Overview:

This system is designed to allow users to compare two products from Amazon based on their reviews and suggest the better option based on the user's expectations (e.g., cheap, durable, etc.). The process involves:

1. **Client-side**: The user inputs Amazon product URLs and their expectations.
2. **Server-side**: The server fetches product reviews, processes them, and generates a recommendation.
3. **Client-side**: The recommendation is displayed back to the user.

Client-Side (React: ProductComparer Component)

1. **Component Initialization**:

   - The React component (`ProductComparer`) initializes several state variables to manage the form inputs and comparison result:
     - `url1`, `url2`: Holds the Amazon product URLs entered by the user.
     - `userExpectation`: Holds the user's expectation (e.g., "cheap", "durable").
     - `comparison`: Holds the final comparison result fetched from the server.
     - `isLoading`: Tracks whether the comparison is in progress, to display a loading indicator.
2. **User Input**:

   - The user enters two Amazon product URLs and their expectations in the form fields.
   - The form fields are controlled components with state tied to `url1`, `url2`, and `userExpectation`.
3. **Triggering the Comparison**:

   - When the user clicks the **Compare Products** button:
     - **`handleCompare`** function is triggered.
     - This sets `isLoading` to `true` (to show the loading indicator).

A **POST request** is sent to the server with the following payload:

```
 {
  "url1": "<Amazon URL 1>",
  "url2": "<Amazon URL 2>",
  "userExpectation": "<user's product expectation>"
}
```

   - The request is sent using the `fetch` API to the `/api/compare` endpoint.

4. **Displaying the Result**:

   - After the request is completed, the server sends back a JSON response containing the recommendation (e.g., product suggestion).
   - The `comparison` state is updated with the received recommendation.
   - The result is displayed in a `<p>` tag with the class `whitespace-pre-wrap` to preserve line breaks and formatting.

Server-Side (Next.js: route.js API Endpoint)

1. **Receiving the Request**:

    - The server-side code listens for **POST requests** at the `/api/compare` endpoint. The `POST` request contains:
        - `url1`: Amazon URL of the first product.
        - `url2`: Amazon URL of the second product.
        - `userExpectation`: A string describing the user's preference or expectation for the product (e.g., "cheap", "durable").
2. **Extracting ASINs**:

    - The `extractASIN` function is used to extract the unique ASIN (Amazon Standard Identification Number) from each of the URLs. This is done using a regular expression to find the ASIN in the Amazon URL.

Example of ASIN extraction:

```
 function extractASIN(amazonUrl) {
  const regex = /\/([A-Z0-9]{10})(?:[\/?]|$)/;
  const match = amazonUrl.match(regex);
  return match ? match[1] : null;
}
```

-
- If either of the URLs does not contain a valid ASIN, the server responds with an error message.

3. **Fetching Product Reviews**:

    - Once the ASINs are extracted, the server sends requests to the **Real-time Amazon Data API** to fetch reviews for both products.
    - The `fetchProductReviews` function handles this by making a GET request to the API with the ASIN and other query parameters (e.g., sorting by most recent, verified purchases only).
    - If reviews are successfully fetched for both products, they are passed to the analysis function.
4. **Error Handling**:

    - If no reviews are fetched (empty response or error in the request), the server sends an error message back to the client.
    - In case of any errors in fetching reviews or analyzing the data, the server logs the error and responds with an appropriate error message.
5. **Analyzing Reviews**:

    - The reviews for both products are then sent to **Gemini API** (Google's generative language model) for analysis.

    - The `analyzeReviewsAndRecommend` function is used to format the reviews and send them to Gemini.
    - The request body for Gemini contains the product reviews and the user's expectation. It asks the model to:
        - Compare the reviews of both products.
        - Provide a recommendation based on the user's expectation (e.g., "which is better for durability?").
        - The response is a textual recommendation in plain text format.

Example payload sent to Gemini:

```
{
 "contents": [
  {
   "role": "user",
   "parts": [
    {
     "text": "Compare the reviews of the following two products and suggest the better one based on the
user's expectation of being 'durable': ..."
    }
   ]
  }
 ]
}
```

6. **Formatting the Response**:
    - The response from Gemini might contain HTML tags (like `<p>`, `<strong>`, etc.). The `markdown-it` library is used to render the HTML content and clean up the HTML entities (e.g., `&quot;` to `"`).
    - Additionally, the `<p>` tags are replaced with newline characters (`\n`) to preserve the paragraph structure in plain text.
    - Any remaining HTML tags are stripped out, and the final plain-text recommendation is returned to the client.
7. **Sending the Response**:

The server sends back the formatted recommendation in plain text in the response body:

```
{
 "recommendation": "I recommend Product 2 for durability..."
}
```

Client-Server Interaction Summary

1. **User Input**:

    - The user enters the URLs and their expectations in the frontend.
2. **Frontend Request**:

    - The frontend sends a **POST request** to `/api/compare` with the product URLs and the user's expectation.
3. **Server Processing**:

    - The server extracts the ASINs from the URLs, fetches the reviews from Amazon, and sends them to Gemini for analysis.
    - The server formats the recommendation and sends it back as a plain-text response.
4. **Frontend Display**:

    - The frontend displays the recommendation in the UI, either as a loading message or the final comparison result.

Flow Diagram:

1. **Frontend**: User enters URLs and expectations → **POST** request to `/api/compare`.
2. **Backend**:
    - Extract ASINs → Fetch reviews from Amazon → Send reviews to Gemini API → Format recommendation → **Return plain-text recommendation**.
3. **Frontend**: Display the recommendation to the user.

Handling Errors:

- **Invalid ASINs**: The server sends an error if either URL does not contain a valid ASIN.
- **No Reviews**: If no reviews are fetched from Amazon, the server responds with an error.
- **API Errors**: If any of the external APIs (Amazon, Gemini) fail, the server sends a fallback error message.
