StravaAPILibary Documentation

Welcome to the comprehensive documentation for **StravaAPILibary**, a powerful .NET library for interacting with the Strava API.

🚀 Quick Start

Get up and running in minutes:

```
using StravaAPILibary.Authentication;
using StravaAPILibary.API;

// Set up credentials
var credentials = new Credentials("your_client_id", "your_client_secret",
"read,activity:read_all");

// Authenticate
var userAuth = new UserAuthentication(credentials, "http://localhost:8080/callback",
"read,activity:read_all");
userAuth.StartAuthorization();

// Exchange code for token
bool success = await userAuth.ExchangeCodeForTokenAsync("your_auth_code");

if (success)
{
    string accessToken = credentials.AccessToken;

    // Make API calls
    var activities = await Activities.GetAthletesActivitiesAsync(accessToken);
    var profile = await Athletes.GetAuthenticatedAthleteProfileAsync(accessToken);
}
```

📚 Documentation Sections

Getting Started

Complete setup guide with step-by-step instructions for installing and configuring the library.

Authentication Guide

Comprehensive guide to OAuth 2.0 flow, token management, and security best practices.

Examples

Practical examples and real-world scenarios showing how to use the library effectively.

API Reference

Complete API documentation with detailed method descriptions, parameters, and examples.

🔧 Features

- **Complete API Coverage** - All Strava API endpoints
- **OAuth 2.0 Authentication** - Secure token management
- **Strongly Typed** - Full IntelliSense support
- **Error Handling** - Comprehensive exception handling
- **Async/Await** - Modern asynchronous programming patterns

📦 Installation

```
dotnet add package StravaAPILibary
```

🔗 Links

- GitHub Repository
- NuGet Package
- Strava API Documentation

Build amazing Strava applications with confidence! 🚀

Getting Started with StravaAPILibary

This guide will walk you through setting up and using the StravaAPILibary to interact with the Strava API.

Prerequisites

Before you begin, ensure you have:

- **.NET 8.0 SDK** or later installed
- **Visual Studio 2022** or **JetBrains Rider** (recommended IDEs)
- **Strava API credentials** (Client ID and Client Secret)
- **Internet connection** for API calls

Step 1: Install the Library

Using NuGet Package Manager

1. Right-click on your project in Solution Explorer
2. Select **Manage NuGet Packages**
3. Search for `StravaAPILibary`
4. Click **Install**

Using Package Manager Console

```
Install-Package StravaAPILibary
```

Using .NET CLI

```
dotnet add package StravaAPILibary
```

Using PackageReference

Add this to your `.csproj` file:

```
<PackageReference Include="StravaAPILibary" Version="1.0.0" />
```

Step 2: Set Up Strava API Credentials

Create a Strava Application

1. Visit Strava API Settings
2. Click **Create Application**
3. Fill in the required information:

- **Application Name**: Your app name
- **Category**: Choose appropriate category
- **Website**: Your website URL
- **Authorization Callback Domain**: `localhost` (for development)

4. Click **Create**
5. Note your **Client ID** and **Client Secret**

Configure Redirect URI

For development, you can use:

- `http://localhost:8080/callback`
- `http://localhost:3000/callback`
- `http://localhost:5000/callback`

Step 3: Basic Authentication Flow

Create Your First Application

```
using StravaAPILibary.Authentication;
using StravaAPILibary.API;

class Program
{
    static async Task Main(string[] args)
    {
        // 1. Set up your credentials
        var credentials = new Credentials(
            "your_client_id_here",
            "your_client_secret_here",
            "read,activity:read_all"
        );

        // 2. Create authentication instance
        var userAuth = new UserAuthentication(
            credentials,
            "http://localhost:8080/callback",
            "read,activity:read_all"
        );

        // 3. Start the authorization process
        Console.WriteLine("Starting Strava authorization...");
        userAuth.StartAuthorization();

        // 4. Wait for user to complete authorization
```

```
        Console.WriteLine("Please complete the authorization in your browser.");
        Console.WriteLine("After authorization, copy the authorization code from the URL.");
        Console.Write("Enter authorization code: ");

        string authCode = Console.ReadLine() ?? string.Empty;

        // 5. Exchange code for access token
        if (!string.IsNullOrWhiteSpace(authCode))
        {
            bool success = await userAuth.ExchangeCodeForTokenAsync(authCode);

            if (success)
            {
                string accessToken = credentials.AccessToken;
                Console.WriteLine("✅ Authentication successful!");
                Console.WriteLine($"Access Token: {accessToken[..10]}...");

                // 6. Make your first API call
                await MakeFirstApiCall(accessToken);
            }
            else
            {
                Console.WriteLine("❌ Authentication failed!");
            }
        }
    }

    static async Task MakeFirstApiCall(string accessToken)
    {
        try
        {
            // Get athlete profile
            var profile = await Athletes.GetAuthenticatedAthleteProfileAsync(accessToken);
            Console.WriteLine($"Welcome, {profile["firstname"]} {profile["lastname"]}!");

            // Get recent activities
            var activities = await Activities.GetAthletesActivitiesAsync(accessToken, page:
1, perPage: 5);
            Console.WriteLine($"Found {activities.Count} recent activities:");

            foreach (var activity in activities)
            {
                Console.WriteLine($"- {activity["name"]} ({activity["type"]})
- {activity["distance"]}m");
            }
        }
```

```
        catch (Exception ex)
        {
            Console.WriteLine($"Error making API call: {ex.Message}");
        }
    }
}
```

Step 4: Understanding the Authentication Flow

OAuth 2.0 Flow

The library implements the standard OAuth 2.0 authorization code flow:

1. **Authorization Request**: User is redirected to Strava to authorize your application
2. **Authorization Code**: Strava returns an authorization code to your redirect URI
3. **Token Exchange**: Your application exchanges the code for an access token
4. **API Calls**: Use the access token to make API requests

Token Management

The library handles token management automatically:

- **Access Token**: Used for API requests (expires in 6 hours)
- **Refresh Token**: Used to get new access tokens (doesn't expire)
- **Token Expiration**: Automatically tracked and handled

Scopes

Request only the scopes you need:

- `read` - Basic profile access
- `activity:read_all` - Read all activities
- `activity:write` - Upload activities
- `profile:read_all` - Detailed profile access
- `profile:write` - Update profile information

Step 5: Common Usage Patterns

Error Handling

```
try
{
    var activities = await Activities.GetAthletesActivitiesAsync(accessToken);
    // Process activities
}
```

```
catch (ArgumentException ex)
{
    Console.WriteLine($"Invalid parameter: {ex.Message}");
}
catch (HttpRequestException ex)
{
    Console.WriteLine($"API request failed: {ex.Message}");
}
catch (JsonException ex)
{
    Console.WriteLine($"JSON parsing failed: {ex.Message}");
}
```

Pagination

```
int page = 1;
int perPage = 30;
bool hasMoreActivities = true;

while (hasMoreActivities)
{
    var activities = await Activities.GetAthletesActivitiesAsync(accessToken, page: page,
perPage: perPage);

    if (activities.Count == 0)
    {
        hasMoreActivities = false;
    }
    else
    {
        // Process activities
        foreach (var activity in activities)
        {
            Console.WriteLine($"Activity: {activity["name"]}");
        }

        page++;
    }
}
```

