Introduction

Welcome to the documentation for Unsplasharp, a modern, asynchronous, and feature-rich .NET library for interacting with the Unsplash API.

What is Unsplasharp?

Unsplasharp is a .NET Standard 2.0 library that provides a convenient and easy-to-use wrapper around the Unsplash API. It allows you to easily integrate Unsplash's vast library of high-quality photos into your .NET applications.

Features

- **Modern and Asynchronous**: Built with `async/await` for non-blocking I/O.
- **Comprehensive Error Handling**: Provides specific exception types for different error scenarios.
- **IHttpClientFactory Integration**: Supports `IHttpClientFactory` for efficient management of `HttpClient` instances.
- **Structured Logging**: Integrates with `Microsoft.Extensions.Logging` for detailed insights.
- **System.Text.Json**: Uses the high-performance `System.Text.Json` for JSON serialization.
- **Full API Coverage**: Provides access to all of the Unsplash API endpoints.

Getting Help

If you need help or have any questions, please feel free to open an issue on GitHub.

Getting Started with Unsplasharp

This comprehensive guide will walk you through setting up and using Unsplasharp in your .NET application, from basic installation to advanced usage patterns.

Table of Contents

Prerequisites

Before you begin, ensure you have:

- **.NET 6.0 or later** (or .NET Framework 4.6.1+, .NET Standard 2.0+)
- **An Unsplash API key** - See Getting Your API Key below
- **Basic knowledge of C# and async/await**

Installation

Package Manager Console

```
Install-Package Unsplasharp
```

.NET CLI

```
dotnet add package Unsplasharp
```

PackageReference (in .csproj)

```
<PackageReference Include="Unsplasharp" Version="*" />
```

Getting Your API Key

1. **Create an Unsplash Account**: Visit unsplash.com and sign up

2. **Join as Developer**: Go to Unsplash for Developers
3. **Create New Application**:
   - Visit your applications dashboard
   - Click "New Application"
   - Accept the API terms
   - Fill in your application details
4. **Get Your Access Key**: Copy the "Access Key" - this is your `ApplicationId`

> **Important**: Keep your API key secure and never expose it in client-side code or public repositories.

Basic Setup

Simple Console Application

```csharp
using Unsplasharp;
using System;
using System.Threading.Tasks;

class Program
{
    static async Task Main(string[] args)
    {
        // Replace with your actual Application ID
        var client = new UnsplasharpClient("YOUR_APPLICATION_ID");

        try
        {
            var photo = await client.GetRandomPhoto();
            Console.WriteLine($"Random photo by {photo?.User?.Name}");
            Console.WriteLine($"URL: {photo?.Urls?.Regular}");
        }
        catch (Exception ex)
        {
            Console.WriteLine($"Error: {ex.Message}");
        }
    }
}
```

ASP.NET Core Setup

Program.cs:

```csharp
using Unsplasharp.Extensions;

var builder = WebApplication.CreateBuilder(args);

// Add Unsplasharp to dependency injection
builder.Services.AddUnsplasharp("YOUR_APPLICATION_ID");

// Or configure from appsettings.json
builder.Services.AddUnsplasharp(options =>
{
    options.ApplicationId = builder.Configuration["Unsplash:ApplicationId"];
    options.ConfigureHttpClient = client =>
    {
        client.Timeout = TimeSpan.FromSeconds(30);
    };
});

var app = builder.Build();
```

Controller:

```csharp
[ApiController]
[Route("api/[controller]")]
public class PhotosController : ControllerBase
{
    private readonly UnsplasharpClient _unsplashClient;

    public PhotosController(UnsplasharpClient unsplashClient)
    {
        _unsplashClient = unsplashClient;
    }

    [HttpGet("random")]
    public async Task<IActionResult> GetRandomPhoto()
    {
        try
        {
            var photo = await _unsplashClient.GetRandomPhoto();
            return Ok(new {
                id = photo?.Id,
                description = photo?.Description,
                author = photo?.User?.Name,
                url = photo?.Urls?.Regular
            });
        }
        catch (Exception ex)
        {
            return StatusCode(500, new { error = ex.Message });
        }
    }
}
```

