**VERT.X PROJECTS**

After a very thorough research into what sort of tools or toolkit or platform for some API I have been planning for some time, vertx came up as the final and possible immediate solution or option for the level of performance I was aiming for.

In my research I used the following websites for my Benchmarking checks and confirmations :

|  |
| --- |
| **Other Benchmarking websites :** |
| <https://dev.to/tuananhpham/popular-backend-frameworks-performance-benchmark-1bkh> |
| <https://www.tiobe.com/tiobe-index/> |
| <https://pypl.github.io/PYPL.html#google_vignette> |
| <https://trends.builtwith.com/framework> |
| <https://benchmarksgame-team.pages.debian.net/benchmarksgame/index.html> |
| <https://programming-language-benchmarks.vercel.app/> |
| <https://programming-language-benchmarks.vercel.app/java> |
| <https://programming-language-benchmarks.vercel.app/java-vs-go> |
| <https://www.quora.com/What-are-the-best-alternatives-to-Java-for-high-performance-backend-development> |
| <https://news.ycombinator.com/item?id=17254152> |
| <https://github.com/rwf2/Rocket/issues/710> |
| <https://just.billywhizz.io/blog/on-javascript-performance-01/> |
| [https://www.techempower.com](https://www.techempower.com/) |

Although, one could argue that the way benchmarking is done or executed, it might not be the same as what you might experience in a production environment. And for that I say : If the framework or platform or tool or toolkit or language of choice performed at this level for a very simple hello world app or sample project or in applying a. very simple hello world json structure, then , that is just it. All the other technologies are been benchmarked with same. You better believe it is just what it is. If you did pull the sample technologies and tested them your results would not be too far away nor apart from what has already been done.  
  
Below is a composite framework scoring ( source : techempower.com )

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **­­­** | | | |  | **Source** : *techempower.com* | | | | |  |  |  |  |
| *Each framework's peak performance in each test type (shown in the colored columns below) is multiplied by the weights shown above. The results are then summed to yield a weighted score. Only frameworks that implement all test types are included. 159 total frameworks ranked, 10 visible, 149 hidden by filters. See filter panel above.* | | | | | | | | | |  | **Hardware :** Citrine Dell R440 Xeon Gold + 10 GbE | | |
| Rnk | Framework | JSON | 1-query | 20-query | Fortunes | Updates | Plaintext | Weighted score | | Min | Max | Average |  |
| 5 | redkale | 1,210,086 | 457,935 | 32,272 | 413,537 | 22,900 | 6,981,831 | **7,096** | 87.80% | 22,900 | 6,981,831 | 1,519,760 |  |
| 7 | vert.x | 1,194,427 | 581,542 | 32,505 | 453,406 | 17,122 | 5,624,745 | **6,847** | 84.70% | 17,122 | 5,624,745 | 1,317,291 |  |
| 11 | jooby | 1,162,248 | 515,795 | 32,196 | 427,682 | 15,889 | 4,801,653 | **6,370** | 78.80% | 15,889 | 4,801,653 | 1,159,244 |  |
| 16 | vertx-web | 1,121,889 | 566,804 | 30,669 | 388,512 | 16,598 | 3,774,206 | **6,107** | 75.50% | 16,598 | 3,774,206 | 983,113 |  |
| 19 | inverno | 1,042,404 | 467,759 | 32,441 | 309,684 | 15,864 | 4,043,496 | **5,566** | 68.80% | 15,864 | 4,043,496 | 985,275 |  |
| 38 | quarkus | 903,185 | 318,897 | 17,610 | 214,275 | 6,697 | 2,861,479 | **3,637** | 45.00% | 6,697 | 2,861,479 | 720,357 |  |
| 50 | actframework | 964,004 | 231,641 | 16,942 | 124,422 | 1,985 | 3,273,101 | **2,911** | 36.00% | 1,985 | 3,273,101 | 768,683 |  |
| 55 | javalin | 512,495 | 211,243 | 16,582 | 161,275 | 10,405 | 897,788 | **2,755** | 34.10% | 10,405 | 897,788 | 301,631 |  |
| 57 | revenj.jvm | 527,667 | 290,147 | 14,330 | 177,853 | 5,206 | 730,158 | **2,543** | 31.40% | 5,206 | 730,158 | 290,894 |  |
| 125 | wicket | 378,043 | 24,624 | 1,426 | 25,882 | 551 | 516,369 | **679** | 8.40% | 551 | 516,369 | 157,816 |  |

I had been pretty much used to developing some very sophisticated API solutions using micronaut for some fintech backend that had processed in excess of $40million by the time it was been considered for an acquisition deal/discussion. And at the point, the final part of the negotiation(s) was left to my C.E.O to take over.

So in my quest to decide on what tools, or tool-kit or framework and platform to use for my next big fintech backend vertx had become the obvious. Meanwhile, I took some time off to consider the differences and general comparison between micronaut and vertx.

**Micronaut Framework vs Vert.x: What are the differences?**

**Introduction**

Micronaut Framework and Vert.x are both popular frameworks used for developing microservices and reactive applications. While they share some similarities, there are key differences that set them apart from each other.

1. **Execution Model**: *Micronaut* adopts a more traditional Java execution model, utilizing compile-time annotation processing to minimize reflection and enhance performance. On the other hand, *Vert.x* is event-driven and non-blocking, making it well-suited for highly concurrent applications.
2. **Language Support**: *Micronaut* primarily focuses on providing support for the Java programming language, although it also offers limited compatibility with Kotlin and Groovy. Alternatively, *Vert.x* is polyglot, meaning it supports multiple languages, including Java, Kotlin, JavaScript, Groovy, Ruby, Python, and more.
3. **Dependency Injection**: *Micronaut* has a built-in dependency injection framework that leverages compile-time DI, leading to faster startup times and decreased memory consumption. Conversely, *Vert.x* does not have its own DI framework and instead encourages the use of external libraries such as Dagger or Spring for dependency injection.
4. **Web Support**: *Micronaut* provides comprehensive support for building RESTful APIs and web applications out-of-the-box, including features like server-side templating and built-in support for HTTP clients. *Vert.x*, on the other hand, offers more low-level control over the web stack and allows for the creation of various types of applications, including not only traditional web apps but also real-time websockets and event-driven web systems.
5. **Concurrency Model**: *Micronaut* leverages thread pools and CompletableFuture for performing tasks concurrently and handling asynchronous operations. *Vert.x*, being an event-driven framework, uses an event loop model and employs a single-threaded model, wherein a single event loop can handle multiple requests concurrently by utilizing non-blocking I/O operations.
6. **Development Approach**: *Micronaut* utilizes a compile-time approach, where it analyzes your project's classpath during the build phase to generate factory classes and metadata. This, in turn, reduces the amount of reflection required at runtime. *Vert.x*, on the other hand, takes a more runtime approach and relies more heavily on dynamic features and runtime reflection.

In Summary, Micronaut Framework focuses on optimizing performance, enables compile-time DI, and provides efficient Java support, while Vert.x emphasizes its polyglot nature, event-driven architecture, and flexible web support.

*[ source : stackshare.io ]*

So basically after trying out a number of the technologies listed thereof, I settled on vert.x.



Starting a vertx project is as simple as getting your preferred IDE eg. IntelliJ Community Package version or Eclipse or Visual Studio Code and creating a maven or gradle project and ensuring a dependency to and for io.vertx is set in place.  
  
Or you can start from the vertx starter link to generate a downloadable package or project to start from after successfully importing into the/an IDE as a project.

You can use the Vert.x starter web application at https://start.vertx.io and generate a project skeleton to download



After successfully importing the project,

You can then run the following command :

$ mvn clean install

You can expect some <Build Successful> kind of message at the end to be sure that everything worked out perfectly, as seen below :

[INFO] Copying com.aireceive.firstvertxapp:starter:pom:1.0.0-SNAPSHOT to project local repository

[INFO] Copying com.aireceive.firstvertxapp:starter:jar:1.0.0-SNAPSHOT to project local repository

[INFO] Copying com.aireceive.firstvertxapp:starter:pom:consumer:1.0.0-SNAPSHOT to project local repository

[INFO] --------------------------------------------------------------------------------------------------------------------------

[INFO] **BUILD SUCCESS**

[INFO] --------------------------------------------------------------------------------------------------------------------------

[INFO] Total time: 3.020 s

[INFO] Finished at: 2024-11-04T14:38:33Z

[INFO] --------------------------------------------------------------------------------------------------------------------------

To install the application and the respective dependencies.

