorities are on the  
    left side of the pivot element. The algorithm then recursively does the same operation  
    on both sub arrays on each side of the pivot until the array is sorted.  
  
    Time complexity: Worst case  $O(n^2)$ , however, average case is  $O(n \log n)$   
    """  
    if high == None:  
        high = len(fires) - 1  
  
    if low < high:  
        pi = EventUtils.partition(fires, low, high)  
        EventUtils.quicksort(fires, low, pi-1)  
        EventUtils.quicksort(fires, pi+1, high)
```

The testing of this code can be viewed above under the `EventStack` section when `EventUtils.load_stack()` is called.

```
EventUtils.plot_map(fires)
```

This function takes an array of fires and creates an interactive map using the `plotly.express.scatter_map` function. `plot_map()` returns a `matplotlib` figure which is rendered later by `plotly.express` in the frontend.

```
@staticmethod
def plot_map(fires):
    """
    Method to plot wildfire events to an interactive map. This function
    returns a matplotlib figure which is rendered later by plotly.express
    on the website.
    """
    # Dict array of wildfire events for plotting
    fires_dict_array = [event.return_dict() for event in fires]
    # Store dict array in dataframe
    df = pd.DataFrame(fires_dict_array)

    df.fillna({'Acres': 0}, inplace=True)

    # Set custom size for each point on the map.
    # The size of each point will be the same as the acres covered, unless
    # it is less than 500, then the point size will be 500.
    df['size'] = df['Acres']

    for i in range(len(df)):
        if df.loc[i, "size"] < 500:
            df.loc[i, "size"] = 500

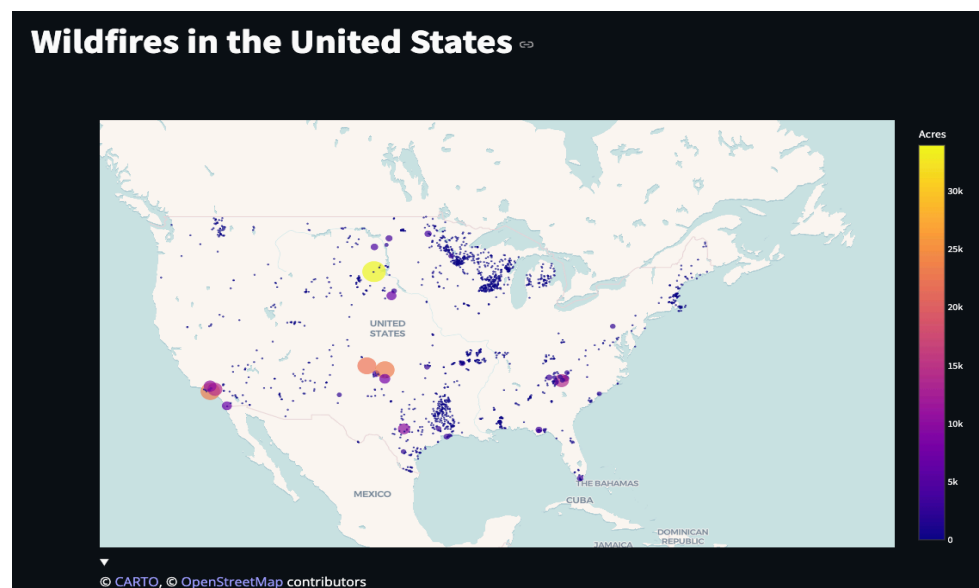
    # plot
    fig = px.scatter_map(df, lat=df["InitialLatitude"], lon=df["InitialLongitude"],
                        hover_name="IncidentName", hover_data={'ID': True, 'InitialLatitude': True,
                        'InitialLongitude': True, 'Acres': True, 'size': False}, color="Acres", zoom=3,
                        size=df["size"], height=700)

    return fig
```

Frontend implementation in `main_page.py`

```
st.markdown("# Wildfires in the United States")
st.plotly_chart(figure, height=700, theme=None)
```

Code output:



```
EventUtils.load_fires(data)
```

This function deconstructs data received from the API, loads it into an array and returns the array. This function extracts only the properties needed from a wildfire event for this program. Each property extracted from the API is turned into an Event object and appended to the returned array.

Time complexity is  $O(k*n)$  where  $O(k)$  is the time complexity of `EventUtils.get_coordinates()` and  $O(n)$  is the number of iterations done by the for loop determined by the size of the data array.

