Exercise 1: Simple Vert.x Hello World Web Server

1. Download the FULL Vert.x distribution from http://vertx.io/ and ensure that the **vertx** command is in your path (vertx/bin/vertx)
2. Create a new file **Exercise1.java** with the following contents:

Exercise1/Exercise1.java

```
import io.vertx.core.AbstractVerticle;
import io.vertx.core.json.JsonObject;
import io.vertx.core.http.HttpServer;

public class Exercise1 extends AbstractVerticle {

    public void start() {
        // Create a new HttpServer
        HttpServer server = vertx.createHttpServer();

        // Create a JSON response
        String response = new JsonObject().put("ok", true).encode();

        // Register a request handler for the HttpServer
        server.requestHandler(req -> req.response().end(response));

        // Listen on port 8080 and interface `127.0.0.1`
        server.listen(8080, "127.0.0.1");
    }
}
```

3. Run the verticle with the command `vertx run Exercise1.java`
4. You should see a message like: `Succeeded in deploying verticle`
5. Open a browser and point it at: http://localhost:8080/

Next Steps: (see HttpServerResponse)

- Modify the example above to add a `Content-Type` response header
- Modify the example above to add an HTTP response code of 201 to the response
- Modify the example above to add an HTTP reason phrase of 'IDUNNO' to the response

Exercise 2: Are you fluent?!?!

Vert.x APIs are written to be fluent. This means that you can chain method calls together so that they form a sort of domain specific language which CAN be easier to read. We will modify our first example to use the fluent API in Vert.x to perform the same operations.

Exercise2/Exercise2.java

```
import io.vertx.core.AbstractVerticle;
import io.vertx.core.json.JsonObject;
import io.vertx.core.json.JsonObject;

public class Exercise2 extends AbstractVerticle {

    public void start() {
        // Create a JSON response
        String response = new JsonObject().put("ok", true).encode();

        vertx.createHttpServer()         // Create a new HttpServer
             .requestHandler(req -> {    // Register a request handler for the
HttpServer
                 req.response().end(response);
             })
             .listen(8080, "127.0.0.1"); // Listen on port 8080 and interface
`127.0.0.1`
    }
}
```

You'll see that we chained the createHttpServer() method, which returns an HttpServer object, to the requestHandler() method. We then chained the requestHandler() method to the listen() method. Each of these chained methods returns the original HttpServer object so that we can make subsequent calls in a fluent manner.

Exercise 3: Handlers

A handler in Vert.x is a form of Callback). Handlers are passed as arguments to some Vert.x methods so that the callback can be executed once a particular asynchronous operation has been completed. Handlers for Vert.x can be written in Groovy in several ways:

Exercise 3.1: Handler classes

The basic Handler in Vert.x is any class which implements the Handler interface. For example:

Exercise3/Exercise3_1.java

```
import io.vertx.core.Handler;
import io.vertx.core.AbstractVerticle;
import io.vertx.core.json.JsonObject;
import io.vertx.core.http.HttpServerRequest;

public class Exercise3_1 extends AbstractVerticle {

    private class RequestHandler implements Handler<HttpServerRequest> {
        public void handle(HttpServerRequest req) {
            String response = new JsonObject().put("ok", true).encode();
            req.response().end(response);
        }
    }

    public void start() {
        vertx.createHttpServer()                    // Create a new HttpServer
             .requestHandler(new RequestHandler())
             .listen(8080, "127.0.0.1");  // Listen on port 8080 and interface
`127.0.0.1`
    }
}
```

As you can see, we pass an instance of the RequestHandler class to the requestHandler() method on the HttpServer object and that instance will handle the HTTP requests.

Exercise 3.2: Method References

Another way to implement handlers removes some of the boiler-plate of having a separate hanlder class for each Callback we want to register. It's called a Method Reference. A method reference is a way of assigning a method to behave as a callback without having to implement a Handler interface on a new class.

