DynaFetch Best Practices Guide

Guidelines for efficient, secure, and maintainable API integration using DynaFetch.

Client Management Best Practices

Create Once, Use Everywhere

Do:

```
// Create client at start of workflow
client = ClientNodes.Create()

// Use same client for multiple requests
response1 = ExecuteNodes.GET(client, url1)
response2 = ExecuteNodes.GET(client, url2)
response3 = ExecuteNodes.POST(client, url3, data)
```

Don't:

```
// Creating new client for each request (inefficient)
client1 = ClientNodes.Create()
response1 = ExecuteNodes.GET(client1, url1)

client2 = ClientNodes.Create()  // Wasteful
response2 = ExecuteNodes.GET(client2, url2)
```

Why: HTTP clients have overhead. Reusing clients improves performance and maintains connection pooling.

Use Base URLs for API-Centric Workflows

Do:

```
client = ClientNodes.CreateWithBaseUrl("https://api.github.com")
users = ExecuteNodes.GET(client, "/users/octocat")
repos = ExecuteNodes.GET(client, "/users/octocat/repos")
issues = ExecuteNodes.GET(client, "/repos/owner/repo/issues")
```

Don't:

```
client = ClientNodes.Create()
users = ExecuteNodes.GET(client, "https://api.github.com/users/octocat")
repos = ExecuteNodes.GET(client, "https://api.github.com/users/octocat/repos")
issues = ExecuteNodes.GET(client, "https://api.github.com/repos/owner/repo/issues")
```

Why: Base URLs reduce repetition and make endpoint management easier.

Set Appropriate Timeouts

Do:

```
// For fast APIs
client = ClientNodes.Create()
ClientNodes.SetTimeout(client, 30)  // 30 seconds

// For slow APIs or large data
client = ClientNodes.Create()
ClientNodes.SetTimeout(client, 300)  // 5 minutes
```

Consider:

- **Fast APIs**: 15-30 seconds
- **Standard APIs**: 60-120 seconds
- **Large data transfers**: 300+ seconds
- **Real-time APIs**: 5-15 seconds

Authentication Best Practices

Use Client-Level Authentication

Do:

```
client = ClientNodes.Create()
ClientNodes.AddDefaultHeader(client, "Authorization", "Bearer " + token)

// Auth automatically included in all requests
response1 = ExecuteNodes.GET(client, url1)
response2 = ExecuteNodes.POST(client, url2, data)
```

Don't:

```
// Adding auth to each request individually (tedious)
request1 = RequestNodes.ByUrl(url1)
request1 = RequestNodes.AddBearerToken(request1, token)
response1 = ExecuteNodes.Execute(client, request1)

request2 = RequestNodes.ByUrl(url2)
request2 = RequestNodes.AddBearerToken(request2, token)  // Repetitive
```

Why: Client-level auth is more maintainable and matches real-world usage patterns.

Authentication Patterns by API Type

REST APIs with Bearer Tokens (OAuth/JWT)

```
ClientNodes.AddDefaultHeader(client, "Authorization", "Bearer " + access_token)
```

APIs with API Keys

```
// Header-based API key
ClientNodes.AddDefaultHeader(client, "X-API-Key", api_key)

// Or custom header name
ClientNodes.AddDefaultHeader(client, "Api-Token", api_key)
```

Service Account Authentication (JWT Assertions)

For Autodesk APS, Google Service Accounts, or RFC 7523 flows:

```
// Generate JWT assertion
jwt = ClientNodes.GenerateJwtAssertion(
    privateKeyPem,
    clientId,
    "https://developer.api.autodesk.com/",
    ["data:read", "data:write"],
    60
)

// Exchange for access token
tokenBody = "grant_type=urn:ietf:params:oauth:grant-type:jwt-bearer&assertion=" +
jwt
client = ClientNodes.Create()
```

```
tokenResponse = ExecuteNodes.POST(client, tokenEndpoint, tokenBody)

// Extract and use token
tokenDict = JsonNodes.ToDictionary(tokenResponse)
accessToken = Dictionary.ValueAtKey(tokenDict, "access_token")
ClientNodes.AddDefaultHeader(client, "Authorization", "Bearer " + accessToken)
```

