

Reactor Netty Reference Guide

Stephane Maldini, Violeta Georgieva

Version 0.9.8.RELEASE

Table of Contents

1. About the Documentation	1
1.1. Latest Version and Copyright Notice	1
1.2. Contributing to the Documentation	1
1.3. Getting Help	1
2. Getting Started	2
2.1. Introducing Reactor Netty	2
2.2. Prerequisites	2
2.3. Understanding the BOM	2
2.4. Getting Reactor Netty	3
3. TCP Server	6
3.1. Starting and Stopping	6
3.2. Writing Data	7
3.3. Consuming Data	8
3.4. Lifecycle Callbacks	8
3.5. TCP-level Configurations	9
3.6. SSL/TLS	13
3.7. Metrics	14
4. TCP Client	17
4.1. Connect and Disconnect	17
4.2. Writing Data	18
4.3. Consuming Data	19
4.4. Lifecycle Callbacks	20
4.5. TCP-level Configurations	21
4.6. Connection Pool	25
4.7. SSL and TLS	28
4.8. Proxy Support	29
4.9. Metrics	30
5. HTTP Server	33
5.1. Starting and Stopping	33
5.2. Routing HTTP	34
5.3. Writing Data	37
5.4. Consuming Data	40
5.5. TCP-level Configuration	43
5.6. SSL and TLS	45
5.7. HTTP Access Log	45
5.8. HTTP/2	46
5.9. Metrics	48
6. HTTP Client	52

6.1. Connect	52
6.2. Writing Data	54
6.3. Consuming Data	57
6.4. TCP-level Configuration	60
6.5. SSL and TLS	61
6.6. Retry Strategies	62
6.7. Metrics	62
7. UDP Server	66
7.1. Starting and Stopping	66
7.2. Writing Data	67
7.3. Consuming Data	68
7.4. Lifecycle Callbacks	69
7.5. Connection Configuration	70
7.6. Metrics	74
8. UDP Client	77
8.1. Connecting and Disconnecting	77
8.2. Writing Data	78
8.3. Consuming Data	79
8.4. Lifecycle Callbacks	80
8.5. Connection Configuration	81
8.6. Metrics	85

Chapter 1. About the Documentation

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.

1.1. Latest Version and Copyright Notice

The **Reactor Netty** reference guide is available as **HTML** documents. The latest copy is available at <https://projectreactor.io/docs/netty/release/reference/index.html>

Copies of this document may be made for your own use and for distribution to others, provided that you do not charge any fee for such copies and further provided that each copy contains this **Copyright Notice**, whether distributed in print or electronically.

1.2. Contributing to the Documentation

The reference guide is written in **AsciiDoc**, and you can find its sources at <https://github.com/reactor/reactor-netty/tree/master/src/docs/asciidoc>.

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 **asciidocctor** Gradle task and checking the rendering. Some of the sections rely on included files, so **GitHub** rendering is not always complete.

1.3. 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](#)
- [Getting Reactor Netty](#)

2.1. Introducing Reactor Netty

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

2.2. Prerequisites

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

It has transitive dependencies on:

- Reactive Streams v1.0.3
- Reactor Core v3.x
- Netty v4.1.x

2.3. Understanding the BOM

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 **BOM** (Bill of Materials) is itself versioned, using a release train scheme with a codename followed by a qualifier. The following list shows a few examples:

```
Aluminium-RELEASE
Californium-BUILD-SNAPSHOT
Aluminium-SR1
Bismuth-RELEASE
Californium-SR32
```

The codenames represent what would traditionally be the **MAJOR.MINOR** number. They (mostly) come from the [Periodic Table of Elements](#), in increasing alphabetical order.

The qualifiers are (in chronological order):

- **BUILD-SNAPSHOT**
- **M1..N**: Milestones or developer previews
- **RELEASE**: The first **GA** (General Availability) release in a codename series
- **SR1..N**: The subsequent **GA** releases in a codename series (equivalent to **PATCH** number — **SR** stands for **Service Release**).

2.4. 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.

2.4.1. 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.

```
<dependencyManagement> ❶
  <dependencies>
    <dependency>
      <groupId>io.projectreactor</groupId>
      <artifactId>reactor-bom</artifactId>
      <version>Californium-RELEASE</version>
      <type>pom</type>
      <scope>import</scope>
    </dependency>
  </dependencies>
</dependencyManagement>
```

❶ Notice the **dependencyManagement** tag. This is in addition to the regular **dependencies** section.

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

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

① Dependency on **Reactor Netty**

② No version tag here

2.4.2. Gradle Installation

Gradle has no core support for Maven BOMs, but you can use Spring's [gradle-dependency-management](#) plugin.

