Project Summary and Documentation

Created By	
Stakeholders	
Status	
Type	Architecture Overview
Created	@May 11, 2022 3:33 PM
Last Edited Time	@May 16, 2022 5:12 PM
Last Edited By	

Introduction:

The Timekeeper is a day-planner PDA that assists the user with keeping track of when to leave for certain events.

Motivations:

During the day, we have many different obligations and places to be. It can be difficult to remember all of these things and get to our desired locations on time. In order to mitigate these challenges, we created a PDA that is able to send notifications to tell the user when they have to leave to arrive on time to different events during the day.

Project Architecture

In order to implement a system that is able to perform the above mentioned activities, we separated the functionality into a few different modules/entities.

- Weatherman
- Navigator
- Scheduler
- UI
- Timekeeper

Each of these modules has a different job and helps the user in different ways to get through the day.

Active Entities:

Active entities can be defined as entities/modules that do work other than just collect information and store it. In our project each module is an active entity and stores some portion of a single user's context.

Weatherman:

The weatherman provides the user with constant weather updates based on the user's current location as well as weather forecasts for where they will be during the day based on the events in their schedule. This also includes displaying helpful tips such as "Its raining today! Remember to bring an umbrella!" or "Be careful! It's really windy today!". This is achieved by polling OpenWeather, a weather API that allows developers to access weather forecasts based on latitude and longitude. The weatherman also communicates with the web server (to display notifications to the user) and the timekeeper.

Navigator:

The Navigator provides accurate traffic and travel time information based on the user’s preferred mode of travel. This is achieved by polling the DistanceMatrix API which allows developers to get estimates for travel time based on preferred mode of transportation.

Scheduler:

The scheduler handles the addition of new events into the system. When a user wants to add a new event the scheduler checks if there are any conflicts and then adds an event. This action also sends out querying messages to the navigator and the weatherman for updated information relating to the event. The scheduler also handles deleting events

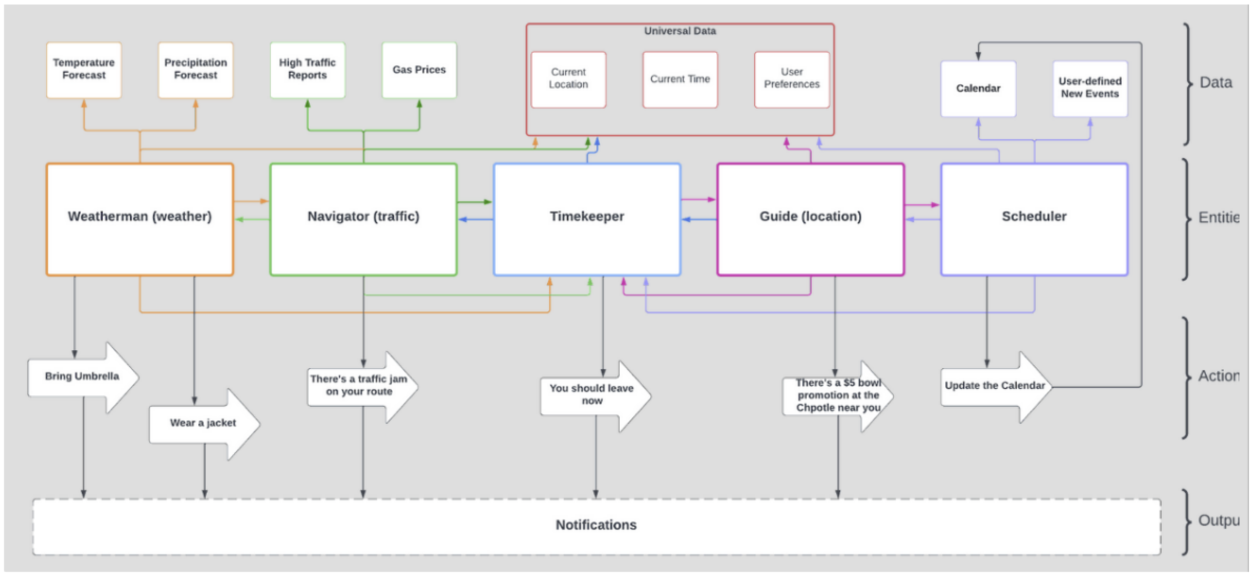
Timekeeper:

The timekeeper consolidates all the information from the weatherman, navigator and scheduler to send notifications to the user such that they leave at an appropriate time to reach their destination. The timekeeper uses the weather information and the travel time/traffic information to create an estimate of the travel time. The timekeeper polls the weatherman and navigator for more current information closer to the original estimated departure time. Then it notifies the user 5 minutes prior to the time they should leave (this time can be parametrized by the user).

