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

Welcome to the StravaAPILibary! This guide will help you get up and running quickly with our .NET library for Strava API integration.

📦 Installation

Via NuGet Package Manager

```
dotnet add package StravaAPILibary
```

Via Package Manager Console

```
Install-Package StravaAPILibary
```

Via .csproj

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

🚀 Quick Start

1. Set Up Your Strava App

First, create a Strava application at Strava API Settings:

1. Go to your Strava account settings
2. Navigate to "API" section
3. Create a new application
4. Note your **Client ID** and **Client Secret**

2. Basic Authentication

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

// Initialize credentials
var credentials = new Credentials(
    clientId: "your_client_id",
    clientSecret: "your_client_secret",
    scopes: "read,activity:read_all"
);

// Create authentication instance
```

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

// Start the authorization process
userAuth.StartAuthorization();
```

3. Handle the Authorization Code

After the user authorizes your app, Strava will redirect to your callback URL with an authorization code:

```
// Exchange the authorization code for an access token
bool success = await userAuth.ExchangeCodeForTokenAsync("your_auth_code");

if (success)
{
    string accessToken = credentials.AccessToken;
    Console.WriteLine($"Access Token: {accessToken}");
}
```

4. Make API Calls

```
// Get athlete profile
var athlete = await Athletes.GetAuthenticatedAthleteProfileAsync(accessToken);
Console.WriteLine($"Athlete: {athlete.FirstName} {athlete.LastName}");

// Get recent activities
var activities = await Activities.GetAthletesActivitiesAsync(accessToken);
foreach (var activity in activities)
{
    Console.WriteLine($"Activity: {activity.Name} - {activity.Distance}m");
}
```

 Authentication Flow

The library supports the OAuth 2.0 authorization code flow:

1. **Authorization Request**: Redirect user to Strava for authorization
2. **Callback Handling**: Receive authorization code from Strava
3. **Token Exchange**: Exchange code for access token
4. **API Calls**: Use access token for API requests

📋 Prerequisites

- **.NET 6.0** or later
- **Strava Account** with API access
- **Strava Application** (Client ID and Secret)

🛠️ Configuration

Environment Variables

For production, store your credentials securely:

```
# .env file
STRAVA_CLIENT_ID=your_client_id
STRAVA_CLIENT_SECRET=your_client_secret
STRAVA_REDIRECT_URI=http://localhost:8080/callback
```

Configuration Class

```
public class StravaConfig
{
    public string ClientId { get; set; }
    public string ClientSecret { get; set; }
    public string RedirectUri { get; set; }
    public string Scopes { get; set; } = "read,activity:read_all";
}
```

🔧 Error Handling

The library provides comprehensive error handling:

```
try
{
    var activities = await Activities.GetAthletesActivitiesAsync(accessToken);
}
catch (StravaApiException ex)
{
    Console.WriteLine($"API Error: {ex.Message}");
    Console.WriteLine($"Status Code: {ex.StatusCode}");
}
catch (Exception ex)
{
```

```
    Console.WriteLine($"Unexpected error: {ex.Message}");
}
```

Next Steps

- **[Authentication Guide]({{ '/articles/authentication/' | relative_url }})** - Detailed OAuth flow
- **[Examples]({{ '/articles/examples/' | relative_url }})** - Real-world usage examples
- **[API Reference]({{ '/api/' | relative_url }})** - Complete API documentation

Need Help?

- Check the **[API Reference]({{ '/api/' | relative_url }})** for detailed method documentation
- Review **[Examples]({{ '/articles/examples/' | relative_url }})** for common use cases
- Visit the **GitHub Repository** for issues and contributions

---

Ready to build amazing Strava applications? Let's get started! 🚀

Authentication Guide

This guide covers the complete OAuth 2.0 authentication flow for the Strava API using StravaAPILibary.

🔐 OAuth 2.0 Overview

Strava uses OAuth 2.0 for secure API access. The library implements the **Authorization Code Flow**, which is the most secure method for web applications.

