



2. Verticles: the basic processing units of Vert.x

 Put simply, a verticle is the fundamental processing unit in Vert.x.

 The role of a verticle is to encapsulate a technical functional unit for processing events,

Much like components in other technologies like Enterprise Java Beans, Spring ,
verticles can be deployed, and they have a life cycle.

Asynchronous programming is key to building reactive applications,
since they have to scale, and verticles are fundamental in Vert.x for structuring
(asynchronous) event processing code and business logic.

Writing a verticle

verticles have private state that may be updated when receiving events, they
can deploy other verticles, and they can communicate via message-passing

▼ Ex-1 : A Sample verticle

verticle that processes two types of events: periodic timers and HTTP requests

```
import io.vertx.core.AbstractVerticle;
import io.vertx.core.Vertx;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;

class HttpServerVerticle extends AbstractVerticle {

    private final Logger logger = LoggerFactory.getLogger(HttpServerVerticle.class);
    private long counter = 1;

    @Override
    public void start() {
        vertx.setPeriodic(5000, id -> {
            logger.info("tick");
        });

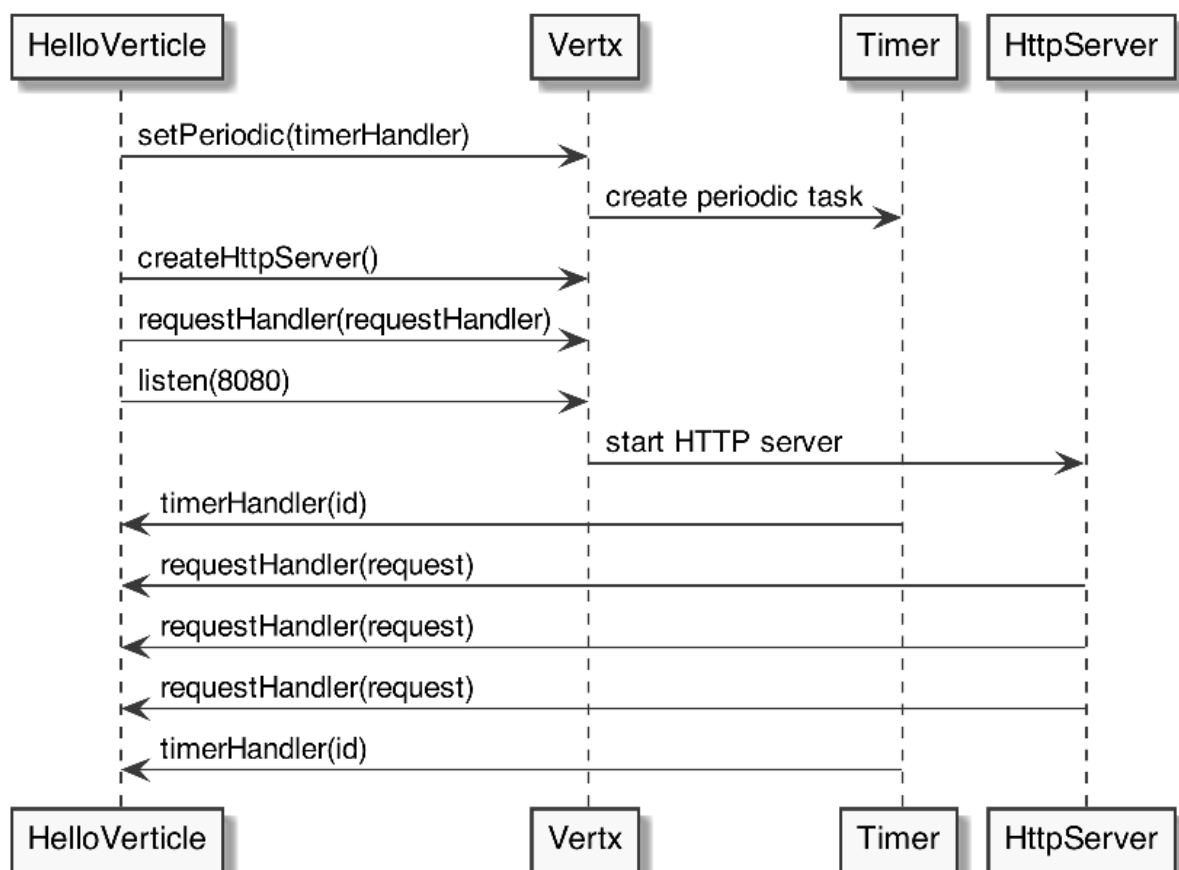
        vertx.createHttpServer()
            .requestHandler(req -> {
                logger.info("Request #{} from {}", counter++,
                    req.remoteAddress().host());
                req.response().end("Hello!");
            })
            .listen(8080);
        logger.info("Open http://localhost:8080/");
    }
}

public class Main{
    public static void main(String[] args) {
        Vertx vertx = Vertx.vertx();
        vertx.deployVerticle(new HttpServerVerticle());
    }
}
```

