Solid server – Proposed architecture v1.2.0 *(status: draft)*

Ruben Verborgh – August 13, 2019

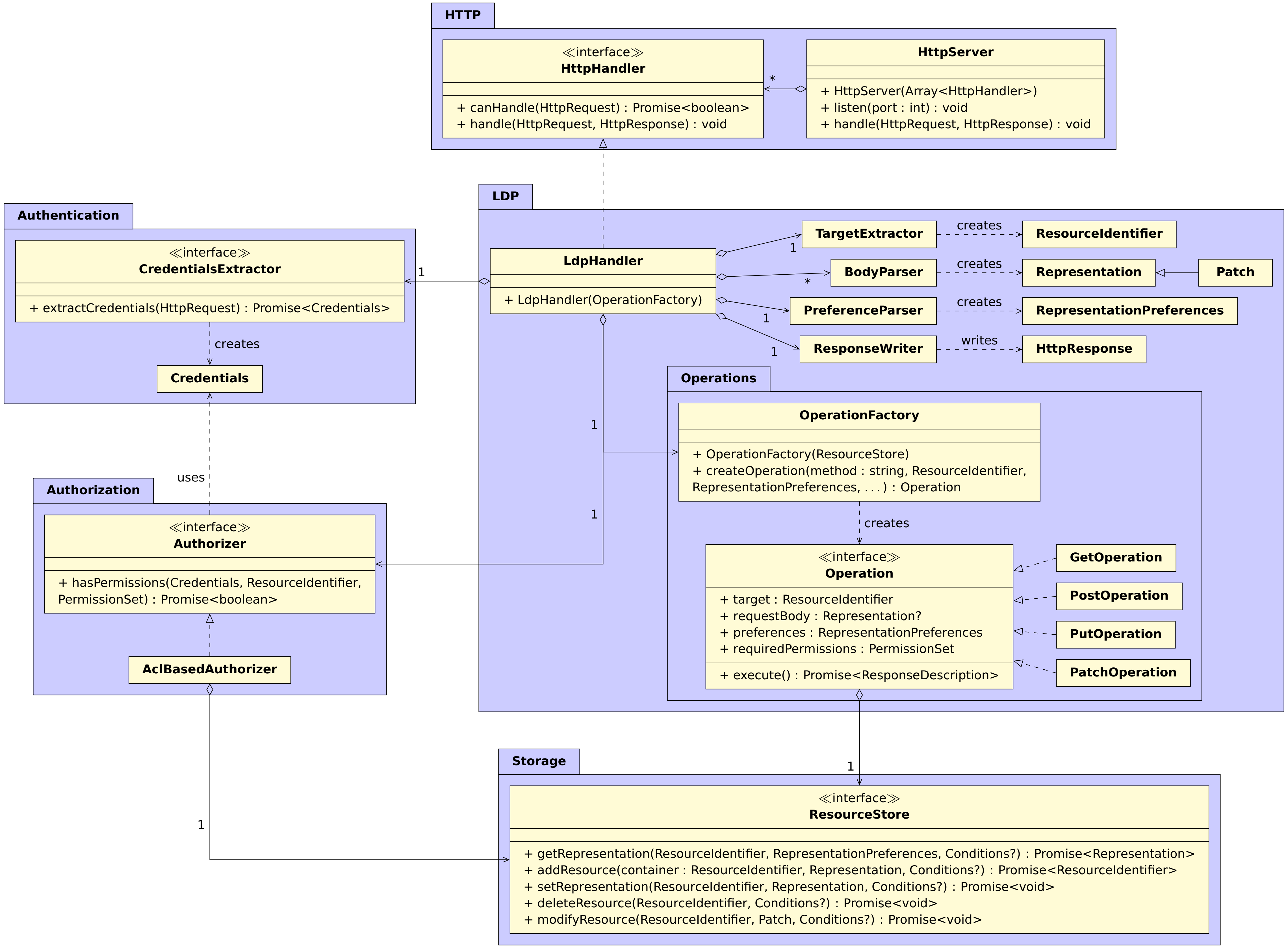
Purpose

This document conveys a personal view on important architectural considerations for a Solid server.
It is intended as a tool for discussion, to raise questions, and to highlight concerns.
It does not have any official standing whatsoever.

