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.

Introduction

This document discusses three patterns presented in the API-X use cases: the filter pattern, direct invocation pattern, and listener pattern. This document does not present or promote a architecture beyond the role of the API-X core, and 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[1], but it does consider what the inputs and outputs of extensions might be[2], 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 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[3].

API-X Core Responsibilities

The API-X core is responsible for error handling, assembling the chain of extensions for each request, coordinating the passing of request and response objects between extensions, and insuring the request or response reaches the intended recipient. Ideally, the API-X core is able to fulfill these responsibilities on a lightweight (minimal shared state), contractual (e.g. using

[1] Indeed, this has been touched on in previous proofs-of-concept.
[2] Addressed further in Use Case Table
[3] The distinction is probably a topic for another discussion and may be sussed out using proof-of-concepts.

interfaces and/or shared semantics) basis, without being aware of extension implementation details.

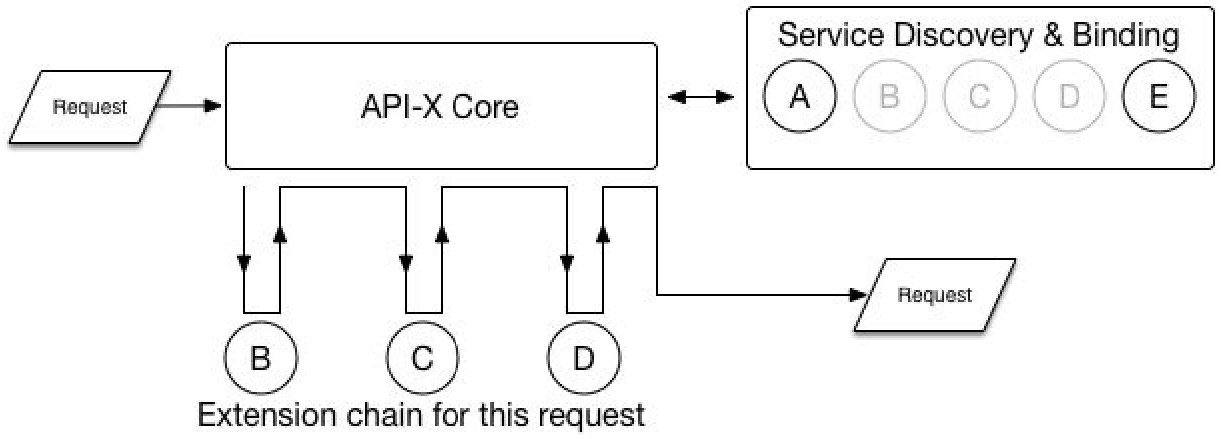
Filter Pattern

In the context of the filter pattern, the API-X core acts as a proxy for the ultimate recipient of the request. Execution of the filter pattern by API-X core allows multiple 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). The API-X core may even *intercept* the request, and fulfill it (i.e. generate and return a response) without the request reaching its intended recipient[4].

Extension Chains

Multiple extensions may have the opportunity to act on each request received by the API-X core.   For each request, API-X core leverages the Service Discovery and Binding (SDB) service to enumerate the available extensions, and then uses information from the SDB to determine which extensions can operate on an incoming or outgoing request. The API-X core assembles a so-called chain of extensions to operate on the request. After the last extension has operated on the request, API-X core forwards the request on to the intended destination.

Figure 1: Graphical representation of HTTP message flow through an extension chain (src)



Note that extensions do not self-identify as being members of a chain; the API-X core is responsible for determining whether or not an extension is a member of the chain (and is free to use other services, like SDB, to help make that decision). The only time an extension will see the request is when the extension is invoked.