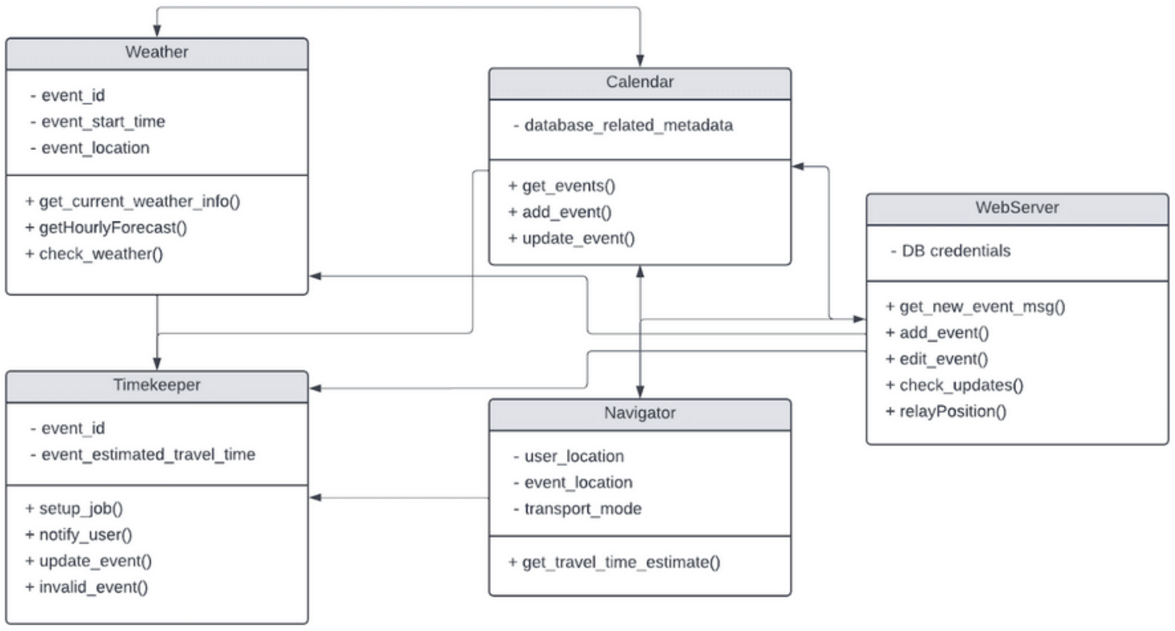
Web Server:

The web server displays notifications to the user and also provides the user with an interface to modify their daily schedule.

Following is a basic overview of the architecture:



Following is a basic UML diagram showing the most important parts of our design:



Overview of Implementation Details:

- Python and MPI to create a multi-process environment
- PostgreSQL to store event information
- Web server created with flask to create a UI for the use