Filtering Activities

```
// Get activities from last 30 days
int after = (int)DateTimeOffset.UtcNow.AddDays(-30).ToUnixTimeSeconds();
```

```
var recentActivities = await Activities.GetAthletesActivitiesAsync(accessToken,
after: after);

// Get activities before a specific date
int before = (int)DateTimeOffset.Parse("2024-01-01").ToUnixTimeSeconds();
var oldActivities = await Activities.GetAthletesActivitiesAsync(accessToken,
before: before);
```

Step 6: Next Steps

Now that you have the basics working, explore:

- **API Reference** - Complete API documentation
- **Authentication Guide** - Advanced authentication topics
- **Examples** - More usage examples
- **Best Practices** - Recommended patterns

Troubleshooting

Common Issues

"Invalid authorization code"

- Ensure the authorization code is copied correctly
- Check that your redirect URI matches your Strava app settings
- Verify the code hasn't expired (codes expire quickly)

"Access token is invalid"

- The access token may have expired
- Use the refresh token to get a new access token
- Check that you're using the correct scope

"API request failed"

- Verify your internet connection
- Check that the Strava API is available
- Ensure your access token is valid

Getting Help

- **Documentation**: https://your-docs-site.com
- **GitHub Issues**: https://github.com/your-repo/issues
- **Strava API Status**: https://status.strava.com

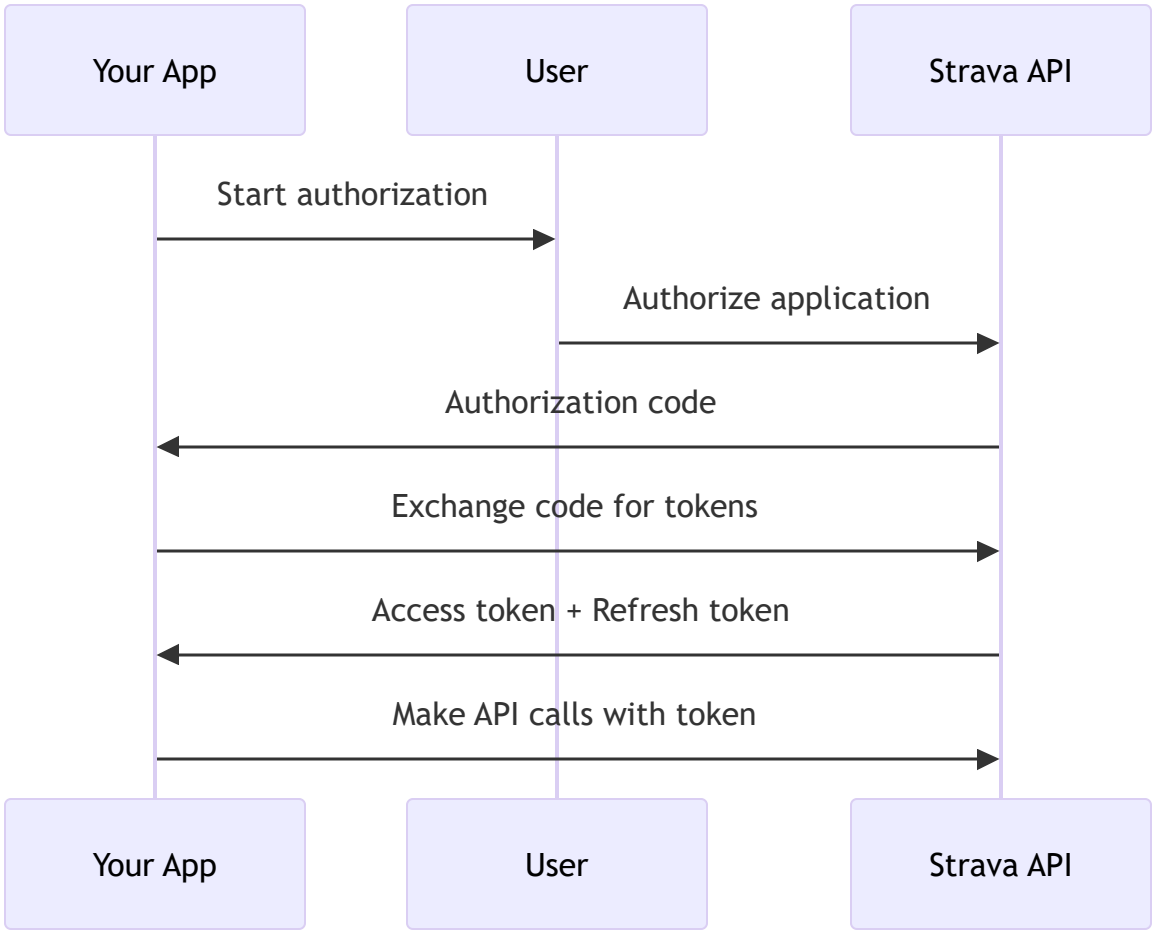
Ready to build amazing Strava applications! 🚀

Authentication Guide

This guide covers authentication with the Strava API using the StravaAPILibary, including OAuth 2.0 flow, token management, and best practices.

OAuth 2.0 Flow Overview

The StravaAPILibary implements the standard OAuth 2.0 authorization code flow:



Setting Up Authentication

1. Create Strava Application

First, create a Strava application to get your credentials:

1. Visit Strava API Settings
2. Click **Create Application**
3. Fill in the required fields:
   - **Application Name**: Your app name
   - **Category**: Choose appropriate category
   - **Website**: Your website URL
   - **Authorization Callback Domain**: `localhost` (for development)

4. Click **Create**
5. Note your **Client ID** and **Client Secret**

2. Configure Credentials

```
using StravaAPILibary.Authentication;

var credentials = new Credentials(
    "your_client_id",
    "your_client_secret",
    "read,activity:read_all"
);
```

3. Initialize Authentication

```
var userAuth = new UserAuthentication(
    credentials,
    "http://localhost:8080/callback",
    "read,activity:read_all"
);
```

Complete Authentication Flow

Step 1: Start Authorization

```
// This opens the browser for user authorization
userAuth.StartAuthorization();
```

The user will be redirected to Strava's authorization page where they can:

- Review the requested permissions
- Authorize or deny the application
- Be redirected back to your callback URL

Step 2: Handle Authorization Code

After authorization, Strava redirects to your callback URL with an authorization code:

```
http://localhost:8080/callback?state=&code=YOUR_AUTHORIZATION_CODE
```

Extract the code from the URL and exchange it for tokens:

```
string authCode = "YOUR_AUTHORIZATION_CODE";
bool success = await userAuth.ExchangeCodeForTokenAsync(authCode);

if (success)
{
    string accessToken = credentials.AccessToken;
    string refreshToken = credentials.RefreshToken;
    DateTime tokenExpiry = credentials.TokenExpiration;

    Console.WriteLine("✅ Authentication successful!");
}
```

Step 3: Use Access Token

```
// Make API calls with the access token
var activities = await Activities.GetAthletesActivitiesAsync(accessToken);
var profile = await Athletes.GetAuthenticatedAthleteProfileAsync(accessToken);
```

Token Management

Access Tokens

- **Lifetime**: 6 hours
- **Usage**: Required for all API calls
- **Storage**: Store securely (environment variables, secure storage)

Refresh Tokens

- **Lifetime**: Indefinite (until revoked)
- **Usage**: Get new access tokens when they expire
- **Storage**: Store securely alongside access tokens