[4] Consider an extension that fulfills requests for resources from a cache

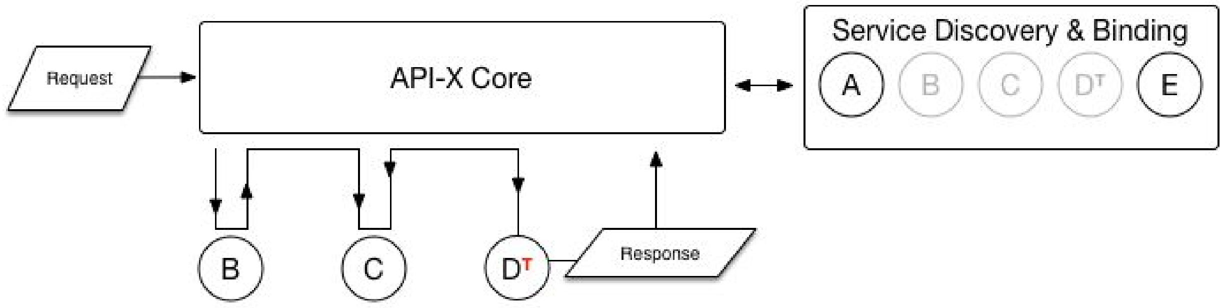
Terminal Extensions

The use cases highlight three different ways an extension may interact with a request (or response):

- Read-only: the extension reads the request, but does not modify the request in any way; e.g. a validation extension that simply provides a thumbs-up or thumbs-down on the content of a request
- Read-write: the extension modifies the request in some way; e.g. a signposting extension that adds a header to the request
- Fulfilling the request: the extension satisfies the request; e.g. an extension that delivers a response from a cache

The last point is interesting because the extension is assuming the responsibility of fulfilling the request by producing a response. Extensions that assume the responsibility for fulfilling a client request are so-called *terminal extensions*. A terminal extension assumes the responsibility of fulfilling the request from API-X core.

Figure 2: Graphical representation of HTTP message flow through a chain with a terminal extension (src)



An extension chain may have at most one terminal extension. If so, the terminal extension must be the last extension in the chain. If a chain does not have a terminal extension, the API-X core assumes the responsibility of fulfilling the request.

Extension Responsibilities

As far as this document is concerned, extensions 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:

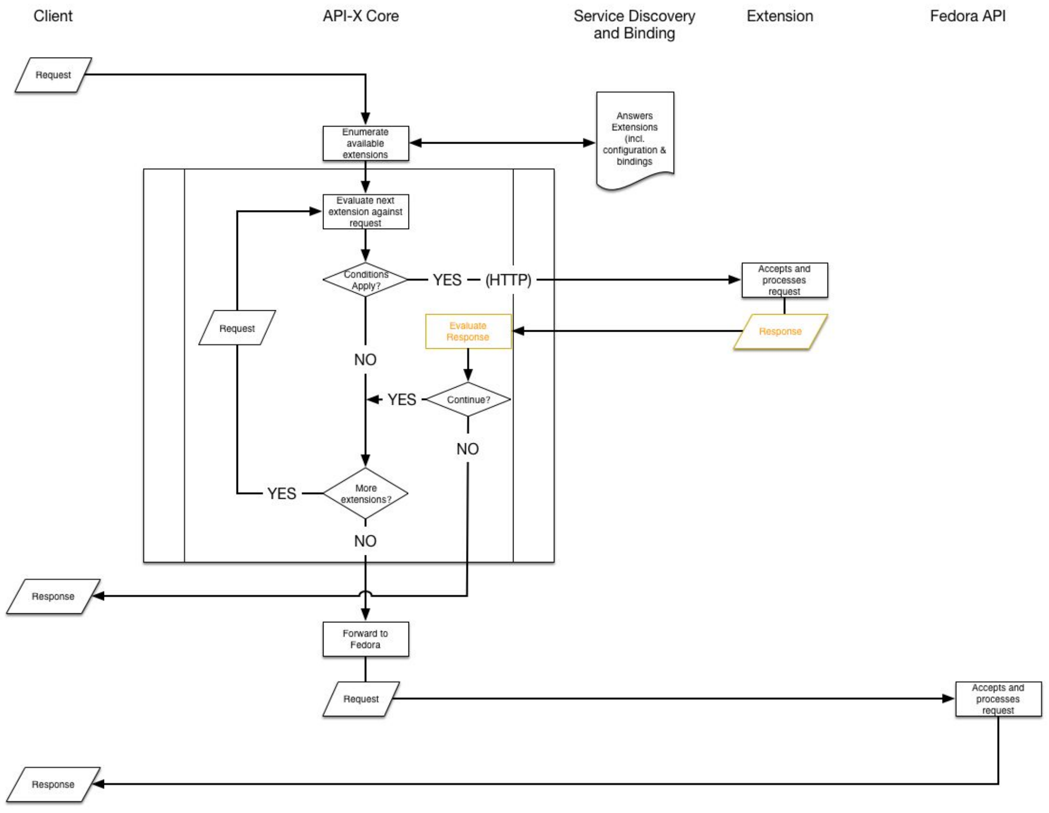
- Unless they are terminal, extensions *are not responsible* for fulfilling the original request of the client.
- Terminal extensions are responsible for producing a response.
- Extensions are not aware of any other extensions operating in the API-X runtime.

