Example

In this lesson, we'll look at an example of asynchronous communication with Atom.

We'll cover the following

Introduction

The example can be found at https://github.com/ewolff/microservice-atom (https://github.com/ewolff/microservice-atom).

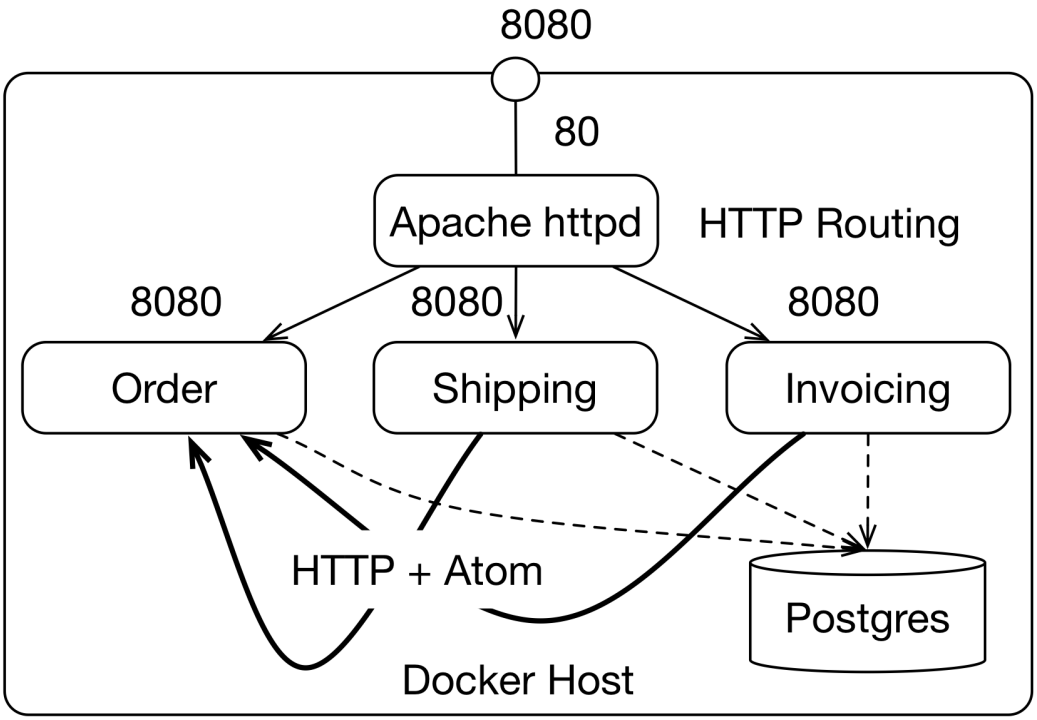
The example for Atom is analogous to the example (https://www.educative.io/collection/page/10370001/5441945024331776/6294257121886208) in the Kafka chapter and is based on the example for events (https://www.educative.io/collection/page/10370001/5441945024331776/6644855302258688/) from chapter 6.

- The **ordering** system creates orders

- The **ordering** system creates orders.

- Based on the data in the order, the **invoicing** microservice creates invoices.
- The **shipping** microservice creates deliveries.

The data models and database schemas are identical to the Kafka example. Only the **communication is now done via Atom**.



Overview of the Atom System

The drawing above shows how the example is structured:

- The **Apache httpd** distributes calls to the microservices.
    - For this purpose, the Apache httpd uses **Docker Compose** service links.
    - Docker compose offers simple **load balancing**. The Apache httpd uses the load balancing of Docker compose to forward external calls to one of the microservice instances.
- The **order microservice** offers an Atom feed from which the

invoicing and shipping microservice can read the information about new orders.

- All microservices use the same **Postgres database**.
    - Within the database, each microservice has its own separate database schema. Thus, the microservices are completely independent regarding the database schema.
    - At the same time, **one database instance is enough to run all microservices**.

Running the example

To start the environment, press `run`.

The Apache httpd load balancer will be available at port 8080, at the link generated below such as: https://x6jr4kg.educative.run/ (https://x6jr4kg.educative.run/). From there you can use the other microservices to add new orders that will eventually also appear in the invoicing and shipping microservice.

You can explore the code running Linux commands such as `ls` (list files in the directory), `cd` (change directory), `cat` (print contents of file), and `pdw` (print working directory) in the given terminal. You can even edit and create more files with `nano` and `vim`.

https://github.com/ewolff/microservice-atom/blob/master/HOW-TO-RUN.md (https://github.com/ewolff/microservice-atom/blob/master/HOW-TO-RUN.md) describes the necessary steps in detail for running the example locally.

```
version: '3'
services:
  apache:
    image: educative1/mapi_msatom_apache
    links:
     - order
     - shipping
     - invoicing
    ports:
     - "8080:80"
  postgres:
    image: educative1/mapi_msatom_postgres
    environment:
      POSTGRES_PASSWORD: dbpass
      POSTGRES_USER: dbuser
  order:
    image: educative1/mapi_msatom_order
    links:
     - postgres
  shipping:
    image: educative1/mapi_msatom_shipping
    links:
     - order
     - postgres
  invoicing:
    image: educative1/mapi_msatom_invoicing
    links:
     - order
     - postgres
```

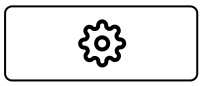
Implementation of the Atom view

The class `OrderAtomFeedView` in the project `microservice-atom-order` implements the Atom feed as a view with the framework Spring MVC.

