Vert.x IN ACTION

Julien Ponge

MEAP

MANNING

MEAP Edition
Manning Early Access Program

Vert.x in Action

Version 1

welcome

Thank you for purchasing the MEAP of *Vert.x in Action*. Asynchronous and reactive applications are an important topic in modern distributed systems, especially as the progressive shift to virtualized and containerized runtime environments emphasize the need for resource efficient, adaptable and dependable application designs.

Asynchronous programming is key to maximizing hardware resource usage, as it allows dealing with more concurrent connections than with the traditional blocking I/O paradigms. Services need to cater for workloads that may drastically change throughout from one hour to the other, hence we need to design code that naturally supports horizontal scalability. Last but not least, failure is inevitable when we have services interacting with other services over the network. Embracing failure is key for designing dependable systems.

Assemble asynchronous programming, horizontal scalability, resilience and you have what we call today *reactive applications*, which can also be summarized without marketing jargon as *"scalable and dependable applications"*.

That being said there is no free lunch and the transition to writing asynchronous and reactive applications is difficult when you have a background in more traditional software stacks. Grokking asynchrony in itself is difficult, but the implications of scalability and resilience on the design of an application are anything but trivial.

This book is aimed at Java developers from all backgrounds who would like to teach themselves both the concepts and practices of building asynchronous and reactive applications. The book uses *Eclipse Vert.x*, a "no-magic" tookit for writing such applications. Developers appreciate Vert.x for its simplicity, embeddability and field-tested performance.

Still, an important consideration while writing this book is that it teaches important concepts that are anything specific to Vert.x, so you can transfer knowledge to other software stacks tomorrow or in 5 years. The book is divided in 3 parts: asynchronous programming, reactive applications, and deployment options.

I hope that you will appreciate this book. Your feedback is essential to making this book a useful resource for our peers, and I look forward to hearing from you at Manning's Author Online forum for my book.

—Dr. Julien Ponge

brief contents

1 Vertx, asynchronous programming and reactive systems

This chapter covers:

- What is Vert.x
- Why distributed systems cannot be avoided
- What are the challenges in programming resource-efficient networked applications
- What is asynchronous and non-blocking programming
- What is a reactive application and why asynchronous programming is not enough
- What are the alternatives to Vert.x

We developers live in a industry of buzzwords, technologies and practices hype cycles. I have long taught to university students elements for designing, programming, integrating and deploying applications, and I have witnessed first-hand how complicated it can be for newcomers to navigate in the wild ocean of current technologies.

Asynchronous and *reactive* are important topics in modern applications and my goal with this book is to help a large audience of developers understand the core concepts behind these terms, gain practical experience, and recognize *when* there are benefits in these approaches. We will use *Eclipse Vert.x*, a toolkit for writing asynchronous applications that has the added benefit of providing solutions for the different definitions of what *"reactive"* means.

Ensuring that readers understand the concepts is a fundamental priority for me with this book. While I want to give you a solid understanding of writing Vert.x applications, I also want to make sure that you can translate skills to other similar and possibly competing technologies, now or 5 years down the road.

1.1 Being distributed and networked is the norm

It was common 20+ years ago to deploy business applications that could perform all operations while running isolated on a single machine. Such applications typically exhibited a graphical user interface, and they had local databases or custom file management for storing data. This is of course a gross exaggeration as networking was already there, so business applications could already take advantage of database servers over the network, networked file storage, and various remote code operations.

Today, an application is more naturally exposed to end-users using web and mobile interfaces. This naturally brings the network into play, hence distributed systems. Also, *service-oriented architectures* allow re-using some functionality by issuing requests to other services, possibly controlled by a third-party provider. Examples would be delegating authentication in a consumer application to popular account providers like *Google*, *Facebook* or *Twitter*, or delegating payment processing to *Stripes* or *PayPal*.

1.2 Not living on an isolated island

Figure 1.1. A networked application / service

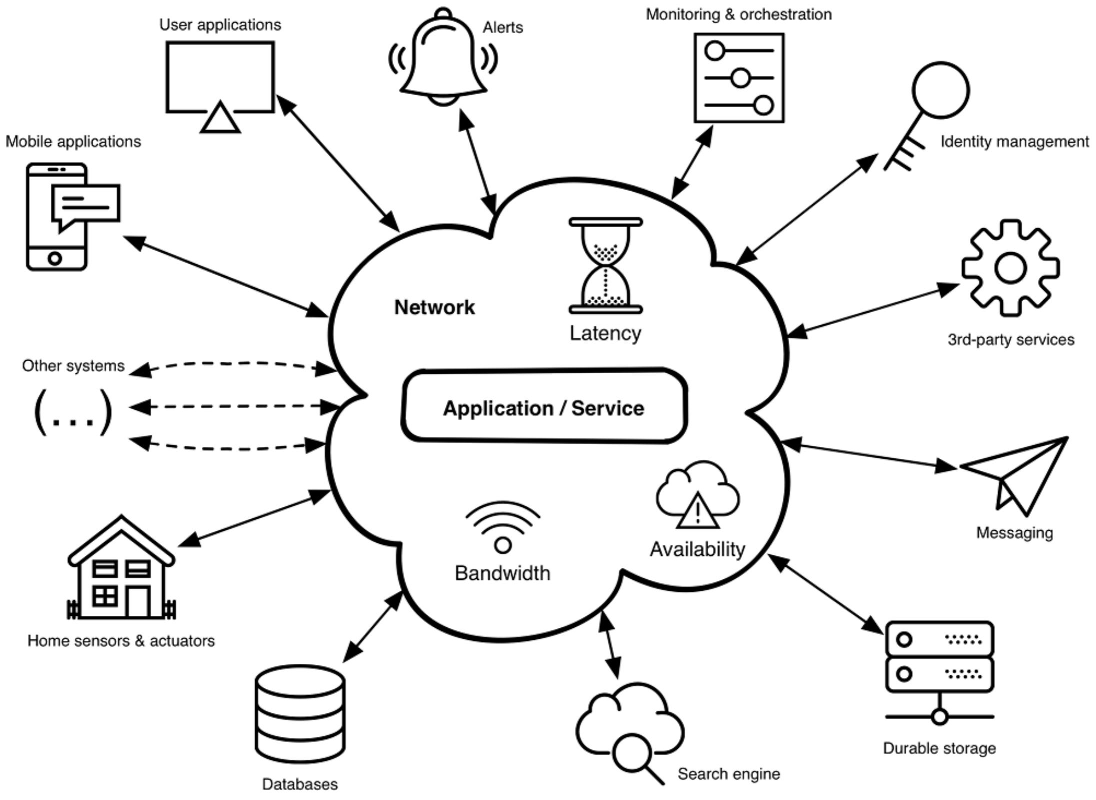


Figure 1.1 is a fictional depiction of what a modern application is: a set of networked services interacting with each other. Here are some of these networked services:

- a database like *PostgresSQL* or *MongoDB* stores data,
- a search engine like *Elastic Search* allows finding information that was previously indexed such as products in a catalog,
- a durable storage service like *Amazon S3* provides persistent and replicated data storage of documents,
- a messaging service can be:
  - a *SMTP* server to programmatically send emails,
  - a bot for interacting with users over messaging platforms such as *Slack*, *Telegram* or *Facebook Messenger*,
  - an integration messaging protocol for application-to-application integration like AMQP,
- an identity management service like *Keycloak* provides authentication and role management for user and service interactions,
- monitoring with libraries like *Micrometer* expose health statuses, metrics and logs so that external orchestration tools can maintain proper quality of service, possibly by starting new service instances or killing existing ones when they fail.

We will see later in this book examples of typical services such as API endpoints, stream processors and edge services.[1] The above list is not exhaustive of course, but the key point is that services rarely live in isolation as they need to talk to other services over the network to function.

1.3 There is no free lunch on the network

The network is exactly where a number of things may go wrong in computing.

1. The bandwidth can fluctuate a lot, so data-intensive interactions between services may suffer. Not all services can enjoy fast bandwidth inside the same data-center and even so, it remains slower than communications between processes on the same machine.
2. The latency also fluctuates a lot, and because services need to talk to services that talk to services to process a given request, all network-induced latency add up to the overall request processing times.
3. Availability is not always for granted: networks fail. Routers fail. Proxies fail. Sometimes someone runs into a network cable and disconnects it. When the network fails a service that sends a request to another service may not be able to know if it is the other service or the network that is down.

In essence modern applications are made of distributed and networked services. They are accessed over networks that introduce some problems on their own, and each service needs to maintain several incoming and outgoing connections.

[1] For readers already familiar with micro-service patterns, "Edge service" is in my opinion a better term to "API gateway".

1.4 The simplicity of blocking APIs

Services need to manage connections to other services and requesters. The traditional and widespread model for managing concurrent network connections is to allocate a thread for each connection. This is the model in many technologies such as *Servlets* in *Jakarta EE* (before additions in version 3), *Spring Framework* (before additions in version 5), *Ruby on Rails*, *Python Flask*and many more.

This model has the advantage of simplicity as it is *synchronous*. Let us take an example in listing 1.3 where a TCP server echoes input text back to the client until it sees a `/quit` terminal input.

The server can be run using the Gradle `run` task from the full example project (`gradle run` in a terminal). By using the `netcat` command-line tool, we can send and receive text:

Listing 1.1. Client-side output of a netcat session.

```
$ netcat localhost 3000
Hello, Vert.x!   ❶
Hello, Vert.x!   ❷
Great
Great
/quit
/quit
$
```

❶ This line is the user input on the command line.

❷ This line is sent by the TCP server.

On the server side, we can see the following trace:

Listing 1.2. Server-side trace

```
$ ./gradlew run
(...)
~ Hello, Vert.x!
~ Great
~ /quit
```

The code in listing 1.3 provides the TCP server implementation. It is a very classical usage of the `java.io` package that provides synchronous I/O APIs.

Listing 1.3. Synchronous "echo" TCP protocol

```
public class SynchronousEcho {
  public static void main(String[] args) throws Throwable {
    ServerSocket server = new ServerSocket();
    server.bind(new InetSocketAddress(3000));
    while (true) {                                   ❶
      Socket socket = server.accept();
      new Thread(clientHandler(socket)).start();
    }
  }

  private static Runnable clientHandler(Socket socket) {
    return () -> {
      try (
        BufferedReader reader = new BufferedReader(
          new InputStreamReader(socket.getInputStream()));
        PrintWriter writer = new PrintWriter(
          new OutputStreamWriter(socket.getOutputStream()))) {
        String line = "";
        while (!"/quit".equals(line)) {
          line = reader.readLine();       ❷
          System.out.println("~ " + line);
          writer.write(line + "\n");      ❸
          writer.flush();
        }
      } catch (IOException e) {
        e.printStackTrace();
      }
    };
  }
}
```

❶ The main application thread plays the role of an *accepting* thread as it receives socket objects for all new connections. The operation blocks when no connection is pending. A new thread is being allocated for each connection.

❷ Reading from a socket may block the thread allocated to the connection, for example when there is not enough data being read.

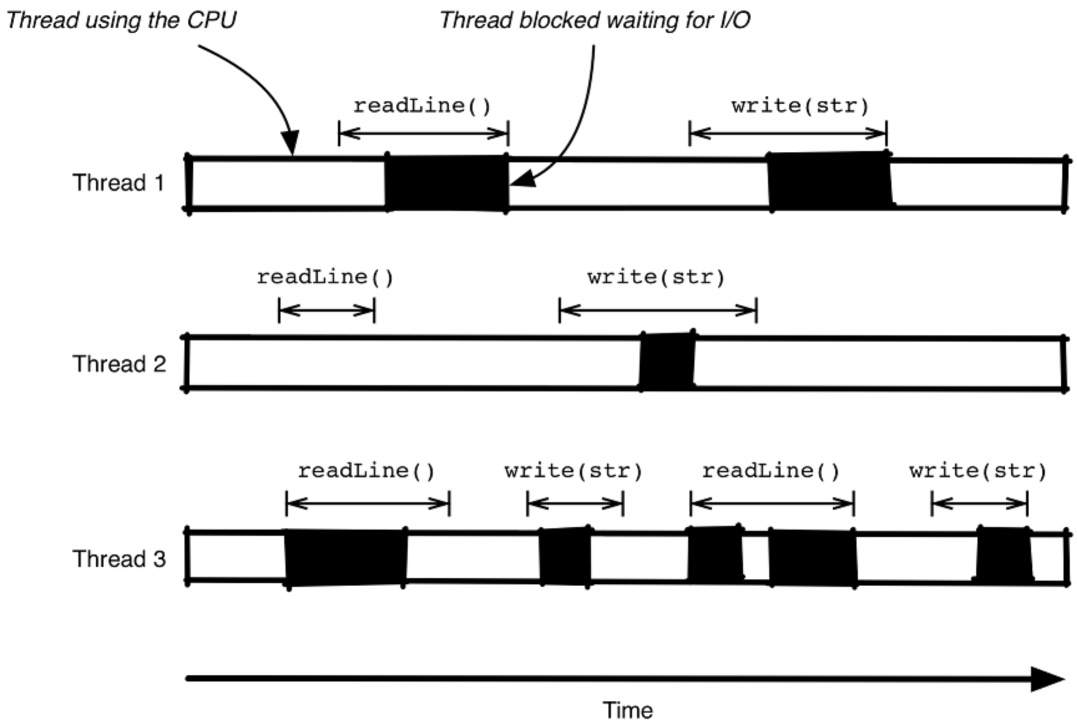
❸ Writing to a socket may also block, for example until the underlying TCP buffer data has been sent over the network.