Your First Request

Let's start with a simple example that demonstrates the basic workflow:

```
using Unsplasharp;
using System;
using System.Threading.Tasks;

public class UnsplashExample
{
    private readonly UnsplasharpClient _client;

    public UnsplashExample()
    {
        _client = new UnsplasharpClient("YOUR_APPLICATION_ID");
    }

    public async Task GetRandomPhotoExample()
    {
        try
        {
            // Get a random photo
            var photo = await _client.GetRandomPhoto();

            if (photo != null)
            {
                Console.WriteLine("=== Random Photo ===");
                Console.WriteLine($"ID: {photo.Id}");
                Console.WriteLine($"Description: {photo.Description ??
"No description"}");
                Console.WriteLine($"Author: {photo.User.Name}
(@{photo.User.Username})");
                Console.WriteLine($"Dimensions: {photo.Width}x{photo.Height}");
                Console.WriteLine($"Likes: {photo.Likes:N0}");
                Console.WriteLine($"Downloads: {photo.Downloads:N0}");
                Console.WriteLine($"Color: {photo.Color}");
                Console.WriteLine($"Regular URL: {photo.Urls.Regular}");
                Console.WriteLine($"Small URL: {photo.Urls.Small}");
                // Check if location data is available
                if (!string.IsNullOrEmpty(photo.Location.Name))
                {
                    Console.WriteLine($"Location: {photo.Location.Name}");
                }

                // Check EXIF data
                if (!string.IsNullOrEmpty(photo.Exif.Make))
                {
                    Console.WriteLine($"Camera: {photo.Exif.Make}
{photo.Exif.Model}");
                }
            }
            else
            {
                Console.WriteLine("No photo received");
            }
        }
        catch (Exception ex)
        {
            Console.WriteLine($"Error getting random photo: {ex.Message}");
        }
    }
}
```

Common Use Cases

1. Search for Photos

```
public async Task SearchPhotosExample()
{
    try
    {
        // Basic search
        var photos = await _client.SearchPhotos("mountain landscape", page: 1,
perPage: 10);

        Console.WriteLine($"Found {photos.Count} photos");
        Console.WriteLine($"Total results:
{_client.LastPhotosSearchTotalResults:N0}");
        Console.WriteLine($"Total pages: {_client.LastPhotosSearchTotalPages}");

        foreach (var photo in photos)
        {
            Console.WriteLine($"- {photo.Id}: {photo.Description ?? "Untitled"}");
            Console.WriteLine($"  By: {photo.User.Name} | Likes: {photo.Likes}");
        }
    }
    catch (Exception ex)
    {
        Console.WriteLine($"Search error: {ex.Message}");
    }
}
```

2. Get Photos from a Collection

```
public async Task GetCollectionPhotosExample()
{
    try
    {
        // First, get collection details
        var collection = await _client.GetCollection("499830"); // Nature collection

        if (collection != null)
        {
            Console.WriteLine($"Collection: {collection.Title}");
            Console.WriteLine($"Description: {collection.Description}");
            Console.WriteLine($"Total photos: {collection.TotalPhotos}");
            Console.WriteLine($"Created by: {collection.User.Name}");

            // Get photos from the collection
            var photos = await _client.GetCollectionPhotos(collection.Id, page: 1,
perPage: 5);

            Console.WriteLine("\nFirst 5 photos:");
            foreach (var photo in photos)
            {
                Console.WriteLine($"- {photo.Description ?? "Untitled"}
by {photo.User.Name}");
            }
        }
    }
    catch (Exception ex)
    {
        Console.WriteLine($"Collection error: {ex.Message}");
    }
}
```