Exercise3/Exercise3_2.java

```
import io.vertx.core.AbstractVerticle;
import io.vertx.core.json.JsonObject;
import io.vertx.core.http.HttpServerRequest;

public class Exercise3_2 extends AbstractVerticle {

    /**
     * Handle HttpServerRequests
     */
    public void handleRequest(HttpServerRequest req) {
        String response = new JsonObject().put("ok", true).encode();
        req.response().end(response);
    }

    public void start() {
        vertx.createHttpServer()                        // Create a new HttpServer
             .requestHandler(this::handleRequest)     // Register a request handler
             .listen(8080, "127.0.0.1");   // Listen on port 8080 and interface
`127.0.0.1`
    }
}
```

Exercise 3.3: Lambdas

Finally, in Java we can use Lambdas. Lambdas are a way of writing a bit of code which can be passed as a value . . . in-line...

Exercise3/Exercise3_3.java

```
import io.vertx.core.AbstractVerticle;
import io.vertx.core.json.JsonObject;

public class Exercise3_3 extends AbstractVerticle {

    public void start() {
        // Create a JSON response
        String response = new JsonObject().put("ok", true).encode();

        vertx.createHttpServer()         // Create a new HttpServer
             .requestHandler(req -> {    // Register a request handler
                 req.response().end(response);
             })
             .listen(8080, "127.0.0.1"); // Listen on port 8080 and interface
`127.0.0.1`
    }
}
```

An alternate way of declaring that closure would be to assign the closure to a variable and then pass the variable into the requestHandler() method as shown below:

Exercise3/Exercise3_3_1.java

```
import io.vertx.core.AbstractVerticle;
import io.vertx.core.json.JsonObject;
import io.vertx.core.http.HttpServerRequest;

public class Exercise3_3_1 extends AbstractVerticle {


    public void reqHandler(HttpServerRequest req) {
        String response = new JsonObject().put("ok", true).encode();
        req.response().end(response);
    }

    public void start() {
        vertx.createHttpServer()         // Create a new HttpServer
             .requestHandler(this::reqHandler) // Register a request handler
             .listen(8080, "127.0.0.1"); // Listen on port 8080 and interface
`127.0.0.1`
    }
}
```

Next Steps: (see HttpServerRequest)

- Modify the example above to include the requested path as an item in the JSON response body
- Modify the example above to include the request headers as a nested JSON object within the response body

Exercise 4: Using Routers

So far, we have seen that we can add a requestHandler() to an HTTP server, but what if we want to have a number of different paths which do different things in our web application? This is where the Vert.x Web module comes in. It gives us a new features like Router and RoutingContext.

Exercise4/Exercise4.java

```
import io.vertx.core.AbstractVerticle;
import io.vertx.ext.web.Router;
import io.vertx.ext.web.RoutingContext;
import io.vertx.core.json.JsonObject;

public class Exercise4 extends AbstractVerticle {

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

        router.get("/")              .handler(this::rootHandler);
        router.get("/something/else").handler(this::otherHandler);

        vertx.createHttpServer()                          // Create a new HttpServer
             .requestHandler(router::accept)              // Register a request handler
             .listen(8080, "127.0.0.1");      // Listen on 127.0.0.1:8080
    }

    public void rootHandler(RoutingContext ctx) {
        ctx.response().end(new JsonObject().put("ok", true).put("path",
ctx.request().path()).encode());
    }

    public void otherHandler(RoutingContext ctx) {
        ctx.response().end(new JsonObject().put("ok", false).put("message", "Something
Else").encode());
    }
}
```

1. You see that we added 2 different routes to the Router instance
2. Each route has a separate handler set via a method reference
3. Finally, we pass the Router's accept method via a method reference as a handler for the HttpServer's requestHandler() method.

Exercise 5: Routes with Path Parameters

In the previous example, we saw that we could specify different paths with different handlers, but what about if we want to capture information FROM the path in a programmatic manner?

Exercise5/Exercise5.java

```
import io.vertx.core.AbstractVerticle;
import io.vertx.ext.web.Router;
import io.vertx.ext.web.RoutingContext;
import io.vertx.core.json.JsonObject;

public class Exercise5 extends AbstractVerticle {

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

        router.get("/")            .handler(this::rootHandler);
        router.get("/customer/:id").handler(this::custHandler);

        vertx.createHttpServer()             // Create a new HttpServer
            .requestHandler(router::accept) // Register a request handler
            .listen(8080, "127.0.0.1");      // Listen on 127.0.0.1:8080
    }

    void rootHandler(RoutingContext ctx) {
        ctx.response().end(new JsonObject()
                                    .put("ok", true).put("path",
ctx.request().path()).encode());
    }

    void custHandler(RoutingContext ctx) {
        ctx.response().end(new JsonObject()
           .put("ok", false)
           .put("custID", ctx.request().getParam("id"))
           .encode());
    }
}
```

