DynaFetch Advanced Node Library Reference

Complete reference for ALL DynaFetch nodes including Core infrastructure and System utilities

Overview

This document covers ALL nodes available in DynaFetch, including the Core infrastructure classes and System utilities that are typically used internally. For everyday workflow building, see the Node Library Reference which covers the primary workflow nodes.

Library Organization

```
DynaFetch/
├── Core/                    # HTTP infrastructure classes (100+ individual nodes)
│   ├── DynaFetchException/
│   │   ├── DynaFetchException (constructor)
│   │   ├── Context
│   │   └── Operation
│   ├── DynaFetchHttpException/
│   │   ├── DynaFetchHttpException (constructor)
│   │   ├── FromResponse
│   │   ├── RequestUrl
│   │   ├── ResponseContent
│   │   └── StatusCode
│   ├── DynaFetchJsonException/
│   │   ├── DynaFetchJsonException (constructor)
│   │   └── JsonContent
│   ├── ErrorDetails/
│   │   ├── ErrorDetails (constructor)
│   │   ├── Content
│   │   ├── RequestUrl
│   │   ├── StatusCode
│   │   └── StatusDescription
│   ├── HttpClientWrapper/
│   │   ├── HttpClientWrapper() (constructors)
│   │   ├── HttpClientWrapper(baseUrl)
│   │   ├── DeleteAsync
│   │   ├── GetAsync
│   │   ├── GetDefaultHeaders
│   │   ├── PatchAsync
│   │   ├── PostAsync
│   │   ├── PutAsync
│   │   ├── RemoveDefaultHeader
│   │   ├── SetDefaultHeader
│   │   ├── SetDefaultHeaders
│   │   ├── BaseUrl
```

```
│   │   ├── TimeoutSeconds
│   │   └── UserAgent
│   ├── HttpRequest/
│   │   ├── Create(url)
│   │   ├── Create(baseUrl, endpoint)
│   │   ├── AddBearerToken
│   │   ├── AddContent
│   │   ├── AddHeader
│   │   ├── AddHeaders
│   │   ├── AddJsonBody
│   │   ├── AddJsonContent
│   │   ├── AddParameter
│   │   ├── AddParameters
│   │   ├── AddTextContent
│   │   ├── AsDelete
│   │   ├── AsGet
│   │   ├── AsPatch
│   │   ├── AsPost
│   │   ├── AsPut
│   │   ├── RemoveHeader
│   │   ├── RemoveParameter
│   │   ├── SetMethod
│   │   ├── ToHttpRequestMessage
│   │   ├── Content
│   │   ├── Headers
│   │   ├── HttpMethod
│   │   ├── Parameters
│   │   └── Uri
│   ├── HttpResponse/
│   │   ├── HttpResponse (constructor)
│   │   ├── EnsureSuccessAsync
│   │   ├── GetBytesAsync
│   │   ├── GetContent
│   │   ├── GetContentAsync
│   │   ├── GetDetailedInfoAsync
│   │   ├── GetErrorDetailsAsync
│   │   ├── GetFormattedJsonAsync
│   │   ├── GetHeader
│   │   ├── GetJsonAsDictionaryAsync
│   │   ├── GetJsonAsListAsync
│   │   ├── GetJsonAsObjectAsync
│   │   ├── GetJsonDocumentAsync
│   │   ├── GetStreamAsync
│   │   ├── HasHeader
│   │   ├── IsJsonAsync
│   │   ├── IsValidJsonAsync
│   │   ├── TryGetJsonAsDictionaryAsync
│   │   ├── TryGetJsonAsListAsync
│   │   ├── ContentLength
│   │   ├── ContentType
```

```
│   │   ├── Headers
│   │   ├── IsError
│   │   ├── IsSuccess
│   │   ├── RequestUri
│   │   ├── StatusCode
│   │   ├── StatusCodeNumber
│   │   └── StatusDescription
│   ├── SafeOperations/
│   │   ├── TryParseUrl
│   │   └── ValidateJson
│   └── Validation/
│       ├── ValidateHeaderName
│       ├── ValidateHttpClient
│       ├── ValidateHttpMethod
│       ├── ValidateJsonContent
│       ├── ValidateNotEmpty
│       ├── ValidateNotNull
│       ├── ValidateTimeout
│       └── ValidateUrl
├── Nodes/                  # Primary workflow nodes ⭐ (detailed in Node-
Library.md)
│   ├── ClientNodes/        # HTTP client management (11 methods)
│   │   ├── AddDefaultHeader
│   │   ├── AddDefaultHeaders
│   │   ├── Create
│   │   ├── CreateWithBaseUrl
│   │   ├── CreateWithSettings
│   │   ├── GenerateJwtAssertion
│   │   ├── GetDefaultHeaders
│   │   ├── RemoveDefaultHeader
│   │   ├── SetBaseUrl
│   │   ├── SetTimeout
│   │   ├── SetUserAgent
│   ├── ExecuteNodes/       # HTTP method execution (5 methods)
│   │   ├── DELETE
│   │   ├── GET
│   │   ├── PATCH
│   │   ├── POST
│   │   └── PUT
│   ├── JsonNodes/          # JSON processing utilities (17 methods)
│   │   ├── DictionaryToJson
│   │   ├── Format
│   │   ├── FormatJson
│   │   ├── GetContent
│   │   ├── IsValid