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.

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 3: Graphical representation of the Remote Filter Pattern (src)


Client
API-X Core
Service Discovery and Binding
Extension
Fedora API
Request
Enumerate available extensions
Answers Extensions (incl. configuration & bindings)
Evaluate next extension against request
Conditions Apply?
YES — (HTTP)
Accepts and processes request
Request
Evaluate Response
Response
NO
YES
Continue?
NO
YES
More extensions?
NO
Response
Forward to Fedora
Request
Accepts and processes request
Response

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. The extension may do whatever it deems necessary to fulfill its obligations, including invoking a remote service (not shown in the graphic) before returning a response.

[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.

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

- The remote extension cannot modify the client request or server response 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 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.
- 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.

It should be noted that remote extensions *are* fit to serve as terminal extensions in an extension chain. A response from a remote, terminal, extension would be forwarded by API-X core to the client. Use cases that may be implemented as terminal extensions are similar to the use cases that fit the direct invocation pattern (discussed below).

Local Filter Pattern

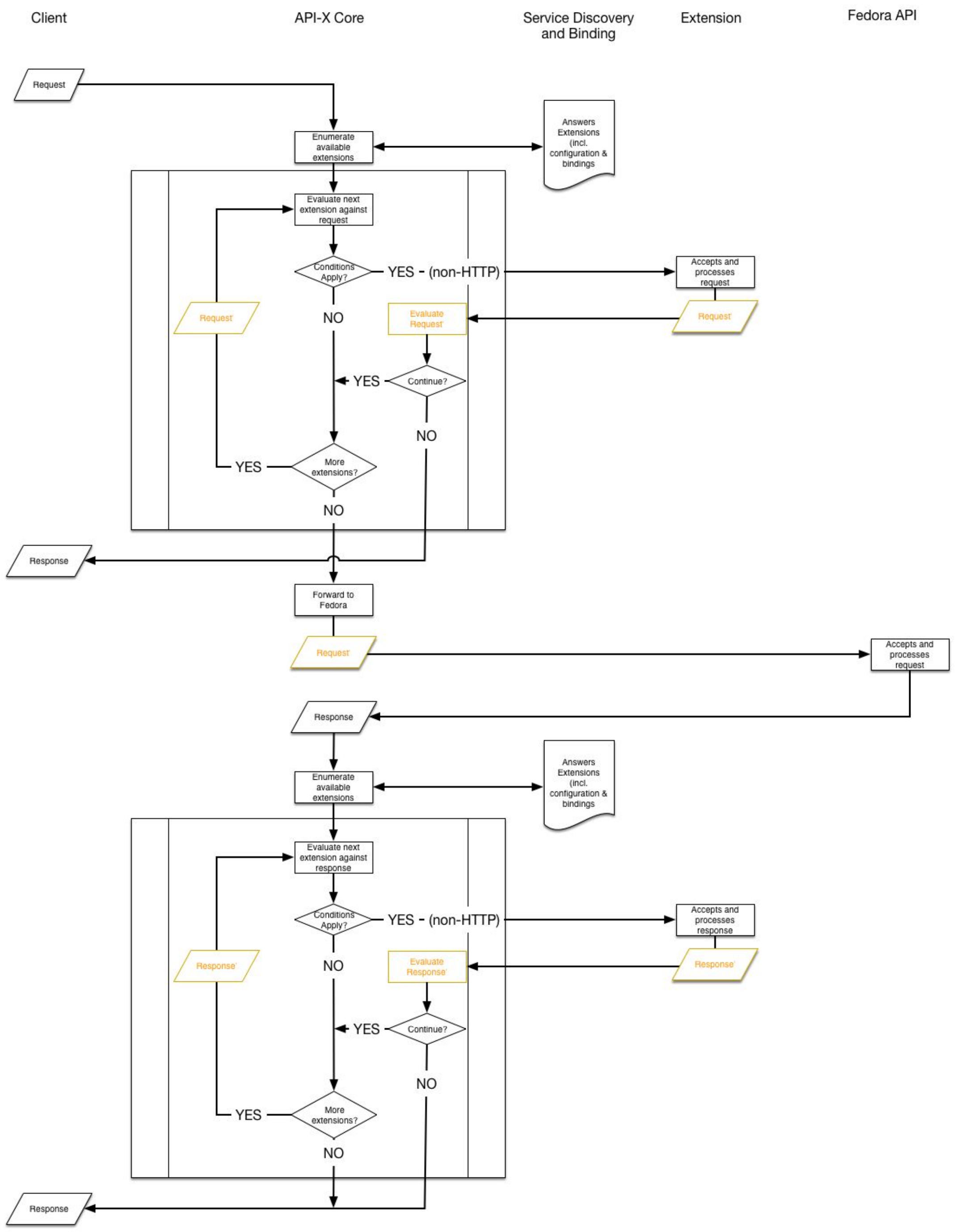
The local filter pattern is characterized by an in-VM[7] interaction with an extension. Unlike the remote filter pattern, local extensions have the opportunity to mutate the request *and* response,