Automatic Token Refresh

The library can automatically refresh expired tokens:

```
// Check if token needs refresh
if (credentials.TokenExpiration <= DateTime.UtcNow.AddMinutes(5))
{
    bool refreshSuccess = await userAuth.RefreshAccessTokenAsync();
    if (refreshSuccess)
    {
        // Token refreshed successfully
        accessToken = credentials.AccessToken;
```

```
    }
}
```

Scopes and Permissions

Available Scopes

Scope	Description	Required for
`read`	Basic profile access	Profile information
`activity:read_all`	Read all activities	Activity data
`activity:write`	Upload activities	Activity upload
`profile:read_all`	Detailed profile access	Detailed profile
`profile:write`	Update profile	Profile updates

Requesting Scopes

```
// Minimal scope for basic functionality
var credentials = new Credentials(clientId, clientSecret, "read");

// Full scope for complete functionality
var credentials = new Credentials(clientId, clientSecret,
"read,activity:read_all,activity:write,profile:read_all");
```

Scope Best Practices

- **Request minimal scopes** - Only request what you need
- **Explain permissions** - Tell users why you need each scope
- **Handle denied scopes** - Gracefully handle when users deny permissions

Advanced Authentication Patterns

1. Persistent Token Storage

```
public class TokenManager
{
    private readonly string _tokenFilePath = "tokens.json";

    public async Task SaveTokensAsync(Credentials credentials)
    {
```

```
        var tokenData = new
        {
            AccessToken = credentials.AccessToken,
            RefreshToken = credentials.RefreshToken,
            TokenExpiration = credentials.TokenExpiration,
            Scope = credentials.Scope
        };

        string json = JsonSerializer.Serialize(tokenData);
        await File.WriteAllTextAsync(_tokenFilePath, json);
    }

    public async Task<Credentials?> LoadTokensAsync(string clientId, string clientSecret)
    {
        if (!File.Exists(_tokenFilePath))
            return null;

        string json = await File.ReadAllTextAsync(_tokenFilePath);
        var tokenData = JsonSerializer.Deserialize<TokenData>(json);

        if (tokenData == null)
            return null;

        var credentials = new Credentials(clientId, clientSecret, tokenData.Scope)
        {
            AccessToken = tokenData.AccessToken,
            RefreshToken = tokenData.RefreshToken,
            TokenExpiration = tokenData.TokenExpiration
        };

        return credentials;
    }
}
```

2. Automatic Token Refresh

```
public class StravaClient
{
    private readonly Credentials _credentials;
    private readonly UserAuthentication _userAuth;

    public StravaClient(string clientId, string clientSecret, string scope)
    {
        _credentials = new Credentials(clientId, clientSecret, scope);
        _userAuth = new UserAuthentication(_credentials, "http://localhost:8080/callback",
```

```
scope);
    }

    public async Task<string> GetValidAccessTokenAsync()
    {
        // Check if token is expired or will expire soon
        if (_credentials.TokenExpiration <= DateTime.UtcNow.AddMinutes(5))
        {
            bool refreshSuccess = await _userAuth.RefreshAccessTokenAsync();
            if (!refreshSuccess)
            {
                throw new InvalidOperationException("Failed to refresh access token");
            }
        }

        return _credentials.AccessToken;
    }

    public async Task<JsonArray> GetActivitiesAsync()
    {
        string accessToken = await GetValidAccessTokenAsync();
        return await Activities.GetAthletesActivitiesAsync(accessToken);
    }
}
```

3. Error Handling

```
public async Task<bool> AuthenticateWithRetryAsync()
{
    int maxRetries = 3;
    int retryCount = 0;

    while (retryCount < maxRetries)
    {
        try
        {
            // Try to refresh token first
            if (!string.IsNullOrEmpty(_credentials.RefreshToken))
            {
                bool refreshSuccess = await _userAuth.RefreshAccessTokenAsync();
                if (refreshSuccess)
                {
                    return true;
                }
            }
```

```
            // If refresh fails, start new authorization
            _userAuth.StartAuthorization();

            Console.WriteLine("Please complete authorization in browser...");
            Console.Write("Enter authorization code: ");

            string authCode = Console.ReadLine() ?? string.Empty;

            if (!string.IsNullOrEmpty(authCode))
            {
                bool success = await _userAuth.ExchangeCodeForTokenAsync(authCode);
                if (success)
                {
                    return true;
                }
            }

            retryCount++;
            Console.WriteLine($"Authentication failed. Retry {retryCount}/{maxRetries}");
        }
        catch (Exception ex)
        {
            Console.WriteLine($"Authentication error: {ex.Message}");
            retryCount++;
        }
    }

    return false;
}
```

Security Best Practices

1. Secure Token Storage

```
// ❌ Don't store tokens in plain text
File.WriteAllText("tokens.txt", accessToken);

// ✅ Use secure storage
await SecureStorage.SaveAsync("strava_access_token", accessToken);
await SecureStorage.SaveAsync("strava_refresh_token", refreshToken);
```

2. Environment Variables

```
// Load credentials from environment variables
string clientId = Environment.GetEnvironmentVariable("STRAVA_CLIENT_ID")
    ?? throw new InvalidOperationException("STRAVA_CLIENT_ID not set");

string clientSecret = Environment.GetEnvironmentVariable("STRAVA_CLIENT_SECRET")
    ?? throw new InvalidOperationException("STRAVA_CLIENT_SECRET not set");
```

3. Token Validation

```
public bool IsTokenValid(Credentials credentials)
{
    return !string.IsNullOrEmpty(credentials.AccessToken) &&
           credentials.TokenExpiration > DateTime.UtcNow.AddMinutes(5);
}
```

Troubleshooting

Common Issues

"Invalid authorization code"

- Authorization codes expire quickly (usually within 10 minutes)
- Ensure the code is copied correctly from the URL
- Check that your redirect URI matches your Strava app settings

"Access token is invalid"

- Access tokens expire after 6 hours
- Use the refresh token to get a new access token
- Verify the token hasn't been revoked by the user

"Refresh token is invalid"

- Refresh tokens can be revoked by users
- Users can revoke access in their Strava settings
- You'll need to re-authenticate the user

"Insufficient scope"

- The requested operation requires a scope not granted
- Check the required scopes for the API call
- Request the appropriate scope during authorization

Debug Information

```
public void LogTokenInfo(Credentials credentials)
{
    Console.WriteLine($"Access Token: {credentials.AccessToken[..10]}...");
    Console.WriteLine($"Refresh Token: {credentials.RefreshToken[..10]}...");
    Console.WriteLine($"Token Expires: {credentials.TokenExpiration}");
    Console.WriteLine($"Scope: {credentials.Scope}");
    Console.WriteLine($"Is Expired: {credentials.TokenExpiration <= DateTime.UtcNow}");
}
```

Next Steps

- **API Reference** - Complete API documentation
- **Examples** - Authentication examples
- **Best Practices** - Security and performance tips

Secure authentication is the foundation of great Strava applications! 💪

Examples

This guide provides practical examples of using the StravaAPILibary for common scenarios and real-world applications.