The server uses the main thread for accepting connections, and each connection is being allocated a new thread for processing I/O. The I/O operations are synchronous, so threads may block on I/O operations.

1.5 Blocking APIs waste resources, increase costs

The main problem with the code in listing 1.3 is that it allocates a new thread for each incoming connection, and threads are anything but cheap resources. A thread needs memory, and also the more threads we have, the more we put pressure on the operating system kernel scheduler as it needs to give CPU time to the threads. We could improve the code in listing 1.3 by using a thread pool for reusing threads after a connection has been closed, but we still need *n* threads for *n* connections at any given point in time.

Figure 1.2. Threads and blocking I/O operations

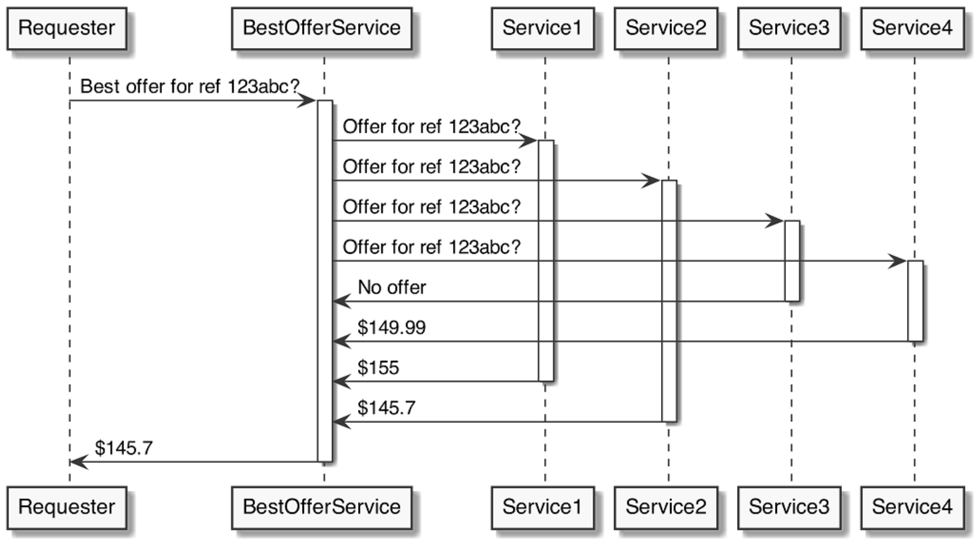


This is illustrated in figure 1.2 where we show the CPU usage over time of 3 threads for 3 concurrent network connections. Input / output operations such as `readLine` and `write` may *block* the thread, meaning that it is being parked by the operating system. This happens for 2 reasons:

1. a read operation may be waiting for data to arrive from the network, and
2. a write operation may have to wait for buffers to be drained when they are full from previous write operation.

A modern operating system can properly deal with a few thousand concurrent threads. While not every networked service is going to face loads with as many concurrent requests, this model still quickly shows its limits when we are talking about ten thousands of concurrent connections.

Figure 1.3. Request processing in a edge service



It is also important to recall that we often need more threads than incoming network connections. To take a concrete example, suppose that we are exposing a HTTP service that offers the best price for a given product reference, and to do that that service needs to request prices to 4 other HTTP services, as illustrated in figure 1.3. Note that this type of service is often called a *edge service* or an *API gateway*. Requesting each service in sequence and then selecting the lowest price would render our service very slow, as each request adds to our own service latency. The efficient way is to start 4 concurrent requests from our service, and then wait and gather their responses. This translates to starting 4 threads, hence for 1000 concurrent network requests we may be using up to 5000 threads in the worst naive case where all requests need to be processed at the same time and we don't use thread pooling or maintain persistent connections from the edge service to the requested services.

Last but not least, applications are often deployed to containerized and/or virtualized environments. This means that applications may not see all the available CPU cores, and their allocated CPU time may be limited. Available memory for processes may also be restricted, so having too many threads also eats the memory budget. Such applications have to share CPU resources with other applications, so if all applications use blocking I/O APIs then there are quickly too many threads to manage and schedule, which requires starting more server / container instances as traffic ramps up. This translates directly to increased operating costs.

1.6 Asynchronous programming with non-blocking I/O

Instead of waiting for I/O operations to complete, we can shift to *non-blocking* I/O. You may have already sampled this with the `select` function in C.

The idea behind non-blocking I/O is to request a (blocking) operation, and move on to doing other tasks until the operation is ready. For example a non-blocking read may ask for up to 256 bytes over a network socket, and the execution thread does other things (like dealing with another connection) until data has been put into buffers, ready for consumption in memory. In this model, many concurrent connections can be multiplexed on a single thread as network latency typically exceeds the CPU time it takes to read incoming bytes.

Java has long had the `java.nio` (*Java NIO*) package that offers non-blocking I/O APIs over files and networks. Back to our previous example of a TCP service that echoes incoming data, here is a possible implementation with Java non-blocking I/O in listings 1.4, 1.5, 1.6 and 1.7.

