

Unsplash Image Downloader

A Node.js application for bulk downloading images from the Unsplash Lite Dataset with intelligent organization and metadata preservation.

Table of Contents

1. [Project Overview](#)
2. [Features](#)
3. [Prerequisites](#)
4. [Installation](#)
5. [Project Structure](#)
6. [Usage](#)
7. [Code Documentation](#)
8. [Output Structure](#)
9. [Error Handling](#)
10. [Performance Considerations](#)
11. [Troubleshooting](#)









Project Overview

The Unsplash Image Downloader is a robust Node.js application designed to process CSV files from the [Unsplash Lite Dataset](#) and automatically download images with their associated metadata. The application organizes downloaded content in a structured directory hierarchy based on photographer and submission year.

Key Capabilities

- **Bulk Image Processing:** Downloads multiple images concurrently using batch processing
- **Intelligent Organization:** Creates directory structures based on photographer and year
- **Metadata Preservation:** Saves complete image metadata as JSON files
- **Descriptive Naming:** Generates meaningful filenames using AI descriptions and photographer information
- **Progress Tracking:** Provides real-time console feedback during processing
- **Error Resilience:** Continues processing even when individual downloads fail
- **Manifest Generation:** Creates a comprehensive index of all successfully downloaded images

Features

-  **Batch Processing:** Processes images in configurable batches to optimize performance
-  **Smart File Naming:** Uses AI descriptions and photographer names for meaningful filenames
-  **Hierarchical Organization:** Organizes files by photographer and year
-  **Metadata Preservation:** Saves complete CSV row data as JSON alongside images
-  **Progress Monitoring:** Real-time console output showing download progress
-  **Error Recovery:** Continues processing despite individual failures
-  **Manifest Creation:** Generates searchable index of downloaded content
-  **Cross-Platform:** Works on Windows, macOS, and Linux

Prerequisites

Before running this application, ensure you have:

- **Node.js** (version 10 or higher)
- **npm** (Node Package Manager)
- **photos.csv** file from the Unsplash Lite Dataset
- Sufficient disk space for image downloads
- Stable internet connection

Installation

Method 1: Quick Start (Windows)

1. Download all project files to a folder
2. Place your `photos.csv` file in the same folder
3. Double-click `start-project.bat`
4. The script will automatically install dependencies and start downloading

Method 2: Manual Setup

1. **Clone or download the project files**
2. **Navigate to the project directory**

```
bash  
cd your-project-folder
```

3. **Install dependencies**

```
bash  
npm install
```

4. Place your CSV file

- Ensure `photos.csv` is in the project root directory

5. Run the application

```
bash  
  
node download.js
```

Project Structure

```
project-root/  
├── download.js      # Main application script  
├── package.json     # Project configuration and dependencies  
├── package-lock.json # Exact dependency versions  
├── start-project.bat # Windows batch file for easy startup  
├── photos.csv       # Unsplash dataset (user-provided)  
└── downloads/       # Generated output directory  
    ├── manifest.json # Index of all downloaded images  
    └── [photographer]/  
        ├── [year]/  
        │   ├── [image].jpg  
        │   └── [image].json
```

Usage

Basic Usage

1. Ensure `photos.csv` is in the project root
2. Run the application:

```
bash  
  
node download.js
```

3. Monitor progress in the console
4. Find downloaded images in the `downloads/` directory

Expected CSV Format

The application expects a tab-separated CSV file with the following columns:

- `photo_id`: Unique identifier for the image
- `photo_image_url`: Direct URL to the image file
- `photographer_username`: Username of the photographer
- `photo_submitted_at`: Submission timestamp

- `ai_description`: AI-generated description of the image
- `photo_location_country`: Country where photo was taken

Code Documentation

Core Dependencies

javascript

```
const fs = require("fs");      // File system operations (sync)
const fsp = require("fs").promises; // File system operations (async)
const path = require("path");   // Cross-platform path utilities
const fetch = require("node-fetch"); // HTTP requests for downloading images
const csv = require("csv-parser"); // CSV parsing with streaming support
```

Main Functions

`downloadImage(url, filePath)`

Downloads an image from a URL and saves it to the specified file path.

javascript

```
async function downloadImage(url, filePath) {
  const res = await fetch(url);          // Fetch image from URL
  if (!res.ok) throw new Error(`Failed to fetch ${url}: ${res.statusText}`);
  const buffer = await res.buffer();     // Convert response to buffer
  await fsp.writeFile(filePath, buffer); // Write buffer to file
}
```

Parameters:

- `url` (string): HTTP URL of the image to download
- `filePath` (string): Local file system path where image should be saved

Error Handling: Throws an error if the HTTP request fails or file cannot be written.

