Marcin Warczygłowa

Nov 10 2015

Real-time Web Application with Websockets and Vert.x

At Allegro, you can sell items at a fixed price (buy now) or at auction. Auctions are still a popular sales format, especially in categories such as antiques and art or clothing. So far, buyers fighting for an item had to refresh the web page in the last seconds of the auction to verify that the offer had not been overbid. This made bidding difficult and less fun. Last year real time bidding process for all mobile users was introduced. In this article I want to show how to create a simple application that provides real-time bidding, based on Allegro auctions. We will use WebSockets, SockJS and the latest, third version of Vert.x. We will create a frontend for fast bidding that communicates with a microservice written in Java 8 and based on Vert.x.

What are Websockets?

WebSocket is asynchronous, bidirectional, full-duplex protocol that provides a communication channel over a single TCP connection. With the WebSocket API it provides bidirectional communication between the website and a remote server. Originally, WebSocket was supposed to be a part of the HTML 5 specification, but a later revision of the protocol is described in a separate document RFC 6455.

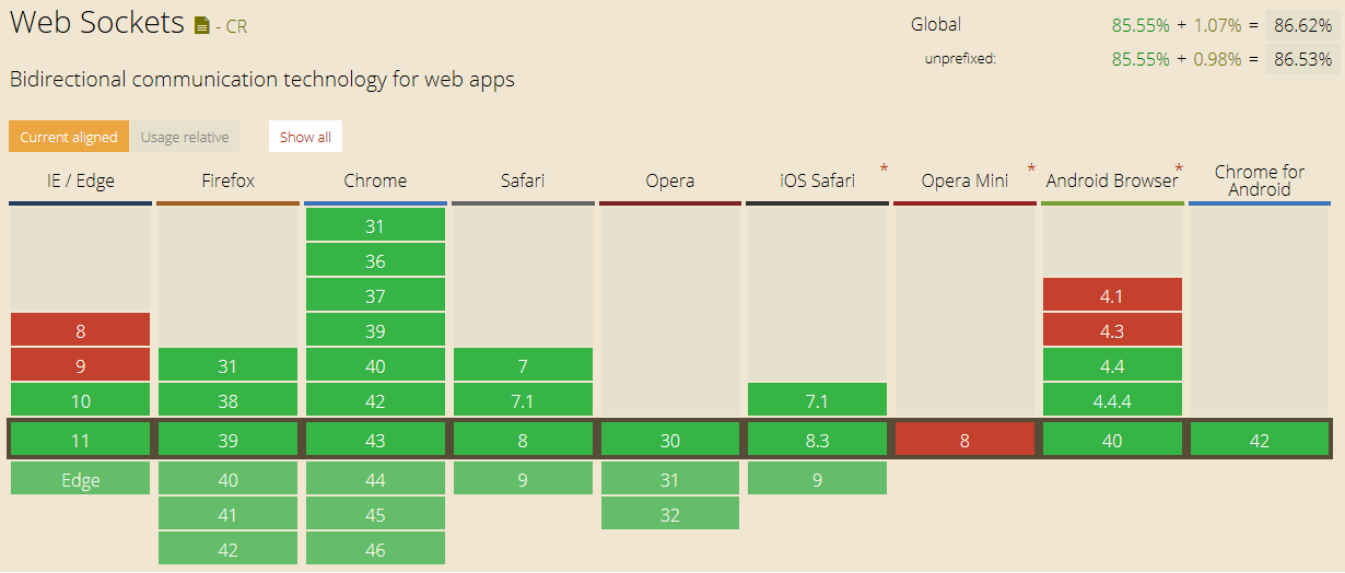
WebSockets solve many problems which prevented the HTTP protocol from being suitable for use in modern, real-time applications. Workarounds like polling are no longer needed, which simplifies application architecture. WebSockets do not need to open multiple HTTP connections, they provide a reduction of unnecessary network traffic and reduce latency.