Flow Steps

1. **Authorization Request** - Redirect user to Strava
2. **User Authorization** - User grants permissions
3. **Authorization Code** - Strava returns code to your app
4. **Token Exchange** - Exchange code for access token
5. **API Access** - Use token for API calls

🚀 Basic Authentication

Step 1: Initialize Credentials

```
using StravaAPILibary.Authentication;

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

Step 2: Create Authentication Instance

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

Step 3: Start Authorization

```
// This opens the user's browser to Strava authorization page
userAuth.StartAuthorization();
```

Step 4: Handle the Callback

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

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

Step 5: Exchange Code for Token

```
// Extract the authorization code from the callback URL
string authCode = "YOUR_AUTH_CODE";

// Exchange the code for an access token
bool success = await userAuth.ExchangeCodeForTokenAsync(authCode);

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

    Console.WriteLine("✅ Authentication successful!");
    Console.WriteLine($"Access Token: {accessToken[..10]}...");
}
```

🔑 Token Management

Access Tokens

- **Lifetime**: 6 hours
- **Usage**: API requests
- **Storage**: Store securely (encrypted)

Refresh Tokens

- **Lifetime**: Indefinite (until revoked)
- **Usage**: Get new access tokens
- **Storage**: Store securely (encrypted)

Automatic Token Refresh

```
// The library automatically handles token refresh
if (credentials.IsTokenExpired())
{
    bool refreshed = await userAuth.RefreshTokenAsync();
    if (refreshed)
    {
        // Token automatically updated in credentials
```

```
        string newAccessToken = credentials.AccessToken;
    }
}
```

📋 Scopes

Request only the scopes your application needs:

Scope	Description	Access Level
`read`	Basic profile access	Public profile
`activity:read_all`	Read all activities	Private activities
`activity:write`	Upload activities	Create activities
`profile:read_all`	Detailed profile	Private profile data
`profile:write`	Update profile	Modify profile

Multiple Scopes

```
var scopes = "read,activity:read_all,activity:write";
var userAuth = new UserAuthentication(credentials, redirectUri, scopes);
```

🛡️ Security Best Practices

1. Secure Storage

```
// Store tokens encrypted
public class SecureTokenStorage
{
    public async Task SaveTokensAsync(string accessToken, string refreshToken)
    {
        var encryptedAccess = await EncryptAsync(accessToken);
        var encryptedRefresh = await EncryptAsync(refreshToken);

        await File.WriteAllTextAsync("tokens.json", JsonSerializer.Serialize(new
        {
            AccessToken = encryptedAccess,
            RefreshToken = encryptedRefresh,
            ExpiresAt = DateTime.UtcNow.AddHours(6)
        }));
```

```
    }
}
```

2. Environment Variables

```
# .env file
STRAVA_CLIENT_ID=your_client_id
STRAVA_CLIENT_SECRET=your_client_secret
STRAVA_REDIRECT_URI=http://localhost:8080/callback
```

3. HTTPS in Production

Always use HTTPS for production applications:

```
var redirectUri = Environment.GetEnvironmentVariable("ENVIRONMENT") == "Production"
    ? "https://yourapp.com/callback"
    : "http://localhost:8080/callback";
```

🔄 Token Refresh Strategy

Automatic Refresh

```
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.IsTokenExpired() || _credentials.WillExpireSoon())
        {
            bool refreshed = await _userAuth.RefreshTokenAsync();
            if (!refreshed)
            {
                throw new InvalidOperationException("Failed to refresh token");
            }
        }

        return _credentials.AccessToken;
    }
}
```

Manual Refresh

```
// Force token refresh
bool success = await userAuth.RefreshTokenAsync();

if (success)
{
    string newAccessToken = credentials.AccessToken;
    string newRefreshToken = credentials.RefreshToken;

    // Update stored tokens
    await UpdateStoredTokensAsync(newAccessToken, newRefreshToken);
}
```

🚨 Error Handling

Common Authentication Errors

