Reactor Netty Reference Guide

Stephane Maldini,Violeta Georgieva

Version 1.2.0-M2, 2024-05-14

Table of Contents

Chapter 1. About the Documentation

Stephane Maldini <@smaldini>; Violeta Georgieva <@violeta_g_g> Version 1.2.0-M2

This section provides a brief overview of `Reactor Netty` reference documentation. You do not need to read this guide in a linear fashion. Each piece stands on its own, though they often refer to other pieces.

Latest Version and Copyright Notice

The `Reactor Netty` reference guide is available as `HTML` documents. The latest copy is available at projectreactor.io/docs/netty/release/reference/index.html

Contributing to the Documentation

The reference guide is written in Asciidoc using Antora, and you can find its sources at github.com/reactor/reactor-netty/tree/main/docs.

If you have an improvement, we will be happy to get a pull request from you!

We recommend that you check out a local copy of the repository so that you can generate the documentation by using the `asciidoctor` Gradle task and checking the rendering. Some of the sections rely on included files, so `GitHub` rendering is not always complete.

> To facilitate documentation edits, you can edit the current page from the `Edit this Page` link located in the upper right corner sidebar. The link opens an edit `UI` directly on `GitHub` for the main source file for the current page. These links are only present in the `HTML5` version of this reference guide. They look like the following link: Edit this Page to About the Documentation.

Getting Help

There are several ways to reach out for help with `Reactor Netty`. You can:

- Get in touch with the community on Gitter.
- Ask a question on stackoverflow.com at `reactor-netty`.
- Report bugs in `Github` issues. The repository is the following: reactor-netty.

> All of `Reactor Netty` is open source, including this documentation.

Chapter 2. Getting Started

This section contains information that should help you get going with `Reactor Netty`. It includes the following information:

- Introducing Reactor Netty
- Prerequisites
- Understanding the BOM and versioning scheme
- Getting Reactor Netty

Introducing Reactor Netty

Suited for Microservices Architecture, `Reactor Netty` offers backpressure-ready network engines for `HTTP` (including Websockets), `TCP`, and `UDP`.

Prerequisites

`Reactor Netty` runs on `Java 8` and above.

It has transitive dependencies on:

- Reactive Streams v1.0.4
- Reactor Core v3.x
- Netty v4.1.x

Understanding the BOM and versioning scheme

`Reactor Netty` is part of the `Project Reactor BOM` (since the `Aluminium` release train). This curated list groups artifacts that are meant to work well together, providing the relevant versions despite potentially divergent versioning schemes in these artifacts.

> The versioning scheme has changed between 0.9.x and 1.0.x (Dysprosium and Europium).

Artifacts follow a versioning scheme of `MAJOR.MINOR.PATCH-QUALIFIER` while the BOM is versioned using a CalVer inspired scheme of `YYYY.MINOR.PATCH-QUALIFIER`, where:

- `MAJOR` is the current generation of Reactor, where each new generation can bring fundamental changes to the structure of the project (which might imply a more significant migration effort)
- `YYYY` is the year of the first GA release in a given release cycle (like 1.0.0 for 1.0.x)
- `.MINOR` is a 0-based number incrementing with each new release cycle
  - in the case of projects, it generally reflects wider changes and can indicate a moderate migration effort
  - in the case of the BOM it allows discerning between release cycles in case two get first

released the same year

- .PATCH is a 0-based number incrementing with each service release
- -QUALIFIER is a textual qualifier, which is omitted in the case of GA releases (see below)

The first release cycle to follow that convention is thus 2020.0.x, codename Europium. The scheme uses the following qualifiers (note the use of dash separator), in order:

- -M1..-M9: milestones (we don't expect more than 9 per service release)
- -RC1..-RC9: release candidates (we don't expect more than 9 per service release)
- -SNAPSHOT: snapshots
- *no qualifier* for GA releases

> Snapshots appear higher in the order above because, conceptually, they're always "the freshest pre-release" of any given PATCH. Even though the first deployed artifact of a PATCH cycle will always be a -SNAPSHOT, a similarly named but more up-to-date snapshot would also get released after eg. a milestone or between release candidates.

Each release cycle is also given a codename, in continuity with the previous codename-based scheme, which can be used to reference it more informally (like in discussions, blog posts, etc…). The codenames represent what would traditionally be the MAJOR.MINOR number. They (mostly) come from the Periodic Table of Elements, in increasing alphabetical order.

> Up until Dysprosium, the BOM was versioned using a release train scheme with a codename followed by a qualifier, and the qualifiers were slightly different. For example: Aluminium-RELEASE (first GA release, would now be something like YYYY.0.0), Bismuth-M1, Californium-SR1 (service release would now be something like YYYY.0.1), Dysprosium-RC1, Dysprosium-BUILD-SNAPSHOT (after each patch, we'd go back to the same snapshot version. would now be something like YYYY.0.X-SNAPSHOT so we get 1 snapshot per PATCH)

Getting Reactor Netty

As mentioned earlier, the easiest way to use Reactor Netty in your core is to use the BOM and add the relevant dependencies to your project. Note that, when adding such a dependency, you must omit the version so that the version gets picked up from the BOM.

However, if you want to force the use of a specific artifact's version, you can specify it when adding your dependency as you usually would. You can also forego the BOM entirely and specify dependencies by their artifact versions.

Maven Installation

The BOM concept is natively supported by Maven. First, you need to import the BOM by adding the following snippet to your pom.xml. If the top section (dependencyManagement) already exists in your pom, add only the contents.

```xml
<dependencyManagement> ①
    <dependencies>
        <dependency>
            <groupId>io.projectreactor</groupId>
            <artifactId>reactor-bom</artifactId>
            <version>2024.0.0-M2</version> ②
            <type>pom</type>
            <scope>import</scope>
        </dependency>
    </dependencies>
</dependencyManagement>
```

① Notice the `dependencyManagement` tag. This is in addition to the regular `dependencies` section.

② As of this writing, `2024.0.0-M2` is the latest version of the `BOM`. Check for updates at github.com/reactor/reactor/releases.

Next, add your dependencies to the relevant reactor projects, as usual (except without a `<version>`). The following listing shows how to do so:

```xml
<dependencies>
    <dependency>
        <groupId>io.projectreactor.netty</groupId>
        <artifactId>reactor-netty-core</artifactId> ①
②
    </dependency>
</dependencies>
<dependencies>
    <dependency>
        <groupId>io.projectreactor.netty</groupId>
        <artifactId>reactor-netty-http</artifactId>
    </dependency>
</dependencies>
```

① Dependency on `Reactor Netty`

② No version tag here

Gradle Installation

The `BOM` concept is supported in Gradle since version 5. The following listing shows how to import the `BOM` and add a dependency to `Reactor Netty`:

```
dependencies {
    // import a BOM
    implementation platform('io.projectreactor:reactor-bom:2024.0.0-M2') ①

    // define dependencies without versions
    implementation 'io.projectreactor.netty:reactor-netty-core' ②
```

```
    implementation 'io.projectreactor.netty:reactor-netty-http'
}
```

① As of this writing, `2024.0.0-M2` is the latest version of the `BOM`. Check for updates at github.com/reactor/reactor/releases.

② There is no third `:` separated section for the version. It is taken from the `BOM`.

Milestones and Snapshots

Milestones and developer previews are distributed through the `Spring Milestones` repository rather than `Maven Central`. To add it to your build configuration file, use the following snippet:

Milestones in Maven

```
<repositories>
    <repository>
        <id>spring-milestones</id>
        <name>Spring Milestones Repository</name>
        <url>https://repo.spring.io/milestone</url>
    </repository>
</repositories>
```

For Gradle, use the following snippet:

Milestones in Gradle

```
repositories {
  maven { url 'https://repo.spring.io/milestone' }
  mavenCentral()
}
```

Similarly, snapshots are also available in a separate dedicated repository (for both Maven and Gradle):

-SNAPSHOTs in Maven

```
<repositories>
    <repository>
        <id>spring-snapshots</id>
        <name>Spring Snapshot Repository</name>
        <url>https://repo.spring.io/snapshot</url>
    </repository>
</repositories>
```

-SNAPSHOTs in Gradle

```
repositories {
  maven { url 'https://repo.spring.io/snapshot' }
  mavenCentral()
```

```
}
```

Support and policies

The entries below are mirroring github.com/reactor/.github/blob/main/SUPPORT.adoc

Do you have a question?

> Search Stack Overflow first; discuss if necessary

If you're unsure why something isn't working or wondering if there is a better way of doing it please check on **Stack Overflow** first and if necessary start a discussion. Use relevant tags among the ones we monitor for that purpose:

- `reactor-netty` for specific reactor-netty questions
- `project-reactor` for generic reactor questions

If you prefer real-time discussion, we also have a few **Gitter channels**:

- `reactor` is the historic most active one, where most of the community can help
- `reactor-core` is intended for more advanced pinpointed discussions around the inner workings of the library
- `reactor-netty` is intended for netty-specific questions

Refer to each project's README for potential other sources of information.

We generally discourage opening GitHub issues for questions, in favor of the two channels above.

Our policy on deprecations

When dealing with deprecations, given a version `A.B.C`, we'll ensure that:

- deprecations introduced in version `A.B.0` will be removed **no sooner than** version `A.B+1.0`
- deprecations introduced in version `A.B.1+` will be removed **no sooner than** version `A.B+2.0`
- we'll strive to mention the following in the deprecation javadoc:
    - target minimum version for removal
    - pointers to replacements for the deprecated method
    - version in which method was deprecated

> This policy is officially in effect as of January 2021, for all modules in `2020.0` BOMs and newer release trains, as well as `Dysprosium` releases after `Dysprosium-SR15`.

> Deprecation removal targets are not a hard commitment, and the deprecated methods **could live on further than these minimum target GA versions** (ie. only

the most problematic deprecated methods will be removed aggressively).

That said, deprecated code that has outlived its minimum removal target version may be removed in any subsequent release (including patch releases, aka service releases) without further notice. So users should still strive to update their code as early as possible.

Chapter 3. TCP Server

Reactor Netty provides an easy to use and configure TcpServer. It hides most of the Netty functionality that is needed to create a TCP server and adds Reactive Streams backpressure.

Starting and Stopping

To start a TCP server, you must create and configure a TcpServer instance. By default, the host is configured for any local address, and the system picks up an ephemeral port when the bind operation is invoked. The following example shows how to create and configure a TcpServer instance:

github.com/reactor/reactor-netty/tree/main/reactor-netty-examples/src/main/java/reactor/netty/examples/documentation/tcp/server/create/Application.java

```
import reactor.netty.DisposableServer;
import reactor.netty.tcp.TcpServer;

public class Application {

    public static void main(String[] args) {
        DisposableServer server =
                TcpServer.create()   ①
                         .bindNow(); ②

        server.onDispose()
              .block();
    }
}
```

① Creates a TcpServer instance that is ready for configuring.

② Starts the server in a blocking fashion and waits for it to finish initializing.

The returned DisposableServer offers a simple server API, including disposeNow(), which shuts the server down in a blocking fashion.

Host and Port

To serve on a specific host and port, you can apply the following configuration to the TCP server:

github.com/reactor/reactor-netty/tree/main/reactor-netty-examples/src/main/java/reactor/netty/examples/documentation/tcp/server/address/Application.java

```java
import reactor.netty.DisposableServer;
import reactor.netty.tcp.TcpServer;

public class Application {

    public static void main(String[] args) {
        DisposableServer server =
                TcpServer.create()
                         .host("localhost") ①
                         .port(8080)        ②
                         .bindNow();

        server.onDispose()
              .block();
    }
}
```

① Configures the `TCP` server host

② Configures the `TCP` server port

To serve on multiple addresses, after having configured the `TcpServer` you can bind it multiple times to obtain separate `DisposableServer`s. All created servers will share resources such as `LoopResources` because they use the same configuration instance under the hood.

github.com/reactor/reactor-netty/tree/main/reactor-netty-examples/src/main/java/reactor/netty/examples/documentation/tcp/server/address/MultiAddressApplication.java

```java
import reactor.core.publisher.Mono;
import reactor.netty.DisposableServer;
import reactor.netty.tcp.TcpServer;

public class MultiAddressApplication {

    public static void main(String[] args) {
        TcpServer tcpServer = TcpServer.create();
        DisposableServer server1 = tcpServer
                .host("localhost") ①
                .port(8080)        ②
                .bindNow();

        DisposableServer server2 = tcpServer
                .host("0.0.0.0") ③
                .port(8081)      ④
                .bindNow();

        Mono.when(server1.onDispose(), server2.onDispose())
                .block();
    }
}
```

① Configures the first `TCP` server host

② Configures the first `TCP` server port

③ Configures the second `TCP` server host

④ Configures the second `TCP` server port

Eager Initialization

By default, the initialization of the `TcpServer` resources happens on demand. This means that the `bind operation` absorbs the extra time needed to initialize and load:

- the event loop groups
- the native transport libraries (when native transport is used)
- the native libraries for the security (in case of `OpenSsl`)

When you need to preload these resources, you can configure the `TcpServer` as follows:

github.com/reactor/reactor-netty/tree/main/reactor-netty-examples/src/main/java/reactor/netty/examples/documentation/tcp/server/warmup/Application.java

```java
import reactor.netty.DisposableServer;
import reactor.netty.tcp.TcpServer;

public class Application {

    public static void main(String[] args) {
        TcpServer tcpServer =
                TcpServer.create()
                         .handle((inbound, outbound) -> inbound.receive().then());

        tcpServer.warmup() ①
                 .block();

        DisposableServer server = tcpServer.bindNow();

        server.onDispose()
              .block();
    }
}
```

① Initialize and load the event loop groups, the native transport libraries and the native libraries for the security

Writing Data

In order to send data to a connected client, you must attach an I/O handler. The I/O handler has access to `NettyOutbound` to be able to write data. The following example shows how to attach an I/O handler:

github.com/reactor/reactor-netty/tree/main/reactor-netty-examples/src/main/java/reactor/netty/examples/documentation/tcp/server/send/Application.java

```java
import reactor.core.publisher.Mono;
import reactor.netty.DisposableServer;
import reactor.netty.tcp.TcpServer;

public class Application {

    public static void main(String[] args) {
        DisposableServer server =
                TcpServer.create()
                         .handle((inbound, outbound) -> outbound.sendString(Mono.just
("hello"))) ①
                         .bindNow();

        server.onDispose()
              .block();
    }
}
```

① Sends `hello` string to the connected clients

Consuming Data

In order to receive data from a connected client, you must attach an I/O handler. The I/O handler has access to `NettyInbound` to be able to read data. The following example shows how to use it:

github.com/reactor/reactor-netty/tree/main/reactor-netty-examples/src/main/java/reactor/netty/examples/documentation/tcp/server/read/Application.java

```java
import reactor.netty.DisposableServer;
import reactor.netty.tcp.TcpServer;

public class Application {

    public static void main(String[] args) {
        DisposableServer server =
                TcpServer.create()
                         .handle((inbound, outbound) -> inbound.receive().then()) ①
                         .bindNow();

        server.onDispose()
              .block();
    }
}
```

① Receives data from the connected clients

Lifecycle Callbacks

The following lifecycle callbacks are provided to let you extend the `TcpServer`:

Callback	Description
`doOnBind`	Invoked when the server channel is about to bind.
`doOnBound`	Invoked when the server channel is bound.
`doOnChannelInit`	Invoked when initializing the channel.
`doOnConnection`	Invoked when a remote client is connected
`doOnUnbound`	Invoked when the server channel is unbound.

The following example uses the `doOnConnection` and `doOnChannelInit` callbacks:

github.com/reactor/reactor-netty/tree/main/reactor-netty-examples/src/main/java/reactor/netty/examples/documentation/tcp/server/lifecycle/Application.java

```
import io.netty.handler.logging.LoggingHandler;
import io.netty.handler.timeout.ReadTimeoutHandler;
import reactor.netty.DisposableServer;
import reactor.netty.tcp.TcpServer;
import java.util.concurrent.TimeUnit;

public class Application {

    public static void main(String[] args) {
        DisposableServer server =
                TcpServer.create()
                         .doOnConnection(conn ->
                             conn.addHandlerFirst(new ReadTimeoutHandler(10, TimeUnit
.SECONDS))) ①
                         .doOnChannelInit((observer, channel, remoteAddress) ->
                             channel.pipeline()
                                    .addFirst(new LoggingHandler
("reactor.netty.examples")))     ②
                         .bindNow();

        server.onDispose()
              .block();
    }
}
```

① `Netty` pipeline is extended with `ReadTimeoutHandler` when a remote client is connected.

② `Netty` pipeline is extended with `LoggingHandler` when initializing the channel.

TCP-level Configurations

This section describes three kinds of configuration that you can use at the TCP level:

- Setting Channel Options
- Wire Logger
- Event Loop Group

Setting Channel Options

By default, the `TCP` server is configured with the following options:

github.com/reactor/reactor-netty/tree/main/reactor-netty-core/src/main/java/reactor/netty/tcp/TcpServerBind.java

```
    TcpServerBind() {
        Map<ChannelOption<?>, Boolean> childOptions = new HashMap<>(MapUtils
.calculateInitialCapacity(2));
        childOptions.put(ChannelOption.AUTO_READ, false);
        childOptions.put(ChannelOption.TCP_NODELAY, true);
        this.config = new TcpServerConfig(
                Collections.singletonMap(ChannelOption.SO_REUSEADDR, true),
                childOptions,
                () -> new InetSocketAddress(DEFAULT_PORT));
    }
```

If additional options are necessary or changes to the current options are needed, you can apply the following configuration:

github.com/reactor/reactor-netty/tree/main/reactor-netty-examples/src/main/java/reactor/netty/examples/documentation/tcp/server/channeloptions/Application.java

```java
import io.netty.channel.ChannelOption;
import reactor.netty.DisposableServer;
import reactor.netty.tcp.TcpServer;

public class Application {

    public static void main(String[] args) {
        DisposableServer server =
                TcpServer.create()
                         .option(ChannelOption.CONNECT_TIMEOUT_MILLIS, 10000)
                         .bindNow();

        server.onDispose()
              .block();
    }
}
```

You can find more about `Netty` channel options at the following links:

- Common ChannelOption
- Epoll ChannelOption
- KQueue ChannelOption
- Socket Options

Wire Logger

Reactor Netty provides wire logging for when the traffic between the peers needs to be inspected. By default, wire logging is disabled. To enable it, you must set the logger `reactor.netty.tcp.TcpServer` level to `DEBUG` and apply the following configuration:

github.com/reactor/reactor-netty/tree/main/reactor-netty-examples/src/main/java/reactor/netty/examples/documentation/tcp/server/wiretap/Application.java

```java
import reactor.netty.DisposableServer;
import reactor.netty.tcp.TcpServer;

public class Application {

    public static void main(String[] args) {
        DisposableServer server =
                TcpServer.create()
                         .wiretap(true) ①
                         .bindNow();

        server.onDispose()
              .block();
    }
}
```

① Enables the wire logging

Wire Logger formatters

Reactor Netty supports 3 different formatters:

- AdvancedByteBufFormat#HEX_DUMP - the default

github.com/reactor/reactor-netty/tree/main/reactor-netty-core/src/main/java/reactor/netty/transport/logging/AdvancedByteBufFormat.java

```
    /**
     * When wire logging is enabled with this format, both events and content will be
logged.
     * The content will be in hex format.
     * <p>Examples:</p>
     * <pre>
     * {@code
     * reactor.netty.http.HttpTests - [d5230a14, L:/0:0:0:0:0:0:0:1:60267 -
R:/0:0:0:0:0:0:0:1:60269] REGISTERED
     * reactor.netty.http.HttpTests - [d5230a14, L:/0:0:0:0:0:0:0:1:60267 -
R:/0:0:0:0:0:0:0:1:60269] ACTIVE
     * reactor.netty.http.HttpTests - [d5230a14, L:/0:0:0:0:0:0:0:1:60267 -
R:/0:0:0:0:0:0:0:1:60269] READ: 145B
     *          +-------------------------------------------------+
     *          |  0  1  2  3  4  5  6  7  8  9  a  b  c  d  e  f |
     * +--------+-------------------------------------------------+----------------+
     * |00000000| 50 4f 53 54 20 2f 74 65 73 74 2f 57 6f 72 6c 64 |POST /test/World|
     * |00000010| 20 48 54 54 50 2f 31 2e 31 0d 0a 43 6f 6e 74 65 | HTTP/1.1..Conte|
     * |00000020| 6e 74 2d 54 79 70 65 3a 20 74 65 78 74 2f 70 6c |nt-Type: text/pl|
     * |00000030| 61 69 6e 0d 0a 75 73 65 72 2d 61 67 65 6e 74 3a |ain..user-agent:|
     * |00000040| 20 52 65 61 63 74 6f 72 4e 65 74 74 79 2f 64 65 | ReactorNetty/de|
     * ...
     * reactor.netty.http.HttpTests - [d5230a14, L:/0:0:0:0:0:0:0:1:60267 -
R:/0:0:0:0:0:0:0:1:60269] WRITE: 38B
     *          +-------------------------------------------------+
     *          |  0  1  2  3  4  5  6  7  8  9  a  b  c  d  e  f |
     * +--------+-------------------------------------------------+----------------+
     * |00000000| 48 54 54 50 2f 31 2e 31 20 32 30 30 20 4f 4b 0d |HTTP/1.1 200 OK.|
     * |00000010| 0a 63 6f 6e 74 65 6e 74 2d 6c 65 6e 67 74 68 3a |.content-length:|
     * |00000020| 20 30 0d 0a 0d 0a                               | 0....          |
     * +--------+-------------------------------------------------+----------------+
     * }
     * </pre>
     */
```

- AdvancedByteBufFormat#SIMPLE

github.com/reactor/reactor-netty/tree/main/reactor-netty-core/src/main/java/reactor/netty/transport/logging/AdvancedByteBufFormat.java

```
    /**
     * When wire logging is enabled with this format, only the events will be logged.
     * <p>Examples:</p>
     * <pre>
     * {@code
     * reactor.netty.http.HttpTests - [230d3686, L:/0:0:0:0:0:0:0:1:60241 -
R:/0:0:0:0:0:0:0:1:60245] REGISTERED
     * reactor.netty.http.HttpTests - [230d3686, L:/0:0:0:0:0:0:0:1:60241 -
R:/0:0:0:0:0:0:0:1:60245] ACTIVE
     * reactor.netty.http.HttpTests - [230d3686, L:/0:0:0:0:0:0:0:1:60241 -
R:/0:0:0:0:0:0:0:1:60245] READ: 145B
     * reactor.netty.http.HttpTests - [230d3686, L:/0:0:0:0:0:0:0:1:60241 -
R:/0:0:0:0:0:0:0:1:60245] WRITE: 38B
     * }
     * </pre>
     */
```

- AdvancedByteBufFormat#TEXTUAL

github.com/reactor/reactor-netty/tree/main/reactor-netty-core/src/main/java/reactor/netty/transport/logging/AdvancedByteBufFormat.java

```
    /**
     * When wire logging is enabled with this format, both events and content will be
logged.
     * The content will be in plain text format.
     * <p>Examples:</p>
     * <pre>
     * {@code
     * reactor.netty.http.HttpTests - [02c3db6c, L:/0:0:0:0:0:0:0:1:60317 -
R:/0:0:0:0:0:0:0:1:60319] REGISTERED
     * reactor.netty.http.HttpTests - [02c3db6c, L:/0:0:0:0:0:0:0:1:60317 -
R:/0:0:0:0:0:0:0:1:60319] ACTIVE
     * reactor.netty.http.HttpTests - [02c3db6c, L:/0:0:0:0:0:0:0:1:60317 -
R:/0:0:0:0:0:0:0:1:60319] READ: 145B POST /test/World HTTP/1.1
     * Content-Type: text/plain
     * user-agent: ReactorNetty/dev
     * ...
     * reactor.netty.http.HttpTests - [02c3db6c, L:/0:0:0:0:0:0:0:1:60317 -
R:/0:0:0:0:0:0:0:1:60319] WRITE: 38B HTTP/1.1 200 OK
     * content-length: 0
     * }
     * </pre>
     */
```

When you need to change the default formatter you can configure it as follows:

github.com/reactor/reactor-netty/tree/main/reactor-netty-examples/src/main/java/reactor/netty/

examples/documentation/tcp/server/wiretap/custom/Application.java

```
import io.netty.handler.logging.LogLevel;
import reactor.netty.DisposableServer;
import reactor.netty.tcp.TcpServer;
import reactor.netty.transport.logging.AdvancedByteBufFormat;

public class Application {

    public static void main(String[] args) {
        DisposableServer server =
                TcpServer.create()
                         .wiretap("logger-name", LogLevel.DEBUG,
AdvancedByteBufFormat.TEXTUAL) ①
                         .bindNow();

        server.onDispose()
              .block();
    }
}
```

① Enables the wire logging, AdvancedByteBufFormat#TEXTUAL is used for printing the content.

Event Loop Group

By default `Reactor Netty` uses an "Event Loop Group", where the number of the worker threads equals the number of processors available to the runtime on initialization (but with a minimum value of 4). When you need a different configuration, you can use one of the `LoopResources#create` methods.

The following listing shows the default configuration for the Event Loop Group:

github.com/reactor/reactor-netty/tree/main/reactor-netty-core/src/main/java/reactor/netty/ReactorNetty.java

```java
    /**
     * Default worker thread count, fallback to available processor
     * (but with a minimum value of 4).
     */
    public static final String IO_WORKER_COUNT = "reactor.netty.ioWorkerCount";
    /**
     * Default selector thread count, fallback to -1 (no selector thread)
     * <p><strong>Note:</strong> In most use cases using a worker thread also as a selector thread works well.
     * A possible use case for specifying a separate selector thread might be when the worker threads are too busy
     * and connections cannot be accepted fast enough.
     * <p><strong>Note:</strong> Although more than 1 can be configured as a selector thread count, in reality
     * only 1 thread will be used as a selector thread.
     */
    public static final String IO_SELECT_COUNT = "reactor.netty.ioSelectCount";
    /**
     * Default worker thread count for UDP, fallback to available processor
     * (but with a minimum value of 4).
     */
    public static final String UDP_IO_THREAD_COUNT = "reactor.netty.udp.ioThreadCount";
    /**
     * Default quiet period that guarantees that the disposal of the underlying LoopResources
     * will not happen, fallback to 2 seconds.
     */
    public static final String SHUTDOWN_QUIET_PERIOD = "reactor.netty.ioShutdownQuietPeriod";
    /**
     * Default maximum amount of time to wait until the disposal of the underlying LoopResources
     * regardless if a task was submitted during the quiet period, fallback to 15 seconds.
     */
    public static final String SHUTDOWN_TIMEOUT = "reactor.netty.ioShutdownTimeout";

    /**
     * Default value whether the native transport (epoll, kqueue) will be preferred,
     * fallback it will be preferred when available.
     */
    public static final String NATIVE = "reactor.netty.native";
```

If you need changes to these settings, you can apply the following configuration:

github.com/reactor/reactor-netty/tree/main/reactor-netty-examples/src/main/java/reactor/netty/examples/documentation/tcp/server/eventloop/Application.java

```java
import reactor.netty.DisposableServer;
import reactor.netty.resources.LoopResources;
import reactor.netty.tcp.TcpServer;

public class Application {

    public static void main(String[] args) {
        LoopResources loop = LoopResources.create("event-loop", 1, 4, true);

        DisposableServer server =
                TcpServer.create()
                         .runOn(loop)
                         .bindNow();

        server.onDispose()
              .block();
    }
}
```

Disposing Event Loop Group

- If you use the default `Event Loop Group` provided by Reactor Netty, invoke `HttpResources#disposeLoopsAndConnections/#disposeLoopsAndConnectionsLater` method.

> ℹ Disposing `HttpResources` means that every server/client that is using it, will not be able to use it anymore!

- If you use custom `LoopResources`, invoke `LoopResources#dispose/#disposeLater` method.

> ℹ Disposing the custom `LoopResources` means that every server/client that is configured to use it, will not be able to use it anymore!

SSL and TLS

When you need SSL or TLS, you can apply the configuration shown in the next listing. By default, if `OpenSSL` is available, `SslProvider.OPENSSL` provider is used as a provider. Otherwise `SslProvider.JDK` is used. Switching the provider can be done through `SslContextBuilder` or by setting `-Dio.netty.handler.ssl.noOpenSsl=true`.

The following example uses `SslContextBuilder`:

github.com/reactor/reactor-netty/tree/main/reactor-netty-examples/src/main/java/reactor/netty/examples/documentation/tcp/server/security/Application.java

```java
import reactor.netty.DisposableServer;
import reactor.netty.tcp.TcpServer;
import reactor.netty.tcp.TcpSslContextSpec;

import java.io.File;

public class Application {

    public static void main(String[] args) {
        File cert = new File("certificate.crt");
        File key = new File("private.key");

        TcpSslContextSpec tcpSslContextSpec = TcpSslContextSpec.forServer(cert, key);

        DisposableServer server =
                TcpServer.create()
                         .secure(spec -> spec.sslContext(tcpSslContextSpec))
                         .bindNow();

        server.onDispose()
              .block();
    }
}
```

Server Name Indication

You can configure the `TCP` server with multiple `SslContext` mapped to a specific domain. An exact domain name or a domain name containing a wildcard can be used when configuring the `SNI` mapping.

The following example uses a domain name containing a wildcard:

github.com/reactor/reactor-netty/tree/main/reactor-netty-examples/src/main/java/reactor/netty/examples/documentation/tcp/server/sni/Application.java

```java
import io.netty.handler.ssl.SslContext;
import io.netty.handler.ssl.SslContextBuilder;
import reactor.netty.DisposableServer;
import reactor.netty.tcp.TcpServer;

import java.io.File;

public class Application {

    public static void main(String[] args) throws Exception {
        File defaultCert = new File("default_certificate.crt");
        File defaultKey = new File("default_private.key");

        File testDomainCert = new File("default_certificate.crt");
        File testDomainKey = new File("default_private.key");

        SslContext defaultSslContext = SslContextBuilder.forServer(defaultCert,
defaultKey).build();
        SslContext testDomainSslContext = SslContextBuilder.forServer(testDomainCert,
testDomainKey).build();

        DisposableServer server =
                TcpServer.create()
                         .secure(spec -> spec.sslContext(defaultSslContext)
                                             .addSniMapping("*.test.com",
                                                     testDomainSpec -> testDomainSpec
.sslContext(testDomainSslContext)))
                         .bindNow();

        server.onDispose()
              .block();
    }
}
```

Metrics

The TCP server supports built-in integration with `Micrometer`. It exposes all metrics with a prefix of `reactor.netty.tcp.server`.

The following table provides information for the TCP server metrics:

metric name	type	description
reactor.netty.tcp.server.connections.total	Gauge	The number of all opened connections. See Connections Total
reactor.netty.tcp.server.data.received	DistributionSummary	Amount of the data received, in bytes. See Data Received

metric name	type	description
reactor.netty.tcp.server.data.sent	DistributionSummary	Amount of the data sent, in bytes. See Data Sent
reactor.netty.tcp.server.errors	Counter	Number of errors that occurred. See Errors Count
reactor.netty.tcp.server.tls.handshake.time	Timer	Time spent for TLS handshake. See Tls Handshake Time

These additional metrics are also available:

`ByteBufAllocator` metrics

metric name	type	description
reactor.netty.bytebuf.allocator.used.heap.memory	Gauge	The number of bytes reserved by heap buffer allocator. See Used Heap Memory
reactor.netty.bytebuf.allocator.used.direct.memory	Gauge	The number of bytes reserved by direct buffer allocator. See Used Direct Memory
reactor.netty.bytebuf.allocator.heap.arenas	Gauge	The number of heap arenas (when `PooledByteBufAllocator`). See Heap Arenas
reactor.netty.bytebuf.allocator.direct.arenas	Gauge	The number of direct arenas (when `PooledByteBufAllocator`). See Direct Arenas
reactor.netty.bytebuf.allocator.threadlocal.caches	Gauge	The number of thread local caches (when `PooledByteBufAllocator`). See Thread Local Caches
reactor.netty.bytebuf.allocator.small.cache.size	Gauge	The size of the small cache (when `PooledByteBufAllocator`). See Small Cache Size
reactor.netty.bytebuf.allocator.normal.cache.size	Gauge	The size of the normal cache (when `PooledByteBufAllocator`). See Normal Cache Size
reactor.netty.bytebuf.allocator.chunk.size	Gauge	The chunk size for an arena (when `PooledByteBufAllocator`). See Chunk Size
reactor.netty.bytebuf.allocator.active.heap.memory	Gauge	The actual bytes consumed by in-use buffers allocated from heap buffer pools (when `PooledByteBufAllocator`). See Active Heap Memory

metric name	type	description
reactor.netty.bytebuf.allocator.active.direct.memory	Gauge	The actual bytes consumed by in-use buffers allocated from direct buffer pools (when `PooledByteBufAllocator`). See Active Direct Memory

`EventLoop` metrics

metric name	type	description
reactor.netty.eventloop.pending.tasks	Gauge	The number of tasks that are pending for processing on an event loop. See Pending Tasks

The following example enables that integration:

github.com/reactor/reactor-netty/tree/main/reactor-netty-examples/src/main/java/reactor/netty/examples/documentation/tcp/server/metrics/Application.java

```
import reactor.netty.DisposableServer;
import reactor.netty.tcp.TcpServer;

public class Application {

    public static void main(String[] args) {
        DisposableServer server =
                TcpServer.create()
                         .metrics(true) ①
                         .bindNow();

        server.onDispose()
              .block();
    }
}
```

① Enables the built-in integration with Micrometer

When TCP server metrics are needed for an integration with a system other than `Micrometer` or you want to provide your own integration with `Micrometer`, you can provide your own metrics recorder, as follows:

github.com/reactor/reactor-netty/tree/main/reactor-netty-examples/src/main/java/reactor/netty/examples/documentation/tcp/server/metrics/custom/Application.java

```java
import reactor.netty.DisposableServer;
import reactor.netty.channel.ChannelMetricsRecorder;
import reactor.netty.tcp.TcpServer;

import java.net.SocketAddress;
import java.time.Duration;

public class Application {

    public static void main(String[] args) {
        DisposableServer server =
                TcpServer.create()
                         .metrics(true, CustomChannelMetricsRecorder::new) ①
                         .bindNow();

        server.onDispose()
              .block();
    }
```

① Enables TCP server metrics and provides `ChannelMetricsRecorder` implementation.

Tracing

The TCP server supports built-in integration with `Micrometer Tracing`.

The following table provides information for the TCP server spans:

contextual name	description
tls handshake	Information and time spent for TLS handshake. See Tls Handshake Span.

The following example enables that integration. This concrete example uses `Brave` and reports the information to `Zipkin`. See the `Micrometer Tracing` documentation for `OpenTelemetry` setup.

github.com/reactor/reactor-netty/tree/main/reactor-netty-examples/src/main/java/reactor/netty/examples/documentation/tcp/server/tracing/Application.java

```java
import brave.Tracing;
import brave.propagation.StrictCurrentTraceContext;
import brave.sampler.Sampler;
import io.micrometer.tracing.CurrentTraceContext;
import io.micrometer.tracing.Tracer;
import io.micrometer.tracing.brave.bridge.BraveBaggageManager;
import io.micrometer.tracing.brave.bridge.BraveCurrentTraceContext;
import io.micrometer.tracing.brave.bridge.BraveTracer;
import reactor.core.publisher.Mono;
import reactor.netty.DisposableServer;
import reactor.netty.observability.ReactorNettyTracingObservationHandler;
```

```
import reactor.netty.tcp.TcpServer;
import zipkin2.reporter.brave.AsyncZipkinSpanHandler;
import zipkin2.reporter.urlconnection.URLConnectionSender;

import static reactor.netty.Metrics.OBSERVATION_REGISTRY;

public class Application {

    public static void main(String[] args) {
        init(); ①

        DisposableServer server =
                TcpServer.create()
                         .metrics(true) ②
                         .handle((inbound, outbound) -> outbound.sendString(Mono.just
("hello")))
                         .bindNow();

        server.onDispose()
              .block();
    }

    /**
     * This setup is based on
     * <a
href="https://micrometer.io/docs/tracing#_micrometer_tracing_brave_setup">Micrometer
Tracing Brave Setup</a>.
     */
    static void init() {
        AsyncZipkinSpanHandler spanHandler = AsyncZipkinSpanHandler
                .create(URLConnectionSender.create
("http://localhost:9411/api/v2/spans"));

        StrictCurrentTraceContext braveCurrentTraceContext =
StrictCurrentTraceContext.create();

        CurrentTraceContext bridgeContext = new BraveCurrentTraceContext
(braveCurrentTraceContext);

        Tracing tracing =
                Tracing.newBuilder()
                       .currentTraceContext(braveCurrentTraceContext)
                       .supportsJoin(false)
                       .traceId128Bit(true)
                       .sampler(Sampler.ALWAYS_SAMPLE)
                       .addSpanHandler(spanHandler)
                       .localServiceName("reactor-netty-examples")
                       .build();

        brave.Tracer braveTracer = tracing.tracer();
```

```
        Tracer tracer = new BraveTracer(braveTracer, bridgeContext, new
BraveBaggageManager());

        OBSERVATION_REGISTRY.observationConfig()
                            .observationHandler(new
ReactorNettyTracingObservationHandler(tracer));
    }
}
```

① Initializes Brave, Zipkin, and the Observation registry.

② Enables the built-in integration with Micrometer.

The result in `Zipkin` looks like:

Access Current Observation

Project Micrometer provides `a library` that assists with context propagation across different types of context mechanisms such as `ThreadLocal`, `Reactor Context` and others.

The following example shows how to use this library in a custom `ChannelHandler`:

github.com/reactor/reactor-netty/tree/main/reactor-netty-examples/src/main/java/reactor/netty/examples/documentation/tcp/server/tracing/custom/Application.java

```
import brave.Tracing;
import brave.propagation.StrictCurrentTraceContext;
import brave.sampler.Sampler;
import io.micrometer.context.ContextSnapshot;
import io.micrometer.context.ContextSnapshotFactory;
```

```java
import io.micrometer.tracing.CurrentTraceContext;
import io.micrometer.tracing.Tracer;
import io.micrometer.tracing.brave.bridge.BraveBaggageManager;
import io.micrometer.tracing.brave.bridge.BraveCurrentTraceContext;
import io.micrometer.tracing.brave.bridge.BraveTracer;
import io.netty.channel.ChannelHandler;
import io.netty.channel.ChannelHandlerContext;
import io.netty.channel.ChannelInboundHandlerAdapter;
import reactor.core.publisher.Mono;
import reactor.netty.DisposableServer;
import reactor.netty.NettyPipeline;
import reactor.netty.observability.ReactorNettyTracingObservationHandler;
import reactor.netty.tcp.TcpServer;
import reactor.netty.tcp.TcpSslContextSpec;
import zipkin2.reporter.brave.AsyncZipkinSpanHandler;
import zipkin2.reporter.urlconnection.URLConnectionSender;

import java.io.File;

import static reactor.netty.Metrics.OBSERVATION_REGISTRY;

public class Application {

    public static void main(String[] args) {
        init(); ①

        File cert = new File("certificate.crt");
        File key = new File("private.key");

        TcpSslContextSpec tcpSslContextSpec = TcpSslContextSpec.forServer(cert, key);

        DisposableServer server =
                TcpServer.create()
                         .metrics(true) ②
                         .doOnChannelInit((observer, channel, address) -> channel
.pipeline().addAfter(
                                 NettyPipeline.SslHandler, "custom-channel-handler",
CustomChannelInboundHandler.INSTANCE)) ③
                         .secure(spec -> spec.sslContext(tcpSslContextSpec))
                         .handle((inbound, outbound) -> outbound.sendString(Mono.just
("hello")))
                         .bindNow();

        server.onDispose()
              .block();
    }

    static final class CustomChannelInboundHandler extends
ChannelInboundHandlerAdapter {

        static final ChannelHandler INSTANCE = new CustomChannelInboundHandler();
```

```
        @Override
        @SuppressWarnings("try")
        public void channelActive(ChannelHandlerContext ctx) {
            try (ContextSnapshot.Scope scope = ContextSnapshotFactory.builder().
build().setThreadLocalsFrom(ctx.channel())) {
                System.out.println("Current Observation in Scope: " +
OBSERVATION_REGISTRY.getCurrentObservation());
                ctx.fireChannelActive();
            }
            System.out.println("Current Observation: " + OBSERVATION_REGISTRY
.getCurrentObservation());
        }

        @Override
        public boolean isSharable() {
            return true;
        }
    }
```

① Initializes Brave, Zipkin, and the Observation registry.