`sanitizeForPath(text)`

Sanitizes text strings for use in file system paths by removing invalid characters.

javascript

```
function sanitizeForPath(text) {
  if (!text) return "_unknown"; // Handle null/undefined input
  return text.replace(/[\\?%*|"<>]/g, '_'); // Replace invalid chars with underscore
}
```

Parameters:

- `text` (string): Raw text that may contain invalid file system characters

Returns: Sanitized string safe for use in file paths

`generateDescriptiveFilename(row)`

Creates meaningful filenames using image metadata.

javascript

```
function generateDescriptiveFilename(row) {  
  const description = row.ai_description || 'untitled';    // Get description or default  
  const photographer = row.photographer_username || 'unknown'; // Get photographer or default  
  const descriptionSlug = description  
    .toLowerCase()           // Convert to lowercase  
    .replace(/[^a-z0-9\s]/g, '') // Remove special characters  
    .replace(/\s+/g, '_')      // Replace spaces with underscores  
    .substring(0, 50);        // Limit length to 50 characters  
  return `${descriptionSlug}_by_${photographer}_${row.photo_id}`;  
}
```

Parameters:

- `row` (object): CSV row data containing image metadata

Returns: Descriptive filename string in format: `description_by_photographer_id`

`processBatch(rows, start, batchSize)`

Processes a batch of images concurrently with error handling.

javascript

```

async function processBatch(rows, start, batchSize) {
  const batch = rows.slice(start, start + batchSize); // Extract batch subset
  return Promise.allSettled( // Handle all promises regardless of failures
    batch.map(async (row) => {
      // Validation
      const id = row["photo_id"];
      const url = row["photo_image_url"];
      if (!id || !url || !url.startsWith("http")) {
        return Promise.reject(new Error(`Invalid data for row: ${JSON.stringify(row)}`));
      }

      try {
        // Directory structure creation
        const photographer = sanitizeForPath(row["photographer_username"]);
        const year = row["photo_submitted_at"]
          ? new Date(row["photo_submitted_at"]).getFullYear()
          : "_unknown_date";
        const targetDir = path.join("downloads", photographer, String(year));
        await fsp.mkdir(targetDir, { recursive: true });

        // File path generation
        const baseFilename = generateDescriptiveFilename(row);
        const imagePath = path.join(targetDir, `${baseFilename}.jpg`);
        const jsonPath = path.join(targetDir, `${baseFilename}.json`);

        // Download and save operations
        await downloadImage(url, imagePath); // Download image
        await fsp.writeFile(jsonPath, JSON.stringify(row, null, 2)); // Save metadata

        console.log(`✅ Saved ${baseFilename}.jpg to ${targetDir}`);

        // Return manifest entry
        return {
          id: row.photo_id,
          description: row.ai_description,
          photographer: row.photographer_username,
          country: row.photo_location_country,
          tags: (row.ai_description || "").split(' '),
          path: imagePath
        };
      } catch (err) {
        console.error(`❌ Error with ${id}:`, err.message);
        return Promise.reject(err);
      }
    })
  );
}

```

```
);  
}
```

Parameters:

- `rows` (array): Complete array of CSV rows
- `start` (number): Starting index for batch processing
- `batchSize` (number): Number of rows to process in this batch

Returns: Promise that resolves to an array of `PromiseSettledResult` objects

`main()`

The primary orchestration function that coordinates the entire download process.

```
javascript
```

```

async function main() {
  const results = [];           // Store all CSV rows
  const manifestData = [];      // Store successful download metadata

  console.log(" 📁 Reading photos.csv...");

  // Stream CSV parsing
  fs.createReadStream("photos.csv")
    .pipe(csv({
      separator: "\\t",          // Tab-separated values
      mapHeaders: ({ header }) => header.trim() // Clean whitespace from headers
    )))
    .on("data", (row) => results.push(row)) // Collect each row
    .on("end", async () => {
      console.log(` 🖼️ Loaded ${results.length} rows`);

      // Batch processing loop
      const batchSize = 20;
      for (let i = 0; i < results.length; i += batchSize) {
        console.log(` ➡️ Processing rows ${i + 1} - ${Math.min(i + batchSize, results.length)}`);
        const batchResults = await processBatch(results, i, batchSize);

        // Collect successful results for manifest
        batchResults.forEach(result => {
          if (result.status === 'fulfilled' && result.value) {
            manifestData.push(result.value);
          }
        });
      }

      // Generate manifest file
      try {
        const manifestPath = path.join("downloads", "manifest.json");
        await fsp.writeFile(manifestPath, JSON.stringify(manifestData, null, 2));
        console.log(` 📄 Manifest file created at ${manifestPath}`);
      } catch (err) {
        console.error(" ❌ Error writing manifest file:", err);
      }

      console.log(" 🏁 All downloads finished!");
    });
}