Listing 1.4. Asynchronous variant of the "echo" service: main loop

```
public class AsynchronousEcho {
  public static void main(String[] args) throws IOException {
    Selector selector = Selector.open();

    ServerSocketChannel serverSocketChannel = ServerSocketChannel.open();
    serverSocketChannel.bind(new InetSocketAddress(3000));
    serverSocketChannel.configureBlocking(false);                    ❶
    serverSocketChannel.register(selector, SelectionKey.OP_ACCEPT);  ❷

    while (true) {
      selector.select();                                             ❸
      Iterator<SelectionKey> it = selector.selectedKeys().iterator();
      while (it.hasNext()) {
        SelectionKey key = it.next();
        if (key.isAcceptable()) {                                    ❹
          newConnection(selector, key);
        } else if (key.isReadable()) {                               ❺
          echo(key);
        } else if (key.isWritable()) {                               ❻
          continueEcho(selector, key);
        }
        it.remove();                                                 ❼
      }
    }
  }
  // (...)
```

❶ We need to put the channel to non-blocking mode.
❷ The selector will notify of incoming connections.
❸ This collects all non-blocking I/O notifications.
❹ We have a new connection.
❺ A socket has received data.
❻ A socket is ready for writing again.
❼ Selection keys need to be manually removed, else they are available again in the next loop iteration.

Listing 1.4 gives the server socket channel preparation code. It opens the server socket channel and makes it non-blocking, then registers a NIO key selector for processing events. The main loop iterators over the selector keys that have events ready for processing, and dispatches to specialized methods depending on the event type (new connections, data has arrived or data can be sent again).

Listing 1.5. Asynchronous variant of the "echo" service: accepting connections

```
  private static class Context {  ❶
    private final ByteBuffer nioBuffer = ByteBuffer.allocate(512);
    private String currentLine = "";
    private boolean terminating = false;
  }

  private static final HashMap<SocketChannel, Context> contexts = new HashMap<>();

  private static void newConnection(Selector selector, SelectionKey key) throws
IOException {
    ServerSocketChannel serverSocketChannel = (ServerSocketChannel) key.channel();
    SocketChannel socketChannel = serverSocketChannel.accept();
    socketChannel
      .configureBlocking(false)
      .register(selector, SelectionKey.OP_READ);  ❷
    contexts.put(socketChannel, new Context());   ❸
  }
```

❶ The `Context` class keeps state related to the handling of a TCP connection.

❷ We set the channel to non-blocking, and declare interest in read operations.

❸ We keep all connection states in a hash map.

Listing 1.5 shows how new TCP connections are being dealt with. The socket channel that corresponds to the new connection is configured as non-blocking, then tracked for further reference in a hash map where it is being associated to some *context object*. The context depends on the application and protocol, and in our case we track the current line, wether the connection is closing, and we maintain a connection-specific NIO buffer for reading and writing data.

Listing 1.6. Asynchronous variant of the "echo" service: echoing data

```
  private static final Pattern QUIT = Pattern.compile("(\\r)?(\\n)?/quit$");

  private static void echo(SelectionKey key) throws IOException {
    SocketChannel socketChannel = (SocketChannel) key.channel();
    Context context = contexts.get(socketChannel);
    try {
      socketChannel.read(context.nioBuffer);
      context.nioBuffer.flip();
      context.currentLine = context.currentLine +
Charset.defaultCharset().decode(context.nioBuffer);
      if (QUIT.matcher(context.currentLine).find()) {
        context.terminating = true;  ❶
      } else if (context.currentLine.length() > 16) {
        context.currentLine = context.currentLine.substring(8);
      }
      context.nioBuffer.flip();  ❷
      int count = socketChannel.write(context.nioBuffer);
      if (count < context.nioBuffer.limit()) {    ❸
        key.cancel();
        socketChannel.register(key.selector(), SelectionKey.OP_WRITE);
      } else {
        context.nioBuffer.clear();
        if (context.terminating) {
          cleanup(socketChannel);
        }
      }
    } catch (IOException err) {
      err.printStackTrace();
      cleanup(socketChannel);
    }
  }
```

❶ If we find a line ending with `/quit`, then we are terminating the connection.

❷ Java NIO buffers need positional manipulations: the buffer has read data, so to write it back to the client we need to flip and return to the start position.

❸ It may happen that not all data can be written, so we stop looking for read operations and declare interest in a notification when the channel can be written again.

Listing 1.6 has the code for the `echo` method. The processing is very simple: we read data from the client socket, then attempt to write it back. If the write operation was only partial, we stop further reads, declare interest in knowing when the socket channel is writable again, then ensure all data is being written.

Listing 1.7. Asynchronous variant of the "echo" service: continuing and closing

```
  private static void cleanup(SocketChannel socketChannel) throws IOException {
    socketChannel.close();
    contexts.remove(socketChannel);
  }

  private static void continueEcho(Selector selector, SelectionKey key) throws
IOException {
    SocketChannel socketChannel = (SocketChannel) key.channel();
    Context context = contexts.get(socketChannel);
    try {
      int remainingBytes = context.nioBuffer.limit() -
context.nioBuffer.position();
      int count = socketChannel.write(context.nioBuffer);
      if (count == remainingBytes) {    ❶
        context.nioBuffer.clear();
        key.cancel();
        if (context.terminating) {
          cleanup(socketChannel);
        } else {
          socketChannel.register(selector, SelectionKey.OP_READ);
        }
      }
    } catch (IOException err) {
      err.printStackTrace();
      cleanup(socketChannel);
    }
  }
}
```

1 We remain in that state until all data has been written back, drop write interest then declare read interest.

Finally 1.7 has the method for closing the TCP connection, and the method for finishing writing a buffer. When all data has been written in `continueEcho`, then we register interest again in reading data.

As this example shows, using non-blocking I/O is doable yet it significantly increases the code complexity compared to the initial version that was using blocking APIs. The echo protocol needs 2 states for reading and writing back data: reading, or finishing writing. For more elaborated TCP protocols you can easily anticipate the need for more complicated state machines.