Basic Examples

Get Athlete Profile

```
using StravaAPILibary.API;

var profile = await Athletes.GetAuthenticatedAthleteProfileAsync(accessToken);

Console.WriteLine($"Name: {profile["firstname"]} {profile["lastname"]}");
Console.WriteLine($"Location: {profile["city"]}, {profile["state"]}");
Console.WriteLine($"Followers: {profile["follower_count"]}");
Console.WriteLine($"Following: {profile["friend_count"]}");
```

Get Recent Activities

```
var activities = await Activities.GetAthletesActivitiesAsync(accessToken, page: 1,
perPage: 10);

foreach (var activity in activities)
{
    Console.WriteLine($"Activity: {activity["name"]}");
    Console.WriteLine($"  Type: {activity["type"]}");
    Console.WriteLine($"  Distance: {activity["distance"]}m");
    Console.WriteLine($"  Duration: {activity["moving_time"]}s");
    Console.WriteLine($"  Date: {activity["start_date_local"]}");
    Console.WriteLine();
}
```

Get Activity Details

```
string activityId = "123456789";
var activity = await Activities.GetActivityByIdAsync(accessToken, activityId);

Console.WriteLine($"Activity: {activity["name"]}");
Console.WriteLine($"  Description: {activity["description"]}");
Console.WriteLine($"  Distance: {activity["distance"]}m");
Console.WriteLine($"  Elevation Gain: {activity["total_elevation_gain"]}m");
```

```
Console.WriteLine($"  Average Speed: {activity["average_speed"]} m/s");
Console.WriteLine($"  Max Speed: {activity["max_speed"]} m/s");
```

Advanced Examples

Activity Analysis

```
public class ActivityAnalyzer
{
    public async Task AnalyzeActivitiesAsync(string accessToken)
    {
        var activities = await Activities.GetAthletesActivitiesAsync(accessToken, page: 1,
perPage: 50);

        var stats = new
        {
            TotalActivities = activities.Count,
            TotalDistance = activities.Sum(a => (double)a["distance"]),
            TotalTime = activities.Sum(a => (int)a["moving_time"]),
            ActivitiesByType = activities.GroupBy(a => (string)a["type"])
                                         .ToDictionary(g => g.Key, g => g.Count())
        };

        Console.WriteLine($"Total Activities: {stats.TotalActivities}");
        Console.WriteLine($"Total Distance: {stats.TotalDistance / 1000:F1}km");
        Console.WriteLine($"Total Time: {TimeSpan.FromSeconds(stats.TotalTime)}");

        foreach (var type in stats.ActivitiesByType)
        {
            Console.WriteLine($"  {type.Key}: {type.Value} activities");
        }
    }
}
```

Segment Explorer

```
public class SegmentExplorer
{
    public async Task ExploreSegmentsAsync(string accessToken, float[] bounds, string
activityType = "running")
    {
        var segments = await Segments.GetExploreSegmentsAsync(accessToken,
bounds, activityType);
```

```
        Console.WriteLine($"Found {segments.Count} segments in the area:");

        foreach (var segment in segments)
        {
            Console.WriteLine($"  {segment["name"]}");
            Console.WriteLine($"    Distance: {segment["distance"]}m");
            Console.WriteLine($"    Average Grade: {segment["average_grade"]}%");
            Console.WriteLine($"    Elevation Difference: {segment["elevation_high"]
- segment["elevation_low"]}m");
            Console.WriteLine();
        }
    }

    public async Task GetStarredSegmentsAsync(string accessToken)
    {
        var starredSegments = await Segments.GetStarredSegmentsAsync(accessToken);

        Console.WriteLine($"You have {starredSegments.Count} starred segments:");

        foreach (var segment in starredSegments)
        {
            Console.WriteLine($"  {segment["name"]} - {segment["distance"]}m");
        }
    }
}
```

Club Activities

```
public class ClubManager
{
    public async Task GetClubInfoAsync(string accessToken)
    {
        var clubs = await Clubs.GetClubsAsync(accessToken);

        foreach (var club in clubs)
        {
            Console.WriteLine($"Club: {club["name"]}");
            Console.WriteLine($"  Members: {club["member_count"]}");
            Console.WriteLine($"  Activities: {club["activity_count"]}");
            Console.WriteLine($"  Description: {club["description"]}");
            Console.WriteLine();

            // Get club activities
            var clubId = (long)club["id"];
            var activities = await Clubs.GetClubActivitiesAsync(accessToken, clubId, page:
```

```
1, perPage: 5);

            Console.WriteLine("  Recent Activities:");
            foreach (var activity in activities)
            {
                Console.WriteLine($"    {activity["athlete"]["firstname"]}
{activity["athlete"]["lastname"]}: {activity["name"]}");
            }
            Console.WriteLine();
        }
    }
}
```

Route Management

```
public class RouteManager
{
    public async Task ExportRouteAsync(string accessToken, long routeId, string outputPath)
    {
        // Get route details
        var route = await Routes.GetRouteByIdAsync(accessToken, routeId);
        Console.WriteLine($"Route: {route["name"]}");
        Console.WriteLine($"  Distance: {route["distance"]}m");
        Console.WriteLine($"  Elevation Gain: {route["elevation_gain"]}m");

        // Export as GPX
        var gpxData = await Routes.GetRouteGpxExportAsync(accessToken, routeId);
        await File.WriteAllTextAsync($"{outputPath}.gpx", gpxData);

        // Export as TCX
        var tcxData = await Routes.GetRouteTcxExportAsync(accessToken, routeId);
        await File.WriteAllTextAsync($"{outputPath}.tcx", tcxData);

        Console.WriteLine($"Route exported to {outputPath}.gpx and {outputPath}.tcx");
    }
}
```

Activity Upload

```
public class ActivityUploader
{
    public async Task UploadActivityAsync(string accessToken, string filePath, string
activityName, string description = "")
    {
```

```
        try
        {
            // Determine file type
            string dataType = Path.GetExtension(filePath).ToLower() switch
            {
                ".gpx" => "gpx",
                ".tcx" => "tcx",
                ".fit" => "fit",
                _ => throw new ArgumentException("Unsupported file format. Use GPX, TCX, or
FIT files.")
            };

            // Upload activity
            var uploadResponse = await Activities.PostActivityAsync(
                accessToken,
                activityName,
                dataType,
                filePath,
                description
            );

            Console.WriteLine($"Upload started. Upload ID: {uploadResponse["id"]}");

            // Monitor upload status
            await MonitorUploadStatusAsync(accessToken, (long)uploadResponse["id"]);
        }
        catch (Exception ex)
        {
            Console.WriteLine($"Upload failed: {ex.Message}");
        }
    }

    private async Task MonitorUploadStatusAsync(string accessToken, long uploadId)
    {
        int maxAttempts = 30; // 5 minutes with 10-second intervals
        int attempts = 0;

        while (attempts < maxAttempts)
        {
            var uploadStatus = await Uploads.GetUploadAsync(accessToken, uploadId);
            string status = (string)uploadStatus["status"];

            Console.WriteLine($"Upload status: {status}");

            if (status == "Your activity is ready.")
            {
```

```
                Console.WriteLine("✅ Upload completed successfully!");
                Console.WriteLine($"Activity ID: {uploadStatus["activity_id"]}");
                break;
            }
            else if (status == "There was an error processing your activity.")
            {
                Console.WriteLine("❌ Upload failed!");
                Console.WriteLine($"Error: {uploadStatus["error"]}");
                break;
            }

            await Task.Delay(10000); // Wait 10 seconds
            attempts++;
        }

        if (attempts >= maxAttempts)
        {
            Console.WriteLine("⏰ Upload monitoring timed out.");
        }
    }
}
```

