Module 1.4: Policy Scopes & Configuration Fundamentals

Understanding Azure API Management's powerful policy engine and how to strategically apply policies at different scopes to control API behavior.

Module Overview

Mastering the APIM Policy Model

Azure API Management policies are the heart of API behavior customization. They provide declarative configuration for everything from security enforcement to request transformation, rate limiting to response caching.

In this module, we'll explore the policy model architecture, understand where policies can be applied across different scopes, and see practical examples of common policy implementations. By the end, you'll be equipped to design sophisticated policy strategies for your APIs.

Learning Objectives

- Understand XML-based policy documents
- Master policy scope hierarchy
- Apply common policy patterns
- Preview transformation capabilities

What Are APIM Policies?

Policies are collections of statements executed sequentially on API requests and responses. They're defined using XML documents and allow you to modify API behavior without changing backend code. Think of policies as middleware that intercepts every API call flowing through your gateway.

Policies enable critical capabilities like authentication, rate limiting, transformation, caching, and routing. They execute at specific points in the request-response pipeline, giving you precise control over how your APIs behave in production environments.

The XML-Based Policy Document Structure

Inbound

Executes before the request is forwarded to the backend service. Perfect for authentication, rate limiting, and request transformation.

Backend

Controls how requests are forwarded to backend services. Handles routing, retry logic, and backend service selection.

Outbound

Processes responses before returning to the client. Used for response transformation, header manipulation, and caching.

On-Error

Executes when errors occur in any section. Enables custom error handling, logging, and graceful failure responses.

Policy Document Example

```
<policies>
    <inbound>
        <base />
        <set-header name="X-Forwarded-For" exists-action="override">
            <value>@(context.Request.IpAddress)</value>
        </set-header>
        <rate-limit calls="100" renewal-period="60" />
    </inbound>
    <backend>
        <base />
    </backend>
    <outbound>
        <base />
        <set-header name="X-Powered-By" exists-action="delete" />
    </outbound>
    <on-error>
        <base />
    </on-error>
</policies>
```

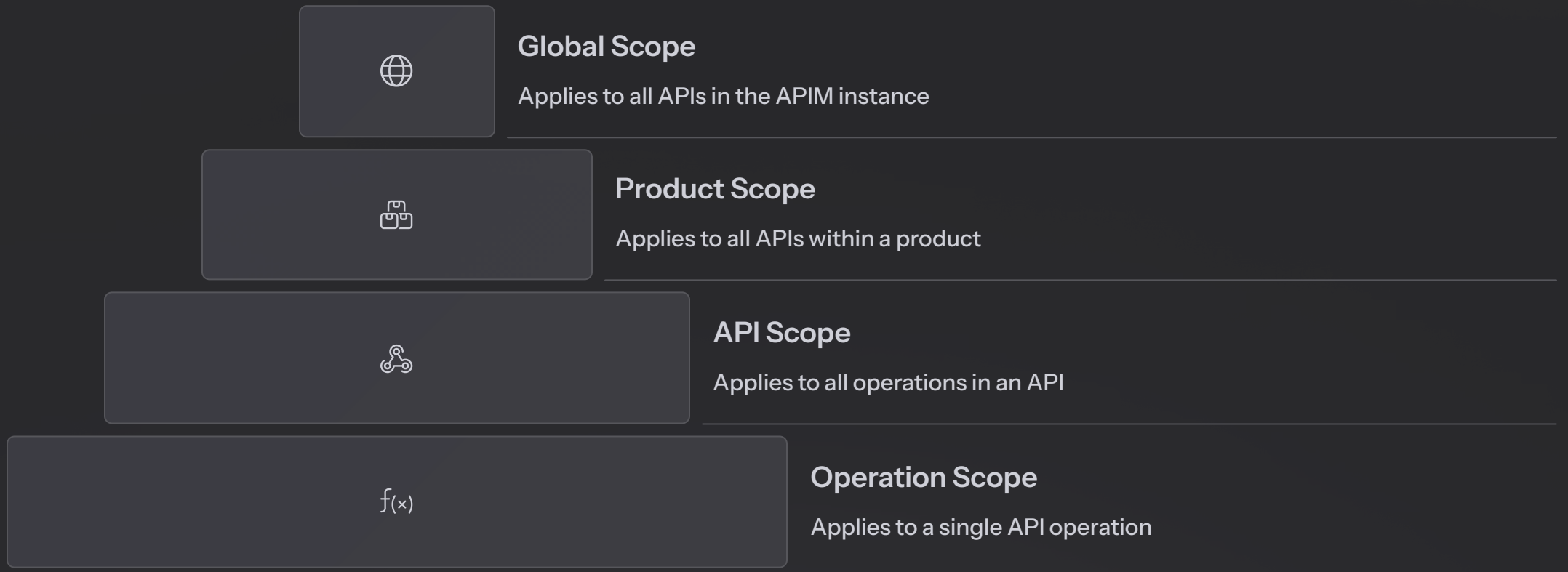
This example shows the four-section structure. The `<base />` element is critical—it indicates where parent scope policies are inherited and executed in the pipeline.

