Error Handling in API Programming

Introduction

When working with external APIs, numerous things can go wrong:

- Network connectivity issues
- Server-side errors
- Authentication failures
- Rate limiting
- Timeout issues
- Invalid requests

Implementing proper error handling ensures your application remains stable and provides meaningful feedback when issues occur.

Error Handling with the Requests Library

Basic Exception Handling

```python
import requests

def get_data_basic(url):
    try:
        response = requests.get(url)
        response.raise_for_status()  # Raises exception for 4XX/5XX responses
        return response.json()
    except requests.exceptions.HTTPError as errh:
        print(f"HTTP Error: {errh}")
    except requests.exceptions.ConnectionError as errc:
        print(f"Connection Error: {errc}")
    except requests.exceptions.Timeout as errt:
        print(f"Timeout Error: {errt}")
    except requests.exceptions.RequestException as err:
        print(f"Request Exception: {err}")
    except ValueError as errv:
        print(f"JSON Parsing Error: {errv}")
    return None
```

Common Exception Types in Requests

Requests provides several exception types:

- `requests.exceptions.HTTPError`: Raised for 4XX/5XX responses when using `raise_for_status()`
- `requests.exceptions.ConnectionError`: Failed connection to the server
- `requests.exceptions.Timeout`: Request timed out
- `requests.exceptions.TooManyRedirects`: Too many redirects
- `requests.exceptions.RequestException`: Base exception for all requests-related errors

Handling Different HTTP Status Codes

```
import requests

def get_data_with_status_handling(url, headers=None):
    try:
        response = requests.get(url, headers=headers)
        # Handle different status codes
        if response.status_code == 200:
            return response.json()
        elif response.status_code == 400:
            print("Bad Request: The server couldn't understand the request")
        elif response.status_code == 401:
            print("Unauthorized: Authentication is required")
        elif response.status_code == 403:
            print("Forbidden: You don't have permission to access this resource")
        elif response.status_code == 404:
            print("Not Found: The requested resource was not found")
        elif response.status_code == 429:
            print("Too Many Requests: You've exceeded the rate limit")
        elif response.status_code >= 500:
            print(f"Server Error: The server returned status code {response.status_code}")
        else:
            print(f"Unexpected status code: {response.status_code}")

        # You might want to examine the response body for error details
        try:
            error_details = response.json()
            print(f"Error details: {error_details}")
        except ValueError:
            print(f"Response text: {response.text[:200]}")  # Print first 200 chars of
response

        return None
    except requests.exceptions.RequestException as e:
        print(f"Request failed: {e}")
        return None
```

Handling Timeouts

```
import requests

def get_data_with_timeout(url, timeout=5):
    try:
        # Set both connection and read timeout to 5 seconds
        response = requests.get(url, timeout=timeout)
        response.raise_for_status()
        return response.json()
    except requests.exceptions.Timeout:
        print(f"Request timed out after {timeout} seconds")
    except requests.exceptions.RequestException as e:
        print(f"Request failed: {e}")
    return None

# Or use a tuple for different connect and read timeouts
def get_data_with_custom_timeouts(url, connect_timeout=3, read_timeout=10):
    try:
        # First value is connect timeout, second is read timeout
        response = requests.get(url, timeout=(connect_timeout, read_timeout))
        response.raise_for_status()
        return response.json()
    except requests.exceptions.Timeout:
        print(f"Request timed out (connect: {connect_timeout}s, read: {read_timeout}s)")
    except requests.exceptions.RequestException as e:
        print(f"Request failed: {e}")
    return None
```