3. Get User Information and Photos

```
public async Task GetUserPhotosExample()
{
    try
    {
        var username = "chrisjoelcampbell";

        // Get user profile
        var user = await _client.GetUser(username);

        if (user != null)
        {
            Console.WriteLine($"User: {user.Name} (@{user.Username})");
            Console.WriteLine($"Bio: {user.Bio}");
            Console.WriteLine($"Location: {user.Location}");
            Console.WriteLine($"Total photos: {user.TotalPhotos:N0}");
            Console.WriteLine($"Total likes: {user.TotalLikes:N0}");
            Console.WriteLine($"Portfolio: {user.PortfolioUrl}");

            // Get user's photos
            var userPhotos = await _client.GetUserPhotos(username, page: 1,
perPage: 5);

            Console.WriteLine($"\nRecent photos by {user.Name}:");
            foreach (var photo in userPhotos)
            {
                Console.WriteLine($"- {photo.Description ?? "Untitled"}");
                Console.WriteLine($"  {photo.Width}x{photo.Height} |
{photo.Likes} likes");
            }
        }
    }
    catch (Exception ex)
    {
        Console.WriteLine($"User error: {ex.Message}");
    }
}
```

4. Download a Photo

```
public async Task DownloadPhotoExample()
{
    try
    {
        var photoId = "qcs09SwNPHY"; // Example photo ID

        // Get photo details
        var photo = await _client.GetPhoto(photoId);

        if (photo != null)
        {
            Console.WriteLine($"Downloading: {photo.Description ?? "Untitled"}");
            Console.WriteLine($"By: {photo.User.Name}");

            // Use HttpClient to download the image
            using var httpClient = new HttpClient();

            // Download different sizes
            var sizes = new Dictionary<string, string>
            {
                ["thumbnail"] = photo.Urls.Thumbnail,
                ["small"] = photo.Urls.Small,
                ["regular"] = photo.Urls.Regular
            };

            foreach (var (sizeName, url) in sizes)
            {
                try
                {
                    var imageBytes = await httpClient.GetByteArrayAsync(url);
                    var fileName = $"{photo.Id}_{sizeName}.jpg";
                    await File.WriteAllBytesAsync(fileName, imageBytes);
                    Console.WriteLine($"Downloaded {sizeName}: {fileName}
({imageBytes.Length:N0} bytes)");
                }
                catch (Exception downloadEx)
                {
                    Console.WriteLine($"Failed to download
{sizeName}: {downloadEx.Message}");
                }
            }
        }
    }
    catch (Exception ex)
    {
        Console.WriteLine($"Download error: {ex.Message}");
    }
}
```

5. Pagination Example

```
public async Task PaginationExample()
{
    try
    {
        var query = "sunset";
        var allPhotos = new List<Photo>();
        var maxPages = 3; // Limit for demo

        Console.WriteLine($"Searching for '{query}' across {maxPages} pages...");

        for (int page = 1; page <= maxPages; page++)
        {
            var photos = await _client.SearchPhotos(query, page: page, perPage: 20);

            if (photos.Count == 0)
            {
                Console.WriteLine($"No more results at page {page}");
                break;
            }

            allPhotos.AddRange(photos);
            Console.WriteLine($"Page {page}: {photos.Count} photos");

            // Be respectful of rate limits
            await Task.Delay(100);
        }

        Console.WriteLine($"Total photos collected: {allPhotos.Count}");
        Console.WriteLine($"Total available:
{_client.LastPhotosSearchTotalResults:N0}");
    }
    catch (Exception ex)
    {
        Console.WriteLine($"Pagination error: {ex.Message}");
    }
}
```