② Enables the built-in integration with Micrometer.

③ Custom `ChannelHandler` that uses context propagation library. This concrete example overrides only `ChannelInboundHandlerAdapter#channelActive`, if it is needed, the same logic can be used for the rest of the methods. Also, this concrete example sets all `ThreadLocal` values for which there is a value in the given `Channel`, if another behaviour is needed please check `context propagation library API`. For example, you may want to set only some of the `ThreadLocal` values.

> When you enable Reactor Netty tracing within a framework, you may need to let Reactor Netty use the `ObservationRegistry` created by this framework. For this purpose you need to invoke `reactor.netty.Metrics#observationRegistry`. You may also need to configure the Reactor Netty `ObservationHandlers` using the API provided by the framework.

Unix Domain Sockets

The `TCP` server supports Unix Domain Sockets (UDS) when native transport is in use.

The following example shows how to use UDS support:

github.com/reactor/reactor-netty/tree/main/reactor-netty-examples/src/main/java/reactor/netty/examples/documentation/tcp/server/uds/Application.java

```java
import io.netty.channel.unix.DomainSocketAddress;
import reactor.netty.DisposableServer;
import reactor.netty.tcp.TcpServer;

public class Application {

    public static void main(String[] args) {
        DisposableServer server =
                TcpServer.create()
                         .bindAddress(() -> new DomainSocketAddress("/tmp/test.sock"))
①
                         .bindNow();

        server.onDispose()
              .block();
    }
}
```

① Specifies `DomainSocketAddress` that will be used

Chapter 4. TCP Client

Reactor Netty provides the easy-to-use and easy-to-configure `TcpClient`. It hides most of the Netty functionality that is needed in order to create a `TCP` client and adds Reactive Streams backpressure.

Connect and Disconnect

To connect the `TCP` client to a given endpoint, you must create and configure a `TcpClient` instance. By default, the `host` is `localhost` and the `port` is `12012`. The following example shows how to create a `TcpClient`:

github.com/reactor/reactor-netty/tree/main/reactor-netty-examples/src/main/java/reactor/netty/examples/documentation/tcp/client/create/Application.java

```
import reactor.netty.Connection;
import reactor.netty.tcp.TcpClient;

public class Application {

    public static void main(String[] args) {
        Connection connection =
                TcpClient.create()      ①
                         .connectNow(); ②

        connection.onDispose()
                  .block();
    }
}
```

① Creates a `TcpClient` instance that is ready for configuring.

② Connects the client in a blocking fashion and waits for it to finish initializing.

The returned `Connection` offers a simple connection API, including `disposeNow()`, which shuts the client down in a blocking fashion.

Host and Port

To connect to a specific `host` and `port`, you can apply the following configuration to the `TCP` client. The following example shows how to do so:

github.com/reactor/reactor-netty/tree/main/reactor-netty-examples/src/main/java/reactor/netty/examples/documentation/tcp/client/address/Application.java

```java
import reactor.netty.Connection;
import reactor.netty.tcp.TcpClient;

public class Application {

    public static void main(String[] args) {
        Connection connection =
                TcpClient.create()
                         .host("example.com") ①
                         .port(80)            ②
                         .connectNow();

        connection.onDispose()
                  .block();
    }
}
```

① Configures the `TCP` host

② Configures the `TCP` port

> The port can be specified also with **PORT** environment variable.

Eager Initialization

By default, the initialization of the `TcpClient` resources happens on demand. This means that the `connect operation` absorbs the extra time needed to initialize and load:

- the event loop group
- the host name resolver
- the native transport libraries (when native transport is used)
- the native libraries for the security (in case of `OpenSsl`)

When you need to preload these resources, you can configure the `TcpClient` as follows:

github.com/reactor/reactor-netty/tree/main/reactor-netty-examples/src/main/java/reactor/netty/examples/documentation/tcp/client/warmup/Application.java

```java
import reactor.core.publisher.Mono;
import reactor.netty.Connection;
import reactor.netty.tcp.TcpClient;

public class Application {

    public static void main(String[] args) {
        TcpClient tcpClient =
                TcpClient.create()
                         .host("example.com")
                         .port(80)
                         .handle((inbound, outbound) -> outbound.sendString(Mono.just
("hello")));

        tcpClient.warmup() ①
                 .block();

        Connection connection = tcpClient.connectNow(); ②

        connection.onDispose()
                  .block();
    }
}
```

① Initialize and load the event loop group, the host name resolver, the native transport libraries and the native libraries for the security

② Host name resolution happens when connecting to the remote peer

Writing Data

To send data to a given endpoint, you must attach an I/O handler. The I/O handler has access to `NettyOutbound` to be able to write data.

github.com/reactor/reactor-netty/tree/main/reactor-netty-examples/src/main/java/reactor/netty/examples/documentation/tcp/client/send/Application.java

```java
import reactor.core.publisher.Mono;
import reactor.netty.Connection;
import reactor.netty.tcp.TcpClient;

public class Application {

    public static void main(String[] args) {
        Connection connection =
                TcpClient.create()
                         .host("example.com")
                         .port(80)
                         .handle((inbound, outbound) -> outbound.sendString(Mono.just
("hello"))) ①
                         .connectNow();

        connection.onDispose()
                  .block();
    }
}
```

① Sends `hello` string to the endpoint.

When you need more control over the writing process, as an alternative for I/O handler you may use `Connection#outbound`. As opposed to I/O handler where the connection is closed when the provided `Publisher` finishes (in case of finite `Publisher`), when using `Connection#outbound`, you have to invoke explicitly `Connection#dispose` in order to close the connection.

github.com/reactor/reactor-netty/tree/main/reactor-netty-examples/src/main/java/reactor/netty/examples/documentation/tcp/client/send/connection/Application.java

```java
import reactor.core.publisher.Mono;
import reactor.netty.Connection;
import reactor.netty.tcp.TcpClient;

public class Application {

    public static void main(String[] args) {
        Connection connection =
                TcpClient.create()
                         .host("example.com")
                         .port(80)
                         .connectNow();

        connection.outbound()
                  .sendString(Mono.just("hello 1")) ①
                  .then()
                  .subscribe();

        connection.outbound()
                  .sendString(Mono.just("hello 2")) ②
                  .then()
                  .subscribe(null, null, connection::dispose); ③

        connection.onDispose()
                  .block();
    }
}
```

① Sends `hello 1` string to the endpoint.

② Sends `hello 2` string to the endpoint.

③ Closes the connection once the message is sent to the endpoint.

Consuming Data

To receive data from a given endpoint, you must attach an I/O handler. The I/O handler has access to `NettyInbound` to be able to read data. The following example shows how to do so:

github.com/reactor/reactor-netty/tree/main/reactor-netty-examples/src/main/java/reactor/netty/examples/documentation/tcp/client/read/Application.java

```java
import reactor.netty.Connection;
import reactor.netty.tcp.TcpClient;

public class Application {

    public static void main(String[] args) {
        Connection connection =
                TcpClient.create()
                         .host("example.com")
                         .port(80)
                         .handle((inbound, outbound) -> inbound.receive().then()) ①
                         .connectNow();

        connection.onDispose()
                  .block();
    }
}
```

① Receives data from a given endpoint

When you need more control over the reading process, as an alternative for I/O handler you may use `Connection#inbound`. As opposed to I/O handler where the connection is closed when the provided `Publisher` finishes (in case of finite `Publisher`), when using `Connection#inbound`, you have to invoke explicitly `Connection#dispose` in order to close the connection.

github.com/reactor/reactor-netty/tree/main/reactor-netty-examples/src/main/java/reactor/netty/examples/documentation/tcp/client/read/connection/Application.java

```java
import reactor.netty.Connection;
import reactor.netty.tcp.TcpClient;

public class Application {

    public static void main(String[] args) {
        Connection connection =
                TcpClient.create()
                         .host("example.com")
                         .port(80)
                         .connectNow();

        connection.inbound()
                  .receive() ①
                  .then()
                  .subscribe();

        connection.onDispose()
                  .block();
    }
}
```

① Receives data from a given endpoint.

Lifecycle Callbacks

The following lifecycle callbacks are provided to let you extend the `TcpClient`.

Callback	Description
`doAfterResolve`	Invoked after the remote address has been resolved successfully.
`doOnChannelInit`	Invoked when initializing the channel.
`doOnConnect`	Invoked when the channel is about to connect.
`doOnConnected`	Invoked after the channel has been connected.
`doOnDisconnected`	Invoked after the channel has been disconnected.
`doOnResolve`	Invoked when the remote address is about to be resolved.
`doOnResolveError`	Invoked in case the remote address hasn't been resolved successfully.

The following example uses the `doOnConnected` and `doOnChannelInit` callbacks:

github.com/reactor/reactor-netty/tree/main/reactor-netty-examples/src/main/java/reactor/netty/examples/documentation/tcp/client/lifecycle/Application.java

```java
import io.netty.handler.logging.LoggingHandler;
import io.netty.handler.timeout.ReadTimeoutHandler;
import reactor.netty.Connection;
import reactor.netty.tcp.TcpClient;
import java.util.concurrent.TimeUnit;

public class Application {

    public static void main(String[] args) {
        Connection connection =
                TcpClient.create()
                         .host("example.com")
                         .port(80)
                         .doOnConnected(conn ->
                             conn.addHandlerFirst(new ReadTimeoutHandler(10, TimeUnit
.SECONDS))) ①
                         .doOnChannelInit((observer, channel, remoteAddress) ->
                             channel.pipeline()
                                    .addFirst(new LoggingHandler
("reactor.netty.examples")))     ②
                         .connectNow();

        connection.onDispose()
                  .block();
    }
}
```

① `Netty` pipeline is extended with `ReadTimeoutHandler` when the channel has been connected.

② `Netty` pipeline is extended with `LoggingHandler` when initializing the channel.

TCP-level Configurations

This section describes three kinds of configuration that you can use at the TCP level:

- Channel Options
- Wire Logger
- Event Loop Group

Channel Options

By default, the `TCP` client is configured with the following options:

github.com/reactor/reactor-netty/tree/main/reactor-netty-core/src/main/java/reactor/netty/tcp/TcpClientConnect.java

```java
    TcpClientConnect(ConnectionProvider provider) {
        this.config = new TcpClientConfig(
```

```
                 provider,
                 Collections.singletonMap(ChannelOption.AUTO_READ, false),
                 () -> AddressUtils.createUnresolved(NetUtil.LOCALHOST.
getHostAddress(), DEFAULT_PORT));
    }
```

If additional options are necessary or changes to the current options are needed, you can apply the following configuration:

github.com/reactor/reactor-netty/tree/main/reactor-netty-examples/src/main/java/reactor/netty/examples/documentation/tcp/client/channeloptions/Application.java

```
import io.netty.channel.ChannelOption;
import reactor.netty.Connection;
import reactor.netty.tcp.TcpClient;

public class Application {

    public static void main(String[] args) {
        Connection connection =
                TcpClient.create()
                         .host("example.com")
                         .port(80)
                         .option(ChannelOption.CONNECT_TIMEOUT_MILLIS, 10000)
                         .connectNow();

        connection.onDispose()
                  .block();
    }
}
```

You can find more about `Netty` channel options at the following links:

- Common ChannelOption
- Epoll ChannelOption
- KQueue ChannelOption
- Socket Options

Wire Logger

Reactor Netty provides wire logging for when the traffic between the peers needs to be inspected. By default, wire logging is disabled. To enable it, you must set the logger `reactor.netty.tcp.TcpClient` level to `DEBUG` and apply the following configuration:

github.com/reactor/reactor-netty/tree/main/reactor-netty-examples/src/main/java/reactor/netty/examples/documentation/tcp/client/wiretap/Application.java

```java
import reactor.netty.Connection;
import reactor.netty.tcp.TcpClient;

public class Application {

    public static void main(String[] args) {
        Connection connection =
                TcpClient.create()
                         .wiretap(true) ①
                         .host("example.com")
                         .port(80)
                         .connectNow();

        connection.onDispose()
                  .block();
    }
}
```

① Enables the wire logging

Wire Logger formatters

Reactor Netty supports 3 different formatters:

- AdvancedByteBufFormat#HEX_DUMP - the default

github.com/reactor/reactor-netty/tree/main/reactor-netty-core/src/main/java/reactor/netty/transport/logging/AdvancedByteBufFormat.java

```
    /**
     * When wire logging is enabled with this format, both events and content will be
logged.
     * The content will be in hex format.
     * <p>Examples:</p>
     * <pre>
     * {@code
     * reactor.netty.http.HttpTests - [d5230a14, L:/0:0:0:0:0:0:0:1:60267 -
R:/0:0:0:0:0:0:0:1:60269] REGISTERED
     * reactor.netty.http.HttpTests - [d5230a14, L:/0:0:0:0:0:0:0:1:60267 -
R:/0:0:0:0:0:0:0:1:60269] ACTIVE
     * reactor.netty.http.HttpTests - [d5230a14, L:/0:0:0:0:0:0:0:1:60267 -
R:/0:0:0:0:0:0:0:1:60269] READ: 145B
     *          +-------------------------------------------------+
     *          |  0  1  2  3  4  5  6  7  8  9  a  b  c  d  e  f |
     * +--------+-------------------------------------------------+----------------+
     * |00000000| 50 4f 53 54 20 2f 74 65 73 74 2f 57 6f 72 6c 64 |POST /test/World|
     * |00000010| 20 48 54 54 50 2f 31 2e 31 0d 0a 43 6f 6e 74 65 | HTTP/1.1..Conte|
     * |00000020| 6e 74 2d 54 79 70 65 3a 20 74 65 78 74 2f 70 6c |nt-Type: text/pl|
     * |00000030| 61 69 6e 0d 0a 75 73 65 72 2d 61 67 65 6e 74 3a |ain..user-agent:|
     * |00000040| 20 52 65 61 63 74 6f 72 4e 65 74 74 79 2f 64 65 | ReactorNetty/de|
     * ...
     * reactor.netty.http.HttpTests - [d5230a14, L:/0:0:0:0:0:0:0:1:60267 -
R:/0:0:0:0:0:0:0:1:60269] WRITE: 38B
     *          +-------------------------------------------------+
     *          |  0  1  2  3  4  5  6  7  8  9  a  b  c  d  e  f |
     * +--------+-------------------------------------------------+----------------+
     * |00000000| 48 54 54 50 2f 31 2e 31 20 32 30 30 20 4f 4b 0d |HTTP/1.1 200 OK.|
     * |00000010| 0a 63 6f 6e 74 65 6e 74 2d 6c 65 6e 67 74 68 3a |.content-length:|
     * |00000020| 20 30 0d 0a 0d 0a                               | 0....          |
     * +--------+-------------------------------------------------+----------------+
     * }
     * </pre>
     */
```

- AdvancedByteBufFormat#SIMPLE

github.com/reactor/reactor-netty/tree/main/reactor-netty-core/src/main/java/reactor/netty/transport/logging/AdvancedByteBufFormat.java

```
    /**
     * When wire logging is enabled with this format, only the events will be logged.
     * <p>Examples:</p>
     * <pre>
     * {@code
     * reactor.netty.http.HttpTests - [230d3686, L:/0:0:0:0:0:0:0:1:60241 -
R:/0:0:0:0:0:0:0:1:60245] REGISTERED
     * reactor.netty.http.HttpTests - [230d3686, L:/0:0:0:0:0:0:0:1:60241 -
R:/0:0:0:0:0:0:0:1:60245] ACTIVE
     * reactor.netty.http.HttpTests - [230d3686, L:/0:0:0:0:0:0:0:1:60241 -
R:/0:0:0:0:0:0:0:1:60245] READ: 145B
     * reactor.netty.http.HttpTests - [230d3686, L:/0:0:0:0:0:0:0:1:60241 -
R:/0:0:0:0:0:0:0:1:60245] WRITE: 38B
     * }
     * </pre>
     */
```

- AdvancedByteBufFormat#TEXTUAL

github.com/reactor/reactor-netty/tree/main/reactor-netty-core/src/main/java/reactor/netty/transport/logging/AdvancedByteBufFormat.java

```
    /**
     * When wire logging is enabled with this format, both events and content will be
logged.
     * The content will be in plain text format.
     * <p>Examples:</p>
     * <pre>
     * {@code
     * reactor.netty.http.HttpTests - [02c3db6c, L:/0:0:0:0:0:0:0:1:60317 -
R:/0:0:0:0:0:0:0:1:60319] REGISTERED
     * reactor.netty.http.HttpTests - [02c3db6c, L:/0:0:0:0:0:0:0:1:60317 -
R:/0:0:0:0:0:0:0:1:60319] ACTIVE
     * reactor.netty.http.HttpTests - [02c3db6c, L:/0:0:0:0:0:0:0:1:60317 -
R:/0:0:0:0:0:0:0:1:60319] READ: 145B POST /test/World HTTP/1.1
     * Content-Type: text/plain
     * user-agent: ReactorNetty/dev
     * ...
     * reactor.netty.http.HttpTests - [02c3db6c, L:/0:0:0:0:0:0:0:1:60317 -
R:/0:0:0:0:0:0:0:1:60319] WRITE: 38B HTTP/1.1 200 OK
     * content-length: 0
     * }
     * </pre>
     */
```

When you need to change the default formatter you can configure it as follows:

github.com/reactor/reactor-netty/tree/main/reactor-netty-examples/src/main/java/reactor/netty/

examples/documentation/tcp/client/wiretap/custom/Application.java

```
import io.netty.handler.logging.LogLevel;
import reactor.netty.Connection;
import reactor.netty.tcp.TcpClient;
import reactor.netty.transport.logging.AdvancedByteBufFormat;

public class Application {

    public static void main(String[] args) {
        Connection connection =
                TcpClient.create()
                         .wiretap("logger-name", LogLevel.DEBUG,
AdvancedByteBufFormat.TEXTUAL) ①
                         .host("example.com")
                         .port(80)
                         .connectNow();

        connection.onDispose()
                  .block();
    }
}
```

① Enables the wire logging, AdvancedByteBufFormat#TEXTUAL is used for printing the content.

Event Loop Group

By default `Reactor Netty` uses an "Event Loop Group", where the number of the worker threads equals the number of processors available to the runtime on initialization (but with a minimum value of 4). When you need a different configuration, you can use one of the `LoopResources#create` methods.

The following listing shows the default configuration for the Event Loop Group:

github.com/reactor/reactor-netty/tree/main/reactor-netty-core/src/main/java/reactor/netty/ReactorNetty.java

```
    /**
     * Default worker thread count, fallback to available processor
     * (but with a minimum value of 4).
     */
    public static final String IO_WORKER_COUNT = "reactor.netty.ioWorkerCount";
    /**
     * Default selector thread count, fallback to -1 (no selector thread)
     * <p><strong>Note:</strong> In most use cases using a worker thread also as a
selector thread works well.
     * A possible use case for specifying a separate selector thread might be when the
worker threads are too busy
     * and connections cannot be accepted fast enough.
     * <p><strong>Note:</strong> Although more than 1 can be configured as a selector
thread count, in reality
     * only 1 thread will be used as a selector thread.
     */
    public static final String IO_SELECT_COUNT = "reactor.netty.ioSelectCount";
    /**
     * Default worker thread count for UDP, fallback to available processor
     * (but with a minimum value of 4).
     */
    public static final String UDP_IO_THREAD_COUNT =
"reactor.netty.udp.ioThreadCount";
    /**
     * Default quiet period that guarantees that the disposal of the underlying
LoopResources
     * will not happen, fallback to 2 seconds.
     */
    public static final String SHUTDOWN_QUIET_PERIOD =
"reactor.netty.ioShutdownQuietPeriod";
    /**
     * Default maximum amount of time to wait until the disposal of the underlying
LoopResources
     * regardless if a task was submitted during the quiet period, fallback to 15
seconds.
     */
    public static final String SHUTDOWN_TIMEOUT = "reactor.netty.ioShutdownTimeout";

    /**
     * Default value whether the native transport (epoll, kqueue) will be preferred,
     * fallback it will be preferred when available.
     */
    public static final String NATIVE = "reactor.netty.native";
```

If you need changes to these settings, you can apply the following configuration:

github.com/reactor/reactor-netty/tree/main/reactor-netty-examples/src/main/java/reactor/netty/examples/documentation/tcp/client/eventloop/Application.java

```java
import reactor.netty.Connection;
import reactor.netty.resources.LoopResources;
import reactor.netty.tcp.TcpClient;

public class Application {

    public static void main(String[] args) {
        LoopResources loop = LoopResources.create("event-loop", 1, 4, true);

        Connection connection =
                TcpClient.create()
                         .host("example.com")
                         .port(80)
                         .runOn(loop)
                         .connectNow();

        connection.onDispose()
                  .block();
    }
}
```

Disposing Event Loop Group

- If you use the default `Event Loop Group` provided by Reactor Netty, invoke `HttpResources#disposeLoopsAndConnections/#disposeLoopsAndConnectionsLater` method.

> Disposing `HttpResources` means that every server/client that is using it, will not be able to use it anymore!

- If you use custom `LoopResources`, invoke `LoopResources#dispose/#disposeLater` method.

> Disposing the custom `LoopResources` means that every server/client that is configured to use it, will not be able to use it anymore!

Connection Pool

By default, `TcpClient` uses a “fixed” connection pool with `500` as the maximum number of active channels and `1000` as the maximum number of further channel acquisition attempts allowed to be kept in a pending state (for the rest of the configurations check the system properties or the builder configurations below). This means that the implementation creates a new channel if someone tries to acquire a channel as long as less than `500` have been created and are managed by the pool. When the maximum number of channels in the pool is reached, up to `1000` new attempts to acquire a channel are delayed (pending) until a channel is closed (and thus a slot is free and a new connection can be opened), and further attempts are declined with an error.

> Connections used by the `TcpClient` are never returned to the pool, but closed. When a connection is closed, a slot is freed in the pool and thus a new connection

> can be opened when needed. This behaviour is specific only for `TcpClient` and is intentional because only the user/framework knows if the actual protocol is compatible with reusing connections. (opposed to `HttpClient` where the protocol is known and Reactor Netty can return the connection to the pool when this is possible)

github.com/reactor/reactor-netty/tree/main/reactor-netty-core/src/main/java/reactor/netty/ReactorNetty.java

```
    /**
     * Default max connections. Fallback to
     * 2 * available number of processors (but with a minimum value of 16)
     */
    public static final String POOL_MAX_CONNECTIONS =
"reactor.netty.pool.maxConnections";
    /**
     * Default acquisition timeout (milliseconds) before error. If -1 will never wait
to
     * acquire before opening a new
     * connection in an unbounded fashion. Fallback 45 seconds
     */
    public static final String POOL_ACQUIRE_TIMEOUT =
"reactor.netty.pool.acquireTimeout";
```

When you need to change the default settings, you can configure the `ConnectionProvider` as follows:

github.com/reactor/reactor-netty/tree/main/reactor-netty-examples/src/main/java/reactor/netty/examples/documentation/tcp/client/pool/config/Application.java

```java
import reactor.netty.Connection;
import reactor.netty.resources.ConnectionProvider;
import reactor.netty.tcp.TcpClient;
import java.time.Duration;

public class Application {

    public static void main(String[] args) {
        ConnectionProvider provider =
                ConnectionProvider.builder("fixed")
                                  .maxConnections(50)
                                  .pendingAcquireTimeout(Duration.ofSeconds(60)) ①
                                  .build();

        Connection connection =
                TcpClient.create(provider)
                         .host("example.com")
                         .port(80)
                         .connectNow();

        connection.onDispose()
                  .block();
    }
}
```

① Configures the maximum time for the pending acquire operation to 60 seconds.

The following listing shows the available configurations:

Configuration name	Description
`disposeInactivePoolsInBackground`	When this option is enabled, connection pools are regularly checked in the background, and those that are empty and been inactive for a specified time become eligible for disposal. By default, this background disposal of inactive pools is disabled.

Configuration name	Description
`disposeTimeout`	When `ConnectionProvider#dispose()` or `ConnectionProvider#disposeLater()` is called, trigger a graceful shutdown for the connection pools, with this grace period timeout. From there on, all calls for acquiring a connection will fail fast with an exception. However, for the provided `Duration`, pending acquires will get a chance to be served. Note: The rejection of new acquires and the grace timer start immediately, irrespective of subscription to the `Mono` returned by `ConnectionProvider#disposeLater()`. Subsequent calls return the same `Mono`, effectively getting notifications from the first graceful shutdown call and ignoring subsequently provided timeouts. By default, dispose timeout is not specified.
`maxConnections`	The maximum number of connections (per connection pool) before start pending. Default to 2 * available number of processors (but with a minimum value of 16).
`metrics`	Enables/disables built-in integration with Micrometer. `ConnectionProvider.MeterRegistrar` can be provided for integration with another metrics system. By default, metrics are not enabled.
`pendingAcquireMaxCount`	The maximum number of extra attempts at acquiring a connection to keep in a pending queue. If -1 is specified, the pending queue does not have upper limit. Default to 2 * max connections.
`pendingAcquireTimeout`	The maximum time before which a pending acquire must complete, or a TimeoutException is thrown (resolution: ms). If -1 is specified, no such timeout is applied. Default: 45 seconds.

If you need to disable the connection pool, you can apply the following configuration:

github.com/reactor/reactor-netty/tree/main/reactor-netty-examples/src/main/java/reactor/netty/examples/documentation/tcp/client/pool/Application.java

```java
import reactor.netty.Connection;
import reactor.netty.tcp.TcpClient;

public class Application {

    public static void main(String[] args) {
        Connection connection =
                TcpClient.newConnection()
                         .host("example.com")
                         .port(80)
                         .connectNow();

        connection.onDispose()
                  .block();
    }
}
```

Disposing Connection Pool

- If you use the default `ConnectionProvider` provided by Reactor Netty, invoke `HttpResources#disposeLoopsAndConnections`/`#disposeLoopsAndConnectionsLater` method.

> Disposing `HttpResources` means that every client that is using it, will not be able to use it anymore!

- If you use custom `ConnectionProvider`, invoke `ConnectionProvider#dispose`/`#disposeLater`/`#disposeWhen` method.

> Disposing the custom `ConnectionProvider` means that every client that is configured to use it, will not be able to use it anymore!

Metrics

The pooled `ConnectionProvider` supports built-in integration with `Micrometer`. It exposes all metrics with a prefix of `reactor.netty.connection.provider`.

Pooled `ConnectionProvider` metrics

metric name	type	description
reactor.netty.connection.provider.total.connections	Gauge	The number of all connections, active or idle. See Total Connections
reactor.netty.connection.provider.active.connections	Gauge	The number of the connections that have been successfully acquired and are in active use. See Active Connections

metric name	type	description
reactor.netty.connection.provider.max.connections	Gauge	The maximum number of active connections that are allowed. See Max Connections
reactor.netty.connection.provider.idle.connections	Gauge	The number of the idle connections. See Idle Connections
reactor.netty.connection.provider.pending.connections	Gauge	The number of requests that are waiting for a connection. See Pending Connections
reactor.netty.connection.provider.pending.connections.time	Timer	Time spent in pending acquire a connection from the connection pool. See Pending Connections Time
reactor.netty.connection.provider.max.pending.connections	Gauge	The maximum number of requests that will be queued while waiting for a ready connection. See Max Pending Connections

The following example enables that integration:

github.com/reactor/reactor-netty/tree/main/reactor-netty-examples/src/main/java/reactor/netty/examples/documentation/tcp/client/pool/metrics/Application.java

```java
import reactor.netty.Connection;
import reactor.netty.resources.ConnectionProvider;
import reactor.netty.tcp.TcpClient;

public class Application {

    public static void main(String[] args) {
        ConnectionProvider provider =
                ConnectionProvider.builder("fixed")
                                  .maxConnections(50)
                                  .metrics(true) ①
                                  .build();

        Connection connection =
                TcpClient.create(provider)
                         .host("example.com")
                         .port(80)
                         .connectNow();

        connection.onDispose()
                  .block();
    }
}
```

① Enables the built-in integration with Micrometer

SSL and TLS

When you need SSL or TLS, you can apply the following configuration. By default, if `OpenSSL` is available, the SslProvider.OPENSSL provider is used as a provider. Otherwise, the provider is SslProvider.JDK. You can switch the provider by using SslContextBuilder or by setting `-Dio.netty.handler.ssl.noOpenSsl=true`.

The following example uses `SslContextBuilder`:

github.com/reactor/reactor-netty/tree/main/reactor-netty-examples/src/main/java/reactor/netty/examples/documentation/tcp/client/security/Application.java

```java
import reactor.netty.Connection;
import reactor.netty.tcp.TcpClient;
import reactor.netty.tcp.TcpSslContextSpec;

public class Application {

    public static void main(String[] args) {
        TcpSslContextSpec tcpSslContextSpec = TcpSslContextSpec.forClient();

        Connection connection =
                TcpClient.create()
                         .host("example.com")
                         .port(443)
                         .secure(spec -> spec.sslContext(tcpSslContextSpec))
                         .connectNow();

        connection.onDispose()
                  .block();
    }
}
```

Server Name Indication

By default, the `TCP` client sends the remote host name as `SNI` server name. When you need to change this default setting, you can configure the `TCP` client as follows:

github.com/reactor/reactor-netty/tree/main/reactor-netty-examples/src/main/java/reactor/netty/examples/documentation/tcp/client/sni/Application.java

```java
import io.netty.handler.ssl.SslContext;
import io.netty.handler.ssl.SslContextBuilder;
import reactor.netty.Connection;
import reactor.netty.tcp.TcpClient;

import javax.net.ssl.SNIHostName;

public class Application {

    public static void main(String[] args) throws Exception {
        SslContext sslContext = SslContextBuilder.forClient().build();

        Connection connection =
                TcpClient.create()
                         .host("127.0.0.1")
                         .port(8080)
                         .secure(spec -> spec.sslContext(sslContext)
                                             .serverNames(new SNIHostName(
"test.com")))
                         .connectNow();

        connection.onDispose()
                  .block();
    }
}
```

Proxy Support

Reactor Netty supports the proxy functionality provided by Netty and provides a way to specify `non proxy hosts` through the `ProxyProvider` builder.

Netty's HTTP proxy support always uses `CONNECT` method in order to establish a tunnel to the specified proxy regardless of the scheme that is used `http` or `https`. (More information: Netty enforce HTTP proxy to support HTTP CONNECT method). Some proxies might not support `CONNECT` method when the scheme is `http` or might need to be configured in order to support this way of communication. Sometimes this might be the reason for not being able to connect to the proxy. Consider checking the proxy documentation whether it supports or needs an additional configuration in order to support `CONNECT` method.

The following example uses `ProxyProvider`:

github.com/reactor/reactor-netty/tree/main/reactor-netty-examples/src/main/java/reactor/netty/examples/documentation/tcp/client/proxy/Application.java

```java
import reactor.netty.Connection;
import reactor.netty.transport.ProxyProvider;
import reactor.netty.tcp.TcpClient;

public class Application {

    public static void main(String[] args) {
        Connection connection =
                TcpClient.create()
                         .host("example.com")
                         .port(80)
                         .proxy(spec -> spec.type(ProxyProvider.Proxy.SOCKS4)
                                            .host("proxy")
                                            .port(8080)
                                            .nonProxyHosts("localhost")
                                            .connectTimeoutMillis(20_000)) ①
                        .connectNow();

        connection.onDispose()
                  .block();
    }
}
```

① Configures the connection establishment timeout to 20 seconds.

Metrics

The TCP client supports built-in integration with `Micrometer`. It exposes all metrics with a prefix of `reactor.netty.tcp.client`.

The following table provides information for the TCP client metrics:

metric name	type	description
reactor.netty.tcp.client.data.received	DistributionSummary	Amount of the data received, in bytes. See Data Received
reactor.netty.tcp.client.data.sent	DistributionSummary	Amount of the data sent, in bytes. See Data Sent
reactor.netty.tcp.client.errors	Counter	Number of errors that occurred. See Errors Count
reactor.netty.tcp.client.tls.handshake.time	Timer	Time spent for TLS handshake. See Tls Handshake Time
reactor.netty.tcp.client.connect.time	Timer	Time spent for connecting to the remote address. See Connect Time

metric name	type	description
reactor.netty.tcp.client.address.resolver	Timer	Time spent for resolving the address. See Hostname Resolution Time

These additional metrics are also available:

Pooled `ConnectionProvider` metrics

metric name	type	description
reactor.netty.connection.provider.total.connections	Gauge	The number of all connections, active or idle. See Total Connections
reactor.netty.connection.provider.active.connections	Gauge	The number of the connections that have been successfully acquired and are in active use. See Active Connections
reactor.netty.connection.provider.max.connections	Gauge	The maximum number of active connections that are allowed. See Max Connections
reactor.netty.connection.provider.idle.connections	Gauge	The number of the idle connections. See Idle Connections
reactor.netty.connection.provider.pending.connections	Gauge	The number of requests that are waiting for a connection. See Pending Connections
reactor.netty.connection.provider.pending.connections.time	Timer	Time spent in pending acquire a connection from the connection pool. See Pending Connections Time
reactor.netty.connection.provider.max.pending.connections	Gauge	The maximum number of requests that will be queued while waiting for a ready connection. See Max Pending Connections

`ByteBufAllocator` metrics

metric name	type	description
reactor.netty.bytebuf.allocator.used.heap.memory	Gauge	The number of bytes reserved by heap buffer allocator. See Used Heap Memory

metric name	type	description
reactor.netty.bytebuf.allocator.used.direct.memory	Gauge	The number of bytes reserved by direct buffer allocator. See Used Direct Memory
reactor.netty.bytebuf.allocator.heap.arenas	Gauge	The number of heap arenas (when `PooledByteBufAllocator`). See Heap Arenas
reactor.netty.bytebuf.allocator.direct.arenas	Gauge	The number of direct arenas (when `PooledByteBufAllocator`). See Direct Arenas
reactor.netty.bytebuf.allocator.threadlocal.caches	Gauge	The number of thread local caches (when `PooledByteBufAllocator`). See Thread Local Caches
reactor.netty.bytebuf.allocator.small.cache.size	Gauge	The size of the small cache (when `PooledByteBufAllocator`). See Small Cache Size
reactor.netty.bytebuf.allocator.normal.cache.size	Gauge	The size of the normal cache (when `PooledByteBufAllocator`). See Normal Cache Size
reactor.netty.bytebuf.allocator.chunk.size	Gauge	The chunk size for an arena (when `PooledByteBufAllocator`). See Chunk Size
reactor.netty.bytebuf.allocator.active.heap.memory	Gauge	The actual bytes consumed by in-use buffers allocated from heap buffer pools (when `PooledByteBufAllocator`). See Active Heap Memory
reactor.netty.bytebuf.allocator.active.direct.memory	Gauge	The actual bytes consumed by in-use buffers allocated from direct buffer pools (when `PooledByteBufAllocator`). See Active Direct Memory

`EventLoop` metrics

metric name	type	description
reactor.netty.eventloop.pending.tasks	Gauge	The number of tasks that are pending for processing on an event loop. See Pending Tasks

The following example enables that integration:

github.com/reactor/reactor-netty/tree/main/reactor-netty-examples/src/main/java/reactor/netty/

examples/documentation/tcp/client/metrics/Application.java

```
import reactor.netty.Connection;
import reactor.netty.tcp.TcpClient;

public class Application {

    public static void main(String[] args) {
        Connection connection =
                TcpClient.create()
                         .host("example.com")
                         .port(80)
                         .metrics(true) ①
                         .connectNow();

        connection.onDispose()
                  .block();
    }
}
```

① Enables the built-in integration with Micrometer

When TCP client metrics are needed for an integration with a system other than `Micrometer` or you want to provide your own integration with `Micrometer`, you can provide your own metrics recorder, as follows:

github.com/reactor/reactor-netty/tree/main/reactor-netty-examples/src/main/java/reactor/netty/examples/documentation/tcp/client/metrics/custom/Application.java

```
import reactor.netty.Connection;
import reactor.netty.channel.ChannelMetricsRecorder;
import reactor.netty.tcp.TcpClient;

import java.net.SocketAddress;
import java.time.Duration;

public class Application {

    public static void main(String[] args) {
        Connection connection =
                TcpClient.create()
                         .host("example.com")
                         .port(80)
                         .metrics(true, CustomChannelMetricsRecorder::new) ①
                         .connectNow();

        connection.onDispose()
                  .block();
    }
```

① Enables TCP client metrics and provides `ChannelMetricsRecorder` implementation.

Tracing

The TCP client supports built-in integration with `Micrometer Tracing`.

The following table provides information for the TCP client spans:

contextual name	description
hostname resolution	Information and time spent for resolving the address. See Hostname Resolution Span.
connect	Information and time spent for connecting to the remote address. See Connect Span.
tls handshake	Information and time spent for TLS handshake. See Tls Handshake Span.