Next Steps: (see Router, Routing With Regular Expressions, Routing Based On MIME Types, Request Body Handling)

- Modify the example above to have a new route which had multiple path parameters
- Modify the example above to use a route with regular expressions
- Modify the example to add a new HTTP POST endpoint which consumes JSON and produces the POSTed JSON

Exercise 6: Programmatically Deploy Verticles

So far, our exercised have done all of their work in a single Verticle (HelloWorld). This is fine for simple applications, but it does not scale well for larger and more complex applications. Each Verticle is single-threaded; so in order to utilize our CPU cores effectively, we need to distribute workloads across multiple Verticles.

Exercise6/EventVerticle.java

```
import io.vertx.core.AbstractVerticle;
import io.vertx.core.eventbus.Message;
import io.vertx.core.json.JsonObject;

public class EventVerticle extends AbstractVerticle {

    @Override
    public void start() {
        vertx.eventBus().consumer("event.verticle", this::doSomething);
    }

    void doSomething(Message<JsonObject> msg) {
        if ((Math.round(Math.random()*1))==1) {
            msg.reply(msg.body());
        } else {
            msg.fail(1, "Random Failure");
        }
    }
}
```

Exercise6/Exercise6.java

```
import io.vertx.core.AsyncResult;
import io.vertx.core.AbstractVerticle;
import io.vertx.core.http.HttpServerResponse;
import io.vertx.ext.web.Router;
import io.vertx.ext.web.RoutingContext;
import io.vertx.core.json.JsonObject;
import io.vertx.core.eventbus.Message;

public class Exercise6 extends AbstractVerticle {

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

        router.get().handler(this::rootHandler);

        vertx.createHttpServer()            // Create a new HttpServer
            .requestHandler(router::accept) // Register a request handler
            .listen(8080, "127.0.0.1");      // Listen on 127.0.0.1:8080
        vertx.deployVerticle("java:EventVerticle.java");
    }

    void rootHandler(RoutingContext ctx) {
        JsonObject msg = new JsonObject().put("path", ctx.request().path());
        vertx.eventBus().send("event.verticle", msg, reply -> this.replyHandler(ctx,
reply));
    }

    void replyHandler(RoutingContext ctx, AsyncResult<Message<Object>> reply) {
        HttpServerResponse response = ctx.response()
                          .putHeader("Content-Type", "application/json");
        if (reply.succeeded()) {
            response.setStatusCode(200)
               .setStatusMessage("OK")
               .end(((JsonObject)reply.result().body()).encodePrettily());
        } else {
            response.setStatusCode(500)
               .setStatusMessage("Server Error")
               .end(new JsonObject().put("error",
reply.cause().getLocalizedMessage()).encodePrettily());
        }
    }
}
```

(**NOTE:** When using `vertx run <VerticleName>` to launch Vert.x applications, the files should be in the current working directory or a child directory referenced by it's relative path)

Several new concepts have been introduced in this example:

- The EventBus - Used to communicate between Verticles in a thread-safe manner
- Deploying Verticles Programmatically
- Handling AsyncResults via Callback
- Using Message objects - Message objects consist of JsonObject or String contents and can be replied to

Exercise 7: Deploy With Futures

Often, the application will need to ensure that certain Verticles are already up and running before proceeding to do other actions. To allow for this, Vert.x provides a way of deploying Verticles with a callback once the deployment is complete.