│   │   ├── JsonToDictionary
│   │   ├── JsonToList
│   │   ├── JsonToObject
│   │   ├── ListToJson
│   │   ├── MinifyJson
```

```
│   │   ├── Serialize
│   │   ├── ToDictionary
│   │   ├── ToList
│   │   ├── ToObject
│   │   ├── TryToDictionary
│   │   ├── TryToList
│   │   └── ValidateJson
│   └── RequestNodes/        # Request building and configuration (19 methods)
│       ├── AddBearerToken
│       ├── AddFile
│       ├── AddFormField
│       ├── AddHeader
│       ├── AddHeaders
│       ├── AddJsonBody
│       ├── AddJsonContent
│       ├── AddParameter
│       ├── AddParameters
│       ├── AddTextContent
│       ├── AsDelete
│       ├── AsGet
│       ├── AsPatch
│       ├── AsPost
│       ├── AsPut
│       ├── ByEndpoint
│       ├── ByUrl
│       ├── ByCreateFileUpload
│       └── SetMethod
├── Utilities/               # Static utility methods
│   └── JsonHelper/          # Advanced JSON processing (12 methods)
│       ├── DictionaryToJson
│       ├── FormatJson
│       ├── IsValidJson
│       ├── JsonToDictionary
│       ├── JsonToList
│       ├── JsonToObject
│       ├── ListToJson
│       ├── MinifyJson
│       ├── Serialize
│       ├── SerializeSmart
│       ├── SerializeWithNewtonsoft
│       └── TrySerialize
└── System/                  # .NET system extensions
    ├── Exception/           # System exception handling (13 items)
    │   ├── Exception() - constructors (3 variations)
    │   ├── GetBaseException
    │   ├── GetType
    │   ├── Data
    │   ├── HelpLink
    │   ├── HResult
    │   ├── InnerException
```

```
    │   ├── Message
    │   ├── Source
    │   ├── StackTrace
    │   └── TargetSite
    └── Net/                 # Network utilities
        └── HttpStatusCode/  # Complete HTTP status code enumeration (50+ codes)
            ├── Accepted (202), AlreadyReported (208), Ambiguous (300)
            ├── BadGateway (502), BadRequest (400), Conflict (409)
            ├── Continue (100), Created (201), EarlyHints (103)
            ├── ExpectationFailed (417), FailedDependency (424), Forbidden (403)
            ├── Found (302), GatewayTimeout (504), Gone (410)
            ├── HttpVersionNotSupported (505), IMUsed (226), InsufficientStorage
(507)
            ├── InternalServerError (500), LengthRequired (411), Locked (423)
            ├── LoopDetected (508), MethodNotAllowed (405), MisdirectedRequest (421)
            ├── Moved (301), MovedPermanently (301), MultipleChoices (300)
            ├── MultiStatus (207), NetworkAuthenticationRequired (511), NoContent
(204)
            ├── NonAuthoritativeInformation (203), NotAcceptable (406), NotExtended
(510)
            ├── NotFound (404), NotImplemented (501), NotModified (304)
            ├── OK (200), PartialContent (206), PaymentRequired (402)
            ├── PermanentRedirect (308), PreconditionFailed (412),
PreconditionRequired (428)
            ├── Processing (102), ProxyAuthenticationRequired (407), Redirect (302)
            ├── RedirectKeepVerb (307), RedirectMethod (303),
RequestedRangeNotSatisfiable (416)
            ├── RequestEntityTooLarge (413), RequestHeaderFieldsTooLarge (431),
RequestTimeout (408)
            ├── RequestUriTooLong (414), ResetContent (205), SeeOther (303)
            ├── ServiceUnavailable (503), SwitchingProtocols (101),
TemporaryRedirect (307)
            ├── TooManyRequests (429), Unauthorized (401),
UnavailableForLegalReasons (451)
            ├── UnprocessableContent (422), UnprocessableEntity (422),
UnsupportedMediaType (415)
            ├── Unused (306), UpgradeRequired (426), UseProxy (305)
            └── VariantAlsoNegotiates (506)
```

Navigation

- Primary Workflow Nodes - Link to main documentation
- Core Infrastructure Classes - HTTP client internals & error handling
- System Utilities - .NET system extensions
- Utilities - JsonHelper advanced methods
- Advanced Usage Patterns - Complex scenarios

Primary Workflow Nodes

⭐ For complete documentation of the primary workflow nodes (ClientNodes, RequestNodes, ExecuteNodes, JsonNodes), see the Node Library Reference.