**Caution :**

I tried the following commands and it did not work out as expected :

$ vertx run com.arieceive.firstvertxapp.starter.MainMerticle

And the application did not run but rather I had the following error :

Unrecognized VM option 'UseBiasedLocking'

Error: Could not create the Java Virtual Machine.

Error: A fatal exception has occurred. Program will exit.

I also tried the same command and later this error :

Can not find io.netty.resolver.dns.macos.MacOSDnsServerAddressStreamProvider in the classpath, fallback to system defaults. This may result in incorrect DNS resolutions on MacOS. Check whether you have a dependency on 'io.netty:netty-resolver-dns-native-macos'

Failed in deploying verticle

java.lang.ClassNotFoundException: com.arieceive.firstvertxapp.starter.MainMerticle

at java.base/jdk.internal.loader.BuiltinClassLoader.loadClass(BuiltinClassLoader.java:581)

at java.base/jdk.internal.loader.ClassLoaders$AppClassLoader.loadClass(ClassLoaders.java:178)

at java.base/java.lang.ClassLoader.loadClass(ClassLoader.java:527)

at io.vertx.core.impl.JavaVerticleFactory.createVerticle(JavaVerticleFactory.java:41)

at io.vertx.core.impl.VerticleManager.doDeployVerticle(VerticleManager.java:217)

at io.vertx.core.impl.VerticleManager.doDeployVerticle(VerticleManager.java:193)

at io.vertx.core.impl.VerticleManager.doDeployVerticle(VerticleManager.java:180)

at io.vertx.core.impl.VerticleManager.deployVerticle(VerticleManager.java:156)

at io.vertx.core.impl.VertxImpl.deployVerticle(VertxImpl.java:794)

at io.vertx.core.impl.VertxImpl.deployVerticle(VertxImpl.java:800)

at io.vertx.core.impl.launcher.commands.VertxIsolatedDeployer.deploy(VertxIsolatedDeployer.java:42)

at java.base/jdk.internal.reflect.NativeMethodAccessorImpl.invoke0(Native Method)

at java.base/jdk.internal.reflect.NativeMethodAccessorImpl.invoke(NativeMethodAccessorImpl.java:62)

at java.base/jdk.internal.reflect.DelegatingMethodAccessorImpl.invoke(DelegatingMethodAccessorImpl.java:43)

at java.base/java.lang.reflect.Method.invoke(Method.java:566)

at io.vertx.core.impl.launcher.commands.ClasspathHandler.deploy(ClasspathHandler.java:169)

at io.vertx.core.impl.launcher.commands.RunCommand.deploy(RunCommand.java:398)

at io.vertx.core.impl.launcher.commands.RunCommand.run(RunCommand.java:260)

at io.vertx.core.impl.launcher.VertxCommandLauncher.execute(VertxCommandLauncher.java:248)

at io.vertx.core.impl.launcher.VertxCommandLauncher.dispatch(VertxCommandLauncher.java:383)

at io.vertx.core.impl.launcher.VertxCommandLauncher.dispatch(VertxCommandLauncher.java:346)

at io.vertx.core.Launcher.main(Launcher.java:45)

So the next thing was to try running the generated directly.

I tried : ( from the root folder of the project )

$ java -jar target/starter-1.0.0-SNAPSHOT.jar

And I got this feedback :

no main manifest attribute, in target/starter-1.0.0-SNAPSHOT.jar

What worked out find in getting application running :

But at last, after trying this :

$ java -jar target/starter-1.0.0-SNAPSHOT-fat.jar

I was then able to get the application running with a successful test using a web browser.  
  
Nov 04, 2024 2:34:20 PM io.netty.resolver.dns.DnsServerAddressStreamProviders <clinit>

WARNING: Can not find io.netty.resolver.dns.macos.MacOSDnsServerAddressStreamProvider in the classpath, fallback to system defaults. This may result in incorrect DNS resolutions on MacOS. Check whether you have a dependency on 'io.netty:netty-resolver-dns-native-macos'

HTTP server started on port 8888

Nov 04, 2024 2:34:20 PM io.vertx.core.impl.launcher.commands.VertxIsolatedDeployer

INFO: Succeeded in deploying verticle

My first vertx web app :

[INFO] Copying com.aireceive.firstvertxwebapp:vertx-web-starter:pom:1.0.0-SNAPSHOT to project local repository

[INFO] Copying com.aireceive.firstvertxwebapp:vertx-web-starter:jar:1.0.0-SNAPSHOT to project local repository

[INFO] Copying com.aireceive.firstvertxwebapp:vertx-web-starter:pom:consumer:1.0.0-SNAPSHOT to project local repository

[INFO] --------------------------------------------------------------------------------------------------------------------------

[INFO] **BUILD SUCCESS**

[INFO] --------------------------------------------------------------------------------------------------------------------------

[INFO] Total time: 3.472 s

[INFO] Finished at: 2024-11-04T16:07:02Z

[INFO] --------------------------------------------------------------------------------------------------------------------------

➜ vertx-web-starter git:(main) ✗ java -jar target/vertx-web-starter-1.0.0-SNAPSHOT-fat.jar

Nov 04, 2024 4:07:18 PM io.netty.resolver.dns.DnsServerAddressStreamProviders <clinit>

WARNING: Cannot find io.netty.resolver.dns.macos.MacOSDnsServerAddressStreamProvider in the classpath, fallback to system defaults. This may result in incorrect DNS resolutions on MacOS. Check whether you have a dependency on 'io.netty:netty-resolver-dns-native-macos'

HTTP server started on port 8889

Nov 04, 2024 4:07:18 PM io.vertx.core.impl.launcher.commands.VertxIsolatedDeployer

INFO: Succeeded in deploying verticle



Alternative to Vert.x for asynchronous and reactive programming :

1. NodeJS
2. Akka
3. Spring Framework
4. Quarkus
5. Netty
6. Scripting languages
7. Native languages

Vert.x is the best compared to all the above in so many ways, thanks to the JVM to its advantage. Some of the other options above might have their own benefits in various ways meanwhile for a high performant production system, Vertx would always win in over 82% of the key metrics been checked against.

Summary to this point :

1. Asynchronous programming allows you to handle multiple multiplex networked connections on a single thread.
2. Handling or managing non-blocking I/O is more complex than the equivalent imperative code base on blocking I/O, even for simple protocols.
3. Asynchronous event processing is simplified by the event loop and the reactor pattern.
4. Despite the demanding workloads and failures, a reactive system is both scalable and resilient, producing responses with consistent latencies.
5. Vert.x is an efficient and approachable toolkit for writing asynchronous and reactive applications on the JVM.

**Verticles**

A verticle is the fundamental processing unit in Vert.x, with a life cycle. The role of a verticle is fundamentally to envelope a single technical functional unit for processing events, such as ;

1. Exposing an HTTP API
2. Responding to requests
3. Providing a repository interface on top of a database
4. Issuing requests to a third-party system

Characteristics of Verticles :

1. Verticles exclusively can communicate with other entities by sending and responding to messages.
2. Verticles have private state that may be updated when receiving events, they can deploy other verticles.
3. Verticles can communicate via message-passing
4. Verticles do not necessarily follow the orthodox definition of actors

Now, let’s build a cute little verticle that processes just two types of events for now :

1. Periodic timers
2. Handle HTTP requests

The life cycle of a verticle is basically :

1. Start , and
2. Stop

The start method is typically used in setting things up and fundamentally initializing handlers.

The stop method is used in doing housekeeping tasks, such as closing an opened database connection(s).

To run a verticle, it can basically be run from the mian of the java class that it is in.

Or you can also run it on the command line using Gradle such as :

**$ ./gradlew run -PmainClass=com.example.hello.HelloVerticle**

**Some few experiments :**

So I had to run a few experiments. I decided to check if I could run a single verticle and rather apply two port numbers to see how the behaviour of a dual starting of the same verticle would be and to my surprise, it is only the first verticle start via either the terminal or by directly running it from the “main” method that responds to all the requests to the running verticle.



