PRODUCER.PY: This file contains the `KafkaEventProducerSimulator` class, which simulates bus events (location and speed) and sends them to Kafka topics

`Producer` from `confluent_kafka` for Kafka interaction
`json` for JSON encoding/decoding
`random` for generating random values
`time` for timestamp and sleep functionality

```
        def __init__(self, bootstrap_servers, num_assets=10):
```

it takes bootstrap_servers as a parameter and num_assets and defaults them to a value of 10

```
        self.producer = Producer({'bootstrap.servers': bootstrap_servers})
```

creates a Kafka Producer instance with the specified bootstrap servers.

```
        self.asset_ids = [f'BUS-{random.randint(1000, 9999)}' for _ in range(num_assets)]
```

generates a list of random bus IDs (e.g., 'BUS-1234') for the specified number of assets

```
        self.routes = {asset_id: self.generate_route() for asset_id in self.asset_ids}
```

creates a dictionary where each asset ID is associated with a randomly generated route.

```
        print(f"Producer-Simulator initialized with bootstrap servers: {bootstrap_servers}")
        print(f"Simulating {num_assets} assets: {self.asset_ids}")
```

these lines print the information.

```
    def delivery_report(self, err, msg):
```

method is a callback function for the Kafka producer to report on message delivery status.

```
    def produce_event(self, topic, event):
```

method produces a single event to a specified Kafka topic.

```
    def generate_route(self):
        return [(random.uniform(33.5, 34.5), random.uniform(-84.5, -83.5)) for _ in range(10)]
```

method generates a random route consisting of 10 latitude-longitude pairs

```
    def produce_bus_event(self, asset_id, event_count):
```

method produces location and speed events for a specific bus (asset).

```
        timestamp = int(time.time() * 1000)
```

gets the current timestamp in milliseconds.

```
        route = self.routes[asset_id]
        current_position = route[event_count % len(route)]
```

determine the current position of the bus based on its route and the event count.

```
        location_event = {
            'type': 'asset_location',
            'assetId': asset_id,
            'timestamp': timestamp,
            'latitude': current_position[0],
            'longitude': current_position[1]
        }
```

creates a location event for the bus.

```
        speed_event = {
            'type': 'asset_speed',
            'assetId': asset_id,
            'timestamp': timestamp,
            'speed': random.uniform(0, 65)
        }
```

creates a speed event for the bus with a random speed between 0 and 65.

```
        self.produce_event('asset_location', location_event)
        self.produce_event('asset_speed', speed_event)
```

produce the location and speed events to their respective Kafka topics.

```
    def run_simulation(self, num_events, interval):
```

method runs the simulation for a specified number of events and time interval.

```
        for i in range(num_events):
            asset_id = random.choice(self.asset_ids)
            self.produce_bus_event(asset_id, i)
            time.sleep(interval)
```

loop produces events for randomly chosen buses at the specified interval.

```
        self.producer.flush()
```

ensures all messages are sent before the simulation ends.

```
if __name__ == "__main__":
    producer_simulator =
KafkaEventProducerSimulator('localhost:9092', num_assets=5)
    producer_simulator.run_simulation(25, 1)
```

simulates 5 buses, producing 25 events (50 total, as each event produces both location and speed data) with a 1-second interval between events.

CONSUMER.PY: This file contains the `KafkaEventConsumer` class, which is responsible for consuming events from Kafka topics.

`Consumer` and `KafkaError` from `confluent_kafka` for Kafka interaction

`json` for JSON decoding

```
    def __init__(self, bootstrap_servers, group_id, topics):
```

takes `bootstrap_servers`, `group_id`, and `topics` as parameters.

```
        self.consumer = Consumer({
            'bootstrap.servers': bootstrap_servers,
            'group.id': group_id,
            'auto.offset.reset': 'earliest'
        })
```

creates a Kafka Consumer instance with the bootstrap servers, group ID, and sets it to read from the earliest available offset.

```
        self.consumer.subscribe(topics)
```

subscribes the consumer to the specified Kafka topics.

```
        print(f"Consumer initialized with bootstrap servers:
{bootstrap_servers}")
        print(f"Subscribed to topics: {topics}")
```

lines print initialization information.

```
    def consume_events(self):
```

method is a generator that continuously consumes events from the subscribed Kafka topics.

```
        print("Starting to consume events...")
        while True:
```

starts an infinite loop to continuously consume messages.

```
            msg = self.consumer.poll(1.0)
```