Legend

The architectural diagram follows standard UML notation.
For more specific symbols that are not part of UML, Node.js/JavaScript/TypeScript conventions were used as follows:
T? represents a value that is either not present or a value of type T.
Promise<T> represents a value that will asynchronously resolve to a value of type T.
Readable<T> represents an asynchronous one-time readable stream of values of type T.
Buffer is an in-memory buffer of bytes, possibly with a character encoding.

Overview of LDP and Access Control

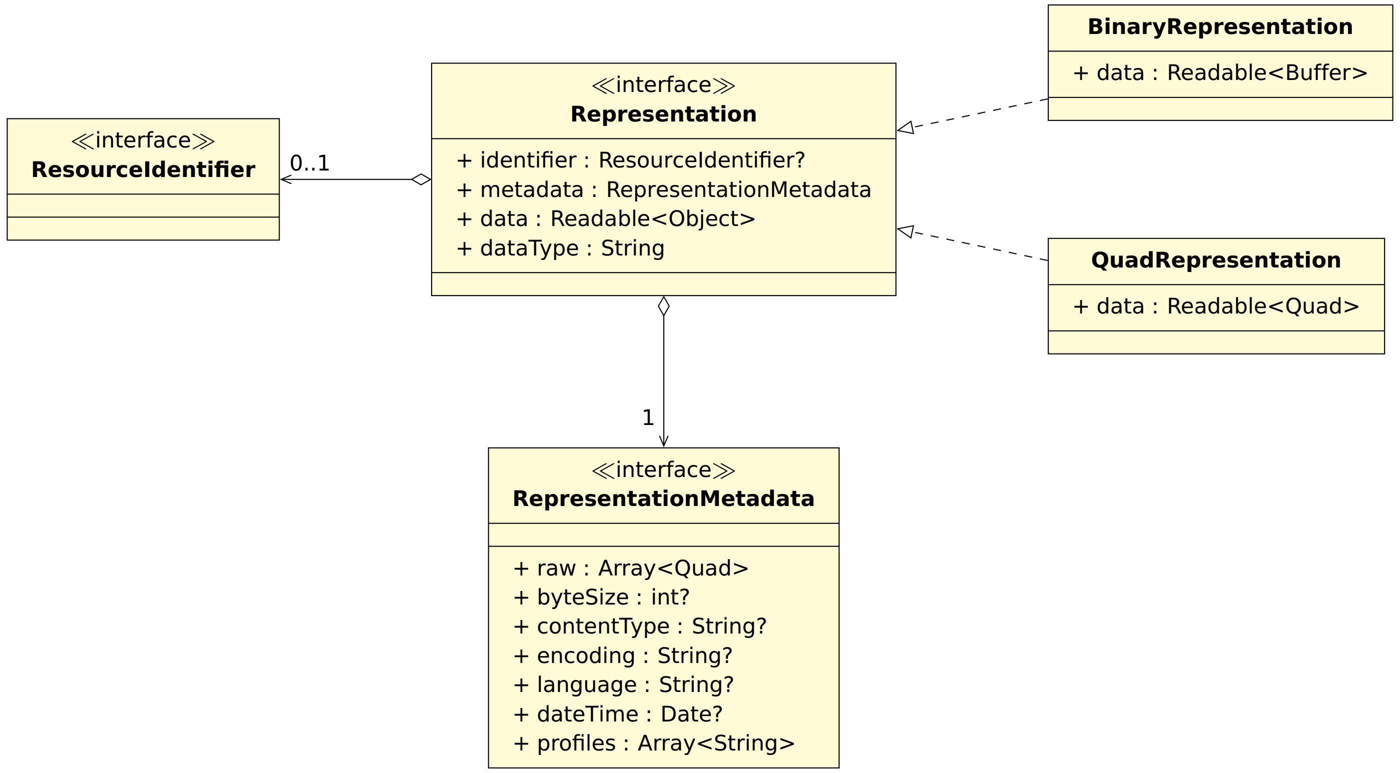


Resources and Representations

The intention of **ResourceIdentifier** and **Representation** is to capture the REST notion of a resource and its representation. In the case of a photograph, the resource is the photograph itself, whereas a representation is a concrete manifestation of that photograph with a certain resolution and file type. In the case of an RDF document, the resource is the RDF graph, and concrete representations serialize that graph into Turtle or specific framings of JSON-LD.

