Atom Caching

In this lesson, we'll study Atom caching.

WE'LL COVER THE FOLLOWING

Tools

At https://validator.w3.org/feed/#validate_by_input, a validator is provided that can check whether an Atom feed is valid, checking to see if all necessary elements are present.

There are systems such as Atom Hopper that offer a server with the Atom format. In this way, an application does not have to generate Atom data, but can post new data to the server. The server then converts the information into Atom. Clients can fetch the Atom data from the server.

Efficient polling of Atom feeds

> Asynchronous communication with Atom requires that the client regularly requests the data from the server and processes new data. This is called **polling**.

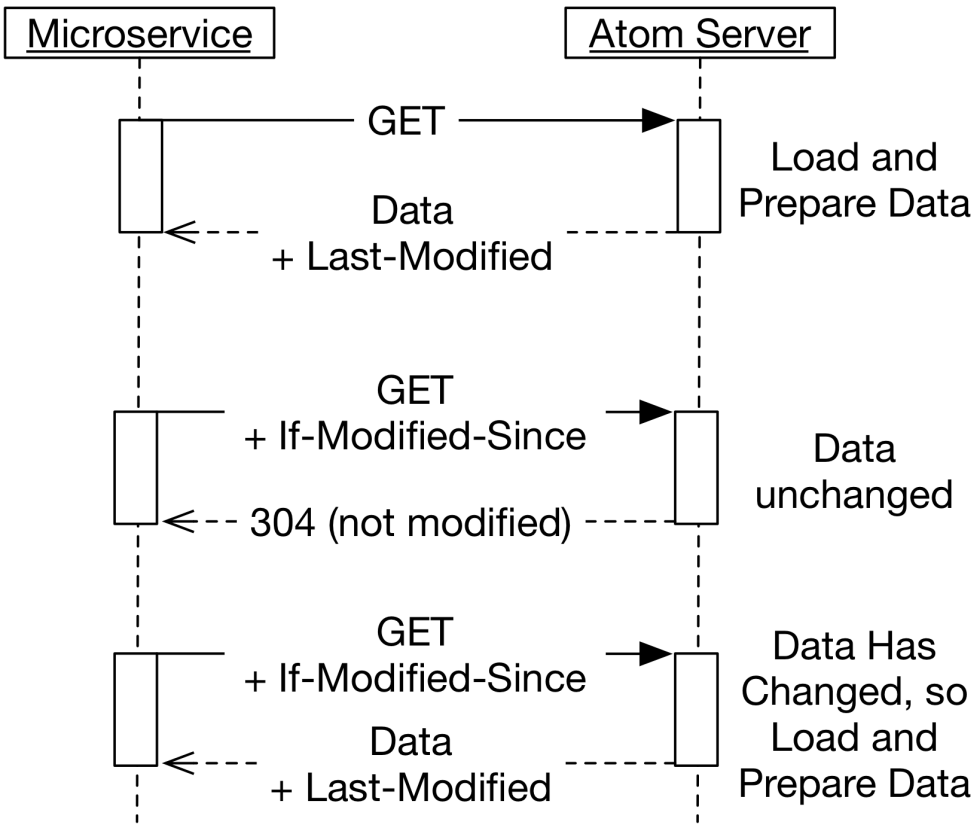
The client can periodically read out the feed and check the `updated` field in the feed to see if the data has changed. If this is the case, the client can use the `updated` elements of the entries to find out which entries are new and process them.

This is very time consuming because the entire feed has to be generated and transmitted. Additionally, only a few entries are read from the feed, most requests will show that there are actually no new entries. For this purpose, it does not make sense to create and transfer the complete feed.

HTTP caching

A very simple way to solve this problem is **HTTP caching**, see the drawing below.

HTTP provides a header with the name `Last-Modified` in the HTTP response. This header indicates when the data was last changed. This header takes over the function of the `updated` field from the feed.



Data unchanged

- The client stores the value of this header. On the next HTTP GET request, the client sends the read value in the `If-Modified-Since` header with the GET request.
- The server can now respond with an HTTP status of `304 (not modified)` if the data has not changed. Then, no data is transferred except for the status code.

Whether data has changed can often be determined very easily. For example, you can implement code that determines the time of the last change in the database. This is much **more efficient than converting all data into an Atom representation**.

Data has changed

- If the data has actually changed, the server responds normally with a status of `200 (OK)`.
- Also, a new value is sent in the `Last-Modified` header so that the client can use HTTP caching again.

The demo implements such an approach:

1. For a request with a set `If-Modified-Since` header, the database is used to determine the time of the last change.
2. This is compared with the value from the header.
3. An HTTP status `304` is returned if no data has changed.
4. In this case, only one database query is required to respond to the HTTP request.

Static Atom feed

An alternative would be to **create the Atom feed once** and store it on a web server as a **static resource**. In this case, dynamic generation is performed once. This approach also works efficiently for very large feeds.

In that case, it would be up to the web server to implement the HTTP caching for the static resource.

ETags

Another approach would be HTTP caching with ETags.

- Here, the **server returns an ETag** together with the data.
- The **ETag can be compared to a version number or checksum**.
- For any further requests, the **client sends the ETag along**.
- The **server uses the ETag to determine whether the data has changed** and **provides data only if new data is available**.

In the example, however, it is much easier to find out whether new orders have been accepted since a certain point in time. It therefore does not make sense to use ETags in the example.

Pagination and filtering

If a client is not interested in all events, it can signal this by setting parameters in the URL.

- This makes it possible to **paginate**. For example, with a URL like http://ewolff.com/order?from=23&to=42.
- The events can be filtered as well: http://ewolff.com/order?name=Wolff.

Of course, **pagination and filtering can be combined with caching**. However, if each client uses its own pagination and filters, caching on the server-side can become inefficient, because too much different data has to be stored for the different clients.

Therefore, it may be necessary to **restrict the pagination and filtering options** in order to increase the efficiency of the cache.

Push vs. pull

HTTP optimizations such as conditional GETs can significantly speed up communication.

- But they remain **pull mechanisms**, the client queries the server.
- In case of a **push mechanism** the client is actively notified by the server about changes.

The push model seems to be more efficient but pulls have the advantage that the client requests new data when it can actually process it. This prevents the

client from handling requests if it is under too much load.

Guaranteed delivery

Atom via HTTP cannot guarantee the delivery of the data. The server only provides the data. Whether it will be read at all is an open question.

Monitoring can help to identify problems and remedy them if necessary, so it would be very unusual if the Atom resources are never needed.

However, the HTTP protocol does not have any measures for issuing receipt acknowledgements.

Old events

In principle, an **Atom feed can contain all events that have ever occurred**. As mentioned in Events, this might be interesting for **event sourcing**, in which case a microservice can reconstruct its state by processing all old events again.

If the data on old events is already stored in a microservice, the microservice needs only to make it available.

- For example, if a service already contains all orders, it can offer this information additionally as an Atom feed if necessary.
- In this case, no additional storage of old events is necessary. Thus, the effort to make the old events accessible is very low.

QUIZ

1 What is an advantage of using a pull model?

In the next lesson, we'll look at Atom in action!