polls the Kafka broker for new messages with a timeout of 1 second.

```
            if msg is None:
                continue
```
if no message is received, the loop continues to the next iteration.

```
            if msg.error():
                if msg.error().code() ==
KafkaError._PARTITION_EOF:
                    print("Reached end of partition")
                    continue
                else:
                    print(f"Consumer error: {msg.error()}")
                    break
```
this block handles any errors that occur during message consumption - if the end of a partition is reached, it continues; for other errors, it breaks the loop.

```
            try:
                event = json.loads(msg.value().decode('utf-8'))
                print(f"Received event: {event}")
                yield event
```
this block decodes the received message as JSON and yields the event - if it works, it prints the received event.

```
            except json.JSONDecodeError:
                print(f"Failed to decode message:
{msg.value()}")
```
if JSON decoding fails, it prints an error message.

```
    def close(self):
        print("Closing consumer...")
        self.consumer.close()
```
method closes the Kafka consumer connection.

```
    consumer = KafkaEventConsumer('localhost:9092',
'gcps_team2', ['asset_location', 'asset_speed'])
```
creates an instance of the `KafkaEventConsumer`, connecting to a local Kafka broker, using the group ID 'gcps_team2', and subscribing to the 'asset_location' and 'asset_speed' topics.

```
    try:
        for event in consumer.consume_events():
            print(f"Main consumer loop: {event}")
```
starts consuming events and prints each received event.

```
    except KeyboardInterrupt:
        print("Stopping consumer...")
```

catches a keyboard interrupt (Ctrl+C) to stop the consumer.

```
    finally:
        consumer.close()
```

ensures that the consumer is closed properly, whether the script exits normally or due to an exception.

MAIN.PY: Main application file that ties everything together. It uses the consumer to receive events, the processor to validate and format the data, and then inserts the processed data into a database.

`KafkaEventConsumer` from a local `consumer` module
`DataProcessor` from a local `processor` module
`pyodbc` for database operations

```
    def __init__(self, connection_string):
        self.connection_string = connection_string
```

constructor for the `DatabaseManager` class. It takes a `connection_string` parameter and stores it as an instance variable.

```
    def insert_events(self, location_event, speed_event):
```

this method is responsible for inserting location and speed events into the database.

```
        query = """
            INSERT INTO asset_events (asset_id, event_type,
latitude, longitude, speed, timestamp)
            VALUES (--------)
        """
```

SQL query template for inserting events.

```
        with pyodbc.connect(self.connection_string) as conn:
            with conn.cursor() as cursor:
                cursor.execute(query,
                    location_event['asset_id'], 'location',
location_event['latitude'], location_event['longitude'], None,
location_event['timestamp'],
                    speed_event['asset_id'], 'speed', None,
None, speed_event['speed'], speed_event['timestamp'])
            conn.commit()
```

establishes a database connection, executes the insert query with the event data, and commits the transaction.

```
        def main():
```

defines the main function of the script.

```
    consumer = KafkaEventConsumer('localhost:9092',
'gcps_team2', ['asset_location', 'asset_speed'])
    processor = DataProcessor()
    db_manager = DatabaseManager('DRIVER={ODBC Driver 17 for SQL
Server};---------'
```

initialize the Kafka consumer, data processor, and database manager.

```
    print("Starting main application...")
    event_pairs = {}
```

prints a start message and initializes a dictionary to store event pairs.

```
    try:
        for event in consumer.consume_events():
```

starts a loop to consume events from Kafka.

```
            processed_event, error =
processor.process_event(event)
            if error:
                print(f"Error processing event: {error}")
                continue
```

processes each event and handles any processing errors.

```
            asset_id = processed_event['asset_id']
            event_type = processed_event['type']

            if asset_id not in event_pairs:
                event_pairs[asset_id] = {}
            event_pairs[asset_id][event_type] = processed_event
```

organizes events by asset ID and event type.

```
            if len(event_pairs[asset_id]) == 2:
                location_event =
event_pairs[asset_id].get('asset_location')
                speed_event =
event_pairs[asset_id].get('asset_speed')
```

checks if both location and speed events are available for an asset.

```
                if location_event and speed_event and
location_event['timestamp'] == speed_event['timestamp']:
                    print(f"Inserting paired events for asset
{asset_id}")
                    db_manager.insert_events(location_event,
speed_event)
                    del event_pairs[asset_id]
                else:
                    print(f"Incomplete or mismatched events for
asset {asset_id}. Waiting for matching event.")
```

block inserts paired events into the database if they match and have the same timestamp.

```
    except KeyboardInterrupt:
        print("Application interrupted. Shutting down...")
    finally:
        consumer.close()
        print("Application shut down complete.")
```

handles shutdown of the application on keyboard interrupt.

```
if __name__ == "__main__":
    main()
```

ensures the `main()` function is called only if the script is run directly.

