Nutanix v4 APIs

Working with VM Categories via Microsoft Excel
Using Powershell, Python and Excel for Automation

Understanding REST APIs, ExtId, Etag, Validation Workflows

October 2025
hardev@nutanix.com

NUTANIX

Learning Objectives

What We'll Cover Today

- **REST API Fundamentals** - HTTP methods, headers, status codes
- **Nutanix v4 API Architecture** - Endpoints, authentication, data models
- **Resource Identification** - ExtIds and unique identifiers
- **Concurrency Control** - ETags and optimistic locking
- **Excel-Driven Automation** - User-friendly parameter entry
- **Validation Workflows** - Pre-flight checks and error handling
- **Real-World Implementation** - PowerShell and Python scripts

> **Goal:** Build confidence in designing and implementing API-driven automation solutions that are both powerful and user-friendly.

REST API Fundamentals

REpresentational State Transfer

Core HTTP Methods

Method	Purpose	Idempotent	Example Use
GET	Retrieve data	✓ Yes	Fetch VM details
POST	Create/Action	✗ No	Associate categories
PUT	Update/Replace	✓ Yes	Update VM config
DELETE	Remove	✓ Yes	Remove categories

Key HTTP Headers

```
Authorization: Basic dXNlcm5hbWU6cGFzc3dvcmQ= Content-Type: application/json If-Match: "etag-value-here" NTNX-Request-Id: 550e8400-e29b-41d4-a716-446655440000
```

Understanding HTTP Status Codes

What Your API is Telling You

Success Codes (2xx)

- **200 OK** - GET request successful
- **201 Created** - Resource created
- **202 Accepted** - Async operation started
- **204 No Content** - Operation successful, no data returned

Client Error Codes (4xx)

- **400 Bad Request** - Invalid syntax/data
- **401 Unauthorized** - Authentication failed
- **403 Forbidden** - Access denied
- **404 Not Found** - Resource doesn't exist
- **409 Conflict** - Resource state conflict
- **412 Precondition Failed** - ETag mismatch

Server Error Codes (5xx)

- **500 Internal Server Error** - Server-side error
- **502 Bad Gateway** - Upstream error
- **503 Service Unavailable** - Temporary unavailability

> **Pro Tip:** Always handle these status codes in your automation scripts. A 412 response indicates an ETag conflict - another process may have modified the resource!

In Our Scripts:

```
if post_response and post_response.status_code == 202: print("ACCEPTED: Operation started successfully") # Extract new ETag and update Excel else: print(f"FAILED: Status code {post_response.status_code}") # Log error and update Excel with failure status
```

Nutanix v4 API Architecture

Modern, Consistent, Resource-Centric Design

Key Characteristics

- **Resource-Centric:** Each resource has a unique ExtId
- **Stateless:** Each request is independent
- **Versioned:** v4.1 ensures API stability
- **Consistent:** Similar patterns across services
- **Secure:** Built-in authentication and authorization

Our Use Case Endpoints

GET

`/vmm/v4.1/ahv/config/vms/{extId}`

Retrieve VM details and current ETag

POST

`/vmm/v4.1/ahv/config/vms/{extId}/$actions/associate-categories`

Associate categories with VM

GET

`/prism/v4.1/config/categories`

List all available categories

```
https://prism-central.example.com:9440/api/ ├── vmm/v4.1/ahv/config/vms # Virtual Machine Management ├──
prism/v4.1/config/categories # Category Configuration ├── prism/v4.1/management/clusters # Cluster Management └──
storage/v4.1/config/volumes # Storage Configuration
```

Understanding ExtIds

External Identifiers – Your Key to Resource Management

What are ExtIds?

> **ExtId (External Identifier):** A unique, immutable identifier for resources in Nutanix. Unlike names which can change, ExtIds provide a stable reference for API operations.

ExtId Characteristics

- **Unique:** Globally unique within the cluster
- **Immutable:** Never changes during resource lifetime
- **UUID Format:** Typically UUID v4 format
- **API-Friendly:** Used in all v4 API endpoints
- **Cross-Reference:** Links resources across services

Example ExtIds

```
VM ExtId: 550e8400-e29b-41d4-a716-446655440000 Category ExtId: f47ac10b-58cc-4372-a567-0e02b2c3d479
```

Why ExtIds Matter

Problem: VM named "WebServer01" could be renamed to "WebServer-Prod01"

Solution: ExtId remains constant regardless of name changes

Retrieving ExtIds

```
# PowerShell - Extract ExtId from API response $vmExtId = $vm.metadata.extId # Python - Same concept vm_extid = vm_data['metadata']['extId']
```

In Our Excel Workflow:
We populate the "VM extId" column by querying the API first, then use these stable identifiers for all subsequent operations.

Etags: Ensuring Data Consistency

Optimistic Locking in Distributed Systems

What is an ETag?

> **ETag (Entity Tag):** A unique identifier for a specific version of a resource. It changes every time the resource is modified, enabling concurrency control without locks.

