DynaWeb to DynaFetch Migration Guide

This guide helps you transition from DynaWeb to DynaFetch with side-by-side comparisons and migration examples.

About This Migration

DynaFetch builds upon the excellent foundation established by **Radu Gidei's DynaWeb package**. DynaWeb pioneered REST API integration in Dynamo and established many patterns that DynaFetch continues and modernizes.

Why Migrate to DynaFetch?

- **Modern .NET 8**: Built for Dynamo 3.0 with latest HTTP capabilities
- **Simplified API**: Fewer nodes needed for common operations
- **Enhanced JSON**: Dual-engine JSON processing (System.Text.Json + Newtonsoft.Json)
- **Better Authentication**: Client-level persistent authentication
- **Improved Error Handling**: More specific error messages and validation
- **Better Performance**: Modern async-to-sync conversion optimized for Dynamo

Attribution

We acknowledge and appreciate Radu Gidei's pioneering work. DynaWeb established the foundation for REST API integration in Dynamo, and DynaFetch continues this legacy with modern enhancements.

Core Concept Changes

DynaWeb Approach (3-Object Pattern)

```
WebClient → WebRequest → WebResponse → Data Processing
```

DynaFetch Approach (2-Object Pattern)

```
HttpClient → HttpResponse → Data Processing
```

Key Difference: DynaFetch eliminates the separate request building step for simple operations, making common workflows more direct.

Migration Examples

Basic GET Request

DynaWeb (4-5 nodes):

```
1. WebClient.ByUrl("https://api.example.com") → client
2. WebRequest.ByUrl("https://api.example.com/data") → request
3. WebClient.Execute(client, request) → response
4. WebResponse.Content(response) → json_string
5. JSON.Deserialize(json_string) → data
```

DynaFetch (3 nodes):

```
1. ClientNodes.Create() → client
2. ExecuteNodes.GET(client, "https://api.example.com/data") → response
3. JsonNodes.ToDictionary(response) → data
```

Migration Steps:

1. Replace `WebClient.ByUrl()` with `ClientNodes.Create()`
2. Replace `WebRequest.ByUrl()` + `WebClient.Execute()` with `ExecuteNodes.GET()`
3. Replace `WebResponse.Content()` + `JSON.Deserialize()` with `JsonNodes.ToDictionary()`

Authentication

DynaWeb Approach:

```
1. WebClient.ByUrl(baseUrl) → client
2. WebRequest.ByUrl(endpoint) → request
3. WebRequest.AddHeader(request, "Authorization", "Bearer " + token) → request
4. WebClient.Execute(client, request) → response
```

DynaFetch Approach:

```
1. ClientNodes.Create() → client
2. ClientNodes.AddDefaultHeader(client, "Authorization", "Bearer " + token) → client
3. ExecuteNodes.GET(client, url) → response
```

Key Improvement: Authentication headers persist across all requests from the same client. No need to add auth to every request.

POST Request with JSON Data

DynaWeb Approach:

```
1. WebClient.ByUrl(baseUrl) → client
2. WebRequest.ByUrl(endpoint) → request
3. WebRequest.AddHeader(request, "Content-Type", "application/json") → request
4. WebRequest.AddStringContent(request, json_data) → request
5. WebRequest.SetMethod(request, "POST") → request
6. WebClient.Execute(client, request) → response
```

DynaFetch Approach:

```
1. ClientNodes.Create() → client
2. ExecuteNodes.POST(client, url, json_data) → response
```

Key Improvement: Content-Type headers and method setting handled automatically.

JSON Processing

DynaWeb Approach:

```
1. WebResponse.Content(response) → json_string
2. JSON.Deserialize(json_string) → object
3. [Manual conversion to Dictionary/List]
```

DynaFetch Approach:

```
1. JsonNodes.ToDictionary(response) → dictionary
// OR
1. JsonNodes.ToList(response) → list
```

Key Improvement: Direct conversion to Dynamo-native Dictionary/List types without intermediate steps.

Node Mapping Reference

Client Management

DynaWeb	DynaFetch	Notes
`WebClient.ByUrl(url)`	`ClientNodes.CreateWithBaseUrl(url)`	Similar functionality
`WebClient.ByUrl("")`	`ClientNodes.Create()`	For clients without base URL

DynaWeb	DynaFetch	Notes
Not available	`ClientNodes.SetTimeout(client, seconds)`	New timeout control
Not available	`ClientNodes.AddDefaultHeader(client, name, value)`	Persistent authentication

Request Building

DynaWeb	DynaFetch	Notes
`WebRequest.ByUrl(url)`	`ExecuteNodes.GET(client, url)`	Direct execution
`WebRequest.AddHeader(request, name, value)`	`ClientNodes.AddDefaultHeader(client, name, value)`	Now client-level
`WebRequest.SetMethod(request, "POST")`	`ExecuteNodes.POST(client, url, data)`	Method-specific nodes
`WebRequest.AddStringContent(request, data)`	Built into `ExecuteNodes.POST()`	Automatic content handling