6. Advanced Search with Filters

```
public async Task AdvancedSearchExample()
{
    try
    {
        // Search with multiple filters
        var photos = await _client.SearchPhotos(
            query: "ocean waves",
            page: 1,
            perPage: 15,
            orderBy: OrderBy.Popular,
            color: "blue",
            orientation: Orientation.Landscape
        );

        Console.WriteLine("Advanced search results:");
        Console.WriteLine($"Query: ocean waves");
        Console.WriteLine($"Filters: blue color, landscape orientation,
popular order");
        Console.WriteLine($"Results: {photos.Count} photos");

        foreach (var photo in photos.Take(5))
        {
            Console.WriteLine($"- {photo.Description ?? "Untitled"}");
            Console.WriteLine($"  {photo.Width}x{photo.Height} |
Color: {photo.Color}");
            Console.WriteLine($"  Likes: {photo.Likes} |
Downloads: {photo.Downloads}");
        }
    }
    catch (Exception ex)
    {
        Console.WriteLine($"Advanced search error: {ex.Message}");
    }
}
```

Error Handling

Unsplasharp provides comprehensive error handling with specific exception types. Here's how to handle different error scenarios:

Basic Error Handling

```
public async Task BasicErrorHandlingExample()
{
    try
    {
        var photo = await _client.GetPhoto("invalid-photo-id");
        // Handle success
    }
    catch (Exception ex)
    {
        Console.WriteLine($"Error: {ex.Message}");
        // Basic error handling - not recommended for production
    }
}
```

Advanced Error Handling (Recommended)

```
public async Task AdvancedErrorHandlingExample()
{
    try
    {
        // Use the exception-throwing version for better error handling
        var photo = await _client.GetPhotoAsync("some-photo-id");

        Console.WriteLine($"Successfully retrieved: {photo.Description}");
    }
    catch (UnsplasharpNotFoundException ex)
    {
        Console.WriteLine("Photo not found - it may have been deleted or
made private");
        // Handle gracefully - maybe show a placeholder or try alternative
    }
    catch (UnsplasharpRateLimitException ex)
    {
        Console.WriteLine($"Rate limit exceeded:
{ex.RateLimitRemaining}/{ex.RateLimit}");
        Console.WriteLine($"Reset time: {ex.RateLimitReset}");

        // Wait until reset time
        if (ex.RateLimitReset.HasValue)
        {
            var waitTime = ex.RateLimitReset.Value - DateTimeOffset.UtcNow;
            if (waitTime > TimeSpan.Zero)
            {
                Console.WriteLine($"Waiting {waitTime.TotalMinutes:F1} minutes...");
                await Task.Delay(waitTime);
                // Retry the request
            }
        }
    }
    catch (UnsplasharpAuthenticationException ex)
    {
        Console.WriteLine("Authentication failed - check your API key");
        // Log the error and potentially refresh credentials
    }
    catch (UnsplasharpNetworkException ex)
    {
        Console.WriteLine($"Network error: {ex.Message}");
        if (ex.IsRetryable)
        {
            Console.WriteLine("This error is retryable - implementing 
exponential backoff");
            // Implement retry logic with exponential backoff
        }
    }
    catch (UnsplasharpTimeoutException ex)
    {
        Console.WriteLine($"Request timed out after {ex.Timeout}");
        // Consider increasing timeout or checking network conditions
    }
    catch (UnsplasharpException ex)
    {
        Console.WriteLine($"Unsplash API error: {ex.Message}");

        // Access rich error context
        if (ex.Context != null)
        {
            Console.WriteLine($"Correlation ID: {ex.Context.CorrelationId}");
            Console.WriteLine($"Timestamp: {ex.Context.Timestamp}");
            Console.WriteLine($"Request URL: {ex.RequestUrl}");

            // Check rate limit info
            if (ex.Context.RateLimitInfo != null)
            {
                var rateLimit = ex.Context.RateLimitInfo;
                Console.WriteLine($"Rate limit: 
{rateLimit.Remaining}/{rateLimit.Limit}");
            }
        }
    }
}
```

