Tech Stack and Database Choice:

For implementing the RESTful service managing and querying event data, I've chosen the following tech stack and database:

Tech Stack:

Programming Language: Python - for its simplicity, readability, and the availability of libraries/frameworks for building RESTful APIs.
Web Framework: Flask - a lightweight and flexible framework that makes it easy to build RESTful APIs quickly.
HTTP Client: Requests library - for making HTTP requests to external APIs.
Data Serialization: JSON - for exchanging data between the client and the server due to its simplicity and compatibility with most programming languages.
Database:

SQLite: Considering the simplicity of the task and the provided dataset being relatively small, SQLite is a suitable choice. It's lightweight, requires no server setup, and is easy to integrate with Python using the sqlite3 module.
Design Decisions and Challenges:

Data Ingestion: The provided CSV dataset needs to be ingested into the system. This process involves parsing the CSV file and inserting its contents into the SQLite database. The challenge here is to handle errors and ensure data integrity during the ingestion process. This can be addressed by implementing error handling mechanisms and validation checks.

API Endpoints Design:

Data Creation API: A POST endpoint /events is designed to allow the addition of events into the system. The decision to use POST is appropriate as it conforms to the RESTful principles for creating a resource.
Event Finder API: A GET endpoint /events/find is designed to list events based on the user's geographical location and specified date. This endpoint follows RESTful principles and accepts query parameters for latitude, longitude, and date.
External API Integration:

External APIs for weather retrieval and distance calculation are integrated into the Event Finder API. This involves making HTTP requests to the external APIs and handling the responses appropriately. Error handling mechanisms are implemented to handle cases such as API failures or invalid responses.
Response Format and Error Handling:

Standard JSON response formats are used for both successful responses and error responses. This ensures consistency and ease of consumption by client applications.
Error handling mechanisms are implemented to provide meaningful error messages and appropriate HTTP status codes for different scenarios, such as invalid requests or internal server errors.
Security Considerations:

Since the task does not specify any authentication or authorization requirements, no security measures are implemented in this version. However, in a production environment, appropriate security measures like authentication, authorization, and input validation would be essential to ensure the integrity and confidentiality of the system.