- The `start` method typically contains setup and initialization for handlers
- The `stop` method is implemented when housekeeping tasks are required, such as closing open database connections.

👁️ Running and first observations

- 👉 event processing happens on a single event-loop thread.
- 👉 both the periodic tasks and HTTP request processing happen on a thread that appears as `vert.x-eventloop-thread-0` in the logs.
- 👉 a verticle instance always executes event processing on the same thread so there is no need for using thread synchronization primitives



More on verticles

There are more things to know about writing and deploying verticles:

- What happens when the event loop is being blocked?
 - How can you defer notification of life-cycle completion in the presence of asynchronous initialization work?
 - How can you deploy and undeploy verticles?
 - How can you pass configuration data?
-

Blocking and the event loop

- 👉 On Event, Handler callbacks are run from event-loop threads.
- 👉 It is important that code running on an event loop spends as little time as possible,
so that the event-loop thread can have a higher throughput
in the number of processed events.
- 👉 This is why no long-running or blocking I/O operations should happen on the event loop.

▼ Ex-2 : An example where the event loop is being blocked

```
class BlockEventLoop extends AbstractVerticle {
    @Override
    public void start() {
        vertx.setTimer(1000, id -> {
            while (true);
        });
    }
}

public class Main{
    public static void main(String[] args) {
        Vertx vertx = Vertx.vertx();
    }
}
```

```
    vertx.deployVerticle(new BlockEventLoop());  
  }  
}
```

The time limit before the blocked thread checker complains is 2 seconds by default,
but it can be configured to a different value

You can use system properties to change the settings:

- - Dvertx.options.blockedThreadCheckInterval=5000 changes the interval to 5 seconds.
- - Dvertx.threadChecks=false disables the thread checker.