First, apply the plugin from the Gradle Plugin Portal. The following example shows how to do so:

```
plugins {
  id "io.spring.dependency-management" version "1.0.7.RELEASE" ①
}
```

① as of this writing, **1.0.7.RELEASE** is the latest version of the plugin. Check for updates.

Then use it to import the BOM. The following listing shows how to do so:

```
dependencyManagement {
  imports {
    mavenBom "io.projectreactor:reactor-bom:Californium-RELEASE"
  }
}
```

Finally, add a dependency to your project, without a version number. The following listing shows how to do so:

```
dependencies {
  compile 'io.projectreactor.netty:reactor-netty' ①
}
```

① There is no third **:** separated section for the version. It is taken from the **BOM**.

2.4.3. 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):

BUILD-SNAPSHOTS in Maven

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

BUILD-SNAPSHOTS in Gradle

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


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.

3.1. 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:

```
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.

3.1.1. Host and Port

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

```
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

3.2. 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:

```
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

3.3. 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:

```
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())
                .bindNow();

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

① Receives data from the connected clients

3.4. Lifecycle Callbacks

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

- `doOnBind`: Invoked when the server channel is about to bind.
- `doOnBound`: Invoked when the server channel is bound.
- `doOnConnection`: Invoked when a remote client is connected
- `doOnUnbound`: Invoked when the server channel is unbound.
- `doOnLifecycle`: Sets up all lifecycle callbacks.

The following example uses the `doOnConnection` callback:

```
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.addHandler(new ReadTimeoutHandler(10,
TimeUnit.SECONDS))) ①
                .bindNow();

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

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

3.5. TCP-level Configurations

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

- [Setting Channel Options](#)
- [Using a Wire Logger](#)
- [Using an Event Loop Group](#)

3.5.1. Setting Channel Options

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

./../main/java/reactor/netty/tcp/TcpServerBind.java

```
ServerBootstrap createServerBootstrap() {
    return new ServerBootstrap()
        .option(ChannelOption.SO_REUSEADDR, true)
        .childOption(ChannelOption.AUTO_READ, false)
        .childOption(ChannelOption.TCP_NODELAY, true)
        .localAddress(new InetSocketAddress(DEFAULT_PORT));
}
```

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

following configuration:

```
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:

- [ChannelOption](#)
- [Socket Options](#)

3.5.2. Using a Wire Logger

Reactor Netty provides wire logging for when the traffic between the peers has 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;

```
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

3.5.3. Using an Event Loop Group

By default, the **TCP** server 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 [LoopResource#create](#) methods.

The default configuration for the **Event Loop Group** is the following:

../../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)
 */
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 changes to the these settings are needed, you can apply the following configuration:

```
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();
    }
}
```

3.6. SSL/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`:


```

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) {
        DisposableServer server =
            TcpServer.create()
                .secure(spec ->
                    SslContextBuilder.forServer(new
File("certificate.crt"),
                                new
File("private.key")))
                .bindNow();

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

```

3.7. 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.data.received	DistributionSummary	Amount of the data received, in bytes
reactor.netty.tcp.server.data.sent	DistributionSummary	Amount of the data sent, in bytes
reactor.netty.tcp.server.errors	Counter	Number of errors that occurred
reactor.netty.tcp.server.tls.handshake.time	Timer	Time spent for TLS handshake

These additional metrics are also available:

ByteBufAllocator metrics

metric name	type	description
reactor.netty.bytebuf.allocator.used.heap.memory	Gauge	The number of the bytes of the heap memory

metric name	type	description
reactor.netty.bytebuf allocator. used.direct.memory	Gauge	The number of the bytes of the direct memory
reactor.netty.bytebuf allocator. used.heap.arenas	Gauge	The number of heap arenas (when <code>PooledByteBufAllocator</code>)
reactor.netty.bytebuf allocator. used.direct.arenas	Gauge	The number of direct arenas (when <code>PooledByteBufAllocator</code>)
reactor.netty.bytebuf allocator. used.threadlocal.caches	Gauge	The number of thread local caches (when <code>PooledByteBufAllocator</code>)
reactor.netty.bytebuf allocator. used.tiny.cache.size	Gauge	The size of the tiny cache (when <code>PooledByteBufAllocator</code>)
reactor.netty.bytebuf allocator. used.small.cache.size	Gauge	The size of the small cache (when <code>PooledByteBufAllocator</code>)
reactor.netty.bytebuf allocator. used.normal.cache.size	Gauge	The size of the normal cache (when <code>PooledByteBufAllocator</code>)
reactor.netty.bytebuf allocator. used.chunk.size	Gauge	The chunk size for an arena (when <code>PooledByteBufAllocator</code>)

The following example enables that integration:

```
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:

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

public class Application {

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

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

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

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.

