JAX-RS RxJava

RxJava3 Flowable and Observable support

Introduction

RxJava 3 Flowable and Observable are supported on the client and the server side starting from CXF 3.2.14 / 3.3.7 as separate module, the below Maven coordinates represent state of CXF 3.2.14 / 3.3.7.

org.apache.cxf/cxf-rt-rs-extension-rx3/3.3.7 and io.reactivex.rxjava3/rxjava/3.0.2+ dependencies are required.

Client

The following simple example uses ObservableRxInvoker. org.apache.cxf.jaxrs.rx3.client.FlowableRxInvoker can be used if needed instead. Reviewing our systests for reactive may help as well.

```
@Test
public void testGetHelloWorldJson() throws Exception {
    String address = "http://localhost:" + PORT + "/rx3/observable/textJson";
    List<Object> providers = new LinkedList<>();
    providers.add(new JacksonJsonProvider());
    providers.add(new ObservableRxInvokerProvider());
    WebClient wc = WebClient.create(address, providers);
    Observable<HelloWorldBean> obs = wc.accept("application/json")
        .rx(ObservableRxInvoker.class)
        .get(HelloWorldBean.class);

    Holder<HelloWorldBean> holder = new Holder<HelloWorldBean>();
    obs.subscribe(v -> {
        holder.value = v;
    });
    Thread.sleep(2000);
    assertEquals("Hello", holder.value.getGreeting());
    assertEquals("World", holder.value.getAudience());
}
```

Server

As a method return value

One simply returns io.reactivex.rxjava3.core.Flowable from the method and the runtime will make sure the response is finalized once the Flowable flow is complete.

The only requirement is that one has to register a custom JAX-RS invoker, org.apache.cxf.jaxrs.rx3.server.ReactiveIOInvoker. It does all the default JAXRSInvoker does and only checks if Flowable is returned - if yes then it links it internally with the JAX-RS AsyncResponse. If needed, io.reactivex.rxjava3.core.Observable can be returned instead (also covered by org.apache.cxf.jaxrs.rx3.server.ReactiveIOInvoker)

Combining Flowable with AsyncResponse

```
JAXRSServerFactoryBean sf = new JAXRSServerFactoryBean();
sf.setProvider(new JacksonJsonProvider());
new ReactiveIOCustomizer().customize(sf); // use a JAXRSServerFactoryCustomizationExtension to customize the
server
sf.setResourceClasses(RxJava3FlowableService.class);
sf.setResourceProvider(RxJava3FlowableService.class,
                       new SingletonResourceProvider(new RxJava3FlowableService(), true));
sf.setAddress("http://localhost:" + PORT + "/");
server = sf.create();
```

```
import org.apache.cxf.jaxrs.reactivestreams.server.JsonStreamingAsyncSubscriber;
import io.reactivex.rxjava3.core.Flowable;
import io.reactivex.rxjava3.schedulers.Schedulers;

    // return a JSON array, write to the output stream as soon as the next JSON object becomes available
@GET
@Produces("application/json")
@Path("textJsonImplicitListAsyncStream")
public void getJsonImplicitListStreamingAsync(@Suspended AsyncResponse ar) {
    Flowable.just("Hello", "Ciao")
        .map(s -> new HelloWorldBean(s))
        .subscribeOn(Schedulers.computation())
        .subscribe(new JsonStreamingAsyncSubscriber<HelloWorldBean>(ar));
}
```

RxJava2 Flowable and Observable support

Introduction

RxJava 2 Flowable and Observable are supported on the client and the server side starting from CXF 3.2.0. Starting with CXF 3.2.3, RxJava and RxJava2 support were split into different modules, the below Maven coordinates represent state of CXF 3.2.3.

org.apache.cxf/cxf-rt-rs-extension-rx2/3.2.3 and io.reactivex.rxjava2/rxjava/2.1.3+ dependencies are required.

Client

The following simple example uses ObservableRxInvoker. org.apache.cxf.jaxrs.rx2.client.FlowableRxInvoker can be used if needed instead. Reviewing our systests for reactive may help as well.

```
@Test
public void testGetHelloWorldJson() throws Exception {
    String address = "http://localhost:" + PORT + "/rx2/observable/textJson";
    List<Object> providers = new LinkedList<>();
    providers.add(new JacksonJsonProvider());
    providers.add(new ObservableRxInvokerProvider());
    WebClient wc = WebClient.create(address, providers);
    Observable<HelloWorldBean> obs = wc.accept("application/json")
        .rx(ObservableRxInvoker.class)
        .get(HelloWorldBean.class);

    Holder<HelloWorldBean> holder = new Holder<HelloWorldBean>();
    obs.subscribe(v -> {
        holder.value = v;
    });
    Thread.sleep(2000);
    assertEquals("Hello", holder.value.getGreeting());
    assertEquals("World", holder.value.getAudience());
}
```

Server

As a method return value

One simply returns io.reactivex.Flowable from the method and the runtime will make sure the response is finalized once the Flowable flow is complete.