Never block even-loop-thread,

Asynchronous notification of life-cycle events

▼ Ex-3 : Example of an asynchronous start life-cycle method

```
import io.vertx.core.AbstractVerticle;  
import io.vertx.core.Future;  
import io.vertx.core.Promise;  
import io.vertx.core.Vertx;  
  
class HttpServerVerticle extends AbstractVerticle {  
  @Override
```

```

public void start(Promise<Void> promise) { // <1>
    vertx.createHttpServer()
        .requestHandler(req -> req.response().end("Ok"))
        .listen(8080, ar -> {
            if (ar.succeeded()) { // <2>
                promise.complete(); // <3>
            } else {
                promise.fail(ar.cause()); // <4>
            }
        });
}

public class Main{
    public static void main(String[] args) {
        Vertx vertx = Vertx.vertx();
        vertx.deployVerticle(new HttpServerVerticle());
    }
}

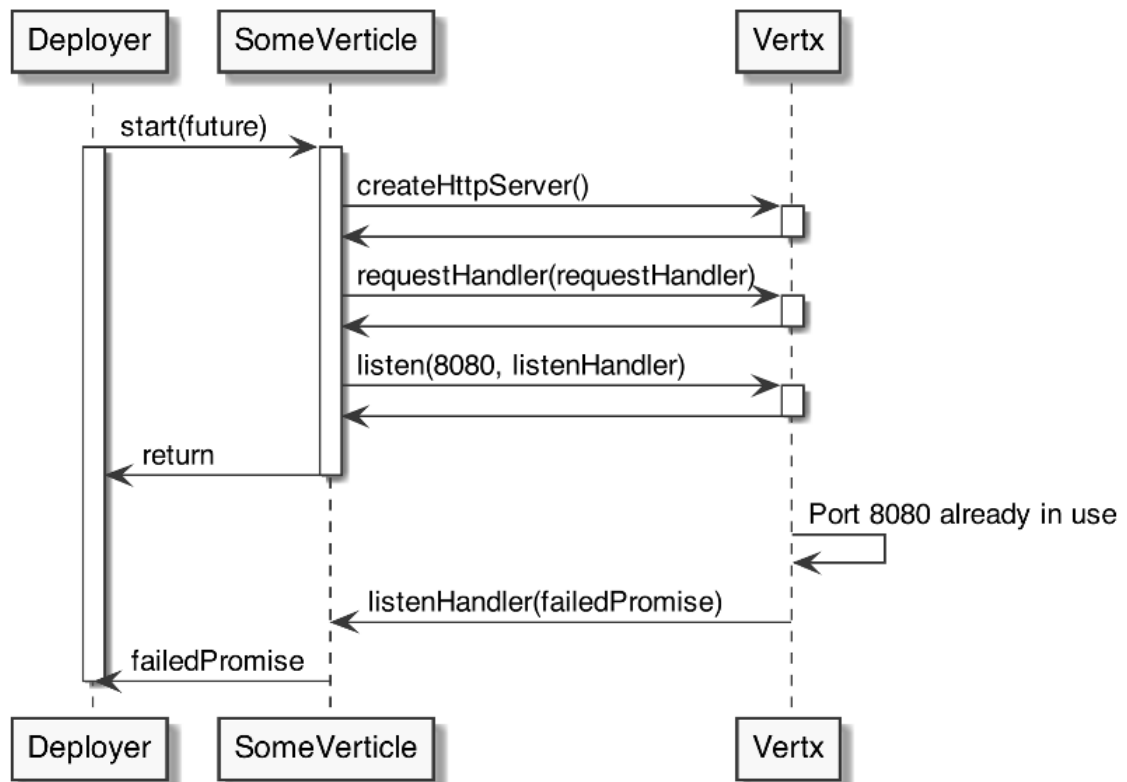
```

The start and stop methods in AbstractVerticle support variants with an argument of type `io.vertx.core.Promise`.

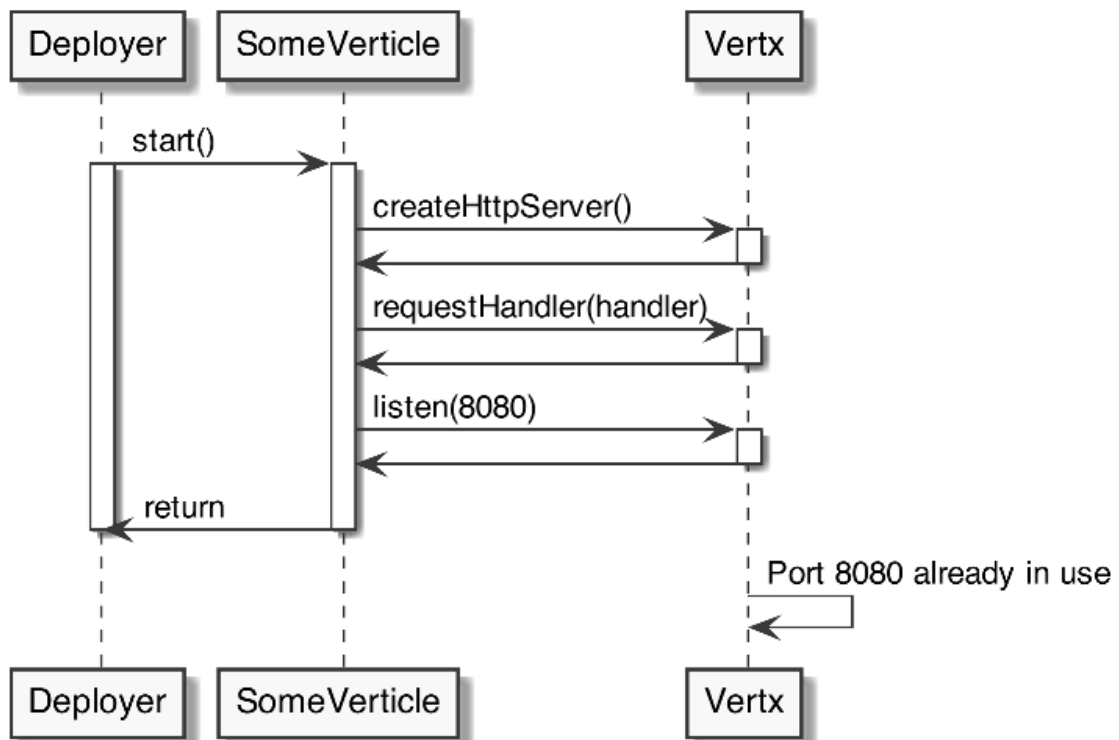
As the name suggests, a Vert.x Promise is an adaptation of the futures and promises model for processing asynchronous results