4.1. 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`:

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

public class Application {

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

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

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

❷ 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.

4.1.1. 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:

```
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 **HTTP** host

② Configures the **HTTP** port

4.2. 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.

```
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.

4.3. 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:

```

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())
                .connectNow();

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

```

① Receives data from a given endpoint

4.4. Lifecycle Callbacks

The following lifecycle callbacks are provided to let you extend the **TCP** client.

- **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.
- **doOnLifecycle**: Sets up all lifecycle callbacks.

The following example uses the **doOnConnected** callback:

```
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.addHandler(new ReadTimeoutHandler(10,
TimeUnit.SECONDS))) ①
                .connectNow();

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

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

4.5. 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](#)

4.5.1. Channel Options

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

```
./../main/java/reactor/netty/tcp/TcpClient.java
```

```
static final Bootstrap DEFAULT_BOOTSTRAP =
    new Bootstrap().option(ChannelOption.AUTO_READ, false)

    .remoteAddress(InetSocketAddressUtil.createUnresolved(NetUtil.LOCALHOST.getHostAdd
ress(), DEFAULT_PORT));
```

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

following configuration:

```
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:

- [ChannelOption](#)
- [Socket Options](#)

4.5.2. Wire Logger

Reactor Netty provides wire logging for when the traffic between the peers has 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:

```
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

4.5.3. Event Loop Group

By default the **TCP** client 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 [LoopResource#create](#) methods.

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

../../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)
 */
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 the these settings, you can apply the following configuration:

```

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();
    }
}

```

4.6. Connection Pool

By default, the **TCP** client uses a “fixed” connection pool with **500** as the maximum number of channels and **45s** as the acquisition timeout. This means that the implementation creates a new channel if someone tries to acquire a channel but none is in the pool. When the maximum number of the channels in the pool is reached, new tries to acquire a channel are delayed until a channel is returned to the pool again. The implementation uses **FIFO** order for channels in the pool. By default, there is no idle time specified for the channels in the pool.

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

```

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();
    }
}

```

If you need to specify an idle time for the channels in the connection pool, you can apply the following configuration:

```

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.ofMillis(30000))
                .maxIdleTime(Duration.ofMillis(60))
                .build();

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

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

```



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.

4.6.1. 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
<code>reactor.netty.connection.provider.total.connections</code>	Gauge	The number of all connections, active or idle
<code>reactor.netty.connection.provider.active.connections</code>	Gauge	The number of the connections that have been successfully acquired and are in active use
<code>reactor.netty.connection.provider.idle.connections</code>	Gauge	The number of the idle connections
<code>reactor.netty.connection.provider.pending.connections</code>	Gauge	The number of requests that are waiting for a connection

The following example enables that integration:

```

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

4.7. 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**:

```
import io.netty.handler.ssl.SslContextBuilder;
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(443)
                .secure(spec ->
spec.sslContext(SslContextBuilder.forClient()))
                .connectNow();

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

4.8. Proxy Support

The TCP client supports the proxy functionality provided by Netty and provides a way to specify “non proxy hosts” through the `ProxyProvider` builder. The following example uses `ProxyProvider`:


```

import reactor.netty.Connection;
import reactor.netty.tcp.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"))
                .connectNow();

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

```

4.9. 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
reactor.netty.tcp.client.data.sent	DistributionSummary	Amount of the data sent, in bytes
reactor.netty.tcp.client.errors	Counter	Number of errors that occurred
reactor.netty.tcp.client.tls.handshake.time	Timer	Time spent for TLS handshake
reactor.netty.tcp.client.connect.time	Timer	Time spent for connecting to the remote address
reactor.netty.tcp.client.address.resolver	Timer	Time spent for resolving the address

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
reactor.netty.connection.provider.active.connections	Gauge	The number of the connections that have been successfully acquired and are in active use
reactor.netty.connection.provider.idle.connections	Gauge	The number of the idle connections
reactor.netty.connection.provider.pending.connections	Gauge	The number of requests that are waiting for a connection

ByteBufAllocator metrics

metric name	type	description
reactor.netty.bytebuf.allocator.used.heap.memory	Gauge	The number of the bytes of the heap memory
reactor.netty.bytebuf.allocator.used.direct.memory	Gauge	The number of the bytes of the direct memory
reactor.netty.bytebuf.allocator.used.heap.arenas	Gauge	The number of heap arenas (when <code>PooledByteBufAllocator</code>)
reactor.netty.bytebuf.allocator.used.direct.arenas	Gauge	The number of direct arenas (when <code>PooledByteBufAllocator</code>)
reactor.netty.bytebuf.allocator.used.threadlocal.caches	Gauge	The number of thread local caches (when <code>PooledByteBufAllocator</code>)
reactor.netty.bytebuf.allocator.used.tiny.cache.size	Gauge	The size of the tiny cache (when <code>PooledByteBufAllocator</code>)
reactor.netty.bytebuf.allocator.used.small.cache.size	Gauge	The size of the small cache (when <code>PooledByteBufAllocator</code>)
reactor.netty.bytebuf.allocator.used.normal.cache.size	Gauge	The size of the normal cache (when <code>PooledByteBufAllocator</code>)
reactor.netty.bytebuf.allocator.used.chunk.size	Gauge	The chunk size for an arena (when <code>PooledByteBufAllocator</code>)

The following example enables that integration:

```

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:

```

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, () -> new CustomChannelMetricsRecorder())
                .connectNow();

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

```

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

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.

5.1. 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:

```
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.

5.1.1. Host and Port

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

```
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

5.2. Routing HTTP

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

```

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.



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

5.2.1. SSE

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

```

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();
}

```

5.2.2. Static Resources

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

```

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.file("/index.html", pathToFile))
                .bindNow();

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

```

5.3. 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:


```
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

5.3.1. 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:

```

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();
    }
}

```

5.3.2. Compression

You can configure the **HTTP** server to send a compressed response, depending on the request header **Accept-Encoding** or (in the case of websocket) the **Sec-WebSocket-Extensions** header.

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:

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

public class Application {

    public static void main(String[] args) {
        DisposableServer server =
            HttpServer.create()
                .compress(true)
                .route(routes -> routes.file("/index.html", pathToFile))
                .bindNow();

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

5.4. 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 `HttpRequest`, to be able to read data.

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

```
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())
                .bindNow();

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

① Receives data from the connected clients

5.4.1. 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 `HttpRequest`. The following example shows how to do so:

```
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();
    }
}
```

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:

```

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.

5.4.2. 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:

../../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;
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;
```

../../main/java/reactor/netty/http/server/HttpRequestDecoderSpec.java

```
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:

```
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))
                .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.

5.5. 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:

```

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()
                .tcpConfiguration(tcpServer ->

tcpServer.option(ChannelOption.CONNECT_TIMEOUT_MILLIS, 10000))
                .bindNow();

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

```

See [TCP Server](#) for more detail about TCP-level configuration.

5.5.1. Wire Logger

Reactor Netty provides wire logging for when you need to inspect the traffic between the peers. 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:

```

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) ❶
                .bindNow();

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

```

❶ Enables the wire logging

5.6. 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`:

```
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) {
        File cert = new File("certificate.crt");
        File key = new File("private.key");
        DisposableServer server =
            HttpServer.create()
                .secure(spec ->
spec.sslContext(SslContextBuilder.forServer(cert,key)))
                .bindNow();

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

5.7. HTTP Access Log

The current logging support provides only the [Common Log Format](#).

You can use `-Dreactor.netty.http.server.accessLogEnabled=true` to enable the `HTTP` access log. By default, it is disabled.

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>
```

5.8. 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 that protocol is not supported “out-of-the-box” by 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:

```

import io.netty.handler.ssl.SslContextBuilder;
import reactor.core.publisher.Mono;
import reactor.netty.DisposableServer;
import reactor.netty.http.HttpProtocol;
import reactor.netty.http.server.HttpServer;
import java.io.File;

public class Application {

    public static void main(String[] args) {
        DisposableServer server =
            HttpServer.create()
                .port(8080)
                .protocol(HttpProtocol.H2) ①
                .secure(spec -> ②
                    SslContextBuilder.forServer(new
File("certificate.crt"),
                                new
File("private.key")))
                .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:

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

public class Application {

    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
```

5.9. 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.data.received	DistributionSummary	Amount of the data received, in bytes
reactor.netty.http.server.data.sent	DistributionSummary	Amount of the data sent, in bytes
reactor.netty.http.server.errors	Counter	Number of errors that occurred
reactor.netty.http.server.data.received.time	Timer	Time spent in consuming incoming data

metric name	type	description
reactor.netty.http.server.data.sent.time	Timer	Time spent in sending outgoing data
reactor.netty.http.server.response.time	Timer	Total time for the request/response

These additional metrics are also available:

ByteBufAllocator metrics

metric name	type	description
reactor.netty.bytebuf.allocator.used.heap.memory	Gauge	The number of the bytes of the heap memory
reactor.netty.bytebuf.allocator.used.direct.memory	Gauge	The number of the bytes of the direct memory
reactor.netty.bytebuf.allocator.used.heap.arenas	Gauge	The number of heap arenas (when <code>PooledByteBufAllocator</code>)
reactor.netty.bytebuf.allocator.used.direct.arenas	Gauge	The number of direct arenas (when <code>PooledByteBufAllocator</code>)
reactor.netty.bytebuf.allocator.used.threadlocal.caches	Gauge	The number of thread local caches (when <code>PooledByteBufAllocator</code>)
reactor.netty.bytebuf.allocator.used.tiny.cache.size	Gauge	The size of the tiny cache (when <code>PooledByteBufAllocator</code>)
reactor.netty.bytebuf.allocator.used.small.cache.size	Gauge	The size of the small cache (when <code>PooledByteBufAllocator</code>)
reactor.netty.bytebuf.allocator.used.normal.cache.size	Gauge	The size of the normal cache (when <code>PooledByteBufAllocator</code>)
reactor.netty.bytebuf.allocator.used.chunk.size	Gauge	The chunk size for an arena (when <code>PooledByteBufAllocator</code>)

The following example enables that integration:

```

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 ❶
            .config()

        .meterFilter(MeterFilter.maximumAllowableTags("reactor.netty.http.server", "URI",
100, MeterFilter.deny()));

        DisposableServer server =
            HttpServer.create()
                .metrics(true, s -> {
                    if (s.startsWith("/stream/")) { ❷
                        return "/stream/{n}";
                    }
                    else if (s.startsWith("/bytes/")) {
                        return "/bytes/{n}";
                    }
                    return s;
                }) ❸
                .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();
    }
}

```

- ❶ Applies upper limit for the meters with **URI** tag
- ❷ Templated URIs will be used as an URI tag value when possible
- ❸ 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:

```
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()
                .metrics(true, () -> new
CustomHttpServerMetricsRecorder()) 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.

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 a `HTTP` client and adds Reactive Streams backpressure.

6.1. Connect

To connect the `HTTP` client to a given `HTTP` endpoint, you must create and configure a `HttpClient` instance. The following example shows how to do so:

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

public class Application {

    public static void main(String[] args) {
        HttpClientResponse response =
            HttpClient.create()           ❶
                .get()                   ❷
                .uri("http://example.com/") ❸
                .response()               ❹
                .block();

    }
}
```

- ❶ Creates a `HttpClient` instance ready for configuring.
- ❷ Specifies that `GET` method will be used.
- ❸ Specifies the path.
- ❹ Obtains the response `HttpClientResponse`

The following example uses `WebSocket`:

```

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.create()
            .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();
    }
}

```

6.1.1. Host and Port

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


```

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

public class Application {

    public static void main(String[] args) {
        HttpClientResponse response =
            HttpClient.create()
                .tcpConfiguration(tcpClient ->
tcpClient.host("example.com")) ①
                .port(80)
②
                .get()
                .uri("/")
                .response()
                .block();
    }
}

```

① Configures the **HTTP** host

② Configures the **HTTP** port

6.2. 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**:

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

public class Application {

    public static void main(String[] args) {
        HttpClientResponse response =
            HttpClient.create()
                .post()
                .uri("http://example.com/")
                .send(ByteBufFlux.fromString(Mono.just("hello"))) ①
                .response()
                .block();
    }
}
```

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

6.2.1. 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:

```

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

public class Application {

    public static void main(String[] args) {
        HttpClientResponse response =
            HttpClient.create()
                .headers(h -> h.set(HttpHeaderNames.CONTENT_LENGTH, 5))
                .post()
                .uri("http://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** (or, in the case of WebSocket, the **Sec-WebSocket-Extensions** header) is added to the request headers. The following example shows how to do so:

```

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

public class Application {

    public static void main(String[] args) {
        HttpClientResponse response =
            HttpClient.create()
                .compress(true)
                .get()
                .uri("http://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|307|308`.
- `followRedirect(BiPredicate<HttpClientRequest, HttpClientResponse>)`: Enables auto-redirect support if the supplied predicate matches.

The following example uses `followRedirect(true)`:

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

public class Application {

    public static void main(String[] args) {
        HttpClientResponse response =
            HttpClient.create()
                .followRedirect(true)
                .get()
                .uri("http://example.com/")
                .response()
                .block();
    }
}
```

6.3. 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:

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

public class Application {

    public static void main(String[] args) {
        String response =
            HttpClient.create()
                .get()
                .uri("http://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

6.3.1. 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.

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

public class Application {

    public static void main(String[] args) {
        String response =
            HttpClient.create()
                .get()
                .uri("http://example.com/")
                .responseSingle((resp, bytes) -> {
                    System.out.println(resp.status()); ①
                    return bytes.asString();
                })
                .block();
    }
}
```

- ① Obtains the status code.