Exercise7/Exercise7.java

```
import io.vertx.core.AsyncResult;
import io.vertx.core.http.HttpServerResponse;
import io.vertx.core.json.JsonObject;
import io.vertx.core.eventbus.Message;
import io.vertx.core.logging.LoggerFactory;
import io.vertx.ext.web.Router;
import io.vertx.ext.web.RoutingContext;
import io.vertx.core.AbstractVerticle;

import static groovy.json.JsonOutput.toJson;

public class Exercise7 extends AbstractVerticle {

    public void start() {
        vertx.deployVerticle("java:EventVerticle.java", this::deployHandler);
    }

    void rootHandler(RoutingContext ctx) {
        JsonObject msg = new JsonObject().put("path", ctx.request().path());
        vertx.eventBus().send("event.verticle", msg, reply -> this.replyHandler(ctx,
reply));
    }

    void replyHandler(RoutingContext ctx, AsyncResult<Message<Object>> reply) {
        HttpServerResponse response = ctx.response()
                          .putHeader("Content-Type", "application/json");
        if (reply.succeeded()) {
            response.setStatusCode(200)
               .setStatusMessage("OK")
               .end(((JsonObject)reply.result().body()).encodePrettily());
        } else {
            response.setStatusCode(500)
               .setStatusMessage("Server Error")
               .end(new JsonObject().put("error",
reply.cause().getLocalizedMessage()).encodePrettily());
```

```
        }
    }

    void deployHandler(AsyncResult<String> res) {
        if (res.succeeded()) {
            LoggerFactory.getLogger("Exercise7").info("Successfully deployed
EventVerticle");

            // If the EventVerticle successfully deployed, configure and start the
HTTP server
            Router router = Router.router(vertx);

            router.get().handler(this::rootHandler);

            vertx.createHttpServer()            // Create a new HttpServer
                .requestHandler(router::accept) // Register a request handler
                .listen(8080, "127.0.0.1");      // Listen on 127.0.0.1:8080
        } else {
            // Otherwise, exit the application
            LoggerFactory.getLogger("Exercise7").error("Failed to deploy
EventVerticle", res.cause());
            vertx.close();
        }
    }
}
```

Exercise7/EventVerticle.java

```
import io.vertx.core.Future;
import io.vertx.core.json.JsonObject;
import io.vertx.core.eventbus.Message;
import io.vertx.core.AbstractVerticle;

public class EventVerticle extends AbstractVerticle {

    @Override
    public void start(Future startFuture) {
        vertx.eventBus().consumer("event.verticle", this::doSomething);

        if ((Math.round(Math.random()*1))==1) {            // Randomly succeed or fail
deployment of EventVerticle
            startFuture.complete();
        } else {
            startFuture.fail("Random deployment failure of EventVerticle");
        }
    }

    void doSomething(Message<JsonObject> msg) {
        if (Math.random()>=0.6666) {
            msg.reply(msg.body());
        } else {
            msg.fail(1, "Random Failure");
        }
    }
}
```

Next Steps:

- Modify the example above to attempt to redeploy EventVerticle in case of a failure (Use maximum of 3 retries)
- Modify the example above to deploy more than one Verticle and call the new Verticle `AnotherVerticle.java`

Exercise 8: Asynchronous Coordination

It is useful to coordinate several asynchronous operations in a single handler for certain situations. To facilitate this, Vert.x provides a CompositeFuture

Exercise8/Exercise8.java

```
import io.vertx.core.AsyncResult;
import io.vertx.core.json.JsonObject;
import io.vertx.core.logging.LoggerFactory;
import io.vertx.core.CompositeFuture;
import io.vertx.core.Future;
import io.vertx.ext.web.Router;
```

```
import io.vertx.core.AbstractVerticle;
import io.vertx.ext.web.RoutingContext;

import java.util.stream.IntStream;

public class Exercise8 extends AbstractVerticle {
    public void start() {
        Future eventVerticleFuture = Future.future();
        Future anotherVerticleFuture = Future.future();

        CompositeFuture.join(eventVerticleFuture,
anotherVerticleFuture).setHandler(this::deployHandler);

        vertx.deployVerticle("java:EventVerticle.java",
eventVerticleFuture.completer());
        vertx.deployVerticle("java:AnotherVerticle.java",
anotherVerticleFuture.completer());
    }

    protected void deployHandler(AsyncResult<CompositeFuture> cf) {
        if (cf.succeeded()) {
            LoggerFactory.getLogger("Exercise8").info("Successfully deployed all
verticles");

            // If the EventVerticle successfully deployed, configure and start the
HTTP server
            Router router = Router.router(vertx);

            router.get().handler(this::rootHandler);

            vertx.createHttpServer()            // Create a new HttpServer
                .requestHandler(router::accept) // Register a request handler
                .listen(8080, "127.0.0.1");      // Listen on 127.0.0.1:8080
        } else {
            IntStream.range(0, cf.result().size()).forEach(x -> {
                if (cf.result().failed(x)) {
                    LoggerFactory.getLogger("Exercise8").error("Failed to deploy
verticle", cf.result().cause(x));
                }
            });
            vertx.close();
        }
    }

    void rootHandler(RoutingContext ctx) {
        ctx.response().end(new JsonObject().put("ok", true).put("path",
ctx.request().path()).encode());
    }
}
```