Retry Logic with Exponential Backoff

```
public async Task<Photo?> GetPhotoWithRetry(string photoId, int maxRetries = 3)
{
    for (int attempt = 1; attempt <= maxRetries; attempt++)
    {
        try
        {
            return await _client.GetPhotoAsync(photoId);
        }
        catch (UnsplasharpNetworkException ex) when (ex.IsRetryable && attempt
< maxRetries)
        {
            var delay = TimeSpan.FromSeconds(Math.Pow(2, attempt)); //
Exponential backoff
            Console.WriteLine($"Attempt {attempt} failed, retrying
in {delay.TotalSeconds}s...");
            await Task.Delay(delay);
        }
        catch (UnsplasharpRateLimitException ex) when (attempt < maxRetries)
        {
            var delay = ex.TimeUntilReset ?? TimeSpan.FromMinutes(1);
            Console.WriteLine($"Rate limited, waiting {delay.TotalMinutes:F1}
minutes...");
            await Task.Delay(delay);
        }
        catch (UnsplasharpNotFoundException)
        {
            // Don't retry for not found errors
            Console.WriteLine($"Photo {photoId} not found");
            return null;
        }
    }

    throw new InvalidOperationException($"Failed to get photo after
{maxRetries} attempts");
}
```

Best Practices

1. Use Dependency Injection

```
// In Program.cs or Startup.cs
services.AddUnsplasharp(options =>
{
    options.ApplicationId = configuration["Unsplash:ApplicationId"];
    options.ConfigureHttpClient = client =>
    {
        client.Timeout = TimeSpan.FromSeconds(30);
    };
});
```

```
// In your service
public class PhotoService
{
    private readonly UnsplasharpClient _client;
    private readonly ILogger<PhotoService> _logger;

    public PhotoService(UnsplasharpClient client, ILogger<PhotoService> logger)
    {
        _client = client;
        _logger = logger;
    }

    public async Task<List<Photo>> GetFeaturedPhotos(int count = 10)
    {
        try
        {
            return await _client.ListPhotos(page: 1, perPage: count,
orderBy: OrderBy.Popular);
        }
        catch (UnsplasharpException ex)
        {
            _logger.LogError(ex, "Failed to get featured photos");
            return new List<Photo>();
        }
    }
}
```

2. Implement Caching

```
public class CachedPhotoService
{
    private readonly UnsplasharpClient _client;
    private readonly IMemoryCache _cache;
    private readonly ILogger<CachedPhotoService> _logger;

    public CachedPhotoService(UnsplasharpClient client, IMemoryCache cache,
ILogger<CachedPhotoService> logger)
    {
        _client = client;
        _cache = cache;
        _logger = logger;
    }

    public async Task<Photo?> GetPhotoAsync(string photoId)
```

```csharp
    {
        var cacheKey = $"photo:{photoId}";

        if (_cache.TryGetValue(cacheKey, out Photo cachedPhoto))
        {
            _logger.LogDebug("Photo {PhotoId} found in cache", photoId);
            return cachedPhoto;
        }

        try
        {
            var photo = await _client.GetPhotoAsync(photoId);

            // Cache for 1 hour
            _cache.Set(cacheKey, photo, TimeSpan.FromHours(1));

            _logger.LogDebug("Photo {PhotoId} cached", photoId);
            return photo;
        }
        catch (UnsplasharpNotFoundException)
        {
            // Cache negative results for shorter time
            _cache.Set(cacheKey, (Photo?)null, TimeSpan.FromMinutes(5));
            return null;
        }
    }
}
```