Each WebSockets connection begins as an HTTP request. In addition, an updated HTTP header indicates that the client wants to change the connection to WebSocket protocol. The initial

Sportradar 1030
Page 1

HTTP connection is replaced by a WebSocket connection using the same underlying TCP/IP connection. At this point, each side can start sending data.

WebSockets are supported by most web browsers (source):



Some examples of good use cases for WebSockets include:

- chat applications
- multiplayer games
- social feeds
- collaborative editing or coding
- sports updates

Websocket API vs SockJS

Unfortunately, WebSockets are not supported by all web browsers. However, there are libraries that provide a fallback when WebSockets are not available. One such library is SockJS. SockJS starts from trying to use the WebSocket protocol. However, if this is not possible, it uses a variety of browser-specific transport protocols. SockJS is a library designed to work in all modern browsers and in environments that do not support WebSocket protocol, for instance behind restrictive corporate proxy. SockJS provides an API similar to the standard WebSocket API. A simple example of using the SockJS library might look like the one below. First, load SockJS library:

```
<script src="//cdn.jsdelivr.net/sockjs/0.3.4/sockjs.min.js"></script>
```

Then we establish the connection to the SockJS server:

```
var sock = new SockJS('http://mydomain.com/my_prefix');

sock.onopen = function() {
    console.log('open');
};

sock.onmessage = function(e) {
    console.log('message', e.data);
};

sock.onclose = function() {
    console.log('close');
};

sock.send('test');
sock.close();
```

Vert.x

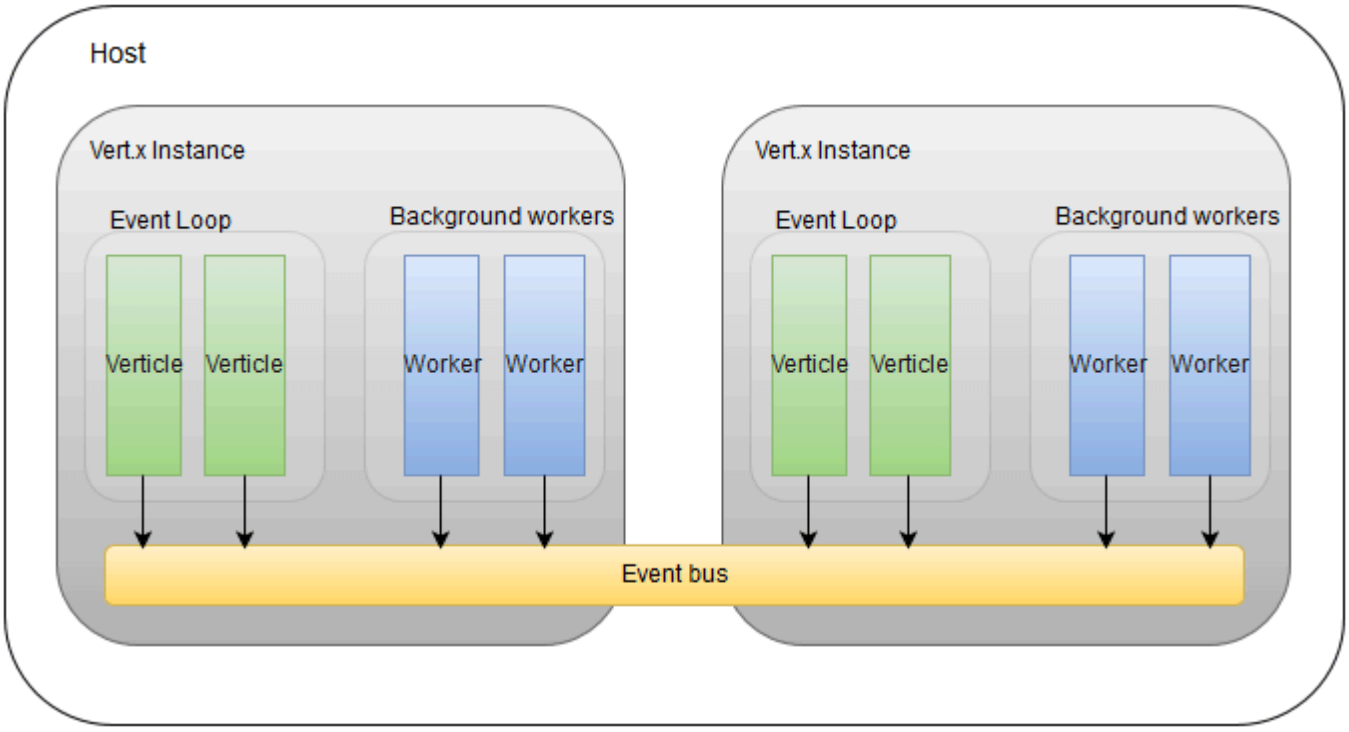
SockJS client requires the server-side part. For the Java language we can use, among other things, Spring Framework Java client & server, Atmosphere Framework or Vert.x. We are going to use the latter.