```
try
{
    bool success = await userAuth.ExchangeCodeForTokenAsync(authCode);
}
catch (StravaApiException ex)
{
    switch (ex.StatusCode)
    {
        case 400:
            Console.WriteLine("Invalid authorization code or redirect URI");
            break;
        case 401:
            Console.WriteLine("Invalid client credentials");
            break;
        case 403:
            Console.WriteLine("Insufficient scopes");
            break;
        default:
            Console.WriteLine($"API Error: {ex.Message}");
            break;
    }
}
catch (Exception ex)
{
    Console.WriteLine($"Unexpected error: {ex.Message}");
}
```

Token Validation

```
public async Task<bool> ValidateTokenAsync(string accessToken)
{
    try
    {
        var athlete = await Athletes.GetAuthenticatedAthleteProfileAsync(accessToken);
        return athlete != null;
    }
    catch (StravaApiException ex) when (ex.StatusCode == 401)
    {
        return false; // Token is invalid
    }
}
```

🔧 Advanced Configuration

Custom HTTP Client

```
var httpClient = new HttpClient();
httpClient.Timeout = TimeSpan.FromSeconds(30);

var userAuth = new UserAuthentication(
    credentials: credentials,
    redirectUri: redirectUri,
    scopes: scopes,
    httpClient: httpClient
);
```

Custom Token Storage

```
public class CustomTokenStorage : ITokenStorage
{
    public async Task SaveTokensAsync(string accessToken, string refreshToken)
    {
        // Custom storage implementation
        await Database.SaveTokensAsync(accessToken, refreshToken);
    }

    public async Task<(string AccessToken, string RefreshToken)> LoadTokensAsync()
    {
        // Custom loading implementation
        return await Database.LoadTokensAsync();
```

```
    }
}
```

📱 Mobile Applications

For mobile apps, use a custom URL scheme:

```
// iOS/Android deep link
var redirectUri = "myapp://strava-callback";

var userAuth = new UserAuthentication(credentials, redirectUri, scopes);
```

🔍 Debugging Authentication

Enable Debug Logging

```
// Set up logging to see detailed authentication flow
var userAuth = new UserAuthentication(credentials, redirectUri, scopes);
userAuth.DebugMode = true; // Enable detailed logging
```

Common Issues

1. **Invalid Redirect URI**
    - Ensure redirect URI matches Strava app settings
    - Check for trailing slashes
2. **Expired Authorization Code**
    - Codes expire quickly (usually 10 minutes)
    - Request new authorization if code expires
3. **Scope Mismatch**
    - Ensure requested scopes match Strava app settings
    - Check for typos in scope names

📚 Related Resources

- **[Getting Started]({{ '/articles/getting-started/' | relative_url }})** - Basic setup guide
- **[Examples]({{ '/articles/examples/' | relative_url }})** - Authentication examples
- **[API Reference]({{ '/api/' | relative_url }})** - Complete API documentation
- [Strava API Docs] - Official documentation

Need help with authentication? Check our examples or create an issue on GitHub! 

Examples

This page contains practical examples and real-world scenarios showing how to use StravaAPILibary effectively.

🚀 Basic Examples

Get Athlete Profile

```
using StravaAPILibary.API;

// Get the authenticated athlete's profile
var athlete = await Athletes.GetAuthenticatedAthleteProfileAsync(accessToken);

Console.WriteLine($"Name: {athlete.FirstName} {athlete.LastName}");
Console.WriteLine($"Location: {athlete.City}, {athlete.Country}");
Console.WriteLine($"Followers: {athlete.FollowerCount}");
Console.WriteLine($"Following: {athlete.FriendCount}");
```

Get Recent Activities

```
// Get the last 10 activities
var activities = await Activities.GetAthletesActivitiesAsync(accessToken, perPage: 10);

foreach (var activity in activities)
{
    Console.WriteLine($"Activity: {activity.Name}");
    Console.WriteLine($"Type: {activity.Type}");
    Console.WriteLine($"Distance: {activity.Distance:F0}m");
    Console.WriteLine($"Duration: {TimeSpan.FromSeconds(activity.MovingTime)}");
    Console.WriteLine($"Date: {activity.StartDate}");
    Console.WriteLine("---");
}
```

Get Activity Details