3. Monitor Rate Limits

```csharp
public class RateLimitAwareService
{
    private readonly UnsplasharpClient _client;
    private readonly ILogger<RateLimitAwareService> _logger;

    public RateLimitAwareService(UnsplasharpClient client,
ILogger<RateLimitAwareService> logger)
    {
        _client = client;
        _logger = logger;
    }

    public async Task<List<Photo>> SearchPhotosWithRateLimit(string query, int
maxResults = 100)
    {
        var allPhotos = new List<Photo>();
        var perPage = 30;
        var maxPages = (maxResults + perPage - 1) / perPage;

        for (int page = 1; page <= maxPages; page++)
        {
            // Check rate limit before making request
            if (_client.RateLimitRemaining < 10)
            {
                _logger.LogWarning("Rate limit running low: {Remaining}/{Max}",
                    _client.RateLimitRemaining, _client.MaxRateLimit);

                // Pause to avoid hitting the limit
                await Task.Delay(TimeSpan.FromSeconds(30));
            }

            try
            {
                var photos = await _client.SearchPhotosAsync(query, page: page,
perPage: perPage);
                allPhotos.AddRange(photos);

                _logger.LogDebug("Page {Page}: {Count} photos, Rate
limit: {Remaining}/{Max}",
                    page, photos.Count, _client.RateLimitRemaining,
_client.MaxRateLimit);

                if (photos.Count < perPage)
                {
                    // No more results
                    break;
                }
            }
            catch (UnsplasharpRateLimitException ex)
            {
                _logger.LogWarning("Rate limit exceeded, stopping search");
                break;
            }
        }

        return allPhotos.Take(maxResults).ToList();
    }
}
```

4. Use Cancellation Tokens

```
public class ResponsivePhotoService
{
    private readonly UnsplasharpClient _client;

    public ResponsivePhotoService(UnsplasharpClient client)
    {
        _client = client;
    }

    public async Task<List<Photo>> SearchWithTimeout(string query, TimeSpan timeout)
    {
        using var cts = new CancellationTokenSource(timeout);

        try
        {
            return await _client.SearchPhotosAsync(query,
cancellationToken: cts.Token);
        }
        catch (OperationCanceledException)
        {
            Console.WriteLine($"Search for '{query}' timed out
after {timeout.TotalSeconds}s");
            return new List<Photo>();
        }
    }

    public async Task<Photo?> GetPhotoWithCancellation(string photoId,
CancellationToken cancellationToken)
    {
        try
        {
            return await _client.GetPhotoAsync(photoId, cancellationToken);
        }
        catch (OperationCanceledException)
        {
            Console.WriteLine("Operation was cancelled");
            return null;
        }
    }
}
```

5. Configuration Management

appsettings.json:

```json
{
  "Unsplash": {
    "ApplicationId": "YOUR_APPLICATION_ID",
    "Secret": "YOUR_SECRET",
    "DefaultTimeout": "00:00:30",
    "MaxRetries": 3,
    "CacheEnabled": true,
    "CacheDuration": "01:00:00"
  },
  "Logging": {
    "LogLevel": {
      "Unsplasharp": "Information"
    }
  }
}
```

Configuration class:

```csharp
public class UnsplashConfiguration
{
    public string ApplicationId { get; set; } = string.Empty;
    public string? Secret { get; set; }
    public TimeSpan DefaultTimeout { get; set; } = TimeSpan.FromSeconds(30);
    public int MaxRetries { get; set; } = 3;
    public bool CacheEnabled { get; set; } = true;
    public TimeSpan CacheDuration { get; set; } = TimeSpan.FromHours(1);
}
```

Service registration:

```csharp
// In Program.cs
var unsplashConfig =
builder.Configuration.GetSection("Unsplash").Get<UnsplashConfiguration>();

builder.Services.AddSingleton(unsplashConfig);
builder.Services.AddMemoryCache();

builder.Services.AddUnsplasharp(options =>
{
    options.ApplicationId = unsplashConfig.ApplicationId;
    options.Secret = unsplashConfig.Secret;
    options.ConfigureHttpClient = client =>
    {
        client.Timeout = unsplashConfig.DefaultTimeout;
    };
});
```

Next Steps

Congratulations! You now have a solid foundation for using Unsplasharp. Here are some recommended next steps:

1. Explore Advanced Features

- API Reference Guide - Complete method documentation
- Error Handling Guide - Comprehensive error handling strategies
- IHttpClientFactory Integration - Production-ready HTTP client management
- Logging Guide - Structured logging and monitoring

2. Common Integration Patterns

- **Image Gallery Applications** - Build photo galleries with search and filtering
- **Background Image Services** - Provide dynamic backgrounds for applications
- **Content Management Systems** - Integrate stock photos into CMS platforms
- **Social Media Tools** - Create tools for social media content creation

3. Performance Optimization

- **Implement caching** to reduce API calls and improve response times
- **Use pagination** effectively for large result sets
- **Monitor rate limits** to avoid service interruptions
- **Implement retry logic** for resilient applications

4. Production Considerations

- **Security**: Never expose API keys in client-side code
- **Monitoring**: Set up logging and monitoring for API usage
- **Error Handling**: Implement comprehensive error handling strategies
- **Testing**: Write unit tests for your Unsplash integration

5. Sample Projects

Consider building these sample projects to practice:

```
// 1. Photo Search CLI Tool
public class PhotoSearchTool
{
    public async Task RunAsync(string[] args)
    {
        var query = args.Length > 0 ? args[0] : "nature";
        var client = new UnsplasharpClient("YOUR_APP_ID");

        var photos = await client.SearchPhotos(query, perPage: 10);

        foreach (var photo in photos)
        {
            Console.WriteLine($"{photo.User.Name}: {photo.Description}");
            Console.WriteLine($"  {photo.Urls.Small}");
        }
    }
}

// 2. Random Wallpaper Downloader
public class WallpaperDownloader
{
    public async Task DownloadRandomWallpaper(string category = "landscape")
    {
        var client = new UnsplasharpClient("YOUR_APP_ID");
        var photo = await client.GetRandomPhoto(query: category,
orientation: Orientation.Landscape);

        if (photo != null)
        {
            using var httpClient = new HttpClient();
            var imageBytes = await httpClient.GetByteArrayAsync(photo.Urls.Full);

            var fileName = $"wallpaper_{photo.Id}.jpg";
            await File.WriteAllBytesAsync(fileName, imageBytes);

            Console.WriteLine($"Downloaded wallpaper: {fileName}");
            Console.WriteLine($"By: {photo.User.Name}");
        }
    }
}

// 3. Photo Collection Manager
public class CollectionManager
{
    private readonly UnsplasharpClient _client;

    public CollectionManager(string appId)
    {
        _client = new UnsplasharpClient(appId);
    }
    public async Task ExploreCollection(string collectionId)
    {
        var collection = await _client.GetCollection(collectionId);
        var photos = await _client.GetCollectionPhotos(collectionId, perPage: 20);

        Console.WriteLine($"Collection: {collection?.Title}");
        Console.WriteLine($"Photos: {photos.Count}");

        foreach (var photo in photos)
        {
            Console.WriteLine($"  - {photo.Description ?? "Untitled"}
by {photo.User.Name}");
        }
    }
}
```

6. Community and Support

- **GitHub Repository**: github.com/rootasjey/unsplasharp
- **Issues and Bug Reports**: Use GitHub Issues for bug reports and feature requests
- **Unsplash API Documentation**: unsplash.com/documentation
- **Unsplash Developer Guidelines**: Follow Unsplash's API guidelines and terms of service

7. Contributing

If you'd like to contribute to Unsplasharp:

1. Fork the repository
2. Create a feature branch
3. Make your changes
4. Add tests for new functionality
5. Submit a pull request

You're now ready to build amazing applications with Unsplasharp! Remember to always respect Unsplash's API rate limits and terms of service, and consider implementing proper error handling and caching for production applications.

Obtaining an API Key

To use the Unsplash API, you need to register as a developer and create an application to get an API key.