Vert.x is a polyglot, non-blocking, event-driven tool-kit for building applications on the JVM. Vert.x is pretty fast, which you can see on TechEmpower Benchmarks. The packages of code that Vert.x executes are called verticles. Verticles can be written in Java, Groovy, Ruby, JavaScript as well as in several programming languages mixed and matched in a single application. Many verticles can be executed concurrently in the same Vert.x instance. A single Vert.x instance runs inside its own JVM instance. Vert.x guarantees that a particular verticle instance is never executed by multiple threads concurrently. Verticles communicate by passing messages using an event bus.

Vert.x applications are mostly written by defining event handlers. Vert.x calls handlers using a thread called an event loop. The event loop delivers events to different handlers in succession as they arrive. None of the Vert.x APIs block threads, so you also need to remember not to block the event loop in handlers. Because nothing blocks, an event loop can potentially deliver a lot of events in a short time. We make guarantees that any specific handler will always be invoked by the same event loop. This means you can write your code as single threaded. Vert.x instance maintains several event loops. The default number of event loops is determined by the number of available cores on the machine.

There are two main types of verticles: standard and worker verticles. Standard verticles are always executed using an event loop thread. Workers are designed for executing blocking code. Workers are like standard verticles but use threads from a special worker thread pool. An alternative way to run blocking code is to use `executeBlocking` method directly from an event loop.

Typical application will consist of multiple verticles running on Vert.x instance:



There can be many Vert.x instances running on the same host or on different hosts on the network. Instances can be configured to cluster with each other forming a distributed event bus over which verticles can communicate. We can create a distributed bus encompassing many browsers and servers.

Frontend to fast bidding

Auction web page contains the bidding form and some simple JavaScript which loads current price from the service, opens an event bus connection to the SockJS server and offers bidding. HTML source code of sample web page on which we bid might look like this:

```
<h3>Auction 1</h3>
<div id="error_message"></div>
<form>
    Current price:
    <span id="current_price"></span>
    <div>
        <label for="my_bid_value">Your offer:</label>
        <input id="my_bid_value" type="text">
        <input type="button" onclick="bid();" value="Bid">
    </div>
    <div>
        Feed:
        <textarea id="feed" rows="4" cols="50" readonly></textarea>
    </div>
</form>
```

We use the **`vertxbus.js`** library to create a connection to the event bus. **`Vertxbus.js`** library is a part of the Vert.x distribution. **`Vertxbus.js`** internally uses SockJS library to send the data to the SockJS server. In the code snippet below we create an instance of the event bus. The parameter to the constructor is the URI where to connect to the event bus. Then we register the handler listening on address **`auction.<auction_id>`** . Each client has a possibility of registering at multiple addresses e.g. when bidding in the auction 1234, they register on the address **`auction.1234`** etc. When data arrives in the handler, we change the current price and the bidding feed on the auction's web page.

```
function registerHandlerForUpdateCurrentPriceAndFeed() {
    var eventBus = new vertx.EventBus('http://localhost:8080/eventbus');
    eventBus.onopen = function () {
        eventBus.registerHandler('auction.' + auction_id, function (message) {
            document.getElementById('current_price').innerHTML = JSON.parse(message).price;
            document.getElementById('feed').value += 'New offer: ' + JSON.parse(message).price +
```