Stream Data Analysis

```
public class StreamAnalyzer
{
    public async Task AnalyzeActivityStreamsAsync(string accessToken, long activityId)
    {
        var streams = await Streams.GetActivityStreamsAsync(
            accessToken,
            activityId,

"time,distance,latlng,altitude,velocity_smooth,heartrate,cadence,watts,temp,moving,grade_smo
oth"
        );

        if (streams.ContainsKey("latlng"))
        {
            var latlngStream = streams["latlng"] as JsonArray;
            Console.WriteLine($"GPS Points: {latlngStream?.Count ?? 0}");
        }

        if (streams.ContainsKey("heartrate"))
        {
            var hrStream = streams["heartrate"] as JsonArray;
```

```
            if (hrStream?.Count > 0)
            {
                var hrValues = hrStream.Select(x => (int)x).ToList();
                Console.WriteLine($"Heart Rate - Avg: {hrValues.Average():F0}, Max:
{hrValues.Max()}, Min: {hrValues.Min()}");
            }
        }

        if (streams.ContainsKey("velocity_smooth"))
        {
            var speedStream = streams["velocity_smooth"] as JsonArray;
            if (speedStream?.Count > 0)
            {
                var speedValues = speedStream.Select(x => (double)x).ToList();
                Console.WriteLine($"Speed - Avg: {speedValues.Average() * 3.6:F1} km/h, Max:
{speedValues.Max() * 3.6:F1} km/h");
            }
        }
    }
}
```

Real-World Scenarios

Fitness Dashboard

```
public class FitnessDashboard
{
    public async Task GenerateDashboardAsync(string accessToken)
    {
        var profile = await Athletes.GetAuthenticatedAthleteProfileAsync(accessToken);
        var activities = await Activities.GetAthletesActivitiesAsync(accessToken, page: 1,
perPage: 30);

        Console.WriteLine("=== FITNESS DASHBOARD ===");
        Console.WriteLine($"Athlete: {profile["firstname"]} {profile["lastname"]}");
        Console.WriteLine($"Location: {profile["city"]}, {profile["state"]}");
        Console.WriteLine();

        // Weekly summary
        var weekAgo = DateTime.UtcNow.AddDays(-7);
        var recentActivities = activities.Where(a =>
            DateTime.Parse((string)a["start_date_local"]) >= weekAgo).ToList();

        var weeklyStats = new
        {
```

```
            Activities = recentActivities.Count,
            Distance = recentActivities.Sum(a => (double)a["distance"]) / 1000,
            Time = recentActivities.Sum(a => (int)a["moving_time"]),
            Elevation = recentActivities.Sum(a => (double)a["total_elevation_gain"])
        };

        Console.WriteLine("This Week:");
        Console.WriteLine($"  Activities: {weeklyStats.Activities}");
        Console.WriteLine($"  Distance: {weeklyStats.Distance:F1} km");
        Console.WriteLine($"  Time: {TimeSpan.FromSeconds(weeklyStats.Time)}");
        Console.WriteLine($"  Elevation: {weeklyStats.Elevation:F0} m");
        Console.WriteLine();

        // Activity breakdown
        var activityTypes = recentActivities.GroupBy(a => (string)a["type"])
                                            .OrderByDescending(g => g.Count());

        Console.WriteLine("Activity Breakdown:");
        foreach (var type in activityTypes)
        {
            var typeStats = type.ToList();
            var avgDistance = typeStats.Average(a => (double)a["distance"]) / 1000;
            Console.WriteLine($"  {type.Key}: {type.Count()} activities, avg
{avgDistance:F1} km");
        }
    }
}
```

Training Plan Generator

```
public class TrainingPlanGenerator
{
    public async Task GenerateTrainingPlanAsync(string accessToken, string targetEvent,
DateTime targetDate)
    {
        var activities = await Activities.GetAthletesActivitiesAsync(accessToken, page: 1,
perPage: 100);
        var profile = await Athletes.GetAuthenticatedAthleteProfileAsync(accessToken);

        Console.WriteLine($"=== TRAINING PLAN FOR {targetEvent.ToUpper()} ===");
        Console.WriteLine($"Target Date: {targetDate:MMMM dd, yyyy}");
        Console.WriteLine($"Days until event: {(targetDate - DateTime.Now).Days}");
        Console.WriteLine();

        // Analyze current fitness level
```

```
        var recentRuns = activities.Where(a =>
            (string)a["type"] == "Run" &&
            DateTime.Parse((string)a["start_date_local"]) >= DateTime.Now.AddDays(-30)
        ).ToList();

        if (recentRuns.Any())
        {
            var avgPace = recentRuns.Average(a => (double)a["average_speed"]);
            var maxDistance = recentRuns.Max(a => (double)a["distance"]);

            Console.WriteLine("Current Fitness Level:");
            Console.WriteLine($"  Average Pace: {60 / (avgPace * 3.6):F1} min/km");
            Console.WriteLine($"  Longest Run: {maxDistance / 1000:F1} km");
            Console.WriteLine();

            // Generate training plan
            GenerateWeeklyPlan(targetDate, maxDistance / 1000);
        }
    }

    private void GenerateWeeklyPlan(DateTime targetDate, double currentLongestRun)
    {
        var weeksUntilEvent = (int)((targetDate - DateTime.Now).TotalDays / 7);

        Console.WriteLine("Recommended Training Plan:");
        for (int week = 1; week <= Math.Min(weeksUntilEvent, 12); week++)
        {
            var weekDate = DateTime.Now.AddDays(week * 7);
            var targetDistance = Math.Min(currentLongestRun + (week * 2), 42.2); // Cap at
marathon distance

            Console.WriteLine($"Week {week} ({weekDate:MMM dd}):");
            Console.WriteLine($"  Long Run: {targetDistance:F1} km");
            Console.WriteLine($"  Tempo Run: {targetDistance * 0.6:F1} km");
            Console.WriteLine($"  Easy Runs: 2x {targetDistance * 0.4:F1} km");
            Console.WriteLine();
        }
    }
}
```

Social Features

```
public class SocialFeatures
{
    public async Task GetSocialFeedAsync(string accessToken)
```

```
    {
        var clubs = await Clubs.GetClubsAsync(accessToken);

        Console.WriteLine("=== SOCIAL FEED ===");

        foreach (var club in clubs)
        {
            Console.WriteLine($"Club: {club["name"]}");

            var clubId = (long)club["id"];
            var activities = await Clubs.GetClubActivitiesAsync(accessToken, clubId, page:
1, perPage: 5);

            foreach (var activity in activities)
            {
                var athlete = activity["athlete"];
                Console.WriteLine($"  {athlete["firstname"]} {athlete["lastname"]}:
{activity["name"]}");
                Console.WriteLine($"    {activity["type"]} - {((double)activity["distance"]
/ 1000):F1} km");
                Console.WriteLine($"    {activity["start_date_local"]}");
                Console.WriteLine();
            }
        }
    }

    public async Task GetActivityKudosAsync(string accessToken, string activityId)
    {
        var kudos = await Activities.GetActivityKudosAsync(accessToken, activityId);

        Console.WriteLine($"Kudos for activity {activityId}:");
        foreach (var kudo in kudos)
        {
            Console.WriteLine($"  {kudo["firstname"]} {kudo["lastname"]}");
        }
    }
}
```