The following example enables that integration. This concrete example uses `Brave` and reports the information to `Zipkin`. See the `Micrometer Tracing` documentation for `OpenTelemetry` setup.

github.com/reactor/reactor-netty/tree/main/reactor-netty-examples/src/main/java/reactor/netty/examples/documentation/tcp/client/tracing/Application.java

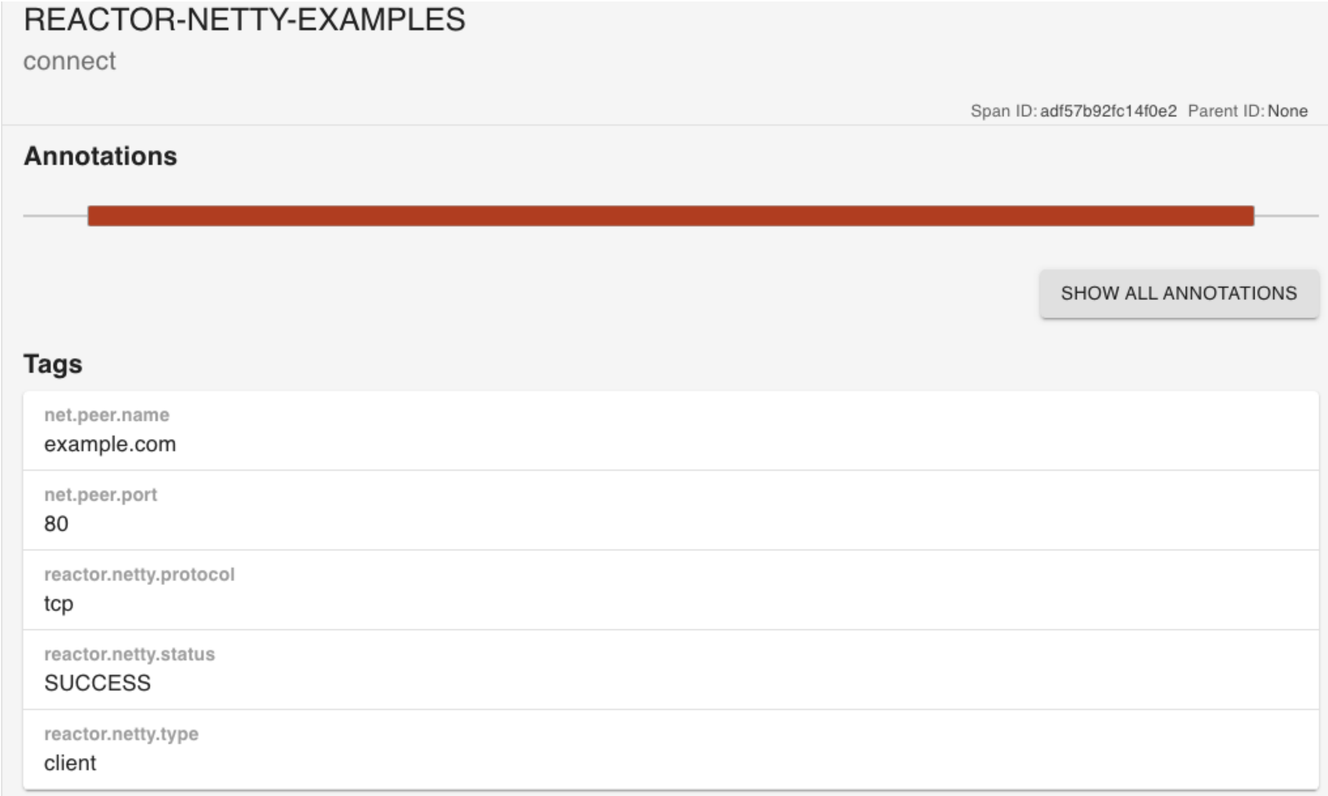
```
import brave.Tracing;
import brave.propagation.StrictCurrentTraceContext;
import brave.sampler.Sampler;
import io.micrometer.tracing.CurrentTraceContext;
import io.micrometer.tracing.Tracer;
import io.micrometer.tracing.brave.bridge.BraveBaggageManager;
import io.micrometer.tracing.brave.bridge.BraveCurrentTraceContext;
import io.micrometer.tracing.brave.bridge.BraveTracer;
import reactor.netty.Connection;
import reactor.netty.observability.ReactorNettyTracingObservationHandler;
import reactor.netty.tcp.TcpClient;
import zipkin2.reporter.brave.AsyncZipkinSpanHandler;
import zipkin2.reporter.urlconnection.URLConnectionSender;

import static reactor.netty.Metrics.OBSERVATION_REGISTRY;

public class Application {

    public static void main(String[] args) {
        init(); ①

        Connection connection =
                TcpClient.create()
                         .host("example.com")
                         .port(80)
                         .metrics(true) ②
```

```
                        .connectNow();

        connection.onDispose()
                  .block();
    }

    /**
     * This setup is based on
     * <a
href="https://micrometer.io/docs/tracing#_micrometer_tracing_brave_setup">Micrometer
Tracing Brave Setup</a>.
     */
    static void init() {
        AsyncZipkinSpanHandler spanHandler = AsyncZipkinSpanHandler
                .create(URLConnectionSender.create
("http://localhost:9411/api/v2/spans"));

        StrictCurrentTraceContext braveCurrentTraceContext =
StrictCurrentTraceContext.create();

        CurrentTraceContext bridgeContext = new BraveCurrentTraceContext
(braveCurrentTraceContext);

        Tracing tracing =
                Tracing.newBuilder()
                       .currentTraceContext(braveCurrentTraceContext)
                       .supportsJoin(false)
                       .traceId128Bit(true)
                       .sampler(Sampler.ALWAYS_SAMPLE)
                       .addSpanHandler(spanHandler)
                       .localServiceName("reactor-netty-examples")
                       .build();

        brave.Tracer braveTracer = tracing.tracer();

        Tracer tracer = new BraveTracer(braveTracer, bridgeContext, new
BraveBaggageManager());

        OBSERVATION_REGISTRY.observationConfig()
                            .observationHandler(new
ReactorNettyTracingObservationHandler(tracer));
    }
}
```

① Initializes Brave, Zipkin, and the Observation registry.

② Enables the built-in integration with Micrometer.

The result in `Zipkin` looks like:



Access Current Observation

Project Micrometer provides `a library` that assists with context propagation across different types of context mechanisms such as `ThreadLocal`, `Reactor Context` and others.

The following example shows how to use this library in a custom `ChannelHandler`:

github.com/reactor/reactor-netty/tree/main/reactor-netty-examples/src/main/java/reactor/netty/examples/documentation/tcp/client/tracing/custom/Application.java

```
import brave.Tracing;
import brave.propagation.StrictCurrentTraceContext;
import brave.sampler.Sampler;
import io.micrometer.context.ContextSnapshot;
import io.micrometer.context.ContextSnapshotFactory;
import io.micrometer.tracing.CurrentTraceContext;
import io.micrometer.tracing.Tracer;
import io.micrometer.tracing.brave.bridge.BraveBaggageManager;
import io.micrometer.tracing.brave.bridge.BraveCurrentTraceContext;
import io.micrometer.tracing.brave.bridge.BraveTracer;
import io.netty.channel.ChannelHandler;
import io.netty.channel.ChannelHandlerContext;
import io.netty.channel.ChannelOutboundHandlerAdapter;
import io.netty.channel.ChannelPromise;
import reactor.netty.Connection;
import reactor.netty.observability.ReactorNettyTracingObservationHandler;
import reactor.netty.tcp.TcpClient;
import zipkin2.reporter.brave.AsyncZipkinSpanHandler;
import zipkin2.reporter.urlconnection.URLConnectionSender;
```

```java
import java.net.SocketAddress;

import static reactor.netty.Metrics.OBSERVATION_REGISTRY;

public class Application {

    public static void main(String[] args) {
        init(); ①

        Connection connection =
                TcpClient.create()
                         .host("example.com")
                         .port(80)
                         .metrics(true) ②
                         .doOnChannelInit((observer, channel, address) -> channel
.pipeline().addFirst(
                                 "custom-channel-handler",
CustomChannelOutboundHandler.INSTANCE)) ③
                         .connectNow();

        connection.onDispose()
                  .block();
    }

    static final class CustomChannelOutboundHandler extends
ChannelOutboundHandlerAdapter {

        static final ChannelHandler INSTANCE = new CustomChannelOutboundHandler();

        @Override
        public boolean isSharable() {
            return true;
        }

        @Override
        @SuppressWarnings({"FutureReturnValueIgnored", "try"})
        public void connect(ChannelHandlerContext ctx, SocketAddress remoteAddress,
SocketAddress localAddress, ChannelPromise promise) {
            try (ContextSnapshot.Scope scope = ContextSnapshotFactory.builder().
build().setThreadLocalsFrom(ctx.channel())) {
                System.out.println("Current Observation in Scope: " +
OBSERVATION_REGISTRY.getCurrentObservation());
                //"FutureReturnValueIgnored" this is deliberate
                ctx.connect(remoteAddress, localAddress, promise);
            }
            System.out.println("Current Observation: " + OBSERVATION_REGISTRY
.getCurrentObservation());
        }
    }
```

① Initializes Brave, Zipkin, and the Observation registry.

② Enables the built-in integration with Micrometer.

③ Custom `ChannelHandler` that uses context propagation library. This concrete example overrides only `ChannelOutboundHandlerAdapter#connect`, if it is needed, the same logic can be used for the rest of the methods. Also, this concrete example sets all `ThreadLocal` values for which there is a value in the given `Channel`, if another behaviour is needed please check `context propagation library API`. For example, you may want to set only some of the `ThreadLocal` values.

> When you enable Reactor Netty tracing within a framework, you may need to let Reactor Netty use the `ObservationRegistry` created by this framework. For this purpose you need to invoke `reactor.netty.Metrics#observationRegistry`. You may also need to configure the Reactor Netty `ObservationHandlers` using the API provided by the framework.

Unix Domain Sockets

The `TCP` client supports Unix Domain Sockets (UDS) when native transport is in use.

The following example shows how to use UDS support:

github.com/reactor/reactor-netty/tree/main/reactor-netty-examples/src/main/java/reactor/netty/examples/documentation/tcp/client/uds/Application.java

```
import io.netty.channel.unix.DomainSocketAddress;
import reactor.netty.Connection;
import reactor.netty.tcp.TcpClient;

public class Application {

    public static void main(String[] args) {
        Connection connection =
                TcpClient.create()
                         .remoteAddress(() -> new DomainSocketAddress(
"/tmp/test.sock")) ①
                         .connectNow();

        connection.onDispose()
                  .block();
    }
}
```

① Specifies `DomainSocketAddress` that will be used

Host Name Resolution

By default, the `TcpClient` uses Netty's domain name lookup mechanism that resolves a domain name asynchronously. This is as an alternative of the JVM's built-in blocking resolver.

When you need to change the default settings, you can configure the `TcpClient` as follows:

github.com/reactor/reactor-netty/tree/main/reactor-netty-examples/src/main/java/reactor/netty/examples/documentation/tcp/client/resolver/Application.java

```
import reactor.netty.Connection;
import reactor.netty.tcp.TcpClient;

import java.time.Duration;

public class Application {

    public static void main(String[] args) {
        Connection connection =
                TcpClient.create()
                         .host("example.com")
                         .port(80)
                         .resolver(spec -> spec.queryTimeout(Duration.ofMillis(500)))
①
                         .connectNow();

        connection.onDispose()
                  .block();
    }
}
```

① The timeout of each DNS query performed by this resolver will be 500ms.

The following listing shows the available configurations. Additionally, `TCP fallback` is enabled by default.

Configuration name	Description
`bindAddressSupplier`	The supplier of the local address to bind to.
`cacheMaxTimeToLive`	The max time to live of the cached DNS resource records (resolution: seconds). If the time to live of the DNS resource record returned by the DNS server is greater than this max time to live, this resolver ignores the time to live from the DNS server and uses use this max time to live. Default to `Integer.MAX_VALUE`.
`cacheMinTimeToLive`	The min time to live of the cached DNS resource records (resolution: seconds). If the time to live of the DNS resource record returned by the DNS server is less than this min time to live, this resolver ignores the time to live from the DNS server and uses this min time to live. Default: 0.

Configuration name	Description
`cacheNegativeTimeToLive`	The time to live of the cache for the failed DNS queries (resolution: seconds). Default: 0.
`completeOncePreferredResolved`	When this setting is enabled, the resolver notifies as soon as all queries for the preferred address type are complete. When this setting is disabled, the resolver notifies when all possible address types are complete. This configuration is applicable for `DnsNameResolver#resolveAll(String)`. By default, this setting is enabled.
`disableOptionalRecord`	Disables the automatic inclusion of an optional record that tries to give a hint to the remote DNS server about how much data the resolver can read per response. By default, this setting is enabled.
`disableRecursionDesired`	Specifies whether this resolver has to send a DNS query with the recursion desired (RD) flag set. By default, this setting is enabled.
`dnsAddressResolverGroupProvider`	Sets a custom function to create a `DnsAddressResolverGroup` given a `DnsNameResolverBuilder`
`hostsFileEntriesResolver`	Sets a custom `HostsFileEntriesResolver` to be used for hosts file entries. Default: `DefaultHostsFileEntriesResolver`.
`maxPayloadSize`	Sets the capacity of the datagram packet buffer (in bytes). Default: 4096.
`maxQueriesPerResolve`	Sets the maximum allowed number of DNS queries to send when resolving a host name. Default: 16.
`ndots`	Sets the number of dots that must appear in a name before an initial absolute query is made. Default: -1 (to determine the value from the OS on Unix or use a value of 1 otherwise).
`queryTimeout`	Sets the timeout of each DNS query performed by this resolver (resolution: milliseconds). Default: 5000.
`resolveCache`	The cache to use to store resolved DNS entries.
`resolvedAddressTypes`	The list of the protocol families of the resolved address.

Configuration name	Description
`retryTcpOnTimeout`	Specifies whether this resolver will also fallback to TCP if a timeout is detected. By default, the resolver will only try to use TCP if the response is marked as truncated.
`roundRobinSelection`	Enables an `AddressResolverGroup` of `DnsNameResolver` that supports random selection of destination addresses if multiple are provided by the nameserver. See `RoundRobinDnsAddressResolverGroup`. Default: `DnsAddressResolverGroup`
`runOn`	Performs the communication with the DNS servers on the given `LoopResources`. By default, the LoopResources specified on the client level are used.
`searchDomains`	The list of search domains of the resolver. By default, the effective search domain list is populated by using the system DNS search domains.
`trace`	A specific logger and log level to be used by this resolver when generating detailed trace information in case of resolution failure.

Sometimes, you may want to switch to the JVM built-in resolver. To do so, you can configure the `TcpClient` as follows:

github.com/reactor/reactor-netty/tree/main/reactor-netty-examples/src/main/java/reactor/netty/examples/documentation/tcp/client/resolver/custom/Application.java

```java
import io.netty.resolver.DefaultAddressResolverGroup;
import reactor.netty.Connection;
import reactor.netty.tcp.TcpClient;

public class Application {

    public static void main(String[] args) {
        Connection connection =
                TcpClient.create()
                         .host("example.com")
                         .port(80)
                         .resolver(DefaultAddressResolverGroup.INSTANCE) ①
                         .connectNow();

        connection.onDispose()
                  .block();
    }
}
```

① Sets the JVM built-in resolver.

Chapter 5. HTTP Server

Reactor Netty provides the easy-to-use and easy-to-configure HttpServer class. It hides most of the Netty functionality that is needed in order to create a HTTP server and adds Reactive Streams backpressure.

Starting and Stopping

To start an HTTP server, you must create and configure a HttpServer instance. By default, the host is configured for any local address, and the system picks up an ephemeral port when the bind operation is invoked. The following example shows how to create an HttpServer instance:

github.com/reactor/reactor-netty/tree/main/reactor-netty-examples/src/main/java/reactor/netty/examples/documentation/http/server/create/Application.java

```
import reactor.netty.DisposableServer;
import reactor.netty.http.server.HttpServer;

public class Application {

    public static void main(String[] args) {
        DisposableServer server =
                HttpServer.create()   ①
                          .bindNow(); ②

        server.onDispose()
              .block();
    }
}
```

① Creates an HttpServer instance ready for configuring.

② Starts the server in a blocking fashion and waits for it to finish initializing.

The returned DisposableServer offers a simple server API, including disposeNow(), which shuts the server down in a blocking fashion.

Host and Port

To serve on a specific host and port, you can apply the following configuration to the HTTP server:

github.com/reactor/reactor-netty/tree/main/reactor-netty-examples/src/main/java/reactor/netty/examples/documentation/http/server/address/Application.java

```java
import reactor.netty.DisposableServer;
import reactor.netty.http.server.HttpServer;

public class Application {

    public static void main(String[] args) {
        DisposableServer server =
                HttpServer.create()
                          .host("localhost") ①
                          .port(8080)        ②
                          .bindNow();

        server.onDispose()
              .block();
    }
}
```

① Configures the `HTTP` server host

② Configures the `HTTP` server port

To serve on multiple addresses, after having configured the `HttpServer` you can bind it multiple times to obtain separate `DisposableServer`s. All created servers will share resources such as `LoopResources` because they use the same configuration instance under the hood.

github.com/reactor/reactor-netty/tree/main/reactor-netty-examples/src/main/java/reactor/netty/examples/documentation/http/server/address/MultiAddressApplication.java

```java
import reactor.core.publisher.Mono;
import reactor.netty.DisposableServer;
import reactor.netty.http.server.HttpServer;

public class MultiAddressApplication {
    public static void main(String[] args) {
        HttpServer httpServer = HttpServer.create();
        DisposableServer server1 = httpServer
                .host("localhost") ①
                .port(8080)        ②
                .bindNow();

        DisposableServer server2 = httpServer
                .host("0.0.0.0") ③
                .port(8081)      ④
                .bindNow();

        Mono.when(server1.onDispose(), server2.onDispose())
                .block();
    }
}
```

① Configures the first HTTP server host

② Configures the first HTTP server port

③ Configures the second HTTP server host

④ Configures the second HTTP server port

Eager Initialization

By default, the initialization of the HttpServer resources happens on demand. This means that the bind operation absorbs the extra time needed to initialize and load:

- the event loop groups
- the native transport libraries (when native transport is used)
- the native libraries for the security (in case of OpenSsl)

When you need to preload these resources, you can configure the HttpServer as follows:

github.com/reactor/reactor-netty/tree/main/reactor-netty-examples/src/main/java/reactor/netty/examples/documentation/http/server/warmup/Application.java

```
import reactor.netty.DisposableServer;
import reactor.netty.http.server.HttpServer;

public class Application {

    public static void main(String[] args) {
        HttpServer httpServer =
                HttpServer.create()
                          .handle((request, response) -> request.receive().then());

        httpServer.warmup() ①
                  .block();

        DisposableServer server = httpServer.bindNow();

        server.onDispose()
              .block();
    }
}
```

① Initialize and load the event loop groups, the native transport libraries and the native libraries for the security

Routing HTTP

Defining routes for the HTTP server requires configuring the provided HttpServerRoutes builder. The following example shows how to do so:

github.com/reactor/reactor-netty/tree/main/reactor-netty-examples/src/main/java/reactor/netty/examples/documentation/http/server/routing/Application.java

```
import reactor.core.publisher.Mono;
import reactor.netty.DisposableServer;
import reactor.netty.http.server.HttpServer;

public class Application {

    public static void main(String[] args) {
        DisposableServer server =
                HttpServer.create()
                          .route(routes ->
                              routes.get("/hello",        ①
                                          (request, response) -> response.sendString
(Mono.just("Hello World!")))
                                    .post("/echo",        ②
                                          (request, response) -> response.send(request
.receive().retain()))
                                    .get("/path/{param}", ③
                                          (request, response) -> response.sendString
(Mono.just(request.param("param"))))
                                    .ws("/ws",            ④
                                          (wsInbound, wsOutbound) -> wsOutbound.send
(wsInbound.receive().retain())))
                          .bindNow();

        server.onDispose()
              .block();
    }
}
```

① Serves a `GET` request to `/hello` and returns `Hello World!`

② Serves a `POST` request to `/echo` and returns the received request body as a response.

③ Serves a `GET` request to `/path/{param}` and returns the value of the path parameter.

④ Serves websocket to `/ws` and returns the received incoming data as outgoing data.

> **i** The server routes are unique and only the first matching in order of declaration is invoked.

SSE

The following code shows how you can configure the `HTTP` server to serve `Server-Sent Events`:

github.com/reactor/reactor-netty/tree/main/reactor-netty-examples/src/main/java/reactor/netty/examples/documentation/http/server/sse/Application.java

```java
import com.fasterxml.jackson.databind.ObjectMapper;
import io.netty.buffer.ByteBuf;
import io.netty.buffer.ByteBufAllocator;
import org.reactivestreams.Publisher;
import reactor.core.publisher.Flux;
import reactor.netty.DisposableServer;
import reactor.netty.http.server.HttpServer;
import reactor.netty.http.server.HttpServerRequest;
import reactor.netty.http.server.HttpServerResponse;

import java.io.ByteArrayOutputStream;
import java.nio.charset.Charset;
import java.time.Duration;
import java.util.function.BiFunction;

public class Application {

    public static void main(String[] args) {
        DisposableServer server =
                HttpServer.create()
                          .route(routes -> routes.get("/sse", serveSse()))
                          .bindNow();

        server.onDispose()
              .block();
    }

    /**
     * Prepares SSE response.
     * The "Content-Type" is "text/event-stream".
     * The flushing strategy is "flush after every element" emitted by the provided
Publisher.
     */
    private static BiFunction<HttpServerRequest, HttpServerResponse, Publisher<Void>>
serveSse() {
        Flux<Long> flux = Flux.interval(Duration.ofSeconds(10));
        return (request, response) ->
                response.sse()
                        .send(flux.map(Application::toByteBuf), b -> true);
    }

    /**
     * Transforms the Object to ByteBuf following the expected SSE format.
     */
    private static ByteBuf toByteBuf(Object any) {
        ByteArrayOutputStream out = new ByteArrayOutputStream();
        try {
            out.write("data: ".getBytes(Charset.defaultCharset()));
            MAPPER.writeValue(out, any);
            out.write("\n\n".getBytes(Charset.defaultCharset()));
```

```
        }
        catch (Exception e) {
            throw new RuntimeException(e);
        }
        return ByteBufAllocator.DEFAULT
                               .buffer()
                               .writeBytes(out.toByteArray());
    }

    private static final ObjectMapper MAPPER = new ObjectMapper();
}
```

Static Resources

The following code shows how you can configure the `HTTP` server to serve static resources:

github.com/reactor/reactor-netty/tree/main/reactor-netty-examples/src/main/java/reactor/netty/examples/documentation/http/server/staticresources/Application.java

```
import reactor.netty.DisposableServer;
import reactor.netty.http.server.HttpServer;

import java.net.URISyntaxException;
import java.nio.file.Path;
import java.nio.file.Paths;

public class Application {

    public static void main(String[] args) throws URISyntaxException {
        Path file = Paths.get(Application.class.getResource("/logback.xml").toURI());
        DisposableServer server =
                HttpServer.create()
                          .route(routes -> routes.file("/index.html", file))
                          .bindNow();

        server.onDispose()
              .block();
    }
}
```

Writing Data

To send data to a connected client, you must attach an I/O handler by using either `handle(…)` or `route(…)`. The I/O handler has access to `HttpServerResponse`, to be able to write data. The following example uses the `handle(…)` method:

github.com/reactor/reactor-netty/tree/main/reactor-netty-examples/src/main/java/reactor/netty/examples/documentation/http/server/send/Application.java

```java
import reactor.core.publisher.Mono;
import reactor.netty.DisposableServer;
import reactor.netty.http.server.HttpServer;

public class Application {

    public static void main(String[] args) {
        DisposableServer server =
                HttpServer.create()
                          .handle((request, response) -> response.sendString(Mono.
just("hello"))) ①
                          .bindNow();

        server.onDispose()
              .block();
    }
}
```

① Sends `hello` string to the connected clients

Adding Headers and Other Metadata

When you send data to the connected clients, you may need to send additional headers, cookies, status code, and other metadata. You can provide this additional metadata by using `HttpServerResponse`. The following example shows how to do so:

github.com/reactor/reactor-netty/tree/main/reactor-netty-examples/src/main/java/reactor/netty/examples/documentation/http/server/send/headers/Application.java

```java
import io.netty.handler.codec.http.HttpHeaderNames;
import io.netty.handler.codec.http.HttpResponseStatus;
import reactor.core.publisher.Mono;
import reactor.netty.DisposableServer;
import reactor.netty.http.server.HttpServer;

public class Application {

    public static void main(String[] args) {
        DisposableServer server =
                HttpServer.create()
                          .route(routes ->
                              routes.get("/hello",
                                  (request, response) ->
                                      response.status(HttpResponseStatus.OK)
                                              .header(HttpHeaderNames.CONTENT_LENGTH,
"12")
                                              .sendString(Mono.just("Hello World!"))))
                          .bindNow();

        server.onDispose()
              .block();
    }
}
```

Compression

You can configure the `HTTP` server to send a compressed response, depending on the request header `Accept-Encoding`.

`Reactor Netty` provides three different strategies for compressing the outgoing data:

- `compress(boolean)`: Depending on the boolean that is provided, the compression is enabled (`true`) or disabled (`false`).
- `compress(int)`: The compression is performed once the response size exceeds the given value (in bytes).
- `compress(BiPredicate<HttpServerRequest, HttpServerResponse>)`: The compression is performed if the predicate returns `true`.

The following example uses the `compress` method (set to `true`) to enable compression:

github.com/reactor/reactor-netty/tree/main/reactor-netty-examples/src/main/java/reactor/netty/examples/documentation/http/server/compression/Application.java

```java
import reactor.netty.DisposableServer;
import reactor.netty.http.server.HttpServer;

import java.net.URISyntaxException;
import java.nio.file.Path;
import java.nio.file.Paths;

public class Application {

    public static void main(String[] args) throws URISyntaxException {
        Path file = Paths.get(Application.class.getResource("/logback.xml").toURI());
        DisposableServer server =
                HttpServer.create()
                          .compress(true)
                          .route(routes -> routes.file("/index.html", file))
                          .bindNow();

        server.onDispose()
              .block();
    }
}
```

Consuming Data

To receive data from a connected client, you must attach an I/O handler by using either `handle(…)` or `route(…)`. The I/O handler has access to `HttpServerRequest`, to be able to read data.

The following example uses the `handle(…)` method:

github.com/reactor/reactor-netty/tree/main/reactor-netty-examples/src/main/java/reactor/netty/examples/documentation/http/server/read/Application.java

```java
import reactor.netty.DisposableServer;
import reactor.netty.http.server.HttpServer;

public class Application {

    public static void main(String[] args) {
        DisposableServer server =
                HttpServer.create()
                          .handle((request, response) -> request.receive().then()) ①
                          .bindNow();

        server.onDispose()
              .block();
    }
}
```

① Receives data from the connected clients

Reading Headers, URI Params, and other Metadata

When you receive data from the connected clients, you might need to check request headers, parameters, and other metadata. You can obtain this additional metadata by using `HttpServerRequest`. The following example shows how to do so:

github.com/reactor/reactor-netty/tree/main/reactor-netty-examples/src/main/java/reactor/netty/examples/documentation/http/server/read/headers/Application.java

```
import reactor.core.publisher.Mono;
import reactor.netty.DisposableServer;
import reactor.netty.http.server.HttpServer;

public class Application {

    public static void main(String[] args) {
        DisposableServer server =
                HttpServer.create()
                          .route(routes ->
                              routes.get("/{param}",
                                  (request, response) -> {
                                      if (request.requestHeaders().contains("Some-
Header")) {
                                          return response.sendString(Mono.just(
request.param("param")));
                                      }
                                      return response.sendNotFound();
                                  }))
                          .bindNow();

        server.onDispose()
              .block();
    }
}
```

Reading Post Form or Multipart Data

When you receive data from the connected clients, you might want to access `POST form` (`application/x-www-form-urlencoded`) or `multipart` (`multipart/form-data`) data. You can obtain this data by using `HttpServerRequest`.

github.com/reactor/reactor-netty/tree/main/reactor-netty-examples/src/main/java/reactor/netty/examples/documentation/http/server/multipart/Application.java

```java
import reactor.core.publisher.Mono;
import reactor.netty.DisposableServer;
import reactor.netty.http.server.HttpServer;

public class Application {

    public static void main(String[] args) {
        DisposableServer server =
                HttpServer.create()
                          .route(routes ->
                              routes.post("/multipart", (request, response) ->
response.sendString(
                                      request.receiveForm() ①
                                             .flatMap(data -> Mono.just('[' + data
.getName() + ']')))))
                          .bindNow();

        server.onDispose()
              .block();
    }
}
```

① Receives POST form/multipart data.

When you need to change the default settings, you can configure the HttpServer or you can provide a configuration per request:

github.com/reactor/reactor-netty/tree/main/reactor-netty-examples/src/main/java/reactor/netty/examples/documentation/http/server/multipart/custom/Application.java

```java
import reactor.core.publisher.Mono;
import reactor.netty.DisposableServer;
import reactor.netty.http.server.HttpServer;

public class Application {

    public static void main(String[] args) {
        DisposableServer server =
                HttpServer.create()
                          .httpFormDecoder(builder -> builder.maxInMemorySize(0))
①
                          .route(routes ->
                              routes.post("/multipart", (request, response) ->
response.sendString(
                                      request.receiveForm(builder -> builder
.maxInMemorySize(256)) ②
                                             .flatMap(data -> Mono.just('[' + data
.getName() + ']')))))
                          .bindNow();

        server.onDispose()
              .block();
    }
}
```

① Configuration on the `HttpServer` that specifies that the data is stored on disk only.

② Configuration per request that specifies that if the data size exceed the specified size, the data is stored on the disk.

The following listing shows the available configurations:

Configuration name	Description
`baseDirectory`	Configures the directory where to store the data on the disk. Default to generated temp directory.
`charset`	Configures the `Charset` for the data. Default to `StandardCharsets#UTF_8`.
`maxInMemorySize`	Configures the maximum in-memory size per data i.e. the data is written on disk if the size is greater than `maxInMemorySize`, else it is in memory. If set to `-1` the entire contents is stored in memory. If set to `0` the entire contents is stored on disk. Default to `16kb`.
`maxSize`	Configures the maximum size per data. When the limit is reached, an exception is raised. If set to `-1` this means no limitation. Default to `-1` - unlimited.

Configuration name	Description
`scheduler`	Configures the scheduler to be used for offloading disk operations in the decoding phase. Default to `Schedulers#boundedElastic()`
`streaming`	When set to `true`, the data is streamed directly from the parsed input buffer stream, which means it is not stored either in memory or file. When `false`, parts are backed by in-memory and/or file storage. Default to `false`. **NOTE** that with streaming enabled, the provided data might not be in a complete state i.e. `HttpData#isCompleted()` has to be checked. Also note that enabling this property effectively ignores `maxInMemorySize`, `baseDirectory`, and `scheduler`.

Obtaining the Remote (Client) Address

In addition to the metadata that you can obtain from the request, you can also receive the `host (server)` address, the `remote (client)` address and the `scheme`. Depending on the chosen factory method, you can retrieve the information directly from the channel or by using the `Forwarded` or `X-Forwarded-*` `HTTP` request headers. The following example shows how to do so:

github.com/reactor/reactor-netty/tree/main/reactor-netty-examples/src/main/java/reactor/netty/examples/documentation/http/server/clientaddress/Application.java

```java
import reactor.core.publisher.Mono;
import reactor.netty.DisposableServer;
import reactor.netty.http.server.HttpServer;

public class Application {

    public static void main(String[] args) {
        DisposableServer server =
                HttpServer.create()
                          .forwarded(true) ①
                          .route(routes ->
                              routes.get("/clientip",
                                  (request, response) ->
                                      response.sendString(Mono.just(request
.remoteAddress() ②

.getHostString()))))
                          .bindNow();

        server.onDispose()
              .block();
    }
}
```

① Specifies that the information about the connection is to be obtained from the `Forwarded` and `X-Forwarded-*` `HTTP` request headers, if possible.

② Returns the address of the remote (client) peer.

It is also possible to customize the behavior of the `Forwarded` or `X-Forwarded-*` header handler. The following example shows how to do so:

github.com/reactor/reactor-netty/tree/main/reactor-netty-examples/src/main/java/reactor/netty/examples/documentation/http/server/clientaddress/CustomForwardedHeaderHandlerApplication.java

```
import java.net.InetSocketAddress;

import reactor.core.publisher.Mono;
import reactor.netty.DisposableServer;
import reactor.netty.http.server.HttpServer;
import reactor.netty.transport.AddressUtils;

public class CustomForwardedHeaderHandlerApplication {

    public static void main(String[] args) {
        DisposableServer server =
                HttpServer.create()
                          .forwarded((connectionInfo, request) -> {  ①
                              String hostHeader = request.headers().get("X-Forwarded-
Host");
                              if (hostHeader != null) {
                                  String[] hosts = hostHeader.split(",", 2);
                                  InetSocketAddress hostAddress = AddressUtils
.createUnresolved(
                                      hosts[hosts.length - 1].trim(),
                                      connectionInfo.getHostAddress().getPort());
                                  connectionInfo = connectionInfo.withHostAddress
(hostAddress);
                              }
                              return connectionInfo;
                          })
                          .route(routes ->
                              routes.get("/clientip",
                                  (request, response) ->
                                      response.sendString(Mono.just(request
.remoteAddress() ②

.getHostString()))))
                          .bindNow();

        server.onDispose()
              .block();
    }
}
```

① Add a custom header handler.

② Returns the address of the remote (client) peer.

HTTP Request Decoder

By default, `Netty` configures some restrictions for the incoming requests, such as:

- The maximum length of the initial line.
- The maximum length of all headers.

- The maximum length of the content or each chunk.

For more information, see `HttpRequestDecoder` and `HttpServerUpgradeHandler`

By default, the `HTTP` server is configured with the following settings:

github.com/reactor/reactor-netty/tree/main/reactor-netty-http/src/main/java/reactor/netty/http/HttpDecoderSpec.java

```
    public static final int DEFAULT_MAX_INITIAL_LINE_LENGTH             = 4096;
    public static final int DEFAULT_MAX_HEADER_SIZE                     = 8192;
    /**
     * Default max chunk size.
     *
     * @deprecated as of 1.1.0. This will be removed in 2.0.0 as Netty 5 does not
support this configuration.
     */
    @Deprecated
    public static final int DEFAULT_MAX_CHUNK_SIZE                      = 8192;
    public static final boolean DEFAULT_VALIDATE_HEADERS                = true;
    public static final int DEFAULT_INITIAL_BUFFER_SIZE                 = 128;
    public static final boolean DEFAULT_ALLOW_DUPLICATE_CONTENT_LENGTHS = false;
```

github.com/reactor/reactor-netty/tree/main/reactor-netty-http/src/main/java/reactor/netty/http/server/HttpRequestDecoderSpec.java

```
    /**
     * The maximum length of the content of the HTTP/2.0 clear-text upgrade request.
     * By default, the server will reject an upgrade request with non-empty content,
     * because the upgrade request is most likely a GET request.
     */
    public static final int DEFAULT_H2C_MAX_CONTENT_LENGTH = 0;
```

When you need to change these default settings, you can configure the `HTTP` server as follows:

github.com/reactor/reactor-netty/tree/main/reactor-netty-examples/src/main/java/reactor/netty/examples/documentation/http/server/requestdecoder/Application.java

```java
import reactor.core.publisher.Mono;
import reactor.netty.DisposableServer;
import reactor.netty.http.server.HttpServer;

public class Application {

    public static void main(String[] args) {
        DisposableServer server =
                HttpServer.create()
                          .httpRequestDecoder(spec -> spec.maxHeaderSize(16384)) ①
                          .handle((request, response) -> response.sendString(Mono.
just("hello")))
                          .bindNow();

        server.onDispose()
              .block();
    }
}
```

① The maximum length of all headers will be `16384`. When this value is exceeded, a TooLongFrameException is raised.

Lifecycle Callbacks

The following lifecycle callbacks are provided to let you extend the `HttpServer`:

Callback	Description
`doOnBind`	Invoked when the server channel is about to bind.
`doOnBound`	Invoked when the server channel is bound.
`doOnChannelInit`	Invoked when initializing the channel.
`doOnConnection`	Invoked when a remote client is connected
`doOnUnbound`	Invoked when the server channel is unbound.

The following example uses the `doOnConnection` and `doOnChannelInit` callbacks:

github.com/reactor/reactor-netty/tree/main/reactor-netty-examples/src/main/java/reactor/netty/examples/documentation/http/server/lifecycle/Application.java

```java
import io.netty.handler.logging.LoggingHandler;
import io.netty.handler.timeout.ReadTimeoutHandler;
import reactor.netty.DisposableServer;
import reactor.netty.http.server.HttpServer;
import java.util.concurrent.TimeUnit;

public class Application {

    public static void main(String[] args) {
        DisposableServer server =
                HttpServer.create()
                          .doOnConnection(conn ->
                              conn.addHandlerFirst(new ReadTimeoutHandler(10,
TimeUnit.SECONDS))) ①
                          .doOnChannelInit((observer, channel, remoteAddress) ->
                              channel.pipeline()
                                     .addFirst(new LoggingHandler
("reactor.netty.examples")))    ②
                          .bindNow();

        server.onDispose()
              .block();
    }
}
```

① `Netty` pipeline is extended with `ReadTimeoutHandler` when a remote client is connected.

② `Netty` pipeline is extended with `LoggingHandler` when initializing the channel.

TCP-level Configuration

When you need to change configuration on the TCP level, you can use the following snippet to extend the default `TCP` server configuration:

github.com/reactor/reactor-netty/tree/main/reactor-netty-examples/src/main/java/reactor/netty/examples/documentation/http/server/channeloptions/Application.java

```java
import io.netty.channel.ChannelOption;
import reactor.netty.DisposableServer;
import reactor.netty.http.server.HttpServer;

public class Application {

    public static void main(String[] args) {
        DisposableServer server =
                HttpServer.create()
                          .option(ChannelOption.CONNECT_TIMEOUT_MILLIS, 10000)
                          .bindNow();

        server.onDispose()
              .block();
    }
}
```

See TCP Server for more detail about TCP-level configuration.

Wire Logger

Reactor Netty provides wire logging for when the traffic between the peers needs to be inspected. By default, wire logging is disabled. To enable it, you must set the logger `reactor.netty.http.server.HttpServer` level to `DEBUG` and apply the following configuration:

github.com/reactor/reactor-netty/tree/main/reactor-netty-examples/src/main/java/reactor/netty/examples/documentation/http/server/wiretap/Application.java

```java
import reactor.netty.DisposableServer;
import reactor.netty.http.server.HttpServer;

public class Application {

    public static void main(String[] args) {
        DisposableServer server =
                HttpServer.create()
                          .wiretap(true) ①
                          .bindNow();

        server.onDispose()
              .block();
    }
}
```

① Enables the wire logging

Wire Logger formatters

Reactor Netty supports 3 different formatters:

- AdvancedByteBufFormat#HEX_DUMP - the default

github.com/reactor/reactor-netty/tree/main/reactor-netty-core/src/main/java/reactor/netty/transport/logging/AdvancedByteBufFormat.java

```
    /**
     * When wire logging is enabled with this format, both events and content will be
logged.
     * The content will be in hex format.
     * <p>Examples:</p>
     * <pre>
     * {@code
     * reactor.netty.http.HttpTests - [d5230a14, L:/0:0:0:0:0:0:0:1:60267 -
R:/0:0:0:0:0:0:0:1:60269] REGISTERED
     * reactor.netty.http.HttpTests - [d5230a14, L:/0:0:0:0:0:0:0:1:60267 -
R:/0:0:0:0:0:0:0:1:60269] ACTIVE
     * reactor.netty.http.HttpTests - [d5230a14, L:/0:0:0:0:0:0:0:1:60267 -
R:/0:0:0:0:0:0:0:1:60269] READ: 145B
     *          +-------------------------------------------------+
     *          |  0  1  2  3  4  5  6  7  8  9  a  b  c  d  e  f |
     * +--------+-------------------------------------------------+----------------+
     * |00000000| 50 4f 53 54 20 2f 74 65 73 74 2f 57 6f 72 6c 64 |POST /test/World|
     * |00000010| 20 48 54 54 50 2f 31 2e 31 0d 0a 43 6f 6e 74 65 | HTTP/1.1..Conte|
     * |00000020| 6e 74 2d 54 79 70 65 3a 20 74 65 78 74 2f 70 6c |nt-Type: text/pl|
     * |00000030| 61 69 6e 0d 0a 75 73 65 72 2d 61 67 65 6e 74 3a |ain..user-agent:|
     * |00000040| 20 52 65 61 63 74 6f 72 4e 65 74 74 79 2f 64 65 | ReactorNetty/de|
     * ...
     * reactor.netty.http.HttpTests - [d5230a14, L:/0:0:0:0:0:0:0:1:60267 -
R:/0:0:0:0:0:0:0:1:60269] WRITE: 38B
     *          +-------------------------------------------------+
     *          |  0  1  2  3  4  5  6  7  8  9  a  b  c  d  e  f |
     * +--------+-------------------------------------------------+----------------+
     * |00000000| 48 54 54 50 2f 31 2e 31 20 32 30 30 20 4f 4b 0d |HTTP/1.1 200 OK.|
     * |00000010| 0a 63 6f 6e 74 65 6e 74 2d 6c 65 6e 67 74 68 3a |.content-length:|
     * |00000020| 20 30 0d 0a 0d 0a                               | 0....          |
     * +--------+-------------------------------------------------+----------------+
     * }
     * </pre>
     */
```

- AdvancedByteBufFormat#SIMPLE

github.com/reactor/reactor-netty/tree/main/reactor-netty-core/src/main/java/reactor/netty/transport/logging/AdvancedByteBufFormat.java

```
/**
 * When wire logging is enabled with this format, only the events will be logged.
 * <p>Examples:</p>
 * <pre>
 * {@code
 * reactor.netty.http.HttpTests - [230d3686, L:/0:0:0:0:0:0:0:1:60241 -
R:/0:0:0:0:0:0:0:1:60245] REGISTERED
 * reactor.netty.http.HttpTests - [230d3686, L:/0:0:0:0:0:0:0:1:60241 -
R:/0:0:0:0:0:0:0:1:60245] ACTIVE
 * reactor.netty.http.HttpTests - [230d3686, L:/0:0:0:0:0:0:0:1:60241 -
R:/0:0:0:0:0:0:0:1:60245] READ: 145B
 * reactor.netty.http.HttpTests - [230d3686, L:/0:0:0:0:0:0:0:1:60241 -
R:/0:0:0:0:0:0:0:1:60245] WRITE: 38B
 * }
 * </pre>
 */
```

- AdvancedByteBufFormat#TEXTUAL

github.com/reactor/reactor-netty/tree/main/reactor-netty-core/src/main/java/reactor/netty/transport/logging/AdvancedByteBufFormat.java

```
/**
 * When wire logging is enabled with this format, both events and content will be
logged.
 * The content will be in plain text format.
 * <p>Examples:</p>
 * <pre>
 * {@code
 * reactor.netty.http.HttpTests - [02c3db6c, L:/0:0:0:0:0:0:0:1:60317 -
R:/0:0:0:0:0:0:0:1:60319] REGISTERED
 * reactor.netty.http.HttpTests - [02c3db6c, L:/0:0:0:0:0:0:0:1:60317 -
R:/0:0:0:0:0:0:0:1:60319] ACTIVE
 * reactor.netty.http.HttpTests - [02c3db6c, L:/0:0:0:0:0:0:0:1:60317 -
R:/0:0:0:0:0:0:0:1:60319] READ: 145B POST /test/World HTTP/1.1
 * Content-Type: text/plain
 * user-agent: ReactorNetty/dev
 * ...
 * reactor.netty.http.HttpTests - [02c3db6c, L:/0:0:0:0:0:0:0:1:60317 -
R:/0:0:0:0:0:0:0:1:60319] WRITE: 38B HTTP/1.1 200 OK
 * content-length: 0
 * }
 * </pre>
 */
```

When you need to change the default formatter you can configure it as follows:

github.com/reactor/reactor-netty/tree/main/reactor-netty-examples/src/main/java/reactor/netty/

examples/documentation/http/server/wiretap/custom/Application.java

```
import io.netty.handler.logging.LogLevel;
import reactor.netty.DisposableServer;
import reactor.netty.http.server.HttpServer;
import reactor.netty.transport.logging.AdvancedByteBufFormat;

public class Application {

    public static void main(String[] args) {
        DisposableServer server =
                HttpServer.create()
                          .wiretap("logger-name", LogLevel.DEBUG,
AdvancedByteBufFormat.TEXTUAL) ①
                          .bindNow();

        server.onDispose()
              .block();
    }
}
```

① Enables the wire logging, AdvancedByteBufFormat#TEXTUAL is used for printing the content.