Response Processing

DynaWeb	DynaFetch	Notes
`WebResponse.Content(response)`	`JsonNodes.GetContent(response)`	Similar raw content access
`WebResponse.StatusCode(response)`	`response.StatusCode`	Direct property access
`JSON.Deserialize(json)`	`JsonNodes.ToDictionary(response)`	Direct response processing
Not available	`JsonNodes.ToList(response)`	New array processing
Not available	`JsonNodes.Format(response)`	New pretty-printing

Step-by-Step Migration Process

Step 1: Install DynaFetch

1. Open Dynamo 3.0+
2. Go to Packages → Search for a Package
3. Search "DynaFetch" and install
4. Keep DynaWeb installed during transition

Step 2: Identify Migration Candidates

Start with these DynaWeb patterns (easiest to migrate):

- Simple GET requests
- Basic authentication patterns
- JSON response processing
- POST requests with JSON data

Step 3: Convert Simple GET Requests First

Before (DynaWeb):

```
WebClient.ByUrl(baseUrl) → client
WebRequest.ByUrl(endpoint) → request
WebClient.Execute(client, request) → response
WebResponse.Content(response) → json
JSON.Deserialize(json) → data
```

After (DynaFetch):

```
ClientNodes.Create() → client
ExecuteNodes.GET(client, fullUrl) → response
JsonNodes.ToDictionary(response) → data
```

Step 4: Migrate Authentication Patterns

Before (DynaWeb) - Auth per request:

```
WebRequest.ByUrl(url) → request
WebRequest.AddHeader(request, "Authorization", "Bearer " + token) → request
WebClient.Execute(client, request) → response
```

After (DynaFetch) - Auth per client:

```
ClientNodes.Create() → client
ClientNodes.AddDefaultHeader(client, "Authorization", "Bearer " + token) → client
ExecuteNodes.GET(client, url) → response
```

Step 5: Update POST Requests

Before (DynaWeb):

```
WebRequest.ByUrl(url) → request
WebRequest.AddHeader(request, "Content-Type", "application/json") → request
WebRequest.AddStringContent(request, jsonData) → request
WebRequest.SetMethod(request, "POST") → request
WebClient.Execute(client, request) → response
```

After (DynaFetch):

```
ExecuteNodes.POST(client, url, jsonData) → response
```

Step 6: Test and Validate

1. Test each migrated workflow independently
2. Verify authentication still works
3. Check JSON processing produces same results
4. Validate error handling behaves correctly

Common Migration Challenges

Challenge 1: Base URL Handling

DynaWeb Pattern:

```
client = WebClient.ByUrl("https://api.example.com")
request = WebRequest.ByUrl("/users/123")  // Relative URL
```

DynaFetch Solutions:

Option A - Use base URL client:

```
client = ClientNodes.CreateWithBaseUrl("https://api.example.com")
response = ExecuteNodes.GET(client, "/users/123")  // Relative URL works
```

Option B - Use full URLs:

```
client = ClientNodes.Create()
response = ExecuteNodes.GET(client, "https://api.example.com/users/123")  // Full
URL
```

Challenge 2: Request Headers vs Default Headers

DynaWeb: Headers added per request **DynaFetch**: Headers added per client (more efficient)

If you need per-request headers, use RequestNodes:

```
request = RequestNodes.ByUrl(url)
request = RequestNodes.AddHeader(request, "Special-Header", "value")
response = ExecuteNodes.Execute(client, request)
```

Challenge 3: Complex Request Building

For complex requests that require fine control, DynaFetch still provides RequestNodes:

DynaWeb Style (still available in DynaFetch):

```
request = RequestNodes.ByUrl(url)
request = RequestNodes.AddHeader(request, "Custom-Header", "value")
request = RequestNodes.AddBearerToken(request, token)
request = RequestNodes.AddJsonBody(request, jsonData)
response = ExecuteNodes.Execute(client, request)
```

Challenge 4: JSON Array Processing

DynaWeb: Manual handling after JSON.Deserialize **DynaFetch**: Direct conversion

```
// DynaWeb - manual list processing
json = WebResponse.Content(response)
data = JSON.Deserialize(json)
// Manual iteration/conversion needed

// DynaFetch - direct list conversion
list = JsonNodes.ToList(response)  // Direct to Dynamo List
```

Challenge 5: File Upload Migration

DynaWeb Pattern:

```
WebClient.ByUrl(baseUrl) → client
WebRequest.ByUrl(endpoint) → request
```

```
WebRequest.AddFile(request, fieldName, filePath) → request
WebClient.Execute(client, request) → response
```

DynaFetch Pattern (Method 1 - DynaWeb Compatible):

```
ClientNodes.Create() → client
ClientNodes.AddDefaultHeader(client, "Authorization", "Bearer " + token)
RequestNodes.ByUrl(url) → request
RequestNodes.AddFile(request, fieldName, filePath, contentType) → request
ExecuteNodes.POST(client, "", request) → response
```