Next Steps: (see CompositeFuture and Async Coordination) * Modify the example above to use a List of futures instead of specifying each future as a parameter. * Remove the CompositeFuture and use composed Futures to load one verticle after another

Exercise 9: Using Shared Data

While you can coordinate between verticles very well using String and JsonObject instances over the EventBus, it is sometimes better to share certain objects across multiple verticles. Vert.x makes this possible via 2 facilities.

Exercise 9.1: Shared Local Map

The Vertx.sharedData() method allows us to get an instance of LocalMap which can store most Immutable data types as well as custom types which implement the Shareable interface. Storing data in a these LocalMap instances makes those objects available to other Verticles without having to use the EventBus to send those objects. **The Shared Local Map has no concurrency controls, so the last writer is always the winner. If assurance of ordered writes is required, then the user must implement their own concurrency controls or only use data structures which ensure thread safety.**

Exercise9/Exercise9_1.java

```
import io.vertx.core.logging.LoggerFactory;
import io.vertx.core.AbstractVerticle;
import io.vertx.core.shareddata.LocalMap;

import java.util.stream.IntStream;

public class Exercise9_1 extends AbstractVerticle {

    @Override
    public void start() throws Exception {

        // Deploy AnotherVerticle 10 times
        IntStream.rangeClosed(1, 10).forEach(i -> {
            vertx.deployVerticle("java:AnotherVerticle.java");
        });

        vertx.setPeriodic(100, this::showDeployedVerticles);
    }

    void showDeployedVerticles(Long t) {
        // Print the list of deployment IDs stored in the shared data local Map
        LoggerFactory.getLogger("Exercise9_1").info("Polling shared data map");
        LocalMap<String, Object> localMap = vertx.sharedData().getLocalMap("shared");
        localMap.keySet().stream().forEach(key -> {
            System.out.println(key+" - "+localMap.get(key));
        });
        System.out.println();
        System.out.println();
    }
}
```

Exercise9/AnotherVerticle.java

```
import io.vertx.core.logging.LoggerFactory;
import io.vertx.core.AbstractVerticle;

public class AnotherVerticle extends AbstractVerticle {

    @Override
    public void start() throws Exception {
        vertx.sharedData().getLocalMap("shared").put(context.deploymentID(),
Thread.currentThread().getName());

        LoggerFactory.getLogger("AnotherVerticle").info("Deployed AnotherVerticle:
${context.deploymentID()}");
    }
}
```

Exercise 9.2: Clustered Async Map

When running in a clustered configuration, sharing objects across Vert.x nodes requires a special feature known as AsyncMap. The AsyncMap is handled by the Vert.x ClusterManager, which is responsible for ensuring that access to the AsyncMap data is handled in a cluster/thread-safe way. In order to use the AsyncMap, Vert.x **MUST** be started in a clustered mode using `vertx run -cluster <Verticle>`

Exercise9/Exercise9_2.java

```
import io.vertx.core.logging.LoggerFactory;
import io.vertx.core.AbstractVerticle;

import java.util.Arrays;
import java.util.List;
import java.util.stream.IntStream;

public class Exercise9_2 extends AbstractVerticle {

    @Override
    public void start() throws Exception {

        // Get a reference to clusterWide map called "shared"
        vertx.sharedData().getClusterWideMap("shared", res -> {
            if (res.succeeded()) {
                // Write to the map and await success
                res.result().put("deployments", Arrays.asList(context.deploymentID()),
res1 -> {
                    // Deploy AnotherVerticle 10 times
                        IntStream.rangeClosed(1, 10).forEach(i -> {
                        vertx.deployVerticle("java:ClusteredVerticle.java");
                    });
                });
            }
        });

        vertx.setPeriodic(100, this::showDeployedVerticles);
    }

    void showDeployedVerticles(Long t) {
        // Print the list of deployment IDs stored in the shared data local Map
        LoggerFactory.getLogger("Exercise9_2").info("Polling shared data map");

        // Get reference to clusterWide map called "shared"
        vertx.sharedData().getClusterWideMap("shared", res -> {
            if (res.succeeded()) {
                // Get the "deployments" value from the AsyncMap
                res.result().get("deployments", res1 -> {

                    // Iterate over list of values
                    ((List<String>)res1.result()).stream().forEach(it -> {
```

```
                        System.out.println(it);
                    });
                    System.out.println();
                    System.out.println();
                });
            }
        });
    }
}
```