Event Loop Group

By default `Reactor Netty` uses an "Event Loop Group", where the number of the worker threads equals the number of processors available to the runtime on initialization (but with a minimum value of 4). When you need a different configuration, you can use one of the `LoopResources#create` methods.

The following listing shows the default configuration for the Event Loop Group:

github.com/reactor/reactor-netty/tree/main/reactor-netty-core/src/main/java/reactor/netty/ReactorNetty.java

```
    /**
     * Default worker thread count, fallback to available processor
     * (but with a minimum value of 4).
     */
    public static final String IO_WORKER_COUNT = "reactor.netty.ioWorkerCount";
    /**
     * Default selector thread count, fallback to -1 (no selector thread)
     * <p><strong>Note:</strong> In most use cases using a worker thread also as a
selector thread works well.
     * A possible use case for specifying a separate selector thread might be when the
worker threads are too busy
     * and connections cannot be accepted fast enough.
     * <p><strong>Note:</strong> Although more than 1 can be configured as a selector
thread count, in reality
     * only 1 thread will be used as a selector thread.
     */
    public static final String IO_SELECT_COUNT = "reactor.netty.ioSelectCount";
    /**
     * Default worker thread count for UDP, fallback to available processor
     * (but with a minimum value of 4).
     */
    public static final String UDP_IO_THREAD_COUNT =
"reactor.netty.udp.ioThreadCount";
    /**
     * Default quiet period that guarantees that the disposal of the underlying
LoopResources
     * will not happen, fallback to 2 seconds.
     */
    public static final String SHUTDOWN_QUIET_PERIOD =
"reactor.netty.ioShutdownQuietPeriod";
    /**
     * Default maximum amount of time to wait until the disposal of the underlying
LoopResources
     * regardless if a task was submitted during the quiet period, fallback to 15
seconds.
     */
    public static final String SHUTDOWN_TIMEOUT = "reactor.netty.ioShutdownTimeout";

    /**
     * Default value whether the native transport (epoll, kqueue) will be preferred,
     * fallback it will be preferred when available.
     */
    public static final String NATIVE = "reactor.netty.native";
```

If you need changes to these settings, you can apply the following configuration:

github.com/reactor/reactor-netty/tree/main/reactor-netty-examples/src/main/java/reactor/netty/examples/documentation/http/server/eventloop/Application.java

```java
import reactor.netty.DisposableServer;
import reactor.netty.http.server.HttpServer;
import reactor.netty.resources.LoopResources;

public class Application {

    public static void main(String[] args) {
        LoopResources loop = LoopResources.create("event-loop", 1, 4, true);

        DisposableServer server =
                HttpServer.create()
                          .runOn(loop)
                          .bindNow();

        server.onDispose()
              .block();
    }
}
```

Disposing Event Loop Group

- If you use the default `Event Loop Group` provided by Reactor Netty, invoke `HttpResources#disposeLoopsAndConnections/#disposeLoopsAndConnectionsLater` method.

> Disposing `HttpResources` means that every server/client that is using it, will not be able to use it anymore!

- If you use custom `LoopResources`, invoke `LoopResources#dispose/#disposeLater` method.

> Disposing the custom `LoopResources` means that every server/client that is configured to use it, will not be able to use it anymore!

SSL and TLS

When you need SSL or TLS, you can apply the configuration shown in the next example. By default, if `OpenSSL` is available, `SslProvider.OPENSSL` provider is used as a provider. Otherwise `SslProvider.JDK` is used. You can switch the provider by using `SslContextBuilder` or by setting `-Dio.netty.handler.ssl.noOpenSsl=true`.

The following example uses `SslContextBuilder`:

github.com/reactor/reactor-netty/tree/main/reactor-netty-examples/src/main/java/reactor/netty/examples/documentation/http/server/security/Application.java

```java
import reactor.netty.DisposableServer;
import reactor.netty.http.Http11SslContextSpec;
import reactor.netty.http.server.HttpServer;
```

```java
import java.io.File;

public class Application {

    public static void main(String[] args) {
        File cert = new File("certificate.crt");
        File key = new File("private.key");

        Http11SslContextSpec http11SslContextSpec = Http11SslContextSpec.forServer
(cert, key);

        DisposableServer server =
                HttpServer.create()
                          .secure(spec -> spec.sslContext(http11SslContextSpec))
                          .bindNow();

        server.onDispose()
              .block();
    }
}
```

Server Name Indication

You can configure the HTTP server with multiple SslContext mapped to a specific domain. An exact domain name or a domain name containing a wildcard can be used when configuring the SNI mapping.

The following example uses a domain name containing a wildcard:

github.com/reactor/reactor-netty/tree/main/reactor-netty-examples/src/main/java/reactor/netty/examples/documentation/http/server/sni/Application.java

```java
import io.netty.handler.ssl.SslContext;
import io.netty.handler.ssl.SslContextBuilder;
import reactor.netty.DisposableServer;
import reactor.netty.http.server.HttpServer;

import java.io.File;

public class Application {

    public static void main(String[] args) throws Exception {
        File defaultCert = new File("default_certificate.crt");
        File defaultKey = new File("default_private.key");

        File testDomainCert = new File("default_certificate.crt");
        File testDomainKey = new File("default_private.key");

        SslContext defaultSslContext = SslContextBuilder.forServer(defaultCert,
defaultKey).build();
```

```
        SslContext testDomainSslContext = SslContextBuilder.forServer(testDomainCert,
testDomainKey).build();

        DisposableServer server =
                HttpServer.create()
                          .secure(spec -> spec.sslContext(defaultSslContext)
                                              .addSniMapping("*.test.com",
                                                      testDomainSpec ->
testDomainSpec.sslContext(testDomainSslContext)))
                          .bindNow();

        server.onDispose()
              .block();
    }
}
```

HTTP Access Log

You can enable the HTTP access log either programmatically or by configuration. By default, it is disabled.

You can use -Dreactor.netty.http.server.accessLogEnabled=true to enable the HTTP access log by configuration.

You can use the following configuration (for Logback or similar logging frameworks) to have a separate HTTP access log file:

```
<appender name="accessLog" class="ch.qos.logback.core.FileAppender">
    <file>access_log.log</file>
    <encoder>
        <pattern>%msg%n</pattern>
    </encoder>
</appender>
<appender name="async" class="ch.qos.logback.classic.AsyncAppender">
    <appender-ref ref="accessLog" />
</appender>

<logger name="reactor.netty.http.server.AccessLog" level="INFO" additivity="false">
    <appender-ref ref="async"/>
</logger>
```

The following example enables it programmatically:

github.com/reactor/reactor-netty/tree/main/reactor-netty-examples/src/main/java/reactor/netty/examples/documentation/http/server/accessLog/Application.java

```
import reactor.netty.DisposableServer;
import reactor.netty.http.server.HttpServer;
```

```java
public class Application {

    public static void main(String[] args) {
        DisposableServer server =
                HttpServer.create()
                          .accessLog(true)
                          .bindNow();

        server.onDispose()
              .block();
    }
}
```

Calling this method takes precedence over the system property configuration.

By default, the logging format is Common Log Format, but you can specify a custom one as a parameter, as in the following example:

github.com/reactor/reactor-netty/tree/main/reactor-netty-examples/src/main/java/reactor/netty/examples/documentation/http/server/accessLog/CustomLogAccessFormatApplication.java

```java
import reactor.netty.DisposableServer;
import reactor.netty.http.server.HttpServer;
import reactor.netty.http.server.logging.AccessLog;

public class CustomLogAccessFormatApplication {

    public static void main(String[] args) {
        DisposableServer server =
                HttpServer.create()
                          .accessLog(true, x -> AccessLog.create("method={}, uri={}",
x.method(), x.uri()))
                          .bindNow();

        server.onDispose()
              .block();
    }
}
```

You can also filter `HTTP` access logs by using the `AccessLogFactory#createFilter` method, as in the following example:

github.com/reactor/reactor-netty/tree/main/reactor-netty-examples/src/main/java/reactor/netty/examples/documentation/http/server/accessLog/FilterLogAccessApplication.java

```java
import reactor.netty.DisposableServer;
import reactor.netty.http.server.HttpServer;
import reactor.netty.http.server.logging.AccessLogFactory;
```

```java
public class FilterLogAccessApplication {

    public static void main(String[] args) {
        DisposableServer server =
                HttpServer.create()
                          .accessLog(true, AccessLogFactory.createFilter(p -> !String
.valueOf(p.uri()).startsWith("/health/")))
                          .bindNow();

        server.onDispose()
              .block();
    }
}
```

Note that this method can take a custom format parameter too, as in this example:

github.com/reactor/reactor-netty/tree/main/reactor-netty-examples/src/main/java/reactor/netty/examples/documentation/http/server/accessLog/CustomFormatAndFilterAccessLogApplication.java.java

```java
import reactor.netty.DisposableServer;
import reactor.netty.http.server.HttpServer;
import reactor.netty.http.server.logging.AccessLog;
import reactor.netty.http.server.logging.AccessLogFactory;

public class CustomFormatAndFilterAccessLogApplication {

    public static void main(String[] args) {
        DisposableServer server =
                HttpServer.create()
                          .accessLog(true, AccessLogFactory.createFilter(p -> !String
.valueOf(p.uri()).startsWith("/health/"), ①
                                  x -> AccessLog.create("method={}, uri={}", x.
method(), x.uri()))) ②
                          .bindNow();

        server.onDispose()
              .block();
    }
}
```

① Specifies the filter predicate to use

② Specifies the custom format to apply

HTTP/2

By default, the HTTP server supports HTTP/1.1. If you need HTTP/2, you can get it through configuration. In addition to the protocol configuration, if you need `H2` but not `H2C (cleartext)`, you must also configure SSL.

> As Application-Layer Protocol Negotiation (ALPN) is not supported “out-of-the-box” by JDK8 (although some vendors backported ALPN to JDK8), you need an additional dependency to a native library that supports it — for example, `netty-tcnative-boringssl-static`.

The following listing presents a simple `H2` example:

github.com/reactor/reactor-netty/tree/main/reactor-netty-examples/src/main/java/reactor/netty/examples/documentation/http/server/http2/H2Application.java

```
import reactor.core.publisher.Mono;
import reactor.netty.DisposableServer;
import reactor.netty.http.Http2SslContextSpec;
import reactor.netty.http.HttpProtocol;
import reactor.netty.http.server.HttpServer;
import java.io.File;

public class H2Application {

    public static void main(String[] args) {
        File cert = new File("certificate.crt");
        File key = new File("private.key");

        Http2SslContextSpec http2SslContextSpec = Http2SslContextSpec.forServer(cert,
key);

        DisposableServer server =
                HttpServer.create()
                          .port(8080)
                          .protocol(HttpProtocol.H2)                     ①
                          .secure(spec -> spec.sslContext(http2SslContextSpec)) ②
                          .handle((request, response) -> response.sendString(Mono.
just("hello")))
                          .bindNow();

        server.onDispose()
              .block();
    }
}
```

① Configures the server to support only `HTTP/2`

② Configures `SSL`

The application should now behave as follows:

```
$ curl --http2 https://localhost:8080 -i
```

```
HTTP/2 200

hello
```

The following listing presents a simple `H2C` example:

github.com/reactor/reactor-netty/tree/main/reactor-netty-examples/src/main/java/reactor/netty/examples/documentation/http/server/http2/H2CApplication.java

```java
import reactor.core.publisher.Mono;
import reactor.netty.DisposableServer;
import reactor.netty.http.HttpProtocol;
import reactor.netty.http.server.HttpServer;

public class H2CApplication {

    public static void main(String[] args) {
        DisposableServer server =
                HttpServer.create()
                          .port(8080)
                          .protocol(HttpProtocol.H2C)
                          .handle((request, response) -> response.sendString(Mono.
just("hello")))
                          .bindNow();

        server.onDispose()
              .block();
    }
}
```

The application should now behave as follows:

```
$ curl --http2-prior-knowledge http://localhost:8080 -i
HTTP/2 200

hello
```

Protocol Selection

github.com/reactor/reactor-netty/tree/main/reactor-netty-http/src/main/java/reactor/netty/http/HttpProtocol.java

```java
 * @author Stephane Maldini
 */
public enum HttpProtocol {

    /**
```

```
     * The default supported HTTP protocol by HttpServer and HttpClient.
     */
    HTTP11,

    /**
     * HTTP/2.0 support with TLS
     * <p>If used along with HTTP/1.1 protocol, HTTP/2.0 will be the preferred
protocol.
     * While negotiating the application level protocol, HTTP/2.0 or HTTP/1.1 can be
chosen.
     * <p>If used without HTTP/1.1 protocol, HTTP/2.0 will always be offered as a
protocol
     * for communication with no fallback to HTTP/1.1.
     */
    H2,

    /**
     * HTTP/2.0 support with clear-text.
     * <p>If used along with HTTP/1.1 protocol, will support H2C "upgrade":
     * Request or consume requests as HTTP/1.1 first, looking for HTTP/2.0 headers
     * and {@literal Connection: Upgrade}. A server will typically reply a successful
     * 101 status if upgrade is successful or a fallback HTTP/1.1 response. When
     * successful the client will start sending HTTP/2.0 traffic.
     * <p>If used without HTTP/1.1 protocol, will support H2C "prior-knowledge":
Doesn't
     * require {@literal Connection: Upgrade} handshake between a client and server
but
     * fallback to HTTP/1.1 will not be supported.
     */
    H2C,

    /**
     * HTTP/3.0 support.
     * @since 1.2.0
     */
```

HTTP/3

By default, the `HTTP` server supports `HTTP/1.1`. If you need `HTTP/3`, you can get it through configuration. In addition to the protocol configuration, you need to add dependency to `io.netty.incubator:netty-incubator-codec-http3`.

The following listing presents a simple `HTTP3` example:

github.com/reactor/reactor-netty/tree/main/reactor-netty-examples/src/main/java/reactor/netty/examples/documentation/http/server/http3/Application.java

```
import reactor.core.publisher.Mono;
import reactor.netty.DisposableServer;
```

```
import reactor.netty.http.Http3SslContextSpec;
import reactor.netty.http.HttpProtocol;
import reactor.netty.http.server.HttpServer;

import java.io.File;
import java.time.Duration;

public class Application {

    public static void main(String[] args) throws Exception {
        File certChainFile = new File("certificate chain file");
        File keyFile = new File("private key file");

        Http3SslContextSpec serverCtx = Http3SslContextSpec.forServer(keyFile, null,
certChainFile);

        DisposableServer server =
                HttpServer.create()
                          .port(8080)
                          .protocol(HttpProtocol.HTTP3)                  ①
                          .secure(spec -> spec.sslContext(serverCtx))    ②
                          .idleTimeout(Duration.ofSeconds(5))
                          .http3Settings(spec -> spec.maxData(10000000) ③
                                                     .maxStreamDataBidirectionalLocal
(1000000)
                                                     .
maxStreamDataBidirectionalRemote(1000000)
                                                     .maxStreamsBidirectional(100))
                          .handle((request, response) -> response.header("server",
"reactor-netty")
                                                                  .sendString(Mono.
just("hello")))
                          .bindNow();

        server.onDispose()
              .block();
    }
}
```

① Configures the server to support only `HTTP/3`

② Configures `SSL`

③ Configures `HTTP/3` settings

The application should now behave as follows:

```
$ curl --http3 https://localhost:8080 -i
HTTP/3 200
server: reactor-netty
content-length: 5
```

```
hello
```

Metrics

The HTTP server supports built-in integration with `Micrometer`. It exposes all metrics with a prefix of `reactor.netty.http.server`.

The following table provides information for the HTTP server metrics:

metric name	type	description
reactor.netty.http.server.streams.active	Gauge	The number of active HTTP/2 streams. See Streams Active
reactor.netty.http.server.connections.active	Gauge	The number of http connections currently processing requests. See Connections Active
reactor.netty.http.server.connections.total	Gauge	The number of all opened connections. See Connections Total
reactor.netty.http.server.data.received	DistributionSummary	Amount of the data received, in bytes. See Data Received
reactor.netty.http.server.data.sent	DistributionSummary	Amount of the data sent, in bytes. See Data Sent
reactor.netty.http.server.errors	Counter	Number of errors that occurred. See Errors Count
reactor.netty.http.server.data.received.time	Timer	Time spent in consuming incoming data. See Http Server Data Received Time
reactor.netty.http.server.data.sent.time	Timer	Time spent in sending outgoing data. See Http Server Data Sent Time
reactor.netty.http.server.response.time	Timer	Total time for the request/response See Http Server Response Time

These additional metrics are also available:

`ByteBufAllocator` metrics

metric name	type	description
reactor.netty.bytebuf.allocator.used.heap.memory	Gauge	The number of bytes reserved by heap buffer allocator. See Used Heap Memory

metric name	type	description
reactor.netty.bytebuf.allocator.used.direct.memory	Gauge	The number of bytes reserved by direct buffer allocator. See Used Direct Memory
reactor.netty.bytebuf.allocator.heap.arenas	Gauge	The number of heap arenas (when `PooledByteBufAllocator`). See Heap Arenas
reactor.netty.bytebuf.allocator.direct.arenas	Gauge	The number of direct arenas (when `PooledByteBufAllocator`). See Direct Arenas
reactor.netty.bytebuf.allocator.threadlocal.caches	Gauge	The number of thread local caches (when `PooledByteBufAllocator`). See Thread Local Caches
reactor.netty.bytebuf.allocator.small.cache.size	Gauge	The size of the small cache (when `PooledByteBufAllocator`). See Small Cache Size
reactor.netty.bytebuf.allocator.normal.cache.size	Gauge	The size of the normal cache (when `PooledByteBufAllocator`). See Normal Cache Size
reactor.netty.bytebuf.allocator.chunk.size	Gauge	The chunk size for an arena (when `PooledByteBufAllocator`). See Chunk Size
reactor.netty.bytebuf.allocator.active.heap.memory	Gauge	The actual bytes consumed by in-use buffers allocated from heap buffer pools (when `PooledByteBufAllocator`). See Active Heap Memory
reactor.netty.bytebuf.allocator.active.direct.memory	Gauge	The actual bytes consumed by in-use buffers allocated from direct buffer pools (when `PooledByteBufAllocator`). See Active Direct Memory

`EventLoop` metrics

metric name	type	description
reactor.netty.eventloop.pending.tasks	Gauge	The number of tasks that are pending for processing on an event loop. See Pending Tasks

The following example enables that integration:

github.com/reactor/reactor-netty/tree/main/reactor-netty-examples/src/main/java/reactor/netty/

examples/documentation/http/server/metrics/Application.java

```
import io.micrometer.core.instrument.Metrics;
import io.micrometer.core.instrument.config.MeterFilter;
import reactor.core.publisher.Mono;
import reactor.netty.DisposableServer;
import reactor.netty.http.server.HttpServer;

public class Application {

    public static void main(String[] args) {
        Metrics.globalRegistry ①
                .config()
                .meterFilter(MeterFilter.maximumAllowableTags
("reactor.netty.http.server", "URI", 100, MeterFilter.deny()));

        DisposableServer server =
                HttpServer.create()
                          .metrics(true, s -> {
                              if (s.startsWith("/stream/")) { ②
                                  return "/stream/{n}";
                              }
                              else if (s.startsWith("/bytes/")) {
                                  return "/bytes/{n}";
                              }
                              return s;
                          }) ③
                          .route(r ->
                              r.get("/stream/{n}",
                                    (req, res) -> res.sendString(Mono.just(req.param
("n"))))
                               .get("/bytes/{n}",
                                    (req, res) -> res.sendString(Mono.just(req.param
("n")))))
                          .bindNow();

        server.onDispose()
              .block();
    }
}
```

① Applies upper limit for the meters with URI tag

② Templated URIs will be used as an URI tag value when possible

③ Enables the built-in integration with Micrometer

> In order to avoid a memory and CPU overhead of the enabled metrics, it is important to convert the real URIs to templated URIs when possible. Without a conversion to a template-like form, each distinct URI leads to the creation of a distinct tag, which takes a lot of memory for the metrics.

Always apply an upper limit for the meters with URI tags. Configuring an upper limit on the number of meters can help in cases when the real URIs cannot be templated. You can find more information at `maximumAllowableTags`.

When HTTP server metrics are needed for an integration with a system other than `Micrometer` or you want to provide your own integration with `Micrometer`, you can provide your own metrics recorder, as follows:

github.com/reactor/reactor-netty/tree/main/reactor-netty-examples/src/main/java/reactor/netty/examples/documentation/http/server/metrics/custom/Application.java

```
import reactor.core.publisher.Mono;
import reactor.netty.DisposableServer;
import reactor.netty.http.server.HttpServer;
import reactor.netty.http.server.HttpServerMetricsRecorder;

import java.net.SocketAddress;
import java.time.Duration;

public class Application {

    public static void main(String[] args) {
        DisposableServer server =
                HttpServer.create()
                          .metrics(true, CustomHttpServerMetricsRecorder::new) ①
                          .route(r ->
                              r.get("/stream/{n}",
                                   (req, res) -> res.sendString(Mono.just(req.param
("n"))))
                               .get("/bytes/{n}",
                                   (req, res) -> res.sendString(Mono.just(req.param
("n")))))
                          .bindNow();

        server.onDispose()
              .block();
    }
```

① Enables HTTP server metrics and provides `HttpServerMetricsRecorder` implementation.

Tracing

The HTTP server supports built-in integration with `Micrometer Tracing`.

The following table provides information for the HTTP server spans:

contextual name	description
<HTTP METHOD>_<URI>	Information and total time for the request. See Http Server Response Span.

The following example enables that integration. This concrete example uses `Brave` and reports the information to `Zipkin`. See the `Micrometer Tracing` documentation for `OpenTelemetry` setup.

github.com/reactor/reactor-netty/tree/main/reactor-netty-examples/src/main/java/reactor/netty/examples/documentation/http/server/tracing/Application.java

```
import brave.Tracing;
import brave.propagation.StrictCurrentTraceContext;
import brave.sampler.Sampler;
import io.micrometer.tracing.CurrentTraceContext;
import io.micrometer.tracing.Tracer;
import io.micrometer.tracing.brave.bridge.BraveBaggageManager;
import io.micrometer.tracing.brave.bridge.BraveCurrentTraceContext;
import io.micrometer.tracing.brave.bridge.BravePropagator;
import io.micrometer.tracing.brave.bridge.BraveTracer;
import io.micrometer.tracing.propagation.Propagator;
import reactor.core.publisher.Mono;
import reactor.netty.DisposableServer;
import
reactor.netty.http.observability.ReactorNettyPropagatingReceiverTracingObservationHand
ler;
import reactor.netty.http.server.HttpServer;
import zipkin2.reporter.brave.AsyncZipkinSpanHandler;
import zipkin2.reporter.urlconnection.URLConnectionSender;

import static reactor.netty.Metrics.OBSERVATION_REGISTRY;

public class Application {

    public static void main(String[] args) {
        init(); ①

        DisposableServer server =
                HttpServer.create()
                          .metrics(true, s -> {
                              if (s.startsWith("/stream/")) { ②
                                  return "/stream/{n}";
                              }
                              return s;
                          }) ③
                          .route(r -> r.get("/stream/{n}",
                              (req, res) -> res.sendString(Mono.just(req.param(
"n")))))
                          .bindNow();
```

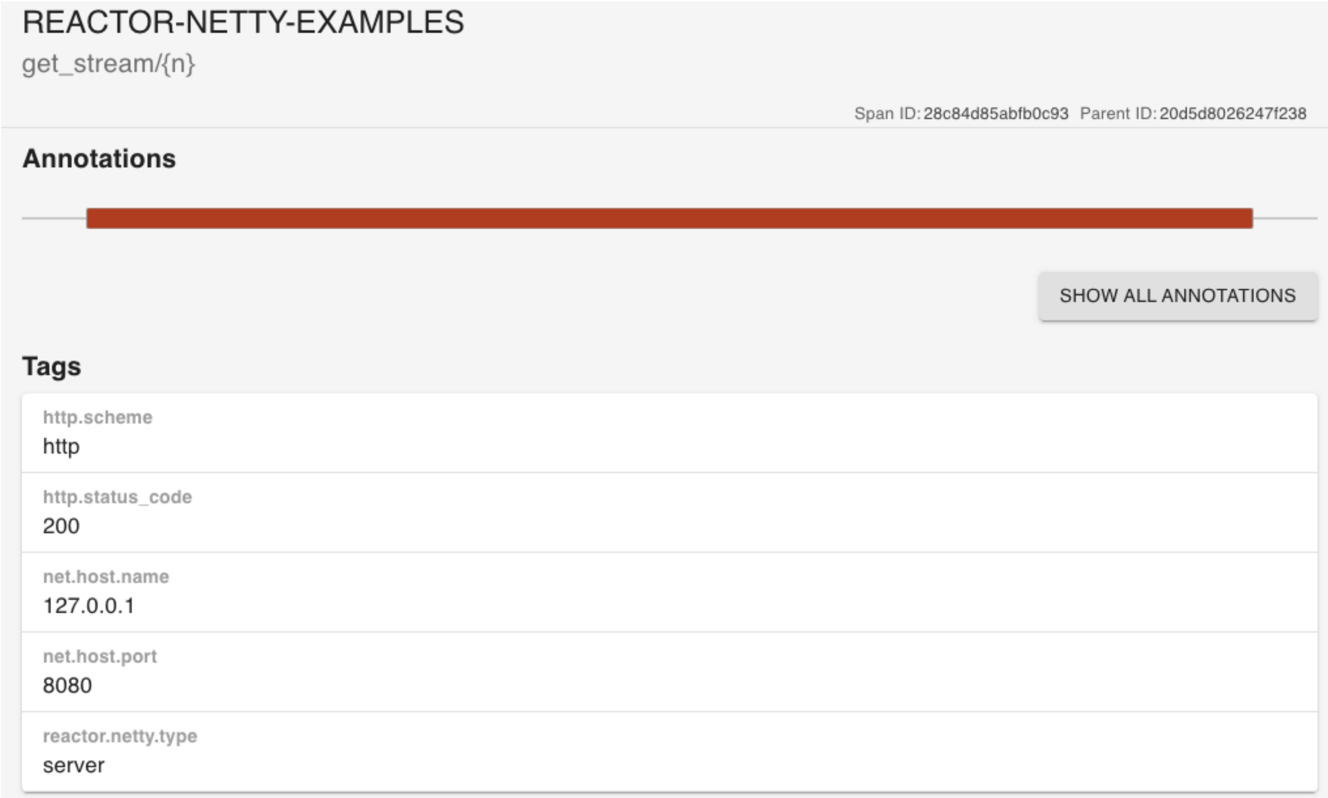
```
        server.onDispose()
                .block();
    }

    /**
     * This setup is based on
     * <a
href="https://micrometer.io/docs/tracing#_micrometer_tracing_brave_setup">Micrometer
Tracing Brave Setup</a>.
     */
    static void init() {
        AsyncZipkinSpanHandler spanHandler = AsyncZipkinSpanHandler
                .create(URLConnectionSender.create
("http://localhost:9411/api/v2/spans"));

        StrictCurrentTraceContext braveCurrentTraceContext =
StrictCurrentTraceContext.create();

        CurrentTraceContext bridgeContext = new BraveCurrentTraceContext
(braveCurrentTraceContext);

        Tracing tracing =
                Tracing.newBuilder()
                       .currentTraceContext(braveCurrentTraceContext)
                       .supportsJoin(false)
                       .traceId128Bit(true)
                       .sampler(Sampler.ALWAYS_SAMPLE)
                       .addSpanHandler(spanHandler)
                       .localServiceName("reactor-netty-examples")
                       .build();

        brave.Tracer braveTracer = tracing.tracer();

        Tracer tracer = new BraveTracer(braveTracer, bridgeContext, new
BraveBaggageManager());

        Propagator propagator = new BravePropagator(tracing);

        OBSERVATION_REGISTRY.observationConfig()
                            .observationHandler(new
ReactorNettyPropagatingReceiverTracingObservationHandler(tracer, propagator));
    }
}
```

① Initializes Brave, Zipkin, and the Observation registry.

② Templated URIs are used as an URI tag value when possible.

③ Enables the built-in integration with Micrometer.

The result in `Zipkin` looks like:



Access Current Observation

Project Micrometer provides `a library` that assists with context propagation across different types of context mechanisms such as `ThreadLocal`, `Reactor Context` and others.

The following example shows how to use this library in a custom `ChannelHandler`:

github.com/reactor/reactor-netty/tree/main/reactor-netty-examples/src/main/java/reactor/netty/examples/documentation/http/server/tracing/custom/Application.java

```
import brave.Tracing;
import brave.propagation.StrictCurrentTraceContext;
import brave.sampler.Sampler;
import io.micrometer.context.ContextSnapshot;
import io.micrometer.context.ContextSnapshotFactory;
import io.micrometer.tracing.CurrentTraceContext;
import io.micrometer.tracing.Tracer;
import io.micrometer.tracing.brave.bridge.BraveBaggageManager;
import io.micrometer.tracing.brave.bridge.BraveCurrentTraceContext;
import io.micrometer.tracing.brave.bridge.BravePropagator;
import io.micrometer.tracing.brave.bridge.BraveTracer;
import io.micrometer.tracing.propagation.Propagator;
import io.netty.channel.ChannelHandler;
import io.netty.channel.ChannelHandlerContext;
import io.netty.channel.ChannelOutboundHandlerAdapter;
import io.netty.channel.ChannelPromise;
import reactor.core.publisher.Mono;
import reactor.netty.DisposableServer;
import
```

```
reactor.netty.http.observability.ReactorNettyPropagatingReceiverTracingObservationHand
ler;
import reactor.netty.http.server.HttpServer;
import zipkin2.reporter.brave.AsyncZipkinSpanHandler;
import zipkin2.reporter.urlconnection.URLConnectionSender;

import static reactor.netty.Metrics.OBSERVATION_REGISTRY;

public class Application {

    public static void main(String[] args) {
        init(); ①

        DisposableServer server =
                HttpServer.create()
                          .metrics(true, s -> {
                              if (s.startsWith("/stream/")) { ②
                                  return "/stream/{n}";
                              }
                              return s;
                          }) ③
                          .doOnConnection(conn -> conn.addHandlerLast
(CustomChannelOutboundHandler.INSTANCE)) ④
                          .route(r -> r.get("/stream/{n}",
                              (req, res) -> res.sendString(Mono.just(req.param(
"n")))))
                          .bindNow();

        server.onDispose()
              .block();
    }

    static final class CustomChannelOutboundHandler extends
ChannelOutboundHandlerAdapter {

        static final ChannelHandler INSTANCE = new CustomChannelOutboundHandler();

        @Override
        public boolean isSharable() {
            return true;
        }

        @Override
        @SuppressWarnings({"FutureReturnValueIgnored", "try"})
        public void write(ChannelHandlerContext ctx, Object msg, ChannelPromise
promise) {
            try (ContextSnapshot.Scope scope = ContextSnapshotFactory.builder().
build().setThreadLocalsFrom(ctx.channel())) {
                System.out.println("Current Observation in Scope: " +
OBSERVATION_REGISTRY.getCurrentObservation());
                //"FutureReturnValueIgnored" this is deliberate
```

```
                ctx.write(msg, promise);
            }
            System.out.println("Current Observation: " + OBSERVATION_REGISTRY
.getCurrentObservation());
        }
    }
```

① Initializes Brave, Zipkin, and the Observation registry.

② Templated URIs are used as an URI tag value when possible.

③ Enables the built-in integration with Micrometer.

④ Custom `ChannelHandler` that uses context propagation library. This concrete example overrides only `ChannelOutboundHandlerAdapter#write`, if it is needed, the same logic can be used for the rest of the methods. Also, this concrete example sets all `ThreadLocal` values for which there is a value in the given `Channel`, if another behaviour is needed please check `context propagation library API`. For example, you may want to set only some of the `ThreadLocal` values.

> When you enable Reactor Netty tracing within a framework, you may need to let Reactor Netty use the `ObservationRegistry` created by this framework. For this purpose you need to invoke `reactor.netty.Metrics#observationRegistry`. You may also need to configure the Reactor Netty `ObservationHandlers` using the API provided by the framework.

Unix Domain Sockets

The `HTTP` server supports Unix Domain Sockets (UDS) when native transport is in use.

The following example shows how to use UDS support:

github.com/reactor/reactor-netty/tree/main/reactor-netty-examples/src/main/java/reactor/netty/examples/documentation/http/server/uds/Application.java

```
import io.netty.channel.unix.DomainSocketAddress;
import reactor.netty.DisposableServer;
import reactor.netty.http.server.HttpServer;

public class Application {

    public static void main(String[] args) {
        DisposableServer server =
                HttpServer.create()
                          .bindAddress(() -> new DomainSocketAddress(
"/tmp/test.sock")) ①
                          .bindNow();

        server.onDispose()
              .block();
    }
```

```
}
```

① Specifies `DomainSocketAddress` that will be used

Timeout Configuration

This section describes various timeout configuration options that can be used in `HttpServer`. Configuring a proper timeout may improve or solve issues in the communication process. The configuration options can be grouped as follows:

- Request Timeout
- Connection Timeout
- SSL/TLS Timeout

Request Timeout

The following listing shows all available request timeout configuration options.

- `readTimeout` - the maximum time between each network-level read operation while reading a given request content (resolution: ms)
- `requestTimeout` - the maximum time for reading a given request content (resolution: ms).

> It is always a good practice to configure a read/request timeout.

To customize the default settings, you can configure `HttpServer` as follows:

github.com/reactor/reactor-netty/tree/main/reactor-netty-examples/src/main/java/reactor/netty/examples/documentation/http/server/read/timeout/Application.java

```
import reactor.netty.DisposableServer;
import reactor.netty.http.server.HttpServer;

import java.time.Duration;

public class Application {

    public static void main(String[] args) {
        DisposableServer server =
                HttpServer.create()
                          .readTimeout(Duration.ofSeconds(5))     ①
                          .requestTimeout(Duration.ofSeconds(30)) ②
                          .handle((request, response) -> request.receive().then())
                          .bindNow();

        server.onDispose()
              .block();
    }
```

```
}
```

① Configures the read timeout to 5 second.

② Configures the request timeout to 30 second.

Connection Timeout

The following listing shows all available connection timeout configuration options.

- `idleTimeout` - The maximum time (resolution: ms) that this connection stays opened and waits for HTTP request. Once the timeout is reached, the connection is closed. By default, `idleTimeout` is not specified, this indicates no timeout (i.e. infinite), which means the connection is closed only if one of the peers decides to close it explicitly.

> It is always a good practice to configure an idle timeout.

To customize the default settings, you can configure `HttpServer` as follows:

github.com/reactor/reactor-netty/tree/main/reactor-netty-examples/src/main/java/reactor/netty/examples/documentation/http/server/idle/timeout/Application.java

```java
import reactor.netty.DisposableServer;
import reactor.netty.http.server.HttpServer;

import java.time.Duration;

public class Application {

    public static void main(String[] args) {
        DisposableServer server =
                HttpServer.create()
                          .idleTimeout(Duration.ofSeconds(1)) ①
                          .bindNow();

        server.onDispose()
              .block();
    }
}
```

① Configures the default idle timeout to 1 second.

SSL/TLS Timeout

`HttpServer` supports the SSL/TLS functionality provided by Netty.

The following list describes the available timeout configuration options:

- `handshakeTimeout` - Use this option to configure the SSL handshake timeout (resolution: ms). Default: 10s.

You should consider increasing the SSL handshake timeout when expecting slow network connections.

- `closeNotifyFlushTimeout` - Use this option to configure the SSL `close_notify` flush timeout (resolution: ms). Default: 3s.
- `closeNotifyReadTimeout` - Use this option to configure the SSL `close_notify` read timeout (resolution: ms). Default: 0s.

To customize the default settings, you can configure `HttpServer` as follows:

github.com/reactor/reactor-netty/tree/main/reactor-netty-examples/src/main/java/reactor/netty/examples/documentation/http/server/security/custom/Application.java

```
import reactor.netty.DisposableServer;
import reactor.netty.http.Http11SslContextSpec;
import reactor.netty.http.server.HttpServer;

import java.io.File;
import java.time.Duration;

public class Application {

    public static void main(String[] args) {
        File cert = new File("certificate.crt");
        File key = new File("private.key");

        Http11SslContextSpec http11SslContextSpec = Http11SslContextSpec.forServer
(cert, key);

        DisposableServer server =
                HttpServer.create()
                          .secure(spec -> spec.sslContext(http11SslContextSpec)
                                              .handshakeTimeout(Duration.ofSeconds(
30))         ①
                                              .closeNotifyFlushTimeout(Duration
.ofSeconds(10))  ②
                                              .closeNotifyReadTimeout(Duration
.ofSeconds(10)))  ③
                          .bindNow();

        server.onDispose()
              .block();
    }
}
```

① Configures the SSL handshake timeout to 30 seconds.

② Configures the SSL `close_notify` flush timeout to 10 seconds.

③ Configures the SSL `close_notify` read timeout to 10 seconds.

Chapter 6. HTTP Client

Reactor Netty provides the easy-to-use and easy-to-configure `HttpClient`. It hides most of the Netty functionality that is required to create an `HTTP` client and adds Reactive Streams backpressure.