Sportradar 1030
Page 5

```
        });
    }
};
```

Any user attempt to bid generates a PATCH Ajax request to the service with information about the new offer made at auction (see `bid()` function). On the server side we publish this information on the event bus to all clients registered to an address. If you receive an HTTP response status code other than `200 (OK)`, an error message is displayed on the web page.

```
function bid() {
    var newPrice = document.getElementById('my_bid_value').value;

    var xmlhttp = (window.XMLHttpRequest) ? new XMLHttpRequest() : new ActiveXObject("Microsoft.XM
    xmlhttp.onreadystatechange = function () {
        if (xmlhttp.readyState == 4) {
            if (xmlhttp.status != 200) {
                document.getElementById('error_message').innerHTML = 'Sorry, something went wrong.
            }
        }
    };
    xmlhttp.open("PATCH", "http://localhost:8080/api/auctions/" + auction_id);
    xmlhttp.setRequestHeader("Content-Type", "application/json");
    xmlhttp.send(JSON.stringify({price: newPrice}));
};
```

Auction Service

Now we are going to create a light-weight RESTful auction service. We will send and retrieve data in JSON format. Let's start by creating a verticle, the basic package of code that Vert.x executes. First we need to inherit from `AbstractVerticle` and override the `start` method. Each verticle instance has a member variable called `vertx`. This provides access to the Vert.x core API. The core API is used to do most things in Vert.x, including HTTP, file system access, event bus etc. For example, to create an HTTP server you call the `createHttpServer` method on `vertx` instance. To tell the server to listen on port 8080 for incoming requests you use the `listen` method.

We need a router with routes. A router takes an HTTP request and finds the first matching route. The route can have a handler associated with it, which receives the request (e.g. route that matches path `/eventbus/*` is associated with `eventBusHandler`). We can do something with the request, and then, end it or pass it to the next matching handler. If you have a lot of handlers it makes sense to split them up into multiple routers. You can do this by mounting a router at a mount point in another router (see `auctionApiRouter` that corresponds to `/api` mount point in code snippet below).

Here's an example verticle:

```
public class AuctionServiceVerticle extends AbstractVerticle {

    @Override
    public void start() {
        Router router = Router.router(vertx);

        router.route("/eventbus/*").handler(eventBusHandler());
        router.mountSubRouter("/api", auctionApiRouter());
        router.route().failureHandler(errorHandler());
        router.route().handler(staticHandler());

        vertx.createHttpServer().requestHandler(router::accept).listen(8080);
    }

    //…
}
```

Now we'll look at things in more detail. We'll discuss Vert.x features used in verticle: error handler, SockJS handler, body handler, shared data, static handler and routing based on method, path etc.

Error handler

As well as setting handlers to handle requests you can also set a handler for failures in routing. Failure in routing occurs if a handler throws an exception, or if a handler calls `fail` method. To render error pages we use error handler provides by Vert.x:

```
private ErrorHandler errorHandler() {
    return ErrorHandler.create();
}
```

SockJS handler

Vert.x provides SockJS handler with the event bus bridge which extends the server-side Vert.x event bus into client side JavaScript.

Configuring the bridge to tell it which messages should pass through is easy. You can specify which matches you want to allow for inbound and outbound traffic using the `BridgeOptions`. If a message is outbound, before sending it from the server to the client side JavaScript, Vert.x will look through any outbound permitted matches. In code snippet below we allow any messages from addresses starting with "auction." and ending with digits (e.g. `auction.1`, `auction.100` etc).