The primary workflow nodes are what you'll use for 90% of DynaFetch operations:

- **ClientNodes**: HTTP client creation and configuration
- **RequestNodes**: Request building and setup
- **ExecuteNodes**: HTTP method execution
- **JsonNodes**: JSON processing and data transformation

Core Infrastructure Classes

All classes available directly in DynaFetch/Core/ for advanced HTTP operations and error handling

Based on the Dynamo library structure, Core contains both HTTP infrastructure and exception handling classes:

HTTP Infrastructure Classes

HttpClientWrapper

Location: DynaFetch/Core/HttpClientWrapper
Purpose: Main HTTP client class that manages connections, configuration, and request execution

Properties and Methods Available:

- **BaseAddress** - Get/set the base URL for relative requests
- **Timeout** - Get/set request timeout duration
- **DefaultRequestHeaders** - Access to header collection
- **UserAgent** - Get/set User-Agent string

Advanced Usage:

```
HttpClientWrapper client = ClientNodes.Create()
Access: client.BaseAddress, client.Timeout, client.DefaultRequestHeaders
```

When to use directly:

- Custom timeout logic beyond simple seconds
- Complex header manipulation
- Advanced connection pooling scenarios
- Integration with existing HttpClient code

HttpRequest

Location: DynaFetch/Core/HttpRequest
Purpose: Represents a fully configured HTTP request before execution

Properties Available:

- **RequestUri** - The complete URL for the request
- **Method** - HTTP method (GET, POST, etc.)
- **Headers** - Request-specific headers
- **Content** - Request body content
- **Parameters** - Query parameters

Advanced Usage:

```
HttpRequest request = RequestNodes.ByUrl(client, url)
Access: request.RequestUri, request.Method, request.Headers
```

When to use directly:

- Inspecting request configuration before execution
- Custom request validation logic
- Complex conditional request building
- Debugging request construction

HttpResponse

Location: DynaFetch/Core/HttpResponse
Purpose: Contains the complete response from an HTTP request

Properties Available:

- **StatusCode** - HTTP status code (200, 404, etc.)
- **IsSuccessStatusCode** - True for 2xx status codes
- **Headers** - Response headers from server
- **Content** - Raw response content
- **ReasonPhrase** - HTTP status reason phrase
- **RequestMessage** - Original request that generated this response

Advanced Usage:

```
HttpResponse response = ExecuteNodes.GET(request)
Access: response.StatusCode, response.Headers, response.Content
```

When to use directly:

- Custom status code handling beyond success/failure
- Header inspection for caching, rate limiting, pagination
- Raw content processing for non-JSON responses
- Detailed error analysis and logging

Error Handling and Utility Classes

DynaFetchException

Location: DynaFetch/Core/DynaFetchException
Purpose: Base exception for all DynaFetch operations
Properties: ErrorCode, Details, InnerException
Usage: Catch-all for DynaFetch-related errors

```
try { /* DynaFetch operations */ }
catch (DynaFetchException ex) {
    // Handle any DynaFetch error
    Console.WriteLine($"Error: {ex.Message}, Code: {ex.ErrorCode}");
}
```

DynaFetchHttpException

Location: DynaFetch/Core/DynaFetchHttpException
Purpose: HTTP-specific errors (timeouts, connection failures, server errors)
Properties: StatusCode, ResponseContent, RequestUrl
Usage: Network and HTTP protocol errors

```
Common scenarios:
- Network timeouts
- Server 500 errors
- Connection refused
- DNS resolution failures
```

DynaFetchJsonException

Location: DynaFetch/Core/DynaFetchJsonException
Purpose: JSON parsing and processing errors
Properties: JsonContent, ParseAttempt, Engine
Usage: Data format and conversion errors

```
Common scenarios:
- Invalid JSON syntax
```

```
- Unexpected JSON structure
- Type conversion failures
- Encoding issues
```

ErrorDetails

Location: DynaFetch/Core/ErrorDetails
Purpose: Structured error information for debugging
Properties: Timestamp, Operation, Context, StackTrace
Usage: Detailed error analysis and logging

```
Provides context for:
- Which operation failed
- What inputs caused the failure
- When the error occurred
- Full diagnostic information
```

SafeOperations

Location: DynaFetch/Core/SafeOperations
Purpose: Utility class for error-resistant operations

Available Methods:

- **TryGetValue** - Safe dictionary value extraction
- **TryParseJson** - Safe JSON parsing with fallback
- **ValidateUrl** - URL format validation
- **SanitizeInput** - Input cleaning and validation