6.3.2. 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:

../../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;
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;
```

../../main/java/reactor/netty/http/client/HttpResponseDecoderSpec.java

```
public static final boolean DEFAULT_FAIL_ON_MISSING_RESPONSE      = false;
public static final boolean DEFAULT_PARSE_HTTP_AFTER_CONNECT_REQUEST = false;
```

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

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

public class Application {

    public static void main(String[] args) {
        String response =
            HttpClient.create()
                .httpResponseDecoder(spec -> spec.maxHeaderSize(16384)) ①
                .get()
                .uri("http://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.

6.4. TCP-level Configuration

When you need configurations on a TCP level, you can use the following snippet to extend the default **TCP** client configuration:

```
import io.netty.channel.ChannelOption;
import reactor.netty.http.client.HttpClient;
import reactor.netty.http.client.HttpClientResponse;

public class Application {

    public static void main(String[] args) {
        HttpClientResponse response =
            HttpClient.create()
                .tcpConfiguration(tcpClient ->

tcpClient.option(ChannelOption.CONNECT_TIMEOUT_MILLIS, 10000))
                .get()
                .uri("http://example.com/")
                .response()
                .block();
    }
}
```

See [TCP Client](#) for more about **TCP** level configurations.

6.4.1. 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:

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

public class Application {

    public static void main(String[] args) {
        HttpClientResponse response =
            HttpClient.create()
                .wiretap(true) ❶
                .get()
                .uri("http://example.com/")
                .response()
                .block();
    }
}
```

❶ Enables the wire logging

6.5. 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`:


```
import io.netty.handler.ssl.SslContextBuilder;
import reactor.netty.http.client.HttpClient;
import reactor.netty.http.client.HttpClientResponse;

public class Application {

    public static void main(String[] args) {
        HttpClientResponse response =
            HttpClient.create()
                .secure(spec ->
spec.sslContext(SslContextBuilder.forClient()))
                .get()
                .uri("https://example.com/")
                .response()
                .block();
    }
}
```

6.6. Retry Strategies

By default, the **HTTP** client retries the request once if it was aborted on the **TCP** level.

6.7. 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
reactor.netty.http.client.data.sent	DistributionSummary	Amount of the data sent, in bytes
reactor.netty.http.client.errors	Counter	Number of errors that occurred
reactor.netty.http.client.tls.handshake.time	Timer	Time spent for TLS handshake
reactor.netty.http.client.connect.time	Timer	Time spent for connecting to the remote address
reactor.netty.http.client.address.resolver	Timer	Time spent for resolving the address
reactor.netty.http.client.data.received.time	Timer	Time spent in consuming incoming data

metric name	type	description
reactor.netty.http.client.data.sent.time	Timer	Time spent in sending outgoing data
reactor.netty.http.client.response.time	Timer	Total time for the request/response

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
reactor.netty.connection.provider.active.connections	Gauge	The number of the connections that have been successfully acquired and are in active use
reactor.netty.connection.provider.idle.connections	Gauge	The number of the idle connections
reactor.netty.connection.provider.pending.connections	Gauge	The number of requests that are waiting for a connection

ByteBufAllocator metrics

metric name	type	description
reactor.netty.bytebuf.allocator.used.heap.memory	Gauge	The number of the bytes of the heap memory
reactor.netty.bytebuf.allocator.used.direct.memory	Gauge	The number of the bytes of the direct memory
reactor.netty.bytebuf.allocator.used.heap.arenas	Gauge	The number of heap arenas (when PooledByteBufAllocator)
reactor.netty.bytebuf.allocator.used.direct.arenas	Gauge	The number of direct arenas (when PooledByteBufAllocator)
reactor.netty.bytebuf.allocator.used.threadlocal.caches	Gauge	The number of thread local caches (when PooledByteBufAllocator)
reactor.netty.bytebuf.allocator.used.tiny.cache.size	Gauge	The size of the tiny cache (when PooledByteBufAllocator)
reactor.netty.bytebuf.allocator.used.small.cache.size	Gauge	The size of the small cache (when PooledByteBufAllocator)
reactor.netty.bytebuf.allocator.used.normal.cache.size	Gauge	The size of the normal cache (when PooledByteBufAllocator)
reactor.netty.bytebuf.allocator.used.chunk.size	Gauge	The chunk size for an arena (when PooledByteBufAllocator)