Why ETags Matter

- **Prevent Lost Updates:** Two users can't overwrite each other's changes
- **Data Integrity:** Ensures you're modifying the version you think you are
- **Performance:** No need for resource locks
- **Conflict Detection:** Graceful handling of concurrent modifications

The ETag Workflow

1. **GET Request:** Retrieve resource and its current ETag
2. **Prepare Update:** Build your modification payload
3. **POST/PUT with If-Match:** Include ETag in If-Match header
4. **Server Validation:** Server checks if ETag matches current version
5. **Success or Conflict:** Operation succeeds or returns 412 if resource changed

Implementation in Our Scripts

```
# 1. GET to retrieve ETag vm_etag =
response.headers.get('ETag', '') # 2. Prepare headers
for POST post_headers = { 'Authorization': auth_header,
'If-Match': vm_etag, 'NTNX-Request-Id':
str(uuid.uuid4()), 'Content-Type': 'application/json' }
# 3. POST with ETag validation response =
requests.post(url, headers=post_headers, json=payload)
```

Excel as User Interface

Making API Automation User-Friendly

Why Excel?

- **Familiar Interface:** Everyone knows Excel
- **Data Validation:** Built-in dropdown lists, formulas
- **Bulk Operations:** Process hundreds of VMs at once
- **Visual Feedback:** Color-coded status updates
- **Audit Trail:** Permanent record of changes
- **Easy Review:** Stakeholder approval before execution

Our Workbook Structure

📊 **ToUpdate Sheet:**
- VM Name & ExtId
- Categories to assign
- Validation status
- Execution results

📋 **VMCategories Sheet:**
- All VMs and current categories
- Reference for existing assignments

🏷️ **AllCategories Sheet:**
- Master list of available categories
- Category names and ExtIds

Data Validation Features

Validation Type	Implementation	Benefit
Category Names	Dropdown from AllCategories sheet	Prevents typos, ensures valid categories
Format Checking	PowerShell validation: "key=value,key=value"	Catches syntax errors before API calls
VM Existence	Cross-reference with VMCategories sheet	Ensures VM ExtIds are valid
Status Tracking	Color-coded cells (Green=OK, Red=Error)	Visual feedback on validation results

Complete Automation Workflow

From Data Collection to Execution

Phase 1: Data Discovery & Preparation

1. **list_vms.ps1:** Query Nutanix API for all VMs, extract names and ExtIds
2. **list_categories.ps1:** Fetch all available categories and their ExtIds
3. **build_workbook.ps1:** Create Excel workbook with current VM-category mappings

Phase 2: User Input & Validation

4. **User edits Excel:** Specify VMs and desired category assignments
5. **update_vm_categories.ps1:** Validate all entries against API data
   - Verify VM ExtIds exist and are unique
   - Validate category names match available categories
   - Check syntax of category assignments
   - Update Excel with validation results

Phase 3: API Execution

6. **update_categories_for_vm.py:** Execute API calls for validated entries
   - For each VM with "OK" validation status:
   - GET VM details and extract current ETag
   - POST category associations with ETag in If-Match header
   - Update Excel with execution results and timestamps

Script Implementation Details

Powershell and Python Working Together

PowerShell Scripts

```
# Basic Auth Header Creation $base64Auth =
[System.Convert]::ToBase64String(
[System.Text.Encoding]::ASCII.GetBytes(
"$($username):$($password)" ) ) $headers = @{
Authorization = "Basic $base64Auth" } # API Call with
Error Handling $response = Invoke-RestMethod -Uri $uri
-Method Get -Headers $headers -SkipCertificateCheck
```

Key PowerShell Features Used:

- **ImportExcel Module:** Excel file manipulation
- **Invoke-RestMethod:** HTTP API calls
- **ConvertTo-Json/From-Json:** Data serialization
- **Hash Tables:** Efficient data lookups

Python Implementation

```
# Authentication Header credentials = f"{username}:
{password}" encoded =
b64encode(credentials.encode('ascii')) auth_header =
f'Basic {encoded.decode('ascii')}' # ETag-aware POST
Request post_headers = { 'Authorization': auth_header,
'If-Match': vm_etag, 'NTNX-Request-Id':
str(uuid.uuid4()), 'Content-Type': 'application/json' }
response = requests.post(url, headers=post_headers,
json=payload)
```

Key Python Libraries:

- **requests:** HTTP client for API calls
- **pandas:** Excel file reading/writing
- **openpyxl:** Excel formatting and styles
- **uuid:** Generate request IDs

Design Choice: PowerShell excels at Windows integration and Excel manipulation, while Python provides robust HTTP handling and data processing. Using both languages leverages their respective strengths.