Best practices for JWT authentication:

- **Store private keys securely**: Never commit keys to version control
- **Use shortest expiration**: Default 60 minutes max, use less if possible
- **Cache tokens**: Don't regenerate JWT for every request - tokens are valid for up to 60 minutes
- **Implement refresh logic**: Exchange new JWT when token expires
- **Key format**: Supports both PKCS#1 and PKCS#8 PEM formats

Token caching pattern:

```
// Check if token is still valid before regenerating
if current_time < token_expiry_time:
    // Reuse existing token
    ClientNodes.AddDefaultHeader(client, "Authorization", "Bearer " + cached_token)
else:
    // Generate new JWT and exchange for fresh token
    new_jwt = ClientNodes.GenerateJwtAssertion(privateKeyPem, clientId, audience,
scopes, 60)
    // Exchange and cache new token
```

Secure Token Management

Do:

- Store tokens in Dynamo's secure parameter system
- Use environment variables for sensitive data
- Implement token refresh workflows for expiring tokens
- Remove tokens from clients when no longer needed

Don't:

- Hard-code tokens in Dynamo graphs
- Store tokens in plain text files
- Share graphs with embedded tokens

JSON Processing Best Practices

Always Check Response Success

Do:

```
response = ExecuteNodes.GET(client, url)

if response.IsSuccessful:
    data = JsonNodes.ToDictionary(response)
    // Process data
else:
    error = response.ErrorMessage
    // Handle error appropriately
```

Don't:

```
response = ExecuteNodes.GET(client, url)
data = JsonNodes.ToDictionary(response)  // May fail if response is error
```

Validate JSON Before Processing

Do:

```
response = ExecuteNodes.GET(client, url)

if response.IsSuccessful:
    if JsonNodes.IsValid(response):
        data = JsonNodes.ToDictionary(response)
        // Safe to process
    else:
        raw_content = JsonNodes.GetContent(response)
        // Handle non-JSON response
```

Use Appropriate JSON Methods

For Objects:

```
user_data = JsonNodes.ToDictionary(response)
name = user_data["name"]
email = user_data["email"]
```

For Arrays:

```
posts_list = JsonNodes.ToList(response)
first_post = posts_list[0]
post_count = List.Count(posts_list)
```

For Debugging:

```
pretty_json = JsonNodes.Format(response)
// Use for human-readable JSON display
```

For Raw Content:

```
raw_content = JsonNodes.GetContent(response)
// Use when you need the original response string
```

Safe JSON Processing

Use `Try` methods when JSON format is uncertain:

```
result = JsonNodes.TryToDictionary(response)
if result != null:
    // Safe to use result
else:
    // Response wasn't a valid JSON object
```

Error Handling Best Practices

Comprehensive Error Checking

Do:

```
response = ExecuteNodes.GET(client, url)

if response.IsSuccessful:
    if JsonNodes.IsValid(response):
        data = JsonNodes.ToDictionary(response)
        // Success path
    else:
        // API returned non-JSON (maybe HTML error page)
        content = JsonNodes.GetContent(response)
        // Log or display content for debugging
```

```
else:
    // HTTP error (404, 500, etc.)
    status = response.StatusCode
    error = response.ErrorMessage
    // Handle based on status code
```

HTTP Status Code Handling

```
if response.StatusCode == 401:
    // Authentication failed - check tokens
elif response.StatusCode == 403:
    // Permission denied - check API permissions
elif response.StatusCode == 404:
    // Resource not found - check URL
elif response.StatusCode == 429:
    // Rate limited - wait and retry
elif response.StatusCode >= 500:
    // Server error - retry or alert
```

Network Error Handling

```
try:
    response = ExecuteNodes.GET(client, url)
catch:
    // Network errors, timeouts, DNS failures
    // Check internet connectivity
    // Verify API endpoint is correct
    // Consider increasing timeout
```