The following example enables that integration:

```

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 ❶
            .config()

        .meterFilter(MeterFilter.maximumAllowableTags("reactor.netty.http.client", "URI",
100, MeterFilter.deny()));

        HttpClient client =
            HttpClient.create()
                .metrics(true, s -> {
                    if (s.startsWith("/stream/")) { ❷
                        return "/stream/{n}";
                    }
                    else if (s.startsWith("/bytes/")) {
                        return "/bytes/{n}";
                    }
                    return s;
                }); ❸

        client.get()
            .uri("http://httpbin.org/stream/2")
            .responseContent()
            .blockLast();

        client.get()
            .uri("http://httpbin.org/bytes/1024")
            .responseContent()
            .blockLast();
    }
}

```

- ❶ Applies upper limit for the meters with **URI** tag
- ❷ Templated URIs will be used as an URI tag value when possible
- ❸ 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:

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

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

public class Application {

    public static void main(String[] args) {
        HttpClientResponse response =
            HttpClient.create()
                .metrics(true, () -> new
CustomHttpClientMetricsRecorder()) ①
                .get()
                .uri("https://httpbin.org/stream/2")
                .response()
                .block();
    }
}
```

① Enables HTTP client metrics and provides [HttpClientMetricsRecorder](#) implementation.

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.

7.1. 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:

```
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()                ❶
                .bindNow(Duration.ofSeconds(30)); ❷

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

❶ Creates a `UdpServer` instance that is ready for configuring.

❷ 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.

7.1.1. Host and Port

In order to serve on a specific host and port, you can apply the following configuration to the `UDP` server:

```
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

7.2. 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**:

```

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()); ❶
                                }
                                else {
                                    return Mono.error(
                                        new Exception("Unexpected type of
the message: " + o));
                                }
                            }
                        )
                    )
                .bindNow(Duration.ofSeconds(30));

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

```

❶ Sends a **hello** string to the remote peer

7.3. 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:

```

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()); ❶
                                }
                                else {
                                    return Mono.error(new
Exception("Unexpected type of the message: " + o));
                                }
                            })))
                .bindNow(Duration.ofSeconds(30));

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

```

❶ Receives data from the remote peer

7.4. Lifecycle Callbacks

The following lifecycle callbacks are provided to let you extend the **UDP** server:

- **doOnBind**: Invoked when the server channel is about to bind.
- **doOnBound**: Invoked when the server channel is bound.
- **doOnUnbound**: Invoked when the server channel is unbound.
- **doOnLifecycle**: Sets up all lifecycle callbacks.

The following example uses the **doOnBound** method:


```
import io.netty.handler.codec.LineBasedFrameDecoder;
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.addHandler(new
LineBasedFrameDecoder(8192))) ①
                .bindNow(Duration.ofSeconds(30));

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

① **Netty** pipeline is extended with **LineBasedFrameDecoder** when the server channel is bound.

7.5. Connection Configuration

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

- [Channel Options](#)
- [Wire Logger](#)
- [Event Loop Group](#)

7.5.1. Channel Options

By default, the **UDP** server is configured with the following options:

```
./../main/java/reactor/netty/udp/UdpServer.java
```

```
static final Bootstrap DEFAULT_BOOTSTRAP =
    new Bootstrap().option(ChannelOption.AUTO_READ, false)
        .localAddress(NetUtil.LOCALHOST, DEFAULT_PORT);
```

If you need additional options or need to change the current options, you can apply the following configuration:

```
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:

- [ChannelOption](#)
- [Socket Options](#)

7.5.2. Wire Logger

Reactor Netty provides wire logging for when the traffic between the peers has 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:

```
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

7.5.3. Event Loop Group

By default, the UDP server uses “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 [LoopResource#create](#) methods.

The default configuration for the “Event Loop Group” is the following:

../../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)
 */
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:

```

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();
    }
}

```

7.6. 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
reactor.netty.udp.server.data.sent	DistributionSummary	Amount of the data sent, in bytes
reactor.netty.udp.server.errors	Counter	Number of errors that occurred

These additional metrics are also available:

ByteBufAllocator metrics

metric name	type	description
reactor.netty.bytebuf.allocator.used.heap.memory	Gauge	The number of the bytes of the heap memory
reactor.netty.bytebuf.allocator.used.direct.memory	Gauge	The number of the bytes of the direct memory
reactor.netty.bytebuf.allocator.used.heap.arenas	Gauge	The number of heap arenas (when PooledByteBufAllocator)

metric name	type	description
reactor.netty.bytebuf allocator. used.direct.arenas	Gauge	The number of direct arenas (when <code>PooledByteBufAllocator</code>)
reactor.netty.bytebuf allocator. used.threadlocal.caches	Gauge	The number of thread local caches (when <code>PooledByteBufAllocator</code>)
reactor.netty.bytebuf allocator. used.tiny.cache.size	Gauge	The size of the tiny cache (when <code>PooledByteBufAllocator</code>)
reactor.netty.bytebuf allocator. used.small.cache.size	Gauge	The size of the small cache (when <code>PooledByteBufAllocator</code>)
reactor.netty.bytebuf allocator. used.normal.cache.size	Gauge	The size of the normal cache (when <code>PooledByteBufAllocator</code>)
reactor.netty.bytebuf allocator. used.chunk.size	Gauge	The chunk size for an arena (when <code>PooledByteBufAllocator</code>)