Message and Class Formats

```
# MESSAGE FORMATS

# Sent by: Scheduler *initially sent from flask to scheduler, then scheduler formats + forwards
# Received by: Weatherman, Navigator
Msg_type.NEW_EVENT:
  msg = event_object

# Sent by: Scheduler
# Received by: Weatherman, Navigator
# Weatherman and Navigator delete the event when they receive this message
Msg_type.DELETE_EVENT:
  msg = {"event_id": event_id}

# Sent by: Weatherman, Navigator, Scheduler
# Received by: Timekeeper
# Timekeeper prints out that Weatherman, Scheduler and Navigator have been intialized
Msg_type.INITIALIZED:
msg = {"event_id": event_id,
      "msg": "weatherman_intialized", # or navigator_intialized or scheduler_intialized
      "date": datetime.now()}

Msg_type.STATUS = {}

# Sent by: Weatherman
# Received by: Timekeeper
Msg_type.UPDATE_EVENT_WEATHER:
  msg = event_object # event stores updated weather forecast for the start time of the event

# Sent by: Weatherman
# Received by: Web, Timekeeper
Msg_type.UPDATE_CURRENT_WEATHER:
  msg = weather_object # weather object that stores information about the weather at the users
            # current location

# Sent by: Weatherman
# Received by: Web, Timekeeper
Msg_type.WEATHER_NOTIFICATION:
  msg = {"alert_msg": str,
        "notification":str,
        "is_alert": bool}

# Sent by: Navigator
# Received by: Timekeeper, Web
Msg_type.UPDATE_ESTIMATE:
  msg = event_object # event stores updated estimate from users current location
          # to the event location of the next closest event

# Sent by: Navigator
# Received by: Timekeeper, Web
Msg_type.RESPONSE_ESTIMATE:
  msg = event_object #event stores the updated estimate

# Sent by: Scheduler
# Received by: Weatherman, Navigator, Timekeeper
Msg_type.UPDATE_USER_LOCATION:
 msg = location_object # location object that stores lat, lon, and address

# Sent by: Timkeeper
# Received by: Navigator
Msg_type.REQUEST_ESTIMATE:
  msg = event_object

# Sent by: Timekeeper
# Received by: Weatherman
Msg_type.REQUEST_WEATHER:
  msg = event_obejct



Send: Msg_type.UPDATE_USER_LOCATION
Should Receive: Msg_type.UPDATE_CURRENT_WEATHER, Msg_type.UPDATE_ESTIMATE

Send: Msg_type.NEW_EVENT
Should Receive: Msg_type.UPDATE_EVENT_WEATHER, Msg_type.RESPONSE_ESTIMATE

Send: Msg_type.REQUEST_WEATHER
Should Receive: Msg_type.UPDATE_EVENT_WEATHER

Send: Msg_type.REQUEST_ESTIMATE
Should Receive: Msg_type.RESPONSE_ESTIMATE
```

```
# CLASS STRUCTURES:

class Weatherman:
  def __init__(self):
    self.events = {}
    self.curr_location = Location()
    self.curr_weather = Weather()
    self.earliest_event_id = 0
    self.next_event = None
    self.time_last_notification = 0
    self.last_alert_msg = ""
    self.notification_flags = {'temp': False, 'pop': False, 'humidity': False, 'wind_speed': False}

class Navigator:
  def __init__(self):
    self.locations = {}
    self.curr_location = Location()
    self.earliest_event_id = 0
    self.next_event = None
```

```
class Calendar:
  def __init__(self, actor):
    self.user_location = None
    self.actor = actor

class Timekeeper:
  def __init__(self, actor):
    self.next_event = None
    self.events = {}
    self.scheduler = BackgroundScheduler()
    self.scheduler.start()
    self.actor = actor

class Event():
  def __init__(self, event_id, preference, weather, event_location, user_location, start_time, end_time, title, description):
    self.event_id = event_id
    self.preference = preference
    self.weather = weather # weather object
    self.event_location = event_location # location object
    self.user_location = user_location # location object
    self.start_time = start_time
    self.end_time = end_time
    self.title = title
    self.description = description
    self.estimate = None # type datetime.timedelta
    self.scheduled_update_time = None
    self.scheduler_status = None


class Weather:
  def __init__(self,temp = None, pop = None, humidity = None, wind_speed = None, description = None, weather_icon_url = None):
    self.temp = temp
    self.pop = pop
    self.humidity = humidity
    self.wind_speed = wind_speed
    self.description = description
    self.weather_icon_url = weather_icon_url

class Location:
  def __init__(self, lat=None, lon=None, address=None):
    self.lat = lat
    self.lon = lon
    self.address = address


class Actor():
    def __init__(self, rank, comm):
        self.rank = rank
        self.comm = comm


class Message():
    def __init__(self, **kwargs):
        self.msg = None
        self.msg_type = None
        self.sender = None
        self.receiver = None
```