API-X Patterns

Preamble

This document presents a biased view of the API-X core as a router of messages between extensions, and coordinating the actors in an HTTP exchange. This document is not an attempt to declare the roles and responsibilities by fiat; it is simply presenting a biased view for discussion.

Introduction

This document explores two patterns of API-X that have been identified: the filter pattern and the direct invocation pattern[1]. This document does not present or promote a soup-to-nuts architecture. Rather, it focuses on the specific patterns discussed below, and the interaction between the actors:

1. The client
2. API-X core
3. Service Discovery and Binding Framework
4. Extensions

It attempts to coalesce and rationalize the requirements, behavior, and responsibilities of the API-X core with respect to the actors, especially extensions. This document does not explore the implementation of extensions[2], but it does consider what the inputs and outputs of extensions might be[3], per the use cases.

Notably, it does *not* promote the idea that API-X core inspect interface definitions of extensions at runtime in order to communicate with bespoke service APIs (e.g. inspecting WSDL or

[1] A last-minute addition speculates on a third pattern, the Listener Pattern.
[2] Indeed, this has been touched on in previous proofs-of-concept.
[3] Addressed further in Use Case Table

SSWAP-like descriptions of a service). It favors a simple implementation, accepting the resulting complexity and limitations in extension or service discovery implementations. More complex features and behaviors can always be added to a simple and robust API-X core implementation at a later time.

This document does not make a distinction between an *extension* and a *service* because it isn't clear to the author what the distinction is[4].

Filter Pattern

The API-X core acts as a proxy for the ultimate recipient of the request. The filter pattern allows extensions the opportunity to operate on an incoming request or outbound response before it is forwarded to its ultimate destination (e.g. a Fedora repository or the client).

API-X Core Responsibilities

The API-X core is responsible for error handling, identifying and assembling the chain of extensions, and coordinating the passing of request and response objects between extensions, before forwarding the request or response to the intended recipient. Ideally, the API-X core is able to fulfill these responsibilities on a lightweight (minimal shared state), contractual (e.g. using interfaces and/or shared semantics) basis, without being aware of extension implementation details.

Extension Responsibilities

Extensions, as far as this document is concerned, are black boxes. They are responsible for performing their function as advertised (e.g. the de-duping extension does what it needs to do in order to de-duplicate incoming binary deposits, or the validation extension does what it needs to do in order to validate incoming RDF).

How extensions perform their function is beyond the scope of this document. However, this document suggests the following limitations on extensions (in the context of the filter pattern): they *are not* responsible for fulfilling the original request of the client, nor are they aware of any other extensions operating in the API-X runtime.