Understanding Policy Scopes

APIM policies can be defined at multiple levels, creating a hierarchical inheritance model. Each scope builds upon the previous, allowing you to set global defaults while customizing specific APIs or operations. This architecture promotes consistency while enabling granular control where needed.

The scope hierarchy determines policy execution order and inheritance behavior. Understanding this hierarchy is essential for designing maintainable, scalable API management strategies.

The Four Policy Scopes



Global Scope Policies

Global policies apply to every API call flowing through your APIM instance, regardless of which API or operation is invoked. This scope is ideal for cross-cutting concerns that should be universally enforced.

Common use cases include:

- Organization-wide security headers
- Global rate limiting or throttling
- Standard logging and telemetry
- CORS configuration for all APIs
- IP filtering and access control

Global policies execute first in the inbound section and last in the outbound section, forming the outer layer of your policy pipeline.

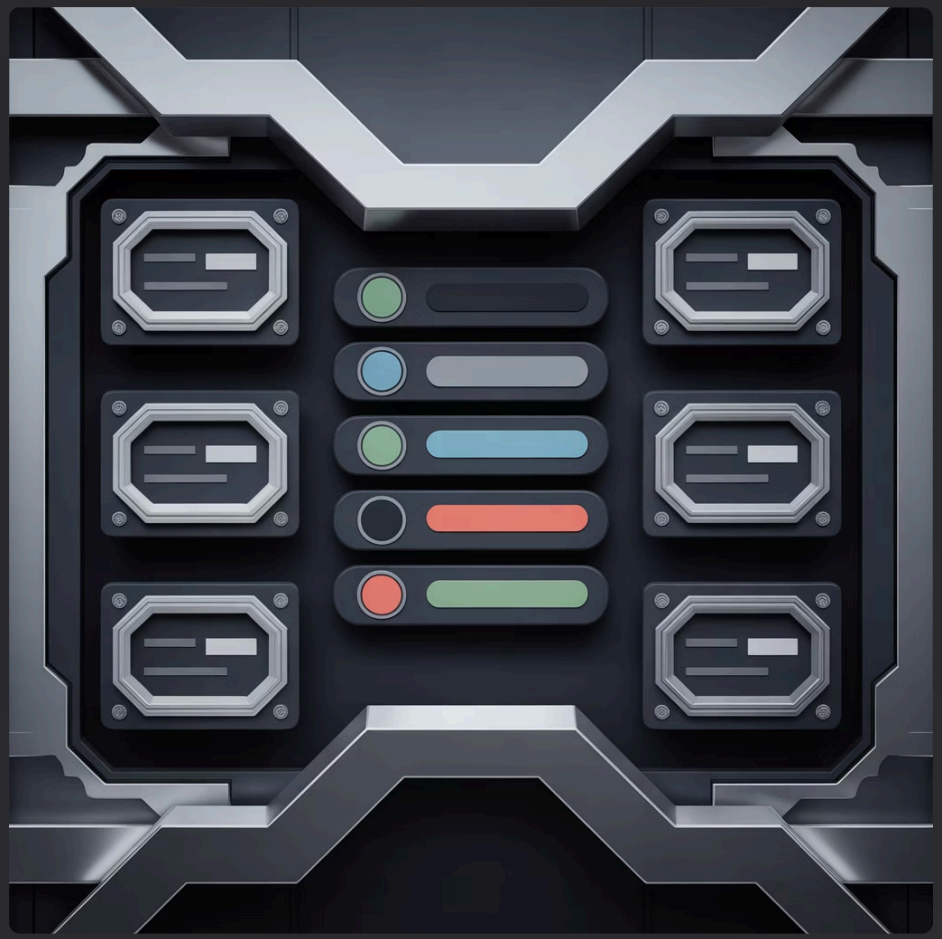
Product Scope Policies

Products in APIM are collections of APIs grouped for subscription and access control. Product-level policies apply to all APIs within that product, enabling you to enforce product-specific rules without duplicating configuration across individual APIs.