Performance Optimization

Minimize API Calls

Do:

```
// Single call for multiple users
users = ExecuteNodes.GET(client, "/users?ids=1,2,3,4,5")
```

Don't:

```
// Multiple calls for individual users (inefficient)
user1 = ExecuteNodes.GET(client, "/users/1")
user2 = ExecuteNodes.GET(client, "/users/2")
user3 = ExecuteNodes.GET(client, "/users/3")
```

Batch Operations When Possible

Do:

```
// Batch create multiple records
batch_data = JsonNodes.DictionaryToJson(multiple_records)
response = ExecuteNodes.POST(client, "/batch-create", batch_data)
```

Use Appropriate Data Structures

For Large Datasets:

- Use `JsonNodes.ToList()` for arrays
- Process in chunks if memory is a concern
- Consider pagination for very large datasets

For Complex Objects:

- Use `JsonNodes.ToDictionary()` for structured data
- Access nested properties with dictionary keys

Efficient JSON Processing

System.Text.Json (Primary):

- DynaFetch automatically uses the fastest JSON engine
- Falls back to Newtonsoft.Json if needed
- No action required from users

Data Management Best Practices

Structuring API Data for Dynamo

Do:

```
// Convert to Dynamo-native types
api_response = JsonNodes.ToDictionary(response)
names = api_response["users"] |> List.Map(user => user["name"])
emails = api_response["users"] |> List.Map(user => user["email"])
```

For Nested Data:

```
response_dict = JsonNodes.ToDictionary(response)
user_data = response_dict["user"]
address_data = user_data["address"]
street = address_data["street"]
```

Preparing Data for API Submission

Do:

```
// Structure data properly
data_dict = Dictionary.ByKeysValues(
    ["name", "email", "age"],
    ["John Doe", "john@example.com", 30]
)
json_string = JsonNodes.DictionaryToJson(data_dict)
response = ExecuteNodes.POST(client, url, json_string)
```

Data Validation Before Submission

Do:

```
// Validate required fields
if data_dict["email"] != null && data_dict["name"] != null:
    json_string = JsonNodes.DictionaryToJson(data_dict)
    response = ExecuteNodes.POST(client, url, json_string)
else:
    // Handle missing required data
```

Security Best Practices

API Key Protection

Do:

- Use Dynamo's parameter system for sensitive data
- Implement key rotation workflows
- Monitor API usage for unauthorized access
- Use environment-specific keys (dev/staging/prod)

Don't:

- Embed API keys directly in graphs
- Share graphs with credentials
- Use production keys in development

Authentication Token Management

For Expiring Tokens:

```
// Check token expiration before use
if token_expires_at > current_time:
    ClientNodes.AddDefaultHeader(client, "Authorization", "Bearer " + token)
else:
    // Refresh token first
    new_token = refresh_token_workflow()
    ClientNodes.AddDefaultHeader(client, "Authorization", "Bearer " + new_token)
```

HTTPS Enforcement

Do:

- Always use HTTPS URLs for production APIs
- Validate SSL certificates (DynaFetch does this automatically)
- Use secure headers when required

```
// Good
url = "https://api.secure-service.com/data"

// Bad for production
url = "http://api.insecure-service.com/data"
```

Workflow Organization Best Practices

Logical Grouping

Group related operations:

```
// Authentication setup
client = ClientNodes.Create()
ClientNodes.AddDefaultHeader(client, "Authorization", "Bearer " + token)

// Data retrieval
users = ExecuteNodes.GET(client, "/users")
```

```
user_details = ExecuteNodes.GET(client, "/users/" + user_id)

// Data processing
users_dict = JsonNodes.ToDictionary(users)
details_dict = JsonNodes.ToDictionary(user_details)
```