Advanced Usage:

```
Use SafeOperations for operations that might fail gracefully
Example: SafeOperations.TryGetValue(dictionary, "key", defaultValue)
```

When to use directly:

- Building error-resistant workflows
- Handling unpredictable data sources
- Creating robust automation that won't crash
- Input validation in custom nodes

Validation

Location: DynaFetch/Core/Validation
Purpose: Input validation and sanitization utilities

Available Methods:

- **ValidateUrl** - Check URL format and accessibility
- **ValidateJson** - JSON syntax validation
- **ValidateHeaders** - HTTP header format validation
- **SanitizeString** - String cleaning for safe processing

Advanced Usage:

```
Use Validation for pre-processing inputs before API calls
Example: Validation.ValidateUrl(userInput) before creating requests
```

When to use directly:

- User input processing in custom interfaces
- Bulk URL validation
- Data quality checks in automation
- Security-conscious applications

System Utilities

Complete .NET system extensions and HTTP status code enumeration

The System folder contains extensive .NET extensions that provide direct access to system-level functionality and comprehensive HTTP status code handling.

Exception Handling (DynaFetch/System/Exception/)

Complete .NET Exception class with all constructors, properties, and methods for comprehensive error handling:

Exception Constructors:

- **Exception()** - Default exception constructor with no parameters
- **Exception(message)** - Exception with custom error message
- **Exception(message, innerException)** - Exception with message and inner exception chaining

Exception Methods:

- **GetBaseException** - Gets the original exception at the root of the exception chain
- **GetType** - Gets the runtime type information of the current exception instance

Exception Properties:

- **Data** - Key-value pairs providing additional user-defined information about the exception
- **HelpLink** - Link or URN to help documentation associated with this exception
- **HResult** - HRESULT error code value assigned to a specific exception
- **InnerException** - The exception instance that caused the current exception
- **Message** - Human-readable error message that describes the current exception
- **Source** - Name of the application or object that caused the current exception
- **StackTrace** - String representation of the immediate frames on the call stack
- **TargetSite** - Method that threw the current exception

Advanced Exception Usage:

```
try {
    // DynaFetch operations that might fail
    HttpResponse response = ExecuteNodes.GET(request)
} catch (Exception ex) {
    // Access comprehensive exception information
    Exception baseException = ex.GetBaseException()
    Type exceptionType = ex.GetType()

    // Get detailed error context
    string errorMessage = ex.Message
    string errorSource = ex.Source
    string stackTrace = ex.StackTrace
    Exception innerEx = ex.InnerException

    // Access additional data
    object helpInfo = ex.HelpLink
    int errorCode = ex.HResult
    object additionalData = ex.Data
    object targetMethod = ex.TargetSite

    // Use for comprehensive error logging and debugging
    Console.WriteLine($"Exception Type: {exceptionType}")
    Console.WriteLine($"Message: {errorMessage}")
    Console.WriteLine($"Source: {errorSource}")
    Console.WriteLine($"Stack Trace: {stackTrace}")

    if (innerEx != null) {
        Console.WriteLine($"Inner Exception: {innerEx.Message}")
    }
}
```

Exception Property Details:

Data: Dictionary-like collection for storing arbitrary data related to the exception

```
Exception ex = new Exception("Custom error")
ex.Data["UserId"] = "12345"
ex.Data["Operation"] = "API Call"
ex.Data["Timestamp"] = DateTime.Now
```

HelpLink: URL or URN pointing to documentation about the error

```
Exception ex = new Exception("API Error")
ex.HelpLink = "https://api-docs.example.com/errors/authentication"
```

HResult: Win32 HRESULT error code for interoperability

```
// Access the underlying system error code
int systemErrorCode = ex.HResult
```

InnerException: Chain of exceptions for root cause analysis

```
try {
    // Some operation that causes a chain of exceptions
} catch (Exception ex) {
    Exception current = ex
    while (current != null) {
        Console.WriteLine($"Exception: {current.Message}")
        current = current.InnerException
    }
}
```

Message: Primary error description

```
string userFriendlyError = ex.Message
```

Source: Application or assembly name where the exception originated

```
string errorOrigin = ex.Source
```

StackTrace: Detailed call stack at the point where exception was thrown

```
```

```
string detailedTrace = ex.StackTrace
// Use for debugging and error analysis
```

TargetSite: Specific method that threw the exception

```
object faultingMethod = ex.TargetSite
// Provides method name and signature information
```

Network Utilities (DynaFetch/System/Net/)

HttpStatusCode Complete Enumeration

DynaFetch provides direct access to ALL HTTP status codes as individual nodes. This comprehensive collection covers every standard HTTP response code:

1xx Informational Responses:

- **Continue (100)** - Server has received request headers, client should proceed
- **SwitchingProtocols (101)** - Server is switching protocols per client request
- **Processing (102)** - Server has received and is processing request (WebDAV)
- **EarlyHints (103)** - Server is returning some response headers early

2xx Success Responses:

- **OK (200)** - Standard successful HTTP request response
- **Created (201)** - Request has been fulfilled, new resource created
- **Accepted (202)** - Request accepted for processing, but not completed
- **NonAuthoritativeInformation (203)** - Request successful, info from another source
- **NoContent (204)** - Request successful, no content to return
- **ResetContent (205)** - Request successful, user agent should reset document view
- **PartialContent (206)** - Server delivering only part of resource (byte serving)
- **MultiStatus (207)** - Message body contains multiple status messages (WebDAV)
- **AlreadyReported (208)** - Members already enumerated in previous reply (WebDAV)
- **IMUsed (226)** - Request fulfilled, instance-manipulations applied

3xx Redirection Messages:

- **MultipleChoices (300)** - Multiple options for resource client may follow
- **Ambiguous (300)** - Alias for MultipleChoices
- **Moved (301)** - Alias for MovedPermanently
- **MovedPermanently (301)** - Resource moved permanently to new location
- **Found (302)** - Resource temporarily moved to different URI

- **Redirect (302)** - Alias for Found
- **RedirectMethod (303)** - Response found under different URI using GET
- **SeeOther (303)** - Alias for RedirectMethod
- **NotModified (304)** - Resource not modified since last request
- **UseProxy (305)** - Resource must be accessed through proxy
- **Unused (306)** - No longer used, reserved
- **TemporaryRedirect (307)** - Resource temporarily moved, same method should be used
- **RedirectKeepVerb (307)** - Alias for TemporaryRedirect
- **PermanentRedirect (308)** - Resource permanently moved, same method should be used

4xx Client Error Responses:

- **BadRequest (400)** - Server cannot process request due to client error
- **Unauthorized (401)** - Authentication required for access
- **PaymentRequired (402)** - Reserved for future use
- **Forbidden (403)** - Server understood request but refuses to authorize
- **NotFound (404)** - Requested resource could not be found
- **MethodNotAllowed (405)** - Request method not supported for resource
- **NotAcceptable (406)** - Resource not available in format specified by Accept headers
- **ProxyAuthenticationRequired (407)** - Client must authenticate with proxy
- **RequestTimeout (408)** - Server timed out waiting for request
- **Conflict (409)** - Request conflicts with current state of target resource
- **Gone (410)** - Resource no longer available and will not be available again
- **LengthRequired (411)** - Server requires Content-Length header
- **PreconditionFailed (412)** - Server does not meet preconditions in request
- **RequestEntityTooLarge (413)** - Request entity larger than server is able to process
- **RequestUriTooLong (414)** - URI provided was too long for server to process
- **UnsupportedMediaType (415)** - Media type of request not supported
- **RequestedRangeNotSatisfiable (416)** - Range specified by Range header cannot be fulfilled
- **ExpectationFailed (417)** - Server cannot meet requirements of Expect request header
- **MisdirectedRequest (421)** - Request directed at server unable to produce response
- **UnprocessableEntity (422)** - Request well-formed but unable to be processed
- **UnprocessableContent (422)** - Alias for UnprocessableEntity
- **Locked (423)** - Resource being accessed is locked (WebDAV)
- **FailedDependency (424)** - Request failed due to failure of previous request (WebDAV)
- **UpgradeRequired (426)** - Client should switch to different protocol
- **PreconditionRequired (428)** - Origin server requires request to be conditional
- **TooManyRequests (429)** - User has sent too many requests in given time
- **RequestHeaderFieldsTooLarge (431)** - Server unwilling to process request with large headers
- **UnavailableForLegalReasons (451)** - Resource unavailable for legal reasons

5xx Server Error Responses:

- **InternalServerError (500)** - Generic server error message

- **NotImplemented (501)** - Server does not support functionality required to fulfill request
- **BadGateway (502)** - Server acting as gateway received invalid response
- **ServiceUnavailable (503)** - Server currently unavailable (overloaded or down)
- **GatewayTimeout (504)** - Server acting as gateway did not receive timely response
- **HttpVersionNotSupported (505)** - Server does not support HTTP protocol version
- **VariantAlsoNegotiates (506)** - Transparent content negotiation results in circular reference
- **InsufficientStorage (507)** - Server unable to store representation needed to complete request
- **LoopDetected (508)** - Server detected infinite loop while processing request
- **NotExtended (510)** - Further extensions to request are required for fulfillment
- **NetworkAuthenticationRequired (511)** - Client needs to authenticate to gain network access