```
// Get detailed information about a specific activity
long activityId = 123456789;
var activity = await Activities.GetActivityByIdAsync(accessToken, activityId);

Console.WriteLine($"Activity: {activity.Name}");
Console.WriteLine($"Description: {activity.Description}");
Console.WriteLine($"Distance: {activity.Distance:F0}m");
Console.WriteLine($"Elevation Gain: {activity.TotalElevationGain:F0}m");
```

```
Console.WriteLine($"Average Speed: {activity.AverageSpeed:F2}m/s");
Console.WriteLine($"Max Speed: {activity.MaxSpeed:F2}m/s");
```

📊 Data Analysis Examples

Calculate Weekly Distance

```
public async Task<double> GetWeeklyDistanceAsync(string accessToken)
{
    // Get activities from the last 7 days
    int after = (int)DateTimeOffset.UtcNow.AddDays(-7).ToUnixTimeSeconds();
    var activities = await Activities.GetAthletesActivitiesAsync(accessToken, after: after);

    double totalDistance = activities.Sum(a => a.Distance);
    return totalDistance; // Returns distance in meters
}

// Usage
double weeklyDistance = await GetWeeklyDistanceAsync(accessToken);
Console.WriteLine($"Weekly Distance: {weeklyDistance / 1000:F1}km");
```

Find Personal Records

```
public async Task<List<SummaryActivity>> FindPersonalRecordsAsync(string accessToken)
{
    var activities = await Activities.GetAthletesActivitiesAsync(accessToken, perPage: 200);

    var personalRecords = new List<SummaryActivity>();

    // Find fastest 5k
    var fastest5k = activities
        .Where(a => a.Distance >= 5000 && a.Distance <= 5100)
        .OrderBy(a => a.MovingTime)
        .FirstOrDefault();

    if (fastest5k != null)
    {
        personalRecords.Add(fastest5k);
    }

    // Find longest distance
    var longestDistance = activities
        .OrderByDescending(a => a.Distance)
        .FirstOrDefault();
```

```
    if (longestDistance != null)
    {
        personalRecords.Add(longestDistance);
    }

    return personalRecords;
}
```

Activity Statistics

```
public class ActivityStats
{
    public int TotalActivities { get; set; }
    public double TotalDistance { get; set; }
    public TimeSpan TotalTime { get; set; }
    public double AverageDistance { get; set; }
    public Dictionary<string, int> ActivityTypes { get; set; } = new();
}

public async Task<ActivityStats> GetActivityStatsAsync(string accessToken, int days = 30)
{
    int after = (int)DateTimeOffset.UtcNow.AddDays(-days).ToUnixTimeSeconds();
    var activities = await Activities.GetAthletesActivitiesAsync(accessToken, after: after);

    var stats = new ActivityStats
    {
        TotalActivities = activities.Count,
        TotalDistance = activities.Sum(a => a.Distance),
        TotalTime = TimeSpan.FromSeconds(activities.Sum(a => a.MovingTime))
    };

    stats.AverageDistance = stats.TotalDistance / stats.TotalActivities;

    // Count activity types
    foreach (var activity in activities)
    {
        if (stats.ActivityTypes.ContainsKey(activity.Type))
        {
            stats.ActivityTypes[activity.Type]++;
        }
        else
        {
            stats.ActivityTypes[activity.Type] = 1;
        }
```

```
    }

    return stats;
}
```

🗺️ Route and Segment Examples

Get Route Information

```
public async Task<Route> GetRouteDetailsAsync(string accessToken, long routeId)
{
    var route = await Routes.GetRouteByIdAsync(accessToken, routeId);