```
# Method to load fires from data into list
@staticmethod
def load_fires(data) -> list:
    """
    load_fires() load wildfire event information received from the API, extracting
    only the properties needed for this program.
    """
    fires = []
    for feature in data['features']:
        properties = feature['properties']

        # Only select wild fires, not prescribed fires
        if properties['poly_FeatureCategory'] != 'Prescribed Fire':
            coords = EventUtils.get_coordinates([feature])
            event = Event(ID=properties['OBJECTID'], IncidentName=properties['poly_IncidentName'], Acres=properties['poly_GISAcres'],
                          CreateDate=properties['poly_CreateDate'], FireBehaviorGeneral=properties['attr_FireBehaviorGeneral'],
                          FireCause=properties['attr_FireCause'], FireCauseSpecific=properties['attr_FireCauseSpecific'],
                          IncidentShortDescription=properties['attr_IncidentShortDescription'], InitialLatitude=coords[1], InitialLongitude=coords[0])
            fires.append(event)
    return fires
```

For testing, we will load the fires from the data received from the API, print out the first element and the size of the array.

```
fires = EventUtils.load_fires(data)
print(f"EventUtils.load_fires() first element in array: \n{fires[0]}\n")
print(f"Size of array {len(fires)}")
print()
```

```
EventUtils.load_fires() first element in array:
Information on wildfire 36832 in Topanga, California:
Incident Name: CALFD-000042
Coverage in Acres: 5.05
Start Date: 2025-01-31 13:36:18.452000
Time Elapsed: 86 days, 0:14:01.935316
Fire Behavior: None
Cause: Human
Estimated Cost: 5943.849999999999
Priority: 6.795633951843956e-07

Description of the Incident:
None

Size of array 1697
```

`EventUtils.load_stack(fires)`

This function takes an array of wildfire events created by `EventUtils.load_fires()` and returns an instance of the `EventStack`. `EventUtils.quicksort()` is called on the array of wildfires then each element in the array is sequentially added to the stack.

The time complexity of this function is  $O(n)$ .

```
@staticmethod
def load_stack(fires) -> EventStack:
    """
    load_stack simple creates a stack from an array of wildfire events,
    with the wildfire with the highest priority index being at the top of the stack.
    To do this, EventUtils.quicksort() is called on the wildfire array,
    then that array is used to create an EventStack object.
    """
    stack = EventStack()
    EventUtils.quicksort(fires)

    for event in fires:
        stack.push(event)

    return stack
```

The testing for this class can be again viewed above under the `EventStack` section where `load_stack()` is called.

```
EventUtils.get_stats(fires)
```

This function returns basic stats of wildfire events in an array or `EventStack`. As of now the stats returned are total acreage covered, total number of fires, and the total estimated suppression cost.

Time complexity for this function is  $O(n)$

```
@staticmethod
def get_stats(fires):
    """
    get_stats is a simple function that returns basic stats of wildfire
    events in an array. As of now the stats returns are: total acreage,
    total number of fires, and total estimated suppression cost.
    """
    stats = dict()

    # Total fires in array
    stats["total_fires"] = len(fires)

    # Total acreage covered
    total_acres = 0
    for event in fires:
        if event.Acres != None:
            total_acres += event.Acres
    stats["total_acres"] = round(total_acres)

    # Total estimated cost
    stats["estimated_cost"] = stats["total_acres"] * COST_PER_ACRE

    return stats

stats = EventUtils.get_stats(fires)
print(f"EventUtils.get_stats() demo: ")
print(f"Total number of fires: {stats['total_fires']}")
print(f"Total acres covered: {stats['total_acres']}")
print(f"Estimated suppression cost: {stats['estimated_cost']}")
print()
```

Code output:

```
EventUtils.get_stats() demo:
Total number of fires: 1697
Total acres covered: 304118
Estimated suppression cost: 357946886
```

```
EventUtils.get_reverse_geocode(lat, lon)
```

This function returns a variety of geographical information in the form of a Python dictionary given a wildfire event's latitude and longitude. The information used in this program is the city, state, and county. This function uses the `reverse_geocode` library to do so.

Reference: <https://pypi.org/project/reverse-geocode/>