A **promise** is used to write an asynchronous result, whereas a **future** is used to view an asynchronous result.

Sequence diagram of starting an HTTP server with a promise and a listen handler



Sequence diagram of starting an HTTP server without a promise and a listen handler



Deploying vertices

Vertices are always deployed (and undeployed) through the Vertx object

You can do so from any method, but the typical way to deploy an application composed of vertices is as follows:

1. Deploy a *main* verticle.
2. The *main* verticle deploys other verticles.
3. The deployed verticles may in turn deploy further verticles.

▼ Ex-4

```
public class EmptyVerticle extends AbstractVerticle {
    private final Logger logger = LoggerFactory.getLogger(EmptyVerticle.class);

    @Override
    public void start() {
        logger.info("Start");
    }

    @Override
    public void stop() {
        logger.info("Stop");
    }
}
```

```
public class Deployer extends AbstractVerticle {

    private final Logger logger = LoggerFactory.getLogger(Deployer.class);

    @Override
    public void start() {
        long delay = 1000;
        for (int i = 0; i < 50; i++) {
            vertx.setTimer(delay, id -> deploy());
            delay = delay + 1000;
        }
    }

    private void deploy() {
        vertx.deployVerticle(new EmptyVerticle(), ar -> {
            if (ar.succeeded()) {
                String id = ar.result();
                logger.info("Successfully deployed {}", id);
                vertx.setTimer(5000, tid -> undeployLater(id));
            } else {
                logger.error("Error while deploying", ar.cause());
            }
        });
    }

    private void undeployLater(String id) {
        vertx.undeploy(id, ar -> {
            if (ar.succeeded()) {
                logger.info("{} was undeployed", id);
            } else {
                logger.error("{} could not be undeployed", id);
            }
        });
    }
}
```

```
}  
}
```

```
public class Main{  
    public static void main(String[] args) {  
        Vertx vertx = Vertx.vertx();  
        vertx.deployVerticle(new Deployer());  
    }  
}
```

☞ By default, Vert.x creates twice the number of event-loop threads as CPU cores.

☞ If you have 8 cores, then a Vert.x application has 16 event loops.

☞ The assignment of verticles to event loops is done in a round-robin fashion.

☞ This teaches us an interesting lesson:
while a verticle always uses the same event-loop thread, the event-loop threads are being shared by multiple verticles,
This design results in a predictable number of threads for running an application.

Passing configuration data

Configuration needs to be passed as JSON data, using the Vert.x JSON API

▼ Ex-5 : Passing configuration data to a verticle

```
public class SampleVerticle extends AbstractVerticle {
    private final Logger logger = LoggerFactory.getLogger(SampleVerticle.class);

    @Override
    public void start() {
        logger.info("n = {}", config().getInteger("n", -1));
    }
}

public class Main{

    public static void main(String[] args) {
        Vertx vertx = Vertx.vertx();
        for (int n = 0; n < 4; n++) {
            JsonObject conf = new JsonObject().put("n", n);
            DeploymentOptions opts = new DeploymentOptions()
                .setConfig(conf);
            vertx.deployVerticle("SampleVerticle", opts);
        }
    }
}
```

When code needs to block



The basic rule when running code on an event loop is that it should not block, and it should run “fast enough.”