[6] Assume a remote extension is invoked with the original client request. How would the remote extension indicate to API-X core that it intends to add or remove a header from the client request?

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

and do so without being constrained by an HTTP exchange. Input to the extension is an HTTP 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 4: Graphical representation of the Local Filter Pattern ([src]())


Client
API-X Core
Service Discovery and Binding
Extension
Fedora API
Request
Enumerate available extensions
Answers Extensions (incl. configuration & bindings
Evaluate next extension against request
Conditions Apply?
YES - (non-HTTP)
Accepts and processes request
Request
Evaluate Request
NO
Continue?
YES
NO
More extensions?
YES
NO
Response
Forward to Fedora
Request
Accepts and processes request
Response
Enumerate available extensions
Answers Extensions (incl. configuration & bindings
Evaluate next extension against response
Conditions Apply?
YES - (non-HTTP)
Accepts and processes response
Response
Evaluate Response
NO
Continue?
YES
NO
More extensions?
YES
NO
Response

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:

- 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.

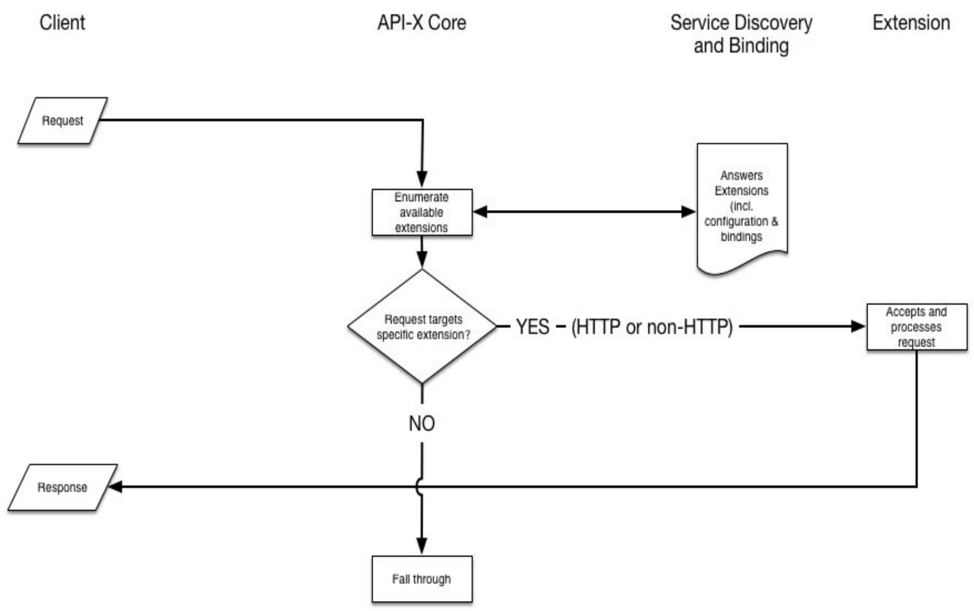
It should be noted that local extensions are also fit to serve a terminal extensions in an extension chain.

