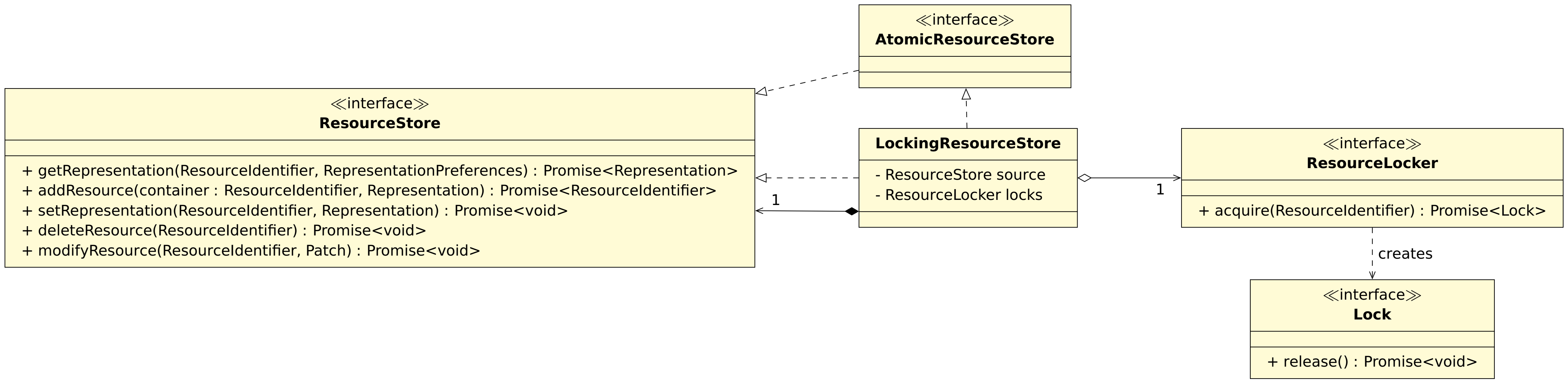
Solid server – Store atomicity *(status: obsolete)*

Ruben Verborgh – August 13, 2019

ResourceStore and atomic operations



The **ResourceStore** interface has been designed such that each of its methods can be implemented in an *atomic* way: for each CRUD operation,[1] only one dedicated method needs to be called. 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.

Some implementations are *not* atomic by default, such as a file system, where a read+append sequence could unknowingly be interrupted by a write that thereby breaks atomicity. Such non-atomic stores could be made atomic by decorating them with a **LockingResourceStore**. This class wraps another **ResourceStore** with a locking mechanism, which can be implemented in different ways. An example method implementation is listed on the right.

```
async function modifyResource(id, patch) {
  const lock = await this._locks.acquire(id);
  try { return await this._source.modifyResource(id, patch); }
  finally { await lock.release(); }
}
```

Design considerations

It is important to emphasize that atomicity is *not* the only reason for the design of the **ResourceStore** interface. The other consideration is in the 5th method `modifyResource`, which allows us to optimize modifications in a backend-specific way. Since we expect small modifications to larger resources to be a common pattern for Solid apps, we need to be able to handle those efficiently.

A simpler implementation with 4 methods could support `PATCH` as follows:

1. call `getRepresentation`
2. apply the patch
3. call `setRepresentation`

However, in addition to violating atomicity (or requiring another locking mechanism), it would also give suboptimal results when the resource is large and the patch is just a single triple. Moreover, it would be unnecessarily complex and slow for the case of triple stores, which support patches natively.

In contrast, `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.

ResourceStore and conditional requests

With the above, we have established that **ResourceStore**:

- supports all CRUD requests;
- can support all types of patches efficiently;
- support atomicity (regardless of native support by the back-end).

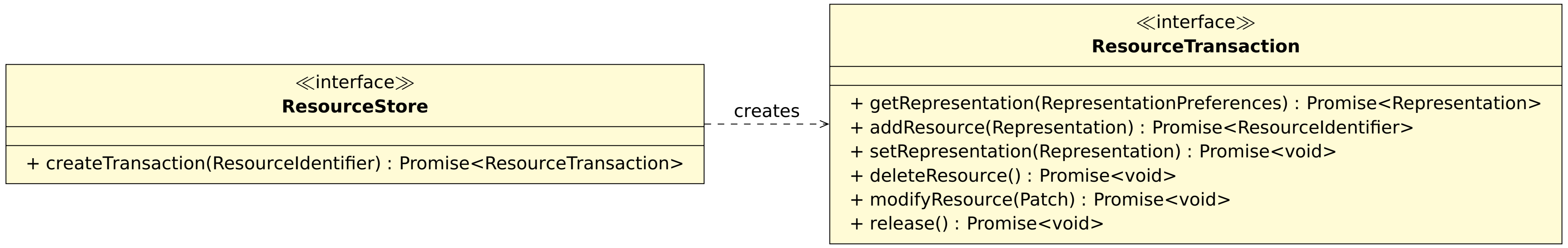
However, the proposed mechanism does *not* support conditional HTTP requests (RFC 7232), which must be aborted if the resource prior to modification does not satisfy certain conditions. These are not supported because:

- **ResourceStore** cannot abort, because it does not know the conditions.
- Callers of **ResourceStore** know the conditions, but they cannot check them in an atomic way, since they would not be able to prevent modifications in between the `getRepresentation` call for checking the conditions, and the subsequent modification call.

Hence, we explore three different extensions to the architecture that aim to support conditional requests, and analyze their properties.

[1] There are 5 operations rather than 4 because we distinguish between full representations update for `PUT` and partial updates for `PATCH`.