In the deployment of verticles it is important to pay attention to any action or function or task or tasks that might attempt to block the eventLoop or life cycle of the said verticle. In Vertx, we have a threadchecker that checks to be sure that no process or action or function or task is taking more than a predefined internally set thread checker time limit. The default might be found to be 2000 as of the time I am doing all these tests and experiments. Meanwhile, there are certain environments such as embedded devices, where processing power is a bit slower, and it is normal to increase the thread-checker threshold for such cases.

You can do that by setting the the system properties to change the value as seen below :

-Dvertx.options.blockedThreadCheckInterval=5000

Or if you prefer to disable it :

-Dvertx.threadChecks=false

It must be noted that this configuration is global and cannot be fine-tuned on a per-verticle basis.

It is such a great robust way to or practice to use asynchronous method variants that accept a callback to notify of any errors, example the listen method in the creation of an HTTP server using vertx.

Do check from screenshot below :



**Deploying Verticles:**

We can have a verticle successfully deploy another verticle and repeat same for another verticle. Meanwhile, there is not direct parent or child relationship setup for verticle deployment.

One other thing to note about verticles are that Vert.x creates double number of event-loop threads as the number of CPU Cores present, by default. Basically implies that if you have 4cores, then a Vert.x application has 8 event loops. And the assignment of the verticles to event loops is/are done in a rounded-robin fashion or style.

In the deployment of verticles and by extension the number of event loops, can be managed in a way so as to obtain a certain number of event loops that has to be available. Meanwhile it is not possible to allocate a given verticle to a specific event loop manually. In practice, this should never be a problem whatsoever. It must be noted that we can plan the deployment order of verticles.



**Passing Configuration Data**

**Worker Verticles**

**Threads Contextualisation**

**Threads Mix Monitoring and Management**

**Event Bus : A very important part of Vert.x**

There are three major patterns in event bus communications :

1. Point-to-point messaging
2. Request-reply messaging
3. Publish/subscribe messaging

The event bus is not a message broker but rather carries volatile event that are been processed asynchronously by verticles, more specifically the Vert.x event bus is not able to do the following :

1. Support message acknowledgements
2. Support message priorities
3. Support message durability to recover from crashes
4. Provide routing rules
5. Provide transformation rules (schema adaptation, scatter/gather, etc.)
6. Distribute fewer messages to an overloaded consumer.

If the developer or engineer wants a way to handle communications relating to events without ever loosing them then a middleware would be very necessary.

**Point-to-Point messaging**

Messages from producers are shared proportionally in a round-robin fashion amongst consumers.

**Latter Sudden challenges in getting vert.x running smoothly & the way out :**

As I progressed into the deeper parts of vert.x development, deployment and its general usage. I was trying to run some vert.x application using version 4.5.10 and I had this error below :

Nov 06, 2024 11:58:45 AM io.netty.resolver.dns.DnsServerAddressStreamProviders <clinit>

WARNING: Can not find io.netty.resolver.dns.macos.MacOSDnsServerAddressStreamProvider in the classpath, fallback to system defaults. This may result in incorrect DNS resolutions on MacOS. Check whether you have a dependency on 'io.netty:netty-resolver-dns-native-macos'

I then compared the ‘pom.xml’ (under properties) file from a previous successfully running vert.x application :

<properties>  
 <maven.compiler.source>1.8</maven.compiler.source>  
 <maven.compiler.target>1.8</maven.compiler.target>  
 <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>  
 <vertx.version>4.0.3</vertx.version>

The new vertx application’s pom.xml file under the properties :

<properties>  
 <maven.compiler.source>1.8</maven.compiler.source>  
 <maven.compiler.target>1.8</maven.compiler.target>  
 <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>  
 <vertx.version>4.5.10</vertx.version>

The new vertx application’s pom.xml file above was then changed to :

<properties>  
 <maven.compiler.source>1.8</maven.compiler.source>  
 <maven.compiler.target>1.8</maven.compiler.target>  
 <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>  
 <vertx.version>4.0.3</vertx.version>

The application was then able to run smoothly without any issues whatsoever.

Access the index.html file :

On a macbook M1 using inteliJ and with the “index.html” at the root of the project, the file could not be rendered.

The file was only able to be rendered after placing the file inside the “resource” folder or package.

**Clustering and distributed event bus**

In attempting to run a cluster that would have brought the application development in vertx to a whole new level of deployment and management, I started getting errors to ensure some infinispan.xml is created before the cluster would run.

I tried running from the main : did not work without the infinispan.xml

I tried running using ./gradlew run : did not work without the infinispan.xml

I tried running using mvn compile exec:java : did not work without the infinispan.xml

**Infinispan Replicated Cache**



**Starting Infinispan Server**

Before you start coding fun stuff, you need to start Infinispan Server. For this tutorial, you need a locally running server instance.

You can do one of the following:

* Pull the container image and run with [Docker](https://www.docker.com/) or [Podman](https://podman.io/).
* Download the server distribution and extract it to your filesystem.

Credentials

By default, Infinispan Server requires user authentication. This tutorial uses admin and secret credentials but you can use any username and password.

**Running the Container Image**

The easiest way to run Infinispan Server locally is to pull the container image.

* Podman

podman run --net=host -p 11222:11222 -e USER="admin" -e PASS="secret" quay.io/infinispan/server:latest

* Docker

docker run -it -p 11222:11222 -e USER="admin" -e PASS="password" infinispan/server:latest

**Running the Server Distribution**

Infinispan Server comes as a bare metal distribution that you can run locally.

1. Download the server distribution from [Infinispan Downloads](https://infinispan.org/download/#stable) and extract it.
2. Open a terminal window in the resulting directory. This is $ISPN\_HOME.
3. Add credentials.

$ ./bin/cli.sh user create admin -p secret

Run Infinispan Server.

$ ./bin/server.sh

**Accessing the Infinispan Console**

Open <http://localhost:11222/> in any browser.

You’ll see the **Welcome to Infinispan Server** page.

[](https://github.com/infinispan/infinispan-server-tutorial/blob/main/images/welcomeConsole.png)

To start using the Infinispan Console, do the following:

1. Select **Go to the console**.
2. Enter your credentials (admin/secret).

Successfully have gotten infinispan running :



I then went on to generate this configuration in XML and JSON :

XML :

<?xml version="1.0"?>

<distributed-cache owners="2" mode="ASYNC" statistics="true">

<encoding media-type="application/x-protostream"/>

<locking concurrency-level="32" acquire-timeout="10"/>

<memory max-size="1024MB" when-full="REMOVE"/>

<persistence passivation="false" availability-interval="1000" connection-attempts="10" connection-interval="50">

<file-store>

<data path="data"/>

<index path="index"/>

</file-store>

</persistence>

</distributed-cache>

Improved to :

<distributed-cache *owners*="2" *mode*="ASYNC" *statistics*="true">  
 <encoding *media-type*="application/x-protostream"/>  
 <locking *concurrency-level*="32" *acquire-timeout*="10"/>  
 <memory *max-size*="1024MB" *when-full*="REMOVE"/>  
 <persistence *passivation*="false" *availability-interval*="1000" *connection-attempts*="10" *connection-interval*="50">  
 <file-store>  
 <data *path*="data"/>  
 <index *path*="index"/>  
 </file-store>  
 </persistence>  
 <infinispan>  
 <cache-container>  
 <transport *initial-cluster-size*="4"  
 *initial-cluster-timeout*="30000" />  
 </cache-container>  
 </infinispan>  
</distributed-cache>

Changed to :

<distributed-cache>

<memory storage="HEAP"/>

</distributed-cache>

Changed to :

<distributed-cache>  
 <memory  
 *storage*="HEAP"  
 *max-count*="500"  
 *when-full*="REMOVE"  
 />  
 <infinispan>  
 <cache-container>  
 <transport *initial-cluster-size*="4"  
 *initial-cluster-timeout*="30000" />  
 </cache-container>  
 </infinispan>  
  
</distributed-cache>

JSON :

{

"distributed-cache": {

"owners": "2",

"mode": "ASYNC",

"statistics": true,

"encoding": {

"media-type": "application/x-protostream"

},

"locking": {

"concurrency-level": "32",

"acquire-timeout": "10"

},

"memory": {

"max-size": "1024MB",

"when-full": "REMOVE"

},

"persistence": {

"passivation": false,

"availability-interval": "1000",

"connection-attempts": "10",

"connection-interval": "50",

"file-store": {

"data": {

"path": "data"

},

"index": {

"path": "index"

}

}

}

}

}

I then copied the XML version into the infinispan.xml file.