```

Application Entry Point


```
// Initialize downloads directory and start processing
fsp.mkdir("downloads", { recursive: true }).then(main);
```

This line ensures the main downloads directory exists before beginning the download process.

Output Structure

The application creates a well-organized directory structure:

```
downloads/
├── manifest.json          # Complete index of downloaded images
├── photographer1/
│   ├── 2023/
│   │   ├── sunset_beach_by_photographer1_abc123.jpg
│   │   ├── sunset_beach_by_photographer1_abc123.json
│   │   ├── mountain_view_by_photographer1_def456.jpg
│   │   └── mountain_view_by_photographer1_def456.json
│   └── 2024/
│       ├── city_lights_by_photographer1_ghi789.jpg
│       └── city_lights_by_photographer1_ghi789.json
└── photographer2/
    └── 2023/
        ├── forest_path_by_photographer2_jkl012.jpg
        └── forest_path_by_photographer2_jkl012.json
```

Manifest File Structure

The `manifest.json` file contains a searchable index of all successfully downloaded images:

```
json
[
  {
    "id": "abc123",
    "description": "Beautiful sunset over calm beach waters",
    "photographer": "photographer1",
    "country": "Maldives",
    "tags": ["Beautiful", "sunset", "over", "calm", "beach", "waters"],
    "path": "downloads/photographer1/2023/sunset_beach_by_photographer1_abc123.jpg"
  }
]
```

Error Handling

The application implements comprehensive error handling:

Network Errors

- Failed HTTP requests are logged and skipped
- Invalid URLs are detected and reported
- Network timeouts are handled gracefully

File System Errors

- Missing directories are created automatically
- File write failures are logged and reported
- Path validation prevents invalid file names

Data Validation

- Missing required fields are detected
- Invalid CSV rows are skipped with logging
- Malformed URLs are identified and handled

Recovery Mechanisms

- Individual failures don't stop batch processing
- Progress continues despite network interruptions
- Partial downloads are properly cleaned up

Performance Considerations

Batch Processing

- Default batch size: 20 concurrent downloads
- Prevents overwhelming the network or file system
- Adjustable by modifying the `batchSize` variable

Memory Management

- Streaming CSV parser prevents loading entire file into memory
- Images are processed as buffers and immediately written to disk
- Metadata is processed incrementally

Network Optimization

- Concurrent downloads within batches
- Proper error handling prevents stuck connections
- Uses efficient node-fetch library for HTTP requests

File System Optimization

- Recursive directory creation reduces system calls
- Path sanitization prevents file system errors
- JSON metadata is formatted for readability

Troubleshooting

Common Issues

"Module not found" errors

Solution: Run `npm install` to install dependencies

"photos.csv not found"

Solution: Ensure the CSV file is in the project root directory with exact filename

Permission denied errors

Solution: Ensure write permissions for the project directory

Network timeout errors





Solution: Check internet connection and try reducing batch size

Disk space errors

Solution: Ensure sufficient free disk space (images can be large)

Debug Information

The application provides detailed console output:

-  Success indicators for completed downloads
-  Error messages with specific failure reasons
-  Progress indicators showing batch completion
-  Final completion message

Performance Tuning

To adjust performance for your system:

1. **Modify batch size** in `main()`:

```
javascript
```

```
const batchSize = 10; // Reduce for slower connections
```

2. Add delays between batches:

javascript

```
await new Promise(resolve => setTimeout(resolve, 1000)); // 1 second delay
```

3. Filter CSV data before processing:

javascript

```
const filteredResults = results.filter(row => row.photographer_username === 'specific_photographer');
```

Support

For additional support or questions about the Unsplash Lite Dataset, refer to:

- [Unsplash Dataset GitHub Repository](#)
- [Node.js Documentation](#)
- [npm Documentation](#)

License

This project is designed to work with the Unsplash Lite Dataset. Please ensure compliance with Unsplash's terms of service and licensing requirements when using downloaded images.