DynaFetch Pattern (Method 2 - Direct Upload):

```
RequestNodes.CreateFileUpload(filePath, fieldName, fileName, contentType) → formData
ExecuteNodes.POST(client, url, formData) → response
```

Key Improvements:

- Auto-detection of MIME types from file extensions
- Support for adding metadata fields with `RequestNodes.AddFormField()`
- Works with POST, PUT, and PATCH methods
- Better error handling for missing files

File Upload with Metadata:

```
// DynaFetch adds ability to include form fields with file
formData = RequestNodes.CreateFileUpload(filePath, "file")
formData = RequestNodes.AddFormField(formData, "description", "Project photo")
formData = RequestNodes.AddFormField(formData, "category", "Construction")
response = ExecuteNodes.POST(client, uploadUrl, formData)
```

Performance Improvements

Faster JSON Processing

DynaFetch uses dual JSON engines:

- **System.Text.Json**: Primary engine (faster)
- **Newtonsoft.Json**: Fallback for compatibility

This provides better performance than DynaWeb's single JSON engine.

Reduced Node Count

Common operations require fewer nodes:

- **Simple GET**: 5 nodes → 3 nodes (40% reduction)
- **Authenticated GET**: 6 nodes → 4 nodes (33% reduction)
- **POST with auth**: 8 nodes → 4 nodes (50% reduction)

Better Error Messages

DynaFetch provides more specific error messages:

- URL validation with suggestions
- Authentication error details
- JSON parsing error specifics
- Network timeout descriptions

Compatibility Notes

What Stays the Same

- Basic workflow concepts (client → request → response)
- JSON data structures and format
- HTTP status codes and error handling
- Authentication token formats

What Changes

- Node names and organization
- Number of nodes required for common operations
- Header management approach (per-client vs per-request)
- JSON processing methods

What's New in DynaFetch

- Client-level default headers for persistent authentication
- Direct JSON-to-Dictionary/List conversion
- Dual JSON engine system for performance
- Method-specific execute nodes (GET, POST, PUT, DELETE, PATCH)
- Enhanced error handling and validation
- Pretty-printing and JSON formatting utilities

Migration Testing Strategy

1. Parallel Testing

Keep both packages installed and test side-by-side:

```
// DynaWeb workflow
[Original DynaWeb nodes] → dynawebResult

// DynaFetch workflow
[New DynaFetch nodes] → dynafetchResult

// Compare results
dynawebResult == dynafetchResult
```

2. Incremental Migration

Migrate workflows one at a time:

1. Start with simplest GET requests
2. Move to authenticated requests
3. Convert POST operations
4. Migrate complex workflows last

3. Validation Checklist

For each migrated workflow:

- ☐ Same API endpoints called
- ☐ Same authentication headers sent
- ☐ Same JSON data submitted (for POST/PUT)
- ☐ Same response data received
- ☐ Error handling works correctly
- ☐ Performance is acceptable

Rollback Strategy

If you need to rollback to DynaWeb:

1. Keep DynaWeb installed during migration period
2. Document which workflows have been migrated
3. Test thoroughly before removing DynaWeb
4. Keep backup copies of working DynaWeb graphs

Migration Timeline Recommendations

Week 1: Preparation

- Install DynaFetch alongside DynaWeb
- Review existing DynaWeb workflows
- Identify simple GET requests for first migration

Week 2: Basic Migration

- Convert simple GET requests
- Test basic JSON processing
- Validate results match DynaWeb

Week 3: Authentication

- Migrate authenticated workflows
- Convert to client-level authentication pattern
- Test with real API endpoints

Week 4: Advanced Features

- Convert POST/PUT/DELETE operations
- Migrate complex workflows
- Performance testing and optimization

Week 5: Validation & Cleanup

- Comprehensive testing of all migrated workflows
- Remove DynaWeb dependency
- Document new DynaFetch patterns for team

Getting Help

Resources

- **DynaFetch Documentation**: API-Documentation.md
- **Best Practices**: Best-Practices.md
- **Troubleshooting**: Troubleshooting.md

Community Support

- Report migration issues on GitHub
- Ask questions in Dynamo community forums
- Share migration experiences with other users

Support for Migration Questions

For specific migration questions:

1. Include your current DynaWeb workflow
2. Describe the expected behavior
3. Show what you've tried with DynaFetch
4. Include any error messages

Acknowledgments

This migration guide is made possible by the foundational work of **Radu Gidei** and the **DynaWeb** project. DynaWeb established the patterns and concepts that made REST API integration possible in Dynamo. DynaFetch continues this legacy with modern enhancements while maintaining the core principles that made DynaWeb successful.

DynaWeb Project: https://github.com/radumg/DynaWeb
Radu Gidei: https://github.com/radumg

Thank you to the Dynamo community for supporting both DynaWeb and DynaFetch development.

Happy migrating! Welcome to modern REST APIs with DynaFetch.