**Replicated cache:**

<replicated-cache>

<expiration lifespan="5000" max-idle="1000" />

</replicated-cache>

<replicated-cache>  
 <expiration *lifespan*="5000" *max-idle*="1000" />  
 <memory *storage*="OFF\_HEAP" *max-count*="500"/>  
  
 <infinispan>  
 <cache-container>  
 <transport *initial-cluster-size*="4"  
 *initial-cluster-timeout*="30000" />  
 </cache-container>  
 </infinispan>  
  
</replicated-cache>

<?*xml version*="1.0"?>  
<?*xml version*="1.0" *encoding*="UTF-8"?>  
<infinispan *xmlns:xsi*="http://www.w3.org/2001/XMLSchema-instance"  
 *xsi:schemaLocation*="urn:infinispan:config:7.1 http://www.infinispan.org/schemas/infinispan-config-7.1.xsd"  
 *xmlns*="urn:infinispan:config:7.1">  
  
 <cache-container *default-cache*="default">  
 <local-cache *name*="tableCache">  
 <eviction *max-entries*="200" />  
 <expiration *lifespan*="600000" />  
 </local-cache>  
 <transport  
 *initial-cluster-size*="4"  
 *initial-cluster-timeout*="30000" />  
 </cache-container>  
  
</infinispan>

Other configurations :

<distributed-cache>

<persistence>

<table-jdbc-store xmlns="urn:infinispan:config:store:sql:15.0"

dialect="H2"

shared="true"

table-name="books">

<connection-pool connection-url="jdbc:h2:mem:infinispan"

username="sa"

password="changeme"

driver="org.h2.Driver"/>

<write-behind modification-queue-size="2048"

fail-silently="true"/>

</table-jdbc-store>

</persistence>

</distributed-cache>

<distributed-cache>

<persistence>

<connection-pool connection-url="jdbc:h2:mem:infinispan;DB\_CLOSE\_DELAY=-1"

username="sa"

password="changeme"

driver="org.h2.Driver"/>

</persistence>

</distributed-cache>

##### Managed datasource configuration

XML

JSON

YAML

<server xmlns="urn:infinispan:server:15.0">

<data-sources>

<!-- Defines a unique name for the datasource and JNDI name that you

reference in JDBC cache store configuration.

Enables statistics for the datasource, if required. -->

<data-source name="ds"

jndi-name="jdbc/postgres"

statistics="true">

<!-- Specifies the JDBC driver that creates connections. -->

<connection-factory driver="org.postgresql.Driver"

url="jdbc:postgresql://localhost:5432/postgres"

username="postgres"

password="changeme">

<!-- Sets optional JDBC driver-specific connection properties. -->

<connection-property name="name">value</connection-property>

</connection-factory>

<!-- Defines connection pool tuning properties. -->

<connection-pool initial-size="1"

max-size="10"

min-size="3"

background-validation="1000"

idle-removal="1"

blocking-timeout="1000"

leak-detection="10000"/>

</data-source>

</data-sources>

</server>

##### Connection pool tuning properties

You can tune JDBC connection pools for managed datasources in your Infinispan Server configuration.

| **Property** | **Description** |
| --- | --- |
| initial-size | Initial number of connections the pool should hold. |
| max-size | Maximum number of connections in the pool. |
| min-size | Minimum number of connections the pool should hold. |
| blocking-timeout | Maximum time in milliseconds to block while waiting for a connection before throwing an exception. This will never throw an exception if creating a new connection takes an inordinately long period of time. Default is 0 meaning that a call will wait indefinitely. |
| background-validation | Time in milliseconds between background validation runs. A duration of 0 means that this feature is disabled. |
| validate-on-acquisition | Connections idle for longer than this time, specified in milliseconds, are validated before being acquired (foreground validation). A duration of 0 means that this feature is disabled. |
| idle-removal | Time in minutes a connection has to be idle before it can be removed. |
| leak-detection | Time in milliseconds a connection has to be held before a leak warning. |

After entering the command :

$ ./gradlew run

[ for vertx version 4.0.3 ] : The error message was seen below :

ERROR [vert.x-eventloop-thread-0] VertxImpl - Failed to initialize clustered Vert.x

org.infinispan.commons.CacheConfigurationException: ISPN000327: Cannot find a parser for element 'distributed-cache' in namespace ''. Check that your configuration is up-to date for Infinispan '11.0.5.Final' and if you have the proper dependency in the classpath

[ for vertx version 4.4.4 ] : The error message was seen below :

ERROR [vert.x-eventloop-thread-0] VertxImpl - Failed to initialize clustered Vert.x

org.infinispan.commons.CacheConfigurationException: ISPN000343: Must have a transport set in the global configuration in order to define a clustered cache

##### Distributed caches

XML

JSON

YAML

<distributed-cache owners="2"

segments="256"

capacity-factor="1.0"

l1-lifespan="5000"

mode="SYNC"

statistics="true">

<encoding media-type="application/x-protostream"/>

<locking isolation="REPEATABLE\_READ"/>

<transaction mode="FULL\_XA"

locking="OPTIMISTIC"/>

<expiration lifespan="5000"

max-idle="1000" />

<memory max-count="1000000"

when-full="REMOVE"/>

<indexing enabled="true"

storage="local-heap">

<index-reader refresh-interval="1000"/>

<indexed-entities>

<indexed-entity>org.infinispan.Person</indexed-entity>

</indexed-entities>

</indexing>

<partition-handling when-split="ALLOW\_READ\_WRITES"

merge-policy="PREFERRED\_NON\_NULL"/>

<persistence passivation="false">

<!-- Persistent storage configuration. -->

</persistence>

</distributed-cache>

##### Replicated caches

XML

JSON

YAML

<replicated-cache segments="256"

mode="SYNC"

statistics="true">

<encoding media-type="application/x-protostream"/>

<locking isolation="REPEATABLE\_READ"/>

<transaction mode="FULL\_XA"

locking="OPTIMISTIC"/>

<expiration lifespan="5000"

max-idle="1000" />

<memory max-count="1000000"

when-full="REMOVE"/>

<indexing enabled="true"

storage="local-heap">

<index-reader refresh-interval="1000"/>

<indexed-entities>

<indexed-entity>org.infinispan.Person</indexed-entity>

</indexed-entities>

</indexing>

<partition-handling when-split="ALLOW\_READ\_WRITES"

merge-policy="PREFERRED\_NON\_NULL"/>

<persistence passivation="false">

<!-- Persistent storage configuration. -->

</persistence>

</replicated-cache>

##### Multiple caches

XML

JSON

YAML

<infinispan

xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xsi:schemaLocation="urn:infinispan:config:15.0 https://infinispan.org/schemas/infinispan-config-15.0.xsd

urn:infinispan:server:15.0 https://infinispan.org/schemas/infinispan-server-15.0.xsd"

xmlns="urn:infinispan:config:15.0"

xmlns:server="urn:infinispan:server:15.0">

<cache-container name="default"

statistics="true">

<distributed-cache name="mycacheone"

mode="ASYNC"

statistics="true">

<encoding media-type="application/x-protostream"/>

<expiration lifespan="300000"/>

<memory max-size="400MB"

when-full="REMOVE"/>

</distributed-cache>

<distributed-cache name="mycachetwo"

mode="SYNC"

statistics="true">

<encoding media-type="application/x-protostream"/>

<expiration lifespan="300000"/>

<memory max-size="400MB"

when-full="REMOVE"/>

</distributed-cache>

</cache-container>

</infinispan>

After trying all possible ways of getting the infinispan right, but with no success, I rather resorted to what I do when I stay with a problem for just so so long and there is still no breakthrough : I ask myself : “Is this going to be needed in production ? and is it the best for the current challenge or problem I ultimately need to solve, globally ?

So I went on to do a quick search :

And in my discovery, I came to realise, I needed to leave the problem behind and rather solve a more relevant one : how to deploy an enterprise application or server solution with all the high standards and not just some fancy localhost or localside ephemeral solutions with infinispan.