```
@staticmethod
def get_reverse_geocode(lat, lon):
    """
    This function returns a variety of geographical information in the form of a Python dictionary
    given a wildfire event's latitude and longitude. The information used in this program is the
    city, state, and county. This function uses the reverse_geocode library to do so.
    Ref: https://pypi.org/project/reverse-geocode/ You, 1 second ago • Uncommitted changes
    """
    if lat != None and lon != None:
        coord = lat, lon
        return (reverse_geocode.get(coord))

    return None
```

```
test_fire = fires[0]
print(
    f"EventUtils.get_reverse_geocode() demo: {EventUtils.get_reverse_geocode(test_fire.InitialLatitude, test_fire.InitialLongitude)}")
```

Code output:

```
EventUtils.get_reverse_geocode() demo: {'country_code': 'US', 'city': 'Topanga', 'latitude': 34.09362, 'longitude': -118.60147, 'population': 8289, 'state': 'California', 'county': 'Los Angeles County', 'country': 'United States'}
```

## Achievement of Project Goals

Using the Streamlit library allowed us to create a frontend interface that seamlessly integrates with the backend to display an aesthetically pleasing and useful user dashboard that can be used by firefighters or other government officials.

### Main Page

The main page of the app displays a colorful and interactive map of all the wildfires currently happening in the US. Under the map are some basic statistics calculated with the `EventUtils.get_stats()` function. The code for this page is found in `main_page.py`:

```
import streamlit as st
from backend.middle import stats, figure

st.markdown("# Wildfires in the United States")
st.plotly_chart(figure, height=700, theme=None)


You, last week • First Commit

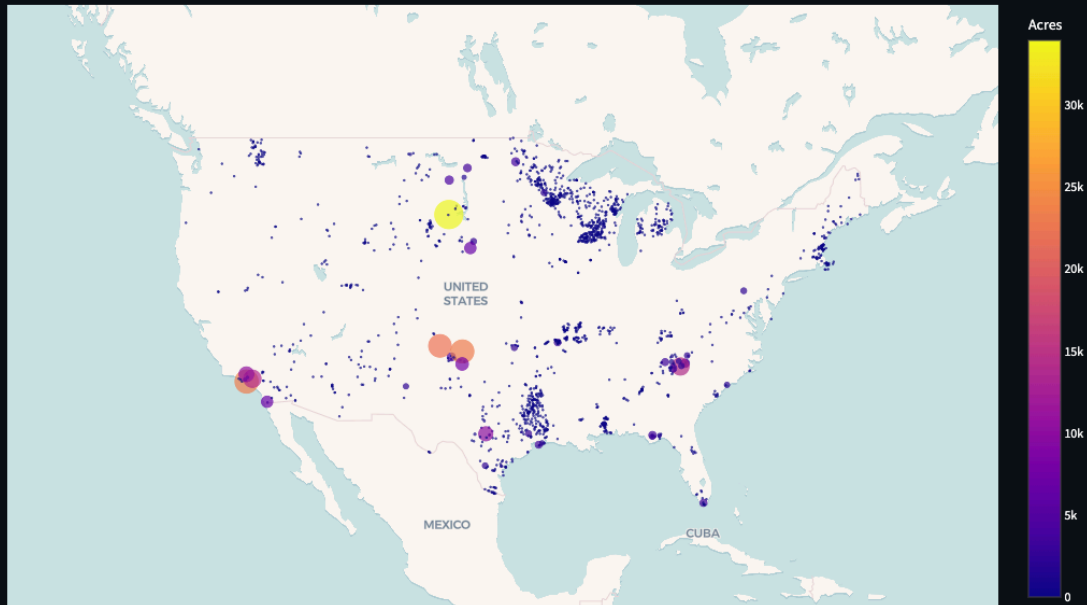

st.markdown("#### Totals")
string = "#### Wildfire Count: &emsp;&emsp;&emsp;&emsp;&emsp;&emsp; {:.20,.0f}".format(
    stats["total_fires"])
st.markdown(string)
st.markdown("#### Acres Covered: &emsp;&emsp;&emsp;&emsp;&emsp;&emsp; {:.20,.0f}".format(
    stats["total_acres"]))
st.markdown("#### Estimated Supression Cost: &emsp;&nbsp; ${:.20,.2f}".format(
    stats["estimated_cost"]))
```

The function `st.plotly_chart()` displays the figure created in `EventUtils.plot_map()` and adjusts the height to fit the webpage. Under that, several `st.markdown()` functions are called to display statistics in the webpage using the markdown language.

Below are photos of the execution of this file.



## Wildfires in the United States



© CARTO, © OpenStreetMap contributors

### Totals

Wildfire Count:	1,697
Acres Covered:	304,118
Estimated Supression Cost:	\$ 357,946,886.00

We believe that the information displayed effectively conveys the scale of different wildfires across the U.S. It is clear from the map what wildfires are the largest and most dangerous and need to be dealt with first.

The statistics displayed are meaningful as well, as it can assist government agencies to determine how many resources are needed to deal with these fires. The “estimated suppression cost” statistic can be useful in determining the budget of fire departments across various states

## User Dashboard

The user page displays more descriptive information about the wildfires occurring in the US. At the top of the page, the user can see the top 3 highest priority fires along with their name, ID, and estimated suppression cost. Below that is a table displaying a complete list of the wildfires. Streamlit provides special functionality with their tables that allows users to download information as a csv and search for an entry by whatever field they want. Finally, at the bottom of the page is a section where a user can find more explicit info on one wildfire by entering its ID.

Code for this page is found in `user_page.py`:

```
import streamlit as st
import pandas as pd
from backend.middle import top_three, fires_dict, stack
# package to format long numbers, i.e. 6,400 -> 6.4k
from millify import millify

You, last week • First Commit ...

# Cache fire data, otherwise streamlit deletes it every time you change the text_input()
@st.cache_data
def get_top_three(_top_three):
    return [fire.return_dict() for fire in _top_three]

@st.cache_data
def get_fires_df(_fires_dict):
    return pd.DataFrame(_fires_dict)

# extract variables from functions
fire1, fire2, fire3 = get_top_three(top_three)

df = get_fires_df(fires_dict)

st.markdown("# User Dashboard")

st.markdown("## Highest Priority Fires")
```

Here we can see functionality for getting the top three priority fires as well as storing all fires in a Pandas data frame. More `st.markdown()` functions are called to display text on the page.

```

a, b, c = st.columns(3)
a.metric(f"{fire1['IncidentName']}, ID: {fire1['ID']}", f"Acres Covered: {millify(fire1['Acres'], precision=1)}",
        "Estimated Cost: ${:20,.2f}".format(fire1['Estimated Cost']), border=True)
b.metric(f"{fire2['IncidentName']}, ID: {fire2['ID']}", f"Acres Covered: {millify(fire2['Acres'], precision=1)}",
        "Estimated Cost: ${:20,.2f}".format(fire2['Estimated Cost']), border=True)
c.metric(f"{fire3['IncidentName']}, ID: {fire3['ID']}", f"Acres Covered: {millify(fire3['Acres'], precision=1)}",
        "Estimated Cost: ${:20,.2f}".format(fire3['Estimated Cost']), border=True)

st.markdown("## Complete List of Fires")
st.markdown(
    "Hover over the table and click the search icon to find a specific fire by ID or any other field.")

table = st.dataframe(df, hide_index=True, column_config={
    "CurrentDate": None, "InitialLatitude": None, "InitialLongitude": None,
    "FireBehaviorGeneral": None, "FireCauseSpecific": None, "IncidentShortDescription": None})

st.markdown("#### Get fire by ID")
id = st.number_input(
    "Enter the ID of the fire to get detailed information on the event", value=38144)
user_dict = stack.get_by_ID(id)