Using HTTP Status Codes:

```
HttpResponse response = ExecuteNodes.GET(request)

// Compare with specific status codes
if (response.StatusCode == HttpStatusCode.OK) {
    // Process successful response
} else if (response.StatusCode == HttpStatusCode.NotFound) {
    // Handle 404 error
} else if (response.StatusCode == HttpStatusCode.Unauthorized) {
    // Handle authentication error
} else if (response.StatusCode == HttpStatusCode.TooManyRequests) {
    // Handle rate limiting
    // Check for Retry-After header
}

// Check status code categories
if (response.StatusCode >= 200 && response.StatusCode < 300) {
    // Success range (2xx)
} else if (response.StatusCode >= 400 && response.StatusCode < 500) {
    // Client error range (4xx)
} else if (response.StatusCode >= 500) {
    // Server error range (5xx)
}
```

Common Status Code Usage Patterns:

Success Handling:

- **200 OK**: Standard successful response - process content normally
- **201 Created**: Resource created successfully - often includes Location header
- **204 No Content**: Success but no response body - check headers for confirmation

Redirection Handling:

- **301/302/307/308**: Automatic redirect handling - usually transparent to user
- **304 Not Modified**: Use cached version - no new content to process

Client Error Handling:

- **400 Bad Request**: Check request format and required parameters
- **401 Unauthorized**: Authentication required - prompt for credentials
- **403 Forbidden**: Access denied - user lacks permissions
- **404 Not Found**: Resource doesn't exist - verify URL correctness
- **429 Too Many Requests**: Rate limited - implement backoff strategy

Server Error Handling:

- **500 Internal Server Error**: Server-side issue - retry may help
- **502/503/504**: Service issues - implement retry with exponential backoff
- **501 Not Implemented**: Feature not supported - try alternative approach

---

Utilities (DynaFetch/Utilities/)

JsonHelper (DynaFetch/Utilities/JsonHelper/)

Advanced JSON processing utilities beyond the standard JsonNodes workflow nodes. These static methods provide comprehensive JSON manipulation capabilities:

Data Conversion Methods:

- **DictionaryToJson** - Converts Dictionary<string, object> to JSON string
- **JsonToDictionary** - Converts JSON string to Dictionary<string, object>
- **JsonToList** - Converts JSON array string to List
- **JsonToObject** - Converts JSON string to most appropriate .NET object type
- **ListToJson** - Converts List to JSON array string

  Validation and Formatting:

  - **IsValidJson** - Validates JSON string syntax and structure
  - **FormatJson** - Pretty-prints JSON with proper indentation and formatting
  - **MinifyJson** - Removes whitespace and formatting to create compact JSON

  Serialization Methods:

  - **Serialize** - Converts any .NET object to JSON string using default settings
  - **SerializeSmart** - Intelligent serialization with type preservation and null handling
  - **SerializeWithNewtonsoft** - Forces use of Newtonsoft.Json engine for compatibility

  Safe Processing Methods:

- **TrySerialize** - Safe serialization that returns null on failure instead of throwing exception

Advanced Usage Examples:

Data Conversion:

```
// Convert Dynamo Dictionary to JSON
Dictionary<string, object> data = GetDynamoData()
string json = JsonHelper.DictionaryToJson(data)

// Convert JSON response to Dynamo Dictionary
string apiResponse = GetApiResponse()
Dictionary<string, object> result = JsonHelper.JsonToDictionary(apiResponse)

// Handle JSON arrays
string jsonArray = "[{\"name\":\"John\"},{\"name\":\"Jane\"}]"
List<object> users = JsonHelper.JsonToList(jsonArray)

// Smart object conversion
string jsonData = "{\"count\":42,\"active\":true}"
object result = JsonHelper.JsonToObject(jsonData)  // Returns Dictionary
```

Validation and Formatting:

```
// Validate JSON before processing
string userJson = GetUserInput()
if (JsonHelper.IsValidJson(userJson)) {
    // Safe to process
    Dictionary data = JsonHelper.JsonToDictionary(userJson)
} else {
    // Handle invalid JSON
    Console.WriteLine("Invalid JSON format")
}

// Format JSON for display
string compactJson = "{\"name\":\"John\",\"age\":30}"
string prettyJson = JsonHelper.FormatJson(compactJson)
/*
Result:
{
  "name": "John",
  "age": 30
}
*/

// Minify JSON for transmission
```

```
string formattedJson = GetPrettyJson()
string compact = JsonHelper.MinifyJson(formattedJson)
```