Now the twist is that :

Both Hazelcast and Infinispan, by default use multicast communications to discover nodes. This is great for local testing and many bare-metal server deployments, but multicast communications are not possible in a Kubernetes

cluster. If you run the containers as is on Kubernetes, the heat sensor services and sensor gateway instances will not be able to communicate over the event bus.

These cluster managers can, of course, be configured to perform service discovery in Kubernetes. We will briefly cover the case of **Hazelcast**, where two discovery modes are possible:

 Hazelcast can connect to the Kubernetes API to listen for and discover pods matching a request, such as a desired label and value.

 Hazelcast can periodically make DNS queries to discover all pods for a given Kubernetes (headless) service.

The DNS approach is more limited.

Instead, let’s use the Kubernetes API and configure Hazelcast to use it. By default, the Hazelcast Vert.x cluster manager reads configuration from a cluster.xml resource. The following listing shows the relevant configuration excerpt of the heat-sensor-service/

src/main/resource/cluster.xml file.

2024/Nov/10th : successful live broadcasting using vertx.

Push and pull/fetch modes.

**Beyond callbacks :**

**Futures & Promises :**

@Override

public void start(Promise<Void> promise) {

vertx.createHttpServer()

.requestHandler(this::handleRequest)

.listen(8080) //returns a future <HttpServer>

.onFailure(promise::fail) //called when the server could not be started

.onSuccess(ok -> {//called on success

System.out.println("http://localhost:8080/");

promise.complete();

});

}

/\*

The listen method(s) or prior httpservers that we saw in earlier examples took a callback, and here it returns a Future<HttpServer>. We then can chain calls to onFailure and onSuccess to define what to do when the server starts, or when an error occurs.

The promise/future interfaces starting from Vert.x 3.8, but the future-based APIs are only available in Vert.x 4.

\*/

**Interoperability with CompletionStage APIs**

CompletionStage<String> cs = promise.future().toCompletionStage();//converts a future to a completionStage

cs

.thenApply(String::toUpperCase) //just like map in Future

.thenApply(str -> "~~~ " + str) //just like map in Future

.whenComplete((str, err) -> {//takes a value or error (ie. str or err)

if (err == null) {

System.out.println(str);

} else {

System.out.println("Oh... " + err.getMessage());

}

});

/\*

we convert the string result to uppercase, prefix it with a string, and eventually call

whenComplete. Note that this is a BiConsumer, and you need to test which of the values or exception parameters is null to know whether the promise completed successfully.

It is also important to note that unless you call an asynchronous CompletionStage

method, the calls are performed on a Vert.x thread.

\*/

**Convert a CompletionStage to a Future**

CompletableFuture<String> cf = CompletableFuture.supplyAsync(() -> {

try {

Thread.sleep(5000);

} catch (InterruptedException e) {

e.printStackTrace();

}

return "5 seconds have elapsed";

});

Future

.fromCompletionStage(cf, vertx.getOrCreateContext())

.onSuccess(System.out::println)

.onFailure(Throwable::printStackTrace);

**Reactive extensions**

Aka : Observable(s) design pattern

Reactive extensions are defined by three things:

 Observing event or data streams (e.g., an incoming HTTP request can be observed)

 Composing operators to transform streams (e.g., merge multiple HTTP request streams as one)

 Subscribing to streams and reacting to events and errors

RxJava v1.x.x does not support back-pressure

RxJava v2.x.x supports back-pressure

**Observable<T>**

* A stream of events of type T. Does not support back-pressure.
* Timer events, observable source where we cannot apply back-pressure like GUI events

**Flowable<T>**

* A stream of events of type T where back-pressure can be applied
* Network data, filesystem inputs

**Single<T>**

* A source that emits exactly one event of type T
* Fetching an entry from a data store by key

**Maybe<T>**

* A source that may emit one event of type T, or none
* Fetching an entry from a data store by key, but the key may not exist

**Completable**

* A source that notifies of some action having completed, but no value is being given
* Deleting files

**Hot Source :**

1. Would still emit event even when there are no subscribers.
2. The subscriber does not get all the events.
3. The subscriber only gets events from the time or period of subscription.
4. Example of hot source(ing) is a periodic timer.

**Cold Source :**

1. Would only emit event if there are subscribers.
2. The subscriber does get all the events.
3. The subscriber only gets events from the time or period of subscription.
4. Example of cold source(ing) is reading file content.

private Single<JsonObject> sendToSnapshot(Single<JsonObject> data) {

return data.flatMap(json -> webClient

.post(4000, "localhost", "")

.expect(ResponsePredicate.SC\_SUCCESS)

.rxSendJsonObject(json)

.flatMap(resp -> Single.just(json))); //and then…

}

**Coroutines**

**Edge Services using coroutines**

**Coroutine trivia :**

“as” is a keyword in Kotlin, so it has to be escaped when used as a method name.

**Choosing asynchronous model appropriate for your project :**

**Some major summary points :**

 Callbacks have expressiveness limitations in relation to composing asynchronous operations, and they can yield that is harder to understand if care is not taken.

 Parallel and sequential asynchronous operations can be composed with other (pure)

asynchronous programming models: futures and promises, reactive extensions,

and coroutines.

 Reactive extensions have an advanced set of composable operators very well suited for event streams.

 Futures and promises are perfect for simple chaining of asynchronous operations.

 Kotlin coroutines affords a language-level support for asynchronous operations.

 There is no universally perfect asynchronous programming model. The choice truly depends on the case of use of such async programming model. The exciting thing about Vert.x is that you can combine these models depending on the challenge or problem domains at hand.

**Beyond the event bus**

**Designing a reactive application**

**Criteria of a reactive application:**

 Back-pressure, as a necessary ingredient in asynchronous stream processing to regulate event throughput.

 Reactive programming as a way to compose asynchronous operations

**One application many services**

**Databases**

PostgresSQL

MongoDB

**Queue Management Services**

**SMTP**

**Event Stats Services**

**Running the application**

**The Web Stack**

1. The construction of an edge service and a public API
2. The Vert.x web client
3. JSON web tokens (JWT) and cross-origin resource sharing (CORS)
4. Serving and integrating a Vue.js reactive application with Vert.x
5. Testing an HTTP API with REST Assured

**Elements from the vertx web stack :**

Advanced routing

Routing with regex

Authentication

HTTP Client

Cookies

Server side session

Server side template rendering

Cross-site request forgery protection

**Modules required :**

Web (router, request handler, HTTP request dispatcher, handler : {

BodyHandler : {

HTTP authentication,

CORS,

CSRF,

Favicon,

HTTP session,

Serving static files,

Virtual Host,

Template rendering

}

})

Client

Auth-JWT

**Routing HTTP Requests**

BodyHandler bodyHandler = BodyHandler.create();// BodyHandler is a predefined handler that extracts HTTP request body payloads.

router.post().handler(bodyHandler);// Here bodyHandler is called for all HTTP POST and PUT requests.

router.put().handler(bodyHandler);

String prefix = "/api/v1";

router.post(prefix + "/register").handler(this::register);

router.post(prefix + "/token").handler(this::token);

// (...) defines jwtHandler, more later

router.get(prefix + "/:username/:year/:month")//We can extract path parameters by prefixing elements with ":".

.handler(jwtHandler) //Handlers can be chained

.handler(this::checkUser)

.handler(this::monthlySteps);

// (...)

**Making HTTP Requests**

**Creating a JWT Handler**

**JWT Handler in a route**

**Checking that a valid JWT token is present**

**Issuing JWT Tokens in vertx**

Working with JWTs

Common JWT libraries make working with JWTs easy. For example, with JJWT1, creating a

new JWT is straightforward, as shown in Example 6-1.

Example 6-1 Creating and validating a signed JWT using JJWT

import java.time.Instant;

import java.time.temporal.ChronoUnit;

import java.util.Date;

import io.jsonwebtoken.Claims;

import io.jsonwebtoken.Jws;

import io.jsonwebtoken.Jwts;

import io.jsonwebtoken.SignatureAlgorithm;

public class JwtExample {

public String createJwt(String secret) throws Exception {

// create and sign the JWT, including a hint

// for the key used to sign the request (kid)

String newJwt = Jwts.builder()

.setHeaderParam("kid", "meaningfulName")

.setSubject("user-12345")

.setAudience("user")

.setIssuedAt(Date.from(Instant.now()))

.setExpiration(Date.from(Instant.now().plus(15, ChronoUnit.MINUTES)))

.signWith(SignatureAlgorithm.HS512, secret)

.compact();

return newJwt;

}

public void validateJwt(String jwtParameter, String secret) throws Exception {

// Validate the Signed JWT!