If you want to be notified when an event occurs on the bridge you can provide a handler when calling the bridge. For example, `SOCKET_CREATED` event will occur when a new SockJS socket is created. The event is an instance of `Future`. When you are finished handling the event you can complete the future with "true" to enable further processing.

To start the bridge simply call `bridge` method on the SockJS handler:

```
private SockJSHandler eventBusHandler() {
    BridgeOptions options = new BridgeOptions()
            .addOutboundPermitted(new PermittedOptions().setAddressRegex("auction\\.[0-9]+"));
    return SockJSHandler.create(vertx).bridge(options, event -> {
        if (event.type() == BridgeEvent.Type.SOCKET_CREATED) {
            logger.info("A socket was created");
        }
        event.complete(true);
    });
}
```

Body handler

The BodyHandler allows you to retrieve the request body, limit the body size and to handle the file upload. Body handler should be on a matching route for any requests that require this functionality. We need BodyHandler during the bidding process (PATCH method request **`/auctions/<auction_id>`** contains request body with information about a new offer made at auction). Creating a new body handler is simple:

```
BodyHandler.create();
```

If request body is in JSON format, you can get it with **`getBodyAsJson`** method.

Shared data

Shared data contains functionality that allows you to safely share the data between different applications in the same Vert.x instance or across a cluster of Vert.x instances. Shared data includes local shared maps, distributed, cluster-wide maps, asynchronous cluster-wide locks and asynchronous cluster-wide counters.

To simplify the application we use the local shared map offer by Vert.x to save information about auctions. The local shared map allows you to share data between different verticles in the same Vert.x instance. To prevent issues due to mutable data, Vert.x only allows simple immutable types such as number, string, Boolean or Buffer to be used in local shared map. Here's an example of using a shared local map in an auction service:

```
public class AuctionRepository {

    //…

    public Optional<Auction> getById(String auctionId) {
        LocalMap<String, String> auctionSharedData = this.sharedData.getLocalMap(auctionId);

        return Optional.of(auctionSharedData)
            .filter(m -> !m.isEmpty())
            .map(this::convertToAuction);
    }

    public void save(Auction auction) {
        LocalMap<String, String> auctionSharedData = this.sharedData.getLocalMap(auction.getId());
```


Sportradar 1030
Page 9

```
        auctionSharedData.put("id", auction.getId());
        auctionSharedData.put("price", auction.getPrice());
    }

    //…
}
```

If you want to store auction data in a database, Vert.x provides a few different asynchronous clients for accessing various data storages (MongoDB, Redis or JDBC client).

Auction API

Vert.x lets you route HTTP requests to different handlers based on pattern matching on the request path. It also enables you to extract values from the path and use them as parameters in the request. Corresponding methods exist for each HTTP method. You can provide as many matchers as you like and they are evaluated in the order you added them. The first matching one will receive the request. If no routes match the request, a `404` status will be returned. This functionality is particularly useful when developing REST-style web applications. If you would like to read something more about designing RESTful APIs see the article Designing RESTful API.

To extract parameters from the path, you can use the colon character to denote the name of a parameter. Regular expressions can also be used to extract more complex matches. Any parameters extracted by pattern matching are added to the map of request parameters.

`Consumes` describes which MIME types the handler can consume. By using `produces` you define which MIME types the route produces. In the code below the routes will match any request with `content-type` header and `accept` header that matches `application/json` .

Let's look at an example of a subrouter mounted on the main router which was created in `start` method in verticle:

```
private Router auctionApiRouter() {
    AuctionRepository repository = new AuctionRepository(vertx.sharedData());
    AuctionValidator validator = new AuctionValidator(repository);
    AuctionHandler handler = new AuctionHandler(repository, validator);
```

```
    Router router = Router.router(vertx);
    router.route().handler(BodyHandler.create());

    router.route().consumes("application/json");
    router.route().produces("application/json");

    router.get("/auctions/:id").handler(handler::handleGetAuction);
    router.patch("/auctions/:id").handler(handler::handleChangeAuctionPrice);

    return router;
}
```