Steps

1. **Create an Unsplash Account**: If you don't have an Unsplash account, you'll need to create one at unsplash.com.
2. **Join as a Developer**: Go to the Unsplash for Developers page and click "Join as a Developer".
3. **Register a New Application**:
    - Go to your applications dashboard.
    - Click the "New Application" button.
    - Read and accept the API terms.
    - Fill out the application details:
        - **Application Name**: A descriptive name for your application.
        - **Description**: A brief description of what your application does.
    - Click "Create Application".
4. **Get Your API Keys**:
    - After creating the application, you'll be redirected to the application's page.
    - You will find your **Access Key** and **Secret Key**.
    - The **Access Key** is what you'll use as your `ApplicationId` in Unsplasharp.

API Usage Limits

The Unsplash API has the following rate limits for new applications:

- **Demo**: 50 requests per hour.
- **Production**: 5000 requests per hour (requires approval from Unsplash).

You can see your current rate limit status in the response headers of each API call. Unsplasharp provides this information in the `RateLimitInfo` property of the `UnsplasharpException` and `ErrorContext` classes.

Downloading a Photo

This section explains how to download an Unsplash photo using the Unsplasharp DSK (Developer SDK) library.

Prerequisites

Before you begin, ensure you have:

- An Unsplash API key. If you don't have one, refer to the Obtaining an API Key guide.
- The Unsplasharp library installed in your project.

Example: Downloading a Photo

To download a photo, you typically need the photo's ID. You can then use the `UnsplasharpClient` to retrieve the photo details and its download link.

```
using Unsplasharp;
using System;
using System.Net.Http;
using System.Threading.Tasks;
using System.IO;

public class PhotoDownloader
{
    public static async Task Main(string[] args)
    {
        // Replace with your actual Unsplash API key
        string apiKey = "YOUR_UNSPLASH_API_KEY";
        // Replace with the ID of the photo you want to download
        string photoId = "PHOTO_ID_TO_DOWNLOAD";
        // Replace with the desired path to save the downloaded photo
        string downloadPath = "path/to/your/downloaded_photo.jpg";

        var client = new UnsplasharpClient(apiKey);

        try
        {
            // Get photo details
            var photo = await client.GetPhoto(photoId);

            if (photo != null &&
!string.IsNullOrEmpty(photo.Links.DownloadLocation))
            {
                // Unsplash requires you to hit the download location endpoint to
register a download
                // before you can download the actual image.
                var downloadLink = await client.GetPhotoDownloadLink(photoId);

                if (!string.IsNullOrEmpty(downloadLink))
                {
                    using (HttpClient httpClient = new HttpClient())
                    {
                        // Download the image
                        byte[] imageBytes = await
httpClient.GetByteArrayAsync(downloadLink);
                        await File.WriteAllBytesAsync(downloadPath, imageBytes);
                        Console.WriteLine($"Photo downloaded successfully
to: {downloadPath}");
                    }
                }
                else
                {
                    Console.WriteLine("Could not retrieve download link for
the photo.");
                }
            }
            else
            {
                Console.WriteLine($"Photo with ID '{photoId}' not found or download
link is missing.");
            }
        }
        catch (Exception ex)
        {
            Console.WriteLine($"An error occurred: {ex.Message}");
        }
    }
}
```

Remember to replace `"YOUR_UNSPLASH_API_KEY"`, `"PHOTO_ID_TO_DOWNLOAD"`, and `"path/to/your/downloaded_photo.jpg"` with your actual values.

Important Considerations

- **API Key:** Always keep your API key secure and do not expose it in client-side code.
- **Download Tracking:** Unsplash requires you to trigger the `download_location` endpoint before you can download the actual image. This is important for their download statistics. The `GetPhotoDownloadLink` method handles this for you.

- **Error Handling:** Implement robust error handling to gracefully manage API errors, network issues, or invalid photo IDs.
- **Rate Limiting:** Be mindful of Unsplash API rate limits. If you exceed them, your requests might be temporarily blocked.