// Exceptions thrown if not valid

Jws<Claims> jwt = Jwts.parser()

.setSigningKey(secret)

.parseClaimsJws(jwtParameter);

// Inspect the claims, like make a new JWT

// (need a signing key for this)

Claims jwtClaims = jwt.getBody();

System.out.println(jwtClaims.getAudience());

System.out.println(jwtClaims.getIssuer());

System.out.println(jwtClaims.getSubject());

System.out.println(jwtClaims.getExpiration());

System.out.println(jwtClaims.getIssuedAt());

System.out.println(jwtClaims.getNotBefore());

}

}

**Cross-Origin Resource Sharing ( C.O.R.S )**

In vertx Corsehandler we can specify :

1. Allowed Origin path
2. Allowed HTTP Header(s)
3. Allowed HTTP Method(s)

Example :

Set<String> allowedHeaders = new HashSet<>();

allowedHeaders.add("x-requested-with");

allowedHeaders.add("Access-Control-Allow-Origin");

allowedHeaders.add("origin");

allowedHeaders.add("Content-Type");

allowedHeaders.add("accept");

allowedHeaders.add("Authorization");

Set<HttpMethod> allowedMethods = new HashSet<>();

allowedMethods.add(HttpMethod.GET);

allowedMethods.add(HttpMethod.POST);

allowedMethods.add(HttpMethod.OPTIONS);

allowedMethods.add(HttpMethod.PUT);

router.route().handler(

CorsHandler

.create("\*")//A CORS handler for all routes

.allowedHeaders(allowedHeaders)

.allowedMethods(allowedMethods)

);

**Checking C.O.R.S support**

**A modern web frontend with vertx**

**A canvas of Vue.js components**

**VueJS application structure and build integration**

**VueJS & Vertx**

$ npm -version

$ brew upgrade npm

$ npm -version

$ sudo npm install -g @vue/cli

$ vue -version

$ cd <path-to-folder>

$ mkdir <name-of-project>

$ vue create <name-of-vue-project>

$ npm run serve

//sample output :

**➜ basic-vuejs-project** **git:(master) $** npm run serve

> basic-vuejs-project@0.1.0 serve

> vue-cli-service serve

INFO Starting development server...

DONE Compiled successfully in 2848ms 2024-11-25 8:43:06 AM

App running at:

- Local: http://localhost:**8080**/

- Network: http://192.000.111.222:**8080**/

Note that the development build is not optimized.

To create a production build, run

npm run build.

**VueJS + Vite + TailwindCSS**

**CLIENT SIDE LOGIC | JAVASCRIPT | PAGE(S) FLUIDITY**

Then input this into the terminal :

**npm init vite vue**

<press Enter>

? Select a framework: › - Use arrow-keys. Return to submit.

❯   Vanilla

    Vue

    React

    Preact

    Lit

    Svelte

    Solid

    Qwik

    Others

**Choose : Vue**

? Select a variant: › - Use arrow-keys. Return to submit.

❯   TypeScript

    JavaScript

    Customize with create-vue ↗

    Nuxt ↗

**Choose : Javascript**

Done. Now run:

**cd vue**

**npm install**

**npm run dev**

vue % npm install

vue % npm run dev

> vue@0.0.0 dev

> vite

  VITE v5.2.11  ready in 950 ms

  ➜  Local:   http://localhost:5173/

  ➜  Network: use --host to expose

  ➜  press h + enter to show help

h

  Shortcuts

  press r + enter to restart the server

  press u + enter to show server url

  press o + enter to open in browser

  press c + enter to clear console

  press q + enter to quit

u

  ➜  Local:   http://localhost:5173/

  ➜  Network: use --host to expose

o

vue % **npm install -S vuex@next**

or :

**npm install vue-router vuex axios**

Remember to remove the  ‘setup’ tag from the <script setup></script>

Now we can comment out the <pre>{{user}}</pre> in the html of the App.vue

**STYLING :**

Upnext , let’s install Tailwindcss, by following the instructions at :

[tailwindcss.com/docs/guides/vite](http://tailwindcss.com/docs/guides/vite)

Commands :

**npm install -D tailwindcss postcss autoprefixer**

<press Enter>

**npx tailwindcss init -p**

<press Enter>

**npm install @headlessui/vue @heroicons/vue @tailwindcss/forms -S**

<press Enter>

We can get UI components here :

<https://tailwindui.com/components>

LESSON(S) LEARNT :

1. All the npm commands and the installations relating to the tailwindcss should all be made while at the root folder path of “vue” within the project in the terminal for all the designs and CSSes to work perfectly.

All the tailwindcss things with the form design is perfectly working now as of 1119GMT.

1. In creating “vue-routes” do not put a route there when its components have not been created yet. Stick with what exist. The best practice may be to create the component before you ever call it anywhere-else.
2. The  value for component in the set of unique path(s) should match the name given to the import at the top of the line.

At the point of linking the JS and functions such as dispatch, then and so on with promises, it is therefore necessary to install “ axios “ using npm.

Command :

**$ npm i -S axios**

If we ever need icons we can check out heroicons :

<https://www.heroicons.com>

### process 2 :

**npm install**

**CLIENT SIDE LOGIC | JAVASCRIPT | PAGE(S) FLUIDITY**

Then input this into the terminal :

**npm init vite vue**

<press Enter>

? Select a framework: › - Use arrow-keys. Return to submit.

❯   Vanilla

    Vue

    React

    Preact

    Lit

    Svelte

    Solid

    Qwik

    Others

**Choose : Vue**

? Select a variant: › - Use arrow-keys. Return to submit.

❯   TypeScript

    JavaScript

    Customize with create-vue ↗

    Nuxt ↗

**Choose : Javascript**

Scaffolding project in

airsurvey\_app/vue...

Done. Now run:

***cd vue***

***npm install***

***npm run dev***

vue % **npm install**

The next command was to help get vite + vue setup nicely :

**npm install vue-router vuex axios**

After the command is issued, set up the styling, the root folder still set to “ vue “ :

**STYLING :**

Upnext , let’s install Tailwindcss, by following the instructions at :

[tailwindcss.com/docs/guides/vite](http://tailwindcss.com/docs/guides/vite)

Commands :

**npm install -D tailwindcss postcss autoprefixer**

<press Enter>

**npx tailwindcss init -p**

<press Enter>

**npm install @headlessui/vue @heroicons/vue @tailwindcss/forms -S**

<press Enter>

## Had to paste this code into the “ tailwind.config.js”

module.exports = {

content: [

'./index.html',

'./src/\*\*/\*.{vue,js,ts,jsx,tsx}',

'./pages/\*\*/\*.{html,js}',

'./components/\*\*/\*.{html,js}',

],

theme: {

extend: {},

},

plugins: [

require('@tailwindcss/forms')

],

}

Running the app :

**npm run dev**

**npm run watch**

**npm run test**

>>>> 2024/11/26 :

This time it took me far lesser time to setup a vue.js project from scratch … just a few minutes, then testing to make sure it all works.

**VueJS router configuration**

**Backend integration in VueJS**

**HTML Template components**

<https://tailwindtemplates.io/templates>

<https://github.com/PhatStraw/Free-Tailwind-CSS-Templates-and-Components>

<https://github.com/slim-python/tailwind-css-free-components>

<https://github.com/markmead/hyperui>

<https://tw-elements.com/docs/standard/integrations/vue-integration/>

<https://www.tailwind-kit.com/components#forms>

<https://tailwindflex.com/>

<https://flowbite.com/docs/components/alerts/>

<https://flowbite.com/docs/getting-started/vue/>

<https://tailwindui.com/components/preview>

<https://tailwindui.com/components>

**Javascript code components**

**Using JWT Token with axios**

<https://jwt.io>

**Serving static content with Vertx**

**Writing integration tests**

Some fundamental tests set in the IntegrationTest class :

1 Register some users.

2 Get a JWT token for each user.

3 Fetch a user’s data.

4 Try to fetch the data of another user.

5 Update a user’s data.

6 Check some activity stats for a user.

7 Try to check the activity of another user.

Test dependencies to run the integration tests

Preamble of the integration test class

Preparing a REST assured request specification

Utility hash maps for the integration testing

Test for registering users

Test code for retrieving JWT tokens

Extracting JSON with REST assured

\*\*\* Attempt to run the test(s) : 2024 Nov., 25th

Setting up docker for Mac M1 : {

* Docker Desktop is free for small businesses (fewer than 250 employees AND less than $10 million in annual revenue), personal use, education, and non-commercial open source projects.
* Otherwise, it requires a paid subscription for professional use.
* Paid subscriptions are also required for government entities.
* Docker Pro, Team, and Business subscriptions include commercial use of Docker Desktop.