Reusable Patterns

Create custom nodes for repeated patterns:

```
// Custom node: "Authenticated GET"
def AuthenticatedGET(token, base_url, endpoint):
    client = ClientNodes.CreateWithBaseUrl(base_url)
    ClientNodes.AddDefaultHeader(client, "Authorization", "Bearer " + token)
    response = ExecuteNodes.GET(client, endpoint)
    return JsonNodes.ToDictionary(response)
```

Documentation in Graphs

Do:

- Add notes explaining API purpose and authentication
- Document expected response formats
- Include example URLs and data structures
- Note any rate limits or special requirements

Testing and Debugging Best Practices

Development vs Production

Development:

```
// Use test endpoints
client = ClientNodes.CreateWithBaseUrl("https://api-staging.example.com")
ClientNodes.SetTimeout(client, 30)  // Shorter timeout for testing
```

Production:

```
// Use production endpoints
client = ClientNodes.CreateWithBaseUrl("https://api.example.com")
ClientNodes.SetTimeout(client, 120)  // Longer timeout for reliability
```

Response Debugging

Use Format for readable JSON:

```
response = ExecuteNodes.GET(client, url)
pretty_json = JsonNodes.Format(response)
// Display pretty_json for debugging
```

Check raw content for errors:

```
if not response.IsSuccessful:
    raw_content = JsonNodes.GetContent(response)
    // Examine raw_content to understand error
```

Performance Monitoring

Track response times:

- Monitor for degraded API performance
- Set timeout appropriately based on typical response times
- Consider caching for frequently accessed data

Rate Limiting Best Practices

Respect API Rate Limits

Common patterns:

- Check API documentation for rate limits
- Implement delays between requests if needed
- Handle 429 status codes gracefully
- Use batch operations when available

Rate limit handling:

```
response = ExecuteNodes.GET(client, url)

if response.StatusCode == 429:
    // Rate limited - wait and retry
    // Check "Retry-After" header if available
    wait_time = response.Headers["Retry-After"] or 60
    // Implement wait logic
```

Efficient API Usage

Do:

- Cache responses when appropriate
- Use pagination for large datasets
- Request only needed fields if API supports field selection
- Use conditional requests (ETags) when supported

Error Recovery Strategies

Retry Logic

For transient errors:

```
max_retries = 3
for attempt in range(max_retries):
    response = ExecuteNodes.GET(client, url)
    if response.IsSuccessful:
        break
    elif response.StatusCode >= 500:
        // Server error - retry
        wait_time = 2^attempt  // Exponential backoff
        // Wait and retry
    else:
        // Client error - don't retry
        break
```

Graceful Degradation

Have fallback strategies:

- Default values for missing data
- Alternative API endpoints
- Cached data for offline scenarios
- User-friendly error messages

Maintenance Best Practices

API Version Management

Track API versions:

```
ClientNodes.AddDefaultHeader(client, "Accept", "application/vnd.api+json;version=2")
```

Monitoring API Changes

- Subscribe to API change notifications
- Test workflows against API updates
- Have rollback plans for breaking changes
- Document API dependencies

File Upload Best Practices

Choose the Right Upload Method

Method 1: AddFile (Recommended for most cases)

Use when: You need method chaining, custom headers, or authentication

```
client = ClientNodes.Create()
ClientNodes.AddDefaultHeader(client, "Authorization", "Bearer " + token)

request = RequestNodes.ByUrl(uploadUrl)
request = RequestNodes.AddFile(request, "file", filePath, "image/png")
response = ExecuteNodes.POST(client, "", request)
```

Advantages:

- Supports method chaining with headers
- DynaWeb-compatible pattern
- Flexible for complex uploads

Method 2: CreateFileUpload (For simple uploads)

Use when: You only need basic file upload without additional configuration

```
formData = RequestNodes.CreateFileUpload(filePath, "file", "photo.jpg",
"image/jpeg")
response = ExecuteNodes.POST(client, uploadUrl, formData)
```

Advantages:

- Simpler for basic uploads
- Direct multipart content creation

File Upload with Metadata

Do:

```
// Create file upload
formData = RequestNodes.CreateFileUpload(filePath, "file")

// Add metadata fields
formData = RequestNodes.AddFormField(formData, "description", "Project
documentation")
formData = RequestNodes.AddFormField(formData, "category", "Construction")
formData = RequestNodes.AddFormField(formData, "project_id", projectId)

// Execute upload
response = ExecuteNodes.POST(client, uploadUrl, formData)
```

Content-Type Handling

Auto-detection (recommended):

```
// Let DynaFetch detect content type from file extension
request = RequestNodes.AddFile(request, "file", "C:\\Photos\\image.jpg")
// Automatically detects "image/jpeg"
```

Manual specification (when needed):

```
// Specify content type explicitly
request = RequestNodes.AddFile(request, "file", filePath, "application/pdf")
```

Supported auto-detected types:

- Images: jpg, png, gif, bmp, webp, svg, ico
- Documents: pdf, doc, docx, xls, xlsx, ppt, pptx
- Text: txt, csv, html, xml, json
- Archives: zip, rar, 7z

File Upload Error Handling

Do:

```
// Validate file exists before upload
if File.Exists(filePath):
    request = RequestNodes.AddFile(request, "file", filePath)
    response = ExecuteNodes.POST(client, uploadUrl, request)

    if response.IsSuccessful:
        // Upload succeeded
        result = JsonNodes.ToDictionary(response)
    else:
        // Check common upload errors
        if response.StatusCode == 413:
            // File too large
        elif response.StatusCode == 415:
            // Unsupported file type
        else:
            error = response.ErrorMessage
else:
    // File not found - handle error
```

Large File Uploads

Set appropriate timeout:

```
client = ClientNodes.Create()
ClientNodes.SetTimeout(client, 600)  // 10 minutes for large files
```

Consider file size limits:

- Check API documentation for max file size
- Handle 413 "Payload Too Large" errors
- Consider chunked uploads for very large files (if API supports)

Secure File Upload Practices

Do:

- Validate file paths before upload
- Check file extensions match expected types
- Use appropriate authentication for uploads
- Verify upload success before proceeding

Don't:

- Upload files without validation
- Ignore upload response errors
- Use hardcoded file paths in production graphs

Custom Content-Type Best Practices

When to Use Custom Content-Types

Use `RequestNodes.AddTextContent` for:

- RFC 7396 JSON Merge Patch (`application/merge-patch+json`)
- RFC 6902 JSON Patch (`application/json-patch+json`)
- API-specific content types
- Any non-standard JSON content type

Do:

```
client = ClientNodes.Create()
ClientNodes.AddDefaultHeader(client, "Authorization", "Basic " + credentials)

request = RequestNodes.ByUrl(url)
request = RequestNodes.AddTextContent(request, jsonData, "application/merge-patch+json")
response = ExecuteNodes.PATCH(client, "", request)
```

Don't:

```
// WRONG - AddJsonBody overrides custom Content-Type
request = RequestNodes.AddHeader(request, "Content-Type", "application/merge-patch+json")
request = RequestNodes.AddJsonBody(request, jsonData)  // Resets to application/json
```

RFC 7396 Merge Patch Best Practices

Prepare data correctly:

```
// Remove ID from body (it's in the URL)
updateData = Dictionary.RemoveKeys(originalData, ["id"])

// Only include fields to update
updateData = Dictionary.ByKeysValues(["name", "status"], ["New Name", "active"])

// Convert to JSON
jsonString = JsonNodes.DictionaryToJson(updateData)

// Use AddTextContent with merge-patch content type
request = RequestNodes.AddTextContent(request, jsonString, "application/merge-patch+json")
```

Common mistakes to avoid:

- Including resource ID in PATCH body (it's already in URL)
- Using wrong Content-Type (causes 415 errors)

- Using AddJsonBody instead of AddTextContent
- Sending full object instead of partial update

Following these best practices will help you build robust, maintainable, and secure API integrations with DynaFetch. Focus on security, error handling, and performance to create reliable workflows that handle real-world scenarios effectively.