Connect

To connect the `HTTP` client to a given `HTTP` endpoint, you must create and configure a `HttpClient` instance. By default, the host is configured for `localhost` and the port is `80`. The following example shows how to do so:

github.com/reactor/reactor-netty/tree/main/reactor-netty-examples/src/main/java/reactor/netty/examples/documentation/http/client/connect/Application.java

```
import reactor.netty.http.client.HttpClient;

public class Application {

    public static void main(String[] args) {
        HttpClient client = HttpClient.create();  ①

        client.get()                       ②
              .uri("https://example.com/") ③
              .response()                  ④
              .block();
    }
}
```

① Creates a HttpClient instance ready for configuring.

② Specifies that `GET` method will be used.

③ Specifies the path.

④ Obtains the response HttpClientResponse

The following example uses `WebSocket`:

github.com/reactor/reactor-netty/tree/main/reactor-netty-examples/src/main/java/reactor/netty/examples/documentation/http/client/websocket/Application.java

```java
import io.netty.buffer.Unpooled;
import io.netty.util.CharsetUtil;
import reactor.core.publisher.Flux;
import reactor.netty.http.client.HttpClient;

public class Application {

    public static void main(String[] args) {
        HttpClient client = HttpClient.create();

        client.websocket()
              .uri("wss://echo.websocket.org")
              .handle((inbound, outbound) -> {
                  inbound.receive()
                         .asString()
                         .take(1)
                         .subscribe(System.out::println);

                  final byte[] msgBytes = "hello".getBytes(CharsetUtil.ISO_8859_1);
                  return outbound.send(Flux.just(Unpooled.wrappedBuffer(msgBytes),
Unpooled.wrappedBuffer(msgBytes)))
                                 .neverComplete();
              })
              .blockLast();
    }
}
```

Host and Port

In order to connect to a specific host and port, you can apply the following configuration to the `HTTP` client:

github.com/reactor/reactor-netty/tree/main/reactor-netty-examples/src/main/java/reactor/netty/examples/documentation/http/client/address/Application.java

```java
import reactor.netty.http.client.HttpClient;

public class Application {

    public static void main(String[] args) {
        HttpClient client =
                HttpClient.create()
                          .host("example.com") ①
                          .port(80);          ②

        client.get()
              .uri("/")
              .response()
              .block();
    }
}
```

① Configures the HTTP host

② Configures the HTTP port

> The port can be specified also with **PORT** environment variable.

Eager Initialization

By default, the initialization of the HttpClient resources happens on demand. This means that the first request absorbs the extra time needed to initialize and load:

- the event loop group
- the host name resolver
- the native transport libraries (when native transport is used)
- the native libraries for the security (in case of OpenSsl)

When you need to preload these resources, you can configure the HttpClient as follows:

github.com/reactor/reactor-netty/tree/main/reactor-netty-examples/src/main/java/reactor/netty/examples/documentation/http/client/warmup/Application.java

```java
import reactor.core.publisher.Mono;
import reactor.netty.ByteBufFlux;
import reactor.netty.http.client.HttpClient;

public class Application {

    public static void main(String[] args) {
        HttpClient client = HttpClient.create();

        client.warmup() ①
              .block();

        client.post()
              .uri("https://example.com/")
              .send(ByteBufFlux.fromString(Mono.just("hello")))
              .response()
              .block(); ②
    }
}
```

① Initialize and load the event loop group, the host name resolver, the native transport libraries and the native libraries for the security

② Host name resolution happens with the first request. In this example, a connection pool is used, so with the first request the connection to the URL is established, the subsequent requests to the same URL reuse the connections from the pool.

Writing Data

To send data to a given `HTTP` endpoint, you can provide a `Publisher` by using the `send(Publisher)` method. By default, `Transfer-Encoding: chunked` is applied for those `HTTP` methods for which a request body is expected. `Content-Length` provided through request headers disables `Transfer-Encoding: chunked`, if necessary. The following example sends `hello`:

github.com/reactor/reactor-netty/tree/main/reactor-netty-examples/src/main/java/reactor/netty/examples/documentation/http/client/send/Application.java

```java
import reactor.core.publisher.Mono;
import reactor.netty.ByteBufFlux;
import reactor.netty.http.client.HttpClient;

public class Application {

    public static void main(String[] args) {
        HttpClient client = HttpClient.create();

        client.post()
              .uri("https://example.com/")
              .send(ByteBufFlux.fromString(Mono.just("hello"))) ①
              .response()
              .block();
    }
}
```

① Sends a `hello` string to the given `HTTP` endpoint

Adding Headers and Other Metadata

When sending data to a given `HTTP` endpoint, you may need to send additional headers, cookies and other metadata. You can use the following configuration to do so:

github.com/reactor/reactor-netty/tree/main/reactor-netty-examples/src/main/java/reactor/netty/examples/documentation/http/client/send/headers/Application.java

```java
import io.netty.handler.codec.http.HttpHeaderNames;
import reactor.core.publisher.Mono;
import reactor.netty.ByteBufFlux;
import reactor.netty.http.client.HttpClient;

public class Application {

    public static void main(String[] args) {
        HttpClient client =
                HttpClient.create()
                          .headers(h -> h.set(HttpHeaderNames.CONTENT_LENGTH, 5)); ①

        client.post()
              .uri("https://example.com/")
              .send(ByteBufFlux.fromString(Mono.just("hello")))
              .response()
              .block();
    }
}
```

① Disables `Transfer-Encoding: chunked` and provides `Content-Length` header.

Compression

You can enable compression on the `HTTP` client, which means the request header `Accept-Encoding` is added to the request headers. The following example shows how to do so:

github.com/reactor/reactor-netty/tree/main/reactor-netty-examples/src/main/java/reactor/netty/examples/documentation/http/client/compression/Application.java

```java
import reactor.netty.http.client.HttpClient;

public class Application {

    public static void main(String[] args) {
        HttpClient client =
                HttpClient.create()
                          .compress(true);

        client.get()
              .uri("https://example.com/")
              .response()
              .block();
    }
}
```

Auto-Redirect Support

You can configure the `HTTP` client to enable auto-redirect support.

Reactor Netty provides two different strategies for auto-redirect support:

- `followRedirect(boolean)`: Specifies whether HTTP auto-redirect support is enabled for statuses `301|302|303|307|308`. When it is `303` status code, `GET` method is used for the redirect.
- `followRedirect(BiPredicate<HttpClientRequest, HttpClientResponse>)`: Enables auto-redirect support if the supplied predicate matches.

The following example uses `followRedirect(true)`:

github.com/reactor/reactor-netty/tree/main/reactor-netty-examples/src/main/java/reactor/netty/examples/documentation/http/client/redirect/Application.java

```java
import reactor.netty.http.client.HttpClient;

public class Application {

    public static void main(String[] args) {
        HttpClient client =
                HttpClient.create()
                          .followRedirect(true);

        client.get()
              .uri("https://example.com/")
              .response()
              .block();
    }
}
```

Consuming Data

To receive data from a given HTTP endpoint, you can use one of the methods from HttpClient.ResponseReceiver. The following example uses the responseContent method:

github.com/reactor/reactor-netty/tree/main/reactor-netty-examples/src/main/java/reactor/netty/examples/documentation/http/client/read/Application.java

```java
import reactor.netty.http.client.HttpClient;

public class Application {

    public static void main(String[] args) {
        HttpClient client = HttpClient.create();

        client.get()
              .uri("https://example.com/")
              .responseContent() ①
              .aggregate()       ②
              .asString()        ③
              .block();
    }
}
```

① Receives data from a given HTTP endpoint

② Aggregates the data

③ Transforms the data as string

Reading Headers and Other Metadata

When receiving data from a given HTTP endpoint, you can check response headers, status code, and other metadata. You can obtain this additional metadata by using `HttpClientResponse`. The following example shows how to do so.

github.com/reactor/reactor-netty/tree/main/reactor-netty-examples/src/main/java/reactor/netty/examples/documentation/http/client/read/status/Application.java

```
import reactor.netty.http.client.HttpClient;

public class Application {

    public static void main(String[] args) {
        HttpClient client = HttpClient.create();

        client.get()
              .uri("https://example.com/")
              .responseSingle((resp, bytes) -> {
                  System.out.println(resp.status()); ①
                  return bytes.asString();
              })
              .block();
    }
}
```

① Obtains the status code.

HTTP Response Decoder

By default, `Netty` configures some restrictions for the incoming responses, such as:

- The maximum length of the initial line.
- The maximum length of all headers.
- The maximum length of the content or each chunk.

For more information, see HttpResponseDecoder

By default, the `HTTP` client is configured with the following settings:

github.com/reactor/reactor-netty/tree/main/reactor-netty-http/src/main/java/reactor/netty/http/HttpDecoderSpec.java

```java
    public static final int DEFAULT_MAX_INITIAL_LINE_LENGTH             = 4096;
    public static final int DEFAULT_MAX_HEADER_SIZE                     = 8192;
    /**
     * Default max chunk size.
     *
     * @deprecated as of 1.1.0. This will be removed in 2.0.0 as Netty 5 does not
support this configuration.
     */
    @Deprecated
    public static final int DEFAULT_MAX_CHUNK_SIZE                      = 8192;
    public static final boolean DEFAULT_VALIDATE_HEADERS                = true;
    public static final int DEFAULT_INITIAL_BUFFER_SIZE                 = 128;
    public static final boolean DEFAULT_ALLOW_DUPLICATE_CONTENT_LENGTHS = false;
```

github.com/reactor/reactor-netty/tree/main/reactor-netty-http/src/main/java/reactor/netty/http/client/HttpResponseDecoderSpec.java

```java
    /**
     * The maximum length of the content of the HTTP/2.0 clear-text upgrade request.
     * By default, the client will allow an upgrade request with up to 65536 as
     * the maximum length of the aggregated content.
     */
    public static final int DEFAULT_H2C_MAX_CONTENT_LENGTH = 65536;

    boolean failOnMissingResponse        = DEFAULT_FAIL_ON_MISSING_RESPONSE;
```

When you need to change these default settings, you can configure the `HTTP` client as follows:

github.com/reactor/reactor-netty/tree/main/reactor-netty-examples/src/main/java/reactor/netty/examples/documentation/http/client/responsedecoder/Application.java

```java
import reactor.netty.http.client.HttpClient;

public class Application {

    public static void main(String[] args) {
        HttpClient client =
                HttpClient.create()
                          .httpResponseDecoder(spec -> spec.maxHeaderSize(16384)); ①

        client.get()
              .uri("https://example.com/")
              .responseContent()
              .aggregate()
              .asString()
              .block();
    }
}
```

① The maximum length of all headers will be 16384. When this value is exceeded, a TooLongFrameException is raised.

Lifecycle Callbacks

The following lifecycle callbacks are provided to let you extend the HttpClient.

Callback	Description
doAfterRequest	Invoked when the request has been sent.
doAfterResolve	Invoked after the remote address has been resolved successfully.
doAfterResponseSuccess	Invoked after the response has been fully received.
doOnChannelInit	Invoked when initializing the channel.
doOnConnect	Invoked when the channel is about to connect.
doOnConnected	Invoked after the channel has been connected.
doOnDisconnected	Invoked after the channel has been disconnected.
doOnError	Invoked when the request has not been sent and when the response has not been fully received.
doOnRedirect	Invoked when the response headers have been received, and the request is about to be redirected.
doOnRequest	Invoked when the request is about to be sent.
doOnRequestError	Invoked when the request has not been sent.

Callback	Description
doOnResolve	Invoked when the remote address is about to be resolved.
doOnResolveError	Invoked in case the remote address hasn't been resolved successfully.
doOnResponse	Invoked after the response headers have been received.
doOnResponseError	Invoked when the response has not been fully received.

The following example uses the doOnConnected and doOnChannelInit callbacks:

github.com/reactor/reactor-netty/tree/main/reactor-netty-examples/src/main/java/reactor/netty/examples/documentation/http/client/lifecycle/Application.java

```
import io.netty.handler.logging.LoggingHandler;
import io.netty.handler.timeout.ReadTimeoutHandler;
import reactor.netty.http.client.HttpClient;
import java.util.concurrent.TimeUnit;

public class Application {

    public static void main(String[] args) {
        HttpClient client =
                HttpClient.create()
                          .doOnConnected(conn ->
                              conn.addHandlerFirst(new ReadTimeoutHandler(10,
TimeUnit.SECONDS)))   ①
                          .doOnChannelInit((observer, channel, remoteAddress) ->
                              channel.pipeline()
                                     .addFirst(new LoggingHandler
("reactor.netty.examples")));      ②

        client.get()
              .uri("https://example.com/")
              .response()
              .block();
    }
}
```

① Netty pipeline is extended with ReadTimeoutHandler when the channel has been connected.

② Netty pipeline is extended with LoggingHandler when initializing the channel.

TCP-level Configuration

When you need configurations on a TCP level, you can use the following snippet to extend the default TCP client configuration (add an option, bind address etc.):

github.com/reactor/reactor-netty/tree/main/reactor-netty-examples/src/main/java/reactor/netty/examples/documentation/http/client/channeloptions/Application.java

```
import io.netty.channel.ChannelOption;
import io.netty.channel.epoll.EpollChannelOption;
//import io.netty.channel.socket.nio.NioChannelOption;
//import jdk.net.ExtendedSocketOptions;
import reactor.netty.http.client.HttpClient;
import java.net.InetSocketAddress;

public class Application {

    public static void main(String[] args) {
        HttpClient client =
                HttpClient.create()
                          .bindAddress(() -> new InetSocketAddress("host", 1234))
                          .option(ChannelOption.CONNECT_TIMEOUT_MILLIS, 10000) ①
                          .option(ChannelOption.SO_KEEPALIVE, true)            ②
                          // The options below are available only when NIO transport
(Java 11) is used
                          // on Mac or Linux (Java does not currently support these
extended options on Windows)
                          // https://bugs.openjdk.java.net/browse/JDK-8194298

//.option(NioChannelOption.of(ExtendedSocketOptions.TCP_KEEPIDLE), 300)

//.option(NioChannelOption.of(ExtendedSocketOptions.TCP_KEEPINTERVAL), 60)

//.option(NioChannelOption.of(ExtendedSocketOptions.TCP_KEEPCOUNT), 8);
                          // The options below are available only when Epoll transport
is used
                          .option(EpollChannelOption.TCP_KEEPIDLE, 300)        ③
                          .option(EpollChannelOption.TCP_KEEPINTVL, 60)        ④
                          .option(EpollChannelOption.TCP_KEEPCNT, 8);          ⑤

        String response =
                client.get()
                      .uri("https://example.com/")
                      .responseContent()
                      .aggregate()
                      .asString()
                      .block();

        System.out.println("Response " + response);
    }
}
```

① Configures the connection establishment timeout to 10 seconds.

② Enables TCP `keepalive`. This means that TCP starts sending `keepalive` probes when a connection is idle for some time.

③ The connection needs to remain idle for 5 minutes before TCP starts sending `keepalive` probes.

④ Configures the time between individual `keepalive` probes to 1 minute.

⑤ Configures the maximum number of TCP `keepalive` probes to 8.

See TCP Client for more about `TCP` level configurations.

Wire Logger

Reactor Netty provides wire logging for when the traffic between the peers needs to be inspected. By default, wire logging is disabled. To enable it, you must set the logger `reactor.netty.http.client.HttpClient` level to `DEBUG` and apply the following configuration:

github.com/reactor/reactor-netty/tree/main/reactor-netty-examples/src/main/java/reactor/netty/examples/documentation/http/client/wiretap/Application.java

```
import reactor.netty.http.client.HttpClient;

public class Application {

    public static void main(String[] args) {
        HttpClient client =
                HttpClient.create()
                          .wiretap(true); ①

        client.get()
              .uri("https://example.com/")
              .response()
              .block();
    }
}
```

① Enables the wire logging

Wire Logger formatters

Reactor Netty supports 3 different formatters:

- AdvancedByteBufFormat#HEX_DUMP - the default

github.com/reactor/reactor-netty/tree/main/reactor-netty-core/src/main/java/reactor/netty/transport/logging/AdvancedByteBufFormat.java

```
    /**
     * When wire logging is enabled with this format, both events and content will be
logged.
     * The content will be in hex format.
     * <p>Examples:</p>
     * <pre>
     * {@code
     * reactor.netty.http.HttpTests - [d5230a14, L:/0:0:0:0:0:0:0:1:60267 -
R:/0:0:0:0:0:0:0:1:60269] REGISTERED
     * reactor.netty.http.HttpTests - [d5230a14, L:/0:0:0:0:0:0:0:1:60267 -
R:/0:0:0:0:0:0:0:1:60269] ACTIVE
     * reactor.netty.http.HttpTests - [d5230a14, L:/0:0:0:0:0:0:0:1:60267 -
R:/0:0:0:0:0:0:0:1:60269] READ: 145B
     *          +-------------------------------------------------+
     *          |  0  1  2  3  4  5  6  7  8  9  a  b  c  d  e  f |
     * +--------+-------------------------------------------------+----------------+
     * |00000000| 50 4f 53 54 20 2f 74 65 73 74 2f 57 6f 72 6c 64 |POST /test/World|
     * |00000010| 20 48 54 54 50 2f 31 2e 31 0d 0a 43 6f 6e 74 65 | HTTP/1.1..Conte|
     * |00000020| 6e 74 2d 54 79 70 65 3a 20 74 65 78 74 2f 70 6c |nt-Type: text/pl|
     * |00000030| 61 69 6e 0d 0a 75 73 65 72 2d 61 67 65 6e 74 3a |ain..user-agent:|
     * |00000040| 20 52 65 61 63 74 6f 72 4e 65 74 74 79 2f 64 65 | ReactorNetty/de|
     * ...
     * reactor.netty.http.HttpTests - [d5230a14, L:/0:0:0:0:0:0:0:1:60267 -
R:/0:0:0:0:0:0:0:1:60269] WRITE: 38B
     *          +-------------------------------------------------+
     *          |  0  1  2  3  4  5  6  7  8  9  a  b  c  d  e  f |
     * +--------+-------------------------------------------------+----------------+
     * |00000000| 48 54 54 50 2f 31 2e 31 20 32 30 30 20 4f 4b 0d |HTTP/1.1 200 OK.|
     * |00000010| 0a 63 6f 6e 74 65 6e 74 2d 6c 65 6e 67 74 68 3a |.content-length:|
     * |00000020| 20 30 0d 0a 0d 0a                               | 0....          |
     * +--------+-------------------------------------------------+----------------+
     * }
     * </pre>
     */
```

- AdvancedByteBufFormat#SIMPLE

github.com/reactor/reactor-netty/tree/main/reactor-netty-core/src/main/java/reactor/netty/transport/logging/AdvancedByteBufFormat.java

```
    /**
     * When wire logging is enabled with this format, only the events will be logged.
     * <p>Examples:</p>
     * <pre>
     * {@code
     * reactor.netty.http.HttpTests - [230d3686, L:/0:0:0:0:0:0:0:1:60241 -
R:/0:0:0:0:0:0:0:1:60245] REGISTERED
     * reactor.netty.http.HttpTests - [230d3686, L:/0:0:0:0:0:0:0:1:60241 -
R:/0:0:0:0:0:0:0:1:60245] ACTIVE
     * reactor.netty.http.HttpTests - [230d3686, L:/0:0:0:0:0:0:0:1:60241 -
R:/0:0:0:0:0:0:0:1:60245] READ: 145B
     * reactor.netty.http.HttpTests - [230d3686, L:/0:0:0:0:0:0:0:1:60241 -
R:/0:0:0:0:0:0:0:1:60245] WRITE: 38B
     * }
     * </pre>
     */
```

- AdvancedByteBufFormat#TEXTUAL

github.com/reactor/reactor-netty/tree/main/reactor-netty-core/src/main/java/reactor/netty/transport/logging/AdvancedByteBufFormat.java

```
    /**
     * When wire logging is enabled with this format, both events and content will be
logged.
     * The content will be in plain text format.
     * <p>Examples:</p>
     * <pre>
     * {@code
     * reactor.netty.http.HttpTests - [02c3db6c, L:/0:0:0:0:0:0:0:1:60317 -
R:/0:0:0:0:0:0:0:1:60319] REGISTERED
     * reactor.netty.http.HttpTests - [02c3db6c, L:/0:0:0:0:0:0:0:1:60317 -
R:/0:0:0:0:0:0:0:1:60319] ACTIVE
     * reactor.netty.http.HttpTests - [02c3db6c, L:/0:0:0:0:0:0:0:1:60317 -
R:/0:0:0:0:0:0:0:1:60319] READ: 145B POST /test/World HTTP/1.1
     * Content-Type: text/plain
     * user-agent: ReactorNetty/dev
     * ...
     * reactor.netty.http.HttpTests - [02c3db6c, L:/0:0:0:0:0:0:0:1:60317 -
R:/0:0:0:0:0:0:0:1:60319] WRITE: 38B HTTP/1.1 200 OK
     * content-length: 0
     * }
     * </pre>
     */
```

When you need to change the default formatter you can configure it as follows:

github.com/reactor/reactor-netty/tree/main/reactor-netty-examples/src/main/java/reactor/netty/

examples/documentation/http/client/wiretap/custom/Application.java

```java
import io.netty.handler.logging.LogLevel;
import reactor.netty.http.client.HttpClient;
import reactor.netty.transport.logging.AdvancedByteBufFormat;

public class Application {

    public static void main(String[] args) {
        HttpClient client =
                HttpClient.create()
                          .wiretap("logger-name", LogLevel.DEBUG,
AdvancedByteBufFormat.TEXTUAL); ①

        client.get()
              .uri("https://example.com/")
              .response()
              .block();
    }
}
```

① Enables the wire logging, AdvancedByteBufFormat#TEXTUAL is used for printing the content.

Event Loop Group

By default `Reactor Netty` uses an “Event Loop Group”, where the number of the worker threads equals the number of processors available to the runtime on initialization (but with a minimum value of 4). When you need a different configuration, you can use one of the `LoopResources#create` methods.

The following listing shows the default configuration for the Event Loop Group:

github.com/reactor/reactor-netty/tree/main/reactor-netty-core/src/main/java/reactor/netty/ReactorNetty.java

```
    /**
     * Default worker thread count, fallback to available processor
     * (but with a minimum value of 4).
     */
    public static final String IO_WORKER_COUNT = "reactor.netty.ioWorkerCount";
    /**
     * Default selector thread count, fallback to -1 (no selector thread)
     * <p><strong>Note:</strong> In most use cases using a worker thread also as a
selector thread works well.
     * A possible use case for specifying a separate selector thread might be when the
worker threads are too busy
     * and connections cannot be accepted fast enough.
     * <p><strong>Note:</strong> Although more than 1 can be configured as a selector
thread count, in reality
     * only 1 thread will be used as a selector thread.
     */
    public static final String IO_SELECT_COUNT = "reactor.netty.ioSelectCount";
    /**
     * Default worker thread count for UDP, fallback to available processor
     * (but with a minimum value of 4).
     */
    public static final String UDP_IO_THREAD_COUNT =
"reactor.netty.udp.ioThreadCount";
    /**
     * Default quiet period that guarantees that the disposal of the underlying
LoopResources
     * will not happen, fallback to 2 seconds.
     */
    public static final String SHUTDOWN_QUIET_PERIOD =
"reactor.netty.ioShutdownQuietPeriod";
    /**
     * Default maximum amount of time to wait until the disposal of the underlying
LoopResources
     * regardless if a task was submitted during the quiet period, fallback to 15
seconds.
     */
    public static final String SHUTDOWN_TIMEOUT = "reactor.netty.ioShutdownTimeout";

    /**
     * Default value whether the native transport (epoll, kqueue) will be preferred,
     * fallback it will be preferred when available.
     */
    public static final String NATIVE = "reactor.netty.native";
```

If you need changes to these settings, you can apply the following configuration:

github.com/reactor/reactor-netty/tree/main/reactor-netty-examples/src/main/java/reactor/netty/examples/documentation/http/client/eventloop/Application.java

```java
import reactor.netty.http.client.HttpClient;
import reactor.netty.resources.LoopResources;

public class Application {

    public static void main(String[] args) {
        LoopResources loop = LoopResources.create("event-loop", 1, 4, true);
        HttpClient client =
                HttpClient.create()
                          .runOn(loop);

        client.get()
              .uri("https://example.com/")
              .responseContent()
              .aggregate()
              .asString()
              .block();
    }
}
```

Disposing Event Loop Group

- If you use the default `Event Loop Group` provided by Reactor Netty, invoke `HttpResources#disposeLoopsAndConnections/#disposeLoopsAndConnectionsLater` method.

> Disposing `HttpResources` means that every server/client that is using it, will not be able to use it anymore!

- If you use custom `LoopResources`, invoke `LoopResources#dispose/#disposeLater` method.

> Disposing the custom `LoopResources` means that every server/client that is configured to use it, will not be able to use it anymore!

Connection Pool

By default, `HttpClient` uses a "fixed" connection pool with `500` as the maximum number of active channels and `1000` as the maximum number of further channel acquisition attempts allowed to be kept in a pending state (for the rest of the configurations check the system properties or the builder configurations below). This means that the implementation creates a new channel if someone tries to acquire a channel as long as less than `500` have been created and are managed by the pool. When the maximum number of channels in the pool is reached, up to `1000` new attempts to acquire a channel are delayed (pending) until a channel is returned to the pool again, and further attempts are declined with an error.

github.com/reactor/reactor-netty/tree/main/reactor-netty-core/src/main/java/reactor/netty/ReactorNetty.java

```
    /**
     * Default max connections. Fallback to
     * 2 * available number of processors (but with a minimum value of 16)
     */
    public static final String POOL_MAX_CONNECTIONS =
"reactor.netty.pool.maxConnections";
    /**
     * Default acquisition timeout (milliseconds) before error. If -1 will never wait
to
     * acquire before opening a new
     * connection in an unbounded fashion. Fallback 45 seconds
     */
    public static final String POOL_ACQUIRE_TIMEOUT =
"reactor.netty.pool.acquireTimeout";
    /**
     * Default max idle time, fallback - max idle time is not specified.
     * <p><strong>Note:</strong> This configuration is not applicable for {@link
reactor.netty.tcp.TcpClient}.
     * A TCP connection is always closed and never returned to the pool.
     */
    public static final String POOL_MAX_IDLE_TIME = "reactor.netty.pool.maxIdleTime";
    /**
     * Default max life time, fallback - max life time is not specified.
     * <p><strong>Note:</strong> This configuration is not applicable for {@link
reactor.netty.tcp.TcpClient}.
     * A TCP connection is always closed and never returned to the pool.
     */
    public static final String POOL_MAX_LIFE_TIME = "reactor.netty.pool.maxLifeTime";
    /**
     * Default leasing strategy (fifo, lifo), fallback to fifo.
     * <ul>
     *     <li>fifo - The connection selection is first in, first out</li>
     *     <li>lifo - The connection selection is last in, first out</li>
     * </ul>
     * <p><strong>Note:</strong> This configuration is not applicable for {@link
reactor.netty.tcp.TcpClient}.
     * A TCP connection is always closed and never returned to the pool.
     */
    public static final String POOL_LEASING_STRATEGY =
"reactor.netty.pool.leasingStrategy";
    /**
     * Default {@code getPermitsSamplingRate} (between 0d and 1d (percentage))
     * to be used with a {@link SamplingAllocationStrategy}.
     * This strategy wraps a {@link PoolBuilder#sizeBetween(int, int) sizeBetween}
{@link AllocationStrategy}
     * and samples calls to {@link AllocationStrategy#getPermits(int)}.
     * Fallback - sampling is not enabled.
     */
    public static final String POOL_GET_PERMITS_SAMPLING_RATE =
"reactor.netty.pool.getPermitsSamplingRate";
```

```
    /**
     * Default {@code returnPermitsSamplingRate} (between 0d and 1d (percentage))
     * to be used with a {@link SamplingAllocationStrategy}.
     * This strategy wraps a {@link PoolBuilder#sizeBetween(int, int) sizeBetween}
{@link AllocationStrategy}
     * and samples calls to {@link AllocationStrategy#returnPermits(int)}.
     * Fallback - sampling is not enabled.
     */
    public static final String POOL_RETURN_PERMITS_SAMPLING_RATE =
"reactor.netty.pool.returnPermitsSamplingRate";
```

When you need to change the default settings, you can configure the `ConnectionProvider` as follows:

github.com/reactor/reactor-netty/tree/main/reactor-netty-examples/src/main/java/reactor/netty/examples/documentation/http/client/pool/config/Application.java

```
import reactor.netty.http.client.HttpClient;
import reactor.netty.resources.ConnectionProvider;

import java.time.Duration;

public class Application {

    public static void main(String[] args) {
        ConnectionProvider provider =
                ConnectionProvider.builder("custom")
                                  .maxConnections(50)
                                  .maxIdleTime(Duration.ofSeconds(20))           ①
                                  .maxLifeTime(Duration.ofSeconds(60))           ②
                                  .pendingAcquireTimeout(Duration.ofSeconds(60)) ③
                                  .evictInBackground(Duration.ofSeconds(120))    ④
                                  .build();

        HttpClient client = HttpClient.create(provider);

        String response =
                client.get()
                      .uri("https://example.com/")
                      .responseContent()
                      .aggregate()
                      .asString()
                      .block();

        System.out.println("Response " + response);

        provider.disposeLater()
                .block();
    }
}
```

① Configures the maximum time for a connection to stay idle to 20 seconds.

② Configures the maximum time for a connection to stay alive to 60 seconds.

③ Configures the maximum time for the pending acquire operation to 60 seconds.

④ Every two minutes, the connection pool is regularly checked for connections that are applicable for removal.

> Notice that only the default `HttpClient` (`HttpClient.create()`) uses `500` as the maximum number of active channels. In the example above, when instantiating a custom `ConnectionProvider`, we are changing this value to `50` using `maxConnections`. Also, if you don't set this parameter the default `maxConnections` is used (2 * available number of processors).

The following listing shows the available configurations:

Configuration name	Description
`disposeInactivePoolsInBackground`	When this option is enabled, connection pools are regularly checked in the background, and those that are **empty** and been inactive for a specified time become eligible for disposal. Connection pool is considered **empty** when there are no active connections, idle connections and pending acquisitions. By default, this background disposal of inactive pools is disabled.
`disposeTimeout`	When `ConnectionProvider#dispose()` or `ConnectionProvider#disposeLater()` is called, trigger a graceful shutdown for the connection pools, with this grace period timeout. From there on, all calls for acquiring a connection will fail fast with an exception. However, for the provided `Duration`, pending acquires will get a chance to be served. Note: The rejection of new acquires and the grace timer start immediately, irrespective of subscription to the `Mono` returned by `ConnectionProvider#disposeLater()`. Subsequent calls return the same `Mono`, effectively getting notifications from the first graceful shutdown call and ignoring subsequently provided timeouts. By default, dispose timeout is not specified.
`evictInBackground`	When this option is enabled, each connection pool regularly checks for connections that are eligible for removal according to eviction criteria like `maxIdleTime`. By default, this background eviction is disabled.

Configuration name	Description
`fifo`	Configure the connection pool so that if there are idle connections (i.e. pool is under-utilized), the next acquire operation will get the `Least Recently Used` connection (LRU, i.e. the connection that was released first among the current idle connections). Default leasing strategy.
`lifo`	Configure the connection pool so that if there are idle connections (i.e. pool is under-utilized), the next acquire operation will get the `Most Recently Used` connection (MRU, i.e. the connection that was released last among the current idle connections).
`maxConnections`	The maximum number of connections (per connection pool) before start pending. Default to 2 * available number of processors (but with a minimum value of 16).
`maxIdleTime`	The time after which the channel is eligible to be closed when idle (resolution: ms). Default: max idle time is not specified.
`maxLifeTime`	The total life time after which the channel is eligible to be closed (resolution: ms). Default: max life time is not specified.
`metrics`	Enables/disables built-in integration with Micrometer. `ConnectionProvider.MeterRegistrar` can be provided for integration with another metrics system. By default, metrics are not enabled.
`pendingAcquireMaxCount`	The maximum number of extra attempts at acquiring a connection to keep in a pending queue. If -1 is specified, the pending queue does not have upper limit. Default to 2 * max connections.
`pendingAcquireTimeout`	The maximum time before which a pending acquire must complete, or a TimeoutException is thrown (resolution: ms). If -1 is specified, no such timeout is applied. Default: 45 seconds.

When you expect a high load, be cautious with a connection pool with a very high value for maximum connections. You might experience `reactor.netty.http.client.PrematureCloseException` exception with a root cause "Connect Timeout" due to too many concurrent connections opened/acquired.

If you need to disable the connection pool, you can apply the following configuration:

github.com/reactor/reactor-netty/tree/main/reactor-netty-examples/src/main/java/reactor/netty/examples/documentation/http/client/pool/Application.java

```java
import reactor.netty.http.client.HttpClient;

public class Application {

    public static void main(String[] args) {
        HttpClient client =
                HttpClient.newConnection()
                          .doOnConnected(conn -> System.out.println("Connection " +
conn.channel()));

        String response =
                // A new connection is established for every request
                client.get()
                      .uri("https://httpbin.org/get")
                      .responseContent()
                      .aggregate()
                      .asString()
                      .block();

        System.out.println("Response " + response);

        response =
                // A new connection is established for every request
                client.post()
                      .uri("https://httpbin.org/post")
                      .responseContent()
                      .aggregate()
                      .asString()
                      .block();

        System.out.println("Response " + response);
    }
}
```

Disposing Connection Pool

- If you use the default `ConnectionProvider` provided by Reactor Netty, invoke `HttpResources#disposeLoopsAndConnections`/`#disposeLoopsAndConnectionsLater` method.

> Disposing `HttpResources` means that every client that is using it, will not be able to use it anymore!

- If you use custom `ConnectionProvider`, invoke `ConnectionProvider#dispose`/`#disposeLater`/`#disposeWhen` method.

> ℹ Disposing the custom `ConnectionProvider` means that every client that is configured to use it, will not be able to use it anymore!

Metrics

The pooled `ConnectionProvider` supports built-in integration with `Micrometer`. It exposes all metrics with a prefix of `reactor.netty.connection.provider`.

Pooled `ConnectionProvider` metrics

metric name	type	description
reactor.netty.connection.provider.total.connections	Gauge	The number of all connections, active or idle. See Total Connections
reactor.netty.connection.provider.active.connections	Gauge	The number of the connections that have been successfully acquired and are in active use. See Active Connections
reactor.netty.connection.provider.max.connections	Gauge	The maximum number of active connections that are allowed. See Max Connections
reactor.netty.connection.provider.idle.connections	Gauge	The number of the idle connections. See Idle Connections
reactor.netty.connection.provider.pending.connections	Gauge	The number of requests that are waiting for a connection. See Pending Connections
reactor.netty.connection.provider.pending.connections.time	Timer	Time spent in pending acquire a connection from the connection pool. See Pending Connections Time
reactor.netty.connection.provider.max.pending.connections	Gauge	The maximum number of requests that will be queued while waiting for a ready connection. See Max Pending Connections

The following table provides information for the HTTP client metrics when it is configured to serve `HTTP/2` traffic:

metric name	type	description
reactor.netty.connection.provider.active.streams	Gauge	The number of the active HTTP/2 streams. See Active Streams

metric name	type	description
reactor.netty.connection.provider.pending.streams	Gauge	The number of requests that are waiting for opening HTTP/2 stream. See Pending Streams

The following example enables that integration:

github.com/reactor/reactor-netty/tree/main/reactor-netty-examples/src/main/java/reactor/netty/examples/documentation/http/client/pool/metrics/Application.java

```
import reactor.netty.http.client.HttpClient;
import reactor.netty.resources.ConnectionProvider;

public class Application {

    public static void main(String[] args) {
        ConnectionProvider provider =
                ConnectionProvider.builder("custom")
                                  .maxConnections(50)
                                  .metrics(true) ①
                                  .build();

        HttpClient client = HttpClient.create(provider);

        String response =
                client.get()
                      .uri("https://example.com/")
                      .responseContent()
                      .aggregate()
                      .asString()
                      .block();

        System.out.println("Response " + response);

        provider.disposeLater()
                .block();
    }
}
```

① Enables the built-in integration with Micrometer

SSL and TLS

When you need SSL or TLS, you can apply the configuration shown in the next example. By default, if `OpenSSL` is available, a SslProvider.OPENSSL provider is used as a provider. Otherwise, a SslProvider.JDK provider is used You can switch the provider by using SslContextBuilder or by setting `-Dio.netty.handler.ssl.noOpenSsl=true`. The following example uses `SslContextBuilder`:

github.com/reactor/reactor-netty/tree/main/reactor-netty-examples/src/main/java/reactor/netty/

examples/documentation/http/client/security/Application.java

```
import reactor.netty.http.Http11SslContextSpec;
import reactor.netty.http.client.HttpClient;

public class Application {

    public static void main(String[] args) {
        Http11SslContextSpec http11SslContextSpec = Http11SslContextSpec.forClient();

        HttpClient client =
                HttpClient.create()
                          .secure(spec -> spec.sslContext(http11SslContextSpec));

        client.get()
              .uri("https://example.com/")
              .response()
              .block();
    }
}
```

Server Name Indication

By default, the HTTP client sends the remote host name as SNI server name. When you need to change this default setting, you can configure the HTTP client as follows:

github.com/reactor/reactor-netty/tree/main/reactor-netty-examples/src/main/java/reactor/netty/examples/documentation/http/client/sni/Application.java

```java
import io.netty.handler.ssl.SslContext;
import io.netty.handler.ssl.SslContextBuilder;
import reactor.netty.http.client.HttpClient;

import javax.net.ssl.SNIHostName;

public class Application {

    public static void main(String[] args) throws Exception {
        SslContext sslContext = SslContextBuilder.forClient().build();

        HttpClient client =
                HttpClient.create()
                          .secure(spec -> spec.sslContext(sslContext)
                                              .serverNames(new SNIHostName(
"test.com")));

        client.get()
              .uri("https://127.0.0.1:8080/")
              .response()
              .block();
    }
}
```

Retry Strategies

By default, the `HTTP` client retries the request once if it was aborted on the `TCP` level.

HTTP/2

By default, the `HTTP` client supports `HTTP/1.1`. If you need `HTTP/2`, you can get it through configuration. In addition to the protocol configuration, if you need `H2` but not `H2C (cleartext)`, you must also configure SSL.

> As Application-Layer Protocol Negotiation (ALPN) is not supported "out-of-the-box" by JDK8 (although some vendors backported ALPN to JDK8), you need an additional dependency to a native library that supports it — for example, `netty-tcnative-boringssl-static`.