Handling Retries

```
import requests
from requests.adapters import HTTPAdapter
from urllib3.util.retry import Retry

def get_with_retry(url, max_retries=3, backoff_factor=0.3,
                  status_forcelist=(500, 502, 503, 504)):
    session = requests.Session()

    # Configure retry strategy
    retry_strategy = Retry(
        total=max_retries,
        backoff_factor=backoff_factor,  # Wait 0.3, 0.6, 1.2 seconds between retries
        status_forcelist=status_forcelist,  # Retry on these status codes
        allowed_methods=["GET", "POST", "PUT", "DELETE", "HEAD", "OPTIONS"]  # Methods to
retry
    )

    # Mount the adapter with our retry strategy for both http and https
    adapter = HTTPAdapter(max_retries=retry_strategy)
    session.mount("http://", adapter)
    session.mount("https://", adapter)

    try:
        response = session.get(url)
        response.raise_for_status()
        return response.json()
    except requests.exceptions.RequestException as e:
        print(f"All retries failed: {e}")
        return None
```

Error Handling with HTTPX Library

HTTPX is a modern, fully featured HTTP client for Python that provides sync and async APIs with support for HTTP/2.

Basic Error Handling

```
import httpx

def get_data_with_httpx(url):
    try:
        with httpx.Client() as client:
            response = client.get(url)
            response.raise_for_status()
            return response.json()
    except httpx.HTTPStatusError as e:
        print(f"HTTP Status Error: {e}")
    except httpx.RequestError as e:
        print(f"Request Error: {e}")
    except ValueError as e:
        print(f"JSON Parsing Error: {e}")
    return None
```

Common Exception Types in HTTPX

HTTPX provides a more streamlined exception hierarchy:

- `httpx.HTTPStatusError`: Raised for 4XX/5XX responses when using `raise_for_status()`
- `httpx.RequestError`: Base class for request-related errors
  - `httpx.ConnectError`: Connection failed
  - `httpx.ReadTimeout`: Timeout reading the response
  - `httpx.WriteTimeout`: Timeout writing the request
  - `httpx.ConnectTimeout`: Timeout establishing connection
  - `httpx.NetworkError`: Network connection error
  - `httpx.TooManyRedirects`: Too many redirects

Handling Different HTTP Status Codes with HTTPX

```
import httpx

def get_data_with_httpx_status_handling(url, headers=None):
    try:
        with httpx.Client() as client:
            response = client.get(url, headers=headers)

            # Handle different status codes
            if response.status_code == 200:
                return response.json()
            elif response.status_code == 400:
                print("Bad Request: The server couldn't understand the request")
            elif response.status_code == 401:
                print("Unauthorized: Authentication is required")
            elif response.status_code == 403:
                print("Forbidden: You don't have permission to access this resource")
            elif response.status_code == 404:
                print("Not Found: The requested resource was not found")
            elif response.status_code == 429:
                print("Too Many Requests: You've exceeded the rate limit")
            elif response.status_code >= 500:
                print(f"Server Error: The server returned status code
{response.status_code}")
            else:
                print(f"Unexpected status code: {response.status_code}")

            # Examine the response body for error details
            try:
                error_details = response.json()
                print(f"Error details: {error_details}")
            except ValueError:
                print(f"Response text: {response.text[:200]}")  # Print first 200 chars

            return None
    except httpx.RequestError as e:
        print(f"Request failed: {e}")
        return None
```

Handling Timeouts with HTTPX

```
import httpx

def get_data_with_httpx_timeout(url, timeout=5.0):
    try:
        # HTTPX uses a single timeout value by default, unlike requests
        with httpx.Client(timeout=timeout) as client:
            response = client.get(url)
            response.raise_for_status()
            return response.json()
    except httpx.TimeoutException:
        print(f"Request timed out after {timeout} seconds")
    except httpx.RequestError as e:
        print(f"Request failed: {e}")
    return None

# Or use specific timeout configurations
def get_data_with_httpx_custom_timeouts(url):
    # Create a specific timeout configuration
    timeout = httpx.Timeout(
        connect=3.0,  # connection timeout
        read=5.0,     # read timeout
        write=5.0,    # write timeout
        pool=2.0      # pool timeout
    )

    try:
        with httpx.Client(timeout=timeout) as client:
            response = client.get(url)
            response.raise_for_status()
            return response.json()
    except httpx.TimeoutException as e:
        print(f"Request timed out: {e}")
    except httpx.RequestError as e:
        print(f"Request failed: {e}")
    return None
```