Spring MVC splits the system into MVC (**model, view, controller**).

- The **controller** contains the **logic**.
- The **model** contains the **data**.
- The **view displays** the data from the model.
    - Spring MVC offers support for a plethora of view technologies

- Spring MVC offers support for a plethora of view technologies for HTML.



- For Atom, Spring uses the Rome Library (https://rometools.github.io/rome/). It offers different Java objects to display data as feeds and entries in the feeds.

Implementation of the controller

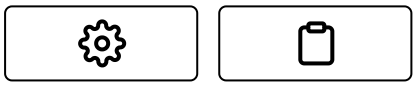
```
public ModelAndView orderFeed(WebRequest webRequest) {
  if ((orderRepository.lastUpdate() != null)
   && (webRequest.checkNotModified(orderRepository.lastUpdate().getTime()))) {
   return null;
  }
  return new ModelAndView(new OrderAtomFeedView(orderRepository),
   "orders", orderRepository.findAll());
}
```

- **(Line 1):** The method `orderFeed()` in the class `OrderController` is responsible for displaying the Atom feed with the help of `OrderAtomFeedView`.
- **(Line 6):** As shown in the listing, `OrderAtomFeedView` is returned as the view and a list of the orders as the model. The view then displays the orders from the model in the feed.

Implementation of HTTP caching on the server

- Spring provides the `checkNotModified()` method in the `WebRequest` class to simplify the handling of HTTP caching. The time of the last

update is passed to the method.



- The `lastUpdate()` method of the class `OrderRepository` determines this time point with a database query. Each order contains the time at which it was placed and `lastUpdate()` returns the maximum value.
- `checkNotModified()` compares this passed value with the value from the `If-Modified-Since` header in the request.
    - If no more recent data needs to be returned, the method returns `true`.
- Then, `orderFeed()` returns *null*, so that Spring MVC returns an HTTP status code 304 (Not Modified).

In this case, the server makes a simple query to the database and returns an HTTP response with a status code; it does not provide any data.

Implementation of HTTP Caching on the client

On the client-side, HTTP caching must also be implemented.

- **(line 1)**: The microservices `microservice-order-invoicing` and `microservice-order-shipping` implement the polling of the Atom feed in the method `pollInternal()` of the class `OrderPoller`.
- **(lines 4/5)**: They set the `If-Modified-Since` header in the request. The value is determined from the variable `lastModified`.
- **(lines 15-17)**: It contains the value of the `Last-Modified` header of the last HTTP response. If no data has been changed in the meantime, the server responses to the GET request directly with an HTTP status `304` and it is clear that no new data exists.
- **(line 11)** Accordingly, data is processed only if the status is not

- **(line 11)** Accordingly, data is processed only if the status is not `NOT_MODIFIED`.

```
public void pollInternal() {
  HttpHeaders requestHeaders = new HttpHeaders();
  if (lastModified != null) {
    requestHeaders.set(HttpHeaders.IF_MODIFIED_SINCE,
     DateUtils.formatDate(lastModified));
  }
  HttpEntity<?> requestEntity = new HttpEntity(requestHeaders);
  ResponseEntity<Feed> response =
    restTemplate.exchange(url, HttpMethod.GET, requestEntity, Feed.class);

  if (response.getStatusCode() != HttpStatus.NOT_MODIFIED) {
    Feed feed = response.getBody();
    ... // evaluate feed data
    if (response.getHeaders().getFirst(HttpHeaders.LAST_MODIFIED) != null) {
      lastModified =
       DateUtils.parseDate(
        response.getHeaders().getFirst(HttpHeaders.LAST_MODIFIED));
    }
  }
}
```

- `pollInternal()` is called by the method `poll()` in the class `OrderPoller`. The user can call this method with a button in the web UI.
- In addition, the microservice calls the method every 30 seconds because of the `@Scheduled` annotation.

Data processing and scaling

If there are **multiple instances** of invoicing and shipping microservices, **each instance polls the Atom feed and processes the data**.

Of course, it is not correct that several instances write an invoice for an order or initiate a delivery because then an order would create multiple invoices or deliveries.

Therefore, **each instance must determine what orders are already processed** and what data is in the database. If another instance has already created the data record for the invoice or delivery of the order, then the entry from the Atom feed for the order must be ignored.

To do this, `ShippingService` and `InvoicingService` use a transaction in which a data record is first searched for in the database. **A new data record is written only if none yet exists**. Therefore, only one instance of the microservices can write the data record.

All others read the data and find out that another instance has already written a data record. With a very large number of instances, this can cause a considerable load on the database. In return, the services are **idempotent**. No matter how often they are called, the state in the database, in the end, is always the same.

Atom cannot send data to a single recipient

This is a disadvantage of Atom. It is not easy to send a message to exactly one instance of a microservice. Instead, the **application has to deal with multiple instances reading the message from the Atom feed**.

Thus, especially when a lot of point-to-point communication is necessary, the Atom approach can be disadvantageous.

The **application must also be able to deal with messages not being processed**:

- If a message has been read, the process can fail before the message has been processed.

- As a result, no data might be written for some messages.

- In this case, however, another instance of the microservice would eventually read the message and process it, so **retries are actually quite easy to implement with Atom**.

In the next lesson, we'll discuss some recipe variations and experiments you can do based on them.

← Back

Atom Caching

Next →

Variations

Mark as Completed

Report an Issue