The GET request returns auction data, while the PATCH method request allows you to bid up in the auction. Let's focus on the more interesting method, namely `handleChangeAuctionPrice`. In the simplest terms, the method might look like this:

```
public void handleChangeAuctionPrice(RoutingContext context) {
    String auctionId = context.request().getParam("id");
    Auction auction = new Auction(
        auctionId,
        new BigDecimal(context.getBodyAsJson().getString("price"))
    );

    this.repository.save(auction);
    context.vertx().eventBus().publish("auction." + auctionId, context.getBodyAsString());

    context.response()
        .setStatusCode(200)
        .end();
}
```

PATCH request to `/auctions/1` would result in variable `auctionId` getting the value 1. We save a new offer in the auction and then publish this information on the event bus to all clients registered on the address on the client side JavaScript. After you have finished with the HTTP response you must call the `end` function on it. If you don't end the response in handler, you should call `next` so that another matching route can handle the request.

Static handler

Vert.x provides the handler for serving static web resources. The default directory from which static files are served is `webroot`, but this can be configured. By default the static handler will set cache headers to enable browsers to cache files. Setting cache headers can be disabled with `setCachingEnabled` method. To serve the auction HTML page, JS files (and other static files) from auction service, you can create a static handler like this:

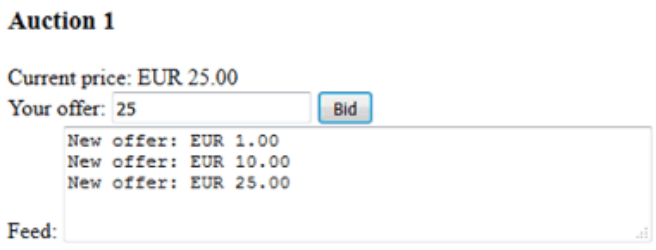
```
private StaticHandler staticHandler() {
    return StaticHandler.create()
        .setCachingEnabled(false);
}
```

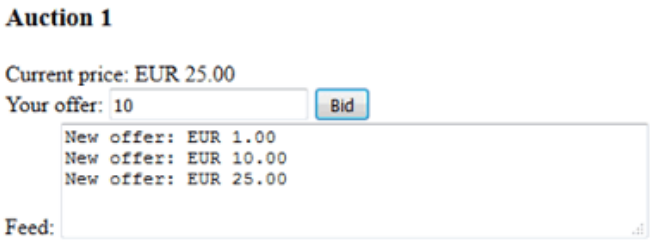
Let's run!

Full application code is available on github.

Clone the repository and run `./gradlew run`.

Open one or more browsers and point them to `http://localhost:8080`. Now you can bid in auction:





Summary

The expectations of users for interactivity with web applications have changed over the past few years. Users during bidding in auction no longer want to press the refresh button to check if the price has changed or the auction is over. Instead, they expect to see the updates in application in real-time.

This article presents the outline of a simple application that allows real-time bidding. Due to the fact that WebSockets are not supported by all browsers we used the SockJS library. We created a lightweight, high-performance and scalable microservice written in Java and based on Vert.x. We discussed what Vert.x offers: an actor-like concurrency model, a distributed event bus and an elegant API that allows you to create applications in no time.

The version 3.0 of Vert.x brought a lot of interesting features. I hope this article encouraged you to familiarize yourself with the capabilities offered by this tool-kit.

java javascript vertx websockets websocket api sockjs

real-time application

Discussion

2 Comments

Login

G

Join the discussion…

LOG IN WITH

OR SIGN UP WITH DISQUS ?

Name

4 Share

Best Newest Oldest

S Suresh Kesavan

4 years ago edited

Thanks, it is very helpful. Please share, if you have the example for sending image and document data like pdf, doc over the Websocket.

0 0 Reply

Binh Thanh Nguyen

7 years ago

Thanks, nice explanation.

0 0 Reply

Subscribe Privacy Do Not Sell My Data

allegro Tech

© 2024 Allegro

Proudly built by engineers