Approach 1 to conditional requests: *transactions*



Description

The original **ResourceStore** methods are moved into a **ResourceTransaction** interface, which gets an additional `release` method to end the transaction. The caller becomes responsible for steering the atomicity (but the implementation remains with the **ResourceStore**).

Implementations do not need to be (and likely would not be) *actual* transactions, in the sense that operations do not *need* to be buffered until the very end when `release` is called. They rather can function as *locks/semaphores* that guarantee no other operations can happen in the meantime.

Analysis

Additional knowledge required by existing components:

- The caller knows how to validate request conditions.
- Every **ResourceStore** implementation must be transaction-aware (or at least *lock*-aware), which is not the case for back-ends such as files.
- Callers of **ResourceStore** must be transaction-aware.

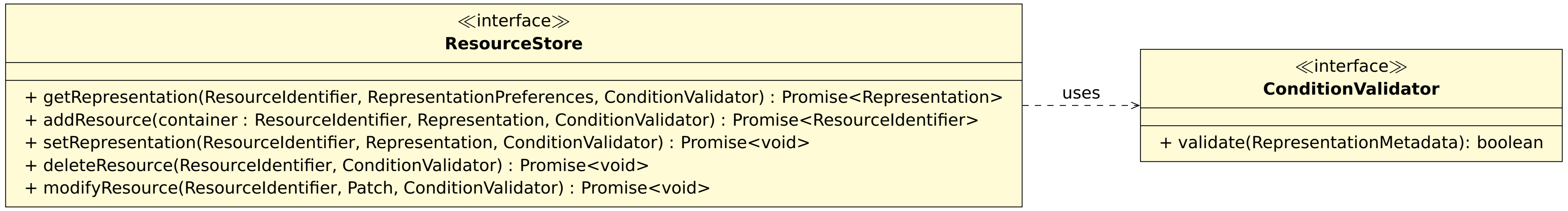
When a conditional request arrives, implementers must:

1. call `createTransaction`
2. call `getRepresentation`
3. check the conditions
4. if the conditions are satisfied, call the modification method
5. call `release`

This comes with a couple of caveats:

- We probably do not want to retrieve the full representation, but only the metadata (lazy loading can do that).
- It might result in the representation (or its metadata) being loaded twice: once by the caller when getting the representation, and once by the store internally when performing the modification. (The transaction can, however, cache this.)
- It assumes that `getRepresentation` succeeds, which might not be the case for append-only stores.
- It assumes that `createTransaction` is sufficiently cheap.
- It assumes that keeping a lock/transaction open is sufficiently cheap.
- In general, it assumes that the caller has the best knowledge for checking the conditions in the cheapest way possible, which is not necessarily true. For instance, for one store it might be expensive to calculate ETag but not the last modified date, whereas it might be the opposite for another. A certain store might even be able to determine that a condition is met *without* retrieving a representation or its metadata (for instance, if its global last-modified date is not later than the requested one).

Approach 2 to conditional requests: *passing a validator*



Description

A **ConditionValidator** is passed to all write methods (and possibly also read) of **ResourceStore**. The store is responsible for calling `validate` at the right moment, and for aborting the modification if validation fails. The caller is responsible for writing the validation code.

(The validator argument *could* be optional; if a store decides to not support conditional requests, it must throw an error if a validator is passed.)

Analysis

Additional knowledge required by existing components:

- The caller knows how to validate request conditions, given metadata.

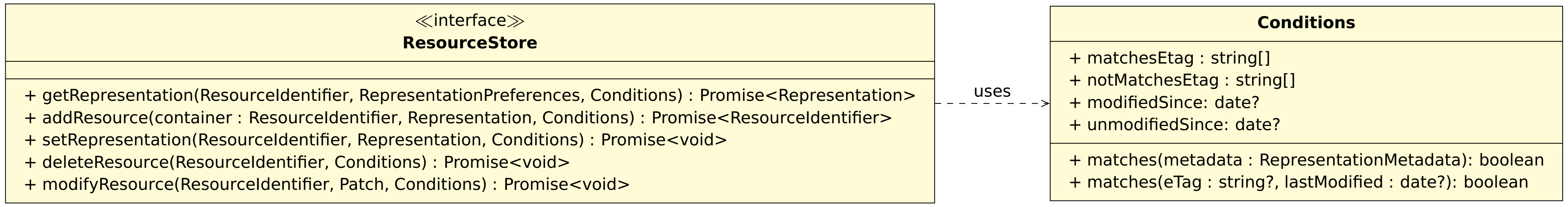
When a conditional request arrives, implementers must:

1. call the modification method, passing in the validation code
2. have **ResourceStore** call the validator at the right time

This comes with a couple of caveats (details in previous section):

- It assumes that metadata is available.
- It assumes that retrieving metadata is sufficiently cheap.
- It assumes that the caller has the best knowledge for checking conditions.
- For every store implementation, it must be tested whether every method checks the conditions.

Approach 3 to conditional requests: *passing the conditions*



Description

The **Conditions** themselves are passed to all write methods (and possibly also read) of **ResourceStore**. The store is responsible for validating conditions at the right moment, and for aborting the modification if validation fails.

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.

(The conditions argument *could* be optional; if a store decides to not support conditional requests, it must throw an error if conditions are passed.)

Analysis

Additional knowledge required by existing components:
(none)

When a conditional request arrives, implementers must:
1. call the modification method, passing in the validation code
2. have **ResourceStore** check the conditions at the right time

This comes with a couple of caveats:
- It assumes that the conditions do not change often.
- For every store implementation, it must be tested whether every method checks the conditions.