Vert.x provides two options for dealing with such cases:

- worker verticles
- `executeBlocking` operation.

Worker verticles

Worker verticles are a special form of verticles that do not execute on an event loop.

Instead, they execute on worker threads, that is, threads taken from special worker pools.

A worker verticle processes events just like an event-loop verticle would do, except that it can take an arbitrarily long time to do so.

It is important to understand two things

- A worker verticle is not tied to a single worker thread
- Worker verticles may only be accessed by a single worker thread at a given time.

To put it simply, like event-loop verticles, worker verticles are single-threaded, but unlike event-loop verticles, the thread may not always be the same.

▼ Ex-6 : A sample worker verticle

```

class WorkerVerticle extends AbstractVerticle {
    private final Logger logger = LoggerFactory.getLogger(WorkerVerticle.class);

    @Override
    public void start() {
        vertx.setPeriodic(10_000, id -> {
            try {
                logger.info("Zzz...");
                Thread.sleep(8000);
                logger.info("Up!");
            } catch (InterruptedException e) {
                logger.error("Woops", e);
            }
        });
    }
}

public class Main{

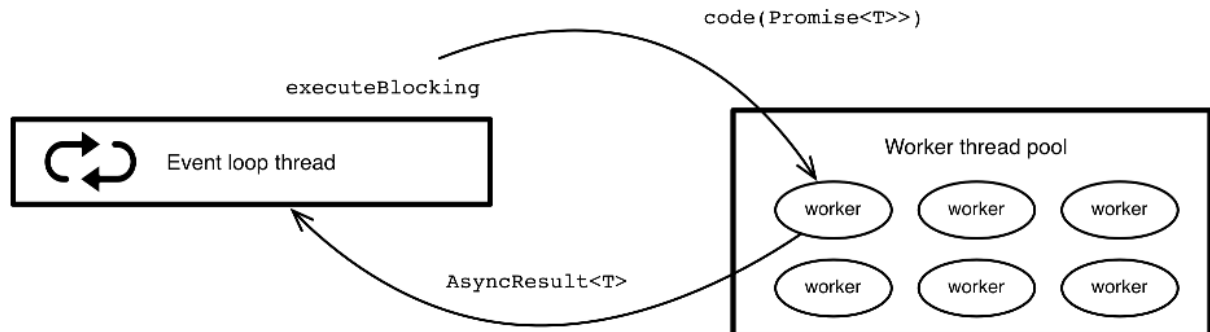
    public static void main(String[] args) {
        Vertx vertx = Vertx.vertx();
        DeploymentOptions opts = new DeploymentOptions()
            .setInstances(1)
            .setWorker(true);
        vertx.deployVerticle("chapter2.worker.WorkerVerticle", opts);
    }
}

```

- ❶ We can block and get no warning!
- ❷ Making a worker verticle is a deployment options flag.

The executeBlocking operation

This method takes some blocking code to execute, offloads it to a worker thread, and sends the result back to the event loop as a new event,