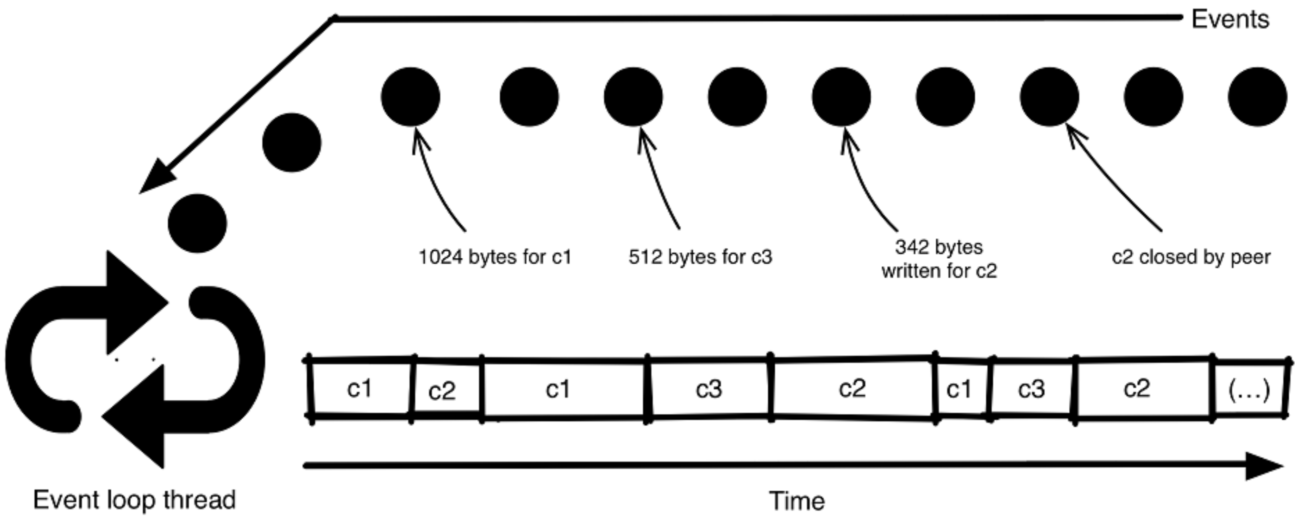
It is also important to note that like most JDK APIs, `java.nio` focuses solely on what it does (here, I/O APIs). It does not provide higher-level protocol-specific helpers like for writing HTTP clients and servers. Also, `java.nio` does not prescribe a threading model, which is still important to properly utilize CPU cores, handle asynchronous I/O events, and articulate with the application processing logic.

NOTE **This is why in practice developers rarely deal with Java NIO. Networking libraries like *Netty* and *Apache Mina* solve the shortcomings of Java NIO, and many toolkits and frameworks are built on top of them. As we will soon discover, Eclipse Vert.x is one of them.**

1.7 Multiplexing event-driven processing: the case of the event loop

A popular threading model for processing asynchronous events is that of the event-loop. Instead of polling for events that may have arrived as we did in the Java NIO example above, events are being pushed to a *the event-loop*.

Figure 1.4. Processing events using an event-loop



As we can see in figure 1.4, events are being queued as they arrive. They can be I/O events, such as data being ready for consumption, or a buffer having been fully written to a socket. They can also be any *other* event, such as a timer firing. A single thread is being assigned to an event-loop, and processing events shall not perform any blocking or long-running operation. Otherwise the thread blocks, defeating the purpose of using an event-loop. Event-loops are quite popular: JavaScript code running in web browsers runs on top of an event-loop. Many graphical interface toolkits such as *Java Swing* also have an event-loop.

Implementing an event-loop is easy.

Listing 1.8. A simple event-loop usage

```
public static void main(String[] args) {
  EventLoop eventLoop = new EventLoop();
  new Thread(() -> {    ❶
    for (int n = 0; n < 6; n++) {
      delay(1000);
      eventLoop.dispatch(new EventLoop.Event("tick", n));
    }
    eventLoop.dispatch(new EventLoop.Event("stop", null));
  }).start();
  new Thread(() -> {   ❷
    delay(2500);
    eventLoop.dispatch(new EventLoop.Event("hello", "beautiful world"));
    delay(800);
    eventLoop.dispatch(new EventLoop.Event("hello", "beautiful universe"));
  }).start();
  eventLoop.dispatch(new EventLoop.Event("hello", "world!"));   ❸
  eventLoop.dispatch(new EventLoop.Event("foo", "bar"));
  eventLoop
    .on("hello", s -> System.out.println("hello " + s))   ❹
    .on("tick", n -> System.out.println("tick #" + n))
    .on("stop", v -> eventLoop.stop())
    .run();
  System.out.println("Bye!");
}

private static void delay(long millis) {   ❺
  try {
    Thread.sleep(millis);
  } catch (InterruptedException e) {
    throw new RuntimeException(e);
  }
}
```

❶ A first thread that dispatches events every second to the event-loop.

❷ A second thread that dispatches 2 events at 2500ms and 3300ms.

❸ Events dispatched from the main thread.

❹ Event handlers defined as Java lambda functions.

❺ This method wraps a possibly *checked* exception into an *unchecked* exception to avoid polluting the `main` method code with exception handling logic.

The code in listing 1.8 shows a usage of an event-loop API whose execution gives the following console output:

Listing 1.9. Event-loop example console output

```
hello world!
No handler for key foo
tick #0
tick #1
hello beautiful world
tick #2
hello beautiful universe
tick #3
tick #4
tick #5
Bye!
```

Listing 1.10. A simple event-loop implementation

```
public final class EventLoop {
  private final ConcurrentLinkedDeque<Event> events = new
ConcurrentLinkedDeque<>();
  private final ConcurrentHashMap<String, Consumer<Object>> handlers = new
ConcurrentHashMap<>();

  public EventLoop on(String key, Consumer<Object> handler) {   ❶
    handlers.put(key, handler);
    return this;
  }

  public void dispatch(Event event) { events.add(event); } ❷
  public void stop() { Thread.currentThread().interrupt(); }

  public void run() {
    while (!(events.isEmpty() && Thread.interrupted())) {   ❸
      if (!events.isEmpty()) {
        Event event = events.pop();
        if (handlers.containsKey(event.key)) {
          handlers.get(event.key).accept(event.data);
        } else {
          System.err.println("No handler for key " + event.key);
        }
      }
    }
  }
}
```

❶ Handler are stored in a map where each key has a handler.

❷ Dispatching is pushing events to a queue.

❸ The event loop looks for events, and finds a handler based on event keys.

More sophisticated event-loop implementations are possible, but the one in listing 1.10 relies on a queue of events and a map of handlers. The event-loop runs on the thread that calls the `run`method, and events can be safely sent from other threads using the `dispatch` method.