Direct Invocation Pattern

The direct invocation pattern is 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 5: Graphical representation of the Direct Invocation Pattern (src)

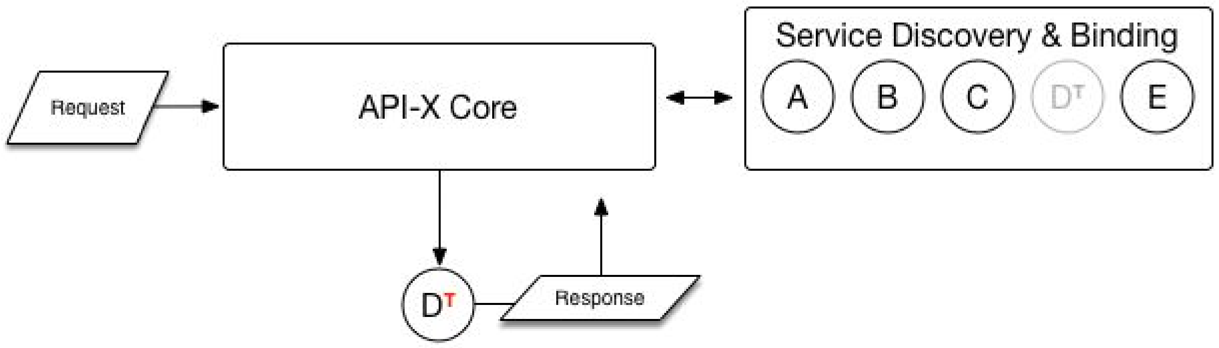
[8] Discovery of a service may be performed by examining `Link` headers of Fedora resources, direct interaction with the Service Discovery and Binding API, or some other out-of-band mechanism.


Client
API-X Core
Service Discovery and Binding
Extension
Request
Enumerate available extensions
Answers Extensions (incl. configuration & bindings
Request targets specific extension?
YES – (HTTP or non-HTTP)
Accepts and processes request
NO
Response
Fall through

Extension Responsibilities Revisited

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

Figure 6: Graphical representation of HTTP message flow through a terminal extension (src)



The direct invocation pattern may be considered a degenerate case of the filter pattern: the client effectively names a terminal extension to handle the request (that is, the extension chain consists of exactly one extension, and the extension must be terminal) using properties of the request (e.g. a request URI identifying a particular extension: `/resource/`**`ext:service`**).

- 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

In the listener pattern the actors are the extension, API-X core, and Fedora event stream. The client doesn't play a role; extensions are reacting to an event stream. API-X core provides a runtime for registering extensions, and provides access to shared components that might be leveraged 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. The JSON-LD compaction use case could be satisfied with an extension that listens to Fedora's even stream, and stores compacted JSON-LD representations

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

[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.

in a cache. The same extension could implement the the direct invocation pattern (or be a terminal extension in a filter pattern) and deliver compacted JSON from the cache.

- 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

Conclusion

Three patterns were derived from the API-X use cases: the filter pattern, the direct invocation pattern, and the listener pattern. The direct invocation pattern can be viewed as a degenerate case of the filter pattern. The listener pattern provides an alternative pattern for integrating an extension with the repository, and suggests that API-X may host components shared by multiple extensions. The filter pattern presents two sub-patterns: the local and remote patterns.

The API-X core is responsible for assembling the chain of extensions for each request, and directing the HTTP messages between extensions.

Extensions interact with requests and responses in three modes: read-only, read-write, and response-generation. An extension may be designated as terminal, where it assumes the responsibility of fulfilling the client's request by generating a response. Remote extensions are limited to read-only and response-generation modes due to the HTTP interaction model.

These patterns cover the majority of the use cases present on the wiki to date. Use cases that are not covered by these patterns are discussed briefly below.

Other use cases

For completeness, these are use cases that were not represented in this paper, with brief notes as to why.

- 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 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 operational aspects of extension management, so it doesn't really contribute as a pattern.

- Hot update of Extension service - Encompasses some of the operational aspects of extension management, so it doesn’t really contribute as a pattern.
- JAX-RS Extensions - Encompasses some of the operational aspects of extension management, so it doesn’t really contribute as a pattern.
- Associating Fedora Object URIs with Web Services - Encompasses some of the operational aspects of extension management, so it doesn’t really contribute as a pattern.