▼ Ex-7 : Using executeBlocking

```

public class Offload extends AbstractVerticle {

    private final Logger logger = LoggerFactory.getLogger(Offload.class);

    @Override
    public void start() {
        vertx.setPeriodic(5000, id -> {
            logger.info("Tick");
            vertx.executeBlocking(this::blockingCode, this::resultHandler);
        });
    }

    private void blockingCode(Promise<String> promise) {
        logger.info("Blocking code running");
        try {
            Thread.sleep(4000);
            logger.info("Done!");
            promise.complete("Ok!");
        } catch (InterruptedException e) {
            promise.fail(e);
        }
    }

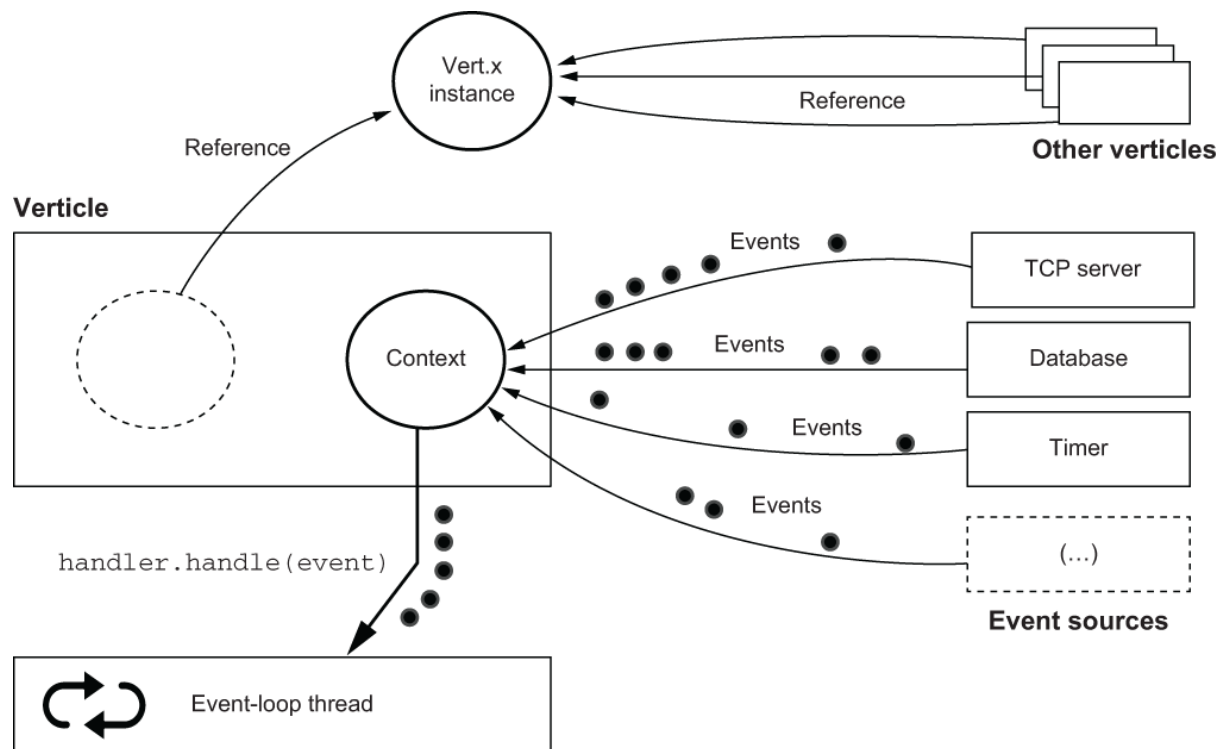
    private void resultHandler(AsyncResult<String> ar) {
        if (ar.succeeded()) {
            logger.info("Blocking code result: {}", ar.result());
        } else {
            logger.error("Woops", ar.cause());
        }
    }
}
  
```

- ❶ `executeBlocking` takes two parameters: the code to run and a callback for when it has run.
 - ❷ The blocking code takes a Promise object of any type. It is used to eventually pass the result.
 - ❸ The Promise object needs to either complete or fail, marking the end of the blocking code execution.
 - ❹ Processing the result on the event loop is just another asynchronous result.
-

So what is really in a verticle?

Verticles and their environment

A. An event-loop verticle and its environment



A verticle object is essentially the combination of two objects:

- The Vert.x instance the verticle belongs to.
- A dedicated context instance that allows events to be dispatched to handlers.



The Vert.x instance exposes the core APIs for declaring event handlers.



The Vert.x instance is being shared by multiple verticles, and there is generally only one instance of Vert.x per JVM process.