For all practical purposes, **ResourceIdentifier** can just be a **URL**; the terminology is mainly used to emphasize the resource/representation notion of REST. Also, there is no **Resource** class, because resources are always manipulated through representations in REST, so we only need to *identify* resources, and only deal with them through their representations.

Crucially, as the diagram below shows, the **Representation** interface can have vastly different underlying in-memory structures, such as strings, binary streams, RDF streams, etc. So they can be photographs as well as RDF streams, and most other classes handling them do not need to care. This enables back-ends to be RDF-aware when they need to, and RDF-oblivious when they do not.

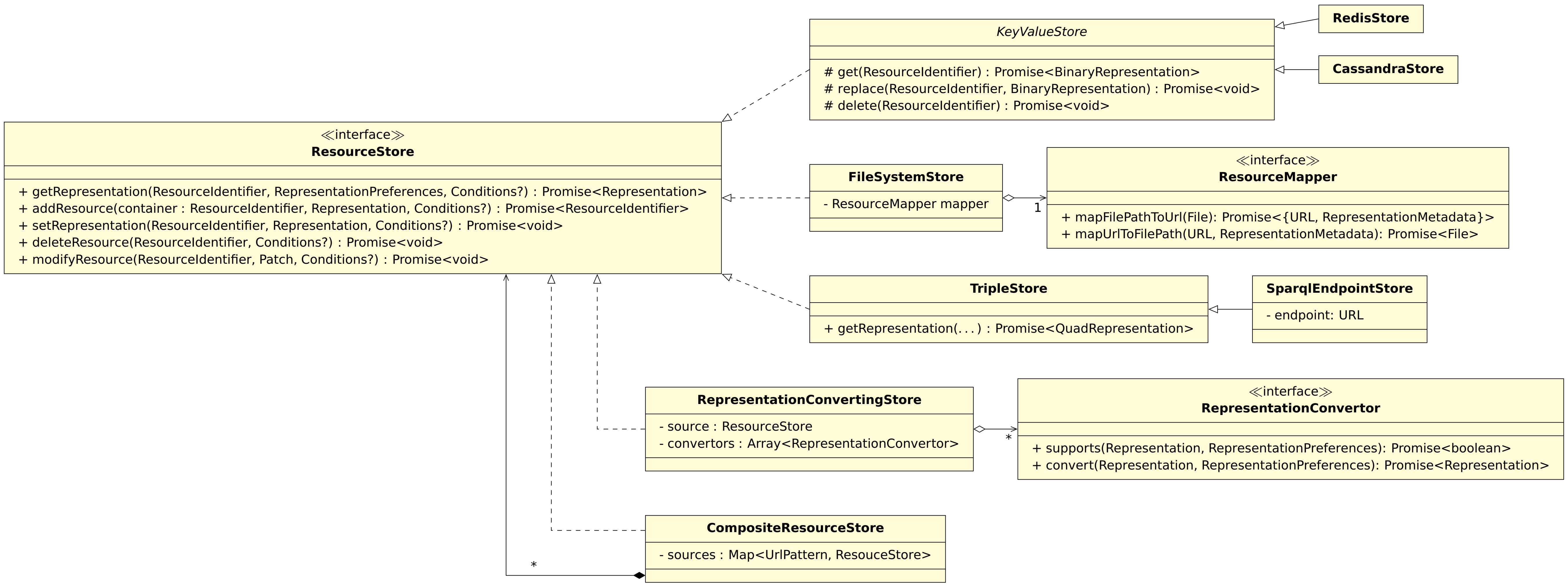


The **dataType** field returns the name of the class that elements of the **data** readable stream will have, for instance, `Buffer` or `Quad`.