Error Handling Examples

Robust API Client

```
public class RobustStravaClient
{
    private readonly string _accessToken;
```

```
    private readonly HttpClient _httpClient;

    public RobustStravaClient(string accessToken)
    {
        _accessToken = accessToken;
        _httpClient = new HttpClient();
        _httpClient.DefaultRequestHeaders.Authorization =
            new System.Net.Http.Headers.AuthenticationHeaderValue("Bearer", accessToken);
    }

    public async Task<JsonArray?> GetActivitiesWithRetryAsync(int maxRetries = 3)
    {
        for (int attempt = 1; attempt <= maxRetries; attempt++)
        {
            try
            {
                return await Activities.GetAthletesActivitiesAsync(_accessToken);
            }
            catch (HttpRequestException ex) when (ex.StatusCode ==
System.Net.HttpStatusCode.TooManyRequests)
            {
                if (attempt < maxRetries)
                {
                    var retryAfter = GetRetryAfterSeconds(ex);
                    Console.WriteLine($"Rate limited. Waiting {retryAfter} seconds...");
                    await Task.Delay(retryAfter * 1000);
                }
                else
                {
                    throw new InvalidOperationException("Rate limit exceeded after multiple
retries", ex);
                }
            }
            catch (HttpRequestException ex) when (ex.StatusCode ==
System.Net.HttpStatusCode.Unauthorized)
            {
                throw new InvalidOperationException("Access token is invalid or
expired", ex);
            }
            catch (Exception ex)
            {
                if (attempt == maxRetries)
                    throw;

                Console.WriteLine($"Attempt {attempt} failed: {ex.Message}");
                await Task.Delay(1000 * attempt); // Exponential backoff
```

```
            }
        }

        return null;
    }

    private int GetRetryAfterSeconds(HttpRequestException ex)
    {
        // Parse Retry-After header if available
        if (ex.Data.Contains("RetryAfter"))
        {
            return (int)ex.Data["RetryAfter"];
        }
        return 60; // Default 1 minute
    }
}
```

Next Steps

- **API Reference** - Complete API documentation
- **Authentication Guide** - Authentication patterns
- **Best Practices** - Performance and security tips

Build amazing Strava applications with these examples! 🚀

Best Practices

This guide covers best practices for using the StravaAPILibary effectively, securely, and efficiently.

🔐 Security Best Practices

1. Secure Credential Management

✅ **Do:**

```
// Use environment variables for sensitive data
string clientId = Environment.GetEnvironmentVariable("STRAVA_CLIENT_ID")
    ?? throw new InvalidOperationException("STRAVA_CLIENT_ID not set");
string clientSecret = Environment.GetEnvironmentVariable("STRAVA_CLIENT_SECRET")
    ?? throw new InvalidOperationException("STRAVA_CLIENT_SECRET not set");

var credentials = new Credentials(clientId, clientSecret, "read,activity:read_all");
```

❌ **Don't:**

```
// Never hardcode credentials
var credentials = new Credentials("12345", "my_secret_key", "read");
```

2. Token Storage

✅ **Do:**

```
// Use secure storage for tokens
public class SecureTokenStorage
{
    public async Task SaveTokensAsync(Credentials credentials)
    {
        // Use platform-specific secure storage
        await SecureStorage.SaveAsync("strava_access_token", credentials.AccessToken);
        await SecureStorage.SaveAsync("strava_refresh_token", credentials.RefreshToken);
        await SecureStorage.SaveAsync("strava_token_expiry",
credentials.TokenExpiration.ToString());
    }

    public async Task<Credentials?> LoadTokensAsync(string clientId, string clientSecret)
    {
        var accessToken = await SecureStorage.GetAsync("strava_access_token");
        var refreshToken = await SecureStorage.GetAsync("strava_refresh_token");
        var expiryStr = await SecureStorage.GetAsync("strava_token_expiry");
```

```
        if (string.IsNullOrEmpty(accessToken) || string.IsNullOrEmpty(refreshToken))
            return null;

        var credentials = new Credentials(clientId, clientSecret, "read,activity:read_all")
        {
            AccessToken = accessToken,
            RefreshToken = refreshToken,
            TokenExpiration = DateTime.Parse(expiryStr)
        };

        return credentials;
    }
}
```

❌ **Don't:**

```
// Never store tokens in plain text files
File.WriteAllText("tokens.txt", accessToken);
```

3. Scope Management

✅ **Do:**

```
// Request minimal scopes
var credentials = new Credentials(clientId, clientSecret, "read,activity:read_all");

// Check if required scope is available
bool hasActivityWrite = credentials.Scope.Contains("activity:write");
if (!hasActivityWrite)
{
    throw new InvalidOperationException("activity:write scope is required for
this operation.");
}
```

❌ **Don't:**

```
// Don't request unnecessary scopes
var credentials = new Credentials(clientId, clientSecret,
"read,activity:read_all,activity:write,profile:read_all,profile:write");
```

⚡ Performance Best Practices

1. Efficient API Usage

✅ **Do:**

```
public class EfficientStravaClient
{
    private readonly HttpClient _httpClient;
    private readonly string _accessToken;

    public EfficientStravaClient(string accessToken)
    {
        _accessToken = accessToken;
        _httpClient = new HttpClient
        {
            Timeout = TimeSpan.FromSeconds(30),
            DefaultRequestHeaders =
            {
                Authorization = new AuthenticationHeaderValue("Bearer", accessToken)
            }
        };
    }

    public async Task<JsonArray> GetActivitiesAsync(int page = 1, int perPage = 200)
    {
        // Use maximum per_page to reduce API calls
        return await Activities.GetAthletesActivitiesAsync(_accessToken, page: page,
perPage: perPage);
    }
}
```

❌ **Don't:**

```
// Don't make many small requests
for (int i = 1; i <= 100; i++)
{
    var activities = await Activities.GetAthletesActivitiesAsync(accessToken, page: i,
perPage: 1);
    // Process single activity
}
```

2. Caching Strategies

✅ **Do:**

```
public class CachedStravaClient
{
    private readonly IMemoryCache _cache;
    private readonly string _accessToken;

    public CachedStravaClient(string accessToken, IMemoryCache cache)
    {
        _accessToken = accessToken;
        _cache = cache;
    }

    public async Task<JsonObject> GetAthleteProfileAsync()
    {
        const string cacheKey = "athlete_profile";

        if (_cache.TryGetValue(cacheKey, out JsonObject? cachedProfile))
        {
            return cachedProfile!;
        }

        var profile = await Athletes.GetAuthenticatedAthleteProfileAsync(_accessToken);

        // Cache for 1 hour (profile doesn't change frequently)
        _cache.Set(cacheKey, profile, TimeSpan.FromHours(1));

        return profile;
    }
}
```

