End to End Implementation of Reactive Streams: from Server to Client

# Problem Statement

We will be implementing Reactive Streaming on the server side, and then consume the same using HTML and JavaScript on the front end. At the server side, we will be using Spring Boot 2.0, Spring Webflux and Netty Server. In the front end, we will be using Server Sent Events to consume this Stream from the server. Please note that only Netty gives the real power of reactive, publishing data on stream and passing it on to the client immediately. If you use Tomcat or any other server, the client will recieve the data only when all of the data is published on the Stream. So, the real power of Reactive Streams is not realised.

# Project Configuration

This would be a Spring Boot 2.0 application. Include the modules spring-boot-starter-web and spring-boot-starter-webflux. In order to run with Netty, you would need to exclude the spring-boot-starter-tomcat from spring-boot-starter-web. Netty is already included with spring-boot-starter-webflux.

The pom.xml woul look like:

<parent>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-parent</artifactId>

<version>2.0.0.RELEASE</version>

<relativePath /> <!-- lookup parent from repository -->

</parent>

<dependencies>

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-web</artifactId>

<exclusions>

<exclusion>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-tomcat</artifactId>

</exclusion>

</exclusions>

</dependency>

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-webflux</artifactId>

</dependency>

# The Server Side Code

## How to make the server reactive?

### Dao

We should start from the Dao layer, as we would be reading the data from a DB. We want to publish the data as soon as we read it from the DB, without storing it in memory. We use the FluxSink for publishing data asynchronously. This is how the Dao looks like:

**public** **interface** BankDetailDao {

**void** publishBankDetails(FluxSink<BankDetail> fluxSink);

}

Note that the method returns null, as we publish the data on to the FluxSink. In the DaoImpl, we use the ResultSetExtractor with the JdbcTemplate to query and extract data. We make the ResultSetExtractor, as lambda function, return null. There is a check on fluxSink.isCancelled(). This helps us terminate the loop when the client closes the connection.

@Override

**public** **void** publishBankDetails(FluxSink<BankDetail> fluxSink) {

jdbcTemplate.query(***BANK\_DETAILS\_SQL***, (ResultSet rs) -> {

***LOGGER***.info("start publishing...");

**int** rowCount = 0;

**while** (rs.next()) {

**if** (fluxSink.isCancelled()) {

***LOGGER***.info("publishing is cancelled");

**return** **null**;

}

fluxSink.next(mapResultSet(rs));

**if** (++rowCount % 2 == 0) {

**try** {

***LOGGER***.info("in delay...");

Thread.*sleep*(1000);

} **catch** (InterruptedException e) {

***LOGGER***.error("error", e);

}

}

}

***LOGGER***.info("completed publishing");

fluxSink.complete();

**return** **null**;

});

}

### Service

The Service creates a Flux and passes the FluxSink on to the Dao.

**public** **interface** BankDetailService {

Flux<BankDetail> getBankDetailsReactive();

}

@Service

**public** **class** BankDetailServiceImpl **implements** BankDetailService {

**private** **final** BankDetailDao bankDetailDao;

@Override

**public** Flux<BankDetail> getBankDetailsReactive() {

**return** Flux.*create*((FluxSink<BankDetail> fluxSink) -> {

bankDetailDao.publishBankDetails(fluxSink);

});

}

}

Note that Flux.*create*() is an asynchronous call.

### Controller

The Controller invokes the srvice and returns the Flux that the service returns.

@RestController

@RequestMapping("/bank-item")

**public** **class** BankDetailController {

**private** **static** **final** Logger ***LOGGER*** = LoggerFactory.*getLogger*(BankDetailController.**class**);

**private** **final** BankDetailService bankDetailService;

@RequestMapping(value = "/reactive", method = RequestMethod.***GET***)

**public** Flux<BankDetail> getBankDetailsReactive() {

***LOGGER***.info("serving reactive content");

**return** bankDetailService.getBankDetailsReactive();

}

}

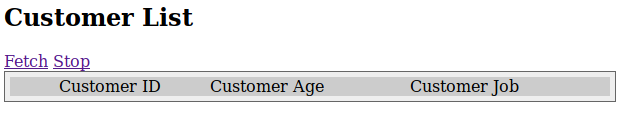
# The Front End Code

## How to build a reactive client?

We would use the ServerSentEvents to consume the stream of data published. Also, we would use RxJs to make our code reactive.

### HTML

This is how our web page looks like:



This is the source:

<html>

<head>

<link rel=*"stylesheet"* type=*"text/css"* href=*"css/reactive-demo.css"*>

</head>

<div>

<div>

<div>

<h2>Customer List</h2>

<a href=*"#"* id=*"fetch"*>Fetch</a>

<a href=*"#"* id=*"stop"*>Stop</a>

</div>

</div>

<div class=*"div-table"* id=*"tableDiv"*>

<div class=*"div-table-row"*>

<div class=*"div-table-col"* align=*"center"*>Customer ID</div>

<div class=*"div-table-col"*>Customer Age</div>

<div class=*"div-table-col"*>Customer Job</div>

</div>

</div>

</div>

<script src=*"js/Rx-5.5.6.js"*></script>

<script src=*"js/react-demo.js"*></script>

<script src=*"js/jquery-3.3.1.js"*></script>

</html>

### JavaScript

Convert the on-click event of the fetch link as a Observable, which is equivalent of a Flux on the server side:

**var** fetchButton = document.getElementById('fetch');

**var** fetchClickStream = Rx.Observable.fromEvent(fetchButton, 'click');

Write a function to trigger a server sent event, and then convert the response into a Subscribe-able function, that can be used to create an Observable:

**var** eventSourceObserver = **function** (observer) {

eventSource = **new** EventSource(url);

eventSource.onmessage = **function**(event) {

observer.next(event.data);

};

eventSource.onerror = **function**() {

console.log("EventSource failed: closing the connection");

eventSource.close();

observer.complete();

eventSource = **null**;

};

};

When the server completes, some how (I think its a bug in the garb of a feature), the eventSource.onerror() gets invoked. Make sure you close the call, otherwise, it would continue sending out requests to the server one after another.

In the reactive world, nothing happens unless you subscribe to an Observable, so:

fetchClickStream.flatMap(event => {

console.log("refresh clicked.");

**return** Rx.Observable.create(eventSourceObserver);

}).map(dataAsText => {

**return** JSON.parse(dataAsText);

}).subscribe(jsonData => {

addRow(jsonData);

});

Note the use of flatMap(): we need to use faltMap() here as the eventSourceObserver returns an Observable. The beauty of reactive programming is that we can lazily chain our data transformation logic, and then the actual action happens in the subscribe() block.

# Running this example

This is a typical Spring Boot Application and can be run by running the main class ReactiveDemoApplication. Browse to <http://localhost:8080/index.html>. Click on Fetch. You will see that as soon as the server publishes the data, it is added as a row on the table almost real time. You can also stop it at any point by clicking on the Stop link.

# Sources

<https://github.com/paawak/spring-boot-demo/tree/master/reactive/reactive-demo>

# References

<https://developer.mozilla.org/en-US/docs/Web/API/Server-sent_events/Using_server-sent_events>

<https://www.w3schools.com/html/html5_serversentevents.asp>

<http://reactivex.io/rxjs/manual/tutorial.html>

[https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#boot-features-webflux](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/" \l "boot-features-webflux)