st.table(user_dict)

```

Here we see code that displays the data. The first line splits a section of the webpage into three columns, a, b, and c. Then calling `.metric` on each of those columns creates the following item that displays information on the top 3 wildfires.

Border 2, ID: 38144 <b>Acres Covered: 6.6k</b> ↑ Estimated Cost: \$ 7,797,354.29	Dugdale, ID: 39053 <b>Acres Covered: 3.7k</b> ↑ Estimated Cost: \$ 4,406,688.00	Persimmon, ID: 37825 <b>Acres Covered: 1.7k</b> ↑ Estimated Cost: \$ 1,990,307.00
--	---	---

Next, the complete table of fires is displayed using `st.dataframe()`. Some columns are hidden so the table takes up less space on the page and only the most meaningful information is displayed.

## Complete List of Fires

Hover over the table and click the search icon to find a specific fire by ID or any other field.

ID	IncidentName	Acres	CreateDate	State	City	County	Priority	Estimated Cost	FireCause
38144	Border 2	6624.77	None	California	Jamul	San Diego County	6624.77	7797354.29	None
39053	Dugdale	3744	None	Minnesota	Red Lake Falls	Red Lake County	3744	4406688	None
37825	Persimmon	1691	None	Texas	Fannett	Jefferson County	1691	1990307	None
38675	KENNETH	998.7378	None	California	Hidden Hills	Los Angeles County	998.7378	1175514.3906	None
37904	Lost Creek	842.0406	None	Florida	Crawfordville	Wakulla County	842.0406	991081.7862	None
37913	Falls Dam	504.7565	None	North Carolina	Badin	Stanly County	504.7565	594098.4005	None
38456	Chestnut Mtn	353	None	Georgia	Calhoun	Gordon County	353	415481	None
38193	Green Acres	345	None	South Dakota	Hot Springs	Fall River County	345	406065	None
38589	Saw Point	336	None	Mississippi	New Augusta	Perry County	336	395472	None
38968	Spring	327.44	None	Montana	Plentywood	Sheridan County	327.44	385396.88	None

Finally, `st.number_input()` takes an integer input from the user and uses that value to display all the properties of a wildfire event using `st.table()`.

### Get fire by ID

Enter the ID of the fire to get detailed information on the event

38144

	value
ID	38144
IncidentName	Border 2
Acres	6624.77
CreateDate	
CurrentDate	2025-04-27 18:19:54.334728
State	California
City	Jamul
County	San Diego County
Priority	6624.77
Estimated Cost	7797354.290000001
InitialLatitude	32.6511630001766
InitialLongitude	-116.859937999204
FireBehaviorGeneral	
FireCause	
FireCauseSpecific	
IncidentShortDescription	

The information displayed on this page can be immensely useful to firefighters and first responders who are looking for specific information regarding a certain wildfire. Moreover, again government agencies can view acreage covered and estimated cost of a particular wildfire to determine how much resource allocated is required for different areas.

## **Discussion and Conclusions**

One thing that could be improved with this project is the chosen data structure. The Stack structure was chosen and coded before we learned in class about the priority queue. The main setback with the `EventStack` implementation is that before adding items to the stack, an array of fires must first be sorted using the quicksort algorithm, causing an increase in computation time. If in the future we choose to use a priority queue, the binary heap will determine the position of a wildfire event as it gets added into the stack, reducing time complexity and eliminating the need for a quicksort algorithm.

Another thing that should be improved in the future is the implementation of the backend. As of now, there is only a frontend server that calls backend files whenever the server is started. There is no actual backend server being run. This makes it so the webpage cannot be updated as new wildfires are entered into the database unless you want to restart the app. Given more time, I would be able to implement a backend server that can dynamically update the data given to send to the frontend, however it was outside the scope of this project.

Various topics that were learned in the course are applied in the project. The most obvious being OOP design principles. Abstraction was used to hide the complexity of `classes.py` from the user on the front-end of the application, ensuring that the user only saw what they needed to and not all the complex functionality happening on the backend. Furthermore, time complexity was assessed for most of the functions used in the program. The largest time complexity for a function in the program was  $O(k*n)$  for `EventUtils.load_fires()`. This was due to `EventUtils.get_coordinates()` having a time complexity of  $O(k)$  and it being called  $n$  times. The stack data structure was another topic learned in class that was implemented in this project. Though it may have not been the most effective choice, the `EventStack` class still successfully handled all the wildfire data and was able to provide useful functionality that was used on the frontend. The most notable being the `EventStack.get_by_ID()` function, which returned a wildfire event based on a given ID in  $O(n)$  time.

## **Overall Quality of Report and Project**

I believe we did a respectable job explaining each part of the project and explaining the uses of all the functions used in this project. It was a little difficult given there were a lot of functions described in `classes.py`, and sometimes it was tricky to explain how other files in the project call on this function to get results. Furthermore, while there are a lot of comments going on throughout the project code, it is possible some of the comments are not very clear at explaining what a function does or where some information may be coming from. However, I think the project report does a good job at clearing up confusions that may arise from just looking at the code.

All issues accounted for; our project successfully implemented what we intended it to do. Meaningful wildfire data and statistics is displayed to anyone that uses our app. For any fire in the U.S. a user can search it up in the User Dashboard page and look at specific information regarding that fire. We believe the most important statistic displayed by this project is the “Estimated Suppression Cost” number. As of now, the estimated suppression cost for all the fires is almost \$360 million. This really puts into perspective the scale of what damage these fires can do. Not only does this project act as a useful resource for firefighters and other government agencies, but it also raises awareness to this large issue that many people don’t think of on a

regular basis. Hopefully, people can look at this project and gain a better understanding of the seriousness of wildfires and recognize the need for action to put these fires to rest.