The only requirement is that one has to register a custom JAX-RS invoker, org.apache.cxf.jaxrs.rx2.server.ReactiveIOInvoker. It does all the default JAXRSInvoker does and only checks if Flowable is returned - if yes then it links it internally with the JAX-RS AsyncResponse. If needed, io.reactivex.Observable can be returned instead (also covered by org.apache.cxf.jaxrs.rx2.server.ReactiveIOInvoker).

Combining Flowable with AsyncResponse

```
JAXRSServerFactoryBean sf = new JAXRSServerFactoryBean();
sf.setProvider(new JacksonJsonProvider());
new ReactiveIOCustomizer().customize(sf); // use a JAXRSServerFactoryCustomizationExtension to customize the
server
sf.setResourceClasses(RxJava2FlowableService.class);
sf.setResourceProvider(RxJava2FlowableService.class,
                       new SingletonResourceProvider(new RxJava2FlowableService(), true));
sf.setAddress("http://localhost:" + PORT + "/");
server = sf.create();
```

```
import org.apache.cxf.jaxrs.reactivestreams.server.JsonStreamingAsyncSubscriber;
import io.reactivex.Flowable;
import io.reactivex.schedulers.Schedulers;

    // return a JSON array, write to the output stream as soon as the next JSON object becomes available
    @GET
    @Produces("application/json")
    @Path("textJsonImplicitListAsyncStream")
    public void getJsonImplicitListStreamingAsync(@Suspended AsyncResponse ar) {
        Flowable.just("Hello", "Ciao")
            .map(s -> new HelloWorldBean(s))
            .subscribeOn(Schedulers.computation())
            .subscribe(new JsonStreamingAsyncSubscriber<HelloWorldBean>(ar));
    }
```

RxJava1 rx.Observable support

Introduction

RxJava 1 rx.Observable is supported on the client and the server side starting from CXF 3.2.0.

org.apache.cxf/cxf-rt-rs-extension-rx/3.2.3 and io.reactivex/rxjava/1.3.0 dependencies are required.

Client

```
import org.apache.cxf.jaxrs.rx.client.ObservableRxInvoker;
import org.apache.cxf.jaxrs.rx.client.ObservableRxInvokerProvider;
import rx.Observable;

    @Test
    public void testObservableWithWebClient() throws Exception {
        String address = "http://localhost:" + PORT + "/observable/textAsync";
        WebClient wc = WebClient.create(address,
                                        Collections.singletonList(new ObservableRxInvokerProvider()));
        Observable<String> obs = wc.accept("text/plain")
            .rx(ObservableRxInvoker.class)
            .get(String.class);
        obs.map(s -> {
            return s + s;
        });

        Thread.sleep(3000);

        obs.subscribe(s -> assertDuplicateResponse(s));
    }
    @Test
    public void testObservableJaxrs21With404Response() throws Exception {
        String address = "http://localhost:" + PORT + "/observable/textAsync404";
        Invocation.Builder b = ClientBuilder.newClient().register(new ObservableRxInvokerProvider())
            .target(address).request();
        b.rx(ObservableRxInvoker.class).get(String.class).subscribe(
            s -> {
                fail("Exception expected");
            },
            t -> validateT((ExecutionException)t));
    }

    private void validateT(ExecutionException t) {
        assertTrue(t.getCause() instanceof NotFoundException);
    }
    private void assertDuplicateResponse(String s) {
        assertEquals("Hello, world!Hello, world!", s);
    }
```

Server

As a method return value

One simply returns an Observable from the method and the runtime will make sure the response is finalized once the Observable flow is complete.

The only requirement is that one has to register a custom JAX-RS invoker, org.apache.cxf.jaxrs.rx.server.ObservableInvoker. It does all the default JAXRSInvoker does and only checks if Observable is returned - if yes then it links it internally with the JAX-RS AsyncResponse.

```
JAXRSServerFactoryBean sf = new JAXRSServerFactoryBean();
sf.setProvider(new JacksonJsonProvider());
new ObservableCustomizer().customize(sf); // use a JAXRSServerFactoryCustomizationExtension to customize the
server
sf.setResourceClasses(RxJavaObservableService.class);
sf.setResourceProvider(RxJavaObservableService.class,
                       new SingletonResourceProvider(new RxJavaObservableService(), true));
sf.setAddress("http://localhost:" + PORT + "/");
server = sf.create();
```