Remote Filter Pattern

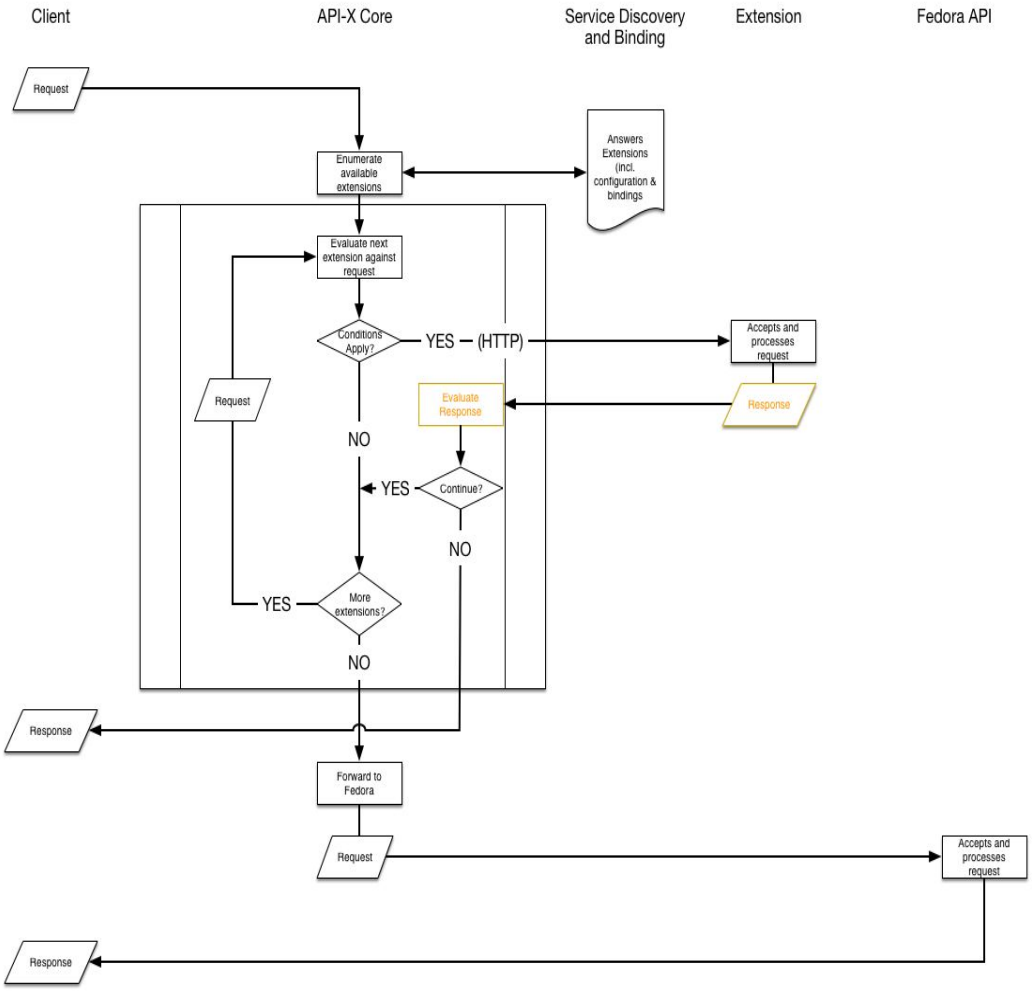
In the Remote Filter pattern, the extension is deployed remotely. For our purposes, "remote" means that some wire protocol (e.g. HTTP[5]) must be used by API-X core to communicate with the extension. This may be a requirement for multiple reasons.

---

[4] The distinction is probably a topic for another discussion and may be sussed out using proof-of-concepts.
[5] There are other protocols besides HTTP that could be used to communicate with remote Extensions, but at this point this document only considers HTTP.

1. The API-X runtime and the extension runtime may differ. For example, the API-X framework may run in Java while an individual extension may be implemented in PHP or Python.
2. Others?

Figure 1: Graphical representation of the Remote Filter Pattern (src)



The remote filter pattern is characterized by an HTTP interaction with an extension (i.e. API-X core invokes the extension using an HTTP request/response). Input to the extension is the client's HTTP request body, and the output from the extension is an HTTP response body. The extension may do whatever it deems necessary to fulfill its obligations, including invoking a remote service (not shown in the graphic).

Assuming that the extension receives the incoming request from the client 'as-is', limitations of this pattern are:

- The extension cannot modify the client request and make those changes visible to subsequent extensions in the chain. The request is supplied to the remote extension with "by value" semantics.[6]
- The remote extension cannot modify the response from the repository.
- The remote extension cannot short-circuit the evaluation of any remaining extensions in the chain except by responding with a 4xx or 5xx HTTP status code.

These are serious limitations, because it implies that any extension that wishes to mutate an incoming request or outbound response is not fit for the remote filter pattern. Extensions that would be fit are those that are able to perform their function without modifying the request or response, and who can communicate the success or failure of their operation via HTTP status codes:

- Validation (`POST, PUT, PATCH, DELETE`) - Extension accepts a HTTP request, performs any validation on the entities according to its configuration, and returns an HTTP code indicating success (2xx) or failure (4xx). Note that the optional validation use case cannot be satisfied by this pattern, because there is no general way to capture a warning from the HTTP response or to receive an entity body that has been transformed by the validation extension. A `100 Continue` response might be appropriate, however, there is the question of how to communicate the validation warning in the response from the extension to the client in a way that allows the API-X core to reason over the response in a generic way.
- Deduplication (`POST, PUT`) - Client supplies an HTTP request for creating or updating a binary file. Extension receives this request, computes a digest over its content, and checks the repository or an index to see if the binary resource already exists. If it already exists, a 409 is returned with an entity body containing the location of the existing resource. If it doesn't exist, a 204 might be returned. The client's request may contain a header that forces the creation or update of the binary resource, effectively bypassing this extension.
- Embargo handling, filter approach (`GET`) - Checks WebAC permissions on request, returns 4xx if the resource is embargoed. Optionally removes WebAC restrictions if embargo period is over.