The following listing presents a simple `H2` example:

github.com/reactor/reactor-netty/tree/main/reactor-netty-examples/src/main/java/reactor/netty/examples/documentation/http/client/http2/H2Application.java

```java
import io.netty.handler.codec.http.HttpHeaders;
import reactor.core.publisher.Mono;
import reactor.netty.http.HttpProtocol;
import reactor.netty.http.client.HttpClient;
import reactor.util.function.Tuple2;

public class H2Application {

    public static void main(String[] args) {
        HttpClient client =
                HttpClient.create()
                          .protocol(HttpProtocol.H2) ①
                          .secure();                 ②

        Tuple2<String, HttpHeaders> response =
                client.get()
                      .uri("https://example.com/")
                      .responseSingle((res, bytes) -> bytes.asString()
                                                           .zipWith(Mono.just(res
.responseHeaders())))
                      .block();

        System.out.println("Used stream ID: " + response.getT2().get("x-http2-stream-
id"));
        System.out.println("Response: " + response.getT1());
    }
}
```

① Configures the client to support only `HTTP/2`

② Configures `SSL`

The following listing presents a simple `H2C` example:

github.com/reactor/reactor-netty/tree/main/reactor-netty-examples/src/main/java/reactor/netty/examples/documentation/http/client/http2/H2CApplication.java

```
import io.netty.handler.codec.http.HttpHeaders;
import reactor.core.publisher.Mono;
import reactor.netty.http.HttpProtocol;
import reactor.netty.http.client.HttpClient;
import reactor.util.function.Tuple2;

public class H2CApplication {

    public static void main(String[] args) {
        HttpClient client =
                HttpClient.create()
                          .protocol(HttpProtocol.H2C);

        Tuple2<String, HttpHeaders> response =
                client.get()
                      .uri("http://localhost:8080/")
                      .responseSingle((res, bytes) -> bytes.asString()
                                                           .zipWith(Mono.just(res
.responseHeaders())))
                      .block();

        System.out.println("Used stream ID: " + response.getT2().get("x-http2-stream-
id"));
        System.out.println("Response: " + response.getT1());
    }
}
```

Protocol Selection

github.com/reactor/reactor-netty/tree/main/reactor-netty-http/src/main/java/reactor/netty/http/
HttpProtocol.java

```
 * @author Stephane Maldini
 */
public enum HttpProtocol {

    /**
     * The default supported HTTP protocol by HttpServer and HttpClient.
     */
    HTTP11,

    /**
     * HTTP/2.0 support with TLS
     * <p>If used along with HTTP/1.1 protocol, HTTP/2.0 will be the preferred
protocol.
     * While negotiating the application level protocol, HTTP/2.0 or HTTP/1.1 can be
chosen.
     * <p>If used without HTTP/1.1 protocol, HTTP/2.0 will always be offered as a
protocol
     * for communication with no fallback to HTTP/1.1.
     */
    H2,

    /**
     * HTTP/2.0 support with clear-text.
     * <p>If used along with HTTP/1.1 protocol, will support H2C "upgrade":
     * Request or consume requests as HTTP/1.1 first, looking for HTTP/2.0 headers
     * and {@literal Connection: Upgrade}. A server will typically reply a successful
     * 101 status if upgrade is successful or a fallback HTTP/1.1 response. When
     * successful the client will start sending HTTP/2.0 traffic.
     * <p>If used without HTTP/1.1 protocol, will support H2C "prior-knowledge":
Doesn't
     * require {@literal Connection: Upgrade} handshake between a client and server
but
     * fallback to HTTP/1.1 will not be supported.
     */
    H2C,

    /**
     * HTTP/3.0 support.
     * @since 1.2.0
     */
```

Proxy Support

Reactor Netty supports the proxy functionality provided by Netty and provides a way to specify `non proxy hosts` through the `ProxyProvider` builder.

Netty's HTTP proxy support always uses `CONNECT` method in order to establish a tunnel to the specified proxy regardless of the scheme that is used `http` or `https`. (More information: Netty enforce HTTP proxy to support HTTP CONNECT method). Some proxies might not support `CONNECT` method when the scheme is `http` or might need to be configured in order to support this way of communication. Sometimes this might be the reason for not being able to connect to the proxy. Consider checking the proxy documentation whether it supports or needs an additional configuration in order to support `CONNECT` method.

The following example uses `ProxyProvider`:

github.com/reactor/reactor-netty/tree/main/reactor-netty-examples/src/main/java/reactor/netty/examples/documentation/http/client/proxy/Application.java

```
import reactor.netty.http.client.HttpClient;
import reactor.netty.transport.ProxyProvider;

public class Application {

    public static void main(String[] args) {
        HttpClient client =
                HttpClient.create()
                          .proxy(spec -> spec.type(ProxyProvider.Proxy.HTTP)
                                             .host("proxy")
                                             .port(8080)
                                             .nonProxyHosts("localhost")
                                             .connectTimeoutMillis(20_000)); ①

        String response =
                client.get()
                      .uri("https://example.com/")
                      .responseContent()
                      .aggregate()
                      .asString()
                      .block();

        System.out.println("Response " + response);
    }
}
```

① Configures the connection establishment timeout to 20 seconds.

Metrics

The HTTP client supports built-in integration with `Micrometer`. It exposes all metrics with a prefix of `reactor.netty.http.client`.

The following table provides information for the HTTP client metrics:

metric name	type	description
reactor.netty.http.client.data.received	DistributionSummary	Amount of the data received, in bytes. See Data Received

metric name	type	description
reactor.netty.http.client.data.sent	DistributionSummary	Amount of the data sent, in bytes. See Data Sent
reactor.netty.http.client.errors	Counter	Number of errors that occurred. See Errors Count
reactor.netty.http.client.tls.handshake.time	Timer	Time spent for TLS handshake. See Tls Handshake Time
reactor.netty.http.client.connect.time	Timer	Time spent for connecting to the remote address. See Connect Time
reactor.netty.http.client.address.resolver	Timer	Time spent for resolving the address. See Hostname Resolution Time
reactor.netty.http.client.data.received.time	Timer	Time spent in consuming incoming data. See Http Client Data Received Time
reactor.netty.http.client.data.sent.time	Timer	Time spent in sending outgoing data. See Http Client Data Sent Time
reactor.netty.http.client.response.time	Timer	Total time for the request/response See Http Client Response Time

These additional metrics are also available:

Pooled `ConnectionProvider` metrics

metric name	type	description
reactor.netty.connection.provider.total.connections	Gauge	The number of all connections, active or idle. See Total Connections
reactor.netty.connection.provider.active.connections	Gauge	The number of the connections that have been successfully acquired and are in active use. See Active Connections
reactor.netty.connection.provider.max.connections	Gauge	The maximum number of active connections that are allowed. See Max Connections
reactor.netty.connection.provider.idle.connections	Gauge	The number of the idle connections. See Idle Connections

metric name	type	description
reactor.netty.connection.provider.pending.connections	Gauge	The number of requests that are waiting for a connection. See Pending Connections
reactor.netty.connection.provider.pending.connections.time	Timer	Time spent in pending acquire a connection from the connection pool. See Pending Connections Time
reactor.netty.connection.provider.max.pending.connections	Gauge	The maximum number of requests that will be queued while waiting for a ready connection. See Max Pending Connections

The following table provides information for the HTTP client metrics when it is configured to serve `HTTP/2` traffic:

metric name	type	description
reactor.netty.connection.provider.active.streams	Gauge	The number of the active HTTP/2 streams. See Active Streams
reactor.netty.connection.provider.pending.streams	Gauge	The number of requests that are waiting for opening HTTP/2 stream. See Pending Streams

`ByteBufAllocator` metrics

metric name	type	description
reactor.netty.bytebuf.allocator.used.heap.memory	Gauge	The number of bytes reserved by heap buffer allocator. See Used Heap Memory
reactor.netty.bytebuf.allocator.used.direct.memory	Gauge	The number of bytes reserved by direct buffer allocator. See Used Direct Memory
reactor.netty.bytebuf.allocator.heap.arenas	Gauge	The number of heap arenas (when `PooledByteBufAllocator`). See Heap Arenas
reactor.netty.bytebuf.allocator.direct.arenas	Gauge	The number of direct arenas (when `PooledByteBufAllocator`). See Direct Arenas
reactor.netty.bytebuf.allocator.threadlocal.caches	Gauge	The number of thread local caches (when `PooledByteBufAllocator`). See Thread Local Caches

metric name	type	description
reactor.netty.bytebuf.allocator.small.cache.size	Gauge	The size of the small cache (when `PooledByteBufAllocator`). See Small Cache Size
reactor.netty.bytebuf.allocator.normal.cache.size	Gauge	The size of the normal cache (when `PooledByteBufAllocator`). See Normal Cache Size
reactor.netty.bytebuf.allocator.chunk.size	Gauge	The chunk size for an arena (when `PooledByteBufAllocator`). See Chunk Size
reactor.netty.bytebuf.allocator.active.heap.memory	Gauge	The actual bytes consumed by in-use buffers allocated from heap buffer pools (when `PooledByteBufAllocator`). See Active Heap Memory
reactor.netty.bytebuf.allocator.active.direct.memory	Gauge	The actual bytes consumed by in-use buffers allocated from direct buffer pools (when `PooledByteBufAllocator`). See Active Direct Memory

`EventLoop` metrics

metric name	type	description
reactor.netty.eventloop.pending.tasks	Gauge	The number of tasks that are pending for processing on an event loop. See Pending Tasks

The following example enables that integration:

github.com/reactor/reactor-netty/tree/main/reactor-netty-examples/src/main/java/reactor/netty/examples/documentation/http/client/metrics/Application.java

```java
import io.micrometer.core.instrument.Metrics;
import io.micrometer.core.instrument.config.MeterFilter;
import reactor.netty.http.client.HttpClient;

public class Application {

    public static void main(String[] args) {
        Metrics.globalRegistry ①
               .config()
               .meterFilter(MeterFilter.maximumAllowableTags
("reactor.netty.http.client", "URI", 100, MeterFilter.deny()));

        HttpClient client =
                HttpClient.create()
                          .metrics(true, s -> {
                              if (s.startsWith("/stream/")) { ②
                                  return "/stream/{n}";
                              }
                              else if (s.startsWith("/bytes/")) {
                                  return "/bytes/{n}";
                              }
                              return s;
                          }); ③

        client.get()
              .uri("https://httpbin.org/stream/2")
              .responseContent()
              .blockLast();

        client.get()
              .uri("https://httpbin.org/bytes/1024")
              .responseContent()
              .blockLast();
    }
}
```

① Applies upper limit for the meters with `URI` tag

② Templated URIs will be used as a URI tag value when possible

③ Enables the built-in integration with Micrometer

> In order to avoid a memory and CPU overhead of the enabled metrics, it is important to convert the real URIs to templated URIs when possible. Without a conversion to a template-like form, each distinct URI leads to the creation of a distinct tag, which takes a lot of memory for the metrics.

> Always apply an upper limit for the meters with URI tags. Configuring an upper limit on the number of meters can help in cases when the real URIs cannot be templated. You can find more information at `maximumAllowableTags`.

When HTTP client metrics are needed for an integration with a system other than Micrometer or you want to provide your own integration with Micrometer, you can provide your own metrics recorder, as follows:

github.com/reactor/reactor-netty/tree/main/reactor-netty-examples/src/main/java/reactor/netty/examples/documentation/http/client/metrics/custom/Application.java

```
import reactor.netty.http.client.HttpClient;
import reactor.netty.http.client.HttpClientMetricsRecorder;

import java.net.SocketAddress;
import java.time.Duration;

public class Application {

    public static void main(String[] args) {
        HttpClient client =
                HttpClient.create()
                          .metrics(true, CustomHttpClientMetricsRecorder::new); ①

        client.get()
              .uri("https://httpbin.org/stream/2")
              .response()
              .block();
    }
```

① Enables HTTP client metrics and provides HttpClientMetricsRecorder implementation.

Tracing

The HTTP client supports built-in integration with Micrometer Tracing.

The following table provides information for the HTTP client spans:

contextual name	description
HTTP <HTTP METHOD>	Information and total time for the request. See Http Client Response Span.
hostname resolution	Information and time spent for resolving the address. See Hostname Resolution Span.
connect	Information and time spent for connecting to the remote address. See Connect Span.
tls handshake	Information and time spent for TLS handshake. See Tls Handshake Span.

The following example enables that integration. This concrete example uses Brave and reports the information to Zipkin. See the Micrometer Tracing documentation for OpenTelemetry setup.

github.com/reactor/reactor-netty/tree/main/reactor-netty-examples/src/main/java/reactor/netty/examples/documentation/http/client/tracing/Application.java

```
import brave.Tracing;
import brave.propagation.StrictCurrentTraceContext;
import brave.sampler.Sampler;
import io.micrometer.tracing.CurrentTraceContext;
import io.micrometer.tracing.Tracer;
import io.micrometer.tracing.brave.bridge.BraveBaggageManager;
import io.micrometer.tracing.brave.bridge.BraveCurrentTraceContext;
import io.micrometer.tracing.brave.bridge.BravePropagator;
import io.micrometer.tracing.brave.bridge.BraveTracer;
import io.micrometer.tracing.propagation.Propagator;
import reactor.netty.http.client.HttpClient;
import
reactor.netty.http.observability.ReactorNettyPropagatingSenderTracingObservationHandle
r;
import reactor.netty.observability.ReactorNettyTracingObservationHandler;
import zipkin2.reporter.brave.AsyncZipkinSpanHandler;
import zipkin2.reporter.urlconnection.URLConnectionSender;

import static reactor.netty.Metrics.OBSERVATION_REGISTRY;

public class Application {

    public static void main(String[] args) {
        init(); ①

        HttpClient client =
                HttpClient.create()
                          .metrics(true, s -> {
                              if (s.startsWith("/stream/")) { ②
                                  return "/stream/{n}";
                              }
                              return s;
                          }); ③

        client.get()
              .uri("https://httpbin.org/stream/3")
              .responseContent()
              .blockLast();
    }

    /**
     * This setup is based on
     * <a
href="https://micrometer.io/docs/tracing#_micrometer_tracing_brave_setup">Micrometer
Tracing Brave Setup</a>.
     */
    static void init() {
```

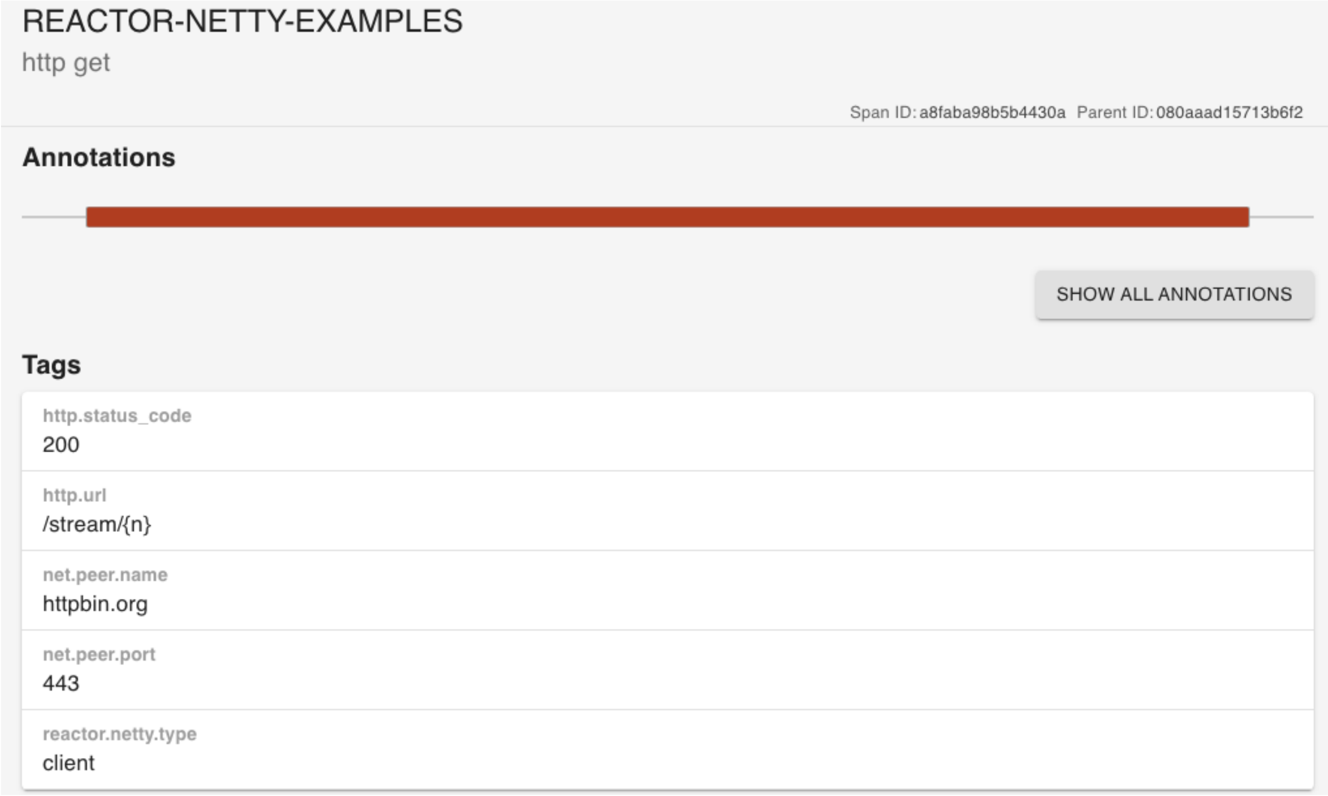
```
        AsyncZipkinSpanHandler spanHandler = AsyncZipkinSpanHandler
                .create(URLConnectionSender.create
("http://localhost:9411/api/v2/spans"));

        StrictCurrentTraceContext braveCurrentTraceContext =
StrictCurrentTraceContext.create();

        CurrentTraceContext bridgeContext = new BraveCurrentTraceContext
(braveCurrentTraceContext);

        Tracing tracing =
                Tracing.newBuilder()
                       .currentTraceContext(braveCurrentTraceContext)
                       .supportsJoin(false)
                       .traceId128Bit(true)
                       .sampler(Sampler.ALWAYS_SAMPLE)
                       .addSpanHandler(spanHandler)
                       .localServiceName("reactor-netty-examples")
                       .build();

        brave.Tracer braveTracer = tracing.tracer();

        Tracer tracer = new BraveTracer(braveTracer, bridgeContext, new
BraveBaggageManager());

        Propagator propagator = new BravePropagator(tracing);

        OBSERVATION_REGISTRY.observationConfig()
                            .observationHandler(new
ReactorNettyPropagatingSenderTracingObservationHandler(tracer, propagator))
                            .observationHandler(new
ReactorNettyTracingObservationHandler(tracer));
    }
}
```

① Initializes Brave, Zipkin, and the Observation registry.

② Templated URIs are used as an URI tag value when possible.

③ Enables the built-in integration with Micrometer.

The result in `Zipkin` looks like:



Access Current Observation

Project Micrometer provides `a library` that assists with context propagation across different types of context mechanisms such as `ThreadLocal`, `Reactor Context` and others.

The following example shows how to use this library in a custom `ChannelHandler`:

github.com/reactor/reactor-netty/tree/main/reactor-netty-examples/src/main/java/reactor/netty/examples/documentation/http/client/tracing/custom/Application.java

```
import brave.Tracing;
import brave.propagation.StrictCurrentTraceContext;
import brave.sampler.Sampler;
import io.micrometer.context.ContextSnapshot;
import io.micrometer.context.ContextSnapshotFactory;
import io.micrometer.tracing.CurrentTraceContext;
import io.micrometer.tracing.Tracer;
import io.micrometer.tracing.brave.bridge.BraveBaggageManager;
import io.micrometer.tracing.brave.bridge.BraveCurrentTraceContext;
import io.micrometer.tracing.brave.bridge.BravePropagator;
import io.micrometer.tracing.brave.bridge.BraveTracer;
import io.micrometer.tracing.propagation.Propagator;
import io.netty.channel.ChannelHandler;
import io.netty.channel.ChannelHandlerContext;
import io.netty.channel.ChannelOutboundHandlerAdapter;
import io.netty.channel.ChannelPromise;
import reactor.netty.NettyPipeline;
import reactor.netty.http.client.HttpClient;
import
```

```
reactor.netty.http.observability.ReactorNettyPropagatingSenderTracingObservationHandle
r;
import reactor.netty.observability.ReactorNettyTracingObservationHandler;
import zipkin2.reporter.brave.AsyncZipkinSpanHandler;
import zipkin2.reporter.urlconnection.URLConnectionSender;

import static reactor.netty.Metrics.OBSERVATION_REGISTRY;

public class Application {

    public static void main(String[] args) {
        init(); ①

        HttpClient client =
                HttpClient.create()
                          .metrics(true, s -> {
                              if (s.startsWith("/stream/")) { ②
                                  return "/stream/{n}";
                              }
                              return s;
                          }) ③
                          .doOnConnected(conn -> conn.channel().pipeline().addAfter
(NettyPipeline.HttpCodec,
                                  "custom-channel-handler",
CustomChannelOutboundHandler.INSTANCE)); ④

        client.get()
              .uri("https://httpbin.org/stream/3")
              .responseContent()
              .blockLast();
    }

    static final class CustomChannelOutboundHandler extends
ChannelOutboundHandlerAdapter {

        static final ChannelHandler INSTANCE = new CustomChannelOutboundHandler();

        @Override
        public boolean isSharable() {
            return true;
        }

        @Override
        @SuppressWarnings({"FutureReturnValueIgnored", "try"})
        public void write(ChannelHandlerContext ctx, Object msg, ChannelPromise
promise) {
            try (ContextSnapshot.Scope scope = ContextSnapshotFactory.builder().
build().setThreadLocalsFrom(ctx.channel())) {
                System.out.println("Current Observation in Scope: " +
OBSERVATION_REGISTRY.getCurrentObservation());
                //"FutureReturnValueIgnored" this is deliberate
```

```
                ctx.write(msg, promise);
            }
            System.out.println("Current Observation: " + OBSERVATION_REGISTRY
.getCurrentObservation());
        }
    }
```

① Initializes Brave, Zipkin, and the Observation registry.

② Templated URIs are used as an URI tag value when possible.

③ Enables the built-in integration with Micrometer.

④ Custom `ChannelHandler` that uses context propagation library. This concrete example overrides only `ChannelOutboundHandlerAdapter#write`, if it is needed, the same logic can be used for the rest of the methods. Also, this concrete example sets all `ThreadLocal` values for which there is a value in the given `Channel`, if another behaviour is needed please check `context propagation library API`. For example, you may want to set only some of the `ThreadLocal` values.

> When you enable Reactor Netty tracing within a framework, you may need to let Reactor Netty use the `ObservationRegistry` created by this framework. For this purpose you need to invoke `reactor.netty.Metrics#observationRegistry`. You may also need to configure the Reactor Netty `ObservationHandlers` using the API provided by the framework.

Unix Domain Sockets

The `HTTP` client supports Unix Domain Sockets (UDS) when native transport is in use.

The following example shows how to use UDS support:

github.com/reactor/reactor-netty/tree/main/reactor-netty-examples/src/main/java/reactor/netty/examples/documentation/http/client/uds/Application.java

```
import io.netty.channel.unix.DomainSocketAddress;
import reactor.netty.http.client.HttpClient;

public class Application {

    public static void main(String[] args) {
        HttpClient client =
                HttpClient.create()
                          .remoteAddress(() -> new DomainSocketAddress
("/tmp/test.sock")); ①

        client.get()
              .uri("/")
              .response()
              .block();
    }
```

```
}
```

① Specifies `DomainSocketAddress` that will be used

Host Name Resolution

By default, the `HttpClient` uses Netty's domain name lookup mechanism that resolves a domain name asynchronously. This is as an alternative of the JVM's built-in blocking resolver.

When you need to change the default settings, you can configure the `HttpClient` as follows:

github.com/reactor/reactor-netty/tree/main/reactor-netty-examples/src/main/java/reactor/netty/examples/documentation/http/client/resolver/Application.java

```
import reactor.netty.http.client.HttpClient;

import java.time.Duration;

public class Application {

    public static void main(String[] args) {
        HttpClient client =
                HttpClient.create()
                          .resolver(spec -> spec.queryTimeout(Duration.ofMillis(
500))); ①

        String response =
                client.get()
                      .uri("https://example.com/")
                      .responseContent()
                      .aggregate()
                      .asString()
                      .block();

        System.out.println("Response " + response);
    }
}
```

① The timeout of each DNS query performed by this resolver will be 500ms.

The following listing shows the available configurations. Additionally, `TCP fallback` is enabled by default.

Configuration name	Description
`bindAddressSupplier`	The supplier of the local address to bind to.

Configuration name	Description
cacheMaxTimeToLive	The max time to live of the cached DNS resource records (resolution: seconds). If the time to live of the DNS resource record returned by the DNS server is greater than this max time to live, this resolver ignores the time to live from the DNS server and uses this max time to live. Default to Integer.MAX_VALUE.
cacheMinTimeToLive	The min time to live of the cached DNS resource records (resolution: seconds). If the time to live of the DNS resource record returned by the DNS server is less than this min time to live, this resolver ignores the time to live from the DNS server and uses this min time to live. Default: 0.
cacheNegativeTimeToLive	The time to live of the cache for the failed DNS queries (resolution: seconds). Default: 0.
completeOncePreferredResolved	When this setting is enabled, the resolver notifies as soon as all queries for the preferred address type are complete. When this setting is disabled, the resolver notifies when all possible address types are complete. This configuration is applicable for DnsNameResolver#resolveAll(String). By default, this setting is enabled.
disableOptionalRecord	Disables the automatic inclusion of an optional record that tries to give a hint to the remote DNS server about how much data the resolver can read per response. By default, this setting is enabled.
disableRecursionDesired	Specifies whether this resolver has to send a DNS query with the recursion desired (RD) flag set. By default, this setting is enabled.
dnsAddressResolverGroupProvider	Sets a custom function to create a DnsAddressResolverGroup given a DnsNameResolverBuilder
hostsFileEntriesResolver	Sets a custom HostsFileEntriesResolver to be used for hosts file entries. Default: DefaultHostsFileEntriesResolver.
maxPayloadSize	Sets the capacity of the datagram packet buffer (in bytes). Default: 4096.
maxQueriesPerResolve	Sets the maximum allowed number of DNS queries to send when resolving a host name. Default: 16.

Configuration name	Description
`ndots`	Sets the number of dots that must appear in a name before an initial absolute query is made. Default: -1 (to determine the value from the OS on Unix or use a value of 1 otherwise).
`queryTimeout`	Sets the timeout of each DNS query performed by this resolver (resolution: milliseconds). Default: 5000.
`resolveCache`	The cache to use to store resolved DNS entries.
`resolvedAddressTypes`	The list of the protocol families of the resolved address.
`retryTcpOnTimeout`	Specifies whether this resolver will also fallback to TCP if a timeout is detected. By default, the resolver will only try to use TCP if the response is marked as truncated.
`roundRobinSelection`	Enables an `AddressResolverGroup` of `DnsNameResolver` that supports random selection of destination addresses if multiple are provided by the nameserver. See `RoundRobinDnsAddressResolverGroup`. Default: `DnsAddressResolverGroup`
`runOn`	Performs the communication with the DNS servers on the given `LoopResources`. By default, the LoopResources specified on the client level are used.
`searchDomains`	The list of search domains of the resolver. By default, the effective search domain list is populated by using the system DNS search domains.
`trace`	A specific logger and log level to be used by this resolver when generating detailed trace information in case of resolution failure.

Sometimes, you may want to switch to the JVM built-in resolver. To do so, you can configure the `HttpClient` as follows:

github.com/reactor/reactor-netty/tree/main/reactor-netty-examples/src/main/java/reactor/netty/examples/documentation/http/client/resolver/custom/Application.java

```
import io.netty.resolver.DefaultAddressResolverGroup;
import reactor.netty.http.client.HttpClient;

public class Application {
```

```java
    public static void main(String[] args) {
        HttpClient client =
                HttpClient.create()
                          .resolver(DefaultAddressResolverGroup.INSTANCE); ①

        String response =
                client.get()
                      .uri("https://example.com/")
                      .responseContent()
                      .aggregate()
                      .asString()
                      .block();

        System.out.println("Response " + response);
    }
}
```

① Sets the JVM built-in resolver.

Timeout Configuration

This section describes various timeout configuration options that can be used in `HttpClient`. Configuring a proper timeout may improve or solve issues in the communication process. The configuration options can be grouped as follows:

- Connection Pool Timeout
- HttpClient Timeout
  - Response Timeout
  - Connection Timeout
  - SSL/TLS Timeout
  - Proxy Timeout
  - Host Name Resolution Timeout

Connection Pool Timeout

By default, `HttpClient` uses a connection pool. When a request is completed successfully and if the connection is not scheduled for closing, the connection is returned to the connection pool and can thus be reused for processing another request. The connection may be reused immediately for another request or may stay idle in the connection pool for some time.

The following list describes the available timeout configuration options:

- `maxIdleTime` - The maximum time (resolution: ms) that this connection stays idle in the connection pool. By default, `maxIdleTime` is not specified.

> When you configure `maxIdleTime`, you should consider the idle timeout configuration on the target server. Choose a configuration that is equal to or less

> than the one on the target server. By doing so, you can reduce the I/O issues caused by a connection closed by the target server.

- `maxLifeTime` - The maximum time (resolution: ms) that this connection stays alive. By default, `maxLifeTime` is not specified.
- `pendingAcquireTimeout` - The maximum time (resolution: ms) after which a pending acquire operation must complete, or a `PoolAcquireTimeoutException` is thrown. Default: 45s.

By default, these timeouts are checked on connection `release` or `acquire` operations and, if some timeout is reached, the connection is closed and removed from the connection pool. However, you can also configure the connection pool, by setting `evictInBackground`, to perform periodic checks on connections.

To customize the default settings, you can configure `HttpClient` as follows:

github.com/reactor/reactor-netty/tree/main/reactor-netty-examples/src/main/java/reactor/netty/examples/documentation/http/client/pool/Application.java

```
import reactor.netty.http.client.HttpClient;
import reactor.netty.resources.ConnectionProvider;

import java.time.Duration;

public class Application {

    public static void main(String[] args) {
        ConnectionProvider provider =
                ConnectionProvider.builder("custom")
                                  .maxConnections(50)
                                  .maxIdleTime(Duration.ofSeconds(20))           ①
                                  .maxLifeTime(Duration.ofSeconds(60))           ②
                                  .pendingAcquireTimeout(Duration.ofSeconds(60)) ③
                                  .evictInBackground(Duration.ofSeconds(120))    ④
                                  .build();

        HttpClient client = HttpClient.create(provider);

        String response =
                client.get()
                      .uri("https://example.com/")
                      .responseContent()
                      .aggregate()
                      .asString()
                      .block();

        System.out.println("Response " + response);

        provider.disposeLater()
                .block();
    }
```

```
}
```

① Configures the maximum time for a connection to stay idle to 20 seconds.

② Configures the maximum time for a connection to stay alive to 60 seconds.

③ Configures the maximum time for the pending acquire operation to 60 seconds.

④ Every two minutes, the connection pool is regularly checked for connections that are applicable for removal.

HttpClient Timeout

This section provides information for the various timeout configuration options at the `HttpClient` level.

> Reactor Netty uses Reactor Core as its Reactive Streams implementation, and you may want to use the `timeout` operator that `Mono` and `Flux` provide. Keep in mind, however, that it is better to use the more specific timeout configuration options available in Reactor Netty, since they provide more control for a specific purpose and use case. By contrast, the `timeout` operator can only apply to the operation as a whole, from establishing the connection to the remote peer to receiving the response.

Response Timeout

`HttpClient` provides an API for configuring a default response timeout for all requests. You can change this default response timeout through an API for a specific request. By default, `responseTimeout` is not specified.

> It is always a good practice to configure a response timeout.

To customize the default settings, you can configure `HttpClient` as follows:

github.com/reactor/reactor-netty/tree/main/reactor-netty-examples/src/main/java/reactor/netty/examples/documentation/http/client/read/timeout/Application.java

```
import reactor.core.publisher.Mono;
import reactor.netty.http.client.HttpClient;

import java.time.Duration;

public class Application {

    public static void main(String[] args) {
        HttpClient client =
                HttpClient.create()
                          .responseTimeout(Duration.ofSeconds(1)); ①

        String response1 =
```

```
                client.post()
                      .uri("https://example.com/")
                      .send((req, out) -> {
                          req.responseTimeout(Duration.ofSeconds(2)); ②
                          return out.sendString(Mono.just("body1"));
                      })
                      .responseContent()
                      .aggregate()
                      .asString()
                      .block();

        System.out.println("Response " + response1);

        String response2 =
                client.post()
                      .uri("https://example.com/")
                      .send((req, out) -> out.sendString(Mono.just("body2")))
                      .responseContent()
                      .aggregate()
                      .asString()
                      .block();

        System.out.println("Response " + response2);
    }
}
```

① Configures the default response timeout to 1 second.

② Configures a response timeout for a specific request to 2 seconds.

Connection Timeout

The following listing shows all available connection timeout configuration options, but some of them may apply only to a specific transport.

- `CONNECT_TIMEOUT_MILLIS` - If the connection establishment attempt to the remote peer does not finish within the configured connect timeout (resolution: ms), the connection establishment attempt fails. Default: 30s.
- `SO_KEEPALIVE` - When the connection stays idle for some time (the time is implementation dependent, but the default is typically two hours), TCP automatically sends a `keepalive` probe to the remote peer. By default, `SO_KEEPALIVE` is not enabled. When you run with `Epoll`/`NIO` (since **Java 11** on **Mac or Linux**) transport, you may also configure:
  - `TCP_KEEPIDLE` - The maximum time (resolution: seconds) that this connection stays idle before TCP starts sending `keepalive` probes, if `SO_KEEPALIVE` has been set. The maximum time is implementation dependent, but the default is typically two hours.
  - `TCP_KEEPINTVL` (Epoll)/`TCP_KEEPINTERVAL` (NIO) - The time (resolution: seconds) between individual `keepalive` probes.
  - `TCP_KEEPCNT` (Epoll)/`TCP_KEEPCOUNT` (NIO) - The maximum number of `keepalive` probes TCP should send before dropping the connection.

Sometimes, between the client and the server, you may have a network component that silently drops the idle connections without sending a response. From the Reactor Netty point of view, in this use case, the remote peer just does not respond. To be able to handle such a use case you may consider configuring `SO_KEEPALIVE`.

To customize the default settings, you can configure `HttpClient` as follows:

github.com/reactor/reactor-netty/tree/main/reactor-netty-examples/src/main/java/reactor/netty/examples/documentation/http/client/channeloptions/Application.java

```
import io.netty.channel.ChannelOption;
import io.netty.channel.epoll.EpollChannelOption;
//import io.netty.channel.socket.nio.NioChannelOption;
//import jdk.net.ExtendedSocketOptions;
import reactor.netty.http.client.HttpClient;
import java.net.InetSocketAddress;

public class Application {

    public static void main(String[] args) {
        HttpClient client =
                HttpClient.create()
                          .bindAddress(() -> new InetSocketAddress("host", 1234))
                          .option(ChannelOption.CONNECT_TIMEOUT_MILLIS, 10000) ①
                          .option(ChannelOption.SO_KEEPALIVE, true)            ②
                          // The options below are available only when NIO transport
(Java 11) is used
                          // on Mac or Linux (Java does not currently support these
extended options on Windows)
                          // https://bugs.openjdk.java.net/browse/JDK-8194298

//.option(NioChannelOption.of(ExtendedSocketOptions.TCP_KEEPIDLE), 300)

//.option(NioChannelOption.of(ExtendedSocketOptions.TCP_KEEPINTERVAL), 60)

//.option(NioChannelOption.of(ExtendedSocketOptions.TCP_KEEPCOUNT), 8);
                          // The options below are available only when Epoll transport
is used
                          .option(EpollChannelOption.TCP_KEEPIDLE, 300)        ③
                          .option(EpollChannelOption.TCP_KEEPINTVL, 60)        ④
                          .option(EpollChannelOption.TCP_KEEPCNT, 8);          ⑤

        String response =
                client.get()
                      .uri("https://example.com/")
                      .responseContent()
                      .aggregate()
                      .asString()
                      .block();
```

```
        System.out.println("Response " + response);
    }
}
```

① Configures the connection establishment timeout to 10 seconds.

② Enables TCP `keepalive`. This means that TCP starts sending `keepalive` probes when a connection is idle for some time.

③ The connection needs to remain idle for 5 minutes before TCP starts sending `keepalive` probes.

④ Configures the time between individual `keepalive` probes to 1 minute.

⑤ Configures the maximum number of TCP `keepalive` probes to 8.

SSL/TLS Timeout

`HttpClient` supports the SSL/TLS functionality provided by Netty.

The following list describes the available timeout configuration options:

- `handshakeTimeout` - Use this option to configure the SSL handshake timeout (resolution: ms). Default: 10s.

  > You should consider increasing the SSL handshake timeout when expecting slow network connections.

- `closeNotifyFlushTimeout` - Use this option to configure the SSL `close_notify` flush timeout (resolution: ms). Default: 3s.
- `closeNotifyReadTimeout` - Use this option to configure the SSL `close_notify` read timeout (resolution: ms). Default: 0s.

To customize the default settings, you can configure `HttpClient` as follows:

github.com/reactor/reactor-netty/tree/main/reactor-netty-examples/src/main/java/reactor/netty/examples/documentation/http/client/security/custom/Application.java

```
import reactor.netty.http.Http11SslContextSpec;
import reactor.netty.http.client.HttpClient;

import java.time.Duration;

public class Application {

    public static void main(String[] args) {
        Http11SslContextSpec http11SslContextSpec = Http11SslContextSpec.forClient();

        HttpClient client =
                HttpClient.create()
                          .secure(spec -> spec.sslContext(http11SslContextSpec)
                                              .handshakeTimeout(Duration.ofSeconds(
```

```
30))        ①
                                                        .closeNotifyFlushTimeout(Duration
.ofSeconds(10))  ②
                                                        .closeNotifyReadTimeout(Duration
.ofSeconds(10))); ③

        String response =
                client.get()
                      .uri("https://example.com/")
                      .responseContent()
                      .aggregate()
                      .asString()
                      .block();

        System.out.println("Response " + response);
    }
}
```

① Configures the SSL handshake timeout to 30 seconds.

② Configures the SSL `close_notify` flush timeout to 10 seconds.

③ Configures the SSL `close_notify` read timeout to 10 seconds.

Proxy Timeout

`HttpClient` supports the proxy functionality provided by Netty and provides a way to specify the connection establishment timeout. If the connection establishment attempt to the remote peer does not finish within the timeout, the connection establishment attempt fails. Default: 10s.

To customize the default settings, you can configure `HttpClient` as follows:

github.com/reactor/reactor-netty/tree/main/reactor-netty-examples/src/main/java/reactor/netty/examples/documentation/http/client/proxy/Application.java

```
import reactor.netty.http.client.HttpClient;
import reactor.netty.transport.ProxyProvider;

public class Application {

    public static void main(String[] args) {
        HttpClient client =
                HttpClient.create()
                          .proxy(spec -> spec.type(ProxyProvider.Proxy.HTTP)
                                             .host("proxy")
                                             .port(8080)
                                             .nonProxyHosts("localhost")
                                             .connectTimeoutMillis(20_000)); ①

        String response =
                client.get()
```

```
                        .uri("https://example.com/")
                        .responseContent()
                        .aggregate()
                        .asString()
                        .block();

        System.out.println("Response " + response);
    }
}
```

① Configures the connection establishment timeout to 20 seconds.

Host Name Resolution Timeout

By default, the HttpClient uses Netty's domain name lookup mechanism to resolve a domain name asynchronously.

The following list describes the available timeout configuration options:

- cacheMaxTimeToLive - The maximum time to live of the cached DNS resource records (resolution: seconds). If the time to live of the DNS resource record returned by the DNS server is greater than this maximum time to live, this resolver ignores the time to live from the DNS server and uses this maximum time to live. Default: Integer.MAX_VALUE.
- cacheMinTimeToLive - The minimum time to live of the cached DNS resource records (resolution: seconds). If the time to live of the DNS resource record returned by the DNS server is less than this minimum time to live, this resolver ignores the time to live from the DNS server and uses this minimum time to live. Default: 0s.
- cacheNegativeTimeToLive - The time to live of the cache for the failed DNS queries (resolution: seconds). Default: 0s.
- queryTimeout - Sets the timeout of each DNS query performed by this resolver (resolution: milliseconds). Default: 5s.

To customize the default settings, you can configure HttpClient as follows:

github.com/reactor/reactor-netty/tree/main/reactor-netty-examples/src/main/java/reactor/netty/examples/documentation/http/client/resolver/Application.java

```
import reactor.netty.http.client.HttpClient;

import java.time.Duration;

public class Application {

    public static void main(String[] args) {
        HttpClient client =
                HttpClient.create()
                          .resolver(spec -> spec.queryTimeout(Duration.ofMillis(
500))); ①
```

```java
        String response =
                client.get()
                      .uri("https://example.com/")
                      .responseContent()
                      .aggregate()
                      .asString()
                      .block();

        System.out.println("Response " + response);
    }
}
```

① The timeout of each DNS query performed by this resolver will be 500ms.

Chapter 7. UDP Server

Reactor Netty provides the easy-to-use and easy-to-configure `UdpServer`. It hides most of the Netty functionality that is required to create a `UDP` server and adds `Reactive Streams` backpressure.