Listing 1.11. A simple event-loop implementation

```
public static final class Event {
  private final String key;
  private final Object data;

  public Event(String key, Object data) {
    this.key = key;
    this.data = data;
  }
}
```

Last but not least, an event is simply a pair of a key and data, as given in listing 1.11 which is a static inner class of `EventLoop`.

1.8 What is a reactive system?

So far we have been discussing how to:

1. leverage asynchronous programming and non-blocking I/O to handle more concurrent connections and use less threads, and
2. use one threading model for asynchronous event processing (the event loop).

By combining these two techniques, we can build scalable and resource-efficient applications. Let us now discuss what is a *reactive system*, and how it goes beyond *"just"* asynchronous programming.

The 4 properties of *reactive systems* are exposed in *The Reactive*

Manifesto [ReactiveManifesto]: responsive, resilient, elastic and message-driven. We are not going to paraphrase the manifesto in this book, so here is a synthetic take on what these properties are about.

Elastic

Elasticity is the ability for the application to work with a variable number of instances. This is useful as elasticity allows responding to traffic spikes by starting new instances, and load-balancing traffic across instances. This has an interesting impact on the code design as shared state across instances needs to be well identified and limited (e.g., server-side web sessions). It is useful when instances report *metrics*, so that an orchestrator can decide when to start or when to stop instances depending on both the network traffic and reported metrics.

Resilient

Resiliency is partially the flip side of the coin of elasticity. When one instance crashes in a group of elastic instances, then resiliency is naturally achieved as traffic can be redirected to other instances, and also a new instance can be started if necessary. That being said there is more to resiliency. When an instance cannot fulfill a request due to some conditions, it tries to still answer in *degraded mode*. Depending on the application domain it may be possible to respond with older cached values, or even respond with empty / default data. It may also be possible to forward a request to some other, non-error instance. In the worst case an instance responds with an error, but in a timely fashion.

Responsive

Responsivity is the result of combining elasticity and resiliency. Consistent response times provide strong service-level agreement guarantees. This is achieved both thanks to the ability to start new instances if need be (to keep response times acceptable), and also because instances still respond quickly when errors arise. It is important to note that responsivity is not possible shall one component rely on a non-scalable resource, like a single central database. Indeed starting more instances do not solve the problem that they all issue request to one resource that is quickly going to be overloaded.

Message-driven

Using asynchronous message passing rather than blocking paradigms like remote-procedure calls is the key enabler to elasticity, resiliency, which lead to responsivity. This also enables dispatching messages to more instances (making the system elastic), and control the flow between message producers and message consumers (this is *back-pressure*, and we will explore it later in this book).

A reactive system exhibits these 4 properties, which make for **dependable** and **resource-efficient** systems.

WARNING

Does asynchronous imply reactive?

This is an important question, as being asynchronous is often presented as being a magic cure to software woes. Clearly, reactive implies asynchronous, but the converse is not necessarily true. As a (not so) fictitious example consider a shopping web application where users can put items in a shopping cart. This is classically done by storing items in a server-side web session. When sessions are being stored in memory or in local files then the system is not reactive, even if it internally uses non-blocking I/O and asynchronous programming. Indeed, an instance of the application cannot take over another one since sessions are application state, and in this case that state is not being replicated and shared across nodes. A reactive variant of this example would use a memory grid service (e.g., Hazelcast, Redis or Infinispan) to store the web sessions, so that incoming requests can be routed to any instance.

1.9 What else does reactive mean?

As *reactive* is a trendy term, it is being used for very different purposes. We just saw what a *reactive system* is, but there are two other popular reactive definitions as summarized in Table 1.1.

Table 1.1. All the reactive things

Reactive?	Description
Systems	Dependable applications that are message-driven, resilient, elastic and responsive.
Programming	Means for reacting to changes and events. Spreadsheet programs are a great example of reactive programming: when cell data changes, then cells having formula depending on affected cells get recomputed automatically. We will see later in this book RxJava, a popular *reactive extensions* API for Java that greatly helps coordinating asynchronous event and data processing. There also exists *functional reactive programming*, a style of programming that we won't cover in this book but for which [FunctionalReactiveProg] is a fantastic resource.
Streams	When systems exchange continuous streams of data then the classical producer / consumer problems arise. Especially it is important to provide *back-pressure* mechanisms so that a consumer can notify a producer when it is emitting too fast. With reactive streams the main goal is to reach the best throughput between systems.

1.10 What is Vert.x?

According to the website at vertx.io/, *"Eclipse Vert.x is a toolkit for building reactive applications on the JVM"*.

Initiated by Tim Fox in 2012, Vert.x is a project now fostered at the vendor-neutral Eclipse Foundation. While the first project iterations started with a strong influence of being a "Node.js for the JVM", it has since significantly deviated to providing an asynchronous programming foundation especially tailored for the specifics of the JVM.

WARNING **The essence of Vert.x**

As you may have guessed by the previous sections of this chapter, the essence of Vert.x is to process asynchronous events, mostly coming from non-blocking I/O, and the threading model is to process events in an event-loop.

It is very important to understand that Vert.x is a *toolkit* and not a *framework*: it does not provide a pre-defined foundation for your application so you are free to use Vert.x as a library inside a larger code base. Vert.x is largely unopinionated on the build tools that you should be using, how you want to structure your code, how you intend to package and deploy it, etc. A Vert.x application is an assembly of modules providing exactly what you need, and nothing more. If you don't need to access a database then your project does not need to depend on database-related APIs.

The Vert.x project is structured as follows:

1. a core project, called `vertx-core`, provides the APIs for asynchronous programming, non-blocking I/O, streaming, and convenient access to networked protocols such as TCP, UDP, DNS, HTTP or WebSockets,
2. a set of modules that are part of the community-supported Vert.x stack, such as a better web API (`vertx-web`) or databases (`vertx-redis`, `vertx-mongo`, etc),
3. a wider ecosystem of projects that provide even more functionality, such as connecting with Apache Cassandra, non-blocking I/O to communicate between system processes, etc.

Vert.x is *polyglot* as it supports most of the popular JVM languages: JavaScript, Ruby, Kotlin, Scala, Groovy and more. Interestingly, the support of these languages is not just through interoperability of these languages with Java. Idiomatic bindings are being generated, so that you can write Vert.x code in these languages that still feels natural. For example the Scala bindings use the Scala future APIs, and the Kotlin bindings leverage custom DSLs and functions with named parameters to simplify some code constructs. And of course, you can mix and match different supported languages within the same Vert.x application.