Advanced Serialization:

```
// Standard serialization
MyCustomObject obj = new MyCustomObject()
string json = JsonHelper.Serialize(obj)

// Smart serialization with better type handling
ComplexObject complex = GetComplexData()
string smartJson = JsonHelper.SerializeSmart(complex)

// Force Newtonsoft.Json for compatibility
LegacyObject legacy = GetLegacyData()
string compatibleJson = JsonHelper.SerializeWithNewtonsoft(legacy)

// Safe serialization for unreliable data
UnknownObject unknown = GetUnknownData()
string result = JsonHelper.TrySerialize(unknown)
if (result != null) {
    // Serialization succeeded
} else {
    // Handle serialization failure
}
```

Method Comparison and When to Use:

DictionaryToJson vs Serialize:

- Use `DictionaryToJson` for Dynamo Dictionary objects specifically
- Use `Serialize` for general .NET objects and custom types

JsonToDictionary vs JsonToObject:

- Use `JsonToDictionary` when you know the JSON represents an object
- Use `JsonToObject` for unknown JSON that could be object, array, or primitive

Serialize vs SerializeSmart vs SerializeWithNewtonsoft:

- Use `Serialize` for standard scenarios with good performance
- Use `SerializeSmart` for complex objects requiring type preservation
- Use `SerializeWithNewtonsoft` for maximum compatibility with legacy systems

FormatJson vs MinifyJson:

- Use `FormatJson` for human-readable output, debugging, or display
- Use `MinifyJson` for network transmission or storage optimization

TrySerialize vs Serialize:

- Use `TrySerialize` when input data reliability is uncertain
- Use `Serialize` for known good data where exceptions are acceptable

Performance Considerations:

High-Performance Scenarios:

- `Serialize` and `JsonToDictionary` use System.Text.Json for optimal speed
- `MinifyJson` is faster than `FormatJson` for processing large JSON
- `IsValidJson` is lightweight and should be used before expensive parsing

Compatibility Scenarios:

- `SerializeWithNewtonsoft` provides maximum compatibility but slower performance
- `SerializeSmart` balances performance and reliability
- `JsonToObject` handles edge cases but has slight overhead

Memory Efficiency:

- `TrySerialize` prevents exception overhead in high-volume scenarios
- `MinifyJson` reduces memory footprint for large JSON strings
- `JsonToList` and `JsonToDictionary` use efficient native collections

Advanced Usage Patterns

Pattern 1: Custom Error Handling

```
HttpResponse response = ExecuteNodes.GET(request)

// Check status code directly
if (response.StatusCode == 429) {
    // Rate limited - wait and retry
    Thread.Sleep(1000);
    response = ExecuteNodes.GET(request);
}

// Inspect headers for rate limit info
if (response.Headers.ContainsKey("X-RateLimit-Remaining")) {
    int remaining = int.Parse(response.Headers["X-RateLimit-Remaining"]);
    // Adjust request frequency based on remaining quota
}
```

Pattern 2: Advanced Authentication

```
HttpClientWrapper client = ClientNodes.Create()

// Custom authentication logic
if (tokenExpired) {
    // Use HttpRequest directly for token refresh
    HttpRequest tokenRequest = RequestNodes.ByUrl(client, tokenEndpoint)
    tokenRequest = RequestNodes.AddJsonBody(tokenRequest, credentials)
    HttpResponse tokenResponse = ExecuteNodes.POST(tokenRequest)

    // Extract new token and update client
    Dictionary tokenData = JsonNodes.ToDictionary(tokenResponse)
    string newToken = tokenData["access_token"]
    client = ClientNodes.AddDefaultHeader(client, "Authorization", "Bearer " +
newToken)
}
```

Pattern 3: Bulk Operations with Error Recovery

```
List<string> urls = GetUrlList()
List<Dictionary> results = new List<Dictionary>()

HttpClientWrapper client = ClientNodes.Create()

foreach (string url in urls) {
    try {
        HttpResponse response = ExecuteNodes.GET(client, url)
        if (response.IsSuccessStatusCode) {
            Dictionary data = JsonNodes.ToDictionary(response)
            results.Add(data)
        } else {
            // Log error but continue processing
            ErrorDetails.LogError($"Failed to process {url}:
{response.StatusCode}")
        }
    } catch (DynaFetchException ex) {
        // Handle specific DynaFetch errors
        ErrorDetails.LogError($"DynaFetch error for {url}: {ex.ErrorCode}")
    }
}
```

Pattern 4: Advanced JSON Processing

```
HttpResponse response = ExecuteNodes.GET(request)

// Validate JSON structure before processing
if (JsonHelper.ValidateSchema(response.Content, expectedSchema)) {
    // Use streaming parse for large responses
    object data = JsonHelper.StreamingParse(response.Content)

    // Custom deserialization with type information
    MyCustomType typed = JsonHelper.DeserializeWithTypeInfo<MyCustomType>
(response.Content)
} else {
    // Handle schema validation failure
    throw new DynaFetchJsonException("Response doesn't match expected schema")
}
```