This scope is particularly valuable when you have different products for internal vs. external consumers, or when you offer tiered service levels. You can apply different rate limits, authentication requirements, or transformation rules based on the product subscription.

For example, a "Premium" product might have higher rate limits and additional features enabled through policies, while a "Free Trial" product has more restrictions.

API Scope Policies



API-level policies apply to all operations within a specific API. This scope is perfect for API-specific configurations that should be consistent across all endpoints.

Typical API-level policies:

- API-specific authentication schemes
- Backend service URL configuration
- API-wide request/response transformations
- Caching strategies for the entire API
- Mock responses during development

This level provides the right balance between broad application and specific control for most scenarios.

Operation Scope Policies

Operation-level policies provide the most granular control, applying only to a specific HTTP method and path combination. This is where you implement endpoint-specific behavior that differs from your API defaults.

Use operation policies when individual endpoints have unique requirements—perhaps one operation needs request validation while others don't, or a specific endpoint requires different rate limits due to its resource intensity. Operation policies override all parent scopes, giving you ultimate flexibility for edge cases.

However, excessive use of operation-level policies can create maintenance challenges. Balance granular control with maintainability by using higher scopes for common patterns.

Policy Inheritance & Execution Flow

1

Global Inbound

First policies executed on incoming requests

2

Product Inbound

Product-specific inbound processing

3

API Inbound

API-level inbound policies execute

4

Operation Inbound

Most specific inbound policies run last

5

Backend Processing

Request forwarded to backend service

6

Outbound (Reverse Order)

Operation → API → Product → Global

The Critical Role of the Base Element

The `<base />` element is one of the most important concepts in APIM policy configuration. It determines where parent scope policies are inherited and executed within the current policy document.

Placing `<base />` at the beginning of a section means parent policies execute first, then your custom policies. Placing it at the end means your policies execute first, then parent policies. Omitting it entirely prevents parent policy inheritance—use with extreme caution.

```
<inbound>
  <base />
  <!-- Parent policies above -->
  <set-header name="Custom"
    exists-action="override">
    <value>MyValue</value>
  </set-header>
  <!-- Your policies below -->
</inbound>
```

Strategic placement of the base element gives you precise control over policy execution order, which is critical when policies depend on each other or when order affects security.

Common Policy Categories

Access Restriction

Authentication, authorization, IP filtering, rate limiting, quota enforcement, and subscription key validation.

Transformation

Request/response body modification, header manipulation, URL rewriting, and format conversion (JSON ↔️ XML).

Caching

Response caching, cache lookup, cache storage, and cache invalidation for improved performance.

Routing & Backend

Backend service selection, load balancing, retry logic, circuit breakers, and mock responses.

Observability

Logging to Application Insights, custom metrics, tracing, and diagnostic information capture.

Cross-Cutting

CORS configuration, JSONP support, response compression, and custom error responses.