3. Batch Processing

✅ **Do:**

```
public class BatchActivityProcessor
{
    public async Task ProcessAllActivitiesAsync(string accessToken)
    {
        int page = 1;
        int perPage = 200; // Maximum to reduce API calls
        var allActivities = new List<JsonNode>();

        while (true)
        {
            var activities = await Activities.GetAthletesActivitiesAsync(accessToken, page:
page, perPage: perPage);
```

```
            if (activities.Count == 0)
                break;

            allActivities.AddRange(activities);
            page++;
        }

        // Process all activities at once
        await ProcessActivitiesBatchAsync(allActivities);
    }

    private async Task ProcessActivitiesBatchAsync(List<JsonNode> activities)
    {
        // Process activities in batches for efficiency
        const int batchSize = 50;

        for (int i = 0; i < activities.Count; i += batchSize)
        {
            var batch = activities.Skip(i).Take(batchSize);
            await ProcessBatchAsync(batch);
        }
    }
}
```

Error Handling Best Practices

1. Comprehensive Exception Handling

✅ **Do:**

```
public class RobustStravaClient
{
    public async Task<JsonArray?> GetActivitiesWithRetryAsync(int maxRetries = 3)
    {
        for (int attempt = 1; attempt <= maxRetries; attempt++)
        {
            try
            {
                return await Activities.GetAthletesActivitiesAsync(_accessToken);
            }
            catch (HttpRequestException ex) when (ex.StatusCode ==
HttpStatusCode.TooManyRequests)
            {
                if (attempt < maxRetries)
```

```
                {
                    var retryAfter = GetRetryAfterSeconds(ex);
                    await Task.Delay(retryAfter * 1000);
                }
                else
                {
                    throw new InvalidOperationException("Rate limit exceeded after multiple
retries", ex);
                }
            }
            catch (HttpRequestException ex) when (ex.StatusCode
== HttpStatusCode.Unauthorized)
            {
                // Token might be expired, try to refresh
                if (await TryRefreshTokenAsync())
                {
                    continue; // Retry with new token
                }
                throw new InvalidOperationException("Access token is invalid and refresh
failed", ex);
            }
            catch (Exception ex)
            {
                if (attempt == maxRetries)
                    throw;

                await Task.Delay(1000 * attempt); // Exponential backoff
            }
        }

        return null;
    }
}
```

2. Token Refresh Logic

✅ **Do:**

```
public class TokenManager
{
    private readonly Credentials _credentials;
    private readonly UserAuthentication _userAuth;

    public async Task<string> GetValidAccessTokenAsync()
    {
```

```
        // Check if token is expired or will expire soon
        if (_credentials.TokenExpiration <= DateTime.UtcNow.AddMinutes(5))
        {
            bool refreshSuccess = await _userAuth.RefreshAccessTokenAsync();
            if (!refreshSuccess)
            {
                throw new InvalidOperationException("Failed to refresh access token. User
needs to re-authenticate.");
            }
        }

        return _credentials.AccessToken;
    }

    public async Task<bool> TryRefreshTokenAsync()
    {
        try
        {
            return await _userAuth.RefreshAccessTokenAsync();
        }
        catch (Exception)
        {
            return false;
        }
    }
}
```

3. Graceful Degradation

✅ **Do:**

```
public class StravaService
{
    public async Task<ActivitySummary> GetActivitySummaryAsync(string accessToken)
    {
        try
        {
            var activities = await Activities.GetAthletesActivitiesAsync(accessToken, page:
1, perPage: 10);

            return new ActivitySummary
            {
                TotalActivities = activities.Count,
                TotalDistance = activities.Sum(a => (double)a["distance"]),
                IsComplete = true
```

```
            };
        }
        catch (HttpRequestException ex) when (ex.StatusCode ==
HttpStatusCode.TooManyRequests)
        {
            // Return partial data when rate limited
            return new ActivitySummary
            {
                TotalActivities = 0,
                TotalDistance = 0,
                IsComplete = false,
                ErrorMessage = "Rate limited - showing cached data"
            };
        }
        catch (Exception ex)
        {
            // Log error and return empty result
            _logger.LogError(ex, "Failed to get activity summary");
            return new ActivitySummary
            {
                TotalActivities = 0,
                TotalDistance = 0,
                IsComplete = false,
                ErrorMessage = "Service temporarily unavailable"
            };
        }
    }
}
```

📊 Monitoring and Logging

1. Structured Logging

✅ **Do:**

```
public class StravaClientWithLogging
{
    private readonly ILogger<StravaClientWithLogging> _logger;

    public async Task<JsonArray> GetActivitiesAsync(string accessToken, int page = 1, int
perPage = 30)
    {
        using var scope = _logger.BeginScope(new Dictionary<string, object>
        {
            ["page"] = page,
```

```
            ["per_page"] = perPage
        });

        _logger.LogInformation("Retrieving activities from Strava API");

        try
        {
            var activities = await Activities.GetAthletesActivitiesAsync(accessToken, page:
page, perPage: perPage);

            _logger.LogInformation("Successfully retrieved {Count}
activities", activities.Count);

            return activities;
        }
        catch (HttpRequestException ex)
        {
            _logger.LogError(ex, "Failed to retrieve activities. Status:
{StatusCode}", ex.StatusCode);
            throw;
        }
        catch (Exception ex)
        {
            _logger.LogError(ex, "Unexpected error while retrieving activities");
            throw;
        }
    }
}
```

2. Metrics Collection

✅ **Do:**

```
public class StravaClientWithMetrics
{
    private readonly IMetrics _metrics;

    public async Task<JsonArray> GetActivitiesAsync(string accessToken)
    {
        using var timer = _metrics.CreateTimer("strava.api.activities.duration");

        try
        {
            var activities = await Activities.GetAthletesActivitiesAsync(accessToken);
```

```
            _metrics.Increment("strava.api.activities.success");
            _metrics.RecordGauge("strava.api.activities.count", activities.Count);

            return activities;
        }
        catch (Exception ex)
        {
            _metrics.Increment("strava.api.activities.error");
            throw;
        }
    }
}
```

🔄 Design Patterns

1. Repository Pattern

✅ **Do:**

```
public interface IStravaRepository
{
    Task<JsonArray> GetActivitiesAsync(int page = 1, int perPage = 30);
    Task<JsonObject> GetActivityAsync(string activityId);
    Task<JsonObject> GetAthleteProfileAsync();
    Task<bool> UpdateActivityAsync(long activityId, string name, string description);
}

public class StravaRepository : IStravaRepository
{
    private readonly string _accessToken;
    private readonly TokenManager _tokenManager;

    public StravaRepository(string accessToken, TokenManager tokenManager)
    {
        _accessToken = accessToken;
        _tokenManager = tokenManager;
    }

    public async Task<JsonArray> GetActivitiesAsync(int page = 1, int perPage = 30)
    {
        string validToken = await _tokenManager.GetValidAccessTokenAsync();
        return await Activities.GetAthletesActivitiesAsync(validToken, page: page,
perPage: perPage);
    }
```

```
    public async Task<JsonObject> GetActivityAsync(string activityId)
    {
        string validToken = await _tokenManager.GetValidAccessTokenAsync();
        return await Activities.GetActivityByIdAsync(validToken, activityId);
    }

    public async Task<JsonObject> GetAthleteProfileAsync()
    {
        string validToken = await _tokenManager.GetValidAccessTokenAsync();
        return await Athletes.GetAuthenticatedAthleteProfileAsync(validToken);
    }

    public async Task<bool> UpdateActivityAsync(long activityId, string name,
string description)
    {
        string validToken = await _tokenManager.GetValidAccessTokenAsync();
        var result = await Activities.UpdateActivityAsync(validToken, activityId,
name, description);
        return result != null;
    }
}
```