Local Filter Pattern

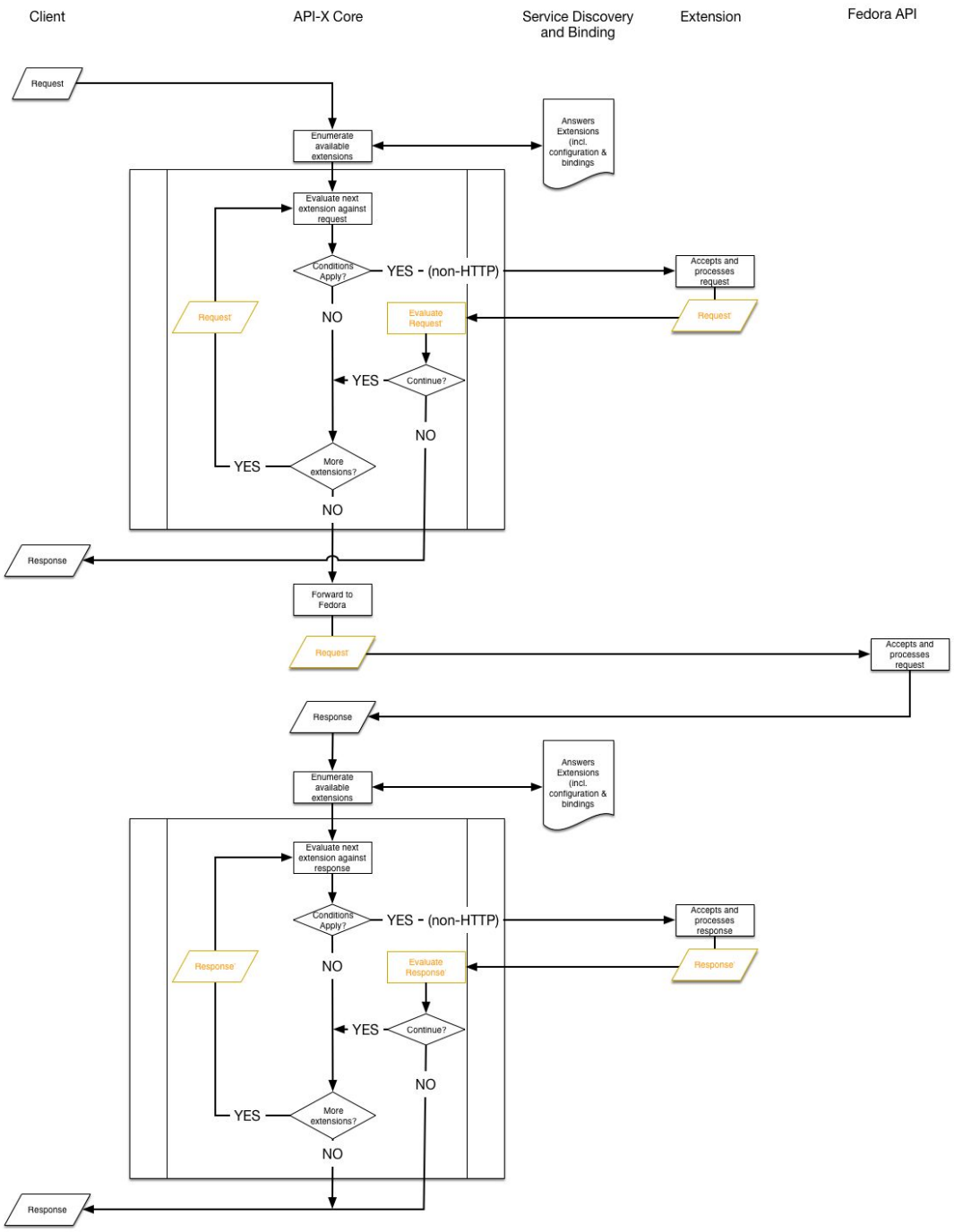
The local filter pattern is characterized by an in-VM[7] interaction with an extension. Unlike the remote filter pattern, local extensions have the opportunity to mutate the request *and* response, and do so without being constrained by an HTTP exchange. Input to the extension is an HTTP