Policy Example 1: Header Manipulation

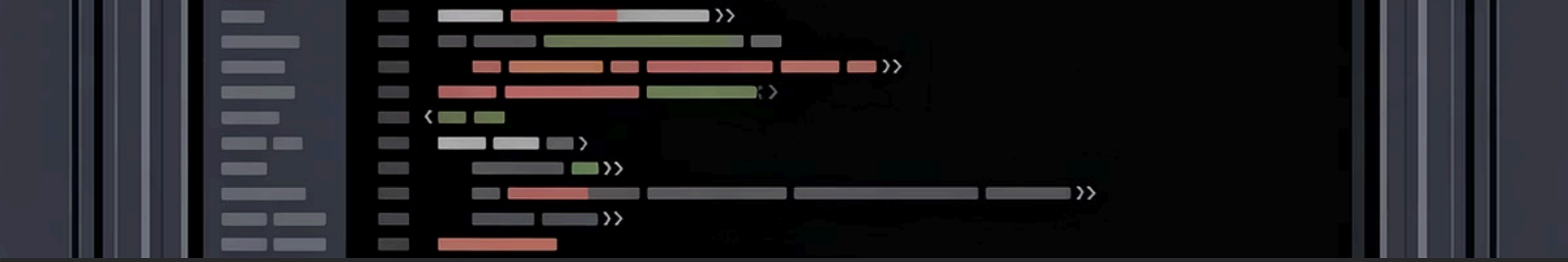
Header manipulation is one of the most common policy operations. The `set-header` policy allows you to add, modify, or remove HTTP headers on requests and responses. This is essential for passing context to backends, removing sensitive information, or adding standard headers for compliance.

```
<inbound>
    <base />
    <!-- Add client IP address for backend logging -->
    <set-header name="X-Forwarded-For" exists-action="override">
        <value>@(context.Request.IpAddress)</value>
    </set-header>

    <!-- Add correlation ID for distributed tracing -->
    <set-header name="X-Correlation-ID" exists-action="skip">
        <value>@(Guid.NewGuid().ToString())</value>
    </set-header>
</inbound>

<outbound>
    <base />
    <!-- Remove server header for security -->
    <set-header name="Server" exists-action="delete" />
</outbound>
```

The `exists-action` attribute controls behavior when the header already exists: `override` replaces it, `skip` leaves it alone, `append` adds another value, and `delete` removes it entirely.



Understanding Policy Expressions

The `@(...)` syntax in APIM policies denotes policy expressions—C# code snippets that execute at runtime. Expressions provide dynamic policy behavior based on request context, allowing you to access request properties, perform calculations, and make decisions.

Common expression uses include extracting values from the current request, generating unique identifiers, performing string manipulation, and implementing conditional logic. The `context` object provides access to the entire request/response pipeline, including headers, body, user identity, subscription information, and more.

Expressions make policies incredibly powerful, transforming them from static configuration into dynamic, intelligent middleware capable of complex decision-making and transformation logic.

Policy Example 2: Backend Service Routing

The `set-backend-service` policy dynamically controls which backend service receives the request. This enables sophisticated routing scenarios like blue-green deployments, A/B testing, canary releases, or routing based on request characteristics.

You can route to different backends based on headers, query parameters, user identity, subscription tier, or any other contextual information available in policy expressions.

```
<inbound>
  <base />
  <choose>
    <when
condition="@(context.Request.Headers.GetValueOrDefault('X-
Beta-User','false') == 'true')">
      <set-backend-service
        base-url="https://beta-api.example.com" />
    </when>
    <otherwise>
      <set-backend-service
        base-url="https://api.example.com" />
    </otherwise>
  </choose>
</inbound>
```

This example routes beta users to a separate backend service, enabling safe testing of new features before broad rollout.

Policy Example 3: URL Path Rewriting

URL rewriting allows you to transform the request path before forwarding to the backend. This is useful when your APIM API surface differs from your actual backend structure, enabling you to present a clean, versioned API facade while maintaining flexibility in backend implementation.

```
<inbound>
    <base />
    <!-- Rewrite /v2/users/{id} to /api/v2/customers/{id} -->
    <rewrite-uri template="/api/v2/customers/{id}" copy-unmatched-params="true" />

    <!-- Or use set-backend-service with URL modification -->
    <set-backend-service
        base-url="https://backend.example.com"
        path="/legacy-api/@(context.Request.MatchedParameters['id'])" />
</inbound>
```

The `rewrite-uri` policy preserves your API contract while mapping to different backend endpoints. The `copy-unmatched-params` attribute ensures query parameters are preserved during rewriting. This separation of concerns allows backend teams to refactor without breaking client contracts.

Request Transformation Example

APIM policies can transform request bodies between formats, add or remove properties, or completely restructure payloads. This is powerful for adapting client requests to match backend expectations without requiring client changes.

```
<inbound>
    <base />
    <!-- Transform JSON request by adding metadata -->
    <set-body>@{
        var body = context.Request.Body.As<JObject>(preserveContent: true);
        body.Add("requestTimestamp", DateTime.UtcNow.ToString("o"));
        body.Add("apiVersion", "2.0");
        body.Add("clientIP", context.Request.IpAddress);
        return body.ToString();
    }</set-body>
</inbound>
```

This example enriches incoming requests with metadata before forwarding to the backend, enabling better auditing and tracking without client-side changes.