PROCESSOR.PY: This file contains the `DataProcessor` class, which processes and validates the events received from Kafka. It ensures that the data is in the correct format before it's inserted into the database.

`datetime` for handling timestamps

```
    def process_event(self, event):
        print(f"Processing event: {event}")
        if 'type' not in event:
            return None, "Missing event type"
```

method is the main entry point for processing events. It checks if the event has a 'type' field and prints the event for logging purposes.

```
        if event['type'] == 'asset_location':
            return self.process_location_event(event)
        elif event['type'] == 'asset_speed':
            return self.process_speed_event(event)
        else:
            return None, f"Unknown event type: {event['type']}"
```

routes the event to the appropriate processing method based on its type. If the type is unknown, it returns an error.

```
    def process_location_event(self, event):
        print("Processing location event")
        required_fields = ['assetId', 'latitude', 'longitude',
'timestamp']
        if not all(field in event for field in required_fields):
            return None, "Missing required fields in location
event"
```

method processes location events. It first checks if all required fields are present in the event.

```
        processed_event = {
            'asset_id': event['assetId'],
            'latitude': event['latitude'],
            'longitude': event['longitude'],
            'timestamp':
datetime.fromtimestamp(event['timestamp'] / 1000.0)
        }
        print(f"Processed location event: {processed_event}")
        return processed_event, None
```

if all required fields are present, it creates a processed event dictionary -- it converts the timestamp from milliseconds to a datetime object.

```
    def process_speed_event(self, event):
        print("Processing speed event")
        required_fields = ['assetId', 'speed', 'timestamp']
        if not all(field in event for field in required_fields):
            return None, "Missing required fields in speed
event"
```

method processes speed events, similar to the location event processing.

```
        processed_event = {
            'asset_id': event['assetId'],
            'speed': event['speed'],
            'timestamp':
datetime.fromtimestamp(event['timestamp'] / 1000.0)
        }
        print(f"Processed speed event: {processed_event}")
        return processed_event, None
```

if all required fields are present for a speed event, it creates a processed event dictionary.

```
if __name__ == "__main__":
    processor = DataProcessor()
    test_events = [
        {
            'type': 'asset_location',
            'assetId': '12345',
            'latitude': 33.9519,
            'longitude': -83.9921,
            'timestamp': 1623456789000
        },
        {
            'type': 'asset_speed',
            'assetId': '67890',
            'speed': 55.5,
            'timestamp': 1623456790000
        },
        {
            'type': 'unknown_type',
            'assetId': '13579'
        }
    ]
```

this block is for testing the `DataProcessor` class -- creates an instance of the processor and defines a list of test events.

```
    for event in test_events:
        processed_event, error = processor.process_event(event)
        if error:
            print(f"Error processing event: {error}")
        else:
            print(f"Successfully processed event:
{processed_event}")
```

oop processes each test event and prints the result or error message.

APP.JSX: Creates a React component that renders a map with a moving bus marker, simulating real-time updates from a WebSocket connection. It uses Mapbox GL JS for the map rendering and includes features like resetting the map view and displaying current bus information.

first set of lines import necessary React hooks, the Mapbox GL JS library, its CSS, and a local CSS file.

```
     const INITIAL_CENTER = [-83.9921, 33.9519]; // Coordinates
for Gwinnett County
     const INITIAL_ZOOM = 13;
```

define the initial map center and zoom level.

```
     function App() {
```

defines the main App component.

```
     const mapRef = useRef(null);
     const mapContainerRef = useRef(null);
     const busMarkerRef = useRef(null);
     const wsRef = useRef(null);
```

These lines create refs for the map, map container, bus marker, and WebSocket connection.

```
     const [center, setCenter] = useState(INITIAL_CENTER);
     const [zoom, setZoom] = useState(INITIAL_ZOOM);
     const [busPosition, setBusPosition] =
useState(INITIAL_CENTER);
     const [busSpeed, setBusSpeed] = useState(0);
```

