# Ad Impression Counter - Test Assignment

Create a concurrent service that tracks ad impressions across multiple campaigns in real-time, with mechanisms to avoid counting duplicate impressions from the same user within a specified time period.

*Note: Consider the task as a guideline. You have a great freedom to experiment, to showcase interesting snippets, ideas, workarounds.*

## Task

The goal is to build an in-memory impression counter that supports concurrent processing, including handling duplicate impressions efficiently and providing a REST API to manage campaigns, track impressions, and retrieve statistics.

## Instructions

1. **Implement Core Data Structures**:
   - `Campaign`: Represents an advertising campaign.
   - `Impression`: Represents each ad view, including `userID` and `timestamp`.
   - `Stats`: Stores aggregated impression statistics for each campaign.
2. **Duplicate Handling**:
   - Track impressions uniquely for each `userID` to avoid duplicate counting.
   - Implement a TTL (Time-To-Live) mechanism to discard duplicate records after a set time period (e.g., one hour).
   - Ensure that each unique impression from a user is only counted once within the specified TTL.
3. **Concurrency**:
   - Use concurrency controls (e.g., channels, mutexes) to handle multiple impressions tracked simultaneously without race conditions.
   - Make the impression tracking system thread-safe.
4. **Build REST API Endpoints**:
   - `POST /api/v1/campaigns` — Register a new campaign.
     - Request Body: `CreateCampaignRequest`
   - `POST /api/v1/impressions` — Track a new impression.
     - Request Body: `TrackImpressionRequest`
   - `GET /api/v1/campaigns/{id}/stats` — Get impression statistics for a specific campaign.
     - Response Body: `Stats`

### Data Structures

```
// Campaign represents an advertising campaign
type Campaign struct {
    ID        string    `json:"id"`
    Name      string    `json:"name"`
    StartTime time.Time `json:"start_time"`
}

// Impression represents a single ad view
type Impression struct {
    CampaignID string    `json:"campaign_id"`
    Timestamp  time.Time `json:"timestamp"`
    UserID     string    `json:"user_id"`
    AdID       string    `json:"ad_id"`
}

// Stats represents aggregated impression statistics
type Stats struct {
    CampaignID  string `json:"campaign_id"`
    LastHour    int64  `json:"last_hour"`
    LastDay     int64  `json:"last_day"`
    TotalCount  int64  `json:"total"`
}
```

### API Requests/Responses

```
type CreateCampaignRequest struct {
    Name      string    `json:"name"`
    StartTime time.Time `json:"start_time"`
}

type TrackImpressionRequest struct {
    CampaignID string `json:"campaign_id"`
    UserID     string `json:"user_id"`
    AdID       string `json:"ad_id"`
}

// Error response format
```

```
type ErrorResponse struct {
    Error string `json:"error"`
    Code  int    `json:"code"`
}
```

## Technical Focus

- **Concurrency**: Use channels and mutexes to handle concurrent updates safely.
- **Duplicate Check**: Implement a TTL for `userID` records to prevent duplicate counting within a specified time period (e.g., one hour).
- **API Design**: Structure endpoints for intuitive use and handle edge cases.
- **Project Structure**: Organize code with scalability in mind, allowing for future growth.
- **Minimum Dependencies**: Prefer standard, or well-known packages.

## Example Workflow

1. **Register a Campaign**:
   - Send a `POST` request to `/api/v1/campaigns` with `CreateCampaignRequest` in the JSON body.
   - Returns the new campaign's ID and a success message.
2. **Track Impressions**:
   - Send a `POST` request to `/api/v1/impressions` with `TrackImpressionRequest` in the body.
   - Check for duplicate impressions (based on `userID` and TTL).
   - Returns a success message.
3. **Get Campaign Statistics**:
   - Send a `GET` request to `/api/v1/campaigns/{id}/stats`.
   - Returns the aggregated `Stats` for the campaign, including `LastHour`, `LastDay`, and `TotalCount`.

## Evaluation Criteria

1. **Concurrency Handling**:
   - Are channels and mutexes used effectively to manage concurrent updates and prevent race conditions?
   - Is the TTL for duplicate check implemented correctly?
2. **API Design**:
   - Are the endpoints structured in a RESTful manner?
   - Is duplicate detection efficient and scalable?
3. **Code Organization and Readability**:
   - Is the project organized and structured to support future expansion?
   - Is the code clear and well-commented?

4. **Error Handling and Documentation**:
    - Are error cases handled appropriately, with clear messages?
    - Is the code and API well-documented?
5. **Testing**:
    - Include tests to validate core functionality (e.g., concurrent impressions and duplicate check).