**Answers**

***Ex1.***

**Deploying the Web Page Online:**

To deploy the web page online, we would need to set up a server infrastructure to host the application. We could use cloud service providers like AWS, Google Cloud Platform, or Microsoft Azure to deploy the application. These platforms offer services such as EC2 instances (AWS), Compute Engine (Google Cloud), or Virtual Machines (Azure) to host our web application. Once we have the infrastructure in place, we can deploy our application using web server technologies like Apache HTTP Server or Nginx. We would need to ensure that the server environment supports the programming language and frameworks used to develop the web page. For example, if we're using Python and Flask for the backend, we would need to install Python and Flask on the server. Additionally, we may need to set up a domain name and configure DNS settings to point to our server's IP address, allowing users to access the web page using a custom domain name.

**Populating the Dropdown Menu from a Database:**

To populate the dropdown menu with values from a database, we would first need to set up a database to store the values. We could use relational databases like MySQL, PostgreSQL, or SQLite, or NoSQL databases like MongoDB, depending on the requirements of our application. Once the database is set up, we would modify the backend code to query the database for the values to populate the dropdown menu. We could use SQL queries to retrieve the values from the database and then pass them to the frontend as JSON or other data formats. On the frontend side, we would modify the code to dynamically populate the dropdown menu with the values received from the backend. We could use JavaScript frameworks like React, Vue.js, or Angular to handle the dynamic rendering of the dropdown menu based on the data received from the backend. Additionally, we would need to set up API endpoints on the backend to handle requests from the frontend and retrieve data from the database. We could use web frameworks like Flask or Django in Python to create these API endpoints and handle the communication between the frontend and the database.

**Ex2.**

**Handling Arbitrary Nesting in JSON:**

To dynamically unnest data from a JSON with arbitrary levels of nesting, we would need to recursively traverse the JSON structure. We would start by checking each key in the JSON object to determine if it contains nested data (i.e., another dictionary or array). If it does, we would recursively call the function to handle the nested data until we reach the leaf nodes containing the desired values. We would then extract these values and construct the dataframe accordingly. Additionally, we might need error handling to account for variations in the JSON structure.

**Scaling the Application:**

To scale the application to handle data for hundreds of countries and refresh the database regularly, we could implement several strategies:

Asynchronous Processing: Use asynchronous programming techniques to parallelize API requests and database operations, allowing for faster processing of large volumes of data.

Batch Processing: Retrieve data for multiple countries in each API request and process them in batches, reducing the number of API calls and database transactions required.

Caching: Implement caching mechanisms to store previously fetched data temporarily, reducing the need to re-fetch data for countries that have not changed between refresh intervals.

Database Optimization: Optimize database operations by using efficient data loading methods, indexing relevant columns for fast queries, and partitioning data to distribute storage and processing load.

Infrastructure Scaling: Utilize cloud services for elastic scaling of resources based on demand, allowing the application to handle fluctuations in workload efficiently.

Monitoring and Alerting: Implement monitoring and alerting systems to detect performance bottlenecks, errors, or downtime, enabling timely intervention and optimization of the application.

**Ex3.**

**Handling Lazy Loaded Content:**

If the Wikipedia table is lazy loaded, meaning it appears after a few seconds from opening the page, we can still scrape the data using web scraping libraries like BeautifulSoup or Scrapy. To handle lazy loading, we may need to use libraries like Selenium that simulate a web browser to interact with the page and wait for the table to load completely before scraping. We can use explicit waits to wait for specific elements to appear on the page before proceeding with scraping.

**Efficient Processing for Thousands of Companies:**

To handle scraping data for thousands of companies efficiently, we can adopt several strategies:

Throttling and Rate Limiting: Implement throttling and rate limiting to control the rate of API requests made to avoid overwhelming the servers or violating API usage limits.

Parallel Processing: Use multiprocessing or asynchronous programming techniques to parallelize the scraping process, allowing multiple API requests to be made simultaneously and speeding up the overall data retrieval process.

Caching: Implement caching mechanisms to store previously fetched data temporarily, reducing the need to re-fetch data for companies that have not changed between script runs.

Batch Processing: Retrieve data for multiple companies in each API request or scraping operation, reducing the total number of requests made and minimizing overhead. Optimized

API Requests: Optimize API requests by batching multiple requests into a single HTTP call whenever possible or using efficient endpoints that provide multiple data points in a single response.

Error Handling and Retry Mechanisms: Implement robust error handling and retry mechanisms to handle transient errors or network issues gracefully, ensuring the script can recover from failures and continue processing without manual intervention.