These lines define state variables for the map center, zoom level, bus position, and bus speed.

```
       useEffect(() => {
         mapboxgl.accessToken =
     'pk.eyJ1Ijoic2FyYWhmYXNoaW5hc2kiLCJhIjoiY20xczg0cWRyMDNtOTJ
     sb2R6cmNiZmRyNyJ9.Utvb8kECGGDYQljL0fknfA';
```

sets up the Mapbox GL JS map and starts by setting the access token.

```
         if (!mapContainerRef.current) {
           console.error('Map container ref is null');
           return;
         }
```

checks if the map container ref is available.

```
         mapRef.current = new mapboxgl.Map({
      container: mapContainerRef.current,
      style: 'mapbox://styles/mapbox/streets-v11',
      center: center,
      zoom: zoom
    });
```

creates a new Mapbox GL JS map instance.

```
    mapRef.current.on('load', () => {
      console.log('Map loaded');

      if (!mapRef.current) {
        console.error('Map reference is null');
        return;
      }
```

sets up an event listener for when the map is loaded.

```
      const el = document.createElement('div');
      el.className = 'bus-marker';
      el.style.backgroundImage =
'url(https://hebbkx1anhila5yf.public.blob.vercel-
storage.com/bus-i33k23ytUTsMTcfzdld0jMMkEOtT6D.png)';
      el.style.width = '40px';
      el.style.height = '40px';
      el.style.backgroundSize = 'cover';
```

creates a DOM element for the bus marker.

```
      busMarkerRef.current = new mapboxgl.Marker(el)
        .setLngLat(busPosition)
        .addTo(mapRef.current);

      console.log('Bus marker added');
    });
```

adds the bus marker to the app

```
    return () => {
      if (mapRef.current) {
        mapRef.current.remove();
      }
    };
  }, []);
```

this function removes the map when the component unmounts.

```
  useEffect(() => {
    const simulateWebSocket = () => {
      return {
        onmessage: null,
        send: () => {},
        close: () => {},
      };
    };

    wsRef.current = simulateWebSocket();
```

this effect simulates a WebSocket connection

```
    const handleMessage = (event) => {
      const data = JSON.parse(event.data);
      if (data.type === 'asset_location') {
        setBusPosition([data.longitude, data.latitude]);
      } else if (data.type === 'asset_speed') {
        setBusSpeed(data.speed);
      }
    };

    wsRef.current.onmessage = handleMessage;
```

sets up a message handler for the simulated WebSocket

```
    const interval = setInterval(() => {
      if (wsRef.current && wsRef.current.onmessage) {
        wsRef.current.onmessage({ data: JSON.stringify({
          type: 'asset_location',
          longitude: busPosition[0] + (Math.random() - 0.5) *
0.0001,
          latitude: busPosition[1] + (Math.random() - 0.5) *
0.0001
        })});
        wsRef.current.onmessage({ data: JSON.stringify({
          type: 'asset_speed',
          speed: Math.random() * 30
        })});
      }
    }, 1000);
```

simulates receiving WebSocket messages every second

```
    return () => {
      clearInterval(interval);
      if (wsRef.current) {
        wsRef.current.close();
      }
    };
  }, [busPosition]);
```

clears the interval and closes the WebSocket when the component unmounts

```
  useEffect(() => {
    if (busMarkerRef.current) {
      busMarkerRef.current.setLngLat(busPosition);
      console.log('Bus marker position updated:', busPosition);
    }
  }, [busPosition]);
```

This effect updates the bus marker position whenever busPosition changes.

```
  const handleButtonClick = () => {
    if (mapRef.current) {
      mapRef.current.flyTo({
        center: INITIAL_CENTER,
        zoom: INITIAL_ZOOM
      });
    }
  };
```

this function handles the reset button click, flying the map back to the initial view.

```
  return (
    <>
      <div className="sidebar">
        Longitude: {busPosition[0].toFixed(4)} | Latitude:
{busPosition[1].toFixed(4)} | Zoom: {zoom.toFixed(2)}
        <br />
        Bus Speed: {busSpeed.toFixed(2)} mph
      </div>
      <button className="reset-button"
onClick={handleButtonClick}>
        Reset
      </button>
      <div id="map-container" ref={mapContainerRef} style={{
width: '100vw', height: '100vh' }} />
    </>
  );
}

export default App;
```

this renders the component, including a sidebar with bus information, a reset button, and the map container.