    Console.WriteLine($"Route: {route.Name}");
    Console.WriteLine($"Distance: {route.Distance:F0}m");
    Console.WriteLine($"Elevation Gain: {route.ElevationGain:F0}m");
    Console.WriteLine($"Type: {route.Type}");
    Console.WriteLine($"Sub Type: {route.SubType}");

    return route;
}
```

Find Segments Near Location

```
public async Task<List<ExplorerSegment>> FindSegmentsNearbyAsync(
    string accessToken,
    double latitude,
    double longitude,
    double radius = 1000)
{
    var segments = await Segments.ExploreSegmentsAsync(
        accessToken,
        bounds: $"{latitude},{longitude},{latitude},{longitude}",
        activityType: "running",
        minCat: 0,
        maxCat: 5
    );

    return segments.Segments
        .Where(s => s.Distance <= radius)
        .OrderBy(s => s.Distance)
        .ToList();
}
```

👥 Social Features

Get Athlete's Followers

```
public async Task<List<SummaryAthlete>> GetFollowersAsync(string accessToken)
{
    var followers = await Athletes.GetAuthenticatedAthleteFollowersAsync(accessToken);

    foreach (var follower in followers)
    {
        Console.WriteLine($"Follower: {follower.FirstName} {follower.LastName}");
        Console.WriteLine($"Username: {follower.Username}");
        Console.WriteLine($"Location: {follower.City}");
        Console.WriteLine("---");
    }

    return followers;
}
```

Get Club Activities

```
public async Task<List<ClubActivity>> GetClubActivitiesAsync(string accessToken,
long clubId)
{
    var activities = await Clubs.GetClubActivitiesByIdAsync(accessToken, clubId);

    foreach (var activity in activities)
    {
        Console.WriteLine($"Athlete: {activity.Athlete.FirstName}
{activity.Athlete.LastName}");
        Console.WriteLine($"Activity: {activity.Name}");
        Console.WriteLine($"Distance: {activity.Distance:F0}m");
        Console.WriteLine($"Type: {activity.Type}");
        Console.WriteLine("---");
    }

    return activities;
}
```

📈 Performance Tracking

Track Progress Over Time

```
public class PerformanceTracker
{
    public async Task<Dictionary<string, List<double>>> TrackProgressAsync(
        string accessToken,
        int weeks = 12)
    {
        var weeklyDistances = new Dictionary<string, List<double>>();

        for (int week = 0; week < weeks; week++)
        {
            var startDate = DateTimeOffset.UtcNow.AddDays(-7 * (week + 1));
            var endDate = startDate.AddDays(7);

            int after = (int)startDate.ToUnixTimeSeconds();
            int before = (int)endDate.ToUnixTimeSeconds();

            var activities = await Activities.GetAthletesActivitiesAsync(
                accessToken,
                after: after,
                before: before
            );

            var weekDistance = activities.Sum(a => a.Distance) / 1000; // Convert to km
            var weekKey = startDate.ToString("yyyy-MM-dd");

            weeklyDistances[weekKey] = new List<double> { weekDistance };
        }

        return weeklyDistances;
    }
}
```

Compare Performance

```
public async Task<PerformanceComparison> ComparePerformanceAsync(
    string accessToken,
    DateTime startDate,
    DateTime endDate)
{
    int after = (int)startDate.ToUnixTimeSeconds();
    int before = (int)endDate.ToUnixTimeSeconds();

    var activities = await Activities.GetAthletesActivitiesAsync(
        accessToken,
        after: after,
```

```
        before: before
    );

    var comparison = new PerformanceComparison
    {
        TotalActivities = activities.Count,
        TotalDistance = activities.Sum(a => a.Distance),
        TotalTime = TimeSpan.FromSeconds(activities.Sum(a => a.MovingTime)),
        AverageSpeed = activities.Average(a => a.AverageSpeed),
        MaxSpeed = activities.Max(a => a.MaxSpeed),
        TotalElevationGain = activities.Sum(a => a.TotalElevationGain)
    };

    return comparison;
}
```

🔧 Error Handling Examples

Robust API Calls

```
public async Task<T?> SafeApiCallAsync<T>(Func<Task<T>> apiCall, int maxRetries = 3)
{
    for (int attempt = 1; attempt <= maxRetries; attempt++)
    {
        try
        {
            return await apiCall();
        }
        catch (StravaApiException ex) when (ex.StatusCode == 429)
        {
            // Rate limit exceeded
            if (attempt < maxRetries)
            {
                int delaySeconds = attempt * 2; // Exponential backoff
                await Task.Delay(delaySeconds * 1000);
                continue;
            }
            throw;
        }
        catch (StravaApiException ex) when (ex.StatusCode == 401)
        {
            // Token expired, try to refresh
            if (attempt < maxRetries)
            {
                bool refreshed = await RefreshTokenAsync();
```