Based on the **dataType** and **metadata** fields, other components can decide whether or not the representation is acceptable to the user agent, and, if this is not the case, convert to a format that is. For instance, a `text/turtle` stream is acceptable for a user agent that requested `text/*`, whereas a `Readable<Quad>` will still require serialization.

The **RepresentationMetadata** interface essentially exposes a set of RDF triples that describe properties about the representation. For convenience, direct getters to common properties can be added, non-binding examples of which are shown in the diagram.

ResourceStore implementations

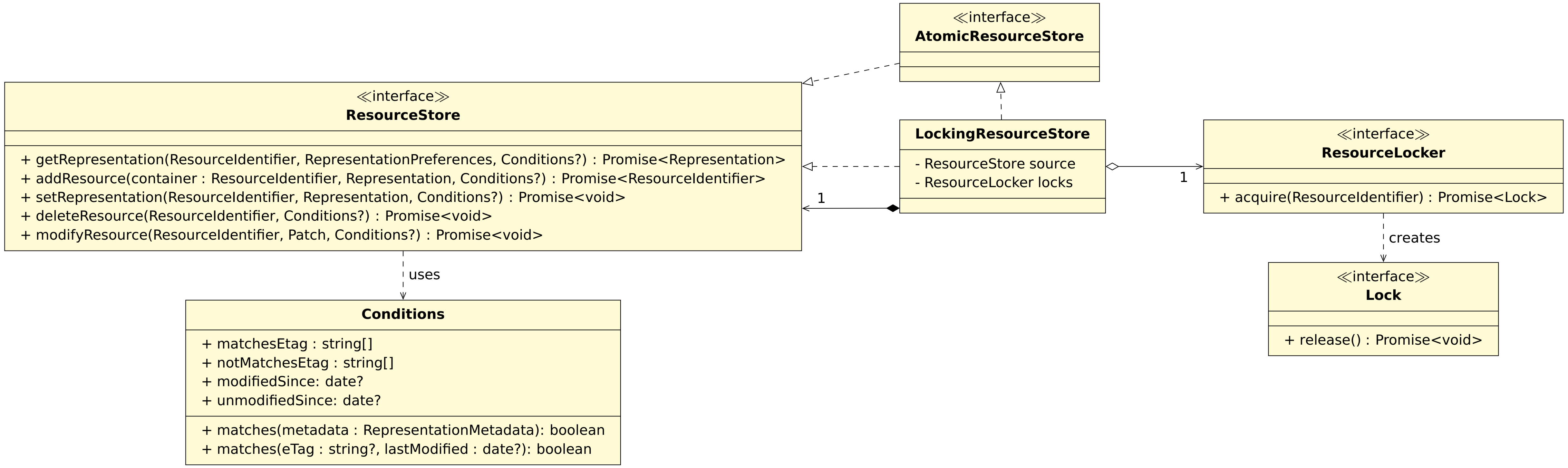


A **ResourceStore** will *try* to satisfy any **RepresentationPreferences** passed to it, but only if this is reasonably easy for the store in question. For instance, a SPARQL endpoint can typically generate N-Triples as easily as Turtle, so it makes sense to directly generate N-Triples if the client prefers this. On the other hand, a file system will typically only have one representation on disk, so it is fine to always serve that single representation, regardless of client preferences.

Optionally, a **RepresentationConvertingStore** can be used to satisfy client preferences more accurately. It has access to **RepresentationConvertor** instances, which could (for instance) convert a stream of quads into Turtle or a specific JSON-LD frame. It can decorate any existing **ResourceStore** to extend it with more kinds of representations such as different content types.

A **CompositeResourceStore** can be used to have multiple back-ends on one pod, each answering to different URL patterns. This mechanism *could* be used also to serve large files like images, or static assets such as apps or scripts.

ResourceStore, atomicity, and conditional requests



The **ResourceStore** interface has been designed such that each of its methods *can* be implemented in an atomic way: for each CRUD operation, only one dedicated method needs to be called. A fifth method enables the optimization of partial updates with PATCH. It is up to the implementer of the interface to (not) make an implementation atomic. For some implementations, such as triple stores or other database back-ends, atomicity is a given. We *could* explicitly indicate atomicity by having such implementations implement the (otherwise empty) **AtomicResourceStore** interface as a tag.

