Reports WebData - GraphQL and MCP Client

New API modes and verbs on WebData for 10.1.5: GraphQL endpoint consumption, MCP client sessions, and the full REST verb set.

Reference **Modules** **Reports** **UI** **WebData** GraphQL and MCP Client

---

New in 10.1.5. The **API** dropdown on the WebData grid adds GraphQL and McpHttp alongside REST, and REST gains PUT, PATCH, and DELETE verbs. Existing solutions open with `API=REST` and unchanged behavior.

A WebData row now represents one of three protocols: REST (the default), GraphQL, or an MCP server over HTTP. The protocol is set by the **API** column on the grid. Each mode exposes its own script methods. Calling a method from the wrong mode throws `InvalidOperationException`, so miswiring surfaces immediately at design and test time.

For base WebData configuration (Name, Encoding, DefaultURL, Authorization, Headers, Editor, File Operations, Padding), see Reports WebData Reference.

API Dropdown Values

Value	Use when	Script methods
REST	You consume or expose a REST API.	`GetRequestAsync`, `PostRequestAsync`, `PutRequestAsync`, `PatchRequestAsync`, `DeleteRequestAsync`.
GraphQL	You consume a GraphQL endpoint.	`QueryAsync()`, `QueryAsync(variablesJson)`.
McpHttp	You consume an MCP server over HTTP (tool discovery and tool calls).	`ListToolsAsync`, `CallAsync()`, `CallAsync(argumentsJson)`.

Mode guard: invoking a GraphQL or MCP method on a REST row, or any cross-mode combination, throws `InvalidOperationException` with the offending method name and the actual row mode.

REST Verbs (PUT, PATCH, DELETE)

In addition to the existing `GetRequestAsync` and `PostRequestAsync`, three new verb methods are available on REST rows.

PUT

Replace a resource with the Editor body.

```
await @Report.WebData.API.PutRequestAsync();
```

PATCH

Apply a partial update to a resource.

```
await @Report.WebData.API.PatchRequestAsync();
```

DELETE

Remove a resource.

```
await @Report.WebData.API.DeleteRequestAsync();
```

All three respect the row's Authorization, Headers, and Editor body exactly like POST. Response handling is identical to the existing verbs (see Reports WebData Reference).

GraphQL

Set **API** to `GraphQL`. The query text lives in the WebData Editor body. Variables are authored in the Editor and overridden at runtime when needed.

```
// Run the Editor query unchanged
string json = await @Report.WebData.GraphAPI.QueryAsync();

// Override variables at runtime
string variables = "{ \"plantId\": \"Plant1\", \"limit\": 25 }";
string json2 = await @Report.WebData.GraphAPI.QueryAsync(variables);

// Inspect the response
@Tag.GraphResponse = json2;
```

A GraphQL response with `errors[]` at HTTP 200 is returned as-is. Your script decides how to react. HTTP-level failures (non-2xx) surface as exceptions identical to REST.

GraphQL OperationName

The WebData row carries an `OperationName` column alongside `ToolName`. When set, the runtime wires it into the JSON-RPC envelope as `{query, variables, operationName}`. When null or empty, no `operationName` is sent and the server picks a default (or errors out if the query document declares more than one named operation).

Set `OperationName` when the query document defines multiple named operations and you need to pick one. A single-operation document does not need it.

The field appears on the WebData grid editor in Designer and on the Data Explorer WebData Tools Reference tab in the Export path. `{{Tag.*}}` tokens in `OperationName` resolve at runtime like any other string property.

Schema migration `2020.1.10` adds the column. Solutions opened in 10.1.5 gain the column automatically. 10.1.4 builds do not see this field.

MCP Client (HTTP)

Set **API** to `McpHttp`. The row represents an MCP client session. Configure the server URL in **DefaultURL**. The runtime opens the session on first call, reuses it for five minutes of idle time, and closes it on runtime stop or after the idle window.

```
// Discover available tools
string toolsJson = await @Report.WebData.McpServer.ListToolsAsync();
@Tag.AvailableTools = toolsJson;

// Call a tool with the Editor-body arguments
string result1 = await @Report.WebData.McpServer.CallAsync();

// Override arguments at runtime
string args = "{ \"target\": \"Plant1/Tank1\", \"mode\": \"read\" }";
string result2 = await @Report.WebData.McpServer.CallAsync(args);
```

A server returning HTTP 404 triggers automatic session re-init and one retry. Any other HTTP error surfaces as an exception. The session is disposed on `OnStop` of the Report runtime, so no background HTTP state survives across runtime restarts.

The MCP client speaks streamable-HTTP with JSON-RPC 2.0 framing. Authorization and Headers on the WebData row flow into every request, so bearer tokens and custom headers work exactly like REST.

MCP Session Handshake

Before the first `tools/list` or `tools/call`, the runtime performs the MCP `2025-03-26` handshake against the server URL:

- Send a JSON-RPC `initialize` request declaring client capabilities and protocol version.
- Capture the `Mcp-Session-Id` value from the response headers and retain it for the life of the session.
- POST a `notifications/initialized` message to acknowledge the handshake.
- Attach the `Mcp-Session-Id` header to every subsequent request. An expired session returns HTTP 404, which triggers automatic re-initialization and one retry of the original call.

See also: WebData Tools Reference — `MCP Session Handshake`. The Data Explorer tester uses the same handshake, so behavior stays symmetric between the runtime `ReportWebData` path and the design-time tester.

Migration

Existing WebData rows open with `API=REST` and unchanged behavior. No property-level migration is required. Setting the API value to GraphQL or McpHttp is a one-click change on the grid.

In this section...