}

Setting up podman for Mac M1

Setting up Colima for Mac M1

install colima, and the docker CLI. We then launch a VM using special configuration flags to use macOS’s virtualisation layer and the translation of x86/amd64 into Apple Silicon.

Colima is only a wrapper to create Lima VMs.

Lima is the virtual machine that will run with the rosetta enhaced compatibility with x86 / amd64 and provide the docker runtime.

brew install colima # we will create the lima vm with this wrapper  
brew install docker # The CLI only

# Create and Configure a super performant vm

Note: adjust your settings (CPU, Memory and Disk according to your needs and hardware)

colima start \  
--profile default \  
--activate \  
--arch aarch64 \  
--cpu 10 \  
--disk 48 \  
--memory 24 \  
--mount ${HOME}:w \  
--mount-inotify \  
--ssh-agent \  
--vm-type vz \  
--vz-rosetta \  
--verbose

# Key Configuration Settings

* **arch**: AARCH64 specifies that we will run an ARM64 machine and not an x86\_64 machine
* **vm-type**: VZ (to use Apple’s Hypervisor.Framework)
* **vz-rosseta**: Enables Rosetta (needs macOS 13.0 or newer)

Reference on how to use Rosetta with Lima and the compatibility modes:

## [Intel-on-ARM and ARM-on-Intel](https://lima-vm.io/docs/config/multi-arch/?source=post_page-----da5100e2557d--------------------------------" \l "slow-mode" \t "_blank)

### [Lima supports two modes for running Intel-on-ARM and ARM-on-Intel: Lima can run a VM with a foreign architecture, just…](https://lima-vm.io/docs/config/multi-arch/?source=post_page-----da5100e2557d--------------------------------" \l "slow-mode" \t "_blank)

[lima-vm.io](https://lima-vm.io/docs/config/multi-arch/?source=post_page-----da5100e2557d--------------------------------" \l "slow-mode" \t "_blank)

# Configure the Shell to replace Docker

Wait for the machine to come up and then let’s setup the docker environment. (Actually, none of this is strictly necessary. You could use the bundled nerdctl tool.)

But follow along if you want a drop-in Docker replacement:

* Place those in your shell’s profile or in the current session at will.

export COLIMA\_VM="default"  
export COLIMA\_VM\_SOCKET="${HOME}/.colima/${COLIMA\_VM}/docker.sock"  
export DOCKER\_HOST="unix://${COLIMA\_VM\_SOCKET}"

# Bonus: Multiple machines simultaneously

At a certain point you might want to run more experiments or even split work / load / whatever.

For example, one limitation of **using Apple’s Hypervisor is that it is not possible to resize the VM’s Disk after being created**. So instead of destroying the VM and recreating it (losing all the containers), you can sidekick another alongside and split the load on that one.

Podman does not officially support this kind of workloads, though it is still possible to achieve the same.

# To create a secondary machine, just do issue another colima command

colima start \  
--profile secondary \  
--activate \  
--arch aarch64 \  
--cpu 1 \  
--disk 20\  
--memory 8 \  
--mount ${HOME}:w \  
--mount-inotify \  
--ssh-agent \  
--vm-type vz \  
--vz-rosetta \  
--verbose

export COLIMA\_VM="secondary"  
export COLIMA\_VM\_SOCKET="${HOME}/.colima/${COLIMA\_VM}/docker.sock"  
export DOCKER\_HOST="unix://${COLIMA\_VM\_SOCKET}"

# Introduction

This post introduces a streamlined method to set up a Podman machine (QEMU) on **Apple Silicon** **for running amd64 (x86\_64) containers**. **We explore two approaches**: **multi-architecture support** and **fully emulated** x86\_64 machines.

**Note**: I will keep every script here and the explanation in this **public repo** too**:** <https://github.com/guillem-riera/podman-machine-x86_64>

# Approach Overview

1. **Mixed Mode, Multi-Architecture Support**: This method enables support for multiple architectures, including x86\_64, on a standard aarch64 machine. It **maintains high performance for the native ARM images and has a performance impact on amd64 images**. It operates on a base aarch64 machine and compatibility with amd64 images is as good as the package qemu-user-static can provide (I haven’t tested for full compatibility).
2. **Full x86\_64 Emulation**: This offers **maximum compatibility at the cost of slower performance**. It’s a fully emulated x86\_64 machine, which means that the containers are also run in fully x86\_64 mode.

**Recommendation**: Always try the **first approach (mixed mode)** before considering the second.

# Requirements

To get started, ensure you have the following installed:

* Homebrew
* Homebrew bundle
* Podman
* QEMU (automatically included as a dependency of Podman)
* jq

The required packages are listed in the Brewfile. Install them using:

brew bundle install

# Setting Up

# 1. Multi-Arch Support on Current Podman Machine

This setup installs the necessary package **qemu-user-static** on your current machine.

This script facilitates this process:

export PODMAN\_MACHINE\_NAME=${PODMAN\_MACHINE\_NAME:-podman-machine-default}  
  
### Stop all podman machine instances  
ALL\_PODMAN\_MACHINES=$(podman machine list | awk '{ print $1 }' | tr -d '\*' | sed 1d | tr '\n' ' ')  
for PODMAN\_MACHINE in ${ALL\_PODMAN\_MACHINES}; do  
 podman machine stop ${PODMAN\_MACHINE}  
done  
  
### Start the target podman machine  
podman machine start ${PODMAN\_MACHINE\_NAME}  
  
### wait for the podman machine to be running  
PODMAN\_MACHINE\_STATUS=$(podman machine inspect ${PODMAN\_MACHINE\_NAME} | jq -r '.[].State')  
while [[ "${PODMAN\_MACHINE\_STATUS}" != "running" ]]; do  
 echo "[Info] Waiting for podman machine '${PODMAN\_MACHINE\_NAME}' to be running, current status: ${PODMAN\_MACHINE\_STATUS}..."  
 sleep 1  
 PODMAN\_MACHINE\_STATUS=$(podman machine inspect ${PODMAN\_MACHINE\_NAME} | jq -r '.[].State')  
done  
  
### Now that the podman machine is running we can install the package  
podman machine ssh "${PODMAN\_MACHINE\_NAME}" 'sudo rpm-ostree install qemu-user-static'  
  
### Stop the podman machine to apply the changes  
podman machine stop ${PODMAN\_MACHINE\_NAME}  
  
### Start the podman machine again  
podman machine start ${PODMAN\_MACHINE\_NAME}  
  
echo "[Info] Done. You can now run multi-architecture images in ${PODMAN\_MACHINE\_NAME}."

Podman can now run multi-architecture images with performance impacts limited to x86\_64 containers.