Handling Retries with HTTPX

```
import httpx
import time

def get_with_httpx_retry(url, max_retries=3, backoff_factor=0.3):
    """Manual retry implementation"""
    retries = 0
    last_exception = None
    while retries <= max_retries:
        try:
            with httpx.Client() as client:
                response = client.get(url)
                response.raise_for_status()
                return response.json()
        except (httpx.HTTPStatusError, httpx.RequestError) as e:
            last_exception = e
            retries += 1

            # Only retry on server errors and connection errors
            if isinstance(e, httpx.HTTPStatusError) and e.response.status_code < 500:
                # Don't retry client errors (except 429)
                if e.response.status_code != 429:
                    break

            if retries <= max_retries:
                sleep_time = backoff_factor * (2 ** (retries - 1))
                print(f"Retry {retries}/{max_retries} after {sleep_time:.2f} seconds")
                time.sleep(sleep_time)
            else:
                print(f"All retries failed. Last error: {e}")

    if last_exception:
        print(f"Request failed after {max_retries} retries: {last_exception}")
    return None
```

Async Error Handling with HTTPX

One of the main advantages of HTTPX is its support for async/await:

```
import httpx
import asyncio

async def get_data_async(url):
    try:
        async with httpx.AsyncClient() as client:
            response = await client.get(url)
            response.raise_for_status()
            return response.json()
    except httpx.HTTPStatusError as e:
        print(f"HTTP Status Error: {e}")
    except httpx.RequestError as e:
        print(f"Request Error: {e}")
    return None

# Example usage in an async function
async def main():
    # Fetch multiple URLs concurrently
    urls = [
        "https://api.example.com/data/1",
        "https://api.example.com/data/2",
        "https://api.example.com/data/3",
    ]

    tasks = [get_data_async(url) for url in urls]
    results = await asyncio.gather(*tasks, return_exceptions=True)
    # Process results (handling any exceptions)
    for i, result in enumerate(results):
        if isinstance(result, Exception):
            print(f"Request {i} failed with error: {result}")
        else:
            print(f"Request {i} succeeded: {result}")

# Run the async main function
if __name__ == "__main__":
    asyncio.run(main())
```

Comparison: Requests vs HTTPX

Similarities

- Both provide similar core functionality
- Both use similar methods like `get()`, `post()`, etc.
- Both have `raise_for_status()` for HTTP error handling
- Both support request and response hooks

Key Differences

- **Async Support**: HTTPX supports async/await, Requests is synchronous only
- **HTTP/2**: HTTPX supports HTTP/2, Requests only supports HTTP/1.1
- **Exception Hierarchy**: HTTPX has a more streamlined exception hierarchy
- **Timeouts**: HTTPX uses a single timeout value by default, Requests uses a tuple for connect and read timeouts
- **Transport Layer**: HTTPX allows custom transport implementations

When to Choose Which

- Use Requests for:
  - Simpler applications where async isn't needed
  - When you need maximum compatibility with existing code
  - When you require the most mature and stable library
- Use HTTPX for:
  - Applications that need async capabilities
  - When you need HTTP/2 support
  - Modern applications that benefit from newer features

Best Practices for API Error Handling

1. **Always handle exceptions**: Never leave API calls without proper exception handling
2. **Use specific exception types**: Catch specific exceptions before general ones
3. **Implement retries with backoff**: Use exponential backoff for retries
4. **Log all exceptions**: Include request details in logs
5. **Provide meaningful error messages**: Parse API error responses when available
6. **Set appropriate timeouts**: Never use infinite timeouts
7. **Use context managers**: Ensure resources are properly cleaned up
8. **Consider rate limiting**: Implement throttling when needed

9. **Validate responses**: Check that responses match expected schema
10. **Include idempotency keys**: For operations that should not be repeated

Conclusion

Effective error handling is essential for building reliable applications that interact with APIs. Both `requests` and `httpx` provide solid foundations for handling errors, with `httpx` offering additional modern features like async support and HTTP/2.

By implementing proper error handling, you can build robust applications that gracefully handle network issues, server errors, and other common API problems.