Robust Error Handling

Planning for the Unexpected

Multi-Layer Validation Strategy

Pre-Flight Validation (PowerShell)

- **Syntax Checking:** Verify "key=value" format for categories
- **Reference Validation:** Ensure VM ExtIds exist in current data
- **Uniqueness Checks:** Detect duplicate VM entries
- **Category Validation:** Verify category names exist in master list

Recovery Strategies

Automatic Retry: Implement exponential backoff for transient failures

Partial Success: Continue processing other VMs even if some fail

Clear Feedback: Update Excel with specific error codes and timestamps

Audit Trail: Log all API interactions for troubleshooting

Common Error Scenarios

- **Network Issues:** Timeouts, connection failures
- **Authentication:** Invalid credentials, expired tokens
- **Resource Changes:** VM deleted, categories modified
- **Concurrency:** ETag conflicts from parallel operations
- **Input Errors:** Malformed data, missing fields

Runtime Error Handling (Python)

```
try: response = requests.post(url, headers=headers, json=payload, timeout=30) if response.status_code == 202: # Success - update Excel with green status update_excel_status(file, sheet, row, 'ACCEPTED', timestamp) elif response.status_code == 412: # ETag conflict - resource was modified by another process update_excel_status(file, sheet, row, 'CONFLICT', timestamp) else: # Other error - log details for troubleshooting update_excel_status(file, sheet, row, f'ERROR {response.status_code}', timestamp) except requests.exceptions.Timeout: update_excel_status(file, sheet, row, 'TIMEOUT', timestamp) except requests.exceptions.ConnectionError: update_excel_status(file, sheet, row, 'CONNECTION_ERROR', timestamp)
```

Best Practices & Key Takeaways

Building Production-Ready API Automation

API Design Principles

- **Always use ETags:** Prevent concurrent modification issues
- **Include Request IDs:** Enable request tracing and debugging
- **Handle all status codes:** Plan for success and failure scenarios
- **Implement timeouts:** Don't let requests hang indefinitely
- **Use HTTPS:** Secure all API communications
- **Validate inputs early:** Catch errors before making API calls

User Experience

- **Familiar interfaces:** Excel is universally understood
- **Visual feedback:** Color-coded status indicators
- **Dry-run capability:** Preview changes before execution
- **Clear documentation:** Examples and error explanations

Production Considerations

Security

- Store credentials securely (avoid hardcoding)
- Use service accounts with minimum required permissions
- Enable audit logging for all operations

Scalability

- Implement rate limiting to avoid API throttling
- Use batch operations where available
- Consider parallel processing for large datasets

Monitoring

- Log all API interactions with timestamps
- Track success/failure rates
- Set up alerts for abnormal patterns

🎯 Key Takeaway

Modern API automation succeeds when it combines technical robustness with user-friendly interfaces. By understanding REST fundamentals, leveraging ExtIds and ETags properly, and providing familiar tools like Excel for user interaction, we can build automation that is both powerful and accessible to non-technical users.

Example: Using a Public REST API

Let's use a public, free API to see this in action. The **Open-Meteo API** is a great, simple example that provides weather data.

The Goal: Get the current temperature for London, UK.

1. **Find the Endpoint:** An endpoint is the specific URL for a resource. The Open-Meteo endpoint for a current weather forecast is a URL with various parameters to specify the location.
    1. `https://api.open-meteo.com/v1/forecast?latitude=51.5072&longitude=-0.1276&current=temperature_2m`
2. **Make a GET Request:** Using the GET method on this URL tells the server, "Give me the data at this location."
3. **Get the Response:** The server will send back data, usually in **JSON (JavaScript Object Notation)**format. JSON is a lightweight data-interchange format that's easy for both humans and computers to read.
4. **Extract the Data:** From the JSON response, you can extract the specific information you need, which is the current temperature. In this case, you would look at the current object and find the value for `temperature_2m`, which is 16.6 degrees Celsius.

JSON Response


```
{
  "latitude": 51.5,
  "longitude": -0.12,
  "generationtime_ms": 0.444,
  "utc_offset_seconds": 0,
  "timezone": "Europe/London",
  "timezone_abbreviation": "GMT",
  "elevation": 45,
  "current_units": {
    "time": "iso8601",
    "interval": "seconds",
    "temperature_2m": "°C"
  },
  "current": {
    "time": "2025-09-17T11:00",
    "interval": 900,
    "temperature_2m": 16.6
  }
}
```

The Postman Tool

Making API requests directly in a web browser is easy for GET requests, but it's not possible for more complex requests like POST, PUT, or DELETE. This is where a tool like Postman is essential.

What is Postman?

Postman is a powerful, user-friendly API platform that simplifies every step of the API lifecycle. It allows you to build, test, document, and share API requests without writing any code.

How to Use Postman for the Open-Meteo Example:

1. Open Postman: Launch the Postman application.
2. Create a New Request: Click the "New" button and select "HTTP Request."
3. Set the Method: In the dropdown menu next to the URL bar, ensure the method is set to GET.
4. Enter the URL: Paste the Open-Meteo API endpoint URL into the URL bar.

   `https://api.open-meteo.com/v1/forecast?latitude=51.5072&longitude=-0.1276&current=temperature_2m`
5. Send the Request: Click the Send button.
6. View the Response: The response will appear in the pane below. You will see the JSON data, the HTTP status code (e.g., 200 OK), and other response details.

Postman makes it easy to add headers, body data for POST requests, and authentication, making it an invaluable tool for any developer or IT professional working with APIs.

Postman Screenshot