[6] A remote extension cannot communicate intent to API-X. What if the extension wants to add a header in the response to the client? How does it indicate this in the extension's response to the API-X request? Similarly, what if it wants to remove a header in the response to the client? What if the extension wants to transform the entity? How does it communicate the transformed entity back to API-X, so that the transformed entity body is carried through the remaining extension invocations and ultimately sent back to the client?

[7] E.g. A Java VM, presuming the API-X core is implemented in Java.

request/response object (or an equivalent abstraction), and the output is the object modified by the extension. The extension may do whatever it deems necessary to fulfill its obligations, including invoking a remote service (not shown in the graphic). The extension must fold the response from any remote service invocations into the request object for the next extension.

Figure 2: Graphical representation of the Local Filter Pattern (src)



In this scenario, invoking the extension does not require the use of HTTP or any other wire protocol by API-X core. Extensions might be invoked using a runtime API (e.g. a Java interface), or implemented using a supported technology (e.g. a Camel route).

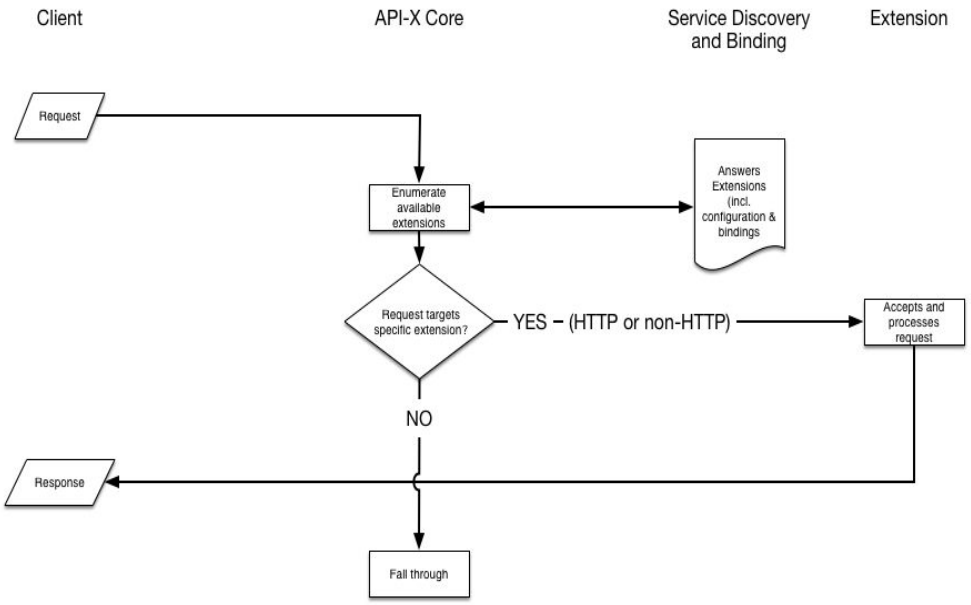
Use cases that fit this pattern (TODO REVIEW):

- MODS XML service - transforms resource metadata into mods xml or dc xml (based on `Prefer` header in the request?)
- Signposting service - modify the response headers using information from the response body.
    - TimeMap and TimeGate `Link` headers
    - Re-write Content-Disposition headers
- Optional validation - extension can modify the entity body as needed: correct a validation error, type the object as invalid, or issue a warning in some form.
- JSON-LD compaction service - transforms the response into compacted JSON-LD based on a `Prefer` header contained in the request.

Direct Invocation Pattern

Characterized by a client discovering[8] an extension, and subsequently interacting directly with that extension, possibly bypassing the API-X framework. If the API-X framework sits between the client and the extension, it shall act as a transparent proxy in this pattern. Requests directed to and responses originating from an extension are not modified by other extensions in the API-X framework, or the API-framework itself, even if the API-X core "sees" the request on the way to or from the direct service invocation endpoint.