Response Transformation Example

Outbound transformation policies modify responses before returning to clients. Common scenarios include filtering sensitive data, adding computed fields, converting formats, or normalizing responses from multiple backends.

```
<outbound>
 <base />
 <set-body>@{
 var response = context.Response.Body.As<JObject>();
 // Remove internal fields
 response.Remove("internalId");
 response.Remove("auditLog");
 // Add HATEOAS links
 response.Add("_links", new JObject(
 new JProperty("self",
 context.Request.Url.ToString())
 ));
 return response.ToString();
 }</set-body>
</outbound>
```

Conditional Policy Execution

The `choose` policy enables conditional logic, allowing different policy paths based on runtime conditions. This is essential for implementing sophisticated routing, transformation, and access control logic that adapts to request characteristics.

```
<inbound>
    <base />
    <choose>
        <when condition="@(context.Request.Method == 'POST')">
            <validate-content unspecified-content-type-action="prevent"
                max-size="1024" errors-variable-name="validationErrors">
                <content type="application/json"
                    validate-as="json" schema-id="post-schema" />
            </validate-content>
        </when>
        <when condition="@(context.Request.Headers.GetValueOrDefault('Authorization','').StartsWith('Bearer '))">
            <validate-jwt header-name="Authorization">
                <issuer-signing-keys>
                    <key>@(context.Deployment.Certificates["jwt-cert"].Thumbprint)</key>
                </issuer-signing-keys>
            </validate-jwt>
        </when>
        <otherwise>
            <return-response>
                <set-status code="401" reason="Unauthorized" />
            </return-response>
        </otherwise>
    </choose>
</inbound>
```

This example demonstrates validation for POST requests and JWT authentication for requests with bearer tokens, rejecting all other scenarios.

Policy Testing Strategy

01

Start at Operation Level

Develop and test new policies at the most specific scope first to limit blast radius during development.

02

Use Test API for Validation

Create a dedicated test API to validate policy behavior without impacting production traffic.

03

Leverage Trace Feature

Enable request tracing in APIM to see exactly which policies execute and their effects on requests.

04

Test Error Scenarios

Validate on-error policies by deliberately triggering errors and verifying your error handling works correctly.

05

Graduate to Higher Scopes

Once validated, move policies to API or global scope to apply consistently across multiple operations.

Common Policy Pitfalls to Avoid

Missing Base Element

Forgetting `<base />` breaks policy inheritance, preventing parent scope policies from executing. This can disable critical security or logging policies.

Policy Order Dependencies

Some policies must execute in specific order. For example, JWT validation must occur before using claims from the token in other policies.

Performance Impact

Complex transformations and external callouts in policies add latency. Always measure performance impact and consider caching strategies.

Error Handling Gaps

Not implementing on-error policies leads to poor error experiences. Always provide meaningful error responses and log failures appropriately.

Scope Overuse

Too many operation-level policies creates maintenance nightmares. Use higher scopes for common patterns and operations only for exceptions.

Best Practices for Policy Design

Design Principles

- Use higher scopes for common patterns and lower scopes for exceptions
- Keep policies simple and focused on single responsibilities
- Always include comprehensive on-error handling
- Document complex policy expressions with XML comments
- Test policies thoroughly in non-production environments first

Performance Considerations

- Minimize body transformations—they're expensive operations
- Cache frequently accessed values using named values
- Use fragment policies to reuse common policy blocks
- Consider backend changes instead of complex transformations
- Monitor policy execution time using Application Insights

Preparing for Advanced Policy Work

You now understand the foundational concepts of APIM policies: the XML document structure with its four sections, the hierarchical scope model from global to operation level, and how policies inherit and execute through the base element mechanism.

We've explored practical examples of common policy categories including header manipulation, backend routing, and basic request/response transformations. These building blocks form the foundation for the sophisticated policy implementations we'll tackle on Day 2.

1

Today's Foundation

Policy model, scopes, and basic examples

2

Tomorrow's Deep Dive

Advanced transformations, security policies, integration patterns, and production-grade implementations

Take time to experiment with these basic policy patterns in your test environment. Understanding scope inheritance and execution order now will make advanced topics much easier to grasp.

Lab

Thank You for Attending!