Starting and Stopping

To start a UDP server, a UdpServer instance has to be created and configured. By default, the host is configured to be `localhost` and the port is `12012`. The following example shows how to create and start a UDP server:

github.com/reactor/reactor-netty/tree/main/reactor-netty-examples/src/main/java/reactor/netty/examples/documentation/udp/server/create/Application.java

```java
import reactor.netty.Connection;
import reactor.netty.udp.UdpServer;
import java.time.Duration;

public class Application {

    public static void main(String[] args) {
        Connection server =
                UdpServer.create()                              ①
                         .bindNow(Duration.ofSeconds(30)); ②

        server.onDispose()
              .block();
    }
}
```

① Creates a `UdpServer` instance that is ready for configuring.

② Starts the server in a blocking fashion and waits for it to finish initializing.

The returned `Connection` offers a simple server API, including `disposeNow()`, which shuts the server down in a blocking fashion.

Host and Port

In order to serve on a specific host and port, you can apply the following configuration to the `UDP` server:

github.com/reactor/reactor-netty/tree/main/reactor-netty-examples/src/main/java/reactor/netty/examples/documentation/udp/server/address/Application.java

```java
import reactor.netty.Connection;
import reactor.netty.udp.UdpServer;
import java.time.Duration;

public class Application {

    public static void main(String[] args) {
        Connection server =
                UdpServer.create()
                         .host("localhost") ①
                         .port(8080)        ②
                         .bindNow(Duration.ofSeconds(30));

        server.onDispose()
              .block();
    }
}
```

① Configures the UDP server host

② Configures the UDP server port

> The port can be specified also with **PORT** environment variable.

Eager Initialization

By default, the initialization of the UdpServer resources happens on demand. This means that the bind operation absorbs the extra time needed to initialize and load:

- the event loop group
- the native transport libraries (when native transport is used)

When you need to preload these resources, you can configure the UdpServer as follows:

github.com/reactor/reactor-netty/tree/main/reactor-netty-examples/src/main/java/reactor/netty/examples/documentation/udp/server/warmup/Application.java

```java
import io.netty.channel.socket.DatagramPacket;
import reactor.core.publisher.Mono;
import reactor.netty.Connection;
import reactor.netty.udp.UdpServer;

import java.time.Duration;

public class Application {

    public static void main(String[] args) {
        UdpServer udpServer =
                UdpServer.create()
                         .handle((in, out) ->
                             out.sendObject(
                                 in.receiveObject()
                                   .map(o -> {
                                       if (o instanceof DatagramPacket) {
                                           DatagramPacket p = (DatagramPacket) o;
                                           return new DatagramPacket(p.content
().retain(), p.sender());
                                       }
                                       else {
                                           return Mono.error(new Exception("Unexpected
type of the message: " + o));
                                       }
                                   })));

        udpServer.warmup() ①
                 .block();

        Connection server = udpServer.bindNow(Duration.ofSeconds(30));

        server.onDispose()
              .block();
    }
}
```

① Initialize and load the event loop group and the native transport libraries

Writing Data

To send data to the remote peer, you must attach an I/O handler. The I/O handler has access to `UdpOutbound`, to be able to write data. The following example shows how to send `hello`:

github.com/reactor/reactor-netty/tree/main/reactor-netty-examples/src/main/java/reactor/netty/examples/documentation/udp/server/send/Application.java

```java
import io.netty.buffer.ByteBuf;
import io.netty.buffer.Unpooled;
import io.netty.channel.socket.DatagramPacket;
import io.netty.util.CharsetUtil;
import reactor.core.publisher.Mono;
import reactor.netty.Connection;
import reactor.netty.udp.UdpServer;

import java.time.Duration;

public class Application {

    public static void main(String[] args) {
        Connection server =
                UdpServer.create()
                         .handle((in, out) ->
                             out.sendObject(
                                 in.receiveObject()
                                   .map(o -> {
                                       if (o instanceof DatagramPacket) {
                                           DatagramPacket p = (DatagramPacket) o;
                                           ByteBuf buf = Unpooled.copiedBuffer(
"hello", CharsetUtil.UTF_8);
                                           return new DatagramPacket(buf, p.sender());
①
                                       }
                                       else {
                                           return Mono.error(new Exception("Unexpected
type of the message: " + o));
                                       }
                                   })))
                         .bindNow(Duration.ofSeconds(30));

        server.onDispose()
              .block();
    }
}
```

① Sends a `hello` string to the remote peer

Consuming Data

To receive data from a remote peer, you must attach an I/O handler. The I/O handler has access to `UdpInbound`, to be able to read data. The following example shows how to consume data:

github.com/reactor/reactor-netty/tree/main/reactor-netty-examples/src/main/java/reactor/netty/examples/documentation/udp/server/read/Application.java

```
import io.netty.channel.socket.DatagramPacket;
import reactor.core.publisher.Mono;
import reactor.netty.Connection;
import reactor.netty.udp.UdpServer;

import java.time.Duration;

public class Application {

    public static void main(String[] args) {
        Connection server =
                UdpServer.create()
                         .handle((in, out) ->
                             out.sendObject(
                                 in.receiveObject()
                                   .map(o -> {
                                       if (o instanceof DatagramPacket) {
                                           DatagramPacket p = (DatagramPacket) o;
                                           return new DatagramPacket(p.content
().retain(), p.sender()); ①
                                       }
                                       else {
                                           return Mono.error(new Exception("Unexpected
type of the message: " + o));
                                       }
                                   })))
                         .bindNow(Duration.ofSeconds(30));

        server.onDispose()
              .block();
    }
}
```

① Receives data from the remote peer

Lifecycle Callbacks

The following lifecycle callbacks are provided to let you extend the `UdpServer`:

Callback	Description
`doOnBind`	Invoked when the server channel is about to bind.
`doOnBound`	Invoked when the server channel is bound.
`doOnChannelInit`	Invoked when initializing the channel.
`doOnUnbound`	Invoked when the server channel is unbound.

The following example uses the `doOnBound` and `doOnChannelInit` callbacks:

github.com/reactor/reactor-netty/tree/main/reactor-netty-examples/src/main/java/reactor/netty/examples/documentation/udp/server/lifecycle/Application.java

```
import io.netty.handler.codec.LineBasedFrameDecoder;
import io.netty.handler.logging.LoggingHandler;
import reactor.netty.Connection;
import reactor.netty.udp.UdpServer;
import java.time.Duration;

public class Application {

    public static void main(String[] args) {
        Connection server =
                UdpServer.create()
                         .doOnBound(conn -> conn.addHandlerLast(new
LineBasedFrameDecoder(8192))) ①
                         .doOnChannelInit((observer, channel, remoteAddress) ->
                             channel.pipeline()
                                    .addFirst(new LoggingHandler
("reactor.netty.examples")))      ②
                         .bindNow(Duration.ofSeconds(30));

        server.onDispose()
              .block();
    }
}
```

① Netty pipeline is extended with LineBasedFrameDecoder when the server channel is bound.

② Netty pipeline is extended with LoggingHandler when initializing the channel.

Connection Configuration

This section describes three kinds of configuration that you can use at the UDP level:

- Channel Options
- Wire Logger
- Event Loop Group

Channel Options

By default, the UDP server is configured with the following options:

github.com/reactor/reactor-netty/tree/main/reactor-netty-core/src/main/java/reactor/netty/udp/UdpServerBind.java

```java
    UdpServerBind() {
        this.config = new UdpServerConfig(
                Collections.singletonMap(ChannelOption.AUTO_READ, false),
                () -> new InetSocketAddress(NetUtil.LOCALHOST, DEFAULT_PORT));
    }
```

If you need additional options or need to change the current options, you can apply the following configuration:

github.com/reactor/reactor-netty/tree/main/reactor-netty-examples/src/main/java/reactor/netty/examples/documentation/udp/server/channeloptions/Application.java

```java
import io.netty.channel.ChannelOption;
import reactor.netty.Connection;
import reactor.netty.udp.UdpServer;
import java.time.Duration;

public class Application {

    public static void main(String[] args) {
        Connection server =
                UdpServer.create()
                         .option(ChannelOption.CONNECT_TIMEOUT_MILLIS, 10000)
                         .bindNow(Duration.ofSeconds(30));

        server.onDispose()
              .block();
    }
}
```

For more information about Netty channel options, see the following links:

- Common ChannelOption
- Epoll ChannelOption
- KQueue ChannelOption
- Socket Options

Wire Logger

Reactor Netty provides wire logging for when the traffic between the peers needs to be inspected. By default, wire logging is disabled. To enable it, you must set the logger `reactor.netty.udp.UdpServer` level to `DEBUG` and apply the following configuration:

github.com/reactor/reactor-netty/tree/main/reactor-netty-examples/src/main/java/reactor/netty/examples/documentation/udp/server/wiretap/Application.java

```java
import reactor.netty.Connection;
import reactor.netty.udp.UdpServer;
import java.time.Duration;

public class Application {

    public static void main(String[] args) {
        Connection server =
                UdpServer.create()
                         .wiretap(true) ①
                         .bindNow(Duration.ofSeconds(30));

        server.onDispose()
              .block();
    }
}
```

① Enables the wire logging

Wire Logger formatters

Reactor Netty supports 3 different formatters:

- AdvancedByteBufFormat#HEX_DUMP - the default

github.com/reactor/reactor-netty/tree/main/reactor-netty-core/src/main/java/reactor/netty/transport/logging/AdvancedByteBufFormat.java

```
    /**
     * When wire logging is enabled with this format, both events and content will be
logged.
     * The content will be in hex format.
     * <p>Examples:</p>
     * <pre>
     * {@code
     * reactor.netty.http.HttpTests - [d5230a14, L:/0:0:0:0:0:0:0:1:60267 -
R:/0:0:0:0:0:0:0:1:60269] REGISTERED
     * reactor.netty.http.HttpTests - [d5230a14, L:/0:0:0:0:0:0:0:1:60267 -
R:/0:0:0:0:0:0:0:1:60269] ACTIVE
     * reactor.netty.http.HttpTests - [d5230a14, L:/0:0:0:0:0:0:0:1:60267 -
R:/0:0:0:0:0:0:0:1:60269] READ: 145B
     *          +-------------------------------------------------+
     *          |  0  1  2  3  4  5  6  7  8  9  a  b  c  d  e  f |
     * +--------+-------------------------------------------------+----------------+
     * |00000000| 50 4f 53 54 20 2f 74 65 73 74 2f 57 6f 72 6c 64 |POST /test/World|
     * |00000010| 20 48 54 54 50 2f 31 2e 31 0d 0a 43 6f 6e 74 65 | HTTP/1.1..Conte|
     * |00000020| 6e 74 2d 54 79 70 65 3a 20 74 65 78 74 2f 70 6c |nt-Type: text/pl|
     * |00000030| 61 69 6e 0d 0a 75 73 65 72 2d 61 67 65 6e 74 3a |ain..user-agent:|
     * |00000040| 20 52 65 61 63 74 6f 72 4e 65 74 74 79 2f 64 65 | ReactorNetty/de|
     * ...
     * reactor.netty.http.HttpTests - [d5230a14, L:/0:0:0:0:0:0:0:1:60267 -
R:/0:0:0:0:0:0:0:1:60269] WRITE: 38B
     *          +-------------------------------------------------+
     *          |  0  1  2  3  4  5  6  7  8  9  a  b  c  d  e  f |
     * +--------+-------------------------------------------------+----------------+
     * |00000000| 48 54 54 50 2f 31 2e 31 20 32 30 30 20 4f 4b 0d |HTTP/1.1 200 OK.|
     * |00000010| 0a 63 6f 6e 74 65 6e 74 2d 6c 65 6e 67 74 68 3a |.content-length:|
     * |00000020| 20 30 0d 0a 0d 0a                               | 0....          |
     * +--------+-------------------------------------------------+----------------+
     * }
     * </pre>
     */
```

- AdvancedByteBufFormat#SIMPLE

github.com/reactor/reactor-netty/tree/main/reactor-netty-core/src/main/java/reactor/netty/transport/logging/AdvancedByteBufFormat.java

```
/**
 * When wire logging is enabled with this format, only the events will be logged.
 * <p>Examples:</p>
 * <pre>
 * {@code
 * reactor.netty.http.HttpTests - [230d3686, L:/0:0:0:0:0:0:0:1:60241 -
R:/0:0:0:0:0:0:0:1:60245] REGISTERED
 * reactor.netty.http.HttpTests - [230d3686, L:/0:0:0:0:0:0:0:1:60241 -
R:/0:0:0:0:0:0:0:1:60245] ACTIVE
 * reactor.netty.http.HttpTests - [230d3686, L:/0:0:0:0:0:0:0:1:60241 -
R:/0:0:0:0:0:0:0:1:60245] READ: 145B
 * reactor.netty.http.HttpTests - [230d3686, L:/0:0:0:0:0:0:0:1:60241 -
R:/0:0:0:0:0:0:0:1:60245] WRITE: 38B
 * }
 * </pre>
 */
```

- AdvancedByteBufFormat#TEXTUAL

github.com/reactor/reactor-netty/tree/main/reactor-netty-core/src/main/java/reactor/netty/transport/logging/AdvancedByteBufFormat.java

```
/**
 * When wire logging is enabled with this format, both events and content will be
logged.
 * The content will be in plain text format.
 * <p>Examples:</p>
 * <pre>
 * {@code
 * reactor.netty.http.HttpTests - [02c3db6c, L:/0:0:0:0:0:0:0:1:60317 -
R:/0:0:0:0:0:0:0:1:60319] REGISTERED
 * reactor.netty.http.HttpTests - [02c3db6c, L:/0:0:0:0:0:0:0:1:60317 -
R:/0:0:0:0:0:0:0:1:60319] ACTIVE
 * reactor.netty.http.HttpTests - [02c3db6c, L:/0:0:0:0:0:0:0:1:60317 -
R:/0:0:0:0:0:0:0:1:60319] READ: 145B POST /test/World HTTP/1.1
 * Content-Type: text/plain
 * user-agent: ReactorNetty/dev
 * ...
 * reactor.netty.http.HttpTests - [02c3db6c, L:/0:0:0:0:0:0:0:1:60317 -
R:/0:0:0:0:0:0:0:1:60319] WRITE: 38B HTTP/1.1 200 OK
 * content-length: 0
 * }
 * </pre>
 */
```

When you need to change the default formatter you can configure it as follows:

github.com/reactor/reactor-netty/tree/main/reactor-netty-examples/src/main/java/reactor/netty/

examples/documentation/udp/server/wiretap/custom/Application.java

```
import io.netty.handler.logging.LogLevel;
import reactor.netty.Connection;
import reactor.netty.transport.logging.AdvancedByteBufFormat;
import reactor.netty.udp.UdpServer;

import java.time.Duration;

public class Application {

    public static void main(String[] args) {
        Connection server =
                UdpServer.create()
                         .wiretap("logger-name", LogLevel.DEBUG,
AdvancedByteBufFormat.TEXTUAL) ①
                         .bindNow(Duration.ofSeconds(30));

        server.onDispose()
              .block();
    }
}
```

① Enables the wire logging, AdvancedByteBufFormat#TEXTUAL is used for printing the content.

Event Loop Group

By default `Reactor Netty` uses an “Event Loop Group”, where the number of the worker threads equals the number of processors available to the runtime on initialization (but with a minimum value of 4). When you need a different configuration, you can use one of the `LoopResources#create` methods.

The following listing shows the default configuration for the Event Loop Group:

github.com/reactor/reactor-netty/tree/main/reactor-netty-core/src/main/java/reactor/netty/ReactorNetty.java

```
    /**
     * Default worker thread count, fallback to available processor
     * (but with a minimum value of 4).
     */
    public static final String IO_WORKER_COUNT = "reactor.netty.ioWorkerCount";
    /**
     * Default selector thread count, fallback to -1 (no selector thread)
     * <p><strong>Note:</strong> In most use cases using a worker thread also as a
selector thread works well.
     * A possible use case for specifying a separate selector thread might be when the
worker threads are too busy
     * and connections cannot be accepted fast enough.
     * <p><strong>Note:</strong> Although more than 1 can be configured as a selector
thread count, in reality
     * only 1 thread will be used as a selector thread.
     */
    public static final String IO_SELECT_COUNT = "reactor.netty.ioSelectCount";
    /**
     * Default worker thread count for UDP, fallback to available processor
     * (but with a minimum value of 4).
     */
    public static final String UDP_IO_THREAD_COUNT =
"reactor.netty.udp.ioThreadCount";
    /**
     * Default quiet period that guarantees that the disposal of the underlying
LoopResources
     * will not happen, fallback to 2 seconds.
     */
    public static final String SHUTDOWN_QUIET_PERIOD =
"reactor.netty.ioShutdownQuietPeriod";
    /**
     * Default maximum amount of time to wait until the disposal of the underlying
LoopResources
     * regardless if a task was submitted during the quiet period, fallback to 15
seconds.
     */
    public static final String SHUTDOWN_TIMEOUT = "reactor.netty.ioShutdownTimeout";

    /**
     * Default value whether the native transport (epoll, kqueue) will be preferred,
     * fallback it will be preferred when available.
     */
    public static final String NATIVE = "reactor.netty.native";
```

If you need changes to these settings, you can apply the following configuration:

github.com/reactor/reactor-netty/tree/main/reactor-netty-examples/src/main/java/reactor/netty/examples/documentation/udp/server/eventloop/Application.java

```java
import reactor.netty.Connection;
import reactor.netty.resources.LoopResources;
import reactor.netty.udp.UdpServer;
import java.time.Duration;

public class Application {

    public static void main(String[] args) {
        LoopResources loop = LoopResources.create("event-loop", 1, 4, true);

        Connection server =
                UdpServer.create()
                         .runOn(loop)
                         .bindNow(Duration.ofSeconds(30));

        server.onDispose()
              .block();
    }
}
```

Disposing Event Loop Group

- If you use the default `Event Loop Group` provided by Reactor Netty, invoke `HttpResources#disposeLoopsAndConnections/#disposeLoopsAndConnectionsLater` method.

> Disposing `HttpResources` means that every server/client that is using it, will not be able to use it anymore!

- If you use custom `LoopResources`, invoke `LoopResources#dispose/#disposeLater` method.

> Disposing the custom `LoopResources` means that every server/client that is configured to use it, will not be able to use it anymore!

Metrics

The UDP server supports built-in integration with `Micrometer`. It exposes all metrics with a prefix of `reactor.netty.udp.server`.

The following table provides information for the UDP server metrics:

metric name	type	description
reactor.netty.udp.server.data.received	DistributionSummary	Amount of the data received, in bytes. See Data Received
reactor.netty.udp.server.data.sent	DistributionSummary	Amount of the data sent, in bytes. See Data Sent

metric name	type	description
reactor.netty.udp.server.errors	Counter	Number of errors that occurred. See Errors Count

These additional metrics are also available:

`ByteBufAllocator` metrics

metric name	type	description
reactor.netty.bytebuf.allocator.used.heap.memory	Gauge	The number of bytes reserved by heap buffer allocator. See Used Heap Memory
reactor.netty.bytebuf.allocator.used.direct.memory	Gauge	The number of bytes reserved by direct buffer allocator. See Used Direct Memory
reactor.netty.bytebuf.allocator.heap.arenas	Gauge	The number of heap arenas (when `PooledByteBufAllocator`). See Heap Arenas
reactor.netty.bytebuf.allocator.direct.arenas	Gauge	The number of direct arenas (when `PooledByteBufAllocator`). See Direct Arenas
reactor.netty.bytebuf.allocator.threadlocal.caches	Gauge	The number of thread local caches (when `PooledByteBufAllocator`). See Thread Local Caches
reactor.netty.bytebuf.allocator.small.cache.size	Gauge	The size of the small cache (when `PooledByteBufAllocator`). See Small Cache Size
reactor.netty.bytebuf.allocator.normal.cache.size	Gauge	The size of the normal cache (when `PooledByteBufAllocator`). See Normal Cache Size
reactor.netty.bytebuf.allocator.chunk.size	Gauge	The chunk size for an arena (when `PooledByteBufAllocator`). See Chunk Size
reactor.netty.bytebuf.allocator.active.heap.memory	Gauge	The actual bytes consumed by in-use buffers allocated from heap buffer pools (when `PooledByteBufAllocator`). See Active Heap Memory

metric name	type	description
reactor.netty.bytebuf.allocator.active.direct.memory	Gauge	The actual bytes consumed by in-use buffers allocated from direct buffer pools (when `PooledByteBufAllocator`). See Active Direct Memory

`EventLoop` metrics

metric name	type	description
reactor.netty.eventloop.pending.tasks	Gauge	The number of tasks that are pending for processing on an event loop. See Pending Tasks

The following example enables that integration:

github.com/reactor/reactor-netty/tree/main/reactor-netty-examples/src/main/java/reactor/netty/examples/documentation/udp/server/metrics/Application.java

```
import reactor.netty.Connection;
import reactor.netty.udp.UdpServer;

import java.time.Duration;

public class Application {

    public static void main(String[] args) {
        Connection server =
                UdpServer.create()
                         .metrics(true) ①
                         .bindNow(Duration.ofSeconds(30));

        server.onDispose()
              .block();
    }
}
```

① Enables the built-in integration with Micrometer

When UDP server metrics are needed for an integration with a system other than `Micrometer` or you want to provide your own integration with `Micrometer`, you can provide your own metrics recorder, as follows:

github.com/reactor/reactor-netty/tree/main/reactor-netty-examples/src/main/java/reactor/netty/examples/documentation/udp/server/metrics/custom/Application.java

```java
import reactor.netty.Connection;
import reactor.netty.channel.ChannelMetricsRecorder;
import reactor.netty.udp.UdpServer;

import java.net.SocketAddress;
import java.time.Duration;

public class Application {

    public static void main(String[] args) {
        Connection server =
                UdpServer.create()
                         .metrics(true, CustomChannelMetricsRecorder::new) ①
                         .bindNow(Duration.ofSeconds(30));

        server.onDispose()
              .block();
    }
```

① Enables UDP server metrics and provides `ChannelMetricsRecorder` implementation.

Unix Domain Sockets

The `UdpServer` supports Unix Domain Datagram Sockets (UDS) when native transport is in use.

The following example shows how to use UDS support:

github.com/reactor/reactor-netty/tree/main/reactor-netty-examples/src/main/java/reactor/netty/examples/documentation/udp/server/uds/Application.java

```java
import io.netty.channel.unix.DomainDatagramPacket;
import io.netty.channel.unix.DomainSocketAddress;
import reactor.core.publisher.Mono;
import reactor.netty.Connection;
import reactor.netty.udp.UdpServer;

import java.io.File;

public class Application {

    public static void main(String[] args) {
        Connection server =
                UdpServer.create()
                         .bindAddress(Application::newDomainSocketAddress) ①
                         .handle((in, out) ->
                             out.sendObject(
                                 in.receiveObject()
                                   .map(o -> {
                                       if (o instanceof DomainDatagramPacket) {
                                           DomainDatagramPacket p =
(DomainDatagramPacket) o;
                                           return new DomainDatagramPacket(p.
content().retain(), p.sender());
                                       }
                                       else {
                                           return Mono.error(new Exception("Unexpected
type of the message: " + o));
                                       }
                                   })))
                         .bindNow();

        server.onDispose()
              .block();
    }
```

① Specifies `DomainSocketAddress` that will be used

Chapter 8. UDP Client

Reactor Netty provides the easy-to-use and easy-to-configure `UdpClient`. It hides most of the Netty functionality that is required to create a `UDP` client and adds Reactive Streams backpressure.

Connecting and Disconnecting

To connect the UDP client to a given endpoint, you must create and configure a UdpClient instance. By default, the host is configured for `localhost` and the port is `12012`. The following example shows how to create and connect a UDP client:

github.com/reactor/reactor-netty/tree/main/reactor-netty-examples/src/main/java/reactor/netty/examples/documentation/udp/client/create/Application.java

```
import reactor.netty.Connection;
import reactor.netty.udp.UdpClient;
import java.time.Duration;

public class Application {

    public static void main(String[] args) {
        Connection connection =
                UdpClient.create()                              ①
                         .connectNow(Duration.ofSeconds(30)); ②

        connection.onDispose()
                  .block();
    }
}
```

① Creates a `UdpClient` instance that is ready for configuring.

② Connects the client in a blocking fashion and waits for it to finish initializing.

The returned `Connection` offers a simple connection API, including `disposeNow()`, which shuts the client down in a blocking fashion.

Host and Port

To connect to a specific `host` and `port`, you can apply the following configuration to the `UDP` client:

github.com/reactor/reactor-netty/tree/main/reactor-netty-examples/src/main/java/reactor/netty/examples/documentation/udp/client/address/Application.java

```java
import reactor.netty.Connection;
import reactor.netty.udp.UdpClient;
import java.time.Duration;

public class Application {

    public static void main(String[] args) {
        Connection connection =
                UdpClient.create()
                         .host("example.com") ①
                         .port(80)            ②
                         .connectNow(Duration.ofSeconds(30));

        connection.onDispose()
                  .block();
    }
}
```

① Configures the `host` to which this client should connect

② Configures the `port` to which this client should connect

> The port can be specified also with **PORT** environment variable.

Eager Initialization

By default, the initialization of the `UdpClient` resources happens on demand. This means that the `connect operation` absorbs the extra time needed to initialize and load:

- the event loop group
- the host name resolver
- the native transport libraries (when native transport is used)

When you need to preload these resources, you can configure the `UdpClient` as follows:

github.com/reactor/reactor-netty/tree/main/reactor-netty-examples/src/main/java/reactor/netty/examples/documentation/udp/client/warmup/Application.java

```java
import reactor.core.publisher.Mono;
import reactor.netty.Connection;
import reactor.netty.udp.UdpClient;

import java.time.Duration;

public class Application {

    public static void main(String[] args) {
        UdpClient udpClient = UdpClient.create()
                                       .host("example.com")
                                       .port(80)
                                       .handle((udpInbound, udpOutbound) ->
udpOutbound.sendString(Mono.just("hello")));

        udpClient.warmup() ①
                 .block();

        Connection connection = udpClient.connectNow(Duration.ofSeconds(30)); ②

        connection.onDispose()
                  .block();
    }
}
```

① Initialize and load the event loop group, the host name resolver, and the native transport libraries

② Host name resolution happens when connecting to the remote peer

Writing Data

To send data to a given peer, you must attach an I/O handler. The I/O handler has access to `UdpOutbound`, to be able to write data.

The following example shows how to send `hello`:

github.com/reactor/reactor-netty/tree/main/reactor-netty-examples/src/main/java/reactor/netty/examples/documentation/udp/client/send/Application.java

```java
import reactor.core.publisher.Mono;
import reactor.netty.Connection;
import reactor.netty.udp.UdpClient;

import java.time.Duration;

public class Application {

    public static void main(String[] args) {
        Connection connection =
                UdpClient.create()
                         .host("example.com")
                         .port(80)
                         .handle((udpInbound, udpOutbound) -> udpOutbound.sendString
(Mono.just("hello"))) ①
                         .connectNow(Duration.ofSeconds(30));

        connection.onDispose()
                  .block();
    }
}
```

① Sends `hello` string to the remote peer.

Consuming Data

To receive data from a given peer, you must attach an I/O handler. The I/O handler has access to `UdpInbound`, to be able to read data. The following example shows how to consume data:

github.com/reactor/reactor-netty/tree/main/reactor-netty-examples/src/main/java/reactor/netty/examples/documentation/udp/client/read/Application.java

```java
import reactor.netty.Connection;
import reactor.netty.udp.UdpClient;
import java.time.Duration;

public class Application {

    public static void main(String[] args) {
        Connection connection =
                UdpClient.create()
                         .host("example.com")
                         .port(80)
                         .handle((udpInbound, udpOutbound) -> udpInbound.receive
().then()) ①
                         .connectNow(Duration.ofSeconds(30));

        connection.onDispose()
                  .block();
    }
}
```

① Receives data from a given peer

Lifecycle Callbacks

The following lifecycle callbacks are provided to let you extend the `UdpClient`:

Callback	Description
`doAfterResolve`	Invoked after the remote address has been resolved successfully.
`doOnChannelInit`	Invoked when initializing the channel.
`doOnConnect`	Invoked when the channel is about to connect.
`doOnConnected`	Invoked after the channel has been connected.
`doOnDisconnected`	Invoked after the channel has been disconnected.
`doOnResolve`	Invoked when the remote address is about to be resolved.
`doOnResolveError`	Invoked in case the remote address hasn't been resolved successfully.

The following example uses the `doOnConnected` and `doOnChannelInit` callbacks:

github.com/reactor/reactor-netty/tree/main/reactor-netty-examples/src/main/java/reactor/netty/examples/documentation/udp/client/lifecycle/Application.java

```java
import io.netty.handler.codec.LineBasedFrameDecoder;
import io.netty.handler.logging.LoggingHandler;
import reactor.netty.Connection;
import reactor.netty.udp.UdpClient;
import java.time.Duration;

public class Application {

    public static void main(String[] args) {
        Connection connection =
                UdpClient.create()
                         .host("example.com")
                         .port(80)
                         .doOnConnected(conn -> conn.addHandlerLast(new
LineBasedFrameDecoder(8192))) ①
                         .doOnChannelInit((observer, channel, remoteAddress) ->
                             channel.pipeline()
                                    .addFirst(new LoggingHandler
("reactor.netty.examples")))        ②
                         .connectNow(Duration.ofSeconds(30));

        connection.onDispose()
                  .block();
    }
}
```

① `Netty` pipeline is extended with `LineBasedFrameDecoder` when the channel has been connected.

② `Netty` pipeline is extended with `LoggingHandler` when initializing the channel.

Connection Configuration

This section describes three kinds of configuration that you can use at the UDP level:

- Channel Options
- Wire Logger
- Event Loop Group

Channel Options

By default, the `UDP` client is configured with the following options:

github.com/reactor/reactor-netty/tree/main/reactor-netty-core/src/main/java/reactor/netty/udp/UdpClientConnect.java

```java
    UdpClientConnect() {
        this.config = new UdpClientConfig(
                ConnectionProvider.newConnection(),
                Collections.singletonMap(ChannelOption.AUTO_READ, false),
                () -> new InetSocketAddress(NetUtil.LOCALHOST, DEFAULT_PORT));
    }
```

If you need additional options or need to change the current options, you can apply the following configuration:

github.com/reactor/reactor-netty/tree/main/reactor-netty-examples/src/main/java/reactor/netty/examples/documentation/udp/client/channeloptions/Application.java

```java
import io.netty.channel.ChannelOption;
import reactor.netty.Connection;
import reactor.netty.udp.UdpClient;
import java.time.Duration;

public class Application {

    public static void main(String[] args) {
        Connection connection =
                UdpClient.create()
                         .host("example.com")
                         .port(80)
                         .option(ChannelOption.CONNECT_TIMEOUT_MILLIS, 10000)
                         .connectNow(Duration.ofSeconds(30));

        connection.onDispose()
                  .block();
    }
}
```

You can find more about Netty channel options at the following links:

- Common ChannelOption
- Epoll ChannelOption
- KQueue ChannelOption
- Socket Options

Wire Logger

Reactor Netty provides wire logging for when the traffic between the peers needs to be inspected. By default, wire logging is disabled. To enable it, you must set the logger `reactor.netty.udp.UdpClient` level to `DEBUG` and apply the following configuration:

github.com/reactor/reactor-netty/tree/main/reactor-netty-examples/src/main/java/reactor/netty/

examples/documentation/udp/client/wiretap/Application.java

```
import reactor.netty.Connection;
import reactor.netty.udp.UdpClient;
import java.time.Duration;

public class Application {

    public static void main(String[] args) {
        Connection connection =
                UdpClient.create()
                         .host("example.com")
                         .port(80)
                         .wiretap(true) ①
                         .connectNow(Duration.ofSeconds(30));

        connection.onDispose()
                  .block();
    }
}
```

① Enables the wire logging

Wire Logger formatters

Reactor Netty supports 3 different formatters:

- AdvancedByteBufFormat#HEX_DUMP - the default

github.com/reactor/reactor-netty/tree/main/reactor-netty-core/src/main/java/reactor/netty/transport/logging/AdvancedByteBufFormat.java

```
    /**
     * When wire logging is enabled with this format, both events and content will be
logged.
     * The content will be in hex format.
     * <p>Examples:</p>
     * <pre>
     * {@code
     * reactor.netty.http.HttpTests - [d5230a14, L:/0:0:0:0:0:0:0:1:60267 -
R:/0:0:0:0:0:0:0:1:60269] REGISTERED
     * reactor.netty.http.HttpTests - [d5230a14, L:/0:0:0:0:0:0:0:1:60267 -
R:/0:0:0:0:0:0:0:1:60269] ACTIVE
     * reactor.netty.http.HttpTests - [d5230a14, L:/0:0:0:0:0:0:0:1:60267 -
R:/0:0:0:0:0:0:0:1:60269] READ: 145B
     *          +-------------------------------------------------+
     *          |  0  1  2  3  4  5  6  7  8  9  a  b  c  d  e  f |
     * +--------+-------------------------------------------------+----------------+
     * |00000000| 50 4f 53 54 20 2f 74 65 73 74 2f 57 6f 72 6c 64 |POST /test/World|
     * |00000010| 20 48 54 54 50 2f 31 2e 31 0d 0a 43 6f 6e 74 65 | HTTP/1.1..Conte|
     * |00000020| 6e 74 2d 54 79 70 65 3a 20 74 65 78 74 2f 70 6c |nt-Type: text/pl|
     * |00000030| 61 69 6e 0d 0a 75 73 65 72 2d 61 67 65 6e 74 3a |ain..user-agent:|
     * |00000040| 20 52 65 61 63 74 6f 72 4e 65 74 74 79 2f 64 65 | ReactorNetty/de|
     * ...
     * reactor.netty.http.HttpTests - [d5230a14, L:/0:0:0:0:0:0:0:1:60267 -
R:/0:0:0:0:0:0:0:1:60269] WRITE: 38B
     *          +-------------------------------------------------+
     *          |  0  1  2  3  4  5  6  7  8  9  a  b  c  d  e  f |
     * +--------+-------------------------------------------------+----------------+
     * |00000000| 48 54 54 50 2f 31 2e 31 20 32 30 30 20 4f 4b 0d |HTTP/1.1 200 OK.|
     * |00000010| 0a 63 6f 6e 74 65 6e 74 2d 6c 65 6e 67 74 68 3a |.content-length:|
     * |00000020| 20 30 0d 0a 0d 0a                               | 0....          |
     * +--------+-------------------------------------------------+----------------+
     * }
     * </pre>
     */
```

- AdvancedByteBufFormat#SIMPLE

github.com/reactor/reactor-netty/tree/main/reactor-netty-core/src/main/java/reactor/netty/transport/logging/AdvancedByteBufFormat.java

```
/**
 * When wire logging is enabled with this format, only the events will be logged.
 * <p>Examples:</p>
 * <pre>
 * {@code
 * reactor.netty.http.HttpTests - [230d3686, L:/0:0:0:0:0:0:0:1:60241 -
R:/0:0:0:0:0:0:0:1:60245] REGISTERED
 * reactor.netty.http.HttpTests - [230d3686, L:/0:0:0:0:0:0:0:1:60241 -
R:/0:0:0:0:0:0:0:1:60245] ACTIVE
 * reactor.netty.http.HttpTests - [230d3686, L:/0:0:0:0:0:0:0:1:60241 -
R:/0:0:0:0:0:0:0:1:60245] READ: 145B
 * reactor.netty.http.HttpTests - [230d3686, L:/0:0:0:0:0:0:0:1:60241 -
R:/0:0:0:0:0:0:0:1:60245] WRITE: 38B
 * }
 * </pre>
 */
```

- AdvancedByteBufFormat#TEXTUAL

github.com/reactor/reactor-netty/tree/main/reactor-netty-core/src/main/java/reactor/netty/transport/logging/AdvancedByteBufFormat.java

```
/**
 * When wire logging is enabled with this format, both events and content will be
logged.
 * The content will be in plain text format.
 * <p>Examples:</p>
 * <pre>
 * {@code
 * reactor.netty.http.HttpTests - [02c3db6c, L:/0:0:0:0:0:0:0:1:60317 -
R:/0:0:0:0:0:0:0:1:60319] REGISTERED
 * reactor.netty.http.HttpTests - [02c3db6c, L:/0:0:0:0:0:0:0:1:60317 -
R:/0:0:0:0:0:0:0:1:60319] ACTIVE
 * reactor.netty.http.HttpTests - [02c3db6c, L:/0:0:0:0:0:0:0:1:60317 -
R:/0:0:0:0:0:0:0:1:60319] READ: 145B POST /test/World HTTP/1.1
 * Content-Type: text/plain
 * user-agent: ReactorNetty/dev
 * ...
 * reactor.netty.http.HttpTests - [02c3db6c, L:/0:0:0:0:0:0:0:1:60317 -
R:/0:0:0:0:0:0:0:1:60319] WRITE: 38B HTTP/1.1 200 OK
 * content-length: 0
 * }
 * </pre>
 */
```

When you need to change the default formatter you can configure it as follows:

github.com/reactor/reactor-netty/tree/main/reactor-netty-examples/src/main/java/reactor/netty/

examples/documentation/udp/client/wiretap/custom/Application.java

```
import io.netty.handler.logging.LogLevel;
import reactor.netty.Connection;
import reactor.netty.transport.logging.AdvancedByteBufFormat;
import reactor.netty.udp.UdpClient;

import java.time.Duration;

public class Application {

    public static void main(String[] args) {
        Connection connection =
                UdpClient.create()
                         .host("example.com")
                         .port(80)
                         .wiretap("logger-name", LogLevel.DEBUG,
AdvancedByteBufFormat.TEXTUAL) ①
                         .connectNow(Duration.ofSeconds(30));

        connection.onDispose()
                  .block();
    }
}
```

① Enables the wire logging, AdvancedByteBufFormat#TEXTUAL is used for printing the content.

Event Loop Group

By default `Reactor Netty` uses an “Event Loop Group”, where the number of the worker threads equals the number of processors available to the runtime on initialization (but with a minimum value of 4). When you need a different configuration, you can use one of the `LoopResources#create` methods.

The following listing shows the default configuration for the Event Loop Group:

github.com/reactor/reactor-netty/tree/main/reactor-netty-core/src/main/java/reactor/netty/ReactorNetty.java

```
    /**
     * Default worker thread count, fallback to available processor
     * (but with a minimum value of 4).
     */
    public static final String IO_WORKER_COUNT = "reactor.netty.ioWorkerCount";
    /**
     * Default selector thread count, fallback to -1 (no selector thread)
     * <p><strong>Note:</strong> In most use cases using a worker thread also as a
selector thread works well.
     * A possible use case for specifying a separate selector thread might be when the
worker threads are too busy
     * and connections cannot be accepted fast enough.
     * <p><strong>Note:</strong> Although more than 1 can be configured as a selector
thread count, in reality
     * only 1 thread will be used as a selector thread.
     */
    public static final String IO_SELECT_COUNT = "reactor.netty.ioSelectCount";
    /**
     * Default worker thread count for UDP, fallback to available processor
     * (but with a minimum value of 4).
     */
    public static final String UDP_IO_THREAD_COUNT =
"reactor.netty.udp.ioThreadCount";
    /**
     * Default quiet period that guarantees that the disposal of the underlying
LoopResources
     * will not happen, fallback to 2 seconds.
     */
    public static final String SHUTDOWN_QUIET_PERIOD =
"reactor.netty.ioShutdownQuietPeriod";
    /**
     * Default maximum amount of time to wait until the disposal of the underlying
LoopResources
     * regardless if a task was submitted during the quiet period, fallback to 15
seconds.
     */
    public static final String SHUTDOWN_TIMEOUT = "reactor.netty.ioShutdownTimeout";

    /**
     * Default value whether the native transport (epoll, kqueue) will be preferred,
     * fallback it will be preferred when available.
     */
    public static final String NATIVE = "reactor.netty.native";
```

If you need changes to these settings, you can apply the following configuration:

github.com/reactor/reactor-netty/tree/main/reactor-netty-examples/src/main/java/reactor/netty/examples/documentation/udp/client/eventloop/Application.java

```java
import reactor.netty.Connection;
import reactor.netty.resources.LoopResources;
import reactor.netty.udp.UdpClient;
import java.time.Duration;

public class Application {

    public static void main(String[] args) {
        LoopResources loop = LoopResources.create("event-loop", 1, 4, true);

        Connection connection =
                UdpClient.create()
                         .host("example.com")
                         .port(80)
                         .runOn(loop)
                         .connectNow(Duration.ofSeconds(30));

        connection.onDispose()
                  .block();
    }
}
```

Disposing Event Loop Group

- If you use the default `Event Loop Group` provided by Reactor Netty, invoke `HttpResources#disposeLoopsAndConnections/#disposeLoopsAndConnectionsLater` method.