The **Conditions** class represents the conditions of an HTTP conditional request. It is passed to all write methods (and possibly also read) of **ResourceStore**. The store is responsible for validating conditions at the right moment and, should validation fail, for aborting the modification by throwing an error.

Some back-ends are not atomic by themselves, such as a file system, where a read+append sequence could unknowingly be interrupted by a write that thereby breaks atomicity. Instead of having to implement a dedicated locking mechanism for every non-atomic back-end, these stores can be made atomic by decorating them with a **LockingResourceStore**. This class wraps another **ResourceStore** and adds a locking mechanism, of which different implementations can exist.

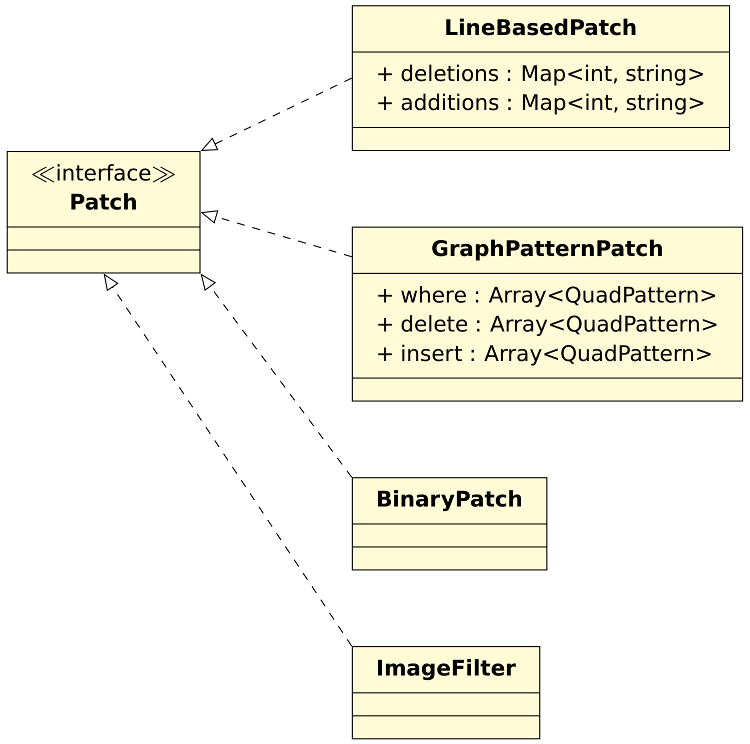
If the store knows how to validate conditions, it can use the raw exposed fields on **Conditions**. If it does not, it can call `matches` with both ETag and the last modified date, or try one of them before the other. Finally, if it knows about neither ETag nor last modified date, it can simply pass the metadata as a whole.

It is important to emphasize that atomicity is *not* the only reason for the design of the **ResourceStore** interface. Another consideration is `modifyResource`, which allows us to optimize modifications in a backend-specific way. Since we expect small modifications to larger resources to be a common for Solid apps, we need to be able to handle those efficiently. `modifyResource` gives implementations the freedom on how to apply patches, such that they can pick whichever option is most efficient for a given patch and, if desired, support atomicity.

The conditions argument is optional, and only passed for conditional requests. If a store decides not to support conditional requests, it must throw an error if conditions are passed.

Patch

A **Patch** contains a description of changes to be made to a certain (representation of a) resource. The **Patch** object itself does not know how to *apply* this patch; it is merely a data object.



A **ResourceStore** *might* have knowledge on how to apply certain types of patches itself. For instance, file-based stores might have built-in support for **LineBasedPatch**, and SPARQL endpoints or in-memory RDF stores likely have built-in support for **GraphPatternPatch**.

There is case to be made for a *Patcher* interface for objects that can apply all patches of a certain type to certain representations. For instance, a **GraphPatternPatch** could be applied to RDF graphs serialized as documents, by a *GraphPatternPatcher* that operates independently of any specific store.