The following example enables that integration:

```
import reactor.netty.Connection;
import reactor.netty.udp.UdpServer;

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:

```
import reactor.netty.Connection;
import reactor.netty.udp.UdpServer;

public class Application {

    public static void main(String[] args) {
        Connection server =
            UdpServer.create()
                .metrics(true, () -> new CustomChannelMetricsRecorder())
                .bindNow(Duration.ofSeconds(30));

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

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

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.

8.1. 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:

```
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.

8.1.1. Host and Port

To connect to a specific `host` and `port`, you can apply the following configuration to the `UDP` client:


```

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

8.2. 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**:

```

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"))) ❶
                .connectNow(Duration.ofSeconds(30));

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

```

❶ Sends **hello** string to the remote peer.

8.3. 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:

```

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

8.4. Lifecycle Callbacks

The following lifecycle callbacks are provided to let you extend the **UDP** client:

- **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.
- **doOnLifecycle**: Sets up all lifecycle callbacks.

The following example uses the **doOnConnected** method:

```
import io.netty.handler.codec.LineBasedFrameDecoder;
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.addHandler(new
LineBasedFrameDecoder(8192))) ①
                .connectNow(Duration.ofSeconds(30));

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

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

8.5. Connection Configuration

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

- [Channel Options](#)
- [Wire Logger](#)
- [Event Loop Group](#)

8.5.1. Channel Options

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

```
./../main/java/reactor/netty/udp/UdpClient.java
```

```
static final Bootstrap DEFAULT_BOOTSTRAP =
    new Bootstrap().option(ChannelOption.AUTO_READ, false)
        .remoteAddress(NetUtil.LOCALHOST, DEFAULT_PORT);
```

If you need additional options or need to change the current options, you can apply the following configuration:

```
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:

- [ChannelOption](#)
- [Socket Options](#)

8.5.2. Wire Logger

Reactor Netty provides wire logging for when the traffic between the peers has 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:

```

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

8.5.3. Event Loop Group

By default, the UDP client uses “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”:

../../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)
 */
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 the these settings, you can apply the following configuration:

```

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();
    }
}

```

8.6. 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
reactor.netty.udp.client.data.sent	DistributionSummary	Amount of the data sent, in bytes
reactor.netty.udp.client.errors	Counter	Number of errors that occurred
reactor.netty.udp.client.connect.time	Timer	Time spent for connecting to the remote address
reactor.netty.udp.client.address.resolver	Timer	Time spent for resolving the address

These additional metrics are also available:

ByteBufAllocator metrics

metric name	type	description
reactor.netty.bytebuf allocator. used.heap.memory	Gauge	The number of the bytes of the heap memory
reactor.netty.bytebuf allocator. used.direct.memory	Gauge	The number of the bytes of the direct memory
reactor.netty.bytebuf allocator. used.heap.arenas	Gauge	The number of heap arenas (when <code>PooledByteBufAllocator</code>)
reactor.netty.bytebuf allocator. used.direct.arenas	Gauge	The number of direct arenas (when <code>PooledByteBufAllocator</code>)
reactor.netty.bytebuf allocator. used.threadlocal.caches	Gauge	The number of thread local caches (when <code>PooledByteBufAllocator</code>)
reactor.netty.bytebuf allocator. used.tiny.cache.size	Gauge	The size of the tiny cache (when <code>PooledByteBufAllocator</code>)
reactor.netty.bytebuf allocator. used.small.cache.size	Gauge	The size of the small cache (when <code>PooledByteBufAllocator</code>)
reactor.netty.bytebuf allocator. used.normal.cache.size	Gauge	The size of the normal cache (when <code>PooledByteBufAllocator</code>)
reactor.netty.bytebuf allocator. used.chunk.size	Gauge	The chunk size for an arena (when <code>PooledByteBufAllocator</code>)

The following example enables that integration:

```
import reactor.netty.Connection;
import reactor.netty.udp.UdpClient;

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:

```
import reactor.netty.Connection;
import reactor.netty.udp.UdpClient;

public class Application {

    public static void main(String[] args) {
        Connection connection =
            UdpClient.create()
                .host("example.com")
                .port(80)
                .metrics(true, () -> new CustomChannelMetricsRecorder())
                .connectNow(Duration.ofSeconds(30));

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

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