Objective

Implement a simplified **Ad Exchange Auction Service** that simulates real-time bidding between multiple bidders based on supply and targeting data. Application must work fast with minimum response delay

---

Description

You need to implement a small web service that exposes **two endpoints**:

1. POST /bid
   - ○ Starts a new auction for a given supply ID.
   - ○ Determines eligible bidders and runs an auction to pick a winner.
   - ○ Returns the winning bidder and price.
2. GET /stat
   - ○ Returns overall service statistics such as total requests, bidder wins, and revenue — grouped **per supply**.

Functional Requirements

1. Static Database Layer

- Instead of a real database, use a **static JSON file** (kept in memory) to describe the relationship between **supplies** and **bidders**.

  Example:

JSON

```
{
  "supplies": {
    "supply1": ["bidder1", "bidder2", "bidder3"],
    "supply2": ["bidder2", "bidder3"]
  },
  "bidders": {
    "bidder1": { "country": "US" },
    "bidder2": { "country": "GB" },
    "bidder3": { "country": "US" }
  }
}
```

2. /bid Endpoint

Method: POST /bid

Request body (JSON):

JSON

```
{
  "supply_id": "supply1",
  "ip": "123.45.67.89",
  "country": "US"
}
```

Logic:

1. **Validate** that the supply_id exists in the static JSON.
2. Apply **rate limiting** – max **3 requests per minute per IP**.
    - On limit exceed, return HTTP 429 Too Many Requests.
3. **Select eligible bidders**:
    - From the supply's list of bidders.
    - Filter only those whose country matches the request's country.
4. **Run the auction**:
    - Each eligible bidder:
        - Generates a random bid price between 0.01–1.00.
        - Skips bidding in **30% of cases** (no bid).
    - The highest bid wins.
    - If all bidders skip or none are eligible, return an error.

JSON

```
{
  "winner": "bidder2",
  "price": 0.83
}
```

6. **Logging (to terminal)** example:

Shell

```
Auction for supply1 (country=US):
bidder1 - price 0.10
bidder2 - price 0.23
bidder3 - no bid
Winner: bidder2 (0.23)
```

3. /stat Endpoint

Method: GET /stat

Response:

- Returns statistics grouped **per supply**

Example:

JSON

```
{
  "supply1": {
    "total_reqs": 10,
    "reqs_per_country": { "US": 5, "GB": 5 },
    "bidders": {
      "bidder1": { "wins": 2, "total_revenue": 0.4, "no_bids": 3 },
      "bidder2": { "wins": 3, "total_revenue": 0.7, "no_bids": 1 },
      "bidder3": { "wins": 0, "total_revenue": 0.0, "no_bids": 6 }
    }
  },
  "supply2": {
    "total_reqs": 4,
    "reqs_per_country": { "GB": 4 },
    "bidders": {
      "bidder1": { "wins": 0, "total_revenue": 0.0, "no_bids": 0 },
      "bidder2": { "wins": 1, "total_revenue": 0.2, "no_bids": 1 },
      "bidder3": { "wins": 0, "total_revenue": 0.0, "no_bids": 3 }
    }
  }
}
```

Non-functional Requirements

- Framework choice is **up to you** (Flask, FastAPI, aiohttp, etc.).

- The app should be **runnable locally** (e.g., python main.py or uvicorn main:app).

- Keep the code **clean, modular, and readable** — structured in logical layers (rate limiter, data access, auction logic, stats).

- Use **in-memory (redis, lru)** data structures for both rate limiting and statistics.

- Include **logging** of auctions to the terminal for verification.
- Add comments (or readme) describing on how this application can scale, how can we store analytics data

Optional requirements

- Add tmax to /bid endpoint parameters. This will represent time in millis - maximum time for a bidder to respond during auction process. Add latency simulation for a bidder to respond. Log message - if delay happened during auction process. Log timeouts count for /stat endpoint
- Instead of static supply and bidder JSON file - use an SQL database. Add create-table statements and insert-into statements into readme file to setup database