2. Factory Pattern

✅ **Do:**

```
public class StravaClientFactory
{
    private readonly IConfiguration _configuration;
    private readonly ILogger<StravaClientFactory> _logger;

    public StravaClientFactory(IConfiguration configuration, ILogger<StravaClientFactory>
logger)
    {
        _configuration = configuration;
        _logger = logger;
    }

    public async Task<IStravaRepository> CreateClientAsync()
    {
        var clientId = _configuration["Strava:ClientId"];
        var clientSecret = _configuration["Strava:ClientSecret"];

        if (string.IsNullOrEmpty(clientId) || string.IsNullOrEmpty(clientSecret))
        {
```

```
            throw new InvalidOperationException("Strava credentials not configured");
        }

        var credentials = new Credentials(clientId, clientSecret, "read,activity:read_all");
        var tokenManager = new TokenManager(credentials);

        // Try to load existing tokens
        var existingTokens = await LoadTokensAsync();
        if (existingTokens != null)
        {
            credentials.AccessToken = existingTokens.AccessToken;
            credentials.RefreshToken = existingTokens.RefreshToken;
            credentials.TokenExpiration = existingTokens.TokenExpiration;
        }

        return new StravaRepository(credentials.AccessToken, tokenManager);
    }
}
```

🧪 Testing Best Practices

1. Unit Testing

✅ **Do:**

```
[TestClass]
public class StravaClientTests
{
    private Mock<IHttpClientFactory> _mockHttpClientFactory;
    private StravaClient _client;

    [TestInitialize]
    public void Setup()
    {
        _mockHttpClientFactory = new Mock<IHttpClientFactory>();
        _client = new StravaClient("test_token", _mockHttpClientFactory.Object);
    }

    [TestMethod]
    public async Task GetActivitiesAsync_ValidToken_ReturnsActivities()
    {
        // Arrange
        var mockHttpClient = new Mock<HttpClient>();
        var mockResponse = new HttpResponseMessage(HttpStatusCode.OK)
        {
```

```
            Content = new StringContent("[{\"id\":123,\"name\":\"Test Activity\"}]")
        };

        mockHttpClient.Setup(x => x.GetAsync(It.IsAny<string>()))
            .ReturnsAsync(mockResponse);

        _mockHttpClientFactory.Setup(x => x.CreateClient(It.IsAny<string>()))
            .Returns(mockHttpClient.Object);

        // Act
        var result = await _client.GetActivitiesAsync();

        // Assert
        Assert.IsNotNull(result);
        Assert.AreEqual(1, result.Count);
        Assert.AreEqual("Test Activity", result[0]["name"]);
    }
}
```

2. Integration Testing

✅ **Do:**

```
[TestClass]
public class StravaIntegrationTests
{
    private string _testAccessToken;

    [TestInitialize]
    public async Task Setup()
    {
        // Use test credentials for integration tests
        _testAccessToken = await GetTestAccessTokenAsync();
    }

    [TestMethod]
    public async Task GetAthleteProfile_ValidToken_ReturnsProfile()
    {
        // Arrange & Act
        var profile = await Athletes.GetAuthenticatedAthleteProfileAsync(_testAccessToken);

        // Assert
        Assert.IsNotNull(profile);
        Assert.IsTrue(profile.ContainsKey("id"));
        Assert.IsTrue(profile.ContainsKey("firstname"));
```

```
        Assert.IsTrue(profile.ContainsKey("lastname"));
    }
}
```

📚 Documentation Best Practices

1. Code Documentation

✅ **Do:**

```
/// <summary>
/// Retrieves the authenticated athlete's activities with optional filtering and pagination.
/// </summary>
/// <param name="accessToken">The OAuth access token for authentication.</param>
/// <param name="page">Page number for pagination. Must be greater than 0.</param>
/// <param name="perPage">Number of activities per page. Must be between 1 and 200.</param>
/// <returns>A <see cref="JsonArray"/> containing the athlete's activities.</returns>
/// <exception cref="ArgumentException">Thrown when access token is invalid.</exception>
/// <exception cref="HttpRequestException">Thrown when the API request fails.</exception>
/// <remarks>
/// This method requires the <c>activity:read_all</c> scope.
/// Activities are returned in reverse chronological order.
/// </remarks>
/// <example>
/// <code>
/// var activities = await GetActivitiesAsync(accessToken, page: 1, perPage: 10);
/// </code>
/// </example>
public async Task<JsonArray> GetActivitiesAsync(string accessToken, int page = 1, int
perPage = 30)
{
    // Implementation
}
```

🚀 Deployment Best Practices

1. Configuration Management

✅ **Do:**

```
{
  "Strava": {
    "ClientId": "your_client_id",
    "ClientSecret": "your_client_secret",
    "RedirectUri": "https://yourapp.com/callback",
```

```
    "DefaultScope": "read,activity:read_all"
  },
  "Logging": {
    "LogLevel": {
      "StravaAPILibary": "Information"
    }
  }
}
```

2. Environment-Specific Settings

✅ **Do:**

```
public class StravaConfiguration
{
    public string ClientId { get; set; } = string.Empty;
    public string ClientSecret { get; set; } = string.Empty;
    public string RedirectUri { get; set; } = string.Empty;
    public string DefaultScope { get; set; } = "read,activity:read_all";
    public int RequestTimeoutSeconds { get; set; } = 30;
    public int MaxRetries { get; set; } = 3;
}

// In Startup.cs
services.Configure<StravaConfiguration>(configuration.GetSection("Strava"));
```

📈 Performance Monitoring

1. Health Checks

✅ **Do:**

```
public class StravaHealthCheck : IHealthCheck
{
    private readonly IStravaRepository _stravaRepository;

    public StravaHealthCheck(IStravaRepository stravaRepository)
    {
        _stravaRepository = stravaRepository;
    }

    public async Task<HealthCheckResult> CheckHealthAsync(HealthCheckContext context,
CancellationToken cancellationToken = default)
    {
        try
```

```
        {
            var profile = await _stravaRepository.GetAthleteProfileAsync();

            if (profile != null && profile.ContainsKey("id"))
            {
                return HealthCheckResult.Healthy("Strava API is accessible");
            }

            return HealthCheckResult.Unhealthy("Strava API returned invalid response");
        }
        catch (Exception ex)
        {
            return HealthCheckResult.Unhealthy("Strava API is not accessible", ex);
        }
    }
}
```

🔗 Next Steps

- **API Reference** - Complete API documentation
- **Authentication Guide** - OAuth flow and token management
- **Examples** - Practical usage examples

Follow these best practices to build robust, secure, and efficient Strava applications! 🚀

Namespace StravaAPILibary.API

Classes

Activities
: Provides methods to interact with Strava's Activities API.

Athletes
: Provides methods to interact with Strava's Athletes API.

Clubs
: Provides methods to interact with Strava's Clubs API.

Gears
: Provides methods to interact with Strava's Gear API.

Routes
: Provides methods to interact with Strava's Routes API.

Segments
: Provides methods for interacting with Strava's Segments API.

SegmentsEfforts
: Provides methods for interacting with Strava's Segment Efforts API.

Streams
: Provides methods for retrieving stream data (detailed time-series data) from Strava.

Uploads
: Provides methods for uploading activities and retrieving upload statuses from the Strava API.