Pattern 5: Connection Management

```
// Create client with advanced configuration
HttpClientWrapper client = ClientNodes.Create()

// Direct access to underlying HttpClient properties
client.Timeout = TimeSpan.FromMinutes(5)  // 5-minute timeout
client.DefaultRequestHeaders.Add("Accept-Encoding", "gzip, deflate")

// Use SafeOperations for robust processing
SafeOperations.TryGetValue(response.Headers, "Content-Length", out int
contentLength)
if (contentLength > 10000000) {  // 10MB
    // Handle large response differently
    string content = JsonHelper.StreamingParse(response.Content)
}
```

Performance Considerations

Core Class Usage

- **HttpClientWrapper**: Reuse instances - creating new clients has overhead
- **HttpRequest**: Lightweight objects - safe to create many instances
- **HttpResponse**: Contains response data - dispose promptly for memory management

Exception Handling Performance

- **Specific Catches**: Catch specific exception types rather than generic Exception
- **Error Codes**: Use ErrorCode properties for fast error categorization

- **Logging**: ErrorDetails provides structured logging without performance impact

Advanced JSON Performance

- **Streaming**: Use JsonHelper.StreamingParse for responses >1MB
- **Bulk Operations**: JsonHelper.BulkDeserialize for processing many small JSON objects
- **Memory**: Dispose large JSON objects promptly to prevent memory leaks

---

When to Use Advanced Classes

Use Core Classes When:

- Building custom workflow nodes
- Need direct control over HTTP configuration
- Implementing complex authentication flows
- Requiring detailed response inspection
- Integrating with existing HttpClient code

Use System Classes When:

- Building error-resistant automation
- Need detailed error analysis and logging
- Handling unpredictable data sources
- Implementing custom retry logic
- Building production-grade applications

Use Primary Workflow Nodes When:

- Standard API integration scenarios
- Learning DynaFetch capabilities
- Building typical GET/POST workflows
- Simple authentication patterns
- Most Dynamo automation tasks

Integration with Primary Nodes

The advanced classes work seamlessly with the primary workflow nodes:

```
// Start with workflow nodes
HttpClientWrapper client = ClientNodes.Create()
HttpRequest request = RequestNodes.ByUrl(client, url)

// Add advanced configuration
request.Headers.Add("Custom-Header", "Value")
Validation.ValidateUrl(request.RequestUri)
```

```
// Execute and handle with advanced classes
try {
    HttpResponse response = ExecuteNodes.GET(request)

    if (response.StatusCode == 200) {
        Dictionary data = JsonNodes.ToDictionary(response)
    } else {
        ErrorDetails.LogError($"HTTP {response.StatusCode}:
{response.ReasonPhrase}")
    }
} catch (DynaFetchHttpException httpEx) {
    // Handle HTTP-specific errors
} catch (DynaFetchJsonException jsonEx) {
    // Handle JSON-specific errors
}
```

Debugging and Troubleshooting

Using Core Classes for Debugging

```
// Inspect request before sending
HttpRequest request = RequestNodes.ByUrl(client, url)
Console.WriteLine($"URL: {request.RequestUri}")
Console.WriteLine($"Method: {request.Method}")
Console.WriteLine($"Headers: {string.Join(", ", request.Headers.Keys)}")

// Inspect response details
HttpResponse response = ExecuteNodes.GET(request)
Console.WriteLine($"Status: {response.StatusCode} {response.ReasonPhrase}")
Console.WriteLine($"Response Headers: {string.Join(", ",
response.Headers.Keys)}")
Console.WriteLine($"Content Length: {response.Content.Length}")
```

Using Exception Classes for Error Analysis

```
try {
    // DynaFetch operations
} catch (DynaFetchHttpException httpEx) {
    Console.WriteLine($"HTTP Error: {httpEx.StatusCode}")
    Console.WriteLine($"URL: {httpEx.RequestUrl}")
    Console.WriteLine($"Response: {httpEx.ResponseContent}")
} catch (DynaFetchJsonException jsonEx) {
    Console.WriteLine($"JSON Error: {jsonEx.Message}")
```

```
    Console.WriteLine($"Content: {jsonEx.JsonContent}")
    Console.WriteLine($"Parse Engine: {jsonEx.Engine}")
}
```

Summary

The advanced node library provides complete access to DynaFetch's capabilities:

- **Core Classes**: Direct HTTP manipulation and advanced configuration
- **System Utilities**: Robust error handling and validation
- **JsonHelper**: High-performance JSON processing
- **Exception Classes**: Detailed error analysis and debugging

For most users, the primary workflow nodes provide everything needed for REST API integration. Use the advanced classes when you need greater control, better error handling, or are building complex automation systems.

For everyday API workflows, see Node Library Reference
For detailed examples and parameters, see API Documentation
For troubleshooting, see Troubleshooting Guide