Exercise9/ClusteredVerticle.java

```
import io.vertx.core.logging.LoggerFactory;
import io.vertx.core.AbstractVerticle;

import java.util.List;

public class ClusteredVerticle extends AbstractVerticle {

    @Override
    public void start() throws Exception {
        // Get a reference to clusterWide map called "shared"
        vertx.sharedData().getClusterWideMap("shared", res -> {
            if (res.succeeded()) {
                // Get the "deployments" list
                res.result().get("deployments", res1 -> {
                    List<String> deploymentList = (List<String>)res1.result();
                    deploymentList.add(context.deploymentID());

                    // Update the "deployments" list
                    res.result().put("deployments", deploymentList, res2 -> {
                        LoggerFactory.getLogger("ClusteredVerticle").info("Deployed
ClusteredVerticle: ${context.deploymentID()}");
                    });
                });
            }
        });
    }
}
```

This cluster-wide data coordination is complex, so it is always advisable to send shared data via the EventBus where possible. The ClusterManager and the AsyncMap implementations ensure that access to and writing of clustered resources are synchronized properly across the entire cluster and thust prevents race conditions. The negative impact being that access to read/write clustered data is much slower.

Exercise 10 - A TCP Echo Server

As a quick introduction to the network server capabilities of Vert.x, Let's implement a TCP Echo

Server. An echo server is a network socket server which accepts incoming data and sends the same data back as a response.

Exercise10/Exercise10.java

```
import io.vertx.core.buffer.Buffer;
import io.vertx.core.json.JsonObject;
import io.vertx.core.logging.LoggerFactory;
import io.vertx.core.net.NetServer;
import io.vertx.core.net.NetServerOptions;
import io.vertx.core.net.NetSocket;
import io.vertx.core.AbstractVerticle;

public class Exercise10 extends AbstractVerticle {

    public void start() {
        JsonObject opts = new JsonObject()
                .put("host", "0.0.0.0")
                .put("port", 1080)
                .put("logActivity", true);

        NetServer server = vertx.createNetServer(new NetServerOptions(opts));
        server.connectHandler(this::connectHandler).listen();
    }

    void connectHandler(NetSocket socket) {
        socket.handler(b -> this.dataHandler(socket, b));
    }

    void dataHandler(NetSocket socket, Buffer b) {
        LoggerFactory.getLogger("Exercise10").info(b.toString());
        socket.write(b);
    }
}
```

There are some new things to learn in this example. For one, there is the introduction of NetServer and it's associated options; but that it mostly self-explanitory. The other thing to make note of is the use of the Buffer object. From the Vert.x API documentation:

> Most data is shuffled around inside Vert.x using buffers. A buffer is a sequence of zero or more bytes that can read from or written to and which expands automatically as necessary to accommodate any bytes written to it. You can perhaps think of a buffer as smart byte array.

You can think of *Buffer*s as a way of pushing around streams of bytes. Buffers also have some convenience methods like `toString()`, `toJsonObject()`, and `toJsonArray()`. You can append to a Buffer using one of the provided append methods which can handle input types like Int/Float/Short/Unsigned/String/Byte/Long/Double. There are also append methods for storing data in the buffer in little-endian byte order.

Next Steps:

- Modify the EchoServer above to take in some text (Latin characters, numbers, spaces, newlines ONLY), ignore non-text, and send back `Hello <text>`.