The context instance holds the access to the thread for executing handlers

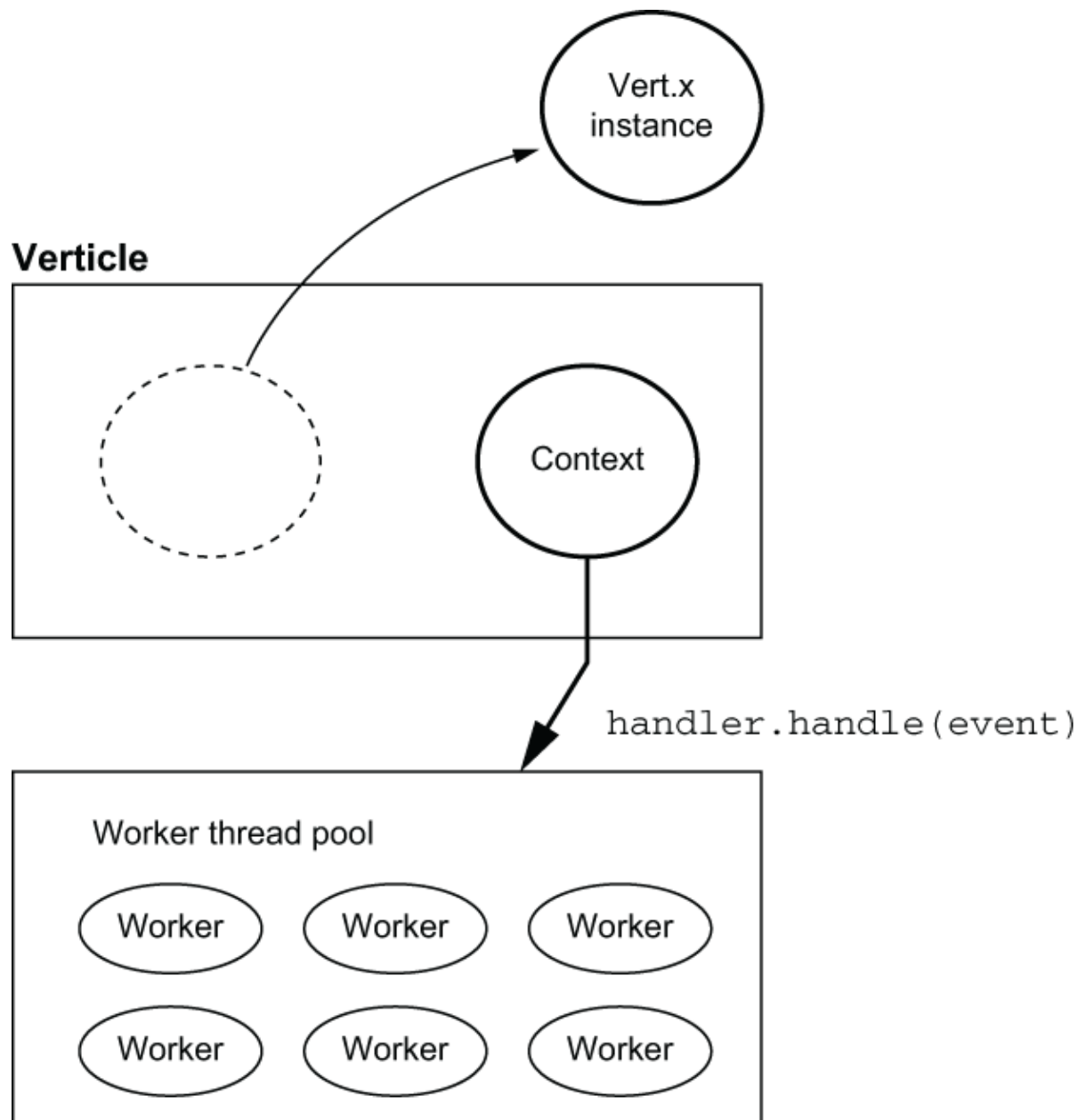


Events may originate from various sources such as timers, database drivers, HTTP servers, and more



Event handling in user-defined callbacks happens through the context. The context instance allows us to call the handler back on the verticle event-loop thread, hence respecting the Vert.x threading model.

B. A worker verticle and its environment



Summary

- Verticles are the core component for asynchronous event processing in Vert.x applications.
- Event-loop verticles process asynchronous I/O events and should be free of blocking and long-running operations.

- Worker verticles can be used to process blocking I/O and long-running operations.
 - It is possible to mix code with both Vert.x and non-Vert.x threads by using event-loop contexts.
-