```
                if (refreshed)
                {
                    continue;
                }
            }
            throw;
        }
        catch (HttpRequestException ex)
        {
            // Network error
            if (attempt < maxRetries)
            {
                await Task.Delay(1000 * attempt);
                continue;
            }
            throw;
        }
    }

    throw new InvalidOperationException("Max retries exceeded");
}

// Usage
var activities = await SafeApiCallAsync(() =>
    Activities.GetAthletesActivitiesAsync(accessToken)
);
```

Batch Processing

```
public async Task ProcessActivitiesInBatchesAsync(
    string accessToken,
    Func<SummaryActivity, Task> processor,
    int batchSize = 50)
{
    int page = 1;
    bool hasMoreActivities = true;

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

```
            );

            if (activities.Count == 0)
            {
                hasMoreActivities = false;
                break;
            }

            // Process activities in parallel
            var tasks = activities.Select(processor);
            await Task.WhenAll(tasks);

            page++;
        }
        catch (StravaApiException ex)
        {
            Console.WriteLine($"Error processing batch {page}: {ex.Message}");
            page++; // Skip this batch and continue
        }
    }
}

// Usage
await ProcessActivitiesInBatchesAsync(accessToken, async activity =>
{
    Console.WriteLine($"Processing: {activity.Name}");
    // Your processing logic here
    await Task.Delay(100); // Rate limiting
});
```

🎯 Real-World Applications

Fitness Dashboard

```
public class FitnessDashboard
{
    public async Task<DashboardData> GetDashboardDataAsync(string accessToken)
    {
        var dashboard = new DashboardData();

        // Get recent activities
        var recentActivities = await Activities.GetAthletesActivitiesAsync(
            accessToken,
            perPage: 10
        );
```

```
        dashboard.RecentActivities = recentActivities;

        // Calculate weekly stats
        var weeklyStats = await GetActivityStatsAsync(accessToken, 7);
        dashboard.WeeklyStats = weeklyStats;

        // Get personal records
        var personalRecords = await FindPersonalRecordsAsync(accessToken);
        dashboard.PersonalRecords = personalRecords;

        return dashboard;
    }
}
```

Training Plan Generator

```
public class TrainingPlanGenerator
{
    public async Task<TrainingPlan> GeneratePlanAsync(
        string accessToken,
        string goal,
        int weeks)
    {
        var plan = new TrainingPlan { Goal = goal, DurationWeeks = weeks };

        // Analyze current fitness level
        var recentStats = await GetActivityStatsAsync(accessToken, 30);

        // Generate weekly targets based on goal and current fitness
        for (int week = 1; week <= weeks; week++)
        {
            var weeklyTarget = new WeeklyTarget
            {
                Week = week,
                TargetDistance = CalculateTargetDistance(recentStats, goal, week),
                TargetTime = CalculateTargetTime(recentStats, goal, week),
                ActivityTypes = GetRecommendedActivities(goal, week)
            };

            plan.WeeklyTargets.Add(weeklyTarget);
        }

        return plan;
```

```
    }
}
```

📚 More Examples

- **[API Reference]({{ '/api/' | relative_url }})** - Complete method documentation
- **[Authentication Guide]({{ '/articles/authentication/' | relative_url }})** - OAuth examples
- **[Getting Started]({{ '/articles/getting-started/' | relative_url }})** - Basic setup

---

Have a specific use case? Check our API reference or create an issue on GitHub! 💡

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.