# How it works?

This bash script automates the setup of multi-architecture support for an existing Podman machine. Here’s a summary of how it works:

1. **Setup**: It sets the PODMAN\_MACHINE\_NAME variable, defaulting to "podman-machine-default" if not already specified.
2. **Stopping All Podman Machine Instances**: The script lists all existing Podman machines, excluding the header line and any active (marked with an asterisk) machines. It then stops each of these machines to ensure a clean setup environment.
3. **Starting the Target Podman Machine**: It starts the target Podman machine specified in PODMAN\_MACHINE\_NAME.
4. **Waiting for the Machine to Run**: The script continuously checks if the target Podman machine has reached the “running” state. It waits in a loop, checking the machine’s status every second.
5. **Installing the Package**: Once the target machine is running, the script remotely connects to it via SSH and installs the qemu-user-static package using sudo rpm-ostree install. This package is crucial for enabling multi-architecture support.
6. **Restarting the Podman Machine**: After the installation, the script stops the Podman machine to apply the changes and then starts it again

# 2. Full x86\_64 Emulation Setup

**Note**: Follow this step only if the first solution doesn’t meet your needs.

## Creating a new emulated Podman Machine (x86\_64)

The following script creates a podman machine and alters it to make it an x86\_64 machine (using QEMU):

# Setup the podman machine for x86\_64 (QEMU), supports only Apple Silicon (Mx) Macs  
  
# Keep all shell arguments in a variable to pass to the podman machine init command:  
EXTRA\_ARGS=${EXTRA\_ARGS:-$@}  
  
## 1. Download Fedora CoreOS image for x86\_64 (QEMU)  
PODMAN\_X86\_64\_MACHINE\_NAME=${PODMAN\_X86\_64\_MACHINE\_NAME:-x86\_64}  
PODMAN\_X86\_64\_MACHINE\_NAME\_EXISTS=$(podman machine list | grep ${PODMAN\_X86\_64\_MACHINE\_NAME} | wc -l | tr -d '[:space:]')  
PODMAN\_QEMU\_IMAGE="fedora-coreos-39.20231101.3.0-qemu.x86\_64.qcow2.xz"  
DOWNLOAD\_DIR=${DOWNLOAD\_DIR:-.}  
  
if [ ${PODMAN\_X86\_64\_MACHINE\_NAME\_EXISTS} -lt 1 ]; then  
 curl -C- -O "https://builds.coreos.fedoraproject.org/prod/streams/stable/builds/39.20231101.3.0/x86\_64/${PODMAN\_QEMU\_IMAGE}"  
 podman machine init --image-path ${DOWNLOAD\_DIR}/${PODMAN\_QEMU\_IMAGE} ${PODMAN\_X86\_64\_MACHINE\_NAME} ${EXTRA\_ARGS}  
else  
 echo "[Info] Machine ${PODMAN\_X86\_64\_MACHINE\_NAME} already exists. If you want to recreate it, run 'podman machine rm ${PODMAN\_X86\_64\_MACHINE\_NAME}'"  
fi  
  
## 2. Change machine settings  
  
### Get the machine config file name  
machineConfigFile="$(podman machine inspect ${PODMAN\_X86\_64\_MACHINE\_NAME} | jq -r '.[].ConfigPath.Path')"  
  
### Change the QEMU binary to x86\_64  
sed -i '' 's/qemu-system-aarch64/qemu-system-x86\_64/g' ${machineConfigFile}  
### Change the firmware to x86\_64  
sed -i '' 's/edk2-aarch64-code/edk2-x86\_64-code/g' ${machineConfigFile}  
### Delete the additional UEFI firmware file (ovmf) and the preceding '-drive' option. The '-drive' option is in a line above the line containing the path to 'x86\_64\_ovmf\_vars.fd'. Both lines must be deleted, but other -drive options must be kept.  
#### using sed to match 2 lines: '-drive' followed by 'x86\_64\_ovmf\_vars.fd'  
sed -i '' '/-drive/{N;/x86\_64\_ovmf\_vars.fd/d;}' ${machineConfigFile}  
### Delete the HVF (Hypervisor Framework) acceleration, which is only available for macOS. This are also 2 lines: '-accel' followed by 'hvf'  
sed -i '' '/-accel/{N;/hvf/d;}' ${machineConfigFile}  
### Delete the TCG acceleration, which seems to work only for Alpha and ARM architectures. This are also 2 lines: '-accel' followed by 'tcg'  
sed -i '' '/-accel/{N;/tcg/d;}' ${machineConfigFile}  
### Change the machine type to q35  
sed -i '' 's/virt,highmem=on/q35/g' ${machineConfigFile}  
### Change the cpu type from 'host' to 'qemu64'  
sed -i '' 's/host/qemu64/g' ${machineConfigFile}

# How it works?

This script is designed to set up a Podman machine specifically for x86\_64 architecture on Apple Silicon (Mx) Macs by modifying the QEMU template that podman generates when it creates a new machine.

Here’s a summary of its functionality and workflow:

**Shell Arguments**: The script stores any arguments passed to it in the EXTRA\_ARGS variable, which will later be used in the podman machine init command.

**Downloading** Fedora CoreOS Image for x86\_64 (QEMU):

* It sets a default name for the Podman x86\_64 machine (PODMAN\_X86\_64\_MACHINE\_NAME) and checks if a machine with this name already exists.
* If the machine does not exist, the script downloads the specified Fedora CoreOS image for x86\_64 using curl.
* After downloading, it initializes a new Podman machine with this image and any extra arguments provided.

**Changing Machine Settings**:

* The script retrieves the configuration file path of the newly created Podman machine.
* Several modifications are made to the machine’s configuration file to adapt it for x86\_64 emulation:
* **Changing QEMU Binary**: Updates the QEMU binary from qemu-system-aarch64 to qemu-system-x86\_64.
* **Changing Firmware**: Adjusts the firmware from edk2-aarch64-code to edk2-x86\_64-code.
* **Removing UEFI Firmware File**: Deletes lines related to the UEFI firmware file (x86\_64\_ovmf\_vars.fd) and its preceding '-drive' option.
* **Removing HVF Acceleration**: Eliminates the Hypervisor Framework (HVF) acceleration settings, as they are only available for macOS.
* **Removing TCG Acceleration**: Removes TCG acceleration settings, which are typically for Alpha and ARM architectures.
* **Changing Machine Type**: Updates the machine type from virt,highmem=on to q35.
* **Changing CPU Type**: Changes the CPU type from host to qemu64.

## Conclusion

The podman offers a convenient way to run x86\_64 containers on Apple Silicon, but you have to do extra steps to enable that.

Whether you require high performance or maximum compatibility, these methods provide a flexible solution to meet your containerization needs.

This is possible because QEMU, the underlaying virtualisation and emulation tool is really awesome!.

# Alternatives

If you are looking for a very high performance and fully open source alternative to Docker Desktop that supports x86\_64 / amd64 architecture with Rosetta, check my newer post on colima:

<https://github.com/guillem-riera/podman-machine-x86_64>

**Messaging and event streaming with Vertx**

Messaging with AMQP

Event streaming with Apache Kafka

Examples of messaging queue brokers :

* 1. AMQP : Advanced Messaging Queuing Protocol
  2. STOMP : Simple Text Oriented Messaging Protocol
  3. RabbitMQ Client
  4. MQTT : Messaging Queuing Telemetric Transport



Sending emails

Integration testing with messaging and event streaming

middleware

Event-Bus TCP Bridge

**Event driven services beyond HTTP with vertx**

**Ingestion from AMQP**

AMQP client configuration :

private AmqpClientOptions amqpConfig() {

return new AmqpClientOptions()

.setHost("localhost")//Credentials are the default ones from the docker image

.setPort(5672) //Credentials are the default ones from the docker image

.setUsername("artemis")//Credentials are the default ones from the docker image

.setPassword("simetraehcapa");//Credentials are the default ones from the docker image

}

// (...)

AmqpClientOptions amqpOptions = amqpConfig();

AmqpReceiverOptions receiverOptions = new AmqpReceiverOptions()

.setAutoAcknowledgement(false) //We would manually acknowledge incoming messages

.setDurable(true); //We want durable messages

AmqpClient.create(vertx, amqpOptions) //Create an AMQP client

.rxConnect()

.flatMap(conn -> conn.rxCreateReceiver("step-events", receiverOptions)) //Create a message receiver from the ‘step-events’ destination

.flatMapPublisher(AmqpReceiver::toFlowable) //Create a flowable AMQP messages

.doOnError(this::logAmqpError) //Error logging

.retryWhen(this::retryLater) //Retry logic

.subscribe(this::handleAmqpMessage); //Subscription that dispatches incoming messages

**May be just IDE trivial :**

So in the way I like to use my IDE (eg. InteliJ) is fundamentally that I have taken time to define different types of elements in the said programming language by certain specific colours. Now with all of the above I shared so far on my progress with Vert.x, the colours were not coming up . . . everything was just plain white.



In order to get the colours to show, after trying to set the project properties, or try to set the compiler levels and or types, none of those helped until , I added a pom to the root of my project and defined the respective modules by name. After adding the pom file to identifying each of the elements as expected. the root, just like magic, the IDE suddenly seems to have recovered and then started