> Disposing `HttpResources` means that every server/client that is using it, will not be able to use it anymore!

- If you use custom `LoopResources`, invoke `LoopResources#dispose/#disposeLater` method.

> Disposing the custom `LoopResources` means that every server/client that is configured to use it, will not be able to use it anymore!

Metrics

The UDP client supports built-in integration with `Micrometer`. It exposes all metrics with a prefix of `reactor.netty.udp.client`.

The following table provides information for the UDP client metrics:

metric name	type	description
reactor.netty.udp.client.data.received	DistributionSummary	Amount of the data received, in bytes. See Data Received
reactor.netty.udp.client.data.sent	DistributionSummary	Amount of the data sent, in bytes. See Data Sent

metric name	type	description
reactor.netty.udp.client.errors	Counter	Number of errors that occurred. See Errors Count
reactor.netty.udp.client.connect.time	Timer	Time spent for connecting to the remote address. See Connect Time
reactor.netty.udp.client.address.resolver	Timer	Time spent for resolving the address. See Hostname Resolution Time

These additional metrics are also available:

`ByteBufAllocator` metrics

metric name	type	description
reactor.netty.bytebuf.allocator.used.heap.memory	Gauge	The number of bytes reserved by heap buffer allocator. See Used Heap Memory
reactor.netty.bytebuf.allocator.used.direct.memory	Gauge	The number of bytes reserved by direct buffer allocator. See Used Direct Memory
reactor.netty.bytebuf.allocator.heap.arenas	Gauge	The number of heap arenas (when `PooledByteBufAllocator`). See Heap Arenas
reactor.netty.bytebuf.allocator.direct.arenas	Gauge	The number of direct arenas (when `PooledByteBufAllocator`). See Direct Arenas
reactor.netty.bytebuf.allocator.threadlocal.caches	Gauge	The number of thread local caches (when `PooledByteBufAllocator`). See Thread Local Caches
reactor.netty.bytebuf.allocator.small.cache.size	Gauge	The size of the small cache (when `PooledByteBufAllocator`). See Small Cache Size
reactor.netty.bytebuf.allocator.normal.cache.size	Gauge	The size of the normal cache (when `PooledByteBufAllocator`). See Normal Cache Size
reactor.netty.bytebuf.allocator.chunk.size	Gauge	The chunk size for an arena (when `PooledByteBufAllocator`). See Chunk Size

metric name	type	description
reactor.netty.bytebuf.allocator.active.heap.memory	Gauge	The actual bytes consumed by in-use buffers allocated from heap buffer pools (when `PooledByteBufAllocator`). See Active Heap Memory
reactor.netty.bytebuf.allocator.active.direct.memory	Gauge	The actual bytes consumed by in-use buffers allocated from direct buffer pools (when `PooledByteBufAllocator`). See Active Direct Memory

`EventLoop` metrics

metric name	type	description
reactor.netty.eventloop.pending.tasks	Gauge	The number of tasks that are pending for processing on an event loop. See Pending Tasks

The following example enables that integration:

github.com/reactor/reactor-netty/tree/main/reactor-netty-examples/src/main/java/reactor/netty/examples/documentation/udp/client/metrics/Application.java

```
import reactor.netty.Connection;
import reactor.netty.udp.UdpClient;

import java.time.Duration;

public class Application {

    public static void main(String[] args) {
        Connection connection =
                UdpClient.create()
                         .host("example.com")
                         .port(80)
                         .metrics(true) ①
                         .connectNow(Duration.ofSeconds(30));

        connection.onDispose()
                  .block();
    }
}
```

① Enables the built-in integration with Micrometer

When UDP client metrics are needed for an integration with a system other than `Micrometer` or you want to provide your own integration with `Micrometer`, you can provide your own metrics recorder, as follows:

github.com/reactor/reactor-netty/tree/main/reactor-netty-examples/src/main/java/reactor/netty/examples/documentation/udp/client/metrics/custom/Application.java

```java
import reactor.netty.Connection;
import reactor.netty.channel.ChannelMetricsRecorder;
import reactor.netty.udp.UdpClient;

import java.net.SocketAddress;
import java.time.Duration;

public class Application {

    public static void main(String[] args) {
        Connection connection =
                UdpClient.create()
                         .host("example.com")
                         .port(80)
                         .metrics(true, CustomChannelMetricsRecorder::new) ①
                         .connectNow(Duration.ofSeconds(30));

        connection.onDispose()
                  .block();
    }
```

① Enables UDP client metrics and provides `ChannelMetricsRecorder` implementation.

Unix Domain Sockets

The `UdpClient` supports Unix Domain Datagram Sockets (UDS) when native transport is in use.

The following example shows how to use UDS support:

github.com/reactor/reactor-netty/tree/main/reactor-netty-examples/src/main/java/reactor/netty/examples/documentation/udp/client/uds/Application.java

```java
import io.netty.channel.unix.DomainSocketAddress;
import reactor.core.publisher.Mono;
import reactor.netty.Connection;
import reactor.netty.udp.UdpClient;

import java.io.File;

public class Application {

    public static void main(String[] args) {
        Connection connection =
                UdpClient.create()
                         .bindAddress(Application::newDomainSocketAddress)
                         .remoteAddress(() -> new DomainSocketAddress("/tmp/test-
server.sock")) ①
                         .handle((in, out) ->
                             out.sendString(Mono.just("hello"))
                                .then(in.receive()
                                        .asString()
                                        .doOnNext(System.out::println)
                                        .then()))
                         .connectNow();

        connection.onDispose()
                  .block();
    }
```

① Specifies `DomainSocketAddress` that will be used

Chapter 9. Appendices

Frequently Asked Questions

Connection to the proxy cannot be established

Netty's HTTP proxy support always uses `CONNECT` method in order to establish a tunnel to the specified proxy regardless of the scheme that is used `http` or `https`. (More information: Netty enforce HTTP proxy to support HTTP CONNECT method). Some proxies might not support `CONNECT` method when the scheme is `http` or might need to be configured in order to support this way of communication. Sometimes this might be the reason for not being able to connect to the proxy. Consider checking the proxy documentation whether it supports or needs an additional configuration in order to support `CONNECT` method.

What is the meaning of the information that is prepended to every log record?

Reactor Netty adds information for the connection at the beginning of every log record (when this is possible). There is a slight difference in the details for the connection when you use `TCP`, `UDP`, `HTTP/1.1` or `HTTP/2`.

TCP and UDP

In case of `TCP` and `UDP`, the following is added at the beginning of every log record: the id of the underlying connection, local and remote addresses.

```
Examples
[a1566d55, L:/[0:0:0:0:0:0:0:1]:53446 - R:/[0:0:0:0:0:0:0:1]:53444]
[a1566d55, L:/[0:0:0:0:0:0:0:1]:53446 ! R:/[0:0:0:0:0:0:0:1]:53444]

Format
[<CONNECTION_ID>, L:<LOCAL_ADDRESS> <CONNECTION_OPENED_CLOSED> R:<REMOTE_ADDRESS>]
<CONNECTION_ID>: a1566d55
<LOCAL_ADDRESS>: [0:0:0:0:0:0:0:1]:53446
<CONNECTION_OPENED_CLOSED>: - (connection opened)
                            ! (connection closed)
<REMOTE_ADDRESS>: [0:0:0:0:0:0:0:1]:53444
```

HTTP/1.1

In case of `HTTP/1.1`, the following is added at the beginning of every log record: the id of the underlying connection, the serial number of the request received on that connection, local and remote addresses.

```
Examples
[a1566d55-5, L:/[0:0:0:0:0:0:0:1]:53446 - R:/[0:0:0:0:0:0:0:1]:53444]
```

```
[a1566d55-5, L:/[0:0:0:0:0:0:0:1]:53446 ! R:/[0:0:0:0:0:0:0:1]:53444]

Format
[<CONNECTION_ID>-<REQUEST_NUMBER>, L:<LOCAL_ADDRESS> <CONNECTION_OPENED_CLOSED>
R:<REMOTE_ADDRESS>]
<CONNECTION_ID>: a1566d55
<REQUEST_NUMBER>: 5
<LOCAL_ADDRESS>: [0:0:0:0:0:0:0:1]:53446
<CONNECTION_OPENED_CLOSED>: - (connection opened)
                            ! (connection closed)
<REMOTE_ADDRESS>: [0:0:0:0:0:0:0:1]:53444
```

HTTP/2

In case of `HTTP/2`, the following is added at the beginning of every log record: the id of the underlying connection, local and remote addresses, the id of the stream received on that connection.

```
Examples
[a1566d55, L:/[0:0:0:0:0:0:0:1]:53446 - R:/[0:0:0:0:0:0:0:1]:53444](H2 - 5)
[a1566d55, L:/[0:0:0:0:0:0:0:1]:53446 ! R:/[0:0:0:0:0:0:0:1]:53444](H2 - 5)

Format
[<CONNECTION_ID>, L:<LOCAL_ADDRESS> <CONNECTION_OPENED_CLOSED>
R:<REMOTE_ADDRESS>]<STREAM_ID>
<CONNECTION_ID>: a1566d55
<LOCAL_ADDRESS>: [0:0:0:0:0:0:0:1]:53446
<CONNECTION_OPENED_CLOSED>: - (connection opened)
                            ! (connection closed)
<REMOTE_ADDRESS>: [0:0:0:0:0:0:0:1]:53444
<STREAM_ID>: (H2 - 5)
```

How can I extract all log records for a particular HTTP request?

Reactor Netty adds information for the connection at the beginning of every log record (when this is possible). Use the id of the connection in order to extract all log records for a particular HTTP request. For more information see What is the meaning of the information that is prepended to every log record?

How can I debug a memory leak?

By default, Reactor Netty uses direct memory as this is more performant when there are many native I/O operations (working with sockets), as this can remove the copying operations. As allocation and deallocation are expensive operations, Reactor Netty also uses pooled buffers by default. For more information, see Reference Counted Objects.

To be able to debug memory issues with the direct memory and the pooled buffers, Netty provides a special memory leak detection mechanism. Follow the instructions for Troubleshooting Buffer

Leaks to enable this mechanism. In addition to the instructions provided by Netty, Reactor Netty provides a special logger (`_reactor.netty.channel.LeakDetection`) that helps to identify where the memory leak might be located inside Reactor Netty or whether Reactor Netty already forwarded the ownership of the buffers to the application/framework. By default, this logger is disabled. To enable it, increase the log level to `DEBUG`.

Another way to detect memory leaks is to monitor `reactor.netty.bytebuf.allocator.active.heap.memory` and `reactor.netty.bytebuf.allocator.active.direct.memory` meters:

- The `reactor.netty.bytebuf.allocator.active.heap.memory` provides the actual bytes consumed by in-use buffers allocated from heap buffer pools
- The `reactor.netty.bytebuf.allocator.active.direct.memory` provides the actual bytes consumed by in-use buffers allocated from direct buffer pools

If the above meters are constantly growing, then it's likely that there is a buffer memory leak.

> Consider reducing the used memory when debugging memory leak issues (e.g `-XX:MaxDirectMemorySize`, `-Xms`, `-Xmx`). The less memory the application has, the sooner the memory leak will happen.

How can I debug "Connection prematurely closed BEFORE response"?

By default, Reactor Netty clients use connection pooling. When a connection is acquired from the connection pool, it is checked to see whether it is still open. However, the connection can be closed at any time after the acquisition. There are many reasons that can cause a connection to be closed. In most cases, the client might not send directly to the server. Instead, there might be other network components (proxies, load balancers, and so on) between them.

If, on the client side, you observe `Connection prematurely closed BEFORE response`, perform the following checks to identify the reason for the connection being closed:

- Obtain a TCP dump and check which peer sends a FIN/RST signal.
- Check your network connection.
- Check your Firewall and VPN.
- Check for any proxies and load balancers.
  - Do they have some kind of idle timeout configuration (the connection is closed when there is no incoming data for a certain period of time)?
  - Do they silently drop the idle connections without sending any signal? In order to verify whether this might be the issue, you can enable the TCP keep-alive as described in the section Connection Timeout. Issues related to TCP keep-alive configuration on various load balancers were reported in the past.
    - github.com/reactor/reactor-netty/issues/764#issuecomment-1011373248
    - github.com/reactor/reactor-netty/issues/1510
    - github.com/reactor/reactor-netty/issues/1843

- Check the target server.
  - Are there configurations related to any of the following?
    - idle timeout (the connection is closed when there is no incoming data for a certain period of time)
    - limit for buffering data in memory
    - multipart exceeds the max file size limit
    - bad request
    - max keep alive requests (the connection is closed when the requests reach the configured maximum number)
    - rate limit configuration
  - Is the target server in a shutting down state?

Consider checking Timeout Configuration. The section describes various timeout configuration options that are available for Reactor Netty clients. Configuring a proper timeout may improve or solve issues in the communication process.

Observability

Observability metadata

Observability - Metrics

Below you can find a list of all metrics declared by this project.

Active Connections

> The number of the connections in the connection pool that have been successfully acquired and are in active use.

Metric name `reactor.netty.connection.provider.active.connections`. **Type** `gauge`.

> KeyValues that are added after starting the Observation might be missing from the *.active metrics.

Fully qualified name of the enclosing class `reactor.netty.http.client.Http2ConnectionProviderMeters`.

Table 1. Low cardinality Keys

Name	Description
`id` *(required)*	ID.
`name` *(required)*	NAME.
`remote.address` *(required)*	Remote address.

Active Connections

> The number of the connections in the connection pool that have been successfully acquired and are in active use.

Metric name `reactor.netty.connection.provider.active.connections`. **Type** `gauge`.

> [!IMPORTANT]
> KeyValues that are added after starting the Observation might be missing from the *.active metrics.

Fully qualified name of the enclosing class `reactor.netty.resources.ConnectionProviderMeters`.

Table 2. Low cardinality Keys

Name	Description
`id` *(required)*	ID.
`name` *(required)*	NAME.
`remote.address` *(required)*	Remote address.

Active Direct Memory

> The actual bytes consumed by in-use buffers allocated from direct buffer pools.

Metric name `reactor.netty.bytebuf.allocator.active.direct.memory`. **Type** `gauge`.

> [!IMPORTANT]
> KeyValues that are added after starting the Observation might be missing from the *.active metrics.

Fully qualified name of the enclosing class `reactor.netty.transport.ByteBufAllocatorMeters`.

Table 3. Low cardinality Keys

Name	Description
`id` *(required)*	ID.
`type` *(required)*	TYPE.

Active Heap Memory

> The actual bytes consumed by in-use buffers allocated from heap buffer pools.

Metric name `reactor.netty.bytebuf.allocator.active.heap.memory`. **Type** `gauge`.

> [!IMPORTANT]
> KeyValues that are added after starting the Observation might be missing from the *.active metrics.

Fully qualified name of the enclosing class `reactor.netty.transport.ByteBufAllocatorMeters`.

Table 4. Low cardinality Keys

Name	Description
`id` *(required)*	ID.
`type` *(required)*	TYPE.

Active Streams

> The number of the active HTTP/2 streams.

Metric name `reactor.netty.connection.provider.active.streams`. **Type** `gauge`.

> KeyValues that are added after starting the Observation might be missing from the *.active metrics.

Fully qualified name of the enclosing class `reactor.netty.http.client.Http2ConnectionProviderMeters`.

Table 5. Low cardinality Keys

Name	Description
`id` *(required)*	ID.
`name` *(required)*	NAME.
`remote.address` *(required)*	Remote address.

Chunk Size

> The chunk size for an arena.

Metric name `reactor.netty.bytebuf.allocator.chunk.size`. **Type** `gauge`.

> KeyValues that are added after starting the Observation might be missing from the *.active metrics.

Fully qualified name of the enclosing class `reactor.netty.transport.ByteBufAllocatorMeters`.

Table 6. Low cardinality Keys

Name	Description
`id` *(required)*	ID.
`type` *(required)*	TYPE.

Connections Active

> The number of http connections, on the server, currently processing

> requests.

Metric name `reactor.netty.http.server.connections.active`. **Type** `gauge`.

> **!** KeyValues that are added after starting the Observation might be missing from the *.active metrics.

Fully qualified name of the enclosing class `reactor.netty.http.server.HttpServerMeters`.

Table 7. Low cardinality Keys

Name	Description
`local.address` *(required)*	Local address.
`uri` *(required)*	URI.

Connections Total

> The number of all opened connections on the server.

Metric name `%s` - since it contains `%s`, the name is dynamic and will be resolved at runtime. **Type** `gauge`.

> **!** KeyValues that are added after starting the Observation might be missing from the *.active metrics.

Fully qualified name of the enclosing class `reactor.netty.channel.ChannelMeters`.

Table 8. Low cardinality Keys

Name	Description
`local.address` *(required)*	Local address.
`uri` *(required)*	URI.

Connect Time

> Connect metric.

Metric name `%s` - since it contains `%s`, the name is dynamic and will be resolved at runtime. **Type** `timer`.

Metric name `%s.active` - since it contains `%s`, the name is dynamic and will be resolved at runtime. **Type** `long task timer`.

> **!** KeyValues that are added after starting the Observation might be missing from the *.active metrics.

> **!** Micrometer internally uses `nanoseconds` for the baseunit. However, each backend

determines the actual baseunit. (i.e. Prometheus uses seconds)

Fully qualified name of the enclosing class `reactor.netty.channel.ConnectObservations`.

Table 9. Low cardinality Keys

Name	Description
`proxy.address` *(required)*	Proxy address, when there is a proxy configured.
`remote.address` *(required)*	Remote address.
`status` *(required)*	STATUS.

Table 10. High cardinality Keys

Name	Description
`net.peer.name` *(required)*	Net peer name.
`net.peer.port` *(required)*	Net peer port.
`reactor.netty.protocol` *(required)*	Reactor Netty protocol (tcp/http etc.).
`reactor.netty.status` *(required)*	Reactor Netty status.
`reactor.netty.type` *(required)*	Reactor Netty type (always client).

Data Received

> Amount of the data received, in bytes.

Metric name `%s` - since it contains `%s`, the name is dynamic and will be resolved at runtime. **Type** `distribution summary` and **base unit** `bytes`.

> KeyValues that are added after starting the Observation might be missing from the *.active metrics.

Fully qualified name of the enclosing class `reactor.netty.channel.ChannelMeters`.

Table 11. Low cardinality Keys

Name	Description
`proxy.address` *(required)*	Proxy address, when there is a proxy configured.
`remote.address` *(required)*	Remote address.
`uri` *(required)*	URI.

Data Sent

> Amount of the data sent, in bytes.

Metric name `%s` - since it contains `%s`, the name is dynamic and will be resolved at runtime. **Type** `distribution summary` and **base unit** `bytes`.

> KeyValues that are added after starting the Observation might be missing from the *.active metrics.

Fully qualified name of the enclosing class `reactor.netty.channel.ChannelMeters`.

Table 12. Low cardinality Keys

Name	Description
`proxy.address` *(required)*	Proxy address, when there is a proxy configured.
`remote.address` *(required)*	Remote address.
`uri` *(required)*	URI.

Direct Arenas

> The number of direct arenas.

Metric name `reactor.netty.bytebuf.allocator.direct.arenas`. **Type** `gauge`.

> KeyValues that are added after starting the Observation might be missing from the *.active metrics.

Fully qualified name of the enclosing class `reactor.netty.transport.ByteBufAllocatorMeters`.

Table 13. Low cardinality Keys

Name	Description
`id` *(required)*	ID.
`type` *(required)*	TYPE.

Errors Count

> Number of errors that occurred.

Metric name `%s` - since it contains `%s`, the name is dynamic and will be resolved at runtime. **Type** `counter`.

> KeyValues that are added after starting the Observation might be missing from the *.active metrics.

Fully qualified name of the enclosing class `reactor.netty.channel.ChannelMeters`.

Table 14. Low cardinality Keys

Name	Description

`proxy.address` *(required)*	Proxy address, when there is a proxy configured.
`remote.address` *(required)*	Remote address.
`uri` *(required)*	URI.

Heap Arenas

> The number of heap arenas.

Metric name `reactor.netty.bytebuf.allocator.heap.arenas`. **Type** `gauge`.

> ❗ KeyValues that are added after starting the Observation might be missing from the *.active metrics.

Fully qualified name of the enclosing class `reactor.netty.transport.ByteBufAllocatorMeters`.

Table 15. Low cardinality Keys

Name	Description
`id` *(required)*	ID.
`type` *(required)*	TYPE.

Hostname Resolution Time

> Hostname resolution metric.

Metric name `%s` - since it contains `%s`, the name is dynamic and will be resolved at runtime. **Type** `timer`.

Metric name `%s.active` - since it contains `%s`, the name is dynamic and will be resolved at runtime. **Type** `long task timer`.

> ❗ KeyValues that are added after starting the Observation might be missing from the *.active metrics.

> ❗ Micrometer internally uses `nanoseconds` for the baseunit. However, each backend determines the actual baseunit. (i.e. Prometheus uses seconds)

Fully qualified name of the enclosing class `reactor.netty.transport.HostnameResolutionObservations`.

Table 16. Low cardinality Keys

Name	Description
`remote.address` *(required)*	Remote address.
`status` *(required)*	STATUS.

Table 17. High cardinality Keys

Name	Description
`net.peer.name` *(required)*	Net peer name.
`net.peer.port` *(required)*	Net peer port.
`reactor.netty.protocol` *(required)*	Reactor Netty protocol (tcp/http etc.).
`reactor.netty.status` *(required)*	Reactor Netty status.
`reactor.netty.type` *(required)*	Reactor Netty type (always client).

Http Client Data Received Time

> Time spent in consuming incoming data on the client.

Metric name `reactor.netty.http.client.data.received.time`. **Type** `timer`.

> KeyValues that are added after starting the Observation might be missing from the *.active metrics.

> Micrometer internally uses `nanoseconds` for the baseunit. However, each backend determines the actual baseunit. (i.e. Prometheus uses seconds)

Fully qualified name of the enclosing class `reactor.netty.http.client.HttpClientMeters`.

Table 18. Low cardinality Keys

Name	Description
`method` *(required)*	METHOD.
`proxy.address` *(required)*	Proxy address, when there is a proxy configured.
`remote.address` *(required)*	Remote address.
`status` *(required)*	STATUS.
`uri` *(required)*	URI.

Http Client Data Sent Time

> Time spent in sending outgoing data from the client.

Metric name `reactor.netty.http.client.data.sent.time`. **Type** `timer`.

> KeyValues that are added after starting the Observation might be missing from the *.active metrics.

> Micrometer internally uses `nanoseconds` for the baseunit. However, each backend determines the actual baseunit. (i.e. Prometheus uses seconds)

Fully qualified name of the enclosing class `reactor.netty.http.client.HttpClientMeters`.

Table 19. Low cardinality Keys

Name	Description
`method` *(required)*	METHOD.
`proxy.address` *(required)*	Proxy address, when there is a proxy configured.
`remote.address` *(required)*	Remote address.
`uri` *(required)*	URI.

Http Client Response Time

> Response metric.

Metric name `reactor.netty.http.client.response.time`. **Type** `timer`.

Metric name `reactor.netty.http.client.response.time.active`. **Type** `long task timer`.

> KeyValues that are added after starting the Observation might be missing from the *.active metrics.

> Micrometer internally uses `nanoseconds` for the baseunit. However, each backend determines the actual baseunit. (i.e. Prometheus uses seconds)

Fully qualified name of the enclosing class `reactor.netty.http.client.HttpClientObservations`.

Table 20. Low cardinality Keys

Name	Description
`method` *(required)*	METHOD.
`proxy.address` *(required)*	Proxy address, when there is a proxy configured.
`remote.address` *(required)*	Remote address.
`status` *(required)*	STATUS.
`uri` *(required)*	URI.

Table 21. High cardinality Keys

Name	Description
`http.status_code` *(required)*	Status code.
`http.url` *(required)*	URL.
`net.peer.name` *(required)*	Net peer name.
`net.peer.port` *(required)*	Net peer port.

reactor.netty.type *(required)*	Reactor Netty type (always client).

Http Server Data Received

> Amount of the data received, in bytes.

Metric name %s - since it contains %s, the name is dynamic and will be resolved at runtime. **Type** distribution summary and **base unit** bytes.

> ❗ KeyValues that are added after starting the Observation might be missing from the *.active metrics.

Fully qualified name of the enclosing class reactor.netty.http.server.HttpServerMeters.

Table 22. Low cardinality Keys

Name	Description
uri *(required)*	URI.

Http Server Data Received Time

> Time spent in consuming incoming data on the server.

Metric name reactor.netty.http.server.data.received.time. **Type** timer.

> ❗ KeyValues that are added after starting the Observation might be missing from the *.active metrics.

> ❗ Micrometer internally uses nanoseconds for the baseunit. However, each backend determines the actual baseunit. (i.e. Prometheus uses seconds)

Fully qualified name of the enclosing class reactor.netty.http.server.HttpServerMeters.

Table 23. Low cardinality Keys

Name	Description
method *(required)*	METHOD.
uri *(required)*	URI.

Http Server Data Sent

> Amount of the data sent, in bytes.

Metric name %s - since it contains %s, the name is dynamic and will be resolved at runtime. **Type** distribution summary and **base unit** bytes.

> ❗ KeyValues that are added after starting the Observation might be missing from the

*.active metrics.

Fully qualified name of the enclosing class `reactor.netty.http.server.HttpServerMeters`.

Table 24. Low cardinality Keys

Name	Description
`uri` *(required)*	URI.

Http Server Data Sent Time

> Time spent in sending outgoing data from the server.

Metric name `reactor.netty.http.server.data.sent.time`. **Type** `timer`.

> KeyValues that are added after starting the Observation might be missing from the *.active metrics.

> Micrometer internally uses `nanoseconds` for the baseunit. However, each backend determines the actual baseunit. (i.e. Prometheus uses seconds)

Fully qualified name of the enclosing class `reactor.netty.http.server.HttpServerMeters`.

Table 25. Low cardinality Keys

Name	Description
`method` *(required)*	METHOD.
`status` *(required)*	STATUS.
`uri` *(required)*	URI.

Http Server Errors Count

> Number of errors that occurred.

Metric name `%s` - since it contains `%s`, the name is dynamic and will be resolved at runtime. **Type** `counter`.

> KeyValues that are added after starting the Observation might be missing from the *.active metrics.

Fully qualified name of the enclosing class `reactor.netty.http.server.HttpServerMeters`.

Table 26. Low cardinality Keys

Name	Description
`uri` *(required)*	URI.

Http Server Response Time

> Response metric.

Metric name `reactor.netty.http.server.response.time`. **Type** `timer`.

Metric name `reactor.netty.http.server.response.time.active`. **Type** `long task timer`.

> KeyValues that are added after starting the Observation might be missing from the *.active metrics.

> Micrometer internally uses `nanoseconds` for the baseunit. However, each backend determines the actual baseunit. (i.e. Prometheus uses seconds)

Fully qualified name of the enclosing class `reactor.netty.http.server.HttpServerObservations`.

Table 27. Low cardinality Keys

Name	Description
`method` *(required)*	METHOD.
`status` *(required)*	STATUS.
`uri` *(required)*	URI.

Table 28. High cardinality Keys

Name	Description
`http.scheme` *(required)*	HTTP scheme.
`http.status_code` *(required)*	Status code.
`net.host.name` *(required)*	Net host name.
`net.host.port` *(required)*	Net host port.
`reactor.netty.type` *(required)*	Reactor Netty type (always server).

Idle Connections

> The number of the idle connections in the connection pool.

Metric name `reactor.netty.connection.provider.idle.connections`. **Type** `gauge`.

> KeyValues that are added after starting the Observation might be missing from the *.active metrics.

Fully qualified name of the enclosing class `reactor.netty.http.client.Http2ConnectionProviderMeters`.

Table 29. Low cardinality Keys

Name	Description

`id` *(required)*	ID.
`name` *(required)*	NAME.
`remote.address` *(required)*	Remote address.

Idle Connections

> The number of the idle connections in the connection pool.

Metric name `reactor.netty.connection.provider.idle.connections`. **Type** `gauge`.

> KeyValues that are added after starting the Observation might be missing from the *.active metrics.

Fully qualified name of the enclosing class `reactor.netty.resources.ConnectionProviderMeters`.

Table 30. Low cardinality Keys

Name	Description
`id` *(required)*	ID.
`name` *(required)*	NAME.
`remote.address` *(required)*	Remote address.

Max Connections

> The maximum number of active connections that are allowed in the connection pool.

Metric name `reactor.netty.connection.provider.max.connections`. **Type** `gauge`.

> KeyValues that are added after starting the Observation might be missing from the *.active metrics.

Fully qualified name of the enclosing class `reactor.netty.resources.ConnectionProviderMeters`.

Table 31. Low cardinality Keys

Name	Description
`id` *(required)*	ID.
`name` *(required)*	NAME.
`remote.address` *(required)*	Remote address.

Max Pending Connections

> The maximum number of requests that will be queued while waiting for a ready connection from the connection pool.

Metric name `reactor.netty.connection.provider.max.pending.connections`. **Type** `gauge`.

> KeyValues that are added after starting the Observation might be missing from the *.active metrics.

Fully qualified name of the enclosing class `reactor.netty.resources.ConnectionProviderMeters`.

Table 32. Low cardinality Keys

Name	Description
`id` *(required)*	ID.
`name` *(required)*	NAME.
`remote.address` *(required)*	Remote address.

Normal Cache Size

> The size of the normal cache.

Metric name `reactor.netty.bytebuf.allocator.normal.cache.size`. **Type** `gauge`.

> KeyValues that are added after starting the Observation might be missing from the *.active metrics.

Fully qualified name of the enclosing class `reactor.netty.transport.ByteBufAllocatorMeters`.

Table 33. Low cardinality Keys

Name	Description
`id` *(required)*	ID.
`type` *(required)*	TYPE.

Pending Connections

> The number of the request, that are pending acquire a connection from the connection pool.

Metric name `reactor.netty.connection.provider.pending.connections`. **Type** `gauge`.

> KeyValues that are added after starting the Observation might be missing from the *.active metrics.

Fully qualified name of the enclosing class `reactor.netty.resources.ConnectionProviderMeters`.

Table 34. Low cardinality Keys

Name	Description
`id` *(required)*	ID.

`name` *(required)*	NAME.
`remote.address` *(required)*	Remote address.

Pending Connections Time

> Time spent in pending acquire a connection from the connection pool.

Metric name `reactor.netty.connection.provider.pending.connections.time`. **Type** `timer`.

> ❗ KeyValues that are added after starting the Observation might be missing from the *.active metrics.

> ❗ Micrometer internally uses `nanoseconds` for the baseunit. However, each backend determines the actual baseunit. (i.e. Prometheus uses seconds)

Fully qualified name of the enclosing class `reactor.netty.resources.ConnectionProviderMeters`.

Table 35. Low cardinality Keys

Name	Description
`id` *(required)*	ID.
`name` *(required)*	NAME.
`remote.address` *(required)*	Remote address.
`status` *(required)*	STATUS.

Pending Streams

> The number of requests that are waiting for opening HTTP/2 stream.

Metric name `reactor.netty.connection.provider.pending.streams`. **Type** `gauge`.

> ❗ KeyValues that are added after starting the Observation might be missing from the *.active metrics.

Fully qualified name of the enclosing class `reactor.netty.http.client.Http2ConnectionProviderMeters`.

Table 36. Low cardinality Keys

Name	Description
`id` *(required)*	ID.
`name` *(required)*	NAME.
`remote.address` *(required)*	Remote address.

Pending Streams Time

> Time spent in pending acquire a stream from the connection pool.

Metric name `reactor.netty.connection.provider.pending.streams.time`. **Type** `timer`.

> KeyValues that are added after starting the Observation might be missing from the *.active metrics.

> Micrometer internally uses `nanoseconds` for the baseunit. However, each backend determines the actual baseunit. (i.e. Prometheus uses seconds)

Fully qualified name of the enclosing class `reactor.netty.http.client.Http2ConnectionProviderMeters`.

Table 37. Low cardinality Keys

Name	Description
`id` *(required)*	ID.
`name` *(required)*	NAME.
`remote.address` *(required)*	Remote address.
`status` *(required)*	STATUS.

Pending Tasks

> Event loop pending scheduled tasks.

Metric name `reactor.netty.eventloop.pending.tasks`. **Type** `gauge`.

> KeyValues that are added after starting the Observation might be missing from the *.active metrics.

Fully qualified name of the enclosing class `reactor.netty.transport.EventLoopMeters`.

Table 38. Low cardinality Keys

Name	Description
`name` *(required)*	NAME.

Small Cache Size

> The size of the small cache.

Metric name `reactor.netty.bytebuf.allocator.small.cache.size`. **Type** `gauge`.

> KeyValues that are added after starting the Observation might be missing from the *.active metrics.

Fully qualified name of the enclosing class `reactor.netty.transport.ByteBufAllocatorMeters`.

Table 39. Low cardinality Keys

Name	Description
`id` *(required)*	ID.
`type` *(required)*	TYPE.

Streams Active

> The number of HTTP/2 streams currently active on the server.

Metric name `reactor.netty.http.server.streams.active`. **Type** `gauge`.

> KeyValues that are added after starting the Observation might be missing from the *.active metrics.

Fully qualified name of the enclosing class `reactor.netty.http.server.HttpServerMeters`.

Table 40. Low cardinality Keys

Name	Description
`local.address` *(required)*	Local address.
`uri` *(required)*	URI.

Thread Local Caches

> The number of thread local caches.

Metric name `reactor.netty.bytebuf.allocator.threadlocal.caches`. **Type** `gauge`.

> KeyValues that are added after starting the Observation might be missing from the *.active metrics.

Fully qualified name of the enclosing class `reactor.netty.transport.ByteBufAllocatorMeters`.

Table 41. Low cardinality Keys

Name	Description
`id` *(required)*	ID.
`type` *(required)*	TYPE.

Tls Handshake Time

> TLS handshake metric.

Metric name `%s` - since it contains `%s`, the name is dynamic and will be resolved at runtime. **Type**

`timer`.

Metric name `%s.active` - since it contains `%s`, the name is dynamic and will be resolved at runtime. **Type** `long task timer`.

> KeyValues that are added after starting the Observation might be missing from the *.active metrics.

> Micrometer internally uses `nanoseconds` for the baseunit. However, each backend determines the actual baseunit. (i.e. Prometheus uses seconds)

Fully qualified name of the enclosing class `reactor.netty.tcp.TlsHandshakeObservations`.

Table 42. Low cardinality Keys

Name	Description
`proxy.address` *(required)*	Proxy address, when there is a proxy configured.
`remote.address` *(required)*	Remote address.
`status` *(required)*	STATUS.

Table 43. High cardinality Keys

Name	Description
`reactor.netty.protocol` *(required)*	Reactor Netty protocol (tcp/http etc.).
`reactor.netty.status` *(required)*	Reactor Netty status.
`reactor.netty.type` *(required)*	Reactor Netty type (client/server).

Total Connections

> The number of all connections in the connection pool, active or idle.

Metric name `reactor.netty.connection.provider.total.connections`. **Type** `gauge`.

> KeyValues that are added after starting the Observation might be missing from the *.active metrics.

Fully qualified name of the enclosing class `reactor.netty.resources.ConnectionProviderMeters`.

Table 44. Low cardinality Keys

Name	Description
`id` *(required)*	ID.
`name` *(required)*	NAME.
`remote.address` *(required)*	Remote address.

Used Direct Memory

> The number of bytes reserved by direct buffer allocator.

Metric name `reactor.netty.bytebuf.allocator.used.direct.memory`. **Type** `gauge`.

> KeyValues that are added after starting the Observation might be missing from the *.active metrics.

Fully qualified name of the enclosing class `reactor.netty.transport.ByteBufAllocatorMeters`.

Table 45. Low cardinality Keys

Name	Description
`id` *(required)*	ID.
`type` *(required)*	TYPE.

Used Heap Memory

> The number of bytes reserved by heap buffer allocator.

Metric name `reactor.netty.bytebuf.allocator.used.heap.memory`. **Type** `gauge`.

> KeyValues that are added after starting the Observation might be missing from the *.active metrics.

Fully qualified name of the enclosing class `reactor.netty.transport.ByteBufAllocatorMeters`.

Table 46. Low cardinality Keys

Name	Description
`id` *(required)*	ID.
`type` *(required)*	TYPE.

Observability - Spans

Below you can find a list of all spans declared by this project.

Connect Span

> Connect Span.

Span name `%s` - since it contains `%s`, the name is dynamic and will be resolved at runtime.

Fully qualified name of the enclosing class `reactor.netty.channel.ConnectSpans`.

Table 47. Tag Keys

Name	Description

`net.peer.name` *(required)*	Net peer name.
`net.peer.port` *(required)*	Net peer port.
`reactor.netty.protocol` *(required)*	Reactor Netty protocol (tcp/http etc.).
`reactor.netty.status` *(required)*	Reactor Netty status.
`reactor.netty.type` *(required)*	Reactor Netty type (always client).

Hostname Resolution Span

> Hostname Resolution Span.

Span name `%s` - since it contains `%s`, the name is dynamic and will be resolved at runtime.

Fully qualified name of the enclosing class `reactor.netty.transport.HostnameResolutionSpans`.

Table 48. Tag Keys

Name	Description
`net.peer.name` *(required)*	Net peer name.
`net.peer.port` *(required)*	Net peer port.
`reactor.netty.protocol` *(required)*	Reactor Netty protocol (tcp/http etc.).
`reactor.netty.status` *(required)*	Reactor Netty status.
`reactor.netty.type` *(required)*	Reactor Netty type (always client).

Http Client Response Span

> Response Span.

Span name `%s` - since it contains `%s`, the name is dynamic and will be resolved at runtime.

Fully qualified name of the enclosing class `reactor.netty.http.client.HttpClientSpans`.

Table 49. Tag Keys

Name	Description
`http.status_code` *(required)*	Status code.
`http.url` *(required)*	URL.
`net.peer.name` *(required)*	Net peer name.
`net.peer.port` *(required)*	Net peer port.
`reactor.netty.type` *(required)*	Reactor Netty type (always client).

Http Server Response Span

> Response Span.

Span name `%s` - since it contains `%s`, the name is dynamic and will be resolved at runtime.

Fully qualified name of the enclosing class `reactor.netty.http.server.HttpServerSpans`.

Table 50. Tag Keys

Name	Description
`http.scheme` *(required)*	HTTP scheme.
`http.status_code` *(required)*	Status code.
`net.host.name` *(required)*	Net host name.
`net.host.port` *(required)*	Net host port.
`reactor.netty.type` *(required)*	Reactor Netty type (always server).

Tls Handshake Span

> TLS Handshake Span.

Span name `%s` - since it contains `%s`, the name is dynamic and will be resolved at runtime.

Fully qualified name of the enclosing class `reactor.netty.tcp.TlsHandshakeSpans`.

Table 51. Tag Keys

Name	Description
`reactor.netty.protocol` *(required)*	Reactor Netty protocol (tcp/http etc.).
`reactor.netty.status` *(required)*	Reactor Netty status.
`reactor.netty.type` *(required)*	Reactor Netty type (client/server).
`remote.address` *(required)*	Remote address.