Figure 3: Graphical representation of the Direct Invocation Pattern (src)



[8] Discovery of a service may be performed by examining `Link` headers of Fedora resources, direct interaction with the Service Discovery and Binding API, or some other out-of-band mechanism.

Extension Responsibilities Revisited

With the direct invocation pattern, extensions *are* responsible for fulfilling the client's request. The client's request targeted a specific extension, therefore the extension assumes the responsibility of fulfilling it.

- Preservation to external storage - expose HTTP API endpoint on a resource that allows the client to re-generate compound digital objects for preservation
- Binary derivative generation - expose HTTP API endpoint on binary resources that allows derivatives to be generated
    - relationship to particular content models triggering a derivative generation process
- JSON-LD compaction service - expose an HTTP API endpoint that emits a compacted JSON representation of RDF resources
- MODS XML service - expose an HTTP API endpoint on RDF resources that emits a MODS XML representation
- Template rendering service - expose HTTP API endpoint on RDF resources that emits an HTML representation of the resource (via Mustache in conjunction with JSON[9])
- Associated Fedora Object URIs with Web Services
- JAX-RS Extensions
- Package Ingest
    - Package Deposit
    - Deposit Workflow State
    - Recover from failed package deposit
- Provenance Stream
- Retrieve Domain Objects / Representation Derived from Graph
- Memento Timegate & Timemap implementation
- Embargo handling, async lifter approach - expose endpoint for triggering a process to update WebAC rules (i.e. lift embargo) based on resource embargo status
- ID Service - exposes service endpoint for managing and querying identifiers

Listener Pattern

A half-baked idea; registered extensions listen to repository event streams and react accordingly. In this pattern the actors are the extension, API-X core, and Fedora. The client doesn't play a role; extensions are reacting to the Fedora event stream. API-X core provides a runtime for registering extensions, and provides access to components that might be leveraged

[9] The prototype Mustache service has a dependency on the compacted JSON service.

by multiple extensions (a component that provides access to a cache of Fedora resources, for example).[10,11]

The listener pattern may provide another pathway for satisfying some use cases. For example, the binary derivative use case could be satisfied with an extension that listens to Fedora's event stream and stores the derivatives in the appropriate place. The same extension could implement the direct invocation pattern and accept HTTP requests to generate (or retrieve) derivatives on demand.

- Binary derivative generation - react to Fedora's event stream by generating derivative binary resources and storing as directed by the extension's configuration.
- ID Service - listen to Fedora event stream to populate a map external to internal identifiers.
- JSON-LD compact cache - react to fedora event stream to populate a cache of fedora resources in JSON-LD compacted form

Unclassified

TODO revisit/review

- Extra-repository access control - unclassified right now because this involves setting up API-X as a proxy in front of all of your infrastructure. It isn't clear this is the recommended use of API-X. This is a slightly more complex use case because it involves authentication and authorization. There are also existing use cases that have high impact and narrower scope that would be useful to address now.
- Caching portion of the JSON-LD compaction service
- Caching triples portion of the MODS XML service
- ESIP OpenSearch API - seem to be questions about how this would be implemented. So rather than picking an approach at this moment I'm punting.
- Distributing API Extensions - Encompasses some of the operation aspects of extension management, so it doesn't really contribute as an extension use case as such.
- Hot update of Extension service - Encompasses some of the operation aspects of extension management, so it doesn't really contribute as an extension use case as such.
- JAX-RS Extensions - Encompasses some of the operation aspects of extension management, so it doesn't really contribute as an extension use case as such.
- Associating Fedora Object URIs with Web Services - Encompasses some of the operation aspects of extension management, so it doesn't really contribute as an extension use case as such.

---

[10] Taking Adam's point, in what way does this extend the Fedora API? This comment is on many of the use cases and if it is a concern, do we remove use cases that are not considered an extension of Fedora's API, or do we acknowledge the "scope creep" of API-X, and rename it to something more meaningful.

[11] The need for these components are seen in the local, remote, and direct patterns as well.