1.11 Your first Vert.x application

It is finally time for us to write a Vert.x application!

Let us continue with the echo TCP protocol that we have been using in various forms since the beginning of this chapter. It will still expose a TCP server on port 3000 where any data is being sent back to the client. We will add 2 other features:

1. the number of open connections will be displayed every 5 seconds, and
2. a HTTP server on port 8080 will respond with a string giving the current number of open connections.

1.11.1 Preparing the project

While not strictly necessary for this example, it is easier to use a build tool. In this book, I will show examples with Maven or Gradle depending on the chapters.

For this project the only third party dependency that we need is the `vertx-core` artifact plus its dependencies. This artifact is on Maven Central under the `io.vertx` group identifier.

An integrated development environment (IDE) like *IntelliJ IDEA Community Edition* is great, and it knows how to create Maven and Gradle projects. You can equally use Eclipse, Netbeans or even a Visual Studio Code.

For this chapter let us use Gradle. A suitable `build.gradle` file would look like listing 1.12.

Listing 1.12. Gradle configuration to build and run VertxEcho

```
plugins {
  id 'java'
  id 'application'
}

sourceCompatibility = 1.8
mainClassName = 'chapter1.firstapp.VertxEcho'    (1)

repositories {
  mavenCentral()
}

dependencies {
  implementation 'io.vertx:vertx-core:3.5.2'
}
```

(1) This is the fully qualified name of the class containing a `main` method so that we can use the `run` Gradle task.

1.11.2 The VertxEcho class

The class implementation is given in listing 1.14. We can run the application with Gradle using the `run` task (`gradle run` or `./gradlew run`):

Listing 1.13. Running VertxEcho

```
$ ./gradlew run

> Task :run
We now have 0 connections
We now have 0 connections
We now have 0 connections
We now have 1 connections
We now have 1 connections
Jul 07, 2018 11:44:14 PM io.vertx.core.net.impl.ConnectionBase
SEVERE: Connection reset by peer
We now have 0 connections
<=========----> 75% EXECUTING [34s]
> :run
```

Listing 1.14. Implementation of the VertxEcho class

```
package chapter1.firstapp;

import io.vertx.core.Vertx;
import io.vertx.core.net.NetSocket;

public class VertxEcho {

  private static int numberOfConnections = 0;    ❶

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

    vertx.createNetServer()
      .connectHandler(VertxEcho::handleNewClient)    ❷
      .listen(3000);

    vertx.setPeriodic(5000, id -> System.out.println(howMany()));    ❸

    vertx.createHttpServer()
      .requestHandler(request -> request.response().end(howMany())) ❹
      .listen(8080);
  }

  private static void handleNewClient(NetSocket socket) {
    numberOfConnections++;
    socket.handler(buffer -> {    ❺
      socket.write(buffer);
      if (buffer.toString().endsWith("/quit\n")) {
        socket.close();
      }
    });
    socket.closeHandler(v -> numberOfConnections--);    ❻
  }

  private static String howMany() {
    return "We now have " + numberOfConnections + " connections";
  }
}
```

❶ As we will see in the next chapter, event handlers are always executed on the same thread, so there is no need for JVM locks or using `AtomicInteger`.
❷ Creating a TCP server requires passing a callback for each new connection.
❸ This defines a periodic task by a callback being executed every 5 seconds.
❹ Similarly to a TCP server, a HTTP server is being configured by given the callback to be executed for each HTTP request.
❺ The buffer handler is being invoked every time a buffer is ready for consumption. Here we just write it back, and also use a convenient string conversion helper to look for a terminal command.
❻ Another event is when the connection closes. We decrement a connections counter that was incremented upon connection.

This example is interesting in that we have few lines of code. It is centered around a plain old Java `main` method, as there is no framework to bootstrap. All we need to create is a `Vertx` context, which in turns offers methods to create tasks, servers, clients and more as we will discover in the next chapters.

While not apparent here, an event-loop is managing the processing of events, be it a new TCP connection, the arrival of a buffer, a new HTTP request, or a periodic task that is being fired. Also, every event handler is being executed on the same (event-loop) thread.

1.11.3 The role of callbacks

As we have just seen in listing 1.14, *callbacks* are the primary method for Vert.x to notify of asynchronous events and pass them to some handlers. Combined with lambda expressions in Java, they make for a concise way to define event handling.

You may have heard or experienced the infamous *"callback hell"* where callbacks get nested into callbacks, leading to code that is difficult to read and reason about:

Listing 1.15. Callback hell illustrated

```
dothis(a -> {
  dothat(b -> {
    andthis(c -> {
      andthat(d -> {
        alsothis(e -> {
          alsothat(f -> {
            // ...
          });
        });
      });
    });
  });
});
```

Be reassured: while the Vert.x core APIs indeed use callbacks, Vert.x provides support for more programming models. Callbacks are the canonical mean for notification in event-driven APIs, but as we will see in the next chapters it is possible to build other abstractions on top of them such as future and promises, reactive extensions or coroutines.

While callbacks have their issues, there are many cases with minimal levels of nesting where they remain a very good programming model with minimal dispatch overhead.

1.11.4 So is this a reactive application?

This is a very good question to ask. It is important to remember that while Vert.x is a toolkit for building reactive applications, using the Vert.x API and modules does not *"auto-magically"* make an application a reactive one. Yet, the event-driven, non-blocking APIs that Vert.x provides tick the first box.

The short answer is that no, this application is not reactive. Resiliency is not the issue as the only errors that can arise are I/O related, and they simply result in discarding the connections. The application is also responsive, as it does not perform any complicated processing. If we benchmarked the TCP and HTTP servers we would get very good latencies with low deviation and very few outliers. Here is an imperfect yet telling quick benchmark with `wrk` [2] :

Listing 1.16. Output of a benchmark session with wrk

```
$ wrk --latency http://localhost:8080/
Running 10s test @ http://localhost:8080/
  2 threads and 10 connections
  Thread Stats   Avg      Stdev     Max   +/- Stdev
    Latency   136.98us  106.91us   7.26ms   97.37%
    Req/Sec    36.62k     4.09k   45.31k    85.64%
  Latency Distribution
     50%  125.00us
     75%  149.00us
     90%  199.00us
     99%  340.00us
  735547 requests in 10.10s, 44.89MB read
Requests/sec:  72830.90
Transfer/sec:      4.45MB
```

The culprit for not being reactive clearly is elasticity. Indeed if we create new instances, each instance maintains its own connection counter. The counter scope is the application, so it should be a shared global counter between all instances.