Exercise 11

Vert.x also has the ability to create UDP servers. Let's see what a UDP echo server would look like in Vert.x:

Exercise11/Exercise11.java

```
import io.vertx.core.AsyncResult;
import io.vertx.core.Handler;
import io.vertx.core.logging.Logger;
import io.vertx.core.logging.LoggerFactory;
import io.vertx.core.datagram.DatagramPacket;
import io.vertx.core.datagram.DatagramSocket;
import io.vertx.core.AbstractVerticle;

public class Exercise11 extends AbstractVerticle {
    private final Logger LOG = LoggerFactory.getLogger("Exercise11");

    @Override
    public void start() throws Exception {
        DatagramSocket socket = vertx.createDatagramSocket();

        socket.listen(1080, "0.0.0.0", this::socketHandler);
    }

    void socketHandler(AsyncResult<DatagramSocket> res) {
        if (res.succeeded()) {
            // Successfully received a datagram
            DatagramSocket socket = res.result();
            socket.handler(p -> this.datagramHandler(socket, p));
        }
    }

    void datagramHandler(DatagramSocket socket, DatagramPacket p) {
        socket.send(p.data(), p.sender().port(), p.sender().host(),
this::sendHandler);
    }

    void sendHandler(AsyncResult<DatagramSocket> sent) {
        if (sent.succeeded()) {
            LOG.info("SUCCESS");
        } else {
            LOG.error("FAILED");
        }
    }
}
```

Next Steps:

- Modify the EchoServer above to take in some text (Latin characters, numbers, spaces, newlines ONLY), ignore non-text, and send back `Hello <text>`.

Exercise 12

HTTP is a mainstay of software these days, so being able to make and handle HTTP requests is vital.

Let's see how Vert.x make HTTP requests in an asynchronous manner:

Exercise12/Exercise12.java

```
import io.vertx.core.http.HttpClient;
import io.vertx.core.logging.Logger;
import io.vertx.core.logging.LoggerFactory;
import io.vertx.core.http.HttpClientResponse;
import io.vertx.core.AbstractVerticle;

public class Exercise12 extends AbstractVerticle {

    private static final Logger LOG = LoggerFactory.getLogger("Exercise12");

    @Override
    public void start() throws Exception {
        vertx.createHttpClient()
                            .getNow("www.google.com", "/", this::responseHandler);
    }

    void responseHandler(HttpClientResponse response) {
        if (response.statusCode()==200 && response.statusMessage()=="OK") {
            LOG.info("Success!");
        } else {
            LOG.warn("Got "+response.statusCode()+" as the response code.");
        }
        vertx.close();
    }
}
```

Next Steps

- Make an HTTP GET request which uses HTTP Basic Authentication
- Make an HTTP POST request which sends a JSON body

Exercise 13

We've covered a number of individual features of Vert.x, Async, and non-blocking APIs in Vert.x, but in this exercise we will try to put a few different ones together. Here's the scenario:

- An HTTP server listening on port 8080
- A web browser will make a request to the '/merged/' endpoint
- The Vert.x application will execute several operations in parallel
- Request the `www.google.com` index page
- Read a file (Your choice, but make it a simple short file) from the filesystem
- Perform a DNS lookup on `www.google.com`
- Once all of the parallel operations are complete, insert the file contents and the dns results into

a `<pre>` block before the ending `</body>` tag in the html retrieved from Google (Note: Groovy Regex Replace)

- Return the modified Google index page to the browser
- If ANY one of the async operations fails, return a 500 HTTP response with the exception's `localizedMessage` value.

There is only one component here which you are not already familiar with, and that is the DNSClient. The DNS client relatively simple, and it will be left up to you to read the documentation and use it.

Next Steps:

- Modify the solution for Exercise 13 so that you can pass in a `config.json` file from which the application will read the settings for:
  - HTTP client host
  - HTTP client URI
  - HTTP client port
  - HTTP client SSL enable/disable
  - DNS hostname to be resolved
  - Filename to be read

Exercise 14

Let's jump back into `vertx-web` again a little deeper... One interesting aspect of the `Router` and `RoutingContext` is that routes can be **chained**. What this means is that if you have a path which starts with `/rest`, and all routes under that path will all do some of the same tasks, you can extract those operations into an earlier route which then calls `RoutingContext.next()` and the request will be processed by other routes which might match. Here's an example:

Exercise14/Exercise14.java

```
import io.vertx.core.json.JsonObject;
import io.vertx.ext.web.Router;
import io.vertx.ext.web.RoutingContext;
import io.vertx.core.AbstractVerticle;

import static io.netty.handler.codec.http.HttpResponseStatus.OK;
import static io.netty.handler.codec.http.HttpResponseStatus.UNAUTHORIZED;

public class Exercise14 extends AbstractVerticle {

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

        router.route().handler(this::authHandler);
```

```
        router.route("/rest/*").handler(this::restHandler);
        router.get("/rest/customer/:id").handler(this::customerByIdHandler);

        vertx.createHttpServer().requestHandler(router::accept).listen(8080,
"0.0.0.0");
    }

    void authHandler(RoutingContext ctx) {
        // Do something to validate authentication
        ctx.put("authenticated", true);
        ctx.next();
    }

    void restHandler(RoutingContext ctx) {
        // All REST requests will have certain common requirements

        ctx.response()
            .putHeader("Content-Type", "application/json") // Set the response
Content-Type to application/json
            .putHeader("Cache-Control", "nocache") // Disable caching for browsers
which respect this header
            .putHeader("Expires", "Tue, 15 Nov 1994 12:45:26 GMT"); // Set some expiry
date in the past to help prevent caching
        ctx.next();
    }

    void customerByIdHandler(RoutingContext ctx) {
        if (ctx.get("authenticated")) {
            // The "authenticated" value is set in the authHandler method/route, and
so it should be present here!!
            ctx.response().setStatusCode(OK.code())
                          .setStatusMessage(OK.reasonPhrase())
                          .end(new JsonObject().put("authenticated",
true).encodePrettily());  // The headers set in restHandler are already set as well!
        } else {
            ctx.response().setStatusCode(UNAUTHORIZED.code())
                          .setStatusMessage(UNAUTHORIZED.reasonPhrase())
                          .end("{}");  // The headers set in restHandler are already
set as well!
        }
    }
}
```

In this, admittedly contrived, example; we see that any request which matches '/rest/customer/:id' will match all of the previous routes as well. Since the handlers for each of those routes are calling `RoutingContext.next()` on the RoutingContext object, ALL of these handlers will be applied in order!

Next Steps: * Do some research in the Vert.x documentation, and determine how to add a **catch-all** route which will handle any previously unhandled requests by sending a custom JSON 404 response * Create a **catch-all** route which will instead serve up static filesystem resources

Exercise 15

Now that we have a basic understanding of event based programming in Vert.x, let's learn how we can test our code when operating in an asynchronous world. In this example we will have a single Verticle which sets itself up to listen on the event bus. And using Spock Framework we will add a behavior test in order to make sure that our Verticle functions as expected. In order to facilitate running the tests, we have a Maven project POM to describe the build environment and handle dependencies. The Maven POM is already configured with the required dependencies for running tests using Spock Framework and the required Groovy libraries to make it all work.

Exercise15/src/main/java/com/redhat/labs/vertx/exercise15/Main.java

```
package com.redhat.labs.vertx.exercise15;

import io.vertx.core.AbstractVerticle;
import io.vertx.core.Future;
import io.vertx.core.json.JsonObject;

public class Main extends AbstractVerticle {

    @Override
    public void start(Future<Void> startFuture) throws Exception {
        vertx.eventBus().consumer("test.address", msg -> {
            msg.reply(new JsonObject().put("ok", true).put("message", "SUCCESS"));
        });

        startFuture.complete();
    }
}
```

Exercise15/src/test/java/com/redhat/labs/vertx/exercise15/MainSpec.groovy

```
Unresolved directive in README.asciidoc -
include::Exercise15/src/test/java/com/redhat/labs/vertx/exercise15/MainSpec.groovy[]
```

① Spock uses Gherkin style syntax in the form of **given**, **when**, **then**

② When testing asynchronous code, Spock gives you an **AsyncConditions** class which can be used to coordinate. The number passed indicates the number of async evaluation blocks which will need to be processed

③ This is an async evaluation block. In this case it checks to ensure that the Verticle is successfully deployed. Any assertions in this block must succeed or the test will fail

④ The **when** block is where we place the code to be tested

⑤ Another async evaluation block, this time checking to ensure that the reply message is correct

⑥ The **then** block is where we tell the system to check our work

⑦ The `async.await(10)` call tells Spock to wait for 10 seconds for the 2 async conditions to complete, and if it times out the test fails.