As this example shows, designing reactive applications is more subtle than just implementing responsive and resource-efficient systems. Ensuring that an application can run as many replaceable instances is surprisingly more engaging, especially as we need to think about *instance state* versus *application state* to make sure that instances are interchangeable.

1.12 What are the alternatives to Vert.x?

As you will see with this book, Vert.x is a compelling technology for building end-to-end reactive applications. Reactive application development is a trendy topic and it is more important to understand the principles than blindly becoming an expert in one specific technology. What you will learn in this book easily transfers to other technologies and I highly encourage you to check them out. Here are the most popular alternatives to Vert.x for asynchronous and reactive programming.

Node.js is an event-driven runtime for writing asynchronous JavaScript applications [NodejsInAction]. It is based on the V8 JavaScript engine that is used by Google Chrome. At first sight Vert.x and Node.js have lots of similarities. Still, they differ greatly. Vert.x runs multiple event-loops by default, unlike Node.js. Also, the JVM has a better JIT compiler and garbage collector, so the JVM is better suited for long-running processes. Last but not least Vert.x supports JavaScript.

Akka is a faithful implementation of the *Actor* model [AkkaInAction]. It runs on the JVM and primarily offers Scala APIs, although Java bindings are also being promoted.

[2] See github.com/wg/wrk

Akka is particularly interesting as actors are message-driven, location transparent, and actors offer supervision features that are interesting for error-recovery. Akka clearly targets the design of reactive applications [ReactiveAppDevelopment]. As we will see in this book Vert.x is no less capable for the task. Vert.x has a concept of *verticles* that are used for processing asynchronous events. As we will see later in this book, verticles are loose form of actors. Interestingly, Vert.x is significantly faster than Akka and most alternatives in established benchmarks such as [TechEmpower].

The older and widespread Spring Framework now integrates a reactive stack [SpringInAction]. It is based around *Project Reactor*, an API for reactive programming that is very similar to RxJava. The focus on *Spring Reactive* is essentially on reactive programming APIs, but it does not necessarily lead to end-to-end reactive applications. Many parts in the Spring Framework employ blocking APIs, so extra care must be taken to limit the exposure to blocking operations. Project Reactor is a compelling alternative to RxJava, but *Spring Reactive* is tied to this API and it may not always be the best way to express certain asynchronous constructions. Vert.x provides more flexibility as it supports callbacks, futures, Java `CompletionStage`, Kotlin coroutines, RxJava and fibers. This means that with Vert.x it is easier to select the right asynchronous programming model for a certain task. Also like with Akka, Vert.x remains significantly faster in [TechEmpower], and applications boot faster than Spring-based ones.

The Netty framework provides non-blocking I/O APIs for the JVM [NettyInAction]. It provides abstractions and platform-specific bug fixes compared to using raw NIO APIs. It also provides threading models. The target of Netty is low-latency and high-performance network applications. While you can certainly build reactive applications with Netty, the APIs remain somehow low-level. Vert.x is one of the many technologies built on top of Netty (Spring Reactive and Akka have Netty integration), and you can get all the performance benefits of Netty with the simpler APIs of Vert.x.

Scripting languages such as Python and Ruby also provide non-blocking I/O libraries such as *Async* (Ruby) and *Twisted* (Python). You can certainly build reactive systems with them. Again, the JVM performance is an advantage for Vert.x, along with the ability to use alternative JVM languages (Ruby is officially supported by Vert.x).

Native languages are becoming trendy again. Instead of using the venerable C/C++ languages, Go [GoInAction], Rust [RustInAction] and Swift [SwiftInDepth] are gaining mindshare. They all tick the boxes for building highly scalable applications, and they certainly can be used for creating reactive applications. That being said most efficient libraries in these languages are fairly low-level, and ultimately the JVM-based Vert.x / Netty combination still ranks favorably in benchmarks [TechEmpower].

1.13 Summary

In this chapter we learned:

- what is asynchronous programming,
- the challenges that non-blocking I/O pose even for simple protocols,

- what is an event-loop, and how it simplifies processing asynchronous events,
- what is a reactive system
- how to get started with Vert.x.

In the next part, we are going to dissect the fundamentals of asynchronous programming with Vert.x.

1.14 References

[ReactiveManifesto] Jonas Bonér, Dave Farley, Roland Kuhn, Martin Thompson and contributors. The Reactive Manifesto. 2014. Published at www.reactivemanifesto.org/

[NodejsInAction] Mike Cantelon, Marc Harter, T.J. Holowaychuk, and Nathan Rajlich. Node.js in Action. 2013. Manning Publications. ISBN 9781617290572.

[AkkaInAction] Raymond Roestenburg, Rob Bakker, and Rob Williams. Akka in Action. 2016. Manning Publications. ISBN 9781617291012.

[TechEmpower] TechEmpower benchmarks. Regularly updated and published at www.techempower.com/benchmarks/

[SpringInAction] Craig Walls. Spring in Action, Fifth edition. 2018. Manning Publications. ISBN 9781617294945.

[GoInAction] William Kennedy with Brian Ketelsen and Erik St. Martin. Go in Action. 2015. Manning Publications. ISBN 9781617291784.

[RustInAction] Tim McNamara. Rust in Action. 2019. Manning Publications. ISBN 9781617294556.

[NettyInAction] Norman Maurer and Marvin Allen Wolfthal. Netty in Action. 2015. Manning Publications. ISBN 9781617291470.

[SwiftInDepth] Tjeerd in 't Veen. Swift in Depth. 2018. Manning Publications. ISBN 9781617295188.

[ReactiveAppDevelopment] Duncan K. DeVore, Sean Walsh, and Brian Hanafee. Reactive Application Development. 2018. Manning Publications. ISBN 9781617292460.

[FunctionalReactiveProg] Stephen Blackheath and Anthony Jones. Functional Reactive Programming. 2016. Manning Publications. ISBN 9781633430105.